Strongly History Independent Hashing with Deletion

Guy E. Blelloch and Daniel Golovin

October 2006
CMU-CS-06-156

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We present a strongly history independent (SHI) hash table that is fast, space efficient, and supports deletions. A hash table that supports deletions is SHI if it has a canonical memory representation up to randomness. That is, the string of random bits and current hash table contents (the set of (key, object) pairs in the hash table) uniquely determine its layout in memory, independently of the sequence of operations from initialization to the current state. Thus, the memory representation of a SHI hash table reveals exactly the information available through the hash table interface, and nothing more. Our construction also reveals a subtle connection between history independent hashing and the Gale-Shapley stable marriage algorithm [7], which may be of independent interest. Additionally, we give a general technique for converting data structures with canonical representations in a pure pointer machine model into RAM data structures of comparable performance and that are SHI with high probability. Thus we develop the last ingredient necessary to efficiently implement a host of SHI data structures on a RAM.

This research is supported by NSF ITR grants CCR-0122581 (The Aladdin Center) and IIS-0121678.

Keywords: data structures, hash table, history independence, unique representation, canonical representation, stable marriage, oblivious data structures

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. To address these concerns, the notion of *history independent* data structures was devised. Roughly, a data structure is history independent if someone with access to the memory 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.

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. In addition to the SHI filesystem mentioned above, one could 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 a of 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 the system crashes, then while rerunning the code on the same input we can be sure that the same memory layout is generated, even if the exact interleaving of the parallel operations is different the second time due to nondeterministic timing issues.

Previous Work. The precursor notion of *oblivious* data structures, where the *pointer structure* of a data structure reveals nothing about its history, was first studied by Micciancio [13]. The two main notions of history independence, which we formally define below, were advanced by Naor and Teague [14], and further studied by Hartline *et al* [9]. Informally, we say a data structure has a *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. Though this is not the definition provided by Naor and Teague, Hartline *et al* [9] prove that it is equivalent to their definition for reversible data structures, where a data structure is said to be reversible if any state is reachable from any other state via a sequence of legal operations. In the case of a hash table, the state can be represented by the set of (key, object) pairs stored in the hash table.

Long before Naor and Teague's work on history independence, 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 require only $O(\log n)$ pair-wise independent hash functions rather than a truly random hash function. Naor and Teague also give a WHI hash table that supports deletion, is space efficient, and takes expected amortized $O(1)$ time for insertions and deletions, and $O(1)$ time for search.

Hartline *et al* [9] gave an alternate characterization of SHI data structures, considered dynamically resizable data structures which are efficient against a non-oblivious adversary selecting the sequence of operations, and prove upper bounds for WHI data structures and lower bounds for SHI data structures.

Buchbinder and Petrank [6] studied the time complexity of WHI and SHI heaps and queues in a comparison based model of computation. They show a large complexity separation between WHI and SHI versions of these data structures.

Snyder [20] considered the problem of building uniquely represented dictionaries in a graph storage model. He proved that in order to uniquely represent a dictionary on n items with a "tree-like" graph of bounded degree, $\Theta(\sqrt{n})$ time per insertion, deletion, and search is both necessary and sufficient. In a similar model, Sundar and Tarjan [21] derive a bound of $\Theta(\sqrt{n})$ time per operation for the task of representing a dictionary with a unique binary search tree. Andersson and Ottmann [3] built on the work of Snyder, and considered uniquely representing a dictionary with graphs of bounded outdegree, and derive a matching lower and upper bound of $\Theta(n^{1/3})$ time per operation in the worst case.

Acar *et al* [1] devised a model and algorithms to automatically *dynamize* static algorithms. That is, they show how to run an off-line algorithm in an on-line environment by maintaining enough state so that as the input changes the algorithm can exploit its earlier work as much as possible rather than restarting the computation from scratch each time. Acar *et al* gave sufficient conditions for strong history independence of dynamized algorithms, as well as a framework for analyzing their performance. As an example, they gave an efficient SHI data structure for dynamic trees.

Our Contributions. We provide the first strongly history independent hash table with the following guarantees:

Theorem 1. *There exists a SHI hash table that can be stored in $(1+\epsilon)n$ slots of space such that the expected time to perform any search, insertion, or deletion is $O(1/\epsilon^2)$ given the hash table has at most n elements.*

Such a hash table can serve as a basis for mapping many oblivious data structures (see above) onto a SHI RAM data structure. For those oblivious data structures in which the user can give each node a unique and compact label this can simply be done by storing the node in the hash table using the node label. In Section 4 we also present a general technique that makes any oblivious data structure implemented on a pure pointer machine SHI with high probability [1], while preserving the space efficiency and time efficiency (up to amortization and expectation) of the original. This result does not require the user to define labels for the nodes since it constructs the labels itself.

The hash table requires a $\Theta(\log n)$-universal[2] hash function. Using the construction of Siegel [19], this hash function uses $O(n^{\Theta(1/t)})$ random bits, where $t = O(1)$ is the (user specified) time to evaluate the function. In total the hash table thus requires $O(n^{\delta})$ random bits, for arbitrarily small constant $\delta > 0$.

Note we do not allow the observer to decide what operations are performed when giving these guarantees. Indeed, since we allow the observer to inspect the random bits in our hash table, 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 independently from the random bits used by the hash table, the hash table operations will take constant time in expectation.

2 Preliminaries

For convenience, we define $[k] := \{0, 1, 2, \ldots, k-1\}$.

We consider the problem of hashing $n = \alpha p$ (key, object) pairs into a table of length 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 model with word size at least $\log |U|$. Thus, a key can be stored in a single word. Lastly, we assume a $c \log n$-universal hash function $h : U \to [p]$, for sufficiently large c, that can be evaluated in constant time. The hash functions of Siegel [18, 19] and Östlin and Pagh [15] are suitable implementations, assuming the keys are integers.

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 [14], however the two definitions were proved equivalent by Hartline *et al* [9] for reversible data structures, which include hash tables that support deletion.

Definition 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.*

[1] Event ε occurs with high probability if $\forall c \in \mathbb{N}$, there is some instantiation of constants in the $O(\cdot)$ terms such that $\mathbf{Pr}[\varepsilon] \geq 1 - n^{-c}$.

[2] See definition 4.

The distribution spoken of is over the random bits that the WHI data structure is allowed to hide from the observer.

Definition 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.*

We also make use of the following definitions.

Definition 3 (Space Overhead). *A hash table for n keys has space overhead ω if it occupies ωn words of memory.*

Definition 4 (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 $\mathcal{H}$.

Note that building an efficient SHI hash table based on separate chaining has the difficulty that it requires a SHI memory allocator, which is precisely the problem we have set for ourselves. We therefore focus on open-address hash schemes.

3 The Hash Table

Our approach is based on exploiting an interesting property of the stable marriage algorithm of Gale and Shapley [7], stated below in Theorem 2. 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 in $M \cup W$, find a *stable* matching $E \subset M \times W$ of size n. Here, a man's preference list is a permutation over W, a woman's preference list is a permutation over M, and persons appearing earlier in the permutation are considered preferable to those appearing later in the permutation. 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'; such a case is called an *instability*.

The Gale-Shapley stable marriage algorithm proceeds as follows. At all times, each person is labeled *single* or *dating*. Initially everyone is labeled single. Each man has a list of women who have rejected him and a woman called his *next prospective mate*, which is the woman he most desires among those who have not yet rejected him. Additionally, each woman has a man called her *current mate*, which is initially null.

While there exists a single man m, m *proposes* to his next prospective mate w. If w is single, they date, by which we mean m and w are labeled dating, and w's current mate is set to m. If w is dating, w compares m to her current mate m'. If she prefers m' to m, she rejects m. If she prefers m to her current mate m', she rejects m' and dates m, in which case m' is labeled free, m is labeled dating, and w's current mate is set to m. When the while loop terminates, the output consists of each woman matched with her current mate.

Note that the algorithm is underspecified in the sense that there may be many single men to select among at the beginning of some executions of the while loop. Thus, there are many different valid executions of this algorithm, corresponding to various ways of selecting among single men. Nevertheless, the following theorem, implicit in [7] and treated explicitly in [11], shows the outcome is not affected by the choice of valid execution.

Theorem 2 ([7]). *Every execution of the Gale-Shapley algorithm results in the same stable matching.*

3.1 Framework

Theorem 2 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 2 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 for all $\mathbf{k}$ and i we can compute the i^{th} slot in $\mathbf{k}$'s preference list in constant time, for each key $\mathbf{k}$ and slot x we can compute x's rank in $\mathbf{k}$'s preference list, and for all slots x, we can compare any two keys with respect to x's preference list in constant time.

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 $\mathbf{next}(x)$.

Psuedocode for insertion and deletion are given in figure 3.1. In the psuedocode, $\mathbf{probe}(\mathbf{k}, i)$ is the i^{th} slot in $\mathbf{k}$'s probe sequence, A is the array of the hash table, and $\mathbf{rank}(\mathbf{k}, x) = i$ if x is the i^{th} slot in $\mathbf{k}$'s probe sequence.

```
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) then 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){
    Find the slot x such that A[x] = k. If no such slot exists, 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. See section 3.1 for definitions of $\mathbf{probe}(\cdot)$, $\mathbf{rank}(\cdot)$, and $\mathbf{next}(\cdot)$.

3.2 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.

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 $\Theta(\log n)$-universal hash function $h : U \to [p]$ that can be evaluated in constant time. The functions

$$\begin{aligned} \mathbf{probe}(\mathbf{k}, i) &:= (h(\mathbf{k}) + i) \mod p \\ \mathbf{rank}(\mathbf{k}, x) &:= (x - h(\mathbf{k})) \mod p \end{aligned}$$

can both be computed in constant time.

Search proceeds in a fashion similar to a standard linear probe hash table. Specifically, we try $\mathbf{probe}(\mathbf{k}, i)$ 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 $\mathbf{probe}(\mathbf{k}, i)$, so we can report that $\mathbf{k}$ is not present.

Deletions are slightly more involved. We supply the psuedocode for $\mathbf{next}(\cdot)$ in figure 3.2. In the linear probe implementation, $\mathbf{next}(x)$ is the slot x' containing the key $\mathbf{k}'$ that probed x but was rejected (or displaced) in favor of another key. Thus $\mathbf{k}'$ residing in slot x' must satisfy $\mathbf{rank}(\mathbf{k}', x) < \mathbf{rank}(\mathbf{k}', x')$. Furthermore, $\mathbf{k}'$ must be the most preferred key (according to x) that satisfies this condition. Since all slot preference lists are the same, $\mathbf{next}(x)$ is the slot that minimizes $(y - x) \mod p$ from among all the slots that contain keys $A[y]$ satisfying $\mathbf{rank}(A[y], x) < \mathbf{rank}(A[y], y)$. This x' is exactly what the code in figure 3.2 computes.

```
next(slot x){
    Set x' = (x + 1) mod (p);
    While (A[x'] not empty){
        If (rank(A[x'], x) < rank(A[x'], x')) then return x';
        Set x' = (x' + 1) mod (p);
    }
    return null;
}
```

Figure 2: Psuedocode for $\mathbf{next}(\cdot)$ in the linear probe based implementation.

Remark: Using a k-universal hash function h, for any $k \geq 1$, would result in a SHI hash table that takes only a constant times longer than a standard implementation of a linear-probing-based hash table using a k-universal hash function. In particular, $k = 2$ is often considered adequite in practice, in which case the SHI version would use $O(\log n)$ random bits and achieve similiar performance. We set $k = \Theta(\log n)$ only to guarantee expected constant time operations.

3.3 Canonical Memory Representation

We first prove that any hash table following the framework described in section 3.1 has slot contents that are canonical up to randomness.

Theorem 3. *For any hash table following the framework described in section 3.1, after the random bits have been fixed there is a unique representation of the slots array for each set of $p-1$ or fewer keys.*

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 2 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 $\mathbf{rank}(\mathbf{k}, w) < \mathbf{rank}(\mathbf{k}, w')$, 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 on this instance. A slot's current mate is simply the key it currently holds. A key's next prospective mate is the first slot in its probe sequence that it has not probed already. A key may probe the same slot multiple times, however if it was rejected by a slot once, all subsequent probes to that slot result in rejection, since the slot's current mate m can only be replaced by a key it prefers more than m, and preference is transitive. When selecting among single keys, the insertion selects the unique key that was just evicted from a slot, if it exists, and otherwise selects the next key slated for insertion in the instruction sequence.

Before applying Theorem 2, we need to reduce the instance size so that $|M| = |W|$. However, it is relatively easy to see that we can safely ignore those slots unmatched in $\{(\mathbf{k}, s(\mathbf{k}))|\mathbf{k} \in S\}$ and thus reduce the problem to one in which $|M| = |W|$.

Finally, consider deletions. WLOG, we can assume that only keys present in the hash table are ever deleted, since calling delete($\mathbf{k}$) on a key $\mathbf{k}$ not in the hash table has no effect on the slot array. Note that we need to show only that any single delete operation maintains the unique representation of the slots array as a function of the keys stored in the hash table. To see this, take any sequence of operations $\rho = (\rho_1, \rho_2, \ldots, \rho_k)$. Remove all searches from ρ, as they do not affect the slot array. Consider the first delete operation in ρ, say $\rho_i = \text{delete}(\mathbf{k}_0)$. By assumption, we can safely remove all operations of the form $\rho_j = \text{insert}(\mathbf{k}_0)$ that come before ρ_i, as well as ρ_i. The resulting sequence ρ' results in the same slot array as ρ, but has one fewer deletion. Thus we can induct on the number of deletions.

We will now show that any single delete operation maintains the unique representation of the slots array. Fix any sequence of operations ρ with exactly one deletion. WLOG, we can assume that ρ has no searches, the deletion is the last operation in ρ, no key is inserted more than once under ρ, and the key to be deleted was the last key to be inserted. That is, $\rho = (\text{insert}(\mathbf{k}'_1), \text{insert}(\mathbf{k}'_2), \ldots, \text{insert}(\mathbf{k}'_r), \text{delete}(\mathbf{k}'_r))$. We will show that delete($\mathbf{k}'_r$) exactly undoes all the changes 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 $A[x]$." When inserting $\mathbf{k}_0$, we suppose that $\mathbf{k}_i$ displaces $\mathbf{k}_{i+1}$ for all $i \in \{0, 1, 2, \ldots d\}$. Let $s(\mathbf{k})$ be the slot containing $\mathbf{k}$ immediately after the operation insert($\mathbf{k}_0$) in ρ, and $s'(\mathbf{k})$ be the slot containing $\mathbf{k}$ immediately before the operation insert($\mathbf{k}_0$) in ρ. It is easy to see that $s(\mathbf{k}) = s'(\mathbf{k})$ for all $\mathbf{k} \notin \{\mathbf{k}_0, \mathbf{k}_1, \ldots, \mathbf{k}_d\}$, and $s(\mathbf{k}_i) = s'(\mathbf{k}_{i+1})$ for all $i \in \{0, 1, \ldots, d-1\}$. Now consider delete($\mathbf{k}_0$). It first finds $x = s(\mathbf{k}_0) = s'(\mathbf{k}_1)$. It then repeatedly sets $A[x]$ to $A[\mathbf{next}(x)]$ and sets x to $\mathbf{next}(x)$ while $\mathbf{next}(x)$ exists. To ensure this is the desired behavior, we must show that $\mathbf{next}(s(\mathbf{k}_i)) = s(\mathbf{k}_{i+1})$. If this were not the case, and $\mathbf{next}(s(\mathbf{k}_i)) = x' \neq s(\mathbf{k}_{i+1})$ and $x' = s(\mathbf{k}')$ for some key $\mathbf{k}'$ then $s(\mathbf{k}_i)$ must prefer $\mathbf{k}'$ to $\mathbf{k}_{i+1}$. Yet under insertion of $\{\mathbf{k}'_1, \ldots, \mathbf{k}'_{r-1}\}$ both $\mathbf{k}'$ and $\mathbf{k}_{i+1}$ probe $s(\mathbf{k}_i)$, from which we infer that slot $s(\mathbf{k}_i)$ cannot contain $\mathbf{k}_{i+1}$ since it contains a key it prefers at least as much as $\mathbf{k}'$. In other words, this implies $s'(\mathbf{k}_{i+1}) \neq s(\mathbf{k}_i)$, which is a contradiction. Thus delete($\mathbf{k}_0$) moves $\mathbf{k}_i$ from $s(\mathbf{k}_i) = \mathbf{next}(s(\mathbf{k}_{i-1}))$ to $s(\mathbf{k}_{i-1}) = s'(\mathbf{k}_i)$ for all $i \in \{1, 2, \ldots, d\}$. Then, since $\mathbf{next}(s(\mathbf{k}_d))$ is null, delete($\mathbf{k}_0$) clears slot $s(\mathbf{k}_d)$. It leaves all other slots unaffected. Thus, after the deletion, the slot array contains exactly keys $\{\mathbf{k}'_1, \ldots, \mathbf{k}'_{r-1}\}$, each of these keys $\mathbf{k}$ is in slot $s'(\mathbf{k})$, and all slots not containing keys are empty.

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

Corollary 1. *The hash table implementation described above is SHI.*

3.4 Space and Time Complexity

The hash table implementation based on linear probing requires $(1+\epsilon)n$ 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 [3], the expected cost for all operations is $O(1/\epsilon^2)$.

We bound the expected time per operation for our hash table implementation by comparing it to a standard linear-probing-based hash table. Recall that this standard hash table selects a hash function $h(\cdot)$, uses probe sequences $\mathbf{probe}(\mathbf{k}, i) = h(\mathbf{k}) + i \mod (p)$, and resolves all collisions in favor of the key already residing in the contested slot. Schmidt and Siegel [16] build on the work of Knuth [12] to prove that in a hash table of size p storing $n := \alpha p$ keys, if $h(\cdot)$ is $c \log n$-universal for a sufficiently large constant c, then the expected cost of an insertion or search is $O(1/(1-\alpha)^2)$.

Theorem 4. *The linear-probe-based hash table implementation described in Section 3.2 performs searches, insertions, and deletions in expected $O(1/(1-\alpha)^2)$ time, where the hash table has p slots and $n = \alpha p$ keys.*

[3]Note that $1/(1-\alpha) = 1 + 1/\epsilon$. We assume that $\epsilon = O(1)$, so that $1/(1-\alpha) = \Theta(1/\epsilon)$.

Proof. First consider only searches and insertions. Fix a set of keys S of size at most $n-1$. It is easy to see that if the standard table and our 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 Schmidt and Siegel, $\mathbf{E}[t'] = O(\mathbf{E}[t]) = O(1/(1-\alpha)^2)$.

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 $O(1/(1-\alpha)^2)$ time as well.

Now consider deletions. Suppose we insert keys S and then delete key $\mathbf{k} \in S$. We can compute $\mathbf{rank}(\cdot)$ in constant time. Looking at the psuedocode for delete and $\mathbf{next}(\cdot)$, it is easy to prove that 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 insert($\mathbf{k}$) operation takes time $O(d_S^h(\mathbf{k}))$ which is $O(1/(1-\alpha)^2)$ in expectation, so the deletion takes $O(1/(1-\alpha)^2)$ in expectation as well. Note that for a generic hash table 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 can rely on the fact that our hash table is SHI to dispense with this concern. □

3.5 Extensions

Dynamic Resizing. We can dynamically resize the hash table by using the standard technique of doubling the hash table 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 [14, 9].

Parallelism. Note that Theorem 2 implies that any SHI hash table in our framework has a SHI slot array under parallel insertions, so long as there is locking at the slot level. Thus both our implementation and Naor and Teague's can be made SHI under parallel inserts with virtually no additional effort. Our implementation supports parallel deletions (without simultaneous insertions) if prior to deleting $\mathbf{k}$ we obtain locks on all slots $h(\mathbf{k}), h(\mathbf{k})+1, h(\mathbf{k})+2, \ldots, h(\mathbf{k})+l-1$, where $h(\mathbf{k})+l$ is the first empty slot encountered and all arithmetic is modulo p. With high probability $l = O(\log n)$, so this locking still has a reasonably fine granularity.

To handle simultaneous parallel inserts and deletes, we make use of the room synchronizations of Blelloch, Cheng, and Gibbons [5]. This approach has the advantage of guaranteeing that no operation is starved.

4 Implementing Other SHI Data Structures

There are several data structures for which we might like to have efficient SHI implementations. For example, for a SHI filesystem we might like a SHI implementation of a balanced binary search tree. In this section we will show a general technique for converting any oblivious data structure implemented on a pure pointer machine into a RAM data structure which is SHI with high probability. (Recall from the introduction that oblivious data structures are those in which the pointer structure of a data structure reveals nothing about its history.) Our derived RAM data structure will have nearly the same space and time efficiency as the original. This reduces the problem of developing a SHI data structure to developing an oblivious data structure on a pure pointer machine. This applies, for example, to the oblivious trees of Micciancio [13], the treaps of Seidel and Aragon [17], and many of the dynamized algorithms of Acar *et al* [1].

We introduce some terminology before formally stating our result. For our purposes, a *pure pointer machine* is a machine with a finite set of registers, which can create nodes called *cons cells*. Each cons cell has two fields, each of which may store either a pointer to a cons cell or a word of data. Similarly, each register may store either a pointer to a cons cell or a word of data. The machine may do arithmetic on data in registers, but not on pointers. All access to cons cells is through following pointers. The machine is called *pure* because the fields of each cons cell may be set only on creation. Thus all data stored in the graph is immutable as in a purely functional language, and the pointer machine can only create DAGs. See [4] for an explanation of various pointer machines models.

For a state, σ, of the pointer machine, let $r_i(\sigma)$ be the contents of register i in state σ, let $G(\sigma)$ be the digraph of cons cells reachable from the registers in state σ, and let $f_i(v, \sigma)$ be the value in the i^{th} field of cons cell $v \in V[G(\sigma)]$ in state σ. Let $F_D(\sigma)$ and $F_C(\sigma)$ denote all fields of vertices in $G(\sigma)$ storing data and pointers to cons cells, respectively. For a pointer p, let $[p]$ denote the object it points to. Two states of the pointer machine, σ_1 and σ_2, are said to be *structurally equal* if for all registers i storing data in σ_1, $r_i(\sigma_1) = r_i(\sigma_2)$, and there exists an isomorphism $g : V[G(\sigma_1)] \to V[G(\sigma_2)]$ such that for all registers i storing pointers to cons cells in σ_1, $g([r_i(\sigma_1)]) = [r_i(\sigma_2)]$ and for all fields $f_j(v, \sigma_1) \in F_D(\sigma_1)$, $f_j(v, \sigma_1) = f_j(g(v), \sigma_2)$, and for all fields $f_j(v, \sigma_1) \in F_C(\sigma_1)$, $f_j(v, \sigma_1) = v' \iff f_j(g(v), \sigma_2) = g(v')$.

We are now ready to state the theorem. Our approach is described in the proof.

Theorem 5. *Any algorithm on a pure pointer machine can be mapped on to a RAM such that with high probability the space overhead is constant, constant time operations in the pointer machine take amortized expected constant time in the RAM, and structural equality of two states in the pointer machine implies equality of the memory representation of the two states in the RAM.*

We use the technique of *hash-consing* [8] to obtain this result. Hash-consing is the technique of hashing a cons cell into memory using both of its fields together as the key [8]. Thus if two cons cells have identical contents, they hash to the same location.

Proof of Theorem 5: We first show how to simulate the pointer machine on a RAM. Let C be the set of cons cells, and let D be the set of data values. To reduce the probability of collisions, we assign labels drawn from a large set $\mathcal{L}$ as follows. Select fast random hash functions $\varphi_D : D \to \mathcal{L}$ and $\varphi_{\mathcal{L}} : \mathcal{L}^2 \to \mathcal{L}$. When a cons cell is created, compute its label using $\varphi_{\mathcal{L}}$ applied to the labels associated with its fields. If truly random functions were used, the probability that there exists two cons cells with the same label is $1 - (\prod_{k=0}^{n-1}(|\mathcal{L}| - k))/|\mathcal{L}|^n$, which is less than $n^2/|\mathcal{L}|$. Thus using the hash functions of [15], with a set of labels of size n^c, where $c > 2$ is user specified, with high probability no two cons cells receive the same label. Should any two cons cells receive the same label, we abort.

When simulating the pointer machine algorithm, we simply store labels (rather than pointers) in fields. We can now use these labels as keys when hashing the cons cells into RAM. To maintain history independence, we will have to implement garbage collection, however if we use a reference counting garbage collector (see e.g. [10]) we can easily amortize the cost of garbage collection against the cost of creating the cons cells.

Because the labels are assigned using φ_D and the labels of a cell's children's labels, it is straightforward to prove via structural induction that two structurally equal states σ_1 and σ_2 with isomorphism g satisfy the condition that $v \in V[G(\sigma_1)]$ and $g(v) \in V[G(\sigma_2)]$ receive the same label. Thus they define the same set of keys (with identical auxiliary data) to be stored in the SHI memory allocator, and have identical memory representations in the RAM. □

We conclude that any algorithm implemented on a pure pointer machine that is *oblivious* in the sense that each state has a unique pointer structure can be implemented as an computation in a RAM which with high probability is SHI, has only constant space overhead, and has constant amortized expected slowdown. Most tree structures, including treaps, red-black trees, and oblivious trees can be implemented on a pure pointer machine.

5 Conclusions

We have shown how to build a space efficient, strongly history independent hash table and memory allocator with expected $O(1)$ time per insertion, deletion, and search by exploiting a subtle connection between history independent hashing and the Gale-Shapley stable marriage algorithm. Our performance guarantees are in stark contrast with strong deterministic lower bounds for SHI heaps and queues, which require $\Omega(n)$ time operations [6], and lower bounds for dictionaries in other models of computation [20, 21, 3]. While this proves that the performance cost of SHI memory allocation is not too high on a RAM model of computation, our memory allocator destroys all locality of reference, which will slow down its performance in practice. It is interesting to ask if destroying reference locality is necessary for a SHI memory allocator. Related to this is the challenge of efficiently allocating objects of varying sizes in a SHI manner, without breaking up large objects into small pieces. Another open problem is whether there exists an efficient SHI hash table that uses only polylog(n) random bits.

Lastly, we have shown how to efficiently map data structures in pure pointer machines that have unique structural representations for each state to data structures in a RAM that are SHI with high probability. Is there such a mapping that guarantees the resulting RAM data structure is SHI?

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

References

[1] Umut A. Acar, Guy E. Blelloch, Robert Harper, Jorge L. Vittes, and Shan Leung Maverick 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] Arne Andersson and Thomas Ottmann. Faster uniquely represented dictionaries. In *FOCS: IEEE Symposium on Foundations of Computer Science*, 1991.

[4] Amir M. Ben-Amram. What is a "pointer machine" ? *SIGACT News*, 26(2):88–95, 1995.

[5] Guy E. Blelloch, Perry Cheng, and Phillip B. Gibbons. Scalable room synchronizations. *Theory Comput. Syst.*, 36(5):397–430, 2003.

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

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

[8] Eiichi Goto. Monocopy and associative algorithms in an extended lisp. Technical Report TR 74-03, University of Tokyo, May 1974.

[9] Jason D. Hartline, Edwin S. Hong, Alexander E. Mohr, William R. Pentney, and Emily Rocke. Characterizing history independent data structures. *Algorithmica*, 42(1):57–74, 2005.

[10] Richard Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, July 1996.

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

[12] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1998.

[13] Daniele 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] Moni Naor and Vanessa 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] Anna Östlin and Rasmus 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] Jeanette P. Schmidt and Alan Siegel. The analysis of closed hashing under limited randomness. In *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 224–234, New York, NY, USA, 1990. ACM Press.

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

[18] Alan 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.

[19] Alan 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.

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

[21] Rajamani Sundar and Robert 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.