History-Independent Dynamic Partitioning: Achieving Operation-Order Privacy in Ordered Data Structures

MICHAEL A. BENDER, Stony Brook University and RelationalAI, USA
MARTÍN FARACH-COLTON, New York University, USA
MICHAEL T. GOODRICH, University of California, Irvine, USA
HANNA KOMLÓS, New York University, USA

A data structure is *history independent* if its internal representation reveals nothing about the history of operations beyond what can be determined from the current contents of the data structure. History independence is typically viewed as a security or privacy guarantee, with the intent being to minimize risks incurred by a security breach or audit. Despite widespread advances in history independence, there is an important data-structural primitive that previous work has been unable to replace with an equivalent history-independent alternative—*dynamic partitioning*. In dynamic partitioning, we are given a dynamic set S of ordered elements and a size-parameter B, and the objective is to maintain a partition of S into ordered groups, each of size $\Theta(B)$. Dynamic partitioning is important throughout computer science, with applications to B-tree rebalancing, write-optimized dictionaries, log-structured merge trees, other external-memory indexes, geometric and spatial data structures, cache-oblivious data structures, and order-maintenance data structures. The lack of a history-independent dynamic-partitioning primitive has meant that designers of history-independent data structures have had to resort to complex alternatives. In this paper, we achieve history-independent dynamic partitioning. Our algorithm runs asymptotically optimally against an oblivious adversary, processing each insert/delete with $O(1)$ operations in expectation and $O(B \log N / \operatorname{loglog} N)$ with high probability in set size N.

CCS Concepts: • **Theory of computation → Design and analysis of algorithms**; **Online algorithms**; **Data structures design and analysis**; *Randomness, geometry and discrete structures*; Database theory; *Data structures and algorithms for data management*; *Theory of database privacy and security*; • **Security and privacy** → File system security; *Database and storage security*;

Additional Key Words and Phrases: algorithms, data structures, history independence, external memory, randomized algorithms, online algorithms

ACM Reference Format:
Michael A. Bender, Martín Farach-Colton, Michael T. Goodrich, and Hanna Komlós. 2024. History-Independent Dynamic Partitioning: Achieving Operation-Order Privacy in Ordered Data Structures. *Proc. ACM Manag. Data* 2, 2 (PODS), Article 108 (May 2024), 27 pages. https://doi.org/10.1145/3651609

Authors' addresses: Michael A. Bender, bender@cs.stonybrook.edu, Stony Brook University and RelationalAI, Stony Brook, NY, USA; Martín Farach-Colton, martin@farach-colton.com, New York University, New York, NY, USA; Michael T. Goodrich, goodrich@acm.org, University of California, Irvine, Irvine, CA, USA; Hanna Komlós, hkomlos@gmail.com, New York University, New York, NY, USA.

ACM 2836-6573/2024/5-ART108
https://doi.org/10.1145/3651609

1 INTRODUCTION

A data structure is said to be ***history independent (HI)*** if its internal representation reveals nothing about the history of the past operations beyond the current contents of the data structure [40, 44]. History independence is typically viewed as a security or privacy guarantee, with the intent being to minimize risks incurred by a security breach or audit. Motivation includes preventing voting machines from leaking information about how people voted from the order in which they voted (e.g., [34]), stopping files revealing damaging and embarrassing information that their creators thought had been erased [31, 35, 36, 44], and databases revealing that some of their data or metadata has been removed (e.g., due to a court order that must remain confidential) [30, 38, 42, 62].

For an example of an important data structure that is not history independent, consider the classic B-tree [12, 26], the well-known indexing structure used in databases, file systems, and key-value stores. B-trees support insertions, deletions, and searching in $O(\log_B N)$ I/Os, where N is the number of elements in the B-tree and B is the disk-block size. B-tree inserts/deletes are supported by ***dynamically partitioning*** the N key-value pairs into leaf blocks and dynamically partitioning the pivot keys at each level of the tree into internal nodes—and all of this partitioning is based on simple splitting and merging rules. B-trees are not history independent, because these splitting/merging rules means that an observer who examines the balance information in a B-tree, including which nodes are mostly full, may learn which elements might have been inserted recently and which were inserted a long time ago.

The challenge in making B-trees history independent is that the dynamic partitioning (i.e., node splitting/merging) is not history independent. Thus, as we describe below, researchers have proposed history-independent alternatives to the B-tree, which have worse performance bounds than B-trees, place restrictions on B, and are more complicated. These include Golovin's B-treap [32] and B-skip-list [31, 33], Bender et al's history-independent external-memory skip list [13] and history-independent cache-oblivious B-tree [13]. A history-independent dynamic partitioning scheme would obviate the need for such alternatives—e.g. by making a regular B-tree itself history independent—and so could many other data structures.

The history of history independence. There are different degrees of history independence [35, 36, 44]. A data structure is ***weakly history independent*** (WHI) if it leaks no additional information to an observer who observes the memory representation once, besides the contents at the time of observation. A data structure is ***strongly history independent*** (SHI) if it leaks no additional information to an observer who sees the memory representation multiple times. A history-independent data structure is ***auditable*** [34] if the data structure can quickly prepare for an observation. [1] In each case, we assume that an observer can see a memory representation before or after, but not during an update operation. History independence was introduced by Micciancio [40], who showed how to build a search tree with a history-independent structure. Naor and Teague [44] showed that even the bit representation of a data structure (e.g., the memory addresses where the tree nodes are stored) can leak information to an observer. They formalized the now classical notions of history independence, including the strong and weak variants, and showed how to build strongly history-independent dictionaries (trees and hash tables). Hartline et al. [35, 36] showed a data structure is SHI if and only if each state of the data structure has a unique (***canonical***) representation, after initial random bits (which could, e.g., be used to define random hash functions) are set, provided the state graph is strongly connected (i.e., all states of the data structures are

[1] For example, the on-disk data structure storing a database's data may have a transaction log. Since the transaction log records recent transactions, by definition it is not history-independent. However, the data structure could be audible, since the transition log could be flushed in preparation for an observation.

mutually reachable). For simplicity, since all data structures considered in this paper have strongly connected state graphs, we take this as a definition of SHI in this paper.

Definition 1.1 (Theorem 1 in [35]). A data structure of an abstract data type whose state graph is strongly connected is ***strongly history independent (SHI)*** if, after all initial random bits have been initialized, each logical state of the abstract data type has a canonical representation in memory.

There is a large literature on designing data structures to be history independent without an asymptotic slow down as compared to their non-history independent counterparts [1, 13, 21, 23, 32, 33, 35, 36, 40, 43, 44], including fast history-independent constructions for cuckoo hash tables [43], linear-probing hash tables [21, 34], other hash tables [21, 44], trees [1, 40] memory allocators [34, 39, 44], write-once memories [41], priority queues [23], union-find data structures [44], external-memory dictionaries (e.g., for a database or key-value store) [13, 31–33], file systems [8, 10, 11, 52], cache-oblivious dictionaries [13], order-maintenance data structures [21], packed-memory arrays/list-labeling data structures [13, 15], and geometric data structures [60]. Given the strong connection between history independence and unique representability [35, 36], some data structures that predate the formalism can be made history independent, including ordered linear probing/Robinhood hashing [2, 24], skip lists [49], treaps [7], and other less well known deterministic data structures with canonical representations [4, 5, 50, 56, 57]. This foundational algorithmic work on history independence has found its way into databases [10, 47, 53] and other systems; there are now voting machines [20], file systems [8, 9, 11], and other storage systems [25] that use history independence as an essential feature.

1.1 Dynamic Partitioning

Despite vigorous advances in history independence, there is one foundational data-structural primitive that researchers have previously been unable to replace with an equivalent history-independent alternative—***dynamic partitioning***. Dynamic partitioning is important throughout computer science, but, as we saw earlier in the example of B-tree rebalancing [12, 26], it is especially important in databases.

Dynamic partitioning is also used in write-optimized dictionaries such as B^ε-trees [19, 22] and log-structured merge trees [45, 54], fusion trees [3, 6, 28, 46, 51, 58, 63], and many other data structures.

In ***dynamic partitioning***, we are given a set S of N ordered elements and a size-parameter B, and the objective it to maintain a partition of S into ***ordered groups*** each of size $\Theta(B)$. By "ordered groups", we mean that for any two groups, all of the elements in one group are smaller than all of the elements in the other. The lack of any history-independent dynamic-partitioning primitive has historically meant that designers of history-independent data structures have had to resort to complex alternatives to avoid this primitive. Often these alternatives come with a significant performance hit compared to non-HI alternatives.

Returning to the B-tree example, in a (non-history-independent) B-tree, dynamic partitioning takes place at each level of the tree. A B-tree is a tree with size-B nodes and fanout $\Theta(B)$, and where all of the leaves are at the same depth. All of the elements stored in a B-tree are maintained in sorted order in the leaf nodes. Thus, as leaves merge and split, the leaves implement a dynamic $\Theta(B)$-partitioning of the elements. Moreover, there is dynamic partitioning at *each* level of the B-tree. The nodes at height 1 store pivot elements that define the partition at the leaves, and these pivots are themselves partitioned into nodes, as defined by the pivots stored at height 2; and so on up the tree.

B-tree rebalancing is fully defined by the ***dynamic partitioning algorithms*** at each level of the tree. Traditional (non-history-independent) B-tree rebalancing/partitioning works as follows.

When a node v gets too full (e.g. it has N_v elements, with $N_v \geq B$), it triggers a split into two nodes, each with at most $\lceil N_v/2 \rceil$ elements. When a node v gets too empty ($N_v < B/3$), it triggers a merge with a neighbor that is also a sibling (which could kick off an immediate split). The result is that each group *deterministically* has at most B elements and at least $B/3$ elements, which implies that its height is $O(\log_B N)$ and its number of nodes is $O(N/B)$.

This lazy splitting and merging policy guarantees good amortized update costs, but exhibits ***hysteresis*** [37]. Specifically, the amortized number of elements per insert/delete that move from one node to another is $O(1)$. This is because after a node splits, there are $\Omega(B)$ inserts/deletes before the node splits/merges again. (In the language of dynamic partitioning, the ***element-movement cost*** is the amortized number of elements per insert/delete that move from one group to another, and the best we can hope for is $O(1)$.) Similarly, after a node triggers a merge, there are $\Omega(B)$ inserts/deletes before another merge is triggered. (In the language of dynamic partitioning, the ***group-update cost*** is the the total number of group boundaries that shift as a result of the insert/delete, and optimal is $O(1/B)$.) The hysteresis of this splitting/merging rule prevents an adversary from inserting an element to trigger a split, deleting that element to trigger a merge, reinserting the element to trigger a split, and so on. But hysteresis in the rebalancing policy—by definition—means that history is taken into account in the rebalancing; hence, the need to prevent an adversary from pumping the splitting/merging algorithm presents a fundamental roadblock to achieving a history independent B-tree.

As noted above, many other database-indexing structures also use dynamic partitioning in some way. In write-optimized dictionaries such as B^ε-trees [18] or log-structured merge trees [45], the dynamic partitioning at each tree level takes place on a subset of S. In some structures, such as cache-oblivious B-trees [16] there is dynamic partition and smaller dynamic partitioning recursively within. Dynamic partitioning is often important in disk-resident data structures, because in order to have good I/O-complexity, we like to access/modify the data structure in chunks of size B, where B is the I/O size.

Dynamic partitioning is used as a simple subroutine in data structures in RAM, rather than on disk. E.g.., in fusion trees [3, 6, 28, 46, 51, 58, 63], partitioning takes place at the level of machine words. (Fusion trees [3, 6, 28, 46, 51, 58, 63] are word-RAM dictionaries that support $o(\log N)$-time searches and updates.) In order-maintenance data structures [14, 27], dynamic partitioning is used to support a kind of "indirection". While the details don't matter for the purpose of this paper, this dynamic $\Theta(\log N)$-partitioning enables the insertion performance to improve from $O(\log N)$, as in earlier data structures [59], to $O(1)$ [27].

1.2 Case Study of (Lack of) History-Independent Dynamic Partitioning

This subsection illustrates how algorithm designers have needed to twist themselves into pretzels to find history-independent replacements for traditional data structures, when they have not had access to history-independent $\Theta(B)$-partitioning. The result has been more complicated and less performant data structures.

Once again, we first revisit the B-tree. One history-independent alternative to the B-tree is Golovin's B-skip-list [31, 33]. To adapt a traditional skip list[2] for external memory, it is natural to modify the promotion probability from $1/2$ to $1/B$. Thus, with high probability in N, there are $O(\log_B N)$ lists instead of $O(\log_2 N)$ lists, as with a traditional skip list. Moreover, the expected

[2]A (traditional) skip list [48] is just a sequence of linked lists. The ***level 0 list*** maintains all of the elements in the skip list. An element is ***promoted*** from level i to level $i+1$ with probability 1/2, which means that that element is stored not only in the level-i list but also in the level-$i+1$ list. There are pointers between every instantiation of an element. The cost for searching/inserting/deleting is $O(\log N)$ with high probability in N, where N is the number of elements in the skip list [55].

number of non-promoted elements in a list between promoted elements is B. Unfortunately, the I/O-performance of an external-memory skip list is simply not as good as that of a traditional B-tree. For starters, the search cost is $O(\log_B N)$ I/Os in expectation, rather than $O(\log_B N)$ I/Os deterministically, as with a traditional B-tree. More problematically, we do not even have good high-probability bounds. In fact, with high probability, there exist elements whose search cost is $O(\log_2 N)$ I/Os [13], which is no better than if those elements were stored in a (non-external memory) binary tree!

The reason for this decreased performance is that this randomized promotion mechanism does an uneven job of dynamically partitioning the leaves. With high probability, there will be as many as $\Theta(B \log N)$ elements between two promotions. On the other hand, we will see as few as $\Theta(1)$ elements between some promotions. That is, the partitioning of a B skip list is randomized with a lot of variance: a group has size $\Theta(B)$ an expectation, but with high probability, some groups have size $\Theta(1)$ whereas others have size $\Theta(B \log N)$.

Bender et al [13] fix this drawback for some parameter choices by complicating the data structure. They first impose $B = \Omega(\text{polylog } N)$, and then they increase the promotion probability from to $1/B$ to $1/\sqrt{B}$. It turns out that with these restrictions, the search cost is $O(\log_B N)$ with high probability. But now, almost all disk blocks are empty—only a $1/\sqrt{B}$-fraction full in expectation. Thus, the range-query performance and space usage are horrible. To fix range queries and space, they then amalgamate nodes into supernodes within the bottom few levels of their external-memory skip list and add extra internal pointers within the supernodes. Thus, for some values of B, the resulting data structure has high-probability (but still not deterministic) search bounds of $O(\log_B N)$. Even given all of this effort: the data structure only works for some parameter choices; searches still only achieve concentration bounds rather than deterministic bounds; and internal nodes are mostly empty, meaning they are still wasting space, leading to worse search performance.

Another approach investigated by Golovin [31, 32] is a B-tree alternative that is built by modifying a treap [7] for external memory. Once again, we have a data structure that is not a B-tree (e.g., leaves are different depths), where search cost guarantees are not deterministic, and where space, range queries, and/or high-probability bounds are not as good as a B-tree. A final approach is to replace a B-tree with a history-independent B-tree [13] based on a history-independent packed-memory array. Now the search cost is deterministic, and all leaves are at the same depth, but for some workloads (e.g., sequential inserts and some relationships between B and N), the performance is not as good as a B-tree.

The bottom line is that without a history-independent dynamic partitioning scheme that deterministically guarantees $\Theta(B)$-sized groups, the above previous approaches have drawbacks, in terms of performance and extra complications. Instead, with a history-independent, dynamic partitioning scheme, we immediately achieve a traditional B-tree whose balance structure is history independent, and where nodes deterministically have $\Theta(B)$ elements. Making it fully history independent is then trivial—just plug-in a history independent memory allocator.

The situation is similar with some other data structures that are based on dynamic partitioning. For example, Blelloch and Golovin propose a history-independent order-maintenance data structure [21]. But with a history-independent partitioning scheme coupled with a history-independent weight-balanced dictionary [21], the data structure follows immediately. There are other data structures (e.g., fusion trees) that have never been made history independent, perhaps because they are too complicated to modify extensively. But given HI dynamic partitioning, they can be made history independent without other modifications. See Appendix A.2.

1.3 Our Results

In this paper, we establish that history-independent dynamic partitioning is achievable. In fact, our schemes are ***strongly*** history-independent, meaning that after some random bits have been initialized—these define a hash function—the partition is ***uniquely representable***. That is, for *any choice* of hash function and value for B, there is a unique way that a set S ($|S| = \Omega(B)$) gets partitioned into groups whose sizes are all *deterministically* $\Theta(B)$.

We give an algorithm that achieves $(B, 2)$-partitioning meaning that (deterministically) every group in the partition has size between $B/2$ and B. The algorithm runs asymptotically optimally, in expectation, against an oblivious adversary.

Theorem 1.2 (group-update cost). *History-independent $(B, 2)$-partitioning of a set S of size $N \geq B$ can be maintained with group-update cost $O(1/B)$ per insertion/deletion in expectation, and $O(\log N/\operatorname{loglog} N)$ with high probability in N.*

Theorem 1.3 (element-movement cost). *History-independent $(B, 2)$-partitioning of a set S of size $N \geq B$ can be maintained with element-movement cost $O(1)$ amortized per insertion/deletion in expectation, and $O(B \log N/\operatorname{loglog} N)$ with high probability in N.*

We extend these bounds to stricter regimes where the smallest partition has size at least B/α, for arbitrary $\alpha > 1$. See Appendix A.1.

We re-emphasize that the partitioning is deterministic. That is, there is no probability that the partitions could get out of bounds.

Our partitioning algorithm is based on the ***protected Cartesian tree***, an auxiliary data structure that we introduce. A Cartesian tree is a well-known transformation of a numerical array into a tree and is used, for example, in fast algorithms for computing least common ancestors [17].

Our actual algorithms do not compute protected Cartesian trees but heavily rely on the relationship that we elucidate between protected Cartesian trees and partitions in our proofs. Because we believe that protected Cartesian trees may have other uses, we also establish some of their algorithmic properties.

Given Theorems 1.2 and 1.3, building a history independent B-tree is straightforward, since we can just treat each level of a the B-tree as a partitioning problem on the pivots of the next level down. As we will see, the analysis of the cost of maintaining HI B-trees is the challenge. This is covered in Section 4. If we directly apply the bounds of Theorems 1.2 and 1.3, we would obtain weak bounds. We are able to show that the HI B-tree can be maintained with $O(\log_B(N)/B)$ I/Os in expectation, and $O\left(\frac{\log N}{\operatorname{loglog} B}\right)$ with high probability; see Theorem 4.5.

1.4 Roadmap

In Section 2, we formally define the (B, α)-partitioning problem and give a static algorithm for $(B, 2)$-partitioning. We extend this to a dynamic algorithm in Section 3. In Section 4, we use our dynamic partitioning algorithm to build a history-independent B-tree. In Appendix A.1, we extend our algorithm to (B, α)-partitioning for smaller values of α. In Appendix A.2, we describe how to use our B-tree construction to build a history-independent fusion tree. In Appendix A.3, we provide proofs that were deferred from the main body of the paper for space reasons.

2 PROTECTING THE FLANKS

As a warm-up for our HI dynamic partitioning solution, let us first describe an algorithm for constructing a canonical static partition that is based on a simple ***protect-the-flanks*** technique. The main idea of the protect-the-flanks technique is to choose pivots for recursively splitting an ordered set so that the smallest and largest $B/2$ elements are not eligible. As we show, this

simple approach is sufficient to define a canonical static partition; the challenge, which we address subsequently, is to maintain such a partition dynamically.

We formalize the partitioning problem as follows:

*Definition 2.1 (**(B, α)-partitioning**).* Let S be an ordered set of size N. Let $\mathcal{G}$ be a partition of S. We call every $G \in \mathcal{G}$ a ***group***; let $G_{\min}$ (resp. $G_{\max}$) be the minimum (resp. maximum) element in G; the ***range*** of group G is $[G_{\min}, G_{\max}]$.

For $B > \alpha > 1$, $\mathcal{G}$ is a ***(B, α)-partition of S***, if it satisfies the following invariants:

(1) **Ordered groups.** The ranges of all groups in $\mathcal{G}$ are disjoint.
(2) **$\Theta(B)$ cardinality.** If $N \geq B/\alpha$ the size of each group $G \in \mathcal{G}$ satisfies $B/\alpha \leq |G| \leq B$, and thus there are $\Theta(N/B)$ groups. Otherwise, there is a single group of cardinality N.

For convenience, we will refer to $(B, \Theta(1))$-partitioning as ***$\Theta(B)$-partitioning***. When S changes dynamically, we have ***dynamic (B, α)-partitioning*** (and $\Theta(B)$-partitioning):

*Definition 2.2 (**dynamic (B, α)-partitioning**).* Maintain a (B, α)-partition of S when elements can be inserted into or deleted from S. A problem instance is defined by a sequence $\sigma = \sigma_1, \sigma_2, \ldots$, where each σ_i is the insertion of an element into S or a deletion of an element from S. The algorithm proceeds in ***rounds***:

- On deletion, the element is deleted from its group; on insertion, the element is added to a new or existing group, subject to the ordered-groups invariant.
- Then a (possibly empty) set of ***shifts*** are performed, where each shift moves (the largest or smallest) k elements (for $1 \leq k \leq B$) from some group G to some (new or existing) group G', subject to the ordered groups invariant.

At the end of all shifts, the $\Theta(B)$-cardinality invariant is satisfied. Each shift has a cost, and the objective is to minimize the sum of the costs of all shifts over the sequence σ.

This definition is general enough that we need to tease apart different aspects of the cost of a shift. The ***element-movement cost*** of a shift of k elements is k. The ***element-movement cost*** of a round is the sum of the element-movement costs of its shifts, plus one for the element that is being added or removed. The ***group-update cost*** of a round is the total number of shifts during that round. So note that the original insertion of an element does not contribute to the group-update cost.

In order to distinguish between different costs, we also define the ***CPU cost*** of an operation to be the number of CPU operations necessary to perform the operation, and we define the ***I/O cost*** of an operation to be the number of I/Os necessary to perform the operation.

2.1 Static $(B, 2)$-Partitioning

In this section we describe a static algorithm for constructing a canonical $(B, 2)$-partition. Before we describe our algorithm, however, let us first give a few more definitions.

Consider a set $S \subseteq \mathcal{U}$, where universe $\mathcal{U}$ admits a total order $\prec$. For notational convenience, let $-\infty$ be the minimal element in $\mathcal{U}$ and ∞ be the maximal element in $\mathcal{U}$. Define ***S restricted by (u, v)***, denoted ***$S[u, v]$***, as follows: $S[u, v] = \{x \in S \mid u \prec x \prec v\}$. Thus, $S = S[-\infty, \infty]$. The ***rank*** of any element $u \in S$ is defined as $\mathbf{rank}_S(u) = |\{x \in S \mid x \preccurlyeq u\}|$. Denote $|S|$ by N. We emphasize that the restriction $S[u, v]$ does not include the endpoints u, v, and so $|S[u, v]| = \mathrm{rank}_S(v) - \mathrm{rank}_S(u) - 1$.

Protected regions of a set. The ***protected region*** of S is the set $P(S) = \{x \in S \mid \mathrm{rank}_S(x) \leq B/2\} \cup \{x \in S \mid \mathrm{rank}_S(x) > N - B/2\}$. Notice that if $|S| < B$, then $P(S) = S$. The ***unprotected region*** of S is $U(S) = S \setminus P(S)$, and we say that element $u \in S$ is ***unprotected*** if $u \in U(S)$. We define the ***region covered by u*** as $\boldsymbol{P_S(u)} = \{x \in S \mid \mathrm{rank}_S(u) - B/2 < \mathrm{rank}(x) \leq \mathrm{rank}_S(u) + B/2\}$,

and say that x is ***covered by u*** if $x \in P_S(u)$. Intuitively, the protected regions of an ordered set are its "flanks."

We also use a random hash function, $h : \mathcal{U} \to [0, 1]$, which maps elements to real numbers between 0 and 1, such that h's random bits are set prior to our construction.[3]

For simplicity of presentation, we assume that there are no hash collisions, i.e., that $h(x)$ is unique for every $x \in \mathcal{U}$. This is relevant, as we often look for the element in a set with minimal hash value.[4]

2.2 A Static Partitioning Algorithm

We define a partition on S by selecting a set of ***pivots*** $\{p_1, \ldots, p_k\}$, where $p_1 \prec p_2 \prec \cdots \prec p_k$. For notational convenience, we will consider $p_0 = -\infty$ and $p_{k+1} = \infty$. The pivots partition S into $k + 1$ groups $G_0, G_1, \ldots, G_k$, where $G_i = \{y \in S \mid p_i \preccurlyeq y \prec p_{i+1}\}$.

For any element $x \in \mathcal{U}$ (which may or may not be in S), we also define the ***predecessor of x***, denoted $\text{pred}(x)$, as the largest $y \in S$ such that $y \prec x$, and the ***successor of x***, denoted $\text{succ}(x)$, as the smallest $y \in S$ such that $x \prec y$.

Since the pivots uniquely define the partition of S, we henceforth focus our algorithms and analysis on the equivalent problem of pivot selection. Our static partitioning algorithm, which is based on the simple protect-the-flanks technique, is as follows.

Algorithm 1: Static $(B, 2)$-Partitioning. We select the pivots recursively. The first pivot we select is the unprotected element with the smallest hash, that is, the element $p \in U(S)$ such that $h(p)$ is minimized. We then recursively partition the two sets $S[-\infty, p]$ and $S[p, \infty]$, selecting pivots from each. The process terminates when no more selections can be made, i.e., when every recursive subproblem has size less than B and therefore all elements are protected. See Figure 1.

2.3 Protected Cartesian Trees

We next introduce the "Protected Cartesian Tree" on a set S, which generalizes the classical notion of a Cartesian tree [61].

Definition 2.3. Let each element x in ordered set S have an associated value, $\text{value}(x)$. A ***Cartesian Tree on S*** is a binary tree on S, such that:

(1) If u is a parent of v, then $\text{value}(u) < \text{value}(v)$.
(2) An in-order traversal of the tree returns the elements of S in sorted order.

For example, $\text{value}(x)$ could be the hash value, $h(x)$. We next define a generalization of the Cartesian tree, which we call the ***B-protected Cartesian tree on S*** (abbreviated simply as the ***protected Cartesian tree on S***).

We define the ***B-protected Cartesian tree on S***, denoted $\boldsymbol{T_S}$ (or $\boldsymbol{T}$ when S is clear from context), to be the internal nodes of a binary search tree, $\overline{T}_S$, which is defined recursively as follows. If $|S| < B$ (i.e., all elements in S are protected), then $\overline{T}_S$ is a single (leaf) node associated with S; hence, in this case, T_S is empty. Otherwise, the root of $\overline{T}_S$ is the element $u \in U(S)$ such that $\text{value}(u)$ is minimized. The left child of u is the root of the recursively constructed tree on $S[-\infty, u]$, denoted $\overline{T}_{S[-\infty,u]}$, and the right child of u is the root of $\overline{T}_{S[u,\infty]}$. Thus, if either recursive Cartesian tree is a leaf in $\overline{T}_S$, there is no corresponding child node in T_S. We refer to $\overline{T}_S$ as the ***closure*** of the protected Cartesian tree, T_S. See Figure 1.

[3] In practice, a pseudo-random hash function that maps elements to values having $3 \log n$ bits should be sufficient.
[4] However, we can also handle hash collisions by tiebreaking deterministically. For example, if $h(x) = h(y)$ for $x \neq y$, say that the element with lower rank wins the tiebreak.

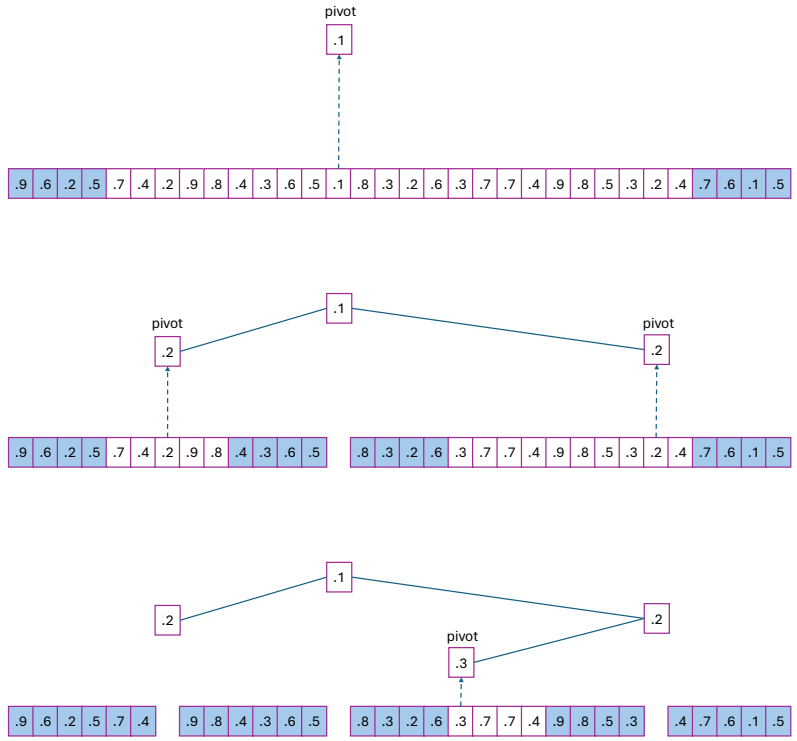

Fig. 1. Recursive calls in the partition algorithm (1). (Top) Level 1; (Middle) level 2; (Bottom) level 3. Each box shows the first digit of value(x) after the decimal point. Here, $B/2 = 4$ and protected regions are shaded light blue. The protected Cartesian tree is shown with solid edges.

We refer to these recursive sets $S[x, y]$ as ***subproblems***, and say that a subproblem is ***nontrivial*** if it contains at least B elements. In general, the protected Cartesian tree on $S[x, y]$ has root $w \in U(S[x, y])$ that minimizes value(w), and left subtree $\overline{T}_{S[x,w]}$ and right subtree $\overline{T}_{S[w,y]}$.

Observe that the protected Cartesian tree exactly specifies the recursive structure of Algorithm 1. Specifically, the nodes are the pivots, and the left (resp. right) child of any node in the tree is the root of its next left (resp. right) recursive subproblem. When Algorithm 1 is used to construct the protected Cartesian tree on set S, we refer to it as ComputeCartesian(S).

Note that because T_S consists of the internal nodes of $\overline{T}_S$, the protected Cartesian tree on S does not contain all elements of S. In fact, at most a $2/B$ fraction of the elements of S appear in T_S. Nevertheless, when we search in a protected Cartesian tree, we perform the search as a search in a standard binary search tree, searching in the closure, $\overline{T}_S$, of T_S, so that the result of the search ends in a leaf of $\overline{T}_S$.

The following lemma is intuitive, and its proof is deferred to Appendix A.3.1 for space.

Lemma 2.4. *For any set S with fixed hash function h, there is a unique protected Cartesian tree T_S with values* value$(x) = h(x)$*, and T_S is either empty or contains $\Theta(N/B)$ nodes.*

We reiterate that the set of nodes selected for the protected Cartesian tree with a random hash function h is exactly the set of pivots selected by Algorithm 1. Thus, our lemmas about building protected Cartesian trees translate directly into results about partitioning.

Corollary 2.5. *Algorithm 1 constructs a canonical protected Cartesian tree and $(B, 2)$-partition.*

Next, we give some lemmas about protected Cartesian trees which are of independent interest. The proof of Lemma 2.6 is deferred to Appendix A.3.1.

Lemma 2.6. *The protected Cartesian tree T_S on S with random hash function h has height $O(\log(N/B))$ with high probability in N.*

Corollary 2.7. *Each search operation in a protected Cartesian tree with a random hash function has CPU cost $O(\log(N/B))$ with high probability in N.*

Lemma 2.8. *It takes $O(N)$ CPU operations to build a protected Cartesian tree with arbitrary values on a set S, where $|S| = N$.*

Proof. It is known that a Cartesian tree can be built in $O(N)$ operations [29]. Given a Cartesian tree T, we can build a protected Cartesian tree by trimming out protected nodes, since the relative priorities between nodes are identical in protected and classical Cartesian trees.

We begin at the root a of the tree, and describe how to trim its left subtree; the right subtree will be symmetric. We first trim $b = \text{pred}(a)$ by removing it from T. We know that b has no right subtree (since a is its direct successor and has higher rank) and we replace b's left child as the new right child of b's parent. We next trim $c = \text{pred}(b)$ from T, noting that c once again has no right subtree, since b has been removed from T and a has higher rank. We repeat this process on the $B/2$ elements which directly precede a in S. The next element x preceding these should not be protected, and so it remains in the tree. We now trim the $B/2$ elements preceding x, and repeat until we have reached the element of lowest rank in S.

Every element S is touched exactly once, either to prune it or decide it remains in T, so this algorithm runs in $O(N)$ operations. ∎

In the next section, we consider the case when S is ***fully dynamic***, meaning that elements are inserted and deleted over time.

3 DYNAMIC HISTORY INDEPENDENT $(B, 2)$-PARTITIONING

Here we give expected and high-probability bounds on the element movement cost and group-update cost of history-independent $(B, 2)$-partitioning.

The overall goal of this section is to establish that the adversary cannot change the partition very much on an insertion. One component is to show that no matter where the adversary chooses to insert elements, the probability that the tree changes is $O(1/B)$. Our partition is based on a protected Cartesian tree, and the issue we run into is that if the adversary chooses to insert at a small number of specific ranks at the beginning or end of the set (e.g., at rank $B/2$), the probability that the protected Cartesian tree changes can be $\omega(1/B)$. Almost all other ranks do not cause problems, but we do not want the adversary to have any options.

Our solution has several steps:

- Define a ***modular*** version of protected Cartesian trees and modular dynamic $(B, 2)$-partitioning. This way there are no endpoints that the adversary can exploit.
- Turn the modular solution into a standard (***linear***) solution by cutting at the point of modularity. This may make partitions at the endpoints too small.
- If an endpoint partition is too small, merge it with its neighbor. This will yield a linear $(3B/2, 3)$-partition, or equivalently, a $(B, 3)$-partition by running this scheme with $B' = 2B/3$.
- Convert the $(B, 3)$-partition into a $(B, 2)$-partition using the (B, α)-partitioning algorithm in Appendix A.1.

See Section 3.4 for more details on this construction.

3.1 Making the Algorithm Modular

The modification to modularize the algorithm is simple: when selecting the root, we choose the element with minimal hash from all of S (i.e., no elements are initially protected). The rest of the algorithm proceeds identically to ComputeCartesian, except that now there is "wrap-around" in the set S. This means that both the protected regions on the two ends of S, and the resulting groups are subject to wrap-around. For example, for element x with rank $\text{rank}_S(x) = B/4$, its protected region is $P_S(x) = \{y \in S \mid \text{rank}_S(y) \leq 3B/4\} \cup \{y \in S \mid \text{rank}_S(y) > N - B/4\}$.

Since it is convenient to continue thinking about building protected Cartesian trees on linear sets, we equivalently ***rotate*** the set S after selecting the first pivot z. For any element $y \in S$, the rank of y in the rotated set is defined to be $\text{rank}^r_S(y) = (\text{rank}_S(y) - \text{rank}_S(z)) \mod |S|$. We now proceed on the rotated set, so left, right, ∞, and $-\infty$ are all well defined. Now the root of the tree will have one child, which is the root of the rotated set. If we think of the protected Cartesian tree on a linear set as having a dummy root which induces a trivial rotation, this is identical to the structure we have previously defined. We reiterate that the protected Cartesian tree is an analytical tool to select the pivots which are the boundary elements in a partition. Therefore, changes in the relative ordering of pivot elements due to a rotation which keep the set of pivots the same would have no impact on a partition of S.

The dynamic version of this algorithm, RecomputeCartesian, is defined in Section 3.2

Lemma 3.1. *Let $S \subseteq \mathcal{U}$ be a set of size $N \geq B$, and T_S, the protected Cartesian tree on S with random hash function h. Then every element in S has probability of $\Theta(1/B)$ of being in T_S.*

Proof. The proof follows by symmetry and because there are $\Theta(N/B)$ pivots in T_S by Lemma 2.4. Specifically, since hash function h is random, any element in S is selected as the root of T_S with equal probability, so every element also has equal probability of initially being protected. Continuing inductively on the left and right subproblems, every element has equal probability of being selected as a node in T_S. ∎

3.2 How the Protected Cartesian Tree Changes After Insertions

In this section, we will show how to maintain a protected Cartesian tree (and thus a modular $(B, 2)$-partition) on a dynamically changing set.

For the remainder of this section, we will fix $S \subseteq \mathcal{U}$ and $x \in \mathcal{U} \setminus S$, and let $S' = S \cup \{x\}$. We will bound the number of pivots that differ between $T_{S'}$ and T_S. We build $T_{S'}$ by selecting a (possibly new) root, and then recursively building the protected Cartesian tree, retaining as much of the previous structure as possible.

Next, we describe the algorithm RecomputeCartesian, which recomputes the protected Cartesian tree after the insertion of x into S.

Observation 1. *Since any set with a fixed hash function has a unique protected Cartesian tree, RecomputeCartesian is strongly history independent by Definition 1.1.*

RecomputeCartesian(S, x)

Case 0: $|S'| < B$. Then both protected Cartesian trees T_S and $T_{S'}$ are empty. Return the empty set.

Case 1: x has the minimum hash in S'. Then x is the new root of S'. Rotate S' and return a tree with x as the root and ComputeCartesian(S') as the child of x.

Case 2: x does not have the minimum hash in S'. Then the prior root z of S still has minimal hash and remains the root of S', and the elements in $P(z)$ become protected. Return a tree with z as the root and RecomputeSubproblem(S, x) as the child of z.

RecomputeSubproblem(R, x)

Let $R' = R \cup \{x\}$.

Base Case: All of R' is protected, i.e., $U(R') = U(R) = \emptyset$. In this case, $|R'| < B$, and both protected Cartesian trees T_R and $T_{R'}$ are empty. Return the empty set.

Case 1: x is inserted into the unprotected region of R, i.e., $U(R') = U(R) \cup \{x\}$. Then we have two cases for the root of R'.

Case 1A: x is the new root of R'. We recursively build the subtrees on the subproblems to the left and right of x. Return a tree with x as the root, ComputeCartesian($R'[-\infty, x]$) as the left child of x, and ComputeCartesian($R'[x, \infty]$) as the right child.

Case 1B: The prior root z of R remains the root of R'. Then the recursive subproblem not containing x is unchanged from R to R', and its protected Cartesian tree $\widetilde{T}$ will also be unchanged. Therefore, we can recurse only on the side containing x. If $z \prec x$, return the tree with z at the root, $\widetilde{T}$ as z's left child, and RecomputeSubroblem($R[z, \infty], x$) as z's right child. If $x \prec z$, return the tree with z at the root, $\widetilde{T}$ as z's right child, and RecomputeSubproblem($R[-\infty, z], x$) as z's left child.

Case 2: x is inserted into the protected region $P(R')$ of R'. Since R' is nontrivial, x's insertion causes one of the boundary elements y of $P(R)$ to become uncovered. We have two subcases, which mirror the subcases of Case 1.

Case 2A: y is the new root of R'. Once again, we recursively build its child subproblems. Return a tree with y as the root, ComputeCartesian($R'[-\infty, y]$) as the left child of y, and ComputeCartesian($R'[y, \infty]$) as the right child.
Notice that since y was a boundary element in $P(R)$, it is now either the smallest or largest rank element in $U(R')$. Therefore, at least one of its child subtrees will be empty.

Case 2B: The prior root z of R remains the root of R'. As in Case 1B, the tree $\widetilde{T}$ of the recursive subproblem not containing y is unchanged. If $z \prec y$, return the tree with z at the root, $\widetilde{T}$ as z's left child, and RecomputeSubproblem($R[z, \infty], y$) as z's right child. If $y \prec z$, return the tree with z at the root, $\widetilde{T}$ as z's right child, and RecomputeSubproblem($R[-\infty, z], y$) as z's left child.

Some remarks are in order as to why Cases 1A/1B and 2A/2B are the only possibilities for root selection of subproblems. That is, why can the root of R' be only the previous root z of R, the newly inserted element x (in Case 1), or the newly unprotected element y (in Case 2)? This follows from the definition of Algorithm 1. The root of any subproblem is the unprotected element with minimal hash. The only possibilities for the minimum hash element are the previous minimum hash element, or an element that has newly appeared in $U(R')$. And at most one element has newly appeared in $U(R')$: either x or y, depending on x's placement.

If z remains the root, the same cases will hold for the recursive subproblem containing x, and so on. If a Case 2 element exists in any subproblem, we call this ***the element unprotected by x*** and denote it $\boldsymbol{u(x)}$. More precisely, $u(x)$ (if it exists) is the element such that there is a subproblem $R \subseteq S$ so that $u(x)$ was a boundary element in $P(R)$, but after x's insertion, $u(x) \in U(R \cup \{x\})$. We emphasize that there can be at most one element $u(x)$, since x gets protected exactly once.

We summarize this discussion with the following useful lemma.

Lemma 3.2. *$T_S \neq T_{S'}$ if and only if exactly one of the following holds: (1) x is a node in $T_{S'}$, or (2) $u(x)$ is a node in $T_{S'}$, where $u(x)$ is the (unique) element unprotected by x's insertion.*
Moreover, if $T_S \neq T_{S'}$, then x or $u(x)$ is the unique node of minimal depth in $T_{S'} \setminus T_S$.

Lemma 3.3. *If $|S| \geq B$, the probability that $T_S \neq T_{S'}$ is $O(1/B)$.*

Proof. By Lemma 3.2, we need only to bound the probability that $x \in T_{S'}$, and if element $u(x)$ exists, that $u(x) \in T_{S'}$. We reiterate that by the strong history independence of protected Cartesian trees, whether any element is a pivot depends only on its hash value and its relative rank, but does

not depend on when it was inserted into S'. The claim follows by applying Lemma 3.1 and a union bound to x and $u(x)$. ∎

Now let's examine what happens if there is at least one new node in $T_{S'}$. In order to avoid repeatedly writing "x or $u(x)$, if $u(x)$ exists", we introduce the notation $\overline{x}$ to denote $u(x)$ if it exists and is a new pivot, and x otherwise. We reiterate that the value of $\overline{x}$ depends on x's (protected or unprotected) placement in the first subproblem (if any) in which a new pivot is found, but the value of $\overline{x}$ is well defined and unique.

We will now bound the (expected and w.h.p.) number of changes in $T_{S'}$. From the algorithm description, as soon as any node changes from T_S to $T_{S'}$, the entire subtree of this node is computed from scratch. However, as we show, many of the pivots in the tree will generally remain the same, and so most of the groups in the companion partition will be unchanged. We begin with some structural lemmas regarding the subproblems corresponding to nodes in the protected Cartesian tree. We will sometimes refer to the ***depth of a subproblem*** as the depth of its root node in the protected Cartesian tree.

For simplicity, the following lemmas consider the case where x is not the (new) root of S'. We will see in Lemma 3.13 that the case where the root changes is equivalent with slight modifications.

Lemma 3.4. *Let $S'[c,d]$ be a subproblem for $T_{S'}$ such that $c, d \in T_S$ and the newly inserted element x is not between c and d. Then $S[c,d]$ is a subproblem for T_S, and $T_{S[c,d]} = T_{S'[c,d]}$.*

Proof. Since c, d appear as pivots in both trees, the only way for $S[c,d]$ not to be a valid subproblem is if there is some element $e \in S[c,d]$ which is selected in T_S before c and d. In particular, e is not covered by c or d in S, and either $h(e) < h(c)$ or $h(e) < h(d)$. But $S'[c,d] = S[c,d]$, since x is the only new element in S' and is not in this range. So e is also not covered by c or d in S', meaning it would have also been chosen first in $T_{S'}$ since the hash values have not changed. This would contradict the fact that $S'[c,d]$ is a valid subproblem. Thus there can be no such e, and so $S[c,d]$ is a valid subproblem.

Since $S'[c,d] = S[c,d]$, $T_{S[c,d]} = T_{S'[c,d]}$ by the uniqueness of protected Cartesian trees. ∎

Lemma 3.5. *Every nontrivial subproblem in S' to the left of $\overline{x}$ has left endpoint that is either $-\infty$ or in $T_{S'} \cap T_S$. In other words, if one of the boundary nodes defining the subproblem is in $T_{S'} \setminus T_S$, it must be the right one.*

Proof. Let $S'[y,z]$ be a nontrivial subproblem such that $z \preccurlyeq \overline{x}$. If both $y, z \in T_S$, we are done, so we focus on the case that at least one of its endpoints is a new pivot in $T_{S'}$. We proceed by induction on the depth of $S'[y,z]$.

Base case: By Lemma 3.2, $\overline{x}$ is the node of lowest depth that is new in $T_{S'}$. Therefore, its left recursive subproblem is the first subproblem to its left with an endpoint in $T_{S'} \setminus T_S$. For this subproblem, $z = \overline{x}$, and y was chosen before $\overline{x}$ in $T_{S'}$ (otherwise we would not be recursing on $S'[y,\overline{x}]$), so by another application of Lemma 3.2, we must have either $y = -\infty$ or $y \in T_S$.

Inductive step: Suppose $y \neq -\infty$, and let $S'[q,w]$ be the subproblem that selects y as a root. We must show that $y \in T_S$.

If $q, w \in T_S$, then $y \in T_S$ by Lemma 3.4 and we are done, so assume at least one of q, w is not in T_S. $S'[q,w]$ has lower depth than $S'[y,z]$, so by the inductive hypothesis, $q \in T_S \cup \{-\infty\}$. Suppose for contradiction that $y \notin T_S$. Then there is some node $v \in T_S$ that covers y. We examine the cases for where v resides in S'.

Case 1: $v \in U(S'[q,w])$. Then in particular, there is an element in $U(S'[q,w])$ with lower hash than y, which contradicts the fact that y was chosen as the root of $S'[q,w]$.

Case 2: $v \in P_{S'}(q)$. Then v would also be covered by q in S since $q \in T_S \cup \{-\infty\}$.

Case 3: $v \in P'_S(w)$. First, note that we must have $z \preccurlyeq w$, i.e., an endpoint of the subproblem which picks y as a root cannot be between y and z. This is because we recurse on $S'[y, w]$ after picking y, and $S'[y, z]$ is a subproblem. Additionally, since $S'[y, z]$ is a nontrivial subproblem, $\text{rank}_{S'}(z) - \text{rank}_{S'}(y) \geq B+1$, and so $\text{rank}_{S'}(w) - \text{rank}_{S'}(y) \geq B+1$. Therefore, v cannot both cover y and be in $P'_S(w)$.

All cases ended in a contradiction, so we must have $y \in T_S$. ∎

A symmetric argument implies the corresponding lemma on the right side of $\overline{x}$.

Lemma 3.6. *Every nontrivial subproblem in S' to the right of $\overline{x}$ has right endpoint that is either ∞ or in $T_{S'} \cap T_S$. In other words, if one of the boundary nodes defining the subproblem is in $T_{S'} \setminus T_S$, it must be the left one.*

Corollary 3.7. *Let $b \in T_{S'} \setminus T_S$ be a new pivot such that $b \neq \overline{x}$. Then the child subtree of b that is closer to $\overline{x}$ is empty.*

Proof. Write $S'[w, z]$ for the subproblem that selects b as its root. Since b is a new pivot, we know at least one of w, z is not in T_S by Lemma 3.4. By Lemma 3.2, b has greater depth than $\overline{x}$, and so both w and z are on the same side of $\overline{x}$ (because we recurse on either side after selecting $\overline{x}$). Suppose that it is the left side. By Lemma 3.5, $w \in T_S$ and $z \notin T_S$. Then Lemma 3.5 also says that $S'[b, z]$ cannot be a nontrivial subproblem, so b's right subtree is empty. The case that b is on $\overline{x}$'s right is symmetric. ∎

Lemma 3.8. *Suppose $T_S \neq T_{S'}$. Then there is a path from $\overline{x}$ to the leaf level containing all changed pivots to the left of $\overline{x}$, and a path from $\overline{x}$ to the leaf level containing all changed pivots to the right of $\overline{x}$.*

Proof. We define the left path recursively from $\overline{x}$. Let x_1 be $\overline{x}$'s left child in $T_{S'}$. If $x_1 \in T_S$, then by Lemma 3.4, there are no changed pivots to the left of x_1, so we continue the path to $\overline{x}$'s right and select x_2 as x_1's right child. If instead $x_1 \notin T_{S'}$, then by Lemma 3.7, x_1 has only one child, so we select this child as x_2. This path contains all changed pivots by construction, since every time we select a direction in the path, the other direction either contains no changes or an empty subtree. The path to the right of $\overline{x}$ is symmetric. ∎

This implies a bound on the total cost to maintain protected Cartesian trees, which is proved in Appendix A.3.2 in the interest of space.

Corollary 3.9. *The protected Cartesian tree T_S on S can be maintained with CPU cost $O(1 + \log(N/B)/B)$ in expectation.*

Recall that we analyze protected Cartesian trees in order to bound the costs of our dynamic partitioning algorithm. When analyzing the number of pivots that change after an insert (equivalently, the number of group boundaries that change), we can in fact show much tighter bounds on the number of changes, as we shall see in the remainder of this section.

We need some additional structural lemmas on the configuration of pivots in order to bound their changes. The next lemma says that when viewed in rank order, all changed pivots must be contiguous (i.e., have no unchanged pivots between them.)

Lemma 3.10. *Let $T_{S'} \setminus T_S = \{y_1, \ldots, y_k\}$ be the set of new pivots, and let $z \in T_{S'} \cap T_S$. Then $z \prec y_1$ or $y_k \prec z$.*

Proof. Suppose for the sake of contradiction that there is some pivot $z \in T_S \cap T_{S'}$ with $y_j, y_{j+1} \in T_{S'} \setminus T_S$ such that $y_j \prec z \prec y_{j+1}$. Suppose also without loss of generality that $y_{j+1} \preccurlyeq \overline{x}$, so that these three pivots appear to the left of $\overline{x}$ in S'. If there exist unchanged pivots to the left of y_j, there is

some pivot $w \in T_S \cap T_{S'}$ with $w \prec y_j$. If all pivots to the left are new, set $w = -\infty$. We can then apply Lemma 3.4 to say that the subproblem $S'[w, z]$ is also a subproblem in S, and will have an identical subtree. But this contradicts the assumption that $y_j \notin T_S$. A symmetric argument shows that the pivots to the right of $\overline{x}$ are also contiguous. ∎

Lemma 3.11. *Let $T_{S'} \setminus T_S = \{y_1, \ldots, y_k\}$ be the set of new pivots. Then their hash values are monotonically decreasing until $\overline{x}$, and monotonically increasing after $\overline{x}$. In other words, $h(y_1) > h(y_2) > \cdots > h(y_{i-1}) > h(\overline{x}) < h(y_{i+1}) < \cdots < y_k$ for some i.*

Proof. Recall that the fact that $y_i = \overline{x}$ for some i follows from Lemma 3.2. By Lemma 3.8, there is a path downwards from $\overline{x}$ containing all of $y_1, y_2, \cdots, y_{i-1}$. By definition of a protected Cartesian tree, any path downwards contains pivots with hashes in increasing order. So it remains to show that the y_j to the left of $\overline{x}$ appear in decreasing order by rank down this path.

Suppose for contradiction that y_j has lower depth than y_ℓ for some $y_j \prec y_\ell \prec \overline{x}$. Since y_j is selected before y_ℓ, we know that y_ℓ was selected as the root of some subproblem $S'[w, z]$, with $y_j \preccurlyeq w$. But this contracts Corollary 3.7, which says that y_j must have an empty right subtree. The chain on the other side of $\overline{x}$ follows by symmetry. ∎

Lemma 3.12. *If $T_{S'} \neq T_S$, the probability of at least k new pivots in $T_{S'}$ is at most $\frac{1}{(\lfloor k/2 \rfloor - 1)!}$.*

Proof. By Lemma 3.2, $\overline{x} \in T_{S'}$. With k new pivots, there must be at least $j = \lfloor k/2 \rfloor$ pivots on at least one of the two sides of $\overline{x}$. Assume, w.l.o.g., that this occurs on the left side, and write $y_1, y_2, \ldots, y_j = \overline{x}$ as the new pivots. By Lemma 3.11, we have $h(y_1) > h(y_2) > \cdots > h(\overline{x})$. First, we show by induction that for each $i \geq 2$, $h(y_i)$ has the minimal hash in $S'[y_1, y_i]$. For the base case, consider $S'[y_1, y_2]$. First, we claim that at least one of y_1, y_2 has the minimal hash in $S'[y_1, y_2]$. This is true of any restriction $S[u, v]$, because at the time the first of the two endpoints is selected as a pivot, all elements in the subproblem are uncovered (and thus would have been selected as a pivot instead if their hash was lower.) And since $h(y_2) < h(y_1)$, it must be the element with minimal hash. Now assume that for any $z \in S'[y_1, y_i]$, $h(y_i) < h(z)$. By an identical argument to the base case, $h(y_{i+1}) < h(z)$ for any $z \in S'[y_i, y_{i+1}]$. And $h(y_{i+1}) < h(y_i)$, so $h(y_{i+1})$ has the minimal hash in $S'[y_1, y_i] \cup S'[y_i, y_{i+1}] = S'[y_1, y_{i+1}]$.

Now it remains to bound the probability of a sequence of new pivots in S' with the above property. By the above argument, the new pivot y_j has the minimal hash of $S'[y_1, y_j]$, which contains at least $\frac{B(j-1)}{2}$ elements. Since all hash values are random, any individual element has the minimal hash over this range with probability $\frac{2}{B(j-1)}$. By the union bound, the minimal hash element in this region is one of the $B/2$ elements with ranks in $[\text{rank}^r(\overline{x}) - B/2, \text{rank}^r(\overline{x})]$ with probability at most $\frac{1}{j-1}$. Then by an identical argument, the probability given y_j of a new pivot y_{j-1} that is lower than all $\frac{B(j-2)}{2}$ elements to its left is at most $\frac{1}{j-2}$. The probability of both such pivots is then $\frac{1}{j-1} \cdot \frac{1}{j-2}$. Continuing inductively, the probability of the entire chain is at most $\frac{1}{(j-1)!} = \frac{1}{(\lfloor k/2 \rfloor - 1)!}$. ∎

Lemma 3.13. *Let S be a set with $|S| \geq B$, and T_S be the protected Cartesian tree on S with random hash function h. Then the number of pivots that change in T_S after an insertion into S is $O(1/B)$ in expectation and $O(\log N / \text{loglog } N)$ with high probability in N.*

Proof. Let $x \in \mathcal{U} \setminus S$, and $S' = S \cup \{x\}$. First consider the case that x is not the root of S'. Let X be the random variable for the number of pivots in $T_{S'} \setminus T_S$, conditioned on $T_{S'} \neq T_S$. We compute $\mathbb{E}[X]$ using Lemma 3.12.

$$\mathbb{E}[X] = \sum_{k=1}^{|T_{S'}|} k \Pr[X \geq k] \leq \sum_{k=1}^{\infty} O\left(\frac{k}{(\lfloor k/2 \rfloor - 1)!}\right) = O(1).$$

By Lemma 3.3, $T_{S'} \neq T_S$ with probability $O(1/B)$. Thus the overall expected number of pivots in $T_{S'} \setminus T_S$ is $O(1/B)$.

Let $c \geq 2$, and let $n = \frac{2c \log N}{\log\log N}$. By Lemma 3.12, the probability of n or more new pivots is at most $\frac{1}{(\lfloor n/2 \rfloor - 1)!} = 2^{-O(n \log n)} = 2^{-O(c \log N)} = O(N^{-c})$.

In the case that x is the root of S', the preceding arguments regarding the chains of changed pivots which must have monotonic hash values still hold, except that the rotation of the set, and thus the definition of left and right, have changed. This may break the monotonic chains into three pieces instead of two. The same bounds hold asymptotically by replacing $k/2$ with $k/3$. ∎

3.3 Deletions

Let $S \subseteq \mathcal{U}$, let $y \in S$, and let $\tilde{S} = S \setminus \{y\}$. Fix the randomness involved in the construction of protected Cartesian trees. Then by the strong history independence of our construction, our algorithm yields a unique T_S and a unique $T_{\tilde{S}}$, no matter the order of operations. By Lemma 3.13, there are $O(1/B)$ changed pivots from $T_{\tilde{S}}$ to T_S in expectation, and $O(\log N/\log\log N)$ with high probability. Thus, we can conclude that the reverse operation of removing y yields the same number of changes.

Corollary 3.14. *Let S be set with $|S| > B$, and T_S, the protected Cartesian tree on S with random hash function h. Then the number of pivots that change in T_S after a deletion is $O(1/B)$ in expectation and $O(\log N/\log\log N)$ with high probability in N.*

3.4 Reduction From Modular Partitioning to Linear Partitioning

To transform a modular partition into a linear partition, we need a scheme to separate the first and last groups. We base our $(B, 2)$-partition on a $(B, 3)$-partition, as described in the proof of the following theorem.

Theorem 3.15 ($(B, 3) - partitioning$). *History-independent $(B, 3)$-partitioning of a set S of size $N \geq B$ can be maintained with: group-update cost $O(1/B)$ per insertion/deletion in expectation and $O(\log N/\log\log N)$ with high probability in N; and element-movement cost $O(1)$ amortized per insertion/deletion in expectation, and $O((B \log N)/\log\log N)$ with high probability in N.*

Proof. The $(B, 3)$-partition follows from the modular partition, above. First run modular $(B', 2)$-partitioning with $B' = 2B/3$. This produces groups of size between $B'/2 = B/3$ and $B' = 2B/3$. Then split the group that crosses the modular boundary obtain a linear partition. If the first (resp. last) group is smaller than $B/3$, merge it with its neighboring group. This yields groups as large as B and as small as $B/3$, yielding a linear $(B, 3)$-partition.

Recall that pivot changes in the protected Cartesian tree correspond to group boundary changes in the dynamic partitioning scheme. Therefore, Lemma 3.13 and Corollary 3.14 immediately imply the group-update costs.

Each group boundary change causes up to B elements to change groups, so applying a factor B to the group-update costs gives the element-movement costs. ∎

To obtain a linear $(B, 2)$-partition, run the (B, α)-partitioning scheme detailed in Appendix A.1 with $\alpha = 2$. Therefore Theorem A.2 immediately implies Theorems 1.2 and Theorem 1.3.

4 HISTORY INDEPENDENT B-TREES

In this section, we construct a dynamic history-independent B-tree using dynamic partitioning. Let $S \subseteq \mathcal{U}$. We construct the B-tree $\mathcal{B}(S)$ as follows.

(1) Build $T_{S_0} = T_S$, the protected Cartesian tree on S, using random hash function h_0. Let $S_1 = \{p_1, \ldots, p_k\}$ denote the set of pivots in T_S, where the p_i are ordered with respect to the ordering on S. Each element in S_1 is a boundary element of the leaf nodes in $\mathcal{B}(S)$. That is, the i-th leaf of $\mathcal{B}(S)$ contains, in rank order, all elements $y \in S$ such that $\text{rank}(p_i) \leq \text{rank}(y) < \text{rank}(p_{i+1})$ (again with the convention that $p_0 = -\infty$.)
(2) Build T_{S_1}, the protected Cartesian tree on S_1, using random hash function h_1 which is independent of h_0. Let $S_2 = \{q_1, \ldots, q_j\}$ denote the set of pivots of T_{S_1} in sorted order. The nodes at height 1 of $\mathcal{B}(S)$ are defined as the elements in S_1 partitioned by the elements of S_2, as above. The parent of the leaf node with left boundary element y is the node at height 1 containing y.
(3) Repeat this process, building the height i nodes from T_{S_i}, until $|S_i| < B$, at which point S_i becomes the root of $\mathcal{B}(S)$.

The use of our history-independent $(B, 2)$ partitioning scheme to build $\mathcal{B}(S)$ immediately implies the following lemmas.

Lemma 4.1. *$\mathcal{B}(S)$ is strongly history independent.*

Lemma 4.2. *Each node in $\mathcal{B}(S)$ contains between $B/2$ and B elements.*

Corollary 4.3. *$\mathcal{B}(S)$ has height $O(\log_B(N))$ deterministically.*

Proof. By Lemma 2.4, $\frac{|T(S_i)|}{|T(S_{i-1})|} = \Theta(1/B)$ for all i. Thus, the process to construct $\mathcal{B}(S)$ terminates in $O(\log_B(N))$ rounds. ∎

Corollary 4.4. *Each search operation in $\mathcal{B}(S)$ has worst cast CPU cost $O(\log(N))$ and worst case I/O cost $O(\log_B(N))$.*

Proof. The I/O cost follows from the height of $\mathcal{B}(S)$. And since the pivots in any node of $\mathcal{B}(S)$ are in rank order, we can perform rank operations in each node in $O(\log B)$ CPU operations. ∎

The following theorem is a consequence of Theorem 1.2 on the group-update cost of dynamic partitioning. We consider the costs incurred by the protected Cartesian trees used to build all levels of a history-independent B-tree, and the way that pivot changes affect higher levels of the B-tree.

Theorem 4.5. *Each insertion/deletion in $\mathcal{B}(S)$ has I/O cost that is the search cost of $O(\log_B N)$, plus an additional update cost of $O(\log_B(N)/B)$ in expectation and $O\left(\frac{\log N}{\log \log B}\right)$ with high probability in N.*

Proof. We refer to insertions in this proof for convenience, but deletions will be symmetric by Corollary 3.14.

We insert the new element in its location, which requires a search from the root to the target leaf. This may induce node splits or merges. If there are no node splits or merges (i.e., no group updates), then there are no additional I/Os for our insert. On the other hand, if there are additional splits/merges at a level, then we bound the I/O cost of these by the group-update cost at that level.

Let $x \in \mathcal{U} \setminus S$, $S' = S \cup \{x\}$. Let $\ell = \Theta(\log_B(N))$ denote the height of $\mathcal{B}(S')$, and let X_i be a random variable for the number of new pivots at height i of $\mathcal{B}(S')$ compared to $\mathcal{B}(S)$. We want to bound $\mathbb{E}[X]$ for $X = X_0 + \cdots + X_\ell$.

Let $k > 0$. Since the protected Cartesian tree selects one pivot every $B/2$ to B elements, in the event that $X_0 = k$, we know that $\Theta(k/B)$ of these are new pivots in T_{S_1}. This is because the k new pivots from the leaf level are immediately new elements in the set S_1, and they form a contiguous run by Lemma 3.10. Therefore, $\Theta(k/B)$ of them are selected as pivots in T_{S_1}. We refer to these as the

automatic pivots at height 1. Similarly, we have $\Theta(k/B^i)$ automatic pivots in T_{S_i}. So in aggregate, when $X_0 = k$, the k new pivots at level 0 of the tree result in $O(k)$ automatic pivots in $\mathcal{B}(S)$.

Let Y_1 be the random variable for the new pivots in T_{S_1} that are not automatic. Once again, k' new pivots counted in Y_1 will induce $O(k')$ automatic new pivots at higher levels. In general, let Y_i be the new pivots that are not automatic from any lower levels, and let $Y = Y_0 + Y_1 + \cdots Y_l$. By the preceding argument, $X = O(Y)$.

Pivot changes form a contiguous run by Lemma 3.10. Since Y_i only counts non-automatic pivots, it is equivalent to counting the possible pivot changes that result from two insertions: the leftmost and rightmost new pivots in S_i. Since each insert into a protected Cartesian tree has expected cost $O(1/B)$, we have $\mathbb{E}[Y_i] \le 2\mathbb{E}[X_i] = O(1/B)$. Therefore, $\mathbb{E}[Y] = O((\log_B N)/B)$. In particular, this term is asymptotically smaller than the search cost.

To get high probability bounds, we union bound over the two non-automatic inserts in Y_i that are to the left and right of the automatic pivots. By Lemma 3.12, $\Pr[Y_i \ge k] \le 2\Pr[X_i \ge k] \le \frac{2}{(\lfloor k/2 \rfloor - 1)!}$. We want to bound the probability that $Y = \sum Y_i \ge \frac{\log N}{\log\log B}$. The Y_i are independent by construction, so for any values $a_1, \ldots, a_\ell$,

$$\Pr[(Y_1 = a_1) \wedge \cdots \wedge (Y_\ell = a_\ell)] = \prod_{i=1}^{\ell} \Pr[(Y_i = a_i)].$$

Write $L = \frac{c \log N}{\log\log B}$, for some constant c. By convexity, the probability that $Y = L$ is maximized when all of the a_i are equal to $a = L/\ell$. This has probability

$$\prod_{i=1}^{\ell} \Pr[(Y_i = a)] \le \left(\frac{1}{(\lfloor a/2 \rfloor - 1)!}\right)^{\ell} = 2^{-O(\ell(a \log a))} = 2^{-O(L \log(L/\ell))}.$$

Also,

$$L \log(L/\ell) = \frac{c \log N}{\log\log B} \log\left(\frac{c \log N}{\log\log B} \cdot \frac{\log B}{\log N}\right) = \frac{c \log N}{\log\log B} \log\left(\frac{c \log B}{\log\log B}\right) = \Theta(c \log N),$$

so we can bound the above probability by $N^{-c'}$ for some constant c'. Finally, we union bound over the polynomially many different combinations of ℓ values that sum to L, so picking constants appropriately, the probability that $Y \ge L$ is bounded by $N^{-c''}$. ∎

The following results from Theorem 4.5, and the fact that we perform $O(B)$ CPU operations within each block when updating group boundaries.

Theorem 4.6. *Each insertion/deletion in $\mathcal{B}(S)$ has CPU cost that is the search cost of $O(\log N)$, plus an additional update cost of amortized $O(\log_B(N))$ in expectation and $O\left(\frac{B \log N}{\log\log B}\right)$ with high probability in N.*

ACKNOWLEDGMENTS

We gratefully acknowledge Alex Conway for his helpful insights.

This work was supported by NSF grants CCF-2212129, CCF-2106999, CCF-2118620, CNS-1938180, CCF-2118832, CCF-2106827, CNS-1938709, and CCF-2247577. Hanna Komlós was also partially funded by the Graduate Fellowships for STEM Diversity.

REFERENCES

[1] Umut A Acar, Guy E Blelloch, Robert Harper, Jorge L Vittes, and Shan Leung Maverick Woo. 2004. Dynamizing static algorithms, with applications to dynamic trees and history independence. In *Proc. of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 531–540.

[2] Ole Amble and Donald Ervin Knuth. 1974. Ordered hash tables. *Comput. J.* 17, 2 (1974), 135–142.

[3] Arne Andersson, Peter Bro Miltersen, and Mikkel Thorup. 1999. Fusion Trees Can Be Implemented With AC0 Instructions Only. *Theoretical Computer Science* 215, 1-2 (1999), 337–344. https://doi.org/10.1016/S0304-3975(98)00172-8

[4] Arne Andersson and Thomas Ottmann. 1991. Faster uniquely represented dictionaries. In *Proc. of the 32nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 642–649.

[5] Arne Andersson and Thomas Ottmann. 1995. New tight bounds on uniquely represented dictionaries. *SIAM J. Comput.* 24, 5 (1995), 1091–1103.

[6] Arne Andersson and Mikkel Thorup. 2007. Dynamic Ordered Sets with Exponential Search Trees. *J. ACM* 54, 3 (2007), 1–40.

[7] Cecilia R Aragon and Raimund G Seidel. 1989. Randomized search trees. In *Proc. of the 30th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 540–545.

[8] Sumeet Bajaj, Anrin Chakraborti, and Radu Sion. 2015. The Foundations of History Independence. *arXiv preprint arXiv:1501.06508* (2015).

[9] Sumeet Bajaj, Anrin Chakraborti, and Radu Sion. 2016. Practical Foundations of History Independence. *IEEE Trans. Inf. Forensics Secur.* 11, 2 (2016), 303–312. https://doi.org/10.1109/TIFS.2015.2491309

[10] Sumit Bajaj and Radu Sion. 2013. Ficklebase: Looking into the future to erase the past. In *Proc. of the 29th IEEE International Conference on Data Engineering (ICDE)*. 86–97.

[11] Sumeet Bajaj and Radu Sion. 2013. HIFS: History independence for file systems. In *Proc. of the ACM SIGSAC Conference on Computer & Communications Security (CCS)*. 1285–1296.

[12] Rudolf Bayer and Edward M. McCreight. 1972. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica* 1, 3 (Feb. 1972), 173–189. https://doi.org/10.1145/1734663.1734671

[13] Michael A. Bender, Jonathan W. Berry, Rob Johnson, Thomas M. Kroeger, Samuel McCauley, Cynthia A. Phillips, Bertrand Simon, Shikha Singh, and David Zage. 2016. Anti-Persistence on Persistent Storage: History-Independent Sparse Tables and Dictionaries. In *Proc. 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS)*. 289–302.

[14] Michael A Bender, Richard Cole, Erik D Demaine, Martin Farach-Colton, and Jack Zito. 2002. Two simplified algorithms for maintaining order in a list. In *Proc. 10th European Symposium on Algorithms (ESA)*. Springer, 152–164.

[15] Michael A. Bender, Alex Conway, Martín Farach-Colton, Hanna Komlós, William Kuszmaul, and Nicole Wein. 2022. Online List Labeling: Breaking the $\log^2 n$ Barrier. In *Proc. 63rd IEEE Annual Symposium on Foundations of Computer Science (FOCS)*. 980–990. https://doi.org/10.1109/FOCS54457.2022.00096

[16] Michael A Bender, Erik D Demaine, and Martin Farach-Colton. 2000. Cache-oblivious B-trees. In *Proc. of the 41st Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 399–409.

[17] Michael A. Bender and Martin Farach-Colton. 2000. The LCA Problem Revisited. In *Proc. Latin American Theoretical INformatics (LATIN)*. 88–94.

[18] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. 2007. Cache-oblivious streaming B-trees. In *SPAA*. ACM, 81–92.

[19] Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. 2015. An Introduction to B^ϵ-Trees and Write-Optimization. *:login; magazine* 40, 5 (Oct. 2015), 22–28.

[20] John Bethencourt, Dan Boneh, and Brent Waters. 2007. Cryptographic methods for storing ballots on a voting machine. In *Proc. of the 14th Network and Distributed System Security Symposium (NDSS)*.

[21] Guy E Blelloch and Daniel Golovin. 2007. Strongly history-independent hashing with applications. In *Proc. 48th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 272–282.

[22] Gerth Stølting Brodal and Rolf Fagerberg. 2003. Lower Bounds for External Memory Dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '03)*. Baltimore, MD, 546–554.

[23] Niv Buchbinder and Erez Petrank. 2003. Lower and Upper Bounds on Obtaining History Independence. In *Advances in Cryptology*. 445–462.

[24] Pedro Celis, Per-Åke Larson, and J. Ian Munro. 1985. Robin Hood Hashing (Preliminary Report). In *26th Annual Symposium on Foundations of Computer Science (FOCS'85)*. Portland, Oregon, USA, 281–288. https://doi.org/10.1109/SFCS.1985.48

[25] Bo Chen and Radu Sion. 2015. HiFlash: A History Independent Flash Device. *CoRR* abs/1511.05180 (2015). arXiv:1511.05180 http://arxiv.org/abs/1511.05180

[26] Douglas Comer. 1979. The Ubiquitous B-Tree. *Comput. Surveys* 11, 2 (June 1979), 121–137.

[27] Paul Dietz and Daniel Sleator. 1987. Two algorithms for maintaining order in a list. In *Proc. of the 19th Annual ACM Symposium on Theory of Computing (STOC)*. 365–372.

[28] M. L. Fredman and D. E. Willard. 1993. Surpassing the Information Theoretic Bound with Fusion Trees. *J. Comput. System Sci.* 47 (1993), 424–436.

[29] Harold N. Gabow, Jon Louis Bentley, and Robert Endre Tarjan. 1984. Scaling and Related Techniques for Geometry Problems. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1984, Washington, DC, USA*, Richard A. DeMillo (Ed.). ACM, 135–143. https://doi.org/10.1145/800057.808675

[30] Sanjam Garg, Shafi Goldwasser, and Prashant Nalini Vasudevan. 2020. Formalizing data deletion in the context of the right to be forgotten. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 373–402.

[31] Daniel Golovin. 2008. *Uniquely Represented Data Structures with Applications to Privacy.* Ph. D. Dissertation. Carnegie Mellon University, Pittsburgh, PA, 2008.

[32] Daniel Golovin. 2009. B-treaps: A uniquely represented alternative to B-Trees. In *Proc. 36th Annual International Colloquium on Automata, Languages, and Programming (ICALP)*. 487–499.

[33] Daniel Golovin. 2010. The B-skip-list: A simpler uniquely represented alternative to B-trees. *arXiv preprint arXiv:1005.0662* (2010).

[34] Michael T. Goodrich, Evgenios M. Kornaropoulos, Michael Mitzenmacher, and Roberto Tamassia. 2017. Auditable Data Structures. In *Proc. IEEE European Symposium on Security and Privacy (EuroS&P)*. 285–300. https://doi.org/10.1109/EuroSP.2017.46

[35] Jason D. Hartline, Edwin S. Hong, Alexander E. Mohr, William R. Pentney, and Emily Rocke. 2002. Characterizing History Independent Data Structures. In *Proceedings of the Algorithms and Computation, 13th International Symposium (ISAAC)*. 229–240. https://doi.org/10.1007/3-540-36136-7_21

[36] Jason D Hartline, Edwin S Hong, Alexander E Mohr, William R Pentney, and Emily C Rocke. 2005. Characterizing history independent data structures. *Algorithmica* 42, 1 (2005), 57–74.

[37] Scott Huddleston and Kurt Mehlhorn. 1982. A New Data Structure for Representing Sorted Lists. *Acta Informatica* 17 (1982), 157–184.

[38] Bert-Jaap Koops. 2011. Forgetting footprints, shunning shadows: A critical analysis of the right to be forgotten in big data practice. *SCRIPTed* 8 (2011), 229.

[39] William Kuszmaul. 2023. Strongly History Independent Storage Allocation: New Upper and Lower bounds. In *Proc. 64rd IEEE Annual Symposium on Foundations of Computer Science (FOCS)*.

[40] Daniele Micciancio. 1997. Oblivious data structures: applications to cryptography. In *Proc. 29th Annual ACM Symposium on Theory of Computing (STOC)*. 456–464.

[41] Tal Moran, Moni Naor, and Gil Segev. 2007. Deterministic history-independent strategies for storing information on write-once memories. In *Proc. 34th International Colloquium on Automata, Languages and Programming (ICALP)*.

[42] Kiyosh Murata and Yohki Orito. 2011. The right to forget/be forgotten. *Ethics in Interdisciplinary and Intercultural Relations* 192 (2011).

[43] Moni Naor, Gil Segev, and Udi Wieder. 2008. History-independent cuckoo hashing. In *Proc. of the 35th International Colloquium on Automata, Languages and Programming (ICALP)*. Springer, 631–642.

[44] Moni Naor and Vanessa Teague. 2001. Anti-persistence: history independent data structures. In *Proc. 33rd Annual ACM Symposium on Theory of Computing (STOC)*. 492–501.

[45] Patrick O'Neil, Edward Cheng, Dieter Gawlic, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385. https://doi.org/10.1007/s002360050048

[46] Mihai Patrascu and Mikkel Thorup. 2014. Dynamic Integer Sets with Optimal Rank, Select, and Predecessor Search. In *55th IEEE Symp. on Foundations of Computer Science (FOCS)*. 166–175.

[47] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. 2016. Arx: A Strongly Encrypted Database System. *IACR Cryptol. ePrint Arch.* (2016), 591. http://eprint.iacr.org/2016/591

[48] William Pugh. 1988. *Incremental computation and the incremental evaluation of functional programs.* Ph. D. Dissertation. Cornell University.

[49] William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (1990), 668–676.

[50] William Pugh and Tim Teitelbaum. 1989. Incremental computation via function caching. In *Proc. of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 315–328.

[51] Rajeev Raman. 1996. Priority Queues: Small, Monotone and Trans-dichotomous. In *4th European Symposium on Algorithms (ESA)*. 121–137.

[52] Daniel S Roche, Adam J Aviv, and Seung Geol Choi. 2015. Oblivious Secure Deletion with Bounded History Independence. *arXiv preprint arXiv:1505.07391* (2015).

[53] Daniel S. Roche, Adam J. Aviv, and Seung Geol Choi. 2016. A Practical Oblivious Map Data Structure with Secure Deletion and History Independence. In *IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 178–197.

https://doi.org/10.1109/SP.2016.19

[54] Russell Sears, Mark Callaghan, and Eric Brewer. 2008. Rose: Compressed, log-structured replication. *Proceedings of the VLDB Endowment* 1, 1 (2008), 526–537.

[55] Sandeep Sen. 1991. Some observations on skip-lists. *Inform. Process. Lett.* 39, 4 (1991), 173–176. https://doi.org/10.1016/0020-0190(91)90175-H

[56] Lawrence Snyder. 1977. On uniquely represented data structures. In *Proc. of the 18th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 142–146.

[57] Rajamani Sundar and Robert Endre Tarjan. 1990. Unique binary search tree representations and equality-testing of sets and sequences. In *Proc. of the 22nd Annual ACM Symposium on Theory of Computing (STOC)*. 18–25.

[58] Mikkel Thorup. 2003. On AC0 Implementations of Fusion Trees and Atomic Heaps. In *14th ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 699–707. http://portal.acm.org/citation.cfm?id=644108.644221

[59] Athanasios K. Tsakalidis. 1984. Maintaining Order in a Generalized Linked List. *Acta Informatica* 21, 1 (May 1984), 101–112.

[60] Theodoros Tzouramanis. 2012. History-independence: a fresh look at the case of R-trees. In *Proc. 27th Annual ACM Symposium on Applied Computing (SAC)*. 7–12.

[61] Jean Vuillemin. 1980. A Unifying Look at Data Structures. *Commun. ACM* 23, 4 (1980), 229–239. https://doi.org/10.1145/358841.358852

[62] Robert Kirk Walker. 2012. The right to be forgotten. *Hastings LJ* 64 (2012), 257.

[63] Dan E. Willard. 1999. Examining Computational Geometry, van Emde Boas Trees, and Hashing from the Perspective of the Fusion Tree. *SIAM J. Comput.* 29 (December 1999), 1030–1049. Issue 3. https://doi.org/10.1137/S0097539797322425

A APPENDIX

In this appendix, we provide details that were omitted in the body of this paper due to space reasons.

A.1 (B, α)-Partitioning

In this section, we generalize Theorem 3.15, which gives a linear $(B, 3)$-partitioning, to yield a linear (B, α)-partitioning for any $\alpha = 1 + \Theta(1)$. Recall that in a (B, α)-partition, we must maintain groups G of size $\frac{B}{\alpha} \leq |G| \leq B$. Since a (B, α)-partition immediately gives a (B, α')-partition of values of $\alpha' > \alpha$, we restrict our attention to the range $1 < \alpha \leq 2$.

We define the following values:

- $\gamma = \alpha - 1$
- $\alpha' = 1 + \gamma/2$
- $\alpha'' = 1 + \gamma/4$
- $B' = B\alpha''/\alpha'$
- $c = \frac{24}{\gamma}$
- $d = \frac{\gamma^2}{200\alpha''}$

We will also use the notation $[n]$ to denote the set $\{1, 2, \ldots, n\}$.

We first describe an algorithm for (B, α)-partitioning. The algorithm has two random blurring parameters. These are not needed for correctness, which is established in Lemma A.1. They will be used to establish performance bounds, which are established in Theorem A.2.

Dynamic (B, α)-Partitioning. Let $S \subseteq \mathcal{U}$ be a dynamic set.

(1) Run dynamic $(cB', 3)$-partitioning on S to produce supergroups $F_1, \ldots, F_h$.
(2) Run dynamic $(dB', 3)$-partitioning within each supergroup F_i to produce minigroups H_ℓ^i.
(3) For each supergroup F_i, let $M_i = |F_i|$. Set $k_i = \left\lceil \frac{M_i + r_i}{B'} \right\rceil$ to be the total number of groups that we generate, where $r_i \in [B']$ is a random blurring parameter that is determined based on a hash function of the first pivot in F_i.
(4) Set $\tau_i = \frac{M_i}{k_i}$ to be the ***minimum threshold*** of each group. Our construction will guarantee that all but the last group has size at least τ_i, and we will give a lower bound on how much the size of the last group dips below τ_i.
(5) For each supergroup F_i, form groups G_i^j by greedily gluing together adjacent minigroups H_i^ℓ from left to right until each group has at least $\tau_i^j = \tau_i + s_i^j$ elements, where $s_i^j \in [dB']$ in each group is a random blurring parameter determined by hashing the value of the first pivot of the group G_i^j. We refer to τ_i^j as the ***threshold*** for group G_i^j. The last group comprises the left-over elements in F_i.
(6) The resulting groups G_i^j form the (B, α)-partition.

Lemma A.1. *This algorithm produces a dynamic (B, α)-partition.*

Proof. The algorithm is strongly history independent, since the supergroups and minigroups are produced by a strongly history independent algorithm, and the groups are determined only by the number of elements in each minigroup and the blurring values, which are based on hashes of their elements.

We need to argue that this algorithm produces a valid (B, α)-partition, that is, that each group G_i has between B/α and B elements. Fix a supergroup F_i. For ease of notation, we will drop the subscript i and write $F = F_i$, $G^j = G_j^i$, $M = M_i$, $k = k_i$, $r = r_i$, $\tau = \tau_i$ and $s^j = s_i^j$.

First we argue that each group formed within F has size at most B. By construction, $k \geq \frac{M}{B'}$ and so $\tau \leq B'$. Additionally, we add minigroups to group G^j until we are *over* its threshold

$\tau^j = \tau + s^j \le \tau + dB'$. And since minigroups are produced by a $(dB', 3)$-partition, they have size at most dB'. Therefore, the size of a group is at most

$$\tau + 2dB' \le B'(1+2d) = B \cdot \frac{\alpha''}{\alpha'}\left(1 + \frac{\gamma^2}{100\alpha''}\right) = B \cdot \frac{100\alpha'' + \gamma^2}{100\alpha'} = B \cdot \frac{100 + 25\gamma + \gamma^2}{100 + 50\gamma} < B,$$

where the last inequality follows since $\alpha \le 2$ means $\gamma \le 1$, and thus $\gamma^2 < 25\gamma$.

Now we must show that each group has size at least B/α. There are two things to prove: (1) our minimum threshold τ produces groups that are large enough, and (2) the last group in F (the group with the highest ranks) is not too small.

Let us first consider a group G^j that is not the last one. By construction $k \le \left\lceil \frac{M+r}{B'} \right\rceil \le \frac{M+2B'}{B'}$ and so $|G^j| \ge \tau \ge \frac{MB'}{M+2B'}$. Therefore,

$$\tau \ge \frac{MB'}{M+2B'} = \frac{B'(M+2B') - 2(B')^2}{M+2B'} = B' - \frac{2(B')^2}{M+2B'}.$$

Since the supergroups are produced from a $(cB', 3)$-partition, we have that $M \ge cB'/3 = \frac{8B'}{\gamma}$. Therefore,

$$\frac{2(B')^2}{2B'+M} \le \frac{2(B')^2}{2B' + 8B'/\gamma} = \frac{B'\gamma}{\gamma+4} = \frac{B'\gamma/4}{\gamma/4+1} = \frac{B'(\alpha''-1)}{\alpha''} = B'(1 - 1/\alpha'').$$

Putting these together, we have that the number of elements in G^j is at least

$$|G^j| \ge \tau \ge B' - \frac{(B')^2}{M+B'} = B' - B'(1-1/\alpha'') = \frac{B'}{\alpha''} = \frac{B}{\alpha'} > \frac{B}{\alpha}, \tag{1}$$

as desired.

Finally, let us consider the last group. We set our minimum threshold τ so that if all prior groups had exactly τ elements, the last group of remaining elements would also have size τ. Each of the previous $k-1$ groups can exceed their thresholds τ^j by at most dB', and so they can exceed the minimum threshold τ by at most $dB' + s^j \le 2dB'$. Therefore, the last group can underflow the minimum threshold by at most $2dB'(k-1)$, meaning that it has size at least $\tau - 2dB'(k-1)$.

Note that in the case that τ is fractional, we may be concerned about accumulations from extra elements in each prior minigroup due to rounding up from their thresholds. In fact, since the minigroup size is always under $\tau + s^j$ *without* the first element of the minigroup (or else we would not have added the last minigroup), the excess number of elements from the last minigroup is at most $dB' - 1$. Therefore $\tau + s^j + dB' \le \tau + 2dB'$ is an accurate bound on the number of elements in each prior minigroup including rounding.

The last group therefore has size at least

$$\begin{aligned}
\tau - 2dB'(k-1) &\geq \tau - 2dB'\left(\frac{M}{B'}+1\right) && \text{(Since } k \leq \tfrac{M}{B'}+2\text{)}\\
&\geq \frac{B}{\alpha'} - 2d(M+B') && \text{(Since } \tau \geq B/\alpha' \text{ by Equation 1)}\\
&\geq \frac{B}{\alpha'} - 2d\left(\frac{24B'}{\gamma}+B'\right) && \text{(Since } M \leq cB' = 24B'/\gamma\text{)}\\
&= \frac{B}{\alpha'} - \frac{\gamma^2}{100\alpha''}\left(\frac{24B'+\gamma B'}{\gamma}\right)\\
&\geq \frac{B}{\alpha'} - \frac{\gamma^2}{100\alpha''}\left(\frac{25B'}{\gamma}\right)\\
&= \frac{B}{\alpha'} - \frac{B'\gamma}{4\alpha''} = \frac{B}{\alpha'} - \frac{B\gamma}{4\alpha'}\\
&= B\left(\frac{4-\gamma}{4\alpha'}\right),
\end{aligned}$$

so it remains to show that $\frac{4-\gamma}{4\alpha'} \geq 1/\alpha$. Recall that $\gamma \leq 1$, so we have

$$\begin{aligned}
\gamma &\geq \gamma^2\\
\gamma - \gamma^2 + 4 + 2\gamma &\geq 4 + 2\gamma\\
4 + 3\gamma - \gamma^2 &\geq 4 + 2\gamma\\
(4-\gamma)(1+\gamma) &\geq 4 + 2\gamma\\
\frac{4-\gamma}{4\alpha'} = \frac{4-\gamma}{4+2\gamma} &\geq \frac{1}{1+\gamma} = \frac{1}{\alpha},
\end{aligned}$$

as desired. ∎

Theorem A.2 ((B,α)-partitioning). *For any $\alpha = 1+\Theta(1)$, history-independent (B,α)-partitioning of a set S of size $N \geq B$ can be maintained with: group-update cost $O(1/B)$ per insertion/deletion in expectation and $O(\log N/\log\log N)$ with high probability in N; and element-movement cost $O(1)$ amortized per insertion/deletion in expectation, and $O(B\log N/\text{loglog}\, N)$ with high probability in N.*

Proof. For the group-update cost, notice that, for constant α, there are $\Theta(1)$ groups per supergroup. Thus, we can afford to rebuild the supergroup whenever a group changes, for a group-update cost of $\Theta(1)$. We need to prove that this happens with probability $O(1/B)$.

There are three events that can cause a rebuild of a supergroup. Suppose that an insertion or deletion affects group i, and for convenience, we will drop the subscript i on all variables.

(1) New pivots of minigroups could be introduced or removed.
(2) The number of groups $k = \lceil (M+r)/B' \rceil$ could change, meaning that τ changes.
(3) On an insert, the size of a minigroup could increase by 1, which means that a particular minigroup, which used to be assigned to one group, now gets assigned to the next group. Or the reverse, where on a delete, the size of a minigroup decreases, which means that a new minigroup gets assigned to one group earlier.

Case (1): On any insert or delete, the probability of a pivot change is $O(1/B)$ by Lemma 3.3.

Cases (2) and (3) are the reason we introduced random blurring, because some particular insertion could cause some threshold to be crossed. If the adversary knew what this threshold was, they could insert and delete across the threshold over and over.

For Case (2), k increases by 1 when $M' + r$ is a multiple of B' and is increased by 1. But r is a random variable that is selected uniformly over $[B'] = [\Theta(B)]$. Thus, the probability that any one insertion is exactly at the point where B' divides $M' + r$ is $O(1/B)$.

For Case (3) instead of picking a threshold for a group size of exactly τ, we change the threshold to be $\tau + s^j$, for a randomly chosen $s^j \in [dB']$ in each group G^j. So now, since there is a random threshold in a group, the probably that any insert triggers a minigroup getting pushed out of/into a group is $O(1/dB') = O(1/B)$.

Finally, we can have changes in the pivots that divide the set into supergroups. The supergroups are formed by a $(B, 3)$-partition, so Theorem 3.15 gives $O(1/B)$ supergroup pivot changes in expectation. And again, we can afford to rebuild all $\Theta(1)$ pivots within the supergroups if a supergroup pivot changes for no extra asymptotic cost. Putting these all together gives the expected group-update cost of $O(1/B)$.

For the high probability bound, since the partition into supergroups is done using a $(B, 3)$-partitioning, Theorem 3.15 gives $O(\log N/\log\log N)$ supergroup boundary changes per insertion/deletion w.h.p., and each supergroup can only have $O(1)$ group boundary changes.

To establish bounds on the element movement cost, notice that each group boundary change causes up to B elements to change groups, so we have amortized element-movement costs of $O(1)$ in expectation and $O((B \log N)/\log\log N)$ with high probability. ∎

A.2 History-Independent Fusion Trees

Fusion trees are search trees that operate in the word-RAM model and support insert and delete update operations and predecessor and successor search operations in $O(\lceil \log N/\log w \rceil)$ time, where w is the word size. For example, if $w \in \Theta(\log N)$, then a fusion tree supports update and search operations in $O(\log N/\log\log N)$ time. It achieves these bounds by using constant-time bit-parallel word operations that are common in modern CPUs, such as bit shifting, bitwise AND, OR, and XOR operations, and the most-significant bit (MSB) operation (which is equivalent to casting an integer to a float and reading the exponent field). Previous implementations of fusion trees, as described in the literature, are not history independent, however; see, e.g., [3, 6, 28, 46, 51, 58, 63].

In this section, we describe how our results imply efficient constructions for history-independent fusion trees. The key observation is to note that a fusion tree is structurally a B-tree with branching factor $B = w^{1/c}$, for a small constant c, which is typically 2, 5, or 6; see, e.g. [3, 6, 28, 46, 51, 58, 63]. Thus, the height of a fusion tree is $O(\log N/\log w)$. Each node, v, in a fusion tree is implemented as a sorted packed array of at most B keys, just like in a B-tree, plus a memory word that stores compact (e.g., $w^{1/c}$-sized) sketches in a packed array for each of the keys stored at v. Moreover, this packed representation supports performing predecessor and successor operations with respect to the set of keys stored at v in $O(1)$ time using bit-parallel word operations [3, 6, 28, 46, 51, 58, 63]. Furthermore, the packed array of sketches stored at a node in a fusion tree is canonical, since it is based on choosing bit positions for the places that have two subtrees in a binary trie for the keys stored at the node; e.g., see [3, 6, 28, 46, 51, 58, 63]. Thus, we can maintain a fusion tree to be history independent by maintaining its corresponding B-tree to be history independent.

Searching in a fusion tree can be done in time proportional to its height, in the word-RAM model [3, 6, 28, 46, 51, 58, 63], by using bit-parallel word operations. This is achieved by performing the comparison and branching operations for each node in $O(1)$ time when searching for a predecessor or successor of search key, x, while traversing down the B-tree structure of the fusion tree. In other words, the search time in a fusion tree corresponds to the I/O complexity for searching in its corresponding B-tree.

In contrast, the cost for performing an insert or delete in a fusion tree instead corresponds to the

CPU cost for an insert or a delete in its corresponding B-tree, since performing a split or merge operation for sibling nodes in a fusion tree takes $\Theta(B)$ time [3, 6, 28, 46, 51, 58, 63]. The reason for this is that doing a split and merge operation for sibling nodes in a fusion tree still requires moving the (full-sized) keys between the nodes as well as the bit-parallel word operations necessary to create new packed arrays of the sketches for the keys for the merged or split nodes. Thus, given our results described above, which we have characterized in terms of CPU costs and I/O costs for history-independent B-trees, we have the following theorem.

Theorem A.3. *One can maintain a strongly history-independent fusion tree of size $O(N)$ in the word-RAM model, with word size w, that supports searching in $O(\lceil \log N / \log w \rceil)$ time, in the worst case, and insertion/deletion in $O(\lceil \log N / \log w \rceil)$ time in expectation.*

A.3 Deferred Proofs

A.3.1 Proofs for Section 2.

Proof of Lemma 2.4. The uniqueness of T_S follows immediately from the recursive construction. The root is unique. Thus, recursively, each of the children is unique.

To prove the size bounds, suppose that $|S| \geq B$, so that T_S is nonempty. Write the nodes of T_S as $t_1 \prec t_2 \prec \cdots \prec t_k$. For any consecutive pair t_i, t_{i+1}, we claim that $B/2 \leq |S[t_i, t_{i+1}]| \leq B$. Indeed, this interval cannot be larger, or there would be an additional pivot in between t_i and t_{i+1}. Conversely, there cannot be fewer than $B/2$ elements between them, or one would be in the protected region of the other (and thus not eligible for selection as a pivot).

Since there are between $B/2$ and B elements in S between successive elements in T_S, we must have that $|T_S|/|S| = \Theta(1/B)$. ∎

Proof of Lemma 2.6. Let w be the root of a subproblem $\bar{S} = S[u, v]$. We say that w is ***good*** if $\frac{|U(\bar{S})|}{4} \leq \text{rank}_{\bar{S}}(w) \leq \frac{3|U(\bar{S})|}{4}$. Since the root is equally likely to be any element in $U(\bar{S})$, the probability that w is good is at least $1/2$ (where the randomness is taken over the choice of hash function h).

Let x be a leaf in T_S, and consider the pivots encountered along the root-to-leaf path to x. After each good pivot, each child interval has size at most $3/4$ of the parent interval, and so the root-to-leaf path to x can have at most $\log_{4/3} N/B = O(\log(N/B))$ good pivots before reaching x. By a standard Chernoff bound, the root-to-leaf path to x has length $O(\log(N/B))$ with high probability in N. Taking a union bound over all $\Theta(N/B)$ leaves, the protected Cartesian tree has height $O(\log(N/B))$ with high probability in N. ∎

A.3.2 Proofs for Section 3.

Proof of Corollary 3.9. Let z be the root of T_S, and let $M = |U(S')|$. If $N < B + o(B)$, then the bound holds trivially, so we consider $N \geq B + \Theta(B)$ and claim that the probability that z is not the root of $T_{S'}$ is $O(1/M)$. Indeed, the only elements that can be selected as the root of $T_{S'}$ by RecomputeCartesian(S, x) are $\overline{x}$ and z. And $\overline{x}$ has the minimal hash in $U(S')$ with probability $1/|U(S')|$ by the randomness of hash values. Since $M = N - B = \Theta(N)$, this probability is $O(1/N)$. In this case, we charge all $\Theta(N/B)$ nodes in the tree to z, since at worst we have to touch all descendants of z in implementing the changed root. Thus the contribution of z to the expected number of operations is $O(1/B)$.

By Lemma 3.8, all changed nodes will be found it at most two root-to-leaf paths. Let z' be a node in one of these paths, where z' is the root of some subproblem $\overline{S}$ of size $N' \geq B + \Theta(B)$, and $M' = |U(\overline{S})|$. Once again, the probability that z' is no longer the root of $T_{\overline{S}}$ is $1/M'$, and in this event we charge z' with all N'/B nodes in its subtree. Since N' and M' differ by $\Theta(B)$, the

contribution of z' to the expected number of changes is $O(1/B)$. By Lemma 2.6, the root-to-leaf path has length $O(\log(N/B))$ with high probability in N, thus we have at most this many nodes contributing $O(1/B)$ to the expectation.

At the leaf level, subproblems may have a higher probability of changing their root because their unprotected region has size $o(B)$. However, since leaves have no children, we change only one node in this event. Thus the final subproblem in the root-to-leaf path contributes at most 1 to the expectation. Putting these together, the expected number of operations to update the tree is $O(\log(N/B)/B + 1)$. ■

Received December 2023; revised February 2024; accepted March 2024