树和森林

吉林大学计算机学院
谷方明
fmgu2002@sina.com

学习目标

- 掌握树和森林的存储方式
- 掌握树和森林的遍历
- 掌握树和森林的创建等操作

树的存储结构

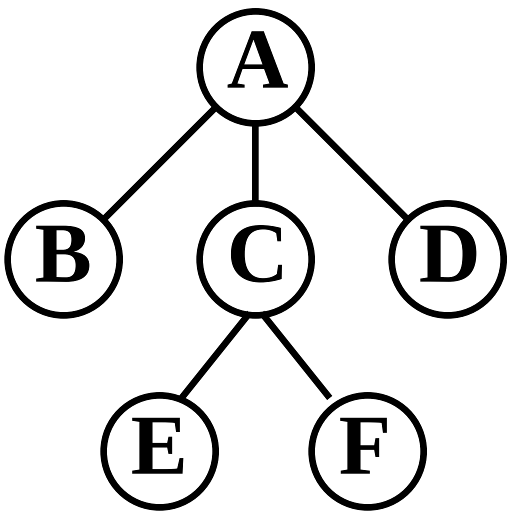
- 顺序存储
  - 类似二叉树

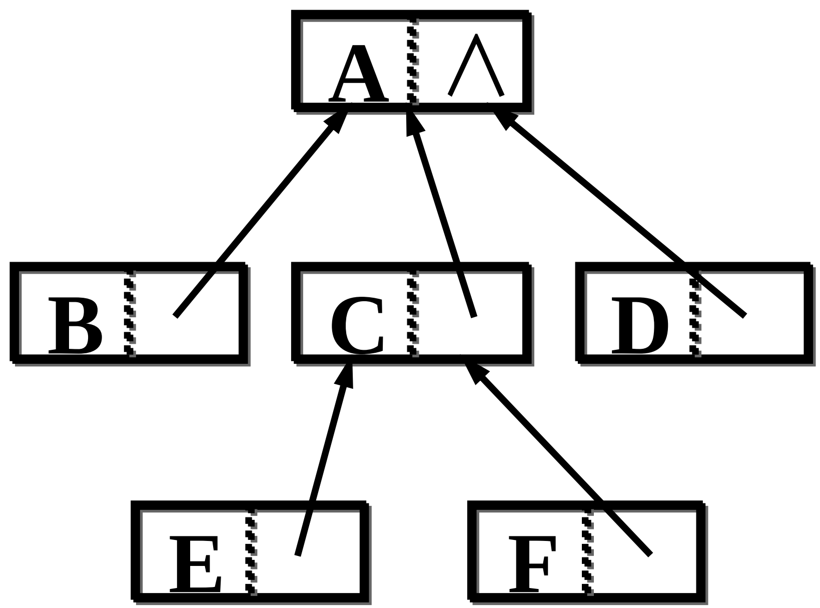
- 链接存储（树的自然表示方法）
  - 双亲链接表示法
  - 孩子链接表示法
  - 孩子兄弟表示法

双亲链接结构

- 双亲链接是有一个指针指向其父结点。简单的双亲链接的结点结构有两个域：**Data**和**Father(**或**Parent)**

Data	**Father**

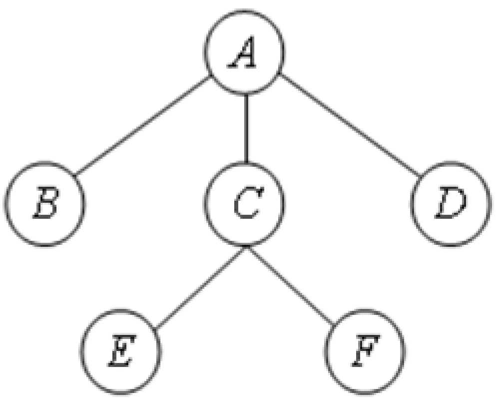




特点

- 简单
- 空间需求小，利用率高
- 双亲链接提供了“向上”访问的能力；
- 但不能利用根作为起始点，很难确定一个结点是否为叶结点，也很难求结点的子结点，且不易实现遍历操作（遍历至少需要叶结点指针集合作为起始点，甚至全部结点的集合）。
- 因此，实现时多采用结构体数组。其静态链表形式，即为Father数组表示法。

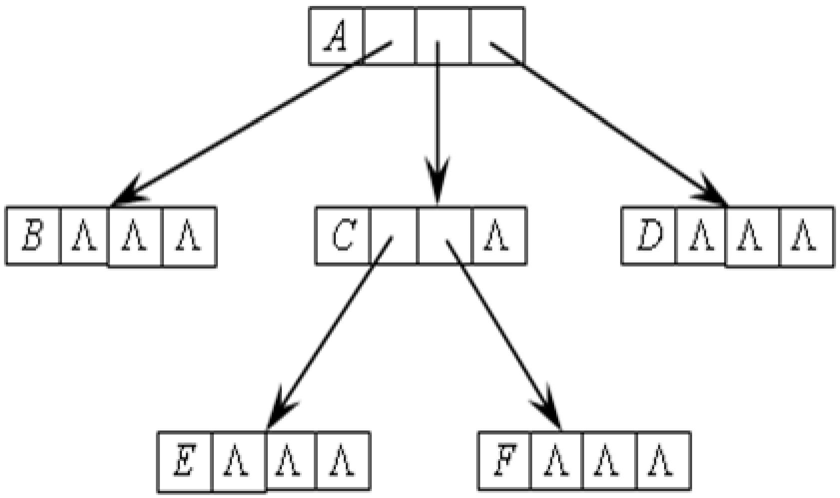
- **Father数组表示法**

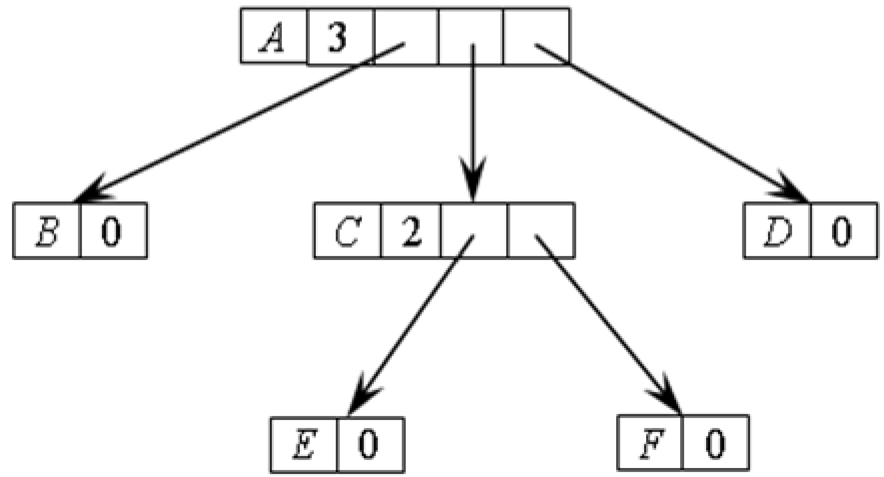


下标	1	2	3	4	5	6
	A	B	C	D	E	F
	0	1	1	1	3	3

- **Father数组中的叶结点：不是任何结点的父亲**

孩子链接结构





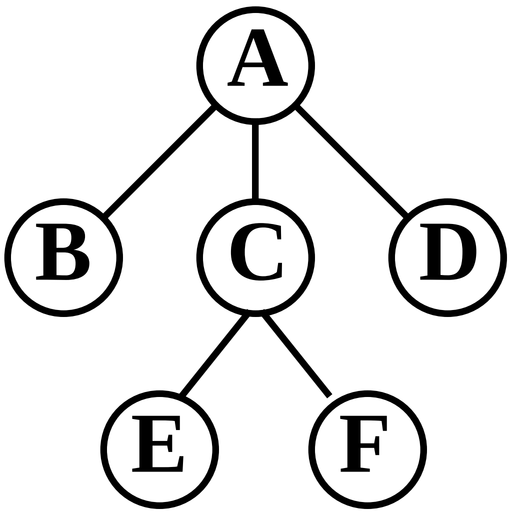
例:结点大小固定

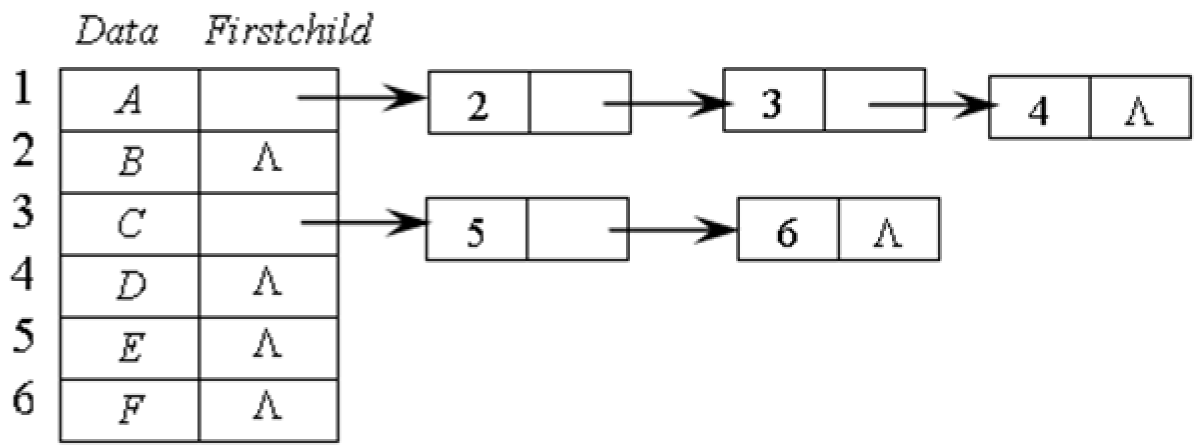
例:结点大小不固定

*会出现大量指针为空的情况，浪费空间。

*节省了空间，但给操作和管理带来不便。

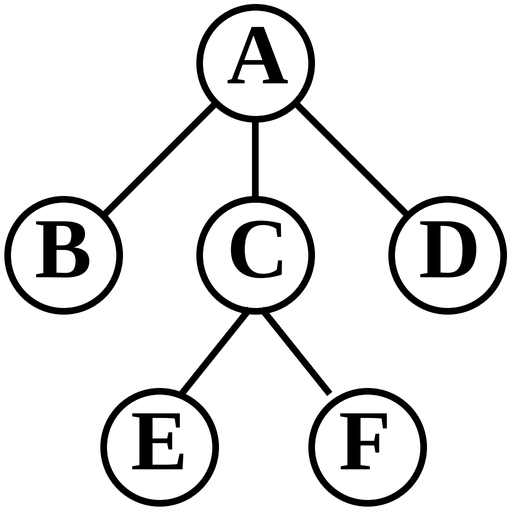
孩子链表表示法

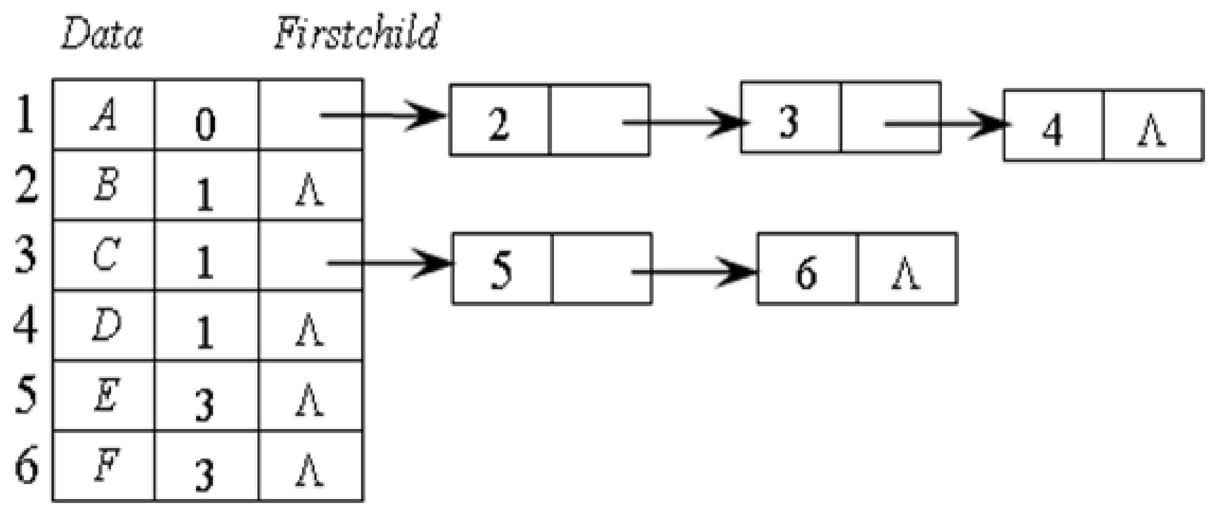




*孩子链表表示法是为树中每个结点设置一个孩子链表，并将这些结点及相应的孩子链表的头指针存放在一个数组中。孩子结点的数据域仅存放了它们在数组空间的下标（见图的邻接链表） 。

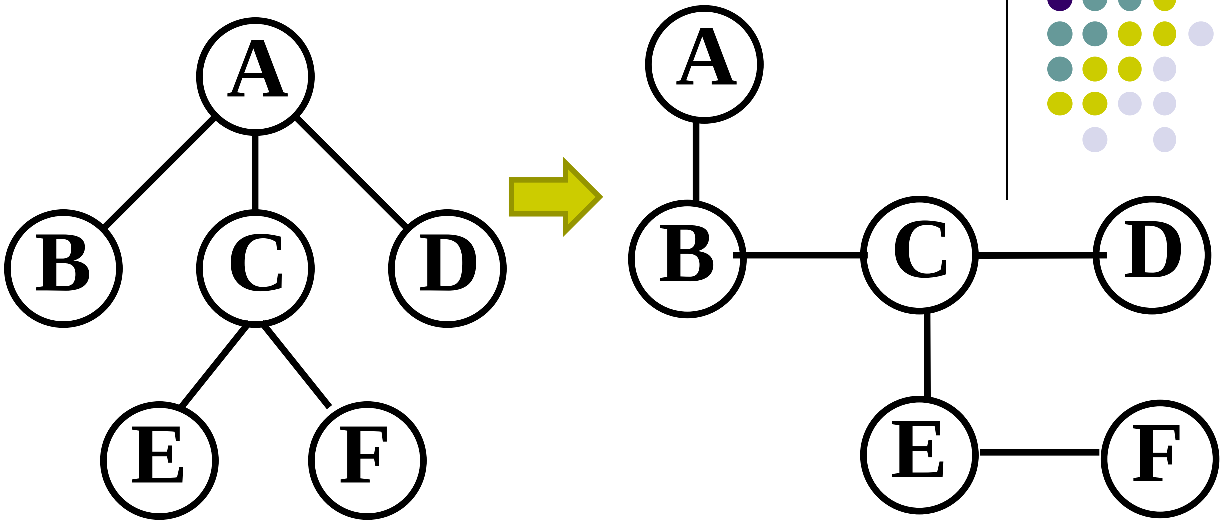
双亲孩子链接表示





*将*Father*数组表示法和孩子链表表示法结合起来，可形成父亲孩子链表表示法。

孩子兄弟表示法



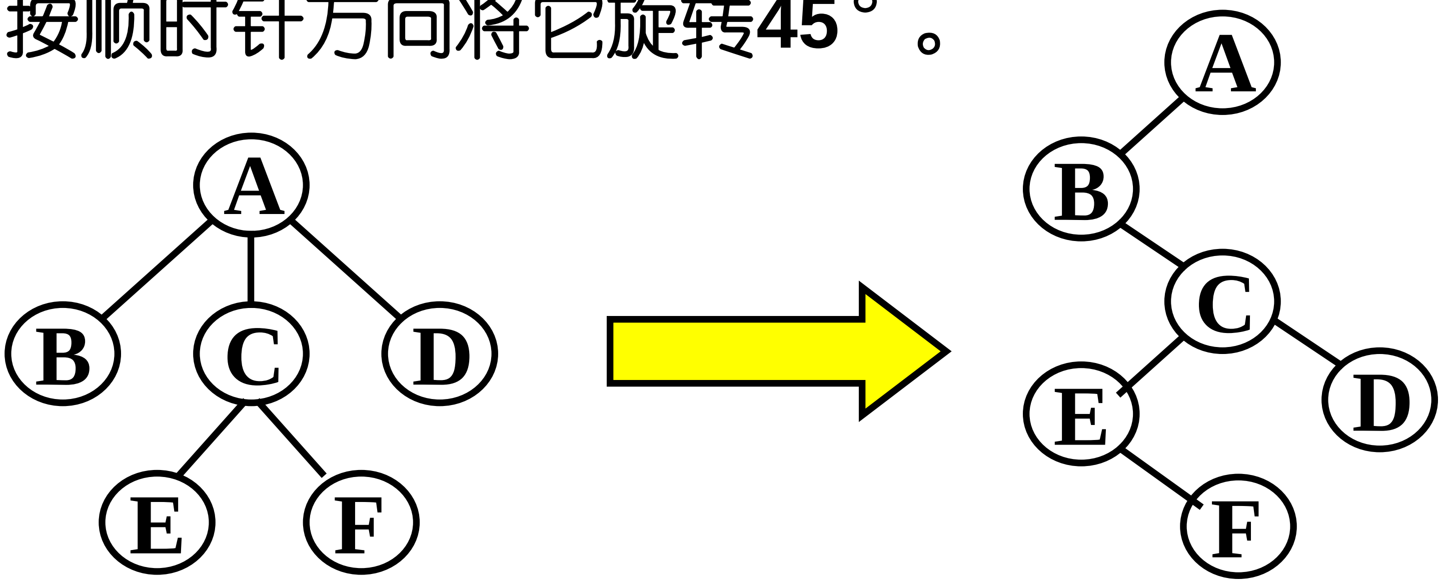
- 要访问一棵树，只需提供垂直访问能力和水平访问能力即可。垂直访问能力即确定结点的某个孩子；水平访问能力即确定结点的兄弟。这样，每个结点只需保留一个孩子和一个兄弟，从而形成了一种二叉链结构，即树的孩子兄弟表示法。
- 树的孩子兄弟表示法将树转化成了二叉树。涉及到森林和二叉树的自然对应

1 树转换成二叉树

① 所有兄弟结点之间加一线；

② 对每个结点，只保留与其长子的连线，去掉该结点与其他孩子的连线。

③ 按顺时针方向将它旋转45°。



编程转换规则（左孩子右兄弟）

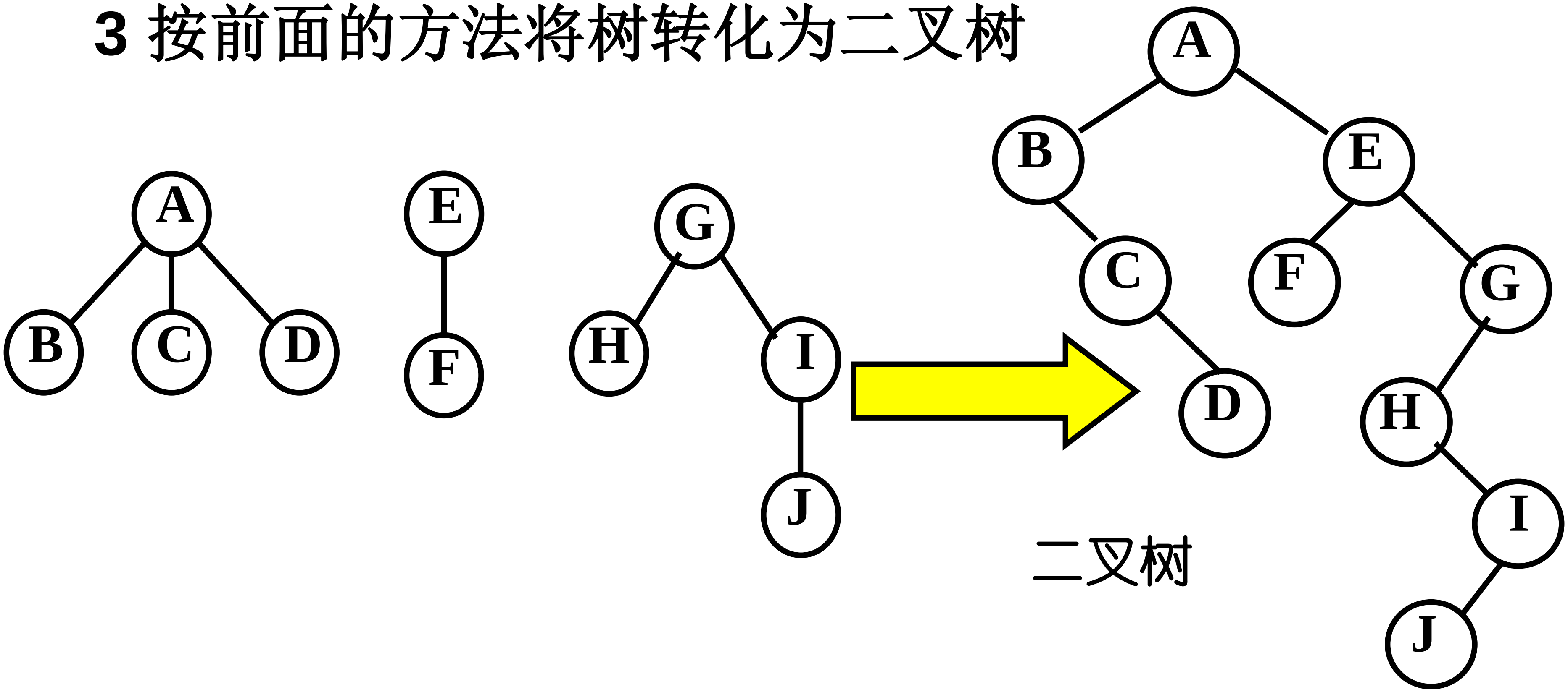
- 每个结点的大孩子作为对应二叉树该结点的左孩子；
- 每个结点的其它孩子形成对应二叉树该结点左孩子的右链
- 这样，在对应的二叉树中，每个结点的左孩子是其原树结点的大孩子，右孩子是其原树结点的大兄弟（下一兄弟）

2 森林转换成二叉树

1 把森林中第一棵树T_1的根作为整个森林的根；

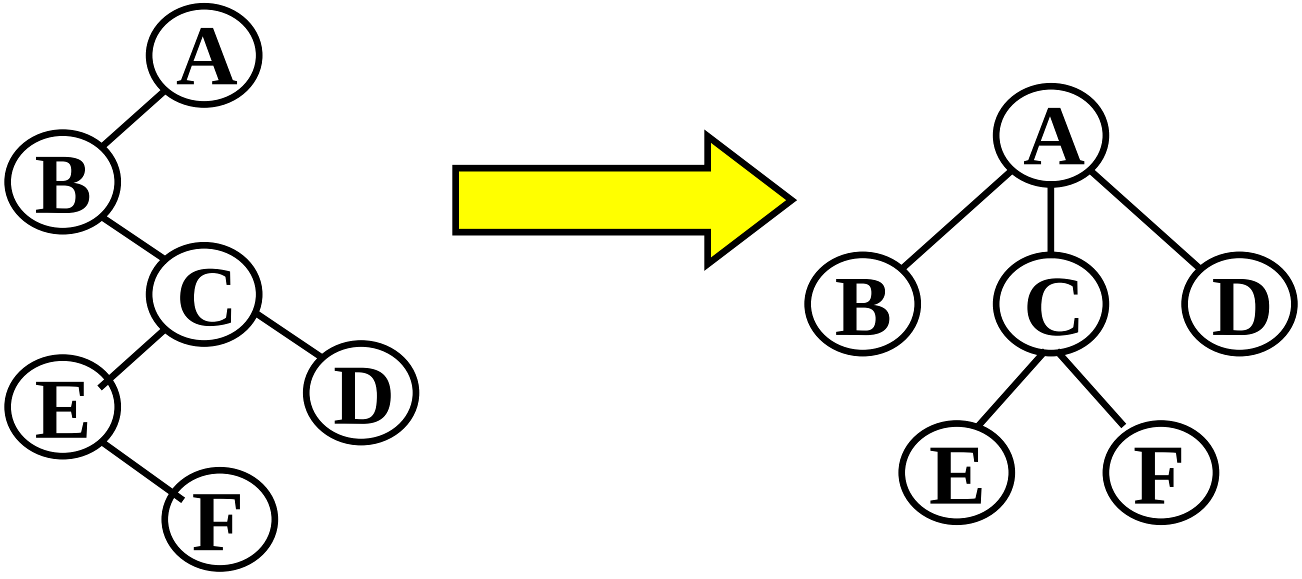
2 把森林中其它树的根依次作为T_1的兄弟结点。

3 按前面的方法将树转化为二叉树

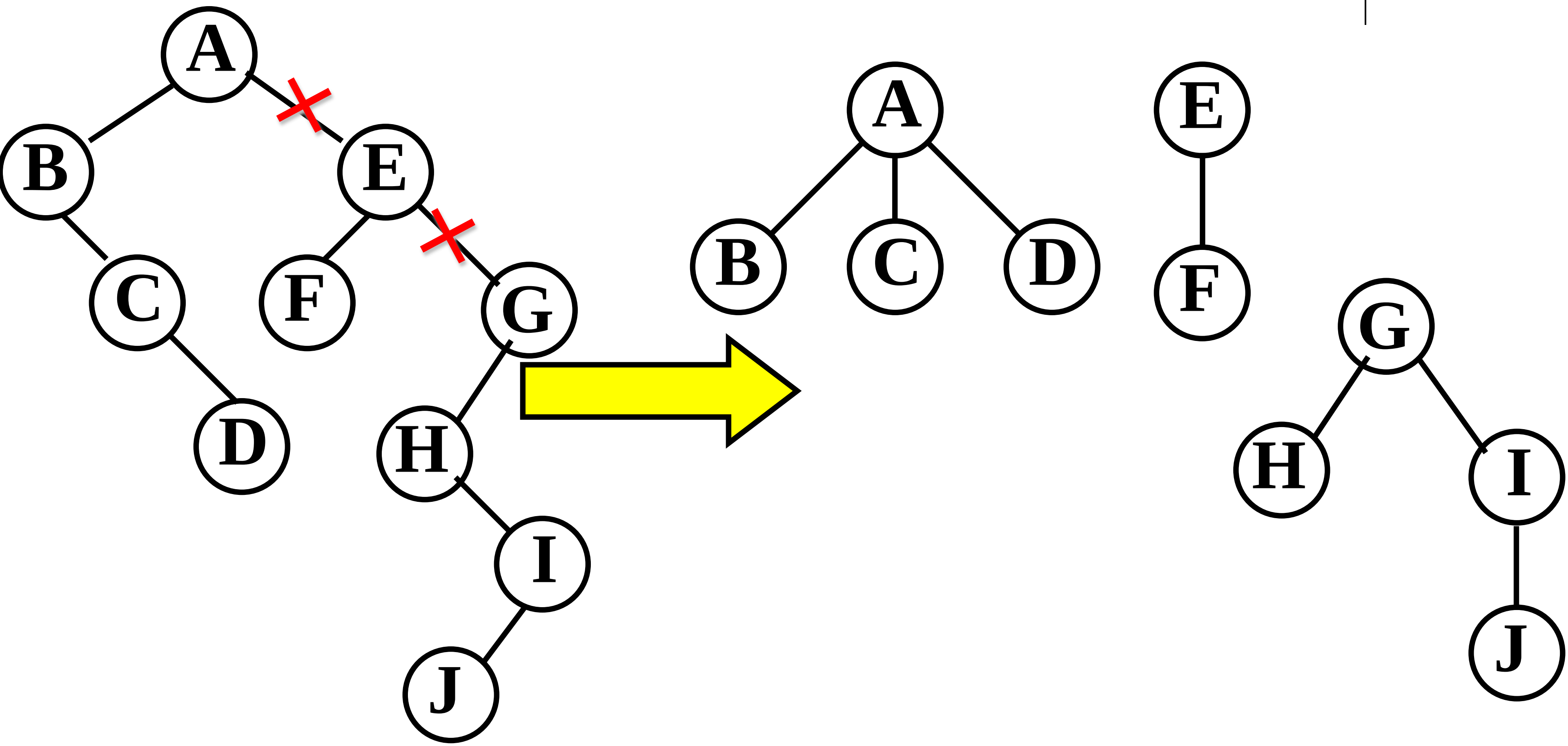


3 二叉树转换成树

- 如果二叉树根结点的右子树为空，反转前面将树转换成二叉树之方法，则能自然地将该二叉树转换成对应的树
  - ✓ 对每个结点，找其大兄弟结点及其父结点，并在两者间加一连线；
  - ✓ 去掉该结点和右孩子之间的连线；
  - ✓ 调整部分连线方向、长短使之成规范图形.


A
B
C
E
D
F
A
B
C
D
E
F

4 二叉树转成森林



森林与二叉树的自然对应

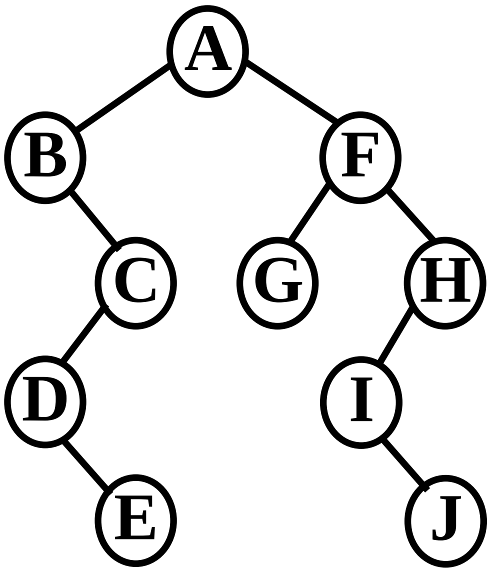
- 由上述转换可以看到，任何一个森林都对应唯一的二叉树。逆转这个过程，任何一个二叉树都对应唯一的森林。称这种变换为森林和二叉树的自然对应。

森林转换成二叉树

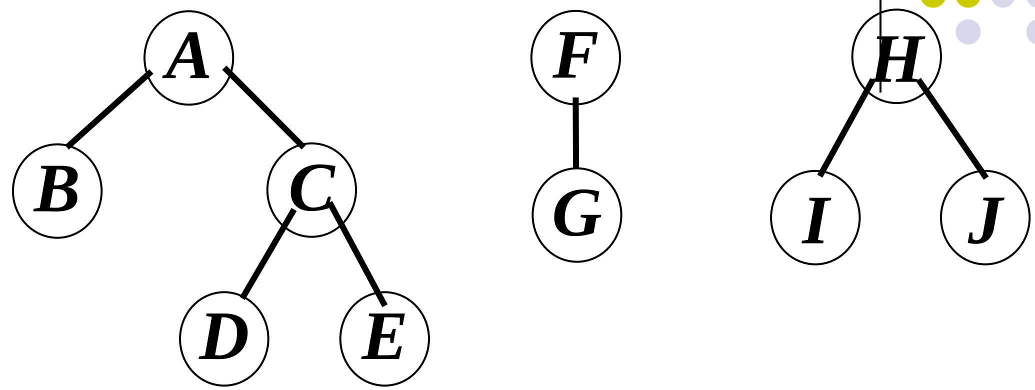
定义 5.9 设$F=(T_1，T_2，…，T_n)$表示由树$T_1, T_2, …, T_n$组成的森林，自然对应下森林 F 的二叉树$B(F)$递归定义如下：

若 $n = 0$，则 $B(F)$ 为空；

若 $n > 0$，则 $B(F)$ 的根是$Root(T_1)$，$B(F)$的右子树是$B((T_2, T_3, …, T_n))$，左子树是 $B((T_{11}, T_{12}, …, T_{1m}))$, 其中$T_{11}, T_{12}, …, T_{1m}$ 是 $Root(T_1)$ 的诸子树。



(*a*)



(*d*)

从上图(*a*)和(*d*)可知： 由森林(*d*)转换成二叉树(*a*)之后，二叉树的根是第一棵树的根，二叉树的左子树是由第一棵树的诸子树转换来的，二叉树的右子树是由第二棵树和第三棵树转换来的。

二叉树转换成森林

定义 5.10 设二叉树*T* 的根是 Root(*T*)，*T* 的左子树是 *L*，*T* 的右子树是 *R*，二叉树 *T* 在自然对应下的森林 *F*(*T*) 递归定义如下：

⑴ 若 Root(*T*) 为空, 则 *F*(*T*) 为空的森林；

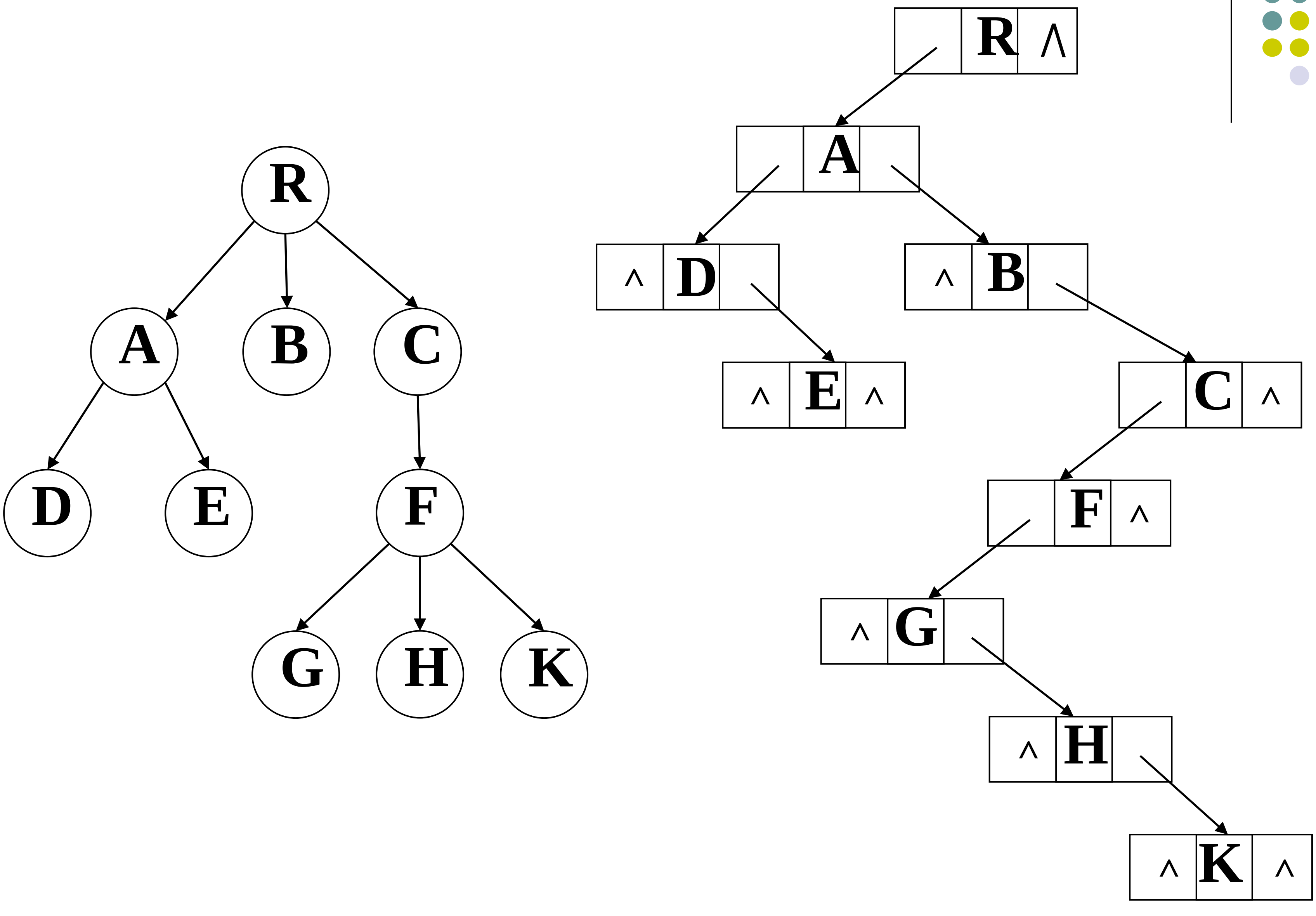
⑵ 若 Root(*T*) 非空, 则 *F*(*T*) 由第一棵树*T*1和森林*F*(*R*)组成, 其中*T*1是以Root(*T*)为根的树, *T*1的诸子树由森林*F*(*L*)组成。

左孩子-右兄弟链接结构

- 结点结构

FirstChild	**Data**	**NextBrother**

*这种存储结构的最大优点是：它和二叉树的二叉链表表示完全一样。可利用二叉树的算法来实现对树的操作。



R
A
B
C
D
E
F
G
H
K
R ∧
A
∧ D
∧ B
∧ E ∧
C ∧
F ∧
∧ G
∧ H
∧ K ∧

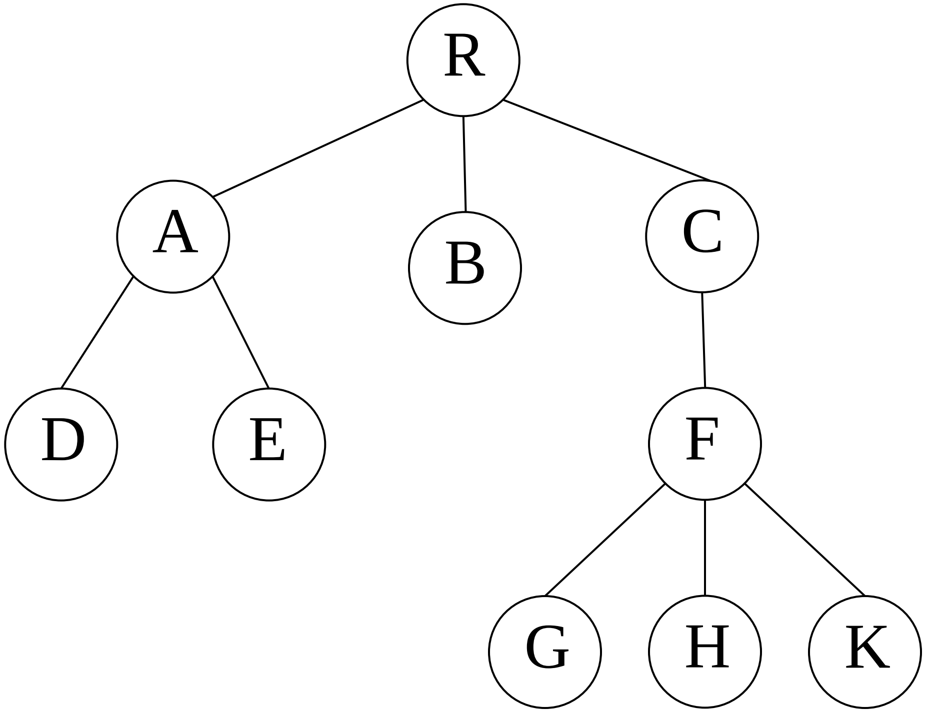
结点结构

```
template<class T>
struct TreeNode
{
        T data ;
        TreeNode<T> *firstChild，*nextBrother ;
};
```

树的先根遍历

（1）访问根结点

（2）从左到右依次先根次序遍历树的诸子树



先根序列

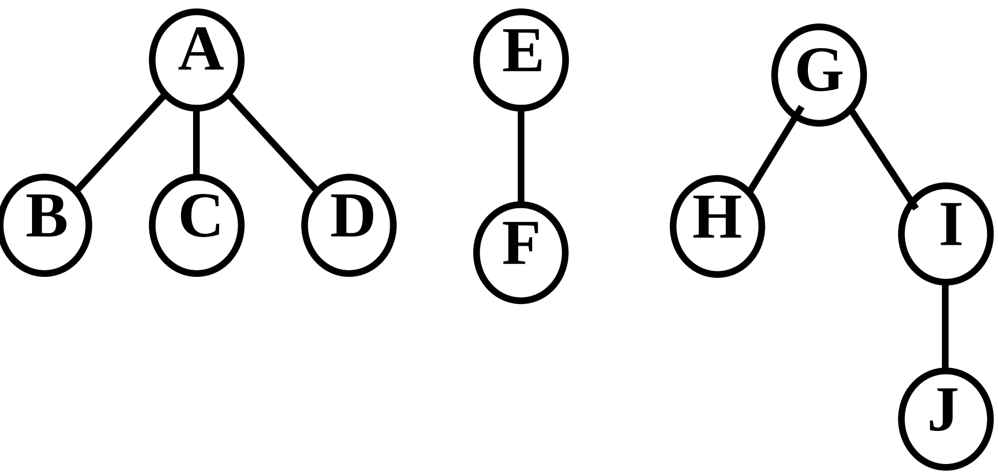
RADEBCFGHK

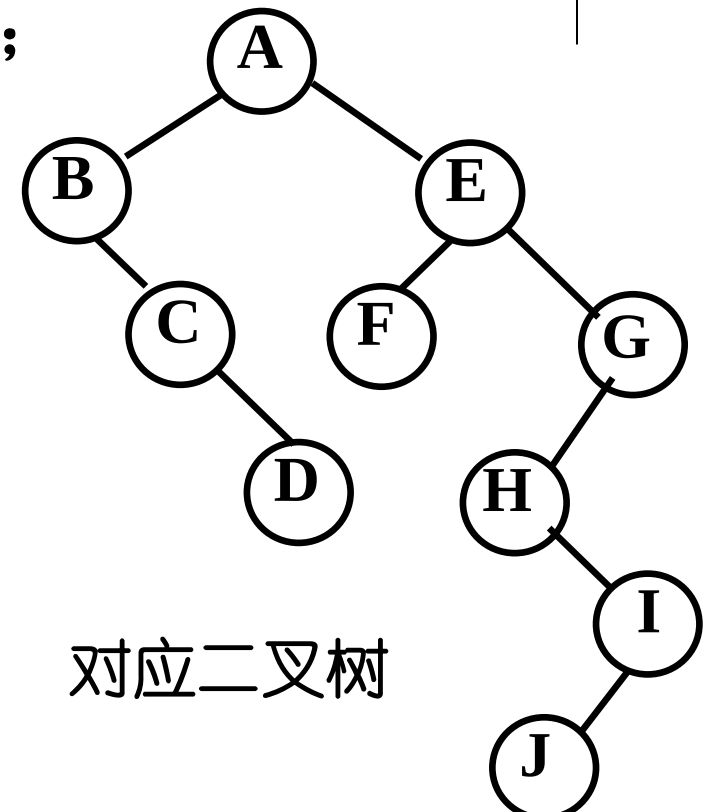
森林的先根遍历

① 访问森林中第一棵树的根结点；

② 先序遍历第一棵树中的诸子树；

③ 先序遍历其余的诸树。





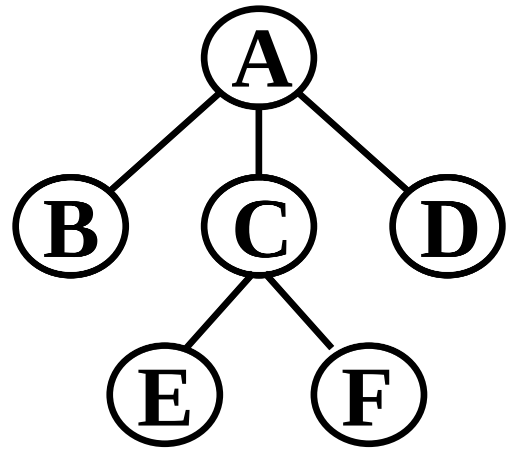
森林的先根遍历序列：

A B C D E F G H I J

二叉树的先根序列：

A B C D E F G H I J

先根遍历（递归）



算法PreOrder(*t*)

P1. [若*t*为空返回]

if (*t* ==NULL) return;

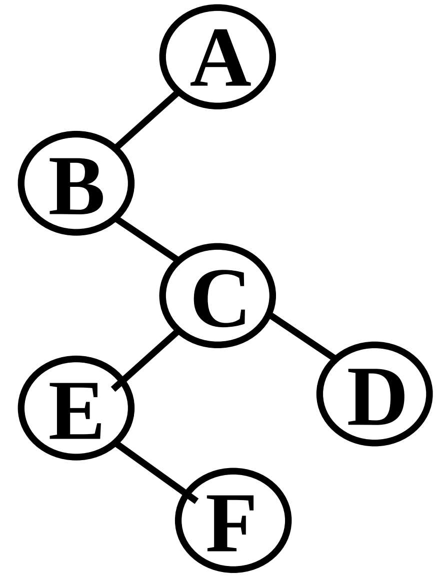
P2. [访问*t*所指结点]

cout<< t->*Data* ;

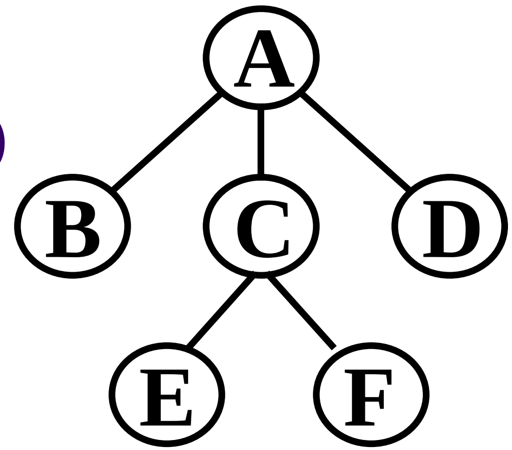
P3. [先根遍历*t*所指结点的诸子树]

for (p = t -> FirstChild ; p ; p = p->NextBrother)

PreOrder (p);



先根遍历（二叉树）



算法Pre(*t*)

Pre1. [若*t*为空返回]

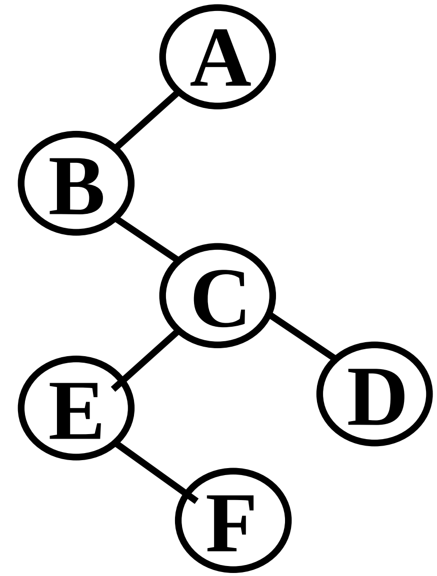
if (*t* ==NULL) return;

Pre2. [访问*t*所指结点]

cout<< t->*Data* ；

Pre(t->FirstChild);

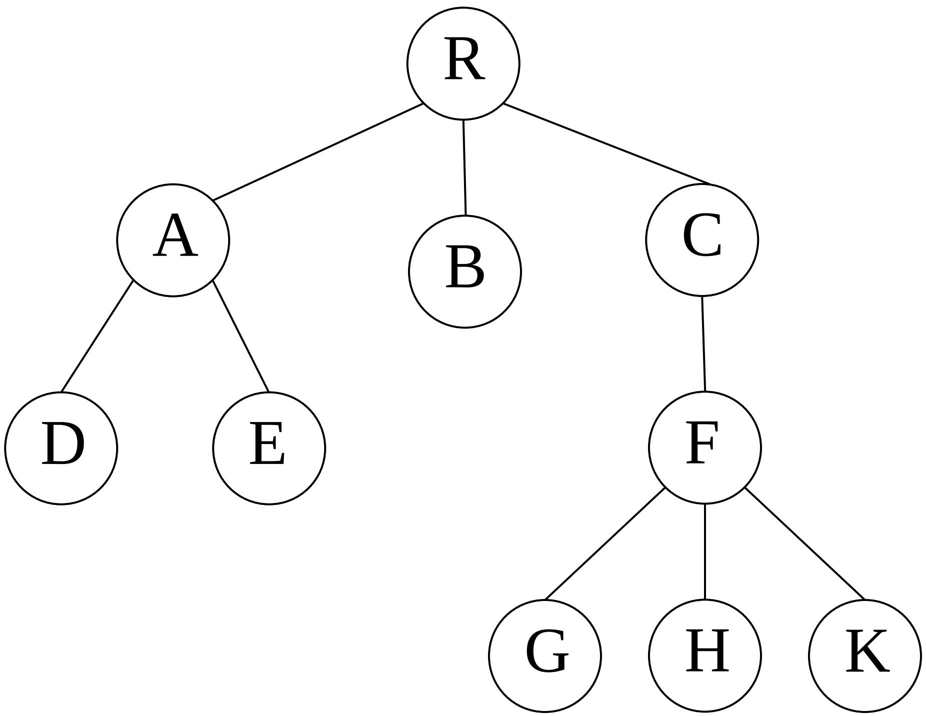
Pre(t->NextBrother);



树的后根遍历

（1）从左到右依次后根次序遍历树的诸子树

（2）访问根结点



后根序列

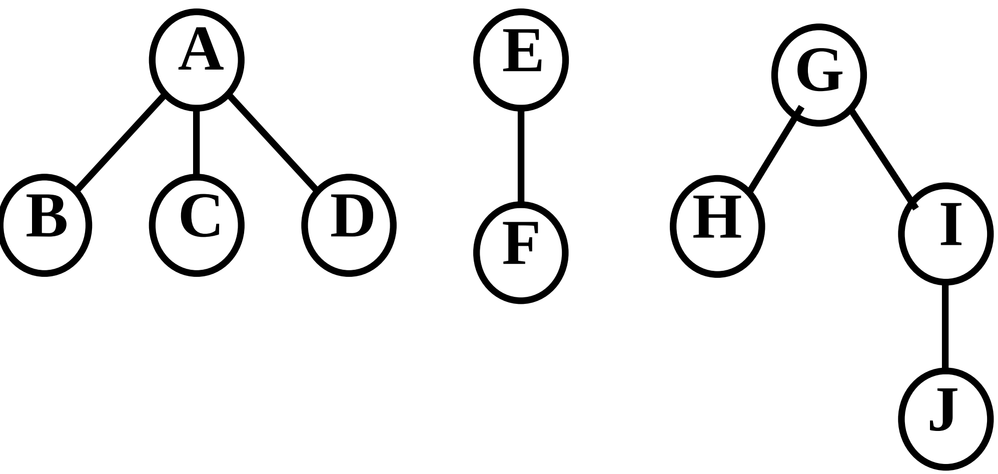
DEABGHKFCR

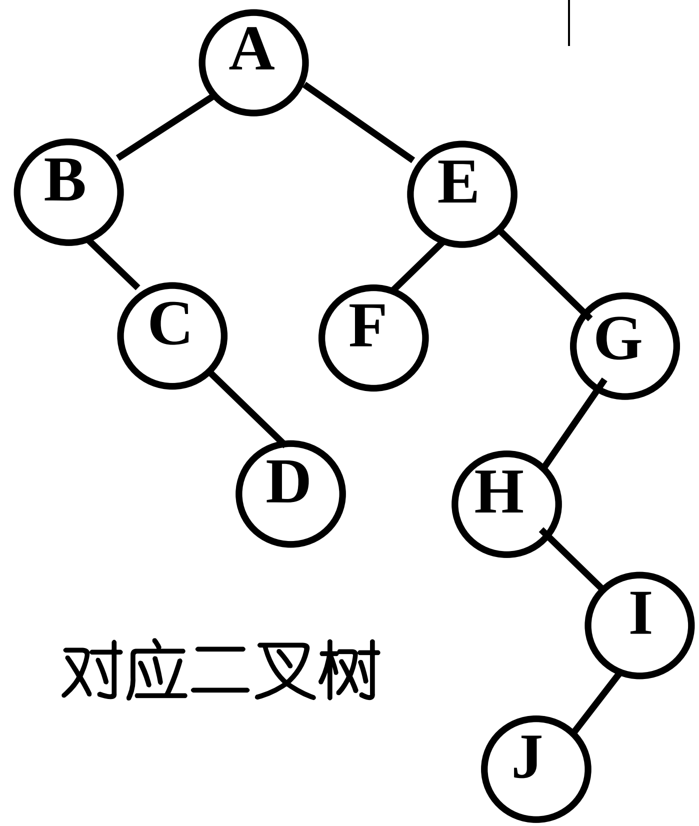
后根遍历森林：

① 后序遍历第一棵树的诸子树；

② 访问森林中第一棵树的根结点；

③ 后序遍历其余的诸树。





对应二叉树

森林的后根遍历序列：

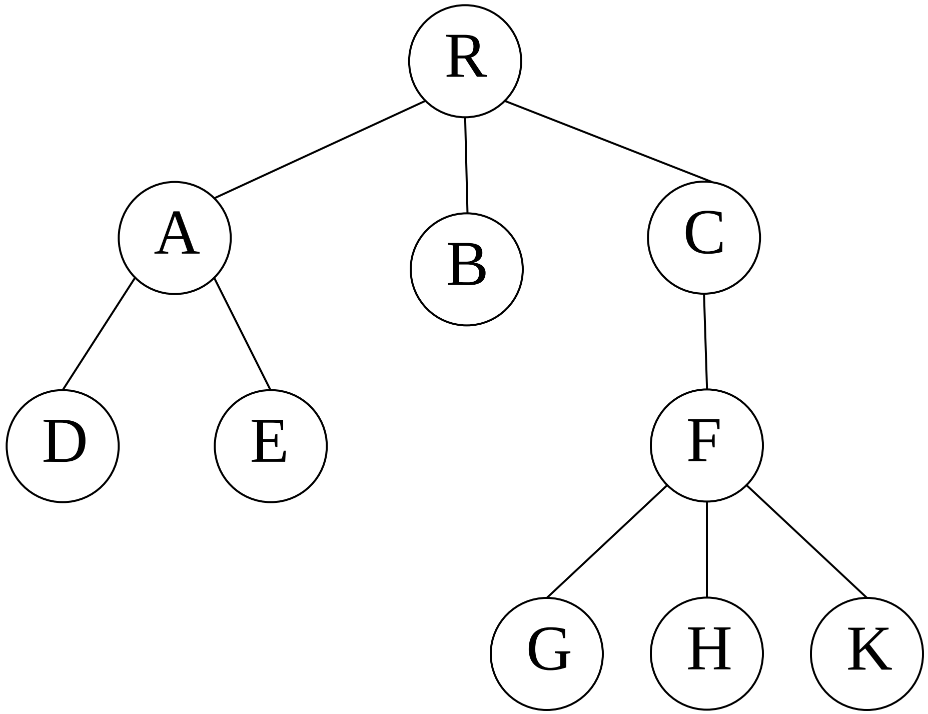
B C D A F E H J I G

二叉树的中根序列：

B C D A F E H J I G

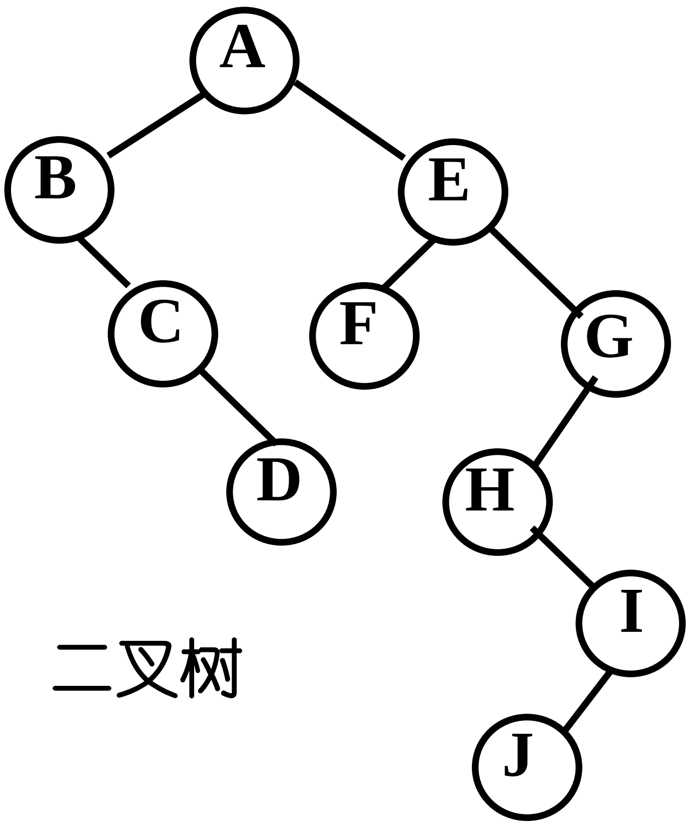
层次序列

（1）从上到下，依次对每层的结点遍历
（2）每层从左到右依次遍历树的诸结点



层次序列
RABCDEFGHK

树和森林的层次遍历



二叉树

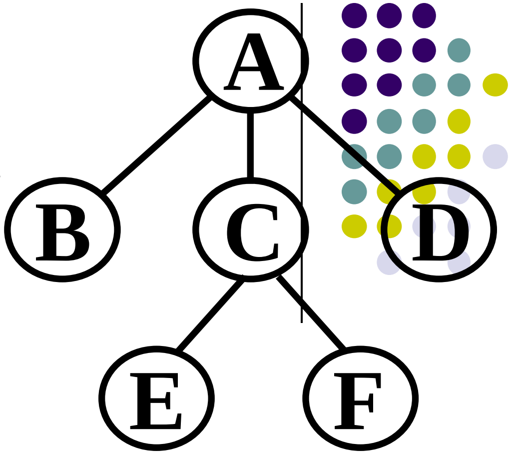
是沿NextBrother链访问的过程，

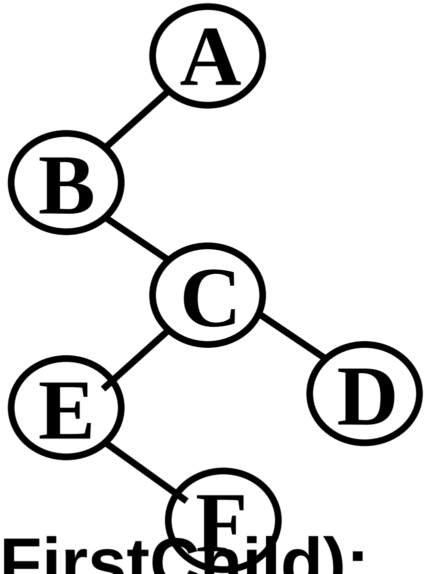
指针 FirstChild起承上启下的作用，

可以使用队列辅助存储。

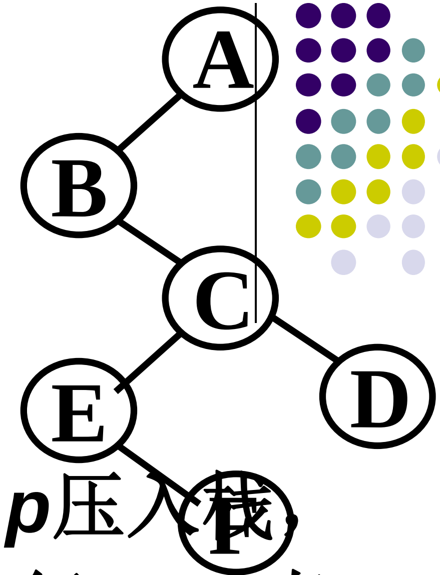
算法LevelOrder（t）

```
// 指针 t 指向与森林自然对应的二叉树的根
L1 [初始化]
    CREATEQueue(Q);
    IF t != NULL THEN Q.Insert ( t )
L2 [利用队列进行层次遍历]
    while( !Q.empty()) {
         p = Q.delete() ;
        while (p!=NULL) {
            cout << p->Data ;
           if(p->FirstChild!=NULL) Q.push(p->FirstChild);
           p = p->NextBrother  }
      }
```





先根遍历迭代算法 I

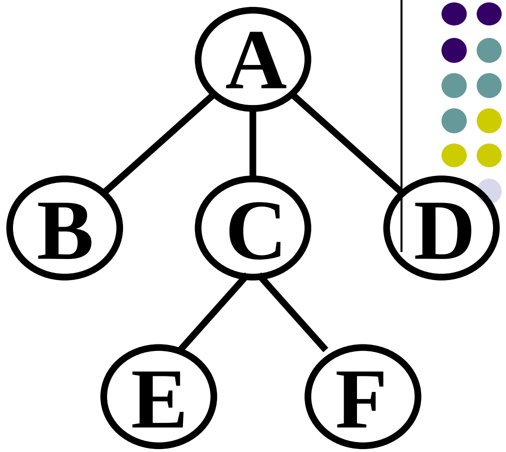


首先，将结点*p*设为根结点。

(1)若结点*p*不为空，访问结点*p*，将结点*p*压入栈，并将其大儿子结点设为结点*p*；反复执行**(1)**，直至结点*p*为空。

(2)从栈中弹出一个结点，若它有大兄弟结点，则将其大兄弟结点压入栈；否则，反复执行**(2)**，直至弹出的结点有大兄弟结点或栈空。

(3)反复执行**(1)(2)**，直至栈为空。



算法NPO(*t*)

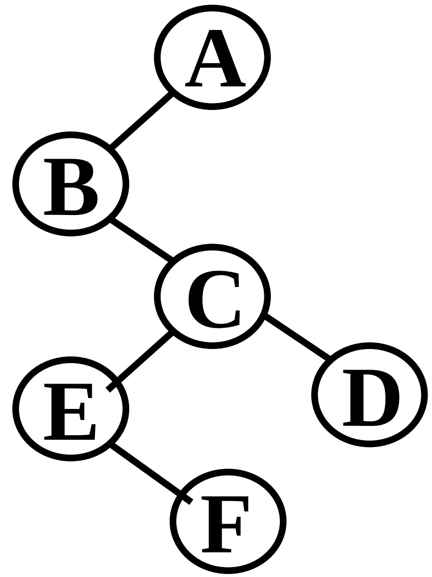
NPO1. [初始化堆栈]

CREATEStack(*S*);

p = *t* ;

NPO2. [若*p*所指结点不为空，访问*p*所指结点，将*p*压入栈，并将其*FirstChild*指针设为*p.*]

```
while (  p!=NULL ) {
        cout<< p->Data ;
        S.Push ( p );
        p = p->firstChild ; }
```



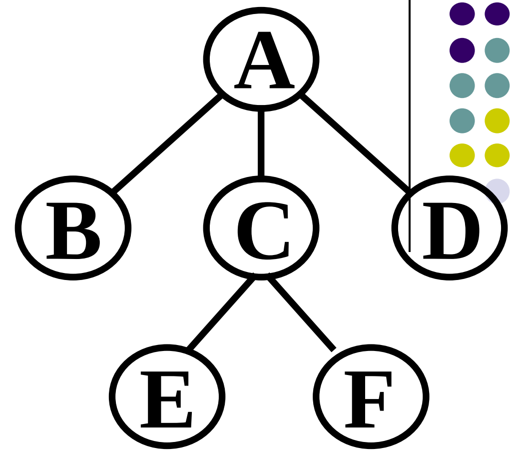
NPO3. [从栈中弹出指针，直至弹出的指针所指结点有大兄弟结点或栈空以至无结点可弹出。]

```
while ( p == NULL && !S.empty()) {
        p = S.Pop() ;
        p = p-> NextBrother;
}
```

NPO4. [重复上述过程]

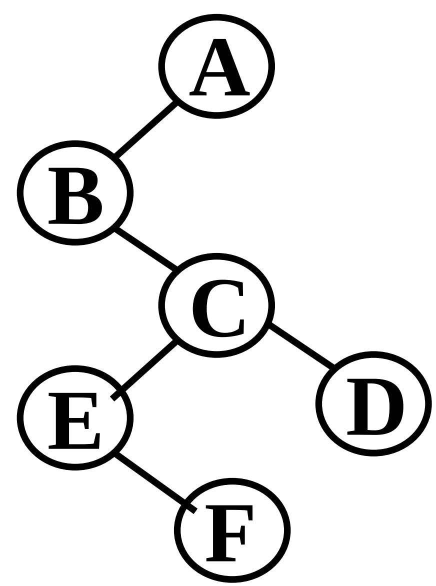
```
if ( p != NULL ) goto NPO2. ▌
```

先根遍历迭代算法 II

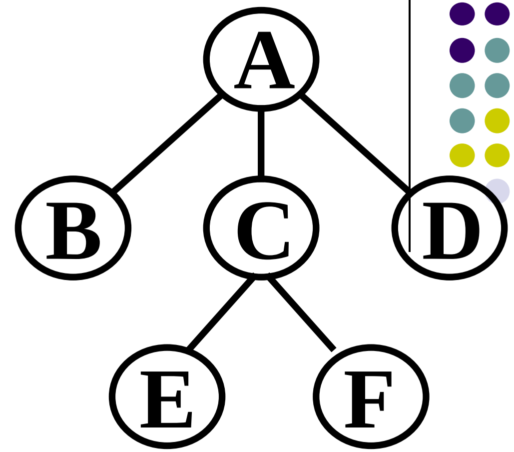


算法NPO(*t*)

```
NPO1. [初始化堆栈]
      CREATE Stack S ;
      IF t != NULL THEN S.push(t);
NPO2. [ 迭代 ]
     while  (!S.empty()) {
            p = S.Pop() ;
            while( p!=NULL ){
                 cout<<p->Data;
                 S.push(p->NextBrother);
                 p=p->FirstChild;
            }
     }
```

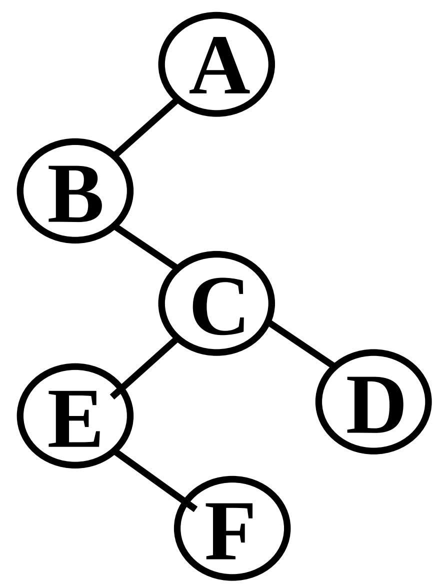


先根遍历迭代算法 III



算法NPO(*t*)

```
NPO1. [初始化堆栈]
      CREATE Stack S ;
      IF t != NULL THEN p = t ;
NPO2. [ 迭代 ]
     while  (!S.empty()) {
            p = S.Pop() ;
            if( p!=NULL ){
                 cout<<p->data;
                 S.push(p->nextBrother);
                 S.push(p->firstChild);
            }
     }
```



搜索大儿子结点（大兄弟结点）

算法GFC(*p*. *q*)

GFC1. [指针*p*所指结点存在，并且存在大儿子结点]

IF $p \neq \Lambda$ AND *FirstChild* $(p) \neq \Lambda$ THEN

RETURN $q \leftarrow$ *FirstChild* (p) .

GFC2. [大儿子结点不存在]

RETURN $q \leftarrow \Lambda$. ▌

```
算法FindFather( t, p. result)
/*查找结点的父结点  */
FF1 [特判]
    IF t = Λ OR p = Λ THEN (result ← Λ. RETURN).
FF2[从t的第一棵子树开始依次搜索诸子树]
   q←FirstChild (t).
   WHILE q≠Λ AND q≠p DO(
      IF q = p THEN result ←t.
      ELSE  FindFather( q, p. result).
      IF result =Λ  THEN  q←NextBrother(q)). ■
```

算法FindTarget(t, $target$. $result$)

/* 搜索指定数据域的结点 */

FT1[t不存在或为所求]

IF $t = \Lambda$ THEN ($result \leftarrow \Lambda$. RETURN.)

IF $Data$ (t) = $target$ THEN ($result \leftarrow$ t. RETURN.)

FT2[从t的第一棵子树开始依次搜索诸子树]

$p \leftarrow FirstChild$ (t) .

WHILE $p \neq \Lambda$ DO

(FindTarget (p , $target$. $result$).

IF $result \neq \Lambda$ THEN RETURN.

$p \leftarrow NextBrother(p)$)). ▌

算法Del (*t*)

/*释放根为*p*的子树所占用的空间 */

Del1. [指针*t*所指结点不存在，则返回]

IF $t = \Lambda$ THEN RETURN.

Del2. [从左到右删除*t*的子树]

$p \leftarrow FirstChild\,(t)$.

WHILE $p \neq \Lambda$ DO

(q $\leftarrow$ *NextBrother* (*p*) .

Del (*p*) .

$p \leftarrow q$.)

AVAIL $\Leftarrow t$. ▌

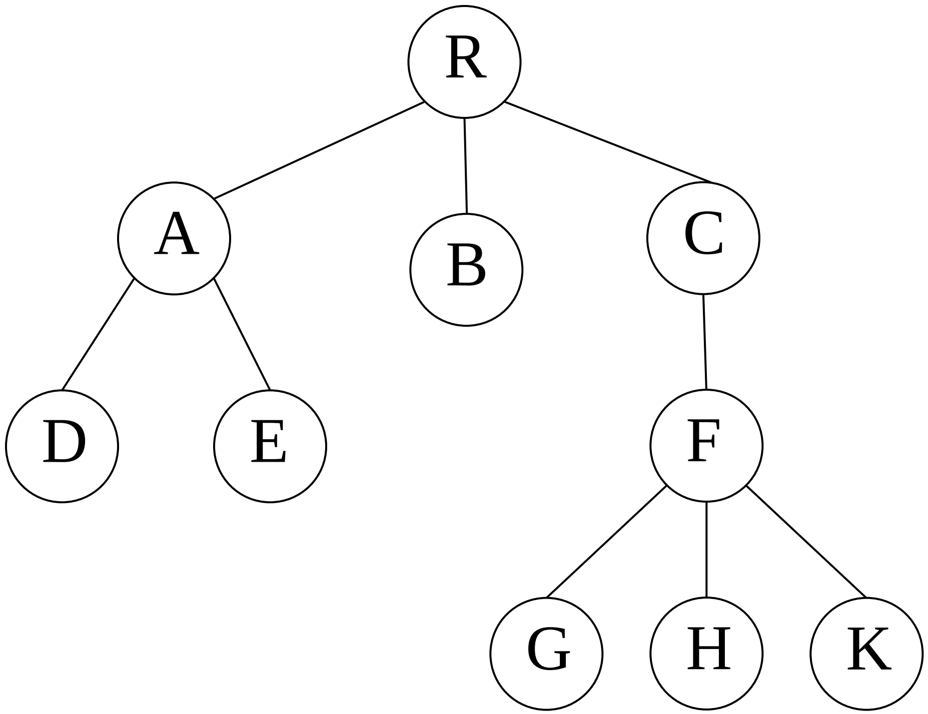
删除子树

三种情况：

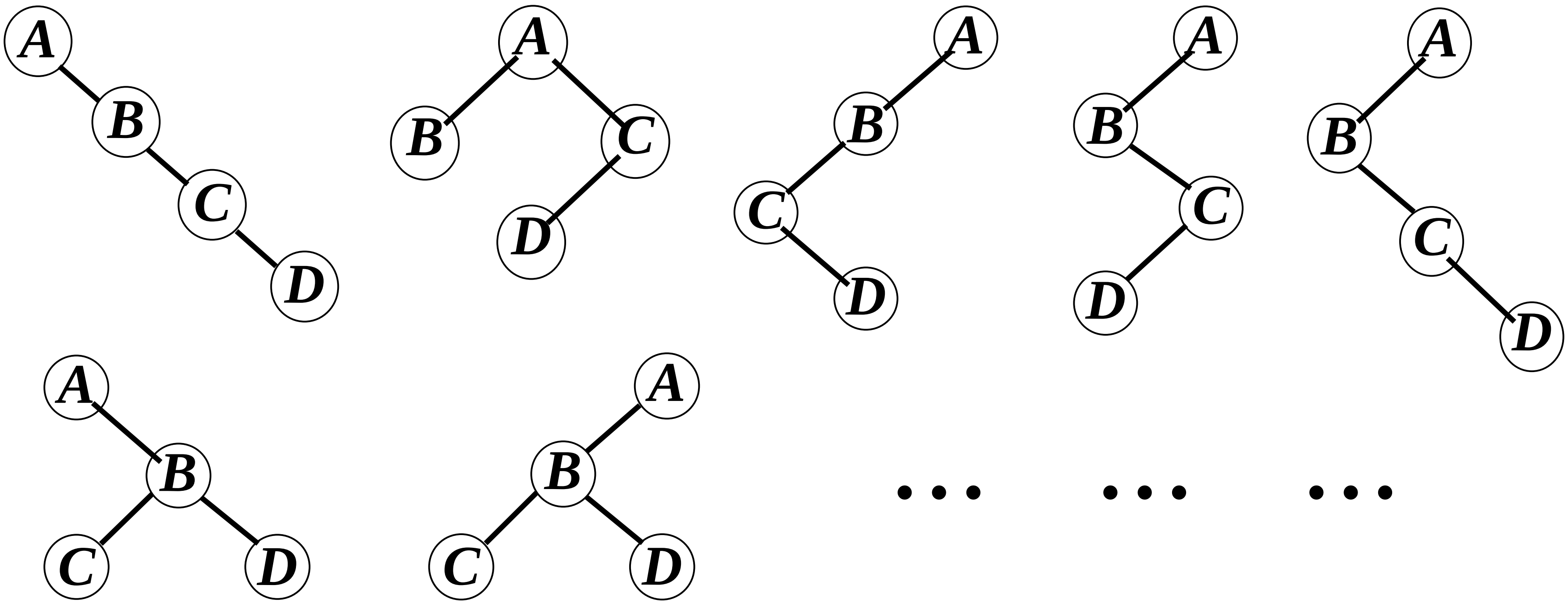
1、删除以根结点**R**为根的树

2、删除以大儿子结点**A**为根的子树

3、删除以结点**B**或**C(**非大儿子**)**为根的子树



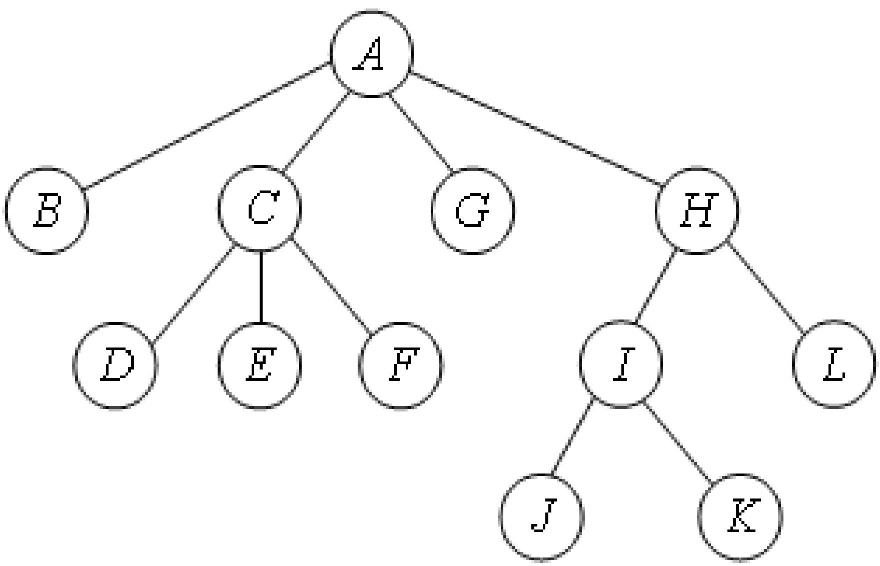
树的顺序表示



先根序列皆为*ABCD*. 显然，单独用先根序列无法确定 树的结构.

例：先根序列： A B C D E F G H I J K L

结点的次数: 4 0 3 0 0 0 0 2 2 0 0 0



定理5.3 如果已知一个树的先根序列和每个结点相应的次数（度），则能唯一确定该树的结构。

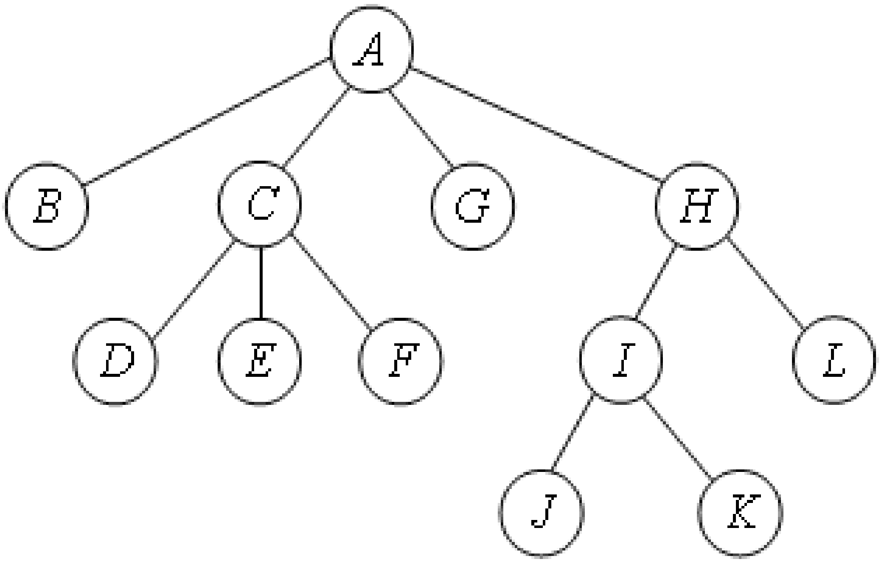
证明：用数学归纳法

1. 若树中只有一个结点，定理显然成立。

2. 假设树中结点个数小于n(n≥2)时定理成立。

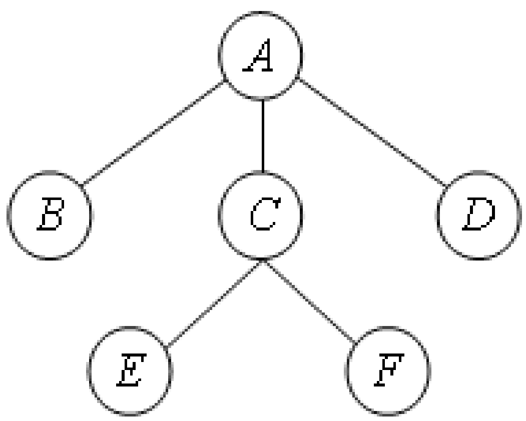
当树中有n个结点时，由树的先根序列可知，第一个结点是根结点，设该结点的次数为k，k≥1，因此根结点有k个子树。第一个子树排在最前面，第k个子树排在最后面，并且每个子树的结点个数小于n，由归纳假设可知，每个子树可以唯一确定，从而整棵树的树形可以唯一确定。证毕。

后根序列	B	D	E	F	C	G	J	K	I	L	H	A
结点次数	0	0	0	0	3	0	0	0	2	0	2	4



已知一个树的层次序列和每个结点次数，则能唯一确定该树的结构。

层次序列	A	B	C	D	E	F
结点的次数	3	0	2	0	0	0



其它表示方式

- 儿子表结束符

  A B) C E) F)) D))

- 括号表示法