Soft Sequence Heaps

Gerth Stølting Brodal*

Abstract

Chazelle [8] introduced the soft heap as a building block for efficient minimum spanning tree algorithms, and recently Kaplan et al. [12] showed how soft heaps can be applied to achieve simpler algorithms for various selection problems. A soft heap trades-off accuracy for efficiency, by allowing $\varepsilon N$ of the items in a heap to be corrupted after a total of $N$ insertions, where a corrupted item is an item with artificially increased key and $0 < \varepsilon \leq \frac{1}{2}$ is a fixed error parameter. Chazelle’s soft heaps are based on binomial trees and support insertions in amortized $O(\log \frac{1}{\varepsilon})$ time and extract-min operations in amortized $O(1)$ time.

In this paper we explore the design space of soft heaps. The main contribution of this paper is an alternative soft heap implementation based on merging sorted sequences, with time bounds matching those of Chazelle’s soft heaps. We also discuss a variation of the soft heap by Kaplan et al. [13], where we avoid performing insertions lazily. It is based on ternary trees instead of binary trees and matches the time bounds of Kaplan et al., i.e. amortized $O(1)$ insertions and amortized $O(\log \frac{1}{\varepsilon})$ extract-min. Both our data structures only introduce corruptions after extract-min operations which return the set of items corrupted by the operation.

1 Introduction

Chazelle in 1998 [6] introduced the soft heap as a heap data structure surpassing the comparison lower bounds of heaps by allowing the controlled corruptions of keys, i.e. artificially increasing the values of the keys of a fraction of the inserted items. The power of soft heaps was demonstrated by Chazelle in [7], who showed how soft heaps could be the key ingredient to compute a minimum spanning tree in time $O(m \cdot \alpha(m, n))$, where $\alpha$ is the inverse of Ackermann’s function, and $n$ and $m$ are the number of vertices and edges in the graph, respectively. Pettie and Ramachandran [15, 16] subsequently achieved an optimal comparison based minimum spanning tree algorithm, also using soft heaps, with running time matching the (still unknown) decision-tree complexity of the problem. 20 years later the soft heap paper by Chazelle [6] was awarded the ESA Test-of-Time Award 2018 for its significance on the development of algorithms for the fundamental minimum spanning tree problem.

Even though soft heaps were crucial for advancing the knowledge on the minimum spanning tree problem, their applications have remained surprisingly sparse in the literature otherwise. One could speculate this is due to their unconventional interface. Kaplan et al. [12] recently presented new applications of soft heaps, and in particular strengthened the requirements for the interface to the soft heap operations, to 1) report when an item is considered corrupted internal to a soft heap, 2) to tag returned items if they are corrupted, and 3) restrict corruptions to only be allowed after the extraction of the current minimum from a soft heap. This modified interface allowed in particular a very simple and elegant solution to the binary heap selection problem, a significant simplification compared to the previous significantly more complex solution by Frederickson [10].

In this paper we explore the design-space of soft heaps. The goal of this paper is to present an alternative and simple implementation of soft heaps supporting the interface of Kaplan et al. [12]. In [12] it was described how the soft heap in [13] could support this interface with minor changes. Our solution is based on merging sorted sequences as opposed to all previous solutions which are all based on heap ordered trees. Similar to all previous solutions, our solution also makes essential use of Chazelle’s car-pooling idea.

1.1 Soft heaps Like a normal priority queue, soft heaps store a set of (key, value) pairs called items, where the keys are from an ordered universe. As opposed to a normal priority, soft heaps are allowed to corrupt the keys of the items by artificially increasing the keys. A soft heap trades-off accuracy for efficiency, by allowing up to $\varepsilon N$ of the items in a heap to be corrupted after a total of $N$ insertions, where $0 < \varepsilon \leq \frac{1}{2}$ is a fixed error parameter. Note that the number of allowed corruptions in the heap is $\varepsilon N$, which can be larger than the current number of items $n$ in the soft heap. In particular
is possible that all keys in a soft heap are corrupted when $\varepsilon N \geq n$.

We call the original key of an item the real key and the increased key the current key. A corrupted key can be increased multiple times by the soft heap, but never lowered from its current key. When performing a sequence of insertions and extract-min operations on a soft heap, the soft heap always returns items correctly with respect to their current keys. The effect of corruptions on the extracted sequence is that an item that gets corrupted internally in the soft heap raises a flag that the extraction of the item may appear later in the sequence of extractions because of the artificially high key, i.e. the user might miss out on getting this item extracted in the correct order. When a corrupted item eventually is extracted from a soft heap its current key is an upper bound on its real key, and the extraction allows space for another item to get corrupted in the soft heap.

In this paper we adopt the soft heap interface described Kaplan et al. [12], that explicitly notifies the user about the corruptions introduced. For an application of this interface we refer the reader to the elegant heap selection algorithm in [12, Section 3].

- \textsc{make-heap}(\cdot) creates an empty soft heap $S$ and returns a reference to $S$.
- \textsc{insert}(S, e) inserts item $e = (k, v)$ with real key $k$ and value $v$ into soft heap $S$.
- \textsc{meld}(S_1, S_2) melds the soft heaps $S_1$ and $S_2$, and returns a reference to the resulting soft heap.
- \textsc{find-min}(S) returns a pair $(e, k)$, where $e$ is an item with minimum current key $k$ in the soft heap $S$.
- \textsc{extract-min}(S) removes an item $e$ from the soft heap $S$, where $e$ has minimum current key $k$ before the operation, and returns the triple $(e, k, C)$, where $C$ is the list of items in the heap that were not corrupted before $e$ was removed from $S$, but became corrupted as a result of removing $e$ from $S$.
- \textsc{delete}(S, e) removes item $e$ from the soft heap $S$. Returns a list $C$ of the items where the key became corrupted by removing $e$. Requires a reference is given to the location of $e$ in the soft-heap.

Kaplan and Zwick in [14] gave a simplified construction based on binary trees with matching amortized performance. Kaplan et al. [13] presented a solution where all operations are amortized constant time except for \textsc{extract-min} and \textsc{delete} which take amortized $O(\log \frac{1}{\varepsilon})$ time, i.e. postponing the dependence on $\varepsilon$ to deletions. All these solutions, like ours, use car-pooling to achieve their efficiency. Essentially car-pooling treats a pool of items as a single item, and assigns all the items in the pool current key equal to the maximum real key in the pool. By appropriately maintaining a collection of pools the bound on the total number of corruptions can be guaranteed within the stated time bounds.

For other models of computation, Thorup et al. [19] presented non-comparison based soft heaps for the RAM model achieving amortized $O(\log \log \frac{1}{\varepsilon})$ time per operation, or amortized expected $O(\sqrt{\frac{\log \log \frac{1}{\varepsilon}}{\log \log \log \frac{1}{\varepsilon}}})$ using randomization. Bhushan and Gopalan [1] considered soft heaps in external memory, achieving amortized $O(\frac{1}{\varepsilon} \log \log \frac{1}{\varepsilon} \cdot \log M/B)$ I/Os per insert, and other operations in non-positive amortized I/Os, where $M$ is the main memory size and $B$ the disk block size, provided $N = O(Bm^{0.8} \sqrt{\log \log m})$ where $m = M/B$.

1.2 Applications of soft heaps

The groundbreaking applications of soft heaps are in the mentioned minimum spanning tree algorithms by Chazelle [7] and Pettie and Ramachandran [16]. Further applications were given by Chazelle [8] who showed how soft heaps can lead to alternative solutions for computing exact and approximate medians in linear time, yielding an alternative solution to the classical selection algorithm by Blum et al. [2], algorithms for finding dynamic percentiles, and approximate sorting algorithms with running time $O(n \log \frac{1}{\varepsilon})$ generating sequences with at most $en^2$ inversions or where each element is assigned a rank within $\varepsilon n$ of its true rank.

Kaplan et al. [12] give further applications of soft heaps. Their main contribution is a very simple algorithm to select the $k$-th smallest item in a binary heap in time $O(k)$, significantly simplifying the previous approach by Frederickson [10] that was achieved over a sequence of improvements starting with running time $O(k \log k)$, and then adding ideas to first improve this to $O(k \log \log k)$, then to $O(k^{3\log^* k})$, $O(k2\log^* k)$, and finally $O(k)$. Kaplan et al. then apply the heap selection algorithm to develop various new selection algorithms: an algorithm for selecting the $k$-th smallest item from a row-sorted matrix with $m$ rows in time $O(m \log k)$, matching a previous bound by Frederickson and Johnson [11], and a new algorithm with output sensitive running time of $O(m + \sum_{i=1}^{m} \log (k_i + 1))$, where $k_i$ is the
The soft heap contains at most $\epsilon N$ items, where $X$ and $Y$ are two unordered sets of $m$ and $n$ items respectively, where $m \geq n$, with running time $O(m \lg \frac{k}{n})$ matching a previous bound of Frederickson and Johnson [11]. Chakrabarti et al. [5] used soft heaps in an experimental study on graph conductance search.

1.3 Results The main contribution of this paper is a new implementation of soft heaps, soft sequence heaps, designed to satisfy the interface of Kaplan et al. [12].

**Theorem 1.1.** A soft sequence heap supports insert in amortized $O(\lg \frac{1}{\varepsilon})$ time and all other soft heap operations in amortized constant time, for a fixed error parameter $0 < \varepsilon < 1$. After a total of $N$ insertions the soft heap contains at most $\varepsilon N$ items with corrupted keys.

A (non-soft) sequence heap is a simple priority queue storing its items in a logarithmic number of sorted sequences (see Section 2). In the literature several priority queues exist based on this idea. Examples are external memory priority queues [3, 4], cache efficient priority queues [17], and efficient RAM priority queues [18]. Sanders [17] coined such an approach a sequence heap. Earlier, Fischer and Paterson [9] developed a priority queue aimed at sequential storage also consisting of a sequence of sorted lists. Our contribution is to adapt the car-pooling idea of Chazelle to sequence heaps.

Table 1 summarizes our contributions and contains a comparison of the essential properties of our contributions to previous work.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>$O(\lg \frac{1}{\varepsilon})$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>ExtractMin</td>
<td>$O(1)$</td>
<td>$O(\lg \frac{1}{\varepsilon})$</td>
</tr>
</tbody>
</table>

2 Sequence heaps

A (non-soft) sequence heap stores items in a logarithmic number of sorted sequences $L_1, L_2, \ldots, L_\ell$, where each sequence $L_i$ is assigned a non-negative integer rank $\text{rank}(L_i)$. The sequences are maintained in a list $L$ in increasing rank order. INSERT($e$) creates at the front of $L$ a new rank zero sequence containing $e$, and repeatedly merges the first two sequences of $L$ if they have equal rank $r$ to a new sequence of rank $r + 1$ until all sequences have distinct ranks. EXTRACT-MIN finds the sequence where the first item has minimum key, and removes and returns this item. Figure 1 shows the result of applying INSERT and EXTRACT-MIN to a sequence heap.

That INSERT and EXTRACT-MIN take amortized $O(\lg N)$ time follows from some simple observations: A sequence of rank $r$ contains $2^r$ items (if also counting deleted items), i.e. the maximum rank of a sequence after $N$ insertions is at most $\lceil \lg N \rceil$; an inserted item can at most participate in a number of merges bounded by the maximum rank; and since insertions ensure that the sequences have distinct rank the time for extracting the minimum is also bounded by the maximum rank.

3 Soft sequence heaps

In this section we describe soft sequence heaps derived by adapting Chazelle’s car-pooling idea to sequence heaps. Below, we first describe the basic ideas used to convert sequence heaps into soft-heaps, next we give the details of the representation and the implementation of the operations, and finally we analyse our construction.

3.1 Corruption-sets and witness-sets To make sequence heaps achieve the performance of soft heaps we essentially adopt two ideas. In the following $0 < \varepsilon < 1$ and $r_0 = \lceil \lg \frac{1}{\varepsilon} \rceil$ is a rank threshold.

**Corruption-sets:** With each item $e$ in a sorted sequence we store a corruption-set $C(e)$ containing items where the key eventually should be raised to $\text{key}(e)$. Whenever the merging of two sequences of equal rank $r - 1$ results in a new sequence of rank $r > r_0$, where $r - r_0$ is even, we prune every second item from the sequence (the first and last items in a sequence are
the witness-set \( W \) in a corrupted set \( C \). When we prune an item \( e \) by \( t \) they can influence the order of the items returned (Lemma 3.3 below). To avoid reporting too many corrupted items \( e \) from the head of its sequence.

Witnesses: The above unfortunately only leaves \( o(N) \) items not pruned from the sequences (see Lemma 3.3 below). To avoid reporting too many corruptions we postpone reporting items as corrupted until they can influence the order of the items returned by \textsc{extract-min}. For this purpose we assign a witness to each item when it is initially added to a corruption-set. An item with a witness is not considered corrupted. When we prune an item \( e \) from a sequence its predecessor \( e' \) in the sequence becomes the witness for \( e \). A corrupted item \( e \) in a soft sequence heap is an item \( e \) in a corrupted set \( C(e') \) without a witness. The current key of \( e \) is then the real key of \( e' \). For an item \( e'' \) we let the witness-set \( W(e'') \) be all the items \( e'' \) is a witness for. When we prune \( e \), we add \( e \) and all items from \( C(e) \) to the corruption-set \( C(e') \) of its successor \( e' \) and to the witness-set \( W(e'') \) of its predecessor. To \( W(e'') \) we also add all items from \( W(e) \), i.e., these items get their witness \( e \) replaced by \( e'' \), where key(\( e'' \)) \( \leq \) key(\( e \)). In general, the witness of an item \( e \) is an item \( e'' \) still in the sequence with key(\( e'' \)) \( \leq \) key(\( e \)). When an item \( e'' \) is deleted from the soft sequence heap we report all items in \( W(e'') \) as corrupted.

![Figure 1: A (non-soft) sequence heap. \textsc{insert}(4) first creates a new rank zero sequence (4), that will be merged with the rank zero sequence (2), creating the rank one sequence (2, 4), that will be merged with the rank one sequence (3, 5), finally creating the rank two sequence (2, 3, 4, 5). \textsc{extract-min} removes the smallest item, here 1, from the head of its sequence.](image)

3.2 The representation details

We let \( e \) denote an item in the heap, key(\( e \)) the real key of \( e \), and value(\( e \)) the value of \( e \). A soft sequence heap \( S \) is represented by a list \( L \) of non-empty sequences \( L_1, L_2, \ldots, L_\ell \) of items. Each sequence \( L_i \) has a rank, rank(\( L_i \)), the sequences appear in strictly increasing rank order, i.e. rank(\( L_i \)) \( < \) rank(\( L_{i+1} \)) for \( 1 \leq i < \ell \). The items in \( L_i \) are sorted in increasing order by key. With each item \( e \) in \( L_i \) we store a corruption-set \( C(e) \) and witness-set \( W(e) \), possibly empty, of items pruned from the sorted sequences but still in the heap. Both sets are represented by cyclic linked lists, with entry points to the last items in the lists.

Each item \( e \) is stored in exactly one \( L_i \) sequence or one corruption-set \( C(e') \). If \( e \in C(e') \) then key(\( e \)) \( \leq \) key(\( e' \)), and \( e \) is possibly also stored in one witness-set \( W(e'') \), where key(\( e'' \)) \( \leq \) key(\( e \)). If \( e \in W(e'') \) and \( e \in C(e') \), then \( e' \) and \( e'' \) are in the same \( L_i \) sequence and \( e'' \) occurs before \( e' \) in the sequence. The corrupted items in a sequence are precisely the items contained in a corruption-set but not in a witness-set, and if a corrupted item \( e \in C(e') \) then the current key of \( e \) is key(\( e' \)). A sequence together with its corruption-sets and witness-sets can be viewed as maintaining a partial order, where \( e'' \leq e \) if \( e \in W(e'') \), \( e \leq e' \) if \( e \in C(e') \), and \( e_1 \leq e_2 \) if \( e_1 \) is before \( e_2 \) in the sequence. See Figure 3 for an example.

To efficiently maintain a reference to the current minimum item, for each sequence \( L_i \), we maintain a \textit{suffix-min} reference. The same idea was used by Chazelle [8] and implicitly by Kaplan et al. [13]. For sequence \( L_i \), suffix-min(\( L_i \)) is a reference to sequence \( L_j \), where \( i \leq j \leq \ell \), such that the first item in \( L_j \) has smallest key among the items in \( L_i \cup L_{i+1} \cup \cdots \cup L_\ell \). By definition suffix-min(\( L_i \)) \( = \) \( L_\ell \) and the first item in the sequence suffix-min(\( L_1 \)) has smallest key among all items in all sequences. The reference suffix-min(\( L_i \)) can be updated as follows, assuming suffix-min(\( L_{i+1} \)) is known: If \( i = \ell \) or key(head(\( L_i \))) \( \leq \) key(head(suffix-min(\( L_{i+1} \)))) then suffix-min(\( L_i \)) \( = \) \( L_i \), otherwise suffix-min(\( L_i \)) \( = \) suffix-min(\( L_{i+1} \)). Here head(\( L_i \)) refers to the first item in the sequence.

3.3 Soft heap operations

We now describe how to implement the operations on a soft sequence heap. Given an error parameter \( \varepsilon \), the rank threshold \( r_0 = \lfloor \log_2 \frac{1}{\varepsilon} \rfloor \) allows us to trade accuracy for improved running time. Sequences with rank \( \leq r_0 \) behave as in a (non-soft) sequence heap and all corruption-sets and witness-sets are not pruned. For an item \( e \) to be pruned and with successor \( e' \) in the sequence, we remove \( e \) from the sequence and add \( e \) to \( C(e') \) together with all items from \( C(e) \). This implements the car-pooling idea of Chazelle.
sets are empty in these sequences.

The helper method prune(L) takes a sorted sequence of items L = e1, e2, ..., em, and prunes e2i from L, for all 1 ≤ i < m/2, i.e. the first and last items are not pruned and the pruned sequence has length \([\frac{m+1}{2}]\). Before pruning an item e2i from L, the items \(\{e_{2i}\} \cup C(e_{2i})\) are appended to C(e_{2i+1}), and \(\{e_{2i}\} \cup W(e_{2i})\) are appended to W(e_{2i-1}), and C(e_{2i}) and W(e_{2i}) cease to exist. Since C(e_{2i}) and W(e_{2i}) sets are cyclic linked lists, this can be done in constant time for each e_{2i}.

To support delete operations we apply lazy deletions, where an item is only marked as being deleted, and remains in the soft sequence heap until it becomes the minimum of the soft sequence heap, where it can be deleted using extract-min. We maintain the invariant that the current minimum item of the soft sequence heap is never an item that has been lazily deleted.

Figure 2 illustrates insert and extract-min on a soft sequence heap.

- make-heap() Creates an empty soft heap S with \(L = \emptyset\).
- find-min(S) Let \(L_i = \text{suffix-min}(L_1)\), \(e = \text{head}(L_i)\) and \(k = \text{key}(e)\). Return \((e, k)\) if \(C(e) = \emptyset\), otherwise return \((e', k)\), where \(e' = \text{head}(C(e))\).
- insert(S, e) First we create a new sequence of rank zero only containing e. This sequence is added to the front of \(L\). While the two first sequences of \(L\) have equal rank \(r\), we merge the two sequences into a sequence of rank \(r + 1\), that replaces the two first sequences in \(L\). Whenever creating a sequence \(L\) of rank \(r > r_0\) where \(r - r_0\) is even, we apply prune(L). Finally we update suffix-min(L1) for the new first sequence \(L_1\).
- meld(S1, S2) Let \(L_1\) and \(L_2\) be the two lists of sequences respectively, and let \(r_1\) and \(r_2\) be the maximal ranks of a sequence in \(L_1\) and \(L_2\), respectively. Merge \(L_1\) and \(L_2\) by non-decreasing rank until one of the lists is empty in time \(O(\min(r_1, r_2))\). Let \(L\) be the resulting list. While two sequences in \(L\) have equal rank, merge the two highest ranked sequences of equal rank \(r\) and apply prune if the resulting sequence has rank \(r + 1 > r_0\) and \((r + 1) - r_0\) is even. Update suffix-min(L_i) for the sequences in the new suffix of \(L\) and return a reference to \(L\). (Merging the two highest ranked sequences of equal rank, en-
In the following, we assume for simplicity that all items have distinct keys. For each item $e$ in a corruption-set $C(e')$ we define an interval $I(e)$, with $\text{key}(e) \in I(e)$. If $e$ also has a witness $e''$, i.e. $e \in W(e'')$, we let $I(e) = [\text{key}(e''), \text{key}(e')]$. If $e$ has no witness, we let $I(e) = (-\infty, \text{key}(e')]$. The interval $I(e)$ captures the partial order maintained by the data structure for item $e$. When pruning an item $e$ from a sequence, if we alternate between just merging two sequences and pruning, the item $e$'s witness-set is updated during the pruning operation. The correctness of the operations, in particular \text{FIND-MIN} and \text{EXTRACT-MIN}, follows from the fact that they always return an item with current key equal to the minimum key of all non-corrupted items in $L_1, \ldots, L_s$.

The following lemma states that the pruning done by \text{PRUNE} is quite aggressive, only leaving $o(N)$ items in the sequences. Fortunately, most pruned items will have witnesses and therefore will not be corrupted.

**Lemma 3.3.** For a soft sequence heap the total number of items in $L_1, \ldots, L_s$ is $O(\sqrt{N}/\varepsilon)$.

**Proof.** The total number of items in $L_1, \ldots, L_s$ is bounded by $\sum_{r=0}^{\lfloor \lg N \rfloor} s_r \leq \sum_{r=0}^{r_0} 2^r + \sum_{r=r_0+1}^{\lfloor \lg N \rfloor} (2^{r_0} + 2^r - 1)$.
Lemma 3.4. Over a sequence of heap operations containing \( N \) insertions, the total length of all sequences created is bounded by \( O(N \lg \frac{1}{\epsilon}) \).

Proof. Over time \( N \) insertions can at most create \( \lceil N/2 \rceil \) distinct sequences of rank \( r \) (an item can only once be merged into a sequence of each rank). By Lemma 3.2 and \( r_0 = \lceil \lg \frac{1}{\epsilon} \rceil \), the total length of all sequences ever created by \( N \) insertions is bounded by

\[
\sum_{r=0}^{\lceil \lg N \rceil} \frac{N}{2^r} \cdot s_r = O \left( \sum_{r=0}^{r_0} \frac{N}{2^r} 2^r + \sum_{r=r_0+1}^{\lceil \lg N \rceil} \frac{N}{2^r} 2^{(r+r_0)/2} \right) = O(N \cdot r_0) = O \left( N \cdot \frac{1}{\epsilon} \right).
\]

The following lemma states the amortized running time of the different heap operations. The bound on the number of corrupted items follows by Lemma 3.8.

Lemma 3.5. Soft sequence heaps support \texttt{insert} in amortized \( O(\lg \frac{1}{\epsilon}) \) time, and the remaining operations in amortized constant time.

Proof. Over a sequence of heap operations, involving \( N \) \texttt{insert} operations, a lot of work can only happen once for each inserted item: each item can at most once be pruning and merging corrupted-sets and witness-sets is \( O(N) \). Similarly each item can at most be extracted once from a \( C(e) \) set, deleted once from a sequence by an \texttt{extract-min} operation, and being reported corrupted at most once. All this work takes total time \( O(N) \). The \texttt{make-heap} and \texttt{find-min} operations clearly take \( O(1) \) worst-case time, and do not need to be considered in the following.

The two sources of non-constant work are when creating new sequences by merging sequences and to update suffix-min(\( L_i \)) references whenever the minimum item in a sequence changes. The merging of sequences happens during \texttt{insert} and \texttt{meld}. By Lemma 3.4 the total length of all sequences created over time is bounded by \( O(N \lg \frac{1}{\epsilon}) \). Since creating a sequence \( L_i \) by merging (and possibly followed by \texttt{prune}) takes time \( O(|L_i|) \), the total time for merging sequences is \( O(N \lg \frac{1}{\epsilon}) \).

The suffix-min(\( L_i \)) references need to be updated during \texttt{insert}, \texttt{meld} and \texttt{extract-min}. During \texttt{insert} only suffix-min(\( L_i \)) needs to be updated, which can be done in constant time. If \texttt{extract-min} removes and returns the first item in a rank \( r \) sequence \( L_i \), then the at most \( r + 1 \) references suffix-min(\( L_1 \)), \ldots, suffix-min(\( L_i \)) need to be updated, in time \( O(r) \). There are at most \( N \texttt{extract-min} \) from sequences of rank \( \leq r_0 \), each with costs of at most \( O(r_0) \), i.e. total cost \( O(N \cdot r_0) \). For each of the \( \lceil N/2 \rceil \) sequences ever created of rank \( r > r_0 \), at most \( s_r \) sequence items can be removed by \texttt{extract-min}, each with an update cost of \( O(r) \). The total time for these \texttt{extract-min} becomes

\[
O \left( \sum_{r=r_0+1}^{\lceil \lg N \rceil} \frac{N}{2^r} \cdot s_r \cdot r \right) = O \left( \sum_{r=r_0+1}^{\lceil \lg N \rceil} \frac{N}{2^r} \cdot 2^{(r+r_0)/2} \cdot r \right) = O(N \cdot r_0).
\]

It follows that all \texttt{insert} and \texttt{extract-min} operations take total time \( O(N \cdot r_0) \).

For \texttt{meld} we need to charge the merging of the \( L_1 \) and \( L_2 \) lists, and for updating the suffix-min(\( L_i \)) references. For this we use a separate potential argument. With a soft heap \( L_1, \ldots, L_i \) we assign a potential \( \Phi = \text{rank}(L_i) \), i.e. equal to the maximal rank of a sequence. Since \texttt{INSERT} only can increase the maximum rank by one, this only increases the cost of insertions by an additive term. For \texttt{meld} on two soft sequence heaps with sequences with maximum rank \( r_1 \) and \( r_2 \), respectively, the resulting heap will have a sequence with maximal rank/potential at most \( \max(r_1, r_2) + 1 \). I.e. potential \( \min(r_1, r_2) - 1 \) is released by \texttt{meld}. By charging a constant potential to \texttt{meld}, a total of \( O(\min(r_1, r_2)) \) released potential will be available for performing the merging of \( L_1 \) and \( L_2 \) and for updating the suffix-min(\( L_i \)) references of old sequences. Since at most \( N-1 \) non-trivial \texttt{meld} operations can be performed (where both heaps contain at least one item), the total additional cost for handling \texttt{meld} is \( O(N) \).

The total work of the sequence of operations (except \texttt{make-heap}, \texttt{find-min}, and trivial \texttt{meld} which take worst-case constant time) is \( O(N \lg \frac{1}{\epsilon}) \), which can be charged \( O(\lg \frac{1}{\epsilon}) \) to each insertion, and constant to the remaining operations.

To bound the number of corruptions, we first bound the size of corruption-sets and witness-sets. Let \( c_r \) and \( w_r \) be a bound on the number of items in the corruption-set \( C(e) \) and witness-set \( W(e) \) for \( e \) in a sequence \( L_i \) of rank \( r \).

Lemma 3.6. \( c_r = w_r = 0 \) for \( r \leq r_0 \), and \( c_r = w_r = 2^{\lceil (r-r_0)/2 \rceil} - 1 \) for \( r > r_0 \).
Proof. Merging \( L_i \) sequences does not change \( C(e) \) and \( W(e) \) sets. Only prune\( (L_i) \) add items to \( C(e) \) and \( W(e) \) sets. When pruning \( e \) in a sequence \( L_i \) with rank \( r > r_0 \) and \( r = r_0 \) even, we append \( e \) and \( C(e) \) to \( C(e') \) for the successor \( e' \) of \( e \) in \( L_i \) (and \( e \) and \( W(e) \) to \( W(e'') \) for the predecessor \( e'' \) of \( e \) in \( L_i \)). By only pruning every second item in a sequence, additions to a corruption-set (witness-set) can only come from the predecessor (successor) item in the sequence. We have the recurrence

\[
c_r = \begin{cases} 
0 & \text{for } r \leq r_0 \\
c_{r-1} & \text{for } r > r_0 \text{ and } r - r_0 \text{ odd} \\
2 \cdot c_{r-1} + 1 & \text{for } r > r_0 \text{ and } r - r_0 \text{ even},
\end{cases}
\]

which solves to \( c_r = 2^{|r-r_0|/2} - 1 \) for \( r > r_0 \), since \( c_{r_0+2p} = \cdot (0 \cdot 2 + 1) \cdot 2 + 1 \cdot 2^p = \sum_{i=0}^{p-1} 2^i = 2^p - 1 \). Similarly we have \( w_r = 2^{|r-r_0|/2} - 1 \).

For a sequence \( L_i \) and a possible key value \( x \) we let \( D(L_i, x) \) denote the set of items \( e \) in corrupted-sets in \( L_i \) where the interval \( I(e) \) contains \( x \), i.e.

\[
D(L_i, x) = \{ e | \exists e' \in L_i : e \in C(e') \land x \in I(e) \}.
\]

Note that the corrupted items in \( L_i \) are exactly \( D(L_i, -\infty) \). We let \( d_r \) denote an upper bound on \(|D(L_i, x)|\) for a rank \( r \) sequence \( L_i \), i.e. \( d_r \) is an upper bound on the number of corruptions in a sequence of rank \( r \).

**Lemma 3.7.** \( d_r = 0 \) for \( r \leq r_0 \), and \( d_r = 2^{r-r_0-1} \) for \( r > r_0 \).

**Proof.** Since corruption-sets are empty for \( r \leq r_0 \) we have \( d_r = 0 \) for \( r \leq r_0 \). When merging two sequences \( L_i \) and \( L_{i+1} \) of rank \( r - 1 \) we have \( D(L_i) \cup D(L_{i+1}, x) = D(L_i, x) \cup D(L_{i+1}, x) \), i.e. \( d_r = 2 \cdot d_{r-1} \). If we apply prune to the merged sequence, we prune an item \( e \) with predecessor \( e'' \) and successor \( e' \). This assigns \( I(e) = [\text{key}(e''), \text{key}(e')] \), and the at most \( c_{r-1} \) items in \( C(e) \) all have their interval extended with \([\text{key}(e), \text{key}(e')]\), and the at most \( w_{r-1} \) items in \( W(e) \) all have their interval extended by \([\text{key}(e''), \text{key}(e')]\). Since the pruning of every second item ensures that the prunings affect disjoint intervals of the key space, it follows that prune increases \( d_r \) additionally by at most \( 1 + \max(c_{r-1}, w_{r-1}) \). Since \( c_{r-1} = w_{r-1} \), we get the following recurrence

\[
d_r = \begin{cases} 
0 & \text{for } r \leq r_0 \\
2 \cdot d_{r-1} & \text{for } r > r_0 \text{ and } r - r_0 \text{ odd} \\
2 \cdot d_{r-1} + c_{r-1} + 1 & \text{for } r > r_0 \text{ and } r - r_0 \text{ even}.
\end{cases}
\]

Using \( c_r = 2^{|r-r_0|/2} - 1 \) (Lemma 3.6), for \( r > r_0 \), the recurrence solves to

\[
d_r = \sum_{i=1}^{\lfloor r-r_0 \rfloor} (2^{|r_0+2i-1-r_0|/2} - 1) \cdot 2^r - 2i
\]

\[
= \sum_{i=1}^{\lfloor r-r_0 \rfloor} 2^{i-1} \cdot 2^r - 2i
\]

\[
= 2^{r-r_0-1} \sum_{i=1}^{\lfloor r-r_0 \rfloor} 2^{-i} < 2^{r-r_0-1}.
\]

Note that when extract-min removes the first item \( e \) in \( L_i \) this case all items in \( W(e) \) to lose their witness. But his only happens when \( C(e) = 0 \), i.e. no interval ends at key(\( e \)), and all intervals for items in \( W(e) \) are extended with \([0, \infty, \text{key}(e)]\). It follows after \( e \) is removed we have \( D(L_i, \text{key}(e)^-) = D(L_i, \text{key}(e)^+ \leq d_r \).

**Lemma 3.8.** The total number of corruptions in a soft sequence heap after \( N \) insertions is bounded by \( \varepsilon N \).

**Proof.** Recall that the sequences \( L_1, L_2, \ldots, L_t \) have distinct rank and that the maximum rank is bounded by \( \lceil \log N \rceil \). For a sequence \( L_t \) of rank \( r \), the number of corruptions is \( |D(L_t, -\infty)| \leq d_r \), i.e. by Lemma 3.7 the total number of corruptions is bounded by

\[
\sum_{r=0}^{\lceil \log N \rceil} d_r = \sum_{r=r_0+1}^{\lceil \log N \rceil} 2^{r-r_0-1} = \sum_{i=0}^{\lceil \log N \rceil-r_0-1} 2^i
\]

\[
= 2^{\lceil \log N \rceil-r_0} - 1 < N/2^{r_0} \leq \varepsilon N.
\]

Theorem 1.1 follows from Lemma 3.8 and Lemma 3.5.

### 3.5 Remarks

Essential to our construction is that we reduce the length of the merged sequences to avoid spending \( \Theta(N \log N) \) time on merging sequences during \( N \) insertions. The presented solution is based on binary merging and applies prune at every second rank – inspired by the "double even fill" car-pooling used by Kaplan et al. [13]. Alternatively, one could achieve the same asymptotic bounds by increasing the merging degree to three (or more) and apply reductions at all ranks \( \geq r_0 \). (In Section 4 we apply a similar idea to the structure of Kaplan et al. [13] to eliminate “double even
fill” from their structure by increasing the degree of the nodes in the trees used).

Comparing our construction to previous constructions, our construction maintains a collection of sorted sequences whereas all previous soft heaps maintain heap ordered binomial trees or binary trees. Similar to our and all previous constructions is the application of carpooling to achieve the improved performance over (non-soft) heaps and the usage of corruption-sets. Whereas our solution allows a sub-linear number of items not to be stored in corruption-sets (Lemma 3.3), previous solutions require a larger number of elements not to be stored in corruption-sets, i.e. our solution maintains order among a smaller subset of items not in corruption-sets. To be able to report when elements should be considered corrupted, i.e. to satisfy the the soft heap interface required by Kaplan et al. [12], we apply witness-sets.

Note that the witness-sets can be removed completely from the construction if deletions are not required to return the set of items getting corrupted by a deletion — but witness-sets are still crucial for the analysis to bound the number of corruptions in a soft sequence heap. Interestingly, this implies a structure where only $o(N)$ items are not in corruption-sets, but still guarantees that only $\varepsilon N$ keys need to be considered corrupted.

4 Heap ordered ternary tree based soft-heaps

Kaplan et al. [13] describe a soft heap implementation based on a forest of perfectly balanced binary trees, and Kaplan et al. [12] describe how to modify the structure to support the interface described in Section 1.1. In particular they apply lazy insertions, to circumvent that the original structure might introduce corruptions during insertions. Their structure also adopts the notion “double even fill”. In this section we discuss a variant of their structure that avoids both these concepts. The performance remains unchanged, i.e. all operations are amortized constant (and independent of $\varepsilon$), except EXTRACT-MIN and DELETE which require amortized time $O(\lg \frac{1}{\varepsilon})$. In the following we skip addressing DELETE (which can be handled by lazy deletions) and MELD (which proceeds very similar as for soft sequence heaps).

4.1 Non-soft forest heaps The basic structures are perfectly balanced heap-ordered trees. We describe the construction generalized by a degree parameter $d \geq 3$, although for our result we only need $d = 3$. [This deviates from [13] that uses $d = 2$.] A rank $r$ tree is a perfectly balanced tree with $d^r$ leaves, where all leaves have depth $r$ and all internal nodes have $d$ children. Each leaf corresponds to a unique insertion. A tree is kept heap ordered by recursively pulling items up in the tree, leaving subtrees empty (i.e. nodes without items), such that the root stores the item with minimum value in the tree.

A simple (non-soft) “forest heap” consists of a list $L$ of trees in non-decreasing rank order, with at most $d - 1$ trees of each rank. The insertion of an item $e$ creates a single node rank zero tree at the front of $L$, storing $e$. While the first $d$ trees of $L$ have equal rank $r$, we link these trees to create a rank $r + 1$ tree: create a new rank $r + 1$ node and make the $d$ rank $r$ roots the children of this node, and recursively fill the node with an item by moving an item with minimum key from a child one level up, recursively filling the child until no item can be moved up. Filling the new rank $r + 1$ root takes time $O(d \cdot (r + 1))$. During $N$ INSERT operations at most $\lfloor \frac{N}{d} \rfloor$ roots of rank $r$ are created. Since the maximal rank of a tree is $\lfloor \lg d \cdot N \rfloor$, the total time to link roots during insertions is at most $O(\sum_{r=1}^{\lfloor \lg d \cdot N \rfloor} \lfloor \frac{N}{d} \rfloor \cdot d \cdot r) = O(N)$, i.e. insertions take amortized constant time. An EXTRACT-MIN operation identifies a root with an item with minimum key, removes this item, and recursively refills a rank $r$ root in time $O(d \cdot r)$. Since there at most $(d - 1) \cdot \lfloor \lg d \cdot N \rfloor$ roots, an EXTRACT-MIN operation takes time $O(d \cdot \lg d \cdot N)$.

4.2 Soft forest heaps To improve performance, carpooling is adapted, to avoid moving each item all the way from a leaf to the root. With each item $e$ at a node we store a corruption-set $C(e)$ of corrupted items $e'$ with $\text{key}(e') \leq \text{key}(e)$. Similarly to other soft heap implementations, we maintain suffix-min references for the roots in $L$. The implementation of INSERT proceeds as described above for the non-soft case using repeated linking of $d$ trees of equal rank in amortized $O(1)$ time, except that we also need to update in constant time the suffix-min reference for the resulting first tree in $L$. [This deviates from the solution in [13] that uses “double even fill” during insertions, which can introduce corruptions, and require the insertions to be buffered and be performed lazily.]

For the implementation of EXTRACT-MIN we use a rank threshold $r_0 = \max(2, \lceil \lg \frac{1}{\varepsilon} \rceil)$. We find the root $v$ storing an item $e$ with minimum non-corrupted key in constant time, using the suffix-min reference of the first tree in $L$. If $C(e) \neq \emptyset$, we return a corrupted item from $C(e)$ with current key equal to $\text{key}(e)$, without generating any corruptions. Otherwise, $e$ will be returned with its real key. Before doing so, we need to refill $v$ with a new item and update suffix-min references for all roots in $L$ from right-to-left starting at $v$. The refilling of an empty node is done with a
twist, possibly creating corruptions. Whenever a node of rank \( r \) (i.e. the height of the subtree rooted at the node is \( r \)) is to be refilled, we refill it recursively as in the non-soft case if \( r \leq r_0 \). If \( r > r_0 \) recursively move two items \( e_1 \) and \( e_2 \) with smallest keys from the subtree to the node, key(\( e_1 \)) \leq \text{key}(e_2)\), and make \( e_1 \) corrupted (and to be returned as corrupted by \text{EXTRACT-MIN}) by appending \( e_1 \) and \( C(e_1) \) to \( C(e_2) \), and leave \( e_2 \) as the new item at the node. [This deviates from the solution in [13], that only recursively double fills for even ranks, which limits the number of corruptions introduced when binary linking is applied.] The amortized analysis of \text{EXTRACT-MIN} is given below.

4.3 Analysis To bound the total number of corruptions, let \( c_r \) denote an upper bound on the size of a corruption-set \( C(e) \) for an item \( e \) stored at a node of rank \( r \). Since corruptions only are introduced at nodes with rank \( r > r_0 \), we have \( c_r = 0 \) for \( r \leq r_0 \). The size of \( C(e_2) \) only increases at a node at rank \( r \), when \( e_2 \) and another item \( e_1 \) are moved up from rank \( r-1 \) nodes, and \( e_1 \) and \( C(e_1) \) are appended to \( C(e_2) \). It follows \( c_r = \sum_{i=0}^{r-r_0+1} 2^i = 2^{r-r_0} - 1 \) for \( r > r_0 \). By summing over all possible nodes, the total number of corruptions in a structure is bounded by \( \sum_{r=r_0}^{N} c_r \leq \sum_{r=r_0+1}^{N} N \cdot d \cdot \left( r - r_0 \right) \leq \varepsilon N \), since \( r_0 = \max \{ 2, \left\lceil \frac{\log \frac{1}{\varepsilon}}{\varepsilon} \right\rfloor \} \).

Next we bound the time spent on refilling nodes during \( \delta \) \text{EXTRACT-MIN} operations. After \( N \) insertions there are at most \( N/2^{r_0} \) nodes with rank \( > r_0 \), and \( N/2^{r_0} \) items in the corrupted-sets. Together with the \( \delta \) deleted items, a total of at most \( \delta + 2N/2^{r_0} \) items need to have moved up to nodes with rank \( > r_0 \). The recursive pull of an item from rank \( r_0 \) to \( r_0 + 1 \) takes worst-case \( O(d \cdot r_0) \) time, i.e. over all \text{EXTRACT-MIN} operations we spend \( O(d \cdot r_0 \cdot \delta + N/2^{r_0}) = O(d \cdot N + d \cdot r_0 \cdot \delta) \) time on recursively filling nodes at ranks \( \leq r_0 \). For moving items up to nodes at rank \( > r_0 \), we observe that whenever we move two items one level up, one of the items gets corrupted — except for the last possible item being moved into a node before the subtree becomes empty. Since an item can at most get corrupted once, and there are at most \( N/2^{r_0} \) nodes with rank \( > r_0 \), at most \( O(N) \) times an item is moved one level up. We conclude that a sequence with \( N \) \text{INSERT} and \( \delta \) \text{EXTRACT-MIN} operations in total spend \( O(d \cdot N + d \cdot r_0 \cdot \delta) \) time on recursively pulling items up during the \text{EXTRACT-MIN} operations.

Finally, we consider the time to update the suffix-min pointers during \text{EXTRACT-MIN}. If the item at a root of rank \( r \) changes, at most \( (d - 1) \cdot (r + 1) \) suffix-min references need to be updated in time \( O(d \cdot r) \). We charge the cost for updating the suffix-min references at the roots of rank \( \leq r_0 \) directly to the \text{EXTRACT-MIN} operation, i.e. \( O(d \cdot r_0) \). For updating roots of ranks \( r_0 + 1 \) to \( r \) we just consider a very rough bound on the total number of different items that can become the root of rank \( r \) trees. At most \( \lfloor N/\varepsilon \rfloor \) trees are ever created of rank \( r \), each such tree contains at most \( \varepsilon \) items, of which only a fraction \( \Theta(\frac{1}{\varepsilon}) \) can reach the root, due to the pruning of every second item reaching nodes of rank \( r_0 + 1, \ldots, r \). In total

\[
O \left( \frac{N}{\varepsilon} \cdot d^r \cdot \frac{1}{2^{r-r_0}} \right)
\]

different items can become the root of rank \( r \) trees. Charging updating \( O(d \cdot (r - r_0)) \) suffix-min references to each of these items, and summing over all ranks gives

\[
O \left( \sum_{r=r_0+1}^{N} \frac{1}{2^{r-r_0}} \cdot d \cdot (r-r_0) \right) = O(N \cdot d)
\]
as an upper bound of updating suffix-min references. We conclude that a sequence with \( N \) \text{INSERT} and \( \delta \) \text{EXTRACT-MIN} operations requires total time

\[
O(d \cdot N + d \cdot r_0 \cdot \delta) = O \left( N + \delta \log \frac{1}{\varepsilon} \right),
\]

for \( d = 3 \) and \( r_0 = \max \{ 2, \left\lceil \frac{1}{\varepsilon} \right\rfloor \} \), i.e. \text{INSERT} takes amortized constant time and \text{EXTRACT-MIN} amortized \( O(\log \frac{1}{\varepsilon}) \) time.

References


Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited.


