

# 算法与数据结构

主讲教师：陈娜

联系方式：[chenna@stdu.edu.cn](mailto:chenna@stdu.edu.cn)

# 知识回顾

## 线性表

- 一般的线性表
- 特殊的线性表：栈、队列
- 线性表的扩展：数组、广义表

## 特点：

- 有序
- 前趋、后继

# 第五章 树和二叉树

# 树型结构：非线性数据结构

树是以分支关系定义的层次结构。

树结构在客观世界广泛存在：

在社会人文领域，人类社会构成、组织机构等也具有树结构关系。

# 教学目标

1. 掌握二叉树的基本概念、性质和存储结构
2. 熟练掌握二叉树的前、中、后序遍历方法
3. 理解线索化二叉树的思想
4. 熟练掌握：霍夫曼树的实现方法、构造霍夫曼编码的方法
5. 掌握：森林与二叉树的转换，树的遍历方法

# 主要内容

5.1 树的定义和基本术语

5.2 二叉树

5.3 遍历二叉树和线索二叉树

5.4 树和森林

5.5 霍夫曼树及其应用

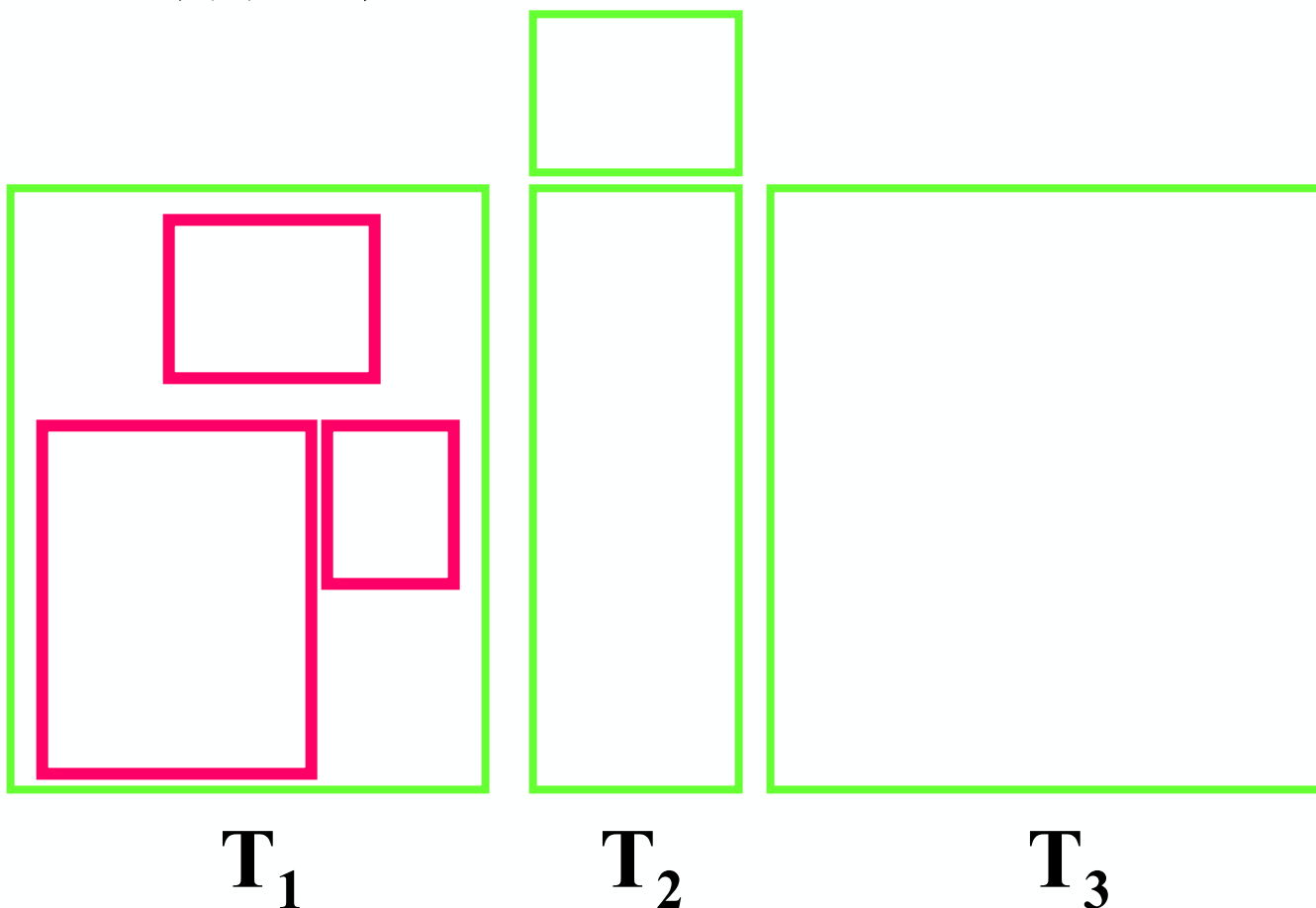
## 5.1.1 树的定义



## 5.1.1 树的定义和基本术语

树是 $n$ 个结点的有限集

Ⓐ

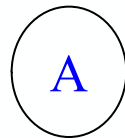




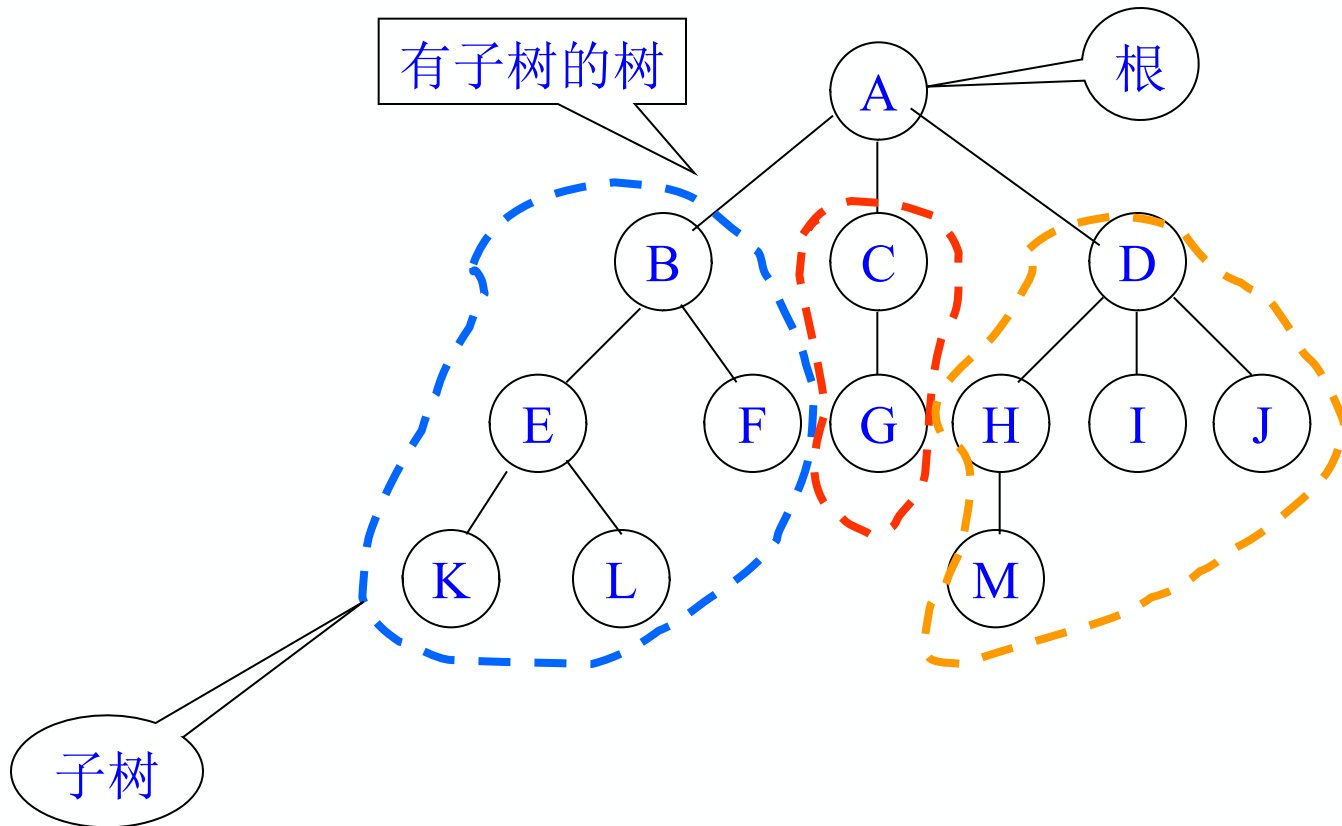
树 (tree) 是  $n$  ( $n \geq 0$ ) 个结点的有限集。在任意一棵非空树中:

- (1) 有且仅有一个特定的称为根的结点;
- (2) 当  $n > 1$  时, 其余结点可分为  $m$  ( $m > 0$ ) 个互不相交的有限集  $T_1, T_2, \dots, T_m$ , 其中每一个集合本身又是一棵树, 并且称为根的子树。

只有根结点的树



有子树的树



# 树的表示法

- ❖ 图形表示法
- ❖ 嵌套集合表示法
- ❖ 广义表表示法
- ❖ 凹入表示法
- ❖ 左孩子—右兄弟表示法

# 图形表示法

西安石油大学

根

电信系

计算机系

自控系

.....

教师

学生

其他人员

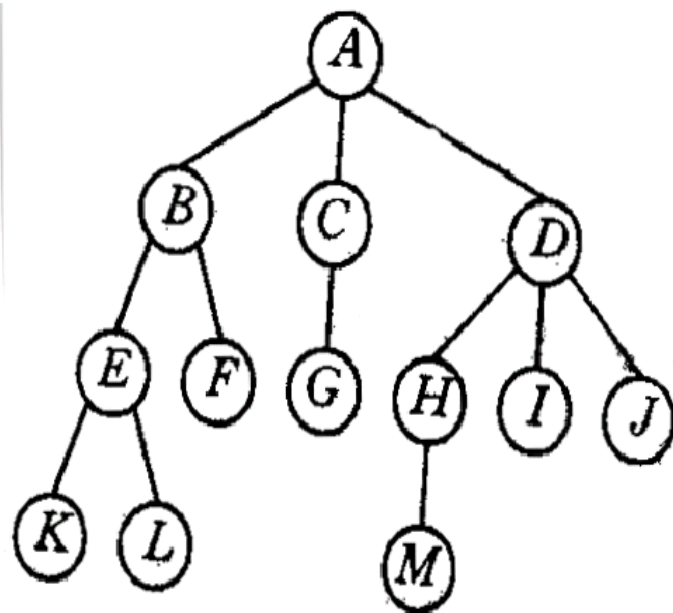
2001级

2002级

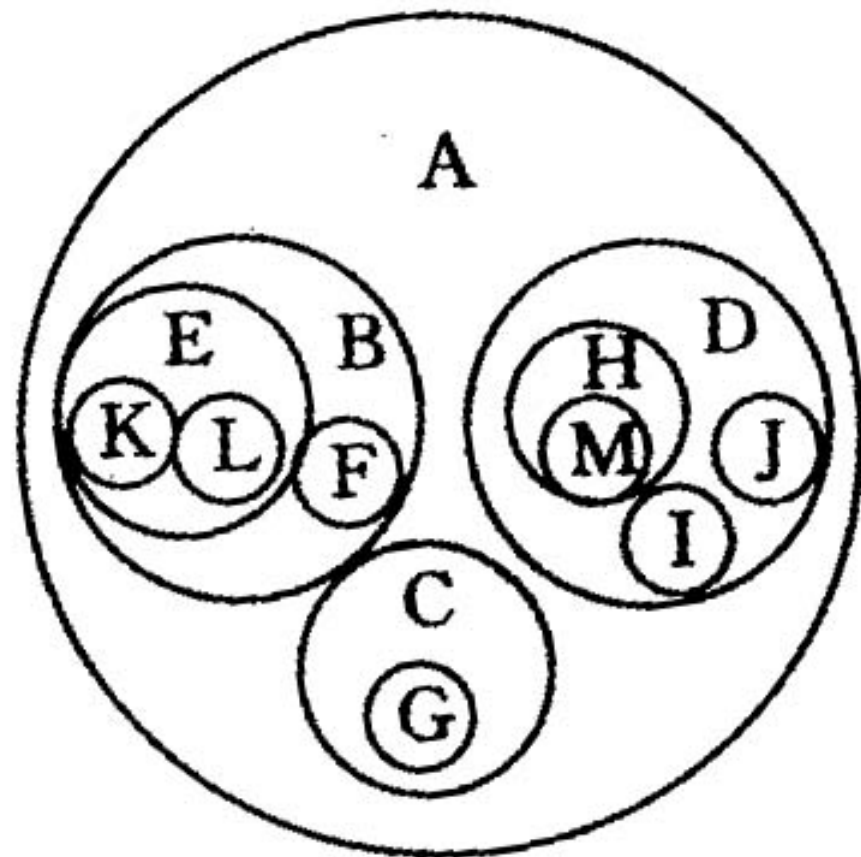
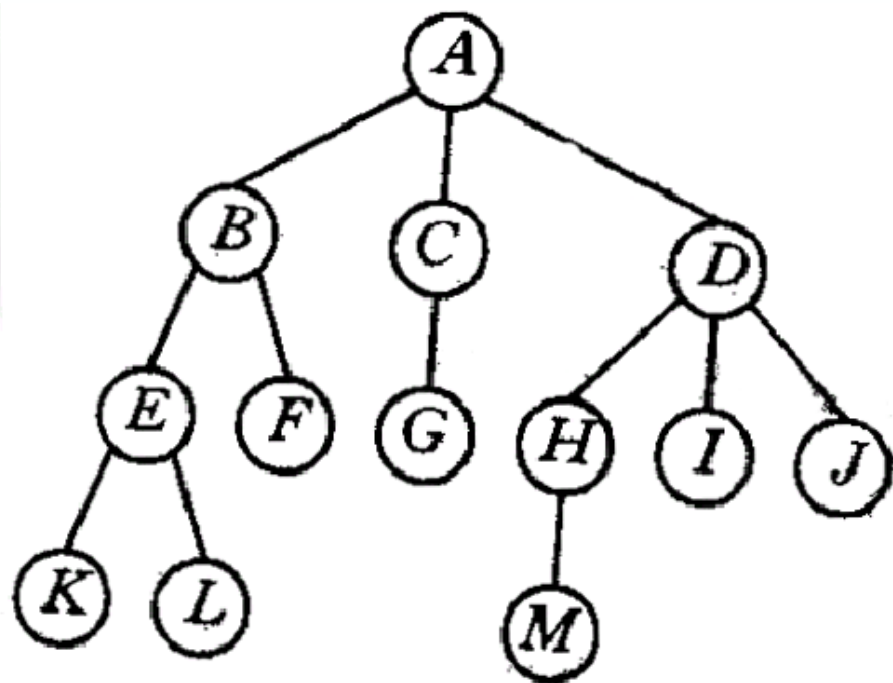
2003级

2004级

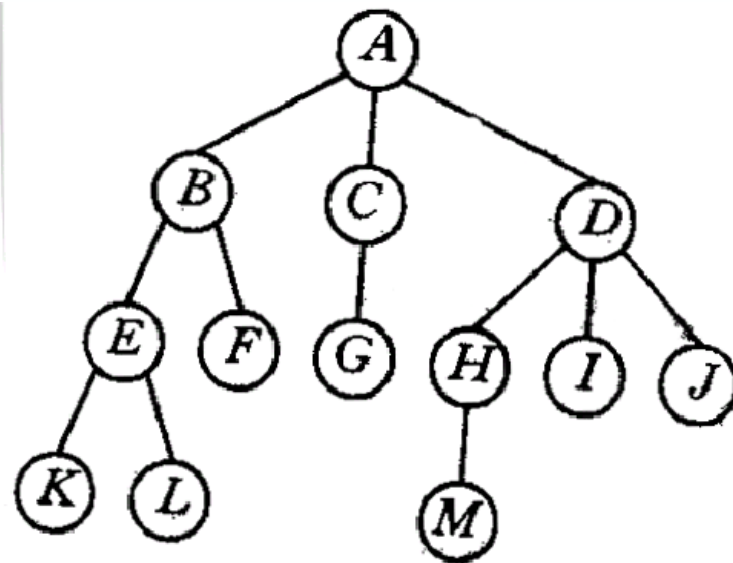
叶子



# 嵌套集合表示法



# 广义表表示法



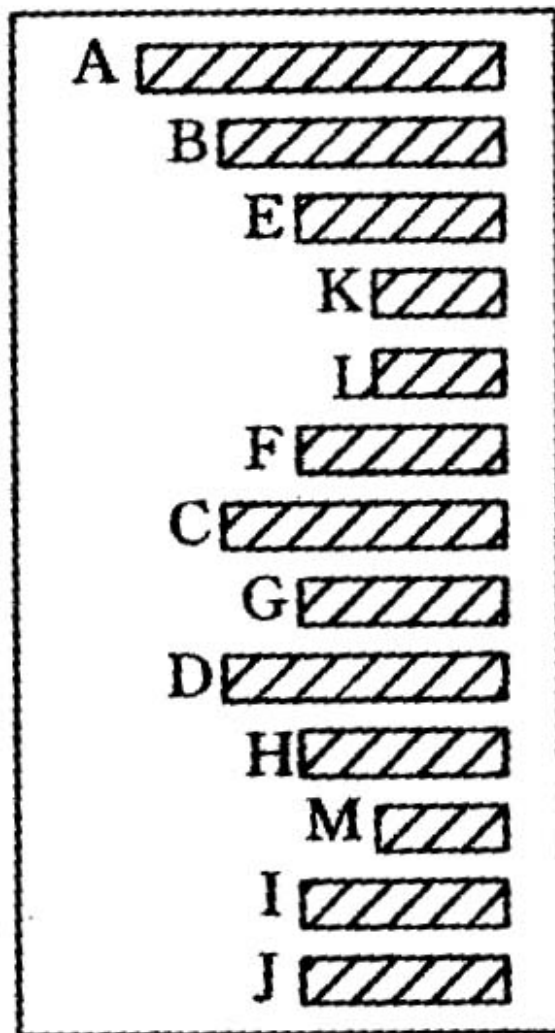
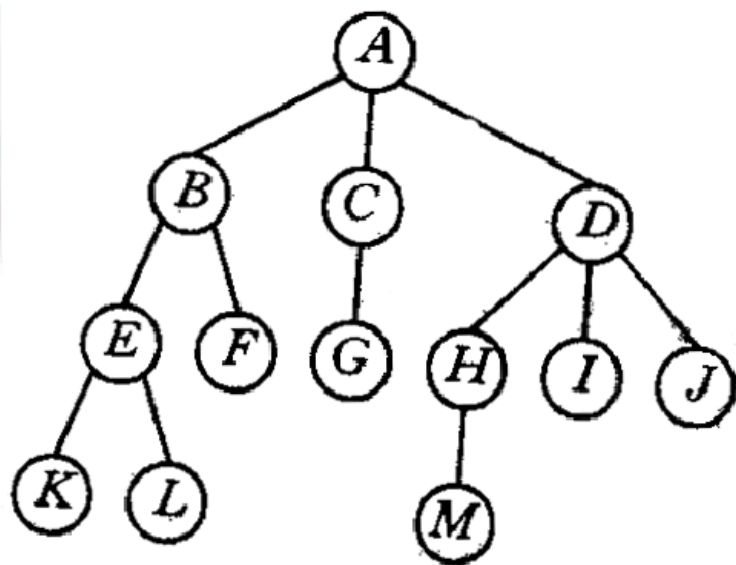
( **A** ( **B** ( E ( K, L ), F ), **C** ( G ), **D** ( H ( M ), I, J ) )

**约定：**

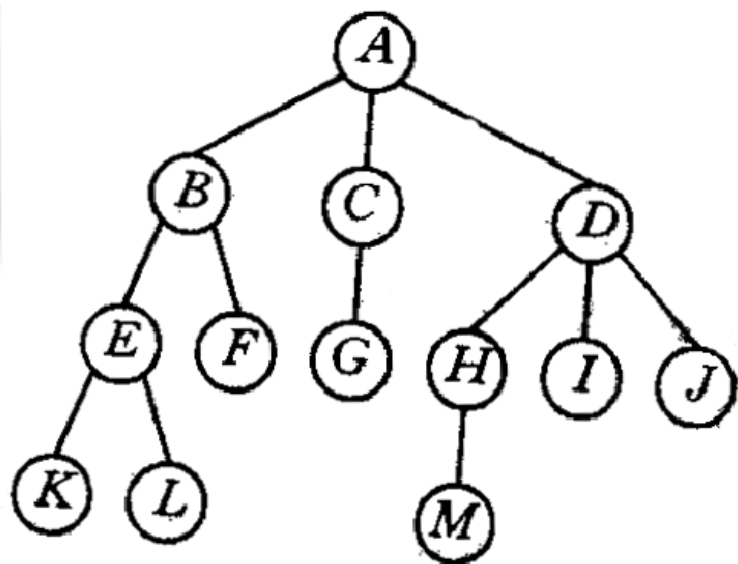
根作为由子树森林组成的表的名字写在表的左边

# 凹入表示法

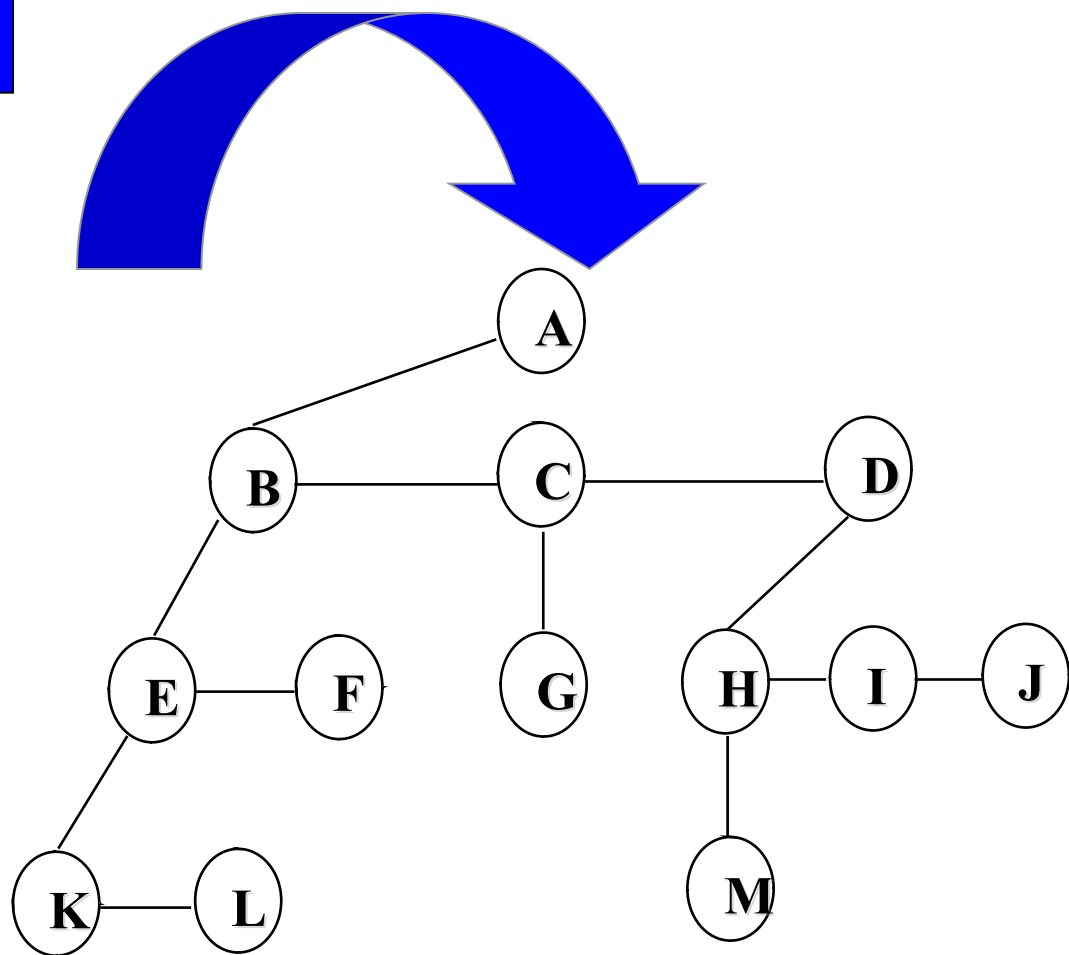
又称目录表示法



# 左孩子—右兄弟表示法



data	
左孩子	右兄弟



多叉树转为  
了二叉树



## 数据对象 D:

D是具有相同特性的数据元素的集合。

## 数据关系 R:

若D为空集，则称为空树。 否则：

- (1) 在D中存在唯一的称为根的数据元素root;
- (2) 当 $n > 1$ 时，其余结点可分为 $m$  ( $m > 0$ )个互不相交的有限集 $T_1, T_2, \dots, T_m$ ，其中每一棵子集本身又是一棵符合本定义的书，称为根root的子树。



# 基本操作:

✧ 查 找 类

✧ 插 入 类

✧ 删 除 类



# 查找类:

**Root(T)** // 求树的根结点

**Value(T, cur\_e)** // 求当前结点的元素值

**Parent(T, cur\_e)** // 求当前结点的双亲结点

**LeftChild(T, cur\_e)** // 求当前结点的最左孩子

**RightSibling(T, cur\_e)** // 求当前结点的右兄弟

**TreeEmpty(T)** // 判定树是否为空树

**TreeDepth(T)** // 求树的深度

**TraverseTree( T, Visit() )** // 遍历



# 插入类:

**InitTree(&T) // 初始化置空树**

**CreateTree(&T, definition)**  
// 按定义构造树

**Assign(T, cur\_e, value)**  
// 给当前结点赋值

**InsertChild(&T, &p, i, c)**  
// 将以c为根的树插入为结点p的第i棵子树



# 删除类:

**ClearTree(&T) // 将树清空**

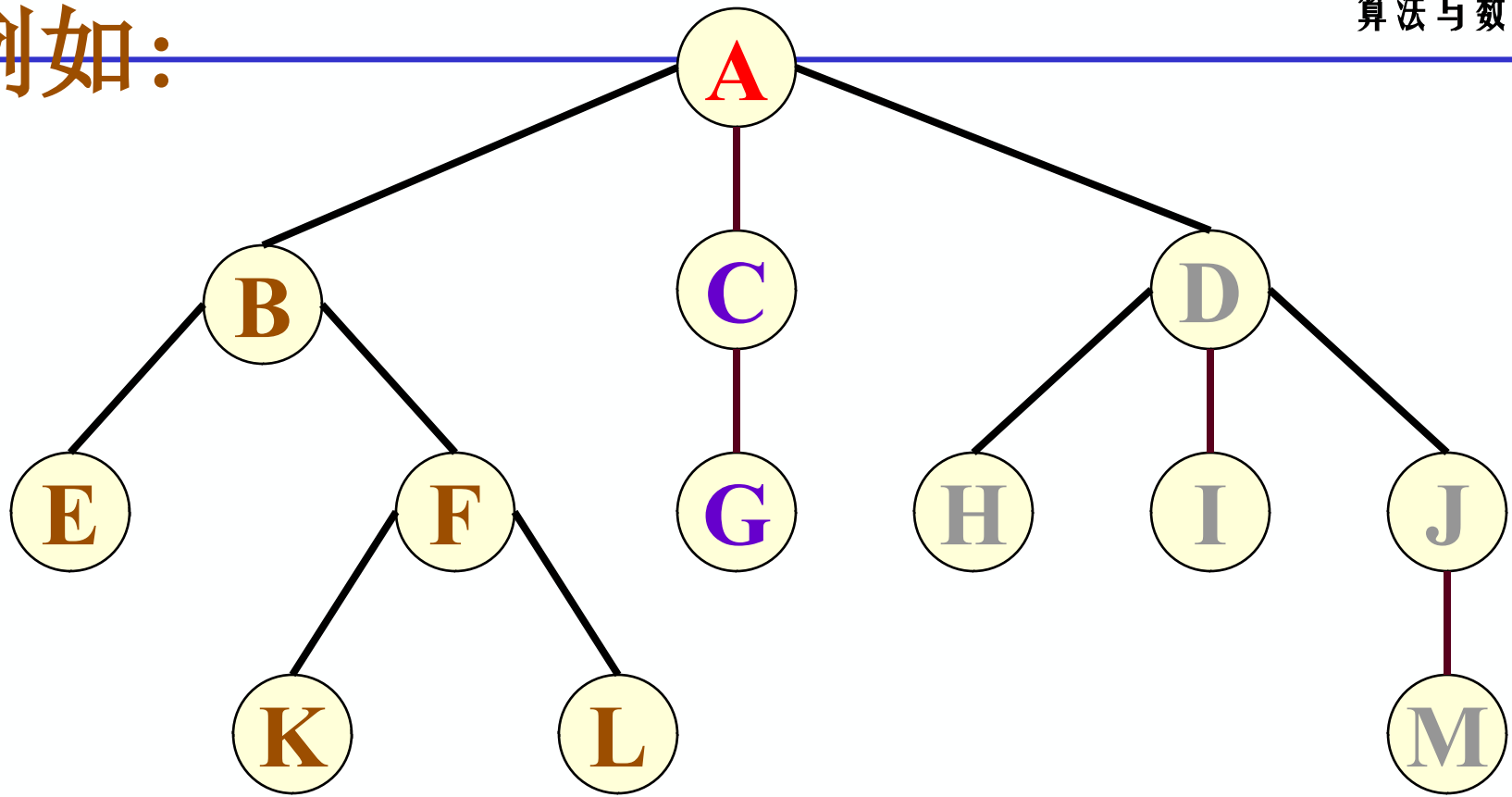
**DestroyTree(&T) // 销毁树的结构**

**DeleteChild(&T, &p, i)**

**// 删除结点p的第i棵子树**



例如：



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

树根
 $T_1$ 
 $T_2$ 
 $T_3$

## 有向树:

- (1) 有确定的根;
- (2) 树根和子树根之间为有向关系。

## 有序树:

子树之间存在确定的次序关系。

## 无序树:

子树之间不存在确定的次序关系。



# 对比树型结构和线性结构的 的结构特点



# 对比树型结构和线性结构的结构特点

## 线性结构

第一个数据元素  
(无前驱)

最后一个数据元素  
(无后继)

其它数据元素  
(一个前驱、  
一个后继)

## 树型结构

根结点  
(无前驱)

多个叶子结点  
(无后继)

其它数据元素  
(一个前驱、  
多个后继)

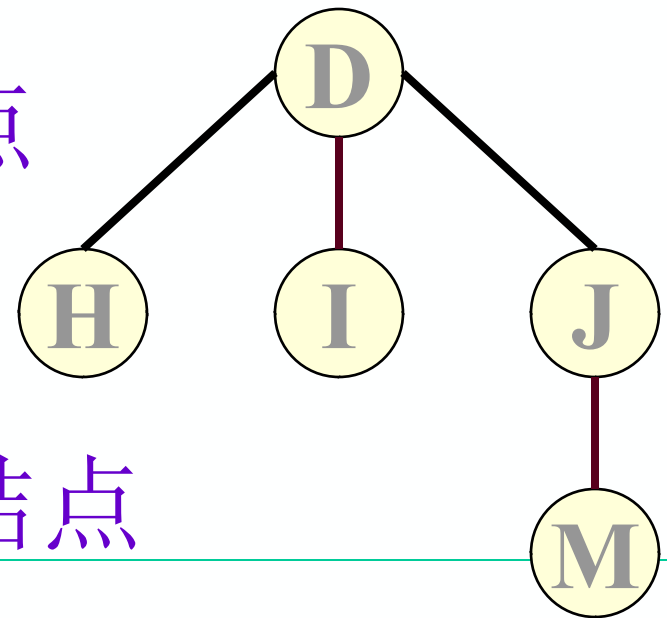
## 5.1.2 基本术语

**结点：** 数据元素+若干指向子树的分支

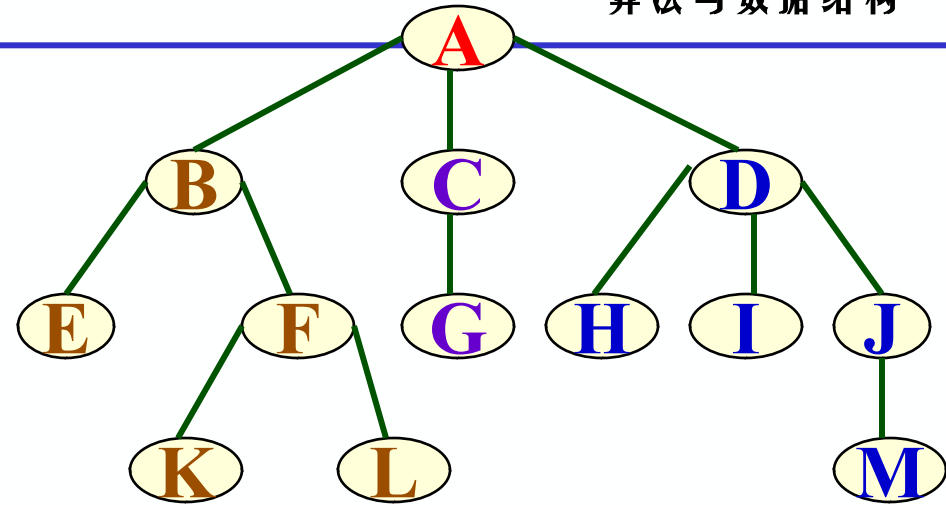
**结点的度：** 分支的个数

**树的度：** 树中所有结点的度的最大值

**叶子结点：** 度为零的结点



**分支结点：** 度大于零的结点



**孩子:**结点的子树的根。

**双亲:**该结点称为孩子的双亲。

**兄弟:**同一个双亲的孩子之间互称兄弟。

**祖先:**从根到该结点所经分支上的所有结点

**子孙:**以某结点为根的子树中的任一结点。

**结点的层次:**假设根结点的层次为1, 第 $l$ 层的结点的子树根结点的层次为 $l+1$

**堂兄弟:**其双亲在同一层的结点

**树的深度:** 树中结点的最大层次

**有序树:** 子树之间存在确定的次序关系

**无序树:** 子树之间不存在确定的次序关系

结点A的度: 3

结点B的度: 2

结点M的度: 0

叶子: K, L, F, G, M, I, J

结点I的双亲: D

结点L的双亲: E

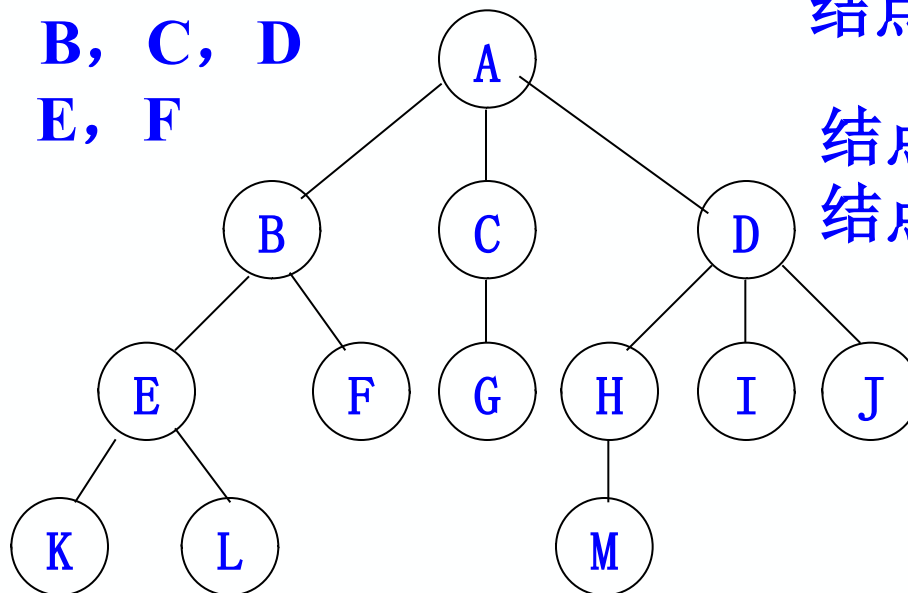
结点A的孩子: B, C, D

结点B的孩子: E, F

结点B, C, D为兄弟

结点K, L为兄弟

树的度: 3



树的深度: 4

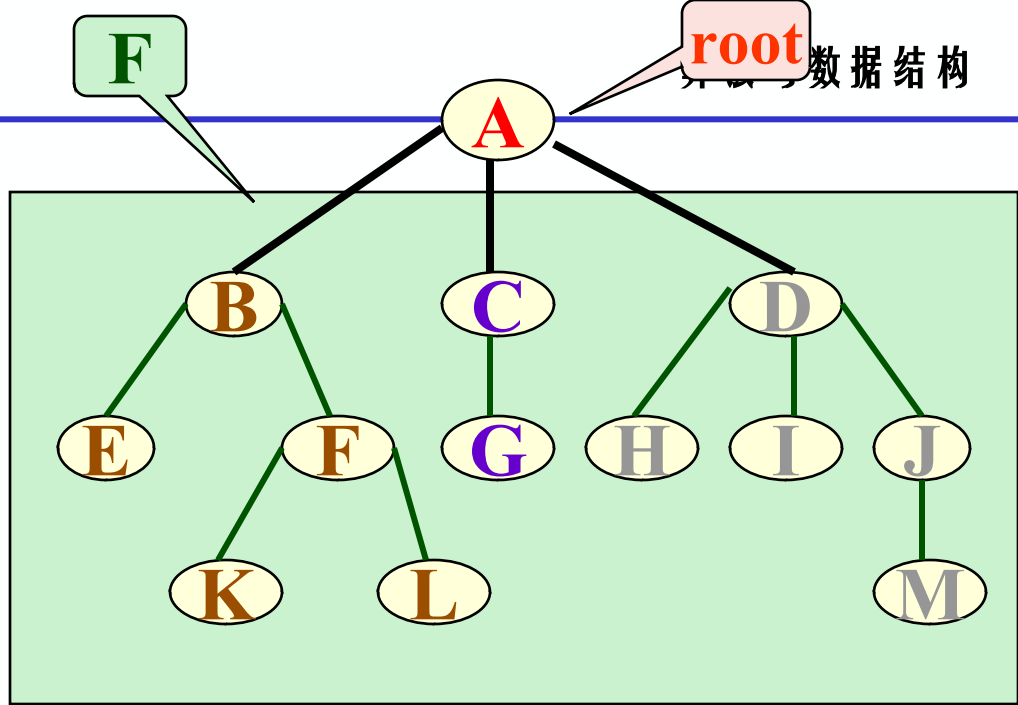
结点A的层次: 1

结点M的层次: 4

结点F, G为堂兄弟

# 森林:

是 $m$  ( $m \geq 0$ ) 棵互不相交的树的集合



任何一棵非空树是一个二元组

$$\text{Tree} = (\text{root}, F)$$

其中: root 被称为根结点

F 被称为子树森林



# 5.2 二叉树

树的操作实现比较复杂。

**解决思路：** 先研究最简单、最有规律的树，然后设法把一般的树转化为简单树。

最简单的树——二叉树

为何要重点研究每结点最多只有两个“叉”的树？

- ✓ 二叉树的结构最简单，规律性最强；
- ✓ 可以证明，所有树都能转为唯一对应的二叉树，不失一般性。

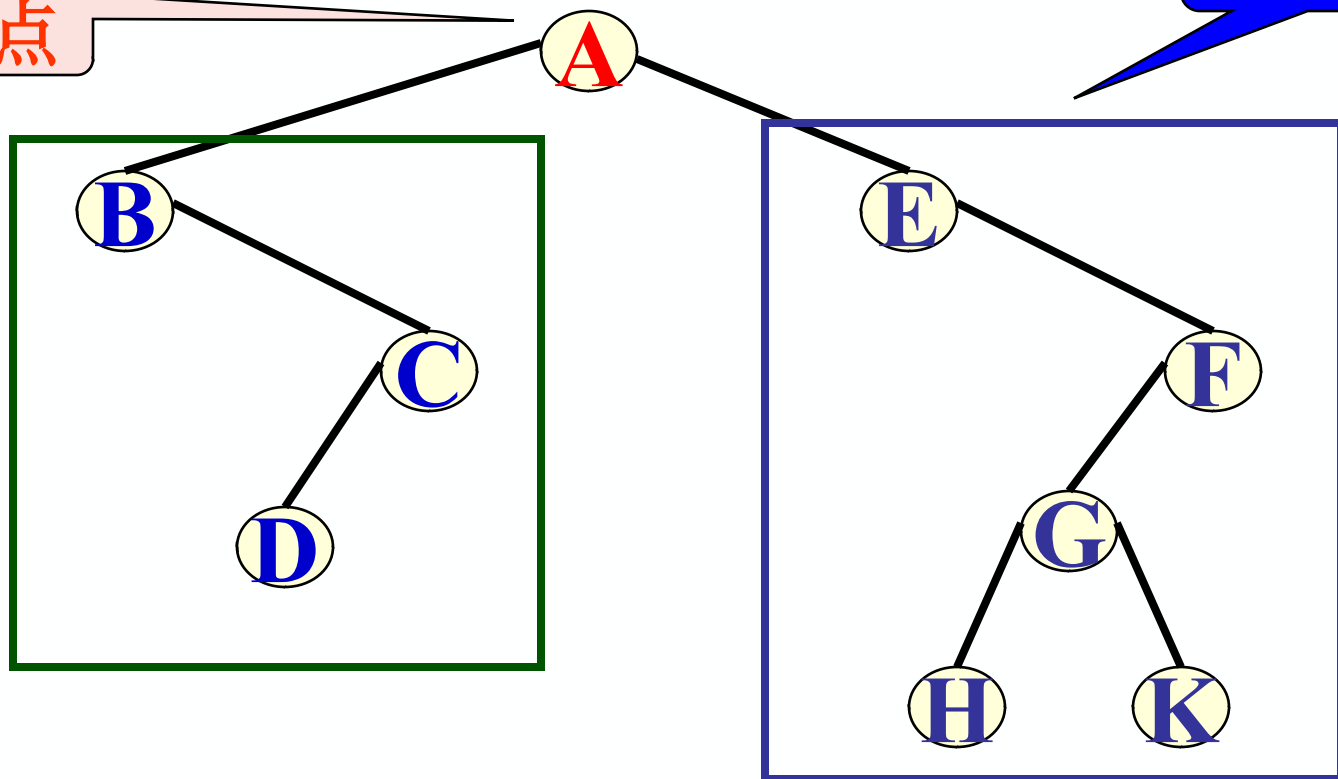




二叉树:或为空树，或是由一个根结点加上最多两棵分别称为左子树和右子树的、互不相交的二叉树组成。

右子树

根结点



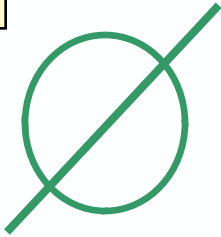
左子树

# 二叉树的特点

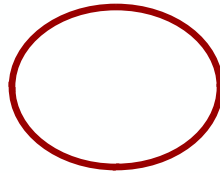
- (1) 每个结点至多有二棵子树(即不存在度大于2的结点);
- (2) 二叉树的子树有左、右之分, 且其次序不能任意颠倒。

# 二叉树的五种基本形态：

空树

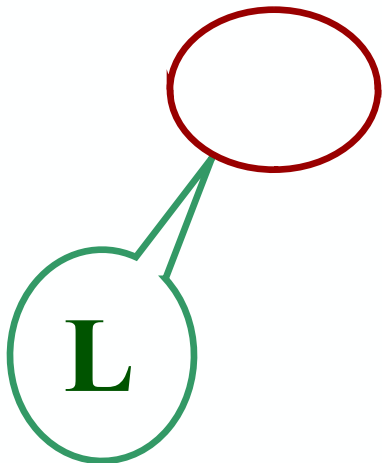


只含根结点

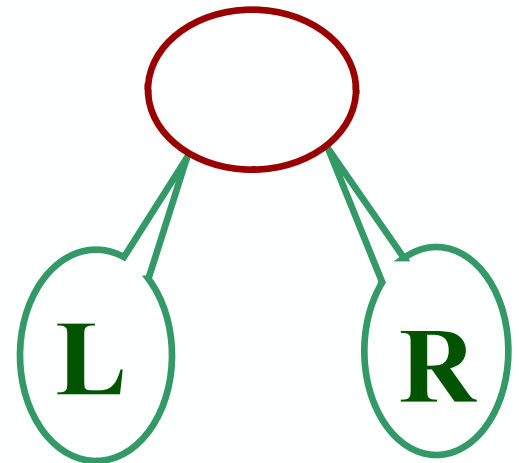
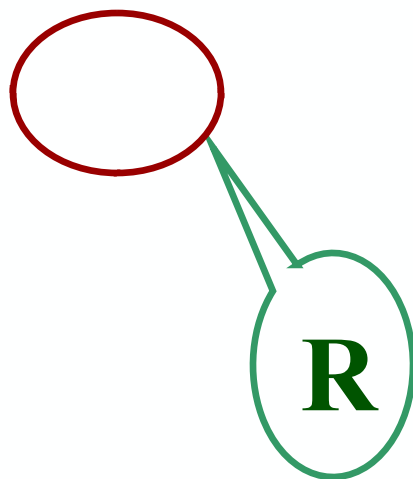


左右子  
树均不  
为空树

右子树为空树

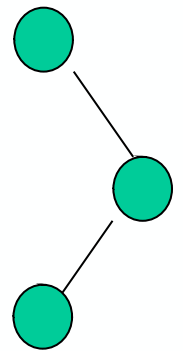
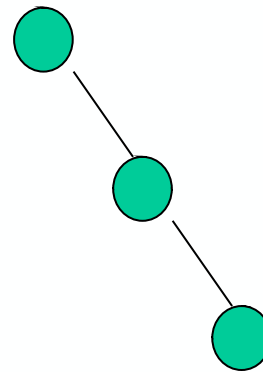
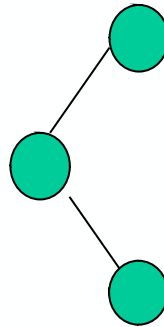
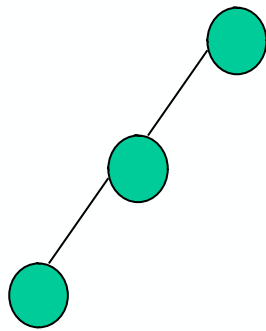
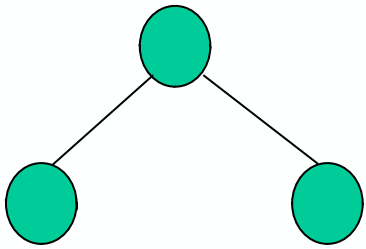


左子树为空树



具有3个结点的二叉树可能有几种不同形态？普通树呢？

5种/2种



# 二叉树的主要基本操作：



查找类



插入类



删除类



**Root(T); Value(T, e); Parent(T, e);**

**LeftChild(T, e); RightChild(T, e);**

**LeftSibling(T, e); RightSibling(T, e);**

**BiTreeEmpty(T); BiTreeDepth(T);**

**PreOrderTraverse(T, Visit());**

**InOrderTraverse(T, Visit());**

**PostOrderTraverse(T, Visit());**

**LevelOrderTraverse(T, Visit());**



**InitBiTree(&T);**

**Assign(T, &e, value);**

**CreateBiTree(&T, definition);**

**InsertChild(T, p, LR, c);**



**ClearBiTree(&T);**

**DestroyBiTree(&T);**

**DeleteChild(T, p, LR);**





# 二叉树 的重要特性

- **性质 1 :**

在二叉树的第  $i$  层上至多有  $2^{i-1}$  个结点。

( $i \geq 1$ )

用归纳法证明:

归纳基:  $i = 1$  层时, 只有一个根结点:

$$2^{i-1} = 2^0 = 1;$$

归纳假设: 假设对所有的  $j$ ,  $1 \leq j < i$ , 命题成立;

归纳证明: 二叉树上每个结点至多有两棵子树,  
则第  $i$  层的结点数  $= 2^{i-2} \times 2 = 2^{i-1}$ 。

- 性质 2 :

深度为  $k$  的二叉树上至多含  $2^k-1$  个结点  
( $k \geq 1$ ) 。

## 证明:

基于性质1, 深度为  $k$  的二叉树上的  
结点数至多为

$$2^0 + 2^1 + \dots + 2^{k-1} = 2^k - 1。$$

### 性质 3 :

对任何一棵二叉树, 若它含有 $n_0$ 个叶子结点、 $n_2$ 个度为 2 的结点, 则必存在关系式:  $n_0 = n_2 + 1$ 。

证明:

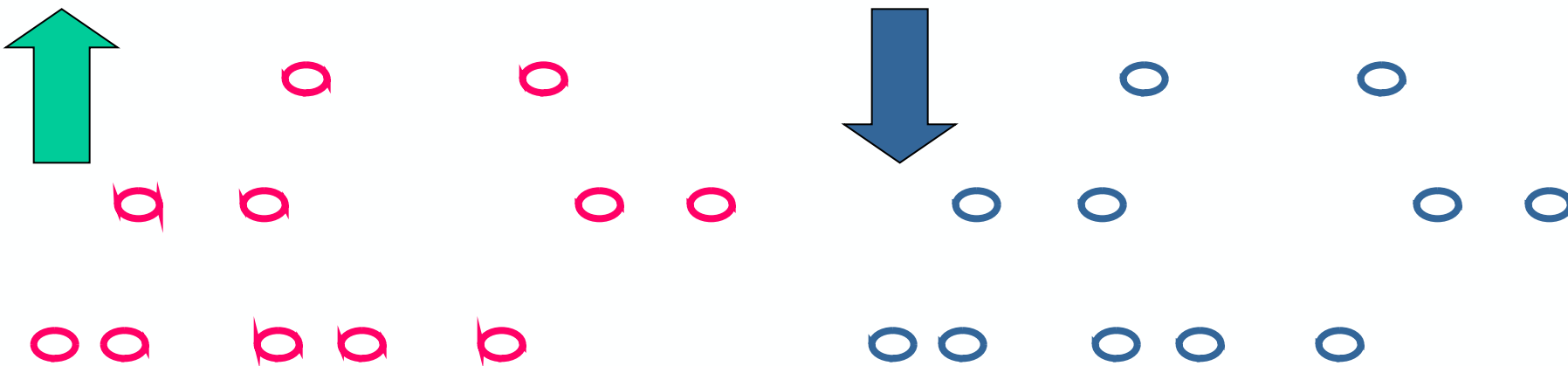
∵ 二叉树中全部结点数  $n = n_0 + n_1 + n_2$  (叶子数 + 1度结点数 + 2度结点数)

又 ∵ 二叉树中全部结点数  $n = B + 1$  (总分支数 + 根结点)

(除根结点外, 每个结点必有一个直接前趋, 即一个分支)

而 总分支数  $B = n_1 + 2n_2$  (1度结点必有1个直接后继, 2度结点必有2个)

**性质3:** 对于任何一棵二叉树，若2度的结点数有 $n_2$ 个，则叶子数 $n_0$ 必定为 $n_2+1$ （即 $n_0=n_2+1$ ）



$$B = n - 1$$

$$B = n_2 \times 2 + n_1 \times 1$$

$$n = n_2 \times 2 + n_1 \times 1 + 1 = n_2 + n_1 + n_0$$

# 特殊形态的二叉树

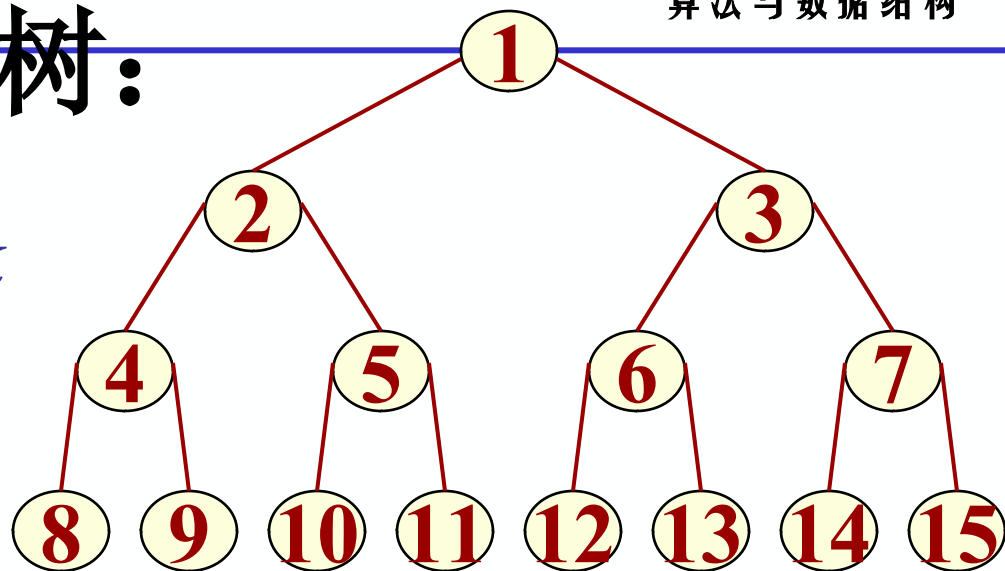
只有最后一层叶子不满，  
且全部集中在左边

**满二叉树：**一棵深度为 $k$  且有 $2^k - 1$ 个结点的二叉树。（特点：每层都“充满”了结点）

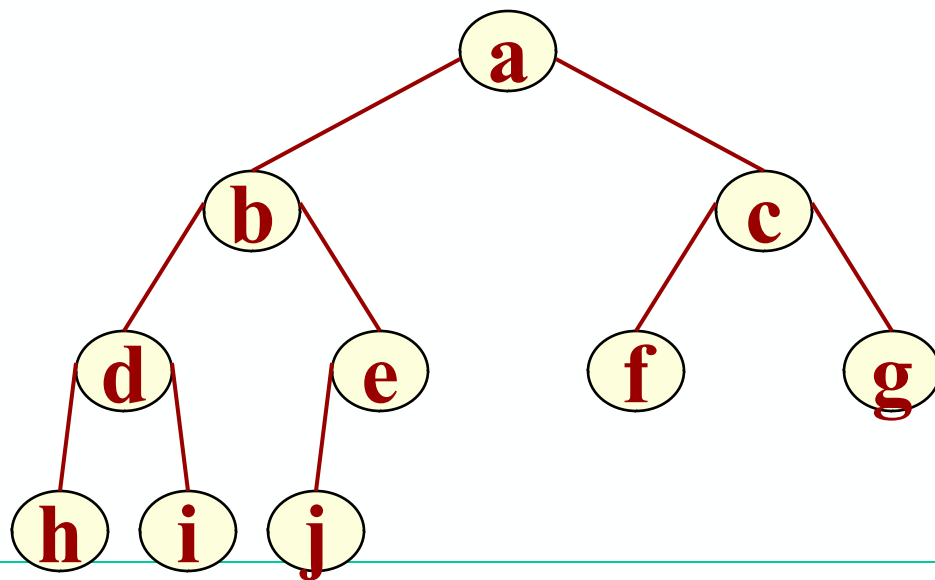
**完全二叉树：**深度为 $k$  的，有 $n$ 个结点的二叉树，当且仅当其每一个结点都与深度为 $k$  的满二叉树中编号从1至 $n$ 的结点一一对应

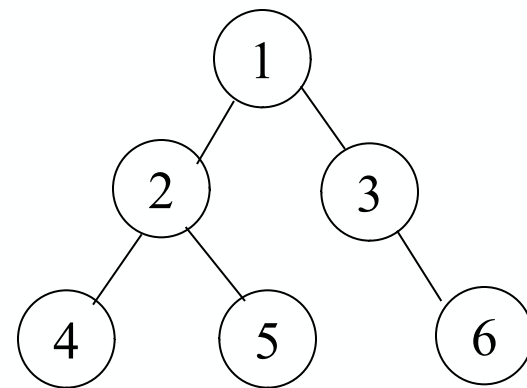
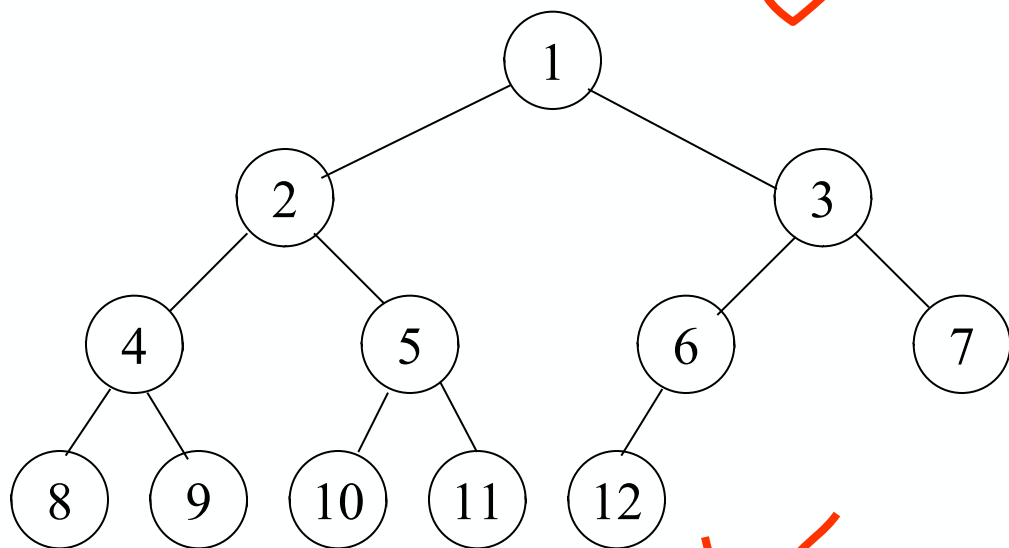
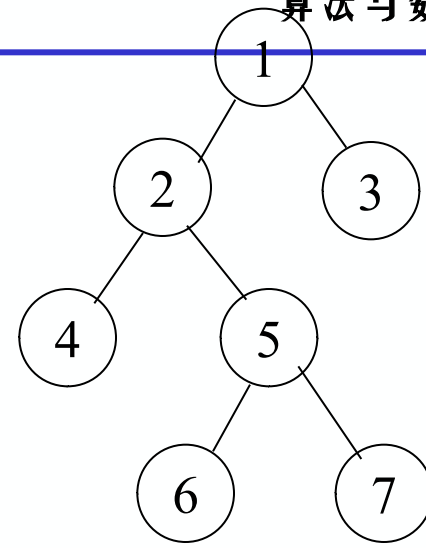
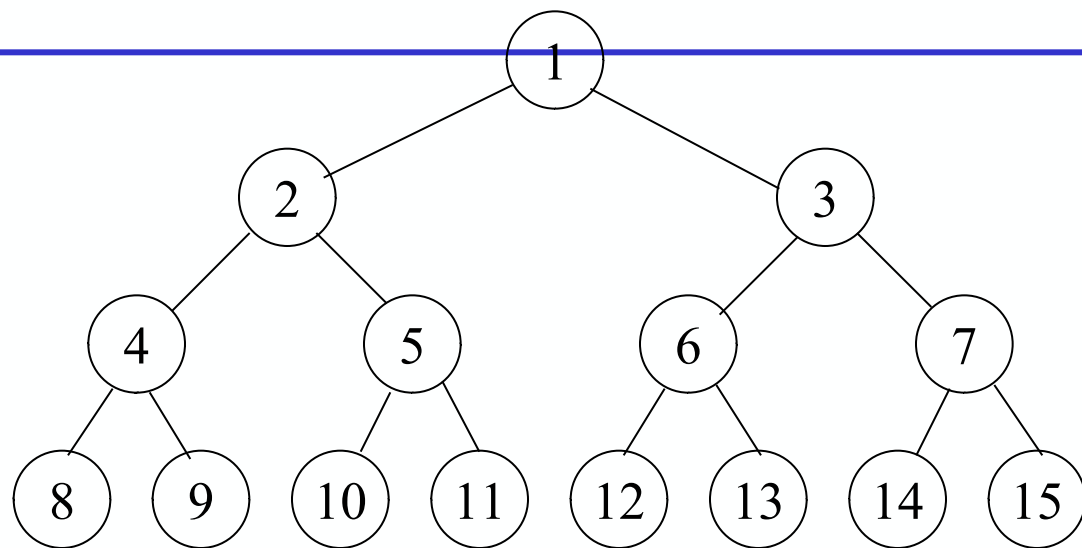
# 两类特殊的二叉树:

**满二叉树:** 深度为 $k$   
且含有 $2^k-1$ 个结点的  
二叉树。



**完全二叉树:** 树  
中所含的  $n$  个结点  
和满二叉树中编号  
为  $1$  至  $n$  的结点一  
一对应。







## 完全二叉树特点:

- (1) 叶子结点只可能在层次最大的两层上出现;
- (2) 对任一结点, 若其右分支下子孙的最大层次为 $i$ , 则其左分支下子孙的最大层次必为 $i$ 或 $i+1$ 。

### 满二叉树与完全二叉树的区别

满二叉树是叶子一个也不少的树, 而完全二叉树虽然前 $k-1$ 层是满的, 但最底层却允许在右边缺少连续若干个结点。满二叉树是完全二叉树的一个特例。

### 为何要研究这两种特殊形式?

因为它们在顺序存储方式下可以复原。

一棵完全二叉树有5000个结点，可以计算出其叶结点的个数是（ **2500** ）。

性质 4 :

具有  $n$  个结点的完全二叉树的深度为  $\lfloor \log_2 n \rfloor + 1$  。  $\lfloor x \rfloor$  表示  $\leq x$  的最大整数。

证明： 设完全二叉树的深度为  $k$

则根据第二条性质得  $2^{k-1}$  (第  $k$  层第一个结点的编号)

$\leq n < 2^k$  (第  $k+1$  层第一个结点的编号)

即  $k-1 \leq \log_2 n < k$

因为  $k$  只能是整数,因此,  $k = \lfloor \log_2 n \rfloor + 1$  。

**证明：**

设完全二叉树的深度为  $k$ ，

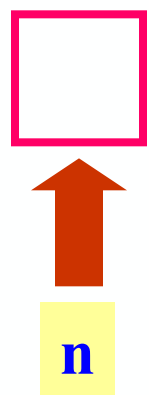
据定义， $k-1$ 层满， $k$ 层最少1个， $n \geq (2^{k-1}-1)+1$ ，

根据第二条性质得  $n \leq 2^k - 1$ ，

因此：  $2^{k-1} \leq n < 2^k$  即  $k-1 \leq \log_2 n < k$

因为  $k$  只能是整数，因此，  $k = \lfloor \log_2 n \rfloor + 1$ 。

**性质4:** 具有 $n$ 个结点的完全二叉树的深度必为 $\lceil \log_2 n \rceil + 1$



← **k-1层**  $2^{k-1} - 1$

← **k层**  $2^k - 1$

$$2^{k-1} - 1 < n \leq 2^k - 1$$

## • 性质 5 :

若对含  $n$  个结点的完全二叉树从上到下且从左至右进行  $1$  至  $n$  的编号，则对完全二叉树中任意一个编号为  $i$  的结点：

(1) 若  $i=1$ ，则该结点是二叉树的根，无双亲，

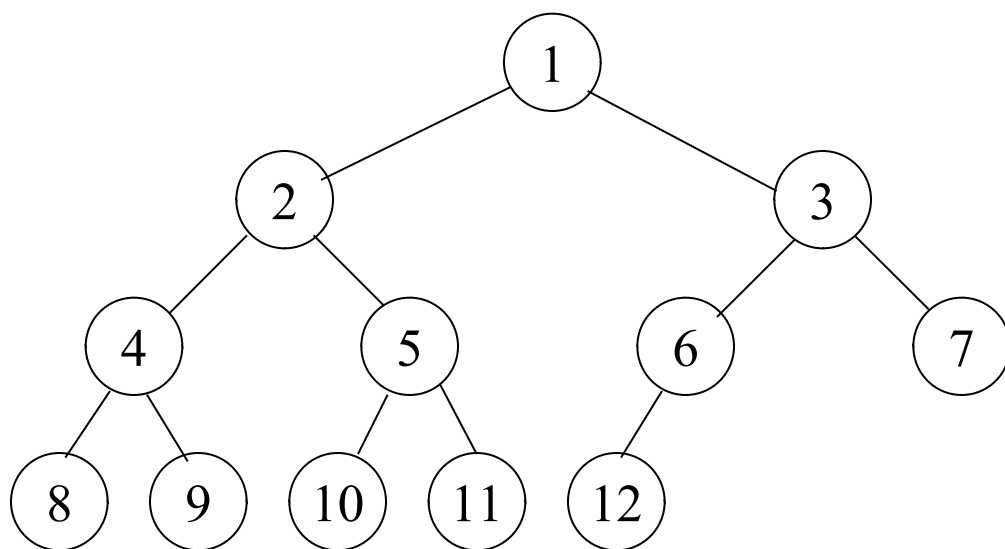
否则，编号为  $\lfloor i/2 \rfloor$  的结点为其**双亲**结点；

(2) 若  $2i > n$ ，则该结点无左孩子，

否则，编号为  $2i$  的结点为其**左孩子**结点；

(3) 若  $2i+1 > n$ ，则该结点无右孩子结点，

否则，编号为  $2i+1$  的结点为其**右孩子**结点。



第 $k$ 层上最后一个结点的编号是 $2^k-1$ ,它的右孩子是第 $k+1$ 层上最后一个结点,编号是 $2^{k+1}-1$ ,右孩子编号是它的双亲结点编号的 $(2^k-1)*2+1$ ,左孩子的编号是 $(2^k-1)*2$

所以, 编号为  $i$  的结点的左孩子结点编号是 $2i$ , 右孩子结点编号是 $2i+1$ ,双亲结点编号 $\lfloor i/2 \rfloor$

# 问 题

具有 $n$ 个结点的非空二叉树的最小深度是多少？最大深度是多少？

答1： 当是满二叉树时深度最小。

最小深度： $\lceil \log_2 n \rceil + 1$

当每个节点都只有左（右）子树时深度最大

最大深度： $n$



## 思考题

1. 具有 $n$ 个结点的完全二叉树中有多少个叶子结点？有多少个度为2的结点？
2. 具有 $n_0$ 个叶子结点的完全二叉树中共有多少个结点？

# 知识回顾

- 树的定义
- 二叉树的定义
- 二叉树的特性

# 二叉树的存储结构

一、二叉树的顺序  
存储表示

二、二叉树的链式  
存储表示

# 一、 二叉树的顺序存储结构

存储方式：用一组地址连续的存储单元依次  
自上而下、自左至右存储完全二叉树上的结  
点元素，即将完全二叉树上编号为 $i$ 的结点元  
素存储在一维数组中下标为 $i-1$ 的分量中。

//-----二叉树的顺序存储表示-----

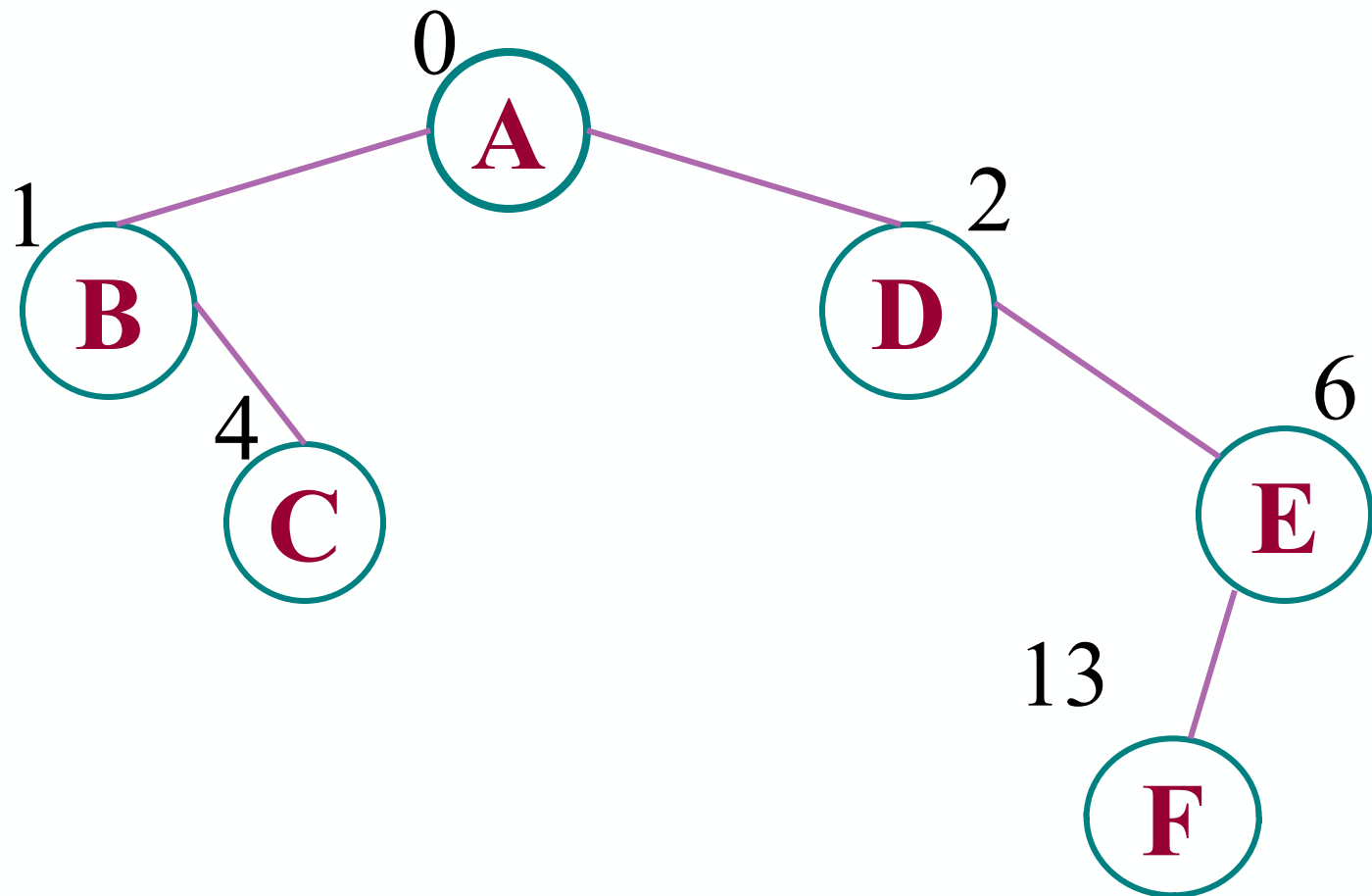
```
#define MAX_TREE_SIZE 100// 二叉树的最大结点数
```

```
typedef TElemType SqBiTree[MAX_TREE_SIZE];
```

```
    // 0号单元存储根结点
```

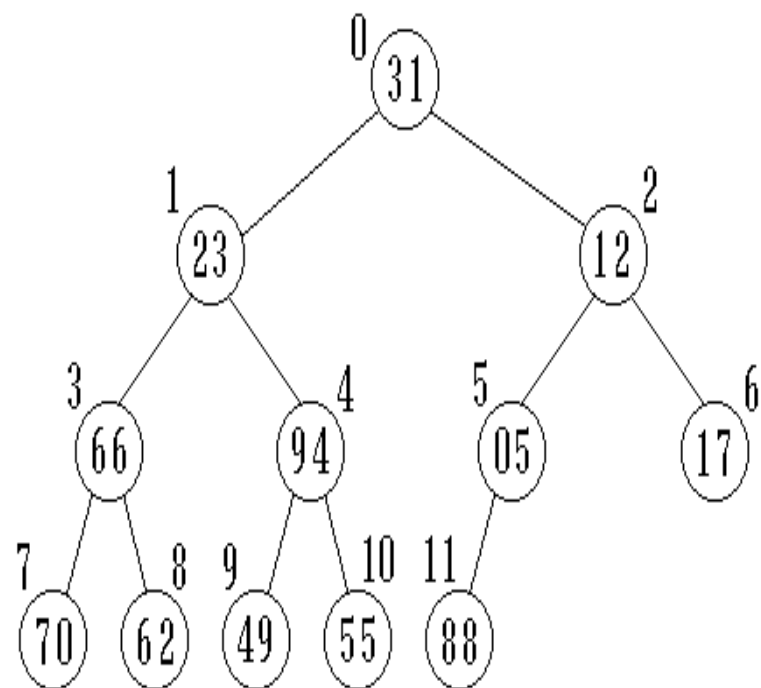
```
SqBiTree bt;
```

例如：



0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	B	D	0	C	0	E	0	0	0	0	0	0	F

0表示不存在此结点

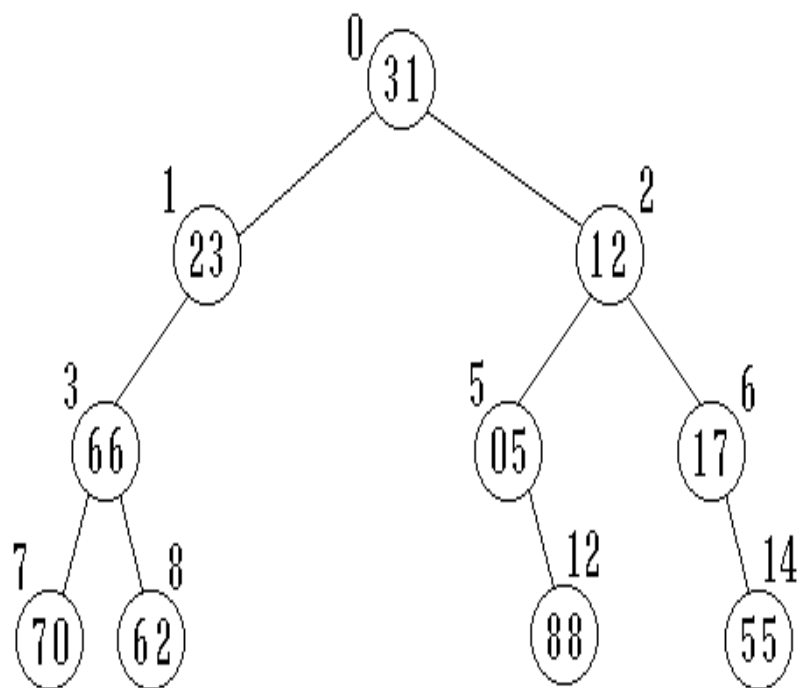


(a)

0	1	2	3	4	5	6	7	8	9	10	11
31	23	12	66	94	05	17	70	62	49	55	88

(b)

**完全二叉树的数组表示**



(a)

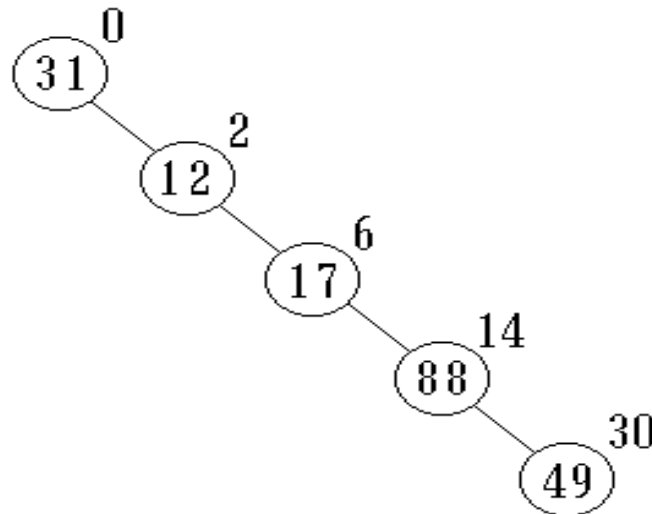
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
31	23	12	66		05	17	70	62				88		55

(b)

**一般二叉树的数组表示**

## 特点:

- 结点间关系蕴含在其存储位置中，浪费空间，适于存满二叉树和完全二叉树（在最坏情况下，深度为 $k$ 且只有 $k$ 个结点的单支树需要长度为 $2^k-1$ 的一维数组）
- 由于一般二叉树必须仿照完全二叉树那样存储，可能会浪费很多存储空间，单支树就是一个极端情况。



## 二、二叉树的链式存储结构

(1) 二叉链表

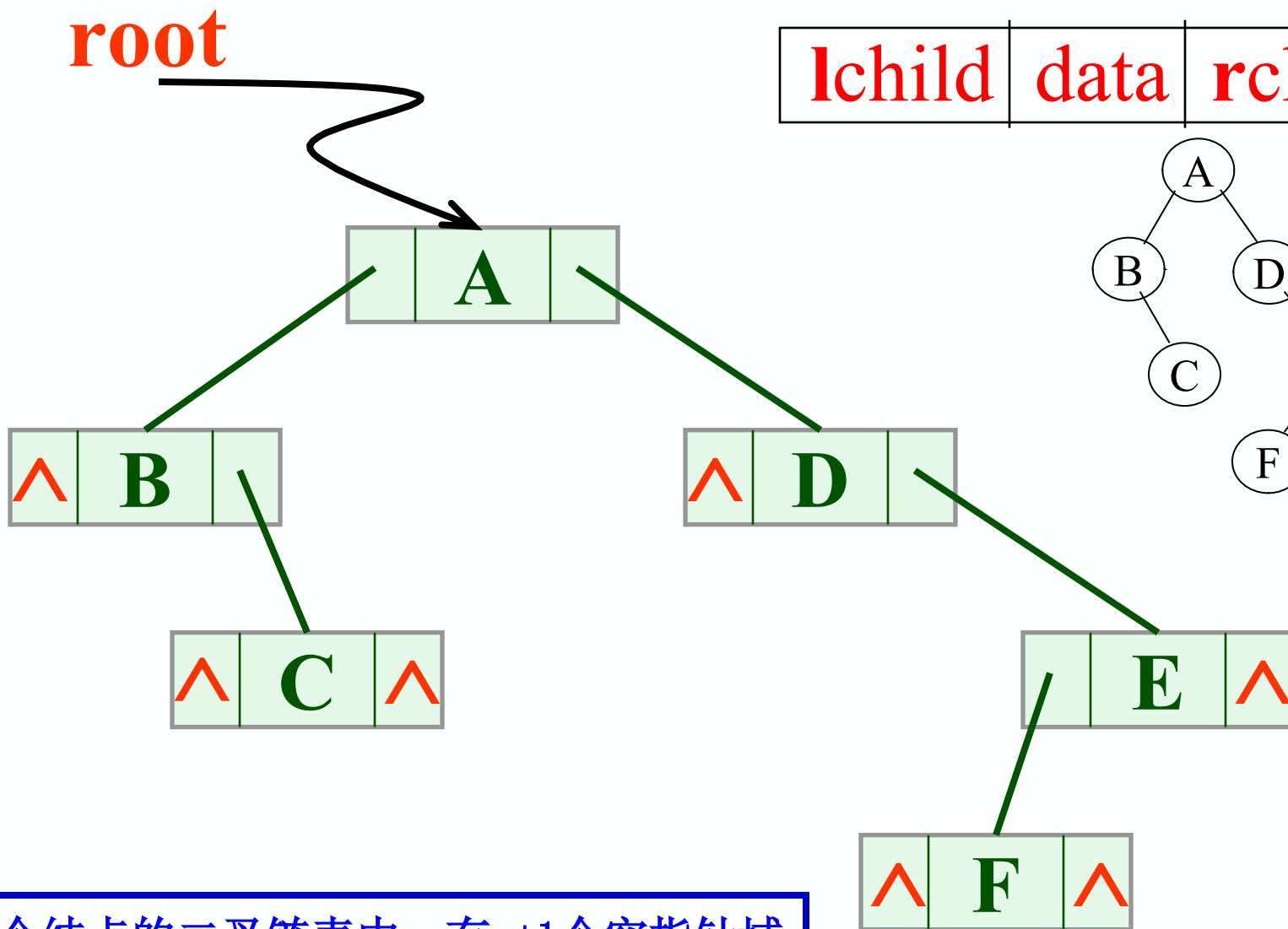
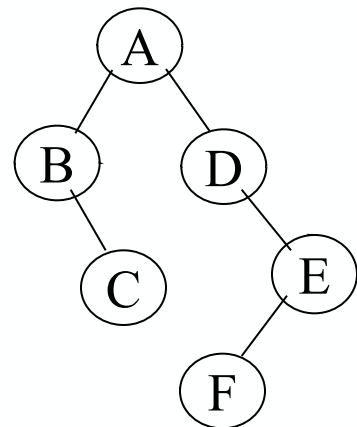
(2) 三叉链表



# 1. 二叉链表

结点结构:

lchild	data	rchild
--------	------	--------



在n个结点的二叉链表中，有n+1个空指针域

C 语言的类型描述如下：

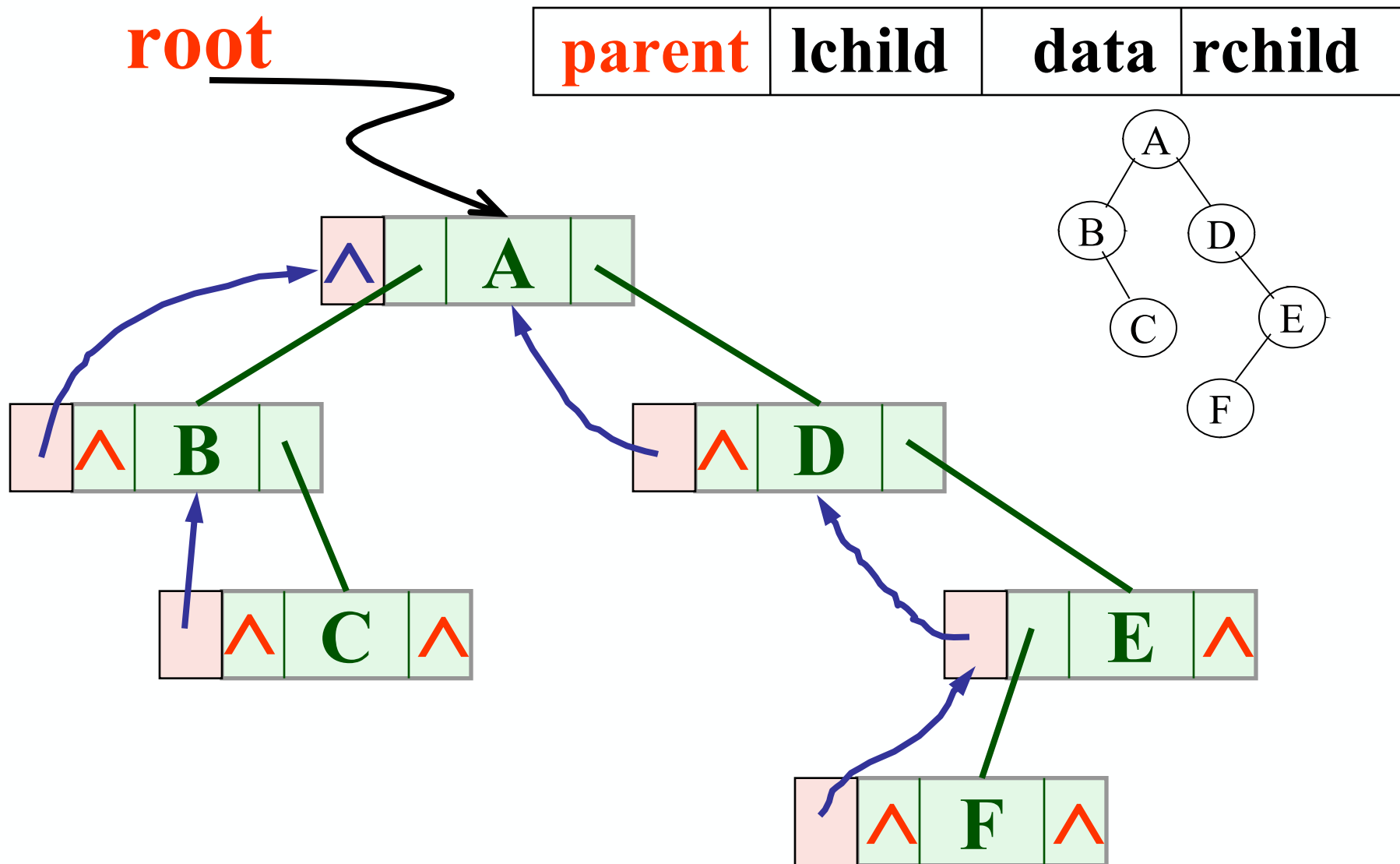
```
typedef struct BiTNode { //结点结构
    TElemType    data;
    struct BiTNode *lchild, *rchild;
    // 左右孩子指针
} BiTNode, *BiTree;
```

结点结构：



## 2. 三叉链表

结点结构:



C 语言的类型描述如下：

```
typedef struct TriTNode { // 结点结构
    TElemType    data;
    struct TriTNode *lchild, *rchild;
                        // 左右孩子指针
    struct TriTNode *parent; //双亲指针
} TriTNode, *TriTree;
```

结点结构：

parent	lchild	data	rchild
--------	--------	------	--------



## 5.3.1

# 二叉树的遍历

一、问题的提出

二、先左后右的遍历算法

三、算法的递归描述

四、中序遍历算法的非递归描述

五、遍历算法的应用举例

# 一、问题的提出

顺着某一条搜索路径**巡访**二叉树中的结点，使得每个结点**均被访问一次**，而且**仅被访问一次**。

“**访问**”的含义可以很广，如：输出结点的信息等。

“**遍历**”是任何类型均有的操作，对线性结构而言，只有一条搜索路径(因为每个结点均只有一个后继)，故不需要另加讨论。而二叉树是非线性结构，**每个结点有两个后继**，则**存在如何遍历**即按什么样的**搜索路径**遍历的问题。

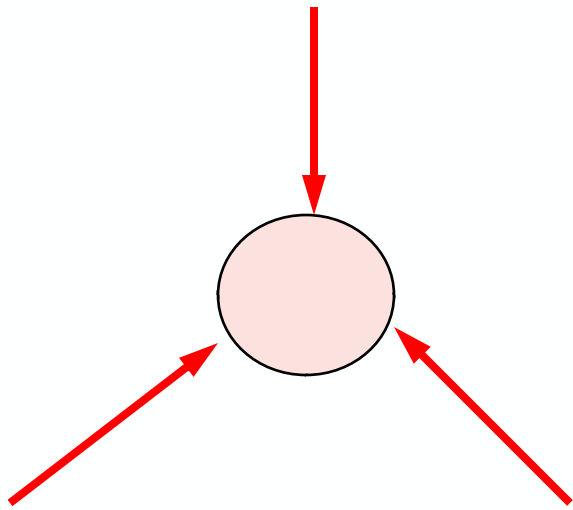


对“二叉树”而言，可以有三条搜索路径：

1. **先上后下**的按层次遍历；
2. **先左**（子树）**后右**（子树）的遍历；
3. **先右**（子树）**后左**（子树）的遍历。



## 二、先左后右的遍历算法



先（根）序的遍历算法

中（根）序的遍历算法

后（根）序的遍历算法

## ● 先（根）序的遍历算法：

若二叉树为空树，则空操作；否则，

- （1）访问根结点；
- （2）先序遍历左子树；
- （3）先序遍历右子树。

## ● 中（根）序的遍历算法：

若二叉树为空树，则空操作；否则，

（1）中序遍历左子树；

（2）访问根结点；

（3）中序遍历右子树。

## ● 后（根）序的遍历算法：

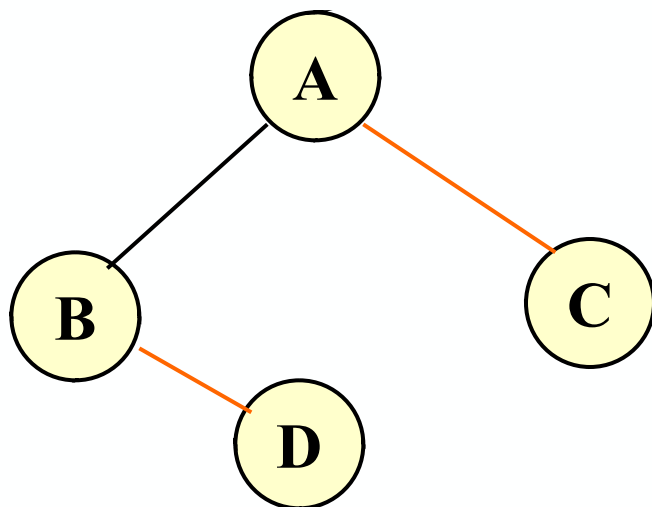
若二叉树为空树，则空操作；否则，

（1）后序遍历左子树；

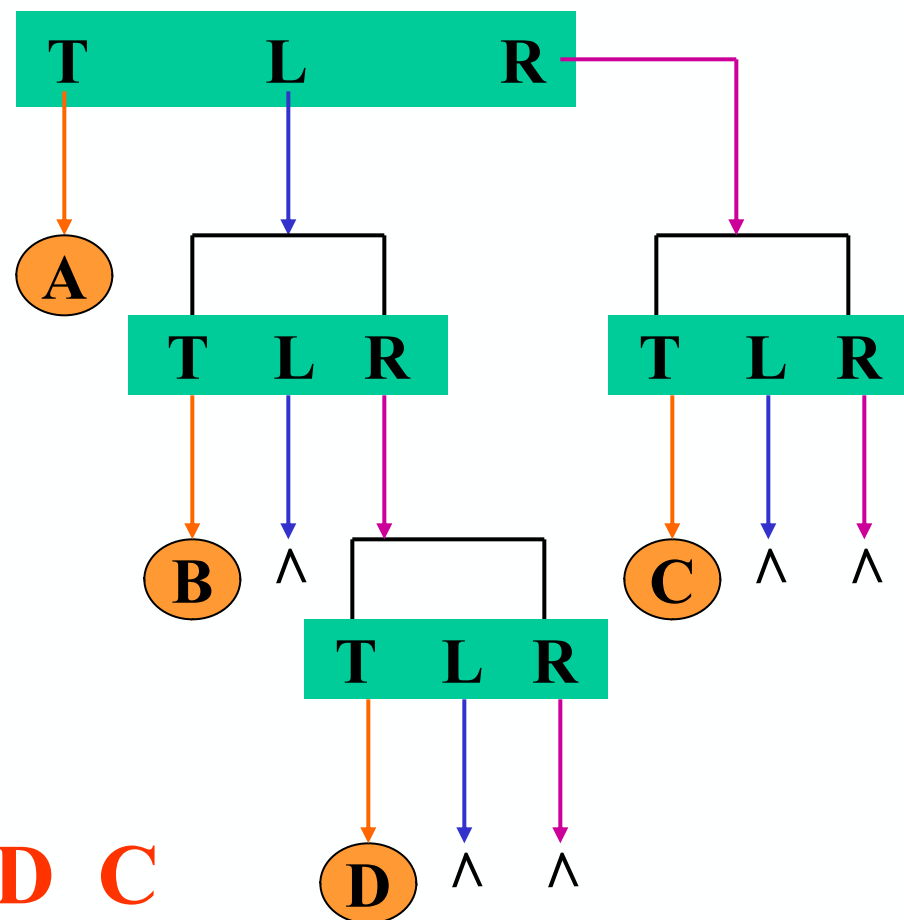
（2）后序遍历右子树；

（3）访问根结点。

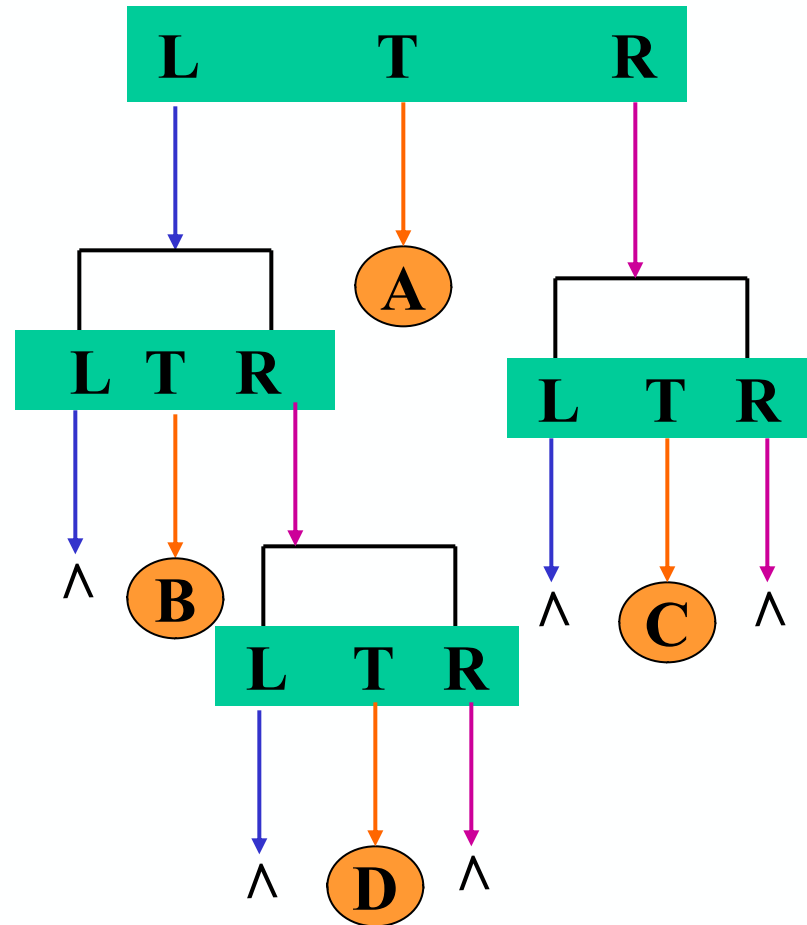
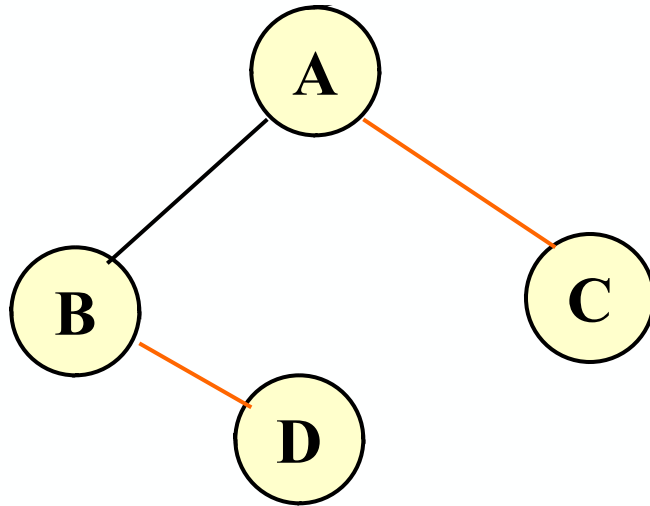
先序遍历:



先序遍历序列: A B D C

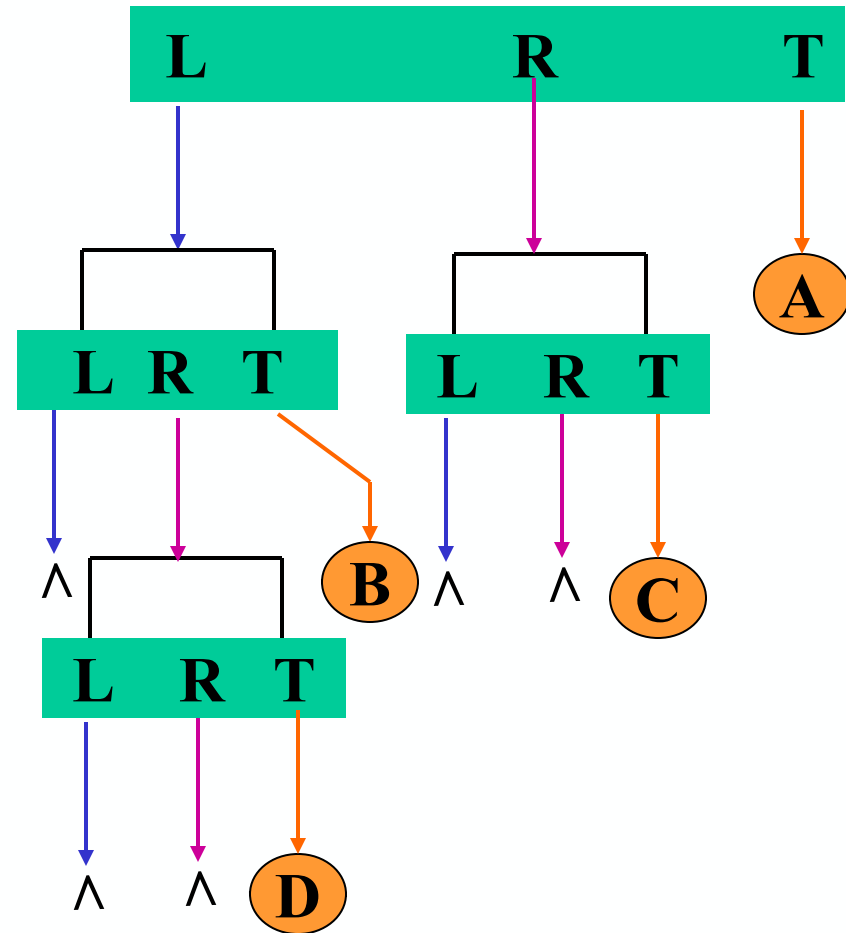
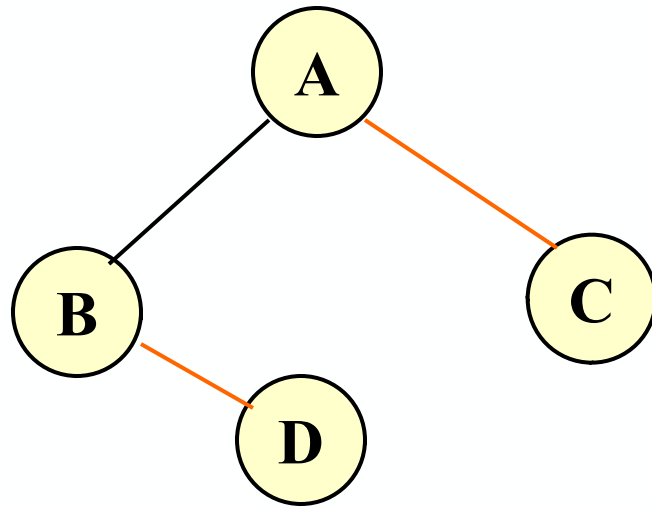


中序遍历:



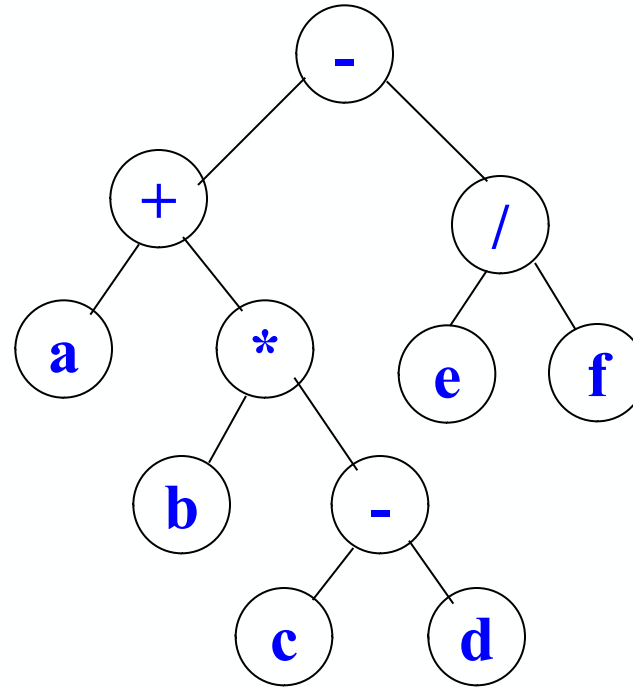
中序遍历序列: B D A C

## 后序遍历:



后序遍历序列: **D B C A**





先序遍历:     - + a \* b - c d / e f  
 中序遍历:     a + b \* c - d - e / f  
 后序遍历:     a b c d - \* + e f / -

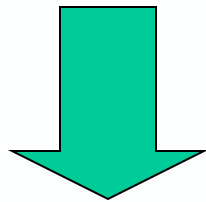


# 遍历的算法实现——用递归形式格外简单！

回忆：

```
long Factorial ( long n ) {  
    if ( n == 0 ) return 1; //基本项  
    else return n * Factorial (n-1); //归纳项}
```

则三种遍历算法可写出：



```
Status PreOrderTraverse(BiTree T){  
    if(T==NULL) return OK; //空二叉树  
    else{  
        cout<<T->data; //访问根结点  
        PreOrderTraverse(T->lchild); //递归遍历左子树  
        PreOrderTraverse(T->rchild); //递归遍历右子树  
    }  
}
```

```

Status PreOrderTraverse(BiTree T){
    if(T==NULL) return OK; else{
        cout<<T->data;
        PreOrderTraverse(T->lchild);
        PreOrderTraverse(T->rchild); }
    }

```

先序序列: **ABDC**

左是空返回

左是空返回  
右是空返回

左是空返回  
右是空返回

主程序

Pre( T )

T → A

printf(A);  
pre(T → L);  
pre(T → R);

T → B

printf(B);  
pre(T → L);  
pre(T → R);

T → C

printf(C);  
pre(T → L);  
pre(T → R);

T → Λ

返回

T → D

printf(D);  
pre(T → L);  
pre(T → R);

T → Λ

返回

T → Λ

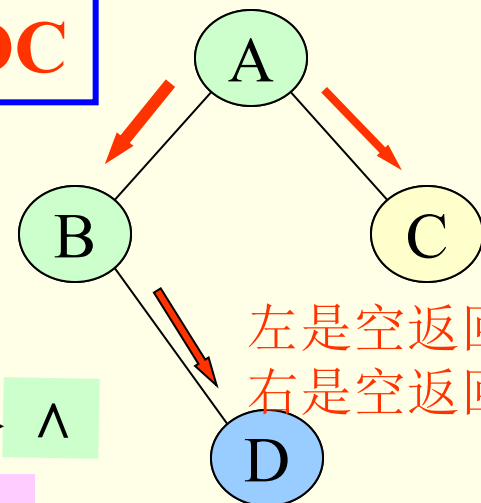
返回

T → Λ

返回

T → Λ

返回

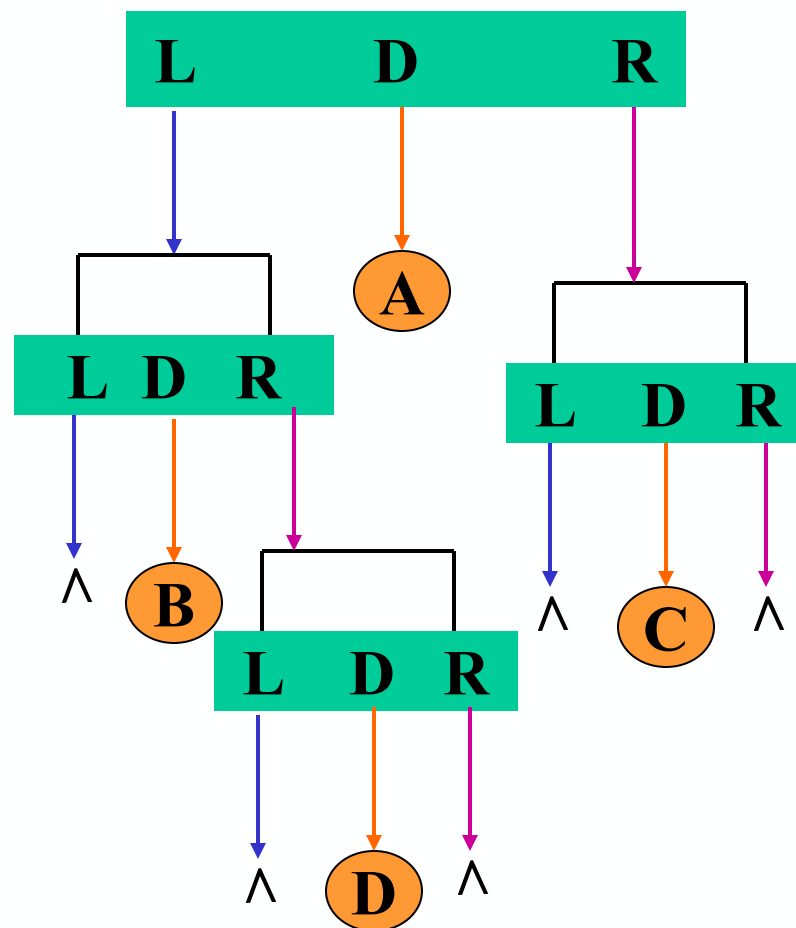
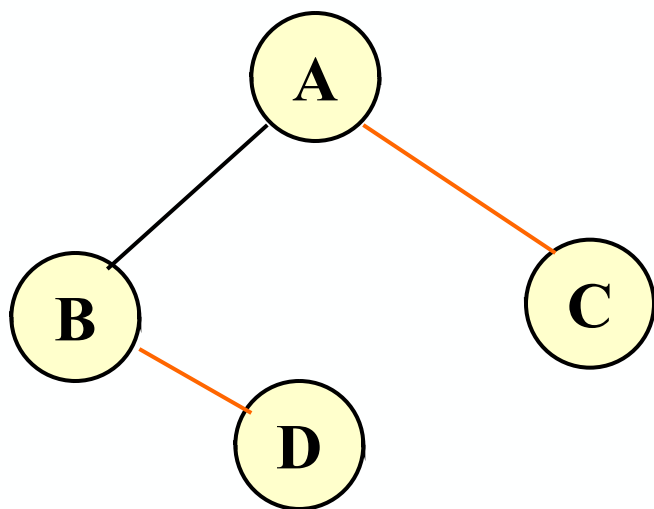


若二叉树为空，则空操作  
否则：

中序遍历左子树 (L)

访问根结点 (D)

中序遍历右子树 (R)



中序遍历序列: B D A C

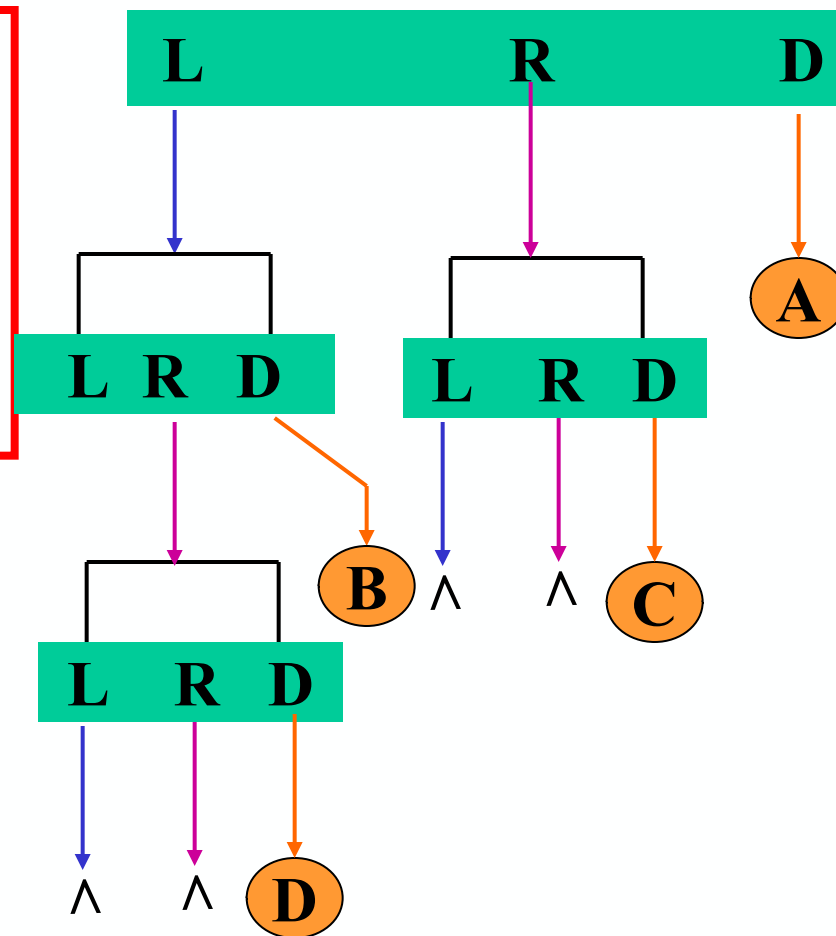
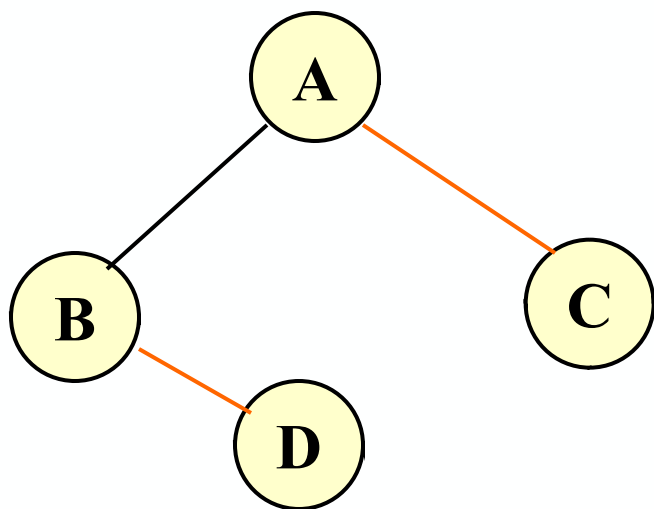
```
Status InOrderTraverse(BiTree T){  
    if(T==NULL) return OK; //空二叉树  
    else{  
        InOrderTraverse(T->lchild); //递归遍历左子树  
        cout<<T->data; //访问根结点  
        InOrderTraverse(T->rchild); //递归遍历右子树  
    }  
}
```

若二叉树为空，则空操作  
否则

后序遍历左子树 (L)

后序遍历右子树 (R)

访问根结点 (D)



后序遍历序列: **D B C A**

```
Status PostOrderTraverse(BiTree T){  
    if(T==NULL) return OK; //空二叉树  
    else{  
        PostOrderTraverse(T->lchild); //递归遍历左子树  
        PostOrderTraverse(T->rchild); //递归遍历右子树  
        cout<<T->data; //访问根结点  
    }  
}
```



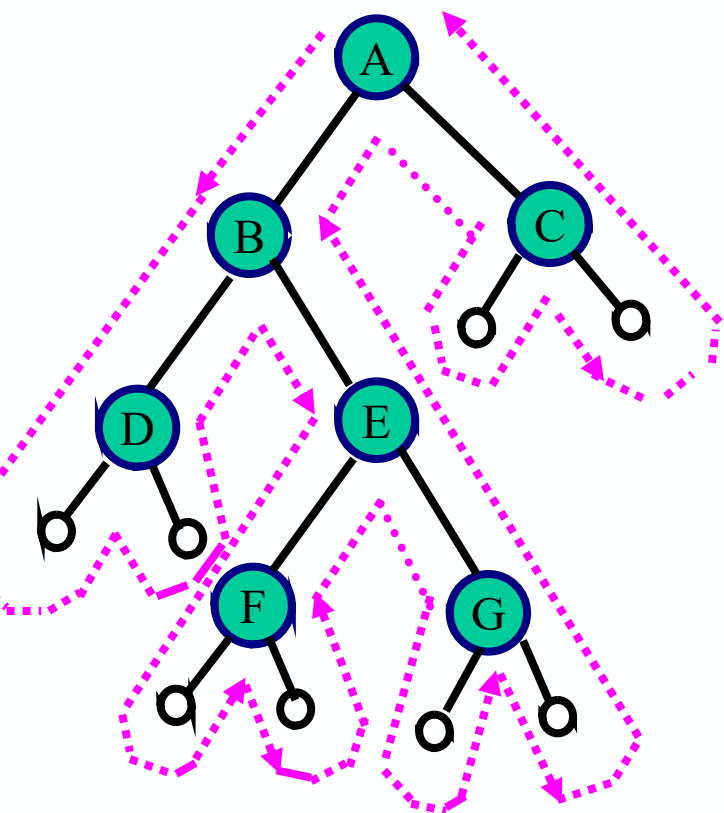
# 遍历算法的分析

```
Status PreOrderTraverse(BiTree T){  
    if(T==NULL) return OK;  
    else{  
        cout<<T->data;  
        PreOrderTraverse(T->lchild);  
        PreOrderTraverse(T->rchild);  
    }  
}
```

```
Status InOrderTraverse(BiTree T){  
    if(T==NULL) return OK;  
    else{  
        InOrderTraverse(T->lchild);  
        cout<<T->data;  
        InOrderTraverse(T->rchild);  
    }  
}
```

```
Status PostOrderTraverse(BiTree T){  
    if(T==NULL) return OK;  
    else{  
        PostOrderTraverse(T->lchild);  
        PostOrderTraverse(T->rchild);  
        cout<<T->data;  
    }  
}
```

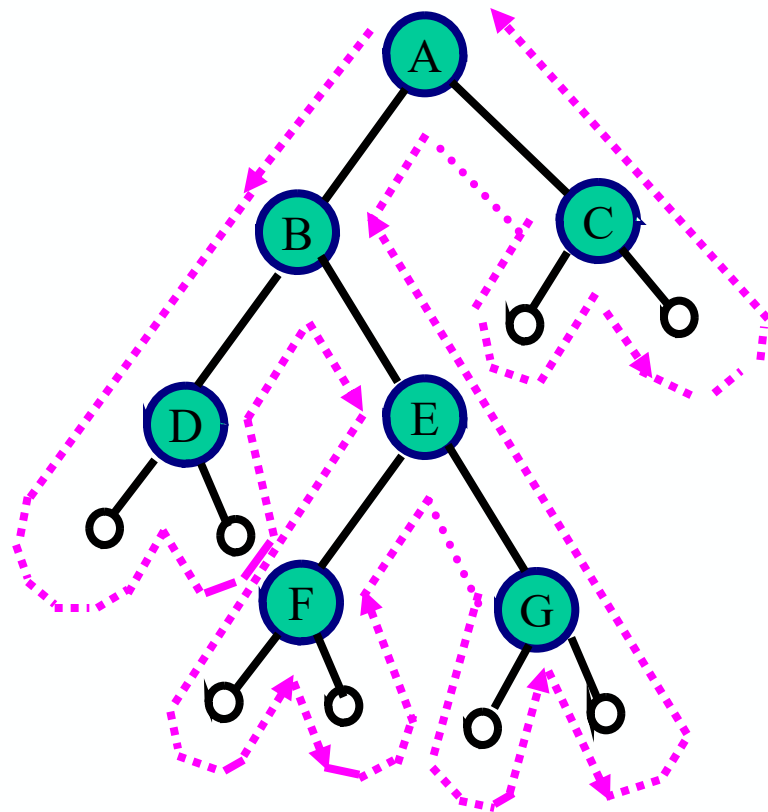
如果去掉输出语句，从递归的角度看，三种算法是完全相同的，或说这三种算法的访问路径是相同的，只是访问结点的时机不同。



从虚线的出发点到终点的路径上，每个结点经过**3次**。

第**1次**经过时访问 = **先序**遍历  
第**2次**经过时访问 = **中序**遍历  
第**3次**经过时访问 = **后序**遍历

时间效率: $O(n)$  //每个结点只访问一次  
空间效率: $O(n)$  //栈占用的最大辅助空间

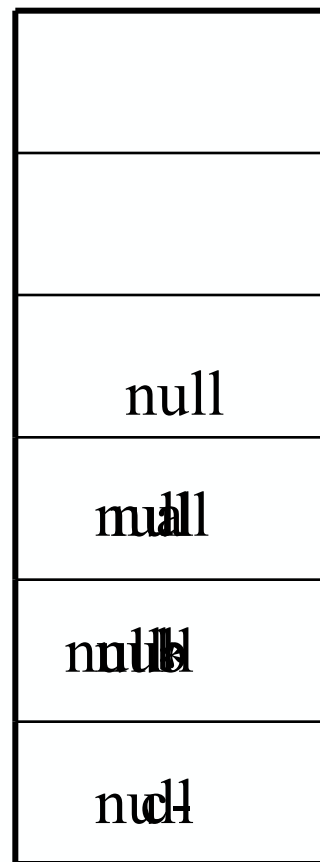
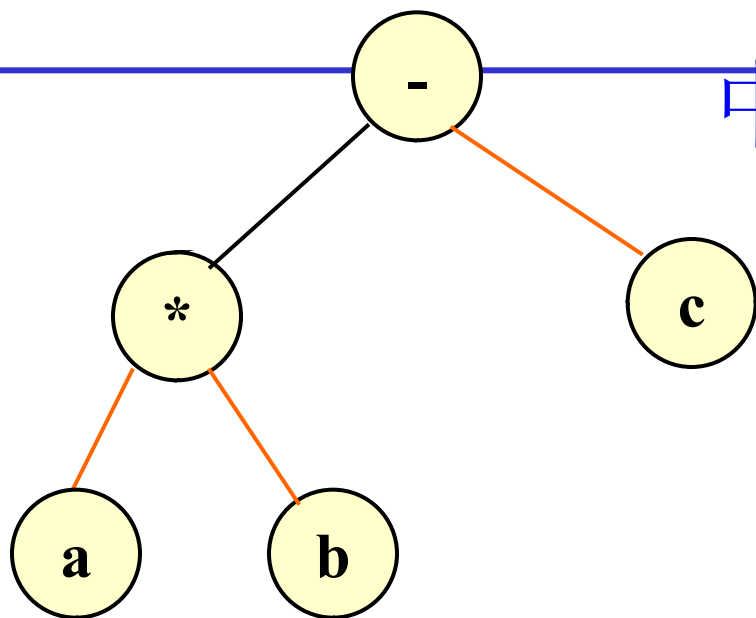


## 四、中序遍历算法的非递归描述

```
Status InOrderTraverse(BiTree T, Status(*Visit)(TElemType e))
{
    InitStack(S);
    Push(S, T); //根指针进栈
    While(!StackEmpty(S)){
        While(GetTop(S, p) && p) Push(S, p->lchild); //向左走到尽头
        Pop(S, p); //空指针退栈
        if(!StackEmpty(S)){ //访问结点,向右一步
            Pop(S, p);
            Visit(p->data);
            Push(S, p->rchild);
        } //if
    } //while
    Return OK;
} //InOrderTraverse
```

算法1

中序遍历序列:  $a * b - c$



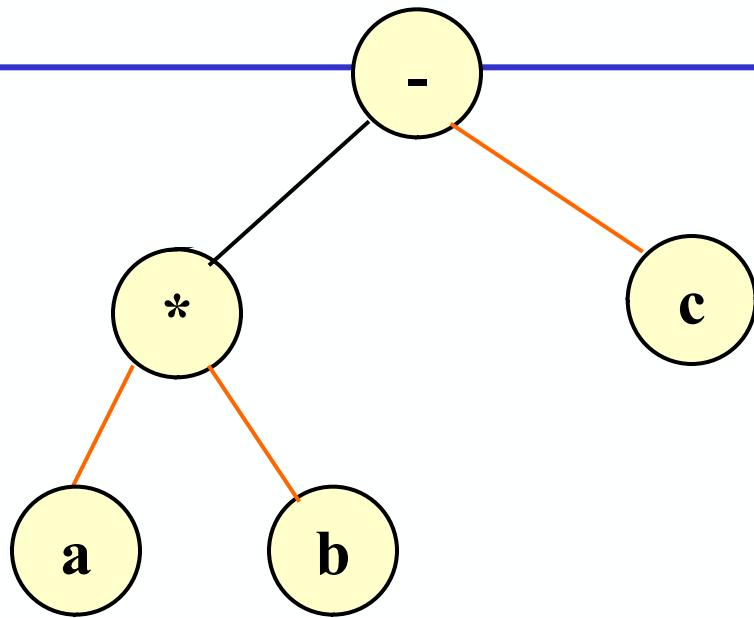
```

While(!StackEmpty(S)){
    While(GetTop(S,p)&&p) Push(S,p->lchild);
    //向左走到尽头
    Pop(S,p); //空指针退栈
    if(!StackEmpty(S)){//访问结点,向右一步
        Pop(S,p);
        if(!Visit(p->data)) return ERROR;
        Push(S,p->rchild);
    }//if
}
    
```

# 中序遍历算法的非递归描述

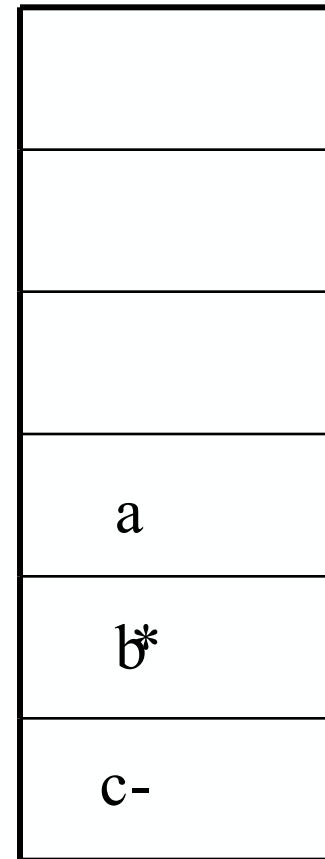
```
status InorderTraverse(BiTree T, status (*Visit)(TelemType e)){
    InitStack(S);
    p=T;
    while(p||!StackEmpty(S)){
        if (p) {Push(S,p); p=p->lchild;}
        //根指针进栈, 遍历左子树
        else{//根指针退栈,访问根结点,遍历右子树
            Pop(S,p);
            Visit(p->data);
            p=p->rchild;
        } // else
    }// while
    Return OK;
}//InorderTraverse
```

算法2

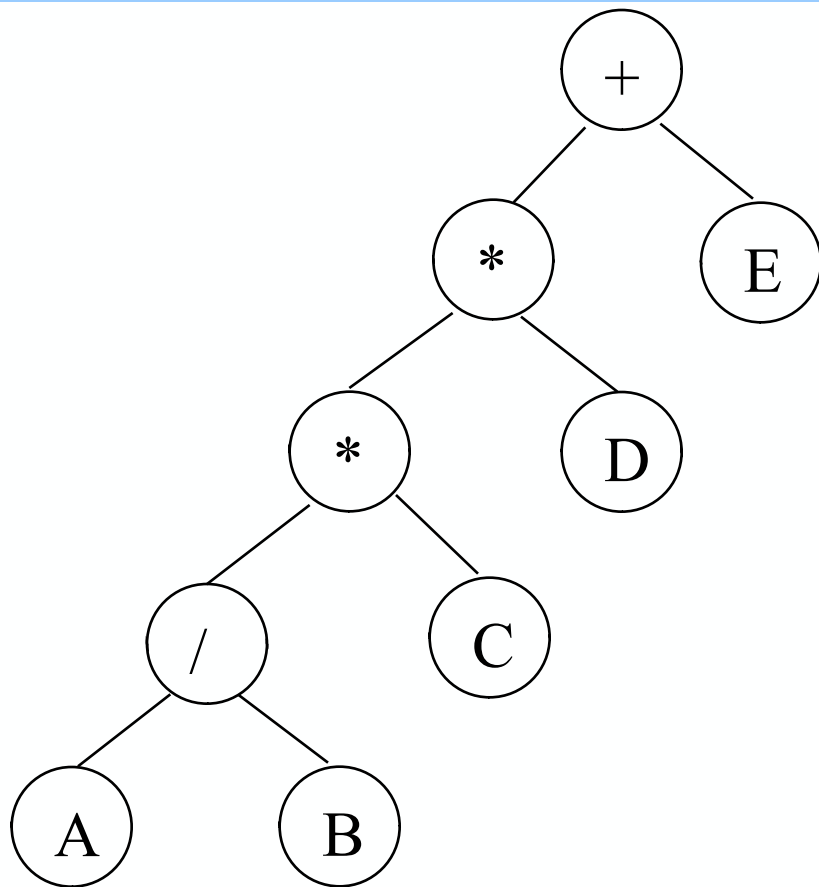


```

p=T;
while(p&&!StackEmpty(S)){
    if (p) {Push(S,p); p=p->lchild;}
    //根指针进栈,遍历左子树
    else{//根指针退栈,访问根结点,遍历右
子树
        Pop(S,p);
        if(!Visit(p->data))return ERROR;
        p=p->rchild;
    } // else
} // while
    
```



# 用二叉树表示算术表达式



先序遍历结果

**+ \* \* / A B C D E**

—前缀表示法

中序遍历结果

**A / B \* C \* D + E**

—中缀表示法

后序遍历结果

**A B / C \* D \* E +**

—后缀表示法

层次遍历结果

**+ \* E \* D / C A B**



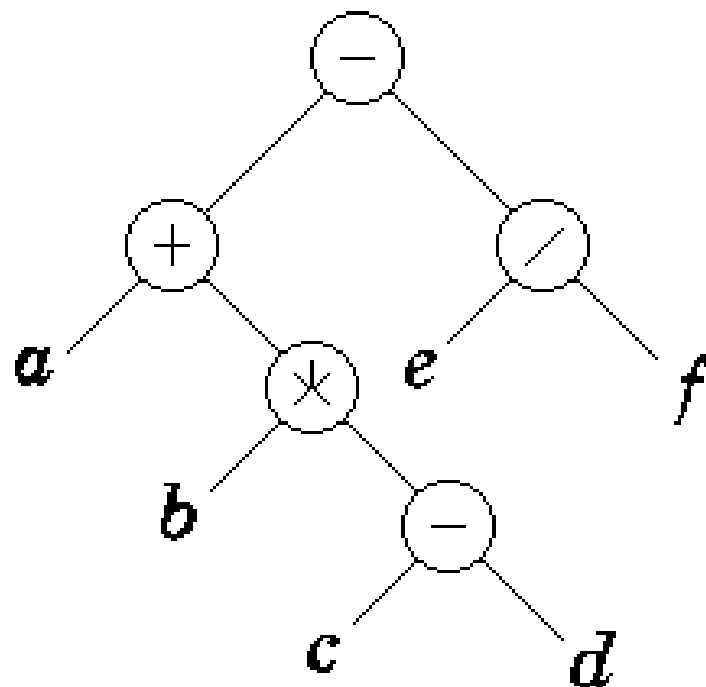
# 层序遍历

层序遍历二叉树算法的框架是

- 若二叉树为空，则空操作；
- 否则，如队列不空，循环：  
根结点入队，并作为当前结点；  
将当前结点的左右孩子入队；  
做出队操作，队首元素作为当前结点
- 最后，出队序列就是层序遍历序列

遍历结果

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



## 层次遍历算法(需要利用队列)

```
void LayerOrder(Bitree T) //层序遍历二叉树
{ if (T)
{
    InitQueue(Q);           //建一个空队（初始化队列）
    EnQueue(Q,T);           //将一个结点插入队尾的函数
    while( !QueueEmpty(Q) )
    { DeQueue(Q, &p);        //队首结点出队(送入p)
      visit(p);
      if(p->lchild) EnQueue(Q,p->lchild); //p的左孩子入队
      if(p->rchild) EnQueue(Q,p->rchild); //p的右孩子入队
    }
}
}
} //LayerOrder
```

当孩子为空时不要  
将空指针入队



# 遍历算法思想的应用---步骤

- 要明确所要编写的算法的功能描述（包括所涉及的各参数或变量的含义）。在此基础上按如下步骤讨论算法的实现：
  - ❖ 如果T为空，则按预定功能实现对空树的操作，以满足功能要求（包括对相应参数，变量的操作）。
  - ❖ 否则，假设算法对T的左右子树都能分别实现预定功能，在此基础上，通过按预定要求适当调用对左右子树的算法的功能，及对当前结点的操作实现对整个二叉树的功能（包括对各变量、参数的操作）。

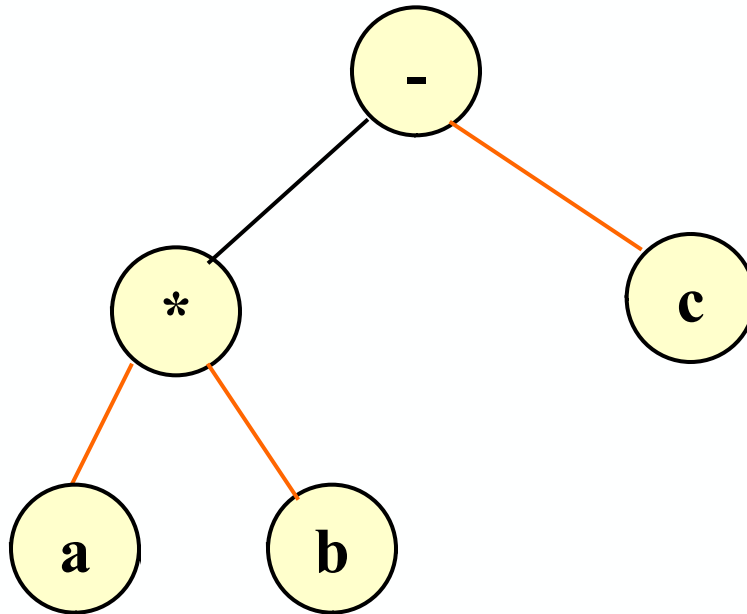
例1：统计二叉树中叶子结点的个数. (先序遍历)

## 算法基本思想：

先序(或中序或后序)遍历二叉树，在遍历过程中查找叶子结点，并计数。

由此，需在遍历算法中增添一个“计数”的参数，并将算法中“访问结点”的操作改为：若是叶子，则计数器增1。

```
void CountLeaf (BiTree T, int& count) {  
    if ( T ) {  
        if ((!T->lchild)&& (!T->rchild))  
            count++;    // 对叶子结点计数  
        CountLeaf( T->lchild, count);  
        CountLeaf( T->rchild, count);  
    } // if  
} // CountLeaf
```



# 例2：求二叉树的深度 (后序遍历)



## 算法基本思想：

首先分析**二叉树的深度**和它的**左、右子树深度**之间的关系。

从二叉树深度的定义可知，二叉树的深度应为其左、右子树深度的最大值加1。由此，需先分别求得左、右子树的深度，算法中“访问结点”的操作为：求得左、右子树深度的最大值，然后加 1 。

```
— int Depth (BiTree T ){ // 返回二叉树的深度
    if ( !T )    depthval = 0;
    else  {
        depthLeft = Depth( T->lchild );
        depthRight= Depth( T->rchild );
        depthval = 1 + (depthLeft > depthRight ?
                        depthLeft : depthRight);
    }
    return depthval;
}
```

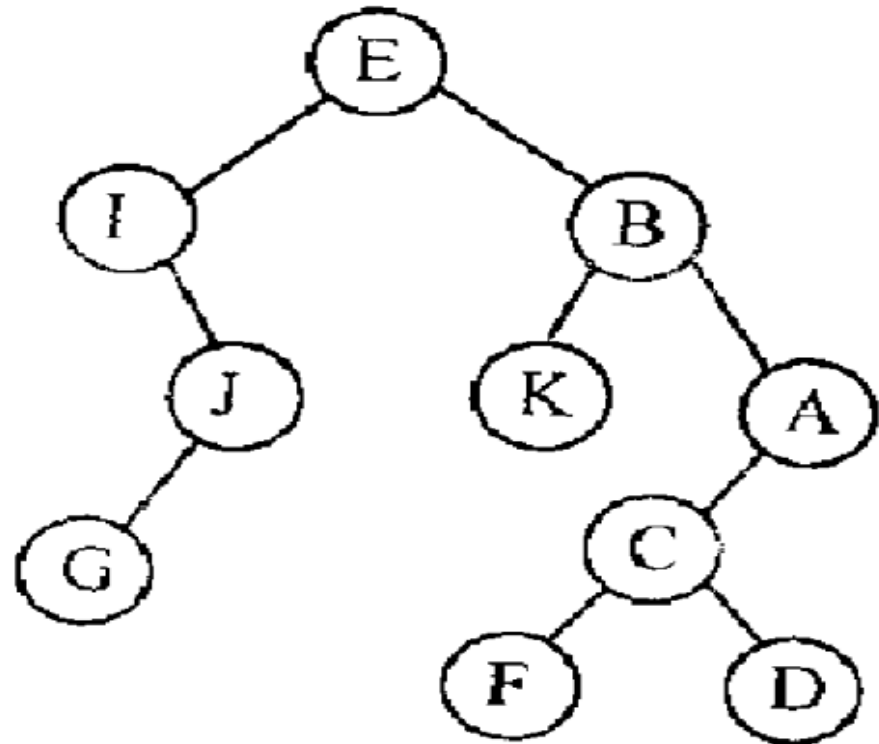
# 知识回顾

(week 8)

- 二叉树的遍历：以一定的次序排列二叉树的结点。--非线性结构的线性化。
- 传统的二叉链表存储结构只存储了结点的左右孩子的信息，没有存储其前趋和后继信息。
- 这些信息只有在遍历过程中才能得到。
- 传统的二叉树遍历方法：--利用堆栈  
效率低

## 练习

- 中序遍历序列
- 后序遍历序列
- 先序遍历序列



# 遍历算法思想的应用

# 例3：建立二叉树的存储结构

# 以字符串的形式 根 左子树 右子树

## 定义一棵二叉树

例如：

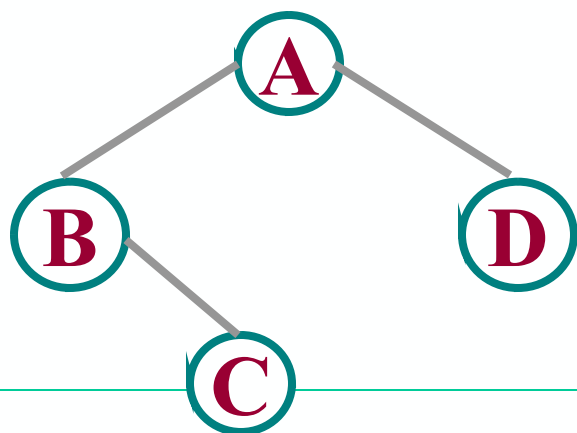
空树

以空白字符 “■” 表示

只含一个根结点的  
的二叉树



以字符串 “A■■” 表示



以下列字符串表示

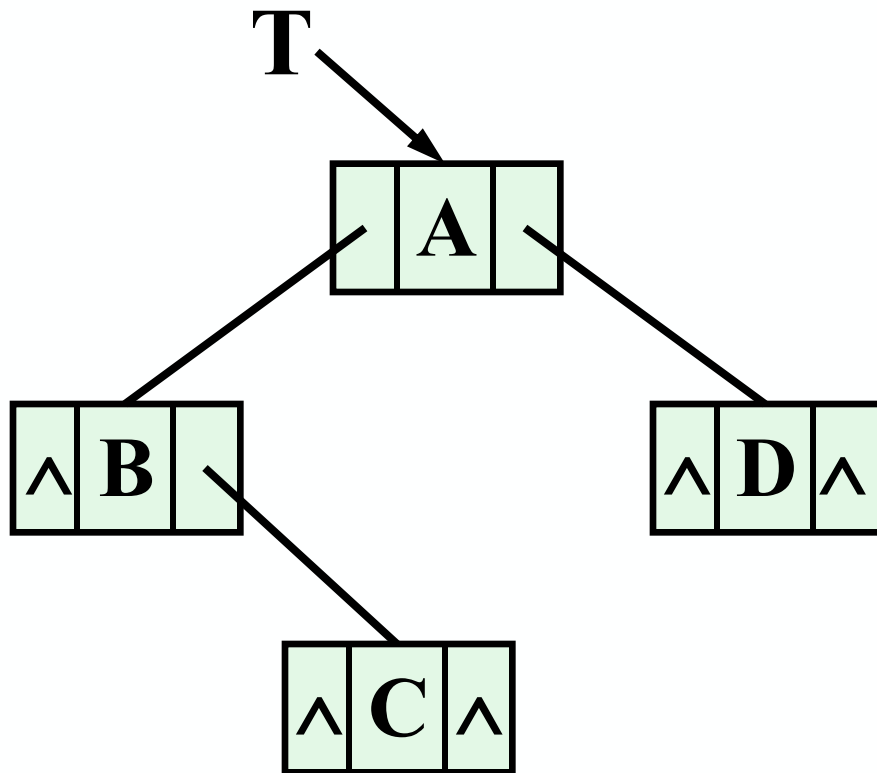
A(B(■,C(■,■)),D(■,■))

```
Status CreateBiTree(BiTree &T) {  
    scanf(&ch);  
    if (ch==' ') T = NULL;  
    else {  
        if (!(T = (BiTNode *)malloc(sizeof(BiTNode))))  
            exit(OVERFLOW);  
        T->data = ch;           // 生成根结点  
        CreateBiTree(T->lchild); // 构造左子树  
        CreateBiTree(T->rchild); // 构造右子树  
    }  
    return OK; } // CreateBiTree
```



上页算法执行过程举例如下:

A B ■ C ■ ■ D ■ ■



```

Status CreateBiTree(BiTree &T) {
    scanf(&ch);
    if (ch==' ') T = NULL;
    else {
        if (!(T = (BiTNode
        *)malloc(sizeof(BiTNode))))
            exit(OVERFLOW);
        T->data = ch;
        CreateBiTree(T->lchild);
        CreateBiTree(T->rchild);
    }
    return OK; }
  
```

- 设计递归函数统计二叉树T中的结点数据域等于k的结点数

```
int count (BiTree *bt)
{
if (bt==NULL)  return 0 ;
else
    return count (bt->lchild)+ count (bt-
>rchild)+(bt->data==k);
}
```

# 作业

- 设计算法统计二叉树中结点的个数。

# 主要内容

5.1 树的定义和基本术语

5.2 二叉树

5.3 遍历二叉树和线索二叉树

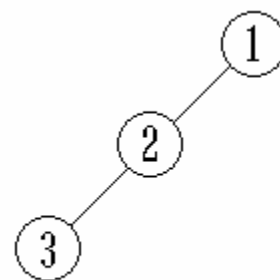
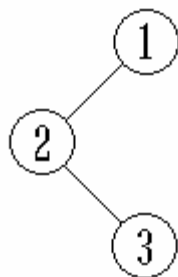
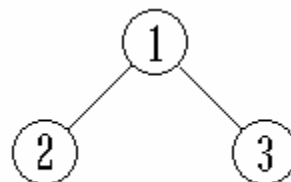
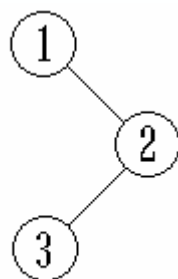
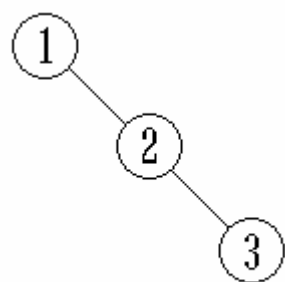
5.4 树和森林

5.5 霍夫曼树及其应用

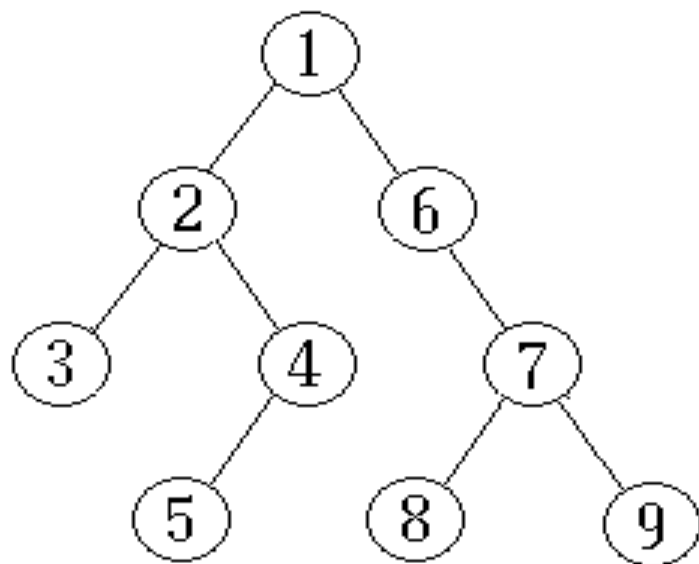
# 问题

已知一棵二叉树的先序遍历序列，能否得到这棵树？

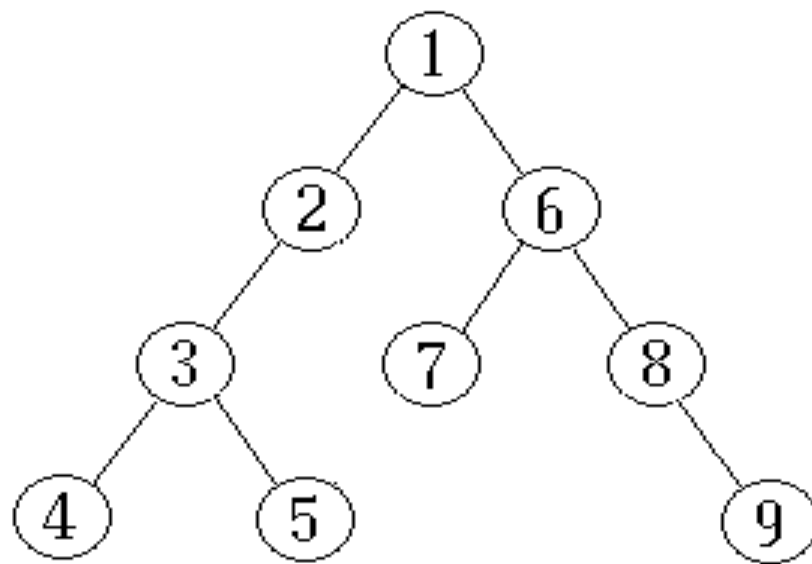
例：一棵树的先序序列为：1， 2， 3。请画出这棵树。



如果先序序列固定不变，给出不同的中序序列，可得到不同的二叉树。



(a)



(b)

先序序列: 1, 2, 3, 4, 5, 6, 7, 8, 9

中序序列a: 3, 2, 5, 4, 1, 6, 8, 7, 9

中序序列b: 4, 3, 5, 2, 1, 7, 6, 8, 9

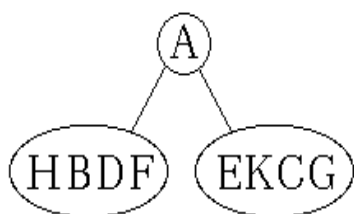


## 结 论

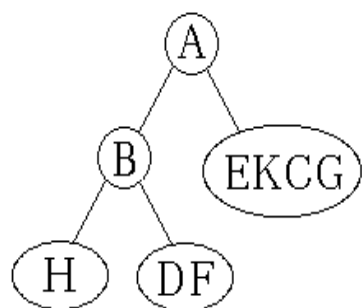
仅已知一棵二叉树的先序遍历序列，**不能**唯一确定这棵树。

**可以证明：**一棵二叉树的**先序序列和中序序列**可以唯一的确定这棵二叉树。

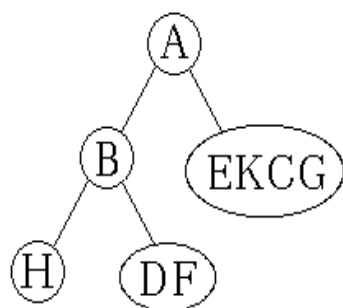
由二叉树的先序序列和中序序列可唯一地确定一棵二叉树。例, 先序序列 { **ABHFDECKG** } 和中序序列 { **HBDFAEKCG** }, 构造二叉树过程如下:



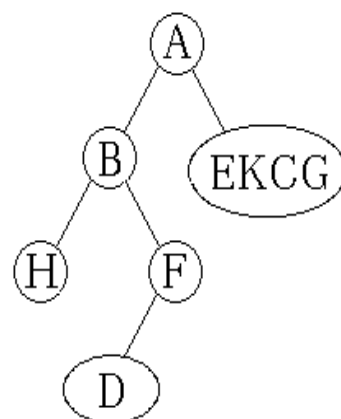
(a) 取A



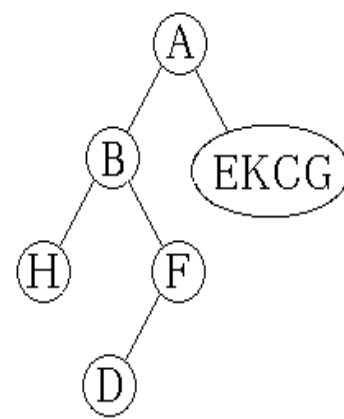
(b) 取B



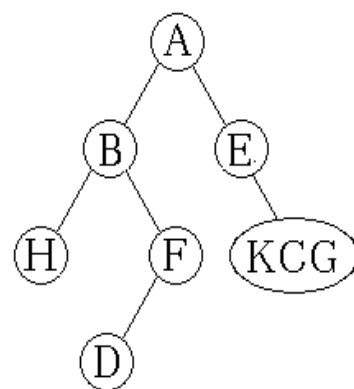
(c) 取H



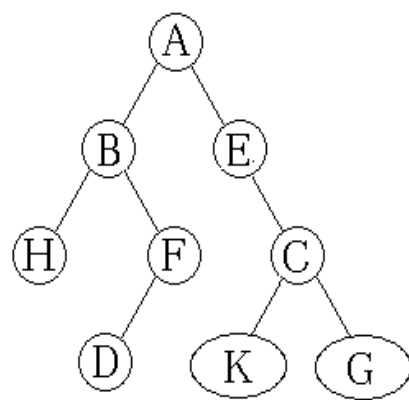
(d) 取F



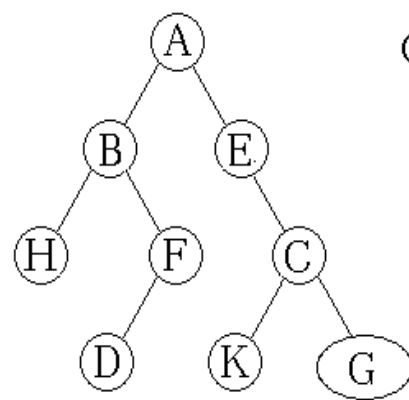
(e) 取D



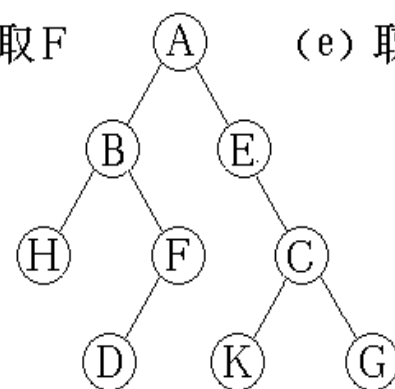
(f) 取E



(g) 取C



(h) 取K

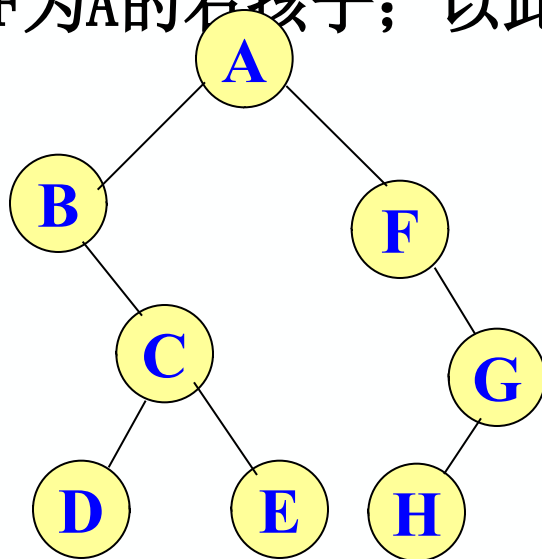


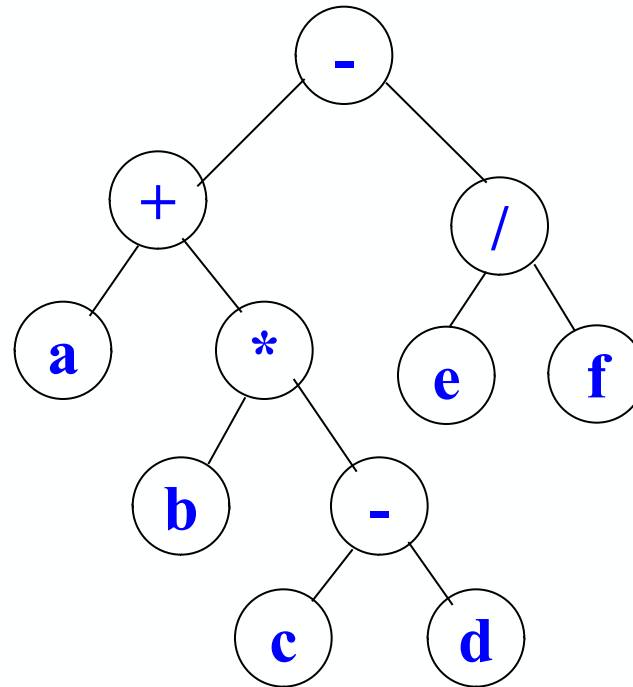
(i) 取G

已知一棵二叉树的中序序列和后序序列分别是BDCEAFHG 和 DECBHGFA，请画出这棵二叉树。

**分析：**

- ①由后序遍历特征，根结点必在后序序列尾部（即A）；
- ②由中序遍历特征，根结点必在中间，而且其左部必全部是左子树的子孙（即BDCE），其右部必全部是右子树的子孙（即FHG）；
- ③继而，根据后序中的DECB子树可确定B为A的左孩子，根据HGF子串可确定F为A的右孩子；以此类推。





先序遍历:  $- + a * b - c d / e f$

中序遍历:  $a + b * c - d - e / f$

后序遍历:  $a b c d - * + e f / -$

**问题：**能否在遍历过程中把结点的前趋和后继信息保存下来，并且提高算法的效率呢？

## 5.3.2

# 线索二叉树

Threaded Binary Tree

# 主要内容

- (1) 什么是线索二叉树
- (2) 基于线索二叉树的遍历方法
- (3) 如何建立线索二叉树

# 线索二叉树的引入和定义

用二叉链表法存储包含 $n$ 个结点的二叉树，结点的指针区域中会有 $n+1$ 个空指针。

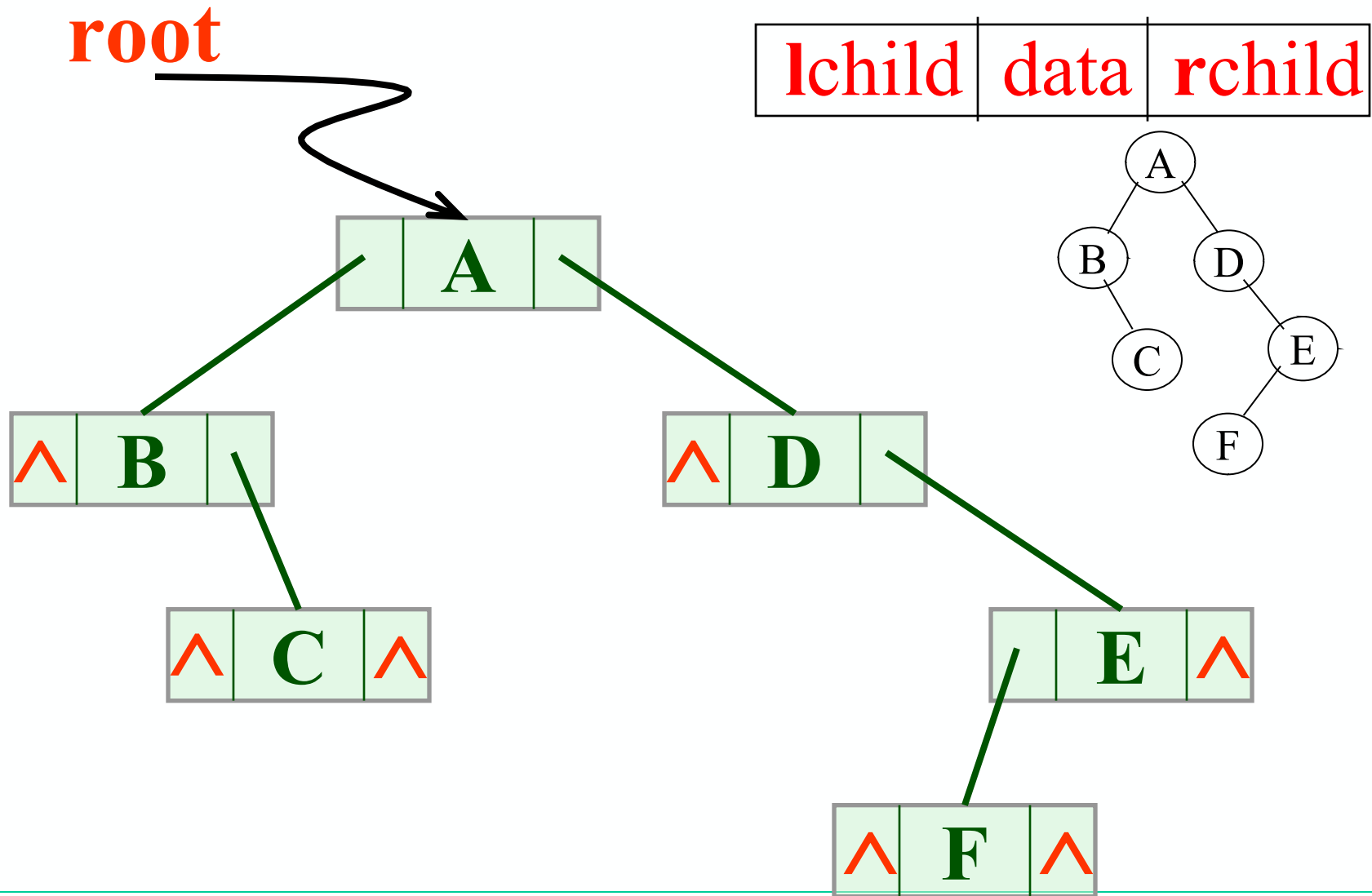
证明：用二叉链表存储包含 $n$ 个结点的二叉树，结点必有 $2n$ 个链域。

除根结点外，二叉树中每一个结点有且仅有一个双亲，即每个结点地址占用了双亲的一个直接后继， $n$ 个结点地址共占用了 $n-1$ 个双亲的指针域。也就是说，只会有 $n-1$ 个结点的链域存放指针。

所以，空指针数目 =  $2n - (n-1) = n+1$  个。



# 结点结构:



思考：二叉链表空间效率这么低，能否利用这些空闲区存放有用的信息或线索？

—可以用它来存放当前结点的直接前驱和后继等线索，以加快查找速度。

这就是**线索二叉树**（Threaded Binary Tree）

# 讨论1：二叉树是1:2的非线性结构，如何定义其直接后继？

对二叉树进行某种遍历之后，将得到一个线性有序的序列。例如对某二叉树的**中序遍历**结果是**B D C E A F H G**，意味着已将该树转为线性排列，显然其中结点**具有唯一前驱和唯一后继**。

先定义遍历规则，然后才能定义直接前驱和后继。

## 讨论2：如何获得这种“直接前驱”或“直接后继”？有何意义？

二叉树中容易找到结点的**左右孩子**信息，但该结点的直接前驱和直接后继只能在某种遍历过程中**动态**获得。

先依**遍历规则**把每个结点对应的**前驱**和**后继线索****预存**起来，这叫做“**线索化**”。

意义：从**任一结点**出发都能快速找到其前驱和后继，且**不必借助堆栈**。

如何预存这类信息？

① ~~每个结点增加两个域：fwd和bwd；~~



存放前驱指针

存放后继指针

缺点：空间效率太低！

② 与原有的左右孩子指针域“复用”，充分利用那 $n+1$ 个空链域。



规定：

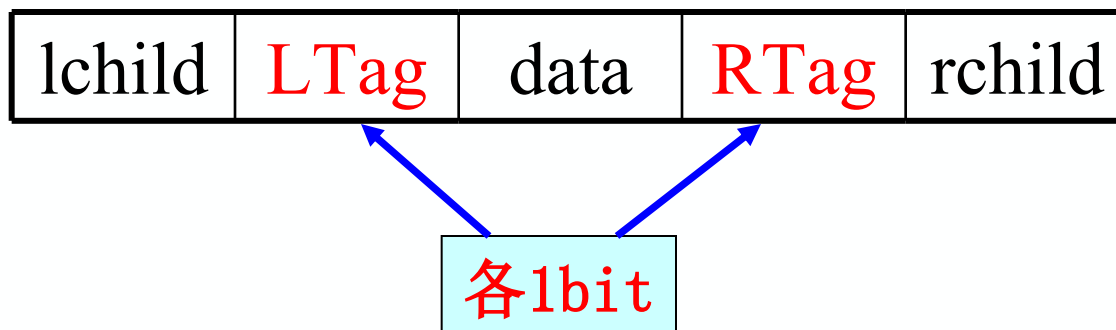
1) 若结点有左子树，则lchild指向其左孩子；否则，lchild指向其直接前驱(即线索)；

2) 若结点有右子树，则rchild指向其右孩子；否则，rchild指向其直接后继(即线索)。

问题：计算机如何判断是孩子指针还是线索指针？

如何区别？

为区别两种不同情况，特增加两个标志域：



约定：

当**LTag**域为**0**时，表示**正常**情况；

当**RTag**域为**1**时，表示**线索**情况。

左(右)孩子

前驱(后继)

## 1. 有关线索二叉树的几个术语

**线索链表：** 用含**Tag**的结点样式所构成的二叉链表。

**线 索：** 指向结点前驱和后继的指针。

**线索二叉树：** 在二叉树的结点上加上线索的二叉树。

**线 索 化：** 对二叉树以**某种次序遍历**使其变为线索二叉树的过程。

**线索化过程就是在遍历过程中修改空指针的过程：**

**将空的lchild改为结点的直接前驱；**

**将空的rchild改为结点的直接后继。**

**非空指针仍然指向孩子结点（称为“正常情况”）**

# 二叉树的二叉线索存储表示

```
typedef enum { Link, Thread } PointerThr;
```

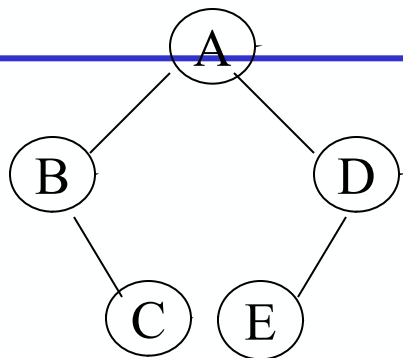
```
// Link==0:指针, Thread==1:线索
```

```
typedef struct BiThrNod {  
    TElemType      data;  
  
    struct BiThrNode *lchild, *rchild; // 左右指针  
  
    PointerThr      LTag, RTag; // 左右标志  
} BiThrNode, *BiThrTree;
```



# 采用中序遍历方法时的约定

- 在二叉树的线索链表上添加一个头结点，并令其lchild域的指针指向二叉树的根结点，其rchild域的指针指向中序遍历时访问的最后一个结点。
- 令二叉树中序序列中的第一个结点的lchild域指针和最后一个结点rchild域的指针均指向头结点。
- 好比为二叉树建立了一个双向线索链表，既可以从第一个结点起顺后继进行遍历，也可从最后一个结点起顺前驱进行遍历。



中序序列: BCAED

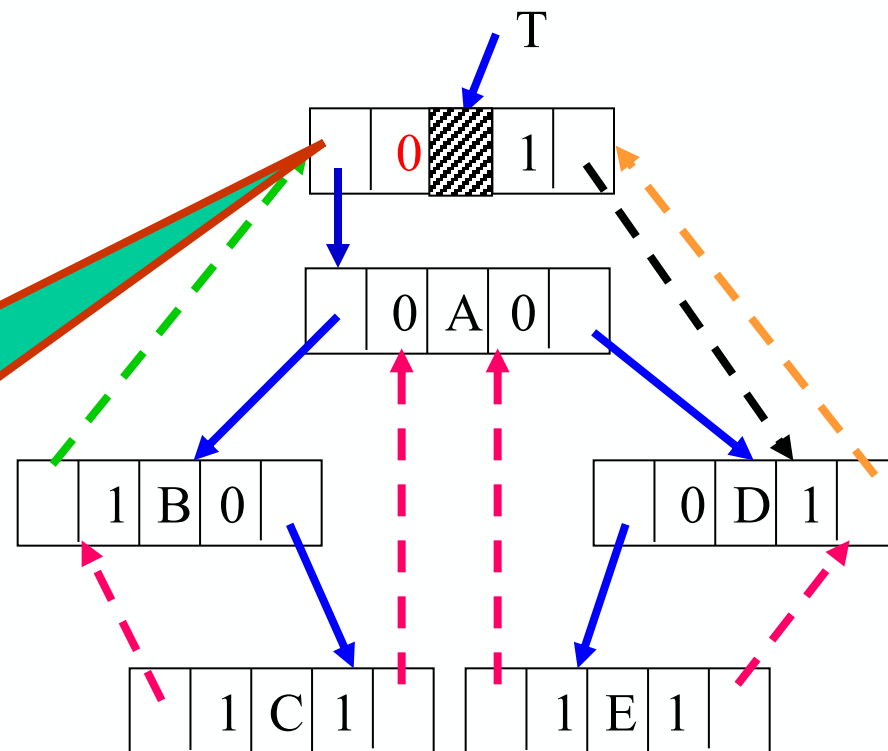
头结点:

Ltag=0, lchild指向根结点

rtag=1, rchild指向遍历序列中最后一个结点

遍历序列中第一个结点的lchild域和最后

一个结点的rchild域都指向头结点



中序序列: BCAED

带头结点的中序线索二叉树

# 线索链表的遍历算法:

由于在线索链表中添加了遍历中得到的“前驱”和“后继”的信息，从而简化了遍历的算法。

```
for ( p = firstNode(T); p; p = Succ(p) )  
    Visit (p);
```

例如:

## 对中序线索化链表的遍历算法

※ 中序遍历的第一个结点 ?

左子树上处于“最左下”（没有左子树）的结点。

※ 在中序线索化链表中结点的后继 ?

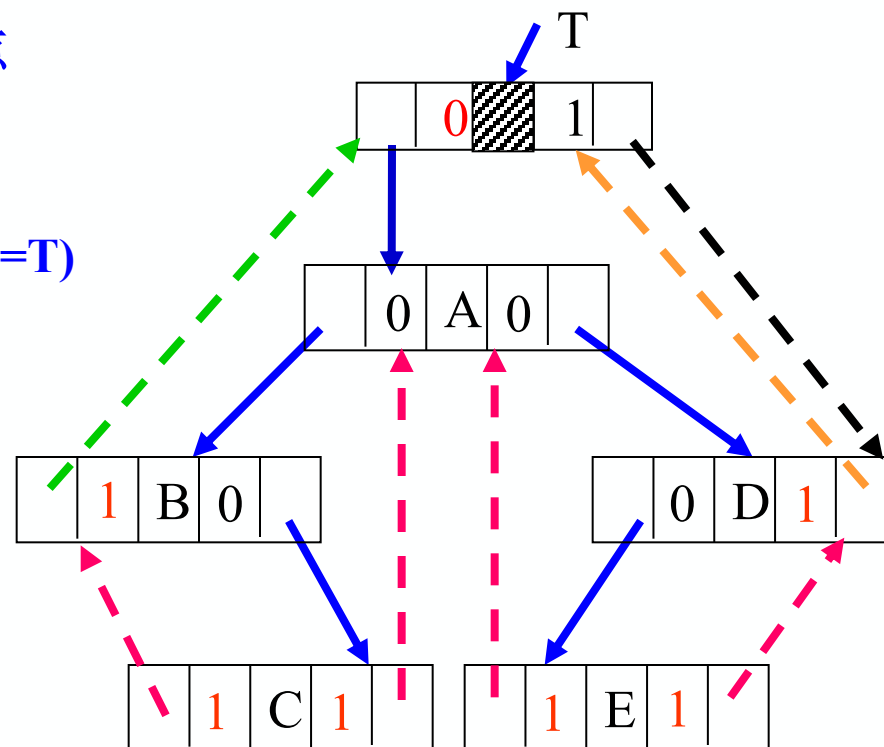
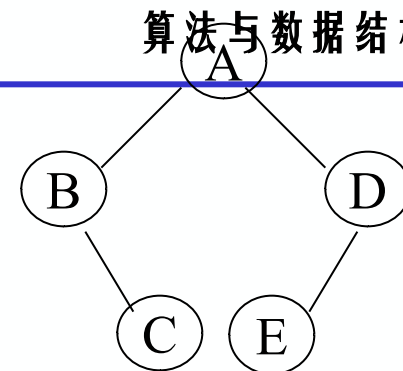
若无右子树，则为后继线索所指结点；

否则为对其右子树进行中序遍历时访问的第一个结点。

```
void InOrderTraverse_Thr(BiThrTree T,  
                           void (*Visit)(TElemType e)) {  
    p = T->lchild;      // p指向根结点  
    while (p != T) {      // 空树或遍历结束时, p==T  
        while (p->LTag==Link) p = p->lchild; // 第1个结点  
        Visit(p->data);  
        while (p->RTag==Thread && p->rchild!=T)  
        { p = p->rchild; Visit(p->data);      // 访问后继结点  
        }  
        p = p->rchild;      // p进至其右子树根  
    }  
} // InOrderTraverse_Thr
```

```

p = T->lchild; // T指向头结点, p指向根结点
while (p != T) {
    // 空树或遍历结束时p==T, p指向头结点
    while (p->LTag == Link) p = p->lchild;
    visit(p->data); //访问左子树为空的结点
    while (p->RTag == Thread && p->rchild != T)
    {
        p = p->rchild; Visit(p->data);
        // 访问后继结点
    }
    p = p->rchild; // p指向其右子树根
}return OK;
    
```



输出: B C A E D

# 知识回顾

给定二叉树的先序遍历序列和中序遍历序列分别是D, A, C, E, B, H, F, G, I和D, C, B, E, H, A, G, I, F, 试画出二叉树。

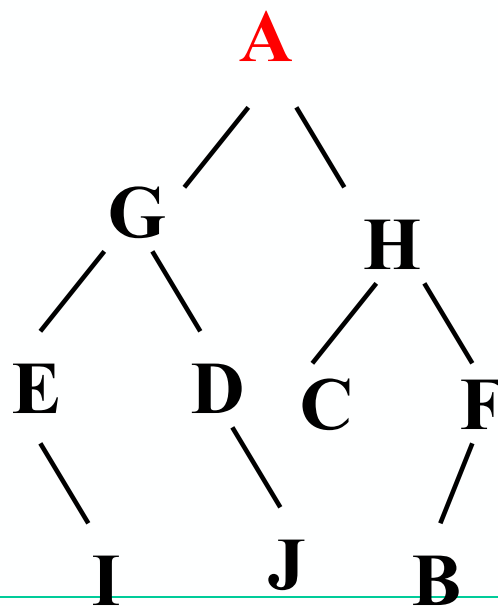
若二叉树中各结点的值均不相同，则：  
由二叉树的前序序列和中序序列，或由其后序序列和中序序列均**能唯一**地确定一棵二叉树，  
但由前序序列和后序序列却**不一定能唯一**地确定一棵二叉树。



- 线索二叉树

例1：带了两个标志的某先序遍历结果如下表所示，请画出对应的二叉树。

Ltag	0	0	1	1	1	1	0	1	0	1
data	A	G	E	I	D	J	H	C	F	B
Rtag	0	0	0	1	0	1	0	1	1	1



**Tag=1**表示线索：  
**Ltag=1**表示前驱  
**Rtag=1**表示后继

# 如何进行二叉树的线索化？

线索化的实质是将二叉链表中的空指针改为指向前驱或后继的线索，而前驱或后继的信息只有在遍历时才能得到，因此线索化的过程即为在遍历的过程中修改空指针的过程。

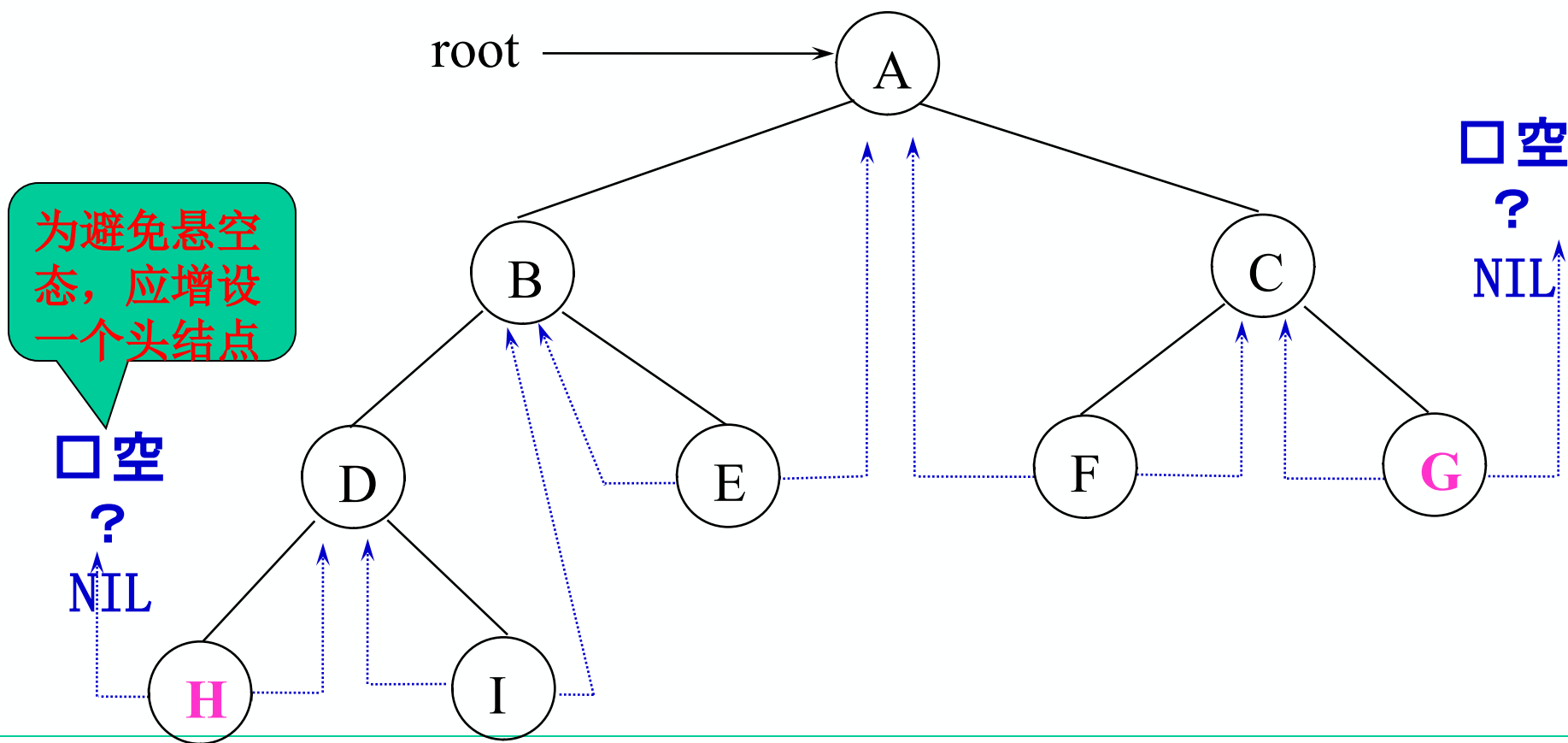
引入一个指针pre，始终指向刚刚访问过的结点，若指针p指向当前访问的结点，则pre指向它的前驱。

# 线索二叉树的生成——线索化

线索化过程就是在遍历过程中修改空指针的过程

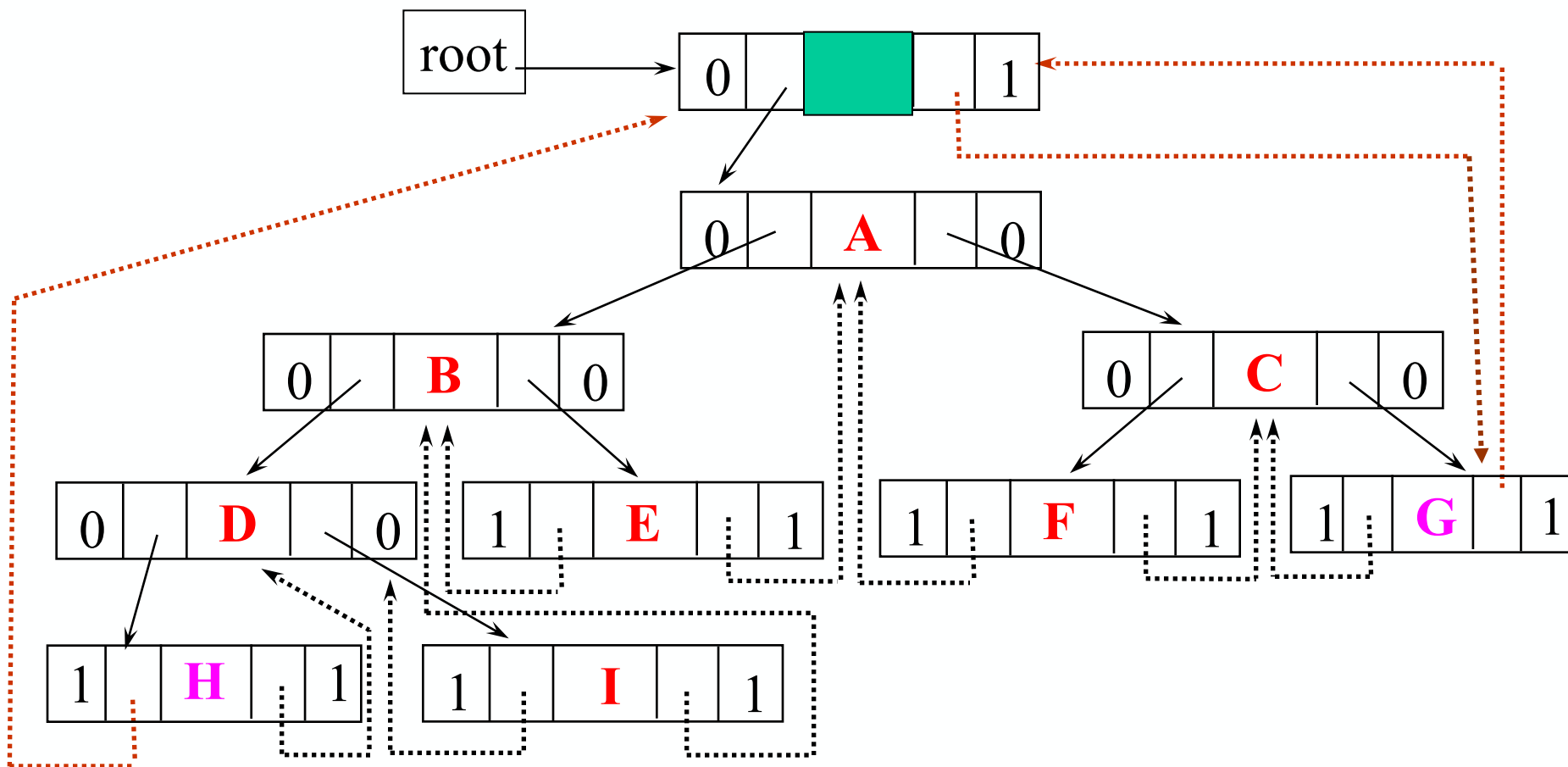
**例2：**画出以下二叉树对应的**中序**线索二叉树。

**解：**对该二叉树**中序**遍历的结果为：**H, D, I, B, E, A, F, C, G**  
所以添加线索应当按如下路径进行：



对应的中序线索二叉树存储结构如图所示：

注：此图中序遍历结果为：**H, D, I, B, E, A, F, C, G**



线索化的实质是将二叉链表中的空指针改为指向前驱或后继的线索，而前驱或后继的信息只有在遍历时才能得到，因此线索化的过程即为在遍历的过程中修改空指针的过程。

引入一个指针pre, 始终指向刚刚访问过的结点，若指针p指向当前访问的结点，则pre指向它的前驱。

```

void InThreading(BiThrTree p) {
    if (p) { // 对以p为根的非空二叉树进行线索化
        InThreading(p->lchild); // 左子树线索化
        if (!p->lchild) // 建前驱线索
            { p->LTag = Thread; p->lchild = pre; }
        if (!pre->rchild) // 建后继线索
            { pre->RTag = Thread; pre->rchild = p; }
        pre = p; // 保持 pre 指向 p 的前驱
        InThreading(p->rchild); // 右子树线索化
    } // if
} // InThreading

```

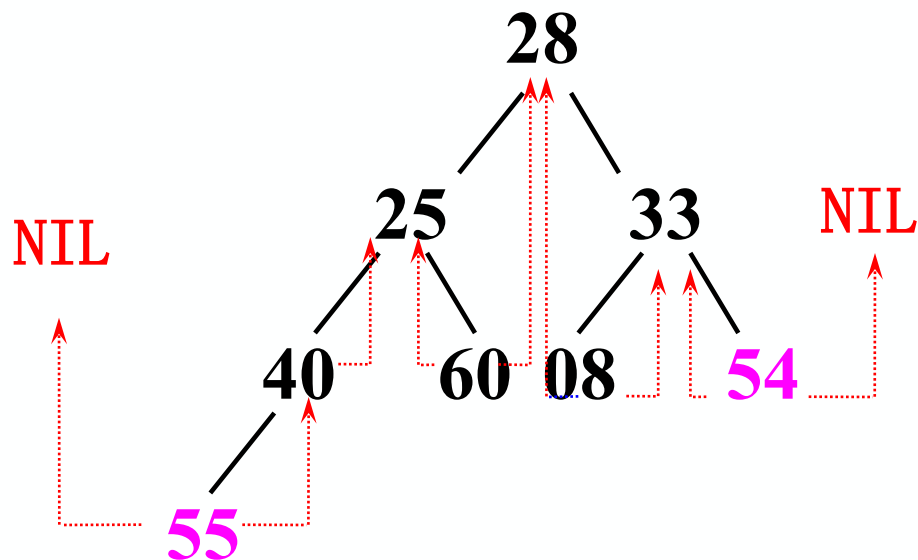
```
Status InOrderThreading(BiThrTree &Thrt,  
                        BiThrTree T) { // 构建中序线索链表  
    if (!(Thrt = (BiThrTree)malloc(  
                                sizeof( BiThrNode))))  
        exit (OVERFLOW);  
    Thrt->LTag = Link; Thrt->RTag = Thread;  
    Thrt->rchild = Thrt;    // 添加头结点  
    ... ..  
    return OK;  
} // InOrderThreading
```



```

if (!T) Thrt->lchild = Thrt;
else {
    Thrt->lchild = T;  pre = Thrt;
    InThreading(T);
    pre->rchild = Thrt; // 处理最后一个结点
    pre->RTag = Thread;
    Thrt->rchild = pre;
}
    
```

例3：给定如图所示二叉树T，请画出与其对应的中序线索二叉树。



解：因为中序遍历序列是：55 40 25 60 28 08 33 54  
对应线索树应当按此规律连线，即在原二叉树中添加虚线。



# 6.4 树和森林



# 主要内容

- 树与二叉树的转换
- 森林与二叉树的转换
- 树与森林的遍历

# 树的三种存储结构

一、双亲表示法

二、孩子链表表示法

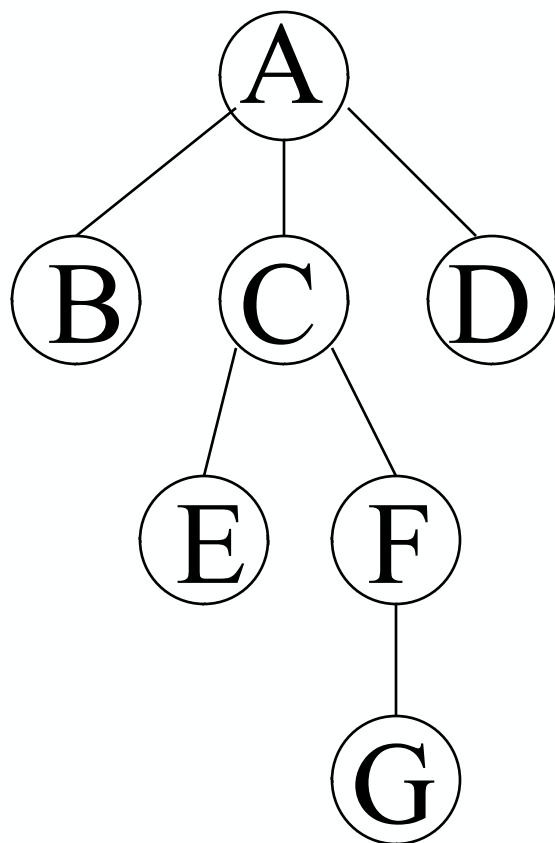
三、树的二叉链表(孩子-兄弟)  
存储表示法

# 一、双亲表示法:

实现：定义结构数组存放树的结点，每个结点含两个域：

- 数据域：存放结点本身信息
- 双亲域：指示本结点的双亲结点在数组中位置

# 一、双亲表示法:



data parent

0

A

-1

1

B

0

2

C

0

3

D

0

4

E

2

5

F

2

6

G

5

n=7

# C语言的类型描述:

```
#define MAX_TREE_SIZE 100
```

结点结构:

<b>data</b>	<b>parent</b>
-------------	---------------

```
typedef struct PTNode {  
    Elem data;  
    int parent; // 双亲位置域  
} PTNode;
```



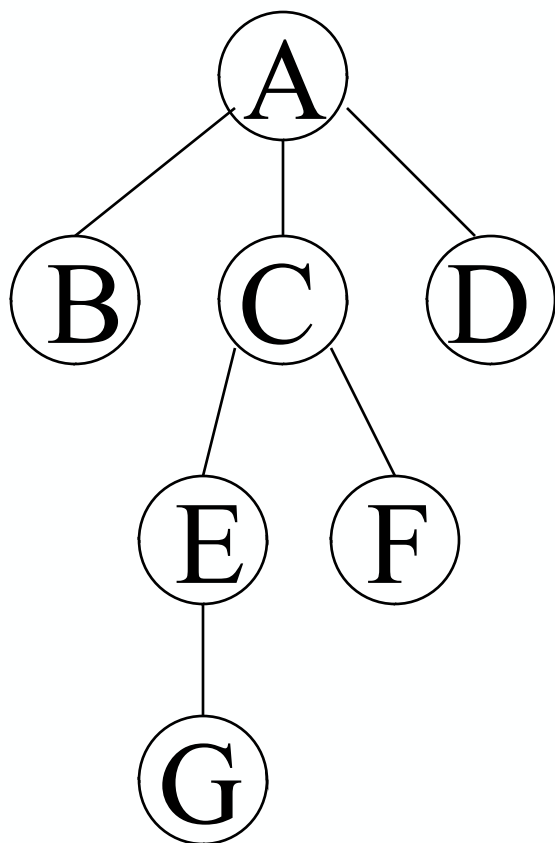
# 树结构:

```
typedef struct {  
    PTNode nodes  
        [MAX_TREE_SIZE];  
    int    n; //结点个数  
} PTree;
```

**特点:** 找双亲容易, 找孩子难



## 二、孩子链表表示法:



data firstchild

0	A	-1	→	1	→	2	→	3	Λ
1	B	0	Λ						
2	C	0	→	4	→	5	Λ		
3	D	0	Λ						
4	E	2	→	6	Λ				
5	F	2	Λ						
6	G	4	Λ						

r=0

n=7

## C语言的类型描述:

孩子结点结构:

<b>child</b>	<b>next</b>
--------------	-------------

```
typedef struct CTNode {  
    int      child;  
    struct CTNode *next;  
} *ChildPtr;
```

# 双亲结点结构

<b>data</b>	<b>firstchild</b>
-------------	-------------------

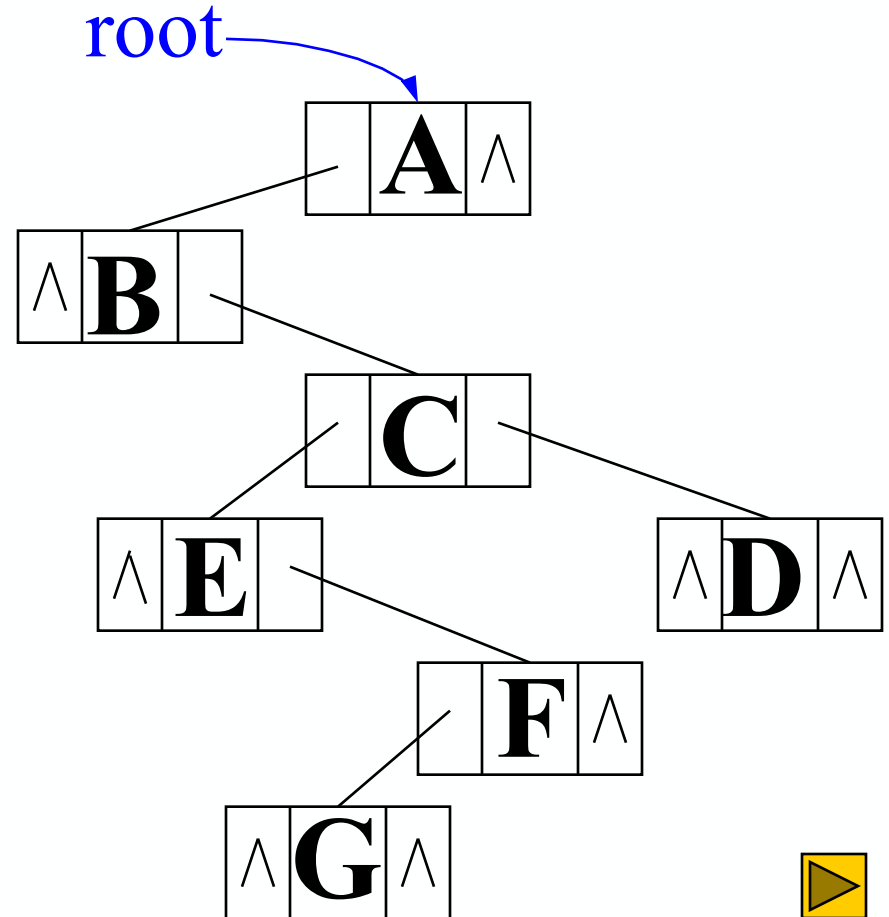
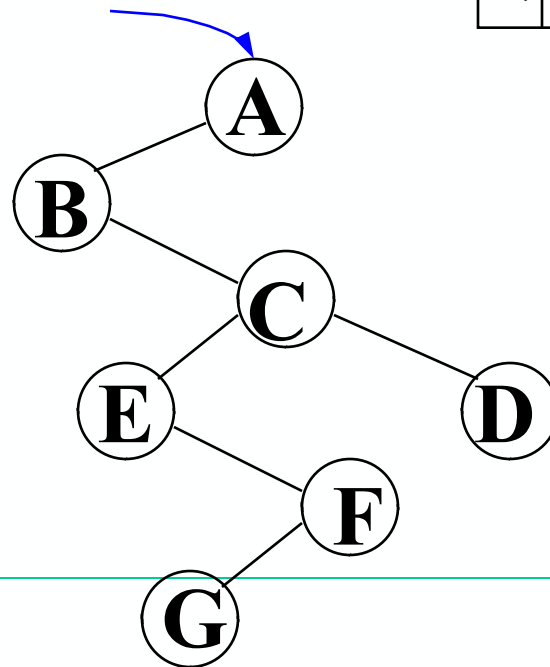
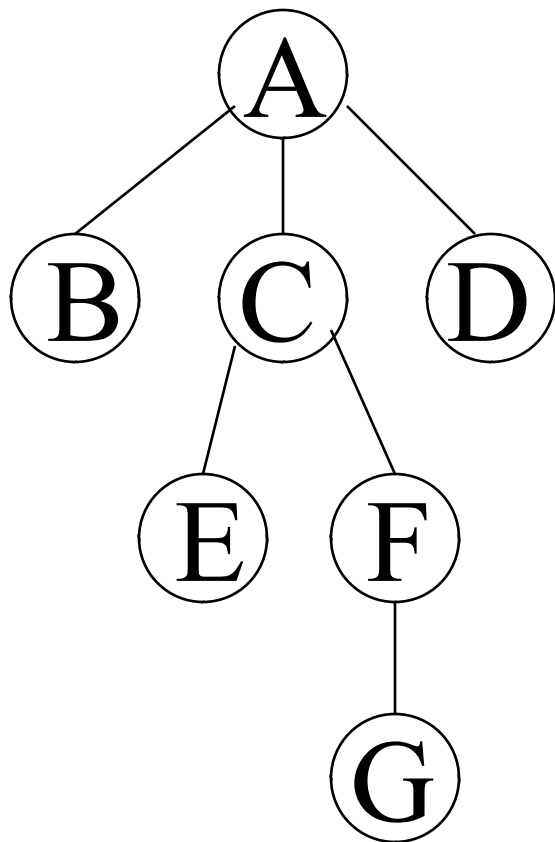
```
typedef struct {  
    Elem    data;  
    ChildPtr firstchild;  
    // 孩子链的头指针  
} CTBox;
```

# 树结构:

```
typedef struct {  
    CTBox  nodes[MAX_TREE_SIZE];  
  
    int    n, r;  
    // 结点数和根结点的位置  
  
} CTree;
```



### 三、树的二叉链表(孩子-兄弟) 存储表示法



# C语言的类型描述:

结点结构:

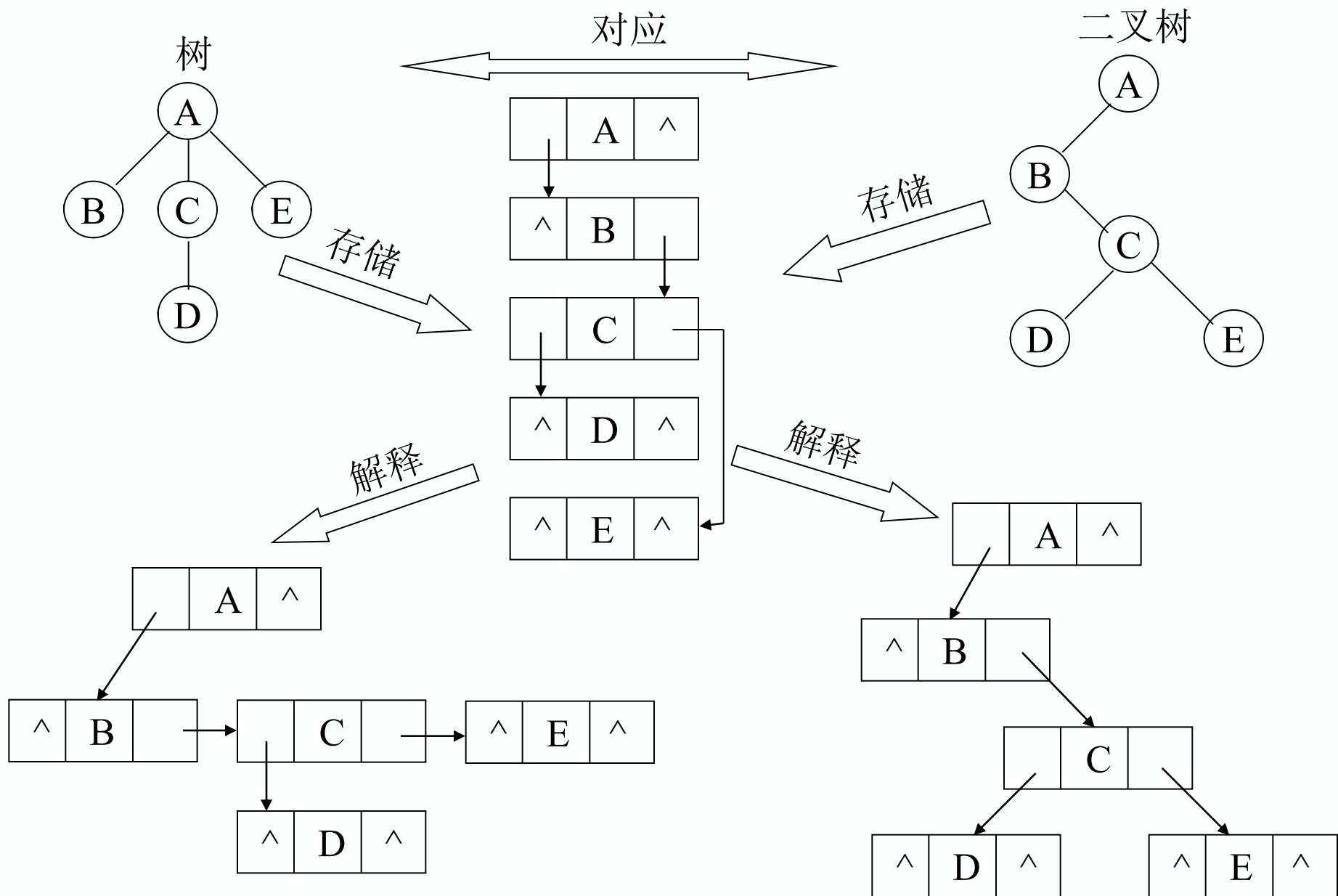
<b>firstchild</b>	<b>data</b>	<b>nextsibling</b>
-------------------	-------------	--------------------

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

# 树与二叉树转换



# — 树与二叉树转换

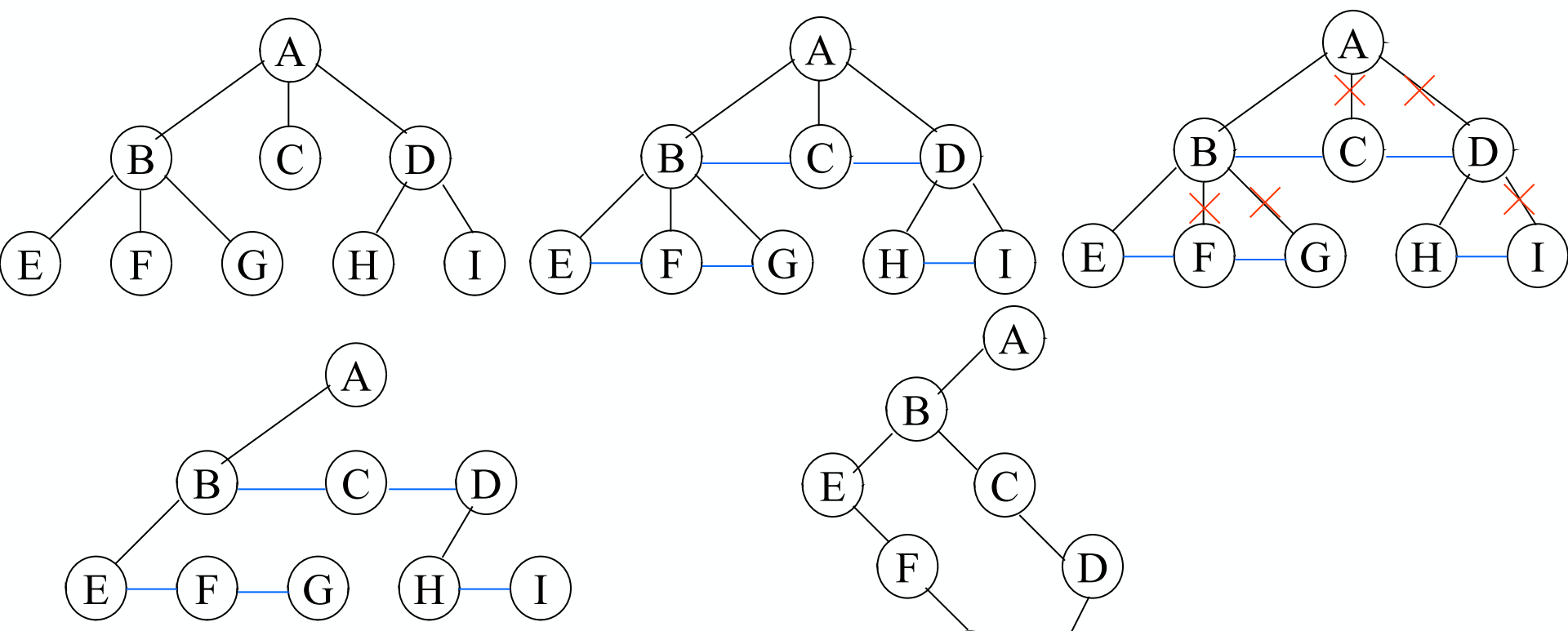


# 将树转换成二叉树

- **加线**：在兄弟之间加一连线
- **抹线**：对每个结点，除了其左孩子外，去除与其其余孩子之间的关系
- **旋转**：以树的根结点为轴心，将整树顺时针转 $45^\circ$

- 将树转换成二叉树

- 加线：在兄弟之间加一连线
- 抹线：对每个结点，除了其左孩子外，去除其与其余孩子之间的关系
- 旋转：以树的根结点为轴心，将整树顺时针转45°



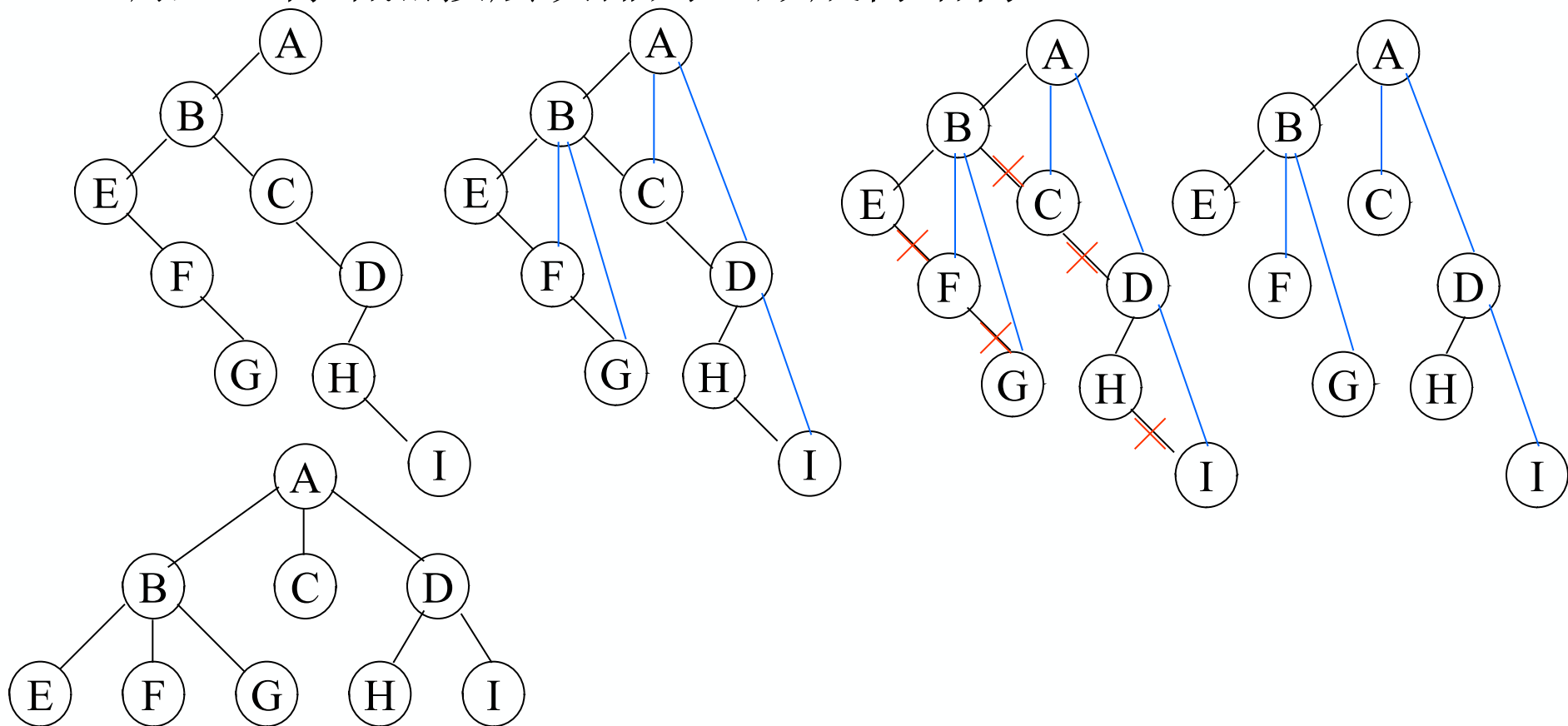
树转换成的二叉树其右子树一定为空

# 将二叉树转换成树

- **加线**：若 $p$ 结点是双亲结点的左孩子，则将 $p$ 的右孩子，右孩子的右孩子，……沿分支找到的所有右孩子，都与 $p$ 的双亲用线连起来
- **抹线**：抹掉原二叉树中双亲与右孩子之间的连线
- **调整**：将结点按层次排列，形成树结构

# 将二叉树转换成树

- 加线：若p结点是双亲结点的左孩子，则将p的右孩子，右孩子的右孩子，……沿分支找到的所有右孩子，都与p的双亲用线连起来
- 抹线：抹掉原二叉树中双亲与右孩子之间的连线
- 调整：将结点按层次排列，形成树结构



# 森林和二叉树的转换

设森林

$$F = ( T_1, T_2, \dots, T_n );$$

$$T_1 = (\text{root}, t_{11}, t_{12}, \dots, t_{1m});$$

二叉树

$$B = ( \text{LBT}, \text{Node}(\text{root}), \text{RBT} );$$

由森林转换成二叉树的转换规则为：

若  $F = \Phi$ ，则  $B = \Phi$ ；

否则，

由  $\text{ROOT}(T_1)$  对应得到  $\text{Node}(\text{root})$ ；

由  $(t_{11}, t_{12}, \dots, t_{1m})$  对应得到 LBT；

由  $(T_2, T_3, \dots, T_n)$  对应得到 RBT。

由二叉树转换为森林的转换规则为：

若  $B = \Phi$ ， 则  $F = \Phi$ ；

否则，

由  $\text{Node}(\text{root})$  对应得到  $\text{ROOT}(T_1)$ ；

由LBT 对应得到  $(t_{11}, t_{12}, \dots, t_{1m})$ ；

由RBT 对应得到  $(T_2, T_3, \dots, T_n)$ 。

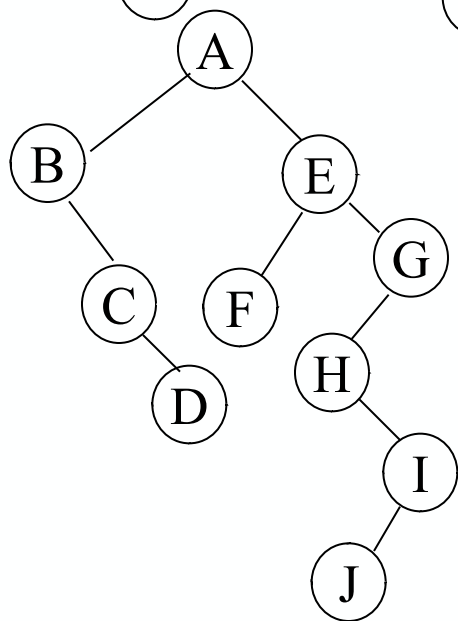
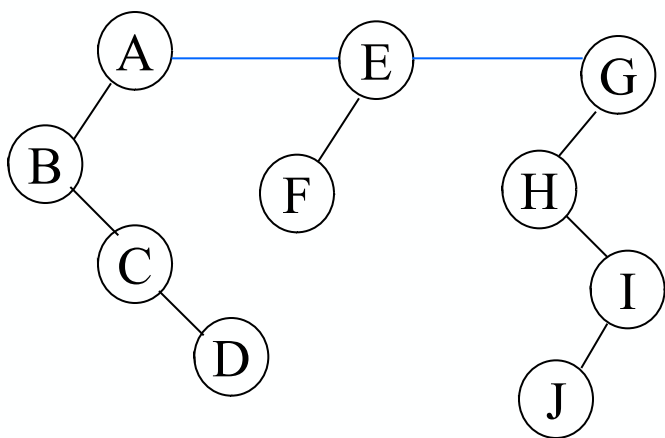
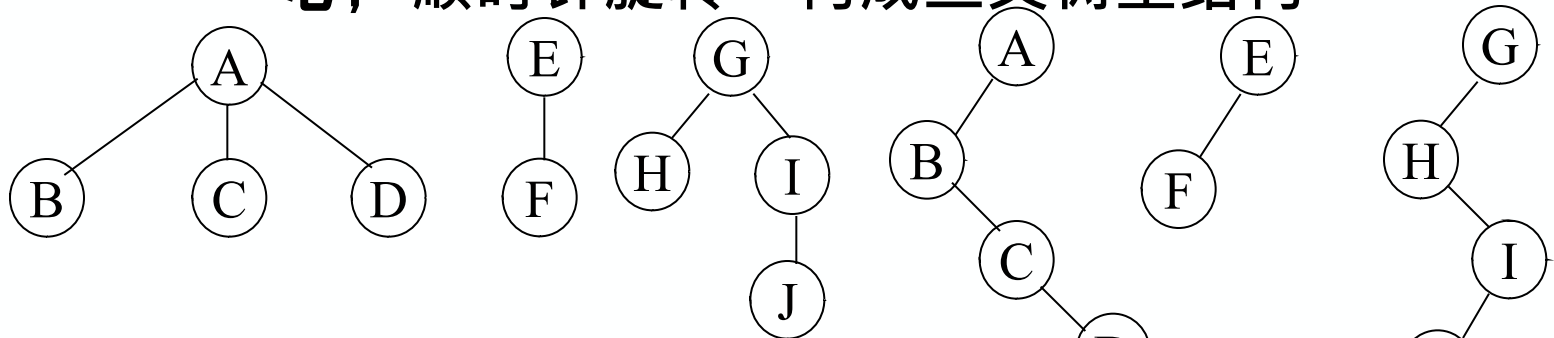


- 森林转换成二叉树

- 将各棵树分别转换成二叉树

- 将每棵树的根结点用线相连

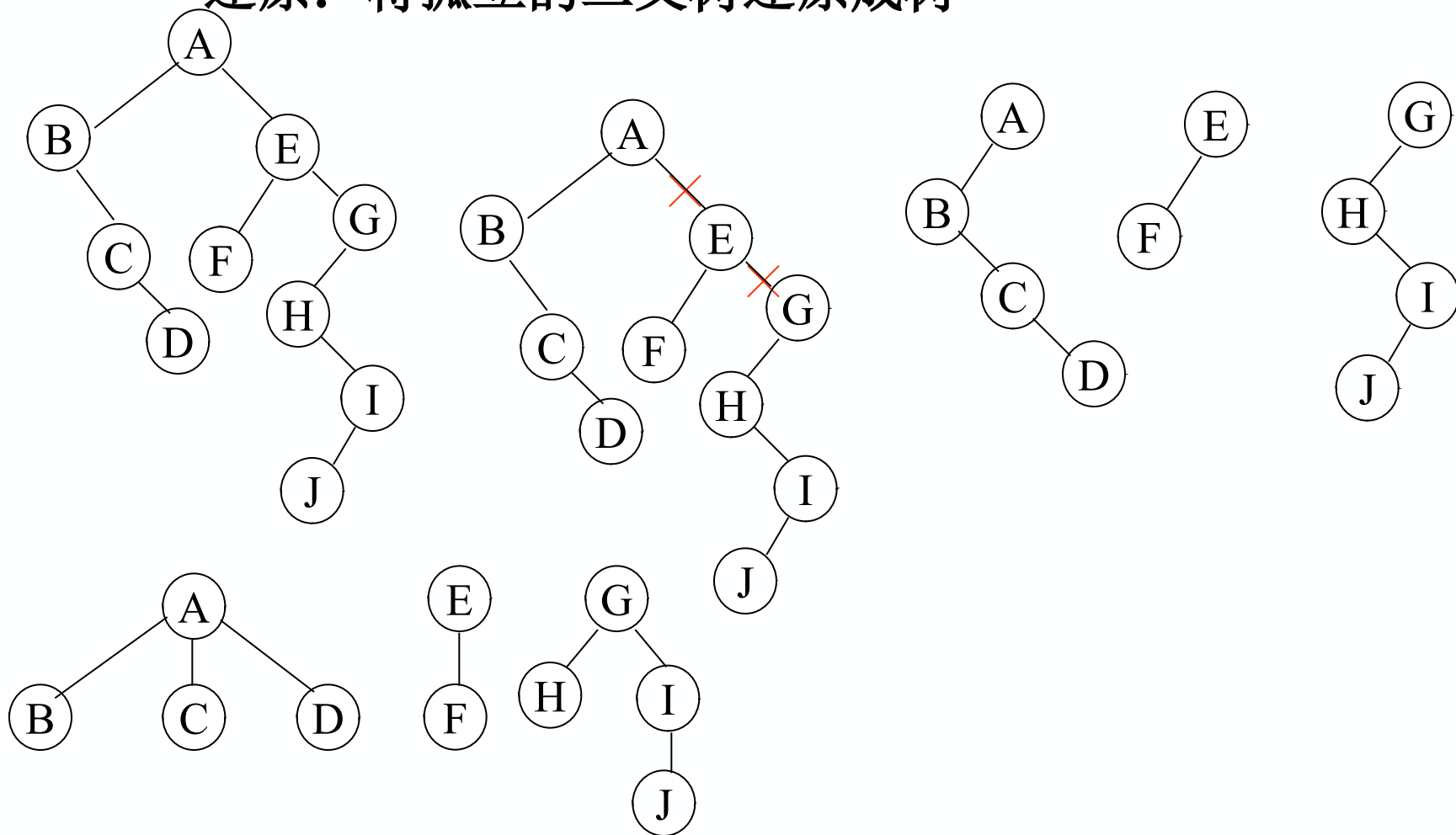
- 以第一棵树根结点为二叉树的根，再以根结点为轴心，顺时针旋转，构成二叉树型结构



- 二叉树转换成森林

- 抹线：将二叉树中根结点与其右孩子连线，及沿右分支搜索到的所有右孩子间连线全部抹掉，使之变成孤立的二叉树

- 还原：将孤立的二叉树还原成树



# 树和森林的遍历



- 一、树的遍历
- 二、森林的遍历

# 树的遍历可有三条搜索路径:

## 先根(次序)遍历:

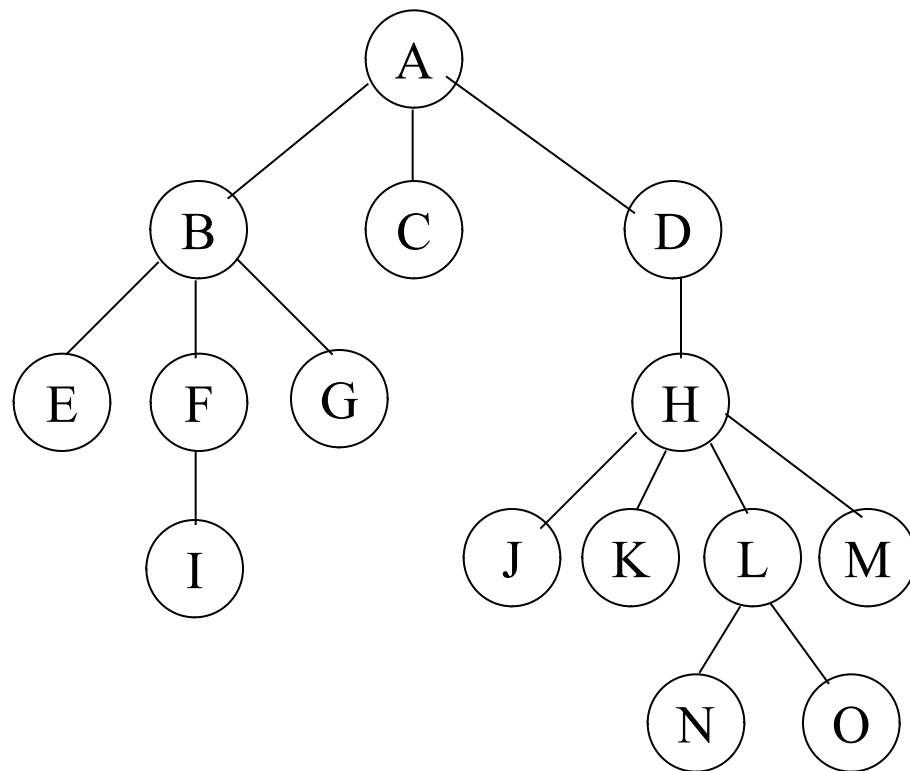
若树不空，则先访问根结点，然后依次先根遍历各棵子树。

## 后根(次序)遍历:

若树不空，则先依次后根遍历各棵子树，然后访问根结点。

## 按层次遍历:

若树不空，则自上而下自左至右访问树中每个结点。

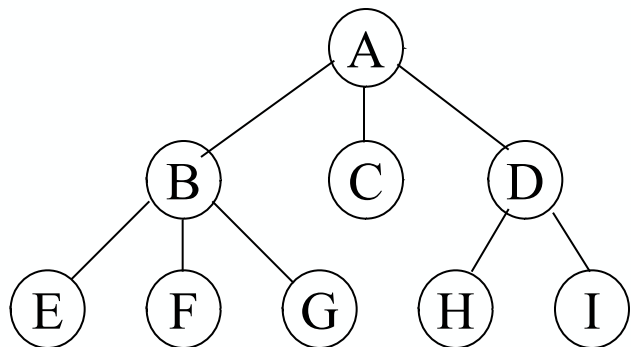


先序遍历: **ABEF I GCDHJ KLNOM**

后序遍历: **E I F G B C J K N O L M H D A**

层次遍历: **A B C D E F G H I J K L M N O**

# 树的遍历和二叉树遍历的对应关系

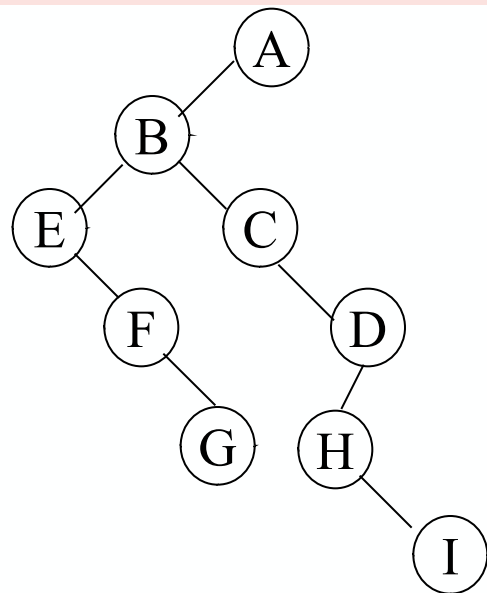


先序遍历:

**A B E F G C D H I**

后序遍历:

**E F G B C H I D A**



先序遍历:

**A B E F G C D H I**

后序遍历:

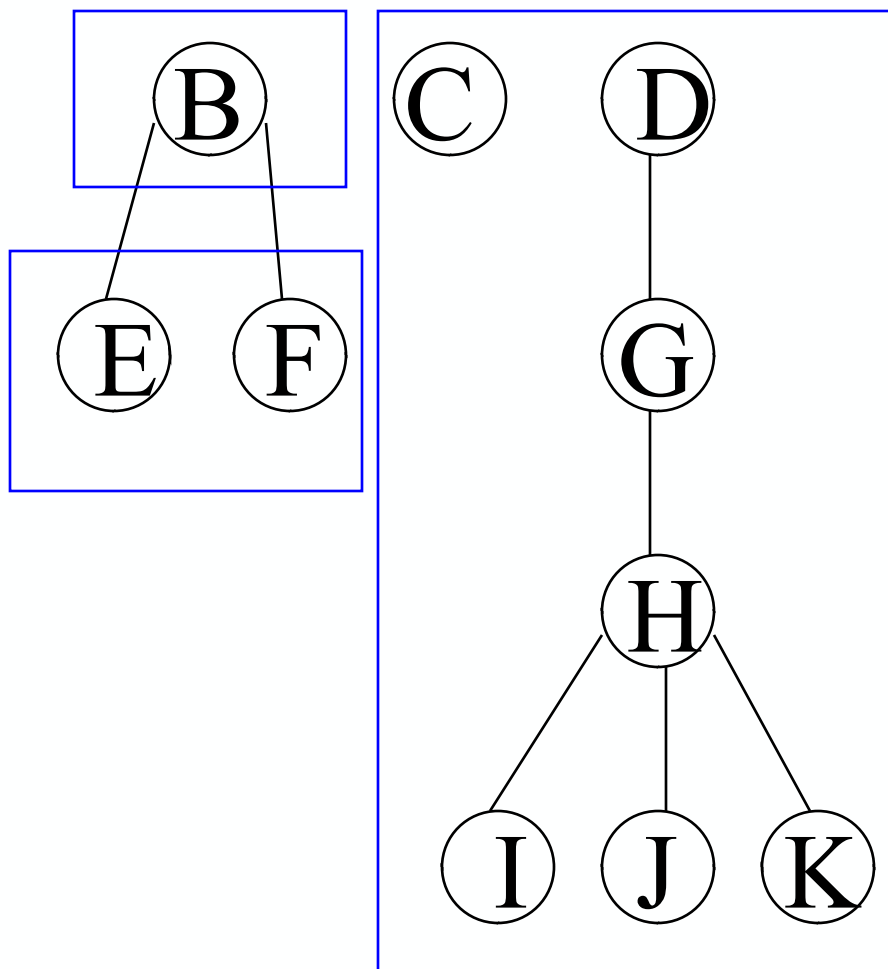
**G F E I H D C B A**

中序遍历:

**E F G B C H I D A**



# 森林的遍历



森林由三部分构成：

1. 森林中第一棵树的根结点；
2. 森林中第一棵树的子树森林；
3. 森林中其它树构成的森林。



# 1. 先序遍历

若森林不空，则

访问森林中第一棵树的根结点；

先序遍历森林中第一棵树的子树森林；

先序遍历森林中(除第一棵树之外)其余树构成的森林。

即：依次从左至右对森林中的每一棵树进行先根遍历。

## 2. 中序遍历

若森林不空，则

中序遍历森林中第一棵树的子树森林；

访问森林中第一棵树的根结点；

中序遍历森林中(除第一棵树之外)其余树构成的森林。

即：依次从左至右对森林中的每一棵树进行后根遍历。

# 树和森林的遍历和二叉 树遍历的对应关系

树

森林

二叉树

先根遍历

先序遍历

先序遍历

后根遍历

中序遍历

中序遍历



## 知识回顾

- 二叉树的先序序列为1, 2, 3, 4, 5, 6, 8, 7 ;  
中序序列为3, 4, 8, 6, 7, 5, 2, 1, 请画出这棵二叉树，并且把这棵二叉树转换成树。

## 6.5 霍夫曼树及其应用

- 最优树的定义 ⇒
- 如何构造最优二叉树 ⇒
- 霍夫曼编码 ⇒


# 一、最优树（也称霍夫曼树）的定义

- ◆ **结点的路径长度**定义为：  
从根结点到该结点的路径上分支的数目。
- ◆ **树的路径长度**定义为：  
树中每个结点的路径长度之和。
- ◆ **结点的带权路径长度**定义为：  
结点的路径长度与结点上**权**的积。

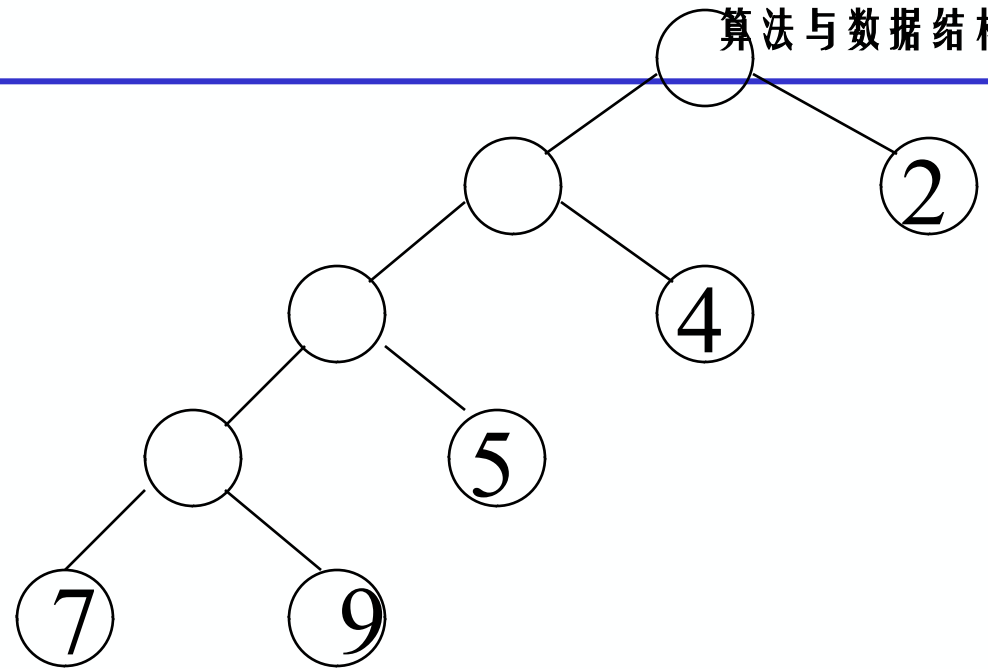
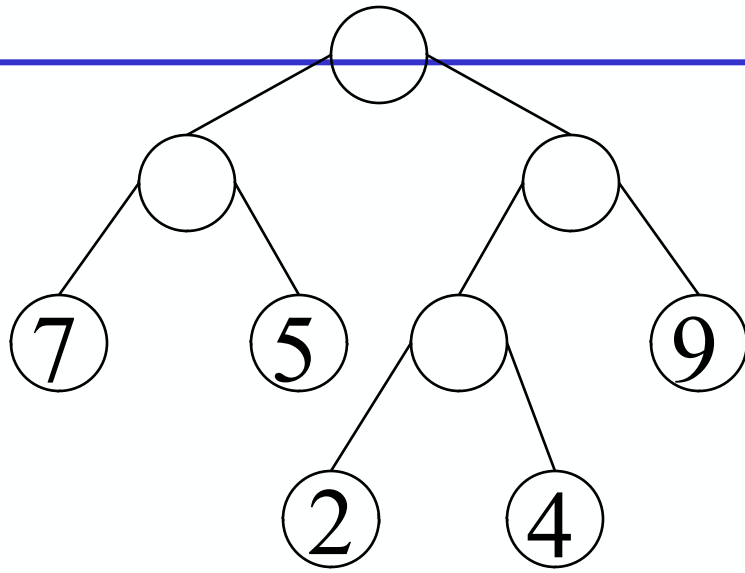
◆ 树的带权路径长度定义为：

树中所有叶子结点的带权路径长度之和

$WPL(T) = \sum w_k l_k$  (对所有叶子结点)。

例如： 

在所有含  $n$  个叶子结点、并带相同权值的  $m$  叉树中，必存在一棵其带权路径长度取最小值的树，称为“最优树”。



$WPL(T)=$

$7 \times 2 + 5 \times 2 + 2 \times 3 +$

$4 \times 3 + 9 \times 2$

$= 60$

$WPL(T)=$

$7 \times 4 + 9 \times 4 + 5 \times 3 +$

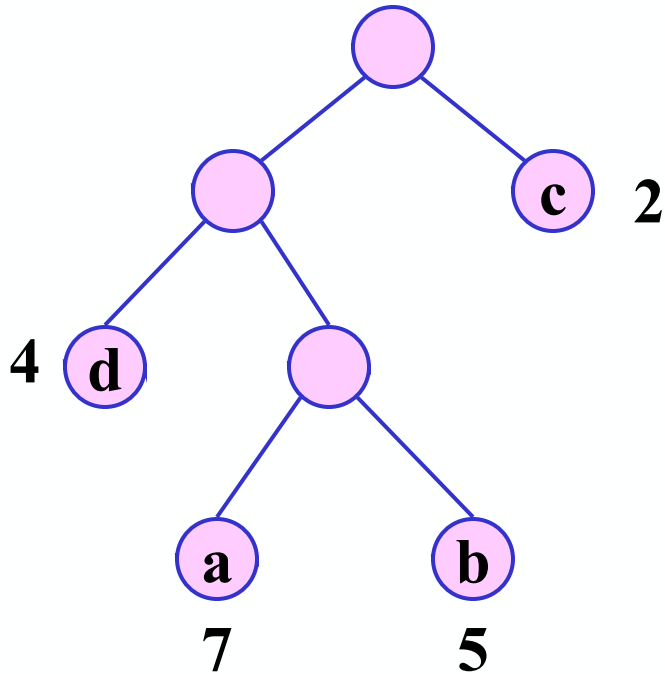
$4 \times 2 + 2 \times 1 = 89$



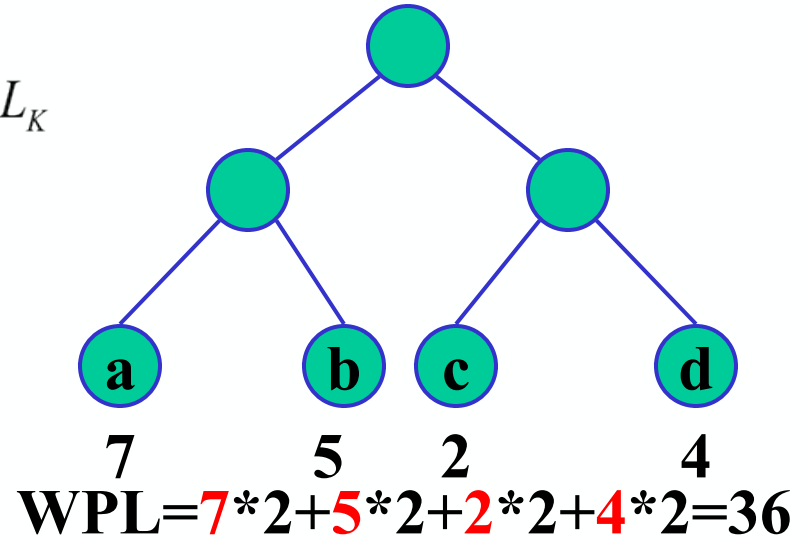


例 有4个结点，权值分别为7，5，2，4，构造有4个叶子结点的二叉树

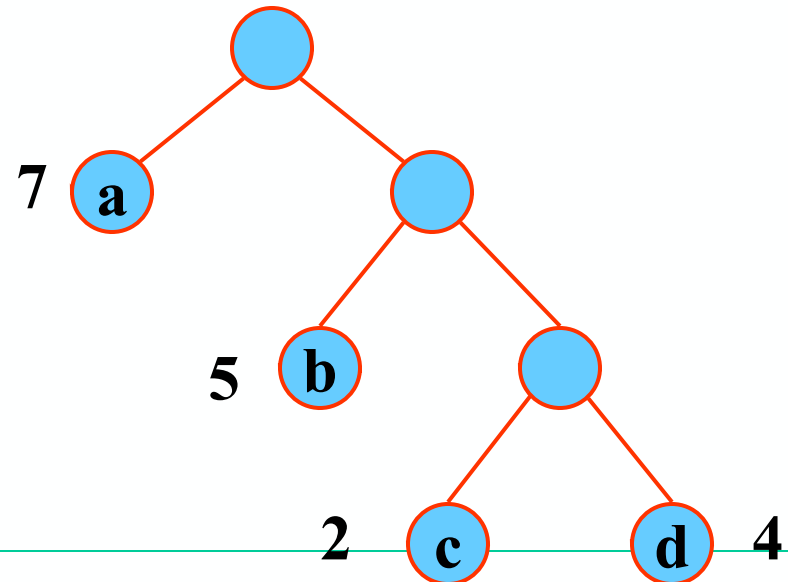
$$WPL = \sum_{k=1}^n W_K L_K$$



$$WPL = 7*3 + 5*3 + 2*1 + 4*2 = 46$$



$$WPL = 7*2 + 5*2 + 2*2 + 4*2 = 36$$



$$WPL = 7*1 + 5*2 + 2*3 + 4*3 = 35$$



## 二、如何构造最优树

(霍夫曼算法) 以二叉树为例:

(1) 根据给定的  $n$  个权值  $\{w_1, w_2, \dots, w_n\}$ , 构造  $n$  棵二叉树的集合

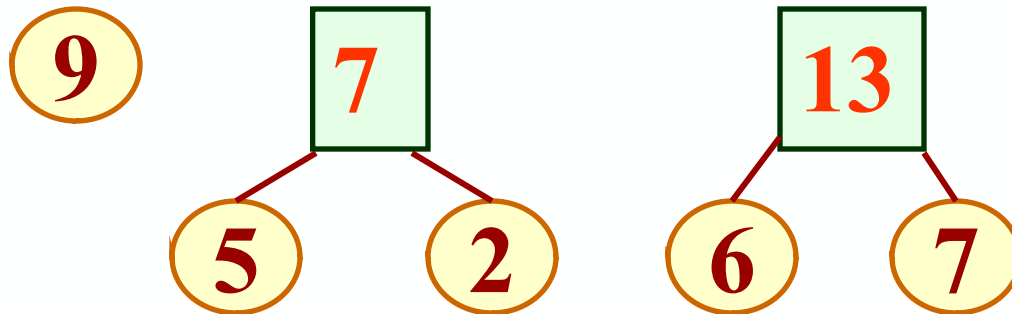
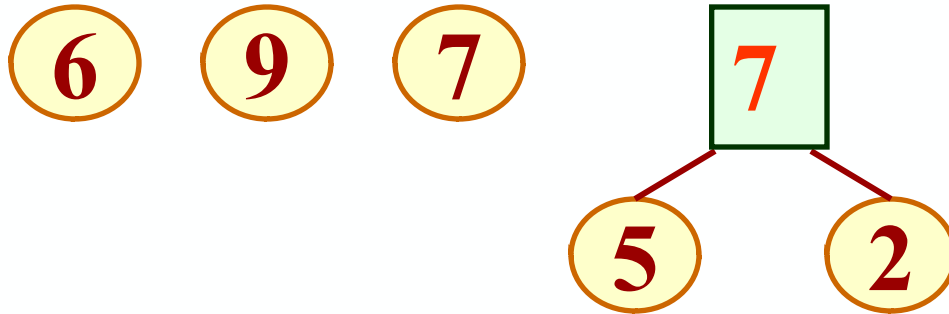
$$F = \{T_1, T_2, \dots, T_n\},$$

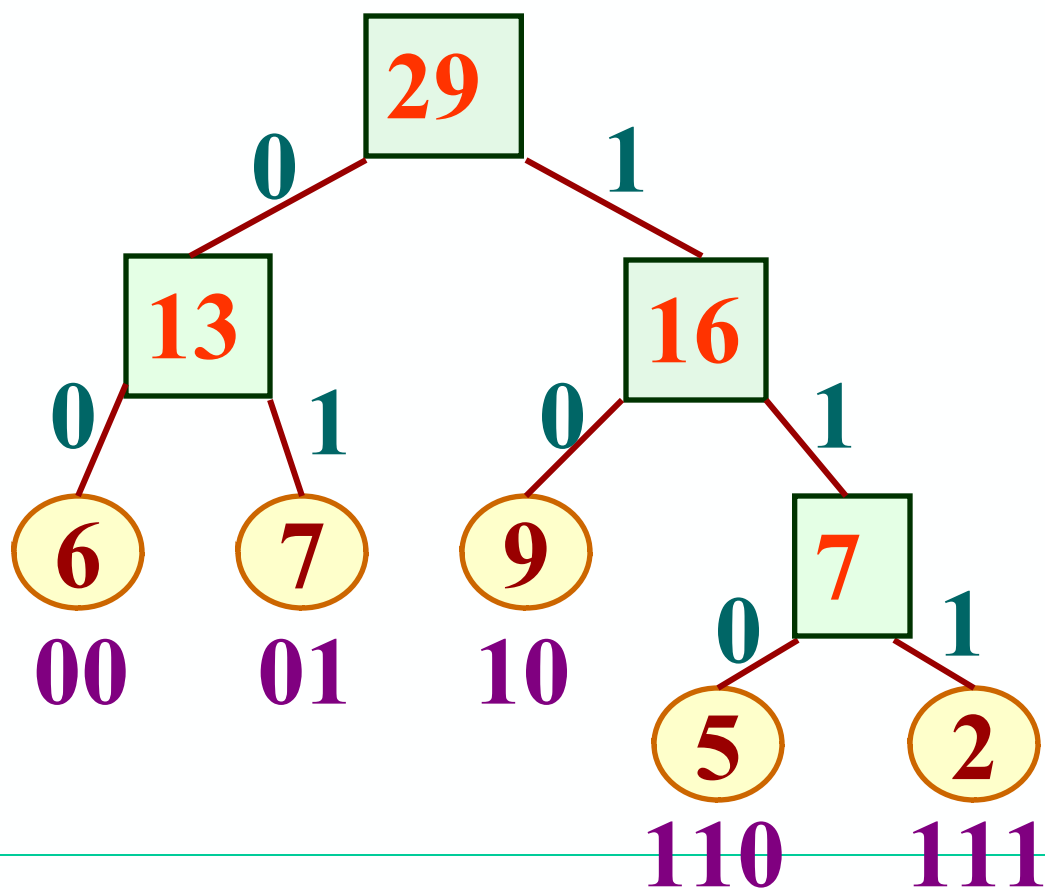
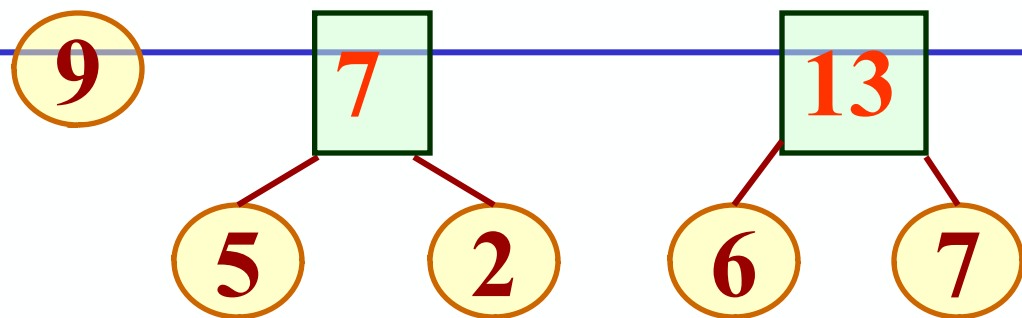
其中每棵二叉树中均只含一个带权值为  $w_i$  的根结点, 其左、右子树为空树;

(2) 在  $F$  中选取其根结点的权值为最小的两棵二叉树，分别作为左、右子树构造一棵新的二叉树，并置这棵新的二叉树根结点的权值为其左、右子树根结点的权值之和；

- (3) 从 $F$ 中删去这两棵树，同时加入刚生成的新树；
- (4) 重复 (2) 和 (3) 两步，直至  $F$  中只含一棵树为止。

例如：已知权值  $W=\{ 5, 6, 2, 9, 7 \}$



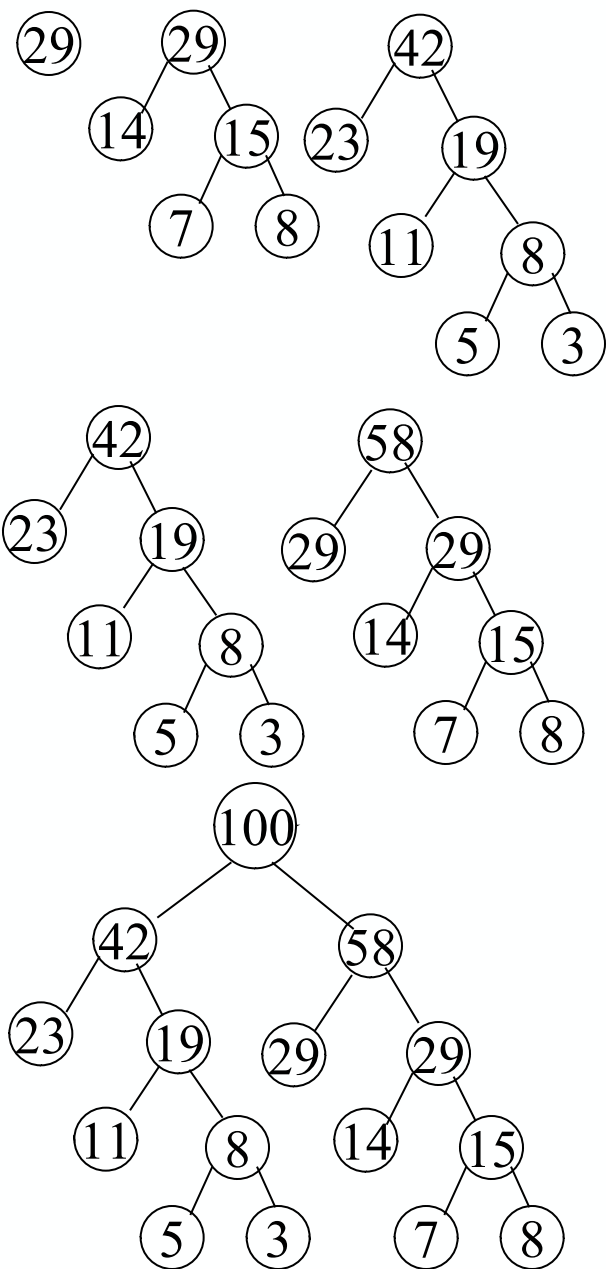
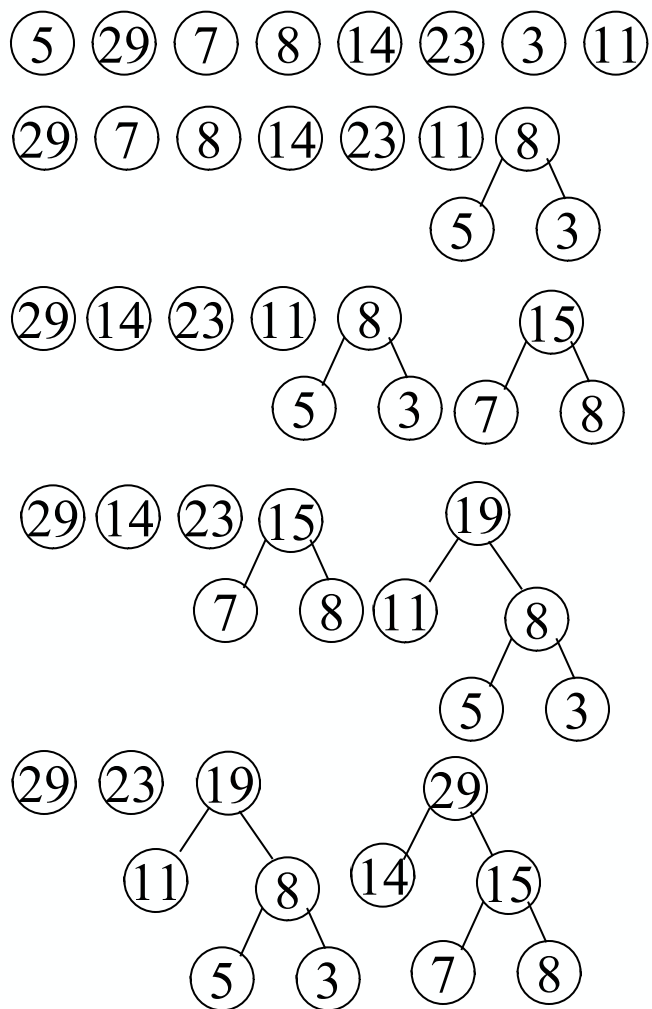


# 霍夫曼树的构造过程

**基本思想：**使权大的结点靠近根

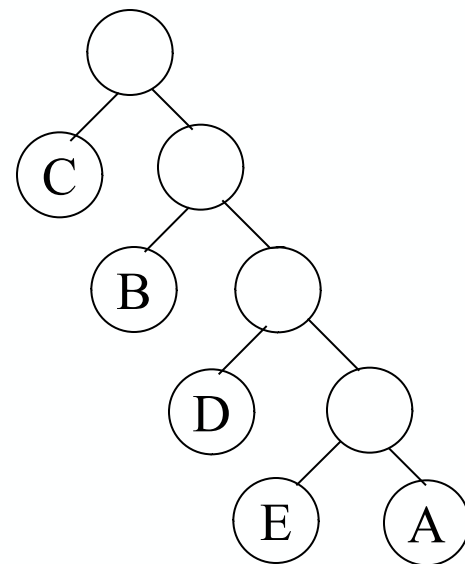
**操作要点：**对权值的**合并、删除与替换**，总是合并当前值最小的两个

例  $w = \{5, 29, 7, 8, 14, 23, 3, 11\}$

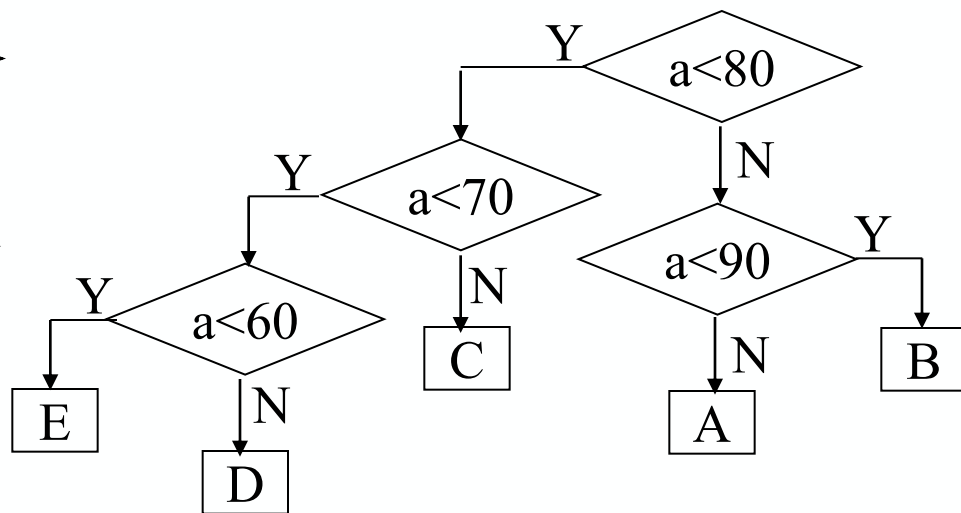
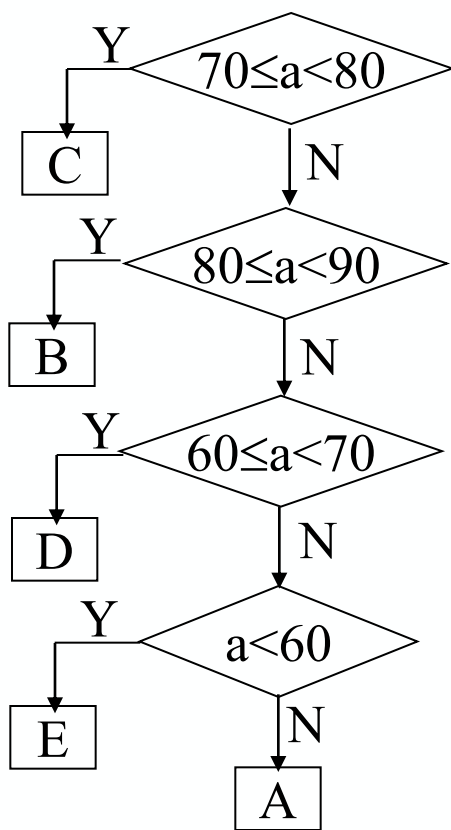
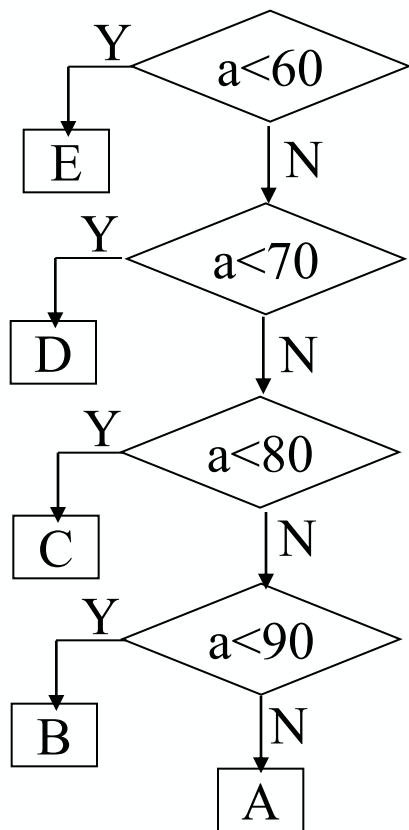




# • Huffman树应用--最佳判定树



等级	E	D	C	B	A
分数段	0~59	60~69	70~79	80~89	90~100
比例	0.05	0.15	0.40	0.30	0.10



## 三、霍夫曼编码

在远程通讯中，要将待传字符转换成二进制的字符串，怎样编码才能使它们组成的报文在网络中传得最快？

A	00
B	01
C	10
D	11

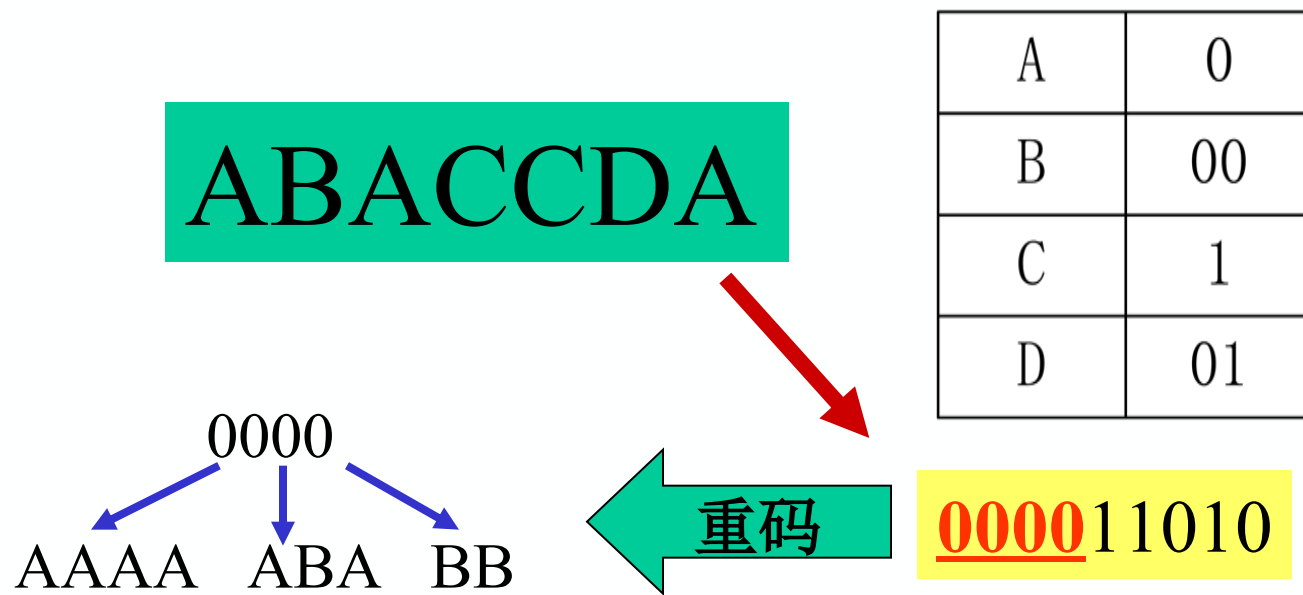
ABACCD A

A	0
B	00
C	1
D	01

000110010101100

000011010

出现次数较多的字符采用尽可能短的编码



**关键：**要设计长度不等的编码，则必须使任一字符的编码都不是另一个字符的编码的**前缀**——**前缀编码**

## 前缀编码:

任一个字符的编码都不是同一字符集中另一个字符的编码的前缀。

A: 0   C:10   B:110   D:111

利用霍夫曼树可以构造一种不等长的二进制编码，并且构造所得的霍夫曼编码是一种最优前缀编码，使所传电文的总长度最短。

采用二叉树设计  
前缀编码

ABACCDAA

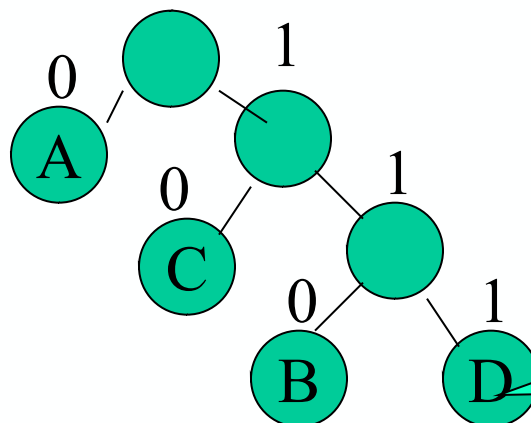
A—0

B—110

C—10

D—111

0110010101110

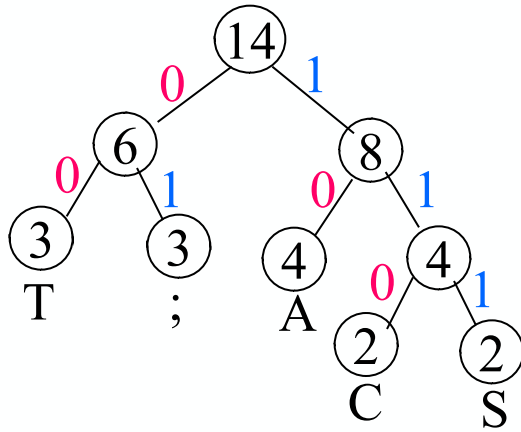


左分支用“0”  
右分支用“1”

# Huffman编码：数据通信用的二进制编码

- **思想：** 根据字符出现频率编码，使电文总长最短
- **编码：** 根据字符出现频率构造Huffman树，然后将树中结点引向其左孩子的分支标“0”，引向其右孩子的分支标“1”；每个字符的编码即为从根到每个叶子的路径上得到的0、1序列。

例 要传输的字符集  $D=\{C,A,S,T, ; \}$   
字符出现频率  $w=\{2,4,2,3,3\}$



T : 00

; : 01

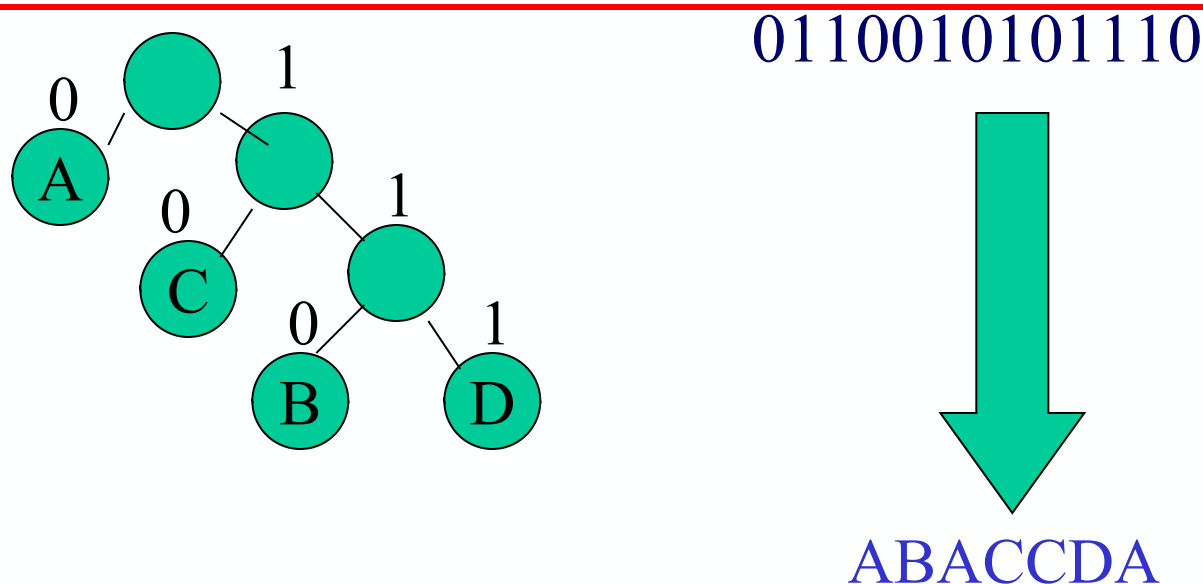
A : 10

C : 110

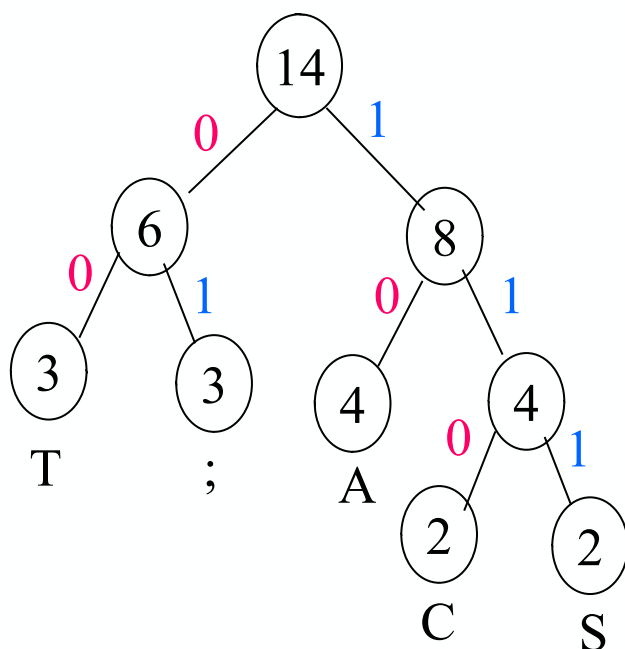
S : 111



分解接收字符串：遇“0”向左，遇“1”向右；一旦到达叶子结点，则译出一个字符，反复由根出发，直到译码完成。



特点：每一码都不是另一码的前缀，绝不会错译！称为前缀码



T : 00  
 ; : 01  
 A : 10  
 C : 110  
 S : 111

例 电文是{CAS;CAT;SAT;AT}  
 其编码 “11010111011101000011111000011000”  
 电文编码为“1101000”  
 译文只能是“CAT”

例，已知某系统在通讯联系中只出现8种字符，其概率分别为0.05,0.29,0.07,0.08,0.14,0.23,0.03,0.11，试设计霍夫曼编码。

0.05 0.29 0.07 0.08 0.14 0.23 0.03 0.11

0.08 0.29 0.07 0.08 0.14 0.23 0.11

0.15 0.29 0.08 0.14 0.23 0.11

0.15 0.29 0.19 0.14 0.23

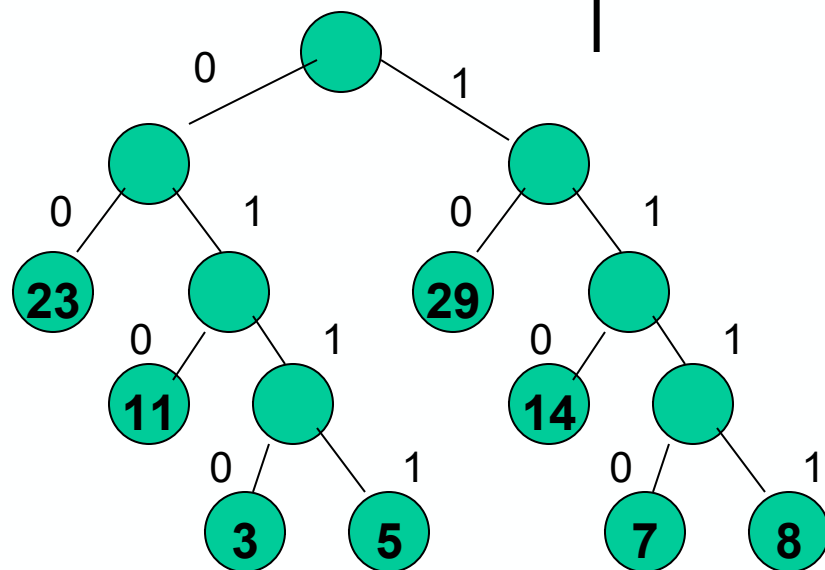
0.29 0.29 0.19 0.23

0.29 0.29 0.42

0.58 0.42

1

0.05: 0111  
0.29: 10  
0.07: 1110  
0.08: 1111  
0.14: 110  
0.23: 00  
0.03: 0110  
0.11: 010



# 霍夫曼树和霍夫曼编码的存储表示

- 由于霍夫曼树中没有度为1的结点，（性质3： $n_0 = n_2 + 1$ ）则一颗有 $n$ 个叶子结点的霍夫曼树共有 $2n - 1$ 个结点，可以存储在 $2n - 1$ 的一维数组中。
- 编码：从叶子到根  
译码：从根到叶子
- 对每个结点：既需要知道双亲，又需要知道孩子。

- ✓根据给定的 $n$ 个权值  $\{w_1, w_2, \dots, w_n\}$ ，构造 $n$ 棵只有根结点的二叉树。
- ✓在森林中选取两棵根结点权值最小的树作左右子树，构造一棵新的二叉树，置新二叉树根结点权值为其左右子树根结点权值之和。
- ✓在森林中删除这两棵树，同时将新得到的二叉树加入森林中。
- ✓重复上述两步，直到只含一棵树为止，这棵树即霍夫曼树。

一棵有 $n$ 个叶子结点的Huffman树有  **$2n-1$**  个结点

✓ 采用顺序存储结构——一维结构数组

✓ 结点类型定义

```
typedef struct  
{ int weght;  
  int parent,lch,rch;  
}*HuffmanTree;
```



# 算法

```
void CreatHuffmanTree (HuffmanTree HT,int n){
```

```
if(n<=1)return;
```

```
m=2*n-1;
```

```
HT=new HTNode[m+1];
```

```
for(i=1;i<=m;++i)
```

```
{HT[i].lch=0;HT[i].rch=0;
```

```
for(i=1;i<=n;++i)cin>>HT[i].
```

例:设 $n=8$ ,  $w=\{5,29,7,8,14,23,3,11\}$

试设计 huffman code ( $m=2*8-1=15$ )

	weight	parent	lch	ild	rchild
1	5	0	0	0	0
·	29	0	0	0	0
·	7	0	0	0	0
·	8	0	0	0	0
·	14	0	0	0	0
·	23	0	0	0	0
8	3	0	0	0	0
	11	0	0	0	0
9		0	0	0	0
·		0	0	0	0
·		0	0	0	0
15		0	0	0	0

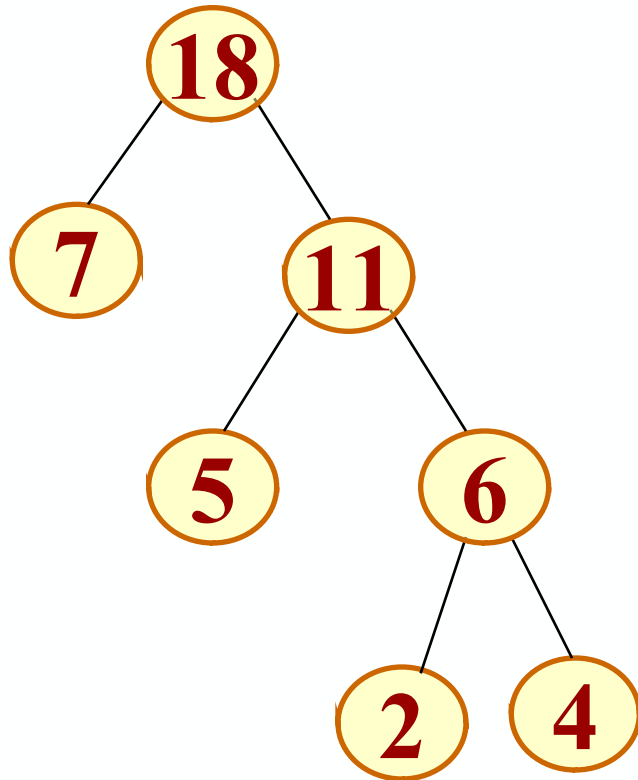


```
for( i=n+1;i<=m;++i)    //构造 Huffman树
{ Select(HT,i-1, s1, s2);
    //在HT[k](1≤k≤i-1)中选择两个其双亲域为0,
    // 且权值最小的结点,
    // 并返回它们在HT中的序号s1和s2
    HT[s1].parent=i; HT[s2].parent=i;
    //表示从F中删除s1,s2
    HT[i].lch=s1; HT[i].rch=s2 ;
    //s1,s2分别作为i的左右孩子
    HT[i].weight=HT[s1].weight + HT[s2].weight;
    //i 的权值为左右孩子权值之和
}
}
```

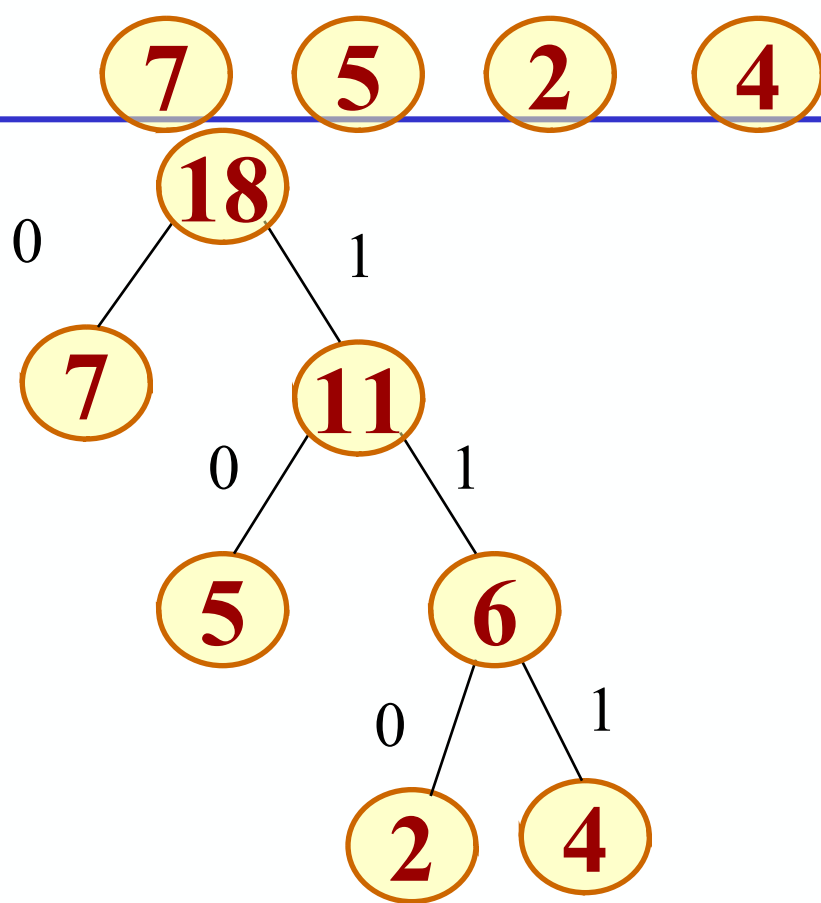
构造Huffman tree后, HT为:

	weight	parent	lchild	rchild
1	5	9	0	0
	29	14	0	0
.	7	10	0	0
	8	10	0	0
.	14	12	0	0
	23	13	0	0
.	3	9	0	0
8	11	11	0	0
9	8	11	1	74
	15	12	3	9
.	19	13	85	
	29	14	62	10
.	42	15		11
	58	15		12
15	100	0	13	14

7 5 2 4



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

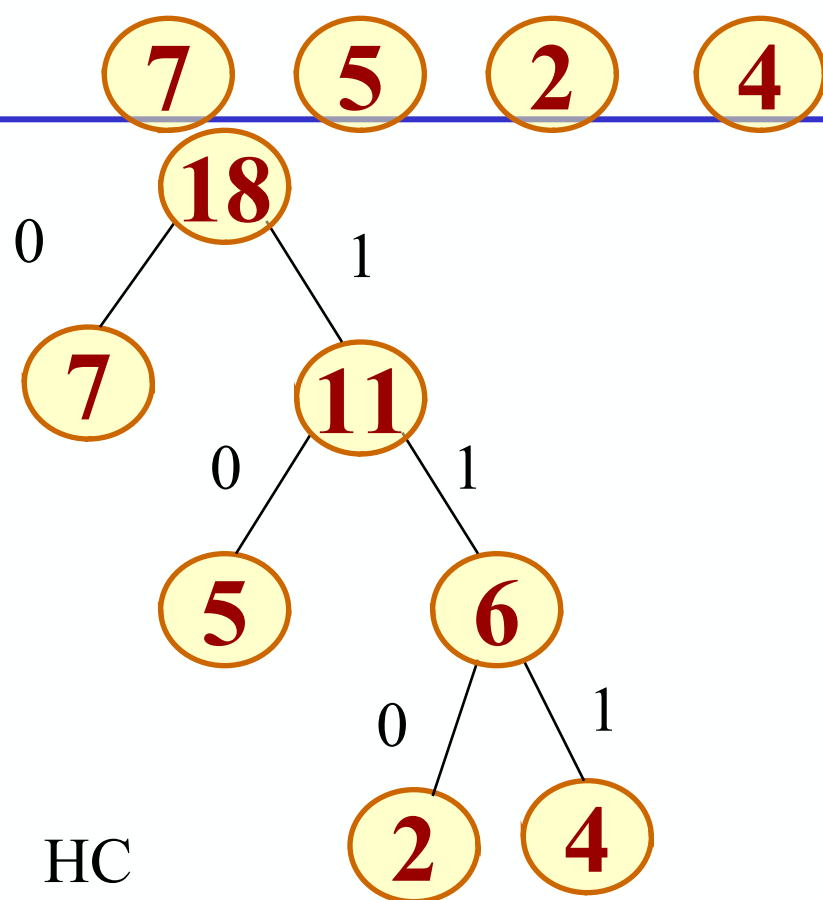


	weight	parent	lchild	rchild
1	7	7	0	0
2	5	6	0	0
3	2	5	0	0
4	4	5	0	0
5	6	6	3	4
6	11	7	2	5
7	18	0	1	6

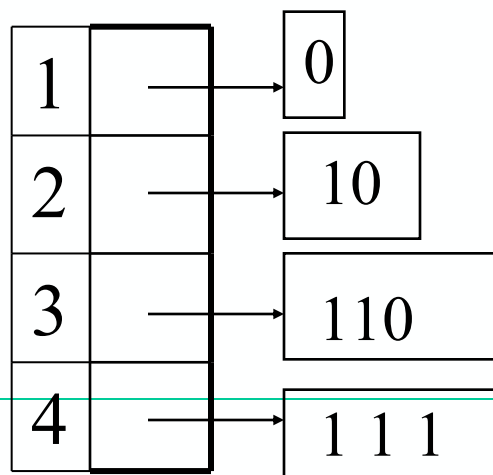
```

void CreatHuffmanCode(HuffmanTree HT, HuffmanCode &HC, int n){
//从叶子到根逆向求每个字符的赫夫曼编码，存储在编码表HC中
HC=new char *[n+1];           //分配n个字符编码的头指针矢量
cd=new char [n];               //分配临时存放编码的动态数组空间
cd[n-1]='\0';                 //编码结束符
for(i=1; i<=n; ++i){          //逐个字符求赫夫曼编码
    start=n-1; c=i; f=HT[i].parent;
    while(f!=0){               //从叶子结点开始向上回溯，直到根结点
        --start;               //回溯一次start向前指一个位置
        if (HT[f].lchild==c) cd[start]='0'; //结点c是f的左孩子，则生成代码0
        else cd[start]='1';      //结点c是f的右孩子，则生成代码1
        c=f; f=HT[f].parent;     //继续向上回溯
    }                           //求出第i个字符的编码
    HC[i]= new char [n-start];   // 为第i 个字符编码分配空间
    strcpy(HC[i], &cd[start]); //将求得的编码从临时空间cd复制到HC的当前行中
}
delete cd;                     //释放临时空间
} // CreatHuffmanCode

```



HC



	weight	parent	lchild	rchild
1	7	7	0	0
2	5	6	0	0
3	2	5	0	0
4	4	5	0	0
5	6	6	3	4
6	11	7	2	5
7	18	0	1	6

- 霍夫曼编码是**不等长编码**
- 霍夫曼编码是**前缀编码**，即任一字符的编码都不是另一字符编码的前缀
- 霍夫曼编码树中没有度为1的结点。若叶子结点的个数为 $n$ ，则霍夫曼编码树的**结点总数为  $2n-1$**
- 发送过程：根据由**霍夫曼树得到的编码表**送出字符数据
- 接收过程：按**左0、右1**的规定，从根结点走到一个叶结点，完成一个字符的译码。反复此过程，直到接收数据结束

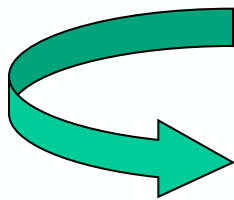
# Huffman编码举例

例：假设用于通信的电文仅由8个字母 {a, b, c, d, e, f, g, h} 构成，它们在电文中出现的概率分别为{ 0.07, 0.19, 0.02, 0.06, 0.32, 0.03, 0.21, 0.10 }，试为这8个字母设计霍夫曼编码。如果用0~7的二进制编码方案又如何？

解：先将概率放大100倍，以方便构造霍夫曼树。

放大后的权值集合  $w = \{ 7, 19, 2, 6, 32, 3, 21, 10 \}$ ，

按霍夫曼树构造规则（合并、删除、替换），可得到霍夫曼树。



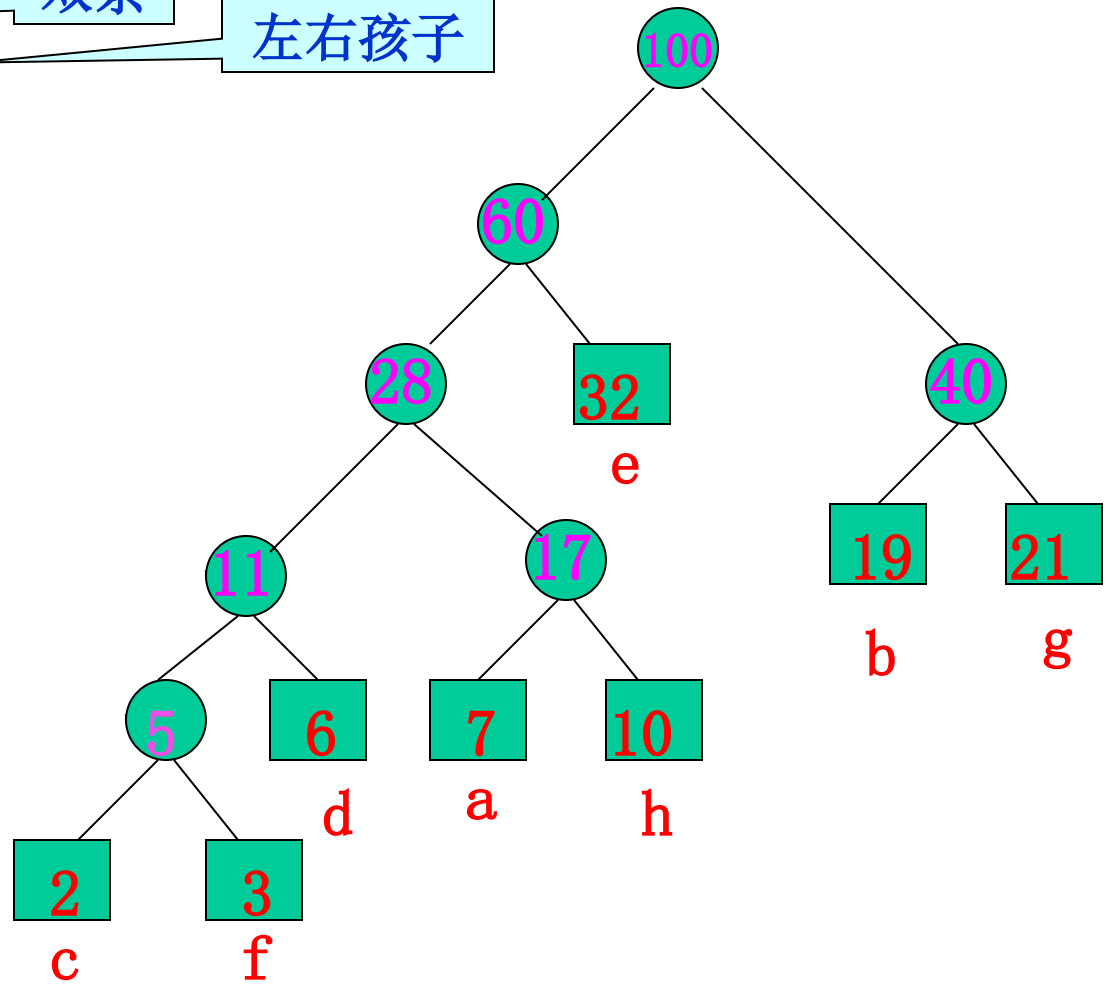


w = { 7, 19, 2, 6, 32, 3, 21, 10 } 在机内存储形式为:

	w	p	l	r
1	7✓	11	0	0
2	19✓	13	0	0
3	2✓	9	0	0
4	6✓	10	0	0
5	32✓	14	0	0
6	3✓	9	0	0
7	21✓	13	0	0
8	10✓	11	0	0
9	5✓	10	3	6
10	11✓	12	9	4
11	17✓	12	1	8
12	28✓	14	10	11
13	40	15	2	7
14	60	15	12	5
15	100	0	13	14

双亲

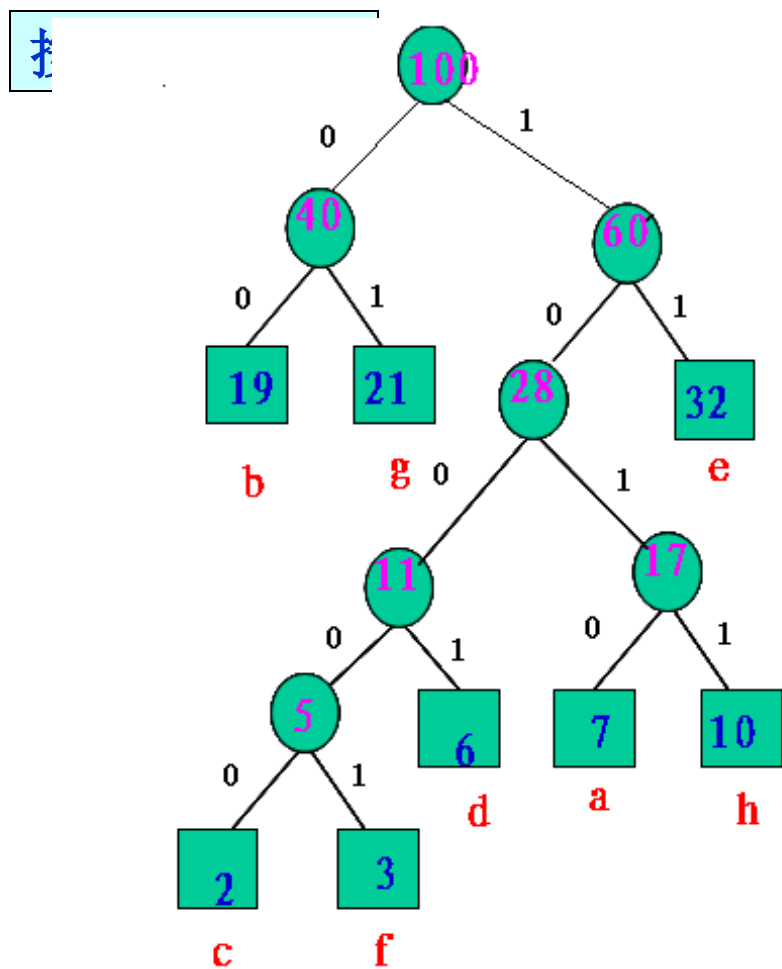
左右孩子



# 对应霍夫曼编码:

符	编码	频率
a	1010	0.07
b	00	0.19
c	10000	0.02
d	1001	0.06
e	11	0.32
f	10001	0.03
g	01	0.21
h	1011	0.10

符	编码	频率
a	000	0.07
b	001	0.19
c	010	0.02
d	011	0.06
e	100	0.32
f	101	0.03
g	110	0.21
h	111	0.10



$$\text{Huffman码的WPL} = 2(0.19+0.32+0.21) + 4(0.07+0.06+0.10) + 5(0.02+0.03) \\ = 1.44+0.92+0.25 = 2.61$$

$$\text{二进制等长码的WPL} = 3(0.19+0.32+0.21+0.07+0.06+0.10+0.02+0.03) = 3$$

# 霍夫曼树的应用

- 网络安全组播密钥的分配

<http://www.docin.com/p-13637283.html>

- 数据压缩

<http://www.educity.cn/zk/sjjg/200801051038191677.htm>

## 小结：哈夫曼树及其应用

### 1. Huffman算法的思路：

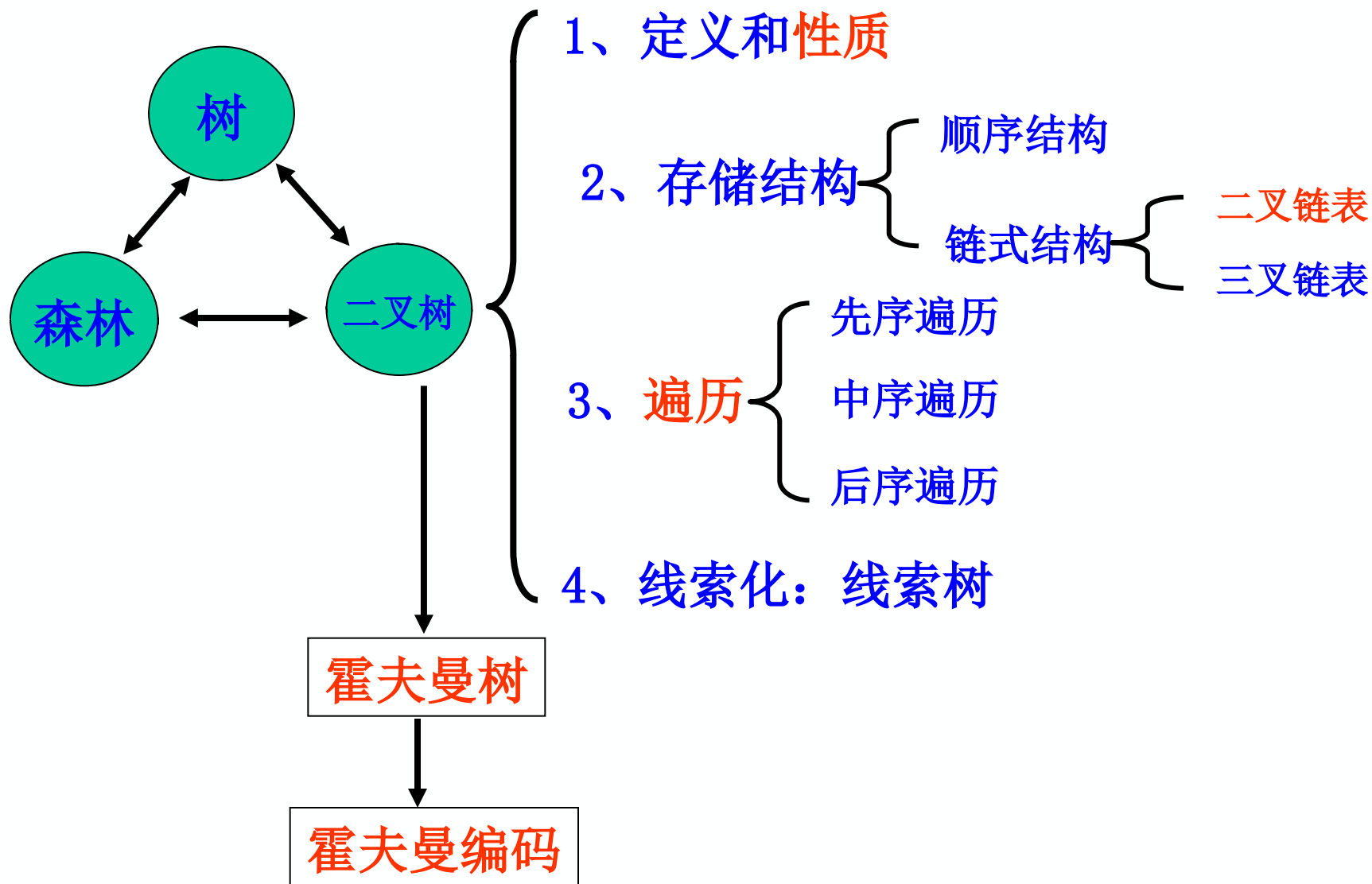
——权值大的结点用短路径，权值小的结点用长路径。

### 2. 构造Huffman树的步骤：

——对权值的合并、删除与替换

### 3. Huffman编码规则： 左“0” 右“1”

——又称为前缀码、最小冗余编码、紧致码等等，它是数据压缩学的基础。



# 本章小结

1. 熟练掌握二叉树的结构特性，了解相应的证明方法。
2. 熟悉二叉树的各种存储结构的特点及适用范围。
3. 掌握各种遍历策略的递归算法，灵活运用遍历算法实现二叉树的其它操作。

4. 熟练掌握二叉树的线索化过程以及在中序线索化树上找给定结点的前驱和后继的方法。

5. 熟悉树的各种存储结构及其特点，掌握树和森林与二叉树的转换方法。掌握 1 至 2 种建立二叉树和树的存储结构的方法。

5. 了解最优树的特性，掌握建立最优树和霍夫曼编码的方法。

# 上机实验说明：设字符集为26个英文字母，其出现频度如下表所示。

算法与数据结构

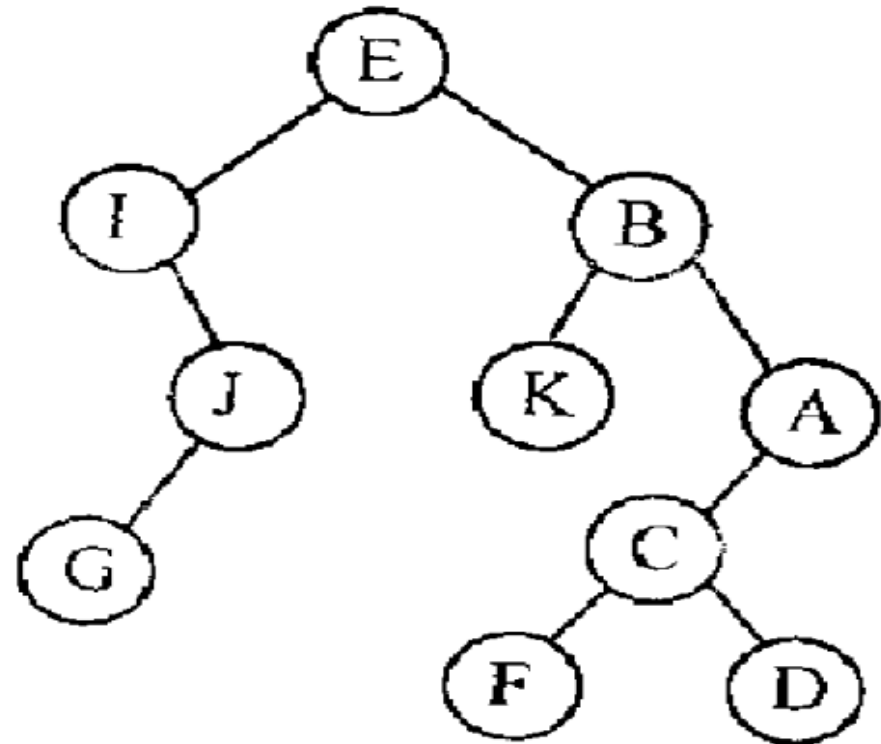
**要求：先建Huffman树，再利用此树对任一字符串文件进行编码和译码——即设计一个Huffman编/译码器**

字符	空格	a	b	c	d	e	f	g	h	i
频度	186	64	13	22	32	103	21	15	47	57
字符	j	k	l	m	n	o	p	q	r	s
频度	1	5	32	20	57	63	15	1	48	51
字符	t	u	v	w	x	y	z			
频度	80	23	8	18	1	16	1			



# 作业

- 画出二叉树对应的森林



# 作业

某子系统在通信联络中只可能出现8种字符，其出现的概率分别为0.06,0.28,0.07,0.08,0.14,0.23,0.03,0.11。（10分）

请：（1）画出赫夫曼树

（2）设计赫夫曼编码

（3）计算其带权路径长度WPL

# 应用

1.

# 1.统计二叉树中叶子节点的数量

```
void CountLeaf (BiTree T, int& count){  
    if ( T ) {  
        if ((!T->lchild)&& (!T->rchild))  
            count++;    // 对叶子结点计数  
        CountLeaf( T->lchild, count);  
        CountLeaf( T->rchild, count);  
    } // if  
} // CountLeaf
```

## 2.统计二叉树中节点的数量

```
void Count_1 (BiTree T, int& count){  
    if ( T ) {  
        count++;  
        Count_1( T->lchild, count);  
        Count_1( T->rchild, count);  
    } // if  
} // Count_1
```

## 2.统计二叉树中节点的数量

```
int Count_2 (BiTree T){  
    if ( T ==NULL)  return 0;  
else  
    {  
        return Count_2( T->lchild)+Count_2( T->rchild)+1;  
    }  
}
```

### 3.统计二叉树中节点数据值为K的节点数量

```
void CountK (BiTree T, int& count){  
    if ( T ) {  
        if ((T->data==k))  
            count++;    // 计数  
        CountLeaf( T->lchild, count);  
        CountLeaf( T->rchild, count);  
    } // if  
} // CountK
```