Strongly History-Independent Hashing with Applications

Guy E. Blelloch *
Computer Science Department
Carnegie Mellon University

Daniel Golovin †
Computer Science Department
Carnegie Mellon University

Abstract

We present a strongly history independent (SHI) hash table that supports search in $O(1)$ worst-case time, and insert and delete in $O(1)$ expected time using $O(n)$ data space. This matches the bounds for dynamic perfect hashing, and improves on the best previous results by Naor and Teague on history independent hashing, which were either weakly history independent, or only supported insertion and search (no delete) each in $O(1)$ expected time.

The results can be used to construct many other SHI data structures. We show straightforward constructions for SHI ordered dictionaries: for n keys from $\{1, \ldots, n^k\}$ searches take $O(\log \log n)$ worst-case time and updates (insertions and deletions) $O(\log \log n)$ expected time, and for keys in the comparison model searches take $O(\log n)$ worst-case time and updates $O(\log n)$ expected time. We also describe a SHI data structure for the order-maintenance problem. It supports comparisons in $O(1)$ worst-case time, and updates in $O(1)$ expected time. All structures use $O(n)$ data space.

1 Introduction

Computer users on a typical system leave significant clues to their recent activities, in the form of logs, unflushed buffers, files marked for deletion but not yet deleted, and so on. This can have significant security implications. Similarly many file formats are effectively data structures and can contain historical information or clues on what once appeared in the file. Hartline *et al.* [10] describe an example of a CIA document that was released as a PDF file in 2000 with classified information redacted by overlaying black boxes. Unfortunately the overlays could easily be removed revealing key information about the CIA's role in the 1953 overthrow of the Iranian Government. History repeats itself. In 2005 the US military released a PDF report on the accidental shooting of Italian secret agent Nicola Calipari in Iraq, again with classified information redacted by overlaying black boxes. Again the overlays could be removed, and among the information revealed was the name of the US military shooter, Mario Lozano.

To address the concern of releasing historical and private information the notion of *history independent* data structures was devised [13, 14]. Roughly, a data structure is history independent if someone with access to the memory (file) layout of the data structure (henceforth called the "observer") can learn no more information than a legitimate user accessing the data structure via its standard interface. We assume the observer is computationally unbounded.

The most stringent form of history independence is called *strong history independence* and requires that the behavior of the data structure under its standard interface along with a collection of randomly generated bits uniquely determine its memory representation. Thus a SHI data structure has a canonical representation up to randomness. Data structures with canonical representations are interesting both theoretically and practically. Theoretically, there are the natural questions about the time and space complexity of such data structures. As a bonus, canonical form may simplify proofs that data structures have other properties. We use this to our advantage when bounding the running time of our hash table.

Practically, there are many applications for SHI data structures. Consider a filesystem which supports deleting a file in a way that provably leaves no trace that it ever existed. Imagine a government wanting to publish the names of all voters in an election without divulging any information about when or in what order they voted. Publishing a SHI hash table would be a natural solution. There are other applications of canonical forms aside from hiding historical information. For example, it makes equality testing of the contents of two data structures very simple based purely on the memory layout and requires no knowledge of the representation or even the contents. It also allows for easy digital signing of a data structure at various times to provide proof that it contained certain data at those times. Again the signing code need know nothing about the representation or contents. Canonical form can also help with debugging computations that have some nondeterminacy in the ordering of operations. If a SHI data structure, for example, is generated via some parallel process and then later

*Supported in part by NSF ITR grants CCR-0122581 (The Aladdin Center). Email: *blelloch@cs.cmu.edu*

†Supported in part by NSF ITR grants CCR-0122581 (The Aladdin Center) and IIS-0121678. Email: *dgolovin@cs.cmu.edu*

the system crashes, then while rerunning the code on the same input we can be sure that whenever the system enters a previously attained state, it has the same memory layout as before, even if the exact interleaving of the parallel operations leading up to this state is different the second time due to nondeterministic timing issues.

Previous Work. Canonical representations were studied by Snyder [21], Sundar and Tarjan [22], and Andersson and Ottmann [3]. They showed lower and upper bounds for the ordered dictionary problem in regards to the canonical form of their pointer representation. Andersson and Ottmann [3], for example, show $\Theta(n^{1/3})$ matching lower and upper bounds. This shows a strong separation from redundant representations which have $\Theta(\log n)$ lower and upper bounds. The model considered, however, disallows even labels on the keys—only comparisons are allowed. If labels are allowed, the lower-bounds do not hold.

Pugh and Teitelbaum [17] describe how canonical representations can be used for incremental computation with applications to dynamic algorithms. The idea is that canonical representations make it easier to test for equality for the purpose of memoization (what they call function caching).

Micciancio [13] defined the notion of *oblivious* data structures in which the *pointer structure* reveals nothing about its history. Oblivious data structures need not have canonical pointer representation since they only require that the probability distribution over possible pointer representations is independent of the sequence of operations.

The two main notions of history independence, which we formally define in section **??**, were advanced by Naor and Teague [14], and further studied by Hartline *et al.* [10]. Informally, we say a data structure has a logical *state*, which maps each allowed operation to an output and the following state. Informally, a data structure is *weakly history independent* (WHI) if any two sequences of operations resulting in the same state result in the same distribution over memory representations. Here, the distributions may vary with a sequence of random bits hidden from the observer. If the observer is allowed to see these random bits yet still obtain no information beyond that provided via the interface, the data structure is said to be *strongly history independent* (SHI). That is, a SHI data structure has, for each fixed sequence of random bits, a canonical memory layout for each state.

Buchbinder and Petrank [5] studied the time complexity of WHI and SHI heaps and queues in a comparison based model of computation similar to those considered in [3, 21, 22]. They show a large complexity separation between WHI and SHI versions of these data structures.

In regards to hash tables, Amble and Knuth [2] developed a SHI hash table that does not support deletions. They showed that it has excellent performance assuming a random hash function is used. Naor and Teague [14] similarly develop an efficient SHI hash table that does not support deletions, but it requires only $O(\log n)$ pair-wise independent hash functions rather than a truly random hash function. In the same paper they also develop a WHI hash table based on dynamic perfect hashing [7] with searches taking $O(1)$ worst-case time and insertions and deletions taking $O(1)$ expected amortized time.

Our Contributions. We first describe a framework for constructing SHI hash tables using a variety of open address hashing schemes. Our framework allows us to exploit a recent breakthrough result of Pagh *et al.* [16], who showed that linear probing with 5-universal hash functions yields expected $O(1/(1-\alpha)^3)$ time operations, where α is the load of the hash table. Specifically, using linear probing with a 5-universal hash function we obtain a SHI hash table that stores $n \leq \alpha p$ keys in p slots of space such that the expected time to perform any search, insertion, or deletion is $O(1/(1-\alpha)^3)$. Our framework reveals a subtle connection between history independent hashing and the Gale-Shapley stable marriage algorithm [9], which may be of independent interest.

We then construct SHI data structures with various worst case guarantees. Unfortunately, these require a potentially exponential amount of randomness, however they serve as the basis for more practical data structures that either (*i*) retain the running times but make the data structure SHI *with high probability* (by which we mean with probability $1 - 1/n^c$ for any user-defined constant $c > 0$) and WHI with certainty, or (*ii*) retain SHI-ness but replace worst case time guarantees with "with high probability" time guarantees. These require only $O(n^\delta)$ random bits for any constant $\delta > 0$, and, in case (*i*), the ability to sample random bits as needed. For ease of exposition we describe the original versions first, and then the (relatively straightforward) modifications needed to get guarantees of form (*i*) and (*ii*).

The first such data structure we present is a SHI hash table that supports search in $O(1)$ worst-case time and updates in $O(1)$ expected time. Our approach is novel, and quite different from previous approaches [7, 8, **?**]. We then use this hash table to construct other SHI data structures. We show straightforward constructions for ordered dictionaries. For n keys from $[1, \ldots, n^k]$ searches take $O(\log \log n)$ worst-case time and updates (insertions and deletions) take $O(\log \log n)$ expected time. For keys in the comparison model searches take $O(\log n)$ worst-case time and updates take $O(\log n)$ expected time. Both structures use $O(n)$ data space.

We also describe a SHI data structure for the order-maintenance (OM) problem. It supports comparisons in $O(1)$ worst-case time, and updates in $O(1)$ expected time. The OM problem is important since it can be used in the framework of Acar *et al.* [1] to generate SHI data structures by *dynamizing* static algorithms. Acar *et al.* gave sufficient conditions for strong history independence of dynamized algorithms based on a SHI OM structure. The SHI OM and SHI hashing imply, for example, a SHI data structure for

dynamic trees supporting updates and queries in $O(\log n)$ expected time and using $O(n)$ data space.

Finally, note we do not allow the observer to decide what operations are performed when giving these guarantees. Since we allow the observer to inspect our hash table's random bits, allowing the observer to select the set S would be disastrous in terms of performance if U is sufficiently large (as it would be for any space efficient hash table). Of course, for any fixed set of n keys selected without knowledge of the hash table's random bits, the hash table operations will take expected constant time.

2 Preliminaries

For convenience, we define $[k] := \{0, 1, 2, \ldots, k-1\}$. Throughout the paper, we let *with high probability* (whp) mean with probability at least $1 - 1/n^c$ where $c > 0$ is any user defined constant.

We consider the problem of hashing $n \leq \alpha p$ (key, object) pairs into a table of length p, for any positive integer p. The quantity $\alpha \in (0, 1)$ is called the *load* of the hash table. Since we are interested in space efficient hashing, we assume the load α is constant.

For our machine model, we assume a standard unit cost RAM with word size at least $\log |U|$, where U is the universe of keys. Thus, a key can be stored in a single word. We also assume the ability to sample various hash functions $h : U \to [p]$ which can be evaluated in $O(1)$ time. For a discussion of efficient $O(1)$-universal hash functions, see [16]. Where $\Theta(\log n)$-universal hash functions are needed, the constructions of Siegel [19, 20] and Östlin and Pagh [15] are suitable, assuming the keys are integers. The latter are also suitable where full randomness is called for.

History independence is defined below. The definition of weak history independence is reproduced from Naor and Teague [14], and is given here for completeness. Our definition of strong history independence differs from that of Naor and Teague, however the two definitions were proved equivalent by Hartline *et al.* [10] for reversible data structures (i.e., those for which there always exists some sequence of operations which returns the data structures to its initial state), which include all the data structures in this paper.

Definition 2.1 (Weak History Independence) *A data structure is* weakly history independent (WHI) *if, for any two sequences of operations X and Y that take the data structure from initialization to state A, the distribution over memory representations after X is performed is identical to the distribution after Y is performed. (The distribution in question is over the random bits that the WHI data structure may hide from the observer.)*

Definition 2.2 (Strong History Independence) *A reversible data structure is* strongly history independent (SHI) *if it has canonical representations up to initial randomness. That is, for each sequence of initial random bits and for each state of the data structure, there is a unique memory representation.*

Note that building an efficient SHI hash table based on separate chaining requires a SHI memory allocator. We therefore focus on open-address hash schemes. We will also need the following definitions.

Definition 2.3 (k-Universal Hash Family) *A family $\mathcal{H}$ of functions from X to Y is* k-universal *if for all distinct $x_1, x_2, \ldots, x_k \in X$ and for all $y_1, y_2, \ldots, y_k \in Y$ $\mathbf{Pr}_{h \in \mathcal{H}}\left[\bigwedge_{i=1}^{k} h(x_i) = y_i\right] \leq |Y|^{-k}$ where h is chosen uniformly at random from $\mathcal{H}$.*

Definition 2.4 (Data Space) *The* data space *of a data structure is the space it uses excluding any space used to store random bits.*

3 Efficient SHI Hashing

Our approach is based on exploiting an interesting property of the stable marriage algorithm of Gale and Shapley [9], stated below in Theorem 3.1. The stable marriage problem is as follows: Given a set M of n men and a set W of n women, and a preference list over the opposite sex for each person, find a *stable* matching $E \subset M \times W$ of size n. A matching E is stable if for all $(m, w), (m', w') \in E$, it is *not* the case that m prefers w' to w and w' prefers m to m'.

Recall that in the Gale-Shapley stable marriage algorithm, the men propose to the women in decreasing order of their preferences, and each woman is tentatively matched with her favorite man among all those who have proposed to her. The algorithm terminates when all men are tentatively matched with some woman.

Note that the algorithm is underspecified in the sense that there may be many men who are not tentatively matched, and they may propose in arbitrary order. Thus, there are many different valid executions of this algorithm, corresponding to various ways of selecting among tentatively unmatched men. Nevertheless, the following theorem, implicit in [9] and treated explicitly in [11], shows the outcome is not affected by the choice of valid execution.

Theorem 3.1 ([9]) *Every execution of the Gale-Shapley algorithm results in the same stable matching.*

Framework. Theorem 3.1 suggests the following approach to constructing a SHI hash table that supports insertions and searches: interpret the keys as men and the slots of the hash table as women, and construct a distribution on stable marriage instances between U and the set of all slots.

This distribution is based on the random bits of the hash table. In particular, the probe sequence for a key $\mathbf{k}$ will equal $\mathbf{k}$'s preference list over the slots. (If the probe sequence has duplicate entries, retain only the first occurrence of each slot to obtain the preference list). The preference lists for each slot will be used to resolve collisions. In this case, Theorem 3.1 ensures that for each set of keys of size at most n, the resulting memory representation is the same no matter what order the keys are inserted in. So the resulting hash table is SHI under insertions. To ensure that the hash table performs well, we must ensure that

1. We can sample efficiently from the distribution on stable marriage instances.
2. Each instance in the distribution can be represented compactly, such that the following operations take constant time: determining the i^{th} slot in $\mathbf{k}$'s preference list, determining slot x's rank in $\mathbf{k}$'s preference list, and comparing any two keys with respect to x's preference list, for arbitrary $i, \mathbf{k}$ and x.
3. For each set of keys $S \subset U$ such that $|S| \leq n$, the expected running time of the Gale-Shapley algorithm on an instance drawn from the distribution and restricted to the set of men S is $O(|S|)$ on every valid execution of the algorithm.

We note that the SHI hash table of Naor and Teague [14], which does not support deletions, fits directly into this framework. Intuitively, to ensure property (3) it makes sense to construct the slot preference lists so that each slot prefers keys that rank it high on their preference lists. Not surprisingly then, Naor and Teague favor what they call "youth-rules" for collision resolution, which do exactly this.

To enable support for deletions, the crux of the matter is efficiently computing, for a slot x currently holding key $\mathbf{k}$, the slot x' of the most preferred key $\mathbf{k}' \neq \mathbf{k}$ (according to x's preference list) that prefers x to its current slot. This requirement is what keeps us from extending the SHI hash table of Naor and Teague to support deletions. Though this is a function of the state of the hash table, we will abuse notation slightly and denote it by $\text{NEXT}(x)$.

Psuedocode for insertion and deletion are given in Figure 1. In the psuedocode, $\text{PROBE}(\mathbf{k}, i)$ is the i^{th} slot in $\mathbf{k}$'s probe sequence, A is the array of the hash table, and $\text{RANK}(\mathbf{k}, x) = i$ if x is the i^{th} slot in $\mathbf{k}$'s probe sequence.

Implementation. Though there are many hashing schemes, such as quadratic probing, that can be implemented in our framework to give efficient SHI hash tables, perhaps the simplest implementation is based on linear probing. Interestingly enough, specializing our framework to linear probing results in essentially the same insertion algorithm as the linear probing specialization of Amble and Knuth's 'Ordered hash table' framework [2]. Of course, our hash table will also support deletions.

```
FIND(key k)
    For (i = 0, 1, 2, ..., p − 1)
        If (A[PROBE(k, i)] is empty OR slot
            PROBE(k, i) prefers k to A[PROBE(k, i)])
            return null;
        Else if (A[PROBE(k, i)] equals k)
            return PROBE(k, i);

INSERT(key k)
    Set x = PROBE(k, 0), i = 0, and k' = k;
    While (A[x] not empty)
        If (A[x] equals k') then return;
        Else if (slot x prefers A[x] to k')
            Increment i; Set x = PROBE(k', i);
        Else (slot x prefers k' to A[x])
            Swap the values of k' and A[x];
            Set i = RANK(k', x); Increment i;
            Set x = PROBE(k', i);
    Set A[x] = k'; return;

DELETE(key k)
    Let slot x = FIND(k).
    If x is null, return.
    While (NEXT(x) is not null)
        Set y = NEXT(x); Set A[x] = A[y]; Set x = y;
    Set A[x] to be empty; return;
```

Figure 1. Psuedocode for a generic SHI hash table following our framework.

To build a hash table for n keys we fix $p = (1 + \epsilon)n$ for some $\epsilon > 0$, and a total ordering on the keys. As long as we can compare two keys in constant time, this ordering can be arbitrary, however for simplicity of exposition we will assume the keys are integers and use the natural ordering. That is, each slot prefers $\mathbf{k}$ to $\mathbf{k}'$ if $\mathbf{k} > \mathbf{k}'$. Then sample a 5-universal hash function $h : U \to [p]$ that can be evaluated in constant time. The functions $\text{PROBE}(\mathbf{k}, i) := (h(\mathbf{k}) + i) \mod (p)$, and $\text{RANK}(\mathbf{k}, x) := (x - h(\mathbf{k})) \mod (p)$ can both be computed in constant time.

Search proceeds in a fashion similar to a standard linear probing hash table. Specifically, we try $\text{PROBE}(\mathbf{k}, i)$ for $i = 0, 1, 2$, etc., until reaching a slot containing $\mathbf{k}$, an empty slot, or a slot containing a key $\mathbf{k}'$ such that $\mathbf{k}' < \mathbf{k}$. In the last case, if $\mathbf{k}$ had been inserted, it would have displaced the current contents of slot $\text{PROBE}(\mathbf{k}, i)$, so we can report that $\mathbf{k}$ is not present.

Deletions are slightly more involved. We supply the psuedocode for $\text{NEXT}(\cdot)$ in Figure 2. In the case of linear probing, $\text{NEXT}(x)$ is the slot x' containing the largest key $\mathbf{k}'$ that probed x but was rejected (or displaced) in favor of another key. Thus $\mathbf{k}'$ residing in slot x' satisfies $\text{RANK}(\mathbf{k}', x) < \text{RANK}(\mathbf{k}', x')$. Furthermore, since all slot preference lists are the same, $\text{NEXT}(x)$ is the slot y that minimizes $(y - x) \mod p$ from among all slots with keys satisfying this condition.

```
NEXT(slot x)
    Set x' = (x + 1) mod (p);
    While (A[x'] not empty)
        If (RANK(A[x'], x) < RANK(A[x'], x'))
            return x';
        Set x' = (x' + 1) mod (p);
    return null;
```

Figure 2. Psuedocode for $\mathrm{NEXT}(\cdot)$ in the linear probing implementation.

Canonical Memory Representation. We first prove that any hash table following our framework is indeed SHI.

Theorem 3.2 *For any hash table following our framework, after fixing the random bits there is a unique representation of the slots array for each set of $p-1$ or fewer keys.*

We will sketch a proof of Theorem 3.2. Fix the hash table's random bits and a set of keys S such that $|S| \leq p-1$. Searches do not change the memory representation of the slots array, and thus we can safely ignore them. Defer the treatment of deletions for the moment. We will use Theorem 3.1 to show that any sequence of insertions resulting in the table having contents S results in the same memory representation. Suppose key $\mathbf{k} \in S$ is stored in slot $s(\mathbf{k}) \in [p]$. Then $\{(\mathbf{k}, s(\mathbf{k}))|\mathbf{k} \in S\}$ is the stable matching output by the Gale-Shapley algorithm on a particular stable marriage instance. In this instance, $M := S$ and $W := [p]$. The preference lists for each $\mathbf{k} \in M$ are built from the probe sequence for $\mathbf{k}$: if $\mathrm{RANK}(\mathbf{k}, w) < \mathrm{RANK}(\mathbf{k}, w')$, then $\mathbf{k}$ prefers w to w'. The preference lists for each $w \in [p]$ can be arbitrary. It is now straightforward to verify that any sequence of insertions corresponds to a valid execution of the stable matching algorithm. Note also that Theorem 3.1 easily extends to the case that $|M| < |W|$, and thus we can apply it to show that there is a unique representation of the slots array if only insertions and searches are permitted.

Finally, consider deletions. Proving that deletions preserve the unique representation property amounts to proving that any two sequences of operations ρ and ρ' resulting in the same hash table contents result in the same representation. Suppose that this holds for any sequence pairs (ρ, ρ') such that ρ has no deletions and ρ' has at most one. Then an easy induction on the maximum number of deletions in $\{\rho, \rho'\}$ yields the desired result. So we consider (ρ, ρ') such that ρ has no deletions and ρ' has exactly one. WLOG, we can assume that the deletion in ρ' deletes a key that was present at the time, that neither ρ nor ρ' has any searches, and that the deletion in ρ' is the last operation in ρ'. So suppose $\rho' = (\mathrm{insert}(\mathbf{k}'_1), \mathrm{insert}(\mathbf{k}'_2), \ldots, \mathrm{insert}(\mathbf{k}'_r), \mathrm{delete}(\mathbf{k}'_r))$. Since insertions maintain the unique representation and ρ contains only insertions, we know that ρ results in the same representation as $(\mathrm{insert}(\mathbf{k}'_1), \mathrm{insert}(\mathbf{k}'_2), \ldots, \mathrm{insert}(\mathbf{k}'_{r-1}))$, so WLOG we let ρ equal this sequence.

We will show that $\mathrm{delete}(\mathbf{k}'_r)$ exactly undoes all the changes $\mathrm{insert}(\mathbf{k}'_r)$ makes to the slot array. Set $\mathbf{k}_0 := \mathbf{k}'_r$. During an insertion, whenever two keys collide and the current key in the slot is evicted, we say that the evicted key is *displaced* by the other key. In the psuedocode, $\mathbf{k}'$ displaces $A[x]$ when we reach the case "x prefers $\mathbf{k}'$ to $A[x]$." We will define $\mathbf{k}_i$ as the key displaced by $\mathbf{k}_{i-1}$ during the insertion of $\mathbf{k}_0 \equiv \mathbf{k}'_r$. Suppose the chain of displaced keys is $\{\mathbf{k}_i | i = 1, 2, \ldots, d\}$. Let $s(\mathbf{k})$ be the slot containing $\mathbf{k}$ immediately after the operation $\mathrm{insert}(\mathbf{k}_0)$ in ρ', and $s'(\mathbf{k})$ be the slot containing $\mathbf{k}$ immediately before the operation $\mathrm{insert}(\mathbf{k}_0)$ in ρ'. It is easy to see that $s(\mathbf{k}_i) = s'(\mathbf{k}_{i+1})$ for all $i \in \{0, 1, \ldots, d-1\}$, and that the keys not in $\{\mathbf{k}_i | i \in [0, d]\}$ are not affected. Now consider $\mathrm{delete}(\mathbf{k}_0)$. It first finds $x = s(\mathbf{k}_0) = s'(\mathbf{k}_1)$. It then repeatedly sets $A[x]$ to $A[\mathrm{NEXT}(x)]$ and sets x to $\mathrm{NEXT}(x)$ while $\mathrm{NEXT}(x)$ exists. To ensure this is the desired behavior, we require that $\mathrm{NEXT}(s(\mathbf{k}_i)) = s(\mathbf{k}_{i+1})$. Fortunately, this is relatively easy to confirm via proof by contradiction, as is the fact that the delete operation correctly clears the slot that used to contain $\mathbf{k}_d$, and leaves all other slots unaffected. Thus ρ' results in the same representation as ρ.

Since the linear probing hash table stores only the slot array, we can immediately infer the following.

Corollary 3.3 *The hash table implementation described above is SHI.*

Space and Time Complexity. The hash table implementation based on linear probing requires $p = n/\alpha$ slots to store n keys and requires no auxiliary memory other than that used to store and compute the hash function. As we will show, the expected cost for all operations is $O(1/(1-\alpha)^3)$.

We bound the expected time per operation for our hash table implementation by comparing it to a standard linear probing hash table. Recall that this standard hash table selects a hash function $h(\cdot)$, uses probe sequences $\mathrm{PROBE}(\mathbf{k}, i) = h(\mathbf{k}) + i \mod (p)$, and resolves all collisions in favor of the key already residing in the contested slot. In a recent breakthrough result, Pagh *et al.* [16] showed that linear probing with 5-universal hash functions yields expected $O(1/(1-\alpha)^3)$ time operations.

Theorem 3.4 *The linear probing hash table implementation described above performs searches, insertions, and deletions in expected $O(1/(1-\alpha)^3)$ time, where the hash table has p slots and $n = \alpha p$ keys.*

Proof: First consider only searches and insertions. Fix a set of keys S of size at most n. It is easy to see that if the standard hash table and our hash table use the same hash function $h(\cdot)$, then after inserting S (using any sequence of operations that does not contain delete operations to do so) both hash tables will have exactly the same set of occupied

slots, even though they likely have different memory representations. Note that for the standard hash table the cost to insert $\mathbf{k} \notin S$ is $\Theta(d_S^h(\mathbf{k}))$, where $d_S^h(\mathbf{k})$ is one plus the smallest i such that slot $(h(\mathbf{k})+i) \mod (p)$ is unoccupied. It is not hard to see that in our hash table, during insertion of $\mathbf{k}$ the while loop is executed at most $d_S^h(\mathbf{k})$ times, and each iteration takes constant time. Thus if the standard table takes time t to insert $\mathbf{k}$ after a sequence of operations ρ, our hash table takes $t' = O(t)$ time. Using the result of Pagh *et al.* [16], $\mathbf{E}[t'] = O(\mathbf{E}[t]) = O(1/(1-\alpha)^3)$.

Searching for $\mathbf{k} \notin S$ takes the same amount of time as inserting $\mathbf{k}$, up to multiplicative constants. Searching for $\mathbf{k} \in S$ similarly takes less time than inserting $\mathbf{k}$, assuming we have inserted all keys in $S \setminus \mathbf{k}$ first. So this is expected $O(1/(1-\alpha)^3)$ time as well.

Now consider deletions. Suppose we insert keys set S and then delete key $\mathbf{k} \in S$. We can compute $\mathrm{RANK}(\cdot)$ in constant time. Looking at the psuedocode for delete and $\mathrm{NEXT}(\cdot)$, it is easy to prove that $\mathrm{delete}(\mathbf{k})$ takes time $O(d_S^h(\mathbf{k}))$. So, as before, we consider inserting all elements of $S \setminus \mathbf{k}$ before inserting $\mathbf{k}$, and then inserting $\mathbf{k}$ last before deleting it. By our analysis above, the $\mathrm{insert}(\mathbf{k})$ operation takes time $O(d_S^h(\mathbf{k}))$ which is $O(1/(1-\alpha)^3)$ in expectation, so the deletion takes $O(1/(1-\alpha)^3)$ in expectation as well. Note that for a hash table which is not SHI this line of reasoning is invalid, because changing the order of insertions might change the memory representation of the hash table and conceivably reduce the amount of time the delete operation takes. However we may safely dispense with this concern because our hash table is SHI. ■

Remark: Our result is modular in the following sense. If linear probing with a hash function drawn from a hash family $\mathcal{H}$ results in $f(\alpha, n)$ expected time for insertions, then using a hash function drawn from $\mathcal{H}$ in our construction results in $O(f(\alpha, n))$ searches, insertions, and deletions.

Dynamic Resizing. We can dynamically resize the hash table using the standard technique of doubling its size and rehashing all keys upon reaching a threshold number of keys. For good performance against an adversary, we select the threshold randomly, as done in previous work [10, 14].

4 SHI Perfect Hashing

Building on the work of Fredman *et al.* [8], Dietzfelbinger *et al.* [7] gave a hash table with $O(1)$ *worst case* time for lookups and $O(1)$ amortized expected time for insertions and deletions, while using space linear in the number of keys. More recently, Pagh and Rodler [**?**] developed a different technique to obtain the same guarantees, called *cuckoo hashing*. In this section we will present a third way to achieve identical time and data-space bounds while maintaining strong history independence, assuming our machine has access to a large sequence of random bits. Unfortunately, for reasons which we will discuss later, we cannot sample random bits "on demand." On the other hand, our approach is novel and relatively simple. Previously the only history independent hash table with these performance guarantees was a WHI hash table due to Naor and Teague [14], who built on the work in [7].

For ease of exposition, we begin with an impractical design that requires exponentially many random bits, and afterwords describe how to modify it for practical use. We will assume that the number of keys to be stored, n, is known in advance[1].

Theorem 4.1 *There exists a SHI hash table that executes insertions and deletions in expected $O(1)$ time, executes search in worst case $O(1)$ time, and uses $O(n)$ data space to store n keys.*

We will describe a simplified version of the hash table that works assuming various low probability events do not occur, and then address these problematic events.

The Simplified Version. We begin with an array with $p = c_0 n$ slots to store the keys, where $c_0 > 1$ is a constant, and a fast $\Omega(\log n)$-universal hash function h mapping keys to slots. We will insert and delete keys from the hash table as we did in section 3 using linear probing, but will need to maintain some additional state. Let $\delta(\mathbf{k})$ be the displacement of key $\mathbf{k}$ in the hash table. Fix a parameter $\beta = \Theta(\log n)$, and maintain a fast $\Omega(\log n)$-universal hash function f from keys to a set of labels $\mathcal{L} := \{1, 2, \ldots, |\mathcal{L}|\}$, where $\beta^3 \leq |\mathcal{L}| = \mathrm{poly}(\beta)$. Each key $\mathbf{k}$ receives a label $f(\mathbf{k})$ when inserted into the hash table. Each slot x will have an *index* I_x associated with it. The index for x will store tuples $(\delta(\mathbf{k}), f(\mathbf{k}))$ for each key $\mathbf{k}$ such that $h(\mathbf{k}) = x$, in sorted order.

Let us assume for the moment that the displacement of any key is $O(\log n)$. In this case, each displacement requires only $O(\log \log n)$ bits to store. Note that the labels require only $O(\log \log n)$ bits to store as well. It is well known that using an $c \log n$-universal hash function (with c sufficiently large) to hash n keys into $p \geq (1+\epsilon)n$ slots (for $\epsilon > 0$) ensures that at most $O(\log n / \log \log n)$ keys are hashed to any one slot with high probability. Assuming this is the case, for each slot x we can store its index I_x using a word-packed vector of only $O(\log n)$ bits. (For simplicity, we will require all index tuples to use the same number of bits.) Word-level parallelism then allows us to perform various queries and updates to the index in constant time. For example, we can insert and delete tuples in constant time, even while maintaining a canonical form (that is, the records should be maintained in sorted order, and if r records are stored in I_x, they must be stored in the first r spaces in the index). Thus, in the course of inserting

[1] We can dispense with this assumption using a SHI variant of the standard resizing technique, as in section 3.

and deleting keys, we can amortize the cost of updating the index tuples for each key which moved against the cost of moving it. We can also answer in constant time queries of the form "for what values of d is there a tuple of the form (d, l) in I_x" for any $l \in \mathcal{L}$. Assuming the labels for each key hashed to slot x are distinct, we can use this fact to do searches in constant time as follows. Find all d such that $(d, f(\mathbf{k}))$ is in $I_{h(\mathbf{k})}$. Since the labels in I_x are distinct, the output contains at most one value for displacement, say d, at which point we may immediately test the slot $(x + d) \mod (p)$ to see if it contains $\mathbf{k}$.

The Full Version. In the simplified version we made three assumptions that are false in general, namely we fixed constants c_1 and c_2 and assumed

1. No slot has more than $c_1 \frac{\log n}{\log \log n}$ keys hashed to it.
2. No key has a displacement more than $c_2 \log n$.
3. For all slots x, the keys hashed to x get distinct labels.

To remove assumption #1, it is tempting to simply sample a new the hash function using fresh randomness whenever it is violated. However, we cannot do this in a naive way and maintain SHI-ness. Instead, we maintain a random permutation π^{hash} on an $\Omega(\log n)$-universal family $\mathcal{H}$ of hash functions mapping the keys to the p slots. The idea is that we will use the first hash function, π_0^{hash}, in π^{hash} until and unless assumption #1 is violated. In that case we will iterate through the hash functions $\{\pi_i^{\text{hash}} | i = 0, 1, \ldots\}$, rehashing everything using the current hash function π_i^{hash} until we find the first one which satisfies the assumption (and the "no block overflow assumption," explained in the treatment of assumption #2 below). We will denote the current hash function by h. To maintain SHI-ness, we will need to take extra precautions during a deletion if the current hash function is not π_0^{hash}. Specifically, we will need to completely clear the hash table, reset the current hash function to π_0^{hash}, and reinsert every key (other than the one to be deleted) from scratch. (Note that it would not be SHI to simply iterate backwards through π^{hash} and stop at π_{i+1}^{hash} if π_i^{hash} is the first hash function we encounter that violates assumption #1.)

We remove assumption #2 by essentially treating the hash table as $\Theta(n/\log n)$ hash tables of size $\Theta(\log n)$. We partition the slots into *blocks*, each block will be a contiguous set of $\beta = \Theta(\log n)$ slots (assume p is a multiple of β). Keys hashed into a block do not leave it, unless h is changed. Rather, the probe sequence wraps around the block boundaries. Formally, if the block B contains slots in the range $[a, b]$, then the probe sequence for a key $\mathbf{k}$ with $h(\mathbf{k}) \in [a, b]$ is given by $\text{PROBE}(\mathbf{k}, i) := a + ((h(\mathbf{k}) - a + i) \mod (\beta))$. This ensures the displacement of any key is at most β, assuming the block has at most β keys hash into it. Call this the "'no block overflow assumption." If π_0^{hash} hashes more than β keys to some block, we will ensure each block has at most β keys in it by iterating through the hash functions $\{\pi_i^{\text{hash}} | i = 1, \ldots\}$, rehashing everything using the current hash function π_i^{hash} until we find the first one which satisfies the no block overflow assumption (as well as assumption #1).

To remove assumption #3, we store a random permutation π^{label} on an $\Omega(\log n)$-universal family $\mathcal{H}'$ of hash functions mapping the keys to a set of labels $\mathcal{L}$, where $\beta^3 \leq |\mathcal{L}| \leq \text{poly}(\beta)$. Each block B will store a pointer to a hash function in π^{label}, which we will call its *label function*, and denote by f_B. Each block's label function initializes to π_0^{label}, and we maintain the invariant that f_B is the first hash function in π^{label} that gives distinct labels to every key which has been hashed to x, for each $x \in B$. Again, we will need to be careful with deletions. Upon deleting a key $\mathbf{k}$ in block B for which $f_B \neq \pi_0^{\text{label}}$, we iterate through $\{\pi_i^{\text{label}} | i = 0, 1, \ldots\}$, relabeling all the keys hashed into block B (excluding $\mathbf{k}$) using π_i^{label} until we find a label function that satisfies assumption #3 for all slots in B with this set of keys.

Analysis. It is relatively straightforward to prove that the above data structure is SHI given the results of section 3, and we omit the details in the interests of space.

The space usage of the data structure is $O(n)$ if we exclude the random bits. The slots and their indices use $O(n)$ space. Using, for example, the hash functions of Östlin and Pagh [15], the single pointer into π^{hash} requires $O(n)$ words of space, and each pointer into π^{label}, of which there are $O(n/\log n)$, requires $O(\log n)$ words of space each (since the hash functions in π^{label} need only be uniform for sets of size $\Theta(\log n)$).

We now analyze the running time of each operation. First, consider a search for key $\mathbf{k}$. By construction, assumptions #1 through #3 hold. Thus in the worst case we need only evaluate $x := h(\mathbf{k})$, compute $\mathbf{k}$'s label l, find any record of the form (d, l) that may exist in I_x, compute $y := \text{PROBE}(\mathbf{k}, d)$, and determine if $\mathbf{k}$ is in slot y. All of these are constant time operations.

Next, consider insertions and deletions of a key $\mathbf{k}$ under "best case circumstances", which we define to mean that $h = \pi_0^{\text{hash}}$, $f_B = \pi_0^{\text{label}}$ where block B contains $h(\mathbf{k})$, and at most $\beta/2$ keys have been hashed into B. In this case, we can amortize each index update (a constant time operation) against a specific operation that moves a key. Thus we can ignore their cost. Note that inserting or deleting $\mathbf{k}$ costs essentially the same as the corresponding operation into a hash table with β slots and at most $\beta/2$ keys, using an $\Omega(\beta)$-wise independent hash function – that is, expected constant time. Now relax the condition that a block B has at most $\beta/2$ keys, and allow it to have up to β keys. The cost to do any operation is bounded by β in this case. Let $|B|$ denote the number of keys in B. We will argue that $\mathbf{Pr}[|B| \geq \beta/2] \leq 1/n^c$ if the parameters are set appropriately, and thus the cost contribution from this case is negligible. We set $p = 4e \cdot n$, $\beta = 2c \log n$, $\gamma := \beta/2$ and

use γ-universal hash functions in π^{hash}. Then we get the following estimate: $\mathbf{Pr}[|B| \geq \gamma] \leq \binom{n}{\gamma}\left(\frac{\beta}{p}\right)^{\gamma} \leq n^{-c}$.

Next, we consider the contribution of various bad events to the expected running time. Consider insertions. The probability that we must rehash everything using the next hash function in π^{hash} can be made as small as $O(n^{-c})$ for any constant $c > 0$ by adjusting various parameters, since this only occurs when more than $c_1 \log n / \log\log n$ keys hash to some slot x, which occurs with probability $O(n^{-c})$, or when some block B gets more than β keys, which we have argued above occurs with probability $O(n^{-c})$. Thus we only have to rehash everything with probability $O(n^{-c})$. Since this takes $O(n)$ expected time, its contribution to the expected running time is negligible. For a set of keys S, call a hash function in π^{hash} *bad for* S if it hashes more than $\beta/2$ keys from S into some block or more than $c_1 \log n / \log\log n$ keys into some slot. Fix S with $|S| \leq n$, let $\varepsilon(S, i)$ denote the event that π_i^{hash} is bad for S, and note that for any i, $\mathbf{Pr}\left[\varepsilon(S,i) \;\middle|\; \bigcap_{j<i} \varepsilon(S,j)\right] \leq \mathbf{Pr}[\varepsilon(S,i)]$ because by the principle of deferred decisions we can imagine selecting π_i^{hash} after $\{\pi_j^{\text{hash}} | j < i\}$, and noting that if the concentration of "bad for S" sets is higher than average among $\{\pi_j^{\text{hash}} \mid j < i\}$, then it must be lower than average among the remaining hash functions, from which we draw π_i^{hash}. Using this fact, we conclude that the total expected time to do all the rehashing is $O\left(\sum_{t\geq 1} tn/n^{ct}\right) = o(1)$.

The expected cost of rehashing after a deletion can be bounded similarly. That is, the probability that $h = \pi_t^{\text{hash}}$ is $O(n^{-ct})$, and the work involved in this case to rehash everything up to t times is $O(tn)$. The total expected time to do the rehashing is thus $O\left(\sum_{t\geq 1} tn/n^{ct}\right) = o(1)$.

Finally, we need to bound the cost due to relabeling within a block. Since the number of keys in the block is less than the independence of the label functions in π^{label}, the labels will be distributed randomly. If γ keys are present in a bucket B, we can bound the probability that all keys in B get distinct labels as follows. The probability that no collisions occur is $\prod_{j=1}^{\gamma-1}\left(1 - \frac{j}{|\mathcal{L}|}\right)$. Using $1 - x \geq e^{-x-x^2}$ for $x \in [0, 1/2]$, we can obtain a lower bound of $\exp\left(-\sum_{j=1}^{\gamma-1}(j/|\mathcal{L}|) - \sum_{j=1}^{\gamma-1}(j/|\mathcal{L}|)^2\right)$, bound the exponent from below by $z := \left(-\gamma^2/2|\mathcal{L}| - \gamma^3/3|\mathcal{L}|^2\right)$, and then use $e^z \geq 1+z$ to conclude that $\mathbf{Pr}[\text{no collisions}] \geq 1 - \frac{\gamma^2}{2|\mathcal{L}|} - \frac{\gamma^3}{3|\mathcal{L}|^2}$. Since we can relabel all γ keys in block B in $O(\beta)$ time, and setting $|\mathcal{L}| \geq \beta^3$ ensures that $\mathbf{Pr}[\text{some collision}] = O(1/\beta)$, the expected work to relabel everything once is $O(1)$. By a similar argument as above, we can bound the probability that we relabel everything in a block t times by $O(\beta^{-t})$, so the total expected work from relabeling is $O\left(\sum_{t\geq 1} t\beta/\beta^t\right) = O(1)$.

Practical Variants. Unfortunately, we cannot resort to sampling random bits "on demand." To see why, suppose we repeatedly insert keys S, then delete them, and S requires the hash table to sample new random bits. Whether or not we retain these new random bits after deleting S, we will violate SHI-ness. Thus we are forced to do all the sampling we will ever need to do during data structure initialization. This will require access to a possibly exponential sized sequence of random bits. We ensure the number of random bits we need is finite by sampling new hash functions without replacement – in other words, we sample a random permutation on a suitable hash family.

This is still hopelessly impractical. The saving grace is that our data structures can be made to inspect only $O(n^{\delta})$ random words with high probability, for any constant $\delta > 0$, if we use the hash functions of Siegel [20] in π^{hash} and those of Östlin and Pagh [15] for π^{label}. This suggests the following approach: set thresholds $\tau^{\text{hash}}, \tau^{\text{label}} \in \mathbb{N}$, and only sample the first τ^{hash} elements of π^{hash} and the first τ^{label} elements of π^{label}. Reasonable values would be $\tau^{\text{hash}} = 1$ and $\tau^{\text{label}} = \Theta(\log n)$. Then, if the data structure should ever need to access a hash function that has not been sampled, enter one of two failure modes.

The first failure mode is to sample fresh random bits on demand from now on. The resulting data structure is SHI with high probability, WHI with certainty, and has the same running time guarantees as before. Furthermore, if you additionally store the fresh random bits and treat them as the needed elements of π^{hash} and/or π^{label}, then other than the random bits, the data structure is SHI. Thus the *only* historical information that might be inferred in this case is that something was inserted and later deleted that forced the data structure to sample additional random bits.

The second failure mode is to store everything as in the SHI hash table of section 3. The whole data structure then remains SHI with certainty, provided on each delete we reinsert everything from scratch to determine if we should still be operating in this failure mode. The expected time guarantees for insertions and deletions still hold, however searches now take $O(1)$ time with high probability rather than with certainty as before.

5 Implementing Other SHI Data Structures

By analogy with the practical variants of SHI perfect hashing, we can define failure modes for all of the following data structures. The result is that either we (*i*) retain the running times but make the data structure SHI with high probability (and WHI with certainty), or (*ii*) retain SHI-ness but replace worst case guarantees with "with high probability" guarantees. We omit the details for lack of space.

Dynamic Ordered Sets. We describe a SHI solution to the dynamic ordered set (or dictionary) problem supporting Insert, Delete, and Predecessor. We assume the keys come

from a totally ordered universe U with comparison $<$ and Predecessor(key **k**) returns the entry with the greatest key less or equal to **k**. Our solutions are straightforward given the SHI hash structure described in the previous section. If the keys come from a universe $U = [m]$ we can use the variant of the Van Emde Boas structure [23] described by Mehlhorn and Naher [12]. The following theorem follows directly from the use of our SHI hash table in the algorithm described by Mehlhorn and Naher.

Theorem 5.1 *A SHI ordered dictionary on n keys from the domain $U = [n^k]$ for any constant k can support predecessor in $O(\log\log n)$ worst-case time, and insertion and deletion in $O(\log\log n)$ expected time, while using $O(n)$ data space.*

In the case that the keys come from a general universe with only comparison we use a variant of treaps [18]. We assume every element has a unique label that can be used by a hash function.

Theorem 5.2 *A SHI ordered dictionary on n keys from a totally ordered domain U can support predecessor in $O(\log n)$ worst-case time, and insertion and deletion in $O(\log n)$ expected time, while using $O(n)$ data space.*

Proof: We store all elements in a treap using a SHI hash table to locate the elements in memory. We use a random hash function to generate priorities. From Lemma 4.8 of [18] we have $\mathbf{Pr}[D(x) \geq 1 + 2c\ln n] < 2(n/e)^{-c\ln(c/e)}$ where $D(x)$ is the depth of an element x. For a set of keys S, we will call a priority function f *good* if the corresponding treap on S using f has depth at most, say, $8\ln n$, and *bad* otherwise. In the unlikely event that the initial priority function is bad, we would ordinarily just respond by generating a new random priority function and reconstructing the tree with the new priorities. To maintain SHI-ness, however, we must be careful. As with dynamic perfect hashing, on initialization we select a random permutation on a suitable family of priority functions π^{prio}, and iterate through $\{\pi_i^{\text{prio}} \mid i = 0, 1, 2, \ldots\}$ until we find the first priority function which is good for the current keys. As with hashing, after deleting a key we will need to reconstruct the treap using each π_i^{prio} in increasing order of i until we find the first one which is good for the current set of keys. However since the probability that we need reconstruct the treap t times is $O(n^{-\lambda t})$ with $\lambda = 4\ln(4/e) \approx 1.545$, the expected reconstruction cost is $o(1)$. ■

SHI Order Maintenance. The *Order-Maintenance* Problem involves creating a data structure that stores a total ordering σ while supporting the following operations:

- Insert(x, y): insert new element y right after x in σ.
- Delete(x): delete element x from σ.
- Compare(x, y): determine if x precedes y in σ.

Dietz and Sleator [6] developed a data structure that supports all three operations in worst case $O(1)$ time. (Note that the inputs are assumed to be pointers to the elements in the data structure.) Bender *et al.* [4] later developed simpler data structures with the same performance guarantees. Both structures depend significantly on the history and we see no simple modifications to make them SHI. In this section we describe a randomized SHI data structure for order maintenance that takes worst-case $O(1)$ time for compare and expected $O(1)$ time for updates. We assume each element has a unique label $l \in U$ that can be hashed. We can use the labels, for example, to place the elements into our SHI hash table.

Bender *et al.* describe a general technique for the *list-labeling problem* based on a certain class of weight-balanced trees. The list-labeling problem is to maintain a dynamic list (with insertions and deletions) along with a mapping from elements in the list to integer labels in the range $[0, u)$ such that the ordering in the list matches the ordering of the labels. The list-labeling problem can be used to implement order-maintenance by using the integer labels for comparison. We say that the *weight* $w(x)$ of a node x in a tree is the number of its descendants (including itself), and the *weight cost* of an operation is the sum of the weights of all modified nodes in the tree plus the runtime for the operation. We then have:

Theorem 5.3 ([4]) *Any balanced-tree structure with (amortized) weight cost $f(n)$ for insertions, maximum degree d, and depth h yields a strategy for list labeling with (amortized) cost $O(f(n))$ and tags from universe $[0, d^h)$.*

For a binary tree, the idea of the technique is to label each node x with the binary representation of the path from the root to x, where left branches yield 0 and right branches 1. Since internal nodes might be a prefix of another, all paths can be terminated with an additional 1 giving the desired (lexicographic) ordering. It is straightforward to show that Theorem 5.3 also applies for expected weight costs, and the expected weight cost for updates to a treap is known to be $O(\log n)$ [18]. Furthermore treaps are history independent. This yields a SHI data structure for the list-labeling problem that supports $O(\log n)$ expected time updates and $O(1)$ time comparisons with high probability (since labels have $O(\log n)$ bits with high probability).

To make updates $O(1)$ amortized time, previous work used a two level structure in which the top level stores a partition of the elements into $\Theta(n/\log n)$ sets of size $O(\log n)$, such that updates to the top level take $O(\log n)$ time and are relatively rare, and updates to the second level take $O(1)$ time (see e.g., [6]). Unfortunately the bottom level technique is highly dependent on history. To achieve history independence we use two levels based on Theorem 5.3 and a third level using state transitions with table lookup; this uses our hashing scheme in an interesting way.

Theorem 5.4 *The SHI order maintenance problem can be supported with $O(1)$ worst-case-time comparisons, and $O(1)$ expected time updates.*

Proof: We dynamically resize the data structure by reconstructing it from scratch whenever the number of elements crosses a threshold in $\{\lceil \alpha \cdot 2^k \rceil \mid k \in \mathbb{Z}_{\geq 1}\}$, where α is chosen uniformly at random from $[1, 2]$. This allows us to assume we have an upper bound, N, on the current number of elements, n, being stored, such that $N/2 \leq n \leq N$. To bound the expected reconstruction cost for each update, note that the probability of any operation crossing a threshold is $O(1/n)$, constructing the necessary lookup tables (defined below) takes $o(n^\epsilon)$ time for any $\epsilon > 0$, and reinserting all the elements takes expected $O(n)$ time (assuming updates take $O(1)$ expected time), so this adds only $O(1)$ time in expectation to any update.

So assume we know N, and $N/2 \leq n \leq N$. We partition the treap as follows based on a size parameter s. A *partition leader of rank s* is any element x in the treap such that its weight $w(x) \geq s$. Any element with $w(x) < s$ is *assigned* to the partition of its least ancestor that is a partition leader. Note this implies no partition has more than $2s - 1$ elements. From Siedel and Aragon [18] we have that for any node x in a treap, $\mathbf{Pr}[w(x) = s] = O(1/s^2)$ for any $1 \leq s < n$ and $\mathbf{Pr}[w(x) = n] = O(1/n)$. Thus for any s, $\mathbf{Pr}[w(x) \geq s] = O(1/s)$, and so each node is a partition leader with probability $O(1/s)$.

Based on this partitioning we use the following scheme. Let $s = \lceil \log N \rceil$ and $s' = \lceil \log \log N \rceil$. We sample a priority function h, and store all partition leaders of rank s (of the treap containing all elements) in a treap by themselves. The partition leaders of rank s' in each top level partition are likewise stored in treaps by themselves with one treap per top level partition. The remaining nodes are stored in treaps, one per second level partition. All these treaps may use h to generate priorities. We use Theorem 5.3 to implement list labeling separately for each treap defined above. We also maintain pointers from each non-leader to its rank s' leader and from each rank s' leader to its rank s leader. Finally, to facilitate updates, for each leader x (of both ranks) with elements assigned to it we maintain the number of elements assigned to x that are less than x in key order, and likewise the number greater than x in key order. To compare nodes x and y, we first compare their rank s leaders using their labels. If x and y have the same rank s leader, compare their rank s' leaders. Finally, if x and y have the same rank s' leader, compare them directly. (Note that to achieve worst case $O(1)$ comparisons we must be able to compare two labels from $[0, 2^h]$ in constant time, where h is the depth of the top-level treap. Thus, we require that $h = O(w)$, where $w = \Omega(\log N)$ is the word size of our machine. We will ensure the treap has sufficiently small depth as we did in the proof of Theorem 5.2, using π^{prio}. As before, the additional expected cost to do this is $o(1)$.) It is possible to show that this construction yields expected $O(\log \log \log n)$ time updates, and we could add levels to achieve expected $O(\log^* n)$ time updates. To get constant time we use table lookup rather than list labeling for the third level.

The idea of the third-level is to maintain the $l \leq 2\lceil \log \log N \rceil$ items per partition in a SHI hash table of size $t = \Theta(\log \log N)$, rather than a treap. As before, each element maintains a pointer to its leader. Since the hash table by itself does not define the ordering among the elements, we represent each possible ordering among the occupied hash-table locations as a distinct state. The number of possible states is at most $\sum_{x=0}^{t} \binom{t}{x} \cdot x! = \sum_{x=0}^{t} \frac{t!}{(t-x)!} \leq \sum_{x=0}^{t} t^x \leq 2 \cdot t^t$. Since t is $O(\log \log N)$, the number of states is $o(N^\epsilon)$ for all $\epsilon > 0$. We can thus represent the state in a single word of memory which we store with each table. To allocate space for the SHI hash tables we can use another SHI hash table with the partition leader as the key.

Each state defines a function $q : [t] \times [t] \to \{<, >, \text{undefined}\}$, where $q(a, b)$ returns whether the element stored at location a comes before or after the element at location b or undefined if either location is unoccupied. This function can be represented as a lookup-table with t^2 2-bit entries. The total space for representing all functions is therefore $o(N^\epsilon (\log \log N)^2)$ bits, which is $o(n^\delta)$ for all $\delta > \epsilon$. To implement the Compare function between elements that fall in the same table (second-level partition) we find the location of each in the table and use q to compare the locations. If two elements appear in different second-level partitions, we compare their partition leaders.

We also need to define state transition tables for updates in a second level partition. The insertion of an element into a hash table can be broken down into a sequence of (possibly zero) swaps, followed by an insertion into an empty slot. Deletion is symmetric. We therefore only need state transitions for insertion into an empty slot, deletion from a slot, and swapping of two slot. Each can be represented as a table with t^2 entries, with one such table per state. For example, when inserting **k** into an empty slot, the transition is specified by the slot y to insert **k** into, as well as the slot (if it exists) containing the key that immediately precedes **k**, among those stored in the relevant hash table. As with the comparison function q, these tables will use $o(n^\delta)$ bits.

The overall scheme for an Insert(x, y) can be outlined as follows. First we must determine if the treap on leaders of rank s must be altered. There are two ways this could happen: y itself could become a leader (its priority is greater than some element on the search path), or it could force an overflow that creates a partition leader (some node of weight $s - 1$ now has weight s). Both possibilities can be detected in $O(1)$ time given the information we maintain. As stated earlier, the probability a node is a partition leader of rank s is $O(1/s)$, and if it is, it will require expected $O(s)$ time to update labels in the top-level treap (Theorem 5.3). Furthermore expected $O(s)$ time might be required to reor-

ganize the expected $O(s)$ elements in all partitions that fall below it. An overflow would require that after the insertion there is an element within s elements of x in the ordering which now has exactly s descendants. Since there are at most $2s$ potential elements, each with probability $O(1/s^2)$ of having exactly s children, the probability of forcing an overflow is $O(1/s)$. When there is an overflow, $O(s)$ work is required. Updating the relevant size information takes $O(1)$ time. The cost of insertion in the top level is therefore $O(1)$ expected time. The same procedure is used on the second level and again requires $O(1)$ expected time. The SHI hash-table along with the state transition tables is used for the third level requiring another $O(1)$ time. Deletion is symmetric. ■

6 Conclusions

We have shown how to build a space efficient, strongly history independent hash table with $O(1)$ worst-case search time and $O(1)$ expected update time. The techniques exploit a connection between history independent hashing and the Gale-Shapley stable marriage algorithm and can be used in conjunction with various probing schemes. We described some SHI data-structures that make use of the SHI hash table. We expect that our results will enable efficient SHI data structures for a large class of problems.

Acknowledgments

We thank Kirk Pruhs, Shan Leung Maverick Woo, and Adam Wierman for helpful discussions.

References

[1] U. A. Acar, G. E. Blelloch, R. Harper, J. L. Vittes, and S. L. M. Woo. Dynamizing static algorithms, with applications to dynamic trees and history independence. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 531–540, 2004.

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

[3] A. Andersson and T. Ottmann. New tight bounds on uniquely represented dictionaries. *SIAM Journal of Computing*, 24(5):1091–1103, 1995.

[4] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *ESA '02: Proceedings of the 10th Annual European Symposium on Algorithms*, pages 152–164, London, UK, 2002. Springer-Verlag.

[5] N. Buchbinder and E. Petrank. Lower and upper bounds on obtaining history independence. In *CRYPTO '03: Proceedings of the Advances in Cryptology*, pages 445–462, 2003.

[6] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 365–372, New York, NY, USA, 1987. ACM Press.

[7] M. Dietzfelbinger, A. 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.

[8] 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.

[9] D. Gale and L. Shapley. College admissions and the stability of marriage. *American Mathematical Monthly*, 69:9–15, 1962.

[10] 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.

[11] J. M. Kleinberg and E. Tardos. *Algorithm design*. Pearson/Addison-Wesley, first edition, 2006.

[12] K. Mehlhorn and S. Naher. Bounded ordered dictionaries in $o(\log \log n)$ time and $o(n)$ space. *Information Processing Letters*, 35(4):183–189, 1990.

[13] D. Micciancio. Oblivious data structures: applications to cryptography. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 456–464, New York, NY, USA, 1997. ACM Press.

[14] M. Naor and V. Teague. Anti-presistence: history independent data structures. In *STOC '01: Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 492–501, New York, NY, USA, 2001. ACM Press.

[15] A. Östlin and R. Pagh. Uniform hashing in constant time and linear space. In *STOC '03: Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 622–628, New York, NY, USA, 2003. ACM Press.

[16] A. Pagh, R. Pagh, and M. Ruzic. Linear probing with constant independence. In *STOC '07: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 318–327, New York, NY, USA, 2007. ACM Press.

[17] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 315–328, 1989.

[18] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.

[19] A. Siegel. On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications. In *FOCS: IEEE Symposium on Foundations of Computer Science*, pages 20–25, 1989.

[20] A. Siegel. On universal classes of extremely random constant time hash functions and their time-space tradeoff. Technical report, New York University, New York, NY, USA, 1995.

[21] L. Snyder. On uniquely representable data structures. In *FOCS '77: IEEE Symposium on Foundations of Computer Science*, pages 142–146. IEEE, 1977.

[22] R. Sundar and R. E. Tarjan. Unique binary search tree representations and equality-testing of sets and sequences. In *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 18–25, New York, NY, USA, 1990. ACM Press.

[23] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.