# The Randomized Complexity of Maintaining the Minimum

Gerth Stølting Brodal* Shiva Chaudhuri† 
Jaikumar Radhakrishnan‡

May 14, 1996

**Abstract**

The complexity of maintaining a set under the operations Insert, Delete and FindMin is considered. In the comparison model it is shown that any randomized algorithm with expected amortized cost $t$ comparisons per Insert and Delete has expected cost at least $n/(e2^{2t}) - 1$ comparisons for FindMin. If FindMin is replaced by a weaker operation, FindAny, then it is shown that a randomized algorithm with constant expected cost per operation exists, but no deterministic algorithm. Finally, a deterministic algorithm with constant amortized cost per operation for an offline version of the problem is given.

## 1 Introduction

We consider the complexity of maintaining a set $S$ of elements from a totally ordered universe under the following operations:

Insert($e$): inserts the element $e$ into $S$,

Delete($e$): removes from $S$ the element $e$ provided it is known where $e$ is stored, and

FindMin: returns the minimum element in $S$ without removing it.

We refer to this problem as the Insert-Delete-FindMin problem. We denote the size of $S$ by $n$. The analysis is done in the comparison model, i.e. the time required by an operation is the number of comparisons it makes. The input is a sequence of operations, given to the algorithm in an on-line manner, that is, the algorithm must process the current operation before it receives the next operation in the sequence. The *worst case* time for an operation is the maximum, over all such operations in all sequences, of the time taken to process the operation. The *amortized* time of an operation is the maximum, over all sequences, of the total number of comparisons performed, while processing this type of operation in the sequence, divided by the length of the sequence.

Worst case asymptotic time bounds for some existing data structures supporting the above operations are listed in Table 1. The table suggests a trade-off between

*BRICS (Basic Research in Computer Science, a Centre of the Danish National Research Foundation), Computer Science Department, Aarhus University, Ny Munkegade, DK-8000 Århus C, Denmark. Supported by the Danish Natural Science Research Council (Grant No. 9400044). This research was done while visiting the Max-Planck Institut für Informatik, Saabrücken, Germany. Email: `gerth@daimi.aau.dk`.

†Max–Planck–Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany. This work was partially supported by the EU ESPRIT LTR project No. 20244 (ALCOM IT). Email: `shiva@mpi-sb.mpg.de`.

‡Tata Institute of Fundamental Research, Mumbai, India. Email: `jaikumar@tcs.tifr.res.in`.

| Implementation | Insert | Delete | FindMin |
|---|---|---|---|
| Doubly linked list | 1 | 1 | $n$ |
| Heap [8] | $\log n$ | $\log n$ | 1 |
| Search tree [5, 7] | $\log n$ | 1 | 1 |
| Priority queue [2, 3, 4] | 1 | $\log n$ | 1 |

Figure 1: Worst case asymptotic time bounds for different set implementations.

the worst case times of the two update operations Insert, Delete and the query operation FindMin. We prove the following lower bound on this tradeoff: any randomized algorithm with expected amortized update time at most $t$ requires expected time $(n/e2^t) - 1$ for FindMin. Thus, if the update operations have expected amortized constant cost, FindMin requires linear expected time. On the other hand if FindMin has constant expected time, then one of the update operations requires logarithmic expected amortized time. This shows that all the data structures in Figure 1 are optimal in the sense of the trade-off, and they cannot be improved even by considering amortized cost and allowing randomization.

For each $n$ and $t$, the lower bound is tight. A simple data structure for the Insert-Delete-FindMin problem is the following. Assume Insert and Delete are allowed to make at most $t$ comparisons. We represent a set by $\lceil n/2^t \rceil$ sorted lists. All lists except for the last contain exactly $2^t$ elements. The minimum of a set can be found among all the list minima by $\lceil n/2^t \rceil - 1$ comparisons. New elements are added to the last list, requiring at most $t$ comparisons by a binary search. To perform Delete we replace the element to be deleted by an arbitrary element from the last list. This also requires at most $t$ comparisons.

The above lower bound shows that it is hard to maintain the minimum. Is it any easier to maintain the rank of *some* element, not necessarily the minimum? We consider a weaker problem called Insert-Delete-FindAny, which is defined exactly as the previous problem, except that FindMin is replaced by the weaker operation FindAny:

FindAny: returns some element in $S$ and its rank.

FindAny is not constrained to return the same element each time it is invoked or to return the element with the same rank. The only condition is that the rank returned should be the rank of the element returned. We give a randomized algorithm for the Insert-Delete-FindAny problem with constant expected time per operation. Thus, this problem is strictly easier than Insert-Delete-FindMin, when randomization is allowed. However, we show that for deterministic algorithms, the two problems are essentially equally hard. We show that any deterministic algorithm with amortized update time at most $t$ requires $n/2^{4t+3}-1$ comparisons for some FindAny operation. This lower bound is proved using an explicit adversary argument. The adversary strategy is simple, yet surprisingly powerful. The same strategy may be used to obtain the well known $\Omega(n \log n)$ lower bound for sorting. An explicit adversary for sorting has previously been given by Atallah and Kosaraju [1].

The previous results show that maintaining any kind of rank information online is hard. However, if the sequence of instructions to be processed is known in advance, then one can do better. We give a deterministic algorithm for the offline Insert-Delete-FindMin problem which has an amortized cost per operation of at most 3 comparisons.

Our proofs use various averaging arguments which are used to derive general combinatorial properties of trees. These are presented in Section 2.2.

# 2 Preliminaries

## 2.1 Definitions and notation

For a rooted tree $T$, let $\mathsf{leaves}(T)$ be the set of leaves of $T$. For a vertex, $v$ in $T$, define $\mathsf{deg}(v)$ to be the number of children of $v$. Define, for $l \in \mathsf{leaves}(T)$, $\mathsf{depth}(l)$ to be the distance of $l$ from the root and $\mathsf{path}(l)$ to be the set of vertices on the path from the root to $l$, not including $l$.

For a random variable $X$, let support$[X]$ be the set of values that $X$ assumes with non-zero probability. For any non-negative real-valued function $f$, defined on support$[X]$, define

$$\mathop{\mathbf{E}}_X[f(X)] = \sum_{x \in \text{support}[X]} \Pr[X=x] f(x), \text{ and } \mathop{\mathbf{GM}}_X[f(X)] = \prod_{x \in \text{support}[X]} f(x)^{\Pr[X=x]}.$$

We will also use the notation $\mathbf{E}$ and $\mathbf{GM}$ to denote the arithmetic and geometric means of a set of values as follows: for a set $R$, and any non-negative real-valued function $f$, defined on $R$, define

$$\mathop{\mathbf{E}}_{r \in R}[f(r)] = \frac{1}{|R|}\sum_{r \in R} f(r), \text{ and } \mathop{\mathbf{GM}}_{r \in R}[f(r)] = \prod_{r \in R} f(x)^{1/|R|}.$$

## 2.2 Some useful lemmas

Let $T$ be the infinite complete binary tree. Suppose each element of $[n] = \{1, \ldots, n\}$ is assigned to a node of the tree (more than one element may be assigned to the same node). That is, we have a function $f : [n] \to V(T)$. For $v \in V(T)$, define $\mathsf{wt}_f(v) = |\{i \in [n] : f(i) = v\}|$, $d_f = \mathbf{E}_{i \in [n]}[\mathsf{depth}(f(i))]$, $D_f = \max\{\mathsf{depth}(f(i)) : i \in [n]\}$ and $m_f = \max\{\mathsf{wt}_f(v) : v \in V(T)\}$.

**Lemma 1** *For every assignment $f : [n] \to V(T)$, the maximum number of elements on a path starting at the root of $T$ is at least $n2^{-d_f}$.*

*Proof.* Let $P$ be a random infinite path starting from the root. Then, for $i \in [n]$, $\Pr[f(i) \in P] = 2^{-\mathsf{depth}(f(i))}$. Then the expected number of elements of $[n]$ assigned to $P$ is

$$\begin{aligned}\sum_{i=1}^{n} 2^{-\mathsf{depth}(f(i))} &= n \mathop{\mathbf{E}}_{i \in [n]}[2^{-\mathsf{depth}(f(i))}] \geq n \mathop{\mathbf{GM}}_{i \in [n]}[2^{-\mathsf{depth}(f(i))}] \\ &= n 2^{-\mathbf{E}_{i \in [n]}[\mathsf{depth}(f(i))]} = n2^{-d_f}.\end{aligned}$$

Since the maximum is at least the expected value, the lemma follows. ∎

**Lemma 2** *For every assignment $f : [n] \to V(T)$, $m_f \geq n/(2^{d_f+3})$.*

*Proof.* Let $H = \{h : m_h = m_f\}$. Let $h$ be the assignment in $H$ with minimum average depth $d_h$ (the minimum exists). Let $m = m_h = m_f$, and $D = D_h$. We claim that

$$\mathsf{wt}_h(v) = m, \text{ for each } v \in V(T) \text{ with } \mathsf{depth}(v) < D. \tag{1}$$

For suppose there is a vertex $v$ with $\mathsf{depth}(v) < D$ and $\mathsf{wt}(v) < m$ (i.e. $\mathsf{wt}(v) \leq m-1$). First, consider the case when some node $w$ at depth $D$ has $m$ elements assigned to it. Consider the assignment $h'$ given by

$$h'(i) \stackrel{\text{def}}{=} \begin{cases} w & \text{if } h(i) = v, \\ v & \text{if } h(i) = w, \\ h(i) & \text{otherwise.} \end{cases}$$

Then $h' \in H$ and $d_{h'} < d_h$, contradicting the choice of $h$. Next, suppose that every node at depth $D$ has less than $m$ elements assigned to it. Now, there exists $i \in [n]$ such that $\mathsf{depth}(h(i)) = D$. Let $h'$ be the assignment that is identical to $h$ everywhere except at $i$, and for $i$, $h'(i) = v$. Then, $h' \in H$ and $d_{h'} < d_h$, again contradicting the choice of $h$. Thus (1) holds.

The number of elements assigned to nodes at depth at most $D-1$ is $m(2^D - 1)$, and the average depth of these elements is

$$\frac{1}{m(2^D-1)} \sum_{i=0}^{D-1} mi2^i = \frac{(D-2)2^D+2}{2^D-1} \geq D-2.$$

Since all other elements are at depth $D$, we have $d_h \geq D-2$. The total number of nodes in the tree with depth at most $D$ is $2^{D+1}-1$. Hence, we have

$$m_f = m \geq \frac{n}{2^{D+1}-1} \geq \frac{n}{2^{d_h+3}-1} \geq \frac{n}{2^{d_f+3}-1}.$$

∎

For a rooted tree $T$, let $W_l = \prod_{v \in \mathsf{path}(l)} \mathsf{deg}(v)$. Then, it can be shown by induction on the height of tree that $\sum_{l \in \mathsf{leaves}(T)} 1/W_l = 1$.

**Lemma 3** *For a rooted tree $T$ with $m$ leaves,* $\underset{l \in \mathsf{leaves}(T)}{\mathbf{GM}} [W_l] \geq m$.

*Proof.* Since the geometric mean is at most the arithmetic mean [6], we have

$$\underset{l}{\mathbf{GM}}[\frac{1}{W_l}] \;\leq\; \underset{l}{\mathbf{E}}[\frac{1}{W_l}] = \frac{1}{m}\sum_l \frac{1}{W_l} \;=\; \frac{1}{m}.$$

Now,

$$\underset{l}{\mathbf{GM}}[W_l] \;=\; \frac{1}{\underset{l}{\mathbf{GM}}[1/W_l]} \;\geq\; m.$$

∎

# 3 Deterministic offline algorithm

We now consider an offline version of the $\mathsf{Insert\text{-}Delete\text{-}FindMin}$ problem. The sequence of operations to be performed is given in advance, however, the ordering of the set elements is unknown. The $i$th operation is performed at time $i$. We assume that an element is inserted and deleted at most once. If an element is inserted and deleted more than once, it can be treated as a distinct element each time it is inserted.

From the given operation sequence, the offline algorithm can compute, for each element $e$, the time, $t(e)$, at which $e$ is deleted from the data structure ($t(e)$ is $\infty$ if $e$ is never deleted).

The data structure maintained by the offline algorithm is a sorted (in increasing order) list $L = (e_1, \ldots, e_k)$ of the set elements that can become minimum elements in the data structure. The list satisfies that $t(e_i) < t(e_j)$ for $i < j$, because otherwise $e_j$ could never become a minimum element.

$\mathsf{FindMin}$ returns the first element in $L$ and $\mathsf{Delete}(e)$ deletes $e$ from $L$, if $L$ contains $e$. To process $\mathsf{Insert}(e)$, the algorithm computes two values, $l$ and $r$, where $r = \min\{i : t(e_i) > t(e)\}$ and $l = \max\{i : e_i < e\}$. Notice that once $e$ is in the data structure, none of $e_{l+1}, \ldots, e_{r-1}$ can ever be the minimum element. Hence, all these elements are deleted and $e$ is inserted into the list between $e_l$ and $e_r$. No comparisons are required to find $r$. Thus, $\mathsf{Insert}(e)$ may be implemented as follows: starting at $e_r$, step backwards through the list, deleting elements until the first element smaller than $e$ is encountered.

The number of comparisons for an insertion is two plus the number of elements deleted from $L$. By letting the potential of $L$ be $|L|$ the amortized cost of Insert is $|L'|-|L|+$ # of element removed during the Insert $+2$ which is at most 3 because the number of elements removed is at most $|L|-|L'|+1$. Delete only decreases the potential, and the initial potential is zero. It follows that

**Theorem 4** *For the offline* Insert-Delete-FindMin *problem the amortized cost of* Insert *is three comparisons. No comparisons are required for* Delete *and* FindMin*.*

# 4 Deterministic lower bound for FindAny

In this section we show that it is difficult for a deterministic algorithm to maintain any rank information at all. We prove

**Theorem 5** *Let $\mathcal{A}$ be a deterministic algorithm for the* Insert-Delete-FindAny *problem with amortized time at most $t = t(n)$ per update. Then, there exists an input, to process which $\mathcal{A}$ takes at least $n/2^{4t+3} - 1$ comparisons for one* FindAny*.*

**The Adversary.** We describe an adversary strategy for answering comparisons between a set of elements.

The adversary maintains an infinite binary tree and the elements currently in the data structure are distributed among the nodes of this tree. New elements inserted into the data structure are placed at the root. For $x \in S$ let $v(x)$ denote the node of the tree at which $x$ is. The adversary maintains the following invariants (A) and (B). For any distribution of the elements among the nodes of the infinite tree, define the *occupancy tree* to be the finite tree given by the union of the paths from every non-empty node to the root. The invariants are

(A) If neither of $v(x)$ or $v(y)$ is a descendant of the other then $x < y$ is consistent with the responses given so far if $v(x)$ appears before $v(y)$ in an preorder traversal of the occupancy tree, and

(B) If $v(x) = v(y)$ or $v(x)$ is a descendant of $v(y)$, the responses given so far yield no information on the order of $x$ and $y$. More precisely, in this case, $x$ and $y$ are incomparable in the partial order induced on the elements by the responses so far.

The comparisons made by any algorithm can be classified into three types, and the adversary responds to each type of the comparison as described below. Let the elements compared be $x$ and $y$.

- $v(x) = v(y)$: Then $x$ is moved to the left child of $v(x)$ and $y$ to the right child and the adversary answers $x < y$.
- $v(x)$ is a descendant of $v(y)$: Then $y$ is moved to the unique child of $v(y)$ that is not an ancestor of $v(x)$. If this child is a left child then the adversary answers $y < x$ and if it is a right child then the adversary answers $x < y$.
- $v(x) \neq v(y)$ and neither is a descendant of the other: If $v(x)$ is visited before $v(y)$ in a preorder traversal of the occupancy tree, the adversary answers $x < y$ and otherwise the adversary answers $y < x$.

The key observation is that each comparison pushes two elements down one level each, in the worst case.

**Maintaining ranks.** We now give a proof of Theorem 5.

Consider the behavior of the algorithm when responses to its comparisons are given according to the adversary strategy above. Define the sequences $S_1 \ldots S_{n+1}$ as follows. $S_1 = \mathsf{Insert}(a_1) \ldots \mathsf{Insert}(a_n)\mathsf{FindAny}$. Let $b_1$ be the element returned in response to the FindAny instruction in $S_1$. For $i = 2, 3, \ldots n$, define

$$S_i = \mathsf{Insert}(a_1) \ldots \mathsf{Insert}(a_n)\mathsf{Delete}(b_1) \ldots \mathsf{Delete}(b_{i-1})\mathsf{FindAny}$$

and let $b_i$ be the element returned in response to the FindAny instruction in $S_i$. Finally, let

$$S_{n+1} = \mathsf{Insert}(a_1) \ldots \mathsf{Insert}(a_n)\mathsf{Delete}(b_1) \ldots \mathsf{Delete}(b_n).$$

For $1 \leq i \leq n$, $b_i$ is well defined and for $1 \leq i < j \leq n$, $b_i \neq b_j$. The latter point follows from the fact that at the time $b_i$ is returned by a FindAny, $b_1, \ldots, b_{i-1}$ have already been deleted from the data structure.

Let $T$ be the infinite binary tree maintained by the adversary. Then the sequence $S_{n+1}$ defines a function $f : [n] \to V(T)$, given by $f(i) = v$ if $b_i$ is in node $v$ just before the $\mathsf{Delete}(b_i)$ instruction during the processing of $S_{n+1}$. Since the amortized cost of an update is at most $t$, the total number of comparisons performed while processing $S_{n+1}$ is at most $2tn$. A comparison pushes at most two elements down one level each. Then, writing $d_i$ for the distance of $f(i)$ from the root, we have $\sum_{i=1}^{n} d_i \leq 4tn$. By Lemma 2 we know that there is a set $R \subseteq [n]$ with at least $n/2^{4t+3}$ elements and a vertex $v$ of $T$ such that for each $i \in R$, $f(b_i) = v$.

Let $j = \min R$. Then, while processing $S_j$, just before the FindAny instruction, each element $b_i$, $i \in R$ is in some node on the path from the root to $f(i) = v$. Since the element returned by the FindAny is $b_j$, it must be the case that after the comparisons for the FindAny are performed, $b_j$ is the only element on the path from the root to the vertex in which $b_j$ is. This is because invariant (B) implies that any other element that is on this path is incomparable with $b_j$. Hence, these comparisons move all the elements $b_i$, $i \in R \backslash j$, out of the path from the root to $f(j)$. A comparison can move at most one element out of this path, hence, the number of comparisons performed is at least $|R| - 1$, which proves the theorem.

## 4.1 Sorting

The same adversary can be used to give a lower bound for sorting. We note that this argument is fundamentally different from the usual information theoretic argument in that it gives an explicit adversary against which sorting is hard.

Consider an algorithm that sorts a set $S$, of $n$ elements. The same adversary strategy is used to respond to comparisons. Then, invariant (B) implies that at the end of the algorithm, each element in the tree must be in a node by itself. Let the function $f : S \to V(T)$ indicate the node where each element is at the end of the algorithm, where $T$ is the infinite binary tree maintained by the adversary. Then, $f$ assigns at most one element to each path starting at the root of $T$. By Lemma 1 we have $1 \geq n2^{-d}$, where d is average distance of an element from the root. It follows that the sum of the distances from the root to the elements in this tree is at least $n \log n$, and this is equal to the sum of the number of levels each element has been pushed down. Since each comparison contributes at most two to this sum, the number of comparisons made is at least $(n \log n)/2$.

# 5 Randomized algorithm for FindAny

We present a randomized algorithm supporting Insert, Delete and FindAny using, on an average, a constant number of comparisons per operation.

## 5.1 The algorithm

The algorithm maintains three variables: $S$, $e$ and *rank*. $S$ is the set of elements currently in the data-structure, $e$ is an element in $S$, and *rank* is the rank of $e$ in $S$. Initially, $S$ is the empty set, and $e$ and *rank* are null. The algorithm responds to instructions as follows.

Insert($x$): Set $S \leftarrow S \cup \{x\}$. With probability $1/|S|$ we set $e$ to $x$ and let *rank* be the rank of $e$ in $S$, that is the number of elements in $S$ strictly less than $e$. In the other case, that is with probability $1 - 1/|S|$, we retain the old value of $e$; that is, we compare $e$ and $x$ and update *rank* if necessary. In particular, if the set was empty before the instruction, then $e$ is assigned $x$ and *rank* is set to 1.

Delete($x$): Set $S \leftarrow S - \{x\}$. If $S$ is empty then set $e$ and *rank* to null and return. Otherwise (i.e. if $S \neq \emptyset$), if $x \equiv e$ then get the new value of $e$ by picking an element of $S$ randomly; set *rank* to be the rank of $e$ in $S$. On the other hand, if $x$ is different from $e$, then decrement *rank* by one if $x < e$.

FindAny: Return $e$ and *rank*.

## 5.2 Analysis

**Claim 6** *The expected number of comparisons made by the algorithm for a fixed instruction in any sequence of instructions is constant.*

*Proof.* FindAny takes no comparisons. Consider an Insert instruction. Suppose the number of elements in $S$ just before the instruction was $s$. Then, the expected number of comparisons made by the algorithm is $s \cdot (1/(s+1)) + 1 \cdot (s/(s+1)) < 2$.

We now consider the expected number of comparisons performed for a Delete instruction. Fix a sequence of instructions. Let $S_i$ and $e_i$ be the values of $S$ and $e$ just before the $i$th instruction. Note that $S_i$ depends only on the sequence of instructions and not on the coin tosses of the algorithm; on the other hand, $e_i$ might vary depending on the coin tosses of the algorithm. We first show that the following invariant holds for all $i$:

$$|S_i| \neq \emptyset \implies \Pr[e_i = x] = \frac{1}{|S_i|} \quad \text{for all } x \in S_i. \tag{2}$$

We use induction on $i$. For $i = 1$, $S_i$ is empty and the claim holds trivially. Assume that the claim holds for $i = l$; we shall show that then it holds for $i = l + 1$. If the $l$th instruction is a FindAny, then $S$ and $e$ are not disturbed and the claim continues to hold.

Suppose the $l$th instruction is an Insert. For $x \in S_l$, we can have $e_{l+1} = x$ only if $e_l = x$ and we retain the old value of $e$ after the Insert instruction. The probability that we retain the old value of $e$ is $|S_l|/(|S_l| + 1)$. Thus, using the induction hypothesis, we have for all $x \in S_l$

$$\Pr[e_{l+1} = x] = \Pr[e_l = x] \cdot \Pr[e_{l+1} = e_l] = \frac{1}{|S_l|} \cdot \frac{|S_l|}{|S_l| + 1} = \frac{1}{|S_l| + 1}.$$

Also, the newly inserted element is made $e_{l+1}$ with probability $\frac{1}{|S_l|+1}$. Since $|S_{l+1}| = |S_l| + 1$, (2) holds for $i = l + 1$.

Next, suppose the $l$th instruction is a $\mathsf{Delete}(x)$. If the set becomes empty after this instruction, there is nothing to prove. Otherwise, for all $y \in S_{l+1}$,

$$\begin{aligned}
\Pr[e_{l+1} = y] &= \Pr[e_l = x \ \&\ e_{l+1} = y] + \Pr[e_l \neq x \ \&\ e_{l+1} = y] \\
&= \Pr[e_l = x] \cdot \Pr[e_{l+1} = y \mid e_l = x] + \Pr[e_l \neq x] \cdot \Pr[e_l = y \mid e_l \neq x] \\
&= \frac{1}{|S_l|} \cdot \frac{1}{|S_{l+1}|} + (1 - \frac{1}{|S_l|}) \cdot \frac{1}{|S_l| - 1} \\
&= \frac{1}{|S_{l+1}|}.
\end{aligned}$$

Thus, (2) holds for $i = l + 1$. This completes the induction.

Now, suppose the $i$th instruction is $\mathsf{Delete}(x)$. Then, the probability that $e_i = x$ is precisely $1/|S_i|$. Thus, the expected number of comparisons performed by the algorithm is

$$(|S_i| - 2) \cdot \frac{1}{|S_i|} < 1.$$

∎

# 6 Randomized lower bounds for FindMin

One may view the problem of maintaining the minimum as a game between two players: the algorithm and the adversary. The adversary gives instructions and supplies answers for the comparisons made by the algorithm. The objective of the algorithm is to respond to the instructions by making as few comparisons as possible, whereas the objective of the adversary is to force the algorithm to use a large number of comparisons.

Similarly, if randomization is permitted while maintaining the minimum, one may consider the randomized variants of this game. We have two cases based on whether or not the adversary is adaptive. An adaptive adversary constructs the input as the game progresses; its actions depend on the moves the algorithm has made so far. On the other hand, a non-adaptive adversary fixes the instruction sequence and the ordering of the elements before the game begins. The input it constructs can depend on the algorithm's strategy but not on its coin toss sequence.

It can be shown that against the adaptive adversary randomization does not help. In fact, if there is a randomized strategy for the algorithm against an adaptive adversary then there is a deterministic strategy against the adversary. Thus, the complexity of maintaining the minimum in this case is the same as in the deterministic case. In this section, we show lower bounds with a non-adaptive adversary.

The input to the algorithm is specified by fixing a sequence of Insert, Delete and FindMin instructions, and an ordering for the set $\{a_1, a_2, \ldots, a_n\}$, based on which the comparisons of the algorithm are answered.

**Distributions.** We will use two distributions on inputs. For the first distribution, we construct a random input $I$ by first picking a random permutation $\sigma$ of $[n]$; we associate with $\sigma$ the sequence of instructions

$$\mathsf{Insert}(a_1), \mathsf{Insert}(a_2), \ldots, \mathsf{Insert}(a_n), \mathsf{Delete}(a_{\sigma(1)}), \mathsf{Delete}(a_{\sigma(2)}), \ldots, \mathsf{Delete}(a_{\sigma(n)}), \tag{3}$$

and the ordering

$$a_{\sigma(1)} < a_{\sigma(2)} < \ldots < a_{\sigma(n)}. \tag{4}$$

For the second distribution, we construct the random input $J$ by picking $i \in [n]$ at random and a random permutation $\sigma$ of $[n]$; the instruction sequence associated with $i$ and $\sigma$ is

$$\mathsf{Insert}(a_1), \ldots, \mathsf{Insert}(a_n), \mathsf{Delete}(a_{\sigma(1)}), \mathsf{Delete}(a_{\sigma(2)}), \ldots, \mathsf{Delete}(a_{\sigma(i-1)}), \mathsf{FindMin}, \tag{5}$$

and the ordering is given, as before, by (4).

For an algorithm $\mathcal{A}$ and an input $I$, let $C_U(\mathcal{A}, I)$ be the number of comparisons made by the algorithm while responding to the Insert and Delete instructions corresponding to $I$; let $C_F(\mathcal{A}, I)$ be the number of comparisons made by the algorithm while responding to the FindMin instructions.

**Theorem 7** *Let $\mathcal{A}$ be a deterministic algorithm for maintaining the minimum. Suppose*

$$\mathop{\mathbf{E}}_{I}[C_U(\mathcal{A}, I)] \leq tn. \tag{6}$$

*Then*

$$\mathop{\mathbf{GM}}_{J}[C_F(\mathcal{A}, J) + 1] \geq \frac{n}{e2^t}.$$

Before we discuss the proof of this result, we derive from it the lower bounds on the randomized and average case complexities of maintaining the minimum. Yao showed that a randomized algorithm can be viewed as a random variable assuming values in some set of deterministic algorithms according to some probability distribution over the set [9]. The randomized lower bound follows from this fact and Theorem 7.

**Corollary 8 (Randomized complexity)** *Let $\mathcal{R}$ be a randomized algorithm for* Insert-Delete-FindMin *with expected amortized time per update at most $t = t(n)$. Then the expected time for* FindMin *is at least $n/(e2^{2t}) - 1$.*

*Proof.* We view $\mathcal{R}$ as a random variable taking values in a set of deterministic algorithms with some distribution. For every deterministic algorithm $\mathcal{A}$ in this set, let

$$t(\mathcal{A}) \stackrel{\text{def}}{=} \mathop{\mathbf{E}}_{I}[C_U(\mathcal{A}, I)]/n.$$

Then by Theorem 7 we have $\mathop{\mathbf{GM}}_{J}[C_F(\mathcal{A}, J) + 1] \geq \left(\frac{n}{e}\right) \cdot 2^{-t(\mathcal{A})}$. Hence,

$$\mathop{\mathbf{GM}}_{\mathcal{R}}[\mathop{\mathbf{GM}}_{J}[C_F(\mathcal{R}, J) + 1] \;\geq\; \mathop{\mathbf{GM}}_{\mathcal{R}}[\left(\frac{n}{e}\right) \cdot 2^{-t(\mathcal{R})}] \;=\; \left(\frac{n}{e}\right) \cdot 2^{-\mathop{\mathbf{E}}_{\mathcal{R}}[t(\mathcal{R})]}.$$

Since the expected amortized time per update is at most $t$, we have $\mathbf{E}_{\mathcal{R}}[t(\mathcal{R})] \leq 2t$. Hence,

$$\mathop{\mathbf{E}}_{\mathcal{R},J}[C_F(\mathcal{R}, J)] + 1 = \mathop{\mathbf{E}}_{\mathcal{R},J}[C_F(\mathcal{R}, J) + 1] \geq \mathop{\mathbf{GM}}_{\mathcal{R},J}[C_F(\mathcal{R}, J) + 1] \geq \frac{n}{e2^{2t}}.$$

Thus, there exists an instance of $J$ with instructions of the form (5), for which the expected number of comparisons performed by $\mathcal{A}$ in response to the last FindMin instruction is at least $n/(e2^{2t}) - 1$. ∎

The average case lower bound follows from the arithmetic-geometric mean inequality and Theorem 7.

**Corollary 9 (Average case complexity)** *Let $\mathcal{A}$ be a deterministic algorithm for* Insert-Delete-FindMin *with amortized time per update at most $t = t(n)$. Then the expected time to find the minimum for inputs with distribution $J$ is at least $n/(e2^{2t}) - 1$.*

*Proof.* $\mathcal{A}$ takes amortized time at most $t$ per update. Therefore,

$$\mathop{\mathbf{E}}_{I}[C_U(\mathcal{A}, I)] \leq 2tn.$$

Then, by Theorem 7 we have

$$\mathop{\mathbf{E}}_{J}[C_F(\mathcal{A}, J)] + 1 = \mathop{\mathbf{E}}_{J}[C_F(\mathcal{A}, J) + 1] \geq \mathop{\mathbf{GM}}_{J}[C_F(\mathcal{A}, J) + 1] \geq \frac{n}{e2^{2t}}.$$

∎

## 6.1 Proof of Theorem 7

**The Decision Tree representation.** Consider the set of sequences in support$[I]$. The actions of a deterministic algorithm on this set of sequences can be represented by a decision tree with *comparison* nodes and *deletion* nodes. (Normally a decision tree representing an algorithm would also have *insertion* nodes, but since, in support$[I]$, the elements are always inserted in the same order, we may omit them.) Each comparison node is labeled by a comparison of the form $a_i : a_j$, and has two children, corresponding to the two outcomes $a_i > a_j$ and $a_i \leq a_j$. Each deletion node has a certain number of children and each edge, $e$, to a child, is labeled by some element $a_e$, denoting that element $a_e$ is deleted by this delete instruction.

For a sequence corresponding to some permutation $\sigma$, the algorithm behaves as follows. The first instruction it must process is $\mathsf{Insert}(a_1)$. The root of the tree is labeled by the first comparison that the algorithm makes in order to process this instruction. Depending on the outcome of this comparison, the algorithm makes one of two comparisons, and these label the two children of the root. Thus, the processing of the first instruction can be viewed as following a path down the tree. Depending on the outcomes of the comparisons made to process the first instruction, the algorithm is currently at some vertex in the tree, and this vertex is labeled by the first comparison that the algorithm makes in order to process the second instruction. In this way, the processing of all the insert instructions corresponds to following a path consisting of comparison nodes down the tree. When the last insert instruction has been processed, the algorithm is at a delete node corresponding to the first delete instruction. Depending on the sequence, some element, $a_{\sigma(1)}$ is deleted. The algorithm follows the edge labeled by $a_{\sigma(1)}$ and the next vertex is labeled by the first comparison that the algorithm makes in order to process the next delete instruction. In this manner, each sequence determines a path down the tree, terminating at a leaf.

We make two simple observations. First, since, in different sequences, the elements are deleted in different orders, each sequence reaches a distinct leaf of the tree. Hence the number of leaves is exactly $n!$. Second, consider the ordering information available to the algorithm when it reaches a delete node $v$. This information consists of the outcomes of all the comparisons on the comparison nodes on the path from the root to $v$. This information can be represented as a poset, $P_v$, on the elements not deleted yet. For every sequence that causes the algorithm to reach $v$, the algorithm has obtained only the information in $P_v$. If a sequence corresponding to some permutation $\sigma$ causes the algorithm to reach $v$, and deletes $a_i$, then $a_i$ is a minimal element in $P_v$, since, in $\sigma$, $a_i$ is the minimum among the remaining elements. Hence each of the elements labeling an edge from $v$ to a child is a minimal element of $P_v$. If this $\mathsf{Delete}$ instruction was replaced by a $\mathsf{FindMin}$, then the comparisons done by the $\mathsf{FindMin}$ would have to find the minimum among these minimal elements. A comparison between any two poset elements can cause at most one of these minimal elements to become non-minimal. Hence, the $\mathsf{FindMin}$ instruction would cost the algorithm $\mathsf{deg}(v) - 1$ comparisons.

**The proof.** Let $T$ be the decision tree corresponding to the deterministic algorithm $\mathcal{A}$. Set $m = n!$. For $l \in \mathsf{leaves}(T)$, let $D_l$ be the set of delete nodes on the path from the root to $l$, and $C_l$ be the set of comparison nodes on the path from the root to $l$.

Each input specified by a permutation $\sigma$ and a value $i \in [n]$, in support$[J]$ causes the algorithm to follow a path in $T$ upto some delete node, $v$, where, instead of a Delete, the sequence issues a FindMin instruction. As argued previously, the number of comparisons made to process this FindMin is at least $\mathsf{deg}(v)-1$. There are exactly $n$ delete nodes on any path from the root to a leaf and different inputs cause the algorithm to arrive at a different delete nodes. Hence

$$\mathop{\mathbf{GM}}_{J}[C_F(\mathcal{A}, J) + 1] \geq \prod_{l \in \mathsf{leaves}(T)} \prod_{v \in D_l} (\mathsf{deg}(v))^{1/nm}. \tag{7}$$

Since $T$ has $m$ leaves, we have using Lemma 3 that

$$\begin{aligned} m &\leq \mathop{\mathbf{GM}}_{l \in \mathsf{leaves}(T)} [\prod_{v \in \mathsf{path}(l)} \mathsf{deg}(v)] \\ &= \mathop{\mathbf{GM}}_{l \in \mathsf{leaves}(T)} [\prod_{v \in C_l} \mathsf{deg}(v)] \cdot \mathop{\mathbf{GM}}_{l \in \mathsf{leaves}(T)} [\prod_{v \in D_l} \mathsf{deg}(v)]. \end{aligned} \tag{8}$$

Consider the first term on the right. Since every comparison node $v$ has arity at most two, we have $\prod_{v \in C_l} \mathsf{deg}(v) = 2^{|C_l|}$. Also, by the supposition of Theorem 7,

$$\mathop{\mathbf{E}}_{l \in \mathsf{leaves}(T)} [|C_l|] = \mathop{\mathbf{E}}_{I}[C_U(A, I)] \leq tn.$$

Thus

$$\mathop{\mathbf{GM}}_{l \in \mathsf{leaves}(T)} [\prod_{v \in C_l} \mathsf{deg}(v)] \;\leq\; \mathop{\mathbf{GM}}_{l \in \mathsf{leaves}(T)} [2^{|C_l|}] \;\leq\; 2^{\mathbf{E}_l[|C_l|]} \;\leq\; 2^{tn}.$$

From this and (8), we have

$$\mathop{\mathbf{GM}}_{l \in \mathsf{leaves}(T)} [\prod_{v \in D_l} \mathsf{deg}(v)] \geq m 2^{-tn}.$$

Then using (7) and the inequality $n! \geq (n/e)^n$, we get

$$\begin{aligned} \mathop{\mathbf{GM}}_{J}[C_F(\mathcal{A}, J) + 1] &\geq \prod_{l \in \mathsf{leaves}(T)} \prod_{v \in D_l} (\mathsf{deg}(v))^{1/nm} \\ &= (\mathop{\mathbf{GM}}_{l \in \mathsf{leaves}(T)} [\prod_{v \in D_l} \mathsf{deg}(v)])^{1/n} \;\geq\; \frac{n}{e 2^t}. \end{aligned}$$

∎

**Remark.** One may also consider the problem of maintaining the minimum when the algorithm is allowed to use an operator that enables it to compute the minimum of some $m$ values in one step. The case $m = 2$ corresponds to the binary comparisons model considered in the proof above. Since an $m$-ary minimum operation can be simulated by $m-1$ binary minimum operations, the above proof yields a bound of

$$\frac{n}{e 2^{2t(m-1)}} - 1.$$

However, by modifying the proof one can show the better bound of

$$\frac{1}{m-1}\left[\frac{n}{e m^{2t}} - 1\right].$$

## References


[1] Mikhail J. Atallah and S. Rao Kosaraju. An adversary-based lower bound for sorting. *Information Processing Letters*, 13:55–57, 1981.

[2] Gerth Stølting Brodal. Fast meldable priority queues. In *Proc. 4th Workshop on Algorithms and Data Structures (WADS)*, volume 955 of *Lecture Notes in Computer Science*, pages 282–290. Springer Verlag, Berlin, 1995.

[3] Svante Carlsson, Patricio V. Poblete, and J. Ian Munro. An implicit binomial queue with constant insertion time. In *Proc. 1st Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 318 of *Lecture Notes in Computer Science*, pages 1–13. Springer Verlag, Berlin, 1988.

[4] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.

[5] Rudolf Fleischer. A simple balanced search tree with $O(1)$ worst-case update time. In *Algorithms and Computation: 4th International Symposium, ISAAC '93*, volume 762 of *Lecture Notes in Computer Science*, pages 138–146. Springer Verlag, Berlin, 1993.

[6] G. H. Hardy, J. E. Littlewood, and G. Polya. Inequalities. Cambridge University Press, Cambridge, 1952.

[7] Christos Levcopoulos and Mark H. Overmars. A balanced search tree with $O(1)$ worst-case update time. *ACTA Informatica*, 26:269–277, 1988.

[8] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

[9] A. C-C. Yao. Probabilistic computations: Towards a unified measure of complexity. In *Proc. of the 17th Symp. on Found. of Comp. Sci.*, 222-227, 1977.