

## 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<sup>①</sup>。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。

<sup>①</sup> 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，都由人物的名字（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

248

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<sup>②</sup>（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。

<sup>②</sup> 由W. Pugh于1989年发明<sup>[52]</sup>

### 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{ $S_0, S_1, S_2, \dots, S_h$ }组成， $h$ 称作跳转表的高度。

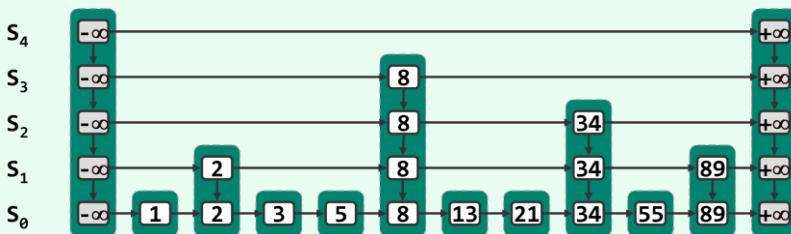


图9.2 跳转表的总体逻辑结构

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。

```
1 template <typename T> void Quadlist<T>::init() { //Quadlist初始化，创建Quadlist对象时统一调用
2     header = new QuadlistNode<T>; //创建头哨兵节点
3     trailer = new QuadlistNode<T>; //创建尾哨兵节点
4     header->succ = trailer; header->pred = NULL; //沿横向联接哨兵
5     trailer->pred = header; trailer->succ = NULL; //沿横向联接哨兵
6     header->above = trailer->above = NULL; //纵向的后继置空
7     header->below = trailer->below = NULL; //纵向的前驱置空
8     _size = 0; //记录规模
9 } //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。

```
1 template <typename K, typename V> V* Skiplist<K, V>::get ( K k ) { //跳转表词条查找算法
2     if ( empty() ) return NULL;
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     return skipSearch ( qlist, p, k ) ? & ( p->entry.value ) : NULL; //查找并报告
6 } //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()**。

```
1 ****
2 * Skiplist词条查找算法（供内部调用）
3 * 入口：qlist为顶层列表，p为qlist的首节点
4 * 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
5 *       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
6 * 约定：多个词条命中时，沿四联表取最靠后者
7 ****
8 template <typename K, typename V> bool Skiplist<K, V>::skipSearch (
9     ListNode<Quadlist<Entry<K, V>>>* &qlist, //从指定层qlist的
10    QuadlistNode<Entry<K, V>>* &p, //首节点p出发
11    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{ $-\infty, -\infty, 8, 8, 8, 8, 13$ }，最终抵达21后报告成功；针对关键码34的查找经过节点{ $-\infty, -\infty, 8, 8$ }，最终抵达34后报告成功；针对关键码1的查找经过节点{ $-\infty, -\infty, -\infty, -\infty, -\infty$ }，最终抵达1后报告成功。而针对关键码80的查找经过节点{ $-\infty, -\infty, 8, 8, 34, 34, 34, 55$ }，最终抵达89后报告失败；针对关键码0的查找经过节点{ $-\infty, -\infty, -\infty, -\infty, -\infty$ }，最终抵达1后报告失败；针对关键码99的查找经过节点{ $-\infty, -\infty, 8, 8, 34, 34, 89$ }，最终抵达 $+\infty$ 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 $\Omega(nh)$ 。

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 $0 \leq k < h$ ， $S_k$ 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：

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

### 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 $O(\log n)$ 。

### ■ 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：

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

反过来， $S_k$ 为空的概率即为：

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

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：

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

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：

$$E(h) = O(\log n)$$

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 $O(\log n)$ 。

### ■ 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 $0 \leq k$ ， $Y$ 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 $\mathcal{O}(\log n)$ 。

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

### 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 }
```

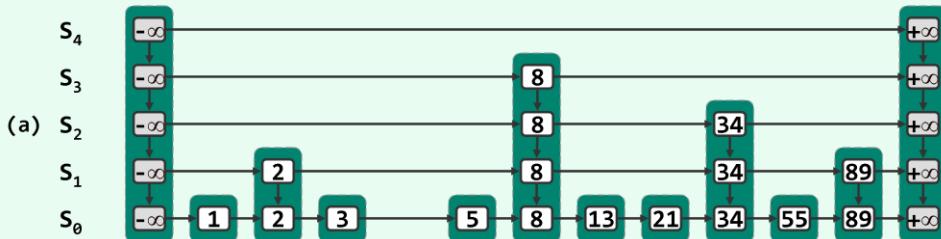
代码9.8 `Skiplist::put()`插入

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。

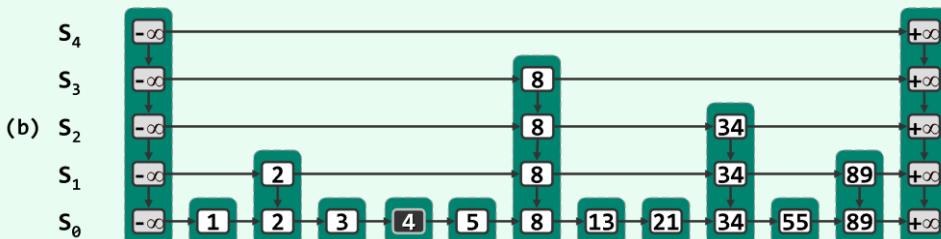
由此可见，新塔最终的（期望）高度，将取决于此前连续的正面硬币事件的（期望）次数。

## ■ 实例

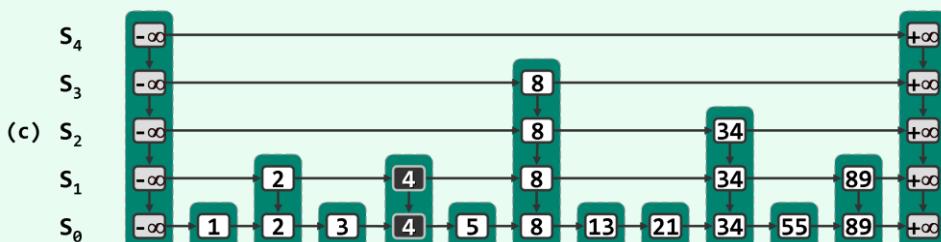
考查如图9.2所示的跳转表。将关键码4插入其中的过程，如图9.3(a~d)所示。



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。

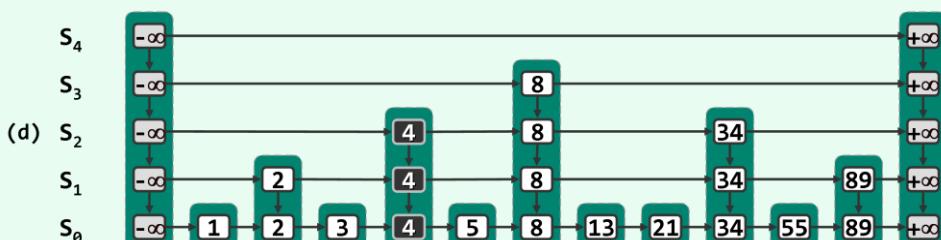


图9.3 跳转表节点插入过程(a~d)，也是节点删除的逆过程(d~a)

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。

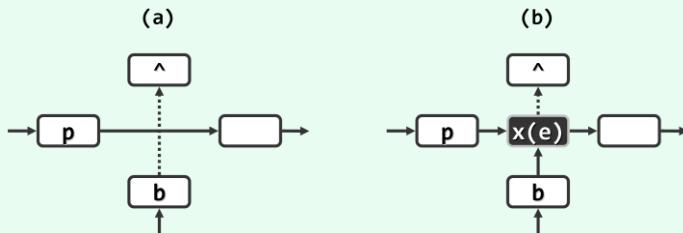


图9.4 四联表节点插入过程

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：

**hash() : key → hash(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

**hash(key) = 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。

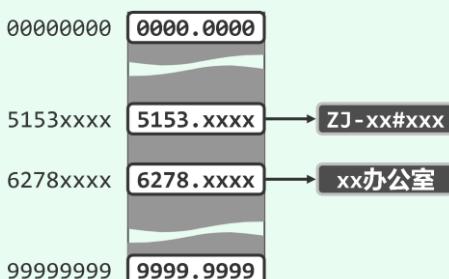


图9.5 直接使用线性数组实现电话簿词典

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 $2^{32} = 4 \times 10^9$ 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<sup>⑨</sup>。而在未来采用IPv6协议之后，尽管实际运

<sup>⑨</sup> 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 $2^{128} = 256 \times 10^{36}$ 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<sup>④</sup>；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：

$$R \gg M > N$$

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 $[0, M)$ 。

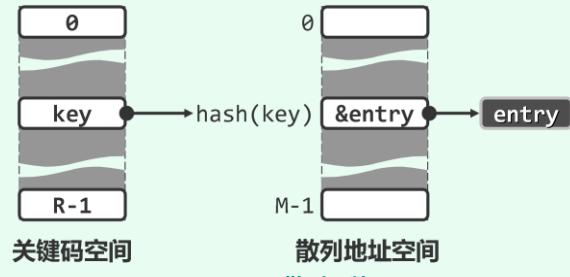


图9.6 散列函数

### ■ 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

<sup>④</sup> 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：

$\text{hash(key)} = \text{key} \bmod M$

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 }

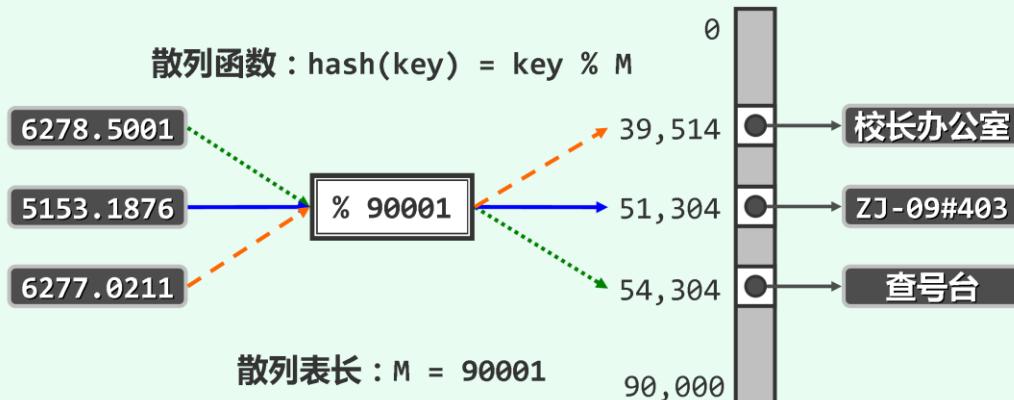


图9.7 除余法

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。

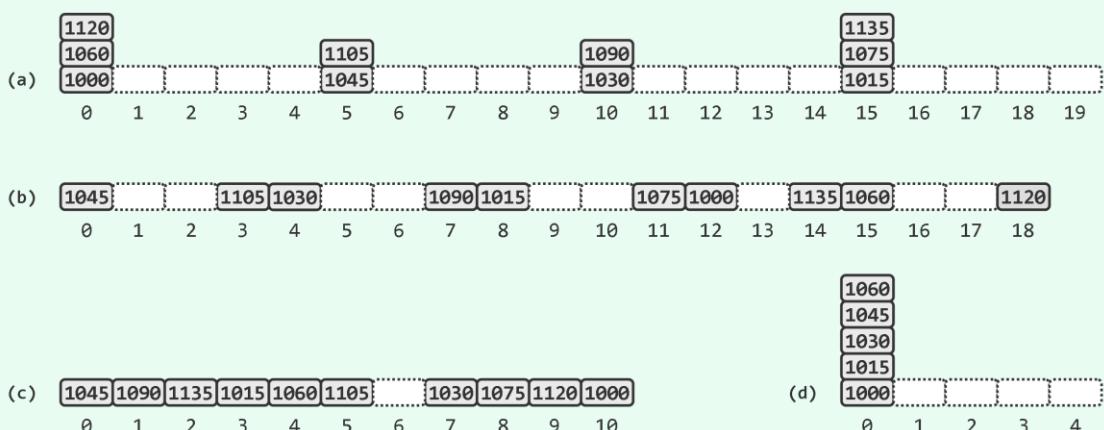


图9.8 素数表长可降低冲突的概率并提高空间的利用率

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) = 45129 = 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<sub>b</sub>) = 110 ^ 011 ^ 011 = 110<sub>b</sub> = 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<sub>b</sub>) = 110 ^ 110 ^ 011 = 011<sub>b</sub> = 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。

```

1 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
2     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
3     while ( c < n ) //从c开始，逐位地
4         if ( B.test ( c ) ) c++; //测试，即可
5     else return c; //返回首个发现的素数
6     return c; //若没有这样的素数，返回n（实用中不能如此简化处理）
7 }

```

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

```

1 template <typename K, typename V> Hashtable<K, V>::~Hashtable() { //析构前释放桶数组及非空词条
2     for ( int i = 0; i < M; i++ ) //逐一检查各桶
3         if ( ht[i] ) release ( ht[i] ); //释放非空的桶
4     release ( ht ); //释放桶数组
5     release ( lazyRemoval ); //释放懒惰删除标记
6 }

```

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。

~	~	~	~	~	~	~	~	~	~	~	~	~	~	~	~	~	~	~	~
1120	~	~	~	~	~	~	~	~	~	~	~	~	~	~	~	1135	~	~	~
1060	~	~	~	~	1105	~	~	~	1090	~	~	~	~	~	~	1075	~	~	~
1000	~	~	~	~	1045	~	~	~	1030	~	~	~	~	~	~	1015	~	~	~

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。

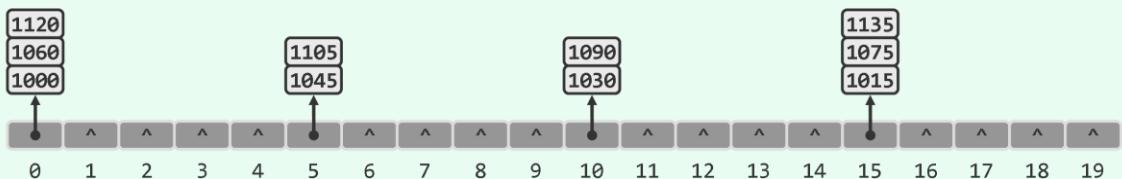


图9.11 利用建立独立链排解散列冲突

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 $D_{\text{overflow}}$ （图(b)），一旦在插入词条时发生冲突就将该词条转存至 $D_{\text{overflow}}$ 中。就效果而言， $D_{\text{overflow}}$ 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。

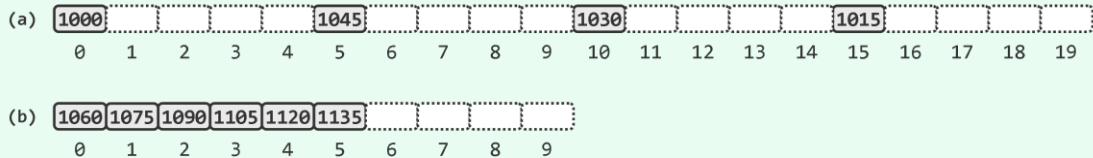


图9.12 利用公共溢出区解决散列冲突

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：

$$ht[(hash(key) + i) \bmod M], \quad i = 1, 2, 3, \dots$$

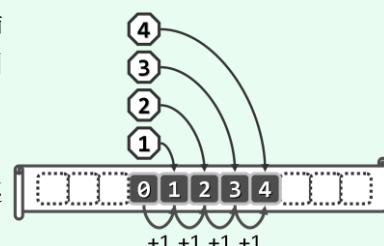


图9.13 线性试探法

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。

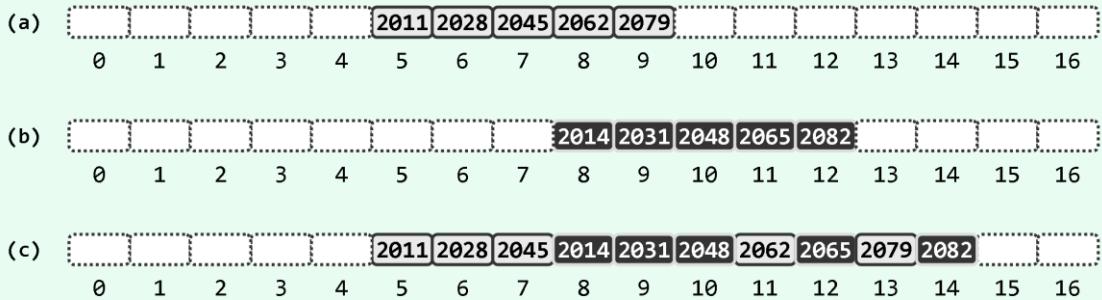


图9.14 线性试探法对应的查找链

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 $\lceil n/2 \rceil$ 。

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。

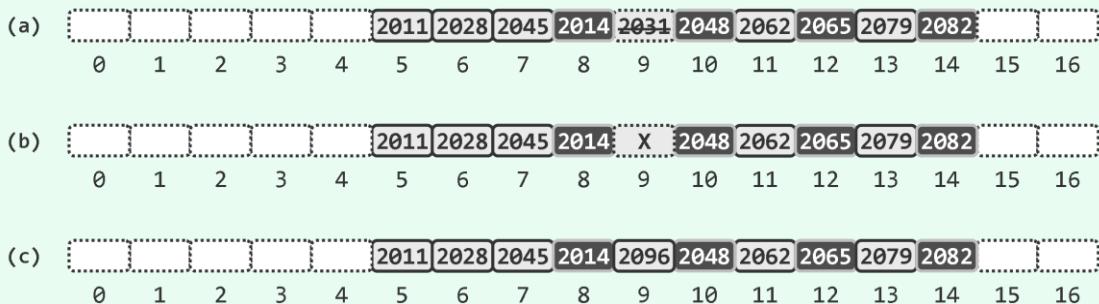


图9.15 通过设置懒惰删除标记，无需大量词条的重排即可保证查找链的完整

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 $\lambda = N / M$ 是最为重要的一个因素。随着 $\lambda$ 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 $\lambda$ 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 $\lambda < 0.5$ 。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。

```

1 /*****
2 * 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
3 * 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：
4 * 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
5 *****/
6 template <typename K, typename V> void Hashtable<K, V>::rehash() {
7     int old_capacity = M; Entry<K, V>* old_ht = ht;
8     M = primeNLT(2 * M, 1048576, ".../_input/prime-1048576-bitmap.txt"); //容量至少加倍
9     N = 0; ht = new Entry<K, V>*[M]; memset(ht, 0, sizeof(Entry<K, V>*)* M); //新桶数组
10    release(lazyRemoval); lazyRemoval = new Bitmap(M); //新开懒惰删除标记比特图
11    for (int i = 0; i < old_capacity; i++) //扫描原桶数组
12        if (old_ht[i]) //将非空桶中的词条逐一
13            put(old_ht[i]->key, old_ht[i]->value); //插入至新的桶数组
14    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
15 }
```

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]`。

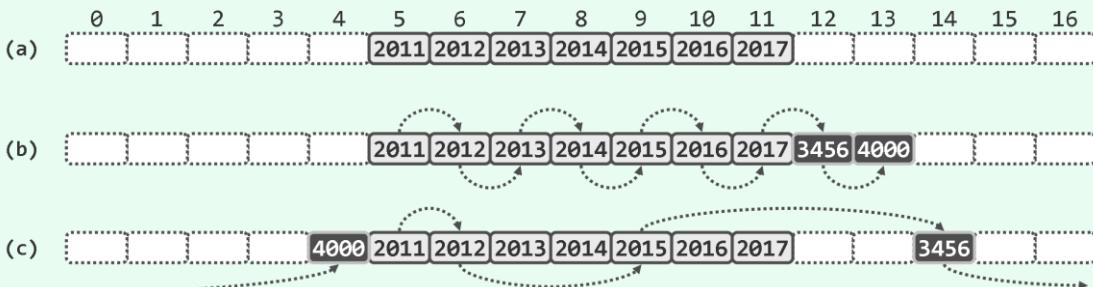


图9.16 线性试探法会加剧聚集现象，而平方试探法则会快速跳离聚集区段

接下来，设拟插入关键码3456和4000。由除余法， $\text{hash}(3456) = \text{hash}(4000) = \text{hash}(2011) = 5$ ，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：

$$(\text{hash}(\text{key}) + j^2) \bmod M, \quad j = 0, 1, 2, \dots$$

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。

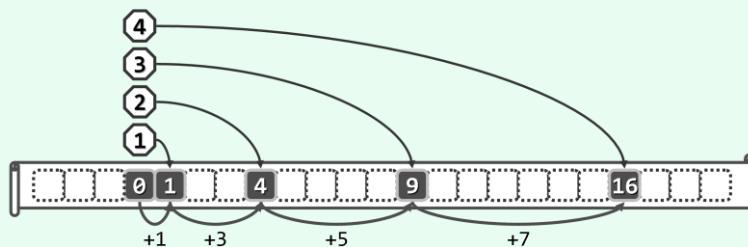


图9.17 平方试探法

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 $\text{ht}[14]$ 。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 $\text{ht}[4]$ 。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 $\lambda > 50\%$ 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 $\text{ht}[0]$ 冲突的词条，并从 $\text{ht}[0]$ 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 $\lambda \leq 50\%$ ，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 $\lambda = 6/11 > 50\%$ 。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`hash2()`，一旦在插入词条(`key, value`)时发现`ht[hash(key)]`已被占用，则以`hash2(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 × hash2(key)] % M`

`[hash(key) + 2 × hash2(key)] % M`

`[hash(key) + 3 × hash2(key)] % M`

...

可见，再散列法是对此前各方法的概括。比如取`hash2(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。

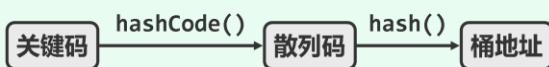


图9.19 分两步将任意类型的关键码，映射为桶地址

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  $1 \sim 26$ ），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 " $x_0x_1\dots x_{n-1}$ " 的散列码取作：

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

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，在此基础上继续补充、扩展和尝试更多的键码类型。

## § 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（ $n \leq M$ ），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 $\Omega(n \log n)$ 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。

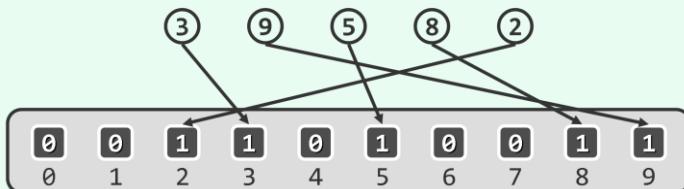


图9.20 利用散列表对一组互异整数排序

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

#### ■ 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？

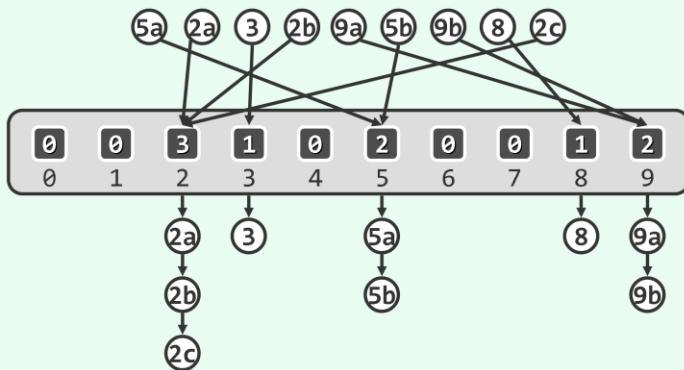


图9.21 利用散列表对一组可能重复的整数排序

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 $n \gg M$ 。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：

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

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 $n \gg M$ 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。

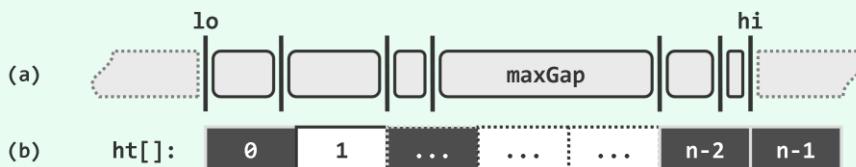


图9.22 利用散列法，在线性时间内确定n个共线点之间的最大间隙

#### ■ 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

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

将各点分别插入对应的桶单元，其中 $x$ 为各点的坐标值， $\text{hash}(x)$ 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：

$$\maxGap \geq w = (hi - lo) / (n - 1)$$

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 $\mathcal{O}(n)$ 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 $\mathcal{O}(n)$ 。故即便在最坏情况下，累计运行时间也不超过 $\mathcal{O}(n)$ 。

## 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 $\{k_t, k_{t-1}, \dots, k_1\}$ 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	4 4 1	2 7 6	3 2 0	2 1 4	6 9 8	2 8 0	1 1 2
以个位排序	3 2 0	2 8 0	4 4 1	1 1 2	2 1 4	2 7 6	6 9 8
以十位排序	1 1 2	2 1 4	3 2 0	4 4 1	2 7 6	2 8 0	6 9 8
以百位排序	1 1 2	2 1 4	2 7 6	2 8 0	3 2 0	4 4 1	6 9 8

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<sub>i</sub>)，1 ≤ i ≤ t。若记

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

则总体运行时间不超过：

$$\begin{aligned} O(n + M_1) + O(n + M_2) + \dots + O(n + M_t) \\ = O(t * (n + M)) \end{aligned}$$

## 第10章

# 优先级队列

此前的搜索树结构和词典结构，都支持覆盖数据全集的访问和操作。也就是说，其中存储的每一数据对象都可作为查找和访问目标。为此，搜索树结构需要在所有元素之间定义并维护一个显式的全序（**full order**）关系；而词典结构中的数据对象之间，尽管不必支持比较大小，但在散列表之类的具体实现中，都从内部强制地在对象的数值与其对应的秩之间，建立起某种关联（尽管实际上这种关联通常越“随机”越好），从而隐式地定义了一个全序次序。

就对外接口的功能而言，本章将要介绍的优先级队列，较之此前的数据结构反而有所削弱。具体地，这类结构将操作对象限定于当前的全局极值者。比如，在全体北京市民中，查找年龄最长者；或者在所有鸟类中，查找种群规模最小者，等等。这种根据数据对象之间相对优先级对其进行访问的方式，与此前的访问方式有着本质区别，称作循优先级访问（**call-by-priority**）。

当然，“全局极值”本身就隐含了“所有元素可相互比较”这一性质。然而，优先级队列并不会也不必忠实地动态维护这个全序，却转而维护一个偏序（**partial order**）关系。其高明之处在于，如此不仅足以高效地支持仅针对极值对象的接口操作，更可有效地控制整体计算成本。正如我们将要看到的，对于常规的查找、插入或删除等操作，优先级队列的效率并不低于此前的结构；而对于数据集的批量构建及相互合并等操作，其性能却更胜一筹。作为不失高效率的轻量级数据结构，优先级队列在许多领域都是扮演着不可替代的角色。

## § 10.1 优先级队列ADT

### 10.1.1 优先级与优先级队列

除了作为存放数据的容器，数据结构还应能够按某种约定的次序动态地组织数据，以支持高效的查找和修改操作。比如4.5节的队列结构，可用以描述和处理日常生活中的很多问题：在银行排队等候接受服务的客户，提交给网络打印机的打印任务等，均属此列。在这类问题中，无论客户还是打印任务，接受服务或处理的次序完全取决于其出现的时刻——先到的客户优先接受服务，先提交的打印任务优先执行——此即所谓“先进先出”原则。

然而在更多实际应用环境中，这一简单公平的原则并不能保证整体效率必然达到最高。试想，若干病人正在某所医院的门诊处排队等候接受治疗，忽然送来一位骨折的病人。要是固守“先进先出”的原则，那么他只能咬牙坚持到目前已经到达的每位病人都已接受过治疗之后。显然，那样的话该病人将承受更长时间的痛苦，甚至贻误治疗的最佳时机。因此，医院在此时都会灵活变通，优先治疗这位骨折的病人。同理，若此时又送来一位心脏病突发的患者，那么医生肯定也会暂时把骨折病人放在一边（如果没有更多医生的话），转而优先抢救心脏病人。

由此可见，在决定病人接受治疗次序时，除了他们到达医院的先后次序，更应考虑到病情的轻重缓急，优先治疗病情最为危重的病人。在数据结构与算法设计中，类似的例子也屡见不鲜。在3.5.3节的选择排序算法中，每一步迭代都要调用**selectMax()**，从未排序区间选出最大者。在5.5.3节的Huffman编码算法中，每一步迭代都要调用**minHChar()**，从当前的森林中选出权重

最小的超字符。在基于空间扫描策略的各种算法中，每一步迭代都要根据到当前扫描线的距离，取出并处理最近的下一个事件点。

从数据结构的角度看，无论是待排序节点的数值、超字符的权重，还是事件的发生时间，数据项的某种属性只要可以相互比较大小，则这种大小关系即可称作优先级（**priority**）。而按照事先约定的优先级，可以始终高效查找并访问优先级最高数据项的数据结构，也统称作优先级队列（**priority queue**）。

### 10.1.2 关键码、比较器与偏序关系

仿照词典结构，我们也将优先级队列中的数据项称作词条（**entry**）；而与特定优先级相对应的数据属性，也称作关键码（**key**）。不同应用中的关键码，特点不尽相同：有时限定词条的关键码须互异，有时则允许词条的关键码雷同；有些词条的关键码一成不变，有些则可动态修改；有的关键码只是一个数字、一个字符或一个字符串，而复杂的关键码则可能由多个基本类型组合而成；多数关键码都取作词条内部的某一成员变量，而有的关键码则并非词条的天然属性。

无论具体形式如何，作为确定词条优先级的依据，关键码之间必须可以比较大小——注意，这与词典结构完全不同，后者仅要求关键码支持判等操作。因此对于优先级队列，必须以比较器的形式兑现对应的优先级关系。出于简化的考虑，与此前各章一样，本章依然假定关键码或者可直接比较，或者已重载了对应的操作符。

需特别留意的另一点是，尽管定义了明确的比较器即意味着在任何一组词条之间定义了一个全序关系，但正如2.7节所指出的，严格地维护这样一个全序关系必将代价不菲。实际上，优先级队列作为一类独特数据结构的意义恰恰在于，通过转而维护词条集的一个偏序关系。如此，不仅依然可以支持对最高优先级词条的动态访问，而且可将相应的计算成本控制在足以令人满意的范围之内。

### 10.1.3 操作接口

优先级队列接口的定义说明如表10.1所示。

表10.1 优先级队列ADT支持的操作接口

操作接口	功能描述
<code>size()</code>	报告优先级队列的规模，即其中词条的总数
<code>insert()</code>	将指定词条插入优先级队列
<code>getMax()</code>	返回优先级最大的词条（若优先级队列非空）
<code>delMax()</code>	删除优先级最大的词条（若优先级队列非空）

需要说明的是，本章允许在同一优先级队列中出现关键码雷同的多个词条，故`insert()`操作必然成功，因此该接口自然不必返回操作成功标志。

### 10.1.4 操作实例：选择排序

即便仍不清楚其具体实现，我们也已经可以按照以上ADT接口，基于优先级队列描述和实现各种算法。比如，实现和改进3.5.3节所介绍的选择排序算法。

具体的构思如下：将待排序的词条组织为一个优先级队列，然后反复调用`delMax()`接口，即可按关键码由大而小的次序逐一输出所有词条，从而得到全体词条的排序序列。

例如，针对某7个整数的这一排序过程，如表10.2所示。

表10.2 优先级队列操作实例：选择排序（当前的最大元素以方框示意）

操作	优 先 级 队 列	输 出
initialization	{ 441, 276, 320, 214, 698, 280, 112 }	
size()	[unchanged]	7
delMax()	{ 441, 276, 320, 214, 280, 112 }	698
size()	[unchanged]	6
delMax()	{ 276, 320, 214, 280, 112 }	441
delMax()	{ 276, 214, 280, 112 }	320
delMax()	{ 276, 214, 112 }	280
delMax()	{ 214, 112 }	276
delMax()	{ 112 }	214
size()	[unchanged]	1
delMax()	{ }	112
size()	[unchanged]	0

### 10.1.5 接口定义

如代码10.1所示，这里以模板类PQ的形式给出以上优先级队列的操作接口定义。

```
1 template <typename T> struct PQ { //优先级队列PQ模板类
2     virtual void insert ( T ) = 0; //按照比较器确定的优先级次序插入词条
3     virtual T getMax() = 0; //取出优先级最高的词条
4     virtual T delMax() = 0; //删除优先级最高的词条
5 };
```

代码10.1 优先级队列标准接口

因为这一组基本的ADT接口可能有不同的实现方式，故这里均以虚函数形式统一描述这些接口，以便在不同的派生类中具体实现。

### 10.1.6 应用实例：Huffman编码树

回到5.4节Huffman编码的应用实例。实际上，基于以上优先级队列的标准接口，即可实现统一的Huffman编码算法——无论优先级队列的具体实现方式如何。

#### ■ 数据结构

为利用统一的优先级队列接口实现Huffman编码并对不同方法进行对比，不妨继续沿用代码5.29至代码5.33所定义的Huffman超字符、Huffman树、Huffman森林、Huffman编码表、Huffman二进制编码串等数据结构。

## ■ 比较器

若将Huffman森林视作优先级队列，则其中每一棵树（每一个超字符）即是一个词条。为保证词条之间可以相互比较，可如代码5.29（145页）所示重载对应的操作符。进一步地，因超字符的优先级可度量为其对应权重的负值，故不妨将大小关系颠倒过来，令小权重超字符的优先级更高，以便于操作接口的统一。

这一技巧也可运用于其它场合。仍以10.1.4节的选择排序为例，在将大小的定义颠倒之后，无需修改其它代码，即可实现反方向的排序。

## ■ 编码算法

经上述准备，代码10.2即可基于统一优先级队列接口给出通用的Huffman编码算法。

```

1 ****
2 * Huffman树构造算法：对传入的Huffman森林forest逐步合并，直到成为一棵树
3 ****
4 * forest基于优先级队列实现，此算法适用于符合PQ接口的任何实现方式
5 * 为Huffman_PQ_List、Huffman_PQ_ComplHeap和Huffman_PQ_LeftHeap共用
6 * 编译前对应工程只需设置相应标志：DSA_PQ_List、DSA_PQ_ComplHeap或DSA_PQ_LeftHeap
7 ****
8 HuffTree* generateTree ( HuffForest* forest ) {
9     while ( 1 < forest->size() ) {
10         HuffTree* s1 = forest->delMax(); HuffTree* s2 = forest->delMax();
11         HuffTree* s = new HuffTree();
12         s->insertAsRoot ( HuffChar ( '^', s1->root()->data.weight + s2->root()->data.weight ) );
13         s->attachAsLC ( s->root(), s1 ); s->attachAsRC ( s->root(), s2 );
14         forest->insert ( s ); //将合并后的Huffman树插回Huffman森林
15     }
16     HuffTree* tree = forest->delMax(); //至此，森林中的最后一棵树
17     return tree; //即全局Huffman编码树
18 }
```

**代码10.2 利用统一的优先级队列接口，实现通用的Huffman编码**

## ■ 效率分析

相对于如代码5.36（147页）所示的版本，这里只不过将`minHChar()`替换为`PQ::delMax()`标准接口。正如我们很快将要看到的，优先级队列的所有ADT操作均可在 $\mathcal{O}(1\log n)$ 时间内完成，故`generateTree()`算法也相应地可在 $\mathcal{O}(n\log n)$ 时间内构造出Huffman编码树——较之原版本，改进显著。同理，通过引入优先级队列，将如代码3.20（81页）所示的`selectMax()`替换为`PQ::delMax()`标准接口，也可自然地将选择排序的性能由 $\mathcal{O}(n^2)$ 改进至 $\mathcal{O}(n\log n)$ 。

自然地，这一结论可以推广至任一需要反复选取优先级最高元素的应用问题，并可直接改进相关算法的时间效率。那么，作为基础性数据结构的优先级队列，是否的确可以保证`getMax()`、`delMax()`和`insert()`等接口效率均为 $\mathcal{O}(1\log n)$ ? 具体地，又应如何实现?

实际上，借助无序列表、有序列表、无序向量或有序向量，都难以同时兼顾`insert()`和`delMax()`操作的高效率（习题[10-1]）。因此，必须另辟蹊径，寻找更为高效的实现方法。

## § 10.2 堆

基于列表或向量等结构的实现方式，之所以无法同时保证`insert()`和`delMax()`操作的高效率，原因在于其对优先级的理解过于机械，以致始终都保存了全体词条之间的全序关系。实际上，尽管优先级队列的确隐含了“所有词条可相互比较”这一条件，但从操作接口层面来看，并不需要真正地维护全序关系。比如执行`delMax()`操作时，只要能够确定全局优先级最高的词条即可；至于次高者、第三高者等其余词条，目前暂时不必关心。

有限偏序集的极值必然存在，故此时借助堆（heap）结构维护一个偏序关系即足矣。堆有多种实现形式，以下首先介绍其中最基本的一种形式——完全二叉堆（complete binary heap）。

### 10.2.1 完全二叉堆

#### ■ 结构性与堆序性

如图10.1实例所示，完全二叉堆应满足两个条件。

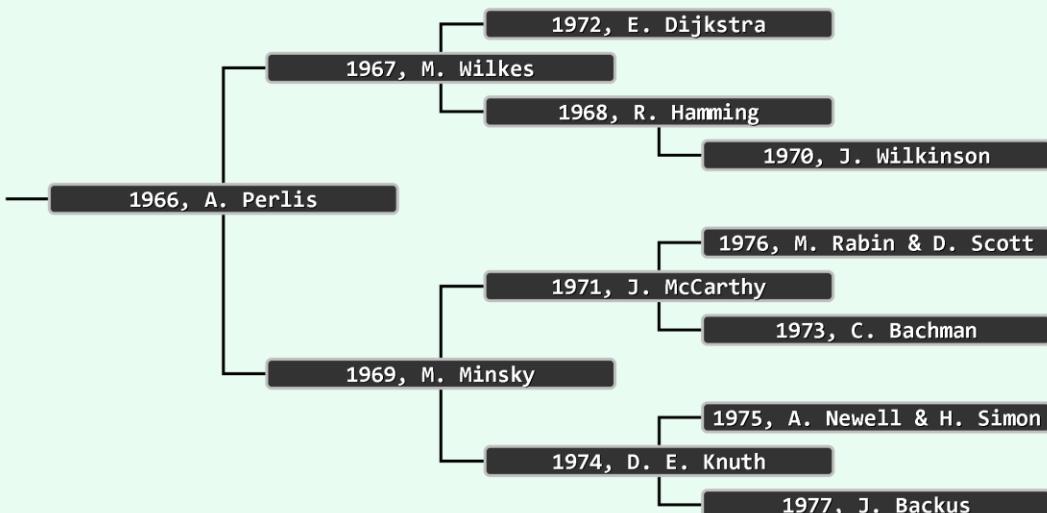


图10.1 以获奖先后为优先级，由前12届图灵奖得主构成的完全二叉堆

首先，其逻辑结构须等同于完全二叉树，此即所谓的“结构性”。如此，堆节点将与词条一一对应，故此后凡不致引起误解时，我们将不再严格区分“堆节点”与“词条”。其次，就优先级而言，堆顶以外的每个节点都不高（大）于其父节点，此即所谓的“堆序性”。

#### ■ 大顶堆与小顶堆

由堆序性不难看出，堆中优先级最高的词条必然始终处于堆顶位置。因此，堆结构的`getMax()`操作总是可以在 $O(1)$ 时间内完成。

堆序性也可对称地约定为“堆顶以外的每个节点都不低（小）于其父节点”，此时同理，优先级最低的词条，必然始终处于堆顶位置。为以示区别，通常称前（后）者为大（小）顶堆。

小顶堆和大顶堆是相对的，而且可以相互转换。实际上，我们不久之前刚刚见过这样的一个实例——在代码5.29中重载Huffman超字符的比较操作符时，通过对超字符权重取负，颠倒优先级关系，使之与算法的实际语义及需求相吻合。

## ■ 高度

结构等同于完全二叉树的堆，必然不致太高。具体地，由5.5.2节的分析结论， $n$ 个词条组成的堆的高度  $h = \lfloor \log_2 n \rfloor = \mathcal{O}(\log n)$ 。稍后我们即将看到，`insert()`和`delMax()`操作的时间复杂度将线性正比于堆的高度  $h$ ，故它们均可在  $\mathcal{O}(\log n)$  的时间内完成。

## ■ 基于向量的紧凑表示

尽管二叉树不属于线性结构，但作为其特例的完全二叉树，却与向量有着紧密的对应关系。

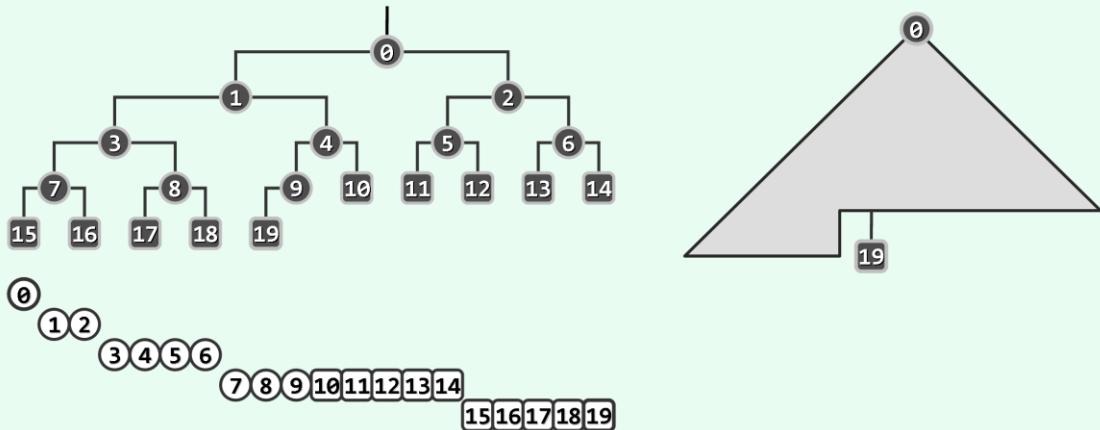


图10.2 按照层次遍历序列，对完全二叉树节点做编号（其中圆形表示内部节点，方形表示外部节点）

由图10.2可见，完全二叉堆的拓扑联接结构，完全由其规模  $n$  确定。按照层次遍历的次序，每个节点都对应于唯一的编号；反之亦然。故若将所有节点组织为一个向量，则堆中各节点（编号）与向量各单元（秩）也将彼此一一对应！

这一实现方式的优势首先体现在，各节点在物理上连续排列，故总共仅需  $\mathcal{O}(n)$  空间。而更重要地是，利用各节点的编号（或秩），也可便捷地判别父子关系。

具体地，若将节点  $v$  的编号（秩）记作  $i(v)$ ，则根节点及其后代节点的编号分别为：

```
i(root) = 0
i(lchild(root)) = 1
i(rchild(root)) = 2
i(lchild(lchild(root))) = 3
```

...

更一般地，不难验证，完全二叉堆中的任意节点  $v$ ，必然满足：

- 1) 若  $v$  有左孩子，则  $i(lchild(v)) = 2 \cdot i(v) + 1$ ;
- 2) 若  $v$  有右孩子，则  $i(rchild(v)) = 2 \cdot i(v) + 2$ ;
- 3) 若  $v$  有父节点，则  $i(parent(v)) = \lfloor (i(v) - 1)/2 \rfloor = \lceil (i(v)/2) \rceil - 1$

最后，由于向量支持低分摊成本的扩容调整，故随着堆的规模和内容不断地动态调整，除标准接口以外的操作所需的时间可以忽略不计。

所有这些良好的性质，不仅为以下基于向量实现堆结构提供了充足的理由，同时也从基本的原理和方法的层面提供了有力的支持。

## ■ 宏

为简化后续算法的描述及实现，可如代码10.3所示预先设置一系列的宏定义。

```

1 #define InHeap(n, i)      (( ( -1 ) < ( i ) ) && ( ( i ) < ( n ) ) ) //判断PQ[i]是否合法
2 #define Parent(i)         ( ( i - 1 ) >> 1 ) //PQ[i]的父节点( floor((i-1)/2), i无论正负 )
3 #define LastInternal(n)   Parent( n - 1 ) //最后一个内部节点( 即末节点的父亲 )
4 #define LChild(i)          ( 1 + ( ( i ) << 1 ) ) //PQ[i]的左孩子
5 #define RChild(i)          ( ( 1 + ( i ) ) << 1 ) //PQ[i]的右孩子
6 #define ParentValid(i)     ( 0 < i ) //判断PQ[i]是否有父亲
7 #define LChildValid(n, i)   InHeap( n, LChild( i ) ) //判断PQ[i]是否有一个( 左 )孩子
8 #define RChildValid(n, i)   InHeap( n, RChild( i ) ) //判断PQ[i]是否有两个孩子
9 #define Bigger(PQ, i, j)    ( lt( PQ[i], PQ[j] ) ? j : i ) //取大者( 等时前者优先 )
10 #define ProperParent(PQ, n, i) /*父子( 至多 )三者中的大者*/ \
11           ( RChildValid(n, i) ? Bigger( PQ, Bigger( PQ, i, LChild(i) ), RChild(i) ) : \
12             ( LChildValid(n, i) ? Bigger( PQ, i, LChild(i) ) : i \
13           ) \
14         ) //相等时父节点优先，如此可避免不必要的交换

```

代码10.3 为简化完全二叉堆算法的描述及实现而定义的宏

## ■ PQ\_CmplHeap模板类

按照以上思路，可以借助多重继承的机制，定义完全二叉堆模板类如代码10.4所示。

```

1 #include "../Vector/Vector.h" //借助多重继承机制，基于向量
2 #include "../PQ/PQ.h" //按照优先级队列ADT实现的
3 template <typename T> class PQ_CmplHeap : public PQ<T>, public Vector<T> { //完全二叉堆
4 protected:
5     Rank percolateDown ( Rank n, Rank i ); //下滤
6     Rank percolateUp ( Rank i ); //上滤
7     void heapify ( Rank n ); //Floyd建堆算法
8 public:
9     PQ_CmplHeap() { } //默认构造
10    PQ_CmplHeap ( T* A, Rank n ) { copyFrom ( A, 0, n ); heapify ( n ); } //批量构造
11    void insert ( T ); //按照比较器确定的优先级次序，插入词条
12    T getMax(); //读取优先级最高的词条
13    T delMax(); //删除优先级最高的词条
14 }; //PQ_CmplHeap

```

代码10.4 完全二叉堆接口

## ■ getMax()

既然全局优先级最高的词条总是位于堆顶，故如代码10.5所示，只需返回向量的首单元，即可在 $O(1)$ 时间内完成getMax()操作。

```
1 template <typename T> T PQ_CmplHeap<T>::getMax() { return _elem[0]; } //取优先级最高的词条
```

代码10.5 完全二叉堆getMax()接口

### 10.2.2 元素插入

本节介绍插入操作`insert()`的实现。因堆中的节点与其中所存词条以及词条的关键码完全对应，故沿用此前的习惯，在不致歧义的前提下，以下对它们将不再严格区分。

#### ■ 算法

如代码10.6所示，插入算法分为两个步骤。

```
1 template <typename T> void PQ_CmplHeap<T>::insert ( T e ) { //将词条插入完全二叉堆中
2     Vector<T>::insert ( e ); //首先将新词条接至向量末尾
3     percolateUp ( _size - 1 ); //再对该词条实施上滤调整
4 }
```

代码10.6 完全二叉堆`insert()`接口的主体框架

首先，调用向量的标准插入接口，将新词条接至向量的末尾。得益于向量结构良好的封装性，这里无需关心这一步骤的具体细节，尤其是无需考虑溢出扩容等特殊情况。

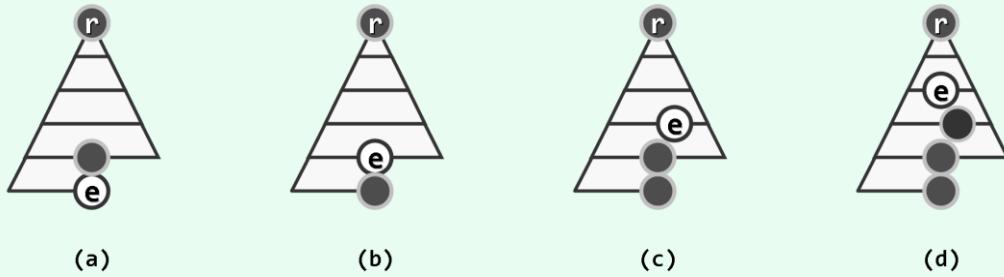


图10.3 完全二叉堆词条插入过程

尽管此时如图10.3(a)所示，新词条的引入并未破坏堆的结构性，但只要新词条`e`不是堆顶，就有可能与其父亲违反堆序性。

当然，其它位置的堆序性依然满足。故以下将调用`percolateUp()`函数，对新接入的词条做适当调整，在保持结构性的前提下恢复整体的堆序性。

#### ■ 上滤

不妨假定原堆非空，于是新词条`e`的父亲`p`（深色节点）必然存在。根据`e`在向量中对应的秩，可以简便地确定词条`p`对应的秩，即 $i(p) = \lfloor (i(e) - 1)/2 \rfloor$ 。

此时，若经比较判定 $e \leq p$ ，则堆序性在此局部以至全堆均已满足，插入操作因此即告完成。反之，若 $e > p$ ，则可在向量中令`e`和`p`互换位置。如图10.3(b)所示，如此不仅全堆的结构性依然满足，而且`e`和`p`之间的堆序性也得以恢复。

当然，此后`e`与其新的父亲，可能再次违背堆序性。若果真如此，不妨继续套用以上方法，如图10.3(c)所示令二者交换位置。当然，只要有必要，此后可以不断重复这种交换操作。

每交换一次，新词条`e`都向上攀升一层，故这一过程也形象地称作上滤（`percolate up`）。当然，`e`至多上滤至堆顶。一旦上滤完成，则如图10.3(d)所示，全堆的堆序性必将恢复。

由上可见，上滤调整过程中交换操作的累计次数，不致超过全堆的高度 $\lceil \log_2 n \rceil$ 。而在向量中，每次交换操作只需常数时间，故上滤调整乃至整个词条插入算法整体的时间复杂度，均为 $O(\log n)$ 。这也是从一个方面，兑现了10.1节末尾就优先级队列性能所做的承诺。

### ■ 最坏情况与平均情况

当然，不难通过构造实例说明，新词条有时的确需要一直上滤至堆顶。然而实际上，此类最坏情况通常极为罕见。以常规的随机分布而言，新词条平均需要爬升的高度，要远远低于直觉的估计（习题[10-6]）。在此类场合中，优先级队列相对于其它数据结构的性能优势，也因这一特性得到了进一步的巩固。

### ■ 实例

通过上滤调整实现插入操作的一个实例，如图10.4所示。图中上方为完全堆的拓扑联接结构，下方为物理上与之对应的线性存储结构。

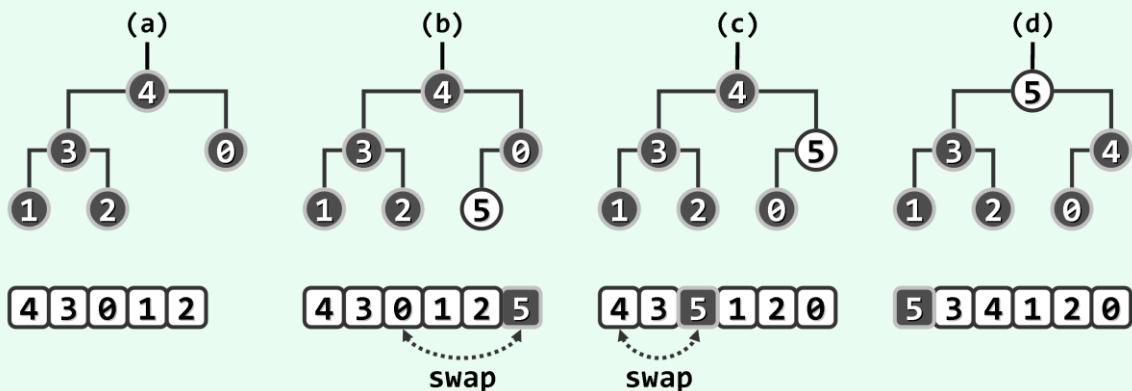


图10.4 完全二叉堆词条插入操作实例

在如图(a)所示由5个元素组成的初始完全堆中，现拟插入关键码为5的新元素。为此，首先如图(b)所示，将该元素置于向量的末尾。此时，新元素5与其父节点0逆序，故如图(c)所示，经一次交换之后，新元素5上升一层。此后，新元素5与其新的父节点4依然逆序，故如图(d)所示，经一次交换后再上升一层。此时因已抵达堆顶，插入操作完毕，故算法终止。

### ■ 实现

以上调整在向量中的具体操作过程，可描述和实现如代码10.7所示。

```

1 //对向量中的第i个词条实施上滤操作, i < _size
2 template <typename T> Rank PQ_CmplHeap<T>::percolateUp ( Rank i ) {
3     while ( ParentValid ( i ) ) { //只要i有父亲(尚未抵达堆顶), 则
4         Rank j = Parent ( i ); //将i之父记作j
5         if ( lt ( _elem[i], _elem[j] ) ) break; //一旦当前父子不再逆序, 上滤旋即完成
6         swap ( _elem[i], _elem[j] ); i = j; //否则, 父子交换位置, 并继续考查上一层
7     } //while
8     return i; //返回上滤最终抵达的位置
9 }
```

代码10.7 完全二叉堆的上滤

其中为简化描述而使用的`Parent()`、`ParentValid()`等快捷方式，均以宏的形式定义如代码10.3所示。

需说明的是，若仅考虑插入操作，则因被调整词条的秩总是起始于n - 1，故无需显式地指

定输入参数*i*。然而，考虑到上滤调整可能作为一项基本操作用于其它场合（习题[10-12]），届时被调整词条的秩可能任意，故为保持通用性，这里不妨保留一项参数以指定具体的起始位置。

### ■ 改进

在如代码10.7所示的版本中，最坏情况下在每一层次都要调用一次`swap()`，该操作通常包含三次赋值。实际上，只要注意到，参与这些操作的词条之间具有很强的相关性，则不难改进为平均每层大致只需一次赋值（习题[10-3]）；而若能充分利用内部向量“循秩访问”的特性，则大小比较操作的次数甚至可以更少（习题[10-4]）。

## 10.2.3 元素删除

### ■ 算法

下面再来讨论`delMax()`方法的实现。如代码10.8所示，删除算法也分为两个步骤。

```
1 template <typename T> T PQ_CmplHeap<T>::delMax() { //删除非空完全二叉堆中优先级最高的词条
2     T maxElem = _elem[0]; _elem[0] = _elem[ --_size ]; //摘除堆顶（首词条），代之以末词条
3     percolateDown ( _size, 0 ); //对新堆顶实施下滤
4     return maxElem; //返回此前备份的最大词条
5 }
```

代码10.8 完全二叉堆`delMax()`接口的主体框架

首先，既然待删除词条*r*总是位于堆顶，故可直接将其取出并备份。此时如图10.5(a)所示，堆的结构性将被破坏。为修复这一缺陷，可如图(b)所示，将最末尾的词条*e*转移至堆顶。

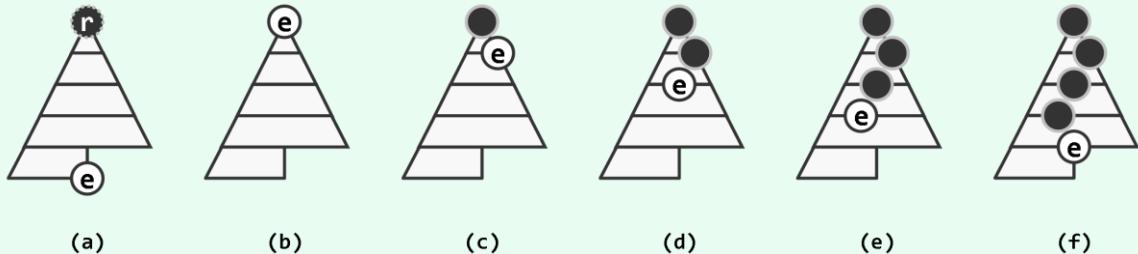


图10.5 完全二叉堆词条删除过程

当然，新的堆顶可能与其孩子（们）违背堆序性——尽管其它位置的堆序性依然满足。故以下调用`percolateDown()`函数调整新堆顶，在保持结构性的前提下，恢复整体的堆序性。

### ■ 下滤

若新堆顶*e*不满足堆序性，则可如图10.5(c)所示，将*e*与其（至多）两个孩子中的大者（图中深色节点）交换位置。与上滤一样，由于使用了向量来实现堆，根据词条*e*的秩可便捷地确定其孩子的秩。此后，堆中可能的缺陷依然只能来自于词条*e*——它与新孩子可能再次违背堆序性。若果真如此，不妨继续套用以上方法，将*e*与新孩子中的大者交换，结果如图(d)所示。实际上，只要有必要，此后可如图(e)和(f)不断重复这种交换操作。

因每经过一次交换，词条*e*都会下降一层，故这一调整过程也称作下滤（*percolate down*）。与上滤同理，这一过程也必然终止。届时如图(f)所示，全堆的堆序性必将恢复；而且，下滤乃至整个删除算法的时间复杂度也为 $\mathcal{O}(\log n)$ ——同样，这从另一方面兑现了此前的承诺。

### ■ 实例

通过下滤变换实现删除操作的一个实例，如图10.6所示。同样地，图中上方和下方分别为完全堆的拓扑结构以及对应的线性存储结构。

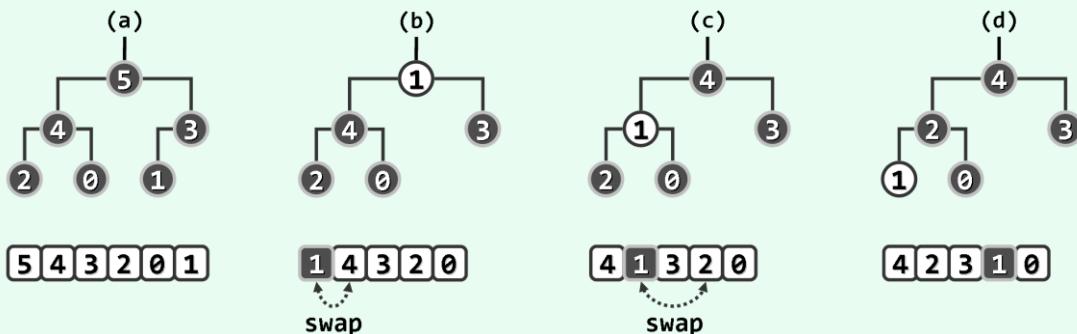


图10.6 完全二叉堆词条删除操作实例

从如图(a)所示由6个元素组成的完全堆中，现拟删除堆顶元素5。为此，首先如图(b)所示将该元素摘除，并将向量的末元素1转入首单元，权作堆顶。此后，1与其孩子节点均逆序。故如图(c)所示，在与其孩子中的大者4交换之后，1下降一层。此后，1与其新的孩子2依然逆序，故如图(d)所示经又一次交换后再下降一层。此时因1已抵达底层，删除操作完毕，算法成功终止。

### ■ 实现

以上调整在向量中的具体操作过程，可描述和实现如代码10.9所示。

```

1 //对向量前n个词条中的第i个实施下滤, i < n
2 template <typename T> Rank PQ_CmplHeap<T>::percolateDown ( Rank n, Rank i ) {
3     Rank j; //i及其(至多两个)孩子中, 堪为父者
4     while ( i != ( j = ProperParent ( _elem, n, i ) ) ) //只要i非j, 则
5         { swap ( _elem[i], _elem[j] ); i = j; } //二者换位, 并继续考查下降后的i
6     return i; //返回下滤抵达的位置(亦i亦j)
7 }
```

代码10.9 完全二叉堆的下滤

这里为简化算法描述使用了宏ProperParent()，其定义如288页代码10.3所示。

出于与上滤操作同样的考虑（习题[10-12]），这里也可通过输入参数*i*，灵活地指定起始位置。此前针对上滤操作所建议的改进方法，有的也同样适用于下滤操作（习题[10-3]），但有的却不再适用（习题[10-4]）。

### 10.2.4 建堆

很多算法中输入词条都是成批给出，故在初始化阶段往往需要解决一个共同问题：给定一组词条，高效地将它们组织成一个堆。这一过程也称作“建堆”（**heapification**）。本节就以完全二叉堆为例介绍相关的算法。当然，以下算法同样也适用其它类型的堆。

### ■ 蛮力算法

乍看起来，建堆似乎并不成其为一个问题。既然堆符合优先级队列ADT规范，那么从空堆起

反复调用标准`insert()`接口，即可将输入词条逐一插入其中，并最终完成建堆任务。很遗憾，尽管这一方法无疑正确，但其消耗的时间却过多。具体地，若共有n个词条，则共需迭代n次。由10.2.2节的结论，第k轮迭代耗时 $\mathcal{O}(\log k)$ ，故累计耗时间量应为：

$$\mathcal{O}(\log 1 + \log 2 + \dots + \log n) = \mathcal{O}(\log n!) = \mathcal{O}(n \log n)$$

或许对某些具体问题而言，后续操作所需的时间比这更多（或至少不更少），以致建堆操作是否优化对总体复杂度无实质影响。但换个角度看，如此多的时间本来足以对所有词条做全排序，而在这里花费同样多时间所生成的堆却只能提供一个偏序。这一事实在某种程度上也暗示着，或许存在某种更快的建堆算法。此外，的确有些算法的总体时间复杂度主要取决于堆初始化阶段的效率，因此探索并实现复杂度为 $\mathcal{O}(n \log n)$ 的建堆算法也十分必要。

### ■ 自上而下的上滤

尽管蛮力算法的效率不尽如人意，其实现过程仍值得分析和借鉴。在将所有输入词条纳入长为n的向量之后，首单元处的词条本身即可视作一个规模为1的堆。接下来，考查下一单元中的词条。不难看出，为将该词条插入当前堆，只需针对调用`percolateUp()`对其上滤。此后，前两个单元将构成规模为2的堆。以下同理，若再对第三个词条上滤，则前三个单元将构成规模为3的堆。实际上，这一过程可反复进行，直到最终得到规模为n的堆。

这一过程可归纳为：对任何一棵完全二叉树，只需自顶而下、自左向右地针对其中每个节点实施一次上滤，即可使之成为完全二叉堆。在此过程中，为将每个节点纳入堆中，所需消耗的时间量将线性正比于该节点的深度。不妨考查高度为h、规模为 $n = 2^{h+1} - 1$ 的满二叉树，其中高度为i的节点共有 $2^i$ 个，因此整个算法的总体时间复杂度应为：

$$\sum_{i=0}^h (i \cdot 2^i) = (d - 1) \times 2^{d+1} + 2 = (\log_2(n + 1) - 2) \cdot (n + 1) + 2 = \mathcal{O}(n \log n)$$

与上面的分析结论一致。

### ■ Floyd算法

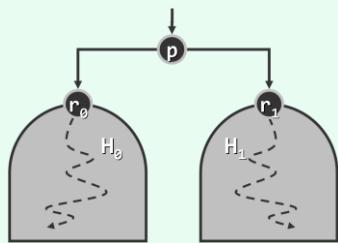


图10.7 堆合并算法原理

为得到更快的建堆算法，先考查一个相对简单的问题：任给堆 $H_0$ 和 $H_1$ ，以及另一独立节点p，如何高效地将 $H_0 \cup \{p\} \cup H_1$ 转化为堆？从效果来看，这相当于以p为中介将堆 $H_0$ 和 $H_1$ 合二为一，故称作堆合并操作。

如图10.7，首先为满足结构性，可将这两个堆当作p的左、右子树，联接成一棵完整的二叉树。此时若p与孩子 $r_0$ 和 $r_1$ 满足堆序性，则该二叉树已经就是一个不折不扣的堆。

实际上，此时的场景完全等效于，在`delMax()`操作中摘除堆顶，再将末位词条（p）转移至堆顶。故仿照10.2.3节的方法，以下只需对p实施下滤操作，即可将全树转换为堆。

如果将以上过程作为实现堆合并的一个通用算法，则在将所有词条组织为一棵完全二叉树后，只需自底而上地反复套用这一算法，即可不断地将处于下层的堆捉对地合并成更高一层的堆，并最终得到一个完整的堆。按照这一构思，即可实现Floyd建堆算法<sup>①</sup>。

<sup>①</sup> 由R. W. Floyd于1964年发明<sup>[57]</sup>

## ■ 实现

上述Floyd算法，可以描述和实现如代码10.10所示。

```
1 template <typename T> void PQ_CmplHeap<T>::heapify ( Rank n ) { //Floyd建堆算法，O(n)时间
2     for ( int i = LastInternal ( n ); InHeap ( n, i ); i-- ) //自底而上，依次
3         percolateDown ( n, i ); //下滤各内部节点
4 }
```

代码10.10 Floyd建堆算法

可见，该算法的实现十分简洁：只需自下而上、由深而浅地遍历所有内部节点，并对每个内部节点分别调用一次下滤算法`percolateDown()`（代码10.9）。

## ■ 实例

图10.8为Floyd算法的一个实例。首先如图(a)所示，将9个词条组织为一棵完全二叉树。多数情况下，输入词条集均以向量形式给出，故除了通过各单元的秩明确对应的父子关系外，并不需要做任何实质的操作。

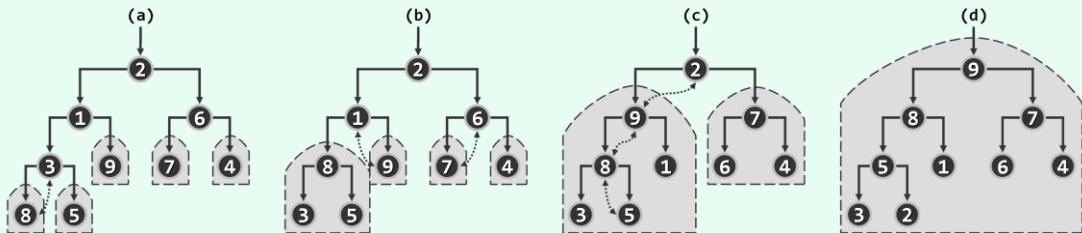


图10.8 Floyd算法实例（虚线示意下滤过程中的交换操作）

此时，所有叶节点各自即是一个堆——尽管其规模仅为1。以下，自底而上地逐层合并。

首先如图(b)所示，在对3实施下滤调整之后，{ 8 }和{ 5 }合并为{ 8, 3, 5 }。接下来如图(c)所示，在对1实施下滤调整之后，{ 8, 3, 5 }与{ 9 }合并为{ 9, 8, 1, 3, 5 }；在对6实施下滤调整之后，{ 7 }与{ 4 }合并为{ 7, 6, 4 }；最后如图(d)所示，在对2实施下滤调整之后，{ 9, 8, 1, 3, 5 }与{ 7, 6, 4 }合并为{ 9, 8, 7, 5, 1, 6, 4, 3, 2 }。

从算法推进的方向来看，前述蛮力算法与Floyd算法恰好相反——若将前者理解为“自上而下的上滤”，则后者即是“自下而上的下滤”。那么，这一细微的差异，是否会对总体时间复杂度产生实质的影响呢？

## ■ 复杂度

由代码10.10可见，算法依然需做n步迭代，以对所有节点各做一次下滤。这里，每个节点的下滤所需的时间线性正比于其深度，故总体运行时间取决于各节点的高度总和。

不妨仍以高度为h、规模为 $n = 2^{h+1} - 1$ 的满二叉树为例做一大致估计，运行时间应为：

$$\sum_{i=0}^h ((d - i) \cdot 2^i) = 2^{d+1} - (d + 2) = n - \log_2(n + 1) = O(n)$$

由于在遍历所有词条之前，绝不可能确定堆的结构，故以上已是建堆操作的最优算法。

由此反观，蛮力算法低效率的根源，恰在于其“自上而下的上滤”策略。如此，各节点所消耗的时间线性正比于其深度——而在完全二叉树中，深度小的节点，远远少于高度小的节点。

### 10.2.5 就地堆排序

本节讨论完全二叉堆的另一具体应用：对于向量中的 $n$ 个词条，如何借助堆的相关算法，实现高效的排序。相应地，这类算法也称作堆排序（heapsort）算法。

既然此前归并排序等算法的渐进复杂度已达到理论上最优的 $\Theta(n \log n)$ ，故这里将更关注于如何降低复杂度常系数——在一般规模的应用中，此类改进的实际效果往往相当可观。同时，我们也希望空间复杂度能够有所降低，最好是除输入本身以外只需 $O(1)$ 辅助空间。

若果真如此，则不妨按照1.3.1节的定义称之为就地堆排序（in-place heapsort）算法。

#### ■ 原理

算法的总体思路和策略与选择排序算法（3.5.3节）基本相同：将所有词条分成未排序和已排序两类，不断从前一类中取出最大者，顺序加至后一类中。算法启动之初，所有词条均属于前一类；此后，后一类不断增长；当所有词条都已转入后一类时，即完成排序。

这里的待排序词条既然已组织为向量，不妨将其划分为前缀H和与之互补的后缀S，分别对应于上述未排序和已排序部分。与常规选择排序算法一样，在算法启动之初H覆盖所有词条，而S为空。新算法的不同之处在于，整个排序过程中，无论H包含多少词条，始终都组织为一个堆。另外，整个算法过程始终满足如下不变性：H中的最大词条不会大于S中的最小词条——除非二者之一为空，比如算法的初始和终止时刻。算法的迭代过程如图10.9所示。

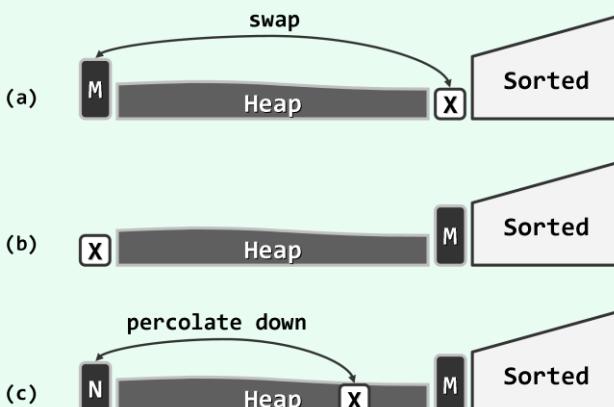


图10.9 就地堆排序

首先如图(a)，取出首单元词条M，将其与末单元词条X交换。M既是当前堆中的最大者，同时根据不变性也不大于S中的任何词条，故如此交换之后M必处于正确的排序位置。故如图(b)，此时可等效地认为S向前扩大了一个单元，H相应地缩小了一个单元。请注意，如此重新分界之后的H和S依然满足以上不变性。至此，唯一尚未解决的问题是，词条X通常不能“胜任”堆顶的角色。

好在这并非难事。仿照此前的词条删除算法（代码10.8），只需对X实施一次下滤调整，即可使H整体的堆序性重新恢复，结果如图(c)所示。

#### ■ 复杂度

在每一步迭代中，交换M和X只需常数时间，对X的下滤调整不超过 $O(\log n)$ 时间。因此，全部n步迭代累计耗时不超过 $O(n \log n)$ 。即便使用蛮力算法而不是Floyd算法来完成H的初始化，整个算法的运行时间也不超过 $O(n \log n)$ 。纵览算法的整个过程，除了用于支持词条交换的一个辅助单元，几乎不需要更多的辅助空间，故的确属于就地算法。

得益于向量结构的简洁性，几乎所有以上操作都可便捷地实现，因此该算法不仅可简明地编码，其实际运行效率也因此往往要高于其它 $O(n \log n)$ 的算法。高运行效率、低开发成本以及低资源消耗等诸多优点的完美结合，若离开堆这一精巧的数据结构实在难以想象。

### ■ 实例

试考查利用以上算法，对向量{ 4, 2, 5, 1, 3 }的堆排序过程。首先如图10.10所示，采用Floyd算法将该向量整理为一个完全二叉堆。其中虚线示意下滤过程中的词条交换操作。

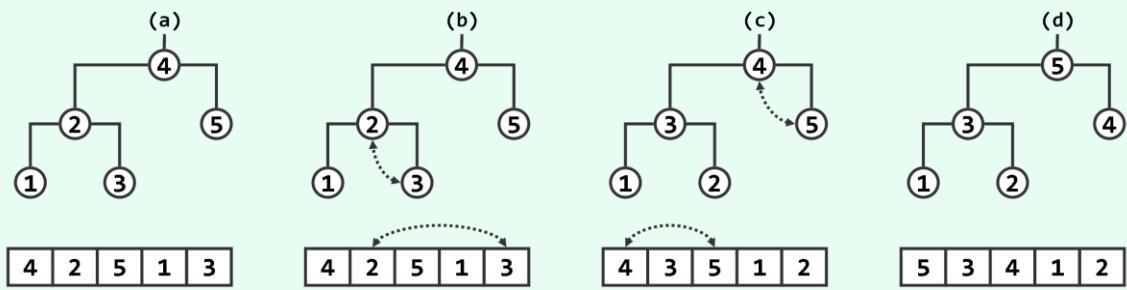


图10.10 就地堆排序实例：建堆

以下如图10.11所示共需5步迭代。请对照以上算法描述，验证各步迭代的具体过程。

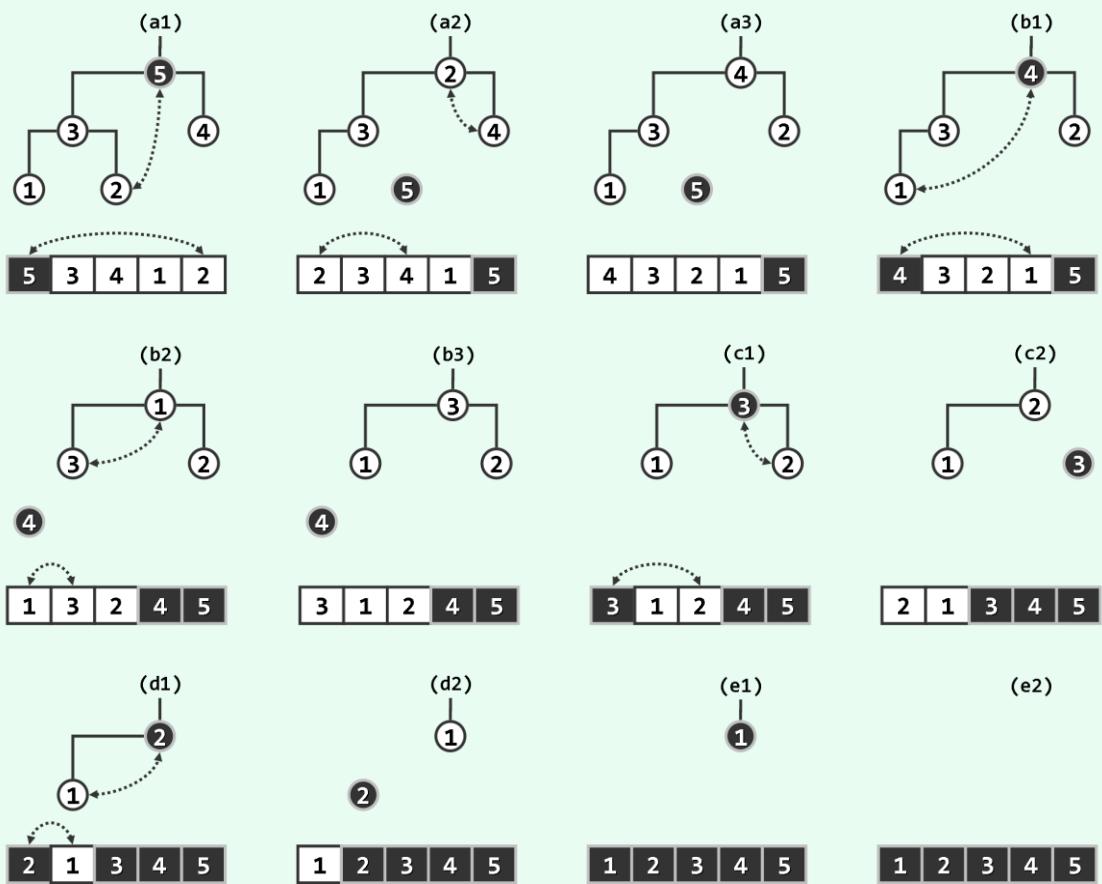


图10.11 就地堆排序实例：迭代

### ■ 实现

按照以上思路，可基于向量排序器的统一规范，实现就地堆排序算法如代码10.11所示。

```

1 template <typename T> void Vector<T>::heapSort ( Rank lo, Rank hi ) { //0 <= lo < hi <= size
2     PQ_CmplHeap<T> H ( _elem + lo, hi - lo ); //将待排序区间建成一个完全二叉堆，O(n)
3     while ( !H.empty() ) //反复地摘除最大元并归入已排序的后缀，直至堆空
4         _elem[--hi] = H.delMax(); //等效于堆顶与末元素对换后下滤
5 }

```

代码10.11 基于向量的就地堆排序

遵照向量接口的统一规范（60页代码2.25），这里允许在向量中指定待排序区间 $[lo, hi]$ ，从而作为通用排序算法具有更好的灵活性。

## § 10.3 \*左式堆

### 10.3.1 堆合并

除了标准的插入和删除操作，堆结构在实际应用中的另一常见操作即为合并。如图10.12，这一操作可描述为：任给堆A和堆B，如何将二者所含的词条组织为一个堆。

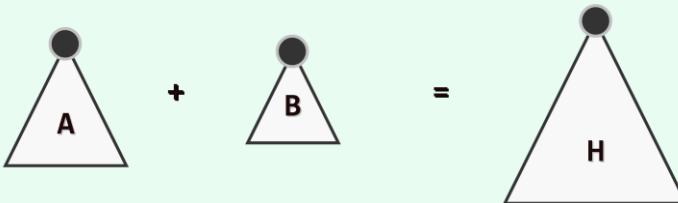


图10.12 堆合并

直接借助已有的接口不难完成这一任务。比如，首先易想到的一种方法是：反复取出堆B的最大词条并插入堆A中；当堆B为空时，堆A即为所需的堆H。这一过程可简洁地描述为：

```

1 while ( ! B.empty() )
2     A.insert( B.delMax() );

```

将两个堆的规模分别记作n和m，且 $n \geq m$ 。每一步迭代均需做一次删除操作和一次插入操作，分别耗时 $\mathcal{O}(\log m)$ 和 $\mathcal{O}(\log(n + m))$ 。因共需做m步迭代，故总体运行时间应为：

$$m \times [\mathcal{O}(\log m) + \mathcal{O}(\log(n + m))] = \mathcal{O}(m \log(n + m)) = \mathcal{O}(m \log n)$$

另一容易想到的方法是：将两个堆中的词条视作彼此独立的对象，从而可以直接借助Floyd算法，将它们组织为一个新的堆H。由10.2.4节的结论，该方法的运行时间应为：

$$\mathcal{O}(n + m) = \mathcal{O}(n)$$

尽管其性能稍优于前一方法，但仍无法令人满意。实际上我们注意到，既然所有词条已分两组各自成堆，则意味着它们已经具有一定的偏序性；而一组相互独立的词条，谈不上具有什么偏序性。按此理解，由前者构建一个更大的偏序集，理应比由后者构建偏序集更为容易。

以上尝试均未奏效的原因在于，不能保证合并操作所涉及的节点足够少。为此，不妨首先打破此前形成的错觉并大胆质疑：**堆是否也必须与二叉搜索树一样，尽可能地保持平衡？**值得玩味的是，对于堆来说，为控制合并操作所涉及的节点数，反而需要保持某种意义上的“不平衡”！

### 10.3.2 单侧倾斜

左式堆<sup>②</sup> (**leftist heap**) 是优先级队列的另一实现方式，可高效地支持堆合并操作。其基本思路是：在保持堆序性的前提下附加新的条件，使得在堆的合并过程中，只需调整很少量的节点。具体地，需参与调整的节点不超过 $\mathcal{O}(\log n)$ 个，故可达到极高的效率。

具体地如图10.13所示，左式堆的整体结构呈单侧倾斜状；依照惯例，其中节点的分布均偏向左侧。也就是说，左式堆将不再如完全二叉堆那样满足结构性。

这也不难理解，毕竟堆序性才是堆结构的关键条件，而结构性只不过是堆的一项附加条件。正如稍后将要看到的，在将平衡性替换为左倾性之后，左式堆结构的 `merge()` 操作乃至 `insert()` 和 `delMax()` 操作均可以高效地实现。

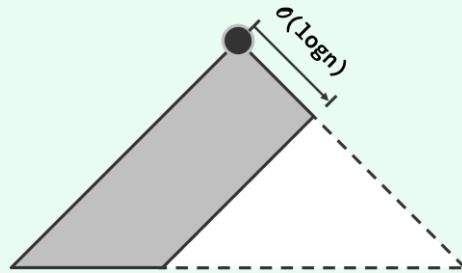


图10.13 整体结构向左倾斜，右侧通路上的节点  
不超过 $\mathcal{O}(\log n)$ 个

### 10.3.3 PQ\_LeftHeap模板类

按照以上思路，可以借助多重继承的机制，定义左式堆模板类如代码10.12所示。

```

1 #include "../PQ/PQ.h" //引入优先级队列ADT
2 #include "../BinTree/BinTree.h" //引入二叉树节点模板类
3
4 template <typename T>
5 class PQ_LeftHeap : public PQ<T>, public BinTree<T> { //基于二叉树，以左式堆形式实现的PQ
6 public:
7     PQ_LeftHeap() { } //默认构造
8     PQ_LeftHeap ( T* E, int n ) //批量构造：可改进为Floyd建堆算法
9     { for ( int i = 0; i < n; i++ ) insert ( E[i] ); }
10    void insert ( T ); //按照比较器确定的优先级次序插入元素
11    T getMax(); //取出优先级最高的元素
12    T delMax(); //删除优先级最高的元素
13 }; //PQ_LeftHeap

```

代码10.12 左式堆PQ\_LeftHeap模板类定义

可见，`PQ_LeftHeap`模板类借助多重继承机制，由`PQ`和`BinTree`结构共同派生而得。

这意味着，`PQ_LeftHeap`首先继承了优先级队列对外的标准ADT接口。另外，既然左式堆的逻辑结构已不再等价于完全二叉树，墨守成规地沿用此前基于向量的实现方法，必将难以控制空间复杂度。因此，改用紧凑性稍差、灵活性更强的二叉树结构，将更具针对性。

其中蛮力式批量构造方法耗时 $\mathcal{O}(n \log n)$ ，利用Floyd算法可改进至 $\mathcal{O}(n)$ （习题[10-13]）。

<sup>②</sup> 由C. A. Crane于1972年发明<sup>[58]</sup>，后由D. E. Knuth于1973年修订并正式命名<sup>[3]</sup>

### 10.3.4 空节点路径长度

左式堆的倾斜度，应该控制在什么范围？又该如何控制？为此，可借鉴AVL树和红黑树的技巧，为各节点引入所谓的“空节点路径长度”指标，并依此确定相关算法的执行方向。

节点 $x$ 的空节点路径长度（null path length），记作 $npl(x)$ 。若 $x$ 为外部节点，则约定 $npl(x) = npl(null) = 0$ 。反之若 $x$ 为内部节点，则 $npl(x)$ 可递归地定义为：

$$npl(x) = 1 + \min(npl(lc(x)), npl(rc(x)))$$

也就是说，节点 $x$ 的 $npl$ 值取决于其左、右孩子 $npl$ 值中的小者。

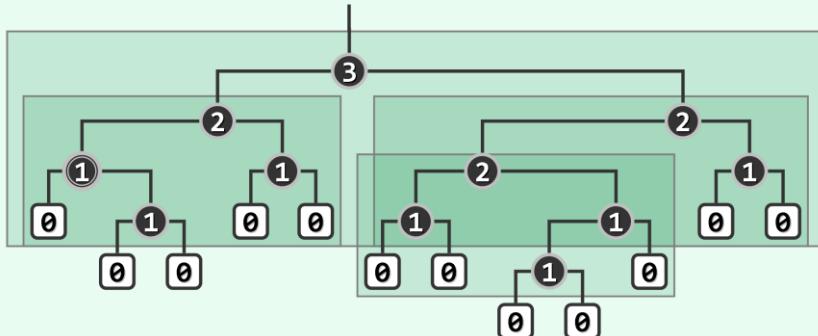


图10.14 空节点路径长度（其中有个节点违反左倾性，以双圈标出）

对照如图10.14所示的实例不难验证： $npl(x)$ 既等于 $x$ 到外部节点的最近距离（该指标由此得名），同时也等于以 $x$ 为根的最大满子树（图中以矩形框出）的高度。

### 10.3.5 左倾性与左式堆

左式堆是处处满足“左倾性”的二叉堆，即任一内部节点 $x$ 都满足

$$npl(lc(x)) \geq npl(rc(x))$$

也就是说，就 $npl$ 指标而言，任一内部节点的左孩子都不小于其右孩子。

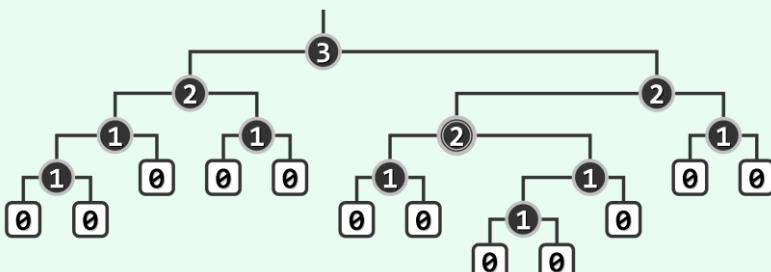


图10.15 左式堆：左孩子的 $npl$ 值不小于右孩子，而前者的高度却可能小于后者

照此标准不难验证，如图10.15所示的二叉堆即是左式堆，而图10.14中的二叉堆不是。

由 $npl$ 及左倾性的定义不难发现，左式堆中任一内节点 $x$ 都应满足：

$$npl(x) = 1 + npl(rc(x))$$

也就是说，左式堆中每个节点的 $npl$ 值，仅取决于其右孩子。

请注意，“左孩子的 $npl$ 值不小于右孩子”并不意味着“左孩子的高度必不小于右孩子”。

图10.15中的双圈节点即为一个反例，其左子堆和右子堆的高度分别为1和2。

### 10.3.6 最右侧通路

从 $x$ 出发沿右侧分支一直前行直至空节点，经过的通路称作其最右侧通路（rightmost path），记作 $rPath(x)$ 。在左式堆中，尽管右孩子高度可能大于左孩子，但由“各节点 $npl$ 值均决定于其右孩子”这一事实不难发现，每个节点的 $npl$ 值，应恰好等于其最右侧通路的长度。

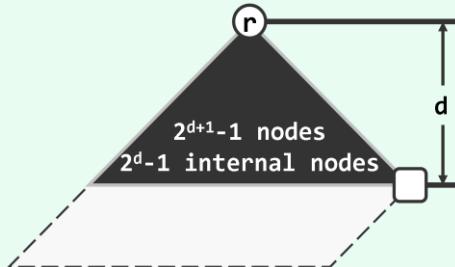


图10.16 左式堆的最右侧通路

根节点 $r$ 的最右侧通路，在此扮演的角色极其重要。如图10.16所示， $rPath(r)$ 的终点必为全堆中深度最小的外部节点。若记：

$$npl(r) = |rPath(r)| = d$$

则该堆应包含一棵以 $r$ 为根、高度为 $d$ 的满二叉树（黑色部分），且该满二叉树至少应包含 $2^{d+1} - 1$ 个节点、 $2^d - 1$ 个内部节点——这也是堆的规模下限。反之，在包含 $n$ 个节点的左式堆中，最右侧通路必然不会长于

$$\lfloor \log_2(n + 1) \rfloor - 1 = O(\log n)$$

### 10.3.7 合并算法

假设待合并的左式堆如图10.17(a)所示分别以 $a$ 和 $b$ 为堆顶，且不失一般性地 $a \geq b$ 。

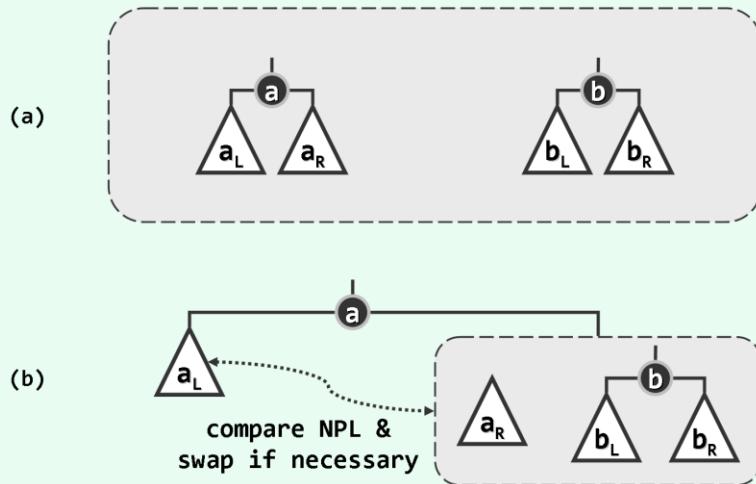


图10.17 左式堆合并算法原理

于是如图(b)，可递归地将 $a$ 的右子堆 $a_R$ 与堆 $b$ 合并，然后作为节点 $a$ 的右孩子替换原先的 $a_R$ 。当然，为保证依然满足左倾性条件，最后还需要比较 $a$ 左、右孩子的 $npl$ 值——如有必要还需将二者交换，以保证左孩子的 $npl$ 值不低于右孩子。

### 10.3.8 实例

考查如图10.18(a)所示的一对待合并左式堆。

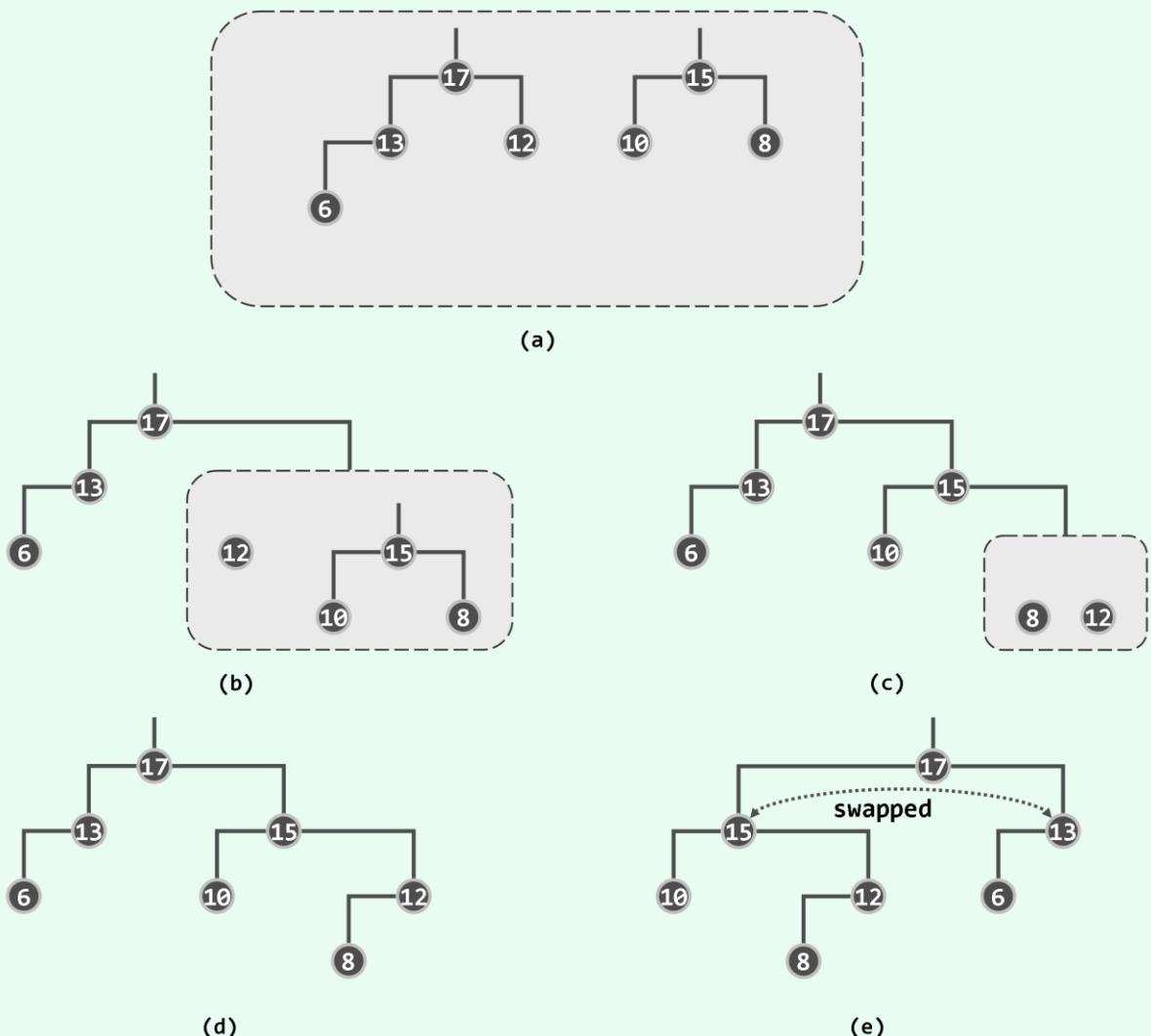


图10.18 左式堆合并算法实例

如图(b)所示，经过优先级比对可确定，应将堆17的右子堆12与堆15合并后，作为节点17新的右子树。为完成这一合并，如图(c)所示，经过优先级对比可确定，应将堆15的右子堆8与堆12合并后，作为节点15新的右子树。注意到此时节点12没有左孩子，故按照退化情况的处理规则，如图(d)所示，可将堆8直接作为节点12的左孩子。

至此，就结构性而言两个堆的合并任务已经完成。但为了保证左倾性依然满足，需要在逐级递归返回的过程，及时比较左右孩子的np1值，如有必要则将二者交换位置。仍继续上例，当如图(d)所示执行到最后一次递归返回时，可以发现根节点17的左、右孩子的np1值分别为1和2，故有必要将子堆13和子堆15交换位置，最终结果如图(e)所示。

### 10.3.9 合并操作的实现

按照以上思路，左式堆合并算法可具体描述和实现如代码10.13所示。

```

1 template <typename T> //根据相对优先级确定适宜的方式，合并以a和b为根节点的两个左式堆
2 static BinNodePosi(T) merge ( BinNodePosi(T) a, BinNodePosi(T) b ) {
3     if ( ! a ) return b; //退化情况
4     if ( ! b ) return a; //退化情况
5     if ( lt ( a->data, b->data ) ) swap ( a, b ); //一般情况：首先确保b不大
6     a->rc = merge ( a->rc, b ); //将a的右子堆，与b合并
7     a->rc->parent = a; //并更新父子关系
8     if ( !a->lcl || a->lcl->npl < a->rc->npl ) //若有必要
9         swap ( a->lcl, a->rc ); //交换a的左、右子堆，以确保右子堆的npl不大
10    a->npl = a->rc ? a->rc->npl + 1 : 1; //更新a的npl
11    return a; //返回合并后的堆顶
12 } //本算法只实现结构上的合并，堆的规模须由上层调用者负责更新

```

**代码10.13 左式堆合并接口merge()**

该算法首先判断并处理待合并子堆为空的边界情况。然后再通过一次比较，并在必要时做一次交换，以保证堆顶a的优先级总是不低于另一堆顶b。

以下按照前述原理，递归地将a的右子堆与堆b合并，并作为a的右子堆重新接入。接下来，还需比较此时a左、右孩子的npl值，如有必要还需做一次交换，以保证前者不小于后者。最后，只需在右孩子npl值的基础上加一，即可得到堆顶a的新npl值。至此，合并遂告完成。

当然，以上实现还足以处理多种退化的边界情况，限于篇幅不再赘述，请读者对照代码，就此独立分析和验证。

### 10.3.10 复杂度

借助递归跟踪图不难看出，在如代码10.13所示的合并算法中，所有递归实例可排成一个线性序列。因此，该算法实质上属于线性递归，其运行时间应线性正比于递归深度。

进一步地，由该算法原理及代码实现不难看出，递归只可能发生于两个待合并堆的最右侧通路上。根据10.3.6节的分析结论，若待合并堆的规模分别为n和m，则其两条最右侧通路的长度分别不会超过 $\mathcal{O}(\log n)$ 和 $\mathcal{O}(\log m)$ ，因此合并算法总体运行时间应不超过：

$$\mathcal{O}(\log n) + \mathcal{O}(\log m) = \mathcal{O}(\log(n + m)) = \mathcal{O}(\log(\max(n, m)))$$

可见，这一效率远远高于10.3.1节中的两个直觉算法。当然，与多数算法一样，若将以上递归版本改写为迭代版本（习题[10-15]），还可从常系数的意义上进一步提高效率。

### 10.3.11 基于合并的插入和删除

若将merge()操作当作一项更为基本的操作，则可以反过来实现优先级队列标准的插入和删除等操作。事实上，得益于merge()操作自身的高效率，如此实现的插入和删除操作，在时间效率方面毫不逊色于常规的实现方式。加之其突出的简洁性，使得这一实现方式在实际应用中受到更多的青睐。

### ■ delMax()

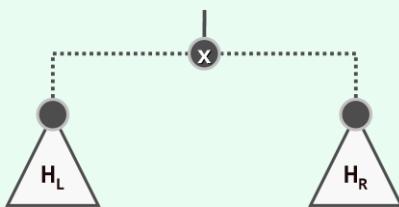


图10.19 基于堆合并操作实现删除接口

基于`merge()`操作实现`delMax()`算法，原理如图10.19所示。考查堆顶`x`及其子堆`HL`和`HR`。

在摘除`x`之后，`HL`和`HR`即可被视作为两个彼此独立的待合并的堆。于是，只要通过`merge()`操作将它们合并起来，则其效果完全等同于一次常规的`delMax()`删除操作。

照此思路，即可基于`merge()`操作实现`delMax()`接口如代码10.14所示。

```

1 template <typename T> T PQ_LeftHeap<T>::delMax() { //基于合并操作的词条删除算法 (当前队列非空)
2     BinNodePosi(T) lHeap = _root->lc; //左子堆
3     BinNodePosi(T) rHeap = _root->rc; //右子堆
4     T e = _root->data; delete _root; _size--; //删除根节点
5     _root = merge ( lHeap, rHeap ); //原左右子堆合并
6     if ( _root ) _root->parent = NULL; //若堆非空，还需相应设置父子链接
7     return e; //返回原根节点的数据项
8 }
```

代码10.14 左式堆节点删除接口`delMax()`

时间成本主要消耗于对`merge()`的调用，故由此前的分析结论，总体依然不超过 $\mathcal{O}(\log n)$ 。

### ■ insert()



图10.20 基于堆合并操作实现词条插入算法

基于`merge()`操作实现`insert()`接口的原理如图10.20所示。假设拟将词条`x`插入堆`H`中。

实际上，只要将`x`也视作（仅含单个节点的）堆，则通过调用`merge()`操作将该堆与堆`H`合并之后，其效果即完全等同于完成了一次词条插入操作。

照此思路，即可基于`merge()`操作实现`insert()`接口如代码10.15所示。

```

1 template <typename T> void PQ_LeftHeap<T>::insert ( T e ) { //基于合并操作的词条插入算法
2     BinNodePosi(T) v = new BinNode<T> ( e ); //为e创建一个二叉树节点
3     _root = merge ( _root, v ); //通过合并完成新节点的插入
4     _root->parent = NULL; //既然此时堆非空，还需相应设置父子链接
5     _size++; //更新规模
6 }
```

代码10.15 左式堆节点插入接口`insert()`

同样，时间成本主要也是消耗于对`merge()`的调用，总体依然不超过 $\mathcal{O}(\log n)$ 。



## 第11章

串

串或字符串（**string**）属于线性结构，自然地可直接利用向量或列表等序列结构加以实现。但字符串作为数据结构，特点也极其鲜明，这可归纳为：结构简单，规模庞大，元素重复率高。

所谓结构简单，是指字符表本身的规模不大，甚至可能极小。以生物信息序列为为例，参与蛋白质（文本）合成的常见氨基酸（字符）只有20种，而构成DNA序列（文本）的碱基（字符）仅有4种。尽管就规模而言，地球系统模式的单个输出文件长达1~100GB，微软Windows系统逾4000万行的源代码长度累计达到40GB，但它们都只不过是由ASCII字符，甚至是可打印字符组成的。因此，以字符串形式表示的海量文本数据的高效处理技术，一直都是相关领域的研究重点。

鉴于字符串结构的上述特点，本章将直接利用C++本身所提供的字符数组，并转而将讲述的重点，集中于各种串匹配算法**indexOf()**的基本原理与高效实现。

## § 11.1 串及串匹配

### 11.1.1 串

#### ■ 字符串

一般地，由n个字符构成的串记作：

$$S = "a_0 \ a_1 \ \dots \ a_{n-1}" , \text{ 其中, } a_i \in \Sigma, 0 \leq i < n$$

这里的 $\Sigma$ 是所有可用字符的集合，称作字符表（**alphabet**），例如二进制比特集 $\Sigma = \{ 0, 1 \}$ 、ASCII字符集、Unicode字符集、构成DNA序列的所有碱基、组成蛋白质的所有氨基酸，等等。

字符串S所含字符的总数n，称作S的长度，记作 $|S| = n$ 。这里只考虑长度有限的串， $n < \infty$ 。特别地，长度为零的串称作空串（**null string**）。请注意，空串并非由空格字符'□'组成的串，二者完全不同。

#### ■ 子串

字符串中任一连续的片段，称作其子串（**substring**）。具体地，对于任意的 $0 \leq i \leq i + k < n$ ，由字符串S中起始于位置i的连续k个字符组成的子串记作：

$$S.substr(i, k) = "a_i \ a_{i+1} \ \dots \ a_{i+k-1}" = S[i, i + k)$$

有两种特殊子串：起始于位置0、长度为k的子串称为前缀（**prefix**），而终止于位置n - 1、长度为k的子串称为后缀（**suffix**），分别记作：

$$prefix(S, k) = S.substr(0, k) = S[0, k)$$

$$suffix(S, k) = S.substr(n - k, k) = S[n - k, n)$$

由上述定义可直接导出以下结论：空串是任何字符串的子串，也是任何字符串的前缀和后缀；任何字符串都是自己的子串，也是自己的前缀和后缀。此类子串、前缀和后缀分别称作平凡子串（**trivial substring**）、平凡前缀（**trivial prefix**）和平凡后缀（**trivial suffix**）。反之，字符串本身之外的所有非空子串、前缀和后缀，分别称作真子串（**proper substring**）、真前缀（**proper prefix**）和真后缀（**proper suffix**）。

### ■ 判等

最后，字符串  $S[0, n]$  和  $T[0, m]$  称作相等，当且仅当二者长度相等 ( $n = m$ )，且对应的字符分别相同（对任何  $0 \leq i < n$  都有  $S[i] = T[i]$ ）。

### ■ ADT

串结构主要的操作接口可归纳为表11.1。

表11.1 串ADT支持的操作看接口

操作 接 口	功 能
<code>length()</code>	查询串的长度
<code>charAt(i)</code>	返回第 $i$ 个字符
<code>substr(i, k)</code>	返回从第 $i$ 个字符起、长度为 $k$ 的子串
<code>prefix(k)</code>	返回长度为 $k$ 的前缀
<code>suffix(k)</code>	返回长度为 $k$ 的后缀
<code>equal(T)</code>	判断 $T$ 是否与当前字符串相等
<code>concat(T)</code>	将 $T$ 串接在当前字符串之后
<code>indexOf(P)</code>	若 $P$ 是当前字符串的一个子串，则返回该子串的起始位置；否则返回 -1

比如，依次对串  $S = "data structures"$  执行如下操作，结果依次如表11.2所示。

表11.2 串操作实例

操 作	输 出	字 符 串 $s$
<code>length()</code>	15	"data structures"
<code>charAt(5)</code>	's'	"data structures"
<code>prefix(4)</code>	"data"	"data structures"
<code>suffix(10)</code>	"structures"	"data structures"
<code>concat("and algorithms")</code>		"data structures and algorithms"
<code>equal("data structures")</code>	false	"data structures and algorithms"
<code>equal("data structures and algorithms")</code>	true	"data structures and algorithms"
<code>indexOf("string")</code>	-1	"data structures and algorithms"
<code>indexOf("algorithm")</code>	20	"data structures and algorithms"

### 11.1.2 串匹配

#### ■ 应用与问题

在涉及字符串的众多实际应用中，模式匹配是最常使用的一项基本操作。比如UNIX Shell的grep工具（General Regular Expression Parser）和DOS的find命令，基本功能都是在指定的字符串中查找<sup>①</sup>特定模式的字符串。又如生物信息处理领域，也经常需要在蛋白质序列中

<sup>①</sup> 这两个命令都是以文件形式来指定待查找的文本串，具体格式分别是：

```
% grep <pattern> <file>
c:\> find "pattern" <file>
```

寻找特定的氨基酸模式，或在DNA序列中寻找特定的碱基模式。再如，邮件过滤器也需根据事先定义的特征串，通过扫描电子邮件的地址、标题及正文来识别垃圾邮件。还有，反病毒系统也会扫描刚下载的或将要执行的程序，并与事先提取的特征串相比对，以判定其中是否含有病毒。

上述所有应用问题，本质上都可转化为和描述为如下形式：

### 如何在字符串数据中，检测和提取以字符串形式给出的某一局部特征

这类操作都属于串模式匹配 (**string pattern matching**) 范畴，简称串匹配。一般地，即：

对基于同一字符表的任何文本串  $T$  ( $|T| = n$ ) 和模式串  $P$  ( $|P| = m$ )：

判定  $T$  中是否存在某一子串与  $P$  相同

若存在（匹配），则报告该子串在  $T$  中的起始位置

串的长度  $n$  和  $m$  本身通常都很大，但相对而言  $n$  更大，即满足  $2 \ll m \ll n$ 。比如，若：

```
T = "Now is the time for all good people to come"
```

```
P = "people"
```

则匹配的位置应该是  $T.indexOf(P) = 29$ 。

## ■ 问题分类

根据具体应用的要求不同，串匹配问题可以多种形式呈现。

有些场合属于模式检测 (**pattern detection**) 问题：我们只关心是否存在匹配而不关心具体的匹配位置，比如垃圾邮件的检测。有些场合属于模式定位 (**pattern location**) 问题：若经判断的确存在匹配，则还需确定具体的匹配位置，比如带毒程序的鉴别与修复。有些场合属于模式计数 (**pattern counting**) 问题：若有多处匹配，则统计出匹配子串的总数，比如网络热门词汇排行榜的更新。有些场合则属于模式枚举 (**pattern enumeration**) 问题：在有多处匹配时，报告出所有匹配的具体位置，比如网络搜索引擎。

### 11.1.3 测评标准与策略

串模式匹配是一个经典的问题，有名字的算法已不下三十种。鉴于串结构自身的特点，在设计和分析串模式匹配算法时也必须做特殊的考虑。其中首先需要回答的一个问题就是，如何对任一串匹配算法的性能作出客观的测量和评估。

多数读者首先会想到采用评估算法性能的常规口径和策略：以时间复杂度为例，假设文本串  $T$  和模式串  $P$  都是随机生成的，然后综合其各种组合从数学或统计等角度得出结论。很遗憾，此类构思并不适用于这一问题。

以基于字符表  $\Sigma = \{0, 1\}$  的二进制串为例。任给长度为  $n$  的文本串，其中长度为  $m$  的子串不过  $n - m + 1$  个 ( $m \ll n$  时接近于  $n$  个)。另一方面，长度为  $m$  的随机模式串多达  $2^m$  个，故匹配成功的概率为  $n / 2^m$ 。以  $n = 100,000$ 、 $m = 100$  为例，这一概率仅有

$$100,000 / 2^{100} < 10^{-25}$$

对于更长的模式串、更大的字符表，这一概率还将更低。因此，这一策略并不能有效地覆盖成功匹配的情况，所得评测结论也无法准确地反映算法的总体性能。

实际上，有效涵盖成功匹配情况的一种简便策略是，随机选取文本串  $T$ ，并从  $T$  中随机取出长度为  $m$  的子串作为模式串  $P$ 。这也是本章将采用的评价标准。

## § 11.2 蛮力算法

### 11.2.1 算法描述

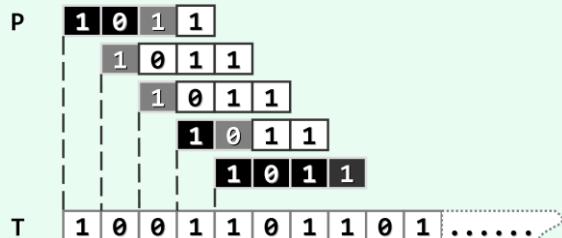


图11.1 串模式匹配的蛮力算法

蛮力串匹配是最直接最直觉的方法。如图11.1所示，可假想地将文本串和模式串分别写在两条印有等间距方格的纸带上，文本串对应的纸带固定，模式串纸带的首字符与文本串纸带的首字符对齐，二者都沿水平方向放置。于是，只需将P与T中长度为m的n - m + 1个子串逐一比对，即可确定可能的匹配位置。

不妨按自左向右的次序考查各子串。在初始状态下，T的前m个字符将与P的m个字符两两对齐。接下来，自左向右检查相互对齐的这m对字符：若当前字符对相互匹配，则转向下一对字符；反之一旦失配，则说明在此位置文本串与模式串不可能完全匹配，于是可将P对应的纸带右移一个字符，然后从其首字符开始与T中对应的新子串重新对比。图中，模式串P的每一黑色方格对应于字符对的一次匹配，每一灰色方格对应于一次失配，白色方格则对应于未进行的一次比对。若经过检查，当前的m对字符均匹配，则意味着整体匹配成功，从而返回匹配子串的位置。

蛮力算法的正确性显而易见：既然只有在某一轮的m次比对全部成功之后才成功返回，故不致于误报；反过来，所有对齐位置都会逐一尝试，故亦不致漏报。

### 11.2.2 算法实现

以下给出蛮力算法的两个实现版本。二者原理相同、过程相仿，但分别便于引入后续的不同改进算法，故在此先做一比较。

```

1 ****
2 * Text    : 0 1 2 . . . i-j . . . i . . n-1
3 *           -----|-----|-----|
4 * Pattern  :          0 . . . j . .
5 *           |-----|
6 ****
7 int match ( char* P, char* T ) { //串匹配算法 (Brute-force-1)
8     size_t n = strlen ( T ), i = 0; //文本串长度、当前接受比对字符的位置
9     size_t m = strlen ( P ), j = 0; //模式串长度、当前接受比对字符的位置
10    while ( j < m && i < n ) //自左向右逐个比对字符
11        if ( T[i] == P[j] ) //若匹配
12            { i++; j++; } //则转到下一对字符
13        else //否则
14            { i -= j - 1; j = 0; } //文本串回退、模式串复位
15    return i - j; //如何通过返回值，判断匹配结果？
16 }
```

如代码11.1所示的版本借助整数*i*和*j*，分别指示T和P中当前接受比对的字符T[i]与P[j]。若当前字符对匹配，则*i*和*j*同时递增以指向下一对字符。一旦*j*增长到*m*则意味着发现了匹配，即可返回P相对于T的对齐位置*i - j*。一旦当前字符对失配，则*i*回退并指向T中当前对齐位置的下一字符，同时*j*复位至P的首字符处，然后开始下一轮比对。

```

1 ****
2 * Text      : 0 1 2 . . . i i+1 . . . i+j . . n-1
3 *           -----|-----|-----|
4 * Pattern   :          0 1 . . . j . .
5 *           |-----|
6 ****
7 int match ( char* P, char* T ) { //串匹配算法 (Brute-force-2)
8     size_t n = strlen ( T ), i = 0; //文本串长度、与模式串首字符的对齐位置
9     size_t m = strlen ( P ), j; //模式串长度、当前接受比对字符的位置
10    for ( i = 0; i < n - m + 1; i++ ) { //文本串从第i个字符起，与
11        for ( j = 0; j < m; j++ ) //模式串中对应的字符逐个比对
12            if ( T[i + j] != P[j] ) break; //若失配，模式串整体右移一个字符，再做一轮比对
13        if ( j >= m ) break; //找到匹配子串
14    }
15    return i; //如何通过返回值，判断匹配结果？
16 }
```

#### 代码11.2 蛮力串匹配算法（版本二）

如代码11.2所示的版本，借助整数*i*指示P相对于T的对齐位置，且随着*i*不断递增，对齐的位置逐步右移。在每一对齐位置*i*处，另一整数*j*从0递增至*m - 1*，依次指示当前接受比对的字符为T[i + j]与P[j]。因此，一旦发现匹配，即可直接返回当前的对齐位置*i*。

### 11.2.3 时间复杂度

从理论上讲，蛮力算法至多迭代*n - m + 1*轮，且各轮至多需进行*m*次比对，故总共只需做不超过 $(n - m + 1) \cdot m$ 次比对。那么，这种最坏情况的确会发生吗？答案是肯定的。

考查如图11.2所示的实例。无论采用上述哪个版本的蛮力算法，都需做*n - m + 1*轮迭代，且各轮都需做*m*次比对。因此，整个算法共需做 $m \cdot (n - m - 1)$ 次字符比对，其中成功的和失败的各有 $(m - 1) \cdot (n - m - 1) + 1$ 和*n - m - 2*次。因*m << n*，渐进的时间复杂度应为 $\mathcal{O}(n \cdot m)$ 。



图11.2 蛮力算法的最坏情况

（也是基于坏字符策略BM算法的最好情况）

310

图11.3 蛮力算法的最好情况

（也是基于坏字符策略BM算法的最坏情况）

当然，蛮力算法的效率也并非总是如此低下。如图11.3所示，若将模式串P左右颠倒，则每经一次比对都可排除文本串中的一个字符，故此类情况下的运行时间为 $\mathcal{O}(n)$ 。实际上，此类最好（或接近最好）情况出现的概率并不很低，尤其是在字符表较大时（习题[11-9]）。

## § 11.3 KMP算法

### 11.3.1 构思

上一节的分析表明，蛮力算法在最坏情况下所需时间，为文本串长度与模式串长度的乘积，故无法应用于规模稍大的应用环境，很有必要改进。为此，不妨从分析以上最坏情况入手。

稍加观察不难发现，问题在于这里存在大量的局部匹配：每一轮的m次比对中，仅最后一次可能失配。而一旦发现失配，文本串、模式串的字符指针都要回退，并从头开始下一轮尝试。

实际上，这类重复的字符比对操作没有必要。既然这些字符在前一轮迭代中已经接受过比对并且成功，我们也就掌握了它们的所有信息。那么，如何利用这些信息，提高匹配算法的效率呢？

以下以蛮力算法的前一版本（代码11.1）为基础进行改进。

#### ■ 简单示例

如图11.4所示，用 $T[i]$ 和 $P[j]$ 分别表示当前正在接受比对的一对字符。

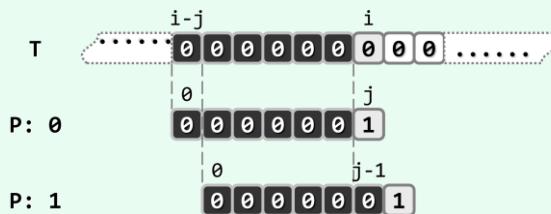


图11.4 利用以往的成功比对所提供的信息，可以避免文本串字符指针的回退

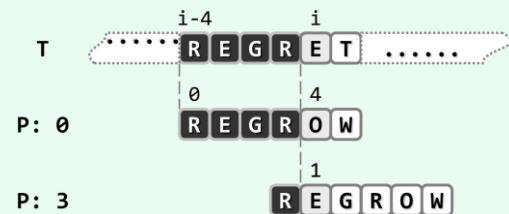


图11.5 利用以往的成功比对所提供的信息，有可能使模式串大跨度地右移

当本轮比对进行到最后一对字符并发现失配后，蛮力算法会令两个字符指针同步回退（即令 $i = i - j + 1$ 和 $j = 0$ ），然后再从这一位置继续比对。然而事实上，指针*i*完全不必回退。

#### ■ 记忆 = 经验 = 预知力

经过前一轮比对，我们已经清楚地知道，子串 $T[i - j, i)$ 完全由'0'组成。记住这一性质便可预测出：在回退之后紧接着的下一轮比对中，前 $j - 1$ 次比对必然都会成功。因此，可直接令*i*保持不变，令 $j = j - 1$ ，然后继续比对。如此，下一轮只需1次比对，共减少 $j - 1$ 次！

上述“令*i*保持不变、 $j = j - 1$ ”的含义，可理解为“令P相对于T右移一个单元，然后从前一失配位置继续比对”。实际上这一技巧可推而广之：利用以往的成功比对所提供的信息（记忆），不仅可避免文本串字符指针的回退，而且可使模式串尽可能大跨度地右移（经验）。

#### ■ 一般实例

如图11.5所示，再来考查一个更具一般性的实例。

本轮比对进行到发现 $T[i] = 'E' \neq 'O' = P[4]$ 失配后，在保持*i*不变的同时，应将模式串P右移几个单元呢？有必要逐个单元地右移吗？不难看出，在这一情况下移动一个或两个单元都是徒劳的。事实上，根据此前的比对结果，此时必然有

$$T[i - 4, i) = P[0, 4) = "REGR"$$

若在此局部能够实现匹配，则至少紧邻于 $T[i]$ 左侧的若干字符均应得到匹配——比如，当 $P[0]$ 与 $T[i - 1]$ 对齐时，即属这种情况。进一步地，若注意到 $i - 1$ 是能够如此匹配的最左侧位置，即可直接将P右移 $4 - 1 = 3$ 个单元（等效于*i*保持不变，同时令 $j = 1$ ），然后继续比对。

### 11.3.2 next表

一般地，如图11.6假设前一轮比对终止于 $T[i] \neq P[j]$ 。按以上构想，指针*i*不必回退，而是将 $T[i]$ 与 $P[t]$ 对齐并开始下一轮比对。那么， $t$ 准确地应该取作多少呢？

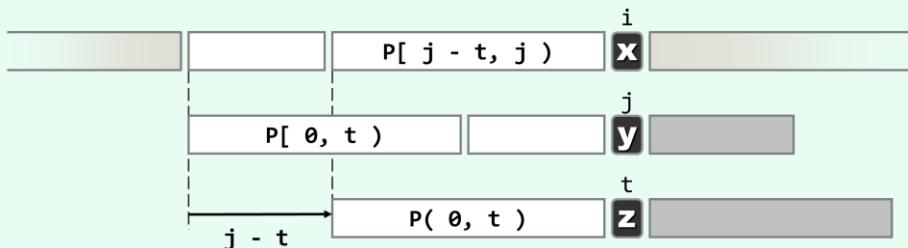


图11.6 利用此前成功比对所提供的信息，在安全的前提下尽可能大跨度地右移模式串

由图可见，经过此前一轮的比对，已经确定匹配的范围应为：

$$P[0, j) = T[i - j, i)$$

于是，若模式串 $P$ 经适当右移之后，能够与 $T$ 的某一（包含 $T[i]$ 在内的）子串完全匹配，则一项必要条件就是：

$$P[0, t) = T[i - t, i) = P[j - t, j)$$

亦即，在 $P[0, j)$ 中长度为 $t$ 的真前缀，应与长度为 $t$ 的真后缀完全匹配，故 $t$ 必来自集合：

$$N(P, j) = \{ \theta \leq t < j \mid P[\theta, t) = P[j - t, j) \}$$

一般地，该集合可能包含多个这样的 $t$ 。但需要特别注意的是，其中具体由哪些 $t$ 值构成，仅取决于模式串 $P$ 以及前一轮比对的首个失配位置 $P[j]$ ，而与文本串 $T$ 无关！

从图11.6还可看出，若下一轮比对将从 $T[i]$ 与 $P[t]$ 的比对开始，这等效于将 $P$ 右移 $j - t$ 个单元，位移量与 $t$ 成反比。因此，为保证 $P$ 与 $T$ 的对齐位置（指针*i*）绝不倒退，同时又不致遗漏任何可能的匹配，应在集合 $N(P, j)$ 中挑选最大的 $t$ 。也就是说，当有多个值得试探的右移方案时，应该保守地选择其中移动距离最短者。于是，若令

$$\text{next}[j] = \max(N(P, j))$$

则一旦发现 $P[j]$ 与 $T[i]$ 失配，即可转而将 $P[\text{next}[j]]$ 与 $T[i]$ 彼此对准，并从这一位置开始继续下一轮比对。

既然集合 $N(P, j)$ 仅取决于模式串 $P$ 以及失配位置 $j$ ，而与文本串无关，作为其中的最大元素， $\text{next}[j]$ 也必然具有这一性质。于是，对于任一模式串 $P$ ，不妨通过预处理提前计算出所有位置 $j$ 所对应的 $\text{next}[j]$ 值，并整理为表格以便此后反复查询——亦即，将“记忆力”转化为“预知力”。

### 11.3.3 KMP算法

312

上述思路可整理为代码11.3，即著名的KMP算法<sup>②</sup>。

这里，假定可通过`buildNext()`构造出模式串 $P$ 的`next`表。对照代码11.1的蛮力算法，只是在`else`分支对失配情况的处理手法有所不同，这也是KMP算法的精髓所在。

<sup>②</sup> Knuth和Pratt师徒，与Morris几乎同时发明了这一算法。他们稍后联合署名发表<sup>[60]</sup>该算法，并以其姓氏首字母命名

```

1 int match ( char* P, char* T ) { //KMP算法
2     int* next = buildNext ( P ); //构造next表
3     int n = ( int ) strlen ( T ), i = 0; //文本串指针
4     int m = ( int ) strlen ( P ), j = 0; //模式串指针
5     while ( j < m && i < n ) //自左向右逐个比对字符
6         if ( 0 > j || T[i] == P[j] ) //若匹配，或P已移出最左侧（两个判断的次序不可交换）
7             { i++; j++; } //则转到下一字符
8         else //否则
9             j = next[j]; //模式串右移（注意：文本串不用回退）
10    delete [] next; //释放next表
11    return i - j;
12 }

```

代码11.3 KMP主算法（待改进版）

#### 11.3.4 $\text{next}[0] = -1$

不难看出，只要  $j > 0$  则必有  $\theta \in N(P, j)$ 。此时  $N(P, j)$  非空，从而可以保证“在其中取最大值”这一操作的确可行。但反过来，若  $j = 0$ ，则即便集合  $N(P, j)$  可以定义，也必是空集。此种情况下，又该如何定义  $\text{next}[j = 0]$  呢？

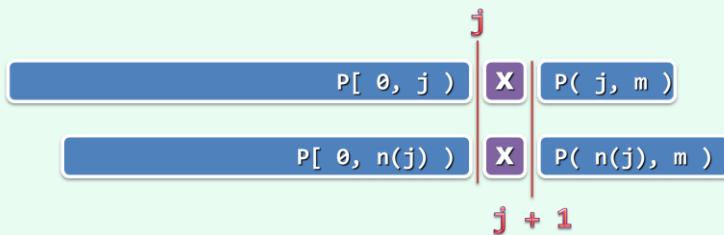
表11.3 next表实例：假想地附加一个通配符  $P[-1]$ 

rank	-1	0	1	2	3	4	5	6	7	8	9
$P[]$	*	C	H	I	N	C	H	I	L	L	A
$\text{next}[]$	N/A	-1	0	0	0	0	1	2	3	0	0

反观串匹配的过程。若在某一轮比对中首对字符即失配，则应将  $P$  直接右移一个字符，然后启动下一轮比对。因此如表11.3所示，不妨假想地在  $P[0]$  的左侧“附加”一个  $P[-1]$ ，且该字符与任何字符都是匹配的。就实际效果而言，这一处理方法完全等同于“令  $\text{next}[0] = -1$ ”。

#### 11.3.5 $\text{next}[j + 1]$

那么，若已知  $\text{next}[0, j]$ ，如何才能递推地计算出  $\text{next}[j + 1]$ ？是否有高效方法？

图11.7  $P[j] = P[\text{next}[j]]$  时，必有  $\text{next}[j + 1] = \text{next}[j] + 1$ 

若  $\text{next}[j] = t$ ，则意味着在  $P[0, j]$  中，自匹配的真前缀和真后缀的最大长度为  $t$ ，故必有  $\text{next}[j + 1] \leq \text{next}[j] + 1$ ——而且特别地，当且仅当  $P[j] = P[t]$  时如图11.7取等号。

那么一般地，若  $P[j] \neq P[t]$ ，又该如何得到  $\text{next}[j + 1]$ ？

此种情况下如图11.8，由next表的功能定义， $\text{next}[j + 1]$ 的下一候选者应该依次是  
 $\text{next}[\text{next}[j]] + 1, \text{next}[\text{next}[\text{next}[j]]] + 1, \dots$

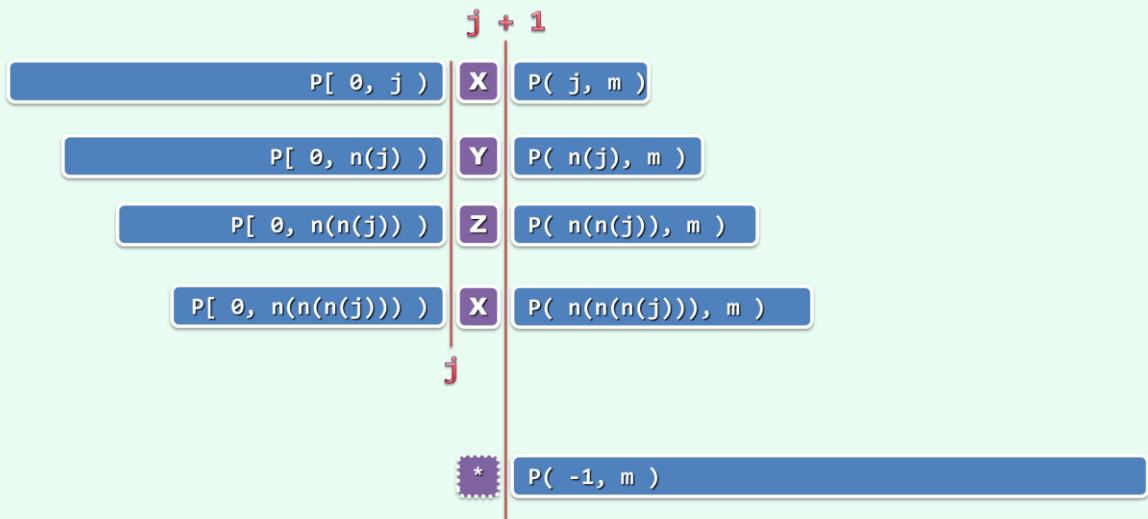


图11.8  $P[j] \neq P[\text{next}[j]]$ 时，必有 $\text{next}[j + 1] = \text{next}[\dots \text{next}[j] \dots] + 1$

因此，只需反复用 $\text{next}[t]$ 替换 $t$ （即令 $t = \text{next}[t]$ ），即可按优先次序遍历以上候选者；一旦发现 $P[j]$ 与 $P[t]$ 匹配（含与 $P[t = -1]$ 的通配），即可令 $\text{next}[j + 1] = \text{next}[t] + 1$ 。

既然总有 $\text{next}[t] < t$ ，故在此过程中 $t$ 必然严格递减；同时，即便 $t$ 降低至 $0$ ，亦必然会终止于通配的 $\text{next}[0] = -1$ ，而不致下溢。如此，该算法的正确性完全可以保证。

### 11.3.6 构造next表

按照以上思路，可实现next表构造算法如代码11.4所示。

```

1 int* buildNext ( char* P ) { //构造模式串P的next表
2     size_t m = strlen ( P ), j = 0; // "主" 串指针
3     int* N = new int[m]; //next表
4     int t = N[0] = -1; //模式串指针
5     while ( j < m - 1 )
6         if ( 0 > t || P[j] == P[t] ) { //匹配
7             j++; t++;
8             N[j] = t; //此句可改进...
9         } else //失配
10            t = N[t];
11     return N;
12 }
```

代码11.4 next表的构造

可见，next表的构造算法与KMP算法几乎完全一致。实际上按照以上分析，这一构造过程完全等效于模式串的自我匹配，因此两个算法在形式上的近似亦不足为怪。

### 11.3.7 性能分析

由上可见，KMP算法借助`next`表可避免大量不必要的字符比对操作，但这意味着渐进意义上的时间复杂度会有实质改进吗？这一点并非一目了然，甚至乍看起来并不乐观。比如就最坏情况而言，共有 $\Omega(n)$ 个对齐位置，而且在每一对齐位置都有可能需要比对多达 $\Omega(m)$ 次。

如此说来，难道在最坏情况下，KMP算法仍可能共需执行 $\Omega(nm)$ 次比对？不是的。以下更为精确的分析将证明，即便在最坏情况下，KMP算法也只需运行线性的时间！

为此，请留意代码11.3中用作字符指针的变量*i*和*j*。若令 $k = 2i - j$ 并考查*k*在KMP算法过程中的变化趋势，则不难发现：`while`循环每迭代一轮，*k*都会严格递增。

实际上，对应于`while`循环内部的`if-else`分支，无非两种情况：若转入`if`分支，则*i*和*j*同时加一，于是 $k = 2i - j$ 必将增加；反之若转入`else`分支，则尽管*i*保持不变，但在赋值 $j = \text{next}[j]$ 之后*j*必然减小，于是 $k = 2i - j$ 也必然会增加。

纵观算法的整个过程：启动时有 $i = j = 0$ ，即 $k = 0$ ；算法结束时 $i \leq n$ 且 $j \geq 0$ ，故有 $k \leq 2n$ 。在此期间尽管整数*k*从0开始持续地严格递增，但累计增幅不超过 $2n$ ，故`while`循环至多执行 $2n$ 轮。另外，`while`循环体内部不含任何循环或调用，故只需 $O(1)$ 时间。因此，若不计构造`next`表所需的时间，KMP算法本身的运行时间不超过 $O(n)$ 。也就是说，尽管可能有 $\Omega(n)$ 个对齐位置，但就分摊意义而言，在每一对齐位置仅需 $O(1)$ 次比对（习题[11-4]）。

既然`next`表构造算法的流程与KMP算法并无实质区别，故仿照上述分析可知，`next`表的构造仅需 $O(m)$ 时间。综上可知，KMP算法的总体运行时间为 $O(n + m)$ 。

### 11.3.8 继续改进

尽管以上KMP算法已可保证线性的运行时间，但在某些情况下仍有进一步改进的余地。

考查模式串 $P = "000010"$ 。按照11.3.2节的定义，其`next`表应如表11.4所示。

在KMP算法过程中，假设如图11.9前一轮比对因 $T[i] = '1' \neq '0' = P[3]$ 失配而中断。于是按照以上的`next`表，接下来KMP算法将依次将 $P[2]$ 、 $P[1]$ 和 $P[0]$ 与 $T[i]$ 对准并做比对。

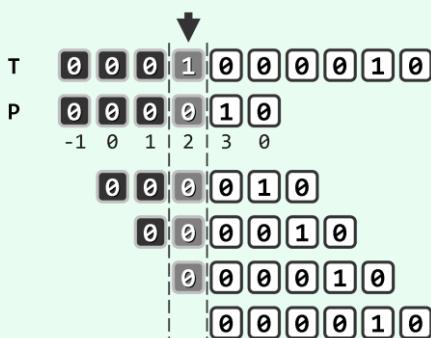


图11.9 按照此前定义的`next`表，仍有可能进行多次本不必要的字符比对操作

表11.4 `next`表仍有待优化的实例

rank	-1	0	1	2	3	4	5
$P[]$	*	0	0	0	0	1	0
$\text{next}[]$	N/A	-1	0	1	2	3	0

从图11.9可见，这三次比对都报告“失配”。那么，这三次比对的失败结果属于偶然吗？进一步地，这些比对能否避免？

实际上，即便说 $P[3]$ 与 $T[i]$ 的比对还算必要，后续的这三次比对却都是不必要的。实际上，它们的失败结果早已注定。

只需注意到 $P[3] = P[2] = P[1] = P[0] = '0'$ ，就不难看出这一点——既然经过此前的比对已发现 $T[i] \neq P[3]$ ，那么继续将 $T[i]$ 和那些与 $P[3]$ 相同的字符做比对，既重蹈覆辙，更徒劳无益。

## ■ 记忆 = 教训 = 预知力

就算法策略而言，11.3.2节引入next表的实质作用，在于帮助我们利用以往成功比对所提供的“经验”，将记忆力转化为预知力。然而实际上，此前已进行过的比对还远不止这些，确切地说还包括那些失败的比对——作为“教训”，它们同样有益，但可惜此前一直被忽略了。

依然以图11.9为例，以往所做的失败比对，实际上已经为我们提供了一条极为重要的信息—— $T[i] \neq P[4]$ ——可惜我们却未能有效地加以利用。原算法之所以会执行后续四次本不必要的比对，原因也正在于未能充分汲取教训。

## ■ 改进

为把这类“负面”信息引入next表，只需将11.3.2节中集合 $N(P, j)$ 的定义修改为：

```
N(P, j) = { 0 ≤ t < j | P[0, t] = P[j - t, j] 且 P[t] ≠ P[j] }
```

也就是说，除“对应于自匹配长度”以外， $t$ 只有还同时满足“当前字符对不匹配”的必要条件，方能归入集合 $N(P, j)$ 并作为next表项的候选。

相应地，原next表构造算法（代码11.4）也需稍作修改，调整为如下改进版本。

```
1 int* buildNext ( char* P ) { //构造模式串P的next表 ( 改进版本 )
2     size_t m = strlen ( P ), j = 0; // "主" 串指针
3     int* N = new int[m]; //next表
4     int t = N[0] = -1; //模式串指针
5     while ( j < m - 1 )
6         if ( 0 > t || P[j] == P[t] ) { //匹配
7             j++; t++;
8             N[j] = ( P[j] != P[t] ? t : N[t] ); //注意此句与未改进之前的区别
9         } else //失配
10            t = N[t];
11     return N;
12 }
```

代码11.5 改进的next表构造算法

由代码11.5可见，改进后的算法与原算法的唯一区别在于，每次在 $P[0, j]$ 中发现长度为 $t$ 的真前缀和真后缀相互匹配之后，还需进一步检查 $P[j]$ 是否等于 $P[t]$ 。唯有在 $P[j] \neq P[t]$ 时，才能将 $t$ 赋予 $next[j]$ ；否则，需转而代之以 $next[t]$ 。

仿照11.3.7节的分析方法易知，改进后next表的构造算法同样只需 $O(m)$ 时间。

## ■ 实例

仍以 $P = "000010"$ 为例，改进之后的next表如表11.5所示。读者可参照图11.9，就计算效率将新版本与原版本（表11.4）做一对比。

表11.5 改进后的next表实例

rank	-1	0	1	2	3	4	5
P[]	*	0	0	0	0	1	0
next[]	N/A	-1	-1	-1	-1	3	-1

利用新的next表针对图11.9中实例重新执行KMP算法，在首轮比对因 $T[i] = '1' \neq '0' = P[3]$ 失配而中断之后，将随即以 $P[next[3]] = P[-1]$ （虚拟通配符）与 $T[i]$ 对齐，并启动下一轮比对。将其效果而言，等同于聪明且安全地跳过了三个不必要的对齐位置。

## § 11.4 \*BM算法

### 11.4.1 思路与框架

#### ■ 构思

KMP算法的思路可概括为：当前比对一旦失配，即利用此前的比对（无论成功或失败）所提供的信息，尽可能长距离地移动模式串。其精妙之处在于，无需显式地反复保存或更新比对的历史，而是独立于具体的文本串，事先根据模式串预测出所有可能出现的失配情况，并将这些信息“浓缩”为一张next表。就其总体思路而言，本节将要介绍的BM算法<sup>③</sup>与KMP算法类似，二者的区别仅在于预测和利用“历史”信息的具体策略与方法。

BM算法中，模式串P与文本串T的对准位置依然“自左向右”推移，而在每一对准位置却是“自右向左”地逐一比对各字符。具体地，在每一轮自右向左的比对过程中，一旦发现失配，则将P右移一定距离并再次与T对准，然后新一轮自右向左的扫描比对。为实现高效率，BM算法同样需要充分利用以往的比对所提供的信息，使得P可以“安全地”向后移动尽可能远的距离。

#### ■ 主体框架

BM算法的主体框架，可实现如代码11.6所示。

```

1 int match ( char* P, char* T ) { //Boyer-Morre算法(完全版, 兼顾Bad Character与Good Suffix)
2     int* bc = buildBC ( P ); int* gs = buildGS ( P ); //构造BC表和GS表
3     size_t i = 0; //模式串相对于文本串的起始位置(初始时与文本串左对齐)
4     while ( strlen ( T ) >= i + strlen ( P ) ) { //不断右移(距离可能不止一个字符)模式串
5         int j = strlen ( P ) - 1; //从模式串最末尾的字符开始
6         while ( P[j] == T[i + j] ) //自右向左比对
7             if ( 0 > --j ) break;
8         if ( 0 > j ) //若极大匹配后缀 == 整个模式串(说明已经完全匹配)
9             break; //返回匹配位置
10        else //否则, 适当地移动模式串
11            i += __max ( gs[j], j - bc[ T[i + j] ] ); //位移量根据BC表和GS表选择大者
12    }
13    delete [] gs; delete [] bc; //销毁GS表和BC表
14    return i;
15 }
```

代码11.6 BM主算法

可见，这里采用了蛮力算法后一版本（310页代码11.2）的方式，借助整数*i*和*j*指示文本串中当前的对齐位置T[i]和模式串中接受比对的字符P[j]。不过，一旦局部失配，这里不再是机械地令*i* += 1并在下一字符处重新对齐，而是采用了两种启发式策略确定最大的安全移动距离。为此，需经过预处理，根据模式串P整理出坏字符和好后缀两类信息。

与KMP一样，算法过程中指针*i*始终单调递增；相应地，P相对于T的位置也绝不回退。

<sup>③</sup> 由R. S. Boyer和J. S. Moore于1977年发明<sup>[61]</sup>

### 11.4.2 坏字符策略

#### ■ 坏字符

如图11.10(a)和(b)所示, 若模式串P当前在文本串T中的对齐位置为i, 且在这一轮自右向左将P与substr(T, i, m)的比对过程中, 在P[j]处首次发现失配:

$$T[i + j] = 'X' \neq 'Y' = P[j]$$

则将'X'称作坏字符 (bad character)。问题是:

接下来应该选择P中哪个字符对准T[i + j], 然后开始下一轮自右向左的比对?

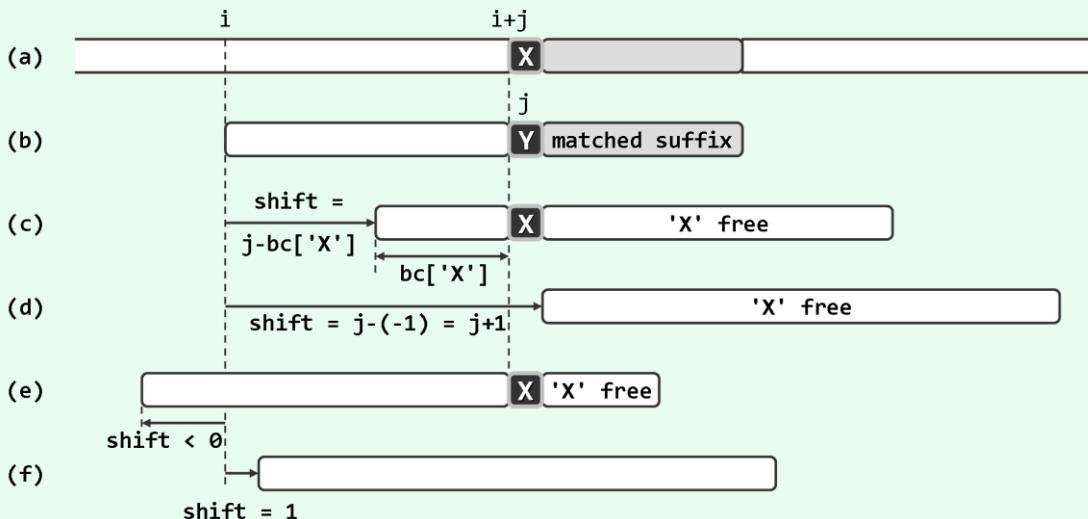


图11.10 坏字符策略：通过右移模式串P，使T[i + j]重新得到匹配

若P与T的某一（包括T[i + j]在内的）子串匹配，则必然在T[i + j] = 'X'处匹配；反之，若与T[i + j]对准的字符不是'X'，则必然失配。故如图11.10(c)所示，只需找出P中的每一字符 'X'，分别与T[i + j] = 'X' 对准，并执行一轮自右向左的扫描比对。不难看出，对应于每个这样的字符'X'，P的位移量仅取决于原失配位置j，以及'X'在P中的秩，而与T和i无关！

#### ■ bc[]表

若P中包含多个'X'，则是否真地有必要逐一尝试呢？实际上，这既不现实——如此将无法确保文本串指针i永不回退——更不必要。一种简便而高效的做法是，仅尝试P中最靠右的字符'X'（若存在）。与KMP算法类似，如此便可在确保不致遗漏匹配的前提下，始终单向地滑动模式串。具体如图11.10(c)所示，若P中最靠右的字符'X'为P[k] = 'X'，则P的右移量即为j - k。

同样幸运的是，对于任一给定的模式串P，k值只取决于字符T[i + j] = 'X'，因此可将其视作从字符表到整数（P中字符的秩）的一个函数：

$$bc(c) = \begin{cases} k & (\text{若 } P[k] = c, \text{ 且对所有的 } i > k \text{ 都有 } P[i] \neq c) \\ -1 & (\text{若 } P[] \text{ 中不含字符 } c) \end{cases}$$

故如代码11.6所示，如当前对齐位置为i，则一旦出现坏字符P[j] = 'Y'，即重新对齐于：

$$i += j - bc[T[i + j]]$$

并启动下一轮比对。为此可仿照KMP算法，预先将函数bc()整理为一份查询表，称作BC表。

## ■ 特殊情况

可用的BC表，还应足以处理各种特殊情况。比如，若P根本不含坏字符'X'，则如图11.10(d)所示，应将该串整体移过失配位置T[i + j]，用P[0]对准T[i + j + 1]，再启动下一轮比对。实际上，上述对bc()函数的定义已给出了应对方法——将BC表中此类字符的对应项置为-1。这种处理手法与KMP算法类似，其效果也等同于在模式串的最左端，增添一个通配符。

另外，即使P串中含有坏字符'X'，但其中最靠右者的位置也可能太靠右，以至于 $k = bc['X'] \geq j$ 。此时的 $j - k$ 不再是正数，故若仍以此距离右移模式串，则实际效果将如图11.10(e)所示等同于左移。显然，这类移动并不必要——匹配算法若果真能够进行至此，则此前左侧的所有位置都已被显式或隐式地否定排除了。因此，这种情况下不妨如图11.10(f)所示，简单地将P串右移一个字符，然后启动下一轮自右向左的比对。

## ■ bc[]表实例

以由大写英文字母和空格组成的字符表 $\Sigma = \{ \square, 'A' \sim 'Z' \}$ 为例。按照以上定义，与模式串"DATA STRUCTURES"相对应的BC表应如表11.6所示。

表11.6 模式串P = "DATA STRUCTURES"及其对应的BC表

rank	-1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
P[]	*	D	A	T	A	□	S	T	R	U	C	T	U	R	E	S

char	□	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
bc[]	4	3	-1	9	0	13	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	12	14	10	11	-1	-1	-1	-1	-1	-1

其中，字符'A'在秩为1和3处出现了两次， $bc['A']$ 取作其中的大者3；字符'T'则在秩为2、6和10处出现了三次， $bc['T']$ 取作其中的最大者10。在该字符串中并未出现的字符，对应的BC表项均统一取作-1，等效于指向在字符串最左端假想着增添的通配符。

## ■ bc[]表构造算法

按照上述思路，BC表的构造算法可实现如代码11.7所示。

```

1 //*****
2 // 0          bc['X']           m-1
3 //  |          |
4 //  .....X*****'X' free ----->|
5 //          .|-----'X' free ----->|
6 //*****
7 int* buildBC ( char* P ) { //构造Bad Character Shift表：O(m + 256)
8     int* bc = new int[256]; //BC表，与字符表等长
9     for ( size_t j = 0; j < 256; j ++ ) bc[j] = -1; //初始化：首先假设所有字符均未在P中出现
10    for ( size_t m = strlen ( P ), j = 0; j < m; j ++ ) //自左向右扫描模式串P
11        bc[ P[j] ] = j; //将字符P[j]的BC项更新为j（单调递增）——画家算法
12    return bc;
13 }
```

代码11.7 BC表的构造

该算法在对BC初始化之后，对模式串P做一遍线性扫描，并不断用当前字符的秩更新BC表中的对应项。因为是按秩递增的次序从左到右扫描，故只要字符c在P中出现过，则最终的bc[c]必将如我们的所期望的那样，记录下其中最靠右者的秩。

若将BC表比作一块画布，则其中各项的更新过程，就犹如画家在不同位置堆积不同的油彩。而画布上各处最终的颜色，仅取决于在对应位置所堆积的最后一笔——这类算法，也因此称作“画家算法”（painter's algorithm）。

代码11.7的运行时间可划分为两部分，分别消耗于其中的两个循环。前者是对字符表 $\Sigma$ 中的每个字符分别做初始化，时间量不超过 $\mathcal{O}(|\Sigma|)$ 。后一循环对模式串P做一轮扫描，其中每个字符消耗 $\mathcal{O}(1)$ 时间，故共需 $\mathcal{O}(m)$ 时间。由此可知，BC表可在 $\mathcal{O}(|\Sigma| + m)$ 时间内构造出来，其中 $|\Sigma|$ 为字符表的规模， $m$ 为模式串的长度。

### ■ 匹配实例

一次完整的查找过程，如图11.11所示。这里的文本串T长度为12(b)，模式串P长度为4(a)。模式串P中各字符所对应的bc[]表项，如图(a)所示。

因这里的字符表涵盖常用的汉字，规模很大，故为节省篇幅，除了模式串所含的四个字符，其余大量字符的bc[]表项均默认统一为-1，在此不再逐个标出。

以下，首先如图(c1)所示，在第一个对齐位置，经1次后比较发现 $P[3] = '常' \neq '非' = T[3]$ 。于是如图(c2)所示，将 $P[bc['非']] = P[2]$ 与 $T[3]$ 对齐，并经3次比较后发现 $P[1] = '名' \neq '道' = T[2]$ 。于是如图(c3)所示，将 $P[bc['道']] = P[-1]$ 与 $T[2]$ 对齐，并经1次比较发现 $P[3] = '常' \neq '名' = T[6]$ 。于是如图(c4)所示，将 $P[bc['名']] = P[1]$ 与 $T[6]$ 对齐，并经过1次比较发现 $P[3] = '常' \neq '名' = T[8]$ 。最后如图(c5)所示，将 $P[bc['名']] = P[1]$ 与 $T[8]$ 对齐，并经4次比较后匹配成功。

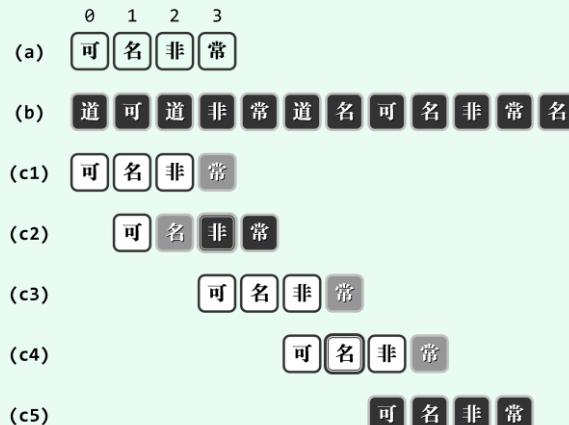


图11.11 借助bc[]表的串匹配

可见，整个过程中总共做过6次成功的（黑色字符）和4次失败的（灰色字符）比较，累计10次，文本串的每个有效字符平均为 $10/11$ 不足一次。

### ■ 复杂度

若暂且不计构造BC表的过程，BM算法本身进行串模式匹配所需的时间与具体的输入十分相关。若将文本串和模式串的长度分别记作n和m，则在通常情况下的实际运行时间往往低于 $\mathcal{O}(n)$ 。而在最好的情况下，每经过常数次比对，BM算法就可以将模式串右移m个字符（即整体右移）。比如，图11.2中蛮力算法的最坏例子，却属于BM算法的最好情况。此类情况下，只需经过 $n / m$ 次比对算法即可终止，故运行时间不超过 $\mathcal{O}(n / m)$ 。

反之，若如图11.3模式串P左右颠倒，则在每一轮比对中，P总要完整地扫描一遍才发现失配并向右移动一个字符。此类情况下的总体运行时间将为 $\mathcal{O}(n \times m)$ ，属于最坏情况。

### 11.4.3 好后缀策略

#### ■ 构思

上述基于坏字符的启发策略，充分体现了“将教训转化为预知力”的构思：一旦发现 $P[j]$ 与 $T[i + j]$ 失配，就将 $P$ 与 $T$ 重新对齐于至少可使 $T[i + j]$ 恢复匹配（含通配）的位置。然而正如上例所揭示的，这一策略有时仍显得不够“聪明”，计算效率将退化为几乎等同于蛮力算法。

参照KMP算法的改进思路不难发现，坏字符策略仅利用了此前（最后一次）失败比对所提供的“教训”。而实际上在此之前，还做过一系列成功的比对，而这些“经验”却被忽略了。

回到如图11.3所示的最坏情况，每当在 $P[0] = '1' \neq '0'$ 处失配，自然首先应该考虑将其替换为字符‘0’（或通配符）。但既然本轮比对过程中已有大量字符‘0’的成功匹配，则无论将 $P[0]$ 对准其中的任何一个都注定会失配。故此时更明智地，应将 $P$ 整体“滑过”这段区间，直接以 $P[0]$ 对准 $T$ 中尚未接受比对的首个字符。果真如此，算法的运行时间将有望降回至 $O(n)$ ！

#### ■ 好后缀

每轮比对中的若干次（连续的）成功匹配，都对应于模式串 $P$ 的一个后缀，称作“好后缀”（good suffix）。按照以上分析，必须充分利用好好后缀所提供的“经验”。

一般地，如图11.12(a)和(b)所示，设本轮自右向左的扫描终止于失配位置：

$$T[i + j] = 'X' \neq 'Y' = P[j]$$

若分别记

$$W = \text{substr}(T, i + j + 1, m - j - 1) = T[i + j + 1, m + i]$$

$$U = \text{suffix}(P, m - j - 1) = P[j + 1, m]$$

则 $U$ 即为当前的好后缀， $W$ 为 $T$ 中与之匹配的子串。

好后缀 $U$ 长度为 $m - j - 1$ ，故只要 $j \leq m - 2$ ，则 $U$ 必非空，且有 $U = W$ 。此时具体地：

根据好后缀所提供的信息应如何确定， $P$ 中有哪个（哪些）字符值得与上一失配字符 $T[i + j]$ 对齐，然后启动下一轮比对呢？

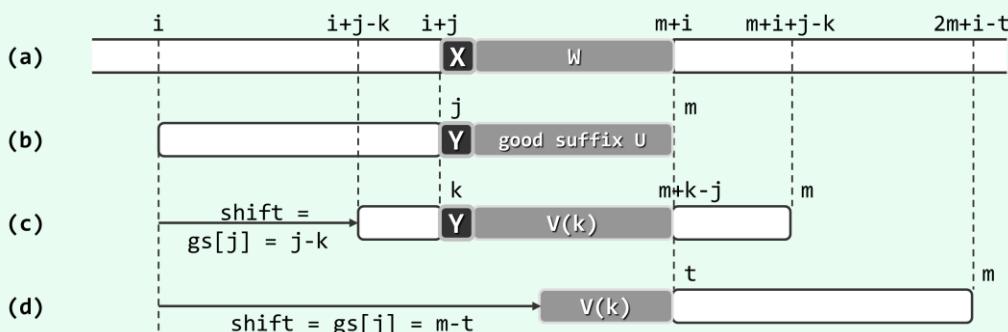


图11.12 好后缀策略：通过右移模式串 $P$ ，使与 $P$ 后缀 $U$ 匹配的 $W$ 重新得到匹配

如图11.12(c)所示，设存在某一整数 $k$ ，使得在将 $P$ 右移 $j - k$ 个单元，并使 $P[k]$ 与 $T[i + j]$ 相互对齐之后， $P$ 能够与文本串 $T$ 的某一（包含 $T[m + i - 1]$ 在内的）子串匹配，亦即：

$$P = \text{substr}(P, i + j - k, m) = T[i + j - k, m + i + j - k]$$

于是，若记：

$$V(k) = \text{substr}(P, k + 1, m - j - 1) = P[k + 1, m - j + k]$$

则必然有：

$$V(k) = W = U$$

也就是说，若值得将 $P[k]$ 与 $T[i + j]$ 对齐并做新一轮比对，则 $P$ 的子串 $V(k)$ 首先必须与 $P$ 自己的后缀 $U$ 相互匹配——这正是从好后缀中“挖掘”出来的“经验”。

此外，还有另一必要条件： $P$ 中这两个自匹配子串的前驱字符不得相等，即 $P[k] \neq P[j]$ 。否则，与第11.3.8节KMP算法的改进同理，在此对齐位置也注定不会出现与 $P$ 的整体匹配。

当然，若模式串 $P$ 中同时存在多个满足上述必要条件的子串 $V(k)$ ，则不妨选取其中最靠右者（对应于最大的 $k$ 、最小的右移距离 $j - k$ ）。这一处理手法的原理，依然与KMP算法类似——如此既不致遗漏匹配位置，亦可保证始终单向地“滑动”模式串，而不致回退。

### ■ gs[]表

如图11.12(c)所示，若满足上述必要条件的子串 $V(k)$ 起始于 $P[k + 1]$ ，则模式串对应的右移量应就是 $j - k$ 。表面上，此右移量同时取决于失配位置 $j$ 以及 $k$ ；然而实际上， $k$ 本身（也因此包括位移量 $j - k$ ）仅取决于模式串 $P$ 以及 $j$ 值。因此可以仿照KMP算法的做法，通过预处理，将模式串 $P$ 事先转换为另一张查找表 $gs[0, m]$ ，其中 $gs[j] = j - k$ 分别记录对应的位移量。

如图11.12(d)所示，若 $P$ 中没有任何子串 $V(k)$ 可与好后缀 $U$ 完全匹配呢？此时需从 $P$ 的所有前缀中，找出可与 $U$ 的某一（真）后缀相匹配的最长者，作为 $V(k)$ ，并取 $gs[j] = m - |V(k)|$ 。

表11.7 模式串 $P = "ICED RICE PRICE"$ 对应的GS表

$j$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$P[j]$	I	C	E	D	□	R	I	C	E	□	P	R	I	C	E
$gs[j]$	12	12	12	12	12	12	12	12	12	12	6	12	15	15	1

考查如表11.7所示的实例。其中的 $gs[10] = 6$ 可理解为：一旦在 $P[10] = 'P'$ 处发生失配，则应将模式串 $P$ 右移6个字符，即用 $P[10 - 6] = P[4] = '□'$ 对准文本串 $T$ 的失配字符，然后启动下一轮比对。类似地， $gs[5] = 12$ 意味着：一旦在 $P[5] = 'R'$ 处发生失配，则应将模式串 $P$ 整体右移12个字符，然后继续启动下一轮比对。当然，也可以等效地认为，以 $P[5 - 12] = P[-7]$ 对准文本串中失配的字符，或以 $P[0]$ 对准文本串中尚未对准过的最左侧字符。

### ■ 匹配实例

基于好后缀策略的匹配实例，如图11.13所示。首先如图(c1)所示，在第



一个对齐位置，经1次比较发现：



$P[7] = '也' \neq '静' = T[7]$   
于是如图(c2)所示，将 $P$ 右移 $gs[7] = 1$ 位，经3次比较发现：



$P[5] = '故' \neq '曰' = T[6]$  (c3)

于是如图(c3)所示，将 $P$ 右移 $gs[5] = 4$ 位，经8次比较后匹配成功。



图11.13 借助 $gs[]$ 表的串匹配：

(a) 模式串 $P$ 及其 $gs[]$ 表；(b) 文本串 $T$

可见，整个过程中总共做10次成功的（黑色字符）和2次失败的（灰色字符）比较，累计12次比较。文本串的每个字符，平均（12/13）不足一次。

## ■ 复杂度

如317页代码11.6所示，可以同时结合以上BC表和GS表两种启发策略，加快模式串相对于文本串的右移速度。可以证明，对于匹配失败的情况，总体比对的次数不致超过 $\mathcal{O}(n)$ <sup>[60][62][63]</sup>。

若不排除完全匹配的可能，则该算法在最坏情况下的效率，有可能退化至与蛮力算法相当。所幸，只要做些简单的改进，依然能够保证总体的比对次数不超过线性（习题[11-7]）。

综上所述，在兼顾了坏字符与好后缀两种策略之后，BM算法的运行时间为 $\mathcal{O}(n + m)$ 。

### 11.4.4 gs[]表构造算法

## ■ 蛮力算法

根据以上定义，不难直接导出一个构造gs[]表的“算法”：对于每个好后缀 $P(j, m)$ ，按照自后向前（ $k$ 从 $j - 1$ 递减至0）的次序，将其与 $P$ 的每个子串 $P(k, m + k - j)$ 逐一对齐，并核对是否出现如图11.12(c~d)所示的匹配。一旦发现匹配，对应的位移量即是 $gs[j]$ 的取值。

然而遗憾的是，这里共有 $\mathcal{O}(m)$ 个好后缀，各需与 $\mathcal{O}(m)$ 个子串对齐，每次对齐后在最坏情况下都需要比对 $\mathcal{O}(m)$ 次，因此该“算法”可能需要 $\mathcal{O}(m^3)$ 的时间。

实际上，仅需线性的时间即可构造出gs[]表（习题[11-6]）。为此，我们需要引入ss[]表。

## ■ MS[]串与ss[]表

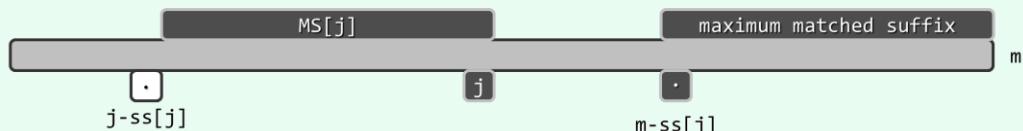


图11.14 MS[j]和ss[j]表的定义与含义

如图11.14所示，对于任一整数 $j \in [0, m]$ ，在 $P[0, j]$ 的所有后缀中，考查那些与 $P$ 的某一后缀匹配者。若将其中的最长者记作 $MS[j]$ ，则 $ss[j]$ 就是该串的长度 $|MS[j]|$ 。特别地，当 $MS[j]$ 不存在时，取 $ss[j] = 0$ 。

综上所述，可定义ss[j]如下：

$$ss[j] = \max\{ \theta \leq s \leq j + 1 \mid P(j - s, j) = P[m - s, m] \}$$

特别地，当 $j = m - 1$ 时，必有 $s = m$ ——此时，有 $P(-1, m - 1) = P[0, m]$ 。

## ■ 实例

表11.8 模式串 $P = "ICED RICE PRICE"$ 对应的SS表

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
P[i]	I	C	E	D	□	R	I	C	E	□	P	R	I	C	E
ss[i]	0	0	3	0	0	0	0	0	4	0	0	0	0	0	15

仍以表11.7中的模式串 $P$ 为例，按照如上定义， $P$ 所对应的ss[]表应如表11.8所示。

比如，其中之所以有 $ss[8] = 4$ ，是因为若取 $j = 8$ 和 $s = 4$ ，则有：

$$P(8 - 4, 8] = P(4, 8] = "RICE" = P[11, 15] = P[15 - 4, 15)$$

实际上，ss[]表中蕴含了gs[]表的所有信息，由前者足以便捷地构造出后者。

### ■ 由ss[]表构造gs[]表

如图11.15所示，任一字符 $P[j]$ 所对应的 $ss[j]$ 值，可分两种情况提供有效的信息。

第一种情况如图(a)所示，设该位置 $j$ 满足：

$$ss[j] = j + 1$$

也就是说， $MS[j]$ 就是整个前缀 $P[0, j]$ 。此时，对于 $P[m - j - 1]$ 左侧的每个字符 $P[i]$ 而言，对应于如图11.12(d)所示的情况， $m - j - 1$ 都应该是 $gs[i]$ 取值的一个候选。

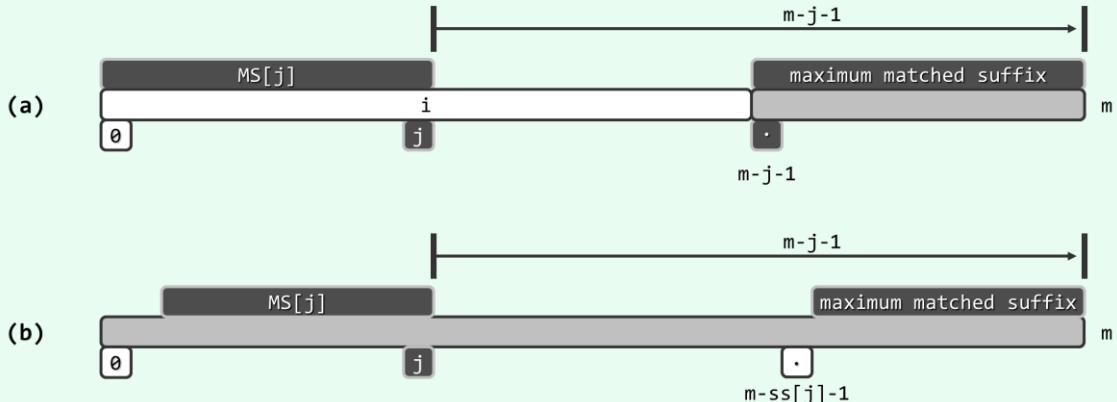


图11.15 由ss[]表构造gs[]表

第二种情况如图(b)所示，设该位置 $j$ 满足：

$$ss[j] \leq j$$

也就是说， $MS[j]$ 只是 $P[0, j]$ 的一个真后缀。此时，对于字符 $P[m - ss[j] - 1]$ 而言，对应于如图11.12(c)所示的情况，若同时还满足：

$$P[m - ss[j] - 1] \neq P[j - ss[j]]$$

则 $m - j - 1$ 也应是 $gs[m - ss[j] - 1]$ 取值的一个候选。

反过来，根据此前所做的定义，每一位置 $i$ 所对应的 $gs[i]$ 值只可能来自于以上候选。进一步地，既然 $gs[i]$ 的最终取值是上述候选中的最小（最安全）者，故仿照构造 $bc[]$ 表的画家算法，累计用时将不超过 $O(m)$ （习题[11-6]）。

### ■ ss[]表的构造

由上可见， $ss[]$ 表的确是构造 $gs[]$ 表的基础与关键。同样地，若采用蛮力策略，则对每个字符 $P[j]$ 都需要做一趟扫描对比，直到出现失配。如此，累计需要 $O(m^2)$ 时间。

为了提高效率，我们不妨自后向前地逆向扫描，并逐一计算出各字符 $P[j]$ 对应的 $ss[j]$ 值。如图11.16所示，因此时必有 $P[j] = P[m - hi + j - 1]$ ，故可利用此前已计算出的 $ss[m - hi + j - 1]$ ，分两种情况快速地导出 $ss[j]$ 。在此期间，只需动态地记录当前的极长匹配后缀：

$$P(lo, hi) = P[m - hi + lo, m]$$

第一种情况如图(a)所示，设：

$$ss[m - hi + j - 1] \leq j - lo$$

此时， $ss[m - hi + j - 1]$ 也是 $ss[j]$ 可能的最大取值，于是便可直接得到：

$$ss[j] = ss[m - hi + j - 1]$$

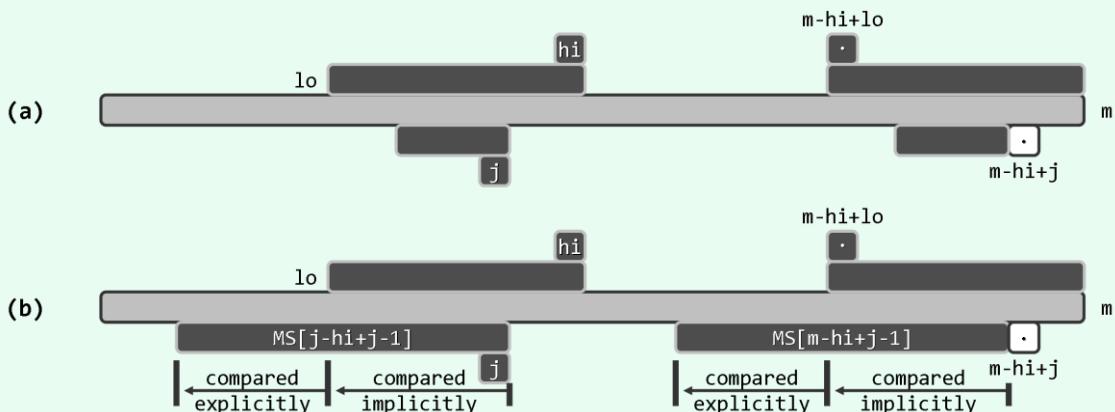


图11.16 构造ss[]表

第二种情况如图(b)所示，设：

$$j - lo < ss[m - hi + j - 1]$$

此时，至少仍有：

$$P(lo, j] = P[m - hi + lo, m - hi + j)$$

故只需将

$$P(j - ss[m - hi + j - 1], lo]$$

与

$$P[m - hi + j - ss[m - hi + j - 1], m - hi + lo)$$

做一比对，也可确定ss[j]。当然，这种情况下极大匹配串的边界lo和hi也需相应左移（递减）。

同样地，以上构思只要实现得当，也只需 $O(m)$ 时间即可构造出ss[]表（习题[11-6]）。

## ■ 算法实现

按照上述思路，GS表的构造算法可实现如代码11.8所示。

```

1 int* buildSS ( char* P ) { //构造最大匹配后缀长度表 : O(m)
2     int m = strlen ( P ); int* ss = new int[m]; //Suffix Size表
3     ss[m - 1] = m; //对最后一个字符而言，与之匹配的最长后缀就是整个P串
4 // 以下，从倒数第二个字符起自右向左扫描P，依次计算出ss[]其余各项
5     for ( int lo = m - 1, hi = m - 1, j = lo - 1; j >= 0; j -- )
6         if ( ( lo < j ) && ( ss[m - hi + j - 1] <= j - lo ) ) //情况一
7             ss[j] = ss[m - hi + j - 1]; //直接利用此前已计算出的ss[]
8         else { //情况二
9             hi = j; lo = __min ( lo, hi );
10            while ( ( 0 <= lo ) && ( P[lo] == P[m - hi + lo - 1] ) ) //二重循环？
11                lo--; //逐个对比处于(lo, hi]前端的字符
12            ss[j] = hi - lo;
13        }
14    return ss;
15 }
```

```

17 int* buildGS ( char* P ) { //构造好后缀位移量表 : O(m)
18     int* ss = buildSS ( P ); //Suffix Size table
19     size_t m = strlen ( P ); int* gs = new int[m]; //Good Suffix shift table
20     for ( size_t j = 0; j < m; j ++ ) gs[j] = m; //初始化
21     for ( size_t i = 0, j = m - 1; j < UINT_MAX; j -- ) //逆向逐一扫描各字符P[j]
22         if ( j + 1 == ss[j] ) //若P[0, j] = P[m - j - 1, m] , 则
23             while ( i < m - j - 1 ) //对于P[m - j - 1]左侧的每个字符P[i]而言 (二重循环? )
24                 gs[i++] = m - j - 1; //m - j - 1都是gs[i]的一种选择
25     for ( size_t j = 0; j < m - 1; j ++ ) //画家算法 : 正向扫描P[]各字符 , gs[j]不断递减 , 直至最小
26         gs[m - ss[j] - 1] = m - j - 1; //m - j - 1必是其gs[m - ss[j] - 1]值的一种选择
27     delete [] ss; return gs;
28 }

```

代码11.8 GS表的构造

### 11.4.5 算法纵览

#### ■ 时间效率的变化范围

以上我们针对串匹配问题，依次介绍了蛮力、KMP、基于BC表、综合BC表与GS表等四种典型算法，其渐进复杂度的跨度范围，可概括如图11.17所示。

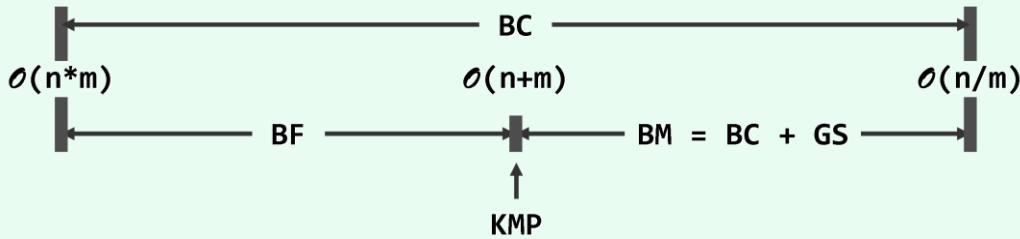


图11.17 典型串匹配算法的复杂度概览

其中，蛮力（BF）算法的时间效率介于 $O(n * m)$ 至 $O(n + m)$ 之间，而且其最好情况与KMP算法相当！当然，后者的优势在于，无论何种情况，时间效率均稳定在 $O(n + m)$ 。因此在蛮力算法效率接近或达到最坏的 $O(n * m)$ 时，KMP算法的优势才会十分明显。

仅采用坏字符启发策略（BC）的BM算法，时间效率介于 $O(n * m)$ 至 $O(n / m)$ 之间。可见，其最好情况与最坏情况相差悬殊。结合了好后缀启发策略（BC + GS）后的BM算法，则介于 $O(n + m)$ 和 $O(n / m)$ 之间。可见，在改进最低效率的同时，保持了最高效率的优势。

#### ■ 单次比对成功概率

饶有意味的是，单次比对成功的概率，是决定串匹配算法时间效率的一项关键因素。

纵观以上串匹配算法，在每一对齐位置所进行的一轮比对中，仅有最后一次可能失败；反之，此前的所有比对（若的确进行过）必然都是成功的。反观诸如图11.2、图11.3的实例可见，各种算法的最坏情况均可概括为：因启发策略不够精妙甚至不当，在每一对齐位置都需进行多达 $\Omega(m)$ 次成功的比对（另加最后一次失败的比对）。

若将单次比对成功的概率记作 $Pr$ ，则以上算法的时间性能随 $Pr$ 的变化趋势，大致如图11.18

所示。其中纵坐标为运行时间，分为 $O(n / m)$ 、 $O(n + m)$ 和 $O(n * m)$ 三档——当然，此处只是大致示意，实际的增长趋势未必是线性的。

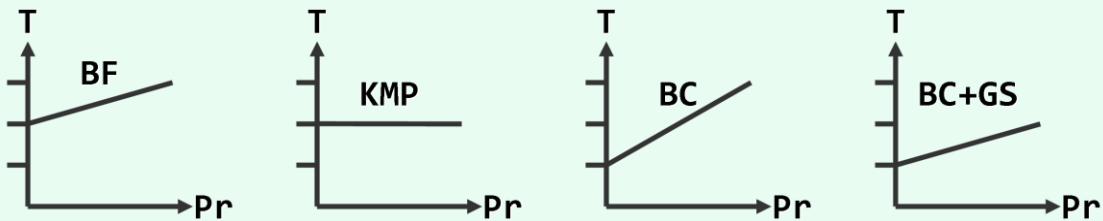


图11.18 随着单次比对成功概率（横轴）的提高，串匹配算法的运行时间（纵轴）通常亦将增加

可见，对于同一算法，计算时间与Pr具有单调正相关关系——这一点不难理解，正如以上分析，消耗于每一对齐位置的平均时间成本随Pr的提高而增加。

### ■ 字符表长度

实际上，在所有字符均等概率出现的情况下，Pr的取值将主要决定于字符表的长度 $|\Sigma|$ ，并与之成反比关系：字符表越长，其中任何一对字符匹配的概率越低。

这一性质可用以解释：在通常的情况下，蛮力算法实际的运行效率并不算太低（习题[11-9]）；不同的串匹配算法，因何各自有其适用的场合（习题[11-10]）。

## § 11.5 \*Karp-Rabin算法

### 11.5.1 构思

#### ■ 凡物皆数

早在公元前500年，先贤毕达哥拉斯及其信徒即笃信“凡物皆数”<sup>④</sup>。近世以来，以克罗内克<sup>⑤</sup>为代表的构造主义数学家曾坚定地认为，唯有可直接构造的自然数才是万物的本源。而此后无论是康托<sup>⑥</sup>还是哥德尔<sup>⑦</sup>，都以他们杰出的发现，为这一思想添加了生动的注脚。

其实，即便是限于本书所涉及和讨论的计算机科学领域，循着这一思路也可导出优美、简洁和高效的数据结构及算法。比如，细细品味第9章后不难领悟到，散列技术亦可视作为这一思想的产物。从这一角度来看，散列之所以可实现极高的效率，正在于它突破了通常对关键码的狭义理解——允许操作对象不必支持大小比较——从而在一般类型的对象（词条）与自然数（散列地址）之间，建立起直接的联系。

那么，这一构思与技巧，可否转而运用于本章讨论的主题呢？答案是肯定的。

#### ■ 串亦为数

为此，可以效仿康托的思路，将任一有限长度的整数向量视作自然数，进而在字符串与自然数之间建立联系。

<sup>④</sup> "All things are numbers.", Pythagoras (570 ~ 495 B.C.)

<sup>⑤</sup> "God made the integers; all else is the work of man.", L. Kronecker (1823 ~ 1891)

<sup>⑥</sup> Georg Cantor (1845 ~ 1918)

<sup>⑦</sup> Kurt Gödel (1906 ~ 1978)

若字母表规模  $|\Sigma| = d$ , 则任一字符串都将对应于一个  $d + 1$  进制<sup>⑧</sup>的整数。以由大写英文字母组成的字母表为例, 若将这些字符依次映射为 [1, 26] 内的自然数, 则每个这样的字符串都将对应于一个  $26 + 1 = 27$  进制的整数, 比如:

$$\text{"CANTOR"} = \langle 3, 1, 14, 20, 15, 18 \rangle_{(27)} = 43,868,727_{(10)}$$

$$\text{"DATA"} = \langle 4, 1, 20, 1 \rangle_{(27)} = 80002_{(10)}$$

从算法的角度来看, 这一映射关系就是一个不折不扣的散列。

### 11.5.2 算法与实现

#### ■ 算法

以上散列并非满射, 但不含 '0' 的任一  $d + 1$  进制值自然数, 均唯一地对应于某个字符串, 故它几乎已是一个完美的算法。字符串经如此转换所得的散列码, 称作其指纹 (**fingerprint**)。

按照这一理解, “判断模式串 P 是否与文本串 T 匹配”的问题, 可以转化为“判断 T 中是否有某个子串与模式串 P 拥有相同的指纹”的问题。具体地, 只要逐一取出 T 中长度为 m 的子串, 并将其对应的指纹与 P 所对应的指纹做一比对, 即可确定是否存在匹配位置——这已经可以称作一个串匹配算法了, 并以其发明者姓氏命名为 **Karp-Rabin** 算法。

该算法相关的预定义如代码 11.9 所示。这里仅考虑了阿拉伯数字串, 故每个串的指纹都已一个 R = 10 进制数。同时, 使用 64 位整数的散列码。

```
1 #define M 97 //散列表长度:既然这里并不需要真地存储散列表,不妨取更大的素数,以降低误判的可能
2 #define R 10 //基数:对于二进制串,取2;对于十进制串,取10;对于ASCII字符串,取128或256
3 #define DIGIT(S, i) ((S)[i] - '0') //取十进制串S的第i位数字值(假定S合法)
4 typedef __int64 HashCode; //用64位整数实现散列码
5 bool check1by1 ( char* P, char* T, size_t i );
6 HashCode prepareDm ( size_t m );
7 void updateHash ( HashCode& hashT, char* T, size_t m, size_t k, HashCode Dm );
```

#### 代码 11.9 Karp-Rabin 算法相关的预定义

算法的主体结构如代码 11.10 所示。除了预先计算模式串指纹 **hash(P)** 等预处理, 至多包含  $|T| - |P| = n - m$  轮迭代, 每轮都需计算当前子串的指纹, 并与目标指纹比对。

```
1 int match ( char* P, char* T ) { //串匹配算法(Karp-Rabin)
2     size_t m = strlen ( P ), n = strlen ( T ); //assert: m <= n
3     HashCode Dm = prepareDm ( m ), hashP = 0, hashT = 0;
4     for ( size_t i = 0; i < m; i++ ) { //初始化
5         hashP = ( hashP * R + DIGIT ( P, i ) ) % M; //计算模式串对应的散列值
6         hashT = ( hashT * R + DIGIT ( T, i ) ) % M; //计算文本串(前m位)的初始散列值
7     }
8     for ( size_t k = 0; ; ) { //查找
```

<sup>⑧</sup>之所以取  $d + 1$  而不是  $d$ , 是为了回避 '0' 字符以保证这一映射是单射

否则若字符串中存在由 '0' 字符组成的前缀, 则无论该前缀长度任何, 都不会影响对应的整数取值

```

9     if ( hashT == hashP )
10    if ( check1by1 ( P, T, k ) ) return k;
11    if ( ++k > n - m ) return k; //assert: k > n - m, 表示无匹配
12    else updateHash ( hashT, T, m, k, Dm ); //否则，更新子串散列码，继续查找
13 }
14 }
```

代码11.10 Karp-Rabin算法主体框架

请注意，这里并不需要真正地设置一个散列表，故空间复杂度与表长M无关。

### ■ 数位与字长

然而就效率而言，将上述方法称作算法仍嫌牵强。首先，直接计算各子串的指纹十分耗时。仍以上述大写英文字母表为例，稍长的字符串就可能对应于数值很大的指纹，比如：

"HASHING" = <8, 1, 19, 8, 9, 14, 7><sub>(27)</sub> = 3,123,974,608<sub>(10)</sub>  
 "KARPRABIN" = <11, 1, 18, 16, 18, 1, 2, 9, 14><sub>(27)</sub> = 3,124,397,993,144<sub>(10)</sub>

另一方面，随着字母表规模d的增大，指纹的位数也将急剧膨胀。以d = 128 = 2^7的ASCII字符集为例，只要模式串长度m = |P| ≥ 10，其指纹的长度就会达到m·log<sub>2</sub>d = 70个比特，从而超出目前通常支持的32 ~ 64位字长。这就意味着，若指纹持续加长，即便不考虑存储所需的空间字长而仅就时间成本而言，无论是指纹的计算还是指纹的比对，都无法在O(1)时间内完成。确切地说，这些操作所需的时间都将线性正比于模式串长度m。于是整个“算法”的时间复杂度将高达O(n \* m)——退回到11.2节的蛮力算法。

### ■ 散列压缩

不妨暂且搁置指纹的快速计算问题，首先讨论指纹的快速比对。既然上述指纹完全等效于字符串的散列码，上述问题也就与我们在9.3.2节中所面临的困境类似——若不能对整个散列空间进行有效的压缩，则以上方法将仅停留于朴素的构思，而将无法兑现为实用的算法。

仿照9.3.3节的思路和方法，这里不妨以除余法为例，通过散列函数hash(key) = key % M，将指纹的数值压缩至一个可以接受的范围。以十进制数字串为例，字母表规模d = 10。

例如，如图11.19所示即为散列表长选作M = 97时，一次完整的匹配过程。

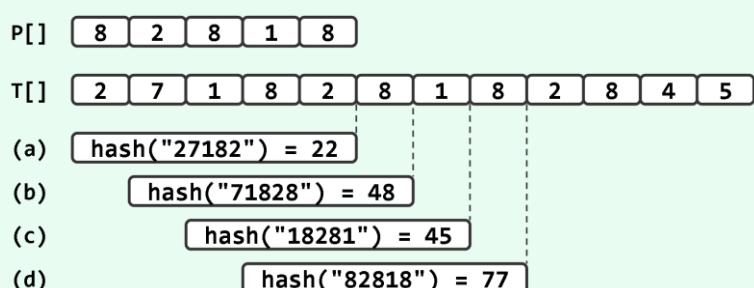


图11.19 Karp-Rabin串匹配算法实例：  
模式串指纹hash("82818") = 82,818 % 97 = 77

首先经预处理，提前计算出模式串P的指纹hash("82818") = 77。

此后，自左向右地依次取出文本串T中长度为m的各个子串，计算其指纹并与上述指纹对比。由图11.19可见，经过三次比对失败，最终确认匹配于substr(T, 3, 5) = P。

可见，经散列压缩之后，指纹比对所需的时间将仅取决于散列表长M，而与模式串长m无关。

## ■ 散列冲突

压缩散列空间的同时，必然引起冲突。就Karp-Rabin算法而言这体现为，文本串中不同子串的指纹可能相同，甚至恰好都与模式串的指纹相同。

仍考查以上实例，但如图11.20所示改换为 $P = "18284"$ ，其指纹 $\text{hash}("18284") = 48$ 。

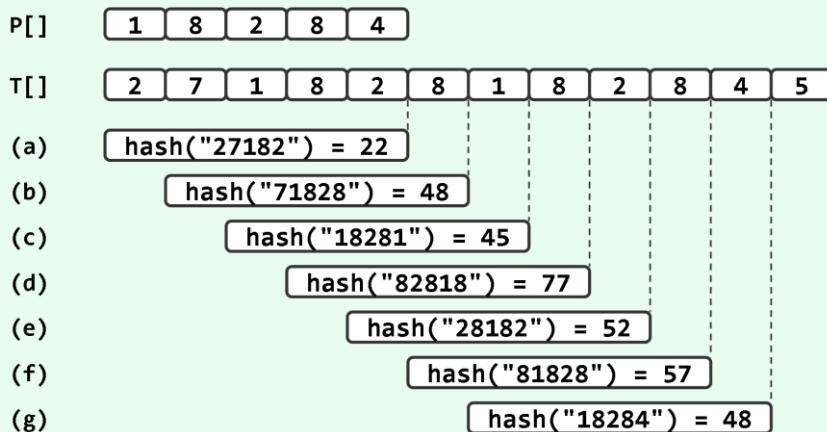


图11.20 Karp-Rabin串匹配算法实例：模式串指纹 $\text{hash}("18284") = 18,284 \% 97 = 48$

于是，尽管在第二次指纹比对时（图(b)）即发现 $\text{hash}("71828") = 48$ ，与模式串的指纹相同，但真正的匹配却应该在第七次比对后（图(g)）才能确认。

既然指纹相同并不是匹配的充分条件，故在发现指纹相等之后，还必须如代码11.11所示，对原字符串做一次严格的逐位比对。

```
1 bool check1by1 ( char* P, char* T, size_t i ) { //指纹相同时，逐位比对以确认是否真正匹配
2     for ( size_t m = strlen ( P ), j = 0; j < m; j++, i++ ) //尽管需要O(m)时间
3         if ( P[j] != T[i] ) return false; //但只要散列得当，调用本例程并返回false的概率将极低
4     return true;
5 }
```

代码11.11 指纹相同时还需逐个字符地比对

尽管这种比对需耗时 $O(m)$ ，但只要散列策略设计得当，即可有效地控制发生冲突以及执行此类严格比对的概率。以此处的除余法为例，若散列表容量选作M，则在“各字符皆独立且均匀分布”的假定条件下，指纹相同的可能性应为 $1/M$ ；而随着M的增大，冲突的概率将急速下降。代码11.9中选取 $M = 97$ 完全是出于演示的需要，实际应用中不妨适当地选用更长的散列表。

## ■ 快速指纹更新

最后，讨论快速指纹计算的实现。对图11.20等实例细加观察不难发现，按照自左向右的次序，任何两次相邻比对所对应的子串之间存在极强的相关性，子串的指纹亦是如此。

实际上，二者仅在首、末字符处有所出入。准确地如图11.21所示，前一子串删除首字符之后的后缀，与后一子串删除末字符之后的前缀完全相同。

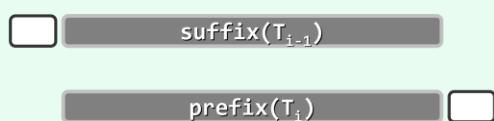


图11.21 相邻子串内容及指纹的相关性

利用这种相关性，可以根据前一子串的指纹，在常数时间内得到后一子串的指纹。也就是说，整个算法过程中消耗于子串指纹计算的时间，平均每次仅为 $O(1)$ 。

该算法的具体实现，如代码11.12所示。

```
1 // 子串指纹快速更新算法
2 void updateHash ( HashCode& hashT, char* T, size_t m, size_t k, HashCode Dm ) {
3     hashT = ( hashT - DIGIT ( T, k - 1 ) * Dm ) % M; //在前一指纹基础上，去除首位T[k - 1]
4     hashT = ( hashT * R + DIGIT ( T, k + m - 1 ) ) % M; //添加末位T[k + m - 1]
5     if ( 0 > hashT ) hashT += M; //确保散列码落在合法区间内
6 }
```

#### 代码11.12 串指纹的快速更新

这里，前一子串最高位对指纹的贡献量应为 $P[0] \times M^{m-1}$ 。只要注意到其中的 $M^{m-1}$ 始终不变，即可考虑如代码11.13所示，通过预处理提前计算出其对应的模余值。

为此尽管可采用代码1.8中的快速幂算法power2()，但考虑到此处仅需调用一次，同时兼顾算法的简洁性，故不妨直接以蛮力累乘的形式实现。

```
1 HashCode prepareDm ( size_t m ) { //预处理：计算R^(m - 1) % M（仅需调用一次，不必优化）
2     HashCode Dm = 1;
3     for ( size_t i = 1; i < m; i++ ) Dm = ( R * Dm ) % M; //直接累乘m - 1次，并取模
4     return Dm;
5 }
```

#### 代码11.13 提前计算 $M^{(m-1)}$



第12章

# 排序

此前各章已结合具体的数据结构，循序渐进地介绍过多种基本的排序算法：2.8节和3.5节分别针对向量和列表，统一以排序器的形式实现过起泡排序、归并排序、插入排序以及选择排序等算法；9.4.1节也曾按照散列的思路与手法，实现过桶排序算法；9.4.3节还将桶排序推广至基数排序算法；10.2.5节也曾完美地利用完全二叉堆的特长，实现过就地堆排序算法。

本章着重于高级排序算法。与以上基本算法一样，其构思与技巧各具特色，在不同应用中的效率也各有千秋。因此在学习过程中，唯有更多地关注不同算法之间细微而本质的差异，留意体会其优势与不足，方能做到运用自如，并结合实际问题的需要，合理取舍与并适当改造。

## § 12.1 快速排序

### 12.1.1 分治策略

与归并排序算法一样，快速排序（quicksort）算法<sup>①</sup>也是分治策略的典型应用，但二者之间也有本质区别。2.8.3节曾指出，归并排序的计算量主要消耗于有序子向量的归并操作，而子向量的划分却几乎不费时间。快速排序恰好相反，它可以在 $\mathcal{O}(1)$ 时间内，由子问题的解直接得到原问题的解；但为了将原问题划分为两个子问题，却需要 $\mathcal{O}(n)$ 时间。

快速排序算法虽然能够确保，划分出来的子任务彼此独立，并且其规模总和保持渐进不变，却不能保证两个子任务的规模大体相当——实际上，甚至有可能极不平衡。因此，该算法并不能保证最坏情况下的 $\mathcal{O}(n \log n)$ 时间复杂度。尽管如此，它仍然受到人们的青睐，并在实际应用中往往成为首选的排序算法。究其原因在于，快速排序算法易于实现，代码结构紧凑简练，而且对于按通常规律随机分布的输入序列，快速排序算法实际的平均运行时间较之同类算法更少。

下面结合向量介绍该算法的原理，并针对实际需求相应地给出不同的实现版本。

### 12.1.2 轴点

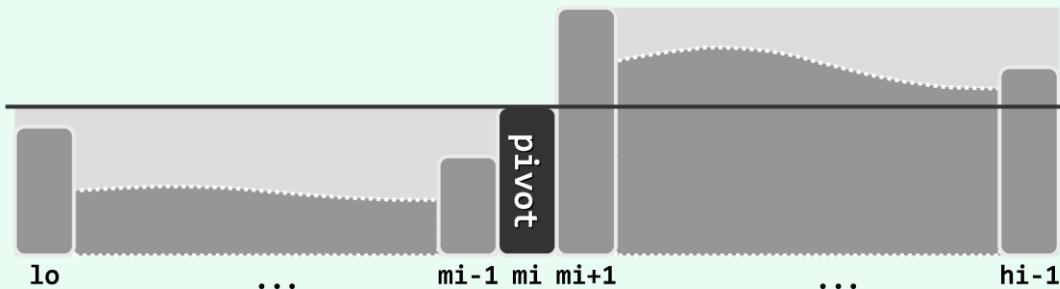


图12.1 序列的轴点（这里用高度来表示各元素的大小）

<sup>①</sup> 由英国计算机科学家、1980年图灵奖得主C. A. R. Hoare爵士于1960年发明<sup>[64]</sup>

如图12.1所示，考查任一向量区间 $S[lo, hi]$ 。对于任何 $lo \leq mi < hi$ ，以元素 $S[mi]$ 为界，都可分割出前、后两个子向量 $S[lo, mi]$ 和 $S(mi, hi)$ 。若 $S[lo, mi]$ 中的元素均不大于 $S[mi]$ ，且 $S(mi, hi)$ 中的元素均不小于 $S[mi]$ ，则元素 $S[mi]$ 称作向量 $S$ 的一个轴点（pivot）。

设向量 $S$ 经排序可转化为有序向量 $S'$ 。不难看出，轴点位置 $mi$ 必然满足如下充要条件：

- a)  $S[mi] = S'[mi]$
- b)  $S[lo, mi]$ 和 $S'[lo, mi]$ 的成员完全相同
- c)  $S(mi, hi)$ 和 $S'(mi, hi)$ 的成员完全相同

因此，不仅以轴点 $S[mi]$ 为界，前、后子向量的排序可各自独立地进行，而且更重要的是，一旦前、后子向量各自完成排序，即可立即（在 $O(1)$ 时间内）得到整个向量的排序结果。

采用分治策略，递归地利用轴点的以上特性，便可完成原向量的整体排序。

### 12.1.3 快速排序算法

按照以上思路，可作为向量的一种排序器，实现快速排序算法如代码12.1所示。

```

1 template <typename T> //向量快速排序
2 void Vector<T>::quickSort ( Rank lo, Rank hi ) { //0 <= lo < hi <= size
3     if ( hi - lo < 2 ) return; //单元素区间自然有序，否则...
4     Rank mi = partition ( lo, hi - 1 ); //在[lo, hi - 1]内构造轴点
5     quickSort ( lo, mi ); //对前缀递归排序
6     quickSort ( mi + 1, hi ); //对后缀递归排序
7 }
```

代码12.1 向量的快速排序

可见，轴点的位置一旦确定，则只需以轴点为界，分别递归地对前、后子向量实施快速排序；子向量的排序结果就地返回之后，原向量的整体排序即告完成。算法的核心与关键在于：

**轴点构造算法partition()应如何实现？可以达到多高的效率？**

### 12.1.4 快速划分算法

#### ■ 反例

事情远非如此简单，我们首先遇到的困难就是，并非每个向量都必然含有轴点。以如图12.2所示长度为9的向量为例，不难验证，其中任何元素都不是轴点。

1	2	3	4	5	6	7	8	0
0	1	2	3	4	5	6	7	8

图12.2 有序向量经循环左移一个单元后，将不含任何轴点

事实上根据此前的分析，任一元素作为轴点的必要条件之一是，其在初始向量 $S$ 与排序后有序向量 $S'$ 中的秩应当相同。因此反过来一般地，只要向量中所有元素都是错位的——即所谓的错排序列——则任何元素都不可能是轴点。

由上可见，若保持原向量的次序不变，则不能保证总是能够找到轴点。因此反过来，唯有通过适当地调整向量中各元素的位置，方可“人为地”构造出一个轴点。

## ■ 思路

为在区间 $[lo, hi]$ 内构造出一个轴点，首先需要任取某一元素 $m$ 作为“培养对象”。

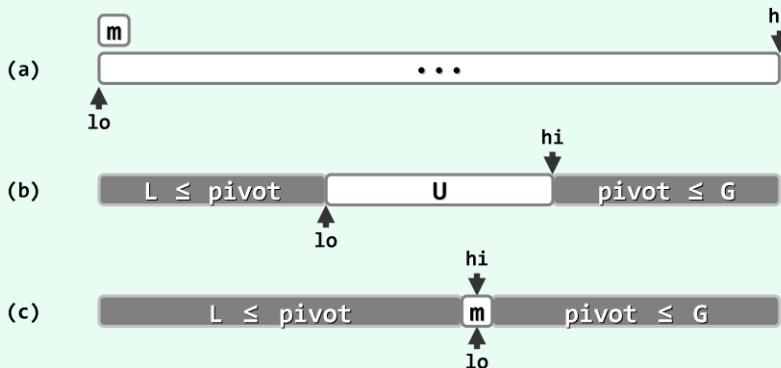


图12.3 轴点构造算法的构思

如图12.3(a)所示，不妨取首元素 $m = S[lo]$ 作为候选，将其从向量中取出并做备份，腾出的空闲单元便于其它元素的位置调整。然后如图(b)所示，不断试图移动 $lo$ 和 $hi$ ，使之相互靠拢。当然，整个移动过程中，需始终保证 $lo$  ( $hi$ ) 左侧 (右侧) 的元素均不大于 (不小于)  $m$ 。

最后如图(c)所示，当 $lo$ 与 $hi$ 彼此重合时，只需将原备份的 $m$ 回填至这一位置，则 $S[lo = hi] = m$ 便成为一个名副其实的轴点。

以上过程在构造出轴点的同时，也按照相对于轴点的大小关系，将原向量划分为左、右两个子向量，故亦称作快速划分（quick partitioning）算法。

## ■ 实现

按照以上思路，快速划分算法可实现如代码12.2所示。

```

1 template <typename T> //轴点构造算法：通过调整元素位置构造区间[lo, hi]的轴点，并返回其秩
2 Rank Vector<T>::partition ( Rank lo, Rank hi ) { //版本A：基本形式
3     swap ( _elem[lo], _elem[lo + rand() % ( hi - lo + 1 )] ); //任选一个元素与首元素交换
4     T pivot = _elem[lo]; //以首元素为候选轴点——经以上交换，等效于随机选取
5     while ( lo < hi ) { //从向量的两端交替地向中间扫描
6         while ( ( lo < hi ) && ( pivot <= _elem[hi] ) ) //在不小于pivot的前提下
7             hi--; //向左拓展右端子向量
8         _elem[lo] = _elem[hi]; //小于pivot者归入左侧子序列
9         while ( ( lo < hi ) && ( _elem[lo] <= pivot ) ) //在不大于pivot的前提下
10            lo++; //向右拓展左端子向量
11         _elem[hi] = _elem[lo]; //大于pivot者归入右侧子序列
12     } //assert: lo == hi
13     _elem[lo] = pivot; //将备份的轴点记录置于前、后子向量之间
14     return lo; //返回轴点的秩
15 }
```

代码12.2 轴点构造算法（版本A）

为便于和稍后的改进版本进行比较，不妨称作版本A。

过程

可见，算法的主体框架为循环迭代；主循环的内部，通过两轮迭代交替地移动 $lo$ 和 $hi$ 。

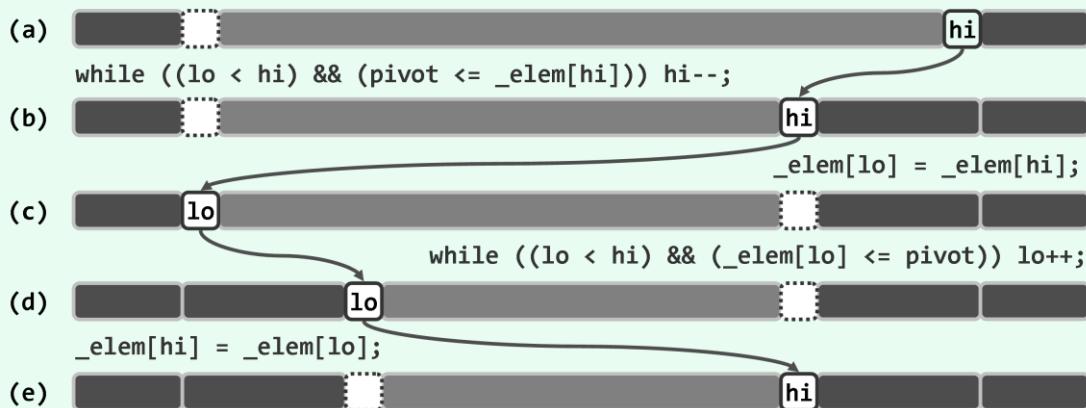


图12.4 轴点构造过程

各迭代的初始状态如图12.4(a)所示。反复地将候选轴点pivot与当前的`_elem[hi]`做比较，只要前者不大于后者，就不断向左移动`hi`（除非`hi`即将越过`lo`）。`hi`无法移动继续时，当如图(b)所示。于是接下来如图(c)所示，将`_elem[hi]`转移至`_elem[lo]`，并归入左侧子向量。

随后对称地，将`_elem[lo]`与`pivot`做比较，只要前者不大于后者，就不断向右移动`lo`（除非`lo`即将越过`hi`）。`lo`无法继续移动时，当如图(d)所示。于是接下来如图(e)所示，将`_elem[lo]`转移至`_elem[hi]`，并归入右侧子向量。

每经过这样的两轮移动， $\text{lo}$ 与 $\text{hi}$ 的间距都会缩短，故该算法迟早会终止。当然，若如图(e)所示 $\text{lo}$ 与 $\text{hi}$ 仍未重合，则可再做两轮移动。不难验证，在任一时刻，在以 $\text{lo}$ 和 $\text{hi}$ 为界的三个子向量中，左、右子向量分别满足12.1.2节所列的轴点充要条件b)和c)。而随着算法的持续推进，中间子向量的范围则不断压缩。当主循环退出时 $\text{lo}$ 和 $\text{hi}$ 重合，充要条件a)也随即满足。至此，只需将pivot“镶嵌”于左、右子向量之间，即实现了对原向量的一次轴点划分。

该算法的运行时间线性正比于被移动元素的数目，线性正比于原向量的规模 $\mathcal{O}(hi - lo)$ 。

■ 实例

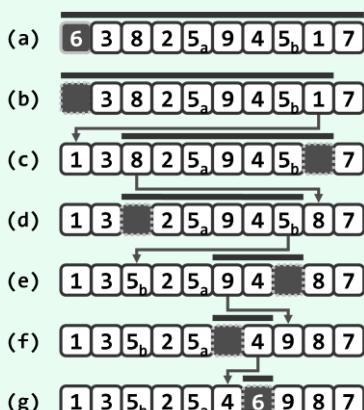


图12.5 轴点构造算法实例

快速划分算法的一次完整运行过程，如图12.5所示。输入序列A如图(a)长度为10，选择 $A[0] = 6$ 作为轴点候选。以下， $hi$ 和 $lo$ 的第一趟交替移动的过程及结果如图(b~c)所示，第二趟交替移动的过程及结果如图(d~e)所示，最后一趟交替移动的过程及结果如图(f~g)所示。

由于lo和hi的移动方向相反，故原处于向量右（左）端较小（大）的元素将按颠倒的次序转移至左（右）端；特别地，重复的元素也将按颠倒的次序转移至相对的一端，因而不再保持其原有的相对次序。由此可见，如此实现的快速排序算法并不稳定。从图12.5实例中数值为5的两个元素的移动过程与最终效果，不难看出这一点。

### 12.1.5 复杂度

#### ■ 最坏情况

上节的分析结论指出，采用代码12.2中的`partition()`算法，可在线性时间内将原向量的排序问题分解为两个相互独立、总体规模保持线性的子向量排序问题；而且根据轴点的性质，由各自排序后的子向量，可在常数时间内得到整个有序向量。也就是说，分治策略得以高效实现的两个必要条件——子问题划分的高效性及其相互之间的独立性——均可保证。然而尽管如此，另一项关键的必要条件——子任务规模接近——在这里却无法保证。事实上，由`partition()`算法划分出的子任务在规模上不仅不能保证接近，而且可能相差悬殊。

反观`partition()`算法不难发现，其划分所得子序列的长度与划分的具体过程无关，而是完全取决于入口处所选的候选轴点。具体地，若在最终有序向量中该候选元素的秩为 $r$ ，则子向量的规模必为 $r$ 和 $n - r - 1$ 。特别地， $r = 0$ 时子向量规模分别为 $0$ 和 $n - 1$ ——左侧子向量为空，而右侧子向量与原向量几乎等长。当然，对称的 $r = n - 1$ 亦属最坏情况。

更糟糕的是，这类最坏情况可能持续发生。比如，若每次都是简单地选择最左端元素`_elem[lo]`作为候选轴点，则对于完全（或几乎完全）有序的输入向量，每次（或几乎每次）划分的结果都是如此。这种情况下，若将快速排序算法处理规模为 $n$ 的向量所需的时间记作 $T(n)$ ，则如下递推关系始终成立：

$$T(n) = T(0) + T(n - 1) + O(n) = T(n - 1) + O(n)$$

综合考虑到其常数复杂度的递归基，与以上递推关系联立即可解得：

$$T(n) = T(n - 2) + 2 \cdot O(n) = \dots = T(0) + n \cdot O(n) = O(n^2)$$

也就是说，其效率居然低到与起泡排序相近。

#### ■ 降低最坏情况概率

那么，如何才能降低上述最坏情况出现的概率呢？读者可能已注意到，代码12.2的`partition()`算法在入口处增加了`swap()`一句，在区间内任选一个元素与`_elem[lo]`交换。就其效果而言，这使得后续的处理等同于随机选择一个候选轴点，从而在一定程度上降低上述最坏情况出现的概率。这种方法称作随机法。

类似地，也可采用所谓三者取中法：从待排序向量中任取三个元素，将数值居中者作为候选轴点。理论分析及实验统计均表明，较之固定选取某个元素或随机选取单个元素的策略，如此选出的轴点在最终有序向量中秩过小或过大的概率更低——尽管还不能彻底杜绝最坏情况。

#### ■ 平均运行时间

以上关于最坏情况下效率仅为 $O(n^2)$ 的结论不免令人沮丧，难道快速排序名不副实？实际上，更为细致的分析与实验统计都一致地显示，在大多数情况下，快速排序算法的平均效率依然可以达到 $O(n \log n)$ ；而且较之其它排序算法，其时间复杂度中的常系数更小。以下就以最常见的场景为例，对采用随机法确定候选轴点的快速排序算法的平均效率做一估算。

假设待排序的元素服从独立均匀随机分布。于是，`partition()`算法在经过 $n - 1$ 次比较和至多 $n + 1$ 次移动操作之后，对规模为 $n$ 的向量的划分结果无非 $n$ 种可能，划分所得左侧子序列的长度分别是 $0, 1, \dots, n - 1$ ，分别决定于所取候选元素在最终有序序列中的秩。按假定条件，每种情况的概率均为 $1/n$ ，故若将算法的平均运行时间记作 $\hat{T}(n)$ ，则有：

$$\begin{aligned}\hat{T}(n) &= (n + 1) + (1/n) \times \sum_{k=1}^n [\hat{T}(k - 1) + \hat{T}(n - k)] \\ &= (n + 1) + (2/n) \times \sum_{k=1}^n \hat{T}(k - 1)\end{aligned}$$

等式两侧同时乘以n，则有：

$$n \cdot \hat{T}(n) = (n + 1) \cdot n + 2 \cdot \sum_{k=1}^n \hat{T}(k - 1)$$

以及同理：

$$(n - 1) \cdot \hat{T}(n - 1) = (n - 1) \cdot n + 2 \cdot \sum_{k=1}^{n-1} \hat{T}(k - 1)$$

以上两式相减，即得：

$$\begin{aligned}n \cdot \hat{T}(n) - (n - 1) \cdot \hat{T}(n - 1) &= 2n + 2 \cdot \hat{T}(n - 1) \\ n \cdot \hat{T}(n) &= (n + 1) \cdot \hat{T}(n - 1) + 2n \\ \hat{T}(n)/(n + 1) &= \hat{T}(n - 1)/n + 2/(n + 1) \\ &= \hat{T}(n - 2)/(n - 1) + 2/(n + 1) + 2/n \\ &= \hat{T}(n - 3)/(n - 2) + 2/(n + 1) + 2/n + 2/(n - 1) \\ &= \dots \\ &= \hat{T}(0)/1 + 2/(n + 1) + 2/n + 2/(n - 1) + \dots + 2/2 \\ &= 2 \cdot \sum_{k=1}^{n+1} (1/k) - 1 \\ &\stackrel{(2)}{=} O(2 \cdot \ln n) = O(2 \cdot \ln 2 \cdot \log_2 n) = O(1.386 \cdot \log_2 n)\end{aligned}$$

正因为其良好的平均性能，加上其形象直观和易于实现的特点，快速排序算法自诞生起就一直受到人们的青睐，并被集成到Linux和STL等环境中。

### 12.1.6 应对退化

#### ■ 重复元素

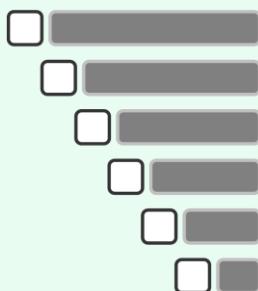


图12.6 `partition()` 算法的退化情况，也是最坏情况

考查所有（或几乎所有）元素均重复的退化情况。对照代码12.2不难发现，`partition()`算法的版本A对此类输入的处理完全等效于此前所举的最坏情况。事实上对于此类向量，主循环内部前一子循环的条件中“`pivot <= _elem[hi]`”形同虚设，故该子循环将持续执行，直至“`lo < hi`”不再满足。当然，在此之后另一内循环及主循环也将随即结束。

如图12.6所示，如此划分的结果必然是以最左端元素为轴点，原向量被分为极不对称的两个子向量。更糟糕的是，这一最坏情况还可能持续发生，从而使整个算法过程等效地退化为线性递归，递归深度为 $O(n)$ ，导致总体运行时间高达 $O(n^2)$ 。

<sup>(2)</sup> 若记 $h(n) = 1 + 1/2 + 1/3 + \dots + 1/n$ ，则有 $\ln(n+1) = \int_{i=1}^{n+1} (1/x) < h(n) < 1 + \int_{i=1}^n (1/x) = 1 + \ln n$

当然，可以在每次深入递归之前做统一核验，若确属退化情况，则无需继续递归而直接返回。但在重复元素不多时，如此不仅不能改进性能，反而会增加额外的计算量，总体权衡后得不偿失。

### ■ 改进

轴点构造算法可行的一种改进方案如代码12.3所示。为与如代码12.2所示同名算法版本A相区别，不妨称作版本B。

```

1 template <typename T> //轴点构造算法：通过调整元素位置构造区间[lo, hi]的轴点，并返回其秩
2 Rank Vector<T>::partition ( Rank lo, Rank hi ) { //版本B：可优化处理多个关键码相同的退化情况
3     swap ( _elem[lo], _elem[lo + rand() % ( hi - lo + 1 )] ); //任选一个元素与首元素交换
4     T pivot = _elem[lo]; //以首元素为候选轴点——经以上交换，等效于随机选取
5     while ( lo < hi ) { //从向量的两端交替地向中间扫描
6         while ( lo < hi )
7             if ( pivot < _elem[hi] ) //在大于pivot的前提下
8                 hi--; //向左拓展右端子向量
9             else //直至遇到不大于pivot者
10                { _elem[lo++] = _elem[hi]; break; } //将其归入左端子向量
11         while ( lo < hi )
12             if ( _elem[lo] < pivot ) //在小于pivot的前提下
13                 lo++; //向右拓展左端子向量
14             else //直至遇到不小于pivot者
15                { _elem[hi--] = _elem[lo]; break; } //将其归入右端子向量
16     } //assert: lo == hi
17     _elem[lo] = pivot; //将备份的轴点记录置于前、后子向量之间
18     return lo; //返回轴点的秩
19 }
```

**代码12.3 轴点构造算法（版本B）**

较之版本A，版本B主要是调整了两个内循环的终止条件。以前一内循环为例，原条件

```
pivot <= _elem[hi]
```

在此更改为：

```
pivot < _elem[hi]
```

也就是说，一旦遇到重复元素，右端子向量随即终止拓展，并将右端重复元素转移至左端。因此，若将版本A的策略归纳为“勤于拓展、懒于交换”，版本B的策略则是“懒于拓展、勤于交换”。

### ■ 效果及性能

对照代码12.3不难验证，对于由重复元素构成的输入向量，以上版本B将交替地将右（左）侧元素转移至左（右）侧，并最终恰好将轴点置于正中央的位置。这就意味着，退化的输入向量能够始终被均衡的切分，如此反而转为最好情况，排序所需时间为 $\mathcal{O}(n \log n)$ 。

当然，以上改进并非没有代价。比如，单趟partition()算法需做更多的元素交换操作。好在这并不影响该算法的线性复杂度。另外，版本B倾向于反复交换重复的元素，故它们在原输入向量中的相对次序更难保持，快速排序算法稳定性的不足更是雪上加霜。

## § 12.2 \*选取与中位数

### 12.2.1 概述

#### ■ k-选取

考查如下问题：

在任意一组可比较大小的元素中，如何找出由小到大次序为k者？

如图12.7(a)所示，也就是要从与这组元素对应的有序序列S中，找出秩为k的元素S[k]，故称作选取（selection）问题。若将目标元素的秩记作k，则亦称作k-选取（k-selection）问题。以无序向量A = { 3, 13, 2, 5, 8 }为例，对应的有序向量为S = { 2, 3, 5, 8, 13 }，其中的元素依次与k = { 0, 1, 2, 3, 4 }相对应。

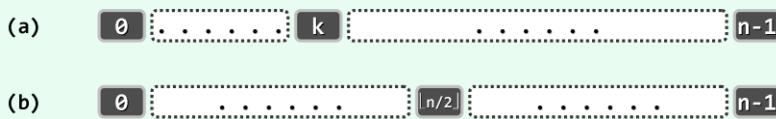


图12.7 选取与中位数

作为k-选取问题的特例，0-选取即通常的最小值问题，而(n - 1)-选取问题即通常的最大值问题。这两个问题都有平凡的最优解，例如List::selectMax()（82页代码3.21）。

在允许元素重复的场合，秩为k的元素可能同时存在多个副本。此时不妨约定，其中任何一个都可作为解答输出。

#### ■ 中位数

如图12.7(b)所示，在长度为n的有序序列S中，位序居中的元素S[ $\lfloor n/2 \rfloor$ ]称作中位数（median）。例如，有序序列S = { 2, 3, 5, 8, 13 }的中位数，为S[ $\lfloor 5/2 \rfloor$ ] = S[2] = 5；而有序序列S = { 2, 3, 5, 8, 13, 21 }的中位数，则为S[ $\lfloor 6/2 \rfloor$ ] = S[3] = 8。

即便对于尚未排序的序列，也可定义中位数——也就是在对原数据集排序之后，对应的有序序列的中位数。例如，无序序列A = { 3, 13, 2, 5, 8 }的中位数为元素A[3] = 5。

由于中位数可将原数据集（原问题）划分为大小明确、规模相仿且彼此独立的两个子集（子问题），故能否高效地确定中位数，将直接关系到采用分治策略的算法能否高效地实现。

#### ■ 蛮力算法

由中位数的定义，可直接得到查找中位数的如下直觉算法：对所有元素做排序，将其转换为有序序列S；于是，S[ $\lfloor n/2 \rfloor$ ]便是所要找的中位数。然而根据2.7.5节的结论，该算法在最坏情况下需要 $\Omega(n \log n)$ 时间。于是，基于该算法的任何分治算法，时间复杂度都会不低于：

$$T(n) = n \log n + 2 \cdot T(n/2) = O(n \log^2 n)$$

这一效率难以令人接受。

综上可见，中位数查找问题的挑战恰恰就在于：

如何在避免全排序的前提下，在 $O(n \log n)$ 时间内找出中位数？

不难看出，所谓中位数查找问题，也可以理解为是选取问题在 $k = \lfloor n/2 \rfloor$ 时的特例。稍后我们将看到，中位数查找问题既是选取问题的特例，同时也是选取问题中的难度最大者。

以下先结合若干特定情况讨论中位数的定位算法，然后再回到一般性的选取问题。

## 12.2.2 众数

### ■ 问题

为达到热身的目的，不妨先来讨论中位数问题的一个简化版本。在任一无序向量A中，若有半以上元素的数值同为m，则将m称作A的众数（**majority**）。例如，向量{ 5, 3, 9, 3, 3, 2, 3, 3 }的众数为3；而虽然3在向量{ 5, 3, 9, 3, 1, 2, 3, 3 }中最多，确非众数。

那么，任给无序向量，如何快速判断其中是否存在众数，并在存在时将其找出？尽管只是以整数向量为例，以下算法不难推广至元素类型支持判等和比较操作的任意向量。

### ■ 必要性与充分性

不难理解但容易忽略的一个事实是：若众数存在，则必然同时也是中位数。否则，在对应的有序向量中，总数超过半数的众数必然被中位数分隔为非空的两组——与向量的有序性相悖。

```
1 template <typename T> bool majority ( Vector<T> A, T& maj ) { //众数查找算法 : T可比较可判等
2     maj = majEleCandidate ( A ); //必要性 : 选出候选者maj
3     return majEleCheck ( A, maj ); //充分性 : 验证maj是否的确当选
4 }
```

代码12.4 众数查找算法主体框架

因此可如代码12.4所示，通过调用**majEleCandidate()**，从向量A中找到中位数maj（如果的确可以高效地查找到的话），并将其作为众数的唯一候选者。

然后再如代码12.5所示，调用**majEleCheck()**在线性时间内扫描一遍向量，通过统计该中位数出现的次数，即可验证其作为众数的充分性，从而最终判断向量A的众数是否的确存在。

```
1 template <typename T> bool majEleCheck ( Vector<T> A, T maj ) { //验证候选者是否确为众数
2     int occurrence = 0; //maj在A[]中出现的次数
3     for ( int i = 0; i < A.size(); i++ ) //逐一遍历A[]的各个元素
4         if ( A[i] == maj ) occurrence++; //每遇到一次maj，均更新计数器
5     return 2 * occurrence > A.size(); //根据最终的计数值，即可判断是否的确当选
6 }
```

代码12.5 候选众数核对算法

那么，在尚未得到高效的中位数查找算法之前，又该如何解决众数问题呢？

### ■ 减而治之

关于众数的另一重要事实，如图12.8所示：

设P为向量A中长度为2m的前缀。若元素x在P中恰好出现m次，则A有众数仅当后缀A-P拥有众数，且A-P的众数就是A的众数。

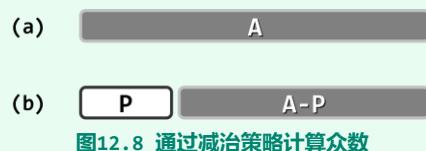


图12.8 通过减治策略计算众数

既然最终总会针对充分性另作一次核对，故不必担心A不含众数的情况，而只需验证A的确拥有众数的两种情况。若A的众数就是x，则在剪除前缀P之后，x与非众数均减少相同的数目，二者数目的差距在后缀A-P中保持不变。反过来，若A的众数为y ≠ x，则在剪除前缀P之后，y减少的数目也不致多于非众数减少的数目，二者数目的差距在后缀A-P中也不会缩小。

## ■ 实现

以上减而治之策略，可以实现为如代码12.6所示的majEleCandidate()算法。利用该算法，自左向右地扫描一遍整个向量，即可唯一确定满足如上必要条件的某个候选者。

```

1 template <typename T> T majEleCandidate ( Vector<T> A ) { //选出具备必要条件的众数候选者
2     T maj; //众数候选者
3     // 线性扫描：借助计数器c，记录maj与其它元素的数量差额
4     for ( int c = 0, i = 0; i < A.size(); i++ )
5         if ( 0 == c ) { //每当c归零，都意味着此时的前缀P可以剪除
6             maj = A[i]; c = 1; //众数候选者改为新的当前元素
7         } else //否则
8             maj == A[i] ? c++ : c--; //相应地更新差额计数器
9     return maj; //至此，原向量的众数若存在，则只能是maj —— 尽管反之不然
10 }
```

代码12.6 候选众数选取算法

其中，变量maj始终为当前前缀中出现次数不少于一半的某个元素；c则始终记录该元素与其它元素的数目之差。一旦c归零，则意味着如图12.8(b)所示，在当前向量中找到了一个可剪除的前缀P。在剪除该前缀之后，问题范围将相应地缩小至A-P。此后，只需将maj重新初始化为后缀A-P的首元素，并令c = 1，即可继续重复上述迭代过程。

对于向量的每个秩i，该算法迭代且仅迭代一步。故其运行时间，因线性正比于向量规模。

### 12.2.3 归并向量的中位数

## ■ 问题

本节继续讨论中位数问题的另一简化版本。考查如下问题：

任给有序向量S<sub>1</sub>和S<sub>2</sub>，如何找出它们归并后所得有序向量S = S<sub>1</sub> ∪ S<sub>2</sub>的中位数？

## ■ 蛮力算法

```

1 // 中位数算法蛮力版：效率低，仅适用于max(n1, n2)较小的情况
2 template <typename T> //子向量S1[lo1, lo1 + n1)和S2[lo2, lo2 + n2)分别有序，数据项可能重复
3 T trivialMedian ( Vector<T>& S1, int lo1, int n1, Vector<T>& S2, int lo2, int n2 ) {
4     int hi1 = lo1 + n1, hi2 = lo2 + n2;
5     Vector<T> S; //将两个有序子向量归并为一个有序向量
6     while ( ( lo1 < hi1 ) && ( lo2 < hi2 ) ) {
7         while ( ( lo1 < hi1 ) && S1[lo1] <= S2[lo2] ) S.insert ( S1[lo1 ++] );
8         while ( ( lo2 < hi2 ) && S2[lo2] <= S1[lo1] ) S.insert ( S2[lo2 ++] );
9     }
10    while ( lo1 < hi1 ) S.insert ( S1[lo1 ++] );
11    while ( lo2 < hi2 ) S.insert ( S2[lo2 ++] );
12    return S[ ( n1 + n2 ) / 2]; //直接返回归并向量的中位数
13 }
```

代码12.7 中位数蛮力查找算法

诚然，有序向量S中的元素 $S[\lfloor(n_1 + n_2)/2\rfloor]$ 即为中位数，但若果真按代码12.7中蛮力算法trivialMedian()将二者归并，则需花费 $\mathcal{O}(n_1 + n_2)$ 时间。这一效率虽不算太低，但毕竟未能充分利用“两个子向量已经有序”的条件。那么，能否更快地完成这一任务呢？

以下首先讨论 $S_1$ 和 $S_2$ 长度同为n的情况，稍后再推广至不等长的情况。

### ■ 减而治之

如图12.9所示，考查 $S_1$ 的中位数 $m_1 = S_1[\lfloor n/2 \rfloor]$ 和 $S_2$ 的逆向中位数 $m_2 = S_2[\lceil n/2 \rceil - 1] = S_2[\lfloor (n - 1)/2 \rfloor]$ ，并比较其大小。n为偶数和奇数的情况，分别如图(a)和图(b)所示。

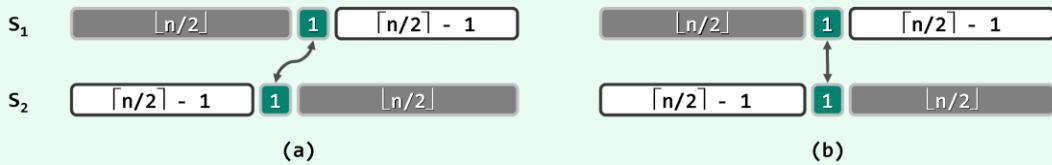


图12.9 采用减而治之策略，计算等长有序向量归并后的中位数

若 $m_1 = m_2$ ，则在 $S = S_1 \cup S_2$ 中，各有 $\lfloor n/2 \rfloor + (\lceil n/2 \rceil - 1) = n - 1$ 个元素不大于和不小于它们，故 $m_1$ 和 $m_2$ 就是S的中位数。若 $m_1 < m_2$ ，则意味着在S中各有 $\lfloor n/2 \rfloor$ 个元素（图中以灰色示意）不大于和不小于它们。可见，这些元素或者不是S的中位数，或者与 $m_1$ 或 $m_2$ 同为S的中位数。无论如何，在清除这些元素之后，S中位数的数值均保持不变。 $m_1 > m_2$ 的对称情况，与此类似。

综合以上分析，只需进行一次比较，即可将原问题的规模缩减大致一半。利用这一性质，如此反复递归，问题的规模将持续地以 $1/2$ 为比例，按几何级数的速度递减，直至平凡的递归基。

整个算法呈线性递归的形式，递归深度不超过 $\log_2 n$ ，每一递归实例仅需常数时间，故总体时间复杂度仅为 $\mathcal{O}(\log n)$ ——这一效率远远高于蛮力算法。

### ■ 实现

以上减而治之策略，可以实现为如代码12.8所示的median()算法。

```

1 template <typename T> //序列S1[lo1, lo1 + n)和S2[lo2, lo2 + n)分别有序, n > 0, 数据项可能重复
2 T median ( Vector<T>& S1, int lo1, Vector<T>& S2, int lo2, int n ) { //中位数算法(高效版)
3     if ( n < 3 ) return trivialMedian ( S1, lo1, n, S2, lo2, n ); //递归基
4     int mi1 = lo1 + n / 2, mi2 = lo2 + ( n - 1 ) / 2; //长度(接近)减半
5     if ( S1[mi1] < S2[mi2] )
6         return median ( S1, mi1, S2, lo2, n + lo1 - mi1 ); //取S1右半、S2左半
7     else if ( S1[mi1] > S2[mi2] )
8         return median ( S1, lo1, S2, mi2, n + lo2 - mi2 ); //取S1左半、S2右半
9     else
10        return S1[mi1];
11 }
```

代码12.8 等长有序向量归并后中位数算法

在向量长度小于3之后，即调用蛮力算法trivialMedian直接计算中位数。否则，分别取出 $m_1$ 和 $m_2$ ，并分三种情况继续线性递归。请体会“循秩访问”方式在此所起的关键性作用。

因属于尾递归，故不难将该算法改写为迭代形式（习题[12-6]）。

## ■ 一般情况

以上算法可如代码12.9所示推广至一般情况，即允许有序向量 $S_1$ 和 $S_2$ 的长度不等。

```

41     int mi1 = lo1 + n1 / 2;
42     int mi2a = lo2 + (n1 - 1) / 2;
43     int mi2b = lo2 + n2 - 1 - n1 / 2;
44     if (S1[mi1] > S2[mi2b]) //取S1左半、S2右半
45         return median (S1, lo1, n1 / 2 + 1, S2, mi2a, n2 - (n1 - 1) / 2);
46     else if (S1[mi1] < S2[mi2a]) //取S1右半、S2左半
47         return median (S1, mi1, (n1 + 1) / 2, S2, lo2, n2 - n1 / 2);
48     else //S1保留，S2左右同时缩短
49         return median (S1, lo1, n1, S2, mi2a, n2 - (n1 - 1) / 2 * 2);
50 }

```

#### 代码12.9 不等长有序向量归并后中位数算法

这一算法与代码12.8中同名算法的思路基本一致，请参照注释分析和验证其功能。

这里也采用了减而治之的策略，可使问题的规模大致按几何级数递减，故总体复杂度亦为 $\mathcal{O}(\log(n_1 + n_2))$ 。更精确地，其复杂度应为 $\mathcal{O}(\log(\min(n_1, n_2)))$ （习题[12-7]）——也就是说，子向量长度相等或接近时，此类问题的难度更大。

### 12.2.4 基于优先级队列的选取

#### ■ 信息量与计算成本

回到一般性的选取问题。蛮力算法的效率之所以无法令人满意，可以解释为：“一组元素中第 $k$ 大的元素”所包含的信息量，远远少于经过全排序后得到的整个有序序列。

花费足以全排序的计算成本，却仅得到了少量的局部信息，未免得不偿失。由此看来，既然只需获取原数据集的局部信息，为何不采用更适宜于这类计算需求的优先级队列结构呢？

#### ■ 堆

以堆结构为例。如图12.10所示，基于堆结构的选取算法大致有三种。

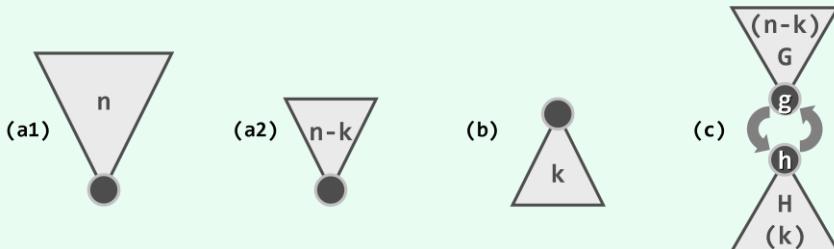


图12.10 基于堆结构的选取算法

第一种算法如图(a1)所示。首先，花费 $\mathcal{O}(n)$ 时间将全体元素组织为一个小顶堆；然后，经过 $k$ 次 $\text{delMin}()$ 操作，则如图(a2)所示得到位序为 $k$ 的元素。这一算法的运行时间为：

$$\mathcal{O}(n) + k \cdot \mathcal{O}(\log n) = \mathcal{O}(n + k \log n)$$

另一算法如图(b)所示。任取 $k$ 个元素，并在 $\mathcal{O}(k)$ 时间以内将其组织为大顶堆。然后将剩余的 $n - k$ 个元素逐个插入堆中；每插入一个，随即删除堆顶，以使堆的规模恢复为 $k$ 。待所有元素处理完毕之后，堆顶即为目标元素。该算法的运行时间为：

$$\mathcal{O}(k) + 2(n - k) \cdot \mathcal{O}(\log k) = \mathcal{O}(k + 2(n - k) \log k)$$

最后一种方法如图(c)。首先将全体元素分为两组，分别构建一个规模为 $n - k$ 的小顶堆G和一个规模为k的大顶堆H。接下来，反复比较它们的堆顶g和h，只要 $g < h$ ，则将二者交换，并重新调整两个堆。如此，G的堆顶g将持续增大，H的堆顶h将持续减小。当 $g \geq h$ 时，h即为所要找的元素。这一算法的运行时间为：

$$\mathcal{O}(n - k) + \mathcal{O}(k) + \min(k, n - k) \cdot 2 \cdot (\mathcal{O}(\log k + \log(n - k)))$$

在目标元素的秩很小或很大（即 $|n/2 - k| \approx n/2$ ）时，上述算法的性能都还不错。比如， $k \approx 0$ 时，前两种算法均只需 $\mathcal{O}(n)$ 时间。然而很遗憾，当 $k \approx n/2$ 时，以上算法的复杂度均退化至蛮力算法的 $\mathcal{O}(n \log n)$ 。因此，我们不得不转而从其它角度寻找突破口。

### 12.2.5 基于快速划分的选取

#### ■ 秩、轴点与快速划分

选取问题所查找元素的位序 $k$ ，就是其在对应的有序序列中的秩。就这一性质而言，该元素与轴点颇为相似。尽管12.1.4节的快速划分算法只能随机地构造一个轴点，但若反复应用这一算法，应该可以逐步逼近目标 $k$ 。

#### ■ 逐步逼近

以上构思可细化如下。首先，调用算法partition()构造向量A的一个轴点 $A[i] = x$ 。若 $i = k$ ，则该轴点恰好就是待选取的目标元素，即可直接将其返回。

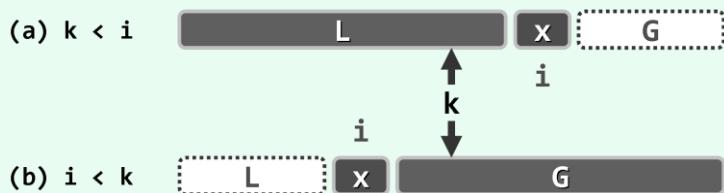


图12.11 基于快速划分算法逐步逼近选取目标元素

反之，若如图12.11所示 $i \neq k$ ，则无非两种情况。若如图(a)， $k < i$ ，则选取的目标元素不可能（仅）来自于处于x右侧、不小于x的子向量（白色）G中。此时，不妨将子向量G剪除，然后递归地在剩余区间继续做 $k$ -选取。反之若如图(b)， $i < k$ ，则选取的目标元素不可能（仅）来自于处于x左侧、不大于x的子向量（白色）L中。同理，此时也可将子向量L剪除，然后递归地在剩余区间继续做 $(k - i)$ -选取。

#### ■ 实现

基于以上减而治之、逐步逼近的思路，可实现quickSelect()算法如代码12.10所示。

```
1 template <typename T> void quickSelect ( Vector<T> & A, Rank k ) { //基于快速划分的k选取算法
2     for ( Rank lo = 0, hi = A.size() - 1; lo < hi; ) {
3         Rank i = lo, j = hi; T pivot = A[lo];
4         while ( i < j ) { //O(hi - lo + 1) = O(n)
5             while ( ( i < j ) && ( pivot <= A[j] ) ) j--; A[i] = A[j];
6             while ( ( i < j ) && ( A[i] <= pivot ) ) i++; A[j] = A[i];
7         } //assert: i == j
8         A[i] = pivot;
```

```

9     if ( k <= i ) hi = i - 1;
10    if ( i <= k ) lo = i + 1;
11 } //A[k] is now a pivot
12 }

```

#### 代码12.10 基于快速划分的k-选取算法

该算法的流程，与代码12.2中的partition()算法（版本A）如出一辙。每经过一步主迭代，都会构造出一个轴点A[i]，然后lo和hi将彼此靠拢，查找范围将收缩至A[i]的某一侧。当轴点的秩i恰为k时，算法随即终止。如此，A[k]即是待查找的目标元素。

尽管内循环仅需 $\mathcal{O}(hi - lo + 1)$ 时间，但很遗憾，外循环的次数却无法有效控制。与快速排序算法一样，最坏情况下外循环需执行 $\Omega(n)$ 次（习题[12-11]），总体运行时间为 $\mathcal{O}(n^2)$ 。

### 12.2.6 k-选取算法

以上从多个角度所做的尝试尽管有所收获，但就k-选取问题在最坏情况下的求解效率这一最终指标而言，均无实质性的突破。本节将延续以上quickSelect()算法的思路，介绍一个在最坏情况下运行时间依然为 $\mathcal{O}(n)$ 的k-选取算法。

#### ■ 算法

该方法的主要计算流程，可描述如算法12.1所示。

```

1 select(A, k)
2 输入：规模为n的无序序列A，秩k ≥ 0
3 输出：A所对应有序序列中秩为k的元素
4 {
5   0) if (n = |A| < Q) return trivialSelect(A, k); //递归基：序列规模不大时直接使用蛮力算法
6   1) 将A均匀地划分为n/Q个子序列，各含Q个元素；//Q为一个不大的常数，其具体数值稍后给出
7   2) 各子序列分别排序，计算中位数，并将这些中位数组成一个序列；//可采用任何排序算法，比如选择排序
8   3) 通过递归调用select()，计算出中位数序列的中位数，记作M；
9   4) 根据其相对于M的大小，将A中元素分为三个子集：L（小于）、E（相等）和G（大于）；
10  5) if (|L| ≥ k) return select(L, k);
11  else if (|L| + |E| ≥ k) return M;
12  else return select(G, k - |L| - |E|);
13 }

```

#### 算法12.1 线性时间的k-选取

#### ■ 正确性

该算法正确性的关键，在于其中第5)步中所涉及的递归。

实际上如图12.12所示，在第4)步依据全局中位数M对所有元素做过分类之后，可以假想地将三个子序列L、E和G按照大小次序自左向右排列。尽管这三个子集都有可能是空集，但无论如何，k-选取目标元素的位置无非三种可能。

其一如图(a)，子序列L足够长（ $|L| \geq k$ ）。此时，子序列E和G的存在与否与k-选取的结果无关，故可将它们剪除，并在L中继续做递归的k-选取。

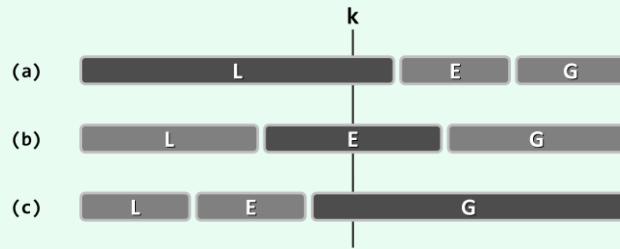


图12.12 k-选取目标元素所处位置的三种可能情况

其次如图(b)，子序列L长度不足 $k$ ，但在加入子序列E之后可以覆盖 $k$ 。此时，E中任何一个元素（均等于全局中位数M）都是所要查找的目标元素，故可直接返回M。

最后如图(c)，子序列L和E的长度总和仍不足 $k$ 。此时，目标元素必然落在子序列G中，故可将L和E剪除，并在G中继续做递归的( $k - |L| - |E|$ )-选取。

### ■ 复杂度

将该select()算法在最坏情况下的运行时间记作 $T(n)$ ，其中n为输入序列A的规模。

显然，第1)步只需 $\mathcal{O}(n)$ 时间。既然Q为常数，故在第2)步中，每一子序列的排序及中位数的计算只需常数时间，累计不过 $\mathcal{O}(n)$ 。第3)步为递归调用，因子序列长度为 $n/Q$ ，故经过 $T(n/Q)$ 时间即可得到全局的中位数M。第4)步依据M对所有元素做分类，为此只需做一趟线性遍历，累计亦不过 $\mathcal{O}(n)$ 时间。

那么，第5)步需要运行多少时间呢？考查第2)步所得各子序列的中位数。若按照这 $n/Q$ 个中位数（标记为m）的大小次序，将其所属子序列顺序排列，大致应如图12.13所示。在这些中位数中的居中者，即为第3)步计算出的全局中位数M。

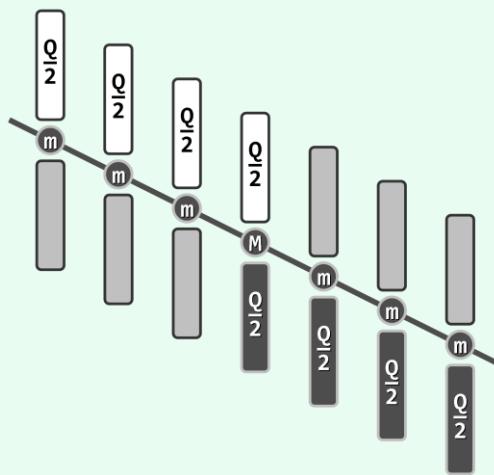


图12.13 各子序列的中位数以及全局中位数

由该图不难发现，至少有一半的子序列中，有半数的元素不小于M（在图中以白色示意）。同理，也至少有一半的子序列中，有半数的元素不大于M（在图中以黑色示意）。反过来，这两条性质也意味着，严格大于（小于）M的元素在全体元素中所占比例不会超过75%。

由此可知，子序列L与G的规模均不超过 $3n/4$ 。也就是说，算法的第5)步尽管会发生递归，但需进一步处理的序列的规模，绝不致超过原序列的 $3/4$ 。

综上，可得递推关系如下：

$$T(n) = cn + T(n/Q) + T(3n/4), c \text{ 为常数}$$

若取  $Q = 5$ ，则有

$$T(n) = cn + T(n/5) + T(3n/4) = O(20cn) = O(n)$$

## ■ 综合评价

上述 `selection()` 算法从理论上证实，的确可以在线性时间内完成  $k$ -选取。然而很遗憾，其线性复杂度中的常系数项过大，以致在通常规模的应用中难以真正体现出效率的优势。

该算法的核心技巧在于第2)和3)步，通过高效地将元素分组，分别计算中位数，并递归计算出这些中位数的中位数  $M$ ，使问题的规模得以按几何级数的速度递减，从而实现整体的高性能。

由此也可看出，中位数算法在一般性  $k$ -选取问题的求解过程中扮演着关键性角色，尽管前者只不过是后者的一个特例，但反过来也是其中难度最大者。

## § 12.3 \*希尔排序

### 12.3.1 递减增量策略

#### ■ 增量

希尔排序<sup>③</sup> (`Shellsort`) 算法首先将整个待排序向量  $A[ ]$  等效地视作一个二维矩阵  $B[ ][ ]$ 。

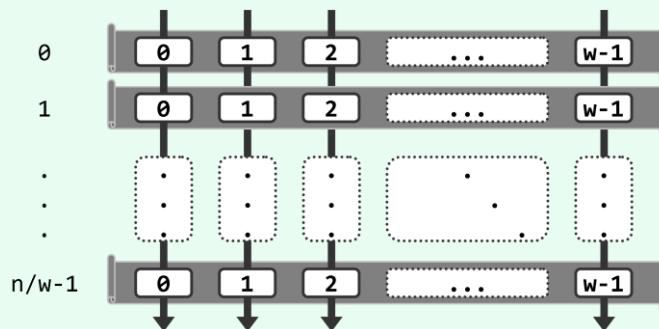


图12.14 将待排序向量视作二维矩阵

于是如图12.14所示，若原一维向量为  $A[0, n)$ ，则对于任一固定的矩阵宽度  $w$ ， $A$  与  $B$  中元素之间总有一一对应关系：

$$B[i][j] = A[i + jw]$$

或

$$A[k] = B[k \% w][k / w]$$

从秩的角度来看，矩阵  $B$  的各列依次对应于整数子集  $[0, n)$  关于宽度  $w$  的某一同余类。这也等效于从上到下、自左而右地将原向量  $A$  中的元素，依次填入矩阵  $B$  的各个单元。

为简化起见，以下不妨假设  $w$  整除  $n$ 。如此， $B$  中同属一列的元素自上而下依次对应于  $A$  中以  $w$  为间隔的  $n/w$  个元素。因此，矩阵的宽度  $w$  亦称作增量 (**increment**)。

<sup>③</sup> 最初版本由 D. L. Shell 于 1959 年发明<sup>[65]</sup>

## ■ 算法框架

希尔排序的算法框架，可以扼要地描述如下：

```

1 Shellsort(A, n)
2 输入：规模为n的无序向量A
3 输出：A对应的有序向量
4 {
5     取一个递增的增量序列：H = { w1 = 1, w2, w3, ..., wk, ... }
6     设k = max{i | wi < n}，即wk为增量序列H中小于n的最后一项
7     for (t = k; t > 0; t--) {
8         将向量A视作以wt为宽度的矩阵Bt
9         对Bt的每一列分别排序：Bt[i], i = 0, 1, ..., wt - 1
10    }
11 }
```

### 算法12.2 希尔排序

## ■ 增量序列

如图12.15所示，希尔排序是个迭代式重复的过程。

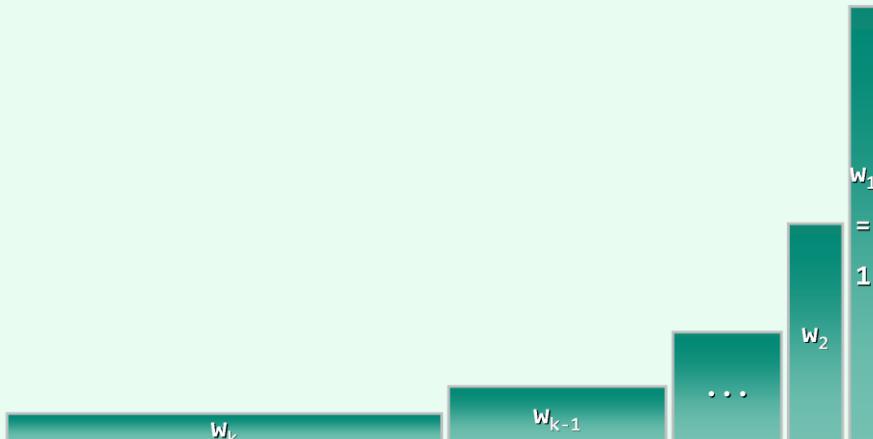


图12.15 递减增量、逐渐逼近策略

每一步迭代中，都从事先设定的某个整数序列中取出一项，并以该项为宽度，将输入向量重排为对应宽度的二维矩阵，然后逐列分别排序。当然，各步迭代并不需要真地从物理上重排原向量。事实上，借助以上一一对应关系，即可便捷地从逻辑上根据其在B[][]中的下标，访问统一保存于A[]中的元素。

不过，为便于对算法的理解，以下我们不妨仍然假想地进行这一重排转换。

因为增量序列中的各项是逆向取出的，所以各步迭代中矩阵的宽度呈缩减的趋势，直至最终使用w<sub>1</sub> = 1。矩阵每缩减一次并逐列排序一轮，向量整体的有序性就得以进一步改善。当增量缩减至1时，如图12.15最右侧所示，矩阵退化为单独的一列，故最后一步迭代中的“逐列排序”等效于对整个向量执行一次排序。这种通过不断缩减矩阵宽度而逐渐逼近最终输出的策略，称作递减增量（*diminishing increment*）算法，这也是希尔排序的另一名称。

以长度为13的向量：

{ 80, 23, 19, 40, 85, 1, 18, 92, 71, 8, 96, 46, 12 }

为例，对应的希尔排序过程及结果如图12.16所示。

秩k	列号	0	1	2	3	4	5	6	7	8	9	10	11	12
元素A[k]		80	23	19	40	85	1	18	92	71	8	96	46	12
分8列逐列排序之后	0	71								80				
	1		8								23			
	2			19								96		
	3				40								46	
	4					12								85
	5						1							
	6							18						
	7								92					
		71	8	19	40	12	1	18	92	80	23	96	46	12
分5列逐列排序之后	0	1					71					96		
	1		8					18					46	
	2			19					85					92
	3				40					80				
	4					12					23			
		1	8	19	40	12	71	18	85	80	23	96	46	92
分3列逐列排序之后	0	1			18			23				40		92
	1		8			12			85			96		
	2			19			46			71				80
		1	8	19	18	12	46	23	85	71	40	96	80	92
分2列逐列排序之后	0	1		12		19		23		71		92		96
	1		8		18		40		46		80		85	
		1	8	12	18	19	40	23	46	71	80	92	85	96
分1列逐列排序之后		1	8	12	18	19	23	40	46	71	80	85	92	96

图12.16 希尔排序实例：采用增量序列{ 1, 2, 3, 5, 8, 13, 21, ... }

## ■ 底层算法

最后一轮迭代等效于向量的整体排序，故无论此前各步如何迭代，最终必然输出有序向量，希尔排序的正确性毋庸置疑。然而反过来，我们却不禁有个疑问：既然如此，此前各步迭代中的逐列排序又有何必要？为何不直接做最后一次排序呢？这涉及到底层排序算法的特性。能够有效支持希尔排序的底层排序算法，必须是输入敏感的，比如3.5.2节所介绍的插入排序算法。

尽管该算法在最坏情况下需要运行 $\mathcal{O}(n^2)$ 时间，但随着向量的有序性不断提高（即逆序对的不断减少），运行时间将会锐减。具体地，根据习题[3-11]的结论，当逆序元素的间距均不超过k时，插入排序仅需 $\mathcal{O}(kn)$ 的运行时间。仍以图12.16为例，最后一步迭代（整体排序）之前，向量仅含两对逆序元素（40和23、92和85），其间距为1，故该步迭代仅需线性时间。

正是得益于这一特性，各步迭代对向量有序性的改善效果，方能不断积累下来，后续各步迭代的计算成本也能得以降低，并最终将总体成本控制在足以令人满意的范围。

### 12.3.2 增量序列

如算法12.2所示，希尔排序算法的主体框架已经固定，唯一可以调整的只是增量序列的设计与选用。事实上这一点也的确十分关键，不同的增量序列对插入排序以上特性的利用程度各异，算法的整体效率也相应地差异极大。以下将介绍几种典型的增量序列。

## ■ Shell序列

首先考查Shell本人在提出希尔算法之初所使用的序列：

$$\pi_{\text{shell}} = \{1, 2, 4, 8, 16, 32, \dots, 2^k, \dots\}$$

我们将看到，若使用这一序列，希尔排序算法在最坏情况下的性能并不好。

不妨取 $[0, 2^N]$ 内所有的 $n = 2^N$ 个整数，将其分为 $[0, 2^{N-1}]$ 和 $[2^{N-1}, 2^N]$ 两组，再分别打乱次序后组成两个随机子向量，最后将两个子向量逐项交替地归并为一个向量。比如 $N = 4$ 时，得到的向量可能如下（为便于区分，这里及以下，对两个子向量的元素分别做了提升和下移）：

$$\begin{array}{cccccccccccc} 11 & 4 & 14 & 3 & 10 & 0 & 15 & 1 & 9 & 6 & 8 & 7 & 13 & 2 & 12 & 5 \end{array}$$

请注意，在 $\pi_{\text{shell}}$ 中，首项之外的其余各项均为偶数。因此，在最后一步迭代之前，这两组元素的秩依然保持最初的奇偶性不变。如果把它们分别比作井水与河水，则尽管井水与河水各自都在流动，但毕竟“井水不犯河水”。

特别地，在经过倒数第二步迭代（ $w_2 = 2$ ）之后，尽管两组元素已经分别排序，但二者依然恪守各自的秩的奇偶性。仍以 $N = 4$ 为例，此时向量中各元素应排列如下：

$$\begin{array}{cccccccccccc} 8 & 0 & 9 & 1 & 10 & 2 & 11 & 3 & 12 & 4 & 13 & 5 & 14 & 6 & 15 & 7 \end{array}$$

准确地，此时元素k的秩为 $(2k + 1) \% (2^N + 1)$ 。对于每一 $1 \leq k \leq 2^{N-1}$ ，与其在最终有序向量中相距k个单元的元素各有2个，故最后一轮插入排序所做比较操作次数共计：

$$2 \times (1 + 2 + 3 + \dots + 2^{N-1}) = 2^{N-1} \cdot (2^{N-1} + 1) = \mathcal{O}(n^2)$$

反观这一实例可见，导致最后一轮排序低效的直接原因在于，此前的各步迭代尽管可以改善两组元素各自内部的有序性，但对二者之间有序性的改善却于事无补。究其根源在于，序列 $\pi_{\text{shell}}$ 中除首项外各项均被2整除。由此我们可以得到启发——为改进希尔排序的总体性能，首先必须尽可能减少不同增量值之间的公共因子。为此，一种彻底的方法就是保证它们之间两两互素。

不过，为更好地理解和分析如此设计的其它增量序列，需要略做一番准备。

## ■ 邮资问题

考查如下问题：

假设在某个国家，邮局仅发行面值分别为4分和13分的两种邮票，那么

1) 准备邮寄平信的你，可否用这两种邮票组合出对应的50分邮资？

2) 准备邮寄明信片的你，可否用这两种邮票组合出对应的35分邮资？

略作思考，即不难给出前一问的解答：使用六张4分面值的邮票，另加两张13分的。但对于后一问题，无论你如何绞尽脑汁，也不可能给出一种恰好的组合方案。

## ■ 线性组合

用数论的语言，以上问题可描述为： $4m + 13n = 35$ 是否存在自然数（非负整数）解？

对于任意自然数 $g$ 和 $h$ ，只要 $m$ 和 $n$ 也是自然数，则 $f = mg + nh$ 都称作 $g$ 和 $h$ 的一个组合（combination）。我们将不能由 $g$ 和 $h$ 组合生成出来的最大自然数记作 $x(g, h)$ 。

这里需要用到数论的一个基本结论：如果 $g$ 和 $h$ 互素，则必有

$$x(g, h) = (g - 1) \cdot (h - 1) - 1 = gh - g - h$$

就以上邮资问题而言， $g = 4$ 与 $h = 13$ 互素，故有

$$x(4, 13) = 3 \times 12 - 1 = 35$$

也就是说，35恰为无法由4和13组合生成的最大自然数。

## ■ $h$ -有序与 $h$ -排序

在向量 $S[\theta, n)$ 中，若 $S[i] \leq S[i + h]$ 对任何 $\theta \leq i < n - h$ 均成立，则称该向量 $h$ -有序（ $h$ -ordered）。也就是说，其中相距 $h$ 个单元的每对元素之间均有序。

考查希尔排序中对应于任一增量 $h$ 的迭代。如前所述，该步迭代需将原向量“折叠”成宽度为 $h$ 的矩阵，并对各列分别排序。就效果而言，这等同于在原向量中以 $h$ 为间隔排序，故这一过程称作 $h$ -排序（ $h$ -sorting）。不难看出，经 $h$ -排序之后的向量必然 $h$ -有序。

关于 $h$ -有序和 $h$ -排序，Knuth<sup>[3]</sup>给出了一个重要结论（习题[12-12]和[12-13]）：

**已经 $g$ -有序的向量，再经 $h$ -排序之后，依然保持 $g$ -有序**

也就是说，此时该向量既是 $g$ -有序的，也是 $h$ -有序的，称作 $(g, h)$ -有序。

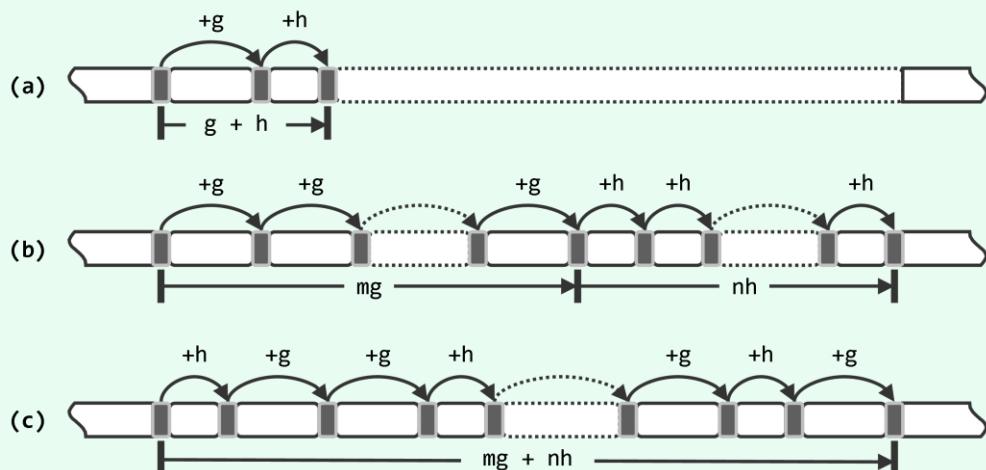


图12.17  $(g, h)$ -有序向量必然 $(mg + nh)$ -有序

考查 $(g, h)$ -有序的任一向量 $S$ 。如图12.17(a)所示，借助有序性的传递律可知，相距 $g + h$ 的任何一对元素都必有序，故 $S$ 必然 $(g + h)$ -有序。推而广之，如图(b)和(c)所示可知，对于任意非负整数 $m$ 和 $n$ ，相距 $mg + nh$ 的任何一对元素都必有序，故 $S$ 必然 $(mg + nh)$ -有序。

### ■ 有序性的保持与加强

根据以上Knuth所指出的性质，随着 $h$ 不断递减， $h$ -有序向量整体的有序性必然逐步改善。特别地，最终 $1$ -有序的向量，即是全局有序的向量。

为更准确地验证以上判断，可如图12.18所示，考查与任一元素 $S[i]$ 构成逆序对（习题[3-11]）的后继元素。

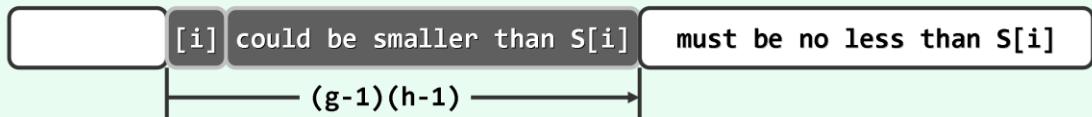


图12.18 经多步迭代，逆序元素可能的范围必然不断缩小

在分别做过 $g$ -排序与 $h$ -排序之后，根据Knuth的结论可知该向量必已 $(g, h)$ -有序。由以上分析，对于 $g$ 和 $h$ 的任一线性组合 $mg + nh$ ，该向量也应 $(mg + nh)$ -有序。因此反过来，逆序对的间距必不可能是 $g$ 和 $h$ 的组合。而根据此前所引数论中的结论，只要 $g$ 和 $h$ 互素，则如图12.18所示，逆序对的间距就绝不可能大于 $(g - 1) \cdot (h - 1)$ 。

由此可见，希尔排序过程中向量的有序性之所以会不断积累并改善，其原因可解释为，向量中每个元素所能参与构成的逆序对持续减少，整个向量所含逆序对的总数也持续减少。与此同时，随着逆序对的减少，底层所采用的插入排序算法的实际执行时间，也将不断减少，从而提高希尔排序的整体效率。以下结合具体的增量序列，就此做出定量的估计。

### ■ $(g, h)$ -有序与排序成本

设某向量 $S$ 已属 $(g, h)$ -有序，且假设 $g$ 和 $h$ 的数值均处于 $\mathcal{O}(d)$ 数量级，以下考查对该向量做 $d$ -排序所需的时间成本。

据其定义， $d$ -排序需将 $S$ 等间距地划分为长度各为 $\mathcal{O}(n / d)$ 的 $d$ 个子向量，并分别排序。由以上分析，在 $(g, h)$ -有序的向量中，逆序对的间距不超过

$$(g - 1) \cdot (h - 1)$$

故就任何一个子向量的内部而言，逆序对的间距应不超过

$$(g - 1) \cdot (h - 1) / d = \mathcal{O}(d)$$

再次根据习题[3-11]的结论，采用插入排序算法可在：

$$\mathcal{O}(d) \cdot (n / d) = \mathcal{O}(n)$$

的时间内，完成每一子向量的排序；于是，所有子向量的排序总体消耗的时间应不超过 $\mathcal{O}(dn)$ 。

### ■ Papernov-Stasevic序列

现在，可以回到增量序列的优化设计问题。按照此前“尽力避免增量值之间公共因子”的思路，Papernov和Stasevic于1965年提出了另一增量序列：

$$\mathcal{Z}_{ps} = \{1, 3, 7, 15, 31, 63, \dots, 2^k - 1, \dots\}$$

不难看出，其中相邻各项的确互素。我们将看到，采用这一增量序列，希尔排序算法的性能可以改进至 $\mathcal{O}(n^{3/2})$ ，其中 $n$ 为待排序向量的规模。

在序列 $\mathcal{H}_{ps}$ 的各项中，设 $w_t$ 为与 $n^{1/2}$ 最接近者，亦即 $w_t = \Theta(n^{1/2})$ 。以下将希尔排序算法过程中的所有迭代分为两类，分别估计其运行时间。

首先，考查在 $w_t$ 之前执行的各步迭代。

这类迭代所对应的增量均满足 $w_k > w_t$ ，或等价地， $k > t$ 。在每一次这类迭代中，矩阵共有 $w_k$ 列，各列包含 $\mathcal{O}(n/w_k)$ 个元素。因此，若采用插入排序算法，各列分别耗时 $\mathcal{O}((n/w_k)^2)$ ，所有列共计耗时 $\mathcal{O}(n^2/w_k)$ 。于是，此类迭代各自所需的时间 $\mathcal{O}(n^2/w_k)$ 构成一个大致以2为比例的几何级数，其总和应线性正比于其中最大的一项，亦即不超过

$$\mathcal{O}(2 \cdot n^2/w_t) = \mathcal{O}(n^{3/2})$$

对称地，再来考查 $w_t$ 之后的各步迭代。

这类迭代所对应的增量均满足 $w_k < w_t$ ，或等价地， $k < t$ 。考虑到此前刚刚完成 $w_{k+1}$ -排序和 $w_{k+2}$ -排序，而来自 $\mathcal{H}_{ps}$ 序列的 $w_{k+1}$ 和 $w_{k+2}$ 必然互素，且与 $w_k$ 同处一个数量级。因此根据此前结论，每一次这样的迭代至多需要 $\mathcal{O}(n \cdot w_k)$ 时间。同样地，这类迭代所需的时间 $\mathcal{O}(n \cdot w_k)$ 也构成一个大致以2为比例的几何级数，其总和也应线性正比于其中最大的一项，亦即不超过

$$\mathcal{O}(2 \cdot n \cdot w_t) = \mathcal{O}(n^{3/2})$$

综上可知，采用 $\mathcal{H}_{ps}$ 序列的希尔排序算法，在最坏情况下的运行时间不超过 $\mathcal{O}(n^{3/2})$ 。

### ■ Pratt序列

Pratt于1971年也提出了自己的增量序列：

$$\mathcal{H}_{pratt} = \{1, 2, 3, 4, 6, 8, 9, 12, 16, \dots, 2^p 3^q, \dots\}$$

可见，其中各项除2和3外均不含其它素因子。

可以证明，采用 $\mathcal{H}_{pratt}$ 序列，希尔排序算法至多运行 $\mathcal{O}(n \log^2 n)$ 时间（习题[12-14]）。

### ■ Sedgewick序列

尽管Pratt序列的效率较高，但因其中各项的间距太小，会导致迭代趟数过多。为此，Sedgewick<sup>[66]</sup>综合Papernov-Stasevic序列与Pratt序列的优点，提出了以下增量序列：

$$\mathcal{H}_{sedgewick} = \{1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, 8929, \dots\}$$

其中各项，均为：

$$9 \cdot 4^k - 9 \cdot 2^k + 1$$

或

$$4^k - 3 \cdot 2^k + 1$$

的形式。

如此改进之后，希尔排序算法在最坏情况下的时间复杂度为 $\mathcal{O}(n^{4/3})$ ，平均复杂度为 $\mathcal{O}(n^{7/6})$ 。更重要的是，在通常的应用环境中，这一增量序列的综合效率最佳。