Resilient Search Trees[*]

Irene Finocchi [†] Fabrizio Grandoni [†] Giuseppe F. Italiano [‡]

Abstract

We investigate the problem of computing in a reliable fashion in the presence of faults that may arbitrarily corrupt memory locations. In this framework, we focus on the design of resilient data structures, i.e., data structures that, despite the corruption of some memory values during their lifetime, are nevertheless able to operate correctly (at least) on the set of uncorrupted values. In particular, we present resilient search trees which achieve optimal time and space bounds while tolerating up to $O(\sqrt{\log n})$ memory faults, where n is the current number of items in the search tree. In more detail, our resilient search trees are able to insert, delete and search for a key in $O(\log n + \delta^2)$ amortized time, where δ is an upper bound on the total number of faults. The space required is $O(n + \delta)$.

1 Introduction

Memories in modern computing platforms are not always fully reliable, and sometimes the content of a memory unit may be lost or corrupted, due to hardware or power failures, cosmic radiations, or malicious attacks. In fault-based cryptanalysis, for instance, some attacks [3, 20] work by manipulating the non-volatile memories of cryptographic devices, in order to induce faults that force the devices to output wrong ciphertexts, which may allow the attacker to break the secret keys used during the encryption. Also applications that make use of large memory capacities at low cost have to deal with problems of memory faults and reliable computation. As an example, consider Web search engines, which need to store and process huge data sets (of the order of Terabytes): for large data structures of this kind, even a small failure probability can result in few bit flips, that may become responsible for erroneous answers to Web searches [9, 13].

Destructive memory errors may be responsible for unpredictable behaviors in classical algorithms and data structures. For instance, if we want to search for a key in a sorted sequence subject to memory faults, corrupted keys may lead the search in the wrong direction. In the design of reliable systems, when specific hardware for fault detection and correction is not available or is too expensive, it makes sense to look for a solution to these problems at the application level, i.e., to design algorithms and data structures that are able to perform the tasks they were designed for, even in the presence of unreliable or corrupted information. Informally, we say that we have a *memory fault* when the correct value that should be stored in a memory location gets altered because of a failure. We say that an algorithm or a data structure is *resilient* to memory faults if, despite the corruption of some memory values during its lifetime, it is nevertheless able to produce a correct output (at least) on the set of uncorrupted values.

In previous work [11], we introduced a faulty-memory random access machine, whose memory locations may suffer from faults: we model faults with an adaptive adversary, which may corrupt up to δ memory words throughout the execution of an algorithm or during the lifetime of a data structure. In this model, we cannot distinguish corrupted values from correct ones and we can exploit only $O(1)$ *safe* memory words, whose content never gets corrupted. In [11, 12] we presented matching upper and lower bounds for resilient sorting and searching. In particular, we designed an $O(n \log n)$ time sorting algorithm that can optimally tolerate up to $O(\sqrt{n \log n})$ memory faults. With respect to searching, we proved that any $O(\log n)$ time (even randomized) searching algorithm can tolerate at most $O(\log n)$ memory faults, and we provided an optimal randomized algorithm. We also designed an almost optimal deterministic algorithm that can tolerate up to $O((\log n)^{1-\epsilon})$ memory faults, for any small positive constant ϵ. Note that these results hold in a static setting, i.e., the set of keys cannot be updated by adding new keys or deleting existing keys.

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 in 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

[*]This work has been partially supported by the Sixth Framework Programme of the EU under Contract Number 507613 (Network of Excellence "EuroNGI: Designing and Engineering of the Next Generation Internet") and by MIUR, the Italian Ministry of Education, University and Research, under Project ALGO-NEXT ("Algorithms for the Next Generation Internet and Web: Methodologies, Design and Experiments").

[†]Dipartimento di Informatica, Università di Roma "La Sapienza", via Salaria 113, 00198, Roma, Italy. Email: {finocchi,grandoni}@di.uniroma1.it.

[‡]Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", via del Politecnico 1, 00133 Roma, Italy. Email: italiano@disp.uniroma2.it.

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 search tree, for instance, may compromise the search property and guide search operations towards wrong directions. Moreover, many pointer-based data structures are highly non-resilient: due to the corruption of one single pointer, the entire data structure may become unreachable and even uncorrupted values may be lost. A natural approach to prevent this would be to use data replication: however, if each piece of data were replicated $\Theta(\delta)$ times, where δ is an upper bound on the total number of faults, one would typically obtain a multiplicative overhead of $\Theta(\delta)$ in terms of both space and running times. A search tree of this kind, would require $O(\delta n)$ space and $O(\delta \log n)$ search and update time, and thus could tolerate only $O(1)$ memory faults while maintaining optimal time and space bounds. In general, using full data replication can be very inefficient when the objects to be maintained are large and complex, such as large database records or long strings: copying such objects can indeed be very costly, and in some cases we might not even know how to do it. For instance, common libraries of algorithmic codes (e.g., the LEDA Library of Efficient Data Types and Algorithms [15] or the Standard Template Library [19]) typically implement data types by means of indirect addressing methods: the objects to be maintained are accessed through pointers, which are moved around in memory instead of the objects themselves, and the algorithm relies on user-defined comparator functions. In these cases, the implementation of the data structure assumes neither the existence of *ad hoc* functions for data replication nor the definition of suitable encoding mechanisms to maintain a reasonable storage cost.

Our Results. In this paper we make a first, significant step towards the design of resilient data structures in unreliable memories. In particular, we present a resilient version of search trees, which implements the classical dictionary operations: *search*, *insert*, and *delete*. In our data structure, searching, inserting or deleting a key can be implemented in $O(\log n + \delta^2)$ amortized time, where n is the number of keys and δ is an upper bound on the total number of faults. The space required is $O(n+\delta)$. This implies that our resilient search trees can tolerate up to $O(\sqrt{\log n}\,)$ memory faults while still achieving optimal time and space bounds.

The main idea behind our approach is to group keys into non-overlapping intervals which span the key-space, and to maintain the intervals in a carefully adapted binary search tree. After insertions or deletions, intervals might be split, merged or modified. This is done in such a way that: (1) each interval contains $\Theta(\delta)$ keys, and (2) the set of intervals, and thus the search tree, is modified only every $\Omega(\delta)$ insertions and/or deletions. The first property allows us to search for a key inside an interval in $O(\delta)$ time. More important, it allows us to store the search tree reliably, while keeping the space linear. In fact, there are $O(n/\delta)$ intervals/nodes only, and each one can afford to use extra space $O(\delta)$ to ensure reliable computations. Thanks to the second property, we can update the search tree reliably (pointers included), 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. To this aim, we generalize the high-level approach used for resilient searching in a sorted array [11]. We remark that its application here is more complicated, since we deal with a more complex data structure (with pointers). The idea is that the search algorithm must be fast on non-faulty instances, and thus it is not possible to perform reliable computations too often. On the contrary, most of the times we need to trust unreliable variables, and only every $\Theta(\delta)$ search steps we can afford to check reliably whether the computation is going towards the right direction. The search algorithm keeps track of these reliable checks by means of checkpoints: if we realize that something wrong happened, we backtrack to the last checkpoint, which is stored in safe memory. The efficiency of the algorithm hinges on the fact that searches towards wrong directions, which do not produce useful results, can be charged to faulty values.

Related Work. Since the pioneering work of von Neumann in the late 50's [21], 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 [1, 5, 8, 10, 14, 16, 17, 18]. 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 done to optimality: Borgstrom and Kosaraju [5], improving over [1, 8, 17], designed an $O(\log n)$ searching algorithm. We note that lies are not well suited at modelling destructive memory faults: in the liar model, algorithms may exploit effectively query replication strategies, which make no sense in our faulty-memory model.

Blum et al. [4] 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 non-resilient 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 [4], 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 [4] 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 in [2], 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 [2] have a small time and space overhead with respect to their non-fault-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 search trees, we remark that, while in our approach search operations are guaranteed to be always correct on uncorrupted data, this is not the case in [2], where even correct data may be temporarily lost until reconstruction of the data structure.

Preliminaries. Without loss of generality, we assume that the keys are distinct, finite real numbers. We denote by n the current number of keys. When searching for a key κ, we either find a key equal to κ or are able to determine that there is no correct key equal to κ. Note that this is the best we can hope for, because memory faults can make κ appear or disappear in the search tree at any time. Insert and delete operations are defined in the natural way with respect to the search above.

We recall that δ is an upper bound on the *total* number of memory faults which may appear during the entire life of the search tree. We also denote by α the *actual* number of faults. Note that $\alpha \leq \delta$. To simplify the expression of the running times, throughout this paper we assume that $\alpha > 0$, i.e., there is at least one fault.[1]

A *reliable variable* X consists of $(2\delta + 1)$ copies of a (classical) variable. The *value* of a reliable variable X is defined as the majority value of its copies (a majority value must exist since at most δ copies can be corrupted). Assigning a value to X means assigning such value to all the copies of X. Note that both reading and updating X can be done in $O(\delta)$ time and $O(1)$ space (using, e.g., the algorithm for majority computation in [6]).

Excluding the keys themselves, all the variables used by our data structure are reliable. However, sometimes we will read only a subset of the copies of a given reliable variable, and compute the majority value over such subset (if no majority value exists, we will return an arbitrary value): in that case the computed value might be faulty.

2 The Algorithm: Buffering Keys into Intervals

In this section we describe a resilient search tree, whose running time will be analyzed in Section 3. Recall that the keys are distinct, finite real numbers.

Data Structure. As mentioned before, we maintain a dynamically evolving set of non-overlapping intervals of the kind

$$(-\infty, a_1],\ (a_1, a_2], \ldots, (a_{h-2}, a_{h-1}],\ (a_{h-1}, +\infty).$$

We assume that initially, when the search tree is empty, there is a unique interval $(-\infty, +\infty)$. Throughout the sequence of operations we maintain the following invariants:

(i) 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 for possibly the leftmost and rightmost intervals (*boundary* intervals).

To implement any search, insert, or delete of a key κ, we first need to find the interval I containing κ. Invariant (i) guarantees that such interval exists.

The intervals are maintained into a standard balanced binary search tree. Throughout the paper we use as a reference implementation an AVL tree [7]. However, the same basic approach also works with other search trees. Intervals are ordered according to, say, their left endpoints. For each node v of the search tree, we store *reliably*:

1. the endpoints of the corresponding interval $I(v)$;
2. the number $|I(v)|$ of keys contained in the interval $I(v)$;
3. the interval $U(v)$ delimited by the smallest and largest endpoints of the intervals contained in the subtree rooted at v;
4. the addresses of the left child, the right child, and the parent of v;

[1] To obtain the running times in the case $\alpha = 0$, it is sufficient to add a term $O(\delta)$.

5. all the information needed to keep the search tree balanced with the implementation considered.

Moreover, we maintain *unreliably*

6. the (unordered) set of keys contained in the corresponding interval $I(v)$ so that it is possible to search, insert or delete a key in time $O(\delta)$.

The $U(v)$'s are crucial in our approach: they will be used to test whether a search is proceeding towards the right subtree. 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 lost in the middle of key insertions and deletions). We can maintain the $U(v)$'s 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 choice is that it makes 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. We use a standard *doubling* technique to ensure that the size of the array is linear in the current number of nodes. The amortized overhead per insertion/deletion of a node is $O(\delta)$.

It remains to describe how to search, insert, and delete a given key κ.

Search. We first find the interval I containing κ, which must exist by Invariant (i). Once I is available, we search κ in the (unordered) set of keys associated to I.

The interval search proceeds in *rounds*. At the beginning of each round we are given a checkpoint node v, which initially is the search tree root, and such that $\kappa \in U(v)$. We keep the index of the current checkpoint and its (reliable) value in safe memory. At the end of the round we find a node v' such that either $\kappa \in I(v')$ or $\kappa \in U(v') \subset U(v)$. Node v' becomes the new checkpoint for the next round.

Each round consists of at most two phases: a first *unreliable phase*, of cost $O(\delta)$, and possibly a second *reliable phase* of cost $O(\delta^2)$. In both phases we perform up to δ classical search steps. Let w be the current node. At the beginning of each phase $w = v$ and at the end $w = v'$. If $\kappa \in I(w)$ we end the phase. Otherwise, depending on the value of κ and of $I(w)$, we proceed the search on the left or right child w' of w, updating w accordingly. We keep in safe memory the information associated with both w and w'. When the search proceeds to a child w'' of w', we evict the information associated with w from safe memory, and load the information associated with w''. This way, when we perform the two search steps from w to w' and from w' to w'' the information associated with w' remains in safe memory.

The differences between the two phases are as follows. In the first (unreliable) phase we work only with the first copy of each variable (index or endpoint), while in the second (reliable) phase we consider all the $(2\delta+1)$ copies, and work with their majority value. We end the first phase, and start the second phase from the initial node v, whenever we find any *inconsistency* in a search step: e.g., $\kappa \notin U(w)$, or $U(w') \not\subset U(w)$, or we find an infeasible index.

The consistency checks performed during the first phase involve the first copy of each variable only. Thus, such checks do not guarantee the reliability of the search. For this reason, at the end of the first phase, we check reliably (in $O(\delta)$ time) whether either $\kappa \in I(v')$ or $\kappa \in U(v') \subset U(v)$. In the former case ($\kappa \in I(v')$), we found the desired interval, and the algorithm halts. In the latter case ($\kappa \in U(v') \subset U(v)$), we skip the second phase of the current round, and start a new round from v'. If none of the two conditions holds, a fault (not recognized by the consistency checks) occurred, and we start the second phase from the initial node v.

At the end of the second phase we must end up in a node v' such that, reliably, either $\kappa \in I(v')$ or $\kappa \in U(v') \subset U(v)$: in the former case we are done, otherwise we start a new round from v'.

Insert. We initially search the interval I containing κ (which must exist) with the procedure described above. Then, if κ is not already in the list of keys associated to 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 non-overlapping 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 [12]. 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 reliable variables only, hence in time $O(\delta \log n)$. Note that Invariants (i) and (iii) are preserved.

Delete. We first search the interval I containing κ (which must exist). If we find κ in the list associated to 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 non-overlapping 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 reliable variables. Observe that Invariants (i) and (ii) are preserved.

3 Analysis

In this section we analyze the time bounds of the resilient search tree described in Section 2. In Section 4 we will describe how to reduce the amortized time complexity via a refined search procedure.

Lemma 3.1. *The space complexity of the resilient search tree described in Section 2 is $O(n + \delta)$.*

Proof. Each node of the search tree requires space $O(\delta)$. By Invariant (iii) there can be at most $(2n/\delta + 2)$ intervals, which is also an upper bound on the number of nodes in the search tree. The claim follows.

Let v and v' be the initial and final nodes of a given round, and let κ be the key to be searched for. We define a round to be *successful* if we find v' such that either (1) $\kappa \in I(v')$, or (2) $\kappa \in U(v') \subset U(v)$ and the depth of v' in the search tree is at least the depth of v plus δ (the inequality holds if the adversary introduces faults that "help" the search). We define a round to be *unsuccessful* otherwise.

Lemma 3.2. *In the resilient search tree of Section 2, each search has amortized cost $O(\log n + \alpha\, \delta^2)$.*

Proof. Let v and v' be the initial and final nodes of a given round, and let κ the key to be searched for. We distinguish three types of rounds, and analyze their running time separately. Note that rounds that end with the second phase are necessarily successful, since only reliable variables are used in the second phase.

(a) Successful rounds which end with the first phase. The total cost of such rounds is trivially $O(\log n + \delta)$.

(b) Successful rounds which end with the second phase. Consider any such round. Its running time is $O(\delta^2)$. Since the first phase failed, there must be a descendant v^* of v, at distance smaller than δ from v, which contains a faulty value. We charge the $O(\delta^2)$ cost to such faulty value. Since no faulty value can be charged more than once during the same search, the total contribution of this type of rounds to the search is $O(\alpha\, \delta^2)$.

(c) Unsuccessful rounds which end with the first phase. Consider any such round. Throughout this round, no inconsistency is detected, otherwise we would have started the second phase. Moreover, $\kappa \in U(v') \subset U(v)$ reliably.

Consider the sequence of δ nodes encountered during the search. Though there might be cycles due to faults, this sequence cannot include v which is kept in safe memory (otherwise we would have found an inconsistency). Since the round is unsuccessful, there must be a node in the sequence which contains a fault at some point during the round. Let v^* be the first such node in the sequence. Note that we do not require that v^* already contains a fault at the time it is considered during the round, nor that it still contains a fault at the end of the round (the adversary may "correct" the fault).

Since the nodes before v^* in the sequence (or v itself if v^* is the first node in the sequence) are not faulty, it must be $\kappa \in U(v^*) \subset U(v)$ reliably. Altogether $\kappa \in U(v')$ and $\kappa \in U(v^*)$. Hence either v' is a proper ancestor of v^* in the search tree, or v' is a descendant of v^* (including the case $v' = v^*$). We next show that the first case cannot happen. Assume by contradiction that v' is a proper ancestor of v^*. Thus v' appears at least twice in the sequence: once before v^*, and once at the end of the sequence. Recall that each time during the search we pass from a node w to one of its (supposed) children w', we check (unreliably) that $\kappa \in U(w') \subset U(w)$. In the next search step, when we consider a (supposed) child w'' of w', we check (unreliably) that $\kappa \in U(w'') \subset U(w')$, with $U(w')$ being kept in safe memory throughout the two search steps. By transitivity and considering that no inconsistency has been detected, the first copy of $U(v')$ must have taken two different values during the search. Hence the adversary must have corrupted the first copy of $U(v')$ at some point. But this contradicts our assumption that v^* is the first faulty node in the sequence.

Hence v' must be a descendant of v^* in the search tree: we charge the $O(\delta)$ cost of this round to the faulty value contained in v^*. Since, also in this case, no faulty value can be charged more than once in the same search, the overall contribution of this third type of rounds to the search is $O(\alpha\, \delta)$. The claim follows.

The following lemma is crucial for the amortized analysis of insert and delete operations.

Lemma 3.3. *In the resilient search tree of Section 2 the intervals, and thus the search tree, are modified every $\Omega(\delta)$ key insertions and/or deletions.*

Proof. 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. To prove the lemma, 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 $|I|$ reaches the 2δ threshold. From the discussion above, this can only happen after at least $\delta/4$ insertions of keys in the interval.

Suppose now $|I|$ 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.

Lemma 3.4. *In the resilient search tree of Section 2, the amortized cost of each key insertion and deletion is $O(\log n + \alpha\,\delta^2)$.*

Proof. By Lemma 3.2, the initial search costs $O(\log n + \alpha\,\delta^2)$. Inserting/deleting a key κ in the list of the found interval I containing κ costs $O(\delta)$. If, after the insertion/deletion, the search tree has to be modified, this can be done in time $O(\delta \log n + \delta^2)$. By Lemma 3.3, the latter cost can be amortized over $\Omega(\delta)$ insertions/deletions. Thus the overall amortized cost per operation is $O(\log n + \alpha\,\delta^2)$.

The following theorem summarizes the discussion above.

Theorem 3.1. *The resilient search tree of Section 2 has amortized cost $O(\log n + \alpha\,\delta^2)$ per operation and space complexity $O(n + \delta)$.*

4 Refined Search

The cost of searching can be reduced to $O(\log n + \alpha\,\delta^{1+\epsilon})$, for any constant $\epsilon > 0$, by carefully adapting the resilient binary searching procedure described in [12]. Roughly speaking, the idea is to read, for every $k = 1, 2, \ldots 1/\epsilon$, the first $(2\delta^{k\,\epsilon} + 1)$ copies of each variable associated to a node (instead of the first one only), every $\delta^{k\,\epsilon}$ nodes encountered during the search. For each value of k, we keep in safe memory the last node which "passed" the corresponding $(2\delta^{k\,\epsilon}+1)$-test. This multi-level set of checkpoints and tests allows us to backtrack in a smarter way, and thus to better amortize the costs. Unfortunately, with this approach ϵ cannot be arbitrarily close to zero. This is because the cost of each round is $O(\delta/\epsilon)$. Even worse, the number of checkpoints stored in safe memory is $\Omega(1/\epsilon)$.

We next describe a simpler, more efficient technique which allows us to reduce the cost of each search to $O(\log n + \alpha\,\delta)$. We refine the search algorithm of Section 2. As before, in each round we start with a node v such that $\kappa \in U(v)$, and we end up with a node v' such that either $\kappa \in I(v')$ or $\kappa \in U(v') \subset U(v)$. However, this time we carry out more phases per round: in particular, the number of phases grows from two to $O(\log \delta)$. In phase k, $k = 1, 2, \ldots, \lceil \log_2(\delta + 1) \rceil$, we execute the search steps, consistency checks included, according to the majority value of the first $(2^k - 1)$ copies of each variable (or any value if there is no majority). In the last phase $k = 1 + \lceil \log_2(\delta + 1) \rceil$ we use all the $(2\delta + 1)$ copies. We recall that the majority value, if any, can be computed in $O(2^k)$ time and $O(1)$ space with the algorithm in [6]. Note that all the phases but the last one are unreliable. However, the degree of "unreliability" decreases from phase to phase. Phase $(k + 1)$ starts only if phase k fails. Observe that, differently from the approach described at the beginning of this section, here we need to store in safe memory only one checkpoint and the index k of the current phase.

Theorem 4.1. *The resilient search tree of Section 2, with the refined search described above, has amortized cost $O(\log n + \alpha\,\delta)$ per operation and space complexity $O(n + \delta)$.*

Proof. From the proof of Lemmas 3.1, 3.2, and 3.4, it is sufficient to show that finding the interval I containing a given key κ, with the refined search procedure, costs $O(\log n + \alpha\,\delta)$ only.

In the following we denote by v and v' the initial and final nodes of the round considered. Recall that a round is *successful* if either we find I containing κ or the depth of v' in the search tree is at least the depth of v plus δ. As in the proof of Lemma 3.2, we distinguish the following three types of rounds:

(a) Successful rounds which end with phase 1. The total cost of these rounds is trivially $O(\log n + \delta)$.

(b) Successful rounds which end with phase k, $\mathbf{k \geq 2}$. Any such round contributes $O(\sum_{i=1}^{k} 2^i \delta) = O(2^k \delta)$ to the total cost. Consider phase $(k - 1)$. Since the phase failed, there must be a node v^*, at distance smaller than δ from v, which contains at least 2^{k-2} faults. We charge the cost of this round evenly against such faults, with a cost of $O(\delta)$ per fault. Since no fault can be charged more than once during the search considered, the total cost of this type of rounds is $O(\alpha\,\delta)$.

(c) Unsuccessful rounds which end with phase k, $\mathbf{k \geq 1}$. Any such round costs $O(2^k \delta)$. No inconsistency is detected during phase k (otherwise we would have started phase $k + 1$). Therefore, generalizing the argument used in the proof of Lemma 3.2, there must be a node v^*, $v^* \neq v$, along the path from v to v' which is affected by at least 2^{k-1} faults (it might be v' itself). We charge the cost of the round to such faults, with an average cost $O(\delta)$ per fault. Also in this case, no fault can be charged more than once. Hence the total cost of this third type of rounds is $O(\alpha\,\delta)$. The claim now follows.

References

[1] J. A. Aslam and A. Dhagat. Searching in the presence of linearly bounded errors. *Proc. 23rd ACM Symp. on Theory of Computing* (STOC'91), 486–493, 1991.

[2] Y. Aumann and M. A. Bender. Fault-tolerant data structures. *Proc. 37th IEEE Symp. on Foundations of Computer Science* (FOCS'96), 580–589, 1996.

[3] J. Blömer and J.-P. Seifert. Fault based cryptanalysis of the Advanced Encryption Standard (AES). *Proc. 7th Int. Conf. on Financial Cryptography* (FC'03), LNCS 2742, 162–181, 2003.

[4] M. Blum, W. Evans, P. Gemmell, S. Kannan and M. Naor. Checking the correctness of memories. *Proc. 32th IEEE Symp. on Foundations of Computer Science* (FOCS'91), 1991.

[5] R. S. Borgstrom and S. Rao Kosaraju. Comparison based search in the presence of errors. *Proc. 25th ACM Symp. on Theory of Computing* (STOC'93), 130–136, 1993.

[6] R. Boyer and S. Moore. MJRTY - A fast majority vote algorithm. University of Texas Tech. Report, 1982.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press/McGraw-Hill Book Company, 2nd Edition, 2001.

[8] A. Dhagat, P. Gacs, and P. Winkler. On playing "twenty questions" with a liar. *Proc. 3rd ACM-SIAM Symp. on Discrete Algorithms* (SODA'92), 16–22, 1992.

[9] M. Farach-Colton. Personal communication. January 2002.

[10] U. Feige, P. Raghavan, D. Peleg, and E. Upfal. Computing with noisy information. *SIAM Journal on Computing*, 23, 1001–1018, 1994.

[11] I. Finocchi and G. F. Italiano. Sorting and searching in the presence of memory faults (without redundancy). *Proc. 36th ACM Symposium on Theory of Computing* (STOC'04), 101–110, 2004.

[12] I. Finocchi, F. Grandoni, and G. F. Italiano. Optimal sorting and searching in the presence of memory faults. *Proc. 33rd Int. Colloquium on Automata, Lang. and Prog.* (ICALP'06), 2006.

[13] M. Henzinger. The past, present and future of Web Search Engines. Invited talk. *31st Int. Coll. Automata, Languages and Programming*, Turku, Finland, July 12–16 2004.

[14] D. J. Kleitman, A. R. Meyer, R. L. Rivest, J. Spencer, and K. Winklmann. Coping with errors in binary search procedures. *Journal of Computer and System Sciences*, 20:396–404, 1980.

[15] K. Mehlhorn and S. Näher. *LEDA: A platform for combinatorial and geometric computing.* Cambridge University Press, 1999.

[16] S. Muthukrishnan. On optimal strategies for searching in the presence of errors. *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms* (SODA'94), 680–689, 1994.

[17] A. Pelc. Searching with known error probability. *Theoretical Computer Science*, 63, 185–202, 1989.

[18] A. Pelc. Searching games with errors: Fifty years of coping with liars. *Theoretical Computer Science*, 270, 71–109, 2002.

[19] P. J. Plauger, A. A. Stepanov, M. Lee, D. R. Musser. *The C++ Standard Template Library*, Prentice Hall, 2000.

[20] S. Skorobogatov and R. Anderson. Optical fault induction attacks. *Proc. 4th Int. Workshop on Cryptographic Hardware and Embedded Systems*, LNCS 2523, 2–12, 2002.

[21] J. von Neumann, Probabilistic logics and the synthesis of reliable organisms from unreliable components. In *Automata Studies*, C. Shannon and J. McCarty eds., Princeton Univ. Press, 43–98, 1956.