Lower and Upper Bounds on Obtaining History Independence

Niv Buchbinder and Erez Petrank *

Computer Science Department, Technion, Haifa, Israel,
{nivb,erez}@cs.technion.ac.il

Abstract. History independent data structures, presented by Micciancio, are data structures that possess a strong security property: even if an intruder manages to get a copy of the data structure, the memory layout of the structure yields no additional information on the data structure beyond its content. In particular, the history of operations applied on the structure is not visible in its memory layout. Naor and Teague proposed a stronger notion of history independence in which the intruder may break into the system several times without being noticed and still obtain no additional information from reading the memory layout of the data structure.

An open question posed by Naor and Teague is whether these two notions are equally hard to obtain. In this paper we provide a separation between the two requirements for comparison based algorithms. We show very strong lower bounds for obtaining the stronger notion of history independence for a large class of data structures, including, for example, the heap and the queue abstract data structures. We also provide complementary upper bounds showing that the heap abstract data structure may be made weakly history independent in the comparison based model without incurring any additional (asymptotic) cost on any of its operations. (A similar result is easy for the queue.) Thus, we obtain the first separation between the two notions of history independence. The gap we obtain is exponential: some operations may be executed in logarithmic time (or even in constant time) with the weaker definition, but require linear time with the stronger definition.

Keywords: History independent data-structures, Lower bounds, Privacy, The heap data-structure, The queue data-structure.

1 Introduction

1.1 History independent data structures

Data structures tend to store unnecessary additional information as a side effect of their implementation. Though this information cannot be retrieved via the

* This research was supported by the E. AND J. BISHOP RESEARCH FUND.

'legitimate' interface of the data structure, it can sometimes be easily retrieved by inspecting the actual memory representation of the data structure. Consider, for example, a simple linked list used to store a wedding guest-list. Using the simple implementation, when a new invitee is added to the list, an appropriate record is appended at the end of the list. It can be then rather discomforting if the bride's "best friend" inspects the wedding list, just to discover that she was the last one to be added. History independent data structures, presented by Micciancio [8], are meant to solve such headaches exactly. In general, if privacy is an issue, then if some piece of information cannot be retrieved via the 'legitimate' interface of a system, then it should not be retrievable even when there is full access to the system. Informally, a data structure is called History independent if it yields no information about the sequence of operations that have been applied on it.

An abstract data structure is defined by a list of operations. Any operation returns a result and the specification defines the results of sequence of operations. We say that two sequences S_1, S_2 of operations yield the same content if for any suffix T, the results returned by T operations on the data structure created by S_1 and on the data structure created by S_2 are the same. For the heap data structure the content of the data structure is the set of values stored inside it.

We assume that in some point an adversary gains control over the data structure. The adversary then tries to retrieve some information about the sequence of operations applied on the data structure. The data structure is called History independent if the adversary cannot retrieve any more information about the sequence other than the information obtainable from the content itself.

Naor and Teague [10] strengthen this definition by allowing the adversary to gain control more than once without being noted. In this case, one must demand for any two sequences of operations and two lists of 'stop' points in which the adversary gain control of the data structure, if in all 'stop' points, the content of the data structure is the same (in both sequences), then the adversary cannot gain information about the sequence of operations applied on the data structure other than the information yielded by the content of the data structure in those 'stop' points. For more formal definition of History independent data structure see section 3.

An open question posed by Naor and Teague is whether the stronger notion is harder to obtain than the weaker notion. Namely, is there a data structure that has a weakly history independent implementation with some complexity of operations, yet any implementation of this data structure that provides strong history independence has a higher complexity.

1.2 The heap

The heap is a fundamental data structure taught in basic computer science courses and employed by various algorithms, most notably, sorting. As an abstract structure, it implements four operations: build-heap, insert, remove-max and increase-key. The basic implementations require a worst case time of $O(n)$

for the build-heap operation (on n input values), and $O(\log n)$ for the other three operations[1]. The standard heap is sometimes called *binary heap*.

The heap is a useful data structure and is used in several important algorithms. It is the heart of the *Heap-Sort* algorithm suggested by Williams [12]. Other applications of heap use it as a *priority queue*. Most notable among them are some of the basic graph algorithms: Prim's algorithm for finding Minimum Spanning Tree [11] and Dijkstra's algorithm for finding Single-Source Shortest Paths [4].

1.3 This work

In this paper we answer the open question of Naor and Teague in the affirmative for the comparison based computation model. We start by providing strong and general lower bounds for obtaining strong history independence. These lower bounds are strong in the sense that some operations are shown to require linear time. They are general in the sense that they apply to a large class of data structures, including, for example, the heap and the queue data structures. The strength of these lower bounds implies that strong data independence is either very expensive to obtain, or must be implemented with algorithms that are not comparison based.

To establish the complexity separation, we also provide an implementation of a weakly history independent heap. A weakly history independent queue is easy to construct and an adequate construction appears in [10]. Our result on the heap is interesting in its own sake and constitutes a second contribution of this paper. Our weakly history independent implementation of the heap requires no asymptotic penalty on the complexity of the operations of the heap. The worst case complexity of the build-heap operation is $O(n)$. The worst case complexity of the increase-key operation is $O(\log n)$. The expected time complexity of the operations insert and extract-max is $O(\log n)$, where expectation is taken over all possible random choices made by the implementation in a single operation. This construction turned out to be non-trivial and it requires an understanding of how uniformly chosen random heaps behave. To the best of our knowledge a similar study has not appeared before.

The construction of the heap and the simple implementation of the queue are within the comparison based model. Thus, we get a time complexity separation between the weak and the strong notions of history independent data structure. Our results for the heap and the queue appear in table 1. The lower bound for the queue is satisfied for either the insert-first or the remove-last operations. The upper bounds throughout this paper assume that operations on keys and pointers may be done in constant time. If we use a more prudent approach and consider the bit complexity of each comparison, our results are not substantially affected. The lower bound on the queue was posed as an open question by Naor and Teague.

[1] The more advanced Fibonacci heaps obtain better amortized complexity and seem difficult to be made History independent. We do not study Fibonacci heaps in this paper.

Operation	Weak History Independence	Strong History Independence
heap:insert	$O(\log n)$	$\Omega(n)$
heap:increase-key	$O(\log n)$	$\Omega(n)$
heap:extract-max	$O(\log n)$	No lower bound
heap:build-heap	$O(n)$	$\Omega(n \log n)$
queue: max {insert-first, remove-last}	$O(1)$	$\Omega(n)$

Table 1. Lower and upper bounds for the heap and the queue

1.4 Related work

History independent data structures were first introduced by Micciancio [8] in the context of incremental cryptography. Micciancio has shown how to obtain an efficient History independent 2-3 tree. In [10] Naor and Teague have shown how to implement a History independent hash table. They have also shown how to obtain a history independent memory allocation. Naor and Teague note that all known implementations of strongly independent data structures are canonical. Namely, for each possible content there is only one possible memory layout. A proof that this must be the case has been shown recently by [5] (and independently proven by us). Andersson and Ottmann showed lower and upper bounds on the implementation of unique dictionaries [1]. However, they considered a data structure to be unique if for each content there is only one possible representing graph (with bounded degree) which is a weaker demand than canonical. Thus, they also obtained weaker lower bounds for the operations of a dictionary.

There is a large body of literature trying to make data structures *persistent*, i.e. to make it possible to reconstruct previous states of the data structure from the current one [6]. Our goal is exactly the opposite, that no information whatsoever can be deduced about the past.

There is considerable research on protecting memories. Oblivious RAM [9] makes the address pattern of a program independent on the actual sequence. it incurs a cost of $poly \log n$. However, it does not provide history independence since it assumes that the CPU stores some secret information; this is an inappropriate model for cases where the adversary gains complete control.

1.5 Organization

In section 2 we provide some notations to be used in the paper. In section 3 we review the definitions of History independent data structures. In section 4 we present the first lower bounds for strongly history independence data structures. As a corollary we state lower bounds on some operations of the heap and queue data structures. In section 5 we review basic operations of the heap. In Section 5.1 we present some basic properties of randomized heaps. In section 6 we show how to obtain a weak history independent implementation of the heap data structure with no asymptotic penalty on the complexity of the operations.

2 Preliminaries

Let us set the notation for discussing events and probability distributions. If S is a probability distribution then $x \in S$ denotes the operation of selecting an element at random according to S. When the same notation is used with a set S, it means that x is chosen uniformly at random among the elements of the set S. The notation $Pr\left[R_1; R_2; \ldots; R_k : E\right]$ refers to the probability of event E after the random processes $R_1, \ldots, R_k$ are performed in order. Similarly, $E\left[R_1; R_2; \ldots; R_k : v\right]$ denotes the expected value of v after the random processes $R_1, \ldots, R_k$ are performed in order.

3 History independent data structures

In this section we present the definitions of History independent data structures. An implementation of a data structure maps the sequence of operations to a memory representation (i.e an assignment to the content of the memory). The goal of a history independent implementation is to make this assignment depend only on the content of the data structure and not on the path that led to this content. (See also a motivation discussion in section 1.1 above).

An abstract data structure is defined by a list of operations. We say that two sequences S_1 and S_2 of operations on an abstract data structure yield the same content if for all suffixes T, the results returned by T when the prefix is S_1, are the same as the results returned when the prefix is S_2. For the heap data structure, its content is the set of values stored inside it.

Definition 1. *A data structure implementation is history independent if any two sequences S_1 and S_2 that yield the same content induce the same distribution on the memory representation.*

This definition [8] assumes that the data structure is compromised once. The idea is that, when compromised, it "looks the same" no matter which sequence led to the current content. After the structure is compromised, the user is expected to note the event (e.g., his laptop was stolen) and the structure must be re-randomized.

A stronger definition is suggested by Naor and Teague [10] for the case that the data structure may be compromised several times without any action being taken after each compromise. Here, we demand that the memory layout looks the same at several points, denoted *stop points* no matter which sequences led to the contents at these points. Namely, if at ℓ stop points (break points) of sequence σ the content of the data structure is $C_1, C_2, \ldots, C_\ell$, then no matter which sequences led to these contents, the memory layout joint distribution at these points must depend only on the contents $C_1, C_2, \ldots, C_\ell$. The formalization follows.

Definition 2. *Let S_1 and S_2 be sequences of operations and let $P_1 = \{i^1_1, i^1_2, \ldots i^1_l\}$ and $P_2 = \{i^2_1, i^2_2, \ldots i^2_l\}$ be two list of points such that for all $b \in \{1, 2\}$ and*

$1 \leq j \leq l$ we have that $1 \leq i_j^b \leq |S_b|$ and the content of data structure following the i_j^1 prefix of S_1 and the i_j^2 prefix of S_2 are identical. A data structure implementation is strongly history independent if for any such sequences the distributions of the memory representations at the points of P_1 and the corresponding points of P_2 are identical.

It is not hard to check that the standard implementation of operations on heaps is not History independent even according to definition 1.

4 Lower bounds for strong history independent data structures

In this section we provide lower bounds on strong history independent data structures in the comparison based model. Naor and Teague noted that all implementations of strong history independent data structure were canonical. In a canonical implementation, for each given content, there is only one possible memory layout. It turns out that this observation may be generalized. Namely, all implementations of (well-behaved) data structure that are strongly independent, are also canonical. This was recently proven in [5] (and independently by us). See section 4.1 below for more details. For completeness, we include the proof in [2].

We use the above equivalence to prove lower bounds for canonical data structures. In subsection 4.2 below, we provide lower bounds on the complexity of operations applied on a canonical data structures in the comparison based model. We may then conclude that these lower bounds hold for strongly history independent data structures in the comparison based model.

4.1 Strong history independence implies canonical representation

Canonical representation is implied by strongly history independent data structures for well-behaved data structures. We start by defining well-behaved data structures, via the *content graph* of the structure. Let C be some possible content of an abstract data-structure. For each abstract data-structure we define its *content graph* to be a graph with a vertex for each possible content C of the data structure. There is a directed edge from a content C_1 to a content C_2 if there is an operation OP with some parameters that can be applied on C_1 to yields the content C_2. Notice that this graph may contain an infinite number of nodes when the elements in the data-structure are not bounded. It is also possible that some vertices have an unbounded degree. We say that a content C is *reachable* there is a sequence of operations that may applied on the empty content and yield C. For our purposes only reachable nodes are interesting. In the sequel, when we refer to the content graph we mean the graph induced by all reachable nodes.

We say that an abstract data structure is *well-behaved* if its content graph is strongly connected. That is, for each two possible contents C_i, C_j, there exists a finite sequence of operations that when applied on C_i yields the content C_j. We may now phrase the equivalence between the strong history independent definition and canonical representations. This lemma appears in [5] and was proven independently by us. For completeness, we include the proof in the full version [2]

Lemma 1. *Any strongly history independent implementation of a well-behaved data-structure is canonical, i.e., there is only one possible memory representation for each possible content.*

4.2 Lower bounds on Comparison based data structure implementation

We now proceed to lower bounds on implementations of canonical data structures. Our lower bounds are proven in the *comparison based* model. A *comparison based* algorithm may only compare keys and store them in memory. That is, the keys are treated by the algorithm as 'black boxes'. In particular, the algorithm may not look at the inner structure of the keys, or separate a key into its components. Other then that the algorithm may, of-course, save additional data such as pointers, counters etc. Most of the generic data-structure implementations are comparison based. An important data structure that is implemented in a non-comparison-based manner is hashing, in which the value of the key is run through the hash function to determine an index. Indeed, for hashing, strongly efficient history independent implementations (which are canonical) exist and the algorithms are not comparison based [10]. Recall that we call an implementation of data structure *canonical* if there is only one memory representation for each possible content.

We assume that a data structure may store a set of keys whose size is unbounded $k_1, k_2, \ldots, k_i, \ldots$. We also assume that there exists a total order on the keys. We start with a general lower bound that applies to many data structures (lemma 2 below). In particular, this lower bound applies to the heap. We will later prove a more specific lemma (see lemma 3 below) that is valid for the queue, and another specific lemma (lemma 4 below) for the operation build-heap of the heap.

In our first lemma, we consider data structures whose content is the set of keys stored in it. This means that the set of keys in the data structure completely determines its output on any sequence of (legitimate) operations applied on the data structure. Examples of such data structures are: a heap, a tree, a hash table, and many others. However, a queue does not satisfy this property since the output of operations on the queue data structure depends on the order in which the keys were inserted into the structure.

Lemma 2. *Let $k_1, k_2, \ldots$ be an infinite set of keys with a total order between them. Let D be an abstract data structure whose content is the set of keys stored inside it. Let I be any implementation of D that is comparison based and canonical Then the following operations on D*

– $\mathsf{insert}(D, v)$
– $\mathsf{extract}(D, v)$
– $\mathsf{increase\text{-}key}(D, v_1, v_2)$ *(i.e. change the value from* v_1 *to* v_2*)*

require time complexity

1. $\Omega(n)$ *in worst case,*
2. $\Omega(n)$ *amortized time.*

Remark 1. property (ii) implies property (i). We separate them for clarity of the representation.

Proof. We start with the first part of the lemma (worst case lower bound) for the insert operation. For any $n \in \mathbb{N}$, let $k_1 < k_2 < \ldots < k_{n+1} < k_{n+2}$ be $n+2$ keys. Consider any sequence of insert operations inserting n of these keys to D. Since the implementation I is comparison based, and the content of the data structure is the set of keys stored inside it, the keys must be stored in the data structure. Since the implementation I is canonical, then for any such set of keys, the keys must be stored in D in the same addresses regardless of the order in which they were inserted into the data structure. Furthermore, since I is comparison based, then the address of each key does not depend on its value, but only on its order within the n keys in the data structure. Denote by d_1 the address used to store the smallest key, by d_2 the address used to store the second key, and so forth, with d_n being the memory address of the largest key (If there is more than one address used to store a key choose one arbitrarily). By a similar argument, any set of $n+1$ keys must be stored in the memory according to their order. Let these addresses be d'_1, d'_2, $\ldots d'_{n+1}$. Next, we ask how many of these addresses are different. Let Δ be the number of indices for which $d_i \neq d'_i$ for $1 \leq i \leq n$.

Now we present a challenge to the data structure which cannot be implemented efficiently by I. Consider the following sequences of operations applied on an empty data-structure: $S = \mathsf{insert}(k_2), \mathsf{insert}(k_3) \ldots \mathsf{insert}(k_{n+1})$. After this sequence of operations k_i must be located in location d_{i-1} in the memory. We claim that at this state either $\mathsf{insert}(k_{n+2})$ or $\mathsf{insert}(k_1)$ must move at least half of the keys from their current location to a different location. This must take at least $n/2 = \Omega(n)$ steps.

If $\Delta > n/2$ then we concentrate on $\mathsf{insert}(k_{n+2})$. This operation must put k_{n+2} in address d'_{n+1} and must move all keys k_i $(2 \leq i \leq n+1)$ from location d_{i-1} to location d'_{i-1}. There are $\Delta \geq n/2$ locations satisfying $d_{i-1} \neq d'_{i-1}$ and we are done. Otherwise, if $\Delta \leq n/2$ then we focus on $\mathsf{insert}(k_1)$. This insert must locate k_1 in address d'_1 and move all keys k_i, $2 \leq i \leq k+1$ from location d_{i-1} to location d'_i. For any i satisfying $d_{i-1} = d'_{i-1}$, it holds that $d_{i-1} \neq d'_i$ (since d'_i must be different from d'_{i-1}). The number of such cases is $n - \Delta \geq n/2$. Thus, for more than $n/2$ of the keys we have that $d_i \neq d'_{i+1}$, thus the algorithm must move them, and we are done.

To show the second part of the lemma for insert, we extend this example to hold for an amortized analysis as well. We need to show that for any integer $\ell \in \mathbb{N}$, there exists a sequence of ℓ operations that require time complexity

$\Omega(n \cdot \ell)$. We will actually show a sequence of ℓ operations each requiring $\Omega(n)$ steps. We start with a data structure containing the keys $l+1, l+2, \ldots, l+n+1$. Now, we repeat the above trick ℓ times. Since there are at least ℓ keys smaller than the smallest key in the structure, the adversary can choose in each step between entering a key larger than all the others or smaller than all the keys in the data structure.

The proof for the extract operation is similar. We start with inserting $n+1$ keys to the structure and then extract either the largest or the smallest, depending on Δ. Extracting the largest key cause a relocation of all keys for which $d'_i \neq d_i$. Extracting the smallest key moves all the keys for which $d_i = d'_i$. One of them must be larger than $n/2$. The second part of the lemma may be achieved by inserting $n + \ell$ keys to the data structure, and then run ℓ steps, each step extracting the smallest or largest value, whichever causes relocations to more than half the values.

Finally, we look at increase-key. Consider an increase-key operation that increases the smallest key to a value larger than all the keys in the structure. Since the implementation is canonical this operation should move the smallest key to the address d_n and shift all other keys from d_i to d_{i-1}. Thus, n relocations are due and a lower bound of n steps is obtained. To show the second part of the lemma for increase-key we may repeat the same operation ℓ times for any $\ell \in \mathbb{N}$.

We remark that the above lemma is tight (up to constant factors). We can implement a canonical data structure that keeps the keys in two arrays. The $n/2$ smaller keys are sorted bottom up at the first array and the other $n/2$ keys are sorted from top to bottom in the other array. Using this implementation, inserting or extracting a key will always move at most half of the keys.

We now move to showing a lower bound on a canonical implementation of the queue data structure. Note that lemma 2 does not hold for the queue data structure since its content is not only the set of values inside it. Recall that a queue has two operations: insert-first and remove-last.

Lemma 3. *In any comparison based canonical implementation of a queue either* insert-first *or* remove-last *work in $\Omega(n)$ worst time complexity. The amortized complexity of the two operations is also $\Omega(n)$.*

Proof. Let $k_1 < k_2 < \ldots < k_{n+1}$ be $n+1$ keys. Consider the following two sequences of operations applied both on an empty queue: $S_1 =$ insert-first(k_1), insert-first$(k_2) \ldots$ insert-first(k_n) and $S_2 =$ insert-first(k_2), insert-first$(k_3) \ldots$ insert-first(k_{n+1}). Since the implementation is comparison based it must store the keys in the memory layout in order to be able to restore them. Also, since the implementation is comparison based, it cannot distinguish between the two sequences and as the implementation is also canonical the location of each key in the memory depends only on its order in the sequence. Thus, the address (possibly more than one address) of k_1 in the memory layout after running the first sequence must be the same as the address used to store k_2 in the second sequence. In general, the address used to store k_i in the first sequence is the same as the address used to store the key k_{i+1} in the second sequence. This means that after running sequence S_1, each of the keys $k_2, k_3, \ldots, k_n$ must reside in a different location than its location after running S_2.

Consider now two more operations applied after S_1: insert-first(k_{n+1}), remove-last (i.e., remove k_1). The content of the data structure after these two operations is the same as the content after running the sequence S_2. Thus, their memory representations must be the same. This means that $n-1$ keys (i.e $k_2, k_3, \ldots, k_n$) must have changed their positions. Thus, either insert or remove-last operation work in worst time complexity of $\Omega(n)$. This trick can be repeated l times showing a series of insert and remove-last such that each pair must move $\Omega(n)$ keys resulting in the lower bound on the amortized complexity.

Last, we prove a lower bound on the build-heap operation in a comparison based implementation of the heap.

Lemma 4. *For any comparison based canonical implementation of a heap the operation* build-heap *must perform $\Omega(n \log n)$ operations.*

Proof. Similarly to sorting, we can view the operation of build-heap in terms of a decision tree. Note that the input may contain any possible permutation on the values $k_1, \ldots, k_n$ but the output is unique: it is the canonical heap with $k_1, \ldots, k_n$. The algorithm may be modified to behave in the following manner: first, run all required comparisons between the keys (the comparisons can be done adaptively), and then, based on the information obtained, rearrange the input values to form the canonical heap.We show a lower bound on the number of comparisons. Each comparison of keys separates the possible inputs to two subsets: those that agree and those that disagree with the comparison made. By the end of the comparisons, each of the $n!$ possible inputs must be distinguishable from the other inputs. Otherwise, the algorithm will perform the same rearrangement on two different inputs, resulting in two different heaps. Thinking of the comparisons as a decision tree, we note that the tree must contain at least $n!$ leaves, each representing a set with a single possible input. This means that the height of the decision tree must be $\Omega(\log(n!)) = \Omega(n \log n)$ and we are done.

4.3 Translating the lower bounds to strong history independence

We can now translate the results of section 4.2 and state the following lemmas:

Lemma 5. *Let D be a well behaved data structure for which its content is the values stored inside it. Let I be any implementation of D which is comparison based and strongly history independent. Then the following operations on D*

– insert(D, v)
– extract(D, v)
– increase-key(D, v_1, v_2) *(i.e. change the value from v_1 to v_2)*

require time complexity

1. *$\Omega(n)$ in worst case,*

2. *$\Omega(n)$ amortized time.*

Proof. The lemma follows directly from lemma 2 and 1.

Corollary 1. *For any strongly history independent comparison based implementation of the heap data structure, the operations* insert *and* increase-key *work in $\Omega(n)$ amortized time complexity. The time complexity of the* build-heap *operation is $\Omega(n \log n)$.*

Proof. The lower bounds on insert and increase-key follow from lemma 5. This is true since the content of the heap data structure is the keys stored inside it and the heap abstract data structure is well behaved. The lower bound on the build-heap operation follows directly from lemma 4 and 1.

Lemma 6. *For any strong history independent comparison based implementation of the queue data structure the worst time complexity of either* insert-first *or* remove-last *is $\Omega(n)$. Their amortized complexity is $\Omega(n)$.*

Proof. The lemma follows directly from lemma 3 and 1.

5 The heap

In this section we review the basics of the heap data structure and set up the notation to be used in the rest of this paper. A good way to view the heap, which we adopt for the rest of this paper, is as an almost full binary tree condensed to the left. Namely, for heaps of $2^\ell - 1$ elements (for some integer ℓ), the heap is a full tree, and for sizes that are not a power of two, the lowest level is not full, and all leaves are at the left side of the tree. Each node in the tree contains a value. The important property of the heap-tree is that for each node i in the tree, its children contain values that are smaller or equal to the value at the node i. This property ensures that the maximal value in the heap is always at the root. Trees of this structure that satisfy the above property are denoted *well-formed heaps*. We denote by $parent(i)$ the parent of a node i and by v_i the content of node i. In a tree that represents a heap, it holds that for each node except for the root:

$$v_{parent(i)} \geq v_i$$

We will assume that the heap contains distinct elements, $v_1, v_2, \ldots, v_n$. Previous work (see [10]) justified using distinct values by adding some total ordering to break ties. In general, the values in the heap are associated with some additional data and that additional data may be used to break ties. The nodes of the heap will be numbered by the integers $\{1, 2, \ldots, n\}$, where 1 is the root 2 is the left child of the root 3 is the right child of the root etc. In general the left child of node i is node $2i$, and the right child is node number $2i + 1$. We denote the number of nodes in the heap H by $size(H)$ and its height by $height(H)$.

We will denote the rightmost leaf in the lowest level *the last leaf.* The position next to the last leaf, where the next leaf would have been had there been another

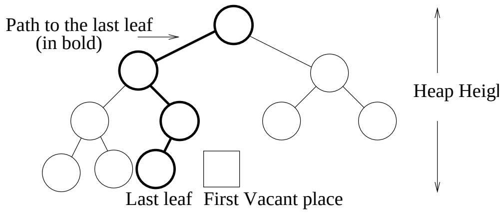

Fig. 1. The height of this heap is 4. The path to the last leaf is drawn in bold. In this example, the path to the first vacant place is the same except for the last edge.

value, is called *the first vacant place*. These terms are depicted in figure 1. Given a heap H and a node i in the heap, we use H^i to denote the sub-heap (or subtree) containing the node i and all its descendants. Furthermore, the sub-heap rooted by the left child of i is denoted H_L^i and the sub-heap rooted by the right child is denoted H_R^i The standard implementation of a heap is described in [3].

5.1 Uniform heaps and basic machinery

In this section we investigate some properties of randomized heaps and present the basic machinery required for making heaps History independent. One of the properties we prove in this section is that the following distributions are equal on any given n distinct values $v_1, \ldots, v_n$.

Distribution Ω_1: Pick uniformly at random a heap among all possible heaps with values $v_1, \ldots, v_n$.

Distribution Ω_2: Pick uniformly at random a permutation on the values $v_1, \ldots, v_n$. Place the values in an (almost) full tree according to their order in the permutation. Invoke build-heap on the tree.

Note that the shape of a size n heap does not depend on the values contained in the heap. It is always the (almost) full tree with n vertices. The distributions above consider the placement of the n values in this tree.

In order to investigate the above distributions, we start by presenting a procedure that inverts the build-heap operation (see [3] and [2] for the definition of build-heap). Since build-heap is a many-to-one function, the inverse of a given heap is not unique. We would like to devise a randomized inverting procedure build-heap$^{-1}(H)$ that gets a heap H of size n as input and outputs a uniformly chosen inverse of H under the function build-heap. Such an inverse is a permutation π of the values $v_1, \ldots, v_n$ satisfying build-heap$(v_{\pi(1)}, \ldots, v_{\pi(n)}) = H$. It turns out that a good understanding of the procedure build-heap^{-1} is useful both for analyzing History independent heaps and also for the actual construction of its operations.

Recall that the procedure build-heap invokes the procedure heapify repeatedly in a bottom-up order on all vertices in the heap. The inverse procedure build-heap^{-1} invokes a randomized procedure heapify^{-1} on all vertices in the heap in

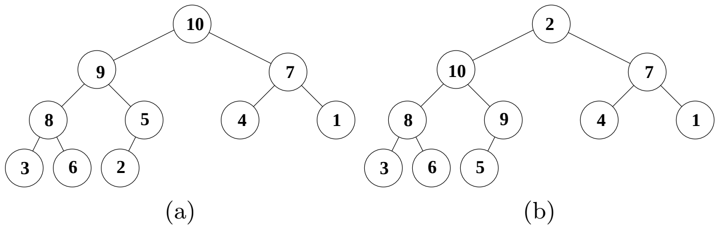

Fig. 2. An example of invoking $\mathsf{heapify}^{-1}(H, 10)$. Node number 10 is the node that contains the value 2. In (b) we can see the output of invoking $\mathsf{heapify}^{-1}$ on the proper heap in (a). The value 2 is put at the root, the path from the root to the father of 2 is shifted down. Note that the two sub-trees in (b) are still well-formed heaps. Applying Heapify on (b) will cause the value 2 at the root to float down back to its position in the original H as in (a)

a top-bottom order, i.e., from the roots to the leaves. We begin by defining the randomized procedure $\mathsf{heapify}^{-1}$. This procedure is a major player in most of the constructions in this paper.

Recall that heapify gets a node and two well-formed heaps as sub-trees of this node and it returns a unified well-formed heap by floating the value of the node down always exchanging values with the larger child. The inverse procedure gets a proper heap H. It returns a tree such that at the root node there is a random value from the nodes in the heap and the two sub-trees of the root are well-formed sub-heaps. The output tree satisfies the property that if we run heapify on it, we get the heap H back. We make the random selection explicit and let the procedure $\mathsf{heapify}^{-1}$ get as input both the input heap H and also the random choice of an element to be placed at the root.

The operation of $\mathsf{heapify}^{-1}$ on input (H, i) is as follows. The value v_i of the node i in H is put in the root and the values in all the path from the root to node i are shifted down so as to fill the vacant node i and make room for the value v at the root. The resulting tree is returned as the output. Let us first check that the result is fine syntactically, i.e., that the two sub-trees of the root are well-formed heaps. We need to check that for any node, but the root, the values of its children are smaller or equal to its own value. For all vertices that are not on the shifted path this property is guaranteed by the fact that the tree was a heap before the shift. Next, looking at the last (lower) node in the path, the value that was shifted into node i is the value that was held in its parent. This value is at least as large as v and thus at least as large as the values at the children of node i. Finally, consider all other nodes on this path. One of their children is a vertex of the path, and was their child before the shift and cannot contain a larger value. The other child was a grandchild in the original heap and cannot contain a smaller value as well.

Claim. Let n be an integer and H be any heap of size n, then for any $1 \leq i \leq n$,

$$\mathsf{heapify}\left(\mathsf{heapify}^{-1}(H, i)\right) = H.$$

```
procedure build-heap^{-1}(H : Heap) : Tree
begin
1.    if (size(H) = 1) then return(H)
2.    Choose a node i uniformly at random among the nodes in the heap H.
3.    H ← heapify^{-1}(H, i)
4.    Return TREE(root(H), build-heap^{-1}(H_L), build-heap^{-1}(H_R))
end
```

Fig. 3. The procedure build-heap$^{-1}(H)$

Proof. Proof omitted (the proof appears in [2]).

An example of invoking heapify$^{-1}(H, i)$ is depicted in figure 2. The complexity of heapify$^{-1}(H, i)$ is linear in the difference between the height of node i and the height of the input heap (or sub-heap), since this is the length of the shifted path. Namely, the complexity of heapify$^{-1}(H, i)$ is $O(height(H) - height(i))$.

Using heapify$^{-1}(H, i)$ we now describe the procedure build-heap$^{-1}(H)$, a randomized algorithm for inverting the build-heap procedure. The output of the algorithm is a permutation of the heap values in the same (almost) full binary tree T underlying the given heap H. The procedure build-heap^{-1} is given in Figure 3. In this procedure we denote by $TREE(root, T_L, T_R)$ the tree obtained by using node "root" as the root and assigning the tree T_L as its left child and the tree T_R as its right child. The procedure build-heap^{-1} is recursive. It uses a pre-order traversal in which the root is visited first (and *heapify*$^{-1}$ is invoked) and then the left and right sub-heaps are inverted by applying build-heap^{-1} recursively.

Claim. For any heap H and for any random choices of the procedure build-heap^{-1},

$$\textsf{build-heap}\left(\textsf{build-heap}^{-1}(H)\right) = H$$

Proof Sketch: The claim follows from the fact that for any i and a heap H, heapify $\left(\textsf{heapify}^{-1}(H, i)\right) = H$, and from the fact that the traversal order is reversed. The heapify operations cancel one by one the heapify^{-1} operations performed on H in the reversed order and the same heap H is built back from the leaves to the root. □

In what follows, it will sometimes be convenient to make an explicit notation of the randomness used by build-heap^{-1}. In each invocation of the (recursive) procedure, a node is chosen uniformly in the current sub-heap. The procedure build-heap^{-1} can be thought of as a traversal of the graph from top to bottom, level by level, visiting the nodes of each level one by one and for each traversed node i, the procedure chooses uniformly at random a node x_i in the sub-heap H^i and invokes heapify$^{-1}(H^i, x_i)$. Thus, the random choices of this algorithm include a list of n choices $(x_1, \ldots, x_n)$ such that for each node i in the heap, $1 \leq i \leq n$, the chosen node x_i is in its sub-tree. The x_i's are independent of the actual values in the heap. They are randomized choices of locations in the heap. Note, for example, that for any leaf i it must hold that $x_i = i$ since there is only one node in the sub-heap H^i. The vector $(x_1, \ldots, x_n)$ is called *proper* if for all i, $i \leq i \leq n$, it holds that x_i is a node in the heap H^i. We will sometimes let the procedure $\mathsf{build\text{-}heap}^{-1}(H)$ get its random choices explicitly in the input and use the notation $\mathsf{build\text{-}heap}^{-1}(H, (x_1, \ldots, x_n))$.

We are now ready to prove some basic lemmas regarding random heaps with n distinct values. In the following lemmas we denote by $\Pi(n)$ the set of all permutations on the values $v_1, v_2 \ldots, v_n$.

Lemma 7. *Each permutation $\pi \in \Pi(n)$ has one and only one heap H and a proper vector $\boldsymbol{X}_n = (x_1, \ldots, x_n)$ such that $(v_{\pi(1)}, \ldots, v_{\pi(n)}) = \mathsf{build\text{-}heap}^{-1}(H, \boldsymbol{X}_n)$.*

Proof. Proof omitted (the proof appears in [2]).

Corollary 2. *If H is picked up uniformly among all possible heaps with the same content then $T = \mathsf{build\text{-}heap}^{-1}(H)$ is a uniform distribution over all $\pi \in \Pi(n)$.*

Proof. As shown, for any permutation π in *support*(H), i.e., a permutation that satisfies $\mathsf{build\text{-}heap}(v_{\pi(1)}, \ldots, v_{\pi(n)}) = H$, there is a unique random vector $(x_1, \ldots, x_n)$, that creates the permutation. Each random (proper) vector has the same probability. Thus, π is chosen uniformly among all permutation in *support*(H). Since H is chosen uniformly among all heaps the corollary follows.

Lemma 8. *Let n be an integer and $v_1, \ldots, v_n$ be a set of n distinct values. Then, for heap H that contains the values $v_1, \ldots, v_n$ it holds that:*

$$Pr\left[\pi \in \Pi(n) : \mathsf{build\text{-}heap}(v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)}) = H\right] = p(H)$$

Where $p(H)$ is a function depending only on n (the size of H). Furthermore, $p(H) = N(H)/|\Pi(n)|$ where $N(H)$ can be defined recursively as follows:

$$N(H) = \begin{cases} 1 & \textit{if } size(H) = 1 \\ size(H) \cdot N(H_L) \cdot N(H_R) & \textit{otherwise} \end{cases}$$

Proof. For any H the probability that $\mathsf{build\text{-}heap}(v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)}) = H$ is the probability that the permutation π belongs to *support*(H). According to lemma 7 the size of *support*(H) is the same for any possible heap H. This follows from the fact that any random vector $(x_1, x_2, \ldots, x_n)$ result in different permutation in *support*(H) and each permutation in *support*(H) has a vector that yield it. The size of *support*(H) is exactly the number of possible random (proper) vectors. This number can be formulated recursively as $N(H)$ depending only on the size of the heap. The probability for each heap now follows.

Corollary 3. *The following distributions Ω_1 and Ω_2 are equal.*

Distribution Ω_1: *Pick uniformly at random a heap among all possible heaps with values $v_1, \ldots, v_n$.*

Distribution Ω_2: *Pick uniformly at random permutation $\pi \in \Pi(n)$ and invoke* $\mathsf{build\text{-}heap}(v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)})$.

Proof. As shown in lemma 8, distribution Ω_2 gives all heaps containing the values $v_1, \ldots, v_n$ the same probability. By definition, this is also the case in Ω_1.

6 Building and maintaining History independent Heap

In this section we present our main theorem and provide some overview of the proof. The full proof appears in [2].

Theorem 1. *There exists a History independent implementation of the heap data structure with the following time complexity. The worst case complexity of the* build-heap *operation is $O(n)$. The worst case complexity of the* increase-key *operation is $O(\log n)$. The expected time complexity of the operations* insert *and* extract-max *is $O(\log n)$, where expectation is taken over all possible random choices made by the implementation.*

Our goal is to provide an implementation of the operations build-heap, insert, extract-max, and increase-key that maintains history independence without incurring an extra cost on their (asymptotic) time complexity. We obtain history independence by preserving the uniformity of the heap. When we create a heap, we create a uniform heap among all heaps on the given values. Later, each operation on the heap assumes that the input heap is uniform and the operation maintains the property that the output heap is still uniform for the new content. Thus, whatever series of operation is used to create the heap with the current content, the output heap is a uniform heap with the given content. This means that the memory layout is history independent and the set of operations make the heap History independent. The rest of the proof describes the implementation for each of the 4 operations. In what follows, we try to highlight the main issues. The full implementation with proof of history independence and time complexity analysis is given in [2].

The operation build-heap This is the simplest since time complexity $O(n)$ is allowed. Given n input values, we permute them uniformly at random and run the (non-oblivious) build-heap. Using the basic machinery from section 5.1, this can be shown to yield a uniformly chosen heap on the input values.

The operation increase-key Here we extend the operation to allow both increasing and decreasing the given value. This extension is useful when implementing the other the operations. To increase a key the standard implementation turns out to work well, i.e., it maintains the uniformity of the distribution on the current heap. To decrease a key, it is enough to use heapify. We show that the scenario is appropriate for the procedure heapify and that uniformity is preserved.

6.1 The operation extract-Max

We start with a naive implementation of extract-max which we call extract-max-try-1. This implementation has complexity $O(n)$. Of-course, this is not an acceptable complexity for the extract-max operation but this first construction will be later modified to make the real History independent extract-Max. The procedure extract-max-try-1 goes as follows. We run build-heap^{-1} on the heap H

procedure extract-max-try-1(H: **Heap**) : **Heap**
begin
1. Choose uniformly at random a proper randomization vector $(x_1, \ldots, x_{n+1})$ for the procedure build-heap^{-1}.
2. $T =$ build-heap$^{-1}(H, (x_1, x_2, \ldots, x_{n+1}))$
3. Let T' be the tree obtained by removing the last node with value v_i from T.
4. H' = build-heap(T')
5. if v_i is the maximum then return (H'). Otherwise:
6. Modify the value at the root to v_i.
7. H'' = heapify(H, 1) (i.e. apply heapify on the root)
8. Return (H'')
end

Fig. 4. The procedure extract-max-try-1

to get a uniform permutation π on the $n+1$ values. Next, we remove the value at the last leaf $v_{\pi(n+1)}$. After this step we get a uniformly chosen permutation of the n values excluding the one we have removed. Next, we run build-heap on the n values to get a uniformly chosen heap among the heaps without $v_{\pi(n+1)}$. If $v_{\pi(n+1)}$ is the maximal value then we are done. Otherwise, we continue by replacing the value at the root (the maximum) with the value $v_{\pi(n+1)}$ and running heapify on the resulting tree to "float" the value $v_{\pi(n+1)}$ down and get a well-formed heap. We will show that this process results in a uniformly chosen heap without the maximum value. The pseudo code of the naive extract-max-try-1 appears in figure 4.

The time consuming lines of the procedure extract-max-try-1 are lines 2,3, and 4 in which we un-build the heap, remove the last leaf and then build it back again. The next step is to note that many of the steps executed are redundant. In particular, build-heap^{-1} runs heapify^{-1} on each node of the heap from top to bottom. Later, after removing the last node, build-heap runs heapify on each node of the heap from bottom to top. These operations cancel each other except for the removed leaf that causes a distortion in the reversed operation. We first claim that it is enough to run heapify^{-1} and heapify on the nodes that belong to the path from the root to the last leaf. Proving this requires some care, but the outcome is a procedure with time complexity $O(\log^2 n)$, since heapify^{-1} and heapify both requires logarithmic time and we need to run them on $O(\log n)$ nodes.

By now, we have an implementation of extract-max requiring worst case time $O(\log^2 n)$. Our last step is to check exactly how the heap is modified by each of these heapify^{-1} and heapify invocations. It turns out that we may save more operations by running these only on vertices "that affect the identity of the last leaf". Identifying these vertices require some technical work, but it turns out that their number equals the height of the vertex that is moved to the last leaf by build-heap^{-1}. Analysis of the expected height of this vertex in our scenario gives a constant number. Thus, on average, we only need to perform heapify^{-1}

and heapify on a constant number of vertices and we are done with operation extract-Max. The details and proofs appear in [2].

6.2 The operation insert

Here, again, we start by providing a procedure insert-try-1 implementing the operation insert with complexity $O(n)$. Again, this allows a construction of a simple and useful implementation that will be improved later. We first choose the location i, $1 \leq i \leq n+1$ to which we insert the new value a. (The choice $i = n+1$ means no insertion.) We put the value a in the node i and remember the value v_i that was replaced at node i. This may yield a tree which is not a well-formed heap because the value a may not "fit" the node i. Hence, we apply increase-key-oblivious on the location i with the new value a. After the new value a is properly placed in the heap, we run build-heap^{-1}. We will show that this yields a uniform permutation of the values $(v_1, v_2, \ldots, v_{i-1}, a, v_{i+1}, \ldots, v_n)$. Now, we add the value v_i at the end of this ordering, getting a uniform permutation on the $n+1$ values $v_1, v_2, \ldots, v_n, a$. Running build-heap on this order of the values yields a random heap on the $n+1$ values.

Then, we scrutinize this simple procedure to identify all redundant steps. Indeed, we are able to obtain a modified procedure that runs in expected time $O(\log n)$ and outputs the same distribution as this simple procedure. The details and proofs appear in [2].

References

1. A. Andersson, T. Ottmann. Faster Uniquely Represented Dictionaries. Proc. 32nd IEEE Sympos. Foundations of Computer Science, pages 642–649, 1991
2. Niv Buchbinder, Erez Petrank. Lower and Upper Bounds on Obtaining History Independence http://www.cs.technion.ac.il/∼erez/publications.html
3. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. Introduction to Algorithms. MIT Press and McGraw-Hill Book Company, 6th edition, 1992.
4. E.W.Dijkstra A note on two problems in connexion with graphs. Numerische Mathematik, 1:269-271, 1959
5. J.D. Hartline, E.S. Hong, A.E. Mohr, W.R. Pentney, and E.C. Rocke. Characterizing History independent Data Structures. ISAAC 2002 pp. 229-240, 2002.
6. J.R.Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan. Making data structures persistent. Journal of Computer and System Sciences, 38(1):86-124, 1989.
7. R.W. Floyd. Algorithm 245 (TREESORT). Communications of the ACM, 7, 1964.
8. D. Micciancio. Oblivious data structures: Applications to cryptography. In Proc. 29th ACM Symp. on Theory of computing, pages 456-464, 1997.
9. O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. Journal of the ACM, 43(3):431-473, 1996.
10. M.Naor and V.Teague. Anti-persistence: History Independent Data Structures. Proc. 33rd ACM Symp. on Theory of Computing, 2001.
11. R.C.Prim. Shortest connection networks and some generalizations. Bell System Technical Journal, 36:1389-1401, 1957
12. J.W.J Williams Algorithm 232 (HEAPSORT). Communication of the ACM, 7:347-348, 1964