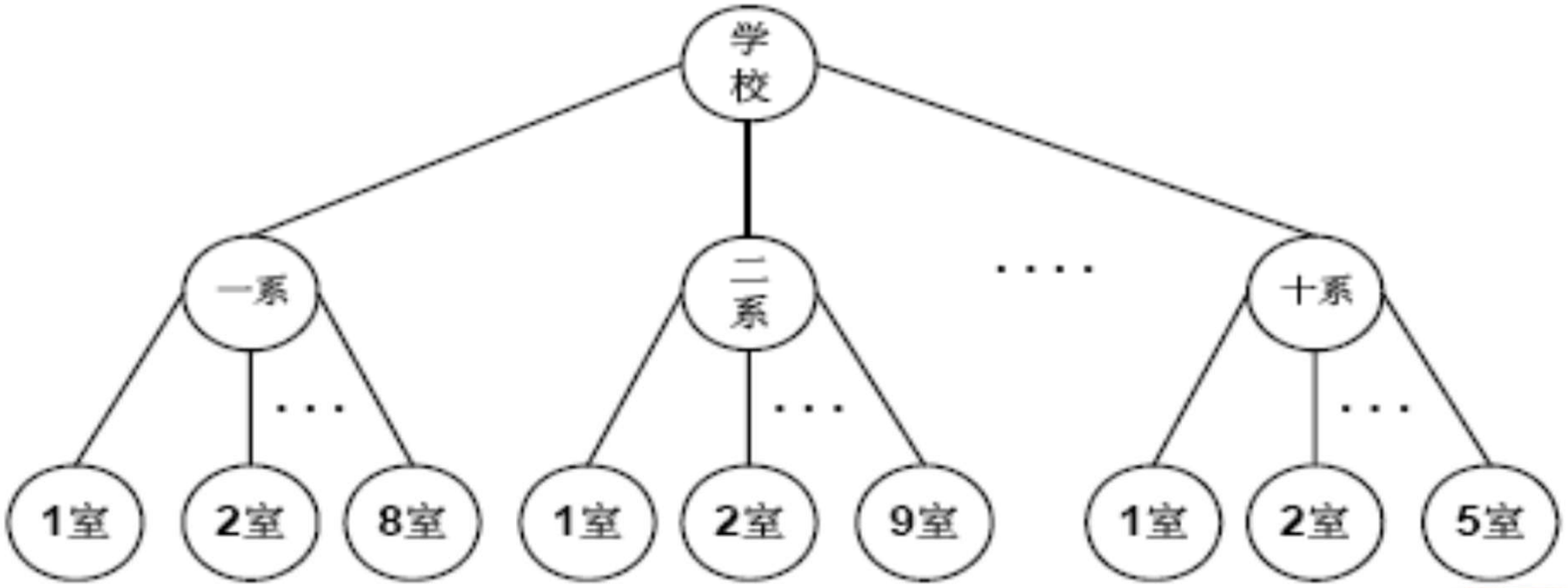
第5章 树

- 树的引入　树又称为树形（数据）结构，其元素结点之间存在明显的分支和层次关系，非常类似于自然界中的树。 ——非线性数据结构
- 本章主要内容
  - 树的基本概念
  - 树的存储
  - 树的遍历
  - 树的应用

树的引入

- 树是一种按层次关系组织起来的分支结构，例如一个学校由若干个系组成，而每个系又可由若干个教研室组成。



树的定义

树的定义

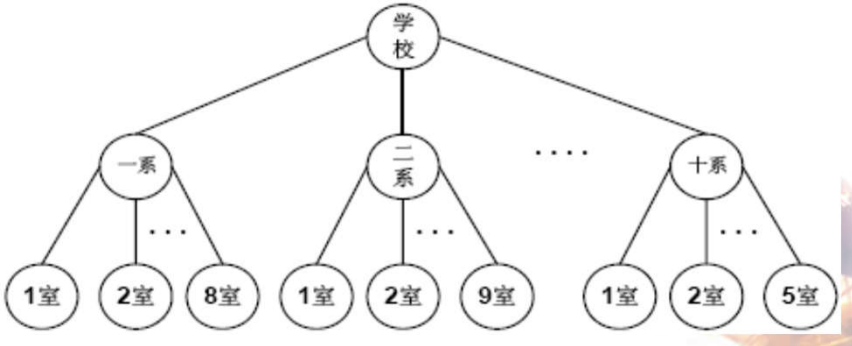
- 树是一种按层次关系组织起来的分支结构。

树的定义

- 树是一种按层次关系组织起来的分支结构。
- 树（Tree）是n（n≥0）个结点的有限集合T，满足两个条件：
  - 有且仅有一个特定的称为根（Root）的结点，它没有前趋；
  - 其余的结点可分成m个互不相交的有限集合$T_1,T_2,\ldots,T_m$，其中每个集合又是一棵树，并称为根的子树。

树的定义

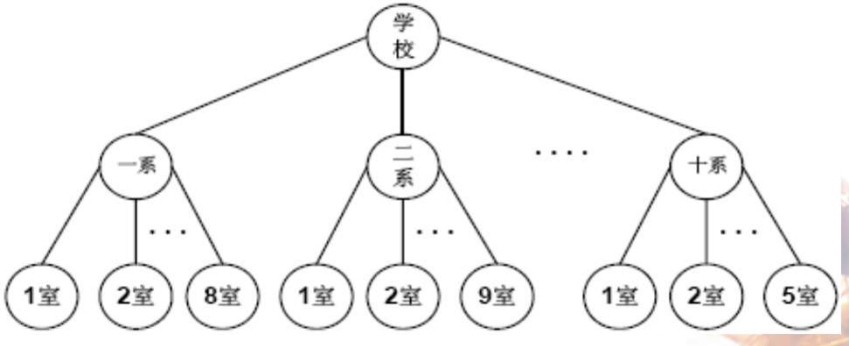
- 树是一种按层次关系组织起来的分支结构。
- 树（Tree）是n（n≥0）个结点的有限集合T，满足两个条件：
  - 有且仅有一个特定的称为根（Root）的结点，它没有前趋；
  - 其余的结点可分成m个互不相交的有限集合$T_1,T_2,\ldots,T_m$，其中每个集合又是一棵树，并称为根的子树。



树的定义

- 树是一种按层次关系组织起来的分支结构。
- 树（Tree）是n（n≥0）个结点的有限集合T，满足两个条件：
  - 有且仅有一个特定的称为根（Root）的结点，它没有前趋；
  - 其余的结点可分成m个互不相交的有限集合$T_1,T_2,\ldots,T_m$，其中每个集合又是一棵树，并称为根的子树。

——递归定义

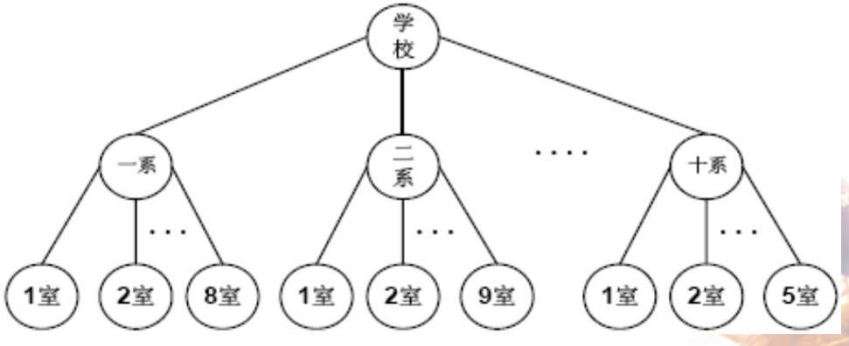


树的定义

- 树是一种按层次关系组织起来的分支结构。
- 树（Tree）是n（n≥0）个结点的有限集合T，满足两个条件：
  - 有且仅有一个特定的称为根（Root）的结点，它没有前趋；
  - 其余的结点可分成m个互不相交的有限集合$T_1,T_2,\ldots,T_m$，其中每个集合又是一棵树，并称为根的子树。

——递归定义

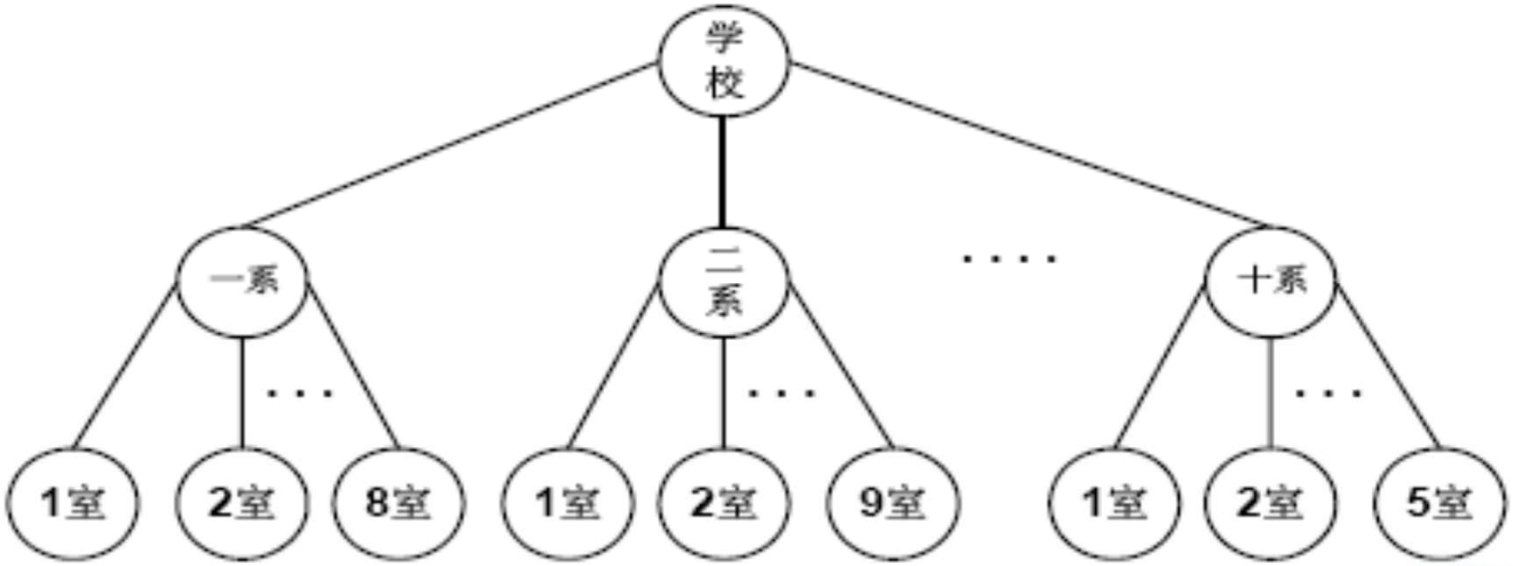
当n=0时的空集合定义为空树



树的表示

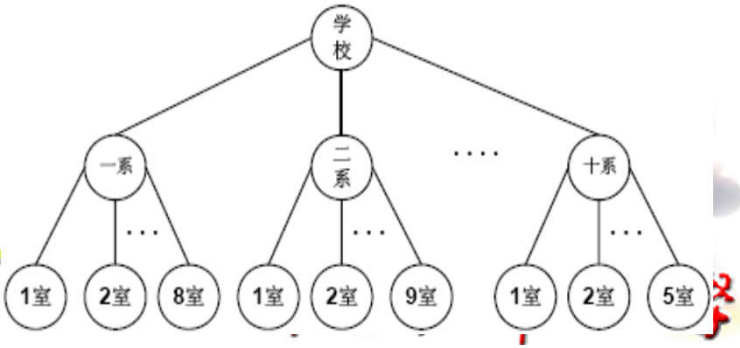
树的表示

- 直观表示法：圆圈表示结点，连线表示结点之间的关系，结点的名字可写在圆圈内或圆圈旁。



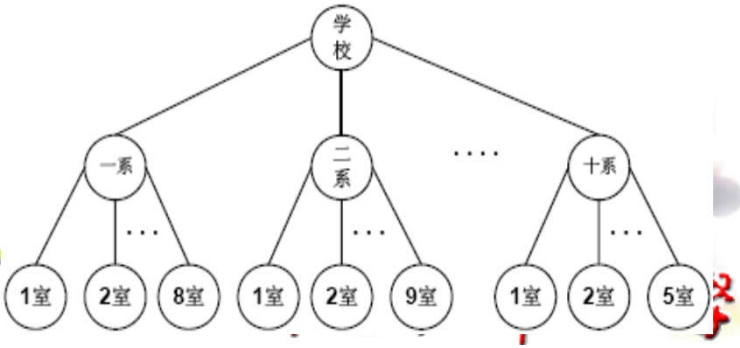
树数据结构中的术语

- 结点： 指树中的一个元素，包含数据项及若干指向其子树的分支。
- 结点的度：指结点拥有的子树个数。如图中的学校结点的度为10，一系结点的度为8。
- 树的度： 指树中最大结点度数。图中的树的度为10。



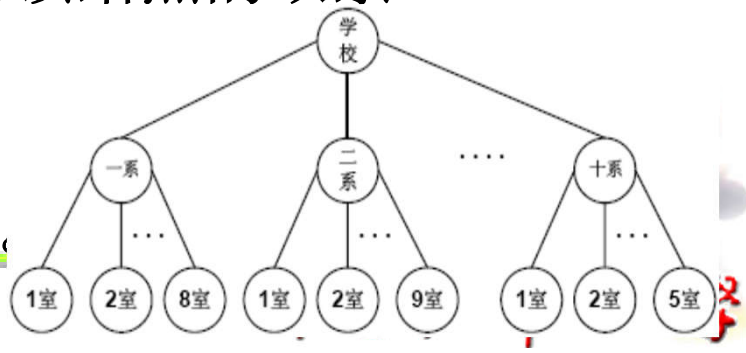
树数据结构中的术语

- 结点： 指树中的一个元素，包含数据项及若干指向其子树的分支。
- 结点的度：指结点拥有的子树个数。如图中的学校结点的度为10，一系结点的度为8。
- 树的度： 指树中最大结点度数。图中的树的度为10。
- 叶子： 指度为零的结点，又称为终端结点。如图中的1室、2室结点。



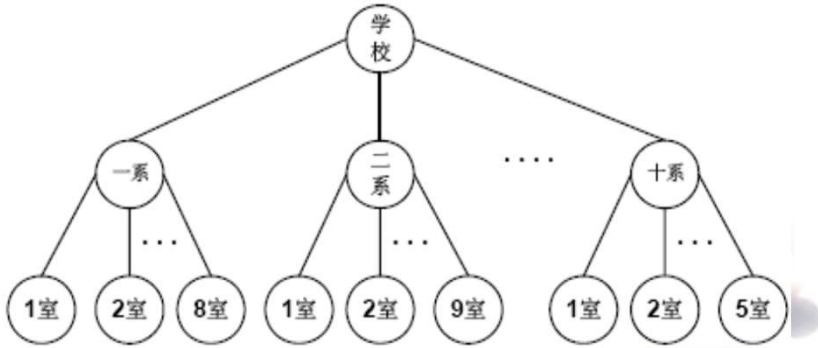
树数据结构中的术语

- 结点： 指树中的一个元素，包含数据项及若干指向其子树的分支。
- 结点的度：指结点拥有的子树个数。如图中的学校结点的度为10，一系结点的度为8。
- 树的度： 指树中最大结点度数。图中的树的度为10。
- 叶子： 指度为零的结点，又称为终端结点。如图中的1室、2室结点。
- 孩子： 一个结点的子树的根称为该结点的孩子。如一系、二系是学校的孩子。
- 双亲： 一个结点的直接上层结点称为该结点的双亲。如学校是一系、二系的双亲。
- 兄弟：同一双亲的孩子互称为兄弟。如图中的一系、二系互为兄弟。



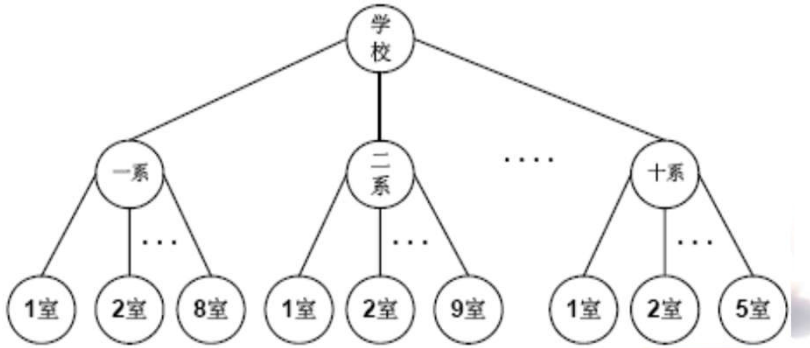
树数据结构中的术语

- 结点的层次：从根结点开始，根结点为第一层，根的孩子为第二层，根的孩子的孩子为第三层，依次类推。图中共分了三层。
- 树的深度：树中结点的最大层次数。图中树的深度为3。



树数据结构中的术语

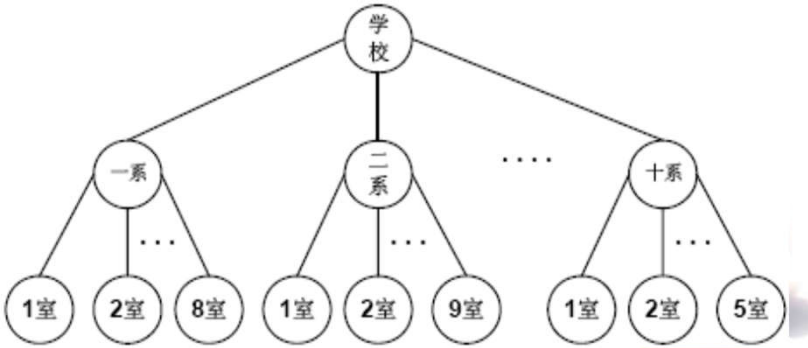
- 结点的层次：从根结点开始，根结点为第一层，根的孩子为第二层，根的孩子的孩子为第三层，依次类推。图中共分了三层。
- 树的深度：树中结点的最大层次数。图中树的深度为3。
- 堂兄弟：双亲在同一层上的结点互称为堂兄弟。如图中一系1室和二系1室互为堂兄弟。



树数据结构中的术语

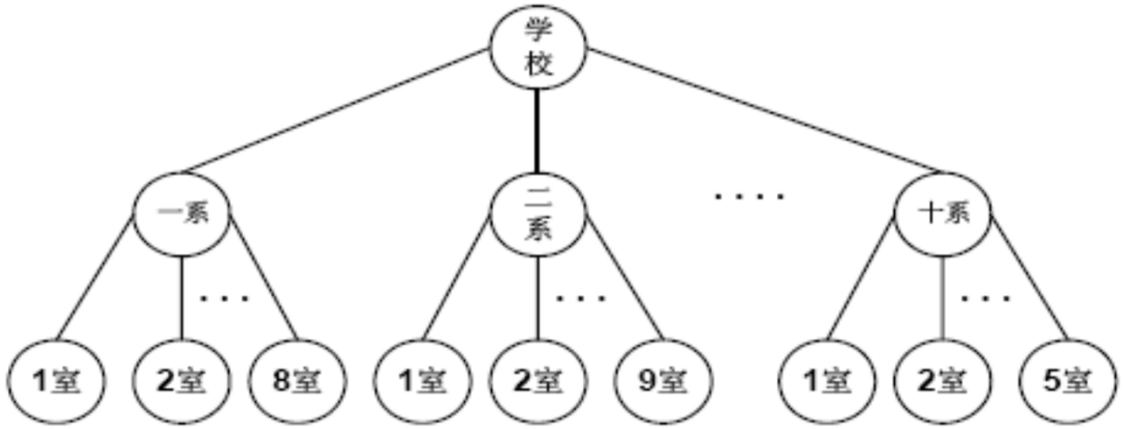
- 结点的层次：从根结点开始，根结点为第一层，根的孩子为第二层，根的孩子的孩子为第三层，依次类推。图中共分了三层。
- 树的深度：树中结点的最大层次数。图中树的深度为3。
- 堂兄弟：双亲在同一层上的结点互称为堂兄弟。如图中一系1室和二系1室互为堂兄弟。
- 路径：若存在一个结点序列$k_1,k_2,\ldots,k_j$，可使k_1到达k_j，则称这个结点序列是k_1到达k_j的一条路径。
- 子孙和祖先：若存在k_1到k_j的一条路径$k_1,k_2,\ldots,k_j$，则$k_1,\ldots,k_{j-1}$为k_j的祖先，而$k_2,\ldots,k_j$为k_1的子孙。

图中学校和各系是教研室的祖先，而系和教研室是学校的子孙。



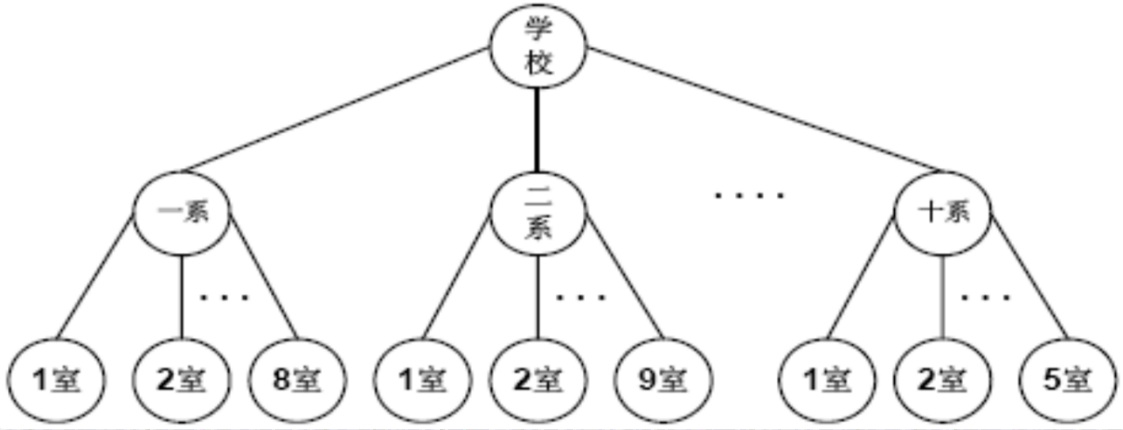
树数据结构中的术语

- 森林：**m(m≥0)**棵互不相交的树的集合构成森林。删除一棵树的根时，就得到了子树构成的森林，当在森林中加上一个根结点时，则森林就变为一棵树。



树数据结构中的术语

- 森林：m(m≥0)棵互不相交的树的集合构成森林。删除一棵树的根时，就得到了子树构成的森林，当在森林中加上一个根结点时，则森林就变为一棵树。
- 有序树和无序树：若将树中每个结点的各个子树都看成是从左到右有次序的（即不能互换），则称该树为有序树，否则为无序树。



对比树和线性结构

线性结构	树结构
存在唯一的没有前驱的"首元素"	存在唯一的没有前驱的"根结点"
存在唯一的没有后继的"尾元素"	存在多个没有后继的"叶子"
其余元素均存在唯一的"前驱元素"和唯一的"后继元素"	其余结点均存在唯一的"前驱(双亲)结点"和多个"后继(孩子)结点"

对比树和线性结构

线性结构	树结构
存在唯一的没有前驱的"首元素"	存在唯一的没有前驱的"根结点"
存在唯一的没有后继的"尾元素"	存在多个没有后继的"叶子"
其余元素均存在唯一的"前驱元素"和唯一的"后继元素"	其余结点均存在唯一的"前驱(双亲)结点"和多个"后继(孩子)结点"

线性结构是一个“序列”，元素之间存在的是“一对一”的关系，

树是一个层次结构，元素之间存在的是"一对多"的关系。

树的存储结构

- 顺序存储
  - 顺序存储时，首先必须对树形结构的结点进行某种方式的线性化，使之成为一个线性序列，然后存储。
- 链式存储
  - 链式存储时，使用多指针域的结点形式，每一个指针域指向一棵子树的根结点。

树的存储结构

- 顺序存储
  - 顺序存储时，首先必须对树形结构的结点进行某种方式的线性化，使之成为一个线性序列，然后存储。
- 链式存储
  - 链式存储时，使用多指针域的结点形式，每一个指针域指向一棵子树的根结点。

由于树的分支数不固定，很难给出一种固定的存储结构，所以本章的剩余内容将集中介绍二叉树

二叉树基本概念

- 定义：
  二叉树是n（n≥0）个结点的有限集，它或为空树（n=0），或由一个根结点及两棵互不相交的、分别称作这个根的左子树和右子树的二叉树构成。

二叉树基本概念

- 定义：
  二叉树是n（n≥0）个结点的有限集，它或为空树（n=0），或由一个根结点及两棵互不相交的、分别称作这个根的左子树和右子树的二叉树构成。

二叉树的定义是递归定义

二叉树基本概念

- 定义：
  二叉树是n（n≥0）个结点的有限集，它或为空树（n=0），或由一个根结点及两棵互不相交的、分别称作这个根的左子树和右子树的二叉树构成。

  二叉树的定义是递归定义

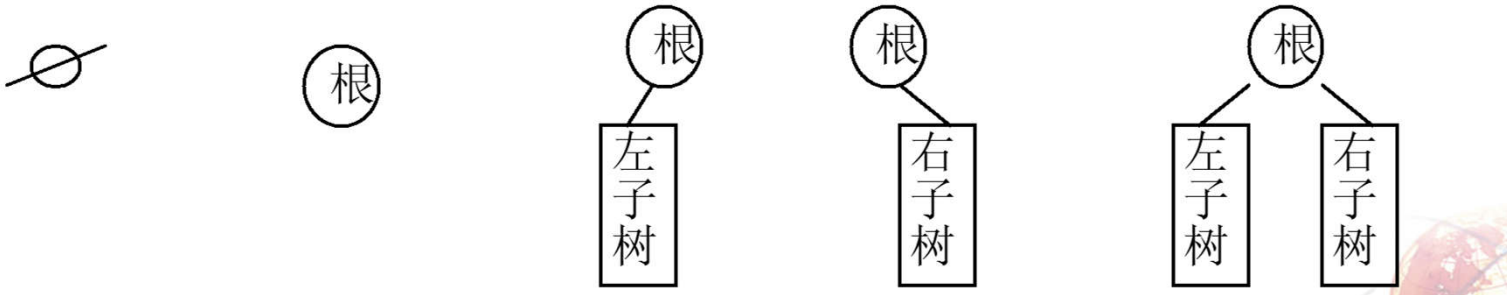
- 二叉树的五种基本形态：

二叉树基本概念

- 定义：
  二叉树是n（n≥0）个结点的有限集，它或为空树（n=0），或由一个根结点及两棵互不相交的、分别称作这个根的左子树和右子树的二叉树构成。

二叉树的定义是递归定义

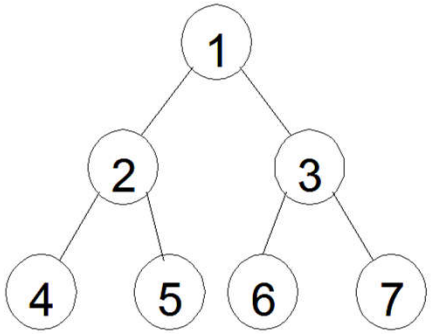
- 二叉树的五种基本形态：



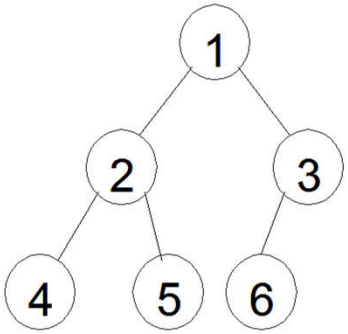
(a) 空二叉树 (b) 仅有根结点 (c) 左子树 (d) 右子树 (e) 左、右子树非空

满二叉树和完全二叉树

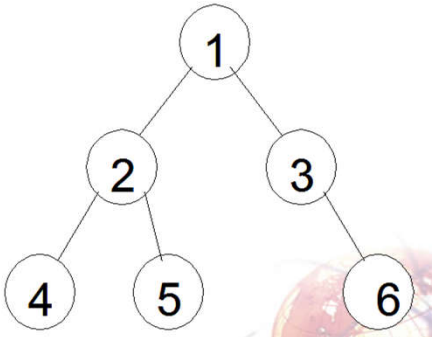
- 一棵深度为k且有2^k-1个结点的二叉树称为满二叉树。
  满二叉树的特点是每一层的结点数都达到该层可具有的最大结点数。不存在度数为1的结点。



k=3 的满二叉树



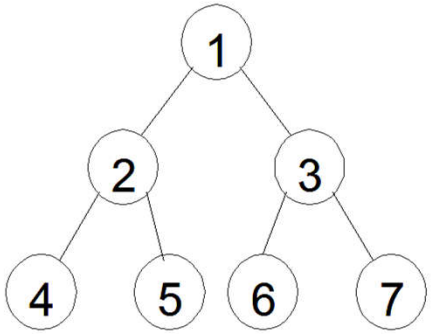
完全二叉树



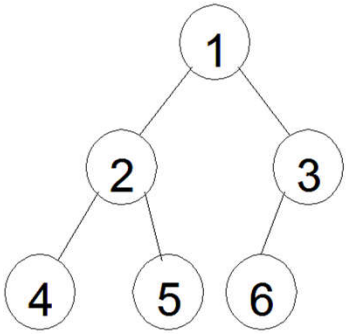
非完全二叉树

满二叉树和完全二叉树

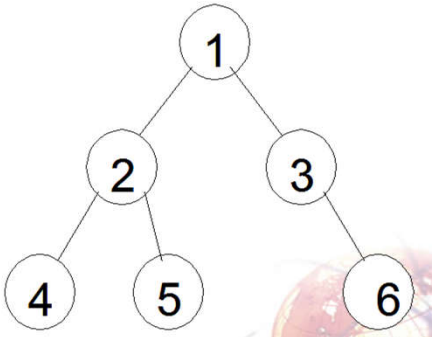
- 一棵深度为k且有2^k-1个结点的二叉树称为满二叉树。满二叉树的特点是每一层的结点数都达到该层可具有的最大结点数。不存在度数为1的结点。
- 如果一个深度为k的二叉树，它的结点按照从根结点开始，自上而下，从左至右进行连续编号后，得到的顺序与满二叉树相应结点编号顺序一致，则称这个二叉树为完全二叉树。



k=3 的满二叉树



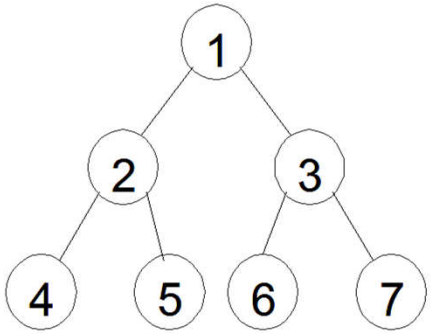
完全二叉树



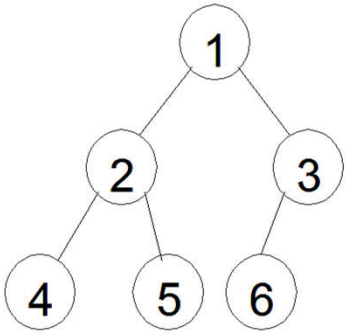
非完全二叉树

满二叉树和完全二叉树

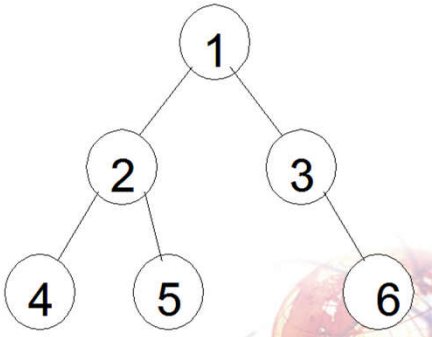
- 一棵深度为k且有2^k-1个结点的二叉树称为满二叉树。满二叉树的特点是每一层的结点数都达到该层可具有的最大结点数。不存在度数为1的结点。
- 如果一个深度为k的二叉树，它的结点按照从根结点开始，自上而下，从左至右进行连续编号后，得到的顺序与满二叉树相应结点编号顺序一致，则称这个二叉树为完全二叉树。
- 注：二叉树是有序树



k=3 的满二叉树



完全二叉树



非完全二叉树

二叉树的性质-1

- 性质1：在二叉树的第i层上至多有2^{i-1}个结点(i≥1)。

二叉树的性质-1

- 性质1：在二叉树的第i层上至多有2^{i-1}个结点($i \geq 1$)。
- 证明：可用数学归纳法予以证明。

当i=1时　有$2^{i-1}=2^0=1$，同时第一层上只有一个根结点，故命题成立。

设当i=k时成立，即第k层上至多有2^{k-1}个结点。

当i=k+1时，由于二叉树的每个结点至多有两个孩子，所以第k+1层上至多有

$2 \times 2^{k-1}=2^k$个结点，故命题成立。

二叉树的性质-2

- 性质2：深度为k的二叉树至多有2^k-1个结点($k\geq 1$)。

二叉树的性质-2

- 性质2：深度为k的二叉树至多有2^k-1个结点($k\geq 1$)。
- 证明：性质1给出了二叉树每一层中含有的最大结点数，深度为k的二叉树的结点总数至多为

故命题成立。

二叉树的性质-2

- 性质2：深度为k的二叉树至多有2^k-1个结点($k\geq 1$)。
- 证明：性质1给出了二叉树每一层中含有的最大结点数，深度为k的二叉树的结点总数至多为

$$\sum_{i=1}^{k} 2^{i-1} = 2^k - 1$$

故命题成立。

二叉树的性质-3

- 性质3：对任何一棵二叉树，如果其终端结点数为n_0，度为2的结点数为n_2，则$n_0=n_2+1$。

二叉树的性质-3

- 性质3：对任何一棵二叉树，如果其终端结点数为n_0，度为2的结点数为n_2，则$n_0=n_2+1$。
- 证明：设度为1的结点数为n_1，则一棵二叉树的结点总数为：

$$n=n_0+n_1+n_2$$

因为除根结点外，其余结点都有一个进入的分支(边)，设B为分支总数，则$n=B+1$。又考虑到分支是由度为1和2的结点发出的，故有$B=2n_2+n_1$，即

$$n=2n_2+n_1+1$$

比较式两式可得$n_0=n_2+1$，证毕。

二叉树的性质-4

- 性质4：具有n个结点的完全二叉树的深度为$\lfloor \log_2 n \rfloor + 1$或$\lceil \log_2 (n+1) \rceil$。
  (⌊x⌋表示不大于x的最大整数，⌈x⌉表示不小于x的最小整数)

二叉树的性质-4

- 性质4：具有n个结点的完全二叉树的深度为$\lfloor \log_2 n \rfloor+1$或$\lceil \log_2(n+1) \rceil$。
  (⌊x⌋表示不大于x的最大整数，⌈x⌉表示不小于x的最小整数)

- 证明：由完全二叉树的定义可知，一个k层的完全二叉树的前k-1层共有$2^{k-1}-1$个结点，第k层上还有若干结点，所以结点总数n满足关系：

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

因而可推出$2^{k-1}\leq n<2^k$，取对数后可得 $k-1\leq \log_2 n<k$。因为k为整数，故有$k-1=\lfloor \log_2 n \rfloor$，即$k=\lfloor \log_2 n \rfloor+1$。同样有$2^{k-1}<n+1\leq 2^k$，取对数得$k-1<\log_2(n+1)\leq k$，因而$k=\lceil \log_2(n+1) \rceil$，证毕。

二叉树的存储结构

- 二叉树的存储结构
  - 顺序存储结构
  - 链式存储结构

完全二叉树的顺序存储结构

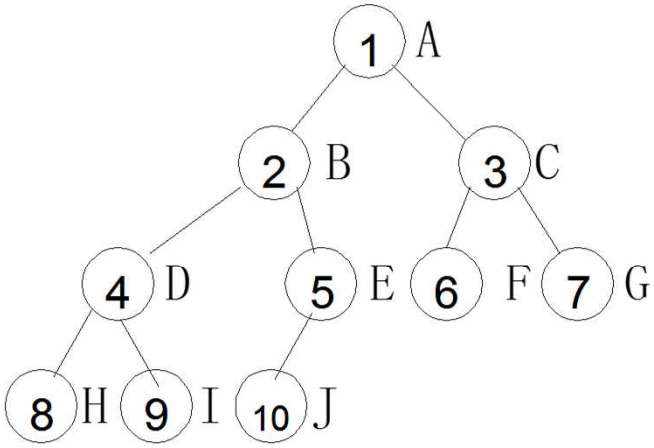
- 在一棵完全二叉树中，按照从根结点起，自上而下，从左至右的方式对结点进行顺序编号，便可得到一个反映结点之间关系的线性序列。

完全二叉树的顺序存储结构

- 在一棵完全二叉树中，按照从根结点起，自上而下，从左至右的方式对结点进行顺序编号，便可得到一个反映结点之间关系的线性序列。
- 下图的完全二叉树的顺序存储结构如下表

完全二叉树的顺序存储结构

- 在一棵完全二叉树中，按照从根结点起，自上而下，从左至右的方式对结点进行顺序编号，便可得到一个反映结点之间关系的线性序列。
- 下图的完全二叉树的顺序存储结构如下表



完全二叉树的结点编号

左图的完全二叉树的顺序存储

编 号	0	1	2	3	4	5	6	7	8	9	10
结点值		A	B	C	D	E	F	G	H	I	J

完全二叉树编号的性质

按照以上的顺序编号方法，可以从一个结点的编号推得它的双亲，左、右孩子和兄弟等的编号。

完全二叉树编号的性质

按照以上的顺序编号方法，可以从一个结点的编号推得它的双亲，左、右孩子和兄弟等的编号。

- 若i=1，则i结点是根结点；若i＞1，则i结点的双亲编号为⌊i/2⌋。

完全二叉树编号的性质

按照以上的顺序编号方法，可以从一个结点的编号推得它的双亲，左、右孩子和兄弟等的编号。

- 若i=1，则i结点是根结点；若i＞1，则i结点的双亲编号为⌊i/2⌋。
- 若2i＞n，则i结点无左孩子，i结点是终端结点；若2i≤n，则2i是结点i的左孩子。
- 若2i+1＞n，则i结点无右孩子；若2i+1≤n，则2i+1是结点i的右孩子。

完全二叉树编号的性质

按照以上的顺序编号方法，可以从一个结点的编号推得它的双亲，左、右孩子和兄弟等的编号。

- 若i=1，则i结点是根结点；若i＞1，则i结点的双亲编号为⌊i/2⌋。
- 若2i＞n，则i结点无左孩子，i结点是终端结点；若2i≤n，则2i是结点i的左孩子。
- 若2i+1＞n，则i结点无右孩子；若2i+1≤n，则2i+1是结点i的右孩子。
- 若i为奇数且不等于1时，结点i的左兄弟是i-1。
- 若i为偶数且小于n时，结点i的右兄弟是结点i+1，否则结点i没有右兄弟。

一般二叉树的顺序存储

一般二叉树的顺序存储

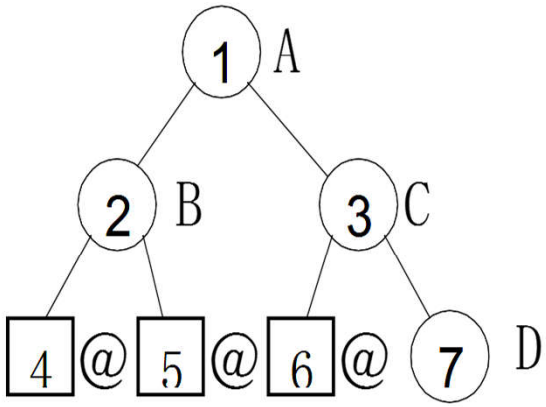
- 按完全二叉树的方式存储一般二叉树
  - 将二叉树映射为完全二叉树（通过虚结点）；
  - 用完全二叉树的方式存储。

一般二叉树的顺序存储

- 按完全二叉树的方式存储一般二叉树
  - 将二叉树映射为完全二叉树（通过虚结点）；
  - 用完全二叉树的方式存储。
- 例

一般二叉树的顺序存储

- 按完全二叉树的方式存储一般二叉树
  - 将二叉树映射为完全二叉树（通过虚结点）；
  - 用完全二叉树的方式存储。
- 例



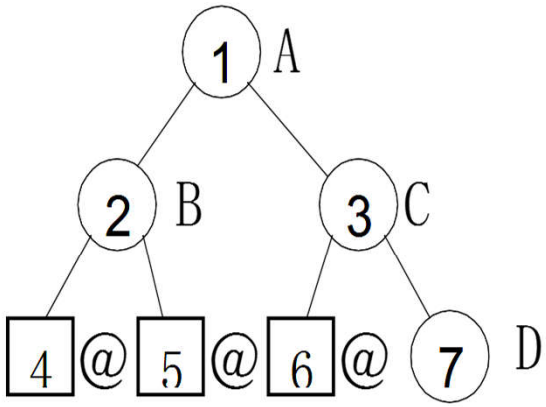
一般二叉树的结点编号

一般二叉树的顺序存储

编 号	0	1	2	3	4	5	6	7
结点值		A	B	C	@	@	@	D

一般二叉树的顺序存储

- 按完全二叉树的方式存储一般二叉树
  - 将二叉树映射为完全二叉树（通过虚结点）；
  - 用完全二叉树的方式存储。
- 例



一般二叉树的结点编号

一般二叉树的顺序存储

编 号	0	1	2	3	4	5	6	7
结点值		A	B	C	@	@	@	D

对于一般二叉树，通过虚结点来构成完全二叉树，虽然保持了结点间的逻辑关系，但也造成了存储空间的浪费。

二叉树的顺序存储结构描述

```
#define maxsize 1024
typedef  int  datatype;

typedef   struct
{   datatype data [maxsize];
    int  last;
} sequenlist;
```

此处的last将存放最后一个结点在表中的位置。

二叉树的链式存储

- 二叉树的链式存储结构

  含有两个指针域来分别指向左孩子指针域(lchild)和右孩子指针域(rchild)，以及结点数据域(data)，故二叉树的链式存储结构也称为二叉链表。

二叉树的链式存储

- 二叉树的链式存储结构

  含有两个指针域来分别指向左孩子指针域(lchild)和右孩子指针域(rchild)，以及结点数据域(data)，故二叉树的链式存储结构也称为二叉链表。

- 二叉链表结点的C语言逻辑描述为：

```
typedef   int  datatype;
struct TreeNode {
        datatype data;
        struct  TreeNode *lchild，*rchild;
};
struct TreeNode  *root;
```

二叉树的链式存储

- 二叉树的链式存储结构
  含有两个指针域来分别指向左孩子指针域(lchild)和右孩子指针域(rchild)，以及结点数据域(data)，故二叉树的链式存储结构也称为二叉链表。
- 二叉链表结点的C语言逻辑描述为：

```
typedef   int  datatype;
struct TreeNode {
        datatype data;
        struct  TreeNode *lchild，*rchild;
};
struct TreeNode  *root;
```

- 注： root是指向根结点的头指针，当二叉树为空时，则root=NULL。若结点某个孩子不存在时，则相应的指针为空。

二叉树的链式存储

- 二叉树的链式存储结构

  含有两个指针域来分别指向左孩子指针域(lchild)和右孩子指针域(rchild)，以及结点数据域(data)，故二叉树的链式存储结构也称为二叉链表。

- 二叉链表结点的C语言逻辑描述为：

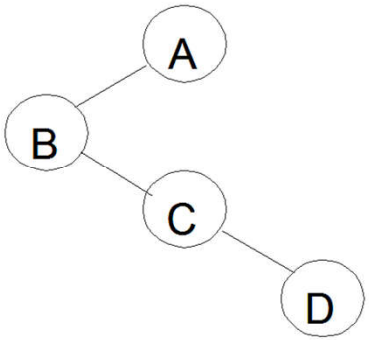
```
typedef   int  datatype;
struct TreeNode {
        datatype data;
        struct  TreeNode *lchild，*rchild;
};
struct TreeNode  *root;
```

- 注： root是指向根结点的头指针，当二叉树为空时，则root=NULL。若结点某个孩子不存在时，则相应的指针为空。

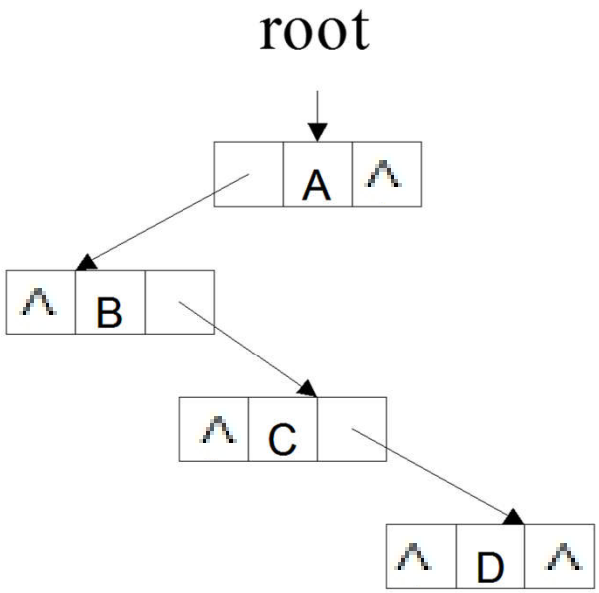
- 三叉链表

  二叉链表中，要寻找某结点的双亲是困难的，故增加一个指向其双亲的指针域parent。

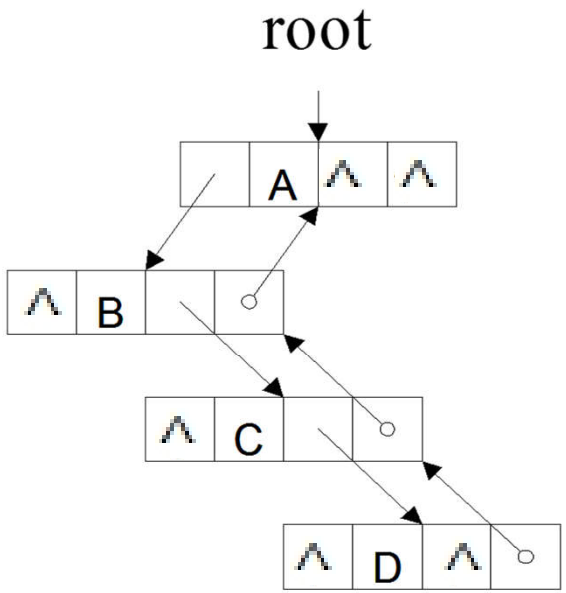
二叉树的链式存储实例



(a) 二叉树



(b)二叉链表



(c)三叉链表

二叉树的建立（链式存储）

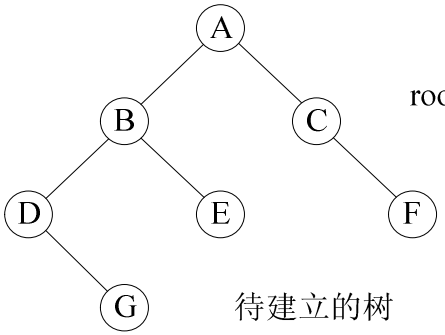
- 算法的基本思想是：
  - 依次输入结点信息，若输入的结点不是虚结点，则建立一个新结点。
  - 若新结点是第1个结点，则令其为根结点，否则将新结点作为孩子链接到它的双亲结点上。
  - 如此反复进行，直到输入结束标志“#”为止。

二叉树的建立（链式存储）

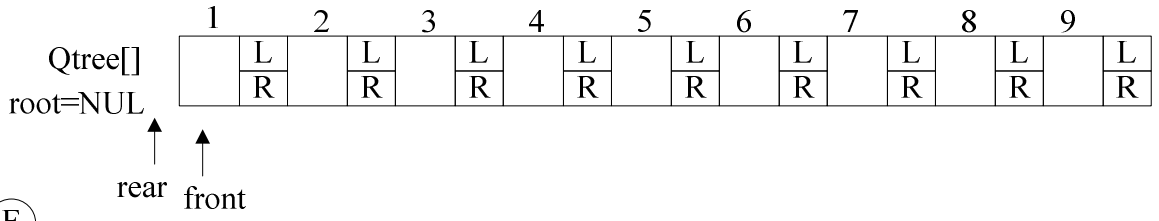
- 算法实现时的考虑：
  - 设置一个指针类型的数组构成的队列来保存已输入结点的地址，并使队尾(rear)指向当前输入的结点；队头(front)指向这个结点的双亲结点。
  - 由于根结点的地址放在队列的第一个单元里，所以当rear为偶数时，则rear所指的结点应作为左孩子与其双亲链接，否则rear所指的结点应作为右孩子与其双亲链接。
  - 若双亲结点或孩子结点为虚结点，则无须链接。
  - 当一个双亲结点与两个孩子链接完毕，则进行出队操作，使队头指针指向下一个待链接的双亲结点。

二叉树的建立实例

- 以下图为例说明



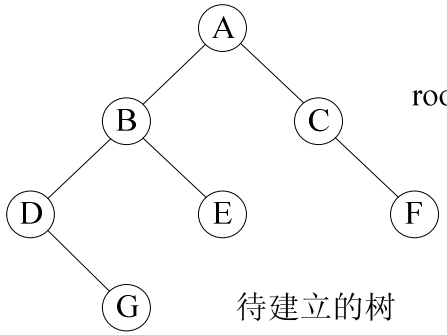
待建立的树



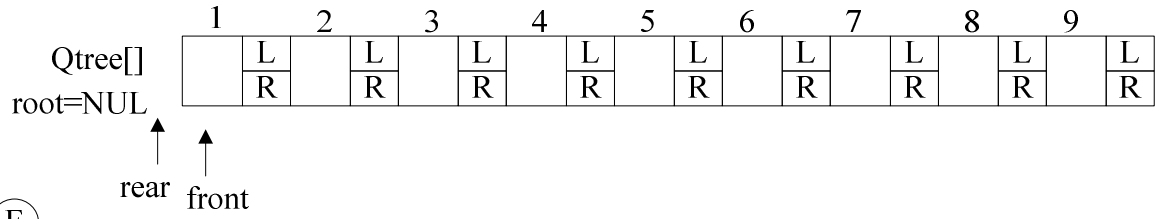
指针类型数组Qtree[]构成队列

二叉树的建立实例

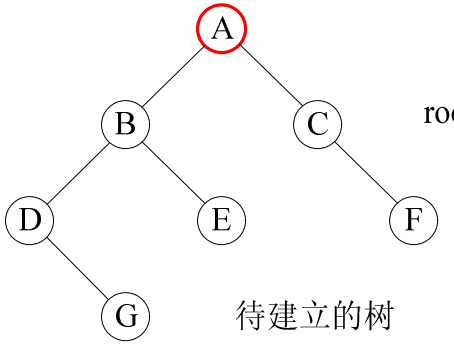
- 以下图为例说明



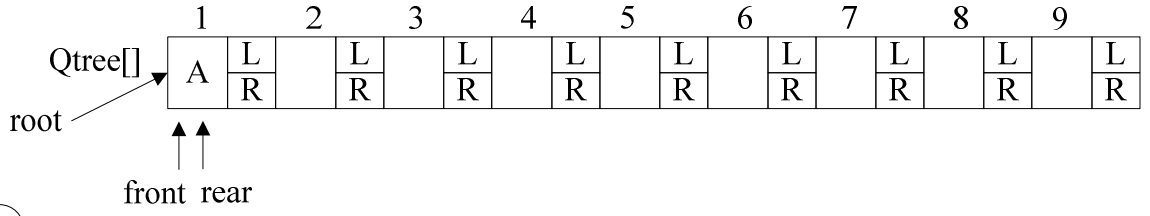
待建立的树



指针类型数组Qtree[]构成队列

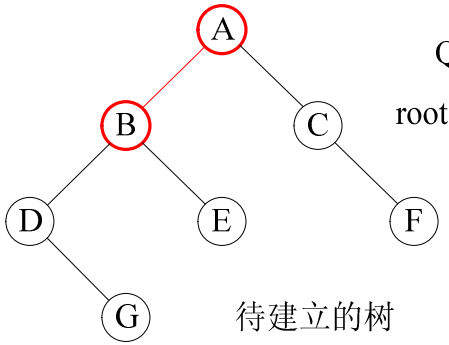


待建立的树

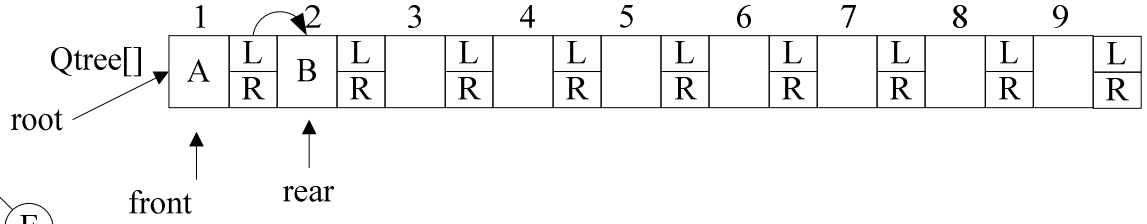


建立A，作为根结点

二叉树的建立实例

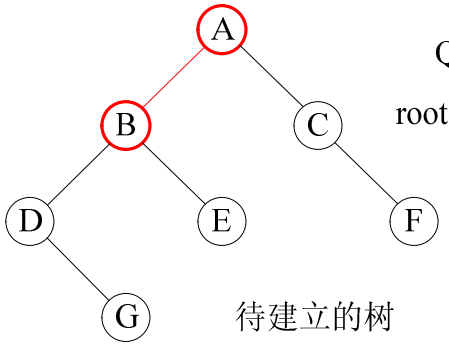


待建立的树

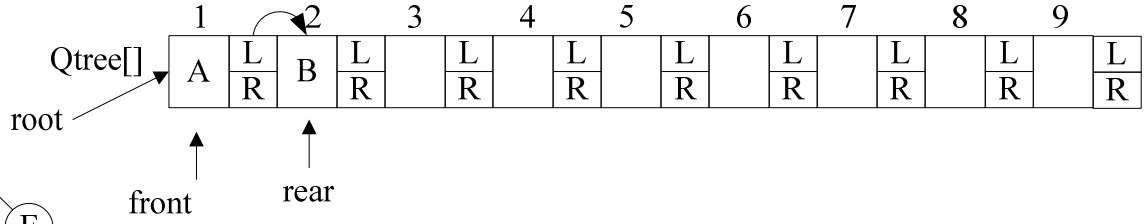


加入B，作为A的左孩子

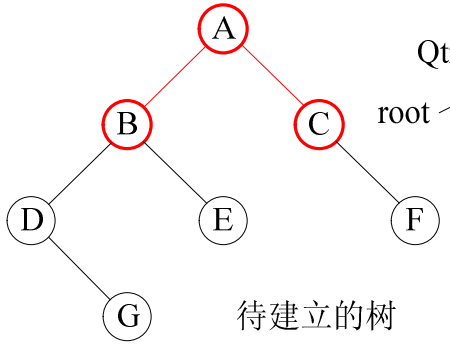
二叉树的建立实例



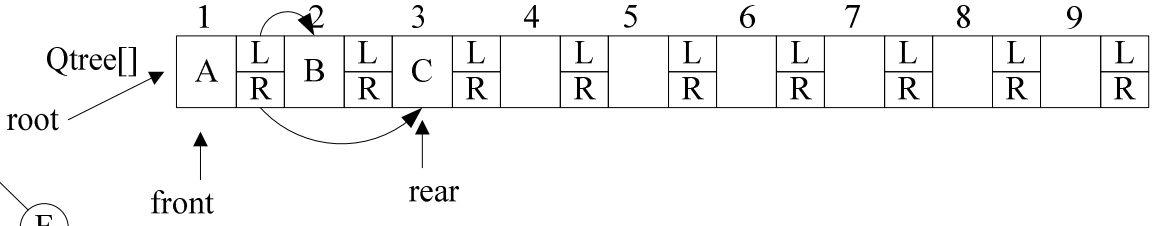
待建立的树



加入B，作为A的左孩子

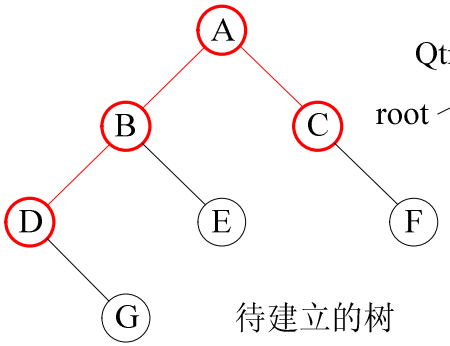


待建立的树

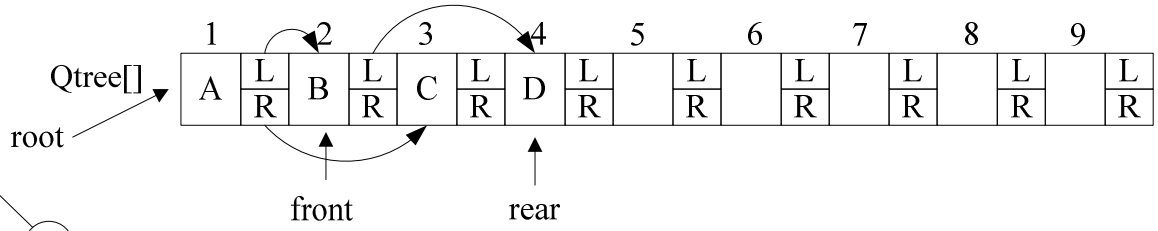


加入C，作为A的右孩子

二叉树的建立实例

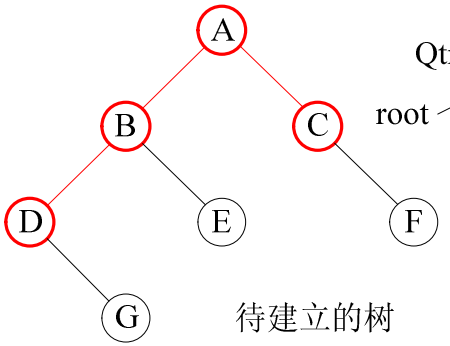


待建立的树

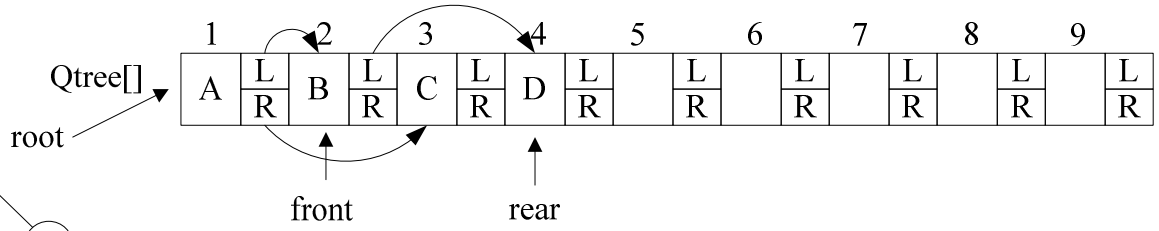


加入D，作为B的左孩子

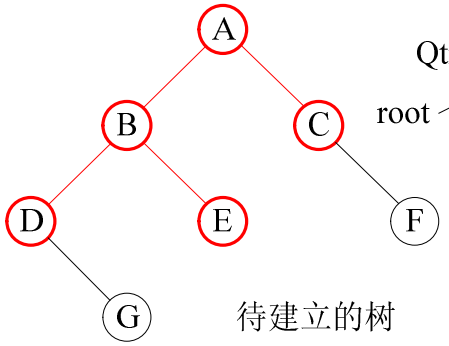
二叉树的建立实例



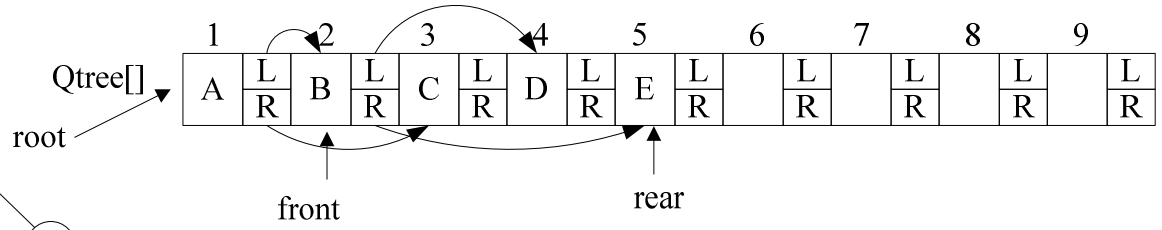
待建立的树



加入D，作为B的左孩子

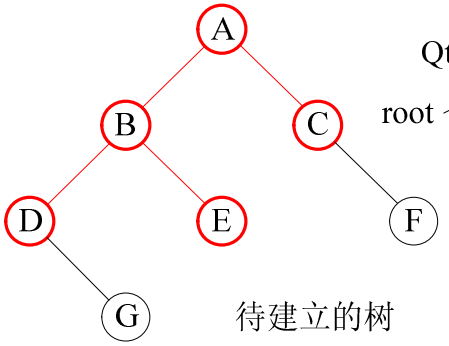


待建立的树

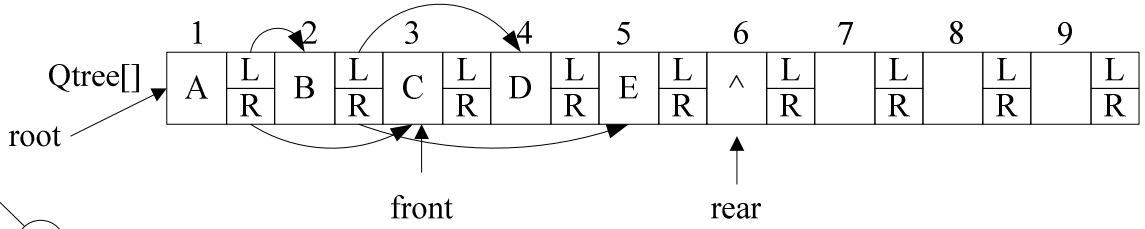


加入E，作为B的右孩子

二叉树的建立实例

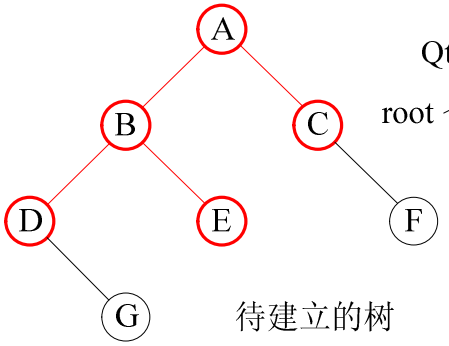


待建立的树

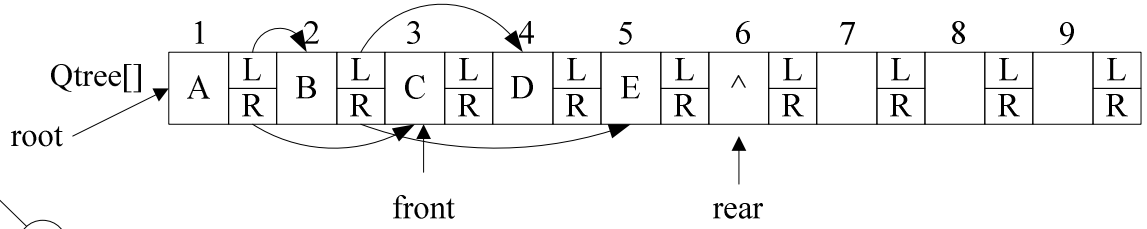


加入@（NULL），作为C的左孩子

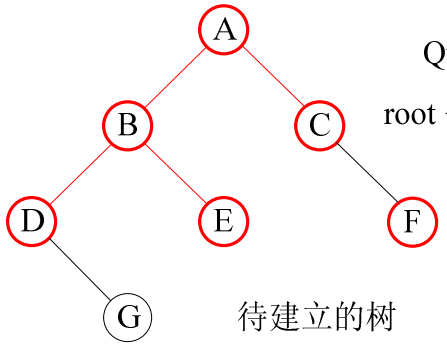
二叉树的建立实例



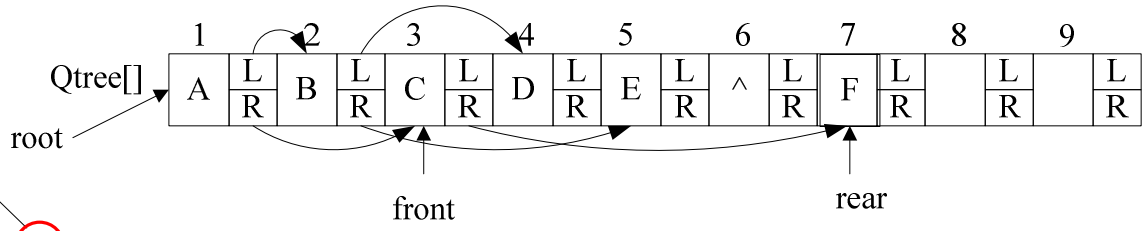
待建立的树



加入@（NULL），作为C的左孩子

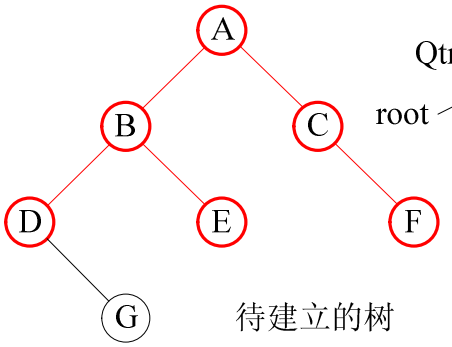


待建立的树

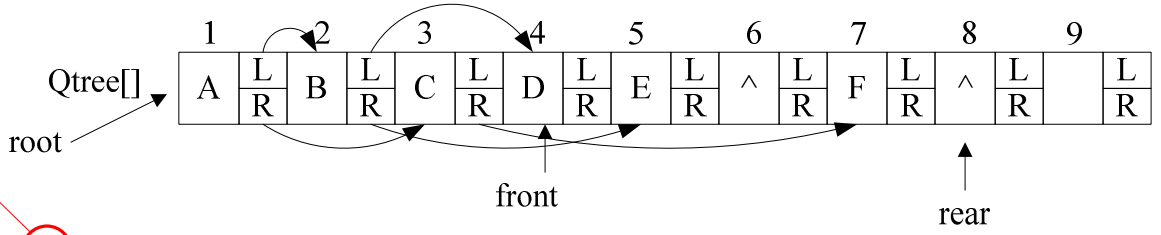


加入F，作为C的右孩子

二叉树的建立实例

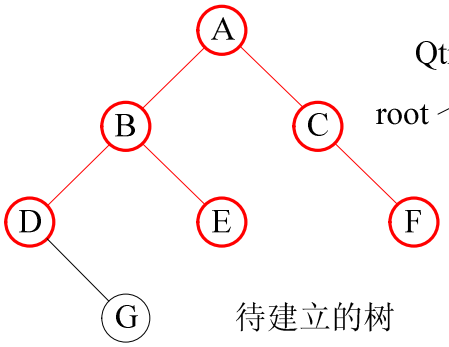


待建立的树

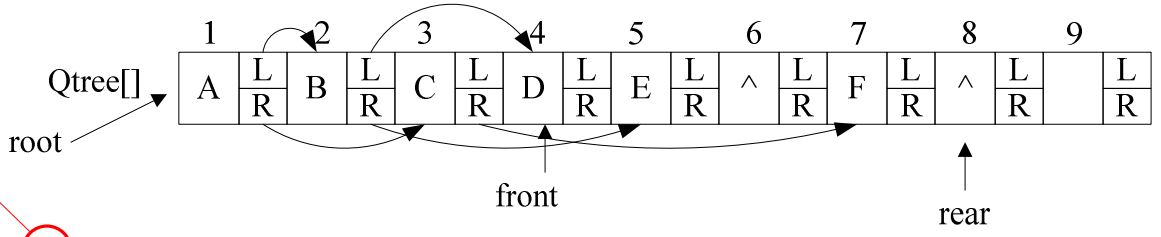


加入@（NULL），作为D的左孩子

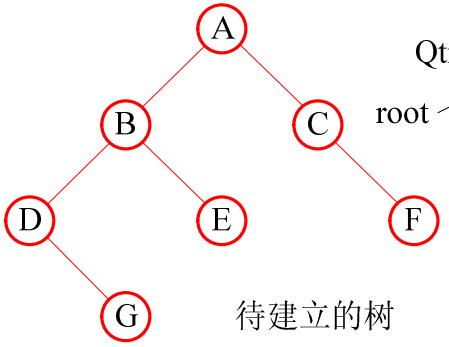
二叉树的建立实例



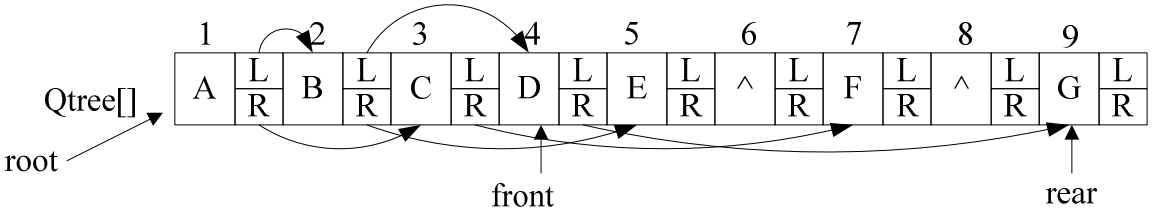
待建立的树



加入@（NULL），作为D的左孩子

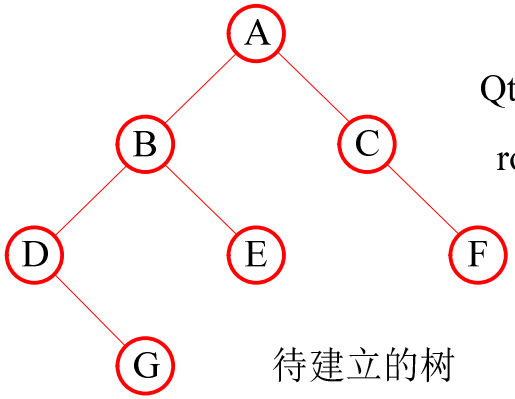


待建立的树

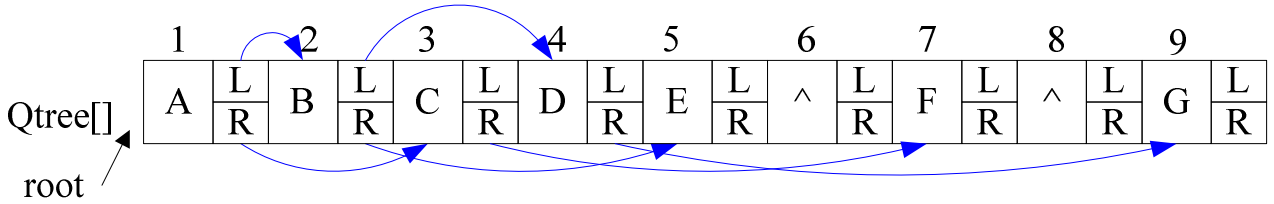


加入G，作为D的右孩子

二叉树的建立实例



待建立的树



输入#结束，所建立的树存放在root所指的数据结构中

二叉树的建立算法

- 算法如下：

```
struct TreeNode *CREATREE( )
                /* 建立二叉树函数，函数返回指向根结点的指针 */
{char ch;              /* 结点信息变量 */
 struct TreeNode  *Q [maxsize];     /*指针数组构成队列 * /
 int front, rear;           /* 队头和队尾指示变量 */
 struct TreeNode  *root, *s;   /* 根结点和中间变量指针*/
 root=NULL;             /* 二叉树置空 */
  front=1;  rear=0;       /* 设置队列指针变量初值 */

/*以上为初始化*/
```

二叉树的建立算法（续）

```
while((ch=getchar())!='#')  /* 输入一个字符，当不是结束符时执行以下操作 */
      {s=NULL;
       if(ch!='@')     /* @表示虚结点，当不是虚结点则建立新结点 */
       {s=(struct TreeNode *) malloc (sizeof (struct TreeNode));
            s→data=ch;  s→lchild=NULL;  s→rchild=NULL;
       }
       rear++;          /* 队尾指针增1，指向新结点地址应存放的单元 */
       Q[rear]=s;          /* 将新结点地址入队或虚结点指针NULL入队 */
       if (rear= =1)  root=s;   /* 输入的第一个结点作为根结点 */
       else   { if (s && Q[front])     /* 孩子和双亲结点都不是虚结点 */
                 if (rear % 2= =0) Q[front]→lchild=s;  /*rear为偶数，新结点是左孩子
*/
                 else Q[front]→rchild=s;   /* rear为奇数且不等于1，新结点是右孩子 */
                 if (rear % 2= =1)
                    front++;     /* 结点* Q[front]的两个孩子处理完毕，出队列 */
                }
     }
  return root;  /* 返回根指针 */
} /* CREATREE */
```

二叉树的遍历

- 定义：

  二叉树的遍历是指按某种搜索路线来巡访二叉树中的每一个结点，使每个结点被且仅被访问一次。

- 二叉树的遍历有两种：
  - 深度优先遍历
  - 广度优先遍历

二叉树的深度优先遍历

- 设以L、D和R分别表示遍历左子树、访问根结点和遍历右子树，则有六种不同的二叉树深度优先遍历方案：
  - DLR，LDR，LRD，DRL，RDL，RLD。

二叉树的深度优先遍历

- 设以L、D和R分别表示遍历左子树、访问根结点和遍历右子树，则有六种不同的二叉树深度优先遍历方案：
  - **DLR，LDR，LRD，DRL，RDL，RLD。**
- 若限定按先左后右进行遍历，则只有三种遍历方案：
  - **DLR(先(前)序(根)遍历)，LDR (中序(根)遍历)和LRD (后序(根)遍历) 。**

先序遍历算法

- 先序遍历算法的遍历过程是：
  - 若二叉树非空，执行以下操作：
    - 访问根结点；
    - 先序遍历左子树；
    - 先序遍历右子树；

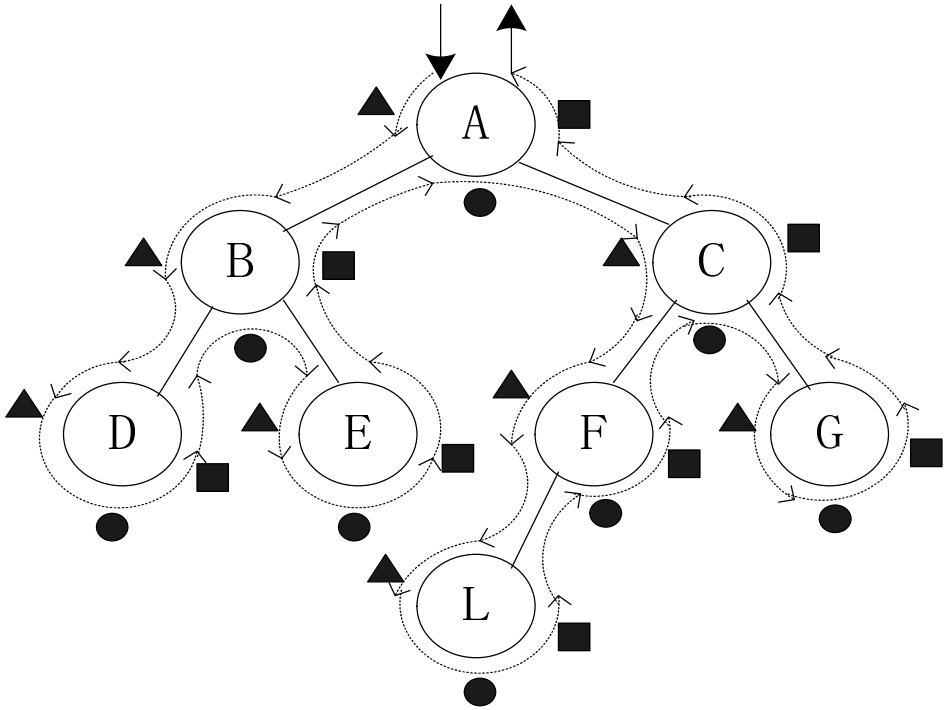
先序遍历算法

- 先序遍历算法的遍历过程是：
  - 若二叉树非空，执行以下操作：
    - 访问根结点；
    - 先序遍历左子树；
    - 先序遍历右子树；
- 先序遍历算法如下：

```
void preorder(bitree *p)
 /* 先序遍历二叉树，p指向二叉树的根结点 */
{ ① if (p!=NULL)  /*  二叉树p非空，则执行以下操作 */
    { ② printf (" %c ", p→data); /*  访问p所结点 */
      ③ preorder (p→lchild);   /* 先序遍历左子树 */
      ④ preorder (p→rchild);   /* 先序遍历右子树 */
    }
⑤ }  /*  preorder */
```

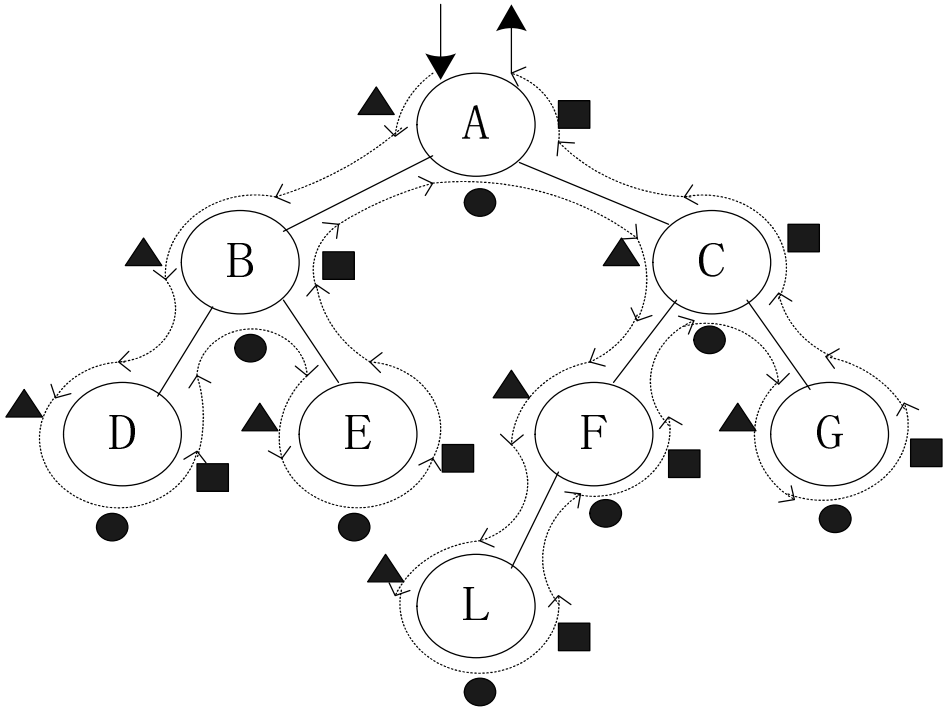
先序遍历算法执行过程

- 先序遍历算法执行过程（如图三角表示）



先序遍历算法执行过程

- 先序遍历算法执行过程（如图三角表示）
- 先序遍历结果序列为**A, B, D, E, C, F, L, G**



先序遍历算法执行过程

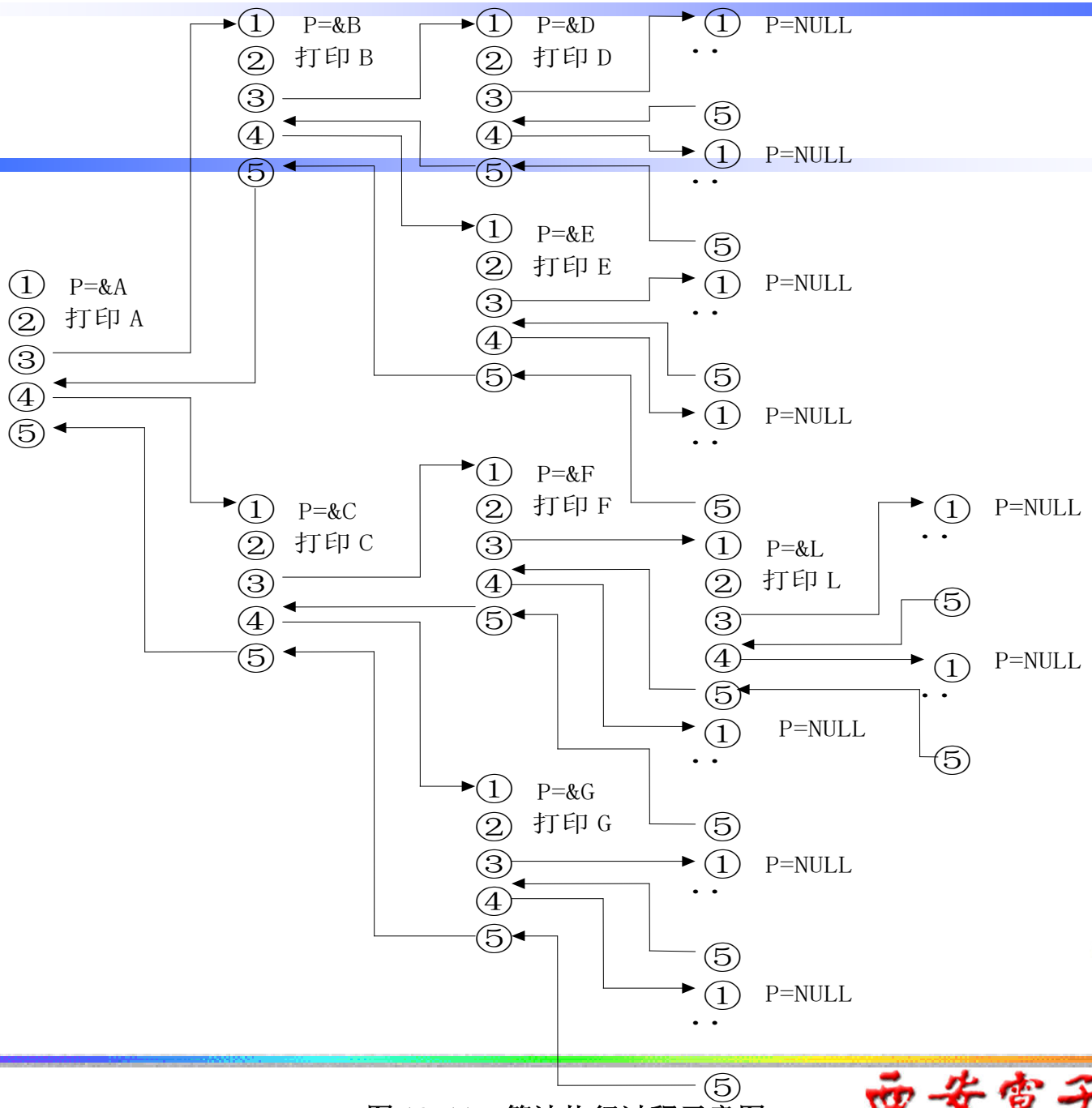
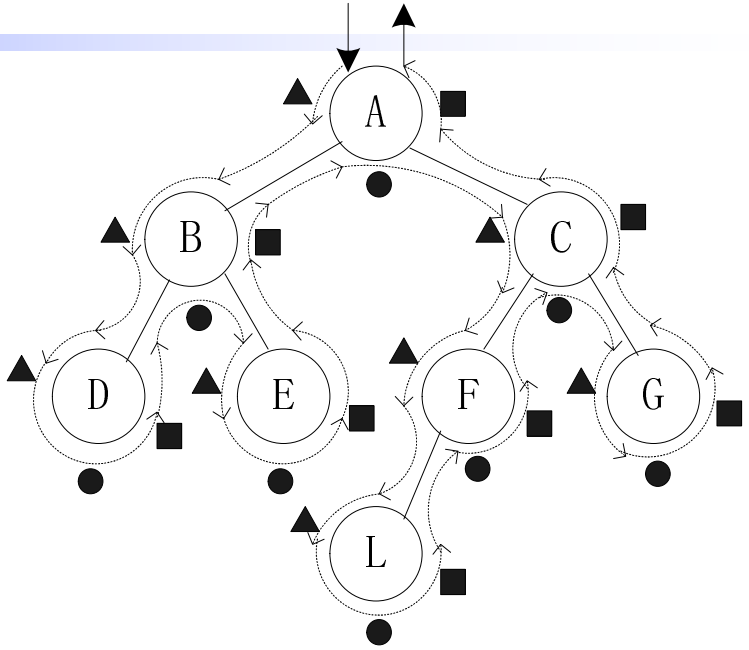


图 10.11　算法执行过程示意图

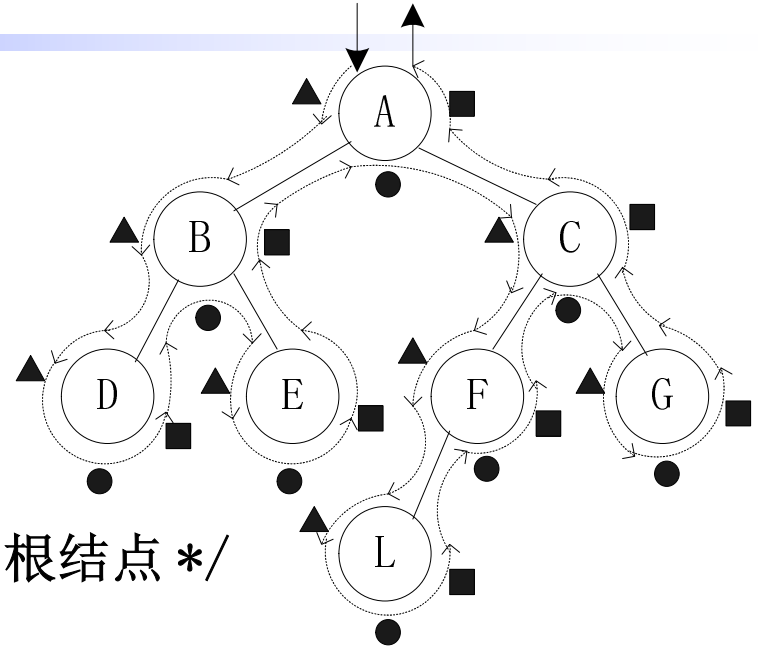
中序遍历算法

- 中序遍历算法的遍历过程是：
  若二叉树非空，执行以下操作：
  - (1) 中序遍历左子树；
  - (2) 访问根结点；
  - (3) 中序遍历右子树。



中序遍历算法

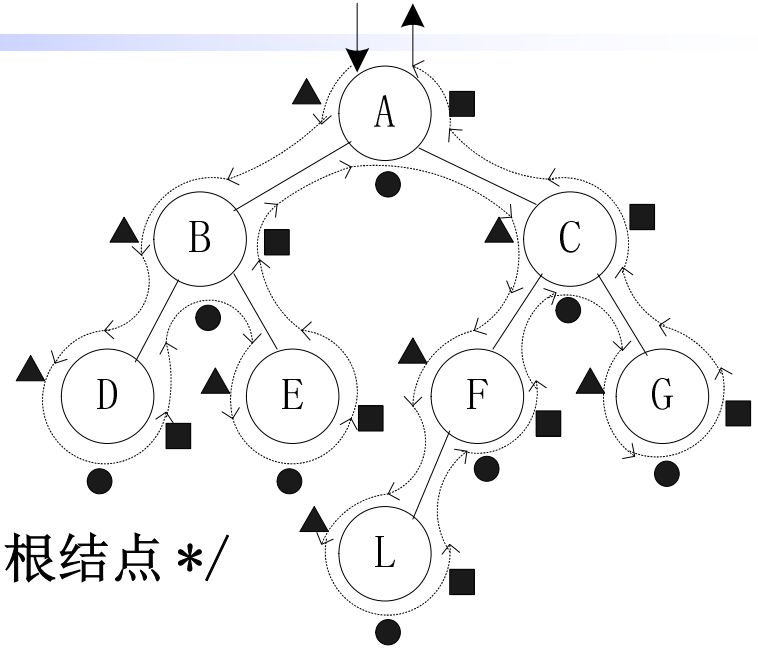
- 中序遍历算法的遍历过程是：
  若二叉树非空，执行以下操作：
  - (1) 中序遍历左子树；
  - (2) 访问根结点；
  - (3) 中序遍历右子树。
- 中序遍历算法的C语言描述如下：




```
void inorder(bitree *p)
/* 中序遍历二叉树，p指向二叉树的根结点 */
{ if (p!=NULL)
   {inorder (p→lchild);
        printf (" %c ",p→data);
    inorder (p→rchild);
   }
}   /*  inorder */
```

中序遍历算法

- 中序遍历算法的遍历过程是：
  若二叉树非空，执行以下操作：
  - (1) 中序遍历左子树；
  - (2) 访问根结点；
  - (3) 中序遍历右子树。



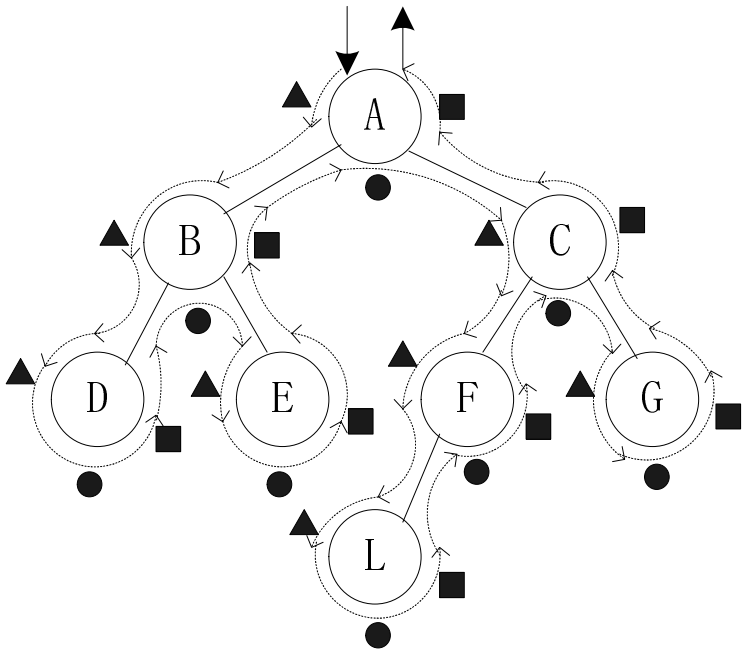
- 中序遍历算法的C语言描述如下：

```
void inorder(bitree *p)
/* 中序遍历二叉树，p指向二叉树的根结点 */
{ if (p!=NULL)
   {inorder (p→lchild);
        printf (" %c ",p→data);
    inorder (p→rchild);
   }
} /* inorder */
```

- 对右上图进行中序遍历（圆点），可得到中序遍历序列D, B, E, A, L, F, C, G。

后序遍历算法

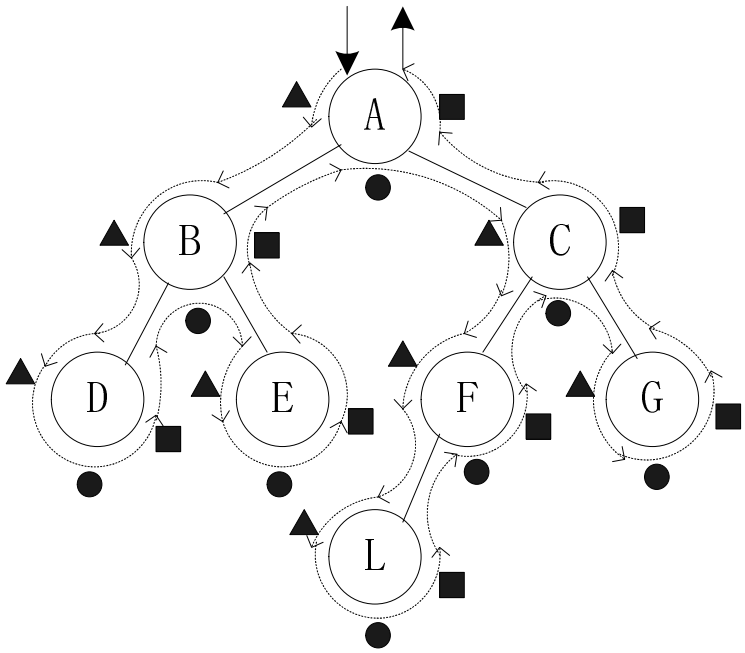
- 后序遍历算法的遍历过程是：
  若二叉树非空，执行以下操作：
  - (1) 后序遍历左子树；
  - (2) 后序遍历右子树；
  - (3) 访问根结点。



后序遍历算法

- 后序遍历算法的遍历过程是：
  若二叉树非空，执行以下操作：
  - (1) 后序遍历左子树；
  - (2) 后序遍历右子树；
  - (3) 访问根结点。
- 后序遍历算法的C语言描述如下：

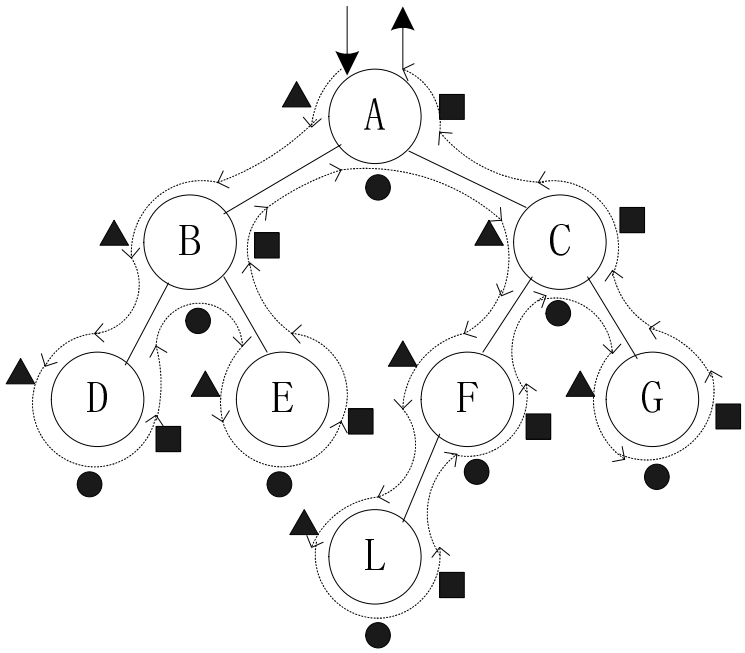
```
void postorder (bitree *p)
  /* p指向二叉树的根结点 */
{ if (p!=NULL)
  {    postorder (p→lchild);
       postorder (p→rchild);
       printf (" %c ", p→data);
  }
  }   /*  postorder */
```



后序遍历算法

- 后序遍历算法的遍历过程是：
  若二叉树非空，执行以下操作：
  - (1) 后序遍历左子树；
  - (2) 后序遍历右子树；
  - (3) 访问根结点。
- 后序遍历算法的C语言描述如下：

```
void postorder (bitree *p)
  /* p指向二叉树的根结点 */
{ if (p!=NULL)
  {    postorder (p→lchild);
       postorder (p→rchild);
       printf (" %c ", p→data);
  }
  }   /*  postorder */
```



- 对右上图进行后序遍历（方框），得到的后序遍历序列D, E, B, L, F, G, C, A。

深度优先的非递归算法

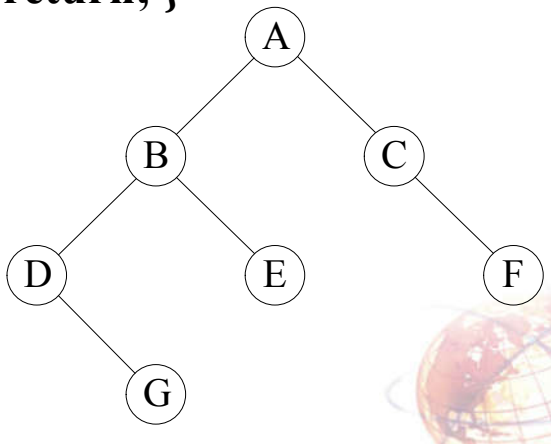
- 使用一个堆栈**stack[N]**来保存每次调用的参数，这个堆栈的栈顶指针为**top**，另设一个活动指针**p**来指向当前访问的结点。下面将讨论中序遍历的非递归算法。

深度优先的非递归算法

- 使用一个堆栈**stack[N]**来保存每次调用的参数，这个堆栈的栈顶指针为**top**，另设一个活动指针**p**来指向当前访问的结点。下面将讨论中序遍历的非递归算法。
- 中序遍历的非递归算法的基本思想是：
  - 当**p**所指的结点非空时，将该结点的存储地址进栈，然后再将**p**指向该结点的左孩子结点；
  - 当**p**所指的结点为空时，从栈顶退出栈顶元素送**p**，并访问该结点，然后再将**p**指向该结点的右孩子结点；
  - 如此反复，直到**p**为空并且栈顶指针**top = -1**为止。

深度优先的非递归算法

```
bitree     *stack[N];      /*  N为所设栈的最大容量 */
void ninorder(p)     /*  非递归中序遍历二叉树，p指向二叉树的根结点 */
bitree  *p;
{  bitree  *s;
    int top;
   if ( p!=NULL)
   {  top = －1;          s = p;
      while( ( top != －1) | | ( s != NULL) )
      {  while( s != NULL)
         {  if ( top = =N－1)          {  printf ( " overflow ");  return; }
           else {  top++;     stack[top] = s;    s = s→lchild;}
         }      //while(s!=NULL)
         s = stack[top];
         top－－;
         printf ( " % c " , s→data);
         s = s→rchild;
      }
   }
   }  /* ninorder */
```



二叉树的广度优先遍历

- 二叉树的广度优先遍历又称为按层次遍历，这种遍历方式是先遍历二叉树的第一层结点，然后遍历第二层结点，……最后遍历最下层的结点。而对每一层的遍历是按从左至右的方式进行。

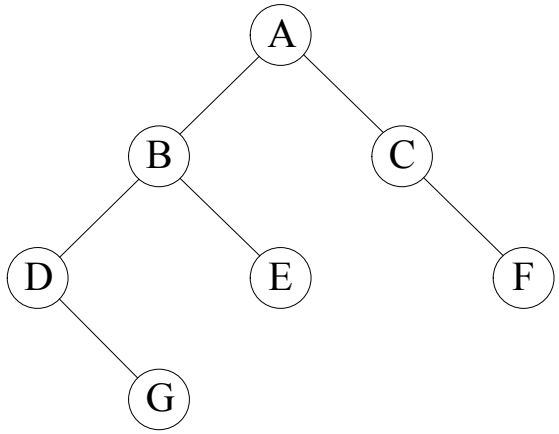
二叉树的广度优先遍历

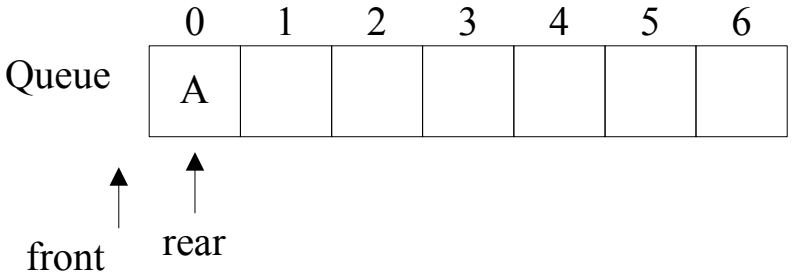
- 二叉树的广度优先遍历又称为按层次遍历，这种遍历方式是先遍历二叉树的第一层结点，然后遍历第二层结点，……最后遍历最下层的结点。而对每一层的遍历是按从左至右的方式进行。
- 基本思想：

  按照广度优先遍历方式，在上层中先被访问的结点，它的下层孩子也必然先被访问，因此在这种遍历算法的实现时，需要使用一个队列。在遍历进行之前先把二叉树的根结点的存储地址入队，然后依次从队列中出队结点的存储地址，每出队一个结点的存储地址则对该结点进行访问，然后依次将该结点的左孩子和右孩子的存储地址入队，如此反复，直到队空为止。

二叉树的广度优先遍历实例

- 以下图为例说明

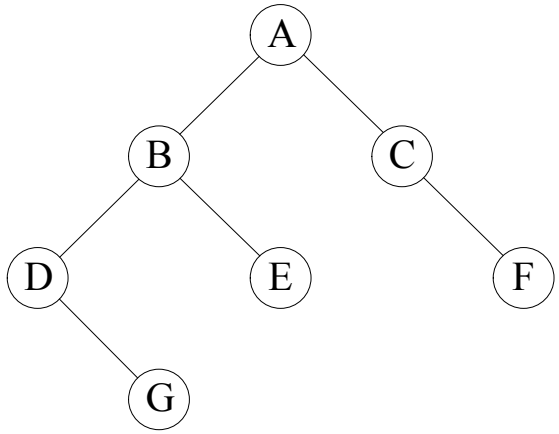


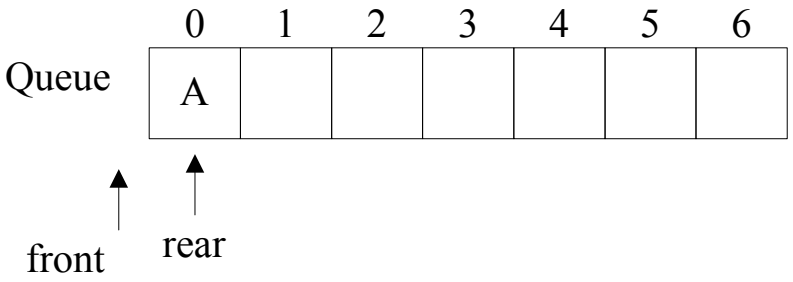


初始状态

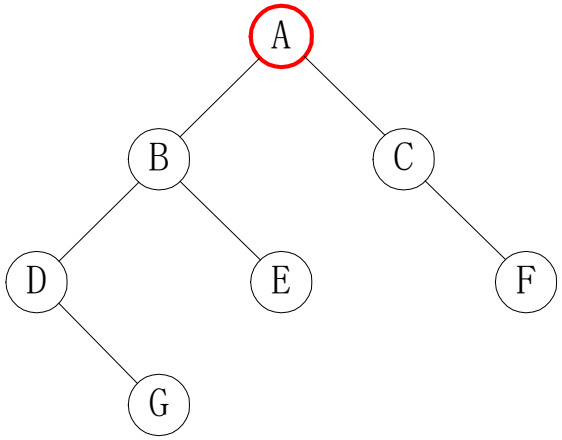
二叉树的广度优先遍历实例

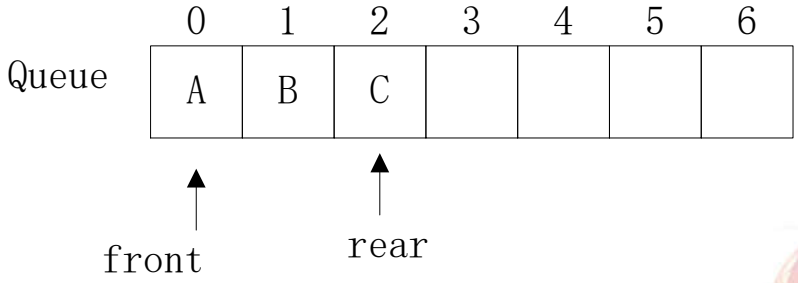
- 以下图为例说明





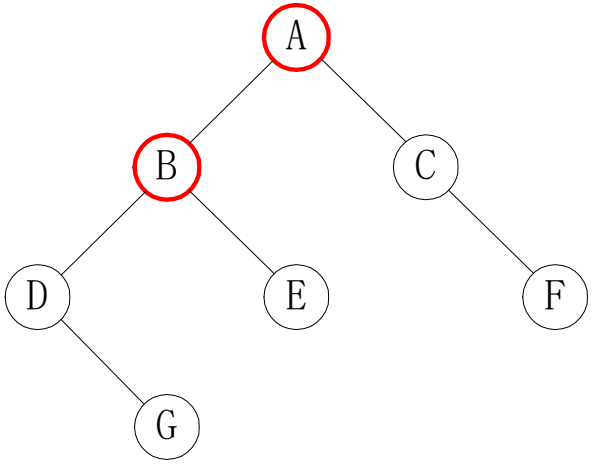
初始状态

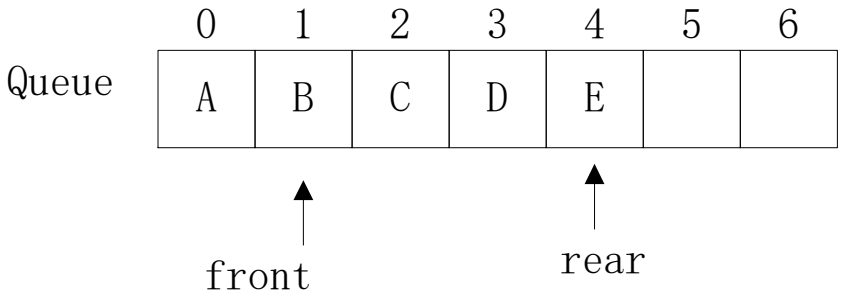




访问A，并将A的孩子入队

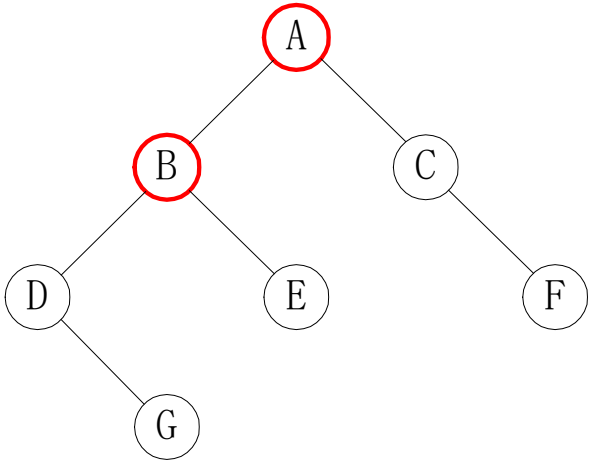
二叉树的广度优先遍历实例

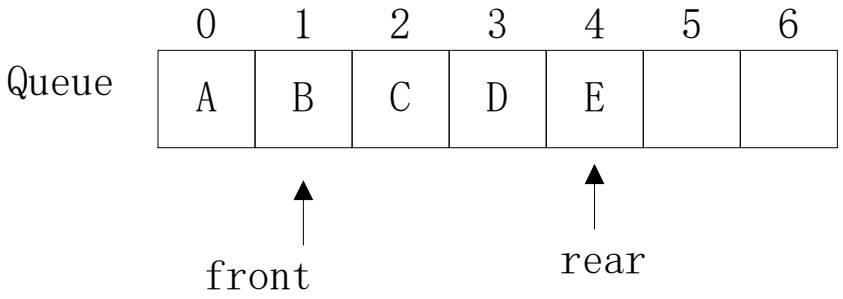




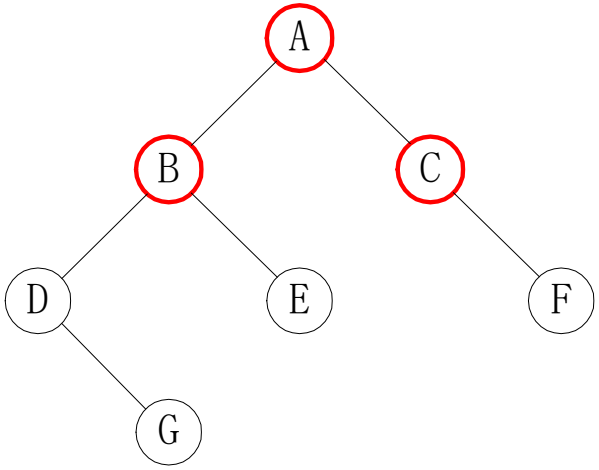
访问B，并将B的孩子入队

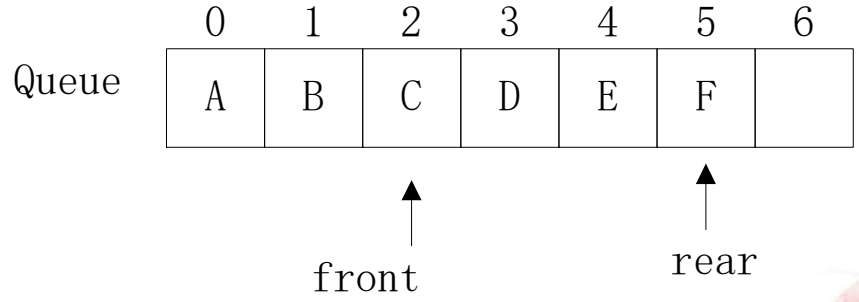
二叉树的广度优先遍历实例





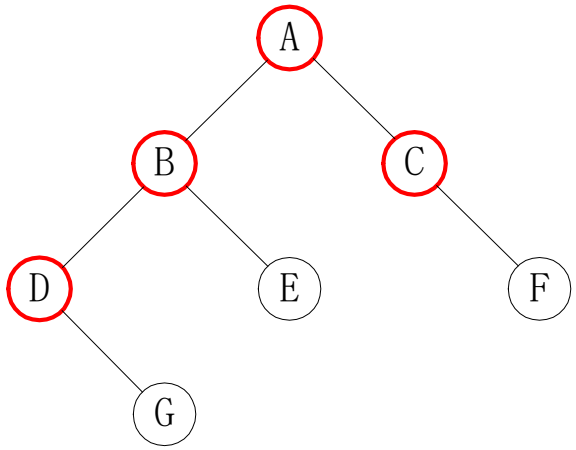
访问B，并将B的孩子入队

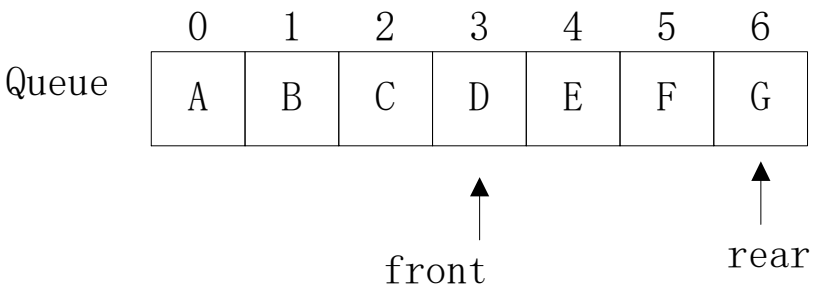




访问C，并将C的孩子入队

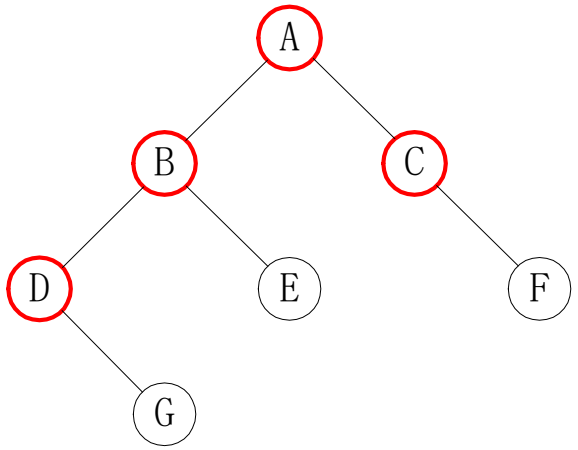
二叉树的广度优先遍历实例

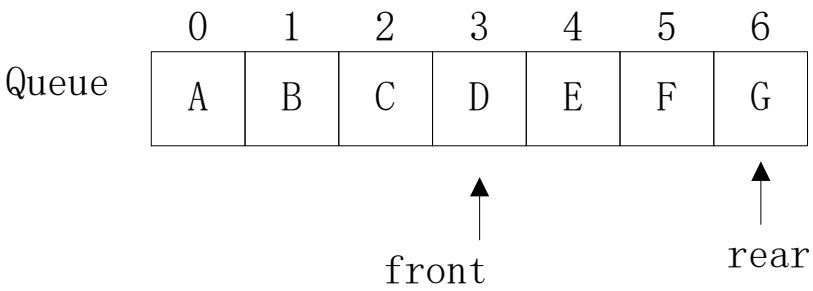




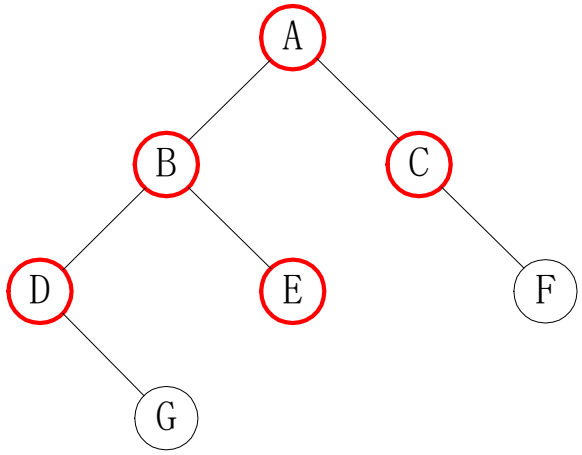
访问D，并将D的孩子入队

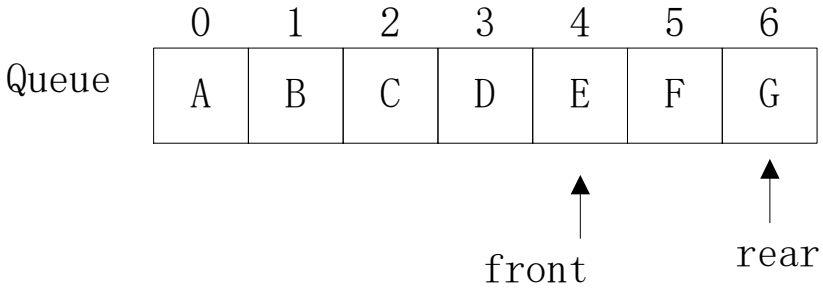
二叉树的广度优先遍历实例





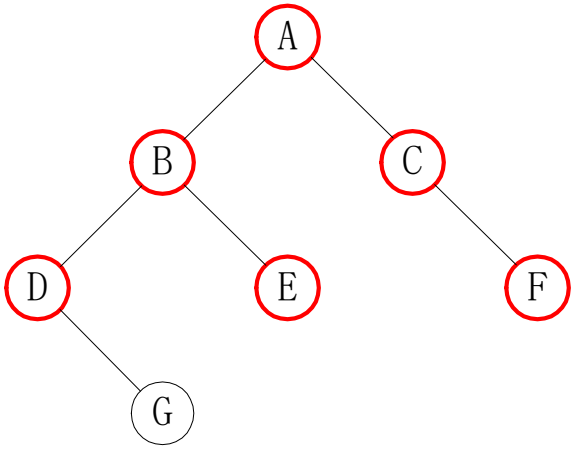
访问D，并将D的孩子入队

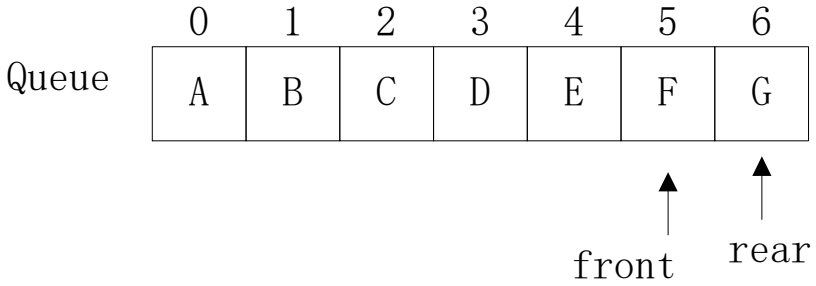




访问E，由于E的孩子为空，不入队

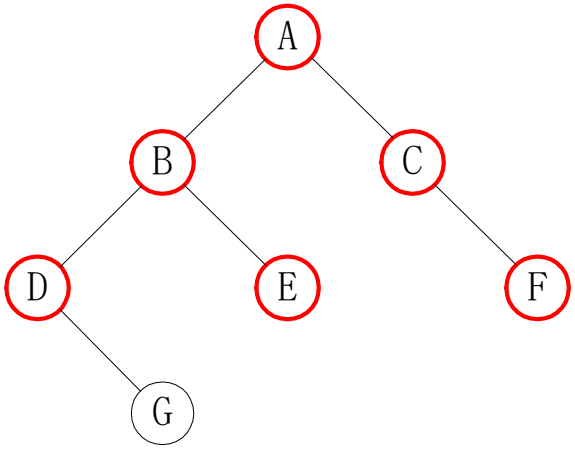
二叉树的广度优先遍历实例

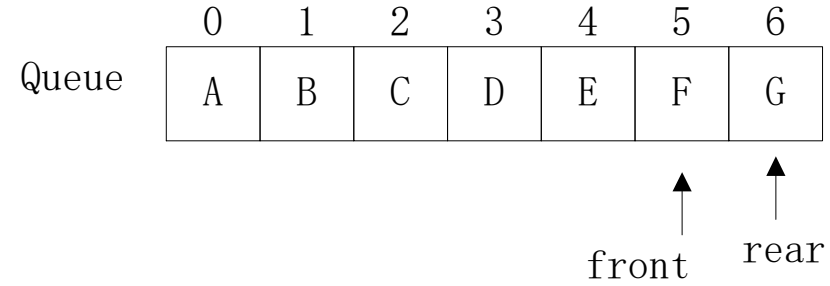




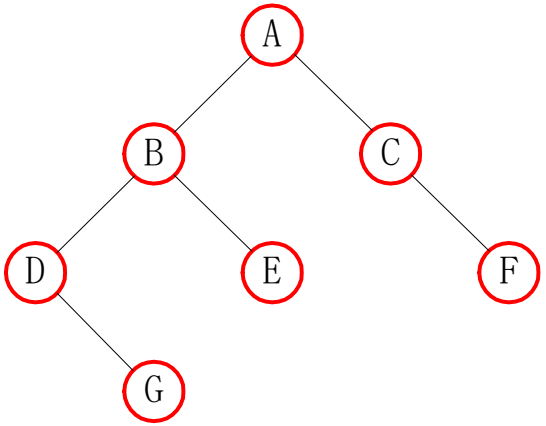
访问F，由于F的孩子为空，不入队

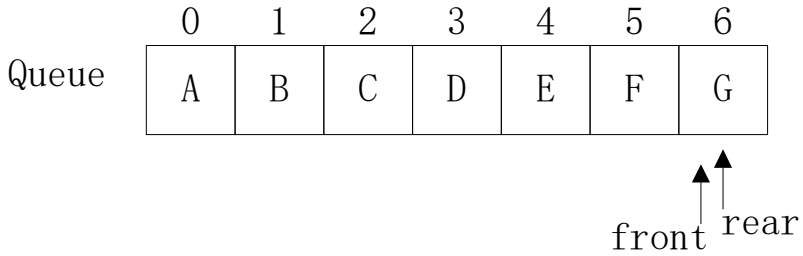
二叉树的广度优先遍历实例





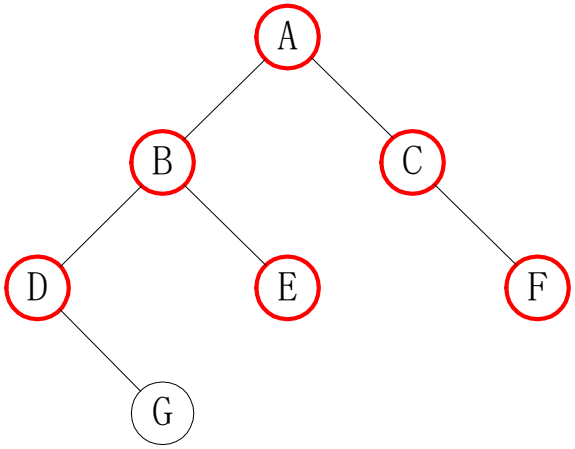
访问F，由于F的孩子为空，不入队

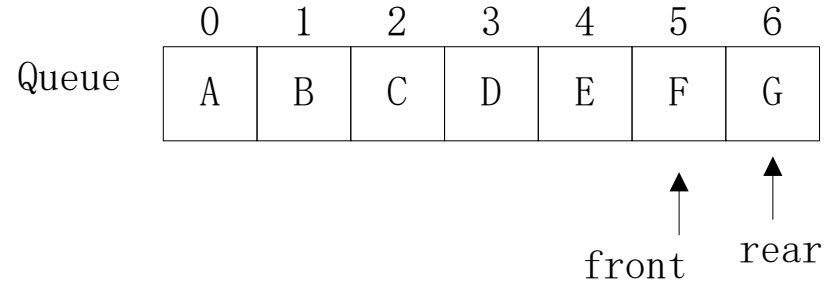




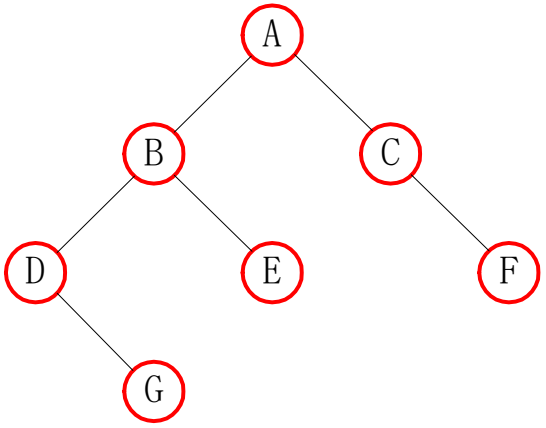
访问G，由于G的孩子为空，不入队

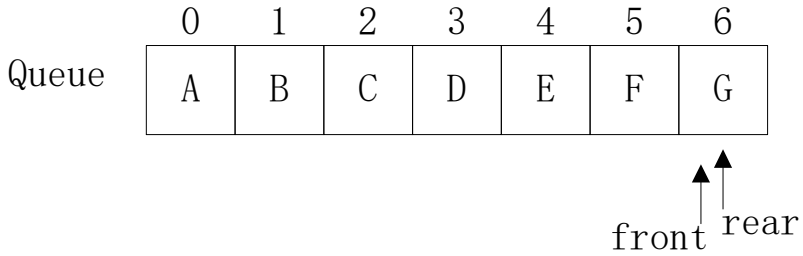
二叉树的广度优先遍历实例





访问F，由于F的孩子为空，不入队





访问G，由于G的孩子为空，不入队

- 队列为空，算法结束。输出续列为：A,B,C,D,E,F,G

二叉树的广度优先遍历算法

```
bitree   *Q[maxsize];      /*  设置指针类型数组来构成队列 */
void Layer (bitree *p)     /*  层次遍历二叉树，p指向二叉树的根结点 */
{ bitree *s;
  if ( p!=NULL )
    {rear=0;               /*  队尾指针置初值 */
     front=-1;             /*  队头指针置初值 */
     Q[rear]=p;            /*  根结点入队 */
     while ( front<rear )  /*  当队非空时, 执行以下操作 */
     { front ++;
       s=Q[front];         /*  从队列中出队队头结点 */
       printf (" %c " , s→data);   /*  访问出队结点 */
       if ( s→lchild!=NULL)        /*s所指结点的左孩子结点入队 */
         { rear++;
           Q[rear]=s→lchild;
         }
       if (s→rchild!=NULL)
         { rear++;         /*  s所指的右孩子结点入队 */
           Q[rear]=s→rchild;
         }
     }
    }
  }  /* Layer */
```

从遍历序列恢复二叉树

- 在已知结点的遍历序列的条件下，来恢复相应的二叉树的问题。
  - 基本思想：
    - 二叉树的先序遍历是先访问根结点，然后按先序遍历方式遍历根结点的左子树，最后按先序遍历方式遍历根结点的右子树，所以在先序遍历序列中，第一个结点必定是二叉树的根结点。
    - 中序遍历是先按中序遍历方式遍历根结点的左子树，然后访问根结点，最后按中序遍历方式遍历根结点的右子树。
    - 在中序遍历序列中，已知的根结点将中序序列分割成两个子序列。
    - 在根结点左边的子序列，是根结点的左子树的中序序列，而在根结点右边的子序列，是根结点右子树的中序序列。
    - 再经过先序序列中对应的左子序列确定其第一个结点是左子树的根结点；通过先序序列中对应的右子树序列确定其第一个结点是右子树的根结点。
    - 这样就确定了二叉树的根结点和左子树及右子树的根结点。利用左子树和右子树的根结点，又可分别将左子序列和右子序列划分成两个子序列。
    - 如此反复，直到取尽先序序列中的结点时，便得到一棵二叉树。

从遍历序列恢复二叉树

- 在已知结点的遍历序列的条件下，来恢复相应的二叉树的问题。
  - 基本思想：
    - 二叉树的先序遍历是先访问根结点，然后按先序遍历方式遍历根结点的左子树，最后按先序遍历方式遍历根结点的右子树，所以在先序遍历序列中，第一个结点必定是二叉树的根结点。
    - 中序遍历是先按中序遍历方式遍历根结点的左子树，然后访问根结点，最后按中序遍历方式遍历根结点的右子树。
    - 在中序遍历序列中，已知的根结点将中序序列分割成两个子序列。
    - 在根结点左边的子序列，是根结点的左子树的中序序列，而在根结点右边的子序列，是根结点右子树的中序序列。
    - 再经过先序序列中对应的左子序列确定其第一个结点是左子树的根结点；通过先序序列中对应的右子树序列确定其第一个结点是右子树的根结点。
    - 这样就确定了二叉树的根结点和左子树及右子树的根结点。利用左子树和右子树的根结点，又可分别将左子序列和右子序列划分成两个子序列。
    - 如此反复，直到取尽先序序列中的结点时，便得到一棵二叉树。

先序和中序、中序和后序可以唯一确定二叉树

从遍历序列恢复二叉树实例

- 已知一棵二叉树的先序序列为A, B, D, G, C, E, F, H，而中序序列为D, G, B, A, E, C, H, F，则可按以下方式来确定相应的二叉树。
  - 首先从先序序列可确定A是二叉树的根结点，再根据A在中序序列中的位置，可知结点DGB在A的左子树上和ECHF在A的右子树上；
  - 根据先序序列确定B和C分别是A的左子树和右子树的根，再根据B和C在DGB和ECHF中的位置，可知DG在B的左子树上和B的右子树为空以及C的左子树仅有结点E和C的右子树包含结点HF；
  - 根据先序序列可知D是B的左子树的根和G是D的右子树的根，E是C的左子树的根和F是C的右子树的根；
  - 最后确定H是F的左子树的根。
  - 构造过程如下图所示。

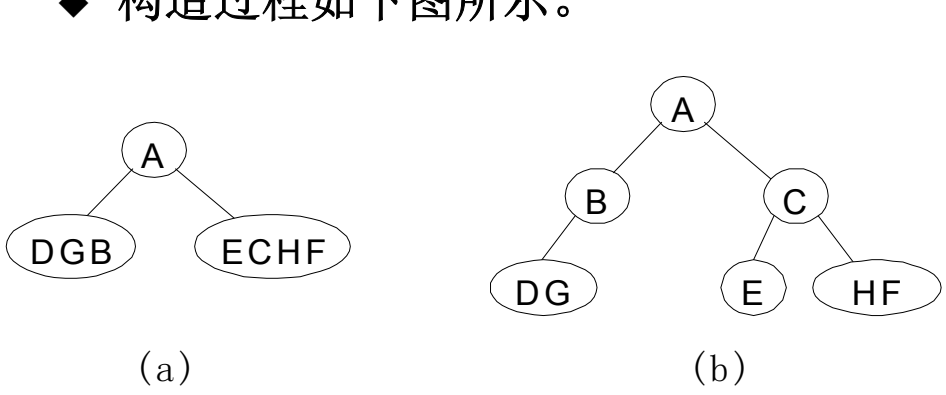


图 10.12　由先序和中序序列构造二叉树的过程

遍历序列恢复二叉树算法思想

- 递归算法的基本思想是：
  - 先将先序和中序序列分别存在两个数组perod[n]和inod[n]中，
  - 取先序序列的第一个元素建立整个树的根结点，并利用中序序列确定根结点的左、右子树结点在先序序列中的位置。
  - 接着再用先序序列和中序序列分别对左子树和右子树进行构造。

遍历序列恢复二叉树算法

```
datatype preod[maxsize], inod[maxsize];   /* 设置先序与中序序列存放数组 */
bitree    *BPI(datatype    preod[ ], datatype inod[ ],int ps,int pe,int is,int ie)
 /*  ps,pe,is,ie是要构造的二叉树的先序和中序序列数组的起始和终点下标 */
{int  m;  bitree *p;
 p = (bitree*)malloc(sizeof(bitree));          /* 构造根结点 */
 p→data=preod[ps];
 m=is;
 while ( inod[m] != preod[ps] )
        m++;                                   /* 查找根结点在中序序列中的位置 */
 if ( m==is )        p→lchild = NULL;      /* 对左子树进行构造 */
 else       p→lchild = BPI( preod, inod, ps+1, ps+m-is, is, m-1 );
 if ( m==ie)         p→rchild=NULL;      /* 对右子树进行构造 */
 else  p→rchild = BPI( preod, inod, ps+m-is+1, pe, m+1,ie);
 return p;
}   /* BPI  */
```

遍历序列恢复二叉树算法

```
datatype preod[maxsize], inod[maxsize];   /* 设置先序与中序序列存放数组 */
bitree    *BPI(datatype    preod[ ], datatype inod[ ],int ps,int pe,int is,int ie)
  /*  ps,pe,is,ie是要构造的二叉树的先序和中序序列数组的起始和终点下标 */
{int  m;  bitree *p;
  if(pe<ps) return NULL;
  if(ie<is) return NULL;
  p = (bitree*)malloc(sizeof(bitree));          /* 构造根结点 */
  p→data=preod[ps];
  m=is;
  while ( inod[m] != preod[ps] )
          m++;                                  /* 查找根结点在中序序列中的位置 */
 p→lchild = BPI( preod, inod, ps+1, ps+m-is, is, m-1 ); /* 对左子树进行构造 */
 p→rchild = BPI( preod, inod, ps+m-is+1, pe, m+1,ie); /* 对右子树进行构造 */
  return p;
}   /* BPI  */
```

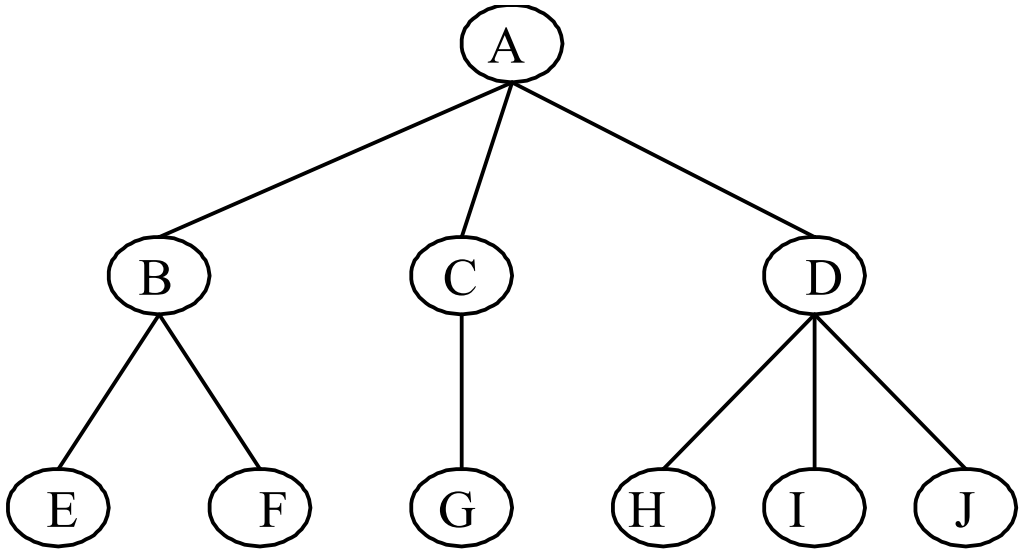
遍历算法的应用

- 统计一棵二叉树中的叶子结点数(叶子结点是二叉树中那些左孩子和右孩子均不存在的结点)。下面给出利用中序遍历实现的算法。

```
int         count =0;
int         countleaf(p)
bitree      *p;
{if ( p != NULL ){
            count = countleaf( p→lchild );  /* 对左子树上的叶子结点计数*/
            if ( (p→lchild= =NULL)&&(p→rchild= =NULL))
                                    count=count+1;
            count = countleaf(p→rchild);   /* 对右子树上的叶子结点计数*/
 }
return  count;
}  /* countleaf  */
```

树和森林

- 讨论树的存储表示，并建立森林与二叉树的对应关系。



树的表示

树的存储结构

1．双亲表示法　树中每个结点的双亲是唯一的，可在存储结点信息的同时，为每个结点存储其双亲结点的地址信息。

树的存储结构

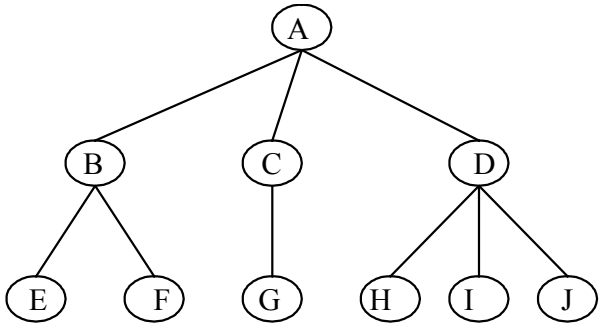
1．双亲表示法　树中每个结点的双亲是唯一的，可在存储结点信息的同时，为每个结点存储其双亲结点的地址信息。

```
#define maxsize  32    /* 结点数目的最大值加1 */
 typedef struct
  {  datatype data;      /* 数据域 */
     int parent ;        /* 双亲结点的下标 */
  }ptree;
  ptree T[maxsize];
```

树的存储结构

1．双亲表示法　树中每个结点的双亲是唯一的，可在存储结点信息的同时，为每个结点存储其双亲结点的地址信息。

```
#define maxsize  32    /* 结点数目的最大值加1 */
 typedef struct
  {  datatype data;      /* 数据域 */
     int parent ;        /* 双亲结点的下标 */
  }ptree;
  ptree T[maxsize];
```



树的双亲表示法

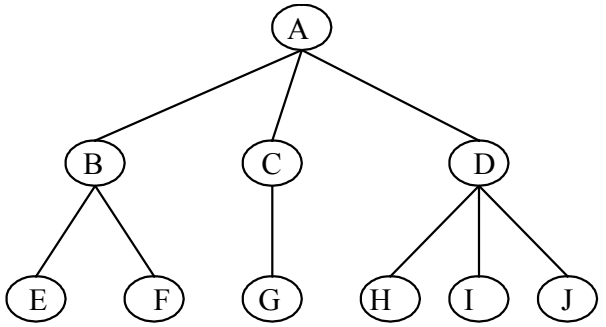
结点	0	1	2	3	4	5	6	7	8	9	10
data		A	B	C	D	E	F	G	H	I	J
parent	-1	0	1	1	1	2	2	3	4	4	4

树的存储结构

1．双亲表示法　树中每个结点的双亲是唯一的，可在存储结点信息的同时，为每个结点存储其双亲结点的地址信息。

```
#define maxsize  32    /* 结点数目的最大值加1 */
 typedef struct
  {  datatype data;      /* 数据域 */
     int parent ;        /* 双亲结点的下标 */
  }ptree;
  ptree T[maxsize];
```

——静态链表



树的双亲表示法

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

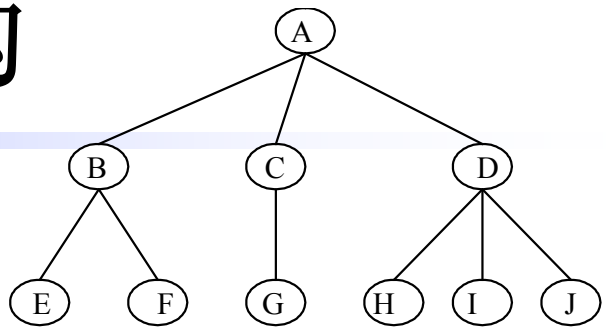
树的存储结构

2．孩子表示法　为树中每个结点建立一个孩子链表，类型说明如下：

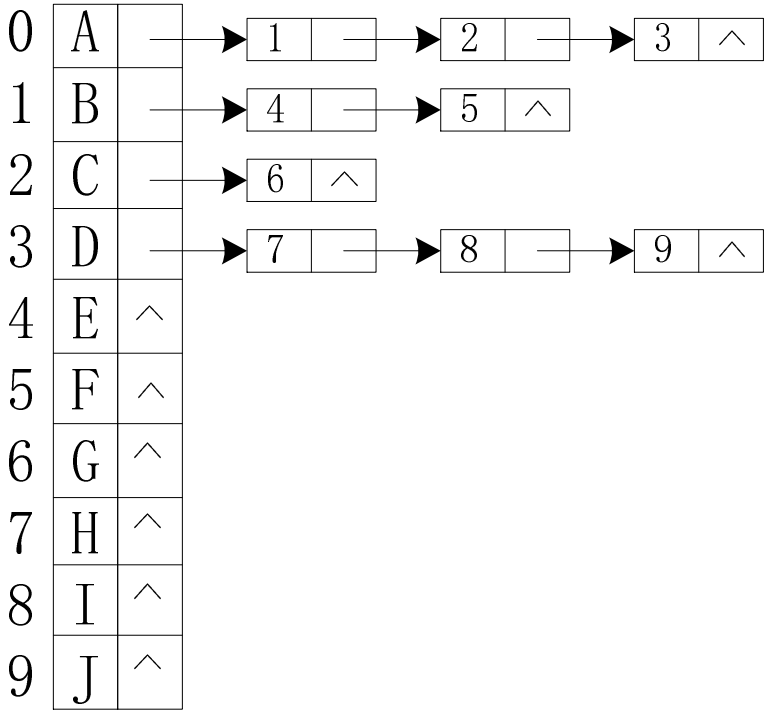
```
typedef struct cnode
 {  int child;          /* 孩子结点序号 */
    struct cnode *next;
 }link;                /* 孩子链表结点 */

 typedef struct
 {  datatype data;      /* 树结点数据 */
    int parent;         /* 双亲指针，双亲孩子表示法中定义 */
    link *headptr;      /* 孩子链表头指针 */
 }ctree;
 ctree T[maxsize];
```

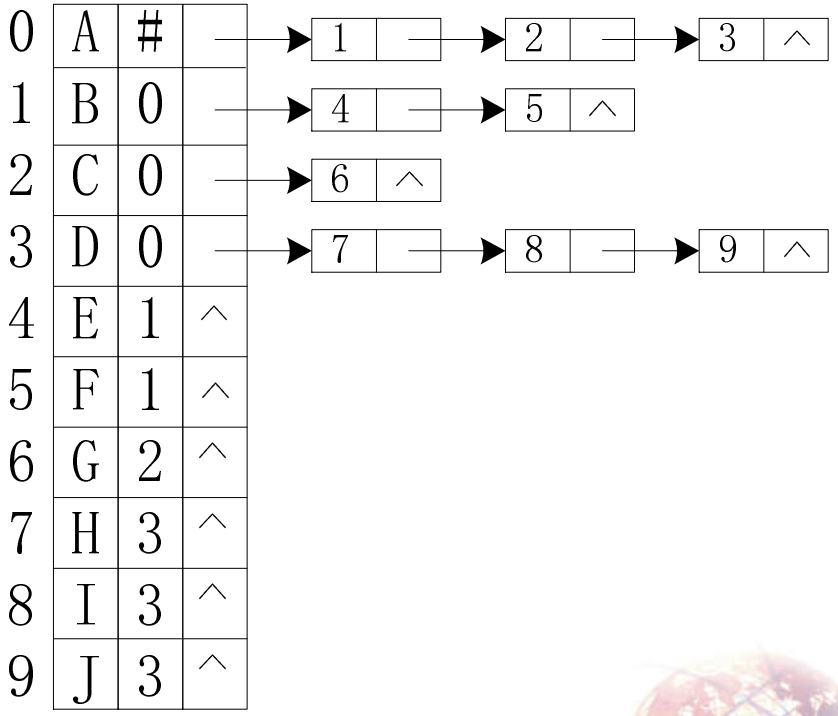
树的存储结构



- 树的孩子表示法



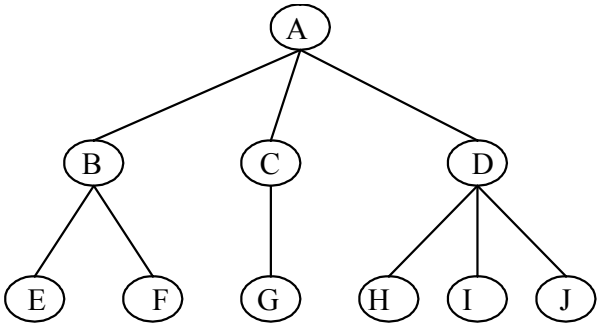
(a)孩子表示法

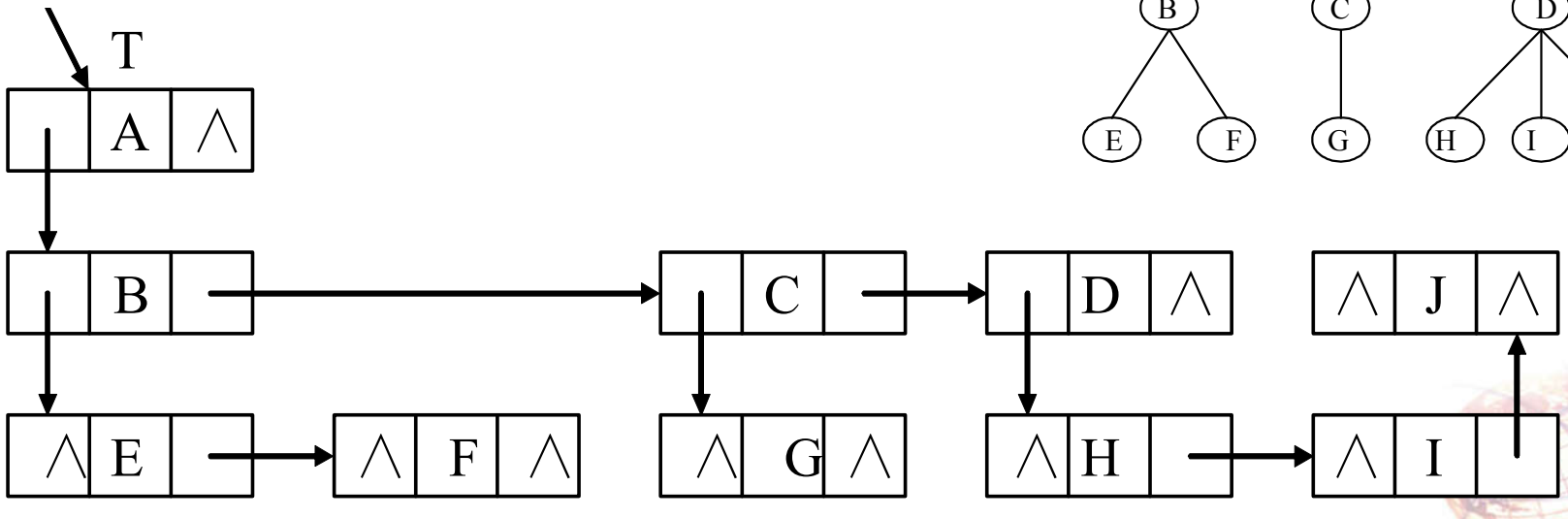


(b)孩子双亲表示法

树的存储结构

3．孩子兄弟表示法　在存储结点信息的同时，附加两个分别指向该结点最左孩子和右邻兄弟的指针域**first**和**next**。（和二叉链表表示一样）





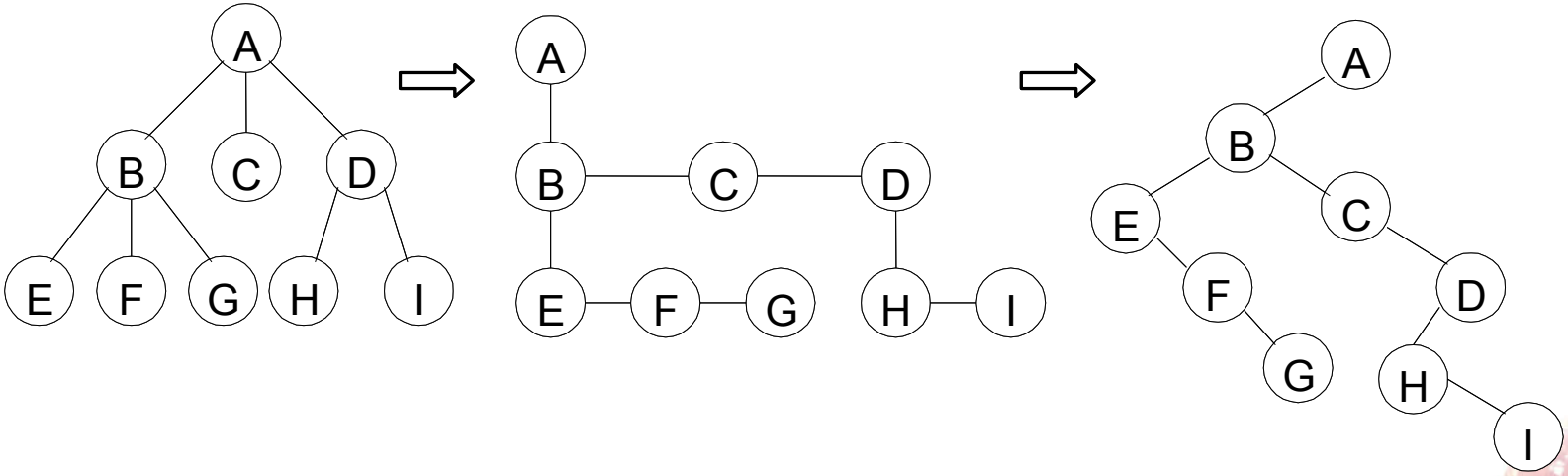
树、森林和二叉树之间的转换

将一棵树转换为二叉树的方法是：

(1)在兄弟（非堂兄弟）之间增加一条连线；

(2)对每个结点，除了保留与其左孩子的连线外，除去与其它孩子之间的连线；

(3)以树的根结点为轴心，将整个树顺时针旋转45度。



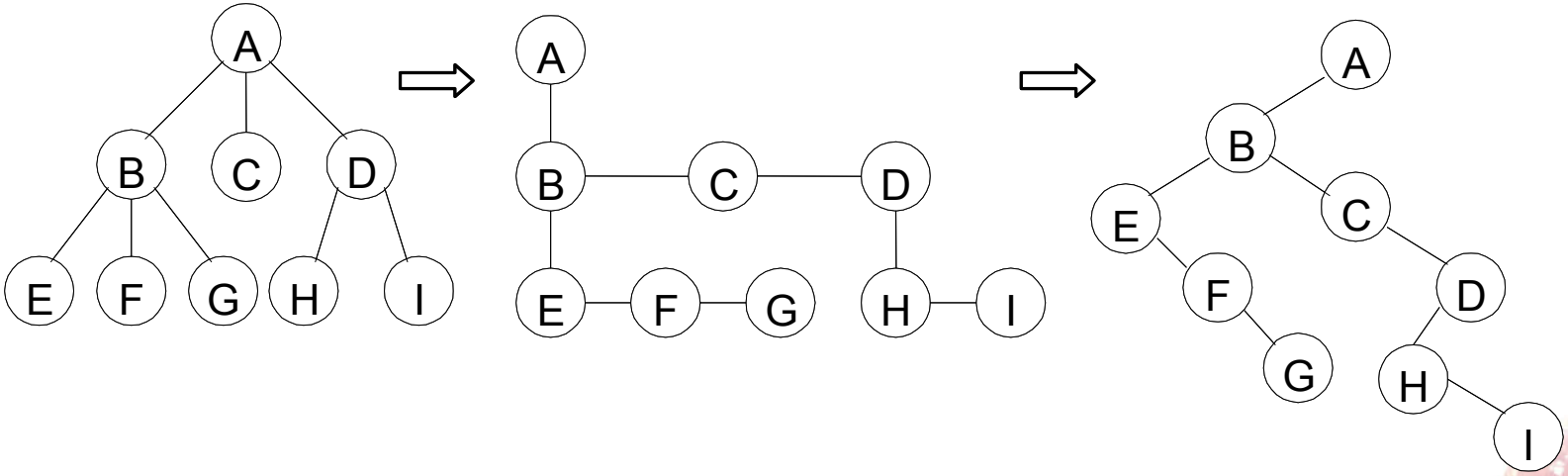
树、森林和二叉树之间的转换

将一棵树转换为二叉树的方法是：

(1)在兄弟（非堂兄弟）之间增加一条连线；

(2)对每个结点，除了保留与其左孩子的连线外，除去与其它孩子之间的连线；

(3)以树的根结点为轴心，将整个树顺时针旋转45度。



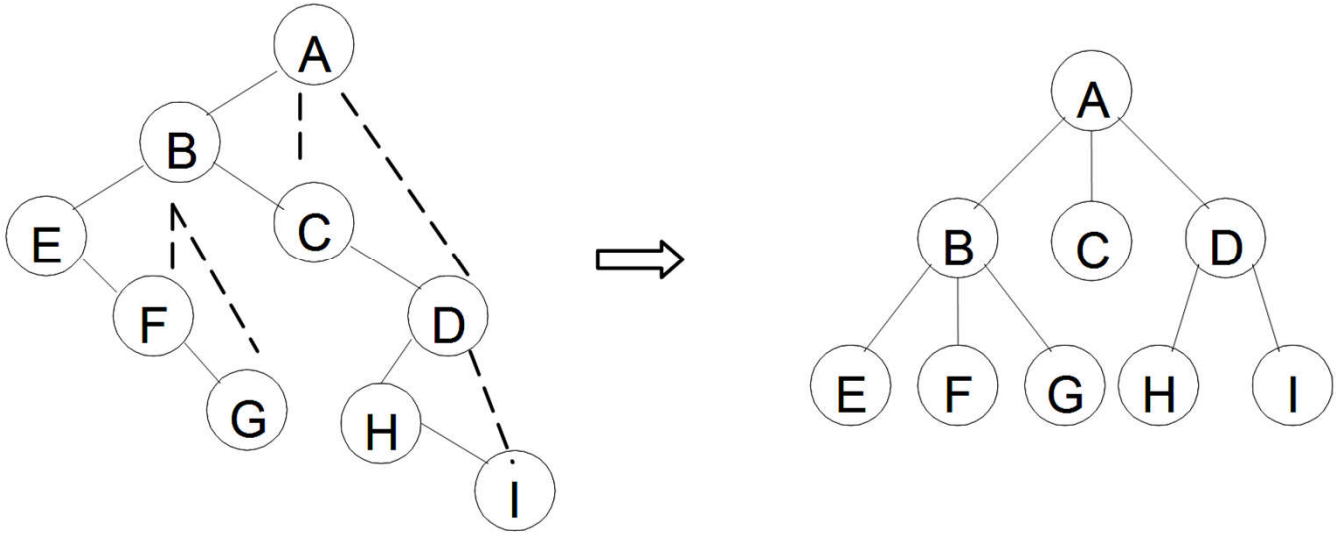
任何一棵树转化为对应的二叉树后，二叉树的右子树为空。

树、森林和二叉树之间的转换

将一棵二叉树转换成树的规则是：

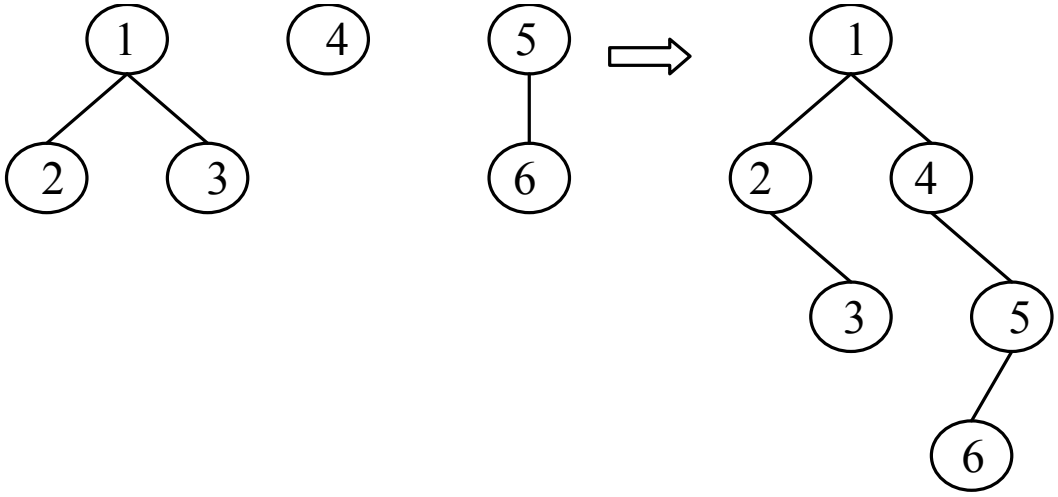
(1)若结点X是双亲Y的左孩子，则把X的右孩子，右孩子的右孩子…都与Y用连线相连。

(2)去掉原有的双亲到右孩子的连线。



树、森林和二叉树之间的转换

- 将一森林转换成二叉树的方法是：
  - 先将森林中的每一棵树转换为二叉树，
  - 再将第一棵树的根作为转换后二叉树的根，第一棵树的左子树作为转换后二叉树根的左子树，
  - 第二棵树作为转换后二叉树的右子树，第三棵树作为转换后的二叉树根的右子树的右子树，如此类推下去。



二叉树的应用

- 通信编码中的应用
  - 哈夫曼树及其编码、译码
- 排序中的应用
  - 二叉排序树

二叉树的应用

- 通信编码
  - 文件中共有100000个字符,其中只有a,b,c,e,d,f共6中字符,每种字符出现的频率如下表,采用定长码和变长码方案的文件编码总长度分别为300000位和224000位.

	a	b	c	d	e	f
频率(千次)	45	13	12	16	9	5
定长码	000	001	010	011	100	101
变长码	0	101	100	111	1101	1100

二叉树的应用

- 译码
  - 前缀码
    - 任何一个字符的代码都不是其它字符代码的前缀,称这种性质为编码的前缀性质,简称为前缀码.
  - 编码的前缀性质保证了译码过程的正确性

	a	b	c	d	e	f
频率(千次)	45	13	12	16	9	5
定长码	000	001	010	011	100	101
变长码	0	101	100	111	1101	1100

哈夫曼树的定义

- 定义
  - 若树中的两个结点之间存一条路径，则路径的长度是指路径所经过的边(即连接两个结点的线段)的数目。
  - 树的路径长度是树根到树中每一结点的路径长度之和。
  - 树的带权路径长度为树中所有叶子结点的带权路径长度之和，记作：

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

其中**n**为树中叶子结点的数目，w_i为叶子结点**i**的权值，l_i为叶子结点**i**到根结点之间的路径长度。

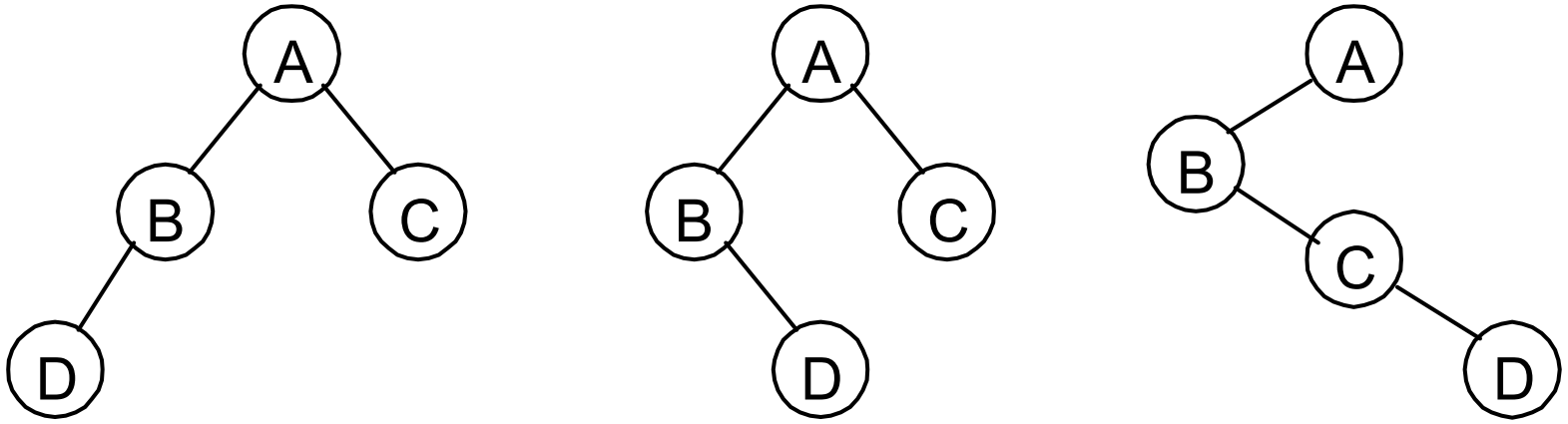
  - 在有**n**个带权叶子结点的所有二叉树中，带权路径长度**WPL**最小的二叉树被称为最优二叉树或哈夫曼树。
- 问题

在结点数目相同的二叉树中，完全二叉树的路径长度最短。如果对于一般二叉树，以带权路径长度为度量，如何构造二叉树才能使其路径长度最短？

路径长度示例

- 设路径长度用**PL**表示，则 **(a)**、**(b)**、**(c)** 三棵二叉树的路径长度分别为：

  PL=0+1+1+2=4；(b) PL=0+1+1+2=4；(c) PL=0+1+2+3=6;

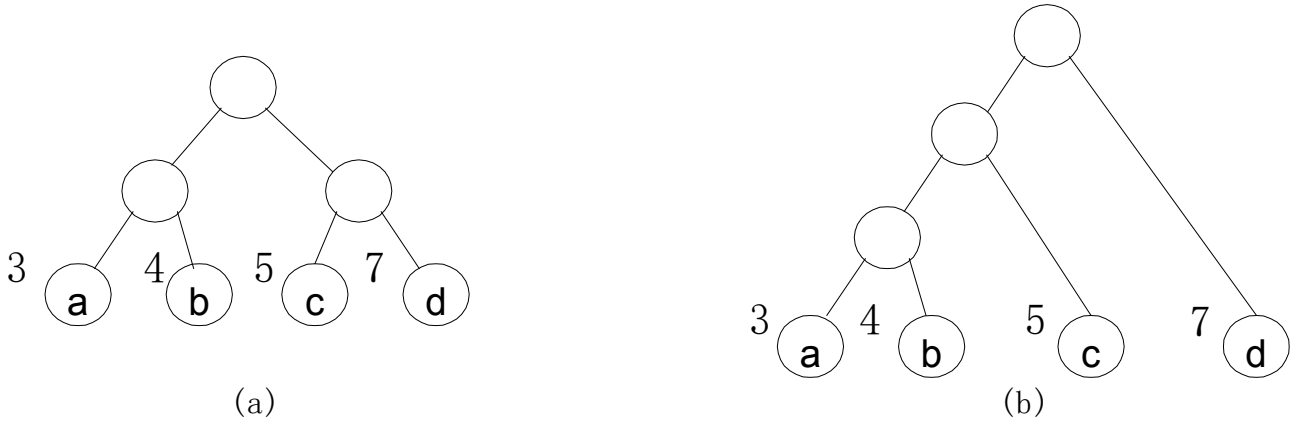


(a)　　(b)　　(c)

图 9-21　四结点构成的三种二叉树

哈夫曼树的不惟一性

- 权值为$w_1,w_2,\ldots,w_n$ 的n个叶子结点形成的二叉树，可以具有多种形态，其中能被称为哈夫曼树的二叉树并不是惟一的。如下图。

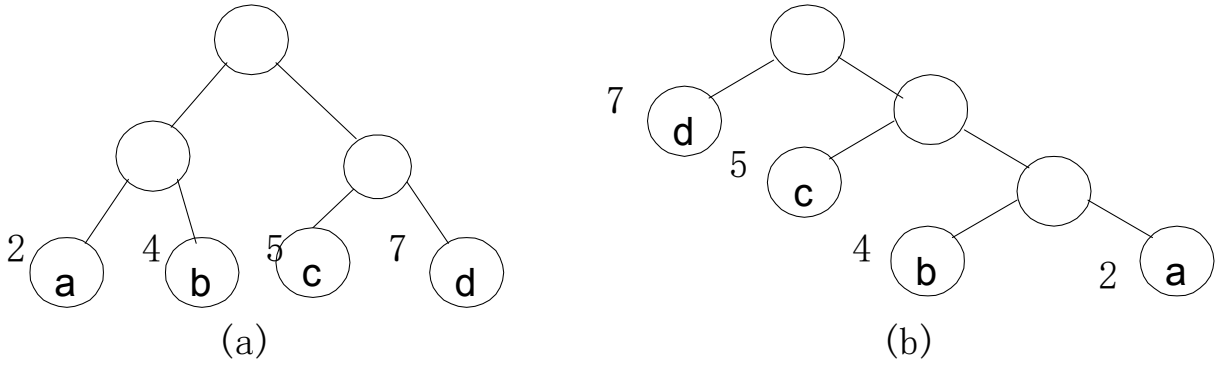


不同形态的哈夫曼树

(a) WPL =3×2+4×2+5×2+7×2=38;
(b) WPL =3×3+4×3+5×2+7=38。

完全二叉树不一定是哈夫曼树

- 在叶子数和权值相同的二叉树中，完全二叉树不一定是最优二叉树。如下图。



完全二叉树与哈夫曼树

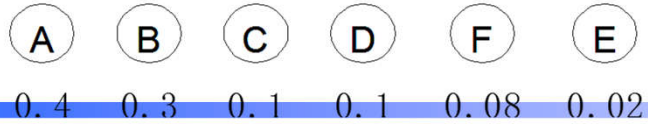
(a) WPL＝2×2＋4×2＋5×2＋7×2＝36；
(b) WPL＝2×3＋4×3＋5×2＋7×1＝35。

哈夫曼树的构造算法

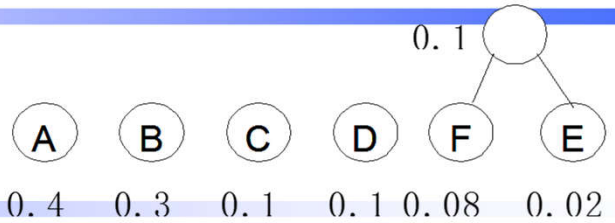
- (1)根据给定的n个权值{w_1, w_2,…,w_n}构成n棵二叉树的集合F＝{T_1,T_2,…, T_n }，其中T_i中只有一个权值为w_i的根结点，左、右子树均为空。
- (2)在F中选取两棵根结点的权值为最小的树作为左、右子树构造一棵新的二叉树，且置新的二叉树的根结点的权值为左、右子树上根结点的权值之和。
- (3)在F中删除这两个棵权值为最小的树，同时将新得到的二叉树加入F中。
- (4) 重复(2)、(3)直到F中仅剩一棵树为止。这棵树就是哈夫曼树。

哈夫曼树的构造实例

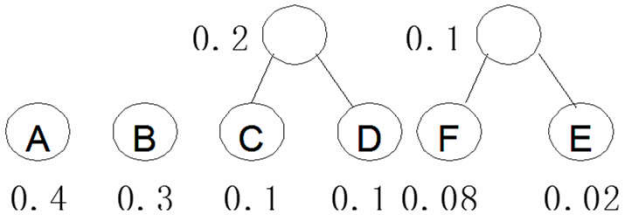
- 权值为0.4、0.3、0.1、0.1、0.02、0.08的6个叶子结点A、B、C、D、E、F构造哈夫曼树来观察哈夫曼树的构造过程，如下图所示。
- 注：
  - 一个有n个叶子结点的初始集合，要生成哈夫曼树共要进行n-1次合并，产生n-1个新结点。
  - 最终求得的哈夫曼树共有2n-1（n+n-1）个结点，并且哈夫曼树中没有度为1的分支结点。
- 我们常称没有度为1的结点的二叉树为严格二叉树。



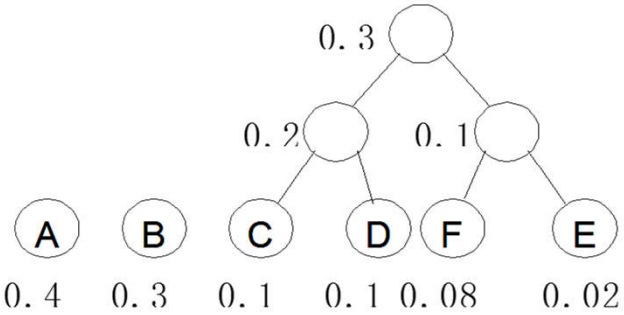
(a)初始集合 F



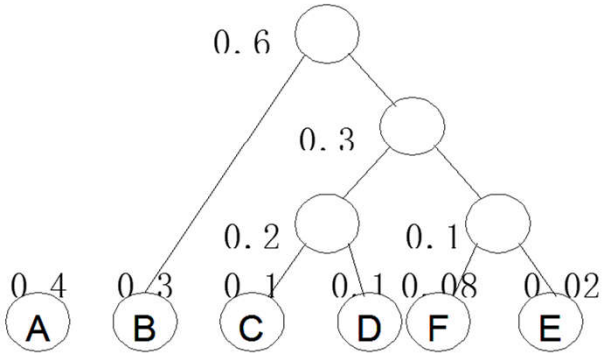
(b)E 和 F 合并



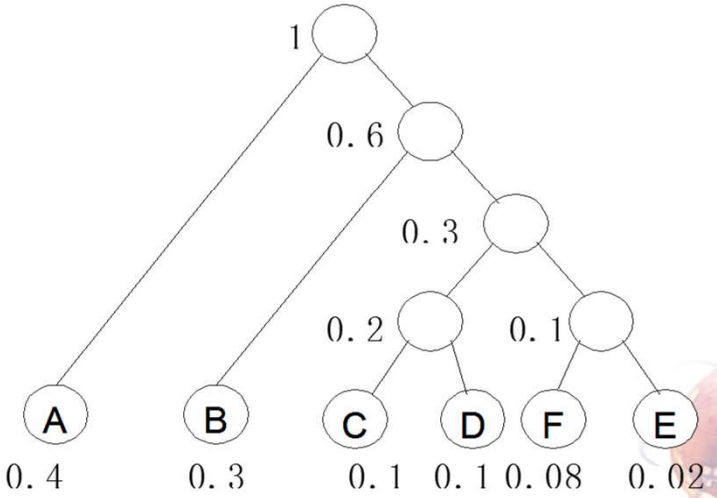
(c)C 和 D 合并



(d)E 和 F 与 C 和 D 合并



(e)B 与 E、F、C、D 的合并



(f)A、B、C、D、E、F 的合并

图 10.16　哈夫曼树的构造过程

构造哈夫曼树算法中的数据结构

- 存储结构

```
#define n 叶子数目
#define  m  2*n-1  /*  结点总数 */
typedef      char datatype;
typedef      struct
{float weight;
  datatype data;
  int lchild, rchild, parent;
} hufmtree;
hufmtree  tree[m];
```

lchild	data 结点值	weight 权值	rchild	parent

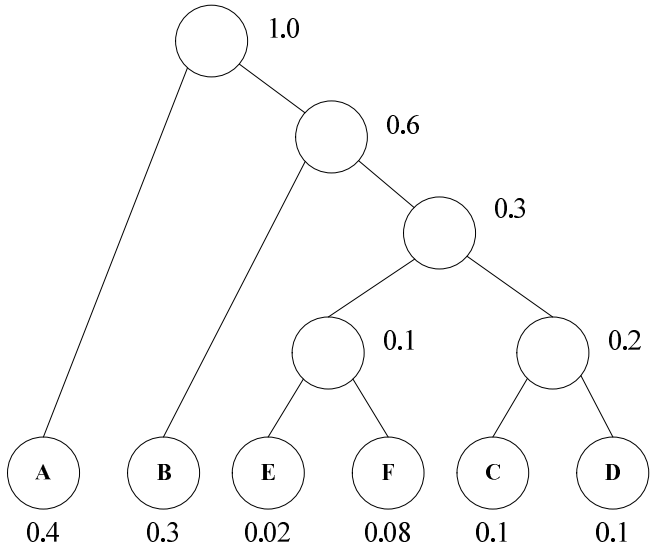
图 10-25　结点的存储结构

构造哈夫曼树算法实现

```
HUFFMAN(hufmtree tree[ ])
{       int  i, j, p1,p2;     char  ch;
        float small1, small2, f;
        for( i=0; i<m; i++)          /* 初始化 */
        { tree[i].parent=-1; tree[i].lchild=-1; tree[i].rchild=-1;
          tree[i].weight=0.0; tree[i].data= '0';       }
        for( i=0; i<n; i++)          /* 输入前n个结点的权值 */
        { scanf(" %f ", &f);   tree[i].weight=f;  scanf(" %c ",&ch);  tree[i].data=ch;  }
         for( i=n; i<m ;i++ )  /* 进行n-1次合并，产生n-1个新结点 */
         { p1=p2=0;   small1=small2=Maxval;   /* Maxval是float类型的最大值 */
            for ( j=0; j<i; j++ )
              { if ( tree[j].parent= =-1)
                  {if ( tree[j].weight<small1 )
                      { small2=small1;     /* 改变最小权，次最小权及对应位置 */
                        small1=tree[j].weight;   p2=p1;    p1=j;
                       } else  if( tree[j].weight<small2 )     /* 改变次小权及位置 */
                        {small2=tree[j].weight;   p2=j;   }
                  }
              }
            tree[p1].parent=i;  /* 给合并的两个结点的双亲域赋值 */
            tree[p2].parent=i;          tree[i].lchild=p1;
            tree[i].rchild=p2;  tree[i].weight = tree[p1].weight+tree[p2].weight;
         }
}    /* HUFFMAN */
```

构造哈夫曼树算法tree[]的变化过程

- 下图中的叶子结点集合构造哈夫曼树的初始状态如左图(a)所示，第一次合并状态如左图(b)所示，结果状态如左图(c)所示。



数组下标	lchild	data	weight	rchild	parent
0	-1	A	0.4	-1	-1
1	-1	B	0.3	-1	-1
2	-1	C	0.1	-1	-1
3	-1	D	0.1	-1	-1
4	-1	E	0.02	-1	-1
5	-1	F	0.08	-1	-1
6	-1	‘0’	0	-1	-1
7	-1	‘0’	0	-1	-1
8	-1	‘0’	0	-1	-1
9	-1	‘0’	0	-1	-1
10	-1	‘0’	0	-1	-1

(a)

数组下标	lchild	data	weight	rchild	parent
0	-1	A	0.4	-1	-1
1	-1	B	0.3	-1	-1
2	-1	C	0.1	-1	-1
3	-1	D	0.1	-1	-1
4	-1	E	0.02	-1	6
5	-1	F	0.08	-1	6
6	4	‘0’	0.1	5	-1
7	-1	‘0’	0	-1	-1
8	-1	‘0’	0	-1	-1
9	-1	‘0’	0	-1	-1
10	-1	‘0’	0	-1	-1

(b)

数组下标	lchild	data	weight	rchild	parent
0	-1	A	0.4	-1	10
1	-1	B	0.3	-1	9
2	-1	C	0.1	-1	7
3	-1	D	0.1	-1	7
4	-1	E	0.02	-1	6
5	-1	F	0.08	-1	6
6	4	‘0’	0.1	5	8
7	2	‘0’	0.2	3	8
8	6	‘0’	0.3	7	9
9	1	‘0’	0.6	8	10
10	0	‘0’	1	9	-1

(c)

图 5-226　哈夫曼树的初始，第一次合并和结果状态

哈夫曼编码的定义及数据结构

- 从哈夫曼树根结点开始，对左子树分配代码0，右子树分配代码1，一直到达叶子结点为止，然后将从树根沿每条路径到达叶子结点的代码排列起来，便得到了哈夫曼编码。
- n个叶子结点的最大编码长度不会超过n-1
- 编码数组结构描述如下：

```
typedef        char      datatype;
typedef        struct
{   char   bits[n];   /*  编码数组位串，其中n为叶子结点数目*/
    int    start;     /*  编码在位串的起始位置 */
    datatype    data;  /*  结点值 */
} codetype;
codetype code[n];
```

哈夫曼编码的算法的基本思想

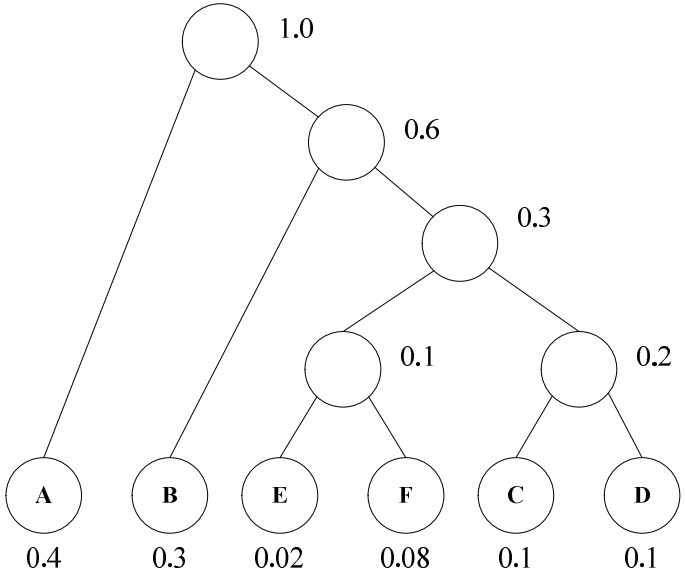
- 从叶子tree[i]出发，利用双亲地址找到双亲结点tree[p]，再利用tree[p]的lchild和rchild指针域判断tree[i]是tree[p]的左孩子还是右孩子，然后决定分配代码的 '0' 还是代码 '1' , 然后以tree[p]为出发点继续向上回潮，直到根结点为止。

哈夫曼编码算法实现

```
HUFFMANCODE(codetype code[ ]，hufmtree tree[ ] )
/* code 存放求出的哈夫曼编码的数组, tree已知的哈夫曼树 */
{ int i, c, p;
  codetype cd;                  /* 缓冲变量 */
  for ( i=0; i<n; i++ )             /* n为叶子结点数目 */
       {cd. start=n;  c=i;              /* 从叶子结点出发向上回溯 */
         p=tree[c].parent;
        cd.data=tree[c].data;
        while( p!=-1 )
            {      cd.start -- ;
                   if( tree[p]. lchild = = c)  cd.bits[cd.start]= '0';
                   else cd.bits [cd.start]='1';
                   c=p;
                   p=tree[c].parent;
            }
         code[i]=cd;   /* 一个字符的编码存入code[i] */
       }
}    /* HUFFMANCODE */
```

哈夫曼编码结果

- 对下图所示的哈夫曼树进行编码，可得到下表所示的编码表。



下标	bits						start	data
0						0	5	A
1					1	0	4	B
2			1	1	1	0	2	C
3			1	1	1	1	2	D
4			1	1	0	0	2	E
5			1	1	0	1	2	F

哈夫曼树译码

- 定义：哈夫曼树译码是指由给定的代码求出代码所表示的结点值。

哈夫曼树译码

- 定义：哈夫曼树译码是指由给定的代码求出代码所表示的结点值。
- 译码的过程是：从根结点出发，逐个读入电文中的二进制代码；若代码为0则走向左孩子，否则走向右孩子；一旦到达叶子结点，便可译出代码所对应的字符。然后又重新从根结点开始继续译码，直到二进制电文结束。

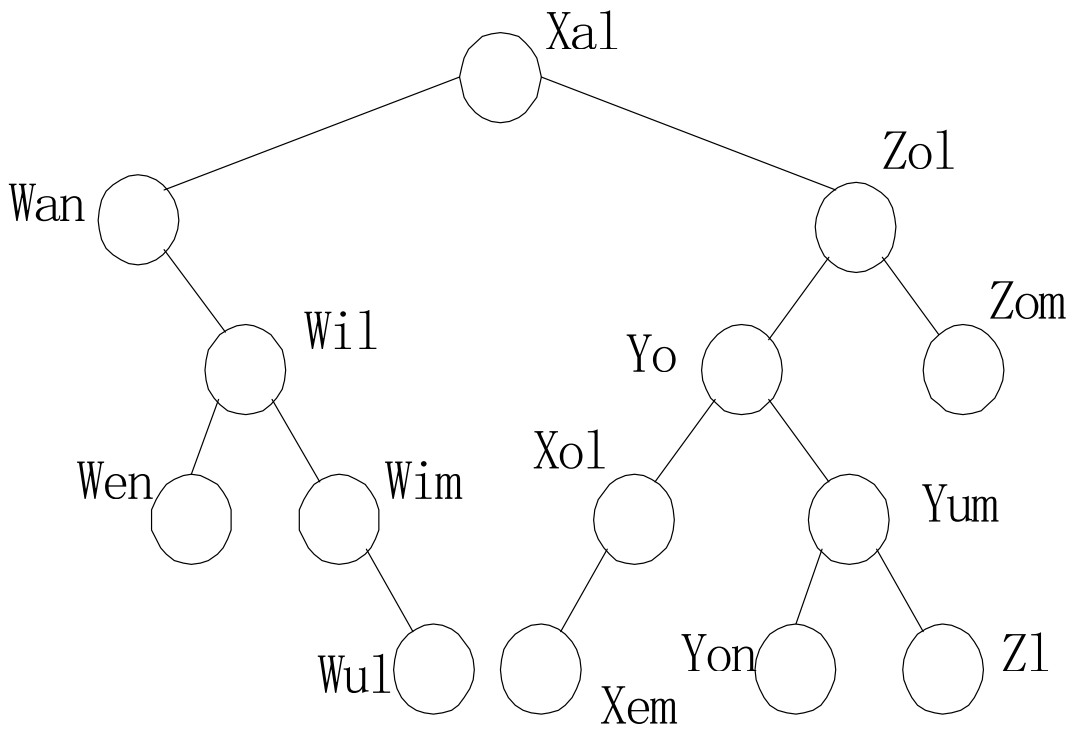
哈夫曼树译码算法

```
HUFFMANDECODE(hufmtree tree[ ])
{int   i, b;
 int endflag=-1;            /*  电文结束标志取-1 */
  i=m-1;                  /*  从根结点开始向下搜索 */
  scanf ( “%d”, &b);                  /* 读入一个二进制代码 */
  while ( b != endflag)    {
        if( b= =0)   i=tree[i].lchild;  /*  走向左孩子 */
        else   i=tree[i].rchild;       /*  走向右孩子 */
        if ( tree[i].lchild= =-1 )       /*  tree[i]是叶子结点 */
            {putchar( tree[i].data);
             i=m-1;              /*    回到根结点 */
            }
       scanf(“%d”, &b);        /*  读入下一个二进制代码 */
  }
  if ((tree[i].lchild!=-1)&&(i!=m-1) ) /*电文读完尚未到叶子结点 */
  printf( “\n ERROR\n”);             /*  输入电文有错 */
} /*  HUFFMANDECODE  */
```

上机作业

- 编写一个程序实现哈夫曼编码和译码。
- 要求：完成基本的编码/译码功能；

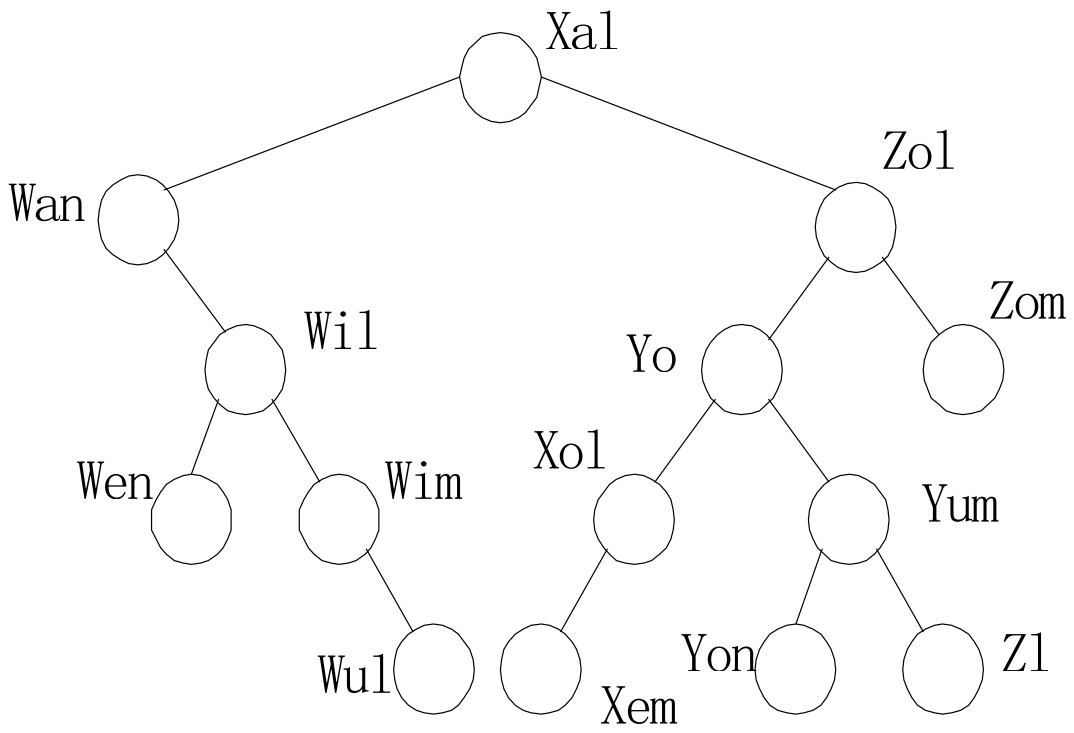
二叉排序树的基本概念



二叉排序树的示例

- 定义：如果一棵二叉树的每个结点对应一个关键码，并且每个结点的左子树中所有结点的码值都小于该结点的关键码值，而右子树中所有结点的关键码值都大于该结点的关键码值，则这个二叉树称为排序二叉树。

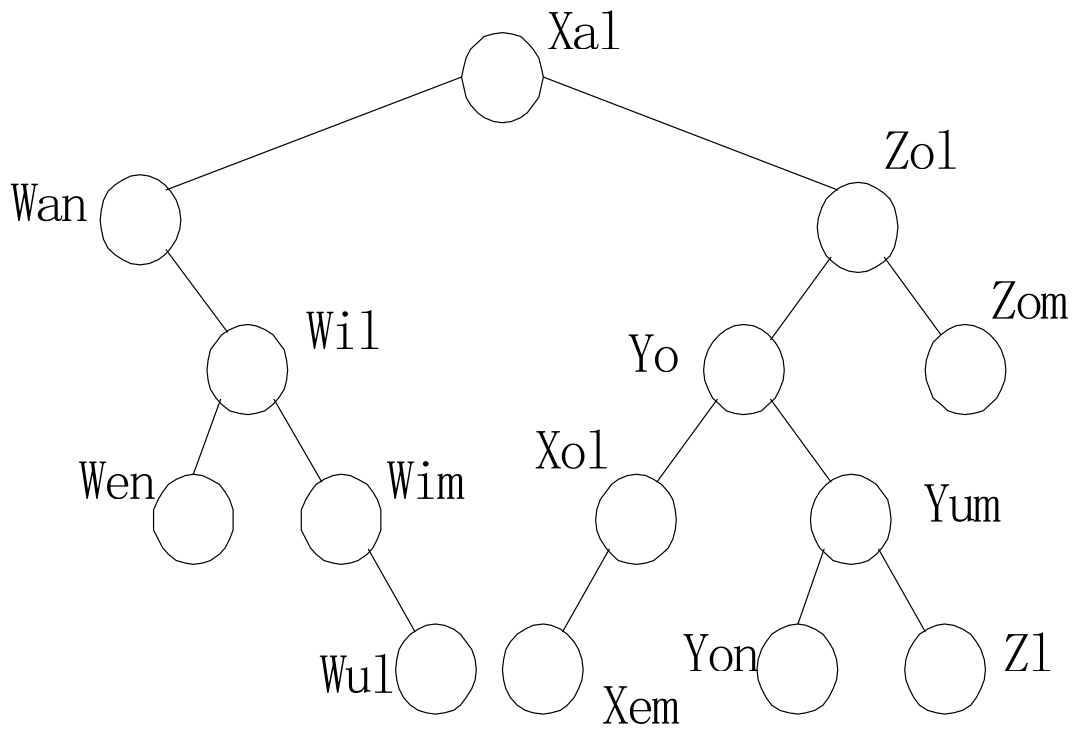
二叉排序树的基本概念



二叉排序树的示例

- 定义：如果一棵二叉树的每个结点对应一个关键码，并且每个结点的左子树中所有结点的码值都小于该结点的关键码值，而右子树中所有结点的关键码值都大于该结点的关键码值，则这个二叉树称为排序二叉树。
- 性质：对二叉排序树进行中序遍历时，可以发现所得到的中序序列是一个递增有序序列。

二叉排序树的基本概念



二叉排序树的示例

- 定义：如果一棵二叉树的每个结点对应一个关键码，并且每个结点的左子树中所有结点的码值都小于该结点的关键码值，而右子树中所有结点的关键码值都大于该结点的关键码值，则这个二叉树称为排序二叉树。
- 性质：对二叉排序树进行中序遍历时，可以发现所得到的中序序列是一个递增有序序列。
- 示例：对图进行中序遍历，可得到序列**Wan, Wen, Wil, Wim, Wul, Xal, Xem, Xol, Yo, Yon, Yum, Zl, Zol, Zom**。

二叉排序树的数据结构

- 二叉排序树的二叉链表存储结构结点结构描述如下：

```
typedef int keytype;
typedef struct node
{
    keytype key;      /*  关键字项 */
    datatype other;   /*  其它数据数据项 */
    struct  node *lchild, *rchild;     /*  左、右指针 */
} bstnode;
```

二叉排序树的构造

- 二叉排序树的构造是指将一个给定的数据元素序列构造为相应的二叉排序树。

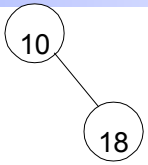
二叉排序树的构造

- 二叉排序树的构造是指将一个给定的数据元素序列构造为相应的二叉排序树。
- 对于任给的一组数据元素{ R_1, R_2, …, Rn }，可按以下方法来构造二叉排序树：
  - (1) 令R_1为二叉树的根;
  - (2) 若R_2<R_1, 令R_2为R_1左子树的根结点，否则R_2为R_1右子树的根结点;
  - (3) 对R_3, …, Rn结点也是依次与前面生成的结点比较以确定输入结点的位置。

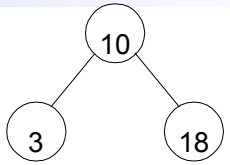
建立二叉排序树的实例

(a) 空树

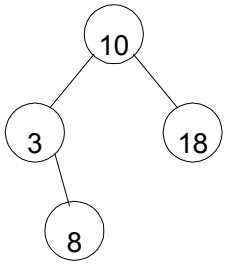
(b) 插入 10



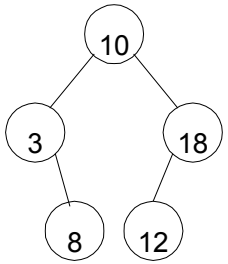
(c) 插入 18



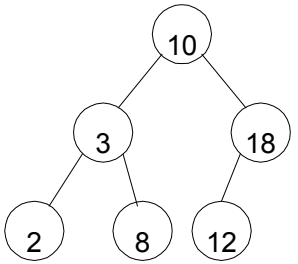
(d) 插入 3



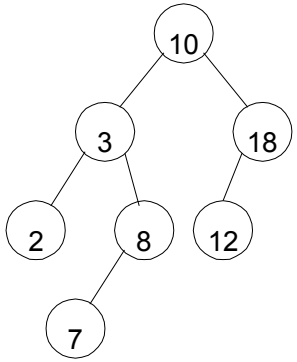
(e) 插入 8



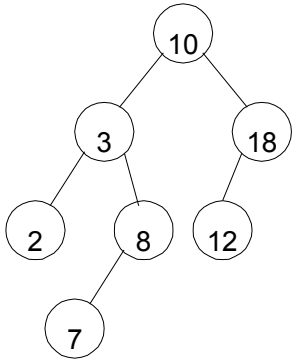
(f) 插入 12



(g) 插入 2



(h) 插入 7



(i) 插入 3

排序二叉树生成过程

二叉排序树的算法实现

- 插入一个结点，可用以下的非递归插入算法来实现：

```
bstnode *INSERTBST (bstnode *t, bstnode *s)
/*    t为二叉排序树的根指针，s为输入的结点指针*/
{ bstnode * f,  *p;
 if (t = = NULL)  return s;/*原树为空, 返回s作为根指针 */
 p=t;
   while ( p!=NULL )
   { f=p;                    /*  f指向 *p结点双亲 */
      if (s→key= =p→key) return t;
                             /*  树中已有结点*s；无须插入 */
      if (s→key< p→key ) p=p→lchild;
                             /*  在左子树中查找插入位置 */
      else p=p→rchild;      /*  在右子树中查找插入位置 */
    }
    if (s→key<f→key)  f→lchild=s;/*将*s插入为*f的左孩子 */
    else f→rchild =s;                  /*  将*s插入为*f的右孩子 */
    return t;
}  /*  INSERTBST */
```

二叉排序树的算法实现

- 生成二叉排序树的算法如下：

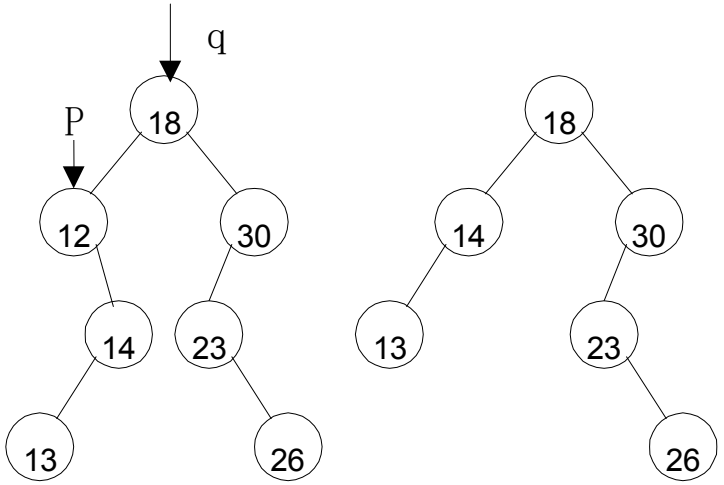
```
bstnode *CREATBST( )        /*  生成二叉排序树 */
{ bstnode  *t, *s;
   keytype  key , endflag=0;    /*  endflag为结点结束标志 */
   datatype  data;
   t=NULL;                  /*  设置二叉排序树的初态为空树 */
  scanf (“ %d ”,&key);        /*  读入一个结点的关键字 */
  while( key != endflag)
                    /*输入未到结束标志时，执行以下操作 */
     {  s= ( bstnode*) malloc( sizeof ( bstnode ) );   /*  申请新结点 */
        s→lchild = s→rchild = NULL;    /*  赋初值 */
        s→key = key;
        scanf ( “ %d ”, &data );         /*  读入结点的其它数据项 */
        s→other = data;
        t=INSERTBST( t, s )           /*  将新结点插入树t中 */
        scanf ( “ %d ” , &key )       /*  读入下一个结点的关键字 */
  }
  return  t;
}   /*  CREATBST */
```

二叉排序树中结点删除的基本要求

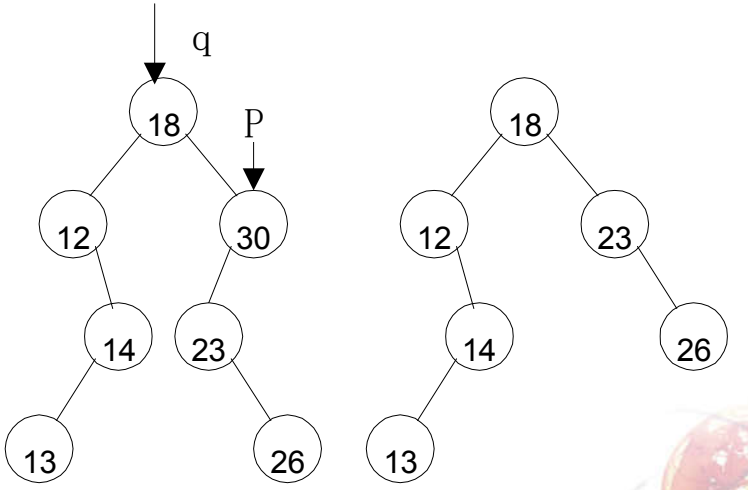
- 二叉排序树中删除某个结点之后，要求保留下来的结点仍然保持二叉排序树的特点，即每个结点的左子树中所有结点的关键码值都小于该结点的关键码值，而右子树中的所有结点的关键码值都大于该结点的关键码值。

二叉排序树中结点删除的基本思想

- 删除的结点由p指出，其双亲结点由q指出，则二叉排序树中结点删除分三种情况考虑：
  - (1) 若p指向叶子结点，则直接将该结点删除。
  - (2) 若p所指结点只有左子树p_L或只有右子树p_R，此时只要使p_L或p_R成为q所指结点的左子树或右子树即可，如下图(a)和(b)所示。



(a) P 仅有右子树删除前后情况

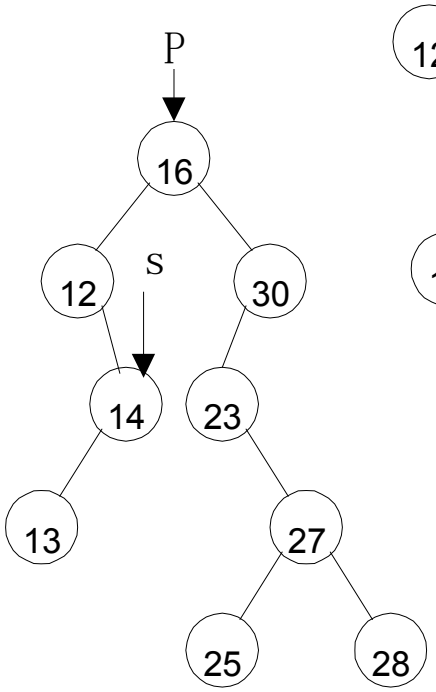


(b) P 仅有左子树删除前后情况

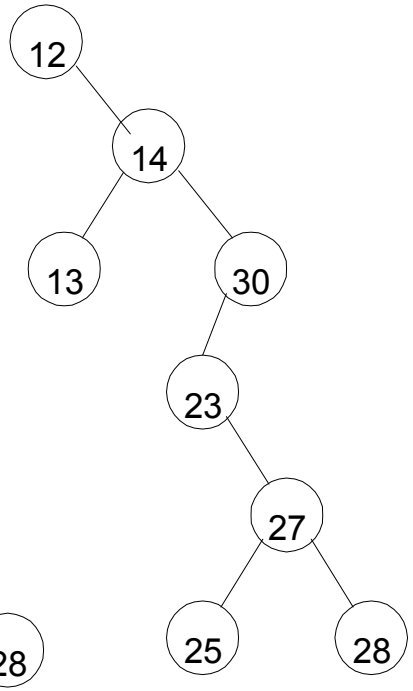
二叉排序树中结点删除的基本思想

- (3) 若p所指结点的左子树p_L和右子树p_R均非空，则需要将p_L和p_R链接到合适的位置上，即应使中序遍历该二叉树所得序列的相对位置不变。
  具体做法有两种：
  1：令p_L直接链接到q的左（或右）孩子链域上，而p_R则下接到p结点中序前趋结点s上（s是p_L最右下的结点）；
  2：以p结点的直接中序前趋或后继替代p所指结点，然后再从原二叉排序树中删去该直接前趋或后继。如下图(b)、(c)、(d)所示。

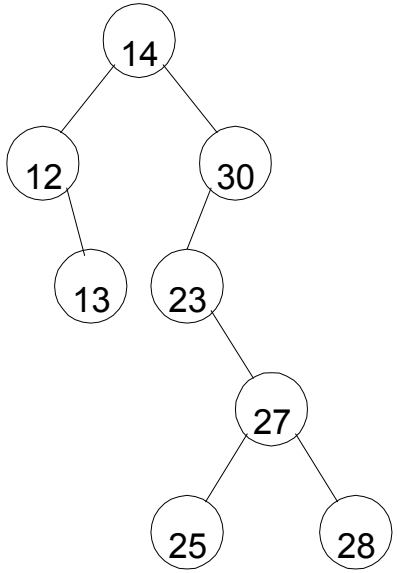
删除左右子树均非空的结点



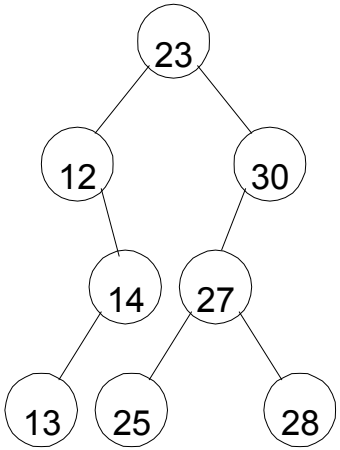
(a) 删除之二叉树



(b) P 右子树直接连接到 S



(c) P 直接前趋代替 P



(d) P 直接后趋代替 P

二叉排序树中结点删除算法

- **bstnode *DELBSTNODE(bstnode *t，keytype k)**

```
/* 在二叉排序树中删去关键字为k的结点 */
{  bstnode *p,*q,*s,*f;
   p=t;  q=NULL;
   while ( p!=NULL )                /*  查找关键字为k的待删结点*p */
   {  if ( p→key= =k)  break;       /*  找到，跳出循环 */
      q=p;                          /*  q指向*p的双亲 */
      if ( p→key>k )  p=p→lchild;  else  p=p→rchild;  }
   if ( p= =NULL ) return  t;       /*  找不到返回原二叉树 */
   if ( p→lchild= =NULL)            /*  p所指结点的左子树为空 */
    {  if (q= =NULL)  t=p→rchild;/*  p所指结点是原二叉排序树的根 */
       else if (q→lchild= =p)       /*  p所指结点是*q的左孩子 */
            q→lchild=p→rchild; /*将p所指右子树链接到*q的左指针域上*/
       else  q→rchild=p→rchild;/*将p所指右子树链接到*q的右指针域上*/
       free(p); /*  释放被删结点 */    }
   else   /*  p所指结点有左子树时，则按图5-31(c)方法进行 */
        { f=p;  s=p→lchild;
          while(s→rchild != NULL )    /*  在pL中查找最右下结点 */
            { f=s;  s=s→rchild; }
          if ( f= =p )  f→lchild=s→lchild;/*将s结点的左子树链接到*f上*/
          else f→rchild=s→lchild;
          p→key=s→key;        /*  将s所指结点的值赋给*p */
          p→other=s→other;     free (s);   /*  释放被删结点 */      }
   return  t;
}   /*  DELBSTNODE */
```

本章小结和习题

- 习 题
  - 14，15，17，21
- 上 机
  - 17，21

第4章习题

26. 稀疏矩阵用三元组表示，求C=A*B

```
#define smax 32   /* 最大非零元素个数的常数 */
#define M 4
#define N 5
#define P 6
typedef int datatype;
typedef struct
{  int      i,j;      /* 行，列号 */
   datatype v;        /* 元素值 */
} node;
typedef struct
{    int m,n,t;       /*行数,列数,非零元素个数*/
     node data[smax];  /* 三元组表 */
} spmatrix;           /* 稀疏矩阵类型 */
static matrix_aa[M][N]={{0,0,2,5,0},{1,0,0,0,6},{0,0,0,0,3},{9,0,0,5,0}};
static matrix_bb[N][P]={{1,0,0,0,0,5},{0,0,4,0,0,8},{0,0,0,0,0,0},{2,0,0,0,0,5},{0,0,0,1,0,0}};
int InitMatrix(spmatrix *AA, spmatrix *BB);
int Muiti(spmatrix *AA, spmatrix *BB, spmatrix *CC);
void PrintMatrix(spmatrix *matrix);
```

```
main()
{   spmatrix *MA, *MB, *MC;
    MA = (spmatrix*)malloc (sizeof(spmatrix));
    MB = (spmatrix*)malloc (sizeof(spmatrix));
    MC = (spmatrix*)malloc (sizeof(spmatrix));
    if(MA==NULL || MB==NULL || MC==NULL)
    {    printf("memory alloc error!\n");}
    InitMatrix(MA, MB);
    PrintMatrix(MA);  PrintMatrix(MB);
    Muiti(MA, MB, MC);
    PrintMatrix(MC);
    free(MA);  free(MB);   free(MC);
}
```

```
int InitMatrix(spmatrix *AA, spmatrix *BB)
{    int i,j,k;
     k = 0;
     for(i=0; i<M; i++)
     {  for(j=0;j<N;j++)
            {if(matrix_aa[i][j] !=0 )
              { AA->data[k].i = i;  AA->data[k].j = j;   AA->data[k].v = matrix_aa[i][j];
                    k++;}
            }
     }
     AA->t = k;    AA->m = M;   AA->n = N;
     k = 0;
     for(i=0; i<N; i++)
     {  for(j=0;j<P;j++)
            { if(matrix_bb[i][j] !=0 )
              {  BB->data[k].i = i;  BB->data[k].j = j;  BB->data[k].v = matrix_bb[i][j];
                    k++;}
            }
     }
     BB->t = k;   BB->m = N;  BB->n = P;
     return 0;
}
```

```
int Muiti(spmatrix *AA, spmatrix *BB, spmatrix *CC)
{   int i,j,ano,bno,cno;
    CC->m = AA->m;  CC->n = BB->n;
    for(i=0; i<smax; i++)    CC->data[i].v=0;
    cno = 0;
    for(i=0; i<AA->m; i++)
       for(j=0; j<BB->n; j++)
         {  for(ano=0; ano<AA->t; ano++)
           {  for(bno=0; bno<BB->t; bno++)
              { if( (AA->data[ano].i==i) && (BB->data[bno].j==j) &&
                                          (AA->data[ano].j == BB->data[bno].i))
                 {CC->data[cno].v += (AA->data[ano].v) * (BB->data[bno].v);}
               }
            }
           CC->data[cno].i = i;   CC->data[cno].j = j;  cno++;
         }
    CC->t = cno;
    return 0;
}
```

```
void PrintMatrix(spmatrix *matrix)
{   int i,j,k,flag;
    int value0=0;
    printf("the matrix(%d x %d) is :\n", matrix->m, matrix->n);
    for(i=0; i<matrix->m; i++)
     {  for(j=0; j<matrix->n; j++)
          {  flag = 0;
             for(k=0; k<matrix->t;k++)
                { if( (matrix->data[k].i==i) && (matrix->data[k].j==j) )
                    {  printf("%d ", matrix->data[k].v);
                          flag = 1;  break;
                     }
                 }
              if(flag == 0)   printf("%d ", value0);
           }
        printf("\n");
     }
}
```