

# 数据结构与算法

## 第6章 树

主讲：赵海燕

北京大学信息科学技术学院  
“数据结构与算法” 教学组

国家精品课 “数据结构与算法”  
<http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/>

张铭，王腾蛟，赵海燕  
高等教育出版社，2008. 6，“十一五”国家级规划教材

# 「主要内容」

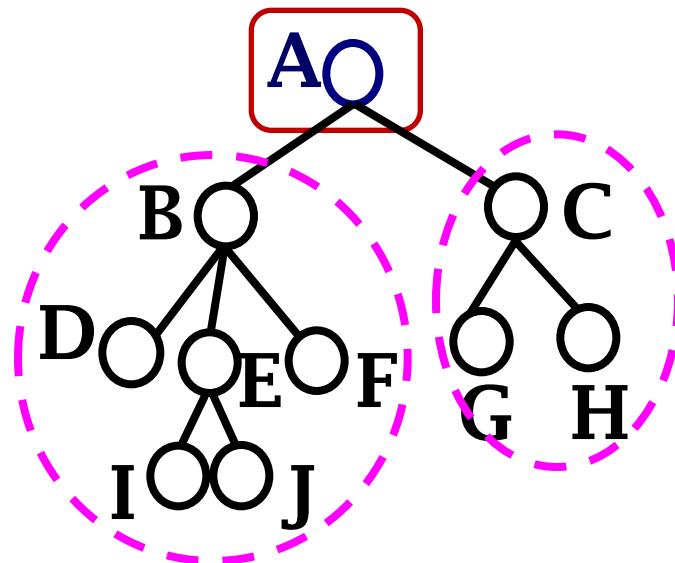
- 树的概念
- 树的链式存储
- 树的顺序存储
- K叉树
- 树知识点总结

# 「树的逻辑结构

- 包含  $n$  个结点的有穷集合  $K (n>0)$ ，且在  $K$  上定义了一个关系  $r$ ，关系  $r$  满足以下条件：
  - 有且仅有一个结点  $k_0 \in K$ ，对于关系  $r$  来说没有前驱。结点  $k_0$  称作树的 **根**
  - 除结点  $k_0$  外， $K$  中的每个结点对于关系  $r$  来说都有且仅有**一个前驱**（父结点）
  - 除结点  $k_0$  外的任何结点  $k \in K$ ，均存在一条从根到结点  $k$  的 **路径**，即，存在一个结点序列  $k_0, k_1, \dots, k_s$ ，使得有序对  $\langle k_{i-1}, k_i \rangle \in r (1 \leq i \leq s)$ ，且  $k_0$  为树根、 $k_s = k$

# 树

- 非空树是包含一个或多个结点的有限集合 $T$ ，使得：
  - 有一个特别标出称作根的结点
  - 除根以外其他结点被分成 $m$ 个( $m \geq 0$ )不相交的集合 $T_1, T_2, \dots, T_m$ ，而且这些集合的每一个又都是树。树 $T_1, T_2, \dots, T_m$ ，称作这个根的子树
- 递归定义：只包含一个结点的树必然仅由根组成，包含 $n > 1$ 个结点的树借助于少于 $n$ 个结点的树来定义



# 「森林 (forest)

- 由**零个或多个**互不相交的树所组成的集合（通常为有序集合），也称**树林**
  - 就**逻辑结构**而言，任何一棵树都是一个二元组 $T = (rt, F)$ ，其中  $rt$  为树的根结点， $F$  是  $m$  ( $m \geq 0$ ) 棵子树构成的森林： $F = (T_1, T_2, \dots, T_m)$ ，其中
    - ◆  $T_i = (r_i, F_i)$  称作根  $rt$  的第  $i$  棵子树；
    - ◆ 当  $m \neq 0$  时，在树根和其子森林之间存在下列关系：  
$$RF = \{<rt, r_i> \mid i=1, 2, \dots, m, m > 0\}$$
- 森林中所有**树根**彼此互为**兄弟**
- 其他概念与术语同树

# 森林

- 自然界的树和森林是不同的概念，而数据结构的树和森林只有微小的差别
  - 一棵树，**删除树根**，其子树就组成了森林
  - 加入一个结点作为**根**，森林就转化成了一棵树



# 「树结构的表示法

- 形式语言表示法
- 树形表示法
- 凹入表表示法
- 文氏图表示法
- 嵌套括号表示法

# 树结构的表示法

- 假设逻辑结构如下所示:

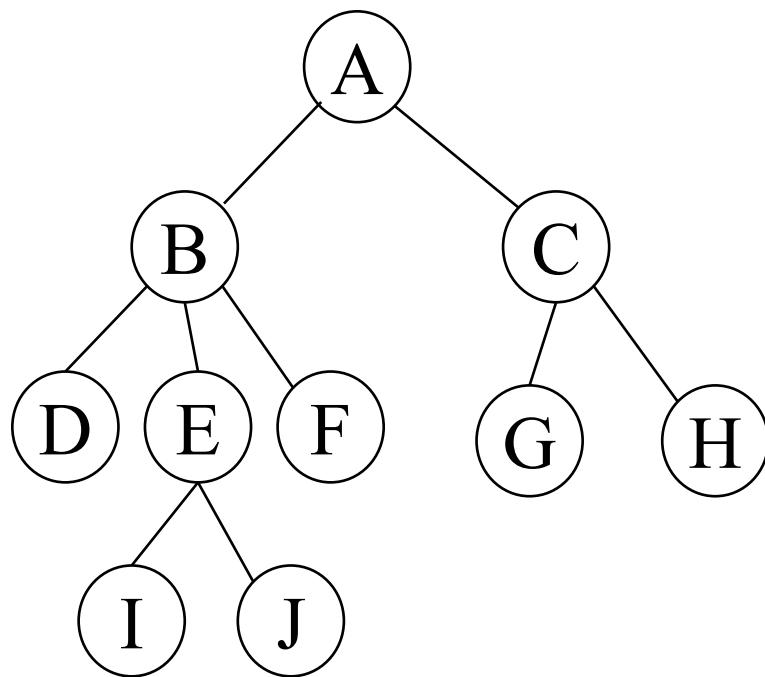
## 结点集合

$$K = \{A, B, C, D, E, F, G, H, I, J\}$$

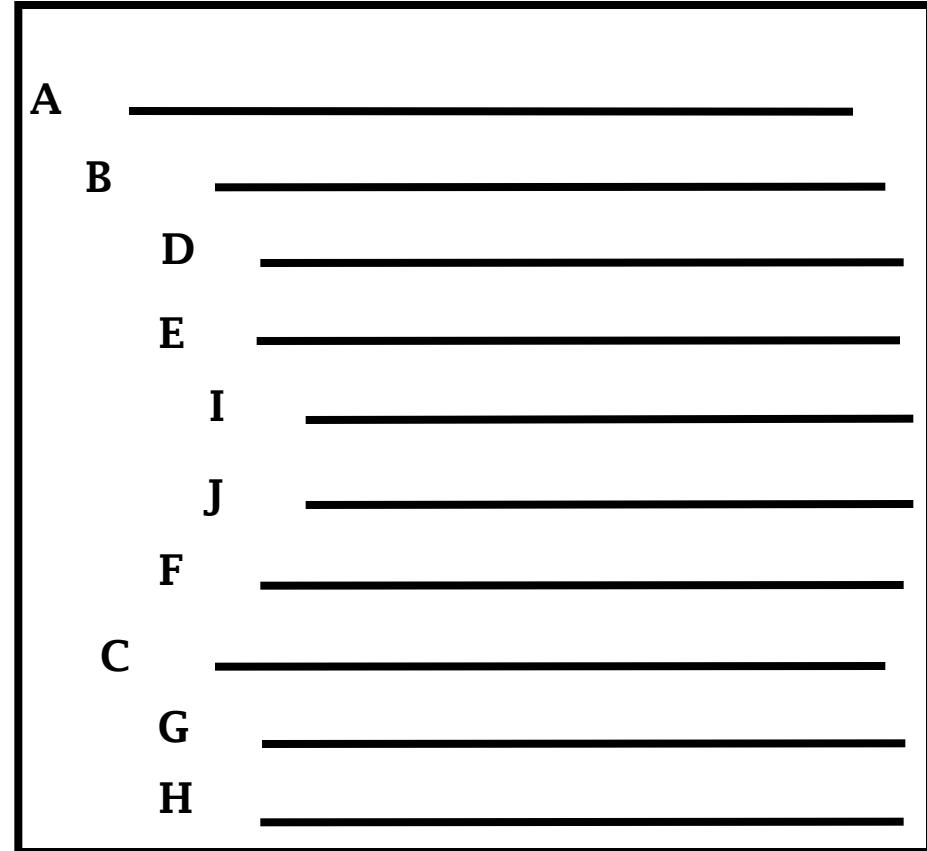
## K上的关系

$$\begin{aligned} r = \{ & \langle A, B \rangle, \langle A, C \rangle, \langle B, D \rangle, \\ & \langle B, E \rangle, \langle B, F \rangle, \langle C, G \rangle, \\ & \langle C, H \rangle, \langle E, I \rangle, \langle E, J \rangle \\ \} \end{aligned}$$

# 「树结构的表示法」

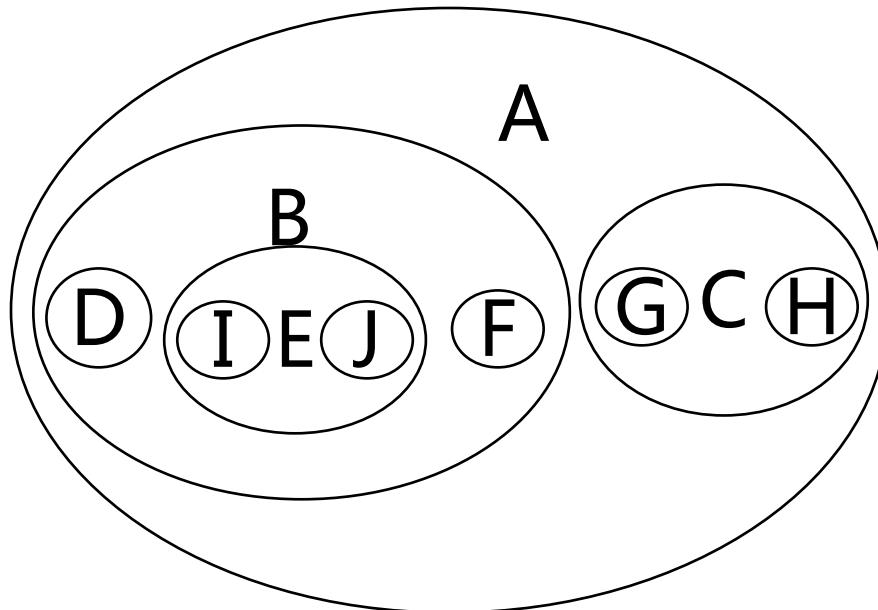


树形表示



凹入表示

# 「树结构的表示法



文氏图

(A(B(D)(E(I)(J)))(C(G)(H)))

嵌套括号表示法

# 「树结构相关概念

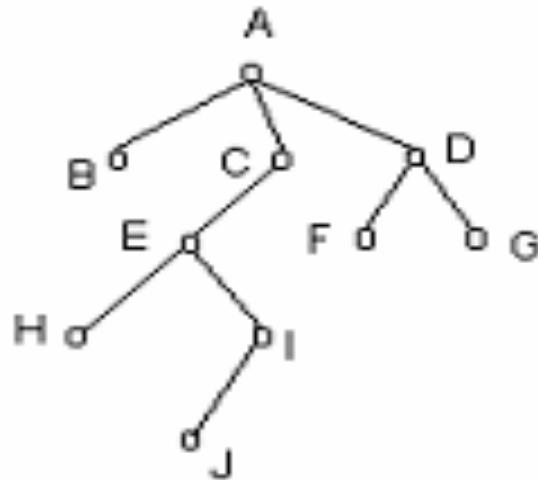
## ■ 有序树 (ordered tree)

- 若树T中的子树 $T_1, T_2, \dots, T_m$ 的相对次序是重要的，则称树T为**有向有序树**，简称**有序树**
- 有序树可以称 $T_1$ 是根的第一棵子树， $T_2$ 是根的第二棵子树，依次……等等

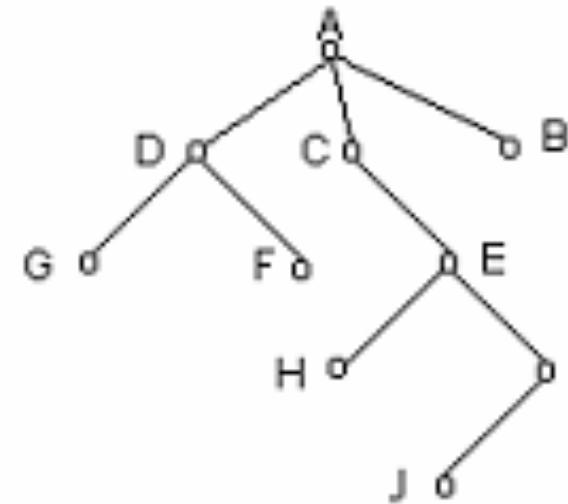
## ■ 无序树

- 对子树的次序不加区别

# 「树结构相关概念」



$t$



$t'$

按**有序树**:  $t \neq t'$

按**无序树**:  $t = t'$

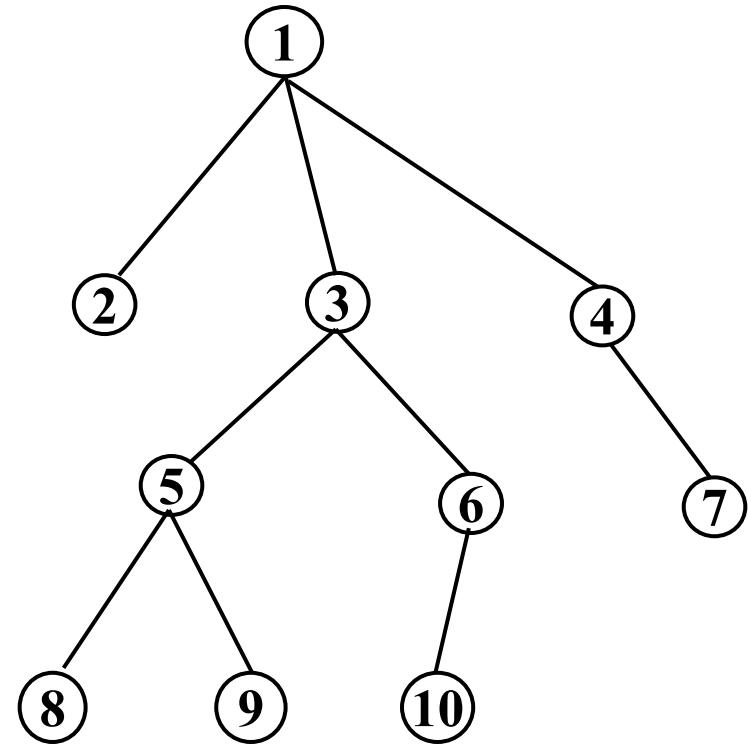
课程讨论一般为**有序树**

# 「有序树中结点次序」

## ■ 同层间可排左右：长子、次子

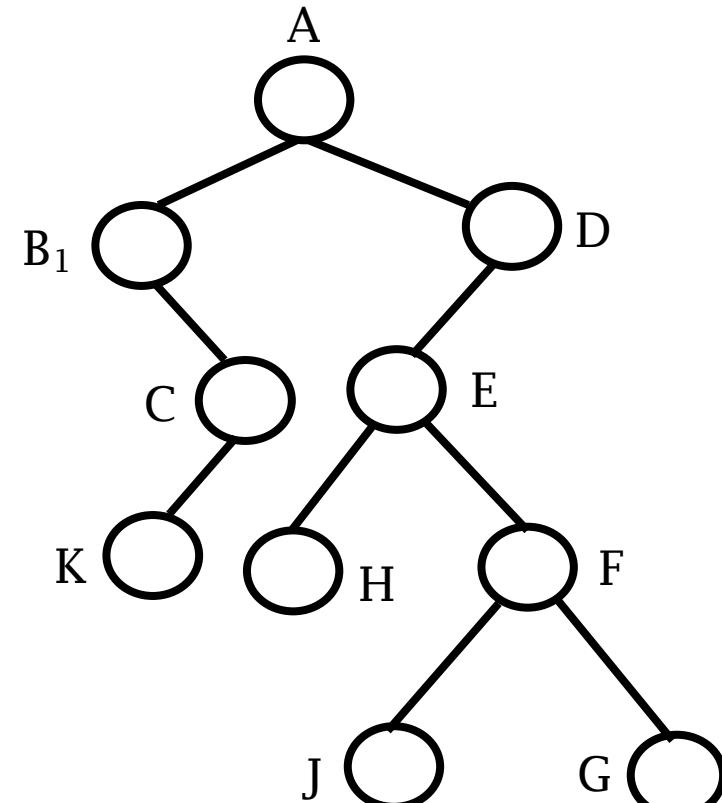
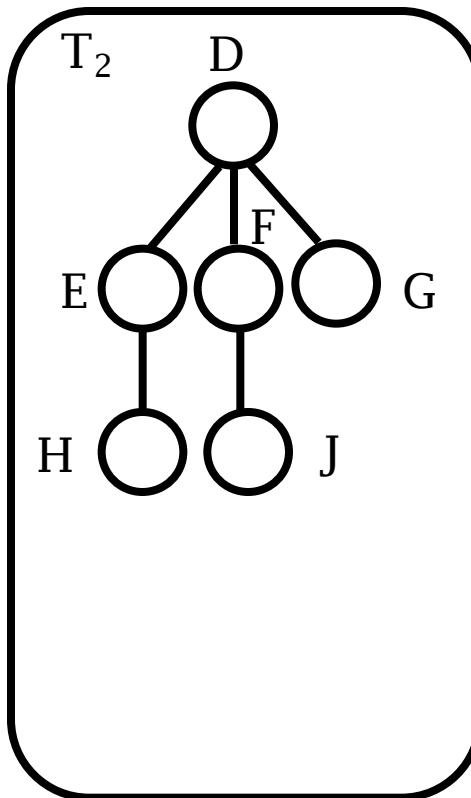
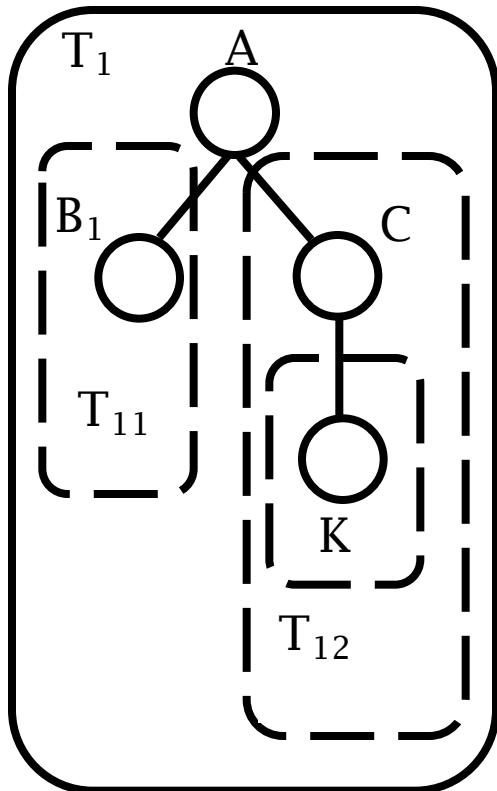
.....

- 最左/右子结点
- **注意：**度为 2 的有序树并不是二叉树
  - ◆ 第一子结点被删除后，第二子结点自然顶替成为第一



# 「森林与二叉树」

二者有关系吗？  
有何关系？



# 森林与二叉树的等价转换

- 树或森林与二叉树之间有一一对应的关系
  - 任何森林都唯一对应到一棵二叉树；反之，任何二叉树也都有唯一的一个森林与之对应

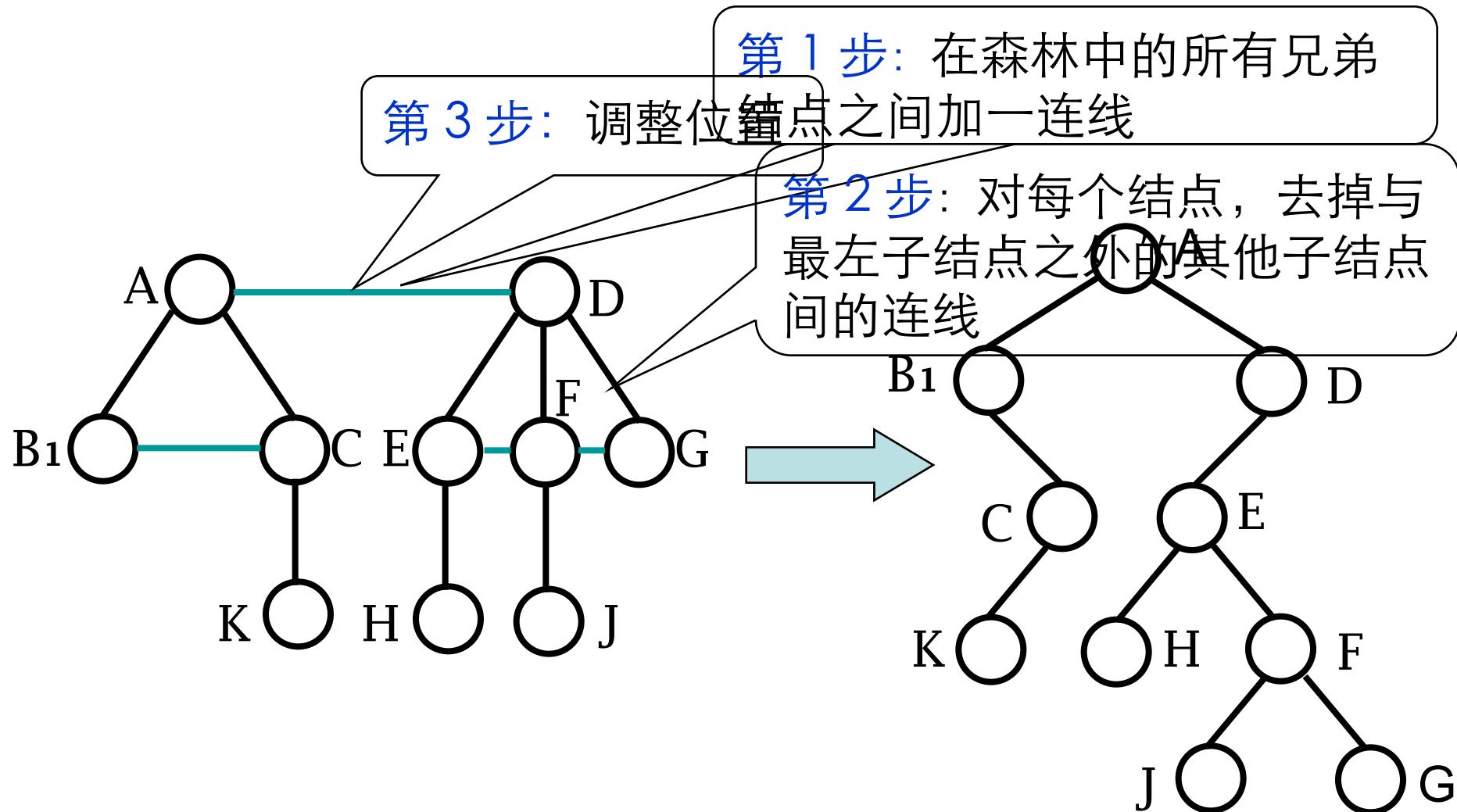
树、森林       $\longleftrightarrow$       二叉树

# 「森林转换为二叉树

## ■ 执行步骤（三步曲）：

- (1) **连线**：在所有相邻的兄弟结点之间连一条线
- (2) **切线**：对每个分支结点，只保留到其最左子结点的连线，删去与其它子结点的连线
- (3) **旋转**：以根结点为轴心，旋转整棵树

# 「森林转换成二叉树示例」

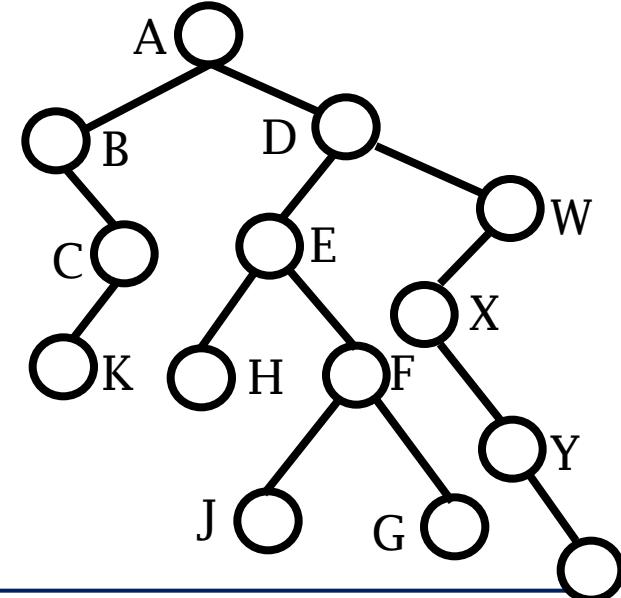
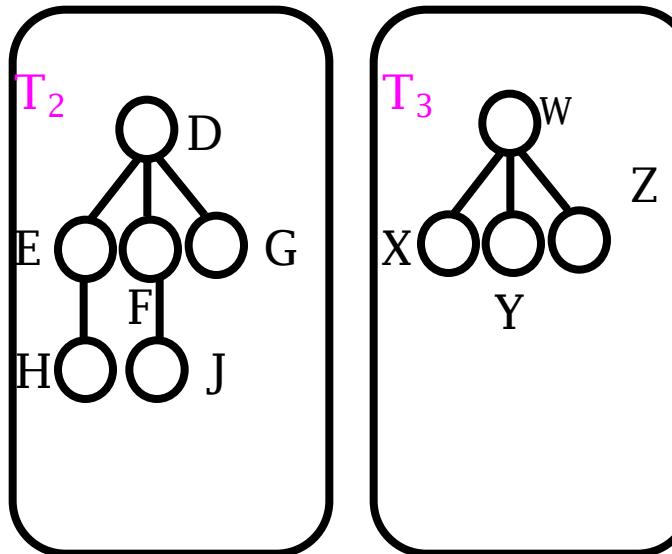
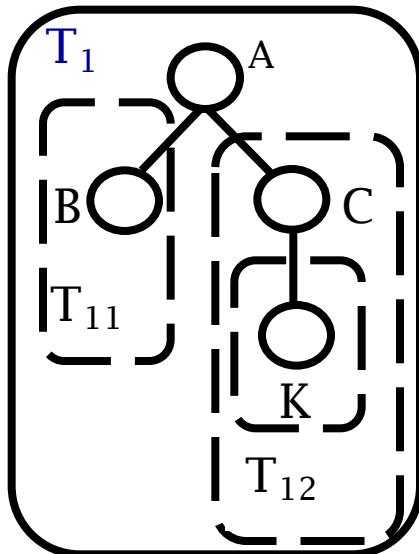


# 森林到二叉树的转换

## ■ 森林转化成二叉树的形式定义：

设有序集合  $F = \{T_1, T_2, \dots, T_n\}$  表示由树  $T_1, T_2, \dots, T_n$  组成的森林，则森林  $F$  可以按**如下规则递归**转换成二叉树  $B(F)$ ：

- 若  $F$  为空，即  $n = 0$ ，则  $B(F)$  为空；
- 若  $F$  非空，则  $B(F)$  的根是森林中第一棵树  $T_1$  的根  $W_1$ ， $B(F)$  的左子树是树  $T_1$  中根结点  $W_1$  的子树森林  $F = \{T_{11}, T_{12}, \dots, T_{1m}\}$  转换成的二叉树  $B(T_{11}, T_{12}, \dots, T_{1m})$ ； $B(F)$  的右子树是从森林  $F' = \{T_2, \dots, T_n\}$  转换而成的二叉树

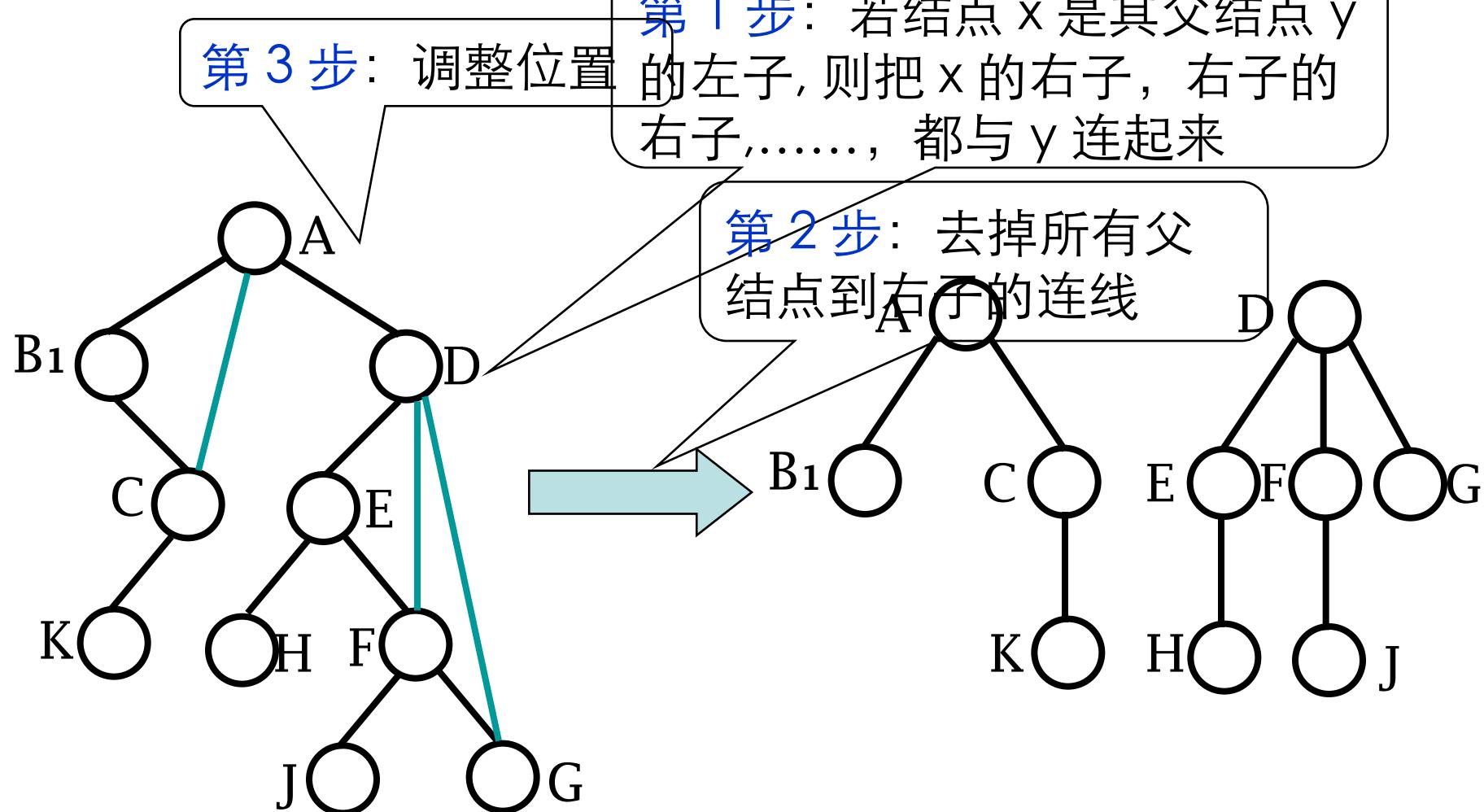


# 二叉树转换为森林

## ■ 执行步骤（三步）

- (1) 旋转：以根为轴，平面逆时针旋转
- (2) 补线：若某结点  $x$  是其父结点  $y$  的左子结点，则将该结点的右子，右子的右子，……，都与  $y$  用线连接起来
- (3) 删线：去掉原二叉树中所有父结点到右子结点的连线

# 二叉树转换为森林示例



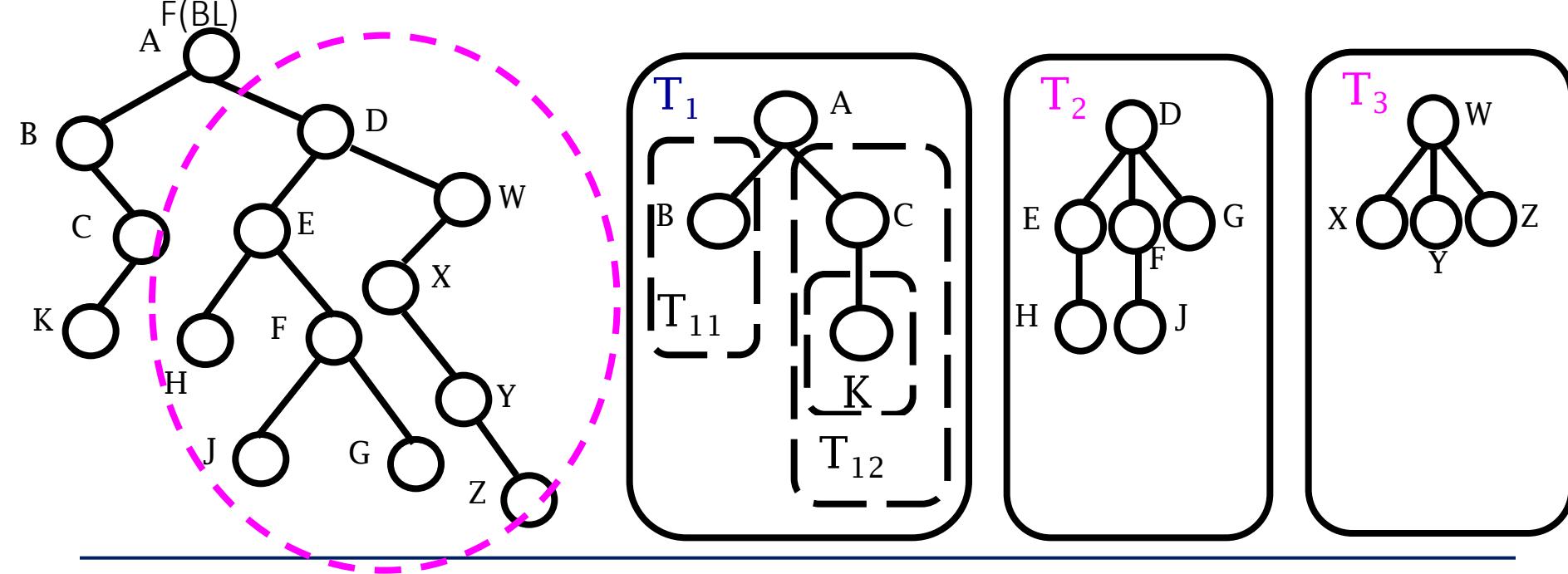
# 二叉树到森林的等价转换

- 二叉树转化成森林或树的形式定义：

设B是一棵二叉树， $r$ 是B的根， $BL$ 是 $r$ 的左子树， $BR$ 是 $r$ 的右子树，则对应于二叉树B的森林或树  $F(B)$  的形式定义：

- 若B为空，则 $F(B)$ 是空森林

- 若B不为空，则 $F(B)$ 是一棵树 $T_1$ 加上森林 $F(BR)$ ，其中树 $T_1$ 的根为 $r$ ， $r$ 的子树为 $F(BL)$



# 思考

- 树也是森林吗？
- 为什么要建立二叉树与森林的对应关系？

# 「树的抽象数据类型

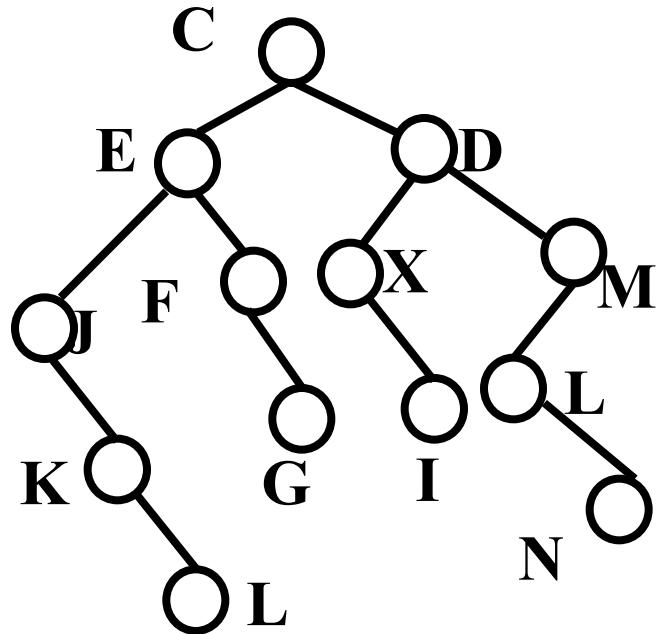
```
template<class T> class TreeNode {           // 树结点的ADT
public:
    TreeNode(const T& value);                // 拷贝构造函数
    virtual ~TreeNode() {};
    bool isLeaf();                            // 判断当前结点是否为叶结点
    T Value();                               // 返回结点的值
    TreeNode<T> *LeftMostChild();           // 返回第一个左孩子
    TreeNode<T> *RightSibling();            // 返回右兄弟
    void setValue(const T& value);           // 设置当前结点的值
    void setChild(TreeNode<T> *pointer);     // 设置左孩子
    void setSibling(TreeNode<T> *pointer);    // 设置右兄弟
    void InsertFirst(TreeNode<T> *node);      // 以第一个左子身份插入结点
    void InsertNext(TreeNode<T> *node);       // 以右兄弟的身份插入结点
};
```

# 「树的抽象数据类型

```
template<class T> class Tree {  
public:  
    Tree();                                // 构造函数  
    virtual ~Tree();                         // 析构函数  
    TreeNode<T>* getRoot();                 // 返回树中的根结点  
    void CreateRoot(const T& rootValue);     // 创建值为rootValue的根结点  
    bool isEmpty();                          // 判断是否为空树  
    TreeNode<T>* Parent(TreeNode<T> *current); // 返回父结点  
    TreeNode<T>* PrevSibling(TreeNode<T> *current); // 返回前一个兄弟  
    void DeleteSubTree(TreeNode<T> *subroot);   // 删除以subroot子树  
    void RootFirstTraverse(TreeNode<T> *root);    // 先根深度优先遍历树  
    void RootLastTraverse(TreeNode<T> *root);     // 后根深度优先遍历树  
    void WidthTraverse(TreeNode<T> *root);        // 广度优先遍历树  
};
```

# 「树/森林的遍历

- 按某一规律系统地访问树/森林中的所有结点，并使每个结点恰好被访问一次
- 遍历方法：
  - 深度优先遍历
  - 广度优先遍历



# 「深度优先遍历森林」

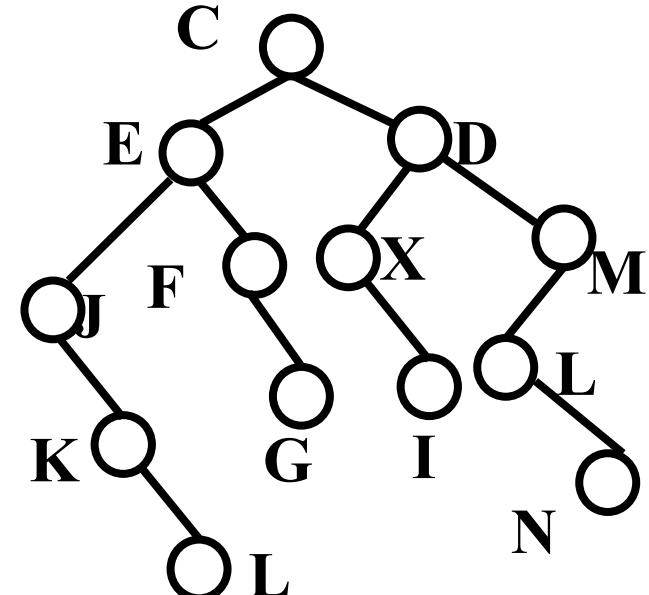
## ■ 先根次序

- a) 访问第一棵树的根
  - b) 在先根次序下遍历第一棵树的子树
  - c) 在先根次序下遍历其他的树
- C,E,J,K,L,F,G,D,X,I,M,L,N

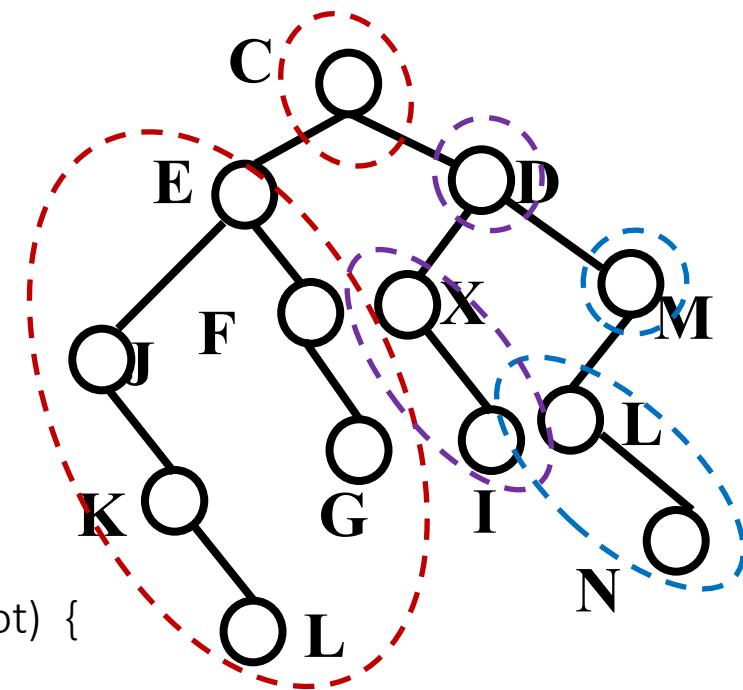
## ■ 后根次序

- a) 在后根次序下遍历第一棵树的子树
- b) 访问第一棵树的根
- c) 在后根次序下遍历其他的树

L,K,J,G,F,E,I,X,N,L,M,D,C



# 先根深度优先遍历森林

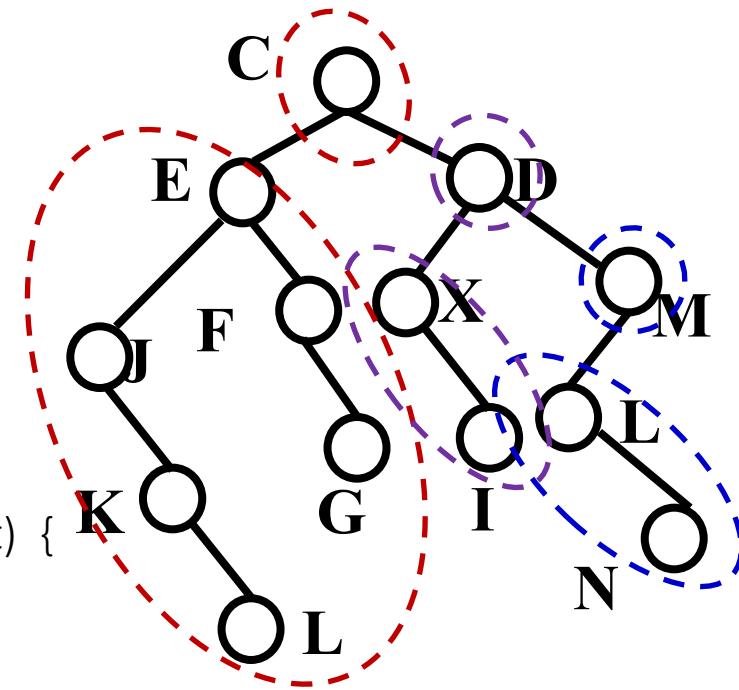


```
template<class T>
void Tree<T>::RootFirstTraverse(TreeNode<T> * root) {
    while (root != NULL) {
        Visit(root->Value());           // 访问当前结点
        // 遍历第一棵树根的子树森林
        RootFirstTTraverse(root->LeftMostChild());
        root = root->RightSibling();   // 遍历其他树
    }
}
```

C,E,J,K,L,F,G,D,X,I,M,L,N

# 后根深度优先遍历森林

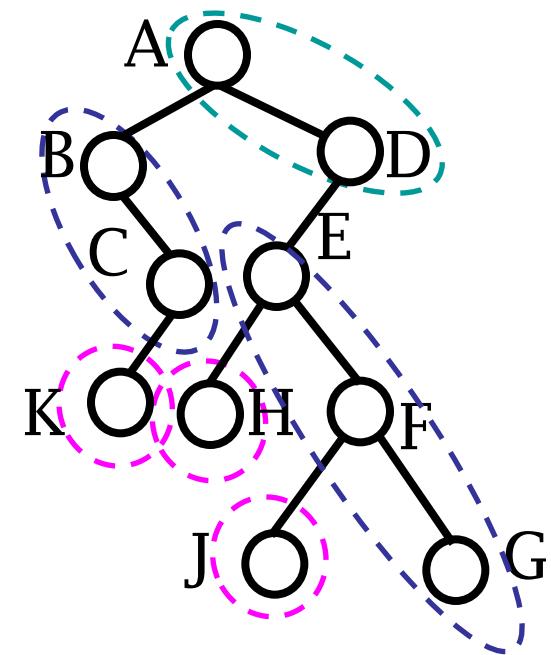
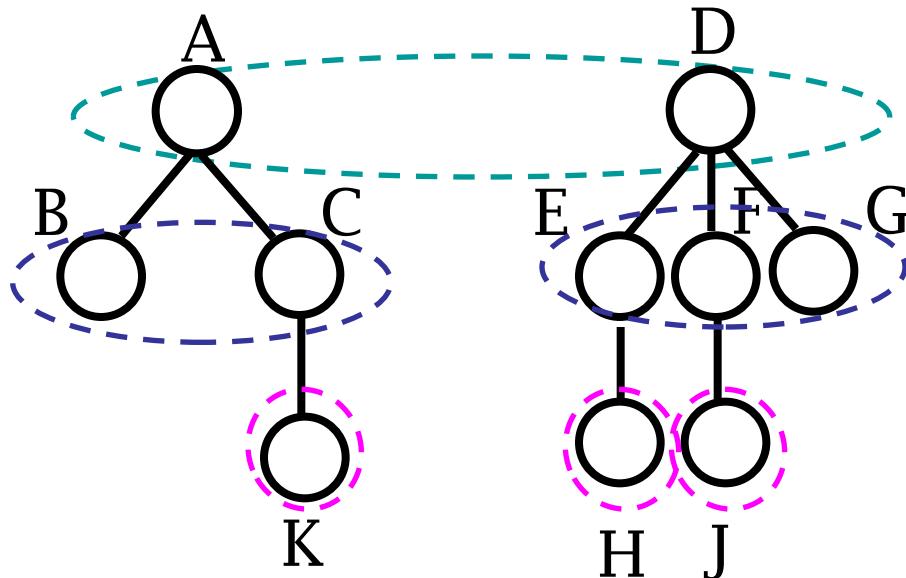
```
template<class T>
void Tree<T>::RootLastTraverse(TreeNode<T> * root) {
    while (root != NULL) {
        // 遍历第一棵树根的子树森林
        RootLastTraverse(root->LeftMostChild());
        Visit(root->Value());           // 访问当前结点
        root = root->RightSibling();    // 遍历其他树
    }
}
```



L, K, J, G, F, E, I, X, N, L, M, D, C

# 按广度遍历树/森林

- 先访问层数为0的结点，然后从左到右逐个访问层数为1的结点，依此类推，直到访问完树中的全部结点



森林广度优先得到层次序列：A D B C E F G K H J

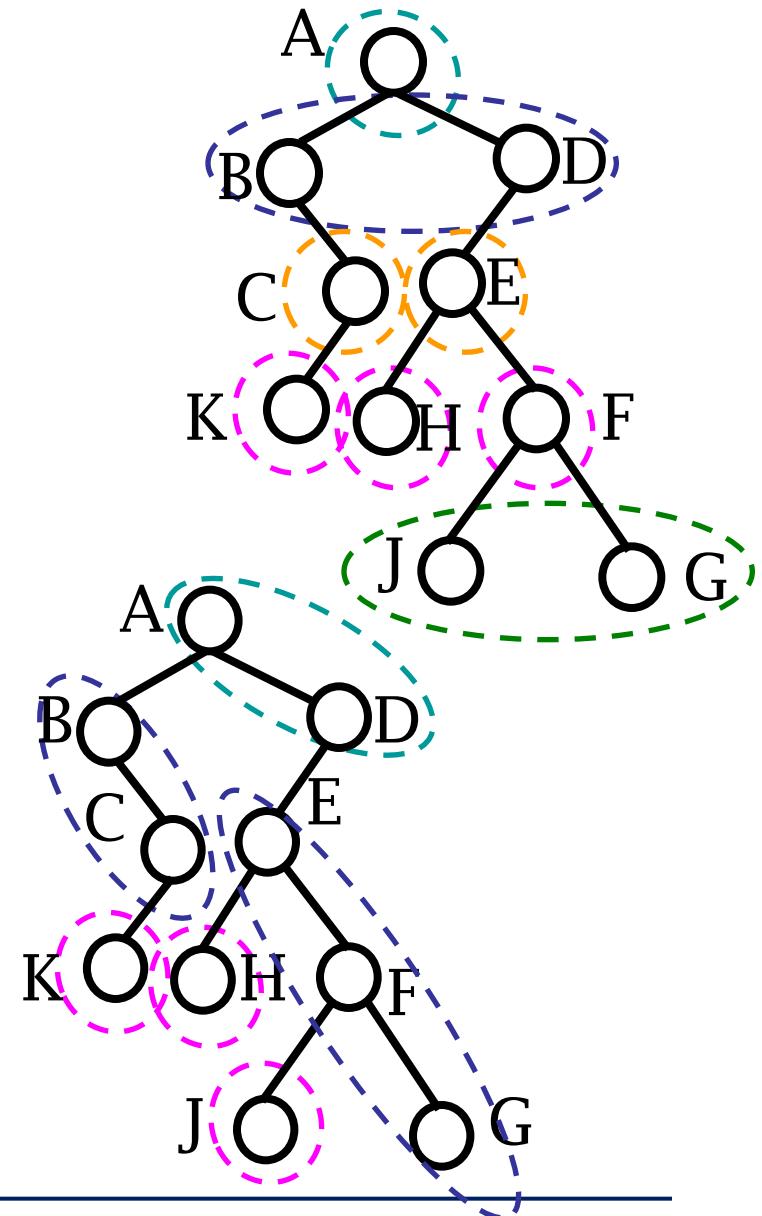
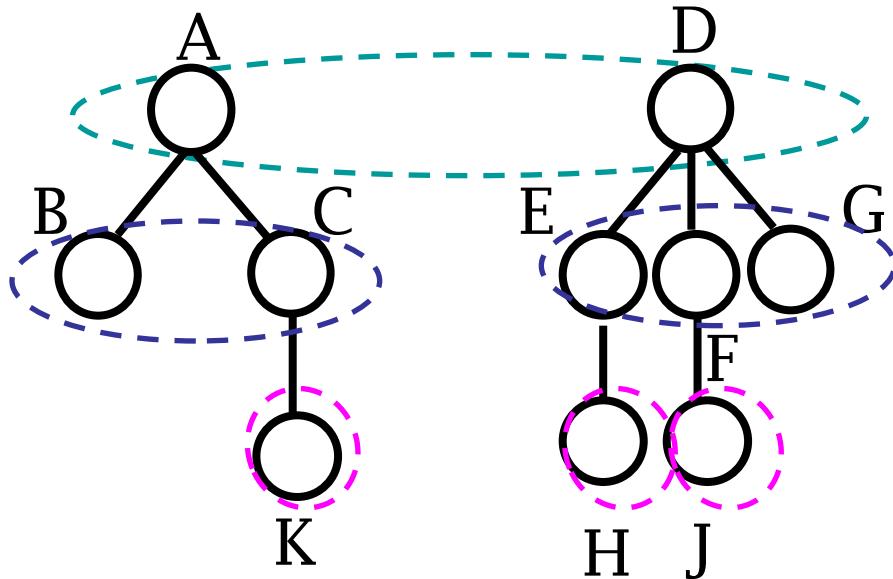
二叉链存储结构的右斜线

# 广度优先遍历森林

```
template<class T>
void Tree<T>::WidthTraverse(TreeNode<T> * root) {
    using std::queue;                                // 使用STL队列
    queue<TreeNode<T>*> aQueue;
    TreeNode<T> * pointer = root;
    while (pointer != NULL) {                         // 当前结点进入队列
        aQueue.push(pointer);
        pointer = pointer->RightSibling();           // pointer指向右兄弟
    }
    while (!aQueue.empty()) {                          // 获得队首元素
        pointer = aQueue.front();                     // 当前结点出队列
        aQueue.pop();
        Visit(pointer->Value());                   // 访问当前结点
        pointer = pointer->LeftMostChild();          // pointer指向最左孩子
        while (pointer != NULL) {                     // 当前结点的子结点进队列
            aQueue.push(pointer);
            pointer = pointer->RightSibling();
        }
    }
}
```

# 思考

- 广度遍历树/森林的各种观点



# 思考

- 树/森林的先根和后根次序遍历与二叉树的哪种遍历次序同？
- 可否定义树/森林的中根次序遍历？
- 森林的非递归深搜框架？
- 树/森林的宽度遍历次序与其对应的二叉树的宽度遍历宽度次序遍历是否相同？

# 「深度优先周游森林」

- 按先根次序遍历森林正好等同于按前序法遍历对应的二叉树
- 按后根次序遍历森林正好等同于按中序法遍历对应的二叉树

不方便仿照中序法定义树的中根遍历，因为当一个根多于两个子结点时无法明确给出根与这些子结点的次序

# 树的存储

# 「树的存储表示

- 链式存储
- 顺序存储

# 「树的存储表示

- 二叉树的链式表示可扩展到树上，几种选择：
  - 为每个结点分配统一大小的数组来存放指针，限制结点可拥有的子结点数目
  - 根据结点当前所拥有的子结点数目为每个结点分配可变长的数组来存放指针
  - 用链表来存放结点的指针
  - 用一个可根据需要动态增长的指针向量

每一种方法都各有千秋，也都存在各自的问题

# 「树的链式存储

- 子结点表表示法 (list of children)
- 左子/右兄结点表示法
- 动态结点表示法
- 动态“左子/右兄”二叉链表表示法
- 父指针表示法

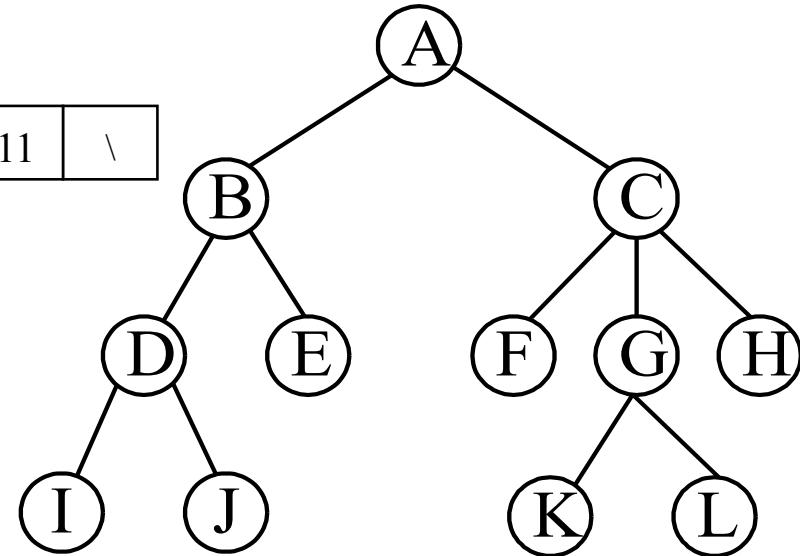
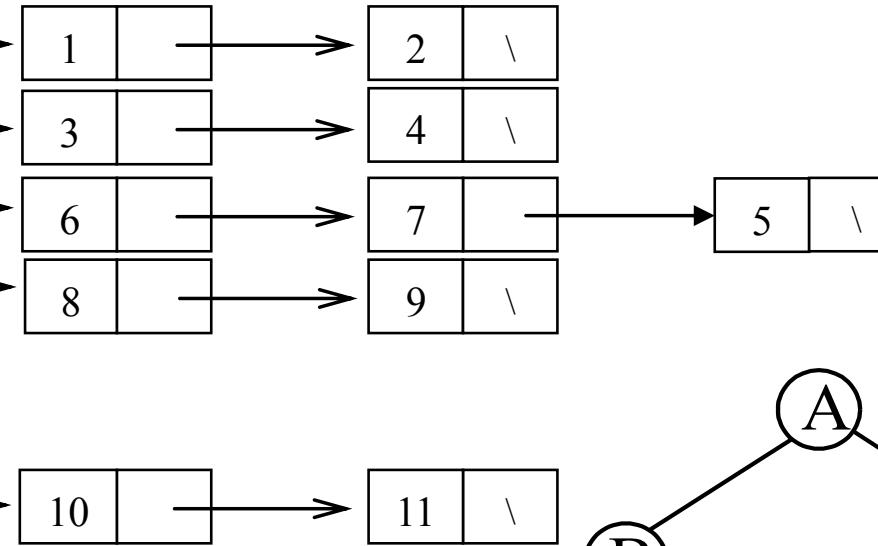
# 「子结点表表示法」

- 每个分支结点均存储其**子结点**信息，子结点按照**从左至右的顺序**形成一个链表
- 采用数组存储树的所有结点，每个数组元素包括
  - 结点值
  - 父指针
  - 子结点链表的指针
- 运算
  - 查找结点的最左子结点，简单
  - 查找给定结点的右兄，较为费时

# 子结点表表示法

索引 值 父结点 子结点

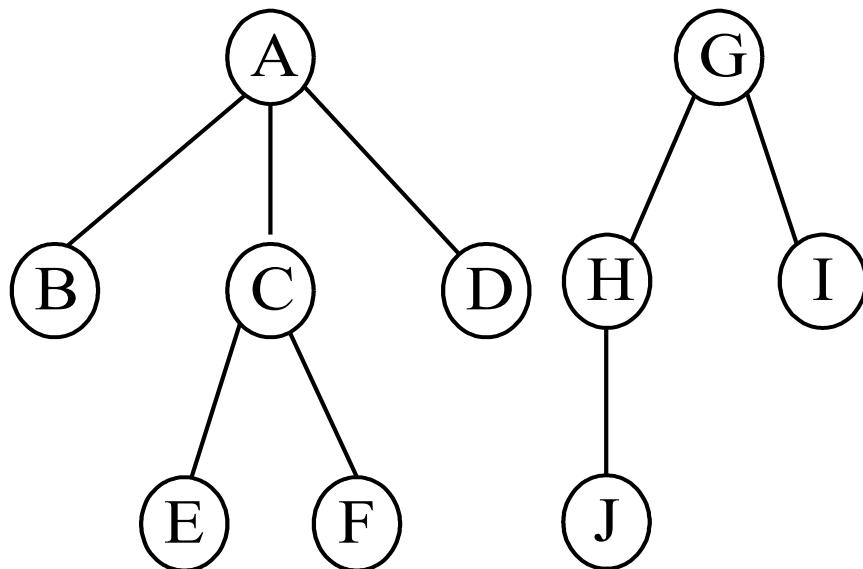
索引	值	父结点	子结点
0	A	\	
1	B	0	
2	C	0	
3	D	1	
4	E	1	\
5	F	2	\
6	G	2	
7	H	2	\
8	I	3	\
9	J	3	\
10	K	6	\
11	L	6	\



# 「左子/右兄表示法

- 每个结点存储内容：
  - 结点值
  - 三个指针，分别指向
    - ◆ 父结点
    - ◆ 最左子结点
    - ◆ 右兄弟结点

# 「左子/右兄表示法」



左子 结点	值	父 结点	右兄弟 结点
	A	/	\
\	B	0	
	C	0	
\	D	0	\
\	E	2	
\	F	2	\
	G	/	\
	H	6	
\	I	6	\
\	J	7	\

# 「左子/右兄表示法」

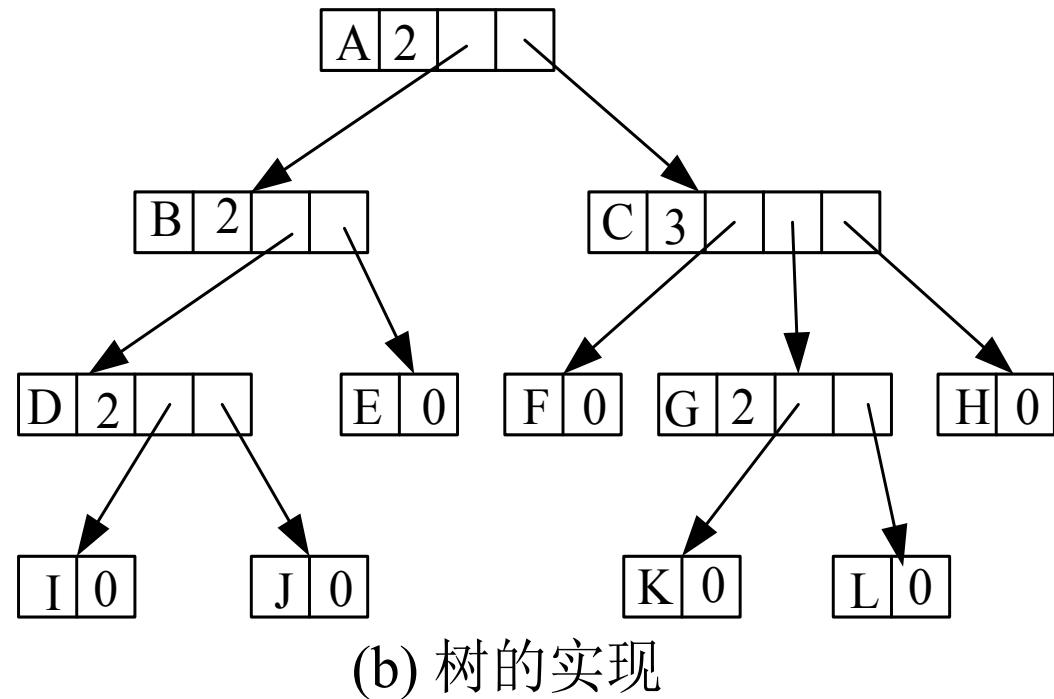
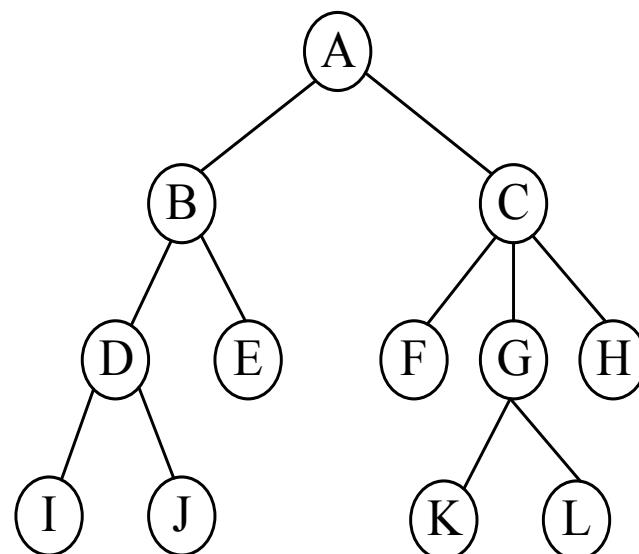
- 运算及ADT
  - 基本操作可通过读取结点中的值来实现
  - 合并两个树
    - ◆ 若两棵树存储在同一个数组中，把其中一个添加为另一棵树的子树，只需简单设置三个指针值即可
- 与子结点表表示法相比，空间效率更高，且结点数组中的每个元素的大小相同而固定

# 「动态结点表示法

- 为每个结点分配可变的存储空间
  - 指向子结点的指针数组作为结点的一部分分配给结点
    - ◆ 每个结点存储一个基于数组的子结点指针表。在子结点的数目不变时，效果最佳；若子结点的数目发生变动（特别是增加），就须提供一种专门的校正机制来改变子结点指针数组的大小
  - 每个结点存储一条子结点指针链表
    - ◆ 本质上与“子结点表”表示法相同，但动态地分配结点空间，而不是把结点分配在数组中

# 动态结点表示法

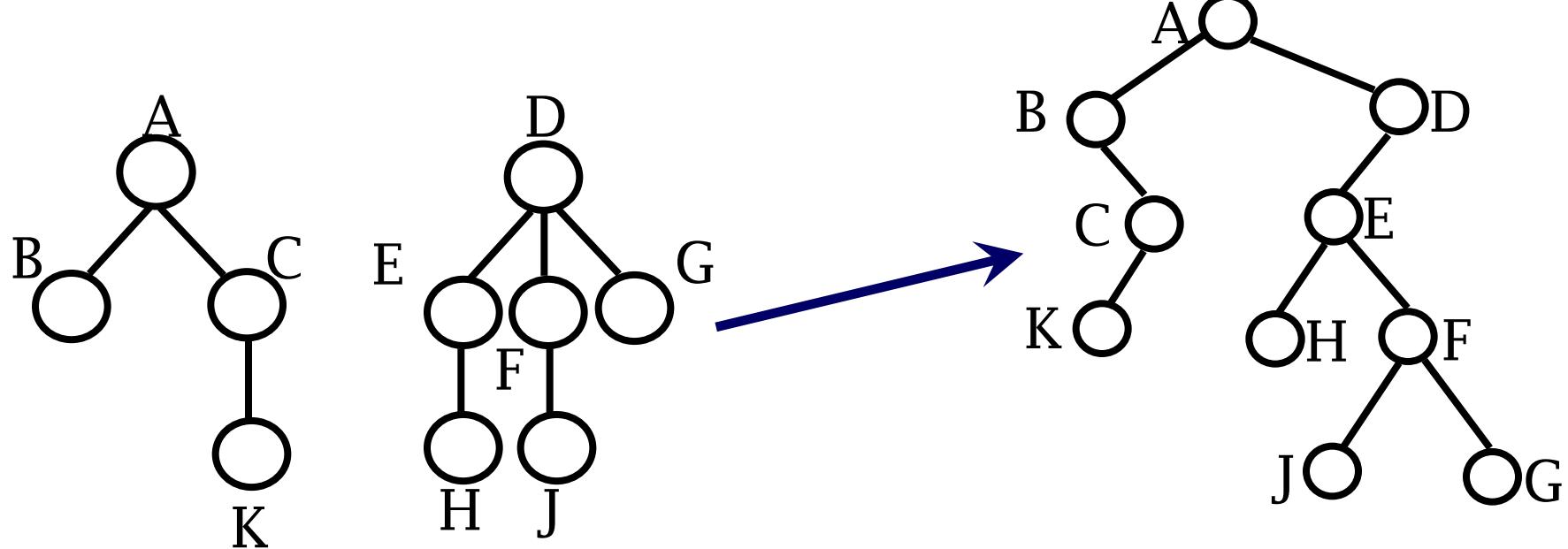
- 结点数目发生变化，需要重新分配存储空间



# 「动态“左子/右兄”表示法

- 本质上，可用**二叉树**来表示树
  - 对应的二叉树某结点的**左子结点**为树中该结点的**最左子结点**；**右子结点**是结点原来的**右兄结点**
  - 根的右链就是森林中每棵树的根结点，因森林中每棵树的根结点可以看成互为兄弟结点
- 由于二叉树的每个结点均包含**固定数目的指针**，而且树的ADT的每个函数均能有效实现，故**动态“左子/右兄”表示法**比的其他几种方法更常用

# 「动态“左子/右兄”表示法」表示法



# 「树结点ADT的实现」

```
// 补充与具体实现相关的私有成员变量申明
private:
    T           m_Value;           // 树结点的值
    TreeNode<T>*      pChild;     // 左子结点
    TreeNode<T>*      pSibling;    // 右兄结点

// 公有成员函数的具体实现
template<class T>TreeNode<T>::TreeNode(const T& value) {

    // 拷贝构造函数
    m_Value = value;
    pChild = NULL;
    pSibling = NULL;
}
```

# 「树结点ADT的实现

```
template<class T>
T TreeNode<T>::Value() {
    // 返回结点的值
    return m_Value;
}
```

```
template<class T>
bool TreeNode<T>::isLeaf() {
    // 如果结点是叶，返回true
    if (pChild == NULL)
        return true;
    return false;
}
```

# 「树结点ADT的实现

```
template<class T>
TreeNode<T> *TreeNode<T>::LeftMostChild() {           // 返回第一个左子结点
    return pChild;
}

template<class T>
TreeNode<T> * TreeNode<T>::RightSibling() {           // 返回右兄弟
    return pSibling;
}

template<class T>
void TreeNode<T>::setValue(T& value) {                // 设置结点的值
    m_Value = value;
}
```

# 「树ADT的实现

// 与具体实现相关的私有成员变量与成员函数的申明

private:

    TreeNode<T>\* root;                        // 树根结点

// 返回current的父节点，由函数Parent调用

    TreeNode<T>\* getParent (TreeNode<T>\* root, TreeNode<T>\* current);

// 删除以root为根的子树的所有结点

    void DeleteNodes(TreeNode<T>\*root);

// 私有成员函数与公有成员函数的具体实现

template <class T>C Tree<T>::Tree() {                // 构造函数

    root = NULL;

}

# 「树ADT的实现

// 析构函数

```
template <class T> Tree<T>::~Tree() {
    while (root)
        DeleteSubTree(root);
}
```

// 返回树中的根结点

```
template <class T> TreeNode<T>* Tree<T>::getRoot() {
    return root;
}
```

# 「树ADT的实现」

```
template<class T>
TreeNode<T>* Tree<T>::Parent(TreeNode<T> *current) {
    using std::queue;
    queue<TreeNode<T*>> aQueue;
    TreeNode<T> *pointer = root;
    TreeNode<T> *father = upperlevelpointer = NULL;
    if (current != NULL && pointer != current) {
        while (pointer != NULL) {
            if (current == pointer)
                break;
            aQueue.push(pointer);
            pointer=pointer-> RightSibling();
        }
        while (!aQueue.empty()) {
            pointer = aQueue.front();
            aQueue.pop();
            upperlevelpointer = pointer;
            pointer = pointer-> LeftMostChild();
            while (pointer) {
                if (current == pointer) {
                    father = upperlevelpointer;
                    break; }
                else {
                    aQueue.push(pointer);
                    pointer = pointer->RightSibling(); }
            }
        }
    }
    aQueue. clear( );
    return father;
}
```

// 使用STL队列  
// 记录父结点  
// 森林中所有根结点进队列  
// 森林中所有第一层根的父为空  
// 当前结点进队列  
// 指针指向右  
// 取队列首结点指针  
// 当前元素出队列  
// 指向上一层的结点  
// 指向最左孩子  
// 当前结点的子结点进队列  
// 返回父结点  
// 清空队列，也可以不写（局部变量）

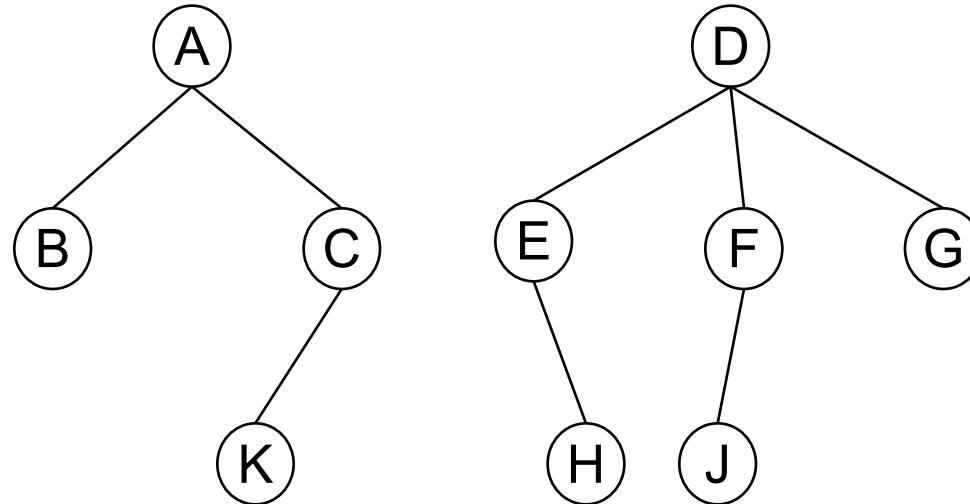
# 「删除以root为根的子树」

```
template <class T>
void Tree<T>::DestroyNodes(TreeNode<T>* root) {
    if (root) {
        DestroyNodes(root->LeftMostChild()); // 递归删除第一子树
        DestroyNodes(root->RightSibling()); // 递归删除其他子树
        delete root; // 删除根结点
    }
}
```

# 「父指针表示法」

- 实现树的一个最简单方法是每个结点只保存一个指向父结点的指针域，即，父指针(parent pointer)表示法
- 运算
  - 查询结点的根
    - ◆ 从一个结点出发找出一条向上到达根的祖先路径
    - ◆  $O(k)$ ,  $k$ 为树高
  - 判断两个结点是否在同一棵树中
    - ◆ 若两个结点沿父指针链到达同一根结点，则它们必在同一棵树中
    - ◆ 否则，若找到的根结点不同，两个结点则不在同一棵树中

# 「父指针数组表示法」



父节点索引  
标记  
结点索引

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

# 父指针表示法

## ■ 优点

- 易于查找父结点及其祖先结点
- 比较节省存储空间
- 易于两棵树的合并

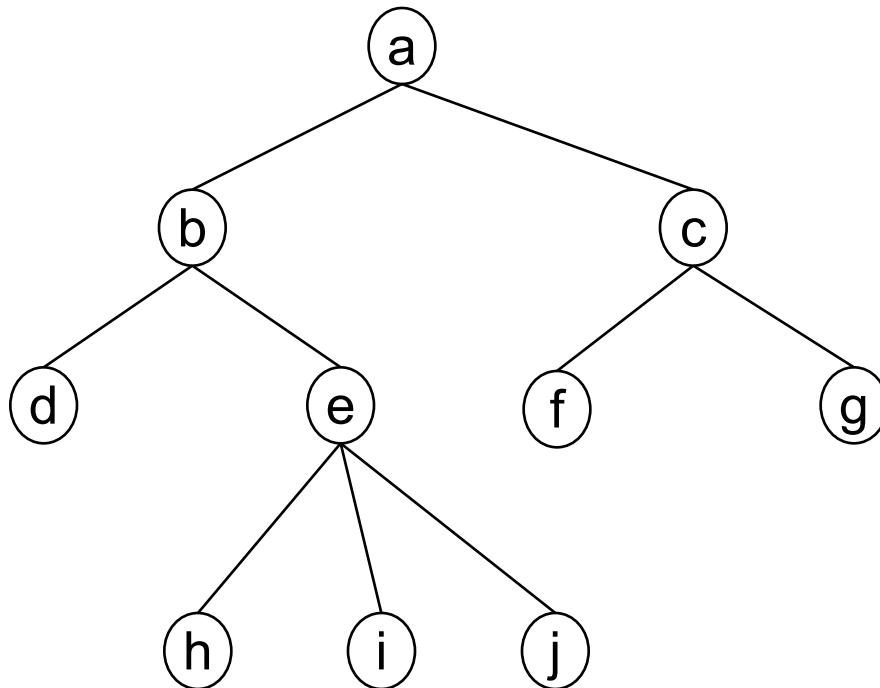
## ■ 缺点

- 查找结点的子结点和兄弟结点较费事（遍查整个数组）
- 难以表示结点之间的**左右次序**（适于**无序树**）

# 「父指针表示法的改进」

结点按某种遍历次序存放于数组中

常见的一种方法是依次存放树的先根序列



	info	parent
0	a	-1
1	b	0
2	d	1
3	e	1
4	h	3
5	i	3
6	j	3
7	c	0
8	f	7
9	g	7

# 「父指针表示法求兄弟结点位置

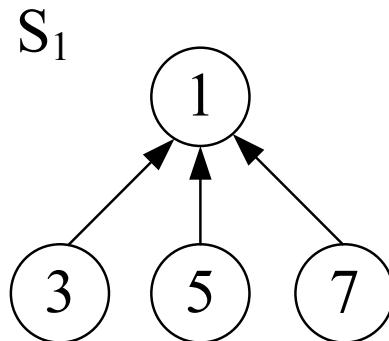
```
int rightSibling_partree(PParTree t, int p)
    int i;
    if (p >= 0 && p < t->n) {
        for (i=p+1; i<t->n; i++)
            if (t->nodelist[i].parent==t->nodelist[p].parent)
                return(i);
    }
    return(-1);
}
```

# 「等价关系」

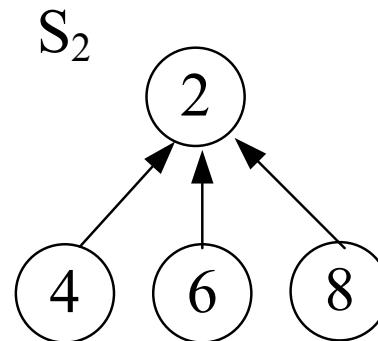
- 一个具有  $n$  个元素的集合  $S$ , 一个定义在集合  $S$  上的  $r$  个关系的关系集合  $R$ 
  - $x, y, z$  表示集合中的元素
- 关系  $R$  为等价关系, 当且仅当如下条件为真:
  - 对于所有的  $x$ , 有  $(x, x) \in R$  (自反)
  - 当且仅当  $(x, y) \in R$  时,  $(y, x) \in R$  (对称)
  - 若  $(x, y) \in R$  且  $(y, z) \in R$ , 则有  $(x, z) \in R$  (传递)
- 若  $(x, y) \in R$ , 则元素  $x$  和  $y$  是等价的

# 等价类 (equivalence classes)

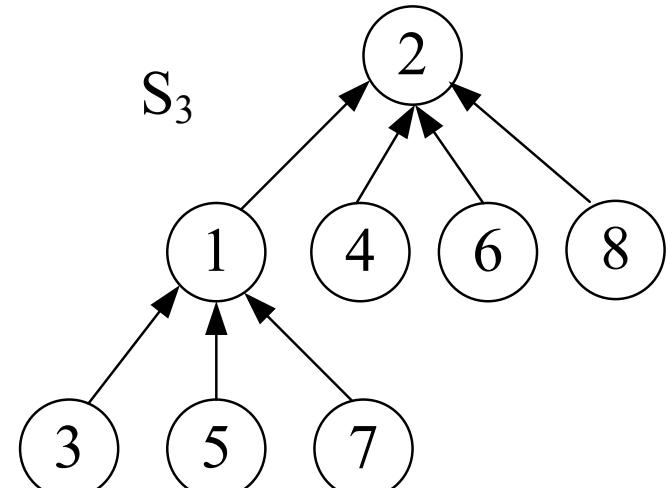
- 相互等价的元素组成的**最大集合**（即，不存在类以外的元素与类内的元素等价）
  - 等价关系具有**自反性、对称性、传递性**
  - 应用：电路板上元件间的线路连接，道路的连接等



(a)



(b)



(c)

# 「划分等价类」

- 需要对集合进行**三种**操作
  - 构造只含有一个元素的集合
  - 判定某个元素所在的子集，以便确定两个给定元素是否在同一个集合之中，即
    - ◆ 搜索包含该元素的等价类，对于同一个集合中的元素返回相同的结果，否则返回不同的结果
  - 归并两个不相交的集合为一个集合

# 「等价类的并查算法」

- 父指针表示法常用来维护由一些不相交子集构成的集合。包含两种基本操作：
  - 判断两个结点是否在同一个集合中，即查找一个给定结点的根结点的过程称为 **Find**
  - 归并两个集合，这个归并过程被称为 **Union**

整个操作以“**Union/Find算法**”（中文以“**并查算法**”）命名

# 「并查集」

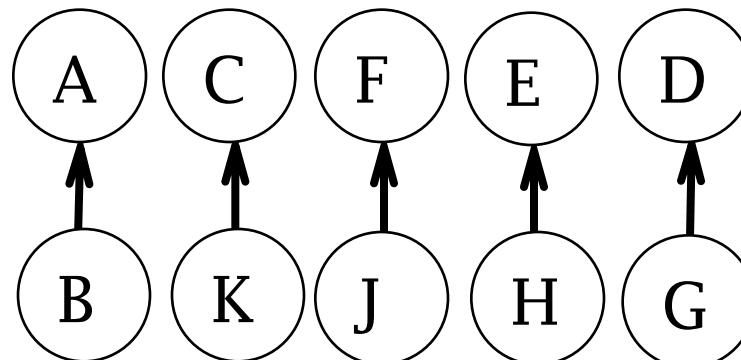
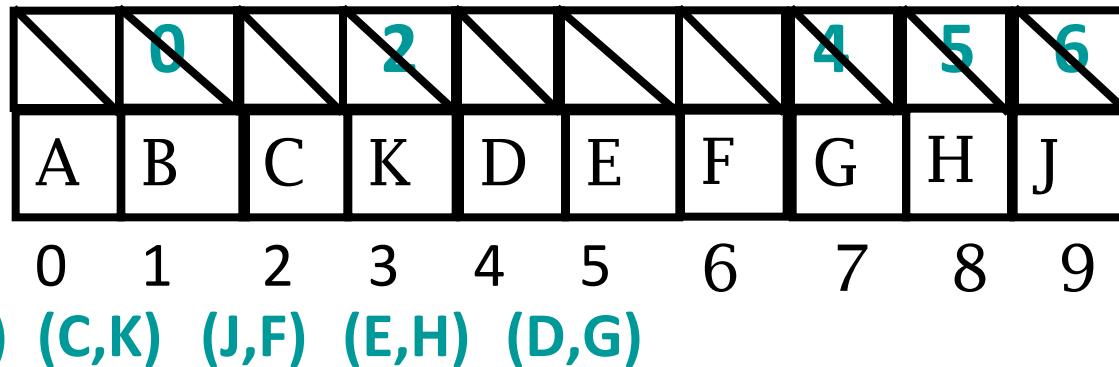
- 一种特殊的集合，由一些不相交子集构成，基本操作包括：
  - `Find`：判断两个结点是否在同一个集合中
  - `Union`：归并两个集合
- 像栈、队列一样，并查集也是一种重要的**抽象数据类型**，多用于求解**等价类问题**

# Union/Find 算法示例

- 10个结点  
 $\{A、B、C、D、E、F、G、H、J、K\}$
- 结点间的等价关系
  - (A, B)
  - (C, K)
  - (J, F)
  - (H, E)
  - (D, G)
  - (K, A)
  - (E, G)
  - (H, J)

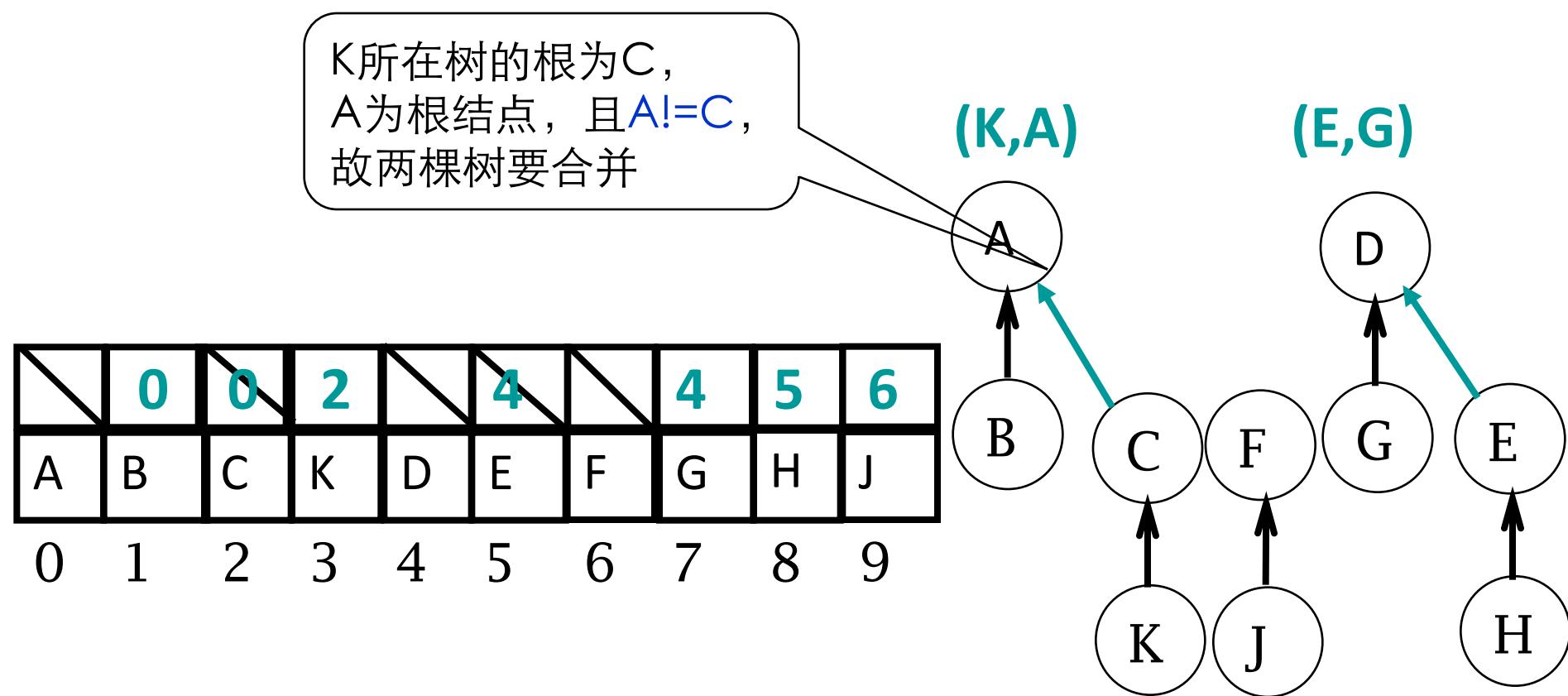
# Union/Find算法示例

- 先对5个等价对  $(A,B)$ 、 $(C,K)$ 、 $(J,F)$ 、 $(H,E)$ 、 $(D,G)$  依次处理



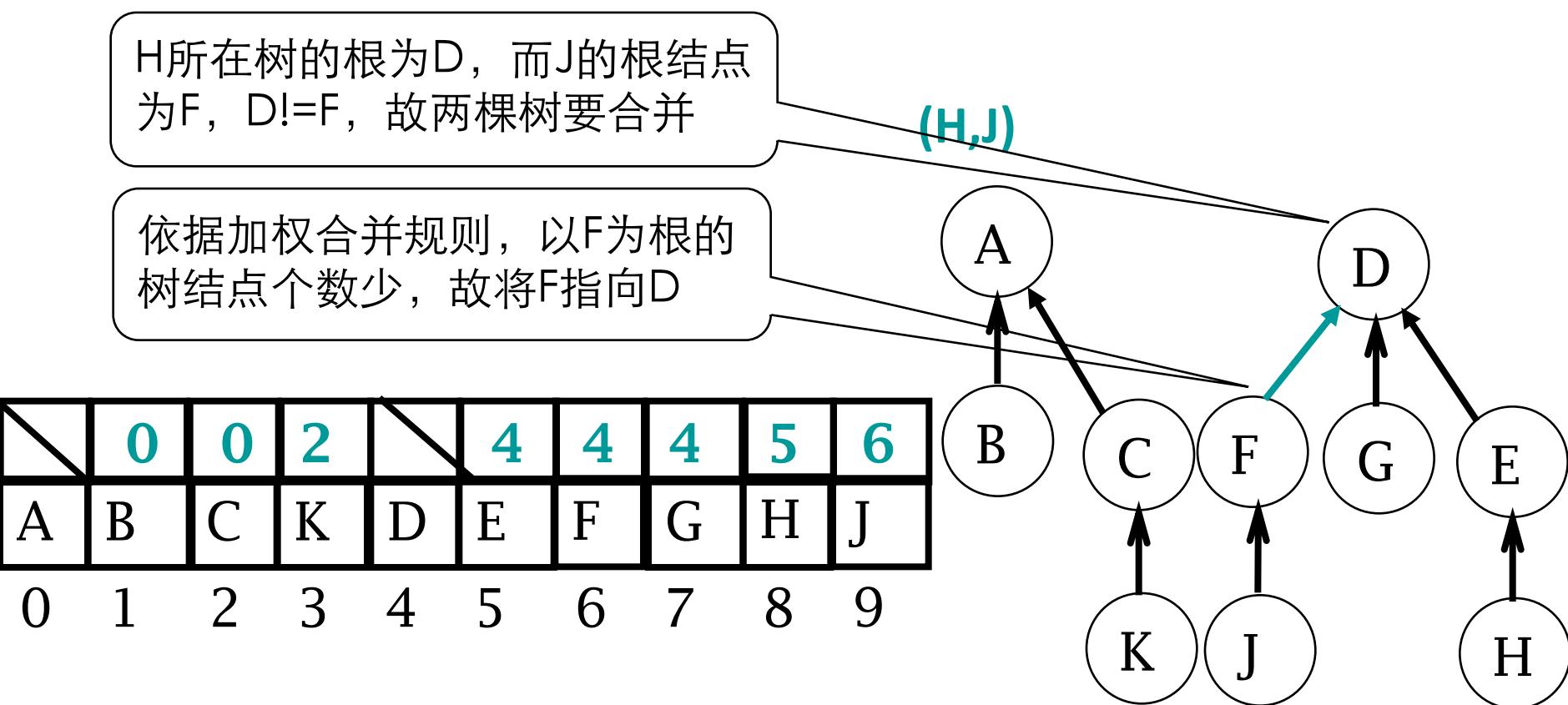
# Union/Find 算法示例

- 再对等价对  $(K, A)$  和  $(E, G)$  进行处理



# Union/Find 算法示例

- 最后使用加权合并规则处理等价对 (H,J)



# 「父指针表示与并查算法实现」

```
template<class T>
class ParTree {                                // 树定义
public:
    ParTreeNode<T>* array;                    // 存储树结点的数组
    int Size;                                  // 数组大小
    ParTreeNode<T>*
    Find(ParTreeNode<T>* node) const;          // 查找node结点的根结点
    ParTree(const int size);                   // 构造函数
    virtual ~ParTree();                        // 析构函数
    void Union(int i,int j);                  // 把下标为i, j的结点合并成一棵子树
    bool Different(int i,int j);              // 判定下标为i, j的结点是否在一棵树中
};
```

# 「父指针表示与并查算法实现

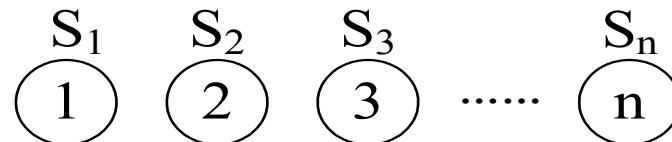
```
template <class T>
ParTreeNode<T>*
ParTree<T>::Find(ParTreeNode<T>* node) const
{
    ParTreeNode<T>* pointer=node;
    while ( pointer->getParent() != NULL )
        pointer=pointer->getParent();
    return pointer;
}
```

# 「父指针表示与并查算法实现」

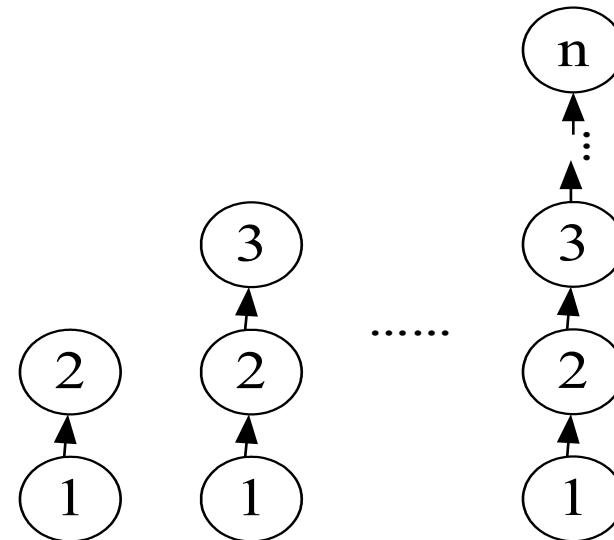
```
template<class T>
void ParTree<T>::Union(int i,int j) {
    ParTreeNode<T>* pointeri = Find(&array[i]);           // 找到结点i的根
    ParTreeNode<T>* pointerj = Find(&array[j]);           // 找到结点j的根
    if (pointeri != pointerj) {
        if(pointeri->getCount() >= pointerj->getCount()) {
            pointerj->setParent(pointeri);
            pointeri->setCount(pointeri->getCount() +
                                  pointerj->getCount());
        }
        else {
            pointeri->setParent(pointerj);
            pointerj->setCount(pointeri->getCount() +
                                  pointerj->getCount());
        }
    }
}
```

# Union/Find的复杂度

- Find:  $O(h)$ , 与树的高度成正比
- Union:  $O(1)$



(a)  $n$ 个集合



(b) 合并操作

# 「重量权衡合并规则」

## ■ 父指针表示法

- 并不限制共享同一父结点的结点数目
- 为使等价对的处理尽可能高效，每个结点到其相应的根结点的距离应尽可能小

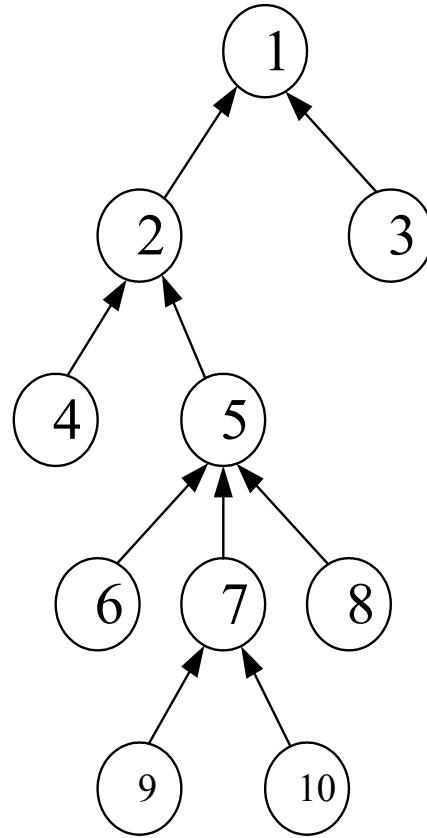
## ■ 重量权衡合并规则 (weighted union rule)

- 将结点较少树的根结点指向结点较多树的根结点。这可将树的整体深度限制在  $O(\log n)$
- 当处理完  $n$  个等价对后，任何结点的深度最多只会增加  $\log n$  次

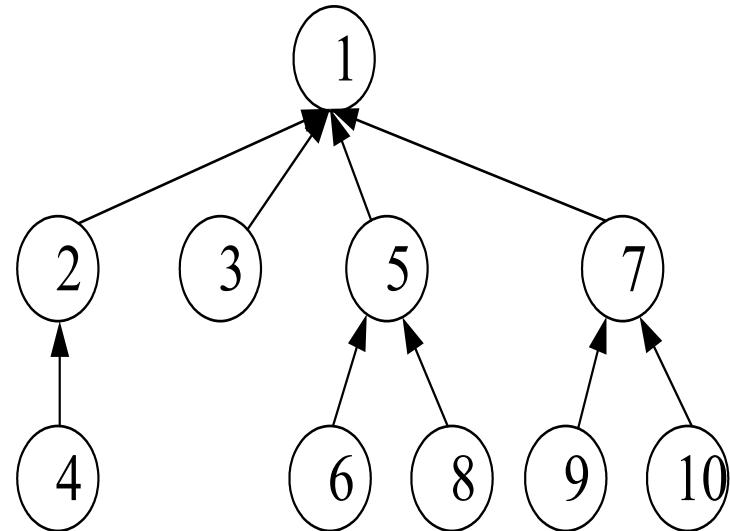
# 「路径压缩 (path compression)

- 一种加速并查集运算的技术，可以产生极浅树
- 当查找一个结点  $X$  的根结点时可以采用路径压缩
  - 设根结点为  $R$ ，则路径压缩把由  $X$  到  $R$  的路径上每个结点的父指针均设置为直接指向  $R$
  - 首先要查找  $R$ ，然后顺着由  $X$  到  $R$  的路径把每个结点的父指针域均设置为指向  $R$

# 路径压缩示例



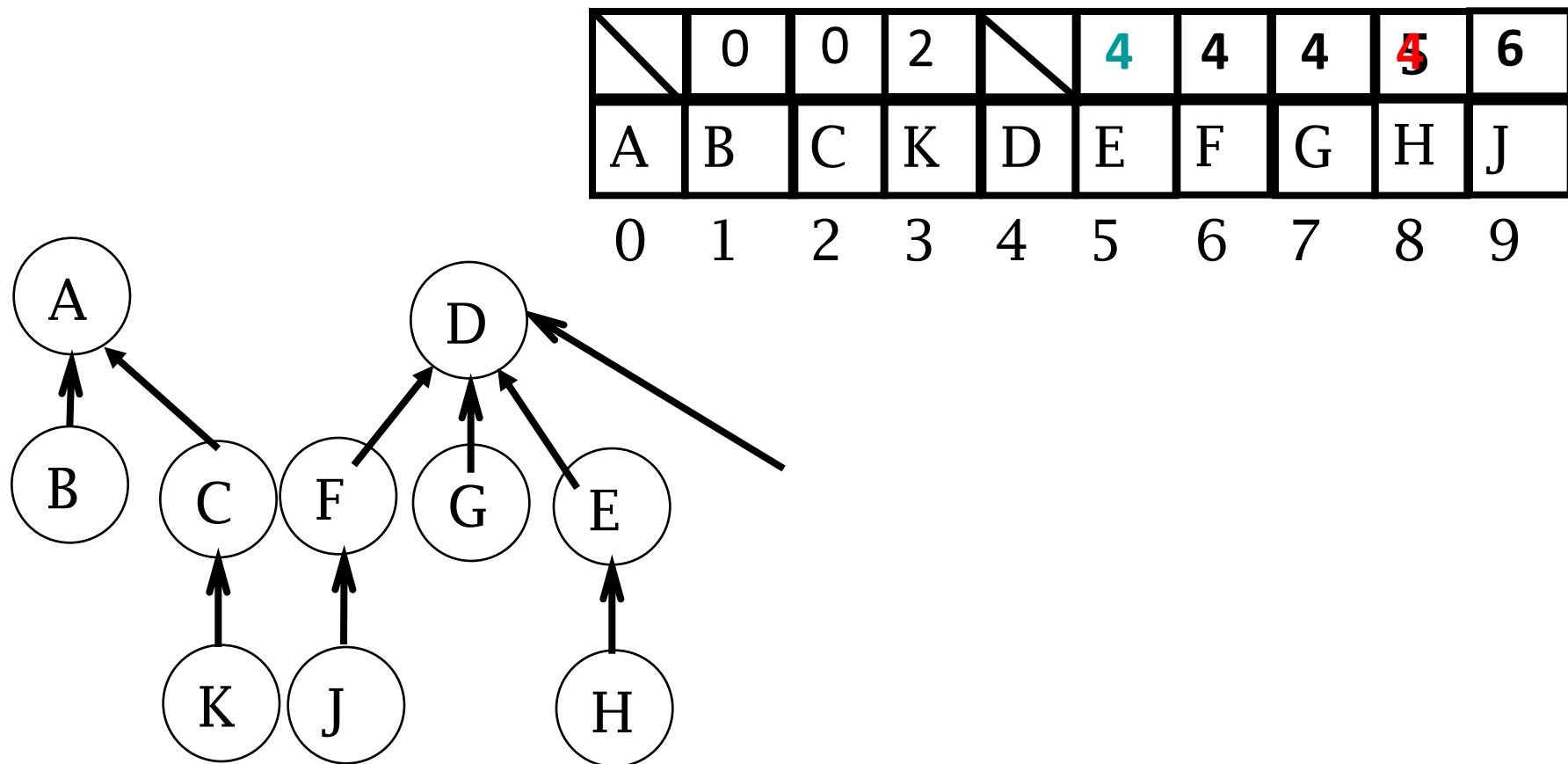
查找结点 7



(b) 路径压缩之后

# 路径压缩示例

- 使用路径压缩规则处理Find(H)



# 带路径压缩的Find

```
template <class T>
ParTreeNode<T>*
ParTree<T>::FindPC(ParTreeNode<T>* node) const {
    if (node->getParent() == NULL)
        return node;
    node->setParent(FindPC(node->getParent()));
    return node->getParent();
}
```

# 「路径压缩的Find代价接近常数」

## ■ 权重 + 路径压缩

- 对  $n$  个结点进行  $n$  次Find操作的开销为  $O(n\alpha(n))$ ， 约为  $\Theta(n \log^* n)$ 
  - ◆  $\alpha(n)$  是单变量Ackermann函数的逆，是一个增长速度比  $\log n$  慢得多但又并非常数的函数
  - ◆  $\log^* n$  是在  $n = \log n \leq 1$  之前要进行的对  $n$  取对数操作的次数
  - ◆  $\log^* 65536 = 4$  (4次log操作)
- Find至多需要一系列  $n$  个Find操作的开销接近于  $\Theta(n)$ 
  - ◆ 在实际应用中，  $\alpha(n)$  往往小于4

# 思考

- 可否使用动态指针方式实现父指针表示法？
- 查阅各种并查集权重和路径压缩优化方法，并讨论各种方法的异同和优劣