候，必须执行代价巨大的再散列这一步，它需要 $O(N)$次磁盘访问。

一种聪明的选择叫做**可扩散列**(extendible hashing)，它使得用两次磁盘访问执行一次查找。插入操作也需要很少的磁盘访问。

回忆第 4 章，B 树具有深度 $O(\log_{M/2} N)$。随着 M 的增长，B 树的深度降低。理论上我们可以选择 M 非常大，使得 B 树的深度为 1。此时，在第一次以后的任何查找都将花费一次磁盘访问，因为根节点很可能存放在主存中。这种方法的问题在于分支系数(branching factor)太高，以至于为了确定数据在哪片树叶上要进行大量的处理工作。如果运行这一步的时间可以减缩，那么我们就将有一个实际的方案。这正是可扩散列使用的策略。

现在假设我们的数据由几个 6 比特整数组成。图 5-24 显示这些数据的可扩散列格式。这里的"树"的根含有 4 个链，它们由这些数据的前两个比特确定。每片树叶有直到 $M=4$ 个元素。碰巧这里每片树叶中数据的前两个比特都是相同的；这由圆括号内的数指出。为了更正式，用 D 代表根所使用的比特数，有时称其为**目录**(directory)。于是，目录中的项数为 2^D。d_L 为树叶 L 所有元素共有的最高位的比特位数。d_L 将依赖于特定的树叶，因此 $d_L \leqslant D$。

设欲插入关键字 100 100。它将进入第三片树叶，但是第三片树叶已经满了，没有空间存放它。因此我们将这片树叶分裂成两片树叶，它们由前三个比特确定。这需要将目录的大小增加到 3。这些变化由图 5-25 反映出来。

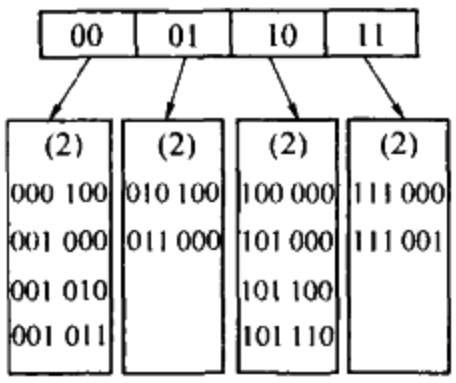

图 5-24　可扩散列：原始数据

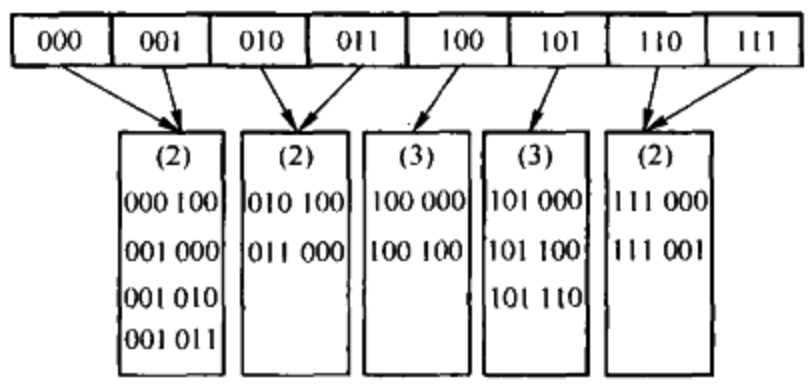

图 5-25　可扩散列：在 100 100 插入及目录分裂后

注意，所有未被分裂的树叶现在各由两个相邻目录项所指。因此，虽然整个目录被重写，但是其他树叶都没有被实际访问。

如果现在插入关键字 000 000，那么第一片树叶就要被分裂，生成 $d_L=3$ 的两片树叶。由于 $D=3$，故在目录中所作的唯一变化是 000 和 001 两个链的更新。见图 5-26。

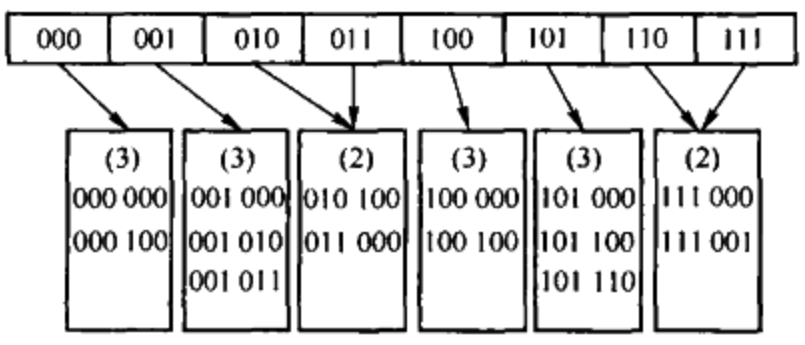

图 5-26　可扩散列：在 000 000 插入及树叶分裂后

这个非常简单的方法提供了对大型数据库 insert 操作和查找操作的快速存取时间。这里，还有一些重要细节我们尚未考虑。

首先，有可能当一片树叶的元素有多于 $D+1$ 个前导比特位相同时需要多次目录分裂。例