Anti-persistence: History Independent Data Structures *

Moni Naor†
Dept. of Computer Science and Applied Math
Weizmann Institute
naor@wisdom.weizmann.ac.il

Vanessa Teague‡
Dept. of Computer Science
Stanford University
vteague@cs.stanford.edu

May 9, 2001

Abstract

Many data structures give away much more information than they were intended to. Whenever privacy is important, we need to be concerned that it might be possible to infer information from the memory representation of a data structure that is not available through its "legitimate" interface. Word processors that quietly maintain old versions of a document are merely the most egregious example of a general problem.

We deal with data structures whose current memory representation does not reveal their history. We focus on dictionaries, where this means revealing nothing about the order of insertions or deletions. Our first algorithm is a hash table based on open addressing, allowing $O(1)$ insertion and search. We also present a history independent dynamic perfect hash table that uses space linear in the number of elements inserted and has expected amortized insertion and deletion time $O(1)$. To solve the dynamic perfect hashing problem we devise a general scheme for history independent memory allocation. For fixed-size records this is quite efficient, with insertion and deletion both linear in the size of the record. Our variable-size record scheme is efficient enough for dynamic perfect hashing but not for general use. The main open problem we leave is whether it is possible to implement a variable-size record scheme with low overhead.

1 Introduction

Computer folklore is rife with stories about files containing information that their creators assumed had been erased only to be revealed in embarrassing circumstances. In general, if protecting privacy is an issue, then if some piece of information cannot be retrieved via the *legitimate* interface of a system, then it should not be retrieveable even when there is full access to the system. For instance, if the order of insertion of elements to a system is not part of the interface, then the system should protect this information in the data's internal representation, in case this representation becomes available (*e.g.* by losing the laptop, sending a file or some other process gaining information about the allocation in main memory or disk).

This work deals with data structures that are history independent, *i.e.* it is impossible to deduce from the memory representation of the data structure any information not revealed by its current state. This problem was first explicitly considered by Micciancio [13] who showed a variant of 2-3 trees with this property. We consider hash tables where the cost of each operation is $O(1)$. Our model of history independence is stronger

*A preliminary version of this paper appears in STOC '01, July 6–8, 2001, Hersonissos, Crete, Greece

†This work was completed while the author was visiting the IBM Almaden Center and Stanford University. Partially supported by DoD MURI "Semantic Consistency in Information Exchange," ONR Grant N00014-97-1-0505, and DARPA Contract N66001-00-C-8015

‡Partially supported by DoD MURI "Semantic Consistency in Information Exchange," ONR Grant N00014-97-1-0505, and DARPA Contract N66001-00-C-8015

than the one in [13] in that we consider the memory *representation* of the dictionary and not just the "shape" of the data structure[1].

We focus on dictionaries, *i.e.* data structures that support insert, lookup and delete from a set. Here the only history not contained in the current state is the order of insertions and deletions that led to it. There are many situations where it is important to keep this secret. For example, if we are maintaining a list of people invited to some event (such as invited speakers at a conference, guests at a wedding or members of a football team), then it might be useful to publish the data but it would be very embarrassing if people discovered that another speaker had been invited before them but declined, or that they were the last to be added to the wedding guest list.

1.1 Summary of Results

We provide definitions of history independent data structure (Section 2). These definitions are applicable to any abstract data structure. We deal mostly with finding history independent implementations of dictionaries where the goal is to obtain $O(1)$ performance per operation for any sequence of operations. We provide two types of such tables — with and without pointers. For the first, in section 3, we develop a framework for hashing schemes based on open addressing (no pointers). We give a sufficient condition for a scheme in our framework to be history independent (that the priority function induces a total order in each cell). We suggest a particular scheme with good performance: the expected amortized cost of insertion and the expected cost of search are $O(1)$. The big advantage of this scheme is space utilization - the space wasted can be as small as we want. The scheme uses only pair-wise independent functions (whereas all previous schemes of this type had to resort to $\log n$-wise independence) and requires only $O(\log n)$ of them. The disadvantage is that it does not support deletions. We then move to data structures with pointers (Section 4). Here we must first resolve the issue of memory management. We show schemes for memory allocation; these work in $O(1)$ time per allocation or deletion for fixed record size, but are more expensive for variable sized records (Sections 4.1 and 4.2). In Section 4.3 we have a history independent dynamic perfect hashing scheme where lookup takes $O(1)$ and insert and delete take expected amortized $O(1)$ steps. Finally we address the famous union find problem and show a history independent scheme where find always takes $O(1)$ operation and union takes $O(\log n)$ amortized work (Appendix A)

1.2 Related Work

As mentioned above, Micciancio [13] was the first to have dealt with history independence explicitly, in the context of search trees; the issue came up implicitly in investigations regarding data structures with *unique* representation (see [2]). In the context of lookup tables, the *ordered hashing* algorithm of Amble and Knuth ([1]) has this uniqueness property. Ordered hashing falls into the open addressing framework we develop in Section 3.

There is a large body of literature trying to make data structures *persistent*, *i.e.* to make it possible to reconstruct previous states of the data structure from the current one ([6]). We are aiming for the opposite, that no information whatsoever can be deduced about the past, hence an alternative name could have been *anti-persistence*.

There is considerable research on methods for protecting memories. Oblivious RAM [8] makes the address pattern of a program independent of the actual sequence; it incurs a cost of polylog n. Note however that it does not provide history independence since it assumes that the CPU stores some secret information; this is an inappropriate model for cases where the adversary gains complete control.

The dictionary problem is one of the most widely studied problems in computer science. Open addressing hashing schemes are covered in detail in Knuth [10]. More recent analysis showed that double hashing where the hash functions are $\log n$-wise independent is good [9, 12, 14, 17]. However, getting such functions requires either investing $\log n$ work per evaluation or using large amounts of randomness and storage (but not more than linear in the table size) to describe the function [16, 4].

[1] However, our performance guarantee is slightly weaker — we prove our results with respect to any (worst-case) sequence of operations chosen without knowing the internal coin flips of the data structure, whereas [13] assumed that the adversary choosing the sequence had access to those choices. Note that for this type of adversary no $O(1)$ dictionary is known, even without the history independence requirement.

A different approach for achieving $O(1)$ performance is via perfect hashing schemes. Here we rely on Dietzfelbinger *et al.*'s ([5]) dynamization of the FKS scheme [7].

2 Preliminaries and Definitions

An abstract data structure (ADS) is defined by a list of operations. Any operation returns a result (which may be null) and the specification defines the results of a sequence of operations. We say that two sequences S_1 and S_2 of operations on an ADS yield the same *content*[2] if for all suffixes T, the results returned by T when the prefix is S_1 are the same as those returned when the prefix is S_2. In a dictionary, two sequences have the same content iff the set they define is the same.

An implementation of a data structure maps the sequence of operations to a memory representation, *i.e.* an assignment to the content of the memory. The goal of a history independent implementation is to make this assignment depend only on the content of the data structure and not on the path that led to this content. That is, we imagine that there is a period of activity in the data structure (*e.g.* insertions, deletions and search in the dictionary example). At some point the adversary gains control of the data structure, *i.e.* sees exactly what is in the memory representing it. There are no secrets left. The adversary should not be able to deduce any more about the sequence of operations that led to the content than the content itself yields. Since we use randomization in our implementation (both for efficiency and to achieve history independence), for a given implementation each sequence of operations induces a distribution on the assignments to the memory. Therefore the definition of history independence is:

Definition 2.1 *A data structure implementation is* history independent *if any two sequences S_1 and S_2 that yield the same content induce the same distribution on the memory representation.*

Stronger definition: Note that the above definition assumed that the adversary gaining control is a one-time event (*e.g.* losing a laptop). However, in some circumstances it may be that the adversary gains periodic control and at several points along the sequence of operations it obtains a "memory dump" *i.e.* the contents of the memory at given points. The requirement is that for any two sequences of operations and two lists of points that yield the same content for all the corresponding points when the memory dumps are made, the distributions on the memory are identical.

Definition 2.2 *Let S_1 and S_2 be sequences of operations and let $P_1 = \{i^1_1, i^1_2, \ldots i^1_\ell\}$ and $P_2 = \{i^2_1, i^2_2, \ldots i^2_\ell\}$ be two lists of points such that for all $b \in \{1, 2\}$ and $1 \le j \le \ell$ we have that $1 \le i^b_j \le |S_b|$ and the content of data structure following the i^1_j prefix of S_1 and the i^2_j prefix of S_2 are identical. A data structure implementation is* strongly history independent *if for any such sequences the distributions of the memory representations at the points of P_1 and the corresponding points of P_2 are identical.*

A large class of data structures can have a history independent implementation and even one satisfying Definition 2.2: if it is possible to decide the lexicographically first sequence that yields the same content as the current one, then we can simply store that first sequence in the implementation. This may of course be rather time consuming, so the questions explored in this paper are which data structures have efficient history independent implementations.

There are various ways in which the above definitions can be extended and relaxed. One is to make the two distributions computationally indistinguishable, rather than identical.[3] Another is to allow some information to be leaked. For example we could call a data structure n-history independent if, for any two sequences S_1 and S_2, if S_1 and S_2 yield the same content and their last n operations are identical, then the distributions on their assignments to memory are identical. In this work we do not resort to these relaxations.

The definition of history independence can also be extended to *underdefined* abstract data structures (UADSs), that is data structures where the same query after the same sequence of operations is permitted several different responses (for example, a priority queue allowing any one of the ϵn top elements to be

[2] We will use the term content to denote the content of the data structure as opposed to memory representation or assignment.

[3] This definition could be utilized in erasing. We could say that a file is erased whenever the blocks allocated for it are computationally indistinguishable from randomness. This would allow us, for instance, to delete an encrypted file by deleting its key.

returned). In this case we need only to rethink the definition of content. We say that two sequences S_1 and S_2 of operations on a UADS yield the same content if there is no suffix T where the set of sequences of results permitted to be returned by T when the prefix is S_1 is different from the set permitted when the prefix is S_2.

Dictionaries: The main data structure that we deal with is the dictionary:

Definition 2.3 *A* dictionary *over a* universe $\mathcal{U} = \{0, 1, \ldots, U-1\}$ *is a partial function S from $\mathcal{U}$ to some set I. The operations Lookup(x), Insert(x), and Delete(x) are available on a dictionary S; Lookup(x) returns $i = S(x)$ if x is in the domain of S, Insert(x) adds x to the domain of S and sets the value of $S(x)$, and Delete(x) removes x from the domain of S.*

Strategies for producing history independence: There are two practical ways to make a data structure history independent. One is to ensure that the representation of the structure is determined by its current content — for example, an array of elements can be kept in a history independent way by sorting the elements and always placing them as close to the 0-th cell as possible. Then the array's contents are exactly the current elements in sorted order, arranged at the beginning of the table, regardless of their insertion (or deletion) order. The second way is to introduce secret randomness into the data structure in such a way that an observer who does not know the random choices cannot infer anything about the history. An example of this is storing elements in an array in some order which is a random permutation of their insertion order, where the permutation is secret and never stored explicitly.

In Section 3 we apply the first idea to construct a history independent hash table. We choose hash functions at the beginning of the operation and at any point the current content of the table and the hash functions uniquely determine the memory representation. The advantage of the approach is that it yields *strong history independence*. In Section 4.1 we use the randomization idea to show how to implement history independent memory allocation of fixed-size records. In Section 4.2 we use a combination of the two techniques to implement history independent memory allocation for records of variable size, which we use in Section 4.3 to obtain dynamic perfect hashing.

3 Data Structures without Pointers: open addressing

In this section we describe a history independent dictionary (without deletion) based upon an open addressing hash table. In open addressing, every element is stored within the table, so there is no need for pointers. The main advantage is that it can be very space efficient. In the traditional version, if there is a clash between two elements, the second one to arrive at the cell is moved to elsewhere according to its sequence of possible positions. Our version also resolves clashes by inserting one element and moving the other, but we don't necessarily move the second element to arrive. Our table has a fixed size of N entries. Its performance depends on the load, which we denote by α, so the dictionary contains αN elements (and the waste is $(1-\alpha)N$). The performance will be given as a function of α.

Define the *configuration* of a hash table to be the set of (*index*, *value*) pairs stored in the table.

Definition 3.1 *Let $\mathcal{P} = h_1, h_2, \ldots$ be a sequence of probe functions such that $h_j : \mathcal{U} \to \{0, \ldots, N-1\}$. Then we say that $\mathcal{P}$ defines the* probe sequence *of an element x as the sequence $(h_1(x), h_2(x), \ldots)$.*

If the hash table configuration is uniquely determined once we have fixed $\mathcal{P}$ for a given set of input values $\{x_1, x_2, \ldots, x_n\}$, then the hash table is history independent (in fact, it is strongly history independent), since in particular, it is independent of the order of insertion of the $\{x_1, x_2, \ldots, x_n\}$.

3.1 Description of the hash table algorithm

Unlike traditional hash tables, where an element is inserted into the first empty cell in its probe sequence and never moved unless deleted, we allow elements to be shifted after they have been placed in a cell (as in [1, 3]). When element x is being inserted and the next cell probed is already occupied by element x', we either move x to the next cell in its probe sequence, or place x in this cell and move x'. We use a "priority function" which, for any cell and for any pair of elements, determines which of the two elements has higher priority at that cell. If two elements hash to the same cell during insertion, the element with higher priority is placed there while the other is moved, regardless of which was inserted first. The lower-priority element is placed in the next cell in its probe sequence that is empty or contains an element of lower priority than it.

Definition 3.2 *Let $p(i,x,y) : \{0,\dots,N-1\} \times \mathcal{U} \times \mathcal{U} \to \{\mathit{True}, \mathit{False}\}$. We say that p is a* priority function *if for all i, the relation $\{(x,y) : p(i,x,y) = \mathit{True}\}$ is a total order. We write p_i for $p(i,\bullet,\bullet)$.*

We say that x has a higher priority than y at cell i if $p(i,x,y)$ is true. A special case is a global priority function, *i.e.* one where for all cells i the function p_i is the same. (This is the case with ordered hashing, [1].)

Note that priority functions can be chosen so that at a particular cell, elements further along in their probe sequence have a lower priority than those not so far along.

The space our algorithm requires includes that directly used for the table, that used to describe the hash functions $h_1, h_2, \dots$ and that used to describe the priority functions $p_1, p_2, \dots$. The only randomness occurs in the choices of hash functions h_i for $i = 1, 2, \dots$ and possibly of priority functions, (if a probabilistic priority function is being used) — given these, the rest of the computation is deterministic.

The insertion algorithm, insert-A is as follows: Given an element x to insert into table t, probe the cells in x's probe sequence until reaching either an empty cell or a cell i containing element y where $p_i(x,y)$ is true (*i.e.* the current item has lower priority than x). If the cell is empty, place x there and halt. If it is occupied by a lower-priority element y, place x there and recursively apply the insertion algorithm to y, using y's probe sequence from i onwards. Pseudo-code for insert-A is included in the Appendix.

Searching in our hash table is identical to doing so in an ordinary hash table, but there is an optimization for unsuccessful search:

Claim 3.1 *When searching for an element x, it is safe to stop as soon as we find an element y of lower priority than x.*

This follows from history independence (which we prove in the next section). We may assume that if x were in the table then it would have been the last element inserted and therefore would have bumped y (or, by transitivity, any element occupying that cell before y) and taken its place. It then could have been bumped only by a higher-priority element during rearrangements caused by recursive applications of insert-A.

3.2 Proof of history independence

We describe a table-construction algorithm, insert-B, that is static and clearly history independent, then show that the table's configuration after applying insert-B to $\{x_1, x_2, \dots, x_n\}$ is identical to that obtained by repeatedly applying insert-A to the elements of that set, in any order.

insert-B deals with sets of elements at a time. Denote the initial set of elements to be inserted by B_0 and the set of unplaced elements at the end of the i-th pass by B_i. The i-th pass of insert-B, for $i \geq 1$, begins by finding the next cell in the probe sequence for each element in B_{i-1} and provisionally placing the element there, then choosing the highest-priority element at each cell (inlcuding possibly one placed there in a previous pass) and actually placing it there. There must be a unique highest-priority element at each cell because B_i is finite and the p_i defines a total order (by definition 3.2). Let B_i be the set of all elements that were not actually placed, or were removed (because a higher-priority element clashed at the same location). This is the end of pass i. This is repeated for $i = 1, 2, \dots$ until all elements have been placed in the table (*i.e.* until $|B_i| = 0$). We keep a record of how far along its probe sequence each element has reached, so that an element that is placed for a few passes and then moved again can be moved correctly into the next cell in its probe sequence.

Because an element may be placed in one pass of insert-B and then pushed out later, it may be that the elements in B_i have not all reached the same distance along their probe sequences. However, for one priority function described later, an element once placed is never moved again and all elements in B_i are up to the $i+1$-th element of their probe sequence.

Theorem 1 *Given a set of n input values $B_0 = \{x_1, x_2, \dots, x_n\}$ and a corresponding set of probe functions $P = \{h_1, h_2, \dots\}$, the hash table configuration of* insert-B(B_0) *is equal to the configuration that results from using* insert-A *to insert the same n elements in any order.*

Proof: For any element $x_i \in B_0$, x_i is moved no further along its probe sequence by algorithm insert-B than it is by the n applications of insert-A. The proof is by induction on the rounds of insert-B. Any element x in B_k must, in the k-th round, have clashed at some cell j with a higher-priority element y. By induction x and y must both reach cell j during repeated execution of insert-A, so x will be moved another step by insert-A also.

Conversely, any element $x_i \in B_0$ is moved no further along its probe sequence by the n applications of insert-A than it is by insert-B. Proof by induction: let x be the first element which, during insert-A, is moved further along its probe sequence than it was during insert-B. Suppose at the end of insert-B, x is located in cell $h_i(x)$ and consider when it is moved to $h_{i+1}(x)$ in insert-A. Then there must be some element y in cell $h_i(x)$ with higher priority there than x. But since x reached $h_i(x)$ during insert-B and remained there, y must not have reached this cell during insert-B—if it had, it would have caused x to move or been replaced by a higher-priority element z which, by transitivity of $p_{h_i(x)}$, would have moved x also. Therefore x is not the first element to be moved further along its probe sequence than it was during insert-B.

Therefore every element in B_0 reaches the same cell at the end of insert-B as it does after the n calls to insert-A. $\diamondsuit$

Corollary 2 *For any choice of priority function (satisfying definition 3.2), the hash table configuration produced after using* insert-A *is independent of the order of insertion of the elements. The scheme is* strongly history independent.

3.3 Choice of priorities and hash functions

In order to completely specify a scheme in our framework we must describe (i) how the functions $h_1, h_2, \ldots$ for the probe sequence are chosen (ii) What priority rules are used.

The following definition is slightly non standard in that it emphasizes the properties we need

Definition 3.3 *A family $H = \{h : U \mapsto \{0 \ldots N-1\}\}$ of hash functions is ϵ-almost pairwise independent if (i) for all $x \in \mathcal{U}$ and random $h \in_R H$ we have that $h(x)$ is uniformly distributed in $\{0 \ldots N-1\}$ (ii) for all $x_1, x_2 \in \mathcal{U}$ such that $x_1 \neq x_2$ and for a random $h \in_R H$ we have $\Pr[h(x_1) = h(x_2)] \leq 1/N + \epsilon$.*

In order to get good run time analysis we propose choosing each h_i independently from the previous $h_1, h_2, \ldots h_{i-1}$ and from an ϵ-almost pair-wise independent family. In more detail, use hash functions of the form: $h_i(x) = (a_i x \bmod U + b_i) \bmod N$, where $a_i \in_R \mathcal{U}$ and $b_i \in_R \{0, \ldots, N-1\}$ are randomly chosen and U is a prime. This produces an ϵ-almost pair-wise independent family with $\epsilon \approx N/U$.

There is nothing in the choice of hash functions that guarantees that we will not cycle on a given element. However we employ the following strategy: we choose $O(\log n)$ hash functions and then resort to linear probing. For all $\delta > 0$, the probability of an element needing more than $O(\log \frac{n}{\delta})$ probes is less than δ (see section 3.4), so we use linear probing with negligible probability.

We now present some examples of priority functions. Our analysis in Section 3.4 is based upon **youth-rules**.

global A single priority function independent of cell. For all cells i, let $p(i, x, y) = p'(x, y)$ for some p' producing a total order. We recommend choosing p' from a pairwise independent family.

youth-rules Call an element "younger" if it has moved less far along its probe sequence and give "younger" elements higher priority. Assume some total order $\leq_t$ for breaking ties. More precisely, let $\mathrm{age}(i, x) = \min\{j | h_j(x) = i\}$ and

$$p(i, x, y) = \begin{cases} \text{True} & \text{if } \mathrm{age}(i,x) < \mathrm{age}(i,y) \\ \text{True} & \text{if } \mathrm{age}(i,x) = \mathrm{age}(i,y) \\ & \text{and } x \leq_t y \\ \text{False} & \text{otherwise} \end{cases}$$

age-rules The opposite of **youth-rules**.

random Choose a random order of the elements at each node. Equivalently, choose a random winner in the case of each clash, subject to the total order constraints.

One advantage of **global** is that it can be used to manipulate the search times of elements — elements with higher priority are likely to travel less far along their probe sequences and hence have a shorter successful search time than those with lower priority. It is shown in [1] to be very efficient. If we use **age-rules** then most elements are likely to be about the same distance along their probe sequence, and consequently take about the same time to search for. By contrast, **youth-rules** tends to increase the spread of probe distances. In the next section we will analyze insertion and search times for **youth-rules**.

3.4 Running time analysis

If we had a set of independent and random hash functions then our algorithm would perform during the insertions as well as "traditional" uniform hashing: we analyze algorithm insert-B (see below); there it is clear that each hash function is evaluated on a given point only once and hence does not "loose" its randomness. In contrast to the complete independence of each hash function needed for this argument, we require only almost pairwise independence from each of the hash functions in the following analysis.

Let $h_1, h_2, \ldots$ be chosen from an ϵ-almost pairwise independent family where $\epsilon \le 1/N(N-1)$ and let the priority function be **youth-rules**. We now show that the amortized per operation expected running time for *any* sequence of insertion, successful search and unsuccessful search is at most $\frac{1}{1-\alpha}$. The expectations are over the choice of the hash functions. We calculate the expected running time of insert-A by analyzing insert-B. Our analysis relies on the following observation about the relationship between them:

Remark 3.1 *Every time an element x is unplaced at the beginning of a pass in* insert-B *corresponds to x making one move along its probe sequence at some time during* insert-A. *(This move may be during the insertion of x or during the insertion of some other element that displaces it.) Hence the total number of steps taken by all elements during* insert-A *is equal to $\sum_{i=0}^{\infty} |B_i|$.*

To analyze insert-B, let $\beta_i = |B_i|/N$, so $\beta_i N$ is the number of unplaced elements at the end of pass i in insert-B. Since $|B_0| = n$, we have $\beta_0 = \alpha$. Then the average number of steps for each insertion of one element is $\frac{1}{n}\sum_{i=0}^{\infty} \beta_i N$. The significance of using **youth-rules** is that all elements in B_i are up to the $(i+1)$-th element in their probe sequence. We will show that in this case β_i decreases quickly as a function of i, so that $\sum_{i=0}^{\infty} \beta_i$ is $O(1)$.

Lemma 3 *For all $i \ge 0$, if $\beta_i N$ is the number of unplaced elements at the end of pass i in* insert-B, *then $E(\beta_{i+1}) \le \alpha\beta_i$.*

Proof: The advantage of using **youth-rules** is that all members of B_i are applying the same function h_{i+1} at this stage and this function is independent of B_i and the locations that have been settled so far. We will compute a lower bound on the expected number of members of B_i that are placed into empty cells in the table during pass $i+1$. For each cell j in the table that is unoccupied at the beginning of the $i+1$-th pass, the probability that it is occupied at the end of the pass is $\Pr[\cup_{x \in B_i} A_{x,j}]$ where $A_{x,j}$ is the event that $h_{i+1}(x) = j$. By the inclusion-exclusion principle and the pair-wise independence of h_{i+1}, this is at least

$$\begin{aligned}
& \sum_{x \in B_i} \Pr[A_{x,j}] - \sum_{\substack{x, x' \in B_i \\ x < x'}} \Pr[A_{x,j} \wedge A_{x',j}] \\
\ge\ & \frac{\beta_i N}{N} - \binom{\beta_i N}{2} \frac{1}{N}\left(\frac{1}{N} + \epsilon\right) \\
\ge\ & \beta_i - \beta_i^2/2 \text{ whenever } \epsilon \le 1/N(N-1)
\end{aligned}$$

Since there are $(1 - \alpha + \beta_i)N$ such empty cells j, the total contribution is at least

$$\begin{aligned}
& (1 - \alpha + \beta_i)(\beta_i - \beta_i^2/2)N \\
=\ & [(1-\alpha)\beta_i + \beta_i(\beta_i - \beta_i^2/2) - (1-\alpha)\beta_i^2/2]N \\
\ge\ & [(1-\alpha)\beta_i + \beta_i(\beta_i - \beta_i^2/2 - \beta_i/2)]N \\
\ge\ & (1-\alpha)\beta_i N
\end{aligned}$$

and we conclude that $E[\beta_i - \beta_{i+1}] \ge (1-\alpha)\beta_i \quad \diamondsuit$

Corollary 4 *The expected number of unsettled elements $\beta_i N$ decreases exponentially in i. More precisely,*

$$E(\beta_i) = \alpha^{i+1}$$

Proof: We know that $\beta_0 = \alpha$ and that $E[\beta_{i+1} | \beta_i = \gamma] \leq \alpha\gamma$. Therefore $E[\beta_{i+1}] \leq \alpha E[\beta_i]$ and the corollary follows by induction. $\diamondsuit$

Definition 3.4 *Define the* probe-time *of an algorithm to be the number of probes required by that algorithm.*

Note that in many applications the probe time dominates other computation such as the hash functions, but to implement **youth-rules** it is necessary when considering displacing element x from cell j to find the least i such that $h_i(x) = j$. It is possible to show that the additional work required is also a constant.

Theorem 5 *For any sequence of insertions the expected amortized insertion probe-time for an element is* $\frac{1}{1-\alpha}$

Proof: This follows from corollary 4 and remark 3.1, which implies that the amortized insertion probe-time is $\frac{1}{n}\sum_{i=0}^{\infty} \beta_i$. $\diamondsuit$

Theorem 6 *For any element $x \in \mathcal{U}$ and any set S, the expected probe-time for successful or unsuccessful search is $\frac{1}{1-\alpha}$.*

Proof: The cases $x \in S$ and $x \notin S$ are identical. Assuming the search reaches step i, it stops at that step if $h_i(x)$ does not clash with any of the settled locations or any of the elements in B_i in pass i of insert-B. (This is not an only if condition.) The first happens with probability $\alpha - \beta_i$, the second with probability at most β_i, so one of the two happens with probability at most α. Therefore the chances of stopping are at least $1 - \alpha$ and from the independence of the hash functions from each other the process is dominated by a geometric distribution and the expected time to stop is at most $1/(1-\alpha)$. $\diamondsuit$

The number of hash functions required: There is a neglibible probability of needing more than $O(\log n)$ hash functions since the probability that any particular element will need more is negligible. To see this, let E_x^i be the event that element x is still unplaced at the end of round i of insert-B. Then for all $x, \Pr[E_x^i | E_x^{i-1}] \leq \alpha$ because when conducting round i, at most αN cells may be occupied by other elements. Hence for all $x \in B_0$ and $i \geq 0$, $\Pr[E_x^i] \leq \alpha^i$, so the probability that there is an element requiring more than i hash functions is:

$$\Pr[\exists x \in B_0, E_x^i] = \Pr[\bigcup_{x \in B_0} E_x^i] \leq \sum_{x \in B_0} \Pr[E_x^i] \leq n\alpha^i$$

For any real $\delta > 0$, the probability of needing more than $\log_{1/\alpha} \frac{n}{\delta}$ is less than δ, because if $l > \log_{1/\alpha} \frac{n}{\delta}$ then

$$\Pr[\exists x \in B_0, E_x^l] < n\alpha^{\log_{1/\alpha} \frac{n}{\delta}} = \delta$$

In general our analysis in this section was pessimistic and it is probably possible to show better dependency on α. See for example Yao's bound on retrieval time for open addressing schemes ([18]). We have implemented our hash table using a variety of different priority functions and found that the performance varies with different priority functions, but that most give an average insertion and search time of $O(\log \frac{1}{1-\alpha})$.

4 Memory management

Data structures containing pointers are more difficult to make history independent than those without pointers, since the way in which memory was allocated to the data structure's parts may reveal something about the order in which they were created.

In this section we describe an algorithm that will make the memory allocation of parts of a data structure history independent. This is necessary because memory allocation of a data structure's parts may reveal information about its history, even if the data structure is carefully designed so that the structure itself (including pointers, array orders etc.) is history independent apart from memory allocation. If the data structure is history independent when ignoring memory allocation and regarding it as a directed graph where two nodes are connected if one has a pointer to the other, then the same data structure using our memory allocation algorithm will be history independent. We require only that the data structure has bounded indegree. This is necessary because the algorithm sometimes moves previously stored records, so it is necessary to update all pointers pointing to a particular record.

We first discuss a simple memory allocation algorithm for fixed size records. Making the algorithm history independent at most doubles the time for insertion and deletion of records. We then generalize this algorithm to records of any size and prove that the worst-case cost is $O(s \log s)$ per deletion and insertion, where s is the size of the record being inserted or deleted. The generalized form allows us to implement history independent dynamic perfect hashing, relying upon 4-wise independence of the hash functions involved. In this case, the expected amortized cost is $O(1)$ per deletion or insertion and the probe-cost of search is always 2.

We view memory as a large one-dimensional array which may be extended (by allocation) at one end only.

4.1 Fixed size records

Suppose we receive a sequence of requests, each of which is a request either to allocate new storage or to free a space previously allocated. It is not known *a priori* how many records will be required, though we do have an upper bound, since there is only a fixed maximum amount of memory available. History independence requires that, given a "dump" of the memory at any point after an insertion or deletion, it is impossible for an adversary to determine anything about the order the elements were inserted in or whether any have been deleted.

Let t be the table in which the records are allocated and let k be the number of records currently allocated, *i.e.* the number inserted but not deleted. Whenever an update is not in progress, all k elements are stored in the first k cells of t and their order is random. All other memory is set to zero. Insertion and deletion are carried out as follows:

Insert: To insert record r_1, choose a number $l \in_R \{0, \ldots, k\}$ at random. If $l = k$ then insert r_1 at $t[k]$. If $l < k$, insert r_1 at $t[l]$ and move the element previously located at $t[l]$ to $t[k]$. Increment k by one.

Delete: To delete the record at location $t[i]$, overwrite $t[i]$ with the record in $t[k]$, zero $t[k]$ and decrement k by one.

History independence relies on the following lemmas:

Lemma 7 (Insertion preserves randomness) *If π is a random permutation of $\{0, 1, \ldots, k-2\}$, then the permutation π' obtained by choosing $l \in \{0, \ldots, k-1\}$ uniformly at random and taking*

$$\pi'(i) = \begin{cases} k-1 & \text{if } i = l \\ \pi(l) & \text{if } i = k-1 > l \\ \pi(i) & \text{otherwise} \end{cases}$$

for $i \in \{0, \ldots, k-1\}$, is a random permutation of $\{0, 1, \ldots, k-1\}$.

Lemma 8 (Deletion preserves randomness) *If π_2 is a random permutation of $\{0, 1, \ldots, k-1\}$, then the permutation π'' obtained by choosing any $l' \in 0, \ldots, k-1$ and taking*

$$\pi''(i) = \begin{cases} \pi_2(k-1) & \text{if } i = l' \leq k-2 \\ \pi_2(i) & \text{otherwise} \end{cases}$$

for $i \in \{0, \ldots, k-2\}$, is a random permutation of $\{0, 1, \ldots, k-2\}$.

Theorem 9 *After any sequence of insert or delete operations, the memory representation is history independent and the space used is equal to the total size of records currently allocated.*

Proof: To prove history independence, it suffices to show that after any sequence of insert or delete requests, the order of records in memory is a random permutation of their insertion order. This follows by induction from lemmas 7 and 8. $\diamondsuit$

Making pointer-based data structures history-independent: Using this method, we can make the memory map of any bounded-indegree, fixed-size record data structure history independent, provided that the *shape* of the original data structure is history independent. This is the case with Micciancio's trees [13] as well as with treaps [15] (as was noted in [15], once the priority function is fixed and different for all values then the treap of a set of values is unique). The only delicate part is to ensure that when a record is moved (during the insertion or deletion of another record), all pointers pointing to that record are updated. Under the assumption of bounded indegree, this update takes constant time. It can easily be implemented using doubly-linked pointers. Also, during insertion or deletion, at most one other element (which has the same size) is moved. Hence the insertion or deletion of any element takes time $O(s)$ where s is its size. When the original structure is a tree we can skip the doubly linked pointers, since there is only one node leading to any given node and we have access to it via the search.

4.2 Variable Record Size

Most data structures use records of a variety of different sizes. The algorithm described above does not work for records of variable sizes because we can no longer guarantee that insertion or deletion runs in time proportional to the record's size.

The main idea of this section is to use a separate table for each range of record sizes, each of which behaves like the fixed-size record tables described above. Inserting or deleting elements into or from one table may require rearranging other tables. In the worst case, insertion into or deletion from this structure can take $O(s \log s)$ where s is the size of the record being inserted or deleted.

The master table t is composed of a number of smaller tables $t_n, t_{n-1}, \ldots, t_0$, stored contiguously in that order. Table t_i stores records with size greater than $\lfloor 2^{i-1} \rfloor$ and less than or equal to 2^i, with each record padded up to size 2^i. We assume that we can allocate new memory after the end of t_0, but not before the beginning of t_n. If we need to insert an element into a table t_i with no spare space, we first rearrange the other tables so as to add one more space of size 2^i to t_i, then do fixed-size insertion; for deletion we first do fixed-size deletion then rearrange the other tables. The rearrangement works as follows:

adding space pre-insertion: Let $|t_i|$ denote the total size of t_i, *i.e.* $|t_i| = 2^i \times$ (number of records in t_i) and s the size of the record to be inserted. Then we need to make space of size $s' = 2^{\lceil \log_2 s \rceil}$ in table $t_{\lceil \log_2 s \rceil}$. Begin by allocating space of size s' immediately after t_0. Working from t_0 to $t_{\lceil \log_2 s \rceil - 1}$, ("right" to "left"), do the following: for each table t_i, if $|t_i| \leq s'$, move all of t_i into the rightmost blocks in the current space. If $|t_i| > s'$, move the first $s'/2^i$ records in t_i into the current space, which they exactly fill.

removing space post-deletion: This is very similar to adding space, except that we shift blocks to the left, working from table $t_{\lceil \log_2 s \rceil - 1}$ to table t_0.

Theorem 10 *The running time for insertion or deletion is at most $s' \log_2 s' = O(s \log s)$ where s is the size of the element to be inserted or deleted. The memory used is at most $2s$.*

To prove history independence taking into account the rearrangement of tables, we use the following lemma.

Lemma 11 (Rearrangement preserves randomness) *If π_3 is a random permutation of {0, 1,. . . ,k-1}, then the permutation π'''_m obtained by letting*

$$\pi'''_m(i) = \begin{cases} \pi_3(i+m) & \textit{if } i \leq k-m-1 \\ \pi_3(i-k+m) & \textit{otherwise} \end{cases}$$

for $i \in \{0, \ldots, k-1\}$, is a random permutation of {0, 1,. . . ,k-1}.

Theorem 12 *At any time after an insert or delete operation, the order of records in memory is history independent.*

4.3 Application: Dynamic Perfect Hashing

In this section we use history independent memory allocation to construct an efficient method for dynamic perfect hash functions. Recall that for a set $S \subset \{1, \ldots, m\}$ a perfect hash function is a mapping of $\{1, \ldots, m\}$ onto $\{1, \ldots, n\}$ which is 1-1 on S. We are given a set of n elements out of $\{1, \ldots, m\}$ and the goal is to build a perfect hash function from $\{1, \ldots, m\}$ to a range which is $O(n)$ with the properties of succinct representation, efficient evaluation and efficient construction.

Current implementations of dynamic perfect hashing are not history independent because they use non history independent memory allocation, which we replace with our scheme from section 4.2. They also do not erase elements as soon as they are deleted, instead tagging them and only erasing during rehashing.

There is another, more subtle, way in which current implementations of dynamic perfect hashing violate history independence. To see the problem in the abstract, suppose we have a set of states Σ, a set of objects H and a function $G : \Sigma \times H \to \{0, 1\}$. For each $\sigma \in \Sigma$, we choose an object h_σ in $H_\sigma = \{h \in H | G(\sigma, h) = 1\}$. In our hashing scheme, Σ will be the set of possible contents of the hash table, H a set of hash functions, and G a predicate that decides whether a given $h \in H$ is "good" for a given $\sigma \in \Sigma$. Suppose that when moving from state σ to state σ' we only change h when necessary. That is, we check whether $G(\sigma', h_\sigma) = 1$ and, if it is, assign $h_{\sigma'}$ to h_σ and otherwise choose $h_{\sigma'}$ uniformly at random from $H_{\sigma'}$. Then the data structure will not in general be history independent since $h_{\sigma'}$ is biased towards $H_\sigma \cap H_{\sigma'}$ and so $h_{\sigma'}$ and σ' together yield information about σ. We will refer to this as the intersection-bias problem.

Outline of the FKS scheme: Our scheme has the same structure as the famed Fredman, Komlós and Szemerédi [7] scheme which we now review: The FKS scheme consists of two levels. The top-level function, denoted by h, maps the elements of $\{1, \ldots, m\}$ into a range of size $O(n)$; all the elements that were sent to the same location i are further hashed using a lower-level hash function h_i. The lower-level hash function h_i should be 1-1 on the subset that was hashed to location i by h. For every i in the range of h we allocate as much space as the range of h_i which we denote by r_i. The perfect hash function is now defined as follows: if $x \in \{1, \ldots, m\}$ is mapped to i by h, then the scheme maps x to $h_i(x) + \sum\limits_{1 \le j < i} r_j$. The size of the range is therefore $\sum_i r_i$.

Let $S_i(h) = \{x | x \in S \text{ and } h(x) = i\}$ and $s_i(h) = |S_i(h)|$, *i.e.* $s_i = s_i(h)$ denotes the number of elements mapped to i. The property we require h to satisfy is that $\sum_{i=1}^{n} \binom{s_i(h)}{2}$ should be $O(n)$. The size of the range of h_i will be $O(\binom{s_i}{2})$. The functions suggested by [7] for both levels were of the form $(k \cdot x \bmod p) \bmod r$ where p is an appropriate prime, r is n for the top level and s_i^2 for the lower level.

The FKS scheme was made dynamic in [5] who showed that choosing the first level hash function as well as the second level ones can be done "on the fly" and in the case that they are not appropriate (*i.e.* $\sum_{i=1}^{n} \binom{s_i(h)}{2}$ is not $O(n)$ or h_i is not 1-1 on $s_i(h)$) then new ones are chosen and rehashing is done. The amortized work is $O(1)$ per operation (search is always $O(1)$ in the worst case).

Description of the new scheme: Our scheme is very similar to that described in [5]; the main differences are:

Top-level hash functions: We use a 4-wise independent function for the top level hash function h (rather than a pair-wise one). This can be realized using a random degree 3 polynomial mod p.

Memory allocation: When we allocate space for the range of h_j, we do so using the memory allocation algorithm described in section 4.2. Each space of size s_j^2 is regarded as one record in table $t_{\lceil \log_2 s_j^2 \rceil}$. If the record becomes too big or too small to be in this table, it is deleted and a corresponding record is inserted into the appropriate table. We will show that most s_i are small, so only $O(|S|)$ space is used even though the size of each bucket is squared. We keep a top-level table with one entry for each element in the range of h, containing a pointer from that element to the record in the master table. This is used for insertion and search for elements of S, and is updated when records are deleted and re-inserted elsewhere.

Erase upon deletion: When a new hash function h_i is chosen then all mappings done by the previous h_i are erased. Whenever an element is deleted it is erased.

Low-level intersection bias: We choose a new h_i every time an element x with $h(x) = i$ is deleted. This defeats the intersection-bias problem for the low-level hash functions. If an element x is inserted then $\{h_i | h_i \text{ perfect on } S_i\} \subseteq \{h_i | h_i \text{ perfect on } S_i \cup \{x\}\}$, so the bias does not cause a problem — we always use an h_i in the intersection of the two sets.

Top-level intersection-bias: Solving the intersection-bias problem for the top-level hash function h is more complicated because it is too expensive to rehash upon every deletion. At the beginning of the algorithm we generate two different possible top-level hash functions χ_1 and χ_2. We call a top-level hash function h "good" for set S if $\sum_{i=1}^{|S|}\binom{s_i(h)}{2} < 2|S|$. We will maintain the condition that the current top-level hash function is

$$h = \chi_{\hat{j}} \text{ where } \hat{j} = \min\{j|\chi_j \text{ is "good" for S}\} \tag{1}$$

If $h = \chi_1$ then this is clearly satisfied. If $h = \chi_2$ then for χ_1 we still maintain an "alternative" top-level table with one entry for each element in the range of χ_1. The i-th entry contains the number $s_i(\chi_1)$. This is updated every time an insertion or deletion is performed. Every time an element is deleted, we use this top-level table to check whether χ_1 is "good" for S. If it is, we set h to χ_1 (thus satisfying condition 1), otherwise we retain the current h, which must still satisfy the condition. We will ensure that the size of each top-level table is $O(|S|)$. When we insert, if $h = \chi_1$ and is no longer "good" but χ_2 is, we assign h to χ_2 and rehash. If neither χ_1 nor χ_2 is "good" for S, we select a new h at every deletion until we reach an S for which either χ_1 or χ_2 is "good". This is very expensive, so we wish to ensure that it happens with very low probability.

Top-level rehashes due to size: As S grows and shrinks, it will probably become necessary to re-choose the top-level hash function, h, so that the size occupied by the master tables and the top-level tables remains $O(|S|)$. One way to do this would be to rehash as $|S|$ reaches powers of two, because then on random insertions and deletions we would expect to have to do only $O(1)$ work per operation to rehash. However, this is susceptible to an adversary inserting 2^i elements, then deleting and reinserting one repeatedly, causing us to rehash on every operation. Our solution is to choose a secret random number ρ_i in each interval $l_i = \{2^i, \ldots, 2^{i+1} - 1\}$ (for i larger than some arbitrary starting value) to use as the rehashing point. Whenever $|S|$ reaches ρ_i via an insertion, or reaches $\rho_{i+1} - 1$ via a deletion, we rehash, recreating all the top-level tables and giving them size ρ_{i+1}. Each ρ_i is independent of the others. If n currently falls in the interval l_i then we store ρ_{i-1}, ρ_i and ρ_{i+1}. In particular we erase any ρ_k for $k > i + 1$ because this would reveal that n has previously attained a higher value and so would violate history independence. This scheme defeats the deleting and reinserting attack because an adversary who does not know the ρ_i is unlikely to be able to guess them.[4]

Theorem 13 *This hash table is history independent*

Proof: The hash table's state is determined by S, the top-level hash functions, the low-level hash functions, the secret random numbers $\{\rho_1, \ldots\}$ and the arrangement of the S_i in memory, so it suffices to show that these are history independent.

No information is revealed by $\{\rho_1, \ldots\}$ because they are chosen uniformily from a fixed range and ρ_i is erased as soon as $|S|$ becomes small enough that its existence would reveal anything.

Consider the top-level hash functions. At any time we have two functions χ_1 and χ_2 chosen at random in a way independent of S, and possibly a third function h chosen uniformly at random from the set of functions "good" for S. Likewise, each low-level hash function h_i is chosen uniformly at random from the set of functions perfect on S_i.

The method of memory allocation does not affect the other variables, and we maintain no information in the hash table that reveals previous memory allocations. Hence by theorem 12, the order of records is history independent. $\diamondsuit$

We will prove that the expected (over the choice of hash functions) cost of an insertion or deletion of an element in this scheme is $O(1)$ and that the space occupied by the table is $O(|S|)$.

Lemma 14 *The total space required for the hash table is $O(|S|)$. Search, successful or not, is always $O(1)$.*

Lemma 15 *The expected time required for each top-level rehash is $O(|S|)$.*

[4] However, an adversary with access to timing information can still perform this attack, by waiting until it finds an insertion that takes a long time—see [11] for a description of related attacks.

Proof: If necessary, a new h satisfying $\sum_{i=1}^{|S|}\binom{s_i}{2} < k_1|S|$ can be found in $O(|S|)$ time as in [5].

Choosing h_i takes expected time s_i because each has a range of s_i^2, so there is a constant probability over choices of h_i that there will be no collisions.

Erasing the old hash table takes time $O(|S|)$ because the total space occupied by it is $O(|S|)$. $\diamondsuit$

Lemma 16 *For any S, the probability that a randomly chosen top-level hash function h is "good" for S is at least $1 - 1/|S|$.*

Proof: Let $S = \{x_1, \ldots x_n\}$ and let $X_{(i,j)}$ be the event that $h(x_i) = h(x_j)$. Then $\sum_{i=1}^{|S|}\binom{s_i(h)}{2} = \sum_{(i,j):i<j} X_{(i,j)}$ since both count the total number of collisions. The $\binom{n}{2}$ random variables $X_{(i,j)}$ are pairwise indpendent because h is 4-wise independent. Each $X_{(i,j)}$ has probability $1/n$. Hence the expectation and variance of $\sum_{(i,j):i<j} X_{(i,j)}$ are linear in n and so by Chebycheff's inequality we can show that

$$\Pr\left[\sum_{i=1}^{|S|}\binom{s_i(h)}{2} < 2|S|\right] \leq 1/n$$

$\diamondsuit$

Lemma 17 *Performing a top-level rehash whenever $h = \chi_1$ is no longer "good" for the current set $|S|$ takes expected amortized time $O(1)$ per insertion or deletion.*

Lemma 18 *Performing a top-level rehash at every operation in the case that neither χ_1 nor χ_2 is "good" for S costs expected amortized time $O(1)$ per insertion or deletion.*

Lemma 19 *The expected amortized time spent on top-level rehashing due to $|S|$ reaching any of the ρ_i from below (or crossing from above) is $O(1)$ per insertion and deletion.*

Proof:

We will prove that for a given sequence of insertions and deletions $\{c_1, \ldots, c_{\hat{j}}\}$, the total expected work (over choices of the ρ_i) is $O(\hat{j})$.

For each $j \leq \hat{j}$, let n_j be the size of S after executing operation c_j. Let $L_i = \{c_j | c_j$ is an insertion and $n_j \in l_i\} \cup \{c_j | c_j$ is a deletion and $n_j \in (l_i \cup \{2^i - 1\}) \setminus \{2^{i+1} - 1\}\}$. These are all the operations that could cause a top-level rehash due to ρ_i. Since the probability of ρ_i taking any particular value in the interval l_i is $1/2^i$ and since by lemma 15, one top-level rehash by an operation in this set takes time at most $O(2^{i+1})$, the expected total work due to top-level rehashing caused by operations in L_i is $O(L_i)$. Since the ρ_i are chosen independently, we can sum over i to prove the lemma. $\diamondsuit$

One difference between our scheme and that in [5] is that when s_i grows and is allocated a larger block we need to perform more work to assure history independence. In order to prove that this results in expected $O(1)$ work per operation we need the following:

Lemma 20 *For any set $S \subset \{1, \ldots m\}$ of n elements and any element $x \in S$, if h is 4-wise independent and $c = s_{h(x)}$ is the number of elements colliding at $h(x)$ we have that $E[c^2 \log c]$ is $O(1)$.*

Proof: We know that for any S and $x \in S$ and function h we have

$$\begin{aligned} c^2 \log c &\leq c^3 \leq 2 \cdot 3! \cdot \binom{c}{3} + O(1) \\ &\leq 2 \cdot 3! \cdot \sum_{\substack{x_1, x_2, x_3 \in S \\ |\{x, x_1, x_2, x_3\}| = 4}} \delta(h(x), h(x_1), h(x_2), h(x_3)) + O(1) \end{aligned}$$

where $\delta(h(x), h(x_1), h(x_2), h(x_3)) = 1$ iff all 4 values are equal and 0 otherwise and where the $O(1)$ term is to handle the case that c is smaller than 3. Letting $p = \Pr[h(x) = h(x_1) = h(x_2) = h(x_3)]$, it follows that

$$E[c^2 \log c] \leq 2 \cdot 3! \cdot \sum_{\substack{x_1, x_2, x_3 \in S \\ |\{x, x_1, x_2, x_3\}| = 4}} p + O(1) = 2 \cdot 3! \frac{\binom{n}{3}}{n^3} + O(1)$$

which is $O(1)$. The equality follows from the 4-wise independence of h. $\diamondsuit$

Theorem 21 *The amortized expected time taken to insert or delete an element is $O(1)$.*

Proof: We have already considered top-level rehashing in lemmas 17–19. We now prove the result for an operation that does not cause a top-level rehash.

Consider inserting x where $h(x) = j$. If h_j is no longer perfect on S_j then we can rechoose it and rehash that record in time $O(s_j^2)$. If there is not enough room to insert x in the current record, (*i.e.* if $(s_j+1)^2 > 2^{\lceil \log_2 s_j^2 \rceil}$), then that record must be deleted and a new one of size $s' = 2^{\lceil \log_2 (s_j+1)^2 \rceil}$ inserted into table $t_{\lceil \log_2 (s_j+1)^2 \rceil}$. By Theorem 10, this takes time at most $2s' \log s'$ so the total work is $O(s_j^2 \log s_j)$ which by Lemma 20 has expected value $O(1)$. A similar argument holds for deletion, except that then we always rechoose h_j. Note that moving records happens rarely though the result still holds even if it happens every time. $\diamondsuit$

Rechoosing randomness: Rechoosing χ_1, χ_2 and $\rho_1, \ldots$ at each step with probability $O(1/|S|)$ would not violate history independence, since their distribution is independent of S. This would be advantageous because, although it would not significantly change the amortized expected work per operation, it would reduce the variance by breaking the sequence of operations up into several independent sequences. This also gives us something somewhat similar to strong history independence—as long as the adversary's many queries are all from periods with different randomness, no information about history is leaked.

5 Open Problems

One of the major problems we have left open is whether it is possible to get a memory allocation scheme (of variable size) with a low overhead, in particular one that takes advantage of the efficiency of storing a file in a large block. This may be significant for file systems, where even if the files are encrypted the positions of the files in the disk might leak undesirable information.

Another issue that we have not addressed is that of clocking or timing attacks — for instance if the adversary knows the time it takes the system to respond to the queries it might deduce some information. This point was raised for performance purposes in [4] and [11]. However it is not clear how to make the techniques history independent.

One interesting theoretical question is whether there is a separation between strong and weak history independence. For example, for queues there is an easy implementation with weak history independence — choose a random starting point in the array and grow the queue from there using the usual algorithm. However, we have not been able to devise an equally fast, strongly history independent version. It is interesting to note that all the strongly history independent data structures we have found have the property that each data structure content (in the sense defined in section 2) has a unique representation. We would be interested to know whether unique representation is necessary for strong history independence. It would also be interesting to find a problem for which there is a separation between the standard version and the history independent one.

Various weakenings of the definition of history independence may be useful for particular applications. For example, when considering a cache system it is necessary to expose some information about the frequency of the different requests (otherwise the cache would be completely ineffective). However, we could require that only frequency-related information would be released.

It would also be interesting to know whether persistence and (computational) anti-persistence could exist in the same data structure—in this case it should be impossible to retrieve any history information except with a secret key, though ordinary operations could be performed without it.

Finally there are many data structures for which the issue of history independence may be relevant. In Appendix A we discuss union-find. The interesting question is whether history independence contradicts good performance for certain problems. We have seen that it does not for hashing.

6 Acknowledgements

We would like to thank Ilya Mironov, John Mitchell and Rasmus Pagh for interesting discussions and useful feedback on our paper.

References

[1] O. Amble and D. Knuth. Ordered hash tables. *The Computer Journal*, 17(2):135–142, 1974. Reprinted in: D. E. Knuth, Selected Papers on Analysis of Algorithms, Center for the Study of Language and Information Lecture Notes, no. 102, Stanford, 2000.

[2] A. Andersson and T. Ottmann. Faster uniquely represented dictionaries. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 642–649, 1991.

[3] R. P. Brent. Reducing the retrieval time of scatter storage techniques. *Comm. of the ACM*, 16:105–109, 1973.

[4] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 6–19, 1990.

[5] 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 J. Comput.* , 23(4):738–761, 1994.

[6] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

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

[8] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM*, 43(3):431–473, 1996.

[9] L. J. Guibas and E. Szemerédi. The analysis of double hashing. *Journal of Computer and System Sciences*, 16:226–274, 1978.

[10] D. E. Knuth. *Sorting and Searching, Volume 3 of The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 1998. First edition 1973.

[11] R. J. Lipton and J. F. Naughton. Clocked adversaries for hashing. *Algorithmica*, 9(3):239–252, 1993.

[12] G. S. Lueker and M. Molodowitch. More analysis of double hashing. *Combinatorica*, 13(1):83–96, 1993.

[13] D. Micciancio. Oblivious data structures: Applications to cryptography. In *Proc. 29th ACM Symp. on Theory of Computing*, pages 456–464, 1997.

[14] J. P. Schmidt and A. Siegel. On aspects of universality and performance for closed hashing. In *Proc. 21st ACM Symp. on Theory of Computing*, pages 355–366, 1989.

[15] R. Seidel and C. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996.

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

[17] A. Siegel and J. Schmidt. The analysis of closed hashing under limited randomness. In *Proc. 22nd ACM Symp. on Theory of Computing*, pages 224–234, 1990.

[18] A. C. Yao. Uniform hashing is optimal. *Journal of the ACM*, 32(3):687–693, 1985.

A History Independent Union-Find

The Union-Find problem requires maintaining a data structure representing a collection of disjoint dynamic sets. Each set is represented by one of its members. The data structure must support the creation of a new set (containing only a specified element), the uniting of two sets (Union) and finding the set containing

```
procedure insertA(x, t)
begin
 k := 0;                          /* Index into its probe sequence */
                                  /* of the current element */
 curr_elt := x; /* Element we are currently moving */
 finished := false
 while(not finished) do
  if t[h_k(curr_elt)] is empty   /* Probed cell is empty. */
                                  /* place x here and halt */
    t[h_k(curr_elt)] := curr_elt;
    finished := true;
  else if p(h_k(curr_elt), t[h_k(curr_elt)], curr_elt)
                                  /* Cell is occupied by a higher-priority */
                                  /* element, so keep moving x. */
    k := k + 1;
  else                            /* Cell is occupied by a lower-priority element. */
                                  /* Put x here and move the other element*/
  swap(t[h_k(curr_elt)], curr_elt)
  Find the least k' so that h_k'(curr_elt) = i.
  k := k' + 1;
 done
end
```

Figure 1: Pseudo-code for insertA

a specified element (Find), where a set is identified by one of its elements. In order to make this history-independent, we need to ensure that the answers returned are independent of the order in which sets were created and united and elements were searched for.

When defining Union-Find we must be careful about the name of the subset returned from the Find, since we do not want to leak information through this channel. We therefore make two proposals. One is to return the name of the smallest element in the subset. The other is to make Find a query of the form, "Are x and y in the same subset?"

We now sketch a history independent implementation of Union-Find at the cost of $O(1)$ per Find and expected amortized $O(\log n)$ computations per Union. The idea is to use two global lookup tables which will be maintained in a history independent manner, as described above. In one table for each element x a record (x, s) is stored, where s is the (current) set to which x belongs. The record is searchable by x. (Having a "sophisticated" data structure for this table is redundant, in case the set of elements is fixed as $1..n$.) The second table has for each set s and index $i \leq |s|$ a record (s, i, x) searchable by (s, i). The mapping of members of s to indices is random. There is also an entry $(s, 0, |s|)$ to indicate how many elements are in s.

The Find operation is trivial: simply look it up in the first table. The Union of two sets s_1 and s_2 is done by taking the smaller set s_1, finding all its elements x via the second table and changing their entry in the first table to (x, s_2), then changing in the second table the entries of the form (s_1, i, x) to $(s_2, |s_2| + i, x)$ and then choosing for each i a random element j between 1 and $|s_2| + i$ and flipping the j-th element and the $|s_2| + i$-th element.

The number of modifications in the table is proportional to the size of the smaller set and therefore the "classical" analysis yields a total work of $O(n \log n)$ insertions and deletions from the lookup table for any number of Union operations, using the scheme of section 4.3.