在图4-2的树中，节点 A 是根。节点 F 有一个父亲 A 并且有儿子 K、L 和 M。每一个节点可以有任意多个儿子，也可能是零个儿子。没有儿子的节点称为**树叶**(leaf)；上图中的树叶是 B、C、H、I、P、Q、K、L、M 和 N。具有相同父亲的节点为**兄弟**(siblings)；因此，K、L 和 M 都是兄弟。用类似的方法可以定义**祖父**(grandparent)和**孙子**(grandchild)关系。

从节点 n_1 到 n_k 的**路径**(path)定义为节点 n_1，n_2，…，n_k 的一个序列，使得对于 $1 \leqslant i < k$ 节点 n_i 是 n_{i+1} 的父亲。这条路径的**长**(length)是为该路径上的边的条数，即 $k-1$。从每一个节点到它自己有一条长为0的路径。注意，在一棵树中从根到每个节点恰好存在一条路径。

对任意节点 n_i，n_i 的**深度**(depth)为从根到 n_i 的唯一的路径的长。因此，根的深度为0。n_i 的**高**(height)是从 n_i 到一片树叶的最长路径的长。因此所有的树叶的高都是0。一棵树的高等于它的根的高。对于图4-2中的树，E 的深度为1而高为2；F 的深度为1而高也是1；该树的高为3。一棵树的深度等于它的最深的树叶的深度；该深度总是等于这棵树的高。

如果存在从 n_1 到 n_2 的一条路径，那么 n_1 是 n_2 的一位**祖先**(ancestor)而 n_2 是 n_1 的一个**后裔**(descendant)。如果 $n_1 \neq n_2$，那么 n_1 是 n_2 的**真祖先**(proper ancestor)而 n_2 是 n_1 的**真后裔**(proper descendant)。

4.1.1 树的实现

实现树的一种方法可以是在每一个节点除数据外还要有一些链，使得该节点的每一个儿子都有一个链指向它。然而，由于每个节点的儿子数可以变化很大并且事先不知道，因此在数据结构中建立到各(儿)子节点直接的链接是不可行的，因为这样会产生太多浪费的空间。实际上解决方法很简单：将每个节点的所有儿子都放在树节点的链表中。图4-3中的声明就是典型的声明。

图4-4指出一棵树如何用这种实现方法表示出来。图中向下的箭头是指向 firstChild(第一儿子)的链，而水平箭头是指向 nextSibling(下一兄弟)的链。因为 null 链太多了，所以没有把它们画出。

在图4-4的树中，节点 E 有一个链指向兄弟(F)，另一链指向儿子(I)，而有的节点这两种链都没有。

```
class TreeNode
{
    Object   element;
    TreeNode firstChild;
    TreeNode nextSibling;
}
```

图4-3 树节点的声明

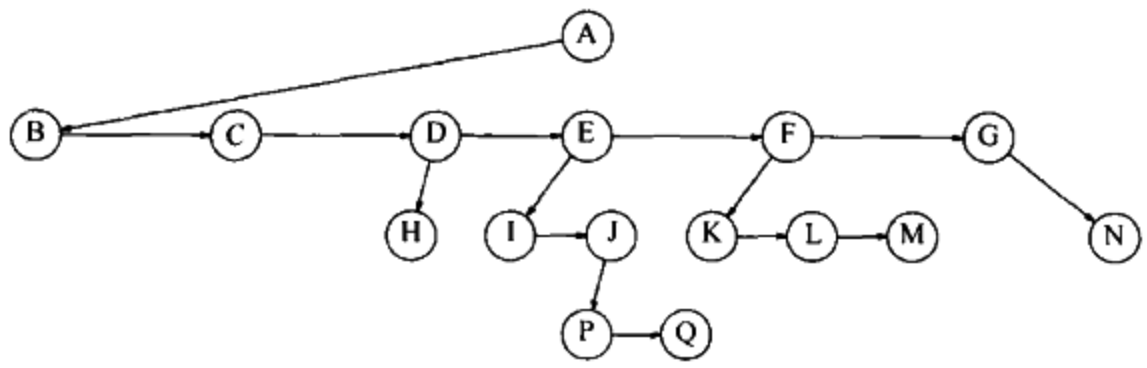

图4-4 在图4-2中所表示的树的第一儿子/下一兄弟表示法

4.1.2 树的遍历及应用

树有很多应用。流行的用法之一是包括 UNIX 和 DOS 在内的许多常用操作系统中的目录结构。图4-5是 UNIX 文件系统中一个典型的目录。

这个目录的根是/usr(名字后面的星号指出/usr本身就是一个目录)。/usr 有三个儿子：mark、alex 和 bill，它们自己也都是目录。因此，/usr 包含三个目录并且没有正规的文件。文件名/usr/mark/book/ch1.r 先后三次通过最左边的子节点而得到。在第一个/后的每个/都表示一条边；结果为一全**路径名**(pathname)。这个分级文件系统非常流行，因为它能够使得用户逻辑地组织数据。不仅如此，在不同目录下的两个文件还可以享有相同的名字，因为它们必然有从根开始的不