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到数据处理的效率。

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(即查找成功)；若线性表中所有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(即查找失败)。

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。但如果被查的元素是线性表中的最后一个元素，或者被查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(即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，都只能用顺序查找。

(2) 即使是有序线性表，如果采用链式存储结构，也只能用顺序查找。

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。本节下面各小节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(即从小到大，但允许相邻元素值相等)的。

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(即中间项以前的部分)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（即中间项以后的部分）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）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，其中的操作有有序表的初始化、查找、插入、删除、输出以及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;
```

但由于在 Visual C++ 6.0 中不支持将运算符重载函数作为友元函数,但在 Visual C++ 6.0 所提供的带后缀 .h 的头文件中支持,因此,编者在调试该程序时将前两行修改了一下。

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,然后再输出该新有序表中的元素。

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 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示意图。

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) 首先查找索引表，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) 然后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，平均情况下大约需要查找 $\log_2(m+1)+n/(2m)$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。

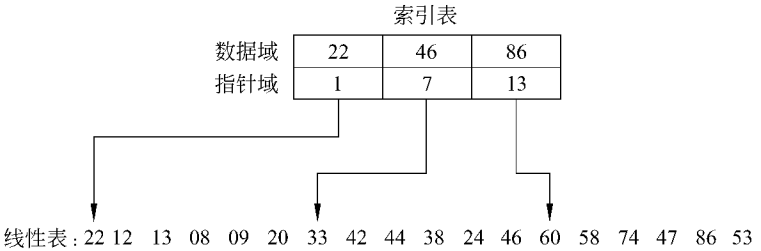

图 3.1　分块有序表例

3.2　哈希表技术

在 3.1 节中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表技术是另一类重要的查找技术。哈希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。本章主要介绍哈希表的基本概念以及工程上常用的几种哈希表。在具体介绍哈希表之前，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 的元素填入到直接查找表，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到表的第 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，也只需做以下两步：

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 个字节，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，长度为 n 的表来说，总是认为 $1\leqslant i\leqslant n$。

2. 哈希表

哈希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其映像函数值也不同。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，它们的映像函数值(即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 $i=\text{INT}(k/3)+1$ 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，引入哈希表的概念。

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 $1\leqslant i\leqslant n$，则称此表为哈希表。其中函数 $i=i(k)$ 称为关键字 k 的哈希码。

由哈希表的这个定义可以看出，在哈希表中允许冲突存在，即在哈希表中允许几个不同的关键字其哈希码相同。如果在哈希表中没有冲突存在，则哈希表就成为直接查找表。

哈希表技术的关键是要处理好表中元素的冲突问题，它主要包括以下两方面的工作：

(1) 构造合适的哈希码，以便尽量减少表中元素冲突的次数。即哈希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. 哈希码的构造

哈希表技术的主要目标是提高查找效率，即要缩短查表的时间。而在查找关键字为 k 的元素时，计算哈希码 $i(k)$ 的工作量也是影响查找效率的一个因素。如果哈希码的计算比较复杂，那么，尽管哈希码冲突的机会很少，也会降低查找的效率。因此，在实际设计哈希

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哈希表中，即哈希码的均匀性要好，以便减少冲突发生的机会。

(2) 哈希码的计算要尽量简单。

以上两个方面在实际应用中往往是矛盾的。为了保证哈希码的均匀性比较好，其哈希码的计算就必然要复杂；反之，如果哈希码的计算比较简单，则其均匀性就比较差。因此，在实际设计哈希码时，要根据具体情况，选择一个比较合理的方案。例如，当哈希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，而哈希码的计算稍微复杂一些是无关紧要的；当哈希表放在计算机内存中时，则应使哈希码的计算尽量简单，此时虽然哈希码冲突机会稍多一些，但在总体上考虑还是划算的。

由于哈希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哈希码的方案。下面仅介绍一些比较简单的哈希码的构造方法。

(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)作为该关键字的哈希码。

在这种方法中，对数字串所截取的位数取决于哈希表的长度 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哈希码的基础，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哈希码，即取哈希码为

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

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

本方法构造的哈希码，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 取其余数作为哈希码，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(即进行截段)作为哈希码。其中常数 ϕ 一般取 0.618033988747、0.6125423371 或 0.6161616161。

构造哈希码的方法有很多，用户还可以根据具体情况自行设计。为了能得到均匀性较好的哈希码，一般需要做多次试验，检查哈希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，适当修改构造哈希码的方法，然后再进行试验，直到哈希码的均

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，哈希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哈希表。本节介绍几种常用的哈希表，在各种哈希表的查找（即填入与取出）方法中，体现了各种不同的处理冲突的方法。

1. 线性哈希表

线性哈希表是一种最简单的哈希表。

设线性哈希表的长度为 n，对线性哈希表的查找过程如下。

(1) 线性哈希表的填入

将关键字 k 及有关信息填入线性哈希表的步骤如下：

① 计算关键字 k 的哈希码 $i=i(k)$。

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，只要哈希表尚未填满，最终总可以找到一个空项，将关键字 k 及有关信息填入到哈希表中。

(2) 线性哈希表的取出

要在线性哈希表中取出关键字 k 的元素，其步骤如下：

① 计算关键字 k 的哈希码 $i=i(k)$。

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，则表示在哈希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，只要哈希表尚未填满，这个过程能够很好地终止，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，以找不到告终。

线性哈希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 $n=12$ 的线性哈希表中。设哈希码为 $i=\mathrm{INT}(k/3)+1$。

填入后的线性哈希表如表 3.2 所示。

表 3.2 线性哈希表的填入

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

线性哈希表的优点是简单，但它有以下两个主要缺点：

(1) 在线性哈希表填入的过程中，当发生冲突时，首先考虑的是下一项，因此，当哈希码的冲突较多时，在线性哈希表中会存在“堆聚”现象，即许多关键字被连续登记在一起，从而