Optimal Resilient Dynamic Dictionaries*

Gerth Stølting Brodal[1], Rolf Fagerberg[2], Irene Finocchi[3],
Fabrizio Grandoni[3], Giuseppe F. Italiano[4], Allan Grønlund Jørgensen[1],
Gabriel Moruz[1], and Thomas Mølhave[1]

[1] BRICS**, MADALGO***, Department of Computer Science,
University of Aarhus, Denmark
{gerth,jallan,gabi,thomasm}@daimi.au.dk
[2] Department of Mathematics and Computer Science,
University of Southern Denmark, Odense, Denmark
rolf@imada.sdu.dk
[3] Dipartimento di Informatica, Università di Roma "La Sapienza", Italy
{finocchi,grandoni}@di.uniroma1.it
[4] Dipartimento di Informatica, Sistemi e Produzione, Università di Roma
"Tor Vergata", Italy
italiano@disp.uniroma2.it

Abstract. We investigate the problem of computing in the presence of faults that may arbitrarily (i.e., adversarially) corrupt memory locations. In the faulty memory model, any memory cell can get corrupted at any time, and corrupted cells cannot be distinguished from uncorrupted ones. An upper bound δ on the number of corruptions and $O(1)$ reliable memory cells are provided. In this model, we focus on the design of resilient dictionaries, i.e., dictionaries which are able to operate correctly (at least) on the set of uncorrupted keys. We first present a simple resilient dynamic search tree, based on random sampling, with $O(\log n + \delta)$ expected amortized cost per operation, and $O(n)$ space complexity. We then propose an optimal deterministic static dictionary supporting searches in $\Theta(\log n + \delta)$ time in the worst case, and we show how to use it in a dynamic setting in order to support updates in $O(\log n + \delta)$ amortized time. Our dynamic dictionary also supports range queries in $O(\log n + \delta + t)$ worst case time, where t is the size of the output. Finally, we show that every resilient search tree (with some reasonable properties) must take $\Omega(\log n + \delta)$ worst-case time per search.

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 [17,22]. Furthermore, latest trends in storage development point out that memory devices are getting smaller and more complex. Additionally, they work at lower voltages and higher frequencies [10]. All these improvements increase the likelihood of soft memory errors, whose rate is expected to increase for both SRAM and DRAM memories [24]. 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, which 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 of erroneous answers to keyword searches [18].

Memory corruptions have been addressed in various ways, both at the hardware and software level. At the hardware level, memory corruptions are tackled using error detection mechanisms, such as redundancy, parity checking or Hamming codes. However, adopting such mechanisms involves non-negligible penalties with respect to performance, size, and cost, thus memories implementing them are rarely found in large scale clusters or ordinary workstations. Dealing with unreliable information has been addressed in the algorithmic community in a variety of different settings, including the liar model [1,5,12,20,23], fault-tolerant sorting networks [2,21,25], resiliency of pointer-based data structures [3], parallel models of computation with faulty memories [9].

A model for memory faults. Finocchi and Italiano [16] introduced the *faulty-memory RAM*. 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). We remark that δ is not a constant, but is a parameter of the model. The pessimistic faulty-memory RAM captures situations like cosmic-rays bursts and memories with non-uniform fault-probability, which would be difficult to be modeled otherwise. The model also assumes 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. An algorithm or a data structure is called *resilient* if it works correctly at least on the set of uncorrupted cells in the input. In particular, a resilient searching algorithm returns a positive answer if there

exists an uncorrupted element in the input equal to the search key. If there is no element, corrupted or uncorrupted, matching the search key, the algorithm returns a negative answer. If there is a corrupted value equal to the search key, the answer can be either positive or negative.

Previous work. Several problems have been addressed in the faulty-memory RAM. In the original paper [16], lower bounds and (non-optimal) algorithms for sorting and searching were given. In particular, it has been proved in [16] that searching in a sorted array takes $\Omega(\log n + \delta)$ time, i.e., up to $O(\log n)$ corruptions can be tolerated while still preserving the classical $O(\log n)$ searching bound. Matching upper bounds for sorting and randomized searching, as well as an $O(\log n + \delta^{1+\varepsilon})$ deterministic searching algorithm, were then given in [14]. Resilient search trees that support searches, insertions, and deletions in $O(\log n + \delta^2)$ amortized time were introduced in [15]. Recently, Jørgensen *et al.* [19] proposed priority queues supporting both insert and delete-min operations in $O(\log n + \delta)$ amortized time. Finally, in [13] it was empirically shown that resilient sorting algorithms are of practical interest.

Our contribution. In this paper we continue the work on resilient dictionaries. We present a simple randomized dynamic search tree achieving $O(\log n + \delta)$ amortized expected time per operation. We then present the first resilient algorithm for deterministically searching in a sorted array in optimal $O(\log n + \delta)$ time, matching the lower bound from [16]. We use this algorithm, to build a resilient deterministic dynamic dictionary supporting searches in $O(\log n + \delta)$ worst case time and updates in $O(\log n + \delta)$ amortized time. Range queries are supported in $O(\log n + \delta + t)$ time where t is the size of the output. Furthermore, we prove a lower bound stating that every resilient dictionary (with some reasonable properties) must take $\Omega(\log n + \delta)$ worst-case time per search.

Preliminaries. We denote by α the *actual* number of faults. Of course $\alpha \leq \delta$. A *resilient variable* x consists of $(2\delta + 1)$ copies of a (classical) variable. The *value* of x is the majority value of its copies. This value is well defined 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]).

2   A Simple Randomized Resilient Dictionary

In this section we present a simple randomized resilient search tree, which builds upon the resilient search tree of [15]. Our search tree takes $O(\log n + \delta)$ amortized time per operation, in expectation. We maintain a dynamically evolving set of intervals $I_1, I_2, \ldots I_h$. Initially, when the set of keys is empty, there is a unique interval $I_1 = (-\infty, +\infty)$. Throughout the sequence of operations we maintain the following invariants:

 (i) The intervals are non-overlapping, and their union is $(-\infty, +\infty)$.
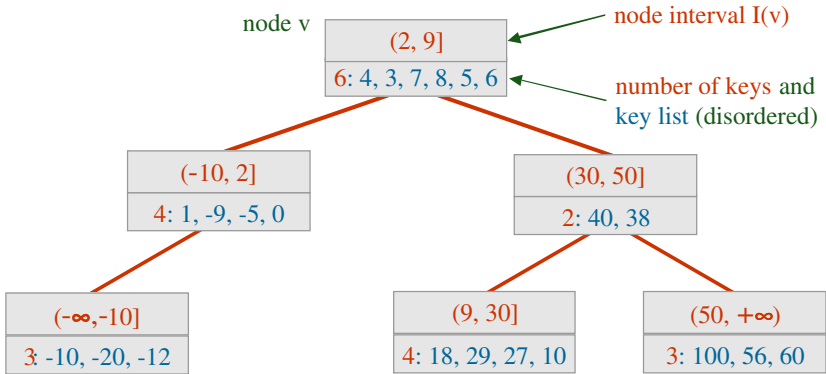(ii) Each interval contains less than 2δ keys.

Fig. 1. A resilient search tree

(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 e, we first need to find the interval $I(e)$ containing e (*interval search*). Invariant (i) guarantees that such an interval exists, and 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 maintained in a standard balanced binary search tree. Throughout the paper we use as a reference implementation an *AVL* tree [11]. 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 maintain the following variables:

1. (reliably) the endpoints of the corresponding interval $I(v)$ and the number $|I(v)|$ of keys contained in the interval $I(v)$;
2. (reliably) 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;
3. (unreliably, i.e., in a single copy) the (unordered) set of current keys contained in $I(v)$, stored in an array of size 2δ.

For an example, see Figure 1. The nodes of the search tree are stored in 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 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 due to doubling is $O(\delta)$. As we will see, this cost can be charged to the cost of the interval search which is performed in each operation: from now on we will not mention this cost any further.

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

Search. Every search is performed by first searching for the interval $I(e)$ containing e (*interval search*), and then by linearly searching for e in $I(e)$. The interval search is implemented as follows. We perform a classical search, where for each relevant resilient variable we consider one of its $2\delta + 1$ copies chosen uniformly at random. We implement the search so as to read at most one random copy of each resilient variable (unless pointers corruption forces the search to cycle). This assumption will turn out to be useful in the analysis. Once the search is concluded, we check reliably (in $O(\delta)$ time) whether the final interval contains e. If not, the search is restarted from scratch. The search is restarted also as soon as an inconsistency is found, for instance if the number of steps performed gets larger than the height of the tree.

Insert. We initially find $I = I(e)$ with the procedure above. Then, if e is not already in the list of keys associated to I, we add e 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 in 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 [14]. 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 $I = I(e)$ using the search procedure above. If we find e in the list of keys associated to I, we delete e. 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 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 to 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. We remark that the implementation of the interval search is the main difference between our improved search tree and the search tree in [15].

Theorem 1. *The resilient search tree above has $O(\log n + \delta)$ expected amortized time per operation and $O(n)$ space complexity.*

Proof. The space complexity is discussed above. Let $S(n, \delta)$ be the expected time needed to perform an interval search. By Invariant (ii), each search operation takes $S(n, \delta) + O(\delta)$ 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. However, it is not hard to show that, by Invariants (ii) and (iii), the search tree is modified every $\Omega(\delta)$ insert and/or delete operations (see [15] for a formal proof of this simple fact). Hence the amortized cost of insert and delete operations is $S(n, \delta) + O(\log n + \delta)$ in expectation.

It remains to bound $S(n, \delta)$. Each search round takes $O(\log n + \delta)$ time. Thus it is sufficient 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 i-th resilient variable considered during the round, $i = 1, 2, \ldots, p$. The probability that all the copies chosen during a given round are faithful is at least

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

Given this event, by the assumptions on the algorithm the resilient variables considered must be all distinct. As a consequence $\sum_{i=1}^{p} \alpha_i \leq \alpha \leq \delta$, and 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. □

3 An Optimal Static Dictionary

In this section we close the gap between lower and upper bounds for deterministic resilient searching algorithms. We present a resilient algorithm that searches for an element in a sorted array in $O(\log n + \delta)$ time in the worst case, which is optimal [16]. The previously best known deterministic dictionary supports searches in $O(\log n + \delta^{1+\varepsilon})$ time [14].

We design a binary search algorithm, which only advance one level in the wrong direction for each corrupted element misleading it. We then design a verification procedure that checks the result of the binary search. We count the number of detected corruptions and adjust our algorithm accordingly to ensure that no element is used more than once. To avoid reading the same faulty value twice, we divide the input array into implicit blocks. Each block consists of $5\delta+1$ consecutive elements of the input and is structured in three segments: the *left*

verification segment, LV, consists of the first 2δ elements, the next $\delta+1$ elements form the *query segment*, Q, and the *right verification segment*, RV, consists of the last 2δ elements of the block. The left and right verification segments, LV and RV, are used only by the verification procedure. The elements in the query segment are used to define $\delta + 1$ sorted sequences $S_0, \ldots, S_\delta$. The j'th element of sequence S_i, $S_i[j]$, is the i'th element of the query segment of the j'th block, and is located at position $\text{pos}_i(j) = (5\delta + 1)j + 2\delta + i$ in the input array.

We store a value $k \in \{0, \ldots, \delta\}$ in safe memory identifying the sequence S_k on which we perform the binary search. Also, k identifies the number of corruptions detected. Whenever we detect a corruption, we change the sequence on which we perform the search by incrementing k. Since there are $\delta + 1$ disjoint sequences, there exists at least one sequence without any corruptions.

Binary search. The binary search is performed on the elements of S_k. We store in safe memory the search key, e, and the left and right sequence indices, l and r, used by the binary search. Initially, $l = -1$ is the position of an implicit $-\infty$ element. Similarly, r is the position of an implicit ∞ to the right of the last element. Since each element in S_k belongs to a distinct block, l and r also identify two blocks B_l and B_r.

In each step of the binary search the element at position $i = \lfloor (l + r)/2 \rfloor$ in S_k is compared with e. Assume without loss of generality that this element is smaller than e. We set l to i and decrement r by one. We then compare e with $S_k[r]$. If this element is larger than e, the search continues. Otherwise, if no corruptions have occurred, the position of the search element is in block B_r or B_{r+1} in the input array. When two adjacent elements are identified as in the case just described, or when l and r become adjacent, we invoke a verification procedure on the corresponding blocks.

The verification procedure determines whether the two adjacent blocks, denoted B_i and B_{i+1}, are correctly identified. If the verification succeeds, the binary search is completed, and all the elements in the two corresponding adjacent blocks, B_i and B_{i+1} are scanned. The search returns true if e is found during the scan, and false otherwise. If the verification fails, we backtrack the search two steps, since it may have been mislead by corruptions. To facilitate backtracking, we store two word-sized bit-vectors, d and f in safe memory. The i'th bit of d indicates the direction of the search and the i'th bit of f indicates whether there was a rounding in computing the middle element in the i'th step of the binary search respectively. We can compute the values of l and r in the previous step by retrieving the relevant bits of d and f. If the verification fails, it detects at least one corruption and therefore k is incremented, thus the search continues on a different sequence S_k.

Verification phase. Verification is performed on two adjacent blocks, B_i and B_{i+1}. It either determines that e lies in B_i or B_{i+1} or detects corruptions. The verification is an iterative algorithm maintaining a value which expresses the confidence that the search key resides in B_i or B_{i+1}. We compute the *left confidence*, c_l, which is a value that quantifies the confidence that e is in B_i or to

the right of it. Intuitively, an element in LV_i smaller than e is consistent with the thesis that e is in B_i or to the right of it. However, an element in LV_i larger than e is inconsistent. Similarly, we compute the *right confidence*, c_r, to express the confidence that e is in B_{i+1} or to the left of it.

We compute c_l by scanning a sub-interval of the left verification segment, LV_i, of B_i. Similarly, the right confidence is computed by scanning the right verification segment, RV_{i+1}, of B_{i+1}. Initially, we set $c_l = 1$ and $c_r = 1$. We scan LV_i from right to left starting at the element at index $v_l = 2\delta - 2k$ in LV_i. Similarly, we scan RV_{i+1} from left to right beginning with the element at position $v_r = 2k$. In an iteration we compare $LV_i[v_l]$ and $RV_{i+1}[v_r]$ against e. If $LV_i[v_l] \leq e$, we increment c_l by one, otherwise it is decreased by one and k is increased by one. Similarly, if $RV_{i+1}[v_r] \geq e$, we increment c_r by one; otherwise, we decrease c_r and increase k. The verification procedure stops when $\min(c_r, c_l)$ equals $\delta - k + 1$ or 0. The verification succeeds in the former case, and fails in the latter.

Theorem 2. *The algorithm is resilient and searches for an element in a sorted array in $O(\log n + \delta)$ time.*

Proof. We first prove that when c_l or c_r decrease during verification, a corruption has been detected. We increase c_l when an element smaller than e is encountered in LV_i, and decrease it otherwise. Intuitively, c_l can been seen as the size of a stack S. When we encounter an element smaller than e, we treat it as if it was pushed, and as if a pop occurred otherwise. Initially, the element g from the query segment of B_i used by the binary search is pushed in S. Since g was used to define the left boundary in the binary search, $g < e$ at that time. Each time an element $LV_i[v] < e$ is popped from the stack, it is *matched* with the current element $LV_i[v_l]$. Since $LV_i[v] < e < LV_i[v_l]$ and $v_l < v$, at least one of $LV_i[v_l]$ and $LV_i[v]$ is corrupted, and therefore each match corresponds to detecting at least one corruption. It follows that if $2t - 1$ elements are scanned on either side during a failed verification, then at least t corruptions are detected.

We now argue that no single corruption is counted twice. A corruption is detected if and only if two elements are matched during verification. Thus it suffices to argue that no element participates in more than one matching. We first analyze corruptions occurring in the left and right verification segments. Since the verification starts at index $2(\delta - k)$ in the left verification segment and k is increased when a corruption is detected, no element is accessed twice, and therefore not matched twice either. A similar argument holds for the right verification segment. Each failed verification increments k, thus no element from a query segment is read more than once. In each step of the binary search both the left and the right indices are updated. Whenever we backtrack the binary search, the last two updates of l and r are reverted. Therefore, if the same block is used in a subsequent verification, a new element from the query segment is read, and this new element is the one initially on the stack. We conclude that elements in the query segments, which are initially placed on the stack, are never matched twice either.

To argue correctness we prove that if a verification is successful, and e is not found in the scan of the two blocks, then no uncorrupted element equal to e

exists in the input. If a verification succeeded then $c_l \geq \delta - k + 1$. Since only $\delta - k$ more corruptions are possible and since an element equal to e was not found, there is at least one uncorrupted element in LV_i smaller than e, and thus there can not be any uncorrupted elements equal to e to the left of B_i in the input array. By a similar argument, if $c_r \geq \delta - k + 1$, then all uncorrupted elements to the right of B_{i+1} in the input array are larger than e.

We now analyze the running time. We charge each backtracking of the binary search to the verification procedure that triggered it. Therefore, the total time of the algorithm is $O(\log n)$ plus the time required by verifications. To bound the time used for all verification steps we use the fact that if $O(f)$ time is used for a verification step, then $\Omega(f)$ corruptions are detected or the algorithm ends. At most $O(\delta)$ time is used in the last verification for scanning the two blocks.   □

4   A Dynamic Dictionary

In this section we describe a linear space resilient dynamic dictionary supporting searches in optimal $O(\log n + \delta)$ worst case time and range queries in optimal $O(\log n + \delta + t)$ worst case time, where t is the size of the output. The amortized update cost is $O(\log n + \delta)$. The previous best known deterministic dynamic dictionary, is the resilient search tree of [15], which supports searches and updates in $O(\log n + \delta^2)$ amortized time.

Structure. The sorted sequence of elements is partitioned into a sequence of *leaf structures*, each storing $\Theta(\delta \log n)$ elements. For each leaf structure we select a guiding element, and these $O(n/(\delta \log n))$ elements are also stored in the leaves of a reliably stored binary search tree. Each guiding element is larger than all uncorrupted elements in the corresponding leaf structure.

For this reliable *top tree* T, we use the binary search tree in [7], which consists of $h = \log |T| + O(1)$ levels when containing $|T|$ elements. In the full version [8] it is shown how the tree can be maintained such that the first $h - 2$ levels are complete. We lay out the tree in memory in left-to-right breadth first order, as specified in [7]. It uses linear space, and supports updates in $O(\log^2 |T|)$ amortized time. Global rebuilding is used when $|T|$ changes by a constant factor.

All the elements in the top tree are stored as resilient variables. . Since a resilient variable takes $O(\delta)$ space, $O(\delta|T|)$ space is used for the entire structure. The time used for storing and retrieving a resilient variable is $O(\delta)$, and therefore the additional work required to handle the resilient variables increases the amortized update cost to $O(\delta \log^2 |T|)$ time.

The leaf structure consists of a top bucket B and b buckets, $B_0, \ldots, B_{b-1}$, where $\log n \leq b \leq 4 \log n$. Each bucket B_i contains between δ and 6δ input elements, stored consecutively in an array of size 6δ, and uncorrupted elements in B_i are smaller than uncorrupted elements in B_{i+1}. For each bucket B_i, the top bucket B associates a guiding element larger than all elements in B_i, a pointer to B_i, and the size of B_i, all stored reliably. Since storing a value reliably uses $O(\delta)$ space, the total space used by the top bucket is $O(\delta \log n)$. The guiding elements of B are stored as a sorted array to enable fast searches.

Searching. The search operation first finds a leaf in the top tree, and then searches the corresponding leaf structure. Let h denote the height of T. If $h \leq 3$, we perform a standard tree search from the root of T using the reliably stored guiding elements. Otherwise, we locate internal nodes, v_1 and v_2, with guiding elements g_1 and g_2, such that $g_1 < e \leq g_2$, where e is the search key. Since $h - 2$ is the last complete level of T, level $\ell = h - 3$ is complete and contains only internal nodes. The breadth first layout of T ensures that elements of level ℓ are stored consecutively in memory. The search operation locates v_1 and v_2 using the deterministic resilient search algorithm from Section 3 on the array defined by level ℓ. The search only considers the $2\delta + 1$ cells in each node containing guiding elements and ignores memory used for auxiliary information. Although they are stored using as resilient variables, each of the $2\delta + 1$ copies are considered a single element by the search. We modify the resilient searching algorithm previously introduced such that it reports two consecutive blocks with the property that if the search key is in the structure, it is contained in one of them. The reported two blocks, each of size $5\delta + 1$, span $O(1)$ nodes of level ℓ and the guiding elements of these are queried reliably to locate v_1 and v_2. The appropriate leaf can be in either of the subtrees rooted at v_1 and v_2, and we perform a standard tree search in both using the reliably stored guiding elements. Searching for an element in a leaf structure is performed by using the resilient search algorithm from Section 3 on the top bucket, B, similar to the way v_1 and v_2 were found in T. The corresponding reliably stored pointer is then followed to a bucket B_i, which is scanned. Range queries can be performed by scanning the level ℓ, starting at v, and reporting relevant elements in the leaves below it.

Updates. Efficiently updating the structure is performed using standard bucketing techniques. To insert an element into the dictionary, we first perform a search to locate the appropriate bucket B_i in a leaf structure, and then the element is appended to B_i and the size updated. When the size of B_i increases to 6δ, we split it into two buckets. We compute a guiding element that splits B_i in $O(\delta^2)$ time by repeatedly scanning B_i and extracting the minimum element. The element m returned by the last iteration is kept in safe memory. In each iteration, we select a new m which is the minimum element in B_i larger than the current m. Since at most δ corruptions can occur, B_i contains at least 2δ uncorrupted elements smaller than m and 2δ uncorrupted elements larger, after $|B_i|/2$ iterations. The new split element is reliably inserted in the top bucket using an insertion sort step in $O(\delta \log n)$ time. Similarly, when the degree the top bucket becomes $4 \log n$, it is split in two new leaf structures in $O(\delta \log n)$ time, and a new guiding element is inserted into the top tree. Deletions are handled similarly.

Theorem 3. *The resilient dynamic dictionary structure uses $O(n)$ space while supporting searches in $O(\log n + \delta)$ time worst case with an amortized update cost of $O(\log n + \delta)$. Range queries with an output size of t is performed in worst case $O(\log n + \delta + t)$ time.*

5   A Lower Bound

In the following we restrict our attention to comparison based dictionaries where the keys are stored in one or more arrays, and the address of each array is maintained in one or more pointers. These assumptions can be partially relaxed. However, we do not discuss such relaxations since they do not add much to the discussion below.

Theorem 4. *Every resilient search tree of the kind above requires $\Omega(\log n + \delta)$ worst-case time per search.*

Proof. Every search tree, even in a system without memory faults, takes $\Omega(\log n)$ worst-case time per search. This lower bound extends immediately to the case of resilient search trees. Hence, without loss of generality, assume that $\log n = o(\delta)$. Under this assumption, it is sufficient to show that the time required by a resilient search operation is $\Omega(\delta)$.

Consider any given search tree ST of the kind considered. Let $K_1, K_2, \ldots, K_p$ be the arrays in unsafe memory where (subset of) the keys are maintained. For each K_i there must be at least one pointer containing its address. Recall that only a constant number of keys can be kept in safe memory. Hence $\Theta(n)$ keys are stored in unsafe memory only.

Suppose there is an array K_i containing $\Omega(n)$ keys. Then, by the lower bound on static resilient searching [14], searching for a given key in K_i takes time $\Omega(\log n + \delta) = \Omega(\delta)$. Hence, let us assume that all the arrays contain $o(n)$ keys. Since there are $\omega(1)$ arrays, not all the corresponding pointers can be kept in safe memory. In particular, there must be an array K_i whose pointers are all maintained in the unsafe memory. Assume that we search for a faithful key e contained in K_i. Suppose by contradiction that ST concludes the search in $o(\delta)$ time. This means that ST can read only $o(\delta)$ pointers to reach K_i, and at most $o(\delta)$ keys of K_i. Then an adversary, by corrupting $o(\delta)$ memory words only, can make ST answer no to the query. This contradicts the resiliency of ST, which in the case considered must always find a key equal to e.   □

References

1. Aslam, J.A., Dhagat, A.: Searching in the presence of linearly bounded errors. In: ACM STOC'91, pp. 486–493
2. Assaf, S., Upfal, E.: Fault-tolerant sorting networks. SIAM J. Discrete Math. 4(4), 472–480 (1991)
3. Aumann, Y., Bender, M.A.: Fault-tolerant data structures. In: IEEE FOCS'96, pp. 580–589
4. Blum, M., Evans, W., Gemmell, P., Kannan, S., Naor, M.: Checking the correctness of memories. In: IEEE FOCS'91
5. Borgstrom, R.S., Rao Kosaraju, S.: Comparison based search in the presence of errors. In: ACM STOC'93, pp. 130–136.
6. Boyer, R., Moore., S.: MJRTY: - A fast majority vote algorithm. University of Texas Tech. Report (1982)

7. Brodal, G.S., Fagerberg, R., Jacob, R.: Cache-oblivious search trees via binary trees of small height. ACM-SIAM SODA'02, 39–48
8. Brodal, G.S., Fagerberg, R., Jacob, R.: Cache-oblivious search trees via trees of small height. Technical Report ALCOMFT-TR-02-53, ALCOM-FT (May 2002)
9. Chlebus, B.S., Gambin, A., Indyk, P.: Shared-memory simulations on a faulty-memory DMM. In: ICALP'96, pp. 586–597
10. Constantinescu, C.: Trends and challenges in VLSI circuit reliability. IEEE micro 23(4), 14–19 (2003)
11. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. The MIT Press/McGraw-Hill Book Company (2001)
12. Feige, U., Raghavan, P., Peleg, D., Upfal, E.: Computing with noisy information. SIAM Journal on Computing 23, 1001–1018 (1994)
13. Petrillo, U.F., Finocchi, I., Italiano, G.F.: The price of resiliency: a case study on sorting with memory faults. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 768–779. Springer, Heidelberg (2006)
14. Finocchi, I., Grandoni, F., Italiano, G.F.: Optimal sorting and searching in the presence of memory faults. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 286–298. Springer, Heidelberg (2006)
15. Finocchi, I., Grandoni, F., Italiano, G.: Resilient search trees. In: ACM-SIAM SODA'07, pp. 547–555.
16. Finocchi, I., Italiano, G.F.: Sorting and searching in faulty memories. Algorithmica ACM STOC'04, pp. 101–110 (Extended abstract) (to appear)
17. Hamdioui, S., Al-Ars, Z., de Goor, J.V., Rodgers, M.: Dynamic faults in random-access-memories: Concept, faults models and tests. Journal of Electronic Testing: Theory and Applications 19, 195–205 (2003)
18. Henzinger, M.R.: Combinatorial algorithms for web search engines - three success stories. In: ACM-SIAM SODA'07 (invited talk)
19. Jørgensen, A.G., Moruz, G., Mølhave, T.: Priority queues resilient to memory faults. In: Dehne, F., Sack, J.-R., Zeh, N. (eds.) WADS 2007. LNCS, vol. 4619, Springer, Heidelberg (2007)
20. Kleitman, D.J., Meyer, A.R., Rivest, R.L., Spencer, J., Winklmann, K.: Coping with errors in binary search procedures. Journal of Computer and System Sciences 20, 396–404 (1980)
21. Leighton, T., Ma, Y.: Tight bounds on the size of fault-tolerant merging and sorting networks with destructive faults. SIAM Journal on Computing 29(1), 258–273 (1999)
22. May, T.C., Woods, M.H.: Alpha-particle-induced soft errors in dynamic memories. IEEE Transactions on Electron Devices 26(2) (1979)
23. Pelc, A.: Searching games with errors: Fifty years of coping with liars. Theoretical Computer Science 270, 71–109 (2002)
24. Tezzaron Semiconductor. Soft errors in electronic memory - a white paper (2004), http://www.tezzaron.com/about/papers/papers.html
25. Yao, A.C., Yao, F.F.: On fault-tolerant networks for sorting. SIAM Journal on Computing 14, 120–128 (1985)