Priority Queues Resilient to Memory Faults

Allan Grønlund Jørgensen[1,⋆], Gabriel Moruz[1], and Thomas Mølhave[1,⋆⋆]

BRICS[⋆⋆⋆], MADALGO[†], Department of Computer Science, University of Aarhus, Denmark. E-mail: {jallan,gabi,thomasm}@daimi.au.dk

Abstract. In the faulty-memory RAM model, the content of memory cells can get corrupted at any time during the execution of an algorithm, and a constant number of uncorruptible registers are available. A resilient data structure in this model works correctly on the set of uncorrupted values. In this paper we introduce a resilient priority queue. The deletemin operation of a resilient priority queue returns either the minimum uncorrupted element or some corrupted element. Our resilient priority queue uses $O(n)$ space to store n elements. Both insert and deletemin operations are performed in $O(\log n + \delta)$ time amortized, where δ is the maximum amount of corruptions tolerated. Our priority queue matches the performance of classical optimal priority queues in the RAM model when the number of corruptions tolerated is $O(\log n)$. We prove matching worst case lower bounds for resilient priority queues storing only structural information in the uncorruptible registers between operations.

1 Introduction

Memory devices continually become smaller, work at higher frequencies and lower voltages, and in general have increased circuit complexity [1]. Unfortunately, these improvements come at the cost of reliability [2, 3]. A number of factors, such as alpha particles, infrared radiation, and cosmic rays, can cause *soft memory errors* where a bit flips and as a consequence the value stored in the corresponding memory cell is corrupted. An unreliable memory can cause problems in most software ranging from the harmless to the very serious, such as breaking cryptographic protocols [4, 5], taking control of a Java Virtual Machine [6] or breaking smart-cards and other security processors [7–9]. Furthermore, many modern computing centers consist of relatively cheap of-the-shelf components, and the large number of individual memories involved in these clusters substantially increase the frequency of memory corruptions in the system. Hence it is crucial that the software running on these machines is robust. Since the amount of cosmic rays increases dramatically with altitude, soft memory

⋆ Supported in part by an Ole Roemer Scholarship

⋆⋆ Supported in part by an Ole Roemer Scholarship from the Danish National Science Research Council and by a Scholarship from the Oticon Foundation.

⋆⋆⋆ Basic Research in Computer Science, research school.

† Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

errors are of special concern in fields like avionics or space research. Furthermore, soft memory error rates are expected to rise for both DRAM and SRAM memories [2].

At the hardware level, the soft memory errors can be handled by means of error detection mechanisms such as parity checking, redundancy or Hamming codes. Unfortunately, implementing these mechanisms incur penalties with respect to performance, size and money. Therefore, memories using these technologies are rarely found in large scale computing clusters or ordinary workstations. On the software level, a series of low-level techniques have been proposed for dealing with the soft memory errors, many of them coping with corrupted instructions. Examples include algorithm based fault tolerance [10], assertions [11], control flow checking [12], or procedure duplication [13].

Traditionally, the work within the algorithmic community has focused on models where the integrity of the memory system is not an issue. In these models, the corruption of even a single memory cell can have a dramatic effect on the output. For instance, a single corrupted value can induce as much as $\Theta(n^2)$ inversions in the output of a standard implementation of mergesort [14]. Replication can help in dealing with corruptions, but is not always feasible, since the time and space overheads are not negligible.

A multitude of algorithms that deal with unreliable information in various ways were developed during the last decades. Aumann and Bender [15] introduced *fault tolerant pointer-based data structures*. In their model, error detection is done upon access, *i.e.* accessing a faulty pointer yields an error message. Obviously, this is not always the case in practice, since a pointer might get corrupted to a valid value and thus an error is not reported. Furthermore, their algorithms allow a certain amount of the data structure to be lost upon corruptions, and this is not accepted in many practical applications. The *liar model* considers algorithms in a comparison model where the result of a comparison is unreliable. Work in this model include fundamental problems such as sorting and searching [16–18]. A standard technique used in the design of algorithms in the liar model is query replication, which is not of much help when memory cells, and not comparisons, are unreliable. Kutten and Peleg [19, 20] introduced the concept of *fault local mending* in the context of distributed networks. A problem is fault locally mendable if there exists a correction algorithm whose running time depends only on the (unknown) number of faults. Some other works studying network fault tolerance include [21–27].

Finocchi and Italiano [14] introduced the *faulty-memory random access machine*, which is a random access machine where the content of memory cells can get corrupted at *any time* and at *any location*. Corrupted cells cannot be distinguished from uncorrupted cells. The model is parametrized by an upper bound δ on the number of corruptions occurring during the lifetime of an algorithm. It is assumed that $O(1)$ reliable memory cells are provided, a reasonable assumption since CPU registers are considered reliable. Also, copying an element is considered an atomic operation, *i.e.* the elements are not corrupted while being copied. An algorithm is *resilient* if it is able to achieve a correct output at least

for the uncorrupted values. This is the best one can hope for, since the output can get corrupted just after the algorithm finishes its execution. For instance a resilient sorting algorithm guarantees that there are no inversions between the uncorrupted elements in the output sequence.

Several important results has been achieved in the faulty-memory RAM. In the original paper, Finocchi and Italiano [14] proved lower bounds and gave (non-optimal) resilient algorithms for sorting and searching. Algorithms matching the lower bounds for sorting and searching(expected time) were presented in [28]. An optimal resilient sorting algorithm takes $\Theta(n \log n + \delta^2)$ time, whereas optimal searching is performed in $\Theta(\log n + \delta)$ time. Furthermore, in [29] a resilient search tree that performs searches and updates in $O(\log n + \delta^2)$ time amortized was developed. Finally, in [30] it was shown that resilient sorting algorithms are of practical interest.

Results. In this paper we design and analyze a priority queue in the faulty-memory RAM model. It uses $O(n)$ space for storing n elements and performs both Insert and Deletemin in $O(\log n + \delta)$ time amortized. Our priority queue matches the bounds for an optimal comparison based priority queue in the RAM model while tolerating $O(\log n)$ corruptions. It is a significant improvement over using the resilient search tree in [29] as a priority queue, since it uses $O(\log n + \delta^2)$ time amortized per operation and thus only tolerates $O(\sqrt{\log n})$ corruptions to preserve the $O(\log n)$ bound per operation. Our priority queue is the first resilient data structure allowing $O(\log n)$ corruptions, while still matching optimal bounds in the RAM model. Our priority queue does not store elements in reliable memory between operations, only structural information like pointers and indices. We prove that any comparison based resilient priority queue behaving this way requires worst case $\Omega(\log n + \delta)$ time for either Insert or Deletemin.

The resilient priority queue is based on the cache-oblivious priority queue by Arge *et al.* [31]. The main idea is to gather elements in large sorted groups of increasing size, such that expensive updates do not occur too often. The smaller groups contain the smaller elements, so they can be retrieved faster by Deletemin operations. We extensively use the resilient merging algorithm in [28] to move elements among the groups. Due to the large sizes of the groups, the extra work required to deal with corruptions in the merging algorithm becomes insignificant compared to the actual work done.

Outline. The remainder of the paper is structured as follows. In Section 2 we define the resilient priority queue and introduce some notation. We give a detailed description of the resilient priority queue in Section 3, while in Section 4 we prove its correctness and complexity bounds. Finally, in Section 5 we prove matching lower bounds for resilient priority queues.

2 Preliminaries

In this section we define the resilient priority queue and introduce some notation used throughout the paper.

Given two sequences X and Y, we let XY denote the *concatenation* of X and Y. A sequence X is *faithfully ordered* if its uncorrupted keys appear in non-decreasing order. Finally, a *reliable value* is a value stored in unreliable memory which can be retrieved reliably in spite of possible corruptions. This is achieved by replicating the given value $2\delta+1$ times. Retrieving a reliable value takes $O(\delta)$ time using the majority algorithm in [32], which scans the $2\delta+1$ values keeping a single majority candidate and a counter in reliable memory.

Definition 1. *A resilient priority queue maintains a set of elements under the operations* Insert *and* Deletemin*. An* Insert *adds an element and a* Deletemin *deletes and returns the minimum uncorrupted element or a corrupted one.*

We note that our definition of a resilient priority queue is consistent with the resilient sorting algorithms introduced in [14]. Given a sequence of n elements, inserting all of them into a resilient priority queue followed by n Deletemin operations yields a faithfully ordered sequence.

3 Fault tolerant priority queue

In this section we introduce the resilient priority queue. It resembles the cache-oblivious priority queue by Arge *et al.* [31]. The elements are stored in faithfully ordered lists and are moved using two fundamental primitives, Push and Pull, based on faithful merging. We describe the structure of the priority queue in Section 3.1 and then introduce the Push and Pull primitives in Section 3.2. Finally, in Section 3.3, we describe the Insert and Deletemin operations.

3.1 Structure

The resilient priority queue consists of an insertion buffer I together with a number of layers $L_0, \ldots, L_k$, with $k = O(\log n)$. Each layer L_i contains an up-buffer U_i and a down-buffer D_i, represented as arrays. Intuitively, the up-buffers contain large elements that are on their way to the upper layers in the priority queue, whereas the down-buffers contain small elements, on their way to lower layers. The buffers in the priority queue are stored as a doubly linked list $U_0, D_0, \ldots, U_k, D_k$, see Figure 1. For each up and down buffer we reliably store the pointers to their adjacent buffers in the linked list and their size. In the reliable memory we store pointers to I, U_0 and D_0, together with $|I|$. Since the position of the first element in U_0 and D_0 is not always the first memory cell of the corresponding buffer, we also store the index of the first element in these buffers in reliable memory. The insertion buffer I contains up to $b = \delta + \log n + 1$ elements. For layer L_i we define the threshold s_i by $s_0 = 2 \cdot (\delta^2 + \log^2 n)$ and $s_i = 2s_{i-1} = 2^{i+1} \cdot (\delta^2 + \log^2 n)$, where n is the number of elements in the priority queue. We use these thresholds to decide whether an up buffer contains too many elements or whether a down buffer has too few. For the sake of

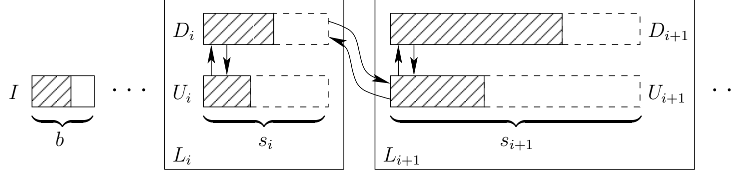

Fig. 1. The structure of the priority queue. The buffers are stored in a doubly linked list using reliably stored pointers. Additionally, the size of each buffer is stored reliably.

simplicity, the up and down buffers are grown and shrunk as needed during the execution such that they don't use any extra space.

To structure the priority queue, we maintain the following invariants for the up and down buffers.

– *Order invariants:*
  1. All buffers are faithfully ordered.
  2. D_iD_{i+1} and D_iU_{i+1} are faithfully ordered, for $0 \le i < k$.
– *Size invariants:*
  3. $s_i/2 \le |D_i| \le s_i$, for $0 \le i < k$.
  4. $|U_i| \le s_i/2$, for $0 \le i < k$.

By maintaining all the up and down buffers faithfully ordered, it is possible to move elements between neighboring layers efficiently, using faithful merging. By invariant 2, all uncorrupted elements in D_i are smaller than all uncorrupted elements in both D_{i+1} and U_{i+1}. This ensures that small elements belong to the lower layers of the priority queue. We note that there is no assumed relationship between the elements in the up and down buffers in the same layer. Finally, the size invariants allow the sizes of the buffers to vary within a large range. This way, $\Omega(s_i)$ Insert or DeleteMin operations occur between two operations on the same buffer in L_i, yielding the desired amortized bounds.

Since the s_i values depend on n, whenever the size of the priority queue increases or decreases by $\Theta(n)$, we perform a global rebuilding. This rebuilding is done by collecting all elements, sorting them with an optimal resilient sorting algorithm [28], and redistributing the output into the down buffers of all the layers starting with L_0. After the global rebuilding, the up buffers are empty and the down buffers full, except possibly the last down buffer.

3.2 Push and pull primitives

We now introduce the two fundamental primitives used by the priority queue. The Push primitive is invoked when an up buffer contains too many elements, breaking invariant 4. It "pushes" elements upwards, repairing the size invariants locally. The Pull operation is invoked when a down buffer contains too few elements, breaking invariant 3. It fills this down buffer by "pulling" elements

from the layer above, again locally repairing the size invariants. Both operations faithfully merge consecutive buffers in the priority queue and redistribute the resulting sequence among the participating buffers. After merging, we deallocate the old buffers and allocate new arrays for the new buffers.

Push. The PUSH primitive is invoked when an up buffer U_i breaks invariant 4, *i.e.* when it contains more than $s_i/2$ elements. In this case we merge U_i, D_i and U_{i+1} into a sequence M using the resilient merging algorithm in [28]. We then distribute the elements in M by placing the first $|D_i| - \delta$ elements in a new buffer D'_i, and the remaining $|U_{i+1}|+|U_i|+\delta$ elements in a new buffer U'_{i+1}. After the merge, we create an empty buffer, U'_i, and deallocate the old buffers. If U'_{i+1} contains too many elements, breaking invariant 4, the PUSH primitive is invoked on U'_{i+1}. When L_i is the last layer, we fill D'_i with the first elements of M and create a new layer L_{i+1} placing the remaining elements of M into D'_{i+1} instead of U'_{i+1}. Since $|D'_i|$ is smaller than $|D_i|$, it could violate invariant 3. This situation is handled by using the PULL operation and is described after introducing PULL.

Unlike the priority queue in [31], the PUSH operation decreases the size of a down buffer. This is required to preserve invariant 2, in spite of corruptions. After a PUSH call, D'_i can contain elements from $U_i \cup U_{i+1}$. Since there is no assumed relationship between elements in $U_i \cup U_{i+1}$ and those in $D_{i+1} \cup U_{i+2}$, we need to ensure that each element in D'_i originating from $U_i \cup U_{i+1}$ is faithfully smaller than the elements in $D_{i+1} \cup U_{i+2}$. Assume the size of D_i is preserved, *i.e.* $|D'_i| = |D_i|$. Consider a corruption that alters an element in D_i to some large value before the PUSH. This corrupted value could be placed in U'_{i+1} and, since $|D'_i| = |D_i|$, an element from $U_i \cup U_{i+1}$ must be placed in D'_i. This new element in D'_i potentially violates invariant 2.

Pull. The PULL operation is called on a down buffer D_i when it contains less than $s_i/2$ elements, breaking invariant 3. In this case, the buffers D_i, U_{i+1}, and D_{i+1} are merged into a sequence M using the resilient merging algorithm in [28]. The first s_i elements from M are written to a new buffer D'_i, and the next $|D_{i+1}|-(s_i-|D_i|)-\delta$ elements are written to D'_{i+1}. The remaining elements of M are written to U'_{i+1}. A PULL is invoked on D'_{i+1}, if it is too small.

Similar to the PUSH operation, the extra δ elements lost by D_{i+1} ensure that the order invariants hold in spite of possible corruptions. That is, a corruption of an element in $D_i \cup D_{i+1}$ to a very large value may cause an element from U_{i+1} to take the place of the corrupted element in D'_{i+1} and this element is possibly larger than some uncorrupted element in $D_{i+2} \cup U_{i+2}$.

After the merge, U'_{i+1} contains δ more elements than U_{i+1} had before the merge, and thus it is possible that it has too many elements, breaking invariant 4. We handle this situation as follows. Consider a maximal series of subsequent PULL invocations on down buffers $D_i, D_{i+1}, \ldots, D_j$, $0 \leq i < j < k$. After the first PULL call on D_i and before the call on D_{i+1} we store a pointer to D_i in the reliable memory. After all the PULL calls we investigate all the affected up buffers, by simply following the pointers between the buffers starting from D_i, and invoke

the Push primitive wherever necessary. The case when Push operations cause down buffers to underflow is handled similarly.

3.3 Insert and deletemin

An element is inserted in the priority queue by simply appending it to the insertion buffer I. If I gets full, its elements are added to U_0 by first faithfully sorting I and then faithfully merging I and U_0. If U_0 breaks invariant 4, we invoke the Push primitive. If L_0 is the only layer of the priority queue and D_0 violates the size constraint, we faithfully merge the elements in I with D_0 instead.

To delete the minimum element in the priority queue, we first find the minimum of the first $\delta+1$ values in D_0, the minimum of the first $\delta+1$ values in U_0, and the minimum element in I. We then take the minimum of these three elements, delete it from the appropriate buffer and return it. After deleting the minimum, we right-shift all the elements in the affected buffer from the beginning up to the position of the minimum. This way we ensure that elements in any buffer are stored consecutively. If D_0 underflows, we invoke the Pull primitive on D_0, unless L_0 is the only layer in the priority queue. If U_0 or D_0 contains $\Theta(\log n + \delta)$ empty cells, we create a new buffer and copy the elements from the old buffer to the new one.

4 Analysis

In this section we analyze the resilient priority queue. We prove the correctness in Section 4.1 and analyze the time and space complexity in Section 4.2.

4.1 Correctness

To prove correctness of the resilient priority queue, we show that the Deletemin operation returns the minimum uncorrupted value or a corrupted value. We first prove that the order invariants are maintained by the Pull and Push operations.

Lemma 1. *The* Pull *and* Push *primitives preserve the order invariants.*

Proof. Recall that in a Pull invocation on buffer D_i, the buffers D_i, U_{i+1}, and D_{i+1} are faithfully merged into a sequence M. The elements in M are then distributed into three new buffers D'_i, U'_{i+1}, and D'_{i+1}, see Figure 2. To argue that the order invariants are satisfied we need to show that the elements of the down buffer on layer L_j, for $0 \leq j < k$, are faithfully smaller than the elements of the buffers on layer L_{j+1}, where k is the index of the last layer. The invariants hold trivially for unaffected buffers. The faithful merge guarantees that $D'_i D'_{i+1}$ as well as $D'_i U'_{i+1}$ are faithfully ordered, and thus the individual buffers are also faithfully ordered. Since invariant 2 holds for the original buffers all uncorrupted elements in D_{i+1} and U_{i+1} are larger than the uncorrupted elements in D_i, guaranteeing that $D_{i-1}D'_i$ is faithfully ordered. Finally, we now show that $D_{i+1}D_{i+2}$ and $D_{i+1}U_{i+2}$ are faithfully ordered.

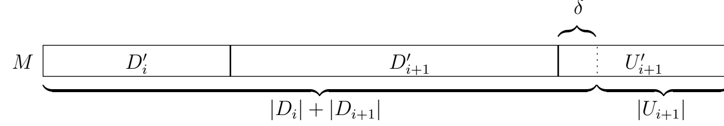

Fig. 2. The distribution of M into buffers.

Let m be the minimum uncorrupted element in $D_{i+2} \cup U_{i+2}$. We need to show that all uncorrupted elements in D'_{i+1} are smaller than m. If no uncorrupted element from U_{i+1} is placed in D'_{i+1}, the invariant holds by the order invariants before the operation. Otherwise, assume that an uncorrupted element $y \in U_{i+1}$ is moved to D'_{i+1}. Since $|U'_{i+1}| = |U_{i+1}| + \delta$ and y is moved to D'_{i+1}, at least $\delta+1$ elements originating from $D_i \cup D_{i+1}$ are contained in U'_{i+1}. Since there can be at most δ corruptions, there exists at least one uncorrupted element, x, among these. By faithful merging, all uncorrupted elements in D'_{i+1} are smaller than x, which means that $y \leq x$. Since x originates from $D_i \cup D_{i+1}$, it is smaller than m. We obtain $y \leq m$.

A similar argument proves correctness of the Push operation. We conclude that both order invariants are preserved by Pull and Push operations. □

Having proved that the order invariants are maintained at all times, we now prove the correctness of the resilient priority queue.

Lemma 2. *The* DeleteMin *operation returns the minimum uncorrupted value in the priority queue or a corrupted value.*

Proof. We recall that the DeleteMin operation computes the minimum of the first $\delta+1$ elements of U_0 and D_0. It compares these values with the minimum of I, found in a scan, and returns the smallest of these elements. Since U_0 and D_0 are faithfully ordered, the minimum of their first $\delta + 1$ elements is either the minimum uncorrupted value in these buffers, or a corrupted value even smaller. Furthermore, according to the order invariants, all the values in layers $L_1, \ldots, L_k$ are faithfully larger than the minimum in D_0. Therefore, the element reported by DeleteMin is the minimum uncorrupted value or a corrupted value. □

4.2 Complexity

In this section we show that our resilient priority queue uses $O(n)$ space and that Insert and DeleteMin take $O(\log n + \delta)$ amortized time. We first prove that the Pull and Push primitives restore the size invariants.

Lemma 3. *If a size invariant is broken for a buffer in L_0, invoking* Pull *or* Push *on that buffer restores the invariants. Furthermore, during this operation* Pull *and* Push *are invoked on the same buffer at most once. No other invariants are broken before or after this operation.*

Proof. Assume that Push is invoked on U_0, and that it is called iteratively up to some layer L_l. By construction of Push, the size invariants for all the up buffers now hold. Since a Push steals δ elements from the down buffers, the layers $L_0, \ldots, L_l$ are traversed again and Pull is invoked on these as needed. The last of these Pull operations might proceed past layer L_l. Similarly, a Pull may cause an up buffer to overflow. However, since the cascading Push operations left $|U_i| = 0$ for $i \leq l$, any new Push are invoked on up buffers only on layer L_{l+1} or higher, thus Push is invoked on each buffer at most once. A similar argument works for the Pull operation. □

Lemma 4. *The resilient priority queue uses $O(n+\delta)$ space to store n elements.*

Proof. The insertion buffer always uses $O(\log n + \delta)$ space. We prove that the remaining layers use $O(n)$ space. For each layer we use $O(\delta)$ space for storing structural information reliably. In all layers, except the last one, the down buffer contains $\Omega(\delta^2)$ elements by invariant 3. This means that for each of these layers the elements stored in the down buffer dominate the space complexity. The structural information of the last layer requires additional $O(\delta)$ space. □

The space complexity of the priority queue can be reduced to $O(n)$ without affecting the time complexity, by storing the structural information of L_0 in safe memory, and by doubling or halving the insertion buffer during the lifetime of the algorithm such that it always uses $O(|I|)$ space.

Lemma 5. *Each* Insert *and* DeleteMin *takes $O(\log n + \delta)$ amortized time.*

Proof. We define the potential function:

$$\Phi = \sum_{i=1}^{k} (c_1 \cdot (\log n - i) \cdot |U_i| + c_2 \cdot i \cdot |D_i|)$$

We use Φ to analyze the amortized cost of a Push operation. In a Push operation on U_i, buffers U_i, D_i, and U_{i+1} are merged. The elements are then distributed into new buffers U'_i, D'_i, and U'_{i+1}, such that $|U'_i| = 0$, $|D'_i| = |D_i| - \delta$, and $|U'_{i+1}| = |U_{i+1}| + |U_i| + \delta$. This gives the following change in potential $\Delta\Phi$:

$$\begin{aligned} \Delta\Phi &= -|U_i| \cdot c_1 \cdot (\log n - i) - \delta \cdot c_2 \cdot i + (|U_i| + \delta) \cdot c_1(\log n - (i+1)) \\ &= -c_1 \cdot |U_i| + \delta(-c_2 \cdot i + c_1 \cdot \log n - c_1 \cdot i - c_1) \ . \end{aligned}$$

Since the Push is invoked on U_i, invariant 4 is not valid for U_i and therefore $|U_i| \geq \frac{s_i}{2} = 2^i (\log^2 n + \delta^2)$. Thus:

$$\Delta\Phi \leq -c_1 \cdot |U_i| + c_1 \cdot \delta \cdot \log n \leq -c_1 \cdot 2^i \cdot (\log^2 n + \delta^2) + c_1 \cdot \delta \cdot \log n \leq -c_1 \cdot c' \cdot |U_i| \ , \quad (1)$$

for some constant $c' > 0$.

Since faithfully merging two sequences of size n takes $O(n+\delta^2)$ time [28], the time used for a Push on U_i is upper bounded by $c_m \cdot (|U_i| + |D_i| + |U_{i+1}| + \delta^2)$, where c_m depends on the resilient merge. This includes the time required for

retrieving reliably stored variables. Adding the time and the change in potential we are able to get the amortized cost less than zero by tweaking c_1 based on equation (1). This is because $|U_i|$ is $\Omega(\delta^2)$ and at most a constant fraction smaller than the participants in the merge.

A similar analysis works for the PULL primitive. We now calculate the amortized cost of INSERT and DELETEMIN. We ignore any PUSH or PULL operations since their amortized costs are negative. The amortized time for inserting an element in I, sorting I, and merging it with U_0 is $O(\log n + \delta)$ per operation. The change in potential when adding elements to L_0 is $O(\log n)$ per element. The time needed to find the smallest element in a DELETEMIN is $O(\log n + \delta)$, and the change in potential when an element is deleted from L_0 is negative.

The cost of global rebuilding is dominated by the cost of sorting, which is $O(n \log n + \delta^2)$. There are $\Theta(n)$ operations between each rebuild, which leads to $O(\log n + \delta)$ time per operation, since $\delta \leq n$, and this concludes the proof. □

Theorem 1. *The resilient priority queue takes $O(n)$ space and uses amortized $O(\log n + \delta)$ time per operation.*

5 Lower bound

In this section we prove that any resilient priority queue takes $\Omega(\log n + \delta)$ time for either INSERT or DELETEMIN in the comparison model, under the assumption that no elements are stored in reliable memory between operations. This implies optimality of our resilient priority queue under these assumptions. We note that the reliable memory may contain any structural information, e.g. pointers, sizes, indices.

Theorem 2. *A resilient priority queue containing n elements, with $n > \delta$, uses $\Omega(\log n + \delta)$ comparisons to perform* INSERT *followed by* DELETEMIN.

Proof. Consider a priority queue Q with n elements, with $n > \delta$, that uses less than δ comparisons for an INSERT followed by a DELETEMIN. Also, Q does not store elements in reliable memory between operations. Assume that no corruptions have occurred so far. Without loss of generality we assume that all the elements in Q are distinct. We prove there exists a series of corruptions C, $|C| \leq \delta$, such that the result of an INSERT of an element e followed by a DELETEMIN returns the same element regardless of the choice of e.

Let $k < \delta$ be the number of comparisons performed by Q during the two operations. We force the result of each comparison to be the same regardless of e by suitable corruptions. In all the comparisons involving e, we ensure that e is the smallest. We do so by corrupting the value which e is compared against if necessary, by adding some positive constant $c \geq e$ to the other value. If two elements different than e are compared, we make sure the outcome is the same as if no corruptions had happened. If one of them was corrupted, adding c to the other one reestablishes their previous ordering. If both of them were corrupted by adding c, their ordering is unchanged and no corruptions are needed. Forcing

any comparison to give the desired outcome requires at most one corruption, and therefore $|C| \leq k < \delta$.

We now consider the value e' returned by DELETEMIN on Q. If $e = e'$ then we choose e to be larger than some element $x \in Q$ not affected by a corruption in C. Such a value exists because the size of the priority queue is larger than δ. Since $e = e' > x$, Q returned an uncorrupted element that was not the minimum uncorrupted element in Q. If $e \neq e'$ we choose e to be smaller than any element in Q. With such a choice of e, no corruptions are required and the value returned by Q was not corrupted, but still larger than e. This proves Q is not resilient.

Adding the classical $\Omega(\log n)$ bound for priority queues in the comparison model the result follows. □

Acknowledgments

We would like to thank Gerth S. Brodal and Lars Arge for their very helpful comments. We would also like to thank the anonymous reviewers for their valuable comments, especially suggestions for simplifying the proof of Lemma 1.

References

1. Constantinescu, C.: Trends and challenges in VLSI circuit reliability. IEEE micro **23**(4) (2003) 14–19
2. Tezzaron Semiconductor: Soft errors in electronic memory - a white paper. http://www.tezzaron.com/about/papers/papers.html (2004)
3. van de Goor, A.J.: Testing Semiconductor Memories: Theory and Practice. ComTex Publishing, Gouda, The Netherlands (1998) ISBN 90-804276-1-6.
4. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults. In: Eurocrypt. (1997) 37–51
5. Xu, J., Chen, S., Kalbarczyk, Z., Iyer, R.K.: An experimental study of security vulnerabilities caused by errors. In: Proc. International Conference on Dependable Systems and Networks. (2001) 421–430
6. Govindavajhala, S., Appel, A.W.: Using memory errors to attack a virtual machine. In: IEEE Symposium on Security and Privacy. (2003) 154–165
7. Anderson, R., Kuhn, M.: Tamper resistance - a cautionary note. In: Proc. 2nd Usenix Workshop on Electronic Commerce. (1996) 1–11
8. Anderson, R., Kuhn, M.: Low cost attacks on tamper resistant devices. In: International Workshop on Security Protocols. (1997) 125–136
9. Skorobogatov, S.P., Anderson, R.J.: Optical fault induction attacks. In: Proc. 4th International Workshop on Cryptographic Hardware and Embedded Systems. (2002) 2–12
10. Huang, K.H., Abraham, J.A.: Algorithm-based fault tolerance for matrix operations. IEEE Transactions on Computers **33** (1984) 518–528
11. Rela, M.Z., Madeira, H., Silva, J.G.: Experimental evaluation of the fail-silent behaviour in programs with consistency checks. In: Proc. 26th Annual International Symposium on Fault-Tolerant Computing. (1996) 394–403
12. Yau, S.S., Chen, F.C.: An approach to concurrent control flow checking. IEEE Transactions on Software Engineering **SE-6**(2) (1980) 126–137

13. Pradhan, D.K.: Fault-tolerant computer system design. Prentice-Hall, Inc. (1996)
14. Finocchi, I., Italiano, G.F.: Sorting and searching in the presence of memory faults (without redundancy). In: Proc. 36th Annual ACM Symposium on Theory of Computing. (2004) 101–110
15. Aumann, Y., Bender, M.A.: Fault tolerant data structures. In: Proc. 37th Annual Symposium on Foundations of Computer Science, Washington, DC, USA, IEEE Computer Society (1996) 580
16. Borgstrom, R.S., Kosaraju, S.R.: Comparison-based search in the presence of errors. In: Proc. 25th Annual ACM symposium on Theory of Computing. (1993) 130–136
17. Lakshmanan, K.B., Ravikumar, B., Ganesan, K.: Coping with erroneous information while sorting. IEEE Transactions on Computers **40**(9) (1991) 1081–1084
18. Ravikumar, B.: A fault-tolerant merge sorting algorithm. In: Proc. 8th Annual International Conference on Computing and Combinatorics. (2002) 440–447
19. Kutten, S., Peleg, D.: Fault-local distributed mending. Journal of Algorithms **30**(1) (1999) 144–165
20. Kutten, S., Peleg, D.: Tight fault locality. SIAM Journal on Computing **30**(1) (2000) 247–268
21. Diks, K., Pelc, A.: Optimal adaptive broadcasting with a bounded fraction of faulty nodes (extended abstract). In: Proc. 5th Annual European Symposium on Algorithms. (1997) 118–129
22. Gasieniec, L., Pelc, A.: Broadcasting with a bounded fraction of faulty nodes. Journal of Parallel and Distributed Computing **42**(1) (1997) 11–20
23. Hastad, J., Leighton, T.: Fast computation using faulty hypercubes. In: Proc. 21st Annual ACM Symposium on Theory of Computing. (1989) 251–263
24. Hastad, J., Leighton, T., Newman, M.: Reconfiguring a hypercube in the presence of faults. In: Proc. 19th Annual ACM Symposium on Theory of Computing. (1987) 274–284
25. Kaklamanis, C., Karlin, A.R., Leighton, F.T., Milenkovic, V., Raghavan, P., Rao, S., Thomborson, C.D., Tsantilas, A.: Asymptotically tight bounds for computing with faulty arrays of processors (extended abstract). In: Proc. 31st Annual Symposium on Foundations of Computer Science. (1990) 285–296
26. Leighton, F.T., Maggs, B.M.: Expanders might be practical: Fast algorithms for routing around faults on multibutterflies. In: Proc. 30th Annual Symposium on Foundations of Computer Science. (1989) 384–389
27. Park, S., Bose, B.: All-to-all broadcasting in faulty hypercubes. IEEE Transactions on Computers **46**(7) (1997) 749–755
28. Finocchi, I., Grandoni, F., Italiano, G.F.: Optimal resilient sorting and searching in the presence of memory faults. In: Proc. 33rd Int. Colloquium on Automata, Languages and Programming. (2006) 286–298
29. Finocchi, I., Grandoni, F., Italiano, G.F.: Resilient search trees. In: Proc. 18th ACM-SIAM Symposium on Discrete Algorithms. (2007) To appear.
30. Petrillo, U.F., Finocchi, I., Italiano, G.F.: The price of resiliency: a case study on sorting with memory faults. In: Proc. 14th Annual European Symposium on Algorithms. (2006) 768–779
31. Arge, L., Bender, M.A., Demaine, E.D., Holland-Minkley, B., Munro, J.I.: Cache-oblivious priority queue and graph algorithm applications. In: Proc. 34th Annual ACM Symposium on Theory of Computing. (2002) 268–276
32. Boyer, R.S., Moore, J.S.: MJRTY: A fast majority vote algorithm. In: Automated Reasoning: Essays in Honor of Woody Bledsoe. (1991) 105–118