SIAM J. COMPUT.
Vol. 42, No. 4, pp. 1660–1673

SOFT HEAPS SIMPLIFIED*

HAIM KAPLAN†, ROBERT E. TARJAN‡, AND URI ZWICK†

Abstract. In 1998, Chazelle [*J. ACM*, 47 (2000), pp. 1012–1027] introduced a new kind of meldable heap (priority queue) called the *soft heap*. Soft heaps trade accuracy for speed: the heap operations are allowed to increase the keys of certain items, thereby making these items *bad*, as long as the number of bad items in the data structure is at most εm, where m is the total number of insertions performed so far, and ε is an error parameter. The amortized time per heap operation is $O(\lg \frac{1}{\varepsilon})$, reduced from $O(\lg n)$, where n is the number of items in the heap. Chazelle used soft heaps in several applications, including a faster deterministic minimum-spanning-tree algorithm and a new deterministic linear-time selection algorithm. We give a simplified implementation of soft heaps that uses less space and avoids Chazelle's *dismantling* operations. We also give a simpler, improved analysis that yields an amortized time bound of $O(\lg \frac{1}{\varepsilon})$ for each deletion, $O(1)$ for each other operation.

Key words. heaps, priority queues, data structures

AMS subject classification. 68P05

DOI. 10.1137/120880185

1. Introduction.

A heap (priority queue) is a data structure consisting of a set of *items*, each item e having a *key* $e.key$ selected from a totally ordered universe. Heaps support the following operations:

`make-heap`(): Create and return a new, empty heap.

`insert`(e, H): Return the heap obtained by inserting item e with predefined key into heap H. Item e must be in no other heap.

`meld`(H_1, H_2): Return a heap containing all the items in heaps H_1 and H_2. Heaps H_1 and H_2 must be distinct.

`find-min`(H): Return an item of minimum key in heap H. If there is more than one item of minimum key in H, `find-min`(H) can return any such item, but repetitions of `find-min`(H) without an intervening change in H (an insertion, deletion, or meld) must return the same item.

`delete-min`(H): Delete `find-min`(H) from heap H and return the resulting heap.

Some heaps also support the following operation:

`delete`(e, H): Delete item e from heap H and return the resulting heap. (This operation requires that e is currently contained in H but in no other heap, and it is given the location of e in H.)

Each heap update operation destroys its input heap or heaps in the process of building its output heap. This guarantees that at all times the current set of heaps is item-disjoint.

*Received by the editors June 7, 2012; accepted for publication (in revised form) April 10, 2013; published electronically August 1, 2013. A preliminary version of this paper appeared in Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms, 2009, pp. 477–485.

http://www.siam.org/journals/sicomp/42-4/88018.html

†School of Computer Science, Tel Aviv University, Tel Aviv, Israel (haimk@tau.ac.il, zwick@tau.ac.il). The first author was partially supported by BSF grant 2006204. The last author's research was supported by BSF grant 2006261.

‡Department of Computer Science, Princeton University, Princeton, NJ and HP labs, Palo Alto, CA (ret@cs.princeton.edu). This author was partially supported at Princeton by BSF grant 2006204 and NSF grants CCF-0830676 and CCF-0832797, and while visiting Stanford by an AFOSR MURI grant. The information contained herein does not necessarily reflect the opinion or policy of the federal government and no official endorsement should be inferred.

We consider an arbitrary sequence of heap operations starting with no heaps. We denote by m, d, and n the number of insertions, the number of deletions, and the number of items in the heap or heaps participating in the current operation, respectively. We restrict our attention to implementations in which the only operations on keys are binary comparisons.

Since n items can be sorted by making an empty heap and doing n insertions followed by n pairs of `find-min` and `delete-min` operations, the $\Omega(n \lg n)$ lower bound for comparison-based sorting implies that at least one of the operations `insert`, `find-min`, and `delete-min` must take $\Omega(\lg n)$ amortized time. The *binomial queue* of Vuillemin [30] supports all the heap operations in $O(\lg n)$ worst-case time per operation. The amortized time is $O(\lg n)$ per `delete-min` or `delete` and $O(1)$ for each of the other operations. The latter bounds can be made worst-case [3]. The *Fibonacci heap* of Fredman and Tarjan [14] extends the binomial queue to support `decrease-key` operations, each of which decreases the key of a given item in a given heap. The amortized time is $O(\lg n)$ per `delete-min` or `delete` and $O(1)$ for each of the other operations, including `decrease-key`. Brodal [4] devised a very complicated data structure that achieves the same bounds in the worst-case; Brodal, Lagogiannis, and Tarjan [5] recently proposed a much simpler data structure with the same bounds. Many variants of these data structures have been proposed. (See [5, 17, 19] and the references therein.)

Chazelle, in designing a fast deterministic minimum-spanning-tree algorithm [7, 8, 9], needed a heap implementation with a $o(\lg n)$ time bound per operation. He obtained such an implementation by trading accuracy for speed: heap operations are allowed to *increase* item keys, thereby making these items *bad* (Chazelle called this *corrupting* their keys), as long as not too many items are bad. Specifically, for an error parameter ε, the number of bad items is allowed to be at most εm. In return, the amortized time per operation is $O(\lg \frac{1}{\varepsilon})$ per insertion and $O(1)$ for each other operation. Chazelle called his data structure the *soft heap*. Soft heaps make errors in `find-min` and `delete-min` operations, but in a controlled way.

Chazelle applied soft heaps to several problems. One is selection (generalized median-finding). Chazelle's linear-time algorithm is an alternative to previous linear-time algorithms [2, 10, 26]. His other sorting-related applications are dynamic maintenance of percentiles and approximate sorting. His most interesting and least immediate application is finding minimum spanning trees. Chazelle's algorithm [7, 8] uses soft heaps to find a minimum spanning tree in a graph with n vertices and m edges in $O(m\alpha(m, n))$ time, where α is a functional inverse of Ackermann's function [28]. In contrast to the earlier randomized linear-time algorithm [22], Chazelle's algorithm is deterministic. Pettie and Ramachandran [24] modified Chazelle's algorithm to find a minimum spanning tree deterministically in minimum time to within a constant factor. The asymptotic running time of this algorithm is still unknown (but is $O(m\alpha(m, n))$ and $\Omega(m)$). They also devised a randomized linear-time minimum-spanning-tree algorithm that uses a small number of random bits [25].

Although soft heaps are a wonderful idea, Chazelle's original version leaves room for improvement. Here we simplify the data structure and tighten its analysis. We use a binary tree representation, which we find more natural than the rooted tree representation used by Chazelle. Our implementation of `delete-min` simplifies that of Chazelle, and we do not need his *dismantling* operations, which rebuild the data structure when it becomes too unbalanced. Our analysis improves Chazelle's time bounds by making the deletions instead of the insertions the expensive operations:

each `delete-min` or `delete` takes $O(\lg \frac{1}{\varepsilon})$ amortized time, and each other operation takes $O(1)$ amortized time.

The remainder of our paper consists of seven sections. Section 2 gives an overview of our version of soft heaps. Section 3 gives a detailed implementation. Sections 4 and 5 verify that the implementation is correct and derive bounds on the amortized running time of the heap operations. Section 6 presents a purely functional implementation and discusses how to do arbitrary deletions. Section 7 discusses variants of the data structure, including Chazelle's original. Section 8 contains final remarks.

2. Overview of binary soft heaps. The building block of our data structure is the *binary tree*. Each node x has a left child $x.left$ and a right child $x.right$, either or both of which can be missing. We denote a missing child by *null*. A node is *binary*, *unary*, or a *leaf* if it has zero, one, or two missing children, respectively. We denote the parent of a node x by $x.parent$; the unique node without a parent is the *root*.

Our binary trees are *heap-ordered*: each node x has a key $x.key$ selected from the universe of item keys, such that if x is not the root, $x.key \geq x.parent.key$. Thus the key of the root is minimum among all the node keys.

The original use of a heap-ordered binary tree to implement a heap is that of Floyd [12, 13] and Williams [31]. The tree nodes contain the heap items, one item per node, with the key of a node equal to that of the item containing it. On such a tree we can do a `delete-min` as follows: Remove the item in the root. Now the root is empty. If it is a leaf, delete it; the heap is now empty. Otherwise, fill the root by comparing the keys of its children and moving the item from the child of minimum key to the root, breaking a tie arbitrarily. Set the key of the root equal to that of its newly empty child. If this child is a leaf, delete it; otherwise, fill it by applying the same method recursively.

The worst-case time for a minimum deletion is $O(h + 1)$, where h is the tree height. If the tree is balanced, $h = O(\lg n)$, and the time for a minimum deletion is $O(\lg n)$. This is too expensive for soft heaps. We reduce the amortized time for minimum deletions by allowing nodes to hold more than one item, all with key equal to the node key. We do a `delete-min` by deleting any item in the root and then filling the root if it is empty. A `delete-min` takes $O(1)$ time unless it empties the root. If filling the root moves many items into it, not just one, then the subsequent $O(1)$-time `delete-min` operations pay for the expense of the filling.

There remains the question of how to accumulate more than one item in a node. But this is easy: we can add items to a node that already contains items just as easily as if it were empty. Specifically, suppose x is a nonleaf node, empty or not. Compare the keys of the children of x. Move all items from a child of minimum key to x, breaking a tie arbitrarily. Set the key of x and of all items previously in x equal to the key of the now-empty child. If this increases $x.key$, then all items previously in x become bad. The node keys remain in heap order. If the now-empty child of x is a leaf, delete it; otherwise, fill it.

With an appropriate representation of the set of items in a node (such as a circular singly linked list), moving all the items from a node to its parent takes $O(1)$ time, independent of the number of items moved. Chazelle calls this "car-pooling." The efficiency of soft heaps comes mostly from car-pooling.

The crucial dilemma is when to do extra fillings. Doing too many creates too many bad items; doing too few makes `delete-min` too slow. Our solution, a simplification of Chazelle's, is to do a second filling at each node of even level in the tree, above a

threshold level $t = \lceil \lg \frac{3}{\varepsilon} \rceil$. Such extra fillings can cause the recursive filling process to branch, thereby filling a set of nodes that form a subtree rather than a single path. Nevertheless, a careful analysis yields the desired amortized efficiency.

To keep track of levels, we give each node x a fixed nonnegative integer *rank* $x.rank$, such that if x is a child, $x.parent.rank = x.rank + 1$. The *rank* of a tree is the rank of its root.

Maintaining tree balance during insertions and melds requires that we represent a heap by a set of trees rather than just one. We need at most one tree per rank, since we can link two trees of equal rank into a single tree as follows: Let the roots of the two trees be x and y. Create a new empty node z. Set $z.left = x$, $z.right = y$, and $z.rank = x.rank + 1$. Fill z. Return z as the root of the new tree.

We do insertions and melds in exactly the same way as in binomial queues. To do an insertion, create a new leaf of rank zero containing the item to be inserted, let the key of the new node be that of its only item, make the node into a one-node tree, make this tree into a one-tree heap, and meld this heap with the existing heap. To meld two heaps, link trees of equal rank until no two trees have equal rank.

Representing a heap by a set of trees rather than a single tree creates one additional problem: in order for `find-min` to take $O(1)$ time, we need to keep track of a root of minimum key. When a `delete-min` empties a root, we can find a new root of minimum key after the empty root is filled by scanning all the roots, but in soft heaps this takes too long. Chazelle's solution is to maintain the roots in increasing order by rank (as in binomial heaps) and to store an extra pointer in each root, the *suffix-min* pointer, indicating a root of minimum key among those of its rank or higher. Updating the *suffix-min* pointers after a root of rank k becomes empty takes $O(k+1)$ time. We use an alternative solution with equivalent efficiency. Our solution avoids the need for extra pointers. We maintain the roots in *findable order*: a root x of minimum key is first, followed by the roots of rank less than $x.rank$ in increasing order by rank, followed by the roots of rank greater than $x.rank$ in findable order.

3. Implementation of binary soft heaps. In this section we develop the details of our data structure. We postpone a discussion of arbitrary deletion to section 6; throughout sections 3 to 5, when we say "deletion" we mean minimum deletion. We represent a soft heap by a singly linked list of roots of node-disjoint binary trees of distinct ranks in findable order. If x is a root, $x.next$ is the next root on the root list containing it. We access the list by the first root: if H is a heap, H is the first root on its root list, or *null* if H is empty. The nodes of the trees contain the items in the heap, one or more items per node, with each item in exactly one node. Each node has a key that is no less than the maximum of the original keys of the items in the node and no less than the key of the parent of the node if it has a parent. The current key of an item is the key of the node containing it. Each node also has a nonnegative integer rank, which is one less than the rank of its parent if it has a parent.

In addition to a *next* pointer, we store with each node x its key $x.key$, its rank $x.rank$, pointers to its left and right children $x.left$ and $x.right$, and the set $x.set$ of items it contains. We represent each such set by a circular singly linked list of its items: if e is an item, $e.next$ is the next item in the set containing it. Access to a set of items is via the *last* item on the list: if x is a node, $x.set$ is the last item in its set, or *null* if the set is empty; if the set is nonempty, $x.set.next$ is the first item in the set. Circular linking allows us to implement set union (car-pooling) in $O(1)$ time. This representation of sets does not support a purely functional implementation of

```
Function defill(x)
    fill(x)
    if x.rank > t and (x.rank mod 2) = 0 and x.left ≠ null then
        fill(x)
```

```
Function fill(x)
    if x.left.key > x.right.key then
        x.left ↔ x.right
    x.key ← x.left.key
    if x.set = null then
        x.set ← x.left.set
    else
        x.set.next ↔ x.left.set.next
    x.left.set ← null
    if x.left.left = null then
        destroy x.left
        x.left ← x.right
        x.right ← null
    else
        defill(x.left)
```

FIG. 3.1. *Implementation of double even filling.*

soft heaps in the absence of arbitrary deletions, but in section 6 we discuss alternative representations that do.

To simplify the implementation of filling, we maintain the invariant that every unary node has a *null* right child (so a node is a leaf if and only if its left child is *null*). To simplify various tests, we define $null.key = \infty$, $null.rank = \infty$, $null.next = null$, and $null.set = null$. An easy way to implement these definitions is to create a globally defined node *Null* whose fields have the appropriate values and replace all occurrences of *null* in the code by *Null*.

The critical operation on soft heaps is `defill` (double even fill), which fills the empty root of a heap-ordered (sub-)tree, given that the tree contains at least one other node and all its other nodes are nonempty. As discussed in section 2, the recursive filling process does an extra filling at each node of even rank above rank t. We implement `defill` as a recursive function that calls a mutually recursive helper function `fill`, which fills a node once but uses `defill` for recursive fillings. Figure 3.1 contains implementations of `defill` and `fill`. (In the code, $a \leftrightarrow b$ stands for *swapping* the values of a and b. Note that $x.set.next \leftrightarrow x.left.set.next$ *catenates* the lists representing $x.set$ and $x.left.set$, assuming that $x.set$ is not empty.)

Given `defill`, the implementation of all the heap operations is straightforward and is analogous to the implementation of these operations on binomial heaps. Figure 3.2 contains implementations of the simplest operations, `make-heap` and `find-min`. Note that `find-min`, as implemented, returns a pair consisting of an item and its current, possibly increased, key. This is needed in some applications of soft heaps.

Keeping each root list in findable order is convenient for `find-min` and `delete-min`, but it is not so convenient for `insert` and `meld`. To do insertions and melds, we

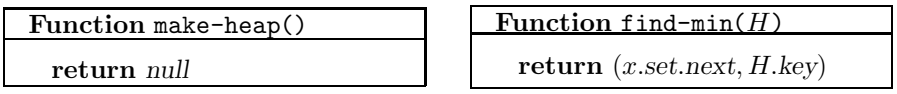

FIG. 3.2. *Implementation of* `make-heap` *and* `find-min`.

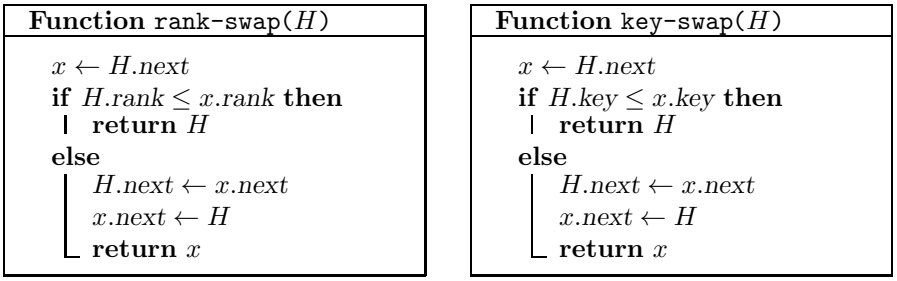

FIG. 3.3. *Implementation of* `rank-swap` *and* `key-swap`.

convert to an alternative order, *meldable order*: the root of minimum rank is first, followed by the remaining roots in findable order. Given a list H in findable order, we can convert H to meldable order by swapping the first two roots if the second has smaller rank. Given a list H in meldable order, we can convert H to findable order by swapping the first two roots if the second has smaller key. Functions `rank-swap` and `key-swap`, implemented in Figure 3.3, do these two transformations, respectively.

To do `delete-min`(H), we delete the first item in $x = H$. This completes the deletion unless it empties $x.set$. If it does, we set $k = x.rank$ and fill x if it is not a leaf or destroy it if it is. Now the list consists of root x (if not destroyed), followed by roots of rank less than k in increasing order by rank, followed by roots of rank greater than k in findable order. To complete the deletion, we need to restore findable order on the roots of rank at most k. To do this, we traverse the list of roots until reaching a root of rank greater than k, using `rank-swap` to move x past all nodes of smaller rank if it still exists. Now all the roots of rank at most k are in increasing rank order. Then we back up along the list of roots, using `key-swap` to restore findable order. Our implementation of `delete-min` uses a recursive function `reorder` to do the list traversal and reordering. Figure 3.4 contains implementations of `delete-min` and `reorder`.

Insertion uses two auxiliary functions. The first, `make-root`(e), returns a new root of rank zero containing the item e with predefined key, not in any other node. The second, `meldable-insert`(x, H), is also used in melding. Its inputs are a root x and a list of roots H in meldable order such that $x.rank \leq H.rank$. This function links x with roots in H of successively higher rank, until H is empty or all its roots have rank greater than that of the current x. Then it converts what is left of H from meldable order to findable order, adds the current x to the front, and returns the result, which is in meldable order. Figure 3.5 contains implementations of `insert`, `meldable-insert`, and `link`.

Melding uses a recursive function `meldable-meld` to do the actual melding. This function is given two root lists in meldable order. It melds them and returns the result in meldable order. It does this by removing from the lists a root x of minimum rank, melding the remainders by calling itself recursively, and inserting x into

```
Function delete-min(H)
    e ← H.set.next
    if e.next ≠ e then
        H.set.next ← e.next
        return H
    else
        H.set ← null
        k ← H.rank
        if H.left = null then
            L ← H.next
            destroy H
            H ← L
        else
            defill(H)
        return reorder(H, k)
```

```
Function reorder(H, k)
    if H.next.rank < k then
        H ← rank-swap(H)
        H.next ← reorder(H.next, k)
    return key-swap(H)
```

FIG. 3.4. *Implementation of* `delete-min` *and* `reorder`.

```
Function insert(e, H)
    return key-swap(meldable-insert(make-root(e), rank-swap(H)))
```

```
Function meldable-insert(x, H)
    if x.rank < H.rank then
        x.next ← key-swap(H)
        return x
    else
        return meldable-insert(link(x, H), rank-swap(H.next))
```

```
Function make-root(e)
    x ← new-node()
    e.next ← e
    x.set ← e
    x.key ← e.key
    x.rank ← 0
    x.left ← null
    x.right ← null
    x.next ← null
    return x
```

```
Function link(x, y)
    z ← new-node()
    z.set ← null
    z.rank ← x.rank + 1
    z.left ← x
    z.right ← y
    defill(z)
    return z
```

FIG. 3.5. *Implementation of* `insert`, `meldable-insert`, *and* `link`.

the result using meldable-insert. Figure 3.6 contains implementations of `meld` and `meldable-meld`.

In our implementation of the heap operations we have aimed for clarity and short code length. There are alternatives that may save up to a constant factor in running time, but these seem to require increasing the code length. These include expanding

```
Function meld(H1, H2)
    return key-swap(meldable-meld(rank-swap(H1), rank-swap(H2)))
```

```
Function meldable-meld(H1, H2)
    if H1.rank > H2.rank then
        H1 ↔ H2
    if H2 = null then
        return H1
    else
        return meldable-insert(H1, meldable-meld(rank-swap(H1.next), H2))
```

FIG. 3.6. *Implementations of* `meld` *and* `meldable-meld`.

the `fill` function in-line to eliminate the mutual recursion and allowing each list of roots to be in either findable or meldable order. There are also alternatives that may save space. For example, node ranks need be maintained explicitly only for roots, since the rank of a nonroot can be computed while traversing the path to it from the root of the tree containing it. Also, nonroots do not need next pointers. Thus it may make sense to have different representations of roots and nonroots. We leave all such improvements to the expert programmer (or the optimizing compiler).

A python implementation of soft heaps, essentially identical to the pseudocode given here, can be found at http://www.cs.tau.ac.il/∼zwick/softheap.txt.

4. Analysis of double even filling. In this section we analyze double even filling. First we show that the number of bad items is at most εm. This implies that our implementation of soft heaps is correct. Then we bound the number of fillings as a function of node rank. This allows us to derive a bound on the total deletion time. The key to these results is that the exponential growth in the `defill` recursion above rank t (two calls on nodes of rank $k-2$ for each call on a node of rank k) is swamped by the faster exponential growth in the number of nodes (four nodes of rank $k-2$ for each node of rank k). Recall that $t = \lceil \lg \frac{3}{\varepsilon} \rceil$.

LEMMA 4.1. *The number of nodes of rank k is at most $m/2^k$.*

Proof. Each node of rank zero is created by an insertion, of which there are m. Each creation of a node of rank $k+1$, by a link operation, converts two roots of rank k into nonroots. Since a nonroot can never become a root, this implies that the number of nodes of rank $k+1$ is at most half the number of nodes of rank k. The lemma follows by induction on k. □

Lemma 4.1 implies that all node ranks are at most $\lg m$, but we do not use this fact directly in our analysis of soft heaps.

We define the *rank* $e.rank$ of an item e to be the rank of the node that contains it, and the *size* $x.size$ of a node x to be the number of items it contains. We use these definitions in our analysis only; the data structure does not store item ranks or node sizes.

Let $s(k)$ be the maximum size of a node of rank k.

LEMMA 4.2. *If $k \le t$, $s(k) = 1$. Suppose $k > t$. Then $s(k) \le s(k-1)$ if k is odd, and $s(k) \le 2s(k-1)$ if k is even.*

Proof. Let x be a node of rank k. If $k = 0$, x has size 1 when it is created, and it is never filled, so the lemma holds. If $k \le t$, or $k > t$ and k is odd, x is filled by

calling `fill` once, which moves the items from a node of rank $k-1$ to x. It follows that $s(k) \le s(k-1)$. By induction, $s(k) = 1$ if $k < t$. If $k > t$ and k is even, x is filled by calling `fill` at most twice, which implies $s(k) \le 2s(k-1)$. □

COROLLARY 4.3. *If $k > t$, $s(k) \le 2^{\lceil (k-t)/2 \rceil}$.*

Proof. A proof follows from Lemma 4.2 by induction on k. □

LEMMA 4.4. *The number of items of rank greater than t is at most εm.*

Proof. By Lemma 4.1 and Corollary 4.3, the number of items of rank greater than $t = \lceil \lg \frac{3}{\varepsilon} \rceil$ is at most

$$\sum_{k>t} \frac{m}{2^k} \cdot s(k) \;\le\; \frac{m}{2^t} \sum_{i \ge 1} \left(\frac{1}{2^{2i-1}} + \frac{1}{2^{2i}} \right) \cdot 2^i = \frac{m}{2^t} \sum_{i \ge 1} \frac{3}{2^i} = \frac{3m}{2^t} \;\le\; \varepsilon m \,,$$

since Corollary 4.3 implies that $s(t+2i-1), s(t+2i) \le 2^i$, for $i \ge 1$. □

THEOREM 4.5. *The number of bad items is at most εm.*

Proof. By Lemma 4.2, only items of rank greater than t can be bad. By Lemma 4.4 there are at most εm such items. □

Now we study the number of fillings as a function of node rank. We call a filling of a node of rank k *low* if $k \le t$ and *high* if $k > t$. Our bound on the number of low fillings is what gives our improvement of Chazelle's bound on the total deletion time.

LEMMA 4.6. *The number of low fillings is at most $td + O(m)$.*

Proof. We charge each low filling to the item moved by the filling. Such an item increases in rank by one. Once an item's rank is at least t, the item can no longer be charged. Thus each item can accumulate a charge of at most t. When we delete an item, we transfer its charge to the deletion. The total charge to deletions is at most td. The rest of the charge is to items that are never deleted. By Lemmas 4.1 and 4.2, the total charge to such items of rank at most t is at most

$$\sum_{k=1}^{t} k \cdot \frac{m}{2^k} = O(m) \,.$$

By Lemma 4.4, the total charge to such items of rank greater than t is at most

$$t \cdot \varepsilon m = \lceil \lg \tfrac{3}{\varepsilon} \rceil \cdot \varepsilon m = O(m) \,,$$

since, $\varepsilon \lceil \lg \frac{3}{\varepsilon} \rceil < 3$ for every $0 < \varepsilon < 1$. □

Our bound on the number of fillings as a function of rank is like Lemma 4.2 and Corollary 4.3. For $k \ge 1$, let $f(k)$ be the maximum number of fillings of a node of rank k, i.e., the maximum over all nodes x of rank k of the number of times `fill`(x) is called. Let $f(0) = 1$. (This corresponds to the "filling" of a node of rank 0 by `make-root`.)

LEMMA 4.7. *For every $k \ge 1$, $f(k) \le 2f(k-1)$. If $k > t$ and k is even, then $f(k) \le f(k-1)$.*

Proof. Let $g(k)$ be the maximum number of times a node x of rank k is filled or destroyed when it is no longer a root. Since the first filling of a node occurs immediately after its creation as a root, and since a node is destroyed at most once, we have $g(k) \le f(k)$. Let x be a node of rank $k \ge 1$. When x is first created, it has two children of rank $k-1$. Each filling of x triggers either a filling or a destruction of a child of x. Hence $f(k) \le 2g(k-1) \le 2f(k-1)$. If $k > t$ and k is even, then each filling of x, except possibly the last one, triggers two operations on children of x, each of which is either a filling or a destruction. The last filling of x, which

may occur when x has a single leaf child, may trigger only one such operation. Thus $2f(k) - 1 \leq 2g(k-1) \leq 2f(k-1)$. Since $f(k)$ and $f(k-1)$ are integers, $f(k) \leq f(k-1)$. ∎

COROLLARY 4.8. *If $k \leq t$, $f(k) \leq 2^k$. If $k > t$, $f(k) \leq 2^{\lceil (k+t)/2 \rceil}$.*

Proof. A proof follows from Lemma 4.7 by induction on k. ∎

LEMMA 4.9. *The number of high fillings is at most $3m$.*

Proof. By Lemma 4.1 and Corollary 4.8, the number of high fillings is at most

$$\sum_{k>t} \frac{m}{2^k} \cdot f(k) \;\leq\; \frac{m}{2^t} \sum_{i \geq 1} \left(\frac{1}{2^{2i-1}} + \frac{1}{2^{2i}} \right) \cdot 2^{t+i} = m \sum_{i \geq 1} \frac{3}{2^i} = 3m \,,$$

since Corollary 4.8 implies that $f(t+2i-1), f(t+2i) \leq 2^{t+i}$, for $i \geq 1$. ∎

5. Running time. In this section we verify the claimed running times of the heap operations. Each `make-heap` or `find-min` operation takes $O(1)$ time worst-case. The time bounds of the remaining operations are amortized.

LEMMA 5.1. *The time spent doing fillings is $O(td + m)$.*

Proof. The time for a call of `defill` is $O(1)$, not counting the time spent doing recursive calls. By Lemmas 4.6 and 4.9 the number of calls of `defill` is $O(td + m)$. ∎

LEMMA 5.2. *Excluding filling, the time spent doing deletions is $O(td + m)$.*

Proof. A deletion that does not empty a root takes $O(1)$ time. A deletion that empties a root of rank k takes $O(k+1)$ time excluding filling, to destroy the emptied root if it cannot be filled and to reorder the roots of rank at most k in findable order. We scale this time so that it is $k+1$ and charge $\min\{t+1, k+1\}$ of it to the deletion. The total charge to deletions is $O(td)$. The uncharged time is $k - t$ for each deletion that empties a root x of rank $k > t$. Node x can be emptied at most $f(k)$ times: it must be filled before it can become empty. By Lemma 4.1 and Corollary 4.8, the total uncharged deletion time is at most

$$\begin{aligned} \sum_{k>t} \frac{m}{2^k} \cdot f(k) \cdot (k-t) &= \frac{m}{2^t} \sum_{i \geq 1} \left(\frac{2i-1}{2^{2i-1}} + \frac{2i}{2^{2i}} \right) 2^{t+i} \\ &= m \sum_{i \geq 1} \left(\frac{2i-1}{2^{i-1}} + \frac{2i}{2^i} \right) = O(m). \end{aligned}$$ ∎

LEMMA 5.3. *Excluding filling, the time spent doing insertions and melds is $O(1)$ per insertion or meld, amortized.*

Proof. We use a potential function argument [27]. We define the potential of a heap to be the number of trees plus the maximum node rank. We define the potential of a collection of heaps to be the sum of their potentials. We define the *amortized time* of an operation to be its actual time plus the increase in potential it causes. Then, for an arbitrary sequence of heap operations starting with no heaps, the total time of the operations is the sum of their amortized times plus the initial potential minus the final potential. Since the initial potential is zero and the final potential is nonnegative, the sum of the amortized times is an upper bound on the sum of the actual times.

Creation of an empty heap has no effect on the potential. Consider a meld of two heaps H_1 and H_2, containing h_1 and h_2 trees and having maximum node ranks r_1 and r_2, respectively. Assume without loss of generality that $r_1 \leq r_2$. Suppose the meld does k links. The meld takes $O(r_1 + k + 1)$ time. The potential of H_1 and H_2

before the meld is $(h_1 + r_1) + (h_2 + r_2)$. The potential of the heap resulting from the meld is at most $(h_1 + h_2 - k) + (r_2 + 1)$, since each link reduces the number of trees by one and the rank of the output heap is at most $r_2 + 1$. Scale the meld time so that it is at most $r_1 + k + 1$. (This is equivalent to multiplying the potential by a constant factor.) The amortized time of the meld is then at most

$$(r_1 + k + 1) \ + \ ((h_1 + h_2 - k) + (r_2 + 1)) \ - \ ((h_1 + r_1) + (h_2 + r_2)) = 2 = O(1).$$

An insertion creates one new tree of rank zero, which increases the potential by one and takes $O(1)$ time plus a meld, so its amortized time is also $O(1)$.

Finally, since delete operations do not create new trees and do not increase ranks, they can only decrease the potential of the data structure. □

THEOREM 5.4. *The amortized time per heap operation is* $O(\lg \frac{1}{\varepsilon})$ *per deletion and* $O(1)$ *for each other operation.*

Proof. A proof is immediate from Lemmas 5.1, 5.2, and 5.3. □

6. Extensions. In this section we extend our data structure in two different ways. First we describe how to make the implementation purely functional [1, 23]. Then we describe how to implement arbitrary deletions in $O(\lg \frac{1}{\varepsilon})$ amortized time per deletion without affecting the amortized time bounds of the other operations.

The implementation in section 3 is imperative, not functional: it modifies fields of existing nodes. We can make the implementation functional by the standard technique of creating new nodes instead of modifying existing ones. We use the following strict definition of "functional": Given a node, we can extract and return any of its fields (function evaluation); given a fixed number of values, we can construct and return a node containing these values in its fields (function extension). No modification or destruction of existing nodes is permitted. It is easy to verify that when the implementation of section 3 modifies a node x, e.g., by a `fill`, it also visits all nodes on the path from the root y of the tree containing x to x, and all roots on the root list from the first root to y, and hence can afford to create new versions of them.

The only remaining problem is the handling of the items. In the nonfunctional implementation of section 3 the items in a node are kept in a circular singly linked list. We need to replace this implementation of sets by a purely functional one. The implementation must support the following set operations, each in $O(1)$ amortized time:

`make-set`(e): Return a new set containing the single element e, previously in no set.

`unite`(X, Y): Return the union of disjoint sets X and Y.

`delete-any`(X): Return a pair consisting of any element e in set X and $X - \{e\}$; if X is empty, return $(null, null)$.

One solution is to represent each set by a stack of its elements: each `unite` becomes catenation of stacks, each `delete-any` becomes a stack pop. Implementation of each stack by a heterogeneous finger search tree [29] gives a solution in which each operation takes $O(1)$ amortized time. Such a tree consists of a standard balanced binary search tree in which the child pointers along the left and right spines (the paths from the root to the first and last nodes in symmetric order) are reversed. Such a tree is accessed via two pointers, to the first and last nodes in symmetric order. In such a tree, every node except the root has exactly one incoming pointer, but the root has two, from its left and right children. To make this data structure purely functional, we need to remove this anomaly, which we can do by making the right child of the root point to *null* instead of to the root. This splits the tree into two

trees, one containing the root and all smaller nodes in symmetric order, accessed from the first node, the other containing the rest of the nodes, accessed from the last node. Updates require concurrent bottom-up traversal of the left and right spines of the original tree. Another solution with an $O(1)$ amortized time bound per operation is to use ordered path compression on compressed trees [6]. A more complicated solution with an $O(1)$ worst-case time bound per operation is based on recursive slow-down [20]. Any of these solutions yields a purely functional implementation of soft heaps.

Such a purely functional implementation preserves the input heaps of each update operation rather than destroying them. This suggests the possibility of implementing *persistent* soft heaps, in which any operation can be applied to any heap or heaps ever created, not just those previously created but not yet modified. For a discussion of persistent data structures, see [11, 18]. There are two problems with obtaining persistent soft heaps via such a functional implementation. The first is that our amortized time bounds break down: they require that each heap ever created be an input to at most one update operation. The second is that there is no easy way to tell whether two heaps are item-disjoint and hence meldable. We leave the exploration of persistent soft heaps to the future.

It is easy to extend the imperative implementation of soft heaps to support deletion of arbitrary items, assuming that one is given a pointer to the location of the item to be deleted. (We know of no purely functional implementation of arbitrary deletion that preserves the $O(\lg \frac{1}{\varepsilon})$ time bound.) One possibility is to use lazy deletion: mark each deleted item; when finding or deleting the minimum, delete each marked item encountered, continuing until finding an unmarked item or emptying the entire heap. Each deletion takes $O(1)$ time for marking and $O(\lg \frac{1}{\varepsilon})$ amortized time when the item is actually deleted.

Lazy deletion is the method used by Chazelle. It has the disadvantage that eventually a heap can consist mostly of deleted items, so the space needed can become superlinear in the number of undeleted items. One can avoid this either by rebuilding the entire data structure after a constant fraction of the inserted items have been deleted, or by doing deletions eagerly. Doing deletions eagerly requires doubly linking the list of items in a node, adding a pointer from the first item in a node to the node itself, adding a pointer from each node to its parent node, and doubly linking the list of trees in a heap. Then an item can be deleted in $O(1)$ time, as can an empty leaf. A deletion takes $O(1)$ time unless it empties a node x. If it does, there are three cases. If x is a root, proceed as in `delete-min`. Otherwise, fill x if it is not a leaf, or delete it if it is. An extension of the analysis in sections 4 and 5 shows that at most εm items are bad, and the amortized time of an arbitrary deletion done eagerly is $O(\lg \frac{1}{\varepsilon})$.

7. Variants. We have developed a version of soft heaps intended to be simple to implement, simple to analyze, and with small constants in the analysis. But there are many variants with the same asymptotic bounds. In this section we briefly discuss some of these variants, including Chazelle's original solution. At least three ideas can be used separately or in combination to obtain such variants: handling unary nodes differently, filling lazily, and changing the rule for extra filling.

When a unary node x becomes empty, instead of filling x from its only child, say y, we can merely replace x by y. An alternative that is almost the same is to fill x from y and then replace the children of x by the children of y. The only difference between these methods is that the latter preserves the rank of x; the former has the effect of replacing the rank of x by that of y. In either case rank differences are no longer exactly one, which makes proving an analogue of Corollary 4.8 harder. Chazelle's

implementation handles unary nodes in the latter way. It does an extra filling of a node x if the rank of x is odd and above the threshold rank t (equivalent to double even filling) or if the rank of x exceeds that of its children by at least two. It also dismantles and rebuilds trees that have become too unbalanced as a result of node deletions. Neither the additional test for an extra filling nor the dismantling is in fact necessary, since an analogue of Corollary 4.8 holds for Chazelle's method without these additions, but Chazelle did not prove such a result.

Instead of filling every node when it becomes empty, we can do filling more lazily. Specifically, we can allow each binary node to have one empty child. When a new root is created by a link, we fill it from a child of minimum key and do no more fillings, neither an extra filling of the root nor any recursive fillings. If the empty child is a leaf, we delete it; otherwise, we leave it empty. To fill a binary node with an empty child, we first fill the empty child, then the binary node. We call this method *lazy filling* and the original method (described in sections 2 and 3) *eager filling*. Lazy and eager filling give the same amortized time bounds; the only advantage of lazy filling is that the time for a link becomes $O(1)$ worst-case. We can use lazy filling in combination with the original way of handling unary nodes or with either of the two methods described above.

Instead of doing extra fillings based on rank, we can do them based on recursion depth or node size. A recursion-depth-based rule is to do an extra filling at alternate levels of the filling recursion above rank t. In combination with lazy filling, this gives a correct implementation of soft heaps, but with bigger bounds in the analogues of Corollaries 4.3 and 4.7 and a much more complicated proof of the analogue of Corollary 4.8. A size-based rule is to repeatedly fill a node of rank $k > t$ until its size is at least b^{k-t} for some constant b such that $1 < b < 2$. The conference version of our paper used this rule. It has at least two disadvantages: node sizes must be maintained explicitly, and an empty node may require an arbitrary number of fillings, not just one or two.

8. Remarks. We have presented a simplified version of Chazelle's soft heaps that uses a constant factor less space and has a simpler, improved analysis. Several open problems remain. Can our time bounds for soft heaps be made worst-case? Is there a persistent implementation of soft heaps with our time bounds, either amortized or worst-case? Is there a version of soft heaps that supports decrease-key operations in $O(1)$ time? This question depends on the semantics of decrease-key and the definition of m: in the simplest version, m is the total number of insertions and decrease-key operations, and a bad item stays bad if its key is decreased (so decreasing the key of a bad item has no effect). What version of soft heaps gives the best trade-off between efficiency and accuracy if we measure efficiency by counting comparisons? Can soft heaps be used to improve the bound on the worst-case number of comparisons needed to find the median of n items, currently $2.95n$ [10]? Can soft heaps be used to give a faster algorithm for optimum branchings [15] or minimum bottleneck branchings [16]? Are there other applications of soft heaps?

REFERENCES

[1] J.W. Backus, *Can programming be liberated from the von neumann style? A functional style and its algebra of programs*, Comm. ACM, 21 (1978), pp. 613–641.

[2] M. Blum, R.W. Floyd, V. Pratt, R.L. Rivest, and R.E. Tarjan, *Time bounds for selection*, J. Comput. System Sci., 7 (1973), pp. 448–461.

[3] G.S. Brodal, *Fast meldable priority queues*, in Proceedings of the 4th International Workshop on Algorithms and Data Structures, Lecture Notes in Comput. Sci. 995, Springer-Verlag, 1995, pp. 282–290.
[4] G.S. Brodal, *Worst-case efficient priority queues*, in Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms, ACM, New York, 1996, pp. 52–58.
[5] G.S. Brodal, G. Lagogiannis, and R.E. Tarjan, *Strict Fibonacci heaps*, in Proceedings of the 44th Annual ACM Symposium on Theory of Computing, ACM, New York, 2012.
[6] A.L. Buchsbaum, R. Sundar, and R.E. Tarjan, *Data-structural bootstrapping, linear path compression, and catenable heap-ordered double-ended queues*, SIAM J. Comput., 24 (1995), pp. 1190–1206.
[7] B. Chazelle, *The Discrepancy Method: Randomness and Complexity*, Cambridge University Press, New York, 2000.
[8] B. Chazelle, *A minimum spanning tree algorithm with inverse-Ackermann type complexity*, J. ACM, 47 (2000), pp. 1028–1047.
[9] B. Chazelle, *The soft heap: An approximate priority queue with optimal error rate*, J. ACM, 47 (2000), pp. 1012–1027.
[10] D. Dor and U. Zwick, *Selecting the median*, SIAM J. Comput., 28 (1999), pp. 1722–1758.
[11] J.R. Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan, *Making data structures persistent*, J. Comput. System Sci., 38 (1989), pp. 86–124.
[12] R.W. Floyd, *Algorithm* 113: *Treesort*, Comm. ACM, 5 (1962), p. 434.
[13] R.W. Floyd, *Algorithm* 245: *Treesort*, Comm. ACM, 7 (1964), p. 701.
[14] M.L. Fredman and R.E. Tarjan, *Fibonacci heaps and their uses in improved network optimization algorithms*, J. ACM, 34 (1987), pp. 596–615.
[15] H.N. Gabow, Z. Galil, T.H. Spencer, and R.E. Tarjan, *Efficient algorithms for finding minimum spanning trees in undirected and directed graphs*, Combinatorica, 6 (1986), pp. 109–122.
[16] H.N. Gabow and R.E. Tarjan, *Algorithms for two bottleneck optimization problems*, J. Algorithms, 9 (1988), pp. 411–417.
[17] B. Haeupler, S. Sen, and R.E. Tarjan, *Rank-pairing heaps*, SIAM J. Comput., 40 (2011), pp. 1463–1485.
[18] H. Kaplan, C. Okasaki, and R.E. Tarjan, *Simple confluently persistent catenable lists*, SIAM J. Comput., 30 (2000), pp. 965–977.
[19] H. Kaplan and R.E. Tarjan, *Thin heaps, thick heaps*, ACM Trans. Algorithms, 4 (2008), pp. 1–14.
[20] H. Kaplan and R.E. Tarjan, *Purely functional, real-time deques with catenation*, J. ACM, 46 (1999), pp. 577–603.
[21] H. Kaplan and U. Zwick, *A simpler implementation and analysis of Chazelle's soft heaps*, in Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms, ACM, New York, 2009, pp. 477–485.
[22] D.R. Karger, P.N. Klein, and R.E. Tarjan, *A randomized linear-time algorithm for finding minimum spanning trees*, J. ACM, 42 (1995), pp. 321–328.
[23] C. Okasaki, *Purely Functional Data Structures*, Cambridge University Press, Cambridge, 1998.
[24] S. Pettie and V. Ramachandran, *An optimal minimum spanning tree algorithm*, J. ACM, 49 (2002), pp. 16–34.
[25] S. Pettie and V. Ramachandran, *Randomized minimum spanning tree algorithms using exponentially fewer random bits*, ACM Trans. Algorithms, 4 (2008), pp. 1–27.
[26] A. Schönhage, M. Paterson, and N. Pippenger, *Finding the median*, J. Comput. System Sci., 13 (1976), pp. 184–199.
[27] R.E. Tarjan, *Amortized computational complexity*, SIAM J. Algebraic Discrete Methods, 6 (1985), pp. 306–318.
[28] R.E. Tarjan, *Efficiency of a good but not linear set union algorithm*, J. ACM, 22 (1975), pp. 215–225.
[29] R.E. Tarjan and C.J. van Wyk, *An $O(n \log\log n)$-time algorithm for triangulating a simple polygon*, SIAM J. Comput., 17 (1988), pp. 143–178.
[30] J. Vuillemin, *A data structure for manipulating priority queues*, Comm. ACM, 21 (1978), pp. 309–314.
[31] J.W.J. Williams, *Algorithm* 232: *Heapsort*, Comm. ACM, 7 (1964), pp. 347–348.