arXiv:2603.01206v1 [cs.DS] 1 Mar 2026

基于划分的简单堆

Gerth Stølting Brodal* John Iacono†
CasperMoldrup Rysgaard Sebastian Wild‡

2026年3月3日

摘要

我们提出一类新型优先队列数据结构：基于划分的简单堆。这些结构由 $\mathcal{O}(\lg n)$ 个双向链表组成；不同链表中的数据之间保持有序，但每个链表内部无序。我们的结构支持摊还 $\mathcal{O}(\lg n)$ 时间的取最小值操作，以及摊还 $\mathcal{O}(\lg \lg\ n)$ 时间的插入和减小键值操作。

这些结构只需对 $\mathcal{O}(\lg n)$ 个元素进行二分查找，并以自然的方式对链表进行二分划分和拼接，以应对链表过大或过小的情况。我们介绍了维护这些链表的三种不同方法，以实现所述摊还运行时间。

1. 引言

优先队列是一种基础的、基于比较的抽象数据类型，广泛应用于许多高效算法中，例如离散事件仿真、Dijkstra最短路径算法和Prim最小生成树算法。优先队列（又称堆）存储一组元素，并须实现以下操作。我们假设每个元素都关联一个来自基于比较的全集的键值：

- 插入(e)：向堆中添加一个元素 e ，并返回指向该元素的指针ptr 。
- 删除最小值()：移除并返回堆中键值最小的元素。
- 减小键值(ptr, 键值)：将堆中ptr所指元素的键值改为更小的键值。

标准的本科计算机科学课程会讲授二叉堆 [Wil64]，它能够在 $\mathcal{O}(\lg n)$ 时间内执行所有操作[1] ，前提是该数据结构当前包含 n 个元素（平衡二叉搜索树也能做到这一点）。然而，优先队列的更新操作可以快得多：效率最高的堆，例如（严格）斐波那契堆 [FT87, BLT25]，执行插入和减小键值的摊还时间为 常数 ，执行删除最小值的时间为 $\mathcal{O}(\lg n)$。

*丹麦奥胡斯大学, {gerth,rysgaard}@cs.au.dk
†布鲁塞尔自由大学，埃尔塞讷，比利时， john.iacono@ulb.be
‡马尔堡大学，德国， wild@informatik.uni-marburg.de；利物浦大学，英国[1]在本文中，我们始终用 lg 表示二进制对数。

尽管理论上最优，这些高效的优先队列 也有一些缺点。首先，它们的结构和分析不太直观，教学和实现起来较为困难。其次，它们的运行时间和空间占用的常数因子都远高于二叉堆；这主要是因为它们本质上依赖于树的基于指针的表示，导致引用局部性较差。

为了简化高效优先队列并提高其实际效率，人们设计了许多替代数据结构，我们将在第 1.1 节中简要回顾。它们都保留了较为复杂的结构和不变式，难以分析，或者更新需要 $\Omega(\lg n)$ 时间。

本文介绍仅基于线性数据结构的简单优先队列实现，并证明，只要采用精心设计的“枢轴遗忘规则”，将集合数量控制在$\mathcal{O}(\lg n)$，就足以通过简单的快速排序式划分，将元素分到无序集合中，从而使插入和减小键值操作的摊还时间达到$\mathcal{O}(\lg \lg\ n)$，删除最小值操作的摊还时间达到$\mathcal{O}(\lg n)$。我们介绍了几种这样的规则，展现了丰富的设计空间。这样一来，我们就更接近前述高效优先队列，同时采用的却是完全基础的技术和简单的设计。以其中或许最简单的规则（“惰性划分堆”）为例，我们在第 2 节和第 3 节给出完整且自成一体的描述，包括适合本科算法课程教学的摊还分析。

1.1. 相关工作

完整的优先队列综述请参见 [Bro13] 。我们从斐波那契堆 [FT87]谈起优先队列中快速减小键值的发展史。斐波那契堆支持常数摊还时间的减小键值和插入操作，以及摊还时间为 $\mathcal{O}(\lg n)$ 的删除最小值操作。此后，人们提出了其他多种支持快速减小键值的堆：

- 配对堆 [FSST86], 是一种仿照伸展树构造的自调整结构，其减小键值操作的渐近摊还运行时间尚不明确。目前已知的结果只有 $\mathcal{O}\left(2^{\sqrt{\lg \lg n}}\right)$ [Pet05] 和 $\Omega(\lg \lg\ n)$ [Fre99]。已有研究证明，多遍变体的减小键值操作具有 $\mathcal{O}(\lg \lg\ n \cdot \lg \lg \lg\ n)$ 的摊还时间 [ST23a]。
- 严格斐波那契堆，其操作具有最坏情况运行时间界 [BLT25]。
- Chan 的震荡堆是为作为斐波那契堆的更简单替代方案而发明的，其减小键值的摊还时间为 $\mathcal{O}(1)$ [Cha13]。
- 秩配对堆 [HST09] 是另一种替代方案，其减小键值的摊还时间为$\mathcal{O}(1)$，但其结构更接近配对堆。
- 违例堆 [Elm10], 、空心堆 [HKTZ17], 以及重新审视的斐波那契堆 [KTZ14]是其他变体，其减小键值的摊还时间为$\mathcal{O}(1)$。
- 瘦型平滑堆 [ST23b] 的减小键值时间为$\mathcal{O}(\lg \lg\ n)$，但它们采用入度为常数的指针模型。

1.1.1. 下界

关于堆中减小键值操作，有两个下界：一个由 Fredman 提出 [Fre99]，另一个由 Iacono 和 Özkan 提出 [IÖ14]。这两个下界都只适用于遵循两种不同特定模型的基于堆的优先队列；这两种模型都包含配对堆，但都不适用于我们的新结构。Fredman 的下界着重于增强数据与减小键值操作时间之间的权衡

。Iacono 和 Özkan 的下界着重于基于指针的结构。这两项结果都表明，$\mathcal{O}(\lg \lg\ n)$ 可以说是堆的一种自然运行时间，但它们都只适用于（不同的）广义堆类别；这些类别相当特殊，只有在针对其对手的论证中才适用。

上述所有内容都表明，对于具有常数入度的指针模型数据结构而言，减小键值的正确运行时间应为$\mathcal{O}(\lg \lg\ n)$。瘦型平滑堆已经实现了这一点，而配对堆被推测也能实现。所有其他堆要么需要 RAM 模型技巧，要么具有非常数入度。例如，斐波那契堆需要对一个大小为$\mathcal{O}(\lg n)$的数组进行随机访问，以便合并秩相同的堆（秩是一个取值范围为对数级的整数）。当然，也可以用平衡二叉搜索树替代这个数组，但这会引入$\mathcal{O}(\log \log\ n)$的额外开销。在 [FT87], 中展示了如何移除这个数组，不过需要使用具有线性入度的节点。

1.1.2. 缓存无关的启发

我们以 Chowdhury 和 Ramachandran [CR18] 以及 Brodal、Fagerberg、Meyer 和 Zeh [BFMZ04] 的工作为灵感来源，因为它们都没有树论色彩。这两项工作都是为缓存无关模型设计的，使用了两个大小递增的数组序列；从表面上看，这类似于我们数据结构的两个副本：元素被插入其中一组，向下渗透，在某个时刻移动到另一组，然后向上渗透。由于是为缓存无关模型设计的，这些列表存储在数组中。

1.1.3. 惰性搜索树

与此相关的还有惰性搜索树 [SW20、SZ22、RW25],，这是一种能够在优先队列和二叉搜索树之间平滑过渡的数据结构；具体取决于使用哪些查询，插入的速度可以媲美最高效的优先队列，同时也支持任意有序字典查询。我们的结构可以视为原始惰性搜索树较低两层的单侧实例，其操作经过了少许修改。

1.1.4. Quickheaps

其他相关研究则更侧重于应用，着眼于提升优先队列实现的实际运行速度。渐近性能固然重要，但对于所有合理规模的输入，常数因子开销和引用局部性带来的影响可能远远超过对数与次对数复杂度之间的差异。Quickheaps [PN06、NP10] 及其“更强”的改进版本 [NPPS11] 将数据存储在单个数组中，并在概念上针对查询连续执行快速选择，同时记住并跳过此前的划分步骤。

如果更新以随机顺序到来，这种方法效果很好；为了增强对对抗性输入的稳健性，更强的 quickheap 借鉴了平衡二叉搜索树的方案，以确保 quickheap 具有良好的摊还性能：要么采用随机化，要么采用某种重量平衡二叉搜索树。由于其设计初衷是解决难得多的任务——维持整棵二叉搜索树的平衡——因此由此得到的 quickheap 拼接规则十分复杂。我们指出，不仅连接规则本身， [NPPS11] 中的摊还分析也与本文中的分析有很大不同。他们还将 quickheap 在外存模型中的良好性能（无论采用哪种变体）列为一项优点。我们的堆也能具备这些相同的优良特性。请注意，quickheap 的作者在考察 quickheap 的外存应用时，排除了基于指针的操作，尤其是减小键值操作。

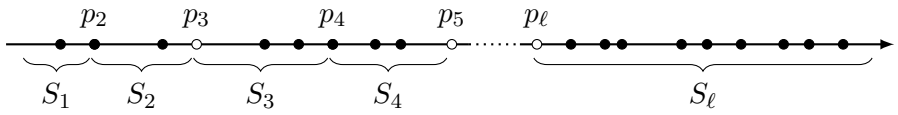


图 1：我们的基于分区的堆框架示意图。枢轴 $p_2 \ldots,\ p_\ell$ 将堆中的 n个元素划分为集合 ℓ 集合 $S_1,\ S_2,\ \ldots\ldots,\ S_\ell$。黑点是堆中的 n 个元素。枢轴可以是堆中的元素（黑色），也可以是先前从堆中移除的元素（白色）。

2. 基于分区的堆框架

这里我们介绍第3–5节中三种数据结构共有的基于分区的堆通用框架。我们假设元素的键值取自一个全序全集，并且元素按照其键值排序。如果向基于分区的堆中插入了重复键，便会采用标准技术，以确保所有并列情况都得到一致处理。

基于分区的堆将当前存储在堆中的元素维护为由$sets\ S_1,\ S_2,\ \ldots S_\ell$构成的一个分区，其中某个 ℓ 为$\mathcal{O}(\lg\ n)$。当前存储在堆中的元素数量为 $n = |S_1| + \cdots + |S_\ell|$。虽然这些集合本身是无序的，但如果 $i < j$，则S_i 中的所有元素都小于 S_j的所有元素

每个集合 S_i 都以双向链表 存储，因为集合的常数时间拼接对我们所有结构都至关重要。对于每个集合，大小 $|S_i|$ 会被显式维护。由于我们将所有元素存储在链表中，我们支持指向任何已插入元素的稳定指针 。

除集合及其大小外，基于分区的堆还存储枢轴 $p_2 \le \cdots \le p_\ell$，这些枢轴是满足 $S_i \subseteq [p_i, p_{i+1})$（对于 $i = 1,\ \ldots\ldots,\ \ell$）的元素；为方便记号起见，假设$p_1$ 和 $p_{\ell+1}$ 分别是键值为 $-\infty$ 和 $+\infty$的枢轴。枢轴可以是当前位于堆中的元素，也可以是曾经位于堆中的元素。图 1 展示了一个示例。

为了实现每个优先队列操作，我们的所有基于分区的堆都有以下共同步骤：

- 插入(e)：在$\mathcal{O}(\lg n)$个枢轴中搜索 e，用$\mathcal{O}(\lg\lg\ n)$时间找到满足 $p_i \le e < p_{i+1}$的集合 S_i。将 e 插入 S_i，并将大小 $|S_i|$加一。返回指针ptr，指向链表 S_i中包含 e 的新节点。

- 删除最小值()：从第一个非空集合 S_i中移除最小元素，并将大小 $|S_i|$减一。

- 减小键值(ptr，键值)：令 e 表示链表节点 ptr 所指向的元素。从其所在的链表中移除此节点。对枢轴进行二分查找，耗时 $\mathcal{O}(\lg\lg\ n)$，以找到移除此节点的集合 S_i，并将其大小 $|S_i|$减一。将 e 的键值减小为 键值，然后按照插入操作重新插入该节点：再次在枢轴中查找，以确定应将其插入哪个集合，将其插入该集合链表的末尾，并将集合大小加一。

这些描述不足以维持关键的不变式 $\ell = \mathcal{O}(\log n)$。第3–5节中的三种堆正是在这一点上有所不同；每种堆都有各自的不变式来实现这一目标，并且都在上述基本实现的基础上增加了重构操作，以维持这些不变式。所有结构都使用这样的技术：通过连接链表，在常数时间内合并集合；并使用线性时间选择算法，在线性时间内拆分集合

以维持各自的不变式。此外，所有结构都使用摊还分析来论证其运行时间。

从非常宏观的层面来看，各种结构的设计理念如下：在第 3 节的惰性分区堆（LP 堆）中，集合的大小可以任意增大，且集合数量没有下界。该结构会维持一个集合大小相对于此前所有集合大小之和的界。在第 4 节的斐波那契堆：新一代 （FH:TNG）以及第 5 节的堆中，集合可以为空，且其大小有严格的上界。在 FH:TNG 中，非空集合的大小有下界；上界和下界均由斐波那契数给出，这天然适用于拼接和拆分，同时能够维持大小不变式。第 5 节的堆不维持集合大小的下界，因此重构算法更简单。

3. 惰性划分堆

为使集合数量始终保持在 ℓ $\mathcal{O}(\lg n)$ 以内，惰性分区堆 （LP 堆）采用以下枢轴遗忘规则，该规则改编自惰性搜索树 [SW20]:

> 如果枢轴 p 所分隔的两个集合中的元素总数，小于堆中更小 元素的数量，就应舍弃该枢轴（图*2*）。

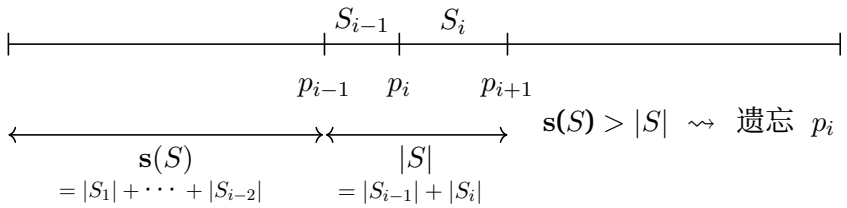


图2： LP 堆中集合的枢轴遗忘规则。此处，集合 S_{i-1} 和 S_i对应的列表将被拼接（从而遗忘 p_i），因为小于 p_{i-1} 的元素比介于 p_{i-1} 和 p_{i+1}之间的元素更多。

在 LP 堆中，每次删除最小值操作都会围绕其中位数对 S_1 进行分区。由于无法保证 $|S_1|$ 有严格的上界，我们需要将删除最小值操作偶尔产生的高成本进行摊还。为此，当集合 S_j、$j = 1$、 $\ldots$, ℓ增长得“过大”时，它们会累积势能，即$\varphi_j := \max\{0, |S_j| - \mathbf{s}(S_j)\}$，其中 $\mathbf{s}(S_j) := |S_1| + \cdots + |S_{j-1}|$ 表示小于 S_j 中所有元素的元素数量（即它们与最小值之间的“安全缓冲区”）。

> 这尤其包括集合 S_1，其势能 $\varphi_1 = |S_1|$ 因此可以支付下一次删除最小值时对 S_1 的扫描与划分。

除这些集合之外，我们还存储一个（静态的）有序数组 T，按此顺序保存当前集合 $S_1, S_2, \ldots, S_\ell$。（每当集合的汇集因一次删除最小值而改变时，我们都负担得起从头重建 T。）为简化伪代码，我们让每个集合 S_i将相应的枢轴 p_i 存储为 $S_i.p$。我们指出，$\mathbf{s}(S_j)$仅在枢轴遗忘阶段需要，并在该阶段按需计算。

3.1. 操作

LP 堆上各操作的伪代码如图 3 所示。在 LP 堆的整个生命周期中，我们维护如下不变式。

不变量（S）： 集合的数量至多为 $\ell \leq 2\lg\, n + 1$。

如果不变式有被违反的风险——这仅可能在删除最小值时发生——我们应用以下连接规则（参见图2）。回想一下，$\mathbf{s}(S_j) := |S_1| + \cdots + |S_{j-1}|$ 是较小元素的数量（即小于 p_j 的元素）：

连接规则 (C): 如果 $|S_j| + |S_{j+1}| < \mathbf{s}(S_j)$，则拼接 S_j 和 S_{j+1}。

这里的连接仅表示将两个集合的元素链表拼接起来，并“遗忘”原先分隔这两个集合的枢轴。此操作耗费常数时间。遗忘枢轴计算$\mathbf{s}(S_j)$，并依次将连接规则 (C) 应用于$\mathcal{S}$。显然，如果（初始）集合数量为 ℓ，遗忘枢轴将耗费$\mathcal{O}(\ell)$时间。注意，遗忘枢轴还会移除所有现有的空集，从而确保对于所有 $1 \le j \le \ell$，$|S_j| \ge 1$。遗忘枢轴已将所有满足 (C) 的集合对连接起来之后，我们可以根据以下引理得到集合数量的一个界。

引理3.1（（C）蕴含（S））： 假设应用连接规则 *(C)* 后，我们得到集合 S_1、.....、S_ℓ，且对于所有 $1 \le j \le \ell$，$|S_j| \ge 1$。那么，对于 n，$\ell \le 2\lg\, n + 1$，其中◁为这些集合的总大小。

证明：我们首先证明集合的前缀呈指数增长：对于所有 $1 \le j \le \ell$，都有$\Sigma(j) := |S_1| + \cdots + |S_j| \ge 2^{(j-1)/2}$ $(*)$。

证明对 j 使用归纳法。由于 $|S_i| \ge 1$ for $1 \le i \le \ell$，所以当 $1 \le j \le 2$ 时命题成立。在 $j \ge 3$ 的归纳步骤中，归纳假设意味着 $\Sigma(j-2) \ge 2^{(j-3)/2}$。根据连接规则 (C)，我们有 $|S_{j-1}| + |S_j| \ge \mathbf{s}(S_{j-1})$；根据定义，$\mathbf{s}(S_{j-1}) = \Sigma(j-2)$。因此 $\Sigma(j) = \Sigma(j-2) + |S_{j-1}| + |S_j| \ge 2 \cdot \Sigma(j-2) \ge 2 \cdot 2^{(j-3)/2} = 2^{(j-1)/2}$，从而证明了 $(*)$。

由于（$*$）对集合的任意前缀都成立，因此特别适用于 $j = \ell$。于是，$\Sigma(\ell) = n$（所有元素），从而得到 $\ell \le 2\lg\, n + 1$，如所声称。□

由引理3.1可知，遗忘枢轴满足(S)。至此，数据结构的描述完毕。

3.2. 势能

为了分析各项操作的摊还成本，我们使用以下势能：

$$\Phi := \sum_{j=1}^{\ell} \varphi_j \qquad \text{with} \qquad \varphi_j := \beta \cdot \max\{0, |S_j| - \mathbf{s}(S_j)\} .$$

（求和遍历数据结构中当前的所有集合。）这里 $\beta \ge 2$ 是一个固定的缩放参数（根据划分的线性成本中的常数因子选取）。

3.3. 插入的摊还成本

执行插入操作时，使用二分查找在 T 中定位包含 e 的集合 S_j，耗时$\mathcal{O}(\lg \lg\, n)$。将 e 添加到 S_j 的链表中需要常数时间。

在 Φ 中，$\mathbf{s}(S_{j'})$ 对所有 $j' > j$ 都增加 1；这绝不会增加 Φ 的值。S_j 的大小增加 1，使 Φ 至多增加 $\beta = \mathcal{O}(1)$。因此，插入的代价为 $\mathcal{O}(\lg \lg\, n)$（摊还和最坏情况均如此）。

LP 堆插入(e)
1 ptr := 新建节点(e) // 获取指向元素的指针2 S : = T.查找(e.键值) // 对枢轴进行二分查找3 S.追加(ptr) // 插入元素4 **返回**ptr

LP 堆减键(ptr, 键值)
1 e : = ∗指针 // 通过指针获取元素2 S : = T.查找(e.键值)3 S.删除(ptr) // 从当前集合中删除元素4 e.键值 : = 键值 // 更新键值5 S : = T.查找(e.键值)6 S.追加(ptr) // 将元素插入新集合

LP 堆删除最小值()
1 最小值 := S_1.删除最小值() // 扫描 $S_1$2 p := 中位数(S_1) // （较大）中位数（确定性线性时间选择 [BFP+73]）3 S_1, S_2 := 分区(S_1, p)4 $S_2.p$:= p5 $\mathcal{S}$: = [$S_1, S_2, S_2, \ldots, S_\ell$] // 由集合构成的双向链表6 遗忘枢轴($\mathcal{S}$) // 修改$\mathcal{S}$7 重建 T 自$\mathcal{S}$8 **返回**最小值

遗忘枢轴($\mathcal{S}$)
// 假设 S = $[S_1, \ldots, S_\ell]$1 S.移除所有空集合()2 A := $S_1$3 s(A) := 0 4 当 S.有下一个(A)5 B :=S.下一个(A) 6 如果 $|A| + |B| <$ s(A) 7 拼接 B 到 A8 S. 移除 (B) 9 否则 10 s(B) := s(A) + $|A|$
11 A := B12 结束如果 13 结束循环

图 3： LP 堆的实现。

3.4. 减小键值的摊还成本

执行减小键值(ptr, 键值)时，指针 ptr 指向的元素 e 的键值会减小为 键值。重新插入 e 的代价为 $\mathcal{O}(\lg\lg n)$（摊还和最坏情况下均如此）。这最多会改变两个集合的大小，其中一个增大，另一个减小，使势能最多增加 β。它还可能改变某些集合 $S_{j'}$ 的 $\mathbf{s}(S_{j'})$；但由于我们在减小元素键值时，会将 e 从 S_i 移到一个具有 $1 \le j \le i$ 的 S_j 中，因此 $\mathbf{s}(S_j)$ 只会增大，不会增加势能。总之，减小键值操作需要 $\mathcal{O}(\lg\lg n + \beta) = \mathcal{O}(\lg\lg n)$ 时间（摊还和最坏情况下均如此）。

减小键值可能会使集合 S_i 变为空集，而 $2 \le i \le \ell$；不过，之后可以通过遗忘枢轴将这些集合移除，且不会增加时间或势能，因为集合数量仍保持有界，从而维持不变式（S）。

3.5. 删除最小值的摊还成本

删除最小值会扫描 S_1 以找到最小元素 e 并将其删除；此外，我们通过围绕其中位数进行划分，将 S_1 分成两个新的集合 S_1 和 S_2。划分时，我们使用确定性选择。对 S_1 进行划分的总体实际代价为 $\mathcal{O}(|S_1|)$。

最后，我们通过调用遗忘枢轴来建立不变式（S）。根据集合数量的不变式（S），并且由于划分集合 S_1 会产生两个新集合，遗忘枢轴所需的摊还 $\mathcal{O}(\lg n)$ 实际时间。

接下来分析势能的变化。首先注意，连接规则 (C) 的设计正是为了确保将 S_{j+1} 连接到 S_j 时，连接前后 φ_{j+1} 和 φ_j 的贡献都为零，因此连接完全不会改变势能。更具体地，假设 $|S_j| + |S_{j+1}| < \mathbf{s}(S_j)$，并按照连接规则 (C) 将 S_j 和 S_{j+1} 连接起来，得到新的集合 S_j。此时势能因 φ_j 和 φ_{j+1} 而减少，并因新的势能 $\overline{\varphi}_j$ 而增加。根据定义，这些势能为

$$\begin{aligned} \varphi_j &= \beta \cdot \max\{0, |S_j| - \mathbf{s}(S_j)\} &&= 0 \\ \varphi_{j+1} &= \beta \cdot \max\{0, |S_{j+1}| - (\mathbf{s}(S_j) + |S_j|)\} &&= 0 \\ \overline{\varphi}_j &= \beta \cdot \max\{0, (|S_j| + |S_{j+1}|) - \mathbf{s}(S_j)\} &&= 0\,. \end{aligned}$$

因此，连接时势能的变化 $\Delta\Phi$ 为 0。

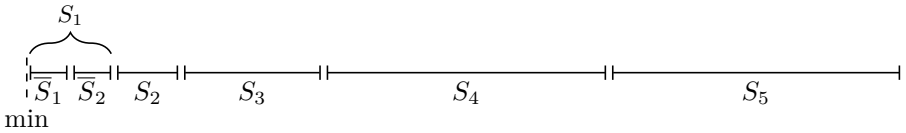


图 4： LP 堆中集合的示意图，以及在删除最小值时将 S_1 划分为 $\overline{S}_1$ 和 $\overline{S}_2$ 的示意图。

接下来分析将集合 S_1 划分为新集合 S_1 和 S_2 对势能的影响。图 4 展示了划分 S_1 后的情况。对于集合 S_1，我们失去 $\overline{\varphi}_1 = \beta|S_1|$，并从 S_1 和 S_2 中获得 $\varphi_1 + \varphi_2 = \beta|S_1|$，因为我们选择较大的中位数 s.t. $\mathbf{s}(S_2) = |S_1| \ge |S_2|$，即 $\varphi_2 = 0$。所有其他集合都失去一个较小的元素，因此势能增加了 $\le \ell \cdot \beta$。根据构造，$|S_1| \le \frac{1}{2}|S_1|$（回想一下，我们在将 S_1 划分为 S_1 和 S_2 之前会从中删除一个元素），所以 $\Delta\Phi \le -\frac{\beta}{2}|S_1| + \ell\beta$。因此，对于足够大的 $\beta \ge 2$，总摊还成本为

$$\mathcal{O}(\Delta\Phi + |S_1| + \lg n) \;=\; \mathcal{O}(\lg n)\,.$$

3.6. 扩展

这里我们将讨论 LP 堆的一些其他性质；理解（或讲授）LP 堆并不需要这些性质。

3.6.1. 更多操作

我们可以通过在 LP-堆插入和 LP 堆减键时以常数时间维护最小值，并在 LP 堆删除最小值后扫描 S_1，以摊还时间支持 LP-堆查找最小值，其时间为$\mathcal{O}(1)$。请注意，$|S_1|$ 可能任意大，也就是说，对于每次 LP-堆查找最小值操作，我们都无法承担扫描 S_1 的开销。

我们还可以支持 LP-堆增加键操作，或一般的 LP-堆删除（ptr）操作，其总摊还成本为 $\mathcal{O}(\beta \lg n + \lg\lg n) = \mathcal{O}(\lg n)$。对于这两种操作，$S_1$ 可能变为空；此外，对于一般的 LP-堆删除（ptr），n 会减小，这可能违反不变量（S），因此需要执行遗忘枢轴以重新建立（S）。这也会使成本增加$\mathcal{O}(\lg n)$。

对给定的 n 个元素集合执行 LP-堆构建操作，可以在$\mathcal{O}(n)$ 时间内完成。为此，我们只需将所有元素声明到 S_1 中。实际代价（如果元素以链表形式给出）甚至可以是常数，但我们必须为 βn 势能的增加付出代价，因此总摊还成本为$\mathcal{O}(n)$。

对两个堆取并集的 LP-堆合并操作难以高效支持。可以使用 LP-堆构建，通过连接两个输入堆来简单地创建一个新的 LP 堆，其摊还成本关于元素总数为线性。

3.6.2. 划分

我们可以用随机化选择 [Hoa61, FR75]，替代确定性的线性时间选择 [BFP+73]，从而得到相同的期望摊还时间。

更重要的是，我们还可以简单地执行一轮（随机）划分：划分后，势能减少 $\min\{|S_1|, |S_2|\}$。不选择中位数作为枢轴，而是在 S_1 的元素中随机选择一个，因此减少量的期望值关于 $|S_1|$ 呈线性，而删除最小值的摊还时间仍为期望 $\mathcal{O}(\lg n)$。

3.6.3. LP 快速堆

快速堆布局（第 1.1 节中有更详细的介绍）为每个集合使用链表提供了一种颇具吸引力的替代方案：虽然插入和删除操作需要进行更多数据移动，但我们可以将所有集合连续存储在一个数组中，并用枢轴将它们分隔开。唯一额外的空间是一个静态数组 T，其中包含 $O(\lg n)$ 个枢轴位置。

相邻集合仍可在常数时间内连接（只需将一个枢轴降为普通元素即可），而且现在可以原地进行划分，从而降低实际运行时间开销。虽然我们仍可通过对枢轴进行二分查找，来查找要插入元素的集合，但插入或删除操作现在必须通过每个集合至多一次交换来实现，因此所有操作的总运行时间为 $\Theta(\lg n)$。

3.6.4. 指针机版本

LP 堆已经处于指针机计算模型之内，除索引 T 外，入度节点均有界。通过用任意平衡二叉搜索树替换有序数组和二分查找，LP 堆便完全处于该模型之中。

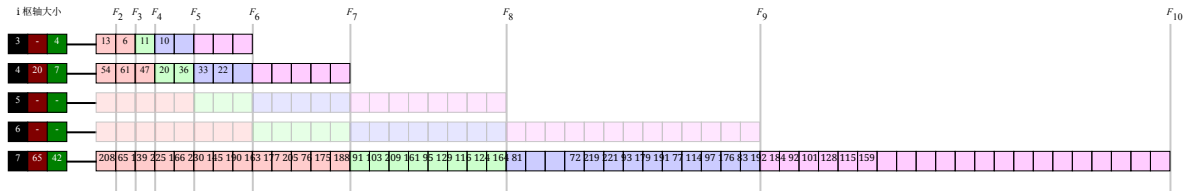


图5：下一代斐波那契堆示意图。虽然每个集合都存储为链表，但此图使用颜色来显示每个链表的大小、斐波那契数和大小势函数之间的关系。

4. 斐波那契堆：新一代

下一代斐波那契堆（FH:TNG）[2] 遵循我们为基于划分的简单堆所建立的框架，仅有少数鲜明特征：集合既可以存在，也可以不存在。若某个集合存在，其大小必须介于第 i个斐波那契数（记为 F_i（$F_1 := F_2 := 1$；$F_i := F_{i-1} + F_{i-2}$））与第 $i+3$个斐波那契数 F_{i+3} 之间，唯一的例外是第一个集合 S_3 没有最小大小。最后，我们要求连续的空集至多八个（连续空不变式），连续的非空集合至多两个（连续非空不变式），以便为拆分和连接留出空间。

这些操作的实现遵循我们的基本框架。唯一的区别在于违反不变量时该如何处理。我们使用不变量恢复操作按如下方式恢复不变量：如果一个集合 S_i 包含 F_{i+3} 个元素，我们称其为满，此时有两种处理方式：如果下面的集合 S_{i+1} 为空，就将 S_i 直接移至 S_{i+1}。我们称之为向下溢出操作。如果S_{i+1} 非空，根据连续非空不变式，S_{i+2} 必为空。我们将 S_i 中的所有 F_{i+3} 个元素拼接到集合 S_{i+1} 中，然后使用线性时间选择从合并后的集合中取出最大的 F_{i+3} 个元素，并将它们放入 S_{i+2}；这样 S_{i+1} 的大小保持不变，但其内容肯定会改变。我们称之为溢出穿越操作。如果一个集合 S_i 达到其最小大小 F_i，它就处于欠满状态；根据 S_{i-1} 是否为空，我们执行向上欠流操作或向下溢出操作。这两种操作分别与两种溢出操作对称。如果违反连续非空不变式，就会出现三个连续的非空集合，且它们下方有一个空列表。我们将其中最上面的两个集合拼接起来，并将合并后的集合放入下方的空槽位；我们称之为向下合并操作。

如果连续空集不变式遭到违反，就会出现九个连续的空集，且其下方有一个非空集合。我们取下方的非空集合，通过线性时间选择将其分裂到正上方的两个集合槽位中。我们称之为向上分裂，其耗时与所涉及集合的大小成线性关系。向下合并只需常数时间，因为它只是合并两个链表；而向上分裂则需要根据集合的大小，在线性时间内执行选择并遍历集合，将其分成两部分。

线性时间拆分与常数时间合并下移之间的不对称性至关重要，并且与我们允许减小键值但不允许增大键值这一事实密切相关！其直观理解是，反复的减小键值操作会迫使执行开销较低的向下合并，因为所有元素都在缓慢地被移入更大的集合。例如，可以通过反复对最大元素执行减小键值，使其成为最小值来实现这一点。然而，拆分操作仅由开销更高的删除最小值操作触发，因此其更高的开销可以得到偿付。

[2]这个名称旨在体现这样一个事实：在原始的斐波那契堆以及这里的结构中，斐波那契数都发挥着至关重要、但截然不同的作用。

在此我们给出分析中使用的势函数。该数据结构的势能是三个组成部分的求和，其中每个组成部分都针对结构中的每个集合定义：

非空势能。定义 $\varphi_i^{\text{nonempty}}$：若 S_i 非空，则其值为 1；若 S_i 为空，则其值为 0。此势能用于支付向下合并操作的成本；该操作会以单位成本使非空集合的数量减少一个。

大小势。当 S_i 接近满载（F_{i+3} 个元素）或欠满（F_i 个元素）时，赋予其势能；当其大小处于允许范围的中间（从 F_{i+1} 到 F_{i+2}个元素）时，势能为 0。这是数组倍增中使用的经典势函数，用于支付下溢和溢出操作的成本。图5用不同颜色标示了这三个范围。

$$\varphi_i^{\text{size}} := \begin{cases} 0 & \text{if } S_i \text{ is empty} \\ F_{i+1} - |S_i| & \text{if } F_i \le |S_i| < F_{i+1} \text{ (green in the figure)} \\ 0 & \text{if } F_{i+1} \le |S_i| \le F_{i+2} \text{ (blue in the figure)} \\ |S_i| - F_{i+2} & \text{if } F_{i+2} < |S_i| \le F_{i+3} \text{ (purple in the figure)} \end{cases}$$

向上势能。此势能赋予上方元素较少的非空集合，用于支付向上分裂操作的成本。直观来说，这是一种机制，使删除最小值操作能够向 Θ 个（$\lg n$）集合各支付单位成本；当大多数上方元素已被删除、需要进行向上分裂时，这些集合便可将这笔成本积攒起来。令$\uparrow_i$ 定义为 $\sum_{j=1}^{i-1} |S_j|$。集合 S_i、$\varphi_i^{\uparrow}$ 的向上势能定义为

$$\varphi_i^{\uparrow} := \begin{cases} 0 & \text{if } i \le 2 \text{ or } S_i \text{ is empty} \\ \max\{0,\, F_{i-3} - \uparrow_i\} & \text{otherwise} \end{cases}$$

数据结构 Φ 的势能是每个φ_i集合的上势能、大小势能和集合势能之和；我们将这个和称为 φ_i：$\Phi := \sum_{i=0}^{k}(\varphi_i^{\uparrow} + \varphi_i^{\text{size}} + \varphi_i^{\text{nonempty}})$。

事实。下面我们给出一系列与这些势能相关的事实。

事实 4.1： 如果 S_i 非空，且 S_{i-1}、S_{i-2} S_{i-3} 中至少有一个非空，那么 $\varphi_i^{\uparrow} = 0$。◁

证明：如果 S_{i-1}、S_{i-2} 和 S_{i-3} 中任何一个非空，它们就会有最小的 F_{i-3} 项。因此$\uparrow_i \ge F_{i-3}$，从而 $\max\{0,\, F_{i-3} - \uparrow_i\} = 0$。□

事实4.2： $\uparrow_i \le F_{i+4} - 1$ ◁

证明：$\uparrow_i = \sum_{j=1}^{i-1} |S_j| \le \sum_{j=1}^{i-1} F_{i+3} = \sum_{j=4}^{i+2} F_i \le F_{i+4} - 1$ 根据经典恒等式 $\sum_{i=1}^{n} F_i = F_{i+2} - 1$ □

事实 4.3： 如果某个操作在 $S_i, S_{i+1}, \ldots S_j$之间重新分配项目，既不插入也不删除，$\varphi_\ell^{\uparrow}$ 在 $\ell \in [i, j]$之外不会改变。◁

证明：回想一下，$\varphi_\ell^{\uparrow}$ 是由 S_ℓ 是否为空以及 $\uparrow_\ell$共同决定的，其中 $\uparrow_\ell$定义为 $\sum_{m=1}^{\ell-1} |S_m|$；如果重新分配的集合完全位于这个求和范围之内或完全位于范围之外，它就不会改变。□

事实 4.4： 对于 $i = O(1)$， $\varphi_i = O(1)$。 ◁

下一个事实源于这样的直觉：集合 S_i 可以拆分为 S_{i-1} 和 S_{i-2}，或者执行逆拼接，而不会增加大小势。这可以由斐波那契数的性质得出：具有特定满度的集合可以拆分为两个满度相同的集合，因此势能也相同。

事实 4.5： 非空集合 S_i 中的元素可以分配到先前为空的集合 S_{i-1}和 S_{i-2} 中，使集合势能保持不变。反过来，也可以将非空集合 S_{i-1} 和 S_{i-2} 中的元素拼接到先前为空的集合 S_i 中，而不增加势能。◁

证明： 一般而言，如果 S_i 包含介于 F_{i+j}和 F_{i+j+1}之间数量的元素，对于 $j \in 0,1,2$，我们可以按比例分配这些元素，使 S_{i-1} 包含介于 F_{i+j-1} 和 F_{i+j} 之间数量的元素，且 S_{i-2} 包含介于F_{i+j-2} 和 F_{i+j-1} 之间数量的元素；这自然可由斐波那契数的定义得出。在任何此类分配中，对于 $j = 0$ 的情形，大小势原本为 $F_{i-1} - |S_i|$，分配后仍为 $F_{i-1} - |S_i|$；对于 $j = 1$，为0；对于 $j = 2$，为 $|S_i| - F_{i+2}$。

对于合并的逆操作，设置 j，使得 S_i 将拥有介于 F_{i+j} 和 F_{i+j+1}之间的元素。如果 $j = 1$，则这一点平凡成立，因为势能将变为零。对于 $j = 0$ 和 $j = 2$ 的情形，少量数学计算即可证实这一直觉：

$$\underbrace{\varphi_i^{\text{size}}}_{\text{after}} = |S_i| - F_{i+1} = \overbrace{|S_{i-1}| - F_i + |S_{i-2}| - F_{i-1}}^{\substack{|S_{i-1}|-F_i\le\varphi_{i-1}^{\text{size}} \\ |S_{i-2}|-F_{i-1}\le\varphi_{i-2}^{\text{size}}}} \le \underbrace{\varphi_{i-1}^{\text{size}} + \varphi_{i-2}^{\text{size}}}_{\text{before}}$$

$$\underbrace{\varphi_i^{\text{size}}}_{\text{after}} = F_{i+3} - |S_i| = \overbrace{F_{i+2} - |S_{i-1}| + F_{i+1} - |S_{i-2}|}^{\substack{F_{i+2}-|S_{i-1}|\le\varphi_{i-1}^{\text{size}} \\ F_{i+1}-|S_{i-2}|\le\varphi_{i-2}^{\text{size}}}} \le \underbrace{\varphi_{i-1}^{\text{size}} + \varphi_{i-2}^{\text{size}}}_{\text{before}}$$

□

4.1. 向上分裂

见图 6。当“一行中至多有 8 个连续空集”这一不变式被违反时，即存在 9 个连续空集时，会调用向上分裂。具体而言，当 S_i和 S_{i-10} 非空且 $S_{i-1} \dots S_{i-9}$ 为空时，会对集合 i 调用该操作。这要求 $i \ge 11$。

将元素从 S_i 移动到 S_{i-1} 和 S_{i-2} 的方式应与它们的大小成比例，即依照黄金比例。更具体地说，如果 S_i中有介于 F_l 和 F_{l-1}之间的元素，则应将这些元素移动，使得 S_{i-1} 中有介于 F_{l-1} 和 F_{l-2} 之间的元素，且 S_{i-3}中有F_{l-2} 和 F_{l-3} 个元素；由于斐波那契数的定义，这当然是可能的。

通过线性扫描，将新的枢轴设为各集合中的最小值。

引理 4.6： 对 $S_i, i \ge 11$ 执行向上分裂操作时，其摊还成本至多为 *0*。◁

证明： 向上分裂操作的名义成本定义为 F_{i-6}，因为该操作所涉及的集合需要线性时间处理。为了计算势变化，我们考察势函数的三个组成部分。

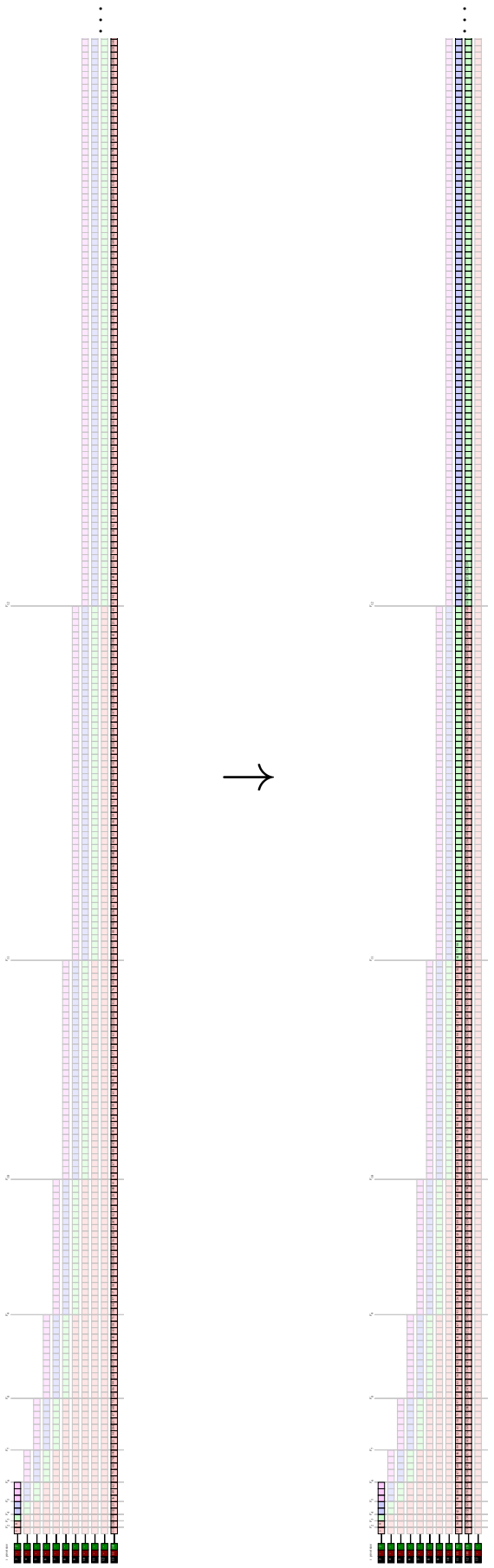


Figure 6: Split-up operation. There are 9 consecutive empty sets in a row $S_2 \dots S_{10}$. The set S_{11} is then proportionally split into sets S_9 and S_{10}. In all cases the nonempty sets remain in the same color range, in this case green, which represents one of the three cases of the size potential. Empty cells continue off the page.

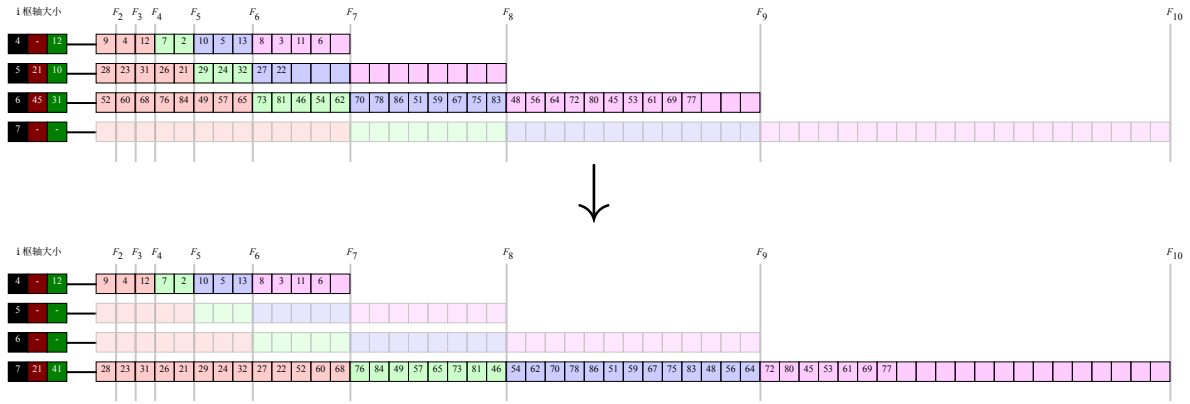


图 7： 当一行中有三个非空集合时执行的向下合并操作，此处为 S_4、S_5、S_6，而 S_7 下方的集合为空。集合 S_5 和 S_6 拼接到集合 S_7 的空槽位中。

变化量 $\varphi_i^{\uparrow}$： 这是 $\sum_{i=0}^{k}\varphi_i^{\uparrow}-\sum_{i=0}^{k}\varphi_i^{\uparrow}$。不过，利用事实 4.3，并回想空集的向上势能为零，可以将其简化为 $\varphi_{i-1}^{\uparrow}+\varphi_{i-2}^{\uparrow}-\varphi_i^{\uparrow}$。现在，我们可以分别界定这些量。首先，根据事实 4.1，有 $\varphi_{i-1}^{\uparrow}=0$。接下来，对于 $\varphi_{i-2}^{\uparrow}$，它是 $\max\{0,\ F_{i-5}-\uparrow_{i-2}\}\le F_{i-5}$。

最后，对于 $-\varphi_i^{\uparrow}$：我们利用 $S_{i-1}\dots S_{i-9}$ 为空这一事实来界定 $\uparrow_i$：

$$\begin{aligned}
-\varphi_i^{\uparrow} &= -\max\{0,\ F_{i-3}-\uparrow_i\} && \text{By definition}\\
&= -\max\{0,\ F_{i-3}-\uparrow_{i-9}\} && \uparrow_i=\uparrow_{i-9}\\
&< -\max\{0,\ F_{i-3}-F_{i-5}-1\} && \uparrow_{i-9}< F_{i-5}-1 \text{ by Fact 4.2}\\
&= -F_{i-4}+1 && i\ge 11
\end{aligned}$$

因此，综合起来可得：

$$\sum_{i=0}^{k}\varphi_i^{\uparrow}-\sum_{i=0}^{k}\overline{\varphi}_i^{\uparrow}=\overline{\varphi}_{i-1}^{\uparrow}+\overline{\varphi}_{i-2}^{\uparrow}-\varphi_i^{\uparrow}\le 0+F_{i-5}-F_{i-4}+1=-F_{i-6}+1$$

势能变化： 0。

势能变化 $\varphi_i^{\text{nonempty}}$： $+1$，因为现在多了一个非空集合。

总之，根据定义，摊还成本等于名义成本加上势能变化。当 $i\ge 11$ 时，摊还成本至多为：
$F_{i-6}-F_{i-6}-1+0+1=0$ □

4.2. 向下合并

见图 7。当违反至多有两个连续非空集合这一不变式时，就会调用向下合并操作。也就是说，存在三个连续的非空集合 S_i、S_{i-1} 和 S_{i-2}，其中 S_{i+1} 为空（或不存在）。集合 S_i 和 S_{i-1} 拼接后形成新集合 S_{i+1}；这里我们利用了这样一个事实：集合以双向链表集合而非数组的形式存储，因此该操作可在常数时间内完成。枢轴 p_{i-2} 设为 p_{i-1}。

引理 4.7： 向下合并操作的摊还成本至多为 *0*。 ◁

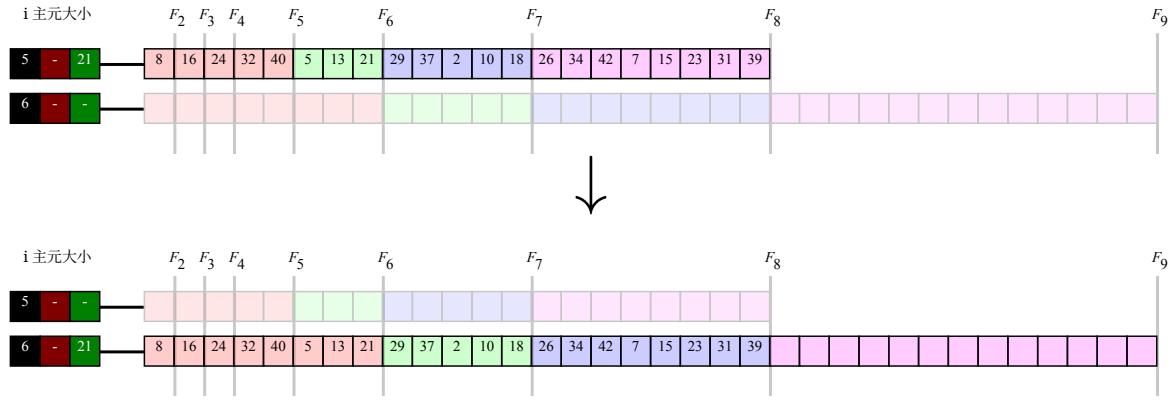


图 8： 在 S_6 为空时，对含有 $F_8 = 21$ 个元素的 S_5 执行向下溢出操作。

证明： 该操作的名义成本为 1。要计算势能的变化，我们分别考察势函数的三个组成部分：

变化量 $\varphi_i^{\uparrow}$： 0。根据事实 4.3，这是 $\varphi_{i+1}^{\uparrow} - \varphi_i^{\uparrow} - \varphi_{i-1}^{\uparrow}$。根据事实 4.1，由于 $S_{i-2} = \overline{S_{i-2}}$ 非空，这三项都为零。

φ_i^{size}的变化量： ≤ 0 根据事实 4.5。

变化量 $\varphi_i^{\text{nonempty}}$： -1 非空集合的数量减少 1 时。

总之，根据定义，摊还成本等于名义成本加上势变化：
$1 + 0 + 0 - 1 = 0$ □

4.3. 向下溢出

见图 8。当集合 S_i 已满（因此有 F_{i+3} 个元素），且其下方的集合 S_{i+1} 为空时，会执行向下溢出操作。该集合会在常数时间内从 S_i 移至 S_{i+1}。

引理 4.8： 当 $i \geq 3$ ◁ 时，集合 S_i 上的向下溢出操作的摊还成本至多为 *0*

证明： 向上溢出的名义成本为 1，因为它是一个常数时间操作。要计算势变化，我们分别考察势函数的三个组成部分：

变化量 $\varphi_i^{\uparrow}$： 根据事实 4.3，这是 $-\varphi_{i+1}^{\uparrow} - \varphi_i^{\uparrow}$，其至多为 $\varphi_{i+1\overline{\circ}}^{\uparrow}$ 根据向上势能的定义，$\varphi_{i+1}^{\uparrow} = \max\{0, F_{i-2} - \uparrow_{i+1}\} \leq F_{i-2}$。

变化量 φ^{size}： 由于 $|S_{i+1}| = F_{i+3}$，$\varphi_{i+1}^{\text{size}} = 0$。因此我们只需考虑 $-\varphi_i^{\text{size}}$：$-(\overbrace{|S_i|}^{F_{i+3}} - F_{i+2}) = F_{i+2} - F_{i+3} = -F_{i+1}$

变化量 $\varphi_i^{\text{nonempty}}$： 为 0，因为非空集合的数量没有变化。

总之，根据定义，摊还成本等于名义成本加上势变化$\leq \overbrace{-F_i}$为
$1 + F_{i-2} - F_{i+1} + 0 \leq 0$。

□

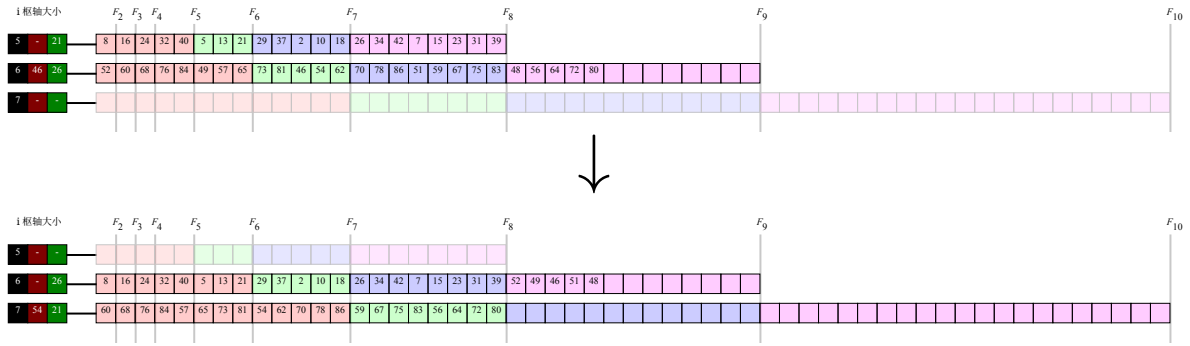


图9： 当 S_6 非空时，对含有 $F_8 = 21$ 个元素的 S_5 执行的一次溢出穿越操作。

4.4. 上溢贯通

参见图9。当集合 S_i 已满且包含 F_{i+3}个元素时，如果其下方的集合 S_{i+1} 非空，就会调用溢出穿越操作。根据连续非空不变式，不可能有超过两个连续的非空集合，因此这意味着 S_{i+2} 为空。

回想一下，在此操作中， S_i 和 S_{i+1} 会合并，并将合并集合中的最大元素移至 S_{i+2}，使 S_{i+1} 的大小保持不变，从而使 $|S_{i+2}| = |S_i| = F_{i+3}$。此移动需要使用线性时间中位数/顺序统计量算法，因此其名义成本为 F_{i-4}，因为 S_{i+2} 是所涉及的最大集合。枢轴 p_{i+1} 和 p_{i+2} 被设为各集合中的最小值。

引理4.9： 当 $i \geq 5$ 时，上溢贯通的摊还成本为 ≤ 0。 ◁

证明： 该操作的名义成本为 F_{i-4}，因为其所需时间与涉及的集合大小成线性关系。为计算势能的变化，我们考察势函数的三个组成部分：

变化 $\varphi^{\uparrow}$： 根据事实 4.3，该值为 $\varphi^{\uparrow}_{i+2} + \varphi^{\uparrow}_{i+1} - \varphi^{\uparrow}_{i+1} - \varphi^{\uparrow}_{i}$。根据事实 4.1， $\varphi^{\uparrow}_{i+1}$ 和 $-\varphi^{\uparrow}_{i+2}$ 均为零。结合 $\varphi^{\uparrow}_{i+1}$ 非负这一事实， $\varphi^{\uparrow}$ 的变化至多为

$$\overline{\varphi}^{\uparrow}_{i+1} = \max\{0,\, F_{i-2} - \uparrow_{i+1}\} \leq F_{i-2}.$$

变化 φ^{size}： 该值为 $\varphi^{\text{size}}_{i+2} + \varphi^{\text{size}}_{i+1} - \varphi^{\text{size}}_{i+1} - \varphi^{\text{size}}_{i}$。根据定义，由于 $|S_{i+2}| = F_{i+3}$， $\varphi^{\uparrow}_{i+2} = 0$。由于 $\varphi^{\text{size}}_{i+1} = \varphi^{\text{size}}_{i+1}$，这意味着 φ^{size} 的变化为 $-\varphi^{\text{size}}_{i}$。根据定义， $-\varphi^{\text{size}}_{i} = -(|S_i| - F_{i+2})$，而由于 $|S_i| = F_{i+3}$，其值为 $-F_{i+1}$。

势变化 $\varphi^{\text{nonempty}}$： 0，因为非空集合的总数没有变化。

总而言之，根据定义，摊还成本等于名义成本加上势变化：
$F_{i-4} + F_{i-2} - F_{i+1} + 0 \leq 0$ □

4.5. 下溢向上

参见图 10。当集合 S_i 欠满，因此有 F_i 个元素，且其上方的集合 S_{i-1} 为空时，就会调用向上欠流操作。在这种情况下，集合和枢轴会在常数时间内从 S_i 移动到S_{i-1}。

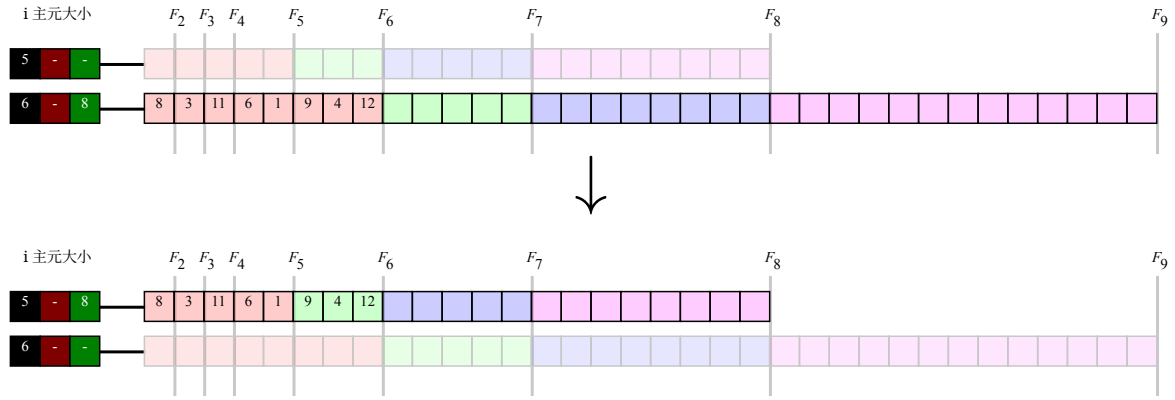


图 10： 在 S_5 为空时，对含有 $F_6 = 8$ 个元素的 S_6 执行向上欠流操作。

引理 4.10： 当 $i \geq 6$ 时，对 S_i 执行向上欠流操作的摊还成本至多为 *0*。◁

证明：该操作的名义成本为 1，因为它是一个常数时间操作。为计算势能的变化，我们分别考察势函数的三个组成部分：

变化： $\varphi^{\uparrow}$**：** 根据事实 4.3，这是 $\varphi^{\uparrow}_{i-1} - \varphi^{\uparrow}_{i}$。由于定义，$\varphi^{\uparrow}_{i-1} - \varphi^{\uparrow}_{i} \leq \varphi^{\uparrow}_{i-1} \leq F_{i-4}$ 满足 $\varphi^{\uparrow}_{i-1} = \max\{0,\ F_{i-4} - \uparrow_{i-1}\}$。

势变化 φ^{size}**：** 这是 $\varphi^{\text{size}}_{i-1} - \varphi^{\text{size}}_{i}$。根据定义，$|S_{i-1}| = F_i$，$\varphi^{\text{size}}_{i-1} = 0$。现在我们可以bound $-\varphi^{\text{size}}_{i}$：

$$-(F_{i+1} - \overbrace{|S_i|}^{i}) \leq -F_{i+1} + F_i = -F_{i-1}.$$

势变化 $\varphi^{\text{nonempty}}$**：** **0**。非空集合的数量没有变化。 □

总之，根据定义，摊还成本等于名义成本加上势变化：$1 + F_{i-4} - F_{i-1} + 0 < 0$，因为 $i \geq 6$。

4.6. 贯穿欠流

见图 11。当集合 S_i 欠满时，会调用向下溢出操作，因此其中有F_i 个元素，且其上方的集合 S_{i-1} 非空。根据连续非空不变式，连续的非空集合不能超过两个；这意味着 S_{i-2}非空。

回想一下，在此操作中，S_i 和 S_{i-1} 会合并，并将合并后集合中的最小元素移至 S_{i-2}，使得 S_{i-1} 的大小保持不变，因此 $|S_{i-1}| = |S_{i-1}|$ 和$|S_{i-2}| = |S_i| = F_i$。这需要使用线性时间中位数/顺序统计量算法，因此名义成本为 F_{i-6}，因为 S_i 是涉及的最大集合。枢轴 p_{i-1} 和 p_{i-2} 分别设为各集合中的最小值。

引理 4.11： 当 $i \geq 3$ 时，向下溢出操作的摊还成本至多为 *0*。 ◁

证明： 该操作的名义成本为 F_{i-6}。为了计算势变化，我们分别考察势函数的三个组成部分：

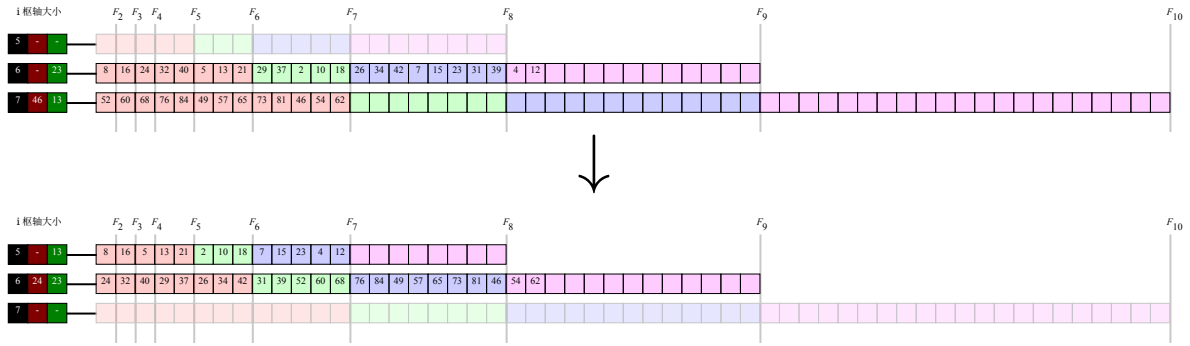


图 11： 当 S_5 非空时，对含有 $F_7 = 13$ 个元素的 S_7 执行的向下溢出操作。

变化量 $\varphi^{\uparrow}$： 根据事实 4.3，该值为 $-\varphi^{\uparrow}_{i-1} + \varphi^{\uparrow}_{i-2} - \varphi^{\uparrow}_{i} - \varphi^{\uparrow}_{i-1}$。根据事实 4.1，$\varphi^{\uparrow}_{i}$ 和 $\varphi^{\uparrow}_{i-1}$ 均为零。此外，$-\varphi^{\uparrow}_{i-1}$ 至多为 0。因此，$\varphi^{\uparrow}$ 的变化至多为 $\varphi^{\uparrow}_{i-2}$，即 $\max\{0, F_{i-5} - \uparrow_{i-2}\} \le F_{i-5}$。

变化量 φ^{size}： 这是 $\varphi^{\text{size}}_{i-2} + \varphi^{\text{size}}_{i-1} - \varphi^{\text{size}}_{i-1} - \varphi^{\text{size}}_{i}$。请注意，虽然 S_{i-1} 的内容会发生变化，但其大小不会改变，因此 $\varphi^{\text{size}}_{i-1} = \varphi^{\text{size}}_{i-1}$。另请注意，根据定义，$S_{i-2}$ 的大小为 F_i，因此根据定义 $\varphi^{\text{size}}_{i-2} = 0$。所以，$\varphi^{\text{size}}$ 的变化量为 $-\varphi^{\text{size}}_{i}$：

$$-\varphi^{\text{size}}_{i} - (F_{i+1} - \overbrace{|S_i|}^{F_i}) \le -(F_{i+1} - F_i) = -F_{i-1}$$

变化量 $\varphi^{\text{nonempty}}$： 0 非空集合的数量不变。

总而言之，根据定义，摊还成本等于名义成本加上势变化：$F_{i-6} + F_{i-5} - F_{i-1} + 0 < 0$。□

4.7. 插入

在插入操作中，会对枢轴执行二分查找，以确定将新插入的元素添加到哪个集合中。如果该元素小于最小的枢轴，则将其添加到第一个集合 S_3中。将此集合记为 j。j 的大小会更新，而 j 的枢轴不会更新，因为它仍然有效。如果该集合变得过满，则会执行一次向下溢出或上溢贯通操作；这会消除集合大小上的不变量违例，但可能导致 j 下方连续出现过多空集，以及 j上方连续出现过多非空集合。前者通过向上分裂修复，后者则通过向下合并修复，而向下合并可能会向下级联。

引理 4.12： 插入操作的摊还成本为 $O(\lg \lg n)$。 ◁

证明： 我们无需分析任何不变量恢复操作，因为已证明这些操作的摊还成本至多为零（但对常数大小集合调用的操作除外；这类操作最多发生常数次，且其摊还成本为常数）。

由于对 $\Theta(\lg n)$ 个集合的枢轴进行二分查找，该操作的名义成本为 $O(\lg \lg n)$。

要计算势变化，我们需要考察势函数的三个组成部分。

变化量 $\varphi_i^{\uparrow}$： 由于它被定义为 $\max\{0,\ F_{i-3} - \uparrow_i\}$，且仅当 $\uparrow_i$ 发生变化时才会递增，因此该势能最多增加 0。

变化 φ_i^{size}： ≤ 1。集合 j 现在多了一个元素，因此大小势能可能增加一个单位。

变化 $\varphi_i^{\text{nonempty}}$： 0，因为集合的数量没有变化。

总之，根据定义，不包括任何不变量恢复操作的摊还成本，等于名义成本加上势能的变化，其上限为：$O(\lg \lg n) + 0 + 1 + 0 = O(\lg \lg n)$。再计入调用不变量恢复操作的成本，这些操作的摊还成本总计为 $O(1)$，则总摊还成本为 $O(\lg \lg n)$。□

4.8. 删除最小值

在删除最小值操作中，我们从最小非空集合中查找并移除最小元素，将其记为 S_j。由于连续空集不变量，$j \leq 11$，因此这只需常数时间，因为 $|S_j| \leq F_{14} = 377$。我们更新集合 j 的大小。如果该集合变为欠满状态，我们执行一次下溢向上或贯穿欠流操作，这可能导致执行一连串级联的向下合并。

引理 4.13： 删除最小值操作的摊还成本为 $O(\lg n)$。 ◁

证明：我们无需分析任何恢复不变量子操作，因为已证明这些操作的摊还成本至多为零（但对常数大小集合执行的子操作除外；这类操作至多发生常数次，且每次的摊还成本都是常数）。

不计恢复不变量子操作，此操作的实际代价为 $O(1)$。
为了计算势能的变化，我们分别考察势函数的三个组成部分。

变化量 $\varphi_i^{\uparrow}$： 由于它对非空集合定义为 $\max\{0, F_{i-3} - \uparrow_i\}$（对空集则为 0），且每个 $O(\lg n)$ $\uparrow_i$ 至多减少 1，因此向上势能至多增加 $O(\lg n)$。

变化量 φ_i^{size}： ≤ 1。集合 j 现在少了一个元素，因此其大小势可能增加一个单位。

$\varphi_i^{\text{nonempty}}$ 的变化： 至多为 0。

总而言之，根据定义，摊还成本等于实际代价加上势能变化，即：$\mathcal{O}(1) + \mathcal{O}(\lg n) + 1 + 0 = \mathcal{O}(\lg n)$。再加上所有不变量恢复操作的调用，其摊还成本总计为 $\mathcal{O}(1)$，得到总摊还成本为 $\mathcal{O}(\lg n)$。□

4.9. 减小键值

此操作首先将元素从其当前集合中移除，S_j，减小其键值，然后将其重新插入适当的集合 S_k；关键在于，$k \leq j$ 因为键值只会减小。需要对先前的值进行两次二分查找（以更新元素原所在集合的大小），并在新集合中进行二分查找（以插入元素并更新集合大小）。两个集合的枢轴仍然有效，无需更改。移除元素后，如果集合欠满，则执行一次下溢向上或贯穿欠流操作，然后根据需要执行向上分裂和向下合并，直至恢复所有不变量；插入元素后，如果集合过满，则执行一次上溢向下或上溢贯通操作，然后根据需要执行向上分裂和向下合并，直至恢复所有不变量。

引理4.14： 减小键值的摊还成本为 $O(\lg\lg n)$。 ◁

证明：我们不必分析任何恢复不变量子操作，因为已证明这些操作的摊还成本至多为零（但对常数大小集合执行的操作除外；这类操作最多发生常数次，且每次的摊还成本为常数）。

名义 成本为 $O(\lg\lg n)$，这是由于执行了两次二分查找操作。
要计算势能的变化，我们需要考察势函数的三个组成部分。

变化 $\varphi_i^{\uparrow}$： 对于非空集合，其定义为 $\max\{0, F_{i-3} - \uparrow_i\}$；对于空集，则为 0。对于 $i < k$ 和 $i > j$，这一数值不会改变；但对于 k 和 j 之间的 i，也就是移除和重新插入之间的集合，$\uparrow_i$ 将增加 1，因此 $\varphi_i^{\uparrow}$ 将减少 1。这里至关重要的是，我们利用了键值只能减小这一事实；如果允许键值增大，此处的势能可能会增加 $\Theta(\lg n)$，但实际上它只会减小。

变化 φ_i^{size}： ≤ 2。插入和删除都可能使受影响集合的大小势发生单位变化。

变化 $\varphi_i^{\text{nonempty}}$： 0，因为集合的数量没有变化。

总之，根据定义，摊还成本等于实际代价加上势变化，其上界为：$\mathcal{O}(\lg\lg n) + 0 + 2 + 0 = \mathcal{O}(\lg\lg n)$。计入任何不变量恢复操作的调用后，这些操作的摊还成本总计为 $\mathcal{O}(1)$，因此总摊还成本为$\mathcal{O}(\lg\lg n)$。□

5. 指数上界集合堆

本节讨论分区式简单堆的另一种变体，它由集合S_1、…、S_ℓ和枢轴 $p_2 \leq \cdots \leq p_\ell$构成。同样，删除最小值的时间为摊还$\mathcal{O}(\lg n)$，而插入和减小键值的时间为摊还$\mathcal{O}(\lg\lg n)$。在这种变体中，集合大小具有指数递增上界，但没有下界，并且允许任意数量的集合为空。这与第 4 节 FH:TNG 更强的结构不变量形成对比。

不变式： $|S_i| < 3 \cdot 2^i$ 对于 $1 \leq i \leq \ell$。

堆操作的实现如下：

- 插入(e)：对枢轴进行二分查找，找到满足 $p_i \leq e < p_{i+1}$ 的 i，并将 e 追加到 S_i。如果$|S_i| = 3 \cdot 2^i$，则递归地将 S_i 推入 S_{i+1}（见下文）。

- 删除最小值()：如果 S_1 为空，先递归地从 S_2 将元素拉入 S_1（见下文）。从 S_1 中删除并返回最小值。如果 $\ell > 1 + \lg n$，令 $S_{\ell-1}$ 为 $S_{\ell-1}$ 和 S_ℓ 的连接，并在返回前丢弃 S_ℓ 和 p_ℓ，也就是说，ℓ 减一（这确保 $\ell \leq 1 + \lg n$）。

- 减小键值(ptr, 键值)：从当前集合中删除ptr 所指向的元素 $e \in S_j$（为了减小大小 $|S_j|$，我们通过对枢轴进行二分查找来找到 j）。将 e 的键值更改为键值，并对枢轴进行二分查找，以找到要添加 e的集合 S_i，然后将 e 追加到 S_i。如果$|S_i| = 3 \cdot 2^i$，则将 S_i 递归推送到 S_{i+1}。

推送。对于插入和减小键值，接收一个额外元素的集合 S_i 的大小可能会变为$|S_i| = 3 \cdot 2^i$。我们将 S_i 的全部 $3 \cdot 2^i$ 个元素 S_{i+1}推送，并设置 $S_i = \emptyset$。通常，将一组元素 X 递归推送到 S_j 时，始终满足 $2^{j-1} \le |X| \le 3 \cdot 2^{j-1}$。如果推送之前 $|S_j| < 2^j$（即 S_j 太小，无法推送到 S_{j+1}），则在常数时间内将 X 与 S_j 拼接，使得 $|S_j| < 5 \cdot 2^{j-1}$，然后结束推送。否则，集合 X 变为 S_j，并将旧的 S_j 递归推送到 S_{j+1}。相应地更新枢轴；当$j = \ell + 1$ 时，会创建一个新集合 $S_{\ell+1} = X$，即 ℓ 增加一。

拉取。将元素递归拉取到 S_1中，或者更一般地说，在 S_i 为空时拉取到 S_i 中的过程如下：如果 S_{i+1} 为空，我们首先递归地将元素拉取到 S_{i+1}中。如果$|S_{i+1}| \le 2^{i-1}$，我们就交换 S_{i+1} 和空的 S_i的角色，即在常数时间内将所有元素从 S_{i+1} 移到 S_i。否则，我们从 S_{i+1}中选出并删除 2^{i-1} 个最小元素，耗时$\mathcal{O}(2^i)$，并令 S_i 为这些元素（并更新 p_{i+1}；这里元素互不相同至关重要，以确保 $\max S_i < p_{i+1} = \min S_{i+1}$）。

5.1. 分析

为证明此结构的摊还性能，我们采用以下势函数$\Phi = \varphi^{\text{insert}} + \varphi^{\text{push}} + \varphi^{\text{pull}}$，其中

$$\varphi^{\text{insert}} := \sum_{i=1}^{\ell} \varphi_i^{\text{insert}} \qquad \varphi^{\text{push}} := \sum_{i=1}^{\ell} \varphi_i^{\text{push}} \qquad \varphi^{\text{pull}} := \sum_{i=1}^{\ell} \varphi_i^{\text{pull}}$$

$$\varphi_i^{\text{insert}} := \max\left\{0,\ |S_i| - 5 \cdot 2^{i-1}\right\} \in [0, 2^{i-1}] \qquad \varphi_i^{\text{push}} := \left\lfloor |S_i|/2^i \right\rfloor \in [0, 3]$$

$$\varphi_i^{\text{pull}} := \max\left\{0,\ 2^{i-1} - \sum_{j=1}^{i} |S_j|\right\} \in [0, 2^{i-1}]$$

基本思路是为每个集合 S_i 赋予一个势能。一个较大的集合会有推送势能φ_i^{push}，当该集合被移至 S_{i+1}时释放一个单位的势能。向 S_i中插入元素可能触发对 S_i的推送。累积的插入势能 $\varphi_i^{\text{insert}}$将在推送$S_i$ 时释放，以弥补拉取势能 φ_i^{pull} 的增加。最后，直观而言，拉取势能与 S_i 中前 2^{i-1} 个“槽位”里的每个“空槽位”相关联；这些势能用于支付从后续各集合 S_{i+1}，……，S_ℓ 中递归填充这些槽位的成本。图12展示了集合大小与各种势能之间的关系。

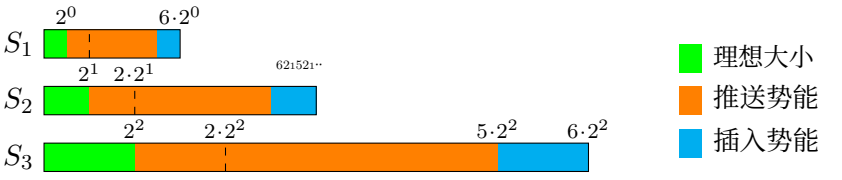


图12： S_i 个集合的集合大小与势能示意图。

引理 5.1： 对于含有 n 个元素的堆，$\ell \le 1 + \lg n$。 ◁

证明： S_ℓ 是在插入操作期间，至少有 $2^{\ell-1}$ 个元素从 $S_{\ell-1}$ 推送到 S_ℓ 时创建的，即$n \ge 2^{\ell-1}$ 意味着 $\ell \le 1 + \lg n$。在删除最小值期间，n 减一。当 $\ell > 1 + \lg n$时，$S_{\ell-1}$与 S_ℓ的连接确保 ℓ减一，并且删除最小值后 $\ell \le 1 + \lg n$。□

引理 5.2： 一次插入操作（不包括递归推送）的最坏情况耗时为 $O(\lg \ell)$，势变化为 $\Delta\Phi \leq 2$。◁

证明：对 i 进行二分查找并追加到 S_i，最坏情况下需要 $\mathcal{O}(\lg \ell)$ 时间。将 $|S_i|$ 增加 1，最多会使每个 $j \geq i$ 对应的 $\varphi_i^{\text{insert}}$ 和 φ_i^{push} 各增加 1，并使 φ_j^{pull} 减少。因此可得 $\Delta\Phi \leq 2$。□

引理 5.3： 一次减小键值操作（不包括递归推送）最坏情况下需要 $O(\lg \ell)$ 时间，且势能变化为 $\Delta\Phi \leq 2$。◁

证明：执行减小键值操作需要对 $\ell - 1$ 枢轴进行两次二分查找，最坏情况下需要 $\mathcal{O}(\lg \ell)$ 时间；将一个元素从 S_i 移至 S_j，最坏情况下需要 $\mathcal{O}(1)$ 时间，其中 $1 \leq j \leq i \leq \ell$。势能 φ_m^{pull} 仅在 $j \leq m < i$ 时可能发生变化，此时势能减少 1；$\varphi_i^{\text{insert}}$ 和 φ_i^{push} 只可能减少，而 $\varphi_j^{\text{insert}}$ 和 φ_j^{push} 最多增加 1。因此可得 $\Delta\Phi \leq 2$。□

引理 5.4： 当 $|S_i| = 3 \cdot 2^i$ 时，将 S_i 递归推送到 S_{i+1} 中需要最坏情况 $O(m - i)$ 时间，且势变化为 $\Delta\Phi \leq -m + i + 3$，其中 m 满足 $i < m \leq \ell + 1$。◁

证明：假设递归推送持续进行，直到将元素推送到 S_m 中以实现 $i < m \leq \ell + 1$，即在推送之前，$i < j < m$ 为 $|S_j| \geq 2^j$，而 $|S_m| < 2^m$ 为 $m - i$。$m - i$ 次推送中的每一次都需要最坏情况常数时间，用于重命名集合或拼接两个双向链表。

下文中，请注意，值上方的横线表示操作后的值。我们首先考虑势能 φ^{insert} 的变化。推送之前，我们有 $|S_i| = 6 \cdot 2^{i-1}$，即 $\varphi_i^{\text{insert}} = 2^{i-1}$。由于推送之后 $|S_i| = 0$，因此有 $\varphi_i^{\text{insert}} = 0$。对于 $i + 1 \leq j \leq m$，在 2^{j-1} 与 $3 \cdot 2^{j-1}$ 个元素之间会被推送到 S_j。因此，递归推送之后 $|S_j| \leq 5 \cdot 2^{j-1}$，对于 $i \leq j \leq m$，即 $\varphi_i^{\text{insert}} = \cdots = \varphi_m^{\text{insert}} = 0$。由此可得 $\Delta\varphi^{\text{insert}} \leq -2^{i-1}$。

对于 φ^{pull}，注意 $|S_j|$ 只会因 $i \leq j \leq m$ 而改变，并且只有这些 φ_j^{pull} 会改变。对于 $i < j < m$，递归推送前后都有 $|S_j| \geq 2^{j-1}$，也就是说，$\varphi_j^{\text{pull}} = \varphi_j^{\text{pull}} = 0$ 增加了 $|S_m|$，即 φ_m^{pull} 只会减少，而 φ_i^{pull} 的最大增幅是从 0 到 2^{i-1}，也就是 $\varphi^{\text{pull}} \leq 2^{i-1}$。

对于 φ^{push}，我们有 $|S_i| = 0$，即 $\varphi_i^{\text{push}} = 0$。对于 $i < j < m$，由于递归推送之前有 $S_j = S_{j-1}$ 和 $|S_{j-1}| \geq 2^{j-1}$，因此有 $\varphi_{j-1}^{\text{push}} \geq 1$ 和 $\varphi_j^{\text{push}} \leq \varphi_{j-1}^{\text{push}} - 1$。由于 $\varphi_i^{\text{push}} = 0$ 和 $\varphi_m^{\text{push}} \leq 2$，可得 $\Delta\varphi^{\text{push}} \leq 2 - (m - i - 1) = -m + i + 3$。□

引理 5.5： 删除最小值操作（不包括递归拉取）需要最坏情况 $O(1)$ 时间，势变化为 $\Delta\Phi \leq \ell$。◁

证明：从 S_1 中移除最小值需要最坏情况常数时间，因为 $|S_1| \leq 6$。减小 $|S_1|$ 只能使 $\varphi_1^{\text{insert}}$ 和 φ_1^{push} 减小，或保持不变。所有其他 $\varphi_j^{\text{insert}}$ 和 φ_j^{push} 都保持不变。将 $|S_1|$ 减小 1，最多会使所有 $1 \leq j \leq \ell$ 对应的 φ_j^{pull} 增加 1。势能的总变化为 $\Delta\Phi \leq \ell$。

由于删除一个元素会使 n 减小 1，因此我们可能有 $\ell > 1 + \lg n$。此时，删除最小值操作可以在最坏情况常数时间内拼接 $S_{\ell-1}$ 和 S_ℓ，并且 ℓ 减小 1，从而确保 $\ell \leq 1 + \lg n$。由于拼接之前 $\ell > 1 + \lg n$，我们有 $|S_{\ell-1}| \leq n < 2^{\ell-1}$，而 $\varphi_{\ell-1}^{\text{push}} = \varphi_{\ell-1}^{\text{insert}} = 0$ 和 $\varphi_{\ell-1}^{\text{pull}}$ 只会减小。因此，在删除最小值操作中执行这项额外检查，只会使 Φ 减小。□

引理 5.6： 当 $|S_1| = 0$ 时，将元素递归拉取到 S_1 中的最坏情况时间为 $O(m + 2^{i-1})$ *time*，势变化为 $\Delta\Phi \leq -m - 2^{i-1} + 1$，其中 i 和 m，且 $1 \leq i < m \leq \ell$。◁

证明：假设将元素拉取到 S_1 中时，我们有 $|S_1| = \cdots = |S_{m-1}| = 0$ 和 $|S_m| > 0$，其中某个 $1 < m \le \ell$。此外，令 i 为满足 $2^{i-1} < |S_m|$ 的最大 $i < m$。也就是说，S_m 将依次与 $S_{m-1}, \ldots, S_{i+1}$ 交换；对于 $1 \le j \le i$，我们通过从 S_{j+1} 中选取 2^{j-1} 个最小元素来填充 S_j，耗时 $\mathcal{O}(2^j)$。因此，最坏情况时间为 $\mathcal{O}\left(m + \sum_{j=1}^{i} 2^j\right) = \mathcal{O}(m + 2^i)$。

执行提取操作后，对于 $|S_j| \le 2^{j-1}$ $1 \le j < m$，即 $\varphi_j^{\text{insert}} = 0$ 和 $-\varphi_j^{\text{push}} = 0$。由于 $|S_m|$ 减小，因此我们有 $\Delta\varphi_m^{\text{insert}} \le 0$ 且 $\Delta\varphi_m^{\text{push}} \le 0$。总的来说，$\varphi^{\text{insert}}$ 和 φ^{push} 只能减小。

对于 φ^{pull}，我们有 φ_m^{pull}、……、$-\varphi_\ell^{\text{pull}}$ 保持不变，因为我们只将元素从 S_m 移至 S_0、……、S_{m-1}。由于至少有一个元素被移至 S_0，因此对于所有 $1 \le j < m$，φ_j^{pull} 至少减少 1。对于 $1 \le j \le i$，恰有 2^j 个元素被移至 S_j，随后这些元素的子集被移至 S_{j-1}、……、S_1，即我们有 $\Delta\varphi_j^{\text{pull}} = -2^{j-1}$。由此可得 $\Delta\varphi^{\text{pull}} \le -\sum_{j=1}^{i} 2^{j-1} - (m - 1 - i)$ $= -2^i - m + i + 2 \le -2^{i-1} - m + 1$，这也是 $\Delta\Phi$ 的一个上界。□

根据引理 5.1，我们有 $\ell = \mathcal{O}(\lg n)$。引理 5.1、5.2 和 5.4 表明，插入的摊销时间为 $\mathcal{O}(\lg \lg n)$。引理 5.1、5.3 和 5.4 表明，减小键值的摊销时间为 $\mathcal{O}(\lg \lg n)$。最后，引理 5.1、5.5 和 5.6 表明，删除最小值的摊销时间为 $\mathcal{O}(\ell) = \mathcal{O}(\lg n)$。

推论 5.7： 插入和减小键值的摊销时间为 $O(\lg\lg n)$，而删除最小值的摊销时间为 $O(\lg n)$。◁

6. 结论

我们介绍了三种优先队列实现，它们遵循一种简单的基于分区的堆框架，并使用基本数据结构和算法。特别是对于最简单的设计——惰性分区堆，我们给出了完整、自成一体的描述和分析。

将数组上的二分查找替换为二分搜索树后，该实现可在指针机模型中运行，且节点的入度仅需为常数，因此能够高效地实现持久化。

由于实现简单且结构线性，我们的实现似乎很适合改造以适用于外存，但支持基于指针的减键操作会更加复杂（正如 [NPPS11] 所指出的）。我们将此留待未来工作。

将我们的规则用作 quickheaps 中的连接规则，有望得到一种极其实用且稳健的优先队列实现，并结合以往已知 quickheaps 变体的优点。我们计划在未来工作中进行详细的实证评估。

细心的读者可能已经注意到，插入和减小键值操作的摊还时间为常数时间，但对 $\mathcal{O}(\lg n)$ 个枢轴进行二分查找的时间除外。若使用融合树 [FW93] 等非指针机模型的数据结构来存储枢轴，运行时间可降至 $\mathcal{O}(\lg_w \lg n)$；在 $w = \Omega(\log n)$ 的标准假设下，这些操作的摊还运行时间为 $\mathcal{O}(1)$。

致谢。我们感谢审稿人提出的宝贵意见。第二作者的工作得到比利时法语区国家科学研究基金会（FNRS）的资助。最后一位作者部分得到英国工程与物理科学研究委员会资助项目 EP/X039447/1 的支持。

引用

[BFMZ04] Gerth Stølting Brodal、Rolf Fagerberg、Ulrich Meyer与Norbert Zeh。面向无向图广度优先搜索与最短路径的缓存无关数据结构与算法。载于Torben Hagerup与Jyrki Katajainen编，《算法理论——SWAT 2004》，第九届斯堪的纳维亚算法理论研讨会，Humlebaek，丹麦，2004年7月8日至10日，会议论文集，《计算机科学讲义》第3111卷，第480–492页。施普林格，2004年。 doi:10.1007/978-3-540-27810-8_41。[BFP+73] Manuel Blum、Robert W. Floyd、Vaughan R. Pratt、Ronald L. Rivest与Robert Endre Tarjan。选择算法的时间界。J. Comput. Syst. Sci.，7(4):448–461，1973年。doi:10.1016/S0022-0000(73)80033-9。[BLT25] Gerth Stølting Brodal、George Lagogiannis与Robert E. Tarjan。严格斐波那契堆。ACM 算法汇刊，21(2):15:1–15:18，2025年。 doi:10.1145/3707692。[Bro13] Gerth Stølting Brodal。优先队列综述。载于Andrej Brodnik、Alejandro López-Ortiz、Venkatesh Raman与Alfredo Viola编，《空间高效的数据结构、流与算法——纪念J. Ian Munro 66岁生日论文集》，《计算机科学讲义》第8066卷，第150–163页。施普林格，2013年。 doi:10.1007/978-3-642-40273-9_11。[Cha13] Timothy M. Chan。地震堆：斐波那契堆的一种简单替代。载于Andrej Brodnik、Alejandro López-Ortiz、Venkatesh Raman与Alfredo Viola编，《空间高效的数据结构、流与算法——纪念J. Ian Munro 66岁生日论文集》，《计算机科学讲义》第8066卷，第27–32页。施普林格，2013年。 doi:10.1007/978-3-642-40273-9_3。[CR18] Rezaul Alam Chowdhury与Vijaya Ramachandran。缓存无关缓冲堆与图中最短路径的缓存高效计算。ACM 算法汇刊，14(1):1:1–1:33，2018年。 doi:10.1145/3147172。[Elm10] Amr Elmasry。违例堆：一种松弛斐波那契类堆。Discret. Math. Algorithms Appl.，2(4):493–504，2010年。 doi:10.1142/S1793830910000838。[FR75] Robert W. Floyd与Ronald L. Rivest。选择算法的期望时间界。ACM 通讯，18(3):165–172，1975年。 doi:10.1145/360680.360691。[Fre99] Michael L. Fredman。论配对堆及相关数据结构的效率。J. ACM，46(4):473–501，1999年。 doi:10.1145/320211.320214。[FSST86] Michael L. Fredman、Robert Sedgewick、Daniel Dominic Sleator与Robert Endre Tarjan。配对堆：一种新型自调整堆。Algorithmica，1(1):111–129，1986年。 doi:10.1007/BF01840439。[FT87] Michael L. Fredman与Robert Endre Tarjan。斐波那契堆及其在改进的网络优化算法中的应用。《ACM杂志》，34(3):596–615，1987年7月。 doi:10.1145/28869.28874。[FW93] Michael L. Fredman与Dan E. Willard。用融合树超越信息论界。J. Comput. Syst. Sci.，47(3):424–436，1993年。 doi:10.1016/0022-0000(93)90040-4。

[HKTZ17] Thomas Dueholm Hansen、Haim Kaplan、Robert E. Tarjan 和 Uri Zwick。空心堆。《ACM 算法汇刊》，13(3):42:1–42:27，2017年。 doi:10.1145/3093240.[Hoa61] C. A. R. Hoare。算法

65 4 7 321 322, 1961 doi: ： 查找。

10.1145/366622.366647
《ACM 通讯》，( ): – 。.[HST09] Bernhard Haeupler、Siddhartha Sen 和 Robert Endre Tarjan。秩配对堆。载于 Amos Fiat 和 Peter Sanders 编，《算法——ESA 2009》，第17届欧洲年度研讨会，丹麦哥本哈根，2009年9月7日至9日，论文集，《计算机科学讲义》第5757卷，第659–670页。施普林格，2009年。doi:10.1007/978-3-642-04128-0 59.[IÖ14] John Iacono 和 Özgür Özkan。为何有些堆支持摊还时间为常数的减键操作，而有些则不支持。载于 Javier Esparza、Pierre Fraigniaud、Thore Husfeldt 和 Elias Koutsoupias 编，《自动机、语言与程序设计——第41届国际专题讨论会，ICALP 2014》，丹麦哥本哈根，2014年7月8日至11日，论文集，第一部分，《计算机科学讲义》第8572卷，第637–649页。施普林格，2014年。 doi:10.1007/978-3-662-43948-7_53.[KTZ14] Haim Kaplan、Robert Endre Tarjan 和 Uri Zwick。重新审视斐波那契堆。CoRR，abs/1407.5750，2014年。网址：

http://arxiv.org/abs/1407.5750， arXiv:1407.5750。[NP10] Gonzalo Navarro 和 Rodrigo Paredes。论排序、堆与最小生成树。Algorithmica，57:585–620，2010年。 doi:10.1007/s00453-010-9400-6.[NPPS11] Gonzalo Navarro、Rodrigo Paredes、Patricio V. Poblete 和 Peter Sanders。更强的 quickheap。国际计算机科学基础期刊，22:945–969，2011年。 doi:10.1142/S0129054111008507.[Pet05] Seth Pettie。迈向对配对堆的最终分析。载于第46届IEEE计算机科学基础年会（FOCS 2005），2005年10月23日至25日，美国北卡罗来纳州达勒姆，论文集，第174–183页。IEEE 计算机学会，2005年。doi:10.1109/SFCS.2005.75.[PN06] Rodrigo Paredes 和 Gonzalo Navarro。最优增量排序。载于2006年第八届算法工程与实验研讨会（ALENEX）论文集，第171–182页。工业与应用数学学会，2006年1月。 doi:10.1137/1.9781611972863.16.[RW25] Casper Moldrup Rysgaard 和 Sebastian Wild。惰性 B 树。载于计算机科学数学基础研讨会（MFCS），《莱布尼茨信息学论文集》第345卷，第87:1–87:19页。达格施图尔城堡——莱布尼茨信息学中心，2025年。 doi:10.4230/LIPICS.MFCS.2025.87.[ST23a] Corwin Sinnamon 和 Robert E. Tarjan。对多遍配对堆近乎紧确的分析。载于 Nikhil Bansal 和 Viswanath Nagarajan 编，《2023年ACM-SIAM离散算法年会论文集》，SODA 2023，意大利佛罗伦萨，1月 22 25, 2023, 535 548 2023 https://doi.org/10.1137/ - 第 –

1.9781611977554.ch23 doi:10.1137/1.9781611977554.CH23
页。SIAM，。网址：，。[ST23b] Corwin Sinnamon 和 Robert E. Tarjan。对 Slim 堆和 smooth 堆的紧确分析。载于 Nikhil Bansal 和 Viswanath Nagarajan 编，《2023年ACM-SIAM离散算法年会论文集》，SODA 2023，意大利佛罗伦萨，1月

22 25, 2023, 549 567 2023 https://doi.org/10.1137/1. - 第 – 页。

9781611977554.ch24 doi:10.1137/1.9781611977554.CH24
SIAM，。网址：，。

[SW20] Bryce Sandlund 和 Sebastian Wild。惰性搜索树。载于 Sandy Irani 编，第*61*届*IEEE*计算机科学基础年会，*FOCS 2020*，美国北卡罗来纳州达勒姆，*11*月 *16 19, 2020*, 704 715 2020 *doi:10.1109/* -第 – 页。*FOCS46700.2020.00071*
IEEE，。。[SZ22] Bryce Sandlund 和 Lingyi Zhang。可选择堆与最优惰性搜索树。载于*2022*年*ACM-SIAM*离散算法年会（*SODA*）论文集，第1962–1975页，2022年。 *doi:10.1137/1.9781611977073.78*.[Wil64] John William Joseph Williams。算法232：堆排序。《*ACM*通讯》，7(6):347–348，1964年。 *doi:10.1145/512274.512284*。

附录

A. LP 堆概念验证实现

为展示 LP 堆的简洁性，我们在下面提供了一个完全自包含的 Python 实现。这段代码旨在作为 LP 堆的详细规范，并针对可读性进行了调整，而非追求效率。

我们首先展示 LP 堆的主要代码，即类 `LazyHeap`；它使用双向链表和一种选择方法，后者将在下文提供。

```
class LazyHeap:
    def __init__(self):
        self.counter = 0
        self.pivots = []
        self.sets = [LinkedList()]

    def insert(self, value):
        node = Node((value, self.counter))
        self.counter += 1
        self._insert(node)
        return node
    def delete(self, node):
        self._remove(node)
        self._merge_sets()

    def extract_min(self):
        S = self.sets[0]
        if len(S) == 0:
            raise ValueError("Heap is empty")
        smallest_node = min(S.iter_nodes(),
                            key=lambda node: node.value)
        S.remove(smallest_node)
        if len(S) >= 2:
            median, first, second = self._partition_set(S)
            self.pivots = [median] + self.pivots
            self.sets = [first, second] + self.sets[1:]
        self._forget_pivots()
        return smallest_node.value[0]

    def decrease_key(self, node, new_value):
        assert new_value <= node.value[0]
        self._remove(node)
        node.value = (new_value, node.value[0])
        self._insert(node)
    def _insert(self, node):
        S = self._find_set(node.value)
        S.append(node)
    def _remove(self, node):
        S = self._find_set(node.value)
        S.remove(node)
```

```
    def _find_set(self, value):
        if len(self.sets) == 1 or value < self.pivots[0]:
            return self.sets[0]        low = 0
        high = len(self.pivots)
        while low + 1 < high:
            mid = (low + high) // 2
            if self.pivots[mid] <= value:
                low = mid              else:
                high = mid
        assert value >= self.pivots[low]
        return self.sets[low + 1]
    def _partition_set(self, S):
        median = select(S, -(-len(S) // 2))
        smaller = []       larger = []
        for node in S.iter_nodes():
            if node.value < median:
                smaller.append(node)           else:
                larger.append(node)
        first = LinkedList()       for node in smaller:
            first.append(node)       second = LinkedList()
        for node in larger:          second.append(node)
        return median, first, second
    def _forget_pivots(self):      new_pivots = []
        new_sets = [self.sets[0]]      outside = 0
        for pivot, S in zip(self.pivots, self.sets[1:]):
            if len(new_sets[-1]) == 0 or
                    outside > len(new_sets[-1]) + len(S):
                new_sets[-1].concat(S) // merge
            else:
                outside += len(new_sets[-1])
                new_pivots.append(pivot)
                new_sets.append(S)
        if len(new_sets[-1]) == 0 and len(new_sets) >= 2:
            new_pivots.pop()           new_sets.pop()
        self.pivots = new_pivots
        self.sets = new sets
```

上面的代码使用了一个循环闭合的双向链表；si

as p指针 /对已插入元素的引用 , we p提供简 一旦节点对象被使用单实现如下：

```
class Node:
    def __init__(self, value):
        self.value = value
```

```
        self.prev = None
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = Node(None)
        self.tail = Node(None)
        self.head.next = self.tail
        self.tail.prev = self.head
        self.size = 0

    def __len__(self):
        return self.size

    def append(self, node):
        self.tail.prev.next = node
        node.prev = self.tail.prev
        node.next = self.tail
        self.tail.prev = node
        self.size += 1

    def remove(self, node):
        node.prev.next = node.next
        node.next.prev = node.prev
        node.prev = None
        node.next = None
        self.size -= 1

    def concat(self, other):
        t = self.tail.prev
        h = other.head.next
        t.next = h
        h.prev = t
        self.tail = other.tail
        self.size += other.size

    def iter_nodes(self):
        at = self.head.next
        while at is not self.tail:
            yield at
            at = at.next

    def __iter__(self):
        for node in self.iter_nodes():
            yield node.value
```

为使内容完整自足，我们还给出了中位数的中位数线性时间确定性选择方法 [BFP+73] 的一个简单实现。请注意，该方法不是原地算法，应视为一种“教学性”实现。（实际应用中的实现肯定会更倾向于使用随机化快速选择变体。）

```
def select(iterable, k):
    def inner(arr, k):
        buckets = [sorted(arr[i:i+5]) for i in range(0, len(arr), 5)]
```

```
5           if len(buckets) == 1:
6               return buckets[0][k]
7
8           medians = [buck[len(buck) // 2] for buck in buckets]
9           pivot = inner(medians, len(medians) // 2)
10
11          smaller = []
12          larger = []
13
14          for value in arr:
15              if value < pivot:
16                  smaller.append(value)
17              else:
18                  larger.append(value)
19
20          if k < len(smaller):
21              return inner(smaller, k)
22          else:
23              return inner(larger, k - len(smaller))
24
25      return inner(list(iterable), k)
```