设计一个闭散列表类，并在其中增加一个自动重散列的功能。这个重散列包括以下功能：（a）当数组中的活动元素超过数组元素的一半，即负载因子达到 0.5，自动扩大数组空间。（b）当数组中标志为“被删除的”元素个数超过用户指定的值时（由构造函数设定），重新散列。

【解】要完成这两项功能，需要在闭散列表类中增加三个数据成员：散列表中的有效元素个数（load）、散列表中被删除的元素个数（dead）以及被删除元素个数的上限(deadline)。要实现功能（a），需要修改insert函数。在添加元素时检查load，如果load已经达到数组规模的一半，则自动扩大空间，否则load加1。实现功能（b）需要修改remove函数。每次成功删除元素后都要将dead加1。如果此时dead大于用户给定的上限deadline，自动重散列。增加了这些因素后的散列表类的定义见代码清单9-24。

代码清单 9-24 散列表类的定义

```
template <class type>
class closeHash{
private:
    struct node {
        type data;
        int state; ;     // 0：空单元  1：有效元素  2：已删除
        node() { state = 0; }
    };

    node *array;
    int size;
    int load;              //散列表中的元素个数
    int dead;              //已被删除的元素个数
    int deadline;          //被删除元素个数的上限
    int (*key) ( const int &x ); //取出元素 x 的键，并转换成整型
    static int defaultKey( const int &k ) { return k ; }//缺省的转换函数

public:
  closeHash(int length = MAXN, int die = MAXN / 2,
    int (*f)(const type &x) = defaultKey):load(0), deadline(die), dead(0)
    {  size = nextPrime(length);  // 选择大于等于 length 的最小素数作为表长
       array = new node[size];
       key = f;
    }

    ~closeHash() {delete [] array; }
    bool find( const type &x ) const;
    bool insert( const type &x );
    bool remove( const type &x );

private:
    void doublespace();
```

```
33.     void rehash();
34.     bool isPrime(int n) ;
35.     int nextPrime(int n) ;
36. };
```

代码清单 9-24 中的散列表增加了如上所述的三个数据成员，还增加了四个私有的成员函数。doubleSpace 用于扩展数组空间。reHash 将被删除的元素真正删除。isPrime 和 nextPrime 用于确定数组规模时选择一个素数，这两个函数的定义与程序设计题 2 中的定义完全一样。

insert 函数的实现见代码清单 9-25。与代码清单 9-4 中的 insert 函数的区别在于增加了 9 到 11 行。当添加一个元素时，有效元素数加 1。然后检查有效元素个数是否超过了表长的一半，如果是的，则扩展空间。

代码清单 9-25 insert 函数的实现

```
1.  template <class type>
2.  bool closeHash<type>::insert( const type &x )
3.  {   int initPos, pos;
4.      initPos = pos = key(x) % size;
5.      do{ if ( array[pos].state != 1 ) {
6.              array[pos].data = x;
7.              array[pos].state = 1;
8.              ++load;             // 有效元素数加 1
9.              if (load*2 >= size ) {   //负载因子达到 0.5，扩大空间
10.         doublespace();
11.       }
12.             return true;
13.         }
14.         if ( array[pos].state == 1 && array[pos].data == x ) return true;
15.         pos = ( pos + 1 ) % size;
16.     } while ( pos != initPos );
17.
18.     return false;
19. }
```

remove 函数的实现见代码清单 9-26。这个函数和代码清单 9-5 的唯一区别是增加了第 9 到 11 行。当成功删除一个元素后，被删元素个数加 1。然后检查被删元素个数是否超过了指定值。如果超过了，则自动调用 reHash 函数。

代码清单 9-26 remove 函数的实现

```
1.  template <class type>
2.  bool closeHash<type>::remove( const type &x )
3.  {   int initPos, pos;
4.
```

```
5.      initPos = pos = key(x) % size;
6.      do{ if (array[pos].state == 0) return false;
7.          if (array[pos].state == 1 && array[pos].data == x){
8.              array[pos].state = 2;
9.              --load;
10.             ++dead;
11.             if (dead > deadline) rehash(); //超过指定的值，重新散列
12.         return true;
13.         }
14.         pos = ( pos + 1 ) % size;
15.     } while ( pos != initPos );
16.
17.     return false;
18. }
```

查找函数的实现与代码清单 9-6 中完全一样。

下面看一看两个私有的成员函数。doubleSpace 函数自动将数组空间扩大一倍。但要注意扩大空间时不是简单地将 size 乘 2，而是要选择一个大于等于 2 倍大小的最小素数。另一个要注意的是：当数组空间扩大时，被删元素个数的上限也要修改，我们简单地将它乘以 2。doubleSpace 函数的实现见代码清单 9-27。

代码清单 9-27 doubleSpace 函数的实现

```
1.  template <class type>
2.  void closeHash<type>::doublespace()
3.  {   node *tmp = array;
4.      int oldsize = size;
5.
6.      size = nextPrime(oldsize * 2);
7.      deadline *= 2;
8.      load = 0;
9.      array = new node[size];
10.     for (int i=0; i<oldsize; i++)
11.         if (tmp[i].state == 1) insert(tmp[i].data);
12.     delete[] tmp;
13.  }
```

最后一个是 reHash 函数，它的实现见代码清单 9-28。

代码清单 9-28 reHash 函数的实现

```
14. template <class type>
15. void closeHash<type>::rehash()   //整理散列表
16. {   dead = 0;       //死亡计数器清 0
17.     node *tmp = array;
```

```
18.         array = new node[size];
19.         for ( int i = 0; i < size; ++i ) {
20.             if ( tmp[i].state == 1 ) insert(tmp[i].data);
21.         }
22.         delete[] tmp;
23.   }
```