Resilient Dictionaries

IRENE FINOCCHI

Università di Roma "La Sapienza"

AND

FABRIZIO GRANDONI AND GIUSEPPE F. ITALIANO

Università di Roma "Tor Vergata"

Abstract. We address the problem of designing data structures in the presence of faults that may arbitrarily corrupt memory locations. More precisely, we assume that an adaptive adversary can arbitrarily overwrite the content of up to δ memory locations, that corrupted locations cannot be detected, and that only $O(1)$ memory locations are safe. In this framework, we call a data structure resilient if it is able to operate correctly (at least) on the set of uncorrupted values. We present a resilient dictionary, implementing search, insert, and delete operations. Our dictionary has $O(\log n + \delta)$ expected amortized time per operation, and $O(n)$ space complexity, where n denotes the current number of keys in the dictionary. We also describe a deterministic resilient dictionary, with the same amortized cost per operation over a sequence of at least δ^{ε} operations, where $\varepsilon > 0$ is an arbitrary constant. Finally, we show that any resilient comparison-based dictionary must take $\Omega(\log n + \delta)$ expected time per search. Our results are achieved by means of simple, new techniques which might be of independent interest for the design of other resilient algorithms.

Categories and Subject Descriptors: E.1 [**Data**]: Data Structures—*Trees*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sorting and searching*; H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Dictionaries*

General Terms: Algorithms, Reliability, Theory

Additional Key Words and Phrases: Memory faults, memory models, unreliable information, software-based fault tolerance, search trees

A preliminary version of this work appeared in Finocchi et al. [2007].
This work was partially supported by the Italian Ministry of University and Research under Project MAINSTREAM "Algorithms for Massive Information Structures and Data Streams".

Authors' addresses: I. Finocchi, Dipartimento di Informatica, Università di Roma "La Sapienza", via Salaria 113, 00198, Roma, Italy, e-mail: finocchi@di.uniroma1.it; F. Grandoni, Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", via del Politecnico 1, 00133 Roma, Italy, e-mail: grandoni@disp.uniroma2.it; G. F. Italiano, Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", via del Politecnico 1, 00133 Roma, Italy, e-mail: italiano@disp.uniroma2.it.

DOI 10.1145/1644015.1644016 http://doi.acm.org/10.1145/1644015.1644016

ACM Reference Format:
Finocchi, I., Grandoni, F., and Italiano, G. F. 2009. Resilient dictionaries. ACM Trans. Algor. 6, 1, Article 1 (December 2009), 19 pages.
DOI = 10.1145/1644015.1644016 http://doi.acm.org/10.1145/1644015.1644016

1. *Introduction*

Memories in modern computing platforms are not always fully reliable, and sometimes the content of a memory word may be temporarily or permanently lost or corrupted. This may depend on manufacturing defects, power failures, or environmental conditions such as cosmic radiation and alpha particles [Hamdioui et al. 2003; May and Woods 1979]. These phenomena can seriously affect the computation, especially if the amount of data to be processed is huge. This is, for example, the case for Web search engines, that store and process Terabytes of dynamic data sets, including inverted indices which have to be maintained sorted for fast document access. For such large data structures, even a small failure probability can result in bit flips in the index, which may become responsible for erroneous answers to keyword searches [Henzinger 2004, 2007].

In many applications involving the usage of large and inexpensive memories, achieving resiliency by means of classical approaches, such as data replication or error-correcting circuitry, could be too expensive in terms of space, time, and money. In view of that, it makes sense to design algorithms and data structures that are *resilient* to memory faults, that is, that are able to compute a correct output (at least) with respect to the set of uncorrupted values, despite the corruption of some memory words before or during their execution. Our aim is to achieve this task without large overheads in terms of space and time complexity. We remark that classical algorithms and data structures are typically not resilient. For instance, if we want to search for a key in a sorted sequence subject to memory faults, corrupted keys may guide the classical binary search in the wrong direction: for this reason we may fail to find not only the corrupted keys, but also the uncorrupted ones.

1.1. PREVIOUS WORK ON FAULTY-RAMS.

In this article we focus on the faulty-RAM model introduced in Finocchi and Italiano [2008]. In this model we assume that there is an adaptive adversary which can corrupt up to δ memory words, in any place and at any time (even simultaneously). By corrupting a memory word, we mean replacing its content with a different, arbitrary value. There is no error detection mechanism to distinguish corrupted memory locations from uncorrupted ones. This pessimistic model captures situations like cosmic-rays bursts and memories with nonuniform fault probability, which would be difficult to be modeled otherwise. We also assume that there are $O(1)$ safe memory words which cannot be accessed by the adversary. Note that without this assumption, no reliable computation is possible: in particular, the $O(1)$ safe memory can store the code of the algorithm itself, which otherwise could be corrupted by the adversary. In the case of randomized algorithms, we assume that the random bits are not accessible to the adversary. Moreover, we assume that reading a memory word (in the unsafe memory) is an atomic operation, that is, the adversary cannot corrupt a memory word after the reading process has started. Without the last two assumptions, most of the power of randomization would be lost in our setting.

A natural approach to the design of algorithms and data structures in the presence of memory faults is data replication. Informally, a *resilient variable* consists of $(2\delta + 1)$ copies $x_1, x_2, \ldots, x_{2\delta+1}$ of a standard variable (see what follows for a formal definition). The value of a resilient variable is given by the majority of its copies. Observe that the value of x is reliable, since the adversary cannot corrupt the majority of its copies. We say that an algorithm/data structure is *trivially resilient* if it implements a nonresilient algorithm/data structure by replacing normal variables with resilient variables. Trivially resilient implementations suffer from a $\Theta(\delta)$ multiplicative overhead in terms of both space and running time. For example a trivially resilient implementation of a standard dictionary based on AVL trees would require $O(\delta n)$ space and $O(\delta \log n)$ time for search, insert, and delete. Thus, it could tolerate only $O(1)$ memory faults while maintaining optimal time and space bounds.

This type of overhead seems unavoidable if one wishes to operate correctly in the faulty-RAM model. For example, with less than $2\delta + 1$ copies of a key, we cannot avoid that its correct value gets lost. Since a $\Theta(\delta)$ multiplicative overhead could be unacceptable in several applications even for small values of δ, the next natural thing to do is relaxing the notion of correctness. We say that an algorithm or data structure is *resilient* to memory faults if, despite the corruption of some memory location during its lifetime, it is nevertheless able to operate correctly (at least) on the set of uncorrupted values. Of course, every trivially resilient algorithm is also resilient. However, as the following examples show, there are resilient algorithms for some natural problems which perform much better than their trivially resilient counterparts. In Finocchi et al. [2006] and Finocchi and Italiano [2008], the *resilient sorting* problem was considered. In this problem, we are given a set K of n keys. We call a key *faithful* if it is never corrupted, and *faulty* otherwise. The problem is to compute a *faithfully sorted* permutation of K, that is, a permutation of K such that the subsequence induced by the faithful keys is sorted. This is the best one can hope for, since the adversary can corrupt a key at the very end of the algorithm execution, thus making faulty keys occupy wrong positions. This problem can be trivially solved in $O(\delta\, n \log n)$ time. An $O(n \log n + \delta^2)$-time sorting algorithm is described in Finocchi et al. [2006]. This result implies that one can sort resiliently in optimal $O(n \log n)$ time while tolerating up to $O(\sqrt{n \log n})$ memory faults: this is the best possible [Finocchi and Italiano 2008]. In the special case of polynomially bounded integer keys, an improved running time $O(n + \delta^2)$ can be achieved [Finocchi et al. 2006].

The *resilient searching* problem was also considered in Finocchi et al. [2006] and Finocchi and Italiano [2008]. Here we are given a sorted sequence K of n keys, and a search key κ. The problem is to return a key (faulty or faithful) of value κ, if K contains a faithful key of that value. If there is no faithful key equal to κ, one can either return *no* or return a (faulty) key equal to κ. Note that also in this case this is the best possible: the adversary may indeed corrupt a key equal to κ at the very beginning of the algorithm execution, or introduce a key of that value at the very end. There is a trivial algorithm which solves this problem in $O(\delta \log n)$ time. The authors show a lower bound of $\Omega(\log n + \delta)$ on the expected running time of any algorithm for the problem. Moreover, they present an optimal $O(\log n + \delta)$ randomized algorithm, and an almost optimal deterministic one, with running time $O(\log n + \delta^{1+\varepsilon})$ for any constant $\varepsilon > 0$. We remark that the aforesaid results hold in the static case, that is, when the set K cannot be modified by adding or deleting keys. In this article we will address the dynamic case.

1.2. RELATED WORK. Since the pioneering work of von Neumann [1956], the problem of computing with unreliable information has been investigated in a variety of different settings. In particular, two-person searching games in the presence of lies have been the subject of extensive research [Aslam and Dhagat 1991; Borgstrom and Kosaraju 1993; Dhagat et al. 1992; Feige et al. 1994; Kleitman et al. 1980; Muthukrishnan 1994; Pelc 1989, 2002]. In these games an adversary chooses a number in a given range, and the algorithm has to guess this number by asking comparison questions. The adversary is allowed to lie under different constraints. Even when the number of lies can grow proportionally with the number of questions, searching can be performed in optimal time: Borgstrom and Kosaraju [1993], improving over Aslam and Dhagat [1991], Dhagat et al. [1992], and Pelc [1989], designed an $O(\log n)$ searching algorithm. We note that lies are not well suited at modeling destructive memory faults: in the liar model, algorithms may exploit effectively query replication strategies, which cannot be applied in our faulty-memory model. Blum et al. [1994] considered the problem of checking the correctness of data structures operating in unreliable memories. Differently from our work, the aim is to recognize incorrect behavior of nonresilient data structures. In particular, given a data structure residing in a large unreliable memory controlled by an adversary and a sequence of operations that the user has to perform on the data structure, the problem is to design a checker that is able to detect with some positive probability any error in the behavior of the data structure while performing the user's operations. The checker can use only a small amount of safe memory and can report a buggy behavior either immediately after a faulty operation or at the end of the sequence. Memory checkers for random access memories, stacks, and queues have been presented in Blum et al. [1994], where lower bounds of $\Omega(\log n)$ on the amount of reliable memory needed in order to check a data structure of size n are also given. All the data structures considered in Blum et al. [1994] appear to be simpler than search trees, since their structure is independent of the values of the data they contain. The problem of dealing with unsafe pointers has been addressed by Aumann and Bender [1996], providing fault-tolerant versions of stacks, linked lists, and binary search trees: here fault tolerance is defined, given worst-case faults, to be the ratio of the total amount of data lost to the actual amount of data corrupted by the faults. The data structures described in Aumann and Bender [1996] have a small time and space overhead with respect to their nonfault-tolerant counterparts, and guarantee that the amount of data lost upon the occurrence of memory faults is small and independent of the size of the data structure. Once a data structure is discovered to contain faults, it can be reconstructed. With respect to dictionaries, we remark that while in our approach search operations are guaranteed to be always correct on uncorrupted data, this is not the case in Aumann and Bender [1996], where even correct data may be temporarily lost until reconstruction of the data structure.

1.3. OUR RESULTS AND TECHNIQUES. After the design of resilient algorithms for fundamental tasks, such as sorting and searching, it seems quite natural to ask whether we can successfully design resilient data structures, without incurring extra time or space overhead. In many applications such as file system design, it is very important that the implementation of a data structure is resilient to memory faults and provides mechanisms to recover quickly from erroneous states that may lead to an incorrect behavior. The design of resilient data structures appears to be quite a challenging task. Indeed, classical data structures, such as search trees and heap-based priority queues, strongly depend upon structural and positional information: the corruption of a key in a standard binary search tree, for instance, may compromise the search property and guide the search towards wrong subtrees. Moreover, many pointer-based data structures are highly nonresilient: due to the corruption of one single pointer, the entire data structure may become unreachable and even uncorrupted values may be lost.

In this article we make a first, significant step towards the design of resilient data structures in unreliable memories. In particular, we consider the natural resilient version of a classical dictionary, that is, a dictionary where the `insert` and `delete` operations are defined as usual, while the `search` operation must solve the resilient search problem described previously. We present a simple randomized (implementation of a) resilient dictionary, achieving $O(\log n + \delta)$ amortized expected time per operation. We also describe a deterministic resilient dictionary with $O(\log n + \delta^{1+\varepsilon})$ amortized time per operation, where $\varepsilon > 0$ is an arbitrarily small constant. More precisely, our deterministic resilient dictionary supports any sequence of σ operations in $O(\sigma(\log n + \delta) + \delta^{1+\varepsilon})$ time. Hence, for $\sigma \geq \delta^{\varepsilon}$ operations, the deterministic result matches the randomized one. These results are complemented by a lower bound stating that every resilient dictionary based on the comparison of keys, like the ones presented here, must take $\Omega(\log n + \delta)$ expected time per search. All our data structures use optimal $O(n)$ space.

The main idea behind our approach is to group keys into nonoverlapping intervals, and to maintain the intervals in a carefully adapted search tree. After insertions or deletions, intervals might be split, merged, or modified. This is done so as to guarantee the following (high-level) properties:

(1) the intervals span the key-space,

(2) each interval contains $\Theta(\delta)$ keys, and

(3) the set of intervals, and thus the search tree, is modified only every $\Omega(\delta)$ insertions and/or deletions.

Property (2) allows us to search for a key inside an interval in $O(\delta)$ time. More importantly, it allows us to store the search tree reliably, that is, to store $(2\delta + 1)$ copies of each relevant variable (keys excluded), while keeping the space linear. Thanks to property (3), we can update the search tree reliably, with low amortized running times.

Another key ingredient of our method lies in the way we search for an interval containing a given key κ (*interval search*). Note that, by property (1), such an interval must always exist. We design two different interval search algorithms: one with expected running time $O(\log n + \delta)$, and another with worst-case running time $O(\log n + \delta^{1+\varepsilon})$. Our randomized $O(\log n + \delta)$ interval search algorithm relies on a useful lemma which essentially states that every trivially resilient deterministic algorithm (with some additional properties) can be sped up by a factor of $O(\delta)$, by means of a clever use of random sampling.

Our deterministic $O(\log n + \delta^{1+\varepsilon})$ interval search algorithm is based instead on the novel notion of *prejudice number*. Recall that each resilient variable x consists of $(2\delta + 1)$ copies $x_1, x_2, \ldots, x_{2\delta+1}$ of a standard variable. Intuitively, the prejudice number $p \in \{1, 2, \ldots, 2\delta + 1\}$ of an algorithm/data structure is the smallest index of the copies of the resilient variables that we still trust. Initially, $p = 1$ (that is, we consider all the copies reliable). Whenever, during the lifetime of the algorithm/data structure, we discover that the pth copy of some resilient variable is corrupted (detecting which variable is not needed), we consider the pth copies of *every* resilient variable unreliable, and hence increment p by one (we also say that the unreliable copies are *discarded*). The idea is that, if via consistency checks we manage to discover faulty behaviors on the pth copies soon, then we can implement some kind of backtracking to safe checkpoints, and charge the wasted computation time to the faulty copies that are discarded. Note that we might discard some copies of variables that were never involved in the computation so far: their unique demerit is to share the same index with some faulty copy (which motivates the name). From this observation, it might seem that the prejudice number is a rather rough way to synthesize the information about the faults discovered. However, we will see that it is enough for our purposes.

As to the lower bound, a bound of $\Omega(\log n)$ on search operations derives from standard results on (nonresilient) dictionaries. In order to prove the $\Omega(\delta)$ part of the lower bound, we use a rather general construction. Intuitively, the main ingredient that we need is that the output depends on a sufficiently large portion of the unsafe memory. Based on that, we show how to construct a scenario which makes answer incorrectly any $o(\delta)$-time search.

1.4. PRELIMINARIES. We recall that δ is an upper bound on the *total* number of memory faults which may appear during the entire life of the data structure. We also denote by α the *actual* number of faults. Note that $\alpha \leq \delta$.

A *resilient variable* $x = (x_1, x_2, \ldots, x_{2\delta+1})$ consists of $(2\delta + 1)$ copies of a (classical) variable. Assigning a value to x means assigning such value to each copy x_i. The value of x is the majority value of its copies, that is, the value which appears in more than one half of the copies. Such value is well defined since at most δ copies can be corrupted, and hence the remaining $\delta + 1$ copies must have the same value. Trivially, updating x can be done in $O(\delta)$ time and $O(1)$ space: just overwrite each copy of x with the new value. The same holds for reading, using, for example, the algorithm for majority computation in Boyer and Moore [1982]. For the sake of self-containedness, let us describe the reading operation in more detail. Let $x_1, x_2, \ldots, x_{2\delta+1}$ be the copies of x. We store a variable y and a counter z in safe memory (these variables are destroyed at the end of the process). Initially, $y = x_1$ and $z = 1$. For $i = 2, \ldots, 2\delta + 1$, we compare x_i to y. If $x_i = y$, we increment z. Otherwise, we decrement z, and, if $z = 0$, we set $y = x_i$ and $z = 1$. We eventually output y as the majority value.

We assume a comparison-based model for keys, that is, keys can only be compared. All the upper and lower bounds presented here hold in this model. For ease of presentation, we will interpret keys as finite real numbers. We denote by $+\infty$ (respectively, $-\infty$) a value larger (respectively, smaller) than any feasible value for a key. Excluding the keys themselves and the variables in safe memory, all the variables used by our data structure are resilient. However, sometimes we will read only a subset of the copies of a given resilient variable, and compute the majority value over such subset (if no majority value exists, we will return an arbitrary value): in this case, the computed value might be faulty.

1.5. ORGANIZATION. The remainder of this article is organized as follows. The randomized and deterministic dictionaries share the same basic structure, which is described in Section 2. The main difference between the two data structures is the way they implement an interval search: our randomized and deterministic interval search procedures are presented in Section 3 and in Section 4, respectively. Section 5 is devoted to the lower bound. Section 6 contains some concluding remarks and lists some open problems.

2. *Buffering Keys into Intervals*

In this section we describe the common, basic structure of our resilient dictionaries. We maintain a dynamically evolving set of intervals. For each interval we maintain its boundaries, and the list of keys contained in that interval (more details are given shortly). Initially there is a unique interval, of boundaries $(-\infty, +\infty)$, and with an empty list of keys. Throughout the sequence of operations we maintain the following invariants.

(i) The intervals are nonoverlapping, and the union of all the intervals is $(-\infty, +\infty)$.
(ii) Each interval contains less than 2δ keys.
(iii) Each interval contains more than $\delta/2$ keys, except possibly for the leftmost and the rightmost intervals (*boundary* intervals).

To implement any search, insert, or delete of a key κ, we first need to find the interval $I(\kappa)$ containing κ (*interval search*). Invariant (i) guarantees that such interval exists, and that it is unique. Invariants (ii) and (iii) have a crucial role in the amortized analysis of the algorithm, as we will clarify later.

The intervals are stored in a standard balanced search tree. For clarity of exposition, we will restrict our attention to *AVL* trees [Landis 1962]. However, the same basic approach can be easily adapted to other types of search trees (such as red-black trees [Guibas and Sedgewick 1978], B-trees [Bayer and McCreight 1972], and so on). Intervals are ordered according to, say, their left endpoints. For each node v of the search tree, we maintain reliably (i.e., in $(2\delta + 1)$ copies) the following variables:

(1) the endpoints of the corresponding interval $I(v)$ and the number $|I(v)|$ of current keys contained in $I(v)$;
(2) the interval $U(v)$ whose boundaries are the smallest and largest endpoint of the intervals contained in the subtree rooted at v;
(3) the addresses of the left child, the right child, and the parent of v, and all the information needed to keep the search tree balanced with the implementation considered.

Moreover, we maintain *unreliably* (i.e., in a single copy) in an array of size 2δ:

(4) the (unordered) set of current keys contained in $I(v)$.

Figure 1 shows an example of a tree of this type.

The intervals $U(v)$ are crucial in our approach: they will be used to test whether a search is proceeding towards the correct direction in the tree. In particular, the search-key κ must be contained in any $U(v)$ encountered during the search, unless a fault misleads the search. Moreover, given a child v' of v, it must be $U(v') \subset U(v)$. At the beginning of each operation, $U(v)$ is exactly the union of all the intervals contained in the subtree rooted at v (though this property might be temporarily

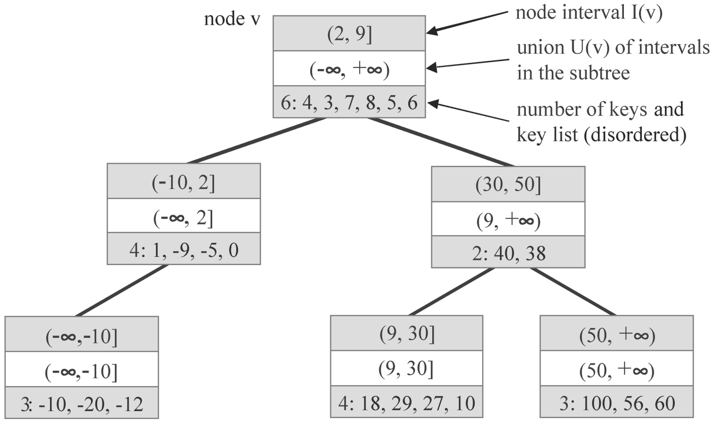

FIG. 1. A resilient search tree.

lost in the middle of key insertions and deletions). Note that in an *AVL* tree we can maintain the intervals $U(v)$ without increasing the asymptotic cost of key insertions and deletions.

The nodes of the search tree are stored into an array. The main reason for this is that it makes it easy to check whether a pointer/index points to a search tree node. Otherwise, the algorithm could jump outside of the search tree by following a corrupted pointer, without even noticing it: this would make the behavior of the algorithm unpredictable. The address of the array and the current number of nodes is kept in safe memory, together with the address of the root node.

We use a standard *doubling* technique [Cormen et al. 2001] to ensure that the size of the array is linear in the current number of nodes. Shortly, when the array is full, we create a new array having twice its size. Vice versa, when at least three quarters of the entries of the array are empty, we create a new array having half its size. In both cases, all nodes are moved to the new array, and the old array is destroyed. The amortized overhead per insertion/deletion of a node due to doubling is $O(1)$ times the cost of inserting/deleting a node, that is, $O(\delta)$ in our case. We will show that each operation which involves the insertion/deletion of a node takes $\Omega(\delta)$ time deterministically, excluding the time spent for doubling. Henceforth, we can charge the amortized cost of doubling to the cost of other operations. In view of this charging argument we will not mention the cost of doubling any further in the analyses.

We next describe how to search, insert, and delete a given key κ.

Search. Every search is performed by first searching the interval $I(\kappa)$ containing κ (*interval search*), and then by linearly searching κ in $I(\kappa)$. The implementation of the interval search is crucial, and will be discussed in Section 3 and in Section 4.

Insert. We initially find $I = I(\kappa)$. Then, if κ is not already in the list of keys associated with I, we add κ to such list. If the size of the list becomes 2δ because of this insertion, we perform the following operations in order to preserve invariant (ii). We delete interval I from the search tree, and we split I into two nonoverlapping subintervals L and R, $L \cup R = I$, which take the smaller and larger half of the keys of I, respectively. In order to split the keys of I in two halves, we use two-way

BubbleSort as described in Finocchi et al. [2006]. This takes time $O(\delta^2)$. Eventually, we insert L and R in the search tree. Both deletion and insertion of intervals from/in the search tree are performed in the standard way (with rotations for balancing), but using resilient variables only, hence in time $O(\delta \log n)$. Note that invariants (i) and (iii) are preserved.

Delete. We first find $I = I(\kappa)$. If we find κ in the list of keys associated with I, we delete κ. Then, if $|I| = \delta/2$ and I is not a boundary interval, we perform, reliably, the following operations in order to preserve invariant (iii). First, we search the interval L to the left of I, and delete both L and I from the search tree. Then we do two different things, depending on the size of L. If $|L| \leq \delta$, we merge L and I into a unique interval $I' = L \cup I$, and insert I' in the search tree. Otherwise ($|L| > \delta$), we create two new nonoverlapping intervals L' and I' such that $L' \cup I' = L \cup I$, L' contains all the keys of L but the $\delta/4$ largest ones, and I' contains the remaining keys of $L \cup I$. Also in this case creating L' and I' takes time $O(\delta^2)$ with two-way BubbleSort. We next insert intervals L' and I' into the search tree. Again, the cost per insertion/deletion of an interval is $O(\delta \log n)$, since we use resilient variables. Observe that invariants (i) and (ii) are preserved.

By invariant (iii), the total number of nodes in the search tree is $O(1+n/\delta)$. Since each node takes $\Theta(\delta)$ space, and thanks to doubling, the space occupied by the search tree is $O(n + \delta)$. This is also an upper bound on the overall space complexity. The space complexity can be reduced to $O(n)$ by storing the variables associated with boundary intervals in the $O(1)$ size safe memory, and by handling the corresponding set of keys via doubling. This change can be done without affecting the running time of the operations and reduces the space usage to $O(n)$.

It remains to describe the implementation of the interval search. As the following lemma shows, this has a crucial impact on the performance of the previously described data structure.

LEMMA 2.1. *Let $S(n, \delta)$ be the worst-case (respectively, expected) time needed to perform an interval search. Then, the worst-case (respectively, expected) amortized time per operation of the aforesaid data structure is $O(S(n, \delta) + \log n + \delta)$. Its space complexity is $O(n)$.*

PROOF. The space complexity is $O(n)$, as discussed before. By invariant (ii), each search operation takes $O(S(n, \delta)+\delta)$ worst-case (respectively, expected) time. The same holds for insert and delete operations, when the structure of the search tree has not to be modified. Otherwise, there is an extra $O(\delta \log n + \delta^2)$ cost. We next show that, by invariants (ii) and (iii), the search tree is modified every $\Omega(\delta)$ insert and/or delete operations. It follows that the amortized cost of insert and delete operations is $O(S(n, \delta) + \log n + \delta)$ (respectively, in expectation).

Initially, the search tree is empty, and there is a unique empty (boundary) interval. This interval is split after at least 2δ insertions. Now consider any newly created interval I. It is sufficient to show that, after the creation of I, $\Omega(\delta)$ insertions or deletions are needed for $|I|$ to reach one of the two critical thresholds $\delta/2$ and 2δ (only the second one when I is a boundary interval). Interval I can be obtained in the following four possible ways:

(1) by splitting in two halves an interval I', $|I'| = 2\delta$. Clearly, $|I| = \delta$.

(2) by adding $\delta/4$ keys to an interval I', $|I'| = \delta/2$. Of course, $|I| = 3\delta/4$.

(3) by removing $\delta/4$ keys from an interval I', $|I'| > \delta$. In this case $3\delta/4 < |I| < 7\delta/4$.
(4) by merging two intervals I' and I'', $|I'| = \delta/2$ and $|I''| \leq \delta$. In this case $|I| \leq 3\delta/2$.

Suppose that $|I|$ reaches the 2δ threshold. From the preceding discussion, this can only happen after at least $\delta/4$ insertions of keys in the interval.

Suppose now that $|I|$ reaches the $\delta/2$ threshold, and I is not a boundary interval. In cases (1), (2), and (3) this happens after at least $\delta/4$ deletions in the interval. In case (4), interval I'' cannot be a boundary interval (otherwise also I would be a boundary interval). Thus $|I''| > \delta/2$ and $|I| > \delta$. Hence in this case the number of deletions must be at least $\delta/2$. The claim follows. □

In Section 3 we will present a simple randomized interval-search procedure with $O(\log n + \delta)$ expected running time. In Section 4 we will describe a deterministic procedure for the same task with worst-case running time $O(\log n + \delta + \alpha\delta^{\varepsilon}) = O(\log n + \delta^{1+\varepsilon})$, where $\varepsilon > 0$ is an arbitrary (small) constant. By Lemma 2.1, this will immediately imply resilient search trees with analogous amortized running times.

3. *Randomized Interval Searching*

In this section we present a simple randomized algorithm to perform an interval search, that is, to find the interval $I(\kappa)$ containing a given search-key κ. Recall that $I(\kappa)$ exists and is unique by invariant (i). Our algorithm takes $O(\log n + \delta)$ expected time. Combining this algorithm with the construction from Section 2, we will obtain a linear-space resilient dictionary with $O(\log n + \delta)$ expected amortized time per operation.

The main insight behind our approach is the following reduction. Consider any (nonresilient) deterministic algorithm $\mathcal{A}'$ to solve some problem, and let $O(t')$ be the running time of $\mathcal{A}'$. We assume that $\mathcal{A}'$ reads each variable at most once. Let us replace each variable of $\mathcal{A}'$ with a resilient variable, and let $\mathcal{A}$ be the resulting trivially resilient algorithm. Observe that the running time of $\mathcal{A}$ is $O(\delta\, t')$. Moreover, let $\mathcal{C}$ be a resilient procedure to check the consistency of the output, and let $O(t'')$ be the running time of $\mathcal{C}$. Then it is possible to obtain a randomized resilient algorithm $\mathcal{R}$ which solves the same problem as $\mathcal{A}$ in expected time $O(t' + t'')$.

Algorithm $\mathcal{R}$ consists of several rounds. In each round, algorithm $\mathcal{R}$ runs algorithm $\mathcal{A}$, but considering only one copy, chosen uniformly at random, of each resilient variable involved. At the end of the round $\mathcal{R}$ checks the correctness of the output produced by means of $\mathcal{C}$. If the check fails, $\mathcal{R}$ starts a new round from scratch. Otherwise, it outputs the computed solution.

LEMMA 3.1. *Algorithm $\mathcal{R}$ described earlier solves the same problem as algorithm $\mathcal{A}$ in expected time $O(t' + t'')$, where $\mathcal{A}$ has a running time $O(t')$ and the consistency of the output can be checked in time $O(t'')$.*

PROOF. The correctness of $\mathcal{R}$ follows immediately from the correctness of $\mathcal{A}$ and $\mathcal{C}$. In fact, $\mathcal{R}$ never halts with a wrong answer. Moreover, it halts if no faulty copy is chosen in a given round, an event which happens with positive probability.

Since each round takes $O(t' + t'')$ time, it remains to show that the expected number of rounds is constant. Consider a given round. Let α_i be the actual number of faults happening at any time on the ith resilient variable considered during the round, $i = 1, 2, \ldots, p = O(t')$. The probability that all the copies chosen during a given round are faithful is at least

$$\left(1 - \frac{\alpha_1}{2\delta + 1}\right)\left(1 - \frac{\alpha_2}{2\delta + 1}\right)\cdots\left(1 - \frac{\alpha_p}{2\delta + 1}\right) \geq \left(1 - \frac{\sum_{i=1}^{p} \alpha_i}{2\delta + 1}\right).$$

Under the same assumption (i.e., all copies chosen being faithful), the behavior of $\mathcal{A}$ and $\mathcal{R}$ in every step of the round is essentially the same. In particular, the two algorithms will consider exactly the same sequence of variables. Recall that by assumption $\mathcal{A}$ never considers the same variable twice. As a consequence, the variables considered by $\mathcal{R}$ are all distinct and hence $\sum_{i=1}^{p} \alpha_i \leq \alpha \leq \delta$. Hence

$$\left(1 - \frac{\sum_{i=1}^{p} \alpha_i}{2\delta + 1}\right) \geq \left(1 - \frac{\delta}{2\delta + 1}\right) \geq \frac{1}{2}.$$

It follows that the expected number of rounds is at most 2. □

We now show how to use Lemma 3.1 in order to solve our interval searching problem.

LEMMA 3.2. *There is an interval searching algorithm of expected running time* $O(\log n + \delta)$.

PROOF. Note that, given any memory address, we can easily check in constant time if that address corresponds to a node of the search tree. Moreover, given any tree node v, we can check reliably in $O(\delta)$ time if $\kappa \in I(v)$. Henceforth, there is a $O(t'') = O(\delta)$ reliable procedure $\mathcal{C}$ to check the consistency of the output of an interval searching algorithm.

There is a trivially resilient interval searching algorithm $\mathcal{A}$ which performs a standard binary search in the search tree described in Section 2, using the reliable variables at each node to guide the search. Note that $\mathcal{A}$ reads each resilient variable at most once, and takes time $O(\delta\, t') = O(\delta \log n)$.

Applying the construction of Lemma 3.1 to $\mathcal{A}$ and $\mathcal{C}$, we obtain the desired interval searching algorithm $\mathcal{R}$, of expected running time $O(t' + t'') = O(\log n + \delta)$. □

THEOREM 3.3. *There is a resilient dictionary taking* $O(n)$ *space and* $O(\log n + \delta)$ *expected amortized time per operation.*

PROOF. It follows immediately from Lemmas 2.1 and 3.2. □

4. *Deterministic Interval Searching*

In this section we present a deterministic interval searching algorithm, whose running time is $O(\log n + \delta + \alpha\delta^{\varepsilon}) = O(\log n + \delta^{1+\varepsilon})$, where $\varepsilon > 0$ is an arbitrary constant parameter. Combining this result with the construction of Section 2, one obtains a resilient dictionary with $O(\log n + \delta^{1+\varepsilon})$ worst-case time per operation.

Like in the randomized case, we start by introducing a simple but useful reduction. Let $\mathcal{A}'$ be a nonresilient deterministic algorithm of running time $O(t')$, let $\mathcal{A}$ be the corresponding trivially resilient algorithm of running time $O(\delta\, t')$, and let $\mathcal{C}$ be a

$O(t'')$-time resilient procedure to check the consistency of the output. Differently from the randomized case, there is no further restriction on $\mathcal{A}'$. In particular, $\mathcal{A}'$ is allowed to read the same variable more than once.

We now describe a resilient deterministic algorithm $\mathcal{D}$ which performs the same task as algorithm $\mathcal{A}$ in $O((1+\alpha)(t'+t''))$ worst-case time, where α is the actual number of faults. Algorithm $\mathcal{D}$ consists of several rounds. In round p, with $p = 1, 2, \ldots, 2\delta+1$, $\mathcal{D}$ runs algorithm $\mathcal{A}$ but considering only the pth copy of each resilient variable involved in the computation. At the end of the round $\mathcal{D}$ checks the consistency of the output by means of $\mathcal{C}$. If the check fails, $\mathcal{D}$ starts a new round (with a larger value of p). Otherwise $\mathcal{D}$ outputs the solution computed in the current round.

LEMMA 4.1. *Algorithm $\mathcal{D}$ described before solves the same problem as algorithm $\mathcal{A}$ in worst-case time $O((1+\alpha)(t'+t''))$, where $\mathcal{A}$ has a running time $O(t')$ and the consistency of the output can be checked in time $O(t'')$.*

PROOF. The correctness of $\mathcal{D}$ follows immediately from the correctness of $\mathcal{A}$ and $\mathcal{C}$. If, in a given round p, all the pth copies considered are faithful, then $\mathcal{D}$ must behave like $\mathcal{A}$ in that round. In particular, $\mathcal{D}$ must halt with the correct output. Otherwise we can charge the $O(t'+t'')$ cost of the round to any pth faulty copy (there must be at least one such faulty copy, since the round failed). Note that each faulty copy can be charged at most once. It follows that there can be at most $(\alpha+1)$ rounds, which gives the claimed time complexity. □

4.1. AN $O(\log n+\delta^2)$ IMPLEMENTATION.

We first present an interval-searching procedure of running time $O(\log n+\delta+\alpha\delta) = O(\log n+\delta^2)$ based on the reduction described before. This will serve as a base for our refined, more sophisticated procedure. Let p be an integer variable kept in safe memory (*prejudice number*). Initially $p = 1$. The search proceeds in rounds. At the beginning of each round, we are given a node v (*checkpoint*) such that $\kappa \in U(v)$ resiliently (i.e., with respect to the majority of the copies of $U(v)$). This implies that $I(\kappa)$ must be contained in the subtree rooted at v. The checkpoint v is kept in safe memory. Initially, v is the root of the search tree, for which $U(v) = (-\infty, +\infty)$. In each round we perform δ classical search steps: starting from a node w, we halt the search or we direct it to the left or right child w' of w, depending on the value of $I(w)$. In each such step we consider the pth copy of each relevant resilient variable (interval endpoints and pointers) only.

At the end of the round we check resiliently whether $\kappa \in U(v') \subset U(v)$, where v' is the final node. If the check fails, we increment p and we restart the round from scratch from v. We do the same if any inconsistency is detected. (For example, a pointer does not point to a node, or $U(w') \not\subset U(w)$ for a child w' of w). Otherwise, we check resiliently whether $\kappa \in I(v')$. If this is the case, we return $I(v')$. Otherwise, we start a new round from v', which becomes the new checkpoint.

Before analyzing the aforesaid procedure, we need some more notation. We define a round:

—*unsuccessful* if an inconsistency is discovered, either during the search or in the final consistency check;

—*apparently successful* if the final consistency check succeeds, $\kappa \notin I(v')$, and the ending node v' is less than $\delta/2$ levels below v in the search tree;

—*really successful* if the final consistency check succeeds, and $\kappa \in I(v')$ or the ending node v' is at least $\delta/2$ levels below v in the search tree.

Note that the algorithm cannot distinguish between really successful and apparently successful rounds. However, this distinction turns out to be useful in the analysis.

LEMMA 4.2. *The interval searching procedure given before has deterministic running time* $O(\log n + \delta + \alpha\delta)$.

PROOF. The correctness of the procedure is trivial. We now discuss the running time.

Each round takes $O(\delta)$ time. We charge the $O(\delta)$ cost of each unsuccessful round to one of the pth faulty copies (there must be at least one such copy), where p is the value of the prejudice number at the time the considered round starts. Like in the proof of Lemma 4.1, each faulty value is charged at most once. Hence the total cost of the unsuccessful rounds is $O(\alpha\,\delta)$.

Consider now an apparently successful round, and let $v = v_1, v_2, \ldots, v_\delta = v'$ be the sequence of (possibly not distinct) nodes visited during the round. Since no inconsistency was noticed, it must be $U(v_1) \supset U(v_2) \supset \ldots \supset U(v_\delta)$, with respect to the pth copies. Hence, by the definition of apparently successful round, at least $\delta/2$ of such copies must be corrupted. Now observe that the same node v_i and the same corrupted copy of $U(v_i)$ can be considered in several apparently successful rounds. In fact, the adversary, by corrupting pointers, may force the algorithm to cycle on the same set of nodes. However, since no inconsistency is detected during the round, the corrupted value of $U(v_i)$ must change in each round (and each change decrements by one the budget of faults of the adversary). As a consequence, there can be at most 2 consecutive apparently successful rounds. We charge the corresponding $O(\delta)$ cost to the next really successful round. Note that such really successful round must exist by the correctness of the algorithm.

The total cost of really successful rounds is $O(\log n + \delta)$. In fact, there can be at most $\lceil 2 \log n/\delta \rceil$ such rounds, each one of cost $O(\delta)$ (including the extra charge coming from apparently successful rounds). Altogether, the running time of the procedure is $O(\log n + \delta + \alpha\,\delta)$. □

4.2. A HIERARCHY OF PREJUDICE NUMBERS. We can reduce the running time by using a hierarchy of prejudice numbers. For the sake of simplicity, let us neglect ceilings and floors. For example, we will consider $\sqrt{\delta}$ as an integer number. We evenly partition (modulo rounding mistakes) the $(2\delta + 1)$ copies of each resilient variables in $\sqrt{\delta}$ *subintervals*. Observe that each interval contains roughly $2\sqrt{\delta} = \Theta(\sqrt{\delta})$ keys. Analogously, we evenly partition each round, consisting of δ search steps, in $\sqrt{\delta}$ *subrounds* of $\Theta(\sqrt{\delta})$ search steps each. We define two prejudice numbers p_0 and p_1, both kept in safe memory: intuitively, p_1 and p_0 indicate, respectively, the first subinterval and the first copy of that subinterval that can still be "trusted." At the end of each subround, we perform a consistency check based on the p_1-th subinterval only. We refer to such checks as $\Theta(\sqrt{\delta})$-checks. Note that the $\Theta(\sqrt{\delta})$-checks are less expensive, but also less reliable: by corrupting $O(\sqrt{\delta})$ values the adversary can make them answer in a wrong way. In any case, if the $\Theta(\sqrt{\delta})$-check fails, we backtrack to the starting node v_1 of the subround (maintained in safe memory). In that case we also increment p_0. When $p_0 \geq \sqrt{\delta}/2$, we increment p_1 and reset p_0 to 1. We backtrack to v_1 and increment p_0 also whenever an inconsistency is detected during any search step of the subround considered. The rest of the algorithm is as before: at the end of each round we perform a (reliable) consistency check on all the $2\delta + 1$ copies ($\Theta(\delta)$-check). If the check fails, we backtrack to the corresponding checkpoint v_2, which is kept in safe memory. We remark that, right before any $\Theta(\delta)$-check, there must be a successful $\Theta(\sqrt{\delta})$-check on the same set of reliable variables (otherwise, the algorithm backtracks before performing the $\Theta(\delta)$-check).

We claim that the aforesaid procedure runs in time $O(\log n + \delta + \alpha\sqrt{\delta})$. In fact, consider a subinterval I. The total cost of backtracking in the subrounds corresponding to I is $O(\sqrt{\delta}) \cdot O(\sqrt{\delta}) = O(\delta)$. Since each discarded subinterval contains at least $\Omega(\sqrt{\delta})$ faulty copies, the total number of subintervals considered is $O(\alpha/\sqrt{\delta}+1)$. Hence the total cost of backtracking in the subrounds is $O(\alpha\sqrt{\delta}+\delta)$. Let us now consider the residual cost, after excluding the cost of backtracking in the subrounds. Each round costs $O(\delta)$ only. Consider any given unsuccessful round, where by unsuccessful we mean that the corresponding $\Theta(\delta)$-check failed (other types of failure are already accounted in the backtracking cost of subrounds). Since we performed the mentioned check, there must be a successful $\Theta(\sqrt{\delta})$-check which answered wrongly. The corresponding subinterval, which contains at least $\Omega(\sqrt{\delta})$ faults, is discarded. From this we can conclude that there are at most $O(\alpha/\sqrt{\delta})$ unsuccessful rounds, of total cost $O(\alpha/\sqrt{\delta}) \cdot O(\delta) = O(\alpha\sqrt{\delta})$. We can bound the cost of the remaining rounds as in Lemma 4.2: we charge the cost of each apparently successful round to the next really successful round. Recall that there cannot be more than two consecutive apparently successful rounds, so each successful round is charged by $O(\delta)$. There can be at most $\lceil 2\log n/\delta \rceil$ really successful rounds, and hence their total cost is $O(\log n + \delta)$. Altogether, the cost of the procedure is $O(\log n + \delta + \alpha\sqrt{\delta})$.

We obtain a better running time by further expanding the hierarchy of prejudice numbers. Let $\varepsilon \leq 1/2$ be any (small) positive constant, and let $k = \lceil 1/\varepsilon \rceil = O(1)$. We evenly partition the $(2\delta+1)$ copies of each resilient variable in $\delta^{1/k}$ subintervals of size $\Theta(\delta^{(k-1)/k})$. We iterate the process on subintervals, until we reach intervals of size $\Theta(\delta^{1/k})$. This way, we generate a hierarchy of $(k-1)$ interval levels, where intervals of level $i, i = 1, 2, \ldots k-1$, have length $\Theta(\delta^{i/k})$. Analogously, we evenly partition each round in $\delta^{1/k}$ subrounds of $\Theta(\delta^{(k-1)/k})$ search steps each, and we iterate the process on subrounds until we reach subrounds of $\Theta(\delta^{1/k})$ search steps. Also in this case we obtain a hierarchy of $(k-1)$ round levels, where there are $\Theta(\delta^{(k-i)/k})$ rounds of level i, $i = 1, 2, \ldots k-1$, each one consisting of $\Theta(\delta^{i/k})$ search steps. Observe also that each round of level $i \geq 2$ is partitioned in $\Theta(\delta^{(i-j)/k})$ rounds of level j, $1 \leq j < i$.

We associate a prejudice number p_i to each interval level $i \in \{1, 2, \ldots, k-1\}$, and we define one extra prejudice number p_0. The interpretation of the prejudice numbers is as before. In particular, the first interval I_{k-1} of level $k-1$ that we still trust is the p_{k-1}-th one. The first interval I_{k-2} of level $k-2$ that we still trust is the p_{k-2}-th subinterval of I_{k-1}. Similarly we can define the first interval I_i of level i that we still trust, $i = 1, 2, \ldots, k-3$. Eventually, the first copy of I_1 that we still trust is the p_0-th one.

At the end of each round of level i we perform a consistency check restricted to the subinterval I_i ($\Theta(\delta^{i/k})$-check). We also maintain a checkpoint node v_i for each i: we backtrack from the current node in the search to v_i whenever a $\Theta(\delta^{i/k})$-check fails. In that case we increment p_{i-1} as well. As soon as $p_{i-1} \geq \delta^{1/k}/2$, we *discard* interval I_i, that is, we increment p_i and reset p_j to 1 for every $j < i$. We backtrack to v_1 and increment p_0 also whenever we find an inconsistency during a search step. The rest of the algorithm is as before. In particular, every δ search steps we perform a reliable consistency check ($\Theta(\delta)$-check), and we backtrack to the corresponding checkpoint v_k if the check fails. All the checkpoints v_i and all the prejudice numbers p_i are kept in safe memory. Note that this is possible since they are constantly many. We remark that each $\Theta(\delta^{i/k})$-check, $2 \leq i \leq k$, follows a successful $\Theta(\delta^{(i-1)/k})$-check on the same set of reliable variables.

LEMMA 4.3. *For any fixed $\varepsilon > 0$, the interval searching procedure presented earlier has deterministic running time $O(\log n + \delta + \alpha\delta^{\varepsilon})$.*

PROOF. Consider the cost of backtracking at rounds of level 1. Each backtrack costs $O(\delta^{1/k})$, and there are at most $\delta^{1/k}/2$ backtracks per interval. Hence the cost of backtracking at a given interval I_1 of level 1 is $O(\delta^{2/k})$. Each discarded interval I_1 of level 1 contains at least $\Omega(\delta^{1/k})$ faulty copies. Hence we can consider at most $O(\alpha/\delta^{1/k} + 1)$ intervals of level 1, with a total backtracking cost of $O(\alpha\delta^{1/k} + \delta^{2/k})$.

Consider now the cost of backtracking at subrounds of level i, $i = 2, 3, \ldots, k-1$, excluding the cost of backtracking at lower levels. Each backtrack costs $O(i\,\delta^{i/k})$. In fact, we have to take into account $\Theta(\delta^{i/k})$ search steps, one $\Theta(\delta^{i/k})$-check, and all the corresponding successful checks of level $j < i$ (which are not accounted in the backtracking cost of lower-level rounds). For each j, there are $\Theta(\delta^{(i-j)/k})$ many $\Theta(\delta^{j/k})$-checks of this type. Hence a total cost of $O(\delta^{i/k})+\sum_{j=1}^{i-1} O(\delta^{(i-j)/k}) \cdot O(\delta^{j/k}) = O(i\,\delta^{i/k})$.

Now consider any interval I_i of level i. We can backtrack at I_i at most $\delta^{1/k}/2$ times, with a total cost of $O(i\,\delta^{i/k}) \cdot O(\delta^{1/k}) = O(i\,\delta^{(i+1)/k})$. Suppose that I_i was discarded. If one of the $\Theta(\delta^{i/k})$-checks on I_i was faulty, there must be $\Omega(\delta^{i/k})$ faulty copies on I_i. Otherwise, a consistency check on each one of the first $\delta^{1/k}/2$ subintervals of I_i of level $(i-1)$ must be faulty. Hence also in this case I_i must contain $\Omega(\delta^{(i-1)/k}) \cdot \Omega(\delta^{1/k}) = \Omega(\delta^{i/k})$ faulty values. We can then conclude that the total number of subintervals of level i considered by the algorithm is $O(\alpha/\delta^{i/k}+1)$. Hence the total cost of backtracking at subrounds of level i is $O(\alpha/\delta^{i/k} + 1) \cdot O(i\,\delta^{(i+1)/k}) = O(\alpha\, i\,\delta^{1/k} + i\,\delta^{(i+1)/k})$.

Altogether, the total cost of backtracking in subrounds is $\sum_{i=1}^{k-1} O(i\,\alpha\,\delta^{1/k} + i\,\delta^{(i+1)/k}) = O(k^2(\alpha\,\delta^{1/k} + \delta)) = O(\alpha\,\delta^{\varepsilon} + \delta)$. Let us now consider the residual cost, after excluding the cost of backtracking in the subrounds. By the same type of argument as befor, each round costs $O(\delta) + \sum_{j=1}^{k-1} O(\delta^{(k-j)/k}) \cdot O(\delta^{j/k}) = O(k\,\delta)$. Consider any given unsuccessful round, where by unsuccessful we mean that the corresponding $\Theta(\delta)$-check failed. Since we performed the latter check (which is reliable), there must be a successful $\Theta(\delta^{(k-1)/k})$-check which answered wrongly. The corresponding interval of level $k-1$ must then contain at least $\Omega(\delta^{(k-1)/k})$ faulty copies, which are discarded. From this we can conclude that there are at most $O(\alpha/\delta^{(k-1)/k})$ unsuccessful rounds, of total cost $O(\alpha/\delta^{(k-1)/k}) \cdot$

$O(k\,\delta) = O(k\,\alpha\,\delta^{1/k}) = O(\alpha\,\delta^{\varepsilon})$. We can bound the cost of the remaining rounds as in Lemma 4.2: we charge the cost of apparently successful rounds, no more than two consecutively, on the next really successful rounds. There can be at most $\lceil 2\log n/\delta \rceil$ really successful rounds, and hence their total cost is $O(\log n + \delta)$. The claimed running time follows. □

THEOREM 4.4. *For any constant $\varepsilon > 0$, there is a resilient dictionary taking $O(n)$ space and $O(\log n + \delta + \alpha\,\delta^{\varepsilon}) = O(\log n + \delta^{1+\varepsilon})$ amortized time per operation.*

PROOF. It follows from Lemmas 2.1 and 4.3. □

We can get a better result over a sequence of operations.

COROLLARY 4.5. *For any constant $\varepsilon > 0$, there is a resilient dictionary taking $O(n)$ space and $O(\sigma(\log n + \delta) + \delta^{1+\varepsilon})$ time over any sequence of σ operations.*

PROOF. Consider the proof of Lemma 4.3. The term $O(\alpha\,\delta^{\varepsilon})$ in the time complexity can be amortized over any sequence of interval searches: it is sufficient to maintain the same prejudice numbers from one search to the next one. The remaining terms in the time complexity contribute with $O(\log n + \delta)$ time per operation. Hence a sequence of σ interval searches costs $\sigma\,S(n,\ \delta) + O(\alpha\,\delta^{\varepsilon}) = O(\sigma(\log n + \delta) + \alpha\,\delta^{\varepsilon})$. From the proof of Lemma 2.1, any sequence of σ operations costs $O(\sigma(\log n + \delta))$, excluding the cost of interval searches. The claim follows. □

This implies that for $\sigma \geq \delta^{\varepsilon}$ operations, the amortized time per operation is $O(\log n + \delta)$. Hence, under that assumption, the deterministic result of this section matches the randomized one of Section 3.

5. *A Lower Bound*

In this section we show that, in the comparison-based model, every resilient dictionary takes $\Omega(\log n + \delta)$ expected time per search, for $\delta = O(n)$. Note that, for $n = o(\delta)$, there is a trivial resilient dictionary with running time $O(n) = o(\delta)$: it is sufficient to keep the keys into an array, whose address and length are stored in safe memory.

THEOREM 5.1. *For $\delta = O(n)$, every comparison-based resilient dictionary takes $\Omega(\log n + \delta)$ expected time per search.*

PROOF. Every comparison-based dictionary, even in a system without memory faults, takes $\Omega(\log n)$ expected time per search. This lower bound extends immediately to the case of resilient dictionaries, since any resilient procedure can be simulated on a safe memory system. Hence, without loss of generality, let us assume that $\log n = o(\delta)$. Under this assumption, it is sufficient to show that the time required by a resilient search operation is $\Omega(\delta)$.

Let *RD* be any given resilient dictionary. Consider first the case that *RD* is deterministic, and assume by contradiction that RD performs any search operation in $o(\delta)$ worst-case time. We restrict our attention to the following scenario. We insert n distinct keys. No fault occurs during these insertions. Then we perform a search operation with search key κ. We next show that there is a choice of κ which makes RD answer incorrectly, which is a contradiction.

Let $\mathcal{S}$, $|\mathcal{S}| = O(1)$, be the keys in safe memory, and $\mathcal{U}$, $|\mathcal{U}| = \Theta(n)$, the keys stored in unsafe memory only. Assume that the search key κ belongs to $\mathcal{U}$. Consider the first memory location a_1 in unsafe memory read by *RD* (a_1 can be interpreted as a physical address). The keys of $\mathcal{S}$ partition the keys of $\mathcal{U}$ in (at most) $|\mathcal{S}| + 1$ equivalence classes under comparisons with keys of $\mathcal{S}$. It follows that *RD* can generate at most $|\mathcal{S}|+1 = O(1)$ different values for a_1, one for each class. Assume from now on that κ belongs to the largest such class $\mathcal{U}'$. Note that $|\mathcal{U}'| = \Theta(n)$. Before RD reads a_1, the adversary sets the content of a_1 to same fixed, arbitrary value different from any key, say zero. The adversary does the same for the following unsafe memory locations $a_2, a_3, \ldots, a_{\delta'}$, $\delta' = o(\delta)$, read by RD. (Equivalently, the adversary might set all the a_i's to zero at the beginning of the search operation). At the end of the process, RD either outputs yes and returns the address of a memory location containing a key of value κ, or returns no.

Observe that the output of the algorithm is exactly the same for every $\kappa \in \mathcal{U}'$. Furthermore, if $\kappa \in \mathcal{U}'$ is not one of the keys initially stored in the locations a_i, $i = 1, 2, \ldots, \delta'$, then κ is faithful (in fact, only the a_i's are corrupted). Therefore, RD is forced to output yes and to return some memory location a', of value κ. This is a contradiction since we can choose two distinct keys κ' and κ'' in $\mathcal{U}'$ satisfying the properties previously described (here we use the assumption $\delta = O(n)$, and hence $|\mathcal{U}'| - \delta' = \Theta(n) \geq 2$ for n large enough).

The randomized case is analogous. Now we assume by contradiction that some resilient dictionary RD performs any search operation in $o(\delta)$ expected time. Then we can use the same construction as before, the main difference being that the sequence $a_1, a_2, \ldots, a_{\delta'}$ is random. However, the adversary, by simulating the search procedure of RD, can generate one such sequence, which shows up with positive probability, and such that $\delta' = o(\delta)$. By setting the corresponding memory locations to zero at the beginning of the search, the adversary makes RD fail with positive probability for a proper choice of κ. This contradicts the correctness of RD. □

We remark that the idea behind the proof of Theorem 5.1 can be adapted to obtain $\Omega(\delta)$ lower bounds for a wide family of resilient algorithms and data structures. Intuitively, the main ingredient that we need is that the output is influenced by the state of the unsafe memory. (Since the safe memory has size $O(1)$, this seems a rather general property). This rules out $o(\delta)$-time procedures, which might not be able to read any uncorrupted memory location in unsafe memory.

6. *Conclusions*

In this article we made a first step towards the design of resilient data structures in unreliable memories. In particular, we have focused on resilient dictionaries, that is, dictionaries where the insert and delete operations are defined as usual, while the search operation must solve the (natural) resilient search problem defined in Finocchi and Italiano [2008]. We have presented both a randomized and a deterministic implementation that achieve $O(\log n + \delta)$ and $O(\log n + \delta^{1+\varepsilon})$ amortized time per operation, respectively, where $\varepsilon > 0$ is an arbitrarily small constant. Our resilient dictionaries use linear space. These results are complemented by a lower bound stating that every comparison-based resilient dictionary must take $\Omega(\log n + \delta)$ expected time per search.

After this work, optimal resilient dictionaries and resilient priority queues were presented in Brodal et al. [2007] and Jørgensen et al. [2007], respectively. The dictionary designed in Brodal et al. [2007] requires linear space, and supports each operation in $O(\log n + \delta)$ amortized worst-case time. With respect to this article, this dictionary is based on rather different techniques. A resilient priority queue maintains a set of elements under the operations insert and deletemin: insert adds an element and deletemin deletes and returns either the minimum uncorrupted value or a corrupted value. Jørgensen et al. [2007] show how to support both insert and deletemin operations in $O(\log n + \delta)$ amortized time per operation. Thus, their priority queue matches the performance of classical optimal priority queues in the RAM model when the number of corruptions tolerated is $O(\log n)$. An essentially matching lower bound is also proved in Jørgensen et al. [2007]: a resilient priority queue containing n elements, with $n > \delta$, must use $\Omega(\log n + \delta)$ comparisons to answer an insert followed by a deletemin.

All the results presented in this article, as well as in Finocchi et al. [2006, 2007] and Finocchi and Italiano [2008], crucially rely on the knowledge of the maximum number δ of memory faults that can affect the algorithm/data structure. Designing δ-*oblivious* algorithms and data structures is a challenging open problem. An experimental study of the performances of our resilient search trees would also be a valuable contribution: the design of resilient algorithms and data structures appears to be important especially when processing massive datasets in large inexpensive memories, and therefore producing practical implementations would be very important for their concrete deployment in real applications.

ACKNOWLEDGMENTS. We are indebted to the anonymous referees for many useful comments.

REFERENCES

ASLAM, J. A., AND DHAGAT, A. 1991. Searching in the presence of linearly bounded errors. In *Proceedings of the 23rd ACM Symposium on Theory of Computing (STOC'91)*. ACM, 486–493.

AUMANN, Y., AND BENDER, M. A. 1996. Fault-Tolerant data structures. In *Proceedings of the 37th IEEE Symposium on Foundations of Computer Science (FOCS'96)*. IEEE, 580–589.

BAYER, R., AND MCCREIGHT, E. 1972. Organization and maintenance of large ordered indexes. *Acta Inf. 1*, 173–189.

BLUM, M., EVANS, W., GEMMELL, P., KANNAN, S., AND NAOR, M. 1994. Checking the correctness of memories. *Algorithmica 12*, 225–244.

BORGSTROM, R. S., AND KOSARAJU, S. R. 1993. Comparison based search in the presence of errors. In *Proceedings of the 25th ACM Symposium on Theory of Computing (STOC'93)*. ACM, 130–136.

BOYER, R., AND MOORE, S. 1982. Mjrty—A fast majority vote algorithm. Tech. rep., University of Texas.

BRODAL, G. S., FAGERBERG, R., FINOCCHI, I., GRANDONI, F., ITALIANO, G. F., JØRGENSEN, A. G., MORUZ, G., AND MØLHAVE, T. 2007. Optimal resilient dynamic dictionaries. In *Proceedings of the 15th Annual European Symposium on Algorithms (ESA'07)*. Lecture Notes in Computer Science, vol. 4698. Springer, 347–358.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2001. *Introduction to Algorithms* 2nd Ed. The MIT Press/McGraw-Hill Book Company.

DHAGAT, A., GACS, P., AND WINKLER, P. 1992. On playing "twenty questions" with a liar. In *Proceedings of the 3rd ACM-SIAM Symposium on Discrete Algorithms (SODA'92)*. SIAM, 16–22.

FEIGE, U., RAGHAVAN, P., PELEG, D., AND UPFAL, E. 1994. Computing with noisy information. *SIAM J. Comput. 23*, 1001–1018.

FINOCCHI, I., GRANDONI, F., AND ITALIANO, G. F. 2006. Optimal sorting and searching in the presence of memory faults. In *Proceedings of the 33rd International Colloquium on Automata, Language and Programming (ICALP'06)*. Lecture Notes in Computer Science. Springer, 286–298.

FINOCCHI, I., GRANDONI, F., AND ITALIANO, G. F. 2007. Resilient search trees. In *Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms (SODA'07)*. SIAM, 547–555.

FINOCCHI, I., AND ITALIANO, G. F. 2008. Sorting and searching in faulty memories. *Algorithmica 52,* 3, 309–332.

GUIBAS, L. J., AND SEDGEWICK, R. 1978. A dichromatic framework for balanced trees. In *Proceedings of the 19th IEEE Symposium on Foundations of Computer Science*. IEEE, 8–21.

HAMDIOUI, S., AL-ARS, Z., DE GOOR, J. V., AND RODGERS, M. 2003. Dynamic faults in random-access-memories: Concept, faults models and tests. *J. Electron. Test. Theory Appl. 19*, 195–205.

HENZINGER, M. R. 2004. The past, present and future of Web search engines (invited talk). In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP'04)*. Lecture Notes in Computer Science. Springer.

HENZINGER, M. R. 2007. Combinatorial algorithms for Web search engines—Three success stories (invited talk). In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA'07)*. SIAM.

JØRGENSEN, A. G., MORUZ, G., AND MØLHAVE, T. 2007. Priority queues resilient to memory faults. In *Proceedings of the 10th Workshop on Algorithms and Data Structures (WADS'07)*. Lecture Notes in Computer Science. Springer, 127–138.

KLEITMAN, D. J., MEYER, A. R., RIVEST, R. L., SPENCER, J., AND WINKLMANN, K. 1980. Coping with errors in binary search procedures. *J. Comput. Syst. Sci. 20*, 396–404.

LANDIS, G. M. A.-V. Y. M. 1962. An algorithm for the organization of information. *Dokl. Akad. Nauk. Sssr 146*, 263–266.

MAY, T. C., AND WOODS, M. H. 1979. Alpha-Particle-Induced soft errors in dynamic memories. *IEEE Trans. Electron. Dev. 26*, 1, 2–9.

MUTHUKRISHNAN, S. 1994. On optimal strategies for searching in the presence of errors. In *Proceedings of the 5th ACM-SIAM Symposium on Discrete Algorithms (SODA'94)*. SIAM, 680–689.

PELC, A. 1989. Searching with known error probability. *Theor. Comput. Sci. 63*, 2, 185–202.

PELC, A. 2002. Searching games with errors—Fifty years of coping with liars. *Theor. Comput. Sci. 270,* 1-2, 71–109.

VON NEUMANN, J. 1956. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In *Automata Studies*, C. Shannon and J. McCarty, Eds. Princeton University Press, 43–98.

RECEIVED MAY 2007; ACCEPTED NOVEMBER 2008