A simpler implementation and analysis of Chazelle's Soft Heaps

Haim Kaplan [*] *Uri Zwick* [*]

Abstract

Chazelle (JACM 47(6), 2000) devised an approximate meldable priority queue data structure, called *Soft Heaps*, and used it to obtain the fastest known deterministic comparison-based algorithm for computing minimum spanning trees, as well as some new algorithms for selection and approximate sorting problems. If n elements are inserted into a collection of soft heaps, then up to εn of the elements still contained in these heaps, for a given *error parameter* ε, may be *corrupted*, i.e., have their keys artificially increased. In exchange for allowing these corruptions, each soft heap operation is performed in $O(\log \frac{1}{\varepsilon})$ amortized time.

Chazelle's soft heaps are derived from the *binomial heaps* data structure in which each priority queue is composed of a collection of *binomial trees*. We describe a simpler and more direct implementation of soft heaps in which each priority queue is composed of a collection of standard *binary* trees. Our implementation has the advantage that no *clean-up* operations similar to the ones used in Chazelle's implementation are required. We also present a concise and unified potential-based amortized analysis of the new implementation.

1 Introduction

Chazelle [4, 2] devised an approximate meldable priority queue data structure, called *Soft Heaps*, and used it to obtain the fastest known deterministic comparison-based algorithm for computing minimum spanning trees (Chazelle [3, 2]), as well as some new algorithms for selection and approximate sorting problems. If n elements are inserted into a collection of soft heaps, then up to εn of the elements still contained in these heaps, for a given *error parameter* ε, may be *corrupted*, i.e., have their keys artificially increased. (Note that n here is the number of elements inserted into the heaps, not the current number of elements in the heaps which may be considerably smaller.) In exchange for allowing these corruptions, each soft heap operation is performed in $O(\log \frac{1}{\varepsilon})$ amortized time.

Soft heaps are also used by Pettie and Ramachandran [8, 9] to obtain an optimal deterministic comparison-based algorithm for finding minimum spanning trees, with a yet unknown running time, and for obtaining a randomized linear time algorithm for the problem that uses only a small number of random bits.

Chazelle's soft heaps are derived from the *binomial heaps* data structure in which each priority queue is composed of a collection of *binomial trees*. We describe a simpler and more direct implementation of soft heaps in which each priority queue is composed of a collection of standard *binary* trees. Our implementation has the advantage that no *clean-up* operations similar to the ones used in Chazelle's implementation are required. We also present a concise and unified potential-based amortized analysis of the new implementation.

1.1 Soft heaps

Soft heaps are approximate meldable priority queue data structures that support the following operations:

`make-heap`(e) – Generate and return a new soft heap containing the single element e whose original key is *key*[e].

`insert`(P, e) – Insert element e, with original key *key*[e], into soft heap P.

`delete`(e) – Delete element e from the soft heap currently containing it. (It is assumed that e is currently contained in exactly one soft heap.)

`meld`(P, Q) – Meld the two soft heaps P and Q, destroying them in the process, and return the melded heap.

`extract-min`(P) – Return an element with the smallest *current* key in soft heap P and delete it from P.

It is important to note that an `extract-min`(P) operation returns an element e with the smallest *current* key contained in P. The current key of an element e may be *larger* then its *original* key *key*[e], which is never changed by the implementation. Elements whose current key is larger than their original key are said to be *corrupted*. Current keys of elements are sometimes

[*]School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel. E–mail: {haimk,zwick}@tau.ac.il. Uri's research was supported by BSF grant no. 2006261. Haim Kaplan is partially supported by BSF grant no. 2006204

raised to speed-up the implementation of soft heap operations. The user has no control as to which elements become corrupted.

If soft heaps were allowed to corrupt all elements then their implementation would be trivial, but they would be useless. A surprisingly useful data structure is obtained if we require that at most εn of the elements still contained in soft heaps are corrupted, where n is the total number of elements inserted so far into soft heaps, and $0 < \varepsilon < 1$ is a prespecified *error parameter*. Note that an element is inserted into a soft heap by either a `make-heap` or an `insert` operation. Also note that corrupted elements that were removed from soft heaps by `extract-min` operations are not counted.

Following Chazelle [4], we describe an implementation of soft heaps with error parameter ε in which the amortized cost of `make-heap` and `insert` operations is $O(\log \frac{1}{\varepsilon})$ and the amortized cost of all other operations is 0.

2 Implementation

In this section we describe the implementation of all soft heap operations, except the `delete` operation which will be added in Section 5. (We note that some of the applications listed by Chazelle [4] do not use `delete` operations.)

2.1 The data structure Each soft heap priority queue is composed of a collection of binary trees. A node x of a binary tree may have a left child $\mathit{left}[x]$ and may have a right child $\mathit{right}[x]$. If x does not have a left child, then $\mathit{left}[x] = \perp$, where $\perp$ represents *null*. Similarly, if x does not have a right child, then $\mathit{right}[x] = \perp$. Every node x in a binary tree has an integer *rank*, denoted by $\mathit{rank}[x]$, associated with it. The rank of a node never changes. If x is a node of rank k, then the ranks of $\mathit{left}[x]$ and $\mathit{right}[x]$, if they exist, are $k-1$. The rank of a tree is defined as the rank of its root.

Each node x, has a *target size* $\mathit{size}[x]$ associated with it. Let $r = \lceil \log_2 \frac{1}{\varepsilon} \rceil + 5$, where ε is the desired error rate. The target size $\mathit{size}[x]$ of a node of rank k is s_k, where

$$s_k = \begin{cases} 1 & \text{if } k \le r, \\ \lceil \frac{3 s_{k-1}}{2} \rceil & \text{otherwise.} \end{cases}$$

(The choice of 3/2 in the definition of s_k is arbitrary. Any constant strictly between 1 and 2 would do.) Thus $s_0 = s_1 = \ldots = s_r = 1$, while $s_{r+1} = 2$, $s_{r+2} = 3$, $s_{r+3} = 5$, $s_{r+4} = 8$, $s_{r+5} = 12$, etc. It is easy to prove that

$$\left(\tfrac{3}{2}\right)^{k-r} \le s_k \le 2\left(\tfrac{3}{2}\right)^{k-r} - 1 \quad \text{for} \quad k \ge r\,.$$

Each node x has a list of elements $\mathit{list}[x]$. The number of elements in $\mathit{list}[x]$ is 'roughly' $\mathit{size}[x]$. (This will be made more precise below.) A node x also has a key $\mathit{ckey}[x]$ which is an upper bound on the keys of the elements contained in $\mathit{list}[x]$. If e is an element contained in $\mathit{list}[x]$, and $\mathit{key}[e] < \mathit{ckey}[x]$, then e is *corrupted.* The data structure behaves as if the key of e is artificially raised to $\mathit{ckey}[x]$. (In the terminology of the previous section, if e is an element of $\mathit{list}[x]$, then $\mathit{ckey}[x]$ is the current key of e.)

Each tree is *heap ordered* with respect to the *ckey* values, i.e., if x is a node and $\mathit{left}[x]$ exists, then $\mathit{ckey}[x] \le \mathit{ckey}[\mathit{left}[x]]$. Similarly, if $\mathit{right}[x]$ exists, then $\mathit{ckey}[x] \le \mathit{ckey}[\mathit{right}[x]]$.

A priority queue P is composed of a sequence of trees, at most one of each rank. The rank $\mathit{rank}[P]$ of P is defined to be the largest rank of a tree in P. The trees composing P are arranged in a linked list in which the trees appear in an increasing order of rank. $\mathit{first}[P]$ points to the tree with the smallest rank belonging to P.

If T is a tree contained in a priority queue P, then $\mathit{root}[T]$ is the root node of T, $\mathit{next}[T]$ is the tree following T in the linked list of P, and $\mathit{prev}[T]$ is the tree preceding T in the list. (Both $\mathit{next}[T]$ and $\mathit{prev}[T]$ may be $\perp$.) Finally, if T is a tree, then $\mathit{sufmin}[T]$ points to the tree whose root has the smallest *ckey* among all the trees that follow T in the linked list of P. (In case of ties, trees that appear earlier in the list are preferred.)

A soft heap containing four binary trees of ranks 0,1,4 and 5, respectively is shown in Figure 1. The numbers within the nodes of the trees are their *ckey* values. Each node has a list $\mathit{list}[x]$ of elements associated with it. (Lists of length 1 are not shown.)

2.2 The `sift` operation As in Chazelle's implementation, the keystone of soft heaps operation is the `sift` operation. As we have mentioned, we would like the number of elements in $\mathit{list}[x]$ to be about $\mathit{size}[x]$. If the number of elements in $\mathit{list}[x]$ drops below $\mathit{size}[x]/2$, and x is not a leaf, we use a `sift`(x) operation to add more elements to $\mathit{list}[x]$.

A `sift`(x) operation works as follows. By exchanging $\mathit{left}[x]$ and $\mathit{right}[x]$, if necessary, we make sure that $\mathit{left}[x]$ exists, and that $\mathit{ckey}[\mathit{left}[x]] \le \mathit{ckey}[\mathit{right}[x]]$, if $\mathit{right}[x]$ exists. (Recall that x is not a leaf.) Then, using what Chazelle refers to as the "data structures' version of car-pooling", we take all the elements of $\mathit{list}[\mathit{left}[x]]$ and move them to $\mathit{list}[x]$. (This can be done in constant time by concatenating the lists $\mathit{list}[x]$ and $\mathit{list}[\mathit{left}[x]]$.) We let $\mathit{ckey}[\mathit{left}[x]]$ be the new $\mathit{ckey}[x]$. The list $\mathit{list}[\mathit{left}[x]]$ is now empty. If $\mathit{left}[x]$ is a leaf, we simply remove it from the tree by setting $\mathit{left}[x]$ to $\perp$. Otherwise, we recursively call `sift`$(\mathit{left}[x])$ to replenish

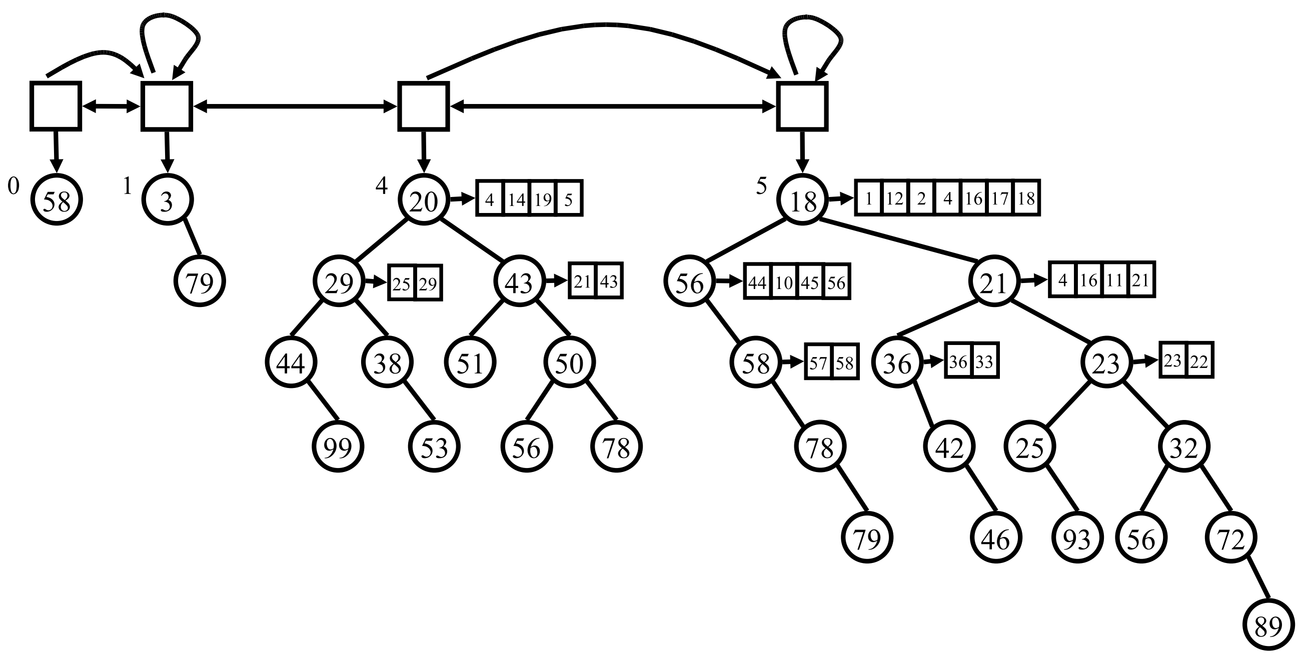

Figure 1: A soft heap composed of four binary trees.

list[*left*[x]]. Finally, if the number of elements in *list*[x] is still below *size*[x] and x still not a leaf, we perform these operations again. (See Figure 2.)

It is easy to check that `sift` operations maintain heap order. We show below that if x is not a leaf after a `sift`(x) operation, then $size[x] \leq |list[x]| \leq 3\,size[x]$.

2.3 The `combine` operation The second most important operation in the implementation of soft heaps is the `combine` operation. A `combine`(x, y) operation takes two root nodes x and y of the same rank, say k, and combines the corresponding trees into a single tree of rank $k + 1$. This is done by generating a new node z and setting $left[z] \leftarrow x$, $right[z] \leftarrow y$ and $list[z] \leftarrow \phi$. A `sift`(z) operation is then performed to move enough elements into *list*[z]. (See Figure 2.) The `combine` operation is of course instrumental in the implementation of `meld` and `insert` operations, as we explain below.

Note that we do not do any rebalancing of the binary trees of the heaps. The only structural changes performed on the trees are: 1) Discarding a leaf (done by `sift`). 2) Combining two trees of rank k into a larger tree of rank $k + 1$ by allocating a new root (done by `combine`).

2.4 The `update-suffix-min` operation An `update-suffix-min`(T) operation updates the *sufmin* pointers of T and all the trees that precede T in the linked list of trees. Such an operation is performed when *ckey*[x], where $x = root[T]$, is changed, e.g., by a `sift`(x) operation, when T is a new tree added to the list of trees, or when the tree following T in the list is deleted.

An `update-suffix-min`(T) operation traverses the list of trees backward from T. If T' is a tree such that *sufmin*[*next*[T']] was already set to its correct value, then *sufmin*[T'] is set to T', if $ckey[root[T']] \leq ckey[root[next[T']]]$, or to *sufmin*[*next*[$T'$]], otherwise. (See Figure 4 below.)

2.5 The `make-heap` operation A `make-heap`(e) operation receives an element e and returns a priority queue P composed of a single tree T containing a single node x of rank 0. (See Figure 3.)

2.6 The `meld` operation A `meld`(P, Q) operation receives two priority queues P and Q and returns a new priority queue obtained by melding P and Q. Melding P and Q is done in a fairly straightforward way. The linked lists of trees of P and Q are combined, keeping a non-decreasing order of rank. Next, if two consecutive trees T_1 and T_2 in the list have the same rank, they are combined using a `combine`($root[T_1], root[T_2]$) operation and the combined tree replaces them in the list. If three consecutive trees T_1, T_2 and T_3 in the list have the same rank, then T_1 is left alone, while T_2 and T_3 are replaced by the combined tree `combine`($root[T_2], root[T_3]$). Fi-

Function `sift`(x)

```
while $|list[x]| < size[x]$ and (not leaf($x$)) do
    if $left[x] = \perp$ or ($right[x] \neq \perp$ and $ckey[left[x]] > ckey[right[x]]$) then
        $left[x] \leftrightarrow right[x]$
    concatenate($list[x], list[left[x]]$)
    $ckey[x] \leftarrow ckey[left[x]]$
    $list[left[x]] \leftarrow \perp$
    if leaf($left[x]$) then
        $left[x] \leftarrow \perp$
    else
        sift($left[x]$)
```

Function `combine`(x, y)

```
$z \leftarrow$ new-node()
$left[z] \leftarrow x$
$right[z] \leftarrow y$
$rank[z] \leftarrow rank[x] + 1$
if $rank[z] \leq r$ then
    $size[z] \leftarrow 1$
else
    $size[z] \leftarrow (3 * size[x] + 1)/2$
sift($z$)
return $z$
```

Figure 2: Implementation of `sift` and `combine`.

nally, if T is the last tree in the list affected by these operations, an `update-suffix-min`(T) is performed to update the *sufmin* pointers. If $k = rank[P] \leq rank[Q]$, then it is easy to perform `meld`(P, Q) operation in $O(k + 1)$ time. (See Figure 3.)

2.7 The `insert` operation To add an element e to priority queue P, we use `make-heap`(e) to generate a priority queue containing the single element e and we then meld this priority queue with P. (See Figure 3.)

2.8 The `extract-min` operation An `extract-min`(P) operation returns an element with a minimum *current* key contained in P. If e is contained in $list[x]$, then the current key of e is $ckey[x]$. The implementation of `extract-min`(P) is extremely simple. Let $T = sufmin[first[P]]$ and let $x = root[T]$. Thus x is the root of a tree of P with the smallest *ckey*. We return an arbitrary element from $list[x]$. Note that $list[x]$ is never empty. If the number of elements in $list[x]$ drops below $size[x]/2$, we call `sift`(x) to replenish x and then `update-suffix-min`(T) to update the *sufmin* pointers. (See Figure 3.)

2.9 Pseudo-code As already mentioned, pseudo-code for `sift` and `combine` is given in Figure 2, and pseudo-code for `make-heap`, `meld`, `insert` and `extract-min` is given in Figure 3. (The reader is advised to have at least a quick look at Figures 2 and 3, as they give a concise and precise definition of the main soft heap operations.)

The fairly standard implementation of operations performed on sequences of trees are given in Figure 4. Among the functions described there are `update-suffix-min`, and the functions `merge-into` and `repeated-combine` called by `meld`. Finally, Figure 5 describes the functions `make-tree` and `make-node`, that generate a new tree and a new node, respectively, and the function `leaf` that checks whether a given node is a leaf.

In `extract-min`, we use a function called `pick-elem` to pick, and delete, an arbitrary element from $list[x]$, which is assumed to be a linked list. In `sift`, we use a function called `concatenate` to concatenate the lists $list[x]$ and $list[left[x]]$. The straightforward implementation of these two functions is not given.

3 Correctness

We start by showing that `extract-min` operations do indeed return elements with minimum current keys:

LEMMA 3.1. *A* `extract-min`(P) *always returns an element of P with a minimal* current *key.*

Proof. All elements inserted into P are contained in the lists of nodes that are part of the trees forming P. All operations performed on soft heaps maintain heap order. Thus, elements with the smallest current key in a tree always reside at the root of that tree. A `extract-min`(P) operation uses the *sufmin* pointer of the first tree in P to access a tree whose root x has a minimal *ckey*, and returns an element e contained in $list[x]$, which is guaranteed to be non-empty. This element has a minimal key, as required. □

The next two lemmas will be used to bound the number of corrupted elements.

```
Function makeheap(e)

    $P \leftarrow$ new-heap()
    $first[P] \leftarrow$ make-tree($e$)
    $rank[P] \leftarrow 0$
    return $P$
```

```
Function meld(P, Q)

    if $rank[P] > rank[Q]$ then $P \leftrightarrow Q$
    merge-into($P, Q$)
    repeated-combine($Q, rank[P]$)
    return $Q$
```

```
Function insert(P, e)

    return meld($P$, make-heap($e$))
```

```
Function extract-min(P)

    if $first[P] = \perp$ then return $\perp$
    $T \leftarrow sufmin[first[P]]$
    $x \leftarrow root[T]$
    $e \leftarrow$ pick-elem($list[x]$)
    if $|list[x]| \leq size[x]/2$ then
        if not leaf($x$) then
            sift($x$)
            update-suffix-min($T$)
        else if $list[x] = \phi$ then
            remove-tree($P, T$)
    return $e$
```

Figure 3: Implementation of main soft heaps operations.

LEMMA 3.2. *If x is a node of rank at most r, then $|list[x]| = 1$. If x is a non-leaf node of rank $k \geq r$, then $\frac{1}{2}\, size[x] \leq |list[x]| < 3\, size[x]$.*

Proof. If x is a node of rank at most r, then $size[x] = 1$. If $list[x]$ becomes empty, then sift(x) brings exactly one element into $list[x]$.

Suppose now that $rank[x] \geq r$. If $|list[x]|$ drops below $\frac{1}{2}\, size[x]$, and x is not a leaf, then sift(x) adds elements to $list[x]$ until either $|list[x]| \geq size[x]$, or until x becomes a leaf. We next prove by induction that $|list[x]| \leq 3size[x]$. If $rank[x] \leq r$, the claim is obvious. We show now that if the claim holds for all vertices of rank $k-1$, then it also holds for all vertices of rank k. Let x be a node of rank k. New elements are added to $list[x]$ only when $|list[x]| < size[x]$. A sift(x) operation concatenates $list[y]$ to $list[x]$, where y is a child of x. As y is of rank $k-1$, we get by induction that $|list[y]| < 3\, size[y] \leq 3 \cdot \frac{2}{3} size[x] = 2\, size[x]$. Thus, $|list[x] \cup list[y]| < 3size[x]$, as claimed. □

LEMMA 3.3. *If n elements are inserted into soft heaps, then the number of nodes of rank k is at most $n/2^k$.*

Proof. By induction on k. A node of rank 0 is generated only when a new element is inserted (using a make-heap operation) into a soft heap. Thus, the number of elements of rank 0 is at most n as claimed. An element of rank k is generated only when two roots of rank $k-1$ are combined. □

LEMMA 3.4. *If n elements are inserted into soft heaps, then the total number of corrupted elements contained in the heaps, at any given time, is at most εn.*

Proof. Each node of rank at most r contains a single element. Thus, all corrupted elements belong to nodes of rank greater than r. By Lemma 3.3, the number of nodes of rank k is at most $n/2^k$. By Lemma 3.2, a node of rank $k > r$ contains at most $3s_k < 6\left(\frac{3}{2}\right)^{k-r}$ elements. As $r = \lceil \log_2 \frac{1}{\varepsilon} \rceil + 5$, the number of corrupted elements is at most

$$\sum_{k>r} \frac{n}{2^k} \cdot 3s_k \;<\; \frac{n}{2^r} \cdot \sum_{k>r} 6\left(\tfrac{1}{2}\right)^{k-r} \left(\tfrac{3}{2}\right)^{k-r}$$

$$= \; \frac{6n}{2^r} \cdot \sum_{i \geq 1} \left(\tfrac{3}{4}\right)^i \;=\; \frac{18n}{2^r} \;<\; \varepsilon n\,.$$

□

4 Amortized analysis

We assign potentials to heaps, trees, and nodes. A heap of rank k has potential $k+1$. A tree whose root is x has potential $(r+2) \cdot del(x)$, where $del(x)$ is the number of elements deleted from x since the last sift(x) operation, or since the creation of x. If x is a root node of rank k, then x has potential $k+7$. If x is a non-root node, it has potential 1.

We start with the analysis of sift. Suppose that x is a node of rank k, that y is a child of x, and that the elements of $list[y]$ are moved to $list[x]$. If $|list[y]| <$

```
Function merge-into(P, Q)

if $rank[P] > rank[Q]$ then abort
$T_1 \leftarrow first[P]$
$T_2 \leftarrow first[Q]$
while $T_1 \neq \perp$ do
    while $rank[T_1] > rank[T_2]$ do
        $T_2 \leftarrow next[T_2]$
    $T_1' \leftarrow next[T_1]$
    insert-tree$(Q, T_1, T_2)$
    $T_1 \leftarrow T_1'$
```

```
Function repeated-combine(Q, k)

$T \leftarrow first[Q]$
while $next[T] \neq \perp$ do
    if $rank[T] = rank[next[T]]$ then
        if $next[next[T]] = \perp$ or
        $rank[T] \neq rank[next[next[T]]]$
        then
            $root[T] \leftarrow$
            combine$(root[T], root[next[T]])$
            $rank[T] \leftarrow rank[root[T]]$
            remove-tree$(Q, next[T])$
    else if $rank[T] > k$ then
        break
    $T \leftarrow next[T]$
if $rank[T] > rank[Q]$ then
    $rank[Q] \leftarrow rank[T]$
update-suffix-min$(T)$
```

```
Function update-suffix-min(T)

while $T \neq \perp$ do
    if $ckey[root[T]] \leq$
    $ckey[root[sufmin[next[T]]]]$ then
        $sufmin[T] \leftarrow T$
    else
        $sufmin[T] \leftarrow sufmin[next[T]]$
    $T \leftarrow prev[T]$
```

```
Function insert-tree(P, T_1, T_2)

$next[T_1] \leftarrow T_2$
if $prev[T_2] = \perp$ then
    $first[P] \leftarrow T_1$
else
    $next[prev[T_2]] \leftarrow T_1$
```

```
Function remove-tree(P, T)

if $prev[T] = \perp$ then
    $first[P] = next[T]$
else
    $next[prev[T]] \leftarrow next[T]$
if $next[T] \neq \perp$ then
    $prev[next[T]] \leftarrow prev[T]$
```

Figure 4: Implementation of `update-suffix-min` and other operations on sequences of trees

$\frac{1}{2}\,size[y]$, then y is a leaf, and y disappears as a result of this operation. The unit potential released by the disappearance of y pays for this operation. If, on the other hand, $|list[y]| \geq \frac{1}{2}\,size[y]$, and hence $|list[y]| \geq \lceil\frac{size[y]}{2}\rceil$, we split the unit cost of the operation among the $|list[y]|$ elements participating in the 'car-pool'. The charge for each element is at most $\lceil\frac{size[y]}{2}\rceil^{-1} = \lceil\frac{s_{k-1}}{2}\rceil^{-1}$. An element is charged at most once at each rank, thus its total 'travel expenses' are

$$\sum_{k\geq 0}\left\lceil\frac{s_k}{2}\right\rceil^{-1} \;\leq\; r + 2\sum_{i\geq 0}\left(\tfrac{2}{3}\right)^i \;=\; r+6 \;=\; O(\log\frac{1}{\varepsilon})\,.$$

We next consider a `combine`(x, y) operation which combines two trees of rank k, rooted at x and y, into a tree of rank $k+1$, rooted at a new node z. The potentials of both x and y decrease from $k+7$ to 1, releasing $2k+12$ units of potential. One of these units pays for the constant cost of the operation, $k+8$ units are assigned to z, one unit is used to increase the potential of the heap containing the newly formed tree, if it is now the tree with the largest rank. The remaining $k+2$ units are used, if needed, to pay for an ensuing `update-suffix-min` operation. (Note that $1+(k+8)+1+(k+2) = 2k+12$.)

A `meld`(P, Q) operation receives two heaps, of ranks k and k' respectively. Suppose that $k \leq k'$. The meld operation first merges the lists of trees of P and Q, effectively destroying P. This takes $O(k+1)$ time, which is paid by the released potential of P. The subsequent `combine` operations pay for themselves and for the en-

```
Function maketree(e)
    $T \leftarrow$ new-tree()
    $root[T] \leftarrow$ make-node($e$)
    $next[T] \leftarrow \perp$
    $prev[T] \leftarrow \perp$
    $rank[T] \leftarrow 0$
    $sufmin[T] \leftarrow T$
    return $T$
```

```
Function makenode(e)
    $x \leftarrow$ new-node()
    $list[x] \leftarrow \{e\}$
    $ckey[x] \leftarrow key[e]$
    $rank[x] \leftarrow 0$
    $size[x] \leftarrow 1$
    $left[x] \leftarrow \perp$
    $right[x] \leftarrow \perp$
    return $x$
```

```
Function leaf(x)
    return ($left[x] = \perp$) and ($right[x] = \perp$)
```

Figure 5: Implementation of `make-tree`, `make-node` and `leaf`.

suing `update-suffix-min` operation. If no `combine` operations are performed, the potential released by the destruction of P pays for the `update-suffix-min` operation.

Finally, we bound the amortized cost of `extract-min`(P) operations. An `extract-min`(P) operation locates an element e with minimal current key in constant time. Suppose that x is the root node containing e. If after deleting e from $list[x]$ we still have $|list[x]| \geq \frac{1}{2}\, size[x]$, or if $|list[x]| > 0$ and x is a leaf, then no further action is taken. Note, however, that $del(x)$, the number of elements deleted from $list[x]$ since the last `sift`(x) operation, is increased by 1, and the potential of tree whose root is x is increased by $r+2$. This total cost of $1 + (r+2) = r+3$ is charged to e, which would never be charged again. If $|list[x]| < \frac{1}{2}\, size[x]$ and x is not a leaf, then a `sift`(x) operation is performed. As x is not a leaf, it must have had at least $size[x]$ elements in its list after the previous `sift`(x) operation. Thus $del(x) \geq \lceil \frac{size[x]}{2} \rceil$, and the potential of the tree, prior to the `sift` operation, is at least $(r+2)\lceil \frac{size[x]}{2} \rceil = (r+2)\lceil \frac{s_k}{2} \rceil$, where $k = rank[x]$. It is not difficult to verify that for every $k \geq 0$ we have

$$(r+2)\left\lceil \frac{s_k}{2} \right\rceil \;\geq\; k+1\ .$$

Indeed, for $0 \leq k \leq r+1$, we have $(r+2)\lceil \frac{s_k}{2} \rceil = r+2 \geq k+1$. For $k = r+2$, we have $s_{r+2} = 3$ and thus $(r+2)\lceil \frac{s_{r+2}}{2} \rceil = 2(r+2) > r+3$. For $k \geq r+3$, we have $(r+2)\lceil \frac{s_k}{2} \rceil \geq \frac{r+2}{2}\left(\frac{3}{2}\right)^{k-r} \geq k+1$, as $\frac{1}{2}\left(\frac{3}{2}\right)^{k-r} \geq \frac{k+1}{r+2}$, for $k \geq r+3$. This decrease of at least $k+1$ in potential of the tree pays for the `update-suffix-min` operation that follows the `sift`(x) operation. Finally, if the root node x is a leaf, and e is the last element in $list[x]$, then x and the tree rooted at x are removed. The $k+7$ units of potential of x are more than enough to pay for the `update-suffix-min` operation performed after the removal of x.

When an element e is inserted into a soft heap, a new heap, a new tree and a new node are created. The heap, of rank 0, is assigned one potential unit, the tree zero units, while the new node, of rank 0, is assigned 7 units. During its life time, at most $r+6$ potential units are charged to e to pay for its movements. Finally, additional $r+3$ units are charged to e when it is deleted. If we change the 1 unit of actual work involved in inserting e into a soft heap of its own, and the 8 potential units assigned to this heap to e, we get that the total charge for e, from its insertion until its deletion, is at most $8 + (r+6) + (r+3) = 2r+17 = O(\log \frac{1}{\varepsilon})$. We have thus proved:

THEOREM 4.1. *The amortized cost of inserting an element into a soft heap with error rate ε is $O(\log \frac{1}{\varepsilon})$. The amortized cost of all other operations is 0.*

5 Adding a `delete` operation

We next consider the implementation of `delete` operations. One option, used by Chazelle [4], is to implement `delete` operations in a *lazy* manner. Deleted elements are simply marked as deleted. If an `extract-min` operation returns an element marked as deleted, the operation is simply called again until a non-deleted element is returned. The drawback of such an implementation is that it is not space optimal, as the space used by deleted elements cannot be reclaimed immediately. (This can be fixed by rebuilding the data structure when more than half of the elements are deleted.)

We implement `delete`(e) operations directly as follows. We delete e from the linked list $list[x]$ currently containing it. This can be easily done in constant time, using the forward and backward pointers of e in the list, without knowing the indentity of x. (The node x can be retrieved using a union-find data structure, but this is too expensive in our context.) We do assume, however, that the first element in a linked list knows to which list it belongs to. Thus, if e is the last remaining element of $list[x]$, we can initiate a `sift`(x) operation to bring new elements into $list[x]$. If x is a leaf, it is removed from the data structure. (To implement this we need a pointer to the parent of x in the tree.)

For each node x we keep a number $num[x]$ that gives the number of elements in $list[x]$, including *ghost* elements that were deleted from $list[x]$ or from lists that were appended to $list[x]$. We may thus have $|list[x]| < num[x]$. When $list[left[x]]$ is appended to $list[x]$, we do $num[x] \leftarrow num[x] + num[left[x]]$. We do not decrement $num[x]$ when an element is deleted from $list[x]$, as we do not know when it happens. (In fact, $num[x]$ values should also be maintained by the implementation of Section 2. All occurrences of $|list[x]|$ in the pseudo-code given in Figures 2 and 3 should by replaced by $num[x]$, and $num[x]$ values should be updated when lists are created and moved.)

A moment's reflection shows that the amortized analysis of Section 4 remains valid. Some of the operations are charged to ghost elements, i.e., elements that were already deleted, but this is legal.

6 Comparison with Chazelle's implementation

The main difference between our implementation and Chazelle's [4] implementation is that our implementation uses *binary* trees, whereas Chazelle's implementation uses *binomial* trees. We believe that our implementation is simpler and more intuitive, as we argue below.

Chazelle's binomial trees are *binarized*. Each binomial tree is represented as a binary tree, with each node of the binomial tree corresponding to a *left path* in the binary tree. Thus, only root nodes or nodes that are right children of their parents have elements and keys associated with them. In our binary trees, all nodes play the same role.

The trees in Chazelle's implementation are actually *partial* binomial trees, as tree nodes that remain without elements are deleted. In a standard binomial tree, a node of rank k has exactly k children. In Chazelle's partial binomial trees, a node of rank k may have less than k children. If the number of children of an empty root node of rank k drops below $k/2$, Chazelle resorts to a *clean-up* operation that breaks the tree into a collection of trees. This slightly complicates the implementation and makes the analysis somewhat subtler. No such complications arise in our implementation.

Another important difference between our implementation and Chazelle's implementation is that we explicitly control the number of elements contained in the list of a node of rank k. We believe that this makes our implementation more intuitive and the analysis more transparent.

The way we implement delete operations is also different from the way suggested by Chazelle. Our implementation is automatically space efficient, without the need for periodic rebuildings.

The changes in the implementations enable us to present a simplified and unified amortized analysis.

7 Concluding remarks

We presented a simpler implementation of Chazelle's soft heaps. It would be interesting to find additional applications of this data structure. It would also be interesting to know whether the soft heap operations could be implemented in $O(\log \frac{1}{\varepsilon})$ *worst-case* time.

In the implementation presented, the amortized number of *comparisons* made for each element inserted into soft heaps is $2\log\frac{1}{\varepsilon} + O(1)$. It is possible to reduce this number to $(1+o(1))\log\frac{1}{\varepsilon}$ by maintaining *sufmin* pointers only for trees of rank greater than, say, $2r$ and keeping the *ckey* values of the roots of trees of rank at most $2r$ in a small priority queue.

It is interesting to point out that if we set $r = \infty$ in our construction, i.e., have $size[x] = 1$ for every node x, we get a standard meldable priority queue data structure in which no corruptions occur. Each operation is performed in $O(\log n)$ amortized time, where n is the total number of elements inserted into priority queues. This may be viewed as an alternative to the celebrated *Binomial heaps* data structure of Vuillemin [11] (See also [5]).

As noted by Chazelle [4, 2], soft heaps give rise to a new linear-time median selection algorithm, very different from the algorithms of Blum *et al.* [1], Schönhage *et al.* [10], and Dor and Zwick [6]. It would be interesting to explore the possibility of using soft heaps to obtain an algorithm that finds the median of n elements using less than $2.95n$ comparisons. The best lower bound on the number of comparisons needed to find the median is currently $(2+\varepsilon)n$, for a fixed, but tiny, $\varepsilon > 0$ (see Dor and Zwick [7]).

Acknowledgment

We would like to thank Bob Tarjan for his comments on an earlier version of this paper.

References

[1] M. Blum, R.W. Floyd, V. Pratt, R.L. Rivest, and R.E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[2] B. Chazelle. *The Discrepancy Method: Randomness and Complexity.* Cambridge University Press, 2000. Available on-line at: `http://www.cs.princeton.edu/~chazelle/pubs/book.pdf`.

[3] B. Chazelle. A minimum spanning tree algorithm with inverse-Ackermann type complexity. *Journal of the ACM*, 47(6):1028–1047, 2000.

[4] B. Chazelle. The soft heap: an approximate priority queue with optimal error rate. *Journal of the ACM*, 47(6):1012–1027, 2000.

[5] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to algorithms*. The MIT Press, 2nd edition, 2001.

[6] D. Dor and U. Zwick. Selecting the median. *SIAM Journal on Computing*, 28:1722–1758, 1999.

[7] D. Dor and U. Zwick. Median selection requires $(2+\epsilon)n$ comparisons. *SIAM Journal on Discrete Mathematics*, 14:312–325, 2001.

[8] S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. *Journal of the ACM*, 49(1):16–34, 2002.

[9] S. Pettie and V. Ramachandran. Randomized minimum spanning tree algorithms using exponentially fewer random bits. *ACM Transactions on Algorithms*, 4(1):1–27, 2008.

[10] A. Schönhage, M. Paterson, and N. Pippenger. Finding the median. *Journal of Computer and System Sciences*, 13(2):184–199, 1976.

[11] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21:309–314, 1978.