如，从原先的例子开始，$D=2$，如果插入 111 010、111 011，并在最后插入 111 100，那么目录大小必须增加到 4 以区分 5 个关键字。这是一个容易考虑到的细节，但是千万不要忘记它。其次，存在重复关键字(duplicate keys)的可能性；若存在多于 M 个重复关键字，则该算法根本无效。此时，需要作出某些其他的安排。

上述可能性指出，这些比特完全随机是相当重要的，它可以通过把那些关键字散列到合理的长整数来实现。

最后，我们介绍可扩散列的某些性能，这些性能是经过非常困难的分析后得到的。这些结果基于合理的假设：位模式(bit pattern)是均匀分布的。

树叶的期望个数为$(N/M)\log_2 e$。因此，平均树叶满的程度为 $\ln 2=0.69$。这和 B 树是一样的，其实这完全不奇怪，因为对于两种数据结构，都是当第$(M+1)$项被添加进来时一些新的节点被建立。

更令人惊奇的结果是目录的期望大小(换句话说即 2^D)为 $O(N^{1+1/M}/M)$。如果 M 很小，那么目录可能过大。在这种情况下，我们可以让树叶包含指向记录的链而不是实际的记录，这样可以增加 M 的值。为了维持更小的目录，这就对每次查找操作增加了第二次磁盘访问。如果目录太大装不进主存，那么第二次磁盘访问无论如何也还是需要的。

小结

散列表可以用来以常数平均时间实现 insert 和查找操作。当使用散列表时注意诸如装填因子这样的细节是特别重要的，否则时间界将不再有效。当关键字不是短的串或整数时，仔细选择散列函数也是很重要的。

对于分离链接散列法，虽然装填因子不很大时性能并不明显降低，但装填因子还是应该接近于 1。对于探测散列算法，除非完全不可避免，否则装填因子不应该超过 0.5。如果使用线性探测，那么性能随着装填因子接近于 1 将急速下降。再散列运算可以通过使散列表增长(和收缩)来实现，这样将会保持合理的装填因子。对于空间紧缺并且不可能声明巨大散列表的情况，这是很重要的。

二叉查找树也可以用来实现 insert 和 contains 运算。虽然平均时间界为 $O(\log N)$，但是二叉查找树也支持那些需要序从而功能更强大的例程。使用散列表不可能找出最小元素。除非准确知道一个字符串，否则散列表也不可能有效地查找它。二叉查找树可以迅速找到在一定范围内的所有项；散列表是做不到的。此外，$O(\log N)$这个时间界也不必比 $O(1)$大很多，这特别是因为使用查找树不需要乘法和除法。

另一方面，散列的最坏情况一般来自于实现的错误，而有序的输入却可能使二叉树运行得很差。平衡查找树实现的代价相当高，因此，如果不需要序的信息以及对输入是否被排序存有怀疑，那么就应该选择散列这种数据结构。

散列有着丰富的应用。编译器使用散列表跟踪源代码中声明的变量。这种数据结构叫做**符号表**(symbol table)。散列表是这种问题的理想应用。标识符一般都不长，因此其散列函数能够迅速被算出，而按字母顺序排列变量通常没有必要。

散列表对于任何图论问题都是有用的，在图论问题中，节点都有实际的名字而不是数字。这里，当输入被读入的时候，顶点按照它们出现的顺序从 1 开始被指定一些整数。再有，输入很可能有一组一组依字母顺序排列的项。例如，顶点可以是计算机。此时，如果一个特定的计算中心把它的计算机列表，成为 ibm1，ibm2，ibm3，…，那么，若使用查找树则在效率方面可能会有戏剧性的效果。