第5章 树形结构

树形结构是一种十分重要的非线性结构，可描述数据元素间一对多的逻辑关系，其结点之间形成分支和层次关系，类似于自然界中的树。在客观世界中，有许多事物本身就呈现树形结构，如家族关系、部门机构设置等，用树来表示就非常简便，也非常形象和自然。另外，有些算法，也常常要借助树形结构来解决。

本章介绍树形结构中树、森林、二叉树等的基本内容[①]并列举一些简单应用，重点是二叉树。在后面几章中，还会涉及到树和二叉树的一些其他应用。

5.1 树的概念

在现实世界中，有很多问题可用树形结构来描述。如一个零部件的组成就可以很自然地表示成一个树形结构。图 5.1 表示的就是某鼠标器的组成：它由电路和机械两大部分组成；前者由接口、按键检测、滚轮检测和主电路组成；后者由滚轮、按键、壳体等组成。这类图形看上去就像一棵倒画的树，“树形结构”即由此得名。

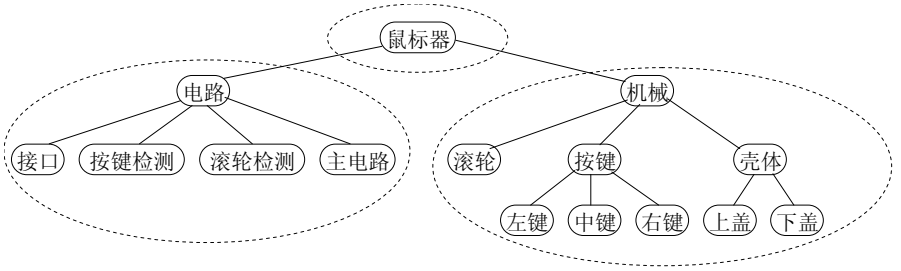


图 5.1　鼠标器的组成示意图

图 5.1 可分解为三个部分：鼠标器本身、电路组成、机械组成，如图中虚线所示。而电路组成和机械组成还可进行类似的分解。分解出的小组成部分仍然保持着树形结构，直到每部分只有一个结点为止。这一特点是一切树形结构都具备的。从所有类似的问题可抽象出树的递归定义。

① 树形结构中的一些术语或名称在不同文献中可能有较大差异，如同一个概念可能名称不同，而同一个名称也可能含义不同，需要注意。

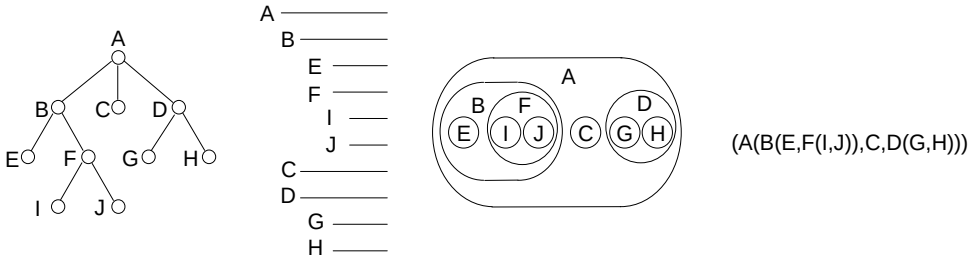
树（Tree）是n（n≥0[①]）个结点的有限集合T，它或者为空（n=0），或者满足以下条件：

（1）有且仅有一个特定的称为**根**（Root）的结点。

（2）其余结点分为 m（m≥0）个互不相交的子集 $T_1, T_2, \cdots, T_m$，其中每个子集又是一棵树，称其为根的**子树**（Subtree）。

从该定义知，只有一个结点的集合是一棵树，该结点就是根，它无子树。如果集合中元素个数大于 1，则它至少含有一棵子树。

在树的树形图表示中，结点一般用圆圈表示，结点的名字写在圆圈旁边，有时也写在圆圈内。除了树形表示法外，在不同的应用场合，树还可有其他表示法。如图 5.2（a）所示的树还可用图 5.2（b）、（c）和（d）来表示，其中（b）用的是凹入表示法，类似于书的目录，适合文本模式下树的屏幕显示和打印输出；（c）采用的是嵌套集合表示法，利用集合的包含关系来描述，树根所在的集合最大；（d）采用的是广义表表示法。



（a）树形表示法　（b）凹入表示法　（c）嵌套集合表示法　（d）广义表表示法

图 5.2　树形结构的表示示意图

另外，在日常工作中表示事物的分类和组成时，也常将图 5.2（a）的树形表示法转动 90°使用，例如图 5.3 对 C/C++语言数据类型的分类表示。

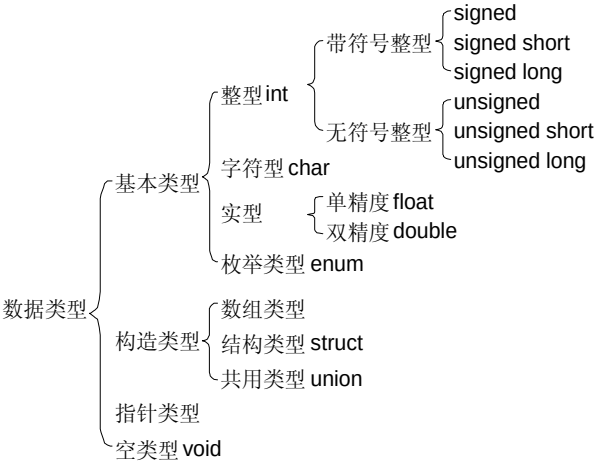


图 5.3　C/C++语言数据类型

① 有的文献只允许 n>0，即没有空树的概念。

一个结点的子树个数称为该结点的**度**（Degree）。一棵树的度是指该树中结点的最大度数。度为 k 的树也称为 k 叉树，它的每个结点最多有 k 个子树。度为零的结点称为**叶子**（Leaf）、叶结点或终端结点。度不为零的结点称为分支结点或非终端结点，除根结点之外的分支结点统称为**内部结点**，根结点也称为**开始结点**。

树中某个结点的子树之根称为该结点的**孩子**（Child）或儿子，相应地，该结点称为孩子的**双亲**（Parents）或父亲。同双亲的孩子互称为**兄弟**（Brother 或 Sibling）。双亲是兄弟关系的结点称为**堂兄弟**。

若树中存在一个结点序列$\{k_1, k_2, \cdots, k_j\}$，使得 k_i 是 k_{i+1} 的双亲，则称这个结点序列为从 k_1 到 k_j 的一条**路径**（Path）或道路，或称从 k_1 到 k_j 有路径。路径中边（即连接两个结点的线段）的个数称为路径长度。路径中的结点序列“自上而下”地通过路径上的各边。若树中结点 k 到 k_s 有路径，则称 k 是 k_s 的**祖先**（Ancestor），k_s 是 k 的**子孙**（Descendant）。

结点的**层数**（Level）是从根开始算起的，根为第 1 层[①]，其他结点的层数为其双亲的层数加 1。树中结点的最大层数，称为树的**高度**（Height）或**深度**（Depth）。

若树中每个结点的各子树从左到右是有次序的（即位置不能互换，否则互换后认为是不同的树），则称该树为**有序树**（Ordered Tree）；否则称为**无序树**（Unordered Tree）。

若两棵树中，各结点对应相等，对应结点的相关关系也对应相等，则称这两棵树**相等**（等价）；若两棵树中，适当地重命名其中一棵中的结点，可以使两者相等，则称这两棵树**同构**。

森林（Forest）是 m（m≥0）棵互不相交的树的集合。显然，删去一棵树的根，就得到一个森林；反之，加上一个结点作树根，森林就变为一棵树。

树形结构的逻辑特征可用结点之间的父子关系来描述：树中任一结点都可有零个或多个直接后继（即孩子），但最多只能有一个直接前趋（即双亲）。树中只有根无前趋（故称开始结点），叶子无后继（故称终端结点）。显然，父子关系是非线性的。祖先与子孙的关系是父子关系的延伸。

树的基本运算有以下 6 种。

（1）初始化 INITIATE(T)：置 T 为空树。

（2）求双亲 PARENT(T, x)：求树 T 中结点 x 的双亲结点。若结点 x 是树 T 的根结点或结点 x 不在树 T 中，则函数值为“空”。

（3）求孩子 CHILD(T, x, i)：求树 T 中结点 x 的第 i 个孩子结点。若结点 x 是树 T 的叶子或无第 i 个孩子或结点 x 不在树 T 中，则函数值为“空”。

（4）插枝 INSERT(T, x, i, Y)：将以结点 Y 为根的树置为树 T 中结点 x 的第 i 棵子树。若原树 T 中无结点 x 或结点 x 的子树个数<i−1，则空操作。

（5）剪枝 DELETE(T, x, i)：删除树 T 中结点 x 的第 i 棵子树。若树 T 中无结点 x 或结点 x 的子树个数少于 i，则空操作。

（6）遍历 TRAVERSE(T)：按某种次序对树中每个结点访问一次且仅访问一次。

在实际应用中，可根据需要对这些基本运算进行适当增减，如增加求根、求右兄弟、

① 有的文献层数从 0 开始计算，这时深度和/或高度仍指最大层数，或最大层数+1。

建树、查找和删除结点等运算。

5.2 二叉树

二叉树是树形结构的一个重要类型，它的存储结构和算法都比较简便，特别适合于计算机处理。即使一般形式的树也可简单地转换为二叉树，再进行相应的处理。所以二叉树显得特别重要。

5.2.1 二叉树的概念

二叉树（Binary Tree）是 n（n≥0）个结点的有限集，它或者为空（n=0），或者由一个根结点及两棵互不相交的、分别称作该根的左子树和右子树的二叉树组成。这是个递归定义。由于二叉树本身及子二叉树都可为空，故二叉树有 5 种形态，如图 5.4 所示。

（a）空　（b）只含根　（c）右子树为空　（d）左子树为空　（e）左右子树均非空

图 5.4　二叉树的 5 种基本形态

在二叉树中，每个结点最多只能有两棵子树，并且有左右之分，显然它不是无序树。但它与度为 2 的有序树也不同。因为有序树的孩子之间虽然有左右之分，但若只有一个孩子，则不分左右；而二叉树即使只有一个孩子，也要严格区分左右。可见，二叉树既不是树的特殊情形，也不是有序树的特殊情形。

例如，图 5.5 是三棵不同的二叉树，它们与图 5.6 的普通树（有序或无序）相似，但不等同。如果将这 4 棵树都看成普通树，则它们就相同了。

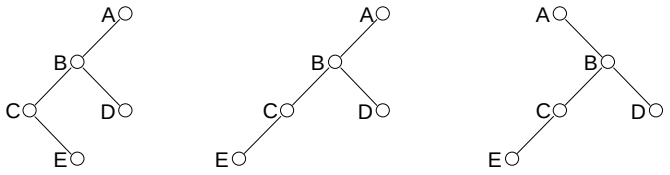


图 5.5　三棵不同的二叉树

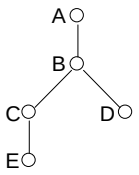


图 5.6　一棵普通树

和树类似，二叉树的基本运算有以下 6 种。

（1）初始化 INITIATE(BT)：加工型运算，建立一棵空二叉树 BT。

（2）求双亲 PARENT(BT, x)：引用型运算，求二叉树 BT 中结点 x 的双亲。若结点 x 是二叉树 BT 的根或二叉树 BT 中无此结点，则函数值为“空”。

（3）求左孩子 LCHILD(BT, x)及右孩子 RCHILD(BT, x)：引用型运算，分别求二叉树 BT 中结点 x 的左孩子及右孩子。若 x 为叶子或 x 不在二叉树 BT 中，则函数值为“空”。

（4）插枝 INSERTLEFT(BT, x, Y)及 INSERTRIGHT(BT, x, Y)：加工型运算，分别将以结点 Y 为根的二叉树置为二叉树 BT 中结点 x 的左子树和右子树。若原树 BT 中无结点 x 或结点 x 已有相应的子树，则空操作。

（5）剪枝 DELLEFT(BT, x)及 DELRIGHT(BT, x)：加工型运算，分别删除二叉树 BT 中结点 x 的左子树和右子树。若 x 无左子树或右子树，或 x 不在 BT 中，则为空操作。

（6）遍历 TRAVERSE(BT)：引用型运算，按某种次序访问二叉树中各个结点，使每个结点只被访问一次。

这些基本运算在实际应用中也可根据需要适当增减，如增加求根、求左右兄弟、建树、查找和删除结点等运算。

5.2.2　二叉树的性质

性质 1　二叉树第 i 层上的结点数最多为 2^{i-1}（i≥1）。

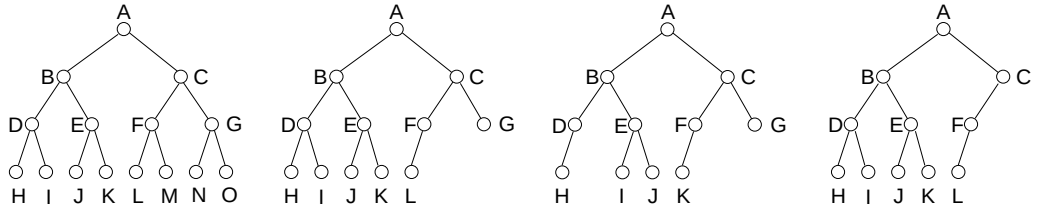
证：使用归纳法。i=1 时，结论显然成立。设 i=k−1 时结论亦成立，即第 k−1 层上的结点数最多为 2^{k-2}，则考虑 i=k 时的情形。由于二叉树每个结点最多有两个孩子，故第 k 层上的结点数最多是第 k−1 层上结点数的 2 倍，即最多 $2\times2^{k-2}=2^{k-1}$ 个，故命题成立。

性质 2　深度为 k 的二叉树至多有 2^k-1 个结点（k≥1）。

证：在深度相同的二叉树中，仅当每一层的结点数都达到最多时，树中结点总数才最多，于是由性质 1 可知，深度为 k 的二叉树的结点数最多为 $2^0+2^1+\cdots+2^{k-1}=2^k-1$。证毕。

如果深度为k的二叉树有 2^k-1 个结点，则称此二叉树为**满二叉树**①（Full Binary Tree）。它的特点是每一层上的结点数都达到了最大，即对给定的高度，它是具有最多结点数的二叉树。满二叉树中不存在度为 1 的结点，每个分支结点均有两棵高度相同的子树，且所有叶子都在最下一层上，见图 5.7（a）。

若一棵二叉树至多只有最下面两层上的结点的度可以小于 2，并且最下层上的结点都集中在该层最左边的若干位置上，则此二叉树称为**完全二叉树**②（Complete Binary Tree）。它相当于在满二叉树的最底层，从右向左连续去掉若干个结点后得到的二叉树，见图 5.7（b）。显然，在完全二叉树中，若一个结点没有左孩子，则它一定没有右孩子，即必定是叶子。



（a）满二叉树　（b）完全二叉树　（c）非完全二叉树　（d）非完全二叉树

图 5.7　完全二叉树和非完全二叉树示例

① 有的文献称此为完美二叉树（Perfect Binary Tree），而满二叉树指本书所称的严格二叉树；也有的文献称此为 Complete Binary Tree，而对应本书完全二叉树的是 Nearly Complete Binary。

② 有的文献称此为顺序二叉树：与同高度的满二叉树前 n 个结点按层次顺序一一对应；也有的文献完全二叉树指本书所称的严格二叉树。

满二叉树是完全二叉树，但完全二叉树不一定是满二叉树。空二叉树和只含一个结点的二叉树既是满二叉树，也是完全二叉树。

性质 3 二叉树中，度为 0 的结点数 n_0 和度为 2 的结点数 n_2 满足 $n_0=n_2+1$。

证：因为二叉树中所有结点的度只可能是 0、1 或 2，所以结点总数 n 应等于 0 度结点数 n_0、1 度结点数 n_1 和 2 度结点数 n_2 之和：

$$n=n_0+n_1+n_2$$

另一方面，度为 1 的结点有一个孩子，度为 2 的结点有两个孩子，故二叉树中孩子结点数为 n_1+2n_2，但根不是任何结点的孩子，故二叉树中的结点总数又可表示为孩子结点数与根之和：

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

结合上面两式，即可导出 $n_0=n_2+1$。

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 $\lceil \log_2(n+1) \rceil$。

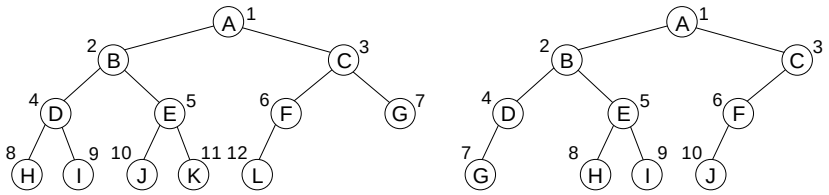
证：设 k 为所求完全二叉树的深度。由完全二叉树定义知道，它的前 k−1 层是深度为 k−1 的满二叉树，结点数为 $2^{k-1}-1$。但第 k 层上还有若干个结点，则总结点数 $n>2^{k-1}-1$。另由性质 2 知道 $n\leqslant 2^k-1$，所以：

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

由该式得 $2^{k-1}<n+1\leqslant 2^k$，取对数后有 $k-1<\log_2(n+1)\leqslant k$，即 $\log_2(n+1)\leqslant k<\log_2(n+1)+1$，由于 k 为整数，即得 $k=\lceil \log_2(n+1) \rceil$。

另外，注意到 n 为整数，由上式还可得 $2^{k-1}\leqslant n<2^k$，取对数后有 $k-1\leqslant \log_2 n<k$，即 $\log_2 n<k\leqslant \log_2 n+1$，由于 k 为整数，又可得 $k=\lfloor \log_2 n \rfloor+1$。证毕。

对二叉树的所有结点，我们可按一定规则对其编号，其中很自然的方法是按层次顺序进行，即**层序编号**。具体说，就是从根结点开始，从上层到下层，每一层从左到右，依次给所有结点标记 1, 2, …, n，其中根的编号为 1，n 为结点总数，见图 5.8 所示。



（a）完全二叉树　　（b）非完全二叉树

图 5.8　二叉树的层序编号

性质 5 在完全二叉树的层序编号中，对任一编号为 i 的结点 x（$1\leqslant i\leqslant n$），有：

（1）若 i>1，则 x 的双亲编号为 $\lfloor i/2 \rfloor$；若 i=1，则 x 是根，无双亲。

（2）若 2i>n，则 x 无左孩子，否则其左孩子的编号为 2i。完全二叉树中的结点若无左孩子则肯定也无右孩子，即为叶子，故编号 $i>\lfloor n/2 \rfloor$ 的结点必定是叶子。

（3）若 2i+1>n，则 x 无右孩子，否则其右孩子的编号为 2i+1。

（4）若 i 为奇数且不为 1，则 x 的左兄弟编号为 i−1；否则无左兄弟。

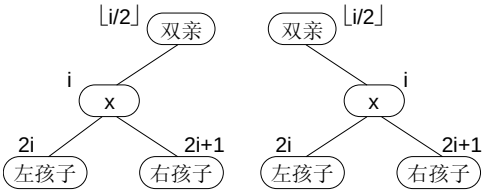
（5）若 i 为偶数且小于 n，则 x 的右兄弟编号为 i+1；否则无右兄弟。

这一性质可用数学归纳法证明（略）。

性质 5 实际上指出了这样一个重要事实：完全二叉树中结点之间的父子关系可由它们层序编号间的关系来表达，如图 5.9 所示。

注意，该性质是对完全二叉树而言的，对非完全二叉树则不成立，如图 5.8（b）中，结点 D 的编号为 4，它的左孩子 G 的编号是 7 而不是 2×4=8。

如果结点编号从 0 开始，上述结果需要略作修改，如编号为 i（0≤i≤n−1）的结点，其双亲为⌊(i−1)/2⌋、左孩子为 2i+1 等。若无特别声明，本书中结点编号一般从 1 开始。



（a）i 为偶数　　（b）i 为奇数

图 5.9　完全二叉树层序编号与父子关系

5.2.3　二叉树的存储

二叉树的存储结构应能体现二叉树的逻辑关系，即能反映结点的双亲和孩子关系。二叉树通常有两类存储结构：顺序存储结构和链式存储结构。

1．二叉树的顺序存储

二叉树的顺序存储就是将所有结点存储到一片连续的存储单元中，并能通过结点间的物理位置关系反映逻辑关系。这实际上就是要将二叉树的所有结点按一定次序排成一个线性序列，并且序列中结点间的次序关系要能反映结点间的逻辑关系。

由性质 5 知，完全二叉树结点的层序编号可反映结点间的逻辑关系，即由结点编号可推算出它的双亲和孩子的编号。所以，完全二叉树的结点可按层序编号的次序存储到一个向量 bt[1..maxsize]中，另外再指出结点数 n。由于 C/C++语言数组的下标从 0 开始，编号为 i 的结点将存放在下标为 i−1 的单元内，即结点编号和数组下标总要进行转换，不方便。为此，我们将数组定义为 bt[maxsize+1]，并从数组下标 1 开始使用。这就是完全二叉树的顺序存储结构，其 C/C++语言描述如下：

```
const int maxsize=100;      //结点数的最大值，假设为 100
typedef struct {
  datatype bt[maxsize+1];   //0 号单元未用
  int n;
} sqbitree;
```

其中数组 bt 的 0 号单元不用，但可用来存放其他信息，如结点数（这时数据域 n 可省略）。由于结点的编号就是数组的下标，所以通过下标间的数值关系就可知道结点之间的父子关系，即不需附加任何信息就可表示逻辑关系。例如，图 5.10 是图 5.8（a）完全二叉树的顺序存储结构示意图，其中 bt[5]的双亲是 bt[2]，其左、右孩子分别是 bt[10]和 bt[11]；bt[9]的双亲是 bt[4]，它没有孩子，因为 i=9，n=12，这时 2i>n。

显然，对完全二叉树而言，顺序存储结构既简单又节省存储空间。

然而，对一般的二叉树，结点间的层序编号关系并不能反映逻辑关系，所以不能直接

采用上述方法。但是，如果在二叉树上补充一些“虚结点”使其成为完全二叉树，则可使用上述方法了。由于要存储“虚结点”，会有一定的空间浪费，在最坏的情况下，一个深度为 k 且只有 k 个结点的右单支树却需要 2^k-1 个结点的存储空间。例如，只有 3 个结点 A、B 和 C 的右单支树，将其添上一些实际上并不存在的“虚结点”后，使它成为如图 5.11(a)所示的完全二叉树，相应的顺序存储结构见图 5.11(b)。图中虚线和“∧”表示虚结点。

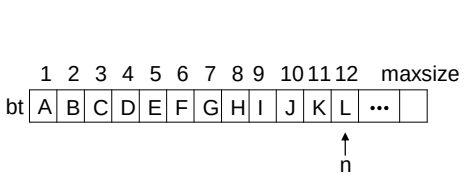


图 5.10 图 5.8（a）完全二叉树的顺序存储

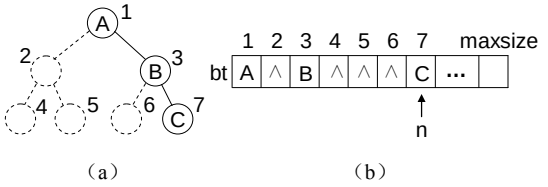


图 5.11 非完全二叉树的顺序存储

2．二叉树的链式存储

从上面看到：用顺序方式存储一般的二叉树将浪费存储空间；另外，这种存储方式下在树中插入和删除结点也不方便。因此，树的最自然的存储方法是采用链式存储。二叉树链式存储的基本思想是，每个结点除了存放本身的数据外，还要根据需要设置指向双亲和左、右孩子的指针，即通过指针来反映逻辑关系。这样，根据指针的设置情况，存储方式可分为一指针式、二指针式和三指针式。具体选用何种方式，主要考虑运算实现的方便程度以及运算的频度。

与普通链式存储一样，二叉树的链式存储可以是“静态”的，也可以是“动态”的。常用的是动态链式存储，这时一般不用担心空间溢出问题，也不必关心存储管理的细节（由系统完成），这使我们能用更多的精力去考虑其他问题。在本章内我们以动态链表为主，也在适当地方介绍一下静态方法。

二叉链表是二叉树最常用的存储结构，其中每个结点除了存储结点本身的数据外，还设置两个指针域 lchild 和 rchild，分别指向该结点的左孩子和右孩子。这是二叉树链式存储的二指针形式。就像单链表由头指针唯一确定一样，一个二叉链表也由指向根结点的根指针唯一确定。为了某些运算的方便，也可给二叉链表增加头结点（但一般并没有这样做）。二叉链表的结点结构为：

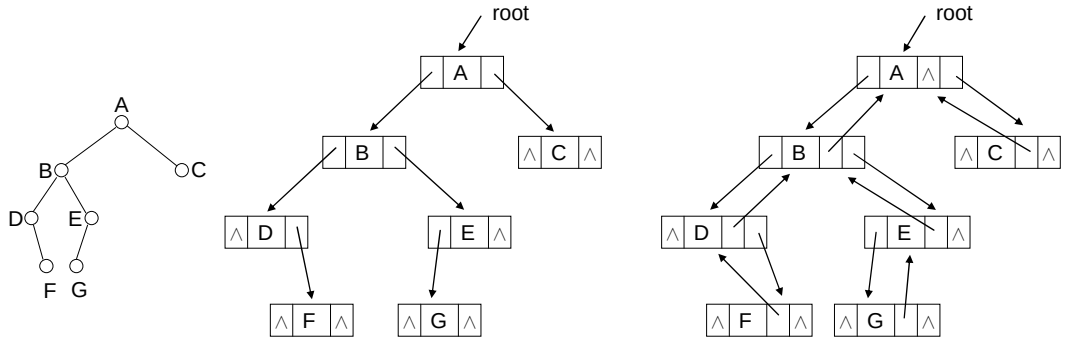
lchild	data	rchild

二叉链表的类型定义如下：

```
typedef struct node * pointer;
struct node {
  datatype data;
  pointer lchild,rchild;
};
typedef pointer bitree;
```

与单链表类似，这里定义了两个相同的类型 pointer 和 bitree，前者用于指向链表中的一般结点，后者用于指向链表的根结点，即代表二叉链表。

图 5.12（b）就是图 5.12（a）所示二叉树的二叉链表。若二叉树为空，则 root=NULL。若结点的某个孩子不存在，则相应的指针为空。



（a）二叉树　　（b）二叉链表　　（c）带双亲指针的二叉链表

图 5.12　二叉链表

下面几节给出的有关二叉树的各种算法，大多数是基于这种存储结构的。如果经常要在二叉树中寻找某结点的双亲（或者祖先等），则可采用三指针形式：在每个结点上再增加一个指向其双亲的指针域parent，形成一个带双亲指针的二叉链表[①]。图 5.12（c）就是图 5.12（a）所示二叉树的带双亲指针的二叉链表。至于一指针形式，就是在每个结点中只存放一个指向双亲的指针，这就是后面将要介绍的树的双亲表示法。

5.3　二叉树的遍历

在二叉树的一些运算中，经常要查找某种特征的结点，或对所有结点逐一进行某种处理，这就涉及二叉树的遍历问题，它是二叉树的一种重要运算。二叉树的**遍历**（Traversal）是指沿某条搜索路径周游二叉树，对每个结点访问一次且仅访问一次。这里的访问是指对结点进行某种处理，处理的内容依具体问题而定，可以是读、写、修改等。

我们知道，遍历一个线性结构很容易，只需从开始结点出发顺序扫描每个结点即可。但二叉树是一个非线性结构，每个结点可以有两个后继，因此，需要寻找某种规律来系统地访问树的各个结点。

5.3.1　二叉树的遍历方法

1．递归遍历

根据定义，一棵非空的二叉树是由根结点、左子树、右子树这三个部分组成的，因此，遍历一棵非空二叉树的问题可分解为 3 个子问题：访问根结点；遍历左子树；遍历右子树。若分别用 D、L 和 R 表示上述 3 个子问题，则有 DLR、LDR、LRD、DRL、RDL、RLD

① 有的文献称之为“三叉链表”。

等 6 种次序的遍历方案。其中前 3 种方案是按先左后右的次序遍历根的两棵子树，后 3 种方案则是按先右后左的次序遍历根的两棵子树，由于二者对称，故我们只讨论前 3 种次序的遍历方案。

对遍历方案 DLR，访问根的操作是在遍历其左、右子树之前进行的，故称之为前序遍历（或先根遍历）。类似地，LDR 和 LRD 分别称为中序遍历（或中根遍历）和后序遍历（或后根遍历）。由于左、右子树本身也是二叉树，对它们的遍历也要按同样的规则进行处理，于是，二叉树的遍历就成为一个递归问题，因而很容易写出 3 种遍历方法的递归定义。

（1）前序遍历

若二叉树非空，则依次进行如下操作：

① 访问根结点。

② 前序遍历左子树。

③ 前序遍历右子树。

（2）中序遍历

若二叉树非空，则依次进行下列操作：

① 中序遍历左子树。

② 访问根结点。

③ 中序遍历右子树。

（3）后序遍历

若二叉树非空，则依次进行以下操作：

① 后序遍历左子树。

② 后序遍历右子树。

③ 访问根结点。

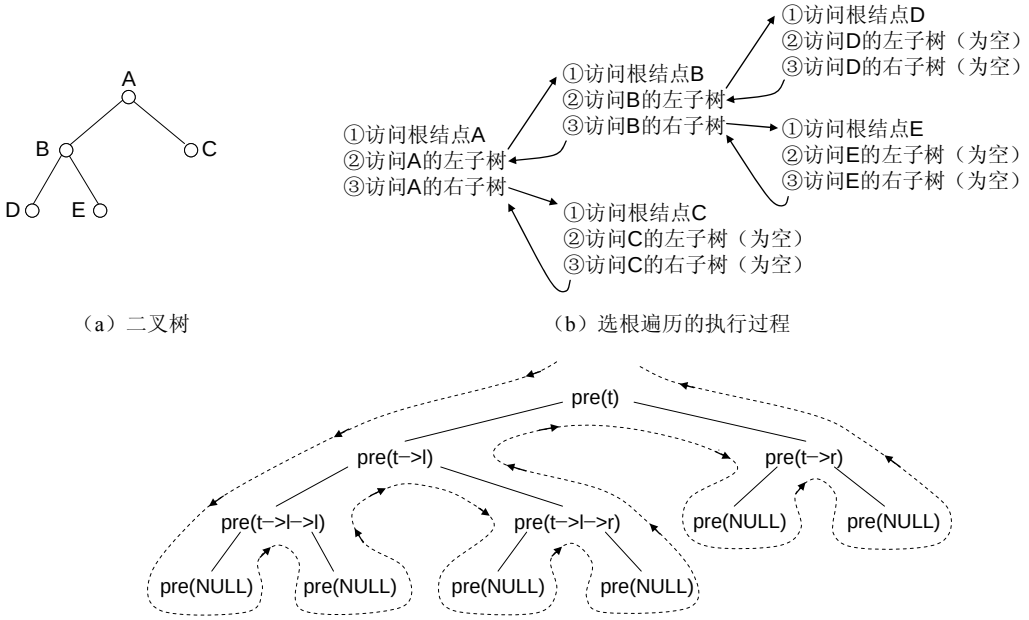
按以上 3 种方法遍历，都可得到所有结点的一个访问序列，分别称为先根（遍历）序列、中根（遍历）序列和后根（遍历）序列。例如，对图 5.12（a）所示的二叉树进行遍历，得到的先根序列为{A, B, D, F, E, G, C}；中根序列为{D, F, B, G, E, A, C}；后根序列为{F, D, G, E, B, C, A}。

由于上述遍历是递归定义的，故很容易写出相应的递归算法。显然递归终止条件是二叉树为空，此时应为空操作。假设对根结点的访问是打印结点数据，则在二叉链表上实现的 3 个遍历算法如下：

```
void preorder(bitree t) {   //先根遍历
  if(t==NULL) return;
  cout<<t->data<<endl;      //访问根
  preorder(t->lchild);      //先根遍历左子树
  preorder(t->rchild);      //先根遍历右子树
}
void inorder(bitree t) {    //中根遍历
  if(t==NULL) return;
  inorder(t->lchild);       //中根遍历左子树
  cout<<t->data<<endl;      //访问根
  inorder(t->rchild);       //中根遍历右子树
}
void postorder(bitree t) { //后根遍历
```

```
    if(t==NULL) return;
    postorder(t->lchild);      //后根遍历左子树
    postorder(t->rchild);      //后根遍历右子树
    cout<<t->data<<endl;       //访问根
}
```

为了便于理解上述 3 种递归算法，以先根遍历为例，对图 5.13（a）所示的二叉树，其先根遍历执行过程如图 5.13（b）所示，将其全部的递归调用关系画出来，得到图 5.13（c）。其中为简洁起见，将函数名 preorder 简写为 pre，将 lchild 和 rchild 分别简写为 l 和 r。虚线上的箭头表示各个递归调用的次序。可见，所有的调用过程（虚线）组成了遍历的搜索路线。递归调用关系完全可以画在原树上，得到更加简洁直观的图 5.14，其中“∧”表示虚结点。可以发现，搜索路线的特点是，从根结点出发，逆时针沿着二叉树外缘移动，每个结点均经过了 3 次。



（a）二叉树　　（b）选根遍历的执行过程

（c）选根遍历的递归调用关系

图 5.13　二叉树先根遍历的递归调用关系

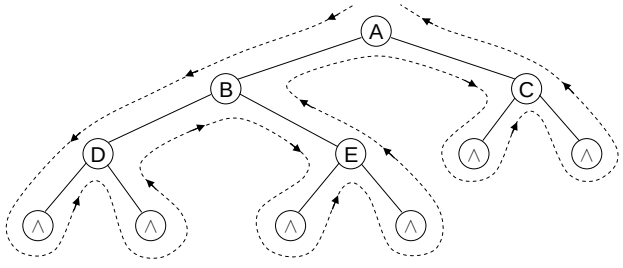


图 5.14　遍历二叉树的搜索路线

也可画出中根遍历和后根遍历的调用关系图，结果发现 3 种遍历的搜索路线是相同的。

这很好理解，因为，假设去掉对根的访问（或用空语句代替），3 种遍历算法就是一样的了，从而具有相同的搜索路线。

它们的区别在于访问结点的“时机”不同：若访问结点均是在第一次经过时进行，则是前序遍历；若访问结点均是在第二次（或第三次）经过时进行，则是中序遍历（或后序遍历）。因此，只要将搜索路线上所有在第一次、第二次和第三次经过的结点分别列表，即可分别得到该二叉树的前序序列、中序序列和后序序列。

遍历算法的基本操作是访问结点，显然，不论按哪种次序遍历，对含 n 个结点的二叉树，其时间复杂度均为 O(n)。所需辅助空间为递归遍历过程中栈的最大容量，即树的深度，最坏情况下为 n，则空间复杂度也为 O(n)。

中序序列具有下列特点（以后有关章节中会用到）：

① 中序序列的第一个结点是二叉树中最左下的结点。

② 中序序列的最后一个结点是二叉树中最右下的结点。

所谓**最左下**的结点，是指从根开始，沿左指针链往下查找，直到找到一个左指针为空（没有左孩子）的结点为止，这个结点就是“最左下”的结点。注意，最左下的结点虽然没有左孩子，但可能有右孩子（也可能无右孩子）。若无右孩子，则它必定是叶子。类似地，**最右下**的结点就是从根开始，沿右指针链往下查找，直到找到一个没有右孩子的结点为止，该结点是“最右下”的结点。

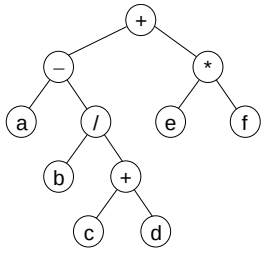
例 5.1 写出图 5.15 所示二叉树的先根、中根和后根遍历序列。

解：先根遍历序列为：`+-a/b+cd*ef`。

中根遍历序列为：`a-b/c+d+e*f`。

后根遍历序列为：`abcd+/-ef*+`。

实际上这种类型的二叉树称为**表达式树**，它的叶子结点表示操作数，分支结点表示操作符。如果一个表达式中的操作符都是双目或单目的，则一个表达式可对应到一棵二叉树。图 5.15 表示的实际上就是表达式 a−b/(c+d)+e*f。其先根、中根和后根遍历序列分别称为表达式的**前缀表示**（prefix，波兰式）、**中缀表示**（infix）和**后缀表示**（postfix，逆波兰式）：各操作符分别在对应操作数之前、之中和之后（但操作数的顺序相同）。



表达式 a−b/(c+d)+e*f 的二叉树

图 5.15 表达式树

中缀表示与原表达式一致，但无括号，这可对中根遍历略作修改：在遍历左子树前打印左括号，在遍历右子树后打印右括号，结果即得原表达式：(a−(b/(c+d))+(e*f))。后缀表示和前缀表示没有歧义，不必采用括号或优先级。

这里顺便说说表达式的求值。后缀表达式和前缀表达式的求值较简单：借助操作数栈，从左向右或从右向左扫描表达式，若遇到操作数，就入栈；若遇到操作符，则操作数出栈，由操作符运算后把结果作为操作数再入栈……。而中缀表达式的求值较麻烦，需要操作数和操作符两个栈，还要考虑各操作符的优先级（对扫描到的操作符分情况处理）。也可先转化为后缀表示或前缀表示后再求值，这要借助操作符栈，也要考虑各操作符的优先级。具体就不细述了。

2．层序遍历

由于树形结构的结点间形成分支和层次关系，所以如果逐层地对结点进行访问，也可

将所有结点都访问到，这就是层序遍历。所谓二叉树的**层序遍历**，是指从第一层（即根）开始，按从上层到下层，每层内按从左到右的顺序对结点逐个访问。如对图 5.12(a)所示的二叉树，其层序遍历序列为 A, B, C, D, E, F, G。实际上，前面对二叉树结点进行层序编号的过程就是一种层序遍历，只是访问结点时的操作是给它一个编号而已。

由于上下层结点之间具有父子关系，在层序遍历中必然是先访问的结点其孩子结点也先访问，即若结点 A 先于结点 B 访问，则结点 A 的孩子也先于结点 B 的孩子访问。这样，在层序遍历中，我们可用一个队列来保存待访问的结点(已访问结点的孩子)。遍历算法为：每访问一个结点，就将它的孩子指针入队，下一个要访问的结点是队头（出队）；这个过程不断进行，直到队列为空。

除了第一个结点（即根）外，其他结点都是从队列中取出并进行处理的。为使根和其他结点的处理一致，可采用**预入队**技术，即在算法开始时先将根结点入队，然后马上出队再进行有关处理。非形式算法如下：

```
入队(根指针);
while(队不空) {
  出队(指针 p);
  访问 p;
  入队(左孩子);
  入队(右孩子);
}
```

假设对结点的访问是打印结点数据，并注意到空指针不必入队，则具体算法如下：

```
void levelorder(bitree t) {     //层序遍历
  pointer p;
  sqqueue Q;                    //循环队列，队列元素为结点指针，类型为 pointer
  if(t==NULL) return;
  init_sqqueue(&Q);
  en_sqqueue(&Q,t);             //根结点入队
  while(!empty_sqqueue(&Q)) {   //队列非空时
    de_sqqueue(&Q,&p);cout<<p->data<<endl;          //出队，访问队头
    if(p->lchild!=NULL) en_sqqueue(&Q,p->lchild);   //左孩子入队
    if(p->rchild!=NULL) en_sqqueue(&Q,p->rchild);   //右孩子入队
  }
}
```

对该算法进行修改，还可输出各结点的层数、树的高度、宽度等。

上面介绍的几种遍历序列都是线性序列，有且仅有一个开始结点和一个终端结点，其余结点都有且仅有一个前趋和一个后继。为了区别树形结构中前趋（即双亲）和后继（即孩子）的概念，对上述各种线性序列，我们在结点的前趋和后继之前冠以相应遍历次序的名称。例如，图 5.12（a）所示二叉树中，结点 B 的层序前趋是 A，层序后继是 C；中序前趋是 F，中序后继是 G 等。但就该树的逻辑结构而言，结点 B 的前趋是 A，后继是 D 和 E。

上述各种遍历都是基于二叉链表的，但显然也可在顺序存储结构上实现。这时，层序遍历由存储结构直接得到（去掉虚结点），递归遍历则注意利用顺序存储规律即可，如某树为空即该树根结点对应的单元为虚结点、结点 i 的左右孩子对应编号为 2i 和 2i+1 的单元等，具体算法就不细述了。

回顾一下链表，为了对所有结点进行某种处理，如累计结点数（求表长），需要从头

开始沿着指针链逐个结点进行，这实际上就是链表的遍历。一般而言，所谓遍历就是对所有数据结点按照某种方式无重复无遗漏地访问，即访问且只访问一次。数据结构中的很多算法，都是基于遍历的，大家在学习时要注意体会和掌握。

对非线性结构树以及下一章将要介绍的图，遍历后可得到所有结点按某种次序排列的一个线性序列，所以遍历也可看成是从非线性结构到线性结构的一种映射方法。

5.3.2 二叉树遍历与递归举例

二叉树的定义是递归的，所以二叉树的很多问题都可以很自然地进行递归处理，其中就包括遍历。本来，遍历只是这些递归处理中的一种，但如果我们广义地看待对根的访问，即相应的处理可以是任何操作，如输出其内容、某种性质的判断、某种信息的处理等，则很多问题都可以归结到遍历上。这样对同一个问题，我们可以直接用递归来处理，也可以用某种特殊的遍历来处理。一般来说，如果问题具有比较明显的逐个结点处理的特点，则用遍历比较直观；否则就按根、左子树、右子树三部分递归处理比较简便。

递归程序一般包括递归出口和递归体两部分。在二叉树中，递归出口一般是二叉树为空或叶子，此时不能再递归，只能回退；递归体一般是对根和左子树、右子树的某种处理。

例 5.2 求二叉树的叶子结点数。

解：这个问题具有明显的逐个结点处理的特点：如果当前结点是叶子，则叶子数加 1。所以可用各种遍历法，以先根遍历为例，算法如下：

```
int num=0;                //num 为叶子数，设为全局量，并初始化为 0
void leaf(bitree t) {
  if(t==NULL) return;     //空树什么都不做
  if(t->lchild==NULL && t->rchild==NULL) num++;//访问根：若为叶子，则叶子数+1
  leaf(t->lchild);        //访问左子树：累计其中的叶子数
  leaf(t->rchild);        //访问右子树：累计其中的叶子数
}
```

上面将叶子数设成全局量而不作为参数，是考虑到递归函数执行时，参数要频繁传递，如果减少参数的个数，可提高递归函数的执行效率。

求叶子数问题，也可直接从递归的角度来考虑：二叉树的叶子数等于左子树和右子树的叶子数之和，但左子树和右子树本身又是二叉树，求其叶子数要递归进行。算法如下：

```
int leaf2(bitree t) {   //叶子数通过函数值返回
  int L,R;
  if(t==NULL) return 0;//空树的叶子为 0
  if(t->lchild==NULL && t->rchild==NULL) return 1; //树中只有一个点，就是叶子
  L=leaf(t->lchild);    //求左子树的叶子数(访问左子树)
  R=leaf(t->rchild);    //求右子树的叶子数(访问右子树)
  return L+R;           //叶子数=左右子树叶子数之和(访问根)
}
```

注意，该算法需要两个递归出口：空和叶子。另外，即使是这个算法，也可看成一种（后根）遍历：先访问子树（求子树叶子数），再访问根（将左、右子树的叶子数加起来作为当前子树的叶子数），但这显然不如直接按递归考虑容易理解。

按类似思想，可求二叉树的高度、度 1 结点数、度 2 结点数等。比如二叉树的高度是左、右子树中高度较大者加 1，但左、右子树本身又是二叉树，故递归。具体算法略。

例 5.3　交换二叉树各结点的左右子树。

解：这个问题显然可按遍历思想处理。以先根遍历为例，算法如下：

```
void exchange(bitree t) {   //先根遍历型
  pointer p;
  if(t==NULL) return;       //空树
  p=t->lchild;t->lchild=t->rchild;t->rchild=p; //交换
  exchange(t->rchild);      //遍历原左子树(现为 t->rchild)
  exchange(t->lchild);      //遍历原右子树(现为 t->lchild)
}
```

注意交换后左右顺序与原来相反，如访问原右子树时参数为 t–>lchild。类似，若采用中根遍历，则在访问右子树时参数也应为 t–>lchild（这时遍历左、右子树时的参数名称上都为 t–>lchild，但含义不同，一个是交换前的，一个是交换后的），否则最终只交换了根结点的左右孩子。显然，采用后根遍历则不存在这些问题。

例 5.4　已知二叉树各结点存放的是字符，判断是否所有结点存放的都是数字字符。

解：这个问题显然可按逐结点遍历处理，但这里不一定要遍历完所有结点：一旦发现某个结点或子树不合要求，就不必对后继子树进行遍历了。算法如下：

```
int detect(bitree t) {
  int x;
  if(t==NULL) return 1;     //空树
  if(t->data<'0'|| t->data>'9')return 0;//访问根：若非数字字符，则跳过子树检查
  x=detct(t->lchild); if(x==0) return 0;//遍历左子树，若为假，则跳过右子树检查
  x=detct(t->rchild);       //遍历右子树
  return x;                 //最后结果由右子树决定
}
```

注意，该算法将空树看做真，否则还需增加一个递归到叶子（并判断真假）的递归出口。

本例也可直接用递归的思想处理：如果根不是数字字符，则当前结果肯定为假，返回；否则当前结果由左子树和右子树的情况共同决定。算法如下：

```
int detect2(bitree t) {
  if(t==NULL) return 1;                 //认为空树符合条件(真)
  if(t->data<'0'||t->data>'9') return 0;   //访问根：若不是数字字符则返回假
  return detct(t->lchild) && detct(t->rchild); //左子树和右子树共同决定
}
```

由逻辑与运算的短路规则，该算法的返回语句当检测到左子树为假时，并不会再检测右子树而直接返回假（实际执行过程与上一算法相同）。

例 5.5　判断两棵二叉树是否等价，即要么都为空；要么根相同，且左右子树分别等价。

解：这个问题比较适合直接按递归思想处理，算法如下：

```
int same(bitree t1,bitree t2) {
  if(t1==NULL && t2==NULL) return 1;    //同时为空树
  if(t1==NULL || t2==NULL) return 0;    //一个为空，另一个非空
```

```
  if(t1->data!=t2->data) return 0;      //根不相等
  return same(t1->lchild,t2->lchild) && same(t1->rchild,t2->rchild);
                                        //左子树和右子树共同决定
}
```

注意，算法在检查两个根中一个空，另一个非空的条件不是(t1==NULL && t2!=NULL) || (t1!=NULL && t2==NULL)，而是 t1==NULL || t2==NULL，这是因为经过前一步两者同时为空的检查后，t1 和 t2 已不可能同时为空了，所以若某一个为空，则另一个必不为空。

另外，算法也利用了逻辑与运算的短路规则，如果左子树不等价，并不会检查右子树。

顺便指出，例 5.4 和例 5.5 对根和左、右子树的检查过程也可总体上写成一句的形式：return 根&&左&&右，但多个条件连在一起后反而显得不够简洁。

5.4 二叉树的生成

二叉树的生成是指如何在内存中建立二叉树的存储结构。建立顺序存储结构较简单，这里仅讨论如何建立二叉链表。要建立二叉链表，需要按某种方式输入二叉树的结点及其逻辑信息。注意到二叉树遍历时，不仅得到了结点信息，而且由序列中结点的先后关系还可获得一定的逻辑信息，如果这些信息足够，就可根据遍历序列生成相应的二叉树。

本节二叉树的生成方法就是基于遍历序列的，相当于遍历问题的逆问题，即由遍历序列反求二叉树，这需要分析和利用二叉树遍历序列的特点。以下分 3 种情况来讨论。

1．层序遍历序列

按完全二叉树的层次顺序，依次输入结点信息来建立二叉链表。这是因为完全二叉树的层序遍历序列中，结点间的序号关系可反映父子关系即逻辑关系。对一般的二叉树，要补充若干个虚结点使其成为完全二叉树后，再按其层次顺序输入。例如，仅含 3 个结点 A、B、C 的右单支树（见图 5.11），按完全二叉树的形式输入的结点序列为：A@B@@@C#，其中@表示虚结点，#表示输入结束。

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。如此重复下去，直至输入字符“#”为止。

这里的关键是新结点与其双亲的链接。由于结点是按层次自左向右输入的，所以先输入的结点，其孩子也必定较先输入。即结点与其孩子具有先进先出的特点，于是可设置一个队列，保存已输入结点的地址。这样，队尾是当前正输入的结点，队头是其双亲结点。当队头结点的两个孩子都输入完毕后，出队，新的队头是下一个要输入孩子的双亲结点。如此下去，直到输入结束符为止。

双亲与孩子的链接方法是：若当前输入的结点编号是偶数，则该结点作为左孩子与其双亲链接；否则作为右孩子与其双亲链。若双亲结点或孩子结点为虚结点，则无需链接。

如果采用顺序队列，则队列是一个指针数组。为使队列元素在数组中的下标与其结点的层序编号一致，数组从下标 1 开始使用。注意，这里不是循环队列。具体算法如下：

```
bitree level_creat() {      //按层序序列建立二叉树，返回根指针
  char ch;
```

```
  pointer Q[maxsize+1];      //非循环队列，有效下标从 1 到 n，maxsize 为最大结点数
  int front,rear;
  pointer root,s;
  root=NULL;                 //置空二叉树
  front=rear=0;              //置空队列
  while(cin>>ch,ch!='#') {   //输入字符，若不是结束符则循环
    if(ch!='@') {            //非虚结点，建立新结点
      s=new node;
      s->data=ch;
      s->lchild=s->rchild=NULL;
    }
    else s=NULL;
    rear++;Q[rear]=s;                    //不管结点是否为虚，都要入队
    if(rear==1) {root=s;front=1;}        //第一个结点是根，要修改头指针，它不是孩子
    else {
      if(s && Q[front])                  //孩子和双亲都不是虚结点，链接之
        if(rear%2==0) Q[front]->lchild=s;   //rear 是偶数，新结点是左孩子
        else          Q[front]->rchild=s;   //rear 是奇数，新结点是右孩子
      if(rear%2==1) front++;             //不论是否为虚，右孩子入队后，双亲出队
    }
  }
  return root;
}
```

2．先根、中根或后根遍历序列

基本思想是输入这 3 个遍历序列中的一个，二叉树的结点就按相应的遍历过程逐个生成。类似于层序遍历，如果不对遍历序列作些补充，是不能完整地反映结点间的逻辑关系的，也就不能得到正确的结果。补充的方法也是增加虚结点，但这里只需对空指针对应的位置进行补充，而不必补充到完全二叉树的形式。

以先根遍历和图 5.12(a)为例，二叉树的先根输入序列为 ABD@F@@EG@@@C@@，其中@表示虚结点。注意，这里不需要结束符。算法过程为，先生成根结点，再生成左子树，然后是右子树，左右子树的生成采用递归，算法如下：

```
bitree pre_creat() {        //由先根序列建立二叉树，返回根指针
  bitree t;
  char ch;
  cin>>ch;
  if(ch=='@') return NULL;  //虚结点
  t=new node;               //生成根结点
  t->data=ch;
  t->lchild=pre_creat();    //生成左子树
  t->rchild=pre_creat();    //生成右子树
  return t;
}
```

该算法的调用形式为“bitree T; T=pre_creat();”等。

若给定的是后根序列，由于第一个结点就是虚结点，为递归出口，于是后面递归体一次也不会执行，从而生成不了二叉树。但把序列倒过来看，则相当于按“根→右→左”的顺序遍历原二叉树，于是可把原序列逆置后，按“根→右→左”的“先根遍历”来生成二叉树。具体算法略。但是，若给定的是中根序列，除了第一个结点为虚结点的问题外，更

主要是因为不能确定谁是根，二叉树不唯一，从而不能生成二叉树。

3．双遍历序列

上面看到，单个的遍历序列一般需要补充虚结点才能完整表示结点间的逻辑关系。但两个或多个信息不完整的序列如果能相互补充，也可能得到完整的信息。这需要分析和利用遍历序列的特点。由遍历方法得到以下几点是显然的：

（1）对前序序列，序列的第一个结点就是整个二叉树的根。

（2）对后序序列，序列的最后一个结点就是整个二叉树的根。

（3）对中序序列，以根为界，序列的前一部分为根的左子树，后一部分为根的右子树；并且，由前一部分构成的子序列是左子树的中序序列，由后一部分构成的子序列是右子树的中序序列。

若给定了前序序列和中序序列，反复利用上面的（1）和（3），就可获得完整的逻辑信息，即由前序序列找到根，由中序序列得到左、右子树；再对每个子树由前序序列找到子树的根，由中序序列得到子树的左、右子树，……，依此类推，每次得到一个结点（子树的根），从而逐渐分离出树的全部信息，也就可以还原和构造出该二叉树。这个过程参见图 5.16。

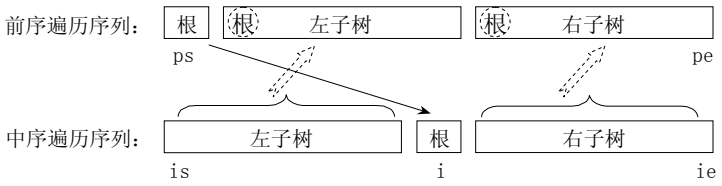


图 5.16　由前序和中序序列建立二叉树

由于构造过程是递归的，可方便地写出相应的递归算法：

```
bitree creat(char Pre[],int ps,int pe,char In[],int is,int ie) {
//由先序和中序建立二叉树，Pre[]和 In[]为遍历序列，ps,pe 以及 is,ie 分别为区间始末点
  bitree t;
  int i;
  if(ps>pe) return NULL;
  t=new node;                    //生成根结点
  t->data=Pre[ps];
  i=is;
  while(In[i]!=Pre[ps]) i++;    //寻找中序序列中根的位置 i
  t->lchild=creat(Pre,ps+1,ps+i-is,In,is,i-1); //生成左子树
  t->rchild=creat(Pre,ps+i-is+1,pe,In,i+1,ie); //生成右子树
  return t;
}
```

注意，在递归中要用参数指明当前正在处理的子树的中序与前序序列在原来整个中序与前序序列中的位置。

一般地，由中序序列与前序序列、中序序列与后序序列、中序序列与层序序列等都能唯一确定二叉树。而由前序序列和后序序列因不能确定左右子树，一般就不能唯一确定二叉树（除非二叉树为空或只有一个结点；或补充其他条件如为严格二叉树等）。比如由前序序列{A, B, D, C}和后序序列{D, B, C, A}可得到如图 5.17 所示的两棵二叉树。

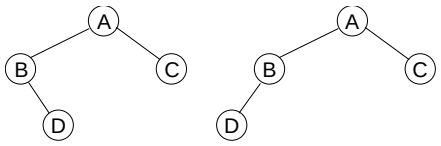


图 5.17　前序序列和后序序列分别相同的两棵二叉树

如果要把一棵二叉树存储到外存（文件）中，我们就可只存储它的中序序列与前序序列，或者中序序列与后序序列等，以后在需要时再将二叉树还原出来。这样，既不必像顺序存储那样，存储大量虚结点，又不必像二叉链表那样，存储大量附加指针。因此，也可看成是二叉树的一种存储方法。

5.5　递归消除

在 5.3.1 节中，我们由遍历的递归定义直接写出了遍历的递归算法。一般地，对一个递归问题，设计出求解该问题的递归算法是比较容易的，并且一般也有较好的可读性。然而，有时需要考虑怎样将一个递归算法转换成等价的非递归算法，这称为“**递归消除**（recursion removal）”。研究递归消除的原因主要有以下几个方面：

第一，递归消除有利于提高算法的时空性能。一般而言，递归算法的时空性能相对较差，非递归算法的时空性能要高得多。比如，递归程序执行时需反复调用自己，与调用其他函数一样，每调用一次，系统就在其运行栈中存放函数的返回地址、参数、临时变量等，随着递归的进行，运行栈不断增长，只有当函数彻底执行完后，才释放它所占用的栈空间。因此，递归算法不但不节省空间，而且一般还比非递归算法浪费空间。如果某个递归算法的时空性能较差，又要被频繁地使用，则将它转换为非递归算法的意义就更大了。

第二，研究递归消除有助于透彻理解递归机制，而这种理解是熟练掌握递归程序设计技能的必要前提。

第三，有些程序设计语言不允许递归，如 FORTRAN 等高级语言和大多数汇编语言，这时就不得不考虑非递归算法。

一般说来，如果将原来由系统管理的递归程序的工作过程（主要是工作栈的工作过程）改为由程序员来模拟和管理，都可实现递归消除，但这并不是说递归消除一定要利用栈；即使要用栈，也不一定要将工作栈的工作过程都模拟出来，比如，若保存的现场信息在返回后不再使用，则可不必保存，从而提高程序效率。本节根据是否需要引入工作栈作为控制机构，将递归消除技术分成两类加以讨论。

5.5.1　简单递归消除

对于简单的递归算法，可以不通过工作栈作控制机制而直接转换成循环算法。这时计算依赖图（函数调用关系图）的分析和化简是一种非常有用的辅助手段。下面通过两个例子来进行说明。

例 5.6　将计算阶乘 n!的递归算法转换成非递归算法。

解：求阶乘的递归算法为：

```
long f(int n) {
  if(n==0) return 1;
  else return n*f(n-1);
}
```

计算依赖图的画法是：对每一个递归调用，在图中用一个结点来表示；对任意两个递归调用 p、q，若 p 的计算直接依赖于 q，即 p 的执行中调用了 q，则在 p 和 q 之间画一条线，并且 p 点的位置高于 q 点。通常只能且只需画出计算依赖图的一部分。

对于计算 n!的递归算法 f(n)来说，以 n=3 为例，f(3)的计算直接依赖于 f(2)，因为 f(3)调用了 f(2)，仅当 f(2)的计算完成之后，f(3)的计算才能完成。类似地，f(2)直接依赖于 f(1)；f(1)直接依赖于 f(0)；而 f(0)不依赖于其他调用，计算由自己独立完成。包含这几个递归调用的计算依赖图如图 5.18 所示。图中虚线表示递归调用和返回的次序，向下的箭头表示递归调用；向上的箭头表示返回。根据计算依赖图的画法和含义，这些虚线和箭头可以省略。

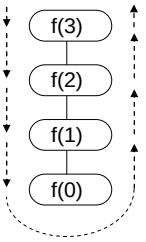


图 5.18 f(n)的部分计算依赖图

对照图 3.6（a）不难看出，图 5.18 只不过是它的一种简化形式。事实上，计算依赖图集中反映了不同递归调用之间的依赖关系。通过对这种关系的分析，可以很方便地写出完成同样任务的循环算法。

从图 5.18 中可以明显看出，递归算法 f(n)的计算实际上由一个自上而下的递归调用阶段和一个自下而上的返回阶段构成；并且所有递归调用都直接或间接地依赖于 f(0)。因此，整个计算完全可以只由后一个阶段完成，即先后依次计算 f(0)、f(1)、f(2)、…、f(n)。这是一个**递推**（recurrence）过程，可用循环完成。引入一个工作变量记录中间结果，则计算 n!的循环算法为：

```
long f1(int n) {
  long x;
  int i;
  x=1;
  for(i=1;i<=n;i++) x=x*i;
  return x;
}
```

显然，该算法的时间复杂度为 O(n)。

作为对比，这里分析一下前面的递归算法 f(n)。在求 f(n)时需先求 f(n−1)；若已知 f(n−1)，则其他运算（乘法和返回）的时间为常量，设为 c，即 T(n)=T(n−1)+c，但 n=0 时不递归直接返回结果，执行时间也为常量，设为 d，即 T(0)=d。于是：

$$\begin{aligned} T(n) &= T(n-1)+c \quad = [T(n-2)+c]+c \\ &= T(n-2)+2c = [T(n-3)+c]+2c \\ &= T(n-3)+3c = \cdots \\ &= T(0)+nc \\ &= d+nc = O(n) \end{aligned}$$

可见这里递归和非递归算法的时间复杂度相同，但显然非递归算法没有函数调用和返

回等时空开销，其实际执行效率要高些。

上述问题比较简单，计算依赖图中没有分支（**单向递归**），且递归调用语句是递归程序的最后一句（**尾递归**），熟练后这类问题可不画计算依赖图而直接用循环消除递归（得到迭代算法）。在较复杂的情况下，计算依赖图为递归消除提供了一种直观的分析工具。

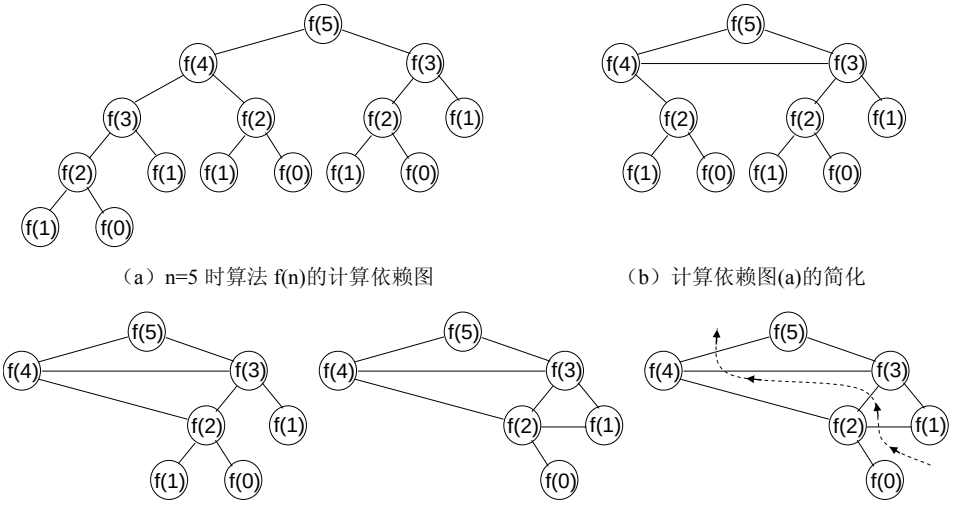
例 5.7　写出计算菲波那契数列的递归函数并消除递归：

$$\mathrm{Fib(n)}=\begin{cases}0 & \mathrm{n}=0\\ 1 & \mathrm{n}=1\\ \mathrm{Fib(n-1)+Fib(n-2)} & \mathrm{n}>1\end{cases}$$

解：由于 Fib(n)是递归定义的，可直接写出它的递归算法如下：

```
int f(int n) {
  if(n==0) return 0;
  else if(n==1) return 1;
  else return f(n-1)+f(n-2);
}
```

n=5 时算法 f(n)的计算依赖图见图 5.19（a）所示。显然，f(n)的计算依赖关系是一个树形结构，树上每个结点的计算直接依赖于其所有孩子，并直接或间接依赖于其所有子孙。



（a）n=5 时算法 f(n)的计算依赖图　　（b）计算依赖图(a)的简化

（c）计算依赖图(b)的简化　　（d）计算依赖图(c)的简化　　（c）循环算法的计算轨迹

图 5.19　算法 f(n)的部分计算依赖图及其简化

从图 5.19（a）可以看出，递归算法 f(n)的效率不高。其中，f(3)被计算了两次，f(2)被计算 3 次，f(1)和 f(0)分别被计算 5 次和 3 次。可以证明（类似附录 E 中菲波那契数列通项的推导），计算 f(n)时所有递归调用的总次数为指数级 $O(2^n)$，效率是非常低的。

显然，如果能把重复的结点合并在一起，就可避免重复计算，从而提高效率。这可自上而下地按下列步骤得到：第一步，将两个 f(3)结点合并，得到子图（b）所示的计算依赖图；其次，将子图（b）中的两个 f(2)结点合并得子图（c）；最后，将子图（c）中的两个 f(1)结点合并，得到子图（d）。这个过程就是计算依赖图的化简。

由图 5.19（d）可知，当 n≤5 时，f(n)的全部计算只依赖于 f(0)和 f(1)。易知此结果对

任意 n≥2 都成立。因此，f(n)的计算可以通过自下而上的循环算法完成；相应的计算轨迹可设计成子图（e）中的虚线。由于当 n≥2 时，每个 f(n)直接依赖于刚刚计算出的前两个值，故引入两个工作量 x、y 分别记录中间结果。具体算法如下：

```
int f1(int n) {
  int x,y,z,i;
  if(n==0) return 0;
  if(n==1) return 1;
  x=0;y=1;
  for(i=2;i<=n;i++) {
    z=x+y;
    x=y;y=z;
  }
  return z;
}
```

在计算 f(n)的过程中，对每个 f(i)（i=0, 1, …, n），此算法只计算一次，故其时间性能明显高于递归算法 f(n)。

一般地，计算依赖图都可看成树型结构（图 5.18 为单支树），故也常称**递归树**（Recursion Tree），它是分析递归算法的一个重要工具，如树的高度对应递归深度，决定了递归所需栈空间的大小；树中结点数对应递归调用次数，决定了递归所需时间的多少等。

5.5.2 基于栈的递归消除

在很多情况下，递归算法的计算依赖图无法简化成线性结构（如图 5.18）或准线性结构（如图 5.19（d）），因而无法直接转换成循环算法。例如，从表面上看，二叉树的 3 种遍历的递归算法与计算 f(n)的递归算法似乎并无太大的区别，只是过程体中包含两处递归调用。然而，遍历算法的计算依赖图是不能简化的，比如在图 5.13（c）和图 5.14 中，各分支结点的实参实际上是不同的（对应二叉树上的不同结点），故不能合并化简。其中，图 5.14 正是先根遍历算法的计算依赖图。

在这种情况下，通常引入一个工作栈作为控制机构以消除递归。在例 3.3 中曾提到，编译系统利用工作栈来保存返回位置以实现过程调用与返回控制，这一思想同样适用于递归消除。下面以先根遍历（以下用 pre 表示）为例，具体讨论如何用工作栈来消除递归。

首先要弄清工作栈的作用方式，即工作栈怎样控制先根遍历的走向。图 5.20（a）所示为二叉链表上的任一结点 x 以及它的左、右子树 X_L 和 X_R。假设 t 是指向结点 x 的指针，则与子图（a）相应的有关递归调用如子图（b）所示。由于是先根遍历，当遍历到结点 x 时，即执行 pre(t)时，有 3 项工作需顺序完成：

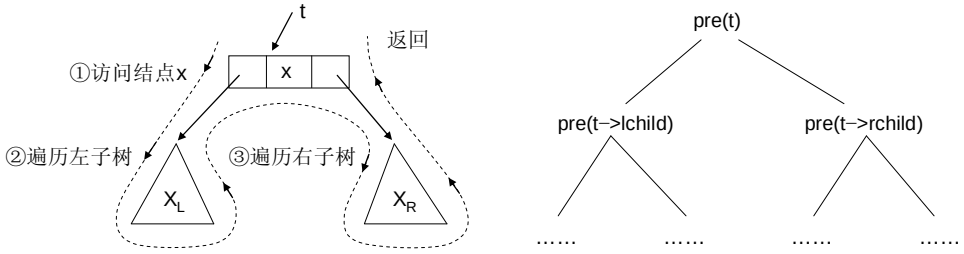
（1）访问结点 x（子树的根）。

（2）遍历 X_L，即调用 pre(t->lchild)。

（3）遍历 X_R，即调用 pre(t->rchild)。

其中，（1）和（2）的连接没有问题，但（2）与（3）如何连接需要考虑。为了执行（3），必须知道 X_R 的根指针，即结点 x 的右指针 t->rchild；但 x 的左子树 X_L 上并没有这个指针。所以在执行（2）之前应将指针 t->rchild 保存起来；在任务（2）完成之后再取出该指针以执行任务（3）。此后，指针 t->rchild 就没有保存价值了。对于先根遍历来说，完成

了任务（3）也就完成了对以结点 x 为根的整个子树的遍历，接下来便是退回到 x 的父结点。



（a）二叉链表的一个局部　　（b）对应的先根遍历递归调用

图 5.20　先根遍历的分析

在整个遍历过程中，较后访问的结点在以后返回时较先返回，相应地，其右子树也较先访问。所以应以栈的方式保存结点信息。

以上保存的是 t->rchild。也可保存结点本身的地址 t，以后再通过 t->rchild 访问其右子树，但显然不及前者好。引入工作栈后，非递归算法在二叉链表的任一结点 x 上的主要操作步骤可归纳如下：

（1）若结点 x 不空，则访问结点 x：visit(p)。

（2）结点 x 的右指针进栈：push(p->rchild)。

（3）遍历 X_L：p=p->lchild，转（1）。

（4）退栈，从栈中得到 x 的右指针：pop(p)。

（5）遍历 X_R：转（1）。

由于子树也是二叉树，所以步骤（3）、（5）对子树的操作即将流程控制转（1）。对图 5.21（a）所示的二叉链表，按上述步骤执行的主要过程如图 5.21（b）～图 5.21（k）所示。图中 lchild 和 rchild 分别缩写为 l 和 r。

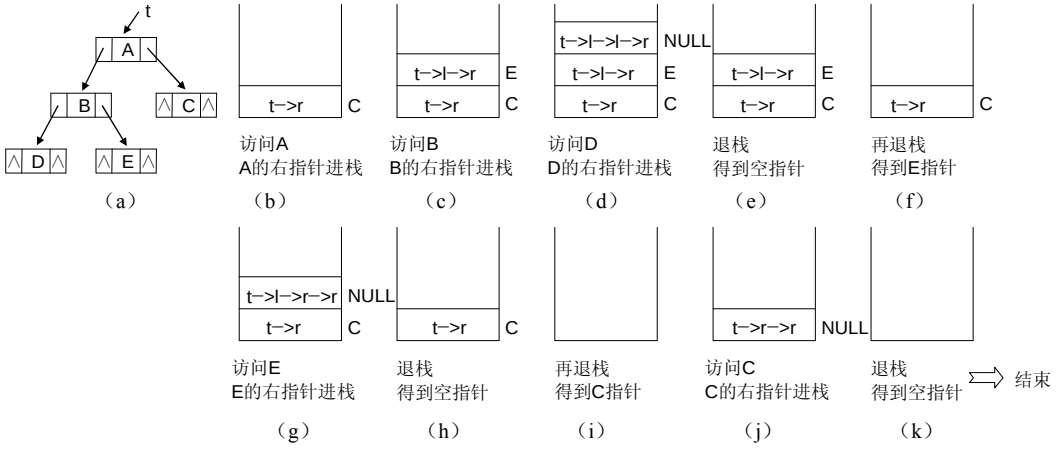


图 5.21　先根遍历的工作栈状态变化示例

从中可见，在执行过程中可能多次出现栈空，但中间几次栈空时取出来的指针不空，只有最后栈空且取出来的指针也空时才终止算法。整个算法如下：

```
void preorder(bitree t) {   //先根遍历算法，非递归
  pointer p,S[maxsize];     //顺序栈
  int top;                  //栈顶指针
  if(t==NULL) return;
  top=-1;
  p=t;
  while(!(p==NULL && top==-1)) {
    while(p!=NULL) {
      visit(p);             //访问根
      S[++top]=p->rchild;   //右指针入栈
      p=p->lchild;          //向左搜索
    }
    p=S[top--];
  }
}
```

该算法的遍历过程如下：从根开始，顺左分支往下搜索，每搜索到一个结点，就访问该结点，同时将其右指针进栈，直到左分支为空。然后，退栈，取出最近保存的一个右指针，如果为空，再退栈取出另一个右指针；否则对右指针对应的右子树进行同样处理。如此一直进行下去，直到所有结点均已处理完。其中，退栈及其以后的操作，对应于递归遍历算法中，返回到已搜索过的最近一个（有右孩子的）结点，然后再对其右子树进行递归处理的过程。

该算法最先访问的是根，它的处理与其他结点是有所不同的：根的值是直接给定的，其他结点的值则是循环中从栈或左指针得到的。为使根和其他结点的处理方式一致，并简化循环条件，可采用**预入栈**技术，即在算法开始时先将根结点入栈，然后马上出栈进行有关处理，整个算法相当于从上述步骤（4）开始。非形式算法和求精后的算法如下：

```
push(根指针);
while(栈不空) {
  pop(指针 p);
  while(p!=NULL) {
    访问 p;
    push(p->rchild);
    p=p->lchild;
  }
}
void preorder2(bitree t) {  //先根遍历算法，非递归，根预入栈
  pointer p,S[maxsize];     //顺序栈
  int top;                  //栈顶指针
  if(t==NULL) return;
  top=-1;
  S[++top]=t;               //根指针入栈
  while(top>=0) {           //栈非空
    p=S[top--];             //出栈
    while(p!=NULL) {
      visit(p);             //访问根
      S[++top]=p->rchild;   //右指针入栈
      p=p->lchild;          //向左搜索
    }
  }
}
```

注意，对该算法，在图 5.21(b)之前还有一个根指针进栈和出栈的过程。

如果将算法中的赋值语句 p=p->lchild 通过栈来等效完成 S[++top]=p->lchild，p=S[top--]，并注意到空指针不必入栈，则上述算法的两个循环可合为一个，算法如下：

```
void preorder3(bitree t) { //先根遍历算法，非递归，根预入栈，改进
  pointer p,S[maxsize];     //顺序栈
  int top;                  //栈顶指针
  if(t==NULL) return;
  top=-1;
  S[++top]=t;               //根指针入栈
  while(top>=0) {           //栈非空
    p=S[top--];             //出栈
    visit(p);               //访问根
    if(p->rchild!=NULL) S[++top]=p->rchild;//右指针入栈
    if(p->lchild!=NULL) S[++top]=p->lchild;//左指针入栈
  }
}
```

中序遍历的非递归算法也很容易。中序遍历时，要先访问左子树，所以每搜索到一个结点时并不立即访问它，只有左子树访问完后下一步才访问根。于是，对搜索到的每一个结点，应当保存（入栈）当前结点自己的指针（而不是其右指针），以便搜索回退时得到需要的返回位置（出栈），再访问之。算法如下：

```
void inorder(bitree t) {    //中根遍历算法，非递归
  pointer p,S[maxsize];     //顺序栈
  int top;                  //栈顶指针
  if(t==NULL) return;
  top=-1;
  p=t;
  while(!(p==NULL && top==-1)) {
    while(p!=NULL) {        //搜索到最左下的结点
      S[++top]=p;
      p=p->lchild;
    }
    p=S[top--];
    visit(p);               //访问根
    p=p->rchild;            //向右搜索
  }
}
```

后序遍历的非递归算法要复杂一些。在后序遍历时，最后访问根结点，所以对任一结点，应先沿它的左分支往下搜索，每搜索到一个结点就进栈，直到左分枝为空；然后退栈，取出最后入栈的结点 x，但此时并不访问它，而是从该结点的右分支（若有的话）的根开始，按同样的方法沿它的右分枝处理；处理完结点 x 的右分枝后才能访问结点 x。

因此，任一结点进栈后，都有两次退回到栈顶：第一次是在处理完它的左分支时，第二次是在处理完它的右分枝时。显然，根结点只有在第二次退回到栈顶时才出栈并访问它。为了区分是第几次退回到栈顶，一般有两种方法。

（1）检查一下刚访问的结点是否是栈顶的右孩子，算法如下：

```
void _postorder(bitree t) {//后根遍历，非递归，无进栈标志
  pointer p,q,S[maxsize];   //顺序栈
  int top;
  if(t==NULL) return;
```

```
  top=-1;
  p=t;q=NULL;                  //左子树尚未遍历，最近已访问结点 q 为空
  while(!(p==NULL && top==-1)) {
    while(p!=NULL) {
      S[++top]=p;
      p=p->lchild;             //向左搜索
    }
    p=S[top--];                //退栈
    if(q!=p->rchild) {
      S[++top]=p;              //再进栈
      p=p->rchild;             //转右子树
      q=NULL;                  //右子树尚未遍历，最近已访问结点 q 为空
    }
    else {
      cout<<p->data<<" "<<endl; //访问根
      q=p;p=NULL;
    }
  }
}
```

其中，q 记录左子树或右子树最近已访问过的结点（子树还未访问时 q 为空）。另外，再进栈过程可省，改为先检测栈顶（取栈顶），而不是先退栈。这里是为了与下述采用标志 flag 的算法结构上一致。

（2）为每个结点设个进栈标志 flag，并随结点一起进出栈。非形式算法如下：

```
p=根;
while(结点未遍历完) {
  while(p 非空) {push(p,flag=1);p=p->lchild;}  //向左搜索
  pop(p,flag);                                 //退栈
  if(flag==1) {push(p,flag=2);p=p->rchild;}    //再进栈，转右子树
  else {visit(p);p=NULL;}                      //访问根
}
```

其中访问根后，置 p=NULL 是为了下一步继续退栈（回退）。这里入栈元素是一个结构体：(指针, flag)，也可单独设标志栈，并与结点栈同步进出。算法略（略显烦琐）。

通过前述例子可见，工作栈在消除递归中的基本作用是提供一种控制机制。在非递归算法执行过程中的某些关键时刻，用栈顶元素来“引导”下一步操作的“走向”。为此必须提前将有关信息进栈保存。例如上面先序遍历的例子，工作栈保存的是各个结点的右指针，也就是该结点右子树的根指针。显然，这些根指针正是先根遍历递归算法中包含的一些递归调用的实参；这些实参在非递归算法中若不及时保存就会丢失。因此，递归算法中的调用与返回控制被工作栈的作用所取代，从而将递归算法转换成非递归算法。

有以下几点需要注意：

（1）图 5.19（a）计算菲波那契数列的递归过程可看成二叉树的后根遍历，据此也可写出利用栈的非递归算法。

（2）由于中序遍历和后序遍历时最先访问的不是根，所以它们相应的非递归算法中没有采用根预入栈技术（也不合适）。

（3）以上通过栈进行递归消除后算法变得复杂了，而时空性能基本未变（相当于把系统栈改为人工管理，可能节省一些不必要的操作，但时间复杂度数量级不变，具体分析略），

似乎意义不大。这里主要是介绍递归消除的原理。当然，对不允许递归的语言就有意义了。

（4）如果采用带双亲指针的二叉链表，递归消除时可不用栈，因为可由双亲指针进行回退。对下面将介绍的线索二叉树，遍历时也可不需要栈。另外，即使需要递归栈，栈空间也可不单独分配，而在原结点空间上就地进行：在向左下结点搜索时，每搜索到一个结点，先将其左孩子指针取出，然后在该处存入其双亲指针，这样在搜索结束后就可由双亲指针进行回退。当然，在回退的同时要恢复原孩子指针。这些内容就不细述了。

5.6　线索二叉树

当用二叉链表作为二叉树的存储结构时，因为每个结点中只有指向其左、右孩子结点的指针域，所以从任一结点出发只能直接找到该结点的左、右孩子，一般情况下无法直接找到该结点在某种遍历序列中的前趋和后继结点。但是，若在每个结点中增加两个指针域来存放遍历序列的前趋和后继信息，又将大大降低存储空间的利用率。注意到n个结点的二叉链表中含有n+1 个空指针域[①]，于是可利用这些空指针域来存放某种遍历次序下的前趋和后继结点的指针。这种附加的指针称为“线索”，加上线索的二叉链表称为线索链表，相应的二叉树称为**线索二叉树**（Threaded Binary Tree）。

这时，结点中的指针可能指的是孩子，也可能指的是线索，为了区分，可在结点中设置两个线索标志位，结点结构为：

lchild	ltag	data	rtag	rchild

其中：

$$\text{左线索标志ltag}=\begin{cases}0 & \text{lchild是指向结点的左孩子的指针}\\ 1 & \text{lchild是指向结点的前趋的左线索}\end{cases}$$

$$\text{右线索标志rtag}=\begin{cases}0 & \text{rchild是指向结点的右孩子的指针}\\ 1 & \text{rchild是指向结点的后继的右线索}\end{cases}$$

以图 5.22（a）所示的中序线索二叉树为例，它的线索链表见图 5.22（d），其中实线表示指针，虚线表示线索。其中结点 D 的左线索为空，表示它是中序序列的开始结点，没有前趋；结点 C 的右线索为空，表示它是中序序列的终端结点，没有后继。显然在线索二叉树中，一个结点为叶结点的充要条件为：它的左、右线索标志均是 1。

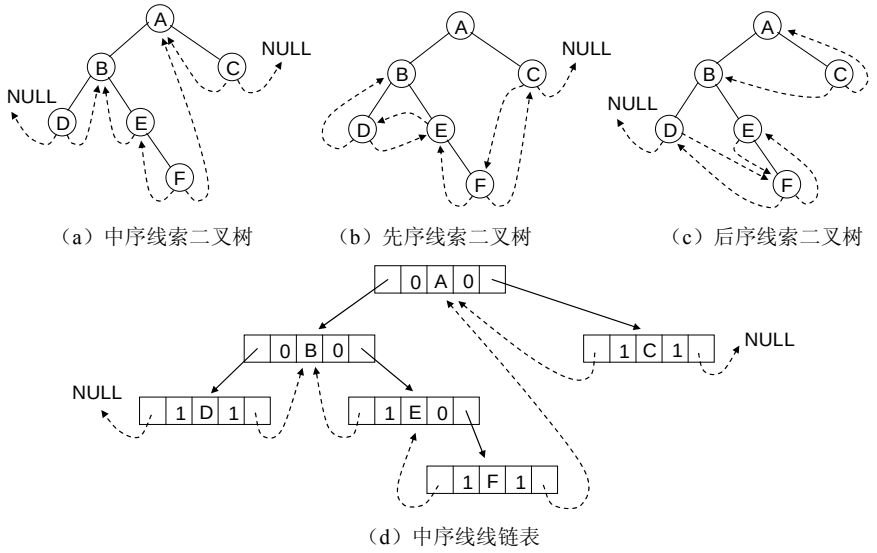
从图 5.22 中可发现如下特点：中序线索一般都是“向上”指的，即指向其祖先结点；而先序和后序线索就不一定（见图 5.22（b）和图 5.22（c）），可以向上指，也可以向下指，还可以同级指。

注意，中序线索链表中有两个空指针，但先序和后序链表中不一定只有一个空指针，也可能有两个空指针（可用只有两个结点的二叉树进行验证）。

有时为了某些运算的方便，也可在线索二叉链表上附加头结点：其左指针或右指针指向根结点，另一指针指向遍历序列的首结点或尾结点。进一步，如果遍历序列的首结点或尾结点的线索为空，还可将它指向头结点，形成某种“循环”链表。

---

① n 个结点共 2n 个指针域，但树中只有 n−1 条边，对应 n−1 个非空指针，故空指针数为 2n−(n−1)=n+1。



（a）中序线索二叉树　（b）先序线索二叉树　（c）后序线索二叉树

（d）中序线线链表

图 5.22　线索二叉树及其存储结构

以上两个线索标志也可合并为一个 tag，比如用 0、1 表示左线索的情况，用 3、4 表示右线索的情况，但不如左右分开处理简便、直观；另外，为了区分结点中的指针指的是孩子还是线索，也可不用标志位，而用负地址表示：将线索以负数的形式存放。这样如果指针域为正则指向孩子，否则表示线索。这个方法不改变结点的存储结构，也不耗费额外的空间，但指针类型毕竟不是整数类型，在使用时需要进行类型转换，这会增加运行时间。

将二叉树变为线索二叉树的过程称为**线索化**。按某种次序将二叉树线索化，只要按该次序遍历二叉树，在遍历过程中用线索取代空指针即可。为此，可用一个指针 pre 始终指向刚访问过的结点，用指针 p 指示当前正在访问的结点。显然，结点*pre 是结点*p 的前趋，而*p 是*pre 的后继。以中序线索化算法为例，该算法与中序遍历算法类似，区别仅在于访问根结点时所做的处理不同。在线索化算法中，访问当前根结点*p 所做的处理是：

（1）若结点*p 为空，则什么也不做。

（2）对结点*p 及其前趋结点*pre 进行相互处理：

① 若结点*p 的左子树为空，则令 p->lchild 为指向其中序前趋结点*pre 的左线索（即 p->lchild=pre），并置左线索标志（即 p->ltag=1）。

② 若结点*pre 存在（即 pre!=NULL），且其右子树为空，则令 pre->rchild 为指向其中序后继结点*p 的右线索（即 pre->rchild=p），并置右线索标志（即 pre->rtag=1）。

（3）将 pre 指向刚刚访问过的结点*p（即 pre=p）。这样，在下一次访问一个新结点*p 时，*pre 为其前趋结点。

下面给出线索链表的形式说明及中序线索化算法：

```
typedef struct node * pointer;      //pointer 类型用于表示树的一般结点
struct node {                       //结点结构
  datatype data;
  pointer lchild,rchild;
  int ltag,rtag;
```

```
};
typedef pointer bitree;              //bitree 类型用于表示树的根结点，即表示树
pointer pre=NULL;                    //全局量，初值为 NULL
void inthread(bitree t) {            //中序线索化
  pointer p;
  if(t==NULL) return;
  p=t;
  inthread(p->lchild);               //左子树线索化
  if(p->lchild==NULL) {p->lchild=pre;p->ltag=1;}   //对 p 建左线索
  if(pre!=NULL && pre->rchild==NULL) {pre->rchild=p;pre->rtag=1;}
                                                    //对 pre 建右线索
  pre=p;
  inthread(t->rchild);               //右子树线索化
}
```

类似地可得前序线索化和后序线索化算法（略）。

建立了线索链表之后，就可讨论线索二叉树上的运算了。下面以中序线索二叉树为例，介绍线索二叉树上两种比较简单又常用的运算。

1．查找某结点*p 的中序前趋和后继

查找结点*p 的中序后继结点分两种情形：

（1）若*p 的右线索标志为 1，则 p–>rchild 为右线索，直接指向*p 的中序后继结点。

（2）若*p 的右线索标志为 0，则*p 的中序后继要进行查找，它是*p 的右子树中序遍历的第一个结点，也就是右子树中“最左下”的结点。

查找结点*p 中序后继结点的算法如下：

```
pointer innext(bitree t) {
  pointer q;
  if(t->rtag==1) return(t->rchild);
  q=t->rchild;
  while(q->ltag==0) q=q->lchild;
  return q;
}
```

类似，查找结点*p 的中序前趋结点也分两种情形：

（1）若*p 的左线索标志为 1，则 p–>lchild 为左线索，直接指向*p 的中序前趋结点。

（2）若*p 的左线索标志为 0，则*p 的中序前趋要进行查找，它是*p 的左子树中序遍历的最后一个结点，也就是左子树中“最右下”的结点。

可见，在中序线索二叉树上找某点*p 的中序前趋和后继都比较方便。如果是非线索二叉树，则找某点的中序前趋和后继时，只能从根开始进行中序遍历。

然而，在前序线索链表中找某点的前序前趋、在后序线索链表中找某点的后序后继并不一定方便（前者找前序后继、后者找后序前趋方便）。这是因为要查找的前趋或后继可能就是给定点的双亲，或在双亲的另一棵子树中，如果线索标志为 0，就需要知道给定点的双亲，而找双亲一般要从根开始进行搜索（除非结点带双亲指针）。如果实际问题正好只需要其中比较方便的一个线索，则这类链表还是有使用价值的，这时只建一个线索就够了。有关内容就不细述了。

2．遍历线索二叉树

这里的遍历是指利用线索进行的遍历。遍历某种线索二叉树，可从该遍历次序下的开

始结点出发，反复找结点在该次序下的后继，直至终端结点（或从该次序下的终端结点出发，反复找结点在该次序下的前趋，直至开始结点）。以中序线索二叉树的中序遍历为例，算法如下：

```
intraver(bitree t) {                    //遍历中序线索二叉树
  pointer p;
  if(t==NULL) return;                   //空树
  p=t;
  while(p->ltag==0) p=p->lchild;//找中序序列的开始结点
  do {
    cout<<p->data<<endl;                //访问结点*p，假设为输出结点内容
    p=innext(p);                        //找*p 的中序后继结点
  } while(p!=NULL);
}
```

由于中序序列的终端结点的后继线索为空，所以 do 语句的终止条件是 p=NULL。显然该算法的时间复杂度仍为 O(n)，但常数因子比非线索遍历算法小，且无需设立递归栈。因此，若经常要对二叉树遍历、查找结点或查找结点在指定次序下的前趋和后继等，采用线索二叉树较好。

注意，对前序线索二叉树，找前序前趋不方便，所以遍历时应从开始结点（最左下结点）出发，按找前序后继的方式进行，否则并不比直接遍历非线索二叉树有优势。类似，对后序线索二叉树，找后序后继不方便，遍历时应从终端结点（根）开始，按找后序前趋的方式进行。

上面介绍的两种运算，线索树均优于非线索树。但如果要在线索二叉树中进行插入和删除运算就不方便了，因为这时除了修改指针外，还要修改相应的线索，运算量几乎与重新进行线索化相当。这部分内容就不介绍了。

5.7 树和森林

本节将建立树、森林和二叉树的对应关系，并讨论树的存储表示及其遍历。

5.7.1 树、森林与二叉树的转换

在树或森林与二叉树之间有一个自然的一一对应关系。任何一个森林或一棵树都可唯一地对应到一棵二叉树；反之，任何一棵二叉树也能唯一地对应到一个森林或一棵树。这样，对树或森林的一些操作就可利用二叉树来实现。

1．树、森林到二叉树的转换

树中每个结点可能有多个孩子，但二叉树中每个结点最多只能有两个孩子。要把树转换为二叉树，就必须找到一种结点与结点之间最多 1 对 2 的关系。注意到树中的每个结点最多只有一个最左边的孩子（长子）和一个右邻的兄弟，据此我们就能很自然地将树转换成二叉树，即将各结点的长子变成其左孩子，右兄弟变成其右孩子，如图 5.23（a）所示。转换后，二叉树中左分支上相邻的结点在原树中是父子关系；右分支上相邻的结点在原树

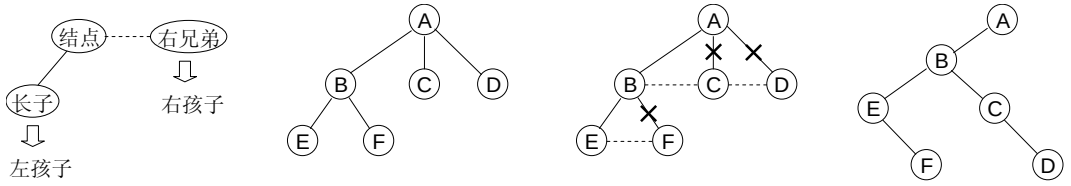
中是兄弟关系。由于根结点没有兄弟，所以树转化后二叉树的根结点没有右子树（为空）。

这里要指出，对于无序树，结点的孩子不分左右，于是谁是长子、谁是右兄弟以及整个转换就变得不确定或不唯一。为此我们约定，对一棵无序树，在转换时，就其当前形态按有序树进行处理。易见，这样转换的结果是唯一的。具体转换时可按如下步骤进行：

（1）在所有兄弟结点之间加一连线。

（2）对每个结点，除了保留与其长子的连线外，去掉该结点与其他孩子的连线。

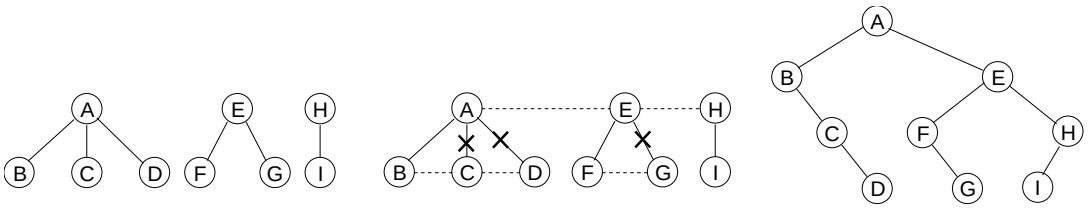
使用上述变换法，图 5.23（b）所示的树就变为图 5.23（c）的形式，它已是一棵二叉树，若将它按顺时针方向旋转约 45°就能更清楚地变为图 5.23（d）所示的二叉树。



（a）转换原理　（b）树　（c）转换到二叉树　（d）旋转结果

图 5.23　树转换为二叉树

将一个森林转换为二叉树的方法是，先将森林中的每一棵树变为二叉树，然后将各二叉树的根结点视为兄弟连在一起。注意，森林转化后二叉树的根结点有右子树（m>0 时）。例如在图 5.24 中，子图（b）是子图（a）的转换结果，子图（c）是子图（b）旋转后的二叉树。



（a）森林　（b）转换到二叉树　（c）旋转结果

图 5.24　森林转换为二叉树

可以给上述转换方法做如下形式定义：设 $F=\{T_1, T_2, \cdots, T_m\}$表示由树 T_1、T_2、…、T_m 组成的森林，则森林 F 对应的二叉树 B(F)为：

（1）若 F 为空（m=0），则 B(F)为空二叉树。

（2）若 F 非空（m>0），则 B(F)的根就是 T_1 的根；B(F)的左子树是由 T_1 去掉根后的子树森林 $F_1=\{T_{11}, T_{12}, \cdots, T_{1m'}\}$转换成的二叉树 $B(F_1)$；B(F)的右子树是由森林 $F'=\{T_2, T_3, \cdots, T_m\}$转换成的二叉树 B(F′)。由于左子树 $B(F_1)$和右子树 B(F′)本身又要由森林转换而来，故整个转换过程需要递归地进行。

2．二叉树到树、森林的转换

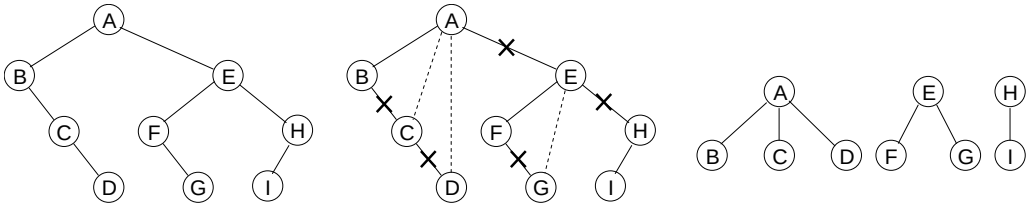
上述树、森林到二叉树的转换过程是可逆的，所以反过来便可将二叉树转换到树或森林，即将二叉树中结点的左孩子变为其长子，右孩子变为其右兄弟。具体过程可如下

进行：

（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、……，都与该结点的双亲用连线连起来。

（2）去掉原二叉树中所有双亲到右孩子的连线。

图 5.25（c）就是用这种方法将图 5.25（a）所示的二叉树处理后的结果，由于原二叉树有右子树，所以最后得到的是森林。



（a）二叉树　　（b）转换到森林　　（c）最后结果

图 5.25　二叉树转换为森林

5.7.2　树的存储

对树而言，一般不能采用顺序存储方式，只能采用链式结构，即除了存储各结点本身的数据外，还要用指针表示结点间的逻辑关系（父子关系）。下面介绍树的 3 种常用存储方法，每种方法中各结点的结构相同。

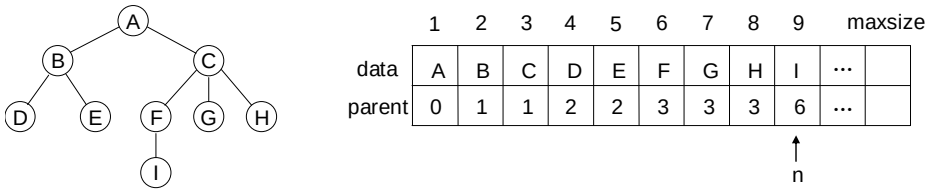
1．双亲链表表示法

该方法就是在存储结点信息的同时，为每个结点附设一个指向其双亲的指针 parent。尽管可用动态链表来实现这种表示法，然而用静态链表更为方便，其类型定义如下：

```
const int maxsize=100;          //结点数的最大值，假设为 100
typedef struct {
  datatype data;                //数据域
  int parent;                   //双亲域(静态指针域)
} pnode;
typedef struct {
  pnode nodes[maxsize+1];       //结点数组，0 号单元未用
  int n;                        //结点数
} ptree;                        //静态双亲链表类型
ptree P;                        //静态双亲链表
```

各个结点在结点数组中的存储顺序原则上是任意的，但一般将根放在开始位置（否则根需要查找），并且常常按层序编号的顺序存放。这里数组下标从 1 开始使用是为了运算上的方便：编号为 i（编号从 1 开始）的结点就直接存放到 i 号单元。但 0 号单元也可用来存放其他信息，如结点总数等（这时 ptree 类型中的数据域 n 可省略）。

在双亲链表中，若结点 i 的双亲是结点 j，则 nodes[i].parent=j，若结点 i 是根结点，则 nodes[i].parent=0。对图 5.26（a）所示的树 T，其双亲链表见图 5.26（b）。



（a）树 T　　（b）树 T 的双亲链表

图 5.26　树的双亲链表表示示意图

显然，在这种存储方式下，找指定结点的双亲或祖先（包括根）十分方便，只要沿着结点的双亲指针搜索即可。但若求指定结点的孩子或子孙，则可能要遍历整个数组。另外，求结点的兄弟也不方便。不过，若结点按层序编号顺序存储，则孩子的下标大于双亲的下标，兄弟结点间的下标从左向右递增，利用这些特点，可在遍历搜索中节省一定的时间。以求结点 i 的长子为例，算法如下：

```
int firstchild(ptree *P,int i) {
  int j;
  for(j=i+1;j<=P->n;j++)
    if(P->nodes[j].parent==i) return j;//第一个双亲为 P[i]的结点是 P[i]的长子
  return 0;                              //未找到
}
```

2．孩子链表表示法

树中各个结点的孩子个数一般不同，如果采用类似二叉链表的方法来表示结点及其孩子的关系，则每个结点设置多少个孩子指针域难以确定。若以树的度 k 来设置，则 n 个结点的树中，其空指针域的数目是 $kn-(n-1)=n(k-1)+1$，这将造成极大的空间浪费。若每个结点按其实际孩子数设置指针，就需要在结点内设置度数域 degree 以指出实际指针数，这样一来，各结点不等长也不同构，且插入、删除时结点大小还需重新调整，使用不便。

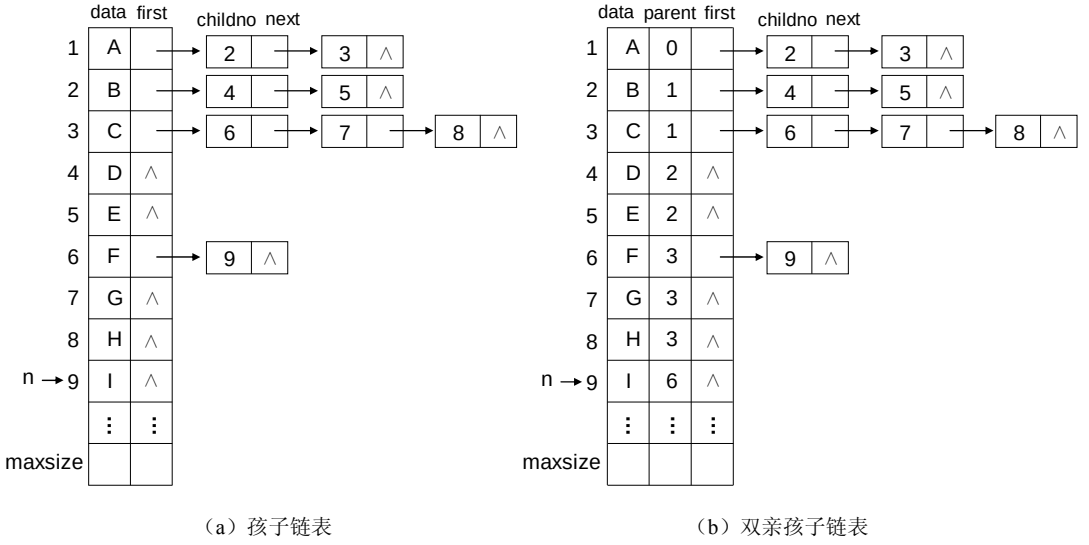
一种比较好的方法是，为树中每个结点各自建立一个孩子链表。每个孩子链表由其头指针唯一确定，为了便于查找，将所有头指针用一个数组集中存放，并与存放结点数据的结点数组合并成一个结构数组，称为表头数组，即数组中每个元素由两部分组成：数据域和指针域。其中，数据域存放结点的内容信息，指针域存放该结点的孩子链表的头指针。其类型定义如下：

```
const int maxsize=100;          //结点数的最大值，假设为 100
typedef struct cnode * pointer; //孩子链表结点指针类型
struct cnode {                  //孩子链结点结构
  int childno;                  //孩子结点的序号
  pointer next;
};
typedef struct {
  datatype data;
  pointer first;
} hnode;                        //表头结点类型
typedef struct {
  hnode nodes[maxsize+1];       //表头数组，0 号单元未用
  int n;                        //结点数
```

```
} childlink;                    //孩子链表类型
childlink T;                    //孩子链表
```

这里表头数组也从 1 号单元开始使用(0 号单元可用来存放其他信息,如根结点位置)。各结点信息在表头数组中一般也按层序编号的顺序存放,这时 1 号单元就是根结点;否则,宜在链表类型中增加一个根结点位置指针,因为在这种链表中找根结点不方便。

图 5.27(a)就是图 5.26(a)所示树的孩子链表表示。其中,若某结点 i 为叶子,则其孩子链表为空,即 nodes[i].first=NULL。对非叶子结点,nodes[i].first!=NULL,对应的孩子链表由结点 i 的所有孩子结点的序号构成。如 nodes[3].first 所指的链表有三个结点 6、7 和 8,表示结点 3 有三个孩子:结点 6、7 和 8。



(a)孩子链表　　(b)双亲孩子链表

图 5.27　树的孩子链表表示示意图

与双亲链表表示法相反,孩子链表表示便于实现涉及孩子和子孙的运算,但不便于实现与双亲有关的运算。因此可将这两种表示法结合起来,形成双亲孩子链表表示法。图 5.27(b)就是图 5.26(a)所示树的双亲孩子链表表示。

3. 孩子兄弟链表表示法

在介绍树转换为二叉树时,实际上利用和证明了这样一个事实:树中结点之间的关系,可用该结点与其最左边的孩子和右邻的兄弟之间的关系来描述。因此,在存储各结点信息的同时,附加两个指针域,分别指向该结点最左边的孩子和右邻的兄弟,就得到树的孩子兄弟链表表示。这其实相当于存储与树对应的二叉树。例如,图 5.26(a)所示树的孩子兄弟链表如图 5.28 所示。

这种存储结构的最大优点是,它和二叉树的二叉链表表示完全一样,只是结点的两个指针的含义有所不同。因此,可利用二叉树的算法来实现对树的操作。

以上方法中逻辑信息是显式表示的,其中指针的空间开销较大。树还有一些“压缩”型的储存方法,逻辑信息用占空间很小的一些相关信息(如左、右孩子标志,结点度数等),并结合特定的存放顺序(如先根、后根、层次顺序等)间接表示,在具体使用时再由这些相关信息求出逻辑信息,即以时间换空间,相对比较烦琐,这里从略。

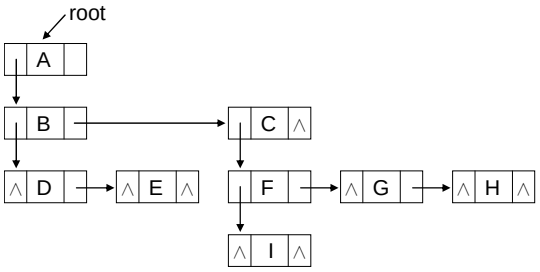


图 5.28　树的孩子兄弟链表表示示意图

5.7.3　树和森林的遍历

在树和森林中，每个结点可有两棵以上的子树，并且各结点子树的个数也不一定相同，因此不便讨论它们中根次序的遍历，但可研究先根和后根次序的遍历。设树 T 如图 5.29 所示，结点 R 为根，根的子树从左到右依次为 T_1、T_2、…、T_k。树的两种遍历方法定义为：

（1）前序遍历树 T

若树 T 非空，则：

① 访问根结点 R。

② 依次前序遍历根 R 的各子树 T_1、T_2、…、T_k。

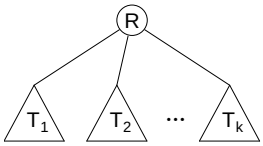


图 5.29　树 T

（2）后序遍历树 T

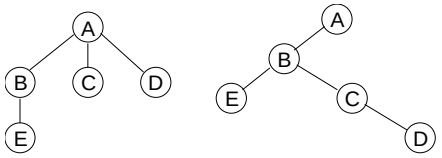
若树 T 非空，则：

① 依次后序遍历根 R 的各子树 T_1、T_2、…、T_k。

② 访问根结点 R。

例如，对图 5.30（a）所示的树进行前序遍历和后序遍历，得到的前序序列和后序序列分别是 ABECD 和 EBCDA。

值得注意的是：前序遍历一棵树恰好等价于前序遍历该树对应的二叉树，后序遍历一棵树恰好等价于中序遍历该树对应的二叉树。这可由树与二叉树的转换关系以及树与二叉树遍历的定义推得。例如在图 5.30 中，子图（b）是子图（a）对应的二叉树，它的前序序列和中序序列正是 ABECD 和 EBCDA。



（a）树　　（b）对应的二叉树

图 5.30　树和对应的二叉树

类似地，可得到森林的两种遍历方法：

（1）前序遍历森林

若森林非空，则：

① 访问森林中第一棵树的根结点。

② 前序遍历第一棵树中根结点的各子树所构成的森林。

③ 前序遍历除第一棵树外其他树构成的森林。

（2）后序遍历森林[①]

若森林非空，则：

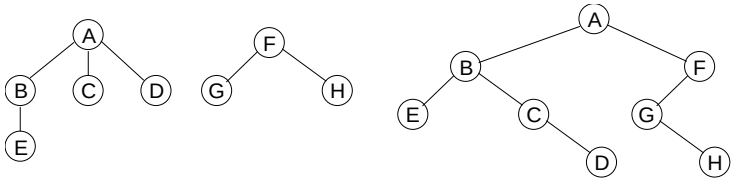
① 后序遍历森林中第一棵树的根结点的各子树所构成的森林。

② 访问第一棵树的根结点。

③ 后序遍历除第一棵树外其他树构成的森林。

简言之，前序遍历森林就是从左到右依次前序遍历森林中的每一棵树，后序遍历森林就是从左到右依次后序遍历森林中的每一棵树。

和遍历树类似，前序遍历森林等价于前序遍历该森林对应的二叉树，后序遍历森林等价于中序遍历该森林对应的二叉树。这同样可由森林与二叉树的转换关系以及森林与二叉树遍历的定义推得。例如，对图 5.31（a）所示的森林进行前序遍历和后序遍历，得到该森林的前序序列和后序序列分别为 ABECDFGH 和 EBCDAGHF。图 5.31（b）是该森林对应的二叉树，它的前序序列和中序序列也分别为 ABECDFGH 和 EBCDAGHF。



（a）森林　　（b）对应的二叉树

图 5.31　森林和对应的二叉树

由上述讨论可知，当用二叉链表作为树和森林的存储结构时，树和森林的前序遍历和后序遍历，可用二叉树的前序遍历和中序遍历算法来实现。另外，对树和森林也可以进行层序遍历。

（1）层序遍历树 T

若树 T 非空，则：

① 访问根结点 R。

② 若第 i（i≥1）层结点已访问，则访问第 i+1 层结点时，按从左到右的次序依次访问第 i+1 层上的结点。

显然，对树按层序遍历得到的遍历序列与结点的层序编号一致。事实上，对结点进行层序编号的过程本身就是一种层序遍历（访问结点的操作就是给它编个号）。

（2）层序遍历森林

这就是将森林中各树的同层结点依次输出。例如，图 5.31（a）所示森林的层序遍历序列为 AFBCDGHE。

注意，层序遍历森林不是依次对森林中的每一棵树进行层序遍历，而是所有树按同层结点依次输出。这不难理解：设想有一个虚的（总）根结点，使森林中的各树作为其子树，

① 有的文献称此为森林的中序遍历，即按第一棵树的根（D）、第一棵树的子树森林（A）、第一棵树外其他树构成的森林（B）三者访问的先后顺序来定义森林的遍历：先根 DAB、中根 ADB、后根 ABD。

则森林的层序遍历就是该树的层序遍历，也即各子树按同层结点依次输出。

易见，层序遍历树或森林，等价于沿右指针方向“层序”遍历对应的二叉树。这里，在对应二叉树中，结点与其右孩子，右孩子的右孩子，……，为同层结点，而左指针起到承上启下的作用。据此不难写出具体的层序遍历算法（略，见习题 5.21）。

5.8　哈夫曼树及其应用

树的应用非常之广，本节以**哈夫曼**（Huffman）树为例介绍树的应用。

5.8.1　最优二叉树（哈夫曼树）

在许多应用中，常常将树中结点赋予一个有某种意义的实数，称为该结点的权。结点到树根之间的路径长度与该结点权的乘积称为该结点的带权路径长度。树的（外部）**带权路径长度**（Weighted Path Length），定义为树中所有叶子结点的带权路径长度之和，通常记为：

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

其中，n 表示叶子结点的数目，w_i 和 l_i 分别表示叶结点 i 的权值和它到根之间的路径长度。在权为 w_1、w_2、…、w_n 的 n 个叶结点的所有二叉树中，带权路径长度 WPL 最小的二叉树称为**最优二叉树**或**哈夫曼树**。

在给定了叶子及其权后，如何构造最优二叉树呢？先看一个例子。

给定 4 个叶子结点 a、b、c 和 d，权分别为 6、3、4 和 8。我们可以构造如图 5.32 所示的 3 棵二叉树（还有多个）。它们的带权路径长度分别为：

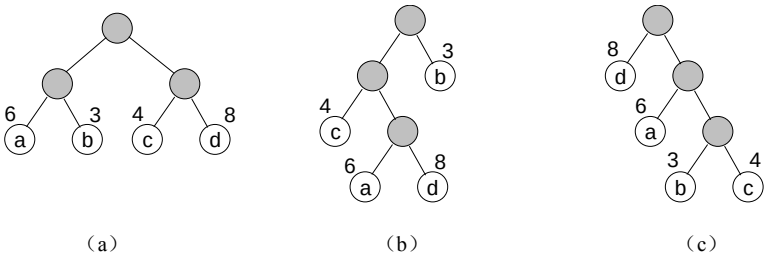


图 5.32　叶子相同的三棵二叉树

（a）WPL=6×2+3×2+4×2+8×2=42

（b）WPL=6×3+8×3+4×2+3×1=53

（c）WPL=3×3+4×3+6×2+8×1=41

其中树（a）为完全二叉树，但它的 WPL 并不是最小；最小的是树（c），下面将看到，它就是哈夫曼树。树（c）有个特点：权越大的叶子离根越近。这可从 WPL 的表达式来理解：如果权 w_i 较大，则希望路径 l_i 较小，这样 $w_i l_i$ 就会较小，从而有利于减小 WPL。不过

这种分析是不严格的，因为 WPL 考虑的是总体情况，而不是个别叶子带权路径的变长或变短，但一般情况下这个特点还是存在的。在此基础上，1952 年哈夫曼给出了一个求最优二叉树的精巧方法，称之为哈夫曼算法，步骤如下：

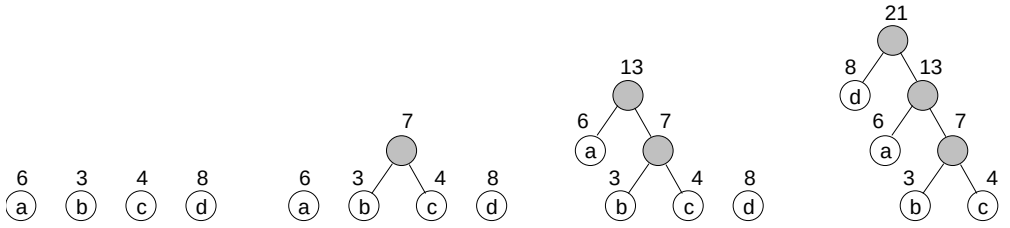
（1）首先，根据给定的 n 个权值 w_1、w_2、…、w_n 构造含有 n 棵二叉树的森林 $F=\{T_1, T_2, \cdots, T_n\}$，其中每棵二叉树 T_i 中只有一个权值为 w_i 的根结点，没有左右子树。

（2）在森林 F 中选出两棵根结点权值最小的树（当这样的树不止两棵时，可从中任选两棵），将这两棵树合并成一棵新的二叉树。这时会增加一个新的根结点，它的权取为原来两棵树的根的权值之和，而原来的两棵树就作为它的左右子树（谁左、谁右无关紧要）。

（3）对新的森林 F 重复步骤（2），直到森林 F 中只剩下一棵树为止。这棵树便是哈夫曼树。

按照这个算法，权越大的叶子合并的时机越晚，它离最终的根也就越近，正好符合上面的定性分析和观察结果。

由哈夫曼算法可知，哈夫曼树不一定唯一（但 WPL 是相同的，并且都为最小）。原因在于每次合并时，原两棵树谁左谁右，以及候选的两棵树有多个时选哪两个等。图 5.33 表示了用该算法从上述 4 个叶子构造哈夫曼树的过程，其中在合并时将根权值小的二叉树作为新根的左子树，其结果（d）就是图 5.32 中的树（c）。



（a）初始森林　（b）b 与 c 合并　（c）a 合并　（d）d 合并

图 5.33　哈夫曼树的构造过程

在哈夫曼算法中，初始森林共有 n 棵二叉树，每棵树中仅有一个结点，它们既是根，又是叶子。算法的第二步是将当前森林中的两棵根结点权值最小的二叉树合并成一棵新二叉树。每合并一次，森林中就减少一棵树。显然，要进行 n−1 次合并，才能使森林中二叉树由 n 棵减少到只剩一棵，即最终的哈夫曼树。另外，每合并一次都要产生一个新结点，合并 n−1 次共产生 n−1 个新结点。由此可知，最终求得的哈夫曼树中共有 n+(n−1)=2n−1 个结点，其中的叶结点就是初始森林中的 n 个孤立结点。

在哈夫曼树中，每个分支结点都是合并过程中产生的，它们的度为 2，所以树中没有度为 1 的分支结点，这类树通常称为**严格二叉树**[①]（Strictly Binary Tree）或**正则二叉树**（Proper Binary Tree）。实际上所有具有n个叶结点的严格二叉树都恰好有 2n−1 个结点。

我们用一个大小为 2n−1 的数组来存储哈夫曼树中的结点，其存储结构为：

```
const int n=20;                    //叶子结点数，假设为 20
```

① 有的文献称此为满二叉树（Full Binary Tree）；有的文献称此为完全二叉树（Complete Binary Tree）；也有其他称呼，如 2-tree、强二叉树等。

```
const int m=2*n-1;          //结点总数
typedef struct {
  float weight;
  int parent,lchild,rchild;
} nodetype;                 //结点类型
typedef nodetype hftree[m];//哈夫曼树类型，数组从 0 号单元开始使用
hftree T;                   //哈夫曼树向量
```

其中，每个结点包括 4 个域，weight 是结点的权值，lchild、rchild 分别为结点的左、右孩子在数组中的下标，叶结点的这两个指针值为−1；parent 是结点的双亲在数组中的下标。这里设置 parent 域不仅可使以后涉及双亲的运算方便，还可在合并时区分根结点和非根结点：若 parent 的值为−1，则该结点无双亲，即为根结点，尚未合并过。之所以要区分根结点与非根结点，是因为每次合并两棵二叉树时，要先在当前森林的所有结点中找两个权值最小的根结点，因此，有必要为每个结点设置一个标记以区分根结点和非根结点。

在上述存储结构上实现的哈夫曼算法可粗略地描述为：

（1）初始化：将初始森林的各根结点（叶子）的双亲和左、右孩子指针置为−1。

（2）输入叶子权：叶子在结点向量 T 的前 n 个分量中，构成初始森林的 n 个根结点。

（3）合并：对森林中的树进行 n−1 次合并，共产生 n−1 个新结点，依次放入结点向量 T 的第 i 个分量中（n≤i≤m−1）。每次合并的步骤是：

① 在当前森林的所有结点 T[j]（0≤j≤i−1）中，选取具有最小权值和次小权值的两个根结点，分别用 p1 和 p2 记住这两个根结点在结点向量 T 中的下标。

② 将根为 T[p1]和 T[p2]的两棵树合并，使其成为新结点 T[i]的左右孩子，得到一棵以新结点 T[i]为根的二叉树。同时修改 T[p1]和 T[p2]的双亲域，使其指向新结点 T[i]，这意味着它们在当前森林中已不再是根。T[p1]和 T[p2]的权值相加后，作为新结点 T[i]的权值。

求精后的哈夫曼算法如下：

```
void huffman(hftree T) {
  int i,j,p1,p2;
  float small1,small2;
  for(i=0;i<n;i++) {                //初始化
    T[i].parent=-1;
    T[i].lchild=T[i].rchild=-1;
  }
  for(i=0;i<n;i++)                  //输入 n 个叶子的权
    cin>>T[i].weight;
  for(i=n;i<m;i++) {                //进行 n-1 次合并，产生 n-1 个新结点
    p1=p2=-1;                       //此句可不要
    small1=small2=FLOAT_MAX;        //FLOAT_MAX 为 float 类型的最大值
    for(j=0;j<=i-1;j++) {           //找两个权值最小的根结点
      if(T[j].parent!=-1) continue; //不考虑已合并过的结点
      if(T[j].weight<small1) {      //修改最小权、次小权及其位置
        small2=small1;
        small1=T[j].weight;
        p2=p1;
        p1=j;
      }
      else if(T[j].weight<small2) { //修改次小权及位置
        small2=T[j].weight;
        p2=j;
```

```
        }
      }
      T[p1].parent=T[p2].parent=i;
      T[i].parent=-1;                    //新根
      T[i].lchild=p1;
      T[i].rchild=p2;
      T[i].weight=small1+small2;
    }
  }
```

注意，哈夫曼树虽然不一定唯一，但算法 huffman 得到的结果是唯一的，因为按该算法执行时，如果中途有两个以上的根权都最小，则它选中的是数组下标较小的两个；并且在合并时，权最小的根作左子树，权次小的根作右子树，故不会出现不确定的情况。

以前面 4 个权为 6、3、4、8 的叶子为例，用上述算法求哈夫曼树的过程见图 5.34。注意，图中只画出了 weight 域的变化情况。因为 n=4，故进行 3 次合并。合并过的结点用阴影表示，以后不再考虑；当前权值最小的两个根结点用下划线表示。

	0	1	2	3	4	5	6
初始森林	6	3	4	8			
第一次合并	6	3	4	8	7		
第二次合并	6	3	4	8	7	13	
第三次合并	6	3	4	8	7	13	21

图 5.34　哈夫曼算法的执行过程

5.8.2　哈夫曼编码与压缩

哈夫曼树的应用很多，本节介绍哈夫曼编码与压缩。

我们在使用计算机的时候，为了减少数据文件所占用的磁盘空间，或者为了提高网络数据传递的时间效率，经常对文件进行压缩以减少文件的大小，如常见的压缩格式有 ZIP、LHA、RAR 等。其实，利用哈夫曼算法，也能进行简单的文件压缩。

假设文件中有 n 个字符，则该文件大小就是 n 字节（byte），或 8n 位（bit），这是因为每个字符的 ASCII 编码为 8 位，最多可以表示 2^8=256 种字符。若实际使用的字符种类较少，则可不必用 8 位编码，比如只有 30 种时只需 5 位编码就可以了（2^5=32>30），这样可一定程度地节省文件长度。一般对 k 种字符编码需要$\lceil \log_2 k \rceil$位。以上各种字符的编码长度都相等，称为**等长码**。但是，文件中各个字符使用的频率是不等的，如英文中 E 和 T 的使用一般就比 Q、X、Z 等频繁几十倍。若采用不等长编码，让使用频率高的字符的编码缩短，就可使文件总长变短。注意，不等长编码后文件应以二进制形式存放。

然而，采用不等长编码时有可能产生多义性。例如，假设 00 表示 E，01 表示 T，0001 表示 W，则当取出的编码信息串是 0001 时，就无法确定原文是 ET 还是 W。产生该问题的原因是 E 的编码与 W 的编码的开始部分（前缀）相同。因此，不等长编码中要求任一字符的编码都不是其他字符编码的前缀，这种编码叫做**前缀（编）码**。显然，等长的 ASCII 码是前缀码。

假设组成文件的字符集是 D={d_1, d_2, …, d_n}，每个字符 d_i 在文件中出现的次数是 c_i，

对应的编码长度是 l_i，则文件总长为 $\sum_{i=1}^{n} c_i l_i$ 。因此，使文件总长最小就是使 $\sum_{i=1}^{n} c_i l_i$ 取最小值。然而，我们不可能对每个文件都去统计其中每个字符具体的出现次数 c_i，但通过对大量文件的统计分析，可以得到每个字符出现的概率 p_i，则 $\sum_{i=1}^{n} p_i l_i$ 表示平均码长。显然，平均码长越小，文件的平均总长就越短。例如，设字符集 D 及其概率分布 P 为：

D={a, b, c, d, e}

P={0.12, 0.40, 0.15, 0.08, 0.25}

在该字符集 D 上的三种不同的前缀编码如图 5.35 所示，其中编码 1、编码 2 和编码 3 的平均码长分别为 3、2.2 和 2.15。可以证明编码 3 就是上述给定概率分布下的**最优前缀码**（即平均码长 $\sum_{i=1}^{n} p_i l_i$ 最小的前缀码）。

字符	概率	编码1	编码2	编码3
a	0.12	000	000	1111
b	0.40	001	11	0
c	0.15	010	01	110
d	0.08	011	001	1110
e	0.25	100	10	10

图 5.35　三种前缀码

观察表达式 $\sum_{i=1}^{n} p_i l_i$ ，如果我们取 p_i 为叶结点的权，取编码长度 l_i 为叶结点的路径长度，则平均码长 $\sum_{i=1}^{n} p_i l_i$ 最小的问题就是一个带权路径长度最小的哈夫曼树的构造问题。于是可以这样求最优前缀码：用 d_1、d_2、…、d_n 作为叶结点，用 p_1、p_2、…、p_n 作叶结点的权，构造一棵哈夫曼树；然后将哈夫曼树中每个分支结点的左分支标 0，右分支标 1，把从根到每个叶子的路径上的标号连接起来，作为该叶子所代表的字符的编码。注意，每个字符 d_i 的编码长度对应叶子的路径长度 l_i。在哈夫曼树中，没有任何叶子是其他叶子的祖先，所以每个叶结点对应的编码不可能是其他叶结点编码的前缀，即上述编码得到的是前缀码。这个过程就是**哈夫曼编码**。据此编码就可对文件进行压缩，这个过程就不细述了。

例如，对图 5.35 给出的字符集及其概率分布，用哈夫曼算法构造的哈夫曼编码树及其对应的哈夫曼编码如图 5.36 所示。

在哈夫曼树的存储结构中，每个结点都存有其双亲指针，所以在求哈夫曼编码的过程中，可以从叶结点出发，向上回溯直到根结点。具体过程是：从叶子 T[i]出发，利用双亲指针找到其双亲 T[p]；再根据 T[p]的孩子指针可以知道 T[i]是 T[p]的左孩子还是右孩子，若是左孩子，则生成代码 0，否则生成代码 1；然后以 T[p]为出发点，继续上述过程，直到根结点。

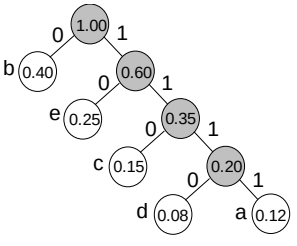
显然，这里生成的代码序列与所求编码次序相反，因此，可以将生成的代码按从后往前的次序依次存放在一个位串 bits 中。虽然各字符的编码长度不同，但最大不会超过 n，所以，bits 的大小可设为 n，然后用一个整型变量 start 来指示编码在位串 bits 中的起始位

置，其类型定义和编码算法如下：

```
typedef struct {
  char bits[n];                    //位串，存放编码
  char ch;                         //字符
  int start;                       //编码在位串中的起始位置
} nodetype;                        //编码结点类型
typedef nodetype codelist[n];      //编码表类型

void encode(codelist codes,hftree T) {
  int i,c,p,start;
  for(i=0;i<n;i++) {               //从叶结点出发向上回溯
    cin>>codes[i].ch;              //读入叶子字符
    start=n;
    c=i;
    p=T[i].parent;                 //取出双亲
    while(p!=-1) {
      start--;
      if(T[p].lchild==c) codes[i].bits[start]='0';//左子树编'0'
      else               codes[i].bits[start]='1';//右子树编'1'
      c=p;
      p=T[p].parent;
    }
    codes[i].start=start;
  }
}
```

以图 5.36（a）所示的哈夫曼树为例，上述算法求出的哈夫曼编码见图 5.37。



（a）哈夫曼编码树

a: 1 1 1 1
b: 0
c: 1 1 0
d: 1 1 1 0
e: 1 0

（b）哈夫曼编码

图 5.36　哈夫曼编码树及其编码

	bits						
	0	1	2	3	4	ch	start
0		1	1	1	1	a	1
1					0	b	4
2			1	1	0	c	2
3		1	1	1	0	d	1
4				1	0	e	3

图 5.37　哈夫曼编码的存储结构

哈夫曼树也可用来译码和解压缩。与编码过程相反，译码过程是从哈夫曼树的根结点出发，逐位读入编码信息，若为 0，则走向左孩子，否则走向右孩子，一旦到达叶结点 T[i] 便译出相应的字符 codes[i].ch。然后，重新从根结点出发继续译码，直到编码压缩文件结束。为简单起见，这里假设编码信息以十进制数字形式逐位输入，则译码算法如下：

```
void decode(codelist codes,hftree T) {
  int i,b,endflag;
  endflag=-1;                      //编码结束标志，假设取-1
  i=m-1;                           //从根结点开始搜索
  while(cin>>b,b!=endflag) {       //读入一个编码位，若不是结束标志则循环
    if(b==0) i=T[i].lchild;        //根据编码位走向左孩子或右孩子
    else     i=T[i].rchild;
    if(T[i].lchild==-1) {          //T[i]为叶子
```

```
        cout<<codes[i].ch;          //译码，输出字符
        i=m-1;                      //回到根结点开始下一次搜索
      }
    }
    if(i!=m-1) cout<<"编码有错！\n";    //编码读完，但结束点不是根，出错
}
```

由于哈夫曼树中没有度为 1 的结点，故上面算法中仅用 T[i].lchild=−1 来判定 T[i]是否为叶结点。另外，正确的结束点是译完最后一个字符后输入−1，此时搜索点为根结点（i=m–1）。

需要指出，哈夫曼编码/解码的主要意义在于最优编码，用于文件压缩则意义不大，因为有更高压缩率的方法（如 ZIP、RAR 等，它们压缩原理不同）。

5.8.3　分类与判定树

本节介绍哈夫曼树的另一个应用：描述快速分类过程。

分类是一种常用运算，其作用是将输入的数据按预定的标准划分成不同的种类。例如，一般来说，工厂生产的产品，质量有高有低，需要根据一定的质量指标对产品进行检测，根据检测的结果将产品划分成不同的等级，这就是一个分类问题。分类问题有时也称判定问题，其中的每次检测也称一次判断。

在分类过程中，对每一次判断，产生两个结果：要么成立（真），要么不成立（假），正好可用二叉树的两个分支来表示。每次判断后产生两个子类，如果某个子类不是所要的最终结果，再对子类继续进行判断，以划分出更细的子类，直到每个子类都对应唯一的一种分类结果。于是整个分类过程可以用一棵二叉树来描述，称之为分类问题的**判定树**。在判定树中，每个分支结点对应一次判断，每个叶子结点对应一种分类结果；根对应整个分类过程的第一次判断。

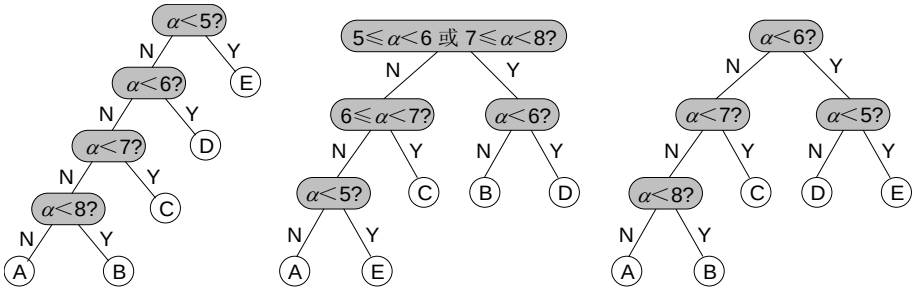
例如，假设某产品的等级标准见图 5.38。图 5.39（a）～图 5.39（c）就是该质量检测问题的三棵判定树，树上的 4 个分支结点对应于 4 次条件判断；树上的 5 个叶子结点对应于分类的 5 种不同的结果，即区分出的 5 个质量等级。

等级	E	D	C	B	A
检测值α	$\alpha<5$	$5\leqslant\alpha<6$	$6\leqslant\alpha<7$	$7\leqslant\alpha<8$	$8\leqslant\alpha$
百分比	0.15	0.2	0.35	0.2	0.1

图 5.38　质量等级标准

由于判定树描述的就是分类方法，因而由判定树很容易写出相应的分类算法。以图 5.39（a）的判定树 1 为例，分类算法如下：

```
char classify(float x) {
  if(x<5) return 'E';
  else if(x<6) return 'D';
  else if(x<7) return 'C';
  else if(x<8) return 'B';
  else return 'A';
}
```



（a）判定树 1　　（b）判定树 2　　（c）判定树 3

图 5.39　判定树示例

显然，在分类中，不同等级的产品所经过的条件判断次数一般是不同的，如按上述算法，E 类产品 1 次判断就可确定，D 类产品要判断 2 次等。另外，如果判定树不同，即使是同一类产品，分类时所经过的判断次数一般也不相同。例如，图 5.39 中的三棵判定树，除 D 类产品都需 2 次判断外，其他各类产品所需的判断次数就不完全相同了。

如果只对个别或少数产品进行分类，则采用该问题的任何一个判定树都可以。但实际中经常需要对大量的产品进行分类，如一个工厂一个月的某零件产品就可能成千上万，这时就要考虑算法的时间性能了。在分类中，主要的操作是条件判断，故以条件判断的次数为标准来分析算法的时间复杂度。

仍以上述产品质量分级问题为例。假设要分类的产品有 N 件，其中每类产品所占的比率已预测出来，见图 5.38 的第三行。设产品需要分成 n 类，每类所占的比率为 p_i（该类产品件数为 Np_i），对该类产品分类所需要的判断次数为 c_i 次，则总的判断次数为：

$$SUM = \sum_{i=1}^{n} Np_ic_i = N\sum_{i=1}^{n} p_ic_i$$

如果使上式达到最小值，则分类算法的时间性能最好。从上式看到，如果将产品所占的比率 p_i 看成是 n 个叶子的权，产品的判断次数 c_i 看成叶子的路径长度，则这个问题就是最优二叉树的构造问题。于是可利用哈夫曼算法构造哈夫曼树，从而得到最佳判定过程。具体方法是，以每个类别的概率为权构造哈夫曼树；在每次合并产生的新结点上加上适当的判定条件。图 5.39(b)就是这样的一棵判定树。

然而，图 5.39(b)虽然“判断次数”最少，但实际执行效果并不好。因为它的每次判断有些不是最基本的，如条件 6≤α<7，它实际包含 2 次基本比较：α>=6 && α<7，故总的“基本比较次数”不一定是最小的。

为此，需要对哈夫曼算法略作修改，使得树中的分支结点，即每次判断为一次基本比较。注意到对检测参数做基本比较时，以检测值为界，所划分出的两个区间（子类）在该参数空间上是“相邻”的（一分为二，两者紧邻，如区间 α<7 和区间 α≥7），故在哈夫曼算法的每次合并时，不能简单地找两个权最小的根，还要求它们对应的空间范围“相邻”，即合并的条件是：在相邻的两个根中找权值和最小的。

对图 5.38，最初权最小的两个根是 A 和 E，但它们不相邻（中间隔着 B、C、D 等区段），故不合并；相邻的权值和最小的两个根是 A 和 B，故对它们合并。A、B 合并后形成

区间 $\overline{BA}$，它和 C 相邻。第二次合并时找到的两个根是 $\overline{BA}$ 和 C，它们合并后形成区间 $\overline{CBA}$，和 D 相邻。第三次合并时找到的两个根是 E 和 D，它们合并后形成区间 $\overline{ED}$，和 $\overline{CBA}$ 相邻。最后是根 $\overline{ED}$ 和 $\overline{CBA}$ 合并，最终结果见图 5.39 的判定树 3。这时：

$$SUM = N\sum_{i=1}^{n} p_i c_i = N(0.15\times 2 + 0.2\times 2 + 0.35\times 2 + 0.2\times 3 + 0.1\times 3) = 2.3N$$

而采用判定树 1 时：

$$SUM' = N\sum_{i=1}^{n} p_i c_i = N(0.1\times 4 + 0.2\times 4 + 0.35\times 3 + 0.2\times 2 + 0.15\times 1) = 2.8N$$

可见，采用最优判定树可使比较次数下降 2.8N−2.3N=0.5N 次，节省了 17.9%，如果 N 很大，节省的比较次数和时间就相当可观了。

最后，与图 5.39 的判定树 3 对应的判定算法如下：

```
char classify3(float x) {
  if(x<6)
    if(x<5) return 'E';
    else    return 'D';
  else if(x<7) return 'C';
  else if(x<8) return 'B';
  else return 'A';
}
```

习　题　五

5.1　树的度是指树中结点的最大度数，但二叉树的度就一定为 2 吗？

5.2　已知某三叉树中度为 1、2、3 的结点数分别为 n_1、n_2、n_3，求其中的叶子结点数。

5.3　满 k 叉树有 n 个结点，则其中叶子结点有多少？

5.4　结点数为 n 高度也为 n 的二叉树是否只有左单枝和右单枝两种？

5.5　试回答下列问题：

（1）只有 3 个结点的树和二叉树各有几种不同的形态？各有多少棵不同的树？

（2）n 个结点的树和二叉树各有几种不同的形态？

5.6　找出所有满足下面条件的二叉树：

（1）先序序列和中序序列相同。

（2）后序序列和中序序列相同。

（3）先序序列和后序序列相同。

（4）先序、中序、后序序列都相同。

（5）先序序列和后序序列相反。

5.7　试回答下列问题：

（1）中序序列的最后一个结点是否就是先序序列的最后一个结点？

（2）怎样输出逆序的后根序列？

5.8　证明：

（1）完全二叉树的叶子数为 $n_0=n-\lfloor n/2\rfloor=\lceil n/2\rceil$。

（2）严格二叉树的叶子数为 $n_0=(n+1)/2$。

即这两种二叉树中叶子和非叶子结点各有一半左右。

5.9 证明：

（1）含 n 个叶子的完全二叉树的结点总数为 2n−1 或 2n。

（2）含 n 个叶子的完全二叉树的深度为$\lceil \log_2 n\rceil+1$ 或$\lfloor \log_2 n\rfloor+2$（它们不一定相等）。

（3）含 n 个叶子的二叉树的深度至少为$\lceil \log_2 n\rceil+1$。

5.10 如果某完全二叉树的叶子数正好为 2 的幂次，如 $n=2^{k-1}$，则该完全二叉树的深度就是 k 吗？

5.11 证明：对非空树，$n=\sum_{i=1}^{n} D(i)+1$，即结点数=所有结点的度数之和+1。

5.12 证明：若森林中有 n 个分支结点，则对应二叉树中有 n+1 个结点没有右孩子。

5.13 已知二叉树的层序序列为 ABCDEFGHIJ，中序序列为 DBGEHJACIF，请画出该二叉树。

5.14 已知某森林的先根序列为 ABCDEFG，后根序列为 BCAFEGD，请画出该森林。

5.15 线索二叉链表就是用结点的空指针域来存放某种遍历的前趋和后继线索，是否线索二叉链表中就没有空指针了？如果有，有几个空指针？

5.16 试编写递归算法，在二叉链表上实现以下功能：

（1）求二叉树的高度。

（2）求二叉树中度为 1 的结点数。

（3）查找值为 x 的结点，若找到则返回该结点的地址；否则返回空指针。

5.17 试编写非递归算法，在二叉链表上实现以下功能：

（1）将二叉树各结点的左右子树交换。

（2）求二叉树中叶子结点数。

5.18 试编写算法，在二叉链表上实现以下功能：

（1）求根*t 到任一给定结点*s 的路径（或，求任一给定结点*s 的祖先）。

（2）求任意两点*p 和*q 的共同祖先。

5.19 试编写算法实现以下功能：

（1）由二叉树的顺序存储结构 A[1..n]求叶子结点数。

（2）由二叉树的顺序存储结构 A[1..n]建立相应的二叉链表。

（3）由二叉链表建立相应的顺序存储结构 A[1..n]。

5.20 试编写算法，在二叉链表上实现以下功能：

（1）判断二叉树是否为完全二叉树。

（2）*求二叉树的宽度。

（3）*求二叉树的高度、每个结点的层数。

5.21 假设森林（或树）用对应的二叉链表存储，试编写以下算法：

（1）求叶子数、高度。

（2）层序遍历。

5.22　能否用二叉树表示父子、兄弟、夫妻三种关系？

5.23　试编写一个将百分制转换为五分制的算法，要求平均比较次数尽可能少。假定学生成绩分布如下：

等级	E	D	C	B	A
分数	0～59	60～69	70～79	80～89	90～100
百分比	0.05	0.15	0.40	0.30	0.10

5.24　已知两个各有 m 和 n 个记录的有序文件可在 O(m+n)的移动次数内合并为一个有 m+n 个记录的有序文件。现有 5 个文件要合并为一个大文件，其中记录数分别有 20、30、10、5 和 30 个。试给出记录移动次数最少的合并步骤。