



# 数据结构与算法

# Data Structure and Algorithm

## 第6章 树



# 目 录

6.1 树的基本概念

6.2 二叉树

6.3 遍历二叉树

6.4 线索二叉树

6.5 树和森林

6.6 哈夫曼树

## 6.1 树的基本概念

### 6.1.1 树的定义

#### 1. 树的定义

树是由 $n$  ( $n \geq 0$ ) 个结点组成的有限集合。若 $n=0$ ，称为空树；若 $n>0$ ，则

:

(1) 有一个特定的称为根 (root) 的结点。它只有直接后继，但没有直接前驱；

(2) 除根结点以外的其它结点可以划分为 $m$  ( $m \geq 0$ ) 个互不相交的有限集合  $T_0, T_1, \dots, T_{m-1}$ ，每个集合  $T_i$  ( $i=0,1,\dots,m-1$ ) 又是一棵树，称为根的子树，每棵子树的根结点有且仅有一个直接前驱，但可以有0个或多个直接后继。

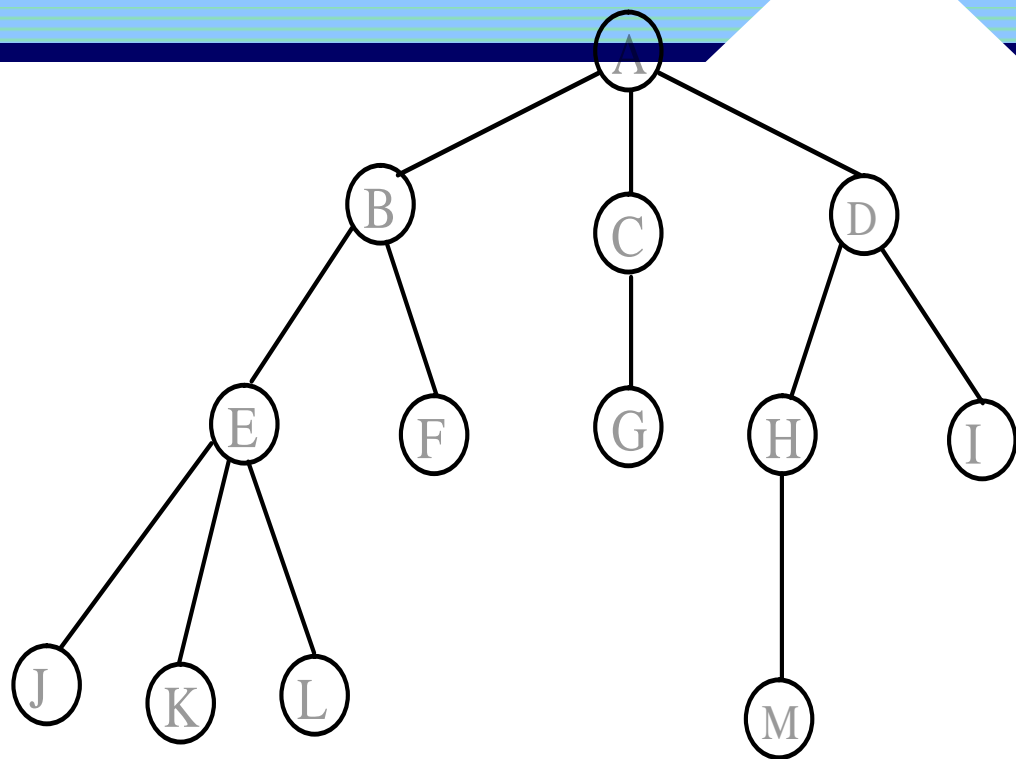
由此可知，树的定义是一个递归的定义，即树的定义中又用到了树的概念。

树的结构参见图6-1。



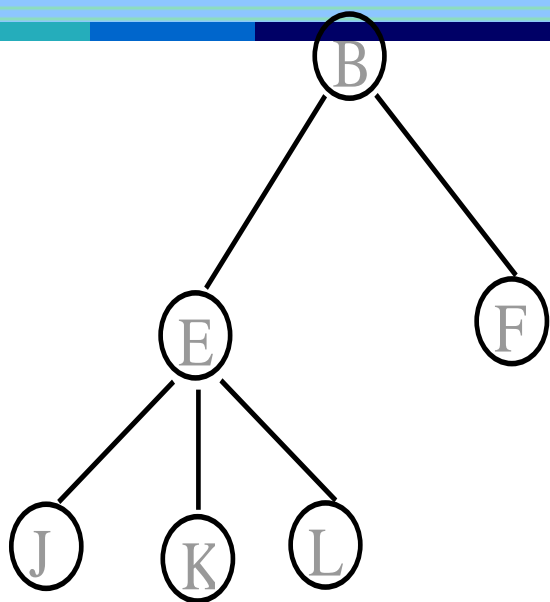
$\emptyset$

A

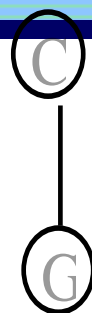


(a) 空树 (b) 仅含有根结点的树 (c) 含有多个结点的树

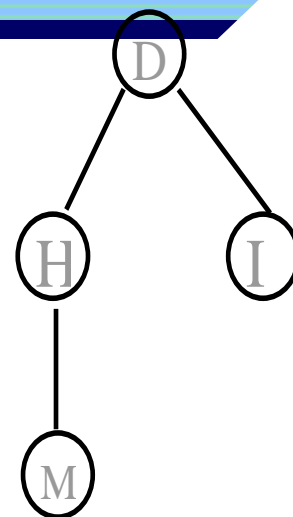
图 6-1 树的示意图



(a)  $T_0$  子树



(b)  $T_1$  子树



(c)  $T_2$  子树

图 6-2 图 6-1(C)的树的三个子树

## 2. 树的逻辑结构描述

一棵树的逻辑结构可以用二元组描述为:

$$\text{tree} = (K, R)$$

$$K = \{k_i \mid 1 \leq i \leq n; n \geq 0, k_i \in \text{elemtype}\}$$

$$R = \{r\}$$

其中,  $n$  为树中结点个数, 若  $n=0$ , 则为一棵空树,  $n > 0$  时称为一棵非空树, 而关系  $r$  应满足下列条件:

- (1) 有且仅有一个结点没有前驱, 称该结点为树根;
- (2) 除根结点以外, 其余每个结点有且仅有一个直接前驱;
- (3) 树中每个结点可以有多个直接后继(孩子结点)。



例如，对图6-1(c)的树结构,可以二元组表示为:

$$K=\{A, B, C, D, E, F, G, H, I, J, K, L, M\}$$
$$R=\{r\}$$
$$r=\{(A,B), (A,C), (A,D), (B,E), (B,F), (C,G), (D,H), (D,I), (E,J), (E,K), (E,L), (H,M)\}$$

### 3. 树的基本运算

树的基本运算可以定义如下几种:

**(1) inittree(&T)**

初始化树T。

**(2) root(T)**

求树T的根结点。



**(3) parent(T,x)**

求树T中，值为x的结点的双亲。

**(4) child(T,x,i)**

求树T中，值为x的结点的第i个孩子。

**(5) addchild(y,i,x)**

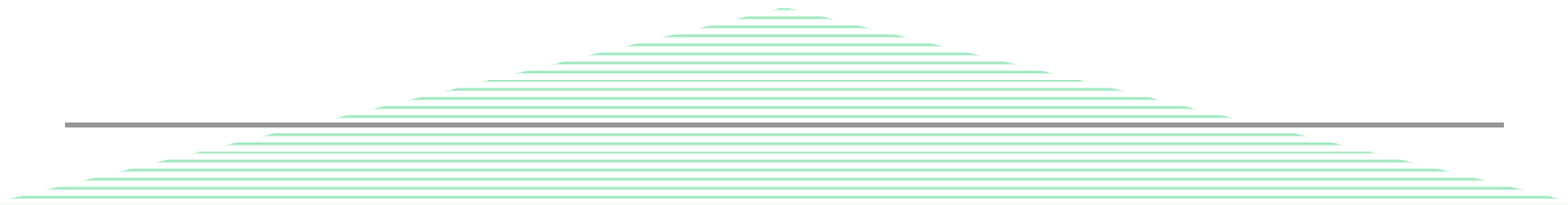
把值为x的结点作为值为y的结点的第i个孩子插入到树中。

**(6) delchild(x,i)**

删除值为x的结点的第i个孩子。

**(7) traverse(T)**

遍历或访问树T。







## 6.1.2 基本术语

### 1. 结点

指树中的一个数据元素以及若干指向子树的分枝，一般用一个字母表示。

### 2. 孩子结点

若结点X有子树，则子树的根结点为X的孩子结点，也称为孩子，儿子，子女等。如图6-1（c）中A的孩子为B，C，D。

### 3. 双亲结点

若结点X有子女Y，则X为Y的双亲结点。





#### 4.祖先结点

从根结点到该结点所经过分枝上的所有结点为该结点的祖先，如图6-1（c）中M的祖先有A，D，H。

#### 5.子孙结点

以某结点为根的子树中的任一结点都称为该结点的子孙。

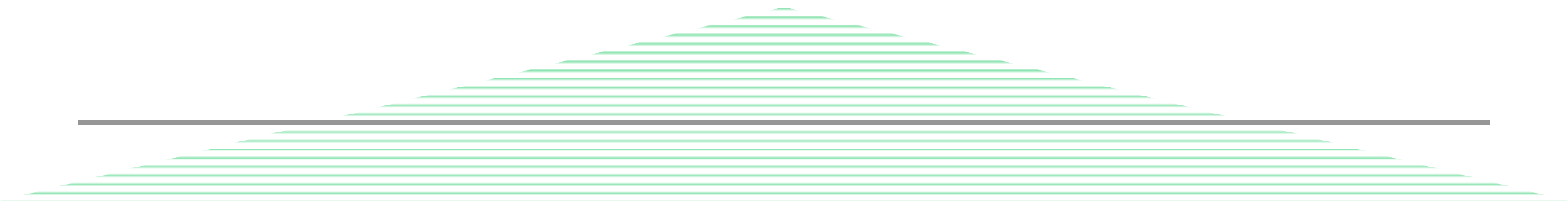
某一结点的子女及子女的子女都为该结点子孙。

#### 6.兄弟结点

具有同一个双亲的结点，称为兄弟结点。

#### 7.堂兄弟结点

双亲在同一层的结点，称为堂兄弟结点。





## 8.树叶（叶子）

没有子结点的结点，称为叶子结点或树叶，也叫终端结点。

## 9.非终端结点

除叶子结点外的所有结点，为分枝结点，也叫非终端结点。

10.结点的度：一个结点包含子树的数目，称为该结点的度。

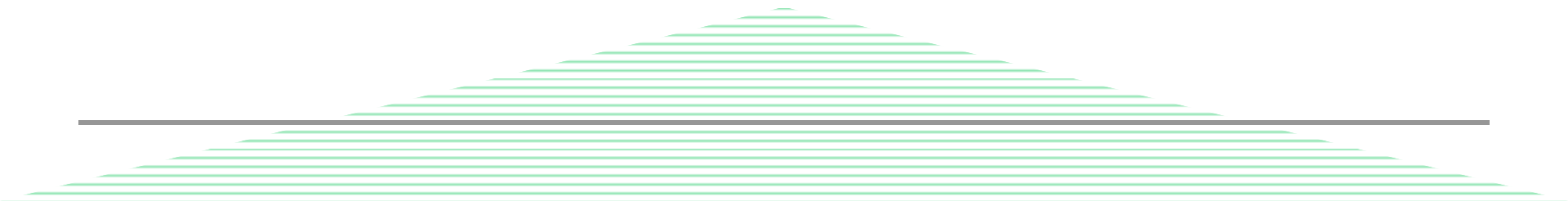
树的度：树中结点度的最大值称为树的度。

## 11.层数

树中结点的关系，一代为一层，根结点的层数为1。

12.结点的高度：由本结点到终端结点的最大层次。

树的高度：树中结点所处的最大层数，如空树的高度为0，只有一个根结点的树高度为1。





### 13. 边

从一个结点大批它的后继结点的直线称为边（edge）。

### 14. 路径

一串连续的边组成一个路径。

### 15. 有序树

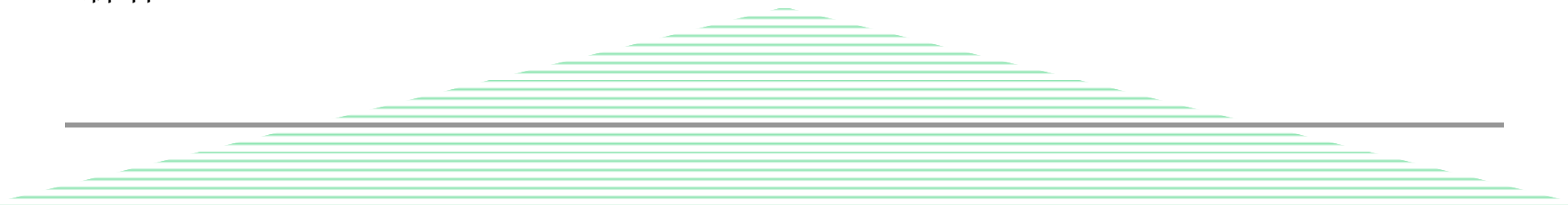
若一棵树中所有子树从左到右的排序是有顺序的，不能颠倒次序。称该树为有序树。

### 16. 无序树

若一棵树中所有子树的次序无关紧要，则称为无序树。

### 17. 森林（树林）

若干棵互不相交的树组成的集合为森林。一棵树可以看成是一个特殊的森林。



### 6.1.3 树的表示

#### 1. 树形结构表示法

具体参 见图6-1 。

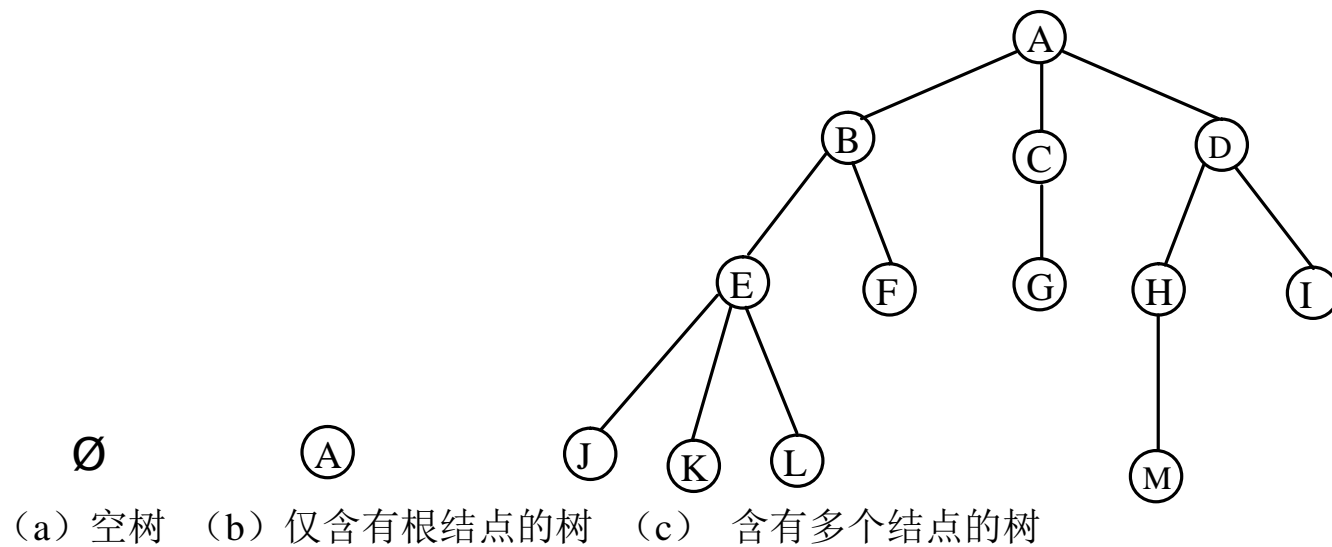


图 6-1 树的示意图

## 2. 凹入法表示法

具体参见图6-3。

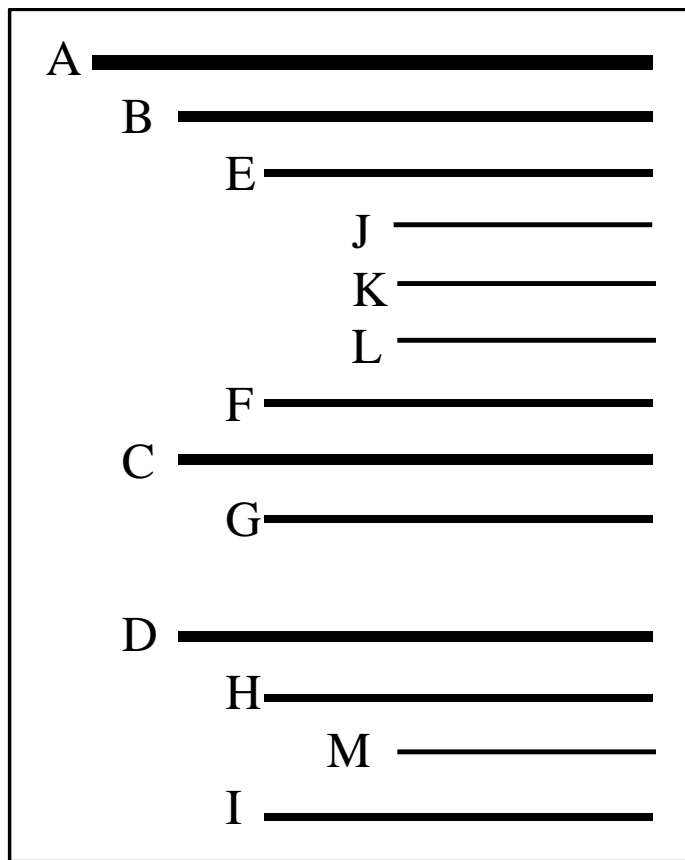


图 6-3 图 6-1(c)的树的凹入法表示

### 3. 嵌套集合表示法(范氏图法)

具体参 见图6-4。

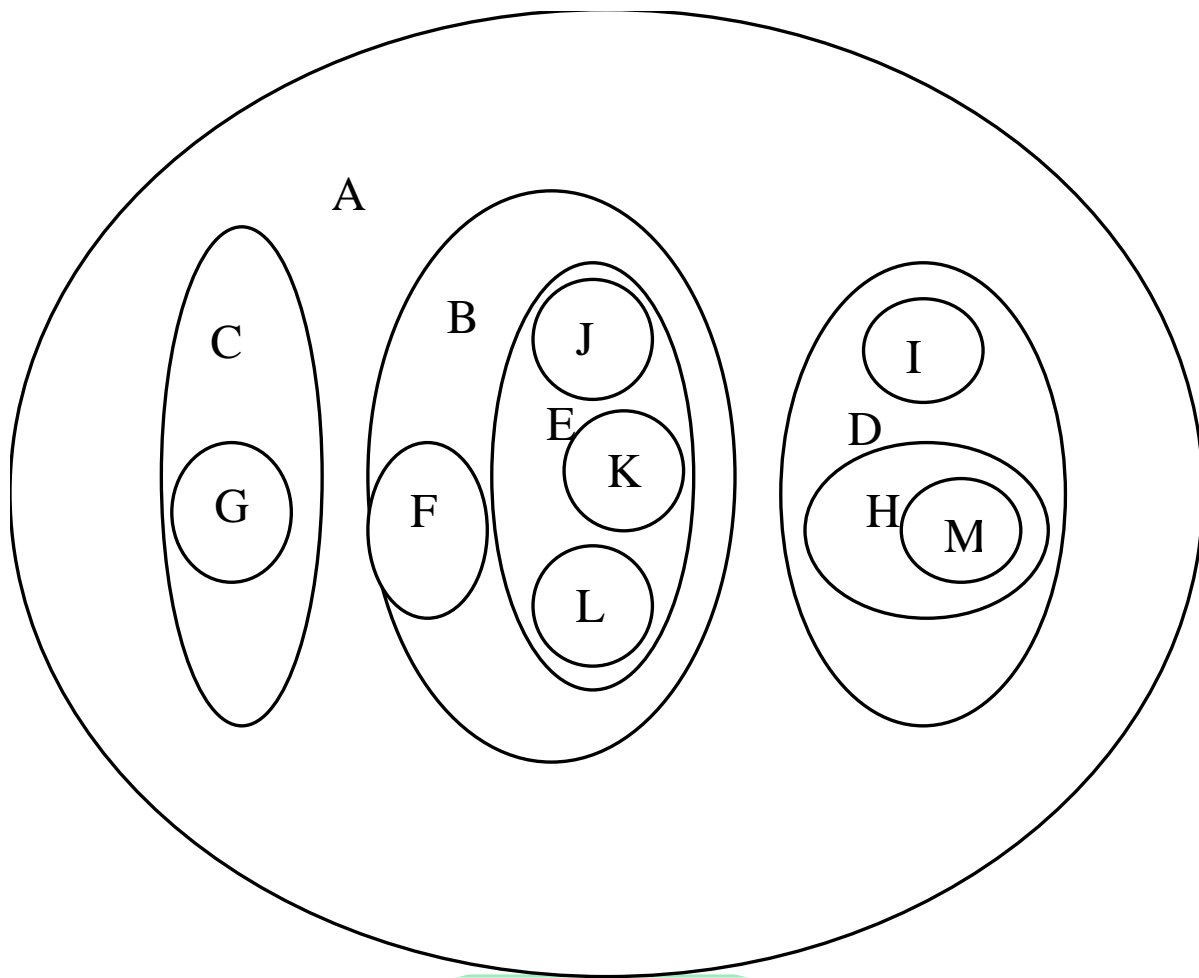


图 6-4

图 6-1(c)的树的集合表示



#### 4. 广义表表示法（嵌套括弧法）

对图6-1（c）的树结构，广义表表示法可表示为：

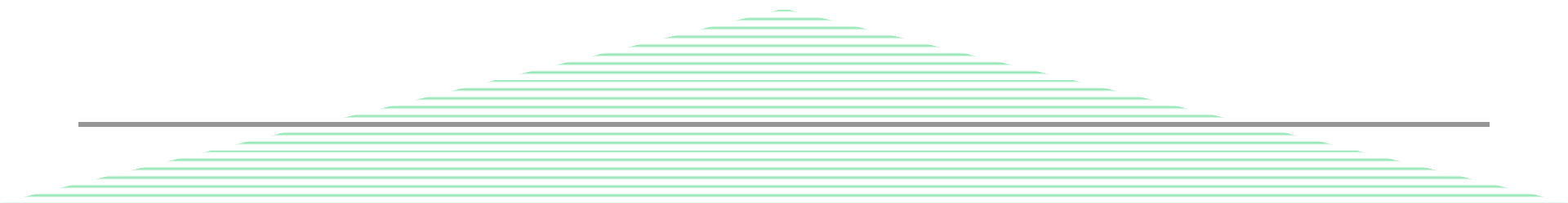
$(A (B (E (J, K, L), F), C (G), D (H (M), I)))$





自做作业:

6.1





## 6.2 二叉树 (BiTree, binary tree)

### 6.2.1 二叉树的定义

#### 1. 二叉树的定义

和树结构定义类似，二叉树的定义也可以递归形式给出：

二叉树是 $n$  ( $n \geq 0$ ) 个结点的有限集，它或者是空集 ( $n=0$ )，或者由一个根结点及两棵不相交的左子树和右子树组成。

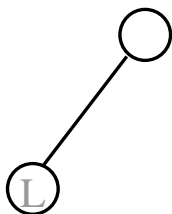


$\emptyset$

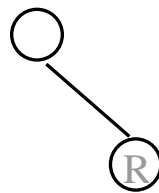
(a)



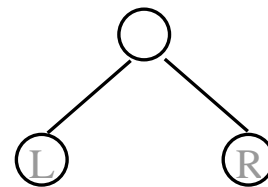
(b)



(c)



(d)



(e)

图 6-5 二叉树的五种不同的形态



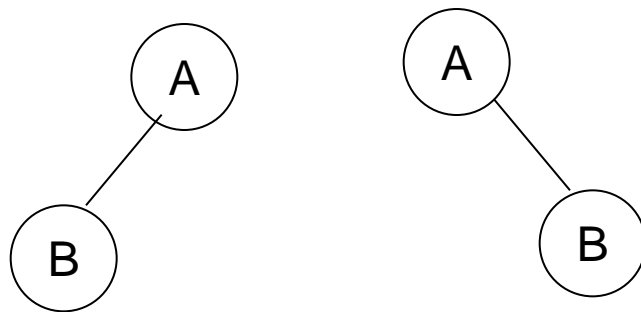
## 二叉树的特点：

- a. 是每个结点最多有两个孩子，或者说，在二叉树中，不存在度大于2的结点
- b. 二叉树是有序树（树为无序树），其子树的顺序不能颠倒.



## 树和二叉树的差异：

- a. 二叉树有次序关系，而树没有
- b. 树中的每一个结点的度 $\geq 0$ ，而二叉树中每一个结点的度为0，1或2
- c. 二叉树并不是树的一个特例



区分：度为2的树 / 二叉树





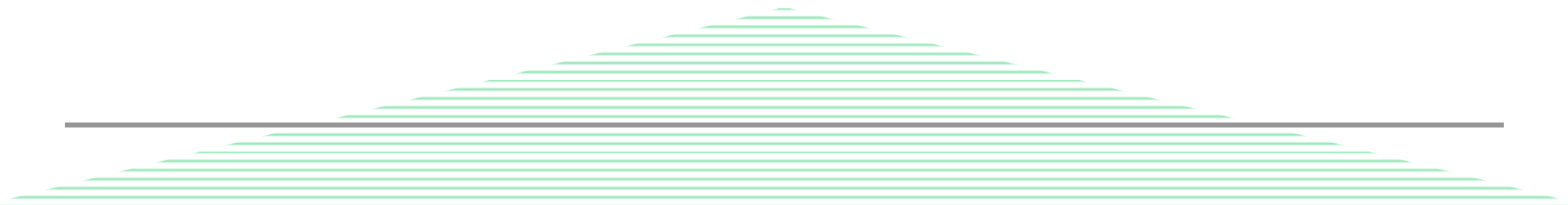
## 6.2.2 二叉树的性质


**性质1** 若二叉树的层数从1开始，则二叉树的第k层结点数，最多为 $2^{k-1}$ 个（ $k>0$ ）。

可以用数学归纳法证明之。

**性质2** 深度（高度）为k的二叉树最大结点数为 $2^k-1$ （ $k>0$ ）。

证明： 深度为k的二叉树，若要求结点数最多，则必须每一层的结点数都为最多，由性质1可知，最大结点数应为每一层最大结点数之和。既为 $2^0+2^1+\dots+2^{k-1}=2^k-1$ 。

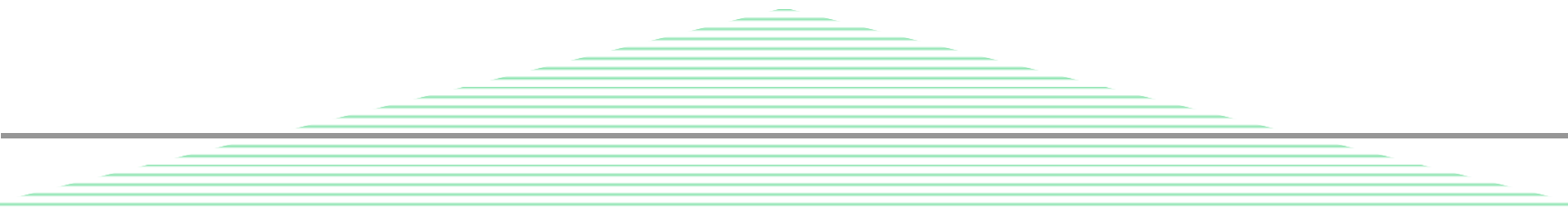




**性质3** 对任意一棵二叉树，如果叶子结点个数为 $n_0$ ，度为2的结点个数为 $n_2$ ，则有 $n_0=n_2+1$ 。

证明：设二叉树中度为1的结点个数为 $n_1$ ，根据二叉树的定义可知，该二叉树的结点数 $n=n_0+n_1+n_2$ 。又因为在二叉树中，度为0的结点没有孩子，度为1的结点有1个孩子，度为2的结点有2个孩子，故该二叉树的孩子结点数为 $n_0*0+n_1*1+n_2*2$ ，而一棵二叉树中，除根结点外所有都为孩子结点，故该二叉树的结点数应为孩子结点数加1即： $n=n_0*0+n_1*1+n_2*2+1$ 因此,有  $n=n_0+n_1+n_2=n_0*0+n_2*1+n_2*2+1$ ，最后得到 $n_0=n_2+1$ 。

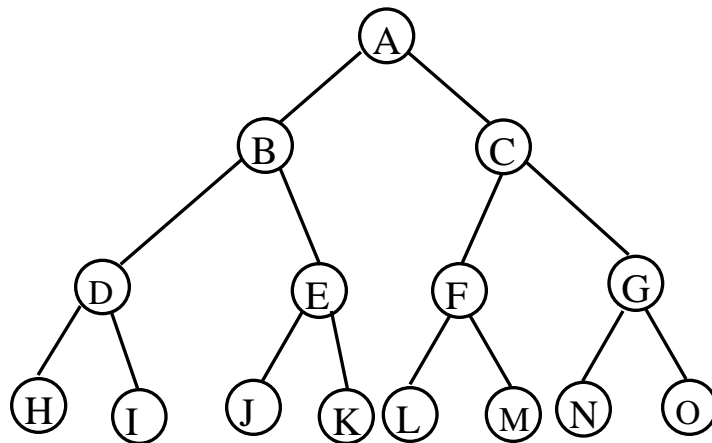
为继续给出二叉树的其它性质，先定义两种特殊的二叉树。





**满二叉树：**深度为 $k$ 具有 $2^k-1$ 个结点的二叉树

即必须是二叉树的每一层上的结点数都达到最大，否则就不是满二叉树。



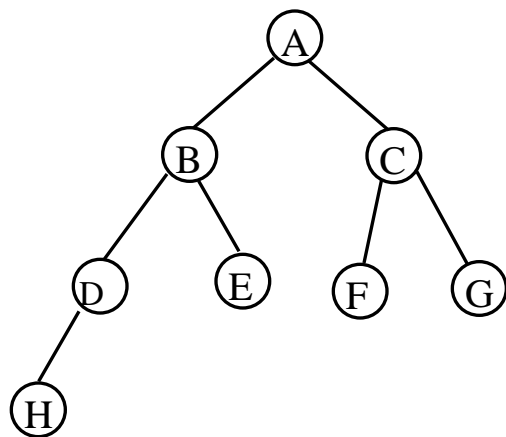
( a ) 满二叉树





**完全二叉树：** 如果一棵具有 $n$ 个结点的深度为 $k$ 的二叉树，它的每一个结点都与深度为 $k$ 的满二叉树中编号为 $1 \sim n$ 的结点一一对应，则称这棵二叉树为完全二叉树。

从完全二叉树定义可知，结点的排列顺序遵循从上到下、从左到右的规律。所谓从上到下，表示本层结点数达到最大后，才能放入下一层。从左到右，表示同一层结点必须按从左到右排列，若左边空一个位置时不能将结点放入右边。



(b) 完全二叉树



两者关系:

从满二叉树及完全二叉树定义还可以知道，满二叉树一定是一棵完全二叉树，反之完全二叉树不一定是一棵满二叉树。满二叉树的叶子结点全部在最底层，而完全二叉树的叶子结点可以分布在最下面两层。



**性质4** 具有 $n$ 个结点的完全二叉树高度为 $\lfloor \log_2(n) \rfloor + 1$  或  $\lceil \log_2(n+1) \rceil$ 。

(注意 $\lfloor x \rfloor$ 表示取不大于 $x$ 的最大整数，也叫做对 $x$ 下取整， $\lceil x \rceil$ 表示取不小于 $x$ 的最小整数，也叫做对 $x$ 上取整。)



**性质5** 如果将一棵有 $n$ 个结点的完全二叉树从上到下，从左到右对结点编号 $1, 2, \dots, n$ ，然后按此编号将该二叉树中各结点顺序地存放于一个一维数组中，并简称编号为 $j$ 的结点为 $j(1 \leq j \leq n)$ ，则有如下结论成立：

- (1) 若  $j=1$ ，则结点 $j$ 为根结点，无双亲，否则 $j$ 的双亲为  $\lfloor j/2 \rfloor$ ；
- (2) 若  $2j \leq n$ ，则结点 $j$ 的左子女为  $2j$ ；否则无左子女，即满足  $2j > n$  的结点为叶子结点；
- (3) 若  $2j+1 \leq n$ ，则结点 $j$ 的右子女为  $2j+1$ ；否则无右子女；
- (4) 若结点 $j$ 序号为奇数且不等于1，则它的左兄弟为  $j-1$ ；
- (5) 若结点 $j$ 序号为偶数且不等于 $n$ ，它的右兄弟为  $j+1$ ；
- (6) 结点 $j$ 所在层数(层次)为  $\lfloor \log_2 j \rfloor + 1$ ；

## 6.2.3 二叉树的存贮结构

### 1. 顺序存贮结构

将一棵二叉树按完全二叉树顺序存放到一个一维数组中，若该二叉树为非完全二叉树，则必须将相应位置空出来，使存放的结果符合完全二叉树形状。如图6-7给出了顺序存贮形式。

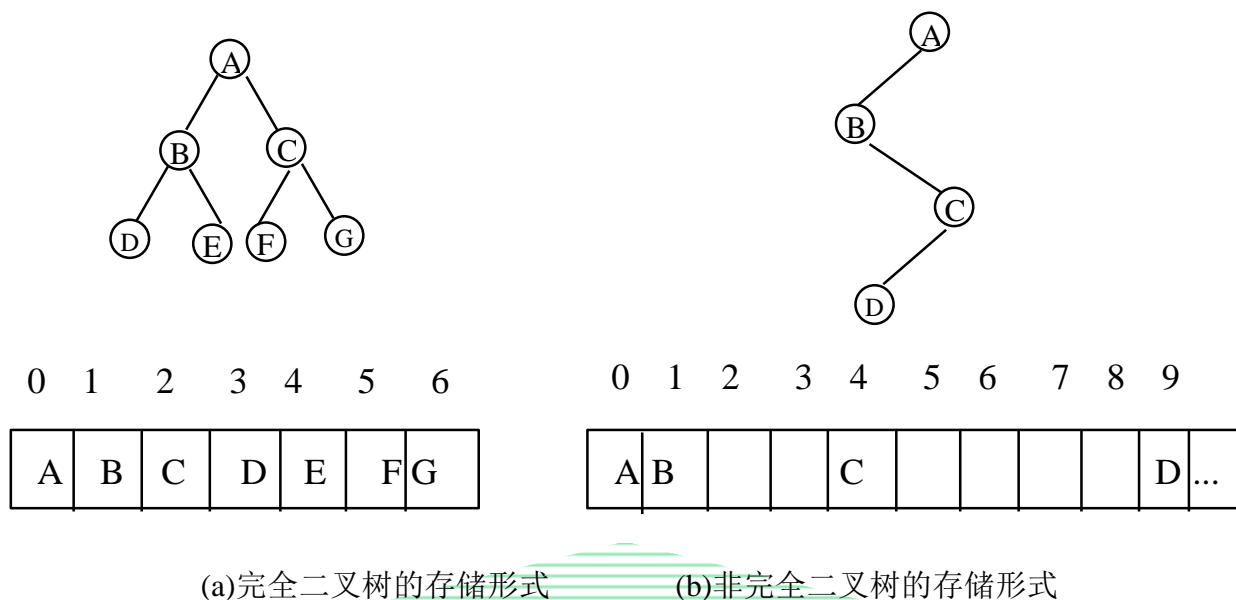


图 6-7 二叉树的顺序存储形式



对于一棵二叉树,若采用顺序存贮时,当它为完全二叉树时,比较方便,若为非完全二叉树,将会浪费大量存贮单元。最坏的非完全二叉树是全部只有右分支,设高度为 $K$ ,则需占用 $2^K-1$ 个存贮单元,而实际只有 $k$ 个元素,实际只需 $k$ 个存储单元。因此,对于非完全二叉树,宜采用下面的链式存储结构。

## 2. 二叉链表存贮结构

### (1) 二叉链表表示

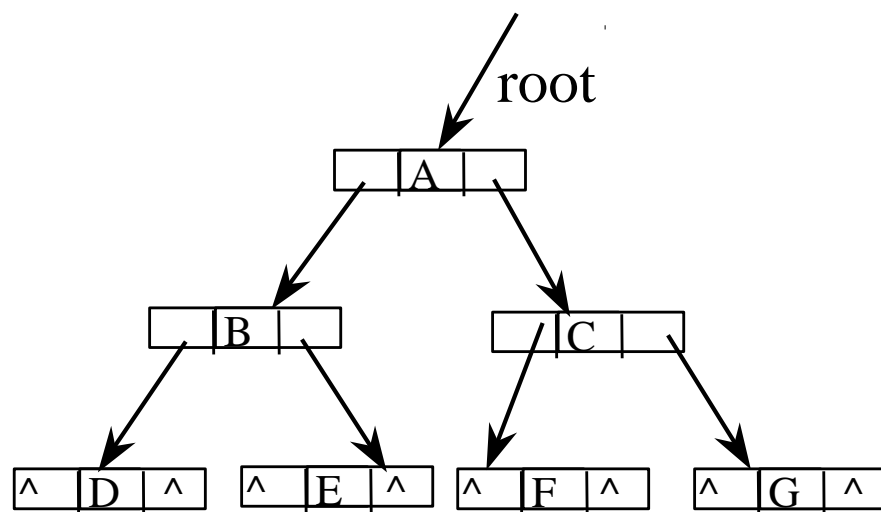
将一个结点分成三部分,一部分存放结点本身信息,另外两部分为指针,分别存放左、右孩子的地址。

二叉链表中一个结点可描述为:

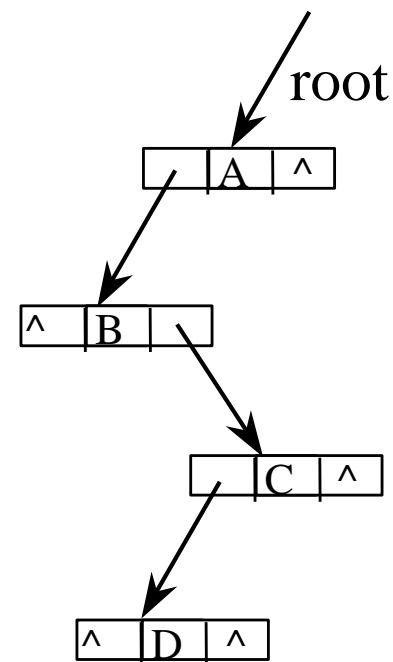




对于图6-7所示二叉树,用二叉链表形式描述见图6-8。



(a)完全二叉树的链表



(b)非完全二叉树的二叉链表

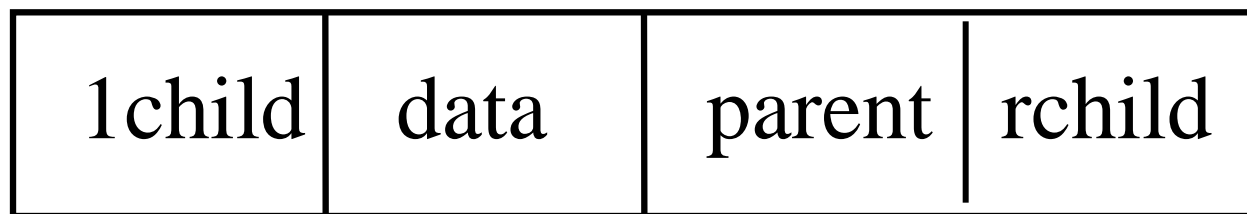
图 6-8 二叉树的二叉链表表示法



对于一棵二叉树,若采用二叉链表存贮时,当二叉树为非完全二叉树时,比较方便,若为完全二叉树时,将会占用较多存贮单元(存放地址的指针)。若一棵二叉树有 $n$ 个结点,采用二叉链表作存贮结构时,共有 $2n$ 个指针域,其中只有 $n-1$ 个指针指向左右孩子,其余 $n+1$ 个指针为空,没有发挥作用,被白白浪费掉了,(当然后面介绍的线索可利用它)。

## (2) 三叉链表表示

将一个结点分成四部分,一部分存放结点本身信息,另外三部分为指针,分别存放左、右孩子的地址以及双亲的地址。







## 6.3 遍历二叉树

### 定义：

所谓遍历二叉树，就是遵从某种次序，访问二叉树中的所有结点，使得每个结点仅被访问一次。

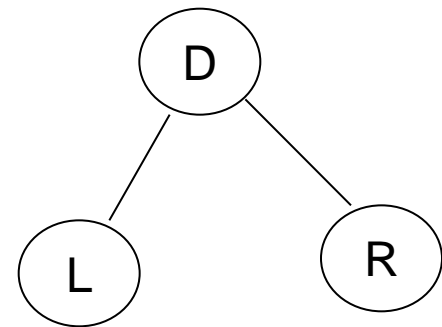
注意：这里提到的“访问”是指对结点施行某种操作，操作可以是输出结点信息，修改结点的数据值等，但要求这种访问不破坏它原来的数据结构。在本书中，我们规定访问是输出结点信息data，且以二叉链表作为二叉树的存贮结构。



## 遍历方式:

令L,R,D分别代表二叉树的左子树(L)、右子树(R)、根结点(D), 则遍历二叉树有6种规则: DLR、DRL、LDR、LRD、RDL、RLD。若规定二叉树中必须先左后右(左右顺序不能颠倒), 则只有DLR、LDR、LRD三种遍历规则:

- DLR称为前根遍历(前序遍历、先序遍历、先根遍历)
- LDR称为中根遍历(中序遍历)
- LRD称为后根遍历(后序遍历)
- 层次遍历



例如，可以利用上面介绍的遍历算法，写出如图6-11所示二叉树的三种遍历序列为：

先序遍历序列：ABDGCEF H

中序遍历序列：BGDAECFH

后序遍历序列：GDBEHFCA

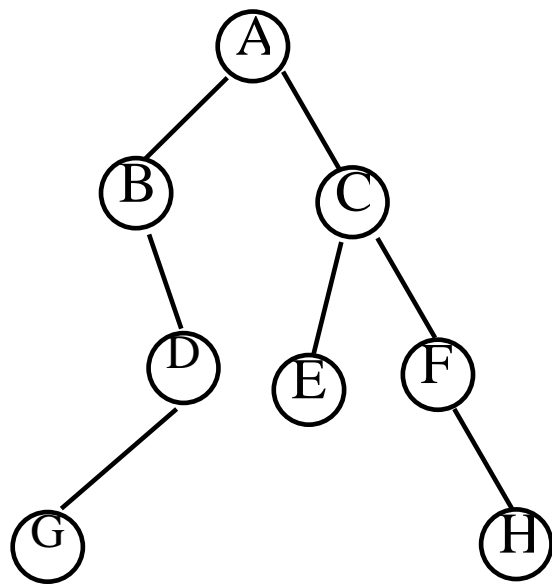


图 6-11 一棵二叉树



### 6.3.1前根遍历（DLR）

所谓前根遍历，就是根结点最先遍历，其次左子树，最后右子树。

前根遍历二叉树的递归遍历算法描述为：

若二叉树为空，则算法结束;否则

- (1) 输出根结点；**
- (2) 前根遍历左子树;**
- (3) 前根遍历右子树;**

算法如下:

```
Status PreOrderTraverse(Bitree T, Status(* Visit )(TElemaType e)){
```

//采用二叉链表存储结构, Visit是对数据元素操作的应用函数, 先序遍历二叉树T的

//递归算法, 对每个数据元素调用函数Visit,最简单的Visit函数是:

```
//          Status PrintElement(TElemType e){
```

```
//              printf(e);
```

```
//              return OK;
```

```
//          }
```

```
If(T){
```

```
    if(Visit(T->data))
```

```
        if(PreOrderTraverse(T->lchild,Visit))
```

```
            if(PreOrderTraverse(T->rchild,Visit))    return OK;
```

```
        return ERROR;
```

```
    }
```

```
    }else return OK;
```

```
}//PreOrderTraverse
```



### 6.3.2中根遍历

所谓中根遍历，就是根在中间，先左子树，然后根结点，最后右子树。

中根遍历二叉树的递归遍历算法描述为：

若二叉树为空，则算法结束；否则

**(1)中根遍历左子树；**

**(2)输出根结点；**

**(3)中根遍历右子树。**



算法如下:

```
Status InOrderTraverse( Bitree T, Status(* Visit )(TElemType e)){  
    //采用二叉链表存储结构，中序遍历二叉树T的递归算法  
    if(T){  
        if( InOrderTraverse (T->lchild, Visit)))  
            if( Visit (T->data) )  
                if( InOrderTraverse(T->rchild,Visit))    return OK;  
        return ERROR;  
    }else return OK;  
} //InOrderTraverse
```



### 6.3.3 后根遍历

所谓后根遍历，就是根在最后，即先左子树，然后右子树，最后根结点。

后根遍历二叉树的递归遍历算法描述为：

若二叉树为空，则算法结束;否则

**(1) 后根遍历左子树:**

**(2) 后根遍历右子树;**

**(3) 访问根结点。**





算法如下:

```
Status PostOrderTraverse( Bitree T, Status(* Visit )(TElemType e)){  
    //采用二叉链表存储结构, 后序遍历二叉树T的递归算法  
    if(T){  
        if( PostOrderTraverse (T->lchild, Visit)))  
            if(PostOrderTraverse (T->rchild, Visit))  
                if(Visit (T->data))    return OK;  
        return ERROR;  
    }else return OK;  
}//InOrderTraverse
```



在编译原理中，有用二叉树来表示一个算术表达式的情形。在一棵二叉树中，若用操作数代表树叶，运算符代表非叶子结点，则这样的树可以代表一个算术表达式。若按前序、中序、后序对该二叉树进行遍历，则得到的遍历序列分别称为前缀表达式(或称波兰式)、中缀表达式、后缀表达式(递波兰式)。具体参见图6-12。

图6-12所对应的前缀表达式： $-*abc$ 。

图6-12所对应的中缀表达式： $a*b-c$ 。

图6-12所对应的后缀表达式： $ab*c-$ 。

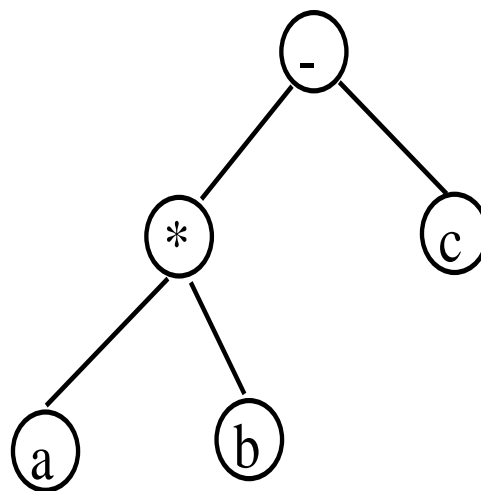


图 6-12 表达式  $a*b-c$  代表的二叉树



## 实验二：二叉树的应用-----表达式处理

- 实验要求： 以递归方式建立表达式的二叉树状结构，再分别输出前序、中序及后序遍历结果，并计算表达式的结果。
- 实验时间：

周五67节 10-414\10-413\409



## 课后思考：

阅读书本**P130**算法**6.2**和算法**6.3**，并用具体二叉树进行示范，找出两算法的区别之处，最后从时间和空间复杂度上进行比较。

- “遍历”是二叉树各种操作的基础，可以在遍历过程中对结点进行各种操作。
- 知遍历次序，可建立二叉树

先序和中序、中序和后序分别可以唯一的确定一棵二叉树。

---

例如1： 先序：ABDGHCEIFJ

中序：BGDHAEICJF

例如2： 后序：GDBEFCA

中序：BGDAECF

- 二叉树中的递归应用：
    - 判断二叉树是否相似
    - 求二叉树高度
    - 二叉树所有结点数
    - 统计二叉树度为0、1和2的结点数
-



作业:


6.14 6.42 6.43 6.47

思考: 6.4、 6.13

### 6.4.1 线索的概念（[见图](#)）

为了不浪费存储空间，我们利用原有的孩子指针为空时来存放直接前驱和后继，这样的指针称为“线索”，加线索的过程称为线索化，加了线索的二叉树，称为线索二叉树，对应的二叉链表称为线索二叉链表。

在线索二叉树中，由于有了线索，无需遍历二叉树就可以得到任一结点在某种遍历下的直接前驱和后继。但是，我们怎样来区分孩子指针域中存放的是左、右孩子信息还是直接前驱或直接后继信息呢？为此，在二叉链表结点中，还必须增加两个标志域ltag、rtag。



增加线索后的二叉链表结点结构可描述如下：

lchild	ltag	data	rtag	rchild
--------	------	------	------	--------

ltag和rtag定义如下：

$$ltag = \begin{cases} 0 & \text{lchild域指向结点的左孩子} \\ 1 & \text{lchild域指向结点在某种遍历下的直接前驱} \end{cases}$$

$$rtag = \begin{cases} 0 & \text{rchild域指向结点的右孩子} \\ 1 & \text{rchild域指向结点在某种遍历下的直接后继} \end{cases}$$





## 2. 线索的分类

根据遍历的不同要求，线索二叉树可以分为：

- (1) 前序前驱线索二叉树(只需画出前驱)
- (2) 前序后继线索二叉树(只需画出后继)
- (3) 前序线索二叉树(前驱和后继都要标出)
- (4) 中序前驱线索二叉树(只需画出前驱)
- (5) 中序后继线索二叉树(只需画出中序后继)
- (6) 中序线索二叉树(中序前驱和后继都要标出)
- (7) 后序前驱线索二叉树(只需画出后序前驱)
- (8) 后序后继线索二叉树(中需画出后序后驱)
- (9) 后序线索二叉树(后前驱和后继都要标出)



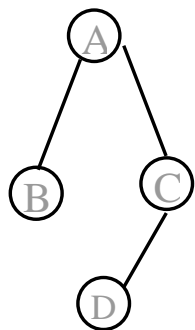
## 6.4.2 线索的描述

### 1. 结点数据类型描述

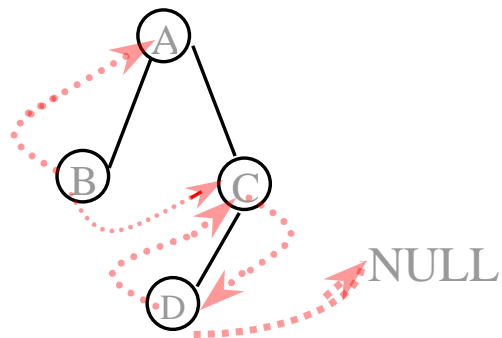
```
struct BiThrTree{  
    Elemtype data;  
    int ltag , rtag;    //左、右标志域  
    BiThrTree *lchild, *rchild;  
};
```

### 2. 线索的画法

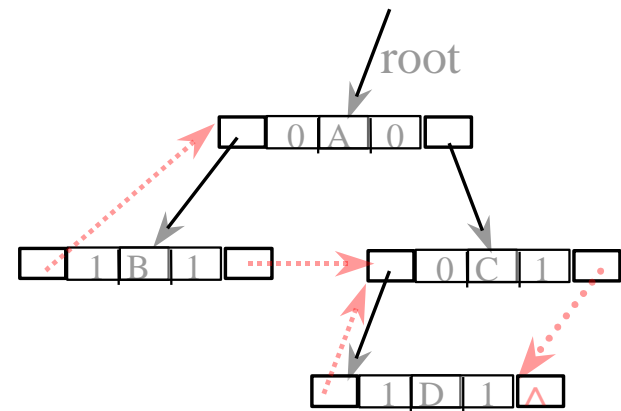
在二叉树或二叉链表中，若左孩子为空，则画出它的直接前驱，右孩子为空时，则画出它的直接后继，左右孩子不为空时，不需画前驱和后继。这样就得到了线索二叉树或线索二叉链表。



(a) 二叉树



(b) 前序线索二叉树

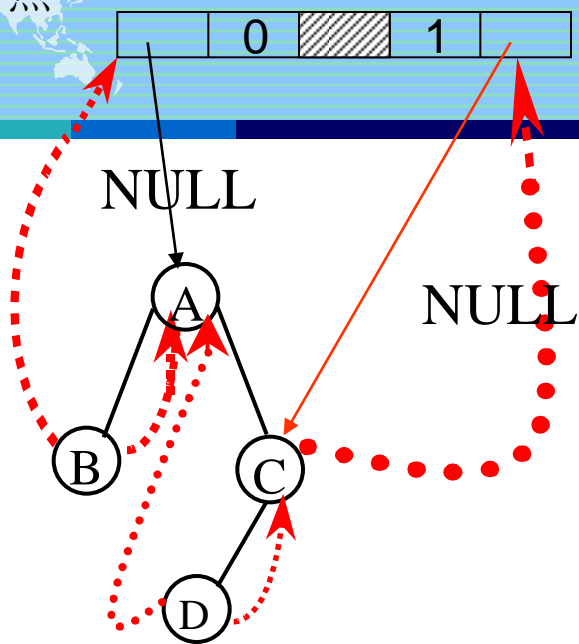


(c) 前序线索二叉链表

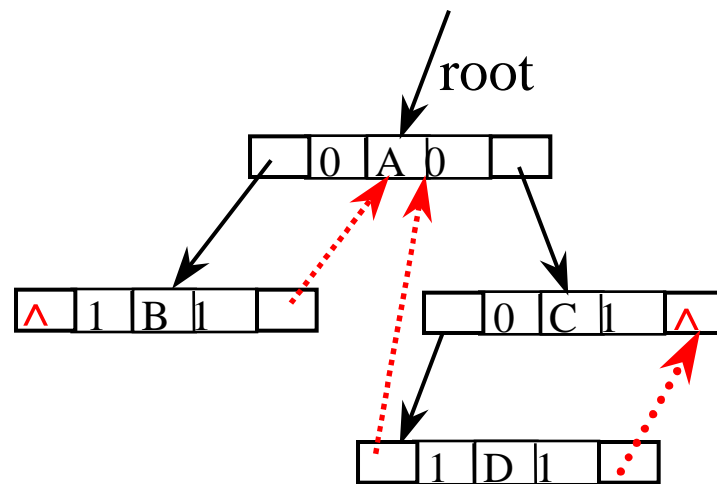
图 6-15 前序线索示意图

前序序列为：ABCD

头结点



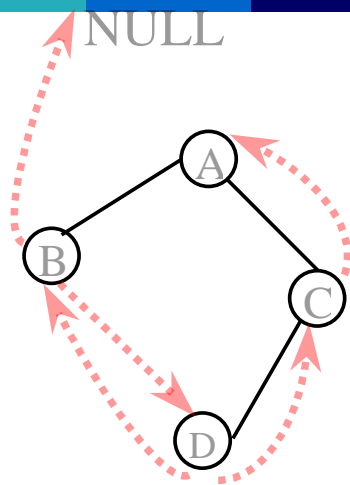
(a) 中序二叉树



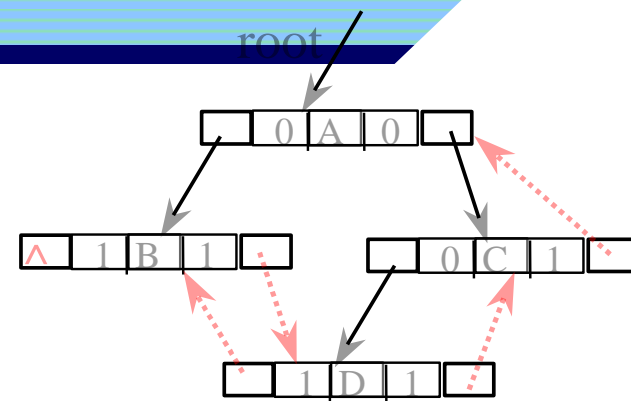
(b) 中序线索二叉链表

图 6-16 中序线索示意图

中序序列为: BADC



(a)后序线索二叉树



(b)后序线索二叉链表

图 6-17 后序线索示意图

后序序列为：BDCA

### 6.4.3 线索二叉树上的运算

#### 1. 线索二叉树上的查找

##### (1) 查找指定结点在中序线索二叉树中的直接后继

若所找结点右标志 $rtag=1$ ，则右孩子域指向中序后继，否则，中序后继应为遍历右子树时的第一个访问结点，即右子树中最左下的结点（参见图 6-19）。从图 6-19 中可知， $x$  的后继为  $x_k$ 。

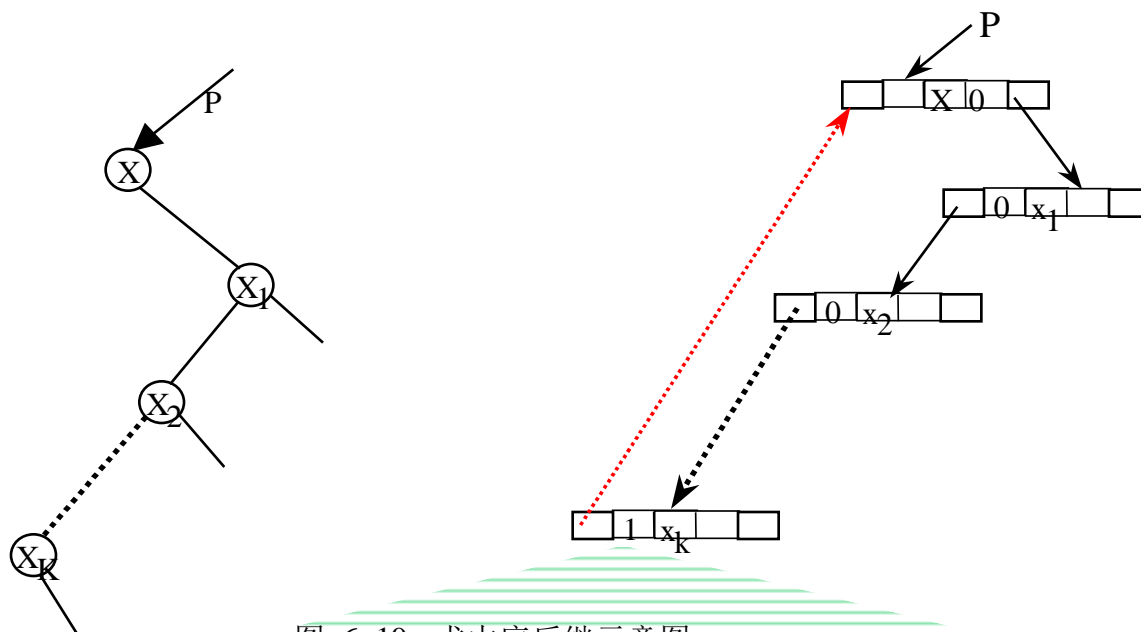


图 6-19 求中序后继示意图

## (2) 查找指定结点在中序线索二叉树中的的直接前驱

若所找结点左标志ltag=1, 则左孩子域指向中序前驱, 否则, 中序前驱应为遍历左子树时的最后一个访问结点, 即左子树中最右下的结点 (参见图 6-20)。从图6-20中可知,  $x$ 的前驱为 $x_k$ 。

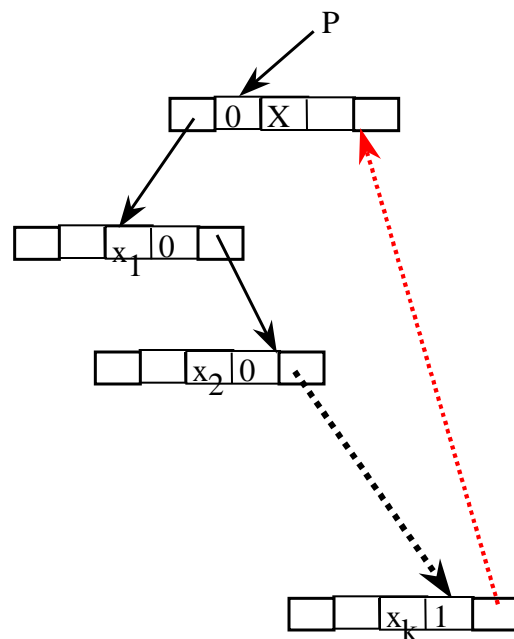
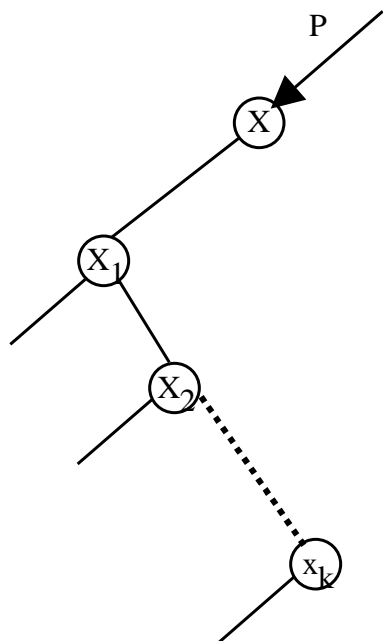
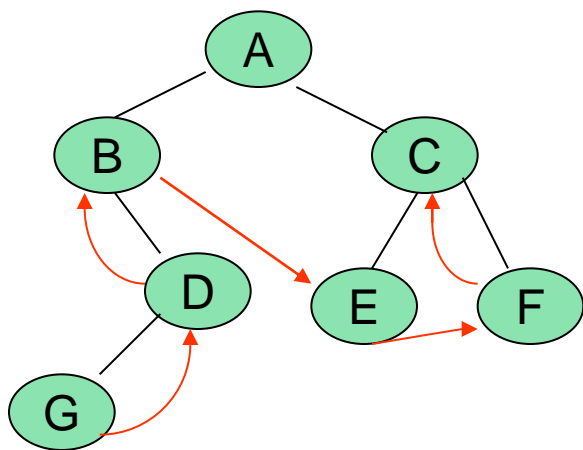


图 6-20 求中序前驱示意图

### (3) 查找指定结点在后序线索二叉树中的后继前驱

在后序线索树上面找后继需知道双亲，用三叉链表作存储结构，分三种情况。（见书P133）



后序: G D B E F C A



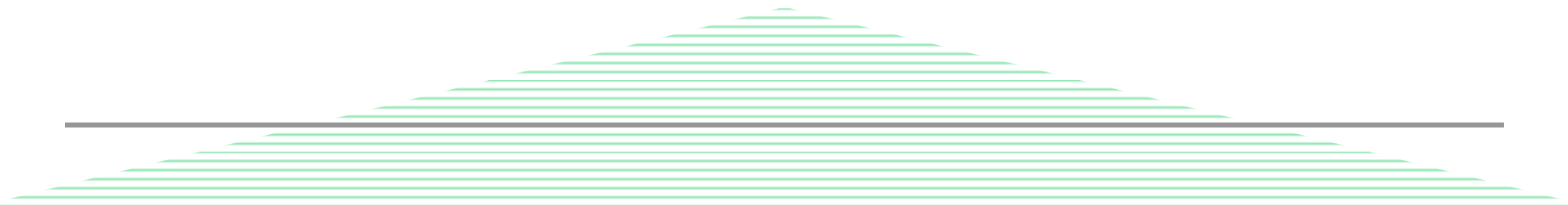


## 2. 线索二叉树上的遍历

---

遍历某种次序的线索二叉树，只要从该次序下的开始结点出发，反复找到结点在该次序下的后继，直到后继为空。这对于中序线索和前序线索二叉树很方便，但对于后序线索二叉树较麻烦(因求后序后继较麻烦)。故后序线索对于遍历没有什么意义。

不需要设栈，时间复杂度为 $O(n)$ 。但是这种方便是以增加线索为代价的，增加线索本身要花费大量时间。所以二叉树是以二链表表示，还是以线索二叉链表示，可根据具体问题而定。



## 中序遍历线索二叉树算法:

```
Status inorder_thread(BiThrTree T){
```

```
    //中序遍历中序线索二叉树，T指向头结点
```

```
    p=T->lchild;
```

```
    if(p!=T){
```

```
        while( p->ltag ==0)    p=p->lchild;
```

```
        while ( p!=T){
```

```
            visit (p->data); p= insue (p);
```

```
        }
```

```
    }
```

```
    return OK;
```

```
}//inorder_thread
```

```
Status insue(BiThrTree p){
```

```
    //求中序线索二叉树中结点P的后继
```

```
    q=p->rchild;
```

```
    if(p->rtag==0){
```

```
        while(q->ltag==0)
```

```
            q=q->lchild;
```

```
    }
```

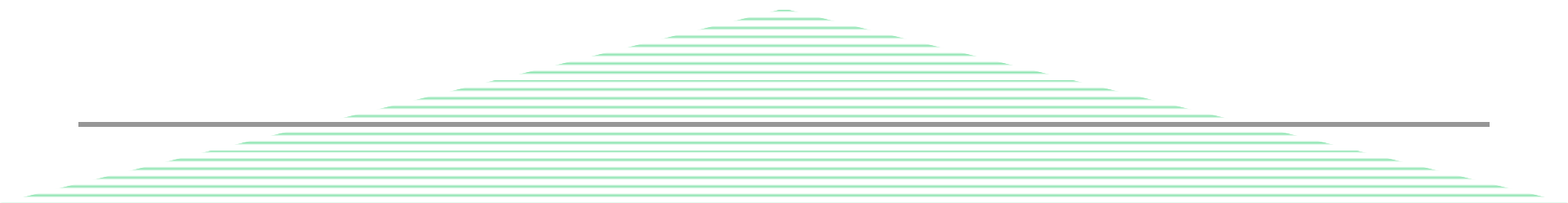
```
    return q;
```

```
}//insue
```



作业:

6.20



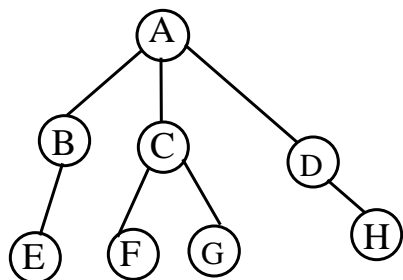
## 6.5 树和森林

### 6.5.1 树的存储结构(3种)

#### 1. 双亲表示

它是以一组连续的存储单元来存放树中的结点，每个结点有两个域：一个是data域，存放结点信息，另一个是parent域，用来存放双亲的位置(指针)。

该结构的具体描述见图6-21。



(a)树的结构

	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[maxsize-1]	
data	A	B	C	D	E	F	G	H	...	
parent	-1	0	0	0	1	2	2	3	...	

(b)树的双亲表示法

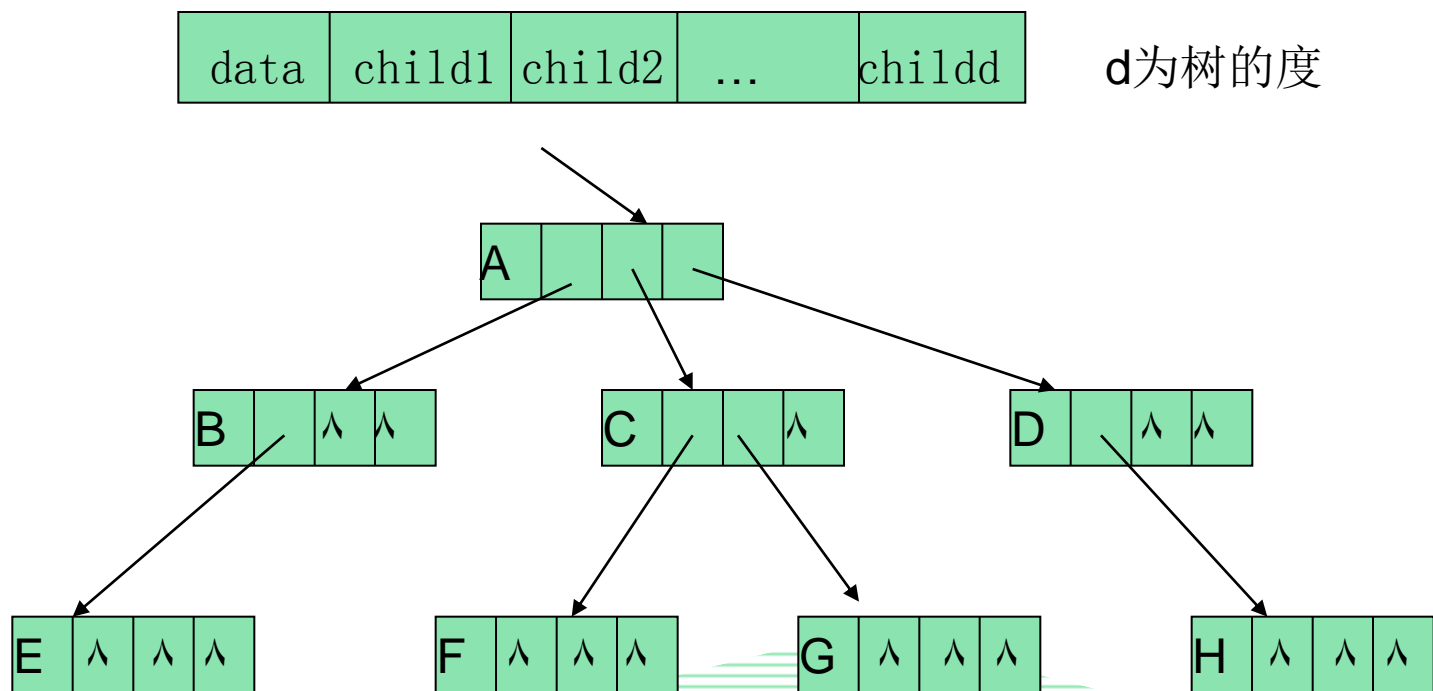
图 6-21 树的双亲表示示意图

## 2. 孩子表示法

**表示一：**由于树中每个结点可能有多棵子树，利用多重链表，即每个结点有多个指针域，其中每个指针指向一棵子树的根结点。

根据不同的结点定义，可分为两种：

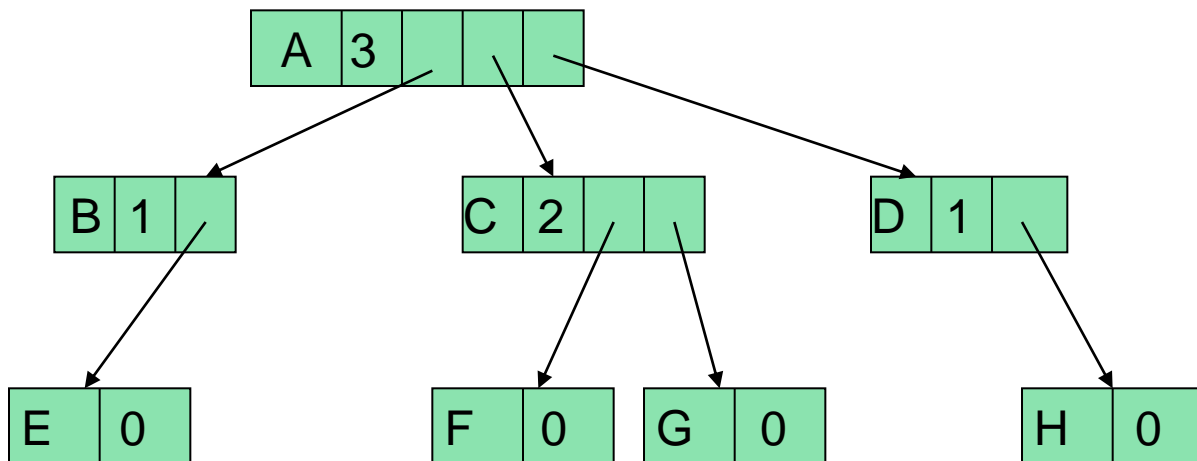
第一种：结点同构型



## 第二种：结点异构型

data	degree	child1	child2	...	child $\bar{d}$
------	--------	--------	--------	-----	-----------------

$\bar{d}$ 为结点的度，  
 $\text{degree} == \bar{d}$



**表示二：** 将一个结点所有孩子链接成一个单链表形，而树中有若干个结点，故有若干个单链表，每个单链表有一个表头结点，所有表头结点用一个数组来描述。具体可分为两种：

1.孩子链表：

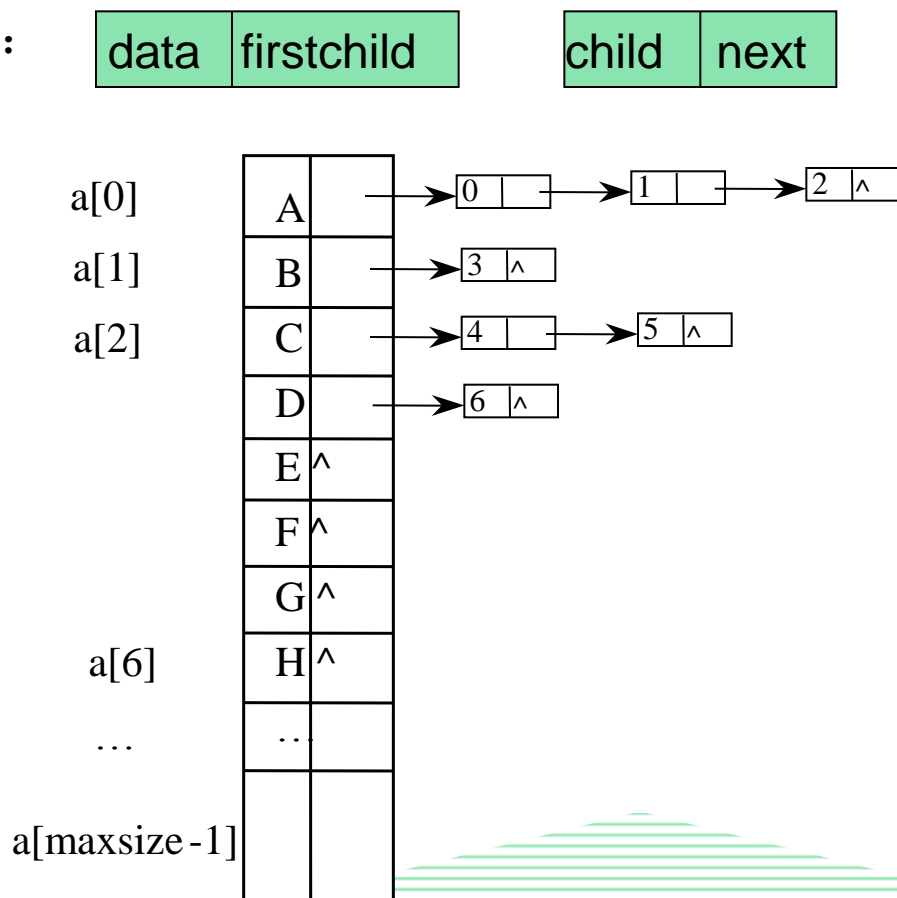


图 6-22 树的孩子表示法（图 6-21（a）中树）示意图

## 2.带双亲的孩子链表:

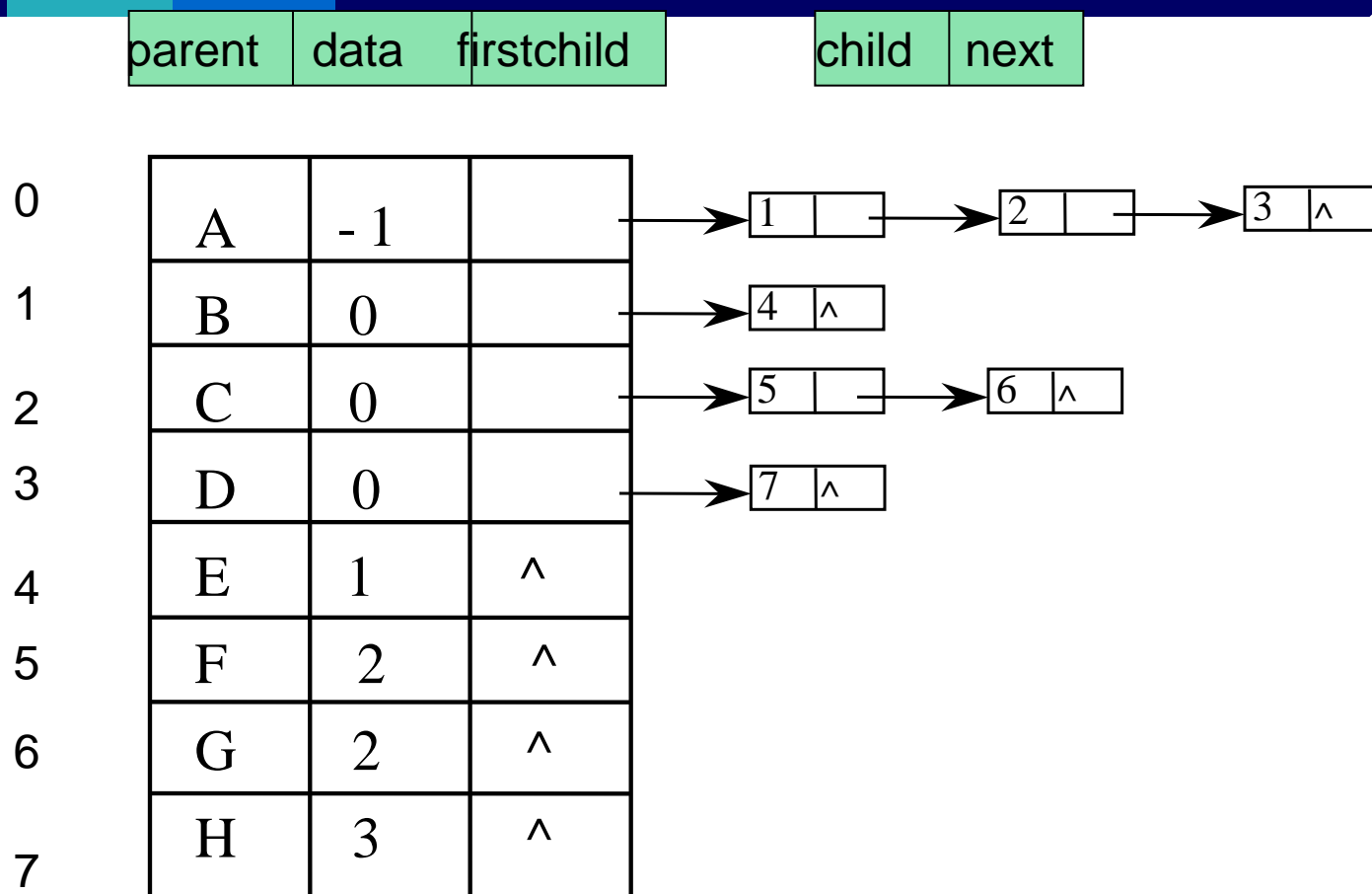


图 6-23 树的双亲孩子表示法示意图





### 3.孩子兄弟表示法

---

类似于二叉链表，但第一根链指向第一个孩子，第二根链指向下一个兄弟。将图6-21(a)的树用孩子兄弟表示法表示，见图6-24。

结点定义：

fistchild	data	nextsibling
-----------	------	-------------



## 6.5.2 树、森林和二叉树的转换

### 1. 树转换成二叉树

可以分为三步：

#### (1) 连线

指相邻兄弟之间连线。

#### (2) 抹线

指抹掉双亲与除左孩子外其它孩子之间的连线。

#### (3) 旋转

只需将树作适当的旋转。

具体实现过程见图6-25。



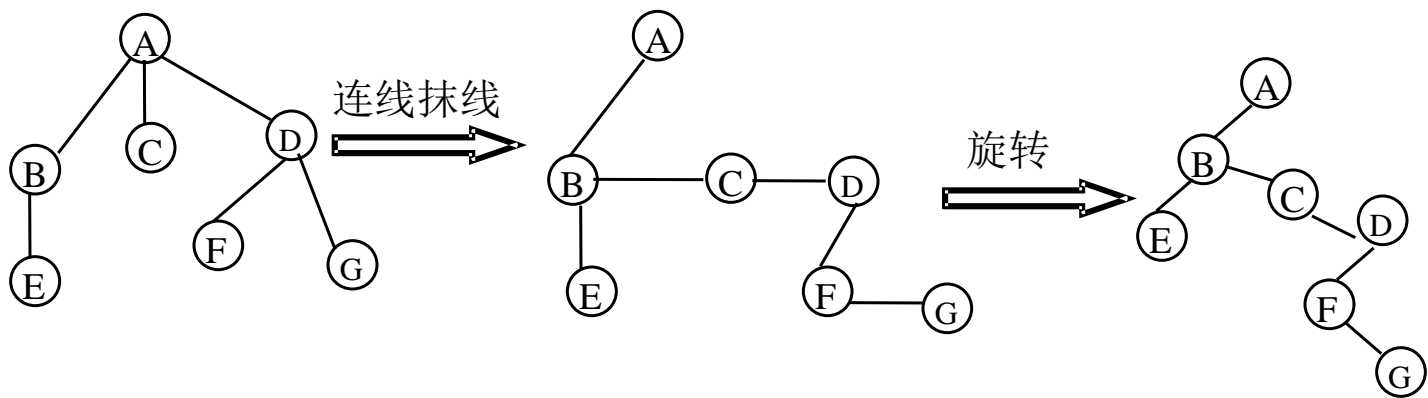


图 6-25 树转换成二叉树示意图



## 2. 森林转换成二叉树

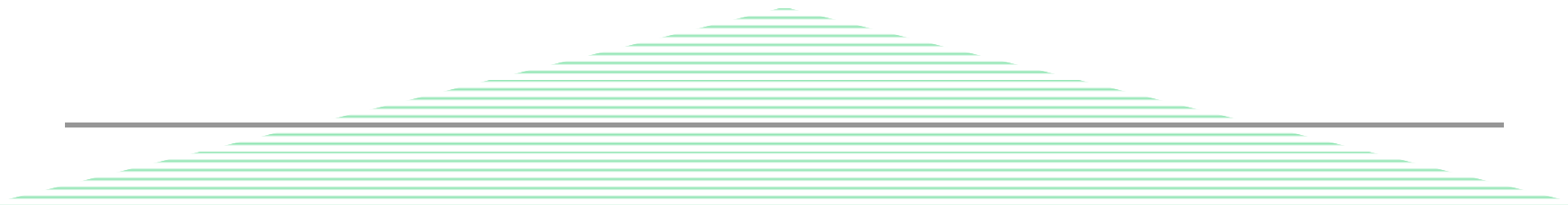
---

### (1) 将森林中每一棵树分别转换成二叉树

这在刚才的树转换成二叉树中已经介绍过。

### (2) 合并

使第 $n$ 棵树接入到第 $n-1$ 棵的右边并成为它的右子树，第  $n-1$  棵二叉树接入到第 $n-2$  棵的右边并成为它的右子树，...，第2棵二叉树接入到第1棵的右边并成为它的右子树，直到最后剩下一棵二叉树为止。



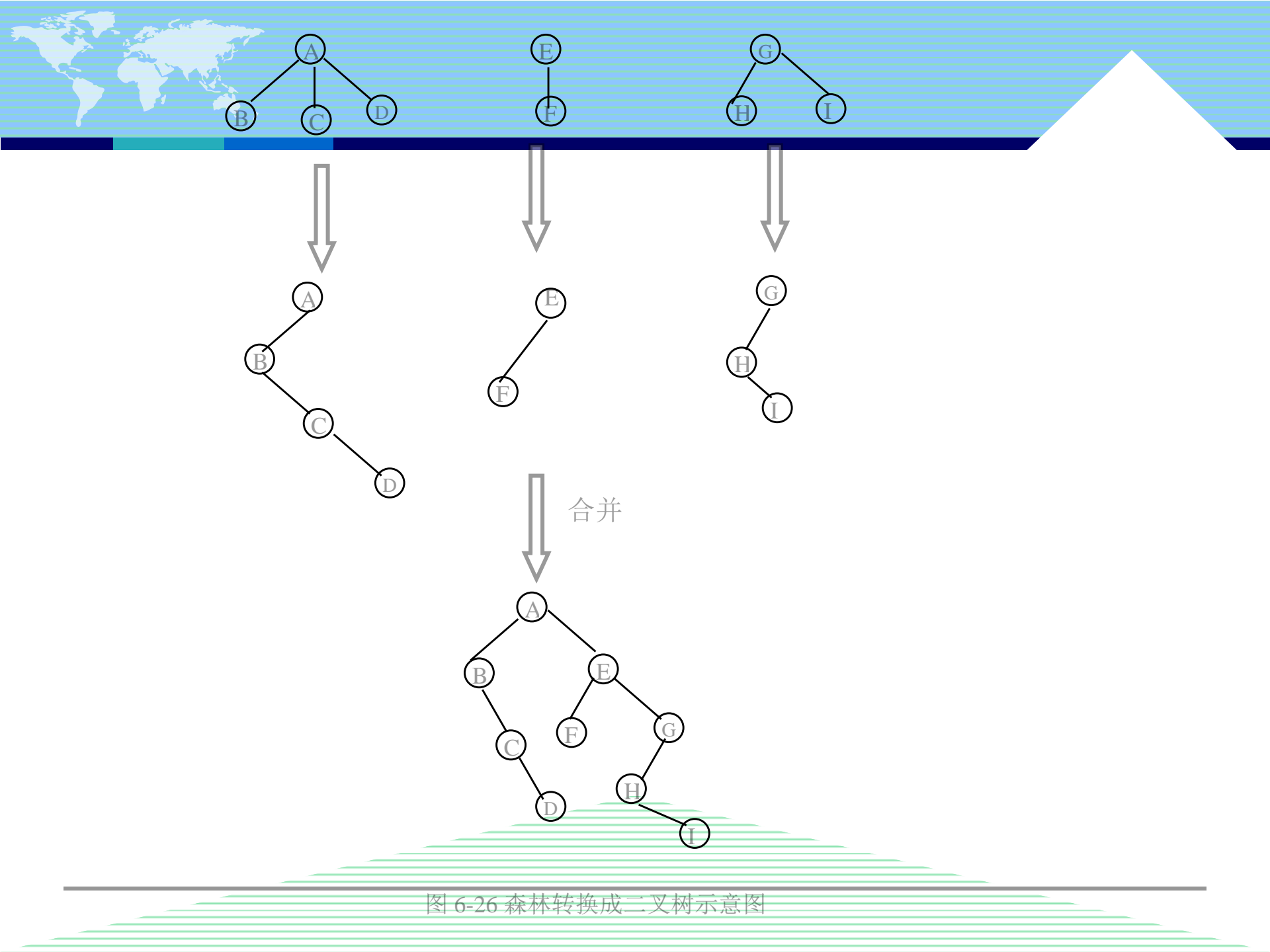


图 6-26 森林转换成二叉树示意图



### 3. 二叉树还原成树或森林

---

#### (1) 右链断开

将二叉树的根结点的右链及右链的右链等全部断开，得到若干棵无右子树的二叉树。

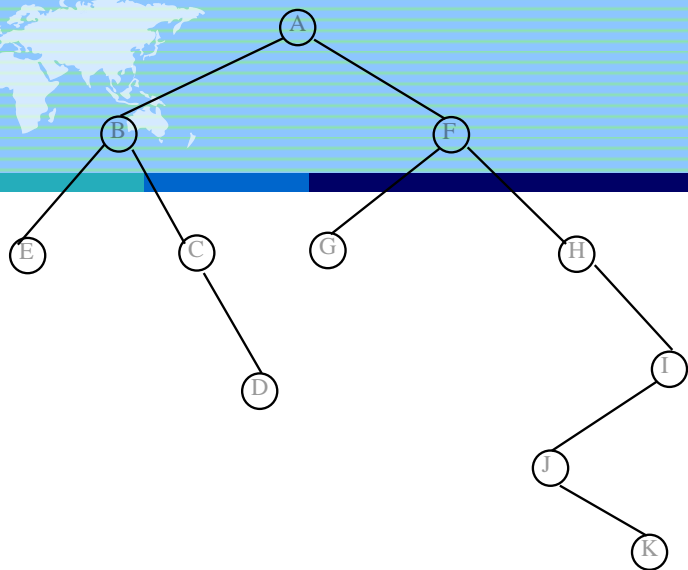
具体操作见图6-27(b)。

#### (2) 二叉树还原成树

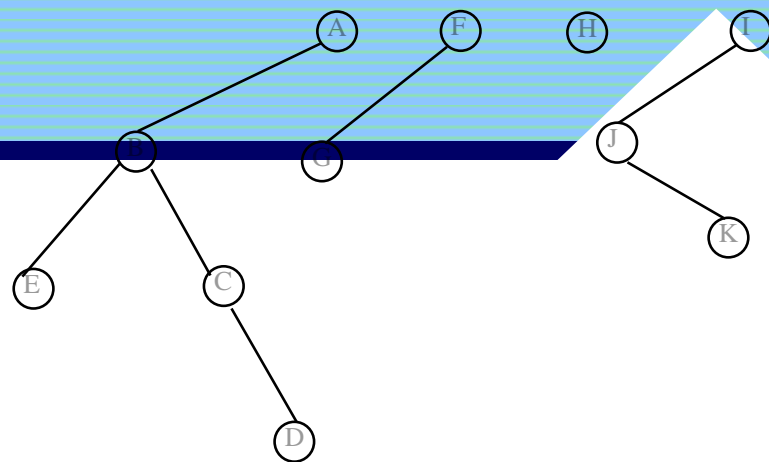
将(1)中得到的每一棵二叉树都还原成树（与树转换成二叉树的步骤刚好相反）。

具体操作步骤见图6-27(c)。

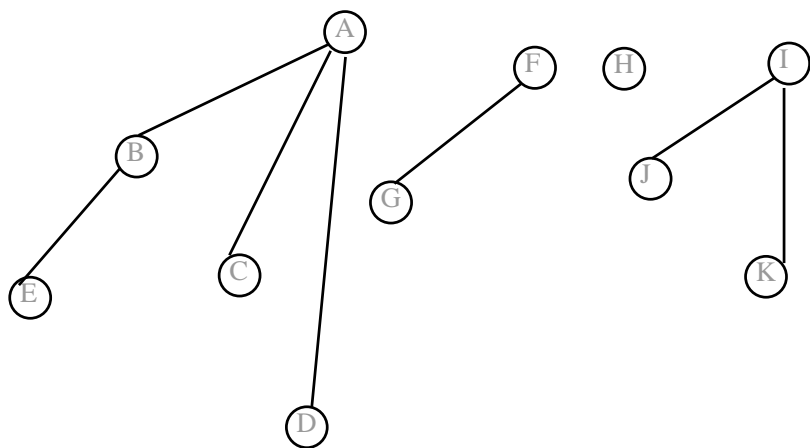




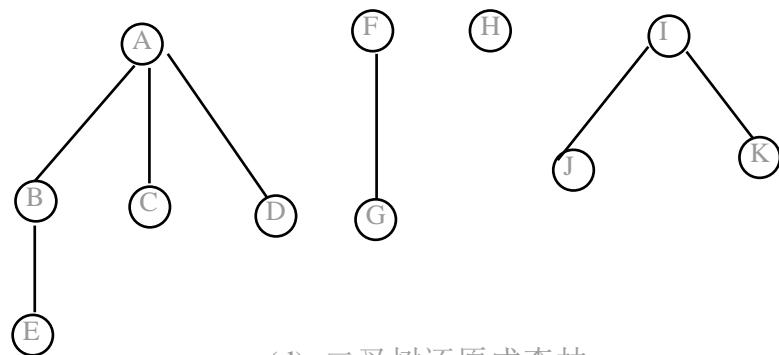
(a) 一个森林得到的二叉树



(b) 断开根的右链得到 4 棵无右子树的二叉树



(c) 四棵二叉树还原成四棵树



(d) 二叉树还原成森林

图 6-27 二叉树还原成森林过程



## 6.5.3 树和森林的遍历

在树和森林中，一个结点可能有两棵以上的子树，所以不宜讨论它们的中序遍历，即树和森林只有先序遍历和后序遍历。

### 1. 树的遍历

#### **(1) 树的先序遍历**

若树非空，则先访问根结点，然后依次先序遍历各子树。

#### **(2) 树的后序遍历**

若树非空，则依次后序遍历各子树，最后访问根结点。

举例







## 2. 森林的遍历

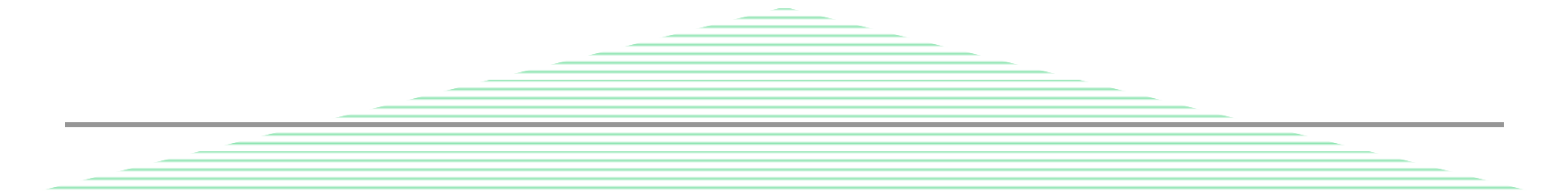
---


### (1) 森林的先序遍历

若森林非空，则先访问森林中第一棵树的根结点，再先序遍历第一棵树各子树，接着先序遍历第二棵树、第三棵树、...、直到最后一棵树。

### (2) 森林的中序遍历

若森林非空，中序遍历森林中第一棵树的根结点的子树森林，访问第一棵树的根结点，接着中序遍历第二棵树、第三棵树、...、直到最后一棵树。





### 3. 二叉树、树、森林遍历的等价关系

树	二叉树	森林
先根	先序	先序
后根	中序	中序



作业:

6.21、6.22、6.23、6.29、6.60



## 6.6 哈夫曼树及其应用 (Huffman, 最优二叉树)

### 6.6.1 基本术语

#### 1. 路径和路径长度

在一棵树中，从一个结点往下可以达到的孩子或子孙结点之间的通路，称为路径。通路中分支的数目称为路径长度。

树的路径长度，从树根到每个结点的路径长度之和。完全二叉树为最短的二叉树。

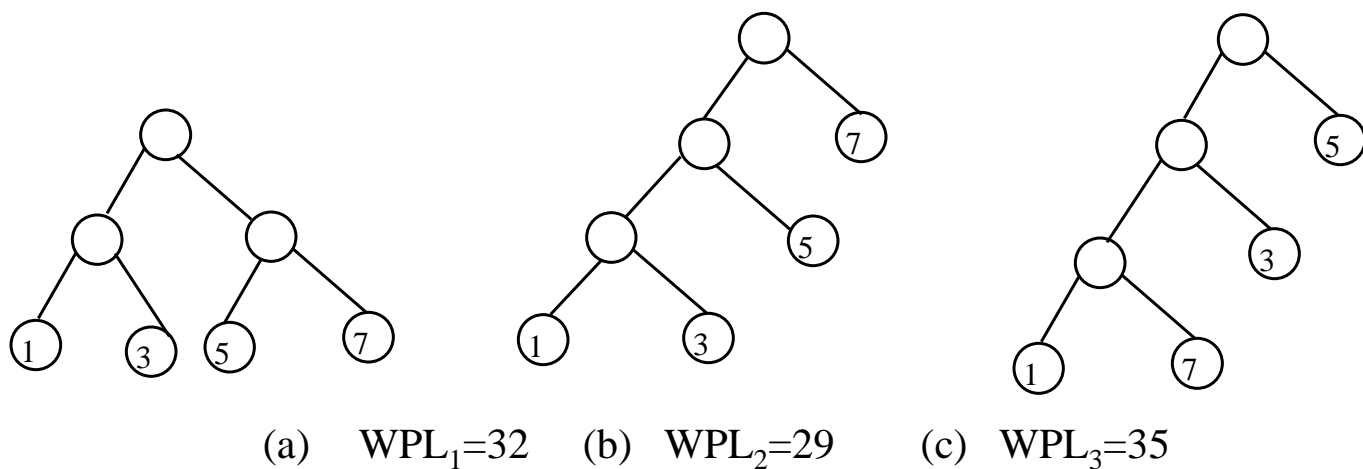
#### 2. 结点的权及带权路径长度

若将树中结点赋给一个有着某种含义的数值，则这个数值称为该结点的权。

结点的带权路径长度为：从根结点到该结点之间的路径长度与该结点的权的乘积。

### 3. 树的带权路径长度

树的带权路径长度规定为所有叶子结点的带权路径长度之和，记为  $wpl = \sum_{i=1}^n w_i l_i$ ，其中  $n$  为叶子结点数目， $w_i$  为第  $i$  个叶子结点的权值， $l_i$  为第  $i$  个叶子结点的路径长度。



$$WPL_1 = 1 \times 2 + 3 \times 2 + 5 \times 2 + 7 \times 2 = 32$$

$$WPL_2 = 1 \times 3 + 3 \times 3 + 5 \times 2 + 7 \times 1 = 29$$

$$WPL_3 = 1 \times 3 + 7 \times 3 + 3 \times 2 + 5 \times 1 = 35$$

图 6-28 具有不同带权路径长度的二叉树



## 6.6.2 哈夫曼树构造

### 1. 哈夫曼树的定义

在一棵二叉树中，若带权路径长度达到最小，称这样的二叉树为最优二叉树，也称为哈夫曼树(Huffman tree)。

### 2. 哈夫曼树的构造

假设有 $n$ 个权值，则构造出的哈夫曼树有 $n$ 个叶子结点。 $n$ 个权值分别设为  $w_1, w_2, \dots, w_n$ , 则哈夫曼树的构造规则为：

(1) 将 $w_1, w_2, \dots, w_n$ 看成是有 $n$ 棵树的森林(每棵树仅有一个结点);

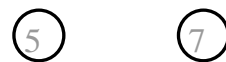


- (2) 在森林中选出两个根结点的权值最小的树合并，作为一棵新树的左、右子树，且新树的根结点权值为其左、右子树根结点权值之和；
- (3) 从森林中删除选取的两棵树，并将新树加入森林；
- (4) 重复(2)、(3)步，直到森林中只剩一棵树为止，该树即为我们所求得的哈夫曼树。

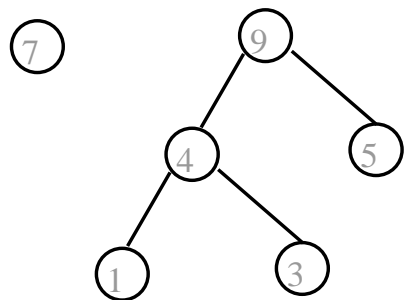
$n$  个权值构造哈夫曼树需 $n-1$ 次合并，每次合并，森林中的树数目减1，最后森林中只剩下一棵树，即为我们求得的哈夫曼树。



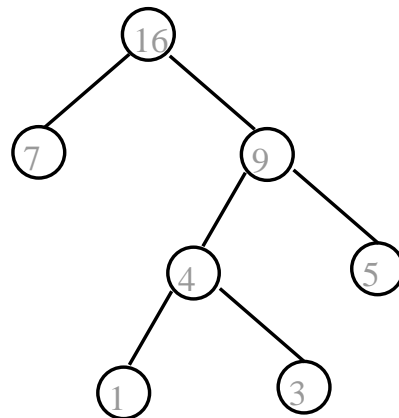
(a) 初如森林



(b) 一次合并后的森林



(c) 二次合并后的森林



(d) 三合并后的森林

图 6-29 哈夫曼树的构造过程





### 6.6.3 哈夫曼树的应用

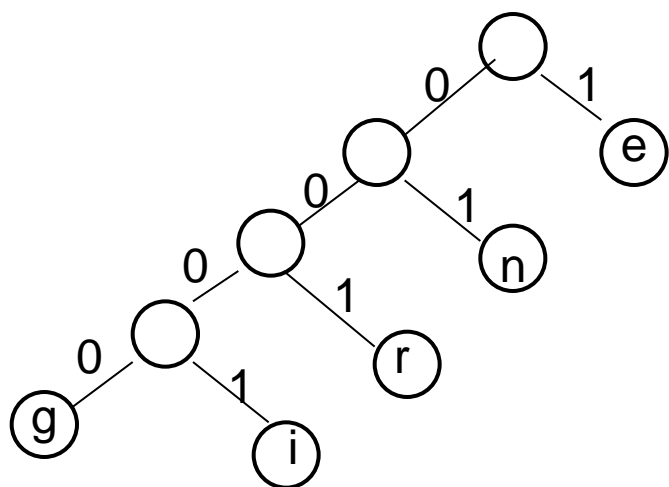
#### 1. 哈夫曼编码

---

通信中，可以采用0,1的不同排列来表示不同的字符，称为二进制编码。而哈夫曼树在数据编码中的应用，是数据的最小冗余编码问题，它是数据压缩学的基础。若每个字符出现的频率相同，则可以采用等长的二进制编码，若频率不同，则可以采用不等长的二进制编码，频率较大的采用位数较少的编码，频率较小的字符采用位数较多的编码，这样可以使字符的整体编码长度最小，这就是最小冗余编码的问题。

而哈夫曼编码就是一种不等长的二进制编码，且哈夫曼树是一种最优二叉树，它的编码也是一种最优编码，在哈夫曼树中，规定往左编码为0，往右编码为1，则得到叶子结点编码为从根结点到叶子结点中所有路径中0和1的顺序排列。

例如，给定权{3, 2, 1, 1, 1}，分别代表数据{e,n,g,i,r}的出现次数，得到的哈夫曼树及编码见图6-32 (假定权值就代表该字符名字)。



(a)哈夫曼树

e 的编码为: 1

n 的编码为: 01

g 的编码为: 0000

i 的编码为: 0001

r 的编码为: 001

(b)哈夫曼编码

图 6-32 构造哈夫曼树及哈夫曼编码

“engineer”编码:

哈夫曼编码: 101000000010111001, 共18位

ASCII码: 一般的存储方式, 一个字母8位, 8\*8 (个字母=64位)



## 2. 哈夫曼译码

---

在通信中，若将字符用哈夫曼编码形式发送出去，对方接收到编码后，将编码还原成字符的过程，称为哈夫曼译码。

例如：0101000111

1周课程设计：P149练习册

作业：6.26

---

