Anti-Persistence on Persistent Storage: History-Independent Sparse Tables and Dictionaries

Michael Bender, Jonathan Berry, Rob Johnson, Thomas Kroeger, Samuel Mccauley, Cynthia Phillips, Bertrand Simon, Shikha Singh, David Zage

Anti-Persistence on Persistent Storage: History-Independent Sparse Tables and Dictionaries

Michael A. Bender[*]
Stony Brook University

Jonathan W. Berry[†]
Sandia National Laboratories

Rob Johnson[*]
Stony Brook University

Thomas M. Kroeger[‡]
Sandia National Laboratories

Samuel McCauley[*]
Stony Brook University

Cynthia A. Phillips[†]
Sandia National Laboratories

Bertrand Simon[§]
Ecole Normale Supérieure de Lyon

Shikha Singh[*]
Stony Brook University

David Zage[¶]
Intel Corporation

ABSTRACT

We present history-independent alternatives to a B-tree, the primary indexing data structure used in databases. A data structure is history independent (HI) if it is impossible to deduce any information by examining the bit representation of the data structure that is not already available through the API.

We show how to build a history-independent cache-oblivious B-tree and a history-independent external-memory skip list. One of the main contributions is a data structure we build on the way—a history-independent packed-memory array (PMA). The PMA supports efficient range queries, one of the most important operations for answering database queries.

Our HI PMA matches the asymptotic bounds of prior non-HI packed-memory arrays and sparse tables. Specifically, a PMA maintains a dynamic set of elements in sorted order in a linear-sized array. Inserts and deletes take an amortized $O(\log^2 N)$ element moves with high probability. Simple experiments with our implementation of HI PMAs corroborate our theoretical analysis. Comparisons to regular PMAs give preliminary indications that the practical cost of adding history-independence is not too large.

Our HI cache-oblivious B-tree bounds match those of prior non-HI cache-oblivious B-trees. Searches take $O(\log_B N)$ I/Os; inserts and deletes take $O(\frac{\log^2 N}{B} + \log_B N)$ amortized I/Os with high probability; and range queries returning k elements take $O(\log_B N + k/B)$ I/Os.

Our HI external-memory skip list achieves optimal bounds with high probability, analogous to in-memory skip lists: $O(\log_B N)$ I/Os for point queries and amortized $O(\log_B N)$ I/Os for inserts/deletes. Range queries returning k elements run in $O(\log_B N + k/B)$ I/Os. In contrast, the best possible high-probability bounds for inserting into the folklore B-skip list, which promotes elements with probability $1/B$, is just $\Theta(\log N)$ I/Os. This is no better than the bounds one gets from running an in-memory skip list in external memory.

1. INTRODUCTION

A data structure is ***history independent*** (HI) if its internal representation reveals nothing about the sequence of operations that led to its current state [43, 47]. In this paper, we study history independence for persistent, disk-resident dictionary data structures.

We give two efficient history-independent alternatives to the B-tree, the primary indexing data structure used in databases. Specifically, we give an HI external-memory skip list and an HI cache-oblivious[1] B-tree.

One of the main contributions of the paper is a data structure we build on the way: a history-independent packed-memory array (PMA). As we explain, the PMA [14, 18] is an unlikely candidate data structure to be made history independent, since traditional PMAs rely on history so fundamentally. The HI PMA is one of the primary building blocks in the HI cache-oblivious B-tree. However, it can also be bolted onto any dictionary data structure, using the PMA to hold the actual elements and deliver fast range queries.

Notions of History Independence

Informally, history independence partially protects a disk-resident data structure when the disk is stolen by—or given to—a third party (the "observer"). This observer can access the data structure through the normal API but can also see

[*]Department of Computer Science, Stony Brook University, Stony Brook, NY 11794-2424 USA. Email: {bender, rob, smccauley,shiksingh}@cs.stonybrook.edu.

[†]MS 1326, PO Box 5800, Albuquerque, NM 87185 USA. Email: {jberry,caphill}@sandia.gov.

[‡]PO Box 969, MS 9011, Livermore, CA 94551 USA. Email: tmkroeg@sandia.gov.

[§]LIP, ENS de Lyon, 46 allee d'Italie, 69364 Lyon, France. Email: bertrand.simon@ens-lyon.fr.

[¶]Intel Corporation, 2200 Mission College Blvd, Santa Clara, CA 95054 USA. Email: zage@cerias.net.

PODS'16, June 26-July 01, 2016, San Francisco, CA, USA

ACM ISBN 978-1-4503-4191-2/16/06.

DOI: http://dx.doi.org/10.1145/2902251.2902276

[1]A ***cache-oblivious*** [29, 30, 51] algorithm or data structure is memory-hierarchy universal, in that it has no memory-hierarchy-specific parameterization (see Section 1.1).

its bit and pointer representation on disk. An HI data structure's bit representation never leaks any information to the observer that he could not learn through the API.

There are two notions of history independence, ***weak history independence*** (WHI) and ***strong history independence*** (SHI) [47]. The notions are distinguished by how many times the observer can look at the data structure—that is by how many times an interloper can steal (or otherwise gain access to) the disk on which the data structure resides. A weakly history-independent data structure is history independent against an observer who can see the memory representation once. A strongly history-independent data structure is history independent against an observer who can see the memory representation multiple times.

In this paper, we focus on weak history independence. From a performance perspective, weak HI is a stronger notion because it allows provably stronger performance guarantees (see Section 2). Protecting against multiple observations reduces achievable performance.

Weak history independence is the appropriate notion of history independence in situations where only one observation is possible. For example, when the data is on a device that can be separated from the owner (say a portable or embedded device), the owner no longer interacts with the device. WHI provides the same level of protection in this case with significantly better performance compared to SHI.

History Independence and Data

History independence in a database can have major advantages, depending on the kind of data that is being stored and the security requirements.

HI data structures naturally support information-theoretically-secure delete. In contrast, with more standard secure delete (where the file system overwrites deleted data with zeros), information about deleted data can leak from the memory representation. For example, it reveals how much data was deleted and where in the keyspace it might have been. In fact, a long history of failed redactions is one of the original motivations for history independence [36]. History independence guarantees that the memory representation will leak no information about these previous (now secure) deletes. Encryption is not a panacea to protect history unconditionally, since determined attackers can recover the key used to encrypt on-disk data [35].

In a database, the source of the data can be more sensitive than the data itself. As a toy example, consider a database of known organized crime members maintained by the police. The police might want to share such a database with select individuals without revealing the order and times in which the data was added to the database. Revealing this order might leak information about how and when the data was collected, which could reveal sources that the police want to stay hidden. Journalists may desire the same property to ensure their sources' anonymity.

In such cases, we need to be particularly careful about avoiding information leaks by side channels. Since leakage can be subtle and hard to quantify, it is beneficial to have a guarantee that nothing extraneous is revealed, which is exactly what history independence guarantees. HI data structures naturally hide the order in which data was inserted.

History Independence in Persistent Storage

This paper focuses on history independence for external-memory dictionaries. This area of history independence was initiated by Golovin [32, 33]; see Section 1.4 for details.

The ubiquitous (non-history-independent) external-memory dictionary is the B-tree.

Our objective is to build history-independent, I/O efficient external-memory dictionaries.We support standard dictionary operations: insertions, deletions, searches, and range queries. Our two external-storage computational models (the external-memory model [3] and the cache-oblivious model [29, 30, 51]) apply to both rotating disks and SSDs.

We give weakly history-independent data structures for persistent storage; these are easy to implement and retain strong performance guarantees. We show that even range-query data structures that seem to be inherently history *dependent*—in particular, packed-memory array or sparse tables [14, 16–18, 38, 41, 66]—can be made weakly history independent. Overall, our results demonstrate that it is possible to build efficient external-memory (and even cache-oblivious) history-independent data structures for indexing.

History Independence in Persistent Storage vs. RAM

History independence has been vigorously explored in the context of data structures that reside in RAM [20, 21, 23, 36, 43, 46, 47, 61] (see Section 1.4), but significantly less so in external memory. Although there is some theoretical work on history-independent on-disk data structures [31–33] and experimental work on history-independent file systems [10–12, 56], the area is substantially less explored.

This lacuna may seem surprising, since many of the classical arguments in support of history independence especially apply to disks. Hard drives are more vulnerable than RAM because they are persistent and easier to steal, making it easier for an attacker to observe on-disk data.

1.1 I/O and Cache-Oblivious Models

We prove our results using the classic models for analyzing on-disk algorithms and data structures: the disk-access machine (DAM) model of Aggarwal and Vitter [3] and the cache-oblivious model of Frigo et al. [29, 30, 51]. The DAM has an internal memory (RAM) of size M and an arbitrarily large external memory (disk). Data is transferred between RAM and disk in blocks of size $B < M$. The performance measure is transfers (I/Os). Computation is free.

The cache-oblivious model extends the DAM model. Now parameters B and M are unknown to the algorithm designer or coder. They can only be used as parameters in analyses.

Thus, an optimal cache-oblivious data structure is not parameterized by any block, cache or RAM size, or memory- or disk-access times. Remarkably, many problems have optimal (and practical) cache-oblivious solutions, including B-trees [14, 15, 22]. Informally, a cache-oblivious B-tree has approximately optimal memory or I/O performance at every level of an unknown multilevel memory hierarchy.

1.2 Packed-Memory Arrays and External-Memory Dictionaries

In this paper, we give a history-independent PMA and two external-memory dictionaries: a history-independent external skip list and a history-independent cache-oblivious B-tree. This subsection puts our results in context.

We first define a PMA. Then we describe the technical issues involved in making a history-independent PMA. We (this paper's authors) were surprised when we first suspected that a history-independent PMA could indeed exist. Here, we try to articulate why the PMA might seem to be an unlikely candidate data structure to make history independent, but why it is nonetheless possible.

We next explain why a history independent PMA leads, almost directly, to an HI cache-oblivious B-tree, the first nontrivial history-independent cache-oblivious data structure.

Finally, we discuss the history-independent external-memory skip list. This HI data structure still offers high-probability guarantees, analogous to in-memory skip lists. See Section 2 for the definition of with high probability, which we also write as "whp".

Although the idea of a B-skip list, which promotes elements with probability $1/B$ rather than probability $1/2$ is folklore and has appeared in the literature repeatedly [1, 25, 26, 33], we prove that its high-probability I/O guarantees are asymptotically no better than those of an in-memory skip list run in external memory.

Packed-Memory Arrays

One of the classic data-structural problems is called ***sequential file maintenance***: maintain a dynamic set of elements in sorted order in a linear-sized array. If there are N elements, then the array has $\Theta(N)$ empty array positions or *gaps* interspersed among the elements to accommodate future insertions. The gaps allow some elements to shift left or right to open slots for new elements—like shifting books on a bookshelf to make room for new books.

Remarkably, there are data structures for these problems that are efficient even for *adversarial* inserts and deletes. Indeed, the number of element moves per update is only $O(\log^2 N)$ both amortized [38, 63] and in the worst case [64–66], which is optimal [24].

In external memory, this data structure is called a ***packed-memory array*** [14,18]. It supports inserts, deletes, and range queries. Given the location where we want to insert or delete (which can be found using a separate indexing structure, e.g., [15, 22]), it takes only $O(1 + (\log^2 N)/B)$ amortized I/Os to shift the elements. Given the starting point, a range query returning k elements costs $\Theta(1 + k/B)$ I/Os.[2]

Prior PMAs operate as follows. To insert a new element after an existing element or to delete an element, find an enclosing subarray or ***range***, and ***rebalance***. This spreads out the elements (and gaps) within that range. The rebalance range is chosen based upon the density within the range, where ranges have minimum and maximum allowed densities. These thresholds depend upon the size of the range: small ranges have high thresholds for the maximum density and low thresholds for the minimum density. The larger a range is, the less variability is allowed in its density. The algorithmic subtlety has to do with choosing the right rebalance ranges and the right minimum and maximum density thresholds for each range size.

However, range densities are very history *dependent*. For example, if you repeatedly insert towards the front of an array or if you repeatedly delete from the back of the array, then the front of the array will be denser than the back. How could we possibly make a version of this data structure, that is history independent—that is, where newly inserted (or deleted) elements do not seem to increase (or decrease) some local density? We answer this question in this paper.

To use a more evocative image, picture a long trough where you are pouring sand in one location (corresponding to inserts) and letting out sand in another location (corresponding to deletes). As the sand piles up, the pile gradually flattens (corresponding to local rebalances). Although rebalances may flatten out the pile, we may still expect a bump for newly arrived sand, and a depression for recently departed sand. Perhaps surprisingly, we can avoid bumps and depressions with mostly local rebalances.

[2]This scanning bound is a further requirement on how the elements are distributed. Beyond the $O(N)$ overall space limitation, only $O(1)$ gaps can separate two consecutive elements.

Cache-Oblivious B-Trees

A B-tree is a dictionary supporting search, insert, delete, and range query operations. There are cache-oblivious versions [13, 15, 22, 40, 55].

A history-independent PMA can immediately yield a history-independent cache-oblivious B-tree. The idea is to take a PMA and "glue" it to a static cache-oblivious B-tree [51]. For details, see [15, 22]. We use a similar method to construct our history-independent cache-oblivious B-tree in Section 3.

External-Memory Skip Lists

The skip list is an elegant search-tree alternative introduced by Pugh [53]. Skip lists are randomized data structures having a weakly history independent pointer structure [31, 53].

Skip lists with N elements support searches, inserts, and deletes in $O(\log N)$ operations whp and range queries returning k elements in $O(\log N + k)$ operations whp [27, 42, 50]. They are heavily used in internal-memory algorithms [5, 8, 9, 28, 34, 37, 39, 49, 57].

This paper gives a simple and provably good external-memory history-independent skip list, which has high probability guarantees. Specifically, our skip list supports insert, deletes, and searches with $O(\log_B N)$ I/Os whp, and range queries returning k elements with $O(\log_B N + k/B)$ whp—which is just the search plus scan cost. Thus, our history independent, external memory skip list has high-probability bounds matching those of a B-tree.

Our challenge is to tweak the folklore B-skip list [1,25,26, 33] as little as possible (and in a history-independent way) so that we can achieve high-probability bounds for searches and inserts while maintaining optimal range queries.

1.3 Results

We begin by giving an efficient, history-independent packed memory array.

THEOREM 1. *There exists a weakly history-independent packed-memory array on N elements which can perform inserts and deletes in $O(\log^2 N)$ amortized element moves with high probability. This PMA requires $O(N)$ space, can perform inserts and deletes in amortized $O\big(\frac{\log^2 N}{B} + \log_B N\big)$ I/Os with high probability, and can perform a range query for k elements in $O(1 + k/B)$ I/Os.*

When $B = \Omega(\log N \log\log N)$ (reasonable on today's systems), then $\frac{\log^2 N}{B} = O(\log_B N)$, so inserts and deletes in our PMA have the same I/O complexity as in a B-tree.

Theorem 1 directly yields a history-independent cache-oblivious B-tree with these performance bounds:

THEOREM 2. *There exists a weakly history-independent, cache-oblivious B-tree on N elements which can perform inserts and deletes in $O\big(\frac{\log^2 N}{B} + \log_B N\big)$ amortized I/Os with high probability. This cache-oblivious B-tree requires $O(N)$ space and can answer a range query for k elements in $O(\log_B N + k/B)$ I/Os, i.e., the search plus the scan cost.*

We give a simple and provably good external-memory history-independent skip list, which has high probability guarantees analogous to in-memory skip lists.

THEOREM 3. *There exists a weakly history-independent, external-memory skip list on N elements which can perform look-ups in $O(\log_B N)$ I/Os with high probability. For a parameter $\varepsilon > 0$, the skip-list requires $O(\log_B N)$ amortized I/Os for inserts and deletes, with a worst case of $O(B^\varepsilon \log N)$ I/Os, all with high probability. This skip-list requires $O(N)$ space and can answer a range query for k elements in $O(\frac{1}{\varepsilon}\log_B N + k/B)$ I/Os with high probability.*

The parameter ε indicates a small trade-off between the worst-case rebuild cost after an update and the cost of medium-size range queries (see Section 6).

We contrast our data structure with the B-skip list, which promotes elements from one level to the next with probability $1/B$. We prove that with high probability, there exist at least $\Omega(\sqrt{NB})$ elements where the cost to search for any one of them is $O(\log \frac{N}{B})$. Thus, the high-probability I/O bounds for searching in a B-skip list are not asymptotically better than for searching in a regular (internal-memory) skip list that is implemented in external memory.

1.4 Related Work

History of History Independence

The history of history independence spans nearly four decades. The central notions of history independence predate its formalism.

One of the key ideas of history independence is unique representation, which was studied as far back as 1977 by Snyder [59]. Many uniquely represented data structures were published before the conception of history independence [4, 6, 7, 52–54, 59, 60]. Similar to unique representation, the idea of randomized structures that are *uniformly represented*, that is, have representations drawn from a distribution irrespective of the past history, emerged with Pugh's skip list [52, 53] and Aragon and Seidel's treap [7].

Nearly a decade later, Micciancio [43] defined ***oblivious data structures*** as those whose topology does not reveal the sequence of operations that led to the current state. In 2001, Naor and Teague [47] strengthened obliviousness to ***history independence*** ("anti-persistence"), generalizing it to include the entire *bit representation* of the structure, including memory addresses.

Hartline et al. [36] proved that a reversible data structure (i.e. one whose state graph is strongly connected) is SHI if and only if it fixes a ***canonical representation*** for each state depending only on initial (possibly random) choices made before any operations are performed.

Buchbinder and Petrank [23] further explored strong versus weak history independence, proving a separation between the two notions for heaps and queues in a comparison-based model.

History-independent data structures are well-studied when the objective is to minimize RAM computations. Examples include SHI hashing [20,46,47], dictionaries and order-maintenance [20], and history-independent data structures for computational geometry [21,61].

Golovin [32, 33] began an algorithmic study of history-independent data structures in external-memory. First, Golovin proposed the B-treap [32], a strongly history-independent external-memory B-tree variant based on treaps [7].

Golovin notes that while the B-treap is a unique-representation data structure supporting B-tree operations with low overhead, from a practical point of view, it is complicated and difficult to implement [33]. Golovin thus proposes a strongly history-independent B-skip list [33] as a simpler alternative to the B-treap. This data structure achieves $O(\log_B N)$ I/Os in expectation for searches and updates and $O(k/B + \log_B N)$ I/Os in expectation for range queries returning k elements.

Golovin's B-skip list builds upon the folklore extension of skip lists (see e.g., [1, 25, 26, 33]): promote an element from one level to the next with probability $1/B$, rather than $1/2$. The folklore B-skip list's I/O bounds are only in expectation, and do not extend to good high probability bounds (see Lemma 15).

Other Applications of History Independence

The theoretical work on history independence in external memory complements the security and experimental work on history-independent data structures for persistent storage, such as file systems, cloud storage, voting systems and databases [10–12, 19, 44, 45, 56].

History independence has many applications in security and privacy. For example, history-independent data structures help to guarantee privacy in incremental signature schemes [43] and vote-storage mechanisms [19, 44, 45].

History-independent data structures often have nice properties. For example, skip lists [53] have weakly history-independent topologies, and are weight balanced [48] in a randomized sense. Canonical representations [36] find applications in other areas besides security, e.g. concurrent data structures [58], equality testing [60] and dynamic and incremental algorithms [2, 52].

2. PRELIMINARIES

An event E_n on a problem of size n occurs ***with high probability*** (***whp***) if $\Pr[E_n] \geq 1 - 1/n^c$ for some constant c. Often the event E_n is parametrized by some constant d, in particular, because the event is defined using Big Oh notation. (See, e.g., Theorem 11.) In this case, we can say more strongly that for every c, there is a d so that $\Pr[E_n] \geq 1 - 1/n^c$. We use high probability guarantees to bound a data structure's performance more tightly than can be done using expectation alone. Even if a data structure performs well in expectation, it may have a large number of poorly-performing operations (see, for example, Lemma 15).

Two instances I_1 and I_2 of a data structure are in the same ***state*** if they cannot be distinguished via any sequence of operations on the data structure. The ***memory representation*** of an instance I of a data structure is the bit representation of I, including data, pointers, unused buffer space, and all auxiliary parts of the structure, along with the physical addresses at which they are stored.

DEFINITION 4 (WEAK HISTORY INDEPENDENCE). *A data structure is weakly history independent (WHI) if, for any two sequences of operations X and Y that take the data structure from initialization to the same state, the distribution over memory representations after X is performed is identical to the distribution after Y.*

2.1 Building Blocks for History Independence

We use history-independent allocation [47] as a black box. We also use the weak history-independent dynamic arrays [36], rather than strongly independent dynamic arrays [36, 47].[3]

Weakly history-independent dynamic arrays take constant amortized time per update with high probability. The idea is to maintain the following invariants. For array A storing n elements: (1) the size $|A|$ is uniformly and randomly chosen from $\{n, \ldots, 2n-1\}$, and, (2) after each insert or delete resize with probability $\Theta(1/|A|)$.

2.2 Performance Advantages of WHI over SHI

We focus on weak history independence because of its performance benefits. In particular, weak history independence allows us to have high-probability guarantees in our data structures.

Strong history independence for reversible data structures requires a canonical representation [36].[4] While canonical representations are useful to have, maintaining them imposes strict limitations on the design and efficiency of the data structure, as argued by several authors [23, 36]. Moreover, amortization, strong history independence, and high-probability guarantees are largely incompatible.

In particular, SHI dynamic arrays cannot achieve the same with high probability guarantees as WHI dynamic arrays.

OBSERVATION 1. *No strongly-history-independent dynamic array can achieve $o(N)$ amortized resize cost per insert or delete with high probability.*

PROOF. Consider a strongly-history-independent dynamic array that needs to be strictly greater than 50% full. (The proof generalizes to arbitrary capacity constraints.) For integer k, the adversary chooses a random $\ell \in \{k, k+1, \ldots, 2k\}$. Then the adversary inserts up to ℓ elements into the array, and then alternates between adding and removing an element from the array, so that the array alternates between having ℓ and $\ell + 1$ elements.

Given the capacity constraints on the array, there must be at least two different canonical representations for the arrays of sizes $k, k+1, \ldots, 2k$. Thus, there is a probability of at least $1/k$ that the adversary forces an array resize (with cost $\Omega(N)$) on every insert and delete in the alternation phase.

Observe that k can be arbitrarily large. No matter how long the data structure runs, it cannot avoid an $\Omega(N)$ resize with probability greater than $1 - 1/k$. □

Most importantly, this observation applies to PMAs as well. A PMA with N elements at a given time is required to have size $\Theta(N)$, so it generalizes the dynamic array. That is, Observation 1 lets us conclude the following:

REMARK 1. *A strongly history-independent PMA cannot give any $o(N)$ amortized with high probability operation bounds. In contrast, our weakly history-independent PMA has a $O(\log^2 N)$ bound (Theorem 1).*

[3]Here it is assumed the contents of the dynamic array are stored (internally) in a history independent manner—thus the size of the array should not depend on the history of inserts and deletes.

[4]A data structure is reversible if the state-transition graph is strongly connected [36], which is true of all structures in this paper and most structures that support deletes.

Observation 1 similarly generalizes to other amortized data structures with large worst-case costs. Thus, while strong history independence provides stronger security guarantees, it can come at a high performance cost.

2.3 Oblivious Adversary and Oblivious Observer

Our performance analyses assume an ***oblivious adversary***, which determines the sequence of operations presented to the data structure. The oblivious adversary cannot see the outcomes of the data structure's coin flips nor the current state of memory. Said differently, the adversary is required to choose the entire sequence of operations before the data structure even starts running. The oblivious adversary is used for analyzing randomized structures such as skip lists [53] or treaps [7].

Our (weak) history-independence analyses assume an ***oblivious observer***. The observer cannot control the input sequence and does not see the data structure's coin flips. The observer gets to observe the data structure's memory representation once. We prove history independence by showing that for every state of the data structure, the distribution of memory representations is the same, regardless of how the data structure got to that state.

3. HISTORY-INDEPENDENT PACKED MEMORY ARRAY

This section gives a history-independent packed memory array (PMA). A PMA is an $\Theta(N)$-sized array that stores a sequence of elements in a user-specified order. There are up to $O(1)$ gaps between consecutive elements to support efficient insertions and deletions.

A PMA with N elements supports the following:

- `Query`(i, j)—return the ith through jth elements of the PMA, inclusive, where $0 \le i \le j < N$.
- `Insert`(i, x)—insert x as the ith element of the PMA, where $0 \le i \le N$. Elements with rank i through $N-1$ before the insert become the elements with $i+1$ though N after the insert.
- `Delete`(i)—delete the ith element of the PMA, where $0 \le i < N$.

The PMA's performance is given in Theorem 1.

3.1 High-Level Structure of HI PMA

Packed-memory arrays and sparse tables in the literature [14, 16–18, 38, 41, 66] are not history independent; the size of the array, densities of the subarrays, and rebalances depend strongly on the history.

We guarantee history independence for our PMA as follows. First, we ensure the size of the PMA is history independent. We resize using the HI dynamic array allocation strategy [36], as summarized in Section 2.1.

Next, we ensure that the N elements in the array of size $N_S = \Theta(N)$ are spread throughout the array according to a distribution (given below) that is independent of past operations.

We maintain this history-independent layout recursively. At the topmost level of recursion, we (implicitly) maintain a set of size $\Theta(N_S/\log N_S)$, which we call the ***candidate set***. The candidate set consists of elements that have rank $N/2 \pm \Theta(N_S/\log N_S)$. We pick a random element from the candidate set, which we call the ***balance element***. If the

balance element has rank r, then we recursively store the first $r-1$ elements in the first half of the array and the remaining $N-r-1$ elements in the second half of the array.

In general, when we are spreading elements out within a subarray A of the PMA, the candidate set has size $\Theta(|A|/\log N_S)$, and as before, the balance element is randomly chosen from this set. The base case is when $|A| = \Theta(\log N_S)$, at which point the elements are spread evenly throughout A.

Thus, how the elements are spread throughout the PMA depends only on the size N_S (which is randomly chosen as described in Section 2.1), the number of elements in the PMA N, and the random choices of all the balance elements.

We maintain the balance elements in each candidate set using a simple generalization of reservoir sampling [62] in which there are deletes, described below. In this particular instance of reservoir sampling, the size of a candidate set at any given level of recursion stays the same, unless the size of the entire PMA changes.

See Figure 1 for an illustration of our PMA. The top part represents how each level of recursion partions the elements into ranges. We show repeated elements across levels to aid visualization; they are only stored once in the data structure (at the bottom level). The division of elements into ranges at each level helps in maintaining the balance elements, which are stored in a separate structure.

3.2 Reservoir Sampling with Deletes

We first review a small tweak on standard reservoir sampling [62], ***reservoir sampling with deletes***, which we use to help build the PMA.

Game: We have a dynamic set of elements. The objective is to maintain a uniformly and randomly chosen ***leader*** of the set, where each element in the set has equal probability of being selected as leader. In other words, we are interested in reservoir sampling with a reservoir of size 1.

Since the set is dynamic, at each step t, an element may be added to or deleted from the set. The adversary is oblivious, which means that the input sequence cannot depend on the particular element chosen as leader.

We can maintain the leader using the following technique. Let n_t denote the number of elements in the set at time t (including any newly-arrived element). Initially, if the set is nonempty, we choose the leader uniformly at random. When a new element y arrives, y becomes the leader with probability $1/n_t$; otherwise the old leader remains. When an element is deleted, there are two cases. If that element was the leader, then we choose a new leader uniformly at random from the remaining elements in the set. If the deleted element was not the leader, the old leader remains.

LEMMA 5 ([62]). *At any time step t, if there are n_t elements in the pool, then each element has a probability $1/n_t$ to be the leader.*

3.3 Detailed Structure of the HI PMA

The size N_S of our history-independent PMA is a random variable that depends on the number of elements N stored in the PMA (similar to dynamic arrays in Section 2.1). We select parameter $\hat{N}$ randomly from $\{N, \ldots, 2N-1\}$, and N_S is a function of $\hat{N}$ (as described below).

We view the PMA as a complete binary tree of ranges, where a ***range*** is a contiguous sequence of array slots. This tree has height $h = \lceil \log \hat{N} - \log\log \hat{N} \rceil$. The root is the entire PMA and has depth 0. The leaves in the binary tree are ranges comprising $\lceil C_L \log \hat{N} \rceil$ slots, and C_L is a constant to be determined later. Thus, the PMA has a total of $N_S = 2^h \lceil C_L \log \hat{N} \rceil \le (2C_L+1)\hat{N} = \Theta(N)$ slots.

Consider a range R in the binary tree with left child R_1 and right child R_2. Recall from Section 3.1 that the ***balance element*** b_R of R is the first element of R_2; all the elements in R of smaller rank than b_R are stored in R_1. The values of b_R for each range/node are stored in a separate tree.

For each non-leaf range R at depth d, define the ***candidate set*** M_R to be the $\lceil c_1 \hat{N} 2^{-d} / \log \hat{N} \rceil$ middle elements of R. More precisely, if R holds ℓ elements, then we fix the size of M_R and set the first element of M_R to be the $1 + \lceil \ell/2 \rceil - \lceil |M_R|/2 \rceil$th element of R.

Our PMA is parameterized by a constant $0 < c_1 < 1 - 6/\log \hat{N}$.[5] A larger c_1 reduces the amortized update time and increases the space. We require $C_L \ge 1 + c_1 + 6/\log \hat{N}$. The value of C_L and c_1 need not change as $\hat{N}$ changes—values such as $c_1 = 1/2$ and $C_L = 2$ will work for sufficiently large $\hat{N}$ (over 4096 in this case).

3.4 Dynamically Maintaining Balance Elements

As elements are inserted into or deleted from the PMA, the candidate set M_R for some range R could change. This change might be caused by a newly inserted or deleted element that belongs to M_R or just because insertions at one end of R cause the median element of R to change.

As the candidate set M_R changes, we maintain the invariant that the balance element b_R is selected uniformly and randomly from M_R. (In particular, this means that b_R is selected history-independently.) We use reservoir sampling with deletes as the basis for maintaining this invariant.

INVARIANT 6. *After each operation, for each range R, balance element b_R is uniformly distributed over the candidate set M_R.*

Whenever one element leaves their candidate set, another one joins, since the candidate set size of each range is fixed between rebuilds of the entire PMA. Thus, we describe how to maintain the candidate set when exactly one element is added, and one leaves.

If the balance element is the element leaving the candidate set, we select the new balance element uniformly at random. Otherwise, when a new element enters the candidate set, it has a $1/|M_R|$ chance of becoming the new balance element (as in reservoir sampling).

When the balance element of a range changes, we rebuild the entire range and all of its subranges. Rebuilding a range of $|R|$ slots can be done in $\Theta(|R|)$ time; see Lemma 10.

3.5 Detecting Changes to the Candidate Set

In order to determine how inserts and deletes affect the candidate set of a range R, we need to know the rank of the element being inserted or deleted, the current candidate set of R, and the rank of the current balance element of R.

The rank of the element being inserted and deleted is specified as part of the insert or delete operation.

To compute the other information, our PMA maintains an auxiliary data structure containing the number of elements

[5] When $\hat{N} \le 64$, no such c_1 exists. For such small $\hat{N}$, we use a dynamic array instead.

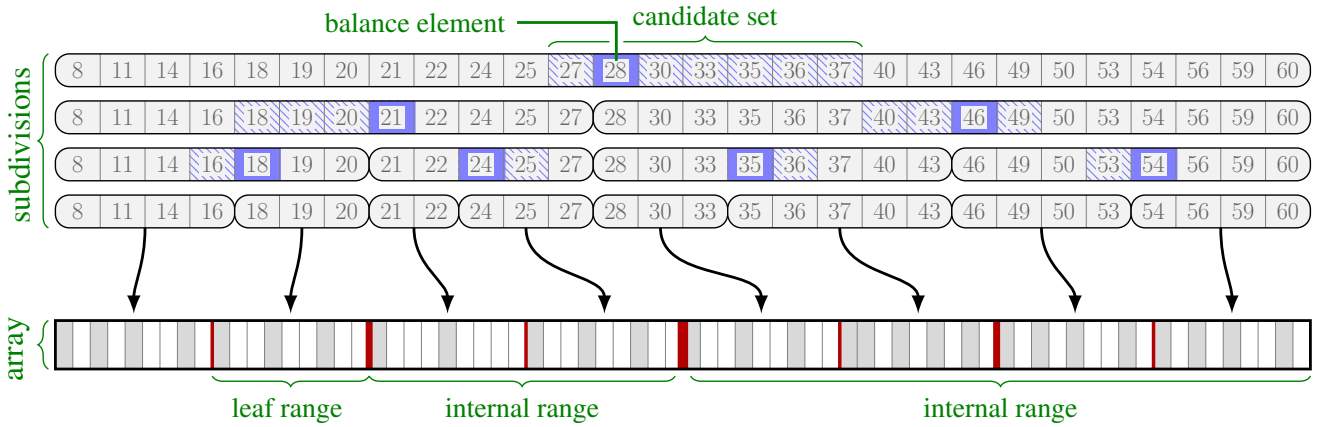

Figure 1: Illustration of our PMA showing the subdivisions of elements into ranges. In each range, the balance element is framed and the candidate set is hatched. The actual array is represented in the bottom, where occupied slots are shaded. The candidate-set size at any given level does not depend on the range. The rank of each balance element is stored in a separate tree.

ℓ_R in each range R. During an insert or delete, as we descend the tree of ranges, we keep track of the ranks of the first and last element in each range as follows. Suppose we are at range R whose first element has rank x. If we descend to R_1, then we know the rank of the first element of R_1 is also x. If we descend to R_2, then the rank of its first element is $x + \ell_{R_1}$. Given the rank of the first element of a range and the number of elements in that range, it is easy to compute its candidate set. Note also that the rank of the balance element of a range R whose first element has rank x is $x + \ell_{R_1}$.

We need to store ℓ_R for each range so that it can be accessed efficiently, both in terms of operations and I/Os. Since the ranges form a complete binary tree, we store the numbers ℓ_R in a binary tree organized in a Van Emde Boas layout (see [14,51]). We call this auxiliary data structure the ***rank tree***.

The Van Emde Boas layout is a deterministic, static, cache-oblivious—and hence history-independent—layout of a complete binary tree. It supports traversing a root-to-leaf path in $O(\log N)$ operations and $O(\log_B N)$ I/Os. Thus, the rank tree is history independent.

Whenever the size of a range changes due to an insert or delete, we update the corresponding entry in the rank tree. Whenever we rebuild a range R in the PMA, we update all entries of the rank tree corresponding to descendants of R. Whenever we rebuild the entire PMA, we rebuild the entire rank tree.

4. CORRECTNESS AND PERFORMANCE

4.1 Balance-Element Structural Lemmas

First, we show correctness—the data structure always finds a slot for any element it needs to store.

LEMMA 7. *At all times, the size of a range is larger than the number of elements it contains.*

PROOF. Consider a range at depth d. This range has $N_S/2^d = 2^{h-d}\lceil C_L \log \hat{N} \rceil \geq C_L \hat{N}/2^d$ slots. We will show that the maximum number of elements it can contain is smaller than this number of slots.

The number of elements in the range is at most half of the elements in its parents range, plus half of the size of the parent's candidate set (rounding up). In other words, if $S(d)$ is the maximum number of elements in a range at depth d, $S(0) \leq \hat{N}$ and

$$\begin{aligned} S(d) &\leq \lceil S(d-1)/2 \rceil + \lceil M_R/2 \rceil \\ &\leq \lceil S(d-1)/2 \rceil + \frac{1}{2}\left\lceil \frac{c_1 \hat{N}}{2^{d-1} \log \hat{N}} \right\rceil + \frac{1}{2} \\ &\leq \frac{S(d-1)}{2} + \frac{c_1 \hat{N}}{2^d \log \hat{N}} + \frac{3}{2}. \end{aligned}$$

By induction,

$$S(d) \leq \frac{\hat{N}}{2^d} + \frac{c_1 d \hat{N}}{2^d \log \hat{N}} + 3.$$

Since $d \leq \log \hat{N}$, and $\hat{N}/2^d \geq \hat{N}/2^h \geq (\log \hat{N})/2$,

$$S(d) \leq \frac{\hat{N}}{2^d}(1 + c_1) + 3 \leq \frac{\hat{N}}{2^d}\left(1 + c_1 + \frac{6}{\log \hat{N}}\right).$$

Since we choose $C_L \geq 1 + c_1 + 6/\log \hat{N}$, we have $S(d) \leq C_L \hat{N}/2^d$. □

As Lemma 7 establishes, each leaf range in the PMA (which has $\Theta(\log N)$ slots) never fills up completely. Using a similar argument, it can be shown that each leaf also contains $\Omega(\log N)$ elements if $c_1 < 1 - 6/\log \hat{N}$. Because the elements are spread out evenly in the leaves, this implies there are $O(1)$ gaps between two consecutive elements.

LEMMA 8. *If $c_1 < 1 - 6/\log \hat{N}$, the leaves are always constant-factor full. There is $O(1)$ space between two consecutive elements in the array.*

Next, we prove weak history independence. As mentioned in Section 3.1, when we are spreading elements out within a subarray A of the PMA, the balance element is randomly chosen from the candidate set. The elements of A are recursively split between its children according to this balance element. The base case is when $|A| = \Theta(\log \hat{N})$, at which point the elements are spread evenly throughout A.

Thus, how the elements are spread throughout the PMA depends only on $\hat{N}$ (and the related N_S), the number of elements in the PMA N, and the random choices of all the balance elements. This immediately gives weak history-independence, as formalized in the following lemma.

LEMMA 9. *This PMA is weakly history independent.*

PROOF. We show that the memory representation of the PMA is based only on N and some randomness (in particular, the random choices made during balance element selection and the random choice of $\hat{N}$).

Let the PMA contain a set of elements S of size N, with a set of balance elements P. Then P partitions the elements of S into leaf ranges.

Since the elements are evenly spaced in each leaf range, the position of each element within the leaf is determined by the number of elements in that leaf. Since S is partitioned into leaf ranges by P, P determines the position of each element in the PMA. Thus P, $\hat{N}$, and N determine the memory representation of the data structure. By Invariant 6, P is selected from a distribution based only on N and $\hat{N}$.

Thus, any two sequences of operations X and Y that insert S into the PMA result in the same distribution on P, and the same distribution on memory representations. □

4.2 Proving the Performance Bounds

We begin by bounding the cost of a rebalance. Then we bound the total number of rebalances.

LEMMA 10. *Rebuilding a range R containing $|R|$ slots takes $O(|R|)$ RAM operations and $O(|R|/B+1)$ I/Os.*

PROOF. The algorithm first recursively chooses the balance elements for all ranges contained in this range R and updates them in the rank tree; this takes $O(|R|)$ time and $O(|R|/B + \log_B |R| + 1) = O(|R|/B + 1)$ I/Os. Then, all elements in R are gathered, and inserted into the appropriate leaf range using a sequence of linear scans. □

Our goal is to bound the cost of our PMA operations. Specifically, we want to show Theorem 11.

THEOREM 11. *Consider k (not necessarily consecutive) operations on a PMA during which its maximum size is N_M, its minimum size is $\Omega(N_M)$, and $k = \Omega(N_M)$. The amortized cost of these operations is $O(\log^2 N_M)$ with high probability with respect to k: in other words, these k operations require $O(k \log^2 N_M)$ total RAM operations with high probability.*

Before proving this theorem, we need some supporting lemmas and definitions.

DEFINITION 12. *A rebuild that is not charged to a range R is called a* **free rebuild**. *Free rebuilds come from two sources: (1) they are rebuilds of an ancestor range (whose cost is charged to the ancestor), or (2) they are triggered by the interstitial operations between the nonconsecutive operations of Theorem 11.*

We further categorize non-free rebuilds by their causes. An **out-of-bounds rebuild** *is a rebuild caused by the pivot leaving the candidate set. A* **lottery rebuild** *is a rebuild caused by deleting the pivot or by inserting into the candidate set an element that becomes a new pivot.*

Gearing up to Lemma 13, we concentrate on the cost of all rebalances at a single depth d. Let M_d be the size of the candidate set at depth d.

We give a lower bound on the probability that two out-of-bounds rebuilds happen in quick succession. This lemma holds regardless of the number of free and lottery rebuilds that happen in between.

LEMMA 13. *After any rebuild of a range R at depth d, consider a sequence of t operations on R, for any $t \in \{1, \ldots, \lfloor M_d/2 \rfloor\}$, with arbitrary free and lottery rebuilds occurring during these operations. The probability $p(t)$ that no out-of-bounds rebuild happens during these t operations is at least $1 - 2t/M_d$.*

PROOF. Let $p_i(t)$ be the probability that no out-of-bounds rebuild happens in the first t time steps, given that exactly i free and lottery rebuilds happen during the first t time steps. We prove the lemma by induction on i.

Define the ***guard number*** as the number of elements between the pivot and the closest endpoint of the candidate set.

First, the base case: for any t, $p_0(t)$ is at least the probability that the guard number after a rebuild is larger than t. Indeed, the balance element cannot be moved closer to an endpoint of the candidate set by more than one element per operation. As the pivot is sampled uniformly after any type of rebuild, we have

$$p_0(t) \geq \Pr[\text{guard number} > t] \geq 1 - 2t/M_d.$$

Now, assume by induction that $p_i(t) \geq 1 - 2t/M_d$ and we want to show $p_{i+1}(t) \geq 1 - t/M_d$. Let t' be the last time step before the $(i+1)$st non-out-of-bounds rebuild.

The following conditions ensure that there are no out-of-bounds rebuilds in the first t operations:

1. there is no out-of-bounds rebuild in the first t' operations, and
2. there is no out-of-bounds rebuild in the subsequent $t - t'$ operations.

These two events are independent since there is a fixed (free or lottery) rebuild between them. The first occurs with probability $p_i(t')$ and the second with probability $p_0(t - t')$. Thus, we have

$$\begin{aligned} p_{i+1}(t) &\geq p_i(t')\, p_0(t - t') \\ &\geq (1 - 2t'/M_d)\,(1 - 2(t - t')/M_d) \geq 1 - 2t/M_d, \end{aligned}$$

and the induction is complete. □

DEFINITION 14. *Consider an out-of-bounds rebuild of a range R at depth d.*

We call this rebuild a **good out-of-bounds rebuild** *if R has only free and lottery rebuilds for the next $M_d/4$ operations.*

By Lemma 13, an out-of-bounds rebuild is good with probability at least $1/2$.

We are now ready to prove Theorem 11.

PROOF OF THEOREM 11. Consider the sequence of rebuilds of ranges at a given depth d. We analyze lottery rebuilds and out-of-bounds rebuilds separately. Since variations in $\hat{N}$ slightly change the candidate set size, let M denote the smallest candidate-set size at depth d over the k operations. However, since $N = \Theta(N_M)$ at all times, $M = \Theta(|M_d|)$.

We give a (weak) bound on k/M which helps show that the high-probability bounds hold with respect to k.

In particular,

$$k/M \geq \frac{k \log N_M}{N_M} \geq \frac{k \log k \log N_M}{N_M \log k} = \Omega(\log k),$$

since $k/\log k = \Omega(N_M/\log N_M)$ if $k = \Omega(N_M)$.

Each operation has probability at most $1/M$ of causing a lottery rebuild. Thus, there are k/M lottery rebuilds in expectation. Then using Chernoff bounds, the probability that we have more than $(1+\delta)k/M$ rebuilds is less than $e^{-\delta k/3M}$. Recall that $k/M = \Omega(\log k)$. Substituting, there are $O(k/M)$ lottery rebuilds with high probability.

Now, we bound the out-of-bounds rebuilds. By the pigeonhole principle, there cannot be more than $k/(M/4) + 2^d = O(k/M)$ good out-of-bounds rebuilds (the second term comes from the number of ranges at depth d).

We bound how many bad out-of-bounds rebuilds can happen before reaching this limit on good out-of-bounds rebuilds. Any out-of-bounds rebuild is good with probability at least $1/2$. Then after $k/M = \Omega(\log k)$ out-of-bounds rebuilds, we obtain $\Theta(k/M)$ good out-of-bounds rebuilds with high probability, again by Chernoff bounds. As we can only get $O(k/M)$ good rebuilds, we have $O(k/M)$ out-of-bounds rebuilds in total.

Therefore, every k operations, there are $O(k/M)$ out-of-bounds and lottery rebuilds of ranges at depth d with high probability. Each rebuild costs $O(M \log N_M)$ RAM operations by Lemma 10 (because that is the number of slots in a range at depth d). Thus, the total rebuild cost for depth d is $O(k \log N_M)$.

Having determined the cost of rebalancing at each depth, we account for the total cost. The amortized rebalance cost, summing over all $h = O(\log N_M)$ levels, is $O(\log^2 N_M)$.

Each resize costs $O(N_M)$ and occurs with probability $O(1/N_M)$ after every insertion. Using Chernoff bounds, there are $O((k \log N_M)/N_M) = \Omega(\log k)$ resizes after k operations with high probability, leading to an additional amortized cost of $O(\log N_M)$.

Finally, each insert or delete requires extra bookkeeping: we must find the appropriate leaf range to insert the element by traversing the rank tree. This traversal takes $O(\log N_M)$ RAM operations, and rebuilding the leaf so that the elements are still evenly spaced takes $O(\log N_M)$ RAM operations.

This gives a total of $O(k \log^2 N_M)$ total RAM operations with high probability.

Using similar analysis, we can also bound the I/O performance. Recall that rebalancing a range of size R takes $O(R/B + 1)$ I/Os, and traversing a tree in the Van Emde Boas layout requires $O(\log_B N_M)$ I/Os. Carrying these terms through the above proof gives the desired bounds. □

To complete the proof of Theorem 1, we need to extend this analysis to handle the PMA changing size significantly.

PROOF OF THEOREM 1. We partition the $k = \Omega(N)$ operations on the PMA into types based on the size of the PMA. In particular, let $\hat{N}_t$ be the value of $\hat{N}$ during the tth operation. Then operation t is of type 0 if $N \geq \hat{N}_t > N/2$, type 1 if $N/2 \geq \hat{N}_t > N/4$, and type i if $N/2^i \geq \hat{N}_t > N/2^{i+1}$ for $0 \leq i \leq \lceil \log N \rceil$.

We analyze each type of operations as a whole, and bound its total cost, summing to $O(k \log^2 N)$ RAM operations in total. Each type is analyzed using two cases.

First, consider a type i which has at least $\sqrt{N}$ total operations. Then by Theorem 11, these operations take amortized $O(\log^2 N)$ RAM operations with high probability.

Second, consider a type i which has less than $\sqrt{N}$ operations. We call the operations of these types ***small-type operations***. We show that the total cost of all small-type operations is a lower-order term.

Since the PMA begins as empty, and each operation can only insert one element, there are at least $N/2^{i+1}$ operations of type i. Thus, each small-type operation t has $\hat{N}_t \leq \sqrt{N}$.

Then overall, a type which has less than $\sqrt{N}$ operations must operate on a PMA of size $\leq \sqrt{N}$. Thus, each type has total cost $O(N)$. Summing over the $O(\log N)$ such types, we get a worst-case total cost of $O(N \log N)$ for small-type operations; amortizing gives a cost of $O(\log N)$ RAM operations.

Thus the total amortized cost is $O(\log^2 N)$ with high probability with respect to N. The I/O cost to rebuild range R is $|R|/B$ by Lemma 10; carrying this term through the above analysis, we obtain the desired I/O bounds. □

4.3 Experimental Results

We implemented a normal PMA and our history-independent PMA. We found that while there was approximately a factor of 7 overhead in the run time, the asymptotic performance matched our analysis.

We also examined the number of element moves required during an insert. Figure 2 shows the number of moves required divided by $N \log^2 N$ vs. the number of elements inserted. The linear nature of this data supports our theoretical analysis.

These tests were run on a Dell server with an Intel Xeon processor (E5-2450 @ 2.10GHz). In these tests, inserting 100 million random numbers took approximately 23 minutes. Additionally the space overhead ranged from 1.8 to 5 times the number of elements.

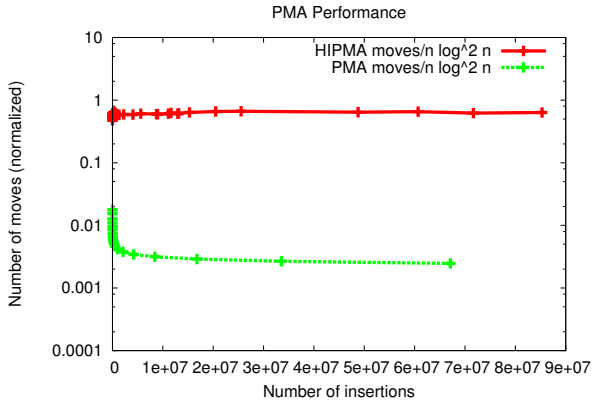

Figure 2: Experimental results of runtime for random inserts on normal and history-independent PMAs.

The history-independence of our PMA depends on the balance elements being uniformly distributed across the candidate set; see Lemma 5. To test this, we inserted values of 1-100,000 sequentially into a history-independent PMA and recorded the position of the balance for each range within this PMA where the candidate set size was eight or greater. We did this test 10,000 times and used the χ^2 goodness-of-fit test to compare these balance elements with a uniform

distribution. To ensure enough samples we only looked at ranges where the expected count for each bucket was ten or greater. This resulted in 148 p-values. If our null hypothesis, that these balances are uniformly distributed, holds, these p-values themselves should be uniformly distributed. Running the χ^2 goodness-of-fit test across these p-values showed a result consistent with data coming from a uniform distribution (p=0.47, n=148). As a result we can say there is no statistically significant evidence of that our balance elements have a deviation from a uniform distribution.

5. HISTORY-INDEPENDENT CACHE-OBLIVIOUS B-TREE

In this section we prove Theorem 2, which establishes the performance of our history-independent cache-oblivious B-tree. This N-element data structure, must, without knowledge of the block size B,

- insert or delete items with $O((\log^2 N)/B + \log_B N)$ amortized I/Os with high probability,
- use $O(N)$ space, and
- answer range queries containing k elements in $O(\log_B N + k/B)$ I/Os.

When $B = \Omega(\log N \log\log N)$, which is reasonable on today's systems, $\frac{\log^2 N}{B} + \log_B N = O(\log_B N)$, so that our history-independent cache-oblivious B-tree matches the I/O complexity of a standard B-tree.

The requirements above are similar to those achieved by our PMA. The key difference is that PMA items are searched by rank (before being inserted, deleted, or completing a range search) rather than value.

By slightly augmenting our PMA, we obtain a cache-oblivious B-tree achieving the above bounds. We call our data structure the ***augmented PMA***.

The augmented PMA has an additional, static-topology tree associated with it. Recall that our PMA has the sizes of each range stored in a complete binary tree, in a Van Emde Boas layout. We store an additional tree storing the *values* of each balance element. The two trees are identically structured, and identically maintained. Thus, this leads to only a constant factor increase in both space and running time. To search (by value) in the augmented PMA, we traverse a path in the new tree of balance-element values. This costs $O(\log_B N)$ I/Os and $O(\log N)$ operations. Once we have found the element, we can determine its rank by traversing the rank tree, summing the sizes of any left children each time we go to a right child.

Once the rank of the element is known, we insert, delete, or perform range queries as in the normal PMA, establishing the desired bounds.

6. HISTORY-INDEPENDENT EXTERNAL-MEMORY SKIP LIST

In this section we prove Theorem 3. We give a history-independent external-memory skip list.

In-memory skip lists support updates and queries in $O(\log N)$ time whp. The natural extension to external memory [1, 25, 26, 33] promotes elements with probability $1/B$ rather than probability $1/2$. We prove that for this extension, the high-probability I/O bounds are asymptotically no better than those of an in-memory skip list run in external memory (Lemma 15).

We build an external-memory skip list with good (i.e., B-tree-like) high-probability bounds for searches, updates, and range queries, while retaining the structure of the folklore B-skip list as much as possible.

6.1 High-Level Structure of the HI External-Memory Skip List

First, we describe why the folklore B-skip list fails to achieve high-probability I/O bounds. Then, we present the approach used in our skip list.

Golovin and others [1, 25, 26, 33] promote elements with probability $1/B$ (rather than $1/2$, as in the in-memory skip list). Consider an ***array***, a sequence of contiguous elements at any level that have not been promoted to the next level (i.e., lie between two ***promoted elements***). These arrays can have size $O(B \log N)$ whp, which is $O(\log N)$ times larger than the expected length. When this list is embedded in an array (analogous to the nodes in a B-tree), then a scan to search for these elements costs $O(\log N)$ I/Os whp.

We change the promotion probability to $1/B^\gamma$, where $1/2 < \gamma < 1 - \log\log B/\log B$. Now, all arrays have size $O(B^\gamma \log N)$ whp, so searches and updates take $O(B^\gamma \log N/B) = O(\log_B N)$ I/Os whp.

While a promotion probability of $1/B^\gamma$ results in fast searches and updates, it slows down range queries. If we pack arrays at the leaf level (the ***leaf arrays***) of the skip list into disk blocks [33], then most disk blocks will be underutilized, containing only B^γ elements on average. Thus, a range query returning k elements may require $O(\log_B N + k/B^\gamma)$ I/Os, which is worse than the target of $O(\log_B N + k/B)$ I/Os.

For efficient range queries, we pack multiple arrays into disk blocks at the leaf level as follows. Contiguous arrays, delimited by elements promoted twice, are packed together into a ***leaf node***. A leaf node is stored consecutively on disk; see Figure 3.

The packing strategy described above permits some leaf nodes to get too large, achieving size $\Theta(B^{2\gamma} \log N)$. If we pack elements as densely as possible, then every new insert would require rewriting the entire node at a cost of $O(B^{2\gamma-1} \log N)$ I/Os, which is worse than $\Theta(\log_B N)$ I/Os.

Similar to PMAs and HI dynamic arrays [36], we leave empty spaces between the elements in the leaves to support efficient inserts. We do so in a way that maintains history independence; see Invariant 16.

6.2 Detailed Structure of the HI External-Memory Skip List

A skip list S is a series of linked lists $\{S_0, S_1, \ldots, S_h\}$, where $S_0 \supseteq S_1 \supseteq \ldots \supseteq S_h$. Let $H = \{0, 1, \ldots, h\}$ be the ***levels*** of the skip list, and h the ***height*** of the skip list. The base list S_0 contains all the elements in the skip list; we call level 0 the ***leaf level***.

Each S_i, for $0 < i \leq h$, stores a sorted subset of the elements from S_{i-1}, along with the special element `front` to mark the beginning of the list; see Figure 3. The function $\mathit{level} : S_0 \to H$ determines the highest list that contains an element x, that is, if $\mathit{level}(x) = i$ then $x \in S_j$ for all $j \leq i$ and $x \notin S_k$ for $i < k \leq h$.

The height of each element is determined randomly, according to ***promotion probability*** p. Specifically, for $i \in H - \{0\}$, if an element is in S_{i-1}, it is also in S_i with probability p. Thus, $\mathit{level}(x)$ of an element x is the number of coin flips before we see a tail, when using a biased coin with probability p of flipping a head.

Next, we show that the folklore B-skip list, which has a

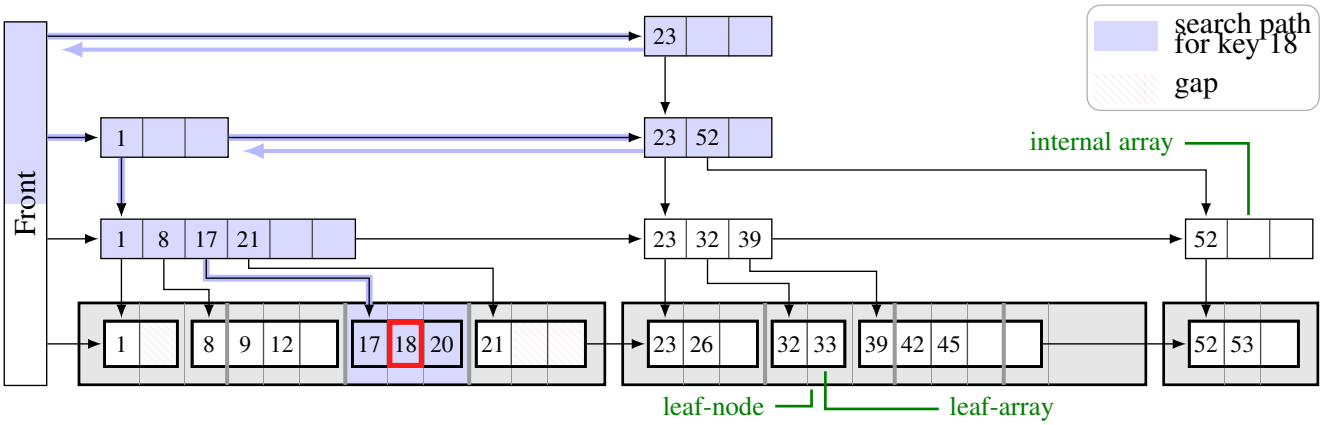

Figure 3: Illustration of a search path for element key 18 in the external skip list ($B = 3$ and $p = 1/2$).

promotion probability of $1/B$, performs poorly for some elements with high probability.

LEMMA 15. *In a B-skip list with promotion probability $p = 1/B$, there exist $\Omega(\sqrt{NB})$ elements with search cost $\Omega(\log(N/B))$ with high probability with respect to N/B.*

For our history-independent external-memory skip list, we choose promotion probability $p = 1/B^\gamma$, where $1/2 < \gamma \leq 1 - \log\log B/\log B$ is a constant. Let $1/p = B^\gamma$ be integral to simplify analysis. We parameterize our running times by $\varepsilon > 0$, with $\gamma = (\varepsilon + 1)/2$.

The parameter γ can be tuned—there is a trade-off between the cost of a range query and the worst-case cost of insertion. Specifically, a range query returning k elements has a cost of $O(\frac{1}{\varepsilon}\log_B N + k/B)$, while the worst-case insert cost is $O(B^\varepsilon \log N)$ I/Os. The expected insert cost remains $O(\log_B N)$ I/Os for all allowed values of γ.

We now describe how to partition the elements into arrays at nonleaf levels. We also describe how to pack the leaf arrays into leaf nodes.

Partitioning Non-Leaf Levels. We partition the list S_i at each level i for $1 \leq i \leq h$ into ***arrays***. The array at level i starts with a ***promoted element***, i.e., element x with *level* $(x) \geq i + 1$. It contains all elements up to (and not including) the next promoted element. The ***size*** of an array is the number of elements stored in it plus any empty slots. We maintain these sorted arrays history independently [36].

Partitioning the Leaf Level. We store the leaf level compactly to support I/O-efficient range queries. The ***leaf arrays*** at the leaf level are packed into a ***leaf node***. Formally, a leaf node $\mathcal{B}$ is a set of contiguous leaf arrays starting at some element x that has been promoted twice, that is, *level* $(x) \geq 2$.

We store the leaf arrays history independently (see Section 2.1), with the following modification: even when a leaf array has n elements with $n \leq B^\gamma$ elements, we maintain its size $n_s \geq B^\gamma$. We call the extra $n_s - n$ array slots ***gaps***. This modification retains history independence.

INVARIANT 16. *Let n be the number of elements in a leaf array of total size n_s then:*
- *If $n \leq B^\gamma$, then n_s is uniform in $[B^\gamma, 2B^\gamma - 1]$.*
- *If $n \geq B^\gamma$, then n_s is uniform in $[n, 2n - 1]$.*

Searches, Insertions, and Deletions. Search is implemented exactly as in a standard skip list: for an element y, start at the top list S_h, and scan right till a value $x > y$ is reached, in which case, descend a level down and continue till y is found or shown not to exist.

To insert or delete an element y, search for the leaf array where y belongs and insert or delete it. This involves shifting elements in the leaf array and causes an array resize with probability $O(1/B^\gamma)$ by Invariant 16. If a resize occurs, rebuild the entire leaf node containing the array.

When inserting y, determine *level* $(y) = \ell$ by tossing a biased coin with probability of heads p. At levels $1 \leq i < \ell$, y starts an array, splitting the existing array into two. If $\ell \geq 2$, y starts a leaf node, splitting the existing leaf node into two.

When deleting y, at levels $1 \leq i < \ell$, merge the leaf array that y started with its predecessor. If $\ell \geq 2$ then merge the leaf node that y started with its predecessor.

6.3 History Independence of External-Memory Skip List

The history independence of our external-memory skip list follows immediately from the following facts:
- *level* (x) for each element x is generated randomly,
- the elements within an array appear in sorted order,
- the size of each array is chosen history-independently (by Invariant 16 for leaf arrays and [36] for non-leaf arrays),
- within a leaf node, the leaf arrays are packed contiguously in sorted order, and,
- each array is allocated in blocks of size $\Theta(B)$ history-independently [47].

6.4 Performance Analysis

We bound the height of the external-memory skip list. The proof is similar to standard skip lists.

LEMMA 17. *An external-memory skip list with promotion probability p has height $h = O(\log_{1/p} N)$ whp.*

PROOF. For any element x, the probability that its level is more than $O(\log_{1/p} N)$ is given by: $\Pr[\textit{level}\,(x) \geq c\log_{1/p} N] \leq p^{c\log_{1/p} N} = 1/N^c$.

Applying the union bound we get, $\Pr[\forall x, \textit{level}\,(x) \geq c\log_{1/p} N] \leq N(1/N^c) = 1/N^{c-1}$. □

Thus, the height of our external-memory skip list with $p = 1/B^\gamma$ is $O(\log_B N)$.

Search. To analyze searches, we bound the size of the arrays and leaf nodes.

An array contains elements between two consecutive promoted elements. Thus, the size of an array is bounded by the length of the longest sequence of tails, when flipping a biased coin with $\Pr[head] = 1/B^\gamma$, which is $O(B^\gamma \log N)$.

LEMMA 18. *The number of I/Os required to perform a search is $O(\log_B N)$ with high probability.*

PROOF. Similar to the *backward analysis* in standard skip lists [53], examine the search path from bottom up starting at the leaf level. Each element visited by the path was either promoted and the search path came from the top, or was not promoted and the search path came from the left. The number of down moves is bounded by the height $h = c \log_B N$ (Lemma 17). At each level, the search path traverses at most two arrays.

The total length of the arrays touched by the search path is bounded by the number of coin flips required to obtain $c \log_B N$ heads, where $\Pr[head] = 1/B^\gamma$. We need $O(B^\gamma \log N)$ coin flips whp. Thus, the number of I/Os during the search at nonleaf levels is $O(\log N/B^{1-\gamma} + \log_B N) = O(\log_B N)$ whp, since $\gamma \leq 1 - \log\log B/\log B$. At the leaf level, the search scans one leaf array, which costs $O(B^\gamma \log N/B) = O(\log_B N)$. □

Insert. To bound the insert cost, we bound the cost of rebuilding a leaf node. The size of a leaf node is the number of elements stored in it plus the number of gaps.

The number of elements in a leaf node is bounded by the length of the longest sequence of tails in N coin flips with $\Pr[head] = p^2 = 1/B^{2\gamma}$, which is $O(B^{2\gamma} \log N)$ whp.

The number of gaps between any k consecutive leaf elements can be shown by a Chernoff bound argument to be $O(k + B^\gamma \log N)$. Thus, the size of a leaf node is $O(B^{2\gamma} \log N)$ whp. Rebuilding it costs $O(B^{2\gamma} \log N/B) = O(B^\varepsilon \log N)$ I/Os.

LEMMA 19. *The cost of performing an insert or delete operation is amortized $O(\log_B N)$ I/Os whp, with a worst case cost of $O(B^\varepsilon \log N)$ I/Os whp.*

PROOF. When an element y is inserted or deleted, the splits and merges at levels $1 \leq i < \mathit{level}(y)$ are dominated by the search cost of $O(\log_B N)$.

The cost of inserting in a leaf array is dominated by the cost of rebuilding the leaf node: $O(B^\varepsilon \log N)$ I/Os whp. The rebuild occurs with probability $O(1/B^\gamma)$. The amortized I/O cost is then $O(B^{2\gamma-1} \log N/B^\gamma) = O(\log N/B^{1-\gamma}) = O(\log_B N)$ in expectation and whp with respect to the number of operations, since $\gamma \leq 1 - \log\log B/\log B$. □

Range Query. To analyze a range query on a range of size k, we bound the number of leaf nodes across which the k elements are spread. That is, we bound the number of heads obtained on k biased coin flips with $\Pr[head] = 1/B^{2\gamma}$.

LEMMA 20. *The number of leaf nodes across which k consecutive leaf elements are stored is $O(\frac{1}{\varepsilon}\log_B N + k/B)$ with high probability.*

LEMMA 21. *A range query returning k elements costs $O(\frac{1}{\varepsilon}\log_B N + k/B)$ I/Os with high probability.*

PROOF. We break the analysis into several cases depending on the source of the cost.

- If the k consecutive elements fit in a single leaf node and there are no gaps, then the size of each such leaf array is $O(B^\gamma \log N)$ whp. Thus, $O(B^\gamma \log N/B + k/B) = O(\log_B N + k/B)$ I/Os suffice.
- If the k consecutive elements span across arrays with gaps. The sum of sizes of the gaps between these elements is $O(k + B^\gamma \log N)$. Thus, a range query takes $O(k/B + B^\gamma \log N/B) = O(\log_B N + k/B)$ I/Os.
- If the k consecutive elements span several leaf nodes, by Lemma 20 they span $O(k/B + \frac{1}{\varepsilon}\log_B N)$ leaf nodes. Every time the range query scan crosses a leaf node boundary, we incur an I/O to bring in the next leaf node, which requires $O(k/B + \frac{1}{\varepsilon}\log_B N)$ I/Os.

Thus, overall a range query scanning k consecutive elements takes $O(\frac{1}{\varepsilon}\log_B N + k/B)$. □

Space. Finally, we bound the space used.

LEMMA 22. *The external skip list on N elements requires $\Theta(N)$ space with high probability.*

PROOF. The non leaf levels of the external skip list store $\Theta(N)$ elements with only constant factor space. At the leaf level, the number of gaps between N elements is $O(N + B^\gamma \log N) = \Theta(N)$ whp. □

Lemmas 18, 19, 21, and 22 prove Theorem 3.

7. CONCLUSION

We show that adding history independence to some external-memory data structures can come at low cost. We focus on history-independent indexing structures, alternatives to the traditional B-tree, the primary indexing structure used in databases. We give HI PMAs and cache-oblivious B-trees with the same asymptotic time and space bounds as their non-HI counterparts. We give a HI skip list that performs even better than the non-HI B-skip list because the bounds are given with high probability rather than in expectation.

Acknowledgments

This research was supported in part by NSF grants CCF 1114809, CCF 1217708, IIS 1247726, IIS 1251137, CNS 1408695, and CCF 1439084, and by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

8. REFERENCES

[1] I. Abraham, J. Aspnes, and J. Yuan. Skip B-trees. In *Proc. of the 9th Annual International Conference on Principles of Distributed Systems (OPODIS)*, page 366, 2006.

[2] U. A. Acar, G. E. Blelloch, R. Harper, J. L. Vittes, 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.

[3] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, Sept. 1988.

[4] O. Amble and D. E. Knuth. Ordered hash tables. *The Computer Journal*, 17(2):135–142, 1974.

[5] A. Anagnostopoulos, M. Goodrich, and R. Tamassia. Persistent authenticated dictionaries and their applications. *Information Security*, pages 379–393, 2001.

[6] 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.

[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] L. Arge, D. Eppstein, and M. T. Goodrich. Skip-webs: efficient distributed data structures for multi-dimensional data sets. In *Proc. of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODS)*, pages 69–76, 2005.

[9] J. Aspnes and G. Shah. Skip graphs. *ACM Transactions on Algorithms*, 3(4):37, 2007.

[10] S. Bajaj, A. Chakraborti, and R. Sion. The foundations of history independence. *arXiv preprint arXiv:1501.06508*, 2015.

[11] 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.

[12] 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.

[13] M. A. Bender, R. Cole, and R. Raman. Exponential structures for efficient cache-oblivious algorithms. In *Proc. of the 29th Annual International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 195–207, 2002.

[14] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.

[15] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. *Journal of Algorithms*, 3(2):115–136, 2004.

[16] M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul. Cache-oblivious string B-trees. In *Proc. of the 25th Annual ACM Symposium on Principles of Database Systems (PODS)*, pages 233–242, 2006.

[17] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul. Concurrent cache-oblivious B-trees. In *Proc. of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 228–237, 2005.

[18] M. A. Bender and H. Hu. An adaptive packed-memory array. *ACM Transactions on Database Systems*, 32(4):26, 2007.

[19] 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.

[20] G. E. Blelloch and D. Golovin. Strongly history-independent hashing with applications. In *Proc. of the 48th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 272–282, 2007.

[21] G. E. Blelloch, D. Golovin, and V. Vassilevska. Uniquely represented data structures for computational geometry. In *Proc. of the 11th Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 17–28, 2008.

[22] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *Proc. of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 39–48, 2002.

[23] N. Buchbinder and E. Petrank. Lower and upper bounds on obtaining history independence. In *Advances in Cryptology*, pages 445–462, 2003.

[24] J. Bulánek, M. Koucký, and M. Saks. Tight lower bounds for the online labeling problem. In *Proc. of the 44th Annual ACM Symposium on Theory of Computing (STOC)*, pages 1185–1198, 2012.

[25] P. Callahan, M. T. Goodrich, and K. Ramaiyer. Topology B-trees and their applications. In *Proc. of the 4th International Workshop on Algorithms and Data Structures (WADS)*, pages 381–392, 1995.

[26] V. Ciriani, P. Ferragina, F. Luccio, and S. Muthukrishnan. Static optimality theorem for external memory string access. In *Proc. of the 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 219–227, 2002.

[27] L. Devroye. A limit theory for random skip lists. *The Annals of Applied Probability*, pages 597–609, 1992.

[28] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *Proc. of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODS)*, pages 50–59, 2004.

[29] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. of the 40th Annual IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 285–298, 1999.

[30] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):4, 2012.

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

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

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

[34] M. T. Goodrich and R. Tamassia. Efficient authenticated dictionaries with skip lists and commutative hashing. *US Patent App*, 10(416,015), 2000.

[35] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

[36] J. D. Hartline, E. S. Hong, A. E. Mohr, W. R. Pentney, and E. C. Rocke. Characterizing history independent data structures. *Algorithmica*, 42(1):57–74, 2005.

[37] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A simple optimistic skiplist algorithm. *Proc. of the 14th Annual Colloquium on Structural Information and Communication Complexity (SIROCCO)*, page 124, 2007.

[38] A. Itai, A. Konheim, and M. Rodeh. A sparse table implementation of priority queues. *Proc. of the 8th Annual International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 417–431, 1981.

[39] R. Jacob, A. Richa, C. Scheideler, S. Schmid, and H. Täubig. A distributed polylogarithmic time algorithm for self-stabilizing skip graphs. In *Proc. of the 28th ACM Symposium on Principles of Distributed Computing (PODS)*, pages 131–140, 2009.

[40] Z. Kasheff. Cache-oblivious dynamic search trees. M.eng., Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 2004.

[41] I. Katriel. Implicit data structures based on local reorganizations. Master's thesis, Technion – Israel Inst. of Tech., Haifa, May 2002.

[42] P. Kirschenhofer and H. Prodinger. The path length of random skip lists. *Acta Informatica*, 31(8):775–792, 1994.

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

[44] D. Molnar, T. Kohno, N. Sastry, and D. Wagner. Tamper-evident, history-independent, subliminal-free data structures on prom storage-or-how to store ballots on a voting machine. In *Proc. of the 27th Annual IEEE Symposium on Security and Privacy (S&P)*, 2006.

[45] T. Moran, M. Naor, and G. Segev. Deterministic history-independent strategies for storing information on write-once memories. In *Proc. of the 34th International Colloquium on Automata, Languages and Programming (ICALP)*, 2007.

[46] 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.

[47] M. Naor and V. Teague. Anti-persistence: history independent data structures. In *Proc. of the 33rd Annual ACM Symposium on Theory of Computing (STOC)*, pages 492–501, 2001.

[48] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2(1):33–43, 1973.

[49] R. Oshman and N. Shavit. The SkipTrie: low-depth concurrent search without rebalancing. In *Proc. of the 32nd Annual ACM Symposium on Principles of Distributed Computing (PODS)*, pages 23–32, 2013.

[50] T. Papadakis, J. I. Munro, and P. V. Poblete. Analysis of the expected search cost in skip lists. In *Proc. of the 2nd Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 160–172, 1990.

[51] H. Prokop. Cache oblivious algorithms. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999.

[52] W. Pugh. *Incremental computation and the incremental evaluation of functional programs*. PhD thesis, Cornell University, 1988.

[53] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[54] 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.

[55] N. Rahman, R. Cole, and R. Raman. Optimised predecessor data structures for internal memory. In *Proc. of the 5th International Workshop on Algorithm Engineering (WAE)*, pages 67–78, 2001.

[56] D. S. Roche, A. J. Aviv, and S. G. Choi. Oblivious secure deletion with bounded history independence. *arXiv preprint arXiv:1505.07391*, 2015.

[57] N. Shavit and I. Lotan. Skiplist-based concurrent priority queues. In *Proc. of the 14th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 263–268, 2000.

[58] J. Shun and G. E. Blelloch. Phase-concurrent hash tables for determinism. In *Proc. of the 26th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 96–107, 2014.

[59] L. Snyder. On uniquely represented data strauctures. In *Proc. of the 18th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 142–146, 1977.

[60] 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.

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

[62] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.

[63] D. E. Willard. Inserting and deleting records in blocked sequential files. Technical Report TM81-45193-5, Bell Labs Tech Reports, 1981. (Cited in [66]).

[64] D. E. Willard. Maintaining dense sequential files in a dynamic environment. In *Proc. of the 14th Annual ACM Symposium on Theory of Computing (STOC)*, pages 114–121, 1982.

[65] D. E. Willard. Good worst-case algorithms for inserting and deleting records in dense sequential files. In *ACM SIGMOD Record*, volume 15:2, pages 251–260, 1986.

[66] D. E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in a good worst-case time. *Information and Computation*, 97(2):150–204, 1992.