The Soft Heap: An Approximate Priority Queue with Optimal Error Rate*

Bernard Chazelle†

NECI Research Tech Report 99-097 (July 1999)
Journal of the ACM, 47(6), 2000, pp. 1012–1027.

Abstract

A simple variant of a priority queue, called a *soft heap*, is introduced. The data structure supports the usual operations: insert, delete, meld, and findmin. Its novelty is to beat the logarithmic bound on the complexity of a heap in a comparison-based model. To break this information-theoretic barrier, the entropy of the data structure is reduced by artificially raising the values of certain keys. Given any mixed sequence of n operations, a soft heap with error rate ε (for any $0 < \varepsilon \leq 1/2$) ensures that, at any time, at most εn of its items have their keys raised. The amortized complexity of each operation is constant, except for insert, which takes $O(\log 1/\varepsilon)$ time. The soft heap is optimal for any value of ε in a comparison-based model. The data structure is purely pointer-based. No arrays are used and no numeric assumptions are made on the keys. The main idea behind the soft heap is to move items across the data structure not individually, as is customary, but in groups, in a data-structuring equivalent of "car pooling." Keys must be raised as a result, in order to preserve the heap ordering of the data structure. The soft heap can be used to compute exact or approximate medians and percentiles optimally. It is also useful for approximate sorting and for computing minimum spanning trees of general graphs.

1 Introduction

We design a simple variant of a priority queue, called a *soft heap*. The data structure stores items with keys from a totally ordered universe, and supports the operations:

- `create` $(\mathcal{S})$: Create an empty soft heap $\mathcal{S}$.
- `insert` $(\mathcal{S}, x)$: Add new item x to $\mathcal{S}$.
- `meld` $(\mathcal{S}, \mathcal{S}')$: Form a new soft heap with the items stored in $\mathcal{S}$ and $\mathcal{S}'$ (assumed to be disjoint), and destroy $\mathcal{S}$ and $\mathcal{S}'$.
- `delete` $(\mathcal{S}, x)$: Remove item x from $\mathcal{S}$.

*A preliminary version of this paper appeared in *"Car-pooling as a data structuring device: the soft heap"*, by Bernard Chazelle, Proc. 6th Ann. Euro. Symp. Alg. (1998), 35–42. This work was supported in part by NSF Grant CCR-93-01254, NSF Grant CCR-96-23768, ARO Grant DAAH04-96-1-0181, and NEC Research Institute.

†Department of Computer Science, Princeton University, and NEC Research Institute, chazelle@cs.princeton.edu and chazelle@research.nj.nec.com

- `findmin` $(\mathcal{S})$: Return an item in $\mathcal{S}$ with the smallest key.

The soft heap may, at any time, increase the value of certain keys. Such keys, and by extension, the corresponding items, are called *corrupted*. Corruption is entirely at the discretion of the data structure and the user has no control over it. Naturally, `findmin` returns the minimum *current* key, which might or might not be corrupted. The benefit is speed: during heap updates, items travel together in packets in a form of "car pooling," in order to save time.

From an information-theoretic point of view, corruption is a way to decrease the entropy of the data stored in the data structure, and thus facilitate its treatment. The entropy is defined as the logarithm, in base two, of the number of distinct key assignments (ie, entropy of the uniform distribution over key assignments). To see the soundness of this idea, push it to its limit, and observe that if every key was corrupted by raising its value to ∞, then the set of keys would have zero entropy and we could trivially perform all operations in constant time. Interestingly, soft heaps show that the entropy need not drop to zero for the complexity to become constant.

Theorem 1.1 *Beginning with no prior data, consider a mixed sequence of operations that includes n inserts. For any $0 < \varepsilon \leq 1/2$, a soft heap with error rate ε supports each operation in constant amortized time, except for insert, which takes $O(\log 1/\varepsilon)$ time. The data structure never contains more than εn corrupted items at any given time. In a comparison-based model, these bounds are optimal.*

Note that this does not mean that only εn items are corrupted in total throughout the sequence of operations. Because of deletes many more items might end up being corrupted. In fact, it is not difficult to imagine a scenario where all items are eventually corrupted; for example, insert n items and then keep deleting corrupted ones. Despite this apparent weakness, the soft heap is optimal and—perhaps even more surprising—useful. The data structure can be implemented on a pointer machine: no arrays are used, and no numeric assumptions on the keys are required. Soft heaps capture three distinct features, which it is useful to understand at the outset.

- If we set $\varepsilon = 1/2n$, then no corruption is allowed to take place and the soft heap behaves like a regular heap with logarithmic insertion time. Unsurprisingly, soft heaps include standard heaps as special cases. In fact, as we shall see, a soft heap is nothing but a modified binomial queue [5].

- More interesting is the fact that soft heaps implicitly feature median-finding technology. To see why, set ε to be a small constant: insert n keys and then perform $\lfloor n/2 \rfloor$ findmins, each one followed by a delete. This takes $O(n)$ time. Among the keys deleted, the largest (original) one is εn away from the median of the n original keys. To obtain such a number in linear time (deterministically), as we just did, typically requires a variant of the median-finding algorithm of Blum et al. [1].

- The previous remark should not lead one to think that a soft heap is simply a dynamic median-finding data structure. Things are more subtle. Indeed, consider a sequence of n inserts of keys in decreasing order, intermixed with n findmins, and set $\varepsilon = 1/2$. Despite the high value of the error rate ε, the findmins must actually return the minimum key at least half the time. The reason is that at most $n/2$ keys inserted can ever be corrupted. Because of the decreasing order of the insertions, these uncorrupted keys must be reported by findmin right after their insertion since they are minimum at the time. The requirement to be correct half the time dooms any strategy based on maintaining medians or near-medians for the purpose of findmin.

2 Applications

Soft heaps are useful for computing minimum spanning trees and percentiles, for finding medians, for near sorting, and generally for situations where approximate rank information is sought. Some examples below:

1. The soft heap was designed with a specific application in mind, minimum spanning trees. It is a key ingredient in what is currently the fastest deterministic algorithm [2] for computing the minimum spanning tree of a graph. Given a connected graph with n vertices and m weighted edges, the algorithm finds a minimum spanning tree in time $O(m\alpha(m,n))$, where α is the classical functional inverse of Ackermann's function.

2. Another, simpler application is the dynamic maintenance of percentiles. Suppose we wish to maintain the grade point averages of students in a college, so that at any time we can request the name of a student with a GPA in the top percentile. Soft heaps support such operations in constant amortized time.

3. Soft heaps give an alternative method for computing medians in linear time (or generally perform linear-time selection [1]). Suppose we want to find the k-th largest element in a set of n numbers. Insert the n numbers into a soft heap with error rate $1/3$. Next, call findmin and delete about $n/3$ times. The largest number deleted has rank between $n/3$ and $2n/3$. After computing this rank, we can therefore remove at least $n/3$ numbers from consideration. We recurse over the remainder in the obvious fashion. This allows us to find the k-th largest element in time proportional to $n + 2n/3 + (2/3)^2 n + \cdots = O(n)$.

4. A fourth application of soft heaps is to approximate sorting. A weak version of near-sorting requires that given n distinct numbers, the algorithm should output them in a sequence whose number of inversions is at most εn^2 (instead of zero for exact sorting). As it turns out, this follows directly from inserting n numbers into a soft heap with error rate ε and then deleting the smallest keys repeatedly. The number I_k of inversions of the k-th deleted number x is the number of keys deleted earlier whose original values are larger than x. But x must have been in a corrupted state during those particular I_k earlier deletions. The total number of keys in a corrupted state, counting over all deletions, is at most εn^2, and so the number of inversions $\sum I_k$ is also bounded by εn^2.

5. A stronger version of near-sorting requires that in the output sequence the rank of no number differs from its true rank in sorted order by more than εn. We show below how soft heaps allow us to do that in $O(n \log 1/\varepsilon)$ time. In particular, this gives us a simple linear time algorithm for this strong form of near-sorting with, say, 1% error in rank. Of course, the result itself is not new. It can be derived trivially from repeated median computation. The whole point is that we do it in a completely different way.

Theorem 2.1 *For any $0 < \varepsilon \leq 1/2$, we can use a soft heap to near-sort n numbers in time $O(n \log 1/\varepsilon)$: this means that the rank of any number in the output sequence differs from its true rank by at most εn.*

Proof: Since the running time we seek to achieve is $O(n \log 1/\varepsilon)$ we can assume that n is large enough and that $1/\varepsilon$ lies between a large constant and $\sqrt{n}$. Given the n numbers (assumed to be distinct for simplicity) insert them into a soft heap and then delete the item returned by findmin until the heap is empty. Using a soft heap with error rate ε, at most εn numbers are corrupted at any given time. Divide the sequence of n pairs (findmin, delete) into time intervals $T_1, \ldots, T_l$, each consisting of $\lceil 2\varepsilon n \rceil$ pairs; without loss of generality, we may assume that $n = \lceil 2\varepsilon n \rceil l$. Let S_i be the set of items deleted during T_i, and let U_i be the subset of S_i that includes only items that were at some point uncorrupted during T_i. (We assume that items are time-stamped when first corrupted.) Finally, let x_i be the smallest original key among the items of U_i, and let s_i be the corresponding item; for convenience, we put $x_0 = -\infty$ and $x_{l+1} = \infty$. Given an item $s \in S_i$ whose original key lies in $[x_j, x_{j+1})$, we define $\rho(s) = |i - j|$. Some simple facts:

(1) $|U_i| \geq \lceil 2\varepsilon n \rceil - \varepsilon n \geq \varepsilon n$: Because at most εn items are corrupted at the beginning of T_i.

(2) The x_i's appear in increasing order, and the original key of any item in U_i lies in $[x_i, x_{i+1})$: Since s_{i+1} is uncorrupted during T_i, its original key x_{i+1} is at least the current (and hence, the original) key of any item deleted during T_i.

(3) $\sum\{\, \rho(s) \mid s \in S_i \setminus U_i \,\} < 2n$: Given $s \in S_i \setminus U_i$, let $[x_j, x_{j+1})$ be the interval that contains the original key of s. As we just observed, the original key of s is less than x_{i+1}, and therefore, $j \leq i$. To avoid being selected by findmin, the item s must have been in a corrupted state during the deletion of x_k, for any $j < k < i$ (if any such k exists). The total number of items in a corrupted state during the deletions of $x_1, \ldots, x_l$ is at most $\varepsilon n l$, and therefore so is the sum of distances $i - j - 1 = \rho(s) - 1$ over all such items s. It follows that $\sum \rho(s) \leq \varepsilon n l + n < 2n$, hence our claim.

(4) The number of items whose original keys fall in $[x_i, x_{i+1})$ is less than $6\varepsilon n$: Indeed, suppose that the item does not belong to $S_i \cup S_{i+1}$. It cannot be selected by findmin and deleted before the beginning of T_i, since s_i was not corrupted yet. By the time s_{i+1} was deleted, the item in question must have been corrupted (it cannot have been deleted yet). So, there can be at most εn such items. Thus, the total number of items with original keys in $[x_i, x_{i+1})$ is at most $2\lceil 2\varepsilon n \rceil + \varepsilon n < 6\varepsilon n$.

Next, for each item $s \in S_i \setminus U_i$, we search which interval $[x_j, x_{j+1})$ contains its original key, which we can do in $O(\rho(s) + 1)$ time by sequential search. By (1–4), this means that in $O(n)$ postprocessing time we have partitioned the set of original keys into disjoint intervals, each containing between εn and $6\varepsilon n$ keys. So, in $O(n \log 1/\varepsilon)$ time, any number can be output in a position at most $6\varepsilon n$ off its rank. Replacing ε by $\varepsilon/6$ completes the proof. □

3 The Data Structure

The data structure is simple but somewhat subtle. This makes it all the more useful to include the actual code of our implementation of soft heaps in C. (It is very short: about 100 lines!) The code should be viewed as a bonus, not a hindrance. We do not base our discussion on it and, in fact, it is possible to skip it in a first reading and still understand soft heaps.

Recall that a binomial tree [5] of rank k is a rooted tree of 2^k nodes: it is formed by the combination of two binomial trees of rank $k-1$, where the root of one becomes the new child of the other root. A soft heap is a sequence of modified binomial trees of distinct ranks, called *soft queues*. The modifications come in two ways:

- A soft queue q is a binomial tree with subtrees possibly missing (somewhat like the trees of a Fibonacci heap [3] after a few deletions). The binomial tree from which q is derived is called its *master tree*. The *rank* of a node of q is the number of children of the corresponding node in the master tree. Obviously, it is an upper bound on the number of children in q. We enforce the following *rank invariant*: the number of children of the root should be no smaller than $\lfloor \text{rank}\,(\text{root})/2 \rfloor$.

- A node v may store several items, in fact, a whole *item-list*. The *ckey* of v denotes the common value of all the current keys of the items in item-list(v): it is an upper bound on the original keys. The soft queue is heap-ordered with respect to ckeys, ie, a ckey of a node does not exceed the ckeys of any of its children. We fix an integer parameter $r = r(\varepsilon)$, and we require that all corrupted items be stored at nodes of rank greater than r.

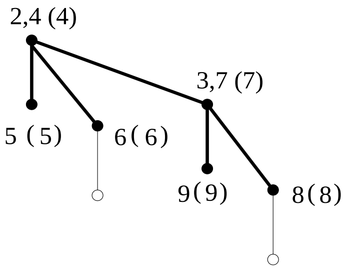

Figure 1: *Two soft queues of rank* 2 *combine to make one soft queue of rank* 3. *Although the edges below 6 and 8 are missing, the rank of both nodes is one (not zero). The soft queue is heap-ordered with respect to ckeys (indicated in parentheses), but not with respect to original keys.*

Turning to the actual C code, an item-list is a singly-linked list of items with one field indicating the original value of the key.

```
typedef struct ILCELL
        { int key;
          struct ILCELL *next;
        } ilcell;
```

A node of a soft queue indicates its ckey and its rank in the master tree. Pointers `next` and `child` give access to the children. If there are none, the pointers are NULL. Otherwise, the node is the parent of a soft queue of rank one less (pointed to by `child`) and the root of a soft queue of rank one less (pointed to by `next`). This is a standard artifice to represent high-degree nodes as sequences of degree-2 nodes. Finally, a pointer `il` gives access to the head of the item-list. To facilitate concatenation of item-lists, we also provide a pointer `il_tail` to the tail of the list.

```
typedef struct NODE
        { int ckey, rank;
          struct NODE *next, *child;,
          struct ILCELL *il, *il_tail;
        } node;
```

The top structure of the heap[1] consists of a doubly-linked list $h_1, \ldots, h_m$, called the *head-list*: each *head* h_i has two extra pointers: one (`queue`) points to the root r_i of a distinct queue, and another (`suffix_min`) points to the root of minimum `ckey` among all r_j's ($j \geq i$). We require that $\text{rank}(r_1) < \cdots < \text{rank}(r_m)$. By extension, the rank of a queue (resp. heap) refers to the rank of its root (resp. r_m). It is stored in the head h_i as the integer variable `rank`.

```
typedef struct HEAD
        { struct NODE *queue;
          struct HEAD *next, *prev, *suffix_min;
          int rank;
        } head;
```

We initialize the soft heap by creating two dummy heads (global variables): `header` gives access to the head-list while `tail`, of infinite rank, represents the end of that list. The functions `new_head` and `new_node` create and initialize a new head and a new node in the trivial manner. The third global variable is the parameter $\mathtt{r} = \mathtt{r}(\varepsilon)$.

```
head *header, *tail; int r;
header = new_head (); tail = new_head ();
tail->rank = INFTY; header->next = tail; tail->prev = header;
printf (``Enter r: ''); scanf (``%d'', &r);
```

4 The Soft Heap Operations

We discuss a minor variant of the data structure, whose code is slightly simpler. It is straightforward to modify the data structure into a full-fledged soft heap. First, our variant bypasses the `create` operation and integrates it within `insert`. Also, note that the operation `delete` can be implemented in the *lazy* style by simply marking the item to be deleted accordingly. Then, actual work is required only when `findmin` returns an item that is marked as deleted. For this reason, we skip the discussion of `findmin` and `delete` altogether, and instead, focus on `deletemin`, the operation which finds an item of minimum key and deletes it. Again, it is immediate to modify the data structure to accommodate `findmin` and `delete` separately.

For each operation we first give an informal description, sometimes using pseudo-code, and then follow up with heavily annotated C code. Except for drivers and I/O code, all that is needed to implement soft heaps is included here.

Inserting an Item

To insert a new item, we create an uncorrupted one-node queue, and we meld it into the heap.

C CODE FOR INSERT

[1] For brevity we drop the "soft."

```
insert (newkey)
int newkey;
{       node *q; ilcell *l;
        l = (ilcell *) malloc (sizeof (ilcell));
        l->key = newkey; l->next = NULL;
        q = new_node (); q->rank = 0; q->ckey = newkey;
        q->il = l; q->il_tail = l;
        meld (q);
}
```

Melding Two Heaps

Consider melding two heaps $\mathcal{S}$ and $\mathcal{S}'$. We begin with a quick overview, and then we discuss the actual implementation of the operation. We break apart the heap of lesser rank, say $\mathcal{S}'$, by melding each of its queues into $\mathcal{S}$. To meld a queue of rank k into $\mathcal{S}$, we look for the smallest index i such that $\text{rank}(r_i) \geq k$. (The dummy head `tail` ensures that i always exists.) If $\text{rank}(r_i) > k$, we insert the head right before h_i, instead. Otherwise, we meld the two queues into one of rank $k+1$, by making the root with the larger key a new child of the other root. If $\text{rank}(r_{i+1}) = k+1$, a new conflict arises. We repeat the process as long as necessary like a carry propagation in binary addition. Finally, we update the `suffix_min` pointers between h_1 and the last head visited. When melding not a single queue but a whole heap, the last step can be done at the very end in just one pass through $\mathcal{S}$. We give the code for melding a soft queue into a soft heap.

C CODE FOR MELD

Let `q` be a pointer (`node *q`) to the soft queue to be melded into the soft heap. First, we scan the head-list until we reach the point at which melding proper can begin. This leads us to the first head of rank at least that of `q`, which is denoted by `tohead`. To facilitate the insertion of the new queue, we also remember the preceding head, called `prevhead`.

```
meld (q)
node *q;
{       head *h, *prevhead, *tohead = header->next;
        node *top, *bottom;
        while (q->rank > tohead->rank) tohead = tohead->next;
        prevhead = tohead->prev;
```

If there is already a queue of the same rank as `q`, we perform the carry propagation, as discussed earlier. When merging two queues, we use the variables `top` and `bottom` to specify which of the two queues end up at/below the root. We create a new node `q` pointing to `top` and `bottom`. Its item-list is inherited from `top`, and its rank is one plus that of top, (ie, `top->rank +1`). Finally, we update `tohead` to point to the next element down the head-list.

```
        while (q->rank == tohead->rank)
           { if (tohead->queue->ckey > q->ckey)
                { top = q; bottom = tohead->queue; }
             else
```

```
          { top = tohead->queue; bottom = q; }
       q = new_node ();
       q->ckey = top->ckey; q->rank = top->rank +1;
       q->child = bottom; q->next = top;
       q->il = top->il; q->il_tail = top->il_tail;
       tohead = tohead->next;
     }   /* end of while loop */
```

We are now ready to insert the new queue in the list of heads. We use a little trick: if a carry has actually taken place, then the head pointed to by `prevhead->next` is now unused and so can be recycled as the head of the new queue. (We omit the garbage collection one might want to carry out to free the newly available space.) Otherwise, we create a new head `h`. We insert `h` between `prevhead` and `tohead`; all the heads inbetween can be discarded. Finally, we call `fix_minlist(h)` to restore the `suffix_min` pointers.

```
        if (prevhead == tohead->prev) h = new_head ();
        else  h = prevhead->next;
        h->queue = q;  h->rank = q->rank;
        h->prev = prevhead; h->next = tohead;
        prevhead->next = h; tohead->prev= h;
        fix_minlist (h);
}
```

C CODE FOR FIX_MINLIST

Prior to calling `fix_minlist(h)`, it is assumed that all `suffix_min` pointers are correct except for those between `header` and `h`. A simple walk from `h` back to `header` updates all the `suffix_mins`.

```
fix_minlist (h)
head *h;
{       head *tmpmin;
        if (h->next == tail) tmpmin = h;
        else tmpmin = h->next->suffix_min;
        while (h != header)
           { if (h->queue->ckey < tmpmin->queue->ckey)
                 tmpmin = h;
             h->suffix_min = tmpmin;
             h = h->prev;
           }
}
```

Deletemin

The `suffix_min` pointer at the beginning of the head-list points to the head `h` with the minimum `ckey` (corrupted or not). The trouble is that the item-list at that node might be empty. In that case, we must refill the item-list with items taken lower down in the queue pointed to by `h`. To do that, we call the function `sift(h->queue, h->rank)`, which replaces the empty item-list by another one. If necessary, we iterate on this process until the new item-list at the root is not empty. The function `sift` is the heart of the soft heap so, without further ado, let us turn our discussion to it.

Taking as argument the node `v` at which the sifting takes place, the function `sift` attempts to move items up the tree towards the root. This is a standard operation in classical heaps. Typically, there is single recursive call in the procedure and the computation tree is a path. The twist is to call the recursion twice once in a while, in order to make the recursion tree branching, ie, truly a tree. This simple modification causes item-lists to collide on the way up, which we resolve by concatenating them. First, some pseudo-code:

`sift(v)`

item-list(`v`) $\leftarrow T \leftarrow \emptyset$;
if `v` has no child
  then set `ckey(v)` to ∞ and return;
1. `sift(v->next)`;
if `ckey(v->next)` $>$ `ckey(v->child)`
  then exchange `v->next` and `v->child`;
$T \leftarrow T \cup$ item-list(`v->next`);
if loop-condition holds **then** goto 1;
item-list(`v`) $\leftarrow T$.

The "loop-condition" statement is what makes soft heaps special. Without it, `sift` would be indistinguishable from the standard deletemin operation of a binomial tree. The loop-condition holds if (i) the goto has not yet been executed during this invocation of `sift` (ie, branching is at most binary), (ii) the rank of v exceeds the threshold `r` and either it is odd or it exceeds the rank of the highest-ranked child of v by at least two. The rank condition ensures that no corruption takes places too low in the queue; the parity condition is there to keep branching from occurring too often; finally the last condition ensures that branching does occur frequently enough. The variable T implements the car-pooling in the concatenation $T \leftarrow T \cup$ item-list(`v->next`). The cleanup is intended to prune the tree of nodes that have lost their item-lists to ancestors and whose `ckeys` have been set to ∞.

C CODE FOR SIFT

The item-list at `v` is worthless and it is effectively emptied at the beginning. We test whether the node `v` is a leaf. If so, we bottom out by setting its `ckey` to infinity (ie, a large integer), which will cause the node to stay at the bottom of the queue. If v is not a leaf then neither `v->next` nor `v->child` is NULL. In fact, this is a general invariant: both are null or neither one is. This might change temporarily within a call to `sift` but it is restored before the call ends.

```
node *sift (v)
node *v;
{       node *tmp;
        v->il = NULL; v->il_tail = NULL;
```

```
if (v->next == NULL && v->child == NULL)
   { v->ckey = INFTY; return v; }
v->next = sift (v->next);
```

The new item-list at `v->next` might now have a large `ckey` which violates the heap ordering. If so, we perform a rotation by exchanging children `v->next` and `v->child`.

```
if (v->next->ckey > v->child->ckey)
   { tmp = v->child;
     v->child = v->next;
     v->next = tmp;
   }
```

Once the children of `v` are in place we update the various pointers at `v`. In particular, the item-list of `v->next` is passed on to `v`, and so is its `ckey`. Recall that while `v->child` is truly a child of `v` in the soft queue, the node `v->next` is a child of `v` only in the binary-tree implementation of the queue.

```
v->il =  v->next->il;
v->il_tail  =  v->next->il_tail;
v->ckey = v->next->ckey;
```

Next in line, the most distinctive feature of soft heaps: the possibility of sifting twice, ie, of creating a branching process in the recursion tree for `sift`. If the loop-condition is satisfied, meaning that the rank of v is odd and large enough, we sift again.

```
if (v->rank > r &&
        (v->rank % 2 == 1 || v->child->rank < v->rank-1))
   { v->next = sift (v->next);
```

As a result of the sifting, another rotation might be needed to restore heap ordering.

```
     if (v->next->ckey > v->child->ckey)
        { tmp = v->child;
          v->child = v->next;
          v->next = tmp;
        }
```

The item-list at `v->next` should now be concatenated with the one at `v`, unless of course, it is empty or no longer defined. The latter case occurs when `ckey` is infinite at both `v->child` and `v->next`. Note that this could not happen after the previous sift.

```
     if (v->next->ckey != INFTY && v->next->il != NULL)
        { v->next->il_tail->next = v->il;
          v->il = v->next->il;
          if (v->il_tail == NULL)
             v->il_tail = v->next->il_tail;
          v->ckey = v->next->ckey;
        }
   }  /* end of second sift */
```

We clean up the queue by removing the nodes with infinite `ckeys`. We do *not* update `v->rank` since rank is defined with respect to the master tree. Note that this is where the rank and the number of children can be made to differ. In fact, we ensure that for any node `v` the ranks of its children (in the binary tree) are always equal, ie, `v->next->rank` = `v->child->rank`.

```
            if (v->child->ckey == INFTY)
               { if (v->next->ckey == INFTY)
                     { v->child = NULL;  v->next = NULL; }
                 else
                     { v->child = v->next->child;
                       v->next = v->next->next; }
               }
            return v;
    }
```

C CODE FOR DELETEMIN

The function `deletemin` returns the item with the smallest `ckey` and deletes it. In practice, safe programming would dictate that we add safety code to warn the user against deleting from an empty heap and things of the sort. We dispense with such niceties here. The first `suffix_min` pointer takes us to the smallest `ckey`, which is what we want unless, of course, the corresponding item-list is empty. In that case, we call `sift`—perhaps more than once—to bring items back to the root. But first, we check whether the rank invariant is violated. Indeed, previous sifting might have caused the loss of too many children of the root and hence a violation of the invariant. We count the children of the root. (Alternatively, we could add a field to keep track of this number.)

```
    deletemin ()
    {   node *sift (), *tmp;
            int min, childcount;  head *h = header->next->suffix_min;
            while (h->queue->il == NULL)
               { tmp = h->queue;  childcount = 0;
                 while (tmp->next != NULL)
                     { tmp = tmp->next; childcount ++; }
```

The advantage in detecting a rank invariant violation so late in the game is that to fix it is much easier since the root's item-list is empty (else, what would we do with it?) If the rank invariant is violated (ie, $\texttt{childcount} < \lfloor \texttt{h} \rightarrow \texttt{rank}/2 \rfloor$), we remove the queue and update the head-list and `suffix_min` pointers. Then, we *dismantle* the root by remelding back its children.

```
                 if (childcount < h->rank/2)
                     { h->prev->next = h->next; h->next->prev = h->prev;
                       fix_minlist (h->prev);
                       tmp = h->queue;
                       while (tmp->next != NULL)
                          { meld (tmp->child);  tmp = tmp->next; }
                     }
```

If the rank invariant holds, we are ready to refill the item-list at the root by calling sift.

```
                    else
                      { h->queue = sift (h->queue);
                        if (h->queue->ckey == INFTY)
                          { h->prev->next = h->next;
                            h->next->prev = h->prev; h = h->prev; }
                        fix_minlist (h);
                      }
                    h = header->next->suffix_min;
                  }   /* end of outer while loop */
```

We are now in a position to delete the minimum-key item.

```
            min = h->queue->il->key;
            h->queue->il = h->queue->il->next;
            if (h->queue->il == NULL)  h->queue->il_tail = NULL;
            return min;
        }
```

5 Complexity Analysis

We prove that the soft heap meets all its claims via a few simple lemmas. We consider a mixed sequence of operations including n inserts. To begin with, we must explain the correspondence between a soft queue and its master tree. When no deletion takes place the equivalence is obvious, and it is trivially preserved through inserts and melds. During sifting, the key observation is that $\mathtt{v} \rightarrow \mathtt{next} \rightarrow \mathtt{rank}$ and $\mathtt{v} \rightarrow \mathtt{child} \rightarrow \mathtt{rank}$ always remain identical. To enforce this equality is what can cause a discrepancy between rank and number of children. But it allows us to think of a rotation as an exchange between soft queues of the same rank (albeit with perhaps missing subtrees). The corresponding master trees having the same rank, they are isomorphic and therefore a rotation has no effect in the correspondence. Similarly, the cleanup prunes away subtrees, with no consequence on the queue/master-tree correspondence.

The interesting aspect of this correspondence is that the leaves of the master tree that are missing from the soft queue correspond to items which have migrated upward to join item-lists of nodes of positive rank. Such items can never again appear in leaves of any soft queue. Note that dismantling a node by remelding its children does not contradict this statement since it merely reconfigures the soft heap.

5.1 The Error Rate

To achieve the desired error rate, we set

$$r \stackrel{\text{def}}{=} 2 + 2\lceil \log 1/\varepsilon \rceil .$$

Lemma 5.1

$$|\text{item-list}\,(v)| \leq \max\Big\{ 1\,,\ 2^{\lceil \text{rank}(v)/2 \rceil - r/2} \Big\}.$$

Proof: Until the first call to `sift`, all item-lists have size one, and the inequality holds. Afterwards, simple inspection shows that all operations have either no effect on the lemma's inequality or sometimes a favorable one (eg, meld). All of them, that is, except for `sift`, which could potentially cause a violation. We show that this is not the case, and prove the lemma's inequality by induction. If `sift(v)` calls itself recursively, via `sift(v→next)`, only once, meaning that the loop-condition is not satisfied, then the item-list of `v → next` (after possible rotation) migrates to a higher-ranking node by itself and the lemma holds by induction. Otherwise, the item-list at `v` becomes the union of the two item-lists associated with `v→next` after each call to `sift(v→next)`. For this to happen, `v→rank` must exceed `r` and one of two conditions must hold: either `v→rank` is odd or it exceeds `v→child→rank`+1. In the first case, after either recursive call, the rank of `v→next` is strictly less than `v→rank`, and by induction the size of either one of the item-lists of `v→next` is at most

$$\max\left\{1\,,\,2^{\lceil(\mathrm{rank}(v)-1)/2\rceil-r/2}\right\}=2^{\lceil(\mathrm{rank}(v)-1)/2\rceil-r/2}=2^{\lceil\mathrm{rank}(v)/2\rceil-1-r/2}.$$

Note that the max disappears because $\mathrm{rank}(v)>r$. In the other case, the size of either one of the item-lists of `v→next` is at most

$$\max\left\{1\,,\,2^{\lceil(\mathrm{rank}(v)-2)/2\rceil-r/2}\right\}=2^{\lceil\mathrm{rank}(v)/2\rceil-1-r/2}.$$

This time, the max disappears because r and the rank of v are both even, and so $\mathrm{rank}(v)\geq r+2$. In sum, the size of the union is at most $2\times 2^{\lceil\mathrm{rank}(v)/2\rceil-1-r/2}$, which proves the lemma. □

Lemma 5.2 *The soft heap contains at most $n/2^{r-3}$ corrupted items at any given time.*

Proof: We begin with a simple observation. If S is the node set of a binomial tree then

$$\sum_{v\in S}2^{\,\mathrm{rank}(v)/2}\leq 4|S|. \tag{1}$$

This follows from the inequality

$$\sum_{v\in S}2^{\,\mathrm{rank}(v)/2}\leq 2^{k+2}-3\cdot 2^{k/2},$$

where k is the rank of the binomial tree. A proof by induction is immediate and can be omitted. Recall that the ranks of the nodes of a queue q are derived from the corresponding nodes in its master tree q'. So, the set R (resp. R') of nodes of rank greater than r in q (resp. q') is such that $|R|\leq|R'|$. Within q', the nodes of R' number a fraction at most $1/2^r$ of all the leaves. Summing over all master trees, we find that

$$\sum_{q'}|R'|\leq n/2^r. \tag{2}$$

There is no corrupted item at any rank $\leq r$, and so by Lemma 5.1 their total number does not exceed

$$\sum_{q'}\sum_{v\in R'}2^{\,(\mathrm{rank}(v)+1-r)/2}=2\sum_{q'}\sum_{v\in R'}2^{\,(\mathrm{rank}(v)-r-1)/2}. \tag{3}$$

Each R' forms a binomial tree by itself, where the rank of node v becomes $\mathrm{rank}(v)-r-1$. So, by (1, 2), the sum in (3) is at most $\sum_{q'}8|R'|\leq n/2^{r-3}$. □

5.2 The Running Time

Only `meld` and `sift` need to be looked at, all other operations being trivially constant-time. Assigning one credit per queue takes care of the carry propagation during a `meld`. Indeed, two queues of the same rank combine into one, which releases one credit to pay for the work. Updating suffix-min pointers can take time, however. Specifically, carries aside, the cost of melding two soft heaps $\mathcal{S}$ and $\mathcal{S}'$ is at most the smaller rank of the two (up to a constant factor). The entire sequence of soft heap melds can be modeled as a binary tree $\mathcal{M}$. A leaf z denotes a one-item heap (its cost is 1). An internal node z indicates the melding of two heaps. Since heaps can grow only through melds, the added size of the master trees in the soft heap at z is proportional to the number $N(z)$ of descendents in $\mathcal{M}$. The cost of node z (ie, of the meld) is $1 + \log \min\{ N(x), N(y) \}$, where x and y are the left and right children of z.[2] A simple recurrence (see eg, [4]) shows that adding together all these costs gives a total melding cost linear in the size of $\mathcal{M}$, ie, $O(n)$.

For this analysis to be correct, no node dismantling should ever take place. We can easily amend it, however, to cover the general case. For the purpose of the analysis, let us not regard the remeldings caused by dismantling as heap melds but as queue melds. The benefit is to leave the tree $\mathcal{M}$ unchanged. The dismantle-induced melds associated with a node z of $\mathcal{M}$ reconfigure the soft heap at z by removing some of its nodes and restoring the rank invariant. This can only decrease the value of $N(z)$, so the previous analysis remains correct.

Of course, the queue melds associated with node x must now be accounted for. Dismantling node v causes no more than $\text{rank}(v)$ queue melds. By the violation of the rank invariant, the node v has at least one missing child of rank $\geq \lceil \text{rank}(v)/2 \rceil$. In the master tree there are at least $2^{\lceil \text{rank}(v)/2 \rceil - 1}$ leaves at or below that child, and all have disappeared from the soft queue. So, we can charge the dismantle-induced melds against these leaves, and conclude that melding takes $O(n)$ time.

Finally, we show that the cost of all calls to `sift` is $O(rn)$. Consider any decreasing sequence of integers. An integer m is called good if it is odd or if its successor is less than $m-1$. Clearly, any subsequence of size two contains at least one good integer. Now, consider the computation tree corresponding to an execution of `sift(v)`. By examining the sequence of ranks along any root-to-leaf path, our previous observation leads us to conclude that along any path of size at least r, at least one branching must occur (not necessarily many more than that because no branching occurs at rank r and below). It follows that, excluding the updating of suffix-min pointers, the running time is $O(rC)$, where C is the number of times the loop-condition succeeds.

It is easy to see, by induction, that if v is the root of a subtree with fewer than two finite `ckeys` in the subtree below, the computation tree of `sift(v)` is of constant size. Conversely, if the subtree contains at least two finite `ckeys` at distinct nodes, then if the loop-condition is satisfied at v, both calls of the form `sift(v→next)` bring finite `ckeys` to the root and two nonempty item-lists are thus concatenated. There can be at most $n-1$ such merges, therefore $C \leq n$ and our claim holds.

We ignored the cost of updating `suffix_min` pointers after each call to `sift`. Maintaining the rank invariant makes the cost of `suffix_min` updating negligible. Indeed, each update takes time proportional to the rank of the queue: (i) if the rank invariant holds, then the updating time is dominated by the cost of `sift` itself (already accounted for); (ii) otherwise,

[2] We use the fact that the rank of a soft queue is exactly the logarithm of the number of nodes in its master tree.

the root v is dismantled, which, as we just saw, releases $2^{\lceil \text{rank}(v)/2 \rceil - 1}$ leaves against which we can charge the updating cost. By Lemma 5.2, the total number of corrupted items is bounded by $n/2^{r-3}$. Setting $r = 2 + 2\lceil \log(1/\varepsilon) \rceil$ proves Theorem 1.1 (except for the optimality claim). $\square$

Remark: The storage is linear in the number of insertions n, but not necessarily in the actual number of items present. If storage is at a premium, here is a simple fix: as soon as the number of live items falls below, say, $n/2$, we reconfigure the soft heap entirely. We reinsert all the uncorrupted items and, separately, we string together the corrupted ones into a single item-list whose common key is set to infinity. The time for reconfiguring the heap can be charged to the $n/2$ deleted elements. Regarding corruption, the reconfiguration merely doubles the number of insertions, and so the number of corrupted items after I (user-requested) insertions will be at most $2\varepsilon I$. So, it suffices to replace ε by $\varepsilon/2$ to achieve an error rate of ε. This modified soft heap is optimal in storage, and as shown below, in time.

6 Optimality

To complete the proof of Theorem 1.1, we show that the soft heap is optimal. Without loss of generality, we can assume that $1/\varepsilon$ lies between a large constant and $\sqrt{n}$ and that $n/\lfloor \varepsilon n \rfloor$ is an integer l. Apply Theorem 2.1 to n distinct numbers and pick out every $2\lfloor \varepsilon n \rfloor$-th element in the output sequence. By the near-sortedness of the output, the chosen subsequence is already sorted. It partitions the set of numbers into disjoint intervals, within which we can easily locate the other numbers in linear time (each element being at most a constant number of intervals off its enclosing interval). From this, in particular, we derive the true rank of the selected numbers. By using linear selection-finding within each interval, we easily, in $O(n)$ time, retrieve the $k\lfloor \varepsilon n \rfloor$-th largest number, for $k = 1, 2, \ldots, l$. We have partitioned the set of n numbers into disjoint intervals of size $\lfloor \varepsilon n \rfloor$. Let $n_1 = \cdots = n_l = \lfloor \varepsilon n \rfloor$. A standard counting argument shows that any comparison-based tree for performing this computation is of height at least

$$\log \binom{n}{n_1, \ldots, n_l} = \Omega(n \log 1/\varepsilon).$$

So, the entire algorithm requires $\Omega(n \log 1/\varepsilon)$ time, but it involves $O(n)$ operations on a soft heap with error rate ε, followed by $O(n)$-time postprocessing. It follows that the $O(\log 1/\varepsilon)$ amortized complexity of the soft heap is optimal. $\square$

Acknowledgments

I wish to thank the anonymous referee for some useful suggestions.

References

[1] Blum, M., Floyd, R.W., Pratt, V., Rivest, R.L., Tarjan, R.E. *Time bounds for selection*, Journal of Computer and System Sciences, 7 (1973), 448–461.

[2] Chazelle, B. *A minimum spanning tree algorithm with inverse-Ackermann type complexity*, to appear. Prelim. version in *A faster deterministic algorithm for minimum spanning trees*, Proc. 38th Ann. IEEE Symp. Found. Comp. Sci. (1997), 22–31.

[3] Fredman, M.L., Tarjan, R.E. *Fibonacci heaps and their uses in improved network optimization algorithms*, J. ACM 34 (1987), 596–615.

[4] Hoffman, K., Mehlhorn, K., Rosenstiehl, P., Tarjan, R.E. *Sorting Jordan sequences in linear time using level-linked search trees*, Inform. and Control 68 (1986), 170–184.

[5] Vuillemin, J. *A data structure for manipulating priority queues*, Commun. ACM 21 (1978), 309–315.