History-Independent Cuckoo Hashing

Moni Naor*‡ Gil Segev†‡ Udi Wieder§

Abstract

Cuckoo hashing is an efficient and practical dynamic dictionary. It provides expected amortized constant update time, worst case constant lookup time, and good memory utilization. Various experiments demonstrated that cuckoo hashing is highly suitable for modern computer architectures and distributed settings, and offers significant improvements compared to other schemes.

In this work we construct a practical *history-independent* dynamic dictionary based on cuckoo hashing. In a history-independent data structure, the memory representation at any point in time yields no information on the specific sequence of insertions and deletions that led to its current content, other than the content itself. Such a property is significant when preventing unintended leakage of information, and was also found useful in several algorithmic settings.

Our construction enjoys most of the attractive properties of cuckoo hashing. In particular, no dynamic memory allocation is required, updates are performed in expected amortized constant time, and membership queries are performed in worst case constant time. Moreover, with high probability, the lookup procedure queries only two memory entries which are independent and can be queried in parallel. The approach underlying our construction is to enforce a canonical memory representation on cuckoo hashing. That is, up to the initial randomness, each set of elements has a unique memory representation.

A preliminary version of this work appeared in *Proceedings of the 35th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 631-642, 2008.

*Incumbent of the Judith Kleeman Professorial Chair, Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot 76100, Israel. Email: `moni.naor@weizmann.ac.il`. Research supported in part by a grant from the Israel Science Foundation.

†Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot 76100, Israel. Email: `gil.segev@weizmann.ac.il`.

‡Most of the work was done at Microsoft Research, Silicon Valley Campus.

§Microsoft Research, Silicon Valley Campus, 1065 La Avenida, Mountain View, CA 94043. Email: `uwieder@microsoft.com`.

1 Introduction

Over the past decade an additional aspect in the design of data structures has emerged due to security and privacy considerations: a data structure may give away much more information than it was intended to. Computer folklore is rich with tales of such cases, for example, files containing information whose creators assumed had been erased, only to be revealed later in embarrassing circumstances[1].

When designing a data structure whose internal representation may be revealed, a highly desirable goal is to ensure that an adversary will not be able to infer information that is not available through the legitimate interface. Informally, a data structure is *history independent* if its memory representation does not reveal any information about the sequence of operations that led to its current content, other than the content itself.

In this paper we design a practical history-independent data structure. We focus on the *dictionary* data structure, which is used for maintaining a set under insertions and deletions of elements, while supporting membership queries. Our construction is inspired by the highly practical cuckoo hashing, introduced by Pagh and Rodler [25], and guarantees history independence while enjoying most of the attractive features of cuckoo hashing. In what follows we briefly discuss the notion of history independence and several of its applications, and the main properties of cuckoo hashing.

Notions of history independence. Naor and Teague [23], following Micciancio [18], formalized two notions of history independence: a data structure is weakly history independent if any two sequences of operations that lead to the same content induce the same distribution on the memory representation. This notion assumes that the adversary gaining control is a one-time event, but in fact, in many realistic scenarios the adversary may obtain the memory representation at several points in time. A data structure is strongly history independent if for any two sequences of operations, the distributions of the memory representation at all time-points that yield the same content are identical. Our constructions in this paper are strongly history independent. An alternative characterization of strong history independence was provided by Hartline et al. [14]. Roughly speaking, they showed that strong history independence is equivalent to having a canonical representation up to the choice of initial randomness. More formal definitions of the two notions are provided in Section 2.

Applications of history-independent data structures. History independent data structures were naturally introduced in a cryptographic setting. Micciancio showed that *oblivious trees*[2] can be used to guarantee privacy in the incremental signature scheme of Bellare, Goldreich and Goldwasser [2, 3]. An incremental signature scheme is private if the signatures it outputs do not give any information on the sequence of edit operations that have been applied to produce the final document.

An additional cryptographic application includes, for example, designing vote storage mechanisms (see [4, 21, 22]). As the order in which votes are cast is public, a vote storage mechanism must be history independent in order to guarantee the privacy of the election process.

History independent data structures are valuable beyond the cryptographic setting as well. Consider, for example, the task of reconciling two dynamic sets. We consider two parties each of which receives a sequence of insert and delete operations, and their goal is to determine the elements in the symmetric difference between their sets. Now, suppose that each party processes

[1]See [5] for some amusing anecdotes of this nature.
[2]These are trees whose shape does not leak information.

its sequence of operations using a data structure in which each set of elements has a canonical representation. Moreover, suppose that the update operations are efficient and change only a very small fraction of the memory representation. In such a case, if the size of the symmetric difference is rather small, the memory representations of the data structures will be rather close, and this can enable an efficient reconciliation algorithm.

Cuckoo hashing. Pagh and Rodler [25] constructed an efficient hashing scheme, referred to as cuckoo hashing. It provides worst case constant lookup time, expected amortized constant update time, and uses roughly $2n$ words for storing n elements. Additional attractive features of cuckoo hashing are that no dynamic memory allocation is performed (except for when the tables have to be resized), and the lookup procedure queries only two memory entries which are independent and can be queried in parallel. These properties offer significant improvements compared to other hashing schemes, and experiments have shown that cuckoo hashing and its variants are highly suitable for modern computer architectures and distributed settings. Cuckoo hashing was found competitive with the best known dictionaries having an average case (but no non-trivial worst case) guarantee on lookup time (see, for example, [10, 25, 27, 29]).

1.1 Related Work

Micciancio [18] formalized the problem of designing oblivious data structures. He considered a rather weak notion of history independence, and devised a variant of 2–3 trees whose shape does not leak information. This notion was strengthened by Naor and Teague [23] to consider data structures whose memory representation does not leak information. Their main contributions are two history-independent data structures. The first is strongly history independent, and supports only insertions and membership queries which are performed in expected amortized constant time. Roughly speaking, the data structure includes a logarithmic number of pair-wise independent hash functions, which determine a probe sequence for each element. Whenever a new element is inserted and the next entry in its probe sequence is already occupied, a "priority function" is used to determine which element will be stored in this entry and which element will be moved to the next entry in its probe sequence. The second data structure is a weakly history-independent data structure supporting insertions, deletions and membership queries. Insertions and deletions are performed in expected amortized constant time, and membership queries in worst case constant time. Roughly speaking, this data structure is a history independent variant of the perfect hash table of Fredman, Komlós and Szemerédi [12] and its dynamic extension due to Dietzfelbinger et al. [7].

Buchbinder and Petrank [6] provided a separation between the two notions of history independence for comparison based algorithms. They established lower bounds for obtaining strong history independence for a large class of data structures, including the heap and the queue data structures. They also demonstrated that the heap and queue data structures can be made weakly history independent without incurring any additional (asymptotic) cost.

Blelloch and Golovin [5] constructed two strongly history-independent data structures based on linear probing. Their first construction supports insertions, deletions and membership queries in expected constant time. This essentially extends the construction of Naor and Teague [23] that did not support deletions. While the running time in the worst case may be large, the expected update time and lookup time is tied to that of linear probing and thus is $O(1/(1-\alpha)^3)$ where α is the memory utilization of the data structure (i.e., the ratio between the number of items and the number of slots). Their second construction supports membership queries in worst case constant time while maintaining an expected constant time bound on insertions and deletions. However, the memory utilization of their second construction is only about 9%. In addition, it deploys a two-level encoding, which may involve hidden constant factors that affect the practicality of the scheme. Furthermore, the worst case guarantees rely on an exponential amount of randomness and serves as a basis for a different hash table with more relaxed guarantees. The goal of our work is to design a hash table with better memory utilization and smaller hidden constants in the running time, even in the worst case.

1.2 Our Contributions

We construct an efficient and practical history-independent data structure that supports insertions, deletions, and membership queries. Our construction is based on cuckoo hashing, and shares most of its properties. Our construction provides the following performance guarantees (where the probability is taken only over the randomness used during the initialization phase of the data structure):

1. Insertions and deletions are performed in expected amortized constant time. Moreover, with high probability, insertion and deletions are performed in time $O(\log n)$ in the worst case.

2. Membership queries are performed in worst case constant time. Moveover, with high probability, the lookup procedure queries only two memory entries which are independent and can be queried in parallel.

3. The memory utilization of the data structure is roughly 50% when supporting only insertions and membership queries. When supporting deletions the data structure allocates an additional pointer for each entry. Thus, the memory utilization in this case is roughly 25%, under the conservative assumption that the size of a pointer is not larger than that of a key.

We obtain the same bounds as the second construction of Blelloch and Golovin [5] (see Section 1.1). The main advantages of our construction are its simplicity and practicality: membership queries would mostly require only two independent memory probes, and updates are performed in a way which is almost similar to cuckoo hashing and thus is very fast. A major advantage of our scheme is that it *does not use rehashing*. Rehashing is a mechanism for dealing with a badly behaved hash function by choosing a new one; using such a strategy for a strongly history-independent data structure is highly problematic (see Section 1.3). Furthermore, our data structure enjoys a better memory utilization, even when supporting deletions. We expect that in any practical scenario, whenever cuckoo hashing is preferred over linear probing, our construction should be preferred over those of Blelloch and Golovin.

1.3 Overview of the Construction

In order to describe our construction we first provide a high-level overview of cuckoo hashing. Then, we discuss our approach in constructing history-independent data structures based on the underlying properties of cuckoo hashing.

Cuckoo hashing. Cuckoo hashing uses two tables T_0 and T_1, each consisting of $r \geq (1+\epsilon)n$ words for some constant $\epsilon > 0$, and two hash functions $h_0, h_1 : \mathcal{U} \to \{0, \ldots, r-1\}$. An element $x \in \mathcal{U}$ is stored either in entry $h_0(x)$ of table T_0 or in entry $h_1(x)$ of table T_1, but never in both. The lookup procedure is straightforward: when given an element $x \in \mathcal{U}$, query the two possible memory entries in which x may be stored. The deletion procedure deletes x from the entry in which it is stored. As for insertions, Pagh and Rodler [25] demonstrated that the "cuckoo approach", kicking other elements away until every element has its own "nest", leads to a highly efficient insertion procedure when the functions h_0 and h_1 are assumed to sample an element in $[r]$ uniformly and independently. More specifically, in order to insert an element $x \in \mathcal{U}$ we first query entry $T_0[h_0(x)]$. If this entry is not occupied, we store x in that entry. Otherwise, we store x at that entry anyway, thus making the previous occupant "nestless". This element is then inserted to T_1 in the same manner, and so forth iteratively. We refer the reader to [25] for a more comprehensive description of cuckoo hashing.

Our approach. Cuckoo hashing is not history independent. The table in which an element is stored depends upon the elements inserted previously. Our approach is to enforce a canonical memory representation on cuckoo hashing. That is, up to the initial choice of the two hash functions, each set of elements has only one possible representation. As in cuckoo hashing, our construction uses two hash tables T_0 and T_1, each consisting of $r \geq (1+\epsilon)n$ entries for some constant $\epsilon > 0$, and two hash functions $h_0, h_1 : \mathcal{U} \to \{0, \dots, r-1\}$. An element $x \in \mathcal{U}$ is stored either in cell $h_0(x)$ of table T_0 or in cell $h_1(x)$ of table T_1.

Definition 1.1. *Given a set $S \subseteq \mathcal{U}$ and two hash functions h_0 and h_1, the* cuckoo graph *is the bipartite graph $G = (L, R, E)$ where $L = R = \{0, \dots, r-1\}$, and $E = \{(h_0(x), h_1(x)) : x \in S\}$.*

The cuckoo graph plays a central role in our analysis. It is easy to see that a set S can be successfully stored using the hash functions h_0 and h_1 if and only if no connected component in G has more edges then nodes. In other words, every component contains at most one cycle (i.e., unicyclic). The analysis of the insertion and deletion procedures are based on bounds on the size of a connected component. The following lemma is well known in random graph theory (see, for example, [15, Section 5.2]):

Lemma 1.2. *Assume $r \geq (1+\epsilon)n$ and the two hash functions are truly random. Let v be some node and denote by C the connected component of v. Then there exists some constant $\beta = \beta(\epsilon) \in (0,1)$ such that for any integer $k > 0$ it holds that $\Pr[|C| > k] \leq \beta^k$.*

In particular, Lemma 1.2 implies that the expected size of each component is constant, and with high probability it is $O(\log n)$, a fact which lies at the heart of the efficiency analysis.

In order to describe the canonical representation that our construction enforces it is sufficient to describe the canonical representation of each connected component in the graph. Let C be a connected component, and denote by S be the set of elements that are mapped to C. In case C is acyclic, we enforce the following canonical representation: the minimal element in S (according to some fixed ordering of $\mathcal{U}$) is stored in *both tables*, and this yields only one possible way of storing the remaining elements. In case C is unicyclic, we enforce the following canonical representation: the minimal element *on the cycle* is stored in table T_0, and this yields only one possible way of storing the remaining elements. The most challenging aspect of our work is dealing with the unlikely event in which a connected component contains more than one cycle.

Rehashing and history independence. It is known [17] that even if h_0 and h_1 are completely random functions, with probability $\Omega(1/n)$ there will be a connected component with more than one cycle. In this case the given set cannot be stored using h_0 and h_1. The standard solution for this scenario is to choose new functions and rehash the entire data. In the setting of strongly history-independent data structures, however, rehashing is particular problematic and affects the practical performance of the data structure. Consider, for example, a scenario in which a set is stored using h_0 and h_1, but when inserting an additional element x it is required to choose new hash functions h'_0 and h'_1, and rehash the entire data. If the new element x is now deleted, then in order to maintain history independence we must "roll back" to the previous hash functions h_0 and h_1, and once again rehash the entire data. This has two undesirable properties: First, when rehashing we cannot erase the description of any previous pair of hash functions, as we may be forced to roll back to this pair later on. When dealing with strongly history-independent data structures, a canonical representation for each set of elements must be determined at the initialization phase of the data structure. Therefore, all the hash functions must be chosen in advance, and this may lead to a high storage overhead (as is the case in [5]). Secondly, if an element that causes a rehash is inserted and deleted multiple times, each time an entire rehash must be performed.

Avoiding rehashing by stashing elements. Kirsch et al. [16] suggested a practical augmentation to cuckoo hashing in order to avoid rehashing: exploiting a secondary data structure for storing elements that create cycles, starting from the second cycle of each component. That is, whenever an element is inserted to a unicyclic component and creates an additional cycle in this component, the element is *stashed* in the secondary data structure. In our case, the choice of the stashed element must be history independent in order to guarantee that the whole data structure is history independent. Kirsch et al. prove the following bound on the number of stashed elements in the secondary data structure:

Lemma 1.3 ([16]). *Assume $r \geq (1+\epsilon)n$ and the two hash functions are truly random. The probability that the secondary data structure has more than s elements is $O(r^{-s})$.*

The secondary data structure in our construction can be any strongly history-independent data structure (such as a sorted list). This approach essentially reduces the task of storing n elements in a history-independent manner to that of storing only a few elements in a history-independent manner. In addition, it enables us to avoid rehashing and to increase the practicality of our scheme.

1.4 Paper Organization

The remainder of this paper is organized as follows. In Section 2 we overview the notion of history independence. In Section 3 we describe our data structure. In Section 4 we propose several possible instantiations for the secondary data structure used in our construction. In Section 5 we analyze the efficiency of our construction, and in Section 6 we provide several concluding remarks. Appendix A provides a formal proof of history independence.

2 Preliminaries

In this section we formally define the notions of weak and strong history independence. Our presentation mostly follows that of Naor and Teague [23]. A data structure is defined by a list of operations. We say that two sequences of operations, S_1 and S_2, yield the same content if for all suffixes T, the results returned by T when the prefix is S_1 are identical to those returned by T when the prefix is S_2.

Definition 2.1 (Weak History Independence). *A data structure implementation is* weakly history independent *if any two sequences of operations that yield the same content induce the same distribution on the memory representation.*

We consider a stronger notion of history independence that deals with cases in which an adversary may obtain the memory representation at several points in time. In this case it is required that for any two sequences of operations, the distributions of the memory representation at all time-points that yield the same content are identical.

Definition 2.2 (Strong History Independence)**.** *Let S_1 and S_2 be sequences of operations, and let $P_1 = \{i_1^1, \ldots, i_\ell^1\}$ and $P_2 = \{i_1^2, \ldots, i_\ell^2\}$ be two lists such that for all $b \in \{1, 2\}$ and $1 \leq j \leq \ell$ it holds that $1 \leq i_j^b \leq |S_b|$, and the content of the data structure following the i_j^1 prefix of S_1 and the i_j^2 prefix of S_2 are identical. A data structure implementation is* strongly history independent *if for any such sequences the distributions of the memory representation at the points of P_1 and at the corresponding points of P_2 are identical.*

Note that Definition 2.2 implies, in particular, that any data structure in which the memory representation of each state is fully determined given the randomness used during the initialization phase is strongly history independent. Our construction in this paper enjoys such a canonical representation, and hence is strongly history independent.

3 The Data Structure

Our data structure uses two tables T_0 and T_1, and a secondary data structure. Each table consists of $r \geq (1 + \epsilon)n$ entries for some constant $\epsilon > 0$. In the insert-only variant each entry stores at most one element. In the variant which supports deletions each entry stores at most one element and a pointer to another element. The secondary data structure can be chosen to be any strongly history-independent data structure (we refer the reader to Section 4 for several possible instantiations of the secondary data structure).

Elements are inserted into the data structure using two hash functions $h_0, h_1 : \mathcal{U} \to \{0, \ldots, r - 1\}$, which are independently chosen at the initialization phase. An element $x \in \mathcal{U}$ can be stored in three possible locations: entry $h_0(x)$ of table T_0, entry $h_1(x)$ of table T_1, or stashed in the secondary data structure. The lookup procedure is straightforward: when given an element $x \in \mathcal{U}$, query the two tables and perform a lookup in the secondary data structure.

In the remainder of this section we first describe the canonical representation of the data structure and some of its useful properties. Then, we describe the insertion and deletion procedures.

3.1 The Canonical Representation

As mentioned in Section 1.3, it is sufficient to consider a single connected component in the cuckoo graph. Let C be a connected component, and denote by S the set of elements that are mapped to C. When describing the canonical representation we distinguish between the following cases:

- C is a tree. In this case the minimal element in S is stored in both tables, and this yields only one possible way of storing the remaining elements.
- C is unicyclic. In this case the minimal element *on the cycle* is stored in table T_0, and this yields only one possible way of storing the remaining elements.
- C contains at least two cycles. In this case we iteratively put in the secondary data structure the largest element that lies in a cycle, until C contains only one cycle. The elements which remain in the component are arranged according to the previous case. We note that this case is rather unlikely, and occurs with only a polynomially small probability.

When supporting deletions each table entry includes additional space for one pointer. These pointers form a cyclic sorted list of the elements of the component (not including stashed elements). When deletions are not supported, there is no need to allocate or maintain the additional pointers.

Orientating the edges. When describing the insertion and deletion procedures it will be convenient to consider the cuckoo graph as a *directed* graph. Given an element x, we orient the edge so that x is stored at its tail. In other words, if x is stored in table T_b for some $b \in \{0,1\}$, we orient its corresponding edge in the graph from $T_b[h_b(x)]$ to $T_{1-b}[h_{1-b}(x)]$. An exception is made for the minimal element of an acyclic component, since such an element is stored in both tables. In such a case we orient the corresponding edge in both directions. The following claims state straightforward properties of the directed graph.

Claim 3.1. *Let $x_1 \to \cdots \to x_k$ be any directed path. Then, given the element x_1 it is possible to retrieve all the elements on this path using k probes to memory. Furthermore, if $x_{\min}$ is a minimal element in an acyclic component C, then for any element x stored in C there exists a directed path from x to $x_{\min}$.*

Proof. For every $1 \le i \le k$ denote by T_{b_i} the table in which x_i is stored. Given x_1, the definition of the graph and the orientation of its edges imply that x_2 is stored in entry $T_{1-b_1}[h_{1-b_1}(x_1)]$. We continue similarly and retrieve in increasing order each x_{i+1} which is stored in entry $T_{1-b_i}[h_{1-b_i}(x_i)]$. For the second part note that if C is acyclic, then by the first property of the canonical representation it must be a tree rooted at the minimal element. ■

Claim 3.2. *Let C be a unicyclic component, and let x^* be any element on its cycle. Then for any element x stored in C there exists a simple directed path from x to x^*.*

Proof. Note that the orientation of the edges guarantees that the cycle is oriented in a consistent manner. That is, the cycle is of the form $y_1 \to \cdots \to y_k \to y_1$. Therefore if x is on the cycle, then the claim follows. Now assume that x is not on the cycle. Let $x = x_1$ and denote by T_{b_1} the table in which x_1 is stored. Denote by x_2 the element stored in entry $T_{1-b_1}[h_{1-b_1}(x_1)]$. Note that $x_2 \neq x_1$ since x_1 is not on the cycle. If x_2 is on the cycle, then we are done. Otherwise we continue similarly, and for every i we denote by x_{i+1} the element stored in entry $T_{1-b_i}[h_{1-b_i}(x_i)]$. Since the component is unicycle, as long as x_i is not on the cycle then $x_{i+1} \notin \{x_1, \ldots, x_i\}$. Therefore we are guaranteed to reach the cycle at some point. ■

3.2 The Insertion Procedure

Given an element to insert x, the goal of the insertion procedure is to insert x while maintaining the canonical representation. Note that one only has to consider the representation of the connected component of the cuckoo graph in which x resides. Furthermore, Lemma 1.2 implies that the size of the component is $O(1)$ on expectation, thus an algorithm which is linear in the size of the component would have a constant expected running time. In the following we show that the canonical memory representation could be preserved without using the additional pointers. The additional pointers are only needed for supporting deletions. If the additional pointers are maintained, then once the element is inserted the pointers need to be updated so that the element is in its proper position in the cyclic linked list. This could be done in a straightforward manner in time linear in the size of the component.

Given an element $x \in \mathcal{U}$ there are four possible cases to consider. The first and simplest case is when both $T_0[h_0(x)]$ and $T_1[h_1(x)]$ are unoccupied, and we store x in both entries. The second and third cases are when one of the entries is occupied and the other is not occupied. In these cases x does not create a new cycle in the graph. Thus, unless x is the new minimal element in an acyclic component it is simply put in the empty slot. If x indeed is the new minimal element in an acyclic component, it is put in both tables and the appropriate elements are pushed to their alternative location, effectively removing the previous minimum element from one of the tables. The fourth case, in which both entries are occupied involves slightly more details, but is otherwise straightforward. In this case x either merges two connected components, or creates a new cycle in a component. The latter case may also trigger the low probability event of stashing an element in the secondary data structure.

In the following we provide a full description of the four cases. One can readily verify that the procedure preserves the canonical representation invariant. For the convenience of the reader we also supply a detailed explanation in Appendix A.1.

Case 1: $\boldsymbol{T_0[h_0(x)] = \perp}$ and $\boldsymbol{T_1[h_1(x)] = \perp}$. In this case x is in the only element of the component, and thus the minimal one. Thus, the canonical representation is maintained by storing x in both $T_0[h_0(x)]$ and $T_1[h_1(x)]$.

Case 2: $\boldsymbol{T_0[h_0(x)] \neq \perp}$ and $\boldsymbol{T_1[h_1(x)] = \perp}$. In this case x is added to an existing connected component, and since we add a new node to this component then x does not create a new cycle. Denote by x_1 the element stored in $T_0[h_0(x)]$, and denote by C its connected component. We begin by identifying whether C is acyclic or unicyclic. We follow the directed path starting at x_1 (as in Claim 3.1) either until we reach an element that appears twice (see Claim 3.1), or until we detect a cycle (see Claim 3.2). The canonical representation guarantees that in the first case the component is acyclic (and we denote by $x_{\min}$ the element that appears twice), and in the second case the component is unicyclic. There are three possible subcases to consider:

1. C is acyclic and $x > x_{\min}$. In this case $x_{\min}$ is the still the minimal element of the component, so we store x in $T_1[h_1(x)]$.
2. C is acyclic and $x < x_{\min}$. In this case x is the new minimal element of the component so it should be stored in both $T_0[h_0(x)]$ and $T_1[h_1(x)]$. In addition, all the elements along the path connecting x_1 and $x_{\min}$ are moved to their other possible location.

   More formally, we first store x in $T_1[h_1(x)]$, and denote by $x \to x_1 \to \cdots \to x_k \to x_{\min} \to x_{\min}$ the directed path connecting x to $x_{\min}$ (note that it is possible that $x_1 = x_{\min}$). For every $1 \leq i \leq k$ we move x_i to its other possible location (this deletes the first appearance of $x_{\min}$ on this path). That is, if x_i is stored in $T_b[h_b(x_i)]$ we move it to $T_{1-b}[h_{1-b}(x_i)]$. Finally, we store x in $T_0[h_0(x)]$.
3. C is unicyclic. In this case x does not lie on a cycle so we can safely store x in $T_1[h_1(x)]$.

Case 3: $\boldsymbol{T_0[h_0(x)] = \perp}$ and $\boldsymbol{T_1[h_1(x)] \neq \perp}$. This case is analogous to Case (2).

Case 4: $\boldsymbol{T_0[h_0(x)] \neq \perp}$ and $\boldsymbol{T_1[h_1(x)] \neq \perp}$. In this case, x either merges two connected components (each of which may be acyclic or unicyclic), or creates a new cycle in a connected component (which, again, may be acyclic or unicyclic). The canonical representation forces us to deal separately with each of these subcases. Roughly speaking, if the component created after x was inserted has at most one cycle then the secondary data structure need not be used and insertion procedure is simple. In the unlikely event of creating a second cycle, the procedure has to find the element to store in the secondary data structure. In the following we show in detail how to do it. It is a rather tedious but otherwise completely straightforward case analysis.

The insertion procedure begins by distinguishing between these subcases, as follows. For each $b \in \{0, 1\}$ denote by x_1^b the element stored in $T_b[h_b(x)]$, and denote by C^b its connected component.

We follow the directed path starting at x_1^b to identify whether C^b is acyclic or unicyclic (as in Case 2). If C^b is acyclic the path discovers the minimal element x_{min}^b in the component (note that $x_{\mathrm{min}}^0 = x_{\mathrm{min}}^1$ if and only if $C^0 = C^1$), and if C^b is unicyclic then the path allows us to discover the maximal element x_{max}^b on the cycle. The three most likely cases are the following:

1. $C^0 \neq C^1$ and both C^0 and C^1 are acyclic. In this case we merge the two components to a single acyclic component. For each $b \in \{0,1\}$ denote by $x_1^b \to \cdots \to x_{k_b}^b \to x_{\mathrm{min}}^b \to x_{\mathrm{min}}^b$ the directed path connecting x_1^b to x_{min}^b. There are two possibilities to consider:

   (a) $x = \min\{x_{\mathrm{min}}^0, x_{\mathrm{min}}^1, x\}$. In this case, all the elements along the two paths are moved to their other possible location, and x is stored in both tables. More specifically, for every $b \in \{0,1\}$ and for every $1 \leq i \leq k_b$ we move x_i^b to its other possible location, and store x in both $T_0[h_0(x)]$ and $T_1[h_1(x)]$.

   (b) $x_{\mathrm{min}}^b = \min\{x_{\mathrm{min}}^0, x_{\mathrm{min}}^1, x\}$ for some $b \in \{0,1\}$. In this case only the elements along on the path connecting x_1^{1-b} and x_{min}^{1-b} are moved, and x is stored in $T_{1-b}[h_{1-b}(x)]$. More specifically, for every $1 \leq i \leq k_{1-b}$ we move x_i^{1-b} to its other possible location, and store x in $T_{1-b}[h_{1-b}(x)]$.

2. $C^0 \neq C^1$ and exactly one of C^0 and C^1 is acyclic. Let $b \in \{0,1\}$ be such that C^{1-b} is acyclic and C^b is unicyclic. Denote by $x_1^{1-b} \to \cdots \to x_{k_{1-b}}^{1-b} \to x_{\mathrm{min}}^{1-b} \to x_{\mathrm{min}}^{1-b}$ the directed path connecting x_1^{1-b} to x_{min}^{1-b}. For every $1 \leq i \leq k_{1-b}$ we move x_i^{1-b} to its other possible location, and store x in $T_{1-b}[h_{1-b}(x)]$.

3. $C^0 = C^1$ and the component is acyclic. In this case x creates the first cycle in the component. Denote by x_{min} the minimal element in the component (i.e., $x_{\mathrm{min}} = x_{\mathrm{min}}^0 = x_{\mathrm{min}}^1$), and for each $b \in \{0,1\}$ denote by $x_1^b \to \cdots \to x_{k_b}^b \to x_{\mathrm{min}} \to x_{\mathrm{min}}$ the path connecting x_1^b to x_{min}. There are two cases to consider:

   (a) The path connecting x_1^0 and x_1^1 contains the two appearances of x_{min}. That is, the path is of the form $x_1^0 \to \cdots \to x_{k_0}^0 \to x_{\mathrm{min}} \leftrightarrow x_{\mathrm{min}} \leftarrow x_{k_1}^1 \leftarrow \cdots \leftarrow x_1^1$. In this case the minimal element on the cycle is x_{min}, and therefore it should now be stored only in T_0. Let $b \in \{0,1\}$ be such that $x_{k_b}^b$ is adjacent to $T_1[h_1(x_{\mathrm{min}})]$. For every $1 \leq i \leq k_b$ we move x_i^b to its other possible location, and store x in $T_b[h_b(x)]$.

   (b) The path connecting x_1^0 and x_1^1 contains at most a single appearance of x_{min}. That is, the path is of the form $x_1^0 \to \cdots \to x_{\ell_0}^0 \to x^* \leftarrow x_{\ell_1}^1 \leftarrow \cdots \leftarrow x_1^1$, for some $\ell_0 \leq k_0$ and $\ell_1 \leq k_1$, where x^* is the first intersection point of the two paths. In this case we denote by $x^* = y_1 \to \cdots \to y_{k^*} \to x_{\mathrm{min}} \to x_{\mathrm{min}}$ the directed path connecting x^* and x_{min}. First, for every $1 \leq i \leq k^*$ in decreasing order we move y_i to its other possible location. Then, we let $z = \min\{x, x_1^0, \ldots, x_{\ell_0}^0, x_1^1, \ldots, x_{\ell_1}^1\}$ (this is the minimal element on the cycle, which should be stored in table T_0) and distinguish between the following two cases:

      i. $z = x$. For every $1 \leq i \leq \ell_0$ we move x_i^0 to its other possible location, and store x in $T_0[h_0(x)]$.

      ii. $z = x_j^b$ for some $b \in \{0,1\}$ and $1 \leq j \leq \ell_b$. If x_j^b is currently stored in T_0, then for every $1 \leq i \leq \ell_{1-b}$ we move x_i^{1-b} to its other possible location, and store x in $T_{1-b}[h_{1-b}(x)]$. Otherwise (x_j^b is currently stored in T_1), for every $1 \leq i \leq \ell_b$ we move x_i^b to its other possible location, and store x in $T_b[h_b(x)]$.

In the unlikely event in which an element has to be put in the secondary data structure we do as follows:

1. $C^0 \neq C^1$ and both C^0 and C^1 are unicyclic. Recall that we identified $x^0_{\max}$ and $x^1_{\max}$ – the maximal elements on the cycles of C^0 and C^1, respectively. Let $b \in \{0,1\}$ be such that $x^{1-b}_{\max} < x^b_{\max}$. Denote by $x^b_1 \to \cdots \to x^b_k \to x^b_{\max}$ the simple directed path connecting x^b_1 to $x^b_{\max}$. For every $1 \leq i \leq k$ we move x^b_i to its other possible location, stash $x^b_{\max}$ in the secondary data structure, and store x in $T_b[h_b(x)]$.

2. $C^0 = C^1$ and the component is unicyclic. In this case the connected component already contains a cycle, and x creates an additional cycle. We denote the existing cycle by $y_1 \to \cdots \to y_k \to y_1$, and denote the paths connecting x^0_1 and x^1_1 to the existing cycle by $x^0_1 \to \cdots \to x^0_{k_0} \to y_1$ and by $x^1_1 \to \cdots \to x^1_{k_1} \to y_\ell$ (for some $1 \leq \ell \leq k$), respectively. Note that we assume without loss of generality that the existing cycle is directed from y_1 to y_ℓ, and note that it is possible that $x^0_1 = y_1$ or $x^1_1 = y_\ell$. There are two possibilities:

   (a) $y_1 \neq y_\ell$. In this case the paths connecting x^0_1 and x^1_1 to the existing cycle do not intersect. Let $x_{\max} = \max\{x, x^0_1, \ldots, x^0_{k_0}, x^1_1, \ldots, x^1_{k_1}, y_1, \ldots, y_k\}$. We stash $x_{\max}$ in the secondary data structure, and distinguish between the following cases:

      i. $x_{\max} = x$. In this case the insertion procedure terminates.

      ii. $x_{\max} = x^b_j$ for some $b \in \{0,1\}$ and $1 \leq j \leq k_b$. For every $1 \leq i \leq j-1$ we move x^b_i to its other possible location, and store x in $T_b[h_b(x)]$.

      iii. $x_{\max} = y_j$ for some $1 \leq j \leq \ell - 1$. For every $1 \leq i \leq j-1$ we move y_i to its other possible location (this clears the entry in which y_1 was stored). Let $x'_{\min} = \min\{x, x^0_1, \ldots, x^0_{k_0}, x^0_1, \ldots, x^1_{k_1}, y_\ell, \ldots, y_k\}$ (this is the minimal element on the new cycle, and it should be stored in T_0). We distinguish between three cases:

         - $x'_{\min} = x$. For every $1 \leq i \leq k_0$ we move x^0_i, and store x in $T_0[h_0(x)]$.
         - $x'_{\min} = x^0_t$ for some $1 \leq t \leq k_0$. If $x'_{\min}$ is currently stored in T_0, then for every $\ell \leq i \leq k$ we move y_i to its other possible location, for every $1 \leq i \leq k_1$ we move x^1_i to its other possible location, and store x in $T_1[h_1(x)]$. Otherwise ($x'_{\min}$ is currently stored in T_1), the for every $1 \leq i \leq k_0$ we move x^0_i, and store x in $T_0[h_0(x)]$.
         - $x'_{\min} = x^1_t$ for some $1 \leq t \leq k_1$, or $x'_{\min} = y_t$ for some $\ell \leq t \leq k$. If $x'_{\min}$ is currently stored in T_0, then for every $1 \leq i \leq k_0$ we move x^0_i, and store x in $T_0[h_0(x)]$. Otherwise ($x'_{\min}$ is currently stored in T_1), then for every $\ell \leq i \leq k$ we move y_i to its other possible location, for every $1 \leq i \leq k_1$ we move x^1_i to its other possible location, and store x in $T_1[h_1(x)]$.

      iv. $x_{\max} = y_j$ for some $\ell \leq j \leq k$. For every $\ell \leq i \leq j-1$ we move y_i to its other possible location (this clear the entry in which y_ℓ was stored). Let $x'_{\min} = \min\{x, x^0_1, \ldots, x^0_{k_0}, x^0_1, \ldots, x^1_{k_1}, y_1, \ldots, y_{\ell-1}\}$ (this is the minimal element on the new cycle, and it should be stored in T_0). We distinguish between three cases:

         - $x'_{\min} = x$. For every $1 \leq i \leq \ell - 1$ we move y_i to its other possible location, for every $1 \leq i \leq k_0$ we move x^0_i to its other possible location, and store x in $T_0[h_0(x)]$.
         - $x'_{\min} = x^1_t$ for some $1 \leq t \leq k_1$. If $x'_{\min}$ is currently stored in T_0, then for every $1 \leq i \leq \ell - 1$ we move y_i to its other possible location, for every $1 \leq i \leq k_0$ we move x^0_i to its other possible location, and store x in $T_0[h_0(x)]$. Otherwise ($x'_{\min}$

is currently stored in T_1), then for every $1 \le i \le k_1$ we move x_i^1 to its other possible location, and store x in $T_1[h_1(x)]$.

- $x'_{\min} = x_t^0$ for some $1 \le t \le k_0$, or $x'_{\min} = y_t$ for some $1 \le t \le \ell - 1$. If $x'_{\min}$ is currently stored in T_0, then for every $1 \le i \le k_1$ we move x_i^1 to its other possible location, and store x in $T_1[h_1(x)]$. Otherwise ($x'_{\min}$ is currently stored in T_1), then for every $1 \le i \le \ell - 1$ we move y_i to its other possible location, for every $1 \le i \le k_0$ we move x_i^0 to its other possible location, and store x in $T_0[h_0(x)]$.

(b) $y_1 = y_\ell$. In this case the paths connecting x_1^0 and x_1^1 to the existing cycle intersect. Denote by x^* their first intersection point. Note that either $x^* = y_1$ or that $x^* = x_{\ell_0}^0 = x_{\ell_1}^1$ for some $\ell_0 \le k_0$ and $\ell_1 \le k_1$. Let $x_{\max} = \max\{x, x_1^0, \ldots, x_{\ell_0-1}^0, x_1^1, \ldots, x_{\ell_1-1}^1, y_1, \ldots, y_k\}$. We stash $x_{\max}$ in the secondary data structure, and distinguish between the following cases:

i. $x_{\max} = x$. In this case the insertion procedure terminates.

ii. $x_{\max} = x_j^b$ for some $b \in \{0, 1\}$ and $1 \le j \le \ell_b - 1$. For every $1 \le i \le j - 1$ we move x_i^b to its other possible location, and store x in $T_b[h_b(x)]$.

iii. $x_{\max} = y_j$ for some $1 \le j \le k$. Denote by $x^* = x_1^* \to \cdots \to x_{k^*}^* \to y_1$ the directed path connecting x^* to y_1. For every $1 \le i \le j - 1$ we move y_i to its other possible location, and for every $1 \le i \le k^*$ we move x_i^* to its other possible location. Now, let $x'_{\min} = \min\{x, x_1^0, \ldots, x_{\ell_0-1}^0, x_1^1, \ldots, x_{\ell_1-1}^1\}$ (this is the minimal element on the new cycle, and it should be stored in T_0). We distinguish between two cases:

- $x'_{\min} = x$. For every $1 \le i \le \ell_0 - 1$ we move x_i^0 to its other possible location, and store x in $T_0[h_0(x)]$.
- $x'_{\min} = x_j^b$ for some $b \in \{0, 1\}$ and $1 \le j \le \ell_b - 1$. If $x'_{\min}$ is currently stored in T_0, then for every $1 \le i \le \ell_{1-b}$ we move x_i^{1-b} to its other possible location, and store x in $T_{1-b}[h_{1-b}(x)]$. Otherwise ($x'_{\min}$ is currently stored in T_1), then for every $1 \le i \le \ell_b - 1$ we move x_i^b to its other possible location, and store x in $T_b[h_b(x)]$.

3.3 The Deletion Procedure

The deletion procedure takes advantage of the additional pointer stored in each entry. Recall that these pointers form a cyclic list of all the elements of a connected component. Note that since the expected size of a connected component is constant, and the expected size of the secondary data structure is constant as well, a straightforward way of deleting an element is to retrieve all the elements in its connected component, reinsert them without the deleted element, and then reinsert all the elements that are stashed in the secondary data structure. This would result in expected amortized constant deletion time. In practice, however, it is desirable to minimize the amount of memory manipulations. In what follows we detail a more refined procedure, which although share the same asymptotic performance, it is much more sensible in practice.

Given an element $x \in \mathcal{U}$ we execute the lookup procedure to verify that x is indeed stored. If x is stashed in the secondary data structure, we just delete x from the secondary data structure. In the more interesting case, where x is stored in one of the tables, the connected component C of x is either acyclic or unicyclic. As in the insertion procedure, we follow the directed path starting at $T_0[h_0(x)]$ either until we reach an element that appears twice, or until we detect a cycle. The canonical representation guarantees that in the first case the component is acyclic (and the element that appears twice is $x_{\min}$), and in the second case the component is unicyclic (and we are able to identify whether x is part of its cycle). There are four possible cases to consider:

Case 1: C is acyclic and $x \neq x_{\min}$. In this case we split C to two acyclic connected components C^0 and C^1. An element belongs to C^0 if the directed path connecting it to $x_{\min}$ does not go through x. All the other elements belong to C^1.

Note that $x_{\min}$ is the minimal element in C^0, and is already stored in both tables. We identify the minimal element in C^1 as follows. We follow the linked list of elements starting from $x_{\min}$, and denote by $x^1_{\min}$ the first element on that list which is different from x, and for which x belongs to the directed path connecting it to $x_{\min}$ (it is possible that there is no such element, and in this case we delete x and update the linked list). This process discovers the directed path connecting $x^1_{\min}$ to x, denoted $x^1_{\min} \to x_1 \to \cdots \to x_k \to x$. For every $1 \leq i \leq k$ we move x_i to its other possible location (note that this deletes x), and store $x^1_{\min}$ in both tables. Finally, we update the linked lists of the two components.

Case 2: C is acyclic and $x = x_{\min}$. In this case we again split C to two acyclic connected components C^0 and C^1. The element x is the minimal element in an acyclic component, and therefore it is stored in both tables. We denote its two appearances by x^0 and x^1. An element belongs to C^0 if the directed path connecting it to x^0 does not go through x^1. All the other elements belong to C^1. That is, an element belongs to C^1 if the directed path connecting it to x^1 does not go through x^0.

For each $b \in \{0, 1\}$ we continue as follows. We follow the linked list of elements starting from x^b, and denote by $x^b_{\min}$ the first element on that list which belongs to C^b (it is possible that there is no such element, and in this case we delete x^b and update the linked list). This process discovers the directed path connecting $x^b_{\min}$ to x^b, denoted $x^b_{\min} \to x^b_1 \to \cdots \to x^b_{k_b} \to x^b$. For every $1 \leq i \leq k_b$ we move x^b_i to its other possible location (note that this deletes x^b), and store $x^b_{\min}$ in both tables. Finally, we update the linked lists of the two components.

Case 3: C is unicyclic and x is part of its cycle. In this case C remains connected when x is deleted. In addition, if there are any stashed elements that belong to C, the minimal such element should be removed from the secondary data structure and inserted using the insertion procedure.

More specifically, by following the linked list starting from x we locate the minimal element $x_{\min}$ currently stored in C (if x is the minimal element than we denote by $x_{\min}$ the first element larger than x). Claim 3.2 guarantees that there is a directed path connecting $x_{\min}$ to x. Denote this path by $x_{\min} \to x_1 \to \cdots \to x_k \to x$. For every $1 \leq i \leq k$ we move x_i to its other possible location (note that this deletes x), and store $x_{\min}$ in both tables. We then update the linked list of the component. Finally, if there are stashed elements that belong to this component, we execute the insertion procedure with the minimal such element.

Case 4: C is unicyclic and x is not part of its cycle. In this case we split C to an acyclic component C^0 and a unicyclic component C^1. An element belongs to C^0 if the directed path connecting it to the cycle goes through x. All the other elements belong to C^1.

We identify the minimal element in C^0 as follows. We follow the linked list of elements starting from x, and denote by $x^0_{\min}$ the first element on that list which is different from x, and for which x belongs to the directed path connecting it to the cycle (it is possible that there is no such element, and in this case we delete x and update the linked list). This process discovers the directed path connecting $x^0_{\min}$ to x, denoted $x^0_{\min} \to x_1 \to \cdots \to x_k \to x$. For every $1 \leq i \leq k$ we move x_i to its other possible location (note that this deletes x), and store $x^0_{\min}$ in both tables. We then update the linked lists of the two components. Finally, if there are stashed elements that belong to C^0, we execute the insertion procedure with the minimal such element.

4 The Secondary Data Structure

In this section we propose several possible instantiations for the secondary data structure. As discussed in Section 1.3, the secondary data structure can be any strongly history-independent data structure. Recall that Lemma 1.3 implies in particular that the expected number of stashed elements is constant, and with overwhelming probability there are no more than $\log n$ stashed elements. Thus, the secondary data structure is essentially required to store only a very small number of elements. Furthermore, since the secondary data structure is probed every time a lookup is performed, it is likely to reside most of the time in the cache, and thus impose a minimal cost.

The practical choice. The most practical approach is instantiating the secondary data structure with a sorted list. A sorted list is probably the simplest data structure which is strongly history independent. When a sorted list contains at most s elements, insertions and deletions are performed in time $O(s)$ in the worst case, and lookups are performed in time $O(\log s)$ in the worst case. In turn, instantiated with a sorted list, our data structure supports insertions, deletions, and membership queries in expected constant time. Moreover, Lemma 1.3 implies that the probability that a lookup requires more than k probes is at most $O(n^{-2^k})$.

Constant worst case lookup time. We now propose two instantiations that guarantee constant lookup time in the *worst case*. We note that these instantiations result in a rather theoretical impact, and in practice we expect a sorted list to perform much better.

One possibility is using the strongly history-independent data structure of Blelloch and Golovin [5], and in this case our data structure supports insertions and deletions in expected constant time, and membership queries in worst case constant time. Another possibility is using any deterministic perfect hash table with constant lookup time. On every insertion and deletion we reconstruct the hash table, and since its construction is deterministic, the resulting data structure is strongly history independent. The repeated reconstruction allows us to use a static hash table (instead of a dynamic hash table), and in this case the construction time of the table determines the insertion and deletion time. Perfect hash tables with such properties were suggested by Alon and Naor [1], Miltersen [19], and Hagerup, Miltersen and Pagh [13]. Asymptotically, the construction of Hagerup et al. is the most efficient one, and provides an $O(s \log s)$ construction time on s elements. Instantiated with their construction, our data structure supports insertions and deletion in expected constant time, and membership queries in worst case constant time.

5 Efficiency Analysis

In this section we examine the efficiency of our data structure, in terms of its update time, lookup time, and memory utilization. The lookup operation, as specified in Section 3, requires probing both tables and performing a lookup in the secondary data structure. As in cuckoo hashing, probing the tables requires only two memory accesses (which are independent and can be performed in parallel). The secondary data structure, as described in Section 4, can be chosen to support membership queries in constant time in the worst case. Furthermore, with high probability the secondary data structure contains only a very small number of elements, and can therefore reside in the cache. In this case the lookup time is dominated by the time required for probing the two tables. Moreover, we note that with high probability (see Lemma 1.3), the secondary storage does not contain any elements, and therefore the lookup procedure only probes the two tables. We conclude that the lookup time is basically similar to that of cuckoo hashing, shown in [25] to be very efficient.

The running time analysis of the insertion and deletion procedures is based on bounds on the size of a connected component and on the number of elements in the secondary data structure. Both the insertion and the deletion procedure involve at most two scans of the relevant connected component or the secondary data structure, and thus are linear in the size of the connected component and of the number of stashed elements. Lemmata 1.2 and 1.3 imply that the expected running time is a small constant.

The memory utilization of the our construction is identical to that of cuckoo hashing when supporting only insertions and membership queries – the number of elements should be less than half of the total number of table entries. The extra pointer needed for supporting deletions reduces the memory utilization to 25% under the conservative assumption that a pointer occupies the same amount of space as a key. If a pointer is actually shorter than a key, the space utilization improves accordingly.

6 Concluding Remarks

On using $O(\log n)$-wise independent hash functions. One possible drawback of our construction, from a purely theoretical point of view, is that we assume the availability of truly random hash functions, while the constructions of Blelloch and Golovin assume $O(\log n)$-wise independent hash functions (when guaranteeing worst case constant lookup time) or 5-wise independent hash functions (when guaranteeing expected constant lookup time). Nevertheless, simulations (see, for example, [25]) give a strong evidence that simple heuristics work for the choice of the hash functions as far as cuckoo hashing is concerned (Mitzenmacher and Vadhan [20] provide some theoretical justification). Thus we expect our scheme to be efficient in practice.

Our construction can be instantiated with $O(\log n)$-wise independent hash functions, and still provide the same performance guarantees for insertions, deletions, and membership queries. However, in this case the bound on the number of stashed elements is slightly weaker than that stated in Lemma 1.3. Nevertheless, rather standard probabilistic arguments can be applied to argue that (1) the expected number of stashed elements is constant, and (2) the expected size of a connected component in the cuckoo graph is constant.

Alternatively, our construction can be instantiated with the highly efficient hash functions of Dietzfelbinger and Woelfel [9] (improving the constructions of Siegel [28] and Ostlin and Pagh [24]). These hash functions are almost n^δ-wise independent with high probability (for some constant $0 < \delta < 1$), can be evaluated in constant time, and each function can be described using only $O(n)$ memory words. One possible drawback of this approach is that the distance to n^δ-independence is only polynomially small.

Memory utilization. Our construction achieves memory utilization of essentially 50% (as in cuckoo hashing), and of 25% when supporting deletions. More efficient variants of cuckoo hashing [8, 11, 26] circumvent the 50% barrier and achieve better memory utilization by either using more than two hash functions, or storing more than one element in each entry. It would be interesting to transform these variants to history-independent data structures while essentially preserving their efficiency.

References

[1] N. Alon and M. Naor. Derandomization, witnesses for Boolean matrix multiplication and construction of perfect hash functions. *Algorithmica*, 16(4-5):434–449, 1996.

[2] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography: The case of hashing and signing. In *Advances in Cryptology - CRYPTO '94*, pages 216–233, 1994.

[3] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography and application to virus protection. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 45–56, 1995.

[4] J. Bethencourt, D. Boneh, and B. Waters. Cryptographic methods for storing ballots on a voting machine. In *Proceedings of the 14th Network and Distributed System Security Symposium*, pages 209–222, 2007.

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

[6] N. Buchbinder and E. Petrank. Lower and upper bounds on obtaining history-independence. *Information and Computation*, 204(2):291–337, 2006.

[7] M. Dietzfelbinger, A. R. Karlin, K. Mehlhorn, F. M. auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

[8] M. Dietzfelbinger and C. Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. *Theoretical Computer Science*, 380(1-2):47–68, 2007.

[9] M. Dietzfelbinger and P. Woelfel. Almost random graphs with simple hash functions. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, pages 629–638, 2003.

[10] Ú. Erlingsson, M. Manasse, and F. McSherry. A cool and practical alternative to traditional hash tables. In *Proceedings of the 7th Workshop on Distributed Data and Structures*, 2006.

[11] D. Fotakis, R. Pagh, P. Sanders, and P. G. Spirakis. Space efficient hash tables with worst case constant access time. *Theory of Computing Systems*, 38(2):229–248, 2005.

[12] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[13] T. Hagerup, P. B. Miltersen, and R. Pagh. Deterministic dictionaries. *Journal of Algorithms*, 41(1):69–85, 2001.

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

[15] S. Janson, T. Łuczak, and A. Ruciński. *Random Graphs.* Wiley-Interscience, 2000.

[16] A. Kirsch, M. Mitzenmacher, and U. Wieder. More robust hashing: Cuckoo hashing with a stash. To appear in *Proceedings of the 16th Annual European Symposium on Algorithms*, 2008.

[17] R. Kutzelnigg. Bipartite random graphs and cuckoo hashing. In *Proceedings of the 4th Colloquium on Mathematics and Computer Science*, pages 403–406, 2006.

[18] D. Micciancio. Oblivious data structures: Applications to cryptography. In *Proceedings of the 29th Annual ACM Symposium on the Theory of Computing*, pages 456–464, 1997.

[19] P. B. Miltersen. Error correcting codes, perfect hashing circuits, and deterministic dynamic dictionaries. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 556–563, 1998.

[20] M. Mitzenmacher and S. Vadhan. Why simple hash functions work: Exploiting the entropy in a data stream. In *Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 746–755, 2008.

[21] 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 (extended abstract). In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 365–370, 2006. The full version is available from the Cryptology ePrint Archive, Report 2006/081.

[22] T. Moran, M. Naor, and G. Segev. Deterministic history-independent strategies for storing information on write-once memories. In *Proceedings of the 34th International Colloquium on Automata, Languages and Programming*, pages 303–315, 2007.

[23] M. Naor and V. Teague. Anti-persistence: History independent data structures. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing*, pages 492–501, 2001.

[24] A. Ostlin and R. Pagh. Uniform hashing in constant time and linear space. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, pages 622–628, 2003.

[25] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[26] R. Panigrahy. Efficient hashing with lookups in two memory accesses. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 830–839, 2005.

[27] K. A. Ross. Efficient hash probes on modern processors. In *Proceedings of the 23nd International Conference on Data Engineering*, pages 1297–1301, 2007.

[28] A. Siegel. On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 20–25, 1989.

[29] M. Zukowski, S. Héman, and P. A. Boncz. Architecture conscious hashing. In *2nd International Workshop on Data Management on New Hardware*, page 6, 2006.

A Proof of History Independence

In this section we prove that our data structure is history independent by showing that it has the canonical representation described in Section 3.1. We fix two hash functions $h_0, h_1 : \mathcal{U} \to \{0, \ldots, r-1\}$, and a sequence of insert and delete operations $\sigma_1, \ldots \sigma_m$. We prove by induction on $1 \leq i \leq k$ that after performing σ_i the data structure has the canonical representation. This essentially reduces to proving that if the data structure is in the canonical representation and an operation is performed, then the canonical representation is not violated. In addition, note that the insertion and deletion procedures affect only a single connected component, and therefore the analysis can focus on a single connected component. In what follows we deal separately with insertions and deletions.

A.1 History Independence of Insertions

As in the insertion procedure, when inserting an element $x \in \mathcal{U}$ there are four possible cases to consider.

Case 1: $T_0[h_0(x)] = \perp$ and $T_1[h_1(x)] = \perp$. In this case a new connected component is created. This component is acyclic, and x is its minimal element. We store x in both $T_0[h_0(x)]$ and $T_1[h_1(x)]$, and this corresponds to the canonical representation of the component.

Case 2: $T_0[h_0(x)] \neq \perp$ and $T_1[h_1(x)] = \perp$. In this case x is added to an existing connected component, and x does not create a new cycle in this component. Therefore, the set of stashed elements of the component does not change. If the component is acyclic and x is larger than the minimal element of the component, then x is stored in $T_1[h_1(x)]$. If the component is acyclic and x is smaller than the minimal element, then x is stored in both tables and there is only one possible way of storing all the other elements. Finally, if the component is unicyclic then x is stored in $T_1[h_1(x)]$. In addition, we did not move any elements that are on the cycle, and therefore the minimal element on the cycle is still stored in T_0. Thus, in all three cases the canonical representation is not violated.

Case 3: $T_0[h_0(x)] = \perp$ and $T_1[h_1(x)] \neq \perp$. See the previous case.

Case 4: $T_0[h_0(x)] \neq \perp$ and $T_1[h_1(x)] \neq \perp$. In this case x either merges two connected components, or create a new cycle in an existing component. There are five possible subcases to consider:

1. Merging two acyclic components. The new component is acyclic as well, and its minimal is either the minimal element of the first component, the minimal element of the second component, or x. The procedure identifies the minimal element, which is then stored in both tables, and there is only one possible way of storing all the other elements.

2. Merging an acyclic component with a unicyclic component. In this case x creates a single unicyclic component. Note that x is not on the cycle of this component, and therefore the set of stashed elements of the new component is the same as that of the unicyclic component. The procedure stores x in the acyclic component, and moves all elements on the directed path from x to the minimal element of the acyclic component to their other possible location. The result is that the minimal element of the acyclic component is now stored in only one of the tables. In addition, we did not move any elements on the cycle of the unicyclic component, and therefore the minimal element on the cycle is still stored in T_0.

3. Merging two unicyclic components. The set of stashed elements of the new component is the union of the corresponding sets of the two components, including the maximal element on the two cycles of the components (these are disjoint cycles, which do not contain x). The insertion procedure identifies the maximal element on the two cycles, and stash it in the secondary data structure. Then, x is inserted by moving all the elements on the directed path that leads to

the stashed element. Note that we did not move any elements on the remaining cycle, and therefore the minimal element on that cycle is still stored in T_0.

4. Creating a cycle in an acyclic component. In this case there are two directed paths connecting the two possible locations of x to the minimal element of the component. The procedure inserts x by moving the elements along the path that will cause the minimal element of the cycle to be stored in T_0.

5. Creating a cycle in a unicyclic component. The set of stashed elements now includes also the maximal element on the two cycles (i.e., either the maximal element on the existing cycle, or the maximal element on the cycle created by x). Once this element is stashed, x is inserted to the location that will cause the minimal element on the cycle to be stored in T_0.

A.2 History Independence of Deletions

As in the deletion procedure, given an element $x \in \mathcal{U}$ which is currently stored, there are several cases to consider. The simplest case is when x is stashed in the secondary data structure, and the procedure just deletes x. This does not affect the representation of the connected component to which x is mapped: the set of stashed elements of this component does not changes (other than removing x). Therefore, in this case the canonical representation is not violated.

We now turn to consider the cases in which x is stored in at least one of the tables. The connected component C in which x is stored is either acyclic or unicyclic. If C is acyclic, the canonical representation after deleting x depends on whether x is the minimal element $x_{\min}$ of the component. If C is unicyclic, the canonical representation after deleting x depends on whether x is part of its cycle. There are four possible cases to consider:

Case 1: C is acyclic and $x \neq x_{\min}$. The deletion of x removes the outgoing edge of x from the cuckoo graph, and this splits C to two acyclic connected components C^0 and C^1. The elements of C^0 are those who are not connected to x after the edge is removed (that is, all elements for which the directed path connecting them to $x_{\min}$ does not go through x). All the other elements belong to C^1.

Note that $x_{\min}$ is the minimal element in C^0, and is already stored in both tables. Therefore, C^0 is in its canonical form. The procedure identified the minimal element in C^1, stores it in both tables, and arranges the other elements of the component in the only possible way. This implies that also C^1 is in its canonical form.

Case 2: C is acyclic and $x = x_{\min}$. This case is almost identical to the previous case. The only difference is that the procedure needs to identify the minimal element in both C^0 and C^1 (and arrange the components accordingly).

Case 3: C is unicyclic and x is part of its cycle. When removing the outgoing edge of x, the component remains connected. In addition, the component is currently acyclic. The procedure first enforces the canonical representation of an acyclic component by identifying the minimal element and storing it in both tables. If the component does not have any stashed elements, then we are done. Otherwise, inserting its minimal stashed element leads to the canonical representation.

Case 4: C is unicyclic and x is not part of its cycle. When removing the outgoing edge of x, the component splits to two connected components: an acyclic component C^0, and a unicyclic component C^1. An element belongs to C^0 if the directed path connecting it to the cycle goes through x. All the other elements belong to C^1.

The unicyclic component C^1 is already in its canonical form (the minimal element on the cycle is still stored in T_0), and any stashed elements that are mapped to this components should remain stashed. As for the acyclic component C^0, the procedure enforces the canonical representation of an acyclic component by identifying the minimal element and storing it in both tables. If the component does not have any stashed elements, then we are done. Otherwise, inserting its minimal stashed element leads to the canonical representation.