Soft heaps: an intuitive overview

Matthew Millican Alexandros Hollender

1 Introduction

1.1 Traditional priority queues and their limitations

The *priority queue* is a fundamental abstract data structure representing a dynamic ordered set from which a user can rapidly remove minimum elements. Priority queues enjoy widespread application in a variety of fields ranging from sequence sorting to data compression, but they are studied perhaps most actively in the context of graph algorithms, where they feature prominently in a number of procedures designed to compute shortest paths and minimum spanning trees over weighted graphs.

A typical implementation of a priority queue can efficiently support most of the following operations:

`make-heap`(x): Create a new priority queue representing the set $\{x\}$.

`insert`(P, x): Insert element x into queue P.

`extract-min`(P): Locate the element of smallest priority in P, remove it from the data structure, and return its value.

`decrease-key`(P, x, k): Given access to the representation of element x in queue P, assign x a new, smaller priority k.

`meld`(P, Q): Destroy priority queues P and Q to return a new queue R representing the set $P \cup Q$.

Prim's algorithm, a classic approach to computing minimum spanning trees on undirected graphs, relies heavily on the use of several of these operations. Given a graph of n nodes and m edges, the algorithm must `insert` every node into a priority queue, call `decrease-key` at most once for each edge, and `extract-min` every node that was originally inserted. Hence, given a priority queue implementation with complexities $f(n)$, $g(n)$, and $h(n)$ for `insert`, `decrease-key`, and `extract-min`, the algorithm has time complexity

$$O\left(nf(n) + mg(n) + nh(n)\right).$$

An ideal priority queue would support these three operations in $O(1)$ time, allowing for linear-time computation of minimum spanning trees. However, achieving constant-time performance on all priority queue operations is provably impossible. A true priority queue that operated within these bounds would be able

to execute a comparison-based sort of any set in worst-case $O(n)$ time by `insert`-ing all elements of the set and then retrieving them in order using `extract-min`. Since any comparison-based sort requires $\Omega(n \log n)$ time in the worst case [1], the desired performance bounds cannot be achieved.

Hence we face a dilemma, arising from the fact that any true priority queue stores the sorted order of its elements in a retrievable form. On one hand, this is an important feature to ensure the correctness of several algorithms that use the data structure. On the other hand, it slows down solutions to problems unrelated to sorting, which is an obstacle we would like to eliminate.

1.2 Soft heaps: using corruption to our advantage

Our goal is to find a priority queue-*like* structure that can support most of the core operations in $O(1)$ time. Intuitively, we expect that this structure will relax its internal ordering in a way that trades accuracy for speed: if `extract-min` is allowed to give incorrect answers, the structure will be freed from the $\Omega(n \log n)$ lower bound for sorting, which is the key property preventing constant-time complexity for f, g, and h.

At the same time, the data structure should not return elements completely arbitrarily: then it would be no better than an ordinary queue or a randomly shuffled array, neither of which is particularly useful for the type of application that demands a priority queue to begin with. So we are looking for a compromise: `extract-min` should return something close to the minimum element in the set, but it should not be obligated to return the exact minimum every time.

The *soft heap* is a modified priority queue that strikes this balance quite nicely. Invented by Chazelle, it operates similarly to more traditional priority queue implementations, with one major exception: whereas items in a true priority queue move through the data structure individually, objects inserted into a soft heap are handled in groups.

Specifically, a soft heap allows multiple items to share a single priority—known as a *current key* or *ckey*—as they travel throughout the heap. This technique speeds up the priority queue operations in an amortized sense by reducing the number of comparisons necessary to move elements around, a phenomenon Chazelle likens to "carpooling" [2].

The drawback is that an item's *ckey* may not match its true priority. Referred to as *corruption*, this defect can cause some items to exit the heap gruesomely out of order if their *ckeys* differ significantly from their correct priorities. But there is some good news: the number of corruptions can be restricted by an *error parameter*, ε, which falls between 0 and 1. Specifically, soft heaps obey the following theorem:

Corruption Theorem. *Let P be a soft heap with error parameter $\varepsilon \in (0, 1)$, and suppose n items have ever been inserted into P. If the current number of items in P is m, then the number of corrupted elements in P is at most*

$$\min\{m, \varepsilon n\}.$$

The ability to select ε presents a trade-off between time and correctness. We can see this in the amortized costs of the soft heap operations below.

`make-heap`(ε, x): Create a new soft heap that represents the set $\{x\}$ and has error parameter ε. Amortized cost: $O(\log \frac{1}{\varepsilon})$.

`insert`(P, x): Insert element x into soft heap P. Amortized cost: $O(\log \frac{1}{\varepsilon})$.

`extract-min`(P): Locate the set of items of minimum *ckey* in soft heap P, and remove and return an arbitrary element from that set. Amortized cost: $O(1)$.

`meld`(P, Q): Destroy soft heaps P and Q (which must have the same value of ε) and return a new soft heap R representing the set $P \cup Q$. Amortized cost: $O(1)$.

For a constant value of ε, it follows that a soft heap can handle a sequence of n insertions and deletions in $O(n)$ time. However, the weaker behavior of `extract-min` and the absence of a `decrease-key` operation mean soft heaps are not usable in classical priority queue algorithms unless the user is willing to accept an approximate solution. Thus the data structure is rapid but demanding: its asymptotic performance is extremely fast, but any algorithm that hopes to produce correct results requires considerable sophistication to work around the effects of corruption.

As an example, let us return to Prim's algorithm. Even if we had an efficient implementation of `decrease-key` that allowed us to run the algorithm in $O(m + n)$ time using a soft heap, we would have guarantee that the spanning tree we generated would actually be a *minimum* spanning tree, so the speed improvement would be of little use.

We might be tempted to throw out the soft heap at this point, but with the right approach it is still possible to compute minimum spanning trees. Indeed, Chazelle invented the soft heap for this very purpose: by handling corrupted items carefully, he was able to devise an MST algorithm that runs in $O(m\alpha(m, n))$ time, where α is the extremely slowly-growing Ackermann inverse function [3]. Soon afterward, the soft heap was used again in another MST algorithm due to Pettie and Ramachandran; their procedure has provably optimal—but unknown—running time, and uses soft heaps to back a Prim-like graph partitioning subroutine [4]. Clearly, then, soft heaps are well worth studying, even if their behavior is confounding.

We are aware of three major implementations of the soft heap. Chazelle's original version [2] stores its items in modified binomial trees, which are represented implicitly as binary trees. This structure makes it easy to rationalize soft heaps as an extension of Vuillemin's binomial heaps [5], but the implementation is somewhat convoluted and requires restructuring operations that obscure the intuition behind the data structure. Kaplan and Zwick [6] followed with a version that still uses binary trees but abandons the connection to binomial trees, and they improved on Chazelle's runtime analysis by presenting a potential-based discussion of the operations. Finally, Kaplan, Tarjan, and Zwick [7] presented another modification with an alternate amortization scheme that makes `extract-min`, rather than `insert`, the expensive $O(\log \frac{1}{\varepsilon})$ operation.

In the remainder of this report, we will present an overview of Kaplan and Zwick's first soft heap implementation [6]. We will provide a bottom-up perspective of the heap's organization and a more explicit analysis of the amortization scheme behind the structure's rapid runtime, with a view to simplifying the

intuition behind item corruption. Finally, we will include a brief look at how soft heaps perform in the practical domain.

2 The structure of a soft heap

A soft heap imposes four distinct levels of organization on the set it represents. Individual items are stored in *cells*. Cells travel together in groups in association with containers called *nodes*. Nodes are the smallest units upon which a soft heap can impose any sort of ordering; to enforce this ordering, they are stored in binary trees. The trees are themselves organized in a list to support efficient `meld` operations, like the trees of a binomial heap.

2.1 Cells and nodes

The structure of a cell is extremely simple, since it exists only to maintain a record of some item x that was originally inserted into the heap. A cell c has a copy $c.elem$ of the item it is tracking. To support constant-time merging of two groups of cells and constant-time removal of a single cell from a group, cells are organized under nodes as linked lists; hence, c also has two fields $c.prev$ and $c.next$ pointing to its neighbors in the item list.

Nodes are considerably more complicated, as they store most of the information used to organize the soft heap. A node n contains eight fields:

- $n.first$ and $n.last$. These point to the first and last cells in n's item list. $n.first$ allows easy extraction of the first element in n's item list, and $n.last$ allows a new item list to be concatenated to the end of n's current list in constant time.
- $n.left$ and $n.right$. These are the left and right child nodes of n in a binary tree.
- $n.ckey$. This is the priority of node n and, by extension, the effective priority of all cells in n's item list. It is an upper bound on the true priorities of all items in n's list; if n gets new elements from another node's item list, it will update its $ckey$ to reflect the new maximum priority in its list.
- $n.nelems$ and $n.size$. $n.nelems$ is the number of items currently in n's item list, while $n.size$ is the "ideal" number of items in n's item list. As we will see later, if $n.nelems$ falls too far below $n.size$, n will refill its item list.
- $n.rank$. This is the maximum possible height that n can have in any binary tree in the heap. Every node in a soft heap keeps the same rank from the moment it is created to the moment it is destroyed, even though the structure of the subtree it roots will change drastically throughout its lifetime.

$n.size$ is a function of $n.rank$ and the heap's error parameter ε. In particular, if n has rank k, then $n.size = s_k$, where

$$s_k = \begin{cases} 1, & \text{if } k \leq r \\ \lceil 3/2 \ s_{k-1} \rceil, & \text{if } k > r, \end{cases} \tag{1}$$

and where r is a *structural parameter* defined as follows:

$$r = \lceil \log_2(1/\varepsilon) \rceil + 5 \tag{2}$$

There are a few observations worth making at this point. First, as we will show in the following sections, a node whose *size* field is 1 will never contain more than one cell in its item list. Since a node's *ckey* is updated to match the largest priority in its new item list whenever it refills itself, it follows that no item held by a node of size 1 can possibly be corrupted. This means that the number of corrupted items in the heap is bounded above by the number of items stored in nodes of rank greater than r.

Second, we can see from equation (1) that a node's size is roughly an exponentially increasing function of its rank above r. Since the Corruption Theorem requires that the number of corrupted items be bounded above by a fraction of the number of insertions that have ever occurred, we want to devise some way of structuring the soft heap so that the number of nodes of rank k is exponentially decreasing with k; otherwise, all the items of a soft heap would rapidly be absorbed into corrupting nodes, making the data structure useless.

Third, it is easy to see from Equation (2) that $r = O(\log \frac{1}{\varepsilon})$, which is precisely the amortized cost of an `insert` operation. This is no coincidence. As we will see in the following sections, the amortization scheme of a soft heap backcharges the "travel expenses" of a cell to the `insert` operation that created that cell. Items in nodes of rank r or less have to pay for their movement on their own, while the items of a node with rank greater than r get to split the cost by "carpooling" throughout the heap. The key to achieving an $O(\log \frac{1}{\varepsilon})$ amortized cost for `insert` will be finding some way to make sure that any cell only ever has to make $O(r)$ movements on its own.

2.2 Trees

As we mentioned before, the nodes of a soft heap are stored in binary trees. These binary trees are heap-ordered: the *ckey* of any node must be less than or equal to those of its children in the binary tree—provided the node actually has children. Each tree in a soft heap is stored under a *tree header* T. In addition to some positional information we will discuss into the following section, T has a field $T.rank$, which is equal to the rank of its root node, and a pointer $T.root$ to the root node itself.

A tree T of rank k may have one of many possible structures depending on whether $k > r$, whether `extract-min` has been called on T, and how T's nodes were arranged before any `extract-min`s occurred. Figure 1 shows an example of a rank-3 tree and how its structure evolves as its elements are withdrawn by `extract-min` calls. Since a soft heap does not attempt to repair the structure of a tree as its nodes are destroyed, the heap can show considerable structural variety, which makes it difficult to reason about the true cost of any single operation.

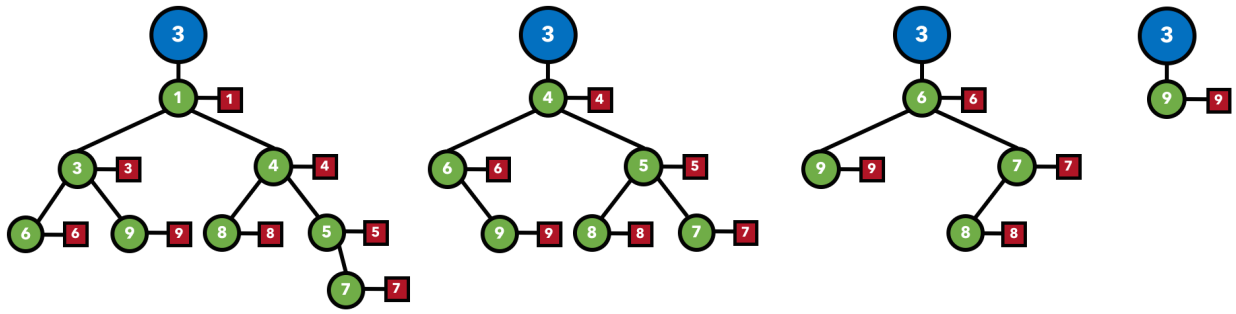


Figure 1: Snapshots of the lifetime of a rank-3 tree, from the moment of its creation until it contains just one node. The tree header is in blue and is marked with the tree's rank. Tree nodes are in green and are marked with their *ckey*s. The length-1 item lists of each node are also shown; cells appear in red. As keys are withdrawn from the tree, it shrinks upward from the leaves until eventually it empties out. A tree of any rank can become this damaged without the heap attempting to restructure it, as the restrictions on the structure of a soft heap tree are significantly less stringent than those we might see in a binomial heap or Fibonacci heap.

2.3 The rootlist

The trees of a soft heap are arranged in increasing order of rank in a doubly linked list called the *rootlist*; this arrangement makes it easier to `meld` two heaps. To support the linked list format, a tree header T contains two pointers $T.prev$ and $T.next$ to its predecessor and successor in the rootlist, if any. A soft heap admits at most one tree for each rank in $\mathbb{N}$.

A tree header also has one more field: a *suffix min* pointer, or $sufmin$. If T is a tree header in a soft heap P, then $T.sufmin$ points to the header whose root node has minimum *ckey* amongst all trees in P whose rank is greater than or equal to $T.rank$. For example, if P contains three trees T_0, T_3 and T_9, with ranks 0, 3, and 9 and root *ckey*s 2, 1, and 7, then $T_0.sufmin$ will point to T_3, $T_3.sufmin$ will point to T_3, and $T_9.sufmin$ will point to T_9. See Figure 2 for an illustration of this situation.

Suffix min pointers are designed to give easy access to the node of minimum *ckey* in a soft heap: since trees are heap-ordered by *ckey*, the minimum-*ckey* node must be the root of its tree, and that tree must be in some "suffix" of the rootlist, meaning it must be the target of the $sufmin$ pointer of the first tree in the data structure. This might seem like a needlessly roundabout way to find the minimum element, since we could just keep a global *min* pointer to the node with the lowest *ckey*. The advantage of $sufmin$ pointers lies in the amount of work that needs to be done to update them. If we kept a single *min* pointer, we would need to iterate over the entire rootlist to update it if we changed the *ckey* of the minimum node; but with $sufmin$ pointers, we only need to update pointers up until the affected tree, since the modified node cannot affect the $sufmin$ of any of the trees that come after it.

For example, if we removed the minimum element of T_3 in Figure 2, the *ckey* of that tree's root would become 8. We would need to revise $T_3.sufmin$ to point to T_9 and $T_0.sufmin$ to point to T_0, but $T_9.sufmin$ would be unaffected by the change, since T_3 is not in T_9's suffix. Thus, as alternatives to a global *min*

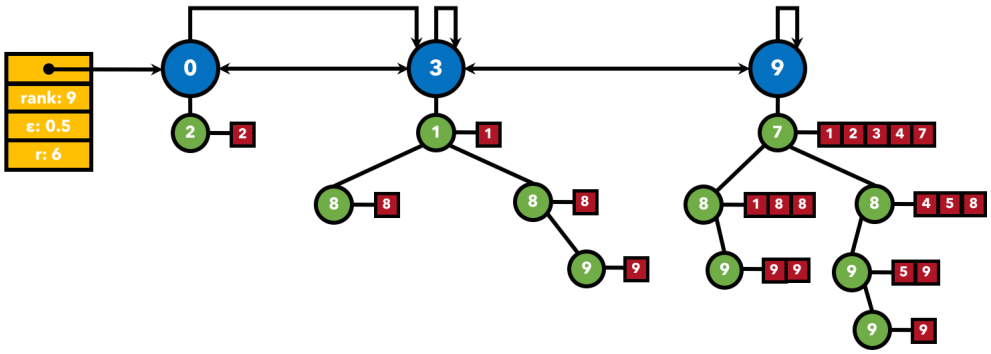


Figure 2: A rank-9 soft heap P with error parameter 0.5, featuring a heap header (in yellow) and showing $sufmin$ pointers. Since the rank-3 tree has lower root *ckey* than the tree preceding it, the first two trees both have their $sufmin$ pointers directed to it. The rank-9 tree cannot "see" this node, so its $sufmin$ pointer is directed towards itself. At any point in the lifetime of P, we can find the node of minimum *ckey* in $O(1)$ time by accessing $P.first.sufmin.root$.

pointer, $sufmin$ pointers allow us to bound the work done in updating them according to the rank of the rightmost tree we modify. Since this cost is defined in terms of the current structure of the heap, we will be able to amortize it away.

The final component of the heap is a *heap header*, P. Colored yellow in Figure 2, it contains four fields:

- $P.first$: a pointer to the lowest-rank tree in the heap.
- $P.rank$: the highest rank of any tree in the heap.
- $P.\varepsilon$: the error parameter with which the heap was created.
- $P.r$: the structural parameter r defined by (2).

3 Implementation

Now that we have a good framework for understanding how the soft heap is organized, we are ready to discuss how the core operations `make-heap`, `extract-min`, `meld`, and `insert` are implemented. We will not attempt to present every detail of the operations, since Kaplan and Zwick provide pseudocode in their original paper [6]. However, we will give a step-by-step description of what the operations do, providing illustrations where they are warranted.

In order to implement the priority queue operations, we first need to discuss three subroutines that restructure the heap: `sift`, which refills a node's item list when its *nelems* field gets too low; `combine`, which converts two trees of equal rank to form a new tree during a `meld`; and `update-suffix-min`, which fixes all appropriate $sufmin$ pointers in the soft heap after an operation modifies the minimum-*ckey* node.

3.1 `sift`

We mentioned previously that a node x should "ideally" have $x.size$ cells in its item list. To ensure that $x.nelems$ is within a constant factor of $x.size$, we call `sift`(x) whenever $x.nelems$ falls below $\frac{1}{2}\,x.size$, refilling x's item list by taking items from x's descendants until $x.nelems$ reaches $x.size$. `sift`(x) works as follows:

1. If x has no child nodes, stop: there is no way to restore the item list.
2. Determine which of x's children has lower *ckey*. In $O(1)$ time, remove that child's item list and concatenate it to x's item list, updating $x.nelems$ as appropriate.
3. Change $x.ckey$ to match the child's *ckey*.
4. If the child we modified has children of its own, call `sift` recursively on the child to repair its item list. If the child does not have children, its own item list cannot be repaired, so destroy it.
5. If $x.nelems \geq x.size$, stop. Otherwise, repeat from step 1.

Since `sift` involves both recursion and iteration, it can drastically affect the shape of the tree containing x; see Figure 3 for an example. It is worth noting that the procedure maintains the heap ordering of x's subtree, since x takes the smaller *ckey* from its two children. This new *ckey* also remains an upper bound on all of the priorities in x's item list.

Most of the work done in a soft heap can be traced back to `sift`s. A top-level `sift` call whose recursion tree is responsible for s item list transfers runs in time $O(s)$. Ideally we would be able to provide a worst-case bound for the runtime of `sift` in terms of the number of items in the heap, but this is infeasible because soft heap trees are so structurally diverse. Nevertheless, we will soon show that with the right amortization scheme it is possible to ignore the cost of a `sift` entirely.

3.2 `combine`

When a `meld` occurs, the two initial soft heaps are likely to have some trees with equal ranks. Since the final heap may only have one tree of each rank, we need some way to combine duplicates to form new trees. The procedure `combine`(T_1, T_2) takes two trees of rank k and unites them to form a tree of rank $k+1$ using a simple procedure:

1. Create a new node z of rank $k+1$. Initialize $z.size$ to s_{k+1} by setting $z.size = \lceil 3/2\ T_1.root.size \rceil$. Set z's children to be $T_1.root$ and $T_2.root$.
2. Call `sift`(z) to fill z's item list from its new children.
3. Create a new header T and initialize its *rank* to $k+1$. Set $T.root = z$.
4. Destroy tree headers T_1 and T_2 and put T into the `meld`ed rootlist in their place (the old headers are assumed to be adjacent in the new rootlist before `combine` is called).

`combine` is asymptotically no more expensive than a single `sift` operation, but the fact that it calls `sift` means we cannot make any particularly useful claims about its worst-case runtime.

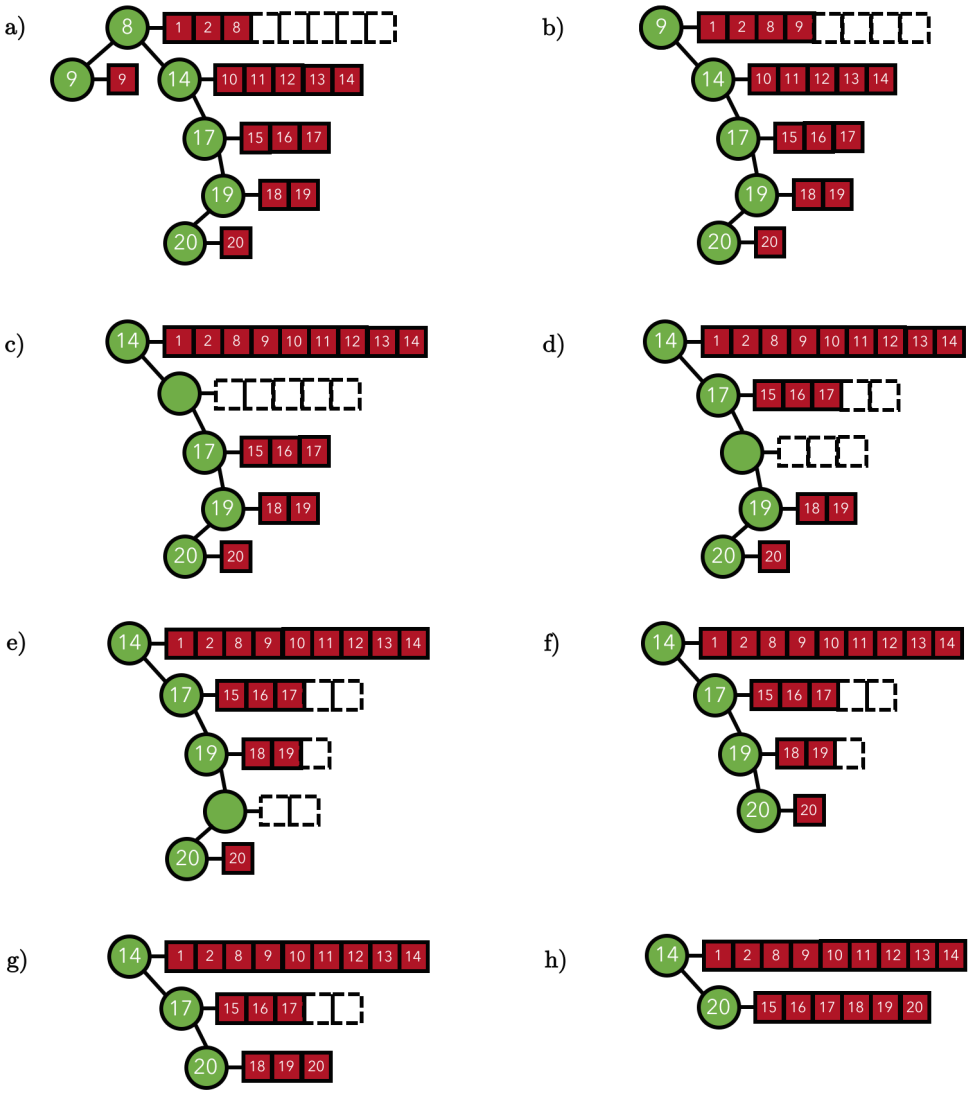


Figure 3: A typical `sift` operation. The root of this subtree has size 8; its right descendants have sizes 5, 3, 2, and 1 as we move down. **a)**: the root has only three elements, so it is deficient. **b)**: the root steals element and *ckey* 9 from its left child; since the child is a leaf, it is destroyed. **c)**: the root is still deficient, so it steals five items and *ckey* 14 from its other child. **d)-f)**: the size-5 child calls `sift` to refill its own item list, stealing three items and *ckey* 17; the size-3 descendant steals two items and *ckey* 19 from its child; and the size-2 descendant steals one item and *ckey* 20 from its child. Since this last child is a leaf, it is destroyed. The size-2 descendant is not completely fixed, but it cannot repair itself any further, so the recursion bottoms out. **g)**: the size-3 descendant has not yet been repaired, so it destroys its child to finish its `sift`. **h)**: the size-5 descendant completes its `sift` by destroying its child, terminating the two uppermost `sift` calls.

3.3 `update-suffix-min`

Since `sift` changes the root-level *ckey* of a tree and `combine` destroys two trees to create a new one with uninitialized $sufmin$ pointer, we must update all $sufmin$ pointers between the heap's first tree and the tree that was modified or newly created; otherwise, `extract-min` will not work properly on future calls.

Given that all headers following the highest-rank affected tree still have correct $sufmin$ pointers, we can repair the broken $sufmin$ pointers by starting from the rightmost modified tree and moving backwards. Explicitly, if T is the highest-rank tree whose $sufmin$ pointer might be inaccurate, then `update-suffix-min`(T) does the following:

1. If $T.next$ is `NULL`, set $T.sufmin = T$.
2. Otherwise, T's suffix min is either T or the suffix min of $T.next$. If $T.next.sufmin.root.ckey \geq T.root.ckey$, set $T.sufmin = T$; otherwise, set $T.sufmin = T.next.sufmin$.
3. If $T.prev$ is `NULL`, the fixup is finished. Otherwise, set $T = T.prev$ and repeat from step 1.

If $T.rank = k$, then the soft heap containing T contains at most $k + 1$ trees up to and including T. Therefore, `update-suffix-min` runs in $O(k + 1)$ time.

3.4 `make-heap`

Now that we have covered the key structural operations, implementing the main priority queue routines is mostly straightforward. `make-heap`(ε, x) inserts element x into a new soft heap with error parameter ε as follows:

1. Make a new rank-0, size-1 node, and initialize its item list to hold a single cell containing x.
2. Make this new node the root of a rank-0 tree header T. Set $T.sufmin$ to itself.
3. Make a new rank-0 heap header P with error parameter ε and structural parameter r defined according to (2). Set $P.first = T$.

3.5 `extract-min`

`extract-min`(P) removes an item from the list of minimum *ckey* in P, locating that list using $sufmin$ pointers:

1. Find the tree T whose root has minimum *ckey* by accessing $P.first.sufmin$. Let x be $T.root$.
2. Save a copy of the first element from x's item list and destroy the cell containing it. Decrement $x.nelems$.
3. If $x.nelems$ is now less than $\frac{1}{2}\, x.size$, its item list is deficient. We must do one of three things:
    a. If x has any children, call `sift`(x) to repair x's item list, then call `update-suffix-min`(T) to reflect the change in root *ckey*.

b. If x has no children and $x.nelems = 0$, T is now empty. Destroy x and T and remove T from P's rootlist. Call `update-suffix-min` on T's predecessor in the rootlist, if there is one. If T was the rightmost tree in the rootlist, update $P.rank$ to match the rank of T's predecessor—again, if there is one. If T was the only tree in P, mark P as empty.
c. If x has no children and $x.nelems > 0$, we cannot repair x, but we also cannot destroy T since it is nonempty. In this case, do nothing.

4. Finally, return the element we saved.

It bears repeating that if there are any corrupted items in the soft heap—that is, if the soft heap contains any nodes of rank greater than r—this operation is not guaranteed to return the minimum-priority element.

3.6 `meld`

`meld`(P, Q) is quite similar to the `meld` operation of a binomial heap: it gathers together all trees of equal rank, iterates over the unified rootlist to merge them into higher-rank trees, and then uses those higher-rank trees as "carries" that might be merged with adjacent trees themselves. The main difference is that we need to fix $sufmin$ pointers once the new heap settles.

In the following outline, we will assume $Q.rank \geq P.rank$. The opposite case is symmetric.

1. Iterate through the rootlist of Q. Each time we encounter a tree T_q in Q whose rank is greater than or equal to that of the tree T_p at the front of P, remove T_p from P and put it in front of T_q. Continue until P is empty. The original trees from P and Q are now lined up inside Q, in nondecreasing order of rank.
2. Iterate over the unified rootlist. Each time we encounter a lone tree of a given rank, skip it. Each time we encounter exactly two trees T_1 and T_2 with the same rank, `combine` them and replace them in the rootlist with the resulting tree T. Each time we encounter exactly three trees with the same rank, skip the first one and then `combine` the second and third.
3. Stop iterating when we encounter a lone tree whose rank is greater than $P.rank$; there are no more trees available to merge after this. Call `update-suffix-min` on this tree to prepare for the next `extract-min`. If this final tree was created by a `combine` on two trees of rank $Q.rank$, then increment $Q.rank$.

Figure 4 contains an illustration of these steps.

3.7 `insert`

`insert`(P, x) is trivial given the operations above: simply `make-heap`$(P.\varepsilon, x)$ and then `meld` the resulting heap into P.

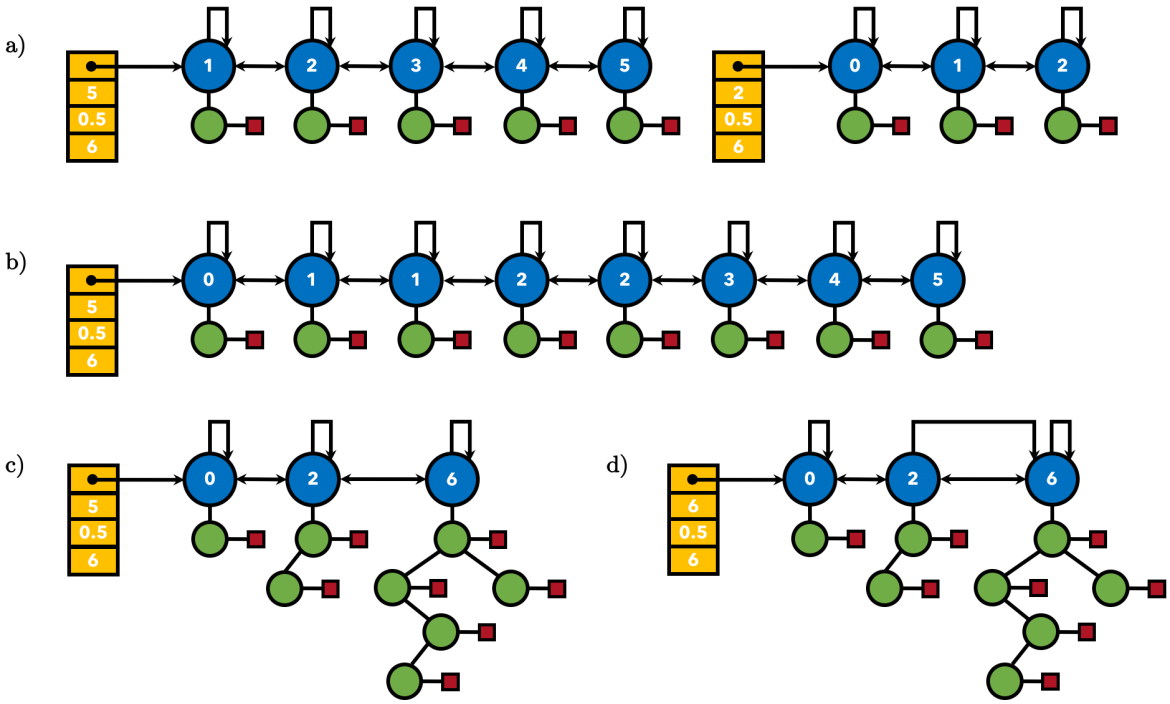


Figure 4: Schematic for a `meld` operation. **a)**: the heaps to be melded. **b)**: the rootlists of the two heaps are unified, leaving a single list of trees in nondecreasing order of rank. **c)**: all trees of equal rank are merged by `combine` in a single scan over the rootlist. **d)**: the rank of the heap header is updated, along with $sufmin$ pointers starting back from the last tree affected by the `meld`.

4 Analysis

In this section we present a formal proof of the Corruption Theorem along with an amortization scheme that bounds the running time of the core soft heap operations. The analysis in this section is adapted from Kaplan and Zwick [6], although our presentation of the amortization scheme is slightly different and includes an explicit potential function.

4.1 Bounding the number of corrupted elements

When we introduced nodes, we remarked that Equation (1) implies a node's *size* is exponentially increasing with its rank above r, so that nodes of high rank corrupt exponentially larger numbers of elements. Based on that fact, we concluded that in order to get a reasonable bound on the number of corrupted elements, we would need to design the soft heap in such a way that the number of nodes of a given rank is exponentially decreasing.

The proof of the Corruption Theorem is effectively a formalization of this reasoning. To build up to the

result, we will first present three lemmas: the first shows that a node's *size* is bounded by an exponential function of its rank; the second shows that an internal node's *nelems* is always within a constant factor of its *size*; and the third shows that our soft heap design guarantees that the number of rank-k nodes is exponentially decaying with k.

Lemma 1. *For $k \le r$, we have $s_k = 1$. For $k \ge r$, we have*

$$\left(\frac{3}{2}\right)^{k-r} \le s_k \le 2\left(\frac{3}{2}\right)^{k-r} - 1. \tag{3}$$

Proof. We proceed by induction on k. For $k < r$, the result follows immediately from Equation (1). For $k = r$, s_k is defined to be 1, which equals both $(3/2)^0$ and $2 \cdot (3/2)^0 - 1$, thus satisfying Equation (3). For $k > r$, assume inductively that the claim holds for $k-1$, which must be greater than or equal to r. Then we have

$$s_k = \left\lceil \frac{3s_{k-1}}{2} \right\rceil \ge \frac{3s_{k-1}}{2} \ge \left(\frac{3}{2}\right)^{k-r},$$

satisfying the lower bound. Furthermore,

$$\begin{aligned} s_k = \left\lceil \frac{3s_{k-1}}{2} \right\rceil &\le \frac{1}{2}(3s_{k-1} + 1) \\ &\le \frac{1}{2}\left(3 \cdot 2\left(\frac{3}{2}\right)^{k-1-r} - 2\right), \quad \text{by Equation (3)} \\ &= 2\left(\frac{3}{2}\right)^{k-r} - 1, \end{aligned}$$

satisfying the upper bound. ∎

Lemma 2. *Let x be a node of rank k. If $k \le r$, then x will always contain exactly one element in its item list until it is destroyed. If $k > r$, then x will always contain between $\frac{1}{2}x.size$ and $3x.size$ elements in its item list until it becomes a leaf.*

Proof. We will use induction to prove the separate claims for $k \le r$ and $k \ge r$.

For $k \le r$, start with $k = 0$. x contains one element when it is created by a `make-heap` operation, and x is destroyed if that element is deleted, since a rank-0 node can never have any children. For $1 \le k \le r$, $x.size = s_k = 1$, so x is only subject to a `sift` if its item list becomes empty. If x has no children, it is destroyed. Otherwise, we may assume inductively that both of x's children contain exactly one element, since they have rank $k-1$; therefore, `sift` will bring exactly one item into x's list, satisfying the claim.

For $k \ge r$, we know that x will always have at least $\frac{1}{2}x.size$ elements as long as it has children, since otherwise we will `sift` it. All we need to show is that $x.nelems \le 3x.size$. We begin with $k = r$: we have already shown that $x.nelems = x.size = 1$, which is clearly less than $3x.size$. For $k > r$, assume inductively that the claim holds for $k-1$. Suppose that the final iteration of `sift` steals the item list from child node

y, which has rank $k-1$. Before this list transfer, $x.nelems$ was at most $x.size - 1$. After the transfer, $x.nelems \leq x.size + 3y.size - 1$, by the inductive hypothesis. Since $y.size = s_{k-1}$, and $s_{k-1} \leq \frac{2}{3}s_k$ by Equation (1), we have

$$x.nelems \leq s_k + 3 \cdot \frac{2}{3}s_k - 1 \leq 3x.size.$$

■

Lemma 3. *If n insertions have ever occurred on a soft heap P, then the number of nodes of rank k in P is at most $n/2^k$.*

Proof. By induction on k. The claim certainly holds for $k = 0$, since rank-0 nodes are created only by insertions. For $k > 0$, assume inductively that the claim holds for $k-1$. A node x of rank k can only be created as a result of a `combine` operation that makes two rank-$(k-1)$ nodes the children of x. These two rank-$(k-1)$ nodes remain x's children until they are deleted, and therefore can never again be `combine`d into a new rank-k node. Thus the number of nodes of rank-k nodes is at most half the maximum number of rank-$(k-1)$ nodes; *i.e.* at most $1/2 \cdot n/2^{k-1} = n/2^k$. ■

We now prove the Corruption Theorem, which is restated here for convenience:

Corruption Theorem. *Let P be a soft heap with error parameter $\varepsilon \in (0, 1)$, and suppose n items have ever been inserted into P. If the current number of items in P is m, then the number of corrupted elements in P is at most*

$$\min\{m, \varepsilon n\}.$$

Proof. If $m \leq \varepsilon n$, the claim is obvious. Otherwise, let r be the structural parameter of P. Every node of rank at most r in P contains a single element. The node's *ckey* thus matches that element, so all corrupted items in P must be in nodes of rank greater than r. Hence, we can bound the number of corrupted elements n_c by

$$n_c \leq \sum_{k=r+1}^{\infty} S_{max}(k) N_{max}(n, k),$$

where $S_{max}(k)$ is the maximum possible number of items in a rank-k node and $N_{max}(n, k)$ is the maximum possible number of rank-k nodes in the heap. Applying Lemmas 1, 2, and 3 gives us

$$n_c \leq \sum_{k=r+1}^{\infty} 3s_k \frac{n}{2^k} \leq \frac{n}{2^r} \sum_{k=r+1}^{\infty} 6 \left(\frac{3}{2}\right)^{k-r} \frac{1}{2^{k-r}} = \frac{6n}{2^r} \sum_{j=1}^{\infty} \left(\frac{3}{4}\right)^j = \frac{18n}{2^r}.$$

Since $r = \lceil \log_2 \frac{1}{\varepsilon} \rceil + 5$, this leaves us with

$$n_c \leq \frac{18n}{(1/\varepsilon)2^5} \leq \varepsilon n,$$

completing the proof.

■

It is important to note that the Corruption Theorem cannot provide a useful bound on the number of corrupted items in a soft heap as a function of the number of items currently in the heap. This is because corrupted items that are pulled out of a soft heap will eventually trigger `sift` operations that pull higher *ckey*s up from child nodes and recorrupt the elements left behind in the item list. This makes it particularly tricky to design algorithms that work well with soft heaps. However, the following corollary to the Corruption Theorem can be of some use in algorithm design:

Corollary 4. *Let e be an element retrieved by a call to* `extract-min` *on a soft heap P with error parameter ε which has been subject to n insertions. If e was not the minimum-priority element in P, there are at most εn elements left behind in P whose true priorities were smaller.*

Proof. All the neglected elements have true priority less than e's true priority. e's true priority is less than or equal to its *ckey*. Since e was extracted first, its *ckey* must have been less than or equal to the *ckey*s of all the neglected elements. Thus all of the elements left behind must have had *ckey*s greater than their priorities at the time of e's extraction, meaning they were corrupted. Applying the Corruption Theorem completes the proof. ∎

4.2 Runtime analysis

We are now ready to prove the amortized costs of the soft heap operations.

Theorem 5. *For a given error parameter ε,* `insert` *and* `make-heap` *run in amortized time $O(\log \frac{1}{\varepsilon})$, while* `meld` *and* `extract-min` *run in amortized time $O(1)$.*

The proof is quite convoluted, so we will give an overview of the underlying intuition first.

Most of the work done in a soft heap can be traced back to `sift` operations and `update-suffix-min` operations, as well as to the `meld` subroutine that lines up trees and calls `combine` on them. Kaplan and Zwick's amortization scheme works by backcharging this work to item insertions, leaving behind a constant amortized cost for `meld` and `extract-min`. These are the two most important features of the amortization:

- Every item list transfer within a `sift` takes real time $O(1)$. The amortization scheme splits this cost among elements that "carpool" together, so that elements pay less and less as they move into higher-rank nodes. It turns out that from the moment of its insertion to the moment of its extraction, a single item never has to pay more than $O(r)$ potential units to travel throughout the heap.

- `extract-min` runs in constant time, except when the number of elements in the target root x drops below $\frac{1}{2}x.size$, in which case we need to call `sift` (which pays for itself by charging elements). Then we need to call `update-suffix-min` to reflect the change in root *ckey*. For this expensive step to occur, a certain number of `extract-min` operations must have deleted elements from x since the last time it was `sift`ed. By amortizing the cost of `update-suffix-min` over all these previous `extract-min` operations, we get an $O(1)$ amortized cost for `extract-min` overall.

Proof. We begin by assigning the following potential to a soft heap P:

$$\Phi(P) = 1 + P.rank + N_{\text{nodes}} + \sum_{x\text{: root}} (x.rank + 4 + (r+2)del(x)) + \sum_{c\text{: cell}} \left(r + \sum_{i=c.rank}^{\infty} \left\lceil \frac{s_i}{2} \right\rceil^{-1} \right) \quad (4)$$

where N_{nodes} is the number of nodes in heap P, $c.rank$ is the rank of the node containing cell c, and $del(x)$ is the number of elements deleted from x since `sift` was last called on x, or since x was created.

Equation (4) effectively assigns potential to each level of the heap structure that we introduced in Section 2. P's rootlist carries potential according to the maximum number of trees it might contain; each tree in P carries potential according to its own rank, as well as according to how soon its root will call `sift`; each node carries one unit of potential; and each cell carries enough potential to pay for every movement it could possibly make. Φ is nonnegative for any soft heap P. If we define the rank of an empty soft heap to be -1, then Φ is 0 when P is empty.

We start off by showing that operations `sift` and `combine` run in amortized time 0, *i.e.* they pay for themselves. We will then be able to move on to the main operations, which use these two as building blocks.

- `sift`(x): Let x be a node of rank k, and let y be the child whose item list it steals. The real cost of an item list transfer is $O(1)$; we need to show that one unit of potential is lost to compensate. There are two cases to analyze:

  1. If $y.nelems < \frac{1}{2}y.size$, then y has to be a leaf by Lemma 2, meaning y will be deleted when its list is emptied. As a result of this deletion, the potential will decrease by at least 1, because N_{nodes} decreases by 1. This unit of potential pays for the transfer.
  2. If $y.nelems \geq \frac{1}{2}y.size$, then $y.nelems \geq \lceil \frac{1}{2}y.size \rceil = \lceil \frac{1}{2}s_{k-1} \rceil$. Each cell c from y's item list is moved to x's item list, and thus $c.rank$ goes from $k-1$ to k. This eliminates the first term of the rightmost sum in Equation (4) for each cell that is transferred, decreasing $\Phi(P)$ by at least

  $$y.nelems \cdot \left\lceil \frac{s_{k-1}}{2} \right\rceil^{-1} \geq \left\lceil \frac{s_{k-1}}{2} \right\rceil \left\lceil \frac{s_{k-1}}{2} \right\rceil^{-1} = 1.$$

  This unit of potential pays for the transfer.

  Since the cost of a `sift` is bounded by the number of item list transfers it causes, `sift` has amortized cost 0.

- `combine`(T_1, T_2): Suppose $T_1.rank = T_2.rank = k$, and let x and y be the trees' roots. `combine` does $O(1)$ real work before calling `sift`. The `sift` operation pays for itself, so we only need to analyze the amortized cost of the work that came before it. `combine` creates a new root z of rank $k+1$ and sets x and y as its children. The creation of z increases N_{nodes} by 1, thus increasing $\Phi(P)$ by 1. Since z becomes the root of a new rank-$(k+1)$ tree, we must add an additional $z.rank + 4 = k + 5$ units, for a total of $k+6$ so far. Finally, the creation of the new tree will increment $P.rank$ if $P.rank$ was k before. To compensate, we must add in another unit of potential; thus, $\Phi(P)$ increases by up to $k+7$ units.

On the other hand, x and y are no longer root nodes. This decreases the potential by $2(k+4)$. Thus, in total, the potential decreases by at least $2(k+4) - (k+7) = k+1$. One unit of potential pays for the $O(1)$ cost of setting up z, so that `combine` pays for itself and still releases k additional units of potential. As we will see, these units can be used to pay for an `update-suffix-min` operation moving backward from the new tree we just created.

We can now examine the heap operations.

- `meld`: Assume we meld two heaps P and Q, where $k = P.rank$ and $Q.rank \geq P.rank$.

  The first step of this operation threads the trees of P into Q's rootlist. This step runs in time $O(k+1)$, since it stops after moving through at most $k+1$ trees in each rootlist. As a result of this transfer, Q takes on all of P's potential, except for the $1 + P.rank$ term at the beginning of Equation (4); these $k+1$ units are released to pay for the rootlist merge.

  After this first step, there are two possibilities:

  1. The merged rootlist contains at most one tree of every rank, meaning `combine` is never executed. In this case, we only need to run `update-suffix-min` starting at the highest ranked tree that came from P. This tree has rank k, so the whole operation runs in time $O(k+1)$. The $k+1$ units of potential released by the destruction of P pay for this operation in addition to the initial rootlist merge.
  2. Q contains some trees with the same rank, in which case we must call `combine`. These operations pay for themselves, but as a result we may need to call `update-suffix-min` on a tree with rank greater than k, in which case we need to find a new source of potential. This potential comes from the excess released by `combine`: if the final tree affected by the `meld` has rank $l > k$, then it was created by a `combine`. That `combine` operation released $l-1$ units of potential by destroying two rank-$(l-1)$ trees; this potential pays for all but a constant amount of the cost of the ensuing `update-suffix-min` operation.

  We have shown that all but a constant amount of the work done by `meld` can be paid for by the release of potential from the argument heaps P and Q. Thus `meld` has amortized cost $O(1)$.

- `extract-min`: Suppose we extract element e from root x. Let $k = x.rank$. Finding x and removing e takes $O(1)$ time, but afterward we might have to call `update-suffix-min` on the predecessor of the tree containing x. There are three cases to consider for this operation:

  1. If x is a leaf and e was its last element, then x is destroyed, which releases at least $k+4+1$ units of potential. Those units are enough to pay for the `update-suffix-min` starting from the predecessor tree for root x, since that tree has rank at most $k-1$. The amortized cost is $O(1)$ in this case.

2. If x is not empty and we do not need to call `sift`—that is, if x is a leaf or if $x.list \geq \frac{1}{2}x.size$—then $del(x)$ increases by 1, increasing Φ by $r+2$. However, the removal of e also means that the potential then decreases by at least r. Furthermore, $x.ckey$ does not change, meaning no `update-suffix-min` is needed. Thus `extract-min` must only take on amortized cost $O(1)$ to pay for the 2-unit potential increase.

3. If x is not a leaf and $x.nelems$ falls below $\frac{1}{2}x.size$, we call `sift` and later call `update-suffix-min` starting at x. The `sift` pays for itself, but we need to make up the $O(k+1)$ cost of `update-suffix-min`. The call to `sift` resets $del(x)$ to 0, releasing $(r+2)\cdot del(x)$ units of potential. Since x is not a leaf, we had $x.nelems \geq x.size$ after the previous `sift` operation. Therefore, x lost at least $\lceil \frac{1}{2}x.size \rceil$ elements since its last `sift`, so that $del(x) \geq \lceil \frac{1}{2}x.size \rceil$. Thus Φ decreases by at least $(r+2)\lceil \frac{1}{2}x.size \rceil = (r+2)\lceil s_k/2 \rceil$ when `sift` is called.

   For the amortized cost of `extract-min` to be constant, we need to show that $(r+2)\lceil s_k/2 \rceil \geq k+1$. First, notice that for $k \leq r+1$ we have $s_k \geq 1$, meaning $\lceil s_k/2 \rceil \geq 1$, and

$$(r+2)\lceil s_k/2 \rceil \geq (r+2) \geq k+1.$$

   For $k \geq r+2$, Lemma 1 gives us

$$(r+2)\lceil s_k/2 \rceil \geq (r+2)\frac{s_k}{2} \geq \frac{r+2}{2}\left(\frac{3}{2}\right)^{k-r}.$$

   We want to show that this quantity is at least $k+1$, which requires that

$$\frac{1}{2}\left(\frac{3}{2}\right)^{k-r} \geq \frac{k+1}{r+2}.$$

   Since $k \geq r+2$, this inequality is satisfied as long as $r \geq 6$. This follows from Equation (2), since $\log_2 \frac{1}{\varepsilon}$ is strictly greater than 0 for $\varepsilon \in (0,1)$. Thus Φ decreases by at least $k+1$, paying for an `update-suffix-min` starting from the tree containing x.

We have paid for all non-constant costs associated with `extract-min`, so its amortized cost is $O(1)$.

- `make-heap`(ε, x): This operation takes real time $O(1)$ and produces a rank-0 heap containing a single rank-0 root node and one cell. By Equation (4), the potential of the new heap is therefore $1+1+4+r+\sum_{k=0}^{\infty}\lceil \frac{s_k}{2} \rceil^{-1}$. By Lemma 1,

$$\sum_{k=0}^{\infty}\left\lceil \frac{s_k}{2} \right\rceil^{-1} \leq r + \sum_{k=r}^{\infty} 2\left(\frac{2}{3}\right)^{k-r} = r+6,$$

  so the newly created heap has at most $2r+12$ units of potential. Since $r = O(\log \frac{1}{\varepsilon})$, it follows that `make-heap` has amortized cost $O(\log \frac{1}{\varepsilon})$.

- `insert`(P, x) consists of a call to `make-heap`(ε, x) followed by a call to `meld`. Therefore, its amortized cost is $O(\log \frac{1}{\varepsilon}) + O(1) = O(\log \frac{1}{\varepsilon})$.

We have shown the amortized costs of all the soft heap operations, so the proof is finally complete.

■

5 The practical performance of soft heaps

It took us a rather horrific slog to show that a soft heap obeys the specifications we laid out in the introduction. In return for our hard work, we have access to graph algorithms that are extremely rapid in an asymptotic sense—but these algorithms are so complex that they are unlikely to see much practical application. Needless to say, this is rather unsatisfying. Do soft heaps actually accomplish anything worth the effort of their implementation?

We conclude this report by attempting to answer that question. We created an implementation of Kaplan and Zwick's soft heap using their pseudocode in [6] and tested its performance using a few different metrics. First we analyzed how quickly the suite of heap operations ran for several values of ε to decide how the real operation costs compare to their amortized costs. Then we analyzed the soft heap's accuracy as an approximate sorting algorithm, recording how much it tends to scramble its input as ε varies. Finally, we compared the runtimes of approximate and exact soft heap sorting algorithms to a variety of standard algorithms to see how the complex organization of the data structure affects its practical performance.

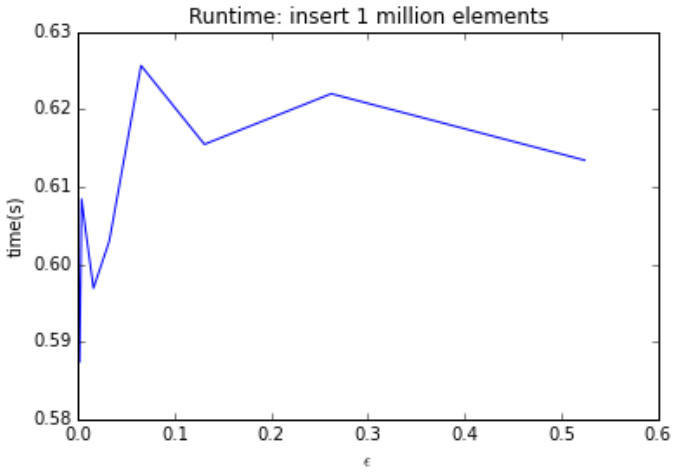
All tests were run on a 16-core Xeon E5-2650 processor with 128GB of RAM under Linux. All testing programs were written in C and compiled with the same options.

5.1 Runtime: Dependency on ε

The amortized analysis of Section 4 gives us an idea of the overall asymptotic performance of the soft heap on a full operation sequence. However, given the intricacy of the potential function (4), the analysis does not tell us much about which operations have highest cost in the worst case. Here we present and analyze empirical results concerning the runtimes of the different operations as they vary with ε.

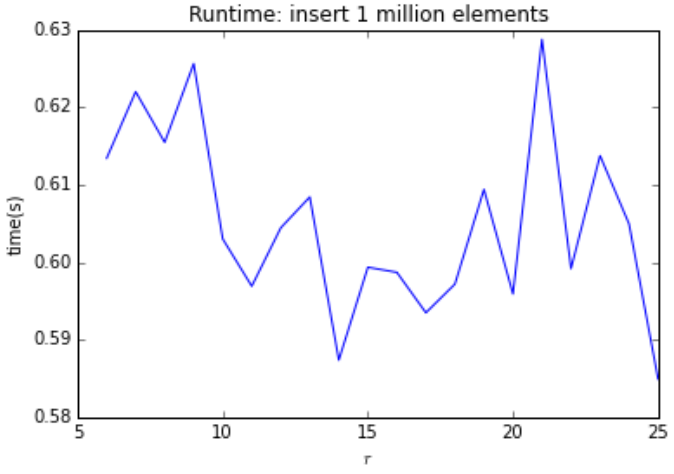
5.1.1 insert

First, we generated 1 million random keys and measured the total time necessary to insert them into an initially empty soft heap, for different values of ε. The following results are averages over 10 trials.



Notice that we did not run any tests for $\varepsilon > 0.5$. This is because ε only influences a soft heap through the choice of structural parameter r according to Equation (2). Hence, it suffices to test only the values of ε that yield every integer value of r between 6 (the minimum) and $\lceil \log_2(n) \rceil + 5$, where n is the number of elements inserted. For all r above this last value, the Corruption Theorem forbids any nodes with size greater than 1, meaning all remaining soft heaps perform identically, acting like standard priority queues.

Since $n = 1$ million, we therefore only needed to test values of r between 5 and 25. Plotting insertion times against r gives us the following:



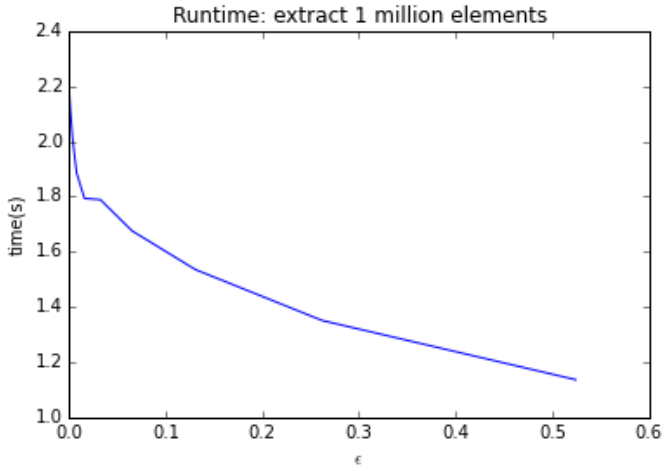
This plot actually gives us more information. Remember that higher values of r correspond to less corruption. Overall, it seems that `insert` effectively has a constant runtime with respect to r (and thus ε).

The variations we see on the plot are small and do not seem to be correlated with the value of ε.

Recall that the amortized analysis gave `insert` an amortized runtime of $O(\log \frac{1}{\varepsilon})$. But in that analysis, `insert` was mostly paying for a potential increase, *i.e.* paying for future costly operations. In reality `insert` just melds a heap of rank 0 with the current heap. Most of the time this single-item `meld` triggers a small number of `sift`s; apparently the cost of this process is not very dependent on ε, at least when it is averaged over all the inserted elements.

5.1.2 extract-min

We also measured the total time necessary to extract 1 million elements from the soft heaps produced in the previous section. These are the results averaged over the 10 trials:

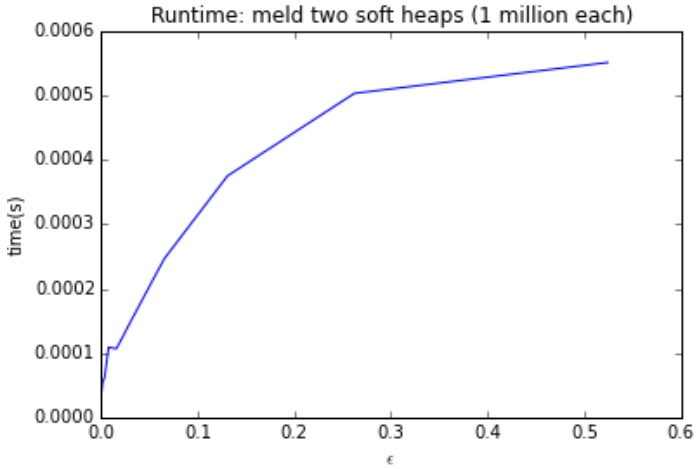


We can see here that the $O(1)$ amortized cost of `extract-min` obscures the true cost of the associated `sift`s and `update-suffix-min`s; in terms of real performance, `extract-min` is actually slower than `insert`. We can also see a clear $O(\log \frac{1}{\varepsilon})$ dependency in the cost of the extraction sequences, which makes sense given the amortized analysis and what we already know about `insert`. Specifically, the amortization scheme tells us that performing n insertions and n extractions should have a total runtime of $O(n \cdot \log \frac{1}{\varepsilon} + n \cdot 1) = O(n \log \frac{1}{\varepsilon})$, which is $O(\log \frac{1}{\varepsilon})$ for the fixed value of n in these trials. Since the cost of the insertion sequence appears to be constant, it is not surprising that the dependency on ε appears in the extraction sequence instead.

Intuitively, we observe this effect because the soft heap corrupts more keys for higher ε, meaning `extract-min` is more likely to do real work $O(1)$ on any given extraction. The operation only does $\omega(1)$ work when the list of elements in a node x drops below $\frac{1}{2}x.size$, and this event becomes progressively less common as ε increases.

5.1.3 meld

The last operation is `meld`. To analyze this operation, we generated two arrays of 1 million random keys each, filled two soft heaps from the arrays, and measured the time necessary to `meld` them for different values of ε. We repeated this experiment 10 times and obtained the following mean runtimes:



Surprisingly, `meld` slows down for higher values of ε. To understand why this happened, we used `callgrind` to analyze how ε impacts execution.

The first thing we noticed is that for a higher value of ε, `sift` tended to execute more instructions on average. One possible explanation is that for high ε, any given node might have to be `sift`ed multiple times until there are enough elements in its list.

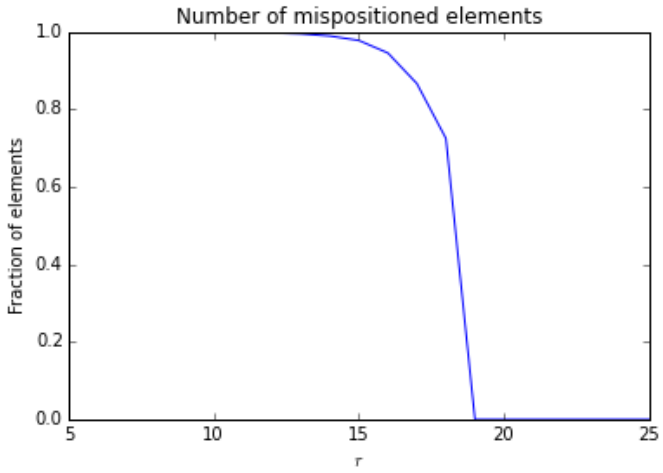
The second thing we noticed is that `sift` resulted in significantly more cache misses for high values of ε. Most of these cache misses (more than 75%) were caused by the concatenation of item lists. Our implementation accesses cells at both ends of each list during each concatenation. If these locations are not close in memory, the operation has poor spatial locality and is quite likely to suffer a cache miss, which hurts the performance of `sift`. When ε is low and most nodes contain only one element, this problem is much less prominent.

5.2 Soft heaps and approximate sorting

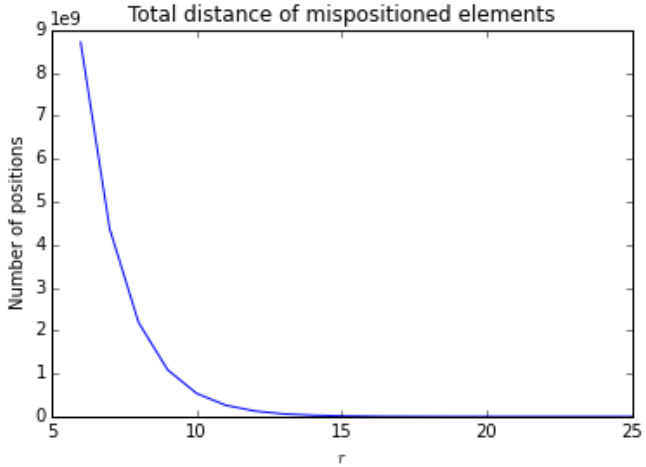
One potential use of soft heaps is as approximate sorters. This can be useful for applications that do not need a perfectly sorted input but produce better results when the input is near-sorted. However, the Corruption Theorem tells us very little about how well a soft heap near-sorts an input sequence, so to analyze its empirical performance, we ran the following test: pick a uniform random permutation of the key set $\{0, \ldots, n-1\}$, insert the keys into the soft heap in the order given by the permutation, and then

examine the order in which they are extracted. We ran this experiment for $n = 1$ million. All the plots in this section are plotted with respect to r rather than ε for ease of analysis.

The first metric we thought of using was the number of mispositioned elements. We obtained the following plot by running one test for each value of r:



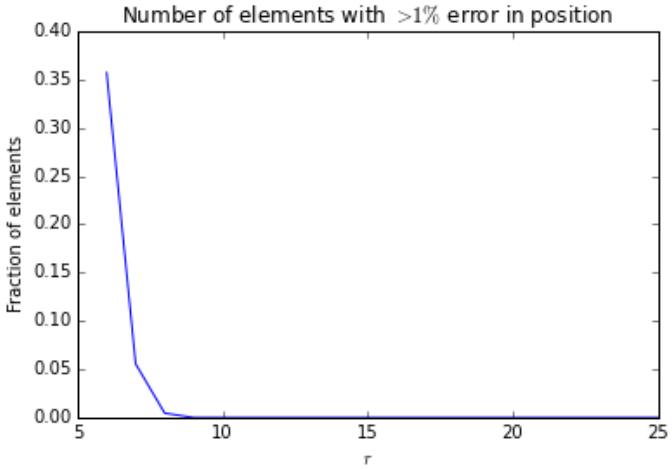
We can see that as soon as even a few elements get shuffled, the number of misplaced elements skyrockets. Hence, this particular metric does not seem very useful. The next idea we had was to compute the distance between the output position and the correct position for each element and to sum the total distance over all elements. This gave us the following plot:



This plot is much more useful. We can see a whole range of r-values where the output is not perfectly

sorted but the elements are quite close to their correct positions. Since the runtime of this approximate sorting routine increases linearly with r, the plot allows us to visualize the trade-off between time and quality of output; these results might be useful for choosing the value of ε that works best for a specific application.

Another similar metric is the number of elements that are mispositioned by more than 1% (with respect to n). This metric yielded the following plot:



The soft heap seems to perform very well with respect to this metric. By choosing ε around 0.1 ($r = 9$), we can get almost all elements to fall within 1% of their correct positions. And even with the lowest possible r, only around 35% of elements were misplaced by more than 1%.

5.3 Soft heaps and exact sorting

Of course, the accuracy of a soft heap as a near-sorting algorithm is meaningless in the practical domain unless there is some observable performance advantage in using it over a standard sorting algorithm. Recall that a soft heap acts as a perfect sorter when $\varepsilon < 1/n$; when this bound is met, we can sort an input array in $O(n \lg n)$ time with the standard procedure of `insert`ing every element and then calling `extract-min` until the soft heap is empty. Asymptotically, then, a soft heap can sort just as well as any other comparison-based sorting algorithm. But due to the heap's structural complexity, we would expect that it performs significantly worse in practice due to high constant factors and poor locality of reference.

To see how well a soft heap holds up against more common sorting algorithms, we sorted a series of randomly generated integer input arrays using a soft heap and compared its running time to mergesort, heapsort, quicksort, radix sort, and the built-in GNU qsort function for a series of exponentially growing input sizes. The results are shown in the first chart of Figure 5, and represent a series of averages over each of 5 trials.

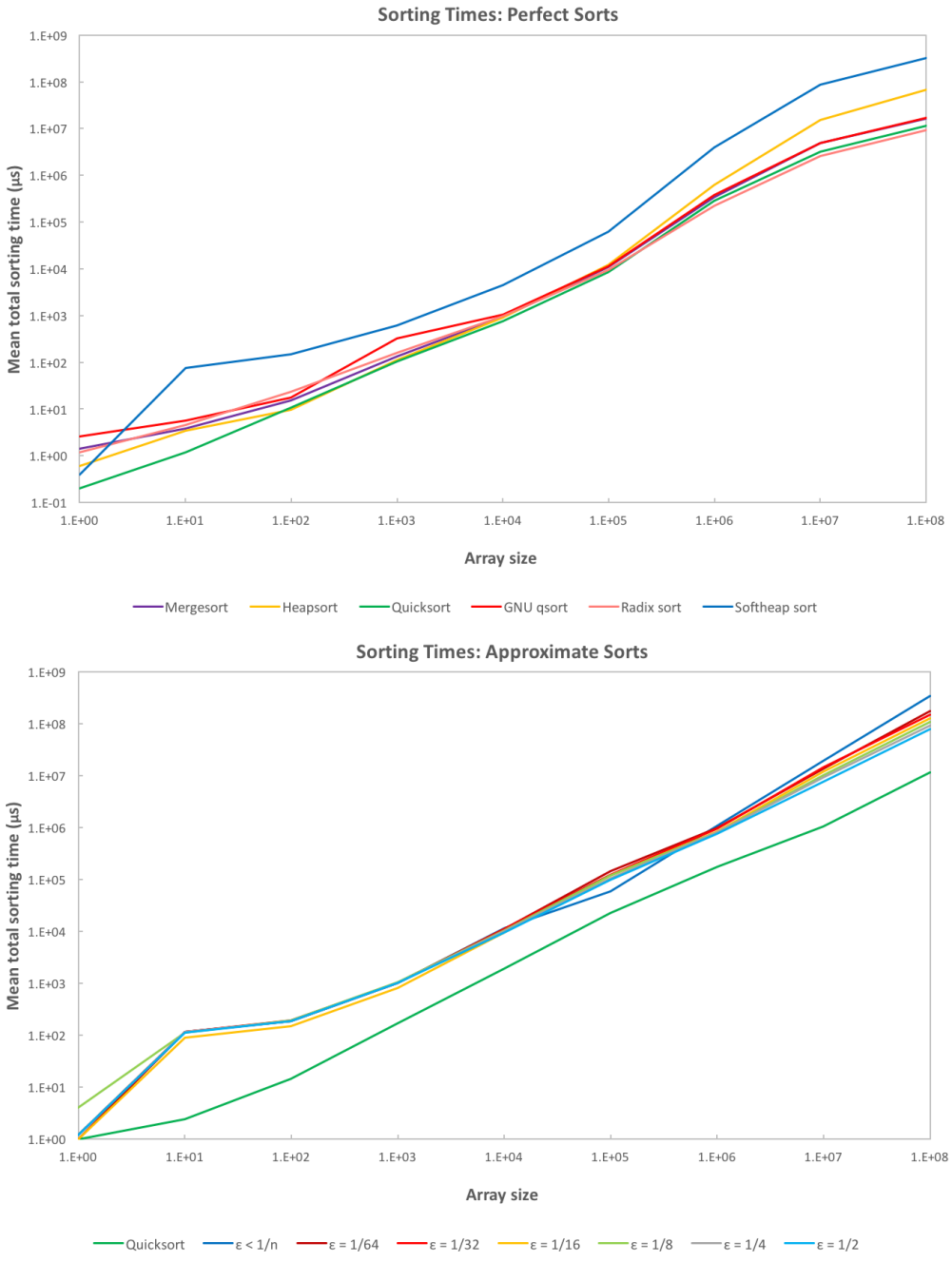


Figure 5: Runtime performance of a soft heap as an exact and approximate sorter in comparison to standard sorting algorithms. Array sizes and running times (in microseconds) are displayed on a log-log plot in each chart. Notice the near-identical performance of all the soft heap sorting routines in the second plot.

It seems the soft heap's sorting performance is pitiful at best. It is consistently at least an order of magnitude slower than all the other sorting algorithms, even though its asymptotic performance appears to be comparable to the comparison-based sorts. This is not particularly surprising, since a soft heap requires a tremendous amount of storage—whereas an in-place algorithm such as quicksort only needs one machine word per element, heapsort requires at least eleven machine words per element: eight for each node and three for each cell, in addition to the tree headers and heap header.

Even when we relax ε and allow the heap to make errors, it still performs quite poorly. The second plot in Figure 5 compares the performance of quicksort and a "perfect" soft heap sort to a variety of approximate soft heap sorts over a range of ε parameters. Increasing ε provides a small runtime benefit—the heap with $\varepsilon = 1/2$ performs about 4 times faster than the perfect heap for arrays larger than 1 million elements—but this is not nearly enough to overcome the runtime gap between the perfect soft heap and quicksort, which runs around 30 times faster across all input sizes. If these values are at all indicative of the soft heap's performance as a general-purpose data structure, then we have little reason to think it will ever prove useful in any practical applications.

6 Final remarks

While soft heaps are beautiful data structures from a theoretical perspective, their intricacy cripples them in a practical sense, as they are far too bulky to perform reasonably fast even when written in a quick language such as C. While our empirical analysis would be more complete with a discussion of some additional soft heap algorithms, *e.g.* Chazelle's $O(n)$ deterministic selection algorithm [2] or the use of a soft heap as an approximate priority queue in comparison to other meldable heaps, we expect the general trend to remain the same.

The results we saw in Section 5.1 are an excellent example of the limits of an amortized analysis: for all the work we put into proving Chazelle's runtime bounds, it turns out that the real costs of `insert` and `extract-min` appear to be flipped with respect to their amortized costs. This may have been a motivating factor in the creation of the soft heap presented later by Kaplan, Tarjan, and Zwick [7], where the amortized bounds match the costs we see here. It would be interesting to profile their implementation as we did with the version presented in [6] to see whether its true operation costs seem to match its amortized costs.

It remains to be seen whether the runtime bounds of a soft heap can be met in a worst-case rather than an amortized sense, or whether there exists an implementation of a soft heap that can compete with the performance of a standard priority queue in the practical domain. Either improvement would significantly increase the flexibility of the data structure, especially the latter: an approximate priority queue with linear-time performance bounds capable of beating a true priority queue could serve as a valuable tool in time-sensitive applications where a small error rate is permissible.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd. ed.). The MIT Press, Cambridge, MA.

[2] Bernard Chazelle. 2000. The soft heap: an approximate priority queue with optimal error rate. *J. ACM* 47, 6, Article 3 (Nov. 2000), 16 pages. DOI:http://dx.doi.org/10.1145/355541.355554

[3] Bernard Chazelle. 2000. A minimum spanning tree algorithm with inverse-Ackermann time complexity. *J. ACM* 47, 6, Article 4 (Nov. 2000), 20 pages. DOI:http://dx.doi.org/10.1145/355541.355562

[4] Seth Pettie and Vijaya Ramachandran. 2002. An optimal minimum spanning tree algorithm. *J. ACM* 49, 1, Article 2 (Jan. 2002), 19 pages. DOI:http://dx.doi.org/10.1145/505241.505243

[5] Jean Vuillemin. 1978. A data structure for manipulating priority queues. *Commun. ACM* 21, 4 (Apr. 1978), 309-315. DOI:http://dx.doi.org/10.1145/359460.359478

[6] Haim Kaplan and Uri Zwick. 2009. A simpler implementation and analysis of Chazelle's soft heaps. *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA09)*. ACM Press, New York, NY, 477-485. DOI:http://dx.doi.org/10.1137/1.9781611973068.53

[7] Haim Kaplan, Robert Tarjan, and Uri Zwick. 2013. Soft heaps simplified. *SIAM J. Comput.*, 42, 4 (Aug. 2013), 1660-1673. DOI:http://dx.doi.org/10.1137/120880185