第7章　跳表和散列

对于一个有n 个元素的有序数组，用折半搜索法进行搜索所需要的时间为 O(log$n$)，而对一个有序链表进行搜索所需要的时间为 O(n)。我们可以通过对有序链表上的全部或部分节点增加额外的指针，来提高搜索性能。在搜索时，可以通过这些指针来跳过链表中若干个节点，因此没有必要从左到右搜索链表中的所有节点。

增加了向前指针的链表叫作跳表。跳表不仅能提高搜索性能，同时也可以提高插入和删除操作的性能。它采用随机技术决定链表中哪些节点应增加向前指针以及在该节点中应增加多少个指针。采用这种随机技术，跳表中的搜索、插入、删除操作的时间均为 O(log$n$)，然而，最坏情况下时间复杂性却变成 $\Theta(n)$。相比之下，在一个有序数组或链表中进行插入 /删除操作的时间为O(n)，最坏情况下为 $\Theta(n)$。

散列法是用来搜索、插入、删除记录的另一种随机方法。与跳表相比，它的插入 /删除操作时间提高到 $\Theta(1)$，但最坏情况下仍为 $\Theta(n)$。尽管如此，在经常将所有元素按序输出或按序号搜索元素时(如寻找第10个最小元素)，跳表的执行效率将优于散列。

本章所给出的应用是关于文本压缩和解压缩的应用，该应用用散列实现。所设计的程序基于当前流行的Liv_Zempel_Welch算法（简称LZW算法）。

7.1　字典

字典（dictionary）是一些元素的集合。每个元素有一个称作 key 的域，不同元素的key 各不相同。有关字典的操作有：

- 插入具有给定关键字值的元素。
- 在字典中寻找具有给定关键字值的元素。
- 删除具有给定关键字值的元素。

抽象数据类型*Dictionary*的描述见ADT 7-1。若仅按照一个字典元素本身的关键字来访问该元素，则称为随机访问（random access）。而顺序访问（sequential access）是指按照关键字的递增顺序逐个访问字典中的元素。顺序访问需借助于 *Begin* (用来返回关键字最小的元素)和 *Next* (用来返回下一个元素)等操作来实现。对于本章中所实现的部分字典，既可以采用随机访问方式，也可以采用顺序访问方式。

ADT7-1　字典的抽象数据类型描述

抽象数据类型*Dictionary* {
实例
　具有不同关键字的元素集合
操作
　Create()：创建一个空字典
　Search(k, x)：搜索关键字为k 的元素，结果放入x；
　　如果没找到，则返回*false*，否则返回true
　Insert (x)：向字典中插入元素x

Delete (*k*, *x*)：删除关键字为*k* 的元素，并将其放入*x*

}

有重复元素的字典（dictionary with duplicates）与上面定义的字典相似，只是它允许多个元素有相同的关键字。在有重复元素的字典中，在进行搜索和删除时需要一个规则来消除岐义。也就是说，如果要搜索(或删除)关键字为*k* 的元素，那么在所有关键字为*k* 值的元素中应该返回(或删除)哪一个呢? 在有些字典应用中，可能需要：删除在某个时间以后插入的所有元素。

例7-1 一个班中注册学习数据结构课程的学生构成了一个字典。当有一个新学生注册时，就要在字典中插入与该学生相关的元素 (记录)。当有人要放弃这门课程时，则删除他的记录。在上课过程中，老师可以查询字典以得到与某特定学生相关的记录或修改记录 (例如，加入或修改考试成绩)。学生的姓名域可作为关键字。

例7-2 在编译器中定义用户描述符的符号表（symbol table）就是一个有重复元素的字典。当定义一个描述符时，要建立一个记录并插入到符号表中。记录中包括作为关键字的描述符以及其他信息，如描述符类型(int，float 等) 和(相关的)存储其值的内存地址。因为同样的描述符名可以定义多次(在不同的程序块中)，所以符号表中必然存在有多个记录具有相同的关键字，搜索结果应是最新插入的元素。只有在程序块的结尾才能进行删除，所有在开始插入的元素最终都要被删除掉。

7.2 线性表描述

字典可以保存在线性序列(e_1 , e_2 , …) 中，其中e_i 是字典中的元素，其关键字从左到右依次增大。为了适应这种描述方式，可以定义两个类 SortedList 和SortedChain。前者采用公式化描述的线性表，如LinearList类(见程序3-1)，而后者则采用链表描述，如Chain类(见程序3-8)。

练习1要求设计SortedlList类。应注意到在SortedList 中可以采用折半搜索法搜索元素，因此对有*n*个元素的字典进行搜索的时间为O(log*n*)。进行插入时，必须确信该字典中没有相同关键字的元素 ,这要通过搜索来实现。然后进行插入，此时要为新元素腾出空间而移动表中 O(*n*)个元素，故插入操作的时间是O(*n*)。删除则首先要找到欲删除的元素，然后再进行删除。在进行搜索之后，还要移动O(*n*)个元素以填补所删除元素的空间，因此删除的时间复杂性为O(*n*)。

程序7-1、程序7-2和程序7-3给出了类SortedChain的定义。E表示链表元素的数据类型， K是链表中排序所用到的关键字。类SortedChainNode与类ChainNode(见程序3-8)一样，只有两个私有成员data和link。类SortedChain是类SortedChainNode的友元。

程序7-1 **类**SortedChain

```
template<class E ,class K>
class SortedChain{
  public:
     SortedChain() {first =0;}
     ~SortedChain();
     bool  IsEmpty() const {return first ==0;}
     int Length() const;
     bool  Search(const  K& k , E& e) const;
     SortedChain<E ,K>&  Delete(const  K& k , E& e);
```

```
    SortedChain<E ,K>&  Insert(const  E& e);
    SortedChain<E ,K>&  DistinctInsert(const  E& e);
  private:
    SortedChainNode<E ,K> *first;
};
```

程序7-2　SortedChain类的成员函数search和delete

```
template<class E, class K>
bool SortedChain<E,K>::Search(const K& k, E& e) const
{// 搜索与k匹配的元素，结果放入e
 // 如果没有匹配的元素，则返回false

  SortedChainNode<E,K> *p = first;

  // 搜索与k相匹配的元素
  for (; p && p->data < k;
      p = p->link);

  // 验证是否与k匹配
  if (p && p->data == k) // 与k相匹配
    {e = p->data; return true;}
  return false; // 不存在相匹配的元素
}

template<class E, class K>
SortedChain<E,K>& SortedChain<E,K>
                ::Delete(const K& k, E& e)
{// 删除与k相匹配的元素
 // 并将被删除的元素放入e
 // 如果不存在匹配元素，则引发异常BadInput

  SortedChainNode<E,K> *p = first,
                       *tp = 0; //跟踪p

  // 搜索与k相匹配的元素
  for (; p && p->data < k; tp = p; p = p->link;
     }

  // 验证是否与k匹配
  if (p && p->data == k) {// 找到一个相匹配的元素
        e = p->data;    // 保存data域

        // 从链表中删除p所指向的元素
        if (tp) tp->link = p->link;
        else first = p->link;  // p是链首节点

        delete p;
```

```
        return *this;}
    throw BadInput();  // 不存在相匹配的元素
}
```

程序7-3 有序链表中的插入操作

```
template<class E, class K>
SortedChain<E,K>& SortedChain<E,K>::DistinctInsert(const E& e)
{// 如果表中不存在关键值与e相同的元素，则插入e
  //否则引发异常BadInput

  SortedChainNode<E,K> *p = first, *tp = 0; // 跟踪p

  // 移动 tp 以便把 e 插入到 tp之后
  for (; p && p->data < e; tp = p, p = p->link);

  // 检查关键值是否重复
  if (p && p->data == e) throw BadInput();

  // 若没有出现重复关键值，则产生一个关键值为e的新节点
  SortedChainNode<E,K> *q = new SortedChainNode<E,K>;
  q->data = e;

  // 将新节点插入到tp之后
  q->link = p;
  if (tp) tp->link = q;
  else first = q;

  return *this;
}
```

类SortedChain提供了两种插入操作，DistinctInsert操作保证链中所有元素有不同的关键字，而Insert允许有相同的关键字。虽然字典应用要求DistinctInsert操作，但有序链表的其他应用可能会用到Insert操作。

析构函数、Length、Output 和<<的重载的代码与类Chain (见程序3-8)一样，在这里不再重复。可以通过删除DistinctInsert 中搜索重复元素的那段代码 (即程序7-3中的第一条*if* 语句)来得到Insert。从代码中可以看出，在有 n个节点的链表中，搜索、插入、删除的时间均为$O(n)$。

在类SortedChain中，需定义有关数据类型 E的操作符<<，==，!=和<，至于用户自定义的数据类型，可以采用 C++的操作符重载机制来定义各种操作符。许多应用要求只根据E的关键字来实现这些操作。实现重载操作的一个简单方法是重载类型转换符以实现类型E到类型K的转换(K的类型为key)。例如，当K是long时，可以在E的类定义中加入如下语句：

```
operator long( ) const {return key；}
```

可以扩充所有的SortedList和SortedChain以提供高效的顺序访问。在这两种情况下 Begin和

Next返回一个元素所需要的时间均为Θ(1)。

练习

1. 采用公式化描述方法定义C++类SortedList。要求提供与SortedChain中同样的成员函数。编写所有函数的代码，并用合适的数据测试代码。

2. 扩充SortedChain类，使其包括顺序访问函数 Begin和Next。Begin用来返回字典中第一个元素， Next 用来返回字典中下一个元素（采用从小到大的排列顺序）。在没有第一个或下一个元素的情况下，这两个函数均返回 0。两个函数的复杂度均应为 Θ(1)。测试代码的正确性。

3. 修改类SortedChain，使用一条具有头节点和尾节点的链表。尾节点用来存放比其他元素值都大的元素。采用尾节点可以简化代码。

7.3 跳表描述

7.3.1 理想情况

在一个使用有序链表描述的具有 n 个元素的字典中进行搜索，至多需进行 n 次比较。如果在链中部节点加一个指针，则比较次数可以减少到 $n/2+1$。搜索时，首先将欲搜索元素与中间元素进行比较。如果欲搜索的元素较小，则仅需搜索链表的左半部分，否则，只要在链表右半部分进行比较即可。

例7-3 图7-1a 的有序链表中有七个元素。该链表有一个头节点和一个尾节点。节点中的数是该节点的值。对该链表的搜索可能要进行七次比较。可以采用图 7-1b 中的办法，把最坏情况下的比较次数减少为四次。搜索一个元素时，首先将它与中间元素进行比较，然后根据得到的结果，或者与链的左半部比较或者与右半部比较。例如如果要找值为 26的元素，只需查找40左边的元素。如果要查找值为75的元素，那么只对40以后的元素进行搜索即可。

也可以象图7-1c 中一样，分别在链表左半部分和右半部分的中间节点再增加一个指针，以便进一步减少最坏情况下的搜索比较次数。在该图中有三条链。 0级链就是图7-1a 中的初始链，包括了所有七个元素。一级链包括第二，四，六个元素，而 2级链只包括第四个元素。为了寻找值为30的元素，首先与中间元素比较。在 2级链中寻找元素所需要的时间为 Θ(1)。由于30＞40，因此要搜索该链左半部分的中间元素。采用 1级链进行搜索所花费的时间为 Θ(1)。又因为30＞24，故需在0级链中继续进行查找，把该元素与链中下一个元素进行比较。

考察另一个例子。设要查找的元素值为 77。首先与40比较， 70＞40，则在1级链中与75比较。由于77＞75，因此在0级链中与75后面的80比较。这时可以得知77不在此字典中。采用图7-1c 中的3级链结构，对所有的搜索至多需三次比较。 3级链结构允许在有序链表中进行折半搜索。

通常0级链包括n个元素， 1级链包括n/2个元素， 2级链包括$n/4$个元素，而每2^i个元素有一个i级链指针。当且仅当一个元素在0～i 级链上，但不在i+1级（若该链存在）链上时，我们就说该元素是i 级链元素。在图7-1c 中，40是2级链上唯一的元素而75是1级链元素。20、30、60、80是0级链元素。

图7-1c 所示的结构为跳表（skip list）。在该结构中有一组有层次的链。 0级链是包含所有

元素的有序链表，1级链是0级链的一个子集。i 级链所包含的元素是i-1级链的子集。图7-1c 中，i 级链上所有的元素均在i-1级链上。

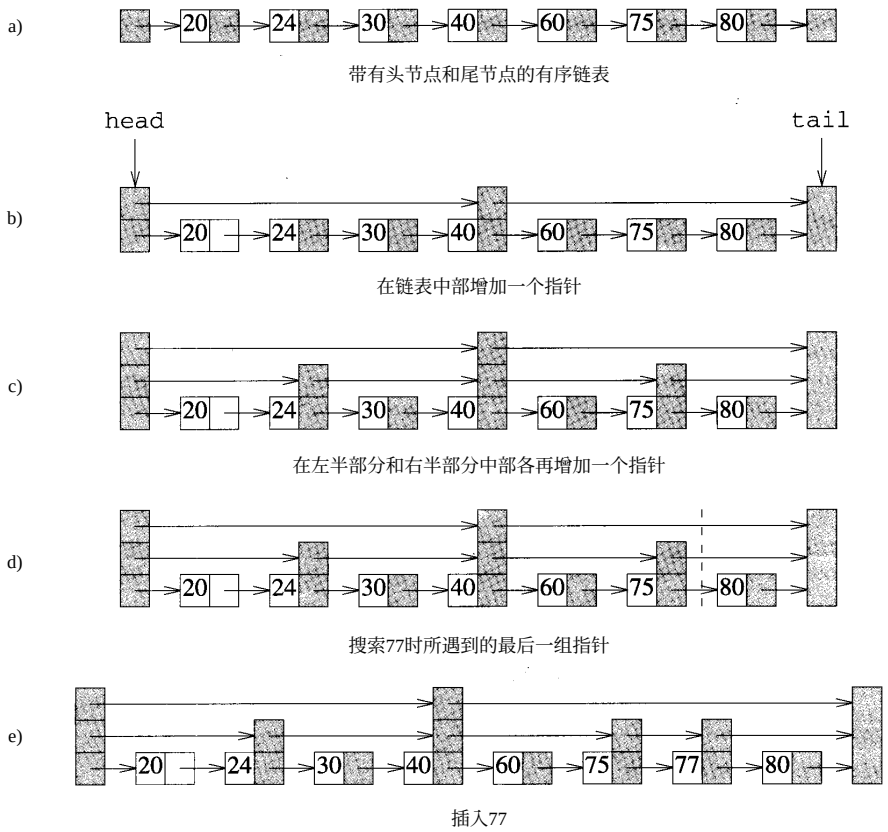


图7-1 有序链表的快速搜索

7.3.2 插入和删除

在进行插入和删除时，要想保持图7-1c 的跳表结构，必须耗时O(n)。注意到在这种结构中，有$n/2^i$ 个元素为i 级链元素，所以在进行插入时应尽量逼近这种结构。在进行插入时，新元素属于i 级链的概率为$1/2^i$。在确定新元素的级时，应考虑各种可能的情况。因此，把新元素作为i 级链元素的可能性为p^i 。图7-1c 中p=0.5。对于一般的p，链的级数为$\lfloor(\log_{1/p} n)\rfloor+1$。在这种情况下，每$p$ 个i -1级链中有一个在i 级链中。

假设要插入的元素为77，首先要通过搜索以确定链中没有此元素。在搜索中，最后一个 2级链指针存储在40的指针域中，而最后一个1级链指针存储在75的指针域中。在图7-1d 中，这几条指针用虚线标出。新元素插在 75和80之间，如图7-1d 中的虚线所示。插入时，要为新元素分配一个级，分配过程由随机数产生器完成随机数产生器将在后面介绍。

若新元素为i 级链元素，则仅影响由虚线断开的0～i 级链指针。图7-1e 给出新插入元素77作为1级链元素时链表的结构。

对于删除操作，我们无法控制其结构。要删除图7-7e 中的元素77，首先要找到77。最后所遇到的链指针是节点40中的2级链指针、节点75中的1级链指针和0级链指针。在这些链指针中，因为77为1级链元素，所以只需改变0级和1级链指针即可。当这些指针变成指向 77后面的元素

时，就得到图7-1d 的结构。

7.3.3 级的分配

在级基本的分配过程中，可以观察到，在一般跳表结构中，i-1级链中的元素属于i 级链的概率为p。假设有一随机数产生器所产生的数在0到RAND_MAX间。则下一次所产生的随机数小于等于CutOff =p*RAND_MAX的概率为p。因此，若下一随机数小于等于CutOff，则新元素应在1级链上。现在继续确定新元素是否在2级链上，这由下一个随机数来决定。若新的随机数小于等于CutOff，则该元素也属于2级链。重复这个过程，直到得到一随机数大于CutOff为止。

故可以用下面的代码为要插入的元素分配级。

```
int lev = 0;
while (rand( )<=CutOff ) lev++ ;
```

这种方法潜在的缺点是可能为某些元素分配特别大的级，从而导致一些元素的级远远超过 $\log_{1/p} N$，其中N为字典中预期的最大数目。为避免这种情况，可以设定一个上限 lev。在有N个元素的跳表中，级MaxLevel的最大值为

$$\lceil \log_{1/p} N \rceil - 1 \tag{7-1}$$

可以采用此值作为上限。

另一个缺点是即使采用上面所给出的上限，但还可能存在下面的情况，如在插入一个新元素前有三条链，而在插入之后就有了 10条链。这时，新插入元素的为9级，尽管在前面插入中没有出现3到8级的元素。也就是说，在此插入前并未插入 3，4，…，8级元素。既然这些空级没有直接的好处，那么可以把新元素的级调整为3。

例7-4　用跳表表示一个最多有1024个元素的字典。设p=0.5，则MaxLevel为$\log_2 1024 - 1 = 9$。若随机数产生器的RAND_MAX = 2^{32} - 1,则CutOff = 2^{31}-1。新产生的随机数小于等于CutOff的概率为0.5。

假定我们首先从一个空字典开始，该字典用具有头节点和尾节点的跳表结构描述，头节点有10个指针，每个指针对应一条链表，且从头节点指向尾节点。

当插入第一个元素时，为其在0到9(MaxLevel)之间分配一个级。若所分配的级为9，则在插入第一个元素时，要修改九个指针。另一方面，由于没有 0，1，…，8级元素，故可以把该元素的级改为0，这样只需修改一条指针即可。

另一种级的分配方法是把随机数产生器的值分为几段。第一段包括范围的 1 - 1/p，第二段包括1/p-1/p^2等等。若产生的随机数不在第i段中，则为此元素分配i-1级。

7.3.4 类SkipNode

跳表结构的头节点需有足够的指针域，以满足可能构造最大级数的需要，而尾节点不需要指针域。每个存有元素的节点都有一个 data域和（级数+1）个指针域。在程序7-4 中可以遇到所有类型的节点。指针域由数组 link表示，其中link[i]表示i 级链指针。构造函数为指针数组分配空间。对于一个lev级链元素，其size值应为lev+1。

程序7-4　**类**SkipNode

```
template<class E, class K>
class SkipNode {
```

```
    friend SkipList<E,K>;
    private:
      SkipNode(int size)
        {link = new SkipNode<E,K> *[size];}
      ~SkipNode() {delete [] link;}
      E data;
      SkipNode<E,K> **link;  // 一维指针数组
  };
```

7.3.5 类SkipList

程序7-5给出了类SkipList的定义。MaxE是字典的最大容量。虽然在给出的代码中允许元素数目超过MaxE，但若元素数目不超过MaxE，平均性能会更好一些。一个元素既在i-1级链上又在i级链上的概率为p，Large是一个比字典中任意一个数均大的值。尾节点的值为Large。0级链上的值（不包括头节点，因其没有值）从左到右按升序排列。

程序7-5 类SkipList

```
template<class E, class K>
class SkipList {
  public:
    SkipList(K Large, int MaxE = 10000, float p = 0.5);
    ~SkipList();
    bool Search(const K& k, E& e) const;
    SkipList<E,K>& Insert(const E& e);
    SkipList<E,K>& Delete(const K& k, E& e);
  private:
    int Level();
    SkipNode<E,K> *SaveSearch(const K& k);
    int MaxLevel;  // 所允许的最大级数
    int Levels;    // 当前非空链的个数
    int CutOff;    // 用于确定级号
    K TailKey;     // 一个很大的key值
    SkipNode<E,K> *head;  // 头节点指针
    SkipNode<E,K> *tail;  // 尾节点指针
    SkipNode<E,K> **last; // 指针数组
};
```

程序7-6给出了构造函数和析构函数。构造函数初始化CutOff、Levels（当前出现的最大级数）、MaxLevel、TailKey(所有元素值均小于此值)和用来为新元素分配级的随机数产生器。构造函数同时也为头节点和尾节点分配空间。在插入和删除操作之前进行搜索时，所遇到的每条链上的最后一个元素均被放入数组last 中。头节点中MaxLevel+1个用于指向各级链的指针被初始化为指向尾节点。构造函数的时间复杂性为Θ(MaxLevel)。

程序7-6 构造函数和析构函数

```
template<class E, class K>
SkipList<E,K>::SkipList(K Large, int MaxE, float p)
```

```
{// 构造函数
  CutOff = p * RAND_MAX;
  MaxLevel = ceil(log(MaxE) / log(1/p)) - 1;
  TailKey = Large;
  randomize( ); // 初始化随机发生器
  Levels = 0;  // 对级号进行初始化

  // 创建头节点、尾节点以及数组last
  head = new SkipNode<E,K> (MaxLevel+1);
  tail = new SkipNode<E,K> (0);
  last = new SkipNode<E,K> *[MaxLevel+1];
  tail->data = Large;

  // 将所有级均置空，即将head指向tail
  for (int i = 0; i <= MaxLevel; i++)
     head->link[i] = tail;
}

template<class E, class K>
SkipList<E,K>::~SkipList()
{// 删除所有节点以及数组last
  SkipNode<E,K> *next;

  // 通过删除0级链来删除所有节点
  while (head != tail) {
    next = head->link[0];
    delete head;
    head = next;
    }
  delete tail;

  delete [] last;
}
```

析构函数释放链表中用到的所有空间。其复杂性为 Θ(0级链的长度)。搜索、插入和删除函数均要求对E进行重载，以便在E的成员之间、E与K的成员之间进行比较。从K到E的赋值和转换也必须定义。当每个元素都有一个整数域 data和一个长整数域key，且元素的值由key给出时，可使用程序7-7所定义的重载。

程序7-7 跳表的操作符重载

```
class element {
  friend void main(void);
  public:
    operator long() const {return key;}
    element& operator =(long y)
    {key = y; return *this;}
  private:
    int data;
    long key;
};
```

SkipList类有两个搜索函数(见程序7-8)。当需要定位一个值为k 的元素时，可用共享成员函数Search。若找到要搜索的元素，则将该元素返回到 e 中，并返回true，否则返回false。Search从最高级链（Levels 级，仅含一个元素）开始查找，一直到0级链。在每一级链中尽可能地逼近要查找的元素。当从for循环退出时，正好处在欲寻找元素的左边。与0级链中的下一个元素进行比较，即可确定要找的元素是否在跳表中。

第二个搜索函数为私有成员函数SaveSearch，由插入和删除操作来调用。SaveSearch不仅包含了Search的功能，而且可把每一级中遇到的最后一个节点存放在数组last之中。

程序7-8 在跳表中搜索

```
template<class E, class K>
bool SkipList<E,K>::Search(const K& k, E& e) const
{// 搜索与k相匹配的元素，并将所找到的元素放入e
 // 如果不存在这样的元素，则返回false
  if (k >= TailKey) return false;

  // 调整指针p，使其恰好指向可能与k匹配的节点的前一个节点
  SkipNode<E,K> *p = head;
  for (int i = Levels; i >= 0; i--)    // 逐级向下
    while (p->link[i]->data < k)       // 在第i级链中搜索
      p = p->link[i];                  // 指针

  // 检查是否下一个节点拥有关键值 k
  e = p->link[0]->data;
  return (e == k);
}

template<class E, class K>
SkipNode<E,K> * SkipList<E,K>::SaveSearch(const K& k)
{// 搜索 k 并保存最终所得到的位置
 // 在每一级链中搜索
  // 调整指针p，使其恰好指向可能与k匹配的节点的前一个节点
  SkipNode<E,K> *p = head;
  for (int i = Levels; i >= 0; i--) {
    while (p->link[i]->data < k)
      p = p->link[i];
    last[i] = p;
    }
  return (p->link[0]);
}
```

程序7-9给出了在跳表中插入元素并为其分配级的代码。若欲插入元素的值不比TailKey小，或表中已有与该值相同的元素，Insert将引发BadInput异常。若没有足够的空间进行插入，则由new引发一个NoMem异常。当元素e 被成功插入后，Insert 将返回跳表。

程序7-9 向跳表中插入元素

```
template<class E, class K>
int SkipList<E,K>::Level()
```

```
{// 产生一个随机级号，该级号 <= MaxLevel
  int lev = 0;
  while (rand() <= CutOff)
    lev++;
  return (lev <= MaxLevel) ? lev : MaxLevel;
}

template<class E, class K>
SkipList<E,K>& SkipList<E,K>::Insert(const E& e)
{// 如果不存在重复，则插入 e
  K k = e; // 抽取关键值
  if (k >= TailKey) throw BadInput(); // 关键值太大

  // 检查是否重复
  SkipNode<E,K> *p = SaveSearch(k);
  if (p->data == e) throw BadInput(); // 重复

  // 不重复，为新节点确定级号
  int lev = Level(); // 新节点的级号
  // fix lev to be <= Levels + 1
  if (lev > Levels) {lev = ++Levels; last[lev] = head;}

  // 产生新节点，并将新节点插入 p之后
  SkipNode<E,K> *y = new SkipNode<E,K> (lev+1);
  y->data = e;
  for (int i = 0; i <= lev; i++) {
    // 插入到第 i级链
    y->link[i] = last[i]->link[i];
    last[i]->link[i] = y;
    }

  return *this;
}
```

程序7-10所给出的代码可用来删除一个值为k的元素，并把所删除的元素放入e中。若没有值为k的元素，则引发BadInput异常。while循环用来修改Levels的值，以找到一个至少包含一个元素的级（除非跳表为空）。若跳表为空，则Levels置为0。

程序7-10 从跳表中删除元素

```
template<class E, class K>
SkipList<E,K>& SkipList<E,K>::Delete(const K& k, E& e)
{// 删除与k相匹配的元素，并将所删除的元素放入 e
 // 如果不存在与 k匹配的元素，则引发异常 BadInput
  if (k >= TailKey) throw BadInput(); // 关键值太大

  // 检查是否存在与 k相匹配的元素
  SkipNode<E,K> *p = SaveSearch(k);
  if (p->data != k) throw BadInput(); // 不存在
```

```
    // 从跳表中删除节点
    for (int i = 0; i <= Levels && last[i]->link[i] == p; i++)
      last[i]->link[i] = p->link[i];

    // 修改级数
    while (Levels > 0 && head->link[Levels] == tail)
      Levels--;

    e = p->data;
    delete p;
    return *this;
  }
```

7.3.6 复杂性

当跳表中有n个元素时，搜索、插入、删除操作的复杂性均为O(n+MaxLevel)。在最坏情况下，可能只有一个MaxLevel级元素，且余下的所有元素均在0级链上。i＞0时，在i 级链上花费的时间为Θ(MaxLevel)，而在0级链上花费的时间为O(n)。尽管最坏情况下的性能较差，但跳表仍不失为一种有价值的数据描述方法。其每种操作 (搜索、插入、删除)的平均复杂性均为O(log$n$)，其证明超出了本书的范围。

至于空间复杂性，注意到最坏情况下所有元素都可能是 MaxLevel级，每个元素都需要MaxLevel+1个指针。因此，除了存储 n个元素(也就是n*sizeof(element))，还需要存储链指针(所需空间为O(n*MaxLevel))。不过，一般情况下，只有n*p个元素在1级链上，n*p^2 个元素在2级链上，n*p^i 在i 级链上。因此指针域的平均值(不包括头尾节点的指针)是$n\sum_i p^i = n/(1-p)$。因此虽然最坏情况下空间需求比较大，但平均的空间需求并不大。当 p=0.5时，平均空间需求(加上n个节点中的指针)大约是2n个指针的空间。

练习

4. 既然跳表中0级链是已排好序的，因此跳表可以支持顺序访问，返回每一个元素的时间为 Θ(1)。在SkipList类中增加顺序访问函数Begin和Next，分别返回字典中第一个元素的指针和下一个元素的指针(元素按从小到大次序排列)，在没有第一个或下一个元素时，二者均应返回0。每个函数的复杂度均应为Θ(1)。试测试代码的正确性。

5. 编写一个级的分配程序，采用本节中所介绍的把随机数的取值范围划分为若干段的策略。

6. 修改类SkipList 以允许有相同值的元素出现。每个链从左到右按递增次序排列。用合适的数据测试代码。

7. 扩充类SkipList，增加删除最小值、最大值元素的函数，以及按升序输出元素的函数。每个函数的复杂性分别是多少？

7.4 散列表描述

7.4.1 理想散列

字典的另一种描述方法就是散列（hash），它是用一个散列函数（hash function）把关键字

映射到散列表（hash table）中的特定位置。在理想情况下，如果元素 e 的关键字为k，散列函数为f，那么e 在散列表中的位置为$f(k)$。要搜索关键字为k 的元素，首先要计算出$f(k)$，然后看表中$f(k)$处是否有元素。如果有，便找到了该元素。如果没有，说明该字典中不包含该元素。在前一种情况中，如果要删除该元素，只需把表中 $f(k)$位置置为空即可。在后一种情况中，可以通过把元素放在$f(k)$位置以实现插入。

例7-5　考察例7-1中的学生记录字典。假设不用学生名，而是用学生ID号(为六位整数) 作为关键字。在一个班中，假设最多有100个学生，他们的ID号在951000和952000之间。函数$f(k)=k-951000$把学生ID号映射到0到1000之间。采用元素类型为E的数组ht[1001]作为散列表，该表被初始化为0，即对于$0 \le i \le 1000$，有ht[i].key=0。要搜索关键字为 k的元素，需计算 $f(k)=k-951000$。如果元素的关键字域不为0，则此元素就在ht[$f(k)$]中。如果为0，则字典中没有该元素。在后一种情况下，可以把该元素插入到相应位置。前一种情况下可以通过把 ht[$f(h)$].key置为0从而实现删除。

在理想情况下，初始化一个空字典需要的时间为 $\Theta(b)$(b为散列表中位置的个数），搜索、插入、删除操作的时间均为 $\Theta(1)$。尽管理想的散列方法在许多场合都适用，但还有许多应用因为关键字变化范围太大而不能创建一个这样的散列表。例如例 7-1中，每个学生的名字被截为最多12个字母长，大写字母由相应的小写字母来代替，特殊的字如连字符要被删掉。若现在用截短了的名字作为关键字。不够12个字母长的关键字可以在其前面加空格以补足12个。每个关键字可以被映射到相应的数值型关键字， a 对应1，b 对应2，…，z 对应26。这时关键字的范围为1到27^{12}-1(zzzzzzzzzzzz)，范围太大，因此不能像理想散列方法那样建立数组 ht。

7.4.2 线性开型寻址散列

1. 方法

当关键字的范围太大，不能用理想方法表示时，可以采用比关键字范围小的散列表以及把多个关键字映射到同一位置的散列函数。虽然有多种函数映射方法，但最常用的还是除法映射。除法散列函数的形式如下：

$$f(k) = k \% D \qquad (7\text{-}2)$$

其中k 为关键字，D是散列表的大小(即位置数)，%为求模操作符。散列表中的位置号从0到D-1，每一个位置称为桶(bucket)。若关键字不是正整数类型 (如int，long, char, unsigned char等)，则在计算f(k)之前必须把它转换成非负整数。对于一个长字符串，可以采用取其 2个字母或4个字母而变成无符号整数或无符号长整数的方法。 f(k)是存储关键字为 k的元素的起始桶（ home bucket）。在良性情况下，起始桶中所存储的元素即是关键字为 K的元素。

图7-2a 中给出一个散列表ht，桶号从0到10。表中有3个元素，除数D为11。因为80%11=3，则80的位置为3，40%11=7，65%11=10。每个元素都在相应的桶中。散列表中余下的桶为空。

现在假设要在表中插入 58。58的起始桶为f(58)=58%11=3。这个桶已被另一个数占用，这时就发生了碰撞（collision）。一般说来，一个桶中可以存储多个元素，因此发生碰撞也没什么了不起的。存储桶中若没有空间时就发生溢出（ overflow）。但在我们的表中，每个桶只能存储一个元素，因此碰撞和溢出会同时发生。那么把 58插到哪儿呢?最简单的方法就是把58存储到表中下一个可用的桶中，这种解决溢出的方法叫作线性开型寻址（ linear open addressing）。

可把58存储在4号桶中。假设下一个要插入的元素值为 24，24%11为2。2号桶为空，则把

24放入2号桶中。此时散列表如图7-2b 所示。现在要插入35。35的起始桶已满。采用线性开型寻址，结果如图7-2c 所示。最后一例，插入98。其起始桶10已满，则插入下一个可用桶0中。由此看来，在寻找下一个可用桶时，表被视为环形的。

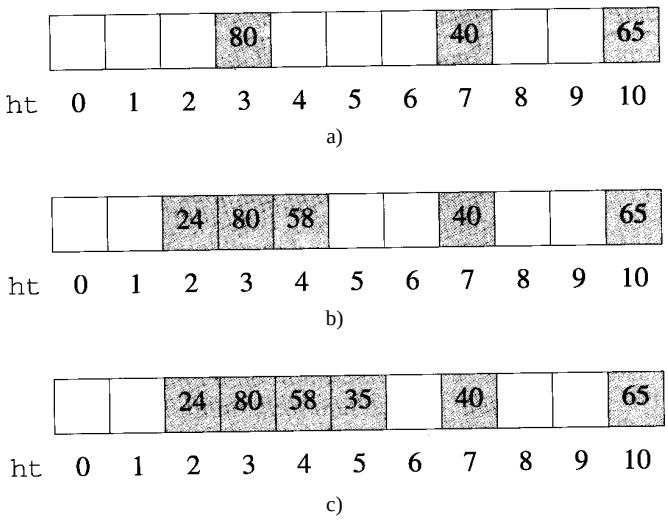


图7-2 散列表

刚才讲述了怎样用线性开型寻法进行插入，现在介绍如何对这样一个散列表进行搜索。首先搜索起始桶f(k)，接着对表中后继桶进行搜索，直到发生以下情况： 1) 存有关键字为K 的桶已找到，即找到了要搜索的元素； 2) 到达一个空桶； 3) 又回到f(k)桶。若发生后两种情况，则说明表中没有关键字为k的元素。

在完成一次删除操作后，必须能保证上述的搜索过程仍能够正常进行。若在图 7-2c 中删掉58，不能仅仅把4号桶置为空，否则就无法找到关键字为 35的元素。删除会带来多个元素的移动。可从欲删除的元素开始逐个检查每个桶以确定要移动的元素，直到到达一个空桶或返回删除操作所对应的桶为止。实现删除的另一种策略是为每个桶增加一个 NeverUsed域。在表初始化时，该域被置为true。当桶中存入一个元素时， NeverUsed域被置为false。这时搜索的结束条件2) 变成：桶的NeverUsed域为true。实现删除时，只需把表中相应位置置为空即可。一个新元素可被插入到从其对应的起始桶开始所找到的第一个空桶中。注意在这种方案中，NeverUsed不会被重置为true。用不了多长时间，所有的(或几乎所有的)桶的NeverUsed域均会被置为false，这时搜索过程可能需要检查所有的桶。在这种情况下，为了提高性能，必须重新组织该表。比如，可把所有余下的元素都插入到一个空的散列表中。

2. C++实现

程序7-11给出了采用线性开型寻址的散列表的类定义。在类定义中假定散列表中每个元素的类型为E，每个元素都有一个类型为K的key域。key是用来计算起始桶的，因此类型K必须能够适应取模操作 %。散列表使用了两个数组， ht和empty。当且仅当 ht[i]中不含有元素时，empty[i]为true。程序7-12给出了构造函数的代码。

程序7-11 散列表的C++类定义

```
template<class E, class K>
```

```
class HashTable {
  public:
    HashTable(int divisor = 11);
    ~HashTable() {delete [ ] ht; delete [] empty;}
    bool Search(const K& k, E& e) const;
    HashTable<E,K>& Insert(const E& e);
  private:
    int hSearch(const K& k) const;
    int D; // 散列函数的除数
    E *ht; // 散列数组
    bool *empty; // 一维数组
};
```

程序7-12 HashTable的构造函数

```
template<class E, class K>
HashTable<E,K>::HashTable(int divisor)
{// 构造函数
  D = divisor;

  // 分配散列数组
  ht = new E [D];
  empty = new bool [D];

  // 将所有桶置空
  for (int i = 0; i < D; i++)
    empty[i] = true;
}
```

程序7-13给出了搜索函数。共享成员函数 Search 在没有找到关键字值为 k 的元素时返回false，否则返回true。并且若找到该元素，则在参数e 中返回该元素。Search 函数调用了私有成员函数hSearch。在满足如下三种情形之一时， hSearch 用来返回b 号桶：1) empty[b]为false且ht[b] 的关键字值为k；2)表中没有关键字值为k 的元素， empty[b]为true， 可把关键字值为k的元素插入到b 号桶中；3)表中没有关键字值为k 的元素， empty[b]为false， ht[b]的关键字值不等于k， 且表已满。

程序7-13 查询函数

```
template<class E, class K>
int HashTable<E,K>::hSearch(const K& k) const
{// 查找一个开地址表
 // 如果存在，则返回k的位置
 // 否则返回插入点（如果有足够空间）
  int i = k % D; // 起始桶
  int j = i;     // 在起始桶处开始
  do {
    if (empty[j] || ht[j] == k) return j;
    j = (j + 1) % D;  // 下一个桶
```

```
    } while (j != i); // 又返回起始桶?

  return j;  // 表已经满
}

template<class E, class K>
bool HashTable<E,K>::Search(const K& k, E& e) const
{// 搜索与k匹配的元素并放入e
 // 如果不存在这样的元素，则返回false
  int b = hSearch(k);
  if (empty[b] || ht[b] != k) return false;
  e = ht[b];
  return true;
}
```

程序7-14给出了Insert 函数的实现代码。函数从一开始就调用 hSearch。若hSearch返回的i 号桶为空，则表中没有关键字为 k 的元素，可以把该元素 e 插入到该桶中。若返回的桶非空，则要么在桶中已包含了关键字为 k 的元素，要么表已满。在前一种情况下， Insert函数引发一个BadInput异常。在后一种情况引发一个 NoMem异常。练习14要求编写Delete函数的代码。

程序7-14 散列表的插入

```
template<class E, class K>
HashTable<E,K>& HashTable<E,K>::Insert(const E& e)
{// 在散列表中插入
  K k = e; // 抽取key值
  int b = hSearch(k);

  // 检查是否能完成插入
  if (empty[b]) {empty[b] = false;
                 ht[b] = e;
                 return *this;}

  // 不能插入，检查是否有重复值或表满
  if (ht[b] == k) throw BadInput(); // 有重复
  throw NoMem(); // 表满
}
```

当E为用户自定义的类或数据类型时，则有必要重载如 %，!=，==等操作符。程序7-7给出了重载的例子。

3. 性能分析

这里只分析时间复杂性。设b为散列表中桶的个数。散列函数中D为除数且$b=D$。初始化表的时间为$\Theta(b)$。当表中有n个元素时，最坏情况下插入和搜索时间均为$\Theta(n)$。当所有n个关键字值都在同一个桶中时即出现最坏情况。通过比较散列在最坏情况下的复杂性与线性表在最坏情况下的复杂性，可以看到二者完全相同。

但散列的平均性能还是相当好的。用U_n和S_n来分别表示在一次成功搜索和不成功搜索中平

均搜索的桶的个数。对于线性开型寻址，有如下公式成立：

$$U_n \sim \frac{1}{2}\left[1+\frac{1}{(1-\alpha)^2}\right]$$

$$S_n \sim \frac{1}{2}\left[1+\frac{1}{1-\alpha}\right]$$

其中$\alpha=n/b$为负载因子（loading factor）。

所以若α=0.5，则在不成功搜索时平均搜索的桶的个数为2.5个，成功搜索时则为1.5个。当α=0.8时，为50.5和5.5。此时假设n至少为51。当负载因子为0.5时，采用线性开型寻址散列表的平均性能要比线性表好得多。

4. 确定D

在实际应用过程中，D的选择对于散列的性能有着重大的影响。当D为素数或D没有小于20的素数因子时，可以使性能达到最佳(D等于桶的个数b)。

为了确定D的值，首先要了解影响成功搜索和不成功搜索性能的因素。通过U_n和S_n的公式，可以确定最大的α值。根据n的值（或是估计值）和α的值，可以得到b的最小许可值。然后找到一个比b大的最小的整数，该整数要么是素数，要么没有小于20的素数因子。这个整数即可作为D和b的值。

例7-6 设计一个有近1000个元素的散列表。要求成功搜索时平均搜索的桶的个数不得超过4，不成功搜索时平均搜索的桶的个数不超过50.5。由U_n的公式，可得到$\alpha\leq$0.9，由S_n的公式，可以得到$4\geq 0.5+1/(2(1-\alpha))$或$\alpha\leq 6/7$。因此，$\alpha\leq\min\{0.9, 6/7\}=6/7$。因此$b$最小为$\lceil(7n/6)\rceil$=1167。$b=D$=37*37=1369，应为一个比较好的值(虽然不是最小的选择)。

另一种计算D的方法是首先根据散列表的最大空间来确定b的最大可能值，然后取D为不大于这个最大值的整数，该整数要么是素数，要么没有小于20的素数因子。例如，如果在表中最多可以分配530个桶，则D和b的最佳选择为23(因23*23=529)。

7.4.3 链表散列

1. 方法

当散列发生溢出时，链表是一种好的解决方法。图7-3给出了散列表在发生溢出时采用链表来进行解决的方法。在上一个例子中，散列函数的除数为11。在该散列表的组织中，每个桶仅含有一个节点指针，所有的元素都存储在该指针所指向的链表中。

在搜索关键字值为k的元素时，首先要计算其起始桶。起始桶号为$k\%D$，然后搜索该桶所对应的链表。在插入时，首先要保证表中不含有相同关键字的元素。当然，此时的搜索仅限于该元素的起始桶所对应的链表。由于每次插入都要首先进行一次搜索，因此把链表按照升序排列比无序排列会更有效。最后，为了删除关键字值为k的元素，首先访问起始桶对应的链表，找到该元素，然后删除。

2. C++ 实现

可以用一组有序链表来实现链表散列，见程序7-15。由于该类引用了SortedChain类的成员，所以必须在类型E上定义操作符cout，==，!=和<。在程序7-7中给出了操作符重载的范例。

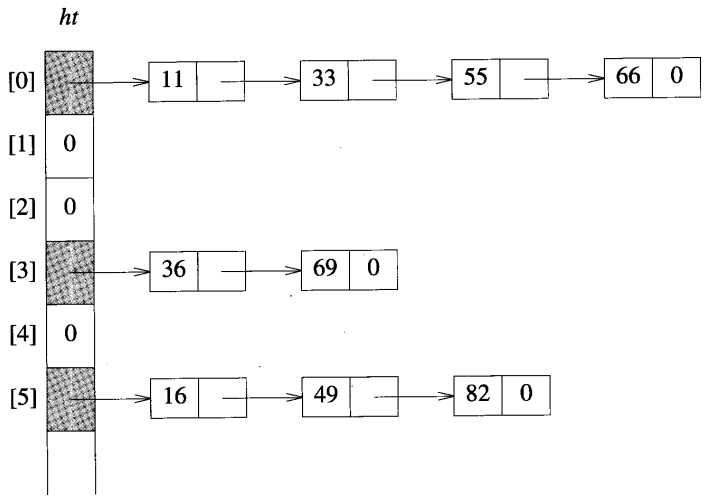


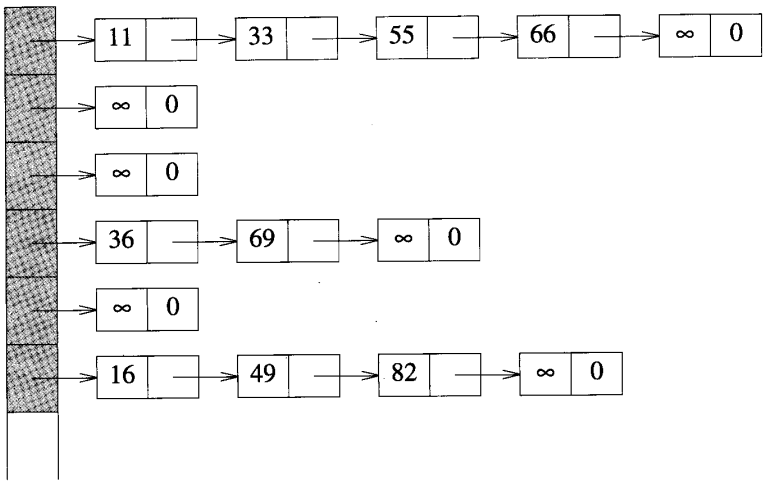
图7-3 链表散列

程序7-15 链表散列的类定义

```
template<class E, class K>
class ChainHashTable {
  public:
    ChainHashTable(int divisor = 11)
      {D = divisor;
       ht = new SortedChain<E,K> [D];}
    ~ChainHashTable() {delete [] ht;}
    bool Search(const K& k, E& e) const
        {return ht[k % D].Search(k, e);}
    ChainHashTable<E,K>& Insert(const E& e)
        {ht[e % D].DistinctInsert(e);
         return *this;}
    ChainHashTable<E,K>& Delete(const K& k, E& e)
        {ht[k % D].Delete(k, e);
         return *this;}
    void Output() const;   // 输出散列表
  private:
    int D;                 // 位置数
    SortedChain<E,K> *ht;  // 链表数组
};
```

3. 一种改进方法

在图7-3所示的每条链表中加上一个尾节点，可以稍稍改进程序的性能。尾节点的关键字值最起码要比散列中所有元素的关键字值都大。在图 7-4中尾节点的关键字用∞来表示。在实际应用中，当关键字为整数时，可以采用在 limits.h文件中定义的INT_MAX常量。由于有了尾节点，在SortedChain的定义中出现的*i* && *i*->data均可改为i->data。在图7-4中每条链有不同的尾节点。在实际实现过程中，所有的链表可共用同一个尾节点。



∞ 表示非常大的关键字

图7-4 带尾节点的链表散列

4. 与线性开型寻址比较

把线性开型寻址与没有尾节点的链表散列进行比较。令 s为每个元素需占用的空间 (以字节为单位)，每个指针和每个整数类型的变量各占 2个字节空间。同时，设散列表中有 b个桶和n个元素。首先注意到当使用线性开型寻址时有 $n \le b$，而使用链表时n可能大于b。

采用线性开型寻址所需要的空间为$b(s+2)$个字节，其中s为每个元素占用的字节数。而使用链表所需要的空间为$2b+2n+ns$字节。当$n < bs/(s+2)$时，链表方法所占用的空间要比开型寻址少。注意，如果在实现线性开型寻址散列时采用一种节约空间的方法，二者间的比较结果可能会发生变化。比如把数组 empty压缩为$b/8$个字节（在程序实现中该数组占用了 $2b$个字节）。另外，如果已知关键字的范围并非整个整数范围时，可以使用一个不可能用到的整数 (-1或INT_MAX)作为空桶的关键字。

在最坏情况下，用两种方法进行搜索时，都要求搜索所有的 n 个元素。链表散列的平均搜索次数可用如下方法计算。对一条有 i 个节点的有序链表的一次不成功搜索可能要搜索 1, 2, 3, … 第i 个节点，其中$i \ge 0$。设每种情况发生的概率都相同，则一次不成功搜索所要搜索的节点数为：

$$\frac{1}{i}\sum_{j=1}^{i} j = \frac{i(i+1)}{2i} = \frac{i+1}{2}$$

其中$i \ge 1$。当$i=0$时平均搜索节点数为0。对于链表散列，假定链表的平均长度为 $n/b=\alpha$。当$\alpha \ge 1$时可用α代替上面公式中的i，从而可以得到：

$$U_n \sim \frac{\alpha+1}{2}, \alpha \geqslant 1$$

当$\alpha < 1$时，由于链表的平均长度为α，搜索次数不可能比节点数多，因此$U_n \le \alpha$。

计算S_n 时，需要知道n 个标识符距其链表头节点的平均距离。为了计算距离，假定各标识符是按升序插入的。当插入第 i 个标识符时，其所在链表的长度为 $(i-1)/b$。由于标识符按升序插入，所以第i 个标识符被插入到相应链表的尾部。因此对该标识符的搜索需查找 $1+(i-1)/b$个节点。还需要注意到，由于标识符是按升序插入的，它与链表头节点的距离并不随以后的插入

操作而改变。假定每个标识符被搜索的概率都相同，则有

$$S_n = \frac{1}{n}\sum_{i=1}^{n}\{1 + (i-1)/b\} = 1 + \frac{n-1}{2b} \sim 1 + \frac{\alpha}{2}$$

把采用链表的公式与采用线性开型寻址的公式相比较，可以看到使用链表时的平均性能要优于使用线性开型寻址。例如，当 α=0.9时，在链表散列中的一次不成功搜索，平均要检查 0.9个元素，而一次成功搜索需要检查 1.45个元素。对于线性开型寻址来说，不成功搜索时需要检查50.5个元素，成功时也需要检查5.5个元素。

5. 与跳表比较

跳表和散列均使用了随机过程来提高字典操作的性能。在使用跳表时，插入操作用随机过程来决定一个元素的级数。这种级数分配不考虑要插入元素的值。在散列中，当对不同元素进行插入时，散列函数随机地为不同元素分配桶，但散列函数需要使用元素的值。

通过使用随机过程，跳表和散列操作的平均复杂性分别为对数时间和常数时间。跳表的最坏时间复杂性为 $\Theta(n+\text{MaxLevel})$，而散列的最坏时间复杂性为 $\Theta(n)$。跳表中指针平均占用的空间约为 Maxlevel + $n/(1-p)$，在最坏情况下可能相当大。链表散列的指针所占用的空间为 $D+n$。

不过，跳表比散列更灵活。例如，只需简单地沿着 0级链就可以在线性时间内按升序输出所有的元素。而采用链表散列时，需要 $\Theta(D+n)$ 时间去收集 n 个元素并且需要 $O(n\log n)$时间进行排序，之后才能输出。对于其他的操作，如查找或删除最大或最小元素，散列可能要花费更多的时间（仅考虑平均复杂性）。

练习

8. 用理想散列来实现字典的 C++类。假定元素的关键字是介于 0～MaxKey之间的整数，MaxKey是在创建字典时由用户确定的。用适当的数据测试代码的正确性。

9. 在理想散列的某些应用中，尽管有足够的空间来构建关键字范围很大的散列，但由于初始化散列所需要的时间 $\Theta(b)$太大以至于这种方法不切实际。例如，若关键字的范围为 1 000 000，而我们只需执行100个操作，百万单位的时间将花费在初始化上，而花在操作本身上的时间仅为100个单位。

针对这种应用，可以对理想散列的方法进行修改，采用两个数组： ht和ele。ht是从0到MaxKey的整数数组(MaxKey是最大可能的关键字)；ele 数组的数据类型为E，其元素个数由要插入数组的不同关键字数决定（一般来说比 MaxKey要小得多）。两个数组都不需要初始化。

由于不会被删除， ele中的位置可被依次编号为 0，1，2，…。计数器LastE表示上次所用到的ele中的位置，其初值为 - 1。ht[j]中的值可能无效也可能是 ele的索引（用来寻找关键字为 j 的元素）。

把上述思想应用于理想散列的初始化、搜索、插入和删除函数。每个函数的复杂性应为 $\Theta(1)$。这些函数应作为类IdealHashTable的共享成员函数。试测试代码的正确性。

10. 指出在线性开型寻址散列中进行顺序访问所存在的困难。

11. 分别用公式化线性表和线性开型寻址编写字典程序。在该练习中不必编写删除函数。假设关键字为整数且 D为961。在表中插入随机产生的 n=500个不同的整数，并在每 m次插入后进行搜索。测出每次搜索的平均时间。从该练习中能得出什么结论？

12. 采用线性开型寻址，在下列各种情况下找到散列函数除数 D的合适的值。

1) n=50 , $S_n \leq 3$, $U_n \leq 20$。

2) n=500 , $S_n \leq 5$, $U_n \leq 60$。

3) n=10 , $S_n \leq 2$, $U_n \leq 10$。

13. 对于以下各种条件，找出散列函数除数D的合适的值。假设采用线性开型寻址。

1) MaxElements≤530。

2) MaxElements≤130。

3) MaxElements≤150。

*14. 编写类HashTable(见程序7-11)的共享成员函数Delete的代码。不要改变类中其他成员函数。代码的最坏时间复杂性是多少?用合适的数据测试代码的正确性。

15. 编写采用线性开型寻址的散列表的类定义，要求在执行删除操作时采纳NeverUsed思想。为每个函数编写完整的 C++代码。当60%空桶的NeverUsed域为false时，要重新组织散列表。编写重新组织散列表的代码，重新组织过程在必要时需要移动元素，并且对每个空桶，其NeverUsed域必须置为true。试测试代码的正确性。

16. 指出在链表散列中进行顺序访问所存在的困难。

17. 开发新的SortedChainWithTail类，为每个有序链表增加一个尾节点。在操作开始时，把欲搜索、插入或删除的元素或关键字放入尾节点中以简化代码。比较有尾节点和没有尾节点时的运行性能。

18. 从低层开发ChainHashTable类。定义自己的类HashNode，其中包含data域和link域，不要使用链表类的任何版本。测试代码。

19. 从类SortedChainWithTail(见练习17)中派生出类ChainHashTable。比较该版本与练习18中ChainHashTable版本的性能。

20. 开发类ChainHashWithTail，每个散列链表皆为有尾节点的有序链表。所有链表的尾节点在物理上为同一个。比较该类与类ChainHashTable(见程序7-15)的运行性能。

21. 为了简化链表散列的插入和删除操作，可以在每个链表中加一个头节点。头节点是对上文中尾节点的补充。所有的插入和删除都在头、尾节点之间进行，因此不会在链表的头部进行插入和删除操作。

1) 所有链表是否能使用同一个头节点？为什么？

2) 在头节点的关键字域设置一个特定的值是否合理？为什么？

3) 编写一个新的链表散列类定义，要求使用头节点和尾节点。编写所有函数的代码。

4) 用合适的数据测试代码的正确性。

5) 指出带头、尾节点，只带尾节点和没有头、尾节点这三种情况的优点和缺点。你会推荐哪种？为什么？

7.5 应用——文本压缩

为了节约空间，常常需要把文本文件采用压缩编码方式存储。例如，一个包含 1000个*x* 的字符串和2000个*y* 的字符串的文本文件在不压缩时占用的空间为 3002字节（每个*x* 或*y* 占用一个字节， 2个字节用来表示串的结尾）。同样是这个文本文件，采用游程长度编码 (run-length coding)，可存储为字符串1000x2000y，仅为10个字母，占用12个字节。若采用二进制表示游程长度(1000和2000)可以进一步节约空间。如果每个游程长度占用 2个字节，则可表示的最大游程长度为2^{16}，这样上例中的字符串只需用 8个字节来存储。当要读取编码文件时，需对其进行解码。由压缩器（compressor）对文件进行编码，由解压器（decompressor）进行解码。

在本节中，采用由Lempel、Ziv 和Welch 所开发的技术，来设计对文本文件进行压缩和解压缩的C++ 代码。这种技术被称为LZW方法，该方法相对简单，采用了理想散列和链表散列。

7.5.1 LZW压缩

LZW压缩方法把文本的字符串映射为数字编码。首先，为该文本文件中所有可能出现的字母分别分配一个代码。例如，要压缩的文本文件为：

```
aaabbbbbbaabaaba
```

字符串由a 和b 组成。为a 分配代码0，为b 分配代码1。字符串和编码的映射关系存储在字典中。每个字典的入口有 2个域： key 和code。由code 表示的字符串存在域key中。本例的初始字典由图7-5的前两列给出 (也就是代码0和1)。

code	0	1	2	3	4	5	6	7
key	a	b	aa	aab	bb	bbb	bbba	aaba

图7-5 aaabbbbbbaabaaba的LZW压缩字典

若初始字典如图7-5所示，LZW压缩器不断地在输入文件中寻找在字典中出现的最长的前缀p，并输出其相应的代码。若输入文件中下一个字符为 c，则为pc分配下一个代码，并插入字典。这种策略称为LZW规则。

用LZW方法来压缩上例字符串。文件中第一个在字典中出现的最长前缀为a，输出其编码0。然后为字符串aa 分配代码2，并插入到字典中。余下字符串中在字典内出现的最长前缀为 aa，输出aa 对应的代码2，同时为字符串aab 分配代码3，并插入到字典中。注意，虽然为aab 分配的代码为3，但仅输出aa 的代码2。后缀b 将作为下一个代码的组成部分。不输出 3是因为编码表不是压缩文件的组成部分。相反，在解压时，编码表由压缩文件重新构造。只要采用 LZW规则，这种重建就是可能的。

紧接2之后，输出b 对应的代码。为bb 分配4，并插入字典中。然后输出bb 的编码，为bbb分配代码5并插入字典中。输出5，并为bbba 分配编码6，然后插入字典中。接下来输出aab 的代码3，同时为aaba 分配代码7并插入字典。因此上例中字符串的编码为0214537。

7.5.2 LZW压缩的实现

1. 输入和输出

压缩器的输入为文本文件，而输出为二进制文件。为了简单起见，假设输入文件名中不包含“ . ”(即文件名中没有扩展名)。如果输入文件名为InputFile，则输出文件名为InputFile.zzz。同时假设用户在命令行中输入要压缩的文件名。若压缩程序为Compress，则命令行为

```
Compress text
```

就可以得到文件text的压缩版本text.zzz。若用户没有输入文件名就应提醒用户输入。

函数SetFiles为输入和输出创建输入输出流（见程序7-16）。它假定函数main的原型为：

```
void main (int argc , char * argv[ ])
```

并假定in和out为全局变量，类型分别为 ifstream和ostream。argc是命令行中参数的个数，argv[i]为指向第i个参数的指针。若命令行为

```
Compress text
```

则argc为2， argv[0]指向字符串Compress , argv[1]指向text。

程序7-16 建立输入输出流

```
void SetFiles(int argc, char* argv[])
```

```
{// 创建输入流和输出流
  char OutputFile[50], InputFile[50];
  // 检查是否提供了文件名
  if (argc >= 2) strcpy(InputFile,argv[1]);
  else {// 没有提供文件名, 提示用户输入
       cout << "Enter name of file to compress"
            << endl;
       cout << "File name should have no extension" << endl;
       cin >> InputFile;}

  // 文件名不应有扩展名
  if (strchr(InputFile,'.')) {
     cerr << "File name has extension" << endl;
     exit(1);}

  // 以二进制方式打开文件
  in.open(InputFile,ios::binary);
  // in.open(InputFile); //对于g++而言
  if (in.fail()) {cerr << "Cannot open " << InputFile << InputFile << endl;
                  exit(1);}
  strcpy(OutputFile,InputFile);
  strcat(OutputFile, ".zzz");
  out.open(OutputFile,ios::binary);
}
```

2. 组织字典

字典中每个元素有2个域：code和key。code为整型，而key为长字符串。然而，每个长度 l ＞1的关键字的前 l-1个字符(称为关键字前缀)都可在字典中找到。因为每个字典入口都有一个不同的代码 (不仅只是有唯一的关键字)，因此可以用代码代替其关键字前缀。在图7-5的例子中，关键字 aa 可以替换为0a，而aaba 可以替换为3a。现在字典的形式如图7-6所示。

code	0	1	2	3	4	5	6	7
key	a	b	0a	2b	1b	4b	5a	3a

图7-6 aabbbbbbaabaaba修改后的LZW压缩字典

为简化对压缩文件的解码，应对每个代码采用相同的位。更进一步，可假设每个代码为 12位长。因此可以最多分配 2^{12}=4096个代码。由于每个字符为 8位，因此关键字可以用长整数 (32位)来表示。低8位用来表示关键字中的最后一个字符，后 12位用来表示其前缀。字典本身可以表示为链表散列。若素数D=4099作为散列函数的除数，存储密度就会小于 1，因为字典中最多有4096个入口。声明

```
ChainHashTable＜element , unsigned long＞ h(D)
```

足以用来建立字典表。在我们的应用程序中，没有使用与 ChainHashTable类的成员函数相关的Delete函数。

3. 输出代码

因为每个代码为12位，每个字符为8位，所以只能输代码的一部分作为一个字符。先输出代码的前8位，余下4位留待以后输出。当要输出下一个代码时，加上先前余下 4位，共16位可

作为2个字符输出。程序7-17给出输出函数的C++代码。maskl为255，mask12为15，excess为4，ByteSize为8。当还有余位待输出时status为1。此时，余下的4位放在变量Leftover中。

程序7-17 输出代码

```
void Output(unsigned long pcode)
{// 输出8 位, 余下的位保存在LeftOver中
  unsigned char c,d;
  if (status) {// 余下4位
    d = pcode & mask1; //右边ByteSize位
    c = (LeftOver << excess) | (pcode >> ByteSize);
    out.put(c);
    out.put(d);
    status = 0;}
  else {
    LeftOver = pcode & mask2; // 右边多余的位
    c = pcode >> excess;
    out.put(c);
    status = 1;}
}
```

4. 压缩

程序7-18给出LZW压缩算法的代码。首先用256个(alpha=256)8位字符及其代码对字典进行初始化，变量used用来保存目前已用的代码。因为每个代码为12位，则最多可分配4096个代码。在字典中找最长前缀，从长度为1, 2, 3…开始,直到发现某一个字符不在字典中为止。同时输出前缀对应的代码,并为该字符串分配一个新的代码(除非4096个代码全部用完)。

程序7-18 LZW压缩器

```
void Compress()
{// Lempel-Ziv-Welch压缩器
  // 定义并初始化代码字典
  ChainHashTable<element, unsigned long> h(D);
  element e;
  for (int i = 0; i < alpha; i++) {// 初始化
    e.key = i;
    e.code = i;
    h.Insert(e);
    }
  int used = alpha; // 所使用的代码数目

  // 输入并压缩
  unsigned char c;
  in.get(c);      // 输入文件的第一个字符
  unsigned long pcode = c; // 前缀代码
  if (!in.eof()) {// 文件长度 > 1
    do {// 处理文件的其余部分
      in.get(c);
```

```
        if (in.eof()) break;  // 完成
        unsigned long k = (pcode << ByteSize) + c;
        // 检查 k的代码是否已经在字典中
        if (h.Search(k, e)) pcode = e.code;  // 在字典中
        else {// k不在表中
             Output(pcode);
             if (used < codes) // 创建新的代码
               {e.code = used++;
                e.key = (pcode << ByteSize) | c;
                h.Insert(e);}
             pcode = c;}
      } while(true);

      // 输出最后一个（部分）代码
      Output(pcode);
      if (status) {c = LeftOver << excess; out.put(c);}
      }

   out.close();
   in.close();
}
```

5. 头文件及main函数

程序7-19给出了压缩程序的头文件、常量定义、数据类型、全局变量和 main函数。

程序7-19　压缩程序的main函数

```
#include <fstream.h>
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include "chash.h"

// 常量
const D = 4099,      // 散列函数的除数
        codes = 4096,  // 2^12
        ByteSize = 8,
        excess = 4,    // 12 - ByteSize
        alpha = 256,   // 2^ByteSize
        mask1 = 255,   // alpha - 1
        mask2 = 15;    // 2^excess - 1

class element {
  friend void Compress();
  public:
    operator unsigned long() const {return key;}
    element& operator =(unsigned long y)
      {key = y; return *this;}
```

```
  private:
    int code;
    unsigned long key;
};

int LeftOver,   // 尚未输出的代码位
   status = 0; // 0 意味着在LeftOver中没有未输出的位
ifstream in;
ofstream out;
void main ( int argc , char* argv [ ] )
{
 SetFiles (argc, argv);
 Compress ( ) ;
}
```

7.5.3 LZW解压缩

解压时要输入代码，然后用代码所表示的文本来替换这些代码。代码到文本的映射可按下面方法重新构造。首先把分配给单一字母的代码插入字典中。象前面一样，字典的入口为代码-文本对。然而此时是根据给定的代码，去寻找相应入口(而不是根据文本)。压缩文件中的第一个代码对应于一个单一的字母，因此可以由该字母代替。对于压缩文件中的其他代码 p，要考虑到两种情况：1) 在字典中；2) 不在字典中。当p 在字典中时，找到与p 相关的文本$text(p)$并输出。并且，由压缩器原理可知，若在压缩文件中代码 q 写在p 之前且$text(q)$ 是与q 对应的文本，则压缩器会为文本$text(q)$ (其后紧跟着$fc(p)$, $text(p)$的第一个字符) 分配一新代码。因此在字典中插入序偶(下一个代码,$text(q)fc(p)$)。情况2) 只有在当前文本段形如 $text(q)text(q)fc(q)$ 且 $text(p)=text(q)fc(q)$ 时才会发生。相应的压缩文件段是 qp。在压缩的过程中,为$text(q)fc(q)$ 分配的代码为p。在解压过程中，在用 $text(q)$代替q 后，又遇到代码p。然而，此时字典中没有与 p 对应的文本。因为这种情况只在解压文本段为 $text(p)text(q)fc(q)$ 时才会发生，因此可以对 p 解码。当遇到一个没有定义代码文本对的代码p时，p 对应的文本为$text(q)fc(q)$,其中q 为p 前面的代码。

用此解码策略来解压前面的例子。字符串 aaabbbbbbaabaaba 被压缩为代码0214537。首先，初始化字典，在其中插入 (0,a)和(1,b)后，得到图7-5字典的前两个入口。压缩文件的第一个代码为0，则应用 a代替。下一个代码 2未定义。因为前一个代码为 0，且$text(0)$=a，$fc(0)$=a，则 $text(2)=text(0)fc(0)$=aa。因此用aa 代替2，并把(2, aa) 插入字典中。下一个代码1由b来替换且把(3 ,$text(2)fc(1)$)=(3, aab) 插入字典中。下一代码 4不在字典中。其前面的代码为 1，则 $text(4)=text(1)fc(1)$=bb。把(4, bb)插入字典中，且在解压文件中输出bb。当遇到下一个代码5时，(5, bbb)被插入字典中，同时把bbb 输出到解压文件中。再下一代码为3，$text(3)$=aab 则把aab 输出，并将序偶(6，$text(5)fc(3)$)=(6，bbba) 插入字典。当遇到7时，把(7, $text$ (3)$fc(3)$) = (7，aaba) 插入字典中并输出aaba。

7.5.4 LZW解压缩的实现

1. 输入/输出

函数Setfiles(见程序7-20)与压缩器中的对应函数功能相同。它输入解压文件的名称，并加上.zzz的扩展名以得到相应的压缩文件名。

程序7-20 建立输入输出流

```
void SetFiles(int argc, char* argv[])
{// 确定文件名
  char OutputFile[50], InputFile[50];

  // 检查是否提供了文件名
  if (argc == 2) strcpy(OutputFile,argv[1]);
  else {// 没有提供文件名, 提示用户输入
       cout << "Enter name of file to decompress"
            << endl;
       cout << "Omit the extension .zzz"  << endl;
       cin >> OutputFile;}

  // 文件名中不应带扩展名
  if (strchr(OutputFile,'.'))
    {cerr << "File name has extension" << endl;
     exit(1);}

  strcpy(InputFile, OutputFile);
  strcat(InputFile, ".zzz");

  // 以二进制方式打开文件
  in.open(InputFile,ios::binary);
  // in.open(InputFile) 对g++而言
  if (in.fail()) {cerr << "Cannot open "
                       << InputFile  << endl;
                  exit(1);}
  out.open(OutputFile,ios::binary);
  // out.open(OutputFile) 对g++而言
}
```

2. 组织字典

因为可根据所给的代码查询字典并且代码总数为 4096，所以可以采用数组 ht[4096] ，把 *text*(*p*)存储在 ht[*p*]中。像理想散列那样使用数组 ht，散列函数 *f* (*k*)=*k*。同时像图 7-6那样，*text* (*p*)可以存储为其前缀代码和最后一个字符 (后缀)。在解压过程中，把 *text* (*p*)的前缀和后缀分别作为整数和字符存储非常方便。因此若 *text* (*p*)=*text* (*q*)*c*，则ht[*p*].suffix字符为c，ht[*p*].code等于*q*。

当采用这种字典组织方式时，见程序 7-21，可从最后一个字符 ht[*p*].suffix开始，按从右到左的次序来构建 *text*(*q*)。程序从表ht中得到代码大于 alpha的后缀值。*text*(*p*)被收集到数组s[ ]中，然后被输出。由于*text*(*p*)是从右到左存储的，因此*text*(*p*)的第一个字符存储在s[size]中。

程序7-21 计算text(code)

```
void Output(int code)
{// 输出与代码相对应的串
  size = -1;
  while (code >= alpha) {// 字典中的后缀
    s[++size] = ht[code].suffix;
```

```
        code = ht[code].prefix;
        }
      s[++size] = code;  // code < alpha

      // 解压所得的串为s[size] ... s[0]
      for (int i = size; i >= 0; i--)
        out.put(s[i]);
    }
```

3. 输入代码

由于12位代码在压缩文件中是按8位字节顺序表示的，所以要把函数Output (见程序7-17)的输出结果转换过来。可由 GetCode函数(见程序7-22)来完成这种转换。此处唯一的新常量是mask，其值为15，它可以帮助我们得到一个字节的低4位。

程序7-22 在压缩文件中提取代码

```
bool GetCode (int& code)
{// 把压缩文件中的下一个代码取入 code
 // 如果没有代码，则返回 false
   unsigned char c, d;
   in.get(c);  // 输入8位
   if (in.eof()) return false;  // 没有代码

   // 检查上一次是否有剩余的位
   // 如果有，则取用其中的4位
   if (status) code = (LeftOver << ByteSize) | c;
   else {// 如果没有剩余的位，则需要另外 4 位
         in.get(d);  // 再取 8 位
         code = (c << excess) | (d >> excess);
         LeftOver = d & mask;}  // 保存余下的 4 位
   status = 1 - status;
   return true;
}
```

4. 解压缩

程序7-23给出了LZW解压器。压缩文件的第一个代码在 while 循环体外解码，而其他代码则在循环体内解码。因为压缩文件中的第一个代码总是在 0到alpha之间，因此它仅表示一个字符，并可以通过把整数转换成无符号字符得到。在每个 while循环的开始， s[size]中存有上次输出的解码文本的第一个字符。为了使第一个循环也满足此条件，可以把 size置为0且s[0]置为压缩文件中第一个代码所对应的唯一的一个字符。

while 循环体不停地从压缩文件中得到代码 ccode并对其解码。 ccode可能有以下两种情况：1) 在字典中。2) 不在字典中。当且仅当ccode < used时，ccode在字典中，其中ht[0:used]是 ht表的已定义部分。在这种情况下用函数Output和LZW规则解码，产生一新代码，其后缀是刚输出的与 ccode相对应的文本的第一个字母。当 ccode没有定义时，即本节开始所讨论的情况，ccode的代码是*text* (pcode)s[size] 。可根据此信息为code创建字典的入口并输出与其相对应的解码后的文本。

程序7-23 LZW解压器

```
void Decompress()
{// 解压缩一个压缩文件
  int used = alpha; //迄今所使用的代码

  // 输入并解压缩
  int pcode,  // 前一个代码
      ccode;  // 当前代码
  if (GetCode(pcode)){// 文件不为空
    s[0] = pcode;   // pcode 代表的字符
    out.put(s[0]);  // 输出pcode对应的串
    size = 0; // s[size] 是所输出的最后一个串的第一个字符

    while(GetCode(ccode)) {// 取另外的代码
      if (ccode < used) {// ccode已定义
        Output(ccode);
        if (used < codes) {// 创建新代码
          ht[used].prefix = pcode;
        ht[used++].suffix = s[size];}}
      else {// 特殊情况, 未定义代码
          ht[used].prefix = pcode;
          ht[used++].suffix = s[size];
          Output(ccode);}
      pcode = ccode;}
      }

  out.close();
  in.close();
}
```

5. 头文件和main函数

程序7-24给出LZW解压缩程序所包含的头文件、常量、类型定义、函数原型和 main函数。

程序7-24 解压缩程序的main函数

```
#include <fstream.h>
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

class element {
  friend void Decompress();
  friend void Output(int);
  private:
    int prefix;
    unsigned char suffix;
};
```

```
// 常量
const codes = 4096,  // 2^12
      ByteSize = 8,
      excess = 4,    // 12 - ByteSize
      alpha = 256,   // 2^ByteSize
      mask = 15;     // 2^excess - 1

// 全局变量
unsigned char s[codes];  // 用于重构文本
int size,                // 重构文本的大小
    LeftOver,            // 上一个代码的剩余位
    status = 0;          // 当且仅当没有剩余位时，其值为0
element ht[codes];       // 字典
ifstream in;
ofstream out;

void main ( int argc, char* argv[ ])
{
  SetFiles ( argc, argv );
  Decompress ( ) ;
}
```

练习

22. 用LZW压缩器产生的压缩文件是否有可能比原文件长呢?如果可能，长多少？

23. 为由以下字母{a, b, …z, 0, 1, …, 9, . , , , ; , : }和换行符组成的文件编写一个LZW压缩器和解压器。试测试程序的正确性。压缩文件是否可能比原文件长？

24. 重新修改LZW压缩和解压缩程序，使得每当压缩/解压缩1024x个字节后，重新初始化代码表。取文本文件长为100K到200K之间，x=10, 20, 30, 40 和50。测试修改后的程序。采用哪种x值的压缩效果最好？

7.6 参考及推荐读物

跳表是由William Pugh提出的。其平均复杂性的分析见论文 Skip lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33, 6, 1990, 668~676。

对于Lempel-Ziv压缩方法的描述是基于T.Welch的论文， A Technique for High-Performance Data Compression. *IEEE Computer*, 1994.6, 8-19。要想得到更好的压缩数据参见 D.Lelewer, D.Hirschberg. Data Compression. *ACM computing Surveys*, 19, 3, 1987, 261~296。