# 第6章 树和二叉树

# 6.1 树的定义和基本术语

## 1. 树的定义

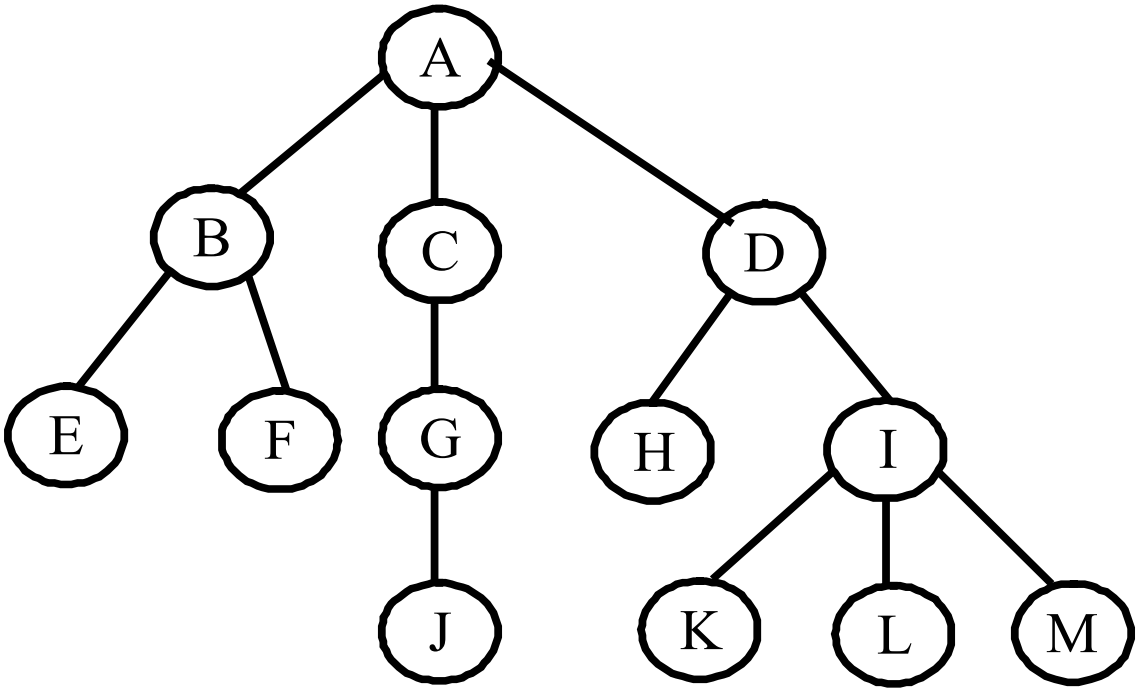
树是n（n≥0）个结点的有限集。当n=0时，称为空树；在任意一棵非空树中满足如下条件：

(1) 有且仅有一个特定的称为根（root）的结点，它没有直接前驱，但有零个或多个直接后继。

(2) 其余n-1个结点可以划分成m（m>0）个互不相交的有限集$T_1$，$T_2$，$T_3$，…，$T_m$，其中$T_i$又是一棵树，称为根root的子树。 每棵子树的根结点有且仅有一个直接前驱，但有零个或多个直接后继。

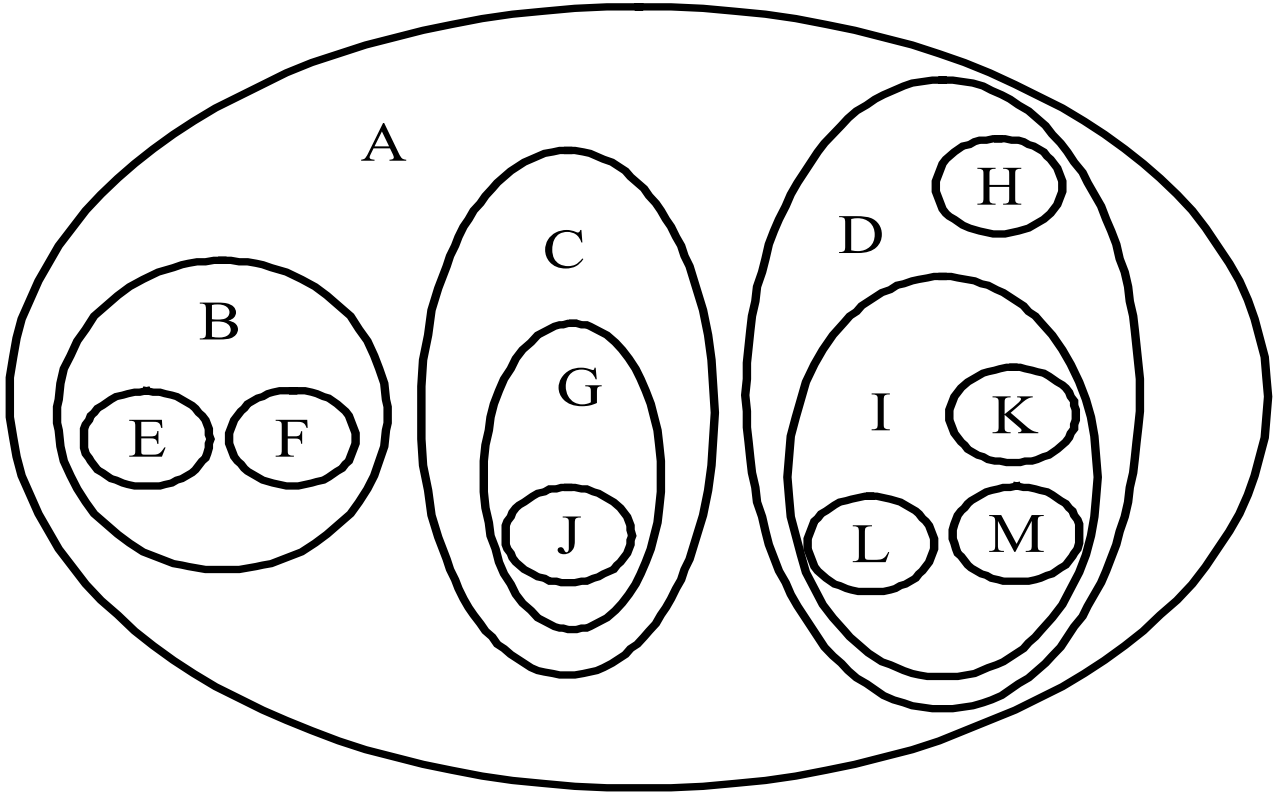
## 2. 树的逻辑表示法

- (1) 树型表示法。这是树的最基本的表示,使用一棵倒置的树表示树结构,非常直观和形象。



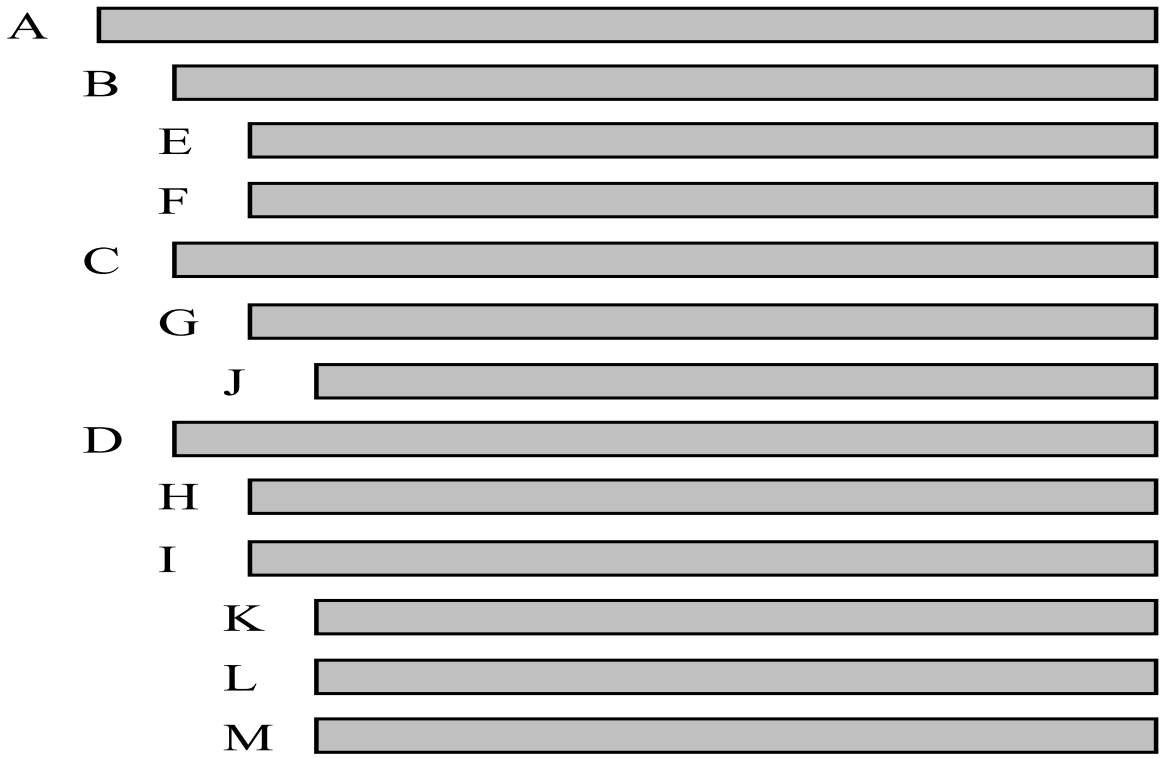

树形表示法

- (2) 文氏图表示法。使用集合以及集合的包含关系描述树结构。




文氏图表示法

- (3)凹入表示法。使用线段的伸缩描述树结构。




凹入表示法

- (4)括号表示法(广义表表示法)。将树的根结点写在括号的左边,除根结点之外的其余结点写在括号中并用逗号间隔来描述树结构。

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

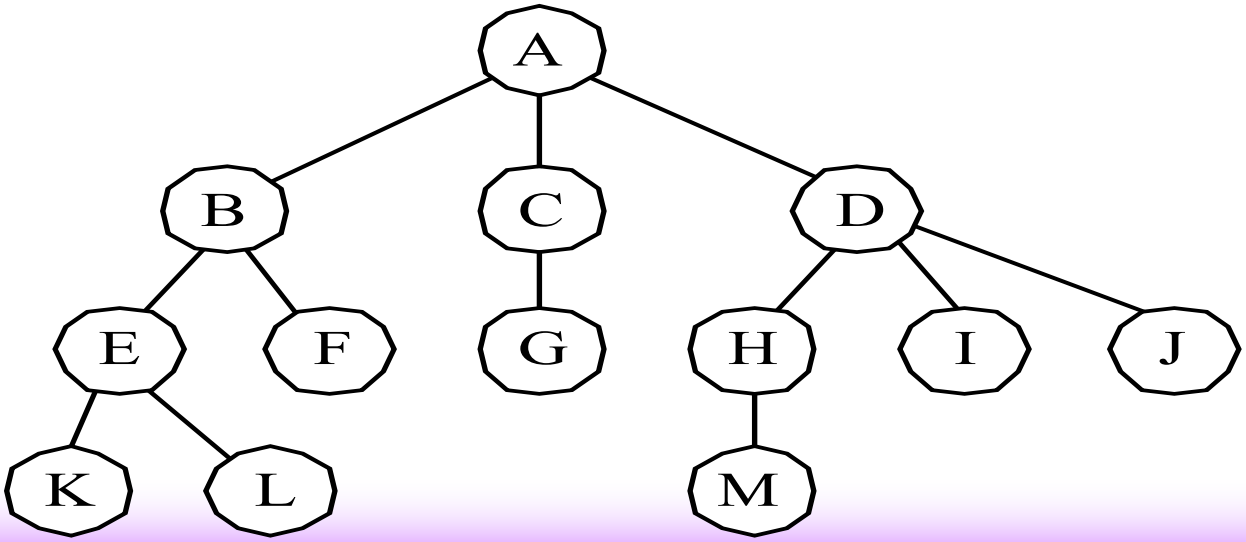
括号表示法

## 3. 树的基本术语

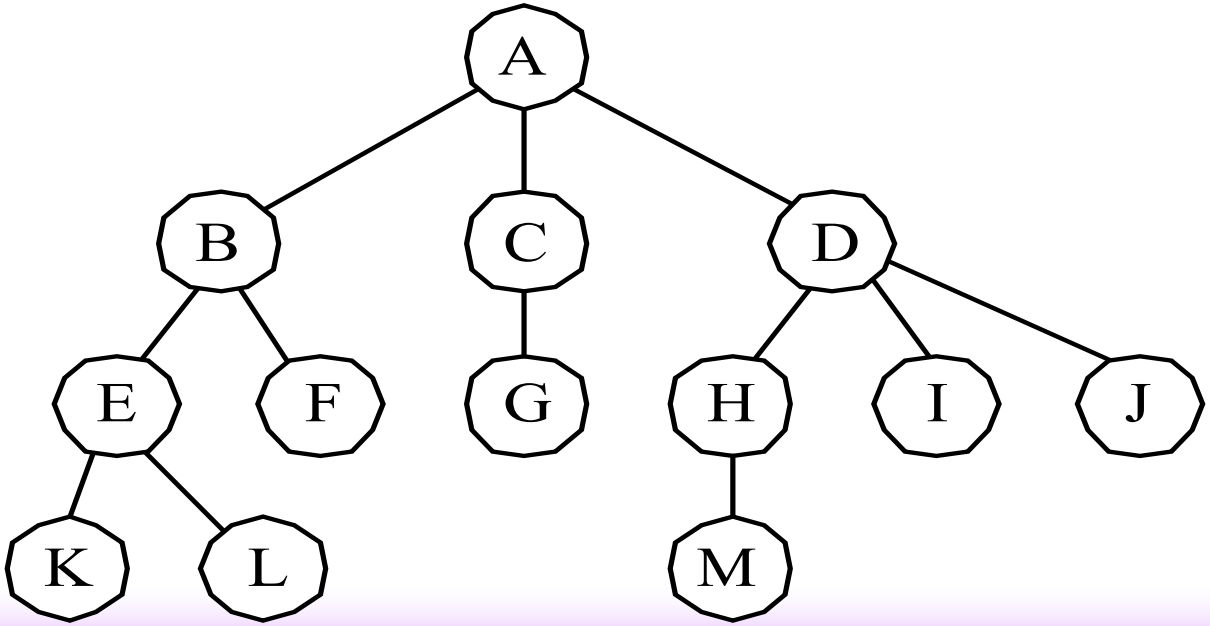
●结点：包含一个数据元素及若干指向其子树的分支。

●结点的度和树的度(degree)：结点拥有的子树个数。树内各结点的度的最大值。

●叶子(leaf)和分支结点：度为0的结点，也称为终端结点。度不为0的结点，也称为非终端结点。除根结点外，分支结点也称为内部结点。



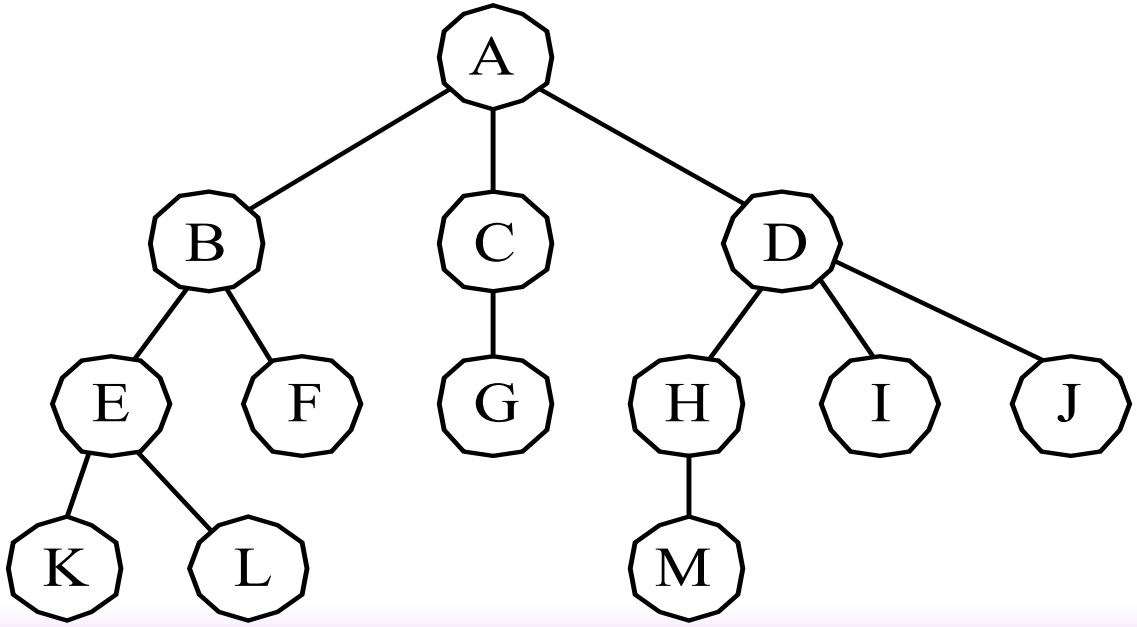
- 孩子、双亲、兄弟、堂兄弟：结点的子树的根称为该结点的孩子。相应地，该结点称为孩子的双亲。同一双亲的孩子之间互称兄弟。其双亲在同一层的结点互为堂兄弟。
- B、C、D是A的孩子。
- A 是B、C 、D的双亲。
- 结点H、I、 J互为兄弟结点。



●路径与路径长度：对于任意两个结点$k_i$和$k_j$，若树中存在一个结点序列$k_i, k_{i1}, k_{i2}, \cdots, k_{in}, k_j$，使得序列中除$k_i$外的任一结点都是其在序列中的前一个结点的后继，则称该结点序列为由$k_i$到$k_j$的一条路径，用路径所通过的结点序列$(k_i, k_{i1}, k_{i2}, \cdots, k_j)$表示这条路径。路径的长度等于路径所通过的结点数目减1(即路径上分支数目)。可见，路径就是从$k_i$出发“自上而下”到达$k_j$所通过的树中结点序列。显然，从树的根结点到树中其余结点均存在一条路径。

●结点的祖先和子孙：从根结点到该结点的路径上的所有结点。以某结点为根的子树中的任一结点都称为该结点的子孙。

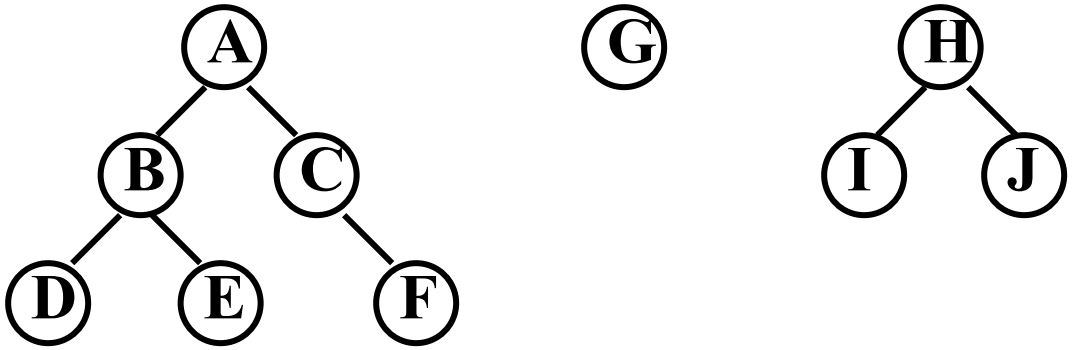
●结点的层次和树的高度(深度)：从根结点开始定义，根为第一层，根的孩子为第二层，依此类推。树中结点的最大层次。



- 有序树和无序树：在树中，如果各子树$T_i$是按照一定的次序从左向右安排的，且相对次序是不能随意改变的，则称为有序树，否则称为无序树。

- 森林： m（m≥0）棵互不相交的树的集合。将一棵非空树的根结点删去，树就变成一个森林；反之，给m棵独立的树增加一个根结点，并把这m棵树作为该结点的子树，森林就变成一棵树。

## 森林




树属于森林。

树是一个二元组 Tree = ( root ， F ) 。

root : 根结点。

F : m(m≥0)棵树的森林， $F = (T_1, T_2, \ldots, T_m)$。

$T_i$ 称为 root 的第 i 棵子树。

## 4. 树的基本运算

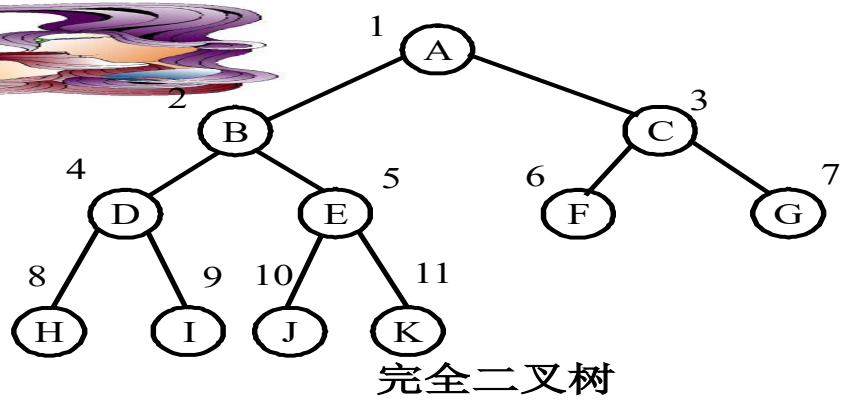
树的运算主要分为三大类：

第一类，寻找满足某种特定关系的结点，如寻找当前结点的双亲结点等；

第二类，插入或删除某个结点，如在树的当前结点上插入一个新结点或删除当前结点的第i个孩子结点等；

第三类，遍历树中每个结点，这里着重介绍。

# 6.2 二 叉 树



完全二叉树

## 1. 二叉树的定义

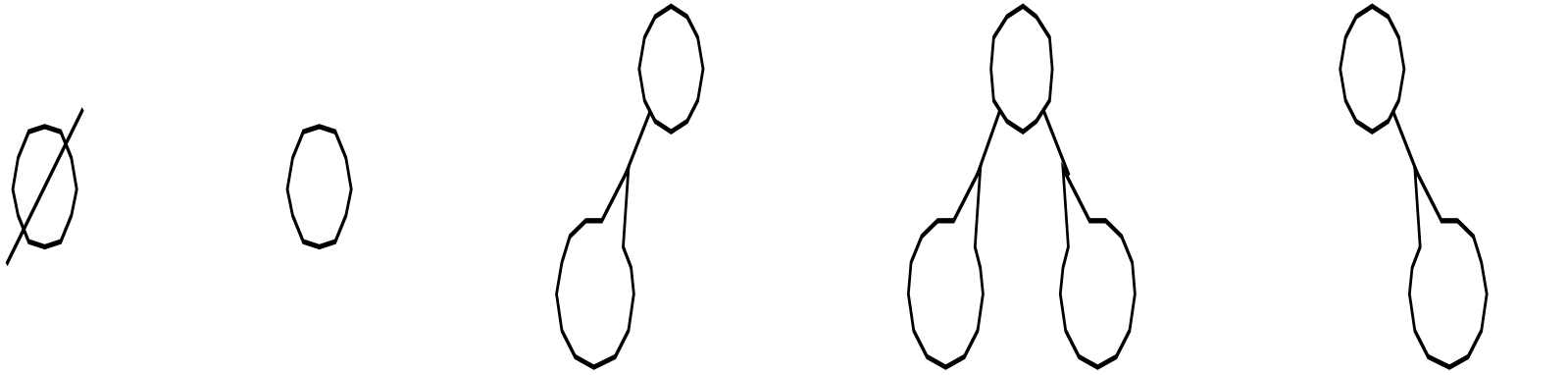
满足以下两个条件的树形结构叫做二叉树（Binary Tree）：

（1） 每个结点的度都不大于2；

（2） 每个结点的孩子结点次序不能任意颠倒。

由此定义可以看出，一个二叉树中的每个结点只能含有0、 1或2个孩子，而且每个孩子有左右之分。把位于左边的孩子叫做左孩子，位于右边的孩子叫做右孩子。

# 二叉树的五种基本形态



(a) 空二叉树

(b) 只有根结点的二叉树

(c) 只有左子树的二叉树

(d) 左右子树均非空的二叉树

(e) 只有右子树的二叉树

## 2. 二叉树的性质

●性质1: 在二叉树的第i层上至多有$2^{i-1}$个结点(i≥1)。

证明： 用数学归纳法。

1) 当$i$=1时，整个二叉树只有一根结点，此时$2^{i-1}=2^0=1$，结论成立。

2) 设i=k时结论成立，即第k层上结点总数最多为$2^{k-1}$个。

现证明当i=k+1时， 结论成立：

因为二叉树中每个结点的度最大为2，则第k+1层的结点总数最多为第k层上结点最大数的2倍，即$2\times 2^{k-1}=2^{(k+1)-1}$，故结论成立。

度为m的树中第i层上至多有$m^{i-1}$个结点, (i≥1)。

●性质2: 深度为k的二叉树至多有$2^k-1$个结点（k≥1）。

证明：因为深度为k的二叉树，其结点总数的最大值是将二叉树每层上结点的最大值相加，所以深度为k的二叉树的结点总数至多为

$$\sum_{i=1}^{k}\text{第}i\text{层上的最大结点个数}=\sum_{i=1}^{k}2^{i-1}=2^k-1$$

深度为k的m叉树至多有$\frac{m^k-1}{m-1}$个结点。

$$\sum_{i=1}^{k}m^{i-1}=m^0+m^1+\ldots+m^{k-1}=\frac{m^k-1}{m-1}$$

- 性质3：对任意一棵二叉树，若终端结点数为$n_0$，度为2的结点数为$n_2$，则$n_0=n_2+1$。

证明：设二叉树中结点总数为n，$n_1$为二叉树中度为1的结点总数，设二叉树中分支数目为B 。

①$n=n_0+n_1+n_2$

除根结点外，每个结点均对应一个进入它的分支：

②$n=B+1$

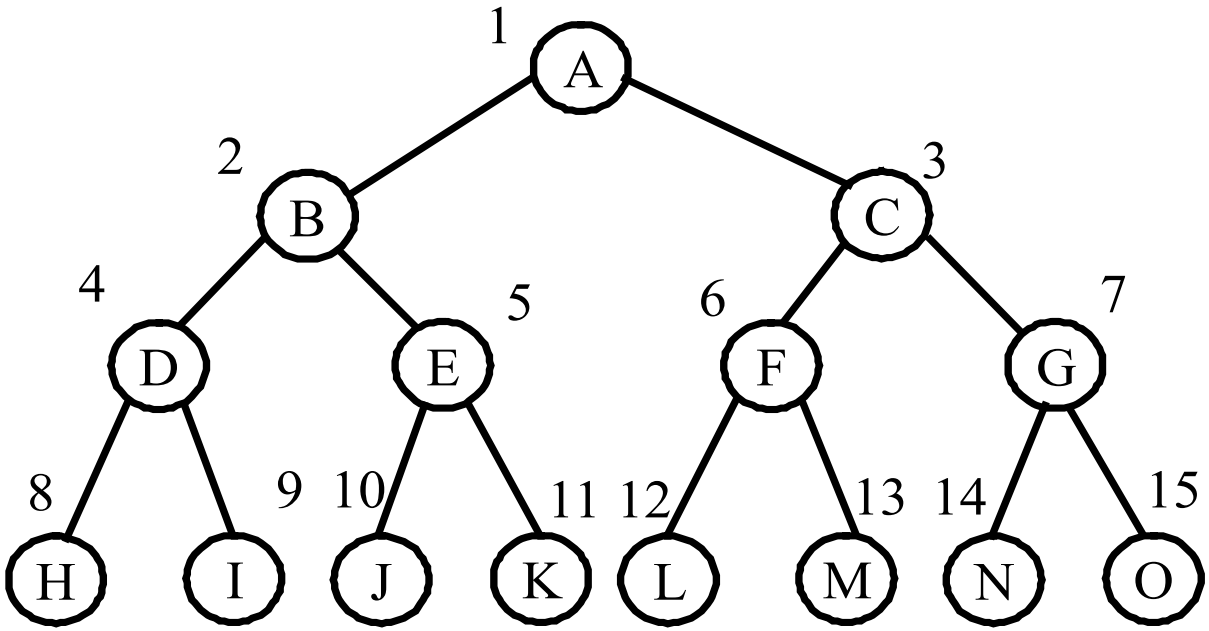
二叉树中的分支都是由度为1和度为2的结点发出

③$B=n_1+2n_2$

## ●满二叉树：

深度为k且有$2^k-1$个结点的二叉树。在满二叉树中，每层结点都是满的，即每层结点都具有最大结点数。

满二叉树的顺序表示，即从二叉树的根开始， 层间从上到下， 层内从左到右，逐层进行编号（1， 2， …，n）。

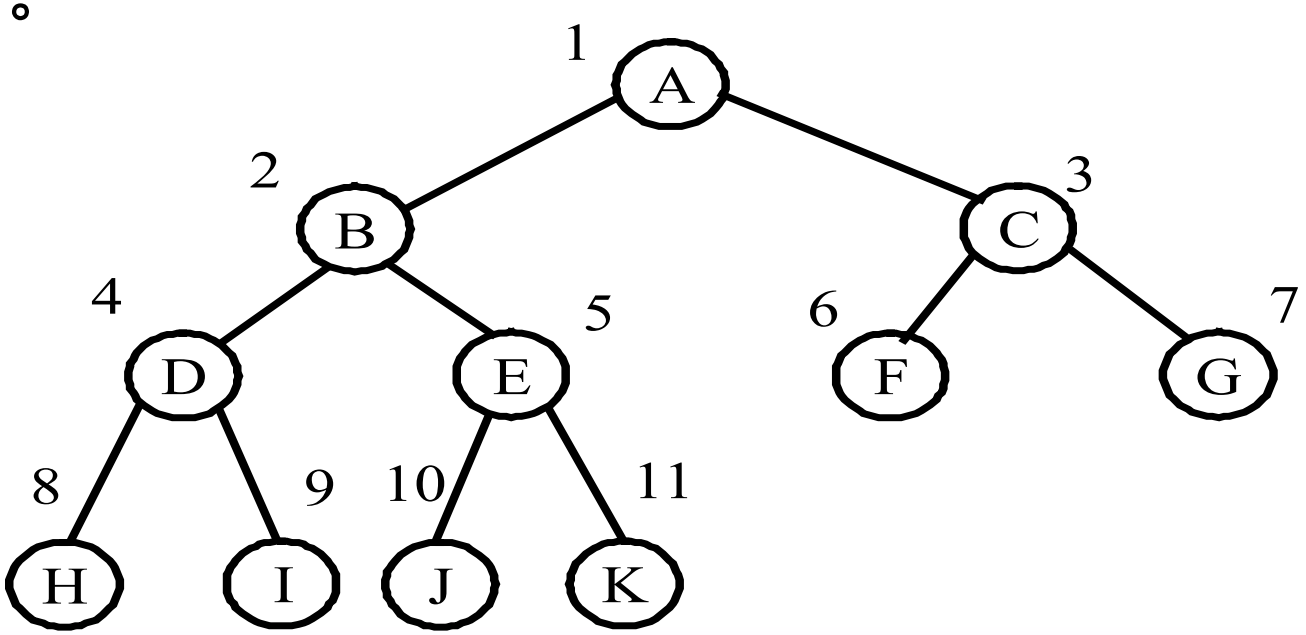



满二叉树

## ●完全二叉树：

深度为k，结点数为n的二叉树，如果其结点1~n的位置序号分别与满二叉树的结点1~n的位置序号一一对应，则为完全二叉树，

满二叉树必为完全二叉树， 而完全二叉树不一定是满二叉树。




完全二叉树

- 性质4：具有n个结点的完全二叉树的深度为$\lfloor \log_2 n \rfloor + 1$或$\lceil \log_2 (n+1) \rceil$。

证明：设n个结点的完全二叉树的深度为k，根据性质2有 $2^{k-1}-1 < n \leq 2^k-1$

可得 $2^{k-1} \leq n < 2^k$，

即 $k-1 \leq \log_2 n < k$

因为k是整数，所以$k-1=\lfloor \log_2 n \rfloor$，$k=\lfloor \log_2 n \rfloor+1$，故结论成立。

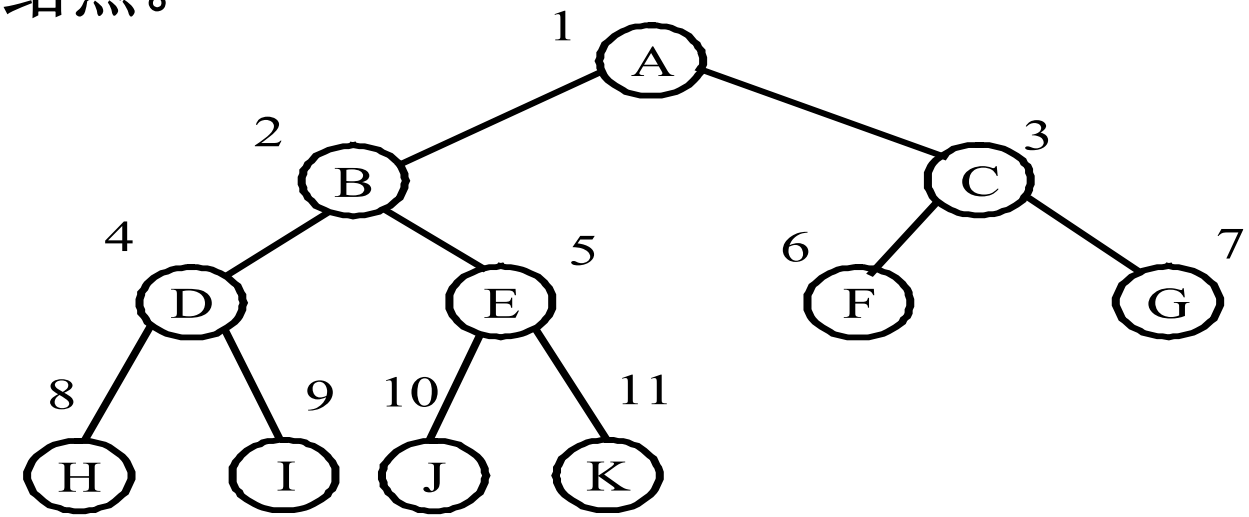
$2^{k-1} < n+1 \leq 2^k \rightarrow k-1 < \log_2(n+1) \leq k \rightarrow$

$k=\lceil \log_2 (n+1) \rceil$

●性质5:对完全二叉树中编号为i的结点(1≤i≤n,n≥1,n为结点数)有:

(1)若i=1,则结点i是二叉树的根，无双亲。

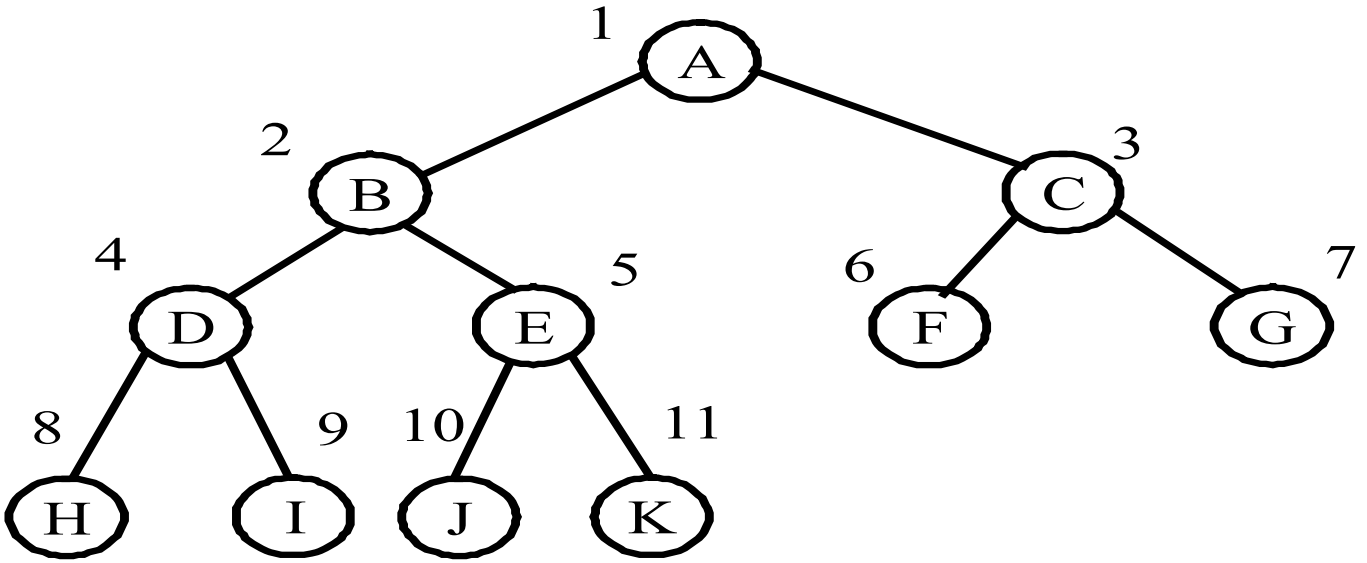
若i>1,则它的双亲结点的编号为⌊i/2⌋。当i为偶数时,其双亲结点的编号为i/2,它是双亲结点的左孩子结点,当i为奇数时,其双亲结点的编号为(i-1)/2,它是双亲结点的右孩子结点。




完全二叉树

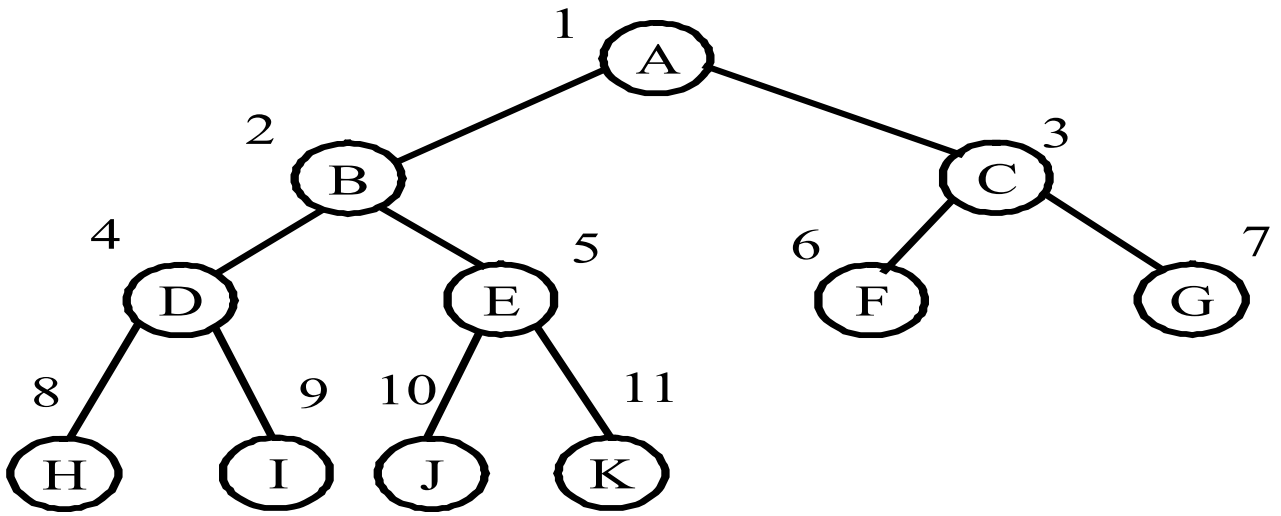
(2)若编号为i的结点有左孩子结点,则左孩子结点的编号为2i；若编号为i的结点有右孩子结点,则右孩子结点的编号为(2i+1)。

当2i>n，则结点i无左孩子，无左孩子则结点i为叶子结点；当2i+1>n，则结点无右孩子。




完全二叉树

(3)若n为奇数,则每个分支结点都既有左孩子结点,也有右孩子结点；若n为偶数,则编号最大的分支结点(编号为n/2)只有左孩子结点,没有右孩子结点,其余分支结点都有左、右孩子结点。



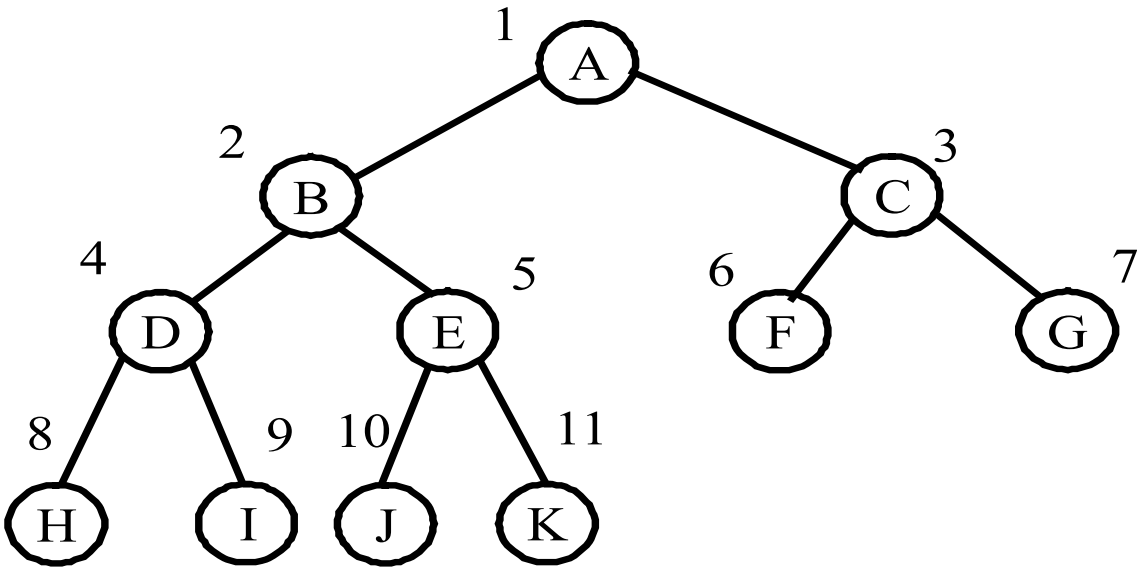

完全二叉树

# 3. 二叉树的存储结构

二叉树的存储结构有两种： 顺序存储结构和链式存储结构。

- 顺序存储结构

编号从小到大的顺序就是结点存放在连续存储单元的先后次序

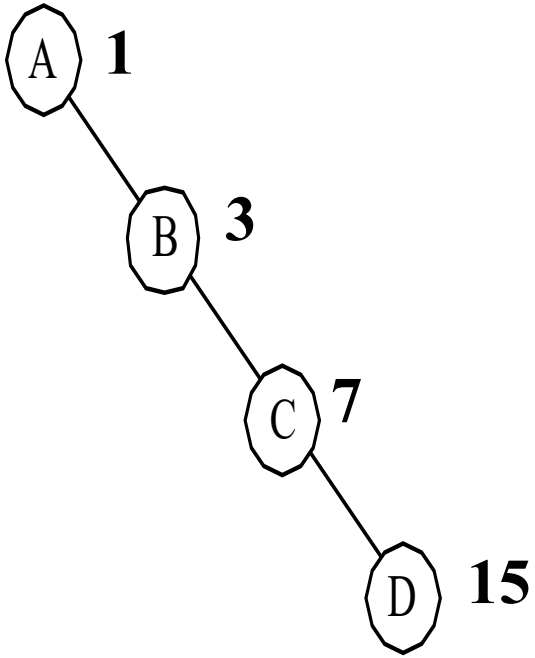



完全二叉树

| A | B | C | D | E | F | G | H | I | J | K | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

完全二叉树： 编号为 i 的元素存储在数组下标为 i−1 的分量中；

**一般二叉树：** 对照完全二叉树，存储在数组的相应分量中；




(a) 单支二叉树

| A | ∧ | B | ∧ | ∧ | ∧ | C | ∧ | ∧ | ∧ | ∧ | ∧ | ∧ | ∧ | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

(b) 顺序存储结构

在最坏情况下，深度为 **k** 的右单支二叉树需要 $2^k-1$ 个存储空间。

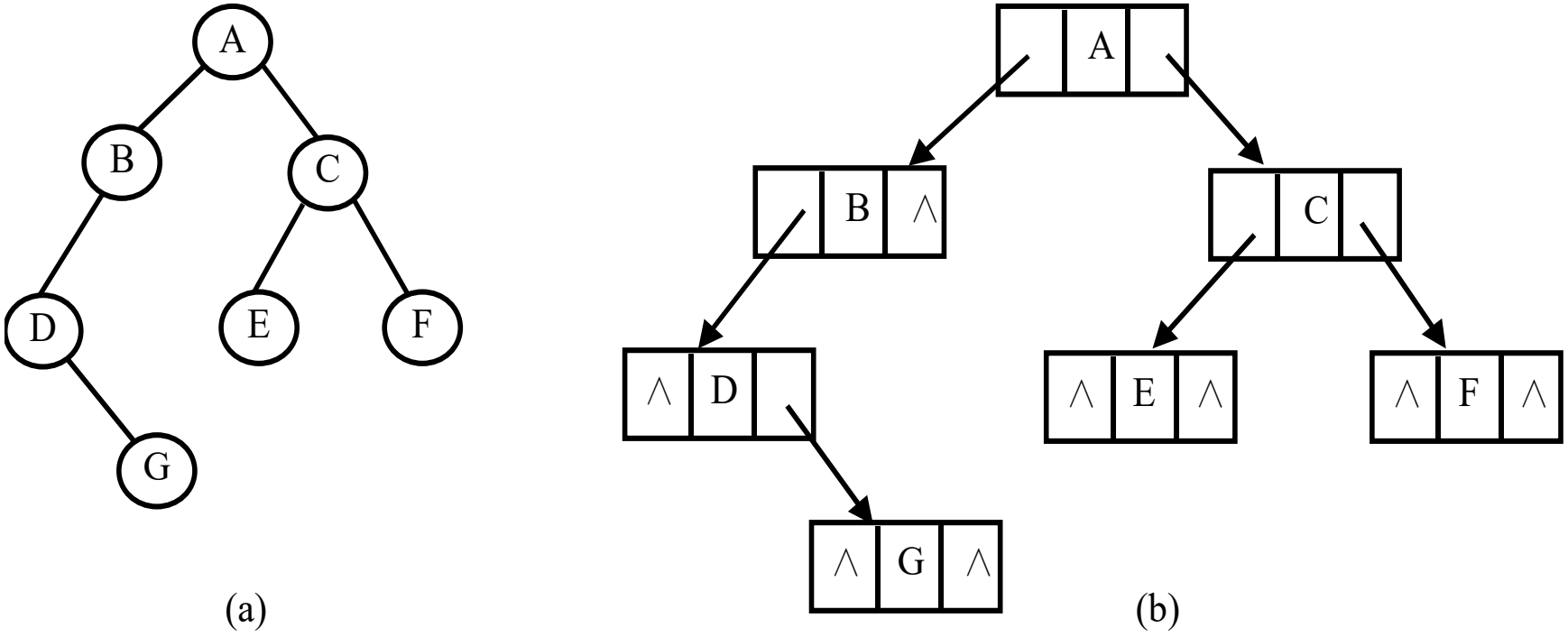
- 二叉树的链式存储结构

```
typedef struct BiTNode {
    ElemType data;
    struct BiTNode *lchild,*rchild;
} BiTNode, *BiTree;
```

data表示值域,用于存储对应的数据元素,

lchild和rchild分别表示左指针域和右指针域,用于分别存储左孩子结点和右孩子结点(即左、右子树的根结点)的存储位置。

- 二叉树及其链式存储结构




**n＋1个空链域，分支数目B=n-1，**
**即非空的链域有n-1个，故空链域有2n-(n-1)=n+1个。**

- 作业：

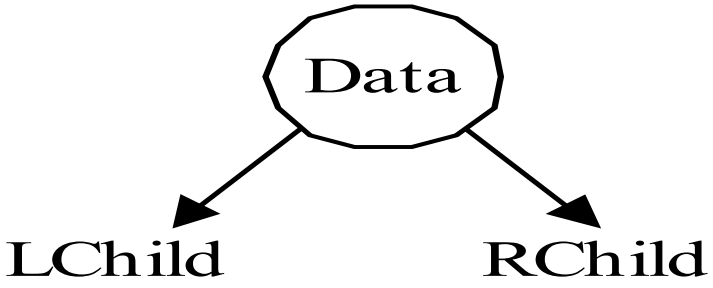
  6.2　6.3　6.5

  6.6　6.8

# 6.3 遍历二叉树和线索二叉树

## 1. 二叉树遍历的概念

二叉树的遍历是指按照一定次序访问树中所有结点, 并且每个结点仅被访问一次的过程。它是最基本的运算, 是二叉树中所有其他运算的基础。

用L、D、R分别表示遍历左子树、访问根结点、 遍历右子树，二叉树的遍历顺序就可以有六种方式：

(1) 访问根，遍历左子树，遍历右子树（记做DLR）。

(2) 访问根，遍历右子树，遍历左子树（记做DRL）。

(3) 遍历左子树，访问根，遍历右子树（记做LDR）。

(4) 遍历左子树，遍历右子树，访问根（记做LRD）。

(5) 遍历右子树，访问根，遍历左子树（记做RDL）。

(6) 遍历右子树，遍历左子树，访问根（记做RLD）。

三种遍历方法的递归定义

- 先序遍历（DLR）操作过程：

若二叉树为空， 则空操作， 否则

(1) 访问根结点；

(2) 按先序遍历左子树；

(3) 按先序遍历右子树。

- 中序遍历（LDR）操作过程：

若二叉树为空，则空操作，否则：

(1) 按中序遍历左子树；

(2) 访问根结点；

(3) 按中序遍历右子树。

- 后序遍历（LRD）操作过程：

  若二叉树为空， 则空操作， 否则：

  (1) 按后序遍历左子树；

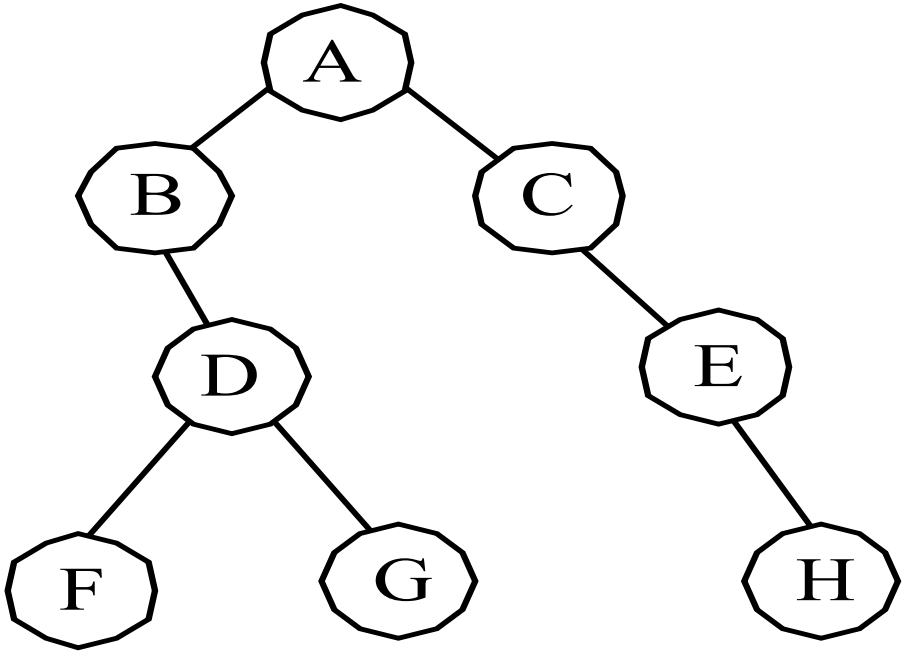
  (2) 按后序遍历右子树；

  (3) 访问根结点。

先序遍历： A、 B、 D、 F、 G、 C、 E、 H 。

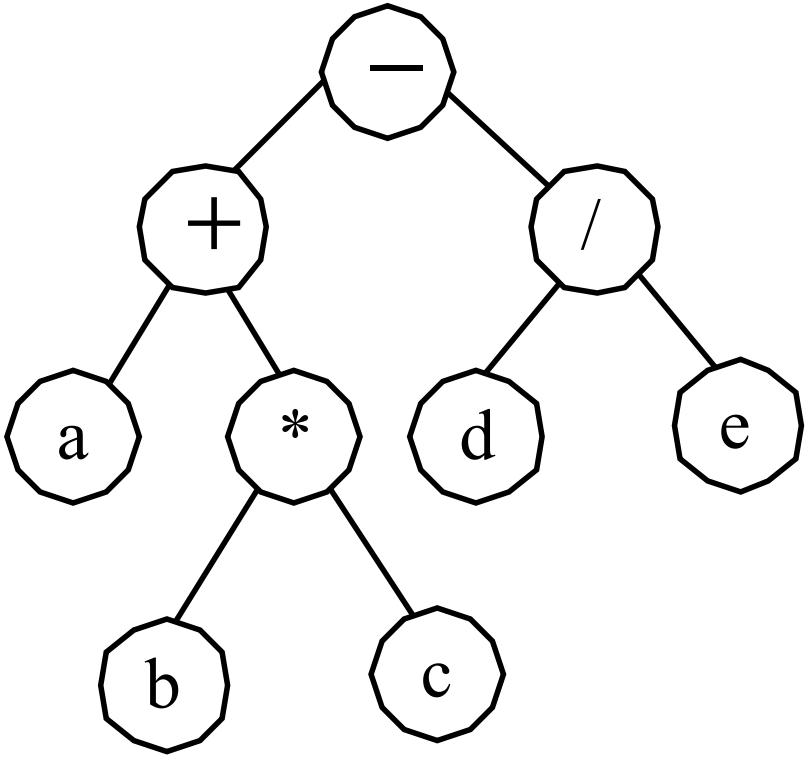
中序遍历： B、 F、 D、 G、 A、 C、 E、 H 。

后序遍历： F、 G、 D、 B、 H、 E、 C、 A 。



- 算术式的二叉树表示

**a+b*c-d/e**




**前缀：　-+a*bc/de（波兰表达式）**
**中缀：　a+b*c-d/e**
**后缀：　abc*+de/-（逆波兰表达式）**

## 2. 二叉树遍历递归算法

- 先序遍历的递归算法

```
void PreOrderTraverse(BiTree  T)
{
    if (T!=NULL){
        printf("%c ",T->data);  /*访问根结点*/
        PreOrderTraverse(T->lchild);
        PreOrderTraverse(T->rchild);
    }
 }
```

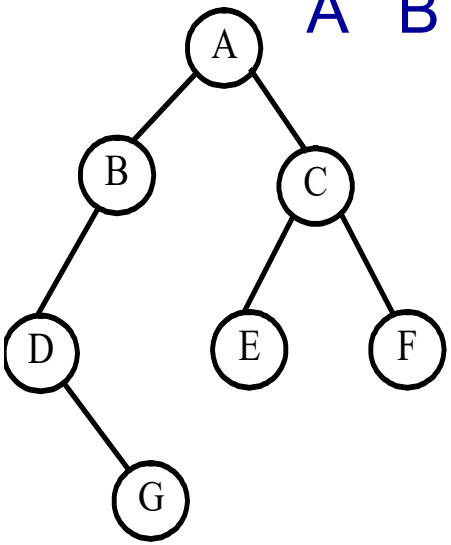
```
void PreOrderTraverse(BiTree  T) 先序：第一次遇到就访问
{
    if (T!=NULL){
        printf("%c ",T->data);  //访问根结点
        PreOrderTraverse(T->lchild); //先序遍历根的左子树
        PreOrderTraverse(T->rchild); //先序遍历根的右子树
    }
}
```

递归调用 →

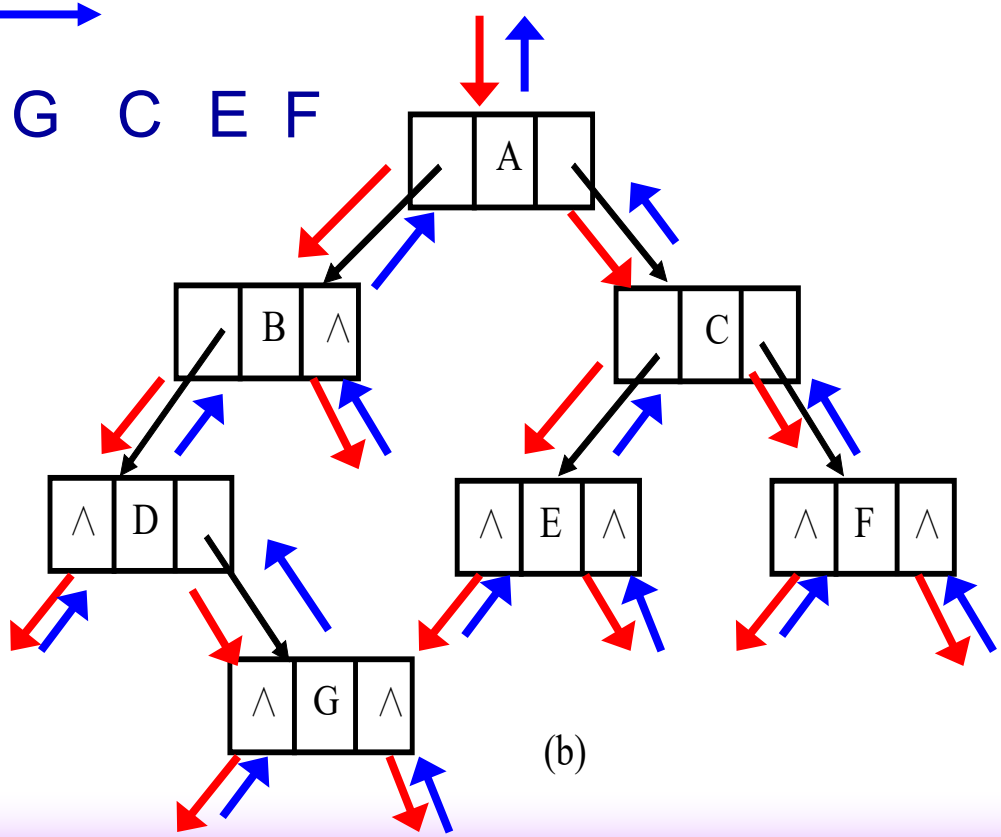
返回 ←

■栈用于保存当前结点的祖先结点

A B D G C E F




(a)




(b)

- 中序遍历的递归算法

```
void InOrderTraverse(BiTree T)
 {
        if (T!=NULL)
        {
            InOrderTraverse(T->lchild);
            printf("%c ",T->data); /*访问根结点*/
            InOrderTraverse(T->rchild);
        }
 }
```

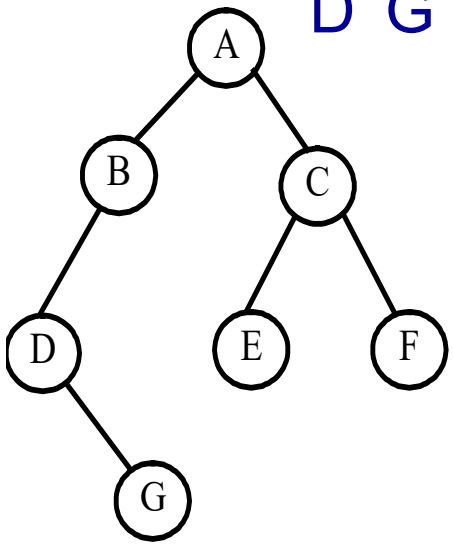
```
void InOrderTraverse(BiTree T)中序：第二次遇到就访问
{
    if (T!=NULL) {
        InOrderTraverse(T->lchild);
        printf("%c ",T->data); /*访问根结点*/
        InOrderTraverse(T->rchild);
    }
}
```

递归调用 →

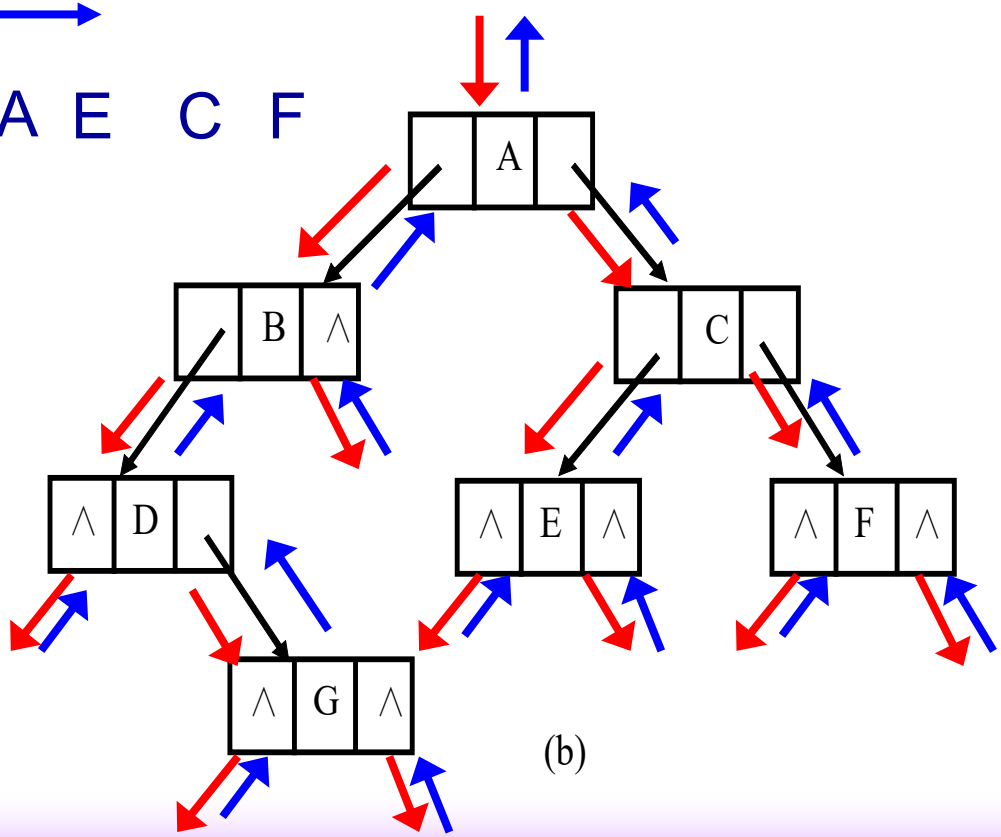
返回 ←

■栈用于保存当前结点的祖先结点

D G B A E C F



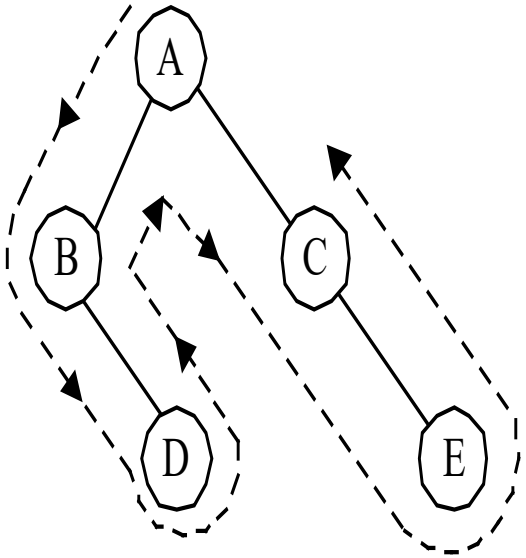

(a)



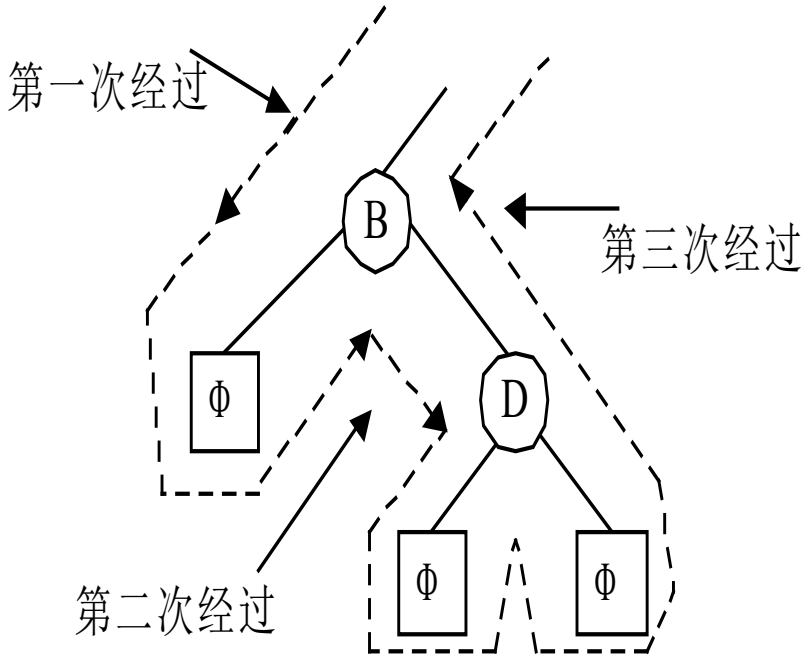

(b)

- 后序遍历递归算法

```
void PostOrderTraverse(BiTree T) {
        if (T!=NULL)
        {
            PostOrderTraverse(T->lchild);
            PostOrderTraverse(T->rchild);
            printf("%c ",T->data); /*访问根结点*/
        }
}
```




(a) 二叉树的遍历走向




(b) 遍历中三次经过结点的情形

```
void XXTraverse(BiTree T) //遍历二叉树的递归算法
{
    if (T!=NULL){
        ①
        XXTraverse(T->lchild);
        ②
        XXTraverse(T->rchild);
        ③
    }
}
printf("%c ",T->data); /*访问根结点*/
```

将递归算法转变为等价的非递归算法，应设置栈。

- 先序遍历void PreOrderTraverse(BiTree T)

```
{
    将根结点进栈;
    while(栈不空){
      出栈p;
      访问p;
      其右孩子不空时，右孩子进栈;
      其左孩子不空时，左孩子进栈;
    }
}
```

- 先序遍历void PreOrderTraverse(BiTree T)

```
{
   InitStack(S);
   Push(S,T);
   while(!StackEmpty(S)){
       Pop(S,p);
       cout<<p->data<<“”;
       if(p->rchild)   Push(S, p->rchild);
       if(p->lchild)    Push(S, p->lchild);
   }
}
```

- 中序遍历void InOrderTraverse(BiTree T)

```
{   InitStack(S);  p = T;
    while( p || !StackEmpty(S)){
        if(p){
             Push(S, p); p = p->lchild;
        }
        else{
            Pop(S, p);     cout<<p->data<<“”;
            p = p->rchild;
        }
    }
}
```

- 后序遍历void PostOrderTraverse(BiTree T)

```
typedef struct {
                BiTree  ptr;
                enum {0,1,2} mark;
} StackNode;
```

mark=0表示刚刚访问此结点,

mark=1表示左子树处理结束返回,

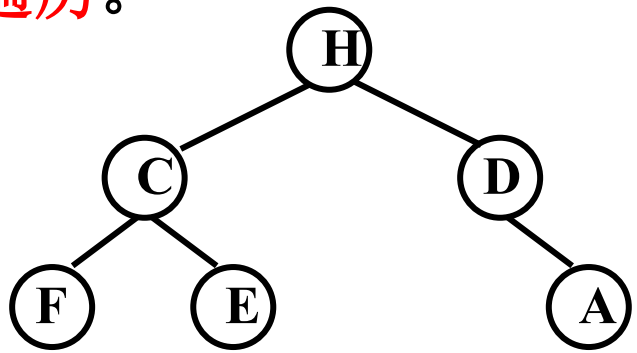
mark=2表示右子树处理结束返回.

每次根据栈顶元素的mark域值决定做何种动作.

- 后序遍历void PostOrderTraverse(BiTree T)

```
{  StackNode a;     InitStack(S);   Push (S,{T,0}); //根结点入栈
   while( !StackEmpty(S) ) {
      Pop( S, a);
      switch( a.mark ) {
         case 0:
            Push(S,{a.ptr,1});
            if(a.ptr->lchild) Push(S,{a.ptr->lchild,0});  break;
         case 1:
            Push(S,{a.ptr,2});
            if(a.ptr->rchild) Push(S,{a.ptr->rchild,0}); break;
         case 2:
            cout<<a.ptr->data<<“”;
      }//switch
   }//while
}
```

对二叉树除可以进行先序、中序、后序的遍历外，还可以进行层次遍历。




层次遍历：H，C，D，F，E，A

过程：

打印 H；

打印 H 的左儿子 C；

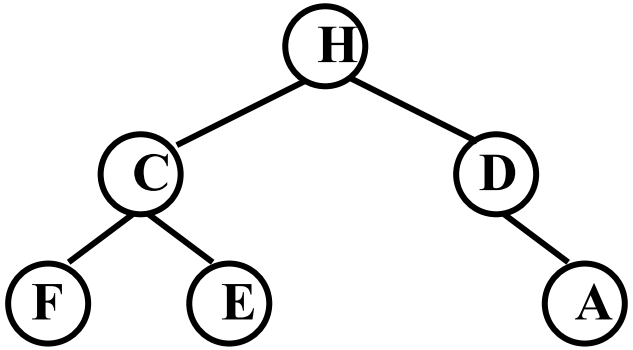
打印 H 的右儿子 D；

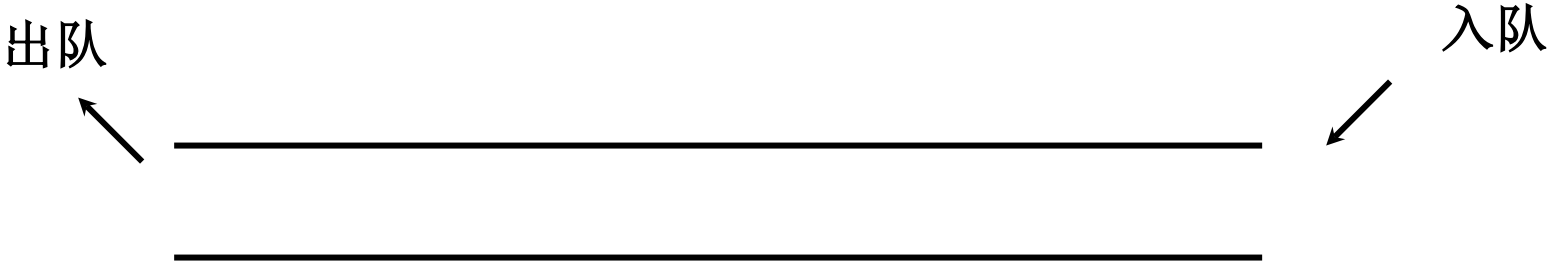
打印 C 的左、右儿子 F、E；

打印 D 的左、右儿子 A；

栈实现?

## 队列实现层次遍历



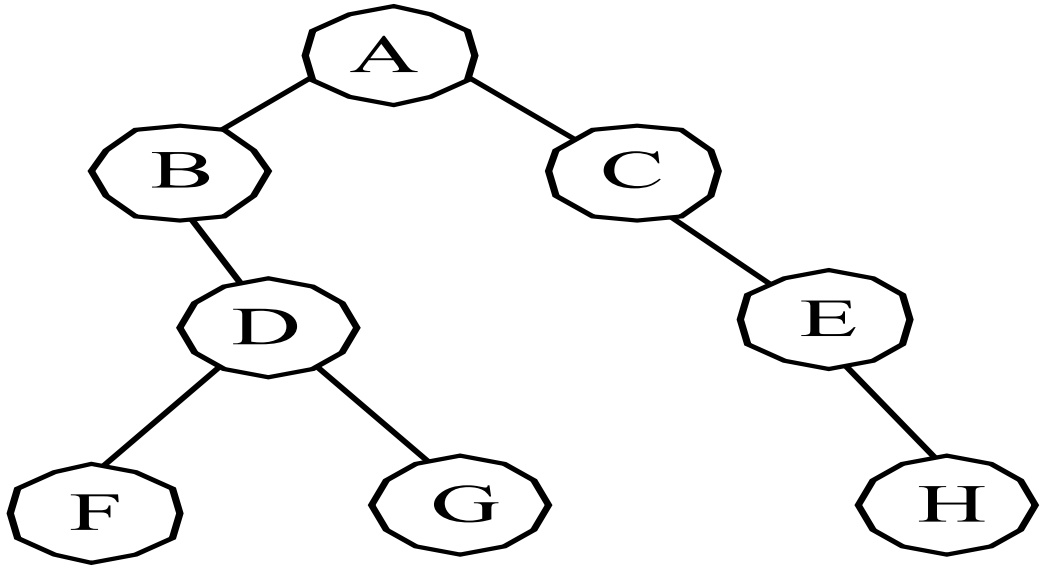




**H C D F E A**

- 作业

**6.33　　6.42**

## 2. 创建二叉链表




“扩展先序遍历序列” AB.DF..G..C.E.H..

- 利用“扩展先序遍历序列”创建二叉链表的算法如下：

```
void CreateBiTree(BiTree &T)
{
    scanf(&ch);
    if(ch == '.' ) T = NULL;
    else{
        T = (BiTree) malloc(sizeof(BiTNode));
        T->data=ch;
        CreateBiTree(T->lchild);
        CreateBiTree(T->rchild);
    }
}
```

## 3. 线索二叉树

有2n-(n-1)=n+1个空链域

一、将二叉树遍历一遍，在遍历过程中便可得到结点的前驱和后继，但这种动态访问浪费时间；

二、充分利用二叉链表中的空链域， 将遍历过程中结点的前驱、 后继信息保存下来。

$$LTag=\begin{cases}0 & \text{lchild域指示结点的左孩子}\\ 1 & \text{lchild域指示结点的遍历前驱}\end{cases}$$

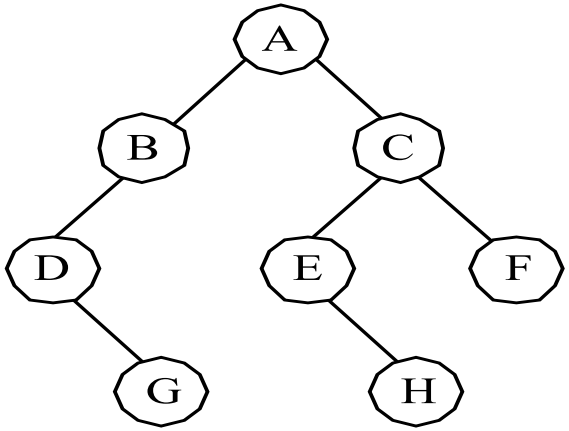
$$RTag=\begin{cases}0 & \text{rchild域指示结点的右孩子}\\ 1 & \text{rchild域指示结点的遍历后继}\end{cases}$$

| lchild | Ltag | Data | Rtag | rchild |
|---|---|---|---|---|

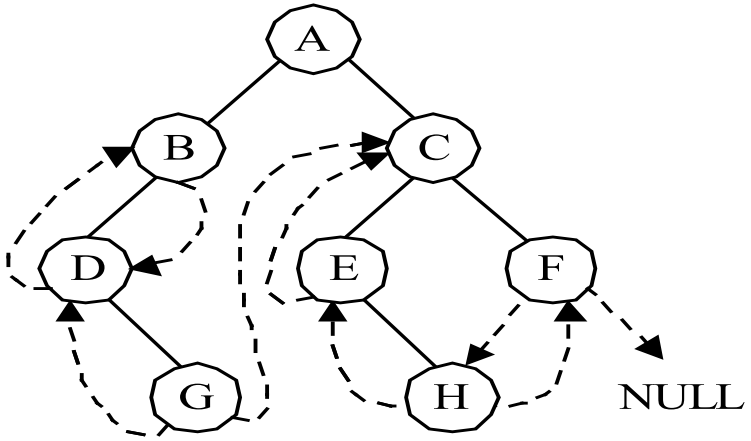
指向前驱和后继结点的指针叫做线索。 以这种结构组成的二叉链表作为二叉树的存储结构，叫做线索链表。对二叉树以某种次序进行遍历并且加上线索的过程叫做线索化。线索化了的二叉树称为线索二叉树。

```
typedef struct BiThrNode
{
        TElemType data;                 /*结点数据域*/
        int             LTag, RTag;     /*增加的线索标记*/
        struct BiThrNode  *lchild;      /*左孩子或线索指针*/
        struct BiThrNode *rchild;       /*右孩子或线索指针*/
}BiThrNode, *BiThrTree;                 /*线索树结点类型定义*/
```

ABDGCEHF

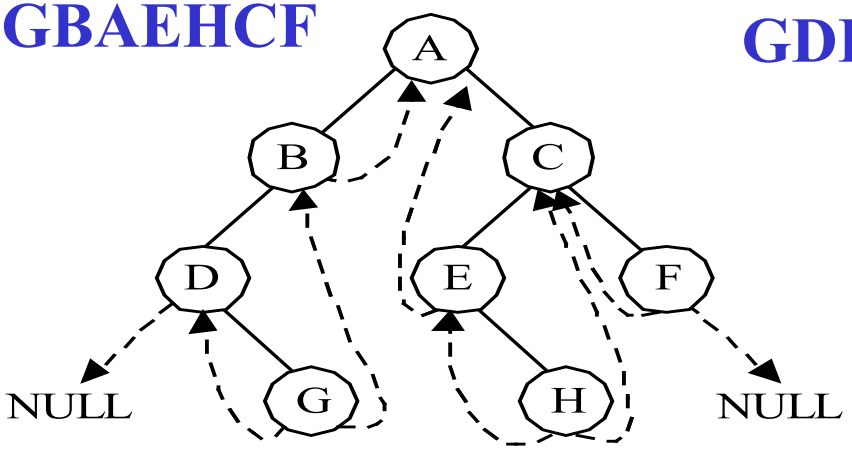



(a) 二叉树



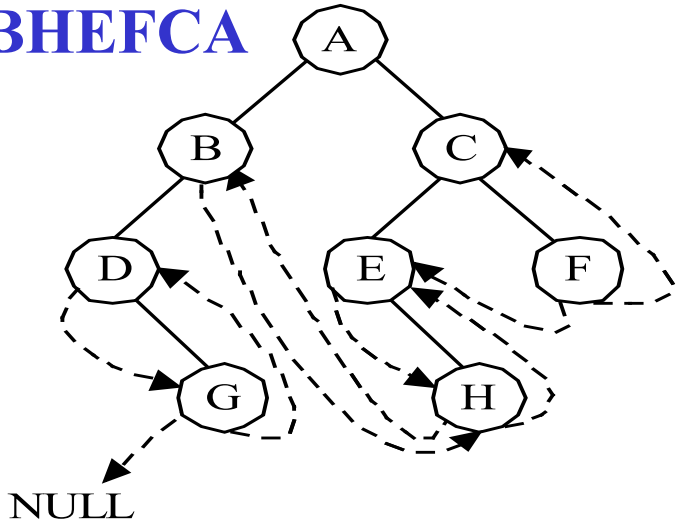

(b )先序线索二叉树

DGBAEHCF




(c) 中序线索二叉树

GDBHEFCA




(d) 后序线索二叉树

●访问线索二叉树

(1)中序线索二叉树中，查找指定结点的前驱和后继

①找结点的中序前驱结点

结点p，无左孩子，左指针指向其前驱，否则p的左子树“最右下”结点。

```
BiThrTree PreNode(BiThrTree p)
{
    pre = p->lchild;
    if(p->LTag == 0) //有左孩子
        while(pre->RTag == 0)
            pre = pre->rchild;
    return pre;
}
```

②找结点的中序后继结点

结点p，无右孩子，右指针指向其后继，否则p的右子树中“最左下”结点。

```
BiThrTree PostNode(BiThrTree p)
{
    post = p->rchild;
    if(p->RTag == 0) //有右孩子
        while(post->LTag == 0)
            post = post->lchild;
    return post;
}
```

(2)后序线索二叉树中，查找指定结点的前驱和后继

①找结点的后序前驱结点

```
if(p->LTag == 1) pre = p->lchild;//无左孩子
if(p->LTag == 0){ //有左孩子
    if(p->RTag == 0) pre = p->rchild; //右孩子为前驱
    if(p->RTag == 1) pre = p->lchild; //左孩子为前驱
}
```

从该结点出发就能找到。

②找结点的后序后继结点 （需要知道该结点的双亲）

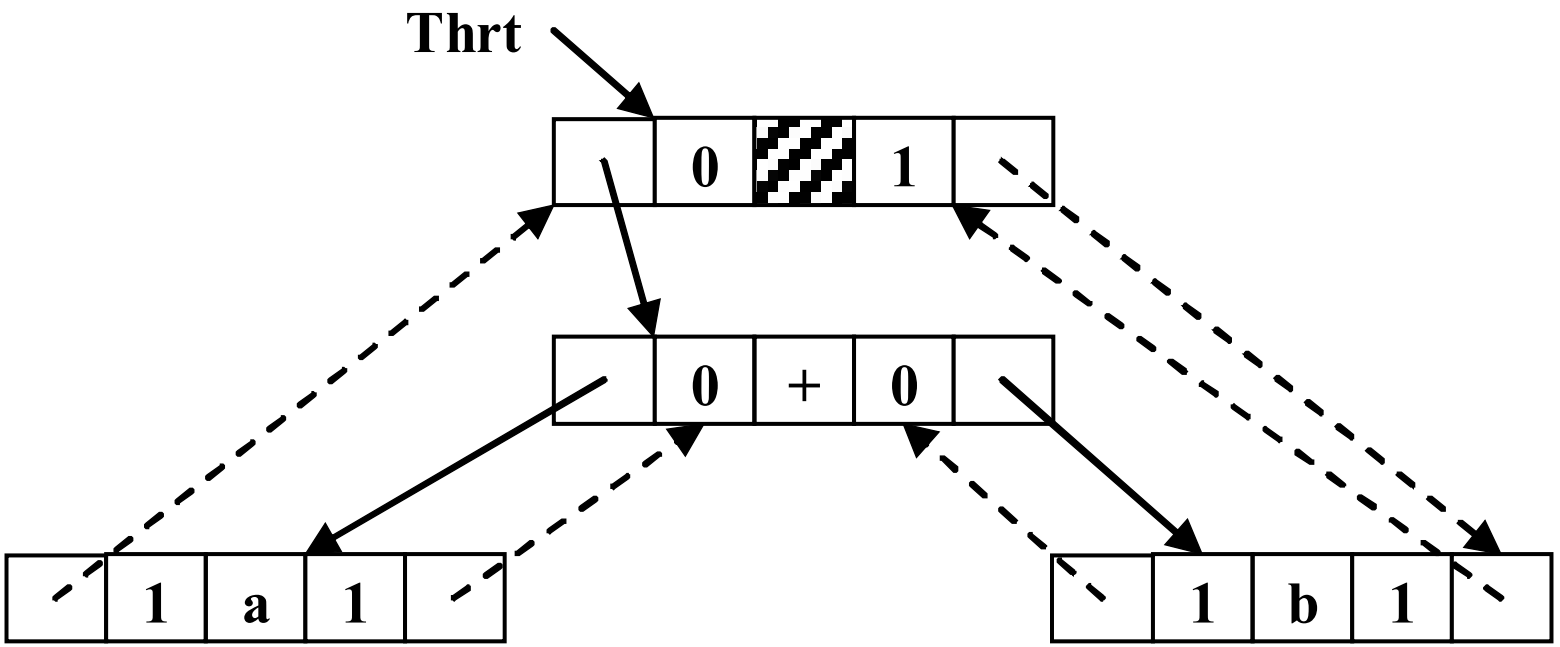
- P为根结点，
  则无后继结点；
- p无右孩子，
  If(p->RTag == 1) post = p->rchild;
- 若p是双亲的右孩子，
  则双亲为p的后继结点；
- 若p是双亲的左孩子，且p没有右兄弟，
  则双亲为其后继结点；
- 若p是双亲的左孩子，且p有右兄弟，
  则p的后继是其双亲右子树中第一个后序遍历到的结点。

(3)先序线索二叉树中，查找指定结点的前驱和后继

①找结点的先序前驱结点
需知道该结点的双亲；

②找结点的先序后继结点
从该结点出发就能找到。

- 带头结点的线索二叉链表



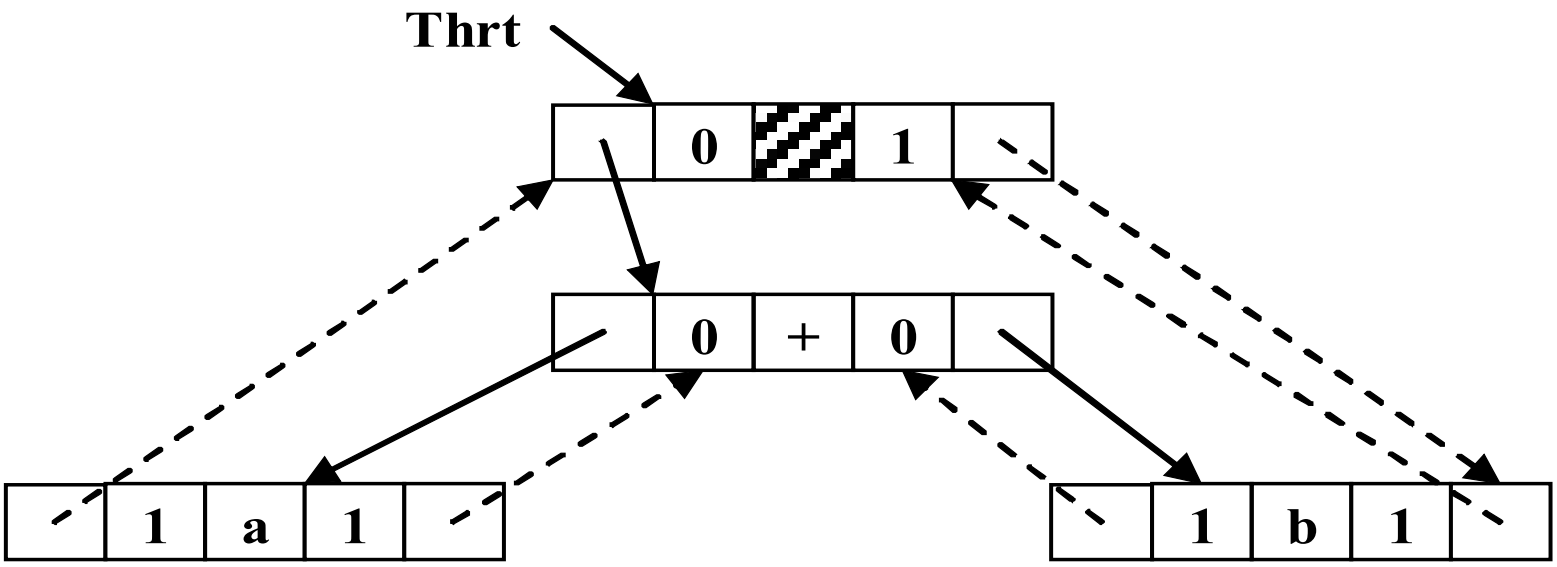

带头结点的中序线索二叉链表

- 中序遍历线索二叉树 //0:有孩子；1：无孩子

```
Void InOrderTraverse_Thr(BiThrTree  T)//T:头结点
{
   p = T->lchild;
   while(p != T){
       while(p->LTag == 0) p=p->lchild;
       cout<<p->data<<“”;
       while((p->RTag==1)&&(p->rchild!=T))
       {   p = p->rchild;
           cout<<p->data<<“”;
       }
       p = p->rchild;
   }
}
```

- 建立线索化链表（以中序为例）

  按某种次序将二叉链表线索化，实质是在遍历过程中用线索取代空指针。



```
Status InOrderThreading(BiThrTree &Thrt, BiThrTree T)
{
    Thrt = (BiThrTree)malloc(sizeof(BiThrNode)); //创建头结点
    Thrt->LTag=0;  Thrt->RTag=1;   Thrt->rchild=Thrt;
     if (!T)   Thrt->lchild=Thrt; //空二叉树
    else {
        Thrt->lchild = T;
         pre = Thrt;   //pre: 刚刚访问过的结点；
         InThreading(T);
         pre->rchild=Thrt;
         pre->RTag=1;
         Thrt->rchild=pre;
    }
    return OK;
}
```

```
BiThrTree   pre = Thrt; /*全局变量, 刚刚访问过的结点*/
void InThreading(BiThrTree p)
{
   ①if (p) {
     ②InThreading(p->lchild);   /*左子树线索化*/
        if (!p->lchild) {  /*前驱线索*/
            p->lchild=pre;   p->LTag=1;
        }   else  p->LTag=0;
   ③    if (!pre->rchild) { /*后继线索*/
            pre->rchild=p;   pre->RTag=1;
        }   else  pre->RTag=0;
         pre = p;
     ④InThreading(p->rchild);  /*右子树线索化*/
   ⑤}
⑥}
```

- 对线索树进行遍历，显然其效率要比传统方式高些。如果程序中经常要进行二叉树的遍历或者需要查找在遍历过程中的前驱和后继，应当采用线索链表表示。

- 作业：

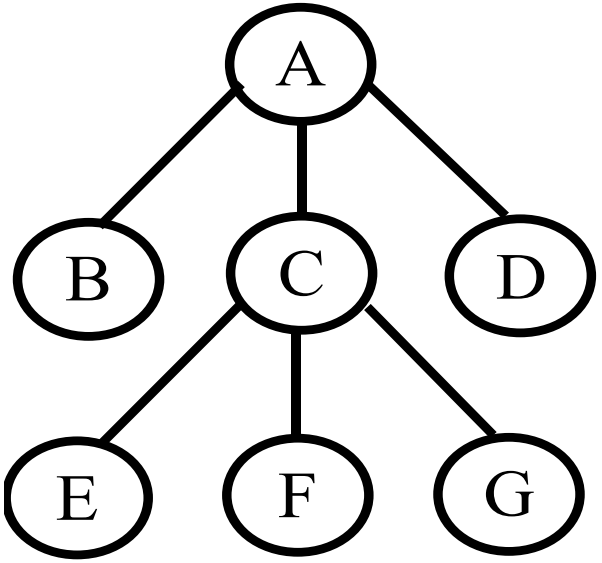
  **6.15**

# 6.4 树和森林

## 1. 树的存储结构（三种方法）

●双亲表示法：用一组连续的空间来存储树中的结点，在保存每个结点的同时附设一个指示器来指示其双亲结点在表中的位置， 其结点的结构如下：

| Data | Parent |
|---|---|




(a)

| | | |
|---|---|---|
| 0 | A | −1 |
| 1 | B | 0 |
| 2 | C | 0 |
| 3 | D | 0 |
| 4 | E | 2 |
| 5 | F | 2 |
| 6 | G | 2 |

(b)

树的双亲存储结构示意图

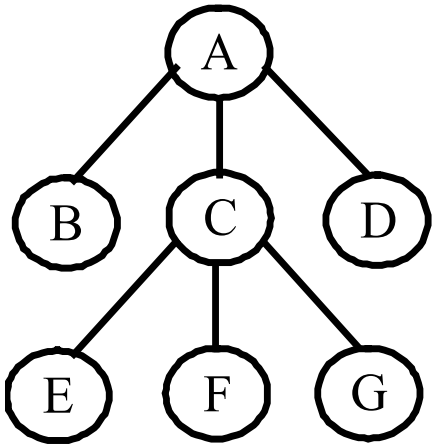
双亲表示法的类型说明：

```
#define MAX_TREE_SIZE  100
typedef struct PTNode{
    TElemType  data;
    int              parent;
}PTNode;

Typedef struct{
    PTNode  nodes[MAX_TREE_SIZE];
    int          r, n;  //根的位置和结点数
}PTree;
```

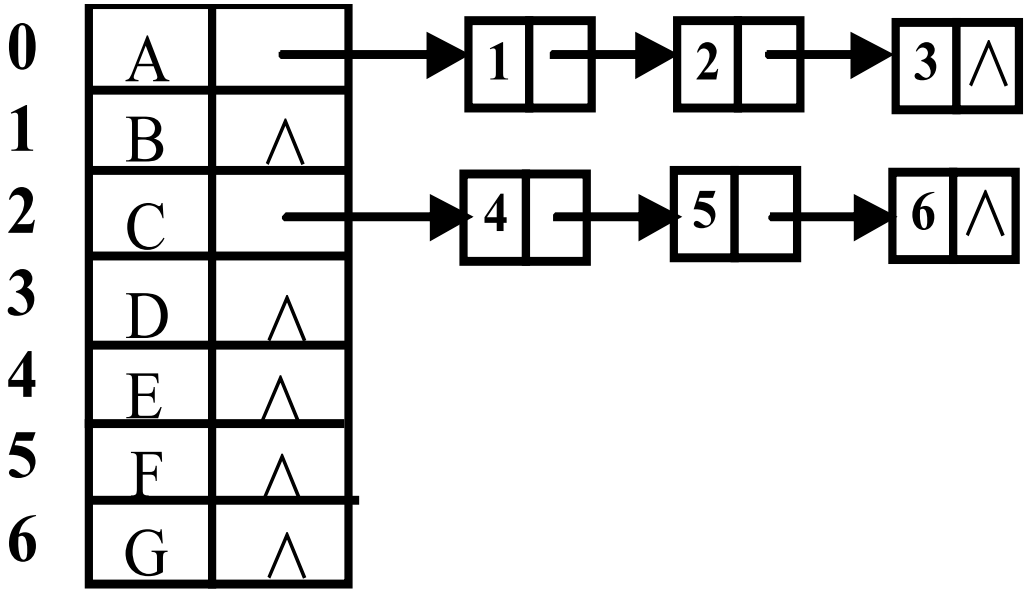
- 孩子表示法：①定长结点长度

空链域个数：nk − (n-1) = n(k-1)+1.

②把每个结点的孩子结点排列起来，构成一个单链表， 称为孩子链表。n个结点共有n个孩子链表（叶子结点的孩子链表为空表），而n个结点的数据和n个孩子链表的头指针又组成一个顺序表。




(a)



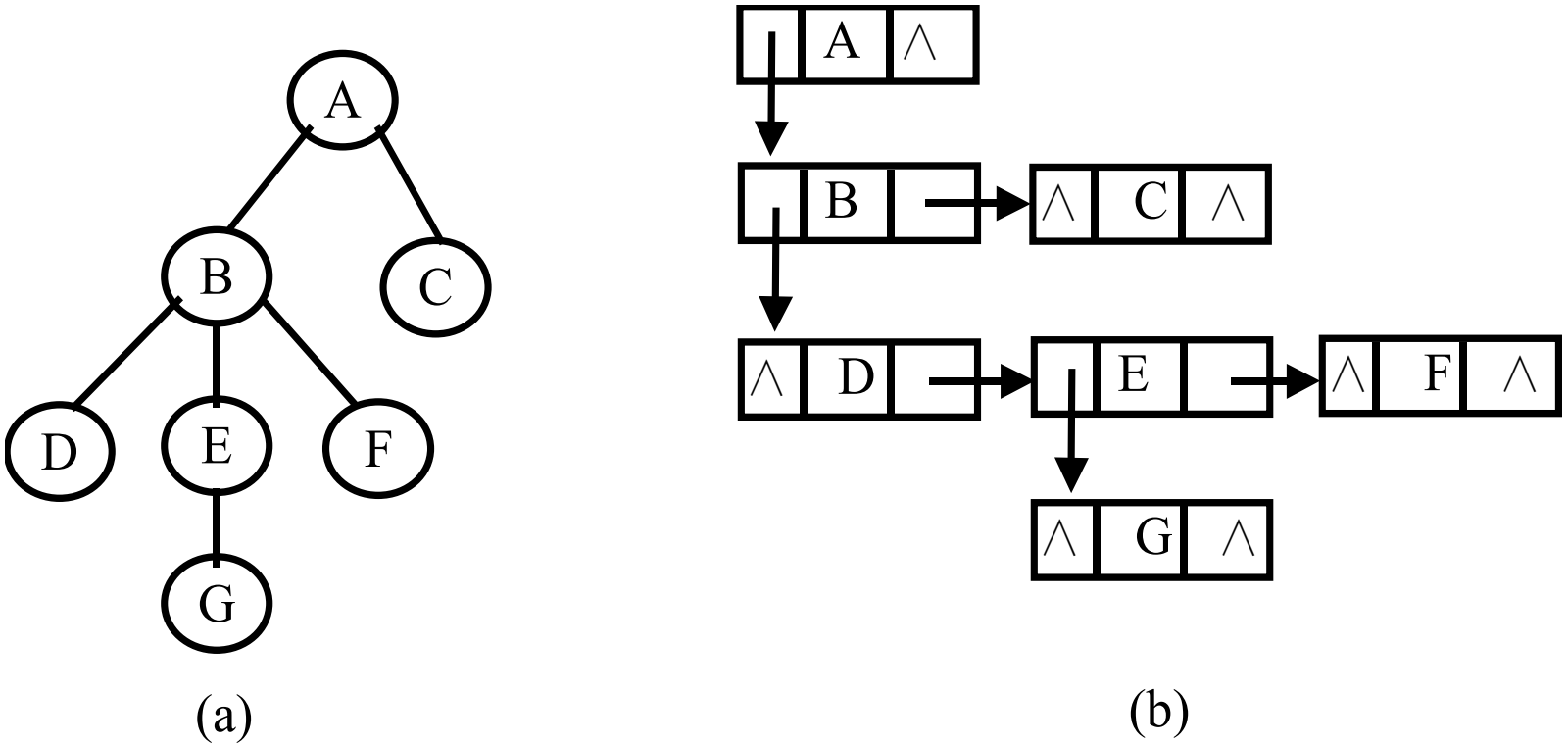

(b)

```
typedef struct CTNode { /* 孩子结点的定义 */
    int     Child;     /* 该孩子结点在线性表中的位置 */
    struct CTNode  *next;  /* 指向下一个孩子结点的指针 */
}*ChildPtr;

typedef struct { /* 顺序表结点的结构定义 */
    TElemType data;           /* 结点的信息 */
    ChildPtr    FirstChild ;  /* 指向孩子链表的头指针 */
}CTBox;

typedef struct { /* 树的定义 */
    CTBox   nodes[MAX_TREE_SIZE];  /* 顺序表 */
    int root, num; /* 根结点的位置和树的结点个数 */
} CTree;
```

- 孩子兄弟表示法

```
typedef struct CSNode {
    ElemType data;    /*结点信息*/
    Struct CSNode  *FirstChild,  *NextSibling;
                        /*第一个孩子, 下一个兄弟*/
}CSNode,  *CSTree;
```

这种存储结构便于实现树的各种操作。




树的孩子兄弟链存储结构示意图

## 2. 树、森林与二叉树的相互转换

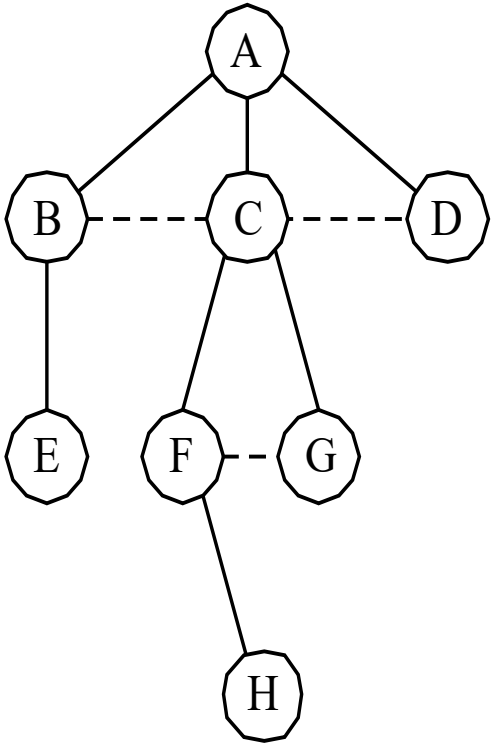
### ●树转换为二叉树

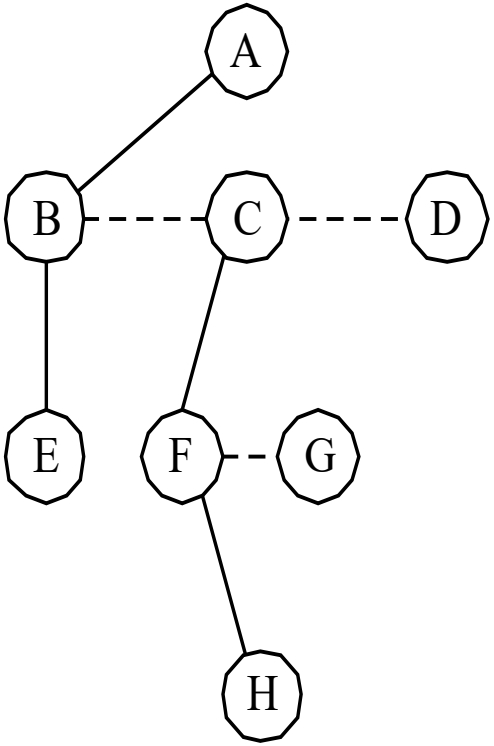
(1)在所有相邻兄弟结点之间加一水平连线。

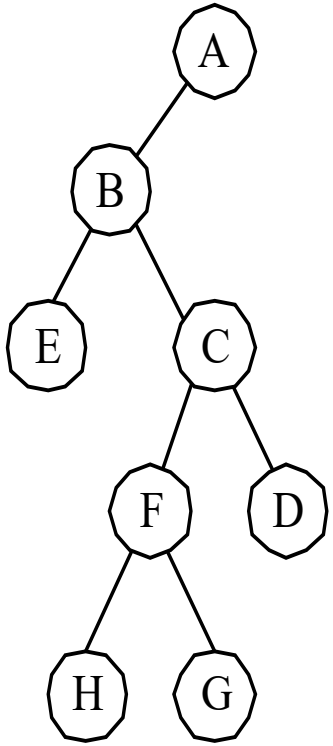
(2)对每个非叶结点k,除了其最左边的孩子结点外,删去k与其他孩子结点的连线。

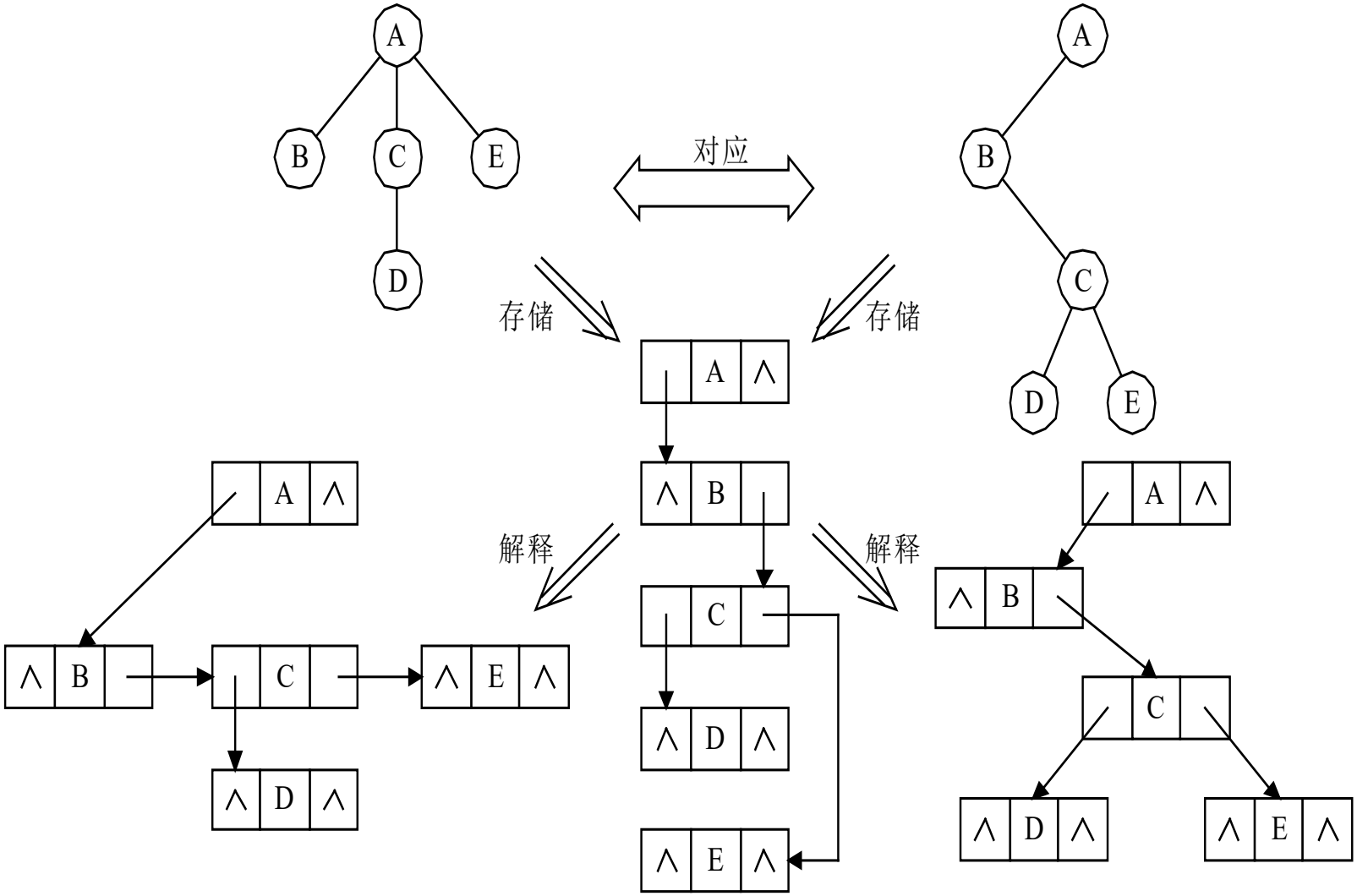
(3)所有水平线段以左边结点为轴心顺时针旋转45度,使之结构层次分明。

树做这样的转换所构成的二叉树是唯一的。


A
B
C
D
E
F
G
H


A
B
C
D
E
F
G
H


A
B
E
C
F
D
H
G



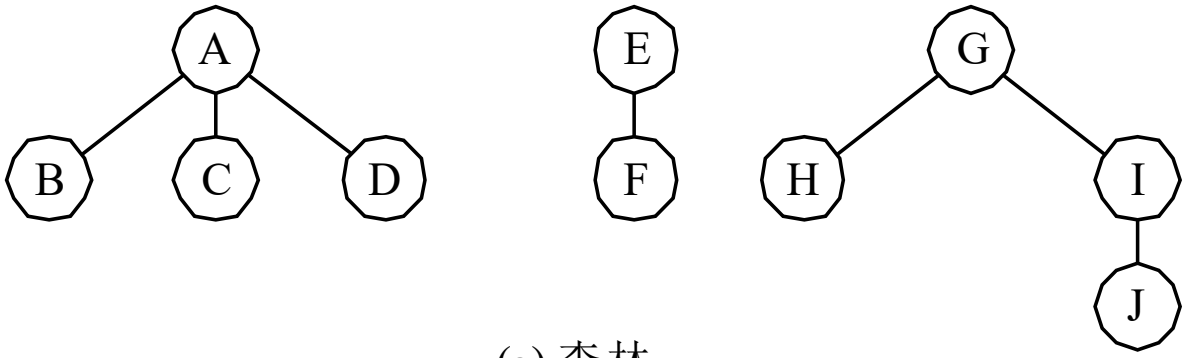

# 树与二叉树的对应

## ●森林转换为二叉树

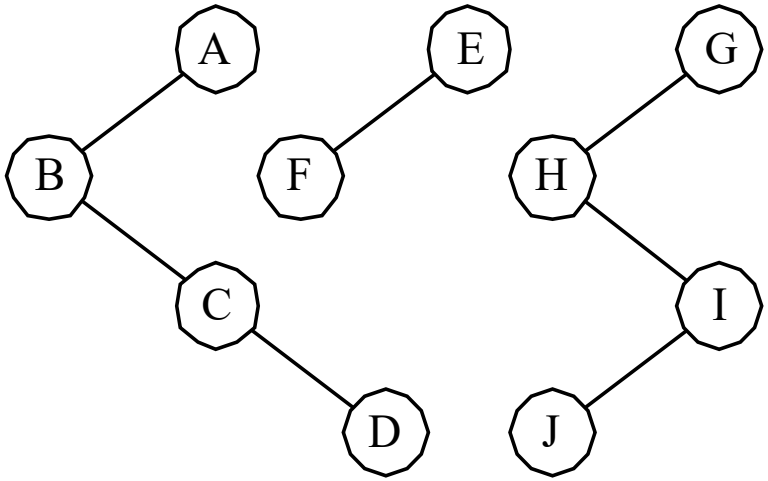
森林也可以方便地用孩子兄弟链表表示。森林转换为二叉树的方法如下：

（1）将森林中的每棵树转换成相应的二叉树。

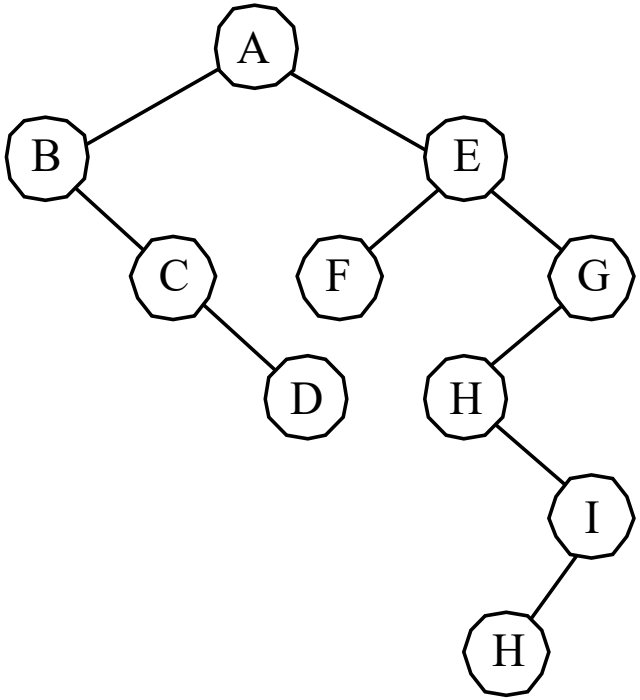
（2）第一棵二叉树不动，从第二棵二叉树开始，依次把后一棵二叉树的根结点作为前一棵二叉树根结点的右孩子， 当所有二叉树连在一起后，所得到的二叉树就是由森林转换得到的二叉树。




(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

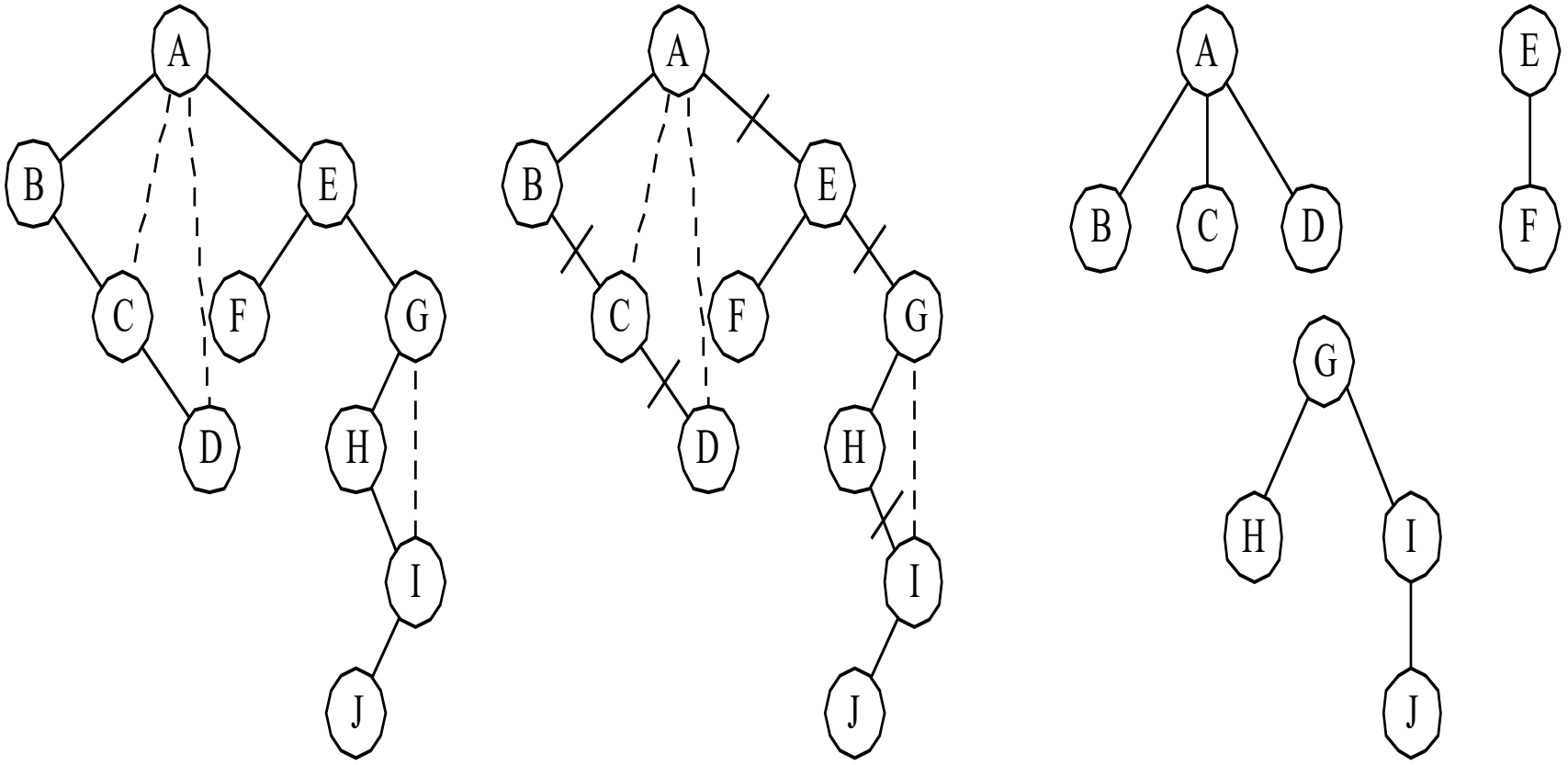
## ●二叉树还原为树或森林

将一棵二叉树还原为树或森林，具体方法如下：

（1） 若某结点是其双亲的左孩子，则把该结点的右孩子、 右孩子的右孩子.....都与该结点的双亲结点用线连起来。

（2） 删掉原二叉树中所有双亲结点与右孩子结点的连线。

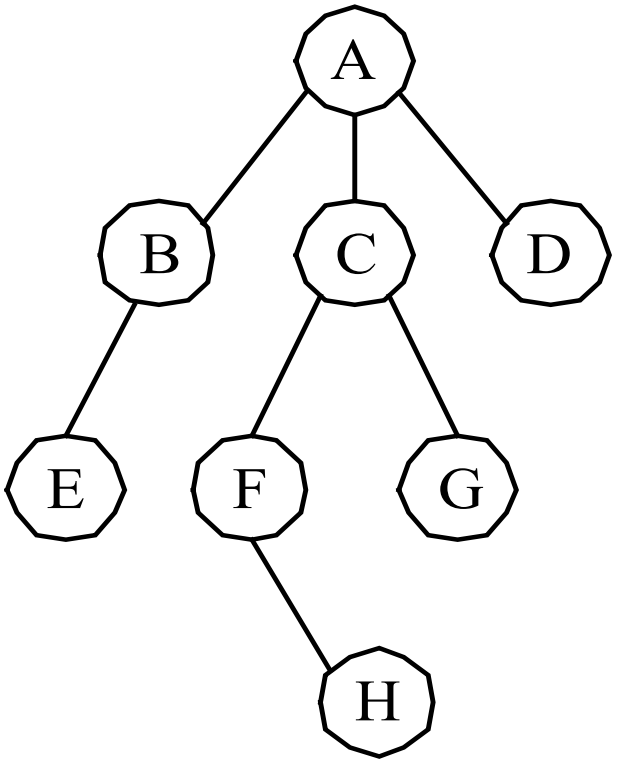
（3） 整理由（1）、（2）两步所得到的树或森林，使之结构层次分明。




(a) 添加连线　　(b) 删除右孩子结点的连线　　(c) 整理

# 二叉树到森林的转换示例

## 3. 树与森林的遍历




●树的遍历（两种）

1） 先根遍历

若树非空，则遍历方法为：

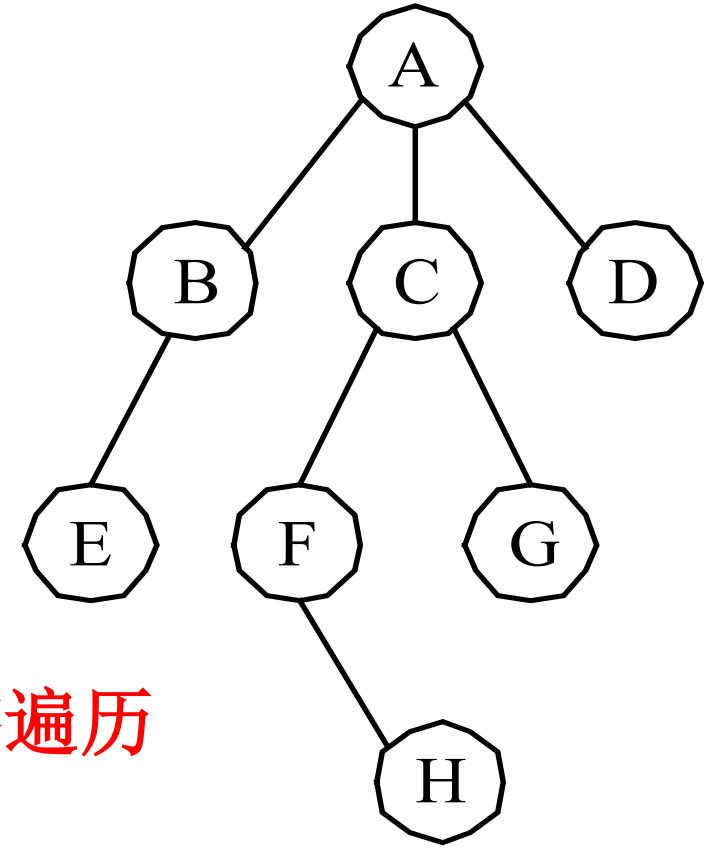
①访问根结点。

②从左到右， 依次先根遍历根结点的每一棵子树。

先根遍历序列
ABECFHGD

等同于转换的二叉树进行先序遍历

2后根遍历
若树非空， 则遍历方法为：
①从左到右， 依次后根遍历根结点的每一棵子树。
②访问根结点。

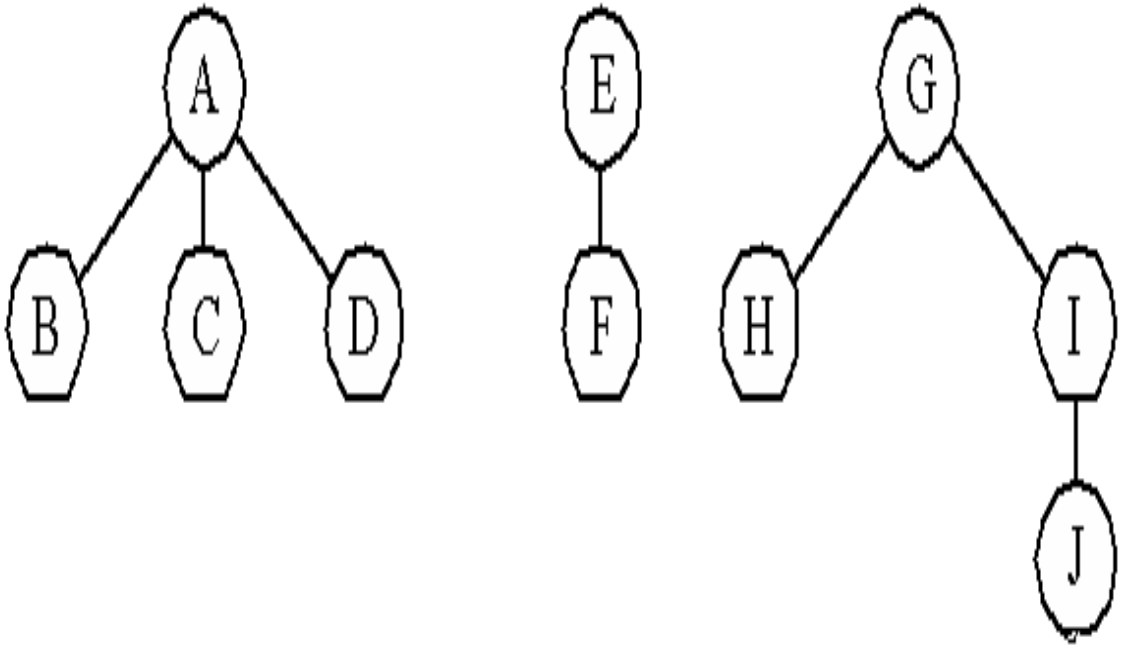



等同于转换的二叉树进行中序遍历

后根遍历序列为EBHFGCDA

●森林的遍历（2种）

1） 先序遍历
若森林非空， 则遍历方法为：
①访问森林中第一棵树的根结点。
②先序遍历第一棵树的根结点的子树森林。
③先序遍历除去第一棵树之后剩余的树构成的森林。




先序遍历序列为
**ABCDEFGHIJ**

等同于转换的二叉树进行先序遍历

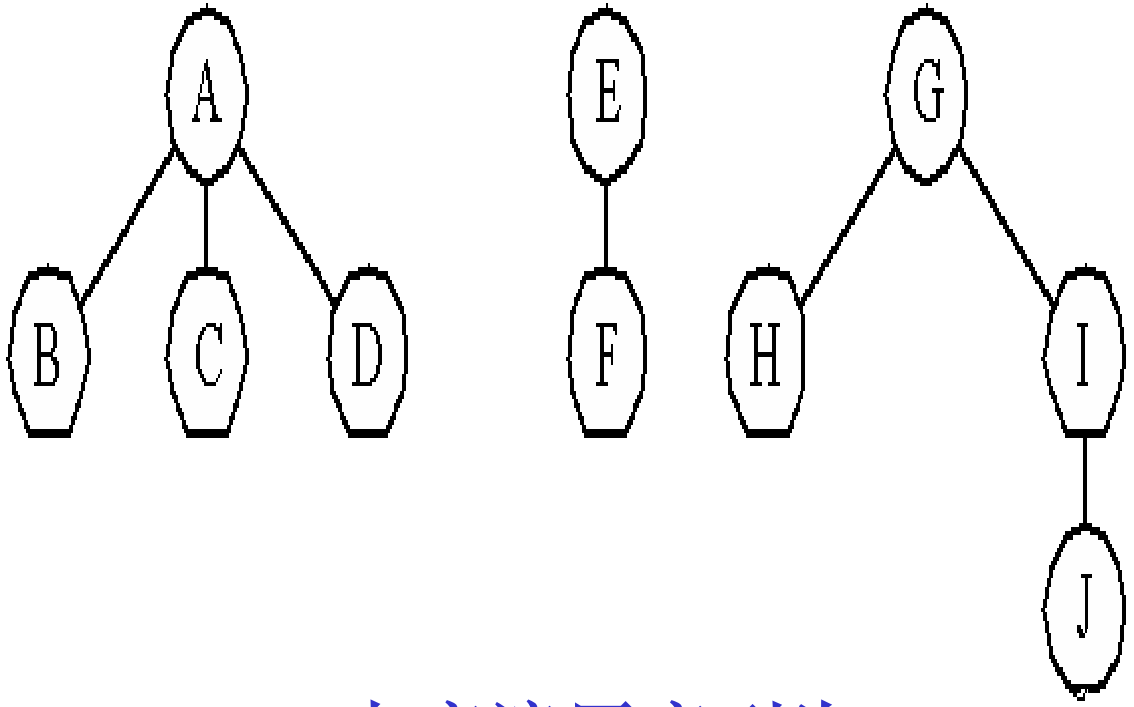
2） 中序遍历

若森林非空， 则遍历方法为：

①中序遍历森林中第一棵树的根结点的子树森林。

②访问第一棵树的根结点。

③中序遍历除去第一棵树之后剩余的树构成的森林。




中序遍历序列为
**BCDAFEHJIG**

等同于转换的二叉树进行中序遍历

4. 几个问题

①给定树的先根遍历序列和后根遍历序列可唯一画出一棵树。

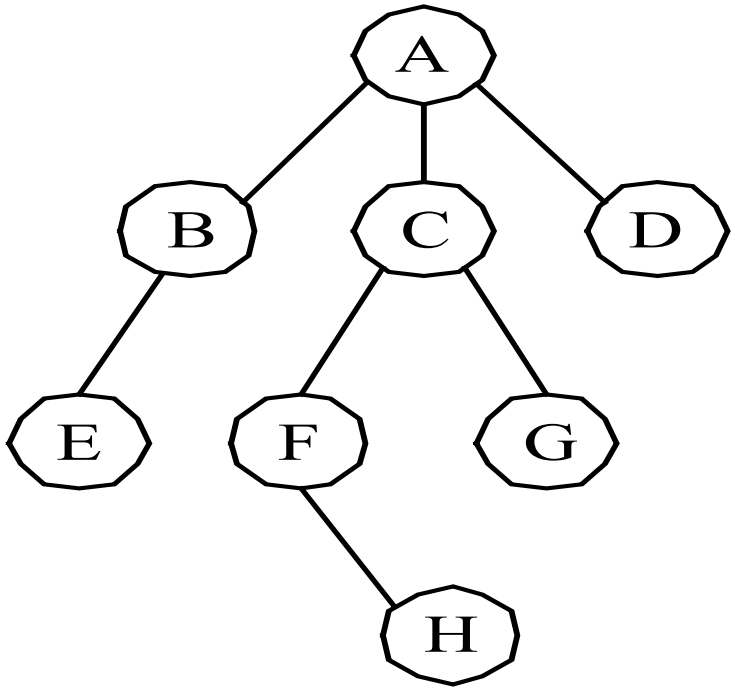
②给定森林的先序遍历序列和中序遍历序列可唯一确定一森林。

③关于二叉树的先序、中序和后序遍历序列确定二叉树的问题。

①给定树的先根遍历序列和后根遍历序列可唯一画出一棵树。

先根遍历序列：A B E C F H G D

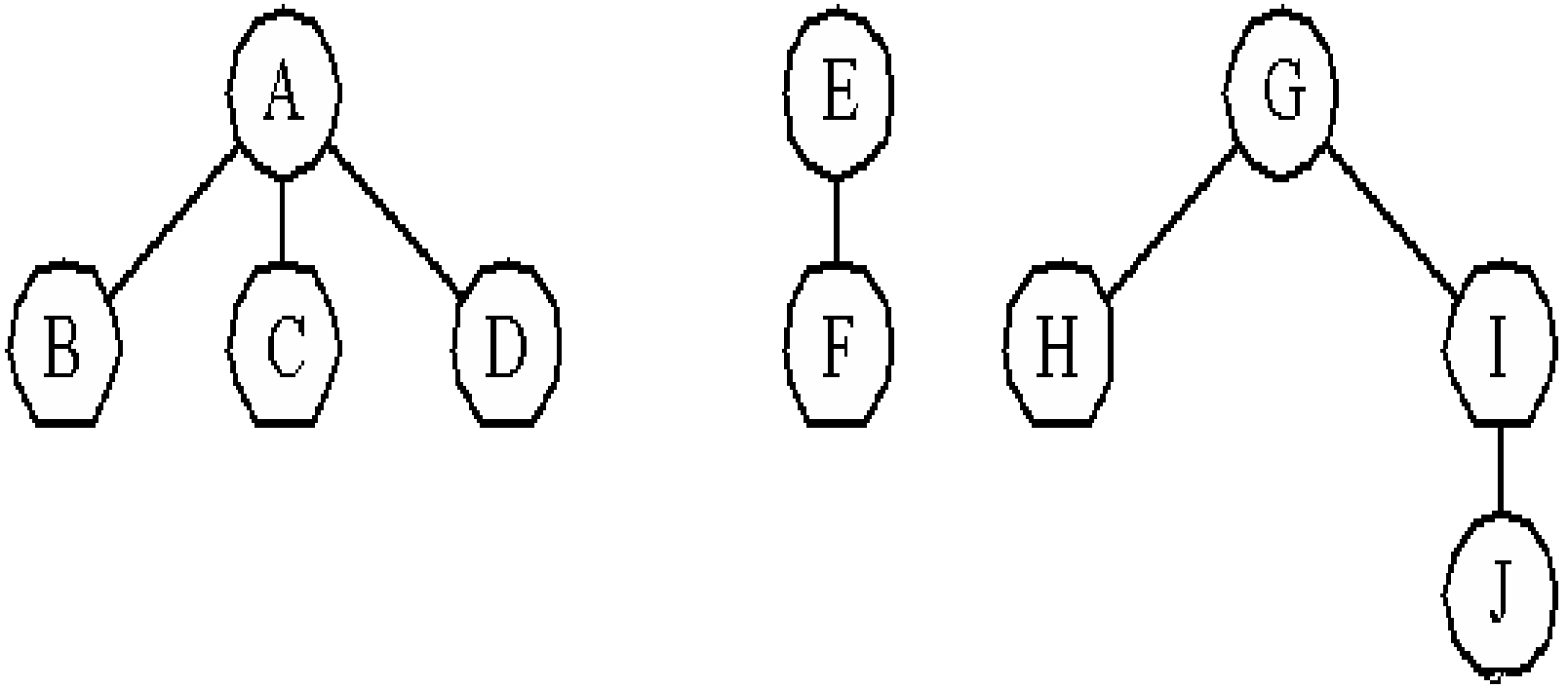
后根遍历序列：E B H F G C D A



②给定森林的先序遍历序列和中序遍历序列可唯一确定一森林。

先序遍历序列：A B C D E F G H I J

中序遍历序列：B C D A F E H J I G



③关于二叉树的先序、中序和后序遍历序列确定二叉树的问题。

●任何n(n≥0)个不同结点的二叉树,都可由它的中序序列和先序序列唯一地确定。

证明：

先序序列是$a_1a_2\ldots a_n$

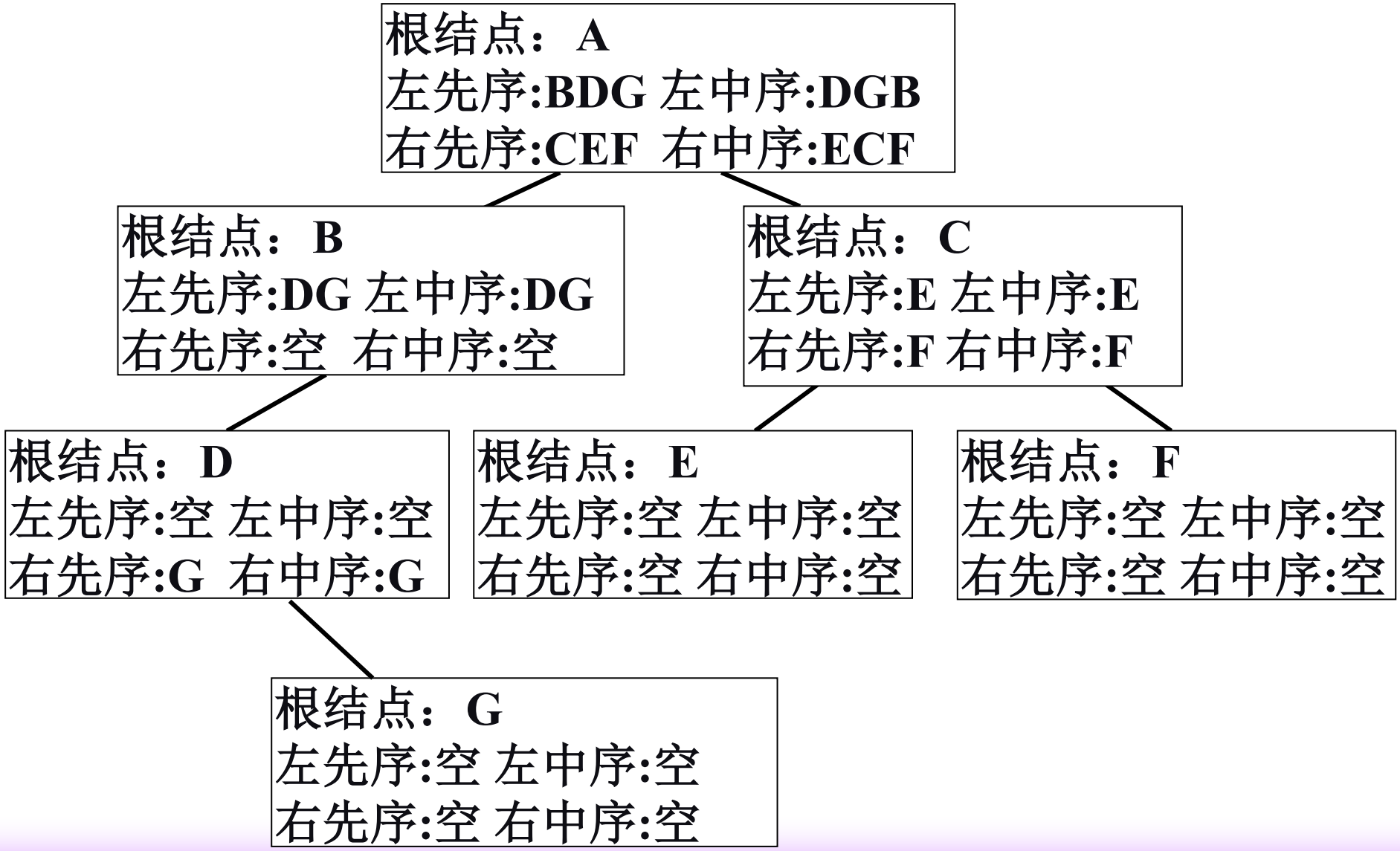
中序序列是$b_1b_2\ldots b_n$

根结点：$a_1$。

在中序序列中与a1相同的结点为：$b_j$。

$$\{b_1\ldots b_{j-1}\}b_j\{b_{j+1}\ldots b_n\} \longleftrightarrow a_1\{a_2\ldots a_k\}\{a_{k+1}\ldots a_n\}$$

例：已知先序序列为ABDGCEF，中序序列为DGBAECF



由先序、中序序列构造二叉树的算法：

```
BiTNode *CreateBT1(char *pre,char *in,int n)
{   BiTNode *s;  char *p;  int k;
    if(n==1){s=malloc(…);s->lchild=s->rchild=NULL;return s;}
    for (p=in;p<in+n;p++)  /*在中序中找子树的位置*/
        if (*p==*pre)    break;
    k = p - in;
    s=(BiTNode *)malloc(sizeof(BiTNode));
    s->data=*pre; s->lchild=s->rchild=NULL;
    if(k)  s->lchild = CreateBT1(pre+1,in,k);//左子树
    if(n-k-1) s->rchild = CreateBT1(pre+k+1,p+1,n-k-1);//右子树
    return s;
}
```

先序：ABDGCEF　　中序：DGBAECF

●任何n(n＞0)个不同结点的二叉树,都可由它的中序序列和后序序列唯一地确定。

证明：

后序序列是$a_1a_2\ldots a_n$

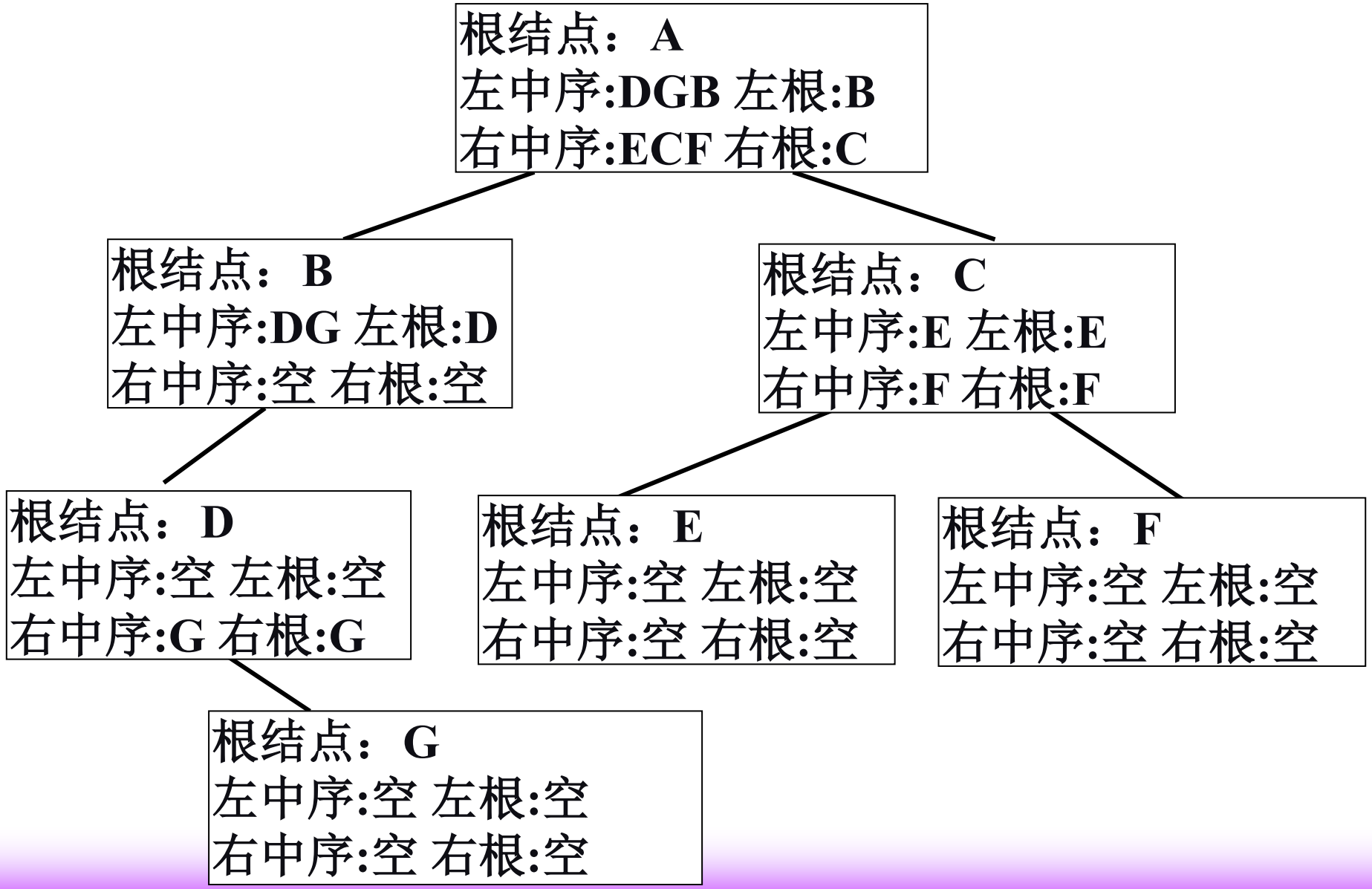
中序序列是$b_1b_2\ldots b_n$

根结点：$a_n$。

在中序序列中与$a_n$相同的结点为：$b_j$。

$\{b_1\ldots b_{j-1}\}b_j\{b_{j+1}\ldots b_n\} \longleftrightarrow \{a_1a_2\ldots a_k\}\{a_{k+1}\ldots a_{n-1}\}a_n$

例：后序序列为GDBEFCA，中序序列为DGBAECF



由后序、中序构造二叉树的算法：

```
BiTNode *CreateBT2(char *post,char *in,int n)
{   BiTNode *s; char *p; int k;
    if(n==1){s=malloc(…);s->lchild=s->rchild=NULL;return s;}
    for (p=in;p<in+n;p++)  /*在中序中找子树的位置*/
        if (*p==*(post+n-1))    break;
    k = p - in;
    s=(BiTNode *)malloc(sizeof(BiTNode)); /*创建二叉树结点s*/
    s->data=*(post+n-1); s->lchild=s->rchild=NULL;
    if(k) s->lchild=CreateBT2(post, in, k); //左子树
    if(n-k-1) s->rchild=CreateBT2(post+k, in+k+1, n-k-1);//右子树
    return s;
}
```

后序:GDBEFCA　　中序:DGBAECF

- 作业：

**6.14　6.19　6.20**

**6.23　6.24　6.28**

# 6.5 树与等价问题

## 离散数学中的定义

- 等价关系：若集合S中的关系R是自反的、对称的和传递的，则称为等价关系。
- 等价类：R是集合S的等价关系，由$[x]_R=\{y|y\in S\wedge xRy\}$给出的集合$[x]_R$称为由$x\in S$生成的一个R等价类。
- 划分：R是S上的等价关系，可以按R将S划分为若干不相交的子集$S_1, S_2, \ldots\ldots$,它们的并即为S，则这些子集Si就是S的R等价类。

## 如何划分等价类？

- 假设集合S有n个元素，m个形如(x,y)的等价偶对确定了等价关系R，求S的划分。

一种算法：

1）令S中每个元素各自形成一个只含单个成员的子集，记为$S_1$ ,$S_2$ ,…,Sn。

2) 重复读入m个偶对，对每个偶对(x,y)，判断x和y所属的子集，设$x \in S_i$，$y \in S_j$，若$S_i \neq S_j$，则将$S_j$并入$S_i$，并置$S_j$为空。

处理完m个偶对后剩下的非空子集就是S的R等价类。

# 划分等价类需要的操作

1) 构造只含单个元素的集合
2) 判定某个元素所属集合
3) 合并两个互不相交的集合

**ADT MFSet：** 若S是MFSet类型的集合，则它由子集Si构成，$S1 \cup S2 \cup \ldots \cup Sn=S$。

基本操作：

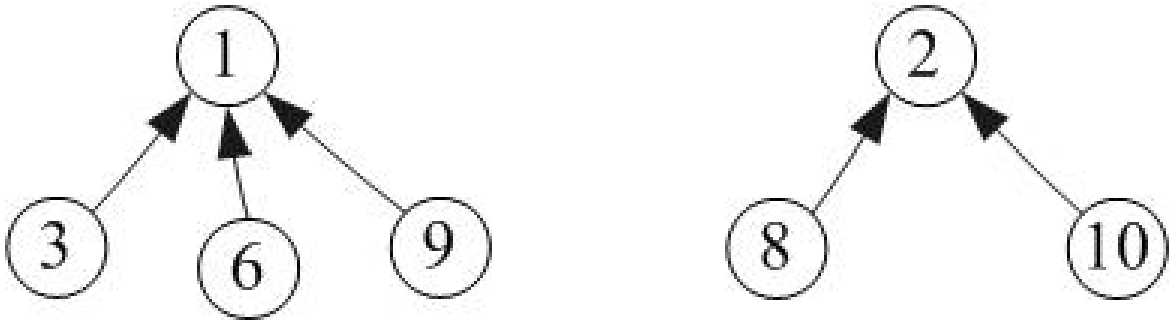
**Initial(&S,n,$x_1$,$x_2$,…,$x_n$)：** 构造由n个子集构成的集合S，每个子集只含单个元素。

**Find(S,x)：** 查找x所属的子集Si。

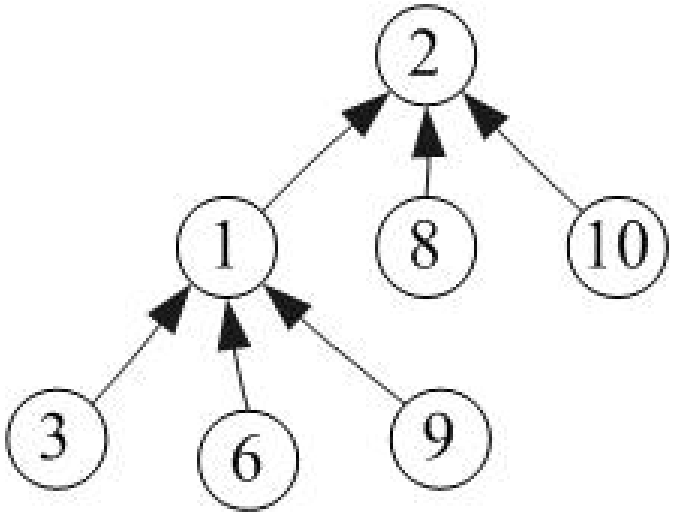
**Merge(&S,i,j)：** 合并两个不相交的集合Si和Sj。

# MFSet类型的实现

- 根据Find和Merge两个操作的特点，用树来实现MFSet。
- 以森林F=($T_1$,$T_2$,…,$T_n$)表示MFSet类型的集合S，每颗树Ti表示一个子集Si。
- 树中每个结点表示子集中的一个成员x。
- 令每个结点中包含一个指向其双亲的指针。
- 约定根结点兼作子集的名称。




集合的合并：



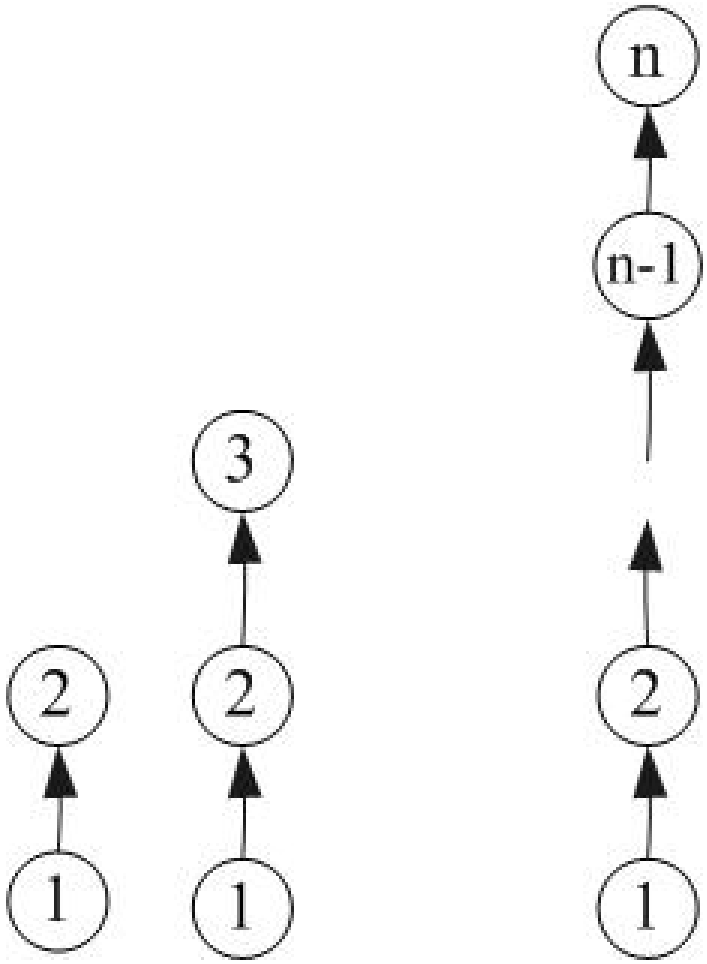

将一棵树的根指向另一颗树的根。

```
#define MAX_TREE_SIZE  100
typedef struct PTNode{
    TElemType  data;
     int              parent;
}PTNode;

Typedef struct{
    PTNode  nodes[MAX_TREE_SIZE];
    int          r, n;  //根的位置和结点数
}PTree;

Typedef PTree  MFSet;
```

```
int find_mfset(MFSet S, int i)
{
    if (i<1 || i>S.n) return -1;
    for (j=i;S.nodes[j].parent>0;j=S.node[j].parent)  ;
     return j;
}
Status merge_mfset(MFSet &S,int i,int j)
{
    if (i<1 || i>S.n || j<1 || j>S.n) return ERROR;
    S.node[i].parent=j;
    return OK;
}
```

时间复杂度分别为O(d)和O(1)，d为树的深度

极端情况：




改进方法？

- **Merge时，总是将成员少的子集根结点指向含成员多的子集的根**
- **修改存储结构，令根结点的parent域存储子集中所含成员数目的负值**
- **可以将find_mfset的复杂度降到O(logn)**

```
Status mix_mfset(MFSet &S, int i, int j)
{
    if (i<1 || i>S.n || j<1 || j>S.n) return ERROR;
    if (S.nodes[i].parent>S.nodes[j].parent) {
        S.nodes[j].parent += S.nodes[i].parent;
        S.nodes[i].parent = j;
    }else{
        S.nodes[i].parent += S.nodes[j].parent;
        S.nodes[j].parent = i;
    }
    return OK;
}
```

# 进一步的改进：Find时压缩路径

- 当所查元素不在第二层时，将所有从根到该元素路径上的元素都变成根结点的孩子

```
int fix_mfset(MFSet &S, int i)
{
    if (i<1 || i>S.n) return -1;
    for (j=i;S.nodes[j].parent>0;j=S.node[j].parent)  ;
    for (k=i;k!=j;k=t)
    {
          t=S.nodes[k].parent;  S.nodes[k].parent=j;
    }
     return j;
}
```

# 6.6 哈夫曼树及其应用

## 1. 哈夫曼树

### ①路径长度

从树中一个结点到另一个结点之间的分支构成这两个结点之间的路径， 路径上的分支数目称做路径长度。

### ②树的路径长度

从树根到每一结点的路径长度之和。

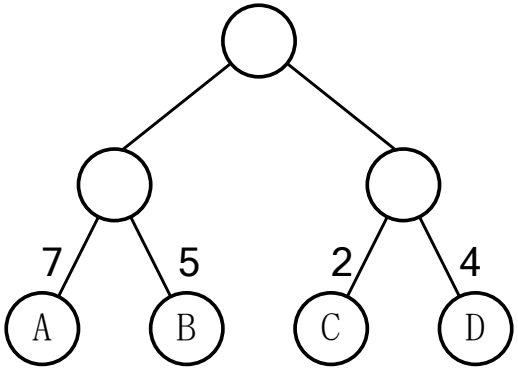
### ③结点的权和带权路径长度

给树的每个结点赋予一个具有某种实际意义的实数，我们称该实数为这个结点的权。在树形结构中，我们把从树根到某一结点的路径长度与该结点的权的乘积，叫做该结点的带权路径长度。

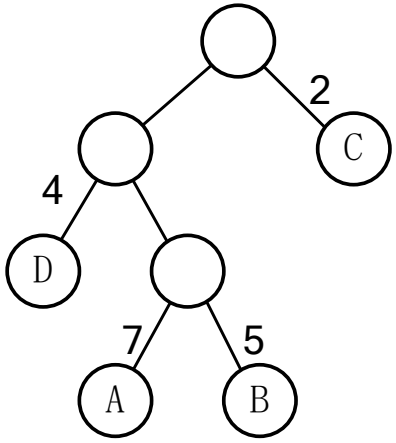
## ④树的带权路径长度WPL (Weighted Path Length of Tree)

树中所有叶子结点的带权路径长度之和，通常记为：

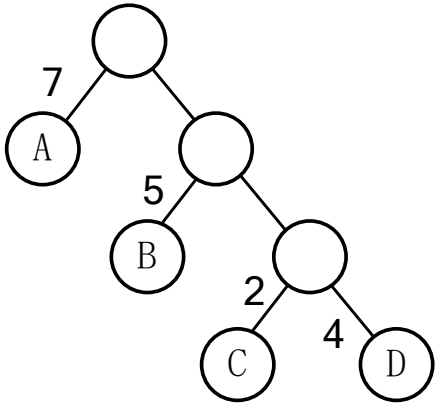
$$WPL = \sum_{k=1}^{n} w_k l_k$$




(a) 带权路径长度为36




(b) 带权路径长度为46




(c) 带权路径长度为35

WPL(a)=7×2+5×2+2×2+4×2=36

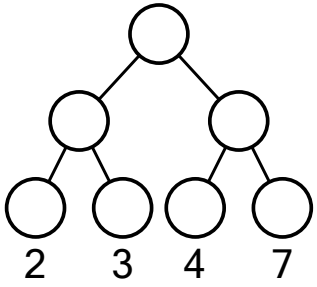
WPL(b)=4×2+7×3+5×3+2×1=46

WPL(c)=7×1+5×2+2×3+4×3=35

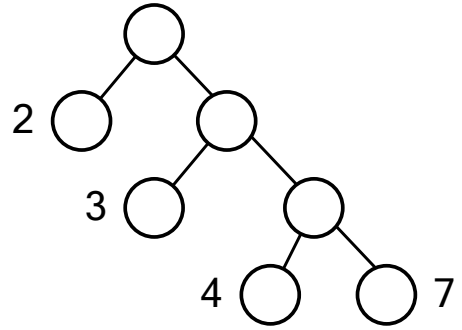
## ⑤哈夫曼树(最优二叉树)

设二叉树具有n个带权值的叶子结点,那么从根结点到各个叶子结点的路径长度与相应结点权值的乘积的和,叫做二叉树的带权路径长度。

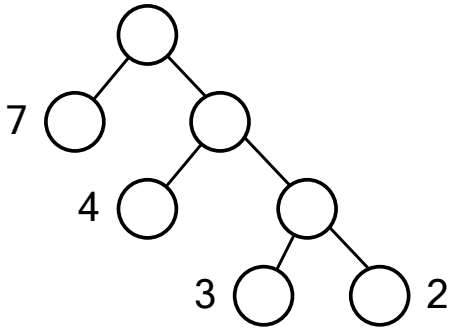
具有最小带权路径长度的二叉树称为哈夫曼树。




(a) WPL=2×2+2×3+2×4+2×7=32




(b) WPL=1×2+2×3+3×4+3×7=41




(c) WPL=1×7+2×4+3×3+3×2=30

## 2. 构造哈夫曼树(哈夫曼算法)

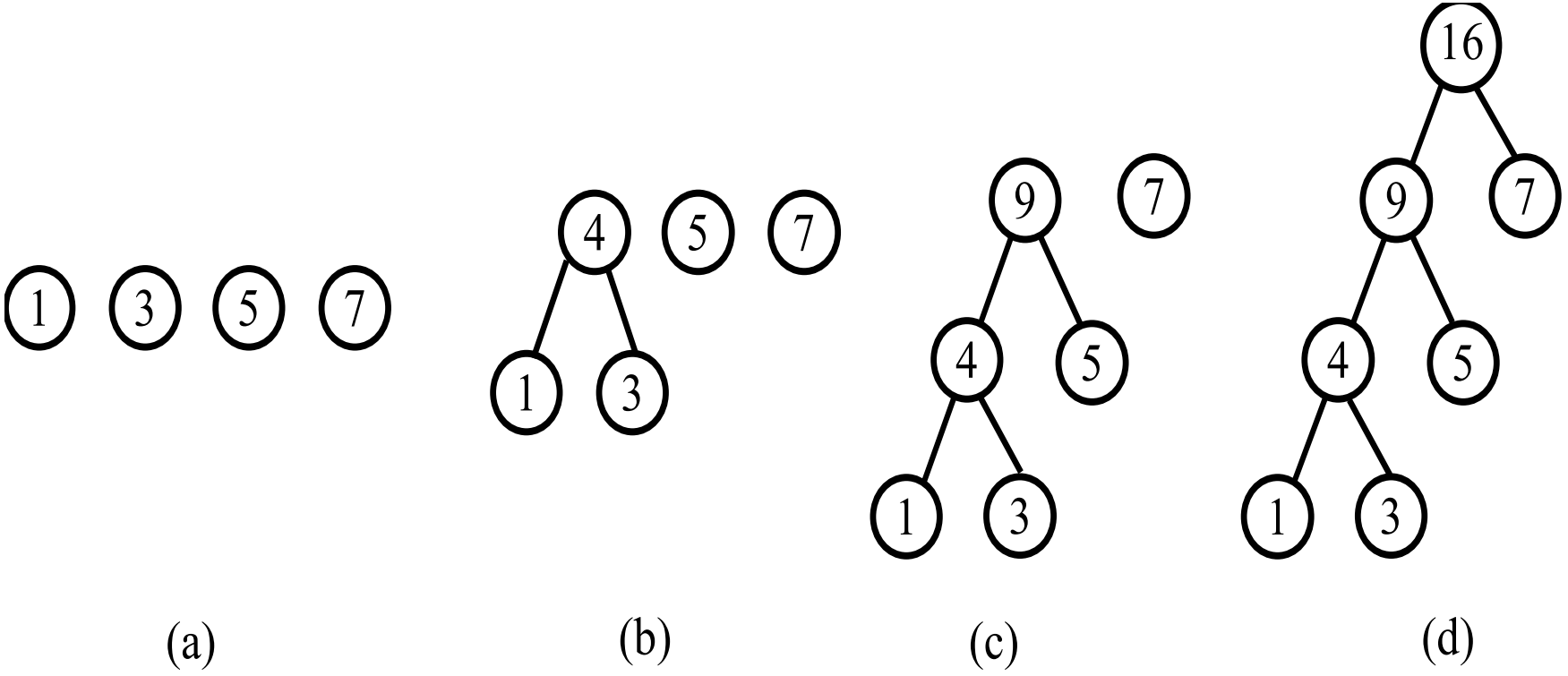
① 由给定的n个权值$\{W_1, W_2, \ldots, W_n\}$，构造n棵只有一个叶子结点的二叉树，从而得到一个二叉树的集合$F=\{T_1, T_2, \ldots, T_n\}$；

② 在F中选取根结点的权值最小和次小的两棵二叉树作为左、右子树构造一棵新的二叉树,这棵新的二叉树根结点的权值为其左、右子树根结点权值之和;

③ 在集合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集合F中;

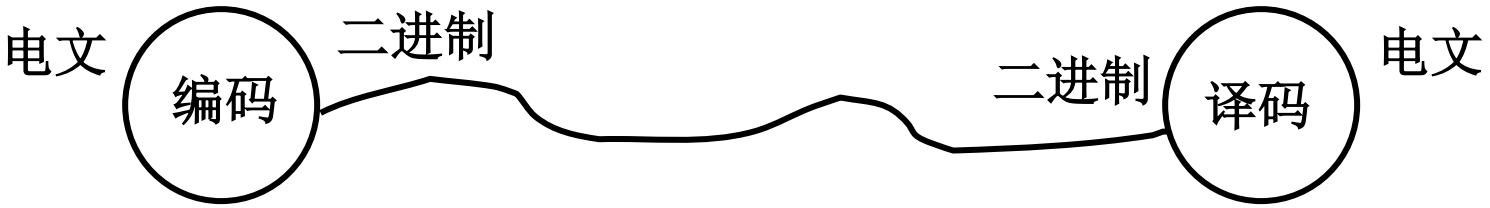
④ 重复(2)、(3)两步,当F中只剩下一棵二叉树时,这棵二叉树便是所要建立的哈夫曼树。

给定权值w=(1,3,5,7)来构造一棵哈夫曼树 。



## 3 哈夫曼编码

### 1. 编码




例，传送 ABACCD，四种字符，可以分别编码为 00,01,10,11。

则原电文转换为 00 01 00 10 10 11。

对方接收后，采用二位一分进行译码。

当然，为电文编码时，总是希望总长越短越好，
如果对每个字符设计长度不等的编码，且让电文中出现次数较多的字符采用较短的编码，则可以减短电文的总长。

例，对 ABACCD 重新编码，分别编码为 0 , 00 , 1 , 01。
A B C D

则原电文转换为 0 00 0 1 1 01。 减短了。

问题： 如何译码？

前四个二进制字符就可以多种译法。

AAAA BB

## 2. 前缀编码

若设计的长短不等的编码，满足任一个编码都不是另一个编码的前缀，则这样的编码称为前缀编码。

例， A , B , C , D 前缀编码可以为 0 , 110 , 10 , 111

利用二叉树设计二进制前缀编码。

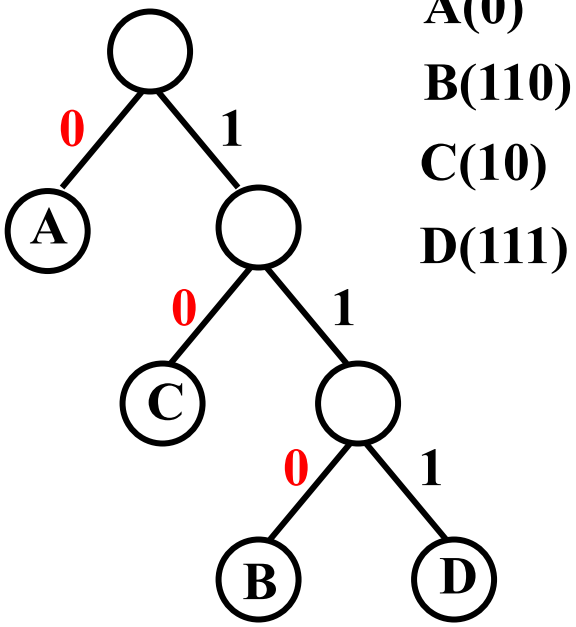
叶子结点表示 A , B , C , D 这 4 个字符

左分支表示 ‘0’，右分支表示 ‘1’

从根结点到叶子结点的路径上经过的二进制符号串作为该叶子结点字符的编码

路径长度为编码长度

证明其必为前缀编码



如何得到最短的二进制前缀编码？

3. 赫夫曼编码

设每种字符在电文中出现的概率 $w_i$ 为，则依此 n 个字符出现的概率做权，可以设计一棵赫夫曼树，使

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

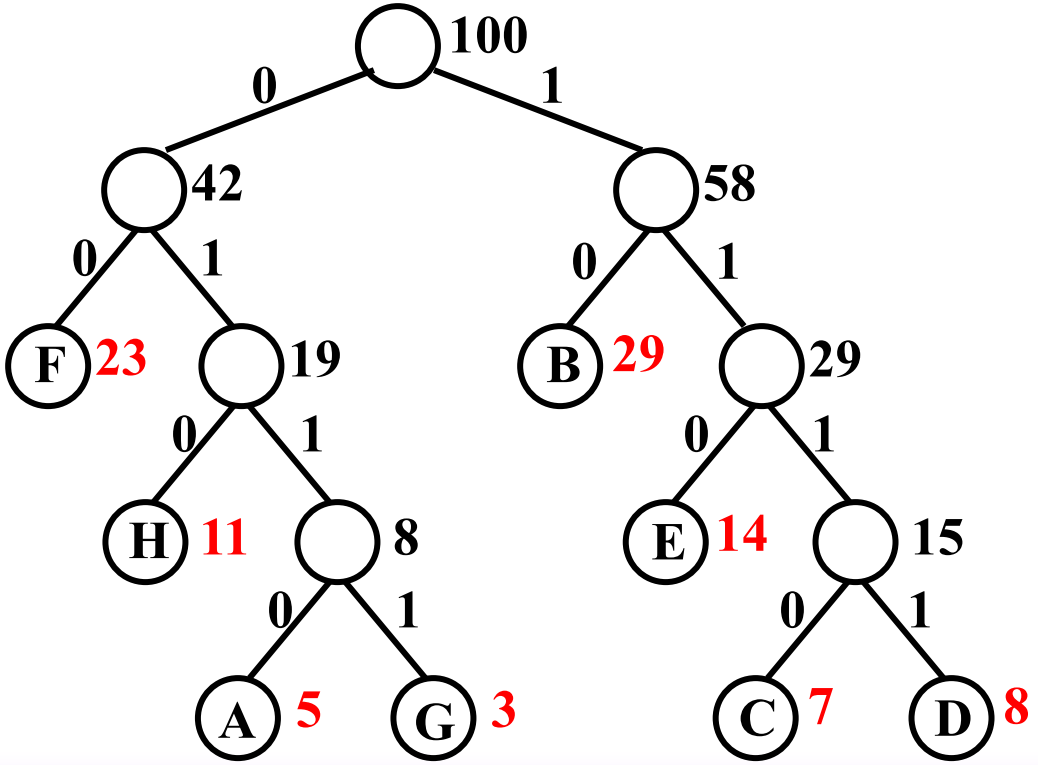
$w_i$ 为叶子结点的出现概率 ( 权)

$l_i$ 为根结点到叶子结点的路径长度

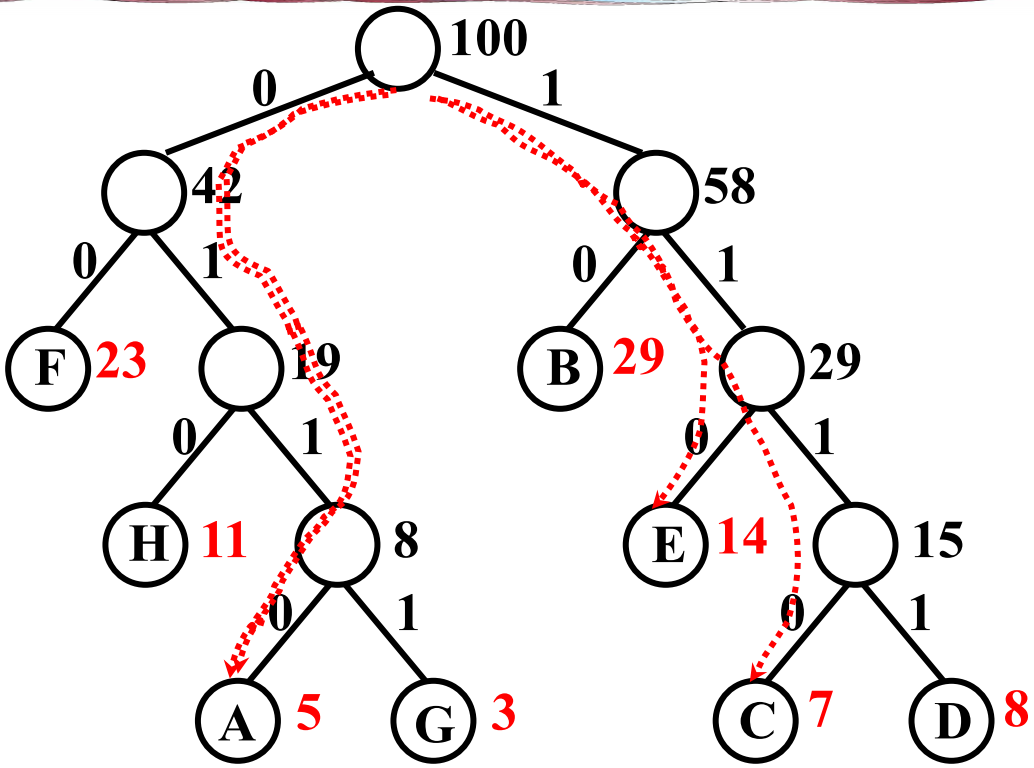
例，某通信可能出现 A B C D E F G H 8 个字符，其概率分别为 0.05 , 0.29 , 0.07 , 0.08 , 0.14 , 0.23 , 0.03 , 0.11 ， 试设计赫夫曼编码

不妨设 w = { 5 , 29 , 7 , 8 , 14 , 23 , 3 , 11 }

排序后 w = { 100 }




A (0110)
B (10)
C (1110)
D (1111)
E (110)
F (00)
G (0111)
H (010)




A (0110)
B (10)
C (1110)
D (1111)
E (110)
F (00)
G (0111)
H (010)

ACEA 编码为 0110 1110 110 0110

如何译码？ A C E A

1. 从根结点出发，从左至右扫描编码，
2. 若为 ‘0’ 则走左分支，若为 ‘1’ 则走右分支，直至叶结点为止，
3. 取叶结点字符为译码结果，返回重复执行 1,2,3 直至全部译完为止

## 4. 哈夫曼编码算法的实现

哈夫曼树中没有度为1的结点(严格的或正则的二叉树)。

n个叶子结点，共有2n-1个结点。

```
typedef struct
{
    unsigned int weight ;  // 结点的权值
    unsigned int parent,  lchild, rchild ;
}HTNode,  * HuffmanTree;  //动态分配数组存储哈夫曼树
typedef char  * *HuffmanCode ; //动态分配数组存储哈夫曼编码
```

```
void HuffmanCoding( HuffmanTree &HT ,
                    HuffmanCode &HC, int *w,  int n)
{
  m=2*n-1;
  HT = (HuffmanTree)malloc((m+1)*sizeof(HTNode));
  for(p=HT+1, i=1; i<=n; ++i,++p,++w) *p ={ *w, 0, 0, 0};
  for( ; i<=m; ++i, ++p) *p = {0, 0, 0, 0};
  for(i=n+1; i<=m; i++){
      select(HT, i-1, s1, s2); //在HT[1..i-1],
      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;
  }
}
```

```
    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[i].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]);
}
free(cd);
```

```
HC=malloc(…);    cd = malloc(…);
p = m; cdlen = 0;   for(i=1;i<=m;++i)  HT[i].weight = 0;
while(p) {
   if(HT[p].weight == 0){//向左
       HT[p].weight = 1;
       if(HT[p].lchild != 0){p = HT[p].lchild; cd[cdlen++]='0';}
       else if(HT[p].rchild == 0){
         HC[p] = (char *)malloc((cdlen+1)*sizeof(char));
         cd[cdlen]='\0'; strcpy(HC[p], cd); }
  }else if(HT[p].weight == 1){//向右
       HT[p].weight = 2;
       if(HT[p].rchild!=0) {p=HT[p].rchild; cd[cdlen++] = '1';}
  }else{
       HT[p].weight=0; p=HT[p].parent; --cdlen;
  }
}
```

- 作业：

  **6.26**

例1：假设二叉树采用二叉链存储结构存储，试设计一个算法，输出一棵给定二叉树的所有叶子结点。

解：输出一棵二叉树的所有叶子结点的递归模型f()如下：

| | |
|---|---|
| f(b)：不做任何事件 | 若b=NULL |
| f(b)：输出b结点的data域 | 若b为叶子结点 |
| f(b)：f(b->lchild);f(b->rchild) | 其他情况 |

```
void DispLeaf(BiTree b)
 {
      if (b!=NULL) {
           if (b->lchild==NULL && b->rchild==NULL)
              printf("%c ",b->data);
           else {
              DispLeaf(b->lchild);
              DispLeaf(b->rchild);
           }
      }
}
```

例2:求高度BiTreeDepth(b)

求二叉树的高度的递归模型f()如下：

f(NULL)=0

f(b)=MAX{f(b->lchild),f(b->rchild)}+1 其他情况

对应的算法如下：

```
int BiTreeDepth(BiTree b)
{    int lchilddep,rchilddep;
     if (b==NULL) return(0); /*空树的高度为0*/
     else {
          lchilddep=BiTreeDepth(b->lchild);
                /*求左子树的高度为lchilddep*/
          rchilddep=BiTreeDepth(b->rchild);
                /*求右子树的高度为rchilddep*/
          return (lchilddep>rchilddep)?
               (lchilddep+1): (rchilddep+1);
     }
}
```

例3：假设二叉树采用二叉链存储结构，设计一个算法判断两棵二叉树是否相似，所谓二叉树t1和t2是相似的指的是t1和t2都是空的二叉树；或者t1和t2的根结点是相似的，以及t1的左子树和t2的左子树是相似的且t1的右子树与t2的右子树是相似的。

解：判断两棵二叉树是否相似的递归模型f()如下：

**f(t1,t2)=true　若t1=t2=NULL**

**f(t1,t2)=false　若t1、t2之一为NULL,另一不为NULL**

**f(t1,t2)=f(t1->lchild,t2->lchild) && f(t1->rchild,t2->rchild)**

**其他情况**

```
int Like(BiTree t1,BiTree t2)
/*t1和t2两棵二叉树相似时返回1, 否则返回0*/
{    int like1,like2;
     if (t1==NULL && t2==NULL) return 1;
     else if (t1==NULL || t2==NULL)  return 0;
     else {
          like1=Like(t1->lchild, t2->lchild);
          like2=Like(t1->rchild, t2->rchild);
          return (like1 && like2);   /*返回like1和like2的与*/
     }
}
```

例4: 编写按层次（同一层从左至右）遍历二叉树的算法。

```
void LayerOrder(Bitree T)//层序遍历二叉树
{
    InitQueue(Q);
    EnQueue(Q,T);
    while(!QueueEmpty(Q)) {
        DeQueue(Q,p);
        visit(p);
        if(p->lchild) EnQueue(Q,p->lchild);
        if(p->rchild) EnQueue(Q,p->rchild);
    }
}
```

# 本章小结

本章的基本学习要点如下：

(1) 掌握树的相关概念，包括树、结点的度、树的度、分支结点、叶子结点、孩子结点、双亲结点、树的深度、森林等定义。

(2) 掌握树的表示，包括树形表示法、文氏图表示法、凹入表示法和括号表示法等。

(3) 掌握二叉树的概念，包括二叉树、满二叉树和完全二叉树的定义。

(4) 掌握树与二叉树的性质。

(5) 重点掌握二叉树的存储结构，包括二叉树顺序存储结构和链式存储结构。

(6) 重点掌握二叉树的基本运算和各种遍历算法的实现。

(7) 掌握线索二叉树的概念和相关算法的实现。

(8) 掌握哈夫曼树的定义、哈夫曼树的构造过程和哈夫曼编码产生方法。

(9) 灵活运用二叉树这种数据结构解决一些综合应用问题。

作业 (4.18) 编写算法，求串s所含不同字符的总数和每种字符的个数。

```
void StrCount(SString  s)
{    SString r;
     int *num = new int[s[0]];
     for(i=1; i<=s[0]; i++)    num[i] =0;
     r[1] = s[1];   num[1] = 1;    r[0] = 1;
     for(i=2; i<=s[0]; i++){
          j = 1;
          while( s[i]!=r[j] && j<=r[0])  ++j;
          ++num[j];
          if(j>r[0]) { r[j] = s[i];    ++r[0]; }
      }
}
```

```
void StrCount(SString s)
{
    int num[256];
    for(i=0; i<256; ++i)   num[i] = 0;
    for(i=1; i<=s[0]; ++i)
          ++num[(int)s[i]];
    j = 0;
    for(i=0; i<256; ++i)
        if(num[i])
            ++j;
}
```

作业（5.25）矩阵相加。

```
typedef struct{
        int    rows, cols;
        int    *elem;   //存储数组中的非零元
        bool  **map;//矩阵中相应元素是否为零元素
} BMMatrix; //用位图表示的矩阵类型
```

$$A=\begin{bmatrix}15 & 0 & 0 & 22\\ 0 & -6 & 0 & 0\\ 91 & 0 & 0 & 0\end{bmatrix}$$

$$elem=(15,22,-6,9)$$

$$map=\begin{bmatrix}1 & 0 & 0 & 1\\ 0 & 1 & 0 & 0\\ 1 & 0 & 0 & 0\end{bmatrix}$$

```
void BMMatrixAdd(BMMatrix A,BMMatrix B,BMMatrix&C)
{   C.rows=A.rows;    C.cols=A.cols; pa=0; pb=0; pc=0;
   for(i=0; i<A.rows; i++) //每一行的相加
       for(j=0; j<A.cols; j++) { //每一个元素的相加
          if(A.map[i][j]&&B.map[i][j]){
              if(A.elem[pa]+B.elem[pb]){
                    C.elem[pc]=A.elem[pa]+B.elem[pb];
                    C.map[i][j]=1;   pc++;
               }
               pa++; pb++;
          }
          else if(A.map[i][j]&&!B.map[i][j]) {
              C.elem[pc]=A.elem[pa];  C.map[i][j]=1;  pa++; pc++;
          }
          else if(!A.map[i][j]&&B.map[i][j]) {
              C.elem[pc]=B.elem[pb]; C.map[i][j]=1;  pb++; pc++;
          }
       }
}
```