# Partition-based Simple Heaps

Gerth Stølting Brodal[1][0000−0001−9054−915X], John Iacono[2][0000−0001−8885−8172], Casper Moldrup Rysgaard[1][0000−0002−3989−123X], and Sebastian Wild[3,4][0000−0002−6061−9177]

[1] Aarhus University, Denmark {gerth,rysgaard}@cs.au.dk
[2] Université libre de Bruxelles, Ixelles, Belgium john.iacono@ulb.be
[3] University of Marburg, Germany wild@informatik.uni-marburg.de
[4] University of Liverpool, UK

**Abstract** We introduce a new family of priority-queue data structures: *partition-based simple heaps*. The structures consist of $\mathcal{O}(\lg n)$ doubly-linked lists; order is enforced among data in different lists, but the individual lists are unordered. Our structures have amortized $\mathcal{O}(\lg n)$ time delete-min and amortized $\mathcal{O}(\lg \lg n)$ time insert and decrease-key. The structures require nothing beyond binary search over $\mathcal{O}(\lg n)$ elements, as well as binary partitions and concatenations of linked lists in natural ways as the linked lists get too big or small. We present three different ways that these lists can be maintained in order to obtain the stated amortized running times.

## 1 Introduction

The *priority queue* is a fundamental comparison-based abstract data type used in numerous efficient algorithms, such as discrete event simulation, Dijkstra's shortest path algorithm and Prim's minimum spanning tree algorithm. A priority queue (a.k.a. *heap*) stores a set of elements and has to implement the following operations, where we assume each element has an associated key from a comparison-based universe:

- Insert($e$): Add an element $e$ to the the heap and return a pointer *ptr* to it.
- Delete-Min(): Remove and return the element in heap with smallest key.
- Decrease-Key($ptr, key$): Change the key of the element pointed to by *ptr* in the heap to a smaller key *key*.

A standard undergraduate computer-science curriculum discusses binary heaps [Wil64], which realize all operations in $\mathcal{O}(\lg n)$ time[5] when the data structure currently contains $n$ elements (as would a balanced binary search tree). However, much faster updates for priority queues are possible: The most efficient heaps such as (strict) Fibonacci heaps [FT87,BLT25] achieve *constant* (amortized) time for Insert and Decrease-Key and $\mathcal{O}(\lg n)$ time for Delete-Min.

[5] Here and throughout we let lg denote the binary logarithm.

While optimal in theory, these *efficient priority queues* have some downsides. First, their structure and analysis are less intuitive to teach and implement. Second, the constant factors in their running time and space usage are substantially higher than for binary heaps; not least because they inherently rely on pointer-based representations of trees, with the ensuing poor locality of reference.

The desire to simplify efficient priority queues and to make them more efficient in practice has resulted in a number of alternative data structures, which we review briefly in Section 1.1. They all retain somewhat complicated structures and invariants, are nontrivial to analyze, or require $\Omega(\lg n)$ time for updates.

In this paper, we present simple priority-queue implementations solely based on linear data structures, showing that simple quicksort-style partitioning of elements into unsorted sets suffices for amortized $\mathcal{O}(\lg \lg n)$ time INSERT and DECREASE-KEY operations and amortized $\mathcal{O}(\lg n)$ time DELETE-MIN operations – provided it is governed by a carefully chosen *"pivot-forgetting rule"* to keep the number of sets in $\mathcal{O}(\lg n)$. We present several such rules, demonstrating a rich design space. This brings us much closer to the efficient priority queues above, but with entirely elementary techniques and simple designs. As a case in point, for the arguably simplest rule ("lazy partition heaps"), we give a fully self-contained description including its amortized analysis suitable for teaching in undergraduate algorithms classes in Sections 2 and 3.

### 1.1 Previous Work

See [Bro13] for a full survey of priority queues. We begin the history of priority queues with fast DECREASE-KEY which begins with the Fibonacci heap [FT87]. They support constant-amortized time DECREASE-KEY and INSERT, and $\mathcal{O}(\lg n)$ amortized-time DELETE-MIN. Since then a variety of other heaps with fast DECREASE-KEY have been introduced:

- The pairing heap [FSST86], which is a self-adjusting structure modeled after splay trees, and which we do not know the asymptotic amortized running time of DECREASE-KEY. Only $\mathcal{O}\left(2^{\sqrt{\lg \lg n}}\right)$ [Pet05] and $\Omega(\lg \lg n)$ [Fre99] are known. A multipass variant has been shown to have an $\mathcal{O}(\lg \lg n \cdot \lg \lg \lg n)$ amortized-time DECREASE-KEY [ST23a].
- The strict Fibonacci heap, which has worst-case running times [BLT25].
- Chan's quake heaps, invented to be a simpler alternative to Fibonacci heap with $\mathcal{O}(1)$ amortized time DECREASE-KEY [Cha13].
- Rank-pairing heaps [HST09] which are another alternative with $\mathcal{O}(1)$ amortized time DECREASE-KEY, but where the structure is closer to the pairing heap.
- Violation heaps [Elm10], hollow heaps [HKTZ17], and Fibonacci heaps revisited [KTZ14] are other variants with with $\mathcal{O}(1)$ amortized-time DECREASE-KEY.
- Slim and smooth heaps [ST23b] have $\mathcal{O}(\lg \lg n)$ DECREASE-KEY, but are in the pointer-based model with constant indegree.

*Lower bounds.* There are two lower bounds for DECREASE-KEY in heaps, one due to Fredman [Fre99] and the other due to Iacono and Özkan [IÖ14]. Both of these bounds only apply to heap-based priority queues that follow two different particular models, each of which includes pairing heaps, and neither of which applies to our new structures. The Fredman lower bound focuses on the tradeoff between augmented data and the decrease-key time. The Iacono and Özkan bound focuses on structures that are pointer-based. Both of these results point to the fact that $\mathcal{O}(\lg \lg n)$ is arguably a natural time for heaps, but both only apply to (different) classes of generalized heaps which are quite specific for their adversary arguments to work.

All of the above points to $\mathcal{O}(\lg \lg n)$ as the right running time for DECREASE-KEY in a pointer model structure with constant indegree. Slim and smooth heaps achieve this, and pairing heaps are conjectured to. All of the other heaps require either RAM-model tricks or non-constant indegree. For example, Fibonacci heaps need random access into an array of size $\mathcal{O}(\lg n)$ in order to combine heaps with the same rank (which is an integer with logarithmic range). One could of course replace this array with a balanced binary search tree, but this would introduce an $\mathcal{O}(\log \log n)$ overhead. In [FT87], it is shown how this array can be removed, albeit by using nodes of linear indegree.

*Cache-oblivious inspiration.* We point to the works of Chowdhury and Ramachandran [CR18] and Brodal, Fagerberg, Meyer, and Zeh [BFMZ04] as sources of inspiration as they are devoid of treeology. Both of these, which were designed for the cache-oblivious model, use two sequences of arrays of increasing size; superficially it looks like two copies of our data structure where items are inserted into one set, percolate down, at some point move to the other set, and then percolate up. Being designed for the cache-oblivious model, the lists are stored in arrays.

*Lazy search trees.* Also related are lazy search trees [SW20,SZ22,RW25], a data structure that smoothly interpolates between priority queues and binary search trees; depending on what queries are used, INSERT can be as fast as in the best priority queues, but arbitrary sorted dictionary queries are supported, as well. Our structures can be seen as a one-sided instance of the lower two tiers of the original lazy search trees with slightly modified operations.

*Quickheaps.* Other related work takes a more application-focused view of making priority queue implementations fast in practice. While asymptotic performance matters, constant-factor overhead and locality of reference can, for all reasonable input sizes, dwarf the distinction between logarithmic and sublogarithmic complexities. Quickheaps [PN06,NP10] and their "stronger" refinements [NPPS11] keep data in a single array and conceptually apply successive quickselect invocations for queries, where past partitioning steps are remembered and skipped.

This works well if updates come in random order; for robustness against adversarial inputs, stronger quickheaps take inspiration from schemes for balancing binary search trees to ensure good amortized performance in quickheaps: either

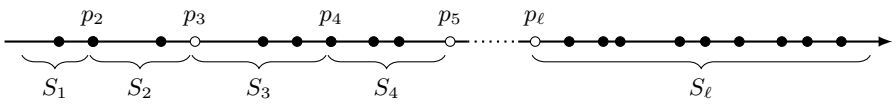


**Figure 1.** Illustration of our partition-based heap framework. The pivots $p_2 \ldots, p_\ell$ partition the $n$ elements in the heap into sets $\ell$ sets $S_1, S_2, \ldots, S_\ell$. Black dots are the $n$ elements in the heap. Pivots can be elements in the heap (black) or elements previously removed from the heap (white).

using randomization or using a form of weight-balanced binary search tree. Being designed for the much harder problem of keeping an entire binary search tree in shape, the resulting concatenation rules for quickheaps are complicated. We point out that not only the concatenation rule itself, but also the amortized analysis in [NPPS11] is substantially different from those in this paper. They also cite favorable performance in the external-memory model as a benefit of quickheaps (in any variant). Our heaps can also have these same favorable properties. Note that the authors of quickheaps excluded pointer-based operations, in particular DECREASE-KEY, from their consideration of quickheaps in external memory.

## 2 Partition-based Heap Framework

Here we describe the generic framework of partition-based heaps common to all three of our data structures in Sections 3–5. We assume that the elements have keys from a totally ordered universe and that elements are ordered with respect to their keys. If ever duplicate keys are inserted into a partition-based heap, standard techniques are used so that all ties are broken consistently.

Partition-based heaps maintain the elements currently stored in the heap as a partition of *sets* $S_1, S_2, \ldots S_\ell$, for some $\ell$ that is $\mathcal{O}(\lg n)$. The number of elements currently stored in the heap is $n = |S_1| + \cdots + |S_\ell|$. While the sets themselves are unordered, if $i < j$, all elements in $S_i$ are less than all elements of $S_j$

Each set $S_i$ is stored as a *doubly-linked list* as constant-time concatenation of sets is vital to all of our structures. For each set, the size $|S_i|$ is explicitly maintained. As we store all elements in linked lists, we support *stable pointers* to any element inserted.

In addition to the sets and their sizes, a partition-based heap stores *pivots* $p_2 \leq \cdots \leq p_\ell$ which are elements such that $S_i \subseteq [p_i, p_{i+1})$ for $i = 1, \ldots, \ell$; for notational convenience, assume $p_1$ and $p_{\ell+1}$ are pivots with keys $-\infty$ and $+\infty$, respectively. Pivots may be either elements currently or previously in the heap. See Figure 1 for an illustration.

To implement each priority queue operation all of our partition-based heaps have the following steps in common:

- INSERT($e$): Search for $e$ among the $\mathcal{O}(\lg n)$ pivots in $\mathcal{O}(\lg \lg n)$ time to find the set $S_i$ with $p_i \leq e < p_{i+1}$. Insert $e$ into $S_i$ and increment the size $|S_i|$. Return a pointer *ptr* to the new node containing $e$ in the linked list for $S_i$.

– DELETE-MIN(): Remove the smallest element from the first nonempty set $S_i$, and decrement the size $|S_i|$.
– DECREASE-KEY($ptr, key$): Let $e$ denote the element in the linked-list node $ptr$ points to. Remove this node from its linked list. Binary search on the pivots in $\mathcal{O}(\lg \lg n)$ to find the set $S_i$ that the node was removed from and decrement the size $|S_i|$. Decrease the key of $e$ to $key$, and re-insert the node as in INSERT, using a second search among the pivots to discover which set to insert it into, insert it at the end of the linked list for the set, and increment the set size.

These descriptions are not enough to maintain the crucial invariant that $\ell = \mathcal{O}(\log n)$. This is where the three heaps in Sections 3–5 differ; each has its own invariants towards this goal, and each augments the basic implementations above with restructuring operations which maintain said invariants. All structures use the technique of combining sets in constant time by concatenating linked-lists and splitting sets in linear time using a linear time selection algorithm to maintain their invariants. Further, all structures use amortized analysis to argue for the running times.

At a very high level, the philosophy of each structure is at follows: In *lazy partition heaps* (LP heaps) in Section 3, the sets are allowed to become arbitrarily large, and there is no lower bound on the number of sets. A bound on the size of a set compared to the sum of all previous sets is maintained. In *Fibonacci heaps: the next generation* (FH:TNG) in Section 4 and the heaps in Section 5, sets are allowed to be empty and there are strict upper bounds on their size. In FH:TNG, there are lower bounds on non-empty set sizes, where both the upper and lower bounds are given by the Fibonacci numbers, which naturally is amiable to concatenating and splitting while maintaining the size invariants. The heaps in Section 5 do not maintain a lower bound on the set sizes which results in simpler restructuring algorithms.

## 3 Lazy Partition Heaps

To keep the number of sets $\ell$ in $\mathcal{O}(\lg n)$ at all times, *lazy partition heaps* (LP heaps) use the following pivot-forgetting rule, which is adapted from lazy search trees [SW20]:

> *A pivot $p$ shall be abandoned if the two sets that it separates together contain fewer elements than there are* smaller *elements in the heap (Figure 2).*

In LP heaps, every DELETE-MIN operation partitions $S_1$ around its median. Since no stringent bound on $|S_1|$ is guaranteed, we need to amortize the occasional high costs for DELETE-MIN. For that, sets $S_j$, $j = 1, \ldots, \ell$, accumulate potential when they have grown "too large", namely $\varphi_j \coloneqq \max\{0, |S_j| - \mathbf{s}(S_j)\}$, where $\mathbf{s}(S_j) \coloneqq |S_1| + \cdots + |S_{j-1}|$ denotes the number of elements smaller than all elements in $S_j$ (their "safety buffer" from the minimum).

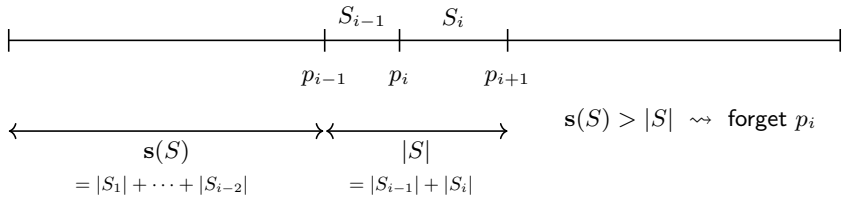


**Figure 2.** The pivot-forgetting rule for sets in LP heaps. Here, the lists for sets $S_{i-1}$ and $S_i$ would be concatenated (thus forgetting $p_i$) since there are more elements smaller than $p_{i-1}$ than between $p_{i-1}$ and $p_{i+1}$.

This in particular includes the set $S_1$, whose potential $\varphi_1 = |S_1|$ can thus pay for the scanning and partitioning of $S_1$ upon the next DELETE-MIN.

Apart from the sets, we store a (static) sorted array $T$ of the current sets $S_1, S_2, \ldots, S_\ell$ (in this order). (Whenever the collection of sets changes because of a DELETE-MIN, we can afford to rebuild $T$ from scratch.) For simplicity of the pseudocode, we let each set $S_i$ store the corresponding pivot $p_i$ as $S_i.p$. We point out that $\mathbf{s}(S_j)$ is only needed during the pivot-forgetting phase and is computed on demand there.

### 3.1 Operations

Pseudo code for the operations on a LP heap is shown in Figure 3. Throughout the lifetime of an LP heap, we maintain the following invariant.

**Invariant (S):** The number of sets is at most $\ell \leq 2 \lg n + 1$.

If the invariant is at risk of violation—which is only possible upon DELETE-MIN—we apply the following concatenation rule (cf. Figure 2). Recall that $\mathbf{s}(S_j) \coloneqq |S_1| + \cdots + |S_{j-1}|$ is the number of *smaller elements* (smaller than $p_j$, that is):

**Concatenation Rule (C):** If $|S_j| + |S_{j+1}| < \mathbf{s}(S_j)$, concatenate $S_j$ and $S_{j+1}$.

Concatenation here simply means we concatenate the linked lists of elements for the two sets and "forgets" the pivot formerly separating the two sets. This operation takes constant time. FORGET-PIVOTS computes $\mathbf{s}(S_j)$ and applies rule (C) to $\mathcal{S}$ successively. Clearly, FORGET-PIVOTS uses $\mathcal{O}(\ell)$ time if the (initial) number of sets is $\ell$. Note that FORGET-PIVOTS also removes all empty sets present, thereby ensuring that $|S_j| \geq 1$ for all $1 \leq j \leq \ell$. After FORGET-PIVOTS has concatenated all pairs of sets that satisfy (C), we obtain a bound on the number of sets from the following lemma.

**Lemma 1 ((C) implies (S)).** *Suppose after applying rule (C), we have sets $S_1, \ldots, S_\ell$, with $|S_j| \geq 1$ for all $1 \leq j \leq \ell$. Then $\ell \leq 2 \lg n + 1$, for $n$ the total size of the sets.*

```
LP-Heap-Insert(e)
1  ptr := New-Node(e) // obtain pointer to element
2  S := T.Find(e.key) // binary search the pivots
3  S.Append(ptr) // insert element
4  return ptr

LP-Heap-Decrease-Key(ptr, key)
1  e := *ptr // obtain element from pointer
2  S := T.Find(e.key)
3  S.Remove(ptr) // remove element from current set
4  e.key := key // update key
5  S := T.Find(e.key)
6  S.Append(ptr) // insert element in new set

LP-Heap-Delete-Min()
1  min := S_1.Delete-Min() // scans S_1
2  p := Median(S_1) // (larger) median (deterministic linear-time selection [BFP+73])
3  S̄_1, S̄_2 := Partition(S_1, p)
4  S̄_2.p := p
5  S := [S̄_1, S̄_2, S_2, ..., S_ℓ] // doubly-linked list of sets
6  Forget-Pivots(S) // modifies S
7  Rebuild T from S
8  return min

Forget-Pivots(S)
   // Assume S = [S_1, ..., S_ℓ]
1  S.Remove-All-Empty-Sets()
2  A := S_1
3  s(A) := 0
4  while S.Has-Next(A)
5        B := S.Next(A)
6        if |A| + |B| < s(A)
7              Concatenate B into A
8              S.Remove(B)
9        else
10             s(B) := s(A) + |A|
11             A := B
12       end if
13 end while
```

**Figure 3.** LP heaps implementation.

*Proof.* We first show that prefixes of sets grow exponentially: for all $1 \le j \le \ell$, we have $\Sigma(j) \coloneqq |S_1| + \cdots + |S_j| \ge 2^{(j-1)/2}$ $(*)$.

The proof is by induction on $j$. The claims holds for $1 \le j \le 2$, since $|S_i| \ge 1$ for $1 \le i \le \ell$. In the inductive step for $j \ge 3$, the inductive hypothesis implies $\Sigma(j-2) \ge 2^{(j-3)/2}$. By concatenation rule (C), we have $|S_{j-1}| + |S_j| \ge \mathbf{s}(S_{j-1})$, and by definition $\mathbf{s}(S_{j-1}) = \Sigma(j-2)$. Therefore $\Sigma(j) = \Sigma(j-2) + |S_{j-1}| + |S_j| \ge 2 \cdot \Sigma(j-2) \ge 2 \cdot 2^{(j-3)/2} = 2^{(j-1)/2}$, which proves $(*)$.

Since $(*)$ holds for any prefix of sets, it does so in particular for $j = \ell$. Then, $\Sigma(\ell) = n$ (all elements), and we get $\ell \le 2 \lg n + 1$ as claimed. □

It follows from Lemma 1 that Forget-Pivots ensures (S). This completes the description of the data structure.

### 3.2 Potential

For analyzing the amortized cost of operations, we use the following potential:

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

(The sum is over all sets currently in the data structure.) Here $\beta \ge 2$ is a fixed scaling parameter (chosen depending on the constant factor in the linear cost of partitioning).

### 3.3 Amortized Cost of Insert

Upon an Insert operation, the set $S_j$ containing $e$ is located using binary search in $T$ in $\mathcal{O}(\lg \lg n)$ time. Adding $e$ to the linked list of $S_j$ takes constant time.

In $\Phi$, $\mathbf{s}(S_{j'})$ increases by 1 for all $j' > j$; this never increases the value of $\Phi$. The size of $S_j$ grows by 1 which increases $\Phi$ by at most $\beta = \mathcal{O}(1)$. Insert thus costs $\mathcal{O}(\lg \lg n)$ (both amortized and worst case).

### 3.4 Amortized Cost of Decrease-Key

When Decrease-Key$(ptr, key)$ is performed, the element $e$ at pointer $ptr$ has its key decreased to $key$. Re-inserting $e$ costs $\mathcal{O}(\lg \lg n)$ (both amortized and worst case). This changes the size of at most two sets, with one increasing and the other decreasing, which increases the potential by at most $\beta$. It may also change $\mathbf{s}(S_{j'})$ for some sets $S_{j'}$, but since we move $e$ from $S_i$ to an $S_j$ with $1 \le j \le i$ when we make the key of the element smaller, $\mathbf{s}(S_{j'})$ can only *increase*, which does not increase the potential. In total, Decrease-Key uses $\mathcal{O}(\lg \lg n + \beta) = \mathcal{O}(\lg \lg n)$ time (both amortized and worst case).

Decrease-Key may leave a set $S_i$ empty, where $2 \le i \le \ell$, however, these may be removed upon a later Forget-Pivots, without an increase in time or potential, as the number of sets remains bounded, maintaining invariant (S).

### 3.5 Amortized Cost of Delete-Min

A DELETE-MIN scans $S_1$ to find the minimum element $e$ and removes it; moreover, we split $S_1$ by partitioning it around its median, resulting in two new sets $\overline{S}_1$ and $\overline{S}_2$. For the partition, we use deterministic selection. Scanning and partitioning $S_1$ has overall actual cost of $\mathcal{O}(|S_1|)$.

Finally, we establish invariant (S) by calling FORGET-PIVOTS. By the invariant (S) on the number of sets, and since partitioning set $S_1$ creates two new sets, FORGET-PIVOTS uses $\mathcal{O}\big(\lg n\big)$ actual time.

It remains to analyze the change in potential. Note first that the concatenation rule (C) is chosen precisely so that when concatenating $S_{j+1}$ into $S_j$, the contributions $\varphi_{j+1}$ and $\varphi_j$, both before and after the concatenation, are all zero, so concatenating does not change the potential at all. More specifically, assume that $|S_j| + |S_{j+1}| < \mathbf{s}(S_j)$, and a concatenation of $S_j$ and $S_{j+1}$ is performed to obtain the new set $\overline{S}_j$, by the concatenation rule (C). Then the potential is decreasing by $\varphi_j$ and $\varphi_{j+1}$ and increasing by the new potential $\overline{\varphi}_j$. By definition these potentials are

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

The change in potential $\Delta\Phi$ is therefore 0 upon a concatenation.

It remains to analyze the influence on the potential of partitioning the set $S_1$ into the new sets $\overline{S}_1$ and $\overline{S}_2$. The situation after partitioning $S_1$ is illustrated in Figure 4. For the set $S_1$, we lose $\varphi_1 = \beta|S_1|$ and gain $\overline{\varphi}_1 + \overline{\varphi}_2 = \beta\big|\overline{S}_1\big|$ from $\overline{S}_1$ and $\overline{S}_2$, as we choose the larger median s. t. $\mathbf{s}(\overline{S}_2) = \big|\overline{S}_1\big| \geq \big|\overline{S}_2\big|$, i.e., $\overline{\varphi}_2 = 0$. All other sets lose one smaller element, so we gain $\leq \ell \cdot \beta$ in potential. By construction, $\big|\overline{S}_1\big| \leq \frac{1}{2}|S_1|$ (recall that we delete an element from $S_1$ before partitioning into $\overline{S}_1$ and $\overline{S}_2$), so $\Delta\Phi \leq -\frac{\beta}{2}|S_1| + \ell\beta$. For a sufficiently large $\beta \geq 2$, the total amortized cost is thus

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

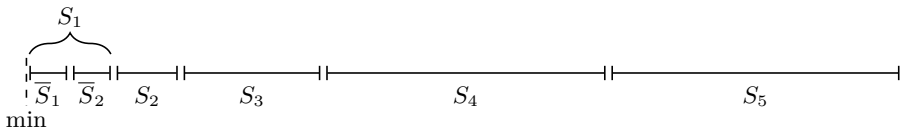


**Figure 4.** Sketch of the sets of an LP heap and the partition of $S_1$ into $\overline{S}_1$ and $\overline{S}_2$ during DELETE-MIN.

### 3.6 Extensions

Here we discuss a few further properties of LP heaps; they are not needed to understand (or teach) LP heaps.

**Further Operations** We can support LP-Heap-Find-Min in $\mathcal{O}(1)$ time, by maintaining the minimum upon LP-Heap-Insert and LP-Heap-Decrease-Key in constant time, and by scanning $\overline{S}_1$ after a LP-Heap-Delete-Min, which maintains the amortized time. Note that $|S_1|$ can be arbitary large, i.e., we cannot afford scanning $S_1$ for each LP-Heap-Find-Min operation.

We can also support an LP-Heap-*Increase*-Key operation or a general LP-Heap-Delete($ptr$) operation, at total amortized cost $\mathcal{O}(\beta \lg n + \lg \lg n) = \mathcal{O}(\lg n)$. For both operations $S_1$ might become empty, and further for general LP-Heap-Delete($ptr$), $n$ decreases which may violate invariant (S), and therefore Forget-Pivots is needed to re-establish (S). This also adds $\mathcal{O}(\lg n)$ to the cost.

An operation LP-Heap-Build from a given collection of $n$ elements can be supported in $\mathcal{O}(n)$ time. For that, we simply declare all elements in $S_1$. The actual cost (if elements are given in a linked list) can even be constant, however we have to pay for the increase in potential of $\beta n$, so the total amortized cost is $\mathcal{O}(n)$.

The LP-Heap-Meld operation of taking the union of two heaps is not easily supported efficiently. One can trivially create a new LP heap from the concatenation of the two input heaps using LP-Heap-Build, at linear amortized cost in the total number of elements.

**Partitioning** We can replace the deterministic linear-time selection [BFP+73] by randomized selection [Hoa61,FR75], leading to the same expected amortized time.

More importantly, we can also simply run a *single round* of (random) partitioning: Upon the partition, the potential decreases by $\min\{|\overline{S}_1|, |\overline{S}_2|\}$. By choosing the pivot not as the median, but at random between the elements of $S_1$, the decrease is therefore *expected* linear in $|S_1|$, and the amortized time of Delete-Min remains *expected* $\mathcal{O}(\lg n)$.

**LP Quickheaps** The *quickheaps layout* (described in more detail in Section 1.1) offers an enticing alternative to the linked lists for each set: At the expense of more data movement during insertions and deletions, we store all sets contiguously in a single array, with the pivots separating them. The only extra space is a static array $T$ with the $O(\lg n)$ pivot positions.

Concatenations of adjacent sets can still be done in constant time (by simply demoting a pivot to an ordinary element), and partitioning can now be done in place, which improves practical overhead in running time. While we can also still binary search the pivots to *find* a set to insert an element into, insertions or deletions must now be implemented with up to one swap per set, leading to overall $\Theta(\lg n)$ running time for all operations.

**Pointer-Machine Version** LP heaps already stay within the pointer-machine model of computation with bounded indegree nodes except for the index $T$. By replacing the sorted array and binary search by any balanced binary search tree, then LP heaps is entirely in the model.

## 4 Fibonacci Heaps: The Next Generation

The next generation Fibonacci heap (FH:TNG)[6] follows our framework for partition-based simple heaps with but a few distinctive features: Sets can either be present or absent. If a set is present, its size is required to be between the $i$th Fibonacci number, denoted as $F_i$ ($F_1 \coloneqq F_2 \coloneqq 1$; $F_i \coloneqq F_{i-1} + F_{i-2}$), and the $i+3$rd, $F_{i+3}$, with the one exception that the first set, $S_3$, has no minimum size. Finally we require that there are at most eight empty sets in a row (*the consecutive empty* invariant) and at most two non-empty sets in a row (*the consecutive nonempty* invariant) to allow room for splits and concatenations.

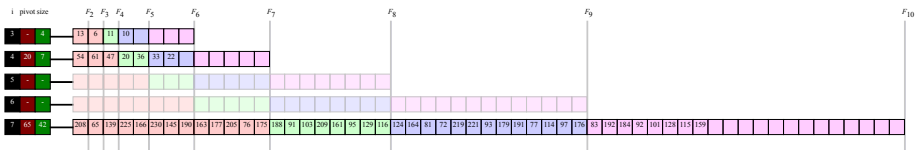


**Figure 5.** Illustration of a next generation Fibonacci heap. While each set is stored as a linked list, this illustration uses colors to show the relationship between each list's size, the Fibonacci numbers, and the size potential function.

The implementation of the operations follows our basic framework. The only difference is what to do when the invariants are violated. We restore them as follows using *invariant-restoring operations*: If a set $S_i$ has $F_{i+3}$ elements we call it *full* and there are two options: If the set below, $S_{i+1}$ is empty, $S_i$ is simply moved to $S_{i+1}$. We call this a *overflow-down operation*. If $S_{i+1}$ is not empty, $S_{i+2}$ must be empty by the consecutive nonempty invariant. We concatenate all $F_{i+3}$ elements of $S_i$ to the set $S_{i+1}$ and then remove the $F_{i+3}$ largest elements of the resultant set using linear-time selection and place them in $S_{i+2}$; this leaves the size of $S_{i+1}$ unchanged but the contents will certainly change. We call this an *overflow-thru operation*. If a set $S_i$ has reached its minimum size $F_i$, it is *underfull*, and we execute either an *underflow-up operation* or an *underflow-thru operation* depending on whether $S_{i-1}$ is empty. These are symmetric to the two overflow operations. If the consecutive-nonempty invariant is violated, there are three nonempty sets in a row with an empty list below. We concatenate the top two of them and place the resultant set in the empty slot below, we call this a *merge-down operation*.

[6] The name is meant to reflect that fact that in the original Fibonacci heaps and the the structure here, Fibonacci numbers play crucial, and entirely different, roles.

If the consecutive-empty invariant is violated, there are nine empty sets in a row with a non-empty set below. We take the nonempty set below and split using linear-time selection into two set slots immediately above. We call this a *split-up*, and this takes linear time in the sets involved. Merge-down takes constant time as it is combining two linked lists, whereas the split-up takes linear time in the size of the sets to perform a selection and traverse the set to split it in two.

The asymmetry of the linear-time split-up with the constant time merge-down is crucial and is deeply connected to the fact that we allow keys to be decreased but not increased! The intuition is that repeated Decrease-Key operations can force inexpensive merge-downs to be performed as everyone is slowly moved to bigger sets. For example, this can happen by repeatedly Decrease-Key-ing the maximum item to make it the minimum. However, split-up operations are only triggered by the more expensive Delete-Min operations and thus their more expensive cost can be paid for.

Here we present the potential function used in the analysis, leaving the straightforward but detailed proofs to the full version. The potential of the data structure is the sum of three components, each of which is defined for each set in the structure:

**Nonempty potential.** Define $\varphi_i^{\text{nonempty}}$ to be 1 if $S_i$ is nonempty and 0 if $S_i$ is empty. This is used to pay for the merge-down operation where the number of nonempty sets decreases by one at unit cost.

**Size potential.** Gives potential as $S_i$ nears being full ($F_{i+3}$ elements) or underfull ($F_i$ elements) and 0 potential when its size is in the middle of its allowable range (from $F_{i+1}$ to $F_{i+2}$ elements). This is the classic potential function used in array doubling and is used to pay for the underflow and overflow operations. Figure 5 is color-coded to highlight these three ranges.

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

**Up potential.** This gives potential to nonempty sets that have few items above. This is used to pay for the split-up operation. The intuition is that this is a mechanism that allows the Delete-Min operation to pay unit cost to each of $\Theta(\lg n)$ sets, which the sets save up in the case most items above have been removed and a split-up needs to happen. Let $\uparrow_i$ be defined to be $\sum_{j=1}^{i-1} |S_j|$. Define the up potential of set $S_i$, $\varphi_i^{\uparrow}$, as

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

## 5 Heaps with Exponential Upper Bounded Sets

In this section we consider another variant of a partition-based simple heap consisting of sets $S_1, \ldots, S_\ell$ and pivots $p_2 \leq \cdots \leq p_\ell$. Again, the time obtained for Delete-Min is amortized $\mathcal{O}(\lg n)$ and for Insert and Decrease-Key amortized $\mathcal{O}(\lg \lg n)$. In this variant we have exponential increasing upper bounds on the sizes of the sets, without any lower bounds, and allowing an arbitrary number of the sets to be empty. This is in contrast to the stronger structural invariants of the FH:TNG in Section 4.

**Invariant:** $|S_i| < 3 \cdot 2^i$ for $1 \leq i \leq \ell$.

The heap operations are implemented as follows:

Insert($e$) Binary search over the pivots to find $i$ where $p_i \leq e < p_{i+1}$, and append $e$ to $S_i$. If $|S_i| = 3 \cdot 2^i$, perform a recursive *push* of $S_i$ to $S_{i+1}$ (see below).

Delete-Min() If $S_1$ is empty, first recursively *pull* elements into $S_1$ from $S_2$ (see below). Delete and return the minimum from $S_1$. If $\ell > 1 + \lg n$, let $S_{\ell-1}$ be the concatenation of $S_{\ell-1}$ and $S_\ell$, and discard $S_\ell$ and $p_\ell$ before returning, i.e., $\ell$ decreases by one (this ensures $\ell \leq 1 + \lg n$).

Decrease-Key(*ptr*, *key*) Delete the element $e$ pointed to by *ptr* from its current set $S_j$ (to decrement the size $|S_j|$ we find $j$ by a binary search over the pivots). Change the key of $e$ to *key*, and perform a binary search over the pivots to find the set $S_i$ where to add $e$, and append $e$ to $S_i$. If $|S_i| = 3 \cdot 2^i$, perform a recursive *push* of $S_i$ to $S_{i+1}$.

*Push.* For Insert and Decrease-Key, the set $S_i$ receiving one more element might get size $|S_i| = 3 \cdot 2^i$. We *push* all $3 \cdot 2^i$ elements of $S_i$ to $S_{i+1}$, and set $S_i = \emptyset$. In general, a recursive push to $S_j$ of a set of elements $X$ always satisfies $2^{j-1} \leq |X| \leq 3 \cdot 2^{j-1}$. If $|S_j| < 2^j$ before the push (i.e., $S_j$ is too small to be pushed to $S_{j+1}$), $X$ is concatenated to $S_j$ in constant time, such that $|S_j| < 5 \cdot 2^{j-1}$, and the push terminates. Otherwise, the set $X$ becomes $S_j$ and the old $S_j$ is recursively pushed to $S_{j+1}$. The pivots need to be updated accordingly, and when $j = \ell + 1$ a new set $S_{\ell+1} = X$ is created, i.e., $\ell$ increases by one.

*Pull.* The recursive *pull* of elements into $S_1$, or more generally into $S_i$ when $S_i$ is empty, is performed as follows: If $S_{i+1}$ is empty, we first recursively pull elements into $S_{i+1}$. If $|S_{i+1}| \leq 2^{i-1}$, we swap the roles of $S_{i+1}$ and the empty $S_i$, i.e., in constant time move all elements from $S_{i+1}$ to $S_i$. Otherwise, we select and delete the $2^{i-1}$ smallest elements from $S_{i+1}$ in time $\mathcal{O}(2^i)$ and let $S_i$ be these (and update $p_{i+1}$; here it is crucial that elements are distinct to ensure $\max S_i < p_{i+1} = \min S_{i+1}$).

### 5.1 Analysis

To prove the amortized performance of this structure we apply the following potential function $\Phi = \varphi^{\text{insert}} + \varphi^{\text{push}} + \varphi^{\text{pull}}$, where

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

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

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

The basic idea is to let each set $S_i$ have a potential. A big set will have a push potential $\varphi_i^{\text{push}}$ that releases one unit of potential when the set is moved to $S_{i+1}$. An insertion into $S_i$ can trigger a push of $S_i$. The saved up insertion potential $\varphi_i^{\text{insert}}$ will be released by the push of $S_i$ to cover the increase in pull potential $\varphi_i^{\text{pull}}$. Finally, pull potential is intuitively associated with each "empty slot" among the first $2^{i-1}$ "slots" in $S_i$, with a potential to cover the cost to be recursively filled from each of the subsequent sets $S_{i+1}, \ldots, S_\ell$. We refer the reader to the full version of the paper for a detailed amortized analysis.

## 6 Conclusion

We presented three priority queue implementation, following a simple partition-based heap framework that uses elementary data structures and algorithms. For the simplest design, the lazy partition heap, we give a fully self-contained and complete description and analysis.

Replacing the binary search on an array by a binary search tree, the implementation works in the pointer-machine model and requires only constant indegree on nodes, and can hence be made persistent efficiently.

Due to its simplicity and linear structure, our implementation appears amenable to adaptations for external memory, but supporting pointer-based decrease-key operations becomes more complicated (as noted by [NPPS11]). We reserve this for future work.

Using our rule as the concatenation rule in quickheaps promises to yield a highly practical and robust priority queue implementation, combining the best of previously known quickheaps variants. A detailed empirical evaluation is planned for future work.

The astute reader may have noticed that Insert and Decrease-Key take amortized constant time, except for the binary searches over the $\mathcal{O}(\lg n)$ pivots. By using a non-pointer-machine model data structure such as a fusion tree [FW93] to store the pivots, the running time can be reduced to $\mathcal{O}(\lg_w \lg n)$, which under the standard assumption that $w = \Omega(\log n)$ results in $\mathcal{O}(1)$ amortized running times for these operations.

**Acknowledgments.** We thank our reviewers for their helpful comments. The second author's work is supported by the Fonds de la Recherche Scientifique — FNRS. The last author is partly supported by EPSRC grant EP/X039447/1.

## References

BFMZ04. Gerth Stølting Brodal, Rolf Fagerberg, Ulrich Meyer, and Norbert Zeh. Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. In Torben Hagerup and Jyrki Katajainen, editors, *Algorithm Theory - SWAT 2004, 9th Scandinavian Workshop on Algorithm Theory, Humlebaek, Denmark, July 8-10, 2004, Proceedings*, volume 3111 of *Lecture Notes in Computer Science*, pages 480–492. Springer, 2004. doi:10.1007/978-3-540-27810-8_41.

BFP+73. Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973. doi:10.1016/S0022-0000(73)80033-9.

BLT25. Gerth Stølting Brodal, George Lagogiannis, and Robert E. Tarjan. Strict Fibonacci heaps. *ACM Trans. Algorithms*, 21(2):15:1–15:18, 2025. doi:10.1145/3707692.

Bro13. Gerth Stølting Brodal. A survey on priority queues. In Andrej Brodnik, Alejandro López-Ortiz, Venkatesh Raman, and Alfredo Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, volume 8066 of *Lecture Notes in Computer Science*, pages 150–163. Springer, 2013. doi:10.1007/978-3-642-40273-9_11.

Cha13. Timothy M. Chan. Quake heaps: A simple alternative to Fibonacci heaps. In Andrej Brodnik, Alejandro López-Ortiz, Venkatesh Raman, and Alfredo Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, volume 8066 of *Lecture Notes in Computer Science*, pages 27–32. Springer, 2013. doi:10.1007/978-3-642-40273-9_3.

CR18. Rezaul Alam Chowdhury and Vijaya Ramachandran. Cache-oblivious buffer heap and cache-efficient computation of shortest paths in graphs. *ACM Trans. Algorithms*, 14(1):1:1–1:33, 2018. doi:10.1145/3147172.

Elm10. Amr Elmasry. The violation heap: a relaxed Fibonacci-like heap. *Discret. Math. Algorithms Appl.*, 2(4):493–504, 2010. doi:10.1142/S1793830910000838.

FR75. Robert W. Floyd and Ronald L. Rivest. Expected time bounds for selection. *Commun. ACM*, 18(3):165–172, 1975. doi:10.1145/360680.360691.

Fre99. Michael L. Fredman. On the efficiency of pairing heaps and related data structures. *J. ACM*, 46(4):473–501, 1999. doi:10.1145/320211.320214.

FSST86. Michael L. Fredman, Robert Sedgewick, Daniel Dominic Sleator, and Robert Endre Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986. doi:10.1007/BF01840439.

FT87. Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, July 1987. doi:10.1145/28869.28874.

FW93. Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993. doi:10.1016/0022-0000(93)90040-4.

HKTZ17. Thomas Dueholm Hansen, Haim Kaplan, Robert E. Tarjan, and Uri Zwick. Hollow heaps. *ACM Trans. Algorithms*, 13(3):42:1–42:27, 2017. doi:10.1145/3093240.

Hoa61. C. A. R. Hoare. Algorithm 65: find. *Commun. ACM*, 4(7):321–322, 1961. doi:10.1145/366622.366647.

HST09. Bernhard Haeupler, Siddhartha Sen, and Robert Endre Tarjan. Rank-pairing heaps. In Amos Fiat and Peter Sanders, editors, *Algorithms - ESA 2009, 17th Annual European Symposium, Copenhagen, Denmark, September 7-9, 2009. Proceedings*, volume 5757 of *Lecture Notes in Computer Science*, pages 659–670. Springer, 2009. doi:10.1007/978-3-642-04128-0_59.

IÖ14. John Iacono and Özgür Özkan. Why some heaps support constant-amortized-time decrease-key operations, and others do not. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I*, volume 8572 of *Lecture Notes in Computer Science*, pages 637–649. Springer, 2014. doi:10.1007/978-3-662-43948-7_53.

KTZ14. Haim Kaplan, Robert Endre Tarjan, and Uri Zwick. Fibonacci heaps revisited. *CoRR*, abs/1407.5750, 2014. URL: http://arxiv.org/abs/1407.5750, arXiv:1407.5750.

NP10. Gonzalo Navarro and Rodrigo Paredes. On sorting, heaps, and minimum spanning trees. *Algorithmica*, 57:585–620, 2010. doi:10.1007/s00453-010-9400-6.

NPPS11. Gonzalo Navarro, Rodrigo Paredes, Patricio V. Poblete, and Peter Sanders. Stronger quickheaps. *Int. J. Found. Comput. Sci.*, 22:945–969, 2011. doi:10.1142/S0129054111008507.

Pet05. Seth Pettie. Towards a final analysis of pairing heaps. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), 23-25 October 2005, Pittsburgh, PA, USA, Proceedings*, pages 174–183. IEEE Computer Society, 2005. doi:10.1109/SFCS.2005.75.

PN06. Rodrigo Paredes and Gonzalo Navarro. Optimal incremental sorting. In *2006 Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX)*, page 171–182. Society for Industrial and Applied Mathematics, January 2006. doi:10.1137/1.9781611972863.16.

RW25. Casper Moldrup Rysgaard and Sebastian Wild. Lazy B-trees. In *Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 345 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 87:1–87:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPICS.MFCS.2025.87.

ST23a. Corwin Sinnamon and Robert E. Tarjan. A nearly-tight analysis of multipass pairing heaps. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, pages 535–548. SIAM, 2023. URL: https://doi.org/10.1137/1.9781611977554.ch23, doi:10.1137/1.9781611977554.CH23.

ST23b. Corwin Sinnamon and Robert E. Tarjan. A tight analysis of slim heaps and smooth heaps. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, pages 549–567. SIAM, 2023. URL: https://doi.org/10.1137/1.9781611977554.ch24, doi:10.1137/1.9781611977554.CH24.

SW20. Bryce Sandlund and Sebastian Wild. Lazy search trees. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 704–715. IEEE, 2020. doi:10.1109/FOCS46700.2020.00071.

SZ22. Bryce Sandlund and Lingyi Zhang. Selectable heaps and optimal lazy search trees. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1962–1975, 2022. doi:10.1137/1.9781611977073.78.

Wil64. John William Joseph Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964. doi:10.1145/512274.512284.