数据结构

Data Structure

张先宜

手机：18056307221 13909696718

邮箱：zxianyi@163.com

QQ：702190939

QQ群： XC数据结构交流群 **275437164**

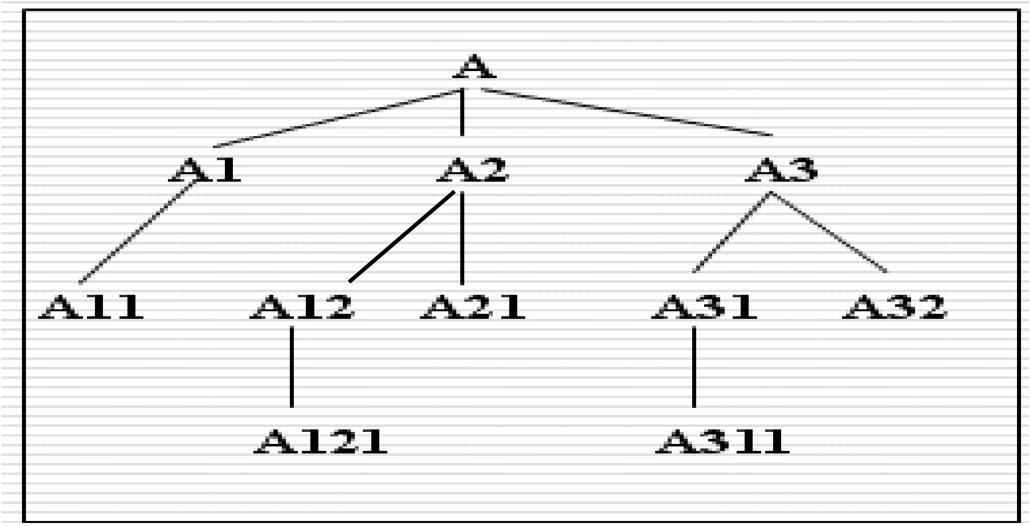
第5章 树（Tree）

【本章内容】

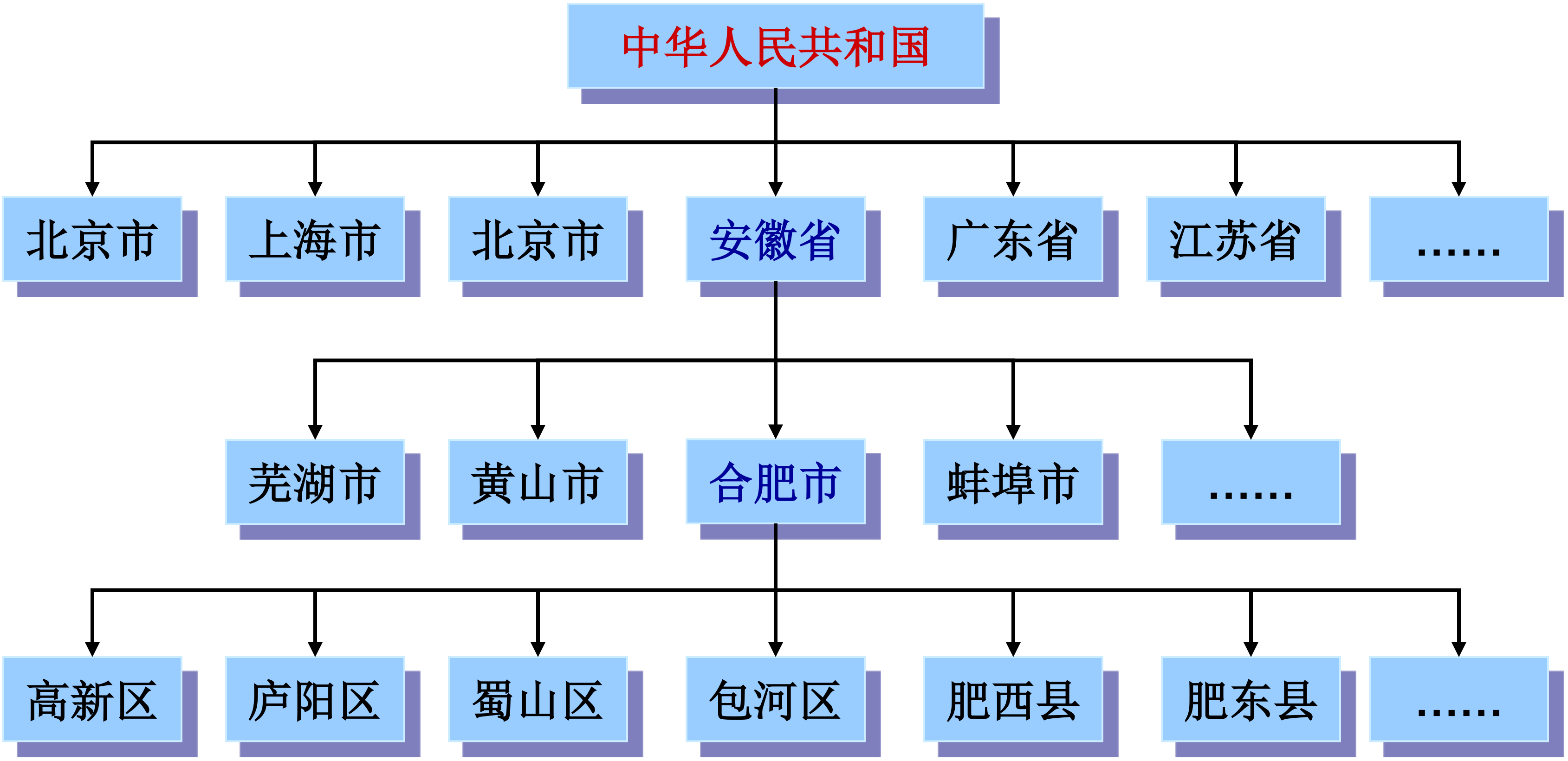
5.1 树的基本概念和术语

- 树是一类重要的非线性数据结构，以二叉树最为常用；
- 树反映元素之间的层次、分支结构关系，类似自然界的树；
- 树型结构的应用：
  - 家族的族谱;
  - 各种社会组织结构;
  - 计算机磁盘文件的组织;
  - **Internet 的域名解析系统 DNS;**
  - …

■ 家族关系图

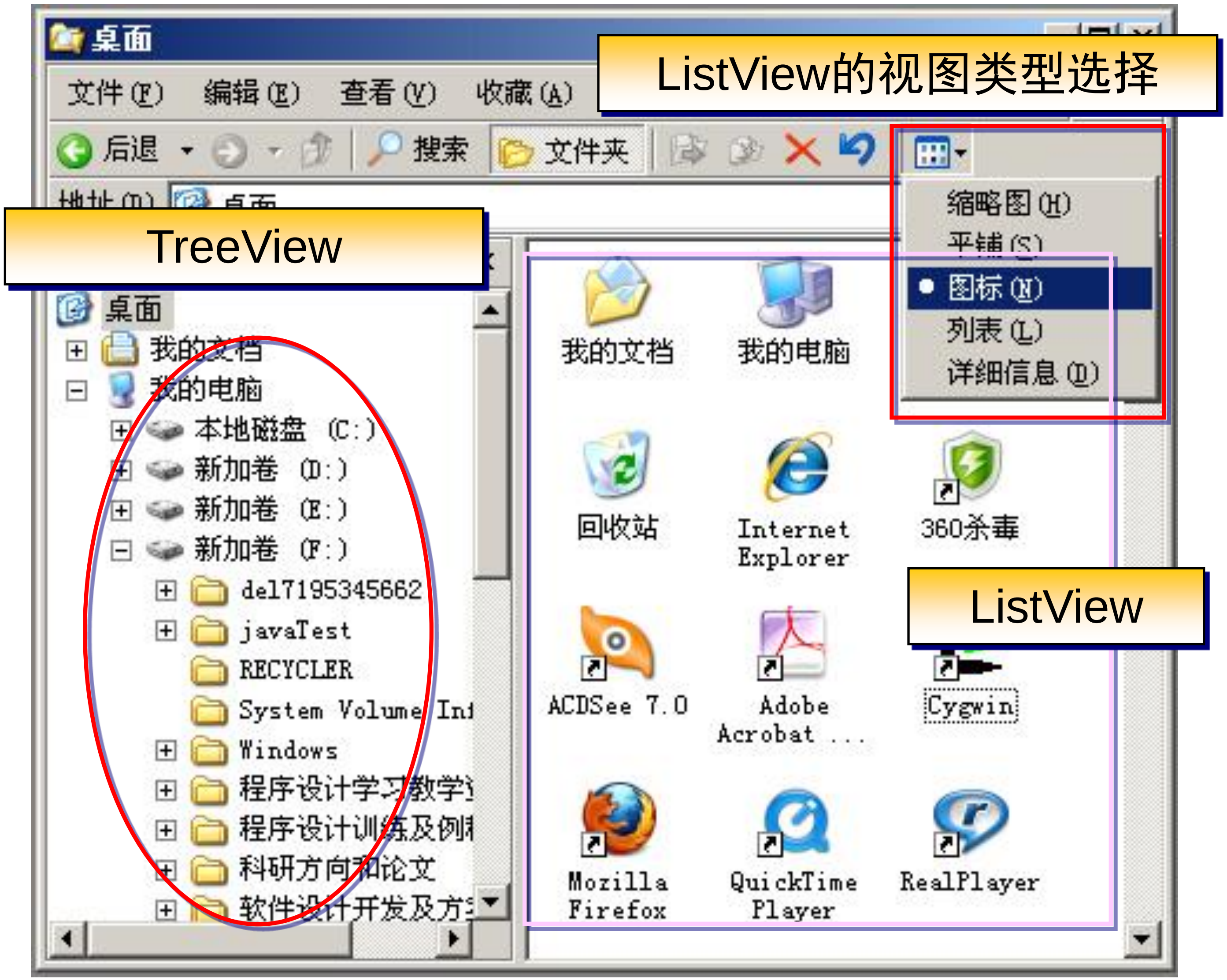


国家行政组织结构图



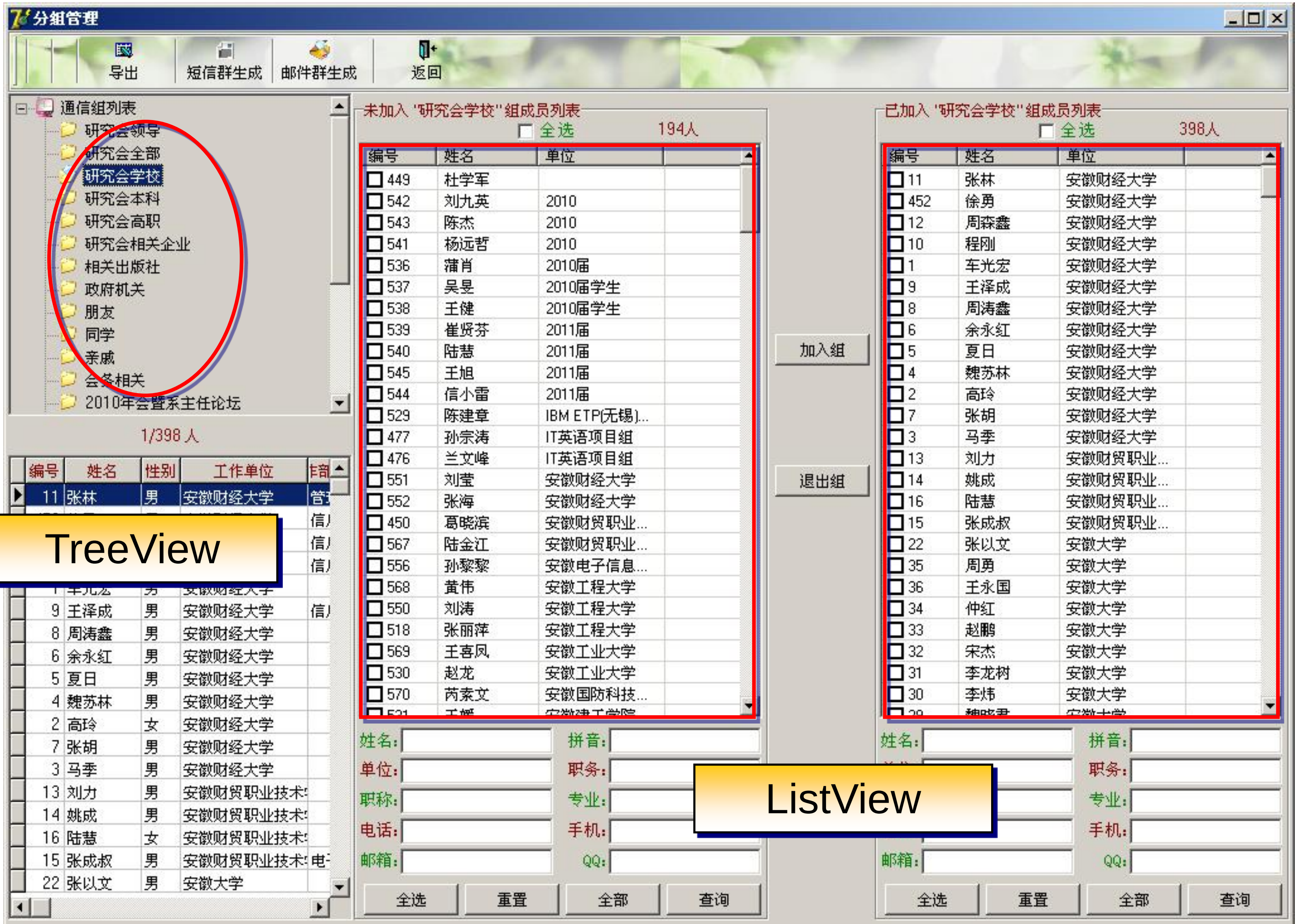
中华人民共和国
北京市
上海市
北京市
安徽省
广东省
江苏省
……
芜湖市
黄山市
合肥市
蚌埠市
……
高新区
庐阳区
蜀山区
包河区
肥西县
肥东县
……

■ 磁盘文件组织



桌面
文件(F) 编辑(E) 查看(V) 收藏(A)
后退 搜索 文件夹
ListView的视图类型选择
缩略图(H)
平铺(S)
图标(N)
列表(L)
详细信息(D)
TreeView
桌面
我的文档
我的电脑
本地磁盘 (C:)
新加卷 (D:)
新加卷 (E:)
新加卷 (F:)
del7195345662
javaTest
RECYCLER
System Volume Inf
Windows
程序设计学习教学
程序设计训练及例
科研方向和论文
软件设计开发及方
我的文档
我的电脑
回收站
Internet Explorer
360杀毒
ListView
ACDSee 7.0
Adobe Acrobat ...
Cygwin
Mozilla Firefox
QuickTime Player
RealPlayer

■ 应用软件中使用



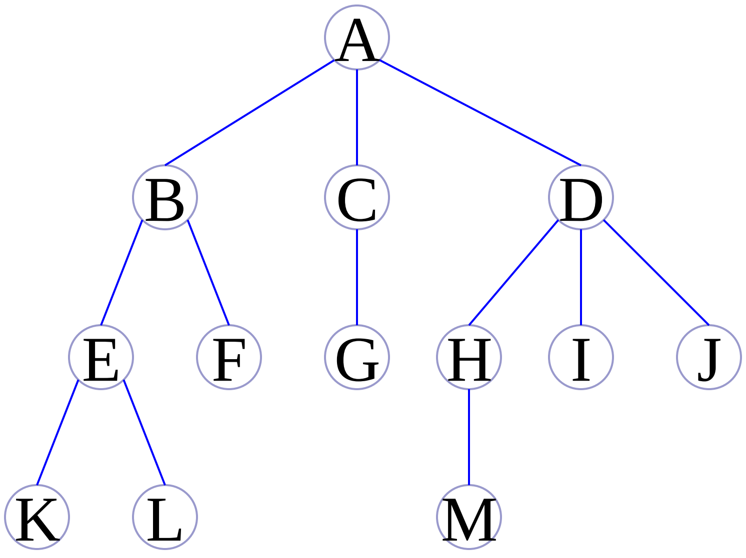
分组管理
导出
短信群生成
邮件群生成
返回
通信组列表
研究会领导
研究会全部
研究会学校
研究会本科
研究会高职
研究会相关企业
相关出版社
政府机关
朋友
同学
亲戚
会务相关
2010年会暨系主任论坛
1/398 人
未加入 "研究会学校" 组成员列表
全选
194人
已加入 "研究会学校" 组成员列表
398人
加入组
退出组
TreeView
ListView
姓名:
拼音:
单位:
职务:
职称:
专业:
电话:
手机:
邮箱:
QQ:
全选
重置
全部
查询

5.1.1 树的定义

- **树（Tree）的定义**
  - **树T是由n个结点组成的有限集合（n > 0）。**
  - **其中有且仅有一个根结点（树根），**
  - **其余结点可划分成m个（m>=0）互不相交的子集T_1, T_2,…,T_m,**
  - **且这些子集也分别构成树——子树**
  - **说明**
    - 树的定义是递归的。即树由子树构成，子树又由更小的子树构成。树和子树有相同的组织方式。
    - 这个定义，树至少有一个结点，没定义空树，少数教材有空树概念。

树结构的表示方法

① **图形表示法**：圆点表示结点，标注结点的值；无向或有向线段连接结点，表示结点的关系。如下图所示：

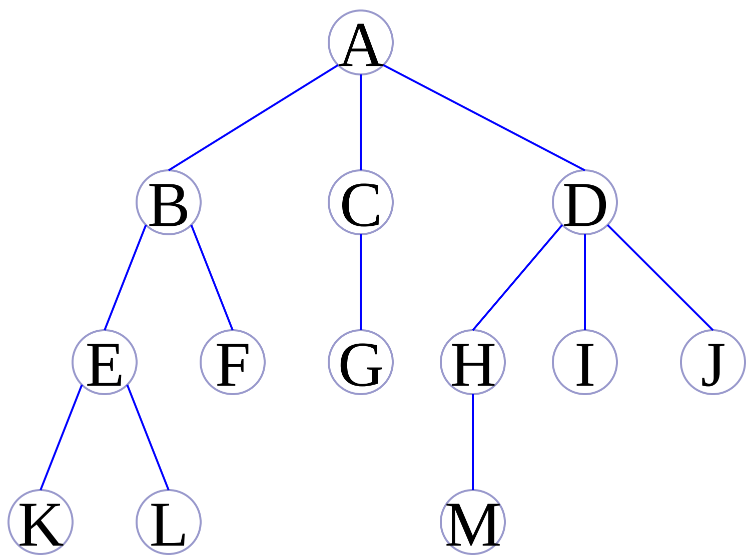


② 嵌套集合表示法：{根结点, {子树1},{子树2},…,{子树m}}。对子树用相同方法表示。

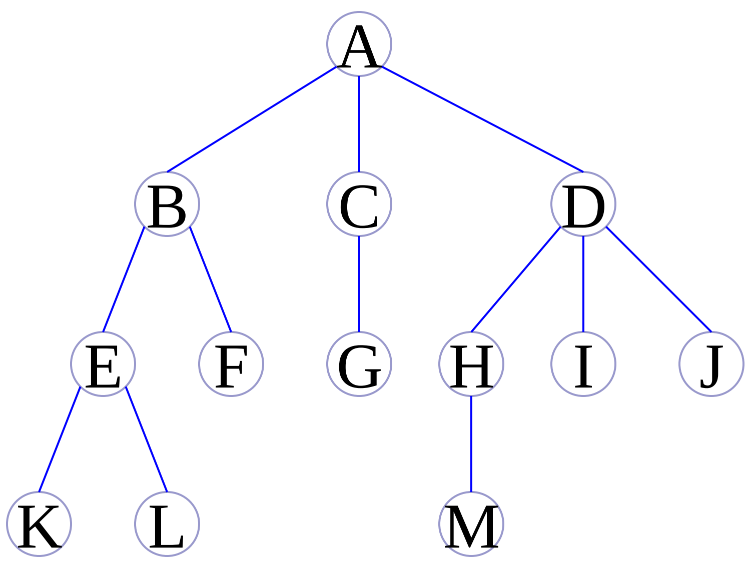
{A,{B,{E,{K},{L}},{F}},{C,{G}},{D,{H,{M}},{I},{J}}}

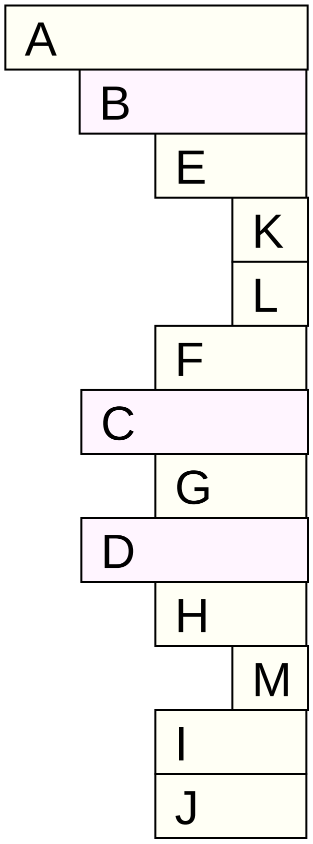
③ 广义表表示：树根作为表头元素，每个子树作为一个子表元素，对子树按照同样方法表示。

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



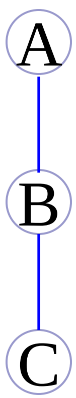
④ **凹入表表示：**如下图所示，结点的层次越深，凹入越多。



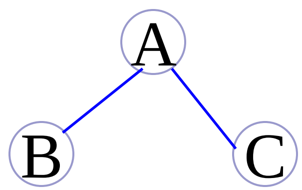


■ 树结构示例：

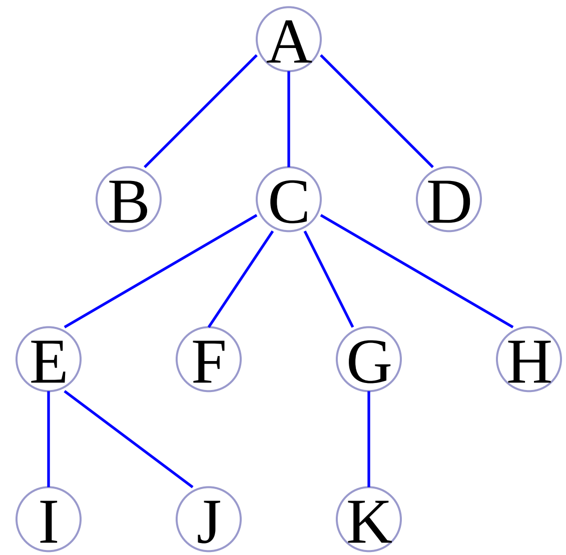
(a) 只有根结点的树



(b) 有一个子树的树

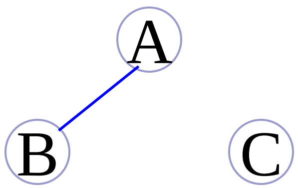


(c) 有2个子树的树

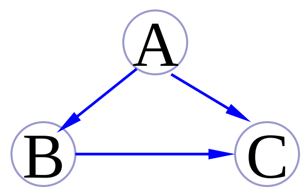


(d) 有多个子树的树

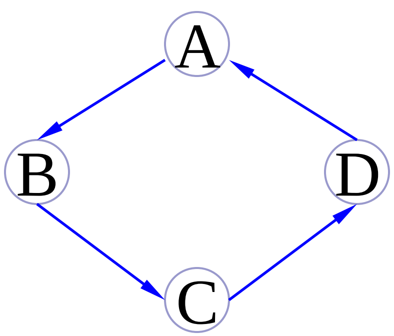
非树结构示例:



(a) 非树，因有 2 个根结点



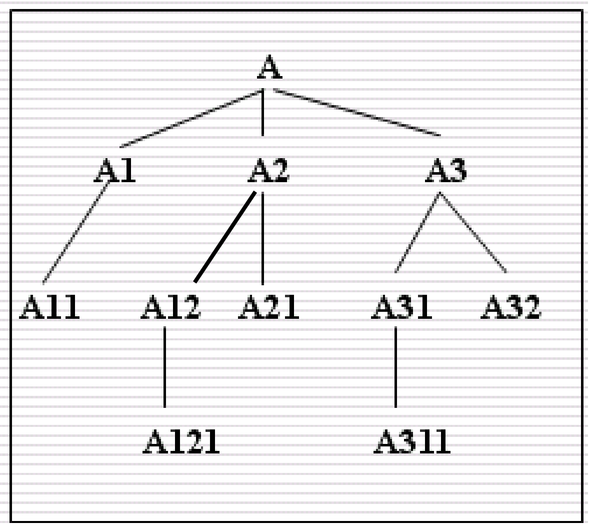
(b) 非树，有闭合路径



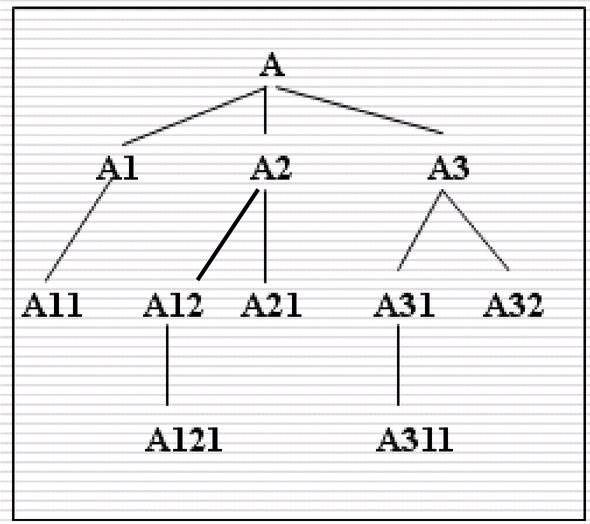
(c) 非树，有闭合路径

5.1.2 树的基本概念

- **1. 结点（节点）**
  - 包含数据域，存放数据元素；
  - 指针域，存放若干指针，指向其上、下层结点。
- **2. 结点的度**
  - 结点拥有的子树数目。
- **3. 叶结点（终端结点）**
  - 度为 0 的结点，
  - 或没有子树的结点。
- **4. 分支结点（非终端结点）**
  - 度不为 0 的结点，
  - 或有子树的结点。



- **5. 树的度**
  - 树内各结点的度的最大值。



【以下为描述结点关系的术语】

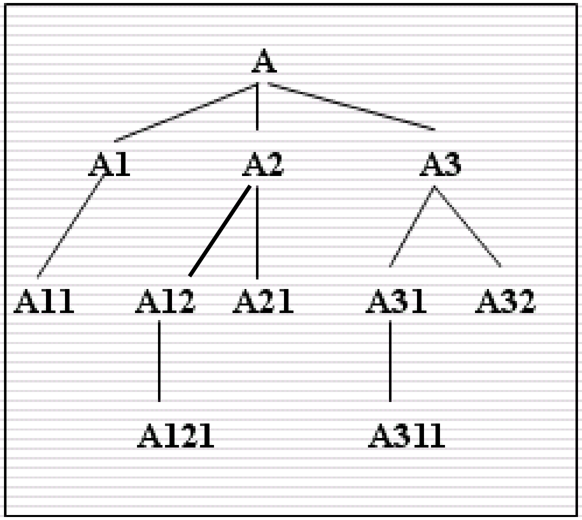
- **6. 孩子结点（子结点、直接后继结点）**
  - 与当前结点有边(edge)直接相连的下一层结点，叫做当前结点的孩子结点、子结点。
  - 子树的根结点
  - 叶子结点没有孩子结点。

- 7. 父结点（双亲结点、直接前驱结点）
  - ☞ 与当前结点有边(edge)直接相连的上一层结点，叫做当前结点的双亲结点、父结点。
  - ☞ 树中根结点没有双亲结点；其它结点有且仅有一个双亲结点。
- 8. 祖先（前驱）结点-- ancestor
  - ☞ 从根结点有路径到达当前结点，路径经过的所有结点，都是当前结点的祖先结点、先驱结点。
- 9. 子孙（后裔）结点-- descendant
  - ☞ 当前结点作为根结点，其子树上的所有结点，都是当前结点的后裔结点。

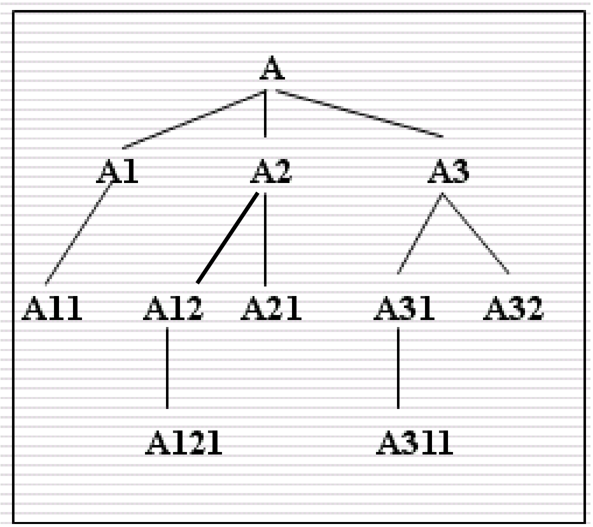
- **10. 兄弟结点(Sibling)**
  - 双亲结点相同的所有结点互称为兄弟结点。
- **11. 堂兄弟结点**
  - 双亲结点在同一层次（深度相同）的结点，互为堂兄弟。

【以下为描述树的层次术语】

- **12. 结点的层次（深度）-- level**
  - 根结点的层次为 1；(也有设为 0 的
  - 其它结点层次等于父结点层次加 1。
- **13. 树的高度/深度--depth**
  - 整个树中结点的最大层次



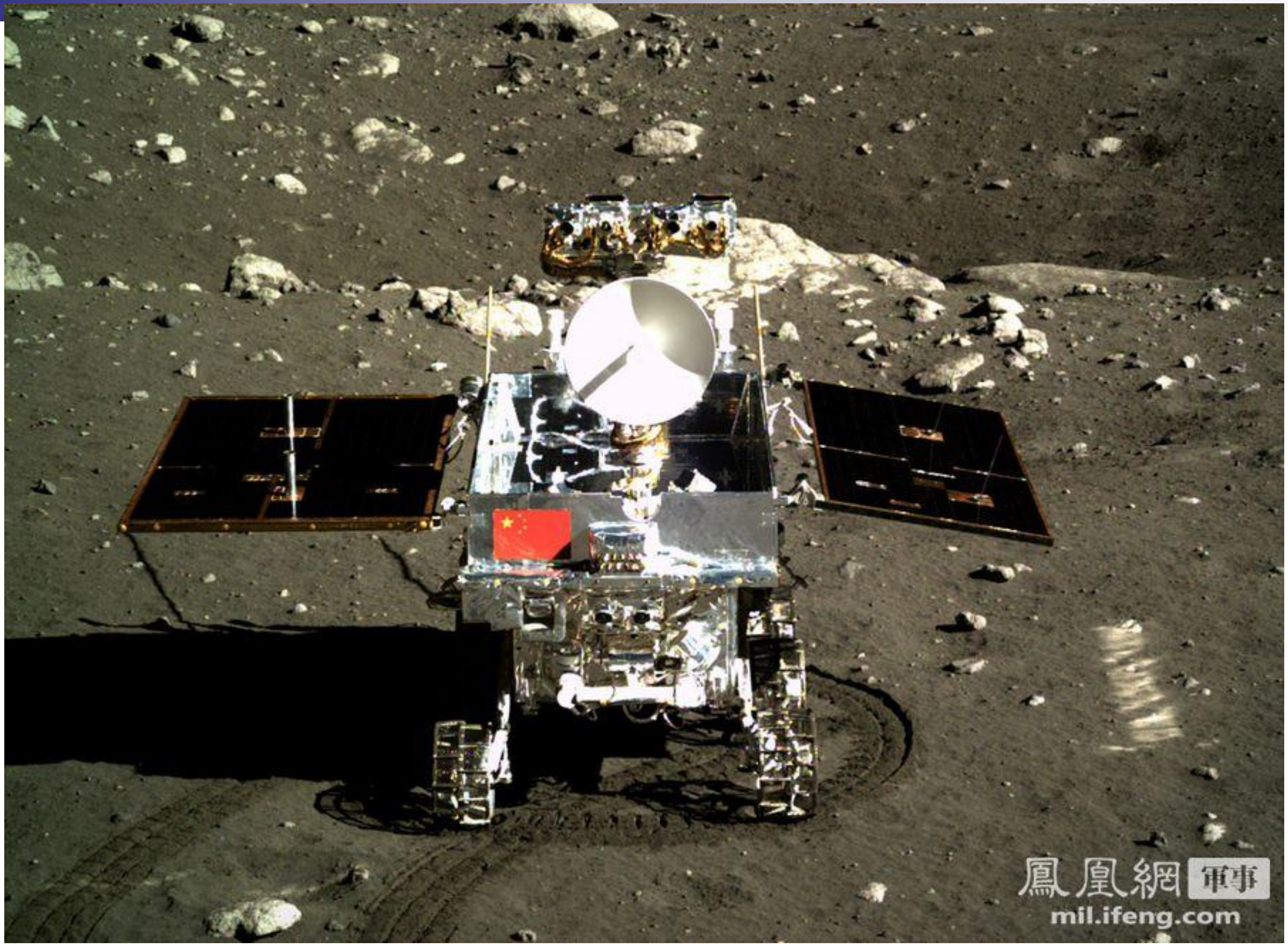
- **14. 有序树**
  - ☞ 同一结点的所有子树，从左至右规定次序。
- **15. 无序树**
  - ☞ 结点的子树不分先后次序。
- **16. 森林**
  - ☞ **m (m≧0)** 棵不相交树的集合。



5.1.3 树的基本运算

① 初始化树：**initialTree(T)**；
② 树的遍历 **–** 先序、中序、后序遍历
③ 查询根结点：**rootOf(T)**；
④ 查询父结点：**fatherOf(T)**；
⑤ 查询孩子结点：**childOf(T)**；
⑥ 查询兄弟结点：**siblingOf(T)**；
⑦ 求树的高度：**height(T)**
⑧ 求解点数 **–** 全部、**2**度、**1**度结点数
⑨ 插入子树：**insertTree (T,S)**；
⑩ 删除结点

…



君子生非异也，善假于物也。

荀子·劝学

单选题 1分

设置

设入栈序列为1 2 3 4 5，则可能的出栈序列为（ ）。

A 1 2 5 3 4

B 3 1 2 5 4

C 1 4 2 3 5

D 3 2 5 4 1

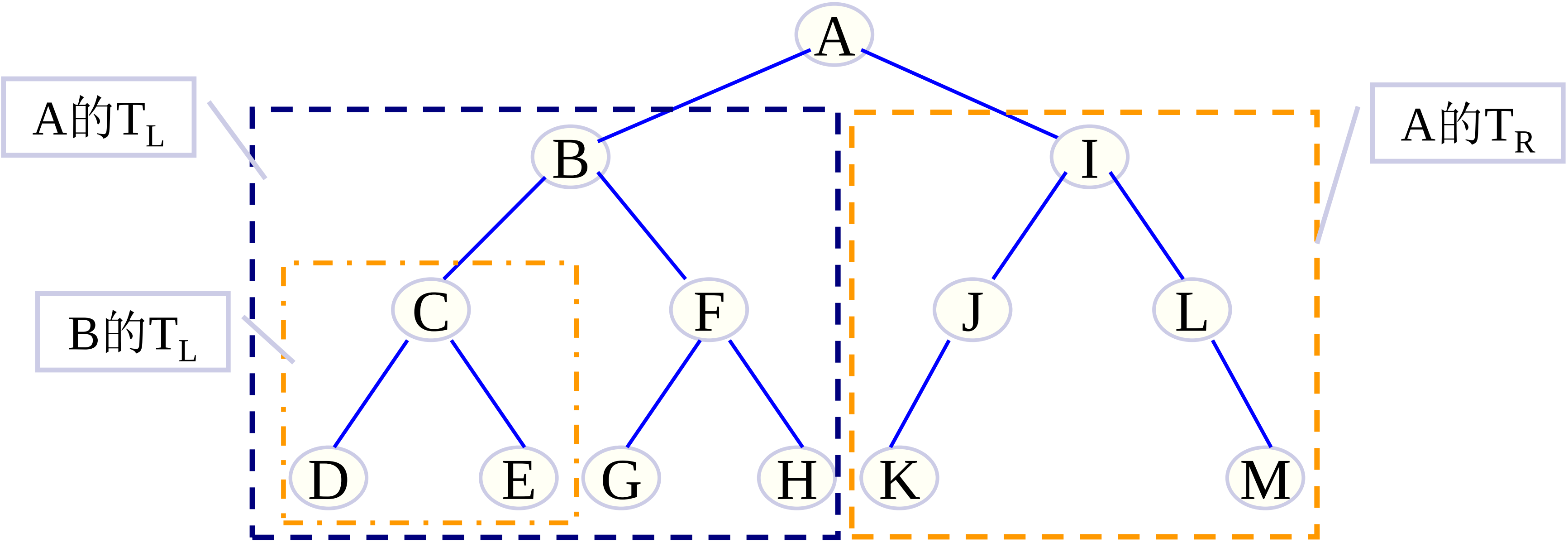
提交

5.2 二叉树(Binary Tree)

- **每个结点最多只有2棵子树；**
- 二叉树是有序树，即使只有一棵子树也要分清是左子树，还是右子树。

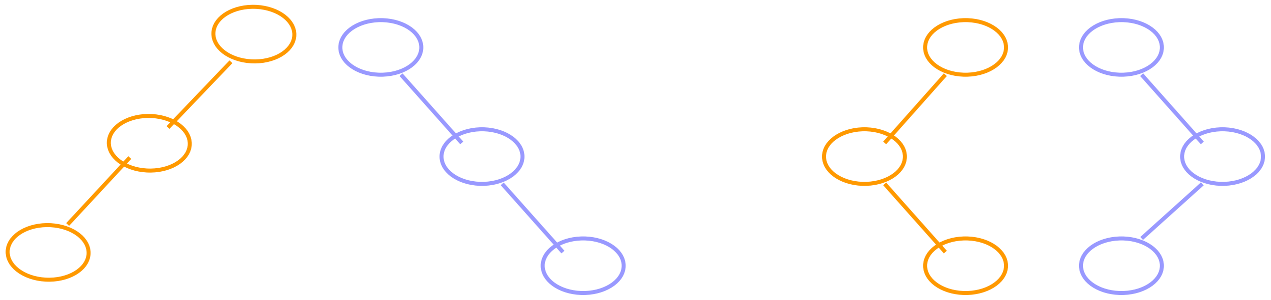
5.2.1 二叉树的基本概念

- **1. 二叉树定义**
  - 二叉树**T：**是**n**个结点组成的有限集合（**n >= 0**），**n=0**为空二叉树，否则：
    - 其中有一个根结点，
    - 其余结点可以划分成两个互不相交的子集**TL**和 **TR**，分别叫做左子树和右子树，
    - 且**TL, TR**也分别构成二叉树。
- 二叉树的定义也是递归的。


A的T_L
B的T_L
A的T_R
A
B
I
C
F
J
L
D
E
G
H
K
M

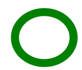
2. 二叉树特点

- 一个结点最多只能有两个孩子结点，或子树；
- 二叉树是有序树，即其左、右子树不能交换位置；即使只有一棵子树也要区分是左子树还是右子树。
- 结点都相同，交换左、右子树后，即为另一棵二叉树。
- (见下图)

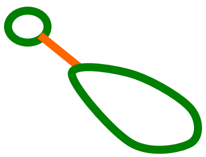


■ 3. 二叉树的五种形态

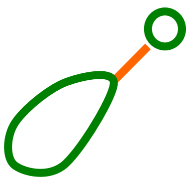
(a) 空树，结点数为**0**



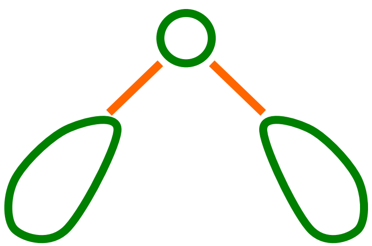
(b) 单结点二叉树，只有一个根结点



(c) 左子树为空，右子树不空



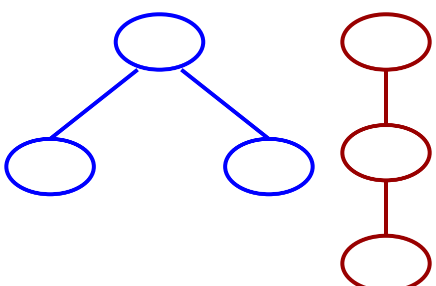
(d) 右子树为空，左子树不空



（e）左右子树均不空

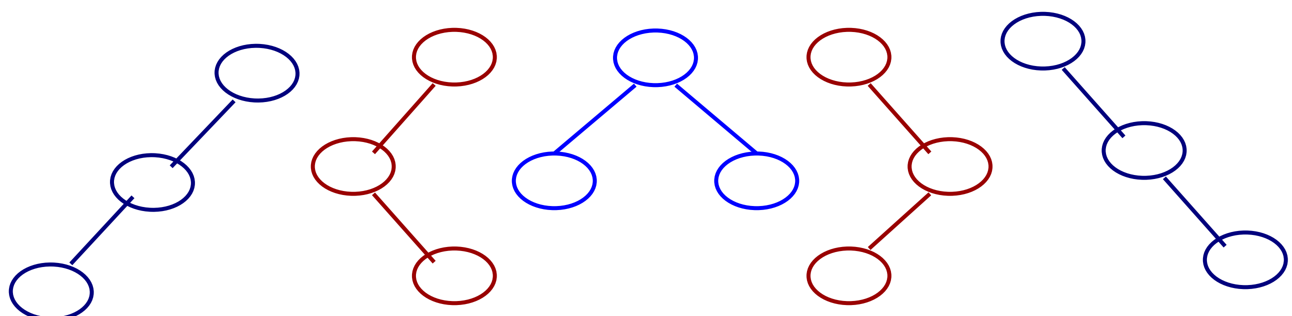
- **4. 二叉树与树的区别：**
  - 是两种不同的结构；
  - 二叉树最多两个子树，树可有多个子树；
  - 二叉树子树有序，树无序。
- **例: 比较三个结点的树与二叉树各有几种不同的形态。**

三个结点的树



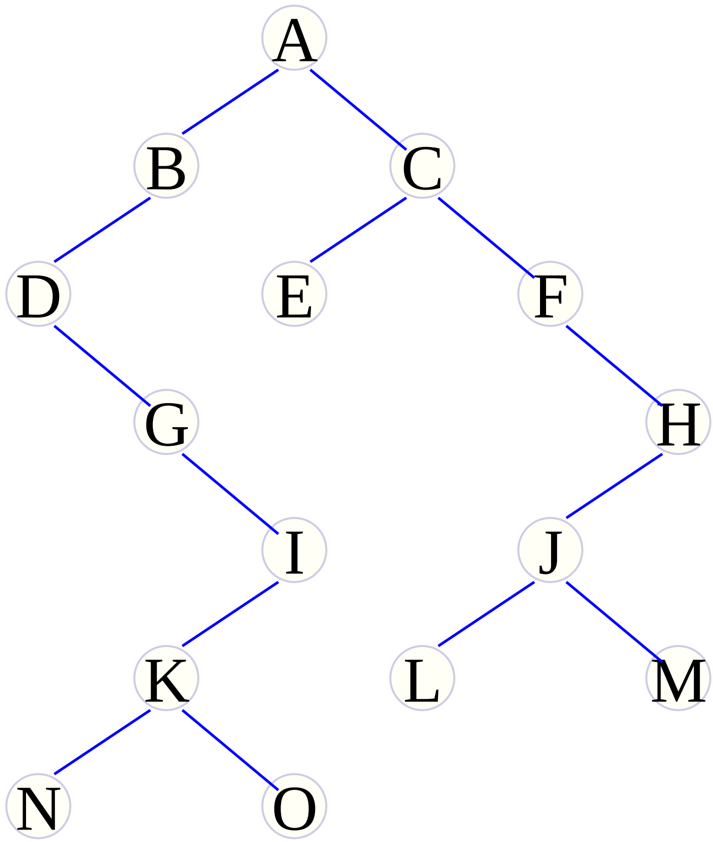
（两棵）

三个结点的二叉树



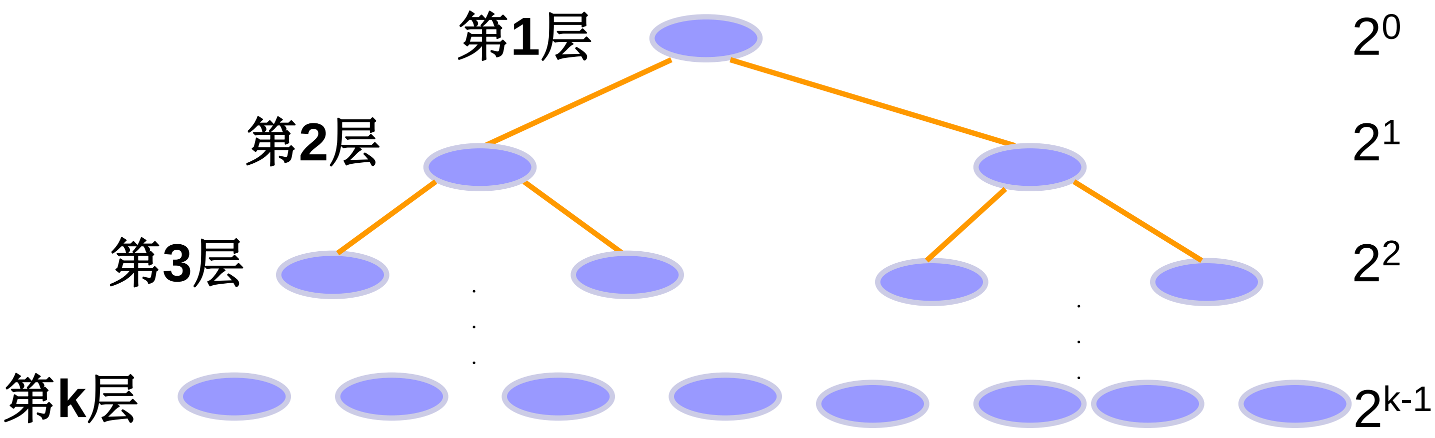
（五棵）

- 例：一颗二叉树



5.2.2 二叉树的性质

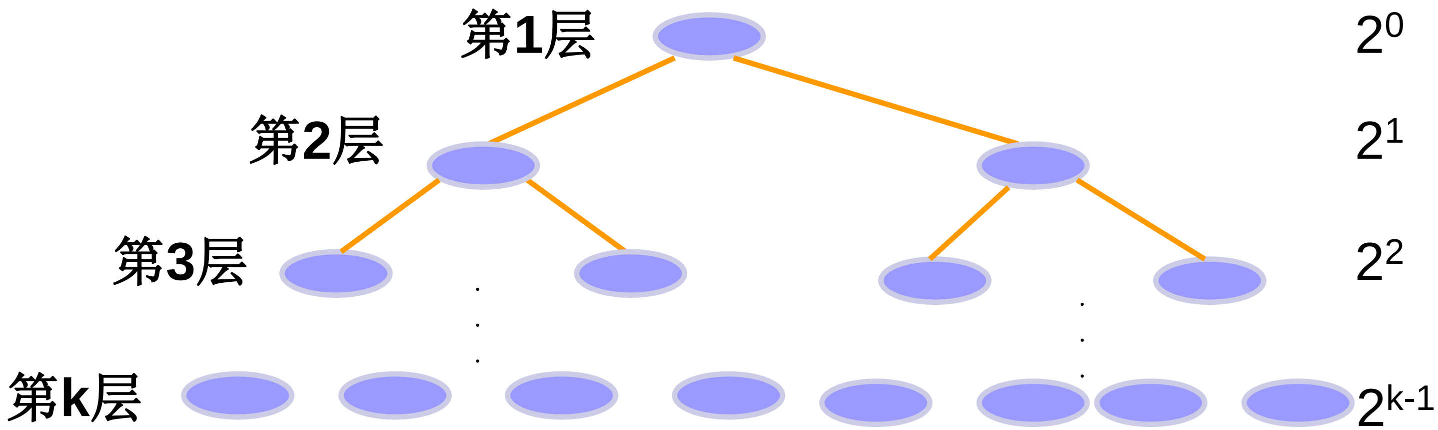
- **【性质1】第i层的结点数≤2^{i-1}；**



总的结点数≤$2^0+2^1+2^2+\dots+2^{k-1}$

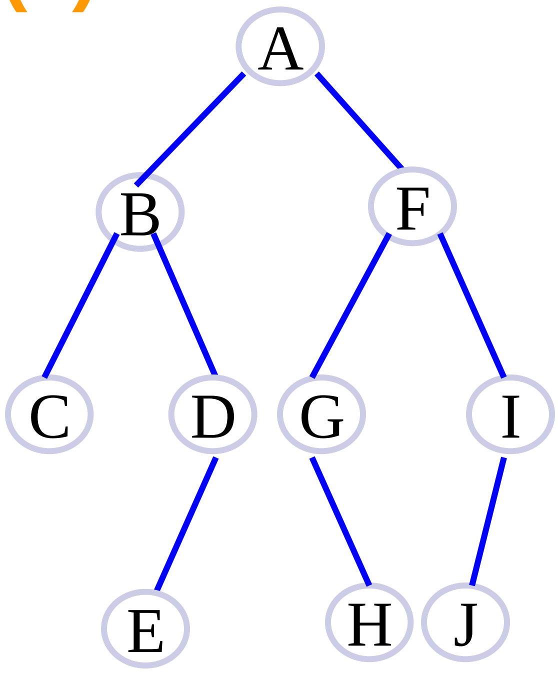
- 性质1 证明
  - 二叉树的第 i 层上，至多有 2^{i-1} 个结点 (i≧1)。
- 证明：-- 数学归纳法
  - i=1，$2^{1-1}=2^0=1$，至多只有 1 个根结点，显然正确；
  - i=2，$2^{2-1}=2^1=2$，至多只有 2 个结点，显然正确；
  - 设 j=i-1 时结论成立，即至多有：$2^{j-1}=2^{i-2}$ 个结点；
  - 当 j=i 时，因为 i-1 层最多 2^{i-2} 个结点，每个结点最多 2 个孩子（直接后继），所以第 i 层的结点数最多为：$2\times2^{i-2}=2^{i-1}$
  - 所以命题成立。

- 【性质2】高度为k（k≥1）的二叉树的结点总数≤2^k-1；



总的结点数≤$2^0+2^1+2^2+\ldots+2^{k-1}=\sum_{i=1}^{k}2^{i-1}=\frac{1-2^k}{1-2}=2^k-1$

- 【性质3】设二叉树的叶子结点数为n_0，度为2的结点数为n_2，则: $n_0=n_2+1$。
- 证明：设总结点数为n，度为1的结点数为n_1，则
  - ☞ $n=n_0+n_1+n_2$ --结点总数 (1)
  - ☞ $n-1=n_1+2n_2$ --分支（边）数 (2)
  - ☞ (1)－(2) 得 $n_0=n_2+1$



- $n-1=n_1+2n_2$ 的由来分析
  - ☞从树根往树叶方向看，1 度结点发出 1个分支（边），2 度结点发出 2 个分支，0 度结点（树叶）不发出分支，为 0，所以二叉树分支（边）总数为：n_1+2n_2
  - ☞从树叶往树根方向看，除了根结点外，每个结点接收1 个分支（边），根结点不接收分支，所以二叉树分支总数为：$n-1$
  - ☞所以：$n-1=n_1+2n_2$

- 或者根据图论结论：
  - 将树视为有向树，边的方向从父结点到子结点，有：边数（分支数）=入度之和=出度之和。
  - 入度之和=n-1
  - 出度之和=n_1+2n_2

【课堂练习】已知一棵二叉树中，有**20**个叶子结点，其中**10**个结点只有左孩子，**15**个结点只有右孩子，求该二叉树的总结点数。

解：

$n_0=20$

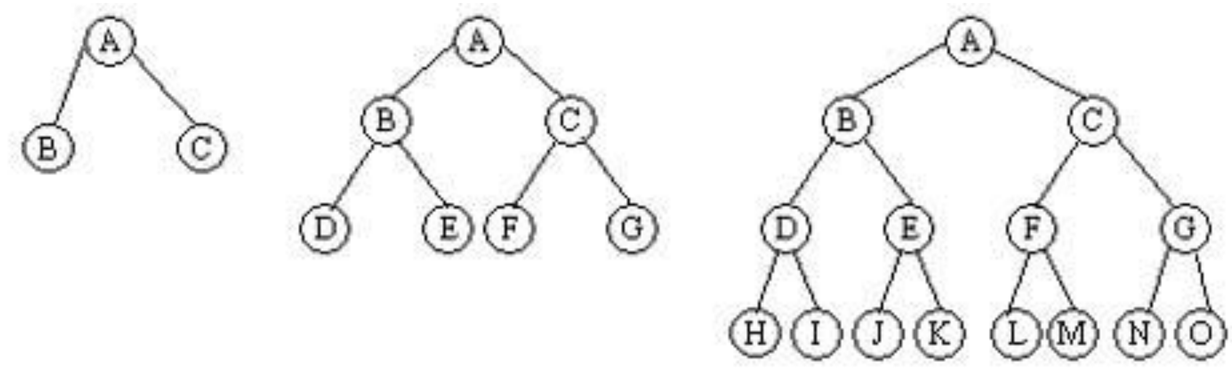
$n_1=10+15=25$

$n_2=n_0-1=20-1=19$ -- 性质3

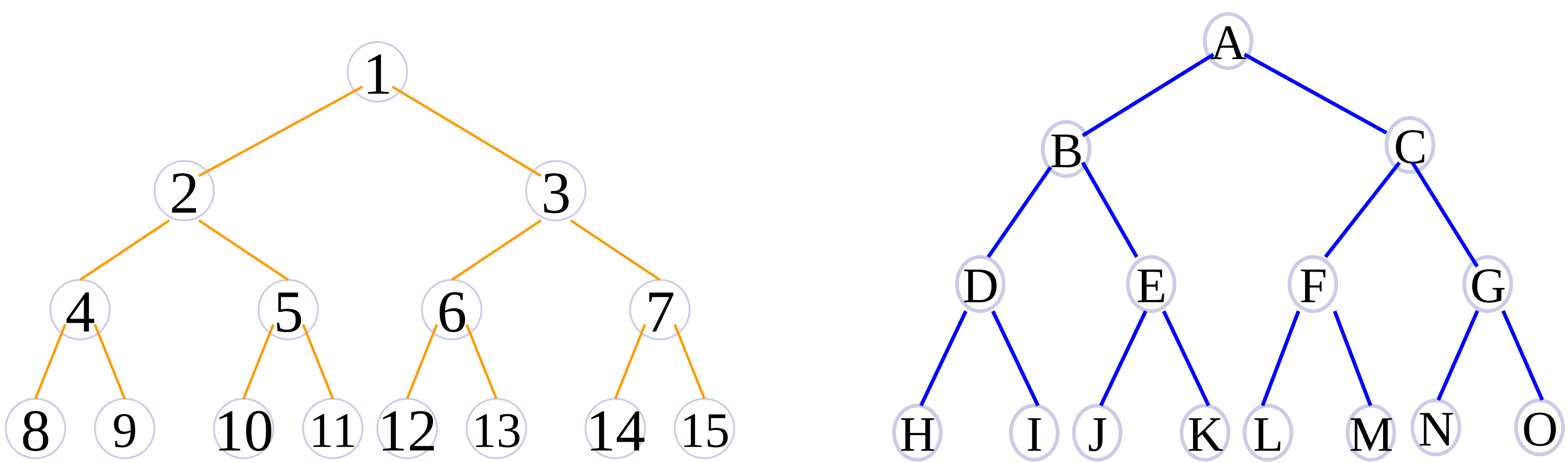
$n=n_0+n_1+n_2=20+25+19=64$

■ 满二叉树

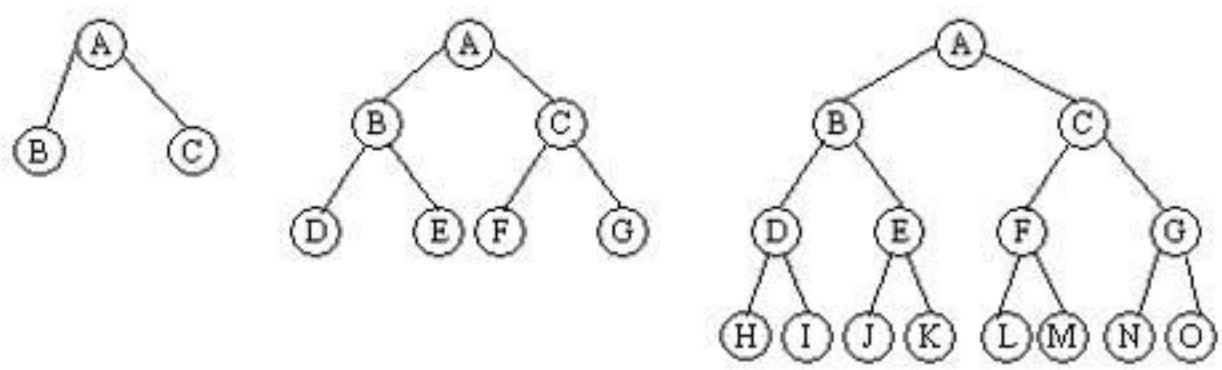
- 高度为k且有2^k-1个结点的二叉树为**满二叉树**。即**每一层都长满了结点的二叉树**。



(a) (b) (c)

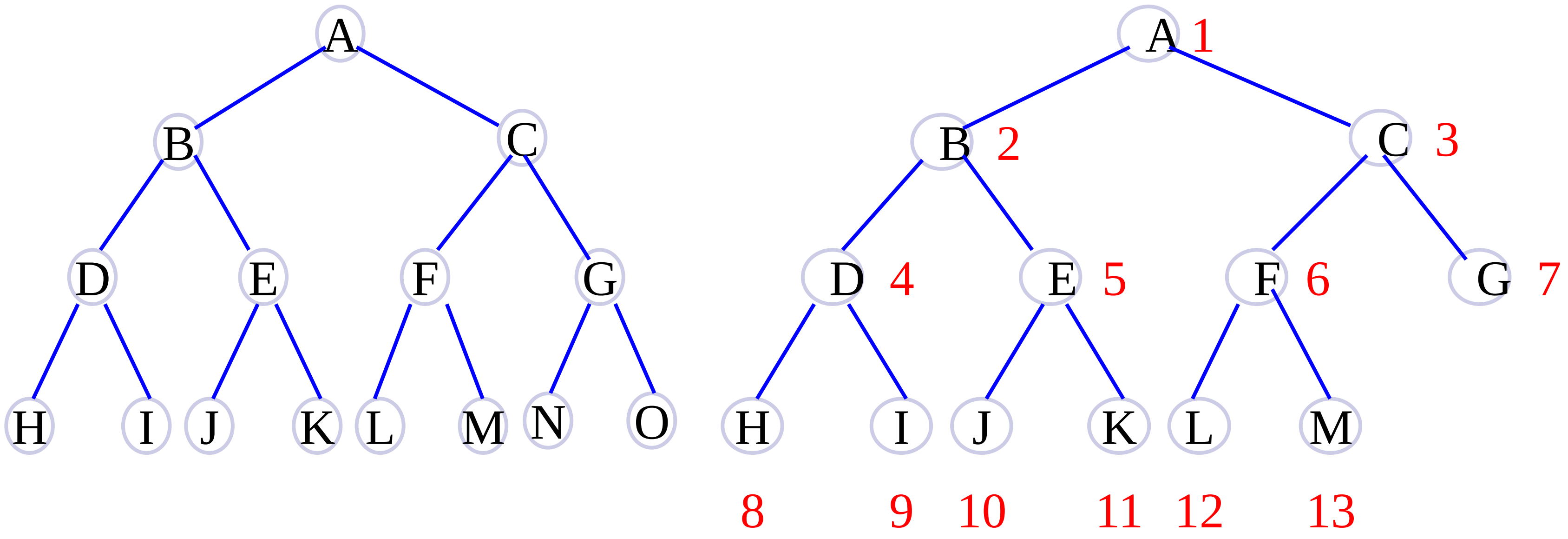


- 满二叉树特点：
  - 每层结点数都达到最大值（长满了结点），即第 i 层，结点数=2^{i-1}；
  - 满二叉树只有度为 0 或 2 的结点，没有度为 1 的结点；
  - 除叶结点外，每个结点均有 2 棵高度相同的子树；
  - 叶结点都在最深层次的同一层上。



(a) (b) (c)

- 完全二叉树
  - 在满二叉树最下一层从右到左依次连续去掉若干个结点的二叉树称为完全二叉树。
  - 最后一个结点之前长满了结点

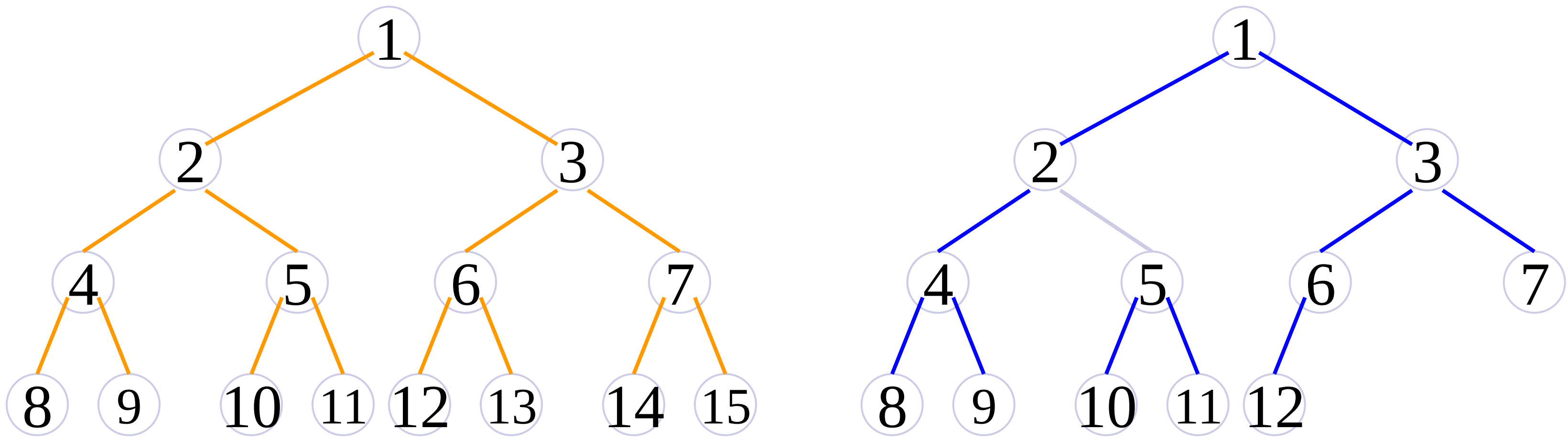


- 完全二叉树特点：
  - 一棵n 个结点、深度为 k的完全二叉树，一棵深度为 k 的满二叉树，同时对结点进行自上而下、自左至右，从 1 开始进行顺序编号；则完全二叉树的结点编号与满二叉树中编号从 1 至 n 的结点编号一一对应；
  - 叶结点只可能出现在最深的 2 层上；
  - 最下层结点一定是从左往右开始放置的；
  - 若某个结点没有左孩子，则其一定没有右孩子；

- ☞只有最深 2 层结点的度可能小于 2。
- ☞最多一个结点度为1。

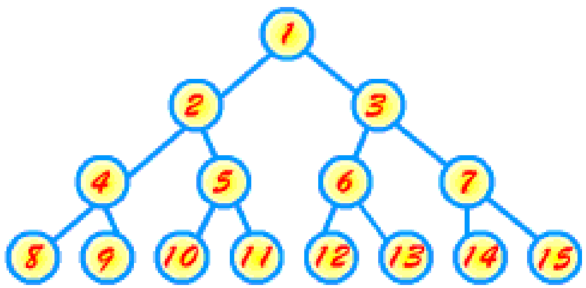
- 满二叉树一定是完全二叉树；反之不然。

- 完全二叉树示例：

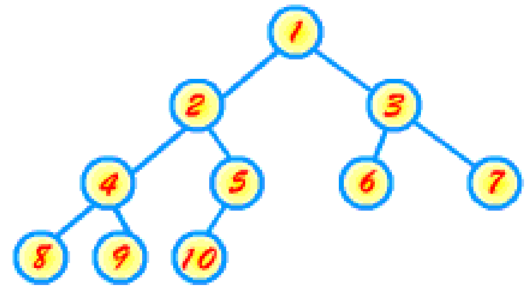


(a) 深度为 4 的满二叉树　　　　**(b) 深度为 4 的完全二叉树**

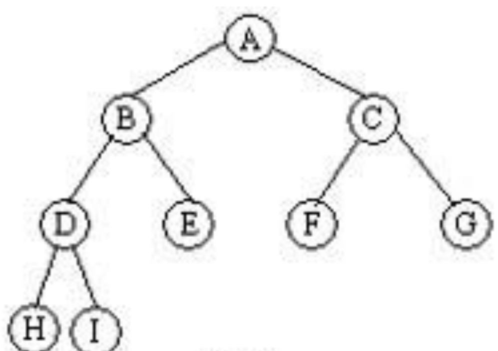
■ 完全二叉树示例：



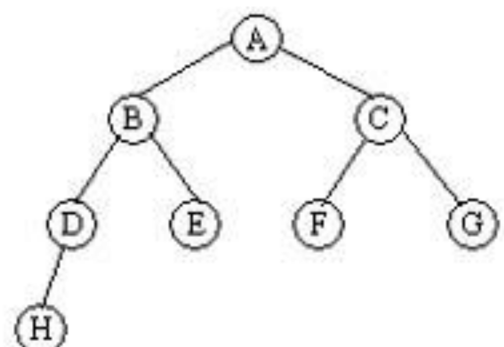
(a) 15个结点的满二叉树



(b) 10个结点的完全二叉树

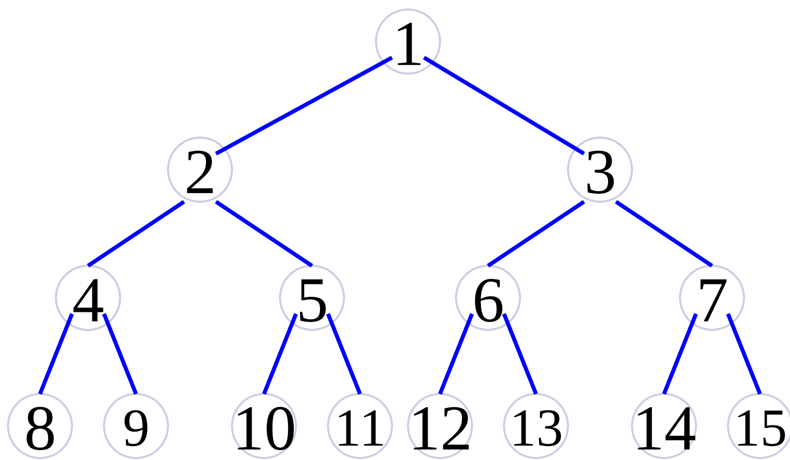


(a)

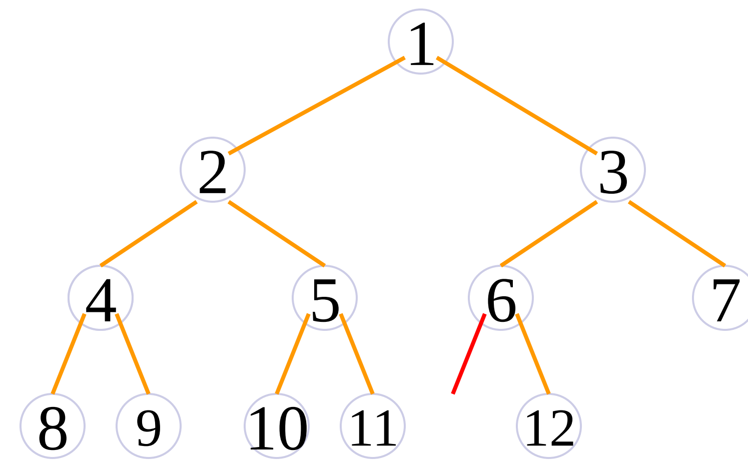


(b)

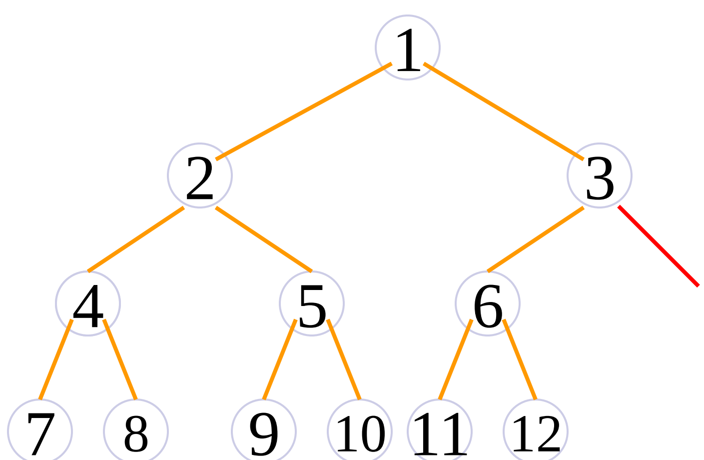
■ 非完全二叉树示例：



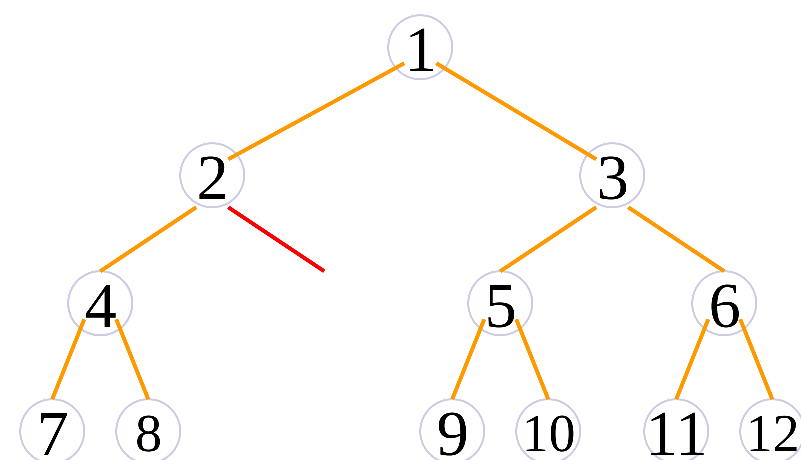
(a) 深度为 **4** 的满二叉树



(b) 非完全二叉树

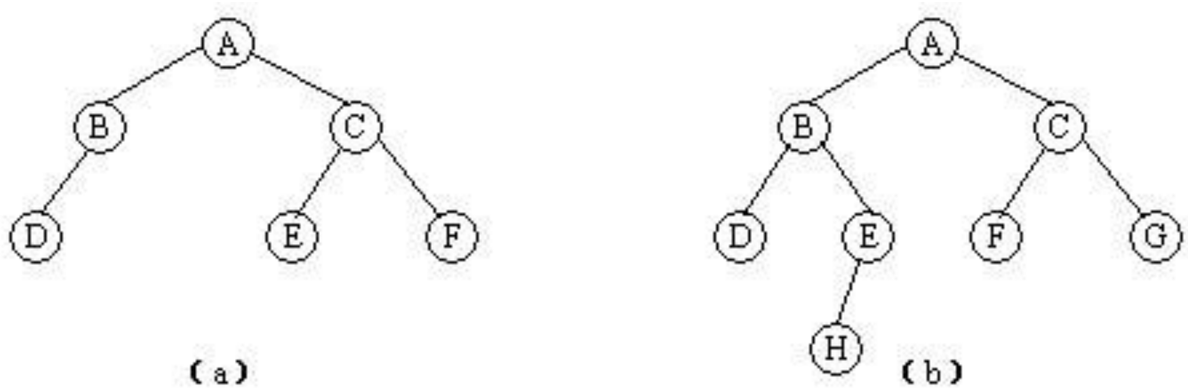


(c) 非完全二叉树



(d) 非完全二叉树

- 非完全二叉树示例：



【性质4】

- 有n个（n≥1）结点的完全二叉树的高度为:

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

☞其中 $\lfloor \log_2 n \rfloor$ 表示对 $\log_2 n$ 取下底整数，

☞即：$\lfloor \log_2 n \rfloor \le \log_2 n$

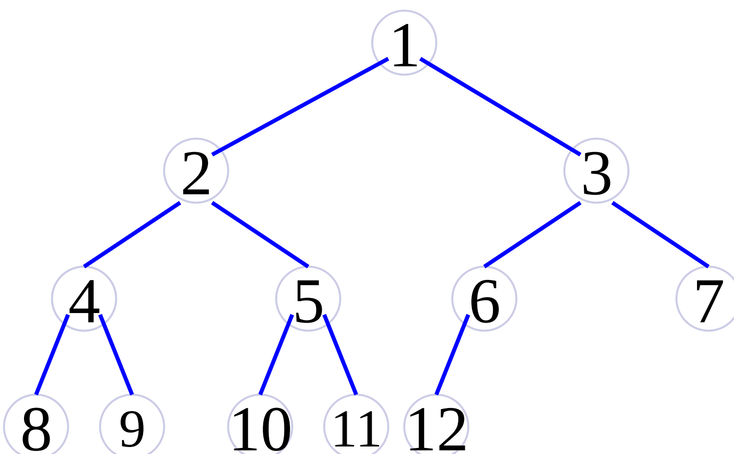
【性质4证明】

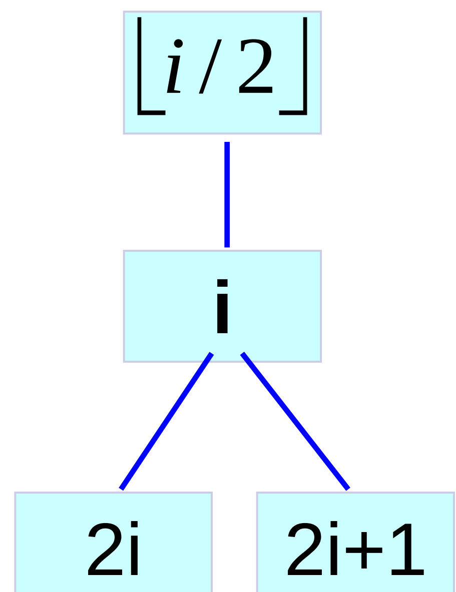
- 设此完全二叉树的深度为 k，则其前 k-1 层构成一棵深度为 k-1 的满二叉树，据满二叉树定义，此 k-1 层满二叉树共有 $2^{k-1}-1$ 个结点；
- 所以，此完全二叉树的结点数：$n > 2^{k-1}-1$ ；
- 又据性质 2，有：$n \le 2^k - 1$ ；
- 于是：$2^{k-1}-1 < n \le 2^k - 1$ ，可得：$2^{k-1} \le n < 2^k$ ；
- 取以 2 为底的对数，得：$k-1 \le \log_2 n < k$ ；
- 又 k 为整数，所以：$k-1 = \lfloor \log_2 n \rfloor$ ，即：

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

【性质5】

- 对完全二叉树进行层次编号，编号为 **i** 的结点，
  - ☞若左孩子存在，则其左孩子的编号为：**2i**；
  - ☞若右孩子存在，则其右孩子的编号为：**2i+1**；
  - ☞其父结点的编号为：$\lfloor i/2 \rfloor$

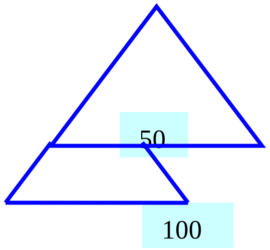




【课堂练习】

（1）求100个结点的完全二叉树的叶子结点数。

- 解:
  - 根据性质4，k=7，到第6层为满二叉树，共有63个结点，第7层有37个结点，编号从64到100。此37个结点为叶子结点。
  - 其中100号结点的父结点为50号结点（性质5）。
  - 所以，第6层从51号结点到63号结点没有子结点，亦为叶子结点，共13个结点
  - 所以：叶子结点数为：13+37=50。编号从51到100。
  - 简单方法--由性质5直接得到。



（2）完全二叉树的第7层有10个结点，问共有几个结点？多少个叶子结点？多少个度为1的结点？

- 解：共有2^6-1+10=73
  - 叶子求法：
    - 方法1：37号到73号都是叶子，共37个叶子结点（性质5）；
    - 方法2：第7层10个结点都是叶子，第6层有2^{6-1}=32个结点，其中5 个结点是第7层10个结点的父亲。

      所以，共有10+32-5=37个叶子结点。
  - 度为1的结点数为0。(第7层偶数个结点)

（3）判断题：完全二叉树最多有1个度为1的结点。（ ）

（4）如何判断编号为i、j的两个结点是否在同一层。

【布置作业】

（p157）--（6）学号3、6、9

- 5.1
- 5.2
- 5.4
- 5.6
- 5.7

路虽远，行则将至；

事虽难，做则必成。

单选题 1分

设置

设循环顺序队列Q[0：M-1]的头指针和尾指针分别为F和R，头指针F总是指向队头元素的前一位置，尾指针R总是指向队尾元素的当前位置，队列中元素个数的计算公式为（ ）。

A R-F

B F-R

C (R-F+M)%M

D (F-R+M)%M

提交

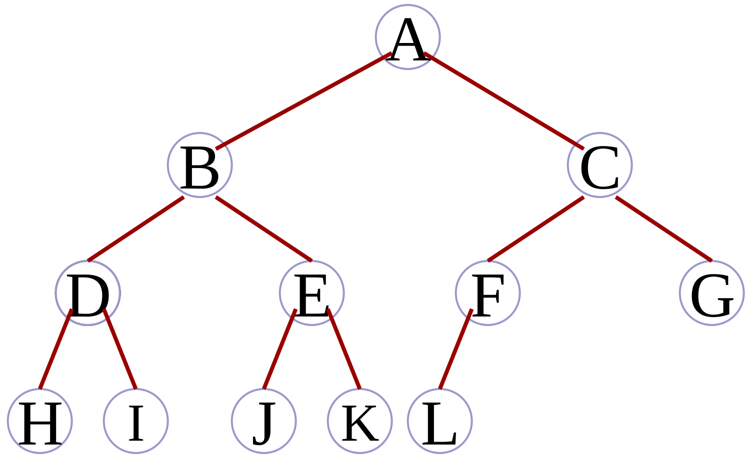
5.2.3 二叉树的顺序存储结构

- 存储一个结构时，不仅要存值，还要存储元素间的关系。

1. 完全二叉树的顺序存储方式

- 用数组存储二叉树各结点的值，按自上而下、自左至右的编号次序存放结点元素；
- 各结点在数组中的位置（数组下标）-- 就是其在完全二叉树中对应结点的编号（注意差1，保留数组下标为0单元不用）。

完全二叉树顺序存储示例：



(a) 深度为 **4** 的完全二叉树

	A	B	C	D	E	F	G	H	I	J	K	L
0	1	2	3	4	5	6	7	8	9	10	11	12

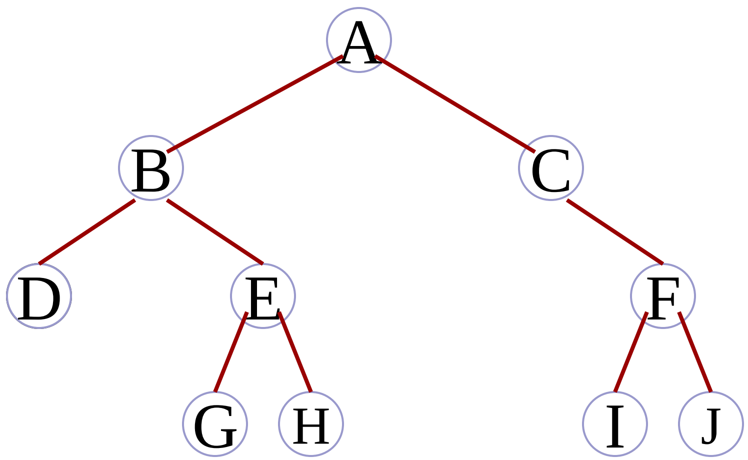
(b) 顺序存储结构示例

- 优点：方便、简洁
  - 定位结点算法简单，由性质 5，此存储方法很容易根据当前结点的编号，计算出其双亲结点，以及左、右孩子结点。
- 缺点：
  - 此存储结构仅适用于完全二叉树；
  - 普通二叉树如何顺序存储呢？
    - 对于普通的二叉树需要转换成完全二叉树进行存储，但会浪费一些存储空间。
  - 进行插入、删除结点操作算法复杂。

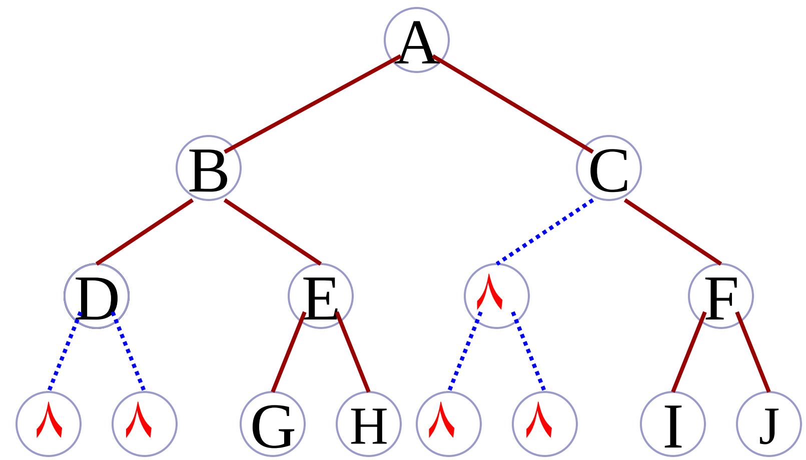
2. 普通二叉树转换为完全二叉树

- 比照相同深度的完全二叉树，补齐缺少的结点，使之成为一棵“完全二叉树”；
- 增补的虚结点用特殊符号区分，比如“∧”;

■ 普通二叉树转为完全二叉树存储示例：



(a) 普通二叉树

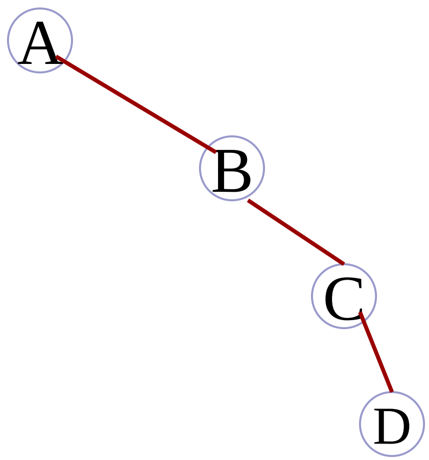


(b) 增补虚结点使之为完全二叉树

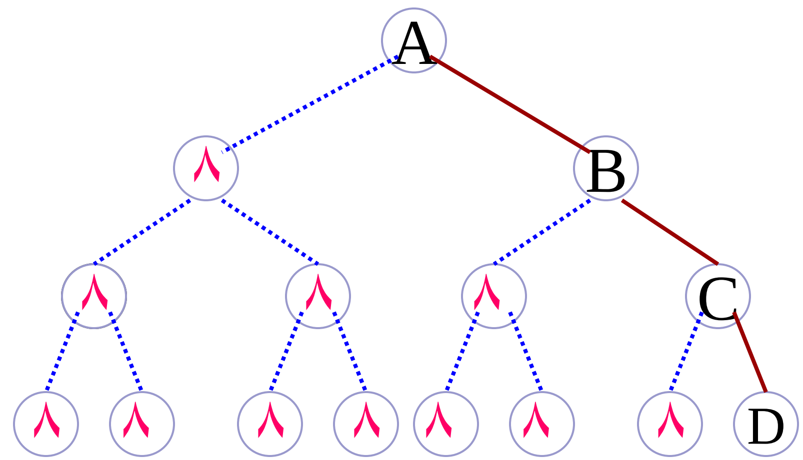
	A	B	C	D	E	∧	F	∧	∧	G	H	∧	∧	I	J
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

(c) 顺序存储结构示例

- 普通二叉树转换为完全二叉树进行顺序存储，造成存储空间损失，最严重的情况如下例，所有结点都只有右孩子（或左子树）的二叉树。



(a) 只有右分支的普通二叉树



(b) 增补虚结点使之为完全二叉树

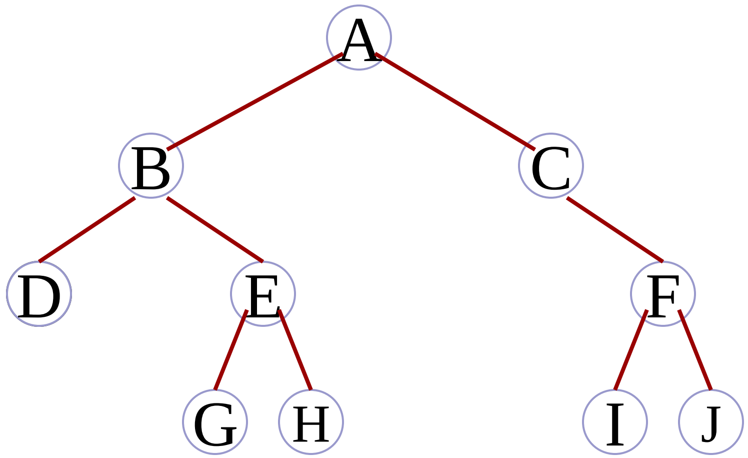
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	A	∧	**B**	∧	∧	∧	**C**	∧	∧	∧	∧	∧	∧	∧	**D**

(c) 顺序存储结构示例

二叉树的一维数组存储

- ☞对元素进行封装：数据元素、左右孩子指针（下标）、父结点指针（下标）。
- ☞将封装好的结点存入一维数组（顺序表）。

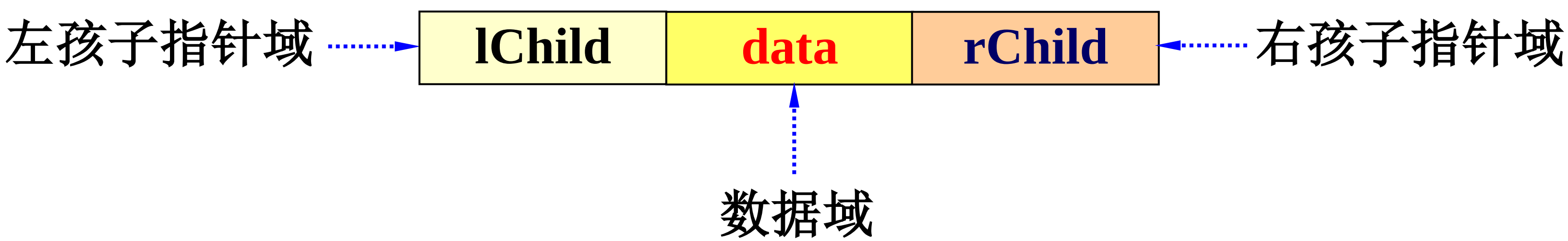
```
typedef struct hfmNode
{
    elementType data;           //元素值
    int parent,lChild,rChild;   //父结点、左右孩子结点指针
}node;

node T[];        //存储二叉树的数组，或用顺序表
```



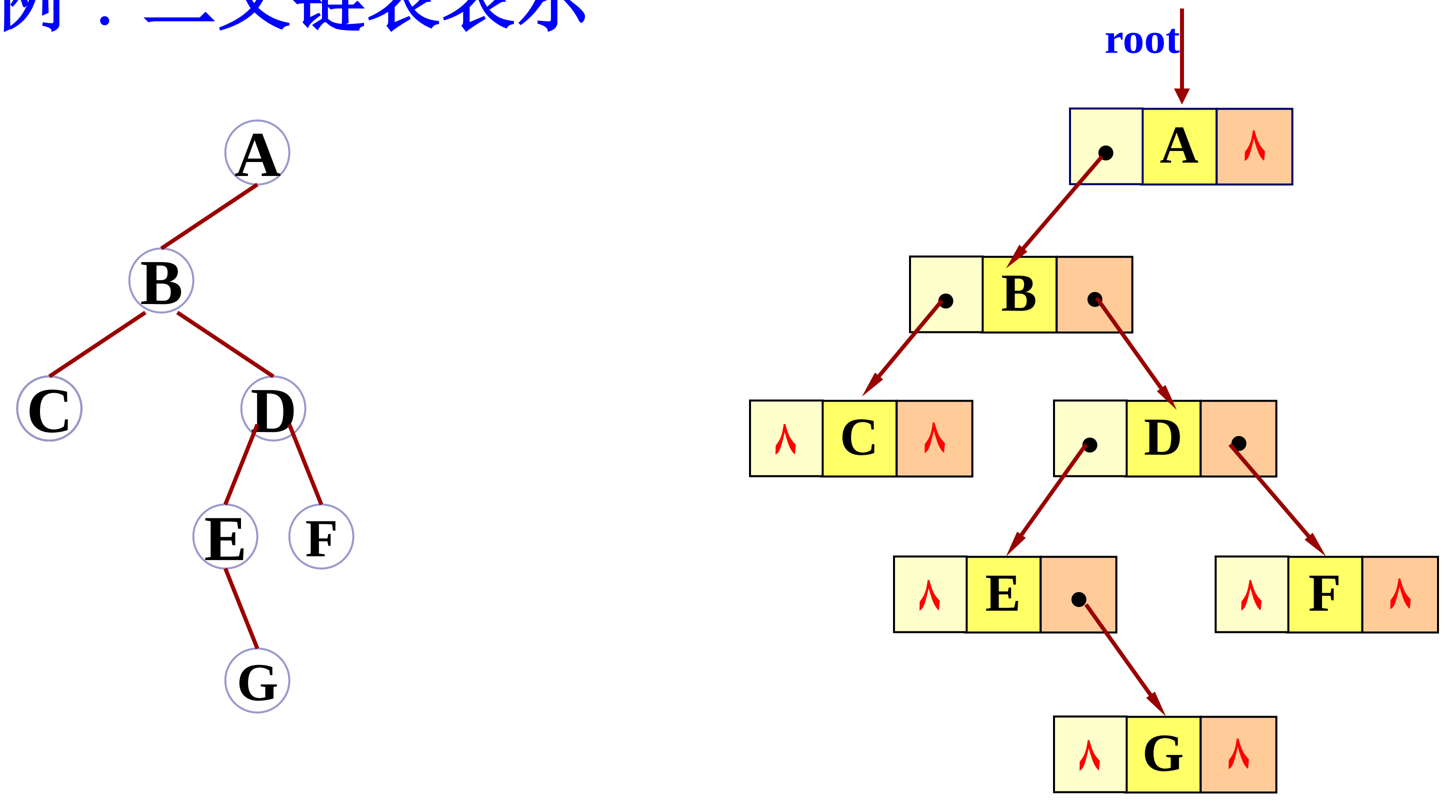
	d	p	l	r
0	A	-1	1	2
1	B	0	3	4
2	C	0	-1	5
3	D	1	-1	-1
4	E	1	6	7
5	F	3	8	9
6	G	4	-1	-1
7	H	4	-1	-1
8	I	5	-1	-1
9	J	5	-1	-1

5.2.4 二叉树的二叉链表表示

- 简称二叉树的链式存储结构
- 结点由三个域组成：
  - 数据域 – 存放结点数据元素；
  - 左孩子指针域 – 存放左孩子结点的地址；
  - 右孩子指针域 – 存放右孩子结点的地址。
- 结点结构形式如图：



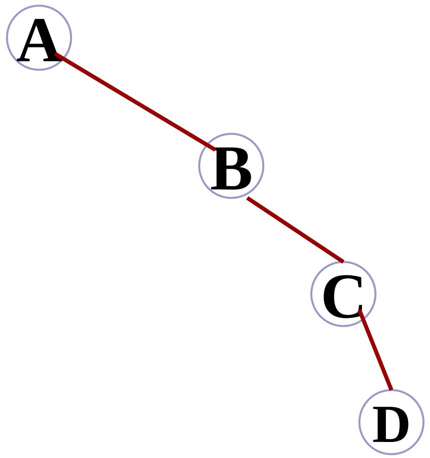
- 例：二叉链表表示



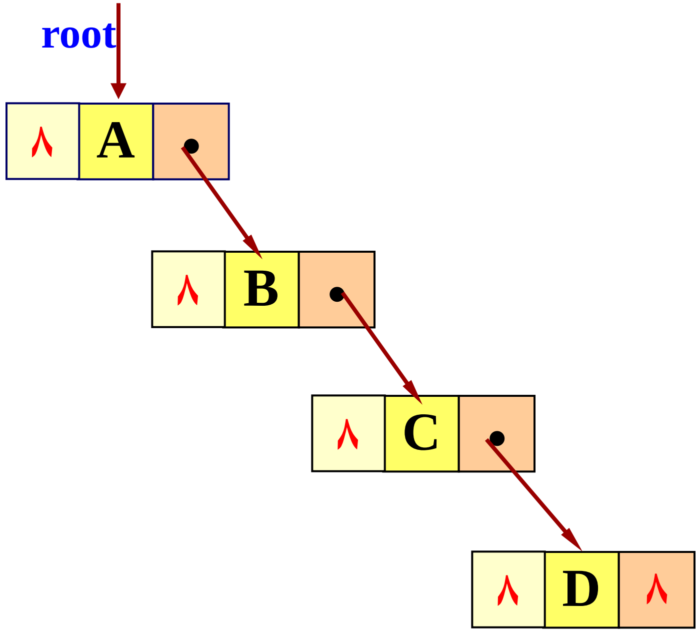
(a) 二叉树

(b) 二叉链表表示

- 例：二叉链表表示



(a) 二叉树



(b) 二叉链表表示

- 二叉树的根结点指针，唯一确定一棵二叉树

- n个结点的二叉链表共有2n个指针域。其中
  - ☞ n-1个指针指向结点（除了根结点，其它结点皆有指针指向）
  - ☞ n+1个指针为空，即: 2n-(n-1) = n+1。

- 二叉链表结点结构的 C 语言描述：

```
typedef struct blNode
{
    elementType  data;   //存放元素数据
                 //左、右孩子结点（子树根）指针。
    struct blNode  *lchild,  *rchild;
} btNode,  *BiTree;
```

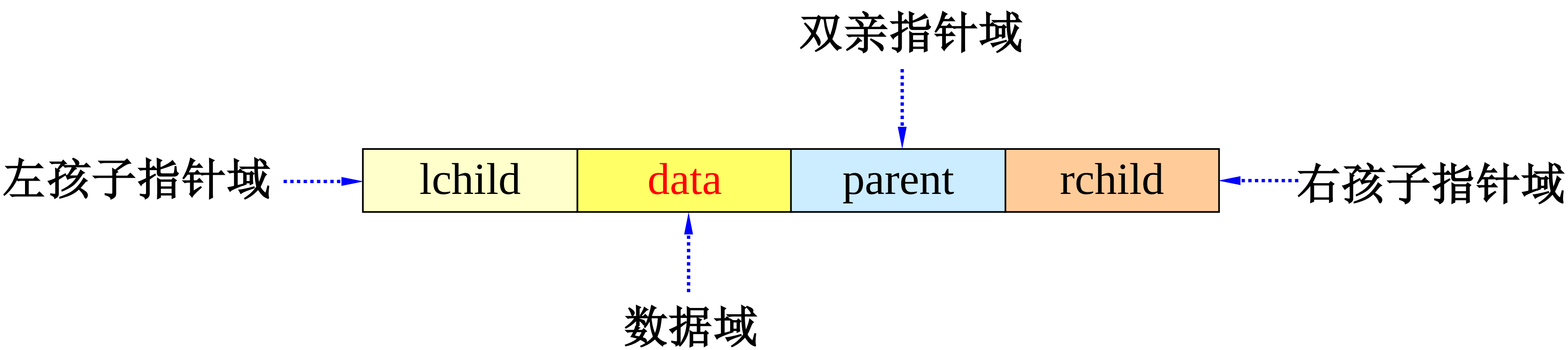
- 二叉链表表示二叉树的 C++ 类描述:

```
class BiTree
{
public:
    BiTree(){};
    ~BiTree(){};
    void PreOrder(btNode *T);      //先序遍历二叉树
    void InOrder(btNode *T);       //中序遍历二叉树
    void PostOrder(btNode *T);   //后序遍历二叉树
    bool Empty(btNode *T);         //判断是否空二叉树
       …                          //二叉树的其它运算
private:
    btNode *root;    //根结点指针，唯一确定一棵二叉树
};
```

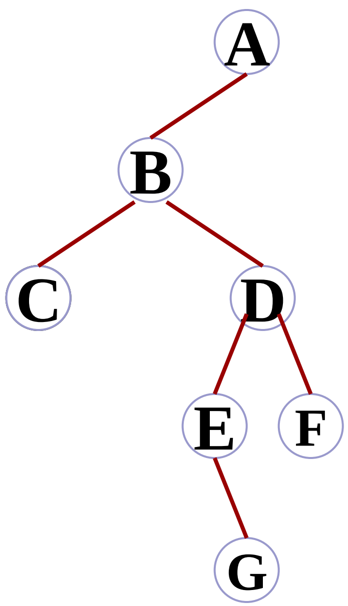
5.2.5 三叉链表存储结构

- ☞二叉链表查找当前结点的孩子结点方便，但查找其双亲结点需要从头开始重新搜索二叉树，为此，可以使用三叉链表结构；
- ☞三叉链表结点结构在二叉链表结构基础上增加一个双亲指针域，存放其双亲结点的地址，即指向双亲。

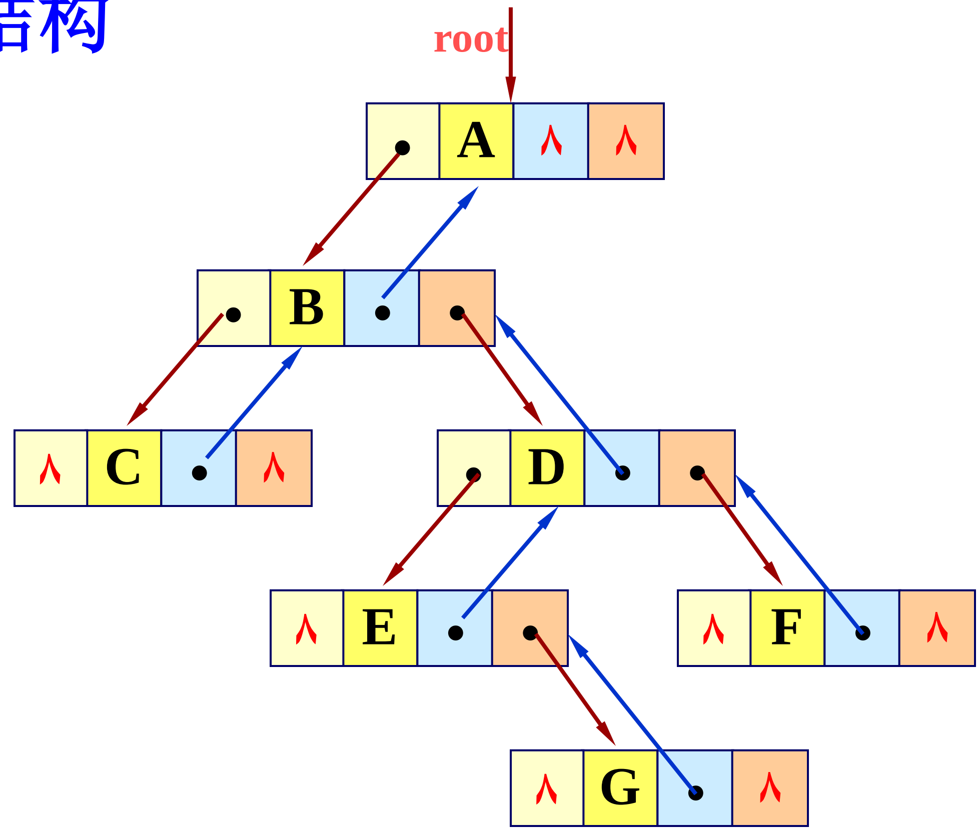
- 三叉链表结点结构如图：



- 例：三叉链表存储结构



(a) 二叉树



(b) 三叉链表表示

- 三叉链表结点结构描述：

```
typedef struct TriTNode
{
    elementType  data;
            // 左、右孩子、双亲指针。
    struct TriTNode  *lchild,  *rchild,
  *parent;
} TriBiNode,  *TriTree;
```

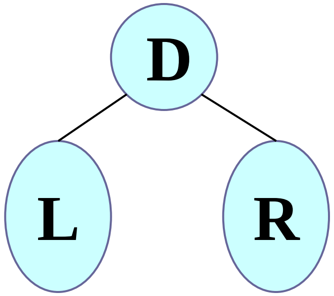
Better education, better jobs.

George W. Bush's electing slogan

5.3 二叉树的遍历及其应用

- 二叉树的遍历 (Traverse)
  - 按照某种次序依次访问二叉树T中每个结点一次且仅一次;
  - 在访问每个结点的过程中，可以对结点进行各种操作。比如：存、取结点信息，对结点进行计数等。
  - 在线性表中也有结点的遍历概念，就是从头到尾访问每个结点，用循环即可实现。因为线性表中结点关系简单，没有专门提及遍历（名称）。

4.3.1 遍历算法基础



- 二叉树形态如图：
- 二叉树的定义是递归的，即一棵非空二叉树由 **3** 个部分组成：根结点、左子树、右子树；
- 若能依次遍历这三个部分，则遍历了整棵二叉树；
- 分别以 **L**、**D**、**R** 表示左子树、根结点、右子树，则全部可能的遍历方案有**6** 种，分别为：

先左后右：	**DLR**	**LDR**	**LRD**
先右后左：	**DRL**	**RDL**	**RLD**
	先根序	中根序	后根序

- 我们约定按"先左后右"，则只有三种遍历方法：**DLR**、**LDR**、**LRD**

☞ 先根（序）遍历 -- DLR；

☞ 中根（序）遍历 -- LDR；

☞ 后根（序）遍历 -- LRD 。

(1) 先根（序）DLR 遍历二叉树的操作定义

若二叉树非空：

① 访问根结点；

② 先序遍历左子树；

③ 先序遍历右子树。

(2) 中根（序）LDR 遍历二叉树的操作定义

若二叉树非空：

① 中序遍历左子树；

② 访问根结点；

③ 中序遍历右子树。

(3) 后根（序）LRD 遍历二叉树的操作定义

若二叉树不空：

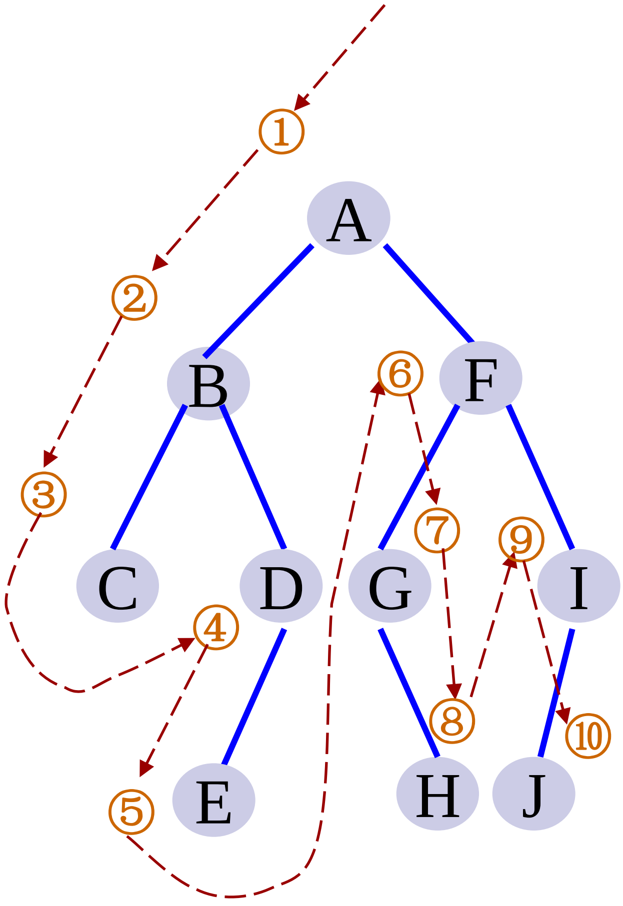
① 后序遍历左子树；

② 后序遍历右子树；

③ 访问根结点。

- 以上三种遍历方式的定义都是递归的。

【例】二叉树遍历



- 先序遍历：

1) **A** $\underbrace{\qquad\qquad}_{A_L}$ $\underbrace{\qquad\qquad}_{A_R}$

2) **A B** $\underbrace{\underbrace{\qquad}_{B_L}\ \underbrace{\qquad}_{B_R}}_{A_L}$ **F** $\underbrace{\underbrace{\qquad}_{F_L}\ \underbrace{\qquad}_{F_R}}_{A_R}$

B B_L B_R F F_L F_R

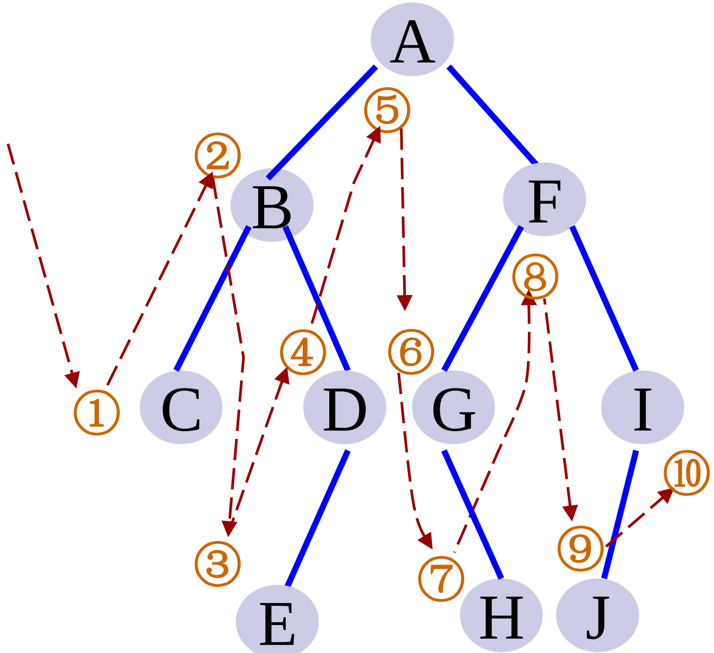
3) **A B C D E F G H I J**

B_L B_R F_L F_R

A_L A_R

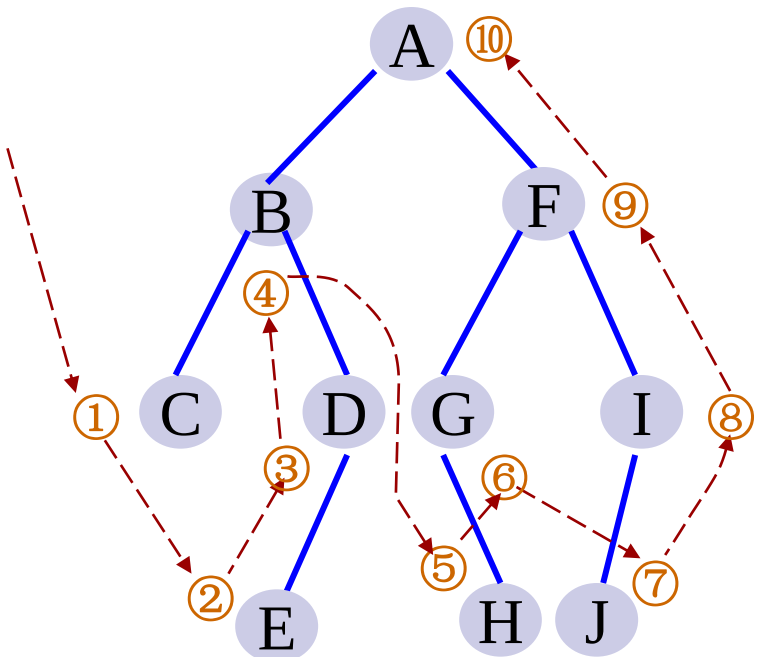
- 先序序列：**ABCDEFGHIJ**

中序遍历



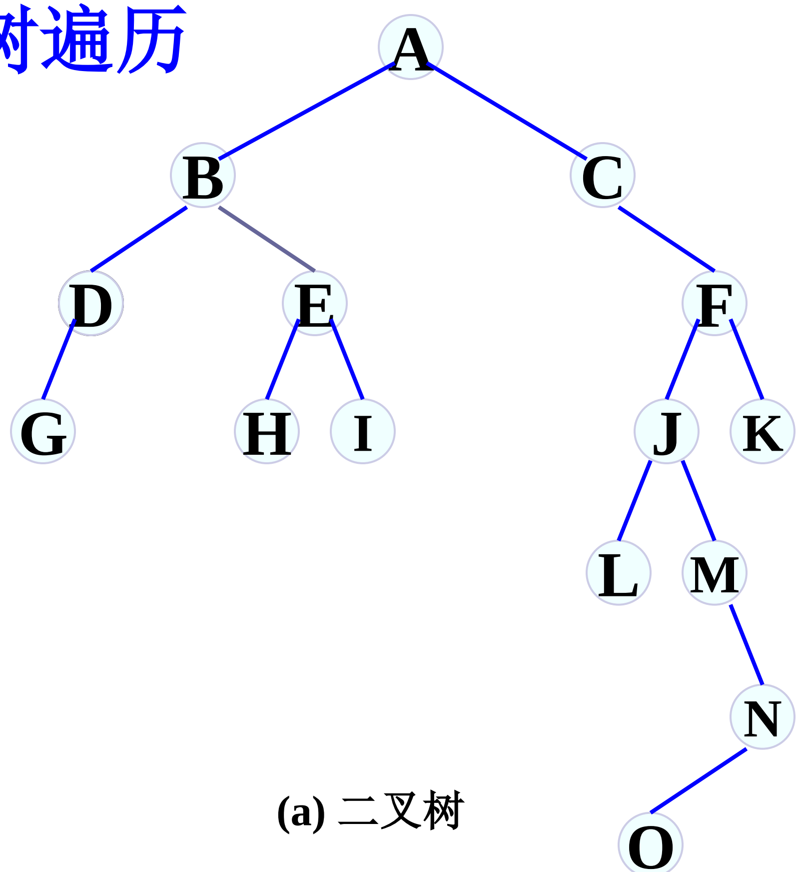
- 中序序列：**CBEDAGHFJI**

■ 后序遍历



■后序序列：**CEDBHGJIFA**

- 【例】二叉树遍历



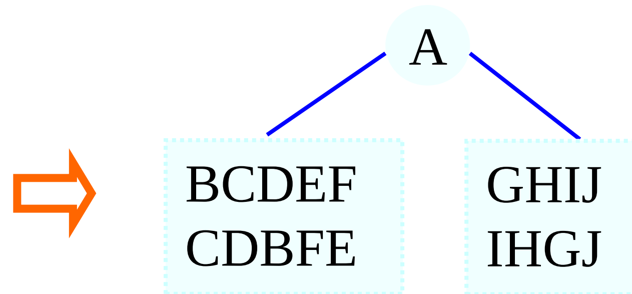
(a) 二叉树

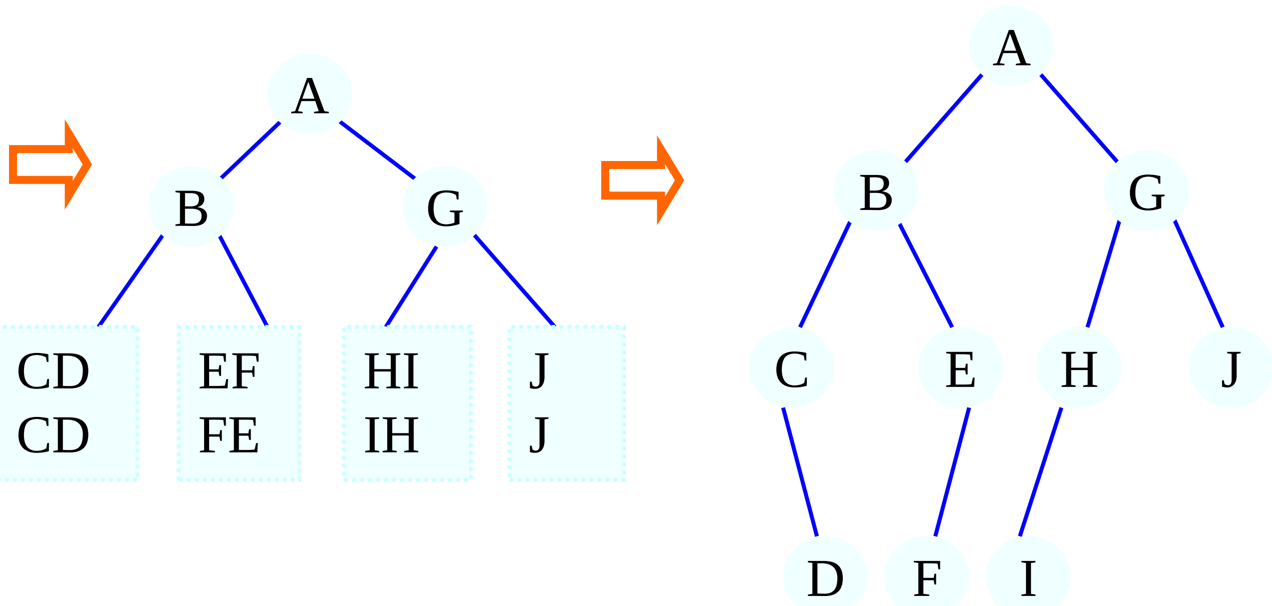
先序遍历：ABDGEHICFJLMNOK

中序遍历：GDBHEIACLJMONFK

后序遍历：GDHIEBLONMJKFCA

- 【例】已知二叉树的先序和中序序列如下，试构造出相应的二叉树。
  - 先序：ABCDEFGHIJ
  - 中序：CDBFEAIHGJ

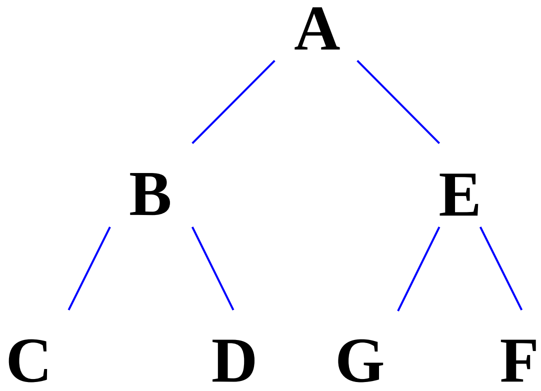




- **结论：**
  - 已知中序序列和后序序列，或中序序列和先序序列，可以唯一的还原一棵二叉树；
  - 而已知先序序列和后序序列不能唯一还原二叉树。
- **课堂练习：**
  - 已知二叉树中序序列和后序序列，还原此二叉树。

中序：**CBDAGEF**

后序：**CDBGFEA**



- 【思考问题】二叉树3中遍历算法中，最先和最后访问的结点有何特点？（或如何找到第一个访问和最后一个访问的结点？）。如何设计算法实现？
- 先序遍历第一个访问的结点
  - 二叉树、子树的根结点。
- 中序遍历第一个访问的结点
  - 第一个左子树为空的结点；可能是叶子结点。
  - 如果左子树存在则在左子树上，否则，即为根结点。
- 后序遍历第一个访问的结点
  - 第一个叶子结点（左子树、右子树皆为空）；
  - 如果左子树存在，则一定在左子树上；
  - 如果左子树为空，则一定在右子树上。

- 先序遍历最后一个访问的结点
  - 叶子结点。
  - 如果右子树为空，左子树存在则在左子树上，否则，右子树不空则在右子树上。
- 中序遍历最后一个访问的结点
  - 右子树为空的结点；可能是叶子结点。
  - 如果右子树存在则在右子树上，否则，即为根结点。
- 后序遍历最后一个访问的结点
  - 根结点。

【布置作业】

（p157-p158）--（7）学号：0、1、4、7

- ☞ 5.8
- ☞ 5.9
- ☞ 5.11
- ☞ 5.12
- ☞ 5.13

落霞与孤鹜齐飞，秋水共长天一色。

--王勃《滕王阁序》

5.3.2 二叉树遍历算法描述

- 先序遍历：

  若二叉树T不空，则：

  ☞访问T的根结点。

  ☞先序遍历T的左子树。

  ☞先序遍历T的右子树。

【先序遍历算法描述】

```
void PreOrder(btNode *T)
{
  if(T)
  {
     visit(T);          //访问根结点。
        //比如：打印当前结点  cout<<T->data<<“ ”;
     PreOrder(T->lChild);   //先序遍历左子树
     PreOrder(T->rChild);   //先序遍历右子树
  }
}
```

【顺序存储--先序遍历算法描述】

```
void preOrder( seqList T, int i )
{
  if( i<=T.listLen )
  {
    visit( T.data[i] );
    preOrder( T, 2*i );              //遍历左子树
    preOrder( T, 2*i+1 );            //遍历右子树
  }
}
```

- 中序遍历：

  若二叉树T不空，则：

  ☞中序遍历T的左子树。

  ☞访问T的根结点。

  ☞中序遍历T的右子树。

【中序遍历算法描述】

```
void InOrder(btNode *T)
{
  if(T)
  {
     InOrder(T->lChild);     //中序遍历左子树
     visit(T);               //访问根结点。
        //比如：打印当前结点  cout<<T->data<<" ";
     InOrder(T->rChild);     //中序遍历右子树
  }
}
```

【顺序存储--中序遍历算法描述】

```
void inOrder( seqList T, int i )
{
  if( i<=T.listLen )
  {
   inOrder( T,2*i );       //遍历左子树
   if( T.data[i]!='/' )
         cout<<T.data[i]<<"  ";
   inOrder( T,2*i+1 );   //遍历右子树
  }
}
```

- 后序遍历：

  若二叉树T不空，则：

  ☞后序遍历T的左子树。

  ☞后序遍历T的右子树。

  ☞访问T的根结点。

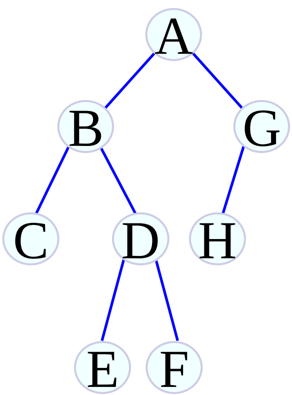
【后序遍历算法描述】

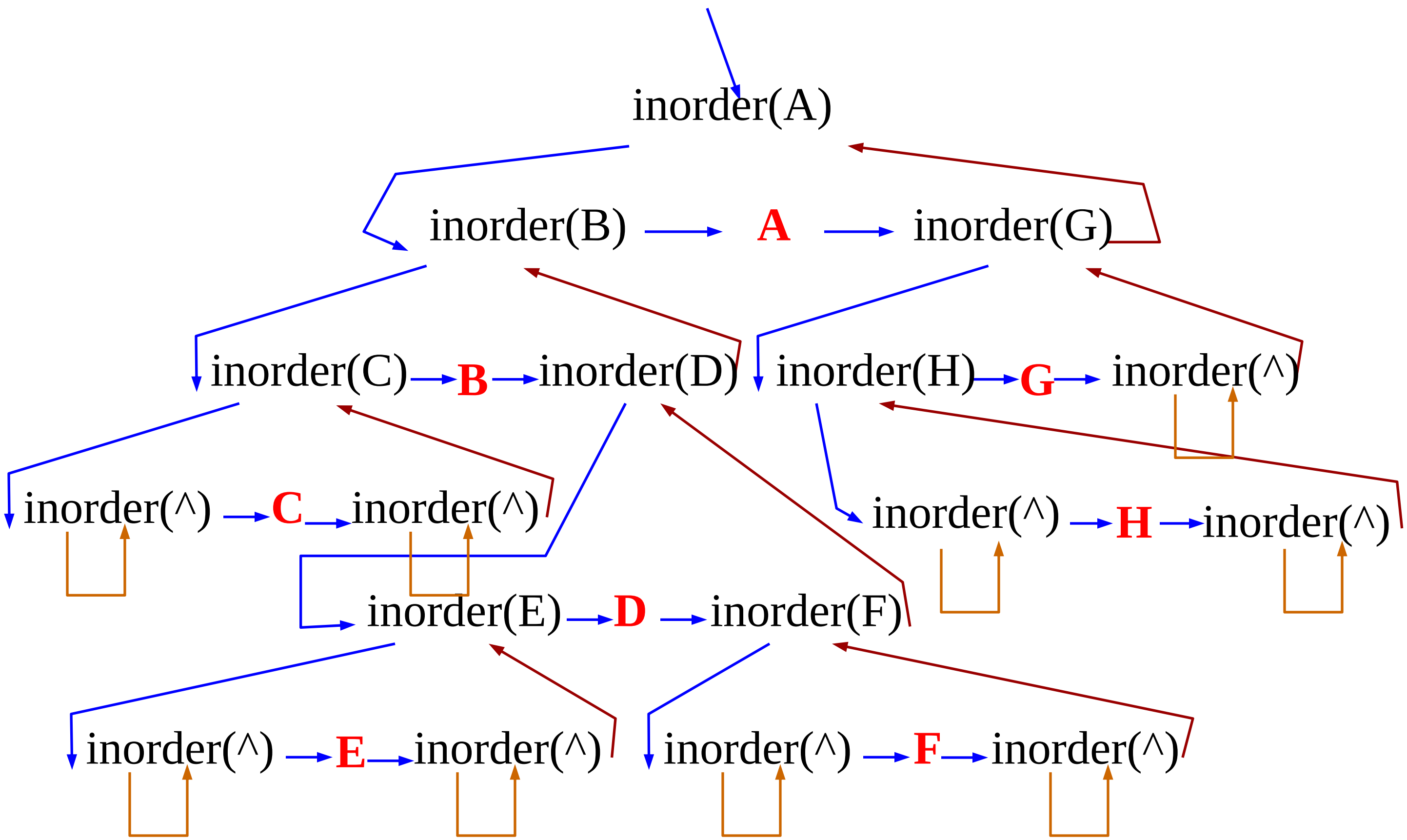
```
void PostOrder(btNode *T)
{
  if(T)
  {
   PostOrder(T->lChild);    //后序遍历左子树
   PostOrder(T->rChild);    //后序遍历右子树
   visit(T);                //访问根结点。
         //比如：打印当前结点 cout<<T->data<<" ";
  }
}
```

【顺序存储--后序遍历算法描述】

```
void postOrder( seqList T, int i )
{
  if( i<=T.listLen )
  {
   postOrder( T, 2*i );       //遍历左子树
   postOrder( T, 2*i+1 );   //遍历右子树
   if(T.data[i]!='/')
              cout<<T.data[i]<<"  ";
  }
}
```

- 模拟下图二叉树执行中序遍历算法的过程。





输出序列

CBEDFAHG

单选题 1分

设置

一棵高度为6的二叉树至多有(   )结点。

A 64

B 63

C 32

D 31

提交

5.3.3 二叉树的遍历算法应用

- 三种遍历次序输出二叉树所有结点值。
- （代码执行演示）
- 修改结点的访问操作，可得到多个问题的求解算法。

【例5.5】 求二叉树中度为2的结点数，并输出值。

【分析】利用一种遍历算法，修改访问函数，当前结点左右子树均不为空时即为度为2的结点，进行计数。

后面的算法用中序遍历完成：

【思考问题】

① 用先序、后序遍历能否求解此问题?

② 怎样求度为1的结点数?

【算法描述】

```
void  inOrder(btNode *T)
{
  if (T!= NULL)
  {
      inOrder(T->lChild);   //中序遍历左子树
      if ( T->lChild!=NULL && T->rChild!=NULL )
     {      //当前结点满足条件（度为2）时输出其值
        cout<< T->data;
        n++;  //n为全局变量
      }
     inOrder(T->rChild);  //中序遍历右子树
  }
}
```

【例4.6】 求二叉树结点数。

【分析】

方法1：设置一个全局变量，遍历二叉树，结点计数。

方法2：递归求解

T == NULL ——结点数=0；

否则：结点数 = 左子树结点数 ＋ 右子树结点数 + 1。

【算法一】

☞ 利用一种遍历算法，修改visit()函数以计数结点，下面算法以中序遍历实现。

```
void  inOrder(btNode *T)
{
  if (T!= NULL)
  {
      inOrder(T->lChild);   //中序遍历左子树
      n++;                  //n为全局变量
      inOrder(T->rChild);   //中序遍历右子树
  }
}
```

【思考问题】先序、后序遍历能否求解？

【算法二】

```
Int GetNodeNumber(btNode*  T )
{
    if ( T == NULL )
        return 0;
    else
      return( GetNodeNumber(T->lChild)
                  +
                 GetNodeNumber(T->rChild)
                  + 1 );
 }
```

【例5.7】 求给定二叉树的高度（深度）。

【分析】

（1）若T为空，则高度为0，遍历结束；

（2）否则，

①假设左、右子树能分别求出高度为hl、hr，则整个二叉树的高度为： max(hl, hr) + 1;

②对于左右子树高度的求解，可按照与整个二叉树相同的方式进行（递归调用）。

【算法描述】

```
int  btHeight( btNode *T )
{
    if ( T == NULL )
        return 0;
    else
        return max(btHeight(T->lChild), btHeight(T->rChild) )
                + 1 ;   //max()函数需要自行实现
}
```

- 【算法实现】

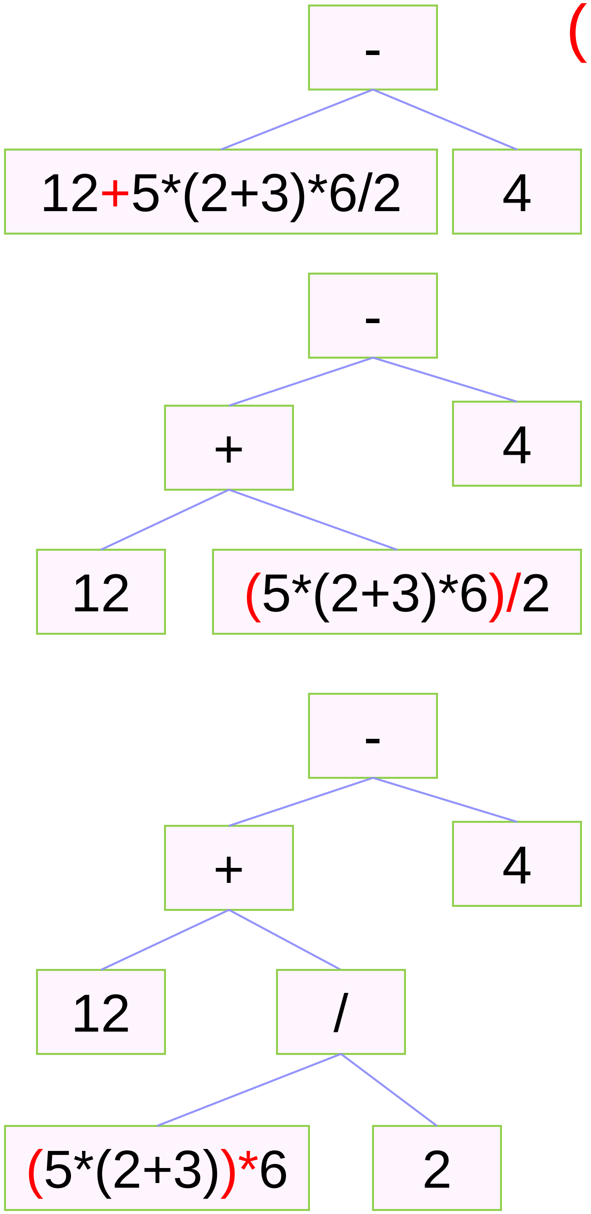
```
int btHeight(btNode* T)
{  int hl,hr;  //临时变量，分别保存左、右子树高度
   if(!T)
     return 0;
   else
   {  hl= btHeight(T->lChild);   //左子树高度
      hr= btHeight(T->rChild);  //右子树高度
      if(hl>hr)
              return hl+1;              //左子树较高
      else
              return hr+1;              //右子树较高
}}
```

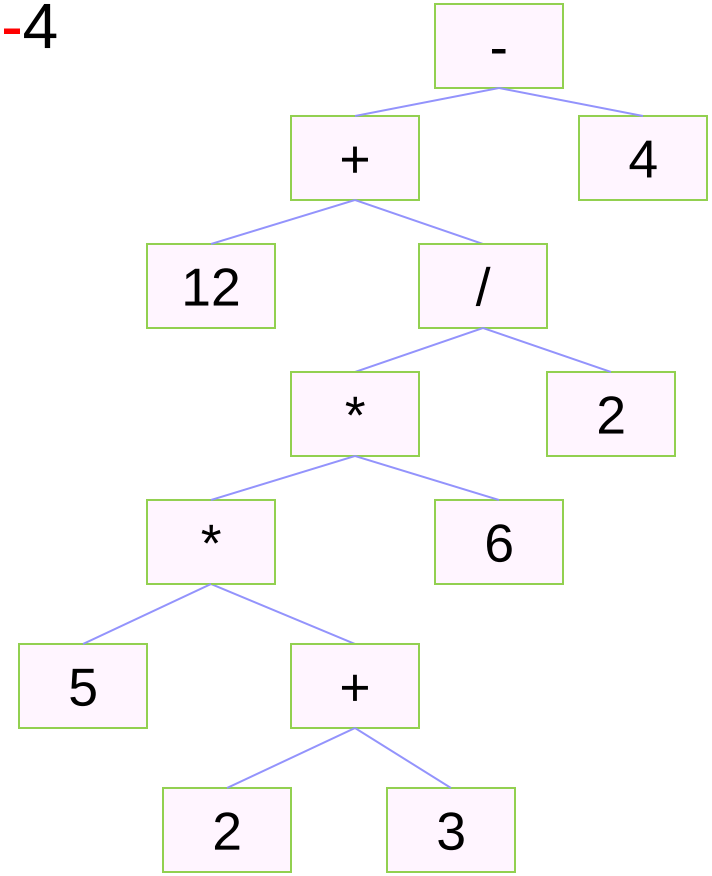
- 【二叉树表示算术表达式】
- 【例】将下列表达式转换为前缀表达式和后缀表达式。

  12+5*(2+3)*6/2-4

- 根据中缀表达式，画出二叉树，运算符作为子树的根结点，运算数作为叶子结点。
- 同一级优先级相同的运算符，先来的优先级高。如果同一级有多个相同优先级的运算符，将前面先来的运算符加括号结合，只留下最后一个运算符，以剩下的一个运算符作为子树根结点。例如本例：**(12+5*(2+3)*6/2)-4**

(12+5*(2+3)*6/2)-4





中序--中缀表达式：12+5*(2+3)*6/2-4

后序--后缀表达式：12 5 2 3 + * 6 * 2 / + 4 -

先序--前缀表达式：- + 12 / * * 5 + 2 3 6 2 4

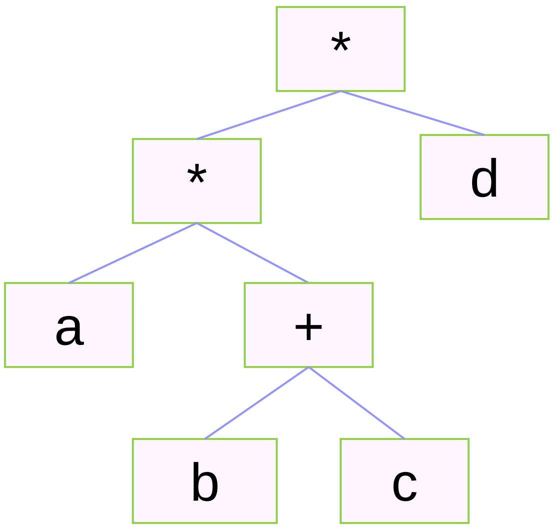
【练习】表达式 a * (b + c) * d 的后缀形式是（　）。

A. abcd*+*　B. abc+*d*

C. a*bc+*d　D. b+c*a*d

【答案】 B. abc+*d*

【提示】先对表达式做(a*(b+c))*d结合。画二叉树。



【例1】表达式 a*(b+c)-d 的后缀表达式是(　)。

A．abcd*+-　　B. abc+*d-

C. abc*+d-　　D. -+*abcd

【解】观察表达式，梳理出得到结果的一级运算符，本题为“-”。

如有多个优先级相同的一级运算符，按照先来的优先级高进行结合（加括号），最后变成1个一级运算符；

以这个一级运算符为根结点，两边的运算数作为左右孩子；

对左右孩子也做这样处理，以运算符为根结点，运算数为左、右孩子结点，画出二叉树。

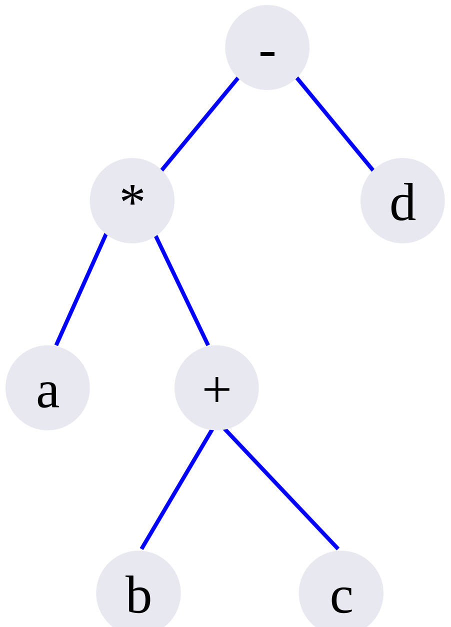
- **a*(b+c)-d** 对应的一棵二叉树如下：

① 先序遍历：前缀表达式：**-*a+bcd**

② 中序遍历：中缀表达式：**a*(b+c)-d**

③ 后序遍历：后缀表达式：**abc+*d-**

- 本题答案为**B**



其它问题

- 取孩子、取双亲、取兄弟
- 先序、中序、后序输出二叉树中所有结点值及其对应层次、序号
- 输出从根结点到每个叶子结点的路径（利用栈、队列、或数组）。输出根结点到当前结点路径。
- 求度为0、1、2的结点数
- 求最近公共祖先
- 插入、删除子树（需规定原有子树的处理）
- ....

单选题 1分

设置

二叉树最多只有一个度为1的结点。

A 错

B 对

提交

5.3.4 二叉树的创建与销毁

- 学习任何数据结构，要实现其各种算法，第一步就要学会创建这种数据结构，在实践环节中这是非常重要的。
- 二叉树的顺序存储结构只要用一个数组就可以了，需要注意的是要把普通二叉树补齐为完全二叉树然后存放在数组中，且数组的0单元最好不存放树的结点，二叉树的这种顺序存储结构创建非常简单，这里不专门介绍。
- 因为二叉树常用二叉链表存储结构，接下来我们介绍基于二叉链表结构的二叉树的创建和销毁。

1. 控制台交互输入创建二叉树

- ☞ 由键盘交互输入二叉树的结点数据来创建二叉树。
- ☞ 本方法基于二叉树先序遍历序列创建二叉树，即键盘输入时按二叉树的先序遍历次序输入结点数据，没有子树时用特殊符号表示，下面的算法中以特殊符号 **“/”**表示没有子树。
- ☞ 二叉树的创建由**2**个函数合作完成：

【创建子树函数】

void createSubTree(btNode *&p, int k)

//p为子树根结点

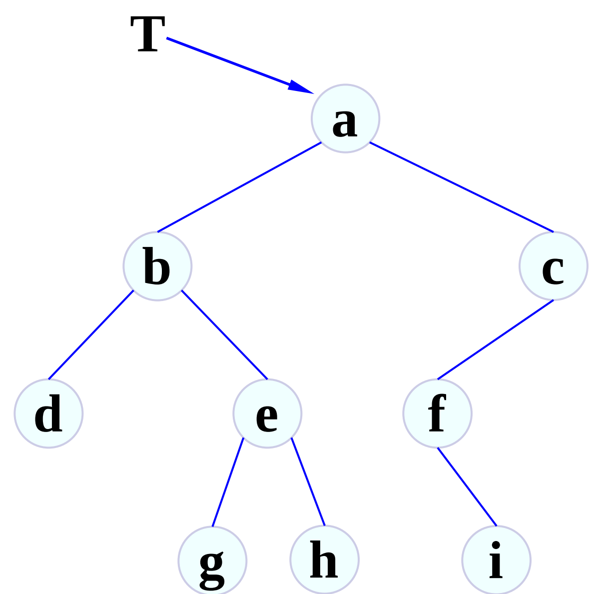
//k=1—创建左子树；k=2—创建右子树

【创建二叉树主控函数】

void createBTConsole(btNode *&T)

- ☞ 算法实现代码见：**createBiTree.h**

- 键盘输入举例：
  - 例如下图二叉树，先序遍历次序为：abdeghcfi，以“/”表示无子树，则相应的键盘输入为：abd//eg//h//cf/i///。其中d后面的2个“//”表示结点d无左子树，也无右子树；结点g和h后面的2个“//”也表示这2个结点既无左子树，又无右子树；结点f后面的1个“/”表示结点f无左子树；结点i后面的3个“///”，其中前2个表示结点i无左子树和右子树，最后1个表示结点c无右子树。



2. 数据文件输入创建二叉树

- 上面介绍的交互式输入创建二叉树一般只适用于树的结点数较少时，当树的结点数较多时，交互输入容易出错，浪费时间，且容易造成内存泄漏。
- 下面介绍一种基于文本文件的二叉树创建方法，即将二叉树的结点信息保存在一个文本文件中，然后用程序自动读入来创建二叉树。
- 定义文本文件的格式：
  ① 标识行：**BinaryTree**—标识这是一个二叉树的数据文件。
  ② 结点行：每个结点一行，结点从上到下严格按照先序遍历次序排列。每行3列。第1列为结点数据；第2列标识有无左子树，1—有左子树，0—无左子树；第3列标识有无右子树，1—有右子树，0—无右子树。
  ③ 注释行、空行：注释行以"//"开始。

- 对下图二叉树，完整数据文件如下：

BinaryTree

a 1 1

b 1 1

d 0 0

e 1 1

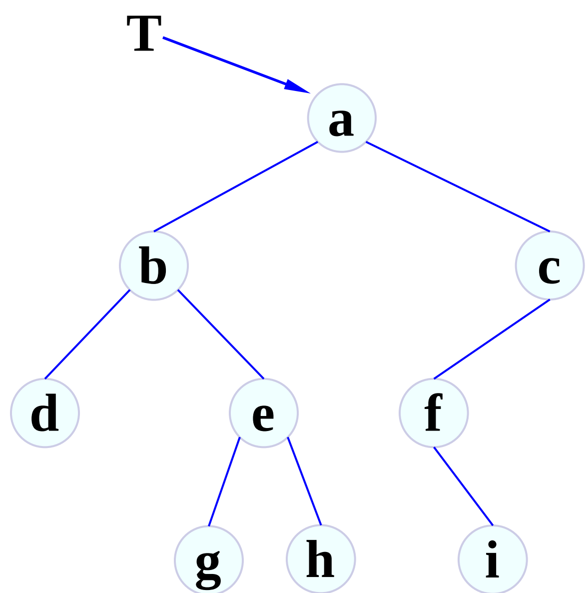
g 0 0

h 0 0

c 1 0

f 0 1

i 0 0



- 数据文件的扩展名随意，只要按文本文件读写即可，例如上述文件不妨命名为**BT9.btr**。

- 本算法二叉树的创建也由2个函数合作完成：

【结点数据读入数组函数】

- ☞ 因为是按先序次序递归创建二叉树，在创建过程中又要记录读取数据的行数，直接从文件中读取数据创建时处理不方便，所以我们先把文件中的数据读取到一个二维数组中，再从数组中读取数据来创建二叉树。从文件读取数据到数组的函数描述如下：

```
bool ReadFileToArray( char fileName[], char strLine[100][3], int & nArrLen )
    // fileName[]存放文件名
    // strLine[][3]存放结点的二维数组，数组的3列对应数据文件的3列
    // nArrLen返回二叉树结点的个数
```

【从数组数据按先序次序创建二叉树】

```
bool CreateBiTreeFromFile( btNode* & pBT,
  char strLine[100][3], int nLen, int & nRow )

    //strLine[100][3]--保存结点数据的二维数组
    //nLen--结点个数
    //nRow--数组当前行号
```

☞ 算法实现及数据文件格式见实际文件和代码。

3. 中序序列加先序（后序）序列创建二叉树

3. 二叉树的销毁

```
void destroyBt(btNode *&T)
{
   if(T)
   {
     destroy(T->lChild);    //递归销毁左子树
     destroy(T->rChild);    //递归销毁右子树
     delete T;              //释放当前结点
   }
}
```

5.3.5 二叉树非递归遍历算法

- 使用栈将递归算法改为非递归遍历算法。

- 非递归先序遍历

```
void PreOrderNR(btNode *T)
{
    btNode *p;
    Stack<btNode*, 100> S;

    p=T;
    while( p!=NULL || S.empty()==false )
    {
      if( p!=NULL )
      {
            cout<<p->data<<", ";  //访问根结点
            S.push(p);            //p 指针入栈，以备遍历右子树
            p=p->lChild;          //遍历左子树
      }
```

```
else      //p为空，但栈不空
    {
            //p为空时，取栈顶元素到p，
            //即上一层结点指针到p
        S.getTop(p);
        S.pop();        //出栈
        p=p->rChild;    //遍历右子树
    }
  }
}
```

- 【思考问题】打印任意结点到根结点的路径上的结点值。

- 非递归中序遍历

```
Void InOrderNR(btNode *T)
{
    btNode *p;
    Stack<btNode*, 100> S;

    p=T;
    while( p!=NULL || S.empty()==false )
    {
      if(p!=NULL)
      {
                //p 指针入栈，当前根结点入栈，
                //以备访问根结点以及遍历右子树
            S.push(p);
            p=p->lChild;          //遍历左子树
      }
```

```
else  //p为空，但栈不空
{
            //p为空时，取栈顶元素，
            //即上一层结点（子树根）指针到p
        S.getTop(p);
        cout<<p->data<<", ";  //访问根结点

        S.pop();                    //出栈
        p=p->rChild;                //遍历右子树
   }
  }
}
```

- 非递归后序遍历

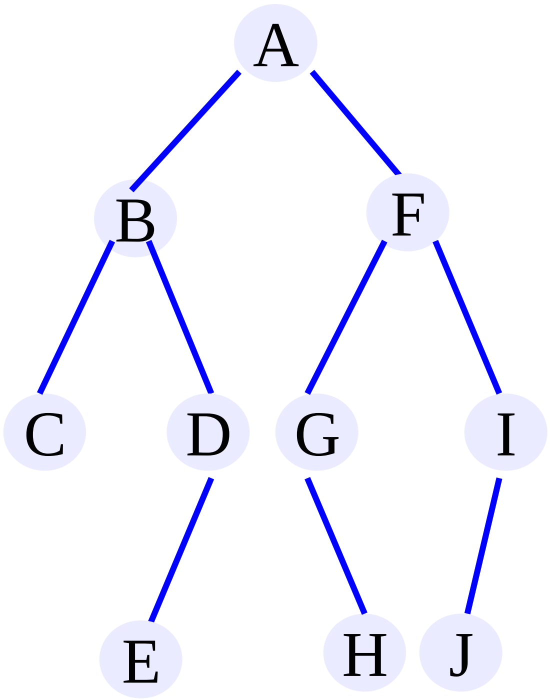
```
Void PostOrderNR(btNode *T)
{
    btNode *p;
    Stack<btNode*, 100> S;

    int tag[100];     //标记遍历左子树、右子树

    p=T;
    while( p!=NULL || S.empty()==false )
    {
      if(p!=NULL)
      {                    //p 指针入栈，当前根结点入栈，
                          //以备访问根结点以及遍历右子树
             S.push(p);
             tag[S.top]=0;       //标记遍历左子树，
                                 //即p的左子树已经遍历
             p=p->lChild;        //遍历左子树
      }
```

```
else            //p==NULL 但是栈不空
{               //取栈顶，但不退栈，以遍历p的右子树
        S.getTop(p);
        if( tag[S.top]==0 )
        {               //说明p的左子树已经遍历，右子树尚未遍历
                tag[S.top]=1;  //设置当前结点遍历右子树标记
                p=p->rChild;   //遍历右子树
        }
        else  //tag[S.top]==1, 说明p的左右子树皆已经遍历，
        {
                S.getTop(p);    //取栈顶
                cout<<p->data<<", ";  //访问某子树根结点
                S.pop();        //根结点已经访问，出栈
                p=NULL;         //取出的根结点p已经访问，
                    //置为空，回去循环取栈顶的下一个元素
        }
}
    }
}
```

5.3.6 二叉树的层次遍历

- 从根结点开始，自上而下、自左往右逐个访问结点。
- 如右图，层次遍历序列：

  ABFCDGIEHJ
- 特点：先访问的结点，其孩子结点也将先被访问。
- 层次遍历需要借助队列实现。
- 层次遍历类似图的广度优先遍历。



■ （1）二叉链表的层次遍历算法描述

```
void hieTraverse( btNode *T )
{
  btNode *p;
    Queue Q;        //定义队列
    enQueue(Q,T);    //根结点入队
    while(!queueEmpty(Q))
    {
      getFront(Q,p);  //取队头到p
      visit(T); //访问根结点。如cout<<p->data<<" ";
      if(p->lChild)
                enQueue(Q, p->lChild);  //p左孩子入队
      if(p->rChild)
                enQueue(Q, p->rChild);  //p右孩子入队
      outQueue(Q);  //当前结点出队
    }
}
```

（2）分立数组存储的层次遍历算法描述

```
void hieTraverse( seqList L, int i )
{ if( i<=0 || i>L.listLen ) return;
   int j;
  Queue Q;        //定义队列
  enQueue(Q,i);   //根结点入队
  while(!queueEmpty(Q))
   {
     getFront(Q,j);  //取队头到j
     visit(L.data[j]); //访问根结点。如cout<<L.data[j]<<" ";
     if(2*j<=L.listLen)
            enQueue(Q, 2*j);     //j左孩子入队
      if(2*j+1<=L.listLen)
            enQueue(Q, 2*j+1);  //j右孩子入队
     outQueue(Q);  //当前结点出队
   }
}
```

子曰：学而时习之，不亦说乎？有朋自远方来，不亦乐乎？人不知而不愠，不亦君子乎？

单选题 1分

设置

完全二叉树，根结点从1编号，则编号为47的结点X的双亲结点的编号为（ ）。

A 22

B 23

C 24

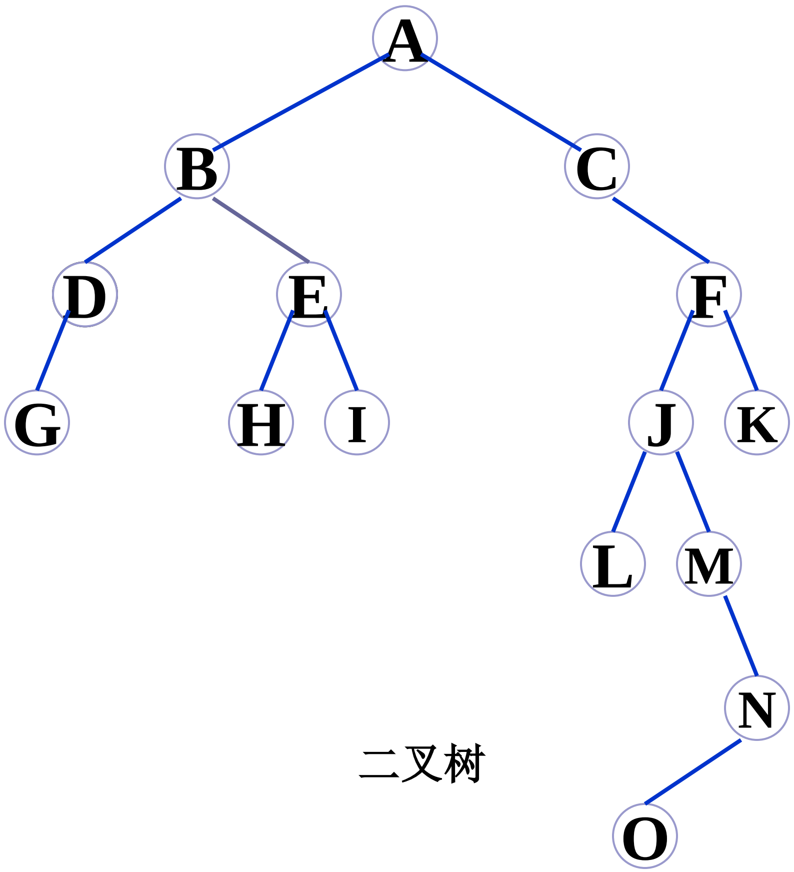
D 94

提交

5.4 线索二叉树

- **Thread Binary Tree**
- 在线性结构中，
  - 除了最后一个结点，其它结点有且仅有一个直接后继结点；
  - 除了第一个结点，其它结点有且仅有一个直接前驱结点。
- 二叉树是非线性结构，没有上述性质；
- 但是，在对二叉树进行先序、中序、后序遍历时，结点的访问次序是确定的，按其遍历次序对结点进行排列，则会得到一个线性结构的结点序列。
  - 在该序列中，除第一个结点外，每个结点有且仅有一个直接前驱结点；
  - 除最后一个结点外，每个结点有且仅有一个直接后继结点。

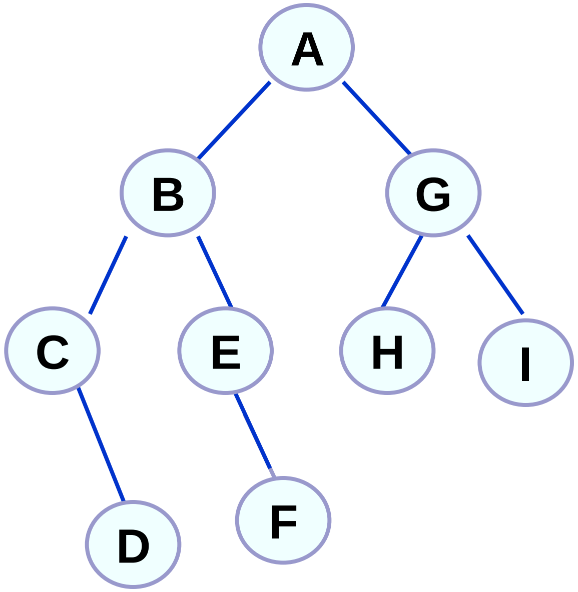
【例】求下图结点E在先序、中序、后序次序中的直接前驱和直接后继结点。



二叉树

- 先序次序：ABDGEHICFJLMNOK
  前驱G；后继H；
- 中序次序：GDBHEIACLJMONFK
  前驱H；后继I；
- 后序次序：GDHIEBLONMJKFCA
  前驱I；后继B；

【思考问题】下图二叉树中结点H在先序、中序和后序次序中的前驱和后继分别是什么？



- 算法中如何寻找一个结点在某种遍历下的前驱和后继呢？容易想到的方法可能是：

  ① 遍历方法：通过指定次序的遍历运算寻找指定结点的前驱或后继。显然，这类方法太费时间，因此不宜采用。--费时间

  ② 增设前驱和后继指针：在每个结点中增设两个指针，分别指示该结点在指定次序下的前驱或后继。这样，就可使前驱和后继的求解较为方便，但这是以空间开销为代价的。--费空间

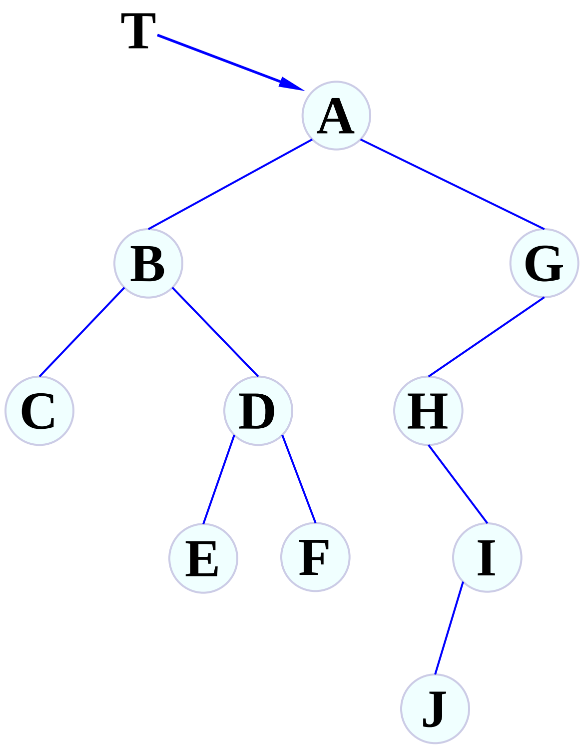
- 是否存在既能少花费时间，又不用花费多余的空间的方法呢？下面要介绍的第三种方法就是一种尝试—利用二叉链表结构的n+1个空指针域。

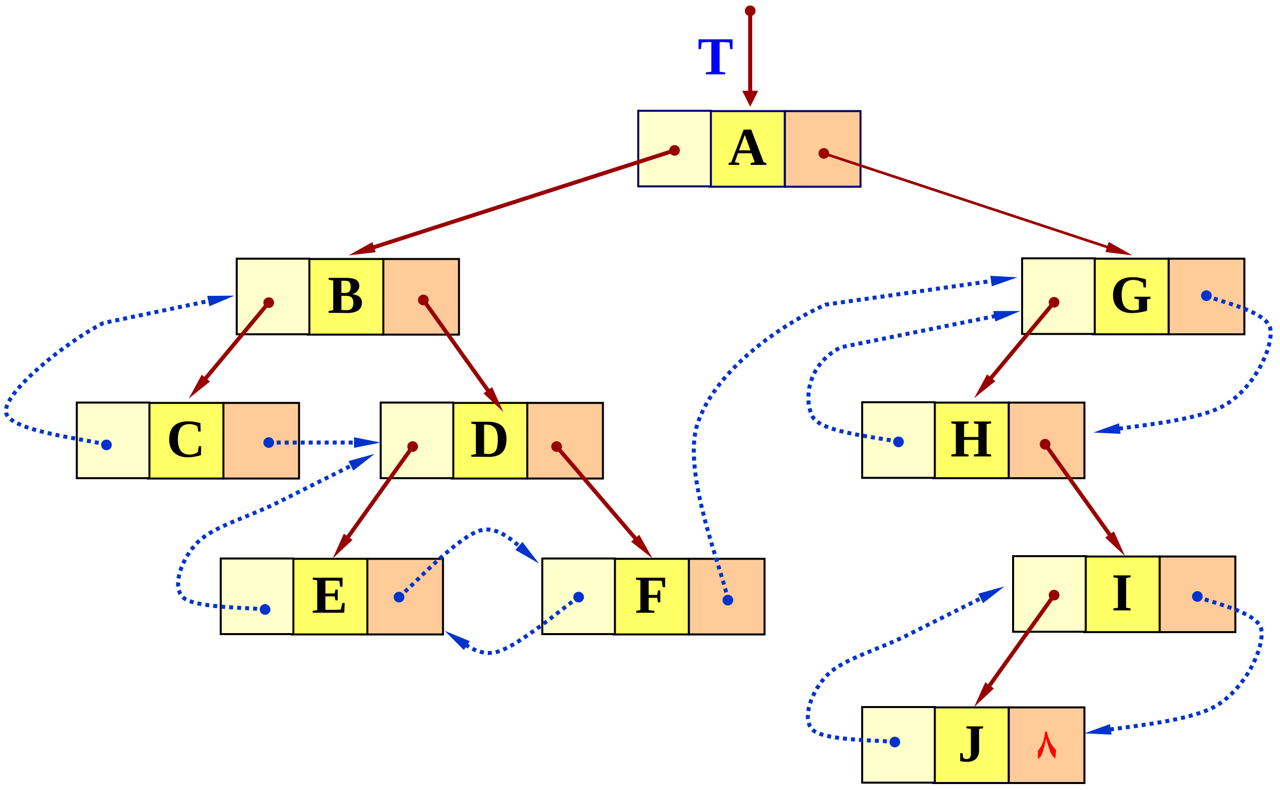
5.4.1 线索二叉树结构

- 利用二叉链表结构中的空指针域：将二叉链表中n+1个空指针域改为指向结点的前驱和后继。
- 具体地说，就是将二叉树各结点中的
  - 空的左孩子指针域改为指向其前驱结点，
  - 空的右孩子指针域改为指向其后继结点。
- 称这种新的指针为（前驱或后继）线索（Thread）。
- 所得到的二叉树被称为线索二叉树。
- 将二叉树转变成线索二叉树的过程被称为线索化。
- 线索二叉树根据所选择的次序可分为先序、中序和后序线索二叉树。

【例如】左图二叉树的先序线索二叉树的二叉链表结构如右图所示，其中线索用虚线表示。

☞ 先序遍历次序为：**ABCDEFGHIJ**

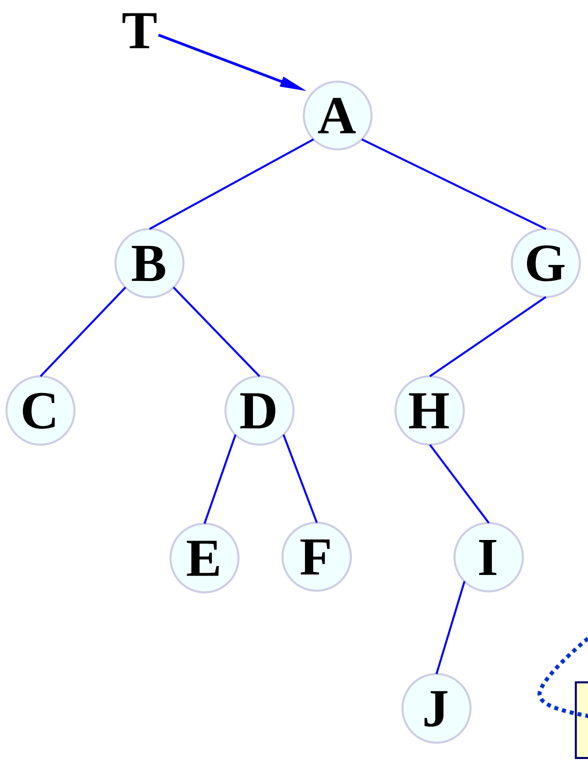


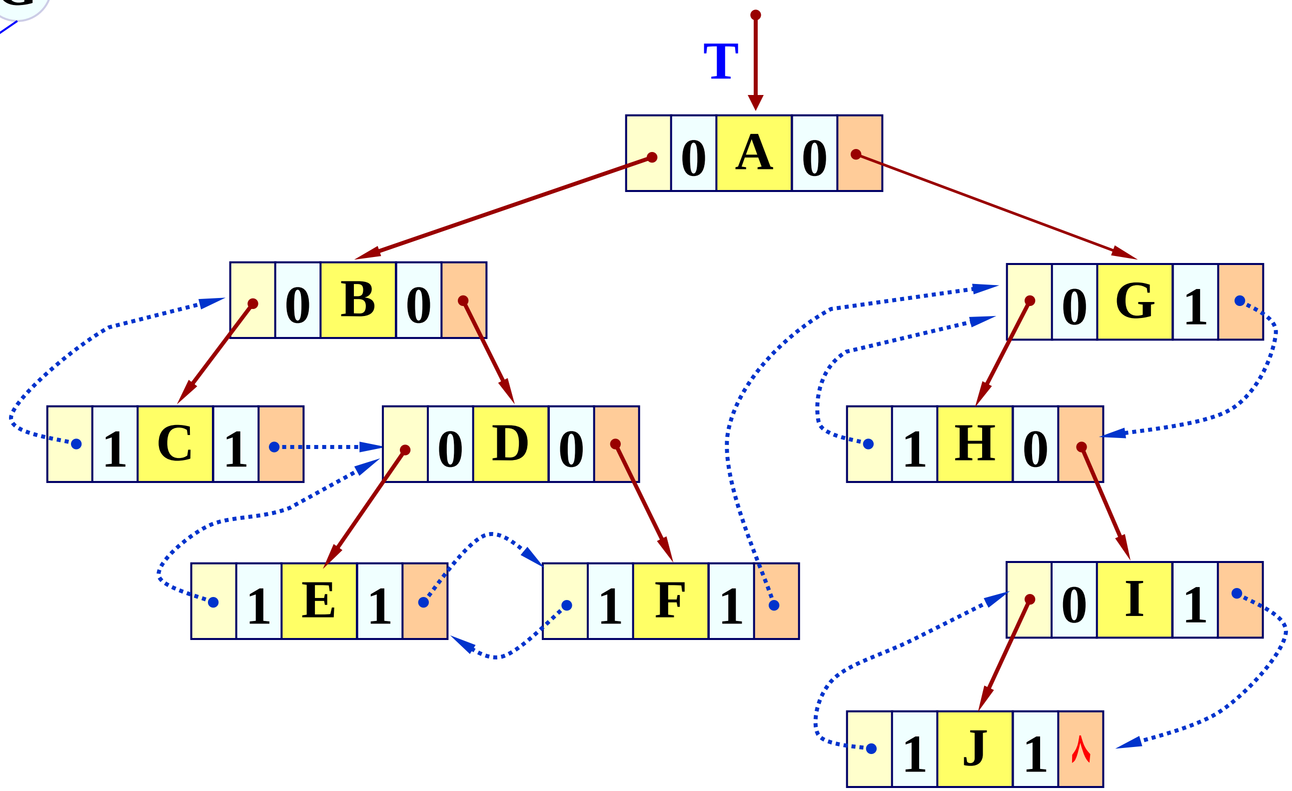


- 算法中如何分辨是孩子指针还是线索呢？
  - ☞ 虽然由图中可以“直观地”区分出来，但在算法中却不行。
- 解决办法：
  - ☞ 左、右孩子指针域分别加标志**lTag**和**rTag**。具体约定如下：
  ① **lTag=0：lChild**指示该结点的左孩子。
  ② **lTag=1：lChild**指示该结点的前驱。
  ③ **rTag=0：rChild**指示该结点的右孩子。
  ④ **rTag=1：rChild**指示该结点的后继。
  - ☞ 增加标志后结点结构如下图所示：

lChild	lTag	data	rTag	rChild

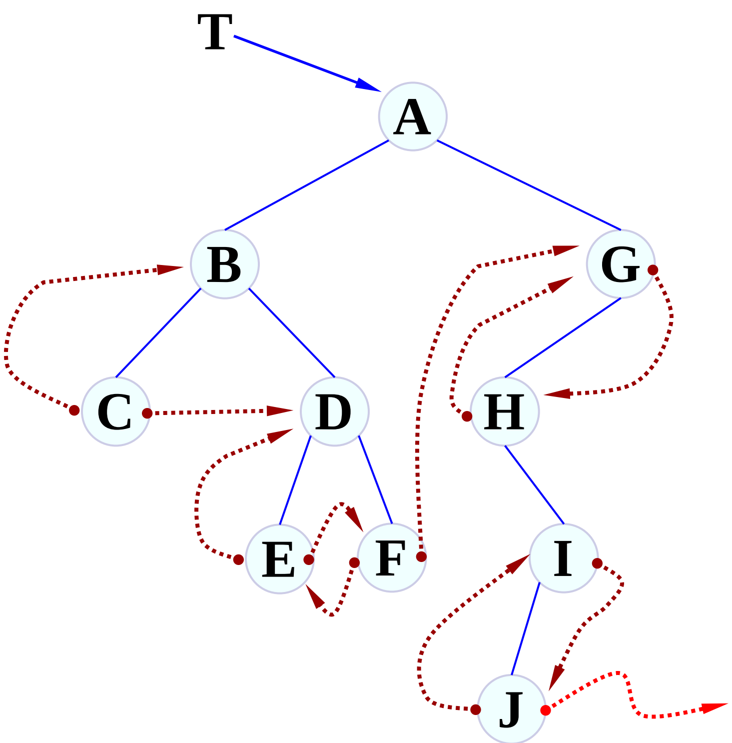
■ 上例中增加标志后的先序线索二叉树结构如图：





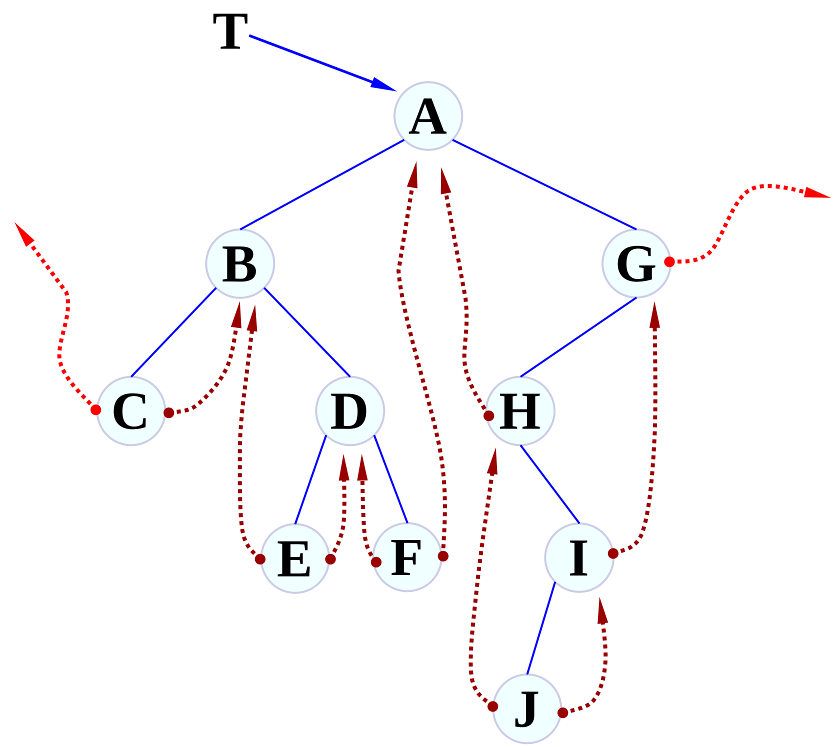
线索二叉树的简略画法

- 直接在原图上用虚线画出线索。左侧为前驱，右侧为后继。



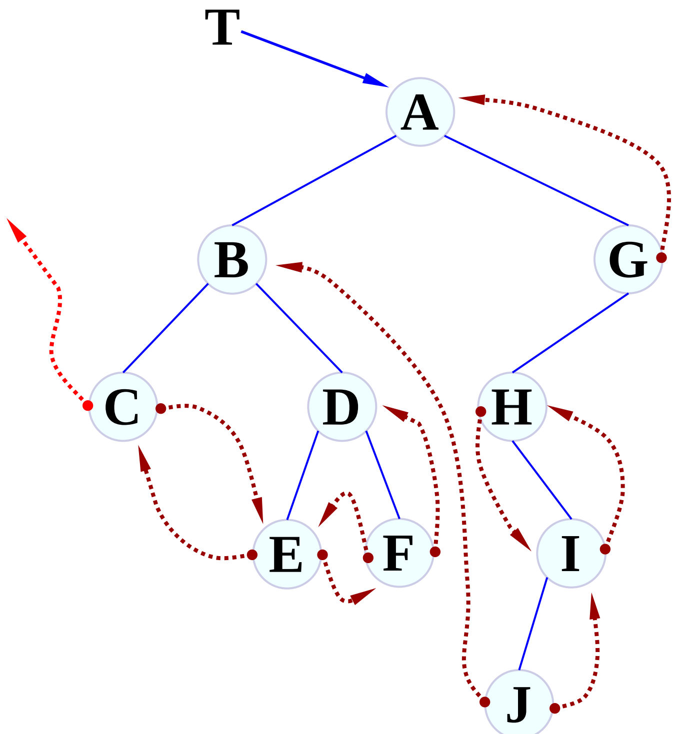
先序线索二叉树

先序序列：ABCDEFGHIJ



中序线索二叉树

中序序列：CBEDFAHJIG



后序线索二叉树

后序序列：CEFDBJIHGA

【例1】左右子树都存在的二叉树，先序线索化后，存在几个空指针?

A、0　B、1　C、2　D、不确定

【例2】左子树不存在的二叉树，先序线索化后，存在几个空指针?

A、0　B、1　C、2　D、不确定

【例3】中序线索化后，存在几个空指针?

A、0　B、1　C、2　D、不确定

【例4】左右子树都存在的二叉树，后序线索化后，存在几个空指针?

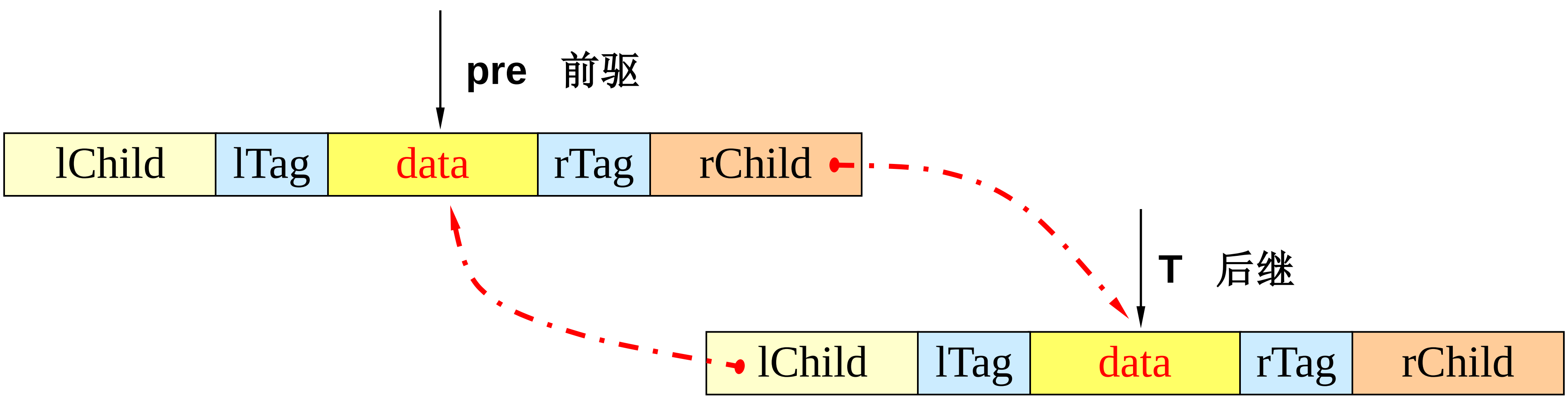
A、0　B、1　C、2　D、不确定

【例5】右子树不存在的二叉树，后序线索化后，存在几个空指针?

A、0　B、1　C、2　D、不确定

5.4.2 二叉树的线索化

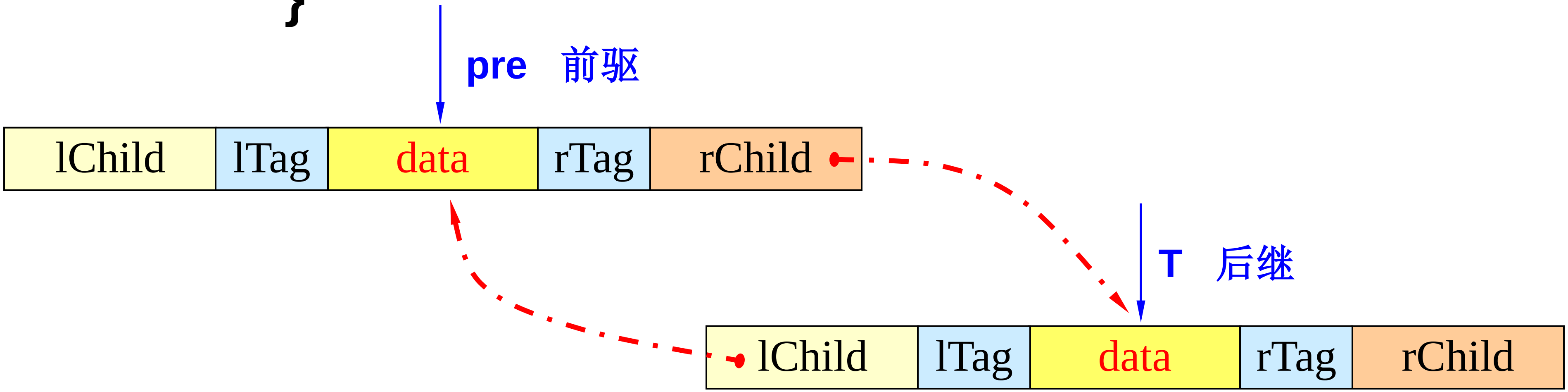
- 二叉树的线索化即将普通二叉树转换为线索二叉树。
- 解决思路：当前结点指针T，前驱结点指针pre。则pre为T的前驱，T为pre的后继，如图。要解决的问题是确定pre->rChild是否要变更为指向T的后继线索；T->lChild是否要变更为指向pre的前驱线索。



pre->rChild

- 如果pre->rChild!=NULL，保持为右孩子指针，置rTag=0；
- 如果pre->rChild==NULL，使其指向后继结点，成为后继线索，且置 rTag=1；即：

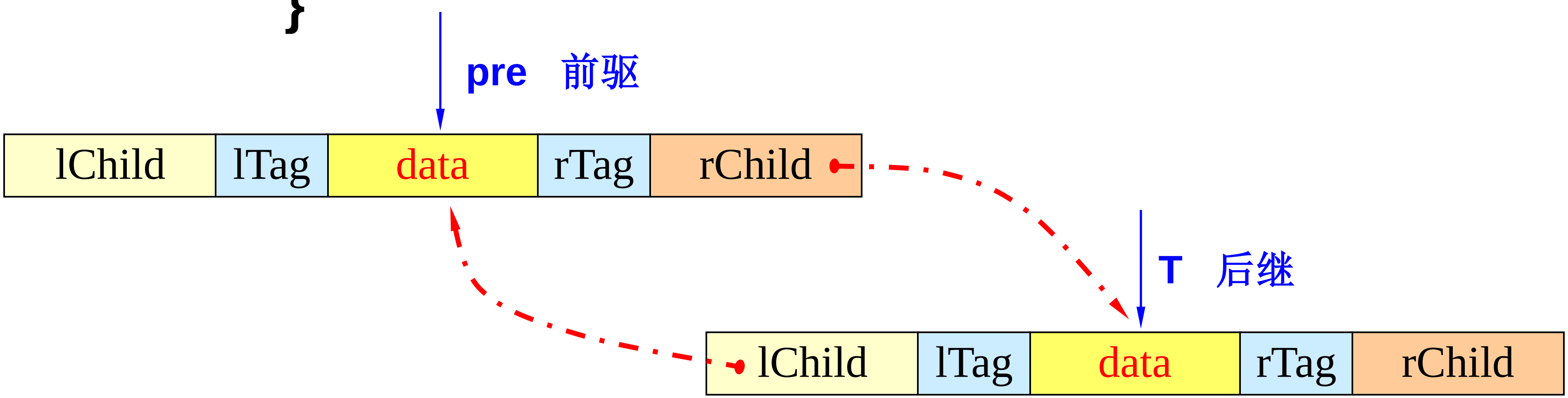
```
if(pre!=NULL && pre->rChild==NULL )
{
    pre->rChild=T;   //pre的右孩子指针为线索
    pre->rTag=1;
}
```



T->lChild

- 如果T->lChild!=NULL，保持为左孩子指针，置lTag=0；
- 如果T->lChild==NULL，使其指向前驱结点，成为前驱线索，且置 lTag=1；

```
if( T!=NULL && T->lChild==NULL)
  {
        T->lChild=pre;   //T的左孩子指针为线索
        T->lTag=1;
  }
```



- 先序线索化算法描述

```
Void PreThreading(btNode* T, btNode* &pre)
{       //pre为前一次访问过的结点指针；T为当前正在访问的结点指针
    btNode *lp,*rp;  //左右孩子临时指针
    if(T!=NULL)
    {
      lp=T->lChild;  //防止递归处理之前T的左、右孩子指针被变为线索
      rp=T->rChild;
      if(T!=NULL && lp==NULL)
      {
                T->lChild=pre;   //T的左孩子指针变为线索
                T->lTag=1;
      }
      if(pre!=NULL && pre->rChild==NULL)
      {
                pre->rChild=T;   //pre的右孩子指针变为线索
                pre->rTag=1;
      }
      pre=T;                         //T变为已经访问的结点
      PreThreading(lp,pre);  //递归处理T的左子树
      PreThreading(rp,pre);  //递归处理T的右子树
    }
}
```

- 中序线索化、后序线索化算法类似，分别在中序遍历、后续遍历算法基础上改造而成。

5.4.3 线索二叉树中前驱和后继的求解

- 先序线索二叉树
  - 前驱（不可求）
  - 后继（易求）
- 中序线索二叉树
  - 前驱（可求）
  - 后继（可求）
- 后序线索二叉树
  - 前驱（易求）
  - 后继（不可求）

- 共有三组六个问题，其中有**2**个问题不可求解。

1. 先序后继的求解

【问题描述】先序线索二叉树中当前结点指针为P，求P的先序后继结点指针。

【分析】假定P的左右子树根结点指针分别为P_L和P_R，则先序遍历的顺序为： $P\ P_L P_R$，可知：

① 如果P的左子树不空，即P->lTag==0，则P的后继为其左子树的根结点，即：P->lChild。

② 否则，如果P的右子树存在，即P->rTag==0，则P的后继为右子树的根结点，即：P->rChild；

③ 否则，P的右子树不存在，即P->rTag==1，则P->rChild为后继线索指针。

以上②和③情况可以合并，即不管P的右子树是否存在，P->rChild都是P的后继结点指针。

- 【求先序后继算法描述】

```
btNode* PreSuc(btNode* P)
{
  if(P->lTag==0)
        return P->lChild;
  else
        return P->rChild;
}
```

2. 中序后继的求解

【问题描述】中序线索二叉树中当前结点指针为P，求P的中序后继结点指针。

【分析】假定P的左右子树根结点指针分别为P_L和P_R，则先序遍历的顺序为：P_L P P_R，可知：

① 如果P的右子树为空，即P->rTag==1，则P->rChild直接为其中序后继线索，即后继结点指针。

② 否则，如果P的右子树存在，即P->rTag==0，则P的后继为其右子树中序遍历第一个访问的结点。P的右子树根结点指针为P_R，P的后继为右子树P_R中第一个左子树为空的结点，即P的后继结点在P_R的左子树上，或为P_R自己。

- 【求中序后继算法描述】

```
btNode* InSuc(btNode* P)
{
  if(P->rTag==1)          //P的右子树为空
      return P->rChild;
  else                    //P的右子树存在
  {
      S=P->rChild;        //取P的右子树根结点指针
          //在PR中寻找第一个左子树为空的结点
      while(S->lTag==0)
         S=S->lChild;
      return S;
  }
}
```

【例5.8】设计按先序遍历先序线索二叉树的非递归算法，且不使用栈。

【分析】

- 首先求出先序线索遍历要反回的第一个结点指针，显然先序遍历中即为根结点指针；
- 循环求出每个结点的后继，并访问。

【算法描述】

```
Void PreTraverseThr(btNode* T)
{
  btNode* p=T;
  while(p!=NULL)
  {
     visit(p);  //访问p结点。
                //可能简单为 cout<<p->data<<“ ”。
     p=PreSuc(p);   //求p的后继结点指针
  }
}
```

- 【思考问题】
  ① 求解其它情况下的前驱、后继算法
  ② 中序线索遍历、后序线索遍历算法
  ③ 通过线索二叉树插入、删除结点算法

...悄悄的我走了，正如我悄悄的来，我挥一挥衣袖，不带走一片云彩。徐志摩

Very quietly I take my leave, As quietly as I came here;
Gently I flick my sleeves, Not even a wisp of cloud will I bring away .

单选题 1分

设置

在有n个结点的二叉树的二叉链表表示中，空指针数为（　　）。

A n-1

B n

C n+1

D 2n

提交

5.5 树和森林

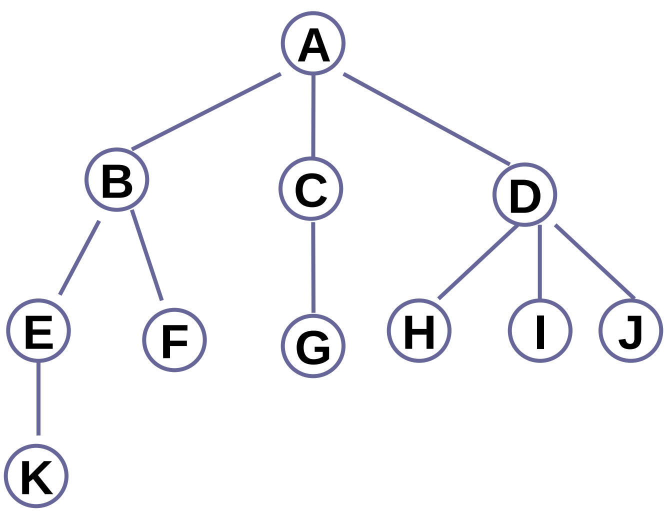
- 本节讨论树和森林的有关内容，包括
  - ☞树和森林的存储形式，
  - ☞树（森林）与二叉树之间的相互转换，
  - ☞树（森林）的遍历等。

- 为描述方便起见，在不作特别说明的情况下，树包括森林。

5.5.1 树（森林）的存储结构

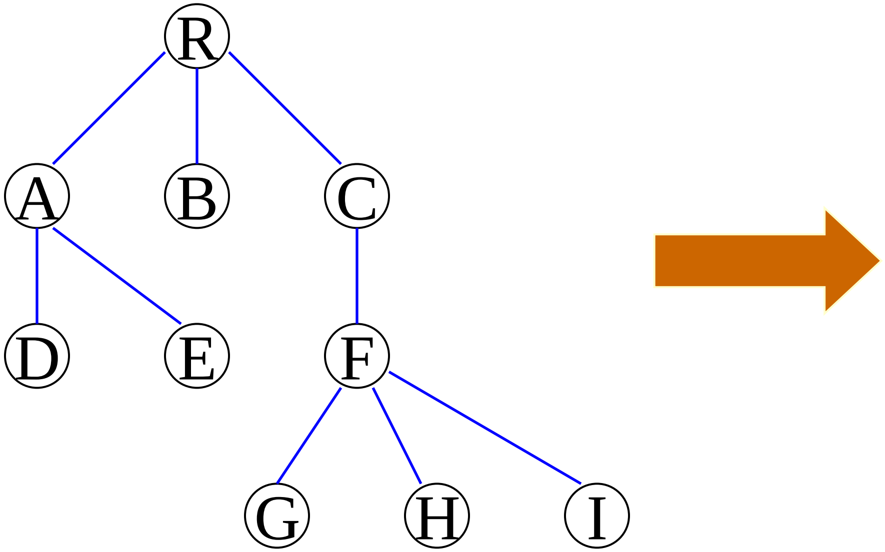
1. 双亲表示法

- 以连续空间存储树的结点，每个结点除数据域外，附设**指示器指示其双亲结点的位置**。例：



	data	parent
0	A	−1
1	B	0
2	C	0
3	D	0
4	E	1
5	F	1
6	G	2
7	H	3
8	I	3
9	J	3
10	K	4

- 例：



相对位置	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	I	6

■ 树（森林）双亲表示的存储结构描述

【结点结构定义】

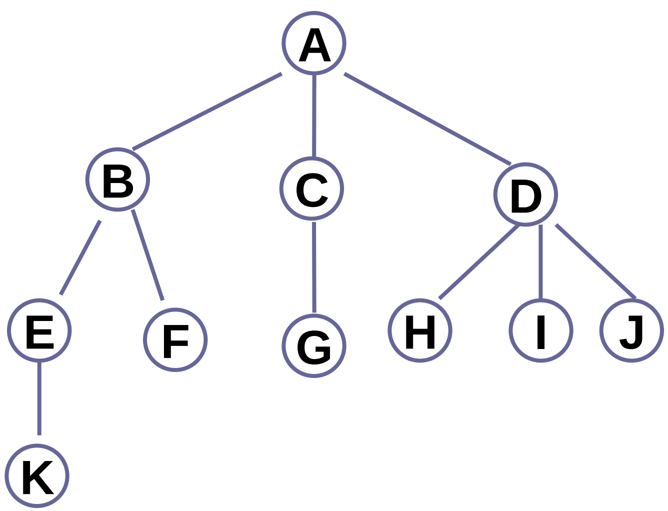
```
#define MaxLen 100
typedef  struct PTree {
        elementType data;        // 数据域
        int parent;              // 指示双亲位置
} PTNode;
```

【树结构定义】

```
typedef struct {
        PTNode node[MaxLen];  // 结点数组
        int n;                // 结点总数
} Ptree;
```

- 双亲表示法优点：
  - 简洁/形式一致；找父结点和祖先容易；
- 双亲表示法缺点：
  - 找孩子结点费时，需要偏历整个结构。
  - 插入和删除时需注意维护结点之间关系。

- 孩子结点求法：
  - 根据当前结点的位置值，扫描整个结构，如果其它结点的双亲位置值与此位置相同，则为它的孩子结点。
- 例下图求结点D的孩子结点：
  - D的位置为3，扫描，当双亲指示为3时，为D的孩子结点，即：H、I、J 3个结点为D的孩子结点。

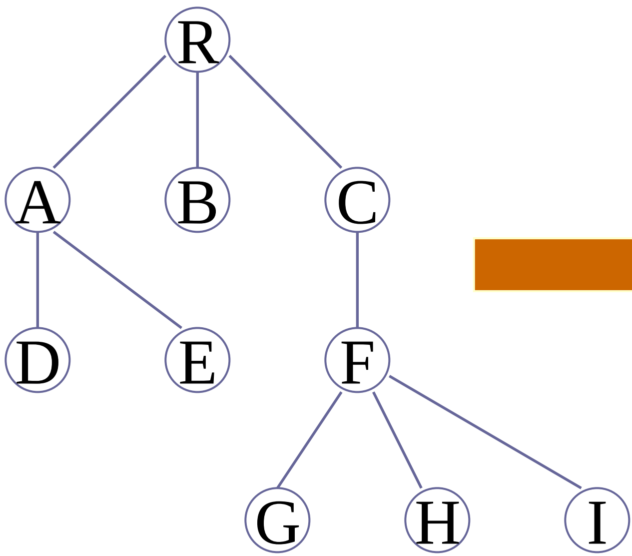


2. 孩子链表表示法

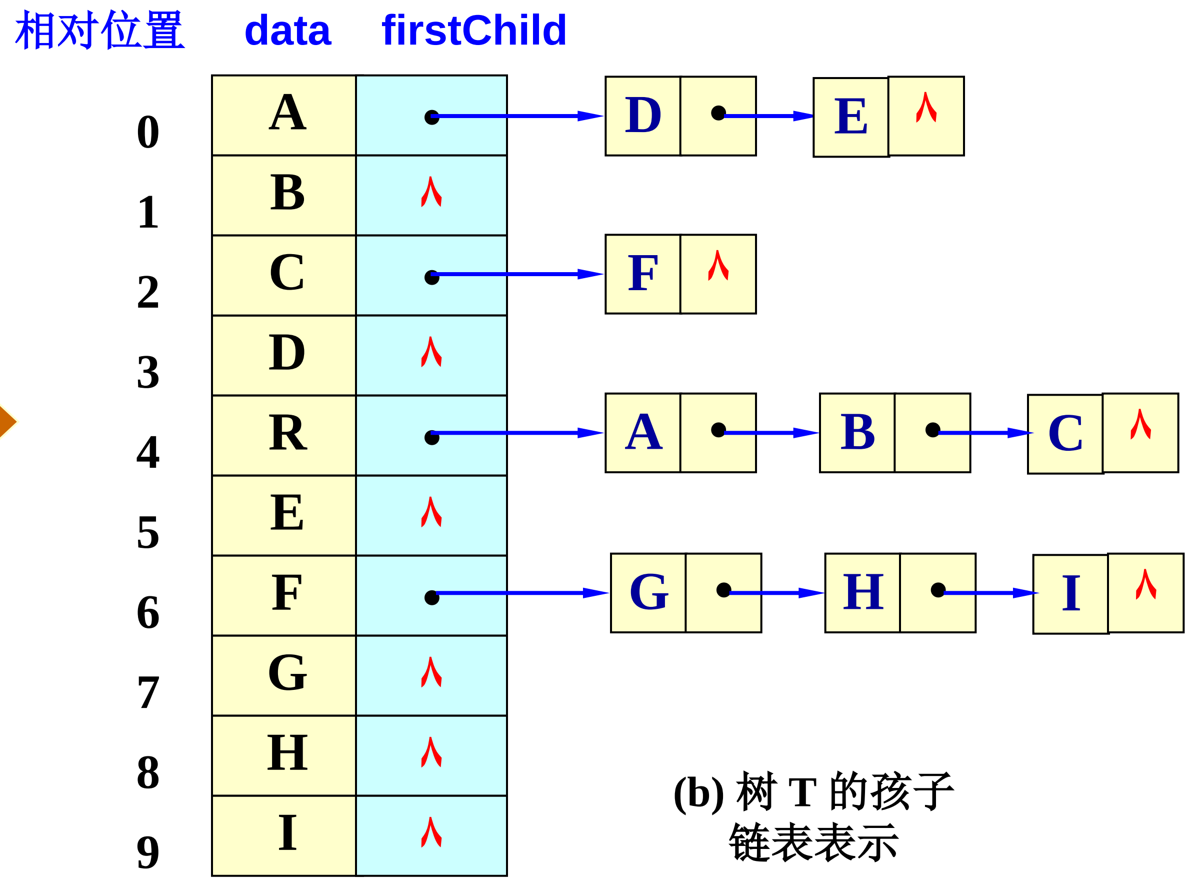
- 结点存放在一张表中，表有**2**个字段：数据域和指针域
  - 数据域—存放结点数据元素；
  - 指针域—指向第一个孩子的指针，叶结点为空；
- 一个结点的所有孩子用一条链表表示。

- 参见图的邻接链表存储。两者存储结构相同。

■ 例：



(a) 树 T



(b) 树 T 的孩子链表表示

- //孩子链表结构描述

【孩子链表的结点结构定义】

```
typedef struct CTNode {
  elementType childData;   // 孩子结点数据或位置。
  struct CTNode *next;       // 指向下一个孩子结点。
}clNode,  *ChildPtr;
```

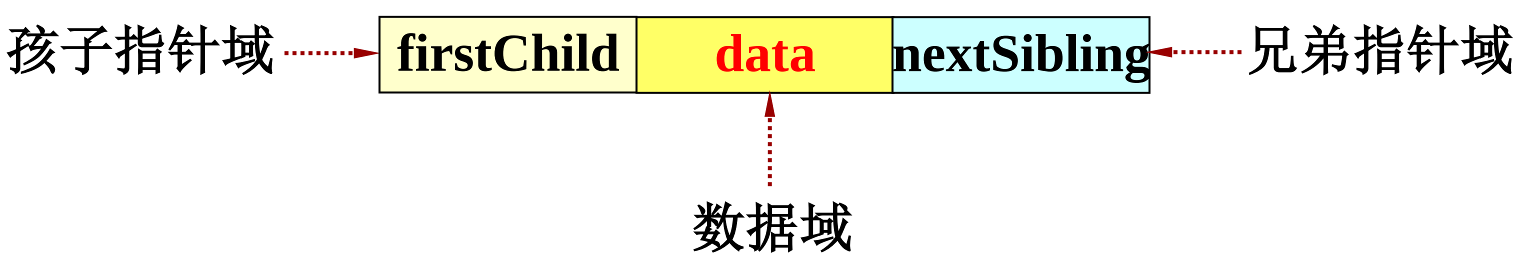
【数组元素类型描述（孩子链表的头结点结构）】

```
typedef struct {
  elementType data;        // 存放结点数据元素。
  clNode*  firstChild;     // 孩子链表的头指针。
}clElement;                //结点数组元素类型

clElement tree[ MaxLen]; // 孩子链表的头结点数组。
```

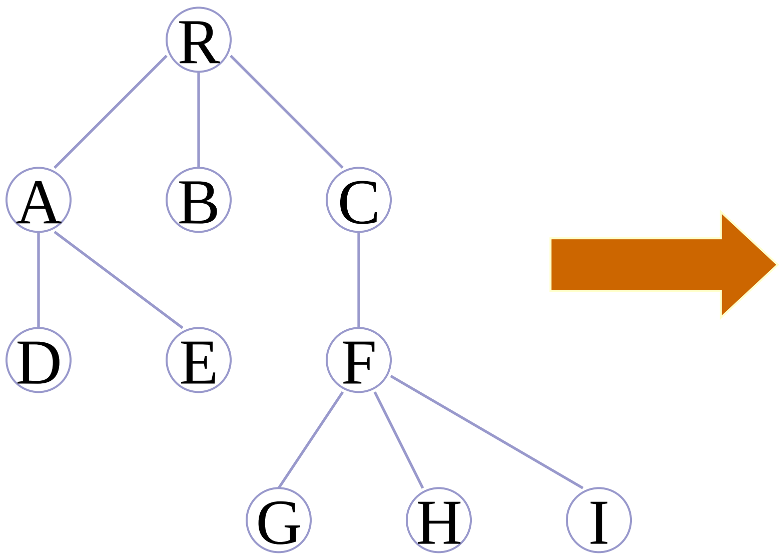
- 孩子链表表示法优点：
  - 找孩子结点、后代容易
- 孩子链表表示法缺点：
  - 结点重复（每个结点存储2次）；
  - 结构不一致

3. 孩子--兄弟链表表示法

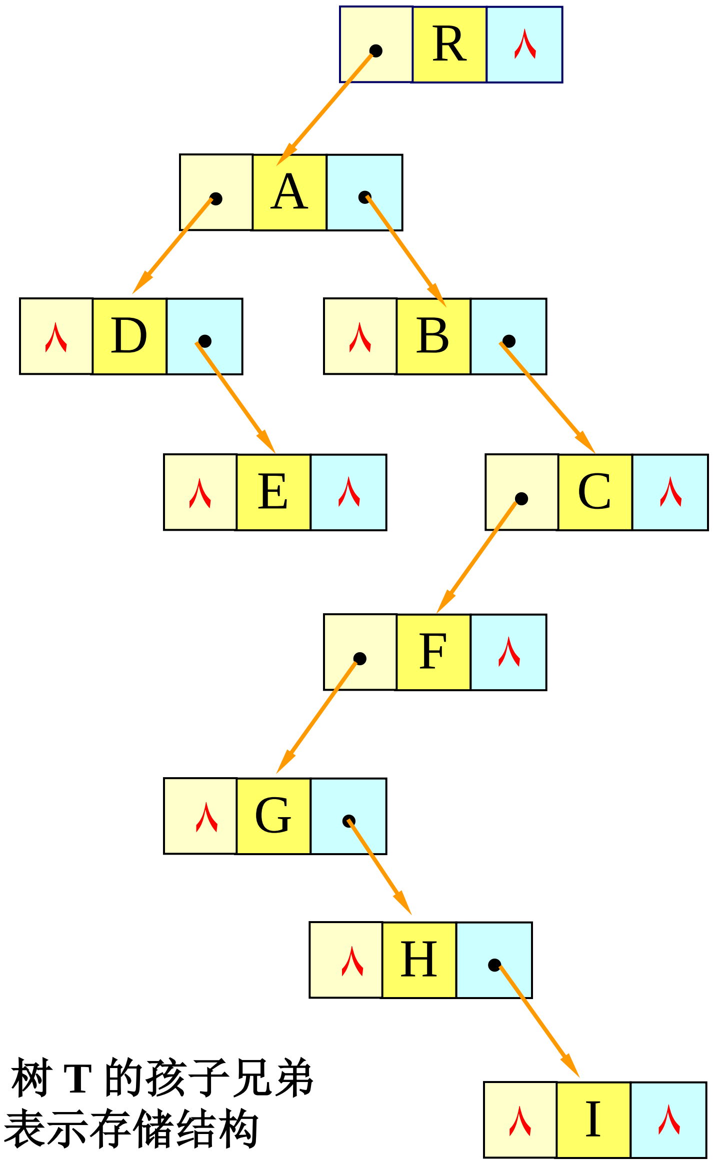
- 又称二叉树表示法、二叉链表表示法；
- 以二叉链表作为树的存储结构；
- 每个结点有 3 个域：
  - 数据域 **data**，存放结点的数据元素；
  - 孩子域 **firstChild**，存放结点从左开始的第一个孩子的地址；
  - 兄弟域 **nextSibling**，存放当前结点右边第一个兄弟的地址。
- 孩子--兄弟链表表示法结点结构，如图：



■ 例：



(a) 树 T



(b) 树 T 的孩子兄弟表示存储结构

- 孩子--兄弟链表表示存储结构描述

```
typedef struct CSNode
{
    elementType  data;
        // 从左至右的第一个孩子和下一个兄弟。
    struct CSNode  *firstChild,  *nextSibling;
} csNode,  *csTree;
```

- 这种存储方法事实上是将树转换为二叉树存储，见后面的树（森林）转换转换为二叉树。

5.5.2 树和森林与二叉树的转换

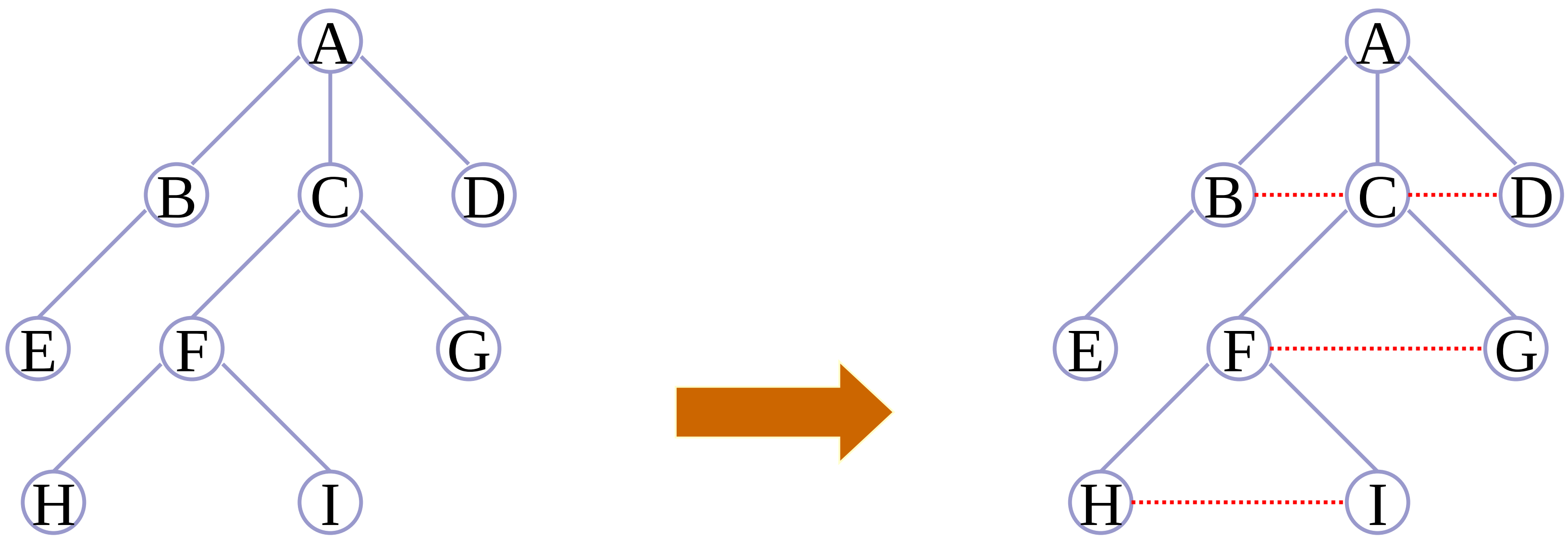
- 基本思想
  - 树（森林）按一定方法转换为二叉树，转换是唯一的；
  - 二叉树还原为树（森林），转换也是唯一的；
  - 这样，任何对树或森林的操作都可以转换为二叉树以后实现，然后再还原为树（森林）。

- 树和二叉树都可以用二叉链表表示，即物理存储结构相同，只是解释（理解）不同而已。

1. 树转换为二叉树

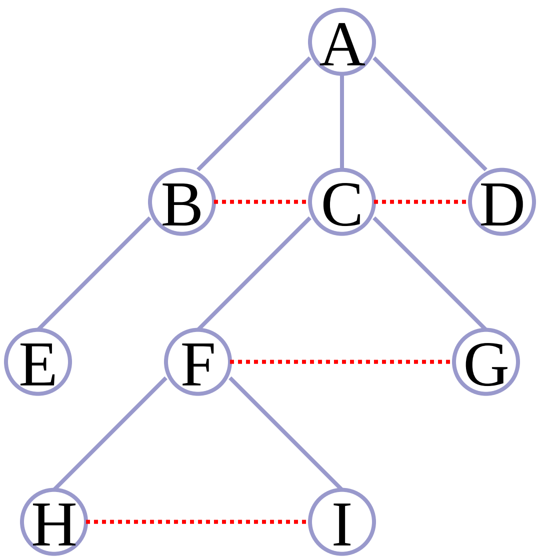
☞步骤

(1) 加线：同一双亲结点的所有孩子两两之间加一条连线；

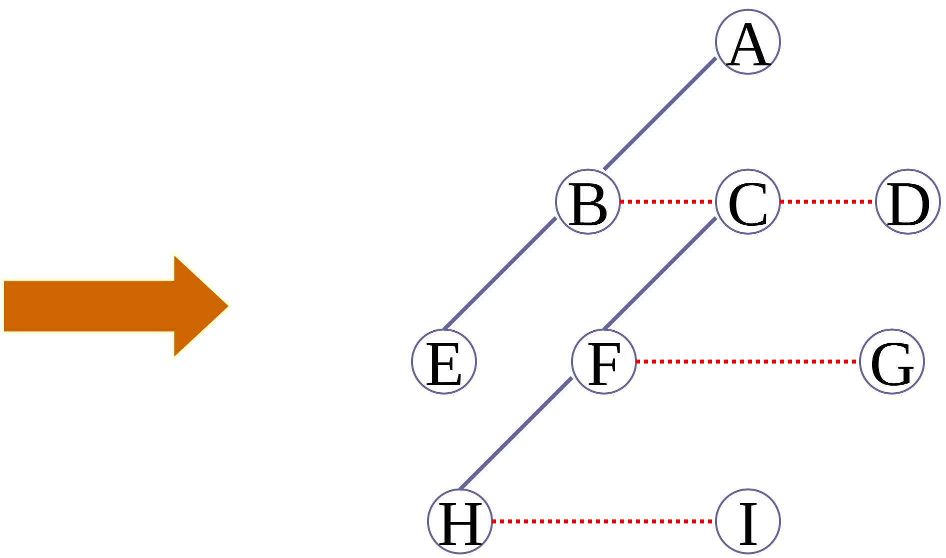


(a) 树 T　　(b) 加线后

(2) 抹线：任何结点，除了其最左的孩子外，抹掉此结点与其它孩子之间的连线（边）--转为二叉树；

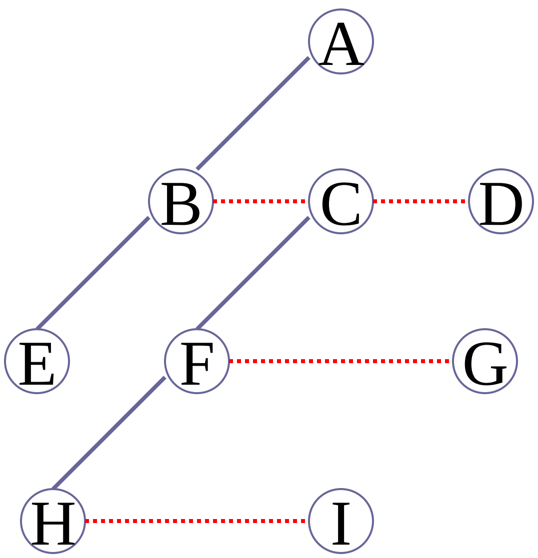


(b) 加线后

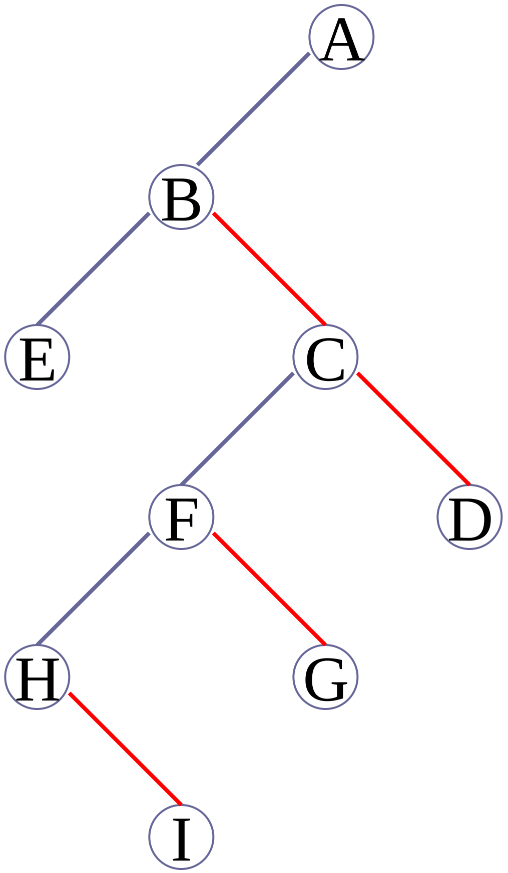


(c) 抹线后—二叉树

(3) 调整：调整结点位置，使之层次分明。



(c) 抹线后—二叉树



(d) 调整后的二叉树

☞转换的二叉树特点

(1) 转换后的二叉树，根结点下只有左子树，而没有右子树；

(2) 转换后的二叉树，各结点的左孩子是其原来最左的孩子，右孩子则为其原先的下一个兄弟。

☞这种方法产生的二叉树是唯一的。

2. 森林转换为二叉树

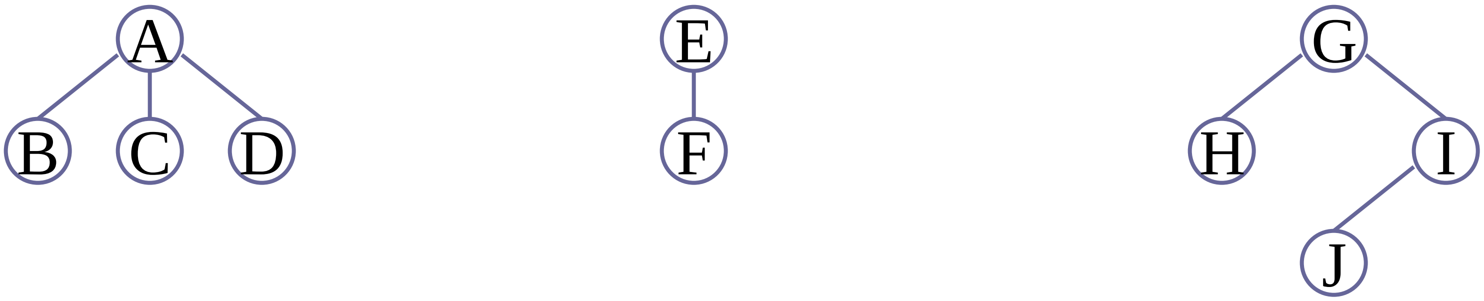
☞**步骤：**

(1) 转换： 每棵树转换为二叉树；

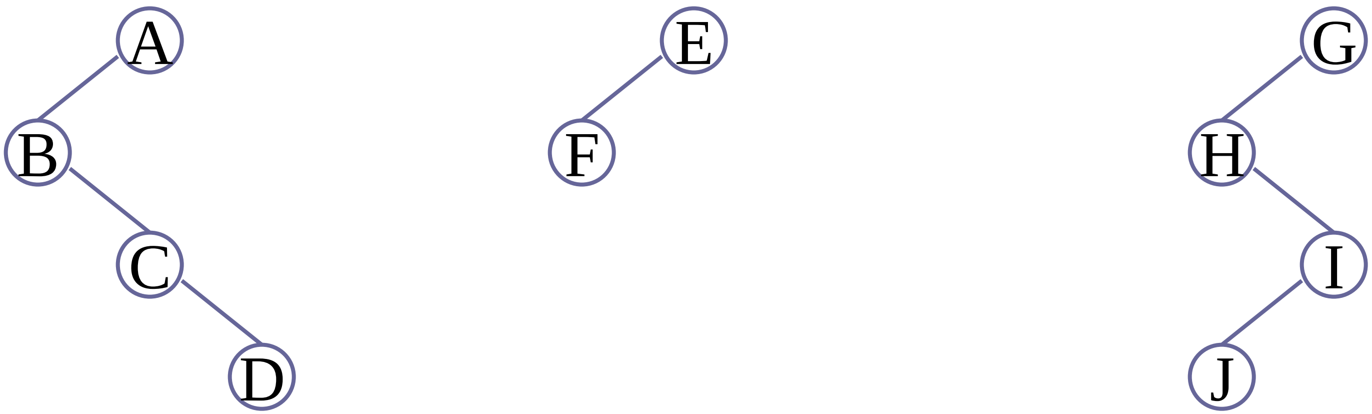
(2) 连线： 将每棵二叉树的根结点视为兄弟结点，加连线；

(3) 调整: 以最左边二叉树的根结点，作为最后的根结点， 调整结点位置，使之层次分明。

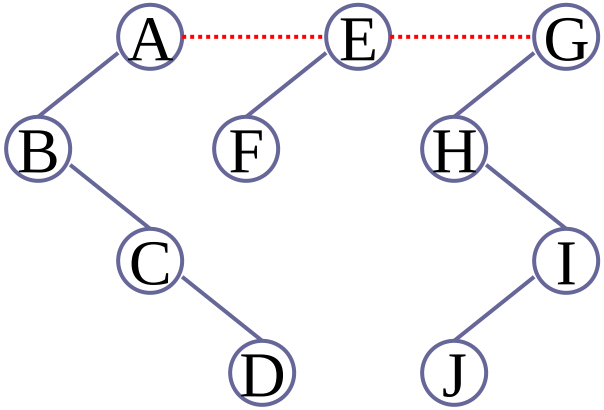
- 森林这样产生的二叉树是维一的。



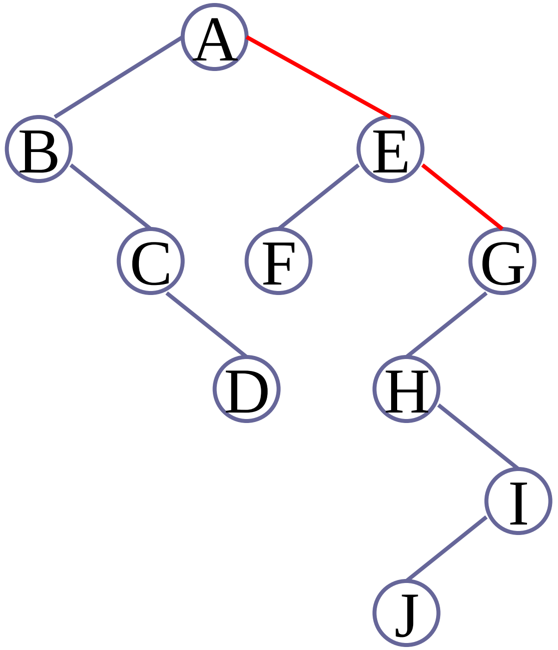
(a) 森林



(b) 每棵树转为二叉树



(c) 连线后 — 二叉树



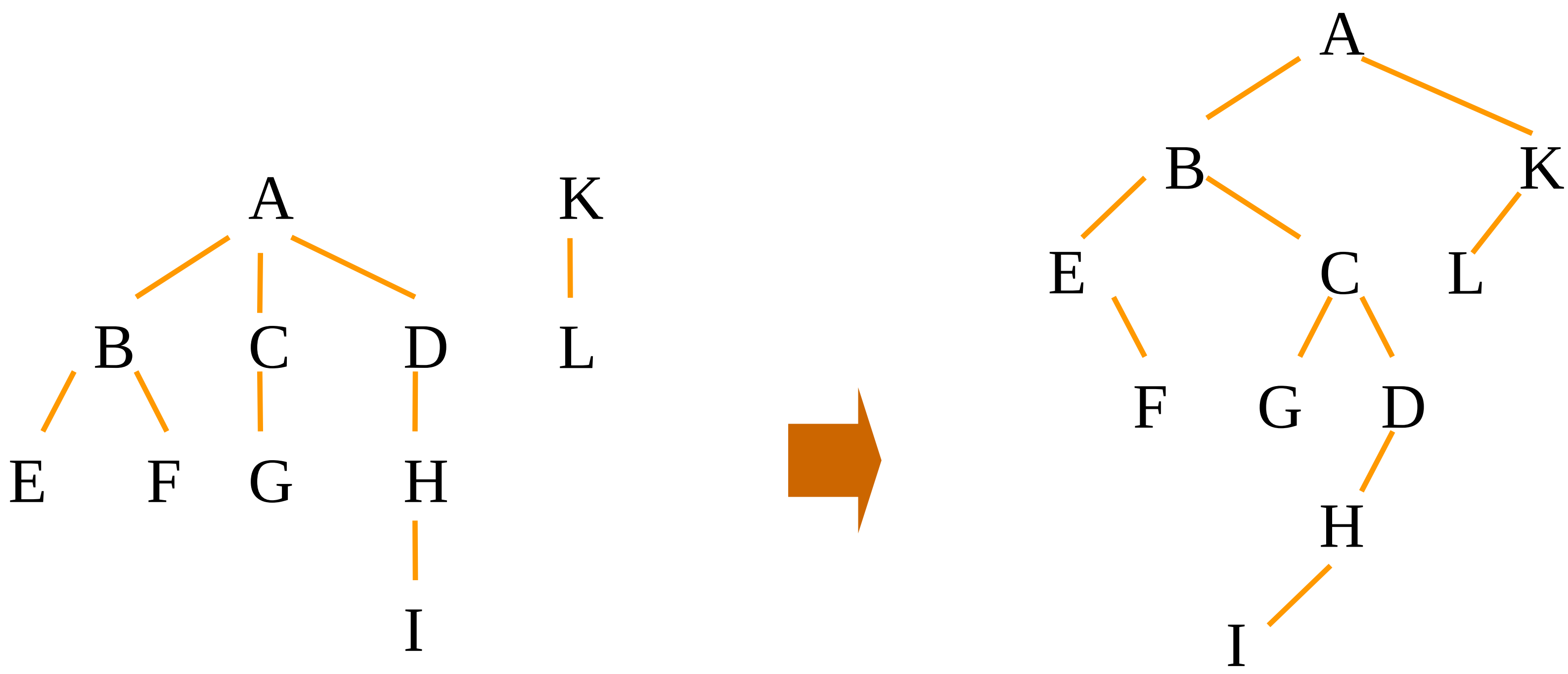
(d) 调整后的二叉树

- 树（森林）转换为二叉树的一般描述
  - ☞树（森林）F=（T1，T2，…，Tm）=> 二叉树 BT
  - ☞如果F不空，则：

    (1) T1的根 => BT的根

    (2) T1的子树 => BT的左子树

    (3) （T2，…，Tm）=> BT的右子树


A
K
B
C
D
L
E
F
G
H
I
A
B
K
E
C
L
F
G
D
H
I

3. 二叉树还原为树

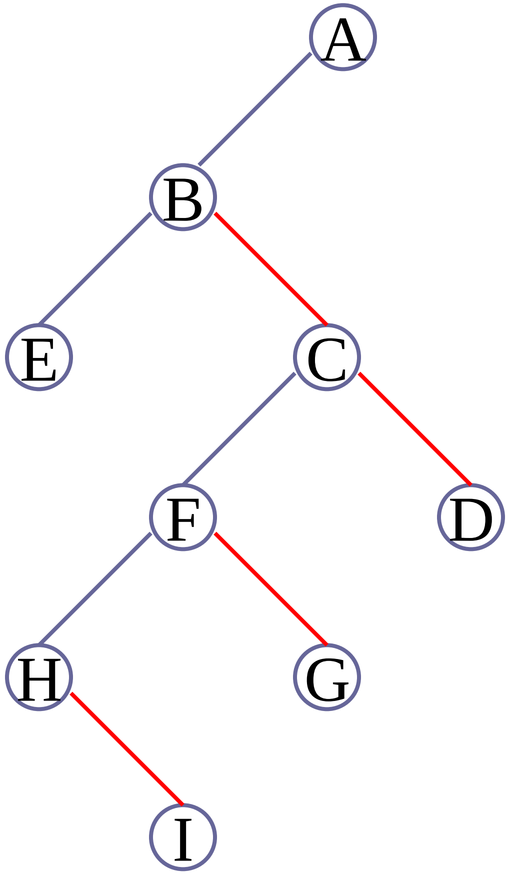
☞步骤：

(1) 加线： 如果结点 p 是某结点的左孩子，则将 p 结点的右孩子、右孩子的右孩子、……，沿着右分支的所有右孩子，都分别与 p 的双亲结点用线连结；

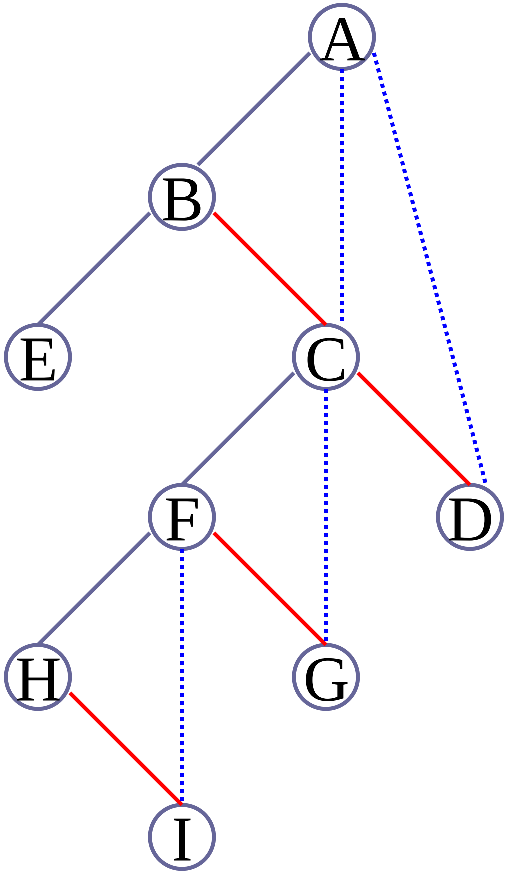
(2) 抹线： 抹掉二叉树中所有结点与其右孩子的连线 -- 转为树；

(3) 调整： 调整结点位置，使之层次分明。

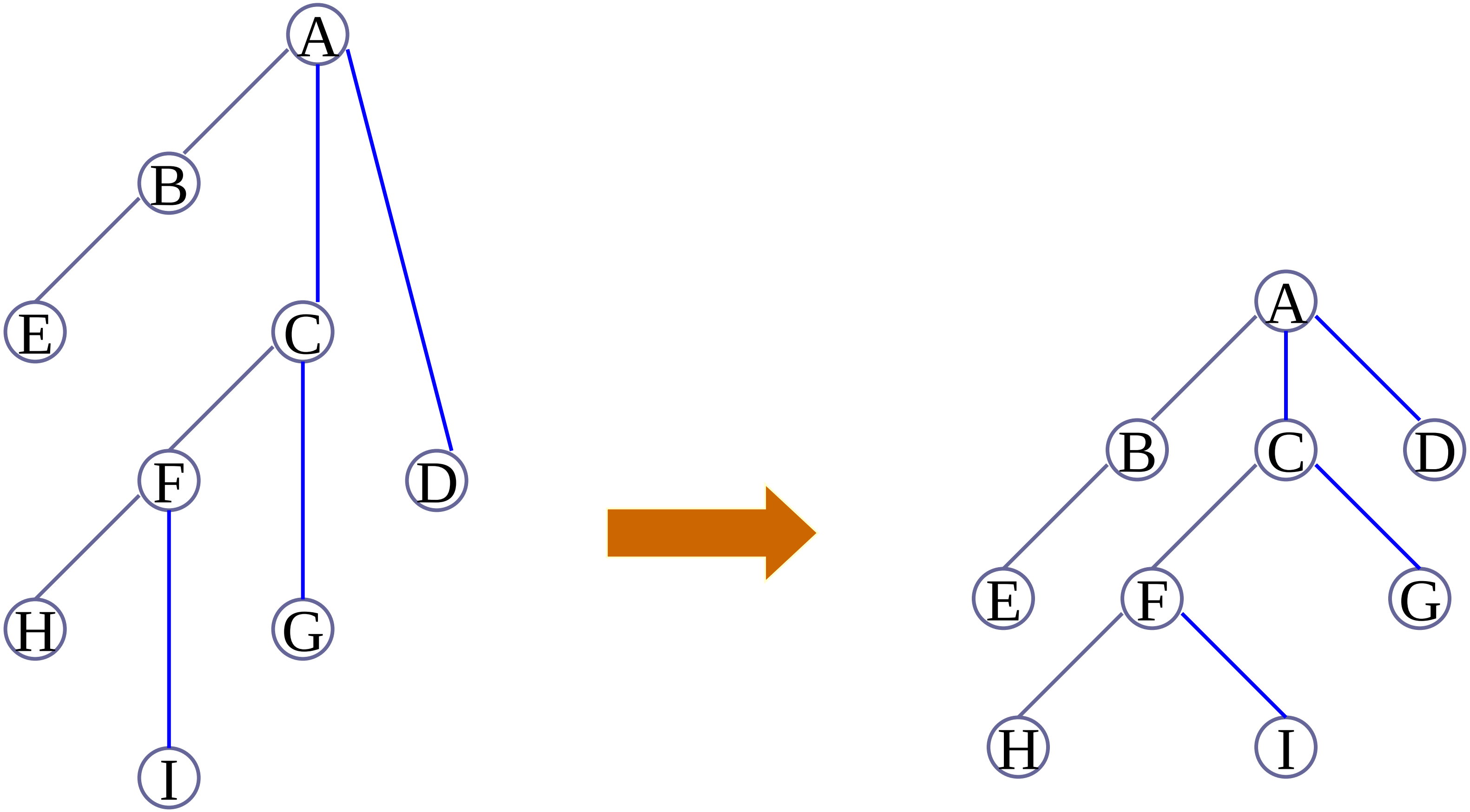
- 没有右子树的二叉树，按这种方法产生的树是唯一的。



(a) 二叉树



(b) 加线后的二叉树



(c) 抹线后还原成为树

(d) 调整后的树

4. 二叉树还原为森林

☞步骤：

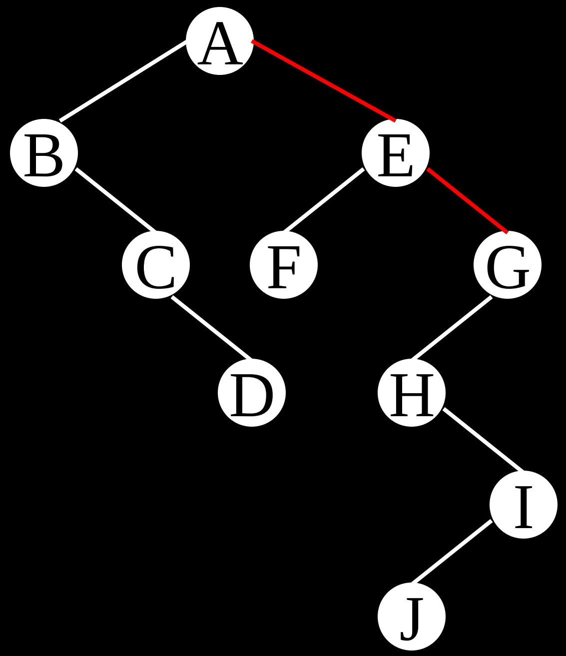
(1)抹线：抹掉根结点与其右子树的连线；

对分离出来的右子树，重复上面操作，直到分离出所有只有左子树的二叉树。

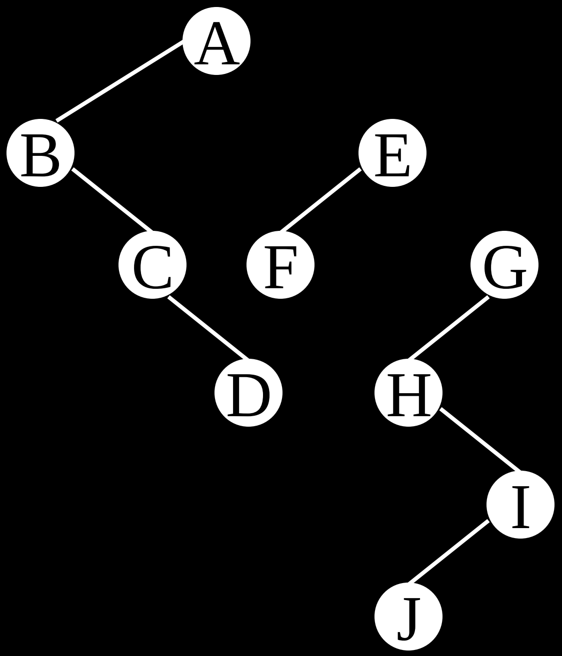
(2)转换：将每棵二叉树分别还原为树；

(3)调整：调整每棵树的结点位置，使之层次分明。

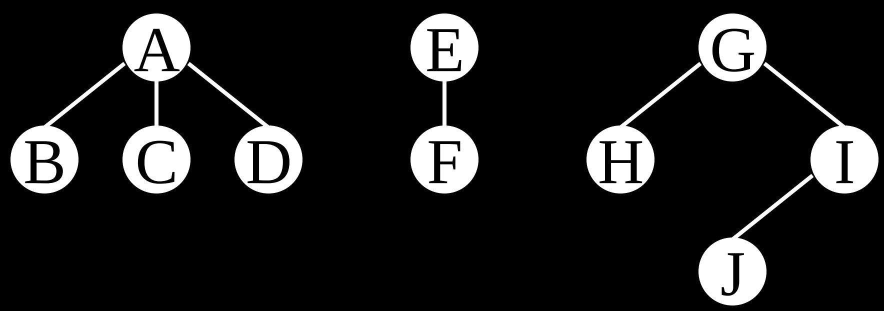
- 这样生成的森林是唯一的。



(a) 二叉树

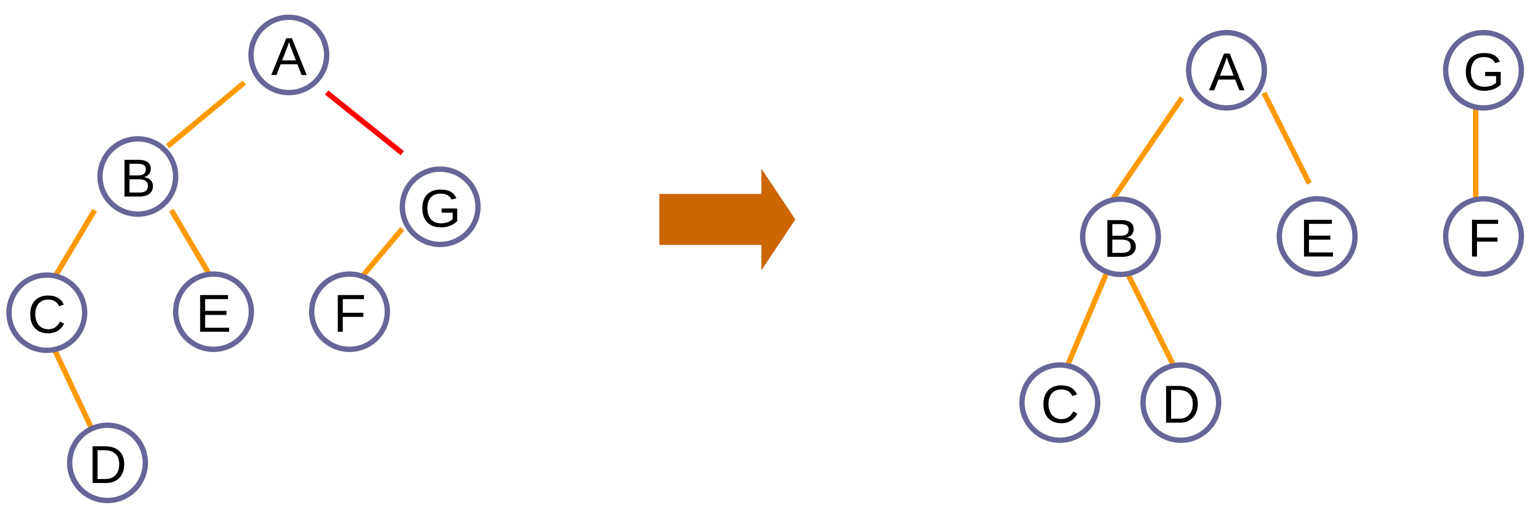


(b) 抹线后

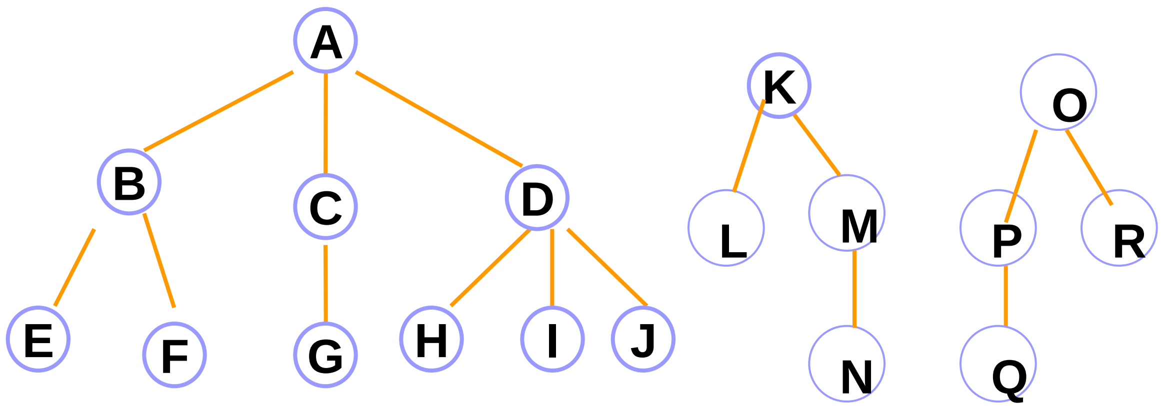


(c)、(d) 转换、调整后的森林

- 二叉树转换为树/森林的一般描述
  - 二叉树 BT => 树/森林 F=（T1，T2，…，Tm）
  - 如果BT不空，则：
    (1) BT的根 => T1的根
    (2) BT的左子树 => T1的子树/森林
    (3) BT的右子树 =>（T2，…，Tm）



■ 练习：将下面的森林转换为对应的二叉树。



【思考问题】

(1) 树（森林）中的叶子结点，在对应的二叉树中有何特征？

☞答：有可能变为分支结点。

(2) 两者分支数是否一定相同？

☞答：分支数相同。

单选题 1分

设置

某二叉树的先序序列和中序序列相同，则该二叉树一定是（　　）的二叉树。

A 任一分支结点只有左子树

B 任一分支结点只有右子树

C 二叉树高度等于结点数

D 没有2度结点

提交

5.5.3 树（森林）的遍历

- ☞ 树和森林遍历方法：先序、后序、层次遍历；
- ☞ 为什么没有中序遍历呢？

1. 树和森林的先序遍历

（1）树的先序遍历

① 先访问根结点；

② 自左至右，依次先序遍历每棵子树。

（2）森林的先序遍历

- ☞ 依次先序遍历森林中的每一棵树。

- 树的先序遍历与转换后的二叉树的先序遍历次序一致。

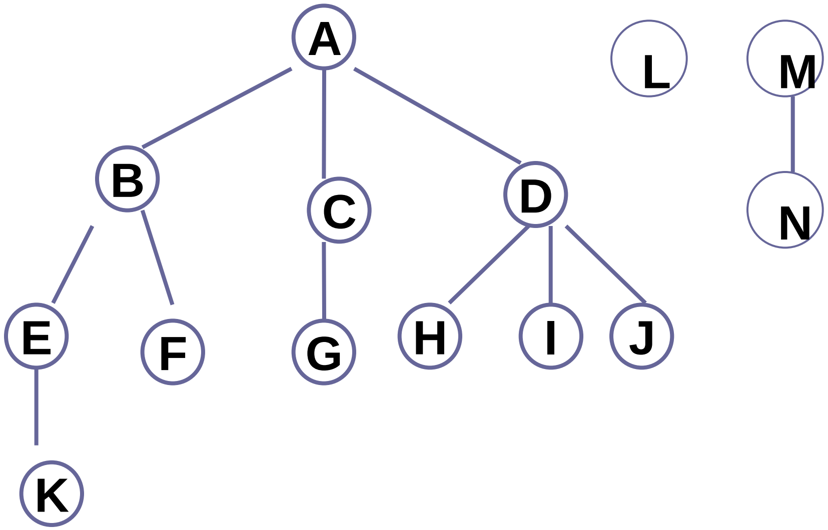
- 树（森林）先序遍历的一般描述
  - ☞先序遍历 树（森林）F = （T1，T2，…，Tm），如果F不空，则：

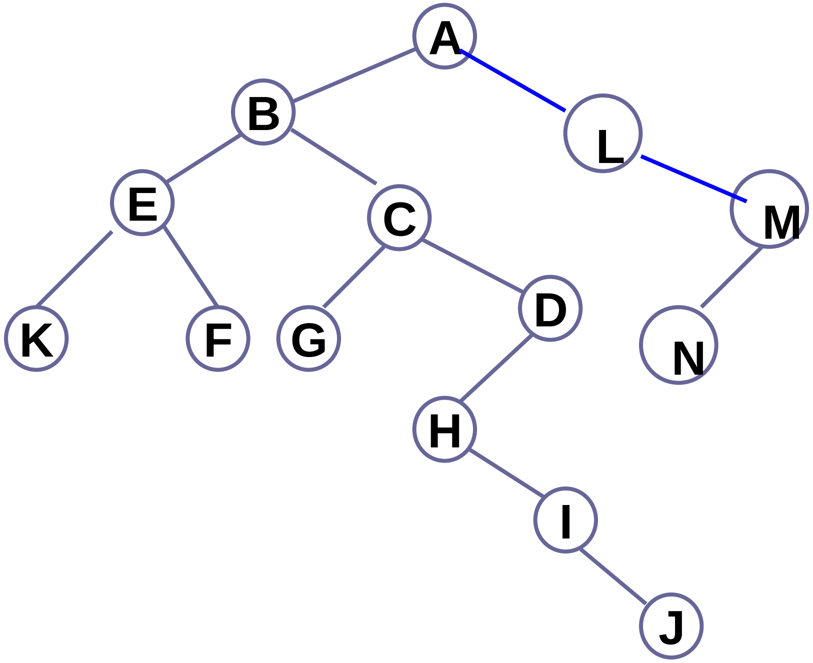
(1) 访问T1的根；

(2) 先序遍历T1的子树；

(3) 先序遍历（T2，T3，…，Tm）；

■ 【例】先序遍历下图的森林





森林的先序序列：

ABEKFCGDHIJLMN

对应的二叉树的先序序列，也是

ABEKFCGDHIJLMN

- 树/森林孩子兄弟链表存储先序遍历算法描述

```
void preOrder( csNode * T )
{
  if ( T != NULL )
  {
      visit ( T );
      preOrder( T -> firstChild );
      preOrder( T -> nextSibling );
  }
}
```

2. 树和森林的后序遍历

（1）树的后续遍历

① 先自左至右依次后序遍历根结点的每棵子树；

② 再访问根结点。

（2）森林的后序遍历

☞ 依次后序遍历森林中的每一棵树。

- 树的后序遍历与转换后的二叉树的中序遍历次序一致。

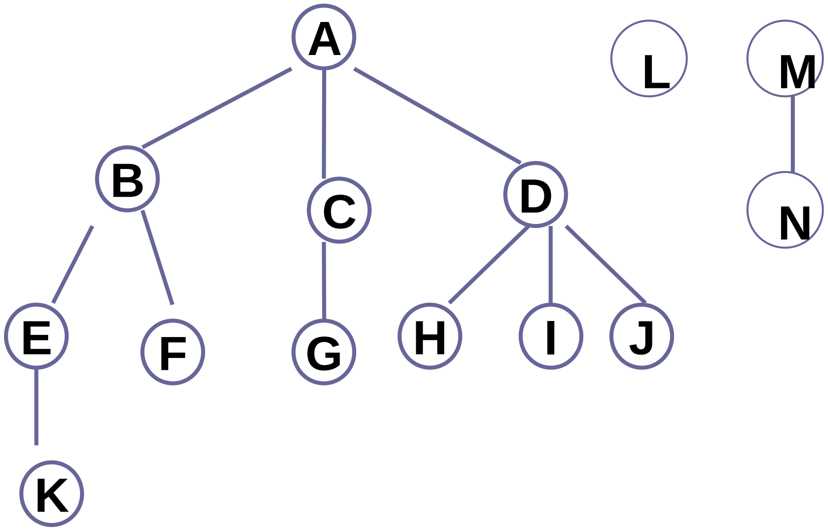
- **树/森林后序遍历的一般描述**
  - ☞后序遍历树/森林 **F = （T1，T2，…，Tm）**，如果**F**不空，则：

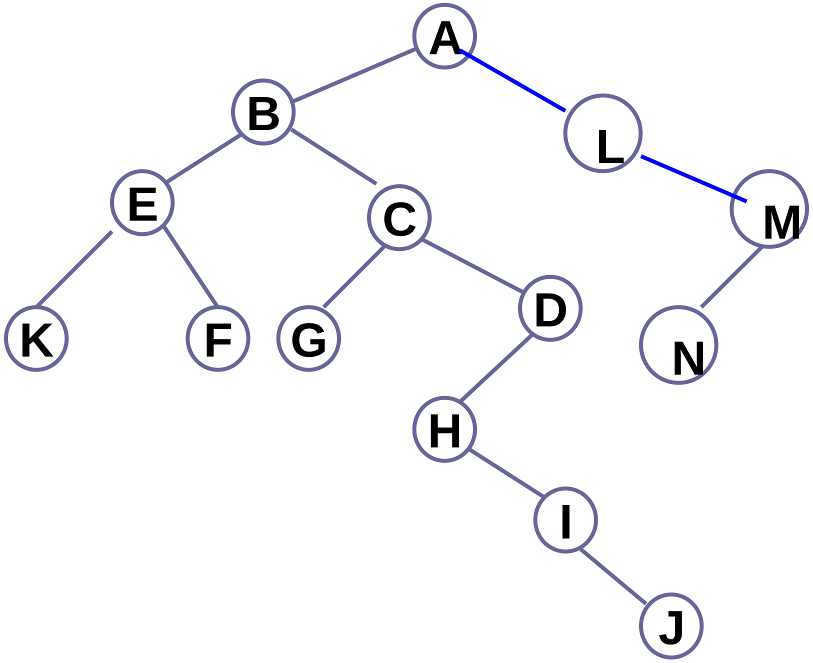
(1) 后序遍历T1的子树；

(2) 访问T1的根；

(3) 后序遍历（T2，T3，…，Tm）；

- 【例】后序遍历下图的森林





森林的后序序列：

KEFBGCHIJDALNM

对应的二叉树的中序序列，也是

KEFBGCHIJDALNM

- 树/森林孩子兄弟链表存储后序遍历算法描述

```
void postOrder( csNode * T )
{
    if ( T != NULL )
    {
        postOrder( T -> firstChild );
        visit ( T );
        postOrder( T -> nextSibling );
    }
}
```

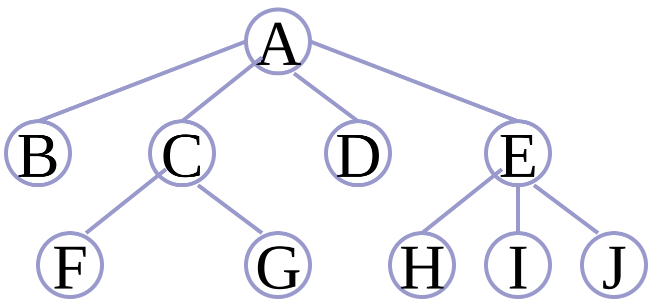
3. 树的层次遍历

（1）树的层次遍历

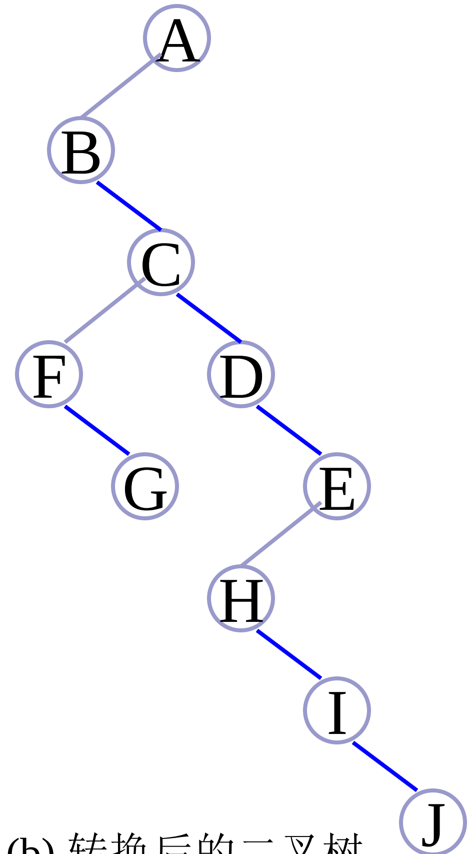
☞自顶向下，每层自左至右，逐个结点访问。

（2）森林的层次遍历

☞依次层次遍历森林中的每一棵树。



(a) 树



(b) 转换后的二叉树

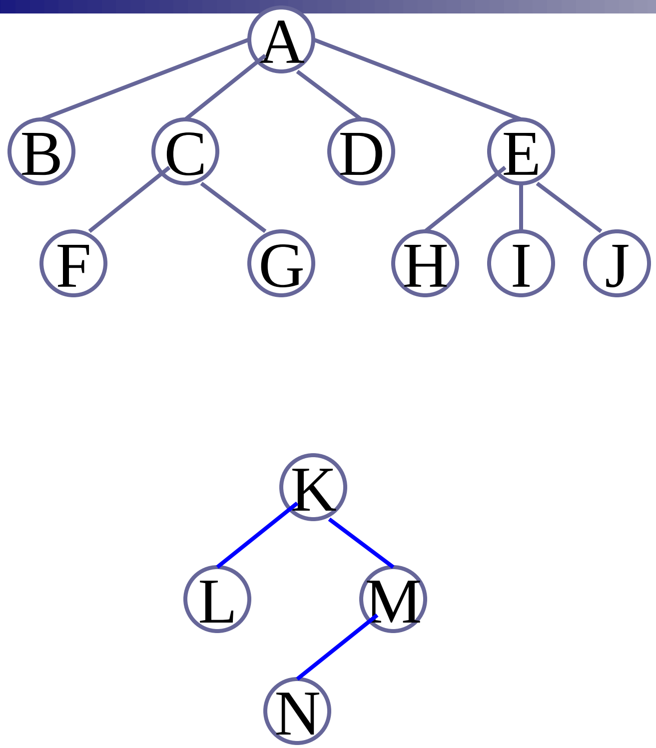
树的先序遍历：ABCFGDEHIJ

树的后序遍历：BFGCDHIJEA

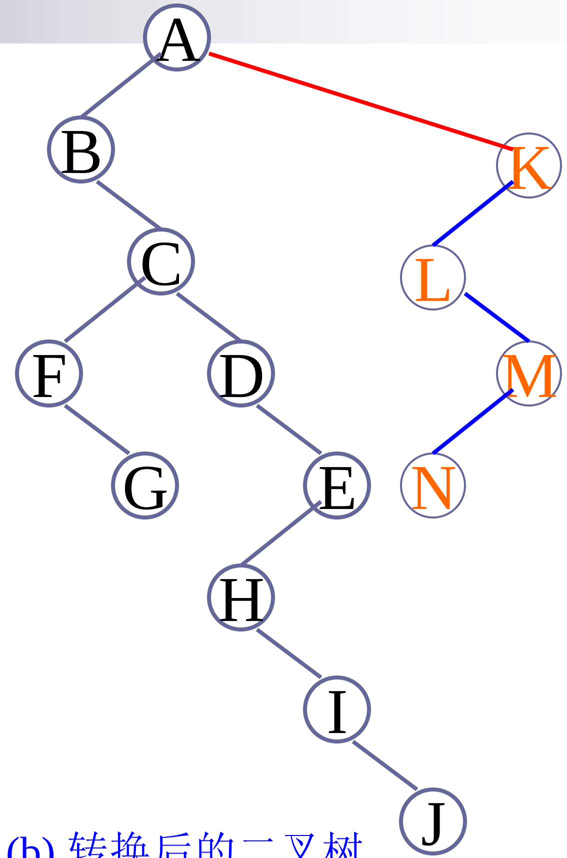
树的层次遍历：ABCDEFGHIJ

二叉树的先序遍历：ABCFGDEHIJ

二叉树的中序遍历：BFGCDHIJEA



(a) 树



(b) 转换后的二叉树

森林的先序遍历：ABCFGDEHIJKLMN 二叉树的先序遍历： ABCFGDEHIJKLMN

森林的后序遍历：BFGCDHIJEALNMK 二叉树的中序遍历： BFGCDHIJEALNMK

森林的层次遍历：ABCDEFGHIJKLMN

【思考问题】

① 设计算法求树/森林中的叶子结点数。

【解】设int leaf(T) —— 返回以T为第一棵树的森林中的叶子数

☞ 分析：

（1）若T为空，　return 0

（2）若T为叶子，return 1+leaf(T->nextSibling)

（3）否则，　return leaf(T->firstChild) +leaf(T-> nextSibling)

② 设计算法求树/森林的高度。

③ 设计算法求树/森林中所有的父子对。

④ ...

【思考问题】

森林采用双亲表示、孩子链表表示，遍历如何实现?

【布置作业】

（p159）--（9）学号：3、6、9

- 5.22
- 5.23
- 5.24
- 5.27
- 5.29

单选题 1分

设置

一棵满二叉树，叶子结点数为n，则此满二叉树结点总数为（ ）。

A 2n-1

B 2n

C 2n+1

D n^2-1

提交

- 5.6 哈夫曼树（Huffman Tree）

本章小结

- 二叉树、树、森林、哈夫曼树基本概念
- 二叉树的5个性质
- 二叉树、树、森林的各种存储结构
- 二叉树遍历
  - 遍历是其它运算的基础，要熟练掌握
  - 进一步理解递归
  - 层次遍历与其他遍历策略不同

- 线索二叉树
  - ☞ 实质是建立结点之间的前驱与后继的关系
  - ☞ 线索化
  - ☞ 找前驱、后继算法
  - ☞ 遍历实现
- 树、森林与二叉树的转换
- 树、森林各种算法的实现
- 哈夫曼树的特性、建立、哈夫曼编码

Thank you！