第 3 章　查找与排序技术

3.1　基本的查找技术

查找是数据处理领域中的一个重要内容，查找的效率将直接影响数据处理的效率。所谓查找，是指在一个给定的数据结构中查找某个指定的元素。通常，根据不同的数据结构，应采用不同的查找方法。

3.1.1　顺序查找

顺序查找又称为顺序搜索。顺序查找一般是指在线性表中查找指定的元素，其基本方法如下：

从线性表的第一个元素开始，依次将线性表中的元素与被查元素进行比较，若相等，则表示找到（查找成功）；若线性表中所有的元素都与被查元素进行了比较但都不相等，则表示线性表中没有要找的元素（查找失败）。

在这种查找方法下，如果线性表中的第一个元素就是被查找的元素，则只需做一次比较就查找成功，查找效率很高。但如果被查找的元素是线性表中的最后一个元素，或者被查找的元素根本不在线性表中，则为了查找这个元素，需要与线性表中所有的元素进行比较，这是顺序查找的最坏情况。在平均情况下，利用顺序查找法在线性表中查找一个元素，大约要与线性表中一半的元素进行比较。

由此可以看出，对于大的线性表来说，顺序查找的效率是很低的。虽然顺序查找的效率不高，但在下列两种情况下也只能采用顺序查找。

（1）如果线性表为无序表（表中元素的排列是无序的），则不管是顺序存储结构还是链式存储结构，都只能顺序查找。

（2）即使是有序线性表，如果采用链式存储结构，也只能顺序查找。

在实际应用中，为了提高查找效率，往往对数据采用特殊的存储结构。本节下面的内容就是讨论能提高查找效率的各种数据存储结构。

3.1.2　有序表的对分查找

对分查找只适用于顺序存储的有序表。在此所说的有序表是指线性表中的元素按值非递减排列（从小到大，但允许相邻元素值相等）。

设有序线性表的长度为 n，被查元素为 x，则对分查找的方法如下。

将 x 与线性表的中间项进行比较：若中间项的值等于 x，则说明查到，查找结束；若 x 小于中间项的值，则在线性表的前半部分（中间项以前的部分）以相同的方法进行查找；若 x 大于中间项的值，则在线性表的后半部分（中间项以后的部分）以相同的方法进行查

找。这个过程一直进行到查找成功或子表长度为 0(说明线性表中没有这个元素)时为止。

显然,当有序线性表为顺序存储时才能采用对分查找,并且,对分查找的效率要比顺序查找高得多。可以证明,对于长度为 n 的有序线性表,在最坏情况下,对分查找只需要比较 $\log_2 n$ 次,而顺序查找需要比较 n 次。

下面对顺序有序表类用 C++ 进行描述,其中的操作包括有序表的初始化、查找、插入、删除、输出以及将两个有序表合并成一个有序表。

```
//SL_List.h
    #include<iostream.h>
    //using namespace std;
    template<class T>                       //模板声明,数据元素虚拟类型为 T
    class  SL_List                          //顺序有序表类
    { private:                              //数据成员
        int mm;                             //存储空间容量
        int nn;                             //有序表长度
        T * v;                              //有序表存储空间首地址
      public:                               //成员函数
        SL_List(){ mm=0; nn=0; return; }    //只定义对象名
        SL_List(int);                       //顺序有序表初始化(指定存储空间容量)
        int search_SL_List(T);              //顺序有序表查找
        int insert_SL_List(T);              //顺序有序表插入
        int delete_SL_List(T);              //顺序有序表删除
        void prt_SL_List();                 //顺序输出有序表中的元素与有序表长度
        friend SL_List operator+(SL_List &, SL_List &);       //有序表合并
    };
    //顺序有序表初始化
    template<class T>
    SL_List<T>::SL_List(int m)
    { mm=m;                                 //存储空间容量
      v=new T[mm];                          //动态申请存储空间
      nn=0;                                 //有序表长度
      return;
    }
    //顺序有序表查找
    template<class T>
    int SL_List<T>::search_SL_List(T x)
    { int i, j, k;
      i=1; j=nn;
      while (i<=j)
         { k=(i+j)/2;                       //中间元素下标
           if (v[k-1]==x) return(k-1);      //找到返回
           if (v[k-1]>x) j=k-1;             //取前半部
           else i=k+1;                      //取后半部
         }
      return(-1);                           //找不到返回
    }
    //顺序有序表的插入
    template<class T>
    int SL_List<T>::insert_SL_List(T x)
    { int k;
```

```
  if (nn==mm)                    //存储空间已满
  { cout<<"上溢!"<<endl; return(-1); }
  k=nn-1;
  while (v[k]>x)                 //从最后一个元素到被删元素之间的元素均后移
  { v[k+1]=v[k]; k=k-1; }
  v[k+1]=x;                      //进行插入
  nn=nn+1;                       //长度增 1
  return(1);                     //成功插入返回
}
//顺序有序表的删除
template<class T>
int SL_List<T>::delete_SL_List(T x)
{ int i, k;
  k=search_SL_List(x);           //查找删除元素的位置
  if (k>=0)                      //表中有这个元素
     { for (i=k; i<nn-1; i++)
         v[i]=v[i+1];            //被删元素到最后一个元素之间的元素均前移
       nn=nn-1;                  //长度减 1
     }
  else                           //表中没有这个元素
      cout<<"没有这个元素!"<<endl;
  return(k);                     //返回删除是否成功标志
}
//顺序输出有序表中的元素与顺序表的长度
template<class T>
void SL_List<T>::prt_SL_List()
{ int i;
  cout<<"nn="<<nn<<endl;         //输出长度
  for (i=0; i<nn; i++)           //依次输出元素
      cout<<v[i]<<endl;
  return;
}
//有序表合并(运算符+重载为友元函数)
template<class T>
SL_List<T>operator+(SL_List<T>&s1, SL_List<T>&s2)
{ int  k=0, i=0, j=0;
  SL_List<T>s;
  s.v=new T[s1.nn+s2.nn];        //动态申请存储空间
  while((i<s1.nn)&&(j<s2.nn))
      { if (s1.v[i]<=s2.v[j])//复制有序表 s1 的元素
          { s.v[k]=s1.v[i]; i=i+1; }
        else                     //复制有序表 s2 的元素
          { s.v[k]=s2.v[j]; j=j+1; }
        k=k+1;
      }
  if (i==s1.nn)                  //复制有序表 s2 中剩余的元素
      for (i=j; i<s2.nn; i++)
      { s.v[k]=s2.v[i]; k=k+1; }
  else                           //复制有序表 s1 中剩余的元素
      for (j=i; j<s1.nn; j++)
      { s.v[k]=s1.v[j]; k=k+1; }
  s.mm=s1.mm+s2.mm;
  s.nn=s1.nn+s2.nn;
```

```
    return(s);
}
```

需要说明的是，在上述程序中，前两行

```
#include<iostream.h>
//using namespace std;
```

应为

```
#include<iostream>
using namespace std;
```

由于在 Visual C++ 6.0 不支持将运算符重载函数作为友元函数，而在 Visual C++ 6.0 所提供的带后缀 h 的头文件中支持，因此，作者在调试该程序时对前两行进行了修改。

下面举例说明。

例 3.1　分别定义容量为 20 与 30 的两个有序表空间，然后依次在这两个空间中分别插入 5 个和 7 个元素，输出这两个有序表中的元素。将这两个有序表合并成一个新的有序表，然后输出该新的有序表中的元素。在这个新的有序表中依次删除 1.5、1.0 和 123.0 三个元素，然后输出该新的有序表中的元素。

主函数程序如下：

```
//ch3_1.cpp
    #include "SL_List.h"
    int main()
    { int k;
      double a[5]={1.5,5.5,2.5,4.5,3.5};
      double b[7]={1.0,7.5,2.5,4.0,5.0,4.5,6.5};
      SL_List<double>s1(20);          //建立容量为 20 长度为 5 的有序表对象 s1
      SL_List<double>s2(30);          //建立容量为 30 长度为 7 的有序表对象 s2
      for (k=0; k<5; k++)             //依次插入有序表 s1 的元素
          s1.insert_SL_List(a[k]);
      for (k=0; k<7; k++)             //依次插入有序表 s2 的元素
          s2.insert_SL_List(b[k]);
      cout<<"输出有序表对象 s1:"<<endl;
      s1.prt_SL_List();               //输出有序表对象 s1
      cout<<"输出有序表对象 s2:"<<endl;
      s2.prt_SL_List();               //输出有序表对象 s2
      SL_List<double>s3;              //建立合并后的有序表对象 s3
      s3=s1+s2;                       //有序表合并
      cout<<"输出合并后的有序表对象 s3:"<<endl;
      s3.prt_SL_List();               //输出合并后的有序表对象 s3
      s3.delete_SL_List(a[0]);        //在有序表 s3 中删除 1.5
      s3.delete_SL_List(b[0]);        //在有序表 s3 中删除 1.0
      s3.delete_SL_List(123.0);       //在有序表 s3 中删除 123.0
      cout<<"输出删除后的有序表对象 s3:"<<endl;
      s3.prt_SL_List();               //输出合并后的有序表对象 s3
      return 0;
    }
```

上述程序的运行结果如下：

```
输出有序表对象 s1:
nn=5
1.5
2.5
3.5
4.5
5.5
输出有序表对象 s2:
nn=7
1
2.5
4
4.5
5
6.5
7.5
输出合并后的有序表对象 s3:
nn=12
1
1.5
2.5
2.5
3.5
4
4.5
4.5
5
5.5
6.5
7.5
没有这个元素!          //指删除的元素 123.0
输出删除后的有序表对象 s3:
nn=10
2.5
2.5
3.5
4
4.5
4.5
5
5.5
6.5
7.5
```

3.1.3 分块查找

分块查找又称为索引顺序查找,它是一种顺序查找的改进方法,用于在分块有序表中进行查找。所谓分块有序表,是指将长度为 n 的线性表分成 m 个子表。各子表的长度可以相等,也可以互相不等,但要求后一个子表中的每一个元素均大于前一个子表中的所有元素。

分块有序表的结构可以分为两部分。

(1) 线性表本身采用顺序存储结构。

(2) 再建立一个索引表。在索引表中,对线性表的每个子表建立一个索引结点,每个结点包括两个域:一是数据域,用于存放对应子表中的最大元素值;二是指针域,用于指示对应子表的第一个元素在整个线性表中的序号。显然,索引表关于数据域是有序的。

索引表中每一个索引结点用 C++ 定义如下:

```
template<class T>
struct  indnode
{ T  key;          //数据域,存放子表中的最大值,其类型与线性表中元素的数据类型相同
  int  k;          //指针域,指示子表中第一个元素在线性表中的序号
};
```

图 3.1 是将一个长度 $n=18$ 的线性表分成 $m=3$ 个子表的分块有序表的示意图。

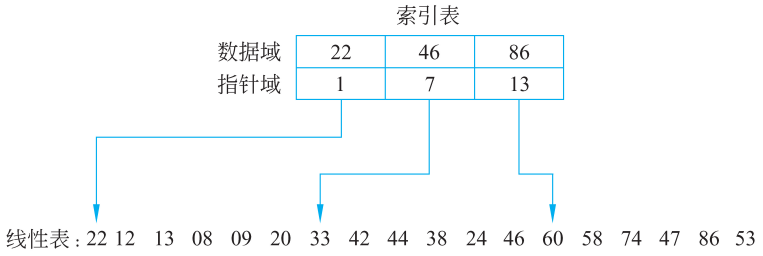

图 3.1　分块有序表例

根据分块有序表的结构,其查找过程可以分以下两步进行:

(1) 查找索引表,以便确定被查元素所在的子表。由于索引表数据域中的数据是有序的,因此可以采用对分查找。

(2) 在相应的子表中用顺序查找法进行具体的查找。

在分块查找中,为了找到被查元素 x 所在的子表,需要对分查找索引表,在最坏情况下需要查找$\log_2(m+1)$次。为了在相应子表中用顺序查找法查找元素 x,在最坏情况下需要查找 n/m(假设各子表长度相等)次;而在平均情况下需要查找 $n/2m$ 次。因此,分块查找的工作量为:在最坏情况下需要查找$\log_2(m+1)+n/m$ 次,平均情况下需要查找$\log_2(m+1)+n/(2m)$次。显然,当 $m=n$ 时,线性表 L 即为有序表,分块查找即为对分查找。当 $m=1$ 时,线性表 L 为一般的无序表,分块查找即为顺序查找。因此,分块查找的效率介于对分查找和顺序查找之间。

分块查找的算法由读者自行编写。

3.2　哈希表技术

在 3.1 节所介绍的各种查找方法中,都是通过被查元素与表中的元素进行直接比较而达到查找的目的。本节所要介绍的哈希(Hash)表技术是另一类重要的查找技术。Hash 表技术的基本思想是对被查元素的关键字做某种运算后直接确定所要查找项目在表中的位置。本节主要介绍 Hash 表的基本概念以及工程上常用的几种 Hash 表。在具体介绍 Hash 表之前,首先介绍直接查找表。

3.2.1 哈希表的基本概念

1. 直接查找技术

设表的长度为 n。如果存在一个函数 $i=i(k)$,对于表中的任意一个元素的关键字 k,满足以下条件:

(1) $1\leqslant i\leqslant n$;

(2) 对于任意的元素关键字 $k_1\neq k_2$,恒存在 $i(k_1)\neq i(k_2)$。

则称此表为直接查找表。其中,函数 $i=i(k)$称为关键字 k 的映像函数。

由直接查找表的定义可以看出,直接查找表中各元素的关键字 k 与表项序号 i 之间存在着一一对应的关系。因此,对直接查找表的查找,只需要根据映像函数 $i=i(k)$计算出待查关键字项目在表中的序号即可。

对直接查找表的操作主要有以下两种。

1) 直接查找表的填入

要将关键字为 k 的元素填入直接查找表,只需做以下两步:

(1) 计算关键字 k 的映像函数 $i=i(k)$;

(2) 将关键字 k 及有关元素信息填入表的第 i 项。

2) 直接查找表的取出

要在直接查找表中取出关键字 k 的元素,只需做以下两步:

(1) 计算关键字 k 的映像函数 $i=i(k)$;

(2) 检查表中第 i 项:

若第 i 项为空,则说明表中没有关键字为 k 的元素;否则取出第 i 项中的元素即可。

如果直接查找表中的各元素在存储空间中均占 m 字节,则关键字为 k 的元素在存储空间中实际的存储地址为

$$\text{直接查找表的首地址}+(i-1)m$$

其中 $i=i(k)$。为了讨论方便,在本节中所讨论的表只考虑其逻辑结构,而不考虑其实际的存储结构。在这种情况下,对长度为 n 的表,总是认为 $1\leqslant i\leqslant n$。

2. Hash 表

Hash 表技术是直接查找技术的推广,其主要目标是提高查找效率。

在直接查找技术中,要求映像函数能使得表中任意两个不同的关键字的映像函数值也不同。即在直接查找表中,不允许元素的冲突存在。但在实际应用中,这一条件一般是很难满足的,即往往会出现这样的情况:对于两个不同的关键字 $k_1\neq k_2$ 有 $i(k_1)=i(k_2)$,这就发生了元素的冲突,即两个不同的元素需要存放在同一个存储单元中。

例 3.2 将关键字序列(09,31,26,19,01,13,02,11,27,16,05,21)依次填入长度为 $n=12$ 的表中。设映像函数为 $i=\text{INT}(k/3)+1$。其中,INT 为取整符。

如果按照直接查找表的填入方法,填入结果如表 3.1 所示。由表 3.1 可以看出,两个不同的关键字元素 01 与 02 的映像函数值(计算得到的表项序号)是相同的,称这两个元素发生了冲突。同样,09 与 11 这两个元素也发生了冲突。

表 3.1　直接查找表的填入

表项序号	1	2	3	4	5	6	7	8	9	10	11	12
按 $i=\mathrm{INT}(k/3)+1$ 填入的关键字 k	01 02	05		09 11	13	16	19	21	26	27	31	

显然，当有元素发生冲突时，是无法进行直接查找的。为此，引入 Hash 表的概念。

设表的长度为 n。如果存在一个函数 $i=i(k)$，对于表中的任意一个元素的关键字 k，满足 $1\leqslant i\leqslant n$，则称此表为 Hash 表。其中函数 $i=i(k)$称为关键字 k 的 Hash 码。

由 Hash 表的这个定义可以看出，在 Hash 表中允许冲突存在，即在 Hash 表中允许几个不同的关键字其 Hash 码相同。如果在 Hash 表中没有冲突存在，则 Hash 表就成为直接查找表。

Hash 表技术的关键是要处理好表中元素的冲突问题，主要包括以下两方面的工作：

(1) 构造合适的 Hash 码，以便尽量减少表中元素冲突的次数，即 Hash 码的均匀性要比较好。

(2) 当表中元素发生冲突时，要进行适当的处理。

3. Hash 码的构造

Hash 表技术的主要目标是提高查找效率，即缩短查表的时间。而在查找关键字为 k 的元素时，计算 Hash 码 $i(k)$的工作量也是影响查找效率的一个因素。如果 Hash 码的计算比较复杂，那么，尽管 Hash 码冲突的机会很少，也会降低查找的效率。因此，在实际设计 Hash 码时，要考虑以下两方面的因素：

(1) 使各关键字尽可能均匀地分布在 Hash 表中，即 Hash 码的均匀性要好，以便减少冲突发生的机会。

(2) Hash 码的计算要尽量简单。

以上两方面在实际应用中往往是矛盾的。为了保证 Hash 码的均匀性比较好，其 Hash 码的计算就必然要复杂；反之，如果 Hash 码的计算比较简单，则其均匀性就比较差。因此，在实际设计 Hash 码时，要根据具体情况选择一个比较合理的方案。例如，当 Hash 表放在慢速的二级存储器中时(用于文件系统中时往往是这种情况)，由于存取表中元素所需的时间较长，因此，在这种情况下，应主要考虑减少表中元素的冲突，而 Hash 码的计算稍微复杂一些是无关紧要的；当 Hash 表放在计算机内存中时，则应使 Hash 码的计算尽量简单，此时虽然 Hash 码冲突机会稍多一些，但在总体上考虑还是划算的。

由于 Hash 码的设计在很大程度上依赖于各关键字的特性，因此，一般很难给出一个普遍适用的方案，只能根据具体情况设计构造 Hash 码的方案。下面仅介绍一些比较简单的 Hash 码的构造方法。

1) 截段法

关键字是一种基本的符号串，而在计算机中就是一个经过编码的二进制数字串。所谓截段法，是指选取与关键字对应的数字串中的一段(一般选取低位数)作为该关键字的 Hash 码。在这种方法中，对数字串所截取的位数取决于 Hash 表的长度 n。一般截取的位数为 $\log_2 n$。

在实际应用中，截段法往往作为选取 Hash 码的基础，其他方法往往是对关键字进行某

种运算后再进行截段。

2) 分段叠加法

这种方法是将关键字的编码串分割成若干段,然后把它们叠加后再进行截段。

3) 除法

这种方法是将关键字的编码除以表的长度,最后所得的余数作为 Hash 码,即取 Hash 码为

$$i = \mathrm{mod}(k, n)$$

其中 k 为关键字,n 为 Hash 表的长度,mod 为模余运算符(在本节中规定,当 $\mathrm{mod}(k, n) = 0$ 时,取 $i = n$)。

本方法构造的 Hash 码,其均匀性比较好,但是以一次除法为代价,在除法较快的机器上可以采用。

4) 乘法

将关键字编码与一个常数 ϕ 相乘后,再除以表长度 n 取其余数作为 Hash 码,即

$$i = \mathrm{mod}(k * \phi, n)$$

或者将关键字编码与 ϕ 相乘后,取乘积低位段中的若干二进制位(进行截段)作为 Hash 码。其中常数 ϕ 一般取 0.618033988747、0.6125423371 或 0.6161616161。

构造 Hash 码的方法有很多,读者可以根据具体情况自行设计。为了能得到均匀性较好的 Hash 码,一般需要做多次试验,以检查 Hash 码的均匀性是否满足要求。若不满足均匀的要求,则要找出不均匀的原因,适当修改构造 Hash 码的方法,然后进行试验,直到 Hash 码的均匀性基本满足要求为止。

3.2.2 几种常用的哈希表

前面提到,Hash 表技术的关键之一是要处理好元素的冲突。采用不同的方法处理冲突就可以得到各种不同的 Hash 表。本节介绍几种常用的 Hash 表,在各种 Hash 表的查找(填入与取出)方法中体现了各种不同的处理冲突的方法。

1. 线性 Hash 表

线性 Hash 表是一种最简单的 Hash 表。

设线性 Hash 表的长度为 n,对线性 Hash 表的查找过程如下。

1) 线性 Hash 表的填入

将关键字 k 及有关信息填入线性 Hash 表的步骤如下。

(1) 计算关键字 k 的 Hash 码 $i = i(k)$。

(2) 检查表中第 i 项的内容:

若第 i 项为空,则将关键字 k 及有关信息填入该项;若第 i 项不空,则令 $i = \mathrm{mod}(i+1, n)$,转(2)继续检查。

显然,只要 Hash 表尚未填满,最终总可以找到一个空项,将关键字 k 及有关信息填入 Hash 表。

2) 线性 Hash 表的取出

要在线性 Hash 表中取出关键字 k 的元素,其步骤如下。

(1) 计算关键字 k 的 Hash 码 $i = i(k)$。

(2) 检查表中第 i 项的内容：

若第 i 项登记着关键字 k，则取出该项元素即可；若第 i 项为空，则表示在 Hash 表中没有该关键字的信息；若第 i 项不空，且登记的不是关键字 k，则令 $i=\mathrm{mod}(i+1,n)$，转(2)继续检查。

显然，只要 Hash 表尚未填满，这个过程就能够很好地终止，要么找到后取出该关键字 k 及有关信息；要么发现了一个空项，以找不到终止。

线性 Hash 表的这种处理冲突的方法又称为开放法。

例 3.3　将关键字序列(09,31,26,19,01,13,02,11,27,16,05,21)依次填入长度为 $n=12$ 的线性 Hash 表。设 Hash 码为 $i=\mathrm{INT}(k/3)+1$。

填入后的线性 Hash 表如表 3.2 所示。

表 3.2　线性 Hash 表的填入

表项序号	1	2	3	4	5	6	7	8	9	10	11	12
关键字 k	01	02	05	09	13	11	19	16	26	27	31	21
冲突次数	0	1	1	0	0	2	0	2	0	0	0	4

线性 Hash 表的优点是简单，但它有以下两个主要缺点：

(1) 在线性 Hash 表填入的过程中，当发生冲突时，首先考虑的是下一项，因此，当 Hash 码的冲突较多时，在线性 Hash 表中会存在“堆聚”现象，即许多关键字被连续登记在一起，从而会降低查找效率。

(2) 在线性 Hash 表的填入过程中，处理冲突时会带来新的冲突。通过比较表 3.1 与表 3.2，可以明显地看出线性 Hash 表的这个缺点。表 3.1 与表 3.2 是同一批序列按同一种 Hash 码填入的结果，只是在表 3.1 中对元素的冲突没有经过处理，而在表 3.2 中对元素的冲突用开放法进行了处理。由表 3.1 可知，这一批元素在使用该 Hash 码填入时，实际上只有两对元素发生了冲突(称为静态冲突)；但从表 3.2 可以看出，随着对冲突的处理，有些本来不发生静态冲突的元素也发生了冲突，这是由于这些元素应有的表项序号，在先前处理其他元素的冲突时被占用了，即线性 Hash 表的填入方法是不顾后效的。

在线性 Hash 表中，查找一个关键字的平均查找次数为

$$E \approx \frac{2-\alpha}{2-2\alpha}$$

其中 α 称为 Hash 表的填满率(也称为装填因子)，它定义为

$$\alpha = m/n$$

其中 n 为 Hash 表的长度，m 为 Hash 表中实际存在的关键字个数。值得注意的是，Hash 表的平均查找次数 E 不依赖于表的长度，而只与表的填满率有关。当 $\alpha=1$(Hash 表已经被填满)时，其平均查找次数为无穷。因此，线性 Hash 表不应填满，否则在 Hash 表中查找一个不存在的关键字元素时会出现无穷循环，这是在设计 Hash 表长度时必须要注意的问题。

线性 Hash 表还有更一般的形式，当发生元素冲突时，将处理方案 $i=\mathrm{mod}(i+1,n)$ 改为 $i=\mathrm{mod}(i+p,n)$。可以证明，只要 p 与 n 互质，这种处理同样可以遍历整个 Hash 表。

最后需要说明一点，在实际设计线性 Hash 表时，为了对表长 n 取模余运算方便，通常

将表的长度 n 设计成 $n=2^k$,此时,某个数对 n 取模余运算时只要取这个数的右边 k 个二进制位即可。

另外,当 $i(k)=1$ 时,线性 Hash 表的查找就是顺序查找。

下面是对线性 Hash 表类的 C++ 描述:

```
//Linear_hash.h
    #include<iostream>
    using namespace std;
    //线性 Hash 表结点类型
    template<class T>
    struct Hnode
    { int flag;                                    //标志表项的空与非空
      T  key;                                      //关键字
    };
    template<class T>                              //模板声明,数据元素虚拟类型为 T
    class  Linear_hash                             //线性 Hash 表类
    { private:                                     //数据成员
        int NN;                                    //线性 Hash 表长度
        Hnode<T> * LH;                             //线性 Hash 表存储空间首地址
      public:                                      //成员函数
        Linear_hash(){ NN=0; return;}
        Linear_hash(int);                          //建立线性 Hash 表存储空间
        void prt_L_hash();                         //顺序输出线性 Hash 表中的元素
        int flag_L_hash();                         //检测线性 Hash 表中空项个数
        void ins_L_hash(int ( * f)(T), T);         //在线性 Hash 表中填入新元素
        int sch_L_hash(int ( * f)(T), T);          //在线性 Hash 表中查找元素
    };

    //建立线性 Hash 表存储空间
    template<class T>
    Linear_hash<T>::Linear_hash(int m)
    { int k;
      NN=m;                                        //存储空间容量
      LH=new Hnode<T>[NN];                         //动态申请线性 Hash 表存储空间
      for (k=0; k<NN; k++)                         //线性 Hash 表各项均为空
          LH[k].flag=0;
      return;
    }

    //顺序输出线性 Hash 表中的元素
    template<class T>
    void Linear_hash<T>::prt_L_hash()
    { int k;
      for (k=0; k<NN; k++)
        if (LH[k].flag==0)                         //该项为空
            cout<<"<NULL>"<<" ";
        else                                       //该项不空
            cout<<"<"<<LH[k].key<<">";
      cout<<endl;
      return;
    }
```

```
//检测线性 Hash 表中空项个数
template<class T>
int Linear_hash<T>::flag_L_hash()
{ int k, count=0;
  for (k=0; k<NN; k++)
    if (LH[k].flag==0) count=count+1;
  return(count);
}

//在线性 Hash 表中填入新元素
template<class T>
void Linear_hash<T>::ins_L_hash(int (*f)(T), T x)
{ int k;
  if (flag_L_hash()==0)                          //线性 Hash 表中已没有空项
  { cout<<"线性 Hash 表已满!"<<endl; return; }
  k=(*f)(x);                                     //计算 Hash 码
  while (LH[k-1].flag)                           //该项不空
  { k=k+1;                                       //下一项
    if (k==NN+1) k=1;
  }
  LH[k-1].key=x; LH[k-1].flag=1;                 //填入并置标志
  return;
}

//在线性 Hash 表中查找元素
template<class T>
int Linear_hash<T>::sch_L_hash(int (*f)(T), T x)
{ int k;
  k=(*f)(x);                                     //计算 Hash 码
  while ((LH[k-1].flag)&&(LH[k-1].key!=x))
  { k=k+1;
    if (k==NN+1) k=1;
  }
  if ((LH[k-1].flag)&&(LH[k-1].key==x))          //找到返回
      return(k);
  return(0);                                     //表中没有这个关键字,返回
}
```

下面是例 3.3 的主函数。

```
//ch3_2.cpp
  #include "Linear_hash.h"
  int hashf(int k);
  int main()
  { int a[13]={9,31,26,19,1,13,2,11,27,16,5,21};
    int k;
    Linear_hash<int>h(12);          //建立容量为 12 的线性 Hash 表空间
    cout<<"填入的原序列:"<<endl;
    for (k=0; k<12; k++)
        cout<<a[k]<<"  ";
    cout<<endl;
    for (k=0; k<12; k++)
        h.ins_L_hash(hashf, a[k]);
    cout<<"依次输出线性 Hash 表中的关键字:"<<endl;
```

```
    h.prt_L_hash();
    cout<<"查找序列各关键字在线性 Hash 表中的位置(表项序号):"<<endl;
    for (k=0; k<12; k++)
        cout<<h.sch_L_hash(hashf, a[k])<<"  ";
    cout<<endl;
    return 0;
}
int hashf(int k)  //Hash 函数
{ return(k/3+1); }
```

上述程序的运行结果如下:

```
填入的原序列:
9  31  26  19  1  13  2  11  27  16  5  21
依次输出线性 Hash 表中的关键字:
<1><2><5><9><13><11><19><16><26><27><31><21>
查找序列各关键字在线性 Hash 表中的位置(表项序号):
4  11  9  7  1  5  2  6  10  8  3  12
```

2. 随机 Hash 表

当 Hash 表的长度 n 设计成 $n=2^k$ 时,还可以定义另外一种 Hash 表——随机 Hash 表。随机 Hash 表与线性 Hash 表的不同之处在于:一旦发生元素冲突,表项序号 i 的改变不是采用加 1 取模的方法,而是用某种伪随机数来改变。下面是随机 Hash 表的填入与取出的过程。

1) 随机 Hash 表的填入

将关键字 k 及有关信息填入随机 Hash 表的步骤如下。

(1) 计算关键字 k 的 Hash 码 $i_0=i(k)$,且令 $i=i_0$。

(2) 伪随机数序列初始化,令 $j=1$(将取随机数指针指向伪随机数序列中的第一个随机数)。

(3) 检查表中第 i 项的内容:

若第 i 项为空,则将关键字 k 及有关信息填入该项;若第 i 项不空,则令 $i=\mathrm{mod}(i_0+\mathrm{RN}(j),n)$,并令 $j=j+1$(将取随机数指针指向下一个随机数),转(3)继续检查。其中 $\mathrm{RN}(j)$ 表示伪随机数序列 RN 中的第 j 个随机数。

伪随机数序列 RN 按下列方法产生。

```
R=1
FOR  j=1  TO  n  DO
  { R=mod(5 * R,4n)
    RN(j)=INT(R/4)
  }
```

例 3.4 将关键字序列(09,31,26,19,01,13,02,11,27,16,05,21)依次填入长度为 $n=2^4=16$ 的随机 Hash 表中。设 Hash 码为 $i=\mathrm{INT}(k/3)+1$。

伪随机数序列:1,6,15,12,13,2,11,8,9,14,7,4,5,10,3,0。

填入后的随机 Hash 表如表 3.3 所示。

表 3.3　随机 Hash 表的填入

表项序号	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
关键字 k	01	02	05	09	13	16	19	21	26	11	31					27
冲突次数	0	1	1	0	0	0	0	0	0	2	0					2

随机 Hash 表克服了线性 Hash 表的“堆聚”现象，但线性 Hash 表的第(2)个缺点依然存在。

2) 随机 Hash 表的取出

要在随机 Hash 表中取出关键字 k 的元素，其步骤如下。

(1) 计算关键字 k 的 Hash 码 $i_0=i(k)$，且令 $i=i_0$。

(2) 伪随机数序列初始化，令 $j=1$(将取随机数指针指向伪随机数序列中的第一个随机数)。

(3) 检查表中第 i 项的内容：

若第 i 项登记着关键字 k，则取出该项元素即可；若第 i 项为空，则表示在 Hash 表中没有该关键字的信息；若第 i 项不空，且登记的不是关键字 k，则令 $i=\mathrm{mod}(i_0+\mathrm{RN}(j),n)$，并令 $j=j+1$(将取随机数指针指向下一个随机数)，转(3)继续检查。其中 RN(j)表示伪随机数序列 RN 中的第 j 个随机数。

必须指出的是，随机 Hash 表的填入与取出所采用的伪随机数序列应是同一个序列。与线性 Hash 表一样，只要随机 Hash 表尚未填满，其填入与取出的过程均能很好地终止。

在随机 Hash 表中，查找一个关键字的平均查找次数为

$$E \approx -\frac{1}{\alpha}\ln(1-\alpha)$$

其中 α 为 Hash 表的填满率。显然，当随机 Hash 表被填满($\alpha=1$)后，其填入与取出过程均不能正常终止。因此，与线性 Hash 表一样，随机 Hash 表一般也不能填满。

下面是对随机 Hash 表类的 C++ 描述：

```
//Rnd_hash.h
    #include<iostream>
    using namespace std;
    //随机 Hash 表结点类型
    template<class T>
    struct Hnode
    { int flag;                                    //标志表项空与非空
      T  key;                                      //关键字
    };
    template<class T>                              //模板声明,数据元素虚拟类型为 T
    class  Rnd_hash                                //随机 Hash 表类
    { private:                                     //数据成员
        int NN;                                    //随机 Hash 表长度
        Hnode<T> * RH;                             //随机 Hash 表存储空间首地址
        int * RND;                                 //随机数序列存储空间首地址
      public:                                      //成员函数
        Rnd_hash(){ NN=0; return;}
        Rnd_hash(int);                             //建立随机 Hash 表存储空间
```

```
    void prt_R_hash();                          //顺序输出随机 Hash 表中的元素
    int flag_R_hash();                          //检测随机 Hash 表中的空项个数
    void ins_R_hash(int ( * f)(T), T);          //在随机 Hash 表中填入新元素
    int sch_R_hash(int ( * f)(T), T);           //在随机 Hash 表中查找元素
};

//建立随机 Hash 表存储空间
template<class T>
Rnd_hash<T>::Rnd_hash(int m)
{ int k, r;
  NN=m;                                         //存储空间容量
  RH=new Hnode<T>[NN];                          //动态申请随机 Hash 表存储空间
  for (k=0; k<NN; k++)                          //随机 Hash 表各项均为空
      RH[k].flag=0;
  RND=new int[NN];                              //动态申请随机数序列存储空间
  r=1;
  for (k=0; k<NN; k++)
  { r=5 * r;
    if (r>=4 * NN) r=r-4 * NN;
    RND[k]=r/4;
  }
  return;
}

//顺序输出随机 Hash 表中的元素
template<class T>
void Rnd_hash<T>::prt_R_hash()
{ int k;
  for (k=0; k<NN; k++)
    if (RH[k].flag==0)                          //该项为空
        cout<<"<NULL>"<<" ";
    else                                        //该项不空
        cout<<"<"<<RH[k].key<<">";
  cout<<endl;
  return;
}

//检测随机 Hash 表中空项个数
template<class T>
int Rnd_hash<T>::flag_R_hash()
{ int k, count=0;
  for (k=0; k<NN; k++)
    if (RH[k].flag==0) count=count+1;
  return(count);
}

//在随机 Hash 表中填入新元素
template<class T>
void Rnd_hash<T>::ins_R_hash(int ( * f)(T), T x)
{ int k, j=0;
  if (flag_R_hash()==0)                         //随机 Hash 表中已没有空项
  { cout<<"随机 Hash 表已满!"<<endl; return; }
  k=( * f)(x);                                  //计算 Hash 码
```

```
    while (RH[k-1].flag)                              //该项不空
    { k=k+RND[j];                                     //下一项
      if (k>NN) k=k-NN;
      j=j+1;                                          //下一个随机数
    }
    RH[k-1].key=x; RH[k-1].flag=1;                    //填入并置标志
    return;
  }

  //在随机 Hash 表中查找元素
  template<class T>
  int Rnd_hash<T>::sch_R_hash(int (*f)(T), T x)
  { int k, j=0;
    k=(*f)(x);                                        //计算 Hash 码
    while ((RH[k-1].flag)&&(RH[k-1].key!=x))
    { k=k+RND[j];                                     //下一项
      if (k>NN) k=k-NN;
      j=j+1;                                          //下一个随机数
    }
    if ((RH[k-1].flag)&&(RH[k-1].key==x))             //找到返回
        return(k);
    return(0);                                        //表中没有这个关键字,返回
  }
```

下面是例 3.4 的主函数。

```
//ch3_3.cpp
  #include "Rnd_hash.h"
  int hashf(int k);
  int main()
  { int a[12]={9,31,26,19,1,13,2,11,27,16,5,21};
    int k;
    Rnd_hash<int>h(16);                          //建立容量为 12 的随机 Hash 表空间
    cout<<"填入的原序列:"<<endl;
    for (k=0; k<12; k++)
        cout<<a[k]<<"  ";
    cout<<endl;
    for (k=0; k<12; k++)
        h.ins_R_hash(hashf, a[k]);
    cout<<"依次输出随机 Hash 表中的关键字:"<<endl;
    h.prt_R_hash();
    cout<<"查找序列各关键字在随机 Hash 表中的位置(表项序号):"<<endl;
    for (k=0; k<12; k++)
        cout<<h.sch_R_hash(hashf, a[k])<<"  ";
    cout<<endl;
    return 0;
  }
  int hashf(int k)                               //Hash 函数
  { return(k/3+1); }
```

上述程序的运行结果如下：

```
填入的原序列:
9  31  26  19  1  13  2  11  27  16  5  21
```

```
依次输出随机 Hash 表中的关键字:
<1><2><5><9><13><16><19><21><26><11><31><NULL><NULL><NULL><NULL><27>
查找序列各关键字在随机 Hash 表中的位置(表项序号):
4  11  9  7  1  5  2  10  16  6  3  8
```

3. 溢出 Hash 表

前面介绍的线性 Hash 表与随机 Hash 表均存在两个致命的缺点:一是在 Hash 表填入过程中不顾后效,从而在填入过程中产生冲突的机会在不断增多;二是当 Hash 表填满时,不能正常进行查找。造成这两个缺点的原因主要是冲突的元素仍然被填入 Hash 表的存储空间,而又无法预测被占用的空间以后是否有元素正常填入。如果将冲突的元素安排在另外的空间,而不占用 Hash 表本身的空间,就不会产生新的冲突,这就是溢出 Hash 表。

溢出 Hash 表包括 Hash 表和溢出表两部分。在 Hash 表的填入过程中,将冲突的元素顺序填入溢出表;而当查找过程中发现冲突时,就在溢出表中进行顺序查找。因此,溢出表是一个顺序查找表,但 Hash 表与溢出表的存储结构是相同的。

下面是溢出 Hash 表的填入与取出的过程。

1) 溢出 Hash 表的填入

将关键字 k 及有关信息填入溢出 Hash 表的步骤如下。

(1) 计算关键字 k 的 Hash 码 $i=i(k)$。

(2) 检查表中第 i 项的内容:

若第 i 项为空,则将关键字 k 及有关信息填入该项;若第 i 项不空,则将关键字 k 及有关信息依次填入溢出表中的自由项。

2) 溢出 Hash 表的取出

要在溢出 Hash 表中取出关键字 k 的元素,其步骤如下。

(1) 计算关键字 k 的 Hash 码 $i=i(k)$。

(2) 检查表中第 i 项的内容:

若第 i 项登记着关键字 k,则取出该项元素;若第 i 项为空,则表示在 Hash 表中没有该关键字的信息;若第 i 项不空,且登记的不是关键字 k,则转入在溢出表中进行顺序查找。

例 3.5 将关键字序列(09,31,26,19,01,13,02,11,27,16,05,21)依次填入长度为 $n=12$ 的溢出 Hash 表中。设 Hash 码为 $i=\mathrm{INT}(k/3)+1$。

填入后的溢出 Hash 表如表 3.4 所示。

表 3.4 溢出 Hash 表的填入

Hash 表	i	1	2	3	4	5	6	7	8	9	10	11	12
	k	01	05		09	13	16	19	21	26	27	31	

溢出表	i	1	2	3	4	…
	k	02	11			

在 Hash 码比较均匀而冲突不多的情况下,溢出表中实际上只有很少的几项,即使采用顺序查找,查找次数也不会很多,因此,其查找效率不会很低,具有一定的实用价值。溢出

Hash 表的缺点是：除 Hash 表本身外，还要增加一个溢出表，当 Hash 码不能遍历 Hash 表本身时，额外的溢出表空间也是一种浪费。

下面是对溢出 Hash 表类的 C++ 描述：

```
//Over_hash.h
    #include<iostream>
    using namespace std;
    //溢出 Hash 表结点类型
    template<class T>
    struct Hnode
    { int flag;                                  //标志表项空与非空
      T  key;                                    //关键字
    };
    template<class T>                            //模板声明,数据元素虚拟类型为 T
    class  Over_hash                             //溢出 Hash 表类
    { private:                                   //数据成员
        int NN;                                  //溢出 Hash 表长度
        int MM;                                  //溢出表长度
        Hnode<T> * H;                            //Hash 表存储空间首地址
        Hnode<T> * R;                            //溢出表存储空间首地址
      public:                                    //成员函数
        Over_hash(){ NN=0; MM=0; return;}
        Over_hash(int, int);                     //建立溢出 Hash 表存储空间
        void prt_O_hash();                       //顺序输出溢出 Hash 表中的元素
        int flag_O_hash();                       //检测溢出表中的空项个数
        void ins_O_hash(int ( * f)(T), T);       //在溢出 Hash 表中填入新元素
        int sch_O_hash(int ( * f)(T), T);        //在溢出 Hash 表中查找元素
        void del_O_hash(int ( * f)(T), T);       //在溢出 Hash 表中删除一个元素
    };

    //建立溢出 Hash 表存储空间
    template<class T>
    Over_hash<T>::Over_hash(int m, int n)
    { int k;
      NN=m;                                      //Hash 表存储空间容量
      MM=n;                                      //溢出表存储空间容量
      H=new Hnode<T>[NN];                        //动态申请 Hash 表存储空间
      R=new Hnode<T>[MM];                        //动态申请溢出表存储空间
      for (k=0; k<NN; k++)                       //Hash 表各项均为空
          H[k].flag=0;
      for (k=0; k<MM; k++)                       //溢出表各项均为空
          R[k].flag=0;
      return;
    }

    //顺序输出溢出 Hash 表中的元素
    template<class T>
    void Over_hash<T>::prt_O_hash()
    { int k;
      cout<<"Hash 表:"<<endl;
```

```
  for (k=0; k<NN; k++)
    if (H[k].flag==0)
        cout<<"<NULL>"<<" ";
    else
        cout<<"<"<<H[k].key<<">";
  cout<<endl;
  cout<<"溢出表:"<<endl;
  for (k=0; k<MM; k++)
    if (R[k].flag==0)
        cout<<"<NULL>"<<" ";
    else
        cout<<"<"<<R[k].key<<">";
  cout<<endl;
  return;
}

//检测溢出表中空项个数
template<class T>
int Over_hash<T>::flag_O_hash()
{ int k, count=0;
  for (k=0; k<MM; k++)
    if (R[k].flag==0) count=count+1;
  return(count);
}

//在溢出 Hash 表中填入新元素
template<class T>
void Over_hash<T>::ins_O_hash(int (*f)(T), T x)
{ int k;
  k=(*f)(x);                                    //计算 Hash 码
  if (H[k-1].flag==0)                           //填入 Hash 表
  { H[k-1].flag=1; H[k-1].key=x; }
  else                                          //填入溢出表
  { k=1;
    while ((k<=MM)&&(R[k-1].flag))  k=k+1;
    if (k>MM) cout<<"溢出表已满!"<<endl;
    else
    { R[k-1].flag=1; R[k-1].key=x; }
  }
  return;
}

//在溢出 Hash 表中查找元素
template<class T>
int Over_hash<T>::sch_O_hash(int (*f)(T), T x)
{ int k, j=0;
  k=(*f)(x);                                    //计算 Hash 码
  if (H[k-1].flag==0)  return(0);               //Hash 表表项为空,找不到
  if (H[k-1].key==x)  return(k);                //在 Hash 表中找到
  k=1;                                          //到溢出表中去找
  while ((k<=MM)&&(R[k-1].flag)&&(R[k-1].key!=x)) k=k+1;
  if ((R[k-1].flag)&&(R[k-1].key==x)) return(-k);    //在溢出表中找到
  return(0);                                    //溢出表中也没有这个关键字,返回
```

```
}

//在溢出 Hash 表中删除一个元素
template<class T>
void Over_hash<T>::del_O_hash(int ( * f)(T), T x)
{ int k, j, kk;
  k=( * f)(x);                          //计算 Hash 码
  if (H[k-1].flag)
     { if (H[k-1].key==x)               //在 Hash 表中找到要删除的元素
         { j=1; kk=0;                   //再到溢出表中去找 Hash 码与之相同的关键字
           while ((j<=MM)&&(R[j-1].flag))
              { if (( * f)(R[j-1].key)==k) kk=j;
                j=j+1;
              }
           if (kk!=0)                   //溢出表中后面的元素依次前移
              { H[k-1].key=R[kk-1].key;
                while ((kk+1<=MM)&&(R[kk].flag))
                  { R[kk-1].key=R[kk].key; kk=kk+1; }
                R[kk-1].flag=0;         //最后一项置空
              }
           else H[k-1].flag=0;
         }
       else                             //在 Hash 表中没有找到,到溢出表中去找
         { j=1;
           while ((j<=MM)&&(R[j-1].flag)&&(R[j-1].key!=x))
                j=j+1;
           if (R[j-1].key==x)           //溢出表中后面的元素依次前移
              { while ((j+1<=MM)&&(R[j].flag))
                   { R[j-1].key=R[j].key; j=j+1; }
                R[j-1].flag=0;          //最后一项置空
              }
           else cout<<"表中没有这个关键字!"<<endl;
         }
     }
  else  cout<<"表中没有这个关键字!"<<endl;
  return;
}
```

下面是例 3.5 的主函数：

```
//ch3_4.cpp
    #include "Over_hash.h"
    int hashf(int k);
    int main()
    { int a[12]={9,31,26,19,1,13,2,11,27,16,5,21};
      int k;
      Over_hash<int>h(12, 10);          //Hash 容量为 12,溢出表容量为 10
      cout<<"填入的原序列:"<<endl;
      for (k=0; k<12; k++)
          cout<<a[k]<<"  ";
      cout<<endl;
      for (k=0; k<12; k++)
          h.ins_O_hash(hashf, a[k]);
```