

DS—第六章

树和二叉树

Trees & Binary Trees

数据结构的三个方面

树在数据结构中的位置

数据的逻辑结构

A . 线性结构

线性表

栈和队列

串

数组和广义表

数据的存储结构

B . 非线性结
构

树形结构

图形结构

A 顺序存储

B 链式存储

数据的运算：检索、排序、插入、删除、修改等。

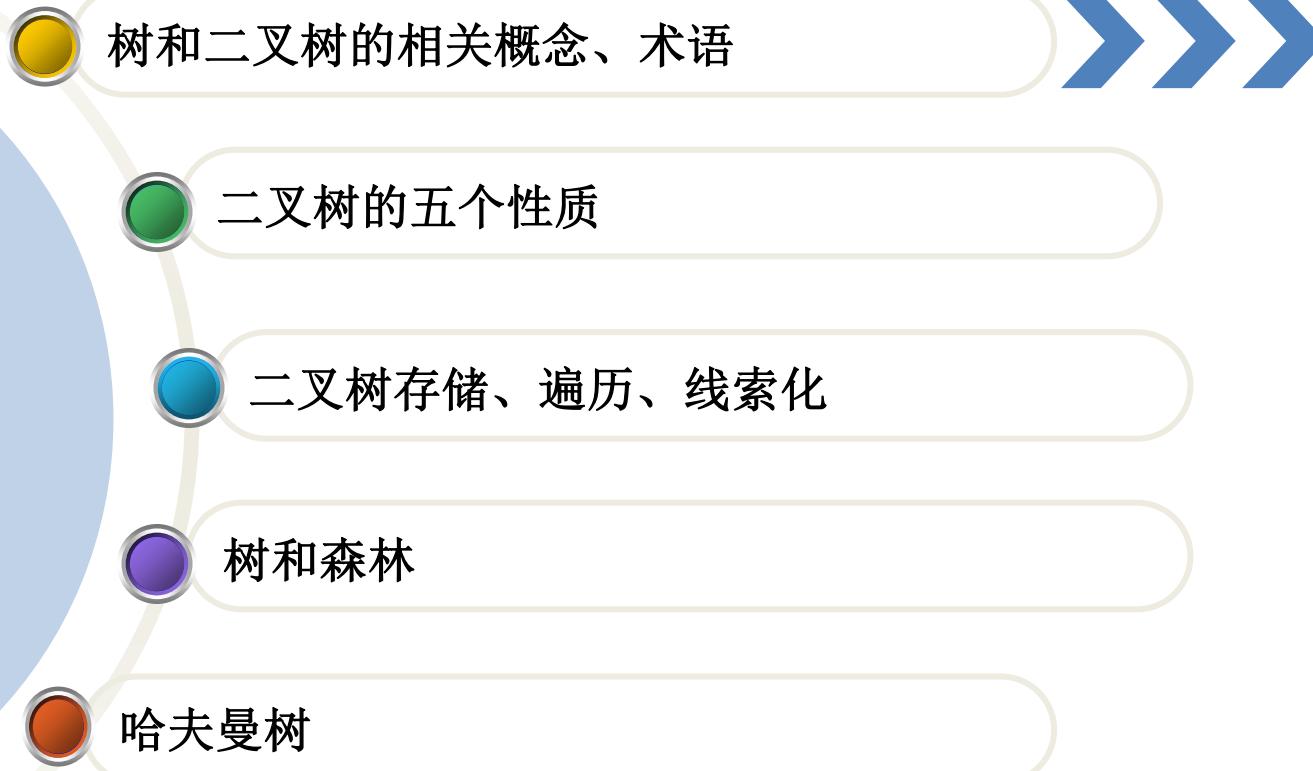
树和二叉树的相关概念、术语

二叉树的五个性质

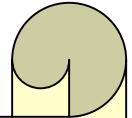
二叉树存储、遍历、线索化

树和森林

哈夫曼树



树的定义



树(Tree)是 $n(n \geq 0)$ 个结点的有限集。若 $n=0$ ，
称为空树；若 $n > 0$ ，则它满足如下两个条件：

- (1) 有且仅有一个特定的称为**根** (Root) 的结点；
- (2) 其余结点可分为 $m (m \geq 0)$ 个互不相交的有限
集 $T_1, T_2, T_3, \dots, T_m$ ，其中每一个集合本身又
是一棵树，并称为根的**子树** (SubTree)。

显然，树的定义是一个递归的定义。

基本术语

1、结点的度(degree)

叶子(leaf)（终端结点）

分支结点（非终端结点）

内部结点（B、C、D、E、H）

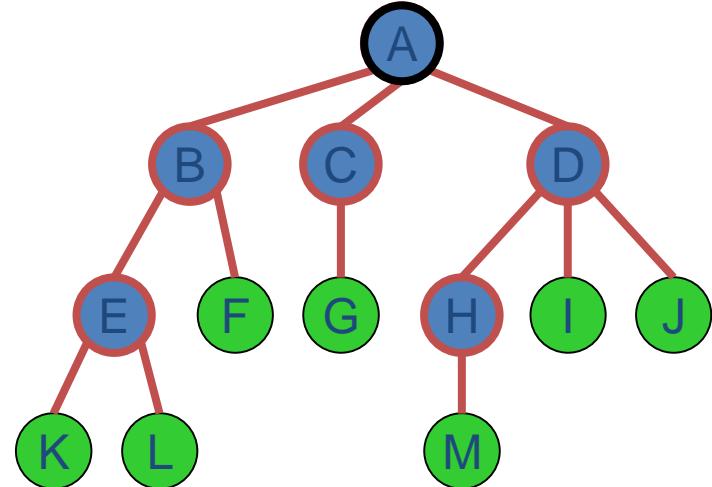
树的度（3）

2、结点的孩子(child)

双亲(parent)（D为H、I、J的双亲）

兄弟(sibling)(H、I、J互为兄弟)

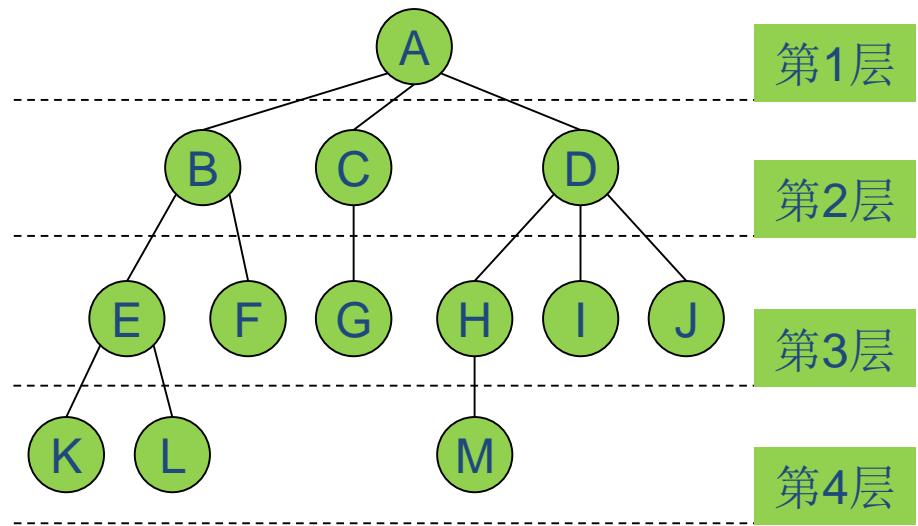
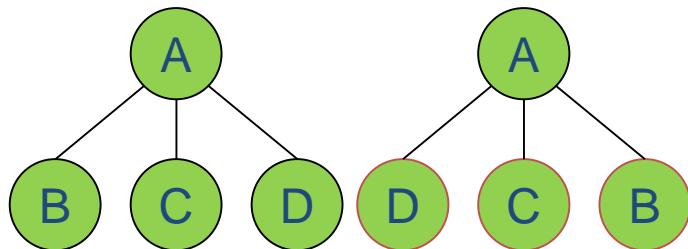
祖先，子孙(B的子孙为E、K、L、F)



3、结点的层次

- 根结点为第一层。
- 某结点在第 i 层，其孩子在第 $i+1$ 层。
- 树的深度(depth)
- 堂兄弟

4、有序树和无序树



5、森林(forest)是 m ($m \geq 0$) 棵互不相交的树的集合。

树的表示形式

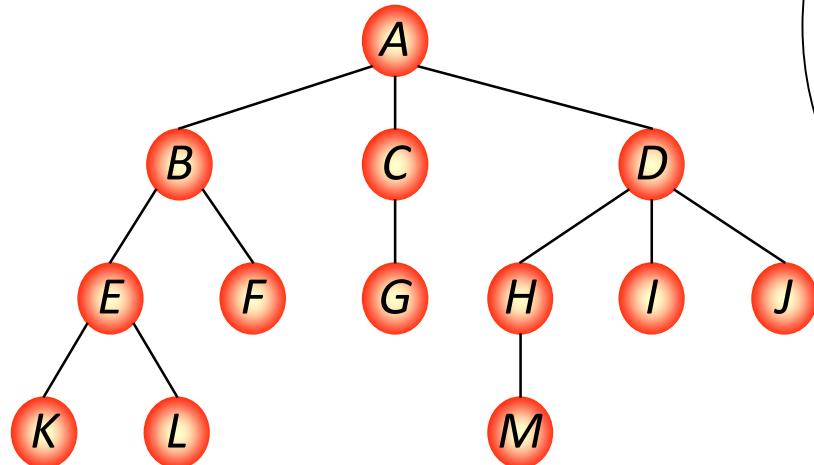
1. 树形表示法

\emptyset

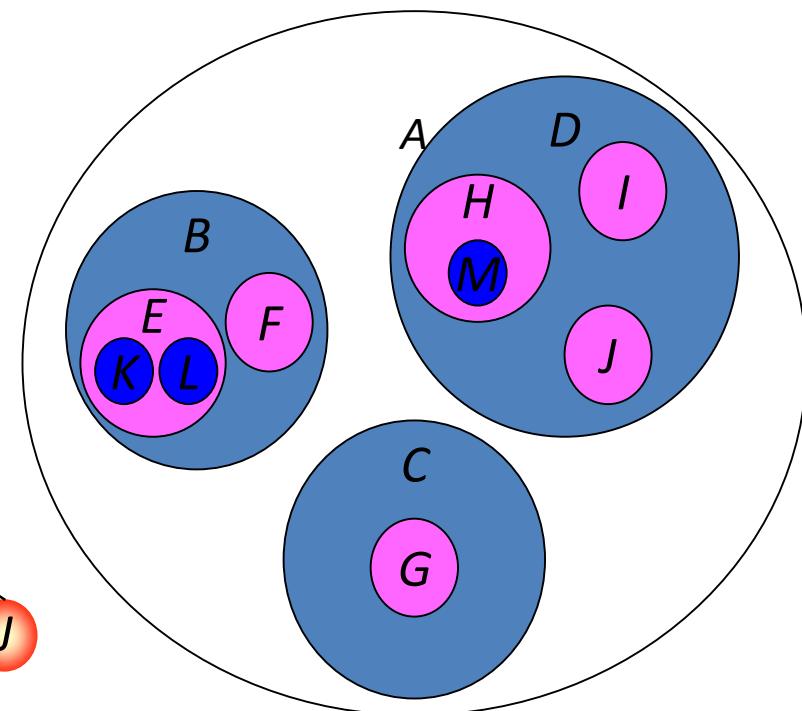
A 空树

A

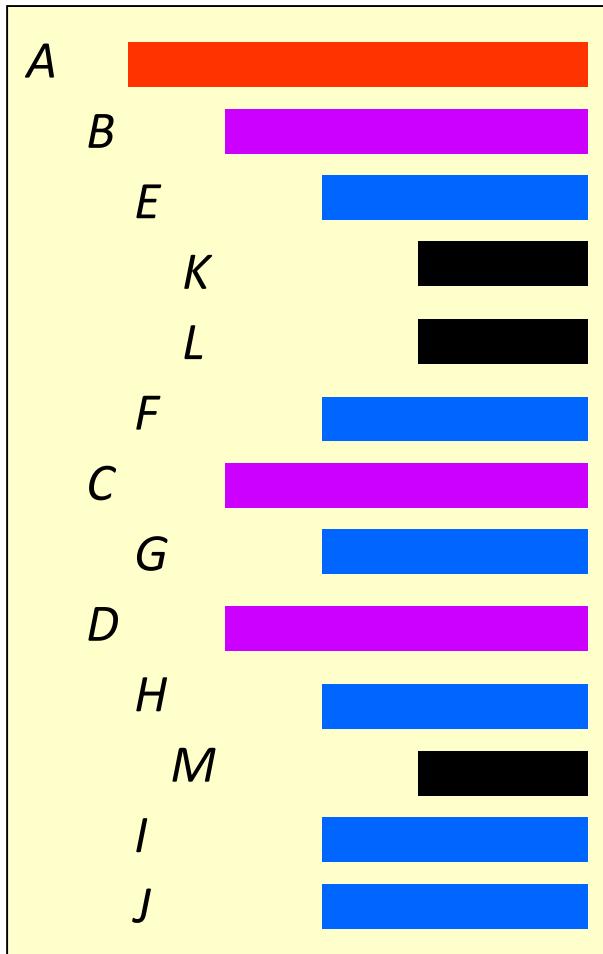
B 仅含有根结点的树



2. 嵌套集合（文氏）表示法

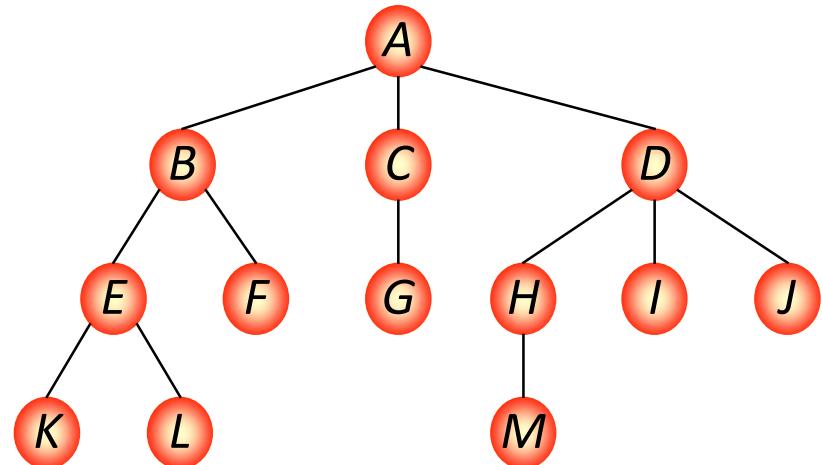


3. 凹入表示法



4. 广义表表示法

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



树的抽象数据类型定义

ADT Tree {

 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 : (略)

 基本操作 P:

 {**结构初始化**}

 InitTree (&T);

 操作结果：构造空树 T。

 CreateTree (&T, definition) ;

 初始条件： definition 给出树 T 的定义。

 操作结果：按 definition 构造树 T。

{销毁结构}

DestroyTree (&T);

初始条件：树 T 存在。

操作结果：销毁树 T。

{引用型操作}

TreeEmpty (T)

初始条件：树 T 存在。

操作结果：若 T 为空树，则返回 TRUE，否则 FALSE。

TreeDepth (T)

初始条件：树 T 存在。

操作结果：返回 T 的深度。

`Root (T)`

初始条件：树 T 存在。

操作结果：返回 T 的根。

`Value (T, cur_e);`

初始条件：树 T 存在， cur_e 是 T 中某个结点。

操作结果：返回 cur_e 的值。

`Assign (T, cur_e, value)`

初始条件：树 T 存在， cur_e 是 T 中某个结点。

操作结果：结点 cur_e 赋值为 value。

Parent (T, cur_e)

初始条件：树 T 存在， cur_e 是 T 中某个结点。

操作结果：若 cur_e 是 T 的非根结点，则返回它的双亲，否则函数值为 “空” 。

LeftChild (T, cur_e)

初始条件：树 T 存在， cur_e 是 T 中某个结点。

操作结果：若 cur_e 是 T 的非叶子结点，则返回它的最左孩子，否则返回 “空” 。

RightSibling (T, cur_e)

初始条件：树 T 存在， cur_e 是 T 中某个结点。

操作结果：若 cur_e 有右兄弟，则返回它的右兄弟，
否则函数值为 “空” 。

TraverseTree (T, Visit()) ;

初始条件：树 T 存在， Visit 是对结点操作的函数。

操作结果：按某种次序对 T 的每个结点调用函数
Visit () 一次且至多一次。一旦 Visit ()
失败，则操作失败。

{加工型操作}

ClearTree (&T);

初始条件：树 T 存在。

操作结果：将树 T 清为空树。

InsertChild (&T, &p, i, c);

初始条件：树 T 存在， p 指向 T 中某个结点， $1 \leq i \leq p$

所指结点的度 + 1，非空树 c 与 T 不相交。

操作结果：插入 c 为 T 中 p 指结点的第 i 棵子树。

DeleteChild (&T, &p, i);

初始条件：树 T 存在， p 指向 T 中某个结点，

$1 \leq i \leq p$ 所指结点的度。

操作结果：删除 T 中 p 所指结点的第 i 棵子树。

}ADT Tree

二叉树

二叉树在树结构的应用中起着非常重要的作用，因为对二叉树的许多操作算法简单，而任何树均可与二叉树相互转换，这样就解决了树的存储结构及其运算中存在的复杂性。

● 定义

二叉树是 $n (n \geq 0)$ 个结点的有限集，它或者是空集 ($n = 0$)，或者由一个根结点及两棵互不相交的分别称作这个根的左子树和右子树的二叉树组成。

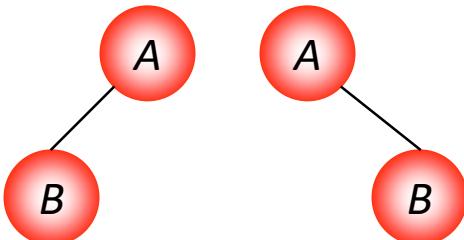
● 特点

- 1、每个结点最多有俩孩子（**二叉树中不存在度大于 2 的结点**）。
- 2、子树有左右之分，其次序不能颠倒。
- 3、二叉树可以是空集合，根可以有空的左子树或空的右子树。

注

二叉树**不是**树的特殊情况，它们是两个概念。

二叉树结点的子树要区分左子树和右子树，即使只有一棵子树也要进行区分，说明它是左子树，还是右子树。树当结点只有一个孩子时，就无须区分它是左还是右。（也就是二叉树每个结点位置或者说次序都是固定的，可以是空，但是不可以说它没有位置，而树的结点位置是相对于别的结点来说的，没有别的结点时，它就无所谓左右了），因此二者是不同的。这是二叉树与树的最主要的差别。

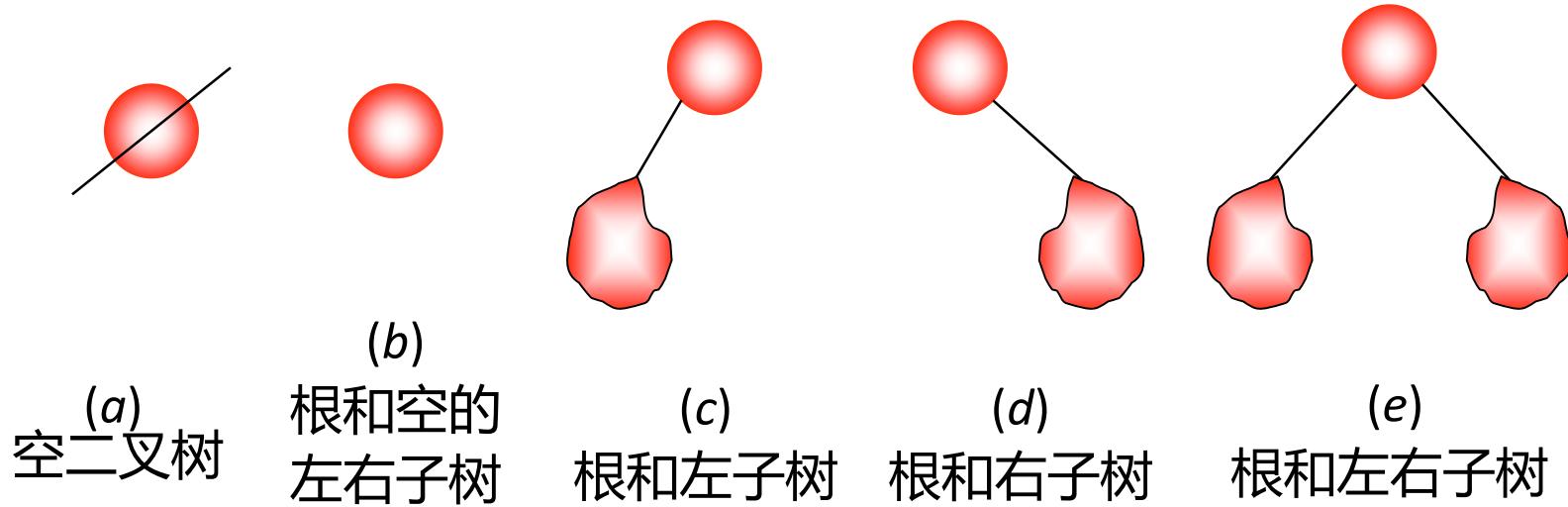


具有两个结点的二叉树有两种状态



具有两个结点的树只有一种状态

二叉树的 5 种基本形态



注：虽然二叉树与树概念不同，
但有关树的基本术语对二叉树都适用。

二叉树的抽象数据类型定义：

ADT BinaryTree {

 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 : (略)

 基本操作 P :

{**结构初始化**}

 InitBiTree(&T);

 操作结果：构造空二叉树 T。

 CreateBiTree(&T, definition);

 初始条件： definition 给出二叉树 T 的定义。

 操作结果：按 definition 构造二叉树 T。

{销毁结构}

DestroyBiTree(&T);

初始条件：二叉树 T 存在。

操作结果：销毁二叉树 T。

{引用型操作}

BiTreeEmpty(T);

初始条件：二叉树 T 存在。

操作结果：若 T 为空二叉树，则返回 TRUE，否则返回 FALSE。

BiTreeDepth(T);

初始条件：二叉树 T 存在。

操作结果：返回 T 的深度。

$\text{Root}(T);$

初始条件：二叉树 T 存在。

操作结果：返回 T 的根。

$\text{Value}(T, e);$

初始条件：二叉树 T 存在， e 是 T 中某个结点。

操作结果：返回 e 的值。

$\text{Parent}(T, e);$

初始条件：二叉树 T 存在， e 是 T 中某个结点。

操作结果：若 e 是 T 的非根结点，则返回它的双亲，
否则返回 “空”。

$\text{LeftChild}(T, e);$

初始条件：二叉树 T 存在， e 是 T 中某个结点。

操作结果：返回 e 的左孩子。若 e 无左孩子则返回 “空”。

`RightChild(T, e);`

初始条件：二叉树 T 存在，e 是 T 中某个结点。

操作结果：返回 e 的右孩子。若 e 无右孩子则返回“空”。

`LeftSibling(T, e);`

初始条件：二叉树 T 存在，e 是 T 中某个结点。

操作结果：返回 e 的左兄弟。若 e 是其双亲的左孩子或
无左兄弟，则返回“空”。

`RightSibling(T, e);`

初始条件：二叉树 T 存在，e 是 T 的结点。

操作结果：返回 e 的右兄弟。若 e 是其双亲的右孩子或
无右兄弟，则返回“空”。

PreOrderTraverse(T, visit());

初始条件：二叉树 T 存在，visit 是对结点操作的应用函数。

操作结果：**先序遍历** T，对每个结点调用函数 visit 一次
且仅一次。一旦 visit() 失败，则操作失败。

InOrderTraverse(T, visit());

初始条件：二叉树 T 存在，visit 是对结点操作的应用函数。

操作结果：**中序遍历** T，对每个结点调用函数 Visit 一次
且仅一次。一旦 visit() 失败，则操作失败。

PostOrderTraverse(T, visit());

初始条件：二叉树T存在，visit 是对结点操作的应用函数。

操作结果：**后序遍历** T，对每个结点调用函数 visit 一次
且仅一次。一旦 visit() 失败，则操作失败。

```
LevelOrderTraverse(T, visit());
```

初始条件：二叉树 T 存在， visit 是对结点操作的应用函数。

操作结果：按层次遍历 T ，对每个结点调用函数 visit 一次且仅一次。一旦 $\text{visit}()$ 失败，则操作失败。

{加工型操作}

ClearBiTree(&T);

初始条件：二叉树 T 存在。

操作结果：将二叉树 T 清为空树。

Assign(&T, &e, value);

初始条件：二叉树 T 存在，e 是 T 中某个结点。

操作结果：结点 e 赋值为 value。

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

初始条件：二叉树 T 存在，p 指向 T 中某个结点，LR 为 0 或 1，非空二叉树 c 与 T 不相交且右子树为空。

操作结果：根据 LR 为 0 或 1，插入 c 为 T 中 p 所指结点的左或右子树。p 所指结点原有左或右子树成为 c 的右子树。

DeleteChild(&T, p, LR);

初始条件：二叉树 T 存在，p 指向 T 中某个结点，LR 为 0 或 1。

操作结果：根据 LR 为 0 或 1，删除 T 中 p 所指结点的左或右子树。

} ADT BinaryTree

树和二叉树的相关概念、术语

二叉树的五个性质



二叉树存储、遍历、线索化

树和森林

哈夫曼树

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 ($i \geq 1$)。

证：采用归纳法证明此性质。

归纳基础：当 $i = 1$ 时只有根结点， $2^{i-1} = 2^0 = 1$ ，命题成立。

归纳假设：设对所有的 j ($1 \leq j < i$)，命题成立，即第 j 层上至多有 2^{j-1} 个结点。需证明 $j = i$ 时命题也成立。

归纳证明：由归纳假设可知第 $i-1$ 层上至多有 2^{i-2} 个结点。
由于二叉树每个结点的度最大为 2，故在第 i 层上最大结点数为第 $i-1$ 层上最大结点数的 2 倍，即： $2 \times 2^{i-2} = 2^{i-1}$ 。证毕。

性质 2：深度为 k 的二叉树至多有 $2^k - 1$ 个结点 ($k \geq 1$)。

证：由性质 1 可知，深度为 k 的二叉树的最大结点数为：

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

证毕。

性质 3：对任何一棵二叉树 T ，如果其叶子数为 n_0 ，度为 2 的结点数为 n_2 ，则 $n_0 = n_2 + 1$ 。

证：设 n_1 为二叉树 T 中度为 1 的结点数。因为二叉树中所有结点的度均 ≤ 2 ，所以其结点总数为：

$$n = n_0 + n_1 + n_2$$

再看二叉树中的分支数，除根结点外，其余结点都只有一个分支进入，设 B 为分支总数，则有：

$$n = B + 1$$

因这些分支都是由度为 1 和 2 的结点射出的，所以有：

$$B = n_1 + 2n_2$$

于是有：
$$n = B + 1 = n_1 + 2n_2 + 1$$

所以有：
$$n_0 + n_1 + n_2 = n_1 + 2n_2 + 1$$

即：
$$n_0 = n_2 + 1 \quad \text{证毕。}$$

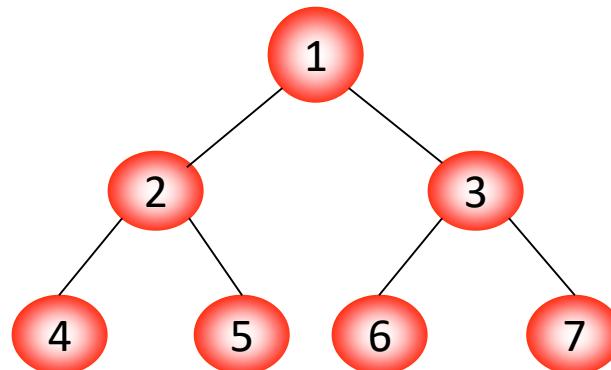
满二叉树 (Full binary tree)

一棵深度为 k 且有 $2^k - 1$ 个结点的二叉树
称为满二叉树。

特点：每一层上的结点数都达到最大。

叶子全部在最底层。

编号规则：从根结点开始，自上而下，自左而右。

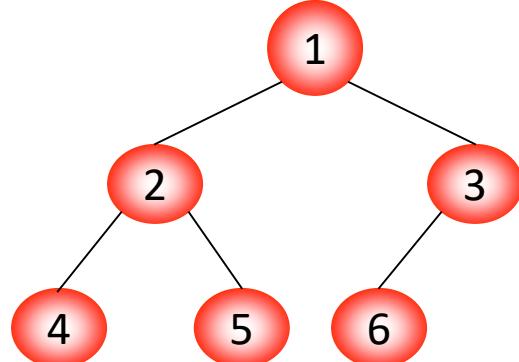


完全二叉树 (Complete binary tree)

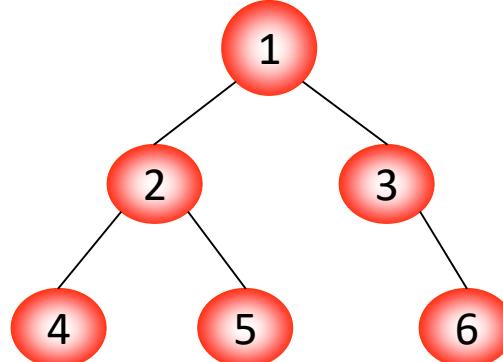
深度为 k 的具有 n 个结点的二叉树，当且仅当其每一个结点都与深度为 k 的满二叉树中编号为 $1 \sim n$ 的结点一一对应时，称之为完全二叉树。

特点：叶子只可能分布在层次最大的两层上。

对任一结点，如果其右子树的最大层次为 L ，则其左子树的最大层次必为 L 或 $L + 1$ 。



完全二叉树



非完全二叉树

满二叉树
一定是
完全二叉树

是定一不

完全二叉树的性质

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

证：假设此二叉树的深度为 k ，则根据性质 2 及完全二叉树的定义得到： $2^{k-1} - 1 < n \leq 2^k - 1$

或 $2^{k-1} \leq n < 2^k$

取对数得： $k - 1 \leq \log_2 n < k$

因为 k 是整数，所以有：

$$k = \lfloor \log_2 n \rfloor + 1$$



性质 5：如果对一棵有 n 个结点的完全二叉树 (深度为 $\lfloor \log_2 n \rfloor + 1$) 的结点按层序编号 (从第 1 层到第 $\lfloor \log_2 n \rfloor + 1$ 层，每层从左到右)，则对任一结点 i ($1 \leq i \leq n$)，有：

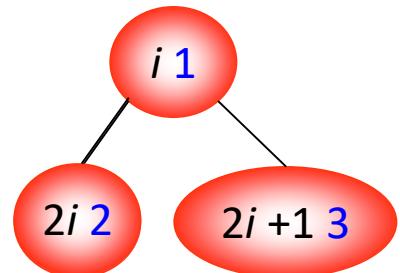
- (1) 如果 $i = 1$ ，则结点 i 是二叉树的根，无双亲；
如果 $i > 1$ ，则其双亲是结点 $\lfloor i / 2 \rfloor$ 。
- (2) 如果 $2i > n$ ，则结点 i 为叶子结点，无左孩子；
否则，其左孩子是结点 $2i$ 。
- (3) 如果 $2i + 1 > n$ ，则结点 i 无右孩子；否则，其
右孩子是结点 $2i + 1$ 。

证：(1) 可以从(2)和(3)推出，所以先证明(2)和(3)。

对于 $i=1$ ，由完全二叉树的定义，其左孩子是

结点 $2 = 2i$ ，若 $2 = 2i > n = 1$ ，即不存在结点 2 ，此

时，结点 i 无左孩子。**(2) 得证。**



结点 i 的右孩子也只能是结点 $3 = 2i + 1$ ，若

$3 = 2i + 1 > n$ ，即不存在结点 3 ，此时结点 i 无右孩子。

(3) 得证。

对于 $i > 1$ ，可分为两种情况：

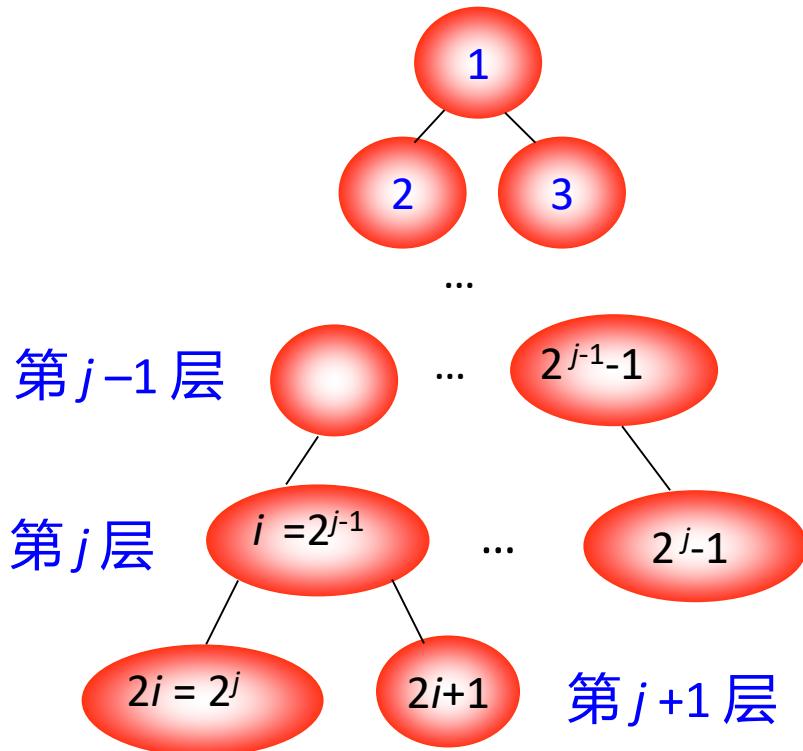
(1) 设第 j ($1 \leq j \leq \lfloor \log_2 n \rfloor$) 层的

首结点的编号为 i (由二叉树的

定义和性质 2 知 $i = 2^{j-1}$)，则其

左孩子必为第 $j+1$ 层的首结点，

其编号为 $2^j = 2 \times 2^{j-1} = 2i$ 。

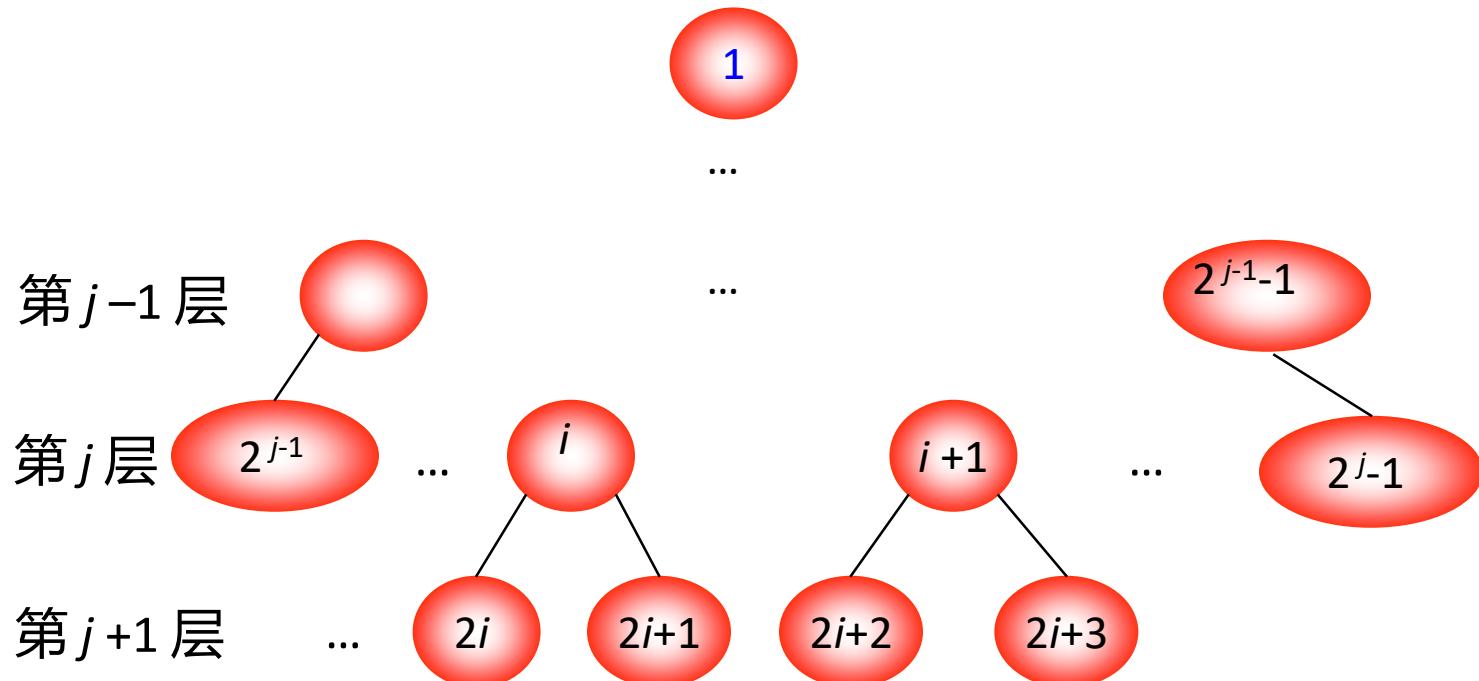


如果 $2i > n$ ，则无左孩子。 (2) 得证。

其右孩子必定为第 $j+1$ 层的第二个结点，

编号为 $2i+1$ 。若 $2i+1 > n$ ，则无右孩子。 (3) 得证。

(2) 设第 j ($1 \leq j \leq \lfloor \log_2 n \rfloor$) 层的某个结点的编号为 i ($2^{j-1} \leq i < 2^j - 1$)，且 $2i + 1 < n$ ，其左右孩子为 $2i$ 和 $2i + 1$ 。则编号为 $i + 1$ 的结点是编号为 i 的结点的右兄弟或堂兄弟。若它有左孩子则其编号必为： $2i + 2 = 2(i + 1)$ ，(2) 得证。若它有右孩子则其编号必为： $2i + 3 = 2(i + 1) + 1$ 。(3) 得证。



下面证明 (1)。

当 $i = 1$ 时：此结点就是根，因此无双亲。

当 $i > 1$ 时：

如果 i 为左孩子，且 i 的双亲为 p ，则有 $i = 2p$ ，

$p = i / 2 = \lfloor i / 2 \rfloor$ ，即 $\lfloor i / 2 \rfloor$ 是 i 的双亲；

如果 i 为右孩子，且 i 的双亲为 p ，则有 $i = 2p + 1$ ，

$p = (i - 1) / 2 = i / 2 - 1 / 2 = \lfloor i / 2 \rfloor$ ，即 $\lfloor i / 2 \rfloor$ 是 i 的双亲。

证毕。

树和二叉树的相关概念、术语

二叉树的五个性质

二叉树存储、遍历、线索化



树和森林

哈夫曼树

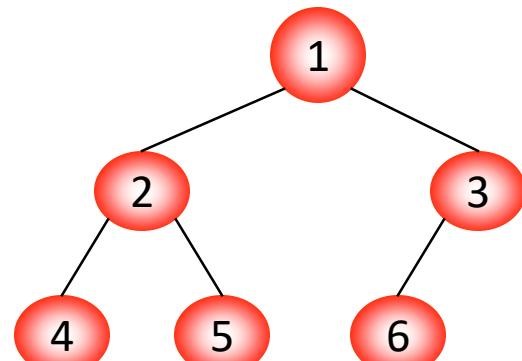
二叉树的存储结构

1、顺序存储结构

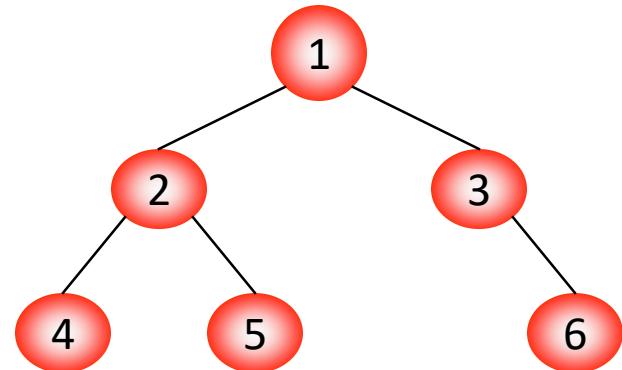
完全二叉树：用一组地址连续的存储单元依次**自上而下、自左至右**存储结点元素，即将编号为 i 的结点元素存储在一维数组中下标为 $i-1$ 的分量中。

一般二叉树：将其每个结点与完全二叉树上的结点相对照，存储在一维数组的相应分量中。

此顺序存储结构仅适用于完全二叉树



完全二叉树

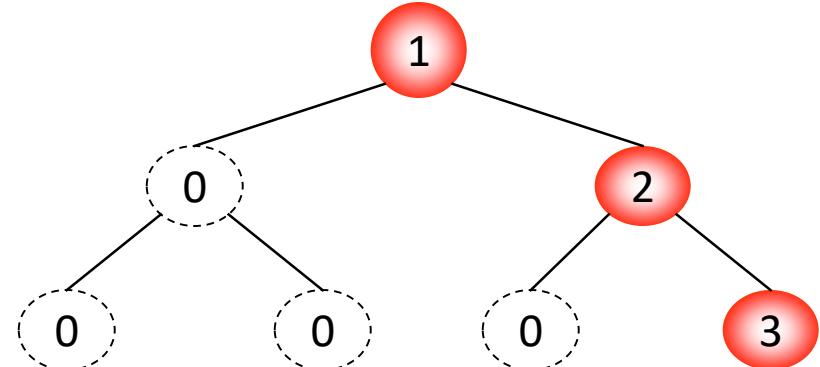


非完全二叉树



最坏情况：深度为 k 的且只有 k 个结点的右单支树需要长度为 $2^k - 1$ 的一维数组。

1	0	2	0	0	0	3
---	---	---	---	---	---	---



右单支树

表示方式：

```
#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

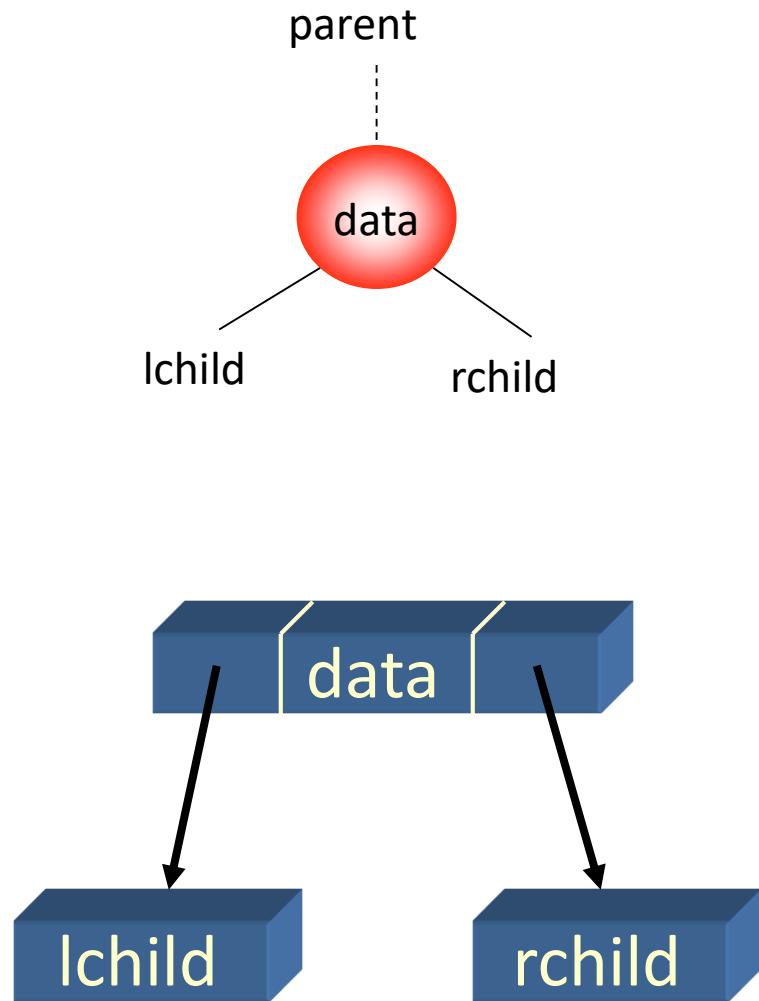
```
SqBiTree bt;
```

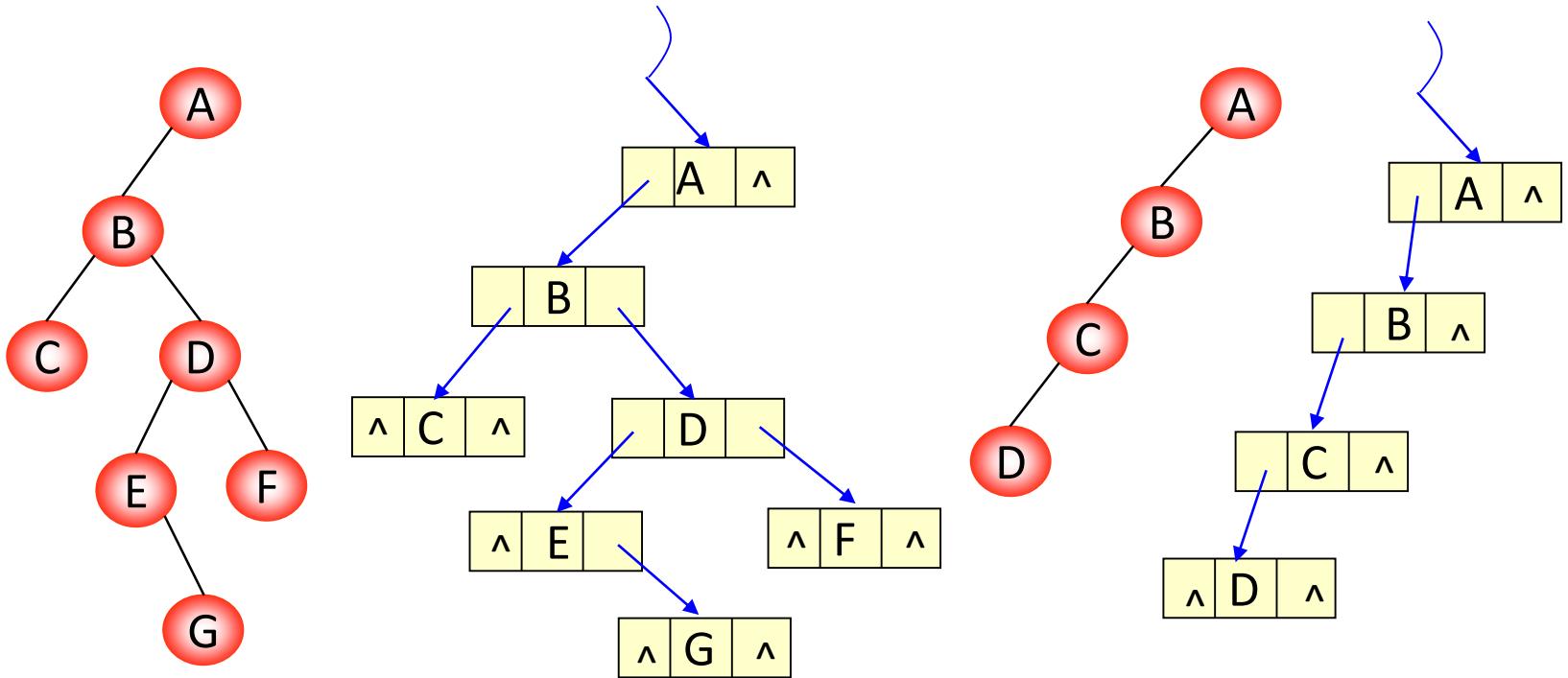
2、链式存储结构

- 二叉树结点的构成
- 存储方式



结点结构



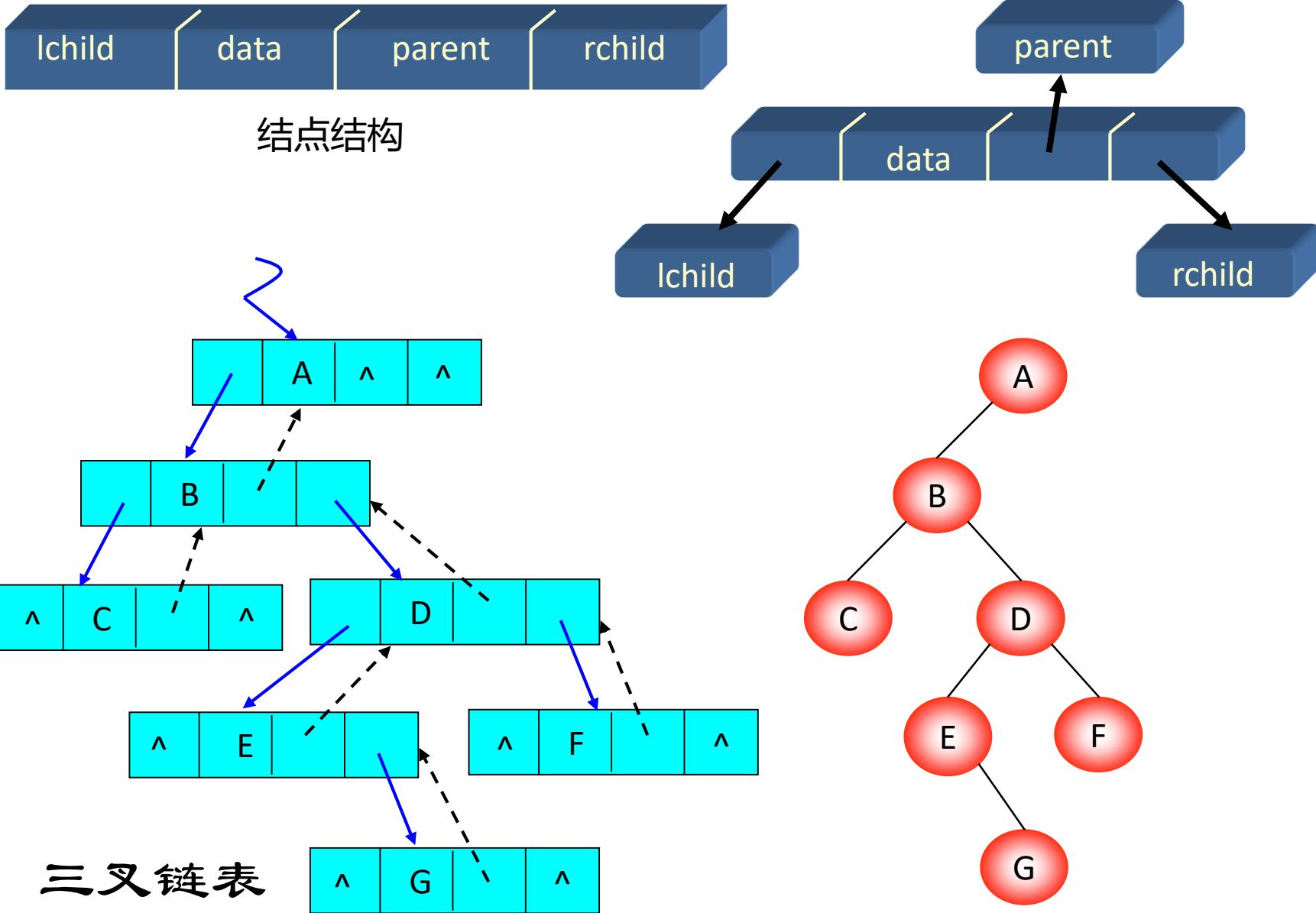


二叉链表

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;
```



遍历二叉树

(重点，是进行其他运算的基础)

● 遍历概念

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。

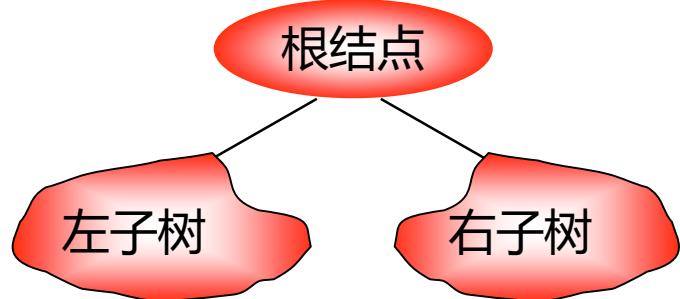
“访问”的含义很广，可以是对结点作各种处理，如：输出结点的信息、修改结点的数据值等，但要求这种访问不破坏原来的数据结构。

它是一个“引用型”操作

● 遍历目的

得到树中所有结点的一个**线性排列**。

● 遍历方法



依次遍历二叉树中的三个组成部分，便是遍历了整个二叉树

假设 : L : 遍历左子树 D : 访问根结点 R : 遍历右子树
则遍历整个二叉树方案共有：

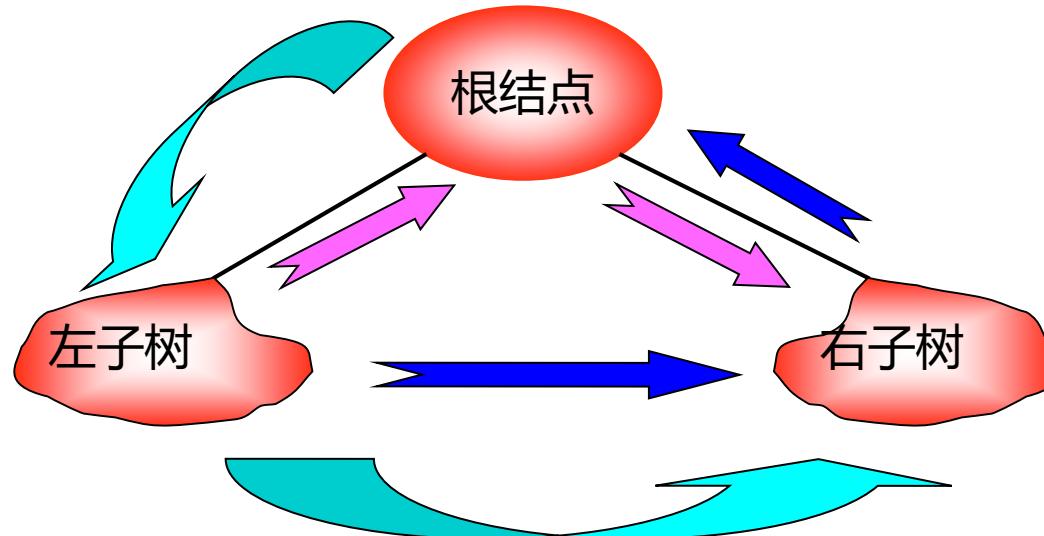
DLR、LDR、LRD、DRL、RDL、RLD 六种。

若规定先左后右，则只有前三种情况：

DLR —— 先（根）序遍历，

LDR —— 中（根）序遍历，

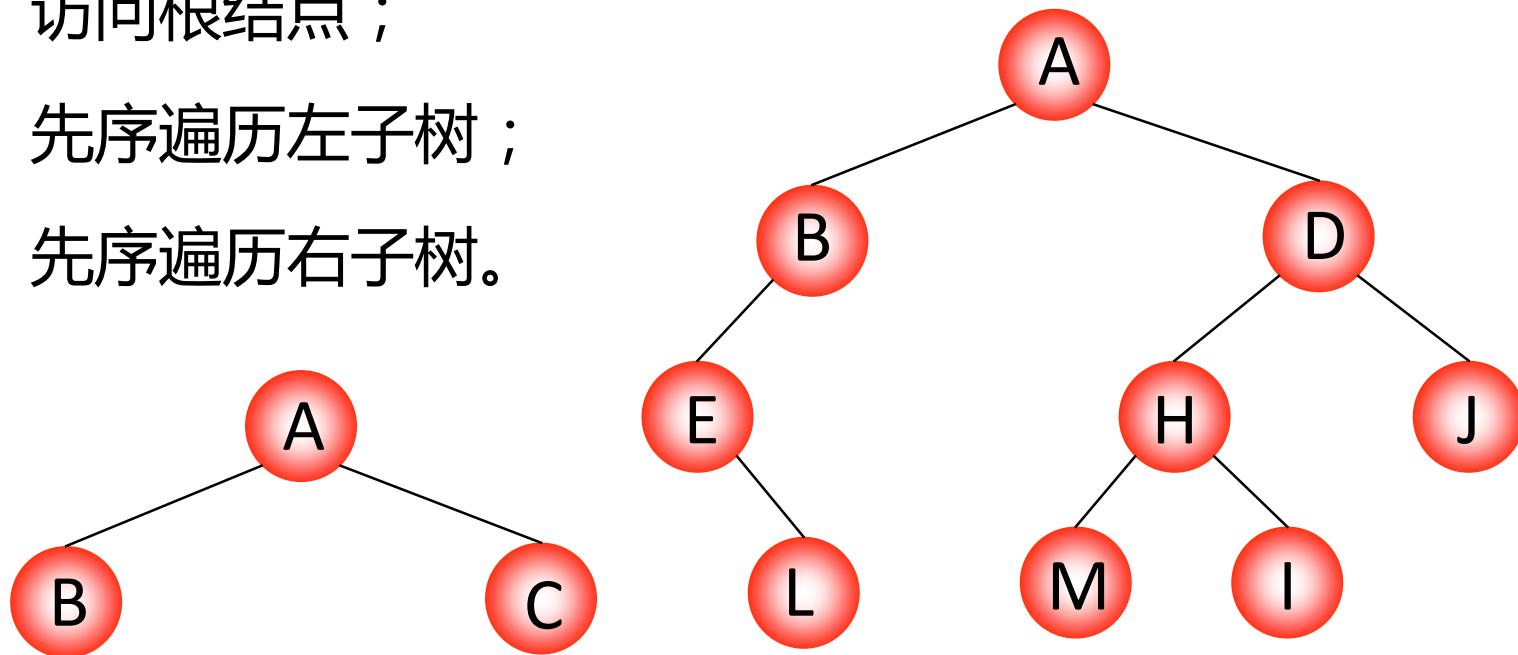
LRD —— 后（根）序遍历。



- 先序遍历二叉树的操作定义：

若二叉树为空，则空操作；否则

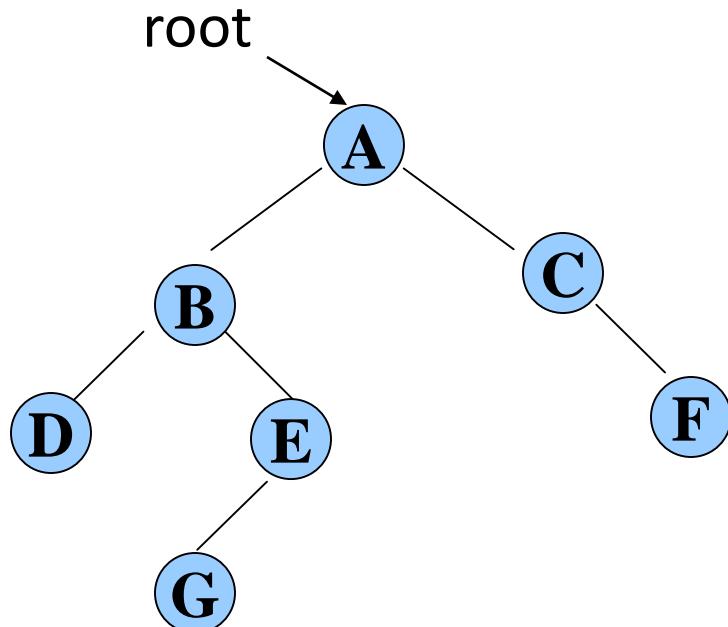
- (1) 访问根结点；
- (2) 先序遍历左子树；
- (3) 先序遍历右子树。



先序遍历的顺序为： ABC

先序遍历的顺序为：
ABELDHMIJ

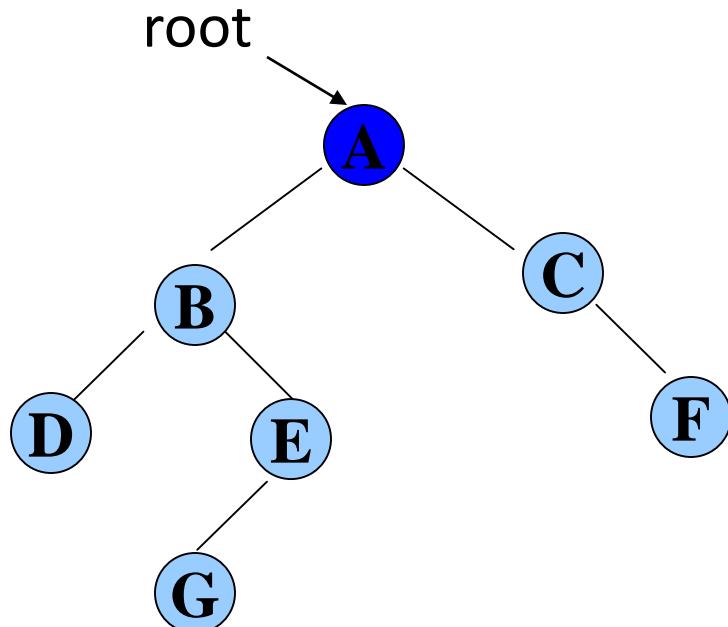
```
void preorder (BiTNode *root) {
    if (root!=NULL) {
        cout<< root->data; //访问根结点
        preorder(root->Lchild); //先序遍历根的左子树
        preorder(root->Rchild); //先序遍历根的右子树
    }//if
}//preorder
```



先序遍历序列:

root: A

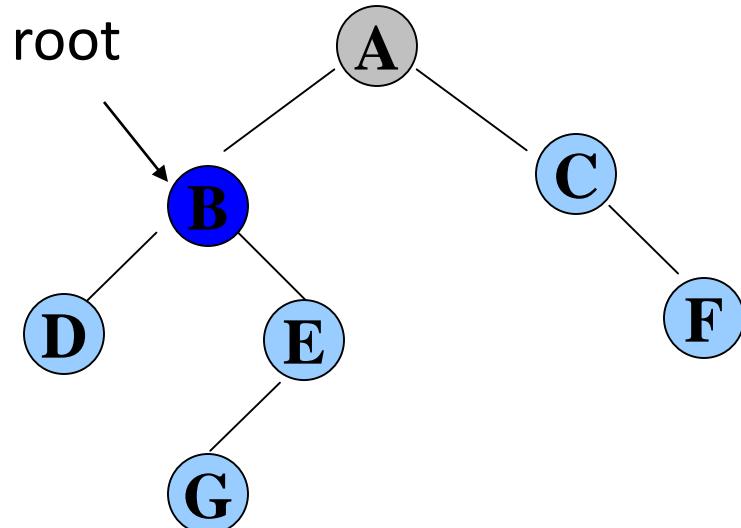
```
void preorder (BiTNode *root) {  
    if (root!=NULL) {  
        cout<< root->data; //访问根结点  
        preorder(root->Lchild); //先序遍历根的左子树  
        preorder(root->Rchild); //先序遍历根的右子树  
    }//if  
}//preorder
```



先序遍历序列: A

root: A

```
void preorder (BiTNode *root) {  
    if (root!=NULL) {  
        cout<< root->data; //访问根结点  
        preorder(root->Lchild); //先序遍历根的左子树  
        preorder(root->Rchild); //先序遍历根的右子树  
    }//if  
}//preorder
```

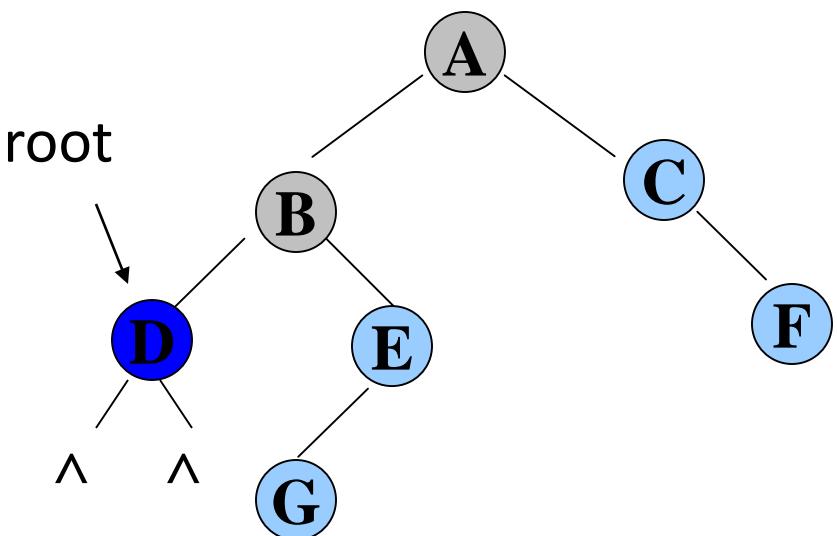


先序遍历序列: A B

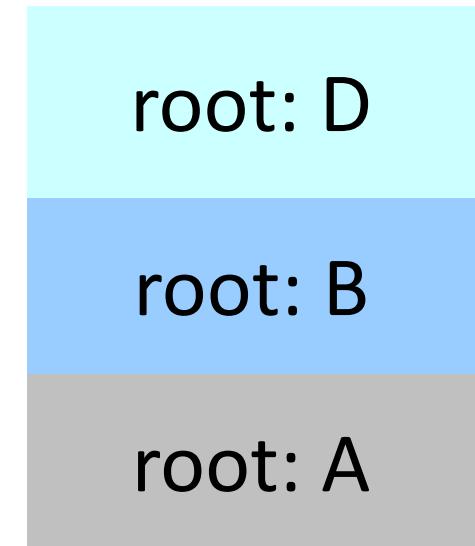
root: B

root: A

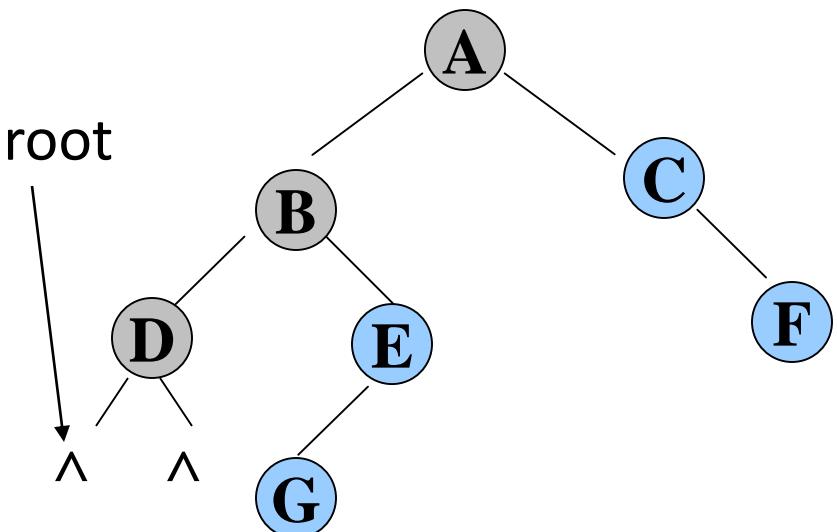
```
void preorder (BiTNode *root) {  
    if (root!=NULL) {  
        cout<< root->data; //访问根结点  
        preorder(root->Lchild); //先序遍历根的左子树  
        preorder(root->Rchild); //先序遍历根的右子树  
    }//if  
}//preorder
```



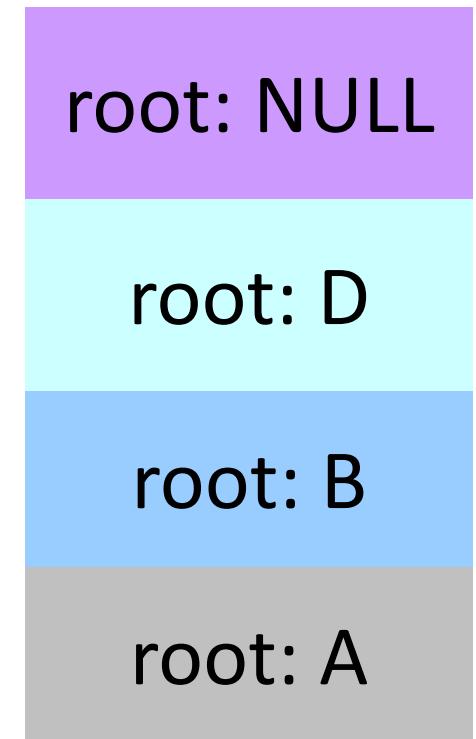
先序遍历序列: A B D



```
void preorder (BiTNode *root) {  
    if (root!=NULL) {  
        cout<< root->data; //访问根结点  
        preorder(root->Lchild); //先序遍历根的左子树  
        preorder(root->Rchild); //先序遍历根的右子树  
    }//if  
}//preorder
```



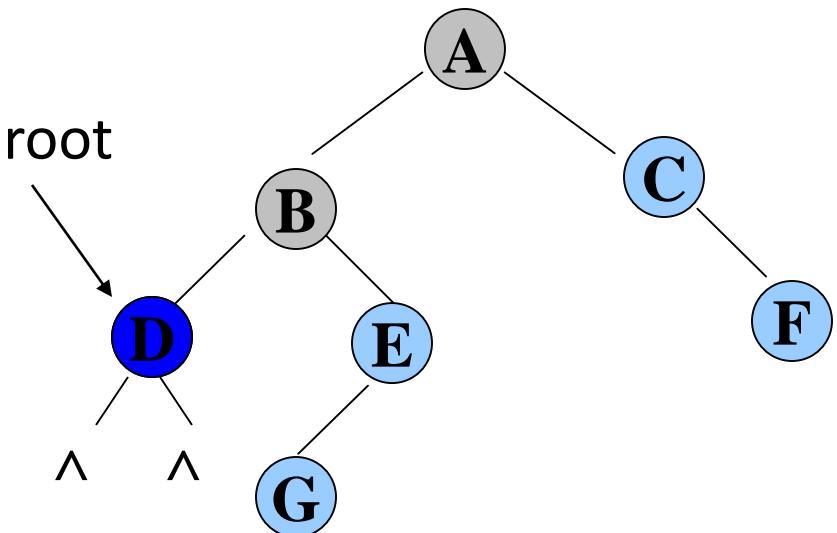
先序遍历序列: A B D



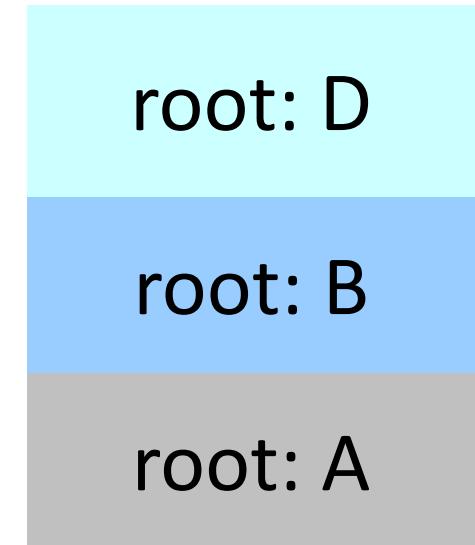
```

void preorder (BiTNode *root) {
    if (root!=NULL) {
        cout<< root->data; //访问根结点
        preorder(root->Lchild); //先序遍历根的左子树
        preorder(root->Rchild); //先序遍历根的右子树
    }//if
}//preorder

```



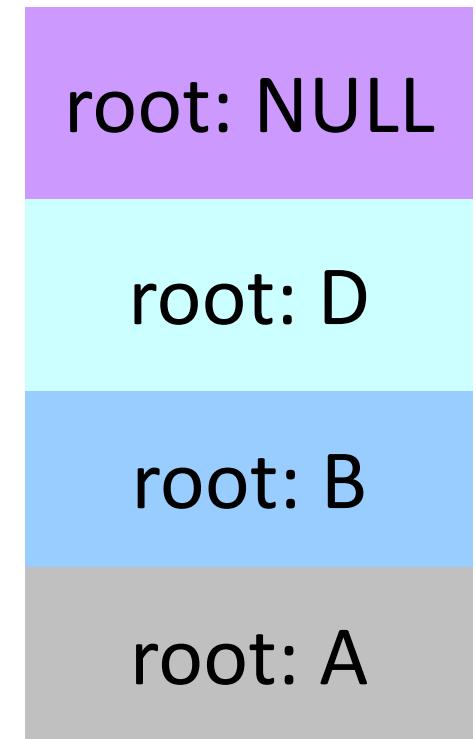
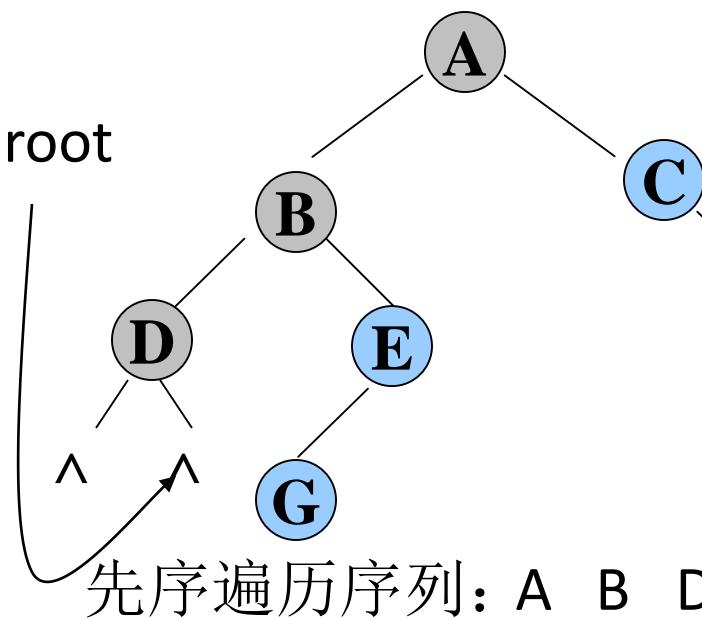
先序遍历序列: A B D



```

void preorder (BiTNode *root) {
    if (root!=NULL) {
        cout<< root->data; //访问根结点
        preorder(root->Lchild); //先序遍历根的左子树
        preorder(root->Rchild); //先序遍历根的右子树
    }//if
}//preorder

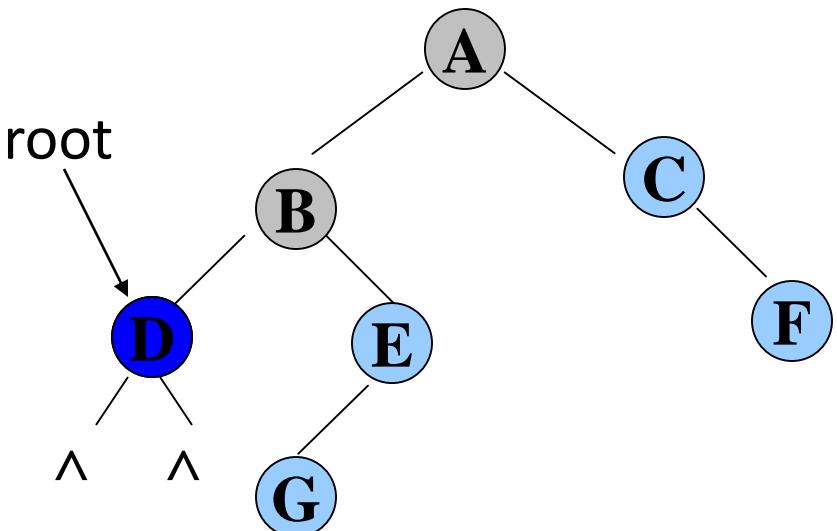
```



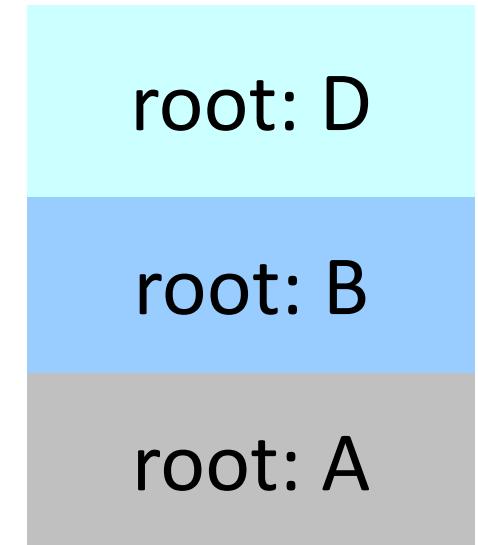
```

void preorder (BiTNode *root) {
    if (root!=NULL) {
        cout<< root->data; //访问根结点
        preorder(root->Lchild); //先序遍历根的左子树
        preorder(root->Rchild); //先序遍历根的右子树
    }//if
}//preorder

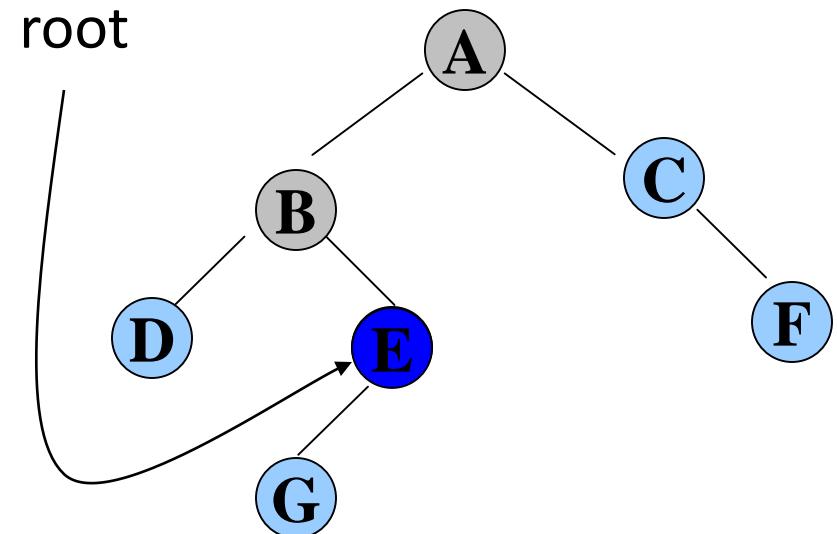
```



先序遍历序列: A B D



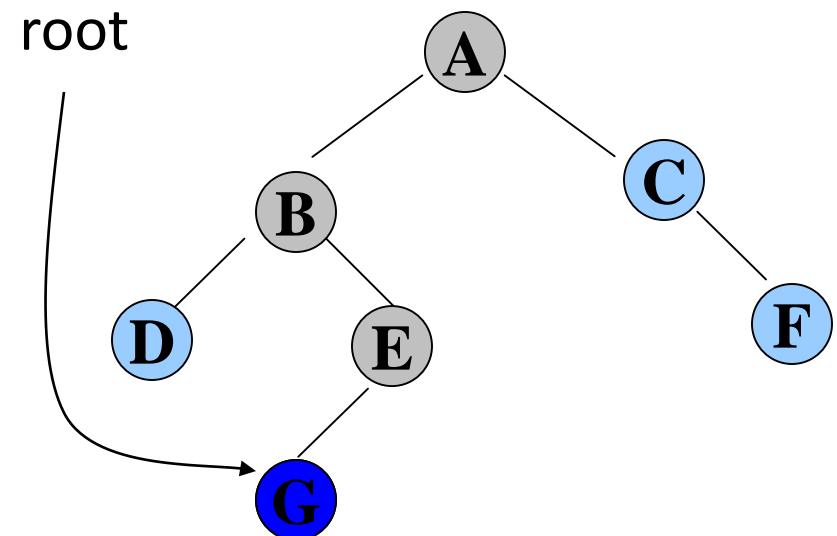
```
void preorder (BiTNode *root) {  
    if (root!=NULL) {  
        cout<< root->data; //访问根结点  
        preorder(root->Lchild); //先序遍历根的左子树  
        preorder(root->Rchild); //先序遍历根的右子树  
    }//if  
}//preorder
```



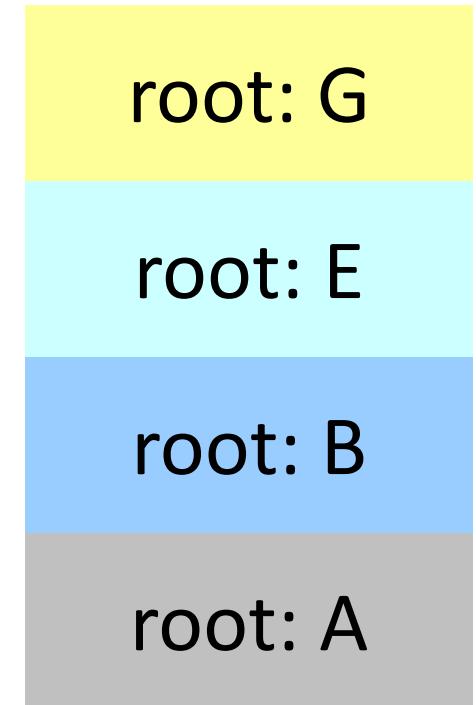
先序遍历序列: A B D E

root: E
root: B
root: A

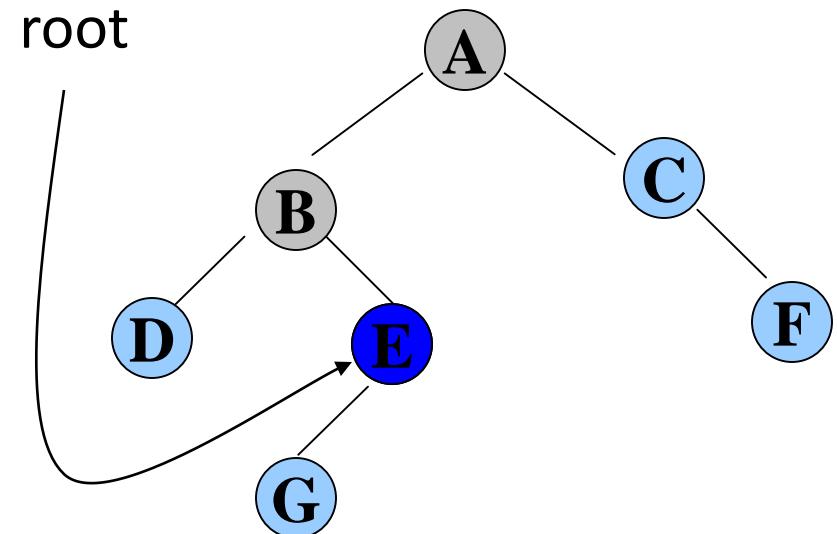
```
void preorder (BiTNode *root) {  
    if (root!=NULL) {  
        cout<< root->data; //访问根结点  
        preorder(root->Lchild); //先序遍历根的左子树  
        preorder(root->Rchild); //先序遍历根的右子树  
    }//if  
}//preorder
```



先序遍历序列: A B D E G



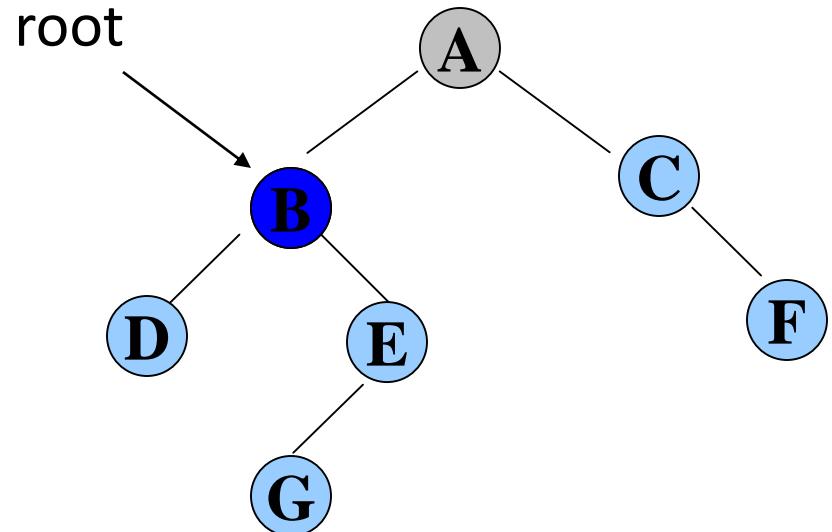
```
void preorder (BiTNode *root) {  
    if (root!=NULL) {  
        cout<< root->data; //访问根结点  
        preorder(root->Lchild); //先序遍历根的左子树  
        preorder(root->Rchild); //先序遍历根的右子树  
    }//if  
}//preorder
```



先序遍历序列: A B D E G

root: E
root: B
root: A

```
void preorder (BiTNode *root) {  
    if (root!=NULL) {  
        cout<< root->data; //访问根结点  
        preorder(root->Lchild); //先序遍历根的左子树  
        preorder(root->Rchild); //先序遍历根的右子树  
    }//if  
}//preorder
```

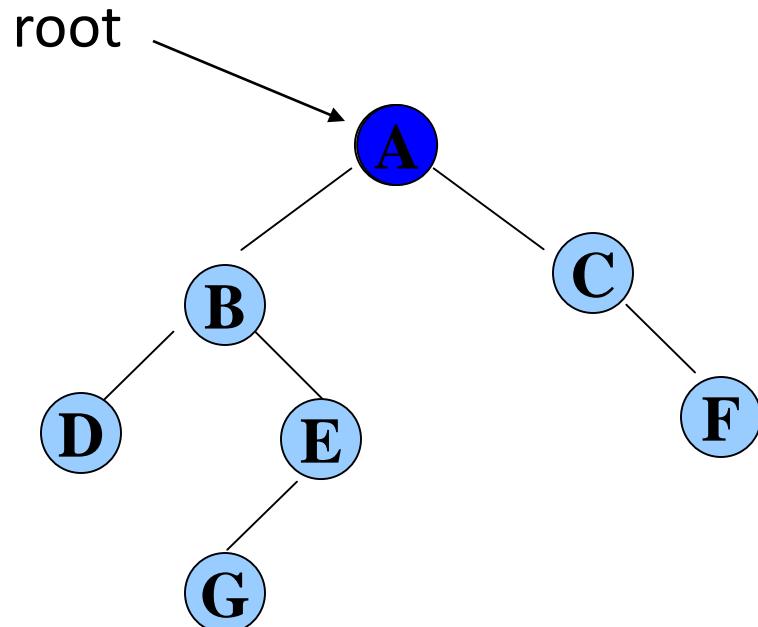


先序遍历序列: A B D E G

root: B

root: A

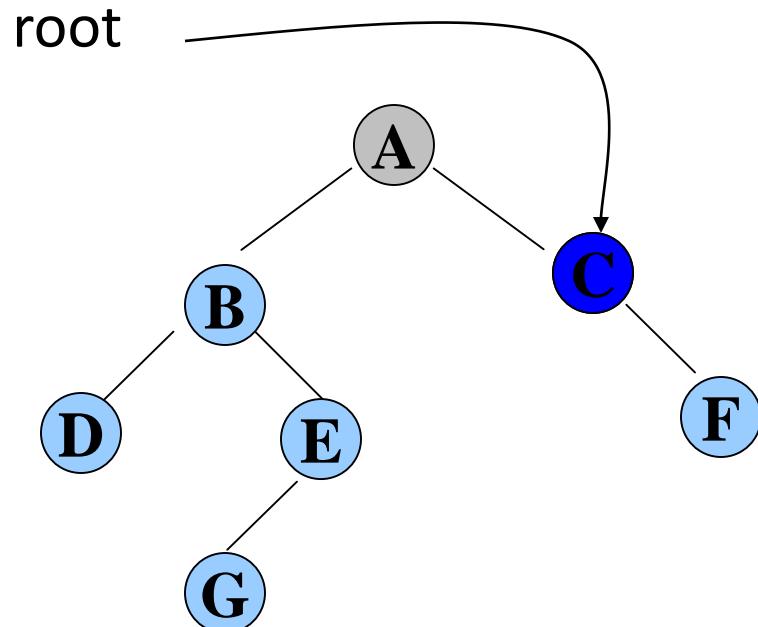
```
void preorder (BiTNode *root) {
    if (root!=NULL) {
        cout<< root->data; //访问根结点
        preorder(root->Lchild); //先序遍历根的左子树
        preorder(root->Rchild); //先序遍历根的右子树
    }//if
}//preorder
```



先序遍历序列: A B D E G

root: A

```
void preorder (BiTNode *root) {  
    if (root!=NULL) {  
        cout<< root->data; //访问根结点  
        preorder(root->Lchild); //先序遍历根的左子树  
        preorder(root->Rchild); //先序遍历根的右子树  
    }//if  
}//preorder
```

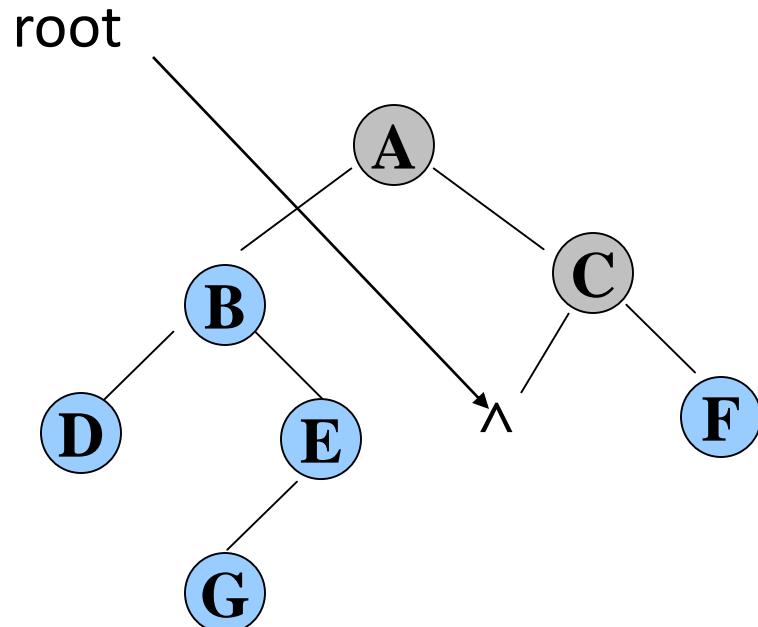


先序遍历序列: A B D E G C

root: C

root: A

```
void preorder (BiTNode *root) {  
    if (root!=NULL) {  
        cout<< root->data; //访问根结点  
        preorder(root->Lchild); //先序遍历根的左子树  
        preorder(root->Rchild); //先序遍历根的右子树  
    }//if  
}//preorder
```



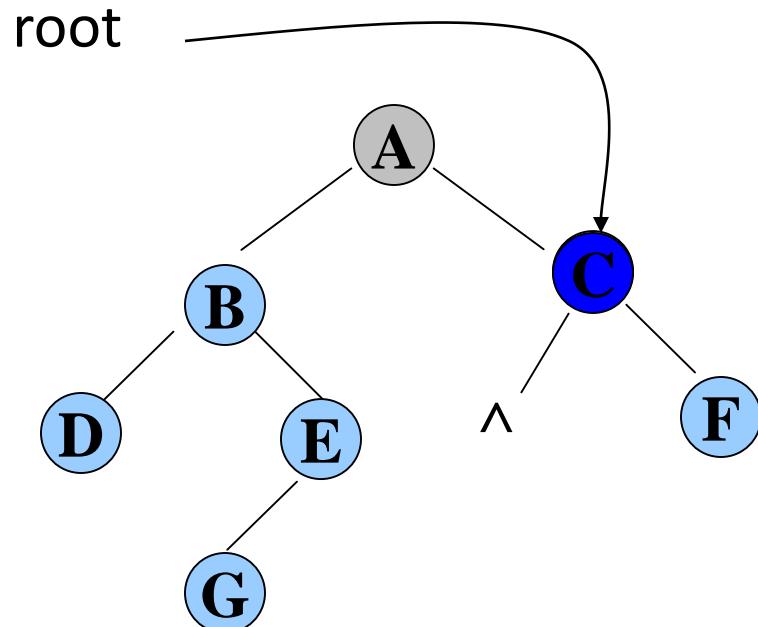
先序遍历序列: A B D E G C

root: NULL

root: C

root: A

```
void preorder (BiTNode *root) {  
    if (root!=NULL) {  
        cout<< root->data; //访问根结点  
        preorder(root->Lchild); //先序遍历根的左子树  
        preorder(root->Rchild); //先序遍历根的右子树  
    }//if  
}//preorder
```

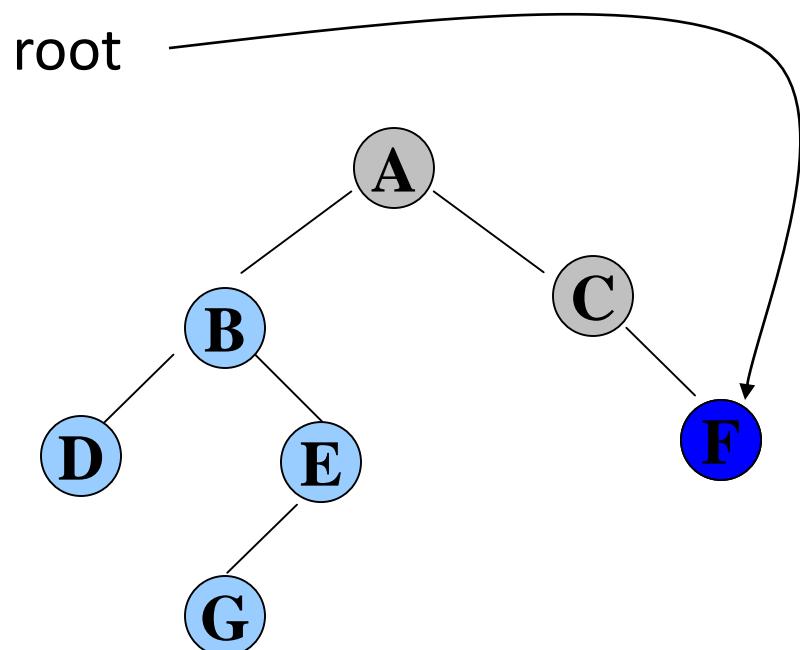


先序遍历序列: A B D E G C

root: C

root: A

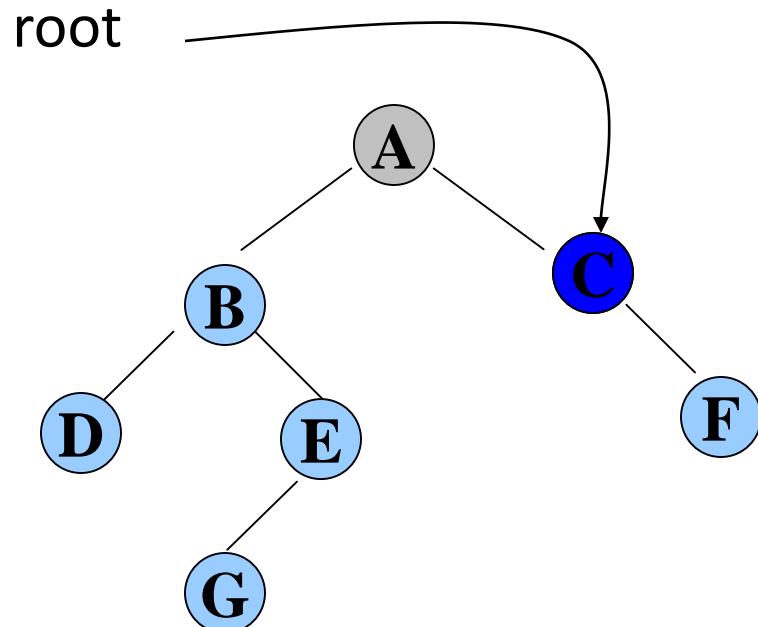
```
void preorder (BiTNode *root) {  
    if (root!=NULL) {  
        cout<< root->data; //访问根结点  
        preorder(root->Lchild); //先序遍历根的左子树  
        preorder(root->Rchild); //先序遍历根的右子树  
    }//if  
}//preorder
```



先序遍历序列: A B D E G C F

root: F
root: C
root: A

```
void preorder (BiTNode *root) {  
    if (root!=NULL) {  
        cout<< root->data; //访问根结点  
        preorder(root->Lchild); //先序遍历根的左子树  
        preorder(root->Rchild); //先序遍历根的右子树  
    }//if  
}//preorder
```

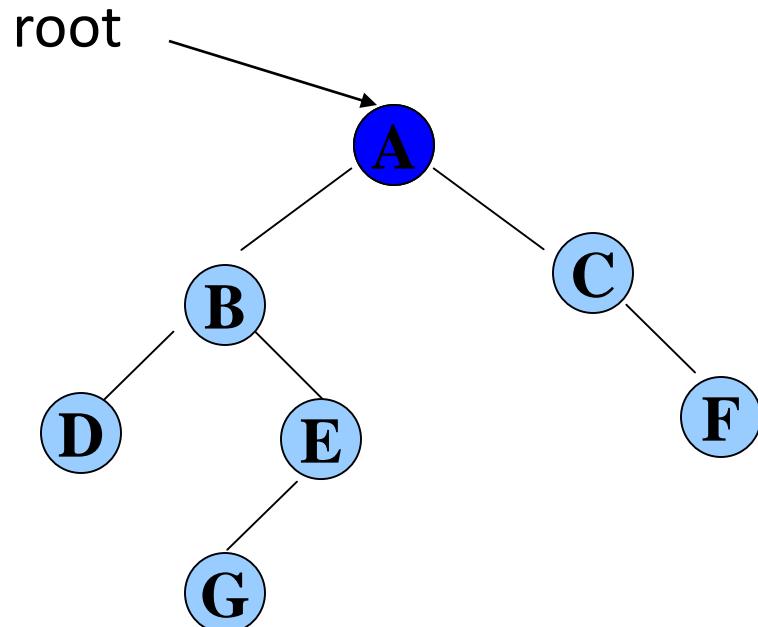


先序遍历序列: A B D E G C F

root: C

root: A

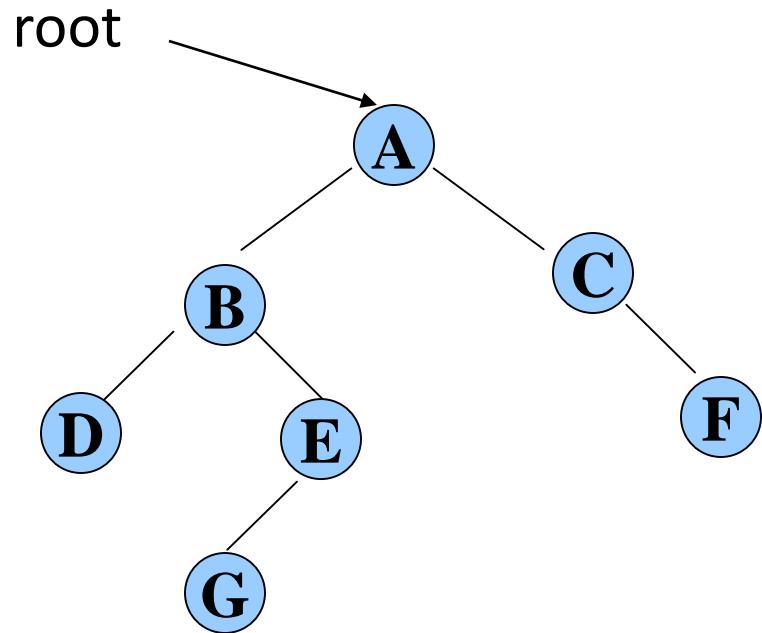
```
void preorder (BiTNode *root) {
    if (root!=NULL) {
        cout<< root->data; //访问根结点
        preorder(root->Lchild); //先序遍历根的左子树
        preorder(root->Rchild); //先序遍历根的右子树
    }//if
}//preorder
```



先序遍历序列: A B D E G C F

root: A

```
void preorder (BiTNode *root) {
    if (root!=NULL) {
        cout<< root->data; //访问根结点
        preorder(root->Lchild); //先序遍历根的左子树
        preorder(root->Rchild); //先序遍历根的右子树
    }//if
}//preorder
```

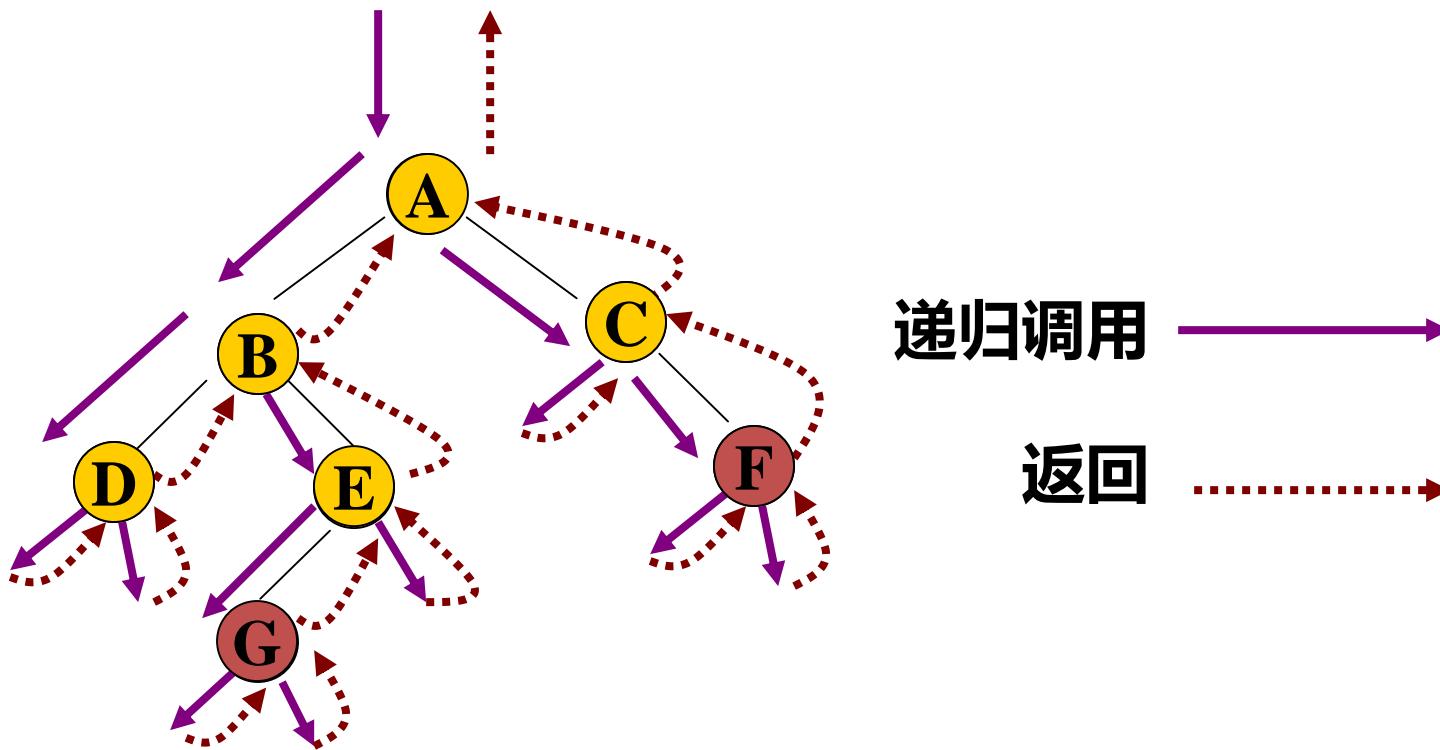


先序遍历序列: A B D E G C F

先序遍历过程

```
void preorder (BiTNode *root) {  
    if (root!=NULL) {  
        cout<< root->data; //访问根结点  
        preorder(root->Lchild); //先序遍历根的左子树  
        preorder(root->Rchild); //先序遍历根的右子树  
    } //if  
} //preorder
```

先序遍历序列: A B D E G C F



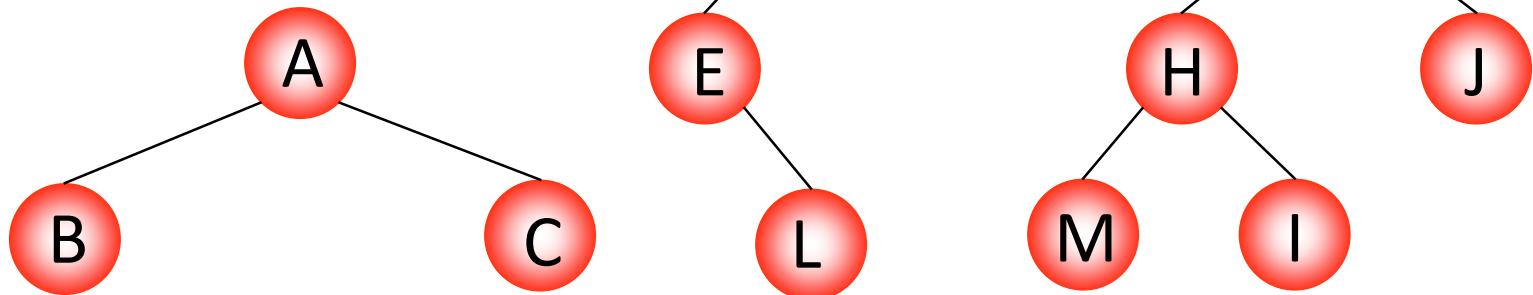
- 中序遍历二叉树的操作定义：

若二叉树为空，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。



中序遍历的顺序为： BAC

中序遍历的顺序为：
ELBAMHIDJ

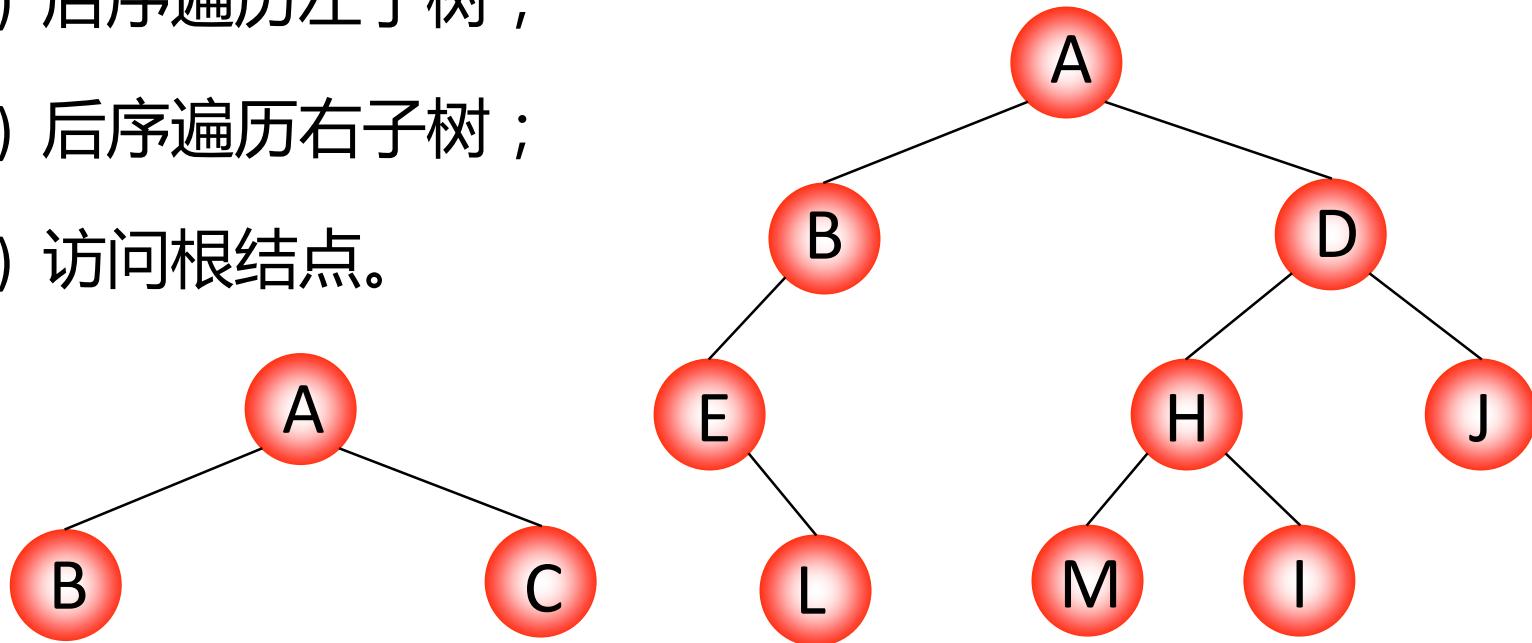
- **后序**遍历二叉树的操作定义：

若二叉树为空，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。

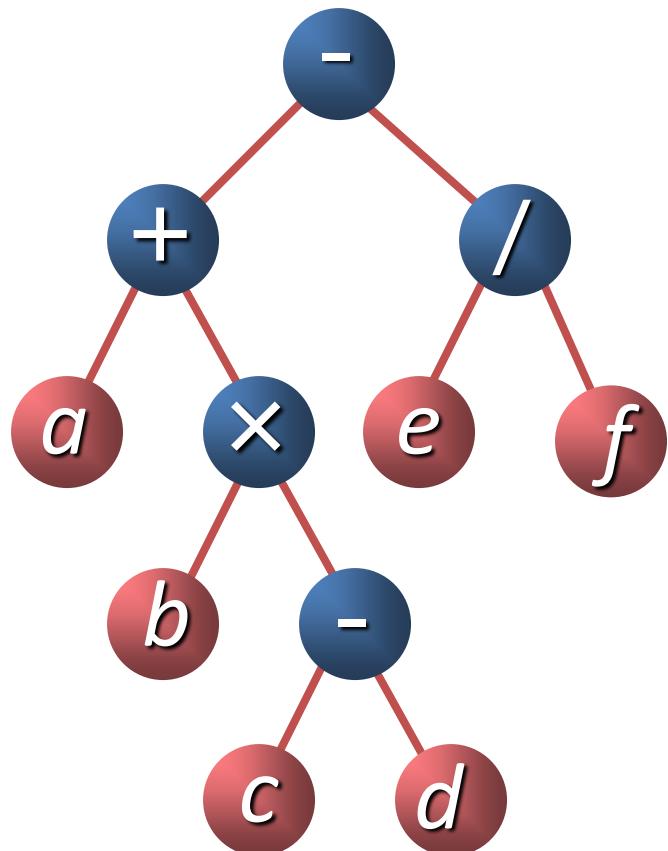


后序遍历的顺序为： BCA

后序遍历的顺序为：
LEBМИHJDA

$$a + b \times (c - d) - e / f$$

例：请写出下图所示二叉树的先序、中序和后序遍历顺序。



遍历结果：

先序： - + a × b - c d / e f

表达式的前缀表示（波兰式）

中序： a + b × c - d - e / f

表达式的中缀表示

后序： a b c d - × + e f / -

表达式的后缀表示（逆波兰式）

● 先序遍历二叉树基本操作的递归算法在二叉链表上的实现：

```
Status PreOrderTraverse (Bitree T, Visit)
```

```
{ // 最简单的 Visit 函数是 :
```

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

```
// {   Printf (e);      // 实用时加上格式串。
```

```
//     return OK; }
```

```
// 调用实例 : PreOrderTraverse (T, PrintElement);
```

```
if (T)
```

```
{ Visit (T->data);
```

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

```
    PreOrderTraverse (T->rchild, Visit);
```

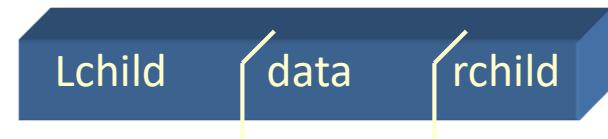
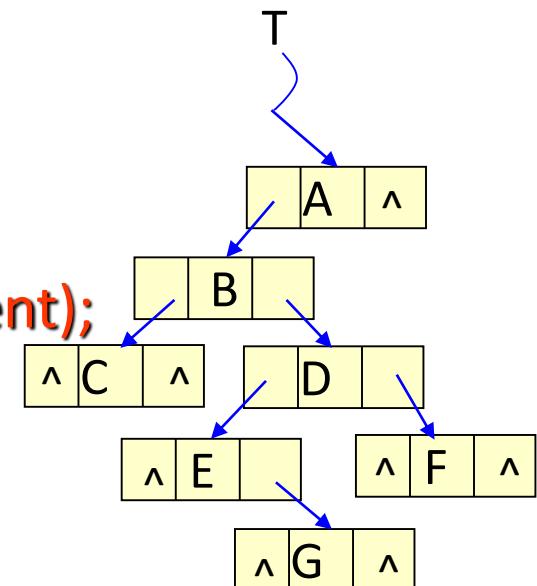
```
        return OK;
```

```
//return ERROR;
```

```
}
```

```
else return OK;
```

```
} // PreOrderTraverse
```

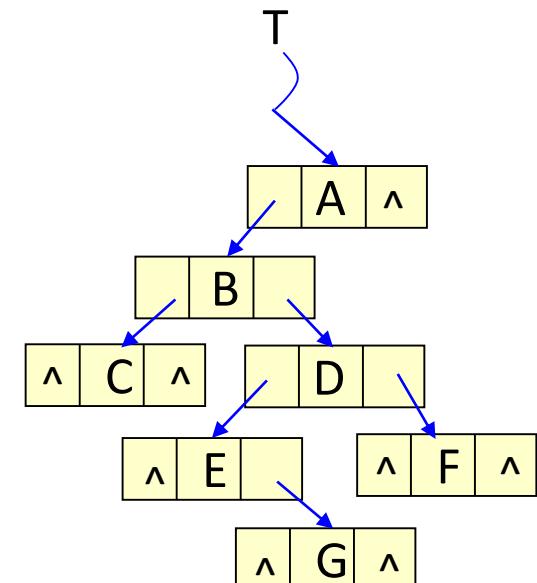


结点结构

● 中序遍历二叉树基本操作的递归算法在二叉链表上的实现：

Status InOrderTraverse (Bitree T, Visit)

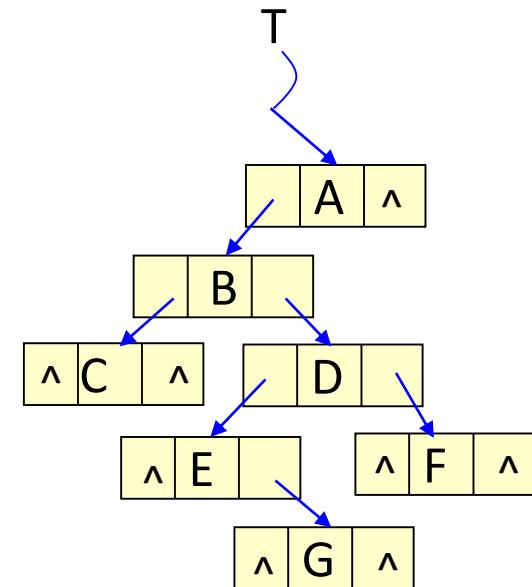
```
{ if (T)
  { if (InOrderTraverse ( T->lchild, Visit ))
    if (Visit (T->data))
      if (InOrderTraverse (T->rchild, Visit))
        return OK;
    return ERROR;
  }
else
  return OK;
} // InOrderTraverse
```



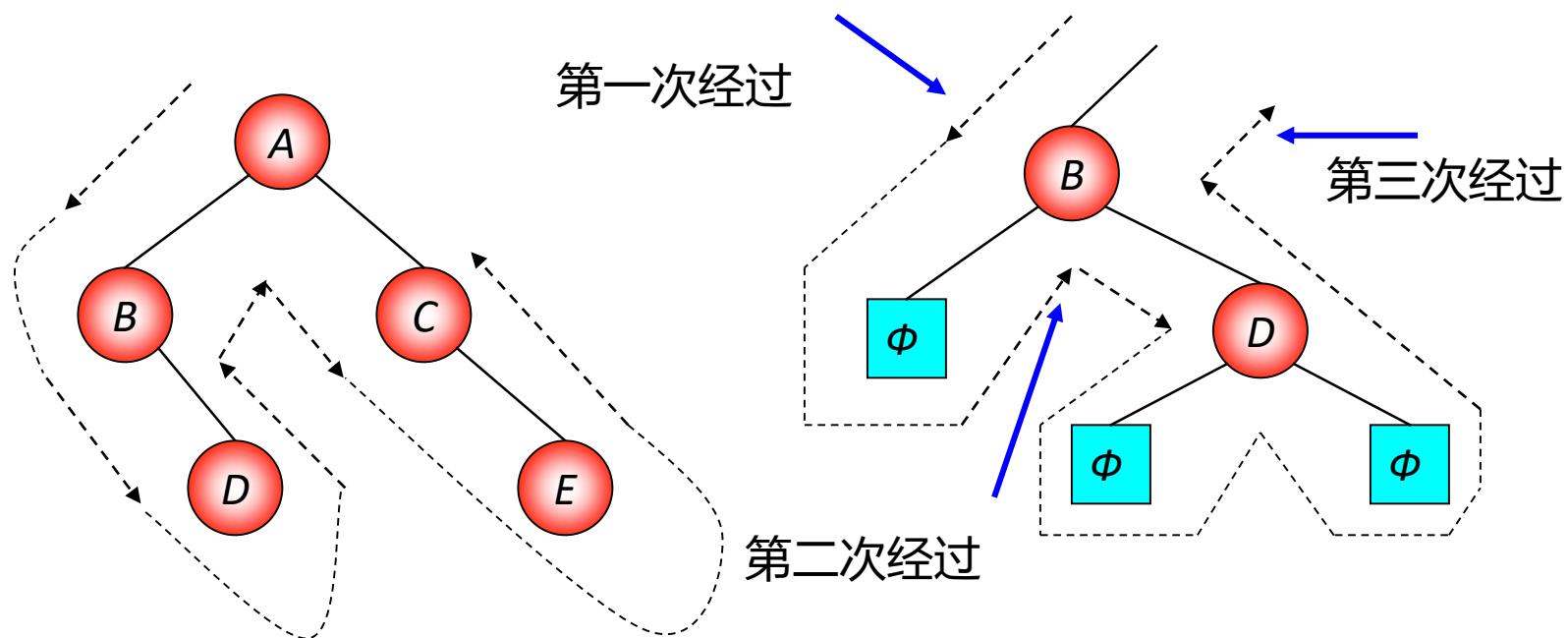
● 后序遍历二叉树基本操作的递归算法在二叉链表上的实现：

Status PostOrderTraverse (Bitree T, Visit)

```
{ if (T)
    { if (PostOrderTraverse ( T->lchild, Visit ))
        if (PostOrderTraverse (T->rchild, Visit))
            if (Visit (T->data))
                return OK;
            return ERROR;
    }
else
    return OK;
} // PostOrderTraverse
```



以中序遍历为例来说明中序遍历二叉树的递归过程

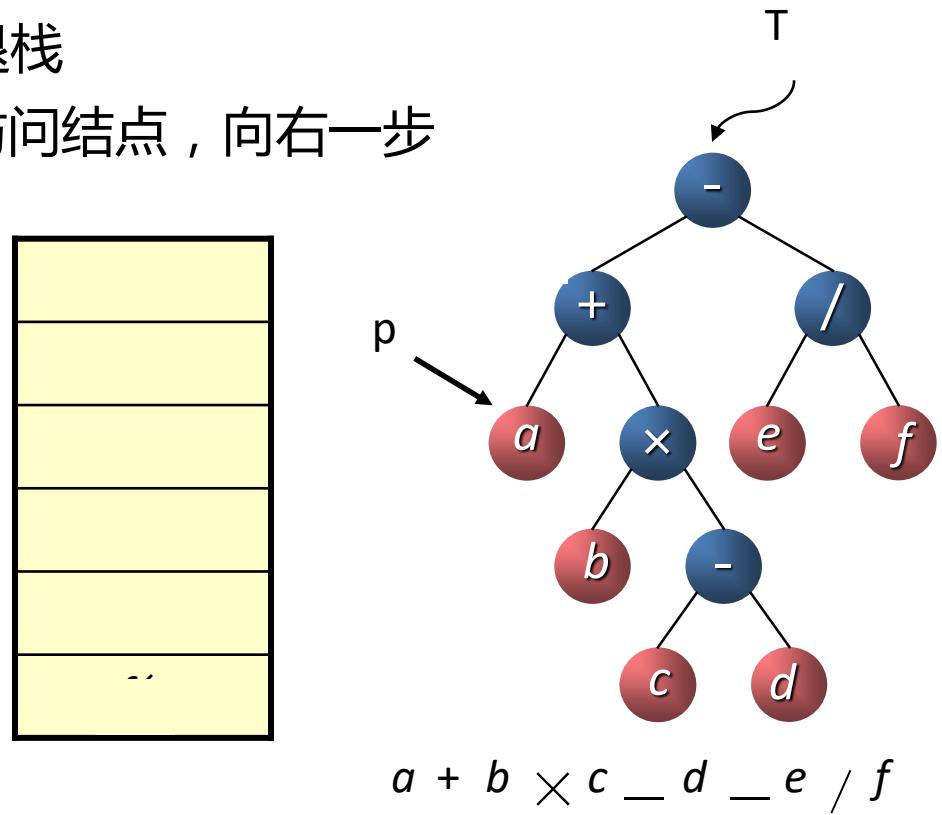


课堂练习

1. 二叉树的前序遍历序列中，任意一个结点均处在其子女结点的前面，这种说法（ ）
 (A) 正确 (B) 错误
2. 由于二叉树中每个结点的度最大为 2，所以二叉树是一种特殊的树，这种说法（ ）
 (A) 正确 (B) 错误
3. 已知某二叉树的后序遍历序列是 dabec。中序遍历序列是 debac，它的前序遍历序列是（ ）。
 (A) acbed (B) decab (C) deabc (D) cedba
4. 某二叉树的前序遍历结点访问顺序是 abdgcefhi，中序遍历的结点访问顺序是 dgbaechf，则其后序遍历的结点访问顺序是（ ）。
 (A) bdgcefha (B) gdbecfha (C) bdgaechf (D) gdbehfca
5. 在一非空二叉树的中序遍历序列中，根右边（ ）
 (A) 只有右子树上的所有结点 (B) 只有右子树上的部分结点
 (C) 只有左子树上的部分结点 (D) 只有左子树上的所有结点
6. 任一二叉树的叶子结点在先、中和后序遍历序列中的相对次序（ ）。
 (A) 不发生改变 (B) 发生改变 (C) 不能确定 (D) 以上都不对

● 中序遍历二叉树基本操作非递归算法在二叉链表上的实现：

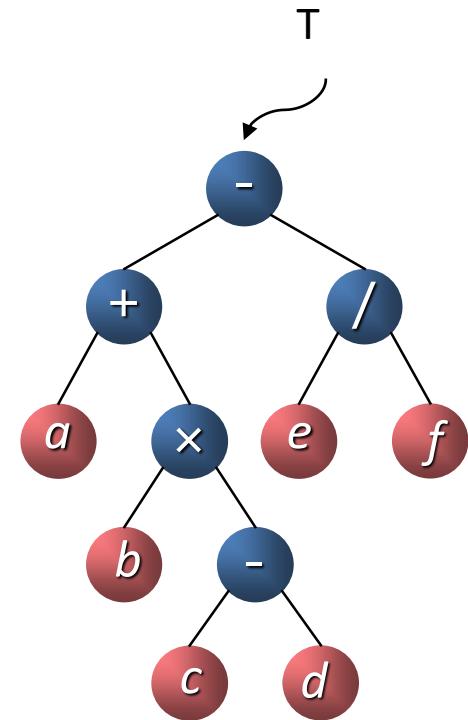
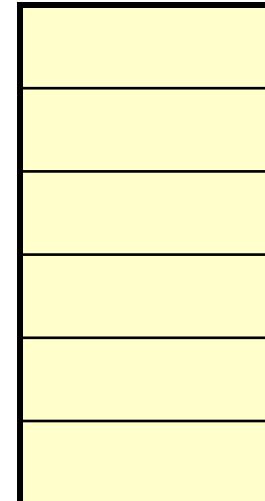
```
Status InOrderTraverse (BiTree T, Status(* Visit)(TElemType e))  
{ InitStack(S); Push(S, T); // 根指针进栈  
    while (!StackEmpty (S) {  
        while (GetTop (S, p) && p) Push (S, p ->lchild); // 向左走到尽头。  
        Pop (S, p); // 空指针退栈  
        if (! StackEmpty (S) { // 访问结点，向右一步  
            Pop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



● 中序遍历二叉树基本操作 非递归 算法在二叉链表上的实现：

```
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);  
            if ( !Visit(p ->data))  
                return ERROR;  
            p = p ->rchild;  
        } // else  
    } // while  
    return OK;  
} // InOrderTraverse
```



二叉树其它操作算法举例

➤ 统计二叉树中叶子结点的个数

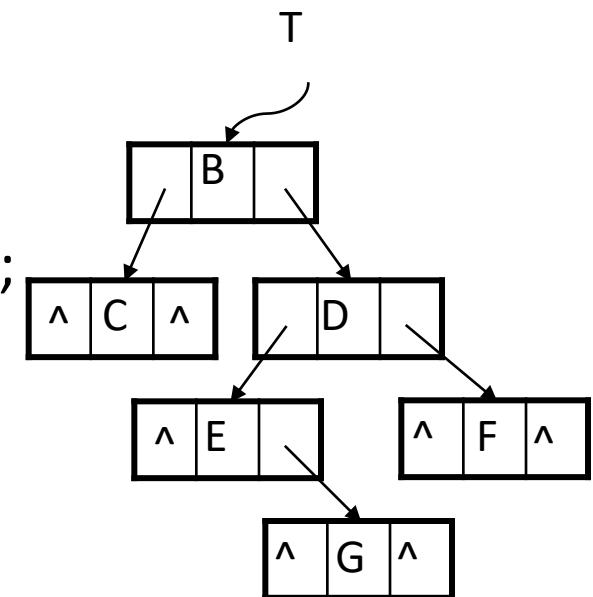
实现此操作只需对二叉树“遍历”一遍，并在遍历过程中对“叶子结点计数”即可。显然这个遍历的次序可以随意，只是为了在遍历时进行计数，需要在算法的参数中设一个“计数器”。

```
void CountLeaf (BiTree T, int &count)
{ // 先序遍历二叉树以 count 返回二叉树中叶子结点的数目
    if (T) {
        if ((!T->Lchild) && (!T->Rchild)) // 无左、右子树
            count++; // 对叶子结点计数
        CountLeaf (T->Lchild, count);
        CountLeaf (T->Rchild, count);
    } // if
} // CountLeaf
```

➤ 求二叉树的深度（后序）

二叉树的深度 = MAX (左子树深度 , 右子树深度) + 1 。

```
void BiTreeDepth(BiTree T)
{ if (!T)
    depth = 0;
else {
    depthleft = BiTreeDepth(T->Lchild);
    depthright = BiTreeDepth(T->Rchild);
    depth = max(depthleft, depthright) + 1;
}
return depth;
}// BiTreeDepth
```



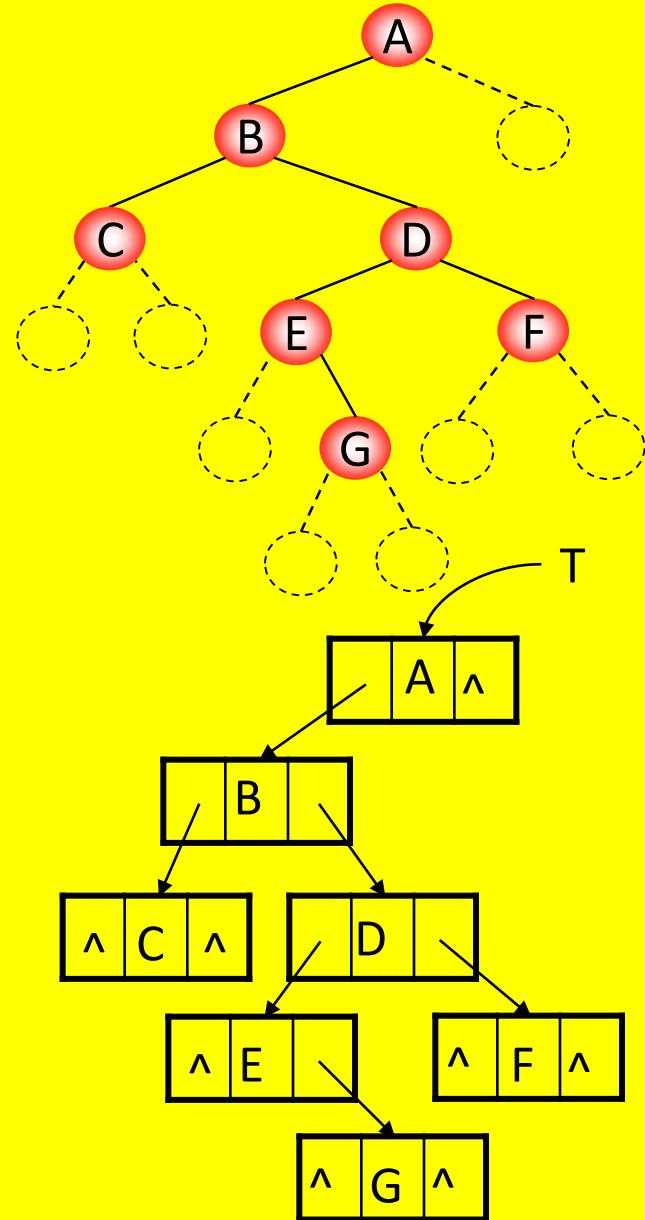
➤ 建立二叉树的存储结构——二叉链表（先序）

ABCDEGF

对右图所示二叉树，按下列顺序读入字符：

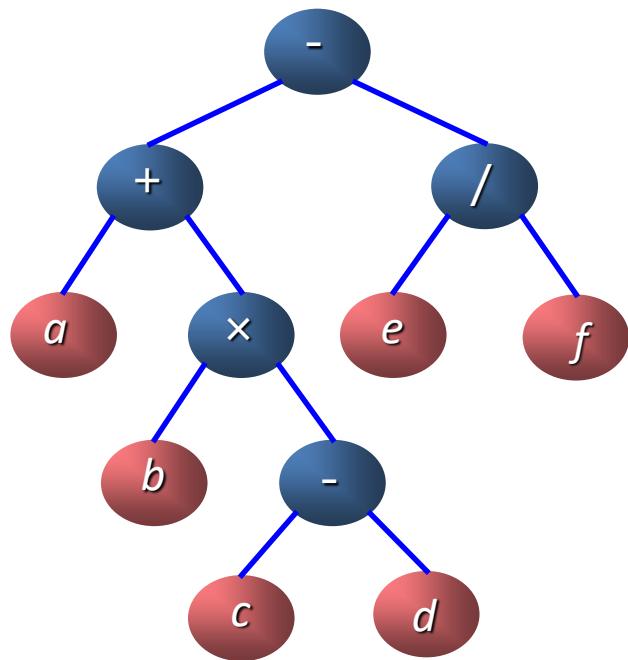
AB CΦΦΦDEΦGΦΦΦFΦΦΦΦ

```
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
```



线索二叉树

问题 1：为什么要研究线索二叉树？



问题 2：如何保存结果以免重复遍历？

办法： 1、另辟空间存放遍历结果。

需要付出额外的存储花销。

遍历结果：

先序： - + a × b - c d / e f

中序：**a + b × c - d - e / f**

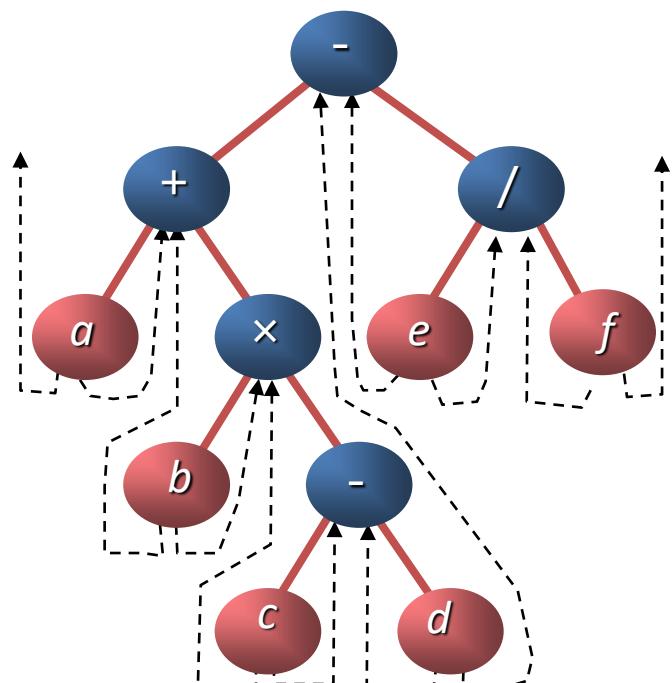
后序： a b c d - × + e f / -

非线性结构

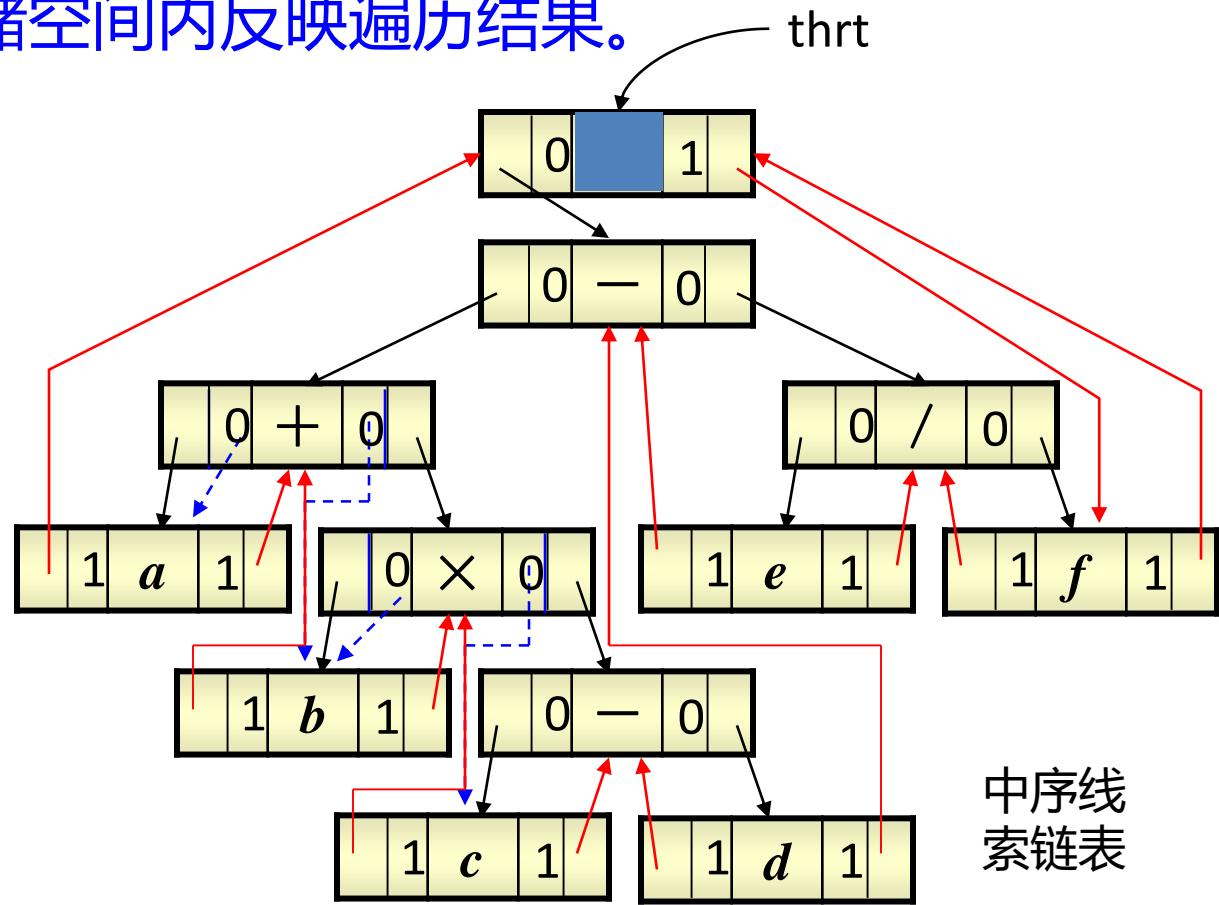
转
化

线性结构

3、在原二叉链表的存储空间内反映遍历结果。



中序线索二叉树



中序线索链表

在线索树上进行遍历的方法：

- 1、从序列中的第一个结点起，依次找后继，直至后继为空。
- 2、从序列中的最后一个结点起，依次找前驱，直至前驱为空。

线索二叉树的存储表示

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

// Link == 0 : 指针 , Thread == 1 : 线索

```
typedef struct BiThrNode {
```

 TElemType data;

```
    struct BiThrNode *lchild, *rchild; // 左右指针
```

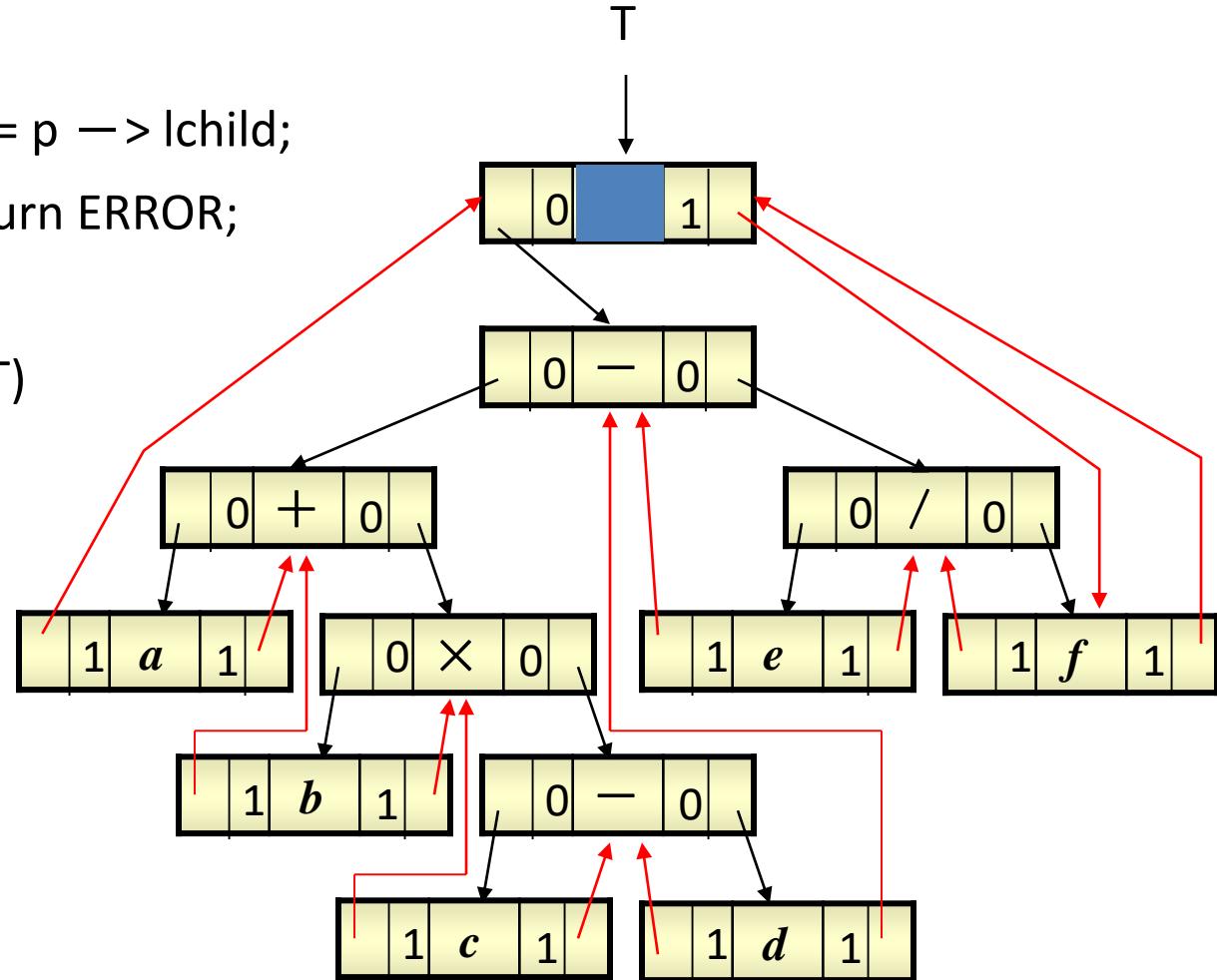
 PointerTag LTag, RTag; // 左右标志

```
} BiThrNode, *BiThrTree;
```

线索链表的遍历算法（中序找后继法）：

```
Status InOrderTraverse_Thr(BiThrTree T, Visit)
```

```
{ p = T -> lchild;  
while (p != T)  
{ while (p -> LTag == 0) p = p -> lchild;  
    if (!Visit(p -> data)) return ERROR;  
    while (p -> RTag == 1  
          && p -> rchild != T)  
    { p = p -> rchild;  
        Visit(p -> data);  
    }  
    p = p -> rchild;  
}  
return OK;  
} // InOrderTraverse_Thr
```

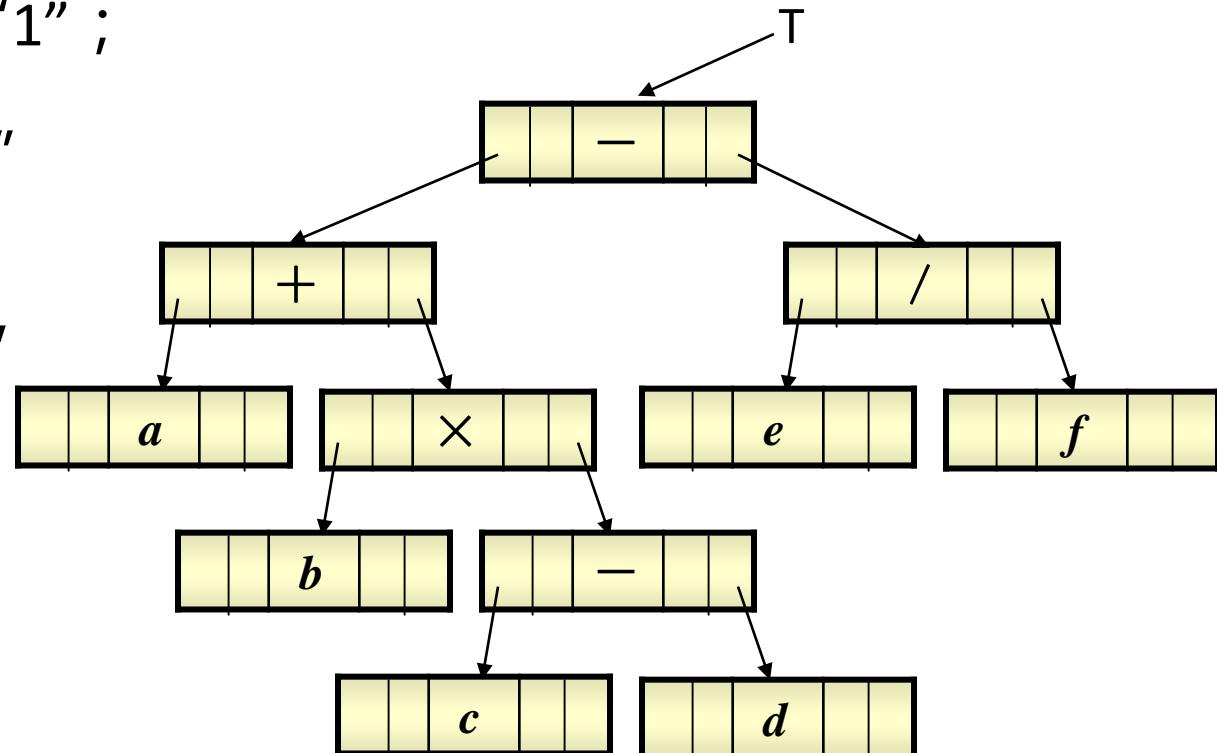


● 按中序建立线索链表（按中序线索化二叉树）

1、若结点没有左子树，则令其左指针指向它的“前驱”并将左指针类型标志改为“1”；

2、若结点没有右子树，则令其右指针指向它的“后继”并将右指针类型标志改为“1”；

3、为了获取“前驱”的信息，需要在遍历过程中添加一个指针 pre ，令其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)
```

```
    { InThreading(p -> lchild); // 左子树线索化
```

```
        if (!p -> lchild)
```

```
        { p -> LTag = 1;
```

```
            p -> lchild = pre; }
```

```
        if (!pre -> rchild)
```

```
        { pre -> RTag = 1;
```

```
            pre -> rchild = p; }
```

```
        pre = p;
```

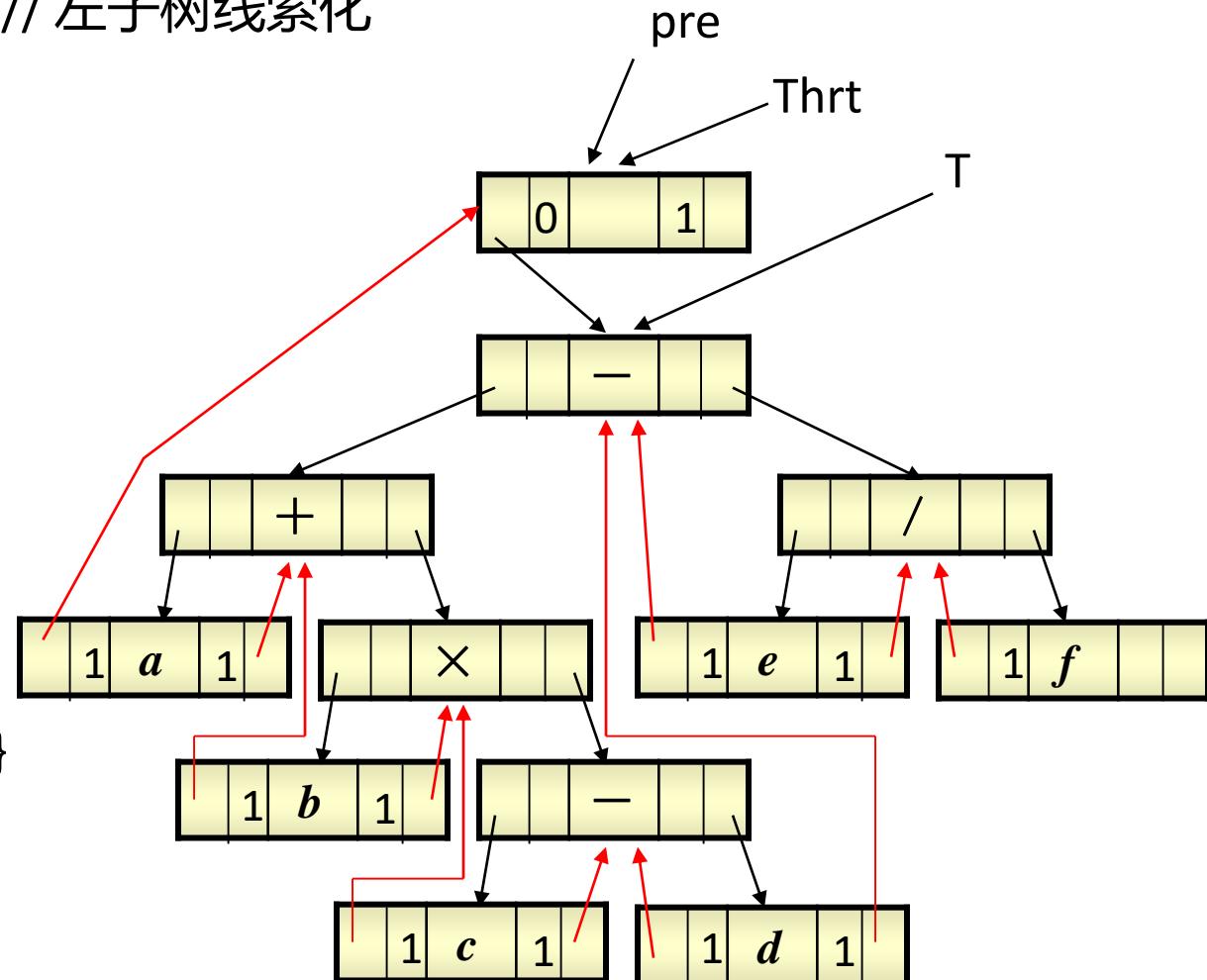
```
// 保持 pre 指向 p 的前驱
```

```
        InThreading(p -> rchild); }
```

```
        // 右子树线索化
```

```
}
```

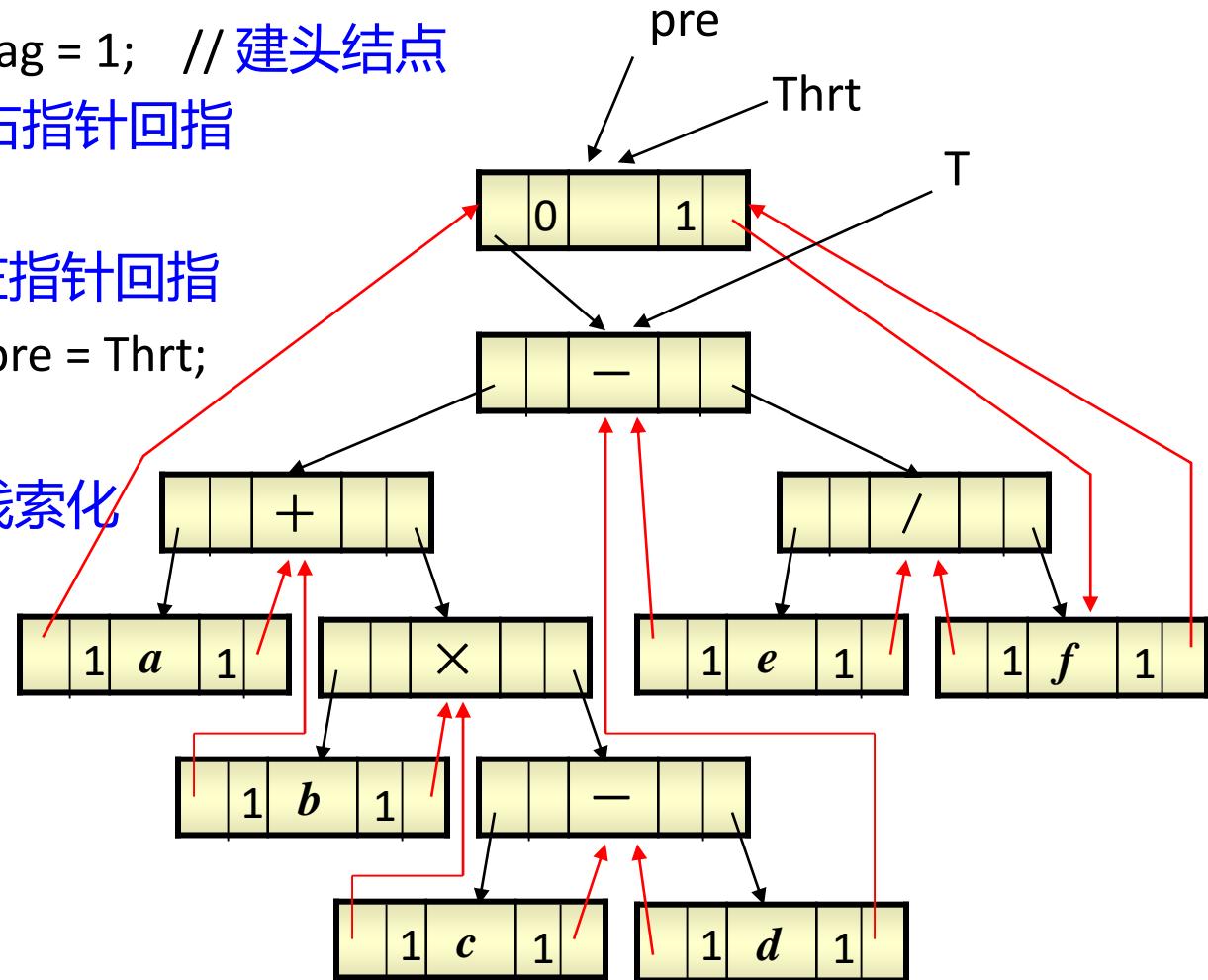
```
} // InThreading
```



```

Status InOrderThreading(BiThrTree &Thrt, BiThrTree T) {
    if (!(Thrt = (BiThrTree)malloc(sizeof(BiThrNode))))
        exit (OVERFLOW);
    Thrt -> LTag = 0; Thrt -> RTag = 1; // 建头结点
    Thrt -> rchild = Thrt; // 右指针回指
    if (!T) Thrt -> lchild = Thrt;
        // 若二叉树空，则左指针回指
    else { Thrt -> lchild = T; pre = Thrt;
        InThreading(T);
            // 中序遍历进行中序线索化
        pre -> rchild = Thrt;
        pre -> RTag = 1;
        // 最后一个结点线索化
        Thrt -> rchild = pre; }
    return OK;
} // InOrderThreading

```



课堂练习

选择题

1. 在线索化二叉树中， t 所指结点没有左子树的充要条件是（）
(A) $t \rightarrow lchild == NULL$ (B) $t \rightarrow ltag == 1$ (C) $t \rightarrow ltag == 1$ 且 $t \rightarrow lchild == NULL$
(D) 以上都不对

2. 二叉树按某种顺序线索化后，任一结点均有指向其前趋和后继的线索，这种说法（）
(A) 正确 (B) 错误

树和二叉树的相关概念、术语

二叉树的五个性质

二叉树存储、遍历、线索化

树和森林

哈夫曼树



树和森林

6.4.1 树的存储结构

1 双亲表示法

实现：定义结构数组

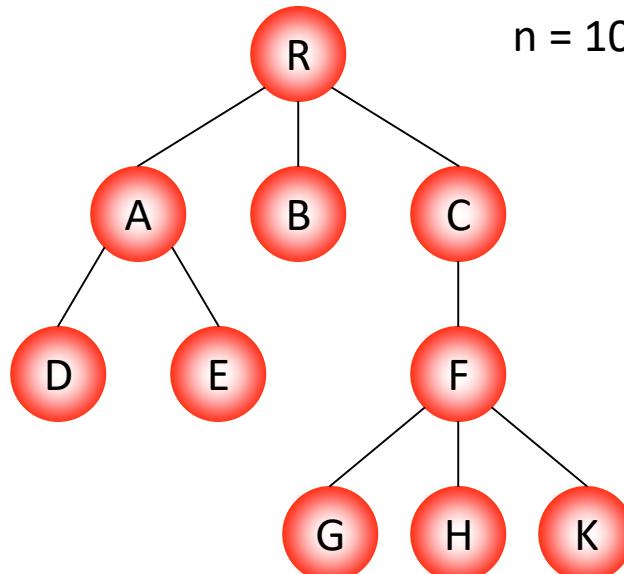
存放树的结点，

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的位置。



数组下标

$r = 0$
 $n = 10$

data parent

0	R	-1
1	A	0
2	B	0
3	C	0
4	D	1
5	E	1
6	F	3
7	G	6
8	H	6
9	K	6

C 语言的类型描述:

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

树结构:

```
#define MAX_TREE_SIZE 100  
  
typedef struct {  
    PTNode nodes[MAX_TREE_SIZE];  
    ...  
} PTree;
```

结点结构:

data	parent
------	--------

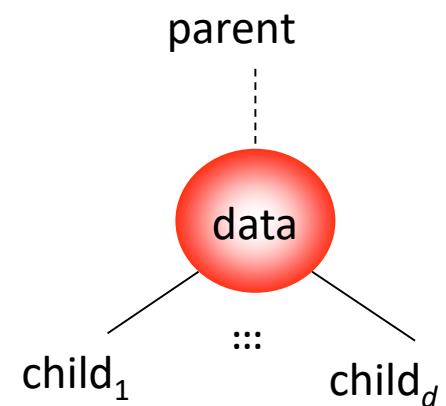
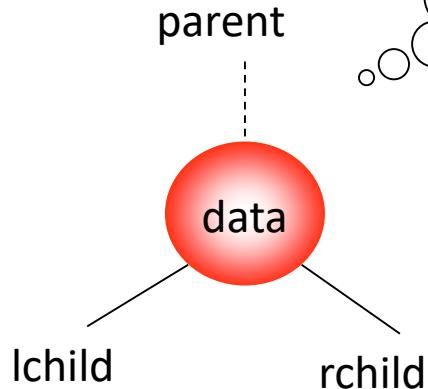
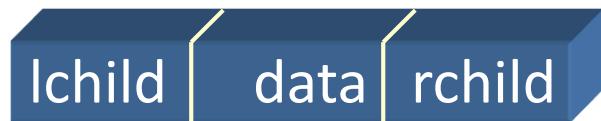
数组下标 data parent

r = 0	0	R	-1
n = 10	1	A	0
	2	B	0
	3	C	0
	4	D	1
	5	E	1
	6	F	3
	7	G	6

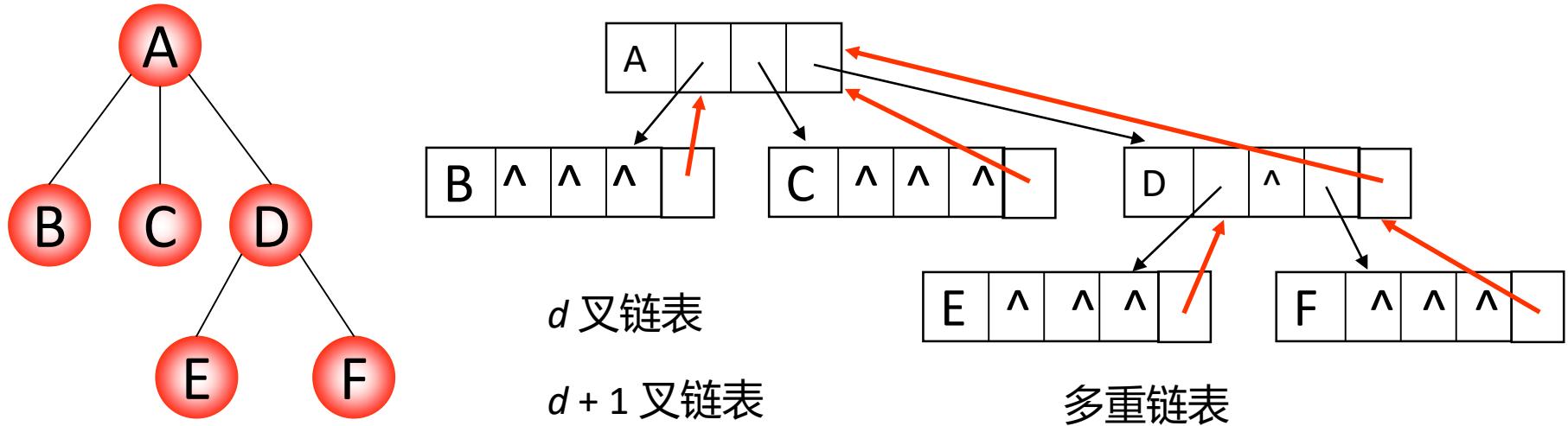
特点：找双亲容易，找孩子难。

2 孩子表示法（树的链式存储结构）

1) 多重链表

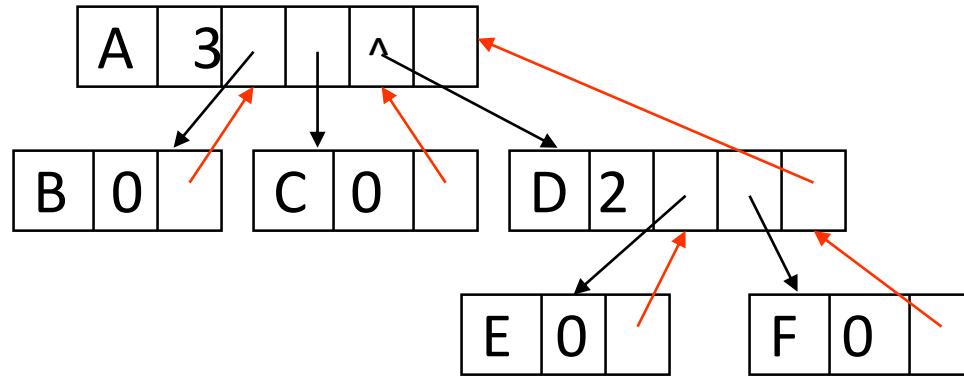
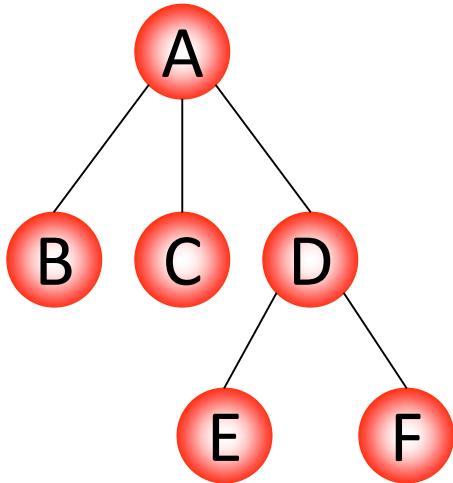
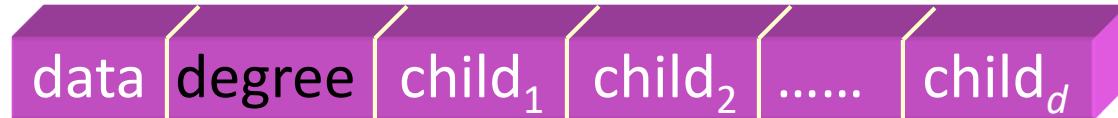


二叉树的
结点结构



在有 n 个结点、度为 d 的树的 d 叉链表中，有 $n \times (d-1) + 1$ 个空链域。

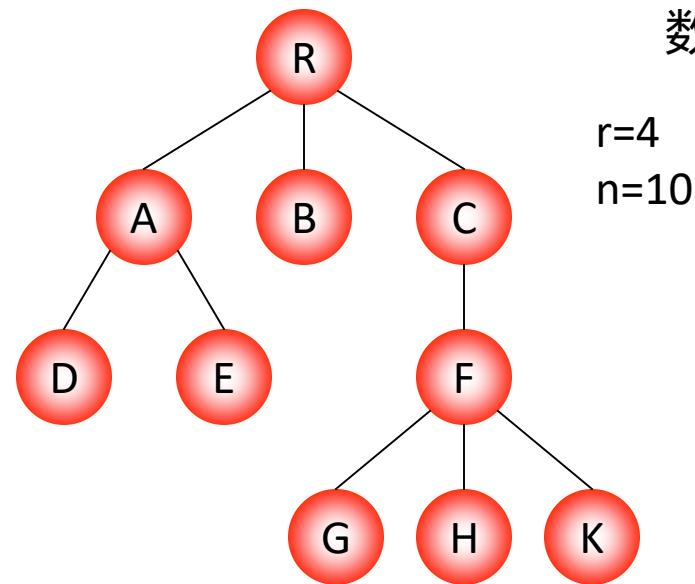
结点同构：
结点的指针个数相等，为**树的度 d** 。



节约存储空间
但操作不方便

结点不同构：
结点的指针个数不相等，
为该结点的度 degree。

2) 孩子链表：把每个结点的孩子结点排列起来，看成是一个线性表，用单链表存储，则 n 个结点有 n 个孩子链表（叶子的孩子链表为空表）。而 n 个头指针又组成一个线性表，用顺序表（含 n 个元素的结构数组）存储。



特点：找孩子容易，
找双亲难。

数组下标	data			firstchild						
0	4	A		<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>3</td><td></td></tr></table> → <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>5</td><td>^</td></tr></table>	3		5	^		
3										
5	^									
1	4	B	^							
2	4	C		<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>6</td><td>^</td></tr></table>	6	^				
6	^									
3	0	D	^							
4	-1	R		<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td></td></tr></table> → <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td><td></td></tr></table> → <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>2</td><td>^</td></tr></table>	0		1		2	^
0										
1										
2	^									
5	0	E	^							
6	2	F		<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>7</td><td></td></tr></table> → <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>8</td><td></td></tr></table> → <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>9</td><td>^</td></tr></table>	7		8		9	^
7										
8										
9	^									
7	6	G	^							
8	6	H	^							
9	6	K	^							

孩子链表
带双亲的孩子链表

C 语言的类型描述：

孩子结点结构 :

child	next
-------	------

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

树结构 :

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

双亲结点结构 :

data	firstchild
------	------------

```
typedef struct {  
    TElemType  data;  
    ChildPtr  firstchild;  
    // 孩子链表头指针  
} CTBox;
```

3 孩子兄弟表示法（二叉树表示法，二叉链表表示法）

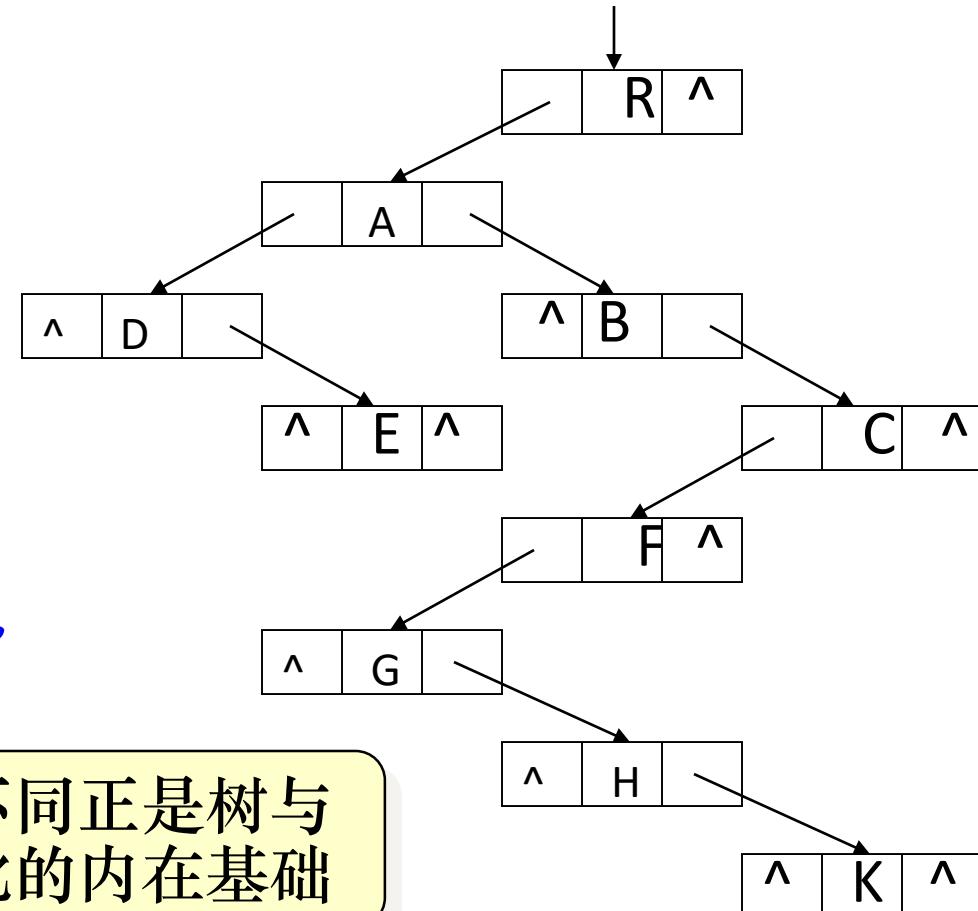
实现：用二叉链表作树的存储结构，链表中每个结点的两个指针域分别指向其第一个孩子结点和下一个兄弟结点

注

孩子兄弟链表的结构

形式与二叉链表完全

相同，但存储结点中指针的含
义不同：**二叉链表中结点的左
右指针分别指向该结点的左右
孩子；而孩子兄弟链表结点的
左右指针分别指向它的“长子”
和“大弟”。**



这种解释上的不同正是树与
二叉树相互转化的内在基础

C 语言的类型描述：

结点结构：

firstchild	data	nextsibling
------------	------	-------------

```
typedef struct CSNode{
```

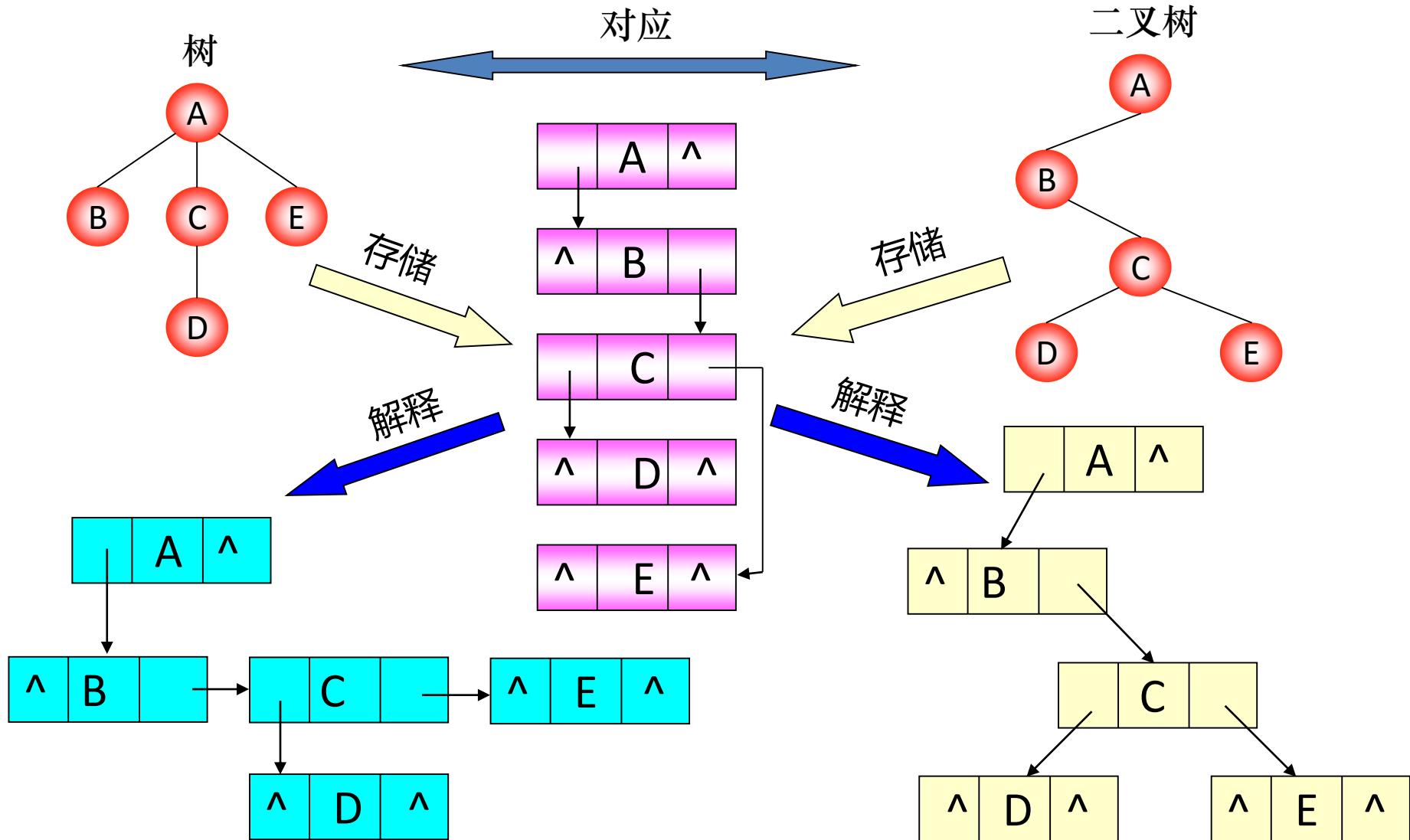
```
    ElemType      data;
```

```
    struct CSNode *firstchild, *nextsibling;
```

```
} CSNode, *CSTree;
```

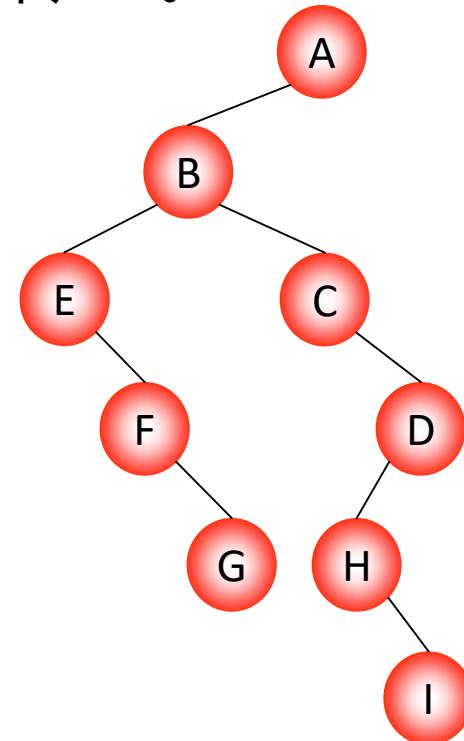
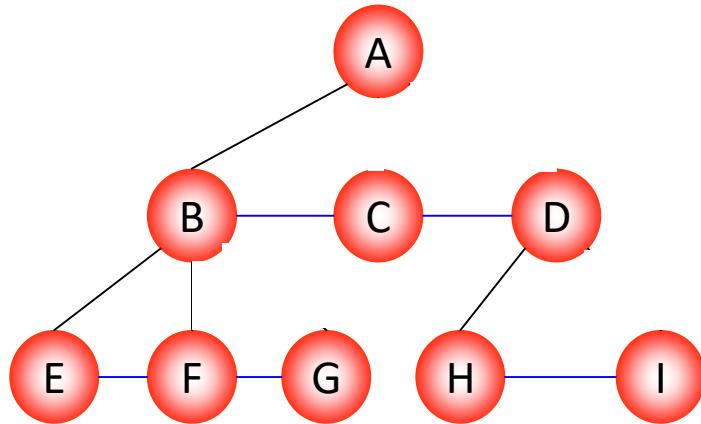
6.4.2 森林与二叉树的转换

● 树与二叉树的转换



● 将树转换成二叉树

- 1、加线：在兄弟之间加一连线
- 2、抹线：对每个结点去除其与孩子之间的关系（第一孩子除外）
- 3、旋转：以树的根结点为轴心，顺时针转45°。



树变二叉树：兄弟相连留长子。

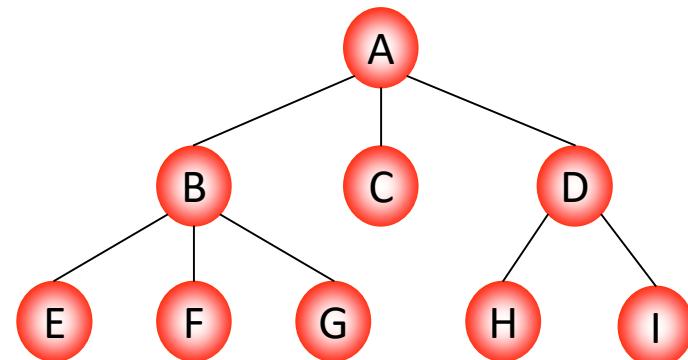
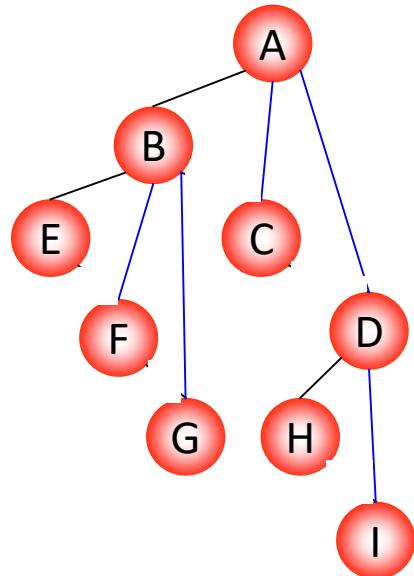
树转换成的二叉树其右子树一定为空。

● 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。

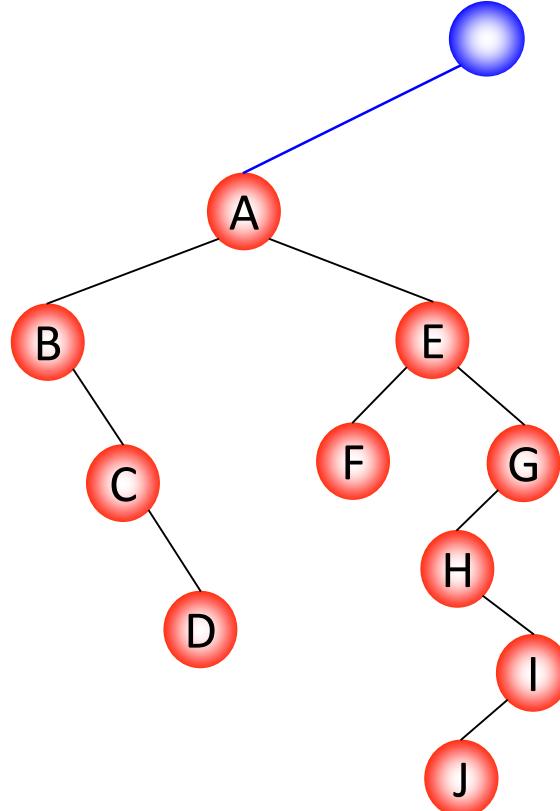
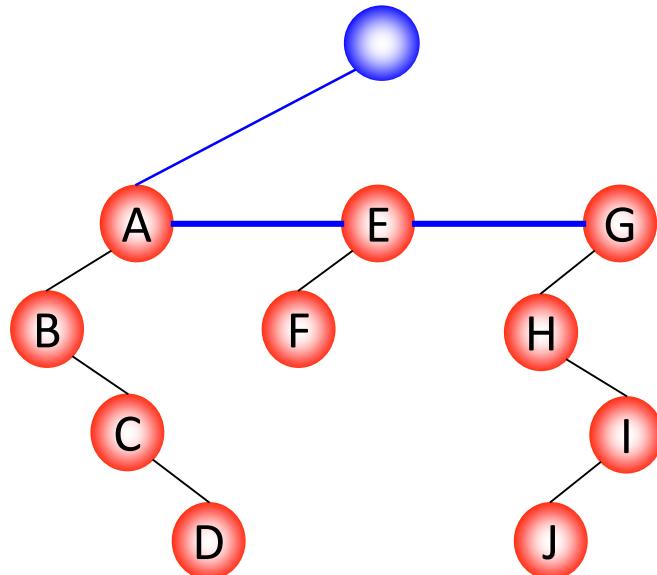
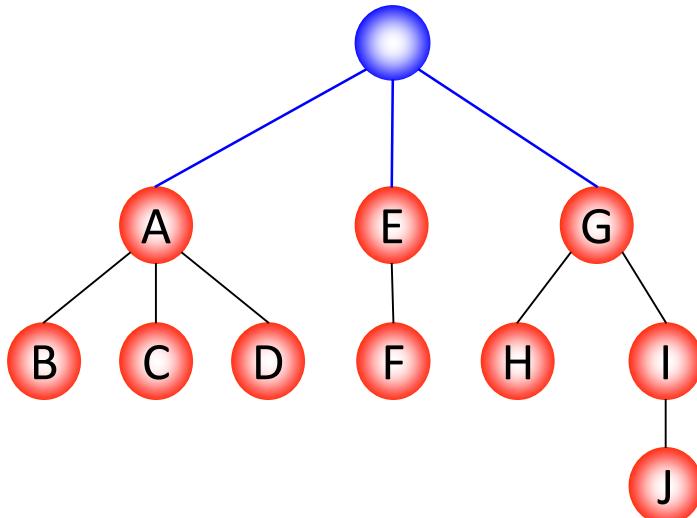
抹线：抹掉原二叉树中双亲与右孩子之间的连线。

调整：将结点按层次排列，形成树结构。



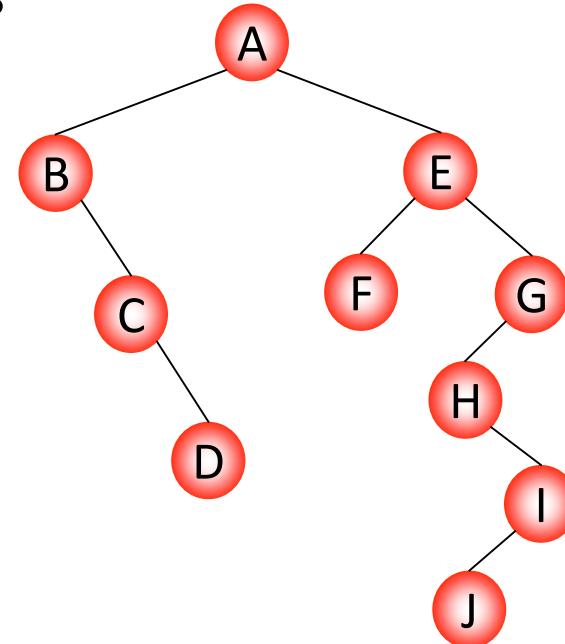
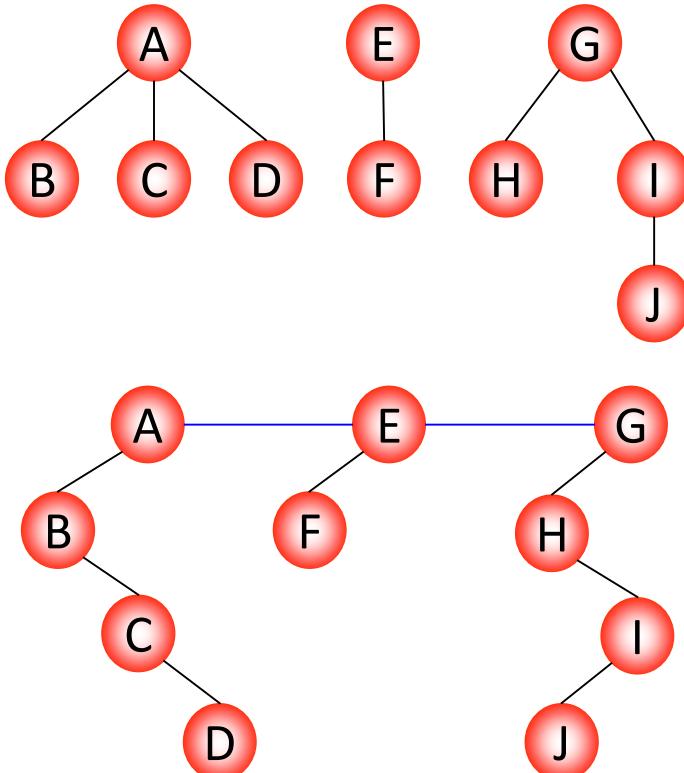
二叉树变树：左孩右右连双亲，去掉原来右孩线。

- 将森林转换成二叉树



- 将森林转换成二叉树

- 将各棵树分别转换成二叉树。
- 将每棵二叉树的根结点用线相连。
- 以第一棵二叉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。

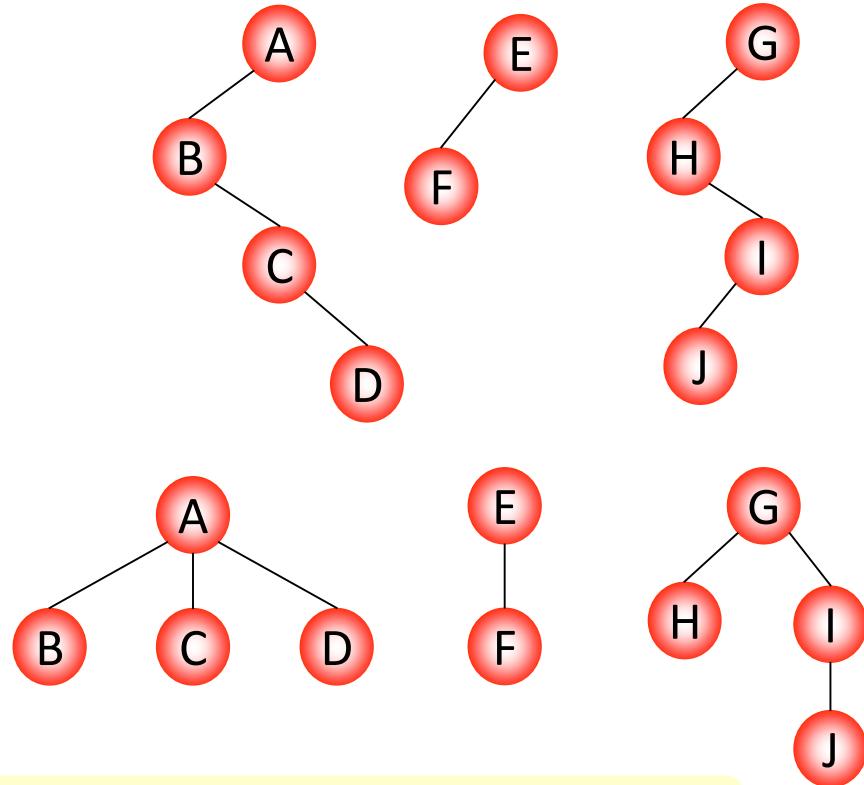
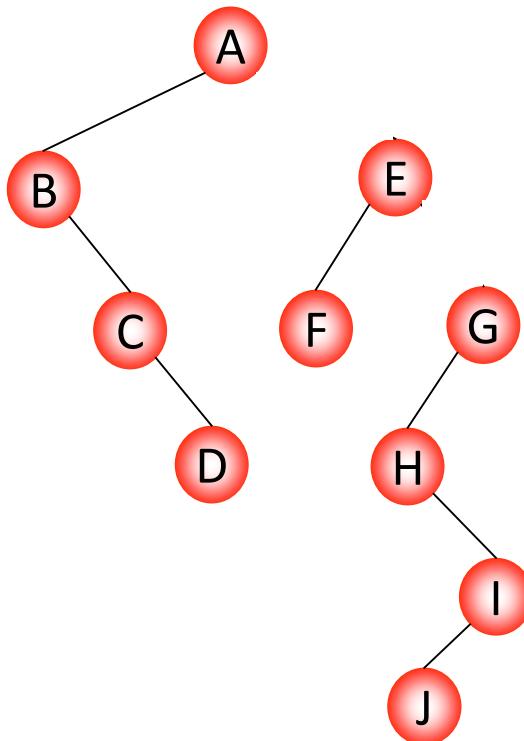


森林变二叉树：树变二叉根相连。

● 将二叉树转换成森林

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。



二叉树变森林：去掉全部右孩线，孤立二叉再还原。

6.4.3 树与森林的遍历

● 树的遍历

1、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。

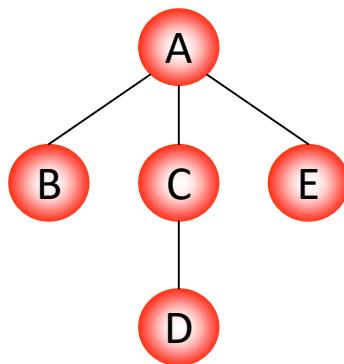
2、后根（次序）遍历：

若树不空，则先依次后根遍历各棵子树，然后访问根结点。

3、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。

遍历结果：



先根遍历： A B C D E

后根遍历： B D C E A

按层次遍历： A B C E D

● 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。

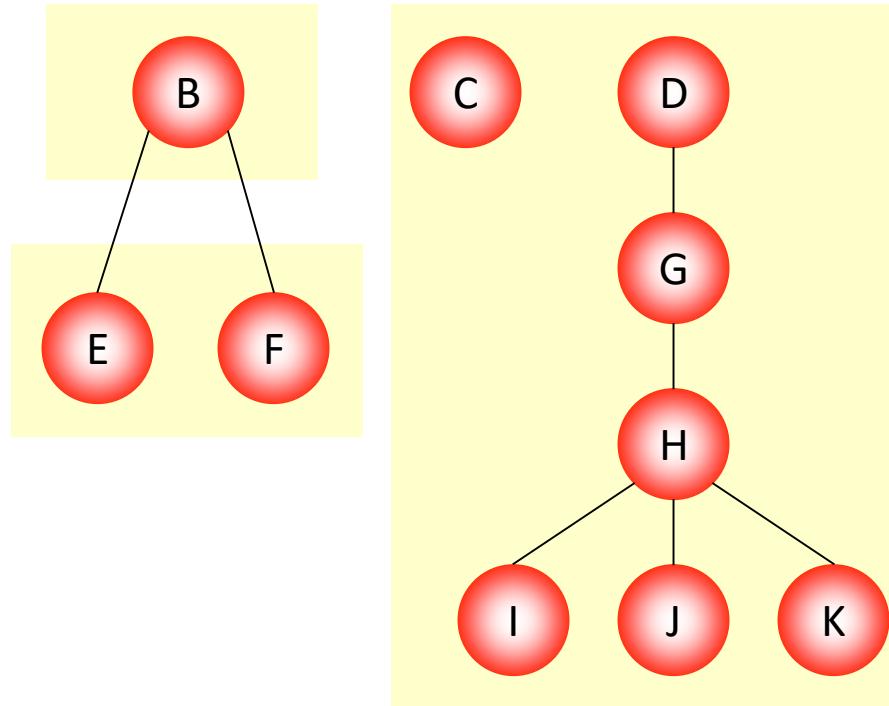
➤ 先序遍历：若森林不空，则

访问森林中第一棵树的根结点；

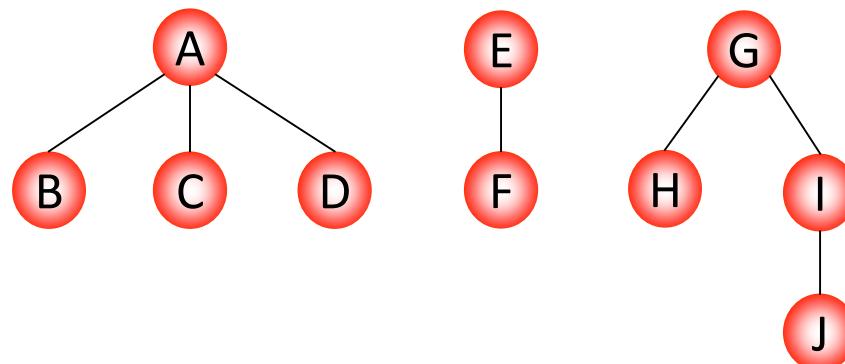
先序遍历森林中第一棵树的子树森林；

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。



- 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。



遍历结果：

先序遍历： A B C D E F G H I J

中序遍历： B C D A F E H J I G

树和二叉树的相关概念、术语

二叉树的五个性质

二叉树存储、遍历、线索化

树和森林

哈夫曼树



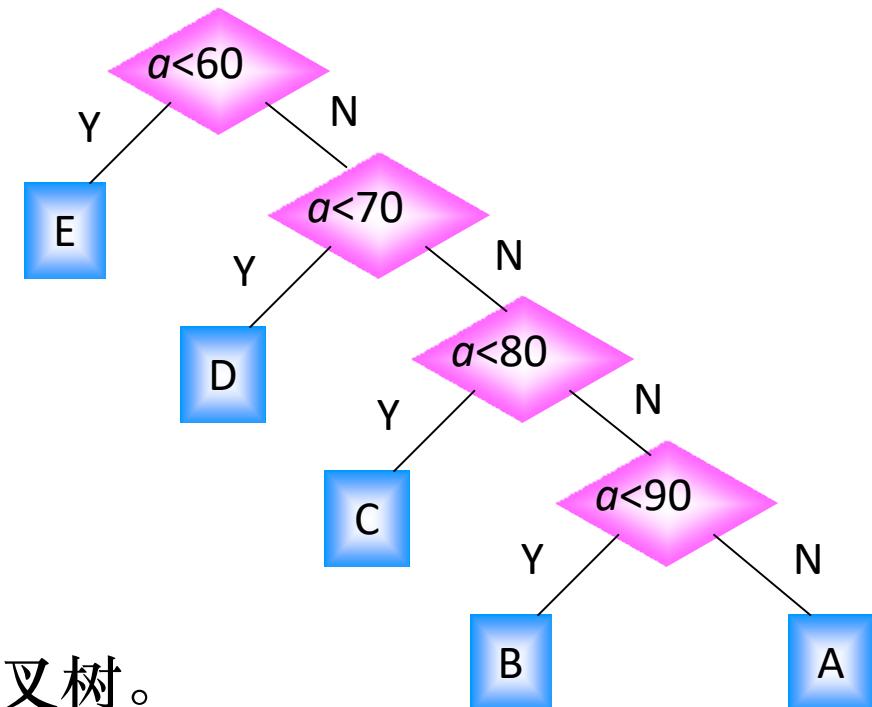
哈夫曼 (Huffman) 树及其应用

问题：什么是哈夫曼树？

例：将学生的百分制成绩转换为五分制成绩： ≥ 90 分: A ,
 $80 \sim 89$ 分: B , $70 \sim 79$ 分: C , $60 \sim 69$ 分: D , < 60 分: E。

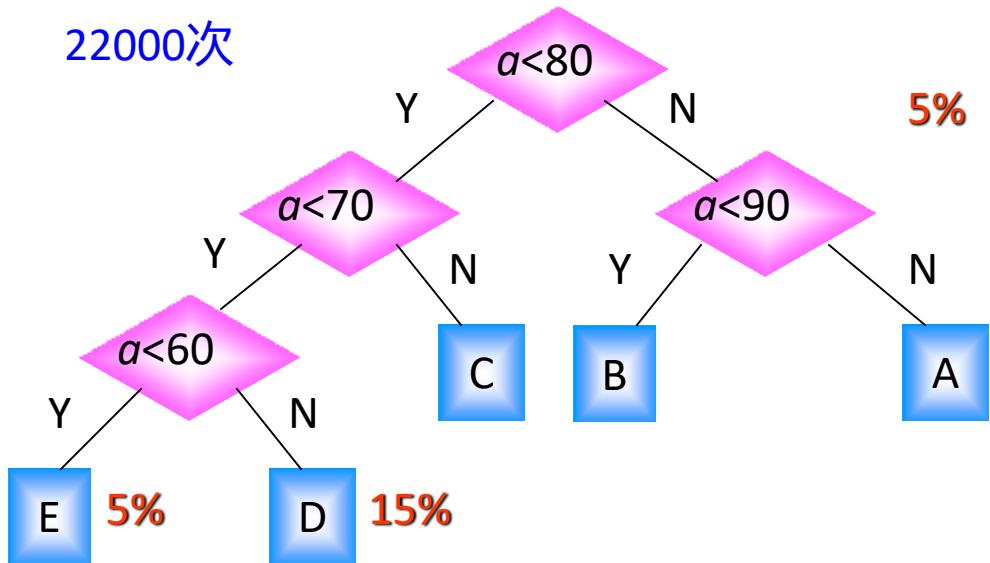
转换程序可用条件语句实现：

```
if ( $a < 60$ )  $b == 'E'$ ;  
else if ( $a < 70$ )  $b == 'D'$ ;  
else if ( $a < 80$ )  $b == 'C'$ ;  
else if ( $a < 90$ )  $b == 'B'$ ;  
else  $b == 'A'$ ;
```

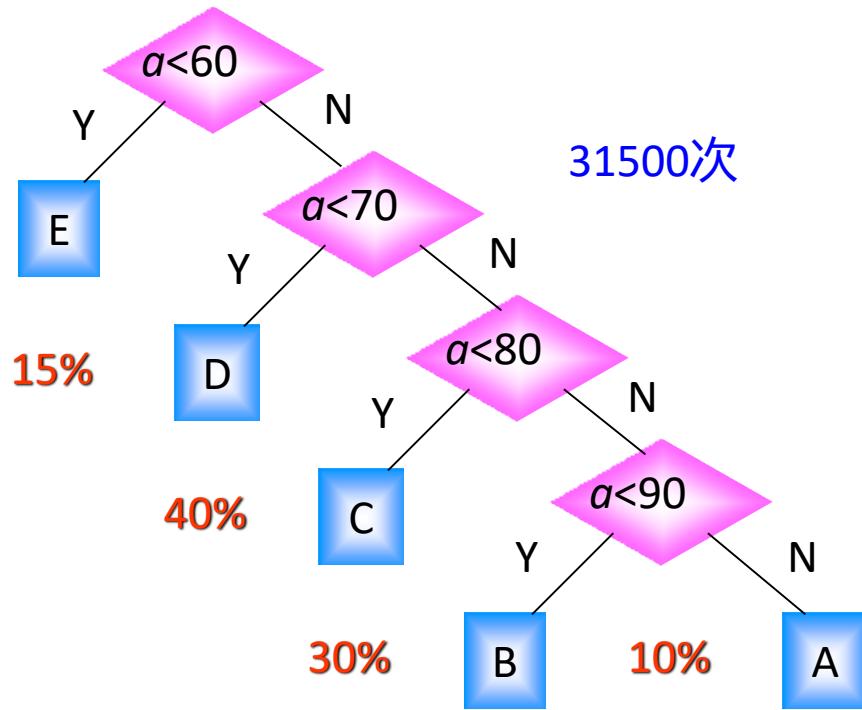


判别树：用于描述分类过程的二叉树。

22000次



31500次



显然：两种判别树的效率是不一样的。

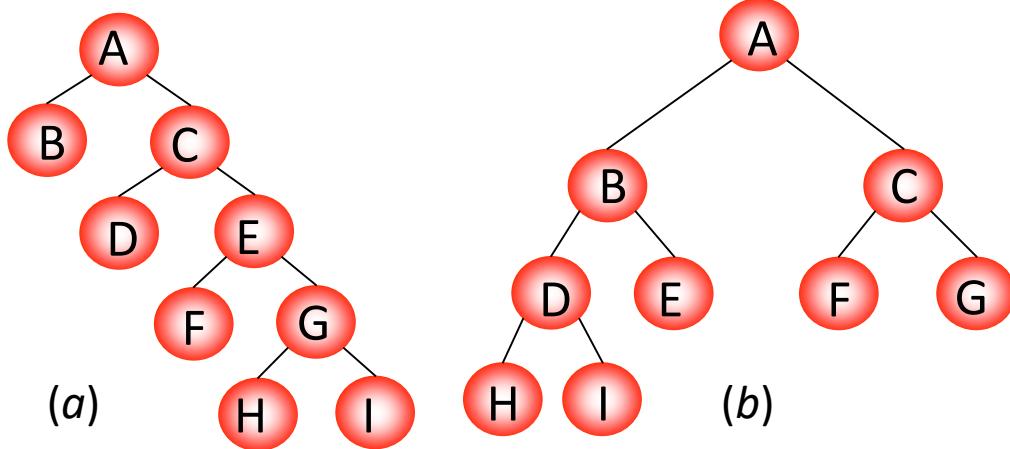
问题：能不能找到一种效率最高的判别树呢？

哈夫曼树

● 基本概念：

路径：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间的路径。

结点的路径长度：两结点间路径上的分支数。



(b) 从 A 到 B, C, D, E, F, G, H, I 的路径长度分别为 1, 1, 2, 2, 2, 2, 3, 3。

树的路径长度：从树根到每一个结点的路径长度之和。记作：TL

$$TL(a) = 0 + 1 + 1 + 2 + 2 + 3 + 3 + 4 + 4 = 20$$

$$TL(b) = 0 + 1 + 1 + 2 + 2 + 2 + 2 + 3 + 3 = 16$$

完全二叉树是路径长度最短的二叉树。

权：将树中结点赋给一个有着某种含义的数值，则这个数值称为该结点的权。

结点的带权路径长度：从根结点到该结点之间的路径长度与该结点的权的乘积。

树的带权路径长度：树中所有叶子结点的带权路径长度之和。

记作：

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

权值
结点到根的路径长度

哈夫曼树： 最优树

带权路径长度 (WPL) 最短的树



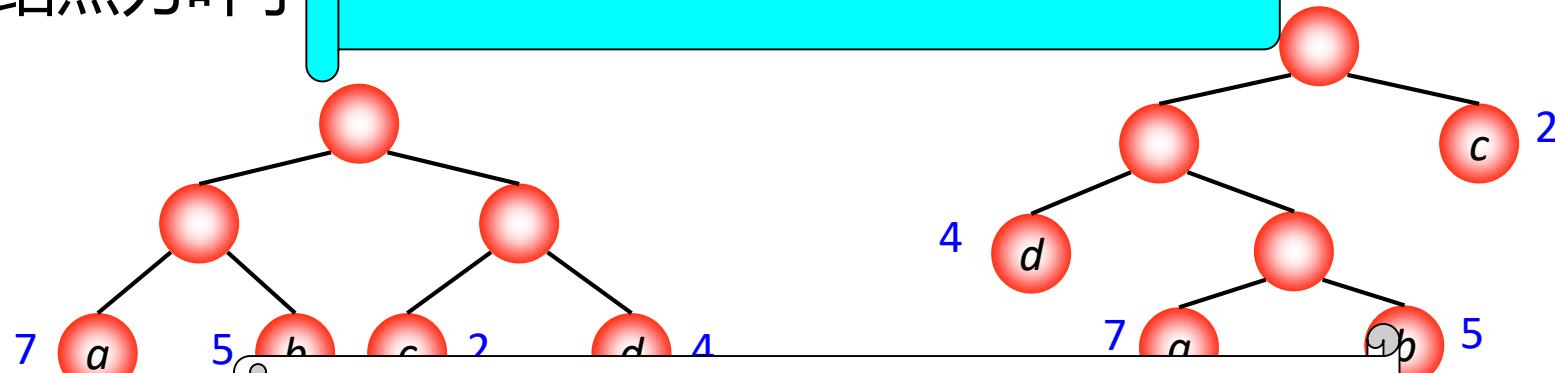
“带权路径长度最短” 是在 “度相同” 的树中比较而得的结果，因此有最优二叉树、最优三叉树之称等等。

哈夫曼树： 最优二叉树

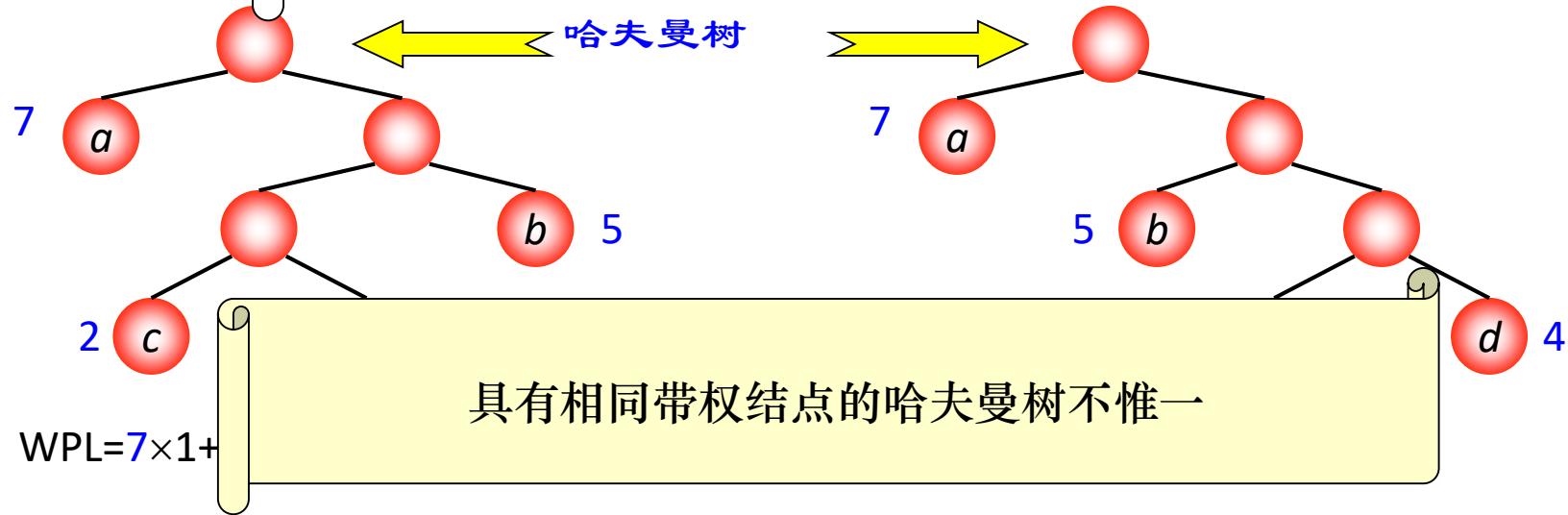
带权路径长度 (WPL) 最短的二叉树

因为构造这种树的算法是由哈夫曼于 1952 年提出的，所以被称为哈夫曼树，相应的算法称为 **哈夫曼算法**。

例：有4个结点，构造以此4个结点为叶子的满二叉树不一定是哈夫曼树



哈夫曼树中权越大的叶子离根越近



● 哈夫曼算法（构造哈夫曼树的方法）

(1)、根据 n 个给定的权值 $\{w_1, w_2, \dots, w_n\}$ 构成 n 棵二叉树的森林 $F=\{T_1, T_2, \dots, T_n\}$ ，其中 T_i 只有一个带权为 w_i 的根结点。

构造森林全是根

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

选用两小造新树

(3)、在 F 中删除这两棵树，同时将新得到的二叉树加入森林中。

删除两小添新人

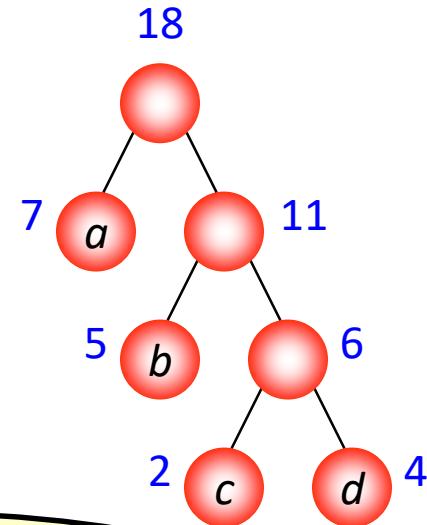
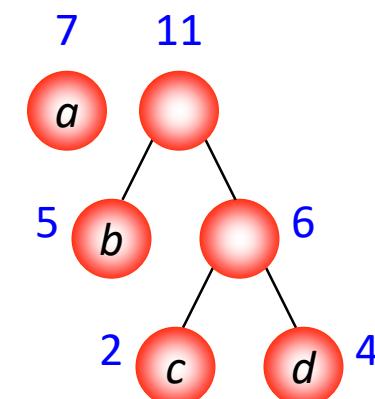
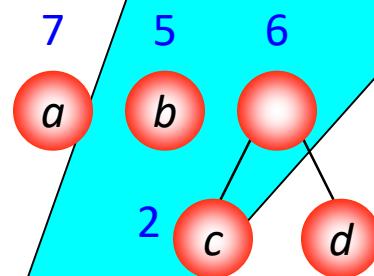
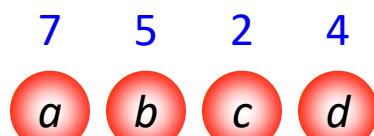
(4)、重复(2)和(3)，直到森林中只有一棵树为止，这棵树即为哈夫曼树。

重复 2、3 剩单根

哈夫曼算法口诀

例：有4个结点 $a, b,$

包含 n 棵树的森林要经过 $n-1$ 次合并才能形成哈夫曼树，共产生 $n-1$ 个新结点

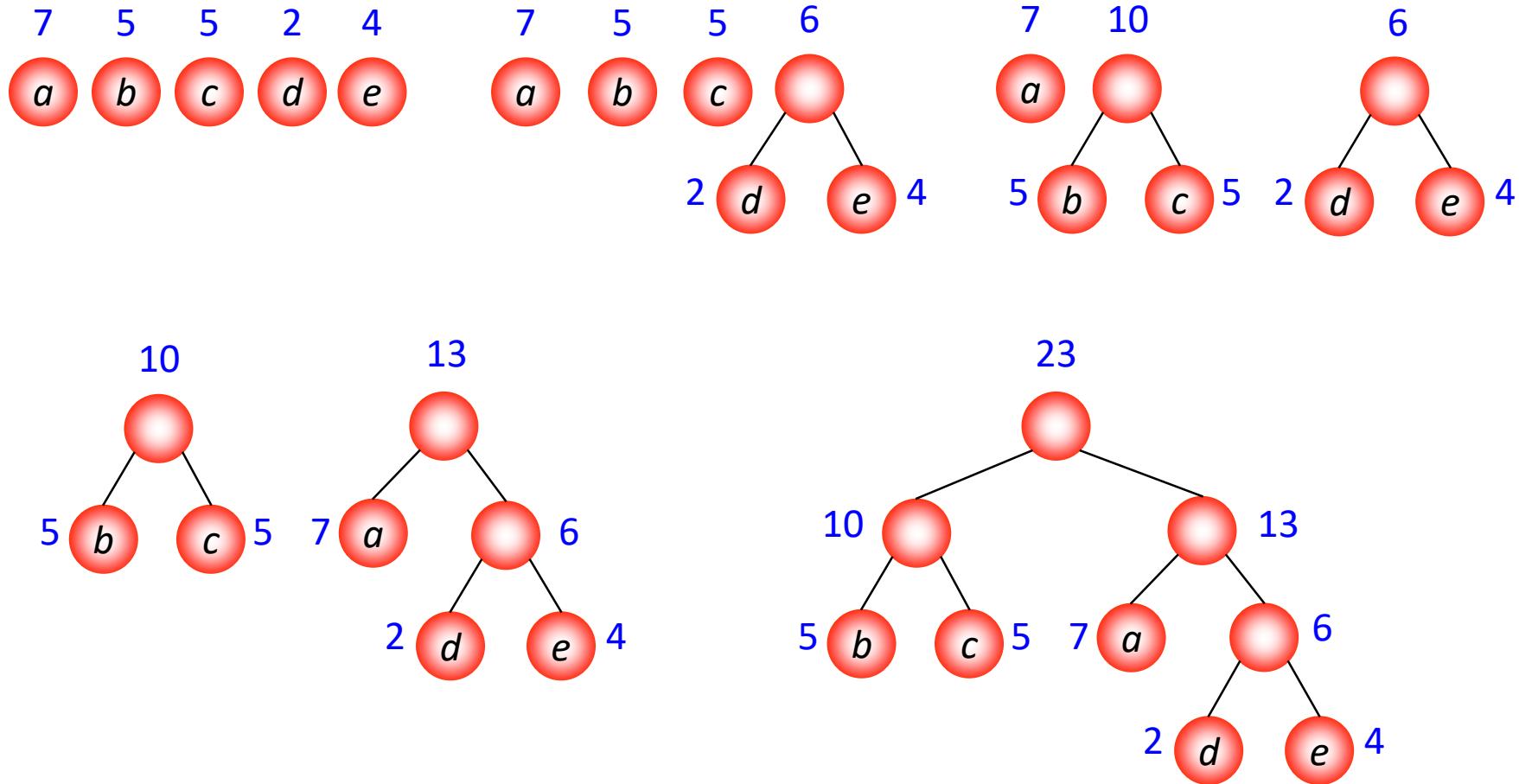


由二叉树的性质 3

包含 n 个叶子结点的哈夫曼树中共有 $2n - 1$ 个结点。

哈夫曼树的结点的度数为 0 或 2，没有度为 1 的结点。

例：有5个结点 a, b, c, d, e ，权值分别为 7, 5, 5, 2, 4，
构造哈夫曼树。



哈夫曼编码

哈夫曼树的应用很广，哈夫曼编码就是其在电讯通信中的应用之一。在电讯通信业务中，通常用二进制编码来表示字母或其他字符，并用这样的编码来表示字符序列。

例：如果需传送的电文为‘ABACCDAA’，它只用到四种字符，用两位二进制编码便可分辨。假设 A, B, C, D 的编码分别为 00, 01, 10, 11，则上述电文便为‘00010010101100’（共 14 位），译码员按两位进行分组译码，便可恢复原来的电文。

能否使编码总长度更短呢？

实际应用中各字符的出现频度不相同



用短(长)编码表示频率大(小)的字符



使得编码序列的总长度最小，使所需总空间量最少



数据的最小冗余编码问题

在上例中，若假设 A, B, C, D 的编码分别为 0, 00, 1, 01，则电文 ‘ABACCCDA’ 便为 ‘000011010’ (共 9 位)。

但此编码存在多义性：可译为 ‘BBCCDA’、‘ABACCCDA’、‘AAAAACCACA’ 等。

➤ 译码的惟一性问题

要求任一字符的编码都不能是另一字符编码的前缀！
这种编码称为**前缀编码**（其实是非前缀码）。

在编码过程要考虑两个问题 {

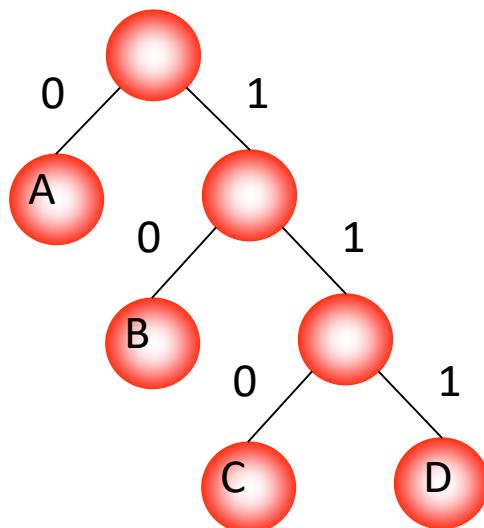
- 数据的最小冗余编码问题
- 译码的惟一性问题

利用最优二叉树可以很好地解决上述两个问题

➤ 用二叉树设计二进制前缀编码

以电文中的字符作为叶子结点构造二叉树。然后将二叉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。**如此得到的即为二进制前缀编码。** ?

例：



编码：
A: 0
B: 10
C: 110
D: 111

任意一个叶子结点都不可能在其它叶子结点的路径中。

➤ 用哈夫曼树设计总长最短的二进制前缀编码

假设各个字符在电文中出现的次数（或频率）为 w_i ，其编码长度为 l_i ，电文中只有 n 种字符，则电文编码总长为：

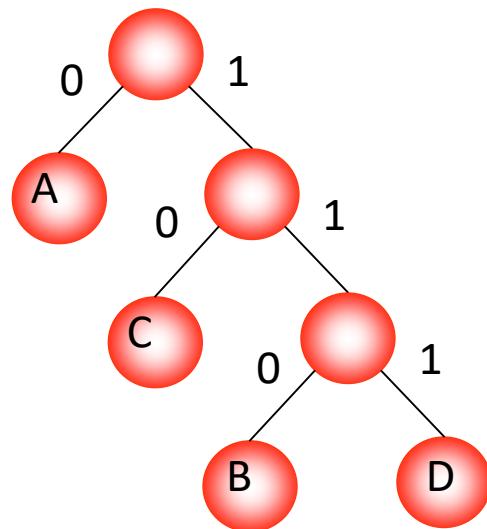
$$WPL = \sum_{i=1}^n w_i l_i \longrightarrow \begin{array}{l} \text{从根到叶子的路径长度} \\ \downarrow \\ \text{叶子结点的权} \end{array}$$

设计电文总长最短的编码 \longleftrightarrow 设计哈夫曼树（以 n 种字符出现的频率作权）

由哈夫曼树得到的二进制前缀编码称为**哈夫曼编码**

例：如果需传送的电文为 ‘ABACCCDA’，即：A, B, C, D 的频率（即权值）分别为 0.43, 0.14, 0.29, 0.14，试构造哈夫曼编码。

解：

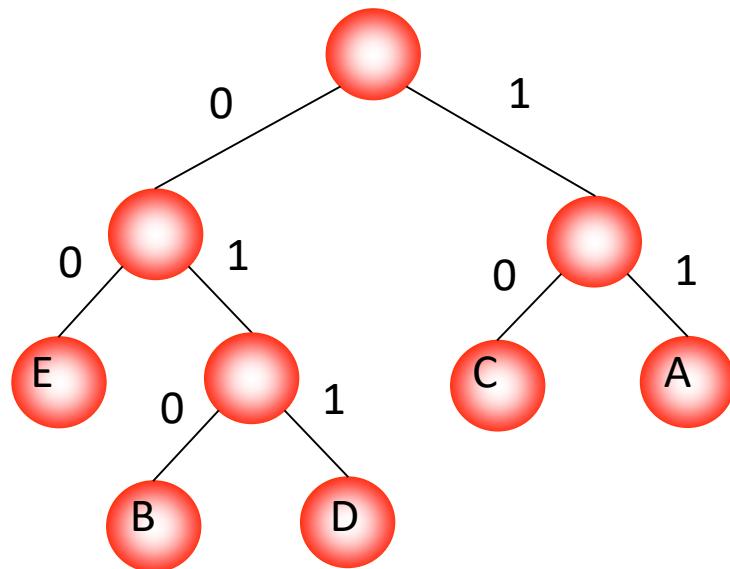


编码	A: 0
	C: 10
	B: 110
	D: 111

则电文 ‘ABACCCDA’ 便为 ‘0110010101110’（共 13 位）。

例：如果需传送的电文为 ‘ABCACCDAAEAE’，
即：A, B, C, D, E 的频率（即权值）分别为
0.36, 0.1, 0.27, 0.1, 0.18，试构造哈夫曼编码。

解：

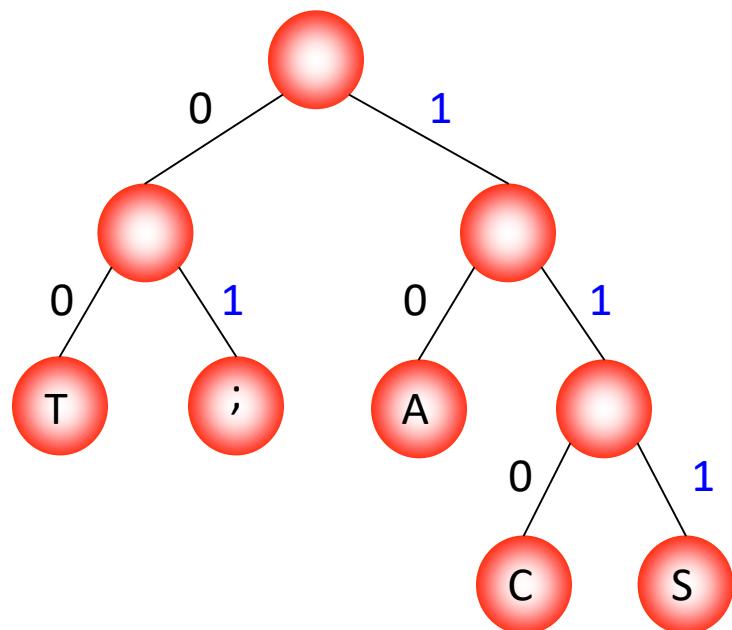


编码：
A: 11
C: 10
E: 00
B: 010
D: 011

则电文 ‘ABCACCDAAEAE’ 便为 ‘110101011101001111001100’
(共 24 位, 比 33 位短)。

➤ 译码

从哈夫曼树根开始，对待译码电文逐位取码。若编码是“0”，则向左走；若编码是“1”，则向右走，一旦到达叶子结点，则译出一个字符；再重新从根出发，直到电文结束。



电文为 “1101000”

译文只能是 “CAT”

作业：

- 1、一棵哈夫曼树有 19 个结点，则其叶子结点的个数是()。
- 2、有七个带权结点，其权值分别为 3, 7, 8, 2, 6, 10, 14，试以它们为叶结点构造一棵哈夫曼树（请按照每个结点的左子树根结点的权小于等于右子树根结点的权的次序构造），并计算出带权路径长度WPL及该树的结点总数。
- 3、有一电文共使用五种字符 a, b, c, d, e，其出现频率依次为 4, 7, 5, 2, 9。
 - (1)、试画出对应的编码哈夫曼树（要求左子树根结点的权小于等于右子树根结点的权）。
 - (2)、求出每个字符的哈夫曼编码。
 - (3)、求出传送电文的总长度。
 - (4)、并译出编码系列11000111000101011的相应电文。
- 4、对于给定的一组权值 $W=\{1, 3, 7, 8, 14, 20, 28\}$ 建立哈夫曼树，并计算带权路径长度。
- 5、假定有 7 个字符 a, b, c, d, e, f, g 出现的概率分别为 0.07, 0.09, 0.14, 0.23, 0.44, 0.58, 0.77，求这 7 个字符的哈夫曼编码。

小结

本章是**重点章**，**二叉树**又是本章的重点内容。

要求：

- 1、熟悉树的定义、存储结构、遍历；
- 2、熟悉二叉树的定义、性质、存储结构、线索化；
- 3、**熟练掌握**二叉树的遍历；
- 4、**熟练掌握**树、森林与二叉树的转换；
- 5、**熟练掌握**哈夫曼树及哈夫曼编码等内容。

Thank You !