第 3 章　树

在第二章中，主要介绍了线性结构，从本章开始，将依次介绍非线性结构。与线性结构不同的是，非线性结构中的每个元素可以有多个前驱或者后继。

本章将重点介绍树形结构。客观世界中存在很多这样的非线性结构，例如家谱，书籍的章节结构等。树形结构被广泛地应用在各种算法中，该结构对于处理嵌套数据具有极好的效果。本章将重点介绍树和二叉树的一些基本概念及操作，同时给出一些实用的例子来加深对本章内容的理解。

3.1　树的基本概念

3.1.1　树的定义和基本术语

树是由 n 个结点组成的有限集合。在这个集合中没有直接前驱的结点叫做根结点。

1）当 n=0 时，它是一个空结构，或者空树。

2）当 n>0 时，所有结点中存在且仅存在一个根结点，其余结点可以分为 m（m≥0）个互不相交的子集合，每一个子集合都是一棵满足定义的树，并称为根的子树。

由此可知，树的定义是一个递归的定义，即树的定义中又用到了树的概念。

如图 3-1 所示，以公司组织机构为例给出了一个树状结构的例子。下面介绍一下有关树的基本术语。

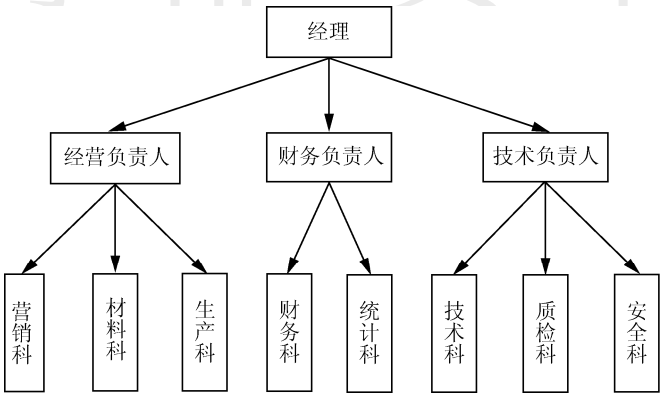


图 3-1　表示为树形结构的公司组织机构图

结点：树形结构中的每个元素称为一个结点，结点包含该元素的值和该元素的逻辑关系信息。例如在图 3-1 中，该组织机构树总共有 12 个结点。

边：用＜m,n＞表示从结点 m 指向结点 n 的连线，代表结点 m 与结点 n 之间存在某

种联系。这种有向连线就称为边。

双亲结点和孩子结点：如果在树形结构中存在从 m 结点指向 n 结点的连线＜m,n＞，则称 m 结点是 n 结点的双亲结点（也称父结点），同时，n 结点称为 m 结点的孩子结点（也称子结点）。如图 3-1，“财务负责人”结点有指向“财务科”结点的连线，则“财务负责人”结点是“财务科”结点的双亲结点，“财务科”结点是“财务负责人”结点的孩子结点。

兄弟结点：如果两个结点有共同的双亲结点，则这两个结点互为兄弟结点。例如图 3-1 中，“财务科”结点和“统计科”结点的双亲结点都是“财务负责人”，因此，“财务科”结点和“统计科”结点互为兄弟结点。

叶子结点：没有孩子结点的结点称作叶子结点或者终端结点。在图 3-1 中，最底层上的结点都是叶子结点。

分支结点：非叶子结点（终端结点）称为分支结点。

结点的度：该结点所拥有的孩子结点的数量就称作该结点的度。例如图 3-1，“经理”结点的度是 3，“财务负责人”的度则是 2。根据叶子结点的定义，所有的叶子结点的度都为 0。

树的度：该树形结构中的所有结点的度的最大值，就是该树的度。因此，图 3-1 中树的度为 3。

结点的层数：结点的层数从根结点起开始定义，根结点的层数为 0。树中其他任一结点的层数为其双亲结点的层数加 1。如图 3-1 中，经理的层数是 0，财务科的层数是 2。

树的深度：树中所有结点的层数的最大值就是该树的深度。空树的深度为 0，图 3-1 中树的深度为 2。

树的高度：树的高度等于树的深度加 1。图 3-1 中树的高度为 3。

路径：从树的一个结点 m 到另一个结点 n，如果存在一个有限集合 $S=\{s_1,s_2,\cdots,s_k\}$，使得 $<m,s_1>,<s_1,s_2>,\cdots,<s_k,n>$ 都是该树中的边，则称从结点 m 到结点 n 存在一条路径。

祖先结点和子孙结点：如果从结点 m 到结点 n 存在一条路径，则称结点 m 是结点 n 的祖先结点，结点 n 是结点 m 的子孙结点。

有序树：如果树中每个结点的所有子树之间存在确定的次序关系，则称该树为有序树。

无序树：树中的每个结点的各子树之间不存在确定的次序关系，则称该树为无序树。

森林：森林是由 m（m≥0）棵互不相交的树组成的集合。

3.1.2 树的基本性质

树具有如下基本性质：

【性质 1】树中的结点数等于其所有结点的度数加 1。

证明：根据树的定义，每个结点的子树之间互不相交，也就是说除了根结点以外的

其他所有结点，每个结点有且仅有一个前驱结点。这说明树中除了根结点以外的其他每个结点都只对应一个分支（边），而每个结点的分支数就是该结点的度数，因而除根结点以外的所有结点的数量等于所有结点的度数之和，因此，加上根结点后可得到结论：树的结点数等于其所有结点的度数加 1。

【性质 2】度为 m 的树，其第 i 层上至多有 m^i 个结点（根结点为第 0 层，i≥0）。

证明：该性质可由数学归纳法来证明。第 0 层上只有根结点，因此将 i=0 代入 m^i 中，结果为 1，也就是第 0 层上至多有 1 个结点，因此当 i=0 时结论成立。假设 i=k 时结论成立，也就是第 k 层上最多有 m^k 个结点.由于树的度是 m，也就是每个结点的度都不大于 m。因此，第 k 层上每个结点的度为 m 时，第 k+1 层上的结点数是最多的。而第 k 层上的结点数至多为 m^k，因此，第 k+1 层上的结点数至多为 $m\times m^k$ 也就是 m^{k+1} 个，这与性质 2 的结论相同，因此性质 2 成立。

【性质 3】高度为 h（深度为 h-1）度为 m 的树至多有 $\frac{m^h-1}{m-1}$ 个结点（m>1）。

证明：在性质 2 中给出度为 m 的树，其第 i 层上至多有 m^i 个结点（i≥0）。因此，高为 h 度为 m 的树，当其每一层上的结点数都达到该层最大的结点数的时候，该树才具有最多的结点数，因此整个树的最大结点数就是该树的每一层的最大结点数之和，也就是 $m^0+m^1+\cdots+m^{h-1}$，计算等比数列得到结果 $\frac{m^h-1}{m-1}$，因此性质 3 成立。

【性质 4】具有 n 个结点的度为 m 的树，其最小高度为 $\lceil \log_m(n(m-1)+1)\rceil$（$\lceil x\rceil$ 代表大于等于 x 的最小整数）。

证明：假设该树的高度为 h，当该树的前 h-1 层的结点数全都达到对应层的最大结点数时，该树具有最小高度，此时，第 h 层的结点数可能等于该层的最大结点数，也可能小于最大结点数。根据树的性质 3 可得结点总数 n 和高度 h 的关系：

$$\frac{m^{h-1}-1}{m-1}<n\leqslant\frac{m^h-1}{m-1}$$

将该式左边部分的不等式化简，得到 $h<\log_m(n(m-1)+1)+1$。

同理化简右部分不等式得到 $h\geqslant\log_m(n(m-1)+1)$。

综合两个不等式，即 $\log_m(n(m-1)+1)\leqslant h<\log_m(n(m-1)+1)+1$

由于高度 h 只能取整数，因此得到 $h=\lceil\log_m(n(m-1)+1)\rceil$。

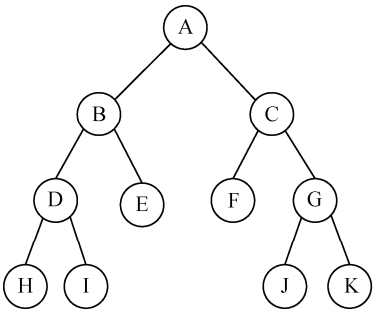
3.1.3　树的逻辑表示方式

树的逻辑表示方法有很多种，这里介绍几种最直观的表示结点之间层次关系的方法。

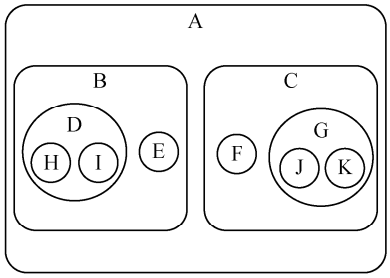
1．树形表示法

将关系集合抽象成一棵倒置的树，是最常用且最直观的方法，它是用一个圆圈表示一个结点，在圆圈内填写该结点的信息，例如名字。结点之间的关系通过连线表示，该

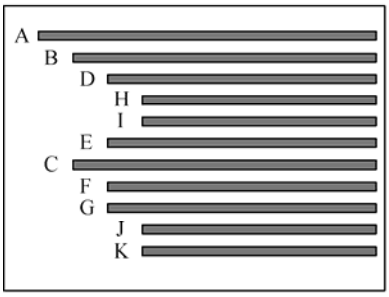
方法是以层次的关系来表示结点间的关系。家族的族谱可以用这种方式来描述，如图 3-2（a）所示，A 有两个孩子 B 和 C，B 有两个孩子 D 和 E，依这样的方式描述下去。



（a）树形表示法



（b）文氏图表示法



（c）凹入表表示法

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

（d）嵌套括号表示法

图 3-2　树的逻辑表示方式

2．文氏图表示法

该方法是用一个圆圈代表一棵子树，圆圈中包含根结点和该根结点的子树（由圆圈表示），而且子树之间的圆圈不相交。公司的组织机构可以由这样的表示法来描述，如图 3-2（b）所示，A 公司含有机构 B 和 C，机构 B 含有子机构 D 和 E，以此类推下去，将整个公司的组织机构表示出来。

3．凹入表表示法

每个结点由一个条形表示，而且子树的根结点的条形比其父结点的条形短，且紧邻其父结点。同一父结点下的各子树的根结点的条形长度一致。书的章节的目录与这种表示方法近似，如图 3-2（c）。

4．嵌套括号表示法

先将根结点放入一对圆括号中，然后把它的子树按照由左而右的顺序放入括号中，而对子树也采用同样方法处理：同层子树与它的根结点用圆括号给括起来，同层子树之

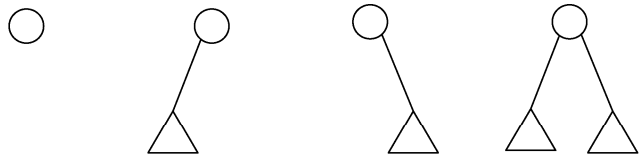
间用逗号隔开，如图 3-2（d）。

3.2　二　叉　树

3.2.1　二叉树的定义和相关概念

二叉树是树形结构中的一种特殊形式。其定义是：所有结点的度小于等于 2 的树称为二叉树。

从二叉树的定义可以看出，二叉树中不存在度大于 2 的结点，也就是每个结点至多可以有两棵子树。因此，二叉树中每个结点都有两棵子树—左子树和右子树，其中左子树或者右子树可以为空，或者左右子树均为空。因此二叉树可以有五种基本的形态，在图 3-3 中给出示意图。由于二叉树的子树有左右之分，次序不可颠倒，因此二叉树是有序树。



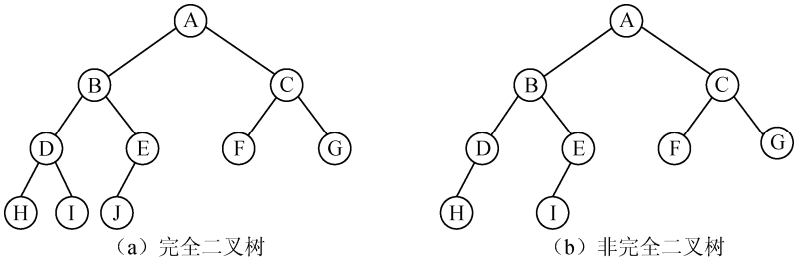
（a）空二叉树　（b）只有根结点的二叉树　（c）右子树为空的二叉树　（d）左子树为空的二叉树　（e）左右子树都非空的二叉树

图 3-3　二叉树的五种基本形态

3.2.2　几种特殊的二叉树

完全二叉树：一棵高度为 h 的二叉树，除最后一层以外的其他所有层上的结点数都达到最大值，而最后一层上的所有结点分布在该层最左边的连续的位置上。

完全二叉树有如下的特点，叶子结点只能在层次最大和次大的两个层次上出现。对任一结点，如果其左子树的高度为 m，则其右子树的高度必为 m 或者 m-1。图 3-4 中给出完全二叉树和非完全二叉树的例子。



（a）完全二叉树　（b）非完全二叉树

图 3-4　完全二叉树与非完全二叉树

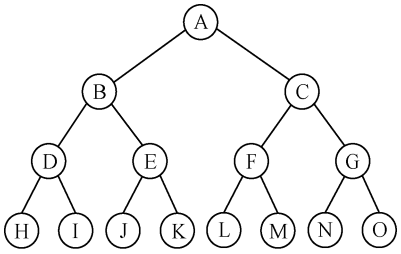


图 3-5　满二叉树

满二叉树：一棵二叉树，如果其所有分支结点都有非空左子树和非空右子树，并且所有叶子结点都在同一层上，这样的二叉树称为满二叉树。高度为 h 的满二叉树有 2^h-1 个结点。满二叉树一定是完全二叉树，但完全二叉树不一定是满二叉树。图 3-5 中给出满二叉树的例子。

满二叉树有如下的特点，每一层上的结点数量都达到最大个数，所有分支结点的度都为 2，叶子结点都出现在最后一层上。

扩充二叉树：把原二叉树所有结点中出现空的子树的位置上都增加特殊的结点——空树叶，得到的二叉树就称为扩充二叉树。也就是对于原来度为 2 的分支结点，不增加空树叶；对于度为 1 的分支结点，增加一个空树叶；对于度为 0 的叶子结点，增加两个空树叶。这里的空树叶又称为外部结点，二叉树中原有的结点称为内部结点。

普通的二叉树经过上述的扩充步骤之后新增加的空树叶（外部结点）的数量等于原来二叉树中结点（内部结点）的数量加 1。

外部路径长度 E：扩充的二叉树里从根结点到每个外部结点的路径长度之和。

内部路径长度 I：扩充的二叉树里从根结点到每个内部结点的路径长度之和。

例如在图 3-6 中（a）图是原二叉树，（b）图中用方框来代表空树叶，即外部结点。在这个扩充二叉树中：

$$E=2+4+4+4+5+5+5+5+4+4+4+3+3=52$$

$$I=1+2+3+3+4+1+2+2+3+3+4=28$$

E 和 I 这两个量之间的关系为 $E=I+2n$，这里的 n 是内部结点的数量。该结论可以用归纳法证明出来，有兴趣的读者可以自己证明。

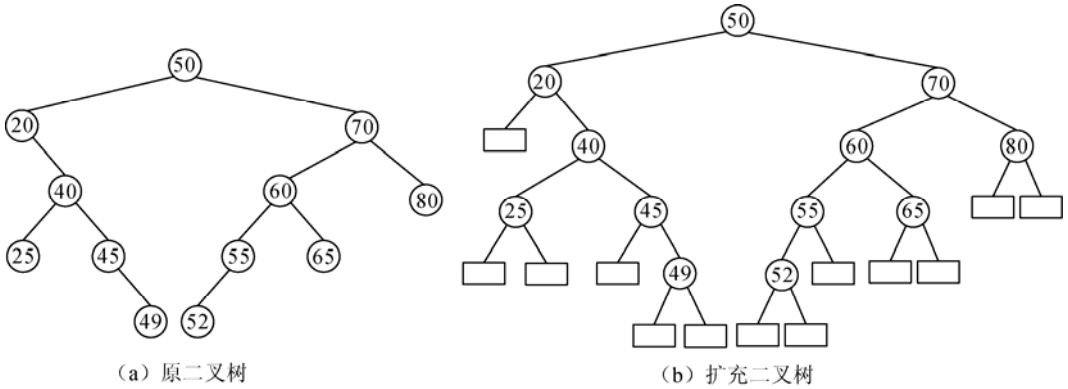


图 3-6　扩充二叉树

3.2.3　二叉树的性质

由于二叉树具有特殊的结构，因此具有一些特殊的性质。

性质 1　任何一棵二叉树，度数为 0 的结点比度数为 2 的结点多一个。

证明：设二叉树的结点总数为 n，其中度数为 0，1，2 的结点的数量分别为 x_0，x_1，x_2，则有

$$n = x_0 + x_1 + x_2 \tag{3.1}$$

根据树的性质 1：树中的结点数等于所有结点的总度数加 1。对于二叉树，就有：

$$n = 0 \cdot x_0 + 1 \cdot x_1 + 2 \cdot x_2 + 1 \tag{3.2}$$

由式（3.1）和（3.2）相减，得：

$$x_0 = x_2 + 1 \tag{3.3}$$

因此，性质 1 的结论成立。

由树的基本性质可以得到二叉树的如下性质：

性质 2　二叉树的第 i 层上至多有 2^i 个结点（$i \geqslant 0$）。

性质 3　高度为 h 的二叉树至多有 $2^h - 1$ 个结点。

满二叉树是二叉树中的一种特殊的结构，因此具有特殊的性质。

性质 4　非空满二叉树的叶子结点的数量等于其分支结点的数量加 1。

证明：假设该满二叉树的高度为 h（深度为 $h-1$），其第 $h-1$ 层上的所有结点都为叶子结点，其他层上的结点都是分支结点，且度都为 2。满二叉树的每层上的结点数都为该层的最大结点数，因此根据树的基本性质 2 得：

第 $h-1$ 层上的结点数为

$$2^{h-1} \tag{3.4}$$

其他所有层的结点数为

$$2^0 + 2^1 + \cdots + 2^{h-2} \tag{3.5}$$

式（3.5）化简结果为 $2^{h-1} - 1$，因此二叉树的性质 4 的结论成立。

性质 5　有 n 个结点的完全二叉树的高度为 $\lceil \log_2(n+1) \rceil$。

证明：完全二叉树是除了最后一层以外，其他所有层的结点数都等于该层最大结点数，因此该完全二叉树是具有 n 个结点且度为 2 的所有树中高度最小的树，满足树的基本性质 4，此时 m=2，代入到 $\lceil \log_m(n(m-1)+1) \rceil$ 中得到该树的高度为 $\lceil \log_2(n+1) \rceil$。

性质 6　如果对一棵有 n 个结点的完全二叉树的结点按层编号（从低层到高层，每层从左到右），则对任一结点 i（$1 \leqslant i \leqslant n$），有：

如果 i=1，则结点 i 无双亲，是二叉树的根。如果 i＞1，则其双亲是结点 i/2。

如果 2i＞n，则结点 i 是叶子结点，否则，其左孩子是结点 2i。

如果 2i+1＞n，则结点 i 无右孩子，否则，其右孩子是结点 2i+1。

此外，若对二叉树的根结点从 0 开始编号，则相应的 i 号结点的双亲结点的编号为 (i−1)/2，左孩子的编号为 2i+1，右孩子的编号为 2i+2。

此性质可采用数学归纳法证明。证明略。

这个性质是一般二叉树顺序存储的重要基础。

3.2.4 二叉树的存储结构

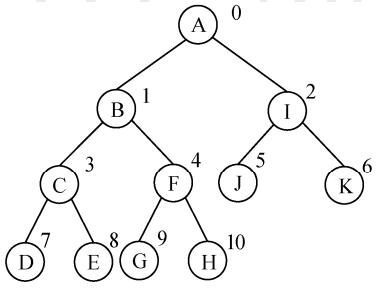
二叉树通常有两种存储结构：顺序存储和链式存储。下面将详细讨论这两种结构。

（1）二叉树的顺序存储结构

二叉树的顺序存储结构由一个一维数组构成，二叉树上的结点按照某种次序分别存入该数组的各个单元中。显然，这里的关键在于结点的存储次序，结点之间的逻辑关系则由这种次序反映出来。

如果将一棵完全二叉树按层对所有结点进行编号，则结点编号之间的数值关系可以准确地反映出结点之间的逻辑关系。因此，对于任意的完全二叉树来说，可以采用“以编号为地址”的策略将结点存入一维数组中。也就是将编号为 i 的结点存入一维数组的第 i 个单元中。如图 3-7，完全二叉树的顺序存储结构示意图。

在这一存储结构中，由于某一结点的存储位置（即下标）同时也是它的编号，因此结点间的逻辑关系可以通过它们下标的数值关系来确定。例如，要在图 3-7 中的顺序存储结构中查找结点 F 的左孩子，根据二叉树的性质 6，从 F 的下标为 4 可以算出，其左孩子的下标为 9，右孩子的下标为 10，对应的结点就是 G 和 H。如果要找结点 K 的左孩子，由于 K 的下标为 6，2*6=12＞11，所以结点 K 为叶子结点，没有左孩子。由此可见，这种存储结构可以方便的实现二叉树的各种运算。



（a）完全二叉树

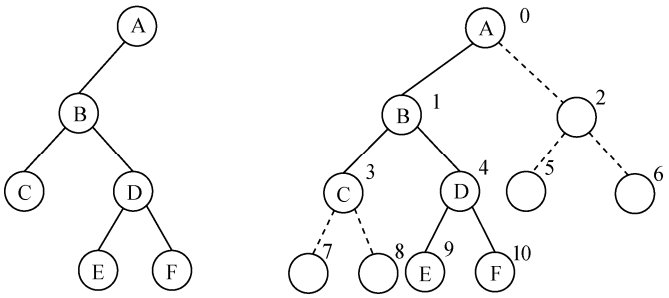
数组	A	B	I	C	F	J	K	D	E	G	H
下标	0	1	2	3	4	5	6	7	8	9	10

（b）顺序存储示意图

图 3-7　完全二叉树的顺序存储结构

但是，假如需要存储的二叉树不是完全二叉树，上面的存储策略就不能直接使用了。因此，必须先将一般二叉树转化为完全二叉树，这个步骤可以通过在非完全二叉树的空缺位置上增加“虚结点”来实现。例如图 3-8 所示。

显然，经过转化后再按层编号进行存储的方法可以解决非完全二叉树的顺序存储问题，但同时也造成了存储空间的浪费，所以二叉树的顺序存储结构一般只应用于一些特殊的情况下。



（a）二叉树　　（b）转化后的完全二叉树

数组	A	B		C	D					E	F
下标	0	1	2	3	4	5	6	7	8	9	10

（c）顺序存储示意图

图 3-8　非完全二叉树的顺序存储结构

（2）二叉树的链式存储结构

二叉树有多种链式存储结构，其中最常用的是二叉链表，二叉链表的结点形式如图 3-9 所示。

leftChild	data	rightChild

图 3-9　二叉链表的结点

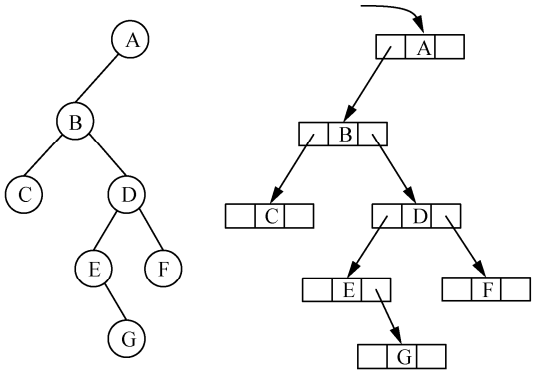
其中 data 域称为数据域，用于存储二叉树结点中的数据元素，leftChild 域称为左孩子域，用于存放指向本结点左孩子的指针（称为左指针），类似地，rightChild 域称为右孩子指针域，用于存放指向本结点右孩子的指针（称为右指针）。二叉链表中的所有存储结点通过它们的左右指针的链接而形成一个整体。此外，每个二叉链表还必须有一个指向根结点的指针，称为根指针。根指针用来标识二叉链表，访问者可以从根指针开始对二叉链表进行访问。

图 3-10（a）和（b）分别表示一棵二叉树及其对应的二叉链表。二叉链表中每个结点的每个指针域必须有一个值，这个值可以是该结点的一个孩子的指针，也可以是空指针。

若二叉树为空，则 root =NULL。若某个结点的某个孩子不存在，则相应的指针为空。具有 n 个结点的二叉树中，一共有 2n 个指针域，其中只有 n−1 个指针用来指向结点的左、右孩子，其余的 n+1 个指针均为空。

在二叉链表这种存储结构上，二叉树的很多基本运算，如求根，求左、右孩子等都很容易实现。但求双亲运算的实现却比较麻烦，而且时间效率不高。由于在给定的实际问题中经常需要求双亲运算，因此选用二叉链表为存储结构效率不高，这时可以采用三叉链表作为存储结构。

三叉链表是二叉树的另一种主要的链式存储结构。三叉链表与二叉链表的主要区别



（a）二叉树　　（b）二叉树的链式存储结构

图 3-10　二叉树及其链式存储结构

在于，它的结点比二叉链表的结点多一个指针域，该域用来存储一个指向其父结点的指针。三叉链表的结点形式如图 3-11 所示。

对于图 3-10 中的二叉树，其三叉链表的表示形式在图 3-12 中给出。

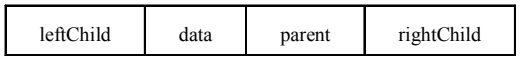


图 3-11　三叉链表的结点形式

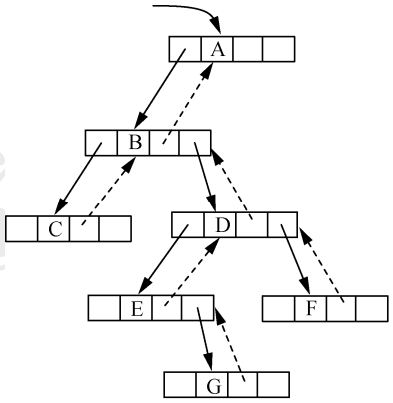


图 3-12　二叉树的三叉链表存储结构

显然，在三叉链表上，二叉树的求双亲运算很容易实现，而且时间性能很好。二叉树的链式存储结构操作方便，表达简明，因而成为二叉树最常用的存储结构。但是该结构也存在弊端，由于存储了大量的指针，空间浪费比较严重。因此，在某些情况下，二叉树的顺序存储结构也很有用处。

3.2.5　二叉树的抽象数据类型

在现实算法中，经常会用到树形结构来解决问题。树中有些运算是对整棵树而言的，有些运算是对结点而言的，因此在代码中必须将树与结点的类分别实现。

例 3.1 给出二叉树结点的抽象数据类型 BinaryTreeNode，例 3.2 给出二叉树的抽象

数据类型 BinaryTree。

【例 3.1】二叉树结点的抽象数据类型。

```
template <class T>
class BinaryTreeNode{
    friend class BinaryTree<T>;
private:
    T element;                          //结点的数据域
    BinaryTreeNode<T> * leftChild;      //结点的左孩子结点
    BinaryTreeNode<T> * rightChild;     //结点的右孩子结点
public:
    BinaryTreeNode();                   //默认构造函数
    BinaryTreeNode(const T& ele);       //给定数据域的值的构造函数
    BinaryTreeNode(const T& ele, BinaryTreeNode<T> * l,
    BinaryTreeNode<T> * r);             //给定数据值和左右孩子结点的构造函数
    BinaryTreeNode<T> * getLeftChild() const;//返回该结点的左孩子结点
    BinaryTreeNode<T> * getRightChild() const;//返回该结点的右孩子结点
    void setLeftChild(BinaryTreeNode<T> * l);//设置该结点的左孩子结点
    void setRightChild(BinaryTreeNode<T> * r);//设置该结点的右孩子结点
    T getValue() const;                         //返回该结点的数据值
    void setValue(const T& val);                //设置该结点的数据域的值
    bool isLeaf() const;  //判断该结点是否是叶子结点，若是，则返回 true
};
```

【例 3.2】二叉树的抽象数据类型。

```
template <class T>
class BinaryTree{
private:
    BinaryTreeNode<T> * root;               //二叉树根结点
public:
    BinaryTree();                           //默认构造函数
    ~BinaryTree();                          //析构函数
    bool isEmpty() const;                   //判断二叉树是否为空树
    BinaryTreeNode<T> * getRoot() const;    //返回二叉树的根结点
    BinaryTreeNode<T> * getParent(BinaryTreeNode<T> * current) const;
                                            //返回 current 结点的父结点
    BinaryTreeNode<T> * getLeftSibling(BinaryTreeNode<T> * current)
const;
                                            //返回 current 结点的左兄弟
    BinaryTreeNode<T> * getRightSibling(BinaryTreeNode<T> * current)
const;
                                            //返回 current 结点的右兄弟
    void breadthFirstOrder(BinaryTreeNode<T> * root);
                                        //广度优先遍历以 root 为根结点的子树
    void preOrder(BinaryTreeNode<T> * root);
                                        //先序遍历以 root 为根结点的子树
```

```
    void inOrder(BinaryTreeNode<T> * root);
                                        //中序遍历以 root 为根结点的子树
    void postOrder(BinaryTreeNode<T> * root);
                                        //后序遍历以 root 为根结点的子树
    void levelOrder(BinaryTreeNode<T> * root);
                                        //按层次遍历以 root 为根结点的子树
    void deleteBinaryTree(BinaryTreeNode<T> * root);
                                        //删除以 root 为根结点的子树
};
```

3.2.6 二叉树的遍历

遍历二叉树也就是按照某种次序，顺着制定的搜索路径访问二叉树中的各个结点，该过程中每个结点被且仅被访问一次。访问操作包括很多种情况，例如读取结点的值，修改结点的数据值等等。虽然该过程可以修改结点的数据值，但是不允许修改结点之间的逻辑关系。

遍历是任意数据结构都具有的操作，但是二叉树的非线性结构的特性使得其可以有多种遍历方法，于是就存在按何种搜索路径进行遍历的问题。由于上述原因，在遍历的时候就必须规定遍历的规则。

根据二叉树的结构特征，可以有两种搜索路径：广度优先遍历和深度优先遍历。

1．广度优先遍历

广度优先遍历也就是按层次遍历，是从最高层（或者最低层）开始，向下（向上）逐层访问每个结点，在每一层上，自左向右（或者自右向左）访问每个结点。

该遍历方法可以使用队列来实现。广度优先遍历为自上向下，自左向右的方法进行遍历，在访问了一个结点之后，它的子结点（如果有的话）按照从左到右的顺序依次放入队列的末尾，然后访问该队列头部的结点，这样的过程满足“第 n 层的结点必须在第 n+1 层的结点之前访问”的条件。被访问过的结点则从队列中出队。

相应的成员函数的实现在例 3.3 中给出。

【例 3.3】广度优先遍历。

```
template<class T>
void BinaryTree<T>::levelOrder(BinaryTreeNode<T> * root)
{
    using std::queue;
    queue<BinaryTreeNode<T> *> nodeQueue;//用队列来存放将要访问的结点
    BinaryTreeNode<T> * pointer = root;

    if(pointer)                          //如果根结点非空，将根结点移入队列
        nodeQueue.push(pointer);
    while(!nodeQueue.empty())
    {
        pointer = nodeQueue.front();    //读取队首结点
```

```
            visit(pointer);                    //访问当前结点
            nodeQueue.pop();                   //将访问过的结点移出队列
            if(pointer->leftChild)
                nodeQueue.push(pointer->leftChild);
            if(pointer->rightChild)
                nodeQueue.push(pointer->rightChild);
                                    //将访问过的结点的左右孩子结点依次加入到队尾
        }
    }
```

2．深度优先遍历

考虑到二叉树的基本结构，设 D 表示根结点，L 表示左子树，R 表示右子树，则对这三个部分进行访问的次序组合共有 6 种：DLR，DRL，LDR，LRD，RDL，RLD。若限定先左后右的顺序，则只剩下以下 3 种遍历方法。

1）前序遍历（DLR，也称为先序遍历、先根遍历）：

① 访问根结点（D）

② 前序遍历左子树（L）

③ 前序遍历右子树（R）

2）中序遍历（LDR，也称为中根遍历）：

① 中序遍历左子树（L）

② 访问根结点（D）

③ 中序遍历右子树（R）

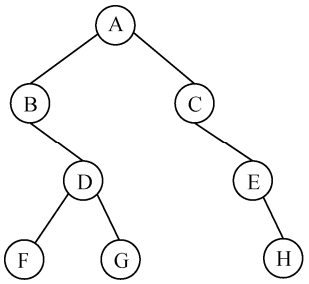
3）后序遍历（LRD，也称为后根遍历）：

① 后序遍历左子树（L）

② 后序遍历右子树（R）

③ 访问根结点（D）

归纳上面的三种遍历方式为：前序遍历的遍历次序为“根-左子树-右子树”，中序遍历的遍历次序为“左子树-根-右子树”，后序遍历的遍历次序为“左子树-右子树-根”。图 3-13 给出 3 种遍历方法的实例。



前序遍历结果：ABDFGCEH
中序遍历结果：BFDGACEH
后序遍历结果：FGDBHECA
层次遍历结果：ABCDEFGH

图 3-13　二叉树的遍历

3．深度优先遍历算法的实现

（1）递归算法

根据二叉树的递归定义，前序、中序和后序遍历算法可以用简单的递归方式来实现。具体实现如下：

【例 3.4】深度优先遍历的三种算法实现。

```
//前序遍历二叉树或其子树
template<class T>
void BinaryTree<T>::preOrder(BinaryTreeNode<T> * root)
{
    if(root != NULL)
    {
        visit(root);                          //访问当前结点
        preOrder(root->leftChild);            //访问左子树
        preOrder(root->rightChild);           //访问右子树
    }
};

//中序遍历二叉树或其子树
template<class T>
void BinaryTree<T>::inOrder(BinaryTreeNode<T> * root)
{
    if(root != NULL)
    {
        inOrder (root->leftChild);            //访问左子树
        visit(root);                          //访问当前结点
        inOrder (root->rightChild);           //访问右子树
    }
}

//后序遍历二叉树或其子树
template<class T>
void BinaryTree<T>::postOrder(BinaryTreeNode<T> * root)
{
    if(root != NULL)
    {
        postOrder(root->leftChild);           //访问左子树
        postOrder (root->rightChild);         //访问右子树
        visit(root);                          //访问当前结点
    }
}
```

（2）深度优先遍历的非递归实现

递归算法的效率一般比非递归算法的效率低，对于二叉树的遍历算法，可以利用非递归算法来实现。

1）前序遍历的非递归算法。对于前序遍历，由其定义可知，其结点的遍历顺序为“根-左子树-右子树”，左、右子树的访问也是按照这个顺序进行，所以该遍历顺序也可以归纳为从根结点开始，沿着左子树往下搜索，每到达一个结点，就访问它，直到访问的结点没有左子树为止，然后自下向上依次遍历这些访问过的结点的右子树。由于自下向上的依次遍历被访问过的每个结点的右子树，所以在设计该算法的时候，可以借助栈来存储每个访问过的结点的右子树的根，以便遍历完每个结点的左子树后，可以转到该结点的右子树。

非递归前序遍历二叉树的算法的主要思想是：每遇到一个结点，先访问该结点，并把该结点的非空右子树的根结点压入到栈中，然后遍历其左子树，重复该过程直到当前访问过的结点没有左子树时停止，然后从栈顶弹出待访问的结点，继续遍历，直到栈为空时停止。

【例 3.5】非递归前序遍历二叉树或其子树。

```
template<class T>
void BinaryTree<T>::PreOrderWithoutRecusion(BinaryTreeNode<T> * root)
{
    using std::stack;

    stack<BinaryTreeNode<T> * > nodeStack;      //存放待访问的结点的栈
    BinaryTreeNode<T> * pointer = root;         //保存根结点

    while(!nodeStack.empty() || pointer)        //栈为空时遍历结束
    {
        if(pointer){
            visit(pointer);                     //访问当前结点
            if(pointer->rightChild != NULL)
                  nodeStack.push(pointer->rightChild);
                                    //当前访问结点的右子树的根结点入栈
            pointer = pointer->leftChild;       //转向访问其左子树
        }
        else {                                  //左子树访问完毕，转向访问右子树
            pointer = nodeStack.top();          //读取栈顶待访问的结点
            nodeStack.pop();                    //删除栈顶结点
        }
    }
}
```

2）中序遍历的非递归算法。

对于中序遍历中访问每个结点的后继结点有如下规则：如果某个结点有右子树，则其后继结点为右子树的最左下侧的结点，如果该结点没有右子树，则其后继结点为其父结点。中序遍历最先访问最左下侧的结点，在搜索到达该结点的过程中，应将搜索路径上的每个结点用栈存储起来，存储的目的是在访问完当前子树时可以顺利转到其父结点。

非递归中序遍历二叉树算法的思想：从根结点开始向左搜索，每遇到一个结点，就将其压入栈中，然后去遍历其左子树，遍历完左子树后，弹出栈顶结点并访问它，然后遍历其右子树。例 3.6 给出了算法实现。

【例 3.6】 非递归中序遍历二叉树。

```
template<class T>
void   BinaryTree<T>::InOrderWithoutRecusion(BinaryTreeNode<T>   *
root)
{
    using std::stack;
    stack<BinaryTreeNode<T> * > nodeStack;      //存储待访问结点
    BinaryTreeNode<T> * pointer = root;         //保存根结点

    while(!nodeStack.empty() || pointer)        //栈为空时遍历结束
    {
        if(pointer){
              nodeStack.push(pointer);          //当前结点入栈
              pointer = pointer->leftChild;     //转向访问其左孩子
        }
        else {                              //左子树访问完毕，转向访问右子树
              pointer = nodeStack.top();        //读取栈顶待访问的结点

              visit(pointer);                   //访问当前结点
              pointer = pointer->rightChild;    //转向其右孩子
              nodeStack.pop();                  //删除栈顶结点
        }
    }
}
```

3）后序遍历的非递归算法。

后序遍历二叉树的非递归方法要复杂些，因为根结点是最后访问的，因此对任一结点，应该先访问其左子树，再访问其右子树，然后访问该结点，因此最先被访问的结点是最左下侧的叶子结点。访问过后某个结点后，找到其父结点，如果父结点的右子树没有被访问过，则访问父结点的右子树，然后再访问该父结点，如果没有右子树，则直接访问该父结点。重复该过程直到所有结点都被遍历到。由于在访问过一个结点之后，需要找到其父结点，因此在顺序向下搜索的过程中，每搜索到一个结点就存储到栈中，以便找到每个结点的父结点。

算法思想：从根结点开始，向左搜索，每搜索到一个结点就将其压入栈中，直到压入栈中的结点不再有左子树为止。读取栈顶结点，如果该结点有右子树且未被访问，则访问其右子树，否则，则访问该结点，并从栈中移除。具体实现在例 3.7 中给出。

【例 3.7】非递归后序遍历二叉树。

```
template <class T>
```

```
void BinaryTree<T>::PostOrderWithoutRecusion(BinaryTreeNode<T> * root)
{
    using std::stack;
    stack<BinaryTreeNode<T> * > nodeStack; //存储待访问结点
    BinaryTreeNode<T> * pointer = root;    //保存根结点
    BinaryTreeNode<T> * pre = root;        //保存前一个被访问的结点

    while(pointer){
        for( ; pointer->leftChild != NULL; pointer = pointer->leftChild)
            nodeStack.push(pointer);       //向左搜索

        while(pointer != NULL && (pointer->rightChild == NULL ||
 pointer->rightChild == pre))
{
                    //当前结点没有右孩子或者右孩子刚被访问过,则访问该结点
            visit(pointer);
            pre = pointer;                 //记录刚被访问过的结点
            if(nodeStack.empty())
                return;
            pointer = nodeStack.top();     //取栈顶结点
            nodeStack.pop();
        }
        nodeStack.push(pointer);
        pointer = pointer->rightChild;     //转向当前结点的右子树
    }
}
```


3.2.7 线索二叉树

前面介绍的二叉树的遍历算法，都是使用了线性结构栈或者队列来存储结点信息以及遍历次序。因此，在执行程序的时候不仅需要花费额外的时间来维护栈或队列中的数据，还要为栈或队列留出足够的空间，这使得算法在时间效率和空间效率上存在一定的问题。除了上述问题外，对于一棵树，如何直接找到任意结点的前驱和后继结点也是一个值得讨论的问题。

在遍历的时候，采用了栈这种数据结构来存放结点，但是如果将栈加入到树形结构中，则可以提高时间效率。在实际操作中是采用在结点中引入线索（thread）的这种方式来将栈加入到树形结构中去的。结点中加入线索的树称为线索树。这里的线索就是指向该结点的前驱和后继的指针，因此加入线索后的结点中应该包含四个指针（左孩子，右孩子，前驱结点，后继结点），这样虽然解决了时间问题，但是指针数量增加了一倍，从而增加了空间上的开销。

n 个结点的二叉链表中含有 n+1 个空指针域，如果利用二叉链表中的这些空指针域来存放指向结点在某种遍历次序下的前驱和后继结点的指针，则可以解决上述的空间问

题。为了区分指针指向的是孩子结点还是前驱、后继结点，对每个结点增加两个标志域，用来记录对应指针的意义。

要实现线索二叉树，就必须定义二叉链表结点的数据结构，如图 3-14 所示。

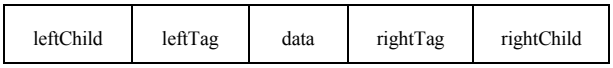


图 3-14　线索二叉树的结点数据结构

说明：

1）leftTag = 0 时，表示 leftChild 指向该结点的左孩子。

2）leftTag = 1 时，表示 leftChild 指向该结点的线性前驱结点。

3）rightTag = 0 时，表示 rightChild 指向该结点的右孩子。

4）rightTag = 1 时，表示 rightChild 指向该结点的线性后继结点。

以上述二叉链表结点为数据结构所构成的二叉链表叫做线索二叉链表，对二叉树以某种次序遍历将其变成线索二叉树的过程叫做线索化。

线索二叉树的结点类的具体代码如下：

【例 3.8】线索二叉树的结点类。

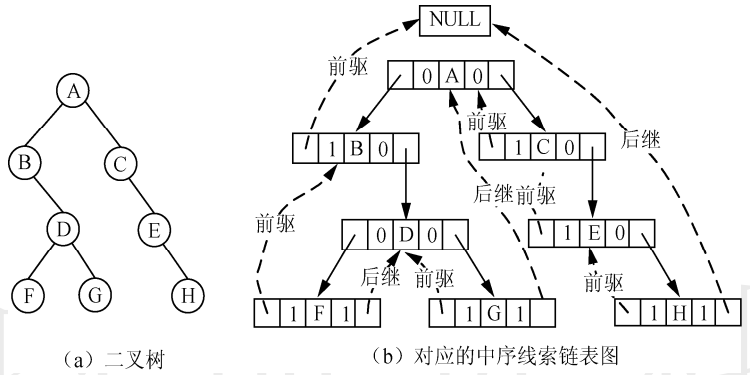
```
template <class T>
class ThreadBinaryTreeNode
{
friend class ThreadBinaryTree<T>;
private:
    int leftTag, rightTag;                  //左右标志位
    ThreadBinaryTreeNode<T> * leftChild;    //前驱或左子树
    ThreadBinaryTreeNode<T> * rightChild;   //后继或右子树
    T element;                              //结点数据域
public:
    ThreadBinaryTreeNode();
    ThreadBinaryTreeNode(const T& ele);     //构造函数
    ThreadBinaryTreeNode(const T& ele, ThreadBinaryTreeNode<T> * l,
ThreadBinaryTreeNode<T> * r );
    ThreadBinaryTreeNode<T> * getLeftChild() const;
    ThreadBinaryTreeNode<T> * getRightChild() const;
    void setLeftChild(ThreadBinaryTreeNode<T> * l);
    void setRightChild(ThreadBinaryTreeNode<T> * r);
    T getValue() const;                     //返回该结点的数据值
    void setValue(const T& val);            //设置该结点的数据域的值
};
```

【例 3.9】线索二叉树类。

```
template <class T>
class ThreadBinaryTree{
```

```
        private:
            ThreadBinaryTreeNode<T> * root;              //根结点指针
        public:
            ThreadBinaryTree();                          //构造函数
            ThreadBinaryTree(ThreadBinaryTreeNode<T> * r);
            ~ThreadBinaryTree();                         //析构函数
            ThreadBinaryTreeNode<T> * getRoot();         //返回根结点指针
            void  InThread(ThreadBinaryTreeNode<T>  *  root,  ThreadBinary
TreeNode<T> * pre );                                     //中序线索化二叉树
            void InOrder(ThreadBinaryTreeNode<T> * root);  //中序遍历
        };
```

图 3-15 给出二叉树经过中序线索化后得到的中序线索链表图，图中实线表示指针，虚线表示线索。结点 B 的左线索为空，表示 B 是中序遍历的开始结点，无前驱。结点 H 的右线索为空，表示 H 是中序遍历的终端结点，无后继。



（a）二叉树　　（b）对应的中序线索链表图

图 3-15　中序线索二叉树及其存储结构

二叉树线索化的实质是：按某种次序遍历二叉树，在遍历过程中用线索取代空指针。该算法应附设一个指针 pre 始终指向刚刚访问过的结点（pre 的初值应为 NULL），而指针 current 指示当前正在访问的结点。结点 pre 是结点 current 的前驱，而 current 是 pre 的后继。中序线索化递归实现的代码如下：

【例 3.10】中序线索化二叉树。

```
        template <class T>
        void ThreadBinaryTree<T>::InThread( ThreadBinaryTreeNode<T>* root,
ThreadBinaryTreeNode<T>* &pre)
        {
            ThreadBinaryTreeNode<T>* current;
            current = root;                          //current 为当前访问结点
            if(current != NULL)
            {
                InThread(current->leftChild, pre); //中序线索化左子树
                if( current->leftChild == NULL)
```

```
            {
                current->leftChild = pre;
                current->leftTag = 1;           //建立前驱线索
            }
            if((pre)&&(pre->rightChild == NULL))
            {
                pre->rightChild = root;
                pre->rightTag = 1;              //建立后继线索
            }
            pre = current;
            InThread(current->rightChild, pre);//中序线索化右子树
        }
    }
```

二叉树的前序线索化和后序线索化方法同上面的中序线索化算法类似，读者可以自己完成实现代码。

对于中序线索二叉树，其对应的遍历方法描述如下：

先从中序线索二叉树的根结点出发，一直沿左指针，找到“最左”结点（它一定是中序遍历的第一个结点），然后反复地查找当前结点在中序下的后继。检索中序线索二叉树某结点的线性后继结点的算法：

1）如果该结点的 rightTag = 1，那么 rightChild 就是它的线性后继结点；

2）如果该结点的 rightTag = 0，那么该结点右子树“最左边”的终端结点就是它的线性后继结点。

因此可以得到中序线索二叉树的遍历算法如下：

【例 3.11】中序遍历线索二叉树。

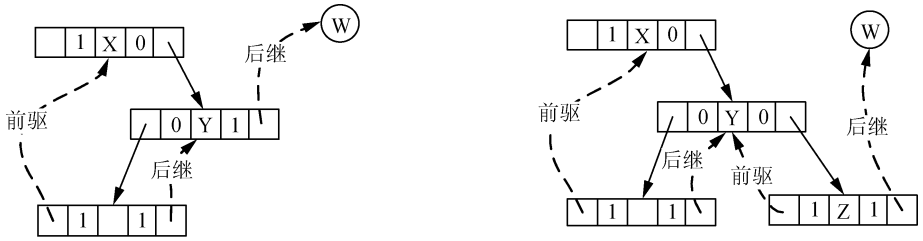
```
template<class T>
void ThreadBinaryTree<T>::InOrder(ThreadBinaryTreeNode<T>* root)
{
    ThreadBinaryTreeNode<T>* current;
    current = root;

    while(current->leftTag == 0)
        current = current->leftChild;       //寻找中序遍历的第一个结点
    while(current)
    {
        visit(current);                     //访问当前结点
        if(current->rightTag == 1)
            current = current->rightChild; //沿线索寻找后继
        else
        {
            current = current->rightChild;
            while(current && current->leftTag == 0)
                current = current->leftChild;  //寻找最左终端结点
        }
```

```
        }
    }
```

二叉树加上线索后，当插入或删除一结点时，可能会破坏原来二叉树的线索，所以在线索二叉树中插入或删除结点的时候，也要修改相关结点的线索以保持正确的前驱后继关系。这里以中序线索二叉树的插入操作为例来说明该运算的步骤。

插入一个结点分两种情况：一种是新结点作为某个结点的左孩子插入，另一种是新结点作为某个结点的右孩子插入。这里以第二种情况为例进行讨论。新结点作为某个结点的右孩子插入的时候，也存在两种情况：

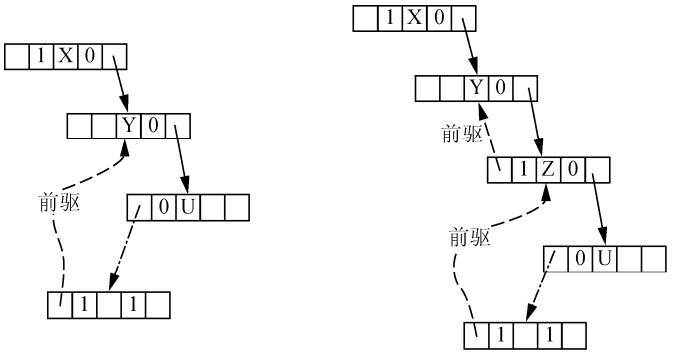
第一种情况如图 3-16 所示，新结点 Z 作为结点 Y 的右孩子插入，结点 Y 的右孩子为空，则插入新结点的方法很简单，只需要将结点 Y 的后继变为结点 Z 的后继，结点 Y 作为结点 Z 的前驱，结点 Z 作为结点 Y 的右孩子即可。



（a）插入前　　（b）插入后

图 3-16　第一种插入情况

第二种情况如图 3-17 所示，若结点 Y 的右孩子不为空，则插入结点 Z 后，Y 的右孩子变为 Z 的右孩子结点，Y 变为 Z 的前驱结点，Z 变为 Y 的右孩子结点。这时还需要修改原来 Y 的右子树中“最左下端”结点的左指针域，使它由原来的前驱指向结点 Y 变为前驱指向结点 Z。



（a）插入前　　（b）插入后

图 3-17　第二种插入情况

插入算法的实现如下：

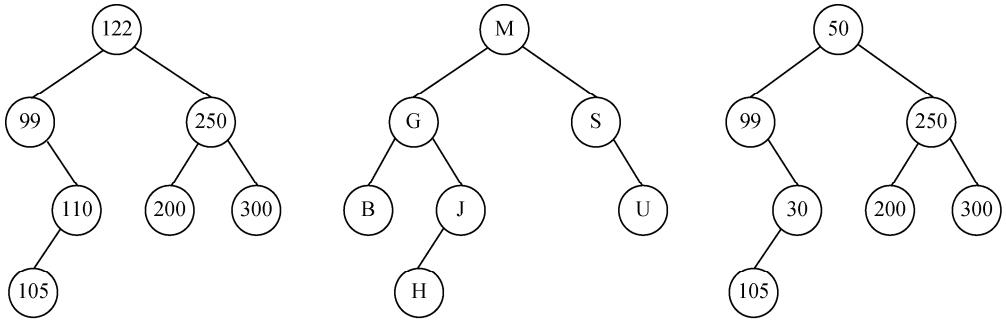
【例 3.12】中序线索二叉树的插入操作。

```
template<class T>
void ThreadBinaryTree<T>::InsertNode(
ThreadBinaryTreeNode<T>* pointer, ThreadBinaryTreeNode<T>* newPointer)
{
    if(pointer->rightTag == 1)              //pointer 无右孩子
    {
        newPointer->rightChild = pointer->rightChild;
                                            //pointer 的后继变为
                                            newPointer 的后继
        newPointer->rightTag = 1;
        pointer->rightChild = newPointer;   //newPointer 成为pointer
                                            的右孩子
        pointer->rightTag = 0;
        newPointer->leftChild = pointer;    //pointer 成为newPointer
                                            的前驱
        newPointer->leftTag = 1;
    }
    else                                    //pointer 有右孩子
    {
        ThreadBinaryTreeNode<T>* s = pointer->rightChild;
        while(s->leftTag == 0)
            s = s->leftChild;   //查找 p 结点的右子树的"最左下端"结点
        newPointer->rightChild = pointer->rightChild;
                        //pointer 的右孩子变为 newPointer 的右孩子
        newPointer->rightTag = 0;
        newPointer->leftChild = pointer;    //pointer 变为newPointer
                                            的前驱
        newPointer->leftTag = 1;
        pointer->rightChild = newPointer;   //newPointer 变为pointer
                                            的右孩子
        s->leftChild = newPointer;
                //newPointer 变为pointer 原来右子树的"最左下端"结点的前驱
    }
}
```

3.2.8 二叉搜索树

二叉搜索树（Binary Search Tree），又称为“二叉排序树”、“二叉查找树”。二叉搜索树可以为空树，也可以是这样定义的二叉树：该树的每个结点都有一个作为搜索依据的关键码，对任意结点而言，其左子树（如果存在）上的所有结点的关键码均小于该结点的关键码，其右子树（如果存在）上的所有结点的关键码都大于该结点的关键码。下面图 3-18 中给出了二叉搜索树的例子，其中（a）、（b）是二叉搜索树，而（c）不是二

叉搜索树。



（a）二叉搜索树　　（b）二叉搜索树　　（c）非二叉搜索树

图 3-18　二叉搜索树举例

二叉搜索树的基本操作有：查找、插入和删除。下面分别介绍这三种基本操作。

1．二叉搜索树的查找

由于二叉搜索树的特殊定义，使得二叉树结点之间的大小关系可以通过结点间的位置关系来得到，因此二叉搜索树的查找操作有规律可循，无需遍历整棵二叉树。常用的算法为分割式查找法，具体步骤如下：

1）若根结点的关键码的值等于待查找的键值，则查找成功。

2）否则，若键值小于根结点的关键码的值，则继续在其左子树中查找；若键值大于根结点的关键码的值，则继续在其右子树中查找。

其实现代码如下：

【例 3.13】二叉搜索树的查找。

```
        template<class T>
        BinarySearchTreeNode<T>*
BinarySearchTree<T>::search(BinarySearchTreeNode<T>* root, T key)
        {
            BinarySearchTreeNode<T>* current = root;
            while((NULL != root) && (key != current->getValue()))
                                    //当前结点的 key 值等于查询的值时，退出循环
            {
            current = (key < current->getValue() ? search(current->leftChild,
key) : search(current->rightChild, key));
                                    //根据当前结点的值的大小决定移动方向
            }
            return current;
        }
```

2．二叉搜索树的插入

将一个结点插入到二叉搜索树中，且要保持二叉搜索树的结构特征，关键是要找到

合适的插入位置。所以，进行插入操作时，先进行插入位置的查找。方法是从根开始，将要插入的结点的关键码与树中每一个结点的关键码进行比较，如果大于该结点，则下一个比较的位置移到该结点的右子结点。否则，下一个比较的位置移到该结点的左子结点上。这样一直比较到树的最“底下”的结点 x（叶子结点）。然后比较关键码与该结点的关键码的大小关系，把需要插入的结点设置为该结点 x 的左子结点，或是右子结点。具体代码如下：

【例 3.14】向二叉搜索树中插入结点。

```
template<class T>
void BinarySearchTree<T>::insertNode(const T& value) {
    BinarySearchTreeNode<T> *p = root, *prev = NULL;
    while (p != 0)
    {                                       //新结点查找位置
        prev = p;                           //记录父结点
        if( p->getValue() < value )
            p = p->rightChild;
        else
            p = p->leftChild;
    }
    if( root == NULL )                      //如果是空树，将新结点作为根结点
        root = new BinarySearchTreeNode<T>(value);
    else if( prev->getValue() < value )
                                //根据关键码决定设置为左子结点还是右子结点
        prev->rightChild = new BinarySearchTreeNode<T>(value);
    else
        prev->leftChild = new BinarySearchTreeNode<T>(value);
}
```


给定一个关键码集合，可以通过结点插入的算法来得到该关键码集合对应的二叉搜索树，方法为：从一棵空的二叉搜索树开始，将关键码按照插入算法一个个地插入，最终得到二叉搜索树。

将关键码集合转化为对应的二叉搜索树，实际上是对集合里的关键码进行了排序。按中序遍历二叉搜索树就能得到按照从小到大的顺序排列好的关键码序列。

3．二叉搜索树的删除

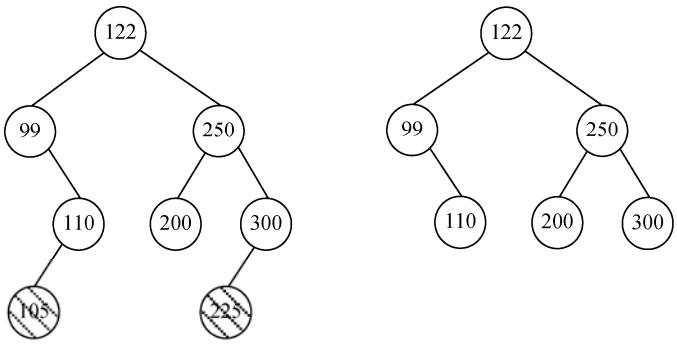
在删除结点操作中，最重要的是如何在删除结点后，仍然保持二叉搜索树的特征。

这里结点的删除分为三种情况：

1）如果被删除的结点 p 没有子树，这种情况最简单，直接将结点 p 删除就可以了，对二叉搜索树的基本特征没有任何影响。

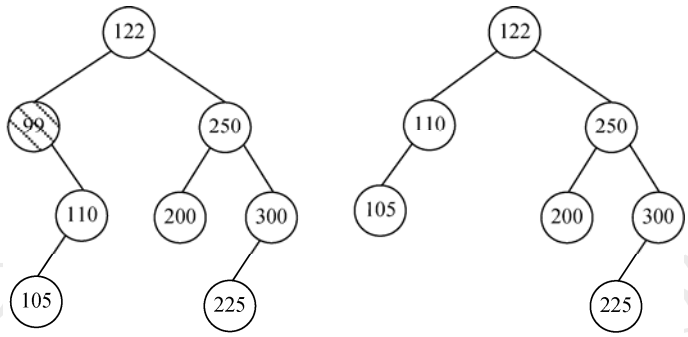
2）如果被删除的结点 p 只有一棵子树，就用那个唯一的子树的根结点来替换要删除的那个结点 p，然后删掉需要删除的结点。这样就不会影响二叉搜索树的基本特征。

这两种情况的删除示例在图 3-19 和图 3-20 中给出，图中带阴影的结点为要删除的结点。



（a）删除叶子结点　　（b）删除后

图 3-19　二叉搜索树删除叶子结点示例



（a）删除只有一个孩子的结点　　（b）删除后

图 3-20　二叉搜索树删除只有一棵子树的结点示例

3）如果被删除结点 p 有两棵子树，这种情况就比前面两种情况复杂多了，解决该问题的关键是要使删除结点后的二叉树仍然保持二叉搜索树的基本性质。这种情况的解决方法有两种：

① 合并删除：查找到被删除的结点 p 的左子树中按中序遍历的最后一个结点 r，将结点 r 的右指针赋值为指向结点 p 的右子树的根，然后用结点 p 的左子树的根代替被删除的结点 p，最后删除结点 p。

图 3-21 给出合并删除方法的示意图，其中结点 p 为要删除的结点，将结点 p 的右子树作为其左子树中最右侧结点的右子树，用结点 p 的左子树的根结点来替换结点 p，最后删除结点 p。

② 复制删除：选取一个合适的结点 r，并将该结点的关键码复制给被删除结点 p，然后将 r 结点删除。结点的选取方法有两种：结点 p 的左子树中关键码最大的结点（左

子树中最右侧的结点）或者结点 p 的右子树中关键码最小的结点（右子树中最左侧的结点）。如果 r 结点有左子树或右子树，则将其唯一的子树放置到 r 结点原来的位置上。

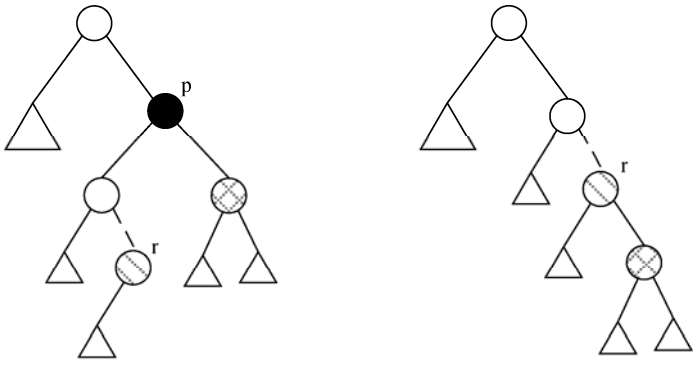


图 3-21　合并删除

图 3-22 中给出复制删除的一个实例，其中（a）中 p 指针指向的结点为要删除的结点，例子中选择了 p 结点的左子树中关键码最大的结点（也可以是右子树中关键码最小的结点 q），也就是 r 指针指向的关键码为 110 的结点，然后将关键码 110 赋值给指针 p 指向的结点，最后将 r 指针指向的结点删除。

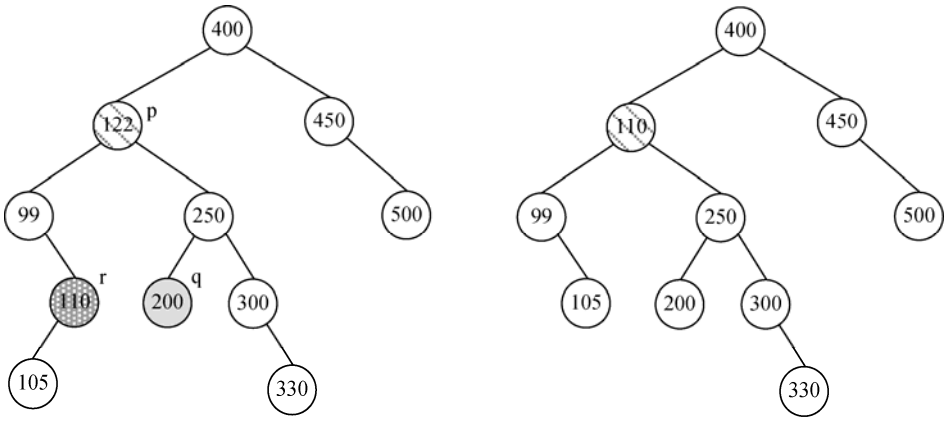


图 3-22　复制删除实例

【例 3.15】合并删除的算法实现。

```
template<class T>

void BinarySearchTree<T>::deleteByMerging(BinarySearchTree<T>*& node)
{
```

```
    BinarySearchTree<T> *tmp = node;
    if(node != NULL)
    {
            //如果被删除结点没有右子树，用其左子树的根结点来代替被删除结点
        if(!node->rightChild)
            node = node->leftChild;
            //如果被删除结点没有左子树，用其右子树的根结点来代替被删除结点
        else if(node->leftChild == NULL)
            node = node->rightChild;
        else{                    //如果被删除结点左右子树都存在
            tmp = node->leftChild;
            while(tmp->rightChild != NULL)
                                 //查找左子树中按中序遍历的最后一个结点
                tmp = tmp->rightChild;
            //将查找到的结点的右指针赋值为被删除结点的右子树的根
            tmp->rightChild = node->rightChild;
            tmp = node;
            node = node->leftChild; //用左子树的根结点代替被删除结点
        }
        delete tmp;
    }
}
```

【例 3.16】复制删除算法实现。

```
template<class T>
void BinarySearchTree<T>::deleteByCopying(BinarySearchTreeNode<T>*&
node)
{
    BinarySearchTreeNode<T>* previous *tmp = node;
            //如果被删除结点没有右子树，用其左子树的根结点来代替被删除结点
    if(node->rightChild == NULL)
        node = node->leftChild;
            //如果被删除结点没有左子树，用其右子树的根结点来代替被删除结点
    else if(node->leftChild == NULL)
        node = node->rightChild;
    else                                  //如果被删除结点左右子树都存在
    {
        tmp = node->leftChild;
        previous = node;
        while(tmp->rightChild != NULL) //查找左子树中关键码最大的结点
        {
            previous = tmp;
            tmp = tmp->rightChild;
        }
        node->value = tmp->value;   //将查找到的结点的值赋值给被删除结点
        if(previous == node)
```

```
                previous->leftChild = tmp->leftChild;
            else
                previous->rightChild = tmp ->leftChild;
        }
        delete tmp;
    }
```

3.2.9 平衡二叉树

由于二叉搜索树的时间复杂度受输入顺序的影响，在最好的情况下复杂度为O(log n)，最坏情况下复杂度为O(n)，为了使二叉搜索树的时间复杂度始终保持O(log n)级的平衡状态，Adelson-Velskii 和 Landis 发明了 AVL 树（平衡二叉树）。下面给出相关定义：

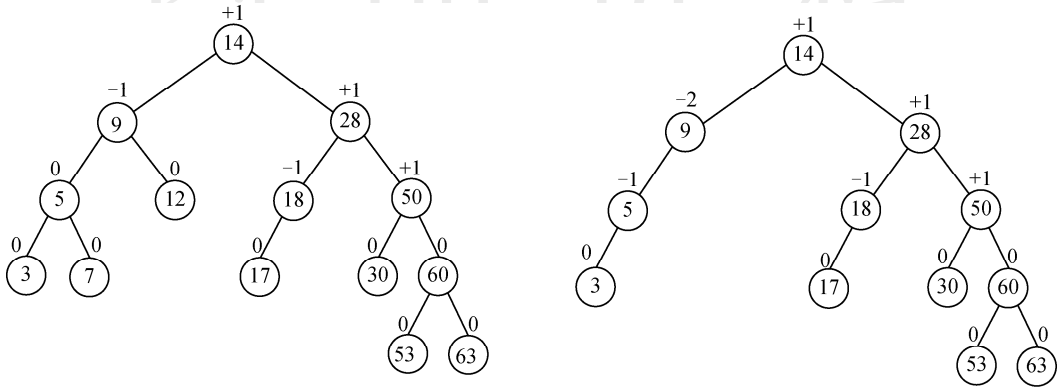
结点的平衡因子：二叉树中某结点的右子树的高度与左子树的高度之差称为该结点的平衡因子。

平衡二叉树：平衡二叉树或者是一棵空树，或者是具有以下性质的二叉搜索树：

1）树中任一结点的平衡因子的绝对值不超过 1；

2）任一结点的左子树和右子树都是平衡二叉树。

因此，对于平衡二叉树中的任一结点来说，其平衡因子只有三个可能的取值：0，1，−1。图 3-23 中给出平衡二叉树的例子。其中（a）为平衡二叉树，因为其每个结点的平衡因子的绝对值都不超过 1，而（b）图却不是平衡二叉树，因为关键码为 9 的结点的平衡因子为−2，不满足平衡二叉树的性质。



（a）平衡二叉树　　（b）非平衡二叉树

图 3-23　平衡二叉树实例

平衡二叉树的基本操作主要有：查找、插入和删除。由于平衡二叉树是一种特殊的二叉搜索树，因此平衡二叉树的查找算法与二叉搜索树的查找算法一致。下面重点介绍一下平衡二叉树的插入和删除操作。

1．平衡二叉树的插入操作

如果要在图 3-23 中的（a）图中插入一个关键码为 2 的结点，按照二叉搜索树的插入过程，其插入后的情况如图 3-24 所示。

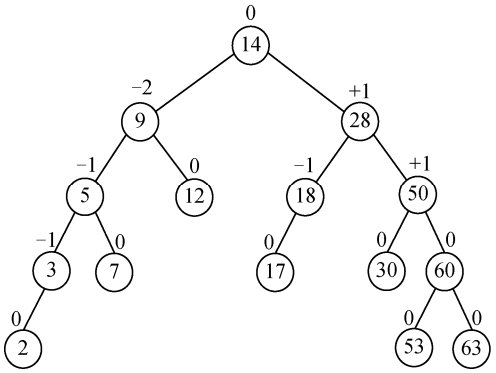


图 3-24　插入结点后的图示

从图 3-24 中可以看出，插入关键码为 2 的结点后，该平衡二叉树的其他结点的平衡因子会被改变，关键码为 3，5 的结点的平衡因子由 0 变为−1，关键码为 9 的结点的平衡因子则由−1 变为−2，而这个变化破坏了原来二叉树的平衡性，使其不再满足平衡二叉树的性质。为了使插入结点后的二叉树仍然保持平衡二叉树的性质，就需要在插入结点后判断插入行为是否破坏了平衡性，如果破坏了平衡性，则调整树的结构使其平衡化。

平衡二叉树插入结点的平衡化调整方法如下：

每插入一个新结点时，AVL 树中相关结点的平衡状态会发生改变。因此，在插入一个新结点后，需要从插入位置沿通向根的路径回溯，检查各结点的平衡因子（左、右子树的高度差）。

如果在某一结点处发现高度不平衡，停止回溯。从发生不平衡的结点起，沿刚才回溯的路径取直接下两层的结点。

如果这三个结点（称为被选取结点）处于一条直线上，则采用单旋转进行平衡化。单旋转可按其方向分为左单旋转和右单旋转，其中一个是另一个的镜像，其方向与不平衡的形状相关。

如果这三个结点处于一条折线上，则采用双旋转进行平衡化。双旋转分为先左后右双旋转和先右后左双旋转两类。

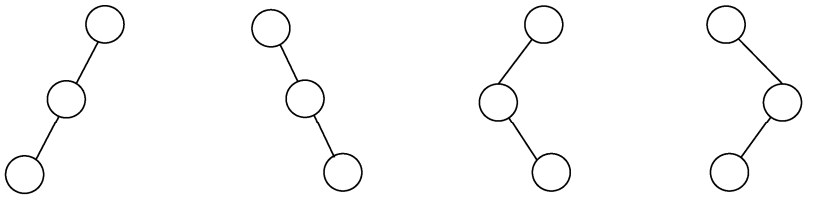
因此，调整的方法有 4 种：右单旋转，左单旋转，左右双旋转，右左双旋转，这四种情况下的三个被选取结点的位置情况分别在图 3-25 中给出。

下面将详细分析这 4 种方法的具体操作。

第一种情况：右单旋转。

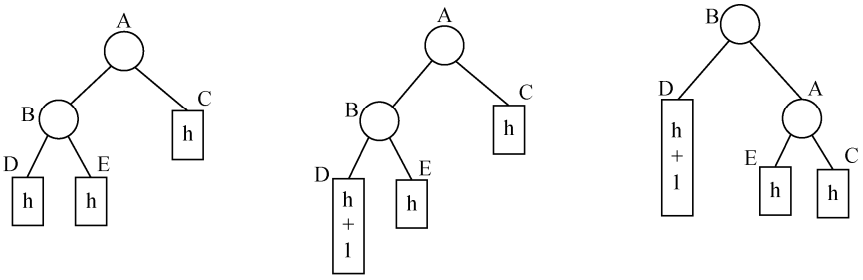
图 3-26（a）中在左子树 D 上插入新结点使其高度增 1，导致结点 A 的平衡因子变

为-2，破坏了平衡性。为使树恢复平衡，从 A 沿插入路径连续取 3 个结点 A、B 和 D，它们处于一条方向为“/”的直线上，符合右单旋转的情况，所以需要做右单旋转。



（a）右单旋转　（b）左单旋转　（c）左右双旋转　（d）右左双旋转

图 3-25　旋转情况示例



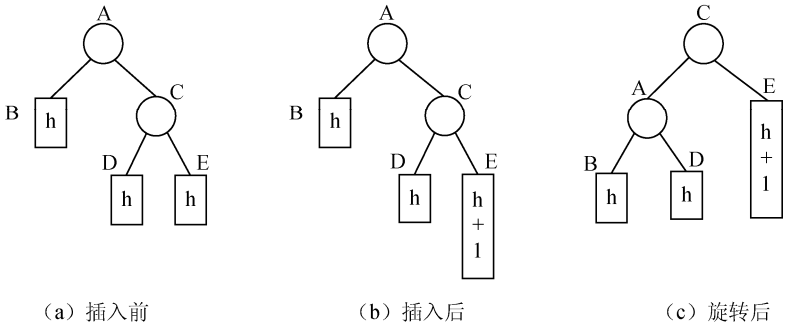
（a）插入前　（b）插入后　（c）旋转后

图 3-26　右单旋转

右单旋转的方法为：以结点 B 为旋转轴，将结点 A 顺时针旋转，即将 A 的左孩子 B 向右上旋转代替 A 成为根结点，将 A 结点向右下旋转成为 B 的右孩子的根结点，而结点 B 的原右子树 E 则作为 A 结点的左子树。

第二种情况：左单旋转。

在图 3-27 的（a）图中，如果在子树 E 中插入一个新结点，该子树高度增加 1，导致结点 A 的平衡因子变成+2，出现不平衡。



（a）插入前　（b）插入后　（c）旋转后

图 3-27　左单旋转

沿插入路径检查三个结点 A、C 和 E。它们处于一条方向为“\”的直线上，符合左单旋转的情况，因此需要做左单旋转。

左单旋转方法：以结点 C 为旋转轴，让结点 A 逆时针旋转。将结点 A 的孩子 C 向左上旋转代替结点 A 成为根结点，将 A 结点向左下旋转成为结点 C 的左子树的根结点，而结点 C 原来的左子树 D 则作为结点 A 的右子树。

第三种情况：左右双旋转。

在图 3-28 的（a）图中，在子树 F 或 G 中插入新结点，得到插入结点后的图（b），从（b）中可以看出该子树的高度增 1，结点 A 的平衡因子变为-2，发生了不平衡。于是需要进行平衡化调整。

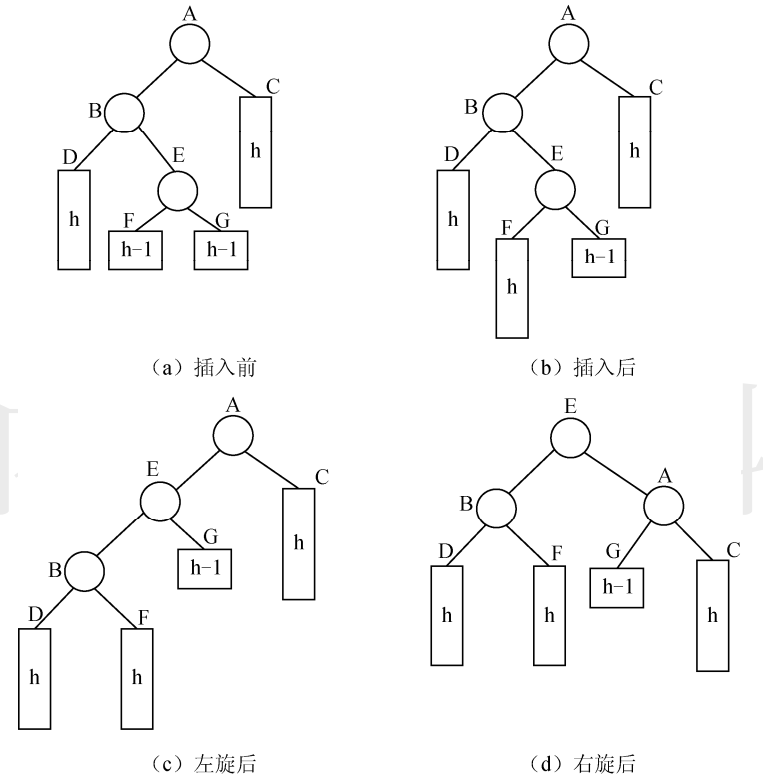


图 3-28　左右双旋转

从结点 A 起沿插入路径选取 3 个结点 A、B 和 E，它们位于一条形如“〈”的折线上，符合左右双旋转的情况，因此需要进行先左后右双旋转。

首先以结点 E 为旋转轴，将结点 B 反时针旋转，做左单旋转（以结点 E 代替原来结点 B 的位置，结点 E 原来的左子树 F 作为结点 B 的右子树）。左单旋转后得到的如图 3-29（c）所示，结点 A，E 仍然不满足平衡性，于是再以结点 E 为旋转轴，将结点 A 顺时针旋转，做右单旋转（参照右单旋转的步骤），得到结果 3-29（d），满足平衡二

叉树的性质，平衡化调整结束。

第四种情况：右左双旋转。

右左双旋转是左右双旋转的镜像。

图 3-29（a）中在子树 F 或 G 中插入新结点，得到插入结点后的图 3-29（b），从图 3-29（b）中可以看出该子树高度增 1，结点 A 的平衡因子变为 2，发生了不平衡。

从结点 A 起沿插入路径选取 3 个结点 A、C 和 D，它们位于一条形如“〉”的折线上，符合右左双旋转情况，需要进行先右后左的双旋转。

首先做右单旋转：以结点 D 为旋转轴，将结点 C 顺时针旋转，以结点 D 代替原来结点 C 的位置，结点 D 的右孩子 G 则作为结点 C 的左孩子。左单旋转后得到的结果如图 3-29（c）所示，结点 A，D 仍然不满足平衡性，再做左单旋转：以结点 D 为旋转轴，将结点 A 反时针旋转，恢复树的平衡。

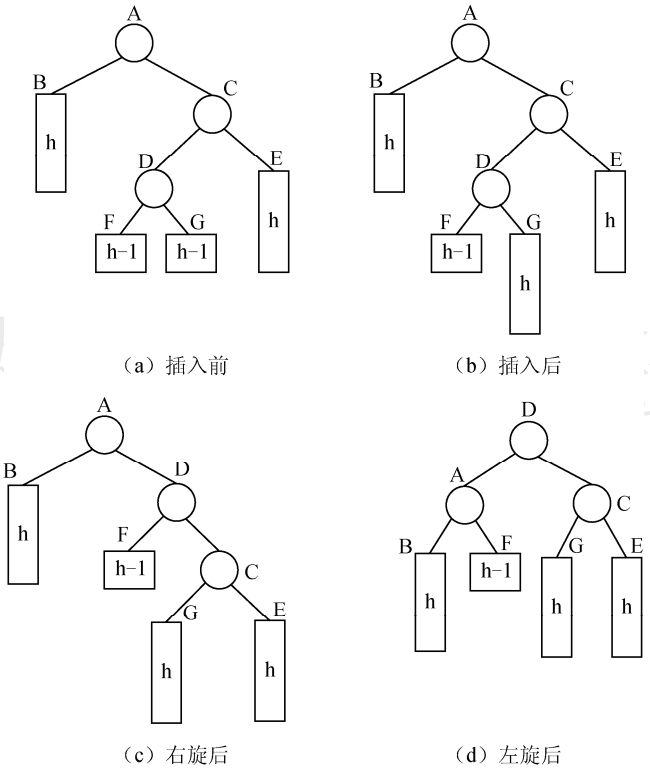


图 3-29　右左双旋转

按照上面介绍的平衡二叉树调整算法，可以从一棵空树开始，通过输入一系列关键码，逐步建立平衡二叉树。下面给出建立平衡二叉树的实例。

例如，输入关键码序列为 { 16, 3, 7, 11, 9, 26, 18, 14, 15 }，插入和调整过程在图 3-30 中给出。图中不带文字的箭头表示插入结点步骤，带文字的箭头表示进行旋转调整，调整方法由箭头上的描述文字给出。

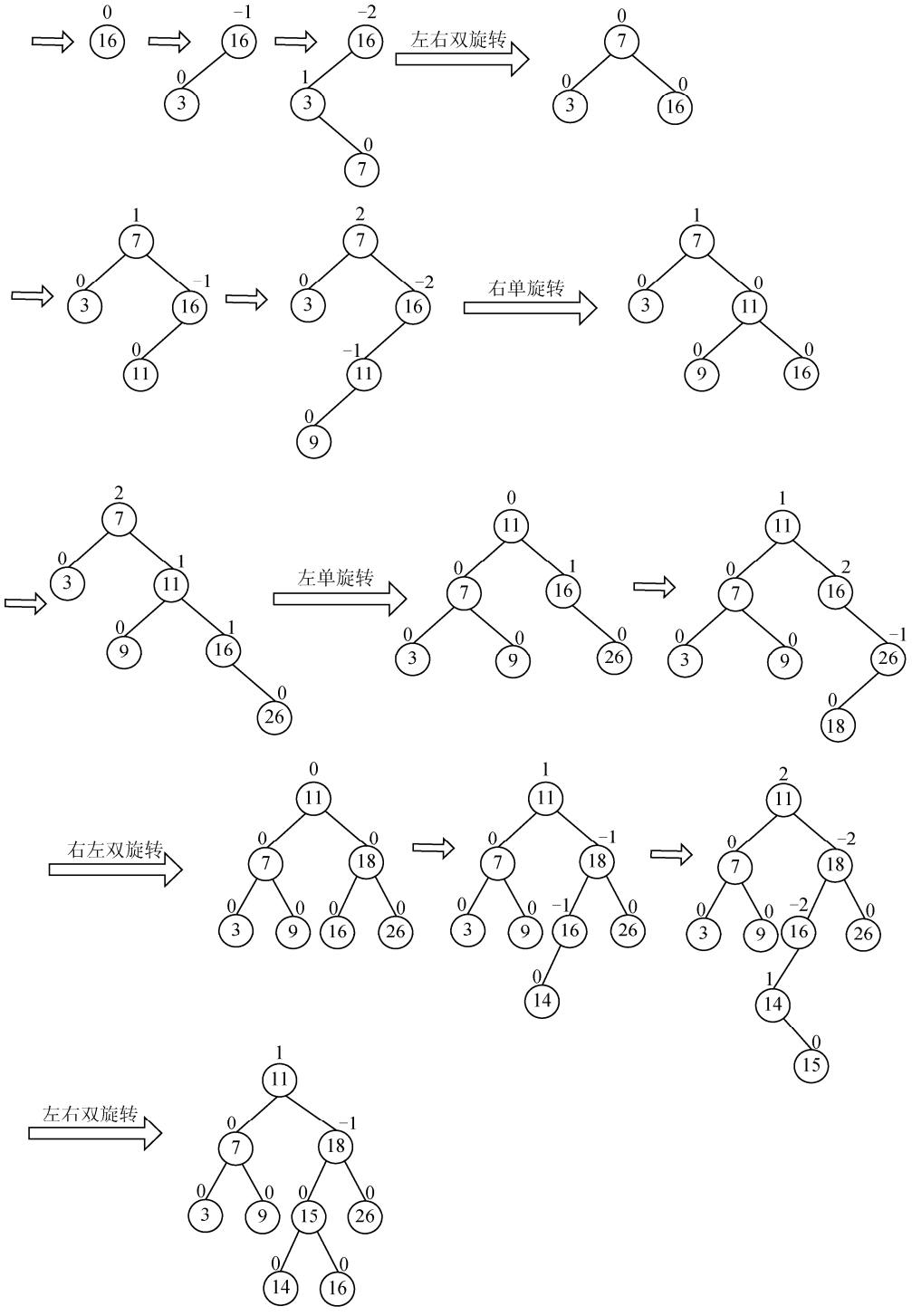


图 3-30　从空树开始建立平衡二叉树

2．平衡二叉树的删除

在平衡二叉树上删除结点 x 的步骤如下：

1）用一般的二叉搜索树的删除算法找到并删除结点 x 。

2）沿根结点到被删除结点的路径的逆向逐层向上回溯，并重新计算 x 的祖先结点的平衡因子，必要时修改 x 祖先结点的平衡因子。

3）回溯途中如果发现某个祖先结点失去了平衡性，就要进行调整使之平衡。

4）如果平衡化调整后，子树的高度比调整前降低了，需要继续回溯，即继续沿着通往根的路径上考查其父结点的平衡因子，重复上面的过程。在平衡二叉树上删除一个结点，可能引起多次平衡化调整。

5）如果平衡化调整后子树的高度不变，则停止回溯。

具体删除过程中会遇到多种情况，下面以在左子树删除结点为例来进行分析。

情况一（图 3-31）：祖先结点 p 的平衡因子原为 0，在左子树上删除结点 x 后，左子树的高度降低 1，右子树的高度不变，因此以祖先结点 p 为根结点的子树高度不变，因此将 p 结点的平衡因子修改为 1 后，调整结束。

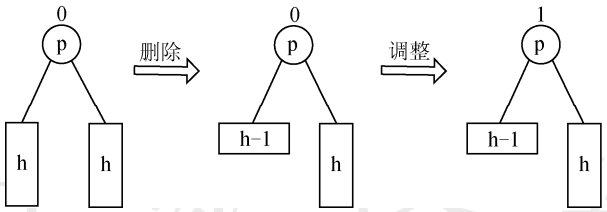


图 3-31　删除调整情况一

情况二（图 3-32）：祖先结点 p 的平衡因子原为-1，在左子树上删除结点 x 后，左子树的高度降低 1，右子树的高度不变，此时左右子树的高度一致，因此祖先结点 p 的平衡因子变为 0，但是以 p 为根结点的子树的高度却减少了 1，所以在将 p 结点的平衡因子修改为 0 后，调整不能停止，应该继续回溯。

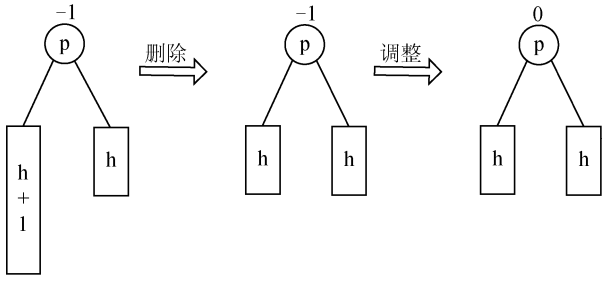


图 3-32　删除调整情况二

情况三（图 3-33）：祖先结点 p 的平衡因子原为 1，这时又分为三种情况来讨论。

1）图 3-33（a）中的情况为祖先结点 p 的右子树根结点 r 的平衡因子为 0 时，从结点 p 的左子树中删除结点 x 后，左子树高度降低 1，右子树高度不变，因此结点 p 的平衡因子变为 2，破坏了平衡性，于是进行相应的左单旋转调整，调整后以结点 r 为根结点的子树的高度不变，因此调整停止。

2）图 3-33（b）中的情况为祖先结点 p 的右子树根结点 r 的平衡因子为 1 时，从结点 p 的左子树中删除结点 x 后，左子树高度降低 1，右子树高度不变，因此结点 p 的平衡因子变为 2，破坏了平衡性，于是进行相应的左单旋转调整，调整后以结点 r 为根结点的子树的高度降低了 1，因此继续回溯。

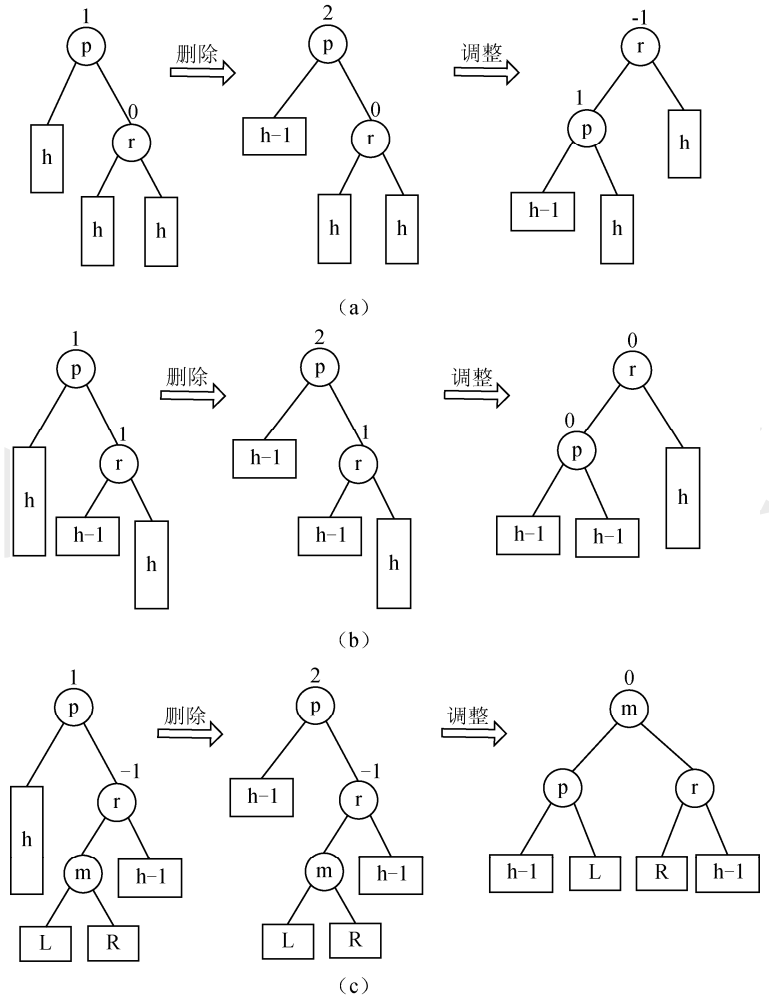


图 3-33　删除调整情况三

3）图 3-33（c）中的情况为祖先结点 p 的右子树根结点 r 的平衡因子为−1 时，从结

点 p 的左子树中删除结点 x 后，左子树高度降低 1，右子树高度不变，因此结点 p 的平衡因子变为 2，破坏了平衡性。由于结点 r 的左子树的根结点是结点 m，从结点 r 的平衡因子和其右子树的高度可以判断出以结点 m 为根结点的子树的高度为 h，而结点 m 的平衡因子大小不确定，但是必定是 1，0，−1 中的一个，所以结点 m 的左子树 L 和右子树 R 的高度关系只能有以下几种组合情况：(L：h−1, R：h−2)，(L：h−1, R：h−1)，(L：h−2, R：h−1)。因此在进行相应的右左双旋转调整后，以结点 m 为根结点的子树的高度降低了 1，因此继续回溯。

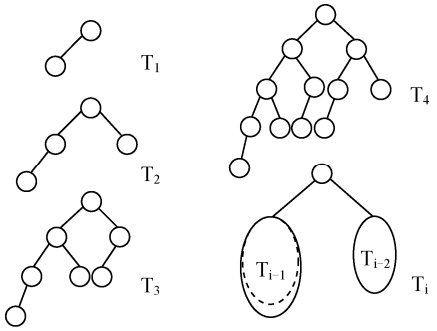


图 3-34　最接近于不平衡的 AVL 树

3．AVL 树的效率

AVL 树的检索、插入和删除效率都是 $O(\log_2 n)$，这是因为具有 n 个结点的 AVL 树的高度一定是 $O(\log n)$。要说明这一点，只需说明有 n 个结点的 AVL 树的最大高度不超过 $K\log_2 n$ 即可，这里 K 是一个小的常数。

为了证明，在此构造一系列临界 AVL 树 T_1，T_2，T_3，…。其中，T_i 的高度是 i，使每棵具有高度 i 的其他 AVL 树都比 T_i 的结点个数多。图 3-34 表示了 T_1，T_2，T_3 和 T_4。为了构造 T_i，先分别构造 T_{i-1} 和 T_{i-2}，T_i 即是以 T_{i-1} 和 T_{i-2} 为其左、右子树的 AVL 树（因为既要保证它们都是 AVL 树，又要要求它们包含最少的结点，所以必然高度相差 1）。对于每一个 i，因此所构造的是在结点数一定时最接近于不平衡的 AVL 树，这种树在结点一样多的 AVL 树中具有最大高度，可用此树来计算 AVL 树高度的上限值。

先计算上面所说的 AVL 树的结点个数，然后建立高度与结点个数的关系。令 t(i)为 T_i 的结点数，从图 3-34 可看出有下列关系成立：

$t(1) = 2$

$t(2) = 4$

$t(i) = t(i-1)+t(i-2)+1$

对于 i>2 此关系很类似于定义 Fibonacci 数关系：

$F(0) = 0$

$F(1) = 1$

$F(i) = F(i-1) + F(i-2)$

对于 i>l 仅检查序列的前几项就可有

$t(i) = F(i+3)-1$

Fibonacci 数满足渐近公式：

$F(i) = \Phi^i / \sqrt{5}$，这里 $\Phi = (1+\sqrt{5})/2$

由此可得近似公式：

$$t(i) \approx \frac{1}{\sqrt{5}}\Phi^{i+3} - 1$$

解出高度 i 与结点个数 t(i)的关系：

$$\Phi^{i+3} \approx \sqrt{5}(t(i)+1)$$

$$i+3 \approx \log_{\Phi}\sqrt{5} + \log_{\Phi}(t(i)+1)$$

由换底公式 $\log_{\Phi} X = \log_2 X / \log_2 \Phi$ 和 $\log_2 \Phi \approx 0.694$ 求出近似上限：

$$i < \frac{3}{2}\log_2(t(i)+1) - 1$$

因为 t(i)是结点个数，即对于一个有 n 个结点的 AVL 树，其高度不超过 $1.5\log_2(n+1)$，这就是所要证明的结果。

二叉搜索树，包括 AVL 树，适用于组织内存中小规模的目录。对于较大的、存放在外存储器上的文件，用二叉搜索树来组织索引就不太合适。若以结点作为内外存交换的单位，则检索中在找到需要的关键码之前平均要对外存进行 $\log_2 n$ 次访问，这是很费时的。在文件索引中大量使用每个结点包含多个关键码的 B 树，尤其是 B+树。

3.2.10　堆与优先队列

在现实应用中，经常会遇到频繁地在一组数据中查找最大值或者最小值的情况，解决这个问题，可以对这组数据进行排序，然后再从已排序序列中找到最大值或者最小值。这种方法虽然可行，但时间开销比较大。对于这种特殊的应用，堆——这种数据结构能够提供较高的效率。

1．堆的定义与实现

最大树（最小树）：每个结点的值都大于（小于）或等于其子结点（如果有的话）的值的树。

最大堆（最小堆）：最大（最小）的完全二叉树。

由最大（最小）堆的定义可以看出，根结点是该完全二叉树中关键码最大（最小）的结点，也就是堆中最大（最小）的元素。

下面以最大堆为例子来分析堆的插入、删除和构建算法。

（1）最大堆的插入

最大堆的插入算法描述如下：

① 将新结点插入到该树的最末尾的位置上。

② 用该结点与其父结点进行比较，如果该结点的关键码大于父结点的关键码，则将两个结点的位置交换。

③ 重复步骤②直到新结点的关键码不再大于父结点的关键码或者新结点成为根结点为止。

图 3-35 给出一个简单的插入结点的例子。当插入关键码为 21 的结点时，首先将该结点插入到完全二叉树的最末尾的位置，然后用该结点与其父结点（关键码为 2）进行比较，该结点的关键码大于父结点的关键码，所以两个结点需要互换位置，交换之后，再用该结点与新的父结点（关键码为 20）进行比较，该结点的关键码仍然大于父结点的关键码，所以需要继续互换位置。交换之后，该结点已经成为根结点，停止比较步骤，得到的结果即为插入关键码为 21 的结点后的最大堆。

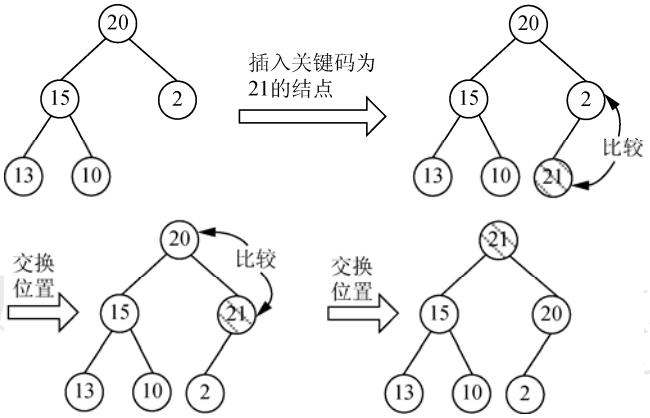


图 3-35　最大堆插入结点的实例

（2）最大堆的删除

最大堆的删除结点方法描述如下：

① 将完全二叉树最末尾结点 m 和待删除结点 p 交换位置，删除结点 p。

② 用结点 m 与其左右孩子结点中关键码较大的一个进行比较，如果该结点的关键码小于该孩子结点的关键码，则将两个结点的位置交换。

③ 重复步骤②直到结点 m 不再小于左右孩子结点中关键码较大的那个或者结点 m 为叶子结点为止。

图 3-36 给出一个最大堆删除结点的例子，当删除关键码为 21 的根结点时，首先将该完全二叉树中最末尾的结点 m 和待删除结点 p 交换位置，再删除结点 p。然后用结点 m 与其孩子结点中较大的一个（也就是关键码为 20 的那个结点）进行比较，结点 m 的关键码较小，所以两个结点要互换位置。再用结点 m 与新的孩子结点中关键码较大的一

个（也就是关键码为 13 的那个结点）进行比较，结点 m 的关键码仍然较小，所以需要继续互换位置。再次交换之后，该结点已经成为叶子，停止比较步骤，得到的结果即为删除根结点后的最大堆。

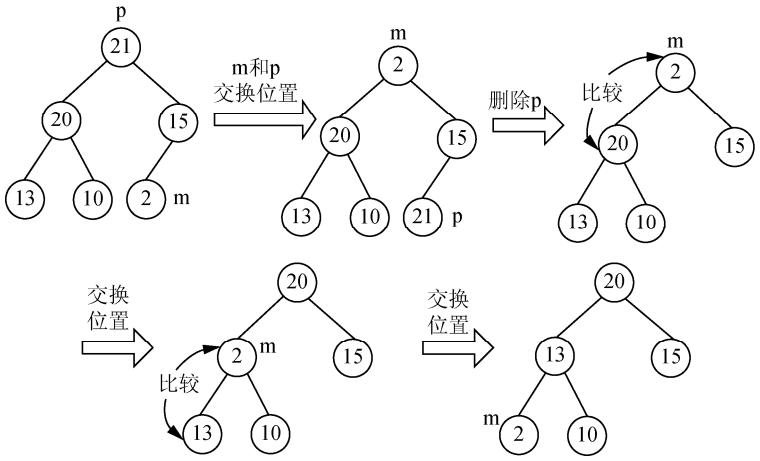


图 3-36　最大堆删除结点的实例

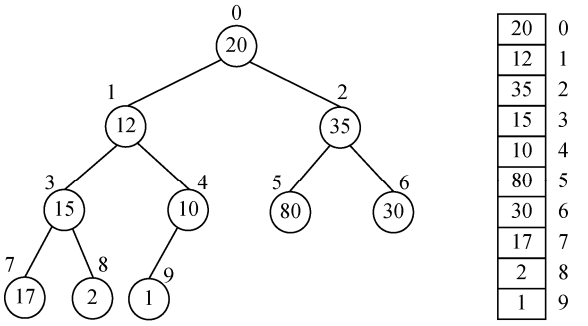
（3）最大堆的构建

可以通过最大堆的插入算法进行构建，即从空堆开始，依次插入各个关键码。该方法的时间复杂度为$O(n\log n)$。还有其他的构建方法，可以将时间复杂度降低到$O(n)$。下面详细介绍这种构建方法。

1964 年 Floyd 提出了一种称为筛选法的算法，该算法首先将待排序的所有关键码放到一棵完全二叉树的各个结点中，显然，该完全二叉树中以任一叶子结点为根的子树已经满足最大堆的性质，不需调整。从第一个具有孩子的结点 i 开始（$i=\lfloor n/2 \rfloor-1$，符号$\lfloor x \rfloor$代表小于等于 x 的最大整数），如果以这个元素为根的子树已是最大堆，则不需调整，否则需调整子树使之成为堆。继续检查 i−1，i−2 等结点为根的子树，直到该二叉树的根结点（其位置为 0）被检查并调整结束后，构建才结束。

由于最大堆是完全二叉树的一种特殊形式，因此参照完全二叉树的性质，可以将最大堆存储到数组中，用数组的下标来表示结点之间的关系。下面给出一个最大堆构建的例子。

假设用数组来存放具有 10 个关键码的最大堆，令 int a[10]={20,12,35,15,10,80,30,17,2,1}，建立最大堆的初始状态如下图 3-37 所示。图 3-37（a）中是将关键码依顺序添加而得到的完全二叉树，每个结点旁边的编号代表关键码在数组中对应的下标。图 3-37（b）中表示关键码在数组中的存放位置，右侧的数字代表下标。



（a）最大堆二叉树表示　　（b）数组表示

图 3-37　最大堆构建的初始状态

按照筛选法的步骤，算法从编号为$i=\lfloor n/2 \rfloor-1$的分支结点开始，按照编号递减的顺序逐个判断以该结点为根的子树是否满足最大堆的性质，若满足，则不做调整，否则需按照筛选法进行调整。具体步骤在图 3-38 中给出：

图 3-38（a）为调整的第一步，$i=\lfloor n/2 \rfloor-1=4$，因此第一个被判断的子树根结点就是编号为 4 的结点，对应的关键码是 10，其孩子结点的关键码是 1，满足最大堆的性质，因此不需调整。

图 3-38（b）中判断下一个子树根结点，即编号为 3 的结点，该子树不满足最大堆的性质，因此调整该子树使其满足最大堆的性质，也就是将子树中关键码较大的 17 与子树根结点 15 互换位置（同时在数组中做相应修改）。

图 3-38（c）中对编号为 2 的子树根结点进行调整，该子树不满足最大堆的性质，因此需要将关键码 80 与 35 互换位置，同时在数组中做相应修改。

图 3-38（d）中对编号为 1 的子树根结点进行调整。调整的过程中首先将关键码 12 与 17 互换位置（在数组中做相应修改），然后判断出经过以上调整后，编号为 1 结点的左子树的最大堆的性质被破坏，需要再次调整，也就是将编号为 3 和 7 的结点再次交换，同时在数组中做相应修改。

图 3-38（e）中对编号为 0 的结点，也就是根结点，进行判断调整。该调整与图（d）一样，需要多步调整，首先将编号为 0 和 2 的结点互换位置，然后再将编号为 2 和 5 的结点再次交换（数组中做相应修改）。

最终图 3-38（f）中得到的数组就是通过筛选法构建出来的最大堆。

上面已经介绍了最大堆的定义以及插入、删除和构建等操作的详细步骤，下面给出其部分代码。

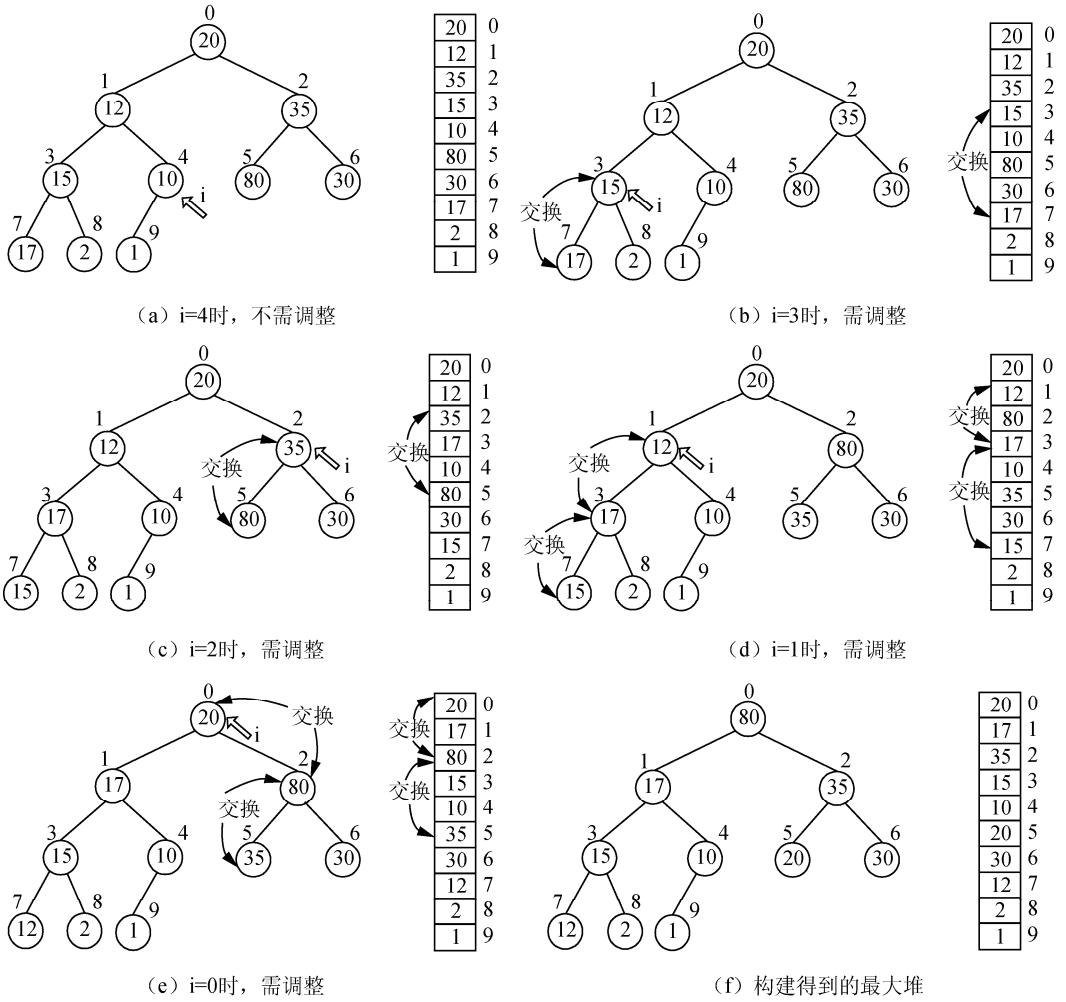


图 3-38　筛选法构建最大堆

【例 3.17】最大堆的定义。

```
template <class T>
class MaxHeap
{
private:
    T* heapArray;                    //存放堆数据的数组
    int CurrentSize;                 //当前堆中元素数目
    int MaxSize;                     //堆所能容纳的最大元素数目
public:
    MaxHeap(T* array,int num,int max);
    virtual ~MaxHeap(){};            //析构函数
    void BuildHeap();
```

```
    bool isLeaf(int pos) const;     //如果是叶结点，返回 TRUE
    int leftchild(int pos) const;   //返回左孩子位置
    int rightchild(int pos) const;  //返回右孩子位置
    int parent(int pos) const;      //返回父结点位置
    bool Remove(int pos, T& node);  //删除给定下标的元素
    void SiftDown(int left);//筛选法函数，参数 left 表示开始处理的数组下标
    void SiftUp(int position);  //从 position 向上开始调整，使序列成为堆
    bool Insert(const T& newNode);  //向堆中插入新元素 newNode
    void MoveMax();                 //从堆顶移动最大值到尾部
    T& RemoveMax();                 //从堆顶删除最大值
};
```

【例 3.18】构建最大堆的函数实现.

```
template<class T>
void MaxHeap<T>::BuildHeap()
{
    for (int i = CurrentSize/2-1; i >= 0; i--)
        SiftDown(i);
}

template<class T>
void MaxHeap<T>::SiftDown(int left)
{
                                    //准备
    int i = left;                   //标识父结点
    int j = 2*i + 1;                //标识左子结点
    T temp = heapArray[i];          //保存父结点的关键码
                                    //过筛
    while(j < CurrentSize)
    {
        if((j < CurrentSize-1)&&(heapArray[j] < heapArray[j+1]))
            j++;
//该结点有右孩子且右孩子的关键码大于左孩子的关键码时，j 指向右子结点
        if(temp<heapArray[j])
        {                       //该结点的关键码小于左右孩子中比较大的那个时
            heapArray[i]=heapArray[j];  //交换对应值
            i = j;
            j = 2*j + 1;                //向下继续判断是否满足最大堆的性质
        }
        else break;
    }
    heapArray[i] = temp;
}
```

从上面的函数定义可知，SiftDown( )函数中执行 while 循环的次数不超过子树的高度，对于由 n 个结点构成的堆，SiftDown( )函数的时间复杂度为 O(logn)。而建堆要不断

地向下调整，大约 n/2 次调用 SiftDown( )函数，因此建堆的时间复杂度的上界是 O(nlogn)。

对于 n 个结点的堆，其对应的完全二叉树的层数是 logn。设 i 表示二叉树的层编号，则第 i 层上的结点数最多为 2^i(i>=0)。建堆过程中，对每一个非叶子结点都调用了一次 SiftDown( )调整算法，而每个数据元素最多向下调整到最底层，即第 i 层上的结点向下调整到最底层的调整次数为 logn−i。因此，建堆的计算时间为

$$\sum_{i=0}^{\log n} 2^i(\log n - i)$$

令 j=logn−i，代入上式得

$$\sum_{i=0}^{\log n} 2^i(\log n - i) = \sum_{j=0}^{\log n} 2^{\log n - j} \cdot j = \sum_{j=0}^{\log n} n \cdot \frac{j}{2^j} < 2n$$

因此，建堆算法的时间复杂度是 O(n)，说明可以在线性时间内把一个无序的序列转化成堆序。

2．优先队列

优先队列是一种特殊的数据结构，其中能被访问和删除的是具有最高优先级的元素，所谓的优先级就是通过一些方法对元素进行比较得到的。其具体定义为：优先队列是这样的一种数据结构，对它的访问或者删除操作只能对集合中通过指定优先级方法得出的具有最高优先级的元素进行。

优先队列实际上可以用堆来实现，或者说堆就是一种优先队列。由于堆的插入元素和删除元素都会破坏堆结构，所以堆的插入和删除操作都要进行结构调整。一般的队列插入元素是在队列的最后进行，而优先队列却不一定插入到最后，是需要按照优先级算法将元素插入到队列中，出队时还是从第一个（优先级最高）的那个元素开始，删除的时候需要对被破坏了的结构进行调整，按优先级来选出优先级最高的那个元素放在第一个位置上。从上面优先队列的插入和删除操作来看，与数据结构——最大堆的操作一致，因此可以采用最大堆来实现优先队列。

优先队列的出队（即删除优先级最高的那个元素）操作，需要在删除结点后，剩下的元素仍需保持按优先级从大到小排列的要求，即剩下的 n-1 个结点值仍然符合最大堆的性质。因此优先队列出队的具体操作为首先用最大堆的最末尾结点的关键码替换根结点的关键码，然后删除最末尾结点，最后对最大堆进行调整。具体实现在例 3.19 中给出。

【例 3.19】用堆实现优先队列的出队操作。

```
template<T>
T& MinHeap<T>::RemoveMin()
{                                              //删除堆顶元素
    if(CurrentSize == 0)
    {
                                               //空堆情况
```

```
            cout << "Can't Delete";
            exit(1);
        }
        else
        {
            temp = heapArray[0];              //取堆顶元素
            heapArray[0] = heapArray[currentSize-1];//将堆尾元素上升至堆顶
            currentSize--;                    //堆中元素数量减 1
            if(CurrentSize > 1)               //堆中元素个数大于 1 时才需要调整
                                              //从堆顶开始筛选
                SiftDown(0);
            return temp;
        }
    }
```

3.2.11 Huffman 编码树

Huffman 树又称最优二叉树，是一类加权路径长度最短的二叉树，在编码设计、决策与算法设计等领域有着广泛应用。3.2.1 节中介绍了扩充二叉树以及外部路径、内部路径的概念，本节将介绍的数据结构与扩充二叉树以及外部路径有关。

1．建立 Huffman 编码树

给出 n 个实数 $W_0, W_1, \cdots, W_{n-1}(n \geqslant 2)$，要求得到一个具有 n 个外部结点的扩充二叉树，该扩充二叉树的每个外部结点 k_i 有一个 W_i 与之对应，作为结点 k_i 的权值，使得带权外部路径长度 $WPL = \sum_{i=0}^{n-1} W_i l_i$ 为最小，其中 l_i 是从根到外部结点 k_i 的路径长度。

例如下图 3-39 中有 3 棵二叉树，都有 4 个叶子结点 a、b、c、d，分别带权 7、5、2、4，求它们各自的带权路径长度 WPL。

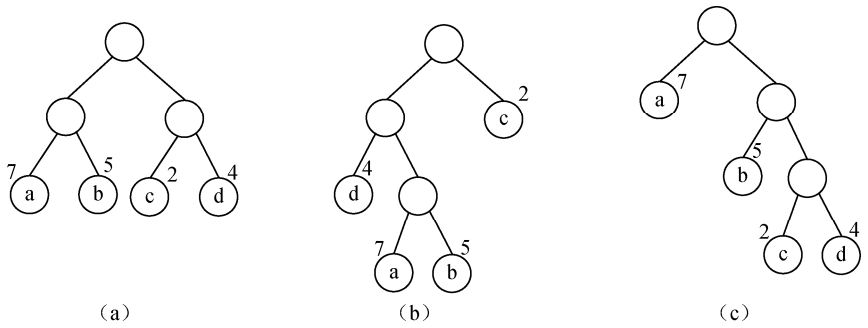


图 3-39　计算带权路径长度的实例

图 3-39（a）中，$WPL = 7\times2 + 5\times2 + 2\times2 + 4\times2 = 36$；

图 3-39（b）中，$WPL = 7\times3 + 5\times3 + 2\times1 + 4\times2 = 46$；

图 3-39（c）中， WPL＝7×1＋5×2＋2×3＋4×3＝35 。

从 a，b，c 三种情况计算出来的结果可以看出，一般情况下，权值越大的叶子离根越近，该二叉树的带权外部路径长度之和就越小。

解决上述问题而得到的扩充二叉树就称为 Huffman（哈夫曼）树，也称作最优二叉树。Huffman 树的定义如下：

假设有 n 个权值 $W_0, W_1, \cdots, W_{n-1}(n \geqslant 2)$，构造一棵有 n 个叶子结点的二叉树，每个叶子结点 k_i 有一个权值 W_i 与之对应，则构造出的所有二叉树中使得带权外部路径长度 WPL 最小的那个二叉树就称作 Huffman 树。

Huffman 树的构建方法如下：

1）根据给定的 n 个权值 $W_0, W_1, \cdots, W_{n-1}(n \geqslant 2)$ 构成 n 棵二叉树的集合 $F = \{T_1, T_2, \ldots T_n\}$，其中每棵二叉树 T_i 中只有一个权值为 W_i 的根结点。

2）在 F 集合中选取两棵根结点权值最小的树作为左、右子树来构造一棵新的二叉树，且置新二叉树的根结点的权值为其左、右子树根结点的权值之和。

3）在 F 集合中删除这两棵树，同时将新得到的二叉树加入集合 F 中。

4）重复 2）和 3），直到 F 集合中只含一棵树为止。

以图 3-39 中的那 4 个权值组成的权值集合{7,5,4,2}来构建 Huffman 树的过程如下图 3-40 所示。

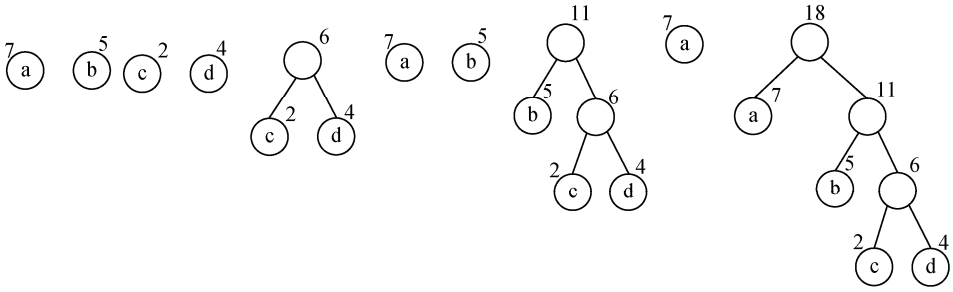


图 3-40　构建哈夫曼树的实例

【例 3.20】Huffman 树的类定义。

```
template <class T>
class HuffmanTree
{
    private:
        HuffmanTreeNode<T>* root;      //Huffman 树的树根
        void MergeTree(HuffmanTreeNode<T> &ht1, HuffmanTreeNode<T>
&ht2,
        HuffmanTreeNode<T>* parent);
        //把 ht1 和 ht2 为根的 Huffman 子树合并成一棵以 parent 为根的二叉树
        void DeleteTree(HuffmanTreeNode<T>* root);
                                          //删除 Huffman 树或其子树
```

```
    public:
        HuffmanTree(T weight[],int n);
                    //构造 Huffman 树，weight 是权值的数组，n 是数组长度
        virtual ~HuffmanTree(){DeleteTree(root);}; //析构函数
}

template<class T>
HuffmanTree<T>::HuffmanTree(T weight[], int n)
{
    MinHeap<HuffmanTreeNode<T>> heap(n);        //定义最小值堆
    HuffmanTreeNode<T>* parent,&firstchild,&secondchild;
    HuffmanTreeNode<T>* NodeList=new HuffmanTreeNode<T>[n];
    for(int i=0;i<n;i++)
    {
        NodeList[i].element=weight[i];
        NodeList[i].parent=NodeList[i].left=NodeList[i].right=NULL;
        heap.Insert(NodeList[i]);           //向堆中添加元素
    }
    for(i=0;i<n-1;i++)          //通过 n-1 次合并建立 Huffman 树
    {
        parent=new HuffmanTreeNode<T>;
        firstchild=heap.RemoveMin();        //选择权值最小的结点
        secondchild=heap.RemoveMin();       //选择权值次小的结点
        MergeTree(firstchild,secondchild,parent);
                                            //合并权值最小的两棵树
        heap.Insert(*parent);               //把 parent 插入到堆中去
        root=parent;                        //建立根结点
    }
    delete []NodeList;
}
```

2．Huffman 编码及其应用

Huffman 树常见的应用在数据通信和数据压缩领域。经 Huffman 编码的信息消除了冗余数据，有效地提高了通信信道的传输效率。

设某电文由 A、B、C、D 四个字符组成，如果把这些字符编码成二进制的 0，1 序列，该如何进行编码呢？一种较为简单的策略是采用定长编码方案，将 A、B、C、D 四种字符编码为 00、01、10、11。设传送的电文为 ABACCD，则原电文转换为 00 01 00 10 10 11。对方接收后，采用二位一分进行译码。

虽然定长编码方案的编码和译码都很方便，但当每个字符出现的概率不等时，这种编码方案极有可能造成冗余。在传送电文时，总是希望编码总长越短越好，如果对每个字符设计长度不等的编码，且让电文中出现次数较多的字符采用较短的编码，则可以减短电文编码的总长。

例如，对 ABACCD 重新编码，A、B、C、D 分别编码为 0、00、1、01。则原电文

转换后的电文编码为 0 00 0 1 1 01。电文编码的总长减短了。那么对转换后的二进制编码该如何译码呢？前四个字符（0000）就有多种译法，可以译成 AAAA，BB 等。

为了避免编码、译码时出现歧义，在变长编码方案中必须保证任一编码都不是另一个编码的前缀，这样的编码称为前缀编码。前缀编码可以利用 Huffman 树来实现。

设 $D=\{d_0,d_1,\cdots,d_{n-1}\}$ $W=\{W_0,W_1,\cdots,W_{n-1}\}$，D 为需要编码的字符集合，W 为 D 中各字符出现的频率，要对 D 里的字符进行前缀编码，使得通信编码长度最短，利用 Huffman 树可以这样做：用 $D=\{d_0,d_1,\cdots,d_{n-1}\}$ 作为外部结点，$W=\{W_0,W_1,\cdots,W_{n-1}\}$ 作为外部结点的权值，构造具有最小带权外部路径长度的 Huffman 树。把每个结点指向其左孩子的边标为 0，指向其右孩子的边标为 1，从根结点到某个叶子结点的路径上的标号连接起来就得到该叶子结点所代表的字符的前缀编码。

例如，某通信可能出现 A B C D E F G H 这 8 个字符，其概率分别为 0.05 , 0.29 , 0.07 , 0.08 , 0.14 , 0.23 , 0.03 , 0.11 ，图 3-41 给出利用 Huffman 算法构造出的编码树及各字符的二进制编码。

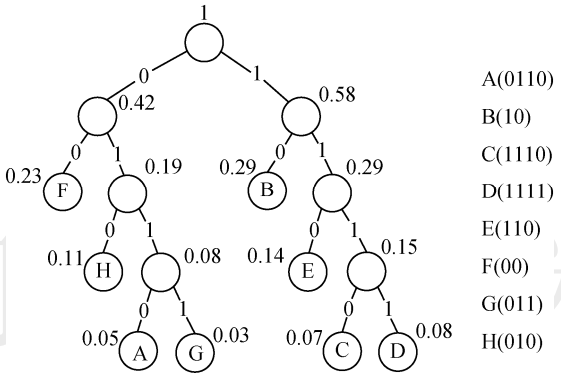


图 3-41　Huffman 编码实例

用 Huffman 算法构造出的扩充二叉树不仅给出了各字符的编码，同时也用来译码，译码过程如下：

1）从根结点出发，从左至右扫描编码。

2）若为‘0’则走左分支，若为‘1’则走右分支，直至叶子结点为止。

3）取叶子结点字符为译码结果。

4）返回重复执行 1，2，3 直至全部译完为止。

3.3　树 与 森 林

3.3.1　二叉树、树、森林之间的转换

由于树形结构的特殊性，这使得二叉树与树、森林之间有一个一一对应的关系。它

们之间可以相互转换，也就是任何一个森林或者一棵树，可以唯一地对应为一棵二叉树，而任意一棵二叉树，也可以唯一地对应为一个森林或者一棵树。因此，对树或者森林的处理，都可以转换为与其对应的二叉树的处理。由于这个特性，二叉树在树的应用中占很重要的地位。

根据二叉树的定义知道其是有序树，而一般的树或者森林都是无序的，为了使二叉树与树、森林之间的转换能够进行，下面的章节中将树和森林看作有序的，每个结点的次序由该结点在图中出现的位置决定。

1．树、森林转换为二叉树

将一棵树转换为二叉树的方法是：

1）树中所有相邻兄弟结点之间加一条连线。

2）对于树中的每个结点，只保留它与第一个孩子结点之间的连线，删去它与其他孩子结点之间的连线。

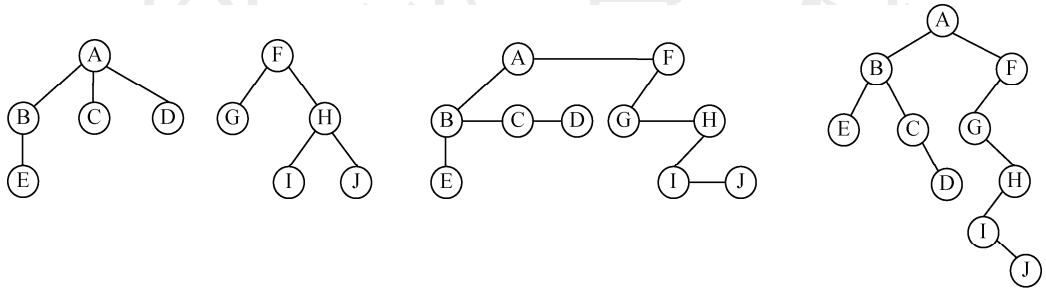
3）以树的根结点为轴心，将整棵树顺时针转动一定的角度，使之结构层次分明。

将一个森林转换为二叉树的方法是：

1）将森林中的每棵树转换成相应的二叉树。

2）第一棵二叉树不动，从第二棵二叉树开始，依次把后一棵二叉树的根结点作为前一棵二叉树根结点的右孩子，当所有二叉树连在一起后，所得到的二叉树就是由森林转换得到的二叉树。

图 3-42 给出了一个森林转换为二叉树的例子。



（a）森林　　（b）转换后的二叉树　　（c）倾斜后的二叉树

图 3-42　森林转换为二叉树

按照上面的转换方法，二叉树里的某个结点的左子结点是它在原来树形结构里的最左边的子结点，它的右子结点是它在原来树形结构里的右边相邻的兄弟结点。

下面给出递归构造方法：

若 $T=\{T_1,T_2,\cdots,T_m\}$ 是 m（$m\geqslant 0$）棵树的序列，则与 T 相对应的二叉树 $\beta(T)$ 的构造方法如下：

1）如果 $m=0$，则 $\beta(T)$ 为空二叉树。

2）如果 $m>0$，则以 T_1 的根结点作为 $\beta(T)$ 的根结点，以 $\beta(T_{1,1},T_{1,2},\cdots,T_{1,r})$ 作为 $\beta(T)$ 的左子树，其中 $T_{1,1},T_{1,2},\cdots,T_{1,r}$ 是 T_1 的子树，以 $\beta(T_2,T_3,\cdots,T_m)$ 作为 $\beta(T)$ 的右子树。

根据上述的构造方法，如果 T 是有序树的序列，那么根据其构造出来的二叉树 $\beta(T)$ 是唯一的。

2．二叉树还原为森林、树

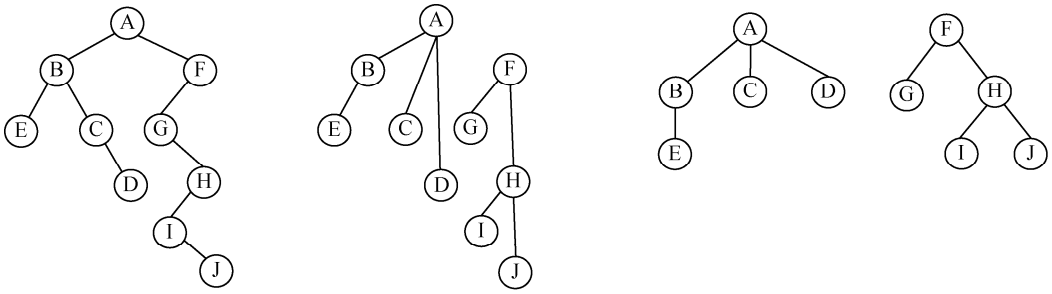
将一棵由森林或者一般树转换得到的二叉树还原为一般的森林或者树的过程如下：

1）若某结点是其双亲的左孩子，则把该结点的右孩子、右孩子的右孩子、……都与该结点的双亲结点用线连起来。

2）删掉原二叉树中所有双亲结点与右孩子结点的连线。

3）整理由（1）、（2）两步所得到的树或森林，使之结构层次分明。

图 3-43 是图 3-42 中得到的二叉树的还原过程：



（a）二叉树　　（b）转换后的森林　　（c）整理后的森林

图 3-43　二叉树的还原过程

下面给出递归还原的方法：

若 $\beta(T)$ 是一棵二叉树，把 $\beta(T)$ 还原为对应的由 m（$m\geqslant 0$）棵树序列组成的森林 $T=\{T_1,T_2,\cdots,T_m\}$ 的步骤如下：

1）如果 $\beta(T)$ 是一棵空二叉树，则 T 为空。

2）如果 $\beta(T)$ 为非空二叉树，则以 $\beta(T)$ 的根结点为森林 T 中的第一棵树 T_1 的根结点，以 $\beta(T)$ 的左子树还原成的森林作为 T_1 中根结点的子树序列 $\{T_{1,1},T_{1,2},\cdots,T_{1,r}\}$，以 $\beta(T)$ 的右子树还原成的森林作为森林 T 中除 T_1 以外的其余树的序列 $\{T_2,T_3,\cdots,T_m\}$。

3.3.2　树和森林的遍历

前面的章节中介绍了二叉树的层次优先遍历和深度优先遍历，树和森林的遍历也是按照某种次序访问树和森林中的每个结点，每个结点被且仅被访问一次。由于树中每个结点的孩子数量没有规定，树和森林的遍历方法同二叉树的遍历方法的种类稍微有些不同，树和森林的深度优先遍历只有先根次序和后根次序两种。

1．树的遍历

（1）深度优先遍历

参照二叉树的先序遍历和后序遍历法，可以定义树的先根次序和后根次序。

1）先根次序：

① 访问根结点。

② 从左到右，依次先根遍历根结点的每一棵子树。

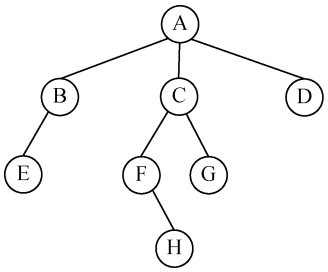


图 3-44　树

例如图 3-44 中的树，按照先根次序遍历的结果为 ABECFHGD，这与将该树先转换为对应的二叉树，再对该二叉树进行先序遍历得到的结果一致。

2）后根次序：

① 从左到右，依次后根遍历根结点的每一棵子树。

② 访问根结点。

例如图 3-44 中的那棵树，按照后根次序遍历的结果为 EBHFGCDA，这与将该树先转换为对应的二叉树，再对该二叉树进行中序遍历得到的结果一致。

（2）广度优先遍历

该遍历方法与二叉树中所讲述的层次优先遍历方法一致。图 3-44 中的树的广度优先遍历结果为 ABCDEFGH。

2．森林的遍历

森林的遍历与树的遍历方法一致。

（1）深度优先遍历

参照树的先根次序遍历和后根次序遍历法，可以定义森林的先根次序和后根次序。

1）先根次序：

若森林非空，则遍历方法为：

① 访问森林中第一棵树的根结点。

② 先根次序周游第一棵树的根结点的子树森林。

③ 先根次序周游其他的树。

例如图 3-45 中的森林，按照先根次序遍历的结果为 ABCDEFGHIJ，这与将该森林先转换为对应的二叉树，再对该二叉树进行先序遍历得到的结果一致。

2）后根次序

若森林非空，则遍历方法为：

① 后根次序周游森林中第一棵树的根结点的子树森林。

② 访问第一棵树的根结点。

③ 后根次序周游其他的树。

例如图 3-45 中的森林，按照后根次序遍历的结果为 BCDAFEHJIG，这与将该森林先转换为对应的二叉树，再对该二叉树进行中序遍历得到的结果一致。

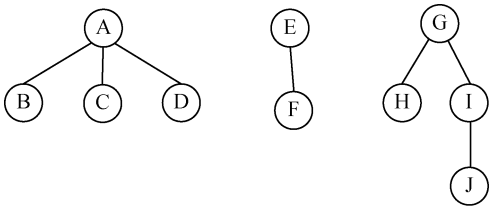


图 3-45　森林

（2）广度优先遍历

该遍历方法就与二叉树中所讲述的层次优先遍历方法一致。图 3-45 中的森林的广度优先遍历结果为 ABCDEFGH。

3.3.3　树的存储

本小节将详细给出树的存储表示方法，并对每种表示方法在空间代价和操作性能上进行优劣分析。树的存储结构根据应用的不同，可以对应有不同的存储方法，下面介绍几种常用的方法。

1．*孩子表示法*

由于树中每个结点可能有多个孩子，因此很自然就可以想到用多重链表来进行存储。在这种多重链表中，用指针将每个结点与孩子结点连接起来。每个结点除了有存放数据信息的 data 域以外，还需要有若干指针来指向其孩子结点。但是不同的结点所拥有的孩子结点的数量不同，因此每个结点中需要的指针域的个数也不同，那么每个结点中到底给出几个指针域呢？为此给出如下两种方案。

（1）定长结点的多重链表

这种方案规定以树的度数作为每个结点指针域的数目，每个结点中的指针域的个数相同。该方法中结点的形式在图 3-46 中给出。不难看出，一棵具有 n 个结点的度数为 k 的树中共有 n*k 个指针域，但是其中有用的只有 n－1 个，因此大部分的指针域空置，造成存储空间的极大浪费。例如图 3-47（a）中所示的树，其定长结点的多重链表表示法如图 3-47（b）所示。

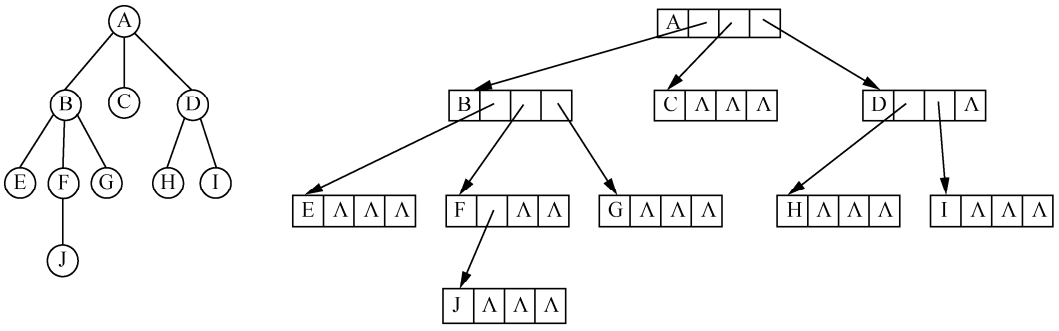
Data	$child_1$	…	$child_n$

图 3-46　定长结点的多重链表中结点的形式

（2）不定长结点的多重链表

上面介绍的定长结点的多重链表由于设定每个结点中的指针域的个数相同，致使存

储空间严重浪费，在这里采取新的方法来减少空间的浪费。该方法规定每个结点中的指针域的个数与该结点的度数一致。为了在操作的时候可以知道每个结点的指针域的个数，需要在结点中设置一个度数域（degree）来指出该结点的度数，具体的结点的形式在图 3-48 中给出。在图 3-49 中则为图 3-47（a）中的树给出其不定长结点的多重链表表示法的对应表示，可以看出这种方法的存储密度较前者有所提高，但由于各结点的结构不同，自然会造成操作上的不方便。



（a）树　　（b）定长结点多重链表表示法

图 3-47　定长结点多重链表表示法示例

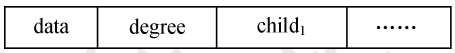


图 3-48　不定长结点的多重链表中的结点形式

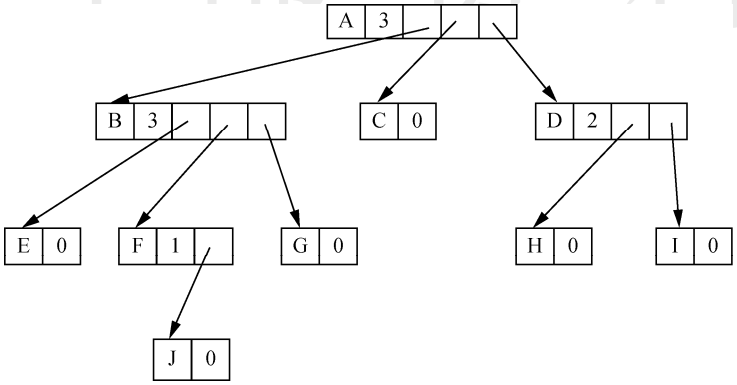


图 3-49　不定长结点的多重链表表示法示例

2．孩子-兄弟表示法

上面介绍的孩子表示法，其中定长结点的多重链表方法的存储空间浪费较严重，而不定长结点的多重链表方法由于结点的结构不同，使得操作复杂，为了解决这两个问题，

使得每个结点中的指针域的数量相同，并且减少空间的浪费量，这里提出一个新的存储方法：孩子-兄弟表示法。这种方法又称二叉树表示法（或二叉链表表示法）。在这种二叉链表的每个结点中除了用于存放数据信息的 data 域外，还有两个指针域 firstChild 和 nextSibling，分别用于指向该结点的第一个孩子结点和它的下一个兄弟结点。图 3-50 给出结点的形式，图 3-51 为图 3-47（a）所示的树的孩子-兄弟链表。在这种存储结构中，树的操作比较方便，且存储密度较高。

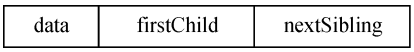


图 3-50　孩子-兄弟表示法中的结点形式

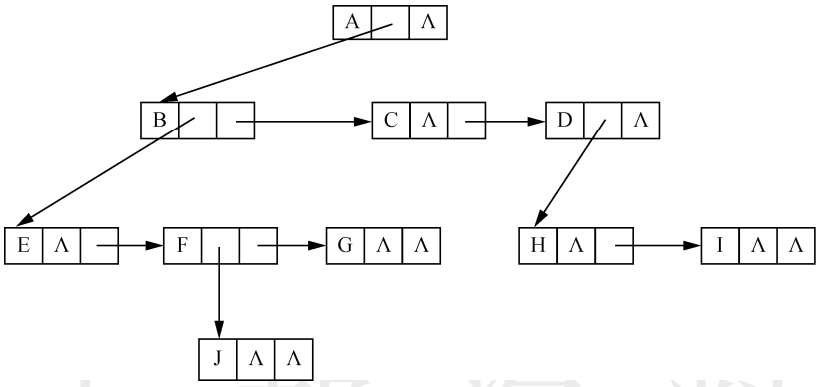


图 3-51　孩子-兄弟表示法示例

3．双亲表示法

在这种方法中，用一组连续的存储单元存储树中的结点，结点的形式如图 3-52 所示。

其中，data 域用于存放有关结点本身的信息，parent 域用于指示该结点的双亲位置。例如，图 3-47（a）所示的树，其双亲表示法如图 3-53 所示。

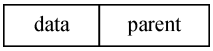


图 3-52　双亲表示法中的结点形式

结点索引	0	1	2	3	4	5	6	7	8	9
数据	A	B	C	D	E	F	G	H	I	J
父结点索引	/	0	0	0	1	1	1	3	3	5

图 3-53　双亲表示法示例

这种存储结构利用了每个结点（除根以外）只有唯一双亲的性质。在这种存储结构下，求结点的双亲十分方便，也很容易求树的根。但是在这种表示法中，在求某个结点的孩子结点时需要遍历整个存储空间。

3.4 树 的 应 用

3.4.1 二叉树：图像压缩算法

1．需求描述

计算机中的图像大致可以分成两类：位图（Bitmap）和矢量图（Metafile）。位图可以视为一个二维的网格。每个点被称为一个像素点，每个像素点有确定的颜色，当很多个像 素点按行和列排列形成了一幅完整的图像。

通常使用的图像大部分都是位图，如数码相机拍摄的照片，都是位图。因为位图可以完美的表示图像细节。但位图也有缺点如文件体积比较大，所以人们开发了很多压缩图像格式来储存位图图像，目前应用最广的是 JPEG 格式。

在图像处理中，经常需要将真彩色图像转换为灰度图像。一个真彩色像素点转换为灰度图时它的亮度值使用如下的公式：

$$Y = 0.299R+0.587G+0.114B$$

这个公式通常都被称为心理学灰度公式。这里面看到绿色分量所占比重最大。因为科学家发现使用上述公式进行转换时所得到的灰度图最接近人眼对灰度图的感觉。

因为灰度图中颜色数量一共只有 256 种（1 个字节），即 256 种亮度（深浅）不同的灰色。因此，转换后的图像常保存为 8 位格式而不是 24 位格式，这样比较节省空间。

而且，灰度图中某种特定颜色出现的频度不是随机的，即经常某一灰度出现在连续区域，在整体上灰度分布也不是均匀的，即有些灰度占图像的绝大部分，而另一些灰度没有出现。故使用压缩编码的方式可以减小存储图像的大小。

假定已经编写了一个预处理程序将一个真彩色图像处理转换成为一个灰度图像，图像的数据保存在 data.dat 中。其中每行都是等长的，行的长度表示图像宽度，行的数量表示图的高度。从图像的最左上角开始，每个像素用一个字节表示，该字节即表示该像素的灰度（0~255）。

需要编写一个应用可以处理 data.dat 以压缩其中的数据，并将编码表和压缩后的编码一并存储到新文件中，并能根据压缩后的文件将原始输入文件解压出来。

2．问题分析

本应用是典型的 Huffman 编码的使用情形，针对灰度编码与针对文本编码并没有本质区别，因此可以使用书上的样例代码，并在此基础上继续工作。

本应用的难点在于如何进行压缩后文件的存储。为此，提出这样一种文件格式来存储压缩文件。文件中的最开始 4 个字节即 0x00-0x03 存储 SCNH 作为文件类型标志，0x04-0x07 存储图像宽度，0x08-0x0B 存储图像高度，0x0C 存储颜色数，0x0D-0x0F 保留，从 0x10 开始存储颜色编码表，每个颜色表的格式如下。

偏移	+0x0	+0x2	+0x4
内容	灰度值	编码长度	编码

其中编码长度指的+0x4 上编码二进制位的个数，另外每个颜色表都是字节对齐的。

接下来考虑压缩编码的存储，变长前缀码可以采用定长存储法或者变长存储法，其中定长存储可以降低读取和写入文件的复杂度，但是文件压缩率不高；而变长存储方式能够体现压缩算法的高压缩率，但是在解压文件时操作较为繁琐，本例中选用较为复杂的变长存储方案。

在颜色表之后第一个字节对齐处开始从图片首行开始存储压缩后的图片编码，行与行之间不需要间隔符，又因为霍夫曼编码都是前缀编码，编码之间也不需要间隔标志。在合理编制编码的情况下，这种方式可以有效降低文件大小。

3．概要设计

根据上面的分析可以得出：在本应用中需要压缩器 Compressor 类，代表颜色的 Color 类，代表编码的 HCode 类及用来实际存储 HCode 的 String 类，霍夫曼树 HuffmanTree 类及其结点类 HuffmanNode，还有在构建 Huffman 树时用到的 MinHeap 类，最后还用到 STL 中的 map 类和 fstream 类。该应用的类图如图 3-54 所示。

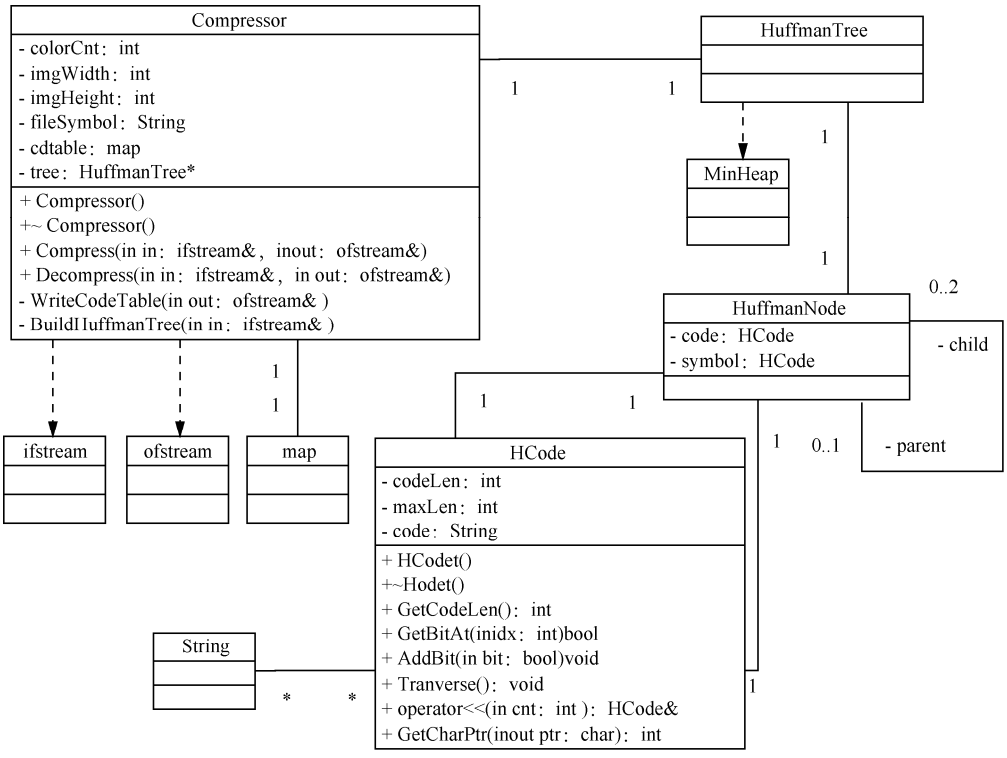


图 3-54　类图

Compressor 类是一个压缩解压的主要类别，其中 Compress 成员函数用来将源文本压缩为目标文件，Decompress 成员函数用来将目标文件解压为源文件。BuildHuffmanTree 成员函数用来统计源文件中的各颜色的出现的次数并根据给定的信息构建霍夫曼树。

HCode 类是对 String 类的一个封装，使其更好的存储编码，String 中的每个元素用来存储 8 位编码，GetCodeLen 成员函数用来取得当前的编码长度，GetBitAt 成员函数取得特定位上的编码，AddBit 函数用来将编码最低位增加一位，Tranverse 用来将编码逆置以便于存储和解码，operator<<操作用来将编码左移，移入的位用 0 填充。

4．详细设计

首先是 Compressor::BuildHuffmanTree 函数，图 3-55 给出了其主要逻辑。

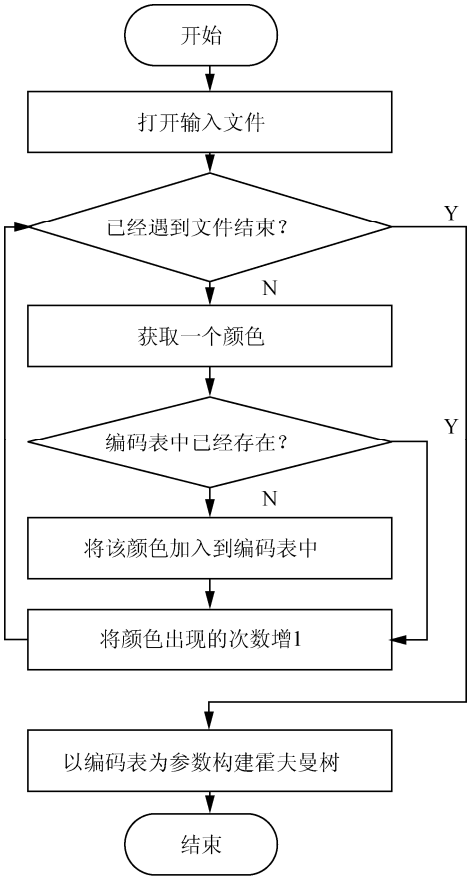


图 3-55　BuildHuffmanTree 函数逻辑图

在霍夫曼树构建完成后，就可以使用 Compressor::Compress 函数对文件进行压缩了，图 3-56 给出其主要流程。

最后本程序还可以根据压缩后的文件将原始文件还原，图 3-57 介绍了 Decompress

函数的主要流程。

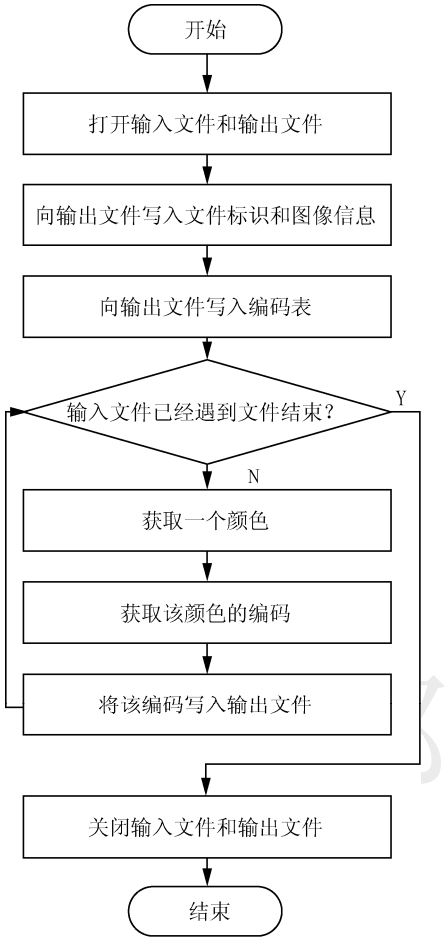


图 3-56　Compress 函数流程图

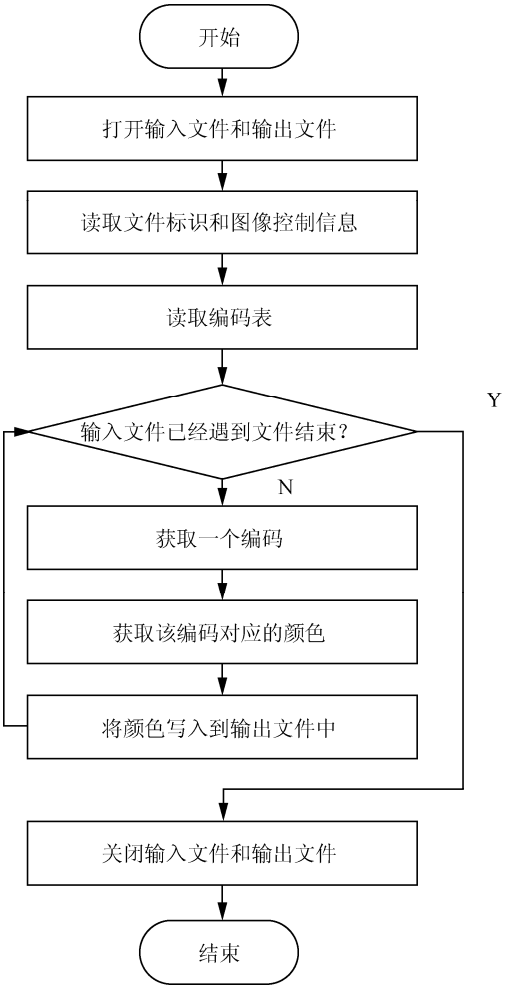


图 3-57　Decompress 函数流程图

3.4.2　树：医院设施管理

1．需求描述

一个医院中包含很多的科室部门，在信息时代，如何使用计算机详细的描述一家医院具有很大的意义。假设医院的结构如图 3-58 所示，从图中可以看出医院是一棵以“医院”为根结点的树，通过前面学过的二叉树和树的知识来模拟医院的结构，对医院有一个详尽的描述。并且实现以下的功能：

1）给定医院的任意两个结点，比如，“楼层”和“病房”，计算并输出每个“楼层”

中包含的“病房”的数量；

2）给定任意结点的名称，查找并输出该结点及其子结点；

3）浏览整个医院的结构。

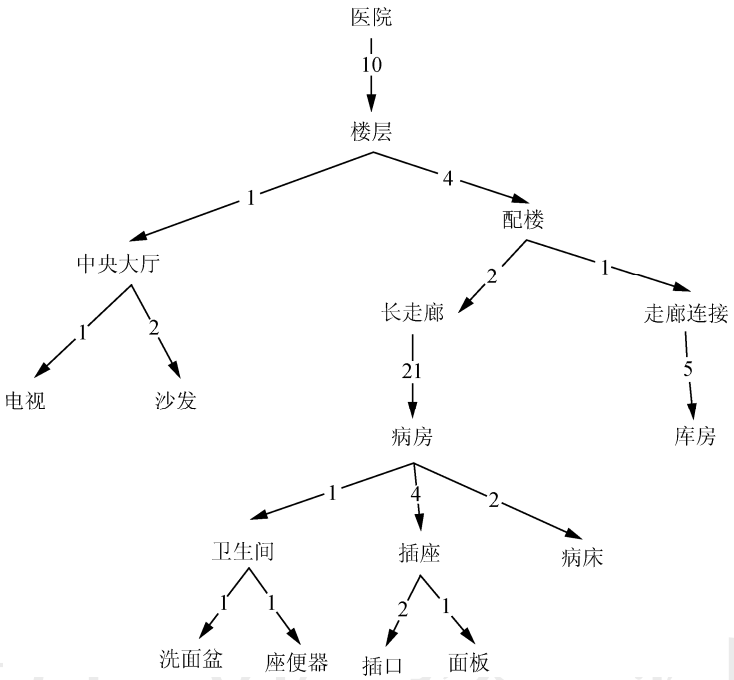


图 3-58　医院结构图

2．问题分析

医院是一个树形的结构，不同于二叉树，普通的树会有不同数量的子结点，因此简单的给出左子结点和右子结点是不能遍历到整棵树的。但是，树和二叉树之间是可以相互转化的，将一棵树转化为二叉树的过程是首先将兄弟结点用连线连接起来，然后保留父结点与第一个子结点的连线，将父结点到其他子结点的连线切掉，最后以根为轴旋转。这个转化过程和树的一种存储方法“左子结点/右兄弟”二叉链表表示法类似。在“左子结点/右兄弟”二叉链表表示法中，对于每个结点需要保存左子结点和其右兄弟，这里的左子结点就相当于树到二叉树转化过程中的与父结点保持连线的第一个子结点，即转化成的二叉树中的左子结点，右兄弟就是转化成的二叉树中的右子结点。对于二叉树的遍历和检索等操作前面已经讲过，因此，选用“左子结点/右兄弟”二叉链表表示法存储树的结构，该方法存储的树结构经过旋转后实际上是一棵二叉树。

计算结点 A 中包含的结点 B 的数量时，采用自底向上的方法，从 B 结点开始依次向上计算父结点，直到出现 A 结点为止，A 包含 B 的数量则为 B 到 A 的每步链接中结点的数量乘积。

查找一个结点时，采用广度遍历的方法，在树的每一层上依次比较，直到找到为止。

浏览整个医院的结构可以采用广度优先遍历方法，也可以是先根深度优先遍历方法和后根优先遍历方法。

3．概要设计

根据前面的描述，医院中的每个结点都应该包含两个属性，一个是该结点的名称，另一个是该结点的数量。因此需要一个 HosNode 类，其中包含一个 String 类型的名称和一个 int 类型的数量值。另外，对于一棵树，需要树的结点 TreeNode 类和树 Tree 类。

TreeNode 类中有数据域、指针域，数据域是医院的结点 HosNode 类型，指针域包含最左边的子结点的指针和其右兄弟的指针。TreeNode 中除了有一些 get、set 方法外，还有 InsertFirst 和 InsertNext 方法，分别代表以第一个左孩子身份插入结点和以其右兄弟的身份插入结点。print 方法用于打印当前结点及其子结点的信息。

在 Tree 中有一个 TreeNode 类型的根结点，构建一棵树时需要一个 insert 方法，计算结点 A 包含结点 B 的数量的方法为 count，该方法包含两个参数，分别为结点 A 和 B 的名称，计算数量时，首先要在树中找到这两个结点，然后从 B 结点开始依次向上找父节点因此还需要一个 Find 和 Parent 方法。另外，该类中还有一些遍历方法，包括广度优先遍历方法、先根深度优先遍历方法和后根深度优先遍历方法。

本应用中各个类之间的关系如图 3-59 所示。

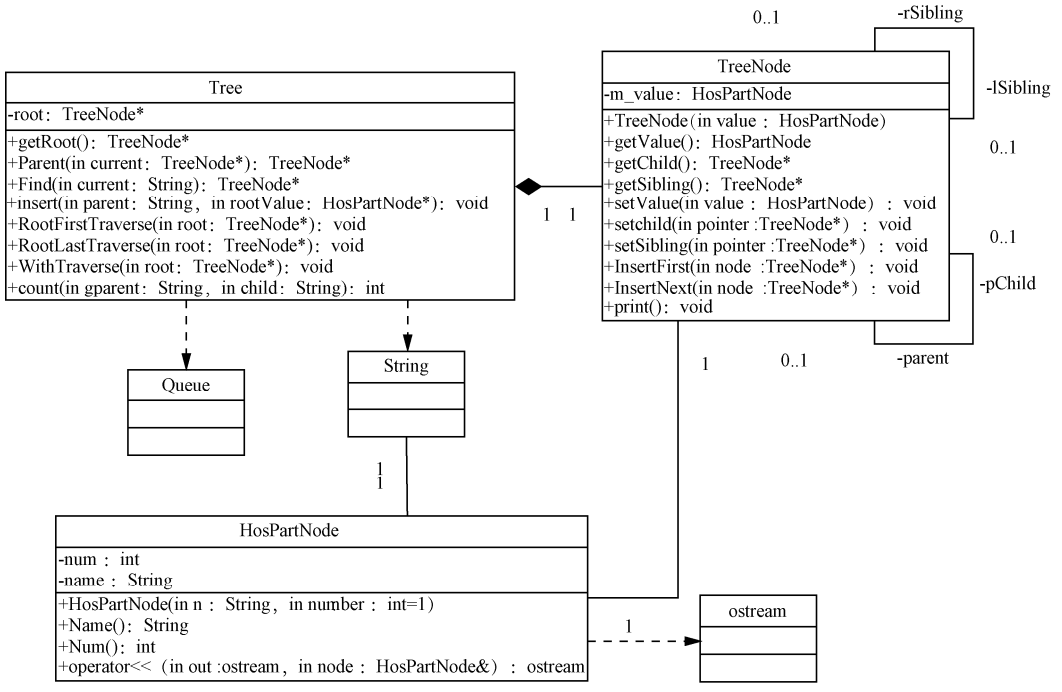


图 3-59　类图

4．详细设计

下面对应用中的一些重要方法进行详细的介绍。

（1）void Tree::insert(string parent, HosNode* value)

在树中，通过 insert 方法逐渐构造一棵树，该方法包含两个参数：待插入结点的父结点的名称和待插入结点值。具体的，插入时，若不存在根结点，则直接使用 parent 构造根结点，将包含 value 值的结点作为根结点的左子结点。否则，首先找到名称为 parent 结点，然后将包含 value 值的结点作为其左子结点。

（2）int Tree::count(string gparent, string child)

该方法的功能为计算出 gparent 中包含的 child 的数量。首先使用 Find 方法找到 gparent 和 child 两个结点，若有一个结点为空，则返回 0。否则，从 child 开始依次向上计算父结点，若按这种方法能找到 gparent 结点，则输出从 child 结点到 gparent 结点路径上的每个结点的数量的乘积，若没有找到 gparent 结点，这说明 gparent 结点不包含 child 结点，此时返回 0。该方法的具体流程如图 3-60 所示。Num 为返回值。

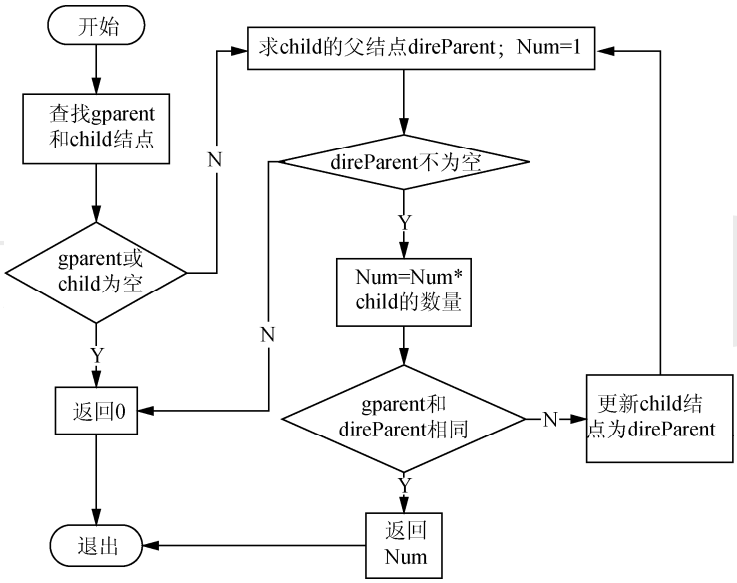


图 3-60　计算给定结点的子结点数量流程图

（3）TreeNode * Tree::Parent(TreeNode * current)

因为 Find 方法和 Parent 方法过程类似，因此这里只介绍 Parent 方法。Parent 方法的功能为返回 current 结点的父结点。从根结点开始逐层的遍历结点。若 current 结点为空，则返回 NULL，否则，将所有的根结点入队列。若队列不为空，则取出队列的首元素，然后将首元素的所有子结点依次与 current 比较，若相同则返回首元素，否则全部进队列。循环对队列进行取首元素和进队列的操作。若直到队列为空仍没有找到 current 结点，则返回 NULL。该方法的详细过程如图 3-61 所示。

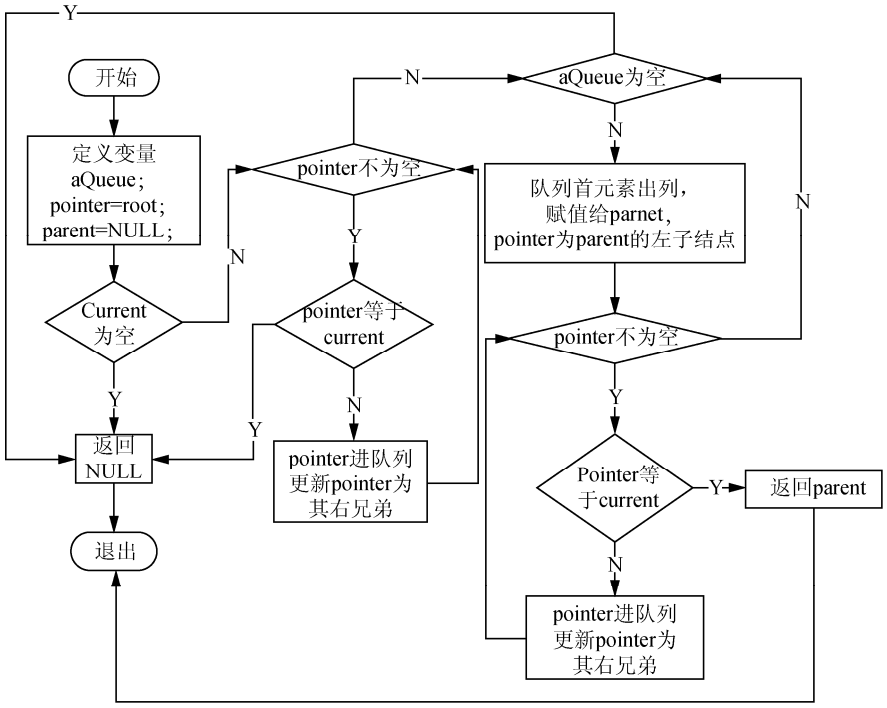


图 3-61　寻找给定结点的父结点流程图

习　题

1．n 个结点可构造出多少种不同形态的二叉树？若有 3 个数据 1,2,3，输入它们构造出来的中序遍历结果都为 1,2,3 的不同二叉树有哪些？

2．具有 33 个结点的完全二叉树的深度是多少？有多少个叶节点？有多少个度为 1 的结点？

3．某二叉树有 20 个叶节点，有 30 个结点仅有一个孩子，求该二叉树的总结点数是多少？

4．试分别找出满足以下条件的所有二叉树。

（1）二叉树的前序序列与中序序列相同。

（2）二叉树的中序序列与后序序列相同。

（3）二叉树的前序序列与后序序列相同。

5．设一棵二叉树以二叉链表表示，试编写有关二叉树的递归算法：

（1）统计二叉树中度为 1 的结点个数；

（2）统计二叉树中度为 2 的结点个数；

（3）统计二叉树中度为 0（叶节点）的结点个数；

（4）统计二叉树的高度；

（5）统计二叉树的宽度，即在二叉树的各层上具有结点数最多的那一层上结点总数；

（6）计算二叉树中各结点中的最大元素的值；

（7）交换每个结点的左孩子结点和右孩子结点；

（8）从二叉树中删去所有叶节点。

6．编写算法判别给定二叉树是否为完全二叉树。

7．在中序线索二叉树中如何查找给定结点的前序后继？如何查找给定结点的后序后继？

8．对于后序线索二叉树进行遍历是否需要栈的支持？为什么？

9．已知一棵二叉树的前序遍历序列为 ABECDFGHIJ，中序遍历序列为 EBCD AFHIGJ。

（1）试画出这棵二叉树并写出它的后序遍历序列；

（2）试画出这棵二叉树的中序线索二叉树。

10．已知序列（50，72，43，85，75，20，35，45，65，30），请以顺序插入方式构造二叉搜索树，并画出删除结点 72 之后的二叉搜索树。

11．对于一个高度为 h 的 AVL 树，其最少结点数是多少？反之，对于一个有 n 个结点的 AVL 树，其最大高度是多少？最小高度是多少?

12．若关键字的输入序列为 20,9,2 ,11,13,30,22,16,17,15,18,10。

（1）试从空树开始顺序输入各关键字建立平衡二叉树。画出每次插入时二叉树的形态，若需要平衡化旋转则做旋转并注明旋转的类型；

（2）计算该平衡二叉搜索树在等概率下的查找成功的平均查找长度；

（3）基于上面建树的结果，画出从树中删除 22，删除 2，删除 10 与 9 后树的形态和旋转类型。

13．假定一组记录的关键码为（46，79，56，38，40，84，50，42），利用筛选法构建最大堆（以树状表示）。

14．写出向最小堆中加入数据 4, 2, 5, 8, 3, 6, 10, 14 时，每加入一个数据后堆的变化。

15．假定用于通信的电文仅由 8 个字母 A,B,C,D,E,F,G,H 组成，各字母在电文中出现的频率分别为 5,25,3,6,10,11,36,4。试为这 8 个字母设计不等长 Huffman 编码，并给出该电文的总码数。

16．在结点个数为 n (n>1)的各棵树中，高度最小的树的高度是多少？它有多少个叶结点？多少个分支结点？高度最大的树的高度是多少？它有多少个叶结点？多少个分支结点？

17．对图 3-62 所示树结构分别进行先根遍历和后根遍历。

18．试写出下列森林（如图 3-63 所示）的先根序列、后根序列和层次序列。

19．画出图 3-64 中的二叉树所对应的森林。

20．已知如下森林（如图 3-65）画出对应的二叉树。

21．试用三种表示法画出图 3-66 所示的树的存储结构。

（1）孩子表示法。

（2）孩子—兄弟表示法。

（3）双亲表示法；

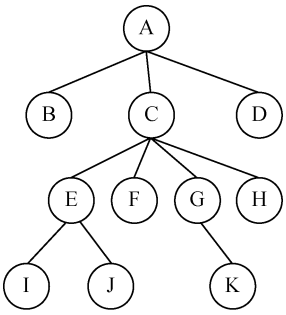


图 3-62　树结构

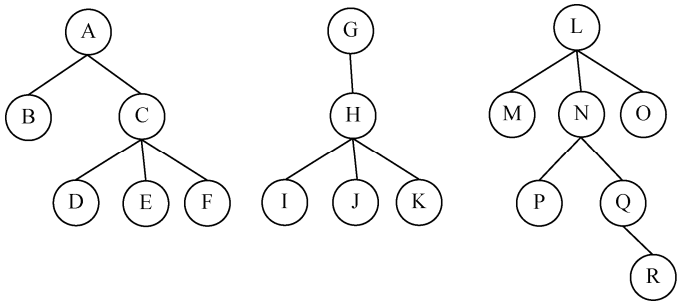


图 3-63　森林树

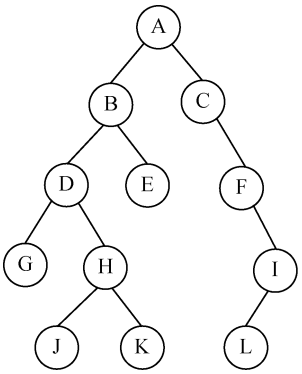


图 3-64　二叉树

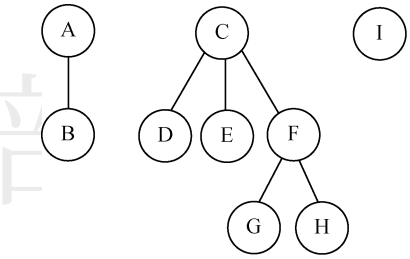


图 3-65　森林树

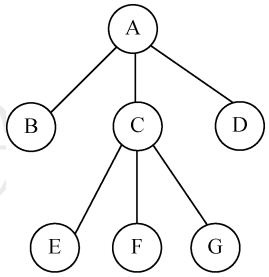


图 3-66　树