Lossless Fault-Tolerant Data Structures with Additive Overhead

Paul Christiano, Erik D. Demaine⋆, and Shaunak Kishore

MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar St., Cambridge, MA 02139, USA, {paulfc,edemaine,skishore}@mit.edu

Abstract. We develop the first dynamic data structures that tolerate δ memory faults, lose no data, and incur only an $\tilde{O}(\delta)$ additive overhead in overall space and time per operation. We obtain such data structures for arrays, linked lists, binary search trees, interval trees, predecessor search, and suffix trees. Like previous data structures, δ must be known in advance, but we show how to restore pristine state in linear time, in parallel with queries, making δ just a bound on the rate of memory faults. Our data structures require $\Theta(\delta)$ words of safe memory during an operation, which may not be theoretically necessary but seems a practical assumption.

1 Introduction

As computer memory systems increase in size, complexity, and density, so does the chance that some bit flips during the lifetime of a computation or database. A survey of practical studies [14] concludes that 1,000–5,000 *soft errors* (not caused by permanent hardware failure) per billion hours per megabit is typical on modern memory devices. On a modern PC with 24 gigabytes of memory, this rate would imply one failure roughly every one to five hours. A recent study on production Ask.com servers [12] suggests that the observed rate of soft errors on ECC SDRAM is only 0.56 per billion hours per megabit, or roughly one failure per year.

While these failure rates are reasonably small, in a complex data structure, a single bit error can be catastrophic. For example, corrupting a single pointer can make most of a data structure unreachable, losing data and likely causing the system to crash. A natural problem, considered over the past 15 years, is to design data structures that do not crash after a small number of memory faults. Here we aim for the stronger property of losing none of the data in the structure, preventing any higher-level algorithm from crashing. More precisely, we develop data structures that tolerate a reasonable rate of memory failures while remaining correct and efficient.

⋆ Supported in part by MADALGO — Center for Massive Data Algorithmics — a Center of the Danish National Research Foundation.

Model. Our model of failure-prone computation is the *faulty RAM* of [11]. At any time, an adversary can corrupt the value of any word in the random-access memory to an arbitrary value, provided that the total number of corrupted words remains at most a known parameter δ. A *fault-tolerant* algorithm nonetheless reports the same result as if no words were changed; if all operations are fault tolerant, we call the data structure *fault tolerant* or *lossless*.

In addition to the main memory, we assume a *working store* of memory that cannot fail (e.g., CPU registers and level-1 cache) during the execution of a single operation. The working store is temporary, and cannot be used to store actual data between operations. This ephemerality means that the model supports multiple data structures with a single working store (CPU hardware). Naturally, our results also hold if there is no working store, but the adversary can modify words only between operations and not during (e.g., external storage media that might be damaged between uses).

While we state results in terms of an upper bound δ on the total number of faults, our data structures in fact support *live recovery.* In $O(\delta)$ time, we can examine a block of $O(\delta)$ consecutive memory locations, correct all errors within the block, restoring it to the state as if the errors never occurred. In this way, the recovery process can proceed in parallel with other operations with minimal locking, can be parallelized over multiple threads, and performs well when memory transfers occur in blocks. Thus δ really becomes an upper bound on the number of faults that can occur between passes of a background linear-time recovery process.

Our results. We develop the first fault-tolerant data structures that are simultaneously lossless and dynamic, as well as the first fault-tolerant data structures that are simultaneously lossless and efficient, requiring only an additive $\tilde{O}(\delta)$ overhead in overall space and per-operation time. Furthermore, we show that our techniques apply to a wide family of dynamic data structures: arrays, linked lists, binary search trees, interval trees, predecessor structures, and suffix trees. By contrast, previous work considered only dictionaries supporting membership, linked lists, and static problems; and either lost data from memory errors, or was static and required a lengthy recovery process.

Table 1 summarizes our results and how they compare to previous work. All of the problems we consider have a trivial $\Omega(\delta)$ lower bound on query time and overall space: with $o(\delta)$ time, everything examined can be corrupted, and with $o(\delta)$ space, some information must be lost. Hence, other than our interval trees which have an additional $\Theta(\log \delta)$ factor, our time and space bounds are optimal. The only suboptimal part is the requirement of $\Theta(\delta)$ words of working store. Improving this bound would require an advance in decoding error-correcting codes of size $\Theta(\delta)$ using failure-prone intermediate results.

Our data structures require $\delta^{O(1)}$ preprocessing time at the very beginning to construct the necessary error-correcting codes. The dynamic interval tree data structure (Section 5.1) uses randomization to solve stabbing queries, making them prone to reporting corrupted results. Assuming an oblivious adversary, each stabbing query is guaranteed to succeed with probability $1 - \varepsilon$ by spend-

Problem	Loss/ error	Time/op. overhead	Space overhead	Working store	Model	Ref.
Dynamic linked list	0	$+O(\delta)$	$+O(\delta)$	$O(\delta)$	RAM	2.7
Dynamic hash table	0	$+O(\delta)$	$+O(\delta)$	$O(\delta)$	RAM	2.8
Dynamic predecessor	0	$+O(\delta)$	$+O(\delta)$	$O(\delta)$	RAM	3.1
Dynamic binary search tree	0	$+O(\delta)$	$+O(\delta)$	$O(\delta)$	RAM	3.1
Dynamic interval tree	0	$+O(\delta \lg \delta), \times \lg \frac{1}{\varepsilon}$	$+O(\delta \lg \delta)$	$O(\delta)$	RAM	5.1
Suffix tree construction	0	$+O(\delta)$	$+O(\delta)$	$O(\delta)$	RAM	4.1
Dynamic linked list	$O(\delta)$	$O(1)$	$O(1)$	$O(\delta)$	pointer	[2]
Dynamic binary search tree	$O(\delta)$	$O(1)$	$O(1)$	$O(\delta)$	pointer	[2]
Static sorting	1	$+O(\delta)$	$+O(\delta)$	$O(1)$	RAM	[9]
Static search	1	$+\delta$	$+O(\delta)$	$O(1)$	RAM	[9]
Dynamic search tree	1	$+O(\delta)$	$+O(\delta)$	$O(1)$	RAM	[8]
Dynamic hash table	1	$+O(\delta)$	$+O(\delta)$	$O(1)$	RAM	[8]
Dynamic priority queues	1	$+O(\delta)$	$+O(\delta)$	$O(1)$	RAM	[3]
Static dictionary, $\delta = O(n)$	0	$O(1)$	$2^{2^{O(\sqrt{\lg n})}}$	$O(1)$	cell probe	[6]

Table 1. Summary of new and previous on fault-tolerant data structures. All time bounds are amortized. Here δ denotes the upper bound on the number of faults, and ε denotes query failure probability (guaranteed to be 0 except where ε appears in bounds). Quantities (including δ) are measured in words (elements).

ing a factor of $O(1/\varepsilon)$ in time. We analyze according to an adaptive adversary (following [11]), which enables the adversary to corrupt any δ additional queries.

Related work. Finocchi and Italiano [11] introduced the faulty-memory RAM model of failure-prone computation that we use, which led to the bulk of the research in fault-tolerant data structures [9,8,3]. All of the results in this line, however, tolerate the loss of one data element per fault, replacing it with an arbitrary corrupted value. The idea behind this tolerance was to obtain better bounds—additive instead of multiplicative $\tilde{O}(\delta)$ overheads—by avoiding replication. (Hence the conference version of [11] had a title ending "…(without redundancy)".). Our data structures show, however, that it is possible to never lose data yet keep additive $\tilde{O}(\delta)$ overheads.

Before the work in the RAM model, Aumann and Bender [2] considered analogous problems in the pointer machine. Here it is necessary to assume that $O(\delta)$ nodes of the structure cannot be corrupted, in order not to lose the entire structure. Furthermore, because the model essentially prevents low-level manipulation of data, without replication pointer-machine data structures necessarily lose data from each error.

The only other work that considers lossless data structures is by de Wolf [6]. His results are all static, and rely on locally decodable codes, for which an optimal trade-off between local decodability and total space is an open problem. As far as we know, the best known upper bound on space for a static dictionary is superpolynomial in n, as indicated in Table 1.

2 Fundamental Techniques

Previous results in fault-tolerant data structures use one of two techniques to store data in faulty memory: data replication or error-correcting codes. On the one hand, a single word can be reliably stored by maintaining $\Theta(\delta)$ copies—the *resilient variables* of [10]. A resilient variable can be accessed quickly, but requires $\Theta(\delta)$ multiplicative space overhead. On the other hand, δ words can be reliably stored in an error-correcting code of size $\Theta(\delta)$. Although error-correcting codes reduce the space overhead, reading a single word from an error-correcting code requires $\Theta(\delta)$ time. Our main contribution is the introduction of data structures that combine these two techniques to achieve simultaneous space and time efficiency.

2.1 Blocked Fault-Tolerant Data Structures

Our data structures consist of "fault-tolerant blocks", which interpolate gracefully between the performance of resilient variables and error-correcting codes. To construct fault-tolerant blocks, we use the linear-time encodable and decodable error-correcting codes due to Spielman [13]. A concise statement of some of their results is the following:

Theorem 2.1. [13] *In $k^{O(1)}$ time, we can compute an $O(k)$-size description C_k of an error-correcting code which can be used as input to encoding and decoding algorithms E and D. Given a string x of k words, $E(x, C_k)$ encodes x into a string y of $O(k)$ words such that any string z differing from y on at most k words can be decoded: $D(z, C_k) = x$.*

We call k the *multiplicity* of the error-correcting code. For our purposes, this quantity is both the number of words stored by the code and the maximum number of word errors the code corrects. We will always set the multiplicity k to a power of 2 between 1 and δ. When creating any fault-tolerant data structure, we precompute C_k for each of these values of k. Because these descriptions have total size $O(\delta)$, we can store them all in safe memory. This precomputation takes $\delta^{O(1)}$ time, constituting the preprocessing needed by our data structures.

A *fault-tolerant block of multiplicity* k stores a k-word string x in $\Theta(\delta)$ space by storing $\lceil \frac{2\delta+1}{k} \rceil$ copies of the error-correcting code $E(x, C_k)$ contiguously in memory. Our data structures will call for fault-tolerant blocks of varying multiplicities. When a small amount of data must be accessed frequently, we will store it in fault-tolerant blocks of constant multiplicity; in this limit, fault-tolerant blocks reduce to resilient variables. When a large amount of data must be accessed infrequently, we will store it in fault-tolerant blocks of multiplicity $\Theta(\delta)$; in this limit, at least in spirit, a fault-tolerant block reduces to a single error-correcting code which corrects δ errors. In between these two extremes, we obtain useful trade-offs between space and time costs.

The data in a fault-tolerant block cannot be corrupted by fewer than $\delta + 1$ word errors. The value stored in an error-correcting code in a fault-tolerant block

of multiplicity k can only be changed if more than k of its words are corrupted. Thus the total multiplicity of all corrupted codes is at most δ, while the total multiplicity of all codes is at least $2\delta + 1$. It follows that the value stored in a fault-tolerant block can be correctly recovered in time $O(\delta)$ by decoding all of its error-correcting codes and taking the majority value. We can remove all errors from a fault-tolerant block by safely reading its value in this way and then writing a new fault-tolerant block which stores that value without errors.

A *blocked fault-tolerant data structure* is an array of fault-tolerant blocks; see Fig. 1. All of our fault-tolerant data structures are blocked. One benefit is that error detection and correction can be performed locally as described above. The problem of removing all errors from a blocked fault-tolerant data structure is embarrassingly parallel. (However, each parallel processor needs its own $O(\delta)$ working store of safe memory.)

Theorem 2.2 (Live recovery). *All errors can be removed from a blocked fault-tolerant data structure of size n by $p \leq \frac{n}{\delta}$ processors in $O(n/p)$ time.*

2.2 Fault-Resistant Operations

We can read a fault-tolerant block of multiplicity k in time $O(k)$ rather than $O(\delta)$ by reading only one of its $\lceil \frac{2\delta+1}{k} \rceil$ error-correcting codes. This speed-up comes at the expense of accuracy: by corrupting $k < \delta$ memory locations, an adversary may change the value returned by this operation. Our general approach is to compose fast but inaccurate operations with slow but reliable end-to-end verification. Whenever an operation fails, we implicitly discover the location of a memory fault, which we may avoid in the future, enabling us to amortize away the cost. In this section, we formalize this approach.

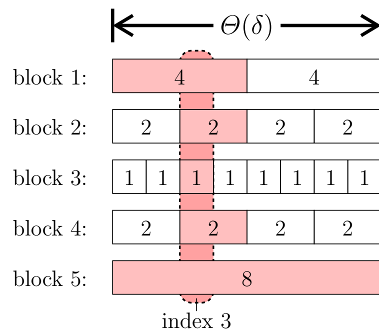

Fig. 1. Blocked data structures consist of a sequence of blocks each divided into $\Theta(\delta/k)$ codes of multiplicity k, for varying (power-of-two) values of k. If the prejudice number is 3, then an operation will read the shaded codes. The index 3 is faulty if any of those codes are faulty.

We adopt the notion of *prejudice numbers* from [10]. In their setting, each variable is copied $2\delta+1$ times, and the prejudice number p represents the minimal "trusted index." Formally, whenever an operation needs to access a variable, it uses the pth copy. If an operation fails, then a fault must have occured in the pth copy of some variable, so the prejudice number is incremented and the operation repeated.

In our setting, the prejudice numbers play a slightly different role; refer to Fig. 1. Each prejudice number $p \in \{1, 2, \ldots, 2\delta + 1\}$ is associated with one error-correcting code from each fault-tolerant block, as follows: in a fault-tolerant block of multiplicity k, the prejudice number p is associated with the $\lceil \frac{p}{k} \rceil$th error-correcting code. Thus k prejudice numbers are assigned to any given error-correcting code of multiplicity k.

An error-correcting code is *faulty* if it no longer decodes to the correct value. A prejudice number is *faulty* if it is associated with some faulty code. Because multiple prejudice numbers may be assigned to the same error-correcting code, a single faulty code may cause multiple indices to become faulty. By simple counting, we can bound the number of faulty indices:

Lemma 2.3. *The number of faulty indices is at most the number of memory faults.*

An operation is fault-*resistant* if it succeeds when given a nonfaulty prejudice number. We can fault-resistantly read a resilient variable in constant time by reading the copy indexed by the prejudice number. Using a similar idea, we can quickly read a fault-tolerant block:

Lemma 2.4. *We can fault-resistantly read a fault-tolerant block of multiplicity k in $O(k)$ time.*

2.3 Fault-Tolerant Operations

An operation is fault-*tolerant* if it always succeeds. In particular, fault-tolerant operations do not depend on the choice of a prejudice number. In this section, we describe our basic approach to implementing fault-tolerant operations, which is to compose a fault-resistant query with fault-tolerant verification.

Lemma 2.5. *If a query Q has a fault-resistant implementation that runs in t_1 time and does not modify the data structure, and it is possible to fault-tolerantly verify the result of Q in t_2 time, then Q has a fault-tolerant implementation with amortized running time $O(t_1 + t_2)$, assuming at least δ invocations.*

Proof (sketch). We start with prejudice number $p = 1$. Whenever we perform an operation, we verify the result. If the operation failed, then we increment the prejudice number p and try again. Because the original operation is fault-resistant, we only need to repeat the operation if the prejudice number is faulty. By Lemma 2.3, this happens at most δ times. Some technical details arise when this transformation is applied in general and when Q is probabilistic, which we address in the full version of the paper. □

2.4 Fault-Tolerant Memory

A *fault-tolerant memory* is a large array implemented by a sequence of fault-tolerant blocks of multiplicity δ. Fault-tolerant memory allows us to establish a relationship with data structures in the external-memory model.

Lemma 2.6. *Suppose that an operation can be completed in the external-memory model using T memory transfers and t computation steps, with block size $B = \delta$ and local memory $M = O(\delta)$. Then the same operation can be completed fault-tolerantly in the faulty RAM model in $O(\delta T + t)$ time.*

This correspondence does not generally produce efficient data structures because it does not make use of fault-tolerant blocks with lower multiplicities, but it is helpful in some cases.

Theorem 2.7. *There is a fault-tolerant linked list that stores n elements using $O(n+\delta)$ space. It supports advancing a pointer k steps, inserting k elements, and deleting k elements in $O(k+\delta)$ time.*

Theorem 2.8. *There is a fault-tolerant dictionary that stores n elements using $O(n+\delta)$ space and supports insertions, deletions, and lookups in $O(\delta)$ time per operation.*

3 Fault-Tolerant Predecessor

A *predecessor data structure* stores a set of keys x_i from some fixed ordered universe and supports insertions, deletions, and the predecessor query: given a key x, find the largest x_i that is at most x. Important examples of predecessor structures include balanced binary search trees, van Emde Boas priority queues, y-fast trees, and fusion trees.

We present a general technique for making any predecessor data structure fault-tolerant. Asymptotically, the transformation introduces only $O(\delta)$ additive space and time overhead. We require two technical conditions: (1) that predecessor queries do not modify the data structure, and (2) that the space use grows at least linearly with the number of keys. Splay trees do not satisfy the first condition, although AVL trees, red-black trees, and scapegoat trees do. Van Emde Boas queues whose size depends on the universe do not satisfy the second condition, although van Emde Boas queues with hashing do.

Our construction is similar in spirit to the resilient dictionaries of [10]. The main difference is that we use error-correcting codes within the leaves of our structure rather than accepting a small number of errors. Our results also hold for general predecessor structures rather than only for binary search trees. Our amortized startup cost is higher, however: our amortization holds only after $O(\delta)$ operations, rather than δ^{ε} operations.

Theorem 3.1. *Suppose that P is a predecessor data structure that stores n keys in space $s(n)$, supports queries in worst-case time $t_q(n)$ without modifying the data structure, and supports updates in amortized time $t_u(n)$. Then there is a fault-tolerant data structure P' that stores n keys in space $O\left(\delta s\left(\frac{n}{\delta}\right)\right)$, supports queries in $O(t_q(n)+\delta)$ amortized time, and supports updates in $O(t_q(n)+t_u(n)+\delta)$ amortized time. The data structure also incurs $\delta^{O(1)}$ worst-case preprocessing cost, plus an amortized startup cost of $O(\delta)$ operations. If the time bounds $t_q(n)$ or $t_u(n)$ are valid only in expectation, then the amortized time bounds for P' are also valid only in expectation.*

Proof (sketch). We divide the keys into $\frac{n}{\delta}$ contiguous groups of size $\Theta(\delta)$. We choose one representative from each group, and put these $\frac{n}{\delta}$ representatives into

an instantiation of P which we call the *summary structure*. This copy of P is stored in resilient variables (i.e., copied $2\delta + 1$ times), while the keys themselves are stored in fault-tolerant memory.

In order to find the predecessor of x, we perform a fault-*resistant* query in the summary structure to find the group representative which precedes x. We then fault-*tolerantly* access the corresponding group, and search for the predecessor of x there. By Lemma 2.5, we can combine these operations to perform a fault-tolerant predecessor query.

In order to insert or delete a key, we first perform a predecessor query to find the group that should contain that key and then modify that group. Every $\Omega(\delta)$ operations, a group may grow too large or too small. When this happens, we either split the group in two or merge it with an adjacent group, updating the summary structure appropriately. □

4 Suffix Trees

A *suffix tree* stores a string S and supports the substring query: given a pattern P, report the positions of all occurrences of P in S. Each character of S may be an arbitrary machine word. Suffix trees occupy linear space, and can be constructed in linear time plus the time required to sort the stored string [7]. In this section, we describe a fault-tolerant suffix tree with the following guarantees:

Theorem 4.1. *There is a fault-tolerant suffix tree that stores a string S of length n in $O(n + \delta)$ space. The tree can be constructed in $O(n \log n + \delta)$ time, and supports substring queries in $O(m + k + \delta)$ amortized time, where m is the length of the pattern and k is the number of its occurrences. The data structure also incurs $\delta^{O(1)}$ worst-case preprocessing cost, plus an amortized startup cost of $O(\delta)$ operations.*

Our construction is based on a new decomposition of a tree into short paths and small subtrees, which we describe first. We then show how to to apply this decomposition to build a suffix tree that supports fault-resistant substring queries. Finally, we describe how to store some extra data which makes it possible to efficiently verify the results of substring queries.

4.1 Fault-Resistant Tries

A trie is a rooted tree in which each edge is assigned a constant-size label. Each vertex may also have a constant-size label. Tries support the retrieval operation: given a pattern P of length m, return the vertex obtained by starting at the root and walking down the tree, following the edges labeled by the corresponding elements of P. Let T be a trie of n nodes. We show how to build an implementation of T that supports fault-resistant retrieval operations.

First we partition T into micro trees and micro paths as described by the following lemma; refer to Fig. 2.

Lemma 4.2. *Any rooted tree on n nodes can be partitioned into a collection of rooted subtrees of at most δ nodes, called* micro trees*, and $O(\frac{n}{\delta})$ paths of length at most δ, called* micro paths*.*

Proof (sketch). We start with the micro-tree/macro-tree decomposition of [1]. Then we split non-branching paths in the macro tree into subpaths of length at most δ. □

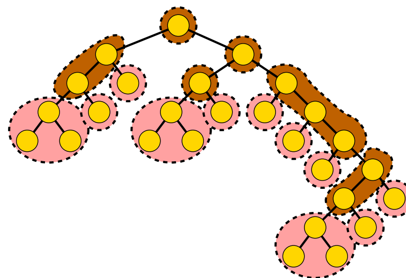

Fig. 2. Our fault-tolerant trie first decomposes into micro trees, rooted subtrees of size at most δ, and splits the remainder into nonbranching paths of length at most δ. Here $\delta = 3$.

The central idea of our construction is to store a micro tree or micro path of size k in a fault-tolerant block of multiplicity k. Every micro path traversed in a retrieval operation is read in its entirety, except perhaps the last one. By storing each path in its own fault-tolerant block, we are able to perform these accesses in total time that is linear in $|P|$. Only one micro tree gets touched during a traversal, which we can access in $O(\delta)$ time.

The use of fault-tolerant blocks of intermediate multiplicity is essential, because a micro path of size k must be stored in $O(\delta)$ space and must be traversable in $O(k)$ time. This is precisely the guarantee afforded by our fault-tolerant blocks.

In order to perform the tree decomposition safely in the presence of memory faults, we use the correspondence observed in Lemma 2.6 and standard techniques for manipulating trees in external memory.

Lemma 4.3. *Given an Euler tour of a trie of size n in fault-tolerant memory, we can fault-tolerantly construct a fault-resistant trie in $O(n + \delta)$ time.*

We can efficiently perform a fault-resistant retrieval query analogously to a retrieval query on a traditional trie. Essentially, each time the retrieval accesses a micro path or micro tree, we read the entire path or tree and load the result into safe memory.

Theorem 4.4. *There is a fault-resistant algorithm to retrieve a pattern P of length m (provided as a stream) in a fault-resistant trie T in $O(m + \delta)$ time.*

4.2 Fault-Tolerant Suffix Trees

Suppose that S is a string of length n stored in external memory in blocks of size B. There is an external-memory algorithm that computes the Euler tour of the compressed suffix tree for S which runs in $O(n \log n)$ time and $O(\frac{n \log n}{B})$ memory transfers [4]. Given access to a string S as a stream, we can construct a

fault-tolerant suffix tree for S as follows. By Lemma 2.6, there is a fault-tolerant algorithm that computes the Euler tour of the suffix tree for S in $O(n \log n + \delta)$ time and that stores the result in fault-tolerant memory. By Lemma 4.4, we can build a fault-resistant version of this trie. Our fault-tolerant suffix tree is composed of the string S and an Euler tour of the suffix tree, stored in fault-tolerant memory, and of the fault-resistant trie. This additional information allows us to perform fault-tolerant queries on the suffix tree.

In order to perform a fault-tolerant suffix tree operation, we first perform a fault-resistant query in the compressed suffix trie. With some care, we can verify the result of this query by fault-tolerantly examining the corresponding location of S. We then apply Lemma 2.5.

5 Interval Trees

An *interval* is a pair $[a_i, b_i]$ of elements from some fixed ordered universe with $a_i \leq b_i$. Interval (a_i, b_i) contains x if $a_i \leq x \leq b_i$. An *interval tree* maintains a set of intervals and supports insertions, deletions, and the stabbing query: given element x, report all intervals containing x. In this section, we describe a fault-tolerant interval tree with the following guarantees:

Theorem 5.1. *There is a fault-tolerant interval tree that stores n intervals in $O(n+\delta \log \delta)$ space, supports fault-tolerant updates in $O(\log n+\delta \log \delta)$ amortized time, and supports fault-tolerant stabbing queries in $O((\log n + k) \log \frac{1}{\varepsilon} + \delta \log \delta)$ amortized time, where k is the number of reported intervals and ε is an upper bound on the probability of failure. The data structure also incurs $\delta^{O(1)}$ worst-case preprocessing cost, plus an amortized startup cost of $O(\delta)$ operations. An adversary may adaptively corrupt the results of δ stabbing queries, and each other stabbing query fails with probability at most ε.*

We follow the construction of interval trees from [5]. The interval tree stores every interval's endpoints in the leaves of a binary tree. Each interval is stored twice at the least common ancestor of the leaves that store its endpoints: once in a list sorted in increasing order of left endpoint, and once in a list sorted in decreasing order of right endpoint. To perform a stabbing query at x, we search the binary search tree for x. Every interval containing x was stored in one of the nodes passed during this search. Whenever we move to the left child of a node, we report all intervals with left endpoint less than x; whenever we move to the right child of a node, we report all intervals with right endpoint at least x. If x is contained in k of the intervals stored at an internal node, we can find all k in $O(k)$ time by scanning the appropriate list until we find an interval not containing x.

The principal difficulty in making this structure fault-tolerant is that we do not know in advance how many intervals we will need to read from an internal node. In order to achieve a runtime $O(\log n + k) + \tilde{O}(\delta)$, we need to report each interval in amortized constant time. But in order to report m intervals from an internal node in $O(m)$ time using the techniques we have seen so far, we need to

store them in a fault-tolerant block of multiplicity $O(m)$. This strategy would incur significant space overhead, because m may be much smaller than δ. To overcome this difficulty, we store the intervals in a *prefix list*, composed of a sequence of fault-tolerant blocks of exponentially increasing size; see Fig. 3.

Theorem 5.2. *A* fault-tolerant prefix list *stores n elements from an ordered universe in $O(n+\delta\log\delta)$ space, supports fault-resistant access to the first k elements in $O(k)$ time for any k, and supports insertions and deletions in $O(\log n+\delta\log\delta)$ time.*

Proof (sketch). We store the first 2^i elements in a fault-tolerant block of multiplicity 2^i for $i=0,1,\ldots,\log\delta$. Because very little data is stored redundantly, the total space overhead is only $O(\delta\log\delta)$, but to read k elements, we can read a single fault-tolerant block of multiplicity at most $2k$, in $O(k)$ time. □

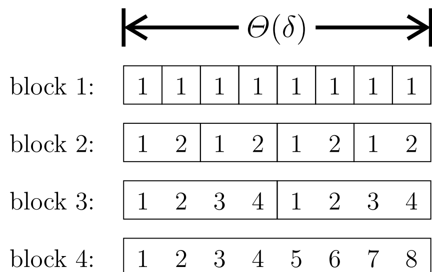

Fig. 3. The prefix list stores δ copies of the first element, $\delta/2$ copies of the first and second elements in a fault-tolerant block of multiplicity 2, $\delta/4$ copies of the first four elements in a fault-tolerant block of multiplicity 4, etc., and 1 copy of the first δ elements.

By a similar argument, a fault-tolerant prefix list supports finding all k elements less than a fixed value x in $O(k)$ time.

We can now implement a fault-resistant interval tree by carefully transforming the classical construction, using a prefix list to store the intervals at each interior node. In contrast with our previous data structures, we have no way to verify the results of an interval stabbing query. Instead, we guarantee that our results are correct with high probability by reading several error-correcting codes at random from each fault-tolerant block we access.

References

1. S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. In *Proc. 39th Annual Symposium on Foundations of Computer Science*, pp. 534–543, 1998.
2. Y. Aumann and M. A. Bender. Fault tolerant data structures. In *Proc. 37th Annual Symposium on Foundations of Computer Science*, pp. 580–589, 1996.
3. G. S. Brodal, R. Fagerberg, I. Finocchi, F. Grandoni, G. F. Italiano, A. G. Jørgensen, G. Moruz, and T. Mølhave. Optimal resilient dynamic dictionaries. In *Proc. 15th Annual European Symposium on Algorithms*, pp. 347–358, 2007.
4. Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 139–149, 1995.
5. Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proc. IEEE*, 80(9):1412–1434, September 1992.

6. R. de Wolf. Error-correcting data structure. In *Proc. 26th International Symposium on Theoretical Aspects of Computer Science*, pp. 313–324, February 2009.
7. M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47:987–1011, November 2000.
8. I. Finocchi, F. Grandoni, and G. F. Italiano. Resilient search trees. In *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 547–553, 2007.
9. I. Finocchi, F. Grandoni, and G. F. Italiano. Optimal resilient sorting and searching in the presence of memory faults. *Theoretical Computer Science*, 410(44):4457–4470, October 2009.
10. I. Finocchi, F. Grandoni, and G. F. Italiano. Resilient dictionaries. *ACM Transactions on Algorithms*, 6(1), 2009.
11. I. Finocchi and G. F. Italiano. Sorting and searching in faulty memories. *Algorithmica*, 52(3):309–332, 2008.
12. X. Li, K. Shen, M. C. Huang, and L. Chu. A memory soft error measurement on production systems. In *Proc. 2007 USENIX Annual Technical Conference*, pp. 21:1–21:6, 2007.
13. D. A. Spielman. Linear-time encodable and decodable error-correcting codes. *IEEE Transactions on Information Theory*, 42(6):1723–1732, 1996.
14. Tezzaron Semiconductor. Soft errors in electronic memory. White paper, January 2004. http://www.tezzaron.com/about/papers/soft_errors_1_1_secure.pdf.