排序和顺序统计量

这一部分介绍了几种解决如下**排序问题**的算法：

输入：一个 n 个数的序列$\langle a_1, a_2, \cdots, a_n\rangle$。

输出：输入序列的一个排列(重排)$\langle a_1', a_2', \cdots, a_n'\rangle$，使得 $a_1' \leqslant a_2' \leqslant \cdots \leqslant a_n'$。

输入序列通常是一个 n 元数组，尽管它可以用链表等其他方式描述。

数据的结构

在实际中，待排序的数很少是单独的数值，它们通常是称为**记录**(record)的数据集的一部分。每个记录包含一个**关键字**(key)，就是排序问题中要重排的值。记录的剩余部分由**卫星数据**(satellite data)组成，通常与关键字是一同存取的。在实际中，当一个排序算法重排关键字时，也必须要重排卫星数据。如果每个记录包含大量卫星数据，我们通常重排记录指针的数组，而不是记录本身，这样可以降低数据移动量。

在某种意义上，正是这些实现细节将一个算法与成熟的程序区分开来。一个排序算法描述确定有序次序的**方法**(method)，而不管我们是在排序单独的数还是包含很多卫星数据的大记录。因此，当关注排序问题时，我们通常假定输入只是由数组成。将一个对数进行排序的算法转换为一个对记录进行排序的程序在概念上是很直接的，当然在具体的工程情境下，其他一些细节问题可能会使实际的编程工作遇到很多挑战。

为什么要排序

很多计算机科学家认为排序是算法研究中最基础的问题，其原因有很多：

- 有时应用本身就需要对信息进行排序。例如，为了准备用户财务报表，银行需要按编号对支票进行排序。
- 很多算法通常把排序作为关键子程序。例如，在一个渲染图形对象的程序中，图形对象是分层叠在一起的，这个程序可能就需要

按“上层”关系来排序对象，以便能按自底向上的次序绘制对象。在本书中，我们将看到大量的算法将排序作为子程序来使用。

- 现有的排序算法数量非常庞大，其中所使用的技术也非常丰富。实际上，很多重要的算法设计技术都体现在多年来研究者所设计的排序算法中。从这个角度看，排序问题还有很好的历史价值。
- 我们可以证明排序问题的一个非平凡下界(在第 8 章中，我们会给出这个证明)。而我们的最佳上界能够与这个非平凡下界渐近相等，这就意味着我们介绍的算法是渐近最优的。而且，我们可以利用排序问题的下界来证明其他问题的下界。
- 在实现排序算法时会出现很多工程问题。某个特定环境下的最快的排序算法可能依赖很多因素，例如，关于关键字和卫星数据的先验知识、计算机主机的内存层次(缓存和虚拟内存)和软件环境。很多这类问题最好在算法层面来处理，而不是通过“代码调优”来解决。

排序算法

我们在第 2 章已经介绍了两种排序算法。插入排序最坏情况下可以在 $\Theta(n^2)$时间内将 n 个数排好序。但是，由于其内层循环非常紧凑，对于小规模输入，插入排序是一种非常快的原址排序算法(回忆一下，如果输入数组中仅有常数个元素需要在排序过程中存储在数组之外，则称排序算法是**原址的**(in place))。归并排序有更好的渐近运行时间 $\Theta(n\lg n)$，但它所使用的 MERGE 过
148 程并不是原址的。

在这一部分中，我们将介绍两种新的排序算法，它们可以排序任意的实数。第 6 章将介绍堆排序，这是一种 $O(n\lg n)$时间的原址排序算法。它使用了一种被称为堆的重要数据结构，堆还可用来实现优先队列。

第 7 章介绍快速排序，它也是一种原址排序算法，但最坏情况运行时间为 $\Theta(n^2)$。然而它的期望运行时间为 $\Theta(n\lg n)$，而且在实际应用中通常比堆排序快。与插入排序类似，快速排序的代码也很紧凑，因此运行时间中隐含的常数系数很小。快速排序是排序大数组的最常用算法。

插入排序、归并排序、堆排序及快速排序都是比较排序算法：它们都是通过对元素进行比较操作来确定输入数组的有序次序。第 8 章首先介绍了决策树模型，可用来研究比较排序算法的性能局限。使用决策树模型，我们可以证明任意比较排序算法排序 n 个元素的最坏情况运行时间的下界为 $\Omega(n\lg n)$，从而证明堆排序和归并排序是渐近最优的比较排序算法。

第 8 章接下来展示了：如果通过比较操作之外的方法来获得输入序列有序次序的信息，就有可能打破 $\Omega(n\lg n)$的下界。例如，计数排序算法假定输入元素的值均在集合$\{0, 1, \cdots, k\}$内。通过使用数组索引作为确定相对次序的工具，计数排序可以在 $\Theta(k+n)$的时间内将 n 个数排好序。因此，当 $k=O(n)$时，计数排序算法的运行时间与输入数组的规模呈线性关系。另外一种相关的排序算法——基数排序，可以用来扩展计数排序的适用范围。如果有 n 个整数要进行排序，每个整数有 d 位数字，并且每个数字可能取 k 个值，那么基数排序就可以在 $\Theta(d(n+k))$时间内完成排序工作。当 d 是常数且 $k=O(n)$时，基数排序的运行时间就是线性的。第 8 章介绍的第三种算法是桶排序算法，它需要了解输入数组中数据的概率分布。对于半开区间$[0, 1)$内服从均匀分布的 n 个实数，桶排序的平均情况运行时间为 $O(n)$。

下表总结了第 2 章和第 6～8 章介绍的排序算法的运行时间，其中 n 表示要排序的数据项数量。对于计数排序，数据项均在集合$\{0, 1, \cdots, k\}$内。对于基数排序，每个数据项都是 d 位数字的整数，每位数字可能取 k 个值。对于桶排序，假定关键字是半开区间$[0, 1)$内服从均匀分布

的 n 个实数。表的最右一列给出了平均情况或期望运行时间，可能与最坏情况运行时间不同。我们忽略了堆排序的平均情况运行时间，因为本书中并未对其进行分析。 149

算　法	最坏情况运行时间	平均情况/期望运行时间
插入排序	$\Theta(n^2)$	$\Theta(n^2)$
归并排序	$\Theta(n\lg n)$	$\Theta(n\lg n)$
堆排序	$O(n\lg n)$	—
快速排序	$\Theta(n^2)$	$\Theta(n\lg n)$(期望)
计数排序	$\Theta(k+n)$	$\Theta(k+n)$
基数排序	$\Theta(d(n+k))$	$\Theta(d(n+k))$
桶排序	$\Theta(n^2)$	$\Theta(n)$(平均情况)

顺序统计量

一个 n 个数的集合的第 i 个顺序统计量就是集合中第 i 小的数。当然，我们可以通过将输入集合排序，取输出的第 i 个元素来选择第 i 个顺序统计量。当不知道输入数据的分布时，这种方法的运行时间为 $\Omega(n\lg n)$，即第 8 章中所证明的比较排序算法的下界。

在第 9 章中，我们展示了即使输入数据是任意实数，也可以在$O(n)$时间内找到第 i 小的元素。我们提出了一种随机算法，其伪代码非常紧凑，它的最坏情况运行时间为 $\Theta(n^2)$，但期望运行时间为 $O(n)$。我们还给出了一种更复杂的算法，最坏情况运行时间为 $O(n)$。

背景

虽然这一部分的大部分内容并不依赖高深的数学知识，但一些章节还是需要一些稍微复杂的数学知识。特别地，快速排序、桶排序和顺序统计量算法的分析要用到概率知识(附录 C 中回顾了概率知识)以及第 5 章中介绍的概率分析和随机算法。顺序统计量算法的最坏情况线性时间分析涉及的数学知识比本部分中其他最坏情况分析要更复杂些。 150

第6章
Introduction to Algorithms, Third Edition

堆 排 序

在本章中，我们会介绍另一种排序算法：堆排序(heapsort)。与归并排序一样，但不同于插入排序的是，堆排序的时间复杂度是$O(n\lg n)$。而与插入排序相同，但不同于归并排序的是，堆排序同样具有空间原址性：任何时候都只需要常数个额外的元素空间存储临时数据。因此，堆排序是集合了我们目前已经讨论的两种排序算法优点的一种排序算法。

堆排序引入了另一种算法设计技巧：使用一种我们称为“堆”的数据结构来进行信息管理。堆不仅用在堆排序中，而且它也可以构造一种有效的优先队列。在后续的章节中，我们还将多次在算法中引入堆。

虽然“堆”这一词源自堆排序，但是目前它已经被引申为“垃圾收集存储机制”，例如在Java和Lisp语言中所定义的。强调一下，我们使用的堆不是垃圾收集存储，并且在本书的任何部分，只要涉及堆，指的都是堆数据结构，而不是垃圾收集存储。

6.1 堆

如图6-1所示，**(二叉)堆**是一个数组，它可以被看成一个近似的完全二叉树(见B.5.3节)。树上的每一个结点对应数组中的一个元素。除了最底层外，该树是完全充满的，而且是从左向右填充。表示堆的数组A包括两个属性：$A.length$(通常)给出数组元素的个数，$A.heap\text{-}size$表示有多少个堆元素存储在该数组中。也就是说，虽然$A[1..A.length]$可能都存有数据，但只有$A[1..A.heap\text{-}size]$中存放的是堆的有效元素，这里，$0 \leqslant A.heap\text{-}size \leqslant A.length$。树的根结点是
151 $A[1]$，这样给定一个结点的下标i，我们很容易计算得到它的父结点、左孩子和右孩子的下标：

```
PARENT (i)
1  return ⌊i/2⌋

LEFT (i)
1  return 2i

RIGHT (i)
1  return 2i+1
```

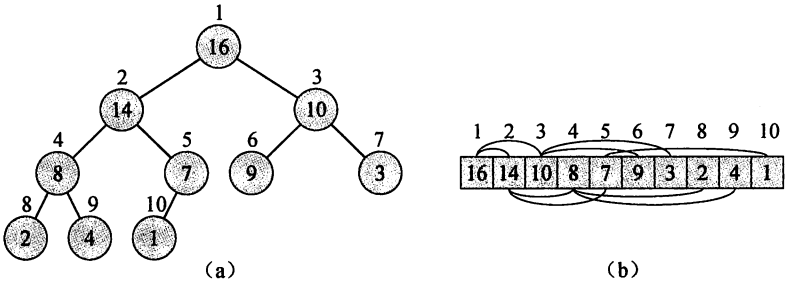

图6-1 以(a)二叉树和(b)数组形式展现的一个最大堆。每个结点圆圈内部的数字是它所存储的数据。结点上方的数字是它在数组中相应的下标。数组上方和下方的连线显示的是父-子关系：父结点总是在它的孩子结点的左边。该树的高度为3，下标为4(值为8)的结点的高度为1

在大多数计算机上，通过将 i 的值左移一位，LEFT 过程可以在一条指令内计算出 $2i$。采用类似方法，在 RIGHT 过程中也可以通过将 i 的值左移 1 位并在低位加 1，快速计算得到 $2i+1$。至于 PARENT 过程，则可以通过把 i 的值右移 1 位计算得到$\lfloor i/2 \rfloor$。在堆排序的好的实现中，这三个函数通常是以“宏”或者“内联函数”的方式实现的。

二叉堆可以分为两种形式：最大堆和最小堆。在这两种堆中，结点的值都要满足堆的性质，但一些细节定义则有所差异。在**最大堆**中，**最大堆性质**是指除了根以外的所有结点 i 都要满足：

$$A[\text{PARENT}(i)] \geqslant A[i]$$

也就是说，某个结点的值至多与其父结点一样大。因此，堆中的最大元素存放在根结点中；并且，在任一子树中，该子树所包含的所有结点的值都不大于该子树根结点的值。**最小堆**的组织方式正好相反：**最小堆性质**是指除了根以外的所有结点 i 都有 152

$$A[\text{PARENT}(i)] \leqslant A[i]$$

最小堆中的最小元素存放在根结点中。

在堆排序算法中，我们使用的是最大堆。最小堆通常用于构造优先队列，在 6.5 节中，我们会再具体讨论。对于某个特定的应用来说，我们必须明确需要的是最大堆还是最小堆；而当某一属性既适合于最大堆也适合于最小堆的时候，我们就只使用“堆”这一名词。

如果把堆看成是一棵树，我们定义一个堆中的结点的**高度**就为该结点到叶结点最长简单路径上边的数目；进而我们可以把堆的高度定义为根结点的高度。既然一个包含 n 个元素的队可以看做一棵完全二叉树，那么该堆的高度是 $\Theta(\lg n)$(见练习 6.1-2)。我们会发现，堆结构上的一些基本操作的运行时间至多与树的高度成正比，即时间复杂度为 $O(\lg n)$。在本章的剩余部分中，我们将介绍一些基本过程，并说明如何在排序算法和优先队列中应用它们。

- MAX-HEAPIFY 过程：其时间复杂度为 $O(\lg n)$，它是维护最大堆性质的关键。
- BUILD-MAX-HEAP 过程：具有线性时间复杂度，功能是从无序的输入数据数组中构造一个最大堆。
- HEAPSORT 过程：其时间复杂度为 $O(n \lg n)$，功能是对一个数组进行原址排序。
- MAX-HEAP-INSERT、HEAP-EXTRACT-MAX、HEAP-INCREASE-KEY 和 HEAP-MAXIMUM 过程：时间复杂度为 $O(\lg n)$，功能是利用堆实现一个优先队列。

练习

6.1-1 在高度为 h 的堆中，元素个数最多和最少分别是多少？

6.1-2 证明：含 n 个元素的堆的高度为$\lfloor \lg n \rfloor$。

6.1-3 证明：在最大堆的任一子树中，该子树所包含的最大元素在该子树的根结点上。 153

6.1-4 假设一个最大堆的所有元素都不相同，那么该堆的最小元素应该位于哪里？

6.1-5 一个已排好序的数组是一个最小堆吗？

6.1-6 值为〈23，17，14，6，13，10，1，5，7，12〉的数组是一个最大堆吗？

6.1-7 证明：当用数组表示存储 n 个元素的堆时，叶结点下标分别是$\lfloor n/2 \rfloor+1$，$\lfloor n/2 \rfloor+2$，…，n。

6.2 维护堆的性质

MAX-HEAPIFY 是用于维护最大堆性质的重要过程。它的输入为一个数组 A 和一个下标 i。在调用 MAX-HEAPIFY 的时候，我们假定根结点为 LEFT(i)和 RIGHT(i)的二叉树都是最大堆，但这时 $A[i]$有可能小于其孩子，这样就违背了最大堆的性质。MAX-HEAPIFY 通过让 $A[i]$的值在最大堆中“逐级下降”，从而使得以下标 i 为根结点的子树重新遵循最大堆的性质。

```
MAX-HEAPIFY (A, i )
 1  l= LEFT (i)
 2  r= RIGHT (i)
 3  if l ⩽ A. heap-size and A[l] > A[i]
 4       largest =l
 5  else largest = i
 6  if r ⩽ A. heap-size and A[r] > A[largest]
 7       largest = r
 8  if larget ≠ i
 9       exchange A[i] with A[largest]
10       MAX-HEAPIFY (A, largest )
```

图 6-2 图示了 MAX-HEAPIFY 的执行过程。在程序的每一步中，从 $A[i]$、$A[\text{LEFT}(i)]$和 $A[\text{RIGHT}(i)]$中选出最大的，并将其下标存储在 *largest* 中。如果 $A[i]$是最大的，那么以 i 为根结点的子树已经是最大堆，程序结束。否则，最大元素是 i 的某个孩子结点，则交换 $A[i]$和
154 $A[largest]$的值。从而使 i 及其孩子都满足最大堆的性质。在交换后，下标为 *largest* 的结点的值是原来的 $A[i]$，于是以该结点为根的子树又有可能会违反最大堆的性质。因此，需要对该子树递归调用 MAX-HEAPIFY。

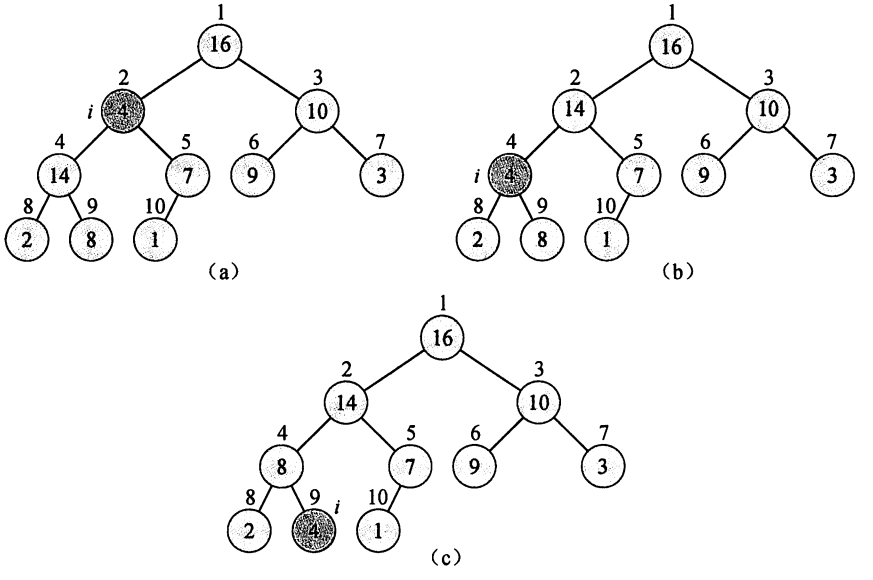

图 6-2 当 *A. heap-size*=10 时，MAX-HEAPIFY(A, 2)的执行过程。(a)初始状态，在结点 i=2 处，$A[2]$违背了最大堆性质，因为它的值不大于它的孩子。在(b)中，通过交换 $A[2]$和 $A[4]$的值，结点 2 恢复了最大堆的性质，但又导致结点 4 违反了最大堆的性质。递归调用 MAX-HEAPIFY(A, 4)，此时 i=4。在(c)中，通过交换 $A[4]$和 $A[9]$的值，结点 4 的最大堆性质得到了恢复。再次递归调用 MAX-HEAPIFY(A, 9)，此时不再有新的数据交换

对于一棵以 i 为根结点、大小为 n 的子树，MAX-HEAPIFY 的时间代价包括：调整 $A[i]$、$A[\text{LEFT}(i)]$和 $A[\text{RIGHT}(i)]$的关系的时间代价 $\Theta(1)$，加上在一棵以 i 的一个孩子为根结点的子树上运行 MAX-HEAPIFY 的时间代价(这里假设递归调用会发生)。因为每个孩子的子树的大小至多为 $2n/3$(最坏情况发生在树的最底层恰好半满的时候)，我们可以用下面这个递归式刻画 MAX-HEAPIFY 的运行时间：

155
$$T(n) \leqslant T(2n/3) + \Theta(1)$$

根据主定理(定理 4.1)的情况 2，上述递归式的解为 $T(n)=O(\lg n)$。也就是说，对于一个树高为

h 的结点来说，MAX-HEAPIFY 的时间复杂度是 $O(h)$。

练习

6.2-1 参照图 6-2 的方法，说明 MAX-HEAPIFY(A，3)在数组 $A=\langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0\rangle$上的操作过程。

6.2-2 参考过程 MAX-HEAPIFY，写出能够维护相应最小堆的 MIN-HEAPIFY(A，i)的伪代码，并比较 MIN-HEAPIFY 与 MAX-HEAPIFY 的运行时间。

6.2-3 当元素 $A[i]$比其孩子的值都大时，调用 MAX-HEAPIFY($A.i$)会有什么结果？

6.2-4 当 $i>A.heap\text{-}size/2$ 时，调用 MAX-HEAPIFY(A，i)会有什么结果？

6.2-5 MAX-HEAPIFY 的代码效率较高，但第 10 行中的递归调用可能例外，它可能使某些编译器产生低效的代码。请用循环控制结构取代递归，重写 MAX-HEAPIFY 代码。

6.2-6 证明：对一个大小为 n 的堆，MAX-HEAPIFY 的最坏情况运行时间为 $\Omega(\lg n)$。(提示：对于 n 个结点的堆，可以通过对每个结点设定恰当的值，使得从根结点到叶结点路径上的每个结点都会递归调用 MAX-HEAPIFY。)

6.3 建堆

我们可以用自底向上的方法利用过程 MAX-HEAPIFY 把一个大小为 $n=A.length$ 的数组 $A[1..n]$转换为最大堆。通过练习 6.1-7 可以知道，子数组 $A(\lfloor n/2\rfloor+1..n)$中的元素都是树的叶结点。每个叶结点都可以看成只包含一个元素的堆。过程 BUILD-MAX-HEAP 对树中的其他结点都调用一次 MAX-HEAPIFY。 156

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

图 6-3 给出了 BUILD-MAX-HEAP 过程的一个例子。

为了证明 BUILD-MAX-HEAP 的正确性，我们使用如下的循环不变量：

> 在第 2～3 行中每一次 **for** 循环的开始，结点 $i+1$，$i+2$，…，n 都是一个最大堆的根结点。

我们需要证明这一不变量在第一次循环前为真，并且每次循环迭代都维持不变。当循环结束时，这一不变量可以用于证明正确性。

初始化：在第一次循环迭代之前，$i=\lfloor n/2\rfloor$，而$\lfloor n/2\rfloor+1$，$\lfloor n/2\rfloor+2$，…，n 都是叶结点，因而是平凡最大堆的根结点。

保持：为了看到每次迭代都维护这个循环不变量，注意到结点 i 的孩子结点的下标均比 i 大。所以根据循环不变量，它们都是最大堆的根。这也是调用 MAX-HEAPIFY(A，i)使结点 i 成为一个最大堆的根的先决条件。而且，MAX-HEAPIFY 维护了结点 $i+1$，$i+2$，…，n 都是一个最大堆的根结点的性质。在 **for** 循环中递减 i 的值，为下一次循环重新建立循环不变量。

终止：过程终止时，$i=0$。根据循环不变量，每个结点 1，2，…，n 都是一个最大堆的根。特别需要指出的是，结点 1 就是最大的那个堆的根结点。

我们可以用下面的方法简单地估算 BUILD-MAX-HEAP 运行时间的上界。每次调用 MAX-HEAPIFY 的时间复杂度是 $O(\lg n)$，BUILD-MAX-HEAP 需要 $O(n)$次这样的调用。因此总的时间复杂度是 $O(n\lg n)$。当然，这个上界虽然正确，但不是渐近紧确的。

我们还可以进一步得到一个更紧确的界。可以观察到，不同结点运行 MAX-HEAPIFY 的时间与

该结点的树高相关，而且大部分结点的高度都很小。因此，利用如下性质可以得到一个更紧确的界：包含 n 个元素的堆的高度为 $\lfloor \lg n \rfloor$(见练习 6.1-2)；高度为 h 的堆最多包含 $\lceil n/2^{h+1} \rceil$ 个结点(见练习 6.3-3)。

在一个高度为 h 的结点上运行 MAX-HEAPIFY 的代价是 $O(h)$，我们可以将 BUILD-MAX-HEAP 的总代价表示为

157

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right)$$

最后的一个累积和的计算可以用 $x=1/2$ 带入公式(A.8)得到，则有

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

于是，我们可以得到 BUILD-MAX-HEAP 的时间复杂度：

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n)$$

因此，我们可以在线性时间内，把一个无序数组构造成为一个最大堆。

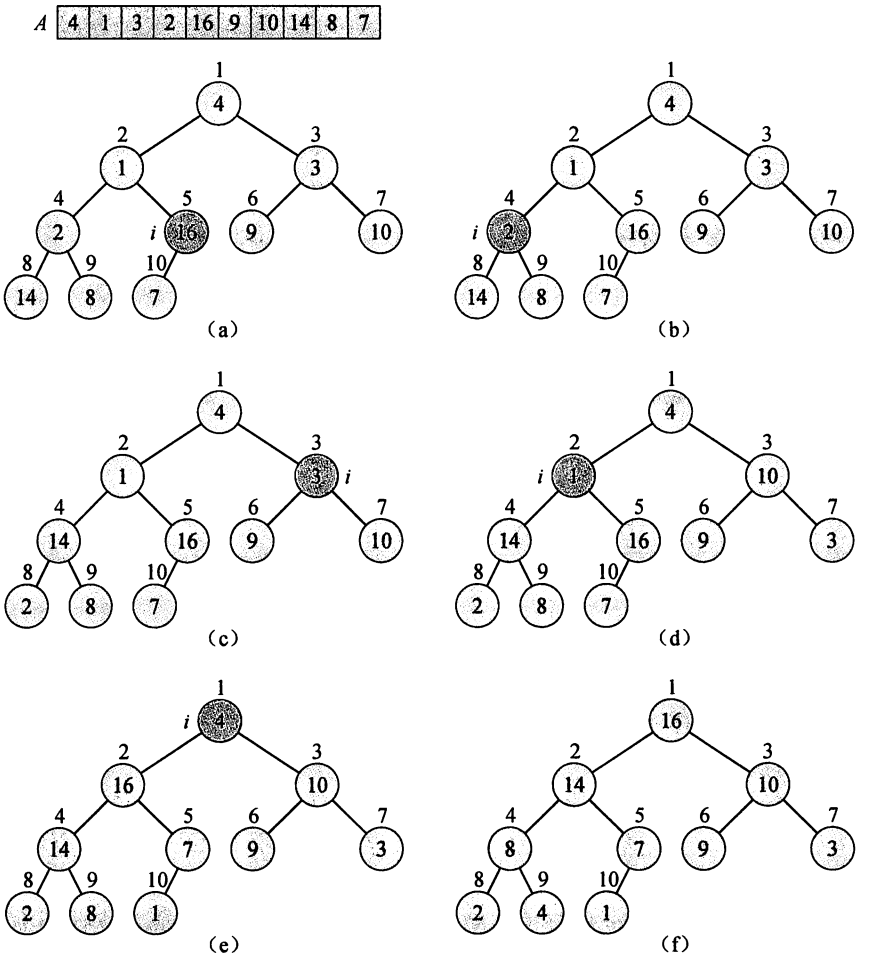

图 6-3 BUILD-MAX-HEAP 的操作过程示意图，显示了在 BUILD-MAX-HEAP 的第 3 行调用 MAX-HEAPIFY 之前的数据结构。(a)一个包括 10 个元素的输入数组及其对应的二叉树。图中显示的是调用 MAX-HEAPIFY(A, i)前，循环控制变量 i 指向结点 5 的情况。(b)操作结果的数据结构。下一次迭代，循环控制变量 i 指向结点 4。(c)～(e)BUILD-MAX-HEAP 中 **for** 循环的后续迭代操作。需要注意的是，任何时候在某个结点调用 MAX-HEAPIFY，该结点的两个子树都是最大堆。(f)执行完 BUILD-MAX-HEAP 时的最大堆

类似地，我们也可以通过调用 BUILD-MIN-HEAP 构造一个最小堆。除了第 3 行的调用替换为 MIN-HEAPIFY(见练习 6.2-2)以外，BUILD-MIN-HEAP 与 BUILD-MAX-HEAP 完全相同。BUILD-MIN-HEAP 可以在线性时间内，把一个无序数组构造成为一个最小堆。

练习

6.3-1 参照图 6-3 的方法，说明 BUILD-MAX-HEAP 在数组 $A=\langle 5, 3, 17, 10, 84, 19, 6, 22, 9\rangle$ 上的操作过程。

6.3-2 对于 BUILD-MAX-HEAP 中第 2 行的循环控制变量 i 来说，为什么我们要求它是从 $\lfloor A.length/2\rfloor$ 到 1 递减，而不是从 1 到 $\lfloor A.length/2\rfloor$ 递增呢？

6.3-3 证明：对于任一包含 n 个元素的堆中，至多有 $\lceil n/2^{h+1}\rceil$ 个高度为 h 的结点？

6.4 堆排序算法

158
~
159

初始时候，堆排序算法利用 BUILD-MAX-HEAP 将输入数组 $A[1..n]$ 建成最大堆，其中 $n=A.length$。因为数组中的最大元素总在根结点 $A[1]$ 中，通过把它与 $A[n]$ 进行互换，我们可以让该元素放到正确的位置。这时候，如果我们从堆中去掉结点 n(这一操作可以通过减少 $A.heap\text{-}size$ 的值来实现)，剩余的结点中，原来根的孩子结点仍然是最大堆，而新的根结点可能会违背最大堆的性质。为了维护最大堆的性质，我们要做的是调用 MAX-HEAPIFY(A, 1)，从而在 $A[1..n-1]$ 上构造一个新的最大堆。堆排序算法会不断重复这一过程，直到堆的大小从 $n-1$ 降到 2。(准确的循环不变量定义见练习 6.4-2。)

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

图 6-4 给出了一个在 HEAPSORT 的第 1 行建立初始最大堆之后，堆排序操作的一个例子。图 6-4 显示了第 2～5 行 **for** 循环第一次迭代开始前最大堆的情况和每一次迭代之后最大堆的情况。

HEAPSORT 过程的时间复杂度是 $O(n\lg n)$，因为每次调用 BUILD-MAX-HEAP 的时间复杂度是 $O(n)$，而 $n-1$ 次调用 MAX-HEAPIFY，每次的时间为 $O(\lg n)$。

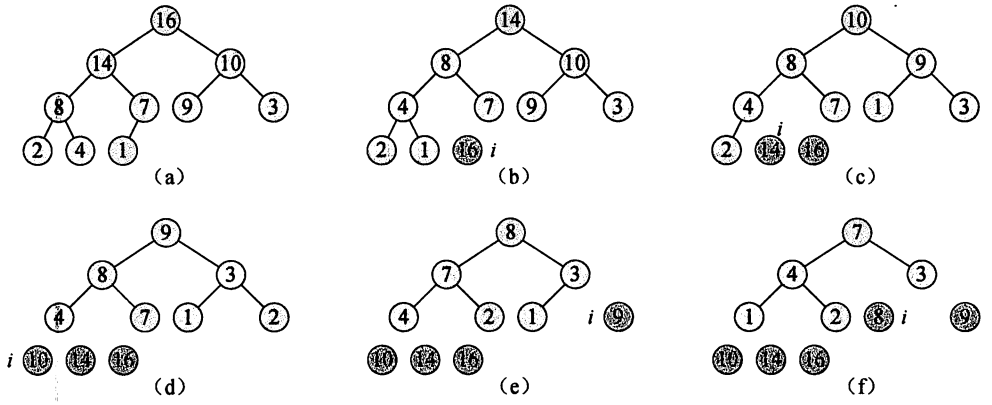

图 6-4 HEAPSORT 的运行过程。(a)执行堆排序算法第 1 行，用 BUILD-MAX-HEAP 构造得到的最大堆。(b)～(j)每次执行算法第 5 行，调用 MAX-HEAPIFY 后得到的最大堆，并标识当次的 i 值。其中，仅仅浅色阴影的结点被保留在堆中。(k)最终数组 A 的排序结果

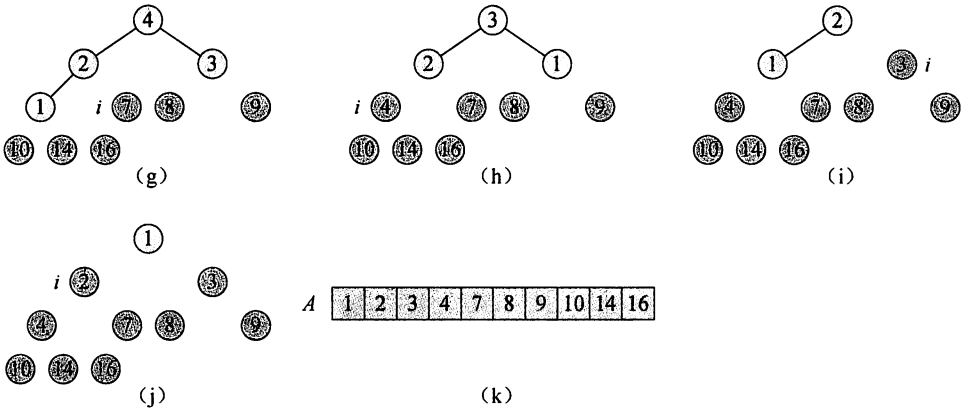

图 6-4 （续）

练习

6.4-1 参照图 6-4 的方法，说明 HEAPSORT 在数组 $A=\langle 5, 13, 2, 25, 7, 17, 20, 8, 4\rangle$ 上的操作过程。

6.4-2 试分析在使用下列循环不变量时，HEAPSORT 的正确性：

在算法的第 2～5 行 **for** 循环每次迭代开始时，子数组 $A[1..i]$ 是一个包含了数组 $A[1..n]$ 中第 i 小元素的最大堆，而子数组 $A[i+1..n]$ 包含了数组 $A[1..n]$ 中已排序的 $n-i$ 个最大元素？

160
~
161

6.4-3 对于一个按升序排列的包含 n 个元素的有序数组 A 来说，HEAPSORT 的时间复杂度是多少？如果 A 是降序呢？

6.4-4 证明：在最坏情况下，HEAPSORT 的时间复杂度是 $\Omega(n\lg n)$。

***6.4-5** 证明：在所有元素都不同的情况下，HEAPSORT 的时间复杂度是 $\Omega(n\lg n)$。

6.5 优先队列

堆排序是一个优秀的算法，但是在实际应用中，第 7 章将要介绍的快速排序的性能一般会优于堆排序。尽管如此，堆这一数据结构仍然有很多应用。在这一节中，我们要介绍堆的一个常见应用：作为高效的优先队列。和堆一样，优先队列也有两种形式：最大优先队列和最小优先队列。这里，我们关注于如何基于最大堆实现最大优先队列。练习 6.5-3 将会要求读者编写最小优先队列过程。

优先队列(priority queue)是一种用来维护由一组元素构成的集合 S 的数据结构，其中的每一个元素都有一个相关的值，称为**关键字**(key)。一个**最大优先队列**支持以下操作：

INSERT(S，x)：把元素 x 插入集合 S 中。这一操作等价于 $S=S\cup\{x\}$。

MAXIMUM(S)：返回 S 中具有最大键字的元素。

EXTRACT-MAX(S)：去掉并返回 S 中的具有最大键字的元素。

INCREASE-KEY(S，x，k)：将元素 x 的关键字值增加到 k，这里假设 k 的值不小于 x 的原关键字值。

最大优先队列的应用有很多，其中一个就是在共享计算机系统的作业调度。最大优先队列记录将要执行的各个作业以及它们之间的相对优先级。当一个作业完成或者被中断后，调度器调用 EXTRACT-MAX 从所有的等待作业中，选出具有最高优先级的作业来执行。在任何时候，调度器可以调用 INSERT 把一个新作业加入到队列中来。

相应地，**最小优先队列**支持的操作包括 INSERT、MINIMUM、EXTRACT-MIN 和 DECREASE-KEY。最小优先队列可以被用于基于事件驱动的模拟器。队列中保存要模拟的事件，每个事件都有一个发生时间作为其**关键字**。事件必须按照发生的时间顺序进行模拟，因为某一事件的模拟结果可能会触发对其他事件的模拟。在每一步，模拟程序调用 EXTRACT-MIN 来选择下一个要模拟的事件。当一个新事件产生时，模拟器通过调用 INSERT 将其插入最小优先级队列中。在第 23 章和第 24 章的内容中，我们将会看到最小优先队列的其他用途，特别是对 DECREASE-KEY 操作的使用。 162

显然，优先队列可以用堆来实现。对一个像作业调度或事件驱动模拟器这样的应用程序来说，优先队列的元素对应着应用程序中的对象。通常，我们需要确定哪个对象对应一个给定的优先队列元素，反之亦然。因此，在用堆来实现优先队列时，需要在堆中的每个元素里存储对应对象的**句柄**(handle)。句柄(如一个指针或一个整型数等)的准确含义依赖于具体的应用程序。同样，在应用程序的对象中，我们也需要存储一个堆中对应元素的句柄。通常，这一句柄是数组的下标。由于在堆的操作过程中，元素会改变其在数组中的位置，因此，在具体的实现中，在重新确定堆元素位置时，我们也需要更新相应应用程序对象中的数组下标。因为对应用程序对象的访问细节强烈依赖于应用程序及其实现方式，所以这里我们不做详细讨论。需要强调的是，这些句柄也需要被正确地维护。

现在，我们来讨论如何实现最大优先队列的操作。过程 HEAP-MAXIMUM 可以在 $\Theta(1)$时间内实现 MAXIMUM 操作。

```
HEAP-MAXIMUM(A)
1  return A[1]
```

过程 HEAP-EXTRACT-MAX 实现 EXTRACT-MAX 操作。它与 HEAPSORT 过程中的 **for** 循环体部分(第 3～5 行)很相似。

```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2        error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size - 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

HEAP-EXTRACT-MAX 的时间复杂度为 $O(\lg n)$。因为除了时间复杂度为 $O(\lg n)$的 MAX-HEAPIFY 以外，它的其他操作都是常数阶的。

HEAP-INCREASE-KEY 能够实现 INCREASE-KEY 操作。在优先队列中，我们希望增加关键字的优先队列元素由对应的数组下标 i 来标识。这一操作需要首先将元素 $A[i]$的关键字更新为新值。因为增大 $A[i]$的关键字可能会违反最大堆的性质，所以上述操作采用了类似于 2.1 节 INSERTION-SORT 中插入循环(算法第 5～7 行)的方式，在从当前结点到根结点的路径上，为新增的关键字寻找恰当的插入位置。在 HEAP-INCREASE-KEY 的操作过程中，当前元素会不断地与其父结点进行比较，如果当前元素的关键字较大，则当前元素与其父结点进行交换。这一过程会不断地重复，直到当前元素的关键字小于其父结点时终止，因为此时已经重新符合了最大堆的性质。(准确的循环不变量表示见练习 6.5-5。) 163

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
```

```
2       error "new key is smaller than current key"
3   A[i] = key
4   while i > 1 and A[PARENT(i)] < A[i]
5       exchange A[i] with A[PARENT(i)]
6       i = PARENT(i)
```

图 6-5 显示了 HEAP-INCREASE-KEY 的一个操作过程。在包含 n 个元素的堆上，HEAP-INCREASE-KEY 的时间复杂度是 $O(\lg n)$。这是因为在算法第 3 行做了关键字更新的结点到根结点的路径长度为 $O(\lg n)$。

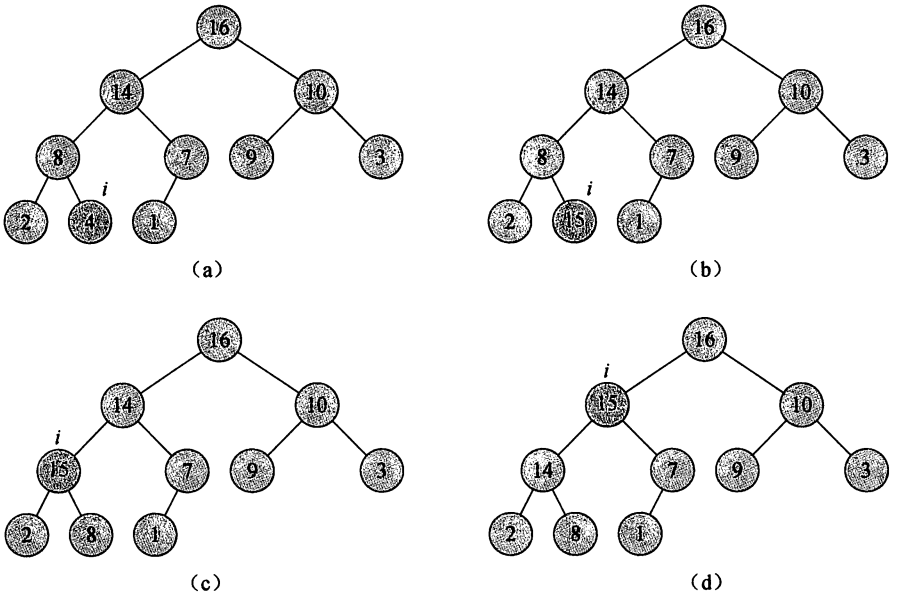

图 6-5 HEAP-INCREASE-KEY 的操作过程。(a)图 6-4(a)中的量大堆，其中下标为 i 的结点以深色阴影显示。(b)该结点的关键字增加到 15。(c)经过第 4～6 行的 **while** 循环的一次迭代，该结点与其父结点交换关键字，同时下标 i 的指示上移到其父结点。(d)经过再一次迭代后得到的最大堆。此时，$A[\text{PARENT}(i)] \geqslant A[i]$。现在，最大堆的性质成立，程序终止

MAX-HEAP-INSERT 能够实现 INSERT 操作。它的输入是要被插入到最大堆 A 中的新元素的关键字。MAX-HEAP-INSERT 首先通过增加一个关键字为 $-\infty$ 的叶结点来扩展最大堆。然后调用 HEAP-INCREASE-KEY 为新结点设置对应的关键字，同时保持最大堆的性质。

```
MAX-HEAP-INSERT(A, key)
1   A.heap-size = A.heap-size + 1
2   A[A.heap-size] = -∞
3   HEAP-INCREASE-KEY(A, A.heap-size, key)
```

在包含 n 个元素的堆上，MAX-HEAP-INSERT 的运行时间为 $O(\lg n)$。

总之，在一个包含 n 个元素的堆中，所有优先队列的操作都可以在 $O(\lg n)$ 时间内完成。

练习

6.5-1 试说明 HEAP-EXTRACT-MAX 在堆 $A=\langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1\rangle$ 上的操作过程。

6.5-2 试说明 MAX-HEAP-INSERT(A, 10)在堆 $A=\langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1\rangle$ 上的操作过程。

6.5-3 要求用最小堆实现最小优先队列，请写出 HEAP-MINIMUM、HEAP-EXTRACT-MIN、HEAP-DECREASE-KEY 和 MIN-HEAP-INSERT 的伪代码。

6.5-4 在 MAX-HEAP-INSERT 的第 2 行，为什么我们要先把关键字设为 $-\infty$，然后又将其增加到所需的值呢？

164 ~ 165

6.5-5 试分析在使用下列循环不变量时，HEAP-INCREASE-KEY 的正确性：

在算法的第 4～6 行 **while** 循环每次迭代开始的时候，子数组 $A[1..A.heap\text{-}size]$ 要满足最大堆的性质。如果有违背，只有一个可能：$A[i]$大于 $A[\text{PARENT}(i)]$。

这里，你可以假定在调用 HEAP-INCREASE-KEY 时，$A[1..A.heap\text{-}size]$是满足最大堆性质的。

6.5-6 在 HEAP-INCREASE-KEY 的第 5 行的交换操作中，一般需要通过三次赋值来完成。想一想如何利用 INSERTION-SORT 内循环部分的思想，只用一次赋值就完成这一交换操作？

6.5-7 试说明如何使用优先队列来实现一个先进先出队列，以及如何使用优先队列来实现栈。（队列和栈的定义见 10.1 节。）

6.5-8 HEAP-DELETE(A, i)操作能够将结点 i 从堆 A 中删除。对于一个包含 n 个元素的堆，请设计一个能够在 $O(\lg n)$时间内完成的 HEAP-DELETE 操作。

6.5-9 请设计一个时间复杂度为 $O(n \lg k)$的算法，它能够将 k 个有序链表合并为一个有序链表，这里 n 是所有输入链表包含的总的元素个数。（提示：使用最小堆来完成 k 路归并。）

思考题

6-1 （用插入的方法建堆） 我们可以通过反复调用 MAX-HEAP-INSERT 实现向一个堆中插入元素，考虑 BUILD-MAX-HEAP 的如下实现方式：

166

```
BUILD-MAX-HEAP′(A)
1  A.heap-size = 1
2  for i = 2 to A.length
3      MAX-HEAP-INSERT(A, A[i])
```

a. 当输入数据相同的时候，BUILD-MAX-HEAP 和 BUILD-MAX-HEAP′生成的堆是否总是一样？如果是，请证明；否则，请举出一个反例。

b. 证明：在最坏情况下，调用 BUILD-MAX-HEAP′建立一个包含 n 个元素的堆的时间复杂度是 $\Theta(n \lg n)$。

6-2 （对 d 叉堆的分析） d 叉堆与二叉堆很类似，但（一个可能的例外是）其中的每个非叶结点有 d 个孩子，而不是仅仅 2 个。

a. 如何在一个数组中表示一个 d 叉堆？

b. 包含 n 个元素的 d 叉堆的高度是多少？请用 n 和 d 表示。

c. 请给出 EXTRACT-MAX 在 d 叉最大堆上的一个有效实现，并用 d 和 n 表示出它的时间复杂度。

d. 给出 INSERT 在 d 叉最大堆上的一个有效实现，并用 d 和 n 表示出它的时间复杂度。

e. 给出 INCREASE-KEY(A, i, k)的一个有效实现。当 $k < A[i]$时，它会触发一个错误，否则执行 $A[i] = k$，并更新相应的 d 叉最大堆。请用 d 和 n 表示出它的时间复杂度。

6-3 （Young 氏矩阵） 在一个 $m \times n$ 的 **Young 氏矩阵**（Young tableau）中，每一行的数据都是从左到右排序的，每一列的数据都是从上到下排序的。Young 氏矩阵中也会存在一些值为∞的数据项，表示那些不存在的元素。因此，Young 氏矩阵可以用来存储 $r \leqslant mn$ 个有限的数。

a. 画出一个包含元素为{9，16，3，2，4，8，5，14，12}的4×4Young氏矩阵。

b. 对于一个 $m\times n$ 的 Young 氏矩阵 Y 来说，请证明：如果 $Y[1,1]=\infty$，则 Y 为空；如果 $Y[m,n]<\infty$，则 Y 为满(即包含 mn 个元素)。

c. 请给出一个在 $m\times n$ Young 氏矩阵上时间复杂度为 $O(m+n)$ 的 EXTRACT-MIN 的算法实现。你的算法可以考虑使用一个递归过程，它可以把一个规模为 $m\times n$ 的问题分解为规模为 $(m-1)\times n$ 或者 $m\times(n-1)$ 的子问题(提示：考虑使用 MAX-HEAPIFY)。这里，定义 $T(p)$ 用来表示 EXTRACT-MIN 在任一 $m\times n$ 的 Young 氏矩阵上的时间复杂度，其中 $p=m+n$。给出并求解 $T(p)$ 的递归表达式，其结果为 $O(m+n)$。

d. 试说明如何在 $O(m+n)$ 时间内，将一个新元素插入到一个未满的 $m\times n$ 的 Young 氏矩阵中。

e. 在不用其他排序算法的情况下，试说明如何利用一个 $n\times n$ 的 Young 氏矩阵在 $O(n^3)$ 时间内将 n^2 个数进行排序。

f. 设计一个时间复杂度为 $O(m+n)$ 的算法，它可以用来判断一个给定的数是否存储在 $m\times n$ 的 Young 氏矩阵中。

本章注记

堆排序算法是由 Williams[357]发明的，他同时描述了如何利用堆来实现一个优先队列。BUILD-MAX-HEAP 则是由 Floyd[106]提出的。

在第 16、23 和 24 章中，我们会使用最小堆实现最小优先队列。在第 19 章中，我们会给出一个针对特定操作改进了时间界的算法实现。在第 20 章中，我们还给出了一个针对关键字来自有限非负整数集合的实现。

如果数据都是 b 位整型数，而且计算机内存也是可寻址的 b 位字所组成的，Fredman 和 Willard[115]给出了如何在 $O(1)$ 时间内实现 MINIMUM 和在 $O(\sqrt{\lg n})$ 时间内实现 INSERT、EXTRACT-MIN 操作的算法。Thorup[337]将时间复杂度的界降低到 $O(\lg\lg n)$。这一性能的提升是以使用了额外的存储空间为代价的，这可以用随机散列方法在线性空间中实现。

优先队列的一个重要的特殊情形是 EXTRACT-MIN 操作序列为**单调的**，即连续 EXTRACT-MIN 操作返回的值随着时间单调递增。这一情况会在一些重要的应用中出现，例如，第 24 章中将会介绍的 Dijkstra 单源最短路径算法和离散事件模拟等。在 Dijkstra 算法中，DECREASE-KEY 的实现效率非常重要。对于单调情形，如果数据是 1，2，…，C 范围内的整数，Ahuja、Melhorn、Orlin 和 Tarjan[8]利用称为基数堆(radix heap)的数据结构，实现了 $O(\lg C)$ 摊还时间内的 EXTRACT-MIN 和 INSERT 操作(摊还分析的内容请参见第 17 章)，以及 $O(1)$ 时间内的 DECREASE-KEY。通过同时使用斐波那契堆(见第 19 章)和基数堆，这一时间界可以从 $O(\lg C)$ 降低到 $O(\sqrt{\lg C})$。通过将 Denardo 和 Fox[85]提出的多层桶结构与前文中提到的 Thorup 设计的堆相结合，Cherkassky、Goldberg 和 Silverstein[65]进一步把这一时间界降低到 $O(\lg^{1/3+\varepsilon}C)$。Raman[291]进一步改进这些结果，将其降低到 $O(\min(\lg^{1/4+\varepsilon}C,\lg^{1/3+\varepsilon}n))$，对任意固定值 $\varepsilon>0$ 都成立。