Resilient Dictionaries for Randomly Unreliable Memory

Stefano Leucci
Department of Algorithms and Complexity, Max Planck Institute for Informatics, Germany
https://www.stefanoleucci.com
stefano.leucci@mpi-inf.mpg.de

Chih-Hung Liu
Department of Computer Science, ETH Zürich, Switzerland
chih-hung.liu@inf.ethz.ch

Simon Meierhans
Department of Computer Science, ETH Zürich, Switzerland
mesimon@student.ethz.ch

Abstract
We study the problem of designing a dictionary data structure that is resilient to memory corruptions. Our error model is a variation of the faulty RAM model in which, except for constant amount of definitely reliable memory, each memory word is randomly unreliable with a probability \( p < \frac{1}{2} \), and the locations of the unreliable words are unknown to the algorithm. An adversary observes the whole memory and can, at any time, arbitrarily corrupt (i.e., modify) the contents of one or more unreliable words.

Our dictionary has capacity \( n \), stores \( N < n \) keys in the optimal \( O(N) \) amount of space, supports insertions and deletions in \( O(\log n) \) amortized time, and allows to search for a key in \( O(\log n) \) worst-case time. With a global probability of at least \( 1 - \frac{1}{n} \), all possible search operations are guaranteed to return the correct answer w.r.t. the set of uncorrupted keys.

The closest related results are the ones of Finocchi et al. [13] and Brodal et al. [6] on the faulty RAM model, in which all but \( O(1) \) memory is unreliable. There, if an upper bound \( \delta \) on the number of corruptions is known in advance, all dictionary operations can be implemented in \( \Theta(\log n + \delta) \) amortized time, thus trading resiliency for speed as soon as \( \delta = \omega(\log n) \).

Our construction does not need to know the value of \( \delta \) in advance and remains fast and effective even when up to a constant fraction of the available memory is corrupted. Our techniques can be immediately extended to implement other data types (e.g., associative containers and priority queues), which can then be used as a building block in the design of other resilient algorithms. For example, we are able to solve the resilient sorting problem in our model using \( O(n \log n) \) time.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis
Keywords and phrases resilient dictionary, unreliable memory, faulty RAM

Digital Object Identifier 10.4230/LIPIcs.ESA.2019.70
Funding Research supported by the SNSF (Swiss National Science Foundation) grant 200021_165524.

1 Introduction
Computing platforms sometimes exhibit temporary or permanent memory failures and they can be expected to become more frequent due to the challenge of providing high amounts of energy on an ever smaller scale, while simultaneously increasing the operation frequency. Most algorithms completely malfunction even if a single memory error occurs. For example, one may consider the Mergesort algorithm, where a constant fraction of elements can be placed out of order following a single corruption. Classical approaches to deal with these...
faults involve data replication or the use of error-correcting code (ECC) memory, and typically require more computational resources (i.e., space and/or time) or dedicated hardware. This gave rise to a line of research focusing on the design of fast and compact algorithms and data structures that are resilient to memory faults, i.e., that provide provable guarantees on their output even when some memory words become corrupted.

A widely used model to capture this kind of corruption is the faulty RAM model [14], in which an algorithm has access to only a constant amount of reliable memory, while all the other words are unreliable. An adversary is then allowed to corrupt (i.e., change the value of) up to $\delta$ unreliable words, and the algorithms’ performances are evaluated as a function of $\delta$. We consider a variation of the above model in which, in addition to the $O(1)$ reliable memory, each of the remaining words is unreliable with a certain probability $p < \frac{1}{2}$ but the algorithm is unaware of which of these locations are reliable. While both settings are theoretical abstractions of the more complex error patterns that can happen in the real-world, it is not hard to come up with examples that closely resemble our error model. For example, when a DRAM module is faulty, its contents are no longer reliable and read and write operations might produce corrupted values. To complicate things further, even though the faulty locations might be contiguous, processors map physical addresses to hardware locations using complex (and often undocumented) mapping functions. This increases memory access parallelism but has the side effect of distributing the unreliable locations over the whole address space.

We focus on the problem of designing a resilient data structure implementing the dictionary abstract data type, that is, we want to maintain a set of up to $n$ keys under insertion and deletions so that we can answer membership queries. While it is easy to adapt any classical (non-resilient) data structure to our model with a multiplicative blow up of $\Theta(\log n)$ in the time and space complexities, one might wonder whether provably good guarantees w.r.t. the number of corruptions can be achieved using the asymptotically optimal $O(n)$ amount of space. In the rest of this paper we show that the answer is indeed affirmative.

Our model

Our model of computation is a random access machine [1] in which each word of memory is either reliable or unreliable. The memory itself is split into two regions:

- A safe region of $O(1)$ words (representing, e.g., processor registers) that are always reliable.
- A region containing all of the remaining words (representing, e.g., the main memory).

Each of these words is independently reliable with probability $1 - p$ and unreliable with probability $p$, for some constant $p < \frac{1}{2}$.

Unreliable words can be affected by corruption phenomena, i.e., their contents can unexpectedly change. Except for the $O(1)$ words in the safe memory region, an algorithm is unaware of which memory locations are unreliable and is unable to detect whether the value stored in an unreliable location has been corrupted. Following [14], we adopt a worst case approach to corruptions: a computationally-unbounded adversary knows the algorithm that is being executed, can observe the contents of the whole memory, and can distinguish reliable from unreliable words. The adversary can, at any point during the execution of the algorithm, simultaneously corrupt the values stored in one or more unreliable words.

---

1 This implicitly adopts a worst-case approach, so that all positive results also hold in the easier case of random, non-malicious corruptions.

2 Finding techniques to reliably discover the mapping functions is an area of active research. Proposed methods use thermal and timing data to correlate physical addresses to memory locations [19, 18, 24].
Our results and techniques

We design a resilient dictionary that stores up to \( n \) keys using the optimal amount of space, supports insertions, deletions, and membership queries (in the following insert, delete, and search) in \( O(\log n) \) amortized time and guarantees with high probability (w.h.p.) that, regardless of the number \( \delta \) of corruptions, all search operations work reliably on all uncorrupted keys. More precisely, the search operation behaves as follows: if the searched key \( k \) belongs to the dictionary and is uncorrupted, then the search operation correctly returns yes. If \( k \) does not belong to the dictionary and no corrupted key equals \( k \), the returned answer is guaranteed to be no. In the remaining cases the answer might be either yes or no.

As a comparison, the closest related results are the ones of Brodal et al. [6] and Finocchi et al. [13] on the faulty RAM model. There the authors show how, given \( \delta \), it is possible to build a dictionary that is resilient to up to \( \delta \) word corruptions, uses \( O(n) \) space, and whose insert, delete, and search operations require \( O(\log n + \delta) \) amortized time, which is tight.\(^3\)

Crucially, the task of selecting a good upper bound \( \delta \) on the number of corruptions at design time can turn out to be problematic: if \( \delta \) is too small then the resulting dictionary can abruptly fail when the \((\delta + 1)\)-th corruption occurs while, if \( \delta = \omega(\log n) \), one is forced to trade-off resiliency for speed as the additive term of \( \delta \) now becomes dominant in the time complexity.

Notice that, in our setting, the expected number of unreliable locations is \( pn = \Theta(n) \), implying that the actual number of unreliable locations is \( \Theta(n) \), except for an exponentially vanishing probability. Since \( \delta \) can be as large as \( \Theta(n) \) (think, e.g., of our faulty DRAM-example), a direct instantiation of the results of [6] and [13] yields the same bounds of the trivial dictionary that uses \( O(n) \) space and time per operation (just store all the keys in an unsorted array). Nevertheless, one might wish for a dictionary that remains resilient w.r.t. a constant fraction of key corruptions and whose operations do not bear the exorbitant cost of \( O(n) \) time per operation. Our results shows that this is indeed possible: when the unreliable locations are random fraction of the available memory, it is possible handle any (unknown) number \( \delta \) of corruptions with the same asymptotic guarantees obtained by the existing fault-tolerant dictionaries in the faulty RAM model when \( \delta = O(\log n) \).

From a high-level point of view, [13] buffers the dictionary keys into groups of size \( \Theta(\delta) \), and organizes the resulting groups into a pointer-based AVL tree. A core subroutine then consists of locating the group responsible for a given key in \( O(\log n + \delta) \) time. Since the corruption of any auxiliary information associated with a group might lead the search astray, the corresponding variables are replicated \( \Theta(\delta) \) times. A first difficulty to overcome in our model is that of removing the dependency from \( \delta \). We do so by using a different locate procedure that is based on a suitable walk over the tree vertices. The behavior of the walk is guided by the observed memory contents, yet we prove that there is an absolute probability of at least \( 1 - \frac{1}{n} \) that no possible walk can be misled by the adversary, even when all the unreliable memory locations are corrupted. Another difficulty is intrinsic in pointer-based structures: if the address stored in a pointer cannot be reliably recovered, the whole pointed object becomes inaccessible. The natural way to deal with this problem involves reading multiple copies of the pointer. However, since reading only \( o(\log n) \) copies would still allow the adversary to corrupt the pointer with a probability larger than \( \frac{1}{n} \), this would lead to a

\(^3\) The dictionary of [13] is randomized and the above bounds hold in expectation, while the one of [6] is deterministic. The dictionary of [13] can be made deterministic by slightly worsening the \( \delta \) additive term in the amortized complexity to \( \delta^{\epsilon+*} \), for a constant \( \epsilon > 0 \) of choice.
Resilient Dictionaries for Randomly Unreliable Memory

slowdown of $\Omega(\log n)$. We avoid this problem altogether by using a dynamic binary search tree (BST) that is nearly balanced, i.e., whose height is at most an additive constant larger than the optimal one [7]. This tree is then embedded into a static and pointer-free complete BST. We are now left with one final technical obstacle: since rebalancing such a dynamic tree following an update operation is more expensive w.r.t. its AVL counterpart, we need to employ a more elaborate 2-level buffering scheme. Namely, we group keys into $O\left(\frac{n}{\log n}\right)$ pages of capacity $O(\log n)$ which are in turn arranged into sorted folders consisting of $O(\log n)$ pages (i.e., $O(\log^2 n)$ keys) each.

We remark that the high probability bound on the correctness of our dictionary does not depend on the number of operations performed. In other words, there is a global probability of at least $1 - \frac{1}{n}$ that the operations in any (arbitrarily long) sequence of inserts, deletes, and searches, are all jointly correct. The following Theorem summarizes our result:

\textbf{Theorem 1.} A resilient dictionary with capacity $n$ can be constructed in $O(1)$ worst-case time. Search, and update operations (i.e., insert and delete) require $O(\log n)$ worst-case and $O(\log n)$ amortized time, respectively. The space required is $O(N)$, where $N$ is the number of elements currently stored in the dictionary. All operations performed during the lifetime of the dictionary are (jointly) correct with probability at least $1 - \frac{1}{n}$.

Although our focus is on the dictionary abstract data type, our techniques immediately extended to other data types (e.g., associative containers and priority queues), as we discuss in Section 6.

\textbf{Other related results}

Apart from the already mentioned results of [13] and [6], several other algorithms and models to deal with faulty computations have been proposed, with considerable attention paid to the problems of sorting and searching. In the faulty RAM model, the resilient sorting problem asks to output a permutation of an input set of $n$ elements such that the set of uncorrupted elements forms a sorted subsequence, while the resilient searching problem asks to locate an uncorrupted key $k$ in such a sorted sequence, if it exists. Roughly speaking, when $\delta$ corruptions happen, one can resiliently sort and search at the cost of an additional additive term in the time complexity that depends only on $\delta$. More specifically, one can search in $\Theta(\log n + \delta)$ time [6, 14] and sort in $O(n \log n + \delta^2)$ time which can be improved to $O(n + \delta^2)$ when elements are polynomially-bounded integers [12]. Moreover, for $\delta = \omega(\sqrt{n \log n})$, all resilient sorting algorithms must require $\omega(n \log n)$ time in the general case [14].

Another popular model dealing with faults considers the elements as opaque objects possessing an unknown linear order that needs to be discovered or approximated through pairwise noisy comparisons, i.e., comparisons that can sometimes be incorrect. If comparison errors happen randomly and resampling is allowed (i.e., a comparison can be repeated multiple times and the results are independent) then one can sort and search, w.h.p., in the optimal $O(n \log n)$ and $O(\log n)$ time, respectively [11]. If resampling is not allowed then one cannot reliably reconstruct the correct linear order. Nevertheless, it is possible to compute a permutation in which each element is misplaced by at most $O(\log n)$ positions, which is asymptotically optimal. Several solutions have been designed for this problem [5, 25, 20, 15, 16, 17] culminating in an optimal $O(n \log n)$ time algorithm. If errors are adversarial then $\Omega(n \log n + \delta n)$ comparisons are needed, and this is tight [3, 21, 22]. Adversarial errors in searching are studied in the context of Rényi-Ulam Games, in which a questioner needs to guess an element from a domain by asking comparison questions to a lying responder. An extensive collection of results exists, depending on the considered domain and on the constraints on the responder’s lies. We refer the interested reader to [23] and [9] for a survey and a monograph.
In [2] the authors define the fault-tolerance of a data structure as the maximum ratio between the amount of data lost as a consequence of a number $\delta$ of corruptions and $\delta$. They provide implementations of linked lists, stacks, and binary search trees with a fault-tolerance of $O(\log \delta)$ or $O(1)$, depending on the specific construction used, while losing only constant multiplicative factors in the time and space requirements. We remark that in [2], unlike both our dictionary and the ones in [13, 6], uncorrupted keys can also be lost until the whole data structure is reconstructed. If $\Theta(\delta)$ words of safe memory are available then, instead of the multiplicative overhead of [2], one can pay only an additive overhead of $\tilde{O}(\delta)$ in the time and space complexities, and the data structure can be reconstructed in linear time [8]. Similarly to [13] and [6], and differently from our dictionary, both [2] and [8] require the value of $\delta$ to be known in advance.

2 Preliminaries

For simplicity we assume that the capacity $n$ of our dictionary is a sufficiently large power of 2, so that $\log n$ is a sufficiently large integer and we do not have to deal with rounding. We also assume, w.l.o.g., that $p \leq \frac{1}{1024}$.

A concept of resilient variable similar to the one used in [13] will also be useful in the sequel. For our purposes, a resilient variable $x$ consists of $20\log n + 1$ consecutive copies $x_1, x_2, \ldots$ of a classical variable. A resilient variable can be written (i.e., assigned to) in $O(\log n)$ time and constant space by simply writing the intended value to each $x_i$. We can then read the value stored in $x$ either resiliently or non-resiliently: a non-resilient read amounts to returning the (possibly corrupted) value of a single $x_i$; a resilient read returns the majority value among those stored in all the copies $x_1, \ldots, x_{20\log n + 1}$, or fails if no majority value exists. Notice that, if at least $10\log n + 1$ copies are uncorrupted, the majority value coincides with the one previously stored in $x$ and can be computed in $O(\log n)$ time using a constant number of words in safe memory (see, e.g., the Boyer-Moore majority vote algorithm [4]). An easy Chernoff-bound argument shows that this is indeed the case for all possible read operations that can be performed, w.h.p, as the following Lemma states (a formal proof will appear in the full version of the paper).

Lemma 2. Suppose that the number of words used to store replicated variables is $O(n)$. With probability at least $1 - n^{-2}$, all resilient read operations return the previously stored value.

In the following we assume that all resilient read operations succeed as we will eventually union bound the success probability of the other operations in our dictionary with that of Lemma 2. Finally, we use $-\infty$ (resp. $+\infty$) to denote a key smaller than (resp. the largest among) all valid keys that can be inserted in our dictionary, and $\perp$ as a placeholder for some invalid key value.

The paper is organized in a bottom-up fashion: Section 3 describes pages, which are groups of $O(\log n)$ keys; In Section 4 we further group pages into sorted folders and show how to quickly locate pages, while Section 5 uses pages and folders to build our dictionary. In Section 6 we summarize our results and provide some concluding remarks.

4 For any constant $\epsilon > 0$ and $p \in \left(\frac{1}{1024}, \frac{1}{2} - \epsilon\right]$, we can simulate the required error probability by storing $\lceil 30\epsilon^{-2} \rceil$ copies of each word, with a strategy similar to the one described in the rest of this section.
3 Pages

Each page $v$ is associated with a contiguous interval $I(v) = (L_v, R_v]$ of keys and will be responsible for storing a set $K(v)$ of $O(\log n)$ keys, where all uncorrupted keys in $K(v)$ will belong to $I(v)$. A page $\pi$ is implemented as contiguous array $A(\pi)$ of capacity $6 \log n$ and three replicated variables storing: the endpoints $L_\pi$ and $R_\pi$ of $A(\pi)$, and the size $|K(\pi)|$ of $\pi$. The keys in $K(\pi)$ are be stored in the first $|K(v)|$ positions of $A(\pi)$ in an arbitrary order. If $|K(\pi)| = 0$ (resp. $|K(\pi)| = 6 \log n$) we say that $\pi$ is empty (resp. full).

Pages will support five basic operations, namely search, insert, delete, split, and merge, as detailed in the following.

Search. Given $k \in I(v)$, the search operation determines whether $k \in K(\pi)$. We simply resiliently read the variable $|K(v)|$ and compare $k$ with the first $|K(v)|$ elements of $A(\pi)$. If any (resp. no) element compares equal to $k$ we report that $k$ belongs to (resp. does not belong to) $\pi$. Clearly, this requires $O(\log n)$ worst-case time.

Insert. The insert operation adds a key $k \in I(\pi)$ to $K(\pi)$ and is only legal if $\pi$ is not full. We first search for $k$ and, if $k \in K(\pi)$, we are immediately done. Otherwise we write $k$ in the $|K(\pi)|$-th position of the array storing the set $K(v)$, where $|K(\pi)|$ is read resiliently. The time required is $O(\log n)$ in the worst-case.

Delete. The delete operation removes a key $k$ from $K(\pi)$. If $k \not\in K(\pi)$, then $\pi$ is unaltered. The operation requires $O(\log n)$ time: after a resilient read of $|K(v)|$, we perform linear search on $A(\pi)$ and, if $k$ is found in some position $i$, we swap the $i$-th and the $|K(v)|$-th element of $A(\pi)$ and decrement $|K(v)|$ by $1$ (which requires a replicated write operation). Due to corruptions, $A(\pi)$ might contain multiple occurrences of $k$, in which case exactly one of them is removed.

Split. The split operation destroys $\pi$ and returns (the addresses of) two newly created pages $\pi_1, \pi_2$ such that: (i) $I(\pi_1) \cap I(\pi_2)$ is the empty interval; (ii) $I(\pi_1) \cup I(\pi_2) = I(\pi)$; (iii) $|K(\pi_1)| = \lfloor |K(\pi)|/2 \rfloor$ and $|K(\pi_2)| = \lceil |K(\pi)|/2 \rceil$; (iv) $K(\pi_1) \cup K(\pi_2) = K(\pi)$.

The difficulty of the split operation lies in computing a key $x \in [L_\pi, R_\pi]$ that allows to split $I(v)$ into two sub-intervals $I(\pi_1) = (L_\pi, x]$ and $I(\pi_2) = (x, R_\pi]$ while preserving the bounds on the sizes of $\pi_1$ and $\pi_2$. We create a new page $\pi_1$ and we populate $A(\pi_1)$ by repeating the following iterative procedure $\lfloor |K(\pi)|/2 \rfloor$ times: in iteration $i$ we search for the smallest key $k_i \in K(v)$, we remove $k_i$ from $K(\pi)$ (in a way similar to our implementation of the delete operation), and we add it into the $i$-th position of $A(\pi_1)$. Finally, we rename $\pi$ as $\pi_2$, we choose $x = \min\{R_\pi, \max\{L_\pi, k_1, k_2, \ldots, \})$, and we and set $I(\pi_1)$ and $I(\pi_2)$ accordingly. It is easy to see that the split operation requires $O(\log^2 n)$ time in the worst case. The following lemma, whose proof is deferred to the full version of the paper, shows the uncorrupted keys are correctly partitioned among $\pi_1$ and $\pi_2$.

**Lemma 3.** Let $k$ be an uncorrupted key belonging to $\pi$ before the split operation. If $k \leq x$ (resp. $k > x$) then $k$ belongs to $K(\pi_1)$ (resp. $K(\pi_2)$) after the split operation.

Merge. The merge operation can be thought of as the opposite of the split operation. It requires two pages $\pi_1$ and $\pi_2$ as inputs such that (i) $I(\pi_1) \cap I(\pi_2)$ is the empty interval; (ii) $I(\pi_1) \cup I(\pi_2)$ is a contiguous interval, and (iii) $|K(\pi_1)| + |K(\pi_2)| \leq 6 \log n$; and produces the following effects: it adds all keys in $K(\pi_2)$ to $K(\pi_1)$, updates $I(\pi_1)$ to $I(\pi_1) \cup I(\pi_2)$, and destroys $\pi_2$. Notice that the order of $\pi_1$ and $\pi_2$ determines which of the two pages is destroyed.
Pages π₁ and π₂ are merged as follows: first we resiliently read |K(π₁)| and |K(π₂)|, and append the |K(π₂)| keys in π₂ to positions |K(π₁)| + 1, . . . , |K(π₁)| + |K(π₂)| of A(π₁). We then update |K(π₁)| to |K(π₁)| + |K(π₂)| and I(π₁) to (min{L₁, L₂}, max{R₁, R₂}) where L₁, L₂, R₁, R₂ are also read resiliently. One can easily check that the overall time complexity of a merge operation is O(log n) in the worst case.

4 Folders

Folders are sorted arrays of capacity 6log n in which elements are pages having pairwise disjoint intervals. Each folder F additionally stores two replicated variables holding the number |F| of pages currently in the array, and the overall number |K(F)| of keys stored in (the pages of) F. With a slight abuse of notation we say that π ∈ F if π is one of the pages of F. We denote by K(F) the union of all sets K(π) for π ∈ F. A folder F will always contain at least one page and, similarly to pages, it is (implicitly) associated with an interval I(F) = (L₁, R₁]. When F is first created it consists of a single innate empty page π with I(π) = I(F). In general, each π ∈ F is responsible for a certain sub-interval of I(F) (formally, the intervals I(π) are pairwise disjoint and satisfy ∪π∈FI(π) = I(F)). Moreover, all pages except the first one will always contain at least log n + 1 keys, and hence we immediately have that |K(F)| ≥ |(F)| · (log n + 1) and that a folder can always accommodate at least (6 log n − 1)(log n + 1) > 6 log² n keys. In fact, we guarantee that F will always contain at most 6log² n keys. Quite naturally, we say that a folder F is empty if it contains no keys, and full if it contains exactly 6log² n keys. If F is not empty, then its first page will also be non-empty.

The pages in the array of F are sorted w.r.t. the order relation < implicitly defined by the order of their intervals, i.e., we say that two pages π₁, π₂ are such that π₁ < π₂ iff I(π₁) precedes I(π₂) (π₁ ≤ π₂, π₁ > π₂, and π₁ ≥ π₂ are defined accordingly). As a consequence, there is no need to explicitly store I(F) since its left and right endpoints coincide with those of the first and last page in F, respectively.

We now discuss the operations supported by folders, most of which are similar to the ones for pages, on which they rely.

Locate and Search. Given a key k ∈ I(F), the locate operation returns the unique page π ∈ F such that k ∈ I(π). Once the locate operation is available, one can easily search for a key k ∈ I(F) by first locating the page π such that k ∈ I(π) and then searching for k in π.

Notice that the locate operation can be easily implemented in O(log² n) worst case time, w.h.p., by binary searching for the page π while reading all the replicated variables resiliently. We now show that it is possible to reduce the time complexity to O(log n) by using a technique similar to the one of [11], where an element in a sorted array A is located through a random walk over the contiguous sub-intervals induced by the elements in A. Such a random walk will observe inconsistent results independently at random and is allowed to backtrack when this happens. Moreover, to guarantee a high probability of success, Ω(log |A|) consistent observations are required before an element is returned. However, the analysis of [11] is not immediately suitable for our case due to two main obstacles, namely: (i) we want the high probability bound on the success probability to hold jointly for all locate operations performed during the lifetime of our dictionary, and (ii) since the adversary has complete control over the unreliable memory locations and a complete knowledge of the memory state, he can cause the inconsistent results to become correlated (e.g., by steering
the walk towards corrupted memory locations). We overcome these obstacles by relating all the problematic realizations of all possible random walks to a collection of paths in a suitable graph. We then show that, with high probability, no possible set of memory corruptions can cause any such path to become viable.

**The locate algorithm.** We start by considering an almost-complete binary tree $T$ in which the $i$-th leaf $\pi_i$ corresponds to the $i$-th page of $F$, and an internal node $v$ represents the (contiguous) interval $I(v) = (L_v, R_v]$ obtained by the union of all intervals $I(\pi)$ where $\pi$ is a leaf of the subtree $T_v$ of $T$ rooted at $v$. Then, we augment such a tree by appending, to each $\pi_i$, a path $P_\pi_i$ consisting of $20 \log n - d(\pi_i)$ copies of $\pi_i$ itself, where $d(\pi_i) \in \lfloor \log 6n \rfloor, \lceil \log 6n \rceil$ is the depth of vertex $\pi_i$ in $T$. See Figure 1 (a) for an example.

It is important to remark that $T$ does not need to be explicitly constructed since each vertex $v \in V(T)$ can be represented by a triple of integers $(i, j, h)$, where $i$ (resp. $j$) is the index, in the array of pages, of the smallest (resp. largest) leaf (i.e., page) in $T_v$, and $h$ is the depth of $v$ in $T$.

We now perform a discrete-time *walk* on $T$ as follows: initially the current vertex $v$ of the walk coincides with the root $r$ of $T$ and, at the generic $i$-th step, we *walk* from $v$ to one of its neighbors. More precisely, if $I(v)$ is obtained by the union of all the intervals $I(\pi)$ for $\pi^- \preceq \pi \preceq \pi^+$ we read the $i$-th copy $\ell$ (resp. $r$) of the replicated variable $L_{\pi^-}$ (resp. $R_{\pi^+}$), we check whether $\ell < k \leq r$ and, if that is not the case, we *walk* from $v$ to the parent of $v$ in $T$ (in the special case $v = r$ we “walk” from $v$ to itself). Otherwise, if $v$ has only one child $u$ in $T$, we *walk* from $v$ to $u$. Finally, if $v$ has two children $u_1$ and $u_2$ in $T$ where $I(u_1)$ precedes $I(u_2)$ and is obtained by the union of all the intervals $I(\pi)$ for $\pi^- \preceq \pi \preceq \pi^+$, we compare $k$ with the $i$-th copy $x$ of $R_{\pi^+}$. If $k \leq x$ we walk to $u_1$, otherwise we walk to $u_2$.

We continue the walk for $20 \log n$ steps so that we are guaranteed that all $\pi$, except possibly for the one reached after the last step, have at least one child. If the vertex $v$ reached at the end of the walk belongs to a path $P_\pi$, we return $\pi$. We say that the locate operation *fails* if either the $v$ does not belong to any path $P_\pi$, or if $k \notin I(\pi)$.

---

$^5$ A binary tree of height $h$ is almost-complete if it is complete up to the $(h - 1)$-th level. Moreover, we also assume that all the leaves on the $h$-th level are left-justified.
Analysis. Let us consider the following related problem. We are given a directed acyclic graph $G$ having maximum out-degree at most $\Delta \geq 2$, a vertex $s \in V(G)$, and a probability $\rho \leq \frac{1}{128\Delta - 1}$. Moreover, each non-sink vertex $v \in V(G)$ has exactly one preferential outgoing edge $(v, v^*) \in E(G)$ and we define $E^*$ as the set of all preferential edges. The vertices of $G$ are independently colored either red or blue with probability at most $\rho$ and at least $1 - \rho$, respectively. Finally, we say that a path $P = \langle v_0, v_1, \ldots, v_k \rangle$ in $G$ is traversable if, for every $v_i \in P$ with $i = 0, \ldots, k - 1$, we have that $v_i$ is red, or $(v_i, v_{i+1})$ is preferential, or both. See Figure 1 (b) for an example. We are now ready to state the following:

\begin{lemma}
Let $\ell$ be a multiple of 4. The probability that $G$ contains a traversable path $P$ of length $\ell$ from $s$ such that $|E(P) \cap E^*| \geq \frac{\ell}{4}$ is at most $2^{-\frac{\ell}{4} + 1}$.
\end{lemma}

\begin{proof}
We will count the number of possible paths of length $\ell$ from $s$ by grouping them according to the number of non-preferential edges they use. More precisely, we say that the type of a path $P = \langle s = v_0, v_1, \ldots, v_k \rangle$ in $G$ is $\kappa$ if $|\{v_i^* \neq v_{i+1} : i \in \{0, \ldots, \ell - 1\}\}| = \kappa$, where $(v_i, v_i^*)$ is the unique preferential edge outgoing from $v_i$. Notice that each path of type $\kappa$ can be described by a set of $\kappa$ pairs: $\{(\tau_1, a_1), \ldots, (\tau_\kappa, a_\kappa)\}$ where $(\tau_i, a_i)$ signifies that when the path reaches $v_{\tau_i}$, it continues towards the $a_i$-th non-preferential outgoing edge from $v_{\tau_i}$ (according to some fixed arbitrary order of the edges). This immediately implies that there are at most $\binom{\ell}{\kappa}(\Delta - 1)^\kappa$ paths of type $\kappa$. Moreover, for any such path to be traversable, it must happen that all the vertices $v_i$ with $i \in \{\tau_1, \ldots, \tau_\kappa\}$ are red (while the remaining vertices can be either blue or red). This happens with probability at most $\rho^\kappa$, and hence the probability that there exists at least one traversable path of type $\kappa$ is at most $\binom{\ell}{\kappa}(\Delta - 1)^\kappa \rho^\kappa$. By summing over all $\kappa \in [\ell/4, \ell]$ we obtain:

$$
\sum_{\kappa=\ell/4}^{\ell} \binom{\ell}{\kappa}(\Delta - 1)^\kappa \rho^\kappa \leq \left( \sum_{\kappa=\ell/4}^{\ell} \binom{\ell}{\kappa} \right) \left( \sum_{\kappa=\ell/4}^{\infty} (\Delta - 1)^\kappa \rho^\kappa \right) \leq \frac{2^\ell (\Delta - 1)^{\ell/4} \rho^{\ell/4}}{1 - (\Delta - 1)\rho} < 2^{-\frac{\ell}{4} + 1}.
$$

\end{proof}

\begin{lemma}
With probability at least $1 - n^{-3}$ no locate operation on folders fails.
\end{lemma}

\begin{proof}
We relate our random walk on $T$ to the traversable paths in a suitable graph $G$. Let $\ell$ be the unique leaf of $T$ such that the key $k$ to locate belongs to the corresponding page $\pi$ and direct each edge of $T$ towards $t$. We define $G$ as the graph whose vertex set consists of $\ell + 1$ copies $u^{(1)}, u^{(2)}, \ldots$ of each vertex $u \in V(T)$ and whose edge set contains, for all $i = 1, \ldots, \ell$, the edge $(r^{(i)}, r^{(i+1)})$ and all the edges $(u^{(i)} v)^{(i+1)}$ where $(u, v)$ is a directed edge in $T$. We color a vertex $u^{(i)}$ of $G$ red if the $i$-th copy of at least one of the (at most 3) replicated variables accessed when a step of the random walk is performed from vertex $u^{(i)}$ is in an unreliable memory location (notice that these replicated variables correspond to the page associated with $u$ and, possibly, with a child of $u$ in $T$). For each non-sink vertex $u^{(i)}$ of $G$ we let $(u, v)$ be the unique (directed) edge outgoing from $u$ in $T$ and we choose the edge from $u^{(i)}$ to $v^{(i+1)}$ as the preferential edge from $u^{(i)}$ in $G$.

The graph $G$ has maximum out-degree 3 and hence, once we choose $\Delta = 3$, $s = r$, $\rho = \frac{1}{128\Delta - 1} > 3p$, and $\ell = 20 \log n$, we can invoke Lemma 4 to conclude that all the traversable paths from $s$ in $G$ of length $20 \log n$ use less than $5 \log n$ non-preferential edges with probability at least $1 - 2n^{-5}$. Since any sequence of vertices $(r = v_1, v_2, \ldots, v_{\ell+1})$ corresponding to a realization of our random walk induces a traversable path $P = \langle s = v_0^{(1)}, v_2^{(2)}, \ldots, v_{\ell+1}^{(\ell+1)} \rangle$ in $G$ we know that, with the aforementioned probability, all possible random walks on $T$ will use at least $15 \log n$ edges directed towards $t$. Each such edge decreases the distance in $T$ from the current vertex of the walk to $t$ by at least 1, while any other edge increases such a distance by at most 1. Since the distance $d_T(r, t)$ from $r$ to $t$ in $T$ is $20 \log n$ by
Resilient Dictionaries for Randomly Unreliable Memory

construction, we conclude that the final vertex $v$ of the walk must satisfy $d_I(v, t) \leq 10 \log n$ or, in other words, $v$ lies on $P_x$ and the locate operation is successful. The claim follows by noticing that, once the locations of the unreliable words are fixed, at most $O(n \log^2 n)$ distinct (colored) graphs $G$ can exist, namely those corresponding to the $O(\log n)$ possible different sizes of $T$, to the $O(\log n)$ different positions of the sought leaf $t$ in $T$, and to the $O(n)$ memory locations at which a folder can be stored.

**Insert.** The insert operation adds a key $k \in I(F)$ into $K(F)$, and it is only legal when $F$ is not full. We first locate the page $\pi$ responsible for $k$ and we say that the insert operation targets $\pi$. Then, if $\pi$ is not full, we simply insert $k$ into $\pi$, we update $|K(F)|$, and we are done. Otherwise, we say that the insert operation is expensive. Let $i$ be the position of $\pi$ in the array of pages. We move all pages in positions $i + 1, \ldots, |F|$ one position to the right (so that the page originally in position $j$ is now in position $j + 1$), and increment $|F|$ by 1. Every read and write operation on the replicated variables of the moved pages is performed resiliently, i.e., a replicated variable is moved by first resiliently reading its value from the old memory location and then resiliently writing $x$ to the new memory location. We now split $\pi$ into two pages $\pi_1$ and $\pi_2$ containing $3 \log n$ keys each, and we store them in positions $i$ and $i + 1$, respectively. Finally, we insert $k$ into the unique page $\pi' \in \{\pi_1, \pi_2\}$ such that $k \in I(\pi')$, and we update $|K(F)|$.

Overall, an insert operation requires $O(\log n)$ worst-case time if it is not expensive and $O(\log^2 n)$ worst-case time otherwise.

**Delete.** The delete operation removes a key $k$ from $K(T)$ if $k \in K(T)$. If $k \notin K(T)$ the operation does nothing. We first locate the page $\pi$ responsible for $k$ and we say that the delete operation targets $\pi$. We now try to delete $k$ from $\pi$ and, if $k$ was not found in $K(\pi)$ we are already done. If $k$ is found, then we decrement $|K(F)|$ by 1 and proceed differently depending on the size and on the position of $\pi$ in the pages array of $F$. Whenever, after the deletion, any of the following conditions is true, no further work is necessary: (D1) $|K(\pi)| > \log n$; or (D2) $\pi$ is the only page of $F$; or (D3) $\pi$ is the first page of $F$ and is non-empty. Otherwise, we say that the delete operation is expensive and we distinguish two cases. If (i) $\pi$ is the first page of $F$, (ii) $F$ has at least 2 pages, and (iii) $\pi$ is empty, we let $\pi'$ be the second page in $F$ and we delete $\pi$ from $F$ as follows: We set the left endpoint of $I(\pi')$ to the left endpoint of $I(\pi)$ (effectively updating $I(\pi')$ to $I(\pi) \cup I(\pi')$), and we shift every page of $F$ other than $\pi$ one position to its left, thus overwriting and destroying $\pi$. Finally, we decrement the value of $|F|$ by 1.

The complementary case is the one in which $|K(\pi)| \leq \log n$ and $\pi$ is not the first page of $F$, which implies that $|K(\pi)| = \log n$. We let $\pi'$ be the page preceding $\pi$ in $F$ and we further distinguish two sub-cases depending on the value of $|K(\pi')|:

- **If $K(\pi') \leq 4 \log n$.** We merge $\pi'$ and $\pi$. Since this operation destroys $\pi$, we shift all pages $\pi'' \in F$ such that $\pi'' > \pi$ by one position to their left, and we decrement $|F|$ by 1. Notice that $\pi'$ now contains at most $4 \log n + \log n = 5 \log n$ keys.

- **If $K(\pi') > 4 \log n$.** We first split $\pi'$ (destroying it) into $\pi_1$ and $\pi_2$. We then merge $\pi_2$ and $\pi$ (destroying $\pi$). We store $\pi_1$ in place of $\pi'$ and $\pi_2$ in place of $\pi$. Notice that $\pi_1$ now contains at least $2 \log n$ and at most $3 \log n$ keys, while $\pi_2$ contains at least $2 \log n + \log n = 3 \log n$ and at most $3 \log n + \log n = 4 \log n$ keys.

---

6 Notice that, before the insert operation, we necessarily had $|F| \leq 6 \log n$ as otherwise $|K(F)|$ is at least $(6 \log n - 1)(\log n + 1)$, which violates our invariant on the number of keys stored in $F$. 

Similarly to the insert operation, when pages are moved in the array of $F$, all their resilient variables are read and written resiliently. Overall, the required worst-case time is $O(\log n)$ if the operation is not expensive, and $O(\log^2 n)$ otherwise.

**Split.** If $|K(F)| \geq 12 \log n$, the split operation destroys $F$ and returns two new folders $F_1$ and $F_2$ that together contain the same set of keys of $F$ and such that: $I(F) = I(F_1) \cup I(F_2)$, $I(F_1) \cap I(F_2) = \emptyset$, and $|K(F)|/2 \leq |K(F_1)| < |K(F)|/2 + 6 \log n$ (implying $|K(F)|/2 - 6 \log n < |K(F_2)| \leq |K(F)|/2$), and all the uncorrupted keys $k \in K(\pi)$ belong to the unique folder $F' \in \{F_1, F_2\}$ such that $k \in I(F')$.

To implement the operation, we look for the first page $\pi$ such that $\sum_{\pi' \preceq \pi} |K(\pi')| \geq |K(F)|/2$. We then construct the folders $F_1$ and $F_2$, where $T_1$ contains all the pages $\pi' \preceq \pi$, and $T_2$ contains all the pages $\pi' \succ \pi$. Notice that the condition $|K(F)| \geq 12 \log n$ ensures that $F_2$ will always contain at least one page from $F$. All involved variables are read and written resiliently.

Overall, the split operation can be performed in $O(\log^2 n)$ time in the worst-case.

**Merge.** The merge operation takes two non-empty folders $F_1$ and $F_2$ such that $|F_1| + |F_2| \leq 6 \log^2 n$, $I(F_1)$ precedes $I(F_2)$, and $I(F_1) \cup I(F_2)$ is a contiguous interval, and returns a new folder $F$ containing the keys in $K(F_1) \cup K(F_2)$ and such that $I(F) = I(F_1) \cup I(F_2)$. The operation destroys $F_1$ and $F_2$.

We first handle some corner cases that only happen if $F_1$ contains a single page $\pi$. If this is the case and $|K(\pi)| \leq 2 \log n$ also holds, then we let $\pi'$ be the first page of $F_2$ and proceed as follows:

- If $|K(\pi)| + |K(\pi')| \leq 5 \log n$, merge $\pi'$ and $\pi$ (destroying $\pi$), and return $F_2$ (where $I(F_2)$ and $|K(F_2)|$ are suitably updated);
- Otherwise, if $|K(\pi')| \geq 4 \log n$, we split $\pi'$ into $\pi_1$ and $\pi_2$, we merge $\pi_1$ and $\pi$ (so that $\pi$ is destroyed), and we return a new folder consisting of $\pi_1$, $\pi_2$ and all pages other than $\pi'$ in $F_2$. Both $\pi_1$ and $\pi_2$ contain between $2 \log n$ and $5 \log n$ keys.
- In the remaining case, we have that $5 \log n < |K(\pi)| + |K(\pi')| < 6 \log n$. We, first merge $\pi$ and $\pi'$, then we split $\pi$ into $\pi_1$ and $\pi_2$, each containing more than $2 \log n$ and at most $3 \log n$ keys. We return the folder consisting of $\pi_1$, $\pi_2$ and all pages other than $\pi'$ in $F_2$.

We now suppose that we are not in any of the above cases, and preprocess $F_1$ and $F_2$ to ensure that the leftmost page $\pi'$ of $F_2$ will always contain at least $2 \log n$ keys. If that is not already the case, we let $\pi$ be the last page of $F_1$ and modify $F_1$ and $F_2$ as follows:

- If $|K(\pi)| + |K(\pi')| \leq 5 \log n$ we merge $\pi'$ and $\pi$. We decrement $|F_1|$ to account for the destroyed page $\pi$ (this might cause $F_1$ to temporarily become empty).
- If $|K(\pi)| \geq 4 \log n$ we first split $\pi$ into $\pi_1$ and $\pi_2$ and then merge $\pi_2$ and $\pi'$. We store $\pi_1$ and $\pi_2$ in place of the, now destroyed, pages $\pi$ and $\pi'$ in $F_1$ and $F_2$, respectively.
- Otherwise we must have $5 \log n < |K(\pi)| + |K(\pi')| < 6 \log n$. We first merge $\pi$ and $\pi'$, then we split $\pi$ into $\pi_1$ and $\pi_2$. We store $\pi_1$ and $\pi_2$ in place of $\pi$ and $\pi'$ in $F_1$ and $F_2$, respectively.

The rest of the operation is now straightforward: we simply create a new folder $F$ by concatenating all pages in $F_1$ with those in $F_2$ and set $|F| = |F_1| + |F_2|$, $|K(F)| = |K(F_1)| + |K(F_2)|$, and $I(F) = I(F_1) \cup I(F_2)$. We remark that the involved pages are simply moved from their previous folder to the new folder $F$, rather than being destroyed and recreated anew. Overall, the merge operation requires $O(\log^2 n)$ time in the worst-case.
Amortized analysis

The following lemma, whose proof is omitted, summarizes the time complexities of the folder’s operations and shows that the $O(\log^2 n)$ time needed by the expensive insertions and deletions can be amortized over $\Omega(\log n)$ non-expensive operations.

Lemma 6. An empty folder can be constructed in $O(\log n)$ worst-case time. Each search operation can be performed in $O(\log n)$ worst-case time. Merge and split operations requires $O(\log^2 n)$ worst-case time. Each insert or delete operation requires $O(\log n)$ amortized time.

5 Our dictionary

We store the dictionary keys into $\eta = O(\frac{n}{\log^2 n})$ folders, such that (i) each folder contains at most $6\log^2 n$ keys and (ii) all but at most one folder contain at least $\log^2 n + 1$ keys. These folders are logically organized into a dynamic nearly-balanced binary search tree $T$, in which each vertex $v_F$ is associated with a folder $F$ and consists of three resilient variables, namely: and a pointer containing the address of folder $F$, and two variables $L_{v_F}$ and $R_{v_F}$ defining an interval $I(v_F) = (L_{v_F}, R_{v_F}]$ where $L_{v_F}$ (resp. $R_{v_F}$) is the leftmost (resp. rightmost) endpoint among of all the intervals of the folders associated with the descendants of $v_F$ in $T$ (including $v_F$ itself). The intervals of the stored folders will be pairwise disjoint and will cover the whole range $(-\infty, +\infty]$. An empty dictionary consists of a single empty folder $F$ with $I(F) = (-\infty, +\infty]$.

As shown in [7], an embedding of a dynamic binary search tree with $N \geq 1$ vertices into a static complete binary tree of height $H(N) = \lceil \log(2N + 1) \rceil - 1$ can be maintained under insertion and deletion of vertices in $O(\log^2 N)$ amortized time per operation, assuming that vertices can be accessed and copied in constant time. For the sake of completeness we briefly sketch the construction of [7]. Let $\rho(u)$ be the ratio between the number of vertices of the subtrees rooted at $u$ in the dynamic and in the static trees, respectively. The root $r$ will always satisfy $\frac{1}{2} \leq \rho(r) \leq \frac{2}{3}$. This is initially true when the dynamic tree consists of single vertex since $H(1) = 1$ and $\rho(r) = \frac{1}{2}$. Whenever an insertion or deletion needs to be performed, we first check whether it will causes the value of $H(N)$ to change and, if this is the case, the operation is handled by rebuilding the updated dynamic tree in a perfectly balanced fashion. The remaining updates are as follows: insertions are performed as in an ordinary binary search tree except when they cause the height of the newly inserted vertex $u$ to exceed $H(N)$. In this case the subtree rooted at the lowest ancestor $v$ of $u$ such that $\frac{1}{3} - \frac{d(v)}{2^H(N)} \leq \rho(v) \leq \frac{1}{2} + \frac{d(v)}{2^H(N)}$ is rebuilt to be perfectly-balanced (recall that $d(v)$ is the depth of vertex $v$). When a vertex $u$ needs to be deleted, it is first pushed down the dynamic tree by iteratively swapping it with either its predecessor or its successor until it becomes a leaf. Then, $u$ is deleted and the subtree rooted on the lowest vertex $v$ that was ancestor of $u$ and that satisfies $\frac{1}{3} - \frac{d(v)}{2^H(N)} \leq \rho(v) \leq \frac{1}{2} + \frac{d(v)}{2^H(N)}$ is rebuilt. The current number of vertices of the dynamic tree is explicitly stored (which allows to compute $\rho(r)$ in constant time) while no additional information needs to be maintained for the values $\rho(v)$ for $v \neq r$ as the complexity needed to compute them (e.g., with a DFS visit of the subtree rooted in $v$) is subsumed by that of the rebalancing step.

We restated the result of [7] using the standard definitions of depth and height employed in this paper. We also fixed the parameters $\tau_1$, $\gamma_2$, and $\gamma_H$ of [7] to $\frac{1}{2}$, $\frac{1}{8}$, and $\frac{1}{16}$ respectively.
Since, in our case, the number of nodes \( \eta \) in \( T \) can be at most \( 1 + \left\lfloor \frac{n}{\log n + 1} \right\rfloor \), we have that \( H(\eta) \leq \log(2\eta + 1) \leq 2 + \log \frac{n}{\log n} \). We therefore embed \( T \) into a static tree \( T \) which, in turn, is implicitly stored in a positional array of \( 2^{H(\eta)+1} - 1 \leq 8 \frac{n}{\log n} \) elements. In this way, given the address of any node of \( T \), we can compute the addresses of its left child, right child, and parent in constant time. Overall, the space required to store \( T \) is at most \( 8 \frac{n}{\log n} \cdot O(\log n) = O(\frac{n}{\log n}) \). Since the resilient variables of a vertex \( v \in V(T) \) can be read and written in \( O(\log n) \) time, we are able to insert and delete vertices from \( T \) in \( O(\log^2 \eta) \cdot O(\log n) = O(\log^3 n) \) amortized time.\(^8\) We distinguish the vertices \( v \) that belong in our static tree \( T \) but not in our logical tree \( T \) by setting their intervals \( I(v) \) to the bogus range \((\bot, \bot]\) and their pointers to some arbitrary address (see Figure 2 for an example). Notice that, when a subtree is rebalanced, the intervals of the affected nodes also need to be recomputed. The same applies to the ancestors of a deleted or newly inserted vertex. In both cases the required time complexity is \( O(\log n) \) per vertex and, since the tree is nearly balanced, it is subsumed by the amortized cost of the operation. We store \( \eta \), the number of keys currently in the dictionary, and the address of the array representing \( T \) as global variables in safe memory.

Implementing the dictionary operations

Using a technique similar to the one described in Section 4 to locate a page in a folder, we are able to locate the folder \( F \) responsible for a given key \( k \) in our tree \( T \) (along with the corresponding node \( v_F \)).

\textbf{Lemma 7.} With probability at least \( 1 - n^{-3} \) no locate operation on the dictionary fails.

Once the locate operation is available, the search operation can be easily implemented by first locating the (unique) folder \( F \) in \( T \) such that \( k \in I(F) \) and searching for \( k \) in \( F \). Moreover, we also use the locate operation to implement the insert and delete operations of our dictionary, together with a suitable strategy for splitting and merging the folders of

\(^8\) Here the amortization is performed over all insertion and deletion of vertices in \( T \). In the sequel we will derive an amortized bound w.r.t. insertions and deletions of keys in our dictionary.
when they become full or contain less than $1 + \log^2 n$ keys. Due to space limitations, a formal description of these operation and the proof of the above lemma are deferred to the full version of this paper.

**Space complexity**

Our dictionary as described so far achieves all the bounds of Theorem 1 except for the one regarding its space complexity, which would be $O(\log^2 n + N)$ instead of the promised $O(N)$, where $N$ is the number of keys in the dictionary. The missing range $N = o(\log^2 n)$ can be handled by lazily building the first page and the first folder of our dictionary, and by employing a suitable halving/doubling strategy [10] to ensure that their size is always linear in the number of keys stored therein. A complete proof of Theorem 1 will appear in the full version of the paper.

**6 Conclusions**

We presented a resilient dictionary that is able to operate correctly on all uncorrupted keys, uses the optimal amount of space, and whose operations require $O(\log n)$ amortized time. All operations are deterministic, i.e., their execution is completely determined by their input and by the observed memory contents. The same construction can also be used if satellite data is attached to the keys, in which case the search operation returns either the data associated with a given uncorrupted key $k$ or that of a key that equals $k$ following a corruption. Moreover, one can easily implement other commonly used operation on BSTs, e.g., we can report all the $N$ stored keys in $O(N)$ worst-case time, find the predecessors or successors in $O(\log n)$ worst-case time, support range queries in $O(\log n + t)$ time where $t$ is the size of the output, and find (resp. extract) the minimum/maximum element in $O(\log n)$ worst-case (resp. amortized) time.

Our data structure can aid in the design of other resilient algorithms and, sometimes, can even be used as a drop-in replacement of corresponding non fault-tolerant implementations. For example, when our dictionary is augmented with the find-min operation described above, it can substitute the heap in the classical heapsort algorithm, which then immediately solves the resilient sorting problem in our error model using $O(n \log n)$ worst-case time.

Finally, we observe that our approach can be combined with a constant replication scheme to ensure that, for any constant $c \geq 1$ of choice, at most $\delta/c$ keys are lost when $\delta$ words are corrupted. In addition, if we slightly worsen the construction time to $O(\log n)$, the time required to report the $N$ stored keys to $O(N + \log n)$, and the the space requirements to $O(N + \log n)$, then our results also hold even when no definitely reliable memory words exist, except for $O(1)$ temporary registers that can only be used to hold intermediate computation results (i.e., whose contents are invalidated at the beginning/end of each dictionary operation).

**References**


