The amortized cost of finding the minimum

Haim Kaplan [*] Or Zamir [†] Uri Zwick [‡]

Abstract

We obtain an essentially optimal tradeoff between the *amortized* cost of the three basic priority queue operations `insert`, `delete` and `find-min` in the comparison model. More specifically, we show that

$$A(\texttt{find-min}) \;=\; \Omega\left(\frac{n}{(2+\epsilon)^{A(\texttt{insert})+A(\texttt{delete})}}\right) \;,$$

$$A(\texttt{find-min}) \;=\; O\left(\frac{n}{2^{\,A(\texttt{insert})+A(\texttt{delete})}}+\log n\right) \;,$$

for any fixed $\epsilon > 0$, where n is the number of items in the priority queue and $A(\texttt{insert})$, $A(\texttt{delete})$ and $A(\texttt{find-min})$ are the *amortized* costs of the `insert`, `delete` and `find-min` operations, respectively. In particular, if $A(\texttt{insert}) + A(\texttt{delete}) = O(1)$, then $A(\texttt{find-min}) = \Omega(n)$, and $A(\texttt{find-min}) = O(n^\alpha)$, for some $\alpha < 1$, only if $A(\texttt{insert}) + A(\texttt{delete}) = \Omega(\log n)$. (We can, of course, have $A(\texttt{insert}) = O(1)$, $A(\texttt{delete}) = O(\log n)$, or vice versa, and $A(\texttt{find-min}) = O(1)$.) Our lower bound holds even if *randomization* is allowed. Surprisingly, such fundamental bounds on the amortized cost of the operations were not known before. Brodal, Chaudhuri and Radhakrishnan, obtained similar bounds for the *worst-case* complexity of `find-min`.

1 Introduction

A *priority queue* (also known as a *heap*) is a basic data structure that maintains a collection S of *items*, each with an associated *key* (or *priority*) taken from a totally ordered universe. The data structure supports the following operations:

- `insert`(x): Insert item x into S.
- `delete`(x): Delete item x from S.
- `find-min`: Return an item with minimal key in S.

We consider priority queue data structures that work in the *comparison model*, i.e., data structures that can only access keys via comparisons. When proving lower bounds, the *time* (or *cost*) of an operation is taken to be the number of comparisons performed while executing it. When proving upper bounds, the time includes all operations. Classical data structures, such as *binary heaps* [22] and *balanced search trees* [1, 16] perform all three operations in $O(\log n)$ worst-case time, where n is the current number of items in the heap. Any such data structure can be converted into a data structure that performs `insert` and `find-min` in $O(1)$ worst-case time and `delete` in $O(\log n)$ worst-case time (see Alstrup et al. [2]). Various priority queue data structures achieve this directly [6, 7, 9, 17, 8]. Similarly, it is possible to implement `insert` in $O(\log n)$ worst-case time and `delete` and `find-min` in $O(1)$ worst-case time [13, 18].

As n items can be sorted using n `insert`, n `find-min` and n `delete` operations, and as sorting requires at least $\Omega(n \log n)$ comparisons, we immediately get that at least one of these three operations must have a worst-case cost, and in fact also an *amortized* cost, of $\Omega(\log n)$. We saw above that if one of the update operations `insert` or `delete` has a worst-case cost of $\Theta(\log n)$, then `find-min` and the other update operation can be implemented in $O(1)$ time. A natural question is then: Is it possible to implement `insert` and `delete` in $O(1)$ time and `find-min` in $O(\log n)$ time? More generally, if `insert` and `delete` are allowed only $O(1)$ time, how fast can `find-min` operations be implemented? Time here can refer to both worst-case or amortized time. We can also consider deterministic or randomized data structures. Perhaps the most interesting variant is when all costs are *amortized*, as in many situations we are more interested in the total cost required to execute a sequence of operations, rather than the cost of individual operations. Surprisingly, the amortized versions of these fundamental questions were not answered before.

Brodal et al. [5] address a mixed (randomized) version of the problem. They show that if the amortized

[*]Blavatnik School of Computer Science, Tel Aviv University, Israel. Research supported by The Israeli Centers of Research Excellence (I-CORE) program (Center No. 4/11), the Israel Science Foundation grant no. 822-10, and the German-Israeli Foundation for Scientific Research and Development (GIF) grant no. 1161/2011. E-mail: `haimk@post.tau.ac.il`.

[†]Blavatnik School of Computer Science, Tel Aviv University, Israel. E-mail: `orzamir@mail.tau.ac.il`.

[‡]Blavatnik School of Computer Science, Tel Aviv University, Israel. Research supported by BSF grant no. 2012338 and by The Israeli Centers of Research Excellence (I-CORE) program (Center No. 4/11). E-mail: `zwick@tau.ac.il`.

(expected) cost of `insert` and `delete` operations is at most t, then the *worst-case* (expected) cost of `find-min` is $\Omega(n/2^{2t})$. We build on the approach of Brodal et al. [5] and resolve the more natural amortized version of the problem. More specifically, we show that if the amortized (expected) cost of `insert` and `delete` operations is at most t, then the amortized (expected) cost of `find-min` operations $\Omega(n/(t2^{2t}))$. (We loose a factor of t in the denominator.) Extending the lower bound of Brodal et al. [5] to the amortized case requires additional non-trivial ideas. We note that there are some data structure problems for which there are substantial gaps between the worst-case and amortized costs of some operations. For example, in the *union-find* problem, there is an $\Omega(\log n/\log\log n)$ lower bound on the worst-case complexity and an $O(\alpha(m,n))$ upper bound on the amortized complexity of the operations, where $\alpha(m,n)$ is the inverse Ackermann function (see Fredman and Saks [15] and Tarjan [20]).

We also show that our $\Omega(n/(t2^{2t}))$ lower bound on the amortized cost of `find-min` is almost tight by describing a lazy version of *binomial heaps* [21] for which the amortized cost of `insert` and `delete` are both t while the amortized cost of `find-min` is $O(n/2^{2t}+\log n)$. (We can also take the amortized cost of `insert` to be 1, and that of `delete` to be $2t-1$, which is slightly stronger as some items may be inserted but not deleted.) An interesting feature of this data structure is that it achieves these bounds simultaneously for all values of t. This improves on a simple data structure of Brodal et al. [5] in which the worst-case cost of `find-min` operations is $O(n/2^t)$. (Note that this bound has 2^t, rather than 2^{2t}, in the denominator.)

To prove our lower bound on the amortized cost of `find-min`, we show that for any (randomized) priority queue data structure, and for any k that divides n, there exists a sequence of the form:

$$n \times \texttt{insert}\ ,\ n/k \times (\ \texttt{find-min}\ ,\ k \times \texttt{delete}\ )$$

on which the data structure performs $\Omega(n\log\frac{n}{k})$ comparisons. In the sequences used, each deleted item is a minimal item. We call such sequences *canonical* sequences. The amortized lower bound then follows using a simple calculation. Brodal et al. [5] obtained their worst-case bound on the cost of `find-min` using sequences that contain a single `find-min` operation.

Following Brodal et al. [5], we actually give two lower bounds on the amortized cost of `find-min` operations. The first one uses a generalization of the comparison tree technique used to obtain the $\Omega(n\log n)$ lower bound for sorting that goes back to Ford and Johnson [14]. The second lower bound uses an explicit adversary that answers comparisons made by any priority queue data structure. The explicit adversary we use is an extension of the adversary of [5], which in turn is an extension of an adversary devised by Borodin et al. [4]. The lower bound obtained using the explicit adversary is weaker than the lower bound obtained using the comparison tree approach, and it can only be used to obtain a lower bound for deterministic data structures. The advantage of the explicit adversary is that it can be used to *efficiently* answer comparisons made by the algorithm, in an *on-line* manner, in a way that forces the data structure to perform many comparisons. Furthermore, the explicit adversary does that without examining the 'internal structure' of the data structure.

The rest of the paper is organized as follows. In Section 2 we introduce some basic notions and definitions. In Section 3 we obtain our lower bound on the amortized (expected) cost of `find-min` operations. In Section 4 we obtain an almost matching upper bound. In Section 5 we obtain an alternative proof of the amortized lower bound for the cost of `find-min` operations using an explicit adversary. In Section 6 we use our lower bounds to obtain a lower bound on the cost of `delete` operations that delete a *non-minimal* item. We end in Section 7 with some concluding remarks and open problems.

2 Preliminaries

We begin with a formal definition of amortized costs.

Definition 2.1. (**Amortized cost**) *A data structure supports operation types $OP_1,\ldots,OP_k$ in* amortized costs *$f_1(n),...,f_k(n)$, respectively, if and only if, it executes every sequence $op_1,\ldots,op_m$ of operations, starting from an empty data structure, using a total cost of at most $\sum_{i=1}^{m} f_{y_i}(n_i)$, where OP_{y_i} is the type of operation op_i, and n_i is the number of items in the data structure at the time op_i is executed, for $1\le i\le m$.*

The *cost* of an operation can be measured in *time*, i.e., number of basic computational steps needed to perform the operation, or in *comparisons*, i.e., the number of pairwise comparisons needed to perform the operation. We can thus speak about amortized time or amortized number of comparisons.

We usually assume that the amortized cost functions $f_1(n),\ldots,f_k(n)$ are non-decreasing in n. If the sequence $op_1,\ldots,op_m$ contains m_j operations of type OP_j, for $1\le j\le k$, and the maximum number of items in the data structure during the execution of the sequence is at most n, then the total cost of performing the operations is at most $\sum_{j=1}^{k} m_j f_j(n)$.

We consider priority queue data structures that work in the *comparison model*. Our lower bounds are on the number of comparisons made while implementing the various operations. In our upper bounds, we take into account all operations, not only comparisons.

To obtain lower bounds on the (amortized) cost of various priority queue operations, we need to exhibit, for every possible implementation, a sequence of operations that forces the data structure to perform many comparisons. The sequences we use have the following simple and natural form:

Definition 2.2. (**Canonical sequences**) *Let $a_1, a_2, \ldots, a_n$ be a sequence of items. Every permutation $\sigma \in S_n$ defines a total order $a_{\sigma(1)} < a_{\sigma(2)} < \cdots < a_{\sigma(n)}$ on the items. For every $\sigma \in S_n$ and $1 \le k < n$, such that k divides n, we define the following sequence SEQ(σ, k) of priority queue operations:*

$$
\begin{array}{cccc}
\multicolumn{4}{c}{\texttt{insert}(a_1)\ ,\ \texttt{insert}(a_2)\ ,\ \ldots\ ,\ \texttt{insert}(a_n)\ ,} \\
\texttt{find-min}\ , & \texttt{delete}(a_{\sigma(1)}) & ,\ldots, & \texttt{delete}(a_{\sigma(k)}), \\
\texttt{find-min}\ , & \texttt{delete}(a_{\sigma(k+1)}) & ,\ldots, & \texttt{delete}(a_{\sigma(2k)}), \\
\vdots & \vdots & \vdots & \vdots \\
\texttt{find-min}\ , & \texttt{delete}(a_{\sigma(n-k+1)}) & ,\ldots, & \texttt{delete}(a_{\sigma(n)}).
\end{array}
$$

The first row above contains n `insert` operations. Each subsequent row contains one `find-min` operation followed by k `delete` operations. Note that in a canonical sequence, an item about to be deleted is always the smallest item currently in the priority queue. The parameter k is determined by the amortized costs assigned to `insert` and `delete` operations. We assumed, for simplicity, that k divides n. If not, the last row should be

$$\texttt{find-min},\ \texttt{delete}(a_{\sigma(k(\lceil \frac{n}{k} \rceil - 1)+1)})\ ,\ldots,\ \texttt{delete}(a_{\sigma(n)}).$$

A canonical sequence contains n `insert` operations, n `delete` operations, and $\lceil n/k \rceil$ `find-min` operations.

To obtain a lower bound for *randomized* data structures, we use the celebrated Yao's min-max principle (Yao [23]) saying that to obtain a lower bound on the expected number of operations performed by a randomized algorithm, it is enough to describe a *distribution* of input instances that forces every *deterministic* algorithm to perform a large expected number of operations. The distribution we use is particularly simple; we use each one of the $n!$ canonical sequences SEQ(σ, k) with probability $1/n!$.

The operation of any comparison-based priority queue data structure on canonical sequences of operations, for a fixed value of the parameter k, can be described by a *comparison-deletion tree* that extends the notion of *comparison trees* used to obtain a lower bound on the number of comparisons performed by a comparison-based sorting algorithm. A similar notion is used in Brodal et al. [5].

Definition 2.3. (**Comparison-deletion trees**) *A* comparison-deletion *tree is a rooted tree composed of* comparison *nodes,* deletion *nodes and* leaves. *Each comparison node is labeled by two integers $i : j$, signifying a comparison of items a_i and a_j, and has two children corresponding to the two possible outcomes $a_i < a_j$ and $a_i > a_j$. (We assume, for simplicity, that all keys are distinct.) For each node v of the tree, we let $\prec_v$ be the partial order on the items corresponding to the outcomes of all comparisons on the path from the root to v. We let* $\min(v)$ *denote the indices of the items that are* minimal *with respect to $\prec_v$. (An item a_i is* minimal *with respect to $\prec_v$ if there is no other item a_j for which $a_j \prec_v a_i$.) Each deletion node v has a child v_j for every $j \in \min(v)$. The edge from v to v_j is labeled j and signifies the deletion of item a_j.*

A portion of a comparison-deletion tree, describing the behavior of a particular data structure on canonical sequences with $n = 4$ and $k = 2$, is given in Figure 1.

A comparison-deletion tree does not contain explicit *insert* and *find-min* nodes, as they are not needed for canonical sequences of operations. *Insert* nodes are not needed as all insertions take place at the beginning of the sequence. As we are interested in amortized bounds, we may assume that no comparisons are performed before all items are inserted. We may, in fact, assume that all comparisons are performed in response to `find-min` operations. Explicit *find-min* nodes are not needed as in canonical sequences we know that `find-min` operations are performed immediately before the first `delete` operation, before the $(k + 1)$-st `delete` operation, etc.

Lemma 2.1. *A comparison-deletion tree describes a comparison-based implementation of a priority-queue data structure that correctly manipulates all canonical sequences SEQ(σ, k), where $\sigma \in S_n$ and k divides n, if and only if every root to leaf path contains exactly n deletion nodes, arranged in n/k groups of k consecutive deletion nodes, and the number of children of the $(ik + 1)$-st deletion node on the path, for $0 \le i < n/k$, is exactly* 1. *Such a comparison-deletion tree contains exactly $n!$ leaves, each corresponding to a unique permutation $\sigma \in S_n$ that determines a total order of the items.*

Proof. The $(ik+1)$-st deletion node on each root to leaf path corresponds exactly to the position of a `find-min` operation. The data structure can correctly report the minimal item, with no further comparisons, if and only if the partial order corresponding to this deletion node contains exactly one minimal item. As the item deleted in a canonical sequence is always the minimal item

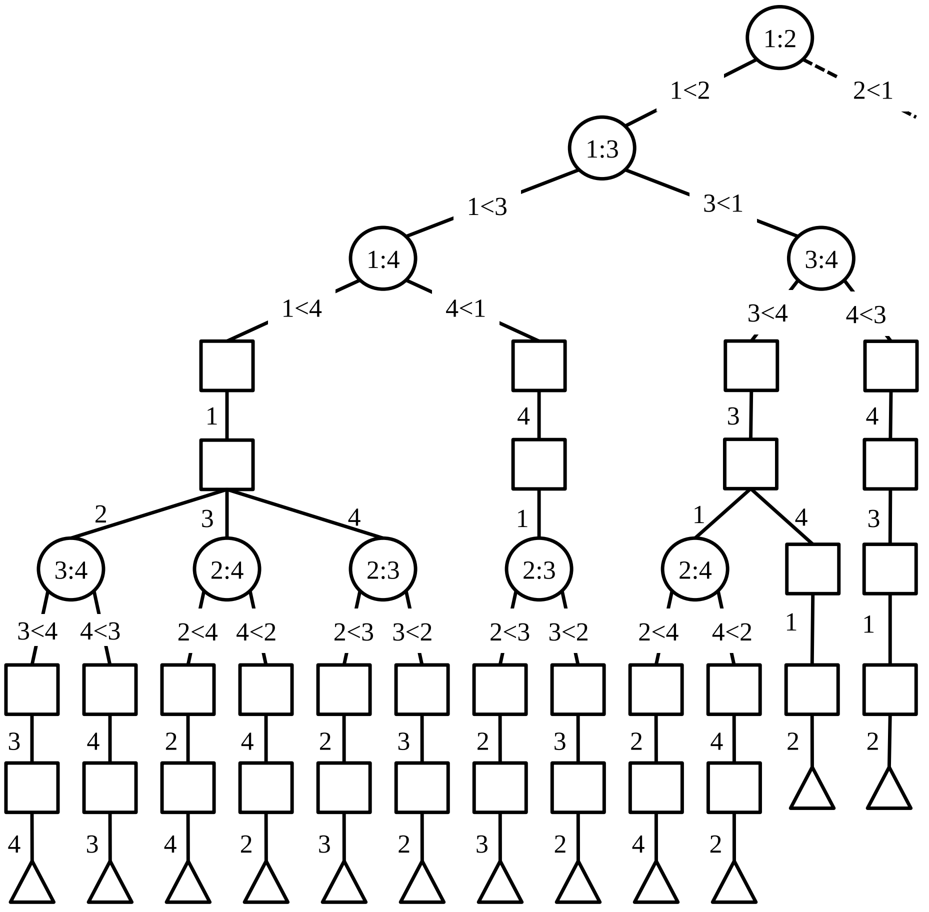

Figure 1: A portion of a comparison-deletion tree corresponding to the case $n = 4$ and $k = 2$. Circles are comparison nodes, squares are deletion nodes while triangles are leaves. The numbers adjacent to edges connecting deletion nodes with their children indicate the index of the item deleted.

contained in the priority queue, the order in which the items are deleted on a given root to leaf path uniquely determines a permutation $\sigma \in S_n$. $\square$

In the next section, we show that each comparison-deletion tree that corresponds to a valid priority queue data structure, contains at least one root to leaf path that contains $\Omega(n \log \frac{n}{k})$ comparison nodes. (The same lower bound holds, in fact, for the *average* number of comparison nodes on all root to leaves paths.) Such a root to leaf path corresponds to a way of answering comparisons made by the data structure in a way that forces it to perform many comparisons. This existence proof, however, is *non-constructive*; to identify such a path we essentially need to construct the whole comparison-deletion tree of the data structure, which is exponential in size.

In Section 5 we obtain an alternative proof of a slightly weaker version of the lower bound using an explicit *adversary* which supplies an efficient way of answering comparisons made the data structure, forcing it to perform many comparisons.

3 Lower bound on amortized cost of `find-min`

We are now ready for the main theorem of this paper.

THEOREM 3.1. (AMORTIZED COST OF `find-min`) *For any, possibly randomized, comparison-based priority queue data structure, if $A(\texttt{insert}) + A(\texttt{delete}) \le 2t$, where $t \ge 4$, then $A(\texttt{find-min}) = \Omega(\frac{n}{t\,2^{2t}})$.*

Here $A(\texttt{insert})$, $A(\texttt{delete})$ and $A(\texttt{find-min})$ are the amortized (expected) costs of the `insert`, `delete` and `find-min` operations, respectively. In the statement of the theorem, $t = t(n)$ is any non-decreasing function of n.

The proof of Theorem 3.1 uses the notion of comparison-deletion trees defined in the previous section. If T is a tree and v a node of T, we let $\deg(v)$ be the number of children of v. We let $Leaves(T)$ be the set of *leaves* of T. If $\ell \in Leaves(T)$, we let $path(\ell)$ be the set of nodes on the path from the root of T to ℓ, not including ℓ. The following simple lemma is used by McDiarmid [19] and Brodal et al. [5]. For completeness we include a proof.

LEMMA 3.1. *For a rooted tree T with m leaves,*

$$\prod_{\ell \in Leaves(T)} \prod_{v \in path(\ell)} \deg(v)^{\frac{1}{m}} \geq m \; .$$

Proof. Let

$$W_\ell = \prod_{v \in path(\ell)} \deg(v) \; .$$

As $\frac{1}{W_\ell}$ is the probability that a random walk that starts at the root and at each node chooses a child uniformly at random, reaches leaf ℓ, we have

$$\sum_{\ell \in Leaves(T)} \frac{1}{W_\ell} = 1 \; .$$

Since the geometric mean is not larger than the arithmetic mean, we obtain that

$$\prod_{\ell \in Leaves(T)} \left(\frac{1}{W_\ell}\right)^{\frac{1}{m}} \leq \frac{1}{m} \sum_{\ell \in Leaves(T)} \frac{1}{W_\ell} = \frac{1}{m} \; .$$

The lemma follows by taking reciprocals. □

We are now ready to prove Theorem 3.1.

Proof. (**of Theorem 3.1**) Let $\mathcal{D}$ be a deterministic priority queue data structure. For any $k < n$, we begin by showing that the expected number of comparisons performed by $\mathcal{D}$ on a random canonical sequence $SEQ(\sigma, k)$, where σ is chosen uniformly at random from S_n, is at least $n(\lg \frac{n}{k} - \lg \lg \frac{n}{k} - \lg \mathrm{e}) - 1$. [1]

Let $T = T_{\mathcal{D}}$ be the comparison-deletion tree corresponding to $\mathcal{D}$. By Lemma 2.1 we know that T has $n!$ leaves and that for every leaf ℓ, the path from the root of T to ℓ contains exactly n deletion nodes. More specifically, the path from the root of T to a leaf ℓ is composed of some comparison nodes, then k deletion nodes, then some more comparison nodes, followed again by k deletion nodes, and so on. (If k does not divide n, then the last group of deletion nodes contains less than k nodes.) Each consecutive sequence of comparison nodes is naturally associated with one of the `find-min` operations of the sequence.

[1] We use $\lg n = \log_2 n$ to denote base 2 logarithms.

For every leaf ℓ of $T = T_{\mathcal{D}}$, we let $c_i(\ell)$ be the number of comparisons associated with the i-th `find-min` operation. We let $d_{i,j}(\ell)$ be the degree of the j-th deletion node following the i-th `find-min` operation. (In other words, $c_i(\ell)$ is the number of comparison nodes immediately preceding the $((i-1)k+1)$-st deletion node, and $d_{i,j}(\ell)$ is the degree of the $((i-1)k+j)$-th deletion node on the path to ℓ.)

For every leaf ℓ, let $c(\ell) = \sum_{i=1}^{n/k} c_i(\ell)$ be the total number of comparisons performed on the path to ℓ. Note that $L = \frac{1}{n!}\sum_\ell c(\ell)$ is precisely the expected number of comparisons performed by $\mathcal{D}$ on a uniformly random canonical sequence $SEQ(\sigma, k)$.

By Lemma 2.1, we know that $d_{i,1}(\ell) = 1$, for every $1 \leq i \leq n/k$. (This, as we saw, is equivalent to saying that the data structure has enough information to answer the i-th `find-min` operation.) The degree of a deletion node is the number of minimal items in the partial order corresponding to it. When a minimal item is deleted, the number of minimal items is reduced by at most 1. Thus $d_{i,j}(\ell) \geq d_{i,j-1}(\ell) - 1$, for every $1 \leq i \leq n/k$ and $1 \leq j \leq k$.

To answer the first `find-min` operation $\mathcal{D}$ must perform at least $n-1$ comparisons. Thus $c_1(\ell) \geq n-1$, for every leaf ℓ. Also, after the first ik items are deleted, the number of minimal items in the partial order is at least $d_{i,k}(\ell) - 1$, so the number of comparisons required to answer the $(i+1)$-st `find-min` is at least $d_{i,k}(\ell) - 2$. Thus $c_{i+1}(\ell) \geq d_{i,k}(\ell) - 2$. Combining these inequalities, we get, for $j \geq 2$,

$$\begin{aligned} d_{i,j}(\ell) \; &\leq \; d_{i,k}(\ell) + (k-j) \\ &\leq \; c_{i+1}(\ell) + 2 + (k-j) \; \leq \; c_{i+1}(\ell) + k \; . \end{aligned}$$

For every leaf $\ell \in Leaves(T)$, let

$$\begin{aligned} W_\ell \; &= \; \prod_{v \in path(\ell)} \deg(v) \\ &= \; \left(\prod_{i=1}^{n/k} 2^{\,c_i(\ell)}\right) \cdot \left(\prod_{i=1}^{n/k} \prod_{j=1}^{k} d_{i,j}(\ell)\right) \\ &\leq \; 2^{\,c(\ell)} \cdot \prod_{i=1}^{n/k} (c_{i+1}(\ell) + k)^{k-1} \; , \end{aligned}$$

where we let $c_{\frac{n}{k}+1}(\ell) = 0$. By Lemma 3.1 we have

$$n! \; \leq \; \left(\prod_\ell W_\ell\right)^{\frac{1}{n!}}$$

$$\leq \; \left(\prod_\ell 2^{c(\ell)} \cdot \prod_{i=1}^{n/k} (c_{i+1}(\ell) + k)^{k-1}\right)^{\frac{1}{n!}} \; ,$$

or after taking logarithms

$$\lg n! \le \frac{1}{n!}\sum_{\ell} c(\ell) + \frac{k-1}{n!}\sum_{\ell}\sum_{i=1}^{n/k}\lg(c_{i+1}(\ell)+k)$$

$$\le \frac{1}{n!}\sum_{\ell} c(\ell) + \frac{k-1}{n!}\cdot n!\cdot\frac{n}{k}\lg\frac{\sum_\ell\sum_{i=1}^{n/k} c_{i+1}(\ell)+k}{n!\cdot\frac{n}{k}}\ ,$$

where the second inequality follows from the inequality $\sum_{i=1}^m \lg x_i \le m\lg\frac{\sum_{i=1}^m x_i}{m}$. Recalling that $c_1(\ell)\ge n-1$ and $c_{\frac{n}{k}+1}(\ell)=0$, we get

$$\sum_{i=1}^{n/k}(c_{i+1}(\ell)+k) \le (\sum_{i=1}^{n/k} c_i(\ell)) - (n-1) + n = c(\ell)+1\ .$$

Thus,

$$n\lg\frac{n}{\mathrm{e}} < \lg n!$$

$$< \frac{1}{n!}\sum_\ell c(\ell) + n\lg\left(\frac{k}{n}\cdot\frac{1}{n!}\sum_\ell (c(\ell)+1)\right)$$

$$< (L+1) + n\lg\left(\frac{k}{n}(L+1)\right)\ ,$$

where, as above, $L=\frac{1}{n!}\sum_\ell c(\ell)$ is the expected cost of random canonical sequence. Dividing both sides by n and moving things around, we get

$$\lg\frac{n}{k} - \lg\mathrm{e} < \frac{L+1}{n} + \lg\frac{L+1}{n}\ .$$

It is easy to check that $x+\lg x\ge y-c$, where $c>0$, implies $x>y-\lg y-c$. Thus,

$$\frac{L+1}{n} > \lg\frac{n}{k} - \lg\lg\frac{n}{k} - \lg\mathrm{e}\ ,$$

and finally

$$L > n\left(\lg\frac{n}{k} - \lg\lg\frac{n}{k} - \lg\mathrm{e}\right) - 1\ .$$

Now, if $A(\texttt{insert}) + A(\texttt{delete}) = 2t$, then

$$A(\texttt{find-min}) \ge \frac{L-2t\cdot n}{n/k}$$

$$> k\left(\lg\frac{n}{k} - \lg\lg\frac{n}{k} - \lg\mathrm{e} - 2\mathrm{t}\right) - 1\ .$$

Letting $k=n/2^s$, where $s=2t+\lg(2t)+\lg e+2$, and assuming $t\ge 4$, we get

$$A(\texttt{find-min}) > \frac{n}{2^s}(s-\lg s-\lg\mathrm{e}-2t) - 1$$

$$\ge \frac{n}{2^s}(\lg(2t)+2-\lg s) - 1 \ge \frac{n}{2^s} - 1 = \frac{1}{8\mathrm{e}}\frac{n}{t\,2^{2t}} - 1\ .$$

□

4 Upper Bound

In this section we present a simple deterministic data structure, a lazy version of *binomial heaps* [21], for which we can prove the following theorem, showing that the lower bound of the preceding version is almost tight.

THEOREM 4.1. *A* Lazy binomial heap *performs* `insert` *using* 1 *amortized comparison and* $O(1)$ *time,* `delete` *using* $2t-1$ *amortized comparisons and* $O(t)$ *time, and* `find-min` *using* $O(\frac{n}{2^{2t}}+\log n)$ *amortized comparisons and time. (Here,* $t=t(n)\ge 1$ *is any non-decreasing positive function.)*

We note that the $+\log n$ term in the amortized cost of `find-min` in Theorem 4.1 is significant only when $A(\texttt{insert})+A(\texttt{delete}) = 2t \ge \lg n - \lg\lg n$. Using an adaptation of priority queues of Elmasry et al. [12] and Edelkamp et al. [10, 11] it is possible to obtain a data structure that supports `insert` using $O(1)$ amortized comparisons, `find-min` requires no comparisons, and `delete` requires $\lg n + O(1)$ amortized comparisons. Thus, for $A(\texttt{insert})+A(\texttt{delete}) = 2t = \lg n + \Omega(1)$, the $+\log n$ term in Theorem 4.1 can be avoided by using a different data structure. The necessity of the $+\log n$ term in the small range $\lg n - \lg\lg n \le 2t \le \lg n + O(1)$ remains unclear.

4.1 Binomial heaps

We begin with a quick review of Binomial heaps. A *binomial tree* of rank k, denoted B_k, is defined recursively as follows: B_0 is composed of a single node; B_k is obtained by *linking* two disjoint copies of B_{k-1}, i.e., making the root with the smaller key the parent of the other. (Some B_k's can be seen in Figure 2, see below.) The *rank* of a node is a binomial tree is defined to be the number of children it has. The rank of a tree is the rank of its root. Thus, the rank of B_k is k. A *binomial heap* is composed of a collection of binomial trees, at most one of each rank. Each node, in each tree, contains an item. Each tree is *heap-ordered*: the key of an item contained in a node is not larger than the keys of the items contained in the children, if any, of that node. This means, in particular, that the item of minimum key must reside in one of the roots.

To `insert` an item into a binomial heap, create a new B_0 and place the item in it. If the heap does not contain a B_0, we are done. If the heap already contains a B_0, the two trees are linked, making the one whose root contains an item with a smaller key the parent of the other. This creates a new B_1. If the heap does not contain a B_1 we are done. Otherwise, the two B_1's are linked to form a B_2, and so on. (The process is similar to process of *incrementing* a binary counter.)

To find the item with minimum key, compare the keys of the items in the roots. Optionally, maintain a pointer to the root holding the smallest item.

Standard binomial heaps do not support the deletion of arbitrary items, only of items that reside in roots, e.g., an item of minimum key. Suppose that an item to be deleted resides in the root of a B_k. It is not difficult to check that removing the root of a B_k creates a disjoint collection of $B_0, B_1, \ldots, B_{k-1}$. The trees in this collection are added to the collection of trees forming the heap. As long as the collection contains two trees of the same rank, a link operation is performed. (The process is similar to the process of *adding* two binary numbers.)

The above process can also be used to support a *meld* operation that we do not consider here.

It is not difficult to check that a binomial heap containing n items is composed of at most $\lg n$ trees (more precisely, the number of 1's in the binary representation of n), and that each of the operations described above requires at most $\lg n$ comparisons and $O(\log n)$ time.

In the next section we present a straightforward *lazy* version of binomial heaps. To analyze this lazy version we need bounds on the number of nodes of each different rank contained in a binomial trees and heaps.

LEMMA 4.1. *A binomial heap holding n items contains at most $\lfloor n/2^i \rfloor$ nodes of rank i, for $i = 0, 1, \ldots, \lfloor \lg n \rfloor$.*

Proof. Each node of rank i is the root of a subtree of size 2^i. Trees rooted at distinct nodes of rank i are distinct.

4.2 Lazy binomial heaps A lazy binomial heap is composed of a binomial heap $\mathcal{H}$ and a list of items $\mathcal{L}$. Each node in $\mathcal{H}$ may be *marked* as deleted. Apart from that, $\mathcal{H}$ has the structure of a binary heap, i.e., each tree in $\mathcal{H}$ is a binomial tree, no two trees in $\mathcal{H}$ have the same rank, and each tree is heap-ordered. We let n be the number of items in the heap, n_0 be the number of items in $\mathcal{L}$ and n_1 be the number of (unmarked) items in $\mathcal{H}$. Thus $n = n_0 + n_1$. We let $N \geq n_1$ be the number of nodes in $\mathcal{H}$, including those marked for deletion.

To `insert` an item into a lazy binomial heap, add the item to $\mathcal{L}$. To `delete` an item, check whether the item is held in $\mathcal{L}$ or in $\mathcal{H}$. If it is held in $\mathcal{L}$, remove it from the list. If it held in $\mathcal{H}$, mark the node containing the item for deletion. Note that insert and delete operations do not perform any comparisons.

A `find-min` operation is implemented as follows. If $N \geq 2n$, rebuild a binomial heap containing the n items and find the item with minimum key. If $N < 2n$, recursively remove each root of $\mathcal{H}$ marked for deletion. (When a root is deleted, its children become new roots, some of which may be marked or deletion.) When none of the resulting roots is marked for deletion, perform linking operations on all the remaining trees and all items in $\mathcal{L}$, which are now also viewed as trees of rank 0, until there is at most one tree of each rank. Then, compare the keys in the remaining roots and return the item with minimal key. After a `find-min` operation the list $\mathcal{L}$ is empty.

A lazy binomial heap, with $n = 26$, $n_0 = 8$ and $n_1 = 18$, is shown in Figure 2.

4.3 Amortized analysis of lazy binomial heaps The amortized analysis of lazy binomial heaps uses a standard potential function argument. We give each item in $\mathcal{L}$ and each root in $\mathcal{H}$ one unit of potential. We give each item in $\mathcal{H}$ marked for deletion additional $2t-1$ units of potential. The potential of the data structure is the sum of the potentials of all items. As `insert` and `delete` perform no comparisons, the amortized number of comparisons performed by `insert` and `delete`, which is the actual number of comparisons performed by each operation plus the change in the potential, is indeed 1 and $2t-1$, respectively.

We next analyze the amortized cost of `find-min`. If $N \geq 2n$, the heap is rebuilt. Before the rebuilding, the heap contained at least $N - n \geq n$ items marks for deletion, having a total potential of at least $(2t-1)n \geq n$, as $t \geq 1$. We thus have enough potential to view each item as the root of a tree of size 1 and give it one unit of potential. We now start to link trees having the same rank. The amortized cost of each link is 0, as a link performs one comparison but reduces the potential by 1. Thus, the amortized cost of the rebuilding process is at most 0. After rebuilding, heap contains at most $\lg n$ trees and hence an item of minimum key can be found using at most $\lg n$ comparisons. The amortized cost of the `find-min` operation, in this case, is at most $\lg n$.

We next consider the case $N < 2n$. The amortized cost of deleting a root of rank k, marked for deletion, is $k - 2t$. (Each one of the k new roots receives one unit of potential, while the $1 + (2t-1) = 2t$ units of potential held by the old root, 1 for being a root and $2t-1$ for being marked for deletion, are removed.)

Let n_k be the number of roots of rank k which are deleted by the process. As $\mathcal{H}$ has the same structure as a binomial heap containing N items, we get by Lemma 4.1, for $k \leq \ln N$, that

$$n_k \;\leq\; \left\lfloor \frac{N}{2^k} \right\rfloor \;\leq\; \frac{2n}{2^k} \;\leq \frac{n}{2^{k-1}} \;.$$

The total increase in potential as a result of deleting

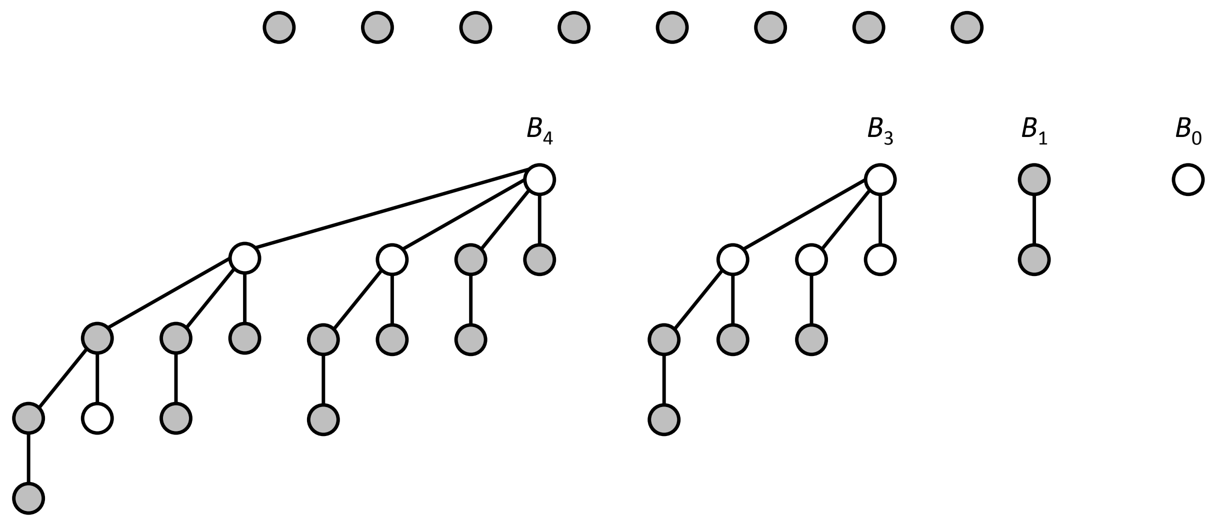

Figure 2: A lazy binomial heap. The items on top row belong to the list $\mathcal{L}$. All other items belong the the binomial heap $\mathcal{H}$. Empty nodes denote items marked for deletion.

marked roots is therefore at most

$$\sum_{k\geq 2t}(k-2t)\,n_k \;\leq\; \sum_{k\geq 2t}(k-2t)\frac{n}{2^{k-1}}$$

$$= \sum_{i\geq 0} i\frac{n}{2^{2t+i-1}} \;=\; \frac{n}{2^{2t}}\sum_{i\geq 0}\frac{i}{2^{i-1}} \;=\; \frac{4n}{2^{2t}}\;.$$

Finally, at most $\lg n$ comparisons are now performed to find the root containing the item of minimal key. These comparisons do not change the potential. This completes the proof of Theorem 4.1.

5 Lower Bound using an explicit adversary

We next describe an efficient explicit adversary using which we can obtain an $\Omega(\frac{n}{4^{A(\texttt{insert})+A(\texttt{delete})}})$ lower bound on $A(\texttt{find-min})$, the amortized number of comparisons required to perform a `find-min` operation by any deterministic comparison based priority queue data structure. The adversary used is a modification of an adversary used by Brodal *et al.* [5] to obtain an $\Omega(\frac{n}{4^{A(\texttt{insert})+A(\texttt{delete})}})$ *worst case* lower bound on the number of comparisons required to perform a `find-any` operation, and in particular a `find-min` operation. (A `find-any` operation is required to return an item x contained in the data strucure along with its *rank*, i.e., the number of items in the data structure not larger than x.)

An adversary has two tasks: *(i)* it chooses the sequence of operations that the data structure should perform; *(ii)* it answers the comparison queries made by the data structure.

5.1 The Adversary The adversary places the items inserted into the priority queue in nodes of a potentially infinite complete binary tree T. A node of T may contain several items. A node of T is *occupied* if it contains at least one item, and *empty*, otherwise. As the number of items is finite, only a finite portion of T is used. We let $\bar{T}$ be the subtree of T composed of all the ancestors of occupied nodes of T. We frequently order the nodes of $\bar{T}$ according to the *in-order* traversal of $\bar{T}$. (The *in-order* of a binary tree T, composed of a root r, a left subtree T_L and a right subtree T_R, is defined recursively as the in-order of the nodes of T_L, followed by r, followed by the in-order of the nodes of T_R.)

We let $v(x)$ denote the vertex T containing item x. For a node $u \in T$ we let *left*(u) and *right*(u) denote the left and right children of u, respectively, and we let $T(u)$ denote the set of descendants of u in T, including u itself.

When an item is inserted into the priority queue, the adversary puts it at the root of the tree. The adversary maintains a partial order $<$ on the items currently in the priority queue. All answers already given by the adversary are consistent with this partial order. The partial order is defined in the following way:

Definition 5.1. (Partial order) *Let x, y be items residing in nodes $v(x), v(y)$ of $\bar{T}$. We say that $x < y$ if and only $v(x)$ is not an ancestor of $v(y)$, $v(y)$ is not an ancestor of $v(x)$, and $v(x)$ appears before $v(y)$ in the in-order traversal of $\bar{T}$.*

Note that x and y are *comparable* in $<$ if and only if none of $v(x)$ and $v(y)$ is an ancestor of the other. If

$v(x)$ is an ancestor of $v(y)$, or vice versa, then x and y are *incomparable* in $<$. In particular, if $v(x) = v(y)$, i.e., x and y reside in the same node, then x and y are incomparable. It is not difficult to check that $<$ is indeed a partial order, i.e., if $x < y$ and $y < z$ then $x < z$.

The adversary responds to a comparison between x and y as follows:

- If $x < y$ or $y < x$ in the partial order, the adversary gives the corresponding response.
- If $v(x) = v(y)$, the adversary moves x to $\mathit{left}(v(x))$, moves y to $\mathit{right}(v(x))$, and answers that $x < y$.
- If $v(x)$ is an ancestor of $v(y)$, the adversary moves x to the child u of $v(x)$ which is not an ancestor of $v(y)$ and answers $x < y$ if $u = \mathit{left}(v(x))$, and $y < x$ if $u = \mathit{right}(v(x))$.
- If $v(y)$ is an ancestor of $v(x)$, the adversary moves y to the child u of $v(y)$ which is not an ancestor of $v(x)$ and answers $x < y$ if $u = \mathit{right}(v(x))$, and $y < x$ if $u = \mathit{left}(v(x))$.

An example showing the behavior of the adversary is given in Figure 3. (The second and third comparisons there, i.e., $a : d$ and $b : c$, are somewhat inefficient. They could be replaced by the single comparison $c : d$.)

As we are interested in amortized complexity, we may assume that the data structure performs comparisons only as a response to `find-min` operations. The data structure finishes the processing of a `find-min` operation only when the partial order $<$ contains a unique minimal element x. The node $v(x)$ containing x must then be the first occupied node in the in-order traversal of $\bar{T}$. Item x must be the only item in $v(x)$. Furthermore, all proper ancestors of $v(x)$ in T must be empty.

The adversary tries to force the data structure to perform many comparisons by using the following sequences of operations.

Definition 5.2. ((n, k)-sequences) *For parameters $n \geq 1$ and $k \geq 1$, the adversary, interacting with a given data structure, issues the following sequence of operations, which we refer to as an (n, k)-sequence.*

- *Perform n* `insert` *operations.*
- *Repeat $\lceil n/k \rceil$ times: Perform a* `find-min` *operation followed by k* `delete` *operations. Each* `delete` *operation deletes an item from the first occupied node in the in-order traversal of $\bar{T}$. (If k does not divide n, then the number of items deleted after the last* `find-min` *operation is less than k.)*

Note the resemblance of an (n, k)-sequence to the canonical sequences introduced in Section 2. The difference is that the adversary used here is *adaptive*, the identity of the deleted items in an (n, k)-sequence depends on the comparisons made by the data structure. This is why the lower bound of this section is valid only for *deterministic* data structures. On the other hand, the adversary of this section uses the same strategy to force any deterministic data structure to perform many comparisons, without having to know in advance how the data structure works.

5.2 A pebbling game To facilitate the proof of the lower bound, we introduce a simple (n, k)-*pebbling game*. The game is played by the data structure. The behavior of the adversary is coded in the rules of the game. The game starts by placing n pebbles at the root of the potentially infinite complete binary tree T. (We use the same notation and terminology as above.)

The goal of the player is to eliminate all the pebbles from the tree. When the first occupied node u in T, according to in-order, contains a single pebble, and all proper ancestors of u are empty, k pebbles are sequentially removed, each from the first occupied node in the tree. We refer to the condition above as the *elimination condition*. (The identity of the first pebble removed is uniquely determined. The next pebbles may be removed from nodes that contain more than one pebble. The identity of the pebble removed in such a case is not important.)

When the elimination condition is not satisfied, the player is allowed to perform one of the following *moves*:

- Take two pebbles p and q that reside in the same vertex v of T, move p to $\mathit{left}(v)$ and q to $\mathit{right}(v)$.
- Take a pebble p placed at vertex v of T such that one of the vertices in $T(\mathit{left}(v))$ contains a pebble q, and move p to $\mathit{right}(v)$.
- Take a pebble p placed at vertex v of T such that one of the vertices in $T(\mathit{right}(v))$ contains a pebble q, and move p to $\mathit{left}(v)$.

The goal of the player is to eliminate all pebbles using minimum number of moves. The following lemma establishes the connection between the problem of designing a data structure that performs a minimal number of comparisons and solving the pebbling game using a minimal number of moves.

Lemma 5.1. *The minimum number of comparisons required by a deterministic algorithm to perform an (n, k)-sequence of priority queue operations issued by the adversary is equal to the number of moves needed to remove all pebbles in an (n, k)-pebbling game.*

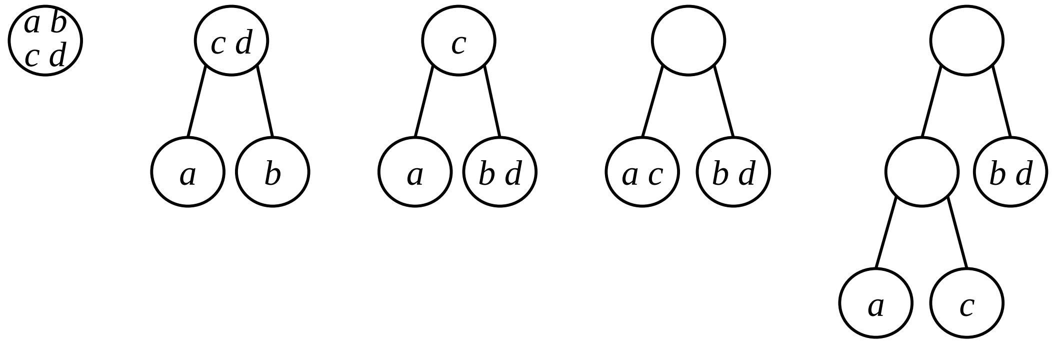

Figure 3: Various states of the tree kept by the adversary. The tree on the left is the tree after the insertion of items a, b, c and d. The following trees are the results of the comparisons $a : b$, $a : d$, $b : c$ and $a : c$. The partial order defined by the rightmost has a unique minimal item. If $k = 3$, then the 3 items removed are a, c and b or d.

Proof. A state of the adversary, i.e., the placement of the items in the binary tree T, corresponds immediately to a state in the pebbling game. The elimination condition of the pebbling game corresponds exactly to a situation in which the partial order $<$ maintained by the adversary contains a unique minimal item. It is easy to check that the movement of the items in the tree as a result of a comparison corresponds exactly to a move in pebbling game. □

Lemma 5.2. *An (n, k)-pebbling game can be solved using a minimal number of moves by first moving all pebbles from the root to the left and right children of the root, then performing moves involving only pebbles in the left subtree, and finally performing moves that only involve pebbles in the right subtree.*

Proof. If the root is occupied and $n > 1$, then the elimination condition does not hold. Thus, all pebbles must eventually leave the root. All moves that do not involve pebbles residing in the root may be delayed until all pebbles leave the root. Similarly, as long as the left subtree is not empty and the elimination condition is not satisfied, moves involving tokens from the right subtree may be delayed. It is not difficult to check that each delayed move may be performed later, and that each move that was moved forward is still a valid move. □

5.3 Lower bound Let $T_k(n)$ be the minimal number of moves required to solve the (n, k)-pebbling game.

Lemma 5.3. *(Recurrence for $T_k(n)$) For any $n, k \geq 1$*

$$T_k(n) \;\geq\; \left\lceil \frac{n}{2} \right\rceil + \min_{1\leq \ell < n} \{\, T_k(\ell) + T_k(n-\ell-k) \,\} \;.$$

Proof. By Lemma 5.2 we may assume that all pebbles are moved out of the root before any other moves are performed and that moves involving pebbles in the right subtree are performed only after the left subtree is cleared. This decomposes the problem into two almost disjoint subproblems.

Removing all pebbles from the root requires at least $\lceil n/2 \rceil$ moves, as each move removes at most two pebbles from the root. Let ℓ be the number of pebbles moved to the left child of the root by an optimal move sequence. Removing all the pebbles from left subtree is exactly an (ℓ, k)-pebbling game which, by definition, requires $T_k(\ell)$ moves. If ℓ is not divisible by k, then the last elimination round of the (ℓ, k)-pebbling game allows us to remove some pebbles from the right subtree 'for free'. However, the number of pebbles thus removed is at most k. Thus, we are left with an (r, k)-pebbling game to be played on the right subtree, where $r \geq n - \ell - k$. It is easy to see that $T_k(r) \geq T_k(n - \ell - k)$. □

Lemma 5.4. *(Lower bound for $T_k(n)$) For all $n, k \geq 1$,*

$$T_k(n) \;\geq\; \frac{n+k}{2} \lg \frac{n+k}{4k} + \frac{k}{2} \;.$$

Proof. By induction on n. For $n \leq k$ we have

$$\frac{n+k}{2} \lg \frac{n+k}{4k} + \frac{k}{2} \;\leq\; 0 \;\leq\; T_k(n) \;.$$

Assume that the claim holds all $n' < n$ we prove it for n. By Lemma 5.3, the induction hypothesis and the convexity of the function $x \lg \frac{x}{4k}$, we have

$$T_k(n) \;\geq\; \frac{n}{2} + \min_{1\leq \ell < n} \{\, T_k(\ell) + T_k(n-\ell-k) \,\}$$

$$\geq\; \frac{n}{2} + \min_{1\leq \ell < n} \left\{ \frac{k{+}\ell}{2} \lg \frac{k{+}\ell}{4k} + \frac{k}{2} + \frac{n{-}\ell}{2} \lg \frac{n{-}\ell}{4k} + \frac{k}{2} \right\}$$

$$\geq\; \frac{n}{2} + k + \frac{n+k}{2} \lg \frac{n+k}{8k} \;=\; \frac{n+k}{2} \lg \frac{n+k}{4k} + \frac{k}{2} \;.$$

□

In particular, we get that $T_1(n) \geq \frac{n+1}{2} \lg \frac{n+1}{4}$. As an $(n,1)$-pebbling game is equivalent to sorting, we get as a corollary, that the explicit adversary of this section forces any deterministic sorting algorithm to perform at least $\frac{n+1}{2} \lg \frac{n+1}{4}$ comparisons. (A similar result is contained in Brodal et al. [5].)

Theorem 5.1. *For any deterministic priority queue and every $k \leq n$ we have:*

$$A(\texttt{find-min}) \;\geq\; \frac{k}{2} \lg \frac{n}{4k} - k\left(A\left(\texttt{insert}\right) + A\left(\texttt{delete}\right)\right).$$

Proof. By Lemma 5.1, any algorithm must make at least $T_k(n)$ comparisons when executing the sequence of operations defined above, consisting of n `insert`, n `delete` and $\lceil \frac{n}{k} \rceil$ `find-min` operations. Using Lemma 5.4 we get that

$$\begin{aligned} A\left(\texttt{find-min}\right) \;&\geq\; \frac{T_k\left(n\right) - n(A\left(\texttt{insert}\right) + A\left(\texttt{delete}\right))}{\lceil \frac{n}{k} \rceil} \\ \geq\; &\frac{k}{n+k}\left(\frac{n+k}{2} \lg \frac{n+k}{4k} - n(A\left(\texttt{insert}\right) - A\left(\texttt{delete}\right)\right) \\ &\geq\; \frac{k}{2} \lg \frac{n}{4k} - k(A(\texttt{insert}) + A(\texttt{delete}))\,. \end{aligned}$$

□

Theorem 5.2. *For any deterministic priority queue,*

$$A\left(\texttt{find-min}\right) \;\geq\; \frac{1}{20} \cdot \frac{n}{4^{A(\texttt{insert})+A(\texttt{delete})}}$$

Proof. Let $A(\texttt{insert}) + A(\texttt{delete}) = 2t$. By Lemma 5.1 we have

$$A(\texttt{find-min}) \;\geq\; \frac{k}{2} \lg \frac{n}{4k} - 2kt$$

This expression is maximized when $k = \frac{n}{4\mathrm{e}4^{2t}}$ and the obtained lower bound is

$$A(\texttt{find-min}) \;\geq\; \frac{\lg \mathrm{e}}{8\mathrm{e}} \frac{n}{4^{2t}} \;>\; \frac{1}{20} \frac{n}{4^{2t}}\,.$$

□

6 Deleting a non-minimum item

As we already mentioned, in any implementation of a priority queue either `delete-min` or `insert` must perform $\Omega(\log n)$ comparisons. What about the complexity of a `delete` operation that deletes an item which is *not* of minimum key in the priority queue?

Let `delete-non-min`(x) denote the operation of deleting item x from the priority queue, given that x is *not* an item with minimum key in the priority queue. Is it possible to design a priority queue in which `find-min`, `insert` and `delete-non-min` take $o(\log n)$ amortized time, while `delete-min` takes $O(\log n)$ amortized time?

Relying on the lower bound of Section 3 (or Section 5) we give a simple reduction that shows that this is not possible.

Given a data structure $\mathcal{B}$ that supports `insert`, `delete-non-min`, `delete-min` and `find-min` operations, we can construct a data structure $\mathcal{B}'$ that supports `insert`, `delete` and `find-min` operations as follows: $\mathcal{B}'$ keeps the items in the structure $\mathcal{B}$. In addition, it inserts into $\mathcal{B}$ an item whose key is smaller than all other keys. We denote this item by $-\infty$. Operations on $\mathcal{B}'$ are performed as follows:

- `insert`(x): Insert x into $\mathcal{B}$.
- `delete`(x): Delete x from $\mathcal{B}$ using `delete-non-min`, as x is not the minimum item in $\mathcal{B}$.
- `find-min`: Delete $-\infty$ from $\mathcal{B}$ using `delete-min`, perform `find-min` on $\mathcal{B}$ and return the result. Finally re-insert $-\infty$ to $\mathcal{B}$ using `insert`.

We clearly have:

$$\begin{aligned} A(\texttt{insert}_{\mathcal{B}'}) \;&=\; A(\texttt{insert}_{\mathcal{B}}) \\ A(\texttt{delete}_{\mathcal{B}'}) \;&=\; A(\texttt{delete-non-min}_{\mathcal{B}}) \end{aligned}$$

$$A(\texttt{find-min}_{\mathcal{B}'}) \;=\; A(\texttt{delete-min}_{\mathcal{B}}) + A(\texttt{find-min}_{\mathcal{B}}) + A(\texttt{insert}_{\mathcal{B}})$$

As an immediate corollary of Theorem 3.1, we get:

Corollary 6.1. *For any priority queue data structure, if $A(\texttt{insert}) = A(\texttt{delete-non-min}) = t$, then $A(\texttt{find-min}) + A(\texttt{delete-min}) + A(\texttt{insert}) = \Omega(\frac{n}{t2^{2t}})$.*

In particular, if $A(\texttt{insert}) = o(\log n)$ and $A(\texttt{delete-non-min}) = o(\log n)$, then $A(\texttt{find-min}) + A(\texttt{delete-min}) + A(\texttt{insert}) = \Omega(n^{1-\epsilon})$, for any $\epsilon > 0$.

For completeness, we note that there is also an easy reduction in the opposite direction. Given a data structure $\mathcal{B}$ supporting `find-min`, `insert`, and `delete` we can construct a data structure $\mathcal{B}'$ supporting `find-min`, `insert`, `delete-non-min`, and `delete-min`. The structure $\mathcal{B}'$ keeps the items in the structure $\mathcal{B}$ and it also maintains a pointer called *min* to the minimum item in $\mathcal{B}$.

- `find-min`: Return the item stored in *min*.
- `insert`(x): Insert x into $\mathcal{B}$ and if $x < min$ update *min* to point to x.
- `delete-non-min`(x): Delete x from $\mathcal{B}$ (as x is not the minimum, *min* remains correct).

- `delete-min`: Delete the item saved in *min*. Perform `find-min` on $\mathcal{B}$ and update *min* to point to the resulting item.

Using Theorem 4.1 we thus get a data structure that supports `find-min` with no comparisons, `insert` using an amortized number of 2 comparisons, `delete-non-min` in an amortized number of $2t - 1$ comparisons, and `delete-min` using an amortized number of $O(\frac{n}{2^{2t}} + \log n)$ comparisons.

7 Concluding remarks and open problems

We obtained almost matching lower and upper bounds, $\Omega(n/(t2^{2t}))$ and $O(n/2^{2t} + \log n)$, respectively, on the (expected) amortized number of comparisons performed by `find-min` operations, if the amortized number of comparisons performed by `insert` and `delete` are both t. Closing the small gap between these two bounds is an interesting open problem.

We also presented an explicit adversary using which a weaker lower bound of $\Omega(n/(4^{2t}))$ on the amortized cost of `find-min` may be obtained. Is there an explicit adversary using which an $\Omega(n/(4^{2t}))$ lower bound on the amortized cost of `find-min` may be obtained? This seems related to perhaps an even more basic open problem: Is there explicit adversary that forces any deterministic sorting algorithm to perform $(1 - o(1))n \lg n$ comparisons? Efficient explicit adversaries that force every comparison based sorting algorithm to perform $(\frac{1}{2} - o(1))n \lg n$ comparisons were obtained by Atallah and Kosaraju [3] and by Brodal et al. [5].

Acknowledgement

We would like to thank Bob Tarjan for helpful discussions and Mikkel Thorup for bringing the Brodal et al. [5] paper to our attention.

References

[1] G.M. Adel'son-Vel'skiĭ and E.M. Landis. An algorithm for organization of information. *Dokl. Akad. Nauk SSSR*, 146:263–266, 1962.

[2] S. Alstrup, T. Husfeldt, T. Rauhe, and M. Thorup. Black box for constant-time insertion in priority queues (note). *ACM Transactions on Algorithms*, 1(1):102–106, 2005.

[3] M.J. Atallah and S.R. Kosaraju. An adversary-based lower bound for sorting. *Information Processing Letters*, 13(2):55–57, 1981.

[4] A. Borodin, L.J. Guibas, N.A. Lynch, and A.C.C. Yao. Efficient searching using partial ordering. *Information Processing Letters*, 12(2):71–75, 1981.

[5] G.B. Brodal, S. Chaudhuri, and J. Radhakrishnan. The randomized complexity of maintaining the minimum. *Nord. J. Comput.*, 3(4):337–351, 1996.

[6] G.S. Brodal. Fast meldable priority queues. In *Proc. of 4th WADS*, pages 282–290, 1995.

[7] G.S. Brodal. Worst-case efficient priority queues. In *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 52–58, 1996.

[8] G.S. Brodal, G. Lagogiannis, and R.E. Tarjan. Strict Fibonacci heaps. In *Proceedings of the 44th ACM Symposium on Theory of Computing (STOC)*, pages 1177–1184, 2012.

[9] J.R. Driscoll, H.N. Gabow, R. Shrairman, and R.E. Tarjan. Relaxed heaps: an alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.

[10] S. Edelkamp, A. Elmasry, and J. Katajainen. Weak heaps engineered. *J. Discrete Algorithms*, 23:83–97, 2013.

[11] S. Edelkamp, J. Katajainen, and A. Elmasry. Strengthened lazy heaps: Surpassing the lower bounds for binary heaps. *CoRR*, abs/1407.3377, 2014.

[12] A. Elmasry, C. Jensen, and J. Katajainen. Multipartite priority queues. *ACM Transactions on Algorithms*, 5(1), 2008.

[13] R. Fleischer. A simple balanced search tree with $O(1)$ worst-case update time. *Int. J. Found. Comput. Sci.*, 7(2):137–150, 1996.

[14] L.R. Ford and S.M. Johnson. A tournament problem. *The American Mathematical Monthly*, 66(5):387 – 389, 1959.

[15] M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proc. of 21st STOC*, pages 345–354, 1989.

[16] L.J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. pages 8–21, 1978.

[17] H. Kaplan, N. Shafrir, and R.E. Tarjan. Meldable heaps and boolean union-find. In *Proceedings of the 34th ACM Symposium on Theory of Computing (STOC)*, pages 573–582, 2002.

[18] C. Levcopoulos and M.H. Overmars. A balanced search tree with $O(1)$ worst-case update time. *Acta Inf.*, 26(3):269–277, 1988.

[19] Colin McDiarmid. Average-case lower bounds for searching. *SIAM Journal on Computing*, 17(5):1044–1060, 1988.

[20] R.E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

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

[22] J.W.J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7:347–348, 1964.

[23] A.C.C Yao. Probabilistic computations: Toward a unified measure of complexity (extended abstract). In *Proc. of 18th FOCS*, pages 222–227, 1977.