行的；它用到队列，而不使用递归所默示的栈。

4.7 B 树

迄今为止，我们始终假设可以把整个数据结构存储到计算机的主存中。可是，如果数据更多装不下主存，那么这就意味着必须把数据结构放到磁盘上。此时，因为大 O 模型不再适用，所以导致游戏规则发生了变化。

问题在于，大 O 分析假设所有的操作耗时都是相等的。然而，现在这种假设就不合适了，特别是涉及磁盘 I/O 的时候。例如，一台 500 – MIPS 的机器可能每秒执行 5 亿条指令。这是相当快的，主要是因为速度主要依赖于电的特性。另一方面，磁盘操作是机械运动，它的速度主要依赖于转动磁盘和移动磁头的时间。许多磁盘以 7200RPM 旋转。即 1 分钟转 7200 转；因此，1 转占用 1/120 秒，或即 8.3 毫秒。平均可以认为磁盘转到一半的时候发现我们要寻找的信息，但这又被移动磁盘磁头的时间抵消，因此我们得到访问时间为 8.3 毫秒(这是非常宽松的估计；9～11 毫秒的访问时间更为普通)。因此，每秒大约可以进行 120 次磁盘访问。若不和处理器的速度比较，那么这听起来还是相当不错的。可是考虑到处理器的速度，5 亿条指令却花费相当于 120 次磁盘访问的时间。换句话说，一次磁盘访问的价值大约是 40 万条指令。当然，这里每一个数据都是粗略的计算，不过相对速度还是相当清楚的：磁盘访问的代价太高了。不仅如此，处理器的速度还在以比磁盘速度快得多的速度增长(增长相当快的是磁盘容量的大小)。因此，为了节省一次磁盘访问，我们愿意进行大量的计算。几乎在所有的情况下，控制运行时间的都是磁盘访问的次数。于是，如果把磁盘访问次数减少一半，那么运行时间也将减半。

在磁盘上，典型的查找树执行如下：设想要访问佛罗里达州公民的驾驶记录。假设有 1 千万项，每一个关键字是 32 字节(代表一个名字)，而一个记录是 256 个字节。假设这些数据不能都装入主存，而我们是正在使用系统的 20 个用户中的一个(因此有 1/20 的资源)。这样，在 1 秒内，我们可以执行 2 千 5 百万次指令，或者执行 6 次磁盘访问。

不平衡的二叉查找树是一个灾难。在最坏情形下它具有线性的深度，从而可能需要 1 千万次磁盘访问。平均来看，一次成功的查找可能需要 $1.38\log N$ 次磁盘访问，由于 $\log 10\,000\,000 \approx 24$，因此平均一次查找需要 32 次磁盘访问，或 5 秒的时间。在一棵典型的随机构造的树中，我们预料会有一些节点的深度要深 3 倍；它们需要大约 100 次磁盘访问，或 16 秒的时间。AVL 树多少要好一些。$1.44\log N$ 的最坏情形不可能发生，典型的情形是非常接近于 $\log N$。这样，一棵 AVL 树平均将使用大约 25 次磁盘访问，需要的时间是 4 秒。

我们想要把磁盘访问次数减小到一个非常小的常数，比如 3 或 4；而且我们愿意写一个复杂的程序来做这件事，因为在合理情况下机器指令基本上是不占时间的。由于典型的 AVL 树接近到最优的高度，因此应该清楚的是，二叉查找树是不可行的。使用二叉查找树我们不能行进到低于 $\log N$。解法直觉上看是简单的：如果有更多的分支，那么就有更少的高度。这样，31 个节点的理想二叉树(perfect binary tree)有 5 层，而 31 个节点的 5 叉树则只有 3 层，如图 4-58 所示。一棵 M 叉查找树(M-ary search tree)可以有 M 路分支。随着分支增加，树的深度在减少。一棵完全二叉树(complete binary tree)的高度大约为 $\log_2 N$，而一棵完全 M 叉树(complete M-ary tree)的高度大约是 $\log_M N$。

我们可以以与建立二叉查找树大致相同的方式建立 M 叉查找树。在二叉查找树中，需要一个关键字来决定两个分支到底取用哪个分支；而在 M 叉查找树中需要 $M1$ 个关键字来决定选取哪个分支。为使这种方案在最坏的情形下有效，需要保证 M 叉查找树以某种方式得到平衡。否则，像二叉查找树，它可能退化成一个链表。实际上，我们甚至想要更加限制性的平衡条件，即不想要