

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

Michael A. Bender
Stony Brook University and RelationalAI

Michael T. Goodrich
University of California, Irvine

Martín Farach-Colton
New York University

Hanna Komlós
New York University

ABSTRACT

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 $O(B \log N / \log \log N)$ with high probability in set size N .

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 [43, 39]. 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., [33]), stopping files revealing damaging and embarrassing information that their creators

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is a minor revision of the paper entitled History-Independent Dynamic Partitioning: Operation-Order Privacy in Ordered Data Structures, published in Proc. ACM Manag. Data, Vol. 2, No. 2 (PODS), Article 108, May 2024. DOI: <https://doi.org/10.1145/3651609>.

thought had been erased [43, 34, 35, 30], and databases revealing that some of their data or metadata has been removed (e.g., due to a court order that must remain confidential) [41, 37, 61, 29].

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 [31] and B-skip-list [30, 32], 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 [43, 34, 35]. 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* [33] if the data structure can quickly prepare for an observation.¹ In each case, we assume that

¹For 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

an observer can see a memory representation before or after, but not during an update operation. History independence was introduced by Micciancio [39], who showed how to build a search tree with a history-independent structure. Naor and Teague [43] 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, 34] 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 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. [Theorem 1 in [34]] *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 [39, 43, 35, 34, 1, 23, 21, 42, 31, 32, 13], including fast history-independent constructions for cuckoo hash tables [42], linear-probing hash tables [21, 33], other hash tables [21, 43], trees [39, 1] memory allocators [43, 33, 38], write-once memories [40], priority queues [23], union-find data structures [43], external-memory dictionaries (e.g., for a database or key-value store) [30, 31, 32, 13], file systems [11, 10, 8, 51], cache-oblivious dictionaries [13], order-maintenance data structures [21], packed-memory arrays/list-labeling data structures [13, 15], and geometric data structures [59]. Given the strong connection between history independence and unique representability [34, 35], some data structures that predate the formalism can be made history independent, including ordered linear probing/Robinhood hashing [2, 24], skip lists [48], treaps [7], and other less well known deterministic data structures with canonical representations [56, 4, 5, 49, 55]. This foundational algorithmic work on history independence has found its way into databases [10, 46, 52] and other systems; there are now voting machines [20], file systems [11, 9, 8], 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 [26, 12], it is especially important in databases.

history-independent. However, the data structure could be audible, since the transition log could be flushed in preparation for an observation.

Dynamic partitioning is also used in write-optimized dictionaries such as B^ϵ -trees [22, 19] and log-structured merge trees [44, 53], fusion trees [57, 28, 62, 3, 6, 45, 50], 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 is 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* [36]. 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 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° -trees [18] or log-structured merge trees [44], 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 [57, 28, 62, 3, 6, 45, 50], partitioning takes place at the level of machine words. (Fusion trees [57, 28, 62, 3, 6, 45, 50] are word-RAM dictionaries that support $o(\log N)$ -time searches and updates.) In order-maintenance data structures [27, 14], 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 [58], 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 [32, 30]. To adapt a traditional skip list² for external memory, it is natural to modify the promotion probability from $1/2$ to $1/B$. Thus, w.h.p. 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 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 high variance: a group has size $\Theta(B)$ in expectation, but with high proba-

²A (traditional) skip list [47] 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 (w.h.p.) in N , where N is the number of elements in the skip list [54].

bility, 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 $B = \Omega(\text{polylog } N)$, then increase the promotion probability from $1/B$ 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 and have worse search performance.

Another approach investigated by Golovin [30, 31] 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 in the full version of this paper.

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 (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 $O(\log N / \log \log N)$ with high probability in N .*

THEOREM 2 (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 $O(B \log N / \log \log N)$ 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 in the full version of this paper.

We re-emphasize that the partitioning is deterministic. That is, there is no probability that the partitions 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 and 2, 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 and 2, 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 $O\left(\frac{\log N}{\log \log B}\right)$ w.h.p.; see Theorem 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. The proofs for all theorems and lemmas can be found the full version of this paper,³ which also contains results on fusion trees and (B, α) -partitioning for smaller values of α .

2. PROTECTING THE FLANKS

As a warm-up for our HI dynamic partitioning solution, let us first describe an algorithm for constructing a canonical

³See <https://dl.acm.org/doi/10.1145/3651609>.

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 ((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 3 (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 $\sigma = \sigma_1, \sigma_2, \dots$, 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 $\text{rank}_S(u) = |\{x \in S \mid x \preceq u\}|$. Denote $|S|$ by N . We emphasize that the restriction $S[u, v]$ does not include the endpoints u, v , and so $|S[u, v]| = \text{rank}_S(v) - \text{rank}_S(u) - 1$.

Protected regions of a set.

The **protected region** of S is the set $P(S) = \{x \in S \mid \text{rank}_S(x) \leq B/2\} \cup \{x \in S \mid \text{rank}_S(x) > N - B/2\}$. 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 $P_S(u) = \{x \in S \mid \text{rank}_S(u) - B/2 < \text{rank}(x) \leq \text{rank}_S(u) + B/2\}$, 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, $h : \mathcal{U} \rightarrow [0, 1]$, which maps elements to real numbers between 0 and 1, such that h ’s random bits are set prior to our construction.⁴

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.⁵

2.2 A Static Partitioning Algorithm

We define a partition on S by selecting a set of **pivots** $\{p_1, \dots, p_k\}$, where $p_1 \prec p_2 \prec \dots \prec p_k$. 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, \dots, G_k , where $G_i = \{y \in S \mid p_i \preceq y \prec p_{i+1}\}$.

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 [60].

⁴In practice, a pseudo-random hash function that maps elements to values having $3 \log n$ bits should be sufficient.

⁵However, 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.

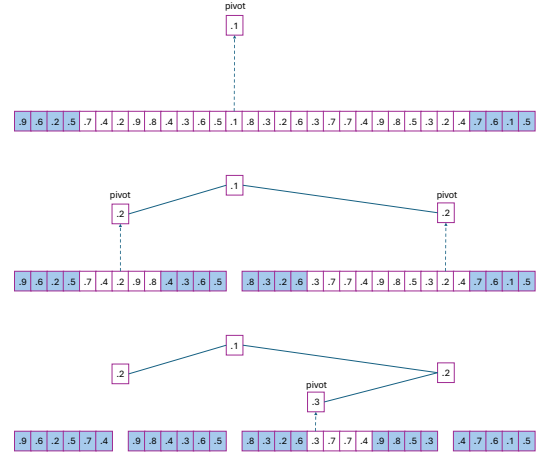


Figure 1: Recursive calls in the partition algorithm (1). (Top) Level 1; (Middle) level 2; (Bottom) level 3. Each box shows the first digit of $\text{value}(x)$ after the decimal point. Here, $B/2 = 4$ and protected regions are shaded light blue. The protected Cartesian tree is shown with solid edges.

DEFINITION 4. 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 T_S (or T when S is clear from context), to be the internal nodes of a binary search tree, \bar{T}_S , which is defined recursively as follows. If $|S| < B$ (i.e., all elements in S are protected), then \bar{T}_S is a single (leaf) node associated with S ; hence, in this case, T_S is empty. Otherwise, the root of \bar{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 $\bar{T}_{S[-\infty, u]}$, and the right child of u is the root of $\bar{T}_{S[u, \infty]}$. Thus, if either recursive Cartesian tree is a leaf in \bar{T}_S , there is no corresponding child node in T_S . We refer to \bar{T}_S as the **closure** of the protected Cartesian tree, T_S . See Figure 1.

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 $\text{value}(w)$, and left subtree $\bar{T}_{S[x, w]}$ and right subtree $\bar{T}_{S[w, y]}$.

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 **CompuCartesian(S)**.

Note that because T_S consists of the internal nodes of \bar{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 .

LEMMA 1. *For any set S with fixed hash function h , there is a unique protected Cartesian tree T_S with values $\text{value}(x) = h(x)$, 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 1. *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.

LEMMA 2. *The protected Cartesian tree T_S with random hash function h has height $O(\log(N/B))$ w.h.p. in N .*

COROLLARY 2. *Each search operation in a protected Cartesian tree with a random hash function has CPU cost $O(\log(N/B))$ w.h.p. in N .*

LEMMA 3. *A protected Cartesian tree with arbitrary values on a set S of size $|S| = N$ can be built in $O(N)$ CPU 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

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 $\text{rank}_S^z(y) = (\text{rank}_S(y) - \text{rank}_S(z)) \bmod |S|$. 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 4. *Let $S \subseteq \mathcal{U}$ with $|S| = N \geq B$, and T_S , the protected Cartesian tree on S with random hash function h . Every element in S has probability of $\Theta(1/B)$ of being 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.*

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 $\text{RECOMPUTESUBPROBLEM}(S, x)$ 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, $\text{COMPUTECARTESIAN}(R'[-\infty, x])$ as the left child of x , and $\text{COMPUTECARTESIAN}(R'[x, \infty])$ 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 \tilde{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, \tilde{T} as z 's left child, and $\text{RECOMPUTESUBPROBLEM}(R[z, \infty], x)$ as z 's right child. If $x \prec z$, return the tree with z at the root, \tilde{T} as z 's right child, and $\text{RECOMPUTESUBPROBLEM}(R[-\infty, z], x)$ 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, $\text{COMPUTECARTESIAN}(R'[-\infty, y])$ as the left child of y , and $\text{COMPUTECARTESIAN}(R'[y, \infty])$ 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 \tilde{T} of the recursive subproblem not containing y is unchanged. If $z \prec y$, return the tree with z at the root, \tilde{T} as z 's left child, and $\text{RECOMPUTESUBPROBLEM}(R[z, \infty], y)$ as z 's right child. If $y \prec z$, return the tree with z at the root, \tilde{T} as z 's right child, and $\text{RECOMPUTESUBPROBLEM}(R[-\infty, z], y)$ 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 $u(x)$. 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 lemma.

LEMMA 5. $T_S \neq T_{S'}$ 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 6. If $|S| \geq B$, the probability that $T_S \neq T_{S'}$ is $O(1/B)$.

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 \bar{x} to denote $u(x)$ if it exists and is a new pivot, and x otherwise. We reiterate that the value of \bar{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 \bar{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 14 that the case where the root changes is equivalent with slight modifications.

LEMMA 7. 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 $T_{S'[c, d]} = T_{S'[c, d]}$.

LEMMA 8. Every nontrivial subproblem in S' to the left of \bar{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.

LEMMA 9. Every nontrivial subproblem in S' to the right of \bar{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. Let $b \in T_{S'} \setminus T_S$ be a new pivot such that $b \neq \bar{x}$. Then the child subtree of b closer to \bar{x} is empty.

LEMMA 10. Suppose $T_S \neq T_{S'}$. Then there is a path from \bar{x} to the leaf level containing all changed pivots to the left of \bar{x} , and a path from \bar{x} to the leaf level containing all changed pivots to the right of \bar{x} .

This implies a bound on the total cost to maintain protected Cartesian trees.

COROLLARY 4. *The protected Cartesian tree T_S on S can be maintained with expected CPU cost $O(1 + \log(N/B)/B)$.*

Recall that we analyze protected Cartesian trees in order to bound the costs of our dynamic partitioning algorithm. When analyzing the number of pivots (equivalently, the number of group boundaries) that change after an insertion, we can in fact show 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 11. *Let $T_{S'} \setminus T_S = \{y_1, \dots, y_k\}$ be the set of new pivots, and let $z \in T_{S'} \cap T_S$. Then $z \prec y_1$ or $y_k \prec z$.*

LEMMA 12. *Let $T_{S'} \setminus T_S = \{y_1, \dots, y_k\}$ be the set of new pivots. Then their hash values are monotonically decreasing until \bar{x} , and monotonically increasing after \bar{x} . In other words, $h(y_1) > h(y_2) > \dots > h(y_{i-1}) > h(\bar{x}) < h(y_{i+1}) < \dots < h(y_k)$ for some i .*

LEMMA 13. *If $T_{S'} \neq T_S$, the probability of at least k new pivots in $T_{S'}$ is at most $\frac{1}{(\lfloor k/2 \rfloor - 1)!}$.*

LEMMA 14. *Let $S \subseteq U$ with $|S| \geq B$, 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 an insertion into S is $O(1/B)$ in expectation and $O(\log N / \log \log N)$ w.h.p. in N .*

COROLLARY 5. *Let S, T_S as in Lemma 14. Then the number of pivots that change in T_S after a deletion is $O(1/B)$ in expectation and $O(\log N / \log \log N)$ w.h.p. in N .*

3.3 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 (*$(B, 3)$ -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 $O(\log N / \log \log N)$ w.h.p. in N ; and element-movement cost $O(1)$ amortized per insertion/deletion in expectation, and $O((B \log N) / \log \log N)$ w.h.p. in N .*

To obtain a linear $(B, 2)$ -partition, run the (B, α) -partitioning scheme with $\alpha = 2$. We state the main (B, α) -partitioning theorem here and refer the reader to Appendix A.1 of the full version of this paper for the details of the scheme and its proofs. Theorem 4 then immediately implies Theorems 1 and Theorem 2.

THEOREM 4 (*(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 $O(\log N / \log \log N)$ w.h.p. in N ; and element-movement cost $O(1)$ amortized per insertion/deletion in expectation, and $O(B \log N / \log \log N)$ w.h.p. in N .*

4. HISTORY INDEPENDENT B-TREES

In this section, we construct a dynamic history-independent B -tree using dynamic partitioning. Let $S \subseteq U$. 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 $S_1 = \{p_1, \dots, p_k\}$ 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 $S_2 = \{q_1, \dots, q_j\}$ 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 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, 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 partitioning scheme to build $\mathcal{B}(S)$ immediately implies the following lemmas.

LEMMA 15. *$\mathcal{B}(S)$ is strongly history independent.*

LEMMA 16. *Each node in $\mathcal{B}(S)$ contains between $B/2$ and B elements.*

COROLLARY 6. *$\mathcal{B}(S)$ has height $O(\log_B(N))$ deterministically.*

COROLLARY 7. *Each search operation in $\mathcal{B}(S)$ has worst case CPU cost $O(\log(N))$ and worst case I/O cost $O(\log_B(N))$.*

The following theorem is a consequence of Theorem 1 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 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 $O\left(\frac{\log N}{\log \log B}\right)$ w.h.p. in N .*

The following results from Theorem 5, and the fact that we perform $O(B)$ CPU operations within each block when updating group boundaries.

THEOREM 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 $O\left(\frac{B \log N}{\log \log B}\right)$ w.h.p. in N .*

5. ACKNOWLEDGEMENTS

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.

6. REFERENCES

- [1] U. A. Acar, G. E. Blelloch, R. Harper, J. L. Vitter, and S. L. M. Woo. Dynamizing static algorithms, with applications to dynamic trees and history independence. In *Proc. of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 531–540, 2004.
- [2] O. Amble and D. E. Knuth. Ordered hash tables. *The Computer Journal*, 17(2):135–142, 1974.
- [3] A. Andersson, P. B. Miltersen, and M. Thorup. Fusion trees can be implemented with AC0 instructions only. *Theoretical Computer Science*, 215(1-2):337–344, 1999.
- [4] A. Andersson and T. Ottmann. Faster uniquely represented dictionaries. In *Proc. of the 32nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 642–649, 1991.
- [5] A. Andersson and T. Ottmann. New tight bounds on uniquely represented dictionaries. *SIAM Journal on Computing*, 24(5):1091–1103, 1995.
- [6] A. Andersson and M. Thorup. Dynamic ordered sets with exponential search trees. *Journal of the ACM*, 54(3):1–40, 2007.
- [7] C. R. Aragon and R. G. Seidel. Randomized search trees. In *Proc. of the 30th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 540–545, 1989.
- [8] S. Bajaj, A. Chakraborti, and R. Sion. The foundations of history independence. *arXiv preprint arXiv:1501.06508*, 2015.
- [9] S. Bajaj, A. Chakraborti, and R. Sion. Practical foundations of history independence. *IEEE Trans. Inf. Forensics Secur.*, 11(2):303–312, 2016.
- [10] S. Bajaj and R. Sion. Ficklebase: Looking into the future to erase the past. In *Proc. of the 29th IEEE International Conference on Data Engineering (ICDE)*, pages 86–97, 2013.
- [11] S. Bajaj and R. Sion. HIFS: History independence for file systems. In *Proc. of the ACM SIGSAC Conference on Computer & Communications Security (CCS)*, pages 1285–1296, 2013.
- [12] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, Feb. 1972.
- [13] M. A. Bender, J. W. Berry, R. Johnson, T. M. Kroeger, S. McCauley, C. A. Phillips, B. Simon, S. Singh, and D. Zage. 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)*, pages 289–302, 2016.
- [14] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Proc. 10th European Symposium on Algorithms (ESA)*, pages 152–164. Springer, 2002.
- [15] M. A. Bender, A. Conway, M. Farach-Colton, H. Komlós, W. Kuszmaul, and N. Wein. Online list labeling: Breaking the $\log^2 n$ barrier. In *Proc. 63rd IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 980–990, November 2022.
- [16] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. of the 41st Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 399–409, 2000.
- [17] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. Latin American Theoretical Informatics (LATIN)*, pages 88–94, 2000.
- [18] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-oblivious streaming b-trees. In *SPAA*, pages 81–92. ACM, 2007.
- [19] M. A. Bender, M. Farach-Colton, W. Jannen, R. Johnson, B. C. Kuszmaul, D. E. Porter, J. Yuan, and Y. Zhan. An introduction to b^e -trees and write-optimization. *login; magazine*, 40(5):22–28, Oct. 2015.
- [20] J. Bethencourt, D. Boneh, and B. Waters. Cryptographic methods for storing ballots on a voting machine. In *Proc. of the 14th Network and Distributed System Security Symposium (NDSS)*, 2007.
- [21] G. E. Blelloch and D. Golovin. Strongly history-independent hashing with applications. In *Proc. 48th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 272–282, 2007.
- [22] G. S. Brodal and R. Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '03)*, pages 546–554, Baltimore, MD, 2003.
- [23] N. Buchbinder and E. Petrank. Lower and upper bounds on obtaining history independence. In *Advances in Cryptology*, pages 445–462, 2003.
- [24] P. Celis, P. Larson, and J. I. Munro. Robin Hood hashing (preliminary report). In *26th Annual Symposium on Foundations of Computer Science (FOCS'85)*, pages 281–288, Portland, Oregon, USA, 21–23 Oct. 1985.
- [25] B. Chen and R. Sion. Hiflash: A history independent flash device. *CoRR*, abs/1511.05180, 2015.
- [26] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.
- [27] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proc. of the 19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 365–372, 1987.
- [28] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. System Sci.*, 47:424–436, 1993.
- [29] S. Garg, S. Goldwasser, and P. N. Vasudevan. Formalizing data deletion in the context of the right to be forgotten. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 373–402. Springer, 2020.
- [30] D. Golovin. *Uniquely Represented Data Structures with Applications to Privacy*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 2008, 2008.
- [31] D. Golovin. B-treaps: A uniquely represented alternative to B-trees. In *Proc. 36th Annual International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 487–499. 2009.
- [32] D. Golovin. The B-skip-list: A simpler uniquely represented alternative to B-trees. *arXiv preprint arXiv:1005.0662*, 2010.
- [33] M. T. Goodrich, E. M. Kornaropoulos,

- M. Mitzenmacher, and R. Tamassia. Auditable data structures. In *Proc. IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 285–300, April 2017.
- [34] J. D. Hartline, E. S. Hong, A. E. Mohr, W. R. Pentney, and E. Roche. Characterizing history independent data structures. In *Proceedings of the Algorithms and Computation, 13th International Symposium (ISAAC)*, pages 229–240, November 2002.
- [35] J. D. Hartline, E. S. Hong, A. E. Mohr, W. R. Pentney, and E. C. Roche. Characterizing history independent data structures. *Algorithmica*, 42(1):57–74, 2005.
- [36] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.
- [37] B.-J. Koops. Forgetting footprints, shunning shadows: A critical analysis of the right to be forgotten in big data practice. *SCRIPTed*, 8:229, 2011.
- [38] W. Kuzmaul. Strongly history independent storage allocation: New upper and lower bounds. In *Proc. 64th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, November 2023.
- [39] D. Micciancio. Oblivious data structures: applications to cryptography. In *Proc. 29th Annual ACM Symposium on Theory of Computing (STOC)*, pages 456–464, 1997.
- [40] T. Moran, M. Naor, and G. Segev. Deterministic history-independent strategies for storing information on write-once memories. In *Proc. 34th International Colloquium on Automata, Languages and Programming (ICALP)*, 2007.
- [41] K. Murata and Y. Orito. The right to forget/be forgotten. *Ethics in Interdisciplinary and Intercultural Relations*, 192, 2011.
- [42] M. Naor, G. Segev, and U. Wieder. History-independent cuckoo hashing. In *Proc. of the 35th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 631–642. Springer, 2008.
- [43] M. Naor and V. Teague. Anti-persistence: history independent data structures. In *Proc. 33rd Annual ACM Symposium on Theory of Computing (STOC)*, pages 492–501, 2001.
- [44] P. O’Neil, E. Cheng, D. Gawlic, and E. O’Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.
- [45] M. Patrascu and M. Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *55th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 166–175, 2014.
- [46] R. Poddar, T. Boelter, and R. A. Popa. Arx: A strongly encrypted database system. *IACR Cryptol. ePrint Arch.*, page 591, 2016.
- [47] W. Pugh. *Incremental computation and the incremental evaluation of functional programs*. PhD thesis, Cornell University, 1988.
- [48] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.
- [49] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Proc. of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 315–328, 1989.
- [50] R. Raman. Priority queues: Small, monotone and trans-dichotomous. In *4th European Symposium on Algorithms (ESA)*, pages 121–137, 1996.
- [51] D. S. Roche, A. J. Aviv, and S. G. Choi. Oblivious secure deletion with bounded history independence. *arXiv preprint arXiv:1505.07391*, 2015.
- [52] D. S. Roche, A. J. Aviv, and S. G. Choi. A practical oblivious map data structure with secure deletion and history independence. In *IEEE Symposium on Security and Privacy (SP)*, pages 178–197. IEEE Computer Society, 2016.
- [53] R. Sears, M. Callaghan, and E. Brewer. Rose: Compressed, log-structured replication. *Proceedings of the VLDB Endowment*, 1(1):526–537, 2008.
- [54] S. Sen. Some observations on skip-lists. *Information Processing Letters*, 39(4):173–176, 1991.
- [55] L. Snyder. On uniquely represented data structures. In *Proc. of the 18th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 142–146, 1977.
- [56] R. Sundar and R. E. Tarjan. 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)*, pages 18–25, 1990.
- [57] M. Thorup. On AC0 implementations of fusion trees and atomic heaps. In *14th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 699–707, 2003.
- [58] A. K. Tsakalidis. Maintaining order in a generalized linked list. *Acta Informatica*, 21(1):101–112, May 1984.
- [59] T. Tzouramanis. History-independence: a fresh look at the case of R-trees. In *Proc. 27th Annual ACM Symposium on Applied Computing (SAC)*, pages 7–12, 2012.
- [60] J. Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4):229–239, 1980.
- [61] R. K. Walker. The right to be forgotten. *Hastings LJ*, 64:257, 2012.
- [62] D. E. Willard. Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM J. Comput.*, 29:1030–1049, December 1999.