

数据结构

李春葆

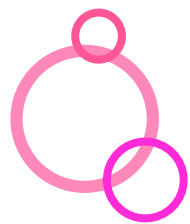
清华大学

树



目录 | CONTENTS

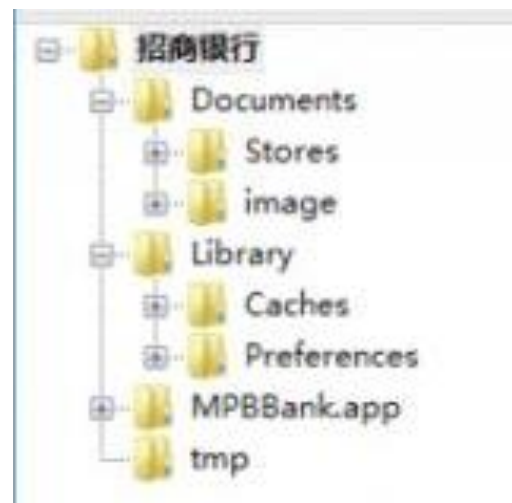
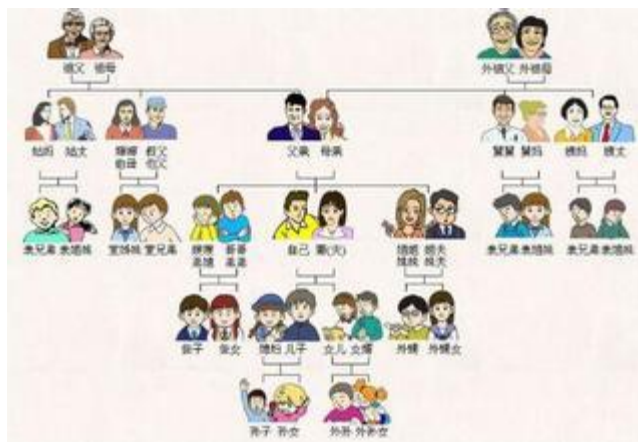
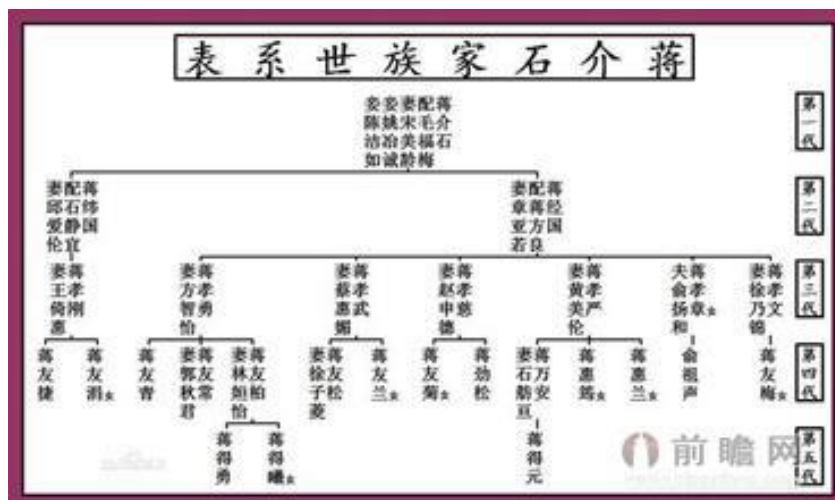
- 01 | 树的基本概念
- 02 | 二叉树概念和性质
- 03 | 二叉树存储结构
- 04 | 二叉树的遍历
- 05 | 二叉树的基本运算及其实现
- 06 | 二叉树的构造
- 07 | 线索二叉树
- 08 | 哈夫曼树



01 树的定义

shudedingyi

-
- 线性结构：一对一
 - 一对多：树



]



基于曲线**二叉树**的微弱边缘检测方法

[在线阅读](#) [下载全文](#)

《计算机工程与设计》2018年第8期2610-2615,共6页 [王会](#) [余阳](#)

四川省教育厅基金项目 (14ZA0366)



基于**二叉树**结构的城轨沙盘联锁系统设计

[在线阅读](#) [下载全文](#)

《铁路计算机应用》2018年第4期9-14,共6页 [蒋文燕](#)

二叉树在计算机联锁中有着较多的应用,根据**二叉树**的结构,将城市轨道交通线路信号点抽象成**二叉树**节点,构建线路的**二叉树**模型,然后在**二叉树**前序遍历的基础上,经过正向搜索算法和逆向搜索算法,完成进路



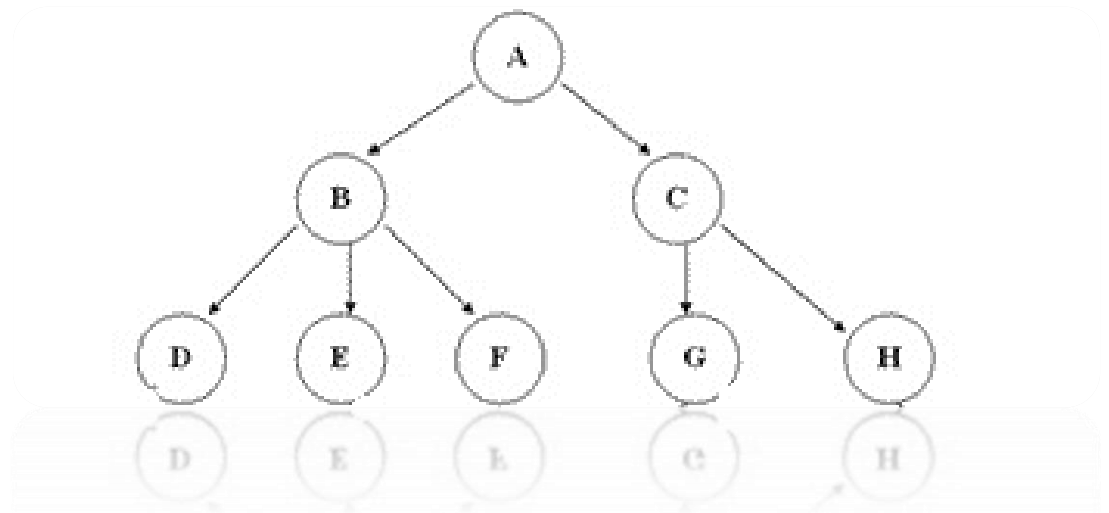
基于**二叉树**存储结构的LZW改进算法

[在线阅读](#) [下载全文](#)

《太原学院学报:自然科学版》2018年第1期29-32,共4页 [崔方送](#)

2017年安徽省高校自然科学研究项目-重点项目 (KJ2017A915); 2016年安徽省高等教育创新发展行动计划 (RW-11-s34)

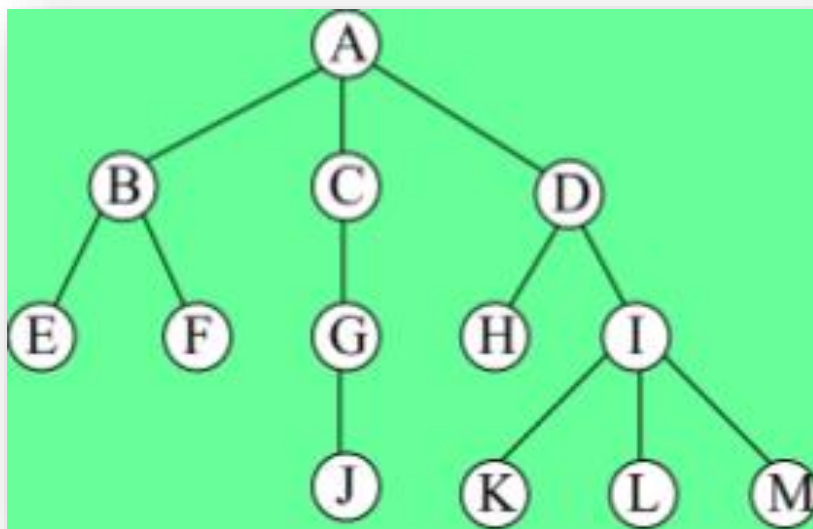
LZW算法是一种高效的自适应数据压缩算法,但在编码过程中,存储字典中词条会重复存放已存字符,从而造成存储空间浪费,文章对此提出了一种改进算法,将重复存放的词条合并为一个词条,从而减少了存储空间,提高了压缩效率。



1.树的定义

■ 形式化定义：

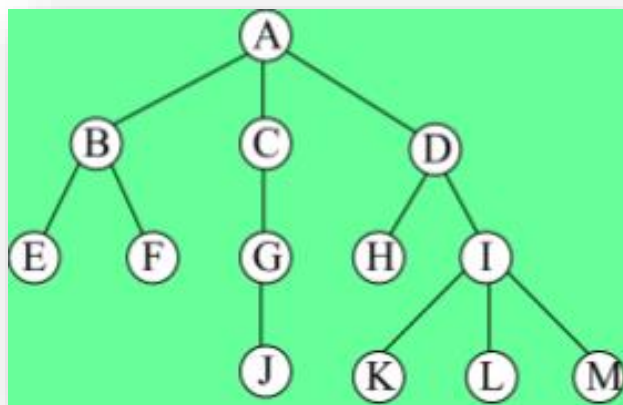
- 树： $T = \{D, R\}$ $D = \{A, B, C, D, E, F, G, H, I, J, K, L, M\}$
- $R = \{r\}$
- $r = \{\langle A, B \rangle, \langle A, C \rangle, \langle A, D \rangle, \langle B, E \rangle, \langle B, F \rangle, \langle C, G \rangle, \langle D, H \rangle, \langle D, I \rangle, \langle G, J \rangle, \langle I, K \rangle, \langle I, L \rangle, \langle I, M \rangle\}$



1.树的定义

■ 形式化定义：

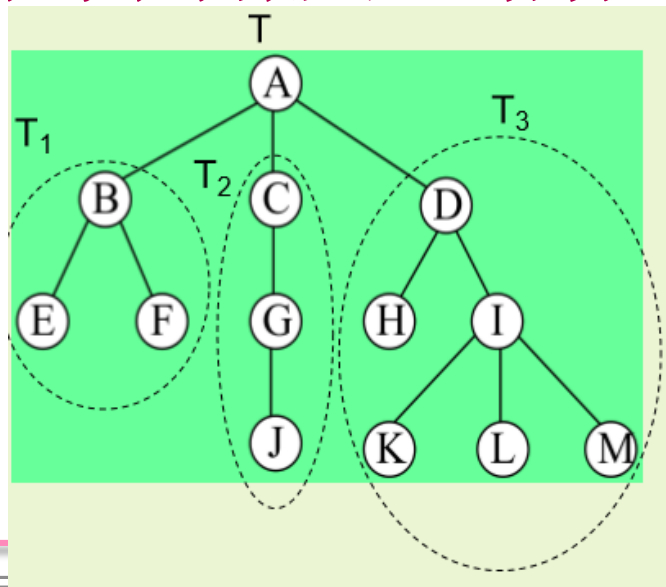
- K 是包含 n 个结点的有穷集合($n>0$), 关系 R 满足以下条件:
 - (1) 有且仅有一个结点 $k_0 \in K$,它对于关系 R 来说没有前驱结点,结点 k_0 称作树的根。
 - (2) 除结点 k_0 外, K 中的每个结点对于关系 R 来说都有且仅有一个前驱结点。
 - (3) K 中每个结点对于关系 R 来说可以有多个后继结点。



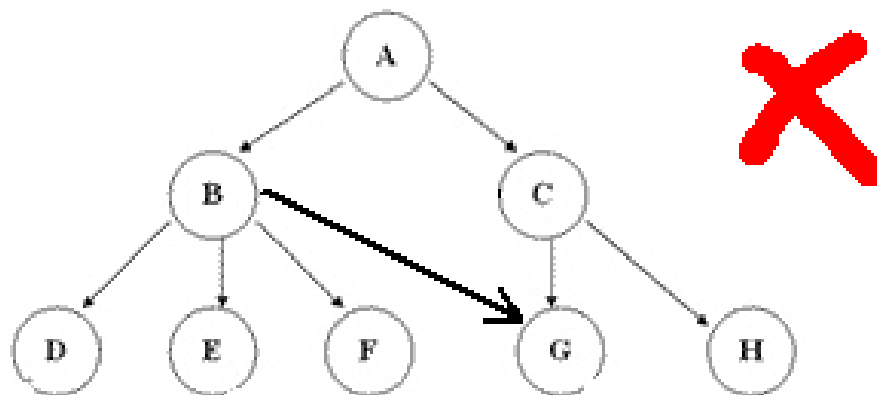
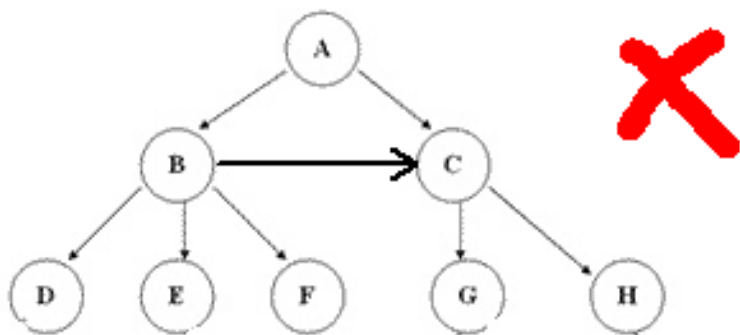
1.树的定义

■ 递归定义：

- 树(Tree)是 $n(n \geq 0)$ 个结点的有限集 T ， T 为空时称为空树，否则它满足如下两个条件：
 - (1) 有且仅有一个特定的称为根(Root)的结点；
 - (2) 其余的结点可分为 $m(m \geq 0)$ 个互不相交的子集 $T_1, T_2, T_3 \dots T_m$ ，其中每个子集又是一棵树，并称其为子树(Subtree)。

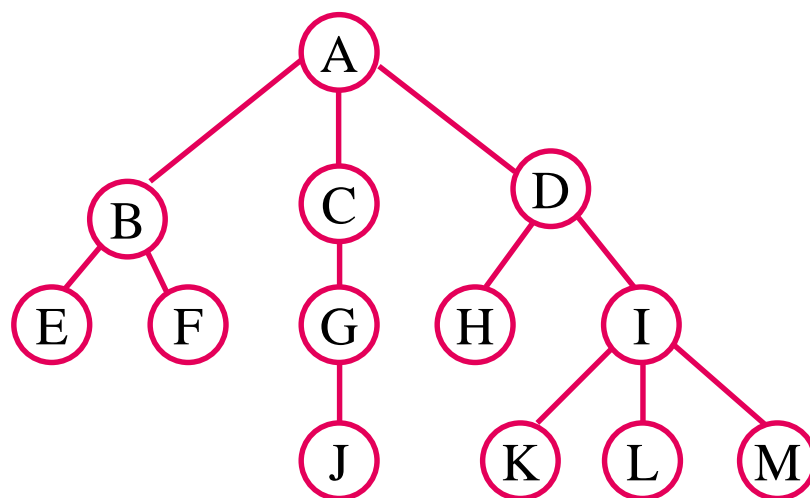


1.树的定义



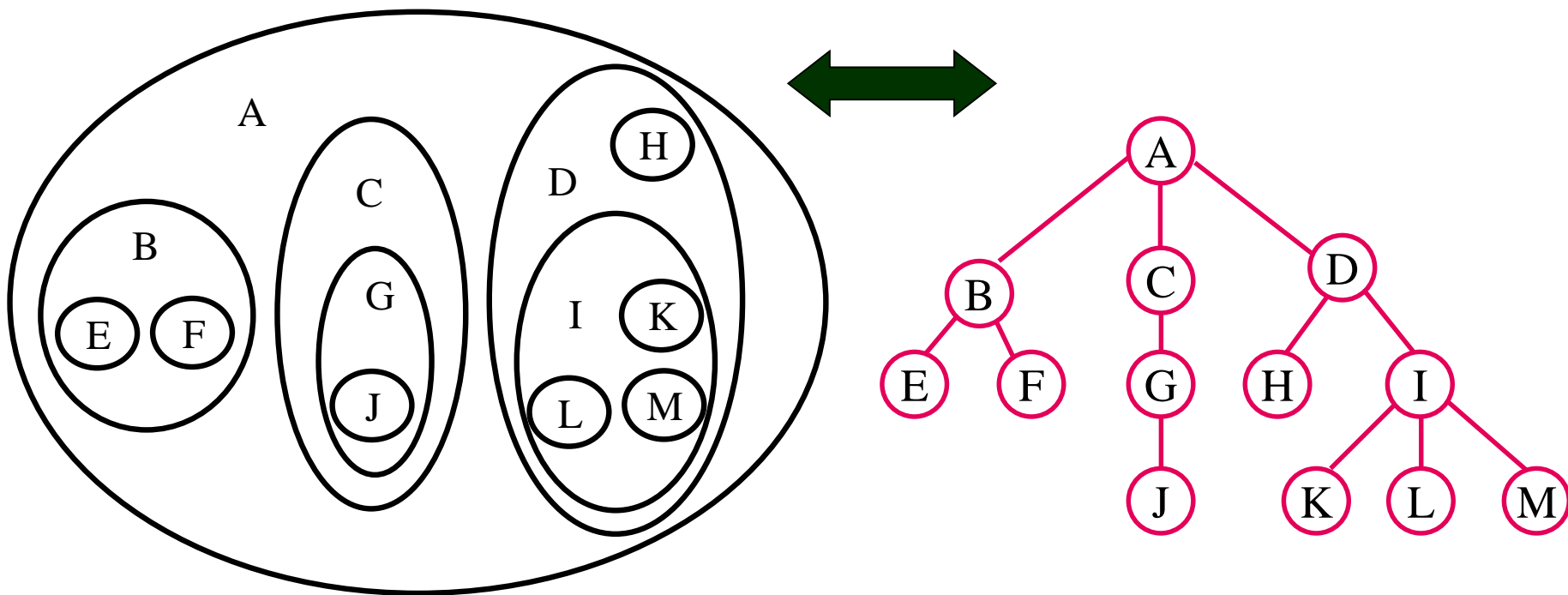
2. 树的表示

- (1) 树形表示法。这是树的最基本的表示, 使用一棵倒置的树表示树结构, 非常直观和形象。下图就是采用这种表示法。



2. 树的表示

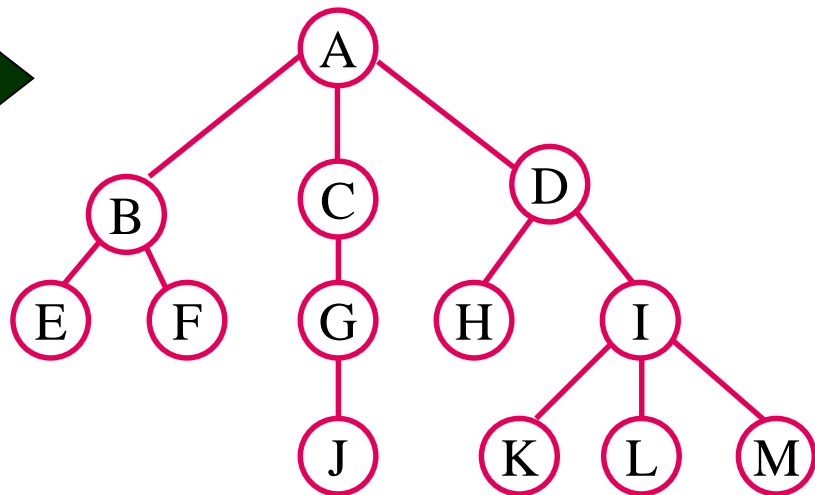
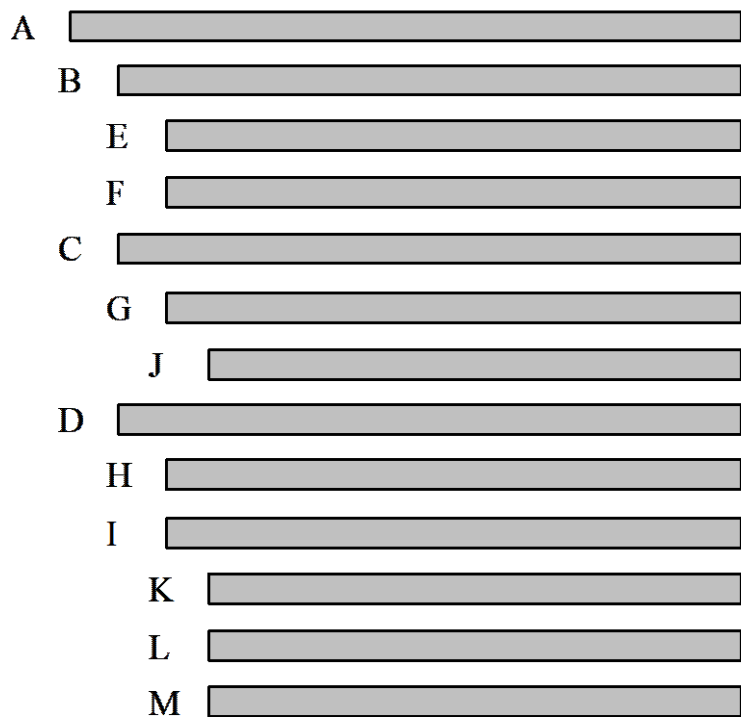
- (2) 文氏图表示法。使用集合以及集合的包含关系描述树结构。下图就是树的文氏图表示法。



文氏图表示法

2. 树的表示

- (3) 凹入表示法。使用线段的伸缩描述树结构。下图是树的凹入表示法。



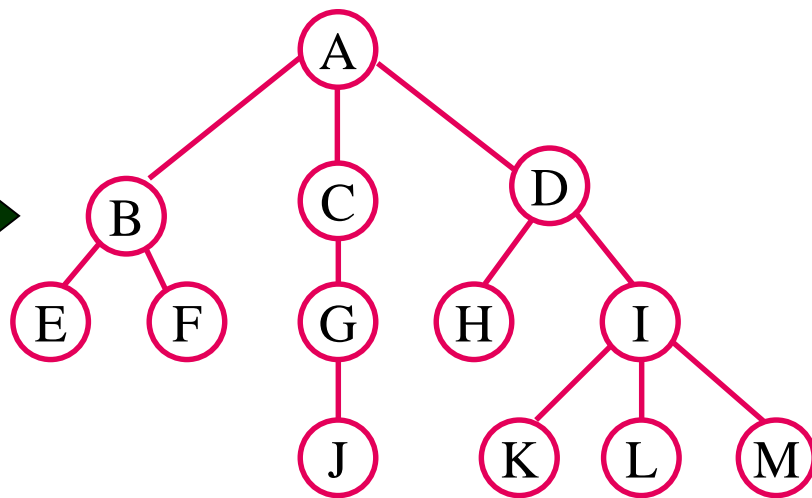
凹入表示法

2. 树的表示

- (4) 括号表示法。将树的根结点写在括号的左边, 除根结点之外的其余结点写在括号中并用逗号间隔来描述树结构。下图是树的括号表示法。

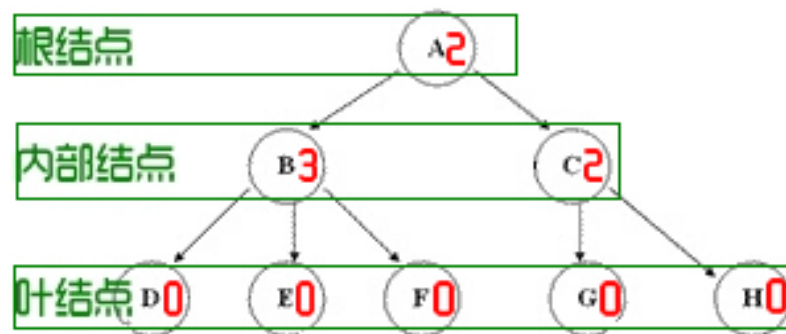
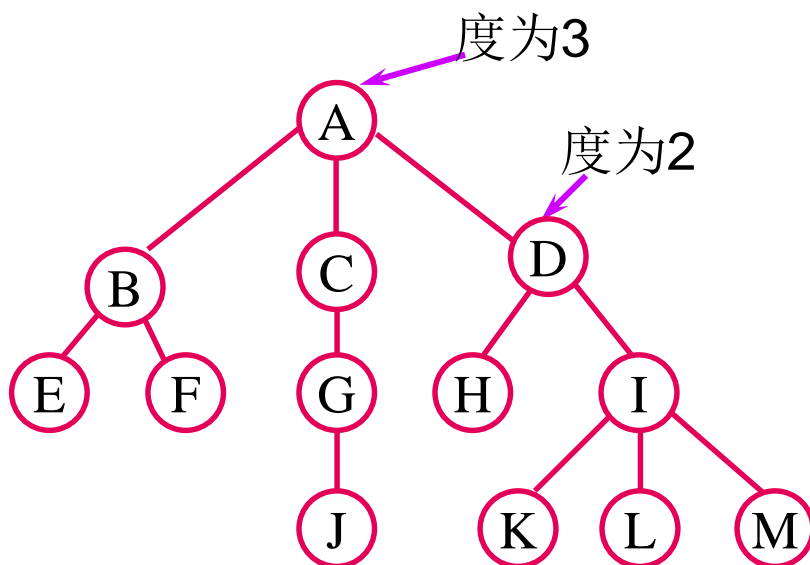
A(B(E,F),C(G(J)),D(H,I(K,L,M)))

括号表示法



3. 树的基本术语

- 1. 结点的度与树的度：
 - 某个结点的子树的个数称为该结点的度。
 - 各结点的度的最大值称为树的度
 - 度为 m 的树称为 m 次树。



3.树的基本术语

■ 2. 分支结点与叶结点：

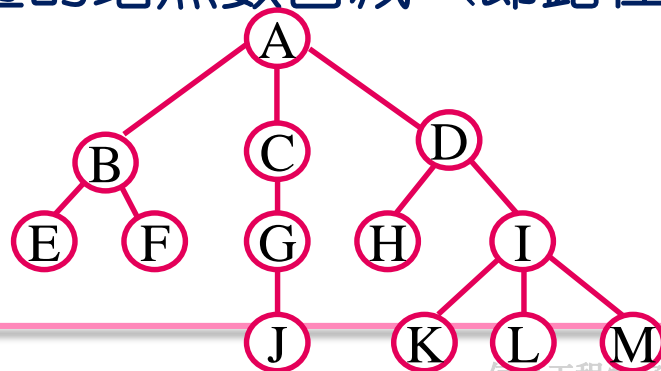
- 度不为零的结点称为非终端结点, 又叫分支结点。
- 度为零的结点称为终端结点或叶结点。
- 在分支结点中, 每个结点的分支数就是结点的度。

3. 树的基本术语

■ 3. 路径与路径长度：

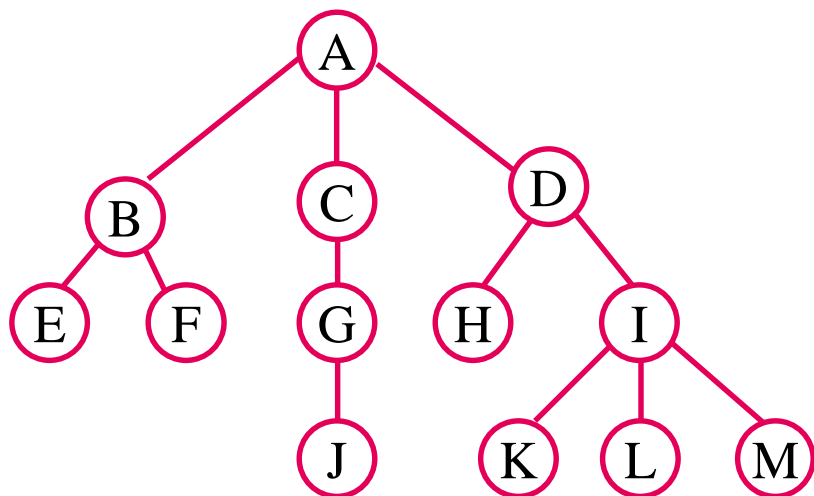
- 对于任意两个结点 k_i 和 k_j , 若树中存在一个结点序列 $k_i, k_{i1}, k_{i2}, \dots, k_{in}, k_j$, 使得序列中除 k_i 外的任一结点都是其在序列中的前一个结点的后继, 则称该结点序列为由 k_i 到 k_j 的一条**路径**
- **路径的表示**：用路径所通过的结点序列 $(k_i, k_{i1}, k_{i2}, \dots, k_j)$ 表示这条路径。
- **路径的长度**：等于路径所通过的结点数目减1 (即路径上分支数目)。

A到K的路径为A,D,I,K,
其长度为3



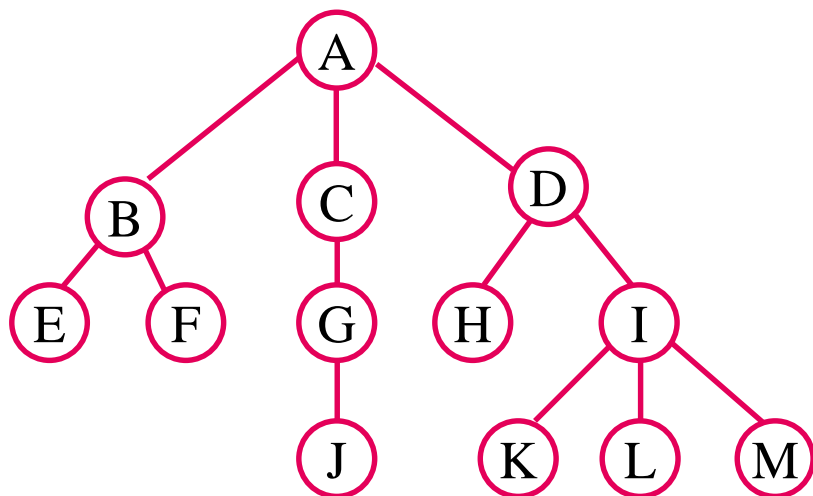
3. 树的基本术语

- 4. 孩子结点、双亲结点和兄弟结点：
 - 每个结点的后继, 被称作该结点的孩子结点(或子女结点)。
 - 相应地, 该结点被称作孩子结点的双亲结点(或父母结点)。



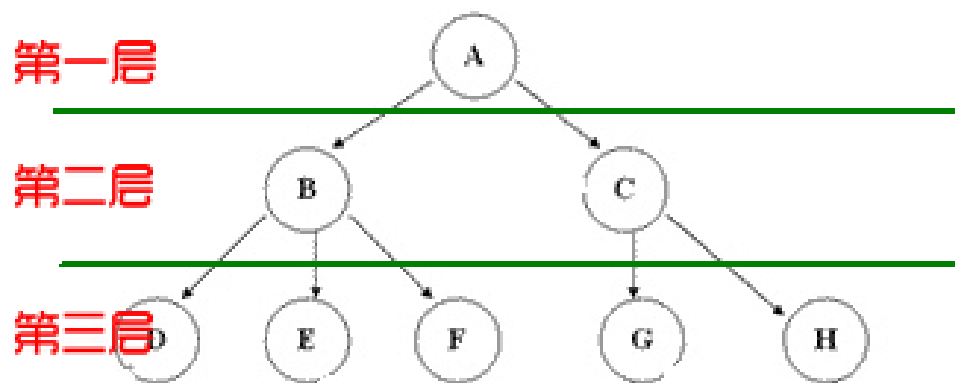
3. 树的基本术语

- 4. 孩子结点、双亲结点和兄弟结点：
 - 具有同一双亲的孩子结点互为兄弟结点。
 - 推广这些关系，每个结点的所有子树中的结点称为该结点的子孙结点，从树根结点到达该结点的路径上经过的所有结点被称作该结点的祖先结点。



3. 树的基本术语

- 5. 结点的层次和树的高度：
 - 结点的层次从树根开始定义, 根结点为第1层, 它的孩子结点为第2层, 一个结点所在的层次为其双亲结点所在的层次加1。
 - 树中结点的最大层次称为树的高度(或树的深度)。



树的基本术语

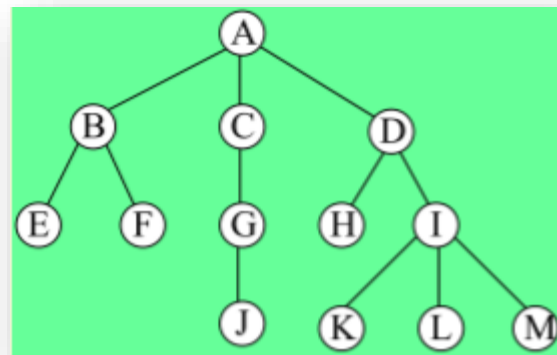
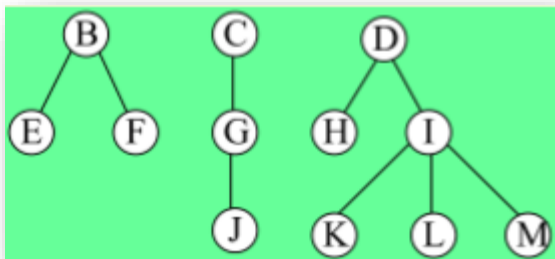
■ 6. 有序树和无序树：

- 若树中各结点的子树是按照一定的次序从左向右安排的，且相对次序是不能随意变换的，则称为有序树，否则称为无序树。

树的基本术语

■ 7. 森林：

- $n (n > 0)$ 个互不相交的树的集合称为森林。森林的概念与树的概念十分相近, 因为只要把树的根结点删去就成了森林。反之, 只要给 n 棵独立的树加上一个结点, 并把这 n 棵树作为该结点的子树, 则森林就变成了树。

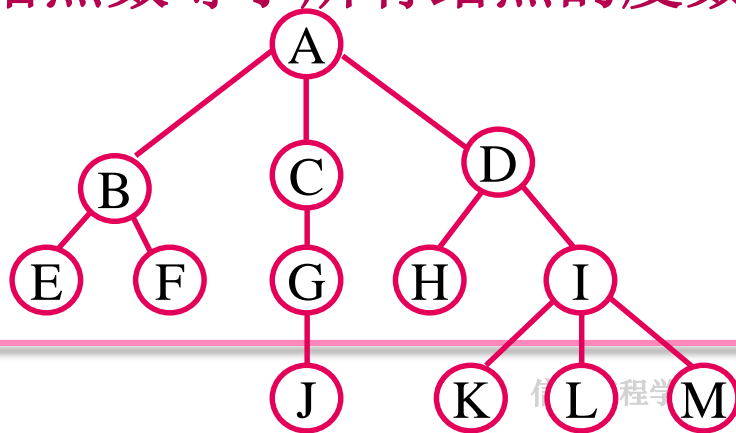


4. 树的性质

- 性质1 树中的结点数等于所有结点的度数加1。

- 证明:

- 根据定义树中,除树根结点外,每个结点有且仅有一个前驱结点。
- 每个结点与指向它的一个分支一一对应
- 推论: 除树根之外的结点数等于所有结点的分支数(度数)
- 结论: 从而可得树中的结点数等于所有结点的度数加1。



- 例：一棵度为4的树T中，若有20个度为4的结点，10个度为3的结点，1个度为2的结点，10个度为1的结点，则树T的叶子结点个数是_____。

B. 82

D. 122

4. 树的性质

- 性质2 度为 m 的树中第 i 层上至多有 m^{i-1} 个结点, 这里应有 $i \geq 1$.
 - 证明(采用数学归纳法)
 - 对于第一层, 因为树中的第一层上只有一个结点, 即整个树的根结点, 而由 $i=1$ 代入 m^{i-1} , 得 $m^{i-1}=m^{1-1}=1$, 也同样得到只有一个结点, 显然结论成立。
 - 假设对于第 $(i-1)$ 层($i \geq 1$)命题成立, 即度为 m 的树中第 $(i-1)$ 层上至多有 m^{i-2} 个结点, 则根据树的度的定义, 度为 m 的树中每个结点至多有 m 个孩子结点, 所以第 i 层上的结点数至多为第 $(i-1)$ 层上结点数的 m 倍, 即至多为 $m^{i-2} \times m = m^{i-1}$ 个, 这与命题相同, 故命题成立。

4. 树的性质

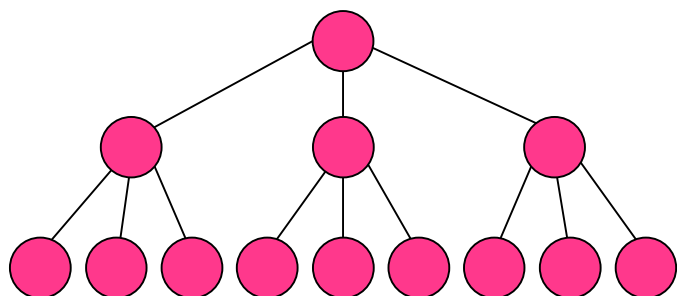
- 性质3 高度为 h 的 m 次树至多有 $\frac{m^h - 1}{m - 1}$ 个结点。

- 证明:

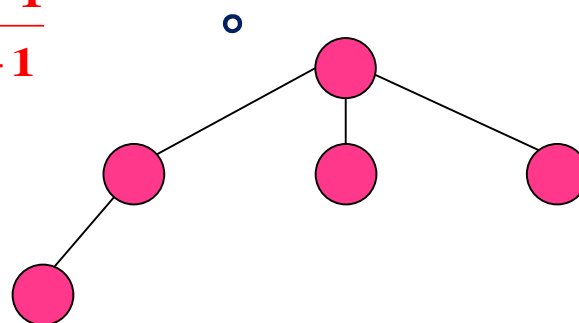
- 由树的性质2可知,第 i 层上最多结点数为 m^{i-1} ($i=1,2,\dots,h$), 显然当高度为 h 的 m 次树(即度为 m 的树)上每一层都达到最多结点数时,整个 m 次树具有最多结点数,因此有:

- 整个树的最多结点数=每一层最多结点数之和

$$= m^0 + m^1 + m^2 + \dots + m^{h-1} = \frac{m^h - 1}{m - 1}$$



$m=3, h=3$, 最多节点情况



$m=3, h=3$, 最少节点情况

4. 树的性质

- 性质4 具有 n 个结点的 m 次树的最小高度为 $\log_m(n(m-1)+1)$
- 证明：设具有 n 个结点的 m 次树的高度为 h , 若在该树中前 $h-1$ 层都是满的, 即每一层的结点数都等于 m^{i-1} 个 ($1 \leq i \leq h-1$), 第 h 层(即最后一层)的结点数可能满, 也可能不满, 则该树具有最小的高度。其高度 h 可计算如下:

4. 树的性质

- 根据树的性质3可得: $\frac{m^{h-1} - 1}{m - 1} < n \leq \frac{m^h - 1}{m - 1}$

乘(m-1)后得: $m^{h-1} < n(m-1) + 1 \leq m^h$

以m为底取对数后得: $h-1 < \log_m (n(m-1) + 1) \leq h$

即 $\log_m (n(m-1) + 1) \leq h < \log_m (n(m-1) + 1) + 1$

因h只能取整数, 所以

$$h = \lceil \log_m (n(m-1) + 1) \rceil$$

结论得证

【例】

- 含n个结点的三次树的最小高度是多少？最大高度是多少？

解：设含n个结点的(为完全三次树时高度最小)的三次树的最小高度为h, 则有：

$$1+3+9+\dots+3^{h-2} < n \leq 1+3+9+\dots+3^{h-1}$$

$$(3^h-1)/2 < n \leq (3^{h+1}-1)/2$$

$$3^{h-1} < 2n+1 \leq 3^h$$

$$\text{即： } h = \lceil \log_3(2n+1) \rceil$$

最大高度为n-2

5. 树的基本运算

- 树的运算主要分为三大类：
 - 第一类, 寻找满足某种特定关系的结点
 - 如寻找当前结点的双亲结点等;
 - 第二类, 插入或删除某个结点
 - 如在树的当前结点上插入一个新结点或删除当前结点的第 i 个孩子结点等;
 - 第三类, 遍历树中每个结点。

树的遍历

- 树的遍历

- 是指按某种方式访问树中的每一个结点且每一个结点只被访问一次。

- 树的遍历运算的算法主要有以下三种方式：

- 先根遍历
 - 后根遍历
 - 层次遍历

树的遍历

■ 1. 先根遍历

- 先根遍历过程为：

- (1) 访问根结点；

- (2) 按照从左到右的次序先根遍历根结点的每一棵子树。

■ 2. 后根遍历

- 后根遍历过程为：

- (1) 按照从左到右的次序后根遍历根结点的每一棵子树；

- (2) 访问根结点。

■ 3. 层次遍历

- 若树不空，则自上而下自左至右访问树中每个结点。

树的遍历

先根遍历的顶点访问次序：

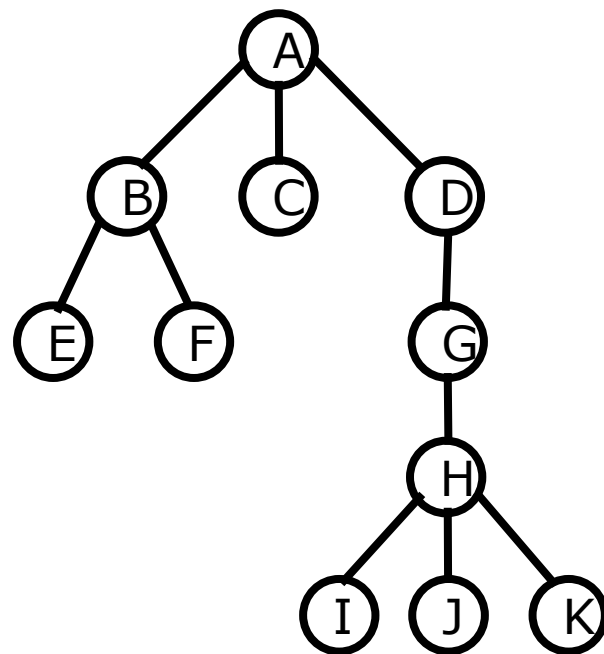
A B E F C D G H I J K

后根遍历的顶点访问次序：

E F B C I J K H G D A

层次遍历的顶点访问次序：

A B C D E F G H I J K



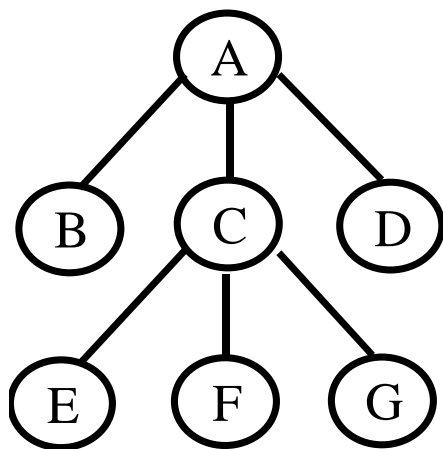
【思考】？

- 存储树，简单的顺序存储结构和链式存储结构可以做到吗？
- 需要考虑到双亲、孩子、兄弟之间的关系

6. 树的存储结构

■ 1. 双亲存储结构

- 这种存储结构是一种顺序存储结构, 用一组连续空间存储树的所有结点, 同时在每个结点中附设一个伪指针指示其双亲结点的位置。



(a)

树的双亲存储结构示意图

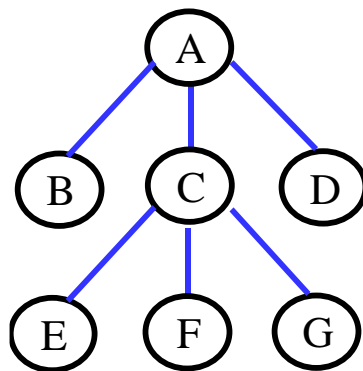
0	A	-1
1	B	0
2	C	0
3	D	0
4	E	2
5	F	2
6	G	2

(b)


■ 双亲存储结构的类型声明如下：

- `typedef struct`
- `{ ElemType data; //结点的值`
- `int parent; //指向双亲的位置`
- `} PTree[MaxSize];`

【思考】：该存储结构的优缺点？



(a)



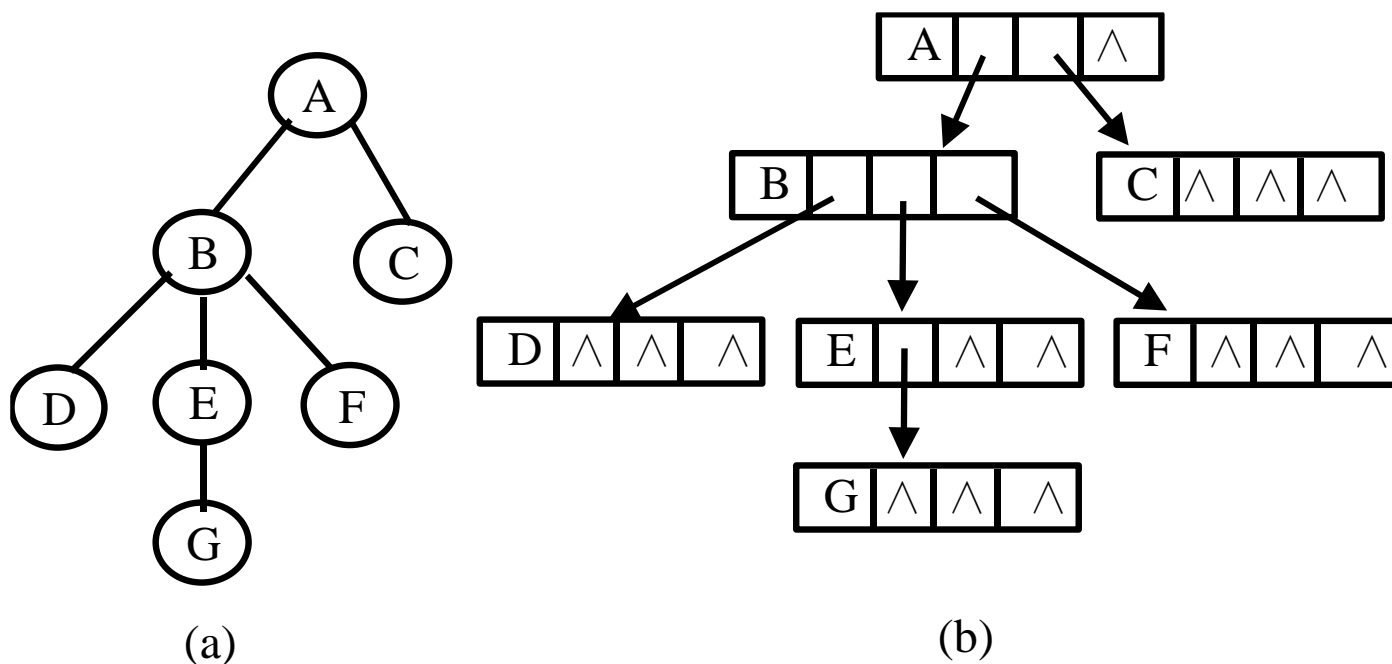
0	A	-1
1	B	0
2	C	0
3	D	0
4	E	2
5	F	2
6	G	2

(b)

6. 树的存储结构

■ 2. 孩子链存储结构

- 孩子链存储结构可按树的度(即树中所有结点度的最大值)设计结点的孩子结点指针域个数。



树的孩子链存储结构示意图

- 孩子链存储结构的节点类型声明如下：

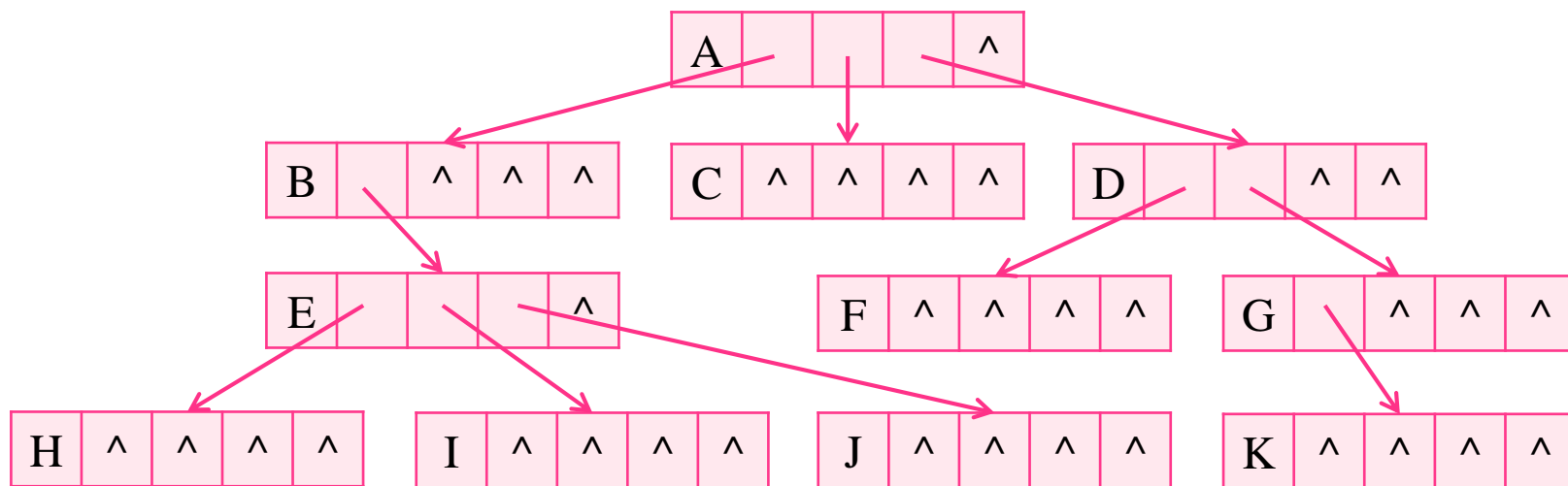
- `typedef struct node`
- `{ ElemType data; //节点的值`
- `struct node *sons[MaxSons]; //指向孩子节点`
- `} TSonNode;`
- 其中，`MaxSons`为最多的孩子节点个数。

【思考】：n个节点的m次树有多少个空指针域？

【思考】：该存储结构的优缺点？

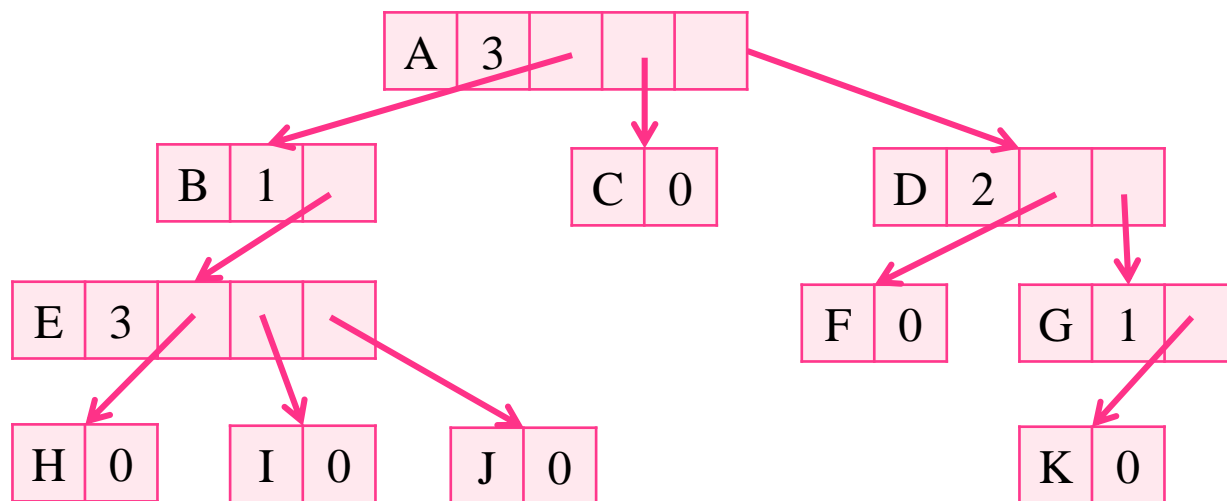
【思考】：树的存储---孩子表示法方案PK

- 方案一：根据树的度，声明足够空间存放子树指针的结点



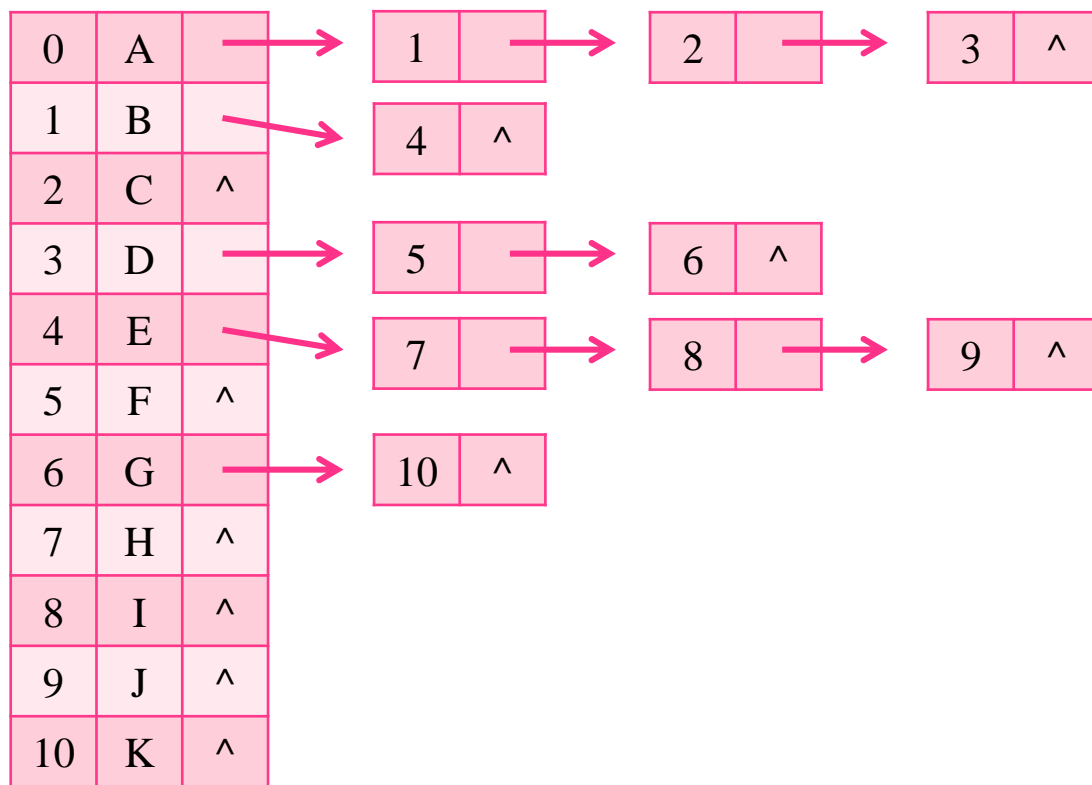
- 缺点？
- 浪费！

■ 方案二：崇尚节俭，节俭光荣

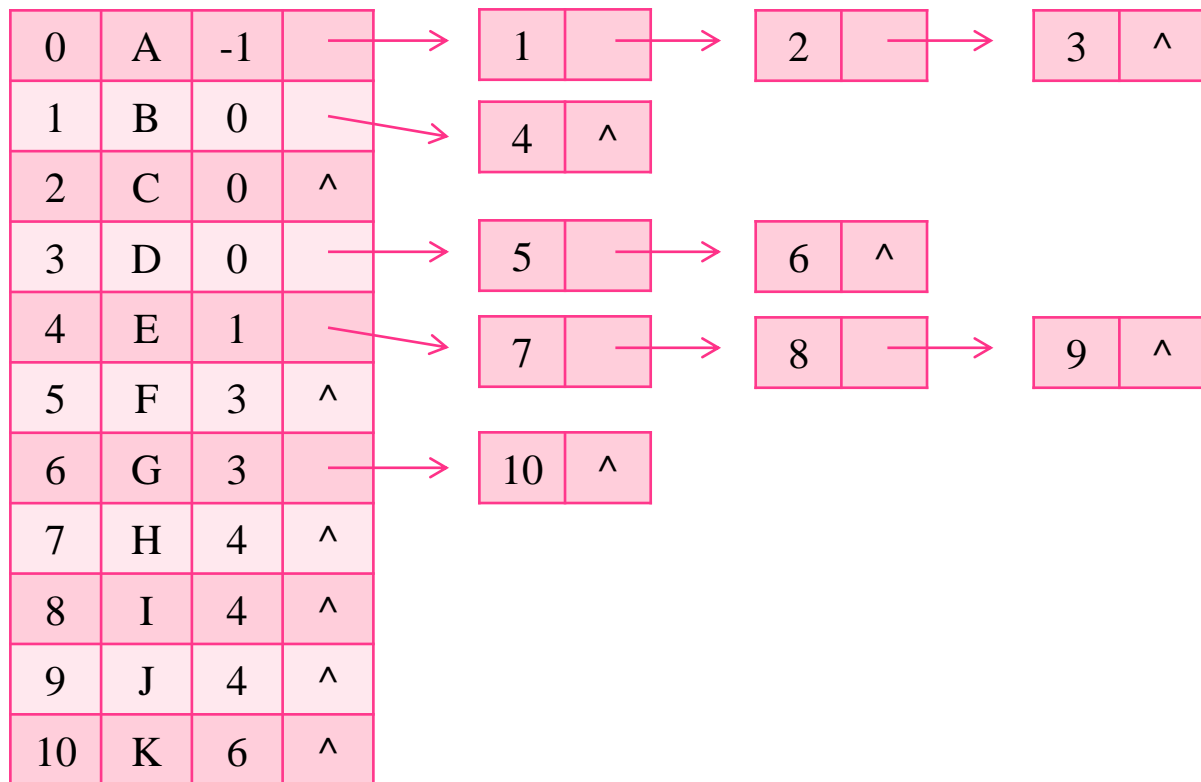


- 缺点？
- 每个结点的度的值不同，初始化和维护起来难度巨大吧

■ 方案三：看看这种设计如何？



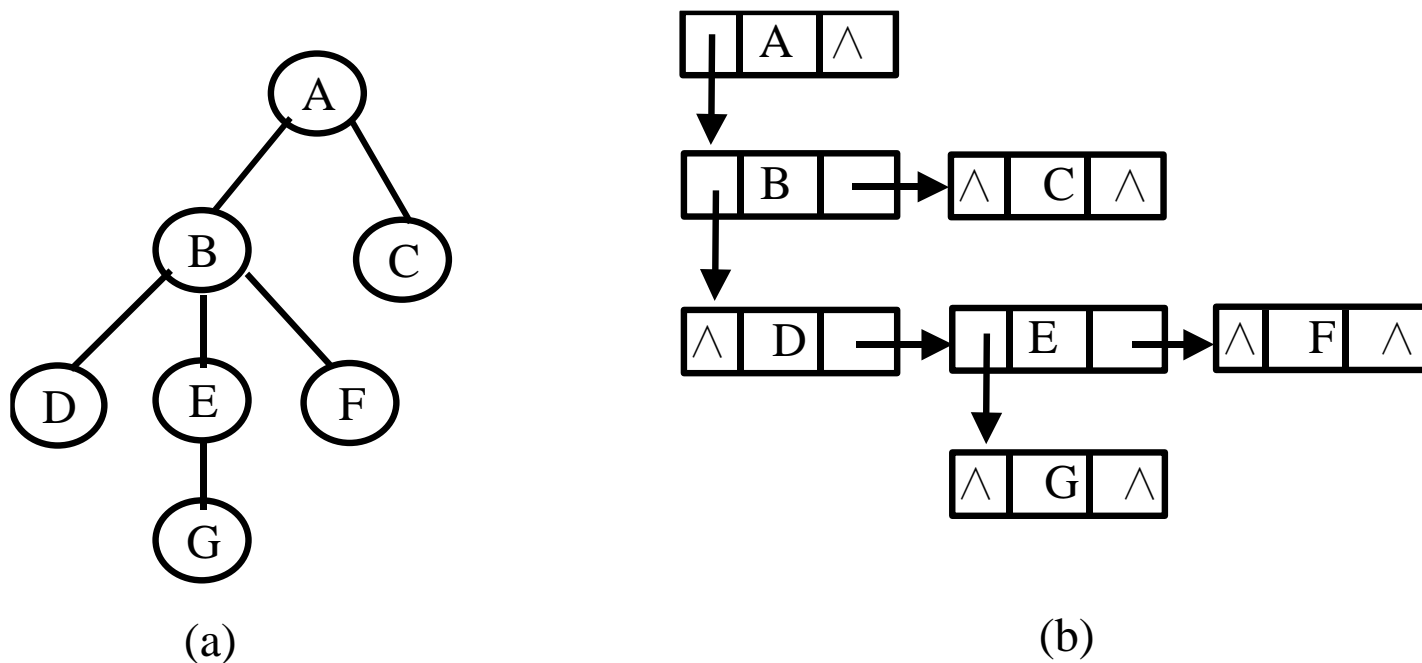
■ 方案四：再来完美一下。能找孩子能找双亲



6. 树的存储结构

■ 3. 孩子兄弟链存储结构

- 每个结点设计三个域：一个数据元素域，一个该结点的第一个孩子结点指针域，一个该结点的下一个兄弟结点指针域。



树的孩子兄弟链存储结构示意图

- 兄弟链存储结构中节点的类型声明如下：

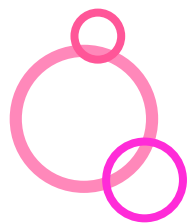
- `typedef struct tnode`
- `{ ElemType data;` `//节点的值`
- `struct tnode *hp;` `//指向兄弟`
- `struct tnode *vp;` `//指向孩子节点`
- `} TSBNODE ;`

- 每个节点固定只有两个指针域。

【思考】：该存储结构的优缺点？

【提示】

- 存储结构的设计是一个非常灵活的过程，只要你愿意，你可以设计出任何你想要的奇葩！
- 一个存储结构设计得是否合理，取决于基于该存储结构的运算是否适合、是否方便，时间复杂度好不好等等。
- 不要拘泥于所学过的有限的数据类型，要把思维放开些，放开些，放开些！



02 二叉树的概念和性质

erchashudexingzhi

二叉树的概念和性质

CONTENTS

01 二叉树的概念

02 二叉树的性质

03 二叉树和树形结构的转换

二叉树的概念和性质

CONTENTS

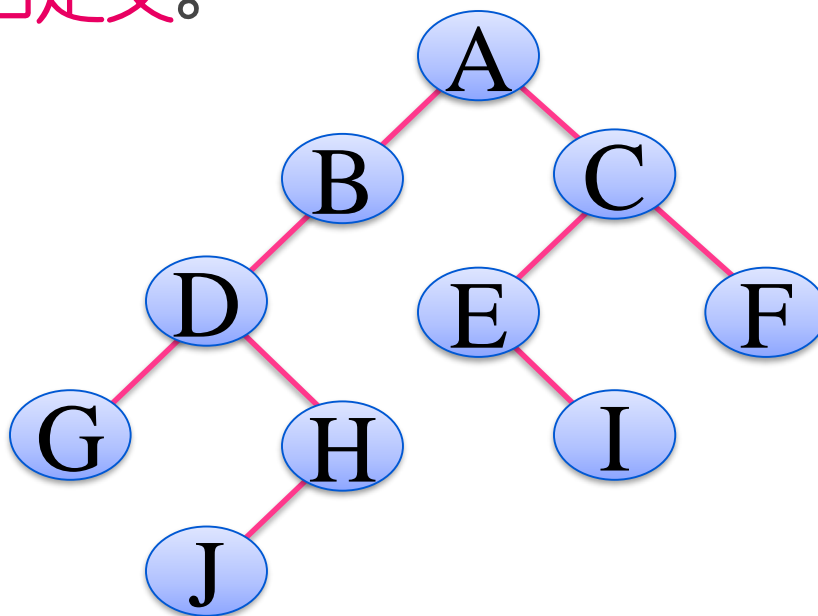
01 二叉树的概念

02 二叉树的性质

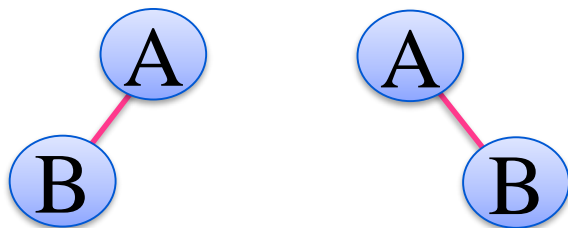
03 二叉树和树形结构的转换

二叉树概念

- 二叉树也称为二次树或二分树
 - 是有限的结点集合, 这个集合或者是空,
 - 或者由一个根结点和两棵互不相交的称为左子树和右子树的二叉树组成。
- 二叉树的定义是一种递归定义。



- 考虑：下面两棵树是否是同一棵树？

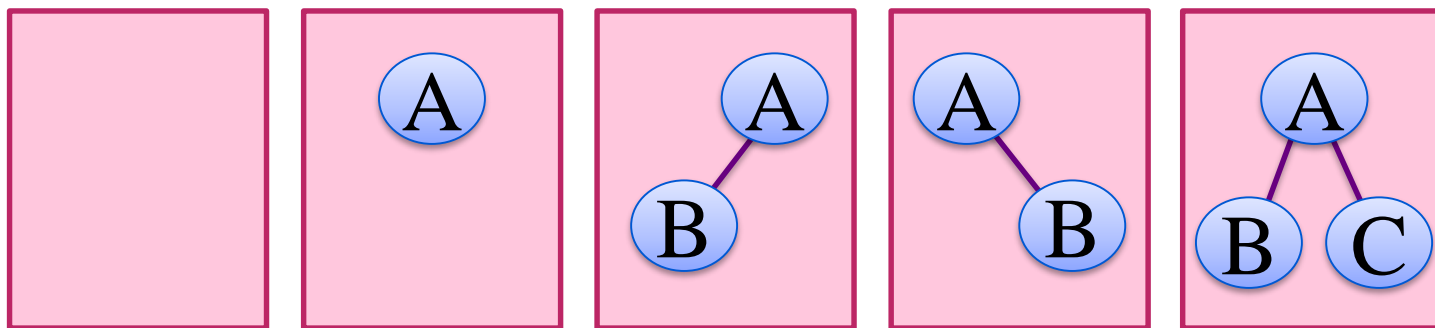


左子树和右子树是有顺序的，次序不能颠倒

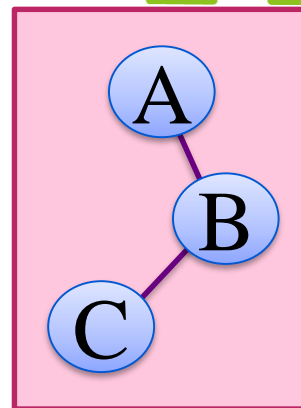
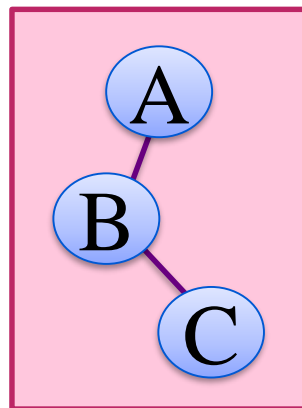
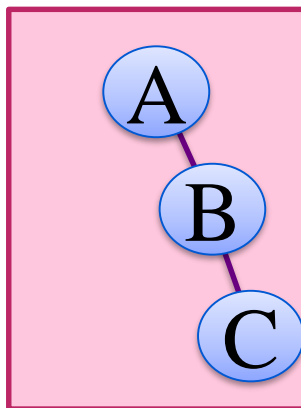
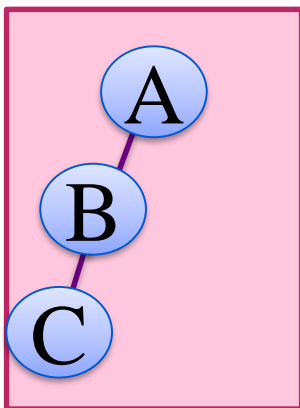
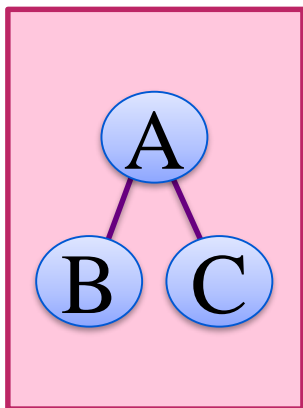
二叉树的五种基本形态

- 空二叉树
- 只有一个根结点
- 根结点只有左子树
- 根结点只有右子树
- 根结点既有左子树又有右子树

任何复杂的二叉树都是这五种基本形态的复合



- 三个结点的二叉树可以有多少种形态？



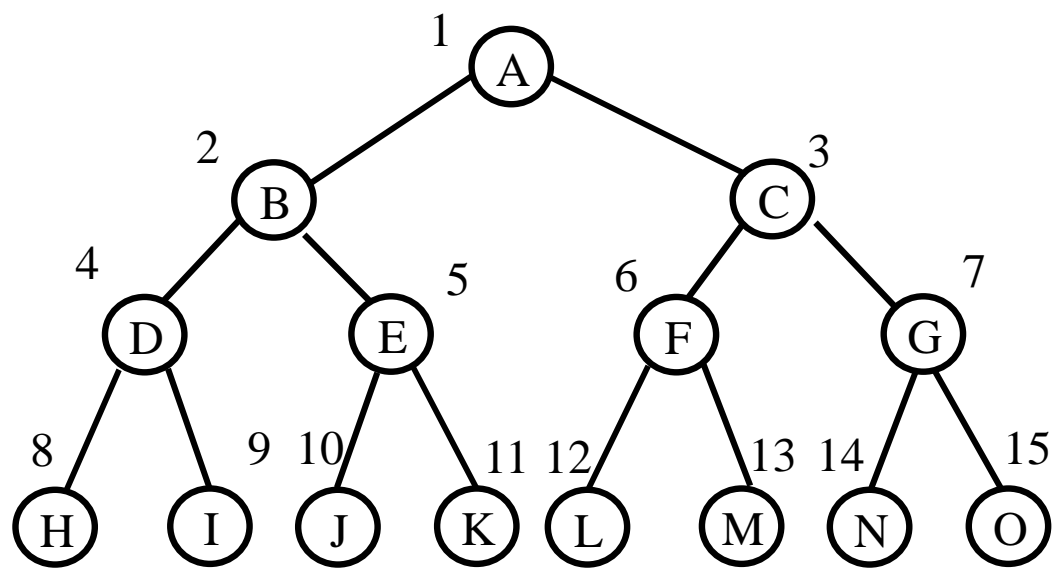
二叉树概念

- 从定义看到, 二叉树是一种特殊的树, 其表示法也与树的表示法一样, 有树形表示法、文氏图表示法、凹入表示法和括号表示法等。

二叉树概念

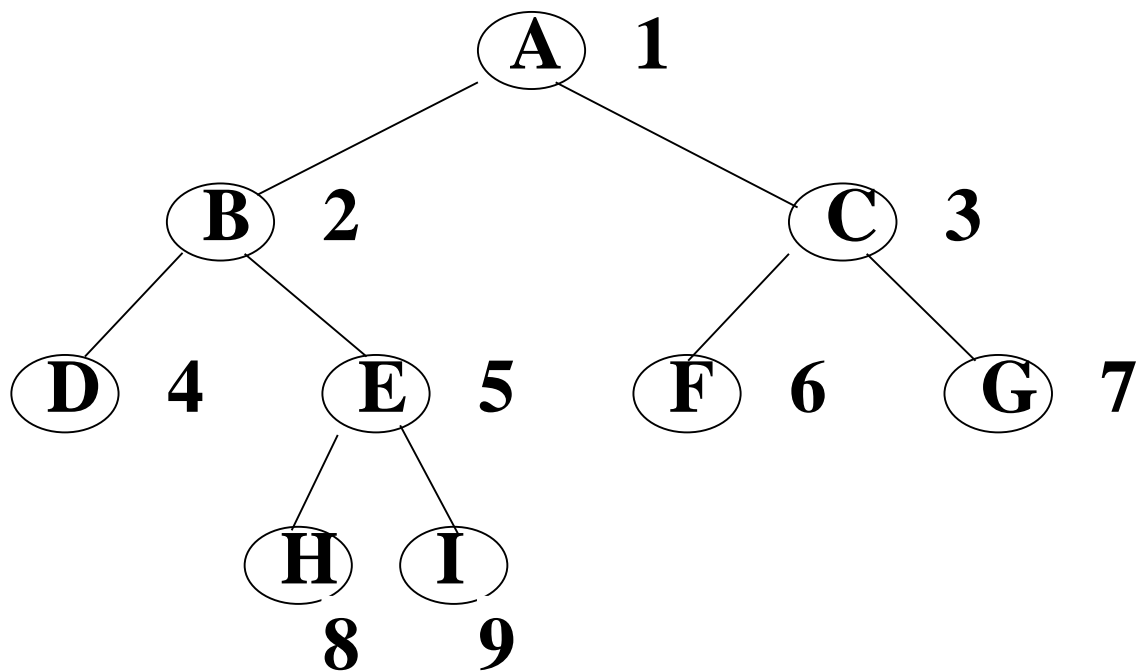
■ 满二叉树

- 在一棵二叉树中, 如果所有分支结点都有左孩子结点和右孩子结点, 并且叶结点都集中在二叉树的最下一层, 这样的二叉树称为满二叉树。



满二叉树

二叉树概念



一棵非满二叉树

- 求解满二叉树的结点个数问题：满二叉树是最严格的二叉树，一旦 n 确定，其树形就确定了，可以计算出高度 h 以及 n_0 、 n_1 和 n_2 。其关系有：

- $h = \log_2(n+1)$

- $n_1 = 0$

- $n = 2^h - 1$

- $n_0 = 2^{h-1}$

- $n_2 = 2^{h-1} - 1$

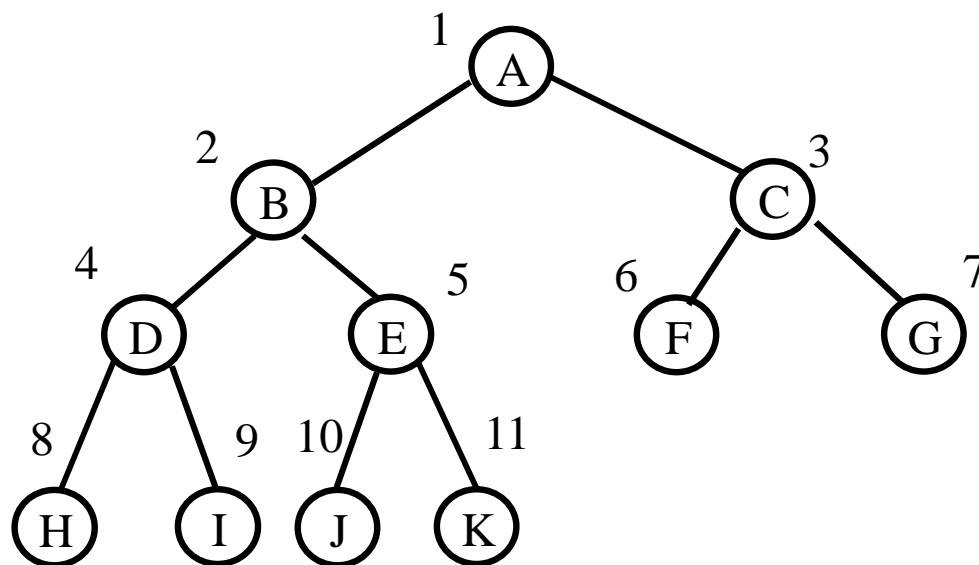
二叉树概念

■ 完全二叉树

- 一棵深度为 k 的有 n 个结点的二叉树，对树中的结点按从上至下、从左到右的顺序进行编号，如果编号为 i ($1 \leq i \leq n$) 的结点与满二叉树中编号为 i 的结点在二叉树中的位置相同，则这棵二叉树称为完全二叉树。

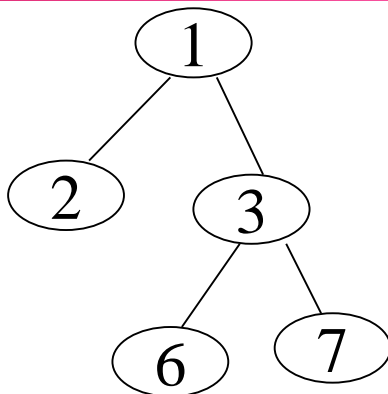
二叉树概念

- 若二叉树中最多只有最下面两层的结点的度数可以小于2, 并且最下面一层的叶结点 都依次排列在该层最左边的位置上。

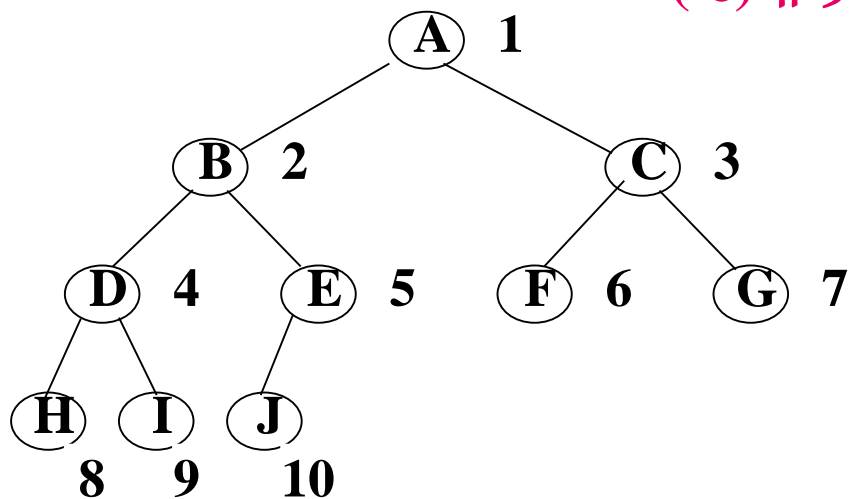


完全二叉树

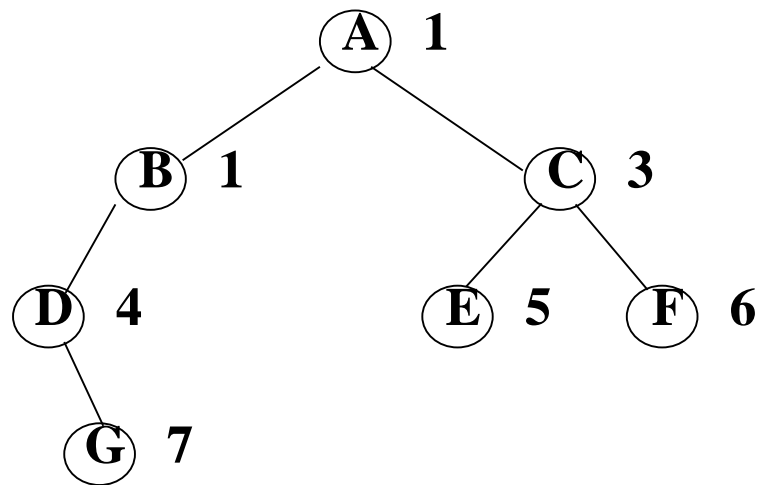
二叉树概念



(c) 非完全二叉树



(a) 一棵完全二叉树



(b) 一棵非完全二叉树

二叉树概念

- 完全二叉树的特点是：

- (1) 所有的叶结点都出现在第 k 层或 $k - 1$ 层。
- (2) 任一结点，如果其右子树的最大层次为 l ，则其左子树的最大层次为 l 或 $l + 1$ 。

- 完全二叉树最典型的应用——堆

- 堆，神奇的优先队列

二叉树的概念和性质

CONTENTS

01 二叉树的概念

02 二叉树的性质

03 二叉树和树形结构的转换

二叉树性质

- 性质1 非空二叉树上叶结点数等于双分支结点数加1。
- $n_0 = n_2 + 1$ (n_0 度为0的结点, 度为2的结点)

证明 设 n 为二叉树的结点总数, n_1 为二叉树中度为1的结点数, 则有:

$$n = n_0 + n_1 + n_2 \quad (1)$$

除根结点外, 其余结点都有一个进入分支, 设 B 为分支总数

$$B = n - 1 \quad (2)$$

由于这些分支都是由度为1和2的结点射出的, 所有有:

$$B = n_1 + 2n_2 \quad (3)$$

综合 (1)、(2)、(3) 式可以得到:

$$n_0 = n_2 + 1$$

二叉树性质

- 性质2 非空二叉树上第 i 层上至多有 2^{i-1} 个结点, 这里应有 $i \geq 1$ 。
- 由树的性质2可推出。

二叉树性质

■ 采用归纳法证明此性质。

- 当 $i=1$ 时，只有一个根结点， $2^{i-1}=2^0=1$ ，成立。
- 现在假定多所有的 j ， $1 \leq j < i$ ，命题成立，即第 j 层上至多有 2^{j-1} 个结点，那么可以证明 $j=i$ 时命题也成立。
由归纳假设可知，第 $i-1$ 层上至多有 2^{i-2} 个结点。
- 由于二叉树每个结点的度最大为2，故在第 i 层上最大结点数为第 $i-1$ 层上最大结点数的二倍，
即 $2 \times 2^{i-2} = 2^{i-1}$ 。

命题得到证明。

【例】

- 已知一棵完全二叉树的第6层（设根为第1层）有8个叶节点，则该完全二叉树的节点个数最多是（ ）。
 - A. 39
 - B. 52
 - C. 111
 - D. 119

二叉树性质

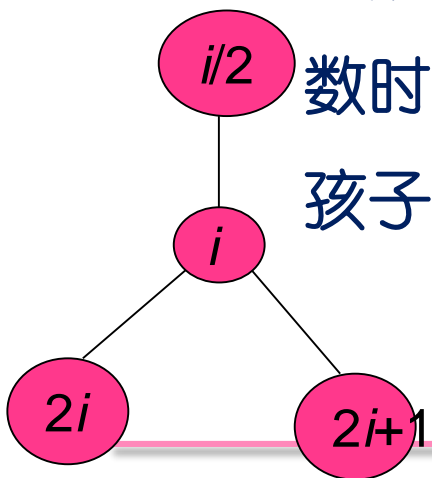
- 性质3 高度为 h 的二叉树至多有 2^h-1 个结点 ($h \geq 1$)。
- 证明
 - 设第 i 层的结点数为 x_i ($1 \leq i \leq h$)，深度为 h 的二叉树的结点数为 M ， x_i 最多为 2^{i-1} ，则有：
 - $$M = \sum_{i=1}^h x_i \leq \sum_{i=1}^h 2^{i-1} = 2^h - 1$$

二叉树性质

- 性质4 对完全二叉树中编号为 i 的结点 ($1 \leq i \leq n, n \geq 1, n$ 为结点数)有:
 - (1) 若 $i \leq n/2$, 即 $2i \leq n$, 则编号为 i 的结点为分支结点, 否则为叶子结点。
 - (2) 若 n 为奇数, 则每个分支结点都既有左孩子结点, 也有右孩子结点; 若 n 为偶数, 则编号最大的分支结点(编号为 $n/2$)只有左孩子结点, 没有右孩子结点, 其余分支结点都有左、右孩子结点。

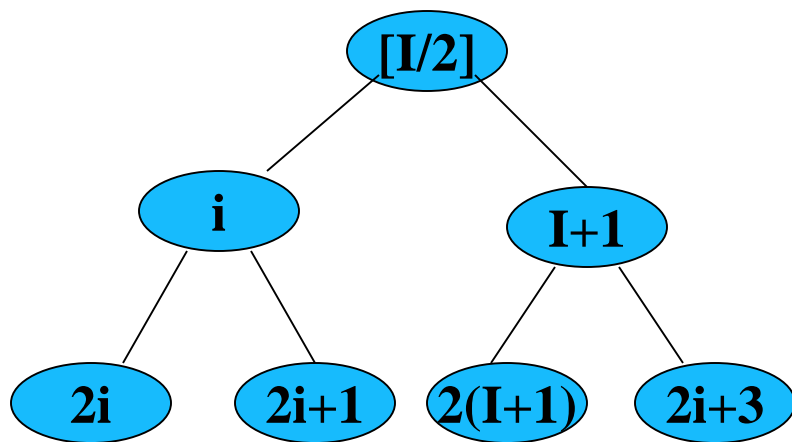
二叉树性质

- (3) 若编号为 i 的结点有左孩子结点, 则左孩子结点的编号为 $2i$; 若编号为 i 的结点有右孩子结点, 则右孩子结点的编号为 $(2i+1)$ 。
- (4) 除树根结点外, 若一个结点的编号为 i , 则它的双亲结点的编号为 $i/2$, 也就是说, 当 i 为偶数时, 其双亲结点的编号为 $i/2$, 它是双亲结点的左孩子结点, 当 i 为奇数时, 其双亲结点的编号为 $(i-1)/2$, 它是双亲结点的右孩子结点。

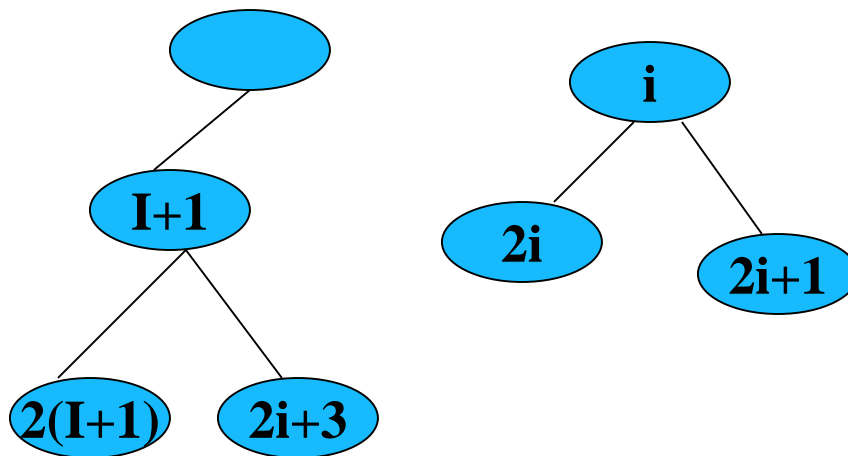


二叉树性质

- 如图 所示为完全二叉树上结点及其左右孩子在结点之间的关系。



(a) I 和 $i+1$ 结点在同一层



(b) I 和 $i+1$ 结点不在同一层

图 完全二叉树中结点 I 和 $i+1$

二叉树性质

- 性质5 具有 n 个 ($n > 0$) 结点的完全二叉树的高度为 $\log_2 n + 1$ 或 $\log_2 n + 1$ 。

证明 根据完全二叉树的定义和性质2可知，当一棵完全二叉树的深度为 k 、结点个数为 n 时，有

$$2^{k-1} - 1 < n \leq 2^k - 1$$

即

$$2^{k-1} \leq n < 2^k$$

对不等式取对数，有

$$k - 1 \leq \log_2 n < k$$

由于 k 是整数，所以有 $k = [\log_2 n] + 1$ 。

二叉树的概念和性质

CONTENTS

01 二叉树的概念

02 二叉树的性质

03 二叉树和树形结构的转换

二叉树与树、森林之间的转换

■ 1. 森林、树转换为二叉树

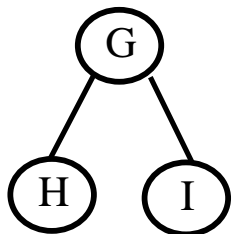
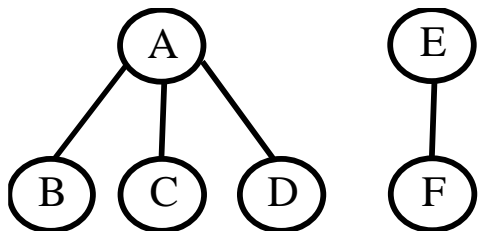
步骤如下：

- (1) 在所有相邻兄弟结点(森林中每棵树的根结点可看成是兄弟结点)之间加一水平连线。
- (2) 对每个非叶结点k, 除了其最左边的孩子结点外, 删去k与其他孩子结点的连线。
- (3) 所有水平线段以左边结点为轴心顺时针旋转45度。

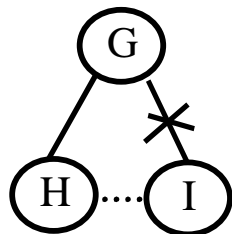
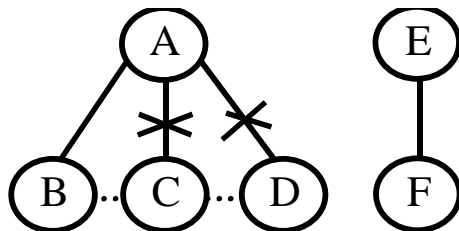
二叉树与树、森林之间的转换

- 通过以上步骤, 原来的森林就转换为一棵二叉树。一般的树是森林中的特殊情况, 由一般的树转换的二叉树的根结点的右孩子结点始终为空, 原因是一般的树的根结点不存在兄弟结点和相邻的树。

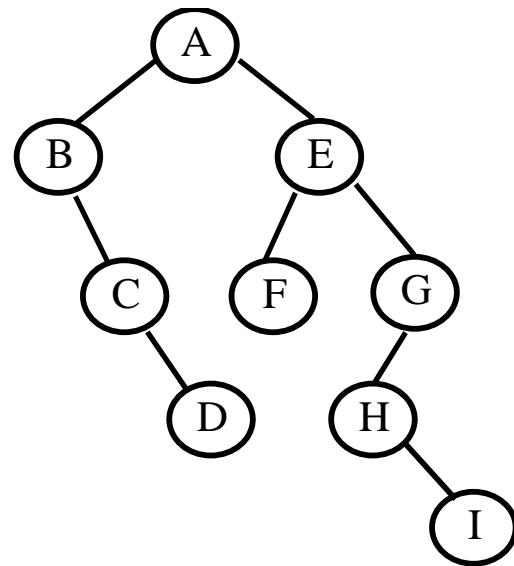
二叉树与树、森林之间的转换



(a)

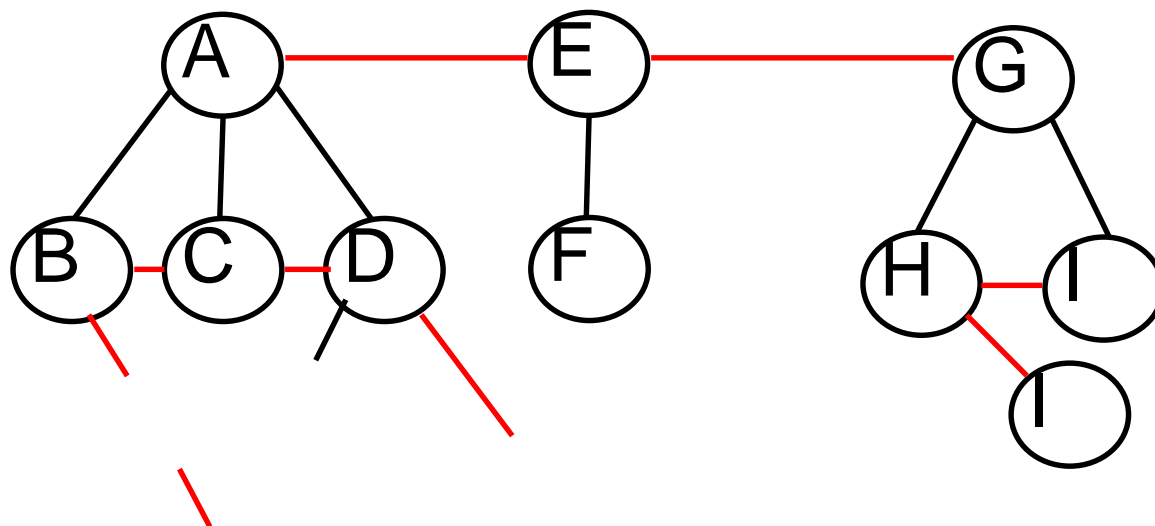


(b)



(c)

将森林转换为二叉树的过程



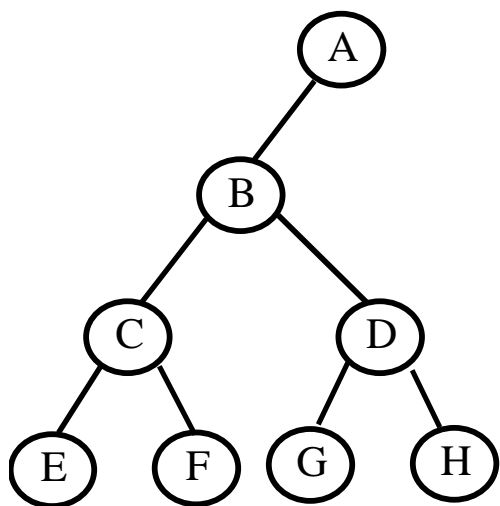
将森林转换为二叉树的过程

二叉树与树、森林之间的转换

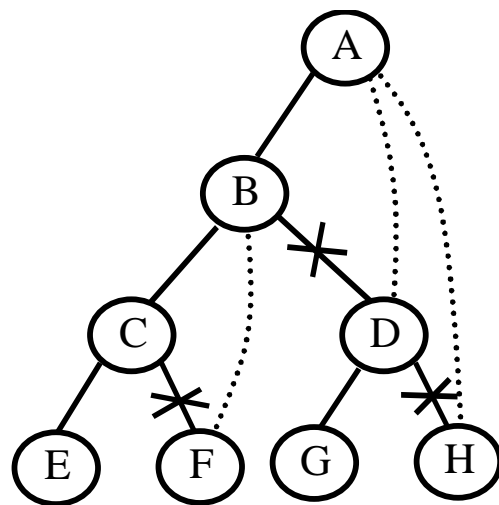
■ 2. 二叉树还原为森林、树

步骤如下：

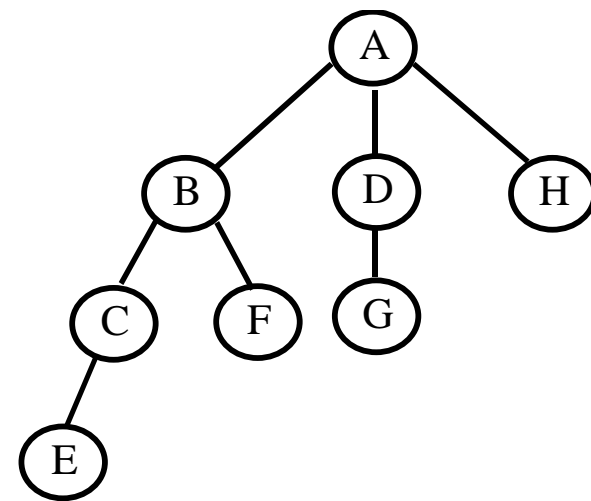
- (1) 对于一棵二叉树中任一结点 k_0 , 沿着 k_0 的左孩子结点 k_1 的右子树方向搜索所有右孩子结点, 即搜索结点序列 k_2, k_3, \dots, k_m , 其中 k_{i+1} 为 k_i 的右孩子结点 ($1 \leq i < m$), k_m 没有右孩子结点。
- (2) 删去 k_1, k_2, \dots, k_m 之间连线。
- (3) 若 k_1 有双亲结点 k , 则连接 k 与 k_i ($2 \leq i \leq m$)。
- (4) 将图形规整化, 使各结点按层次排列。



(a)

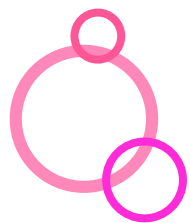


(b)



(c)

将一棵二叉树还原为树的过程



03 二叉树的存储结构

erchashudecunchujiegou

二叉树的存储结构

CONTENTS

01 二叉树的顺序存储结构

02 二叉树的链式存储结构

二叉树的存储结构

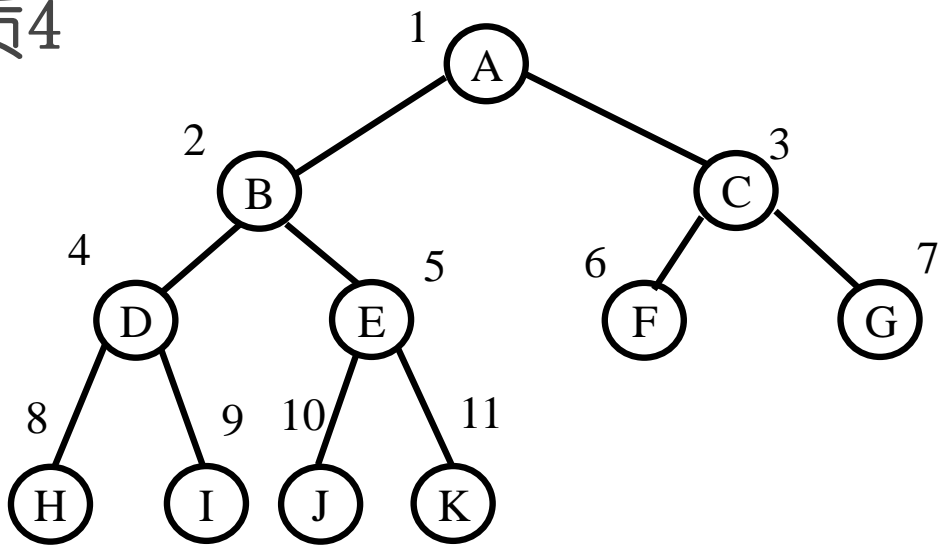
CONTENTS

01 二叉树的顺序存储结构

02 二叉树的链式存储结构

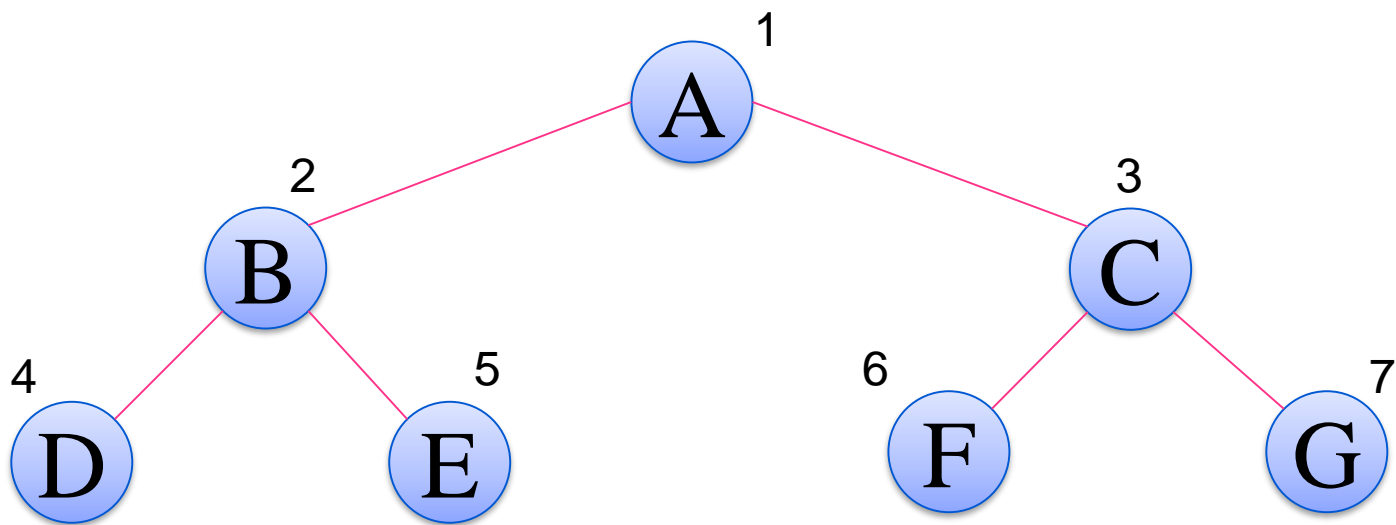
二叉树的顺序存储结构

- 二叉树的顺序存储结构中结点的存放次序是：
 - 对该树中每个结点进行编号。
 - 其编号从小到大的顺序就是结点存放在连续存储单元的先后次序。
- 利用二叉树的性质4



二叉树的顺序存储结构

■ 【完全二叉树】



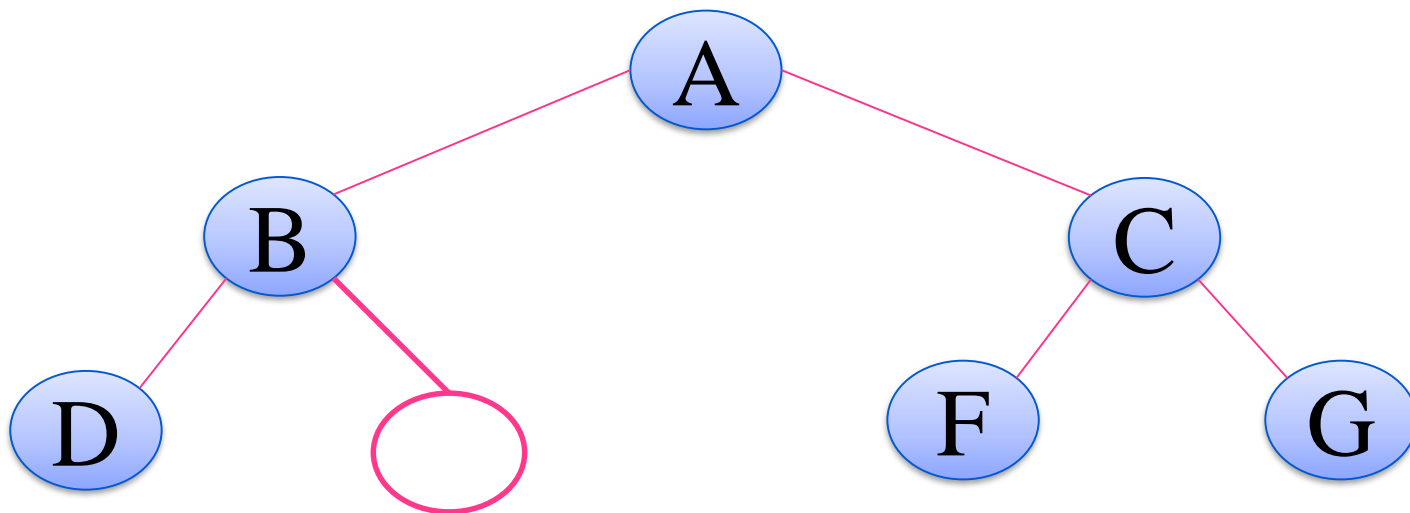
元素	A	B	C	D	E	F	G
下标	1	2	3	4	5	6	7

编号为 i 的节点的左孩子编号为 $2i$ ；右孩子编号为 $2i+1$

除根节点外，编号为 i 的节点它的双亲节点的编号为 $\lfloor i/2 \rfloor$

二叉树的顺序存储结构

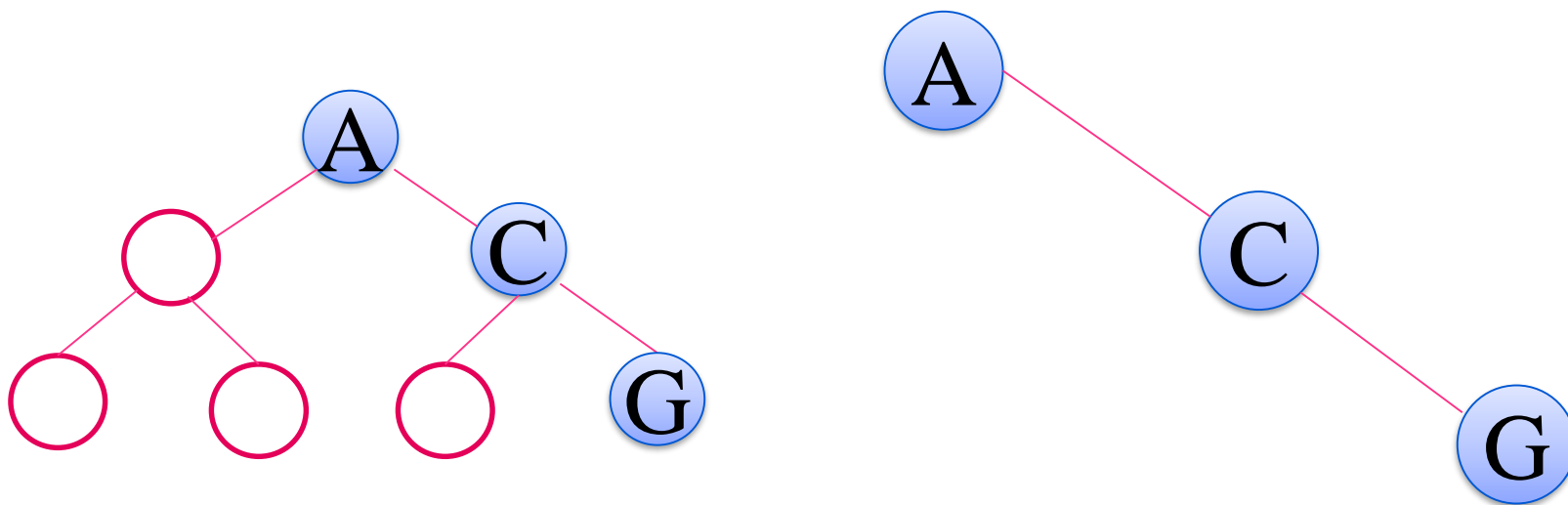
■ 【一般二叉树】



元素	A	B	C	D	^	F	G
下标	1	2	3	4	5	6	7

二叉树的顺序存储结构

■ 【极端情况——单分支情况】



改造后的完全二叉树

元素	A	^	C	^	^	^	G
下标	1	2	3	4	5	6	7

- 【总结】

二叉树的顺序存储方式的适用性不强

二叉树的存储结构

CONTENTS

01 二叉树的顺序存储结构

02 二叉树的链式存储结构

二叉树的链式存储结构

- 所谓二叉树的链式存储结构是指，用链表来表示一棵二叉树，即用链来指示元素的逻辑关系。
- (1) 二叉链表存储
 - 链表中每个结点由三个域组成，除了数据域外，还有两个指针域，分别用来给出该结点左孩子和右孩子所在的链结点的存储地址。

lchild	data	rchild
--------	------	--------

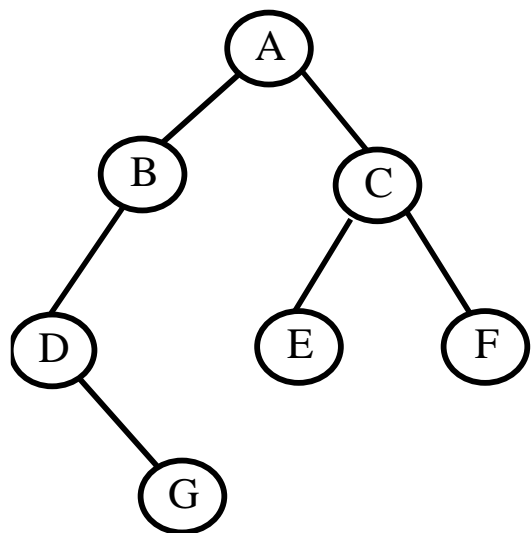
(1) 二叉链表存储

- 在二叉树的链接存储中, 结点的结构如下:

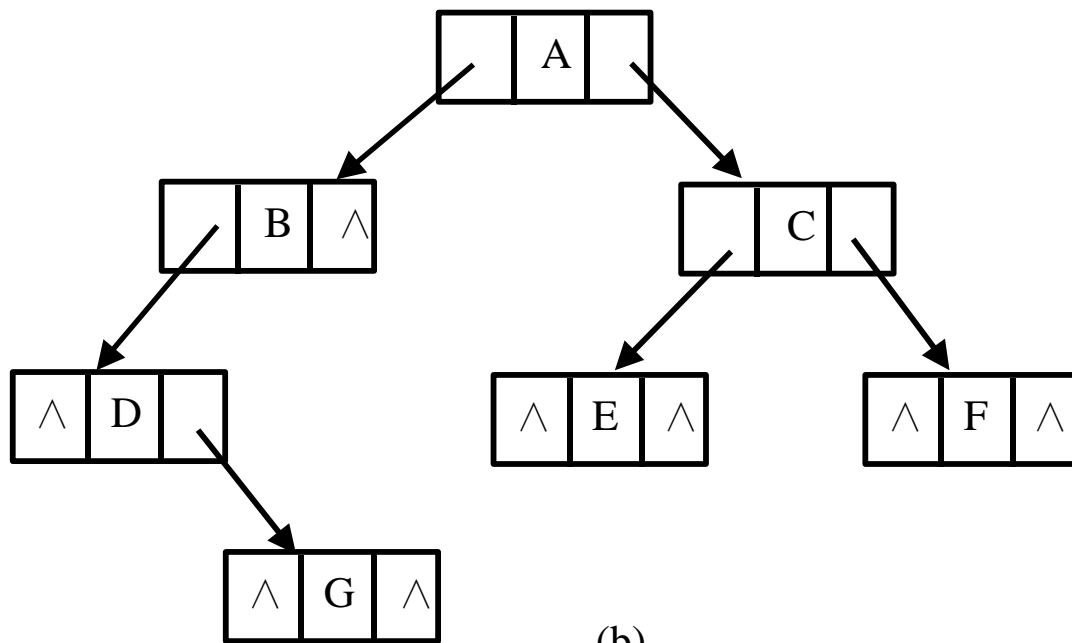
```
typedef struct node
{ ElemType data;
    struct node *lchild,*rchild;
} BTreeNode;
```

- data: 值域, 存储对应的数据元素
- lchild: 左指针域, 存储左孩子结点
- rchild: 右指针域, 存储右孩子结点

(1) 二叉链表存储



(a)

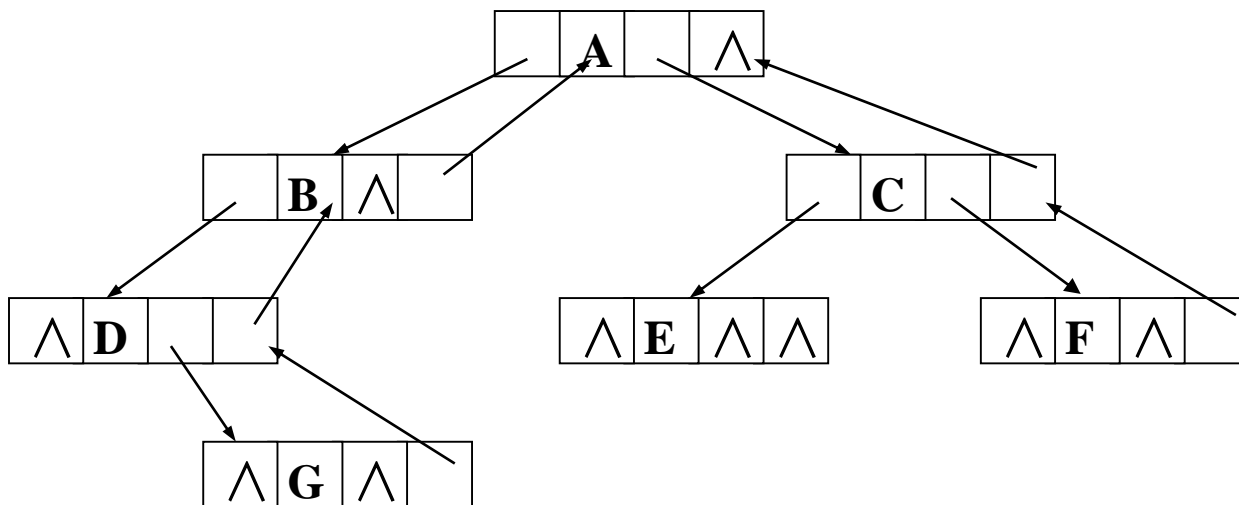


(b)

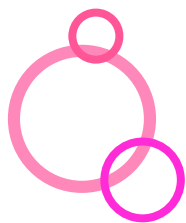
二叉树及其链式存储结构

(2) 三叉链表存储

lchild	data	rchild	parent
--------	------	--------	--------



二叉树的三叉链表表示示意图



04 二叉树的基本运算

shudedingyi

二叉树的基本运算及其实现

CONTENTS

01

二叉树的基本运算概述

02

二叉树的基本运算算法实现

二叉树的基本运算

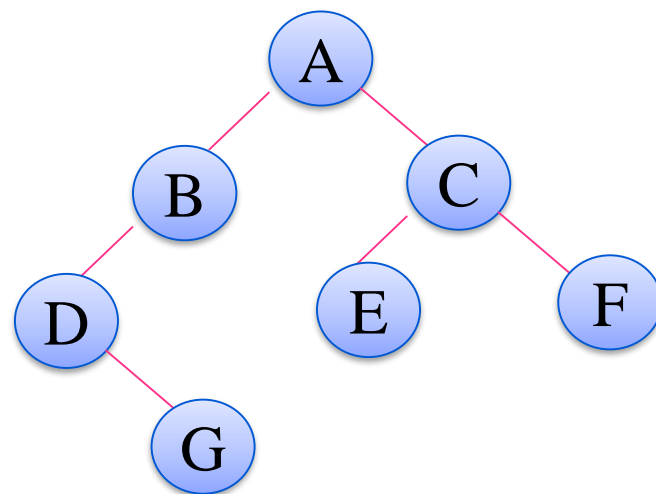
- 归纳起来, 二叉树有以下基本运算:
- (1) 创建二叉树 `CreateBTNode(*b, *str)`:
 - 根据二叉树括号表示法的字符串 `*str` 生成对应的链式存储结构。
- (2) 查找结点 `FindNode(*b, x)`:
 - 在二叉树 `b` 中寻找 `data` 域值为 `x` 的结点, 并返回指向该结点的指针。
- (3) 找孩子结点 `LchildNode(p)` 和 `RchildNode(p)`
 - 分别求二叉树中结点 `*p` 的左孩子结点和右孩子结点。

二叉树的基本运算

- (4) 求高度BTNodeDepth(*b) :
 - 求二叉树b的高度。若二叉树为空, 则其高度为0; 否则, 其高度等于左子树与右子树中的最大高度加1。
- (5) 输出二叉树DispBTNode(*b)
 - 以括号表示法输出一棵二叉树。

二叉树的基本运算算法实现

- (1) 创建二叉树CreateBTNode(*b, *str)
- 例: str ---- **A(B(D(,G)),C(E,F))**
- 用ch扫描采用括号表示法表示二叉树的字符串。
- 使用一个栈st保存双亲结点, 根据k的值判断其后处理的节点是保存在栈中的左孩子还是右孩子



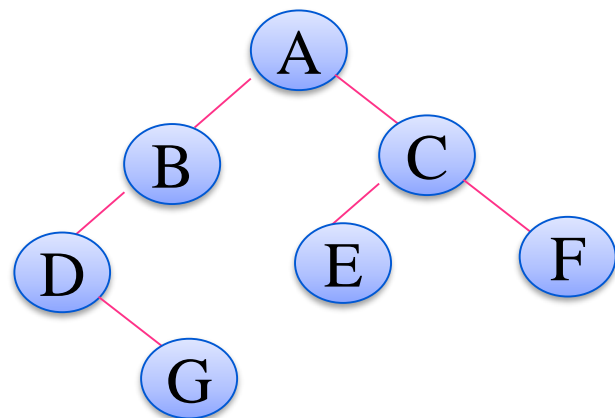
二叉树的基本运算算法实现

分以下几种情况：

- ① 若ch=' ('：则将前面刚创建的结点作为双亲结点进栈，并置k=1，表示其后创建的结点将作为这个结点的左孩子结点；
- ② 若ch=')'：表示栈中结点的左右孩子结点处理完毕，退栈；
- ③ 若ch=' , '：表示其后创建的结点为右孩子结点；
- ④ 其他情况，表示要创建一个结点，并根据k值建立它与栈中结点之间的联系。

当k=1时，表示这个结点作为栈中结点的左孩子结点

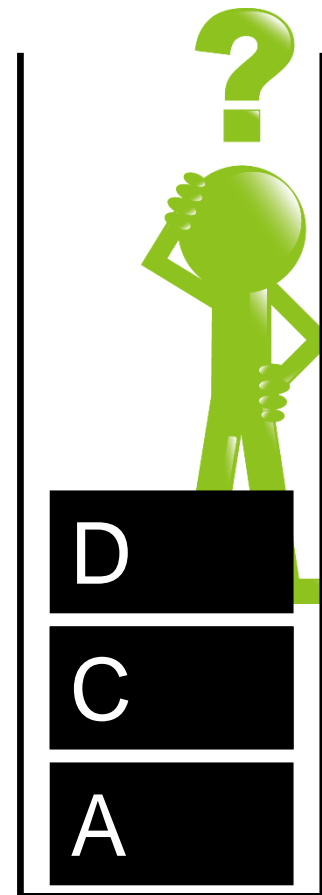
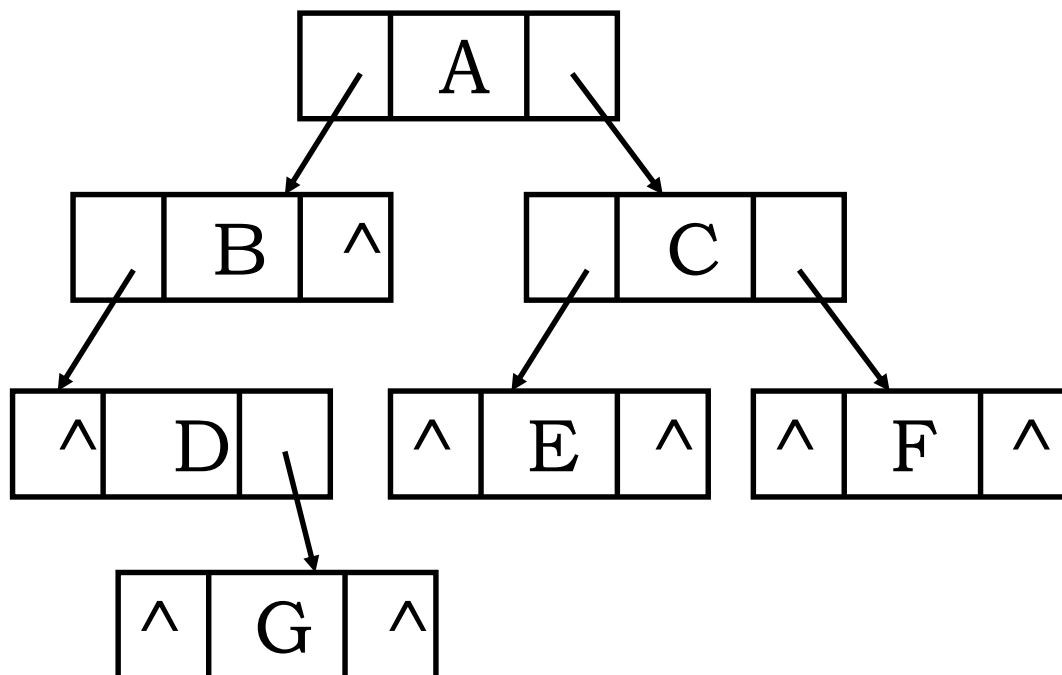
k=2时，表示这个结点作为栈中结点的右孩子结点。



$A(B(D(, G)), C(E, F))$

根据括号表示法字符串构造二叉树

A (B (D (, G)) , C (E , F))



k= 2

二叉树的基本运算算法实现

```
void CreateBTNode(BTNode * &b,char *str)
{
    BTNode *St[MaxSize],*p=NULL;
    int top=-1,k,j=0;
    char ch;
    b=NULL;                                /*建立的二叉树初始时空*/
    ch=str[j];
    while (ch!='\0')                        /*str未扫描完时循环*/
    {
        switch(ch)
        {
            case '(':top++;St[top]=p;k=1; break; /*为左孩子*/
            case ')':top--;break;
            case ',':k=2; break;                /*为孩子结点右结点*/
        }
    }
}
```

二叉树的基本运算算法实现

```
default:p=(BTNode *)malloc(sizeof(BTNode));
```

```
p->data=ch;p->lchild=p->rchild=NULL;
```

```
if (b==NULL)                /**p为二叉树的根结点*/
```

```
    b=p;
```

```
else                          /*已建立二叉树根结点*/
```

```
    { switch(k)
```

```
        {
```

```
            case 1:St[top]->lchild=p;break;
```

```
            case 2:St[top]->rchild=p;break;
```

```
        }
```

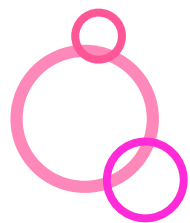
```
    }
```

```
}
```

```
j++;ch=str[j];
```

```
}
```

```
}
```



05 二叉树的遍历

shudedingyi

二叉树的遍历

CONTENTS

01 二叉树的遍历的概念

02 二叉树的遍历递归算法

03 二叉树和遍历非递归算法

二叉树的遍历

CONTENTS

01 二叉树的遍历的概念

02 二叉树的遍历递归算法

03 二叉树和遍历非递归算法

二叉树遍历的概念

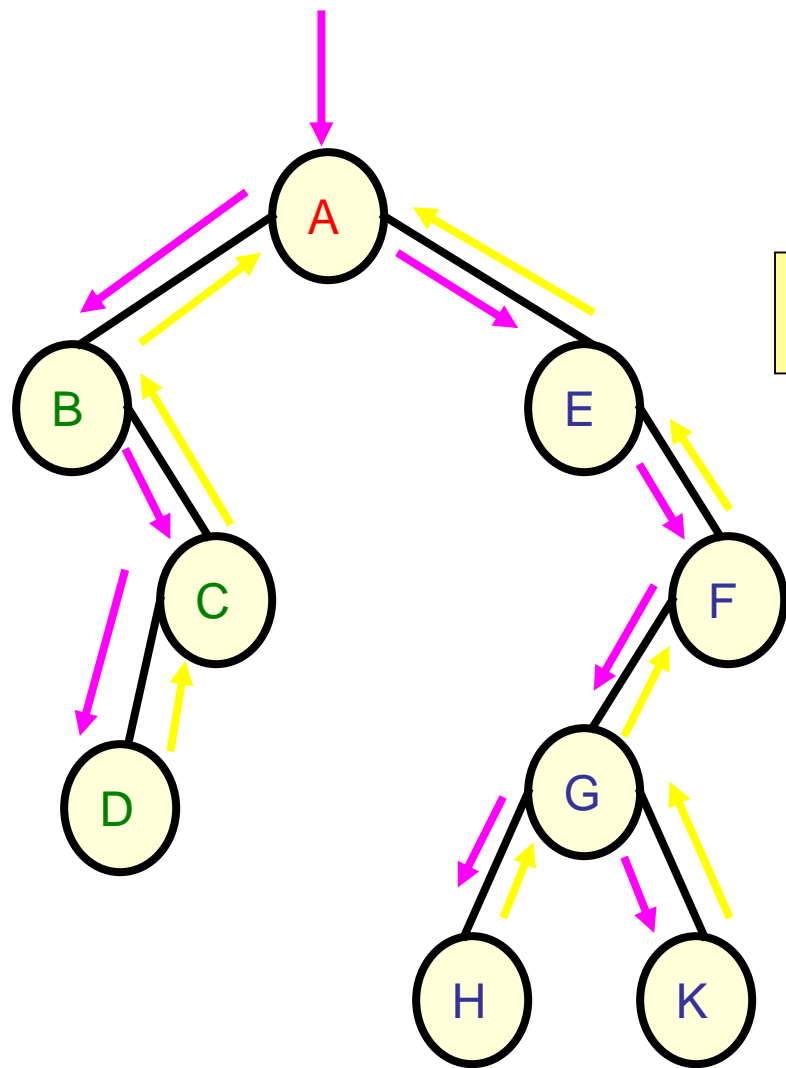
- 二叉树的遍历是指按照一定次序访问树中所有结点, 并且每个结点**仅被访问一次**的过程。
- **它是最基本的运算, 是二叉树中所有其他运算的基础。**

二叉树的遍历方式

■ 1. 先序遍历

先序遍历二叉树的过程是：

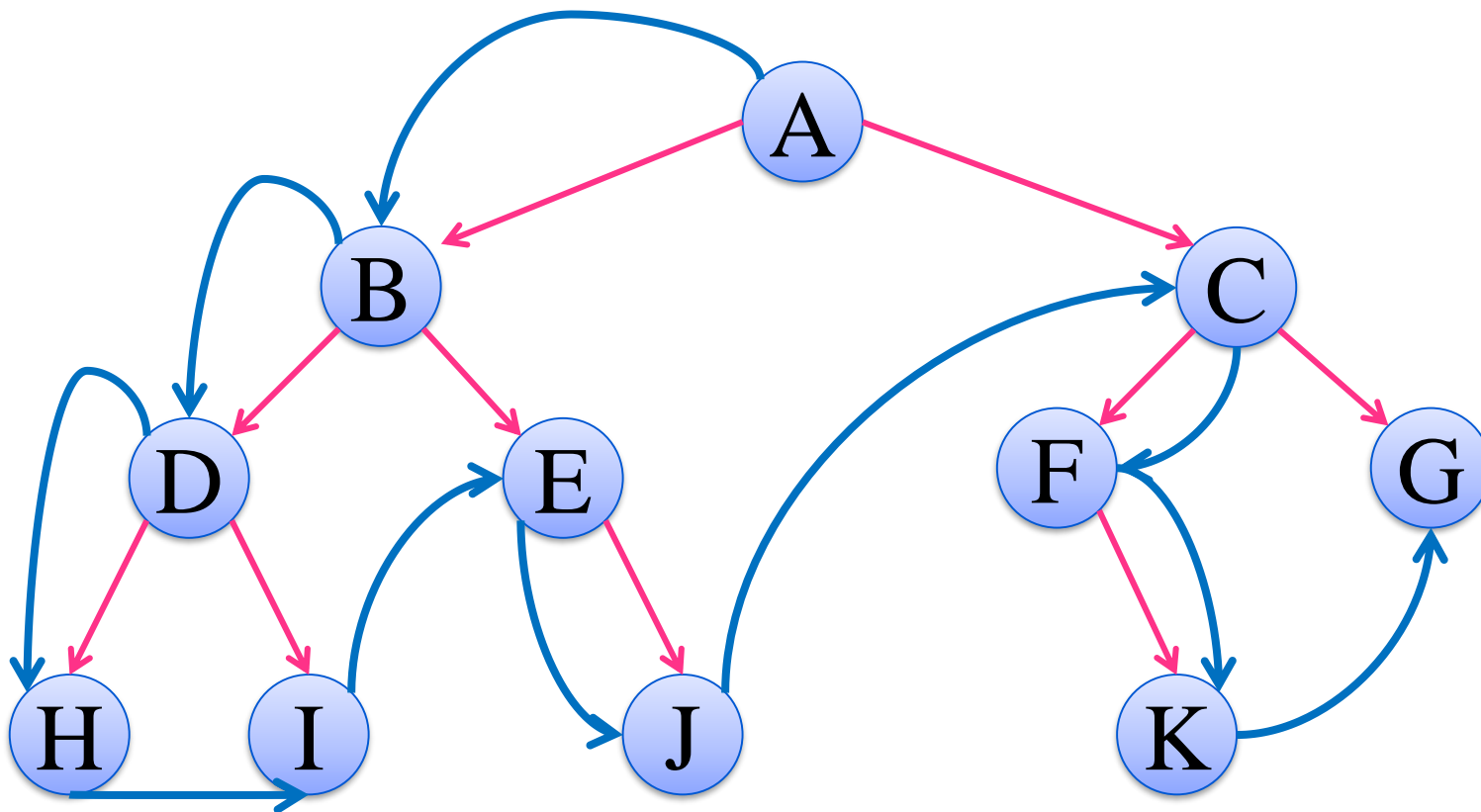
- (1) 访问根结点；
- (2) 先序遍历左子树；
- (3) 先序遍历右子树。



先序序列:

A	B	C	D	E	F	G	H	K
---	---	---	---	---	---	---	---	---

二叉树的遍历方法



- 遍历的顺序为：ABDHIEJCFKG

二叉树的遍历方式

■ 2. 中序遍历

中序遍历二叉树的过程是：

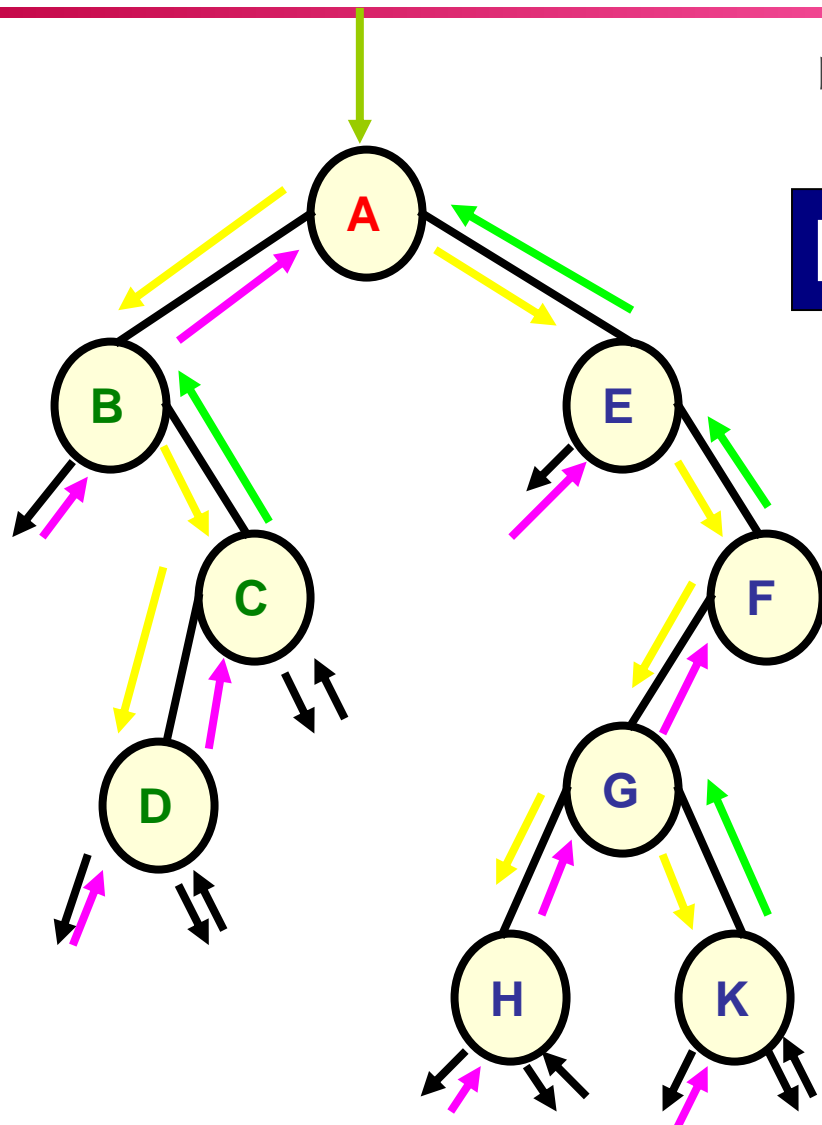
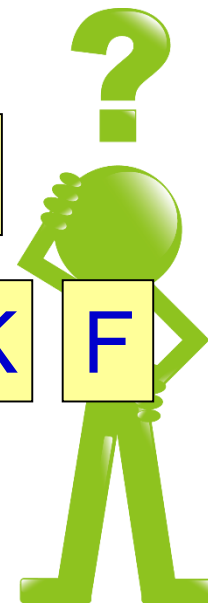
- (1) 中序遍历左子树；
- (2) 访问根结点；
- (3) 中序遍历右子树。

中序序列:

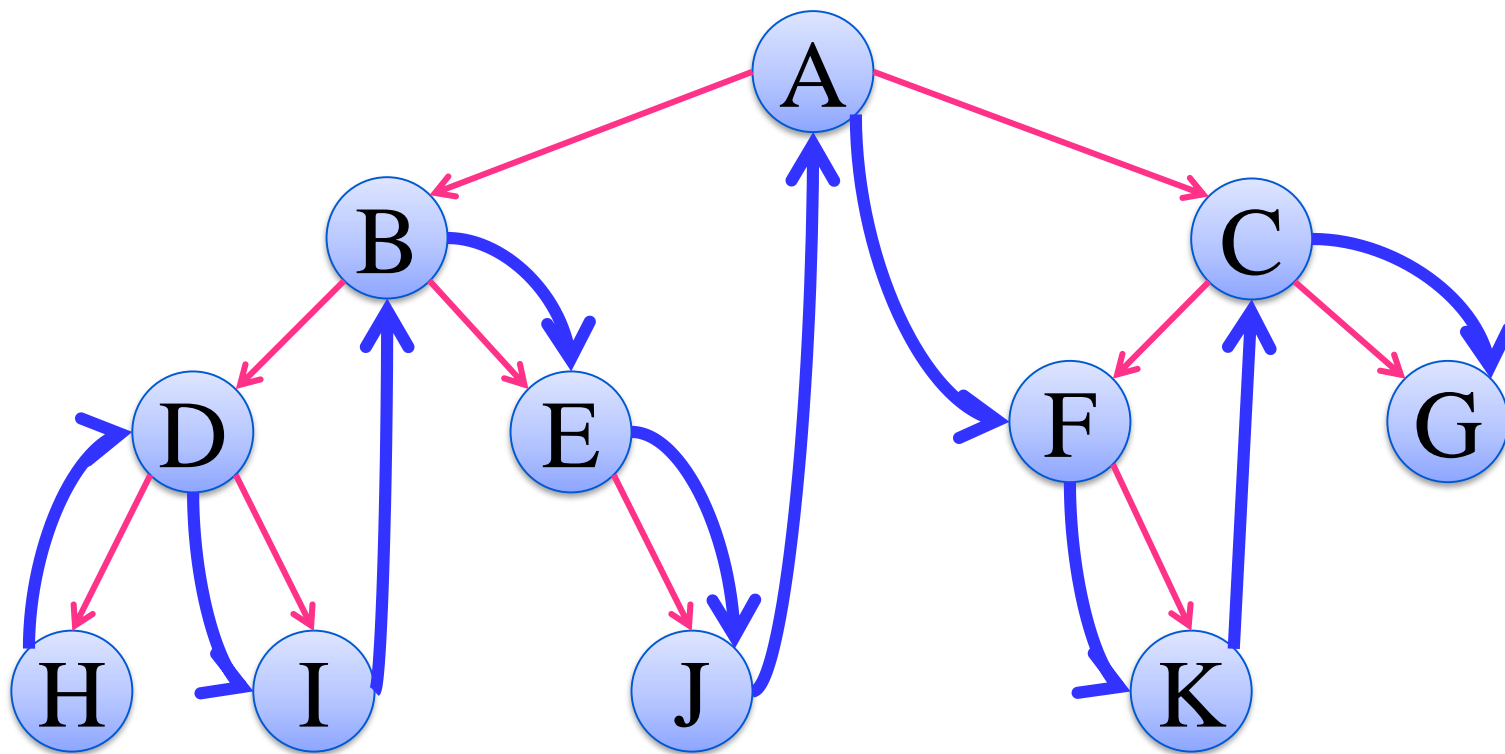
B D C

E H G K F

A



二叉树的遍历方法



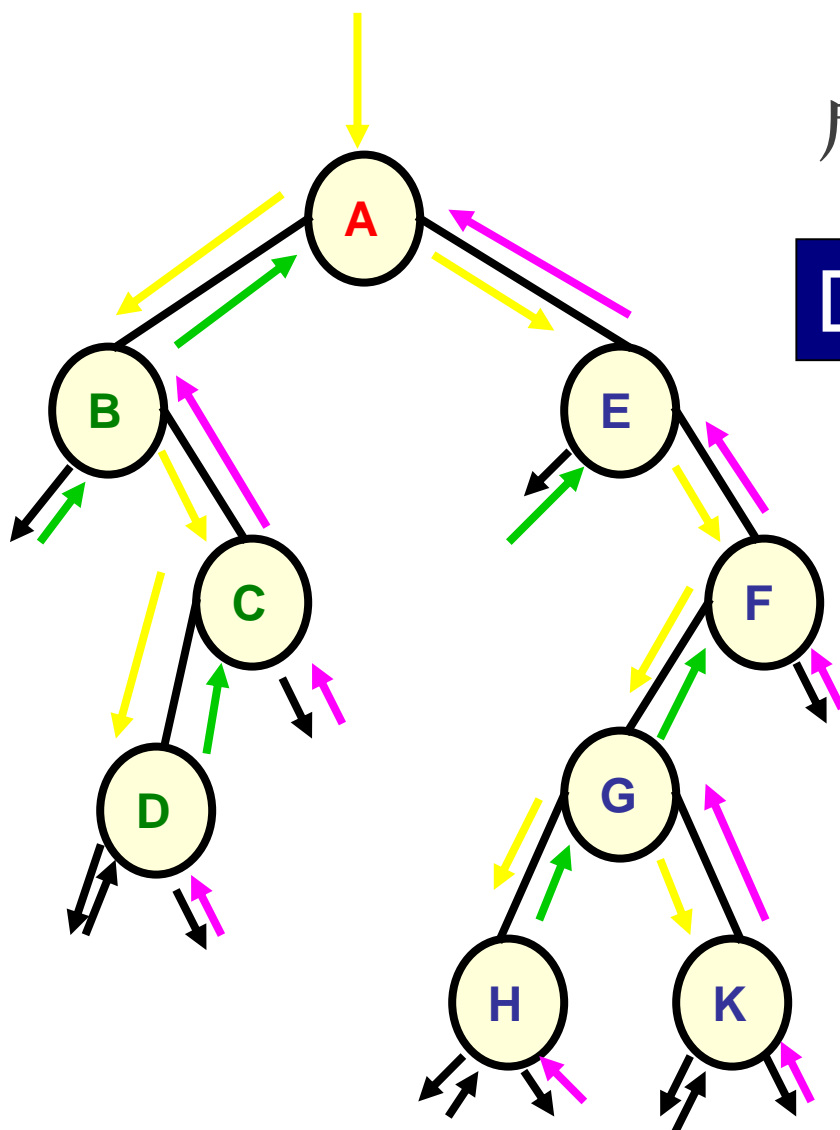
- 遍历的顺序为：HDIBEJAFKCG

二叉树的遍历方式

■ 3. 后序遍历

后序遍历二叉树的过程是：

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

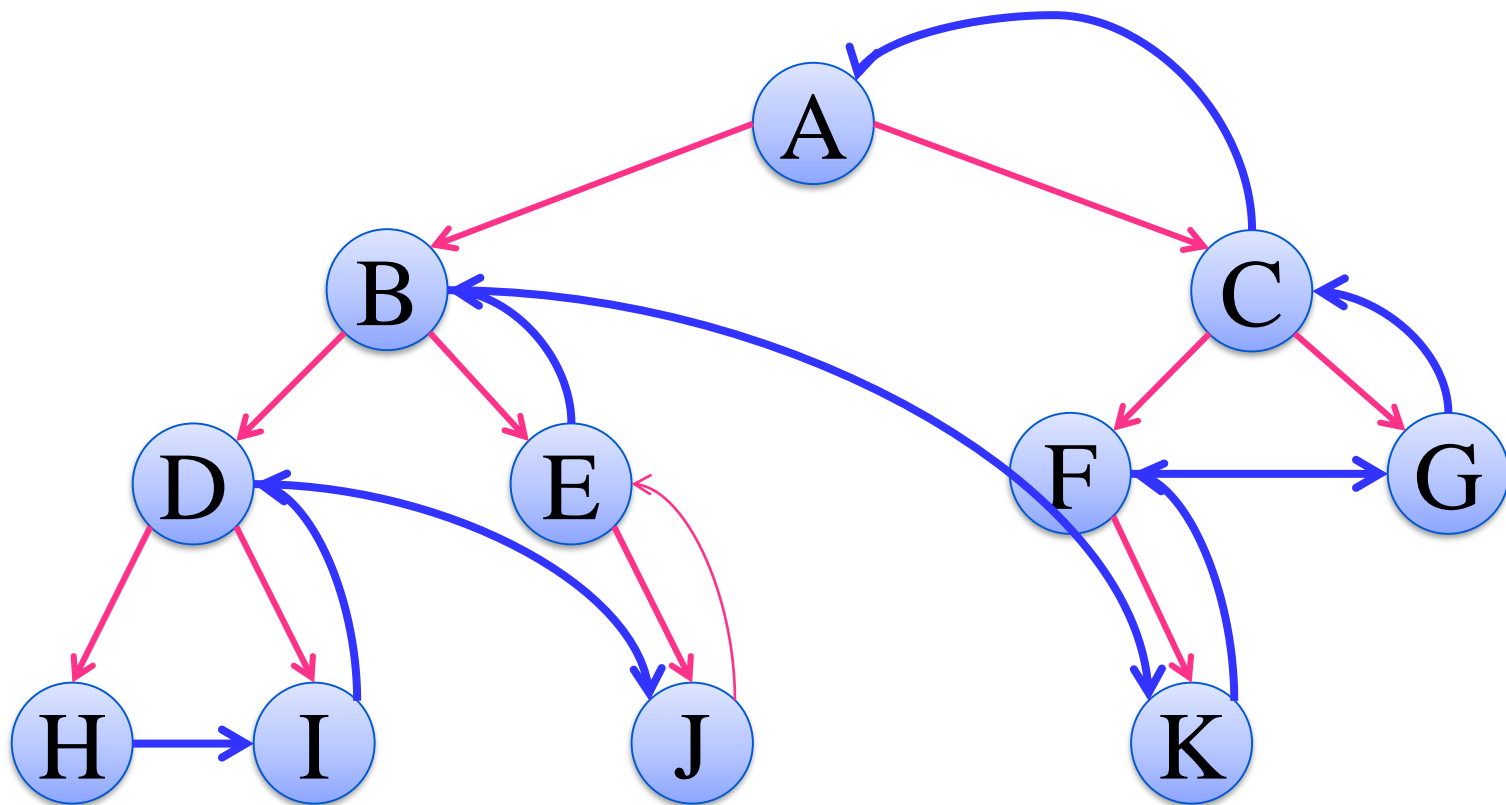


后序序列:

D C B

H K G F E
A

二叉树的遍历方法



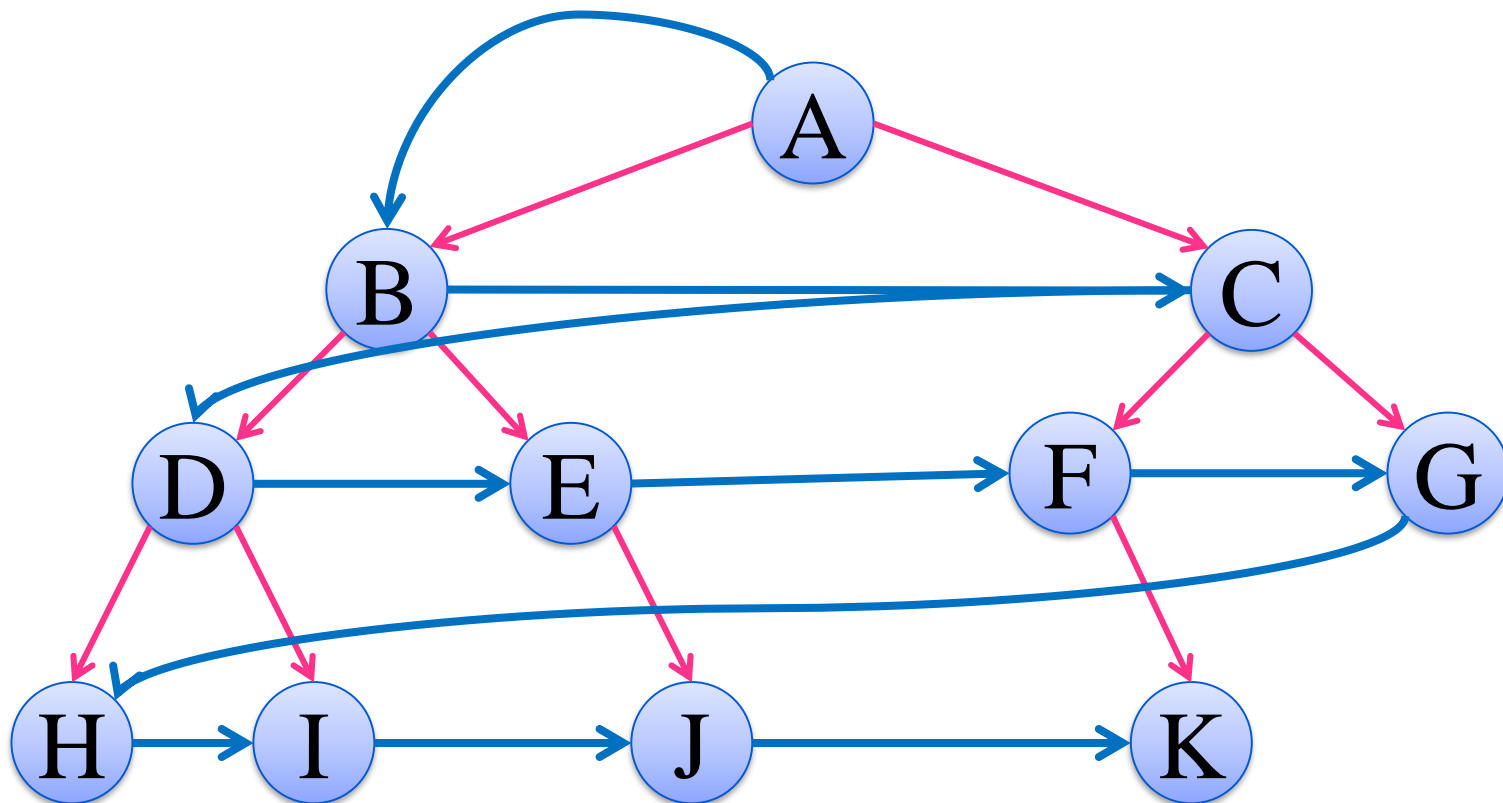
- 遍历的顺序为：HIDJEBKFGCA

二叉树的遍历方法

- 层序遍历：

- 若树为空，则空操作返回，否则从树的第一层，也就是根结点开始访问，从上而下逐层遍历，在同一层中，按从左到右的顺序对结点逐个访问。

二叉树的遍历方法



- 遍历的顺序为：ABCDEF GHI JK

-
- 研究这么多遍历的方法干啥呢？



二叉树的基本运算算法实现

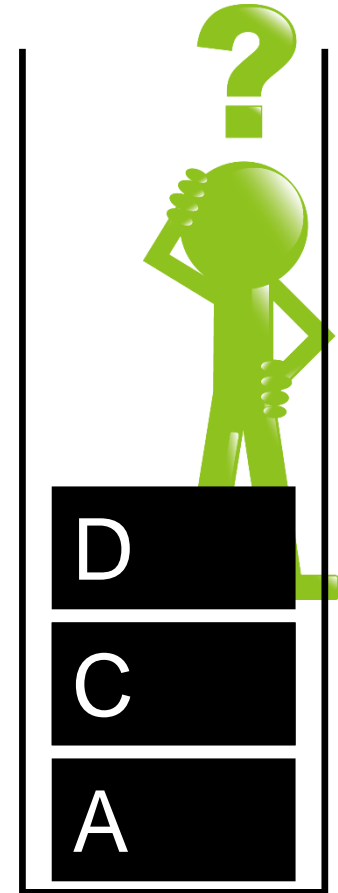
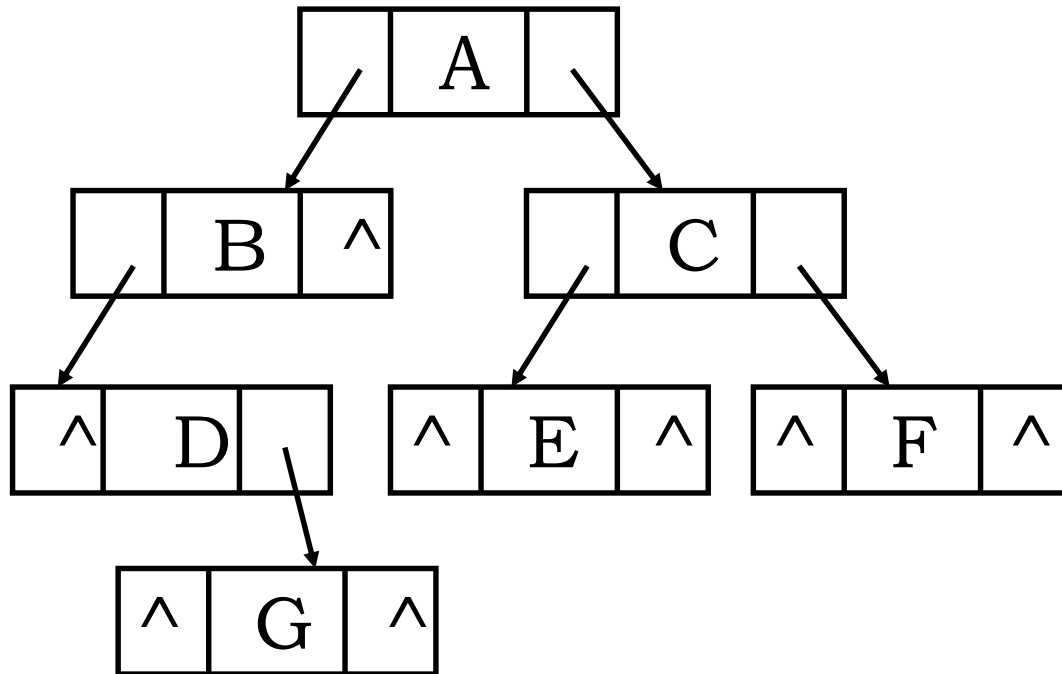
- (1) 创建二叉树CreateBTNode(*b, *str)
- 用ch扫描采用括号表示法表示二叉树的字符串。分以下几种情况：
 - ① 若ch='('：则将前面刚创建的结点作为双亲结点进栈, 并置k=1, 表示其后创建的结点将作为这个结点的左孩子结点;
 - ② 若ch=')'：表示栈中结点的左右孩子结点处理完毕, 退栈;
 - ③ 若ch=', '：表示其后创建的结点为右孩子结点;

二叉树的基本运算算法实现

- ④ 其他情况, 表示要创建一个结点, 并根据 k 值建立它与栈中结点之间的联系.
- 当 $k=1$ 时, 表示这个结点作为栈中结点的左孩子结点, 当
- $k=2$ 时, 表示这个结点作为栈中结点的右孩子结点。

根据括号表示法字符串构造二叉树

A (B (D (, G)) , C (E , F))



k= 2

二叉树的基本运算算法实现

```
void CreateBTNode(BTNode * &b,char *str)
{
    BTNode *St[MaxSize],*p=NULL;
    int top=-1,k,j=0;
    char ch;
    b=NULL;
    ch=str[j];
    while (ch!='\0')
    {
        switch(ch)
        {
            case '(':top++;St[top]=p;k=1; break; /*为左孩子*/
            case ')':top--;break;
            case ',':k=2; break; /*为孩子结点右结点*/
        }
    }
}
```

A (B (D (, G)) , C (E , F))

二叉树的基本运算算法实现

```
default:p=(BTNode *)malloc(sizeof(BTNode));
```

```
p->data=ch;p->lchild=p->rchild=NULL;
```

```
if (b==NULL)                /**p为二叉树的根结点*/
```

```
    b=p;
```

```
else                          /*已建立二叉树根结点*/
```

```
    { switch(k)
```

```
        {
```

```
            case 1:St[top]->lchild=p;break;
```

```
            case 2:St[top]->rchild=p;break;
```

```
        }
```

```
    }
```

```
}
```

```
j++;ch=str[j];
```

```
}
```

```
}
```

A (B (D (, G)) , C (E , F))

二叉树遍历的递归算法

- 由二叉树的三种遍历过程直接得到如下三种递归算法如下：

```
void PreOrder(BTNode *b) /*先序遍历的递归算法*/
{
    if (b!=NULL)
    { printf("%c ",b->data);    /*访问根结点*/
      PreOrder(b->lchild);
      PreOrder(b->rchild);
    }
}
```

二叉树遍历的递归算法

```
void InOrder(BTNode *b)    /*中序遍历的递归算法*/
{
    if (b!=NULL)
    {inOrder(b->lchild);
        printf("%c ",b->data); /*访问根结点*/
        InOrder(b->rchild);
    }
}
```

二叉树遍历的递归算法

```
void PostOrder(BTNode *b)      /*后序遍历递归算法*/
{
    if (b!=NULL)
    {
        PostOrder(b->lchild);
        PostOrder(b->rchild);
        printf("%c ",b->data);  /*访问根结点*/
    }
}
```

- 假设二叉树采用二叉链存储结构存储, 试设计一个算法, 输出一棵给定二叉树的所有叶子结点。

f(b) :	不做任何事件	若b=NULL
f(b) :	输出*b结点的data域	若*b为叶子结点
f(b) :	f(b->lchild);f(b->rchild)	其他情况

【例】

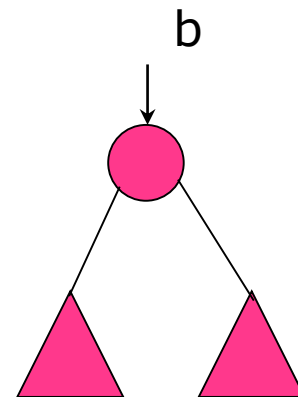
```
void DispLeaf(BTNode *b)
{
    if (b!=NULL)
    {
        if (b->lchild==NULL && b->rchild==NULL)
            printf("%c ",b->data);
        else
        {
            DispLeaf(b->lchild);
            DispLeaf(b->rchild);
        }
    }
}
```

二叉树遍历非递归算法（先序1）

步骤如下：

根左右

```
if (当前b树不空)
{
    根节点b进栈;
    while (栈不空)
    {
        出栈节点p并访问之;
        若*p节点有右孩子, 将其右孩子进栈;
        若*p节点有左孩子, 将其左孩子进栈;
    }
}
```



二叉树遍历非递归算法 (先序1)

```
void PreOrder1(BTNode *b)
```

```
{    BTNode *St[MaxSize],*p; int top=-1;
```

```
    top++; St[top]=b;
```

```
    while (top>-1)
```

```
    {    p=St[top]; top--;
```

```
        printf("%c ",p->data);
```

```
        if (p->rchild!=NULL)
```

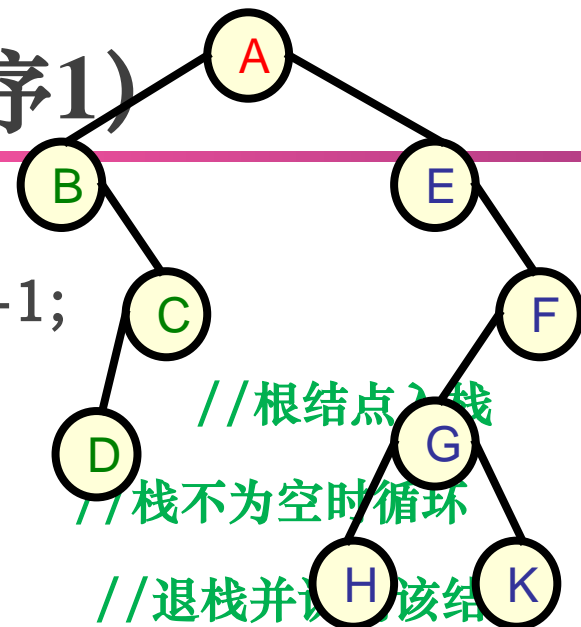
```
        {    top++; St[top]=p->rchild;    }
```

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

```
        {    top++; St[top]=p->lchild;    }
```

```
    }
```

```
}
```

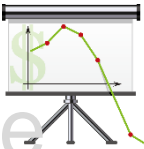


//右孩子结点入栈

//左孩子结点入栈

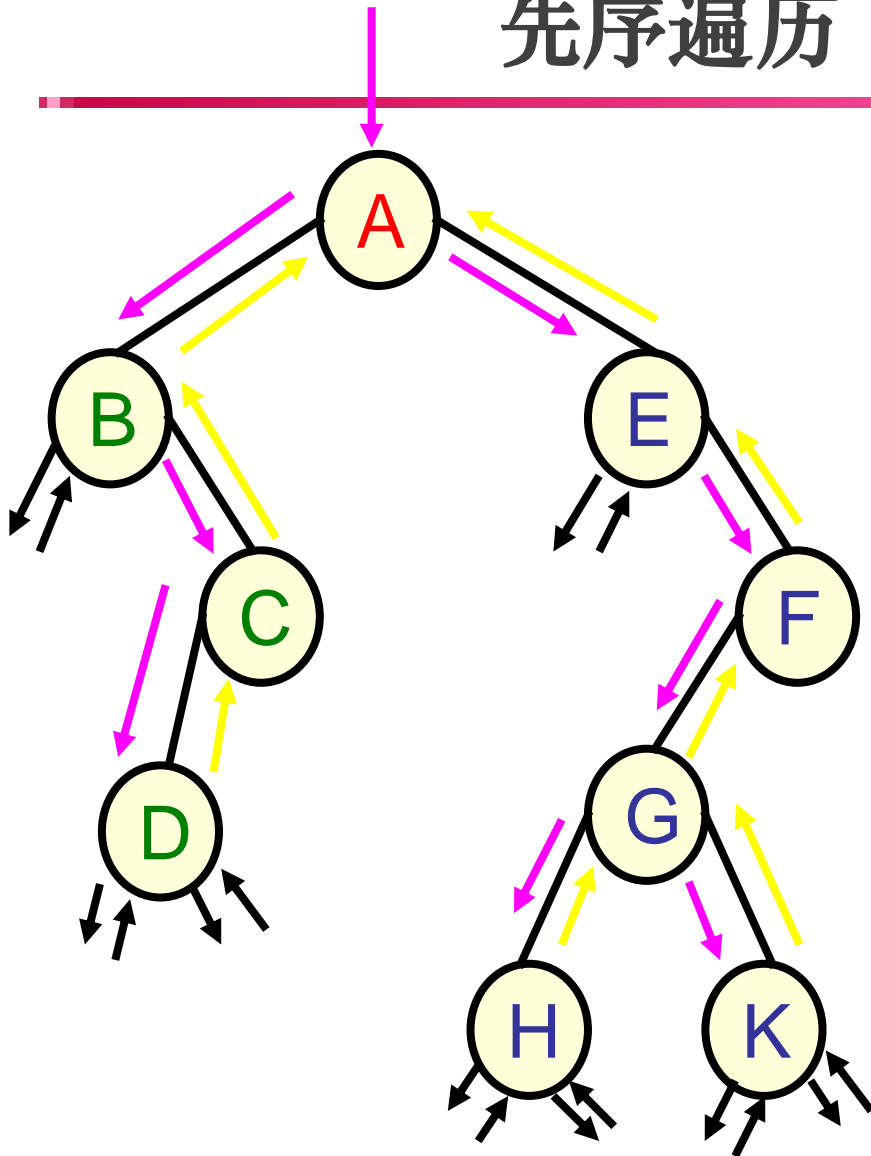
二叉树遍历非递归算法（先序2）

```
if (当前b树不空)
{
    p = b;
    while (栈不空或者p!=NULL)
    {
        while (p有左孩子)
        {
            访问p所指节点;将p进栈; p=p->lchild
        }
        if (栈不空)
        {
            出栈p;
            p = p->rchild;
        }
    }
}
```



code

先序遍历



^H

^K

^F

先序序列:

A B C D E F G H K

二叉树遍历非递归算法（先序2）

```
void PreOrder2(BTNode *b)
{
    BTNode *St[MaxSize], *p; int top=-1;
    p=b;
    while (top>-1 || p!=NULL)
    {
        while (p!=NULL)                                //扫描*p的所有左节点并进栈
        {
            printf("%c ", p->data); //访问之
            top++; St[top]=p;
            p=p->lchild;
        }
        *
        if (top>-1)
        {
            p=St[top]; top--;                            //出栈*p节点
            p=p->rchild;                                   //处理右子树
        }
    }
}
```

二叉树遍历非递归算法（中序）

■ 2. 中序遍历非递归算法

(2) 第二种方法（常规方法）

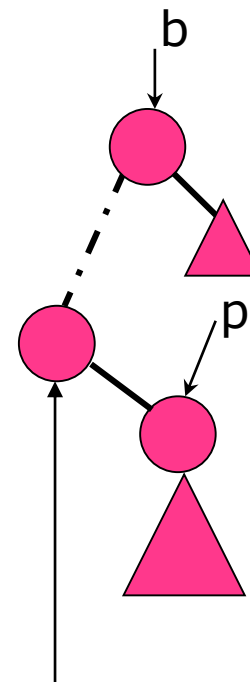
- 由中序遍历过程可知，采用一个栈保存需要返回的结点指针，先扫描（并非访问）根结点的所有左结点并将它们一一进栈。
- 然后出栈一个结点，显然该结点没有左孩子结点或者左孩子结点已访问过（进一步说明该结点的左子树均已访问），则访问它。然后扫描该结点的右孩子结点，将其进栈，再扫描该右孩子结点的所有左结点并一一进栈，如此这样，直到栈空为止。

二叉树遍历非递归算法（中序）

步骤如下：

左根右

```
if (当前b树不空)
{
    p = b;
    while (栈不空或者p!=NULL)
    {
        while (p有左孩子)
        {
            将p进栈;
            p = p->lchild;
        }
        if (栈不空)
        {
            出栈p并访问之;
            p = p->rchild;
        }
    }
}
```



b的最左下节点

■ 说明：

- (1) 所有左下孩子进栈，体现先访问左子树的特点。
- (2) 当所有左下孩子进栈后，栈顶节点p没有左孩子（也就是没有左子树）或者其左子树均已访问，所以可以访问p节点。
- (3) 当访问p节点后，转向其右孩子，采用同样的方式中序遍历右子树。

二叉树遍历非递归算法（中序）

```
void InOrder2(BTNode *b)
```

```
{    BTNode *St[MaxSize],*p; int top=-1;
```

```
    p=b;
```

```
    while (top>-1 || p!=NULL)
```

```
    {    while (p!=NULL)
```

//扫描*p的所有左结点并进栈

```
        {    top++; St[top]=p;
```

```
            p=p->lchild;
```

```
        }
```

```
        if (top>-1)
```

```
        {    p=St[top];top--;
```

//出栈*p结点

```
            printf("%c ",p->data);
```

//访问之

```
            p=p->rchild;
```

//扫描*p的右孩子结点

```
        }
```

```
    }
```

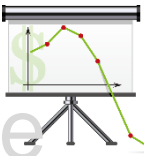
```
}
```

找*b的最左下
结点

二叉树遍历非递归算法（后序）

- 3. 后序遍历非递归算法
- (2) 第二种方法（常规方法）
 - 由后遍历过程可知，采用一个栈保存需要返回的结点指针，先扫描根结点的所有左结点并一一进栈，出栈一个结点**b*即当前结点，然后扫描该结点的右孩子结点并入栈，再扫描该右孩子结点的所有左结点并入栈。当一个结点的左右孩子结点均访问后再访问该结点，如此这样，直到栈空为止。

- 难点：如何判断一个结点*b的右孩子结点已访问过，为此用p保存刚刚访问过的结点（初值为NULL），若 $b \rightarrow rchild == p$ 成立（在后序遍历中，*b的右孩子结点一定刚好在*b之前访问），说明*b的左右子树均已访问，现在应访问*b。



code

```
void PostOrder2(BTNode *b)
{
    BTNode *St[MaxSize]; BTNode *p;
    int flag, top=-1;    // 栈指针置初值
    do
    {
        while (b!=NULL) // 将*b的所有左结点进栈
        {
            top++; St[top]=b;
            b=b->lchild;
        }
        p=NULL; // p指向栈顶结点的前一个已访问的结点
        flag=1; // 设置b的左孩子为已访问过
```

找最左下结点

```
while (top!= -1 && flag==1)
```

```
{   b=St[top];   //取出当前的栈顶元素
```

```
if (b->rchild==p)
```

b的右孩子不存在或已访问过

```
{   printf("%c ",b->data); //访问*b结点
```

```
    top--;p=b;    //p指向则被访问的结点
```

```
}
```

```
else
```

```
{   b=b->rchild;   //b指向右孩子结点
```

```
    flag=0; //设置b的左孩子未访问
```

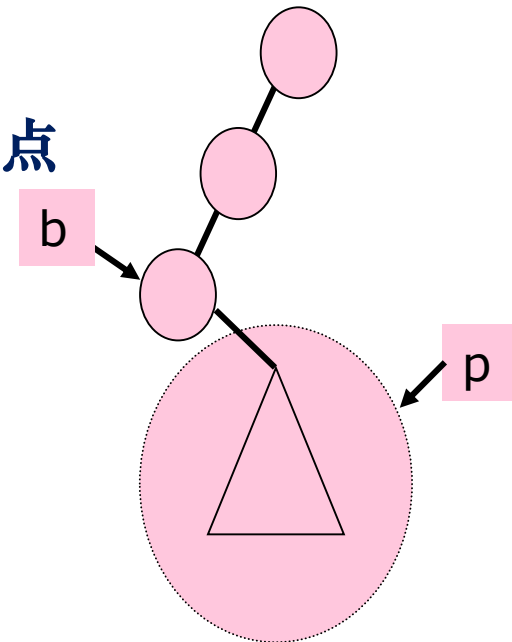
```
}
```

```
}
```

```
} while (top!= -1);
```

```
}
```

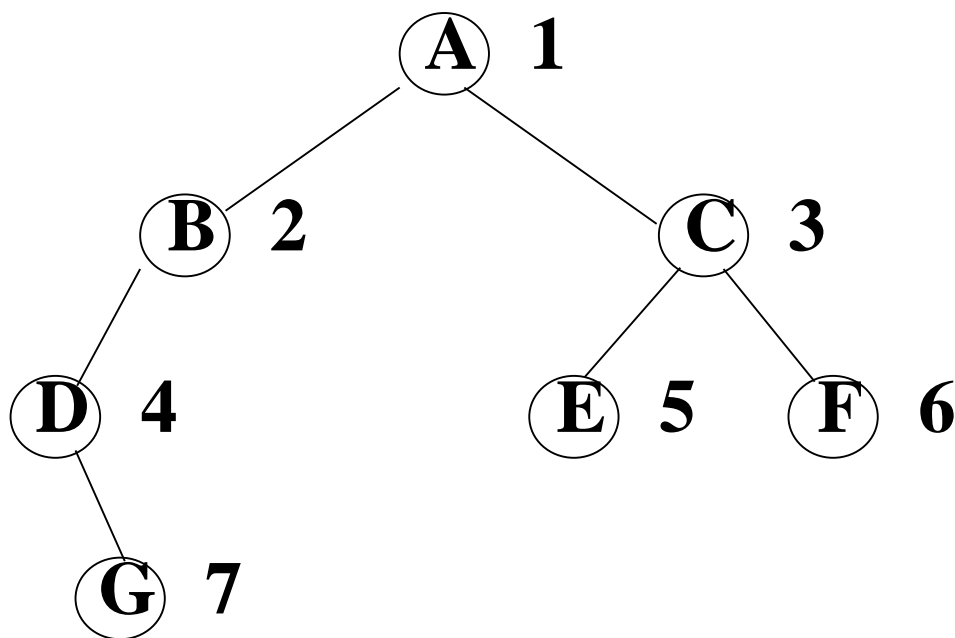
后序序列中最后访问的
结点是根结点，也可以
说一个结点的右孩子访
问了，则其左、右孩子
均已访问。



-
- 从上述过程可知，栈中保存的是当前结点*b的**所有祖先结点（均未访问过）**。
 - 例如，求一个结点的所有祖先结点。

遍历算法-----层次遍历

■ 层次遍历



ABCDEF G

遍历算法-----层次遍历

```
void LevelOrder(BTNode *b)
{
    BTNode *p;
    BTNode *qu[MaxSize]; //定义环形队列,存放节点指针
    int front,rear;       //定义队头和队尾指针
    front=rear=-1;        //置队列为空队列
    rear++;
    qu[rear]=b;            //根节点指针进入队列
    while (front!=rear)    //队列不为空
    {
        front=(front+1)%MaxSize;
        p=qu[front];       //队头出队列
        printf("%c ",p->data); //访问节点
        if (p->lchild!=NULL) //有左孩子时将其进队
        {
            rear=(rear+1)%MaxSize;
            qu[rear]=p->lchild;
        }
        if (p->rchild!=NULL) //有右孩子时将其进队
        {
            rear=(rear+1)%MaxSize;
            qu[rear]=p->rchild;
        }
    }
}
```

二叉树表达式

其先序序列为：

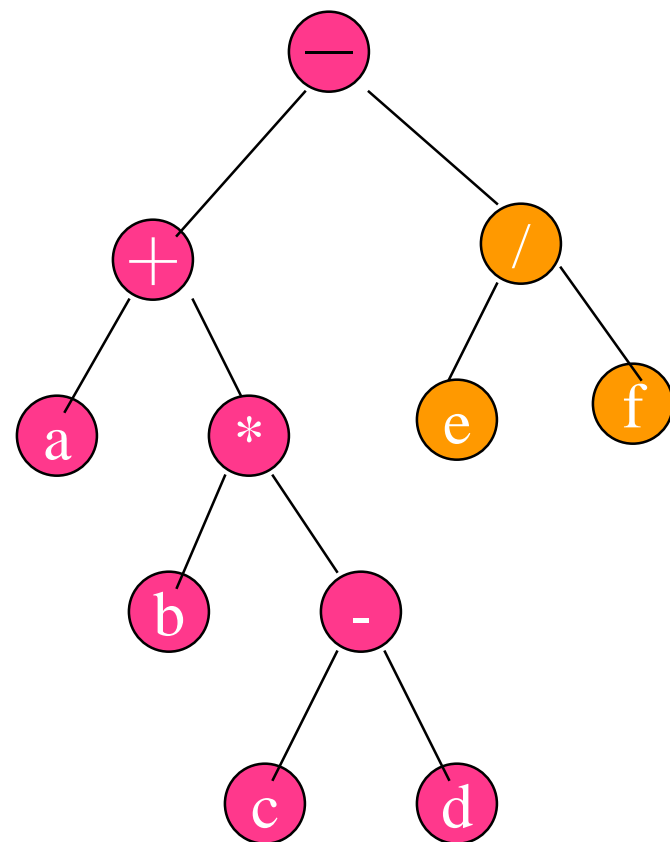
$-+a*b-cd/ef$

按中序遍历, 其中序序列为：

$a+b*c-d-e/f$

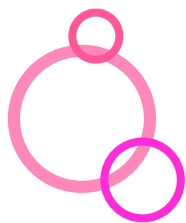
按后序遍历, 其后序序列为：

$abcd-*+ef/-$



中缀形式的算术

表达式为大家所熟悉, 但对于计算机, 使用后缀易于求值



05 二叉树的基本运算

shudedingyi

二叉树的基本运算及其实现

CONTENTS

01

二叉树的基本运算概述

02

二叉树的基本运算算法实现

二叉树的基本运算

- 归纳起来, 二叉树有以下基本运算:
- (1) 创建二叉树 `CreateBTNode(*b, *str)`:
 - 根据二叉树括号表示法的字符串 `*str` 生成对应的链式存储结构。
- (2) 查找结点 `FindNode(*b, x)`:
 - 在二叉树 `b` 中寻找 `data` 域值为 `x` 的结点, 并返回指向该结点的指针。
- (3) 找孩子结点 `LchildNode(p)` 和 `RchildNode(p)`
 - 分别求二叉树中结点 `*p` 的左孩子结点和右孩子结点。

二叉树的基本运算

- (4) 求高度BTNodeDepth(*b) :
 - 求二叉树b的高度。若二叉树为空, 则其高度为0; 否则, 其高度等于左子树与右子树中的最大高度加1。
- (5) 输出二叉树DispBTNode(*b)
 - 以括号表示法输出一棵二叉树。

二叉树的基本运算算法实现

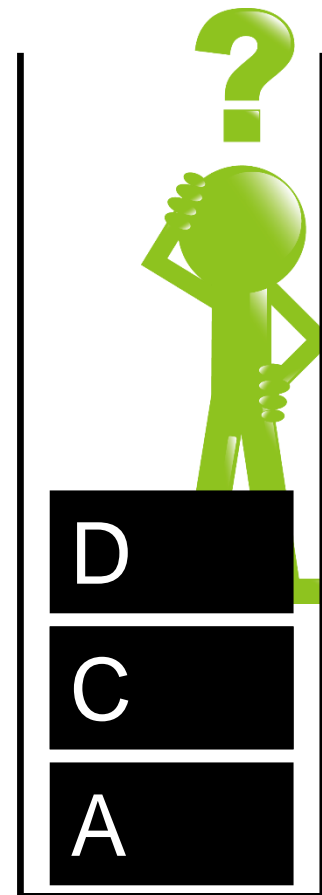
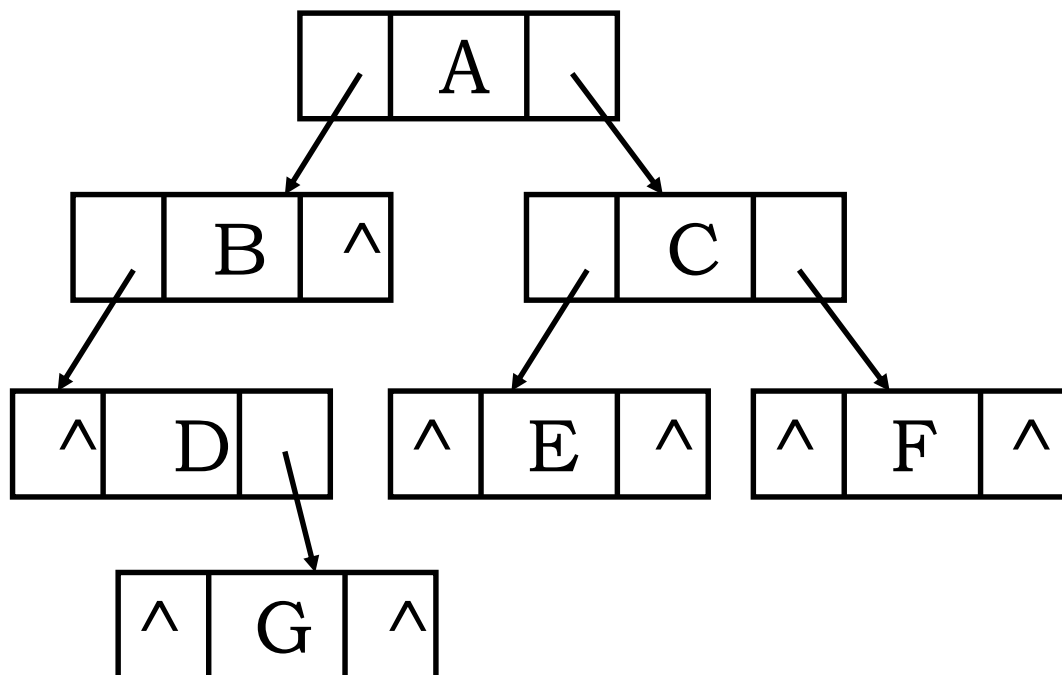
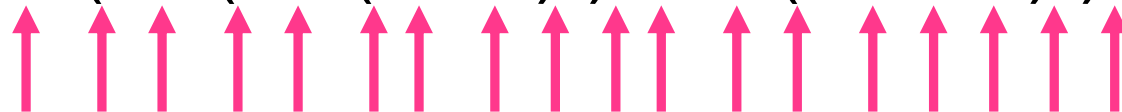
- (1) 创建二叉树CreateBTNode(*b, *str)
- 用ch扫描采用括号表示法表示二叉树的字符串。分以下几种情况：
 - ① 若ch='('：则将前面刚创建的结点作为双亲结点进栈, 并置k=1, 表示其后创建的结点将作为这个结点的左孩子结点;
 - ② 若ch=')'：表示栈中结点的左右孩子结点处理完毕, 退栈;
 - ③ 若ch=', '：表示其后创建的结点为右孩子结点;

二叉树的基本运算算法实现

- ④ 其他情况, 表示要创建一个结点, 并根据 k 值建立它与栈中结点之间的联系.
- 当 $k=1$ 时, 表示这个结点作为栈中结点的左孩子结点, 当
- $k=2$ 时, 表示这个结点作为栈中结点的右孩子结点。

根据括号表示法字符串构造二叉树

A (B (D (, G)) , C (E , F))



k= 2

二叉树的基本运算算法实现

```
void CreateBTNode(BTNode * &b,char *str)
{
    BTNode *St[MaxSize],*p=NULL;
    int top=-1,k,j=0;
    char ch;
    b=NULL;                                /*建立的二叉树初始时空*/
    ch=str[j];
    while (ch!='\0')                        /*str未扫描完时循环*/
    {
        switch(ch)
        {
            case '(':top++;St[top]=p;k=1; break; /*为左孩子*/
            case ')':top--;break;
            case ',':k=2; break;                /*为孩子结点右结点*/
        }
    }
}
```


二叉树的基本运算算法实现

```
default:p=(BTNode *)malloc(sizeof(BTNode));
```

```
p->data=ch;p->lchild=p->rchild=NULL;
```

```
if (b==NULL)                /**p为二叉树的根结点*/
```

```
    b=p;
```

```
else                          /*已建立二叉树根结点*/
```

```
{    switch(k)
```

```
{
```

```
    case 1:St[top]->lchild=p;break;
```

```
    case 2:St[top]->rchild=p;break;
```

```
}
```

```
}
```

```
}
```

```
j++;ch=str[j];
```

```
}
```

```
}
```

- (2) 采用先序遍历递归算法查找值为x的结点。找到后返回其指针, 否则返回NULL。算法如下:

```
BTNode *FindNode(BTNode *b, ElemType x)
{
    BTNode *p;
    if (b==NULL) return NULL;
    else if (b->data==x) return b;
    else
    {
        p=FindNode(b->lchild,x);
        if (p!=NULL) return p;
        else return FindNode(b->rchild,x);
    }
}
```

(3) 找孩子结点LchildNode(p)和RchildNode(p)
直接返回*p结点的左孩子结点或右孩子结点的指针。算法如下：

```
BTNode *LchildNode(BTNode *p)
{
    return p->lchild;
}
BTNode *RchildNode(BTNode *p)
{
    return p->rchild;
}
```

- (4) 求高度BTNodeDepth(*b)

求二叉树的高度的递归模型f()如下:

$$f(\text{NULL})=0$$

$$f(b)=\text{MAX}\{f(b\rightarrow\text{lchild}), f(b\rightarrow\text{rchild})\}+1 \quad \text{其他情况}$$

对应的算法如下:

```
int BTNodeDepth(BTNode *b)
{   int lchilddep,rchilddep;
    if (b==NULL) return(0);           /*空树的高度为0*/
    else
    {   lchilddep=BTNodeDepth(b->lchild);
        /*求左子树的高度为lchilddep*/
        rchilddep=BTNodeDepth(b->rchild);
        /*求右子树的高度为rchilddep*/
        return(lchilddep>rchilddep)?
            (lchilddep+1):(rchilddep+1);
    }
}
```

- (5) 输出二叉树DispBTNode(*b)

- 其过程是：对于非空二叉树b, 先输出其元素值, 当存在左孩子结点或右孩子结点时, 输出一个“(”符号, 然后递归处理左子树, 输出一个“, ”符号, 递归处理右子树, 最后输出一个“) ”符号。
- 对应的递归算法如下：

```
void DispBTNode(BTNode *b)
```

```
{ if (b!=NULL)
```

```
{   printf("%c",b->data);
```

```
    if (b->lchild!=NULL || b->rchild!=NULL)
```

```
    {   printf("(");
```

```
        DispBTNode(b->lchild);    /*递归处理左子树*/
```

```
        if (b->rchild!=NULL) printf(",");
```

```
        DispBTNode(b->rchild);    /*递归处理右子树*/
```

```
        printf(")");
```

```
    }
```

```
}
```

```
}
```

【例】

- 二叉链存储结构中, 设计一个算法判断两棵二叉树是否相似, 所谓二叉树 t_1 和 t_2 是相似的指的是 t_1 和 t_2 都是空的二叉树; 或者 t_1 和 t_2 的根结点是相似的, 以及 t_1 的左子树和 t_2 的左子树是相似的且 t_1 的右子树与 t_2 的右子树是相似的。
- 解: 判断两棵二叉树是否相似的递归模型 $f()$ 如下:
 - $f(t_1, t_2) = \text{true}$ 若 $t_1 = t_2 = \text{NULL}$
 - $f(t_1, t_2) = \text{false}$ 若 t_1 、 t_2 之一为 NULL , 另一不为 NULL
 - $f(t_1, t_2) = f(t_1 \rightarrow \text{lchild}, t_2 \rightarrow \text{lchild}) \quad \& \quad f(t_1 \rightarrow \text{rchild}, t_2 \rightarrow \text{rchild})$ 其他情况

对应的算法如下:

```
INT Like(BTNode *b1,BTNode *b2)
```

```
/*t1和t2两棵二叉树相似时返回1,否则返回0*/
```

```
{  int like1,like2;
```

```
    if (b1==NULL && b2==NULL) return 1;
```

```
    else if (b1==NULL || b2==NULL) return 0;
```

```
    else
```

```
    {   like1=Like(b1->lchild,b2->lchild);
```

```
        like2=Like(b1->rchild,b2->rchild);
```

```
        return (like1 & like2);    /*返回like1和like2的与*/
```

```
    }
```

```
}
```

【例】

- 假设二叉树采用二叉链存储结构, 设计一个算法Level()求二叉树中指定结点的层数。
- 解: 本题采用递归算法, 设h返回p所指结点的高度, 其初值为0。找到指定的结点时返回其层次; 否则返回0。lh作为一个中间变量在计算搜索层次时使用, 其初值为1。
- 对应的算法如下:

```
int Level(BTNode *b,ElemType x,int h)
```

```
/*找到*p结点后h为其层次,否则为0*/
```

```
{   if (b==NULL)   return(0);           /*空树时返回0*/
```

```
    else if (b->data==x) return(h);     /*找到结点p时*/
```

```
    else
```

```
    {   l=Level(b->lchild,x,h+1); /*在左子树中递归查找*/
```

```
        if (l!=0) /*左子树中未找到时在右子树中递归查  
找*/
```

```
        return(Level(b->rchild,x,h+1));
```

```
    else return(l);
```

```
    }
```

```
}
```

- 【例】假设二叉树采用二叉链存储结构, 设计一个算法输出从每个叶子结点到根结点的路径。
 - 解: 这里用层次遍历方法, 设计的队列为非循环顺序队列(类似于第3章3.2.4小节中求解迷宫问题时使用的队列)将所有已扫描过的结点指针进队, 并在队列中保存双亲结点的位置。当找到一个叶子结点时, 在队列中通过双亲结点的位置输出该叶子结点到根结点的路径。对应的算法如下:

```
void AllPath(BTNode *b)
```

```
{ struct snode
```

```
    { BTNode *node;                /*存放当前结点指针*/
```

```
        int parent;                /*存放双亲结点在队列中的位置*/
```

```
    } q[MaxSize];                  /*定义顺序队列*/
```

```
    int front,rear,p;              /*定义队头和队尾指针*/
```

```
    front=rear=-1;                 /*置队列为空队列*/
```

```
    rear++;q[rear].node=b;         /*根结点指针进入队列*/
```

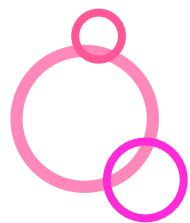
```
    q[rear].parent=-1;             /*根结点没有双亲结点*/
```

```
while (front<rear)                                /*队列不为空*/
{
    front++;
    b=q[front].node;                               /*队头出队列*/
    if (b->lchild==NULL && b->rchild==NULL)

        {   printf("%c到根结点路径:",b->data);
p=front;
while (q[p].parent!=-1)
{   printf("%c->",q[p].node->data);
    p=q[p].parent;   }
printf("%c\n",q[p].node->data);
```

```
}
```

```
if (b->lchild!=NULL)                /*左孩子结点入队列*/
{
    rear++; q[rear].node=b->lchild;
    q[rear].parent=front; }
    if (b->rchild!=NULL)            /*右孩子结点入队列*/
{
    rear++; q[rear].node=b->rchild;
    q[rear].parent=front; }
}                                    /*end of while*/
}
```



06

二叉树的构造

shudedingyi

二叉树的构造

- 同一棵二叉树具有惟一先序序列、中序序列和后序序列, 但不同的二叉树可能具有相同的先序序列、中序序列和后序序列。
- 显然, 仅由一个先序序列(或中序序列、后序序列), 无法确定这棵二叉树的树形。但是, 如果同时知道一棵二叉树的先序序列和中序序列, 或者同时知道中序序列和后序序列, 就能确定这棵二叉树。
- **【思考】**：给定先序、中序和后序遍历序列中任意两个, 是否可以唯一确定这棵二叉树的树形？

- 定理：任何 n ($n \geq 0$) 个不同结点的二叉树, 都可由它的中序序列和先序序列唯一地确定。

由遍历序列恢复二叉树

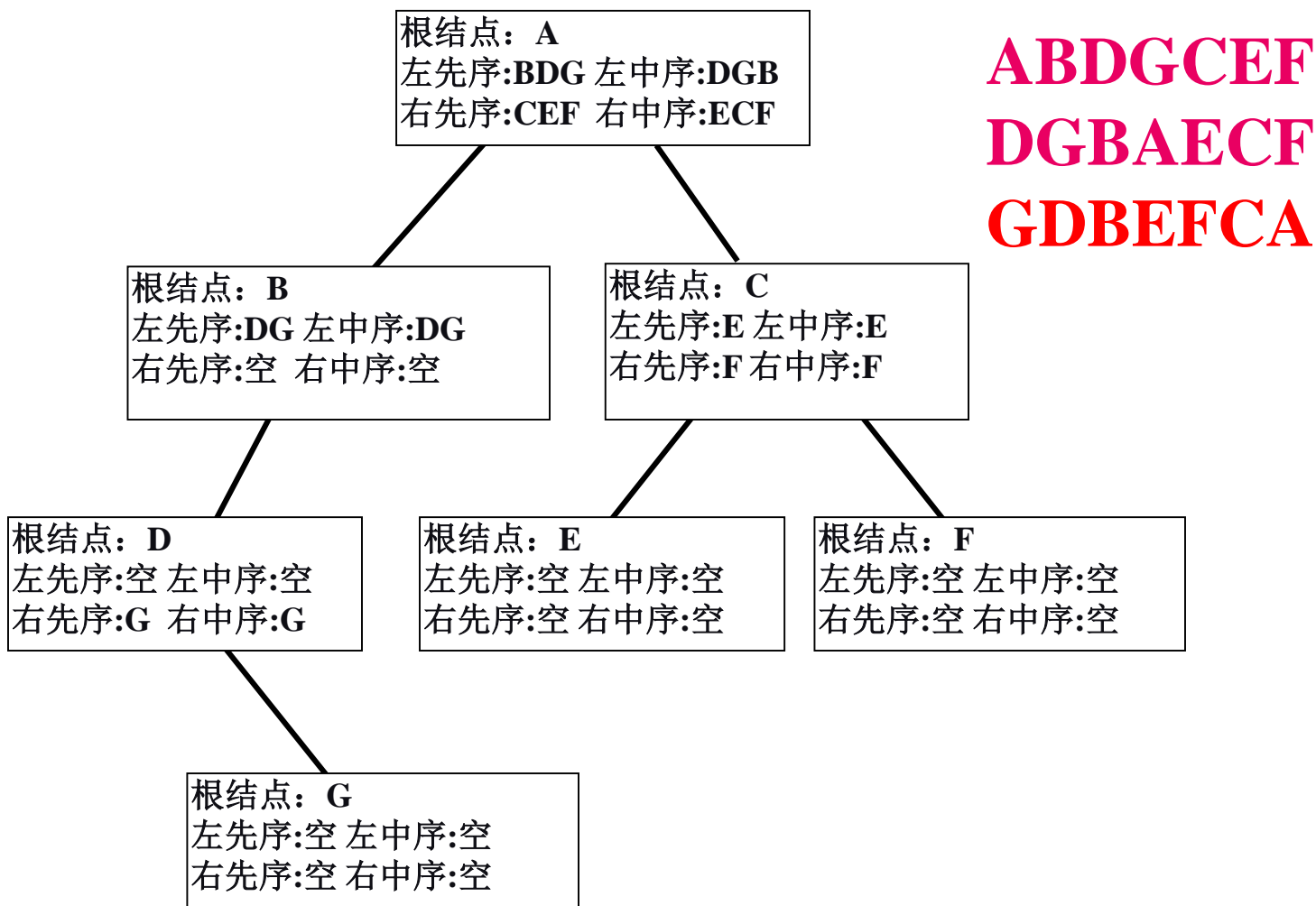
1. 若已知结点的先序序列和中序序列
2. 已知后序序列和中序序列

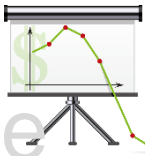
- 采用数学归纳法证明。
 - 当 $n=0$ 时, 二叉树为空, 结论正确。
 - 假设结点数小于 n 的任何二叉树, 都可以由其先序序列和中序序列唯一地确定。
 - 已知某棵二叉树具有 n ($n > 0$) 个不同结点, 其先序序列是 $a_0a_1\cdots a_{n-1}$; 中序序列是 $b_0b_1\cdots b_{k-1}b_kb_{k+1}\cdots b_{n-1}$ 。
 - 因为在先序遍历过程中, 访问根结点后, 紧跟着遍历左子树, 最后再遍历右子树。所以, a_0 必定是二叉树的根结点, 而且 a_0 必然在中序序列中出现。也就是说, 在中序序列中必有某个 b_k ($0 \leq k \leq n-1$)

- 由于 b_k 是根结点,而在中序遍历过程中,先遍历左子树,再访问根结点,最后再遍历右子树。所以在中序序列中, $b_0b_1\cdots b_{k-1}$ 必是根结点 b_k (也就是 a_0)左子树的中序序列,即 b_k 的左子树有 k 个结点(注意, $k=0$ 表示结点 b_k 没有左子树。)而 $b_{k+1}\cdots b_{n-1}$ 必是根结点 b_k (也就是 a_0)右子树的中序序列,即 b_k 的右子树有 $n-k-1$ 个结点(注意, $k=n-1$ 表示结点 b_k 没有右子树。))。
- 另外,在先序序列中,紧跟在根结点 a_0 之后的 k 个结点 $a_1\cdots a_k$ 就是左子树的先序序列, $a_{k+1}\cdots a_{n-1}$ 这 $n-k-1$ 就是右子树的先序序列。

- 根据归纳假设, 由于子先序序列 $a_1 \cdots a_k$ 和子中序序列 $b_0 b_1 \cdots b_{k-1}$ 可以唯一地确定根结点 a_0 的左子树, 而子先序序列 $a_{k+1} \cdots a_{n-1}$ 和子中序序列 $b_{k+1} \cdots b_{n-1}$ 可以唯一地确定根结点 a_0 的右子树。
- 综上所述, 这棵二叉树的根结点已经确定, 而且其左、右子树都唯一地确定了, 所以整个二叉树也就唯一地确定了。

【例如】 已知先序序列为ABDGCEF,中序序列为DGBAECF,则构造二叉树的过程如下所示





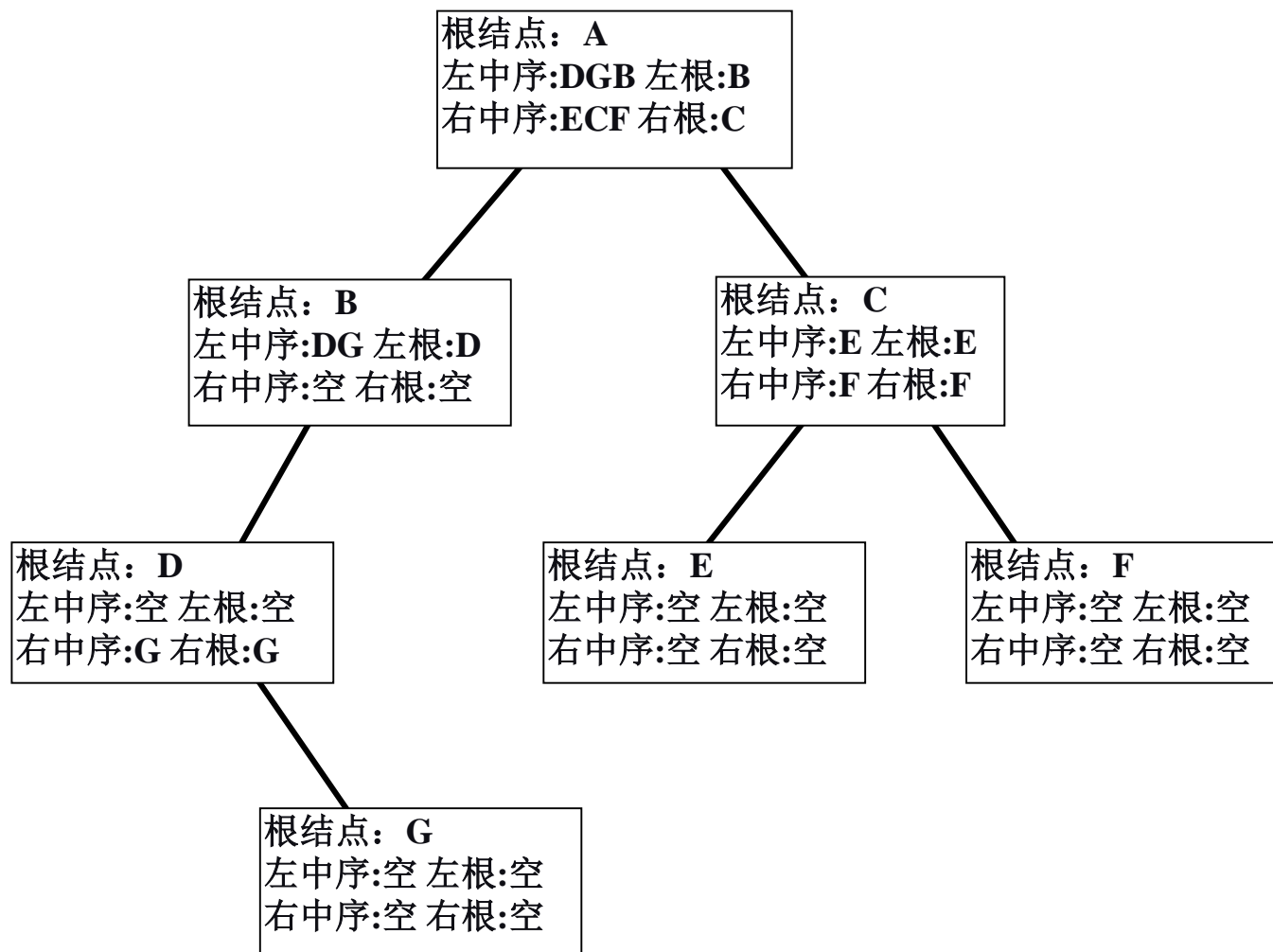
构造二叉树的算法

由上述定理得到以下构造二叉树的算法：

```
BTNode *CreateBT1(char *pre, char *in, int n)
{
    BTNode *s; char *p; int k;
    if (n <= 0) return NULL;
    s = (BTNode *) malloc(sizeof(BTNode)); /*创建结点*s*/
    s->data = *pre;
    for (p = in; p < in + n; p++) /*在中序中找为*ppos的位置k*/
        if (*p == *pre)
            break;
    k = p - in;
    s->lchild = CreateBT1(pre + 1, in, k); /*递归构造左子树*/
    s->rchild = CreateBT1(pre + k + 1, p + 1, n - k - 1); /*构造右子树*/
    return s;
}
```

- **【定理】**：任何 n ($n > 0$) 个不同结点的二叉树, 都可由它的中序序列和后序序列唯一地确定。
- 同样采用数学归纳法证明。
- 实际上, 对于根结点 a 的左右子树, 在确定左右子树的子中序序列后, 不需要确定左右子树的整个子后序序列, 只需确定子中序序列中全部字符在后序序列中最右边的那个字符即可, 因为这个字符就是子树的根结点。

【例】已知中序序列为DGBAECF,后序序列为GDBEFCA。对应的构造二叉树为:



构造二叉树的算法

```
BTNode *CreateBT2(char *post,char *in,int n,int m)
{
    BTNode *s;char *p,*q,*maxp;int maxpost,maxin,k;
    if (n<=0) return NULL;
    maxpost=-1;
    for (p=in;p<in+n;p++)          /*求in在最右边的那个字符*/
        for (q=post;q<post+m;q++) /*在in中用maxp指向这个字符,用
maxin标识它在in中的下标*/
            if (*p==*q)
            {
                k=q-post;
                if (k>maxpost)
                {
                    maxpost=k;  maxp=p; maxin=p-in; }
            }
}
```

构造二叉树的算法

```
s=(BTNode *)malloc(sizeof(BTNode));           /*创建二叉树结点*s*/
s->data=post[maxpost];
s->lchild=CreateBT2(post,in,maxin,m);          /*递归构造左子树*/
s->rchild=CreateBT2(post,maxp+1,n-maxin-1,m);
                                              /*递归构造右子树*/

return s;
}
```

二叉树计数问题：

具有 n 个节点的不同形态的二叉树的数目在一些涉及二叉树的平均情况复杂性分析中是很有用的。设 B_n 是含有 n 个节点的不同二叉树的数目。由于二叉树是递归地定义的，所以我们很自然地得到关于 B_n 的下面的递归方程：

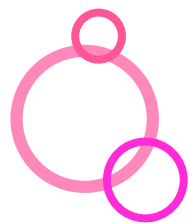
$$B_n = 1 \quad \text{当 } n=0$$

$$B_n = \sum_{i=0}^{n-1} B_i B_{n-i-1} \quad \text{当 } n>0$$

即一棵具有 $n>1$ 个节点的二叉树可以看成是由一个根节点、一棵具有 i 个节点的左子树和一棵具有 $n-i-1$ 个节点的右子树所组成。

$$\text{求得： } B_n = \frac{C_{2n}^n}{n+1}$$





07 线索二叉树

shudedingyi

线索二叉树

■ 线索二叉树的概念

- 对于具有 n 个结点的二叉树, 采用二叉链存储结构时, 每个结点有两个指针域, 总共有 $2n$ 个指针域, 又由于只有 $n-1$ 个结点被有效指针所指向(n 个结点中只有树根结点没有被有效指针域所指向), 则共有 $2n - (n-1) = n+1$ 个空链域,
- 遍历二叉树的结果是一个结点的线性序列。可以利用这些空链域存放指向结点的前驱和后继结点的指针。这样的指向该线性序列中的“前驱”和“后继”的指针, 称作**线索**。

线索二叉树

- 在结点的存储结构上增加两个标志位来区分：
左标志ltag : 0 表示lchild指向左孩子结点
 1 表示lchild指向前驱结点
右标志rtag : 0 表示rchild指向右孩子结点
 1 表示rchild指向后继结点

这样, 每个结点的存储结构如下:

ltag	lchild	data	rchild	rtag
------	--------	------	--------	------

线索二叉树

- 按上述原则在二叉树的每个结点上加上线索的二叉树称作线索二叉树。对二叉树以某种方式遍历使其变为线索二叉树的过程称作按该方式对二叉树进行线索化。

线索二叉树

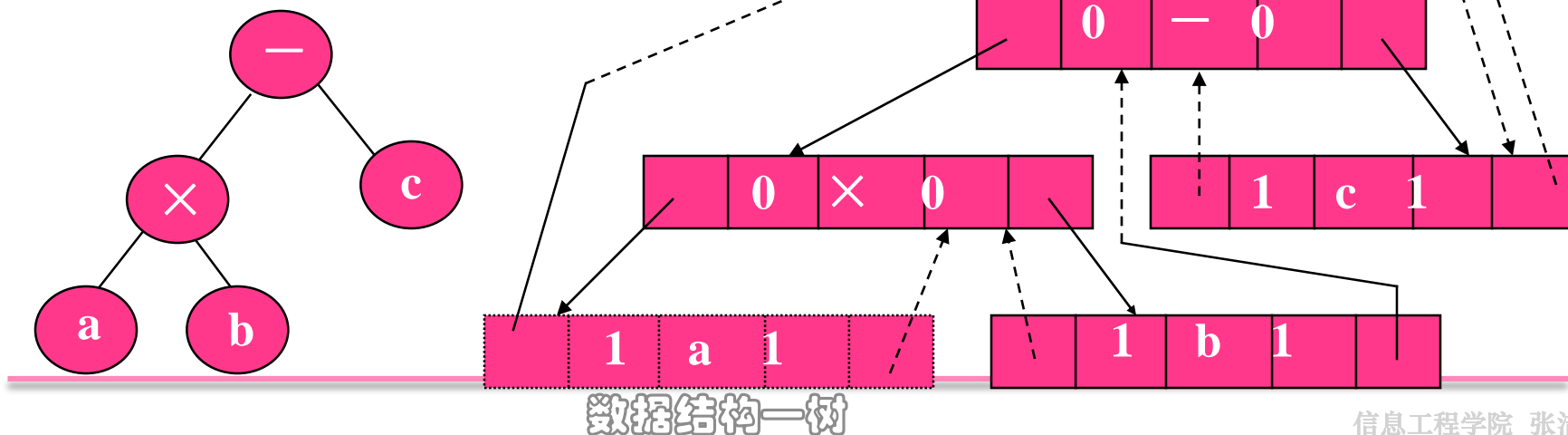
- 称这种结点结构为**线索链表**；
- 其中指示前驱和后继的链域称为**线索**；
- 加上线索的二叉树称为**线索二叉树**；
- 对二叉树以某种次序遍历使其变为线索二叉树的过程称为**线索化**。
- 按中序遍历得到的线索二叉树称为**中序线索二叉树**；
按先序遍历得到的线索二叉树称为**先序线索二叉树**；
按后序遍历得到的线索二叉树称为**后序线索二叉树**；

线索二叉树

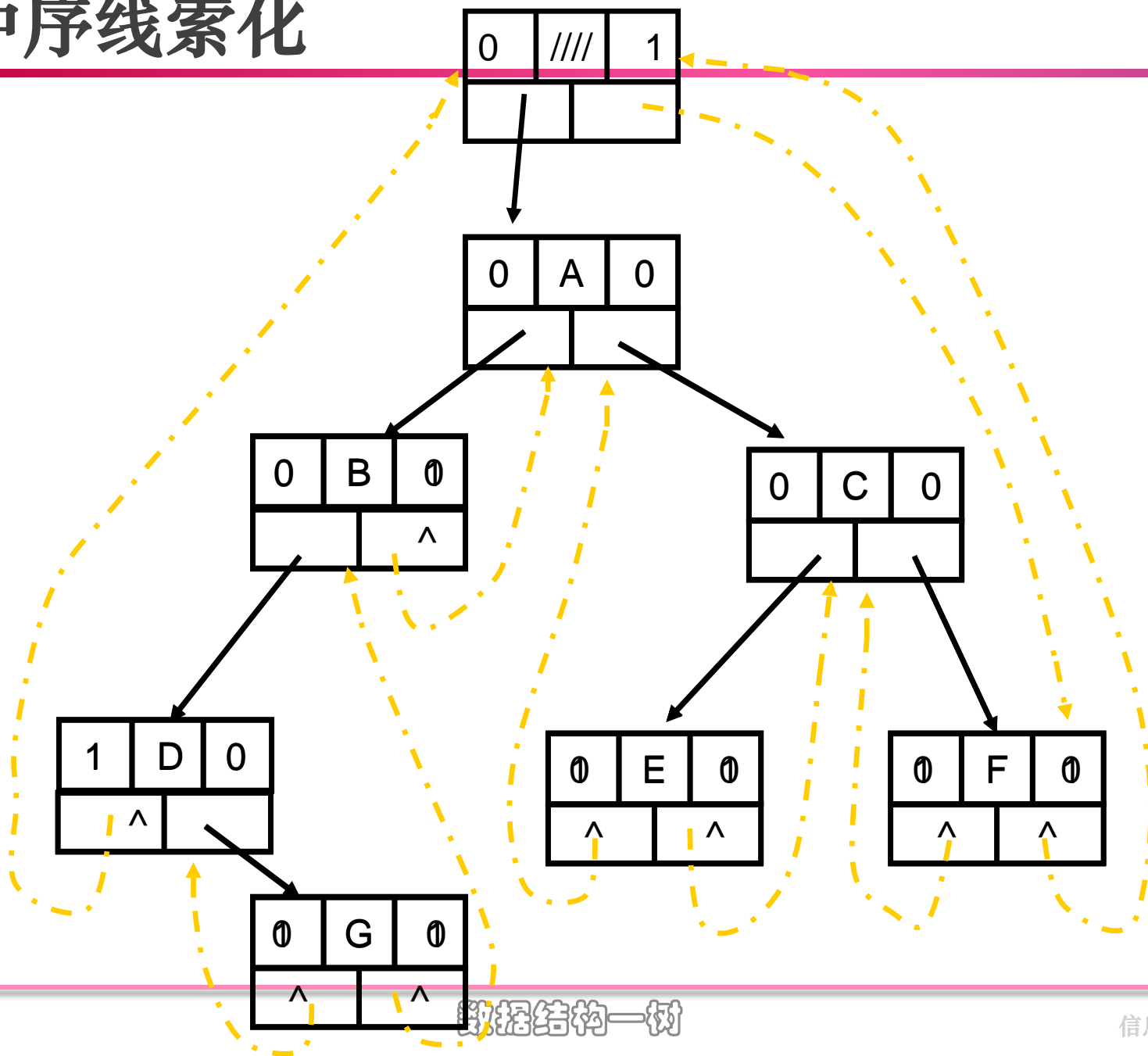
整体结构

- 增设一个头结点，令其lchild指向二叉树的根结点，ltag=0、rtag=1；
- 并将该结点作为遍历访问的第一个结点的前驱和最后一个结点的后继；
- 最后用头指针指示该头结点。

A*b-c



中序线索化



线索二叉树

- 为使算法设计方便, 在线索二叉树中再增加一个头结点。头结点的data域为空; lchild指向无线索时的根结点, ltag为0; rchild指向按某种方式遍历二叉树时的最后一个结点, rtag为1。
- 图中实线表示二叉树原来指针所指的结点, 虚线表示线索二叉树所添加的线索。

线索化二叉树

- 建立线索二叉树, 或者说, 对二叉树线索化, 实质上就是遍历一棵二叉树。
 - 在遍历的过程中, 检查当前结点的左、右指针域是否为空。如果为空, 将它们改为指向前驱结点或后继结点的线索。另外, 在对一棵二叉树添加线索时, 我们创建一个头结点, 并建立头结点与二叉树的根结点的线索。对二叉树线索化后, 还须建立最后一个结点与头结点之间的线索。
- 下面以中序线索二叉树为例, 讨论建立线索二叉树的算法。

线索化二叉树

- 为了实现线索化二叉树, 将前面二叉树结点的类型定义修改如下:

```
typedef struct node
{
    ElemType data;                /*结点数据域*/
    int ltag, rtag;               /*增加的线索标记*/
    struct node *lchild;         /*左孩子或线索指针*/
    struct node *rchild;         /*右孩子或线索指针*/
} TBTNode;                      /*线索树结点类型定义*/
```

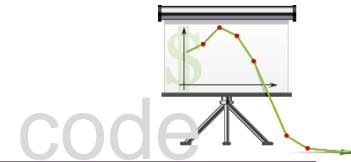
线索化二叉树

- 下面是建立中序线索二叉树的算法。CreaThread(b) 算法是将以二叉链存储的二叉树b进行中序线索化, 并返回线索化后头结点的指针root。Thread(p) 算法用于对于以*p为根结点的二叉树中序线索化。在整个算法中p总是指向当前被线索化的结点, 而pre作为全局变量, 指向刚刚访问过的结点, *pre是*p的前驱结点, *p是*pre的后继结点。

线索化二叉树

- CreaThread(b) 算法思路是：先创建头结点*root, 其lchild域为线索, rchild域为链指针。将rchild指针指向*b, 如果b二叉树为空, 则将其lchild指向自身。否则将*root的lchild指向*b结点, p指向*b结点, pre指向*root结点。再调用Thread(b)对整个二叉树线索化。最后加入指向头结点的线索, 并将头结点的rchild指针域线索化为指向最后一个结点(由于线索化直到p等于NULL为止, 所以最后一个结点为*pre)。

线索化二叉树

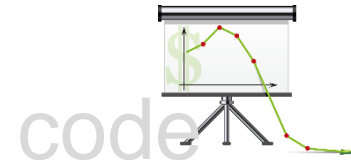


```
TBTNode *CreaThread(TBTNode *b)    /*中序线索化二叉树*/
{
    TBTNode *root;
    root=(TBTNode *)malloc(sizeof(TBTNode));
    root->ltag=0;root->rtag=1; root->rchild=b;
    if (b==NULL) root->lchild=root;    /*空二叉树*/
    else
    {
        root->lchild=b;
        pre=root;                    /*pre是*p的前驱结点,供加线索用*/
        Thread(b);                  /*中序遍历线索化二叉树*/
        pre->rchild=root;          /*加入指向头结点的线索*/
        pre->rtag=1;
        root->rchild=pre;          /*头结点右线索化*/
    }
    return root;
}
```

线索化二叉树

- Thread(p) 算法思路是：类似于中序遍历的递归算法，在p指针不为NULL时，先对*p结点的左子树线索化；若*p结点没有左孩子结点，则将其lchild指针线索化为指向其前驱结点*pre，否则表示lchild指向其左孩子结点，将其ltag置为1；若*pre结点的rchild指针为NULL，将其rchild指针线索化为指向其后继结点*p，否则rchild表示指向其右孩子结点，将其rtag置为1，再将pre指向*p；最后对*p结点的右子树线索化。

线索化二叉树



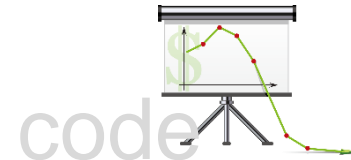
```
TBTNode *pre;                                /*全局变量*/
void Thread(TBTNode *&p)  /*对二叉树b进行中序线索化*/
{  if (p!=NULL)
    {  Thread(p->lchild);                      /*左子树线索化*/
        if (p->lchild==NULL)                  /*前驱线索*/
        {  p->lchild=pre; p->ltag=1;}  /*建立结点的前驱线索*/
        else  p->ltag=0;
        if (pre->rchild==NULL)                /*后继线索*/
        {  pre->rchild=p;pre->rtag=1;}/*建立前驱的后继线索*/
        else  pre->rtag=0;
        pre=p;
        Thread(p->rchild);                    /*递归调用右子树线索化*/
    }
}
```

中序遍历(递归)

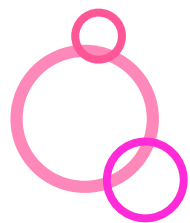
遍历线索化二叉树

- 遍历某种次序的线索二叉树, 就是从该次序下的开始结点出发, 反复找到该结点在该次序下的后继结点, 直到终端结点。
- 在中序线索二叉树中, 开始结点就是根结点的最左下结点, 而求当前结点在中序序列下的后继和前驱结点的方法如教材中表7. 2所示, 最后一个结点的rchild指针被线索化为指向头结点。利用这些条件, 在中序线索化二叉树中实现中序遍历的算法如下:

线索化二叉树



```
void ThInOrder(TBTNode *tb)
{   TBTNode *p=tb->lchild;           /*p指向根结点*/
    while (p!=tb)
    {   while (p->ltag==0) p=p->lchild; /*找开始结点*/
        printf("%c",p->data);          /*访问开始结点*/
        while (p->rtag==1 && p->rchild!=tb)
        {   p=p->rchild;
            printf("%c",p->data);
        }
        p=p->rchild;
    }
}
```



08

哈夫曼树

shudedingyi

哈夫曼树

CONTENTS

01 哈夫曼树的定义

02 构造哈夫曼树

03 哈夫曼编码

哈夫曼树的意义：

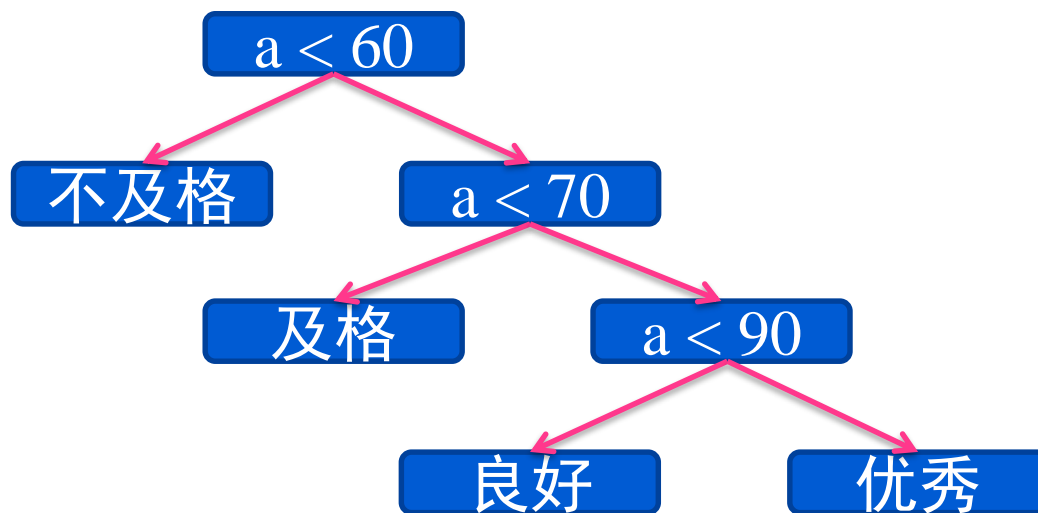
- 在数据膨胀、信息爆炸的今天，数据压缩的意义不言而喻。谈到数据压缩，就不能不提哈夫曼（Huffman）编码，哈夫曼编码是首个实用的压缩编码方案，即使在今天的许多知名压缩算法里，依然可以见到哈夫曼编码的影子。
- 在数据通信中，用二进制给每个字符进行编码时不得不面对的一个问题是如何使电文总长最短且不产生二义性。根据字符出现频率，利用哈夫曼编码可以构造出一种不等长的二进制，使编码后的电文长度最短，且保证不产生二义性。

- 以下这段代码在效率上有什么问题？

```
if( a < 60 )  
    printf( “不及格” );  
else if( a < 70 )  
    printf( “及格” );  
else if( a < 90 )  
    printf( “良好” );  
else  
    printf( “优秀” );
```



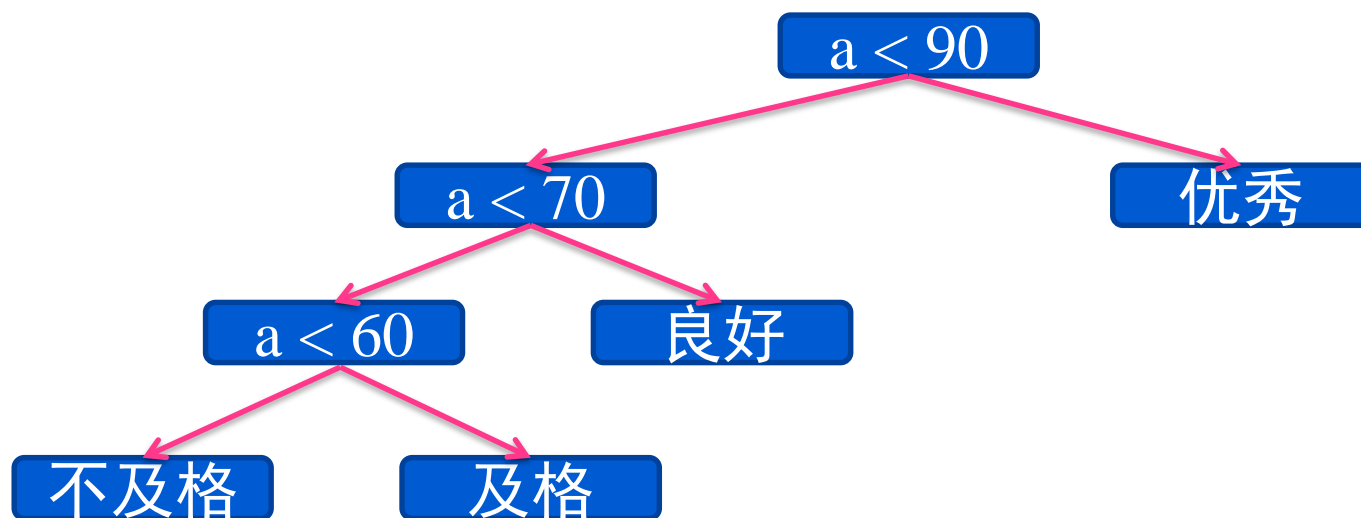
哈夫曼树



分数	0~59	60~69	70~89	90~100
所占比例	5%	15%	70%	10%

哈夫曼树

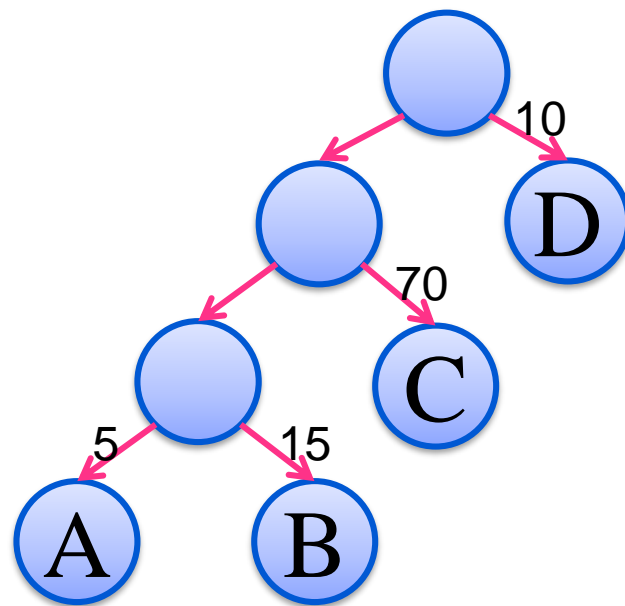
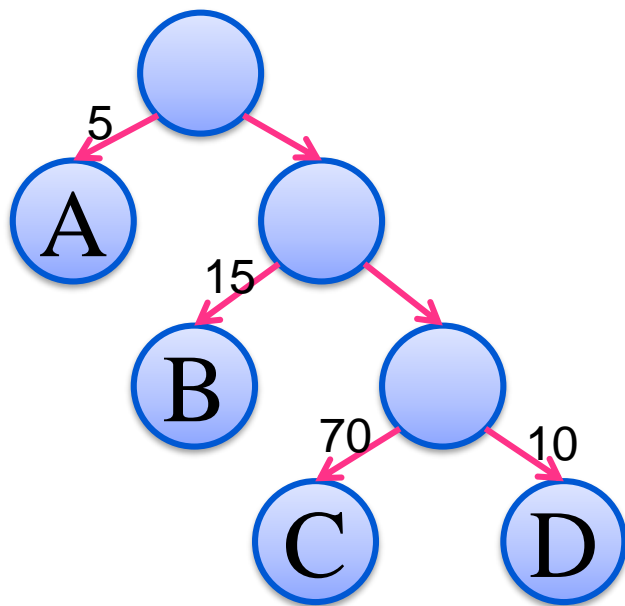
- 若判断流程改为以下：



- 效果可能有明显的改善

哈夫曼树定义与原理

- 先把这两棵二叉树简化成叶子结点带权的二叉树（注：树结点间的连线相关的数叫做权，**Weight**）。



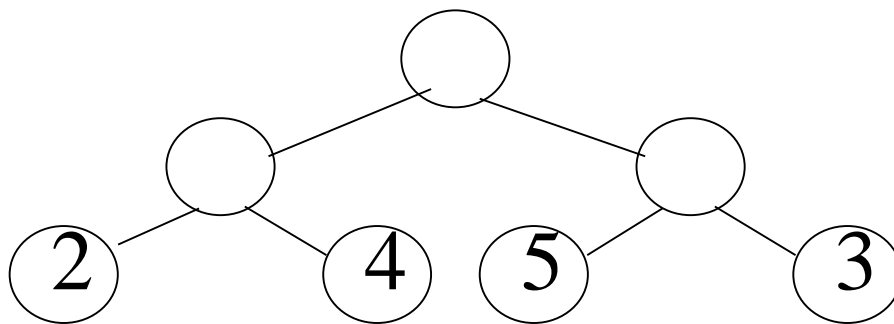
哈夫曼树的定义

- 设二叉树具有n个带权值的叶子结点, 那么从根结点到各个叶子结点的路径长度与相应结点权值的乘积的和, 叫做二叉树的带权路径长度。

$$WPL = \sum_{i=1}^n w_i l_i$$

- 具有最小带权路径长度的二叉树称为哈夫曼树。

哈夫曼树的定义



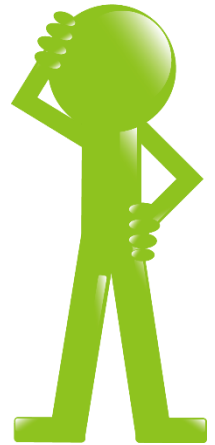
一个带权二叉树

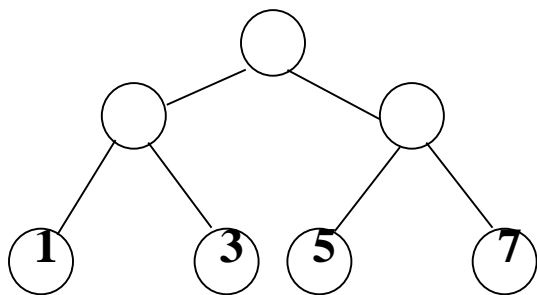
$$WPL = 2 \times 2 + 4 \times 2 + 5 \times 2 + 3 \times 2 = 28$$

-
- 若给出一组具有确定权值的叶结点，请问其带权路径长度的概念。

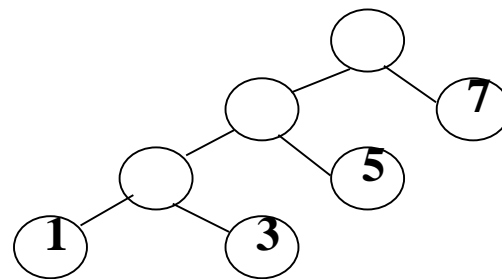
- 例如：

给出4个叶结点，设其权值分别为1，3，5，7

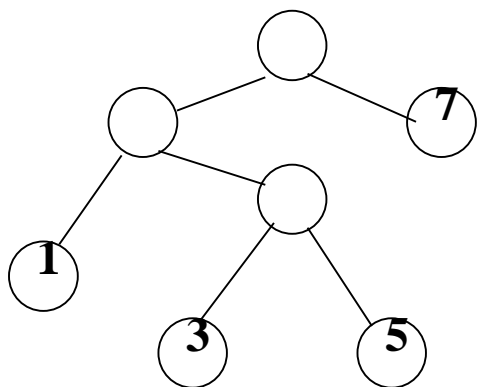




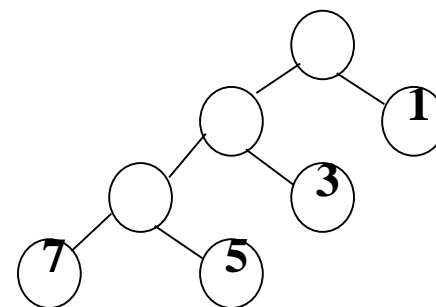
(a) $\text{WPL} = 1 \times 2 + 3 \times 2 + 5 \times 2 + 7 \times 2 = 32$



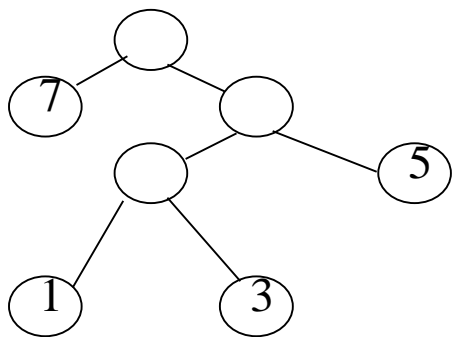
(b) $\text{WPL} = 1 \times 3 + 3 \times 3 + 5 \times 2 + 7 \times 1 = 29$



(c) $\text{WPL} = 1 \times 2 + 3 \times 3 + 5 \times 3 + 7 \times 1 = 33$

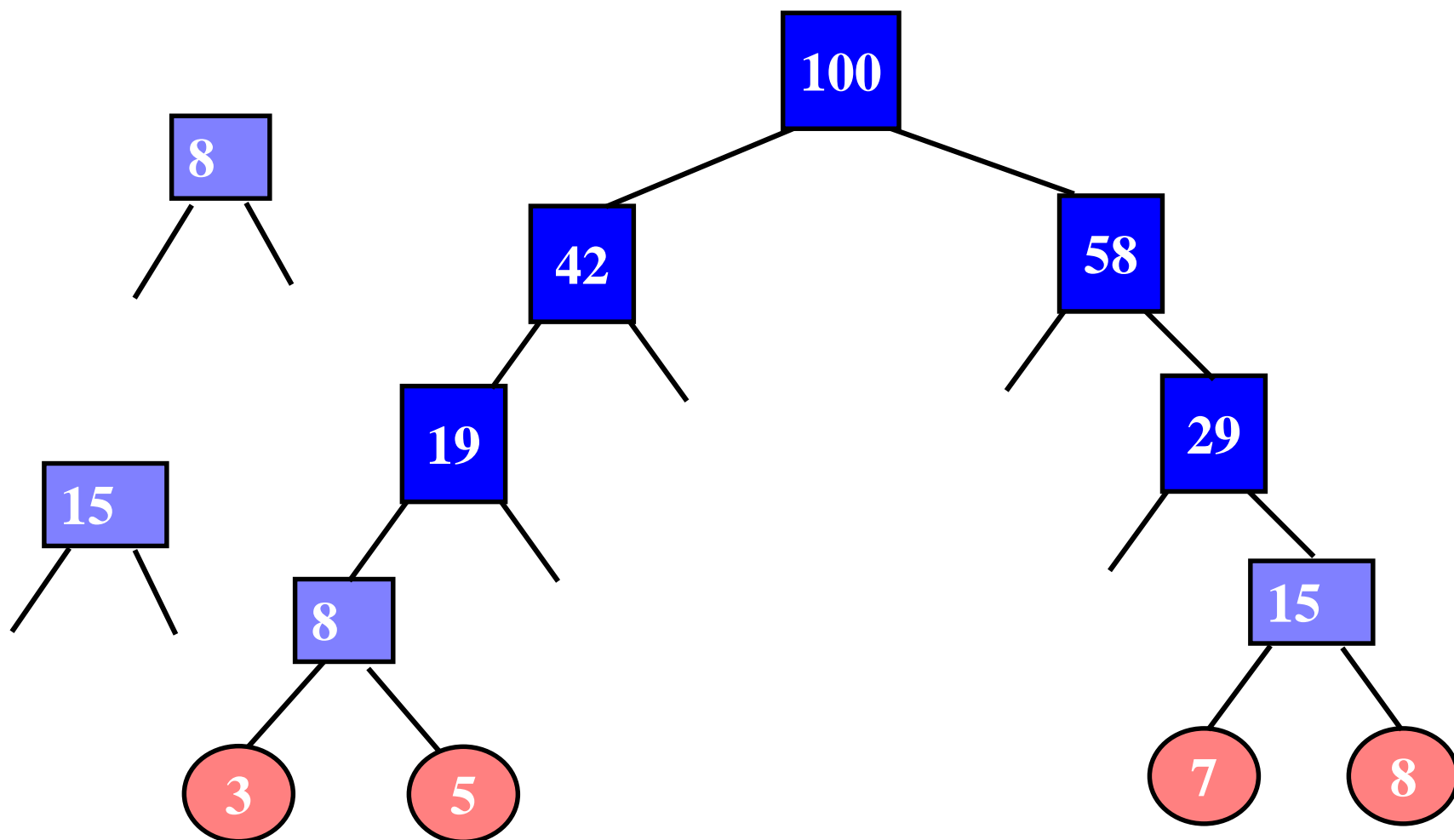
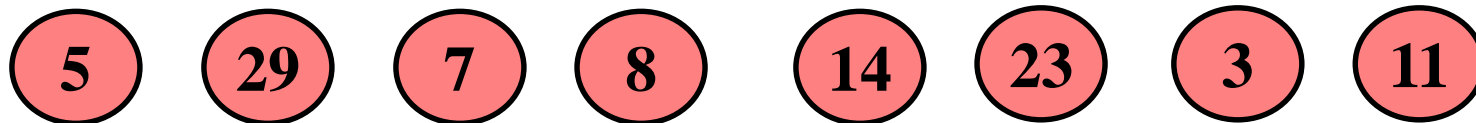


(d) $\text{WPL} = 7 \times 3 + 5 \times 3 + 3 \times 2 + 1 \times 1 = 43$



(e) $\text{WPL} = 7 \times 1 + 5 \times 2 + 3 \times 3 + 1 \times 3 = 29$

$W = \{ 0.05, 0.29, 0.07, 0.08, 0.14, 0.23, 0.03, 0.11 \}$



思考：哈夫曼树是否是唯一的？

完全二叉树是路径长度最短的二叉树。

- (1) 完全二叉树并不一定是Huffman树；
- (2) HT是权值越大的越靠近根结点；
- (3) HT不唯一，但WPL一定相等。



HT不唯一性, 可能出现在:

- (1) 构造新树时, 左、右孩子未作规定。
- (2) 当有多个权值相同的树, 可作有候选树时, 选择谁未作规定。

哈夫曼树的构造方法

由相同权值的一组叶子结点所构成的二叉树有不同的形态和不同的带权路径长度。

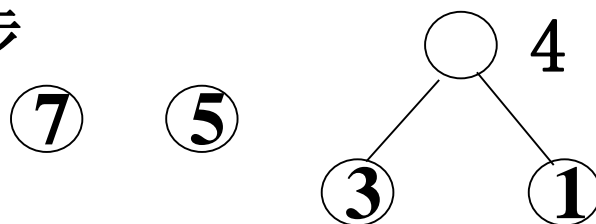
- (1) n 个权值 $\{W_1, W_2, \dots, W_n\}$ 构造 n 棵只有一个叶结点的二叉树作为集合 F ；
- (2) 选取根结点的权值最小和次小的两棵二叉树作为左、右子树构造一棵新的二叉树，这棵新的二叉树根结点的权值为其左、右子树根结点权值之和；
- (3) 删除作为左、右子树的两棵二叉树，并将新建立的二叉树加入到集合 F 中；
- (4) 重复 (2) (3) 两步，当 F 中只剩下一棵二叉树时，这棵二叉树便是所要建立的哈夫曼树。

哈夫曼树的构造方法

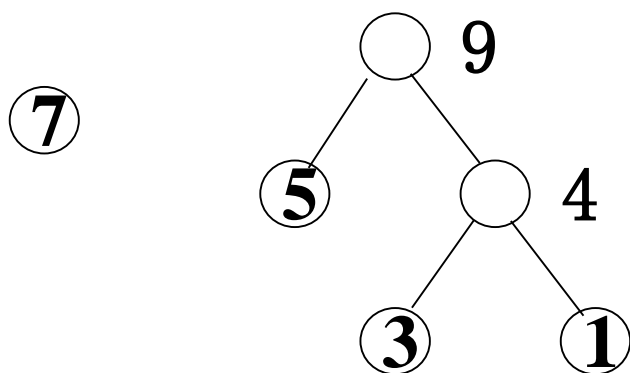
第一步



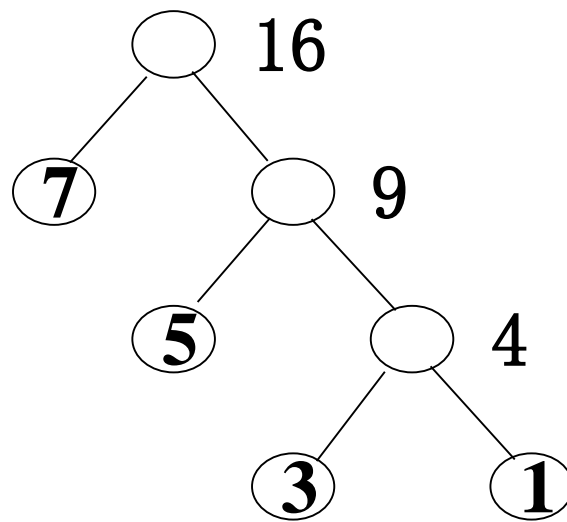
第二步



第三步



第四步



哈夫曼树的构造算法

- 用`ht[]`数组存放哈夫曼树, 对于具有 n 个叶子结点的哈夫曼树, 总共有 $2n-1$ 个结点。
- 【算法思路】：
 - n 个叶子结点只有`data`和`weight`域值, 先将所有 $2n-1$ 个结点的`parent`、`lchild`和`rchild`域置为初值-1。处理每个非叶子结点`ht[i]` (存放在`ht[n] ~ ht[2n-2]`中): 从`ht[0] ~ ht[i-2]`中找出根结点(即其`parent`域为-1)最小的两个结点`ht[lnode]`和`ht[rnode]`, 将它们作为`ht[i]`的左右子树, `ht[lnode]`和`ht[rnode]`的双亲结点置为`ht[i]`, 并且 $ht[i].weight = ht[lnode].weight + ht[rnode].weight$ 。如此这样直到所有 $2n-1$ 个非叶子结点处理完毕。
- 构造哈夫曼树的算法如下:

ht[]数组在构造哈夫曼树过程中的变化

No	weig	pare	lchi	rchi
0	5	-3	-1	-1
1	29	13	-1	-1
2	7	-9	-1	-1
3	8	-9	-1	-1
4	14	11	-1	-1
5	23	12	-1	-1
6	3	-3	-1	-1
7	11	10	-1	-1

No	weig	pare	lchi	rchi
8	8	10	7	1
9	15	11	3	4
10	19	12	8	7
11	29	13	4	9
12	42	14	10	5
13	58	14	1	11
14	100	-1	12	13

哈夫曼树的构造算法

- 在构造哈夫曼树时，可以设置一个结构数组HuffNode保存哈夫曼树中各结点的信息，根据二叉树的性质可知，具有 n 个叶子结点的哈夫曼树共有 $2n - 1$ 个结点，所以数组HuffNode的大小设置为 $2n - 1$ ，数组元素的结构形式如下：

weight	lchild	rchild	parent
---------------	---------------	---------------	---------------

- 其中，weight域保存结点的权值，lchild和rchild域分别保存该结点的左、右孩子结点在数组HuffNode中的序号，从而建立起结点之间的关系。

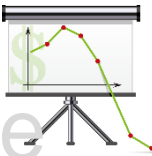
- 用ht[]数组存放哈夫曼树，对于具有n个叶子节点的哈夫曼树，总共有 $2n-1$ 个节点。树中每个节点结构如下：

- `typedef struct`
- `{ char data; //节点值`
- `float weight; //权重`
- `int parent; //双亲节点`
- `int lchild; //左孩子节点`
- `int rchild; //右孩子节点`
- `} HTNode;`



code

```
void CreateHT(HTNode ht[],int n)
{   int i,j,k,lnode,rnode;           float min1,min2;
    for (i=0;i<2*n-1;i++)           /*所有结点的相关域置初值-1*/
        ht[i].parent=ht[i].lchild=ht[i].rchild=-1;
    for (i=n;i<2*n-1;i++)           /*构造哈夫曼树*/
    {   min1=min2=32767; lnode=rnode=-1;
        for (k=0;k<=i-1;k++)
            if (ht[k].parent==-1) /*未构造二叉树的结点中
查找*/
```





```
{   if (ht[k].weight<min1)
        {   min2=min1;rnode=lnode;
            min1=ht[k].weight;lnode=k; }
        else if (ht[k].weight<min2)
            {   min2=ht[k].weight;rnode=k; }
        }                                     /*if*/
        ht[lnode].parent=i;ht[rnode].parent=i;

ht[i].weight=ht[lnode].weight+ht[rnode].weight;
        ht[i].lchild=lnode;ht[i].rchild=rnode;
    }
}
```

哈夫曼编码（ Huffman 树的应用）

1、问题的提出

通讯中常需要将文字转换成二进制字符串电文进行传送。文字  电文，称为编码。

反之，收到电文后要将电文转换成原来的文字，
电文  文字，称为译码。

哈夫曼树在编码问题中的应用

- 例如，假设要传送的电文为ABACCCDA，电文中只含有A，B，C，D四种字符，

表 :字符的四种不同的编码方案

字符	编码	字符	编码	字符	编码	字符	编码
A	000	A	00	A	0	A	01
B	010	B	01	B	110	B	010
C	100	C	10	C	10	C	001
D	111	D	11	D	111	D	10
(a)		(b)		(c)		(d)	

哈夫曼树及其应用

例如：需将文字“ABACCD A”转换成电文。文之中有四种字符，用2位二进制便可分辨。

编码方案1:

A	B	C	D
00	01	10	11

则上述文字的电文为：00010010101100 共14位。

译码时，只需每2位一译即可。

特点：等长等频率编码，译码容易，但电文不一定最短。

哈夫曼树及其应用

编码方案2:

A	B	C	D
0	00	1	01

采用不等长编码，让出现次数多的字符用短码。

则上述文字的电文为：000011010 共9位。

但无法译码，它即可译为BBCCACA，也可译为AAAACCCDA等。

哈夫曼树及其应用

编码方案3:

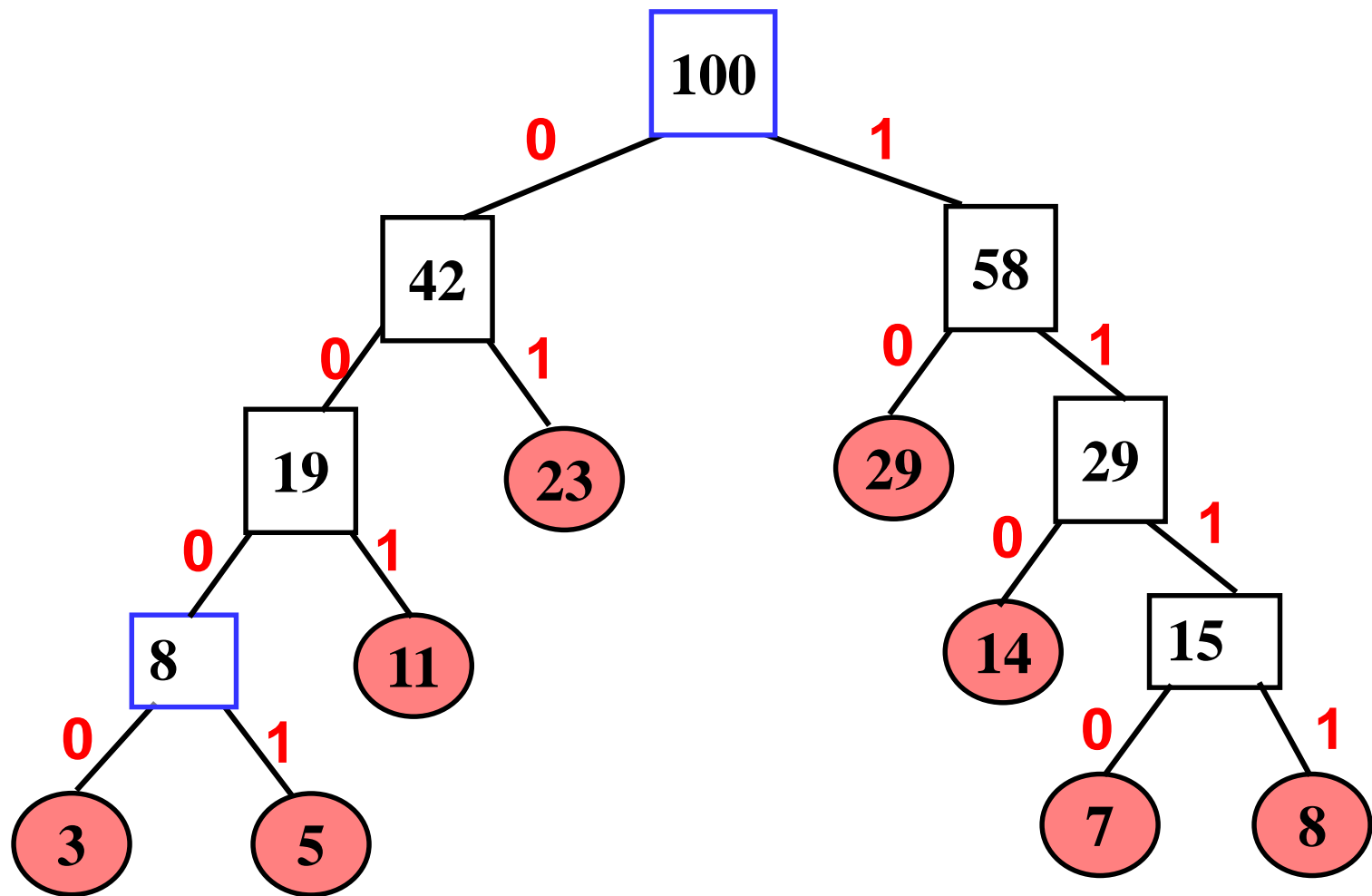
A	B	C	D
0	110	10	111

采用不等长编码，让出现次数多的字符用短码，且任一编码不能是另一编码的前缀。

则上述文字的电文为：0110010101110 共13位。

哈夫曼树及其应用

- 哈夫曼树可用于构造使电文的编码总长最短的编码方案。
- 【具体做法如下】：
 - 设需要编码的字符集合为 $\{d_1, d_2, \dots, d_n\}$ ，它们在电文中出现的次数或频率集合为 $\{w_1, w_2, \dots, w_n\}$ ，以 d_1, d_2, \dots, d_n 作为叶结点， w_1, w_2, \dots, w_n 作为它们的权值，构造一棵哈夫曼树，规定哈夫曼树中的左分支代表0，右分支代表1，则从根结点到每个叶结点所经过的路径分支组成的0和1的序列便为该结点对应字符的编码，我们称之为哈夫曼编码。

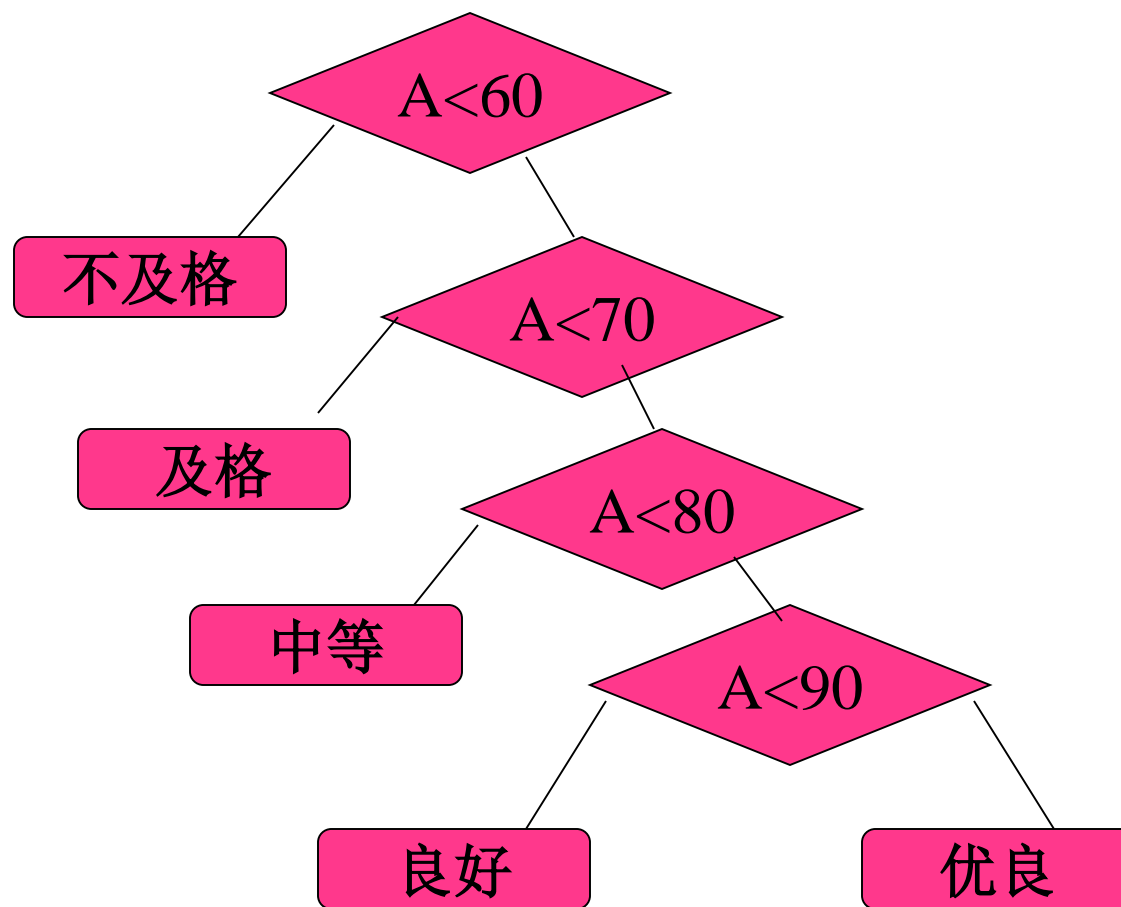


3:0000 5:0001 11:001 7:1000
8:1111 23:01 29:10 14:110

哈夫曼树在判定问题中的应用

- 例如，要编制一个将百分制转换为五级分制的程序。显然，此程序很简单，只要利用条件语句便可完成。如：

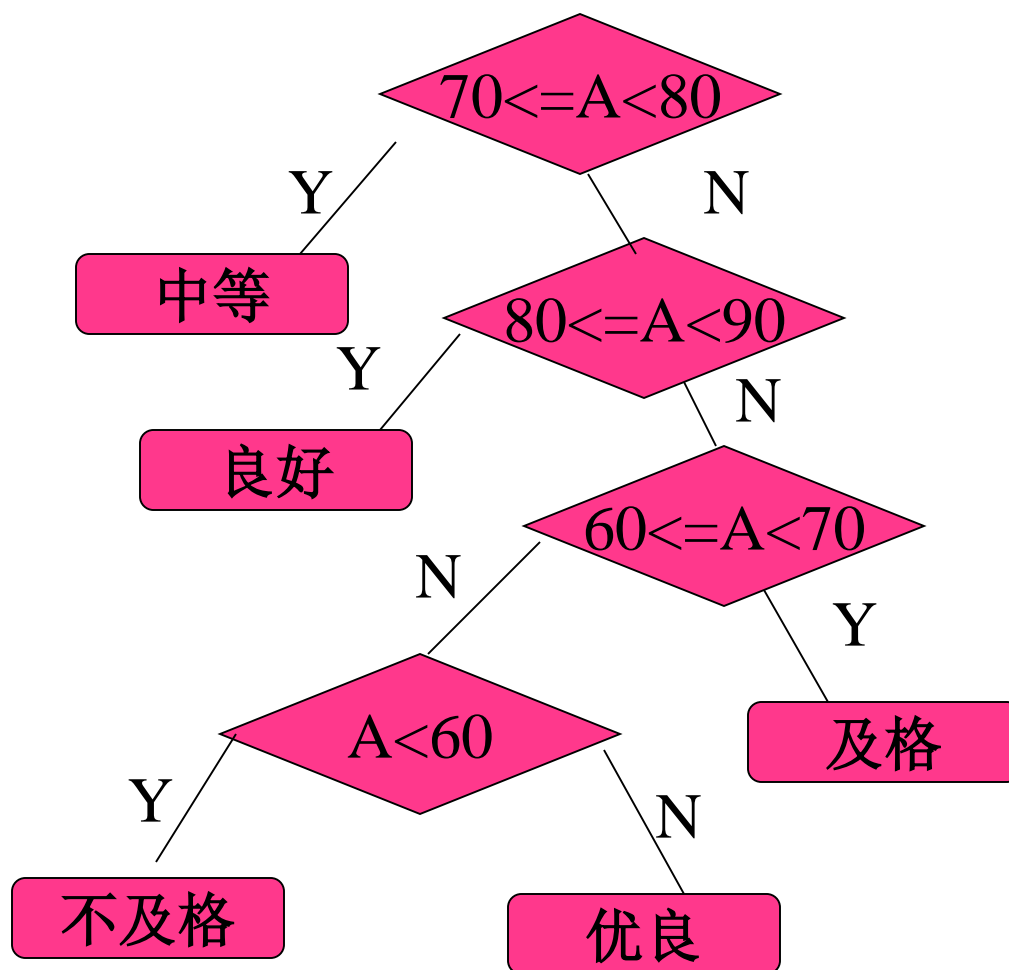
- `if (a<60) b=" bad" ;`
- `else if (a<70) b=" pass"`
- `else if (a<80) b=" general"`
- `else if (a<90) b=" good"`
- `else b=" excellent" ;`

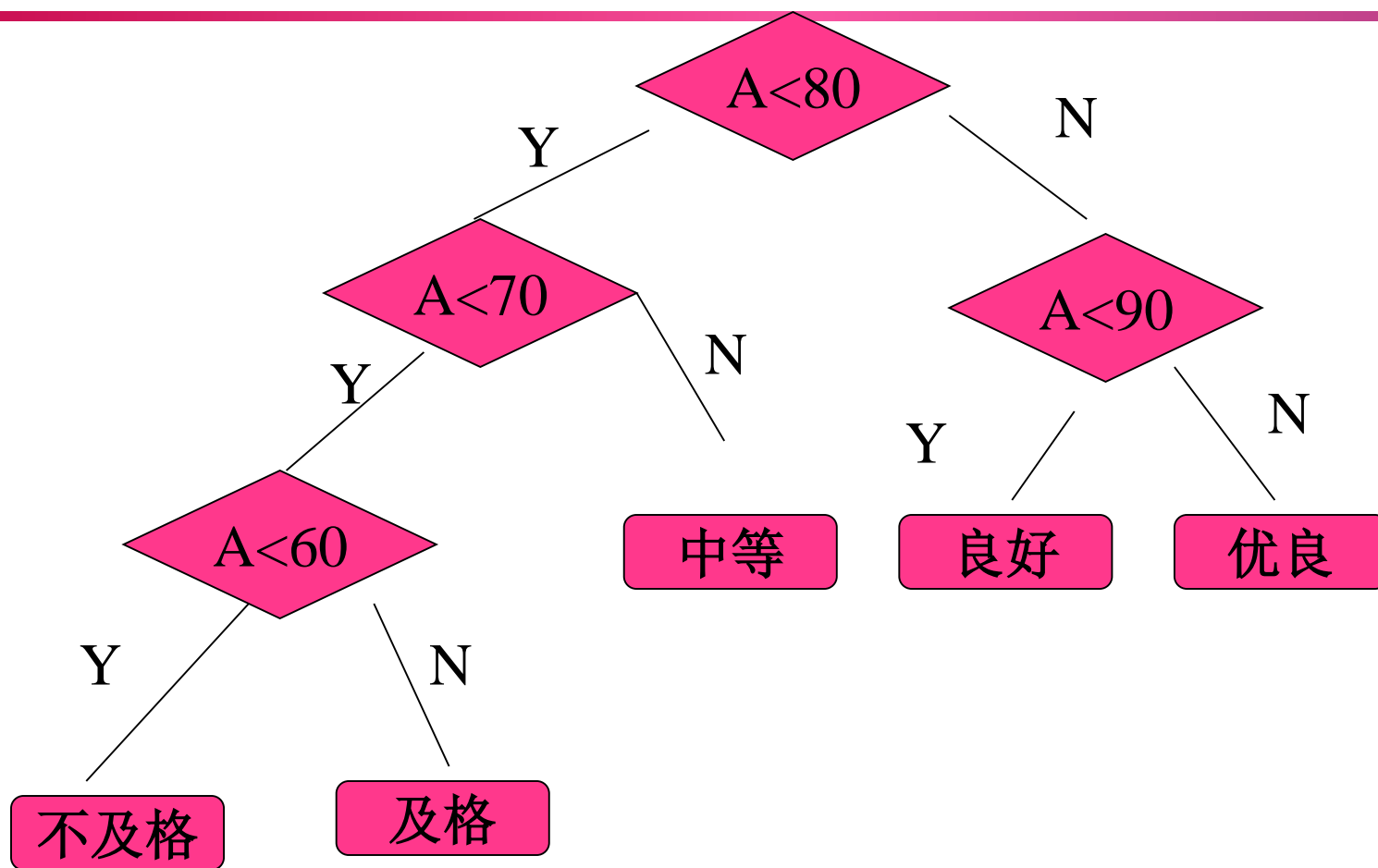


(a)

如果上述程序需反复使用，而且每次的输入量很大，则应考虑上述程序的质量问题，即其操作所需要的时间。因为在实际中，学生的成绩在五个等级上的分布是不均匀的，假设其分布规律如下表所示：

分数	0—59	60—69	70—79	80—89	90—100
比例数	0.05	0.15	0.40	0.30	0.10



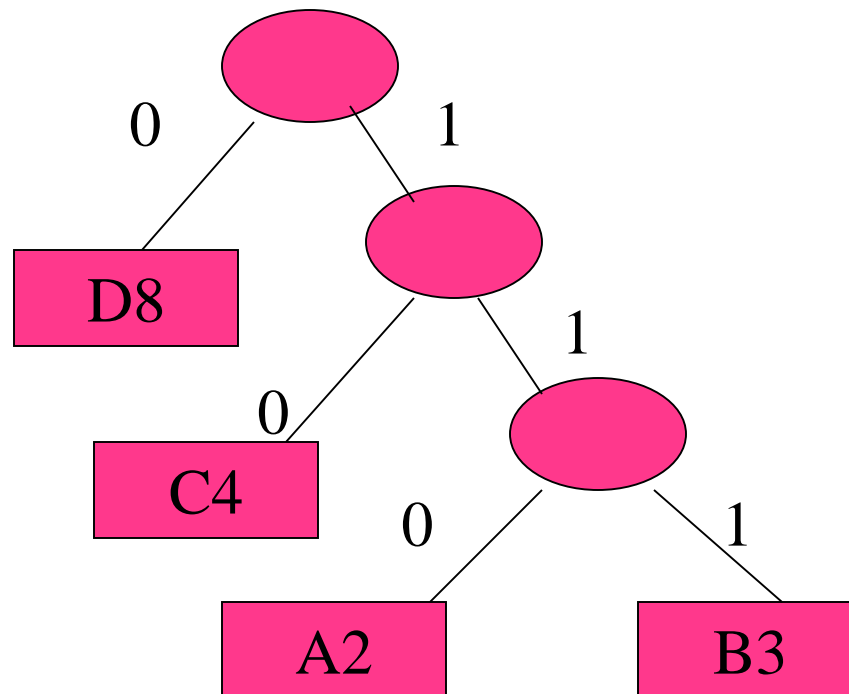


练习：

如果有这样一个文本：

ABCD ACBD BCDCD DDDD

字符集为{A, B, C, D}每个字符相应的出现次数为{2, 3, 4, 8}，求其哈夫曼树。



- 为了实现构造哈夫曼树的算法, 设计哈夫曼树中每个结点类型如下:

```
typedef struct
{ char data;           /*结点值*/
  float weight;        /*权重*/
  int parent;          /*双亲结点*/
  int lchild;          /*左孩子结点*/
  int rchild;          /*右孩子结点*/
} HTNode;
```

- 为了实现构造哈夫曼编码的算法, 设计存放每个结点哈夫曼编码的类型如下:

```
typedef struct
{
    char cd[N];                /*存放当前结点的哈夫曼码*/
    int start;                 /*存放哈夫曼码在cd中的起始位置*/
} HCode;
```

根据哈夫曼树求对应的哈夫曼编码的算法如下：

```
void CreateHCode(HTNode ht[],HCode hcd[],int n)
```

```
{   int i,f,c; HCode hc;
    for (i=0;i<n;i++)           /*根据哈夫曼树求哈夫曼编码*/
    {   hc.start=n;c=i; f=ht[i].parent;
        while (f!=-1) /*循环直到无双亲结点即到达树根结点*/
        {   if (ht[f].lchild==c) /*当前结点是左孩子结点*/
                hc.cd[hc.start--]='0';
            else /*当前结点是双亲结点的右孩子结点*/
                hc.cd[hc.start--]='1';
            c=f;f=ht[f].parent; /*再对双亲结点进行同样的操作*/
        }
        hc.start++;/*start指向哈夫曼编码最开始字符*/
        hcd[i]=hc;
    }
```

本章小结

本章的基本学习要点如下：

(1)掌握树的相关概念,包括树、结点的度、树的度、分支结点、叶子结点、孩子结点、双亲结点、树的深度、森林等定义。

(2)掌握树的表示,包括树形表示法、文氏图表示法、凹入表示法和括号表示法等。

(3)掌握二叉树的概念,包括二叉树、满二叉树和完全二叉树的定义。

(4)掌握二叉树的性质。

(5)重点掌握二叉树的存储结构,包括二叉树顺序存储结构和链式存储结构。

(6)重点掌握二叉树的基本运算和各种遍历算法的实现。

(7)掌握线索二叉树的概念和相关算法的实现。

(8)掌握哈夫曼树的定义、哈夫曼树的构造过程和哈夫曼编码产生方法。

(9)灵活运用二叉树这种数据结构解决一些综合应用问题。

练习

教材中p170的习题1、2、4、6和7。

上机实习题

教材中p170题1、3和7。