第9章

词典

借助数据结构来表示和组织的数字信息，可将所有数据视作一个整体统筹处理，进而提高信息访问的规范性及其处理的效率。例如，借助关键码直接查找和访问数据元素的形式，已为越来越多的数据结构所采用，这也成为现代数据结构的一个重要特征。

词典（dictionary）结构，即是其中最典型的例子。逻辑上的词典，是由一组数据构成的集合，其中各元素都是由关键码和数据项合成的词条（entry）。映射（map）结构与词典结构一样，也是词条的集合。二者的差别仅仅在于，映射要求不同词条的关键码互异，而词典则允许多个词条拥有相同的关键码[①]。除了静态查找，映射和词典都支持动态更新，二者统称作符号表（symbol table）。实际上，“是否允许雷同关键码”应从语义层面，而非ADT接口的层面予以界定，故本章将不再过分强调二者的差异，而是笼统地称作词典，并以跳转表和散列表为例，按照“允许雷同”和“禁止雷同”的语义，分别实现其统一的接口。

尽管此处词典和映射中的数据元素，仍表示和实现为词条形式，但这一做法并非必须。与第7章和第8章的搜索树相比，符号表并不要求词条之间能够根据关键码比较大小；与稍后第10章的优先级队列相比，其查找对象亦不仅限于最大或最小的词条。在符号表的内部，甚至也不需要按照大小次序来组织数据项——即便各数据项之间的确定义有某种次序。实际上，以散列表为代表的符号表结构，将转而依据数据项的数值，直接做逻辑查找和物理定位。也就是说，对于此类结构，在作为基本数据单位的词条内部，关键码（key）与数值（value）的地位等同，二者不必加以区分。此类结构所支持的这种新的数据访问方式，即所谓的循值访问（call-by-value）。相对于此前各种方式，这一方式更为自然，适用范围也更广泛。

有趣的是，对这种“新的”数据访问方式，在程序设计方面已有一定基础的读者，往往会或多或少地有些抵触的倾向；而刚刚涉足这一领域的读者，却反过来会有似曾相识的亲切之感，并更乐于接受。究其原因在于，循值访问方式与我们头脑中原本对数据集合组成的理解最为接近；不幸的是，在学习C/C++之类高级程序语言的过程中，我们思考问题的出发点和方向都已逐步被这些语言所同化并强化，而一些与生俱来的直觉与思路则逐渐为我们所淡忘。比如，在孩子们的头脑中，班级的概念只不过是同伴们的一组笑脸；随着学习内容的持续深入和思维方式的反复塑化，这一概念将逐渐被一组姓名所取代；甚至可能进而被抽象为一组学号。

既已抛开大小次序的概念，采用循值访问方式的计算过程，自然不再属于CBA式算法的范畴，此前关于CBA式算法下界的结论亦不再适用，比如在9.4节我们将看到，散列式排序算法将不再服从2.7节所给的复杂度下界。一条通往高效算法的崭新大道，由此在我们面前豁然展开。

当然，为支持循值访问的方式，在符号表的内部，仍然必须强制地在数据对象的数值与其物理地址之间建立某种关联。而所谓散列，正是在兼顾空间与时间效率的前提下，讨论和研究赖以设计并实现这种关联的一般性原则、技巧与方法，这些方面也是本章的核心与重点。

① 事实上，某些文献中所定义的词典和映射结构，可能与此约定恰好相反

§9.1 词典ADT

9.1.1 操作接口

除通用的接口之外，词典结构主要的操作接口可归纳为表9.1。

表9.1 词典ADT支持的标准操作接口

操 作 接 口	功 能 描 述
get(key)	若词典中存在以key为关键码的词条，则返回该词条的数据对象；否则，返回NULL
put(key, value)	插入词条(key, value)，并报告是否成功
remove(key)	若词典中存在以key为关键码的词条，则删除之并返回true；否则，返回false

实际上，包括Snobol4、MUMPS、SETL、Rexx、Awk、Perl、Ruby、PHP、Java和Python等在内，许多编程语言都以各自不同形式，支持类似于以上词典或映射ADT接口功能的基本数据结构，有的甚至将它们作为基本的数据类型，统称作关联数组（associative array）。

9.1.2 操作实例

比如，可如图9.1所示，将三国名将所对应的词条组织为一个词典结构。其中的每一词条，都由人物的字（style）和姓名（name）构成，分别作为词条的关键码和数据项。

图9.1 三国人物的词典结构

以初始包含关、张、马、黄四将的词典为例，在依次执行一系列操作的过程中，该词典结构内容的变化以及对应的输出如表9.2所示。

表9.2 词典结构操作实例

操 作	词 典 结 构	输 出
size()	("Yunchang", "Yu GUAN") ("Yide", "Fei ZHANG") ("Mengqi", "Chao MA") ("Hansheng", "Zhong HUANG")	4
put("Bofu", "Ce SUN")	("Yunchang", "Yu GUAN") ("Yide", "Fei ZHANG") ("Mengqi", "Chao MA") ("Hansheng", "Zhong HUANG") ("Bofu", "Ce SUN")	true
size()	[unchanged]	5
get("Yide")	[unchanged]	"Fei ZHANG"
get("Zilong")	[unchanged]	NULL

表9.2 词典结构操作实例（续）

操 作	词 典 结 构	输 出
put("Yide", "Fei CHANG")	("Yunchang", "Yu GUAN") ("Yide", "Fei CHANG") ("Mengqi", "Chao MA") ("Hansheng", "Zhong HUANG") ("Bofu", "Ce SUN")	true
size()	[unchanged]	5
get("Yide")	[unchanged]	"Fei CHANG"
remove("Mengqi")	("Yunchang", "Yu GUAN") ("Yide", "Fei CHANG") ("Hansheng", "Zhong HUANG") ("Bofu", "Ce SUN")	"Chao MA"
size()	[unchanged]	4

请特别留意以上第二次put()操作，其拟插入词条的关键码"Yide"，在该词典中已经存在。由该实例可见，插入效果等同于用新词条替换已有词条；相应地，put()操作也必然会成功。这一处理方式被包括Python和Perl在内的众多编程语言普遍采用，但本章采用的约定与此略有不同。跳转表将允许同时保留多个关键码雷同的词条，查找时任意返回其一；散列表则维持原词条不变，返回插入失败标志——也就是说，更接近于映射的规范。

9.1.3 接口定义

这里首先以如代码9.1所示模板类的形式定义词典的操作接口。

```
1 template <typename K, typename V> struct Dictionary { //词典Dictionary模板类
2    virtual int size() const = 0; //当前词条总数
3    virtual bool put ( K, V ) = 0; //插入词条（禁止雷同词条时可能失败）
4    virtual V* get ( K k ) = 0; //读取词条
5    virtual bool remove ( K k ) = 0; //删除词条
6 };
```

代码9.1 词典结构的操作接口规范

其中，所有操作接口均以虚函数形式给出，留待在派生类中予以具体实现。

另外，正如此前所述，尽管词条关键码类型可能支持大小比较，但这并非词典结构的必要条件，Dictionary模板类中的Entry类只需支持判等操作。

9.1.4 实现方法

不难发现，基于此前介绍的任何一种平衡二叉搜索树，都可便捷地实现词典结构。比如，Java语言的java.util.TreeMap类即是基于红黑树实现的词典结构。然而这类实现方式都在不经意中假设“关键码可以比较大小”，故其所实现的并非严格意义上的词典结构。

以下以跳转表和散列表为例介绍词典结构的两种实现方法。尽管它们都在底层引入了某种“序”，但这类“序”只是内部的一种约定；从外部接口来看，依然只有“相等”的概念。

§9.2 *跳转表

第2章所介绍的有序向量和第3章所介绍的有序列表，各有所长：前者便于静态查找，但动态维护成本较高；后者便于增量式的动态维护，但只能支持顺序查找。为结合二者的优点，同时弥补其不足，第7章和第8章逐步引入了平衡二叉搜索树，其查找、插入和删除操作均可在$\mathcal{O}(\log n)$时间内完成。尽管如此，这些结构的相关算法往往较为复杂，代码实现和调试的难度较大，其正确性、鲁棒性和可维护性也很难保证。

设计并引入跳转表[②]（skip list）结构的初衷，正是在于试图找到另外一种简便直观的方式，来完成这一任务。具体地，跳转表是一种高效的词典结构，它的定义与实现完全基于第3章的有序列表结构，其查询和维护操作在平均的意义下均仅需$\mathcal{O}(\log n)$时间。

9.2.1 Skiplist模板类

跳转表结构以模板类形式定义的接口，如代码9.2所示。

```
1  #include "../List/List.h" //引入列表
2  #include "../Entry/Entry.h" //引入词条
3  #include "Quadlist.h" //引入Quadlist
4  #include "../Dictionary/Dictionary.h" //引入词典
5
6  template <typename K, typename V> //key、value
7  //符合Dictionary接口的Skiplist模板类（但隐含假设元素之间可比较大小）
8  class Skiplist : public Dictionary<K, V>, public List<Quadlist<Entry<K, V>>*> {
9  protected:
10    bool skipSearch (
11       ListNode<Quadlist<Entry<K, V>>*>* &qlist,
12       QuadlistNode<Entry<K, V>>* &p,
13       K& k );
14 public:
15    int size() const { return empty() ? 0 : last()->data->size(); } //底层Quadlist的规模
16    int level() { return List::size(); } //层高 == #Quadlist，不一定要开放
17    bool put ( K, V ); //插入（注意与Map有别——Skiplist允许词条重复，故必然成功）
18    V* get ( K k ); //读取
19    bool remove ( K k ); //删除
20 };
```

代码9.2 Skiplist模板类

可见，借助多重继承（multiple inheritance）机制，由Dictionary和List共同派生而得的Skiplist模板类，同时具有这两种结构的特性；此外，这里还重写了在Dictionary抽象类（代码9.1）中，以虚函数形式定义的get()、put()和remove()等接口。

② 由W. Pugh于1989年发明[52]

9.2.2 总体逻辑结构

跳转表的宏观逻辑结构如图9.2所示。其内部由沿横向分层、沿纵向相互耦合的多个列表{ S_0, S_1, S_2, ..., S_h }组成，h称作跳转表的高度。

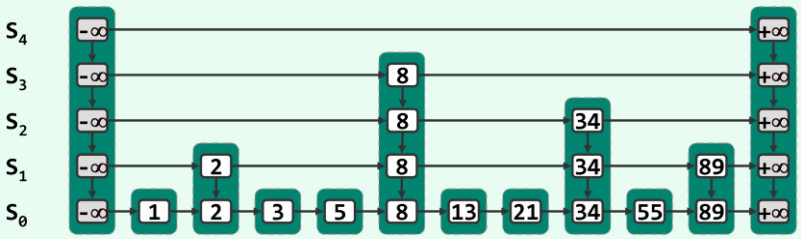


图9.2 跳转表的总体逻辑结构

每一水平列表称作一层（level），其中S_0和S_h分别称作底层（bottom）和顶层（top）。与通常的列表一样，同层节点之间可定义前驱与后继关系。为便于查找，同层节点都按关键码排序。需再次强调的是，这里的次序只是内部的一种约定；对外部而言，各词条之间仍然只需支持判等操作即可。为简化算法实现，每层列表都设有头、尾哨兵节点。

层次不同的节点可能沿纵向组成塔（tower），同一塔内的节点以高度为序也可定义前驱与后继关系。塔与词典中的词条一一对应。尽管塔内的节点相互重复，但正如随后将要看到的，这种重复不仅可以加速查找，而且只要策略得当，也不至造成空间的实质浪费。

高层列表总是低层列表的子集，其中特别地，S_0包含词典中的所有词条，而S_h除头、尾哨兵外不含任何实质的词条。不难看出，跳转表的层高h必然决定于最大的塔高。

9.2.3 四联表

按上述约定，跳转表内各节点沿水平和垂直方向都可定义前驱和后继，支持这种联接方式的表称作四联表（quadlist），它也是代码9.2中Skiplist模板类的底层实现方式。

■ Quadlist模板类

四联表结构可如代码9.3所示，以模板类的形式定义接口。

```
1  #include "QuadlistNode.h" //引入Quadlist节点类
2  template <typename T> class Quadlist { //Quadlist模板类
3  private:
4     int _size; QlistNodePosi(T) header; QlistNodePosi(T) trailer; //规模、头哨兵、尾哨兵
5  protected:
6     void init(); //Quadlist创建时的初始化
7     int clear(); //清除所有节点
8  public:
9  // 构造函数
10    Quadlist() { init(); } //默认
11 // 析构函数
12    ~Quadlist() { clear(); delete header; delete trailer; } //删除所有节点，释放哨兵
13 // 只读访问接口
```

```
14    Rank size() const { return _size; } //规模
15    bool empty() const { return _size <= 0; } //判空
16    QlistNodePosi(T) first() const { return header->succ; } //首节点位置
17    QlistNodePosi(T) last() const { return trailer->pred; } //末节点位置
18    bool valid ( QlistNodePosi(T) p ) //判断位置p是否对外合法
19    { return p && ( trailer != p ) && ( header != p ); }
20 // 可写访问接口
21    T remove ( QlistNodePosi(T) p ); //删除（合法）位置p处的节点，返回被删除节点的数值
22    QlistNodePosi(T) //将*e作为p的后继、b的上邻插入
23    insertAfterAbove ( T const& e, QlistNodePosi(T) p, QlistNodePosi(T) b = NULL );
24 // 遍历
25    void traverse ( void (* ) ( T& ) ); //遍历各节点，依次实施指定操作（函数指针，只读或局部修改）
26    template <typename VST> //操作器
27    void traverse ( VST& ); //遍历各节点，依次实施指定操作（函数对象，可全局性修改节点）
28 }; //Quadlist
```

代码9.3 Quadlist模板类

此处定义的接口包括：定位首节点、末节点，在全表或某一区间查找具有特定关键码的节点，删除特定节点，以及插入特定节点。通过它们的相互组合，即可实现跳转表相应的接口功能。

■ 四联表节点

作为四联表的基本组成元素，节点QuadlistNode模板类可定义如代码9.4所示。

```
1  #include "../Entry/Entry.h"
2  #define QlistNodePosi(T)  QuadlistNode<T>* //跳转表节点位置
3
4  template <typename T> struct QuadlistNode { //QuadlistNode模板类
5     T entry; //所存词条
6     QlistNodePosi(T) pred;  QlistNodePosi(T) succ; //前驱、后继
7     QlistNodePosi(T) above; QlistNodePosi(T) below; //上邻、下邻
8     QuadlistNode //构造器
9     ( T e = T(), QlistNodePosi(T) p = NULL, QlistNodePosi(T) s = NULL,
10      QlistNodePosi(T) a = NULL, QlistNodePosi(T) b = NULL )
11       : entry ( e ), pred ( p ), succ ( s ), above ( a ), below ( b ) {}
12    QlistNodePosi(T) insertAsSuccAbove //插入新节点，以当前节点为前驱，以节点b为下邻
13    ( T const& e, QlistNodePosi(T) b = NULL );
14 };
```

代码9.4 QuadlistNode模板类

为简化起见，这里并未做严格封装。对应于水平的前驱与后继，这里为每个节点设置了一对指针pred和succ；垂直方向的上邻和下邻则对应于above和below。主要的操作接口只有insertAsSuccAbove()，它负责创建新节点，并将其插入于当前节点之后、节点b之上。

■ 初始化与构造

由代码9.3可见，四联表的构造，实际上是通过调用如下init()函数完成的。

```
template <typename T> void Quadlist<T>::init() { //Quadlist初始化，创建Quadlist对象时统一调用
   header = new QuadlistNode<T>; //创建头哨兵节点
   trailer = new QuadlistNode<T>; //创建尾哨兵节点
   header->succ = trailer; header->pred = NULL; //沿横向联接哨兵
   trailer->pred = header; trailer->succ = NULL; //沿横向联接哨兵
   header->above = trailer->above = NULL; //纵向的后继置空
   header->below = trailer->below = NULL; //纵向的前驱置空
   _size = 0; //记录规模
} //如此构造的四联表，不含任何实质的节点，且暂时与其它四联表相互独立
```

代码9.5 Quadlist对象的创建

9.2.4 查找

查找是跳转表至关重要和最实质的操作，词条的插入和删除等其它操作均以之为基础，其实现效率也将直接影响到跳转表结构的整体性能。

■ get()

在跳转表中查找关键码k的具体过程，如代码9.6所示。

```
template <typename K, typename V> V* Skiplist<K, V>::get ( K k ) { //跳转表词条查找算法
   if ( empty() ) return NULL;
   ListNode<Quadlist<Entry<K, V>>*>* qlist = first(); //从顶层Quadlist的
   QuadlistNode<Entry<K, V>>* p = qlist->data->first(); //首节点开始
   return skipSearch ( qlist, p, k ) ? & ( p->entry.value ) : NULL; //查找并报告
} //有多个命中时靠后者优先
```

代码9.6 Skiplist::get()查找

■ skipSearch()

由上可见，实质的查找过程，只不过是从某层列表qlist的首节点first()出发，调用如代码9.7所示的内部函数skipSearch()。

```
/*****************************************************************************************
 * Skiplist词条查找算法（供内部调用）
 * 入口：qlist为顶层列表，p为qlist的首节点
 * 出口：若成功，p为命中关键码所属塔的顶部节点，qlist为p所属列表
 *       否则，p为所属塔的基座，该塔对应于不大于k的最大且最靠右关键码，qlist为空
 * 约定：多个词条命中时，沿四联表取最靠后者
 *****************************************************************************************/
template <typename K, typename V> bool Skiplist<K, V>::skipSearch (
   ListNode<Quadlist<Entry<K, V>>*>* &qlist, //从指定层qlist的
   QuadlistNode<Entry<K, V>>* &p, //首节点p出发
   K& k ) { //向右、向下查找目标关键码k
```

```
12    while ( true ) { //在每一层
13       while ( p->succ && ( p->entry.key <= k ) ) //从前向后查找
14          p = p->succ; //直到出现更大的key或溢出至trailer
15       p = p->pred; //此时倒回一步，即可判断是否
16       if ( p->pred && ( k == p->entry.key ) ) return true; //命中
17       qlist = qlist->succ; //否则转入下一层
18       if ( !qlist->succ ) return false; //若已到穿透底层，则意味着失败
19       p = ( p->pred ) ? p->below : qlist->data->first(); //否则转至当前塔的下一节点
20    }  //课后：通过实验统计，验证关于平均查找长度的结论
21 }
```

代码9.7 Skiplist::skipSearch()查找

这里利用参数p和qlist，分别指示命中关键码所属塔的顶部节点，及其所属的列表。qlist和p的初始值分别为顶层列表及其首节点，返回后它们将为上层的查找操作提供必要的信息。

■ **实例**

仍以图9.2为例，针对关键码21的查找经过节点{ -∞, -∞, 8, 8, 8, 8, 13 }，最终抵达21后报告成功；针对关键码34的查找经过节点{ -∞, -∞, 8, 8 }，最终抵达34后报告成功；针对关键码1的查找经过节点{ -∞, -∞, -∞, -∞, -∞ }，最终抵达1后报告成功。而针对关键码80的查找经过节点{ -∞, -∞, 8, 8, 34, 34, 34, 55 }，最终抵达89后报告失败；针对关键码0的查找经过节点{ -∞, -∞, -∞, -∞, -∞ }，最终抵达1后报告失败；针对关键码99的查找经过节点{ -∞, -∞, 8, 8, 34, 34, 89 }，最终抵达+∞后报告失败。

9.2.5 空间复杂度

■ **“生长概率逐层减半”条件**

不难理解，其中各塔高度的随机分布规律（如最大值、平均值等），对跳转表的总体性能至关重要。反之，若不就此作出显式的限定，则跳转表的时间和空间效率都难以保证。

比如，若将最大塔高（亦即跳转表的层高）记作h，则在极端情况下，每个词条所对应塔的高度均有可能接近甚至达到h。果真如此，在查找及更新过程中需要访问的节点数量将难以控制，时间效率注定会十分低下。同时，若词条总数为n，则在此类情况下，跳转表所需的存储空间量也将高达Ω(nh)。

然而幸运的是，若能采用简明而精妙的策略，控制跳转表的生长过程，则在时间和空间方面都可实现足够高的效率。就效果而言，此类控制策略必须满足所谓“生长概率逐层减半”条件：

对于任意0 ≤ k < h，S_k中任一节点在S_{k+1}中依然出现的概率，始终为1/2

也就是说，S_0中任一关键码依然在S_k中出现的概率，等于2^{-k}。这也可等效地理解和模拟为，在各塔自底而上逐层生长的过程中，通过投掷正反面等概率的理想硬币（fair coin），来决定是否继续增长一层——亦即，对应于当前的词条，是否在上一层列表中再插入一个节点。

那么，在插入词条的过程中，应该如何从技术上保证这一条件始终成立呢？具体的方法稍后将在9.2.7节介绍，目前不妨暂且假定这一条件的确成立。

■ 节点总数的期望值

根据数学归纳法，“生长概率逐层减半”条件同时也意味着，列表S_0中任一节点在列表S_k中依然出现的概率均为$1/2^k = 2^{-k}$。因此，第k层列表所含节点的期望数目为：

$$E(|S_k|) = n \times 2^{-k}$$

亦即，各层列表的规模将随高度上升以50%的比率迅速缩小，故空间总体消耗量的期望值应为：

$$E(\Sigma_k|S_k|) = \Sigma_k E(|S_k|) = n \times (\Sigma_k 2^{-k}) < 2n = \mathcal{O}(n)$$

9.2.6 时间复杂度

在由多层四联表组成的跳转表中进行查找，需访问的节点数目是否会实质性地增加？由以上代码9.7中查找算法skipSearch()可见，单次纵向或横向跳转本身只需常数时间，故查找所需的时间应取决于横向、纵向跳转的总次数。那么，是否会因层次过多而导致横向或纵向的跳转过多呢？以下从概率的角度，分别对其平均性能做出估计，并说明其期望值均不超过$\mathcal{O}(\log n)$。

■ 期望高度与纵向跳转次数

考查第k层列表S_k。

S_k非空，当且仅当S_0所含的n个节点中，至少有一个会出现在S_k中，相应的概率应为：

$$Pr(|S_k| > 0) \le n \times 2^{-k} = n/2^k$$

反过来，S_k为空的概率即为：

$$Pr(|S_k| = 0) \ge 1 - n/2^k$$

可以看出，这一概率将随着高度k的增加，而迅速上升并接近100%。

以第$k = 3 \cdot \log n$层为例。该层列表S_k为空，当且仅当$h < k$，对应的概率为：

$$Pr(h < k) = Pr(|S_k| = 0) \ge 1 - n/2^k = 1 - n/n^3 = 1 - 1/n^2$$

一般地，$k = a \cdot \log n$层列表为空的概率为$1 - 1/n^{a-1}$，$a > 3$后这一概率将迅速地接近100%。这意味着跳转表的高度h有极大的可能不会超过 $3 \cdot \log n$，h的期望值应为：

$$E(h) = \mathcal{O}(\log n)$$

按照代码9.7的skipSearch()算法，查找过程中的跳转只能向右或向下（而不能向左倒退或向上爬升），故活跃节点的高度必单调非增，每一高度上的纵向跳转至多一次。因此，整个查找过程中消耗于纵向跳转的期望时间不超过跳转表高度h的期望值$\mathcal{O}(\log n)$。

■ 横向跳转

skipSearch()算法中的内循环对应于沿同一列表的横向跳转，且此类跳转在同一高度可做多次。那么，横向跳转与上述纵向跳转的这一差异，是否意味着这方面的时间消耗将不受跳转表高度h的控制，并进而对整体的查找时间产生实质性影响？答案是否定的。

进一步观察skipSearch()算法可知，沿同一列表的横向跳转所经过的节点必然依次紧邻，而且它们都应该是各自所属塔的塔顶。若将同层连续横向跳转的次数记作Y，则对于任意的$0 \le k$，Y取值为k对应于“k个塔顶再加最后一个非塔顶”联合事件，故其概率应为：

$$Pr(Y = k) = (1 - p)^k \cdot p$$

这是一个典型的几何分布（geometric distribution），其中$p = 1/2$是塔继续生长的概率。因此，Y的期望值应为：

$$E(Y) = (1 - p) / p = (1 - 1/2) / (1/2) = 1$$

也就是说，在同一高度上，彼此紧邻的塔顶节点数目的期望值为1 + 1 = 2；沿着每条查找路径，在每一高度上平均只做常数次横向跳转。因此，整个查找过程中所做横向跳转的期望次数，应依然线性正比于跳转表的期望高度，亦即$O(\log n)$。

■ 其它

除以上纵向和横向跳转，skipSearch()还涉及其它一些操作，但总量亦不超过$O(\log n)$。比如，内层while循环尽管必终止于失败节点（key更大或溢出至trailer），但此类节点在每层至多一个，访问它们所需的时间总量仍不超过跳转表的期望高度$E(h) = O(\log n)$。

9.2.7 插入

■ put()

将词条(k, v)插入跳转表的具体操作过程，可描述和实现如代码9.8所示。

```
1  template <typename K, typename V> bool Skiplist<K, V>::put ( K k, V v ) { //跳转表词条插入算法
2     Entry<K, V> e = Entry<K, V> ( k, v ); //待插入的词条（将被随机地插入多个副本）
3     if ( empty() ) insertAsFirst ( new Quadlist<Entry<K, V>> ); //插入首个Entry
4     ListNode<Quadlist<Entry<K, V>>*>* qlist = first(); //从顶层四联表的
5     QuadlistNode<Entry<K, V>>* p = qlist->data->first(); //首节点出发
6     if ( skipSearch ( qlist, p, k ) ) //查找适当的插入位置（不大于关键码k的最后一个节点p）
7        while ( p->below ) p = p->below; //若已有雷同词条，则需强制转到塔底
8     qlist = last(); //以下，紧邻于p的右侧，一座新塔将自底而上逐层生长
9     QuadlistNode<Entry<K, V>>* b = qlist->data->insertAfterAbove ( e, p ); //新节点b即新塔基座
10    while ( rand() & 1 ) { //经投掷硬币，若确定新塔需要再长高一层，则
11       while ( qlist->data->valid ( p ) && !p->above ) p = p->pred; //找出不低于此高度的最近前驱
12       if ( !qlist->data->valid ( p ) ) { //若该前驱是header
13          if ( qlist == first() ) //且当前已是最顶层，则意味着必须
14             insertAsFirst ( new Quadlist<Entry<K, V>> ); //首先创建新的一层，然后
15          p = qlist->pred->data->first()->pred; //将p转至上一层Skiplist的header
16       } else //否则，可径自
17          p = p->above; //将p提升至该高度
18       qlist = qlist->pred; //上升一层，并在该层
19       b = qlist->data->insertAfterAbove ( e, p, b ); //将新节点插入p之后、b之上
20    }//课后：调整随机参数，观察总体层高的相应变化
21    return true; //Dictionary允许重复元素，故插入必成功——这与Hashtable等Map略有差异
22 }
```

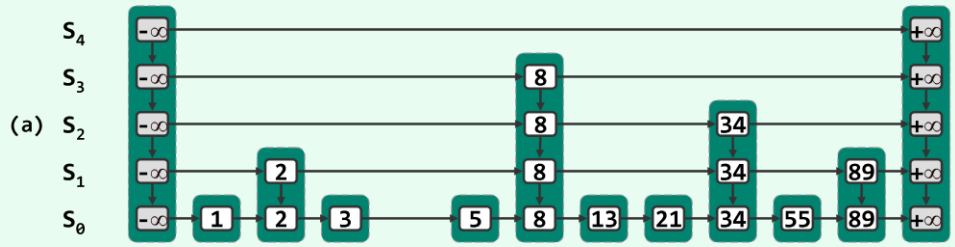
代码9.8 Skiplist::put()插入

这里通过逻辑表达式“rand() % 2”来模拟投掷硬币，并保证“生长概率逐层减半”条件。也就是说，通过（伪）随机整数的奇偶，近似地模拟一次理想的掷硬币实验。只要（伪）随机数为奇数（等价于掷出硬币正面），新塔就继续生长；一旦取（伪）随机数为偶数（等价于掷出反面），循环随即终止（生长停止），整个插入操作亦告完成。

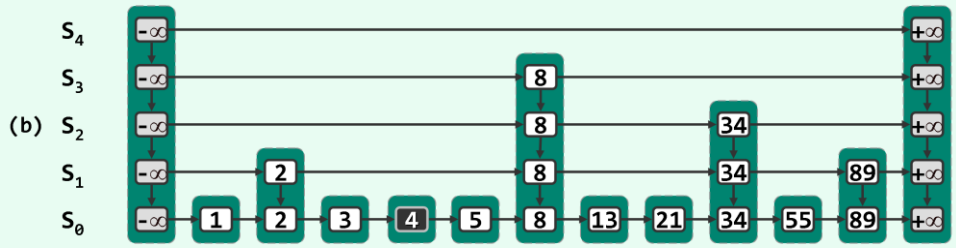
由此可见，新塔最终的（期望）高度，将取决于此前连续的正面硬币事件的（期望）次数。

■ 实例

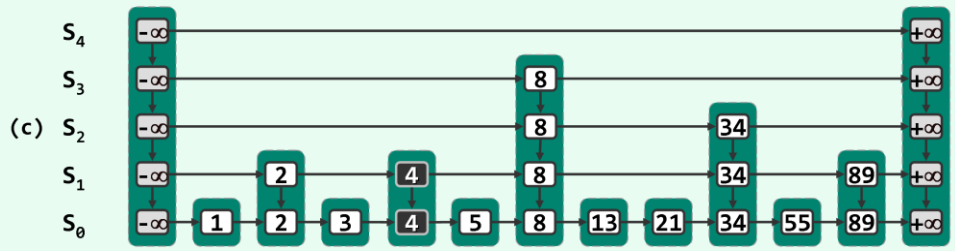
考查如图9.2所示的跳转表。将关键码4插入其中的过程，如图9.3(a~d)所示。



首先如图(a)所示，经过查找确定，应紧邻于关键码3右侧实施插入。



然后如图(b)所示，在底层列表中，创建一个节点作为新塔的基座。



此后，假定随后掷硬币的过程中，前两次为正面，第三次为反面。于是如图(c)和(d)所示，新塔将连续长高两层后停止生长。

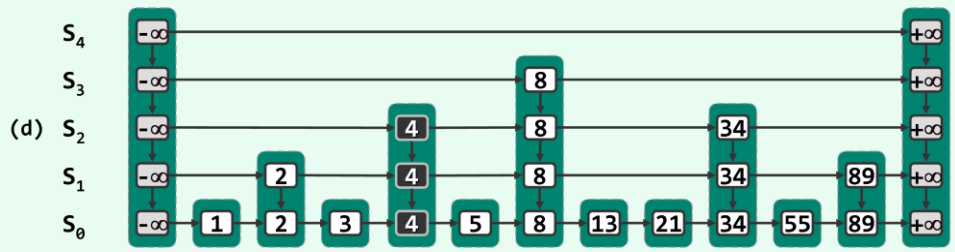


图9.3 跳转表节点插入过程(a~d)，也是节点删除的逆过程(d~a)

新塔每长高一层，塔顶节点除须与原塔纵向联接，还须与所在列表中的前驱与后继横向联接。

■ insertAfterAbove()

可见，QuadlistNode节点总是以塔为单位自底而上地成批插入，且每一节点都是作为当时的新塔顶而插入。也就是说，QuadlistNode节点的插入都属于同一固定的模式：创建关键码为e的新节点，将其作为节点p的后继和节点b（当前塔顶）的上邻“植入”跳转表。

因此，代码9.3只需提供统一的接口insertAfterAbove()，其具体实现如代码9.9所示。

```
1 template <typename T> QlistNodePosi(T) //将e作为p的后继、b的上邻插入Quadlist
2 Quadlist<T>::insertAfterAbove ( T const& e, QlistNodePosi(T) p, QlistNodePosi(T) b = NULL )
3 {  _size++; return p->insertAsSuccAbove ( e, b );  } //返回新节点位置（below = NULL）
```

代码9.9 Quadlist::insertAfterAbove()插入

■ insertAsSuccAbove()

上述接口的实现，需转而调用节点p的insertAsSuccAbove()接口，如代码9.10所示完成节点插入的一系列实质性操作。

```
1 template <typename T> QlistNodePosi(T) //将e作为当前节点的后继、b的上邻插入Quadlist
2 QuadlistNode<T>::insertAsSuccAbove ( T const& e, QlistNodePosi(T) b = NULL ) {
3    QlistNodePosi(T) x = new QuadlistNode<T> ( e, this, succ, NULL, b ); //创建新节点
4    succ->pred = x; succ = x; //设置水平逆向链接
5    if ( b ) b->above = x; //设置垂直逆向链接
6    return x; //返回新节点的位置
7 }
```

代码9.10 QuadlistNode::insertAsSuccAbove()插入

具体过程如图9.4(a)所示，插入前节点b的上邻总是为空。

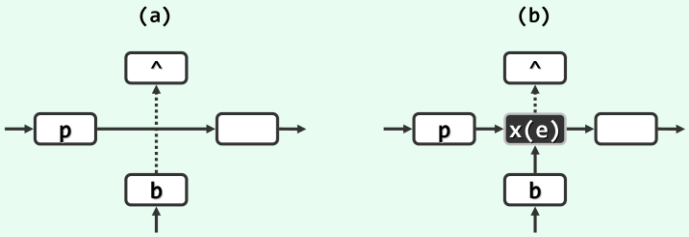


图9.4 四联表节点插入过程

首先，创建一个关键码为e的节点，其前驱和后继分别设为当前节点（p）及其后继（p->succ），上邻和下邻分别设为NULL和节点b。然后，沿水平和垂直方向设置好逆向的链接。最终结果如图(b)所示。

因这里允许关键码雷同，故在插入之前无需查找确认是否已有某个词条的关键码为e。

■ 时间复杂度

新塔每长高一层，都要紧邻于该层的某一节点p之后创建新的塔顶节点。准确地，在不大于新关键码的所有节点中，节点p为最大者；若这样的节点有多个，则按约定，p应取其中最靠后者。然而，若在每一层都从首节点开始，通过扫描确认p的位置，则最坏情况下可能每一层四联表都几乎需要遍历，耗时量将高达$\Omega(n)$。然而实际上，各层四联表中的位置p之间自底而上存在很强的关联性，利用这一性质即可保证高效而精准地确定各高度上的插入位置p。

具体地如代码9.8所示，每次都从当前节点p的前驱出发，先上升一层，然后自右向左依次移动，直到发现新节点在新高度上的前驱。接下来，只需将p更新为该前驱的位置，并将新塔顶节点插入于p之后，新塔顶节点的插入即告完成。实际上，新塔每增长一层，都可重复上述过程完成新塔顶节点的插入。

整个过程中p所经过的路径，与关键码的查找路径恰好方向相反。由9.2.6节的结论，被访问节点的期望总数不超过$\mathcal{O}(\log n)$，因此这也是插入算法运行时间期望值的上界。

9.2.8 删除

■ **Skiplist::remove()**

从跳转表中删除关键码为k词条的具体操作过程，如描述为代码9.11。

```
1  template <typename K, typename V> bool Skiplist<K, V>::remove ( K k ) { //跳转表词条删除算法
2     if ( empty() ) return false; //空表情况
3     ListNode<Quadlist<Entry<K, V>>*>* qlist = first(); //从顶层Quadlist的
4     QuadlistNode<Entry<K, V>>* p = qlist->data->first(); //首节点开始
5     if ( !skipSearch ( qlist, p, k ) ) return false; //目标词条不存在，直接返回
6     do { //若目标词条存在，则逐层拆除与之对应的塔
7        QuadlistNode<Entry<K, V>>* lower = p->below; //记住下一层节点，并
8        qlist->data->remove ( p ); //删除当前层节点，再
9        p = lower; qlist = qlist->succ; //转入下一层
10    } while ( qlist->succ ); //如上不断重复，直到塔基
11    while ( !empty() && first()->data->empty() ) //逐一地
12       List::remove ( first() ); //清除已可能不含词条的顶层Quadlist
13    return true; //删除操作成功完成
14 }
```

代码9.11 Skiplist::remove()删除

这一过程的次序，与插入恰好相反。以如图9.3(d)所示的跳转表为例，若欲从其中删除关键码为4的词条，则在查找定位该词条后，依次删除塔顶。关键码删除过程的中间结果如图(c)和(b)所示，最终结果如图(a)。

■ **Quadlist::remove()**

在基于四联表实现跳转表中，QuadlistNode节点总是以塔为单位，自顶而下地成批被删除，其中每一节点的删除，都按照如下固定模式进行：节点p为当前的塔顶，将它从所属横向列表中删除；其下邻（若存在）随后将成为新塔顶，并将在紧随其后的下一次删除操作中被删除。

Quadlist模板类（代码9.3）为此定义了接口remove()，其具体实现如代码9.12所示。

```
1  template <typename T> //删除Quadlist内位置p处的节点，返回其中存放的词条
2  T Quadlist<T>::remove ( QlistNodePosi(T) p ) { //assert: p为Quadlist中的合法位置
3     p->pred->succ = p->succ; p->succ->pred = p->pred; _size--;//摘除节点
4     T e = p->entry; delete p; //备份词条，释放节点
5     return e; //返回词条
6  }
7
8  template <typename T> int Quadlist<T>::clear() { //清空Quadlist
9     int oldSize = _size;
10    while ( 0 < _size ) remove ( header->succ ); //逐个删除所有节点
11    return oldSize;
12 }
```

代码9.12 Quadlist::remove()删除

这里各步迭代中的操作次序，与图9.4(a)和(b)基本相反。略微不同之处在于，因必然是整塔删除，故可省略纵向链接的调整。

其中clear()接口用以删除表中所有节点，在代码9.3中也是析构过程中的主要操作。

■ **时间复杂度**

如代码9.11所示，词条删除算法所需的时间，不外乎消耗于两个方面。

首先是查找目标关键码，由9.2.6节的结论可知，这部分时间的期望值不过$\mathcal{O}(\log n)$。其次是拆除与目标关键码相对应的塔，这是一个自顶而下逐层迭代的过程，故累计不超过h步；另外，由代码9.12可见，各层对应节点的删除仅需常数时间。

综合以上分析可知，跳转表词条删除操作所需的时间不超过$\mathcal{O}(h) = \mathcal{O}(\log n)$。

§9.3 散列表

散列作为一种思想既朴素亦深刻，作为一种技术则虽古老却亦不失生命力，因而在数据结构及算法中占据独特而重要地位。此类方法以最基本的向量作为底层支撑结构，通过适当的散列函数在词条的关键码与向量单元的秩之间建立起映射关系。理论分析和实验统计均表明，只要散列表、散列函数以及冲突排解策略设计得当，散列技术可在期望的常数时间内实现词典的所有接口操作。也就是说，就平均时间复杂度的意义而言，可以使这些操作所需的运行时间与词典的规模基本无关。尤为重要的是，散列技术完全摒弃了“关键码有序”的先决条件，故就实现词典结构而言，散列所特有的通用性和灵活性是其它方式无法比拟的。

以下将围绕散列表、散列函数以及冲突排解三个主题，逐层深入地展开介绍。

9.3.1 完美散列

■ **散列表**

散列表（hashtable）是散列方法的底层基础，逻辑上由一系列可存放词条（或其引用）的单元组成，故这些单元也称作桶（bucket）或桶单元；与之对应地，各桶单元也应按其逻辑次序在物理上连续排列。因此，这种线性的底层结构用向量来实现再自然不过。为简化实现并进一步提高效率，往往直接使用数组，此时的散列表亦称作桶数组（bucket array）。若桶数组的容量为R，则其中合法秩的区间[0, R)也称作地址空间（address space）。

■ **散列函数**

一组词条在散列表内部的具体分布，取决于所谓的散列（hashing）方案——事先在词条与桶地址之间约定的某种映射关系，可描述为从关键码空间到桶数组地址空间的函数：

$$\text{hash}() : \text{key} \rightarrow \text{hash}(\text{key})$$

这里的hash()称作散列函数（hash function）。反过来，hash(key)也称作key的散列地址（hashing address），亦即与关键码key相对应的桶在散列表中的秩。

■ **实例**

以学籍库为例。若某高校2011级共计4000名学生的学号为2011-0000至2011-3999，则可直接使用一个长度为4000的散列表A[0~3999]，并取

$$\text{hash}(\text{key}) = \text{key} - 20110000$$

从而将学号为x的学生学籍词条存放于桶单元A[hash(x)]。

如此散列之后，根据任一合法学号，都可在$O(1)$时间内确定其散列地址，并完成一次查找、插入或删除。空间性能方面，每个桶恰好存放一个学生的学籍词条，既无空余亦无重复。这种在时间和空间性能方面均达到最优的散列，也称作完美散列（perfect hashing）。

实际上，Bitmap结构（习题[2-34]）也可理解为完美散列的一个实例。其中，为每个可能出现的非负整数，各分配了一个比特位，作为判定它是否属于当前集合的依据；散列函数也再简单不过——各比特位在内部向量中的秩，就是其所对应整数的数值。

遗憾的是，以上实例都是在十分特定的条件下才成立的，完美散列实际上并不常见。而在更多的应用环境中，为兼顾空间和时间效率，无论散列表或散列函数都需要经过更为精心的设计。以下就是一个更具一般性的实例。

9.3.2 装填因子与空间利用率

■ 电话查询系统

假设某大学拟建立一个电话簿查询系统，覆盖教职员工和学生所使用的共约25000门固定电话。以下，主要考查其中反查功能的实现，即如何高效地由电话号码获取机主的信息。

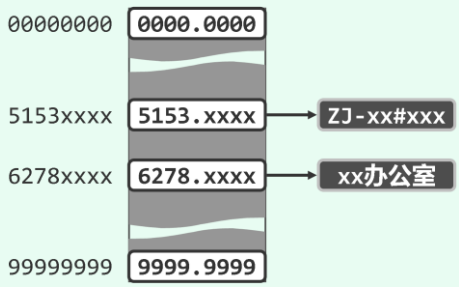


图9.5 直接使用线性数组实现电话簿词典

这一任务从数据结构的角度可理解为，设计并实现一个词典结构，以电话号码为词条关键码，支持根据这种关键码的高效查询。若考虑到开机、撤机和转机等情况，还应支持词条的插入和删除等动态操作。仿照学籍库的例子，可如图9.5引入向量，将电话号码为x 的词条存放在秩为x的单元。如此，不仅词条与桶单元一一对应，而且无论是静态的查找还是动态的插入和删除，每次操作仅需常数时间！

然而进一步分析之后不难发现，这一方案在此情况下并不现实。从理论上讲，在使用8位编号系统时，整个城市固定电话最多可能达到10^8门。尽管该校人员所涉及的固定电话仅有25000门，但号码却可能随机分布在[0000-0000, 9999-9999]的整个范围内。这就意味着，上述方案所使用数组的长度大致应与10^8相当。每个词条占用的空间即便按100字节估计，该数组也至少需要占用10GB的空间。也就是说，此时的空间有效利用率仅为 25000 / 10^8 = 0.025%，绝大部分的空间实际上处于闲置状态。

■ IP节点查询

另一个类似的例子是，根据IP地址获取对应的域名信息。按照32bit地址的协议，理论上可能的IP地址共有$2^{32} = 4\times10^9$个，故此时若直接套用以上方法采用最简单的散列表和散列函数，将动辄征用100~1000GB的空间。另一方面，尽管大多数IP并没有指定域名，但任一IP都有可能具有域名，故这种方法的空间利用率也仅为5%左右[③]。而在未来采用IPv6协议之后，尽管实际运

[③] 据威瑞信（VeriSign）公司2010年11月发布的《2010年第三季度域名行业报告》，截至2010年第三季度底，全球顶级域名（Top Level Domain, TLD）的注册总数已达到2.02亿，平均约每20个IP中才有一个IP具有域名

行中的节点数目在短时间内不会有很大的变化，但允许使用的IP地址将多达2^128 = 256×10^36个——如此庞大的地址空间根本无法直接使用数组表示和存放[④]；即便有如此规模的存储介质，其空间利用率依然极低。

■ 兼顾空间利用率与速度

此类问题在实际应用中十分常见，其共同的特点可归纳为：尽管词典中实际需要保存的词条数N（比如25000门）远远少于可能出现的词条数R（10^8门），但R个词条中的任何一个都有可能出现在词典中。仿照2.4.1节针对向量空间利用率的度量方法，这里也可以将散列表中非空桶的数目与桶单元总数的比值称作装填因子（load factor）。从这一角度来看，上述问题的实质在于散列表的装填因子太小，从而导致空间利用率过低。

无论如何，散列方法的查找和更新速度实在诱人，也的确可以完美地适用于学籍库之类的应用。那么，能否在保持优势的前提下，克服其在存储空间利用率方面的不足呢？答案是肯定的，但需要运用一系列的技巧，其中首先就是散列函数的设计。

9.3.3 散列函数

9.3.10节将介绍一般类型关键码到整数的转换方法，故不妨先假定关键码均为[0, R)范围内的整数。将词典中的词条数记作N，散列表长度记作M，于是通常有：

R >> M > N

如图9.6所示，散列函数hash()的作用可理解为，将关键码空间[0, R)压缩为散列地址空间[0, M)。

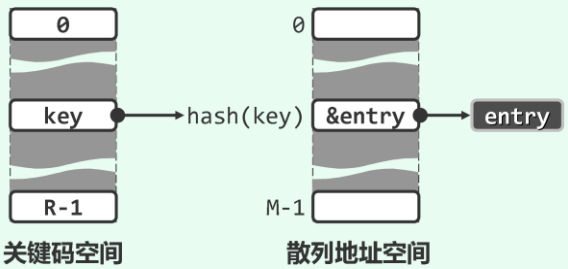


图9.6 散列函数

■ 设计原则

作为好的散列函数，hash()应具备哪些条件呢？首先，必须具有确定性。无论所含的数据项如何，词条E在散列表中的映射地址hash(E.key)必须完全取决于其关键码E.key。其次，映射过程自身不能过于复杂，唯此方能保证散列地址的计算可快速完成，从而保证查询或修改操作整体的O(1)期望执行时间。再次，所有关键码经映射后应尽量覆盖整个地址空间[0, M)，唯此方可充分利用有限的散列表空间。也就是说，函数hash()最好是满射。

当然，因定义域规模R远远大于取值域规模M，hash()不可能是单射。这就意味着，关键码不同的词条被映射到同一散列地址的情况——称作散列冲突（collision）——难以彻底避免。尽管9.3.5节将会介绍解决冲突的办法，但若能在设计和选择散列函数阶段提前做些细致而充分的考量，便能尽可能地降低冲突发生的概率。

在此，最为重要的一条原则就是，关键码映射到各桶的概率应尽量接近于1/M——若关键码均匀且独立地随机分布，这也是任意一对关键码相互冲突的概率。就整体而言，这等效于将关键码空间“均匀地”映射到散列地址空间，从而避免导致极端低效的情况——比如，因大部分关键

[④] 截至2010年，人类拥有的数字化数据总量为1.2ZB（1ZB = 2^70 = 10^21字节），预计到2020年可达35ZB

码集中分布于某一区间，而加剧散列冲突；或者反过来，因某一区间仅映射有少量的关键码，而导致空间利用率低下。

总而言之，随机越强、规律性越弱的散列函数越好。当然，完全符合上述条件的散列函数并不存在，我们只能通过先验地消除可能导致关键码分布不均匀的因素，最大限度地模拟理想的随机函数，尽最大可能降低发生冲突的概率。

■ **除余法（division method）**

符合上述要求的一种最简单的映射办法，就是将散列表长度M取作为素数，并将关键码key映射至key关于M整除的余数：

```
hash(key)  =  key mod M
```

仍以校园电话簿为例，若取M = 90001，则以下关键码：

```
{ 6278-5001, 5153-1876, 6277-0211 }
```

将如图9.7所示分别映射至

```
{ 54304, 51304, 39514 }
```

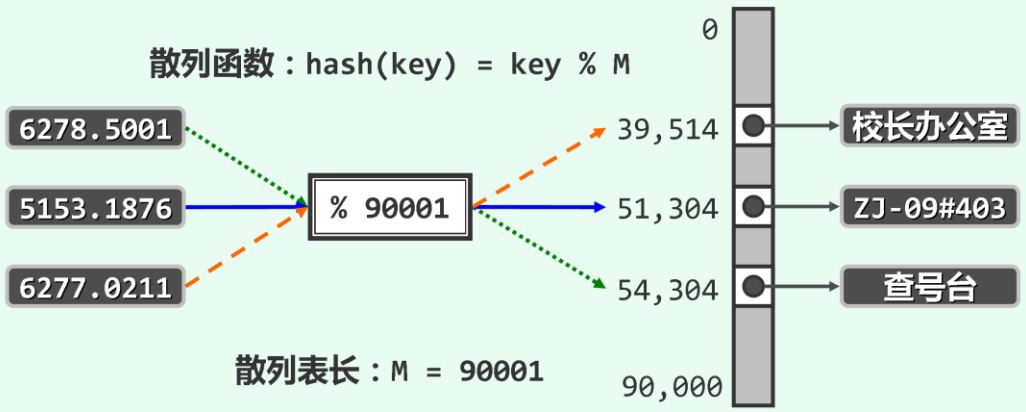


图9.7 除余法

请注意，采用除余法时必须将M选作素数，否则关键码被映射至[0, M)范围内的均匀度将大幅降低，发生冲突的概率将随M所含素因子的增多而迅速加大。

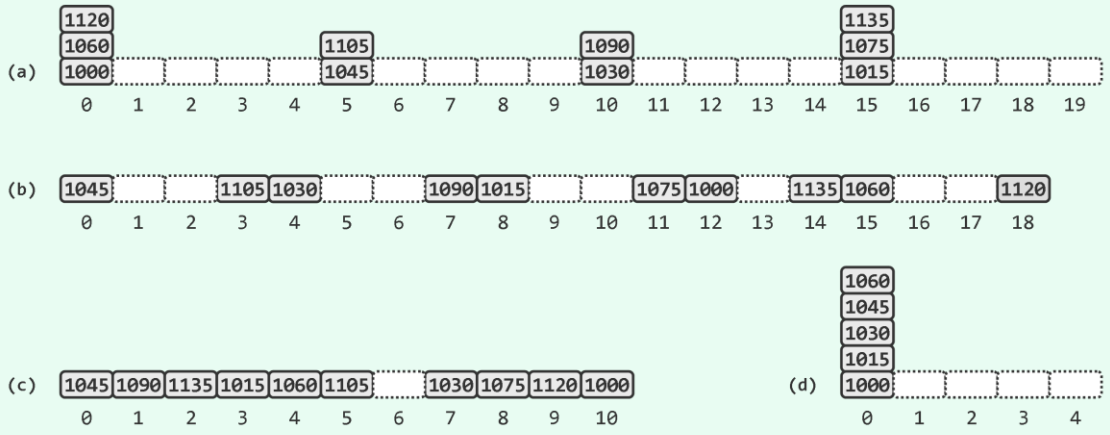


图9.8 素数表长可降低冲突的概率并提高空间的利用率

在实际应用中，对同一词典内词条的访问往往具有某种周期性，若其周期与M具有公共的素因子，则冲突的概率将急剧攀升。试考查一例：某散列表从全空的初始状态开始，插入的前10个词条对应的关键码是等差数列{ 1000, 1015, 1030, ..., 1135 }。

如图9.8(a)所示，若散列表长度取作M = 20，则其中每一关键码，都与另外一或两个关键码相冲突；而反过来，散列表中80%的桶，此时却处于空闲状态。

词条集中到散列表内少数若干桶中（或附近）的现象，称作词条的聚集（clustering）。显然，好的散列函数应尽可能此类现象，而采用素数表长则是降低聚集发生概率的捷径。

一般地，散列表的长度M与词条关键码间隔T之间的最大公约数越大，发生冲突的可能性也将越大（习题[9-6]）。因此，若M取素数，则简便对于严格或大致等间隔的关键码序列，也不致出现冲突激增的情况，同时提高空间效率。

比如若改用表长M = 19，则如图9.8(b)所示没有任何冲突，且空间利用率提高至50%以上。再如，若如图9.8(c)所示取表长M = 11，则同样不致发生任何冲突，且仅有一个桶空闲。

当然，若T本身足够大而且恰好可被M整除，则所有被访问词条都将相互冲突。例如，若如图9.8(d)所示将表长取作素数M = 5且只考虑原插入序列中的前5个关键码，则所有关键码都将聚集于一个桶内。不难理解，相对而言，发生这种情况的概率极低。

■ MAD法（multiply-add-divide method）

以素数为表长的除余法尽管可在一定程度上保证词条的均匀分布，但从关键码空间到散列地址空间映射的角度看，依然残留有某种连续性。比如，相邻关键码所对应的散列地址，总是彼此相邻；极小的关键码，通常都被集中映射到散列表的起始区段——其中特别地，0值居然是一个“不动点”，其散列地址总是0，而与散列表长度无关。

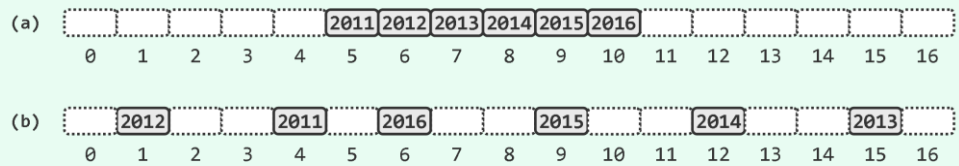


图9.9 MAD法可消除散列过程的连续性

例如，在如图9.9(a)所示，将关键码：

{ 2011, 2012, 2013, 2014, 2015, 2016 }

插入长度为M = 17的空散列表后，这组词条将存放至地址连续的6个桶中。尽管这里没有任何关键码的冲突，却具有就“更高阶”的均匀性。

为弥补这一不足，可采用所谓的MAD法将关键码key映射为：

(a × key + b) mod M，其中M仍为素数，a > 0，b > 0，且a mod M ≠ 0

此类散列函数需依次执行乘法、加法和除法（模余）运算，故此得名。

尽管运算量略有增加，但只要常数a和b选取得当，MAD法可以很好地克服除余法原有的连续性缺陷。仍以上述插入序列为例，当取a = 31和b = 2时，按MAD法的散列结果将图9.9(b)所示，各关键码散列的均匀性相对于图9.9(a)有了很大的改善。

实际上，此前所介绍的除余法，也可以看做是MAD法取a = 1和b = 0的特殊情况。从这一角度来看，导致除余法连续性缺陷的根源，也可理解为这两个常数未发挥实质的作用。

■ 更多的散列函数

散列函数种类繁多，不一而足。数字分析法（selecting digits）从关键码key特定进制的展开中抽取出特定的若干位，构成一个整型地址。比如，若取十进制展开中的奇数位，则有

hash(123456789) = 13579

又如所谓平方取中法（mid-square），从关键码key的平方的十进制或二进制展开中取居中的若干位，构成一个整型地址。比如，若取平方后十进制展开中居中的三位，则有

hash(123) = 15129 = 512

hash(1234567) = 1524155677489 = 556

再如所谓折叠法（folding），是将关键码的十进制或二进制展开分割成等宽的若干段，取其总和作为散列地址。比如，若以三个数位为分割单位，则有

hash(123456789) = 123 + 456 + 789 = 1368

分割后各区段的方向也可以是往复折返式的，比如

hash(123456789) = 123 + 654 + 789 = 1566

还有如所谓位异或法（xor），是将关键码的二进制展开分割成等宽的若干段，经异或运算得到散列地址。比如，仍以三个数位为分割单位，则有

hash(411) = hash(110011011_b) = 110 ^ 011 ^ 011 = 110_b = 6

同样地，分割后各区段的方向也可以是往复折返式的，比如

hash(411) = hash(110011011_b) = 110 ^ 110 ^ 011 = 011_b = 3

当然，为保证上述函数取值落在合法的散列地址空间以内，通常都还需要对散列表长度M再做一次取余运算。

■ （伪）随机数法

上述各具特点的散列函数，验证了我们此前的判断：越是随机、越是没有规律，就越是好的散列函数。按照这一标准，任何一个（伪）随机数发生器，本身即是一个好的散列函数。比如，可直接使用C/C++语言提供的rand()函数，将关键码key映射至桶地址：

rand(key) mod M

其中rand(key)为系统定义的第key个（伪）随机数。

这一策略的原理也可理解为，将“设计好散列函数”的任务，转换为“设计好的（伪）随机数发生器”的任务。幸运的是，二者的优化目标几乎是一致的。

需特别留意的是，由于不同计算环境所提供的（伪）随机数发生器不尽相同，故在将某一系统中生成的散列表移植到另一系统时，必须格外小心。

9.3.4 散列表

■ Hashtable模板类

按照词典的标准接口，可以模板类的形式，定义Hashtable类如代码9.13所示。

```
1 #include "../Dictionary/Dictionary.h" //引入词典ADT
2 #include "../Bitmap/Bitmap.h" //引入位图
3
```

```
4  template <typename K, typename V> //key、value
5  class Hashtable : public Dictionary<K, V> { //符合Dictionary接口的Hashtable模板类
6  private:
7     Entry<K, V>** ht; //桶数组，存放词条指针
8     int M; //桶数组容量
9     int N; //词条数量
10    Bitmap* lazyRemoval; //懒惰删除标记
11 #define lazilyRemoved(x)  (lazyRemoval->test(x))
12 #define markAsRemoved(x)  (lazyRemoval->set(x))
13 protected:
14    int probe4Hit ( const K& k ); //沿关键码k对应的查找链，找到词条匹配的桶
15    int probe4Free ( const K& k ); //沿关键码k对应的查找链，找到首个可用空桶
16    void rehash(); //重散列算法：扩充桶数组，保证装填因子在警戒线以下
17 public:
18    Hashtable ( int c = 5 ); //创建一个容量不小于c的散列表（为测试暂时选用较小的默认值）
19    ~Hashtable(); //释放桶数组及其中各（非空）元素所指向的词条
20    int size() const { return N; } // 当前的词条数目
21    bool put ( K, V ); //插入（禁止雷同词条，故可能失败）
22    V* get ( K k ); //读取
23    bool remove ( K k ); //删除
24 };
```

代码9.13 基于散列表实现的映射结构

作为词典结构的统一接口，put()、get()和remove()等操作的具体实现稍后介绍。

这里还基于Bitmap结构（习题[2-34]），维护了一张与散列表等长的懒惰删除标志表lazyRemoval[]，稍后的9.3.6节将介绍其原理与作用。

■ 散列表构造

散列表的初始化过程如代码9.14所示。

```
1  template <typename K, typename V> Hashtable<K, V>::Hashtable ( int c ) { //创建散列表，容量为
2     M = primeNLT ( c, 1048576, "../../_input/prime-1048576-bitmap.txt" ); //不小于c的素数M
3     N = 0; ht = new Entry<K, V>*[M]; //开辟桶数组（还需核对申请成功），初始装填因子为N/M = 0%
4     memset ( ht, 0, sizeof ( Entry<K, V>* ) *M ); //初始化各桶
5     lazyRemoval = new Bitmap ( M ); //懒惰删除标记比特图
6  }
```

代码9.14 散列表构造

为了加速素数的选取，这里不妨借鉴习题[2-36]中的技巧，事先计算出不超过1,048,576的所有素数，并存放于文件中备查。于是在创建散列表（或者重散列）时，对于在此范围内任意给定的长度下限c，都可通过调用primeNLT()，迅速地从该查询表中找到不小于c的最小素数M作为散列表长度，并依此为新的散列表申请相应数量的空桶；同时创建一个同样长度的位图结构，作为懒惰删除标志表。

```
int primeNLT ( int c, int n, char* file ) { //根据file文件中的记录，在[c, n)内取最小的素数
   Bitmap B ( file, n ); //file已经按位图格式，记录了n以内的所有素数，因此只要
   while ( c < n ) //从c开始，逐位地
      if ( B.test ( c ) ) c++; //测试，即可
      else return c; //返回首个发现的素数
   return c; //若没有这样的素数，返回n（实用中不能如此简化处理）
}
```

代码9.15 确定散列表的素数表长

如代码9.15所示，从长度下限c开始，逐个测试对应的标志位，直到第一个足够大的素数。

■ 散列表析构

```
template <typename K, typename V> Hashtable<K, V>::~Hashtable() { //析构前释放桶数组及非空词条
   for ( int i = 0; i < M; i++ ) //逐一检查各桶
      if ( ht[i] ) release ( ht[i] ); //释放非空的桶
   release ( ht ); //释放桶数组
   release ( lazyRemoval ); //释放懒惰删除标记
}
```

代码9.16 散列表析构

在销毁散列表之前，如代码9.16所示，需在逐一释放各桶中的词条（如果存在）之后，释放整个散列表ht[]以及对应的懒惰删除表lazyRemoval[]。

9.3.5 冲突及其排解

■ 冲突的普遍性

散列表的基本构思，可以概括为：

开辟物理地址连续的桶数组ht[]，借助散列函数hash()，将词条关键码key映射为桶地址hash(key)，从而快速地确定待操作词条的物理位置。

然而遗憾的是，无论散列函数设计得如何巧妙，也不可能保证不同的关键码之间互不冲突。比如，若试图在如图9.7所示的散列表中插入电话号码6278-2001，便会与已有的号码5153-1876相冲突。而在实际应用中，不发生任何冲突的概率远远低于我们的想象。

考查如下问题：某课堂的所有学生中，是否有某两位生日（birthday，而非date of birth）相同？这种情况也称作生日巧合。那么，发生生日巧合事件的概率是多少？

若将全年各天视作365个桶，并将学生视作词条，则可按生日将他们组织为散列表。如此，上述问题便可转而表述为：若长度为365的散列表中存有n个词条，则至少发生一次冲突的概率$P_{365}(n)$有多大？不难证明（习题[9-8]），只要学生人数$n \geq 23$，即有$P_{365}(n) > 50\%$。请注意，此时的装填因子仅为$\lambda = 23/365 = 6.3\%$。

不难理解，对于更长的散列表，只需更低的装填因子，即有50%的概率会发生一次冲突。鉴于实际问题中散列表的长度M往往远大于365，故“不会发生冲突”只是一厢情愿的幻想。因此，我们必须事先制定一整套有效的对策，以处理和排解时常发生的冲突。

■ 多槽位（multiple slots）法

最直截了当的一种对策是，将彼此冲突的每一组词条组织为一个小规模的子词典，分别存放于它们共同对应的桶单元中。比如一种简便的方法是，统一将各桶细分为更小的称作槽位（slot）的若干单元，每一组槽位可组织为向量或列表。

图9.10 通过槽位细分排解散列冲突

例如，对于如图9.8(a)所示的冲突散列表，可以如图9.10所示，将各桶细分为四个槽位。只要相互冲突的各组关键码不超过4个，即可分别保存于对应桶单元内的不同槽位。

按照这一思路，针对关键码key的任一操作都将转化为对一组槽位的操作。比如put(key, value)操作，将首先通过hash(key)定位对应的桶单元，并在其内部的一组槽位中，进一步查找key。若失败，则创建新词条(key, value)，并将其插至该桶单元内的空闲槽位（如果的确还有的话）中。get(key)和remove(key)操作的过程，与此类似。

多槽位法的缺陷，显而易见。首先由图9.10可见，绝大多数的槽位通常都处于空闲状态。准确地讲，若每个桶被细分为k个槽位，则当散列表总共存有N个词条时，装填因子

$$\lambda' = N/(kM) = \lambda/k$$

将降低至原先的1/k。

其次，很难在事先确定槽位应细分到何种程度，方可保证在任何情况下都够用。比如在极端情况下，有可能所有（或接近所有）的词条都冲突于单个桶单元。此时，尽管几乎其余所有的桶都处于空闲状态，该桶却会因冲突过多而溢出。

■ 独立链（separate chaining）法

冲突排解的另一策略与多槽位（multiple slots）法类似，也令相互冲突的每组词条构成小规模的子词典，只不过采用列表（而非向量）来实现各子词典。

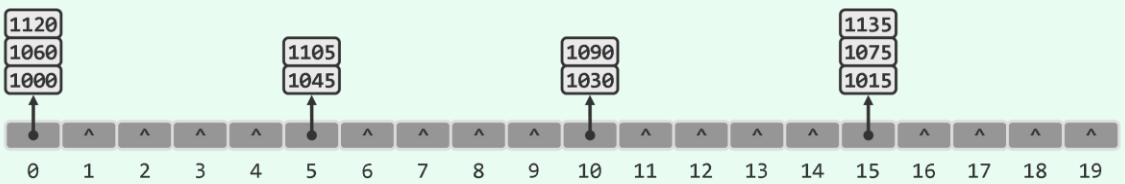


图9.11 利用建立独立链排解散列冲突

仍以图9.8(a)中的冲突为例，可如图9.11所示令各桶内相互冲突的词条串接成一个列表，该方法也因此得名。

既然好的散列函数已能保证通常不致发生极端的冲突，故各子词典的规模往往都不是很大，大多数往往只含单个词条或者甚至是空的。因此，采用第3章的基本列表结构足矣。

相对于多槽位法，独立链法可更为灵活地动态调整各子词典的容量和规模，从而有效地降低空间消耗。但在查找过程中一旦发生冲突，则需要遍历整个列表，导致查找成本的增加。

■ 公共溢出区法

公共溢出区（overflow area）法的思路如图9.12所示，在原散列表（图(a)）之外另设一个词典结构$D_{overflow}$（图(b)），一旦在插入词条时发生冲突就将该词条转存至$D_{overflow}$中。就效果而言，$D_{overflow}$相当于一个存放冲突词条的公共缓冲池，该方法也因此得名。

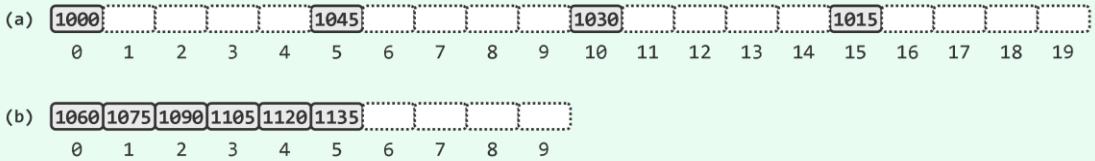


图9.12 利用公共溢出区解决散列冲突

这一策略构思简单、易于实现，在冲突不甚频繁的场合不失为一种好的选择。同时，既然公共溢出区本身也是一个词典结构，不妨直接套用现有的任何一种实现方式——因此就整体结构而言，此时的散列表也可理解为是一种递归形式的散列表。

9.3.6 闭散列策略

尽管就逻辑结构而言，独立链等策略便捷而紧凑，但绝非上策。比如，因需要引入次级关联结构，实现相关算法的代码自身的复杂程度和出错概率都将加大大增加。反过来，因不能保证物理上的关联性，对于稍大规模的词条集，查找过程中将需做更多的I/O操作。

实际上，仅仅依靠基本的散列表结构，且就地排解冲突，反而是更好的选择。也就是说，若新词条与已有词条冲突，则只允许在散列表内部为其寻找另一空桶。如此，各桶并非注定只能存放特定的一组词条；从理论上讲，每个桶单元都有可能存放任一词条。因为散列地址空间对所有词条开放，故这一新的策略亦称作开放定址（open addressing）；同时，因可用的散列地址仅限于散列表所覆盖的范围之内，故亦称作闭散列（closed hashing）。相应地，此前的策略亦称作封闭定址（closed addressing）或开散列（open hashing）。

当然，仅仅能够为冲突的词条选择一个可用空桶还不足够；更重要地，在后续的查找过程中应能正确地找到这个（些）词条。为此，须在事先约定好某种具体可行的查找方案。

实际上，开放定址策略涵盖了一系列的冲突排解方法，包括线性试探法、平方试探法以及再散列法等。因不得使用附加空间，装填因子需要适当降低，通常都取$\lambda \leq 0.5$。

■ 线性试探（linear probing）法

如图9.13所示，开放定址策略最基本的一种形式是：在插入关键码key时，若发现桶单元ht[hash(key)]已被占用，则转而试探桶单元ht[hash(key) + 1]；若ht[hash(key) + 1]也被占用，则继续试探ht[hash(key) + 2]；...；如此不断，直到发现一个可用空桶。当然，为确保桶地址的合法，最后还需统一对M取模。因此准确地，第i次试探的桶单元应为：

```
ht[ (hash(key) + i) mod M ],  i = 1, 2, 3, ...
```

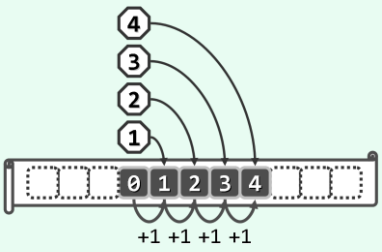


图9.13 线性试探法

如此，被试探的桶单元在物理空间上依次连贯，其地址构成等差数列，该方法由此得名。

■ 查找链

采用开放地址策略时，散列表中每一组相互冲突的词条都将被视作一个有序序列，对其中任何一员的查找都需借助这一序列。对应的查找过程，可能终止于三种情况：

（1）在当前桶单元命中目标关键码，则成功返回；

（2）当前桶单元非空，但其中关键码与目标关键码不等，则须转入下一桶单元继续试探；

（3）当前桶单元为空，则查找以失败返回。

考查如图9.14所示长度为M = 17的散列表，设采用除余法定址，采用线性试探法排解冲突。

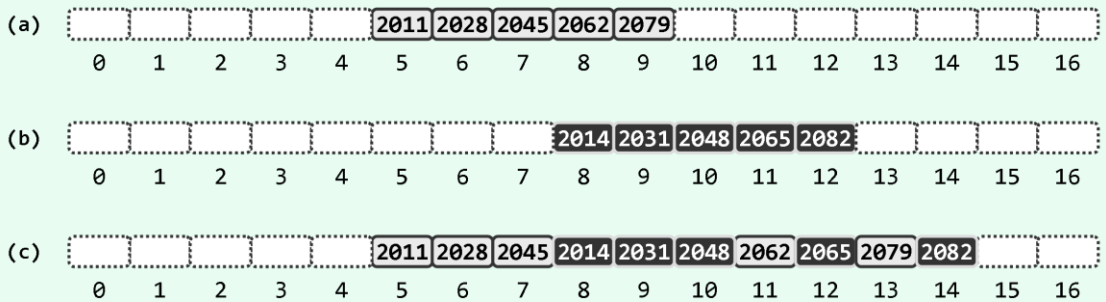


图9.14 线性试探法对应的查找链

若从空表开始，依次插入5个相互冲突的关键码{ 2011, 2028, 2045, 2062, 2079 }，则结果应如图(a)所示。此后，针对其中任一关键码的查找都将从：

```
ht[hash(key)] = ht[5]
```

出发，试探各相邻的桶单元。可见，与这组关键码对应的桶单元ht[5, 10)构成一个有序序列，对其中任一关键码的查找都将沿该序列顺序进行，故该序列亦称作查找链（probing chain）。类似地，另一组关键码{ 2014, 2031, 2048, 2065, 2082 }对应的查找链，如图(b)所示。

可见，沿查找链试探的过程，与对应关键码此前的插入过程完全一致。因此对于长度为n的查找链，失败查找长度就是n + 1；在等概率假设下，平均成功查找长度为⌈n/2⌉。

需强调的是，尽管相互冲突的关键码必属于同一查找链，但反过来，同一查找链中的关键码却未必相互冲突。仍以上述散列表为例，若将以上两组关键码合并，并按从小到大的次序逐一插入空散列表，结果将如图(c)所示。可见，对于2079或2082等关键码而言，查找链中的关键码未必与它们冲突。究其原因在于，多组各自冲突的关键码所对应的查找链，有可能相互交织和重叠。此时，各组关键码的查找长度将会进一步增加。仍以这两组关键码为例，在图(c)状态下，失败查找长度分别为为11和8，而在等概率假设下的平均成功查找长度分别为：

```
( 1 + 2 + 3 + 7 + 9 ) / 5  =  4.4
( 1 + 2 + 3 + 5 + 7 ) / 5  =  3.6
```

■ 局部性

由上可见，线性试探法中组成各查找链的词条，在物理上保持一定的连贯性，具有良好的数据局部性，故系统缓存的作用可以充分发挥，查找过程中几乎无需I/O操作。尽管闭散列策略同时也会在一定程度上增加冲突发生的可能，但只要散列表的规模不是很小，装填因子不是很大，则相对于I/O负担的降低而言，这些问题都将微不足道。也正因为此，相对于独立链等开散列策略，闭散列策略的实际应用更为广泛。

■ 懒惰删除

查找链中任何一环的缺失，都会导致后续词条因无法抵达而丢失，表现为有时无法找到实际已存在的词条。因此若采用开放定址策略，则在执行删除操作时，需同时做特别的调整。

仍以图9.14(c)为例，若为删除词条ht[9] = 2031而如图9.15(a)所示，按常规方法简单地将其清空，则该桶的缺失将导致对应的查找链“断裂”，从而致使五个后继词条“丢失”——尽管它们在词典中的确存在，但查找却会失败。

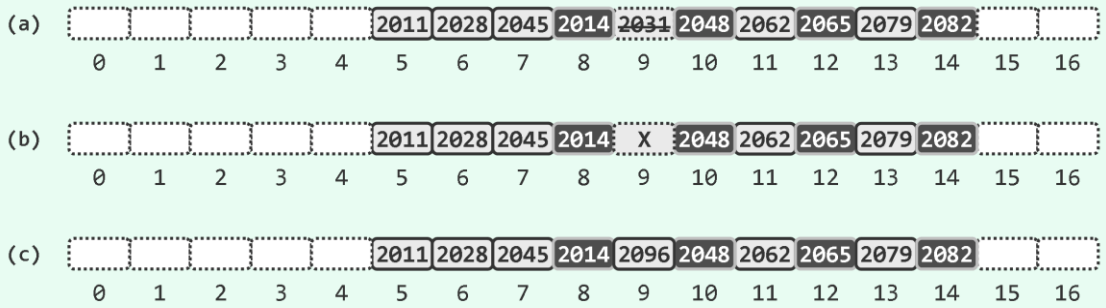


图9.15 通过设置懒惰删除标记，无需大量词条的重排即可保证查找链的完整

为保持查找链的完整，一种直观的构想是将后继词条悉数取出，然后再重新插入。很遗憾，如此将导致删除操作的复杂度增加，故并不现实。简明而有效的方法是，为每个桶另设一个标志位，指示该桶尽管目前为空，但此前确曾存放过词条。

在Hashtable模板类（代码9.13）中，名为lazyRemoval的Bitmap对象（习题[2-34]）扮演的就是这一角色。具体地，为删除词条，只需将对应的桶ht[r]标志为lazilyRemoved(r)。如此，该桶虽不存放任何实质的词条，却依然是查找链上的一环。如图9.15(b)所示，在将桶ht[9]作此标记（以X示意）之后，对后继词条的查找仍可照常进行，而不致中断。这一方法既可保证查找链的完整，同时所需的时间成本也极其低廉，称作懒惰删除（lazy removal）法。

请注意，设有懒惰删除标志位的桶，应与普通的空桶一样参与插入操作。比如在图9.15(b)基础上，若拟再插入关键码2096，则应从ht[hash(2096)] = ht[5]出发，沿查找链经5次试探抵达桶ht[9]，并如图(c)所示将关键码2096置入其中。需特别说明的是，此后不必清除该桶的懒惰删除标志——尽管按照软件工程的规范，最好如此。

■ 两类查找

采用“懒惰删除”策略之后，get()、put()和remove()等操作中的查找算法，都需要做相应的调整。这里共分两种情况。

其一，在删除等操作之前对某一目标词条的查找。此时，对成功的判定条件基本不变，但对失败的判定条件需兼顾懒惰删除标志。在查找过程中，只有在当前桶单元为空，且不带懒惰删除标记时，方可报告“查找失败”；否则，无论该桶非空，或者带有懒惰删除标志，都将沿着查找链继续试探。这一查找过程probe4Hit()，可具体描述和实现如代码9.18所示。

其二，在插入等操作之前沿查找链寻找空桶。此时对称地，无论当前桶为空，还是带有懒惰删除标记，均可报告“查找成功”；否则，都将沿查找链继续试探。这一查找过程probe4Free()，可具体描述和实现如代码9.21所示。

9.3.7 查找与删除

■ get()

```
1 template <typename K, typename V> V* Hashtable<K, V>::get ( K k ) //散列表词条查找算法
2 {  int r = probe4Hit ( k ); return ht[r] ? & ( ht[r]->value ) : NULL;  } //禁止词条的key值雷同
```

代码9.17 散列表的查找

词条查找操作接口，可实现如代码9.17所示。可见，其实质的过程只不过是调用以下的probe4Hit(k)算法，沿关键码k所对应的查找链顺序查找。

■ probe4Hit()

借助如代码9.18所示的probe4Hit()算法，可确认散列表是否包含目标词条。

```
1 /*****************************************************************************************
2  * 沿关键码k对应的查找链，找到与之匹配的桶（供查找和删除词条时调用）
3  * 试探策略多种多样，可灵活选取；这里仅以线性试探策略为例
4  *****************************************************************************************/
5 template <typename K, typename V> int Hashtable<K, V>::probe4Hit ( const K& k ) {
6    int r = hashCode ( k ) % M; //从起始桶（按除余法确定）出发
7    while ( ( ht[r] && ( k != ht[r]->key ) ) || ( !ht[r] && lazilyRemoved ( r ) ) )
8       r = ( r + 1 ) % M; //沿查找链线性试探：跳过所有冲突的桶，以及带懒惰删除标记的桶
9    return r; //调用者根据ht[r]是否为空，即可判断查找是否成功
10 }
```

代码9.18 散列表的查找probe4Hit()

首先采用除余法确定首个试探的桶单元，然后按线性试探法沿查找链逐桶试探。请注意，这里共有两种试探终止的可能：在一个非空的桶内找到目标关键码（成功），或者遇到一个不带懒惰删除标记的空桶（失败）。否则，无论是当前桶中词条的关键码与目标关键码不等，还是当前桶为空但带有懒惰删除标记，都意味着有必要沿着查找链前进一步继续查找。该算法统一返回最后被试探桶的秩，上层调用者只需核对该桶是否为空，即可判断查找是否失败。

可见，借助懒惰删除标志，的确可以避免查找链的断裂。当然，在此类查找中，也可将懒惰标志，等效地视作一个与任何关键码都不相等的特殊关键码。

■ remove()

词条删除操作接口可实现如代码9.19所示。

```
1 template <typename K, typename V> bool Hashtable<K, V>::remove ( K k ) { //散列表词条删除算法
2    int r = probe4Hit ( k ); if ( !ht[r] ) return false; //对应词条不存在时，无法删除
3    release ( ht[r] ); ht[r] = NULL; markAsRemoved ( r ); N--; return true;
4    //否则释放桶中词条，设置懒惰删除标记，并更新词条总数
5 }
```

代码9.19 散列表元素删除（采用懒惰删除策略）

这里首先调用probe4Hit(k)算法，沿关键码k对应的查找链顺序查找。若在某桶单元命中，则释放其中的词条，为该桶单元设置懒惰删除标记，并更新词典的规模。

9.3.8 插入

■ put()

```
1 template <typename K, typename V> bool Hashtable<K, V>::put ( K k, V v ) { //散列表词条插入
2    if ( ht[probe4Hit ( k ) ] ) return false; //雷同元素不必重复插入
3    int r = probe4Free ( k ); //为新词条找个空桶（只要装填因子控制得当，必然成功）
4    ht[r] = new Entry<K, V> ( k, v ); ++N; //插入（注意：懒惰删除标记无需复位）
5    if ( N * 2 > M ) rehash(); //装填因子高于50%后重散列
6    return true;
7 }
```

代码9.20 散列表元素插入

词条插入操作的过程，可描述和实现如代码9.20所示。调用以下probe4Free(k)算法，若沿关键码k所属查找链能找到一个空桶，则在其中创建对应的词条，并更新词典的规模。

■ probe4Free()

如代码9.21所示，借助probe4Free()算法可在散列表中找到一个空桶。

```
1 /*****************************************************************************************
2  * 沿关键码k对应的查找链，找到首个可用空桶（仅供插入词条时调用）
3  * 试探策略多种多样，可灵活选取；这里仅以线性试探策略为例
4  *****************************************************************************************/
5 template <typename K, typename V> int Hashtable<K, V>::probe4Free ( const K& k ) {
6    int r = hashCode ( k ) % M; //从起始桶（按除余法确定）出发
7    while ( ht[r] ) r = ( r + 1 ) % M; //沿查找链逐桶试探，直到首个空桶（无论是否带有懒惰删除标记）
8    return r; //为保证空桶总能找到，装填因子及散列表长需要合理设置
9 }
```

代码9.21 散列表的查找probe4Free()

采用除余法确定起始桶单元之后，沿查找链依次检查，直到发现一个空桶。

与在probe4Hit()过程中一样，懒惰标志在此也等效于一个特殊的关键码；不同之处在于，在probe4Free()查找过程中，假想的该关键码与任何关键码都相等。

■ 装填因子

就对散列表性能及效率的影响而言，装填因子λ = N / M是最为重要的一个因素。随着λ的上升，词条在散列表中聚集的程度亦将迅速加剧。若同时还采用基本的懒惰删除法，则不带懒惰删除标记的桶单元必将持续减少，这也势必加剧查找成本的进一步攀升。尽管可以采取一些弥补的措施（习题[9-16]），但究其本质而言，都等效于将懒惰删除法所回避的调整操作推迟实施，而且其编码实现的复杂程度之高，必将将令懒惰删除法的简洁性丧失殆尽。

实际上，理论分析和实验统计均一致表明，只要能将装填因子λ控制在适当范围以内，闭散列策略的平均效率，通常都可保持在较为理想的水平。比如，一般的建议是保持λ < 0.5。这一原则也适用于其它的定址策略，比如对独立链法而言，建议的装填因子上限为0.9。当前主流的编程语言大多提供了散列表接口，其内部装填因子的阈值亦多采用与此接近的阈值。

■ 重散列（rehashing）

其实，将装填因子控制在一定范围以内的方法并不复杂，重散列即是常用的一种方法。

回顾代码9.20中的Hashtable::put()算法可见，一旦装填因子上升到即将越界（这里采用阈值50%），则可调用如代码9.22所示的rehash()算法。

```
/*****************************************************************************************
 * 重散列算法：装填因子过大时，采取“逐一取出再插入”的朴素策略，对桶数组扩容
 * 不可简单地（通过memcpy()）将原桶数组复制到新桶数组（比如前端），否则存在两个问题：
 * 1）会继承原有冲突；2）可能导致查找链在后端断裂——即便为所有扩充桶设置懒惰删除标志也无济于事
 *****************************************************************************************/
template <typename K, typename V> void Hashtable<K, V>::rehash() {
   int old_capacity = M; Entry<K, V>** old_ht = ht;
   M = primeNLT ( 2 * M, 1048576, "../../_input/prime-1048576-bitmap.txt" ); //容量至少加倍
   N = 0; ht = new Entry<K, V>*[M]; memset ( ht, 0, sizeof ( Entry<K, V>* ) * M ); //新桶数组
   release ( lazyRemoval ); lazyRemoval = new Bitmap ( M ); //新开懒惰删除标记比特图
   for ( int i = 0; i < old_capacity; i++ ) //扫描原桶数组
      if ( old_ht[i] ) //将非空桶中的词条逐一
         put ( old_ht[i]->key, old_ht[i]->value ); //插入至新的桶数组
   release ( old_ht ); //释放原桶数组——由于其中原先存放的词条均已转移，故只需释放桶数组本身
}
```

代码9.22 散列表的重散列

可见，重散列的效果，只不过是将原词条集，整体“搬迁”至容量至少加倍的新散列表中。与可扩充向量同理，这一策略也可使重散列所耗费的时间，在分摊至各次操作后可以忽略不计。

9.3.9 更多闭散列策略

■ 聚集现象

线性试探法虽然简明紧凑，但各查找链均由物理地址连续的桶单元组成，因而会加剧关键码的聚集趋势。例如，采用除余法，将7个关键码{ 2011, 2012, 2013, 2014, 2015, 2016, 2017 }依次插入长度M = 17的散列表，则如图9.16(a)所示将形成聚集区段ht[5, 12)。

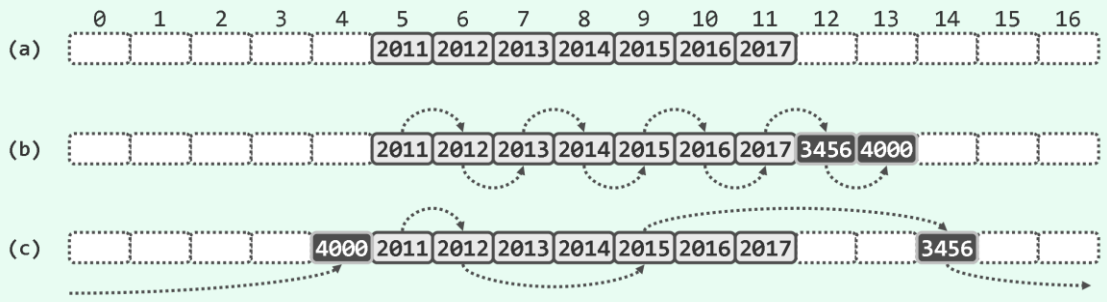


图9.16 线性试探法会加剧聚集现象，而平方试探法则会快速跳离聚集区段

接下来，设拟插入关键码3456和4000。由除余法，hash(3456) = hash(4000) = hash(2011) = 5，故对二者的试探都将起始于桶单元ht[5]。以下按照线性试探法，分别经8次和9次试探后，

它们将被插入于紧邻原聚集区段右侧的位置。结果如图9.16(b)所示，其中的虚弧线示意试探过程。可见，聚集区段因此扩大，而且对这两个关键码的后续查找也相应地十分耗时（分别需做8次和9次试探）。如果再考虑到聚集区段的生长还会加剧不同聚集区段之间的相互交叠，查找操作平均效率的下降程度将会更加严重。

■ 平方试探（quadratic probing）法

采用9.3.3节的MAD法，可在一定程度上缓解上述聚集现象。而平方试探法，则是更为有效的一种方法。具体地，在试探过程中若连续发生冲突，则按如下规则确定第j次试探的桶地址：

$$(hash(key) + j^2) \bmod M, \quad j = 0, 1, 2, \ldots$$

如图9.17所示，各次试探的位置到起始位置的距离，以平方速率增长，该方法因此得名。

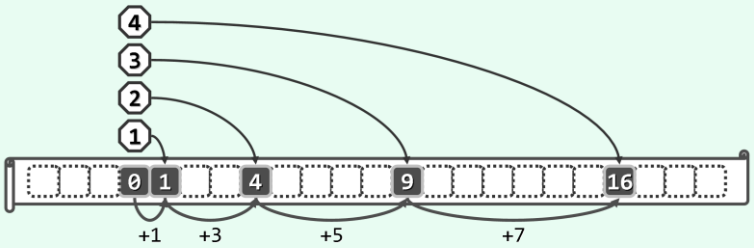


图9.17 平方试探法

仍以图9.16(a)为例。为插入3456，将依次试探秩为5、6、9、14的桶单元，最终将其插至ht[14]。接下来为插入4000，将依次试探秩为5、6、9、14、21 ≡ 4的桶单元，并最终将其插至ht[4]。最终的结果如图9.16(c)所示。

■ 局部性

可见，聚集区段并未扩大，同时针对这两个关键码的后续查找，也分别只需3次和4次试探，速度得以提高至两倍以上。平方试探法之所以能够有效地缓解聚集现象，是因为充分利用了平方函数的特点——顺着查找链，试探位置的间距将以线性（而不再是常数1的）速度增长。于是，一旦发生冲突，即可“聪明地”尽快“跳离”关键码聚集的区段。

反过来，细心的读者可能会担心，试探位置加速地“跳离”起点，将会导致数据局部性失效。然而幸运的是，鉴于目前常规的I/O页面规模已经足够大，只有在查找链极长的时候，才有可能引发额外的I/O操作。仍以由内存与磁盘构成的二级存储系统为例，典型的缓存规模约为KB量级，足以覆盖长度为$\sqrt{1024/4} \approx 16$的查找链。

■ 确保试探必然终止

线性试探法中，只要散列表中尚有空桶，则试探过程至多遍历全表一遍，必然终止。那么，平方试探法是否也能保证这一点呢？

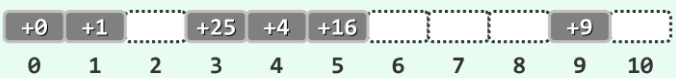


图9.18 即便散列表长取为素数（M = 11），在装填因子λ > 50%时仍可能找不到实际存在的空桶

考查如图9.18所示的实例。这里取M = 11为素数，黑色的桶已存有词条，白色的桶为空。现假设拟插入一个与ht[0]冲突的词条，并从ht[0]出发做平方试探。因为任何整数的平方关于11的余数，恰好只可能来自集合{ 0, 1, 3, 4, 5, 9 }，故所有试探必将局限于这6个非空桶，从而出现“明明存在空桶却永远无法抵达”的奇特现象。

好消息是：只要散列表长度M为素数且装填因子λ ≤ 50%，则平方试探迟早必将终止于某个空桶（习题[9-14]）。照此反观前例，之所以会出现试探无法终止的情况，原因在于当前的装填因子λ = 6/11 > 50%。当然，读者也可从另一角度对上述结论做一验证（习题[9-15]）。

■ （伪）随机试探（pseudo-random probing）法

既然在排解冲突时也需尽可能保证试探位置的随机和均匀分布，自然也可仿照9.3.3节的思路，借助（伪）随机数发生器来确定试探位置。具体地，第j次试探的桶地址取作：

rand(j) mod M .....（rand(i)为系统定义的第j个（伪）随机数）

同样地，在跨平台协同的场合，出于兼容性的考虑，这一策略也须慎用。

■ 再散列（double hashing）法

再散列也是延缓词条聚集趋势的一种有效办法。为此，需要选取一个适宜的二级散列函数$hash_2()$，一旦在插入词条(key, value)时发现ht[hash(key)]已被占用，则以$hash_2(key)$为偏移增量继续尝试，直到发现一个空桶。如此，被尝试的桶地址依次应为：

$[hash(key) + 1 \times hash_2(key)] \% M$

$[hash(key) + 2 \times hash_2(key)] \% M$

$[hash(key) + 3 \times hash_2(key)] \% M$

...

可见，再散列法是对此前各方法的概括。比如取$hash_2(key) = 1$时即是线性试探法。

9.3.10 散列码转换

作为词典的散列表结构，既不能假定词条关键码所属的类型天然地支持大小比较，更不应将关键码仅限定为整数类型。为扩大散列技术的适用范围，散列函数hash()必须能够将任意类型的关键码key映射为地址空间[0, M)内的一个整数hash(key)，以便确定key所对应的散列地址。由关键码到散列地址的映射，如图9.19所示通常可分解为两步。

首先，利用某一种散列码转换函数hashCode()，将关键码key统一转换为一个整数——称作散列码（hash code）；然后，再利用散列函数将散列码映射为散列地址。

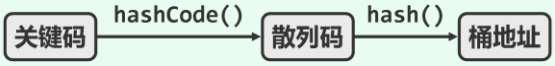


图9.19 分两步将任意类型的关键码，映射为桶地址

那么，这里的散列码转换函数hashCode()应具备哪些条件呢？

首先，为支持后续尺度不同的散列空间，以及种类各异的散列函数，作为中间桥梁的散列码，取值范围应覆盖系统所支持的最大整数范围。其次，各关键码经hashCode()映射后所得的散列码，相互之间的冲突也应尽可能减少——否则，这一阶段即已出现的冲突，后续的hash()函数注定无法消除。最后，hashCode()也应与判等器保持一致。也就是说，被判等器判定为相等的词条，对应的散列码应该相等；反之亦然（习题[9-20]）。

以下针对一些常见的数据类型，列举若干对应的散列码转换方法。

■ 强制转换为整数

对于byte、short、int和char等本身即可表示为不超过32位整数的数据类型，可直接将它们的这种表示作为其散列码。比如，可通过类型强制将它们转化为32位的整数。

■ 对成员对象求和

long long和double之类长度超过32位的基本类型，不宜强制转换为整数。否则，将因原有数位的丢失而引发大量冲突。可行的办法是，将高32位和低32位分别看作两个32位整数，将二者之和作为散列码。这一方法，可推广至由任意多个整数构成的组合对象。比如，可将其成员对象各自对应的整数累加起来，再截取低32位作为散列码。

■ 多项式散列码

与一般的组合对象不同，字符串内各字符之间的次序具有特定含义，故在做散列码转换时，务必考虑它们之间的次序。以英文为例，同一组字母往往可组成意义完全不同的多个单词，比如"stop"和"tops"等。而玩过“Swipe & Spell”之类组词游戏的读者，对此应该理解更深。

若简单地将各字母分别对应到整数（比如1 ~ 26），并将其总和作为散列码，则很多单词都将相互冲突。即便是对于句子等更长的字符串，这一问题也很突出，且此时发生冲突的可能性远高于直观想象。比如依照此法，以下三个字符串均相互冲突：

```
"I am Lord Voldemort"
"Tom Marvolo Riddle"
"He's Harry Potter"
```

以下则是此类冲突的另一实例：

```
"Key to improving your programming skill"
"Learning Tsinghua Data Structure and Algorithm"
```

为计入各字符的出现次序，可取常数$a \geq 2$，并将字符串"$x_0x_1...x_{n-1}$"的散列码取作：

$$x_0a^{n-1} + x_1a^{n-2} + ... + x_{n-2}a^1 + x_{n-1}$$

这一转换等效于，依次将字符串中的各个字符，视作一个多项式的各项系数，故亦称作多项式散列码（polynomial hash code）。其中的常数a非常关键，为尽可能多地保留原字符串的信息以减少冲突，其低比特位不得全为零。另外，针对不同类型的字符串，应通过实验确定a的最佳取值。实验表明，对于英语单词之类的字符串，a = 33、37、39或41都是不错的选择。

■ hashCode()的实现

针对若干常见类型，代码9.23利用重载机制，实现了散列码的统一转换方法hashCode()。

```
1 static size_t hashCode ( char c ) { return ( size_t ) c; } //字符
2 static size_t hashCode ( int k ) { return ( size_t ) k; } //整数以及长长整数
3 static size_t hashCode ( long long i ) { return ( size_t ) ( ( i >> 32 ) + ( int ) i ); }
4 static size_t hashCode ( char s[] ) { //生成字符串的循环移位散列码（cyclic shift hash code）
5    int h = 0; //散列码
6    for ( size_t n = strlen ( s ), i = 0; i < n; i++ ) //自左向右，逐个处理每一字符
7       { h = ( h << 5 ) | ( h >> 27 ); h += ( int ) s[i]; } //散列码循环左移5位，再累加当前字符
8    return ( size_t ) h; //如此所得的散列码，实际上可理解为近似的“多项式散列码”
9 } //对于英语单词，"循环左移5位"是实验统计得出的最佳值
```

代码9.23 散列码转换函数hashCode()

读者可视具体应用的需要，在此基础之上继续补充、扩展和尝试更多的关键码类型。

§9.4 *散列应用

9.4.1 桶排序

■ 简单情况

考查如下问题：给定[0, M)内的n个互异整数（n ≤ M），如何高效地对其排序？

自然，2.8节向量排序器或3.5节列表排序器中的任一排序算法，均可完成这一任务。但正如2.7.5节所指出的，CBA式排序算法注定在最坏情况下需要Ω(nlogn)时间。实际上，针对数值类型和取值范围特定的这一具体问题，完全可在更短的时间内完成排序。

为此，引入长度为M的散列表。比如，图9.20即为取M = 10和n = 5的一个实例。

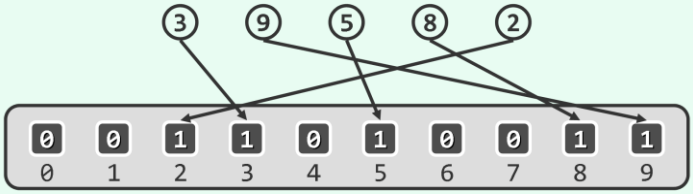


图9.20 利用散列表对一组互异整数排序

接下来，使用最简单的散列函数hash(key) = key，将这些整数视作关键码并逐一插入散列表中。最后，顺序遍历一趟该散列表，依次输出非空桶中存放的关键码，即可得到原整数集合的排序结果。

该算法借助一组桶单元实现对一组关键码的分拣，故称作桶排序（bucketsort）。

该算法所用散列表共占$\mathcal{O}(M)$空间。散列表的创建和初始化耗时$\mathcal{O}(M)$，将所有关键码插入散列表耗时$\mathcal{O}(n)$，依次读出非空桶中的关键码耗时$\mathcal{O}(M)$，故总体运行时间为$\mathcal{O}(n + M)$。

■ 一般情况

若将上述问题进一步推广：若允许输入整数重复，又该如何高效地实现排序？

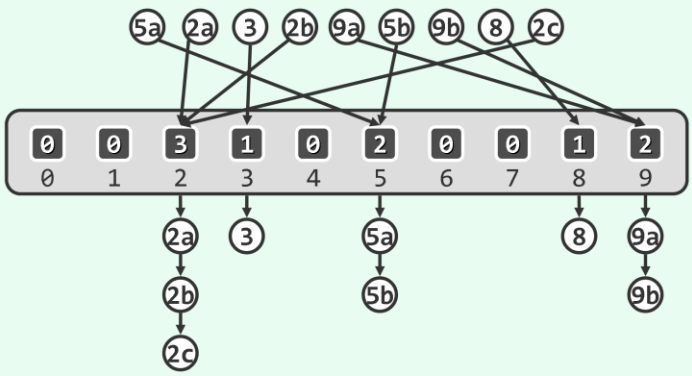


图9.21 利用散列表对一组可能重复的整数排序

依然可以沿用以上构思，只不过这次需要处理散列冲突。具体地如图9.21所示，不妨采用独立链法排解冲突。在将所有整数作为关键码插入散列表之后，只需一趟顺序遍历将各非空桶中的独立链依次串接起来，即可得到完整的排序结果。而且只要在串联时留意链表方向，甚至可以确保排序结果的稳定，故如此实现的桶排序算法属于稳定算法。

如此推广之后的桶排序算法，依然只需为维护散列表而使用$O(M)$的额外空间；算法各步骤所耗费的时间也与前一算法相同，总体运行时间亦为$O(n + M)$。

其实，这一问题十分常见，它涵盖了众多实际应用中的具体需求。此类问题往往还具有另一特点，即$n >> M$。比如，若对清华大学2011级本科生按生日排序，则大致有$n = 3300$和$M = 365$。而在人口普查之后若需对全国人口按生日排序，则大致有：

$$n > 1,300,000,000 \quad 和 \quad M < 365 \times 100 = 36,500$$

再如，尽管邮局每天需要处理的往来信函和邮包不计其数，但因邮政编码不过6位，故分拣系统若使用“散列表”，其长度至多不过10^6。

参照此前的分析可知，在$n >> M$的场合，桶排序算法的运行时间将是：

$$O(n + M) = O(\max(n, M)) = O(n)$$

线性正比于待排序元素的数目，突破了$\Omega(n \log n)$的下界！

其实这不足为奇。以上基于散列表的桶排序算法，采用的是循秩访问的方式，摒弃了以往基于关键码大小比较式的设计思路，故自然不在受到CBA式算法固有的下界约束。正因为此，桶排序在算法设计方面也占有其独特的地位，以下即是一例。

9.4.2 最大间隙

试考查如下问题：任意n个互异点都将实轴切割为$n + 1$段，除去最外侧无界的两段，其余有界的$n - 1$段中何者最大？若将相邻点对之间的距离视作间隙，则该问题可直观地表述为，找出其中的最大间隙（maximum gap）。比如，图9.22(a)就是$n = 7$的实例。

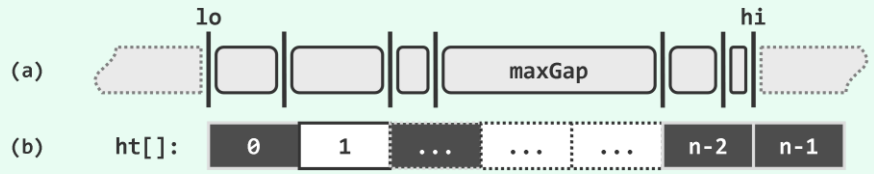


图9.22 利用散列法，在线性时间内确定n个共线点之间的最大间隙

■ 平凡算法

显而易见的一种方法是：先将各点按坐标排序；再顺序遍历，依次计算出各相邻点对之间的间隙；遍历过程中只需不断更新最大间隙的记录，则最终必将得到全局的最大间隙。

该算法的正确性毋庸置疑，但就时间复杂度而言，第一步常规排序即需$\Omega(n \log n)$时间，故在最坏情况下总体运行时间将不可能少于这一下界。

那么，能否实现更高的效率呢？采用散列策略即可做到！

■ 散列

具体方法如图9.22(b)所示。首先，通过一趟顺序扫描找到最靠左和最靠右的点，将其坐标分别记作lo和hi；然后，建立一个长度为n的散列表，并使用散列函数

$$hash(x) = \lfloor (n - 1) * (x - lo) / (hi - lo) \rfloor$$

将各点分别插入对应的桶单元，其中x为各点的坐标值，$hash(x)$为对应的桶编号。其效果相当于：将有效区间$[lo, hi)$均匀地划分为宽度$w = (hi - lo) / (n - 1)$的$n - 1$个左闭右开区间，分别对应于第0至$n - 2$号桶单元；另外，hi独自占用第$n - 1$号桶。

然后，对散列表做一趟遍历，在每个非空桶（黑色）内部确定最靠左和最靠右的点，并删除所有的空桶（白色）。最后，只需再顺序扫描一趟散列表，即可确定相邻非空桶之间的间隙，记录并报告其中的最大者，即为全局的最大间隙。

■ 正确性

该算法的正确性基于以下事实：n - 1个间隙中的最宽者，绝不可能窄于这些间隙的平均宽度，而后者同时也是各桶单元所对应区间的宽度，故有：

maxGap ≥ w = (hi - lo) / (n - 1)

这就意味着，最大间隙的两个端点绝不可能落在同一个桶单元内。进一步地，它们必然来自两个不同的非空桶（当然，它们之间可能会还有若干个空桶），且左（右）端点在前一（后一）非空桶中应该最靠右（左）——故在散列过程中只需记录各桶中的最左、最右点。

■ 复杂度

空间方面，除了输入本身这里只需维护一个散列表，共占用$O(n)$的辅助空间。

无论是生成散列表、找出各桶最左和最右点，还是计算相邻非空桶之间的间距，并找出其中的最大者，该算法的每一步均耗时$O(n)$。故即便在最坏情况下，累计运行时间也不超过$O(n)$。

9.4.3 基数排序

■ 字典序

正如9.3.10节所指出的，实际应用环境中词条的关键码，未必都是整数。比如，一种常见的情形是，关键码由多个域（字段）组合而成，并采用所谓的字典序（lexicographical order）确定大小次序：任意两个关键码之间的大小关系，取决于它们第一个互异的域。

请注意，同一关键码内各字段的类型也未必一致。例如日期型关键码，可分解为year（年）、month（月）和day（日）三个整数字段，并按常规惯例，以“年-月-日”的优先级定义字典序。

再如扑克牌所对应的关键码，可以分解为枚举型的suite（花色）和整型的number（点数）。于是，若按照桥牌的约定，以“花色-点数”为字典序，则每副牌都可按大小排列为：

♠A > ♠K > ♠Q > ♠J > ♠10 > ... > ♠2 >
♥A > ♥K > ♥Q > ♥J > ♥10 > ... > ♥2 >
♦A > ♦K > ♦Q > ♦J > ♦10 > ... > ♦2 >
♣A > ♣K > ♣Q > ♣J > ♣10 > ... > ♣2

一般地，对于任意一组此类关键码，又该如何高效地排序呢？

■ 低位优先的多趟桶排序

这里不妨假定，各字段类型所对应的比较器均已就绪，以将精力集中于如何高效实现依字典序的排序。实际上通过重写比较器，以下算法完全可以推广至一般情况。

假设关键码由t个字段$\{ k_t, k_{t-1}, \ldots, k_1 \}$组成，其中字段$k_t$（$k_1$）的优先级最高（低）。

于是，以其中任一字段k_i为关键码，均可调用以上桶排序算法做一趟排序。稍后我们将证明，只需按照优先级递增的次序（从k_1到k_t）针对每一字段各做一趟桶排序，即可实现按整个关键码字典序的排序。

这一算法称作基数排序（radixsort），它采用了低位字段优先（least significant digit first）的策略。其中所做桶排序的趟数，取决于组成关键码的字段数。

■ 实例

表9.3给出了一个基数排序的实例，其中待排序的7个关键码均可视作由百位、十位和个位共三个数字字段组成。

表9.3 基数排序实例

输入序列	441	276	320	214	698	280	112
以个位排序	320	280	441	112	214	276	698
以十位排序	112	214	320	441	276	280	698
以百位排序	112	214	276	280	320	441	698

可见，在分别针对个位、十位和百位做过一趟桶排序之后，最终的确得到了正确的排序结果。这一成功绝非偶然或幸运，整个算法的正确性可用数学归纳法证明。

■ 正确性与稳定性

我们以如下命题作为归纳假设：在经过基数排序的前i趟桶排序之后，所有词条均已按照关键码最低的i个字段有序排列。

作为归纳的起点，在i = 1时这一假设不证自明。现在假定该命题对于前i - 1趟均成立，继续考查第i趟桶排序做过之后的情况。

任取一对词条，并比较其关键码的第i个字段，无非两种情况。其一，二者的这一字段不等。此时，由于刚刚针对该字段做过一趟桶排序，故二者的排列次序不致颠倒。其二，二者的这一字段相等。此时，二者的大小实际上取决于最低的i - 1个字段。若采用9.4.1节所实现的桶排序算法，则得益于其稳定性，由归纳假设可知这对词条在前一趟桶排序后正确的相对次序将得以延续。整个基数排序算法的正确性由此得证。

由以上分析也可发现，如此实现的基数排序算法同样也是稳定的。

■ 复杂度

根据以上基数排序的流程，总体运行时间应等于其中各趟桶排序所需时间的总和。

设各字段取值范围为$[0, M_i)$，$1 \le i \le t$。若记

$$M = \max\{ m_1, m_2, \ldots, m_t \}$$

则总体运行时间不超过：

$$\mathcal{O}(n + M_1) + \mathcal{O}(n + M_2) + \ldots + \mathcal{O}(n + M_t)$$
$$= \mathcal{O}(t * (n + M))$$