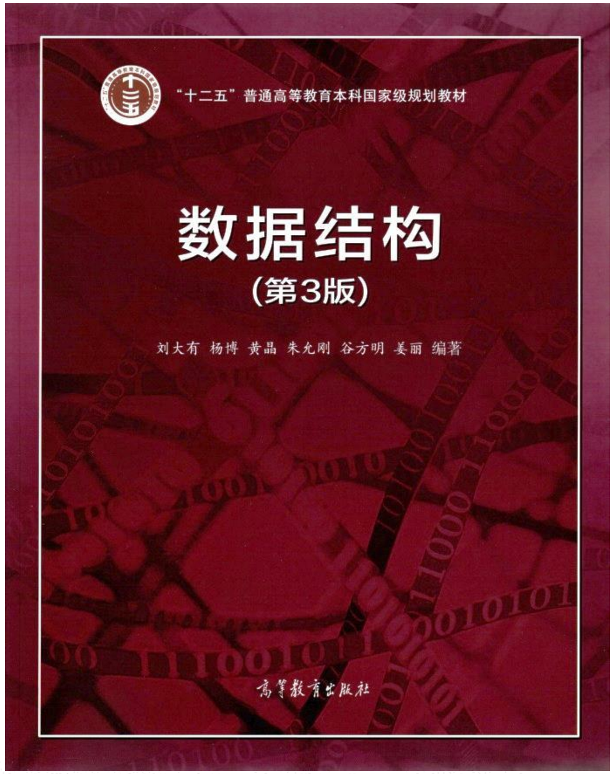


树的存储和操作

- 树与二叉树的转换
- 树的存储结构
- 树和森林的遍历

数据之法
结构之美
算法之道

Last updated on 2022.10

zhuyungang@jlu.edu.cn

李煜东
2012年全国中学生信息学奥赛NOI金牌
2015年ACM-ICPC竞赛亚洲区域赛冠军
2017年毕业于北京大学
任职Google软件工程师

在大约一年半的时间里，我依靠3000道以上的巨大刷题量，才从一名连深度优先搜索都写不对的初学者，成长为NOI金牌得主，并入选国家集训队。

在思维的迷宫里，有的人凭天生的灵感直奔终点；有的人以持久的勤勉，铸造出适合自己的罗盘。

树与二叉树的转换

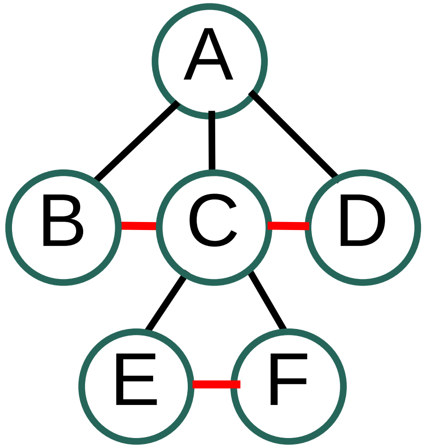
1) 树转换成二叉树
2) 森林转换成二叉树
3) 二叉树转换成树
4) 二叉树转换成森林

（1） 树转换成二叉树

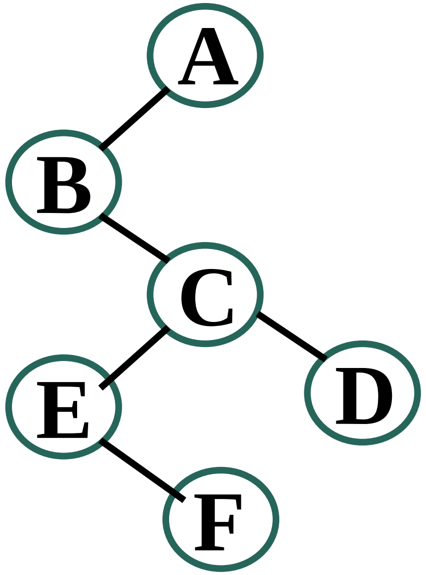
① 在所有兄弟结点间加一条连线；

② 对每个结点与其子结点的连线：保留与其左孩子的连线，去掉与其它孩子间的连线；

③ 调整部分连线方向、长短使之成为符合二叉树规范的图形。



树



由树转换的二叉树

1.一个结点的左孩子仍然是该结点的左孩子

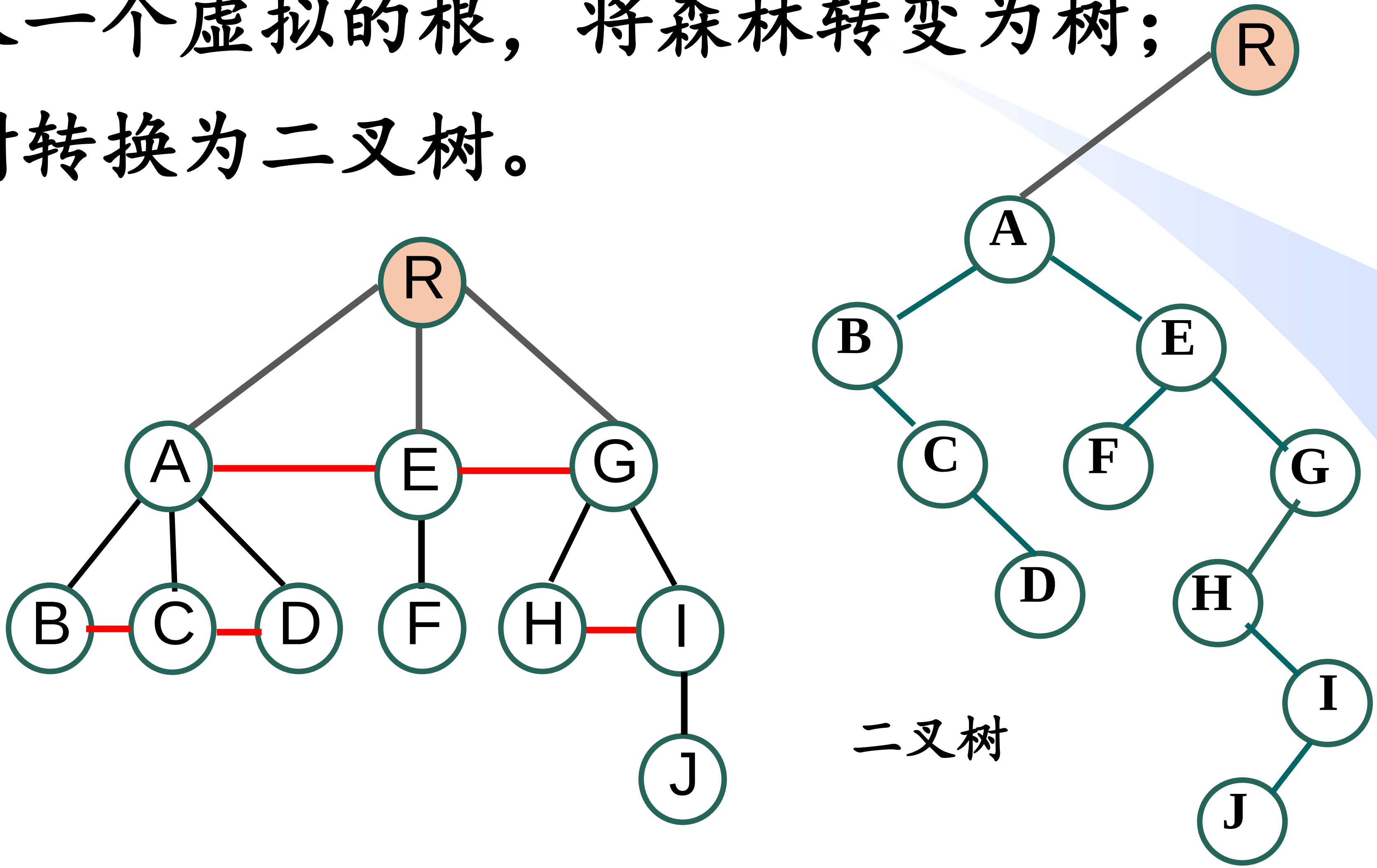
2.一个结点的右兄弟变成该结点的右孩子

3.转换后的二叉树的根结点无右孩子

（2）森林转换成二叉树（方法1）

① 引入一个虚拟的根，将森林转变为树；

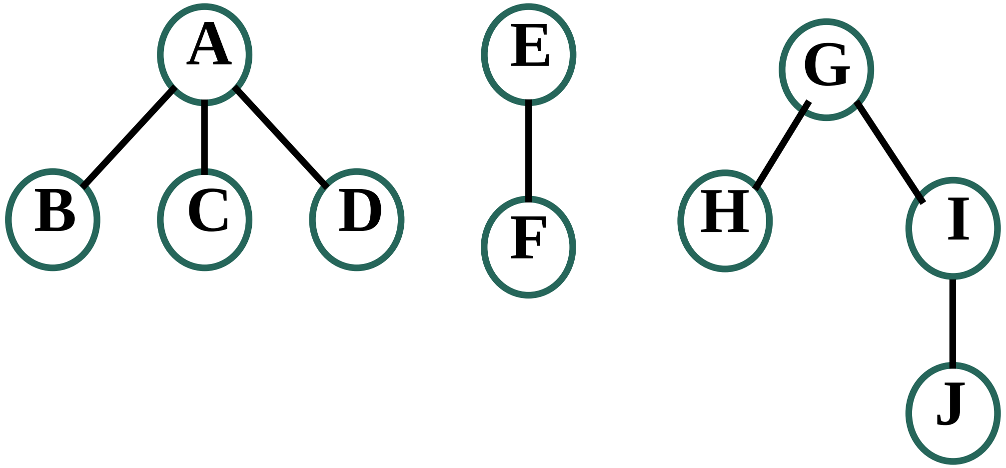
② 将树转换为二叉树。



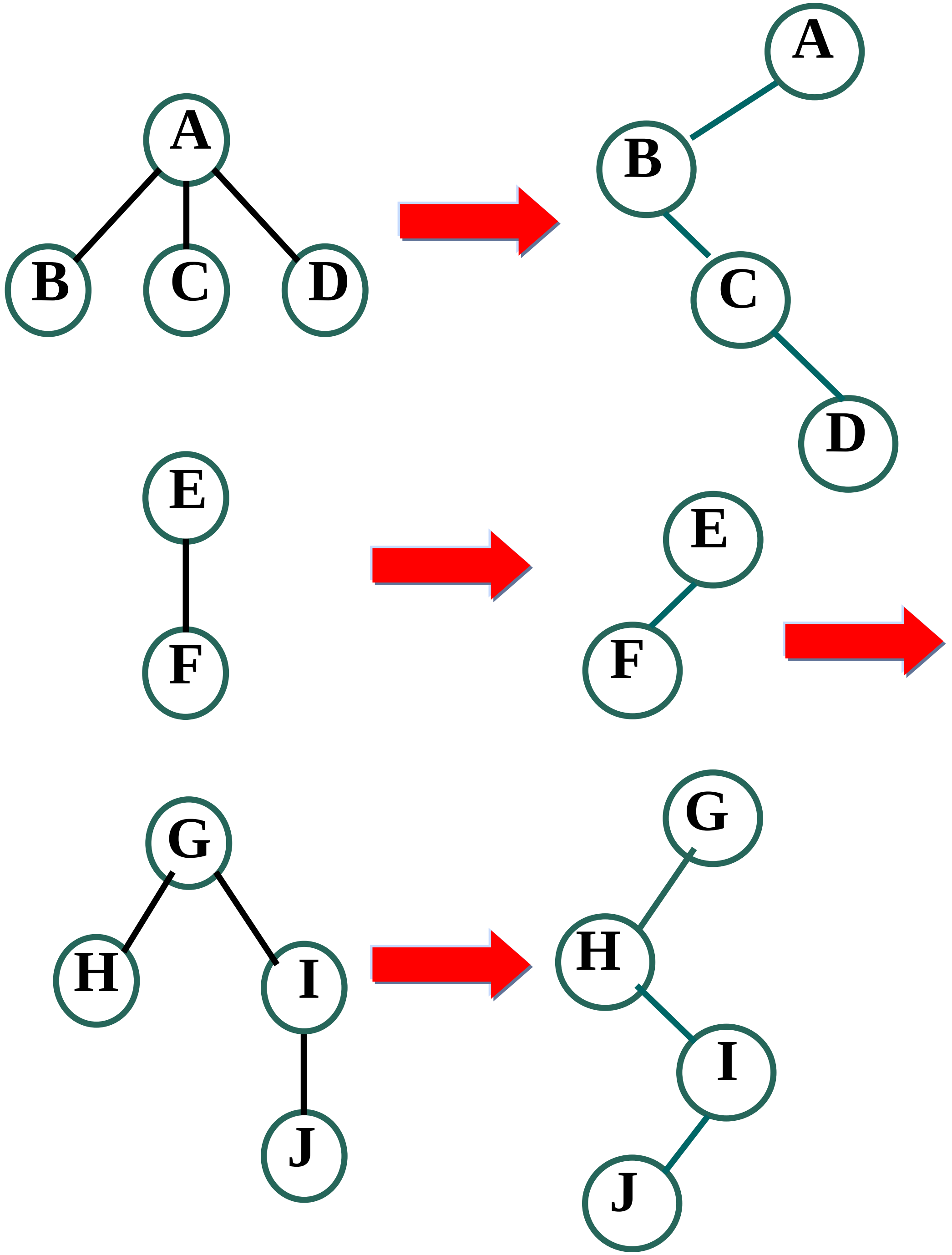
（2）森林转换成二叉树（方法2）

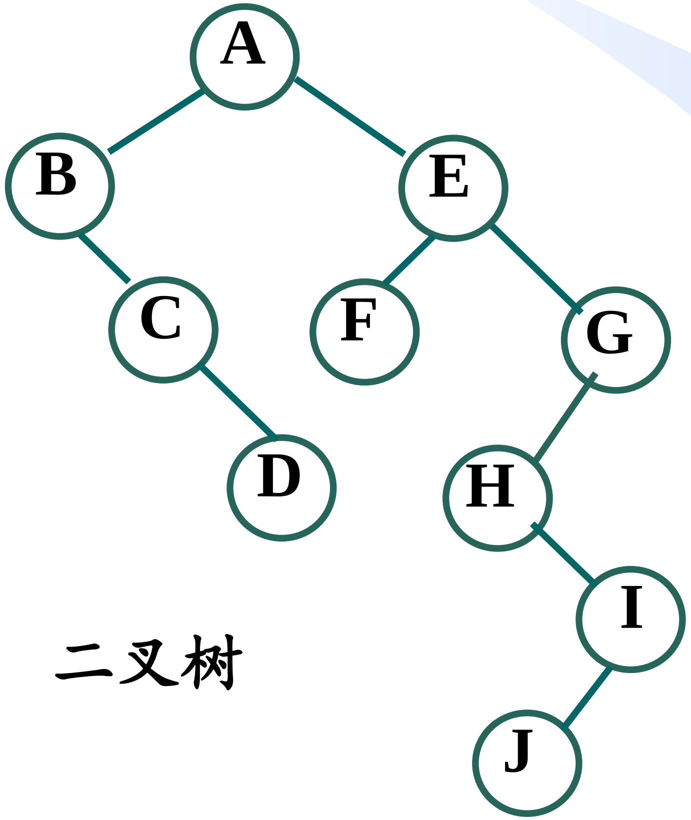
① 首先将每个树转换为二叉树；

② 将以第一个二叉树的根为总根，将其它二叉树的根依次作为上一个二叉树根结点的右孩子。



将以第一个二叉树的根为总根，将其它二叉树的根依次作为上一个二叉树根结点的右孩子





二叉树

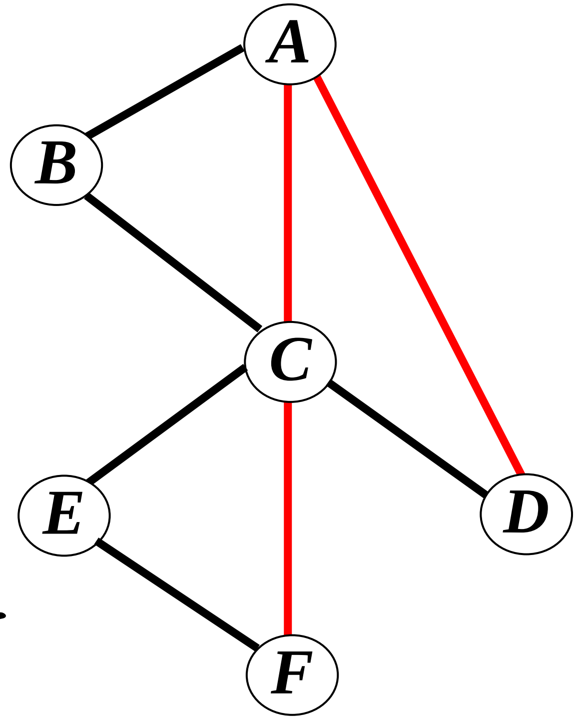
（3）二叉树转换成树

二叉树的右子树为空，则可转换为一棵树。

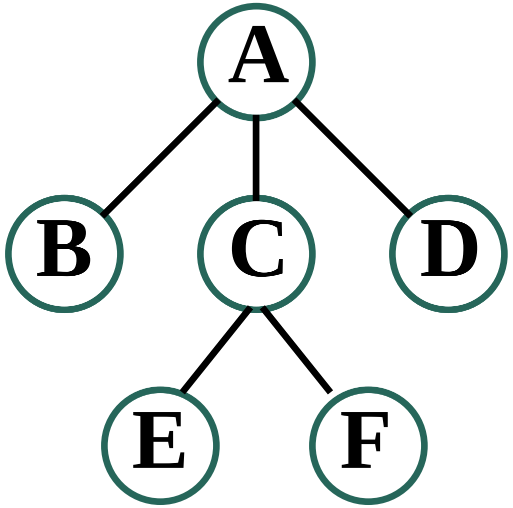
① 对每个结点，若该结点有左孩子，将左孩子的右孩子、右孩子的右孩子……与该结点用线连接起来；

② 去掉所有父结点和右孩子之间的连线；

③ 调整部分连线方向、长短使之成规范图形.



二叉树



由二叉树转换的树

1.一个结点的左孩子仍然是该结点的孩子

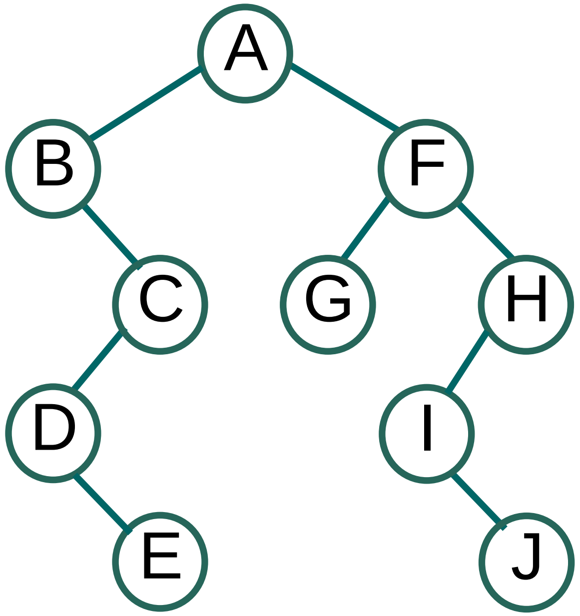
2.一个结点的右孩子是该结点的兄弟。

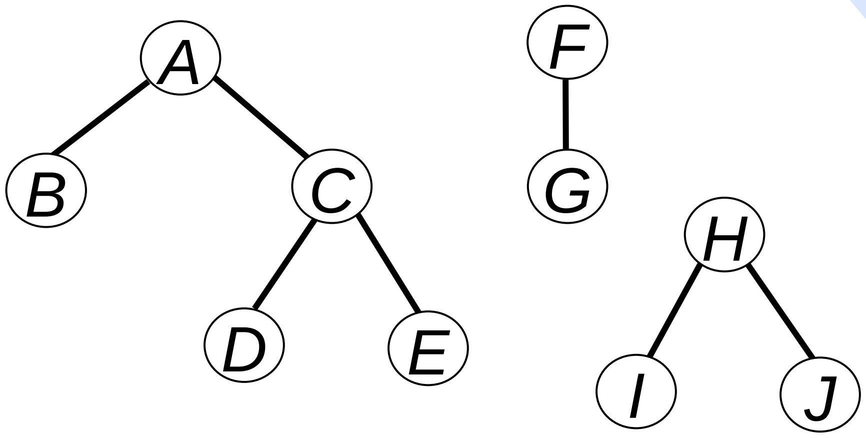
（4）二叉树转成森林

二叉树的右子树不为空，则可转换为森林。

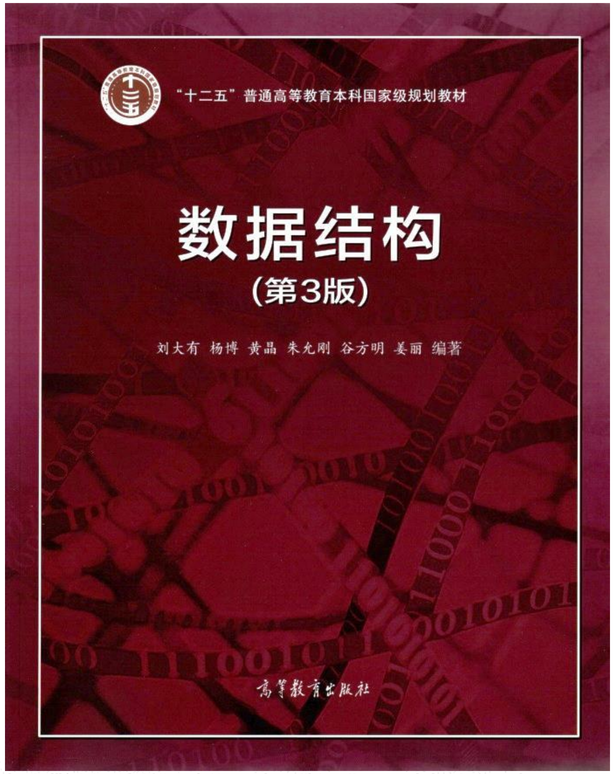
① 从根出发，断开其与右孩子的连线，得到多个二叉树；

② 将每个二叉树按以上方法转化为树。





- 任一森林都对应一棵二叉树。
- 任一棵二叉树都对应唯一的一个森林。
- 称这个变换为森林与二叉树之间的自然对应。



树的存储和操作

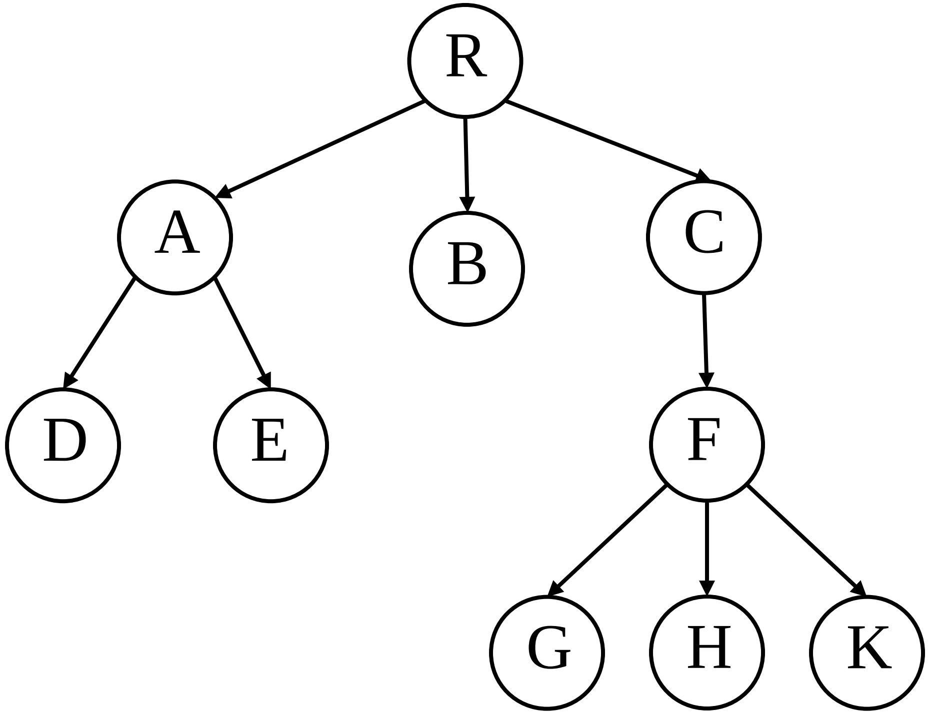
- 树与二叉树的转换
- **树的存储结构**
- 树和森林的遍历

数据之法
结构之美
算法之道

Last updated on 2022.10

zhuyungang@jlu.edu.cn

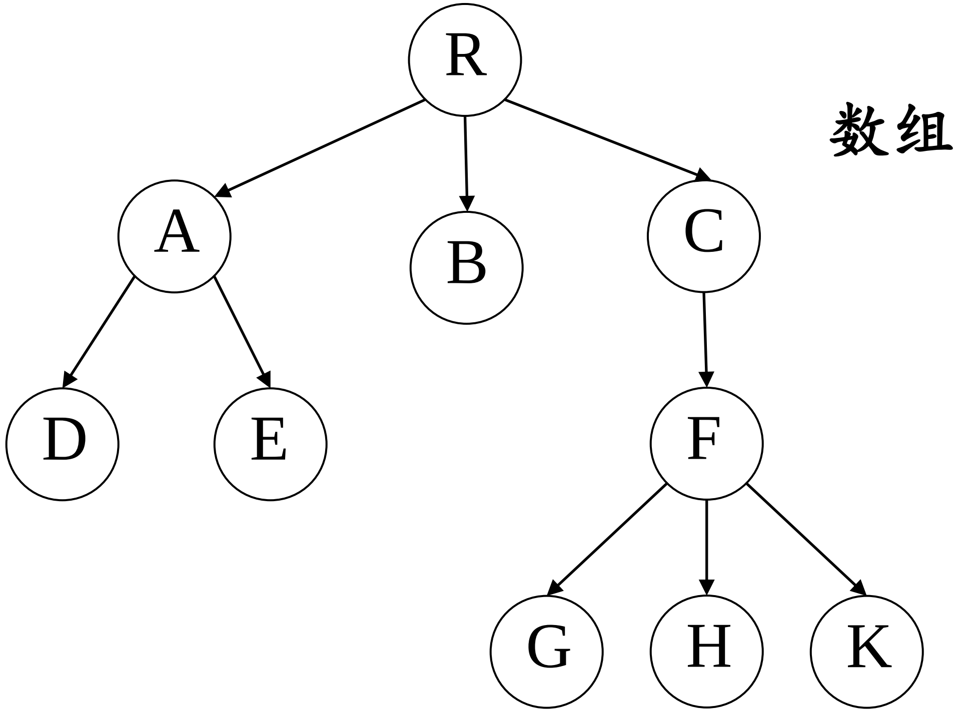
双亲表示法：层次顺序+父结点下标



数组下标：		
1	**R**	**0**
2	**A**	**1**
3	**B**	**1**
4	**C**	**1**
5	**D**	**2**
6	**E**	**2**
7	**F**	**4**
8	**G**	**7**
9	**H**	**7**
10	**K**	**7**

- 按层次遍历顺序对结点编号
- 编号为*i*的结点存放在数组的第*i*个位置
- 便于涉及父结点的操作；
- 求结点的子结点时需要遍历整棵树。

孩子表示法:层次顺序+子结点下标



数组下标:

1	R	2	3	4
2	A	5	6	0
3	B	0	0	0
4	C	7	0	0
5	D	0	0	0
6	E	0	0	0
7	F	8	9	10
8	G	0	0	0
9	H	0	0	0
10	K	0	0	0

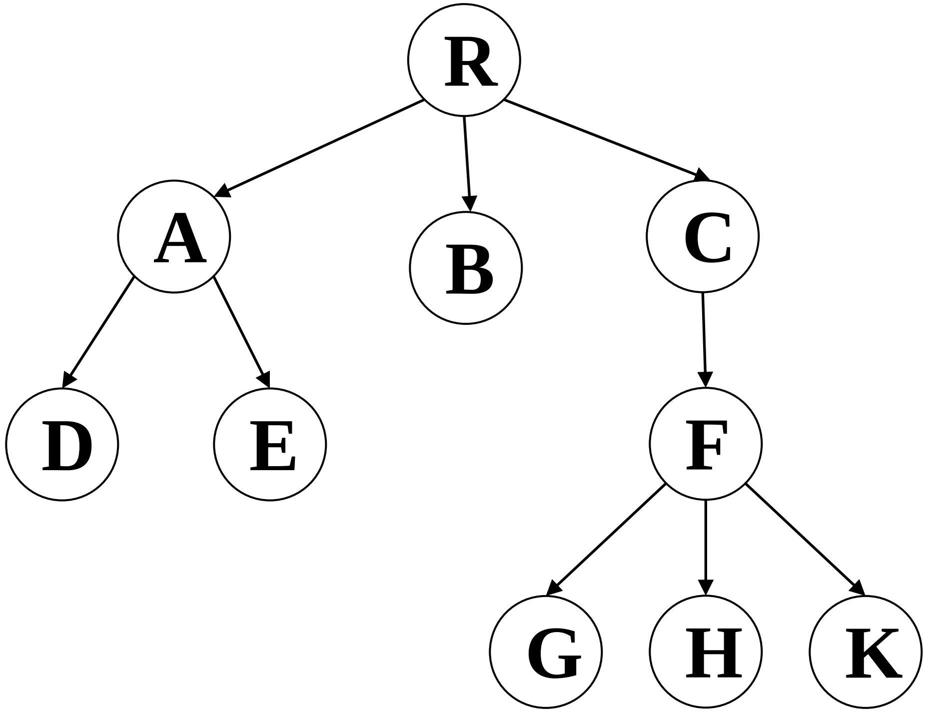
- 按层次遍历顺序对结点编号
- 编号为*i*的结点存放在数组的第*i*个位置
- 便于涉及孩子的操作,求父结点不方便
- 空间浪费。

树的先根序列及结点度表示法

树的先根遍历的定义

（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　R A D E B C F G H K

定理　如果已知一个树的先根序列和每个结点的度，则能唯一确定该树的结构。

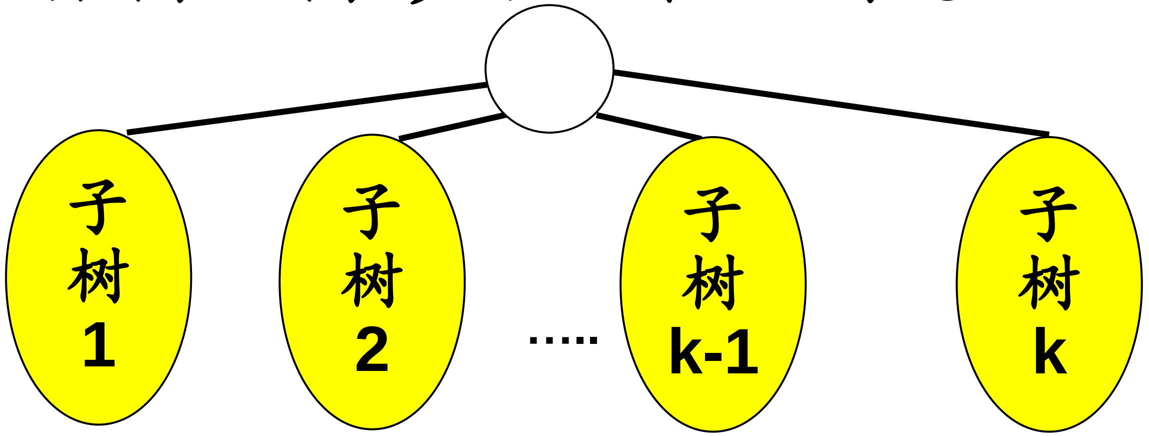
证明：用数学归纳法

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。

3. 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，设该结点的度为k，k≥1，因此根结点有k个子树。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。

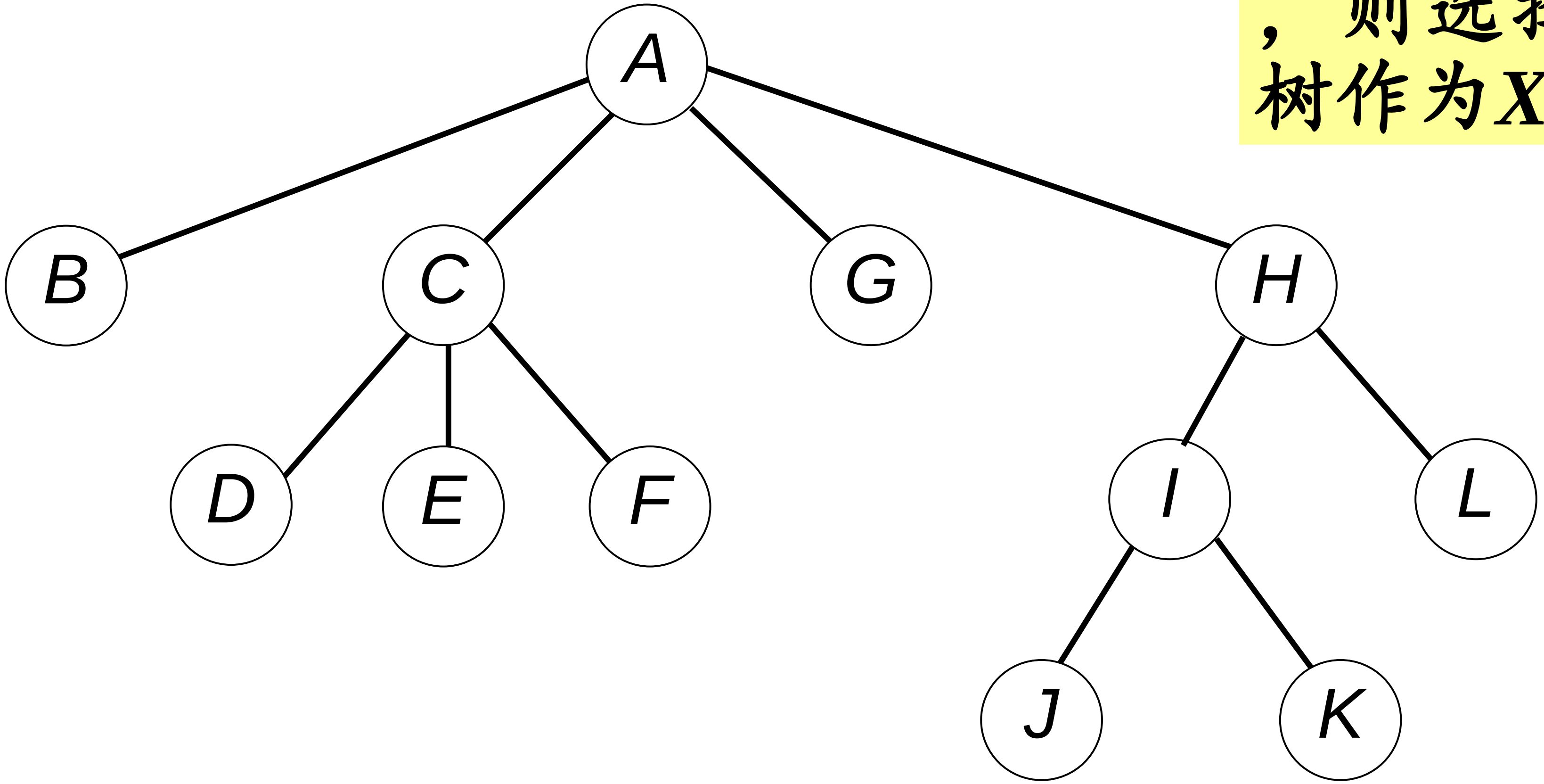
证毕。



先根序列: A B C D E F G H I J K L

结点度序列: 4 0 3 0 0 0 0 2 2 0 0 0

从右往左，结点X的度为k，则选择离X最近的k个子树作为X的子树

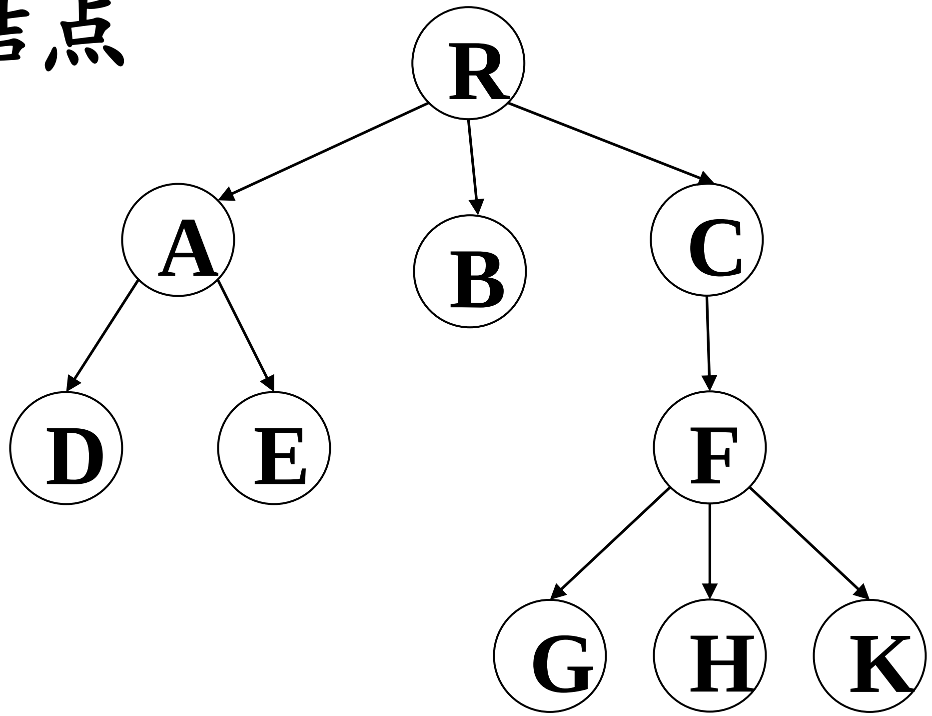


树的后根序列及结点度表示法

树的后根遍历的定义

（1）从左到右依次后根遍历根结点的诸子树

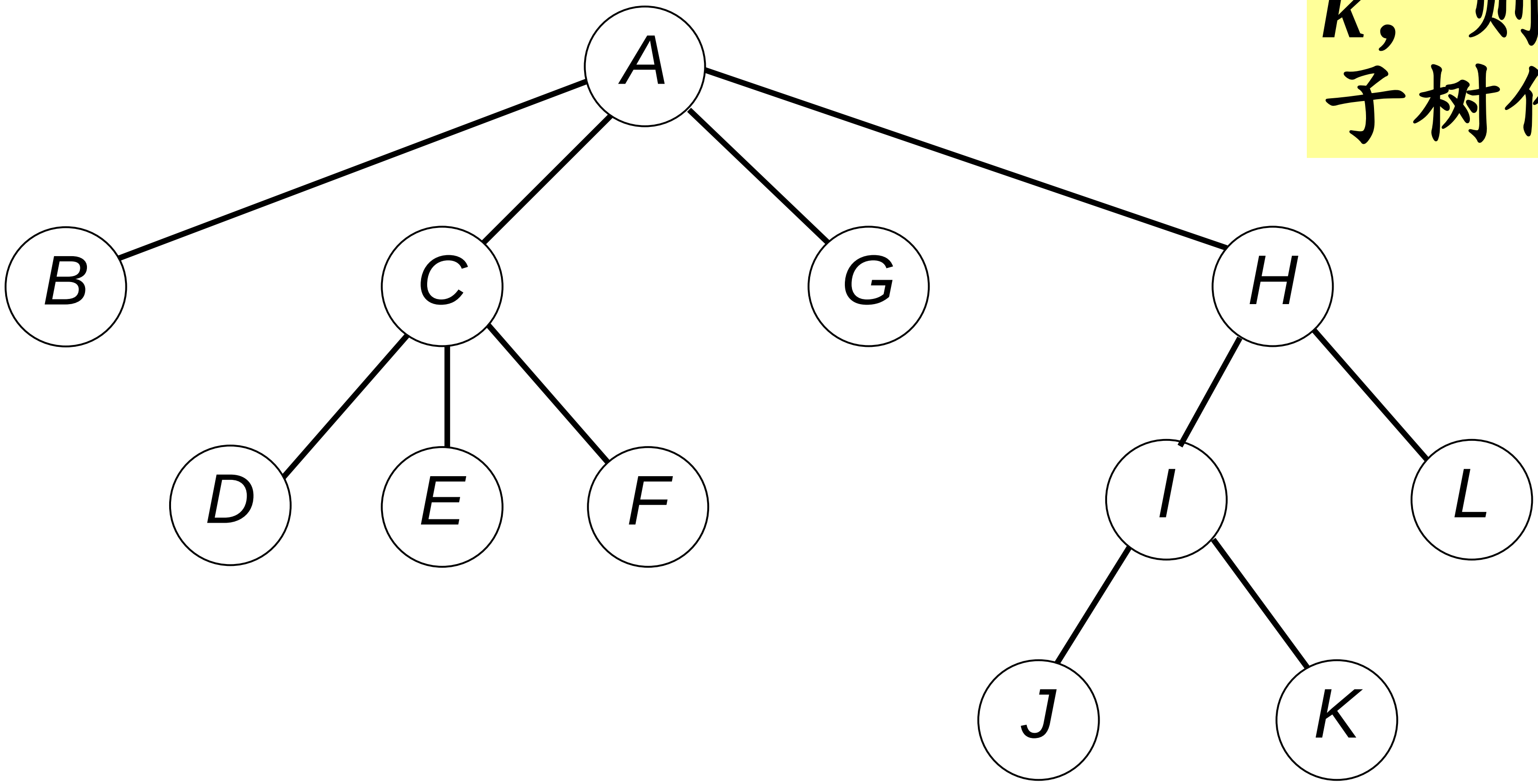
（2）访问根结点



后根序列　**D E A B G H K F C R**

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

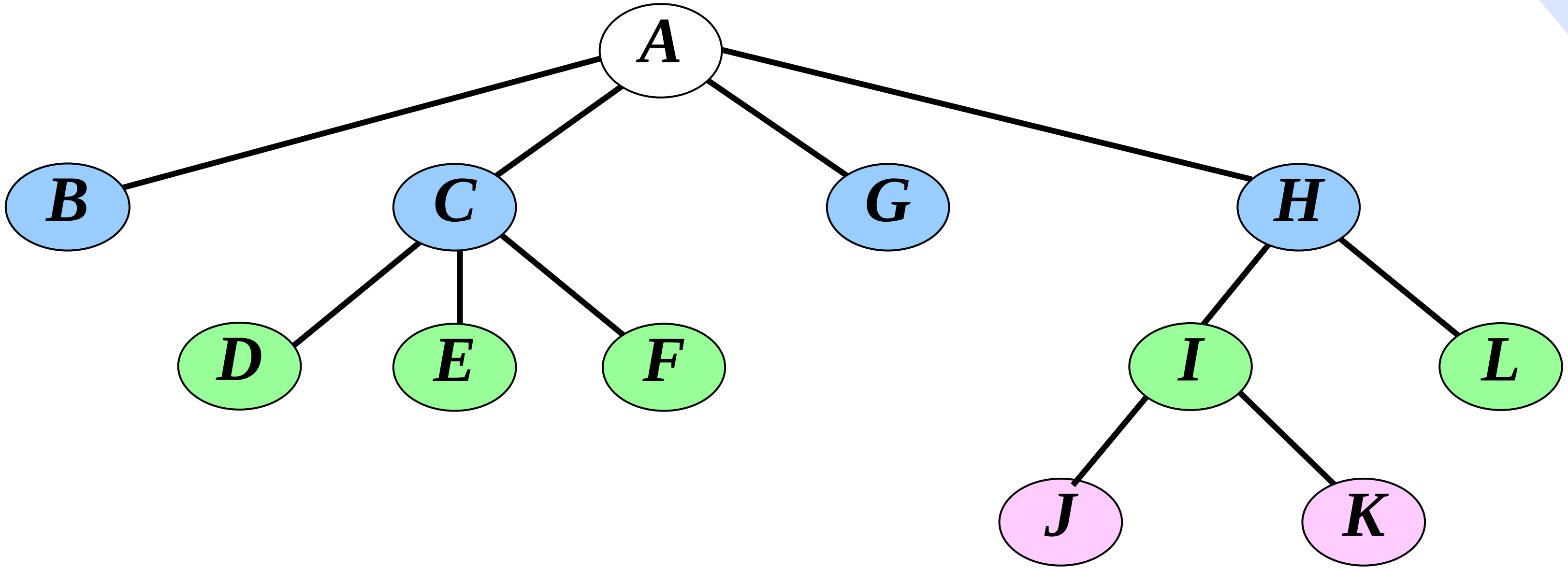
从左往右，结点*X*的度为*k*，则选择离*X*最近的*k*个子树作为*X*的子树



层次序列和结点度数表示法

由一棵树的层次序列和每个结点的度，能唯一确定树的结构。

层次序列	A	B	C	G	H	D	E	F	I	L	J	K
结点度	4	0	3	0	2	0	0	0	2	0	0	0

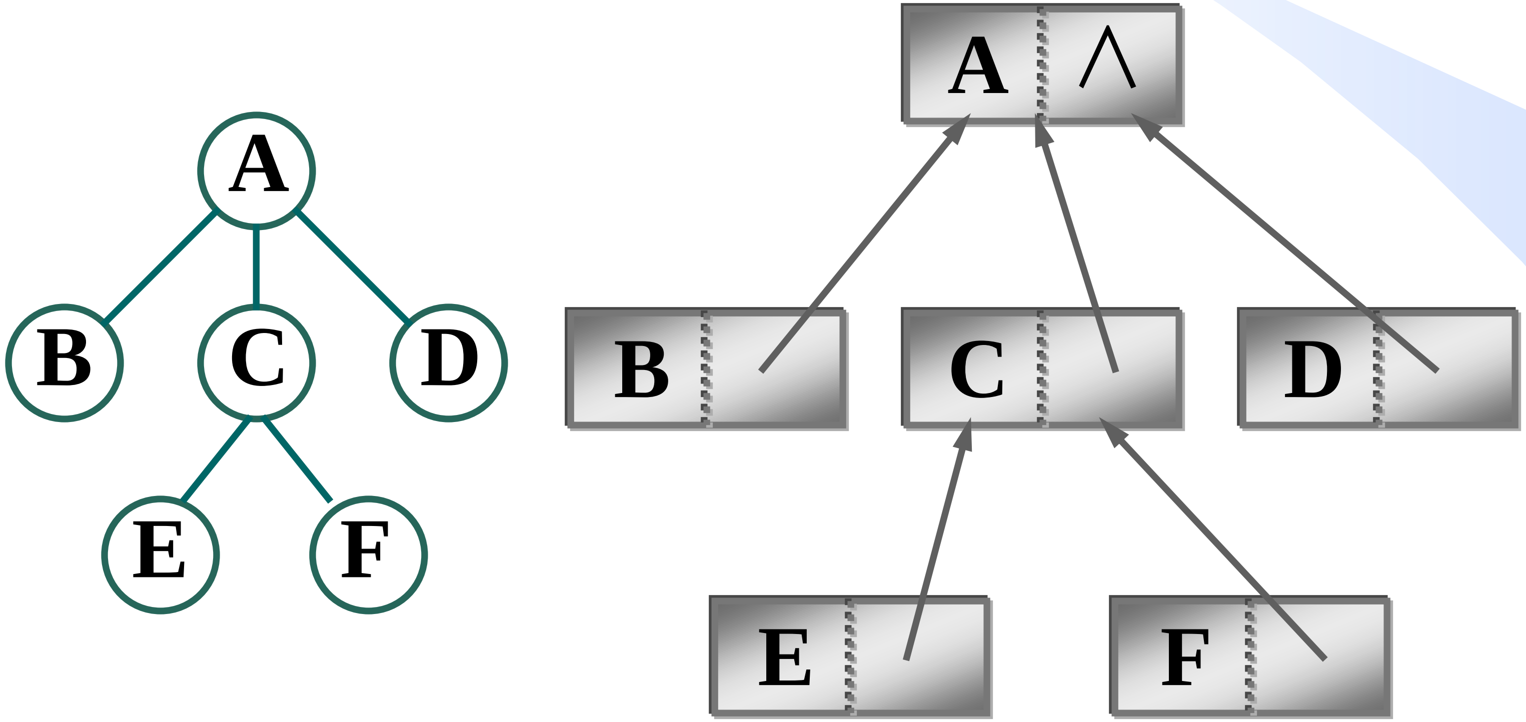


总结

- “遍历序列+结点度序列”顺序表示法不保存指针，节省空间，但牺牲了结点访问的便利性。
- 更适合作为树的输入输出，用于树的创建、压缩存储等。

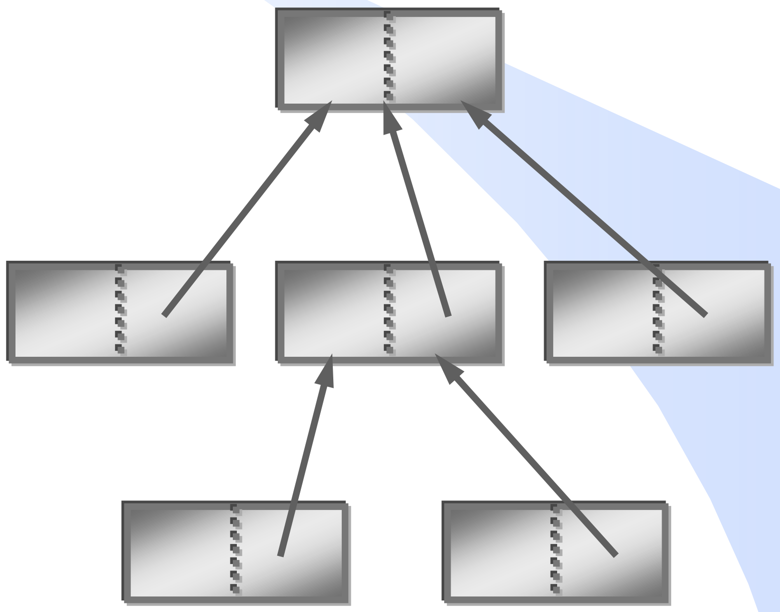
树的链接存储结构

(1) Father链接结构



Father 链接结构

每个结点有*Data* 和*Father* 两个域，*Father* 域存放指向父结点的指针。

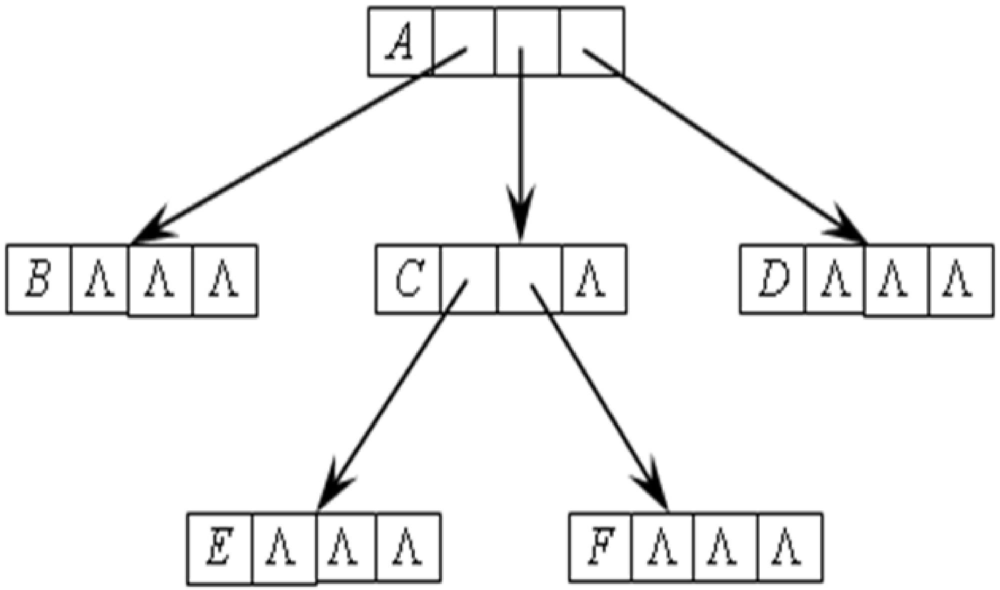


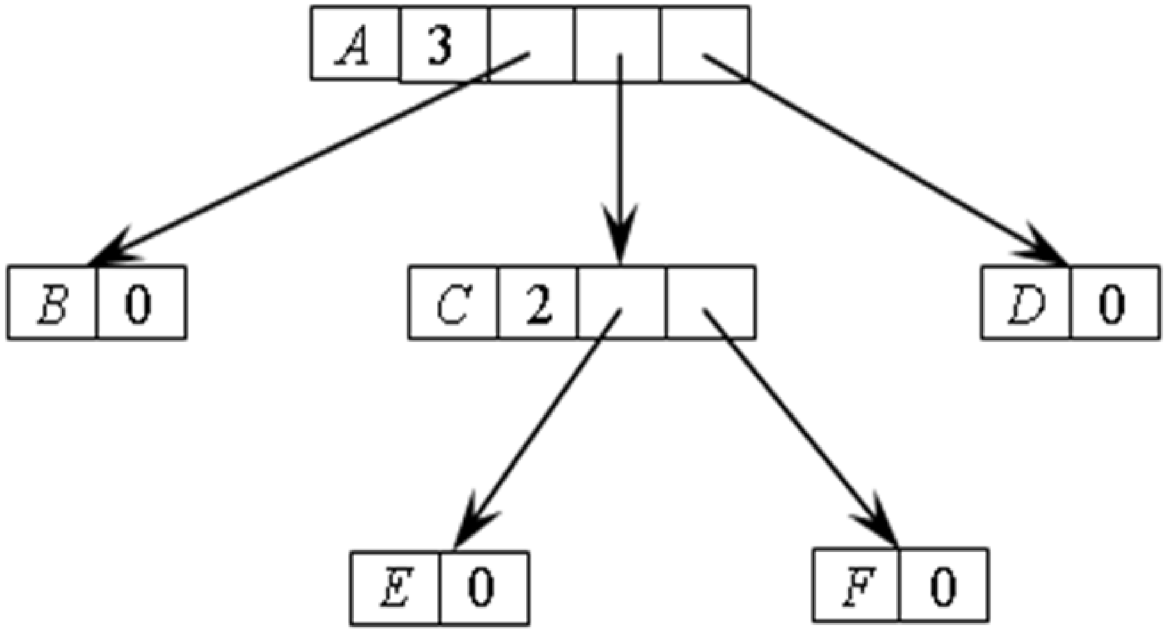
优点：找父结点方便。

缺点：

- ✓ 找子结点难；
- ✓ 很难确定一个结点是否为叶结点；
- ✓ 不易实现遍历操作。

孩子链表链接结构

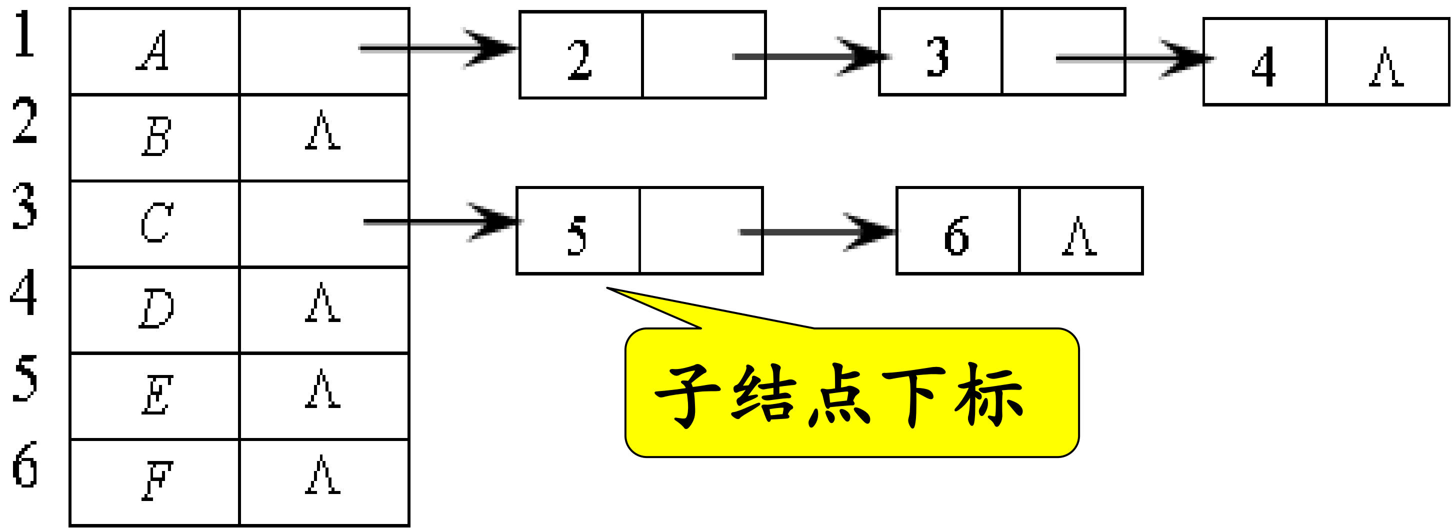


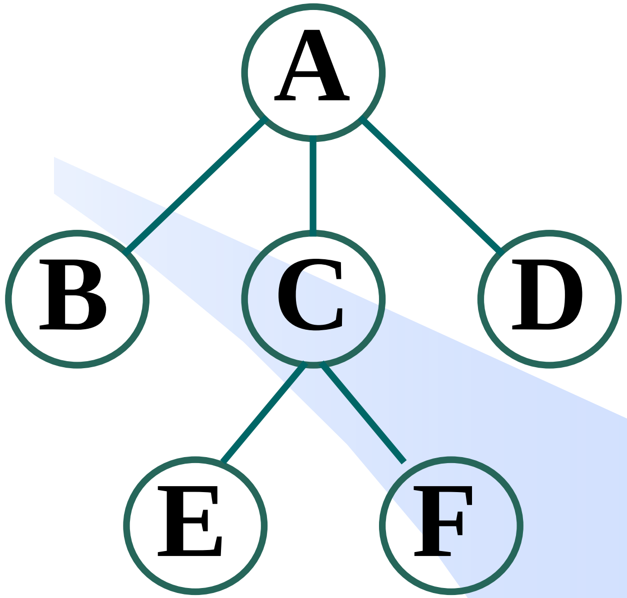


每个结点的指针数以整棵树中孩子最多的结点为准，大量指针为空，浪费空间

每个结点占据的存储空间不等，给操作和维护带来不便

孩子链表链接结构

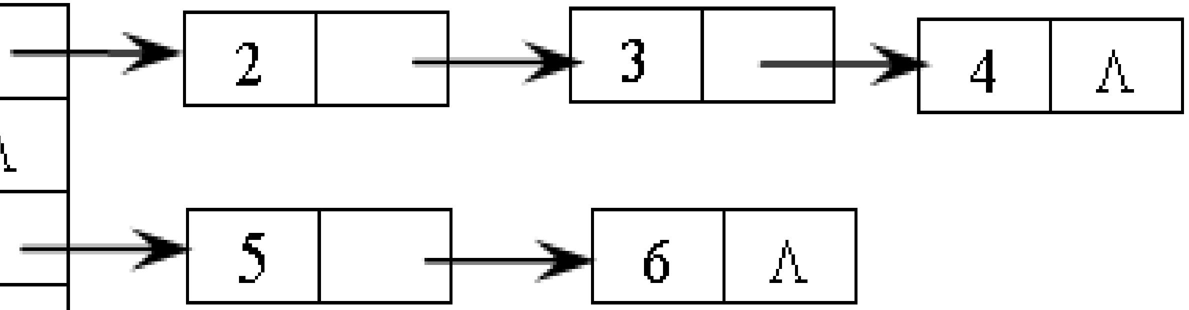




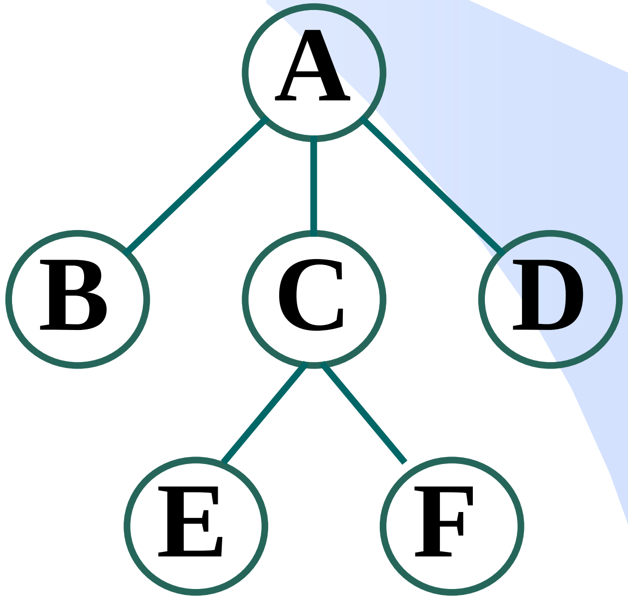
- ✓ 为树中每个结点设置一个孩子链表。
- ✓ 将这些结点及相应的孩子链表的头指针存放在一个数组中。
- ✓ 孩子结点的数据域存放它们在数组中的下标。
- ✓ 便于实现涉及子结点的操作，但不便于实现与父结点有关的操作。

父亲孩子链表链接结构

下标	结点数据	父亲下标	
1	A	0	
2	B	1	Λ
3	C	1	
4	D	1	Λ
5	E	3	Λ
6	F	3	Λ



父结点下标

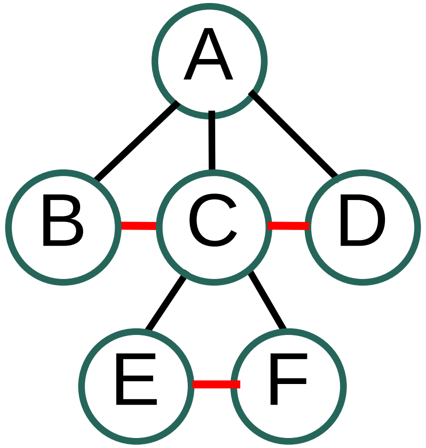


左孩子-右兄弟链接结构

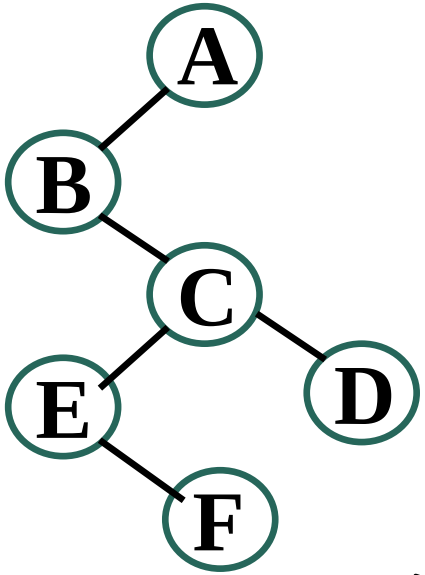
回顾：树与二叉树的转换

树结点的左孩子 ⇔ 二叉树结点的左孩子

树结点的右兄弟 ⇔ 二叉树结点的右孩子



树



二叉树

左孩子-右兄弟链接结构

二叉树结点结构

Left	Data	Right

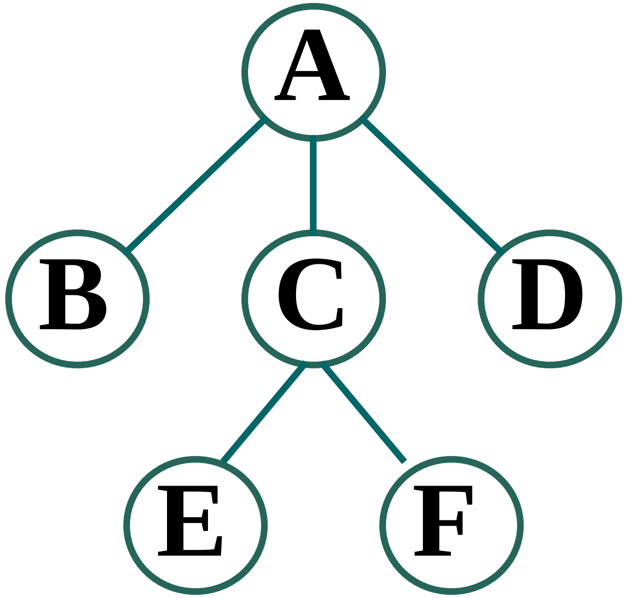
树结点的左孩子 ⇔ 二叉树结点的左孩子
树结点的右兄弟 ⇔ 二叉树结点的右孩子

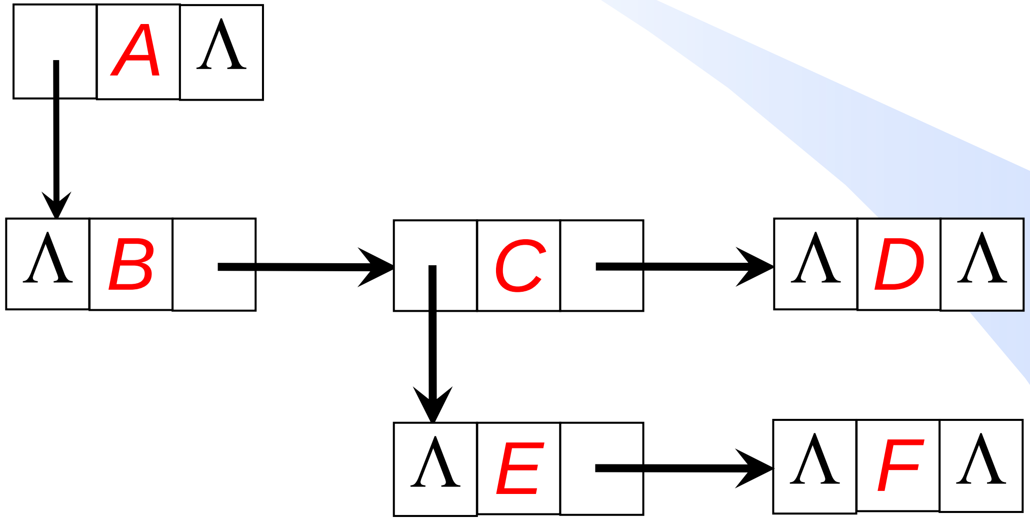
树结点结构

firstChild	Data	nextBrother

优点：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框架来实现对树的操作。

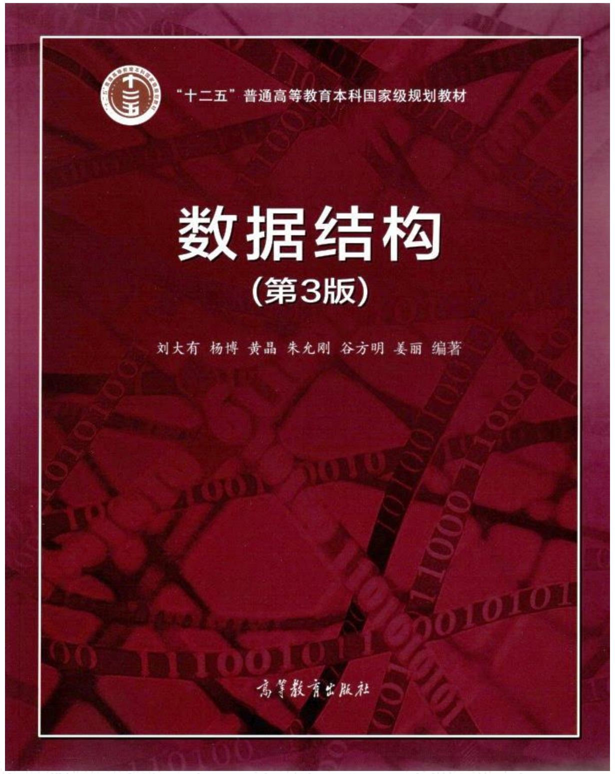
左孩子-右兄弟表示法示例：





结点结构

firstChild	Data	nextBrother



树的存储和操作

- 树与二叉树的转换
- 树的存储结构
- **树和森林的遍历**

数据之法
结构之美
算法之道

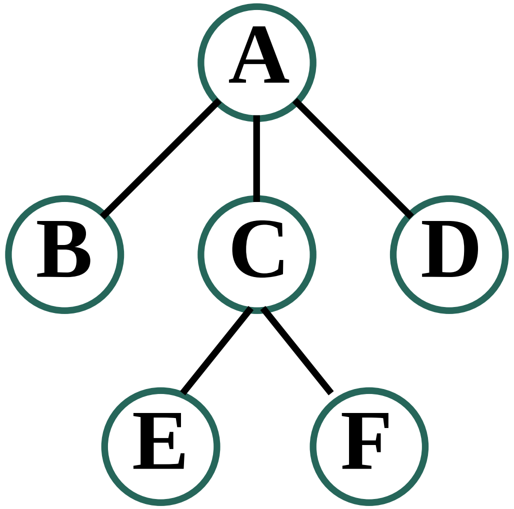
Last updated on 2022.10

zhuyungang@jlu.edu.cn

树的孩子-兄弟链接存储结构上的主要操作

① 遍历
② 搜索父结点
③ 搜索指定数据域的结点
④ 删除以给定结点为根的子树

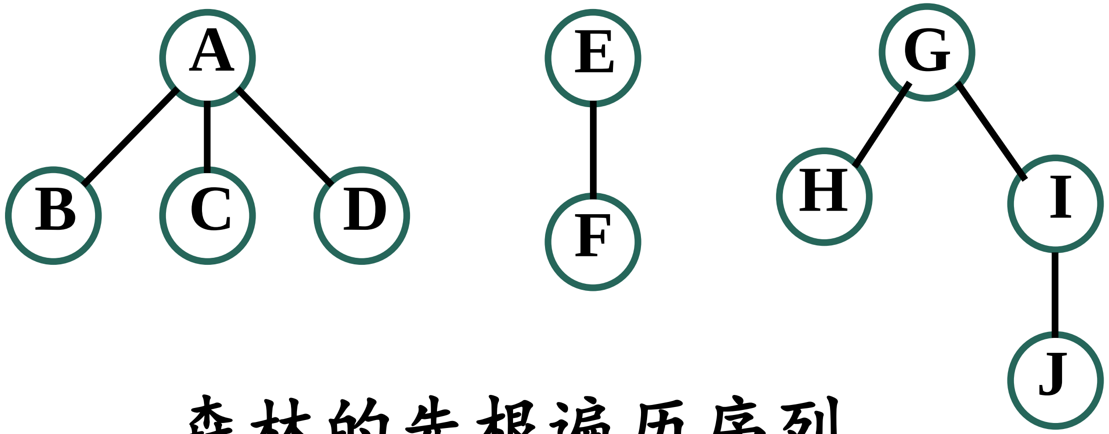
树的先根遍历：先访问树的根结点，然后依次先根遍历每棵子树。



先根遍历序列：A B C E F D

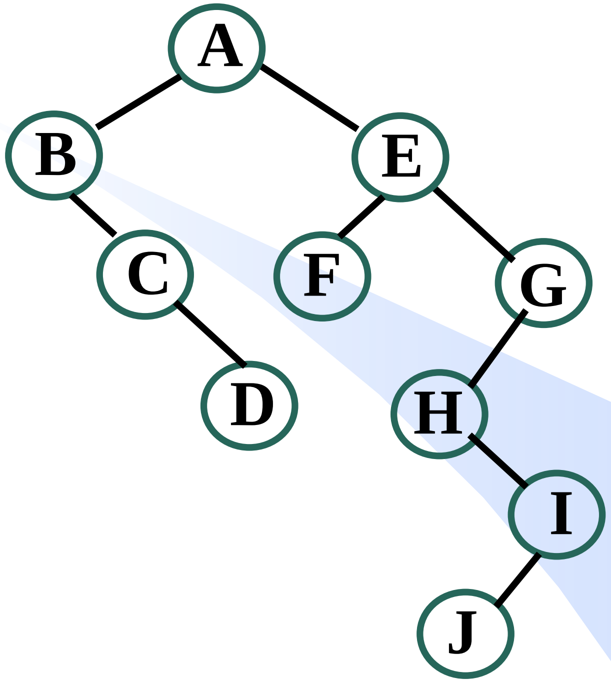
森林的先根遍历

① 访问森林中第一棵树的根结点；
② 先根遍历第一棵树中的诸子树；
③ 先根遍历其余的诸树。



森林的先根遍历序列

A B C D E F G H I J



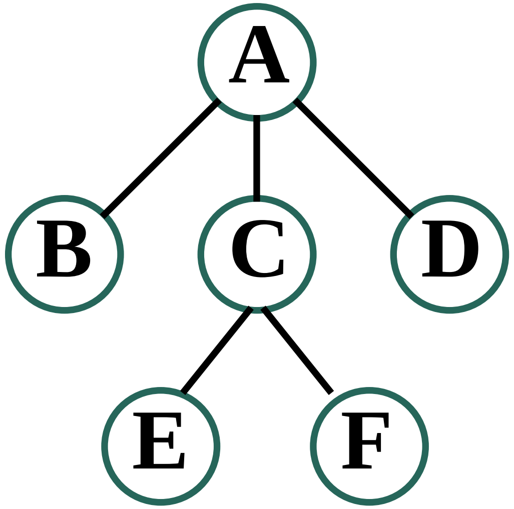
二叉树的先根序列

A B C D E F G H I J

森林的先根序列与它自然对应下的二叉树的先根序列相等

树的后根遍历：

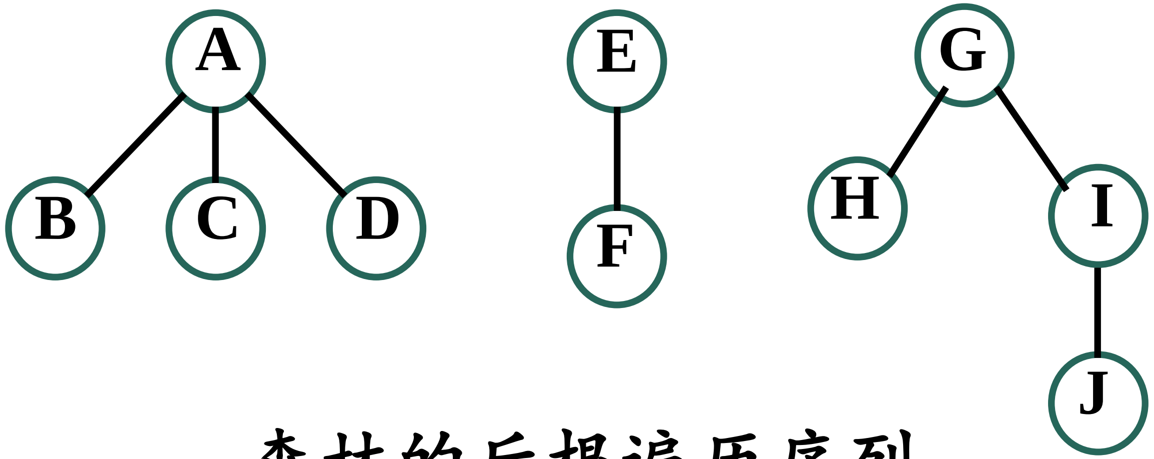
先依次后根遍历每棵子树，然后访问树的根结点。

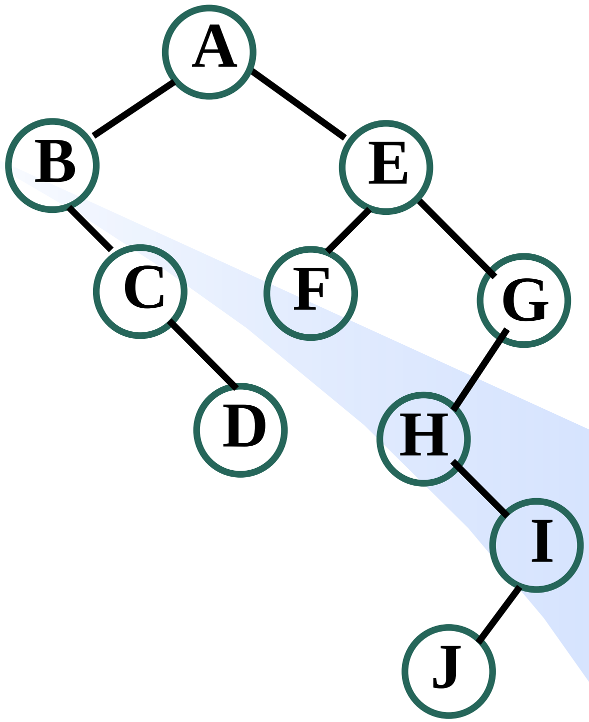


后根遍历序列：B E F C D A

森林的后根遍历

① 后根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。





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

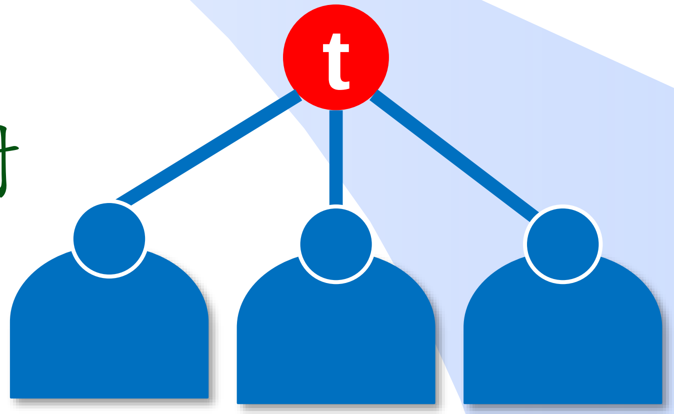
B C D A F E H J I G

森林的后根遍历序列与其对应的二叉树的中根遍历序列相等

树的左孩子-右兄弟表示法解决问题的常用框架

firstChild	Data	nextBrother

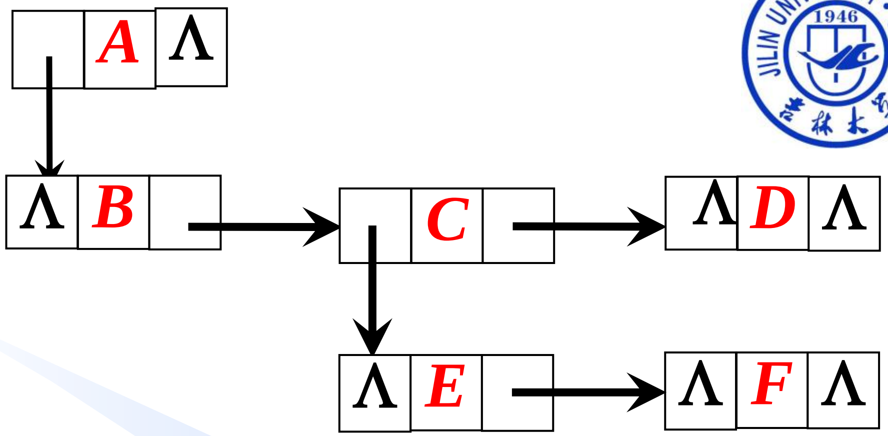
```
F(Node* t){
  …处理根结点…
  p=t->FirstChild;//通过指针p依次处理t的子树
  while(p!=NULL){
     F(p);           //处理以p为根的子树
     p=p->NextBrother; //p指向下一颗子树
  }
}
```

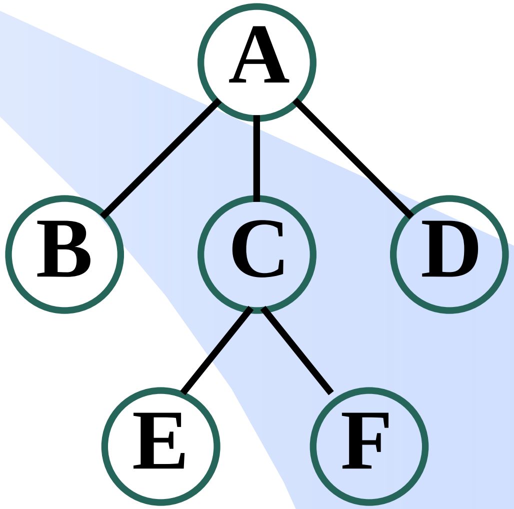



```
for(p=t->FirstChild; p!=NULL; p=p->NextBrother)
     F(p);
```

①先根遍历以t为根的树——递归算法

```
void PreOrder(Node* t){
   if(t==NULL) return;
   visit(t); //访问t
   Node* p=t->FirstChild; //找t第一个孩子
   while(p!=NULL){//先根遍历t的各子树
      PreOrder(p); //先根遍历以p为根的子树
      p=p->NextBrother;//令p指向t的下一个孩子
   }
}
```



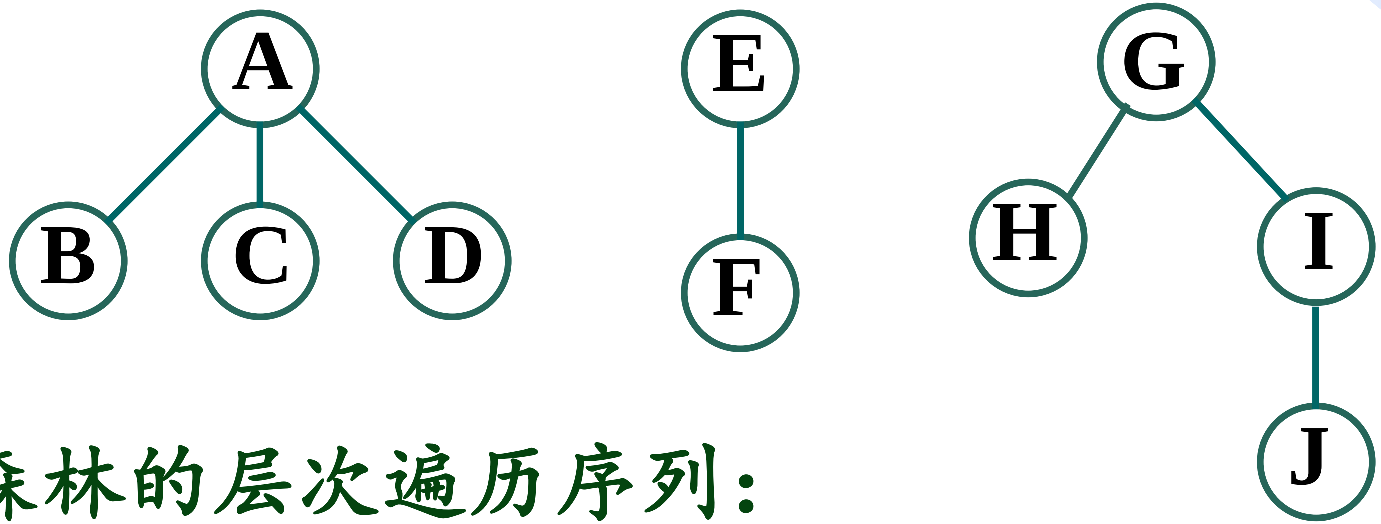



```
for(Node* p=t->FirstChild; p!=NULL; p=p->NextBrother)
     PreOrder(p);
```

树和森林的层次遍历

树和森林的层次次序遍历访问结点的次序：从第0层到最后一层，且同层结点的访问次序是从左到右。

[例]

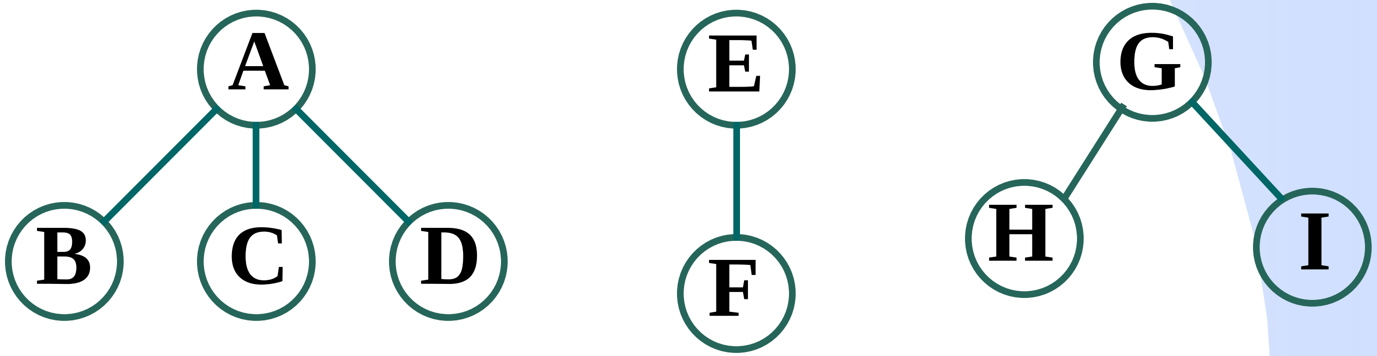


森林的层次遍历序列：

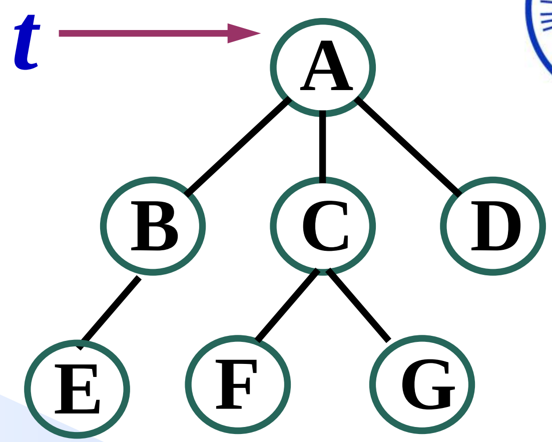
A E G B C D F H I J

层次遍历森林，指针t指向森林中第一棵树的根

```
void LevelOrder(Node* t){
   Queue Q; Node* p;
   for(p=t; p; p=p->NextBrother)
      Q.ENQUEUE(p);           //每棵树的根都入队
   while(!Q.IsEmpty()){
      p=Q.DEQUEUE();          //出队一个结点p
      visit(p);               //访问p
      for(p=p->FirstChild; p; p=p->NextBrother)
         Q.ENQUEUE(p);        //p的每个孩子都入队
   }
}
```



搜索指定数据域的结点

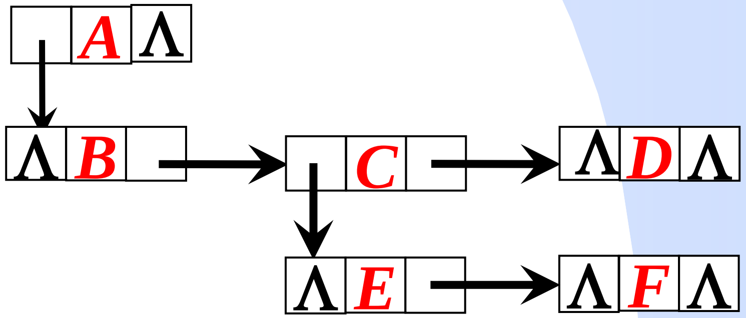


FindTarget (*t* , *target*)

在以t为根的树中找数据值等于target的结点

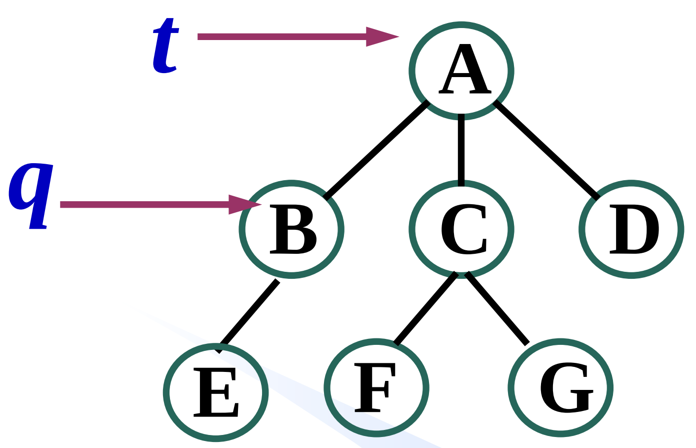
- ❖若t的数据域为target，则返回t;
- ❖否则，p指向t的左孩子，在以p为根的树中递归查找;
- ❖若未找到，p指向其右兄弟，继续进行上述查找，直到找到数据域等于target的结点或p为空。

```
Node* FindTarget(Node* t, int target){
    if(t==NULL) return NULL; //树空
    if(t->data==target) return t; //t即为所求
    Node* p=t->FirstChild; //p指向t的左孩子
    while(p!=NULL){
        Node* ans=FindTarget(p,target);//在子树p中找
        if(ans!=NULL) return ans; //若找到则返回
        p=p->NextBrother;//若未找到，则在下棵子树里找
    }
    return NULL; //整个树中未找到
}
```



```
Node* FindTarget(Node* t, int target){
    if(t==NULL) return NULL;
    if(t->data==target) return t;
    Node *p, *ans;
    for(p=t->FirstChild; p; p=p->NextBrother){
        ans = FindTarget(p,target);
        if(ans!=NULL) return ans;
    }
    return NULL;
}
```

搜索父结点



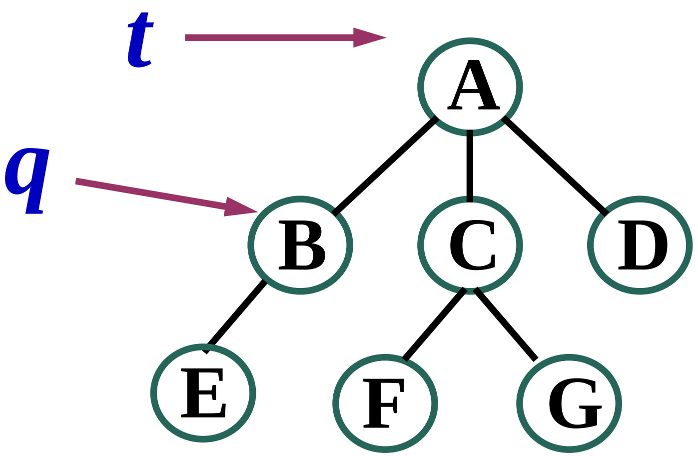
FindFather(t, p)

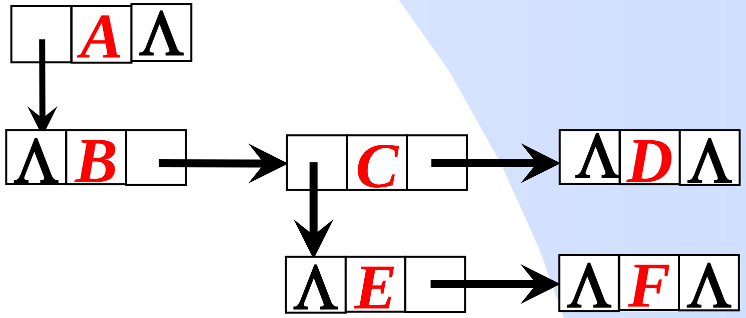
在以t为根的树中找p的父结点

算法思想：

- 令q指向t的左孩子，若q=p，t就是p的父结点；否则，在以q为根的子树中找p的父结点，即 FindFather(q, p);
- 若未找到，令q指向t的下一个孩子，即q的右兄弟，重复上述步骤，直至找到p的父结点或扫描完t的所有子结点。

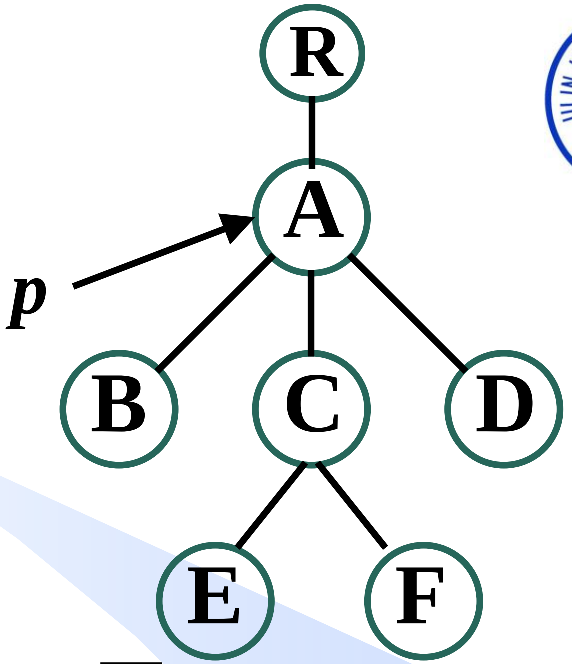
```
Node* FindFather(Node* t, Node* p){
   //在以t为根的树中找p的父结点
   if(t==NULL || p==NULL || p==t) return NULL;
   for(Node* q=t->FirstChild; q; q=q->NextBrother){
     //通过q循环访问t的每棵子树
     if(q==p) return t; //若q=p，则p的父结点就是t
     Node* ans=FindFather(q,p); //在以q为根的子树中找p的父亲
     if(ans!=NULL) return ans;
   }
   return NULL;
}
```

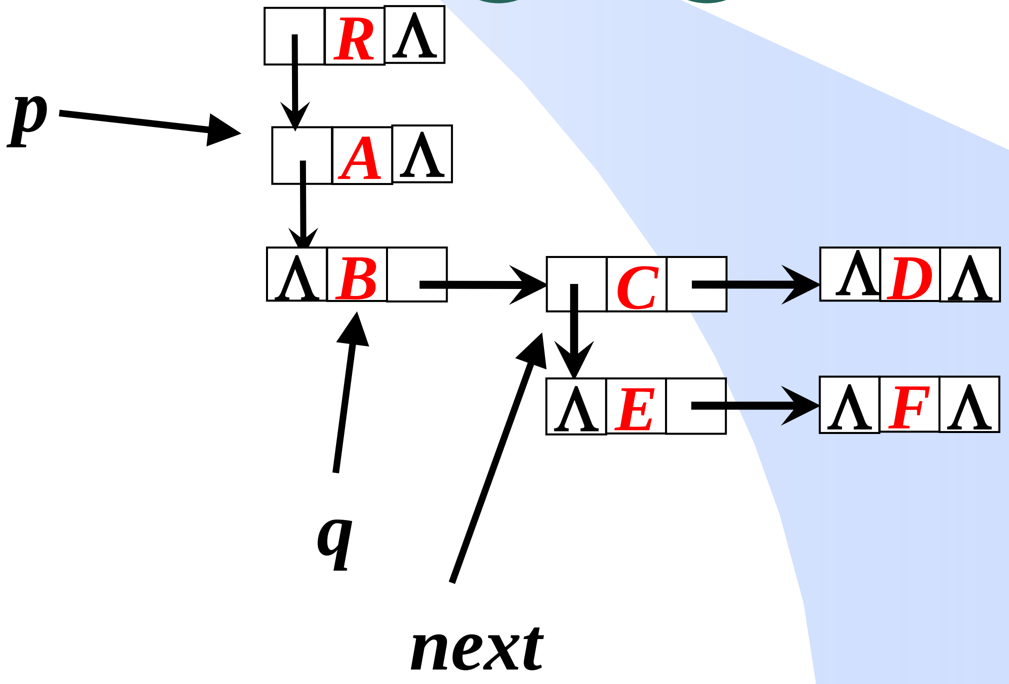




释放树空间

```
void Del(Node* p){ //释放根为p的子树
   if(p==NULL) return;
   //从左到右删除p的子树
   Node* q=p->FirstChild;
   while(q!=NULL){
     Node* next=q->NextBrother;
     Del(q);
     q=next;
   }
   delete p;
}
```





删除子树

```
void DelSubTree(Node *t,Node *p ){
  //在以t为根的树中删除以p为根的子树
  if(t==NULL || p==NULL) return;
  Node* fa = FindFather(t,p);
  if(fa==NULL) {Del(p); return;} //若p无父亲, 直接删p
  if(fa->FirstChild == p){//若p有父亲, 且是父亲的左孩子
     fa->FirstChild=p->NextBrother; Del(p); return;
  } //若p有父亲, 但不是父亲的左孩子
  Node* q=fa->FirstChild;
  while(q->NextBrother!=p)
     q=q->NextBrother; //找p的左侧兄弟
  q->NextBrother = p->NextBrother;
  Del(p);
}
```