第5章 查 找

在实际应用中，经常需要在数据集中查找某个数据，如在电话本中查找某个人的电话，在网上商城中查找某个产品等。查找是数据结构中一种重要的运算，也是许多计算机应用程序的核心功能。

查找就是在一组记录集合中找到关键字等于给定值的某个记录或者找到属性值符合条件的某些记录，若表中存在这样的记录表示查找成功，此时查找的结果是给出这个记录的全部信息或指示出该记录在数据集中的位置；若数据集中不存在关键字等于给定值的记录，表示查找不成功。其中关键字是数据元素（或记录）中某个数据项的值，用它可以标识（或识别）数据元素（或记录）。若该关键字可以唯一地标识一个数据元素，则该关键字称为主关键字。能够标识若干数据元素的关键字为次关键字。

人们最熟悉的查找策略是顺序查找，即在数据集中从第一个元素开始，依次比较，直到找到目标数据（查找成功）或者将所有元素都比较完（查找失败）。比如在一副扑克牌中找到红心 A，就可以依次检查每一张扑克牌，最多需要检查 54 张牌。但是当数据集规模很大时，比如互联网中的网页接近 20 亿个（2000 年美国互联网委员会估计），如果采用顺序查找策略来查找某个页面需要花费几天的时间，这种查找策略显然不适用。

为了提高在大规模数据中查找的效率，往往需要对数据的存储进行特殊处理。最常见的方法是建立索引和预排序。索引可以理解为一个特殊的目录。索引的建立需要消耗一定的存储空间，但是查找时可以充分利用索引的信息大大地提高查找效率。预排序是指在查找前对数据元素进行排序，对于排好序的数据进行查找可以采用有效的折半查找方式。

查找分为静态查找和动态查找两种类型。静态查找是指在查找过程中不更改数据集中的数据，主要包括顺序查找法、折半查找法和分块查找法；而动态查找指在查找不成功时将要查找的数据添加到数据集中，主要包括二叉搜索树、平衡二叉搜索树、B-树、B+树。

查找过程中的基本操作是进行关键字的比较，因此在讨论各种查找技术时通过分析关键字的比较次数来度量各种查找技术的效率。对于含有 n 个记录的数据集 $\{data_1, data_2, ..., data_n\}$，如果要找的记录是 $data_i$ 的概率是 P_i，且 $sum(P_i)=1$，并且查找到 $data_i$ 需要经过 C_i 次比较，则定义该查找技术在查找成功时的平均查找长度为：

$$ASL=\sum_{i=1}^{n}P_iC_i$$

上述查找方法建立在关键字“比较”的基础上，用来处理数据记录的存储位置和记录的关键字不存在确定关系的数据集中的查找问题。这些查找方法在数据集中查找记录时需要进行一系列关键字的比较。而理想的情况是希望不经过任何比较，一次找到目标

记录，这就必须在记录的存储位置和它的关键字之间建立一个确定的对应关系，使每个关键字和唯一的存储位置相对应。从而在查找时，只要根据这个对应关系就可以确定相应记录的存储位置，不需要进行关键字的比较。这种对应关系称为散列函数，这种查找技术称为散列查找。

本章主要介绍静态查找、动态查找和散列查找三类查找技术的思想、实现，并分析各种查找技术在查找成功时的平均查找长度和查找不成功时的关键字比较次数。

5.1 静 态 查 找

5.1.1 顺序查找法

顺序查找的查找过程为：从表中第一个记录开始，逐个进行记录的关键字和给定值的比较，若某个记录的关键字和给定值比较相等，则查找成功；否则，若至最后一个记录的关键字与给定值都不等，表明表中没有所查记录，查找不成功。表中数据元素之间不必拥有逻辑关系，即它们在表中可以任意排序。

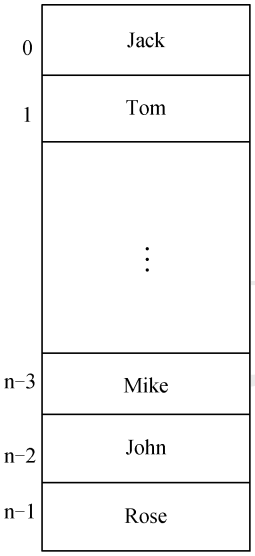


图 5-1 顺序查找示例

比如有 n 个不相同的学生名字存放在如下的一维数组中，则查找“Jack”时需要经过 1 次比较，而查找“Rose”需要经过 n 次比较，如图 5-1 所示。

当数据采用顺序存储结构（数组）存储时，相应的顺序查找算法如例 5.1 所示。

【例 5.1】基于顺序表的顺序查找算法。

```
        //顺序查找过程，Array[]为待查找的数据记录集合，n 为集合的记录个数，key 为要查找的数据记录
        int Search(T Array[], T key,int n)
        {
            for (int index = 0; index < n; index ++)
            {//从数组起始位置遍历数组，length 表示数组长度
                if( Array[index] == key)//如果数组元素等于待查找关键字
                {
                    return index;       //查找成功，返回该数组元素所在的位置
                }
            }
            return -1;                  //查找不成功，返回-1
        }
```

分析上面的顺序查找算法，如果查找的数据记录 key 在顺序表的第 i（$0 \leqslant i < n$）个位置时，需要执行 i+1 次关键字的比较，如果查找顺序表中每个记录的概率相等（即$\frac{1}{n}$），

则等概率下查找成功的平均查找长度是：

$$ASL=\sum_{i=0}^{n-1}P_iC_i=\sum_{i=0}^{n-1}\frac{1}{n}(i+1)=\frac{1}{n}\sum_{i=1}^{n}i=\frac{n+1}{2}$$

可以看出，成功查找的平均比较次数约为线性表长度的一半。如果查找的关键字不在线性表里，则需要进行 n+1 次比较才能确定查找失败。

顺序查找的优点是：对表的特性没有要求，数据元素可以任意排序。插入数据元素可以直接加到表尾，时间复杂度为O(l)。缺点是：顺序查找的平均查找长度较大，平均和最坏情况下时间复杂度都是O(n)，当数据规模较大时，查找效率低。

采用顺序查找时，数据记录也可以采用单链表来存储，其查找成功的平均查找长度和查找不成功的比较次数与顺序表的顺序查找相同。

5.1.2 折半查找法

顺序查找易实现，易分析，适用于在小规模数据中进行查找的情况。如果数据集的规模很大时，就要寻找一个较快的算法。折半查找法（也称为二分查找法）就是一种较快的查找方法，但是查找的前提是数据记录有序地存储在线性表中。对于任何一个线性表，若其中的所有数据元素按关键字的某种次序进行不增或不减的排序就称为有序表。例如，关键字为整数或实数，则数值的大小是一种次序关系；若关键字为字符串，则字典顺序是一种次序关系。因此折半查找也称为有序表的查找。

对有序表进行查找的时候，由于数据元素的有序性，将表中一个元素 data[i] 的关键字 k 与查询元素的关键字 key 进行比较时，比较结果分为三种情况（以递增顺序的线性表为例）：

1）k = key，查询成功，data[i]为待查元素；

2）k > key，说明待查元素排在 data[i]之前；

3）k < key，说明待查元素排在 data[i]之后。

因此在一次比较之后，若没有找到待查元素，则根据比较结果缩小查询范围。折半查找法的基本思想是：每次将待查区间中间位置上的数据元素的关键字与给定值 key 进行比较，若相等则查找成功；若小于给定值 key，则将查找范围缩小到中间位置的右边区域；若大于给定值 key，则将查找范围缩小到中间位置的左边区域；在新的区间内重复上述过程，直到查找成功或查找范围长度为 0（查找不成功）为止。

例如，已知如下具有 11 个数据记录的有序表（关键字即为数据元素的值）int Array[11]={10，15，17，23，38，46，54，65，82，89，95}，利用折半查找法查找关键字为 23 和 85 的数据记录。

首先给有序表的记录进行顺序编号 1-11，使用两个标记 left 和 right 分别指示当前查找范围的左边记录编号和右边记录编号，用标记 mid 指示当前查找范围的中间记录编号，即 $mid=\left\lfloor\frac{(left+right)}{2}\right\rfloor$。在此例中，left 和 right 的初始值分别为 1 和 11，即[1,11]

为初始的查找范围。

在查找 key=23 时，首先在初始查找范围内计算中间位置 $\text{mid}=\left\lfloor\frac{(\text{left}+\text{right})}{2}\right\rfloor=\frac{1+11}{2}=6$。

编号:	1	2	3	4	5	6	7	8	9	10	11
数据:	10	13	17	23	38	46	54	65	82	89	92
	↑left					↑ mid					↑right

查看中间位置的数据元素 Array[mid]=46，大于查找的关键字 23，说明待查找的关键字 23 在[left,mid-1]标记的记录范围之间，将 right 重新定位到 mid-1 位置处，更新查找范围。

编号 :	1	2	3	4	5	6	7	8	9	10	11
数据 :	10	13	17	23	38	46	54	65	82	89	92
	↑left				↑right						

在新的查找范围（当前查找范围）内，重新计算新的中间位置元素 $\text{mid}=\left\lfloor\frac{(\text{left}+\text{right})}{2}\right\rfloor=\frac{1+5}{2}=3$。

编号 :	1	2	3	4	5	6	7	8	9	10	11
数据 :	10	13	17	23	38	46	54	65	82	89	92
	↑left		↑mid		↑right						

比较当前查找范围内的中间位置的数据元素 Array[mid]=17 和待查关键字 23，因为 23 > Array[mid]，说明待查记录 23 只可能在[mid+1,right]所标记的数据记录范围内出现，将 left 更新为 mid+1，进一步缩小查找范围。

编号 :	1	2	3	4	5	6	7	8	9	10	11
数据 :	10	13	17	23	38	46	54	65	82	89	92
				↑left	↑right						

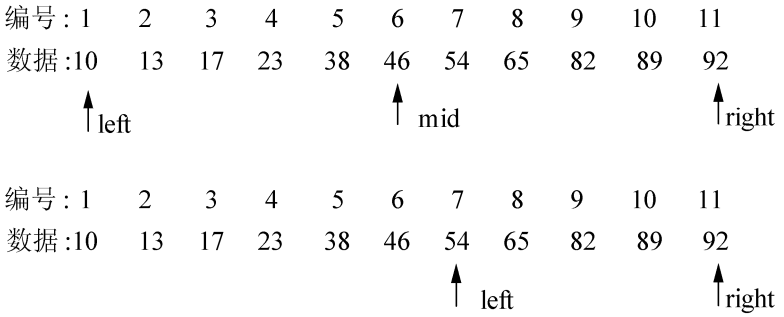
计算当前查找范围的中间位置 mid=4，比较 Array[mid] =23，正好与所查关键字 23 相同，查找成功。共进行了 3 次关键字的比较。

编号 :	1	2	3	4	5	6	7	8	9	10	11
数据 :	10	13	17	23	38	46	54	65	82	89	92
				↑left	↑right						
				↑mid							

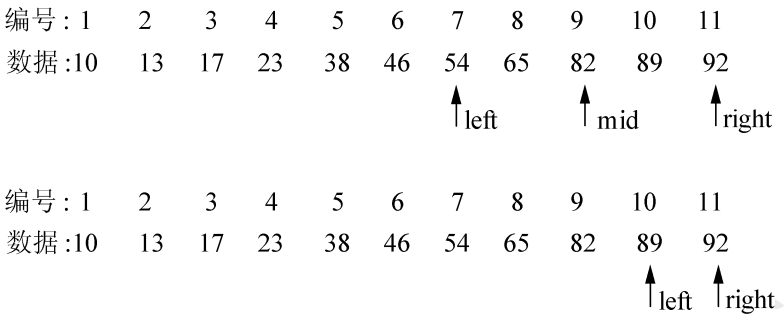
在查找关键字 key=85 时，初始的查找范围依然是整个数据记录集合，即 left=1，

right=11。

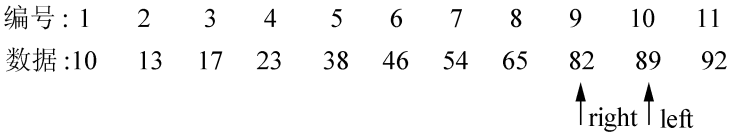
首先查看当前查找范围的中间 middle=6 元素，因为 key > Array[mid]，所以修正查找范围的左边界，即令 left=mid+1=7。



在当前查找范围[7，11]中重新计算 mid 的值为 9，且 key > Array[mid]，继续修正查找范围的左边界，即 left=mid+1=10。



在当前查找范围[10，11]中重新计算 mid 的值为 10，且 key < Array[mid]，修正查找范围的右边界，即 right=mid-1=9。



检查当前查找范围[10，9]，发现 right<left，说明数据集中不存在 85，确定查找不成功，结束查找。共经过了 3 次比较就可以确定在这个具有 11 个记录的有序表中不存在记录 85。

以上的例子也说明在折半查找中，需要对数据元素通过编号进行访问即随机访问，因此数据只能采用顺序存储结构（数组）来存储。

折半查找的算法实现如例 5.2 所示。

【例 5.2】折半查找算法。

```
template <class T>
int BiSearch(T Array[], T key, int n)
{ //折半查找过程，Array[0,…,n-1]为待查找的有序数据记录，key 为查找的记录
```

```
        int left = 0;           //定义查找范围的左端
        int right = n-1;        //定义查找范围的右端，n 表示数组的长度
        int mid;                //定义查找范围的中间点
        while (left <= right)
        {//如果查找范围有效，则进行查找，否则结束查找
            mid = (left + right)/2;
            if (key < Array[mid])
            { //将查找范围缩小到中间元素的左边
                right = mid - 1;
            }
            else if (key > Array[mid])
            {//将查找范围缩小到中间元素的右边
                left = mid + 1;
            }
            else
                return mid;         //查找成功，返回该元素所在位置
        }
        return -1;                  //查找不成功，返回-1
    }
```

折半查找是以位于当前查找范围的中间位置的元素和待查找元素比较，若相等，则查找成功，若不等，则缩小查找范围，直至新的查找范围的中间元素等于待查找元素或者查找范围无效时为止。

利用判定树可以分析出折半查找的性能。

折半查找法的查找过程可以用二叉树来描述，树中每个结点表示算法中参与比较的数据元素编号，这种二叉树被叫做判定树。

性质 1：具有 n 个结点的判定树的高度为$\lfloor \log_2 n \rfloor+1$。（只有根结点的判定树的高度为 1）

证明：虽然判定树不是完全二叉树，但由于折半查找时，每次都是中间元素与待查元素比较，因此判定树的叶子结点所在层次之差最多为 1。所以 n 个结点的判定树的高度和 n 个结点的完全二叉树的高度$\lfloor \log_2 n \rfloor+1$相同。

如上面所述的折半查找的例子，相应的判定树如图 5-2 所示。在查找 23 时，首先与根结点标记的记录进行比较，根据比较的结果，由于 23<Array[6]继续和 Array[3]标记的数据进行比较，由于 23> Array[3]，继续和 Array[4]比较，经过比较后确定查找成功。这个比较过程是由根到结点④的路径，进行比较的次数恰为路径上结点个数或为结点④所处的层数（根结点的层数为 1）。类似的，找到有序表中任一记录的过程就是走一条从根结点到所查找记录在树中的结点的路径。查找该记录的比较次数为该记录在树中的层次数。因此，折半查找法查找成功时比较次数最多为判定树的高度，即$\lfloor \log_2 n \rfloor+1$。

折半查找法在查找不成功时仍是沿着一条从根结点到叶子结点的路径进行比较，因此最多进行$\lfloor \log_2 n \rfloor+1$次关键字的比较。

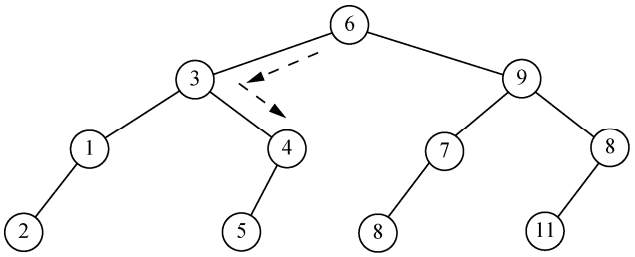


图 5-2　描述折半查找法的判定树及查找 23 的过程

为了计算折半查找的平均查找长度，假设有序表的长度为 $n=2^h-1$，其中 h 为对应判定树的高度，则此判定树一定是一棵满二叉树。所以，判定树中层次为 1 的结点有 1 个，层次为 2 的结点有 2 个，……，层次为 h 的结点有 2^{h-1} 个。假设表中每个记录的查找概率相等（$P_i=\frac{1}{n}$），则折半查找法在查找成功时的平均查找长度为：

$$\begin{aligned}ASL=\sum_{i=1}^{h}P_iC_i&=\frac{1}{n}\sum_{j=1}^{h}j\cdot 2^{j-1}=\frac{1}{n}(\sum_{i=0}^{h-1}2^i+2\sum_{i=0}^{h-2}2^i+\ldots+2^{h-1}\sum_{i=0}^{0}2^i)\\&=\frac{1}{n}[h\cdot 2^h-(2^0+2^1+\ldots+2^{h-1})]\\&=\frac{1}{n}[(h-1)2^h+1]\\&=\frac{1}{n}[(n+1)(\log_2(n+1)-1)+1]\\&=\frac{(n+1)}{n}\log_2(n+1)-1\end{aligned}$$

对任意较大的 n(n>50)，可以有如下的近似结果：

$$ASL=\log_2(n+1)-1$$

显然，折半查找法的查找效率比顺序查找高，但折半查找只适用于顺序存储的有序表，而且向有序表中新增或删除数据时操作比较复杂。

5.1.3　分块查找

分块查找又称索引顺序查找，利用折半查找的思想改进顺序查找，从而既有较快的查找速度又便于数据集的灵活更改。在分块查找中，首先将 n 个数据元素划分为 m 块（m≤n）。每一块中的数据不必有序，但块与块之间必须“按块有序”：即第 1 块中任一元素的关键字都必须小于第 2 块中任一元素的关键字；而第 2 块中任一元素又都必须小于第 3 块中的任一元素；…；第 m-1 块中任一元素又都必须小于第 m 块中的任一元素。对每个数据块建立一个索引项，形成具有 m 个索引项的索引表。索引项包括两项内容：块中的最大关键字（关键字项），该块中第一个记录的位置（指针项）。索引表中的各个索引项按照关键字有序。图 5-3 给出了一个顺序表和相应的索引表，其中每块具有 5 个记录。

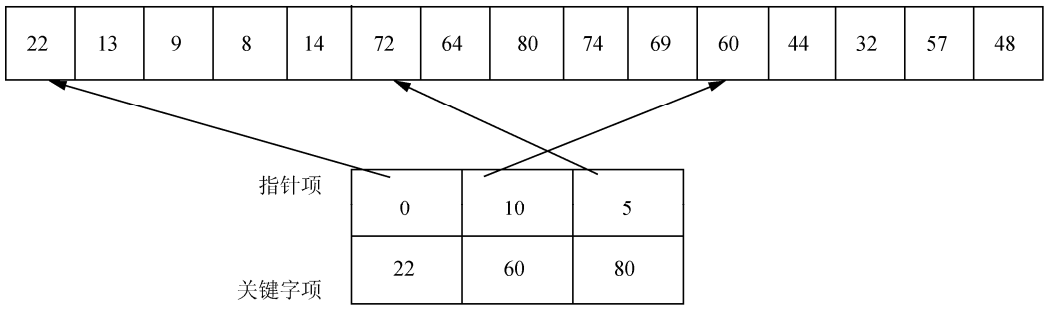


图 5-3　分块查找示例

分块查找分两个阶段。第一阶段，根据索引表确定查找的记录所在的数据块。由于索引表按照关键字有序，因此也可以利用折半查找法。第二阶段，在已确定的数据块中用顺序查找法进行查找。如在上面的顺序表中查找 key=32，先利用顺序查找法或者折半查找法确定 key 所在的数据块。由于 22<key<60，所以关键字为 key 的记录如果存在，则只能在第 2 个数据块中。根据第 2 个索引项的指针值，定位第 2 个块的第一个记录的下标为 10，然后利用顺序查找法依次检查该块中记录，直到找到为止（查找成功）。如果该数据集中不存在关键字为 key 的记录（如 key=77），根据索引表先确定可能出现在第 k 块中，将这个数据块的所有记录都检查完时就可以确定查找失败。

分块查找的平均查找长度为

$$ASL = Lb + Lw$$

其中 Lb 是确定待查记录所在块的平均查找长度，Lw 是在块中查找待查记录的平均查找长度。如果数据集有 n 个记录，平均分成 k 块，每块含有 m 个记录，即 $m=\frac{n}{k}$。设每个记录的查找概率相等，则每个索引项的查找概率为 $\frac{1}{k}$，块中每个记录的查找概率为 $\frac{1}{m}$。若在索引表中使用折半查找，则平均查找长度为 $\left\lfloor \log_2 \frac{n}{m} \right\rfloor + 1$；在每个块中使用顺序查找的平均查找长度为 $\frac{m+1}{2}$，所以分块查找的平均查找长度

$$ASL \approx \log_2(\frac{n}{m}+1)-1+\frac{m+1}{2} \approx \log_2(\frac{n}{m}+1)+\frac{m}{2}。$$

5.2　动 态 查 找

动态查找是指在查找过程中，如果查找失败，就把待查找的记录插入到数据集中。显然动态查找的数据集是通过查找过程而动态生成的。动态查找法主要通过树结构来实现。在第 3 章介绍的二叉搜索树和平衡二叉树就是两种动态查找方法。

在大规模数据查找中，大量数据信息存储在外存磁盘。在查找时需要从磁盘中读取

数据。磁盘 I/O 操作的基本单位为块（block）。位于同一盘块中的所有数据都能被一次性全部读取出来。磁盘上数据必须用一个三维地址唯一标识：柱面号、盘面号、块号（磁道上的盘块）。读/写磁盘上某一指定数据需要下面三个步骤：

1）根据柱面号使磁头移动到所需要的柱面上，这一过程被称为定位或查找 。

2）根据盘面号来确定指定盘面上的磁道。

3）盘面确定以后，盘片开始旋转，将指定块号的磁道段移动至磁头下。

经过上面三个步骤，指定数据的存储位置就被找到。这时就可以开始读/写操作了。也就是说，数据存取时间由下列几个时间段组成：

1）查找时间（seek time）Ts：完成上述步骤（1）所需要的时间。这部分时间代价最高，最大可达到 0.1s 左右。

2）等待时间（latency time）Tl：完成上述步骤（3）所需要的时间。由于盘片绕主轴旋转速度很快，一般为 7200 转/分（电脑硬盘的性能指标之一，家用的普通硬盘的转速一般有 5400rpm（笔记本）、7200rpm 几种）。因此一般旋转一圈大约 0.0083s。

3）传输时间（transmission time）Tt：数据通过系统总线传送到内存的时间，一般传输一个字节(byte)大概 0.02us=2*10^(-8)s。

磁盘 IO 代价主要花费在查找时间 Ts 上。因此数据信息尽量存放在同一盘块，同一磁道中。或者至少放在同一柱面或相邻柱面上，以求在读/写信息时尽量减少磁头来回移动的次数，避免过多的查找时间 Ts。

当采用二叉搜索树结构来存储这样的数据时，二叉树的结点就要分布在不同的数据块上。如图 5-4 所示，查找成功时平均需要访问两个磁盘块。如果程序频繁使用该二叉搜索树，这种访问就会显著增加程序的执行时间。此外，在该二叉搜索树中插入和删除数据也要访问多个数据块。虽然，当二叉搜索树的结点数据都在内存中时查找效率很高，当从存储在磁盘上的大规模数据中查找时，二叉搜索树的性能优点就无法体现了。

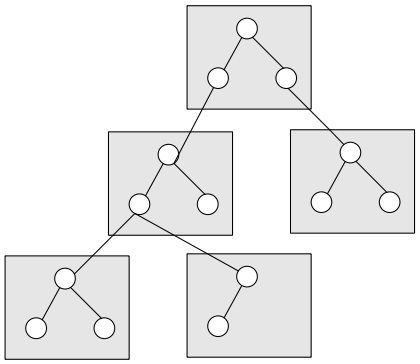


图 5-4　二叉搜索树的结点数据存放在不同的磁盘块上

在大规模数据查找中，大量信息存储在外存磁盘中，选择正确的数据结构可以显著降低查找磁盘中数据的时间。B−树和 B+树就是两种常见的高效外存数据结构。

5.2.1 B-树

“B-树”也称“B 树”（Balanced Tree），是一种平衡的多路查找树。

一棵 m 阶的 B-树，或为空树，或为满足下列特性的 m 叉树：

1）树中每个结点至多有 m 棵子树；

2）根结点至少有两棵子树

3）除根之外的所有非叶结点至少有$\left\lceil \frac{m}{2} \right\rceil$棵子树；

4）所有叶子结点都出现在同一层。

5）有 n 棵子树的非叶结点包含的信息如下（恰好包含 n-1 个关键字）：

$(n, T_0, K_1, T_1, K_2, T_2, ..., K_n, T_n)$

其中，n（$\frac{m}{2}-1 \leqslant n \leqslant m-1$）为关键字的个数，n+1 为子树个数；$K_i$（$i=1,...,n$）为关键字，且关键字从小到大排序，即$K_i < K_{i+1}$（$i=1,...,n-1$）；$T_i$（$i=0,1,...,n$）为指向子树根结点的指针，且指针$T_{i-1}$所指子树中所有结点的关键字均小于$K_i$（$i=1,...,n$），$T_n$所指子树中所有结点的关键字均大于$K_n$。

根据这些条件，B-树往往至少是半满的，有较少的层，而且是完全平衡的。

性质 1 含有 n 个结点的 m 阶 B-树的最大高度$\leqslant \log_{m/2}\left(\frac{N+1}{2}\right)+1$

证明：先讨论深度为 l+1 的 m 阶 B-树所具有的最少结点数。根据 B-树的定义，第一层至少有 1 个结点；第二层至少有 2 个结点；由于除根之外的每个非叶结点至少有$\frac{m}{2}$棵子树，则第三层至少有$2\cdot\frac{m}{2}$个结点；……以此类推，第 l+1 层至少有$2\cdot\left(\frac{m}{2}\right)^{l-1}$个结点。对于每个含有 k 个关键字的叶子结点增加 k+1 个扩充叶结点作为查找不成功的标记。设查找不成功的扩充结点个数为 N+1，则

$$N+1 \geqslant 2\cdot(\frac{m}{2})^{l-1}$$

即：

$$l \leqslant \log_{\frac{m}{2}}\left(\frac{N+1}{2}\right)+1$$

例如图 5-5 为一棵 4 阶 B-树，每个结点最多有 3 个关键字，4 棵子树，每个结点中的关键字都是有序的。

在 B-树上进行查找的过程和二叉搜索树的查找类似。例如，在上述图中查找关键字 256 的过程如下：首先从根结点指针开始，找到 a 结点，判断 256<375，则选择左边路径，找到结点 b，b 结点有 3 个关键字，依次判断大小得 256>236，则寻找到结点 g，结点 g 只有一个关键字就是 256，与所查关键字相同，则查找成功。查找不成功的过程也类似，例如在上述 B-树中查找关键字 400，仍然从根结点开始，扫描的结点依次是 a，

c，i，i 是叶结点，有两个关键字，分别是 393 和 396，没有所查关键字 400，则查找不成功。

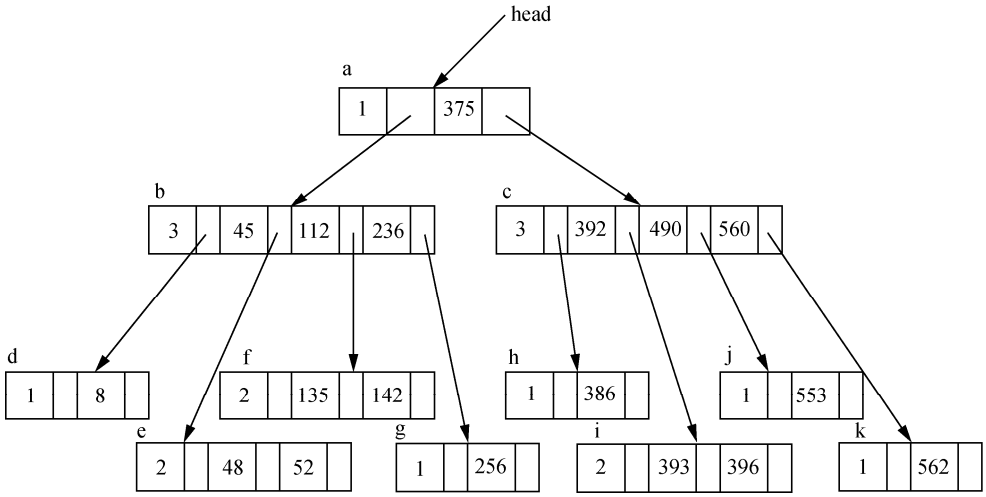


图 5-5 4 阶 B-树

B-树的查找与二叉搜索树的查找不同的是，要先确定待查找记录所在的结点，然后再在结点中查找该记录。

例 5.3 定义了 B-树的数据结构，并给出了查找操作的实现。实际应用中，B-树主要用于文件的索引，会涉及外存的存取，本书不做探讨。

【例 5.3】B-树定义及查找操作。

```
template <class Key>
class BNode                              // 定义 B 树结点
{
public:
    int n;                               // 关键字的个数
    BNode<key> *parent;                  // 指向父结点的指针
    Key key[maxNum+1];                   // 存储关键字的数组，
                    // 其中 maxNum 为结点中关键字的最大个数，0 号单元未用
    BNode<Key> *ptr[keyNum+1]; // 存储指针的数组，0 号单元使用
};

template<class Key>
class Result                             // 定义用于在 B 树中查询关键字的数据结构
{
public:
    BNode<Key> *r;                       // 指向找到的结点
    int i;                               // 记录所查关键字在结点中的位置
    int flag;                            // 是否查找到关键字的标志
    Result(BNode * r,int i,int flag);
}
```

```
template<class Key>
class BTree                          //定义 B 树
{
    BNode<Key> *root;
    int m;                           // B 树的阶
public:
    BTree();                         // 构造函数
    Result<Key> &Search(const Key& x); // 查询关键字 x 所在的结点
    BNode<Key>* InsertBTree(Key x,BNode* p,int i); // 插入关键字 x
void DeleteBTree(Key x);                  // 删除关键字 x
……
};
template<class Key>
Result<Key>& BTree::Search(const Key& x)
{// 在 m 阶 B-树中查找关键字 x，返回结果 Result。如查找成功，则 flag=1，指针 r
// 指向的结点中的第 i 个关键字就是所查关键字 x；如果查找不成功，则 flag=0，
// 等于 x 的关键字应插入在指针 r 所指的结点中第 i 个关键字和第 i+1 个关键字之间
    BNode *p = root;
    BNode *q = NULL;
    bool found = false;
    int i = 0;
    while (p && !found)
    {
        for(i=1;i<=n;i++)   //在 p->key[1,...,n]中查找，确定 i，使得:p->
                            //key[i]<=x<=key[i+1]
    {
      if(p->key[i]>=x)
        break;
    }
        if(i>0 && p->key[i] == x) found = true;    // 找到所查关键字
        else
        {
            q = p;
            p = p->ptr[i-1];
        }
    }
        if(found)  return Result(p,i,1);
        else   return Result(q,i,0);
    }
}
```

对 B-树进行查找包含两种基本操作：（1）在 B-树中找结点；（2）在结点中找关键字。由于 B-树通常存储在磁盘上，则前一查找操作是在磁盘上进行的，而后一查找

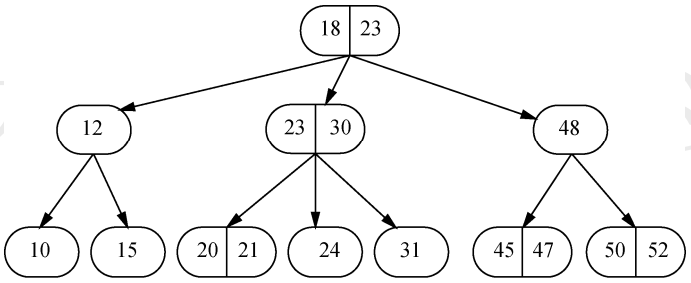
操作是在内存中进行的，即在磁盘上找到指针 p 所指结点后，先将结点中的信息读入内存，然后再利用顺序查找或者二分查找查询关键字为 x 的记录。显然，在磁盘上进行一次查找比在内存中进行一次查找耗费时间多，因此，在磁盘上进行查找的次数，即包含待查记录的关键字的结点在 B−树上的层次数，是决定 B-树查找效率的首要因素。

在磁盘上进行查找的次数的最坏情况是当包含待查记录关键字的结点在 B-树的最大层次上，即最坏情况下的查找次数为 $\log_{m/2}\left(\frac{N+1}{2}\right)+1$。

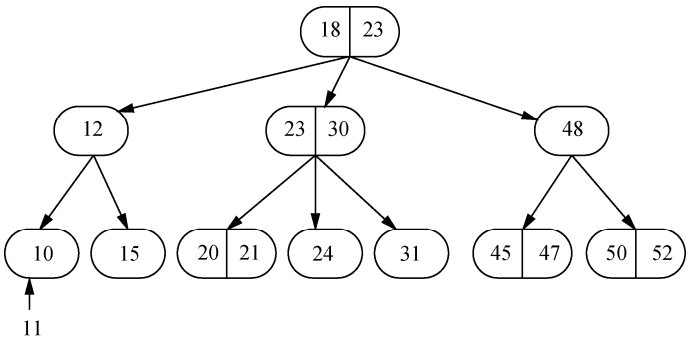
m 阶 B-树的生成同普通树一样，从空树起，逐个插入关键字。同普通树不同的是，每次插入一个关键字后要保持 B−树的形状。常用的插入策略是首先将关键字添加到最底层的某个叶结点中，若该结点不满（关键字个数不超过 m−1），则插入完成，否则要将叶结点“分裂”成两个叶结点，并将一个关键字提升到双亲结点中。如果双亲结点已满，则重复刚才的过程，直到到达根结点，并创建一个新的根结点。

下面以 3 阶 B−树为例说明向 m 阶 B−树中插入关键字时的三种情形。

1）将关键字插入到叶结点后，叶结点中的关键字个数不超过 m−1，如图 5-6（a）所示的 3 阶的 B−树中，插入关键字 11 时，首先根据查找过程确定 11 应该插入的叶结点位置，如图 5-6（b）所示，然后插入。由于插入之后的叶结点中有两个关键字，符合 3 阶 B−树的性质要求，因此直接插入即可，如图 5-6（c）所示。

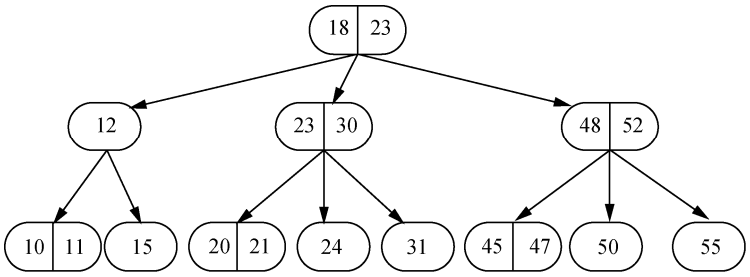


（a）



（b）

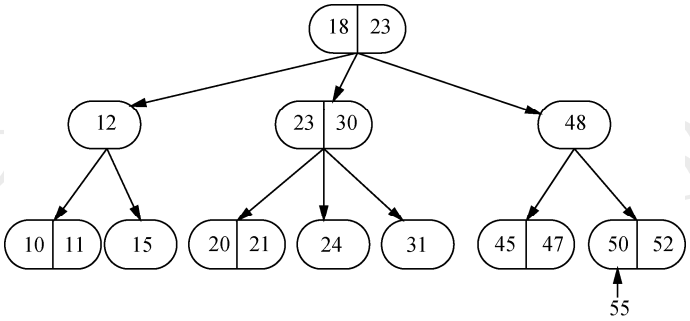
图 5-6　3 阶 B−树插入情形一



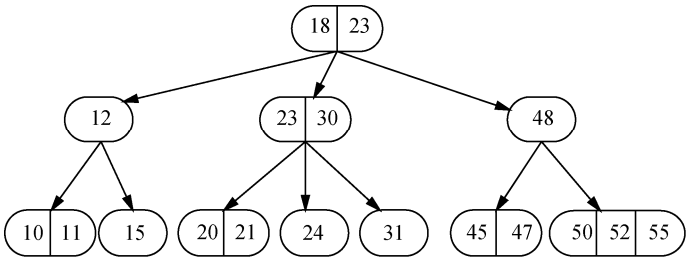
（c）

图 5-6　3 阶 B-树插入情形一（续）

2）插入关键字后叶结点的关键字已满。这种情形下，要分裂叶结点，创建一个新的叶结点，将已经满的叶结点中一半关键字移动到新叶结点中，并将新叶结点合并到B-树中。把中间的关键字提升到双亲结点中，同时在双亲结点中设置一个指向新叶结点的指针。如在图 5-6（c）中插入关键字 55。由于 55 欲插入的叶结点中已有两个关键字，如图 5-7（a）所示，插入 55 后需要分裂该结点，将 52 上提至双亲结点中，如图 5-7（b-c）所示。

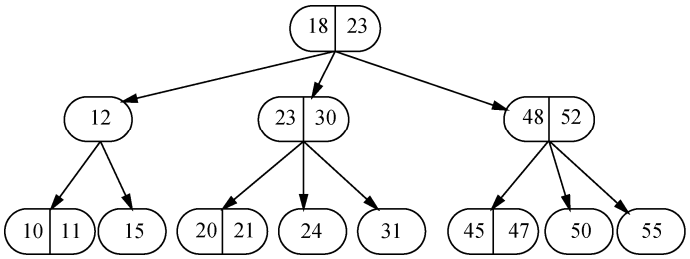


（a）



（b）

图 5-7　3 阶 B-树插入情形二



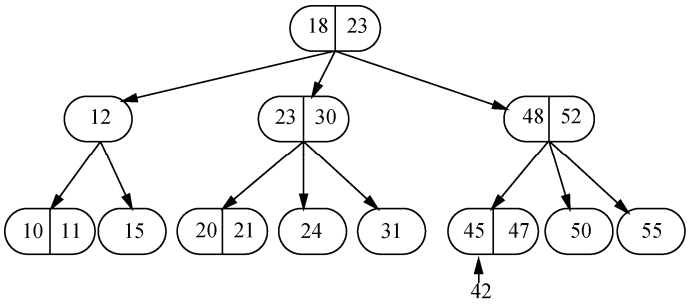
（c）

图 5-7　3 阶 B−树插入情形二（续）

结点的“分裂”原则如下：

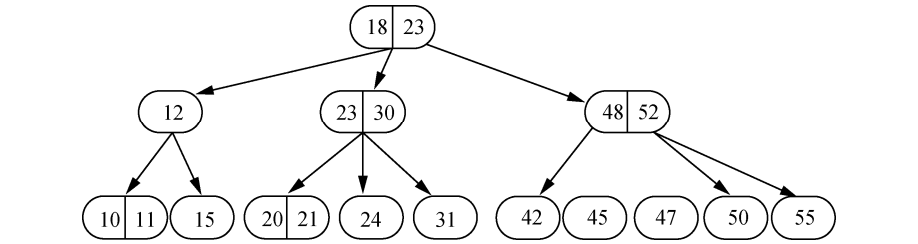
假设，p 结点中已有 m-1 个关键字，当插入一个关键字之后，结点中含有信息为：$(m,T_0,K_1,T_1,K_2,T_2,...,K_m,T_m)$，且其中 $K_i < K_{i+1}$（$1 \leqslant i \leqslant m$），此时可将 p 结点分裂为 p 和 q 两个结点，其中 p 结点中含有信息为 $\left(\frac{m}{2}-1,T_0,K_1,T_1,K_2,T_2,...,K_{\frac{m}{2}-1},T_{\frac{m}{2}-1}\right)$；q 结点中含有信息为 $\left(m-\frac{m}{2},T_{\frac{m}{2}},K_{\frac{m}{2}+1},T_{\frac{m}{2}+1},...,K_m,T_m\right)$，而关键字 $K_{\frac{m}{2}}$ 和指针 q 一起插入到 p 的双亲结点中。

3）如果 B−树的根结点的关键字是满的，必须分裂根结点，并将根结点中间的关键字提升到新建的根结点中。这是唯一会引起 B−树高度增长的情形。如在图 5-7（c）中插入关键字 42，首先要确定插入的叶结点位置如图 5-8(a)，插入后引起叶结点 p 分裂，如图 5-8（b），并将中间的关键字 45 提升到双亲结点 q 中，如图 5-8（c）；导致 q 结点关键字满，继续分裂，将中间关键字 48 提升到双亲结点 z 中，如图 5-8（d）；又导致根结点的关键字是满的，因此分裂根结点，创建新的根结点，并将原根结点的中间的关键字提升到新的根结点中，如图 5-8（e）。

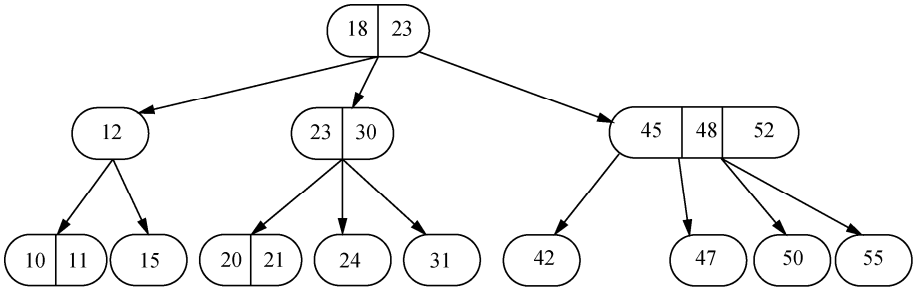


（a）

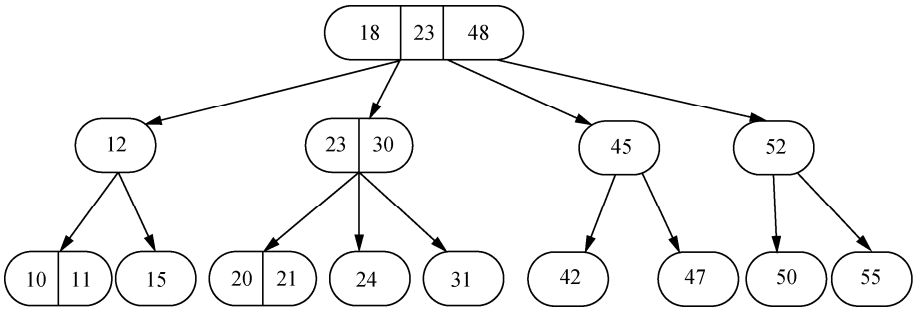
图 5-8　3 阶 B−树插入情形三



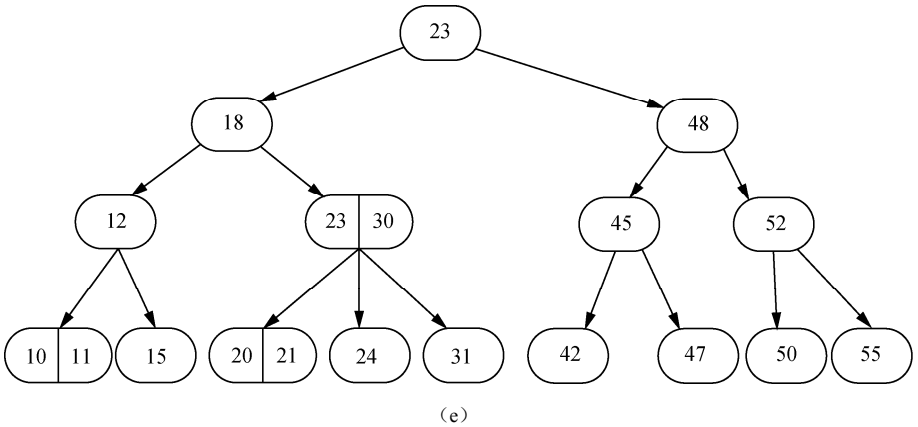
（b）



（c）



（d）



（e）

图 5-8　3 阶 B−树插入情形三（续）

在 B-树上插入关键字的算法如下描述。

输入：待插入的关键字 x，插入的结点位置 p，x 在 p 中的位置 i

输出：插入并调整后的 m 阶 B-树的根结点

步骤 1：初始化 finished=false；

步骤 2：将 x 插入到结点 p 中；

步骤 3：如果 p 的关键字个数大于 m，创建新结点 ap，并将 p→key[m/2+1..m], p→ptr[m/2..m]移入到新结点 ap 中，x=p→key[m/2]，p=p-parent；x 在 p 中的插入位置作为 i；重复步骤 2；否则到步骤 4；

步骤 4：返回根结点。

B-树的删除在很大程度上是插入操作的逆过程。首先找到要删除的关键字所在结点，若该结点为最底层的叶子结点，且删除后该结点中的关键字数目不少于$\left\lceil \frac{m}{2} \right\rceil$，则删除完成，否则要进行“合并”结点的操作。假若所删关键字为非叶结点中的关键字 K_i，则可将 T_i 所指子树中的最小关键字 Y（肯定位于叶结点中）替代 K_i，然后将问题转化为从叶结点中删去关键字 Y 的情形。如图 5-9 所示，若删除关键字 48，可以用 f 结点中的 55 替代 48，然后在 f 结点中删去 55。

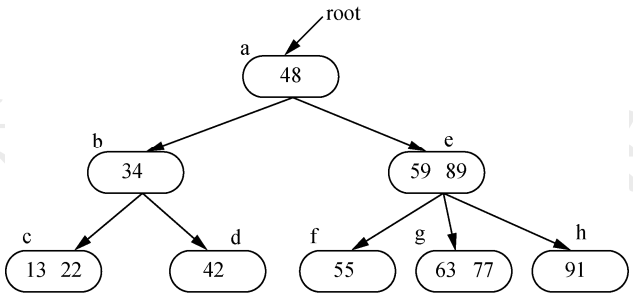


图 5-9　3 阶 B-树删除关键字示例

下面详细讨论删除叶结点中的关键字的 3 种情形：

1）被删关键字所在结点中的关键字数目不小于$\left\lceil \frac{m}{2} \right\rceil$，则只需从该结点中删去该关键字 K_i 和相应指针 T_i，树的其他部分不变。例如，从图 5-9 中删除关键字 22，删除后的 B-树如图 5-10 所示。

2）被删关键字所在结点中的关键字数目等于（$\left\lceil \frac{m}{2} \right\rceil$-1），而与该结点相邻的右兄弟（或左兄弟）结点中的关键字数目大于（$\left\lceil \frac{m}{2} \right\rceil$-1），则需将其右兄弟（或左兄弟）结点中的最小（或最大）的关键字上移至双亲结点中，而将双亲结点中划分这两个结点的关键字下移至被删关键字所在结点中。例如，从图 5-10 中删去 55，需将其右兄弟结点

中的 63 上移至 e 结点中，而将 e 结点中的 59 下移至 f 中，从而使 f 和 g 中的关键字数目均小于（$\left\lceil \frac{m}{2} \right\rceil$-1），而双亲结点中的关键字数目不变，如图 5-11 所示。

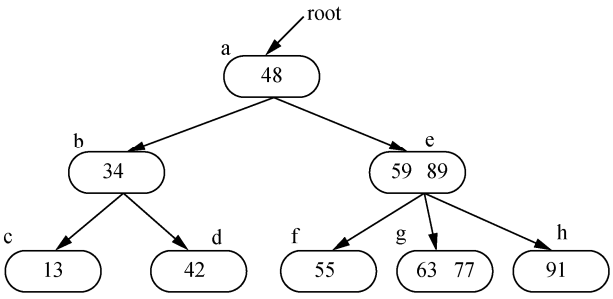


图 5-10　B-树删除情形一

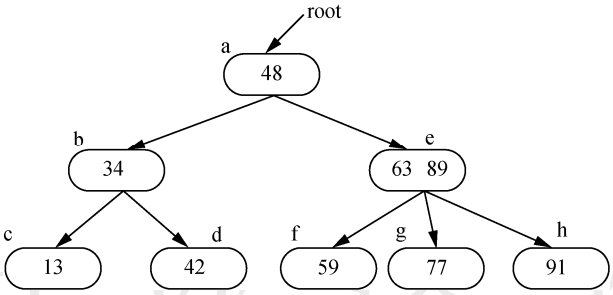


图 5-11　B-树删除情形二

3）被删关键字所在结点和其相邻的左右兄弟结点中的关键字数目均等于（$\left\lceil \frac{m}{2} \right\rceil$-1）。假设该结点有左兄弟，且其左兄弟结点地址由双亲结点中的指针 T_i 所指，则在删去关键字之后，将该结点中剩余的关键字和指针以及双亲结点中划分该结点与其左兄弟结点的关键字 K_{i+1} 一起，合并到 T_i 所指的左兄弟结点中（若没有左兄弟，则合并到右兄弟中）。例如，从图 5-11 中删除 77，则应将 g 中的空指针和双亲 e 结点中的 63 一起合并至左兄弟结点 f 中。删除后的树如图 5-12 所示。

若在图 5-12 中继续删除 42，d 结点的空指针和其双亲结点结点 b 的关键字合并到结点 c 中，使得结点 b 的关键字不足，继续将双亲 b 结点中的剩余信息（“指针 c”）和其双亲 a 结点中关键字 48 一起合并至右兄弟 e 中，删除后的 B-树如图 5-13。

在 B-树上删除关键字的算法如下描述。

输入：待删除的关键字 x；

输出：无

步骤 1：确定 x 所在的结点 p。

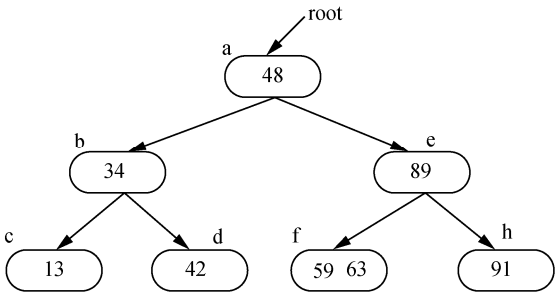


图 5-12　B−树删除情形三

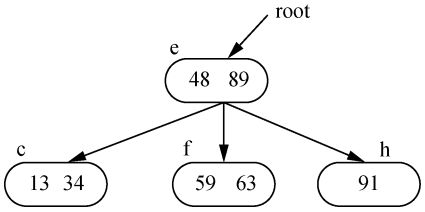


图 5-13　B−树删除导致 B−树高度减少的情形

步骤 2：如果 p 为空则结束；如果 p 是叶结点则直接删除关键字；否则在 x 所划分的左边子树（或右边子树）的底层叶结点 q 中找到小于 x 的最大关键字（或大于 x 的最小关键字）s，将 p 的关键字 x 替换为 s; p=q，并从 p 中删除 s。

步骤 3：如果 p 中的关键字个数大于等于[m/2]−1，结束。如果 p 有左兄弟结点 left，且左兄弟结点的关键字个数大于[m/2]−1，则到步骤 4；如果 p 有右兄弟结点 right，且右兄弟结点的关键字个数大于[m/2]−1，则到步骤 5；如果 p 的双亲结点是根结点 root 且根结点只有一个关键字，则到步骤 6；否则到步骤 7。

步骤 4：将 p 的双亲结点中划分 left 和 p 的关键字 kp 移动到结点 p 中，作为第一个关键字；将 p 的左兄弟结点的最后一个关键字移动到双亲结点中原 kp 的位置；将 p 的左兄弟结点的最后一个子树移动到结点 p 中，作为第一棵子树，结束。

步骤 5：将 p 的双亲结点中划分 p 和 right 的关键字 kp 移动到结点 p 中，作为最后一个关键字，将 p 的右兄弟结点的第一个关键字移动到双亲结点中原 kp 的位置；将 p 的右兄弟结点的第一个子树移动到结点 p 中，作为最后一棵子树。

步骤 6：如果 p 有左兄弟结点，合并 p、p 的左兄弟以及双亲结点，形成一个新的根结点；否则合并 p、p 的右兄弟以及双亲结点，形成一个新的根结点，结束。

步骤 7：合并 p、p 的左兄弟（或右兄弟）结点 q 以及双亲结点中划分 p 和 q 的关键字；p＝p 的双亲结点；重复步骤 3。

5.2.2　B+树

B+树也是一种多叉树结构，通常用于数据库和文件系统中。B+树的特点是能够保持数据稳定有序，其插入与修改拥有较稳定的时间复杂度。B+树是 B−树的变形树。在 B+树中，对数据的引用指向叶结点，内部结点的关键字只是充当划分子树的分界值。叶结点被链接成一个序列。

一棵 m 阶 B+树或为空树，或为满足下列特性的 m 叉树：

1）每个结点至多有 m 棵子树；

2）根结点至少有两棵子树；

3）除根以外的所有非叶结点至少有$\left\lceil \frac{m}{2} \right\rceil$棵子树；

4）所有叶子结点都出现在同一层；

5）有 n 棵子树的非叶结点包含的信息如下（恰好包含 n 个关键字）：

$(n, T_1, K_1, T_1, K_2, T_2, ..., K_n, T_n)$

其中，n（$\left\lceil \frac{m}{2} \right\rceil \leqslant n \leqslant m$）为关键字的个数，即子树个数；$K_i$（$i=1,...,n$）为关键字，且关键字从小到大排序，即$K_i < K_{i+1}$（$i=1,...,n-1$）；$T_i$（$i=1,...,n$）为指向子树根结点的指针，且指针$T_i$所指子树中所有结点的关键字均不大于$K_i$（$i=1,...,n$）。

图 5-14 是一个 2 阶 B+树的简单例子，通常在 B+树上有两个指针，一个指向根结点，另一个指向关键字最小的叶结点。因此在 B+树中既可以从最小关键字开始顺序查找也可以从根结点开始进行随机查找。

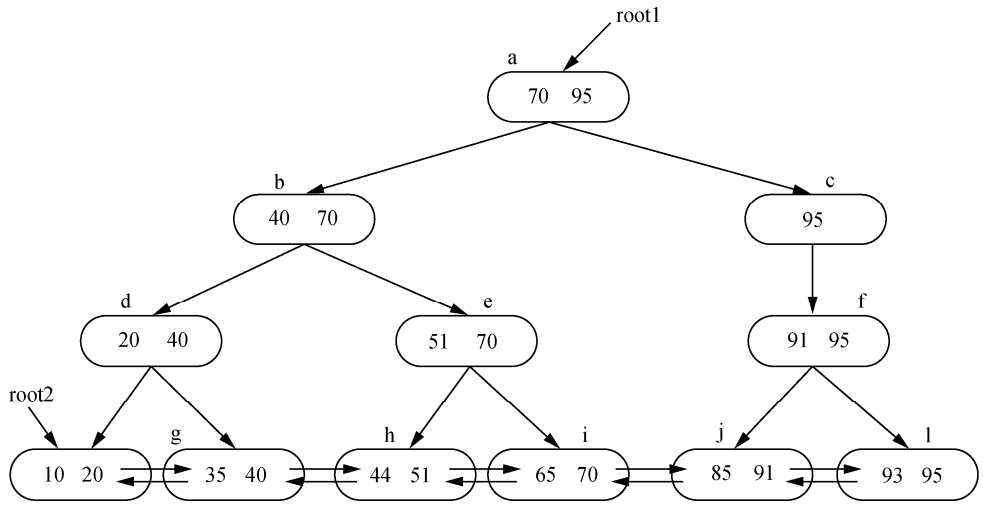


图 5-14　B+树举例

在 B+树上进行查找的过程与 B-树类似，只是在查找时，无论非叶结点上的关键字是否与给定关键字相同，都要沿着相应的子树继续向下查找，一直查找到叶子结点。所以每次查找的路线都是从根结点到叶结点的一条路径。例如，在图 5-14 中查找关键字 70，则需要从根结点 a 开始比较，由于 70 不大于 a 的第一个关键字，因此继续在 a 的第一棵子树中查找；由于 70 大于 b 的第一个关键字但是不大于 b 的第二个关键字，因此继续在 b 的第二棵子树中查找；由于 70 大于 e 的第一个关键字但是不大于 e 的第二个关键字，因此继续在 e 的第二棵子树中查找；最后在叶结点 i 中查找到 70，确定查找成功。

与 B-树相同，B+树的插入也仅在叶结点上进行，当结点中的关键字个数大于 m 时

要分裂成两个结点，每个叶结点含有关键字个数分别是$\left\lceil \frac{m}{2} \right\rceil$和$\left\lceil \frac{m+1}{2} \right\rceil$，把原结点中的关键字 K_i（$1 \leqslant i \leqslant \left\lceil \frac{m}{2} \right\rceil$）及相应的子树指针移到新叶结点中，并把新叶点中的最后关键字 $K_{\lceil m/2 \rceil}$ 复制到双亲结点中（与 B−树不同）。如果双亲结点是满的，分裂过程与 B−树相同。下面举例说明 B+树的插入操作。

在图 5-15 所示的 3 阶 B+树中，插入关键字 20 时，先根据查找过程确定应将 20 插入到叶结点位置，插入后叶结点的关键字个数不超过 m，插入完成，如图 5-16 所示。

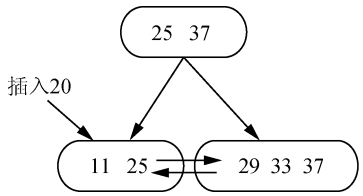


图 5-15　3 阶 B+树

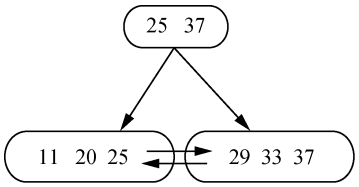


图 5-16　在 3 阶 B+树中插入关键字 20

在图 5-16 所示的 3 阶 B+树中继续插入关键字 23，与上面的插入类似，但是 23 插入到的叶结点 p 之后，p 结点的关键字个数超过了 m，如图 5-17 所示。因此需要分裂 p，把其前 2 个关键字及空指针移动到新结点 q 中，并把 q 中最后一个关键字复制到其双亲结点中，同时将结点 q 合并到 B+树中。

在图 5-18 所示的 3 阶 B+树中继续插入关键字 30 的插入及分解过程如图 5-19 所示，由于根结点进行了分裂，B+树的高度增加一层。

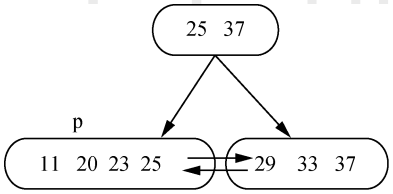


图 5-17　在 3 阶 B+树中插入 23 导致 p 结点满

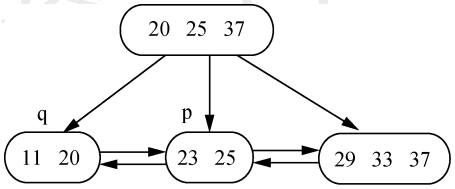


图 5-18　在 3 阶 B+树中插入 23 后分裂结点

B+树的删除也是在叶结点中进行。当叶结点中的最大关键字被删除时，其在非终端结点中的副本仍可以作为一个“分界关键字”存在。如在图 5-20 中删除关键字 25 时，先确定其所在的叶结点 p，从 p 中删除关键字，如图 5-21 所示。注意，关键字 25 没有从内部结点中删除。若因删除而使叶结点中关键字的个数小于$\left\lceil \frac{m}{2} \right\rceil$时，如果该叶结点的兄弟结点关键字个数大于$\left\lceil \frac{m}{2} \right\rceil$，则将该叶结点和其兄弟结点中的关键字重新分配，否则删除该叶结点，将剩余的关键字合并到其兄弟结点中，过程同 B−树相似。

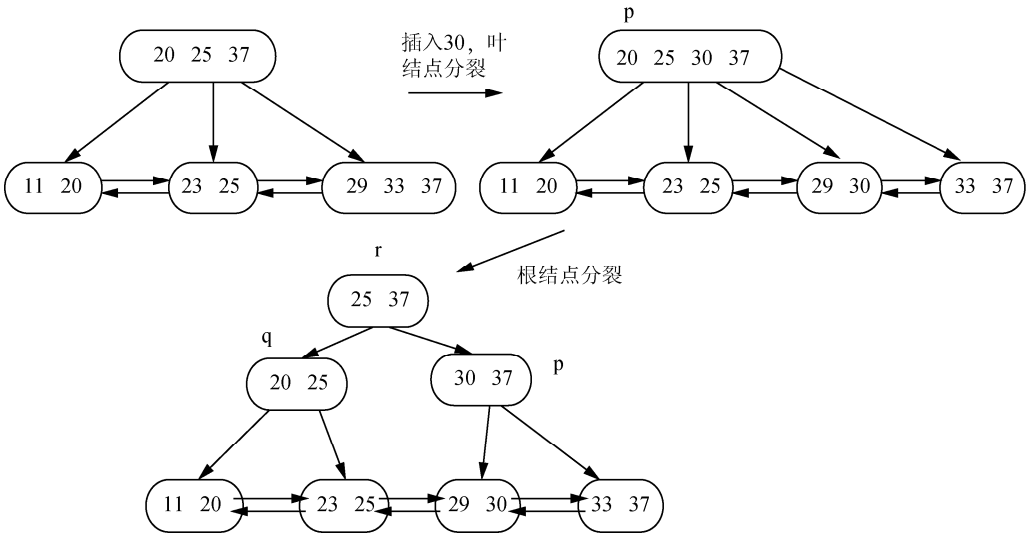


图 5-19　在 3 阶 B+树中插入 30 后 B+树高度增加

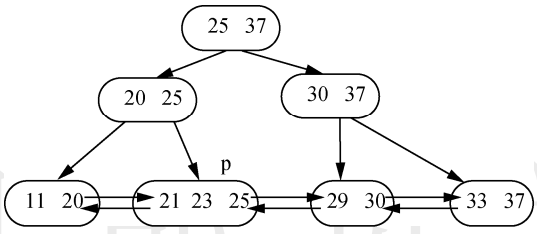


图 5-20　先确定删除的关键字所在叶结点

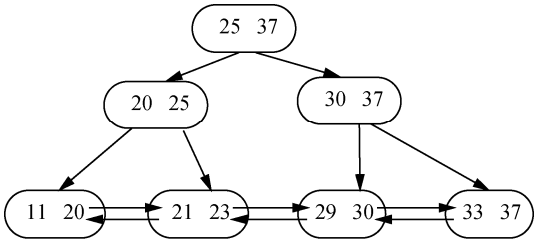


图 5-21　删除关键字 25 后的 B+树

图 5-22 演示了删除后进行关键字重新分配的情形。在删除关键字 20 时，先确定其所在的叶结点 p，删除后 p 结点的关键字个数小于$\left\lceil \frac{m}{2} \right\rceil$，但其右兄弟结点 q 的关键字个数大于$\left\lceil \frac{m}{2} \right\rceil$，因此将其右兄弟结点的最小关键字 21 移动到 p 中，并更新双亲结点的相应分界值。

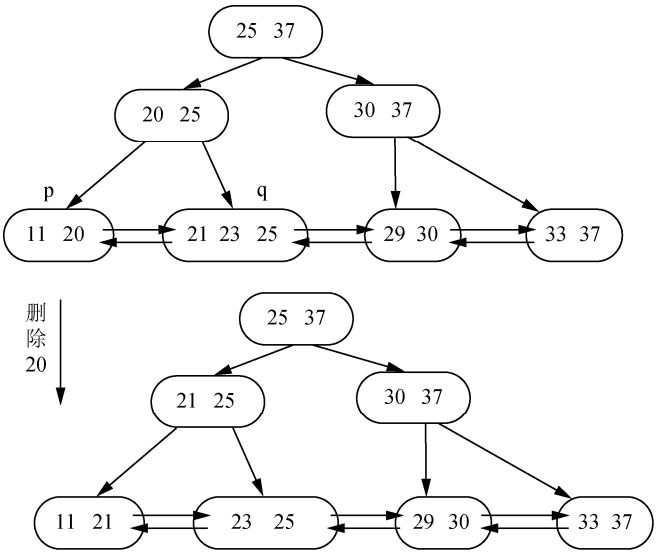


图 5-22　删除关键字后进行关键字的重新分配

图 5-23 演示了在删除时出现结点合并的情形。由于在叶结点 p 删除关键字 30 后，p 结点及其兄弟结点的关键字个数均不大于$\left\lceil \frac{m}{2} \right\rceil$，因此要将这两个叶结点合并成一个叶结点，同时删除双亲结点 pre 中划分这两个结点的关键字 30，进而导致 pre 结点的关键字个数小于$\left\lceil \frac{m}{2} \right\rceil$，由于 pre 的兄弟结点 pre_q 的关键字个数大于$\left\lceil \frac{m}{2} \right\rceil$，因此将这两个结点的关键字进行重新分配，否则要继续合并。

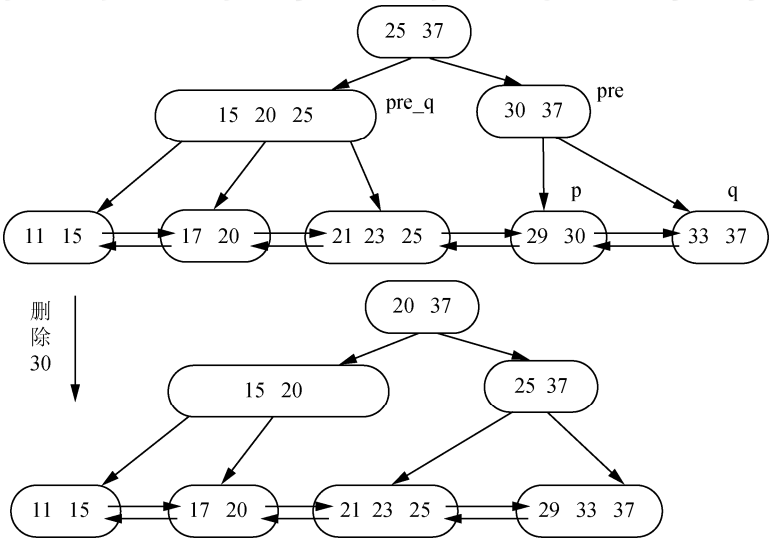


图 5-23　删除关键字后进行结点的合并

5.3 散　　列

5.3.1 散列的概念

前面讨论的静态查找和动态查找结构中，数据记录在结构中的相对位置是随机的，和记录的关键字之间不存在确定的关系，因此，在结构中查找数据记录时需进行一系列关键字的比较。这一类查找建立在“比较”基础上；在静态查找时，比较的结果为“等”或“不等”两种可能，在动态查找时，比较的结果为“大于”，“等于”和“小于”三种可能。查找的效率依赖于查找过程中所进行的比较次数。

一种理想的查找方法是不进行任何比较，一次存取便得到所查记录，这就要在记录的存储位置和记录的关键字之间建立一种关系。例如，读取指定下标的数组元素，就是根据数组的起始存储地址以及数组下标值而直接计算出来的，时间复杂度为O(1)，与数组元素的个数无关。虽然理想的情况是在关键字和结构中一个唯一的存储位置建立对应关系，但在实际应用中，只能近似地获得这样的结果。

散列方法就是在记录的关键字与它的存储位置之间建立一个确定的对应函数关系，使得每个关键字与结构中一个唯一的存储位置相对应。在查找时，首先对记录的关键字进行函数计算，把函数值（散列值）当做该记录的存储位置，在结构中按此位置取记录比较。若关键字相等，则查找成功。在存放记录时，按照散列函数计算存储位置，并按此位置存放记录。在散列方法中使用的映射函数称为散列函数。按照散列方法构造出来的表或结构称为散列表。

散列方法的核心是：由散列函数确定关键字与散列地址之间的对应关系，通过这种关系来实现存储并进行查找。一般情况下，散列表的存储空间采用顺序存储结构——数组，散列地址就是数组的下标。散列方法的目标就是设计一个散列函数 f，$0 \leqslant f(K) < L$，K 为记录的关键字，L 为散列表的长度。散列函数值 f(K) 就是关键字为 K 的记录对应的存储地址。

例如，将关键字{John,Mike,Peter,Rose,Tom,Dave }存放于长度为 26 的散列表中，取关键字中第一个字母（不区分大小写）在字母表{a,b,c,…,z}中的序号（0~25）作为该关键字的散列值。则关键字集合构成的散列表如表 5-1 所示。

如果在散列表 5-1 中，继续插入关键字 Mary 时，Mary 的散列地址是 12，但是这个位置已经被占用了。这种对于不同的关键字，由散列函数得到的散列地址相同的现象称为“冲突”，发生冲突的两个关键字称为散列函数的同义词。在理想情况下散列函数应该运算简单并且应该保证任何两个不同的关键字映射到不同的存储单元。但实际上这是不可能的，因为存储单元的数量有限，而关键字个数可能远远大于散列表长度。因此在散列方法中，需要考虑的两个问题：

表 5-1　散列表存储示例

散列地址	关键字	散列地址	关键字
0		13	
1		14	
2		15	Peter
3	Dave	16	
4		17	Rose
5		18	
6		19	Tom
7		20	
8		21	
9	John	22	
10		23	
11		24	
12	Mike	25	

1）构造使关键字均匀分布的散列函数，提高散列质量避免发生冲突；

2）设计冲突解决方法，处理冲突。

本章主要介绍几种常用的散列函数和解决冲突的方法。

散列方法的查找性能用平均查找长度 ASL（Average Search Length）来衡量。根据查找成功与否，又有查找成功的平均查找长度 ASLsucc 和查找不成功的平均查找长度 ASLunsucc 之分。查找成功的平均查找长度是指查找到散列表中已有的数据记录的平均探查次数。查找不成功的平均查找长度 ASLunsucc 是指在表中查找不到待查的记录，但找到插入位置的平均探查次数。

影响散列表查找效率的因素包括：散列函数是否均匀、处理冲突的方法以及散列表的装填因子。其中散列表的装填因子如下定义：

$$\alpha = \frac{\text{填入表中的元素个数}}{\text{散列表的长度}}$$

装填因子 α 是散列表装满程度的标志因子。由于表长是定值，α 与“填入表中的元素个数”成正比，所以，α 越大，填入表中的元素较多，产生冲突的可能性就越大；α 越小，填入表中的元素较少，产生冲突的可能性就越小。实际上，散列表的平均查找长度是装填因子 α 的函数，只是不同处理冲突的方法有不同的函数。

5.3.2　散列函数

在介绍散列函数之前，假设处理的关键字为整型情形，在实际应用中，如果关键字不为整型，可以构建关键字与正整数之间的一一对应，从而把原关键字的查询转化为正整数的查询问题。

构造散列函数有以下几点要求：

1）散列函数的定义域必须包括需要存储的全部关键字，如果散列表允许有 m 个地

址时，其值域必须在 0 到 m−1 之间。

2）散列函数计算出来的地址应能均匀分布在整个地址空间中：若 key 是从关键字集合中随机抽取的一个关键字，散列函数应能以同等概率取 0 到 m-1 中的每一个值。

3）散列函数应是简单的，能在较短的时间内计算出结果。

下面介绍几种常用的散列函数构造方法。在实际问题中应根据关键字的特点，选用适当的方法。

1．直接定址法

散列函数是关键字的线性函数，线性函数值作为散列地址：

$$Hash(key) = a \times key + b$$

其中，a, b 为常数。这类散列函数是一对一的映射，一般不会产生冲突。但是，它要求散列地址空间的大小与关键字集合的大小相同。

例如，有一组关键字如下：{ 1548, 1569, 1527, 1530, 1505, 1558, 1547, 1501 }。散列函数为 Hash (key) = key -1500。

则 Hash (1548) = 48　Hash (1569) = 69
Hash (1527) = 27　Hash (1530) = 30
Hash (1505) = 5　Hash (1558) = 58
Hash (1547) = 47　Hash (1501) = 1

数据记录存储在各个关键字对应的散列地址中。

2．数字分析法

设有 n 个 d 位数，每一位可能有 r 种不同的符号。这 r 种不同的符号在各位上出现的频率不一定相同，可根据散列表的大小，选取其中各种符号分布均匀的若干位作为散列地址。

例如，散列表地址范围有 3 位数字，观察下列关键字后，取各关键码的④⑤⑥位作为记录的散列地址。也可以把第①，②，③和第⑤位相加，舍去进位位，变成一位数，与第④，⑥位合起来作为散列地址。还有其他方法。

8	5	3	2	4	8
8	5	2	3	6	9
8	5	1	5	3	7
8	5	2	6	9	0
8	5	2	8	1	5
8	5	2	5	5	6
8	5	3	1	4	7
8	5	1	1	2	1
①	②	③	④	⑤	⑥

数字分析法仅适用于事先明确知道表中所有关键码每一位数值的分布情况，它完全依赖于关键码集合。如果换一个关键码集合，选择哪几位要重新决定。

3．除留余数法

设散列表中允许的地址数为 m，取一个不大于 m，但最接近于或等于 m 的质数 p，并构造如下散列函数：

$$Hash(key) = key \% p \qquad p \leqslant m$$

其中，“%”是整数除法取余的运算。这是一种最简单也最常用的散列函数构造方法。

例如：有一个关键字 key = 48，散列表大小 m = 19，即 H[19]。取质数 p= 19。散列函数 Hash (key) = key % p。则 48 的散列地址为 Hash (48) = 48 % 19 = 10。

需要注意的是，在除留余数方法中，p 的选择很重要。如果选取的 p 不是质数，则可能会浪费散列地址。比如，选择 p 为偶数，设 key 值都为奇数，则 Hash(key) = key % p，结果为奇数，一半的存储单元被浪费。如果选 p 为 95，设 key 值都为 5 的倍数，则 Hash(key) = key % p，结果为 0，5，10，15，…，90，4/5 的存储单元被浪费。

4．平方取中法

此方法在词典处理中使用十分广泛。先计算关键字的平方值，从而扩大相近数的差别，然后根据散列表长度取中间的几位数（往往取二进制的比特位）作为散列函数值。因为一个乘积的中间几位数与乘数的每一数位都相关，所以由此产生的散列地址较为均匀。

例如，将一组关键字（0100，0110，1010，1001，0111）平方后得
（0010000，0012100，1020100，1002001，0012321）

若取表长为 1000，则可取中间的三位数作为散列地址：
（100，121，201，020，123）。

5．基数转换法

将关键字转换成另一种进制的数，然后计算散列地址。例如十进制的 345，转换成 9 进制数就是 423，可以用这个值对散列表长取余作为其散列地址。

6．折叠法

把关键字自左到右分成位数相等的几部分，每一部分的位数应与散列表地址位数相同，只有最后一部分的位数可以短一些。把这些部分的数据叠加起来，就可以得到具有该关键字的记录的散列地址。叠加方法有两种：

1）移位法——把各部分的最后一位对齐相加；

2）分界法——各部分不折断，沿各部分的分界来回折叠，然后对齐相加，将相加的结果作为散列地址。

例如，设给定的关键字为 key = 96234657，若散列表长为 1000，则划分结果为每段 3 位。上述关键字就可以划分为 3 段：

962　　346　　57

把这 3 部分相加后，去掉超出地址位数的最高位，仅保留最低的 3 位，作为散列地址。

移位法：962+346+57 = 1365；

分界法：962+643+57 = 1662；

一般当关键字的位数很多，而且关键字每一位上数字的分布大致比较均匀时，可用这种方法得到散列地址。

5.3.3　冲突解决方法

冲突是指对不同的关键字，由散列函数得到相同的散列地址的现象。几乎所有的散列函数都会出现多个关键字同时映射到同一个位置的现象。通过恰当的设计散列函数可以尽量避免冲突的出现，但是由于存储地址是有限的，因此不能完全避免冲突。冲突问题需要用某种方法来处理，从而保证冲突得到解决。

常用的解决冲突的方法有三种：开放定址法、链接法、桶定址法。

1．开放定址法

开放定址法把所有的记录直接存储在散列表中。设某个记录的关键字为 key，则该记录的初始探查位置为 f(key)。如果要插入一个记录 R，而另一个记录已经占据了 R 的初始探查位置（发生冲突），那么就把 R 存储在表中的其他地址中，由具体的冲突解决策略确定后继探查地址。

开放定址法解决冲突的基本思想是：当冲突发生时，使用某种方法为关键字 key 生成一个探查地址序列 A_0，A_1，…，A_i，…，A_{m-1}。其中 $A_0 = f(key)$，即为 key 的初始探查位置；所有 A_i ($0<i<m$)是后继探查地址。当插入 key 时，若初始探查位置已被别的数据元素占据，则按上述地址序列依次探查，将找到的第一个空闲位置 A_i 作为关键字 key 的存储位置；若所有后继探查地址都不空闲时，说明该散列表已满，报告溢出。相应的，查找关键字 key 时，首先与初始探查地址中数据比较，如果不相等，将按同样的后继地址序列依次查找，查找成功时返回该位置 A_i；如果沿探查序列查找时遇到了空闲的地址，则说明表中没有待查的关键字。删除关键字 key 时，也按同样的探查地址序列依次查找，如果查找到 key 的存储位置 A_i 后，则删除该位置 A_i 上的数据元素（删除操作实际上只是对该结点加以删除标记）；如果遇到了空闲地址，则说明表中没有待删除的关键字。

依据探查地址序列的生成方法，开放定址法主要可以分为线性探查法、二次探查法、伪随机探查法和双散列法。

（1）线性探查法

将散列表看成是一个环形表。设散列表长为 m。若在初始探查地址 d（即 f(key) = d ）发生冲突，则依次探查下述地址单元：$A_i = (d + i)\%m$，其中 0<i<m，直到找到一个空闲地址。若沿着该探查地址序列探查一遍之后，又回到了地址 d，则意味着失败。

例如，设散列表为 HT[23]，散列函数为 Hash(key)=key%23。存放的关键字序列是 32，75，29，63，48，94，25，46，22，55。采用线性探查法处理冲突。

上述关键字的探查地址序列分别是：

Hash(32)=9； Hash(75)=6；

Hash(29)=6（冲突），继续探查 7 号地址；

Hash(63)=17； Hash(48)=2；

Hash(94)=2（冲突），继续探查 3 号地址；

Hash(25)=2（冲突），继续探查 3 号地址（冲突），继续探查 4 号地址；

Hash(46)=0； Hash(22)=22；

Hash(45)=22（冲突），继续探查(22+1)%23=0 号地址（冲突），继续探查(22+2)%23=1 号地址；

将上述关键字存储到散列表后的情形如图 5-24 所示。

地址号	0	1	2	3	4	5	6	7	8	9	10	11	12	…	17	…	22
关键字	46	45	48	94	25		75	29		32					63		22
探查次数	1	3	1	2	3		1	2		1					1		1

图 5-24 线性探查法示例

利用线性探查法解决冲突后，在查找一个关键字时，首先根据其值计算出初始探查地址 Hash(key)，如果该地址中的关键字与查找的关键字相等，则查找成功；如果不等，则继续探查(Hash(key)+1)%m，(Hash(key)+2)%m，…，(Hash(key)+m-1)%m 地址中的关键字，直到查找成功或查找回到 Hash(key)的位置（查找失败）或探查到空闲位置（查找失败）。

例如，在上面的散列表中，查找 45，则需要依次探查以下地址 22，0，1，经过 3 次关键字的比较确定查找成功；如果要查找 69，则需要依次探查以下地址 0，1，2，3，4，5，因为 5 号地址空闲，因此可以确定不存在 69，查找失败，即 69 可以插入到 5 号地址。

对上面的散列表进行查找时，查找成功的平均查找长度为：

ASLsucc =(1+3+1+2+3+1+2+1+1+1)/10 = 1.6

查找不成功的平均查找长度为：

ASLunsucc=(6+5+4+3+2+3+2+2+2+7+13)/23 = 49/23

用线性探查法解决冲突时，当散列表中 i,i+1，…，i+k 的位置上已有结点时，散列地址为 i，i+1，…，i+k+1 的结点都将插入在位置 i+k+1 上。这种散列地址不同的结点

争夺同一个后继散列地址的现象称为冲突的一次聚集或堆积(Clustering)。冲突的聚集将造成不是同义词的结点也处在同一个探查地址序列之中，从而增加了探查序列的长度，即增加了查找时间。若散列函数不好或装填因子过大，都会使聚集现象加剧。

（2）二次探查法

为改善线性探查法的冲突“聚集”问题，减少查找成功时的平均探查次数，可使用二次探查法解决冲突。

二次探查法生成的后继探查地址不是连续的，而是跳跃的，以便为后续数据元素留下空间从而减少冲突聚集。设散列表长为 m，要存放的记录的关键字的初始探查地址为 d，二次探查法的探查序列依次是：d，$(d+1^2)\%m$，$(d-1^2)\%m$，$(d+2^2)\%m$，$(d-2^2)\%m$，…也就是说，发生冲突时，后继探查地址在初始探查地址 d 的两端。

例如，将关键字序列（27，71，40，49）存放到表长为 11 的散列表，散列函数为 Hash(key) = key%11，采用二次探查法解决冲突。

则各个关键字序列的探查地址依次为：

Hash(27)=5；

Hash(71)=5(冲突)，继续探查(5+1)%11=6；

Hash(40)=7；

Hash(49)=5(冲突)，继续探查(5+1)%11=6(冲突)，继续探查(5-1)%11=4；

相应的散列表如图 5-25 所示。

地址号	0	1	2	3	4	5	6	7	8	9	10
关键字					49	27	71	40			
探查次数					3	1	2	1			

图 5-25　二次探查法示例

对于该散列表，查找成功的平均查找长度为：

ASLsucc = (1+2+1+3)/4 = 7/4

查找不成功的平均查找长度为：

ASLunsucc = (3+4+4+2+7)/11 = 20/11

由于二次探查法生成的探查地址序列摇摆在初始探查地址两端，因此不易探查到整个散列表的所有位置，也就是说，后继探查地址序列可能难以包括散列表的所有存储位置。另外，虽然二次探查法排除了一次冲突聚集，但是仍然避免不了冲突的聚集，因为对散列到相同地址的关键字，采用的是同样的后继探查地址序列，这称为冲突的“二次聚集”。

（3）伪随机探查法

伪随机探查法是通过一个随机数生成器来生成随机的探查地址序列，可以防止冲突的“二次聚集”。

如果散列表长为 m，散列函数为 Hash()。随机数生成器第 i 次（$i \geq 1$）生成的随机

数为 p_i，则关键字 key 的探查地址序列为

Hash(key)，(Hash(key)+p_1)%m，……，(Hash(key)+p_i)%m，……

例如，设散列表长为 11，随机数生成器生成的前四个随机数为 r_1=1，r_2=5，r_3=3，r_4=17。假定关键字 k 的初始探查地址为 2，则 k 的探查地址序列为 2，3，7，5，8。

伪随机探查法的问题在于对于相同的关键字会产生不同的探查地址序列。所以在产生探查地址序列之前，相同的关键字必须使用相同的随机数种子来对随机数生成器进行初始化。

（4）双散列探查法

引起冲突二次聚集的原因是探查地址序列的方法是初始探查地址的函数，因此，当两个关键字的初始探查地址相同时，后继的探查序列就会相同。为了避免冲突的二次聚集，需要将探查地址序列定义为关键字的函数，而不是初始探查地址的函数。双散列探查法使用两个散列函数，第一个散列函数 Hash 用来计算数据记录的初始探查地址；第二个散列函数 ReHash 用来解决冲突。双散列法生成的探查序列为：

$$H_i = (Hash(key) + i * ReHash(key)) \% m，\ i = 0,1,2,\cdots, m-1$$

双散列法最多经过 m 次探查就会遍历表中所有位置，回到 Hash(key)位置。

例如，将一组关键字{ 22, 41, 53, 46, 30, 13, 01, 67 }存放到表长为 11 的散列表中，散列函数为 Hash(x)＝(3x) % 11，再散列函数为 ReHash(x) = (7x) % 10 +1。

各个关键字的探查地址序列为：

$H_0(22) = 0$；　$H_0(41) = 2$ ；　$H_0(53) = 5$；　$H_0(46) = 6$ ；

$H_0(30) = 2$　冲突；　继续探查 $H_1 = (2+1) \%11 = 3$

$H_0(13) = 6$　冲突；　继续探查 $H_1 = (6+2) \%11 = 8$

$H_0(01) = 3$　冲突；　继续探查 $H_1 = (3+8) \%11 = 0$

冲突；　继续探查 $H_2 = (0+8) \%11 = 8$　冲突；

继续探查 $H_3 = (8+8)\%11 = 5$　冲突；

继续探查 $H_4 = (5+8) \%11 = 2$　冲突；

继续探查 $H_5 = (2+8) \%11 = 10$

$H_0(67) = 3$　冲突；　继续探查 $H_1 = (3+10) \%11 = 2$　冲突；

继续探查 $H_2 = (2+10)\%11 = 1$

将各个关键字存储到散列后的情形如图 5-26 所示。

地址号	0	1	2	3	4	5	6	7	8	9	10
关键字	22	67	41	30		53	46		13		01
探查次数	1	3	1	2		1	1		2		6

图 5-26　双散列方法示例

对于上面的散列表，查找成功的平均查找长度

$$ASLsucc = (4+2+2+6+3)/8 = 17/8$$

双散列方法的查找不成功情况下平均搜索长度分析较为复杂。因为每一散列位置的移位量有 10 种：1, 2, …, 10。先计算每一散列位置各种移位量情形下找到下一个空闲地址的探查次数，求出平均值；再计算各个位置的平均比较次数的总平均值。

由于开放定址散列方法不使用链表结构，而是把所有记录都存储在散列表中，因此开放定址散列方法又称为“闭散列法”。

2．链接法

在链接法解决冲突的方法中，散列表中的每个地址都是一个链表的表头，关联着一个链表结构。散列到相同地址的记录都放在这个地址关联的链表中。应用这种方法的散列表不会产生溢出，因为链表会在加入新的关键字时扩展。链接法又称为“开散列法”、“拉链法”。

例如，设关键字为 {18 , 14 , 01 , 68 , 27 , 55 , 79 }，散列函数 Hash(key)=key %13，散列地址为 {5,1,1,3,1,3,1}。采用链接法解决冲突时的散列表如图 5-27 所示。

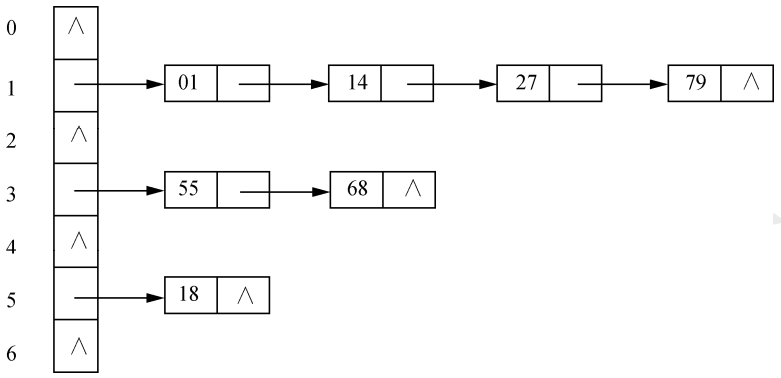


图 5-27　链接法解决冲突示例

通常情况下将一个链表中内容进行排序存放从而提高查找的执行性能。排序方式有多种：根据输入顺序，根据值的顺序或者根据访问频率的顺序。对于查找不成功的情况，根据值的顺序排列最为方便，因为一旦在链表中遇到一个比待查找的关键字大的关键字，就知道散列表中没有待查的关键字。反之，如果链表中的关键字没有排序，则需要访问每个关键字。

链接法处理冲突简单，不会出现冲突的聚集现象，所以平均查找长度较短。例如给定一个大小为 m，存储 n 个关键字的散列表，在理想情况下散列函数把 n 个记录均匀放置在表中 m 个位置上，使得每一个同义词的链表中平均有 n/m 个记录。假定表中的地址数比存储的记录数多，则出现包含多于一个记录的链表的可能性会很小。这样，散列方法的平均代价就是 O(1) 。然而，如果冲突使得许多记录集中分布到有限的几个链表中，

那么访问一个记录的代价就会更高，因为必须查询同义词链表中的许多元素。

如果整个散列表存储在内存中，用链接法比较容易实现。但是，如果整个散列表存储在磁盘中，将每个同义词存储在一个新地址的链接法就不太合适。因为一个同义词链表中的元素可能存储在不同的磁盘块中，这就会导致在查询一个特定关键字时多次访问磁盘，从而增加了查找时间。

3．桶定址法

桶定址法的基本思想是把记录分为若干存储桶，每个存储桶包含一个或多个存储位置，一个存储桶内的各存储位置用指针连接起来。散列函数 f 将关键字 key 映射到 f(key)号存储桶。如果桶已经满了，可以使用前面介绍的开放定址法来处理。

例如，利用桶定址法将关键字{16，13，14，27，20，24，31，35}存放到表长为 11 的散列表中，散列函数为 Hash(x)=x%11。如果某个地址关联的桶满了，则继续采用线性探查法解决冲突。

根据散列函数，将各个关键字存放到散列表后的情形如图 5-28 所示。在存储 35 时，其散列地址为 2，但是此时 2 号桶已经满了，采用线性探查法来探查临近的 3 号桶，从而存储到 3 号桶的空闲位置。在此之后，散列地址为 2、3 或 4 的关键字都将竞争 4 号桶，显然出现了冲突的聚集。

桶号	关键字	关键字
0		
1		
2	13	24
3	14	35
4		
5	16	27
6		
7		
8		
9	20	31
10		

图 5-28　桶定址法示例

5.3.4　散列算法设计与分析

在给定散列函数和冲突解决方法后，散列算法的具体实现如例 5.4 所示。

【例 5.4】散列查找。

```
template<class Key, class T> //Key 为关键码的类型，T 为存储在 HashTable
                             //中的元素的类型
class HashTable
```

```
{
private:
    T* HT;
    int maxSize;
    int currentSize;
    int probe(Key k,int i) const{return k%maxSize+i;};
                                          //探查函数，i 为探查的次数编号
    int hash(Key k) const{return k%maxSize;};          //散列函数
public:
    HashTable(int size)
    {
        maxSize=size;
        currentSize=0;
        HT=new T[maxSize];
    };
    ~HashTable(){delete []HT;}
    bool hashInsert(const T& item) const;      //插入数据
    bool hashSearch(const Key& k,T & item);    //检索数据
    bool hashDelete(const Key& k);             //删除数据
};
template <class Key, class T>
bool HashTable<Key,T>::hashInsert(const T& item) const
{
    int home=0;                                //存储的初始探查位置
    int i=0;                                   //探查的序列编号
    int pos=home=hash(item.getKey());
    while(!HT[pos].empty())
    {
        if(HT[pos]==item)
            return false;                      //发现重复
        i++;
        pos=(home+probe(item.getKey(),i))%maxSize; //probe 生成探查
                                                   //步长
    }
    HT[pos]=item;
    return true;
}
template <class Key, class T>
bool HashTable<Key,T>::hashSearch(const Key& k,T& item)
{
    int home=hash(k);
    int i=0;
    int pos=home;
    while(!HT[pos].empty())
    {
        if(HT[pos].getKey()==k)
        {
            item=HT[pos];
            return true;
        }
```

```
            i++;
            pos=(home+probe(k,i))%maxSize;
        }
        return false;
    }
    template <class Key, class T>
    bool HashTable<Key,T>::hashDelete(const Key& k)
    {
        int home=hash(k);
        int i=0;
        int pos=home;
        while(!HT[pos].empty())
        {
            if(HT[pos].getKey()==k)
            {
                HT[pos]=EMPTY;          //设置删除标志
                return true;
            }
            i++;
            pos=(home+probe(k,i))%maxSize;
        }
        return false;
    }
```

散列函数在记录的关键字与记录的存储位置之间直接建立了映射关系。当选择的散列函数能够得到均匀的地址分布时，在查找过程中可以不做多次探查。但是散列冲突增加了查找的时间。冲突的出现，与散列函数的选取（地址分布是否均匀），处理冲突的方法（是否产生堆积）有关。

通常情况下，开散列法优于闭散列法；在散列函数中，用除留余数法作散列函数优于其他类型的散列函数。当装填因子 α 较高时，选择的散列函数不同，散列表的搜索性能差别很大。一般情况下，散列的平均查找性能优于一些传统的技术，如平衡树，但是散列表在最坏情况下性能很不好。如对一个有 n 个关键码的散列表执行一次查找或插入操作，最坏情况下需要 O(n)的时间。

散列方法中不同的冲突解决方法也影响了散列表的平均查找长度，如表 5-2 所示。

表 5-2 各种冲突解决方法的平均查找长度

处理冲突的方法		平均查找长度 ASL	
		查找成功 Sn	查找不成功 Un
开放定址法	线性探查法	$\frac{1}{2}(1+\frac{1}{1-\alpha})$	$\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$
	伪随机探查法 二次探查法 双散列法	$-(\frac{1}{\alpha})\log_e(1-\alpha)$	$\frac{1}{1-\alpha}$
链接法		$1+\frac{\alpha}{2}$	$\alpha+e^{-\alpha}\approx\alpha$

其中 α 是散列表的装填因子，表明了表中的装满程度。装填因子越大，说明表越满，再插入新元素时发生冲突的可能性就越大。散列表的查找性能，即平均查找长度依赖于散列表的装填因子，而不直接依赖于存储的记录个数或散列表长度。

5.4 查找的应用：通讯录

1．需求描述

随着手机用户的逐渐增加，手机通讯录成为人们生活中必不可少的一个工具，一个具有良好性能的通讯录能够帮助用户快速的实现添加、查询等操作。这类通讯录的设计需要应用线性表、分块查找和散列查找的相关知识。本节介绍一个具有常用功能的简易手机通讯录的设计。

通讯录的功能如图 5-29 所示，具体描述如下：

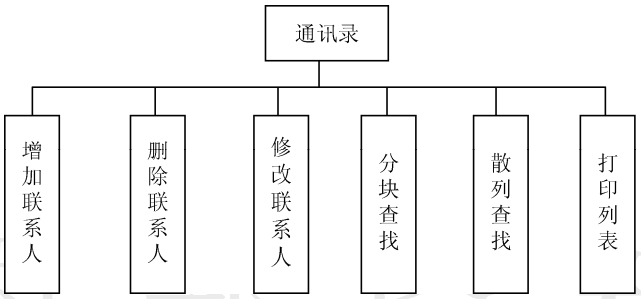


图 5-29 通讯录功能图

1）增加联系人，给出联系人的姓名、联系电话和 email 地址，增加联系人的过程中同时对联系人按姓名从小到大排列 。要求联系人姓名不能为空；

2）删除联系人，给出联系人的姓名进行删除操作；

3）修改联系人信息，联系人信息包括姓名、电话、email 地址，可以对任意信息进行修改；

4）分块查找联系人信息，如按照名字（汉语拼音）的首字母分块显示，并查询姓名以该字母开始的联系人；

5）散列查找联系人信息，给定联系人姓名后迅速查找该联系人的相关信息；

6）打印通讯录列表，向用户展示出通讯录中所有联系人的信息，包括姓名、联系电话和 email 地址。

2．问题分析

为了简化问题，这里限制通讯录中存储的姓名都为英文姓名，姓名、联系电话和 email 地址都是字符串类型。下面对通讯录中各个功能可能存在的问题进行解释。

本应用中，考虑到通用的手机通讯录都是按照姓名排序的，在增加联系人的时候按

照升序插入。

分块查找是指把线性表分成若干块，每块中结点的存放次序是任意的，但是块与块之间是按照关键值递增或递减的。分块查找中需要一个索引表来存储分块的信息。在通讯录中，按照姓名的首字母共分为 26 块，因为这里的联系人都是升序排列的，所以每块中的联系人也是升序排列的，先在索引表中保存每块的起始和结束位置即可。在每块中可以采用二分查找算法。

在散列查找中，首先需要构建一个散列表，散列表中存储所有联系人的信息。将联系人加入到散列表中时需要一个散列函数和一个冲突检测的探查函数。对于联系人，使用姓名作为关键字，散列的初始位置定义为姓名中每个字母的 ASCII 值相加再对散列表的最大容量取余数，若发生冲突则采用线性探查法依次向后探查。利用散列查找某一联系人的过程和增加联系人的过程是相同的。首先找到该联系人的初始散列位置，若该位置上的联系人和待查找的联系人不相同，则依次向后探查。

3．概要设计

通讯录是一个良好的封装联系人信息的结构，下面分析在通讯录中需要用到的类。首先，需要一个通讯录类 ContactList，ContactList 中应该包括一个存储联系人信息的容器，这里，选择定长的线性表 ArrayList，以及用于散列查找的散列表 HashTable。另外，还应该有一个封装联系人的各种信息的类 ContactInfor，联系人的信息包括姓名、联系电话、email，都采用字符串类型。ArrayList 和 HashTable 类在前面章节中都有介绍，这里只给出 ContactInfor 类和 ContactList 类的详细介绍。

ContactInfor 类中有成员变量 name，phoneNum 和 email，分别代表联系人姓名、联系电话和 email 地址。成员方法中有成员变量的 get 方法和重载的一些常用的运算符，以便在后续操作。另外，还有一个获取联系人关键字的 getKey 方法，该方法返回姓名的每个字母的 ASCII 相加值。

ContactList 中包括一个存储所有联系人的容器 ArrayList 和一系列的对联系人的操作。包括添加联系人方法 addContact，更新联系人信息方法 updateContact，删除联系人方法 delContact，分块查询方法 blockQuery，二分查询方法 binarySearch，散列查询方法 hashQuery。

该应用中每个类之间的关系如图 5-30 所示。

4．详细设计

下面对通讯录中的一些重要方法的设计进行详细解释。

bool ContactList::addContact(string name, string phoneNum,string email)

该方法以联系人的姓名、联系电话和 email 地址为参数。增加联系人的过程中因为是升序插入，首先要找到待插入的位置，然后调用 ArrayList 中的 insert 方法。详细过程如图 5-31 所示，其中 pos 代表待插入的位置，contact.Length()是通讯录的当前长度，contacts[pos]代表 pos 处的联系人，contacts.Insert(pos,contact)代表将联系人 contact 加入到位置 pos 处。

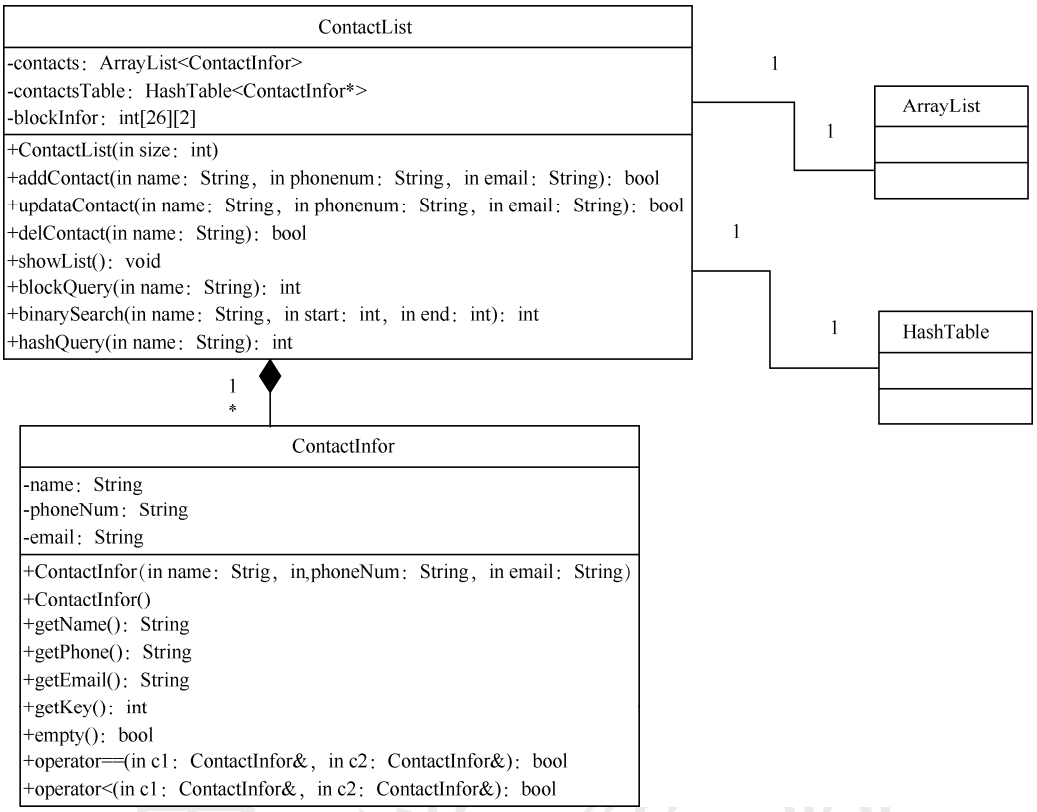


图 5-30　通讯录类图

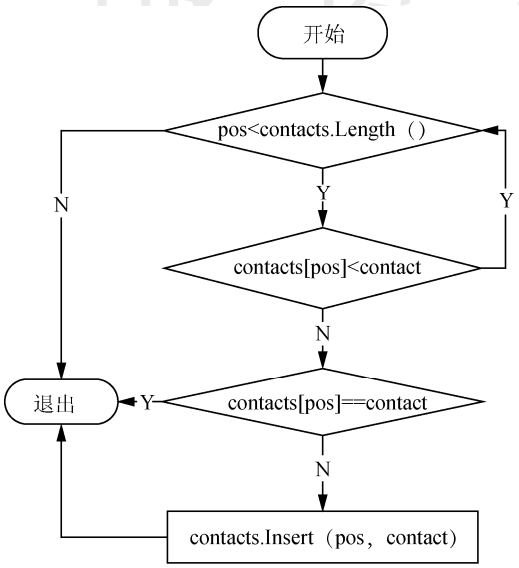


图 5-31　增加联系人流程图

（1）ContactInfor* ContactList::blockQuery(string name)

该方法以联系人的姓名作为关键值索引，分块查找中首先要对数据进行分块、建立索引表，然后进行查找，详细流程如图 5-32 所示。其中，数据分块、建立索引表就是按照姓名首字母分为 26 块，分块的信息存储在 blockInfor 中，blockInfor[i][0]、blockInfor[i][1]存放第 i 块（设以字母 a 或‘A’开始为第 0 块）的起始和结束位置，其流程如图 5-33 所示。建立好分块信息后，根据提供的姓名的首字母检索分块信息获得对应的起始和结束位置，然后在这个范围内进行折半查找。

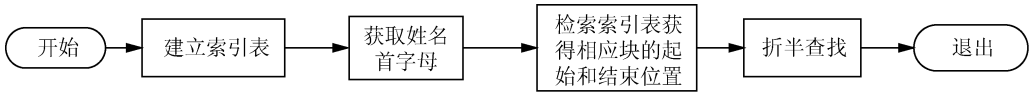


图 5-32　分块查找流程图

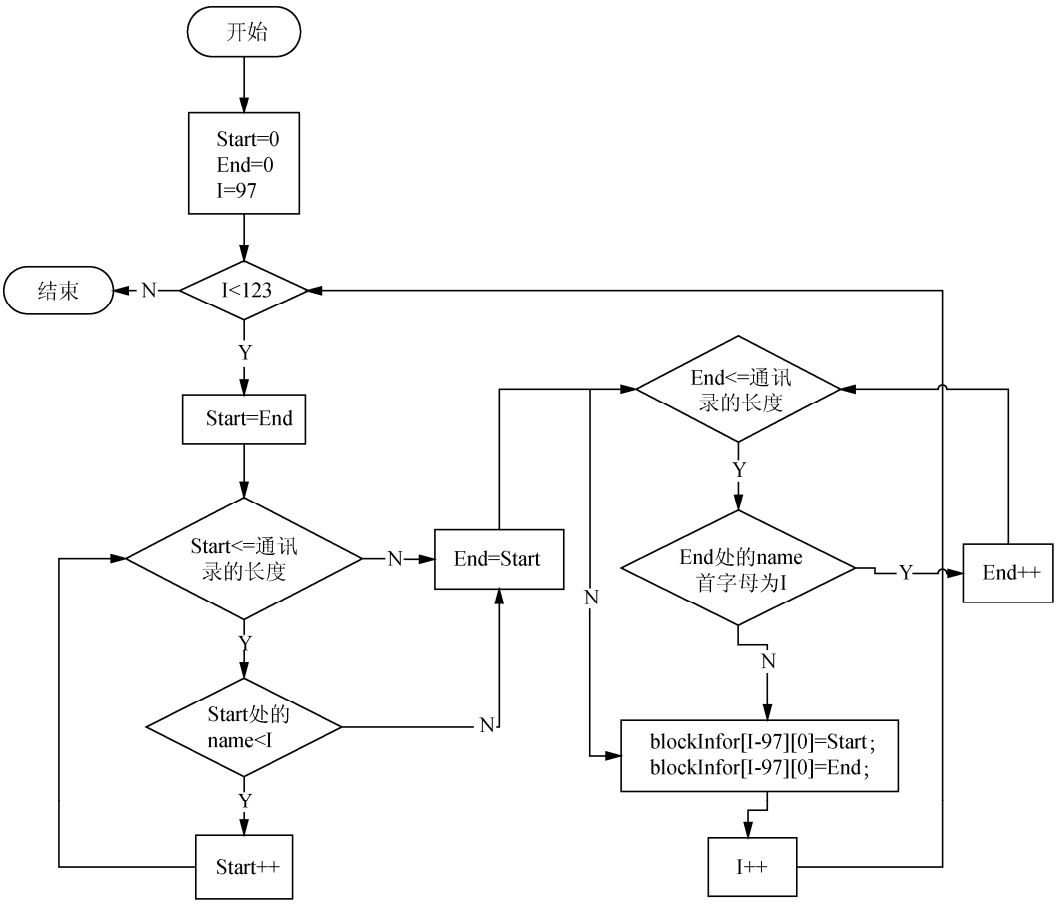


图 5-33　建立索引表流程图

（2）ContactInfor* ContactList::hashQuery(string name)

该方法同样以联系人姓名作为关键字索引，首先将 ArrayList 中的联系人复制到 ContactsTable 中，复制过程采用前面提到的散列和探查函数，然后再按照姓名在生成的散列表中查找。查找时调用散列表中提供的 hashSearch 方法。

习　　题

1．假定对有序表：（13,14,25,27,34,38,40,49,53,67,79,85）进行折半查找，试回答下列问题：

（1）画出描述折半查找过程的判定树；

（2）若查找元素 49，需依次与哪些元素比较?

（3）若查找元素 80，需依次与哪些元素比较?

（4）假定每个元素的查找概率相等，求查找成功时的平均查找长度。

2．用分块查找法在具有 4000 个数据元素的表中进行查找时，数据表分成多少块最理想？每块的理想长度是多少？若每块长度为 50 ，平均查找长度是多少？

3．设有一棵空的 3 阶 B-树，依次插入关键字 30,20,10,40,80,58,47,50,29,22,56,98,99，请画出该树。

并给出依次删除 29、47 之后的 B-树。

4．简要叙述 B-树与 B+树的区别。

5．设散列函数为 Hash(x)=3 x %11，散列表长为 11，输入关键字序列（32,13,49,24,38,21,4,12）。写出按下述两种解决冲突的方法构造的散列表。

（1）线性探查法；

（2）链接法；

（3）分别求出等概率下查找成功和查找失败的平均查找长度。

6．简述在使用线性探查法解决冲突的散列表中插入关键字、删除关键字的算法。

7．设输入的关键字序列为{71，23，73，99，44，89}，散列表长为 10，散列函数为 Hash(x) = x%10，请写出下列方法的散列结果。

（1）使用线性探查法解决冲突的开放定址散列表；

（2）使用双散列探查法解决冲突的开发定址散列表，其中第二散列函数为 RH(x) = 7-x%7；

（3）使用链接法解决冲突的散列表；

（4）分别求出等概率下查找成功和查找失败的平均查找长度。

8．设有 250 个关键字要存储到散列表中，要求利用线性探查法解决冲突，同时要求查找某个关键字所需的比较次数不超过 3 次，问该散列表至少要有多少个存储空间？

9．设采用单链表存储不重复的正整型数据元素，且每个结点含有 5 个元素（若最后一个结点的数据元素不满 5 个，以 0 补充），试编写一个算法查找值为 n（n>0）的

数据元素所在的结点指针以及在该结点中的序号，若链表中不存在该数据元素则返回空指针。

10．已知顺序表中有 m 个记录，表中记录不依关键字有序排列，编写算法为该顺序表建立一个有序的索引表，索引表中的每一项由该记录的关键字和该记录在顺序表中的序号构成，要求算法的时间复杂度在最好的情况下能达到 O(m)。