第 5 章 树

树是一种重要的、应用广泛的非线性数据结构。树结构模仿自然界的树，我们日常生活中的许多问题可以用树形结构描述，比如：家族成员关系，事实上包括各种生物的遗传关系，如图 5-1 所示。行政组织机构，大到一个国家，小到一个企业、学校的组织结构，如图 5-2 所示。此外像各种物质的分类，教科书的目录等都表现为树形结构。

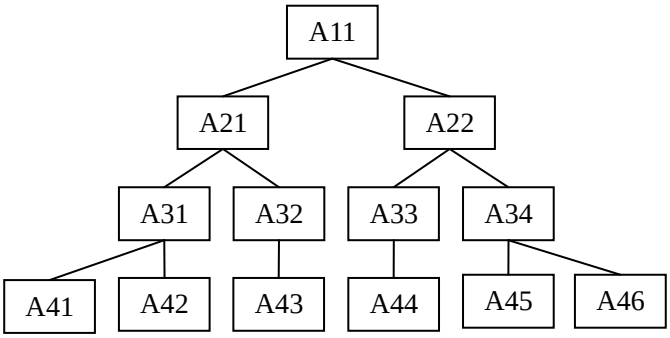


图 5-1 家族成员关系示意图

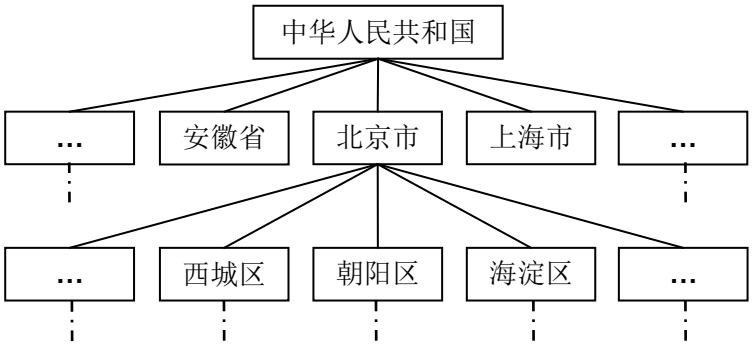


图 5-2 行政组织机构示意图

树结构在计算机系统中的应用案例也是随处可见，比如：磁盘文件的组织，如图 5-3 所示。可视化软件中树经常被用作层次数据的一种可视化表现形式和用户交互手段，许多语言提供 TreeView 组件来实现这个功能，如图 5-4 所示。可视化软件中的菜单结构。Internet 中的域名系统 DNS。软件开发中的功能结构图。编译器中表示源程序的语法结构等。

图 5-3 磁盘文件目录结构示意图

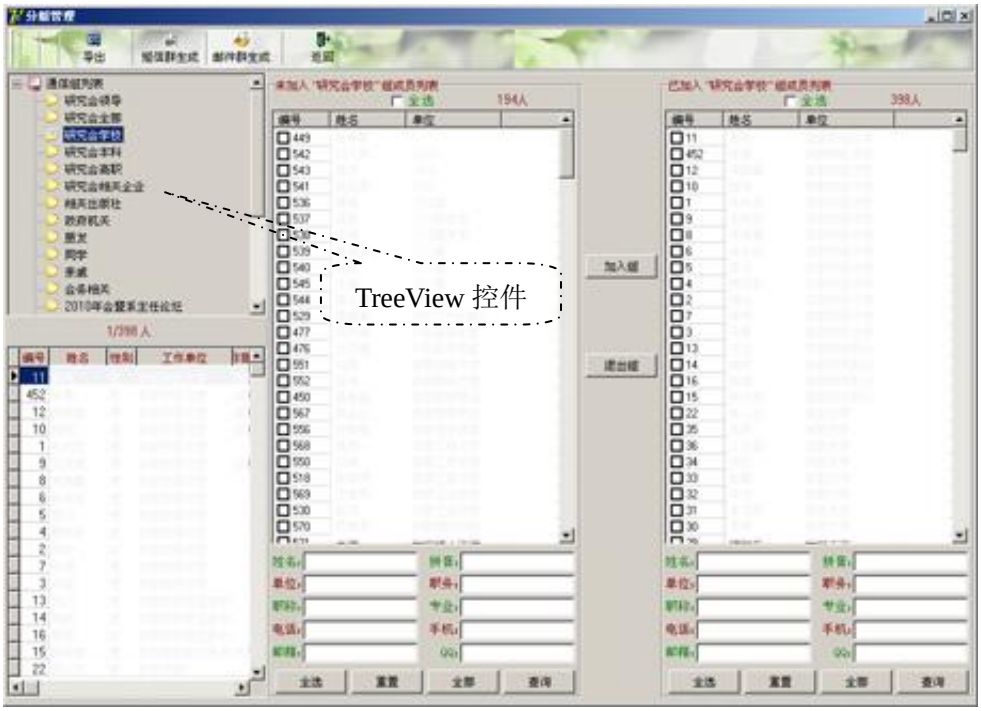


图 5-4 数据的树形表现和交互示意图

这种结构形式的共同特点是具有明显的层次特点，并且其中的每个元素最多只有一个前驱（或父辈），但可能有多个后继（或后代），都可抽象表示为本章的树形结构。

树形结构（包括树和二叉树）是一种非常重要的结构。由于树形结构中的各子结构与整个结构具有相似的特性，因而其算法大多采用递归形式，这对许多初学者来说是一个难点。本章系统介绍树结构的基本概念和术语，二叉树的基本概念、性质和存储结构，重点介绍二叉树的遍历这一基本运算及其应用，介绍线索二叉树的有关知识。在此基础上介绍树和森林的有关实现。最后结合哈夫曼树介绍二叉树的应用。

5.1 树的概念和基本运算

5.1.1 树的定义

树（Tree）是 n（n>0）个结点构成的有限集合。对树 T：

①有且仅有一个结点叫**根**，

②除根结点外其余结点可划分为 m 个互不相交的子集 T1,T2……Tm（m≥0），并且这 m 个子集每个子集本身又构成一棵树，称为 T 的**子树**（SubTree）。

这个树结构的定义是一个递归的定义，即在树的定义中又用到树的定义，这是由树结构的特点所决定的——每棵子树和树具有相同的结构组织形式。和许多教材类似，本书也采用递归方式描述。

需要说明一点，此处有关树的定义没有给出空树的概念，即树中结点数目至少为 1，这与大多数教材一致。也可能在部分教材中有空树的概念，因此提醒读者注意。

为了讲解和分析树结构，我们需要用形式化的方式来表示树结构，常见树结构的表示形式有（注意不是存储表示）：

（1）图形表示法：

用结点表示数据元素，用边连接相关的上下层结点构成图形。这种表示形式形象、直观、层次分明，在数据结构中常被使用。

如图 5-5 所示为一棵树的图形表示示例，其中每个结点用一个椭圆表示，其元素值标注在圆圈内。

（2）广义表表示法。如图 5-5 所示的树的广义表形式为（A,(B,(E),(F,(J)),(G)),(C),(D,(H,(K)),(I))）。

（3）嵌套集合表示法，类似于地图表示形式。很显然，这种表示形式难以清晰地表示层次较多的树结构。

（4）凹入表表示法（类似于书本中的目录形式）。

显然，图形表示形式最为直观、清晰，因此，在各种教材中都将此作为主要的表示形式。

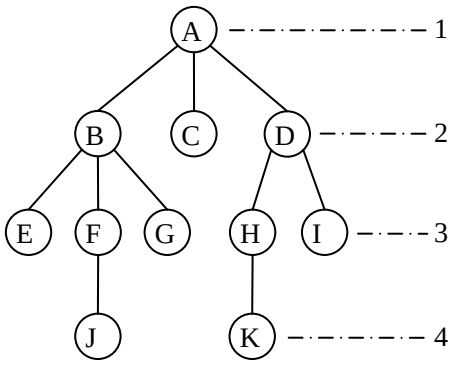


图 5-5 树结构的图形表示示意图

5.1.2 树的基本概念和术语

（1）结点（node）

也称为节点，表示数据元素。在链式存储结构中要附加相关的指针。

（2）结点的度（degree）

结点的子树数目，或分支数。例图 5-5 中，结点 A 和 B 的度为 3，D 的度为 2，F 和 H 的度为 1，C、E、G、I、J 和 K 的度为 0。

（3）树的度

树内各结点的度的最大值。例图 5-5 中，结点度的最大值为 3，则此树的度即为 3。

（4）叶子结点

度为 0 的结点叫叶子结点（树叶），或终端结点。例图 5-5 中的结点 C、E、G、I、J 和 K。

（5）分支结点

也叫非终端结点，指度不为 0 的结点，或具有子树的结点。除了叶结点外都是分支结点。例图 5-5 中的 A、B、D、F 和 H 结点。

（6）结点的层次（level）

规定根结点的层次为 1；其它结点的层次等于父结点层次加 1。结点的层次也叫结点的深度。注意有些教材规定根结点的层次为 0，请读者阅读时注意。例图 5-5 中，结点 A 的层次为 1，B、C 和 D 层次为 2，E、F 等层次为 3，J 和 K 层次为 4。

（7）树的高度（height）

树中各结点的最大层次数，也称为树的深度（depth），或树的层次。例图 5-5 中，树中结点的最大层次数为 4，所以树的高度为 4。

（8）孩子结点（child）

当前结点所有子树的根结点，简称子结点。用图论的术语描述即当前结点下一层与当前结点有边相连的结点。叶子结点没有孩子结点。例图 A 的孩子结点有 B、C 和 D，B 的子节点有 E、F 和 G。C 等叶子结点没有子结点。

（9）后裔结点（descendant）

当前结点作为根结点，其子树上的所有结点，都是当前结点的后裔结点，也称子孙结点。例图 5-5 中，除 A 外其它所有结点都是 A 的后裔，H、I 和 K 是 C 的后裔结点。

（10）双亲结点（parent）

与当前结点有边直接相连的上一层结点，也叫父结点。父、子结点关系是相互对应的。树中根结点没有父结点，其它结点有且仅有一个父结点。例图 5-5 中，结点 A 没有父结点，B、C 和 D 的父节点为 A，K 的父结点为 H。

（11）祖先结点（ancestor）

从根结点有路径到达当前结点，路径经过的所有结点，都是当前结点的祖先结点，或称先驱结点。例图 5-5 中，A、B 和 F 都是结点 I 的祖先，A、D 和 H 都是结点 K 的祖先。

（12）兄弟结点(Sibling)

双亲结点（父节点）相同的所有结点互称为兄弟结点。例图 5-5 中，B、C 和 D 是兄弟

结点，E、F 和 G 是兄弟结点，H 和 I 是兄弟结点。

（13）堂兄弟结点

双亲结点在同一层次（深度相同）的结点，互为堂兄弟，或同一层次上的结点互为堂兄弟结点。例图 5-5 中，B、C 和 D 也是堂兄弟结点，E、F、G、H 和 I 互为堂兄弟结点，J 和 K 互为堂兄弟结点。

（14）有序树和无序树

同一结点的所有子树，从左至右规定次序叫有序树，若结点的子树不分先后次序叫做无序树。

（15）森林（forest）

m (m≧0) 棵不相交树的集合。

树结构的特点：与线性表及其他结构相比，树结构有明显的差异：每个结点至多有一个直接前驱（即父结点），但却可以有多个后继结点（即孩子结点）。

5.1.3 树的基本操作

对树（包括森林）可执行如下的基本运算：

（1）初始化树：InitialTree(T)

建立树或森林 T 的初始结构。

（2）遍历树：Traverse(T)

按规定次序访问树中每个结点一次且仅一次。类似线性表中的搜索操作。

（3）插入子树：InsertTree(T,S)

将以 S 为根的子树作为 T 的第一个子树插入到树中。

（4）插入兄弟结点：InsertSibling(T, S)

将以结点 S 为根的树作为 T 的兄弟子树插入到树中。

（5）删除子树：DeleteTree(T)

删除树 T 中指定的子树。

（6）查询根结点：RootOf(T)

查询结点 T 所在树的根结点。

（7）查询父结点：FatherOf(T)

查询结点 T 的父结点。

（8）查询孩子结点：ChildOf(T)

查询结点 T 的所有或某个孩子结点。

（9）查询兄弟结点：SiblingOf(T)

查询结点 T 的所有或某个兄弟结点。

（10）求树的高度：TreeHeight(T)

返回树的高度。

树的运算中**遍历操作**是最基本也是最重要的一种运算，其它的很多运算都是借助遍历操作来完成的。

这里列举的只是一些常见的基本运算，还有一些基本运算，这里不再细述，在具体遇见时再进行介绍。

由于树和森林的存储结构涉及到后面将要介绍的二叉树结构，因此，有关树和森林的存

储结构及运算将在二叉树之后再作介绍。

5.2 二叉树

二叉树是一种特殊类型的树结构，是树结构的一个重要形式，也是最常用的树结构。二叉树的存储和运算算法较一般的树简单和直观，而一般的树结构又可转换为二叉树形式，因此可借助二叉树的存储结构和运算来实现树结构的运算。由此可知，二叉树是本章及整个课程的重点，理解其概念、性质及存储结构，并熟练写出有关算法是最基本的要求。

5.2.1 二叉树的基本概念

1. 二叉树的定义

二叉树（Binary Tree）T 是 n 个结点的有限集合，其中 n≥0，当 n=0 时，为**空树**，否则：

（1）有且仅有一个结点为**根结点**；

（2）其余结点划分为**两个**互不相交的子集 TL、TR，并且 TL、TR 分别构成一棵二叉树，叫作 T 的**左子树**和**右子树**。

二叉树的定义也是递归的。图 5-6 是一棵二叉树的实例。

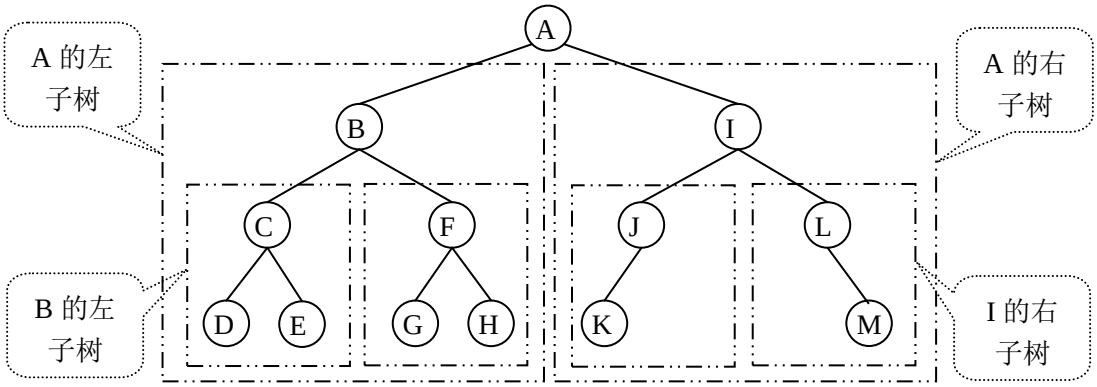
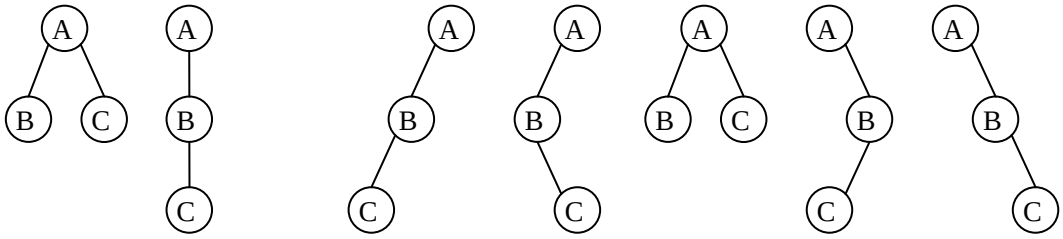


图 5-6 二叉树结构示意图

2. 二叉树的特点

（1）二叉树中每个结点最多只能有两棵子树（两个分支、两个子结点）。即二叉树中结点的度数只有三种情况：0 度叶子结点，1 度或 2 度结点。一般的树就没有这个限制，一个结点可有多个子结点（多棵子树）。

（2）二叉树是有序树，其子树区分为左子树、右子树，即使只有一棵子树也要区分是左子树，还是右子树，这是初学者容易忽视的问题。一般的树结构是无序树，一个结点的所有子树没有次序之分。对二叉树，相同的结点，结点的安放位置不同就会构成不同形态的二叉树。如图 5-7 所示，三个结点可以构成 2 棵不同形态的树；但可以构成 5 棵不同形态的二叉树。



(a) 3 个结点构成树的形态　　(b) 3 个结点构成二叉树的形态

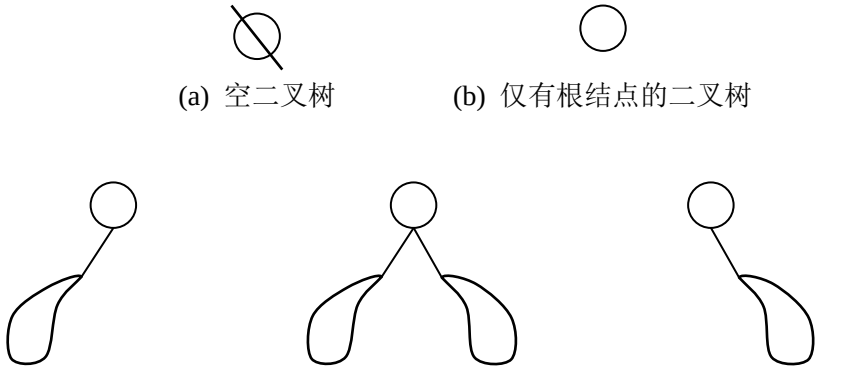
图 5-7　3 个结点构成树和二叉树的形态对比

由此可知，二叉树与树本质上是完全不同的两种结构。

树中有关层次、度等概念可引用到二叉树中，在此不再细述。

2. 二叉树的五种形态

根据二叉树的定义和树中结点的多少，我们可以归纳出二叉树的所有可能形态，总共五种，如图 5-8 所示：



(a) 空二叉树　　(b) 仅有根结点的二叉树

(c) 只有左子树　　(d) 左右子树均非空　　(e) 只有右子树

图 5-8　二叉树的五种形态

理解和熟悉二叉树的这些形态，有利于二叉树相关运算算法的分析和设计。

5.2.2 二叉树的性质

二叉树的性质是二叉树的重要内容，理解二叉树的性质有助于二叉树有关内容的学习。下面介绍二叉树的五个重要性质。对于其中较简单直观的性质，没有给出证明。

性质 1：在二叉树的第 i 层上的结点数≤2^{i-1}（i>0）。

性质 2：深度为 k 的二叉树的结点数≤2^k-1（k>0）。

性质 3：对任一棵非空的二叉树 T，如果其叶子数为 n_0，度为 2 的结点数为 n_2，则有下面的关系式成立：$n_0=n_2+1$。

【证明】设 T 的总结点数为 n，度为 1 的结点数为 n_1，则 T 的结点数满足下面关系式：

$$n=n_0+n_1+n_2 \qquad (a)$$

下面还需要再从 T 的分支数来讨论：一方面，从叶子结点往树根方向看，在这 n 个结点中，除根以外，每个结点有一个分支进入，因此其总分支数为 n-1。另一方面，从根结点往树叶方向看，度为 2 的结点发出 2 个分支，度为 1 的结点发出 1 个分支，则分支总数为 n_1+2n_2，因而有下面关系式成立：

$$n-1=n_1+2n_2 \qquad (b)$$

综合(a)和(b)两式得：$n_0=n_2+1$。 得证。

性质 1 说明了每层结点数目的上限；性质 2 则指出了给定层数的二叉树中的结点数目的上限；性质 3 描述了二叉树中叶子结点数与度为 2 的结点数之间的关系。

下面要讨论的性质 4 涉及到满二叉树和完全二叉树这两种特殊而又重要的二叉树。

满二叉树：所谓**满二叉树**是指每层都有最大数目结点的二叉树，即高度为 k 的满二叉树中有 2^k-1 个结点。

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

可以对满二叉树中结点进行编号。从根结点开始，编号为 1，按层自上而下，从左至右进行连续编号。这种编号对下面讨论的完全二叉树很有用。

完全二叉树：是指在满二叉树的最下层**从右到左连续地删除**若干个结点所得到的二叉树。

一棵深度为 k，结点数为 n 的完全二叉树，一棵深度为 k 的满二叉树，按上述约定同时进行编号，则这两棵树在编号 1 到 n 之间的结点一一对应。

完全二叉树叶结点只可能出现在最深的 2 层上；最下层结点一定是从左往右先放置左孩子，再放右孩子的方式依次开始放置的；若某个结点没有左孩子，则其一定没有右孩子；只有最深 2 层结点的度可能小于 2；最多只有一个结点的度为 1。满二叉树一定是完全二叉树；反之，完全二叉树则不一定是满二叉树。

例如高度为 4 的满二叉树有 $2^4-1=15$ 个结点，如图 5-9 所示。图 5-10 所示的二叉树则是一棵有 12 个结点，高度为 4 的完全二叉树。

性质 4：有 n 个结点的完全二叉树(n>0)的高度（深度）为 $\lfloor \log_2 n \rfloor+1$。

性质 4 给出了给定结点数的完全二叉树的高度的求解公式，即有 n 个结点的完全二叉树的高度为 $\lfloor \log_2 n \rfloor+1$。

【证明】设此完全二叉树的高度为 k，结点数为 n。则此树从 k-1 层到根结点必构成一棵高度为 k-1 的满二叉树，结点数为 2^{k-1}-1。必有：

$$n>2^{k-1}-1 \qquad (a)$$

根据性质 2，有：

$$n\leqq 2^k-1 \qquad (b)$$

由(a)和(b)两式得：$2^{k-1}-1<n\leqq 2^k-1$ (c)

由(c)式可得：$2^{k-1}\leqq n<2^k$ (d)

对(d)式取对数得：$k-1<=\log_2 n<k$ (e)

又 k 为整数，所以：

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

原问题得证。

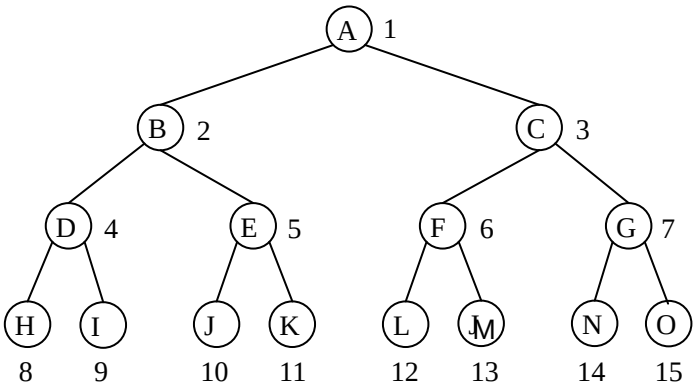


图 5-9 高度为 4 的满二叉树示意图

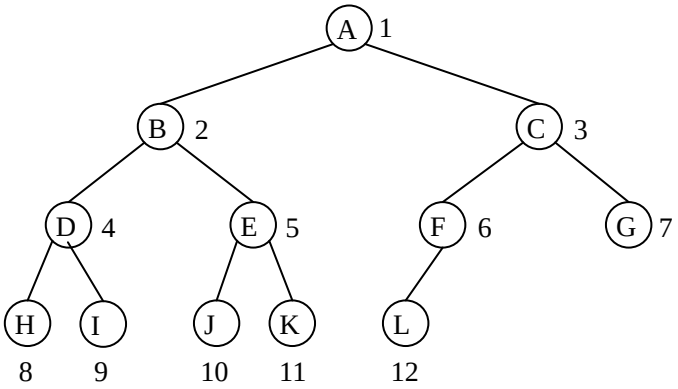


图 5-10 高度为 4 的完全二叉树示意图

性质 5：在编号的完全二叉树中，当前结点的编号为 i，则有：

（1）如果 i 结点的左孩子存在，其编号必为 2i；

（2）如果 i 结点的右孩子存在，其编号必为 2i+1；

（3）如果 i 结点的父结点存在，其编号比为$\lfloor i/2 \rfloor$。

这个性质证明过程相对复杂，这里不做证明。这一关系用图形形式表示如图 5-11 所示。

性质 4 和性质 5 是完全二叉树的两个重要特性。性质 5 在后面讨论的二叉树顺序存储结构以及在排序算法中都要用到。

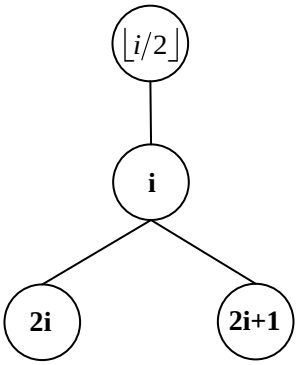


图 5-11 完全二叉树结点编号之间的关系

【例 5.1】已知完全二叉树有 100 个结点，则该二叉树有多少个叶子结点？

【解】本题有多种求解方法，例如，根据性质 3 来求解，或结合性质 1 和 2 来求解。如果用性质 5，则可以更方便地求解，下面讨论其求解过程。

若认为每个结点均已编号，则最大的编号为 100，其父结点编号为 50（见图 5-12），从 51 到 100 均为叶子，因此叶子数为 100-50=50。

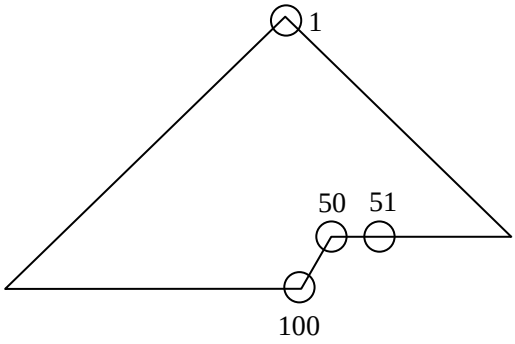


图 5-12 例 4.1 求解示意图

5.2.3 二叉树的存储结构

与线性结构类似，二叉树也可采用顺序存储方式和链式存储方式（即二叉链表存储方式）两类存储结构。下面分别讨论这两种存储方式。

存储一个结构时，不仅需要存储元素的值，同时还要能体现出结点之间的关系，否则并没有意义。对二叉树来说，选择的存储结构就是既要存储结点的值，又要能反映结点之间的层次关系、父子关系及兄弟关系。

1. 顺序存储结构

（1）完全二叉树的顺序存储

二叉树的顺序存储结构也是用数组存储数据元素（结点的值），但是如果简单地将二叉

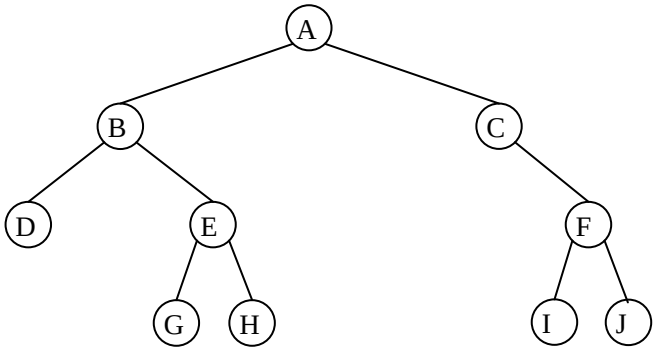
树所有结点的值“挤”在数组的前 n 个单元中，便不能体现出结点间的相互关系。但对完全二叉树和满二叉树，我们前面介绍了一种结点编号的方式，如果以元素编号对应数组下标，将元素存放到数组下标与编号相同的对应单元，这样利用性质 5，我们就可以根据元素对应的数组下标（即元素编号）来求其父结点、左孩子和右孩子结点的位置（数组下标）。假设当前元素数组下标为 i，若父节点存在，存放的数组单元为 i/2；若左孩子存在，存放的数组单元为 2i；右孩子在 2i+1 单元。**注意**：数组下标从 0 开始，我们可以保留数组的 0 单元不用，这样使元素在数组中的下标与编号完全对应相同。例图 5-10 所示完全二叉树的顺序存储结构如图 5-13 所示。

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

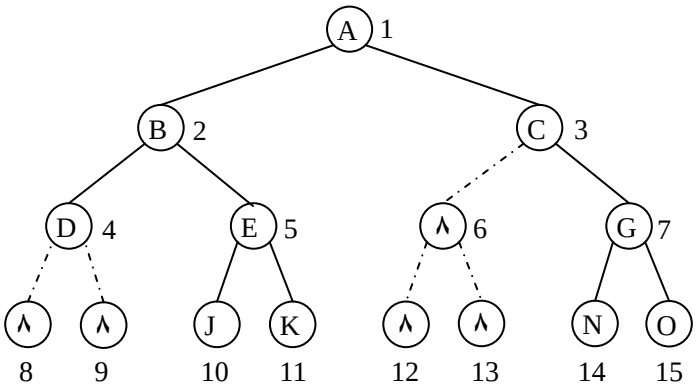
图 5-13 完全二叉树顺序存储结构示意图

（2）一般二叉树的顺序存储

一般二叉树没有完全二叉树和满二叉树上述的编号和数组下标的对应关系，怎样实现顺序存储呢？人很聪明总能想出办法，一般二叉树相对完全二叉树只是中间缺少了一些结点，我们可以设法补齐缺少的结点，使成为一棵完全二叉树，这样我们就可以借用完全二叉树的方法来存储一般二叉树了。对增补的结点我们存入特殊的值以区分有效元素，比如下面图示中我们用“∧”符号表示增补的结点。图 5-14 所示是一个转换为完全二叉树的实例。



(a) 一棵非完全二叉树



(b)补齐缺少的结点使成为等高的完全二叉树

图 5-14 一般二叉树转换为完全二叉树

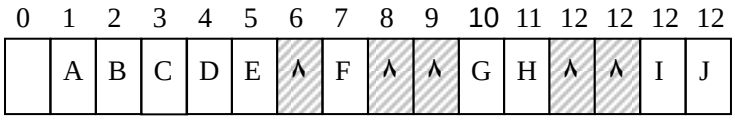
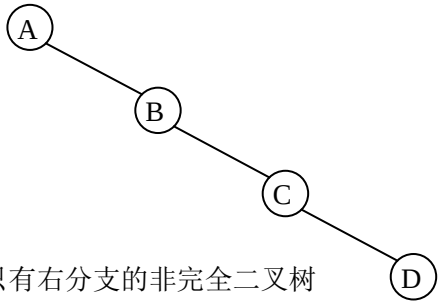
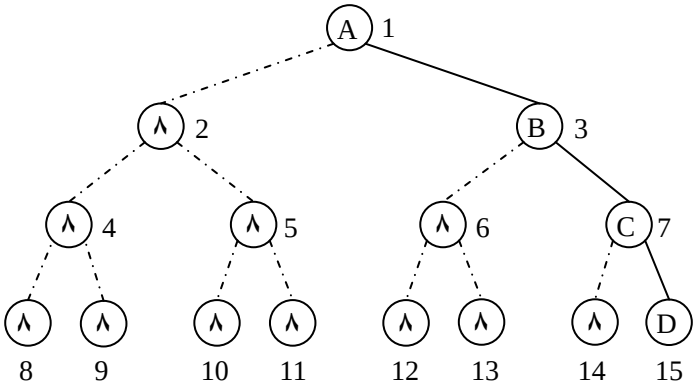


图 5-15 图 4-14 二叉树的顺序存储结构示意图

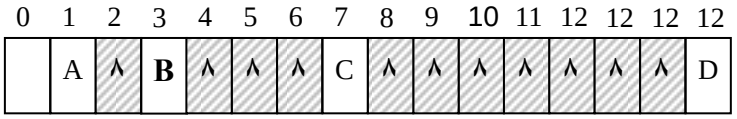
图 5-15 是图 5-14 所示一般二叉树转换为完全二叉树后的顺序存储示意图。从这个例子可以看出，虽然我们通过补齐结点解决了一般二叉树的顺序存储问题，但这种方法的缺陷是要浪费部分存储空间，本例中就有 5 个存储单元的浪费。最极端情况是二叉树除了叶子结点，其它每个结点只有一棵右子树，若此二叉树有 k 个结点，则高度即为 k，补齐结点后将成为高度为 k 的满二叉树，需要的总存储空间为 2^k-1 个单元，而有效使用单元只有 k 个。图 5-16 是一棵只有右分支，高度为 4 的二叉树转换的例子，转换后总共需要 15 个存储单元，实际有用单元只有 4 个。



(a)只有右分支的非完全二叉树



(b)补齐结点使成为等高的完全二叉树



(c) (a)所示二叉树的顺序存储结构示意图

图 5-16 只有右分支二叉树转换为完全二叉树

除了上述缺陷外，像顺序表一样二叉树的顺序存储结构在插入结点时，需要移动元素。为此我们需要能更加合理利用空间和方便操作的二叉树存储结构，于是就有了下述的二叉树的链式存储结构。

2. 二叉链表存储结构

二叉树的二叉链表存储结构中，结点由三个域构成：一个数据域用来存储数据元素，不妨设为 data；两个指针域，分别存放指向两个孩子结点的指针，不妨分别设为 lchild 和 rchild，结点结构形式如图 5-17 所示。

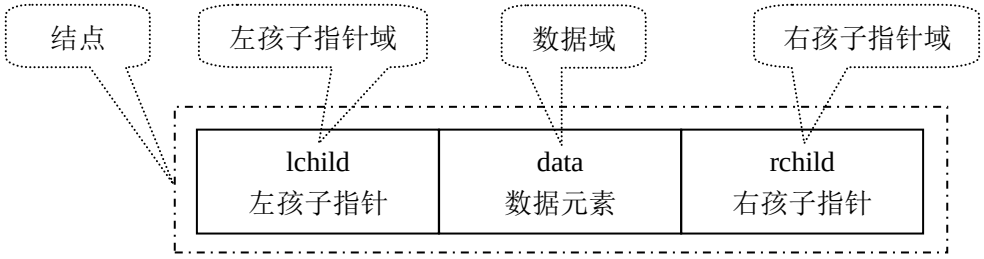


图 5-17 二叉链表结点结构示意图

【二叉链表存储结构描述】

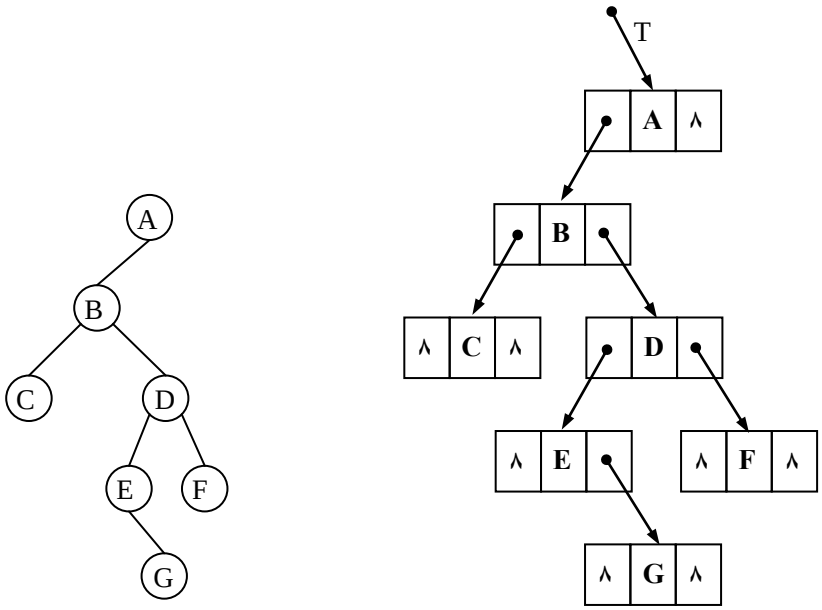
```
typedef char elementType;   //不妨将 elementType 定义为字符类型
typedef struct lBNode
{
    elementType data;                   //存放数据元素
    struct lBNode *lChild, *rChild;     //左、右孩子指针
}BiNode, *BiTree;
```

用这样的结点构造链表，每个结点有两个分叉的指针，分别指向其左右孩子结点，或者说指向其左右子树的根结点指针（尽管可能其值为空），因此称这样的链表为**二叉链表**。

二叉链表是二叉树的基本链式存储结构，后面讨论的二叉树的各种运算实现都是基于这个存储结构。线性链表中头指针可以唯一确定一个链表，与此类似，**二叉链表中根结点的指针可以唯一确定一棵二叉树**，因为有了根结点指针，再通过结点的左右孩子指针我们就可以搜索到树上的所有结点。因此常用根结点指针表示整个二叉链表，作为二叉链表的名称。后面讨论的二叉树的各种运算函数，都只要将二叉树的根结点指针作为参数传递到函数中即可，它代表了整个二叉链表。

一棵 n 个结点的二叉树，采用二叉链表存储时，共有 2n 个指针域。其中有 n-1 个指针非空（除了根节点，其它结点皆有指针指向），剩下 2n-(n-1)=n+1 个指针域为空。

举一个例子，对图 5-18(a)的二叉树，画出其二叉链表结构的图形表示，如图 5-18(b)，其中 T 是根结点指针，代表这个二叉链表。



(a) 一棵二叉树　　(b)二叉链表图形表示

图 5-18 二叉树链表结构图形表示

3. 三叉链表存储结构

在二叉链表上当前位置搜索孩子和后裔结点比较方便，但如果要从当前位置往上搜索双亲结点或祖先结点则无法办到，必须从根结点开始重新搜索。为了方便往上搜索双亲和祖先，我们可以改造二叉链表的结点结构，增加一个指向双亲结点的指针 parent，这样每个结点就有三个指针域，称这种结构的链表叫三叉链表。

【三叉链表存储结构描述】

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

三叉链表结构能提供双向搜索，但每个结点增加一个指针域，需要更多的内存开销。

5.3 二叉树的遍历

所谓**遍历（Traverse）二叉树**是指按某种次序访问二叉树中每个结点一次且仅一次。在访问每个结点的过程中，我们就可以对结点进行各种操作。比如：存、取结点信息，对结点进行计数等。事实上前面学习的线性表中也涉及到元素（结点）的遍历，只是线性表结点之间关系相对简单，我们称这种遍历操作为搜索。

需要强调的是**二叉树的遍历操作是二叉树其它各种运算的基础**。真正理解这一运算的实现及其含义有助于许多二叉树运算的算法设计和实现。然而，许多初学者开始时的学习效果并不理想，原因之一是理解其内在规律的困难。为此，本节先讨论其基本方法和算法实现，在此基础上讨论其应用，并通过实例加以说明。

5.3.1 遍历算法的实现

1. 基本遍历方法讨论

二叉树是一种非线性结构，其遍历操作就不像线性表搜索那么容易，线性表通过不断查找直接后继结点进行搜索，显然这个方法对二叉树行不通，因为二叉树中结点可能有两个子结点，如何继续访问结点就是个问题。那么，二叉树中，如何确保既不遗漏，也不重复地访问所有结点呢？这就要求我们要换一种构思方式。根据前面二叉树的递归定义，任何一棵二叉树都可以抽象为由三个部分构成：根结点（D）、左子树（L）和右子树（R）。如图 5-19 所示。

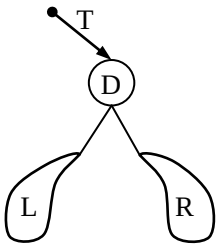


图 5-19　二叉树的一般形式示意图

针对图 5-19 抽象出的二叉树的一般形式，容易联想到如果我们能设法访问这三个部分中的每个结点一次且仅一次，那么就实现了遍历。这里根结点为单个结点可以直接访问，如果**左右子树 L 和 R 也能分别遍历出来**（即能各自单独地做到不重复、不遗漏地访问其中的每个结点），那么就完成了整棵树的遍历。接下来我们还要解决这三个部分的遍历和访问次序问题。首先对左右子树 L 和 R 的遍历次序，可以规定“先左后右”，即先遍历左子树 L，后遍历右子树 R；也可以是“先右后左”，即先遍历右子树，再遍历左子树。其次，对根结点 D 的访问次序，可以在遍历 L 和 R 之前首先访问；也可以在遍历 L 和 R 中一棵子树之后的中间访问；还可以在遍历 L 和 R 完成之后，最后访问。根据上面的情况总共可有 6 中次序组合，如表 5-1 所示。表中：D 表示访问根结点，L 表示遍历左子树，R 表示遍历右子树。

表 5-1 二叉树访问次序组合

	先左后右	先右后左
先序	DLR	DRL
中序	LDR	RDL
后序	LRD	RLD

表 5-1 中，我们以根结点的访问为基点，第二行先访问根结点，再遍历两颗子树，叫做**先根序遍历**，简称**先序遍历**；第三行先遍历一棵子树，再访问根结点，叫做**中根序遍历**，简称**中序遍历**；第四行先遍历两棵子树，再访问根结点，叫做**后根序遍历**，简称**后序遍历**。

我们一般约定左右子树的遍历次序采用“先左后右”次序，那么我们用到的遍历次序组后就剩下 3 中：**DLR、LDR 和 LRD**，即表 5-1 中第二列的内容。

现在，需要讨论的另一个问题是：如何实现左右子树的遍历？

事实上图 5-19 给出的二叉树的一般形式是递归的，即图中的左右子树 L 和 R 也是二叉树，所以我们可以**递归地使用同样的遍历策略来遍历这两棵子树**。即**对左右子树，可采用整棵二叉树相同的方式进行遍历**。

比如先序遍历中（DLR），我们对 L 和 R 也同样采用先序方式遍历。L 和 R 也可以抽象为根结点、更小的左右子树三个部分组成，这种抽象和分解可以一直持续下去，直到左右子树为单个结点。对这些更小的子树我们一样递归地使用先序策略进行遍历，直到左右子树为单结点时，就可以直接访问了，递归结束。中序遍历和后续遍历情况类似。

在本章后面的内容中，你将看到：**以上的讨论过程不仅只适用于二叉树的遍历，而且适用于二叉树各种运算的实现**。

基于以上讨论，我们可以给出二叉树三种遍历算法的自然语言描述：

【先序遍历--DLR】

若二叉树 T 非空，则：

①访问 T 的根结点。

②先序遍历 T 的左子树。

③先序遍历 T 的右子树。

【中序遍历--LDR】

若二叉树 T 非空，则：

①中序遍历 T 的左子树。

②访问 T 的根结点。

③中序遍历 T 的右子树。

【后序遍历--LRD】

若二叉树 T 非空，则：

①后序遍历 T 的左子树。

②后序遍历 T 的右子树

③访问 T 的根结点。。

有关算法的实现将在稍后给出。

2. 有关遍历方法的例题

为正确理解遍历算法的求解过程，下面给出几个实例。

【例 5.2】假设访问二叉树 T 的结点的操作是打印结点的值，请分别写出图 5-20 所示的二叉树的先序、中序和后序序列。

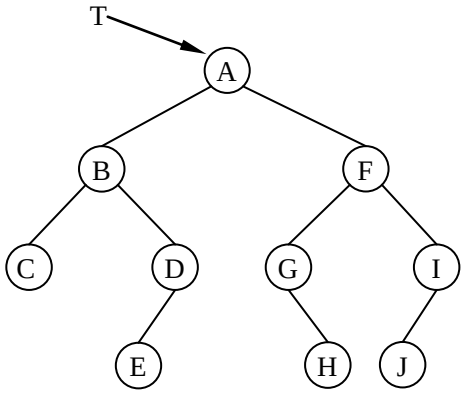


图 5-20 例 4.2 的二叉树

【解】为了写出各遍历序列，我们采用“分步填空”的方式：对整棵二叉树来说，先分别写出其遍历序列的三部分，即根结点和左右子树遍历序列。由于其中的左右子树的遍历序列暂时还不知道，因此先用空格来占位置，待下一步继续分解。例如：在求先序序列时，先写出如下的序列：

①A_____ _____

A_L（A 的左子树）A_R（A 的右子树）

然后对 A 的左右子树 A_L 和 A_R 分别用同样的方法，即也是按三部分划分、填空的方式进行。因此，对每个结点来说，都是先写出以此为根的子树的遍历序列的三部分，而将其中未知部分先用空格来占位，然后对其进行分解，直到全部填完空为止。基于这一方法的余下的求解序列如下：

②A B ___ F ___ ___

B B_L B_R F F_L F_R

A_L A_R

③A B C DE FG HIJ

B_L B_R F_L F_R

A_L A_R

④由此可知先序遍历序列为 ABCDEFGHIJ。

以同样的方式可得其它遍历序列分别如下，在此不再细述，请自己练习。

中序：CBEDAGHFJI

后序：CEDBHGJIFA

【例 5.3】已知二叉树的先序和中序序列如下，试构造出相应的二叉树。

先序：ABCDEFGHIJ

中序：CDBFEAIHGJ

【分析】这是前一类问题的逆问题，同样需要运用二叉树遍历思想求解。由二叉树的结构可知，如果能由这两个序列确定出根结点，并分别构造出其左右子树，即完成了问题的求解。因此，下面分别讨论这两个问题的求解：

①**根结点的确定**：由先序遍历的描述可知，先序序列中的第一个结点为根结点。因此，本题中的根为 A。

②**左右子树的确定**：在中序序列中，根结点处于中序序列中间某位置，将树上结点分割成两部分，A 左边部分是左子树结点，A 右边部分是右子树结点。因此本题中由根结点 A 的位置，可划分出左子树的中序序列为（CDBFE），右子树的中序序列为（IHGJ）。现在，如果能知道左右子树的先序序列，即可由此而采用相同的方式分别构造出左右子树，从而完成本题的求解。由先序遍历方法的描述可知，左、右子树的先序序列在先序序列中也是各自连在一起的，根据已经取得的子树中序序列，可到先序序列中取得左右子树的先序序列，本题中分别为 BCDEF 和 GHIJ。因此，左子树的构造需要由其先序序列 BCDEF 和中序序列 CDBFE 来求解，右子树的构造需要由其先序序列 GHIJ 和中序序列 IHGJ 来求解，而这又和原题形式相同，因此可采用同样的方式，即由先序序列确定根，中序序列分左右，求解过程如图 5-21 所示。

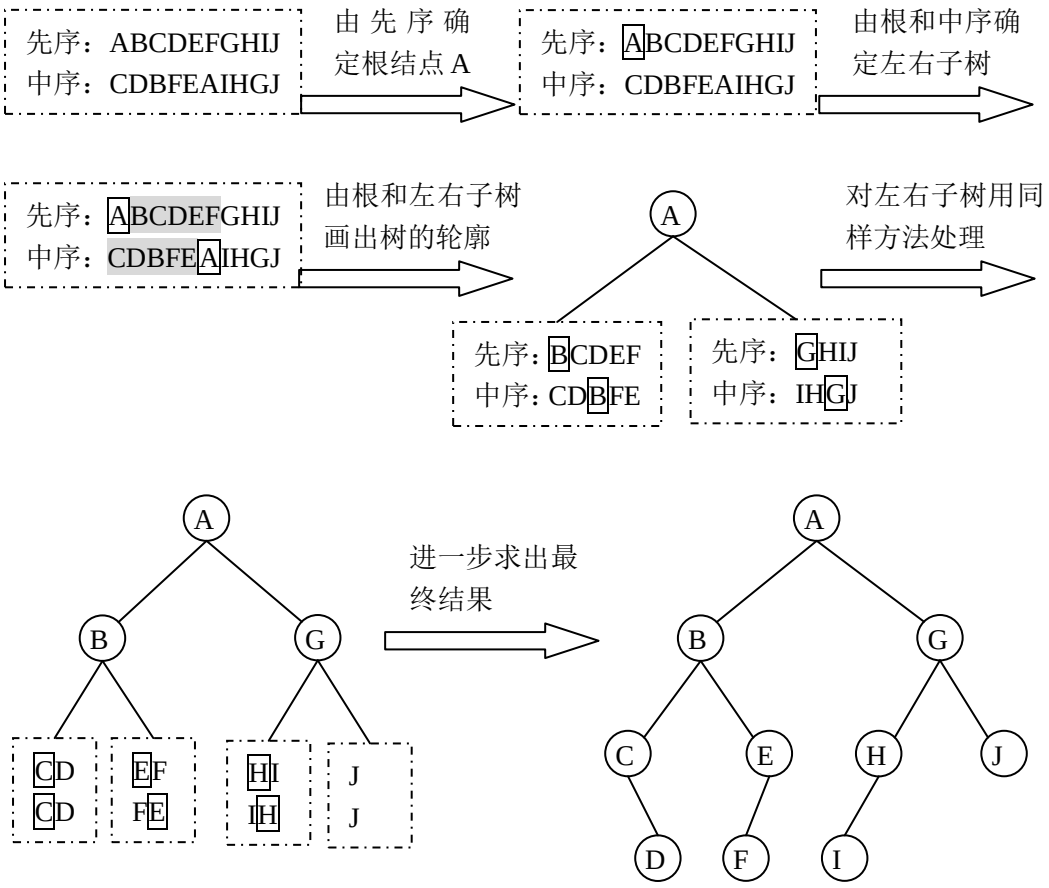


图 5-21 由先序和中序序列重构二叉树过程示意图

【思考问题】由本例我们知道由二叉树的先序序列和中序序列可以重构二叉树，那么由中序序列和后序序列是否可以重构呢？先序序列和后序序列是否可以重构呢？

由解题过程可以看出这是一个递归的求解过程，每一次处理都需要能划分出根结点、左子树和右子树。中序和后序序列组合的话，可由后序序列确定根结点，在后序序列的最后。有了根结点我们就可以在中序序列中划分出左右子树，所以中序序列和后序序列组合也能重

构二叉树。先序序列和后序序列组合呢？这种组合我们可以找到根结点，但却没法划分左右子树，所以不能重构二叉树。**结论**：先序和中序序列组合，中序和后序序列组合可以重构二叉树。先序和后序序列组合不能重构二叉树。

3. 二叉树遍历算法

前面我们已经给出了三种遍历算法的自然语言描述，现在我们给出遍历的代码描述。从以上对遍历方法的讨论可知，对二叉树的遍历是在对各子树分别遍历的基础之上进行的。由于各子树的遍历和整个二叉树的遍历方式相同，因此，可借助于对整个二叉树的遍历算法来实现对左右子树的遍历。也就是说，要采用递归调用方式来实现对左右子树的遍历。假定二叉树采用二叉链表存储结构，根结点指针为 T，则左右孩子结点的指针就是 T->lchild 和 T->rchild，用函数 visit(T)表示访问根结点。

（1）先序遍历算法

【先序遍历递归算法描述】

```
void PreOrderTraverse(BiNode* T)
{
        if(T)
        {
          visit(T);        //访问根结点
          PreOrderTraverse(T->lChild);   //递归调用先序遍历左子树
          PreOrderTraverse(T->rChild);   //递归调用先序遍历右子树
        }
}
```

其中访问根结点的操作 visite(T)在不同的场合下可以有不同的要求，其中最常用的操作是输出结点的值，比如：cout<<T->data，实现时可根据需要自行设计 visit(T)函数，下同。

第 3 章中我们介绍过使用栈可以把递归函数转换为非递归函数，下面给出先序遍历的非递归实现。

【先序遍历非递归算法描述】

```
void PreTraverseNR(BiNode* T)
{
        BiNode* p;
        seqStack S;
        initStack(S);   //初始化栈
        p=T;
        while(p || !stackEmpty(S))
        {
          if(p)
          {
              cout<<p->data<<", ";   //访问根结点，即 visit(T)功能
              pushStack(S, p);          //p  指针入栈
              p=p->lChild;              //遍历左子树
          }
          else
          {
```

```
                popStack(S, p);   //p 为空时，将上一层的根结点指针弹出
                p=p->rChild;      //遍历右子树
            }
        }
}
```

这里给出非递归算法不是为了掌握它，只是为了与递归算法进行对比。比较一下两个算法就可以发现**递归算法的“优美和魅力”**，看上去挺复杂的遍历算法，用递归算法实现只不过短短的 3 行代码，简单明了，很容易阅读和理解。非递归算法就不可同日而语了，首先代码量大大增加，其次，代码阅读起来晦涩难懂，实现时还容易出错。再来讨论一下两种算法的时空性能，两种算法都使用栈，只不过递归调用使用了系统自动提供的函数调用栈，非递归中使用的是自定义的软件栈。非递归是对递归的模拟，两者对栈的使用过程是相同的，即调用时相关数据入栈，返回时相关数据出栈。所以两个算法的空间需求基本相同。时间性能上非递归算法一般比递归算法差，因为系统提供的函数调用栈操作是经过充分优化的，而用户定义的软件栈由编译器编译自动生成栈的操作代码，性能不会优于系统栈。综上所述，解决同一个问题对比递归和非递归实现，递归实现时算法设计简单，代码简短，代码容易阅读理解，运行稳定性好，**如果都使用栈**递归算法的时空性能不逊于非递归算法，所以我们一般会首选递归实现方法。唯一的例外是递归调用太深，导致函数调用栈空间不够用，出现“栈溢出错”，这时就可能需要将其转换为非递归算法，定义足够大的软件栈以满足需求。

许多材料上笼统的说同一问题的非递归算法会优于递归算法这是不准确的。如果非递归算法不需使用栈，只需要使用循环（迭代）方法，的确非递归算法的时空性能会优于递归算法。如果非递归算法中也使用栈，那么这个非递归算法性能上一般会比递归算法差。

同样，我们以对比的方式给出中序遍历和后序遍历的递归和非递归算法描述。

（2）中序遍历算法

【中序遍历递归算法描述】

```
void InOrderTraverse(BiNode* T)
{
        if(T)
        {
            InOrderTraverse(T->lChild);    //递归调用中序遍历左子树
            visit(T);                      //访问根结点
            InOrderTraverse(T->rChild);    //递归调用中序遍历右子树
        }
}
```

【中序遍历非递归算法描述】

```
void InTraverseNR(BiNode* T)
{
        BiNode* p;
        seqStack S;
        initStack(S);   //初始化栈
        p=T;
        while(p || !stackEmpty(S))
        {
```

```
            if(p)
            {           //根结点先入栈，以便左子树遍历结束，返回访问根结点
                pushStack(S,p);
                p=p->lChild;        //遍历左子树
            }
            else   //p 为空--访问根结点、遍历右子树
            {
                popStack(S, p);          //某子树的根结点出栈
                cout<<p->data<<", ";   //访问某子树根结点，即 visit(T)
                p=p->rChild;             //遍历 p 的右子树
            }
        }
}
```

（3）后序遍历算法

【后序遍历递归算法描述】

```
void PostOrderTraverse(BiNode* T)
{
        if(T)
        {
            PostOrderTraverse(T->lChild);   //递归调用中序遍历左子树
            PostOrderTraverse(T->rChild);   //递归调用中序遍历右子树
            visit(T);                       //访问当前根结点
        }
}
```

【后序遍历非递归算法描述】

```
void PostTraverseNR(BiNode* T)
{
        BiNode* p;
        seqStack S;
        int tag[MaxLen];   //标记左子树、右子树
        int n;
        initStack(S);   //初始化栈
        p=T;
        while(p || !stackEmpty(S))
        {
            if(p)
            {
                pushStack(S,p);
                tag[S.top]=0;     //标记遍历左子树
                p=p->lChild;     //循环遍历左子树
            }
```

```
            else   //p==NULL，  但是栈不空
            {
                stackTop(S,p);   //取栈顶，但不退栈，以便遍历 p 的右子树
                if(tag[S.top]==0)   //说明 p 的左子树已经遍历结束，右子树尚未遍历
                {
                    tag[S.top]=1;   //设置当前结点遍历右子树标记
                    p=p->rChild;   //遍历右子树
                }
                else   //tag[S.top]==1，说明 p 的左右子树皆已经遍历，
                {
                    popStack(S,p);   //退栈
                    cout<<p->data<<", ";   //访问某子树根结点，即 visit(T)
                    //上面出栈的 p 已经没用，置空，回去循环取栈顶的下一个元素
                    p=NULL;
                }
            }
        }
}
```

4. 遍历算法的执行过程

二叉树的遍历算法以递归方式给出，可能对大多数初学者来说仍不太容易理解，下面我们用第 3 章介绍的模拟递归调用执行过程方法来模拟遍历算法的执行过程，以加深理解。

【例 5.4】对图 5-22 所示二叉树的中序遍历算法模拟递归调用的执行过程。

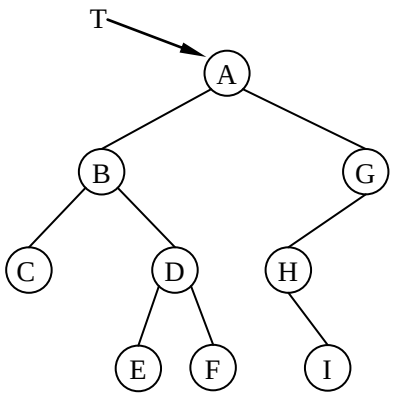


图 5-22 例 4.4 的二叉树

【解】按前面所讨论的方法及中序遍历算法可知，执行 InOrderTraverse(T)等价于依次执行下面三个操作，即 InOrderTraverse (T->lchild)，visite(T)，InOrderTraverse (T->rchild)。为简化描述我们引用下面一些符号：我们用较短的符号 InOrder(T)代替 InOrderTraverse(T)表示中序遍历；用结点的名称表示结点的指针，比如 InOrder(T)表示为 InOrder(A)，InOrder(T->lchild)表示为 InOrder(B)，InOrder(T->rchild)表示为 InOrder(G)等，空指针用“^”；访问根结点 visit(T)相应为 visit(A)、visit(B) 等我们用结点名称加方框表示为：A、B 等。有

了上面这些符号，三个操作就可以表示为：InOrder(B)；A；InOrder(G)。由此可得算法执行的过程如图 5-23 所示。

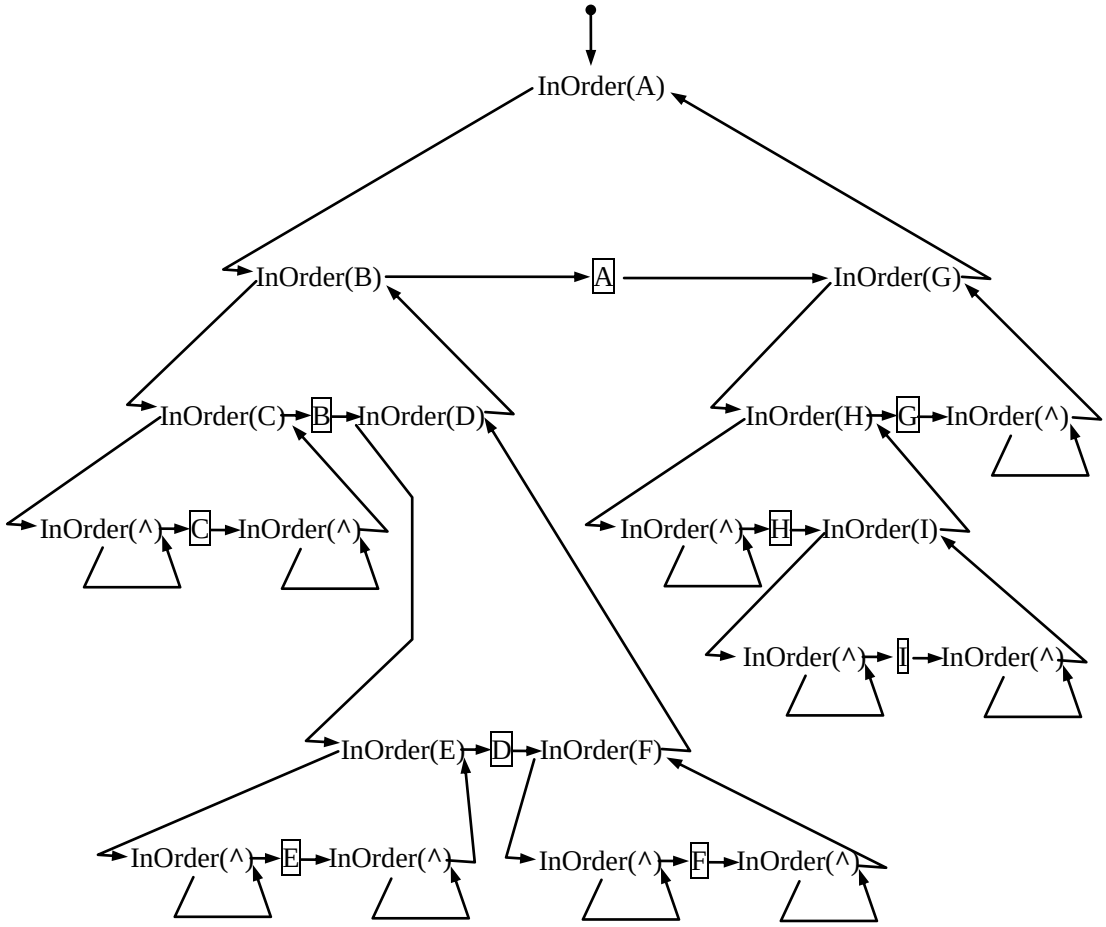


图 5-23 中序遍历递归算法执行过程示意图

由图 5-23 所示的的执行流程可知，其输出序列是：CBEDFAHIG。

5.3.2 二叉树的创建与销毁

读者如果想实际上机体验二叉树的遍历算法以及后续的二叉树其它算法，第一步就要会创建二叉树，创建二叉树是其它算法实现的基础。事实上我们学习任何数据结构，要实现其各种算法，第一步就要学会创建这种数据结构，在实践环节中这是非常重要的。我们前面学习的线性表、栈、队列等结构创建起来相对简单，所以没有专门介绍它们的创建方法。

二叉树的顺序存储结构只要用一个数组就可以了，需要注意的是要把普通二叉树补齐为完全二叉树然后存放在数组中，且数组的 0 单元最好不存放树的结点，二叉树的这种顺序存

储结构创建非常简单，这里不专门介绍。

因为二叉树常用二叉链表存储结构，接下来我们介绍基于二叉链表结构的二叉树的创建和销毁。

1. 控制台交互输入创建二叉树

即由键盘交互输入二叉树的结点数据来创建二叉树。接下来我们介绍一种基于二叉树先序遍历次序创建二叉树的方法，即键盘输入时按二叉树的先序遍历次序输入结点数据，没有子树时用特殊符号表示，下面的代码中以特殊符号“/”表示没有子树。二叉树的创建由 2 个函数合作完成，程序如下：

【创建子树函数】

```
void createSubTree(BiNode* q, int k)
{       //q 为当前根结点
        //k=1--左子树；k=2--右子树
    BiNode* u;
    elementType x;
    cin>>x;          //键盘读入结点数据
    if(x!='/')       //x=='/'表示没有子树
    {
        u=new BiNode;
        u->data=x;
        u->lChild=NULL;
        u->rChild=NULL;
        if(k==1)
            q->lChild=u;     //新结点 u 为当前根结点 q 的左子树
        if(k==2)
            q->rChild=u;     //新结点 u 为当前根结点 q 的右子树
        createSubTree(u,1);  // 递归创建 u 的左子树
        createSubTree(u,2);  // 递归创建 u 的右子树
    }
}
```

【创建二叉树函数】

```
void createBiTree(BiNode* & BT)
{
    BiNode* p;
    char x;
    cout<<"请按先序序列输入二叉树，（‘/’无子树）:"<<endl;
    cin>>x;
    if(x=='/')
        return;          //空树，退出
    BT=new BiNode;   //创建根节点，指针为 BT
    BT->data=x;
    BT->lChild=NULL;
    BT->rChild=NULL;
```

```
    p=BT;                    //当前节点指针
    createSubTree(p,1);   //创建根节点左子树
    createSubTree(p,2);   //创建根节点右子树
}
```

例如对图 5-24 中二叉树，先序遍历次序为：abdeghcfi，以“/”表示无子树，则相应的键盘输入为：abd//eg//h//cf/i///。其中 d 后面的 2 个“//”表示结点 d 无左子树，也无右子树；结点 g 和 h 后面的 2 个“//”也表示这 2 个结点既无左子树，又无右子树；结点 f 后面的 1 个“/”表示结点 f 无左子树；结点 i 后面的 3 个“///”，其中前 2 个表示结点 i 无左子树和右子树，最后 1 个表示结点 c 无右子树。

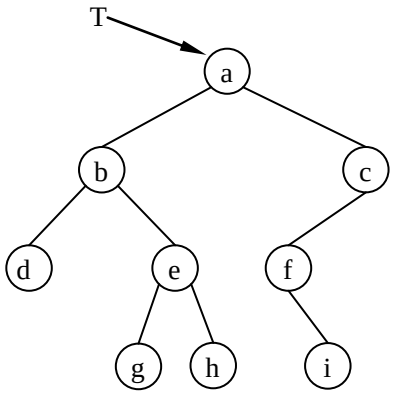


图 5-24 一棵二叉树

2. 数据文件输入创建二叉树

上面介绍的交互式输入创建二叉树一般只适用于树的结点数较少时，当树的结点数较多时，交互输入容易出错，浪费时间，且容易造成内存泄漏。下面我们介绍一种基于文本文件的二叉树创建方法，即将二叉树的结点信息保存在一个文本文件中，然后用程序自动读入来创建二叉树。我们先来定义文本文件的格式。

标识行：BinaryTree—标识这是一个二叉树的数据文件。

结点行：每个结点一行，结点严格按照先序遍历次序排列。每行 3 列。第 1 列为结点数据；第 2 列标识有无左子树，1—有左子树，0—无左子树；第 3 列标识有无右子树，1—有右子树，0—无右子树。

对图 5-24 所示二叉树，完整数据文件如下：

```
BinaryTree
a 1 1
b 1 1
d 0 0
e 1 1
g 0 0
h 0 0
c 1 0
f 0 1
i 0 0
```

数据文件的扩展名随意，只要按文本文件读写即可，例如上述文件不妨命名为BiTree9.bit。

接下来介绍基于上述数据文件创建二叉树的函数。

【结点数据读入数组函数】

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

```
bool ReadFileToArray(char fileName[], char strLine[100][3], int & nArrLen)
{       // fileName[]存放文件名
        // strLine[][3]存放结点的二维数组，数组的 3 列对应数据文件的 3 列
        // nArrLen 返回二叉树结点的个数
    FILE* pFile;   //定义二叉树的文件指针
    char str[1000];   //存放读出一行文本的字符串
    pFile=fopen("BiTree9.bit","r");
    if(!pFile)
    {
        printf("二叉树数据文件打开失败！\n");
        return false;
    }
        //读取文件第 1 行，判断二叉树标识 BinaryTree 是否正确
    if(fgets(str,1000,pFile)!=NULL)
    {
        if(strcmp(str,"BinaryTree\n")!=0)
        {
            printf("打开的文件格式错误！\n");
            fclose(pFile);        //关闭文件
            return false;
        }
    }
    nArrLen=0;
    while(fscanf(pFile,"%c   %c   %c\n",
          &strLine[nArrLen][0],&strLine[nArrLen][1],&strLine[nArrLen][2])!=EOF)
    {                   //循环读取结点行数据，存入数组，结点总数加 1
        nArrLen++;
    }
    fclose(pFile); //关闭文件
    return true;
}
```

【从数组数据按先序次序创建二叉树】

```
bool CreateBiTreeFromFile(BiNode* & pBT, char strLine[100][3],int nLen, int & nRow)
{
        //strLine[100][3]--保存结点数据的二维数组
```

```
        //nLen--结点个数
        //nRow--数组当前行号
    if((nRow>=nLen) || (nLen==0))
        return false;    //数据已经处理完毕，或者没有数据，退出
        //根据数组数据递归创建二叉树
    pBT=new BiNode;   //建立根结点
    pBT->data=strLine[nRow][0];
    pBT->lChild=NULL;
    pBT->rChild=NULL;
    int nRowNext=nRow;              //保留本次递归的行号
    if(strLine[nRowNext][1]=='1')
    {                   //当前结点有左子树，读下一行数据，递归调用创建左子树。
        nRow++;        //行号加 1
        CreateBiTreeFromFile(pBT->lChild, strLine,nLen,nRow);
    }
    if(strLine[nRowNext][2]=='1')
    {                   //当前结点有右子树，读下一行数数，递归调用创建右子树。
        nRow++;      //行号加 1
        CreateBiTreeFromFile(pBT->rChild, strLine,nLen,nRow);
    }
    return true;
}
```

3. 二叉树的销毁

创建了二叉链表存储结构的二叉树，使用完毕后应当释放此二叉树占用的内存，因为在创建二叉树时使用 malloc()函数或 new 操作符动态申请了内存，当这个二叉树不再需要时，必须手工释放动态申请的内存，否则造成内存泄漏。下面给出销毁二叉链表存储结构二叉树的程序。

```
void DestroyBiTree(BiNode* pBT)
{
    if(pBT)
    {
        DestroyBiTree(pBT->lChild);     //递归销毁左子树
        DestroyBiTree(pBT->rChild);     //递归销毁右子树
        delete pBT;                      //释放当前根结点
    }
}
```

5.3.3 二叉树遍历算法的应用

前面曾指出二叉树的遍历算法是二叉树算法的基础，下面结合实例对此展开讨论。根据所运用的基本思想来分，可划分为两个层次，其一是简单、直接的应用，其二是具有一定深

度的方法的应用。

1. 二叉树遍历算法的简单应用

二叉树的三种遍历算法都能**对二叉树 T 中的每个结点执行一次且仅执行一次访问操作**。即每个结点都会执行仅一次的 visit()访问。二叉树有些问题的求解，我们只要通过定义不同的 visit()函数即可实现，这样的应用就是直接基于遍历算法实现的。下面给出几个典型问题的求解。这些问题都只需修改 visit()函数，直接基于遍历算法进行求解。

【例 5.5】 设计算法按中序次序输出二叉树 T 中度为 2 的结点的值。

【解】本题可直接基于中序遍历算法，我们只需要简单修改 visit()函数，在访问结点时判断一下其度数是否为 2，为 2 的话就打印结点的值，否则不打印。度数为 2 的结点的判定条件是它的 lchild 和 rchild 指针都不为空。为简单起见，我们把访问代码直接放到遍历算法中，取消 visit()函数。为适应打印不同类型的结点值，这里采用 C++中的输出语句来输出表达式的值。算法如下：

【算法描述】

```
void InTraverse(BiNode* T)
{
    if(T)
    {
        InTraverse(T->lChild);  //递归调用中序遍历左子树
        if(T->lchild!=NULL && T->rchild!=NULL)
            cout<<T->data<<;    //打印度数为 2 的结点值
        InTraverse(T->rChild);  //递归调用中序遍历右子树
    }
}
```

【思考问题】怎样按先序和后序次序输出度数为 2 的结点？

【例 5.6】设计算法求二叉树 T 的结点数。

【解】本算法不是要输出每个结点的值，所要求的仅是求出其中的结点数。我们只要将遍历算法中的 visit()函数改造为计数操作即可，即使用一个计数变量（初值为 0），每当要访问一个结点时，计数加 1，遍历结束即求出了总结点数。最直观的方式是我们设置一个全局变量 num，初始化时 num=0，用其计数结点。我们仍然采用中序遍历算法来求解。算法如下：

【算法描述】

```
void GetNodeNumber(BiNode* pBT)
{
    if(pBT!=NULL)
    {
        GetNodeNumber(pBT->lChild);     //递归遍历左子树
        num++;                          //计数结点，num 为全局变量
        GetNodeNumber(pBT->rChild);     //递归遍历右子树
    }

}
```

【思考问题】

①本算法用中序遍历计数结点个数，用先序遍历和后序遍历算法是否可以完成呢？如何完成？

②本算法用一个全局变量计数结点个数，用引用和指针变量是否可以完成计数呢？

【引用变量计数代码】

```
void GetNodeNumber(BiNode* pBT, int & nNodeNum)
{
  if(pBT!=NULL)
  {

      GetNodeNumber(pBT->lChild, nNodeNum);
      nNodeNum++;
      GetNodeNumber(pBT->rChild, nNodeNum);
  }
}
```

（二）二叉树遍历算法思想的应用（*）

还有一些问题仅按照前述方法不能方便地实现求解，或者算法形式较繁杂。另外，许多初学者在编写、阅读及证明递归算法时，过于陷入细节问题，因而对所编写的算法难以放心。为此，本小节讨论这两方面的问题。

一般来说，当用前面所讨论的方法不能有效地求解某些问题时，可尝试采用下面所讨论的方法来实现求解。这种方法在讨论遍历算法时已经用到了，在此作进一步归纳，描述如下：

首先，要明确所要编写的算法的功能描述（包括所涉及的各参数或变量的含义）——这在递归算法中尤其要注意。在此基础上按如下步骤讨论算法的实现：

①如果 T 为空，则按预定功能实现对空树的操作，以满足功能要求（包括对相应参数，变量的操作）。

②否则，假设算法对 T 的左右子树都能分别实现预定功能，在此基础上，通过按预定要求适当调用对左右子树的算法的功能，及对当前结点的操作实现对整个二叉树的功能（包括对各变量、参数的操作）。

这一方法中的难点是第二点，即假设算法对左右子树均能正确求解这一点上。

【例 5.7】设计算法求解给定二叉树的高度。

【解】由于求二叉树的高度难以采用由遍历算法简单变化的方式来实现，因此，需要采用本小节所讨论的方法来求解。分析如下:

(1) 若 T 为空时,则其高度为 0，求解结束。

(2) 否则，若 T 不为空,其高度应是其左右子树高度的最大值再加 1。假设其左右子树的高度能求解出来，则算法求解容易实现。而其左右子树的高度的求解又可通过递归调用本算法来完成。据此讨论可写出几种形式的算法,下面给出有值函数形式的算法。

设函数 int Height(BiNode *T) 表示“返回二叉树 T 的高度”，因此 m=Height(T->lchild)、n=Height(T->rchild) 分别代表 T 的左右子树的高度，则树的高度为 m、n 中较大者再加 1，由前面的讨论可得算法如下:

【算法描述】

```
int BiTreeDepth(BiNode* T)
```

```
{
    int m,n;
    if(T==NULL)
        return 0;  //空树，高度为 0
    else
    {
        m=BiTreeDepth(T->lChild);  //求左子树高度（递归）
        n=BiTreeDepth(T->rChild);  //求右子树高度（递归）
        if(m>n)
            return m+1;
        else
            return n+1;  //return (m>n?m:n)+1;  //简略写法
    }
}
```

5.4 线索二叉树

在二叉树中可能会要求求解某结点在某种次序下的前驱或后继结点，并且各结点在每种次序下的前驱、后继的差异较大。例如，图 5-25 中的二叉树的结点 D 在先序次序下的前驱、后继分别是 C、E；在中序次序中的前驱、后继分别是 E、F；在后序次序的前驱后继分别是 F、B。这种差异使得求解较为麻烦。如何有效地实现这一运算？对此，有几种考虑：

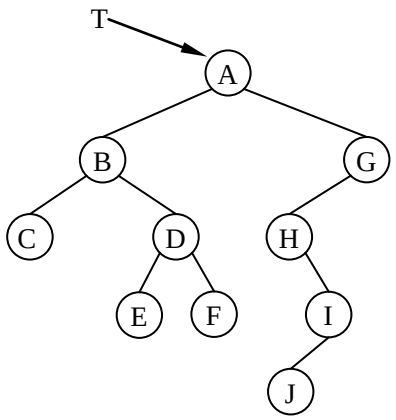


图 5-25 一棵二叉树

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

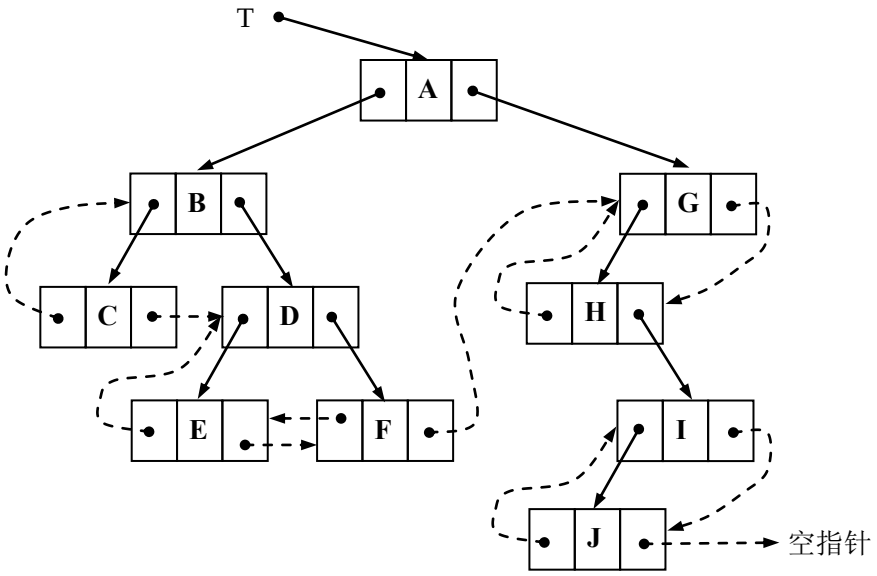
增设前驱和后继指针：在每个结点中增设两个指针，分别指示该结点在指定次序下的前驱或后继。这样，就可使前驱和后继的求解较为方便，但这是以空间开销为代价的。

是否存在既能少花费时间，又不用花费多余的空间的方法呢？下面要介绍的第三种方法就是一种尝试。

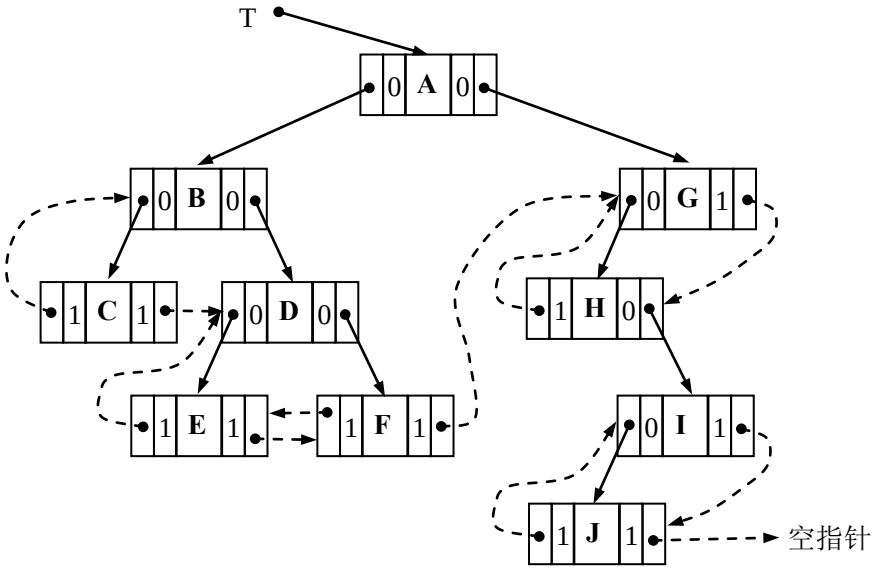
5.4.1 线索二叉树结构

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

例如，图 5-25 中二叉树的先序线索二叉树的二叉链表结构如图 5-26(a)所示，其中线索用虚线表示：



（a）未加区分标志的先序线索二叉树的二叉链表示例



（b）先序线索二叉树链表结构示例

图 5-26 线索二叉树的二叉链表形式示例

然而，仅仅按照这种方式简单地修改指针的值还不行，因为这将导致难以区分二叉链表中各结点的孩子指针和线索（虽然由图中可以“直观地”区分出来，但在算法中却不行）。例如，图 5-25 中结点 C 的 lchild 指针域所指向的结点是其左孩子还是其前驱？为此，在每个结点中需再引入两个区分标志 ltag 和 rtag，并且约定如下：

ltag=0： lchild 指示该结点的左孩子。

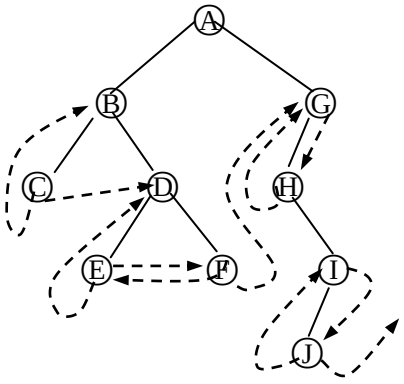
ltag=1： lchild 指示该结点的前驱。

rtag=0： rchild 指示该结点的右孩子。

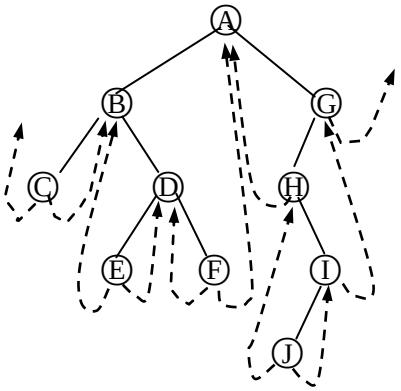
rtag=1： rchild 指示该结点的后继。

这样一来，图 5-26(a)中的二叉链表事实上变成了图 5-26(b)所示的形式。这是线索二叉树的内部存储结构形式。

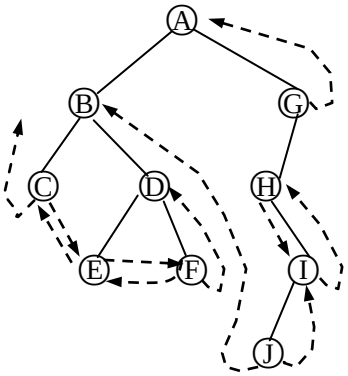
为简便起见，通常将线索二叉树画成如图 5-27 所示的形式。



(a)先序线索二叉树



(b)中序线索二叉树



(c)后序线索二叉树

图 5-27　线索二叉树示例

5.4.2 线索二叉树中前驱后继的求解

回到前面所描述的问题，即求解给定结点在指定次序下的前驱和后继。显然，共有三组六个问题：（1）先序线索二叉树中求解先序前驱和先序后继；（2）中序线索二叉树中求解中序前驱和中序后继（3）后序线索二叉树中求解后序前驱和后序后继。下面仅侧重讨论先序后继、中序后继求解的求解。

1. **先序后继的求解：**

【问题描述】在先序线索二叉树中求解指针 P 所指结点的后继结点（的指针）。

【分析】此处所谓 P 所指结点的先序后继即是在先序序列中紧跟在*P 后面的结点。按先序遍历的定义，以 P 为根指针的子树的遍历次序是 P 、P_L、R_R。其中 P 代表 P 所指结点，P_L 和 P_R 分别代表先序遍历 P 所指结点的左、右子树。由这一顺序可知：

（1）若*P 有左孩子（即左子树不空），则其左子树 P_L 中的第一个元素就是*P 的后继，

而 P_L 中的先序序列的第一个结点即是其根结点，即 P 的左孩子（指针为 P->lchild）。

（2）否则，若*P 有右孩子，则其右孩子就是其后继，其指针为 P->rchild。

（3）否则，P->rchild 即是后继线索。

由此可得求解先序后继的算法。

【算法描述】

```
BiNode   *preSuc(BiNode *P)
{
    if (P->ltag==0)
        return(P->lchild);
    else   return(P->rchild);
}
```

2. 中序后继的求解

【问题描述】在中序线索二叉树中求解 P 所指结点的中序后继。

【分析】设*P 的左右子树分别为 P_L 和 P_R，由于中序遍历的次序为 P_L，P，P_R，因此，其求解讨论如下：

（1）设 P_R 不空，则 P_R 中（中序序列）的第一个结点为*P 的后继。P_R 中的第一个结点有什么规律？如何求解？下面先做分析。不妨设二叉树形式如图 5-28 所示：

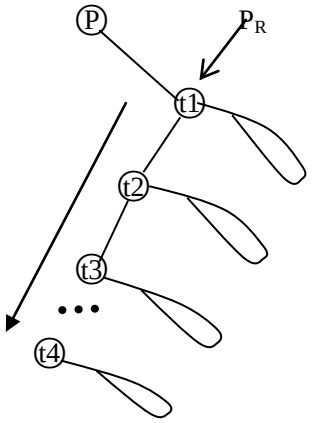


图 5-28 二叉树

若 t1 的左子树为空，则 t1 为 P_R 的第一个结点，否则，P_R 中的第一个中序结点应在子树 t2 中。类似地，若 t2 的左子树为空，则 t2 为 P_R 的第一个结点，否则，P_R 的第一个中序结点应在子树 t3 中。以此类推可知，其求解过程就是沿着箭头方向找到左下方的第一个没有左孩子的结点。也就是说，P_R 中的第一个结点就是在以 P_R 为根指针的子树中的最左下的第一个没有左孩子的结点。

（2）若 P_R 为空（即*P 无右孩子），则*P 的右孩子指针为后继线索，即 P->rchild 为其中序后继的指针。

综合上述分析可得求解中序后继的算法。

【算法描述】

```
BiNode *inSuc(BiNode *p)
{   Bnode *q=p->rchild;
    if (p->rtag==1)   return(q);
```

```
        else { while (q->ltag==0)
                    q=q->lchild;     //沿左下方向搜索
               return(q);
             }
    }
```

同理可知，中序前驱的求解与中序后继的求解相似，后序前驱的求解与先序后继的求解相似。另外，通过分析可知，在先序线索二叉树中求解结点的先序前驱是难以实现的。同样，在后序线索二叉树中求解结点的后序后继也是难以实现的。

通过引用上述求解后继的算法，可写出遍历线索二叉树的非递归算法。

例：先序遍历先序线索二叉树 T 的非递归算法如下：

```
void preOrder(BiNode *T)
{BiNode *P=T;         //先指向第一个结点
 while (P!=NULL)
     {viste(P);           //访问当前结点
      P=preSuc(P);     //求解其后继结点
      }
}
```

在线索二叉树中可能会涉及到插入、删除结点或子树的操作。在线索二叉树中插入一个结点时，不仅要按要求将结点作为指定结点的左孩子或右孩子插入进去，还要修改相应结点的线索及标志，以使插入结点后的二叉树仍满足相应的线索二叉树的特性。这一运算有一定的难度，故不再详细讨论。

在实际上机求解有关线索二叉树的问题时，还需要将二叉树转换成线索二叉树，即二叉树的线索化。考虑到这一内容的难度较大，故不在此处讨论，有兴趣的读者可参考有关书籍。

5.5 树和森林

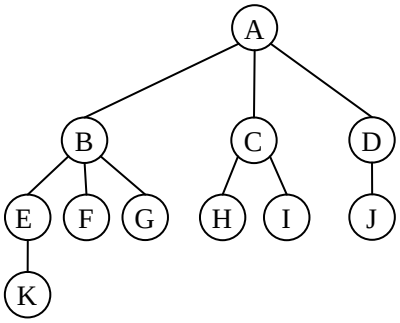
下面讨论树和森林的有关内容，包括树和森林的存储形式，树（森林）与二叉树之间的相互转换，树（森林）的遍历等。为描述方便起见，在不作特别说明的情况下，树包括森林。

5.5.1 树的存储结构

树有多种存储结构，其中最常见的是双亲表示法、孩子链表表示法和孩子-兄弟链表表示法。不同的结构对有关运算的实现有较大的差异。

1. 双亲表示法

双亲表示法通过给出树中每个结点的双亲来表示树。由于每个结点最多有一个双亲结点，因此结点的存储信息包括两部分：结点本身的值 data 和双亲结点在该表中的地址。例如图 5-29（a）中的树的双亲表示如图 5-29（b）所示。其中根结点 A 的双亲地址不存在，不妨设为-1。



（a）树

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

（b）双亲表示法

图 5-29　树的双亲表示法示例

有关其存储结构的说明包括两部分：结点的说明和存储数组的说明。

```
struct    tnode        //结点说明
{    datatype    data;
     int    parent;
 }
struct tnode treelist[maxnum];       //整个树的存储数组说明
```

其中 parent 指示父结点的下标，data 存放结点的值。

这种方法便于搜索相应结点的父结点及祖先结点，但若要搜索结点的孩子结点及其后代结点，则需搜索整个表，因此较费时间。若要求更有效地搜索其后代结点，则需要重新选择存储结构。

2. 孩子链表表示法

孩子链表表示法就是分别将每个结点的孩子结点连成一个链表，然后将各表头指针放在一个表中构成一个整体结构，如图 5-29（a）所示的树用孩子链表表示法如图 5-30 所示。

孩子链表表示法的描述中需要分别说明结点和总体结构两部分，其类型描述如下：

```
typedef struct       //结点类型描述
   {      int data;
          listnode *next;
   }    listnode;
typedef struct         //数组元素类型描述
   {      datatype info;
          listnode    *firstchild;
   } arrelemnt;
arrelemnt tree[maxnum];     //整体结构类型描述
```

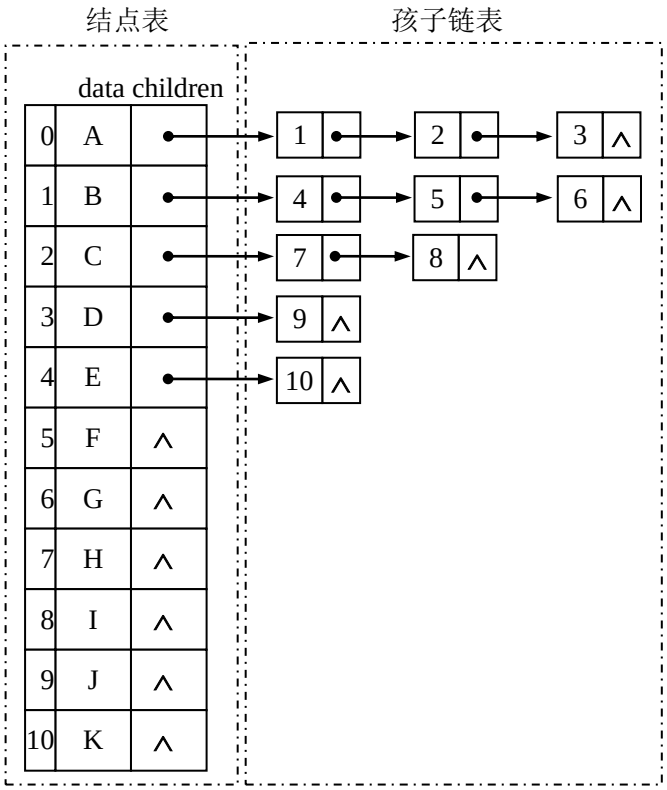


图 5-30 孩子链表表示法

与双亲表示法相反，这种结构便于搜索任意结点的孩子结点及后代结点，但不便于搜索各结点的双亲结点和祖先结点。我们在下一章学习图的表示法中会见到类似的方法—图的邻接链表表示法。

3. 孩子－兄弟链表表示法：

所谓孩子－兄弟链表表示法是指以这样的链表来存储树：对树中每个结点用一个链表结点来存储，每个链表结点中除了存放结点的值外，还有两个指针，一个用于指示该结点的第一个孩子，另一个用于指示该结点的下一个兄弟结点，故有此名。孩子－兄弟链表表示法结点结构如图 5-31 所示。这样，图 5-29(a)中的树的孩子兄弟链表结构如图 5-32 所示。

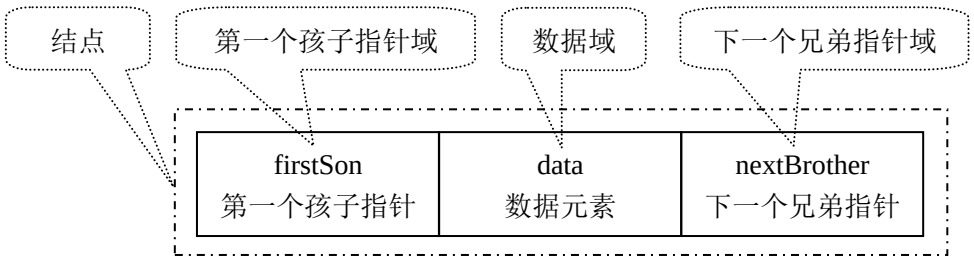


图 5-31 孩子—兄弟链表结点结构

若将该链表按顺时针方向旋转 45^0，如图 5-33 所示，可以看出该链表事实上就是前面所介绍的二叉树的存储结构之一，即二叉链表结构。由于每个二叉链表对应唯一一棵二叉树，

由此可知，每棵树对应唯一一棵二叉树。也正因为如此，将这种表示法称为二叉链表表示法或二叉树表示法。在稍后的内容中我们将会看到任何一棵树都可以按特定的方法转换为唯一的一棵二叉树，这里的孩子-兄弟表示其实就是树转换为二叉树后的二叉链表表示。

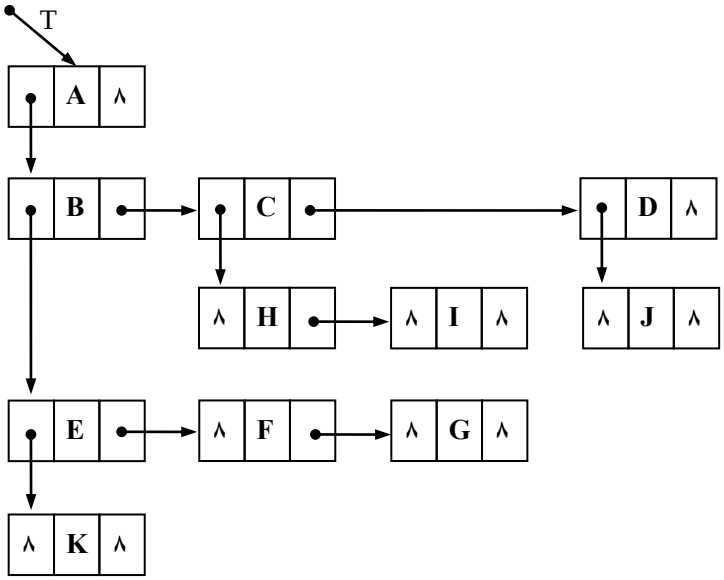


图 5-32 孩子－兄弟链表表示法示例

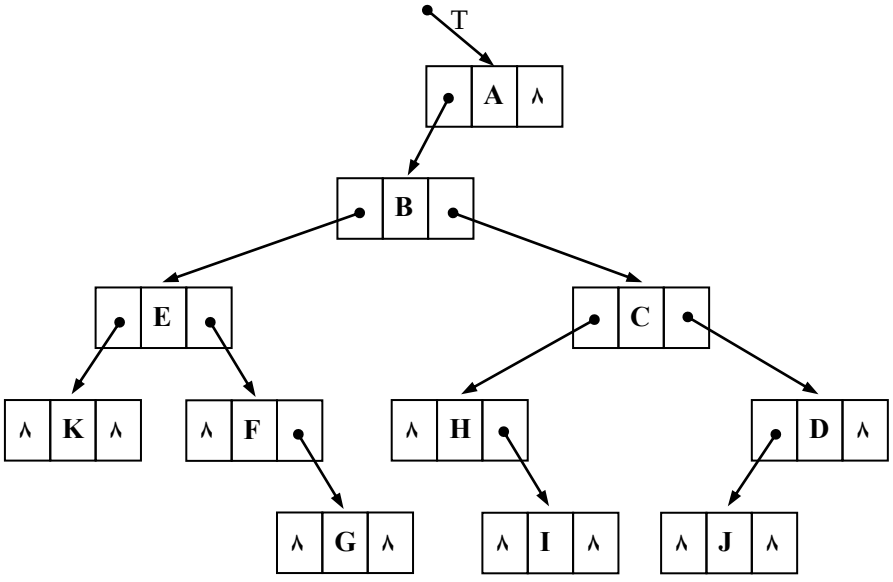


图 5-33　图 5-29(a)所示树对应的二叉链表表示

如果要存储森林的话，可以这样实现：将每棵树的根结点看作是兄弟结点。在这种约定下即可方便地实现森林的存储。

二叉链表存储结构的描述与二叉树存储结构的描述相同，主要说明结点的结构。

每个结点（不妨用 tnode 表示其类型）中由三部分组成：即结点的值（不妨用 data 表示），指向第一个孩子结点的指针（不妨记为 firstson）和指向下一个兄弟结点的指针（不妨记为 nextbrother）。描述如下：

```
typedef struct
{     datatype data;
      struct tnode *firstson, *nextbrother;
}tnode;
```

由于树和森林可以转换为二叉链表存储形式，因此可借助于二叉树问题的求解方法实现对树和森林的运算。由此可进一步体会到二叉树运算的基础性。

5.5.2 树（森林）与二叉树的转换

基本思想：树（森林）按一定方法转换为二叉树，转换是唯一的；二叉树还原为树（森林），转换也是唯一的。这样，任何对树或森林的操作都可以转换为二叉树以后实现，然后再还原为树（森林）。由前面的介绍可知树和二叉树都可以用二叉链表表示，即物理存储结构可以相同，只是解释不同而已。这样，借助于树或森林的孩子－兄弟链表表示法，可以将森林转化成唯一的一个二叉链表结构或二叉链表;反之由二叉链表也可唯一地对应到一个森林。因此，可将森林转换为二叉树的形式，并借助于二叉树的有关算法实现对森林的运算。这样就需要研究树（森林）与二叉树的相互转换。下面我们先给出直观的转换方法，然后再讨论树（森林）的一般方法。

1. 树到二叉树的转换

① 加线：同一双亲结点的所有孩子之间加一条连线；

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

③ 调整：调整结点位置，使之层次分明。

例如，如图 5-34 所示，(a)为一棵普通的树；(b)给相同双亲结点的兄弟结点之间添加连线；(c)对每个结点，保留到最左孩子的连线，抹去此结点到其它孩子的连线，至此已经将普通的树转换为一棵二叉树；(d)调整树中结点和连线，使层次分明，即为最终二叉树。

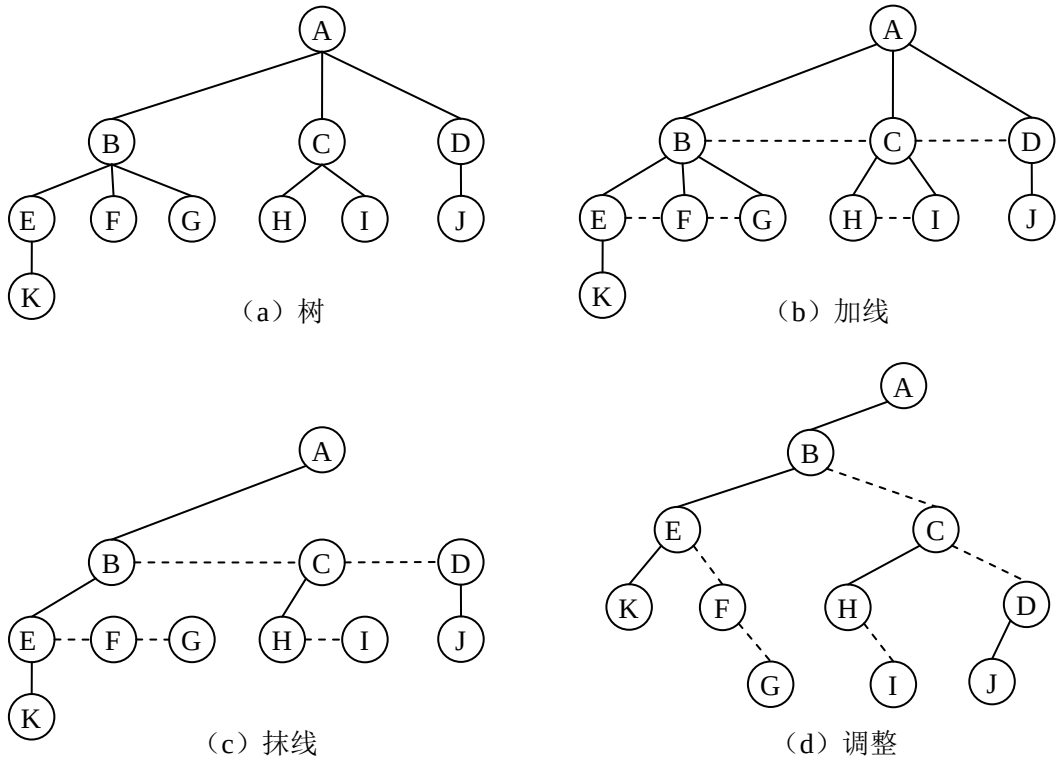


图 5-34　将树转化为二叉树的过程

普通树转换为二叉树具有这样的特点：(1) 转换后的二叉树，根结点下只有左子树，而没有右子树；(2) 转换后的二叉树，各结点的左孩子是其原来最左的孩子，右孩子则为其原先的下一个兄弟。这种方法产生的二叉树是维一的。

2. 森林到二叉树的转换

① 转换：将森林中的每棵树转换为二叉树；

② 连线：将每棵二叉树的根结点视为兄弟结点，加连线；

③ 调整：以最左边二叉树的根结点，作为最后的根结点， 调整结点位置，使之层次分明，即为最终二叉树。

例如，图 5-35 所示，为 3 棵树的森林转换为二叉树的过程。

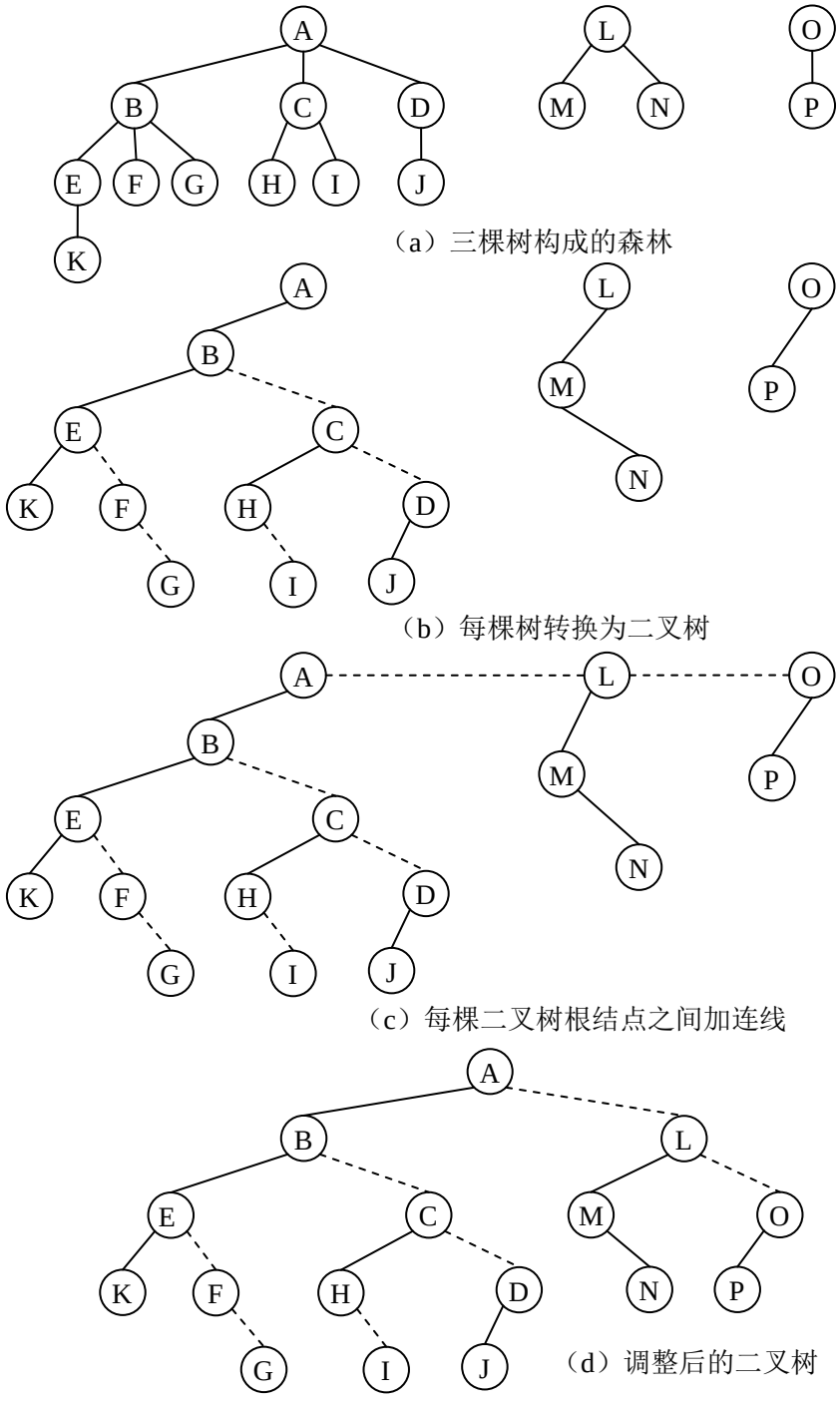


图 5-35　将森林转化为二叉树的过程

3. 森林（树）到二叉树转换的一般过程

由前面的存储结构可知，若要将森林转换为二叉树，需将森林中的每个结点进行这样的转换：左边指针指向其第一个孩子（作为左孩子形式），右边指针指向其下一个兄弟（作为右孩子形式）。森林中各棵树的根当作兄弟来转换。如果森林用有序表 T=(T_1，T_2，……，T_m)来表示，则将森林（树）T 转换为对应的二叉树 BT 的形式化描述如下：

如果 m=0，则 BT 为空；否则依次作如下操作：

① 将 T_1 的根作为 BT 的根；

② 将 T_1 的子树森林转换为 BT 的左子树；

③ 将(T_2，T_3，……，T_m)转换为 BT 的右子树。

由描述可知，转换过程可分三部分：即根、子树及兄弟森林三部分，而第二、第三部分的求解与原题求解类似，是一个递归的求解过程。

例如，对图 5-35（a）中 3 棵树构成的森林的转换的层次如图 5-36 所示，其中各层次中的一个整体用一个框框住，并用编号标注。其中“1”标记为根结点；“2”标记为左子树；“3”标记为右子树。限于篇幅，并没有将所有层次都标注出来。转换后的二叉树如图 5-35(d)所示。

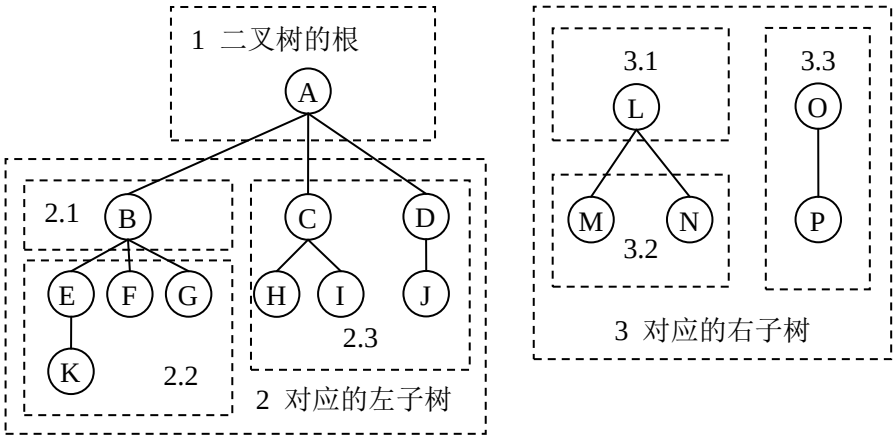


图 5-36　森林（树）转化为二叉树的层次标注过程

4. 二叉树到树的转换

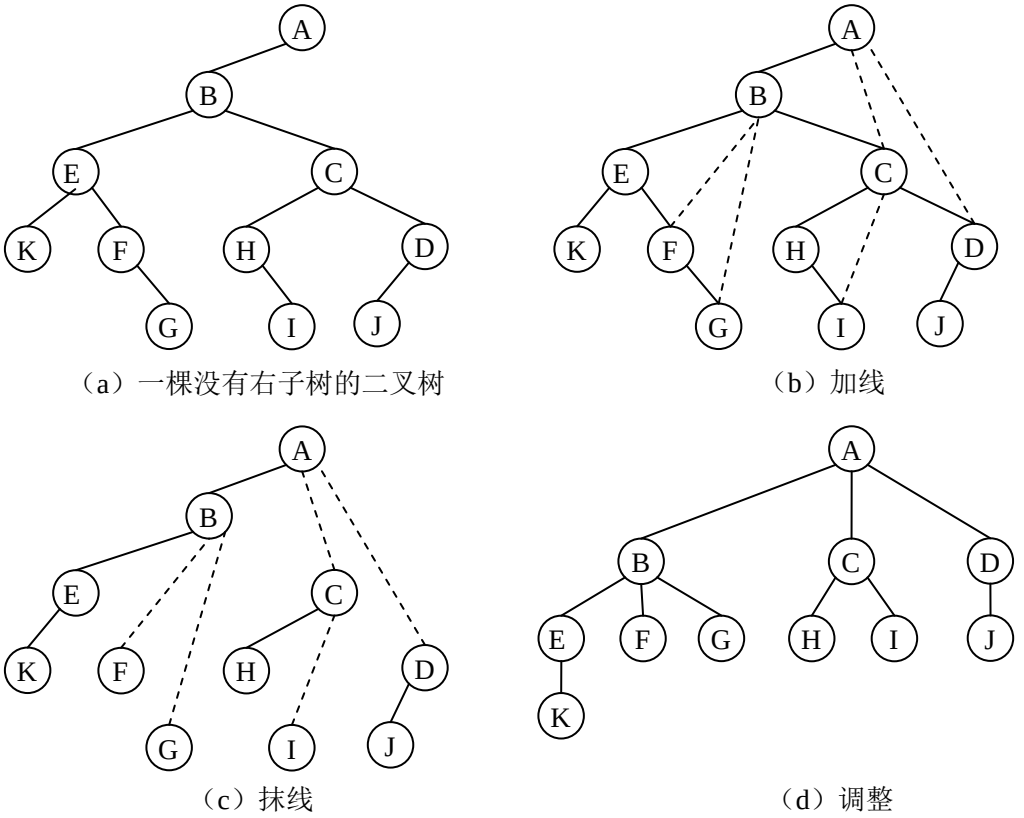
二叉树转换为树的先决条件是此二叉树根结点下只有左子树，没有右子树。转换步骤如下：

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

②抹线：抹掉二叉树中所有结点与其右孩子的连线 - 转为树；

③调整：调整结点位置，使之层次分明。

例如，图 5-37 所示，为将二叉树转换为树的过程。



（a）一棵没有右子树的二叉树　　（b）加线

（c）抹线　　（d）调整

图 5-37　将树转化为二叉树的过程

5. 二叉树到森林的转换

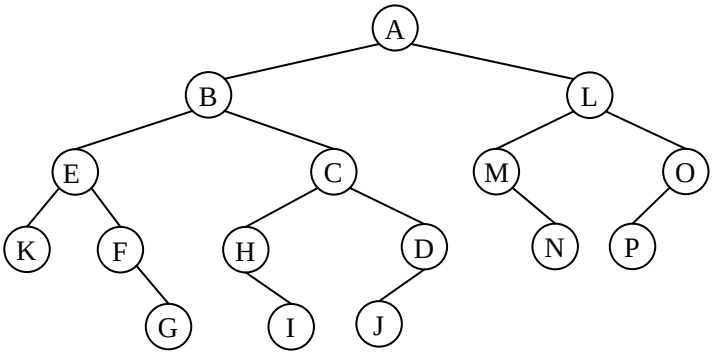
转换步骤如下：

①抹线：抹掉根结点与其右子树的连线；对分离出来的右子树，重复上面操作，直到分离出所有只有左子树的二叉树。

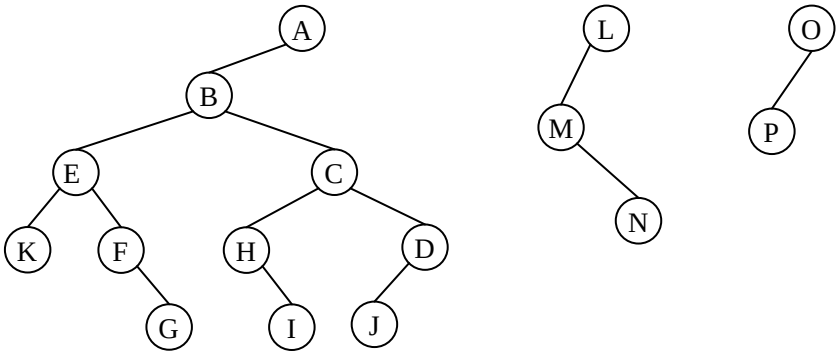
②转换：将每棵二叉树分别还原为树；

③调整：调整每棵树的结点位置，使之层次分明。

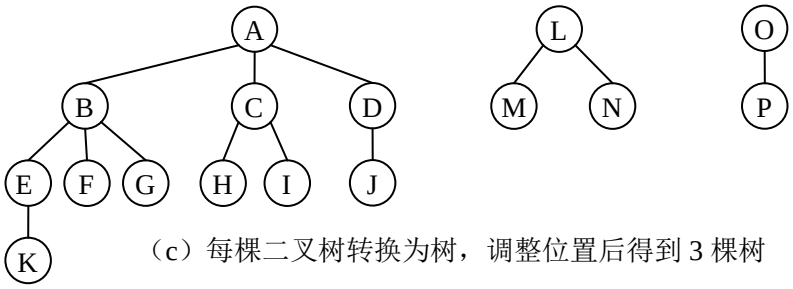
例如，图 5-38 所示为将二叉树转换为森林的过程。



（a）一棵二叉树



（b）抹线，分割出只有左子树的 3 棵二叉树



（c）每棵二叉树转换为树，调整位置后得到 3 棵树

图 5-38　将二叉树转换为森林的过程

6. 二叉树到森林（树）转换的一般过程

将二叉树 BT 转换为森林 T＝(T_1，T_2……T_m)的形式化描述如下：

若 BT 不空，则依次执行如下操作：

① 将 BT 的根转换为 T_1 的根；

② 将 BT 的左子树转换为 T_1 的子树森林；

③ 将 BT 的右子树转换为（T_2，……T_m）。

5.5.3 树（森林）的遍历

和二叉树等结构类似的是，遍历算法也是树（森林）的基本算法。根据访问根结点和子树中结点之间的次序关系，树的遍历可分为先序遍历和后序遍历。所谓先序遍历，就是每个结点在其所有子树之前访问；所谓后序遍历，就是每个结点在其所有子树之后访问。

1. 森林的先序遍历

森林 T=(T_1，T_2……T_m)，若 T 不空，则先序遍历依次执行如下操作：

① 访问 T_1 的根；

② 先序遍历 T_1 的子树森林；

③ 先序遍历森林（T_2，T_3……T_m）；

由此描述可知，森林的先序遍历过程也可采用前面所介绍的二叉树遍历的填空方法来描述，请有兴趣的读者自己练习。图 5-39 所示森林的先序遍历所产生的先序序列为：

ABEKFGCHIDJLMNOP

若将此结果和对应的二叉树的先序序列相比，发现两者完全相同。分析森林与二叉树间转换的描述可知，这是由其内在的对应关系所决定的，由归纳法也可证明这一点。

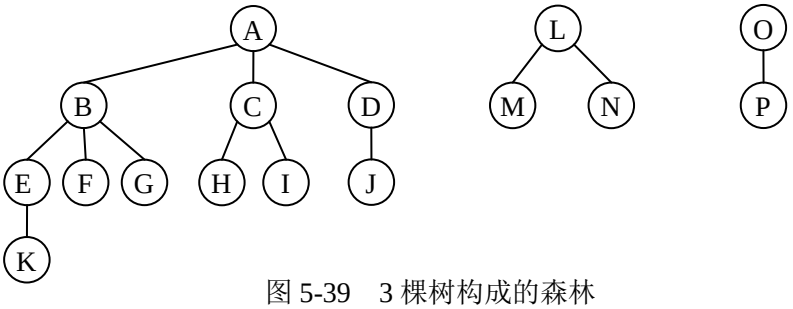


图 5-39　3 棵树构成的森林

如果采用前面的类型描述，则先序遍历森林的算法如下：

```
void preorder(Tnode *t)                 //先序遍历森林
  { if (t!=NULL)
    { visite(t);                        //访问结点
      preorder(t->firstson);            //先序遍历 t 的子树森林
      preorder(t->nextbrother);         //先序遍历 t 的兄弟森林
     }
  }
```

2. 森林的后序遍历

森林 T=(T_1，T_2……T_m)，如果 T 不空，则后序遍历依次执行如下操作：

① 后序遍历 T_1 的子树森林；

② 访问 T_1 的根；

③ 后序遍历森林（T_2，T_3……T_m）；

对图 5-39 中的森林的后序遍历序列为：KEFGBHICJDAMNLPO。

将这一序列与所对应的二叉树的中序序列相比，发现两者也相同！事实上，可以证明，对任意一个森林，其后序遍历序列与所对应的二叉树的中序序列相同。由后序遍历的描述及转换过程也可证明这一关系。也正因为如此，有些教科书中将这一遍历方法称为中序遍历，但笔者认为不太合适，因为这是针对树和森林而不是针对二叉树进行的遍历。

假设用二叉链表表示，则后序遍历算法如下：

```
void postorder(Tnode *t)
  {if   (t!=NULL)
     { postorder(t->firstson);          //后序遍历树 t 的子树森林
```

```
        visite(t);                          //访问树 t 的根结点
        postorder(t->nextbrother);          //后序遍历 t 的后续兄弟森林林
      }
    }
```

3. 树的层次遍历

首先访问第 1 层的根结点，然后依次访问每一层的结点，同一层按从左到右的顺序依次访问结点。

例如，图 5-40 中的树，层次遍历访问结点的序列为：ABCDEFGHIJK。

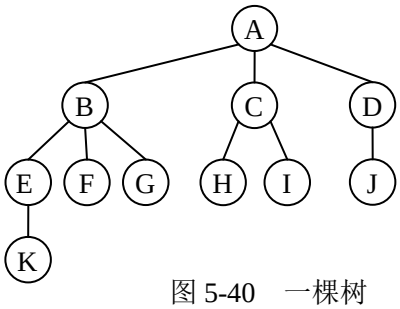


图 5-40　一棵树

5.6 哈夫曼树

在软件设计以及许多数据处理中，经常需要对数据进行压缩，以节省存储空间。压缩的基本原理是什么？有许多问题的求解方法不唯一，且各方法的求解效率有较大的差异，如何选择高效的求解方法？本节要介绍的哈夫曼树为此提供了一种基本的方法。下面先通过一个简单的例子来引入有关内容。

例：设计一个将百分成绩转换为等级制的算法，具体要求如下：

A：90～100

B：80～89

C：70～79

D：60～69

E：0～59

分析：对学过程序设计课程的读者来说，这一问题较为简单，有多种求解方法。例如图 5-41 中的几个判断的流程都能正确地求解。

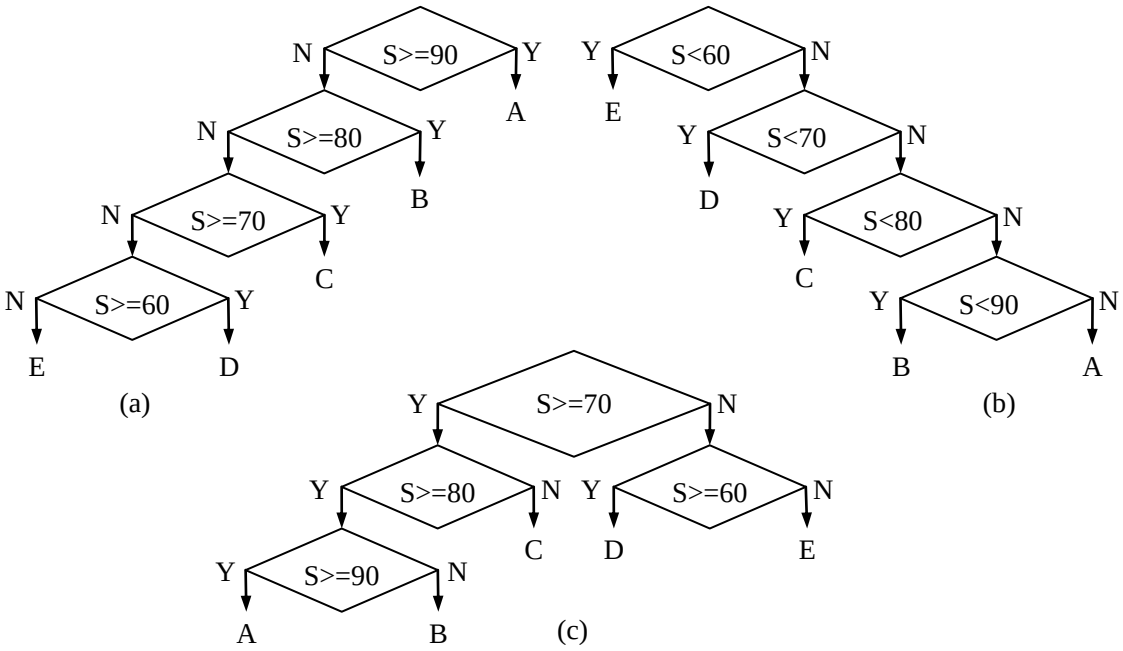


图 5-41 几个不同的判断流程

然而，不同的方法在时间性能上有很大的差异。可能有的读者对此感到不解，看不出这些算法在转换一个成绩表的时间性能上的差异。事实上，此处所谓的时间性能指的是统计意义上的，即对大批量数据处理时的时间性能。

例如，假设算法要转换 10000 个成绩，且各区间的分布如下：

90～100——5%

80～89——20%

70～79——30%

60～69——25%

0～59——20%

由这一分布可知，图 5-41（a）所示的判断流程中，每个处于 A 级的成绩只需判断一次即可；B 级的则需 2 次判断；C 级的需 3 次；D、E 级的需 4 次。因此，对 10000 个成绩来说，共需的判断次数是：

500×1＋2000×2＋3000×3＋2500×4＋2000×4＝31500 次

用图 5-41（b）来判断，需要的判断次数是：

500×4＋2000×4＋3000×3＋2500×2＋2000×1＝26000 次

用图 5-41（c）来判断，需要的判断次数是：

500×3＋2000×3＋3000×2＋2500×2＋2000×2＝22500 次

由此可知，不同的判断方法的判断次数是有差异的。现在问题是，如何判断才能保证时间最少？更进一步来说，对任一类似的问题，如何判断才能最省时间？

5.6.1 问题描述及求解方法

如果注意观察一下，就可发现：判断流程像一棵二叉树，其中分支（判断）结点对应

于二叉树的分支结点；而每个结论（推出什么）则对应于二叉树的叶子结点；得到一个结论所作的比较次数则是从根结点到该叶子结点的分支数（比层次数少 1）；每个结论成立的次数作为叶子结点的权值。

因此，上述问题可借助于二叉树这一模型来作更一般性的描述：

给定一组数值{w_1，w_2，……w_n}作为叶子结点的权值，构造一棵二叉树。如果Σw_iL_i最小，则称此二叉树为**最优二叉树**，也称**哈夫曼树**。并称Σw_iL_i为**带权路径长度**。

如何构造哈夫曼树呢？哈夫曼(Huffman)给出了一个带有一般规律的算法，俗称**哈夫曼算法**。描述如下：

(1)根据给定的 n 个权值{$w_1, w_2, \cdots, w_n$}，构成 n 棵二叉树的集合 T={$T_1, T_2, \cdots, T_n$}，其中每个 T_i 只有一个带权为 w_i 的根结点，其左右子树均空。

(2)从 T 中选两棵根结点的权值最小的二叉树，不妨设为 T_1' 、T_2' 作为左右子树构成一棵新的二叉树 T_1'，并且置新二叉树的根值为其左右子树的根结点的权值之和。

(3)将新二叉树 T_1' 并入到 T 中，同时从 T 中删除 T_1' 、T_2' 。

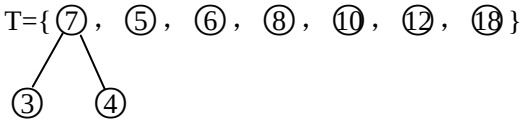
(4)重复(2)、(3)，直到 T 中只有一棵树为止。这棵树便是哈夫曼树。

例：以集合{3,4,5,6,8,10,12,18}为叶子结点的权值构造哈夫曼树，并计算其带权路径长度。

求解：按构造算法，首先将这些数变成单结点的二叉树集合

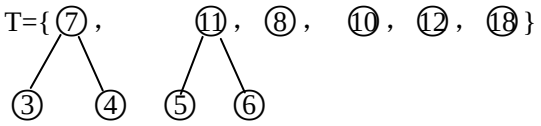
T={③，④，⑤，⑥，⑧，⑩，⑫，⑱}

然后从 T 中选出两个根值最小的二叉树{③，④}作为左右子树造出一棵新的二叉树，根为 T，同时从 T 中去掉这两棵子树，得结果如下。

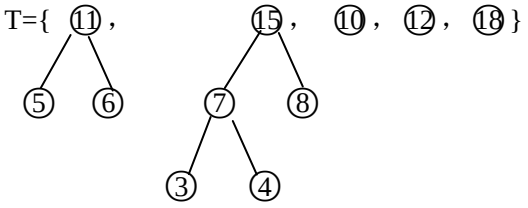


然后再重复这一操作过程，即选择最小的两个子树构造一棵新的二叉树，直到 T 中仅有一棵二叉树为止。操作过程如下：

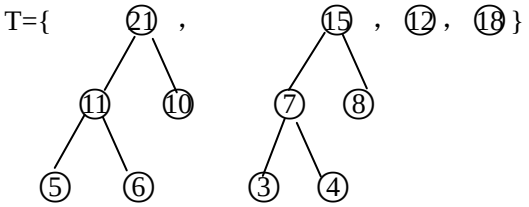
从 T 中选择根值为 5 和 6 的两棵树构成一棵树，结果如下：



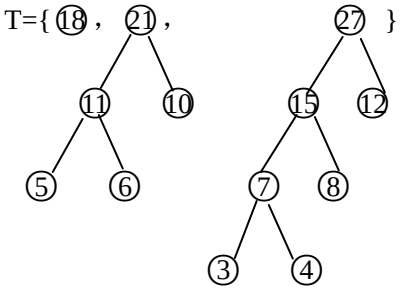
从 T 中选择根值为 7 和 8 的两棵树构成一棵树，结果如下：



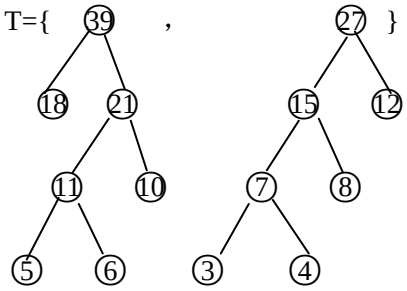
从 T 中选择根值为 10 和 11 的两棵树构成一棵树，结果如下：



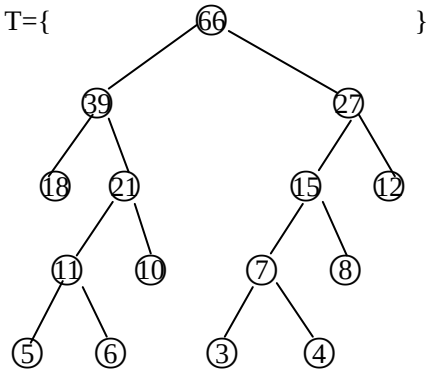
从 T 中选择根值为 12 和 15 的两棵树构成一棵树，结果如下：



从 T 中选择根值为 18 和 21 的两棵树构成一棵树，结果如下：



合并这两棵树构成一棵树即得到哈夫曼树，结果如下：



带权路径长度 WPL＝（3＋4＋5＋6）×4＋（8＋10）×3＋（12＋18）×2＝186

现在，我们对这一哈夫曼树作一个计算：将所有分支结点的值加起来，即

66＋39＋27＋21＋15＋11＋7＝186

这一值正好等于带权路径长度。可以证明，任意一棵哈夫曼树的带权路径长度均具有这一性质，即等于所有分支结点权值之和。根据这一性质求解 WPL 要快捷得多。

5.6.2 应用实例

下面通过一个实例来说明哈夫曼树的应用。

例：已知一个文件中仅有 8 个不同的字符，各字符出现的个数分别是 3、4、8、10、16、18、20、21。试重新为各字符编码，以节省存储空间。

解：常规情况下，一个系统中的字符的内部编码是等长的，例如，西文字符的长度是 8 位（bit），汉字的编码长度是 16 位。在这种情况下，无压缩可言。然而，如果给各字符的编码不等长，则可实现压缩。压缩的方法可借助于哈夫曼树来实现求解。首先，将所给出的各字符的个数作为权值来构造一棵哈夫曼树，然后对此编码可以得到哈夫曼编码，这些编码就可以作为各字符的新编码。具体求解如下：

（1）以所给出的数据集{3,4,8,10,16,18,20,21}所构造的哈夫曼树如图 5-42 所示。

（2）编码：约定每个结点的左分支标记为 0，右分支标记为 1，如图 5-42 所示。则从根结点到当前结点路径上经过的 0、1 序列即为当前结点的哈夫曼编码。例如，结点值为 10 的结点的编码为 001，结点值为 18 的结点的编码为 011。

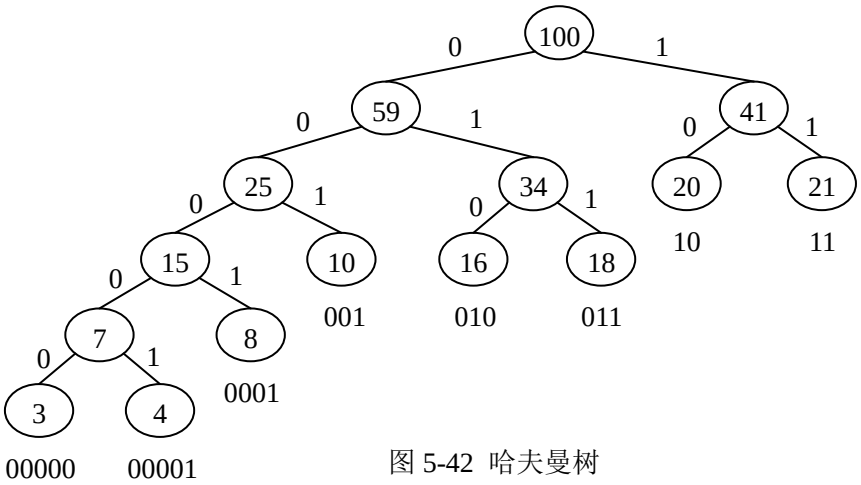


图 5-42 哈夫曼树

（3）新文件的长度：将各叶子结点所对应的编码作为对应字符的新编码可节省存储空间。其中出现个数为 3 的字符的编码为 00000，出现个数为 4 的字符的编码为 00001 等。依照这一方法来编码，可知重新编码的文件的长度为各字符的个数乘以其长度之积的和，也即为哈夫曼树的带权路径长度的值：

（3＋4）×5＋8×4＋（10＋16＋18）×3＋（20＋21）×2＝281

也就是说，在对文件中的字符按新的编码存储时，100 个字符所占用的位数共有 281 位。如果采用等长方式，则每个字符需要 3 位，因此共需要 300 位。由此可知，这一不等长编码能节省存储空间。如果字符数目较多，并且其频度有较大的差异，则其压缩程度会更明显。

本章小结

树结构是实际工作中常见的结构，是软件设计中常用的非线性结构，其中每个结点最多有一个前驱结点，但可能有多个后继结点。树和森林的有关概念和术语借助于现实生活中有关概念，因而便于理解。树和森林的存储有多种常用的方法，其中较为实用的方法是二叉链

表（或二叉树）存储法，因而与二叉树之间存在一一对应关系。对树和森林的遍历是树结构运算的基础，根据访问结点的次序可分为先序和后序两种方法，分别对应于所对应的二叉树的先序和中序遍历，因而可借助于二叉树的遍历运算来实现。

二叉树是本章以及本书中最重要的内容之一，二叉树的五个性质揭示了二叉树的主要特性，满二叉树和完全二叉树是两种特殊的二叉树。二叉树可采用顺序存储和二叉链表两种存储形式，其中前者只适用于规模较小或接近于完全二叉树的二叉树。二叉树的遍历有先序、中序和后序三种次序，是二叉树运算的基础。

为了方便地求解前驱和后继结点，在二叉链表结点中增设了前驱和后继指针，从而得到线索二叉树。在线索二叉树中，六个求解问题中的四个能方便地实现，但先序前驱和后序后继这两个问题不能求解。在线索二叉树中插入结点或子树是基本运算之一。二叉树线索化是将二叉树转换成线索二叉树的过程，线索化算法有一定的难度。

哈夫曼树是二叉树的应用之一，可用于数据压缩等问题的描述。