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
size()	报告优先级队列的规模，即其中词条的总数
insert()	将指定词条插入优先级队列
getMax()	返回优先级最大的词条（若优先级队列非空）
delMax()	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。

```
/*****************************************************************************************
 * 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
 *****************************************************************************************
 * 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
 * 为Huffman_PQ_List、Huffman_PQ_ComplHeap和Huffman_PQ_LeftHeap共用
 * 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
 *****************************************************************************************/
HuffTree* generateTree ( HuffForest* forest ) {
   while ( 1 < forest->size() ) {
      HuffTree* s1 = forest->delMax(); HuffTree* s2 = forest->delMax();
      HuffTree* s = new HuffTree();
      s->insertAsRoot ( HuffChar ( '^', s1->root()->data.weight + s2->root()->data.weight ) );
      s->attachAsLC ( s->root(), s1 ); s->attachAsRC ( s->root(), s2 );
      forest->insert ( s ); //将合并后的Huffman树插回Huffman森林
   }
   HuffTree* tree = forest->delMax(); //至此，森林中的最后一棵树
   return tree; //即全局Huffman编码树
}
```

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$\mathcal{O}(\log n)$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$\mathcal{O}(\log n)$？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。

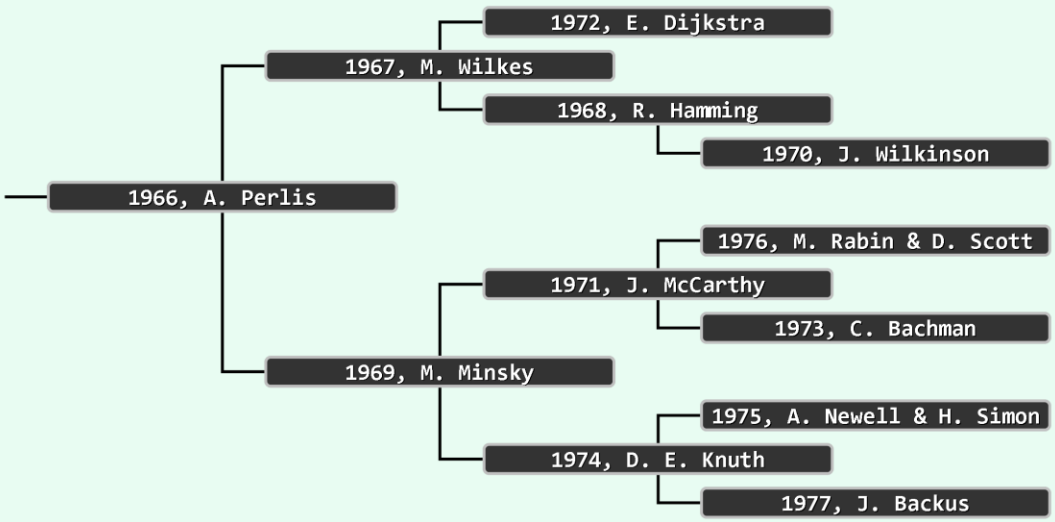

图10.1 以获奖先后为优先级，由前12届图灵奖得主构成的完全二叉堆

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。

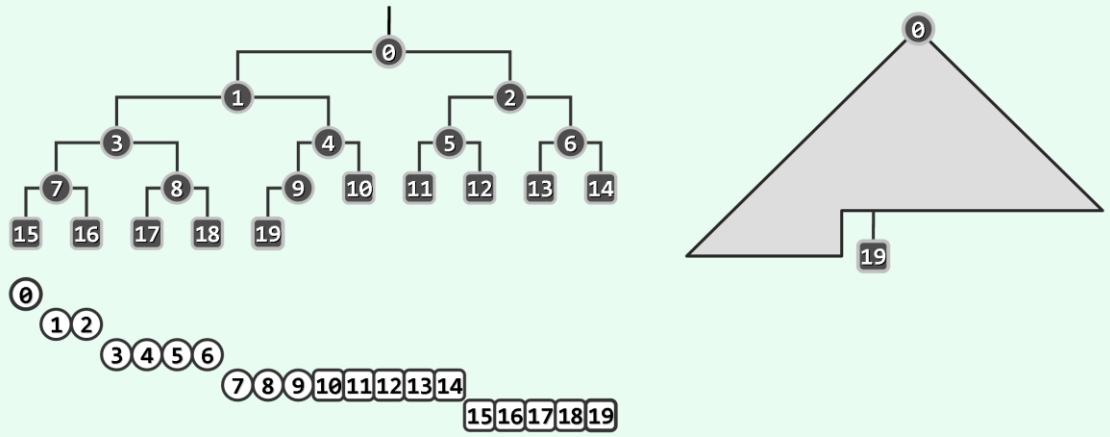

图10.2 按照层次遍历序列，对完全二叉树节点做编号（其中圆形表示内部节点，方形表示外部节点）

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·i(v) + 1;
2）若v有右孩子，则i(rchild(v)) = 2·i(v) + 2;
3）若v有父节点，则i(parent(v)) = ⌊(i(v) - 1)/2⌋ = ⌈(i(v)/2⌉ - 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。

```
#define  InHeap(n, i)      ( ( ( -1 ) < ( i ) ) && ( ( i ) < ( n ) ) ) //判断PQ[i]是否合法
#define  Parent(i)         ( ( i - 1 ) >> 1 ) //PQ[i]的父节点（floor((i-1)/2)，i无论正负）
#define  LastInternal(n)   Parent( n - 1 ) //最后一个内部节点（即末节点的父亲）
#define  LChild(i)         ( 1 + ( ( i ) << 1 ) ) //PQ[i]的左孩子
#define  RChild(i)         ( ( 1 + ( i ) ) << 1 ) //PQ[i]的右孩子
#define  ParentValid(i)    ( 0 < i ) //判断PQ[i]是否有父亲
#define  LChildValid(n, i) InHeap( n, LChild( i ) ) //判断PQ[i]是否有一个（左）孩子
#define  RChildValid(n, i) InHeap( n, RChild( i ) ) //判断PQ[i]是否有两个孩子
#define  Bigger(PQ, i, j)  ( lt( PQ[i], PQ[j] ) ? j : i ) //取大者（等时前者优先）
#define  ProperParent(PQ, n, i) /*父子（至多）三者中的大者*/ \
            ( RChildValid(n, i) ? Bigger( PQ, Bigger( PQ, i, LChild(i) ), RChild(i) ) : \
            ( LChildValid(n, i) ? Bigger( PQ, i, LChild(i) ) : i \
            ) \
            ) //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_ComplHeap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。

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

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。

```
template <typename T> T PQ_ComplHeap<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_ComplHeap<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。

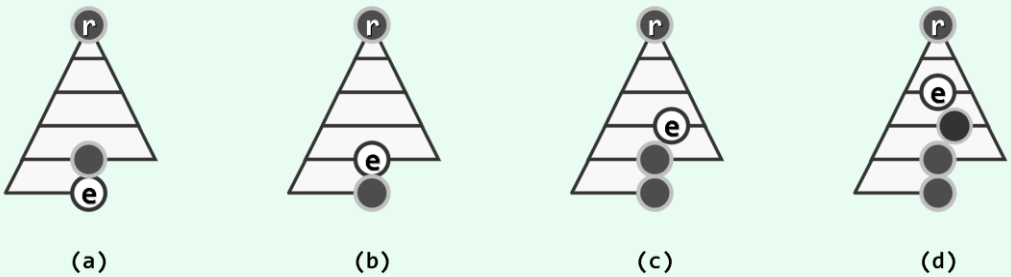

图10.3 完全二叉堆词条插入过程

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) = ⌊(i(e) - 1)/2⌋。

此时，若经比较判定e ≤ 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⌊$\log_2 n$⌋。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。

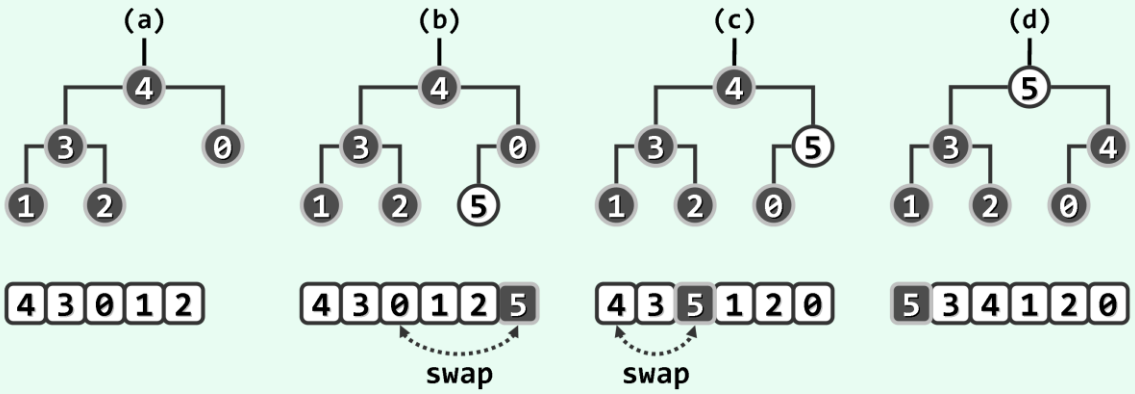

图10.4 完全二叉堆词条插入操作实例

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_ComplHeap<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_ComplHeap<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。

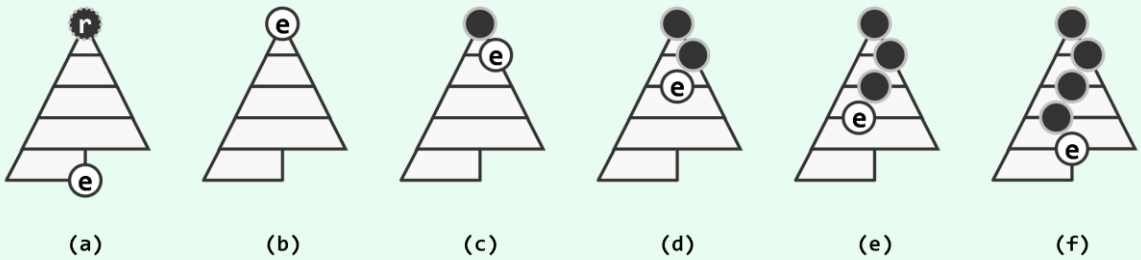

图10.5 完全二叉堆词条删除过程

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$O(\log n)$——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。

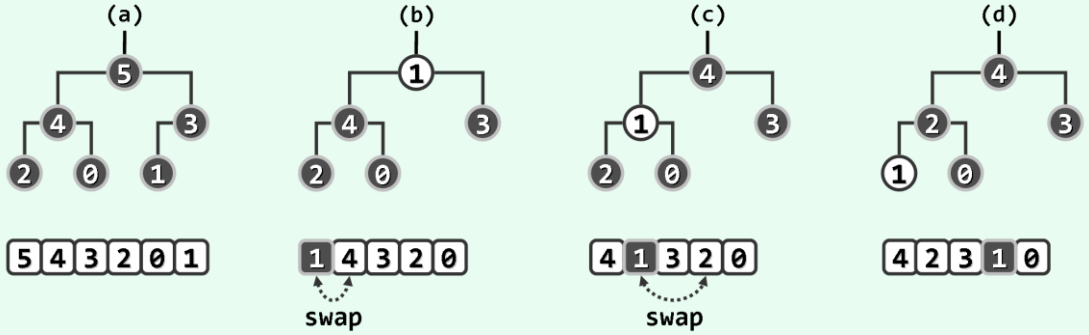

图10.6 完全二叉堆词条删除操作实例

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_ComplHeap<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：

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

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

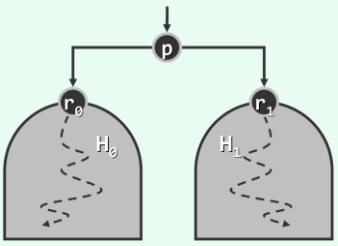

图10.7 堆合并算法原理

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[①]。

[①] 由R. W. Floyd于1964年发明[57]

■ 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_ComplHeap<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。

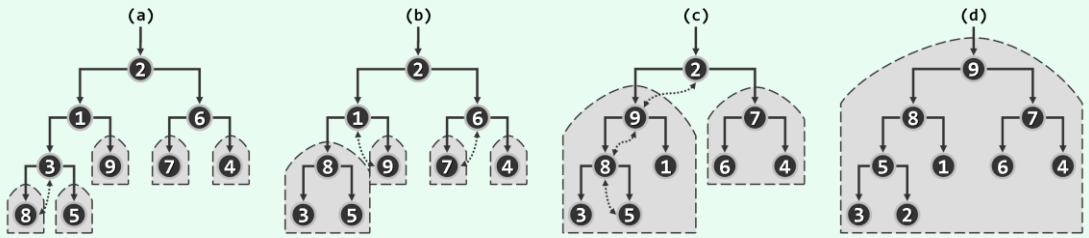

图10.8 Floyd算法实例（虚线示意下滤过程中的交换操作）

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高度，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：

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

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$\mathcal{O}(n\log n)$，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$\mathcal{O}(1)$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。

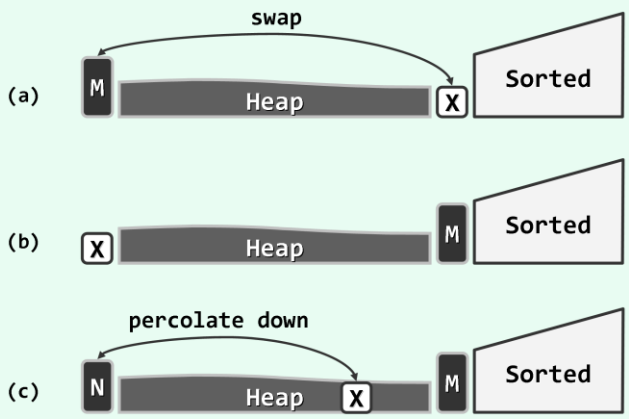

图10.9 就地堆排序

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$\mathcal{O}(\log n)$时间。因此，全部n步迭代累计耗时不超过$\mathcal{O}(n\log n)$。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$\mathcal{O}(n\log n)$。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$\mathcal{O}(n\log n)$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。

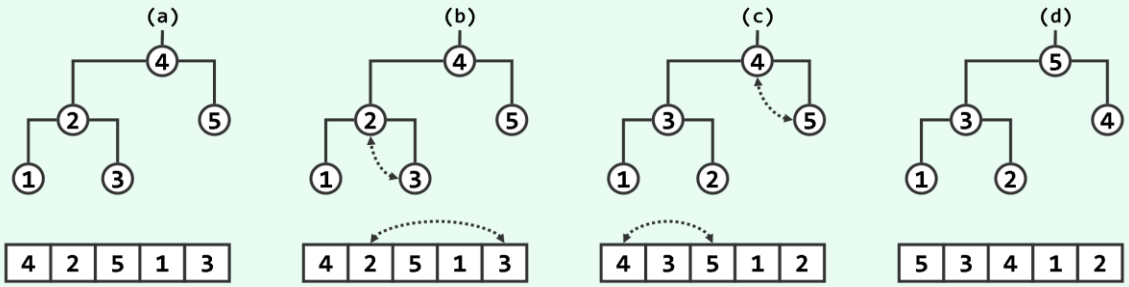

图10.10 就地堆排序实例：建堆

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。

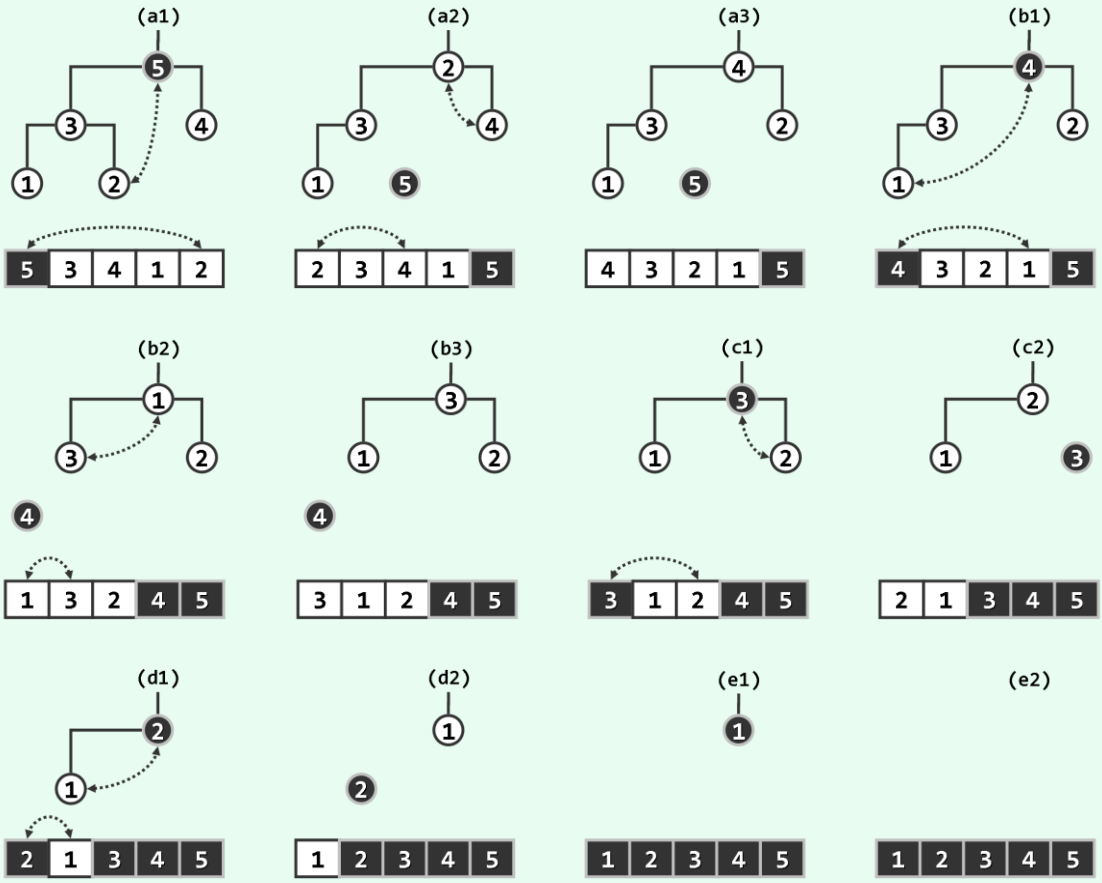

图10.11 就地堆排序实例：迭代

■ 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_ComplHeap<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。

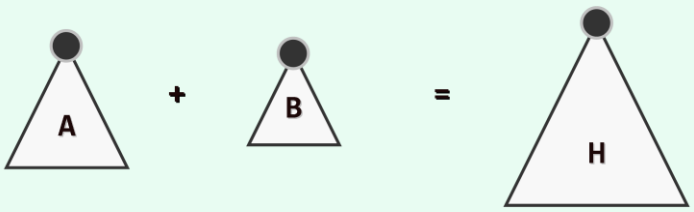

图10.12 堆合并

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 ≥ 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[②]（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。

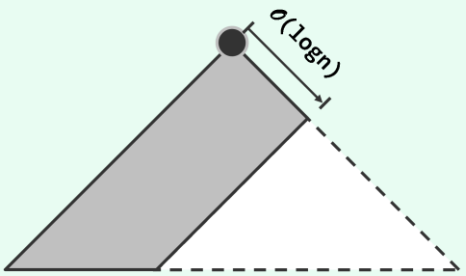

图10.13 整体结构向左倾斜，右侧通路上的节点不超过$\mathcal{O}(\log n)$个

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]）。

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

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。

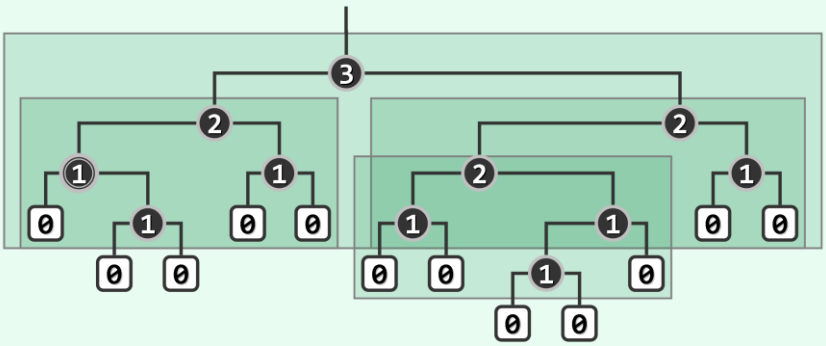

图10.14 空节点路径长度（其中有个节点违反左倾性，以双圈标出）

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。

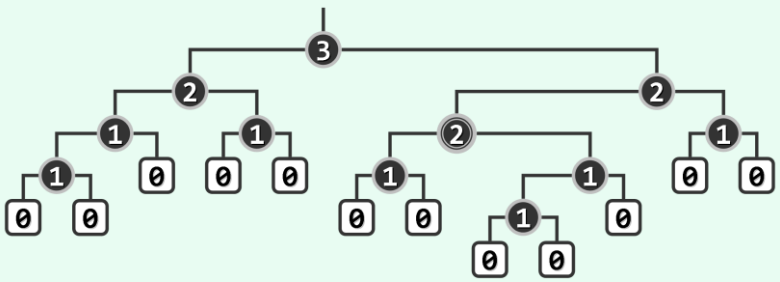

图10.15 左式堆：左孩子的npl值不小于右孩子，而前者的高度却可能小于后者

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。

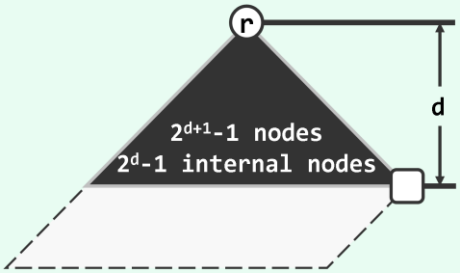

图10.16 左式堆的最右侧通路

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

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

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 ≥ b。

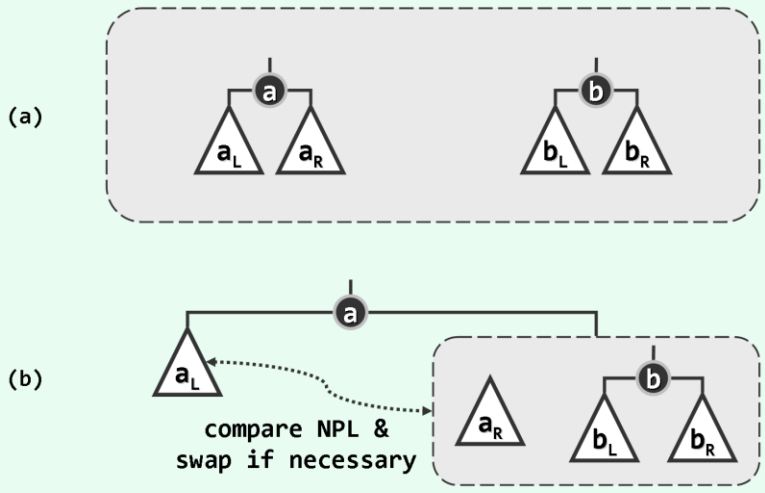

图10.17 左式堆合并算法原理

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。

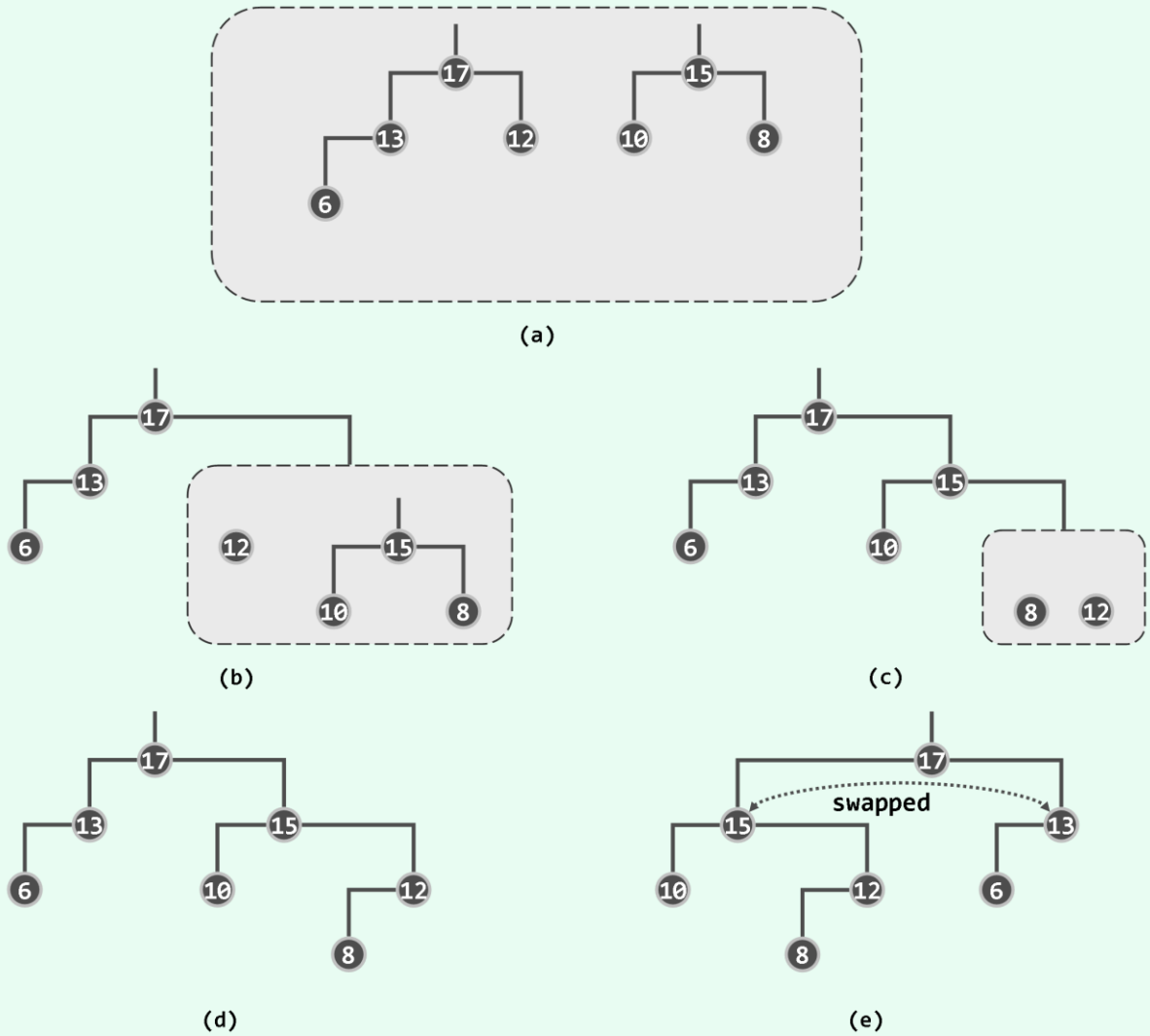

图10.18 左式堆合并算法实例

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npl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npl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->lc || a->lc->npl < a->rc->npl ) //若有必要
9        swap ( a->lc, 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：

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

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()

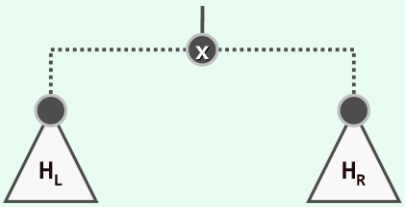

图10.19 基于堆合并操作实现删除接口

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H_L和H_R。

在摘除x之后，H_L和H_R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$O(\log n)$。

■ insert()

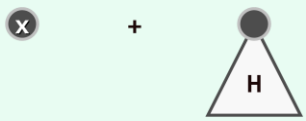

图10.20 基于堆合并操作实现词条插入算法

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$O(\log n)$。