Lower and Upper Bounds on Obtaining History Independence [⋆]

Niv Buchbinder [a] and Erez Petrank [a,*]

[a] *Computer Science Department, Technion, Haifa 32000, Israel. Phone: +972-4-8294942*

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 history of operations applied on the structure beyond the information obtainable from the content itself. 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.

Key words: History independent data-structures, Lower bounds, Privacy, The heap data-structure, The queue data-structure

[⋆] This research was supported by THE ISRAEL SCIENCE FOUNDATION (grant no. 36/03) and by the E. AND J. BISHOP RESEARCH FUND. An extended abstract of this paper appeared in [3] Crypto 2003, California.

[*] Corresponding author

Email addresses: `nivb@cs.technion.ac.il` (Niv Buchbinder), `erez@cs.technion.ac.il` (Erez Petrank).

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 '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 [9], 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 of operations S_1 and S_2 *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 at 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 history other than the information obtainable from the content itself.

Naor and Teague [11] 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 gains 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 a 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, extract-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 a *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 [13]. 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 [12] and Dijkstra's algorithm for finding Single-Source Shortest Paths [5].

1.3 This work

In this paper we answer the open question of Naor and Teague in the affirmative for the comparison-based computational 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 [11]. Our result

[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

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. The worst case complexity of these two operations is $O(\log^2 n)$. 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 structures. 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.

1.4 Related work

History independent data structures were first introduced by Micciancio [9] in the context of incremental cryptography [2]. Micciancio has shown how to obtain an efficient history independent 2-3 tree. In [11] 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 [6] (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). This demands only that the shape of the data structure without considering the addresses of the elements is the same, which is a weaker demand than canonical layout. 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 [7]. 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 [10] makes the address pattern of a program independent of the program's computation. They showed how to simulate any t steps of a RAM machine with m memory cells, by an oblivious RAM machine with $O(m \cdot \text{poly} \log m)$ memory cells using $tO(t \cdot \text{poly} \log t)$ steps. This result, however, 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 notation 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 independent 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 and 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 notation $x \in_R S$ is used, 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 motivating 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 [9] 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 [11] 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 $P_1 = \{i_1^1, i_2^1, \ldots i_l^1\}$*

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 $1 \leq i^b_j \leq |S_b|$ and the content of data structure following the i^1_j prefix of S_1 and the i^2_j prefix of S_2 are identical[2] *. 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 [6] (and independently by us). See section 4.1 below for more details. For completeness, we include the proof in section A.

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

For well-behaved data structures canonical representation is implied by strongly history independent 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

[2] The two lists of 'stop' points do not have to be ordered, as long as the 'stop' points are consistent with some possible transitions of the abstract data structure. This observation is highlighted in [6]

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* if there is a sequence of operations that may be 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 [6] and was proven independently by us. For completeness, we include the proof in section A.

Lemma 3 *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.*

We remark that the proof of this lemma uses the fact that the definition of strong history independence allows the adversary to choose 'stop' points at the same position. One relaxation of this definition that allows the adversary to choose only distinct 'stop' points is also considered by [6]. This relaxation does not imply canonical layout, but some other relaxed property referred in [6] as *canonical distribution*. Another possible relaxation is to demand that the distributions are only statistically or computationally indistinguishable. The proof does not extend immediately to these cases, but it can be proven that in this case the data structure must be 'almost' canonical. Another possible relaxation is to assume both the above relaxations together. These studies are beyond the scope of this paper and we do not discuss them here.

4.2 Lower bounds on Comparison based data structure implementation

We now proceed to prove 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 [11]. 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 4 below). In particular, this lower bound applies to the heap. We will later prove a more specific lemma (see lemma 8 below) that is valid for the queue, and another specific lemma (lemma 7 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 search tree, a dictionary, 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 4 *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,*

- 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.

Remark 5 *property (ii) implies property (i). We separate them for clarity of the representation.*

Remark 6 *In fact our proof establishes a stronger claim. We prove that for each of the above operations and for any data structure of size n, there exist some parameters for the operation such that the operation requires time complexity $\Omega(n)$.*

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. 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, $\dots d'_{n+1}$. 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: $S = \mathsf{insert}(k_2)$, $\mathsf{insert}(k_3) \dots \mathsf{insert}(k_{n+1})$. 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, $2 \leq i \leq n+1$ 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.

We remark that the data structure may also store the same key in multiple addresses. In this case we consider all the addresses used to store the key k_i when the data structure consists of n keys and the same for a data structure consisting of $n + 1$ keys. Δ then counts how many of these sets of addresses are different. Two sets of addresses used to store a key k_i are the same only if they consist of exactly the same memory addresses. Using this notation it follows that if $\Delta > n/2$ then inserting k_{n+2} will force the data structure making changes in at least $n/2$ of the sets. When $\Delta \leq n/2$ inserting k_1 is again forcing the data structure making at least $n/2$ operations.

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. We can implement a canonical data structure that keeps the keys in two arrays. The $n/2$ smaller keys are sorted bottom up in 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. Since the memory layout consists of only one long array, we may store the first (virtual) array in the odd memory addresses while the second (virtual) array is stored in the even addresses.

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

Lemma 7 *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 $v_1, \ldots, v_n$ but the output is unique: it is the canonical heap with $v_1, \ldots, v_n$. 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. Applying the same rearrangement on two different trees (permutations) results in two different heaps (in which some of the keys are arranged differently in each heap) since the difference in the original trees (permutations) reflects a difference in the result. 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. □

Finally, We show a lower bound on a canonical implementation of the queue data structure. Note that lemma 4 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 8 *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) ... insert-first(k_n) and $S_2 =$ insert-first(k_2), insert-first(k_3) ... 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. □

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 9 *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 4 and 3. □

A special case of the above lemma is the heap.

Corollary 10 *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 9. 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 7 and 3. □

Last, we may also state a lower bound on the queue data structure.

Lemma 11 *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 8 and 3. □

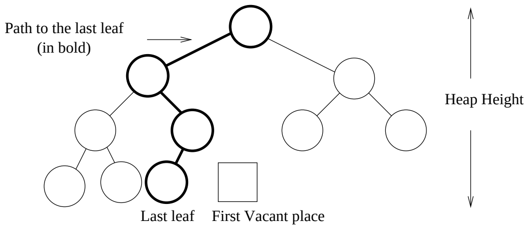

Fig. 1. The height of this heap is 4. The path to the last leaf is drawn in bold. In this example, the path to the first vacant place is the same except for the last edge in the path.

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 well-formed 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 [11]) 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 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 sub-tree) containing the node i and all its descendants. We will use the notation H_L^i for the sub-heap rooted at the left child of the ith node. This is the heap H^{2i}. Respectively, the sub-heap rooted at the right child of the ith node is denoted H_R^i (which is the sub-heap H^{2i+1}).

We now describe the standard implementation of build-heap. This scheme was first suggested by Floyd [8]. The procedure build-heap gets n values in its input and builds a heap for these values with complexity $O(n)$. The procedure build-heap and its main procedure heapify (to be described below) are of major importance to the rest of this paper. They are used extensively in all constructions.

The procedure heapify assumes that the left sub-tree and the right sub-tree of the current node are already arranged as two well-formed heaps. This is clearly true for any leaf (whose children are empty sub-heaps). Now, focusing on a node whose descendants are arranged as two sub-heaps, the procedure heapify makes the node and its two sub-trees a well-formed (one larger) heap. heapify(i, H_L^i, H_R^i) gets as input a node i and its two sub-trees H_L^i and H_R^i that are assumed to be well-formed heaps[3]. The tree H^i is not necessarily a well-formed heap since the value v in the node i may be smaller than the values in i's children, violating the heap property. heapify lets the value v at location i "float down" in the heap making the sub-tree H^i a sub-heap. More specifically, between the two children of i, let i_m be the child with the larger value v_m. If $v \geq v_m$ then we are done. Otherwise, the values of nodes i and i_m are switched, thus, floating down the value v one level. This operation is repeated for the value v until it is placed in a node whose two children contain smaller values. Note that since we switch with the larger child, this child may legitimately become the parent of its sibling. The complexity of running heapify(i, H_L^i, H_R^i) on a node at height h is $O(h)$. In the worst case, the value v floats down all the way to a leaf.

build-heap can now be described recursively as follows: First apply build-heap on each of the sub-trees of the root's children recursively. This results in two well-formed sub-heaps of height at most $h - 1$ and a value at the root that may violate the heap property. Next apply heapify on the root to make the whole tree become a well-formed heap. For the base case note that one node is always a well-formed heap. When applying recursively the procedure build-heap it does not work on a sequential array, except for the top level. That is, when applying build-heap on some sub-tree H^i the actual values of the heap are stored at locations $\{i, 2i, 2i+1, 4i, \ldots\}$. In a full implementation this non-sequential operation should be considered. In order to solve the problem one would probably like to add to the procedure build-heap, and to all other procedures discussed herein, one more parameter, the 'offset' value i. In order

[3] The last two parameters are redundant since they may be obtained from the parent node, yet, it will be useful to have a clear notation of these two in the input.

to simplify our discussion we ignore this extra parameter.

In order to show that build-heap on n values has complexity $O(n)$, we solve a recursive function for a heap of height h (with $h = \lceil \log(n+1) \rceil$). The time complexity of building a heap of height h is the time needed for building two sub-heaps of height $h-1$ and applying heapify on the root.

$$T(h) = O(h) + 2 \cdot T(h-1)$$

Expanding the recursive function we get:

$$T(h) = \sum_{i=1}^{h} \lceil \frac{n}{2^i} \rceil O(i) = O(n \cdot \sum_{i=1}^{h} \frac{i}{2^i}) = O(n).$$

The other three operations on the heap have time complexity $O(h) = O(\log n)$. The standard implementation of extract-max operation is as follows. Extract the maximum value stored at the root of the heap tree. Take the value at the last leaf and put it at the root. Now, the two sub-heaps under the root are well-formed, but the value at the root may violate the max-heap property. Therefore, we apply heapify on the root and let the value 'float' down to its 'right' location. The implementation of increase-key operation is also simple. When we increase the key of some node in the heap, it may violate the max-heap property because it can now be larger than its parent. Therefore, in the standard implementation, we let the value at this node 'float' up by exchanging place with its parent until it reaches its 'correct' place. Using increase-key we can implement the insert operation easily. Just add new leaf at the next vacant place in the heap with value of $-\infty$. Then use increase-key on that leaf with the value to be inserted. For more details and motivation the reader is referred to books on data structures and algorithms (see for example, [4]).

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 section 5 above 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 $\text{build-heap}^{-1}(H)$ 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 $\text{build-heap}(v_{\pi(1)}, \ldots, v_{\pi(n)}) = H$. 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 recursively build-heap on each of the root's children sub-trees. Next it applies heapify on the value at the root to create a well-formed heap. The inverse procedure build-heap^{-1} invokes first a randomized procedure heapify^{-1} on the value at the root of the heap. This creates two well-formed sub-heaps and a (random heap) value at the root, which is not necessarily in its proper position. Next, we apply recursively build-heap^{-1} on each of the sub-heaps. We begin by defining the randomized procedure heapify^{-1}. This procedure is a major player in most of the constructions in this paper. An example of an execution of the procedure heapify^{-1} appears in Figure 2.

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 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 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

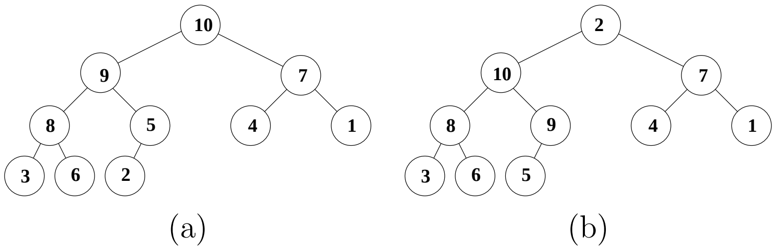

Fig. 2. An example of invoking $\mathsf{heapify}^{-1}(H, 10)$. Node number 10 is the node that contains the value 2. In (b) we can see the output of invoking $\mathsf{heapify}^{-1}$ on the proper heap in (a). The value 2 is put at the root, the path from the root to the father of 2 is shifted down. Note 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)

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 larger value as well.

Claim 12 *Let n be an integer and H be any heap of size n, then for any $1 \leq i \leq n$,*

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

Proof: After running $\mathsf{heapify}^{-1}(H, i)$, the value v from node i is placed in the root. When running the procedure heapify on the resulting tree, the value v floats down. We argue that v floats down exactly along the shifted path replacing each of its values, thus shifting all path values up back to their original location. When v floats down heapify exchange v's place with the child that contains the higher value. Upon starting the descend, v must choose the path first node, since this is the maximum value in the heap (previously shifted down by $\mathsf{heapify}^{-1}(H, i)$ to make room for v) [4]. Next, any node on the shifted path has one path child and one non-path child. The value in it's path child must be larger than the value in the other child. The reason is that before the path shifted down, the path child was a parent of the non-path child (in a well-formed heap). Thus, each node on this path is larger than its sibling and so heapify must choose to replace v with that child down the path towards building back the heap H. Finally, when v reaches its original node i it will stop floating down since the children of node i have not been modified by $\mathsf{heapify}^{-1}$ and they still contain values that are not larger than v, and we are done. □

[4] Here we use the fact that the values in the heap are distinct. If we have two nodes with the same values, then Claim 12 becomes false.

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

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

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

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

Proof Sketch: The claim follows from the fact that for any $1 \leq i \leq n$, $H = \mathsf{heapify}\left(\mathsf{heapify}^{-1}(H, i)\right)$, 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 $(x_1, x_2, \ldots, x_n)$ 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 sub-heaps. 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 $(x_1, x_2, \ldots, x_n)$ is called *proper* if for all i, $1 \leq i \leq n$, it holds that x_i is a node in the heap H^i. The set of proper vectors of size n is thus a cartesian product of sets, one for each node in the heap of size n consisting of all nodes in its sub-heap. We will sometimes let the procedure $\textsf{build-heap}^{-1}(H)$ get its random choices explicitly in the input and use the notation $\textsf{build-heap}^{-1}(H, (x_1, \ldots, x_n))$.

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 14 *Each permutation $\pi \in \Pi(n)$ of values has one and only one heap H and a proper vector $\vec{X}_n = (x_1, x_2, \ldots, x_n)$ such that $(v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)}) =$ $\textsf{build-heap}^{-1}(H, \vec{X}_n)$.*

Proof: We first prove that each permutation $\pi \in \Pi(n)$ has at most one heap H and one proper random vector $\vec{X}_n$ such that $(v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)}) =$ $\textsf{build-heap}^{-1}(H, (x_1, x_2, \ldots, x_n))$. By claim 13 we know that there is only one heap on which $\textsf{build-heap}^{-1}$ may yield the permutation π. This is the heap satisfying $H = \textsf{build-heap}(v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)})$. Therefore we only need to claim that taking any heap H: For each distinct (proper) vector $(x_1, x_2, \ldots, x_n)$ the permutation induced on the values $v_1, \ldots, v_n$ by applying the procedure $\textsf{build-heap}^{-1}(H, (x_1, \ldots, x_n))$ is distinct.

Consider any two distinct proper vectors $(x_1, x_2, \ldots, x_n)$ and $(y_1, y_2, \ldots, y_n)$. Suppose the first different value in these vectors appears in location i. In this case, until $\textsf{build-heap}^{-1}$ is applied on the sub-heap H^i the procedure $\textsf{build-heap}^{-1}(H, (x_1, x_2, \ldots, x_n))$ creates the same tree as $\textsf{build-heap}^{-1}(H, (y_1, y_2, \ldots, y_n))$. But then, node i exchanges values with node $x_i \neq y_i$ and causes a different value to be put in node i. In the rest of the traversal the value in node i is not modified. Thus, the output of $\textsf{build-heap}^{-1}(H, (x_1, x_2, \ldots, x_n))$ is different from the output of $\textsf{build-heap}^{-1}(H, (y_1, y_2, \ldots, y_n))$.

By now we have shown that for any permutation π there is at most one heap H and random vector $(x_1, \ldots, x_n)$ such that $\pi = \textsf{build-heap}^{-1}(H, (x_1, \ldots, x_n))$. We now show that for any permutation π, there exist a heap H and a random (proper) vector $(x_1, \ldots, x_n)$ such that $\pi = \textsf{build-heap}^{-1}(H, (x_1, \ldots, x_n))$.

Denote by *support*(H) the set of all permutations $\pi \in \Pi(n)$ that satisfy:

$$\textsf{build-heap}(v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)}) = H$$

That is *support*(H) contains all the permutation that result in the heap H. Since $\textsf{build-heap}$ is deterministic these sets are a partition of all possible permutation.

We prove that for any permutation $\pi \in \Pi(n)$ in *support*(H) there exists a proper vector $(x_1, \ldots, x_n)$ such that $\textsf{build-heap}^{-1}(H, (x_1, \ldots, x_n))$ yields the order of elements as in π. Since, any permutation is in some set the claim follows.

We will prove this by induction on the height of the heap. If $height(H) = 1$ then there is only one permutation π in *support*$(H)$and the random vector $\{1\}$ yield this permutation.

Consider any heap H of height h and any permutation $\pi \in \Pi(n)$ such that $H =$ build-heap$(v_{\pi(1)}, \ldots, v_{\pi(n)})$. Considering the operation of the procedure build-heap we extract the last operation of heapify on the root and get: $H =$build-heap$(v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)}) =$ heapify$(v_{\pi(1)}, H_L =$build-heap$(v_{\pi(2)}, v_{\pi(4)}, v_{\pi(5)}, \ldots)$, $H_R =$build-heap$(v_{\pi(3)}, v_{\pi(6)}, v_{\pi(7)}, \ldots))$

In the last operation of heapify the element $v_{\pi(1)}$ floats down to the ith position creating the heap H. Now taking $x_1 = i$ will cause $\textsf{build-heap}^{-1}$ in its first step creating exactly H_L, H_R and putting $v_{\pi(1)}$ back at the root. Since H_L and H_R are of height $h - 1$, we can use the induction hypothesis. We get that there exist two series $(x_2, x_4, x_5, \ldots)$ and $(x_3, x_6, x_7, \ldots)$ that yields the order elements as in $\pi_L = \pi(2), \pi(4), \ldots$ and $\pi_R = \pi(3), \pi(6), \ldots$. Merging the series along with x_1 creates the desired proper vector $(x_1, x_2, \ldots, x_n)$. □

Corollary 15 *If H is picked up uniformly among all possible heaps with the same content then $T =$ $\textsf{build-heap}^{-1}$(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 build-heap$(v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)}) = H$, there is a unique random vector $(x_1, \ldots, x_n)$, that creates the permutation. Each random (proper) vector has the same probability. Therefore π is chosen uniformly among all permutation in *support*(H). Since H is picked up uniformly among all heaps the corollary follows. □

Lemma 16 *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 $v_1, v_2, \ldots, v_n$ it holds that:*

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

Where $p(H)$ is 1/ the number of heaps of size n, and it is a function depending only on n (the size of H). Furthermore, $p(H) = N(H)/|\Pi(n)|$ where $N(H)$ is the number of proper vectors of size n, and 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 build-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 14 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 17 *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 $\pi \in_R \Pi(n)$ and invoke* build-heap$(v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)})$.

Proof: As shown in lemma 16, distribution Ω_2 gives all heaps containing the values $v_1, v_2, \ldots, v_n$ the same probability. By definition, this is also the case in the distribution Ω_1. □

6 Building and maintaining History independent Heap

In this section we prove our main theorem.

Theorem 18 *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. The worst case complexity of the operations* insert *and* extract-max *is $O(\log^2 n)$.*

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,

```
procedure build-heap-oblivious($v_1, \ldots, v_n$: Values) : Heap
begin
1.    Choose $\pi \in_R \Pi(n)$ uniformly at random.
2.    $H$ = build-heap($v_{\pi(1)}, \ldots, v_{\pi(n)}$).
3.    Return ($H$)
end
```

Fig. 4. The procedure build-heap-oblivious($v_1, \ldots, v_n$).

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.

This method of obtaining and proving weak history independence is essentially the same as in [9]. In [9] Micciancio obtained weak history independent 2-3 trees by defining a procedure that build such trees from scratch yielding some distribution on the structure of the trees. Then, showing that building such trees by any other sequence of operations yield the same distribution. Notice, however, that in the case of 2-3 trees this was not a uniform distribution on all possible trees as in our case.

6.1 The build-heap operation

We start with the randomized implementation of the operation build-heap-oblivious. We implement it by applying a random permutation on the input values and then invoking the standard build-heap procedure. The pseudo-code appears in figure 4.

Lemma 19 *For any $n \in \mathbb{N}$ and for any n distinct values $(v_1, \ldots, v_n)$, the distribution of heaps output by* build-heap-oblivious$(v_1, \ldots, v_n)$ *is a uniform distribution over all possible heaps containing the values $v_1, \ldots, v_n$.*

Proof: The assertion follows from corollary 17. □

6.2 The increase-key operation

We now provide an implementation of the increase-key operation. This implementation is similar to the standard implementation of increase-key for standard (non-oblivious) heaps. However, we extend this operation by allowing

```
procedure increase-key-oblivious($H$: Heap, $i$: location $value$: the new value)
    : Heap
begin
    if $value < v_i$
        $v_i \leftarrow value$
        $H^i \leftarrow \mathsf{heapify}(i, H^i_L, H^i_R)$
    Otherwise:
        $v_i \leftarrow value$
        while $i \neq 1$ and $v_i > v_{\mathsf{parent}(i)}$
            exchange the values at $i$ and parent($i$).
            $i \leftarrow \mathsf{parent}(i)$
end
```

Fig. 5. The procedure increase-key-oblivious

both increasing and decreasing the key. Such an operation will be useful for us in the implementations of insert and extract-max (see below). In the standard implementation of increase-key the node whose key is to be increased is identified and its value is increased. The update may create a tree that is not a well-formed heap. To make the tree a well-formed heap again, the standard implementation traverses the path from the node toward the root to find the new proper place for the modified value. During this traversal, it repeatedly compares the value of the node to its parent, exchanging them if the child is larger than its parent, when the comparison shows that the node key is smaller than its parent the procedure terminates, and the tree obtained is a well-formed heap.

Our history independent implementation of increase-key is the same as the standard one. We will assert that it is good enough. Implementing the operation in case the key at node i has decreased is done by invoking the heapify procedure on H^i. Note that H^i is an appropriate input for heapify. The root node (node i) may contain any value, but its two sub-trees H^i_L and H^i_R are well-formed heaps. Thus, heapify floats the value down to a proper location modifying H^i into a well-formed heap. The pseudo-code of increase-key-oblivious is provided in figure 5.

We now show that the operation of modifying v_i to v using increase-key-oblivious is a one-to-one transformation from the set of all heaps with values $v_1, v_2, \ldots, v_n$ to the set of all heaps with values $v_1, v_2, \ldots, v_{i-1}$, v, $v_{i+1}, \ldots, v_n$. Furthermore, the inverse operation is exactly applying increase-key-oblivious to modify v back to v_i. The one-to-one property will be used to show that uniformity is maintained by the increase-key-oblivious operation.

Lemma 20 *For any heap H, any vertex i of H, and any (distinct) new value v not contained in H, Let $H' =$ increase-key-oblivious(H, i, v). Let j be the vertex with value v in H' and let v_i be the value of node i in H (the value that* *was modified). Then $H =$ increase-key-oblivious(H', j, v_i).*

Proof: Let us check the case that $v > v_i$. The other case is similar. When running increase-key-oblivious, the new value v floats up until it reaches the root or a parent with a larger value. While going up, all values on the propagation path are shifted one vertex down. We now argue that if we modify the new value v in vertex j back to v_i, and apply increase-key-oblivious on the node j, then v_i floats back exactly along this shifted path returning v_i to vertex i. This is true since now increase-key-oblivious applies heapify on vertex j and heapify keeps switching v_i with its child that contains the higher value. To note that v_i indeed goes down along the propagation path, we note that the path vertex must be the larger child since it was the parent of its sibling before v floated up along that path. Therefore, increase-key-oblivious will always choose to exchange v_i with the previously shifted child returning all vertices in the propagation path back to their previous locations. The value v_i will stop floating exactly in vertex i since the children of vertex i still contain the original values v_{2i} and v_{2i+1}, and since H was well-formed, these values must be smaller than v_i. □

We are now ready to prove that the procedure increase-key-oblivious is history independent. We will show that if the input heap H is distributed uniformly among all heaps with the values $\{v_1, v_2, \ldots, v_n\}$ then the output heap H' is distributed uniformly among all heaps with the values $\{v_1, v_2, \ldots, v_i', \ldots, v_n\}$ where v_i' is the new value that was assigned to vertex i (and perhaps moved by increase-key-oblivious to a different location).

Claim 21 *Let H be a heap of size n uniformly distributed among all heaps with the values $\{v_1, v_2, \ldots, v_n\}$, let i be any number $1 \leq i \leq n$, and let v_i' be a value not contained in H. Then H' =increase-key-oblivious(H, i, v_i') is distributed uniformly among all heaps with the values $\{v_1, v_2, \ldots, v_n\} \setminus \{v_i\} \cup \{v_i'\}$.*

Proof: From lemma 16 we know that for any given n values, the number of heaps of size n with these values depends only on n (and not on the actual values). Now, by lemma 20, we know that increase-key-oblivious gives a one-to-one correspondence between equal sized sets. Thus, the probability that a heap with values $\{v_1, v_2, \ldots, v_n\} \setminus \{v_i\} \cup \{v_i'\}$ appears in the output of increase-key-oblivious equals the probability that its corresponding heap with values $\{v_1, v_2, \ldots, v_n\}$ appears in the input. By the conditions of the lemma, the latter is uniform. □

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

Fig. 6. The procedure extract-max-try-1

6.3 The extract-Max operation

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 simplest implementation, given the tools we developed so far, is to apply the randomized procedure build-heap^{-1} on the heap H (of size $n+1$) to get a uniformly chosen permutation on the values $v_1, v_2, \ldots, v_{n+1}$, then replace the maximum value with the value that turned out last, and re-build the heap from the obtained random permutation on the first n values (excluding the maximum value that is now in location $n+1$).

In order to be able to improve the procedure, we start with a similar, yet somewhat different naive implementation of extract-max denoted extract-max-try-1. We run build-heap^{-1} on the heap H 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. Later, we will show that this process contains many redundant steps and actually running only $O(\log n)$ of the steps in this procedure suffices to receive the same output. The pseudo code of the naive extract-max-try-1 appears in figure 6.

Claim 22 *Let $v_1, \ldots, v_{n+1}$ be $n+1$ (distinct) values and let H be a uniformly* *distributed heap over all heaps with values $v_1, \ldots, v_{n+1}$. Then, invoking procedure* extract-max-try-1 *on H implies the following properties on the heap H' created in step 4.*

(1) The value that is contained in H but not in H' is uniformly distributed over the values $v_1, \ldots, v_{n+1}$.

(2) Given that v_i is contained in H and not in H', then H' is uniformly distributed over all possible heaps with content of $\{v_1, \ldots, v_{n+1}\} \setminus \{v_i\}$.

Proof: By corollary 15 and since the input heap is uniformly distributed, we get that build-heap$^{-1}(H)$ is a uniformly chosen permutation of the values $\{v_1, \ldots, v_{n+1}\}$. Thus, removing the last value in the permutation we get a uniformly chosen removed value, and when conditioning on v_i being removed, we get a uniform permutation over the values $\{v_1, \ldots, v_{n+1}\} \setminus \{v_i\}$. From corollary 17 we know that applying build-heap on this permutation results in uniformly chosen heap among all possible heaps with content $\{v_1, \ldots, v_{n+1}\} \setminus \{v_i\}$. □

Claim 23 *Let $v_1, \ldots, v_{n+1}$ be $n+1$ (distinct) values, let m denote the index of the maximum value (i.e., v_m is the maximum value), and let H be a uniformly distributed heap over all heaps with values $v_1, \ldots, v_{n+1}$. Then, invoking procedure* extract-max-try-1 *on H yields an output heap that is uniformly distributed over all possible heaps with content $\{v_1, \ldots, v_{n+1}\} \setminus \{v_m\}$.*

Proof: By claim 22, for any i, $1 \leq i \leq n+1$, conditioned on v_i being removed, the heap H' created in step 4 is uniformly distributed over all heaps with content $\{v_1, \ldots, v_n, v_{n+1}\} \setminus \{v_i\}$. If $i = m$, i.e., v_i is the maximal value then we are done. Otherwise, v_m must appear in the root of H'. We note that step 6 and 7 implement increase-key-oblivious decreasing the value of the root from v_m to v_i. By claim 21 the resulting heap is uniform if the input heap H' is uniform among all heaps of its content. By claim 22 this is correct for any i, $1 \leq i \leq n+1$. Thus, we get that for any choice of i (and so, also for a random i), the resulting heap is uniformly distributed over all heaps with content $\{v_1, \ldots, v_{n+1}\} \setminus \{v_m\}$ and we are done. □

Note that in the above proof we did not need to use the first part of claim 22. The index i just happens to be uniformly distributed. Also, we might have replaced steps 6 and 7 in procedure extract-max-try-1 with an invocation of increase-key-oblivious. We chose to write steps 6 and 7 explicitly for clarity.

The major reduction of time complexity is presented in our next step in which we construct procedure extract-max-try-2. It performs a small part of procedure extract-max-try-1 achieving the same output. This improvement reduces the complexity of extract-max operation from $O(n)$ to $O(\log^2(n))$. We will then show how to further push the complexity down to $O(\log n)$. The intuition of the saving is as follows. We look at the steps executed by procedure build-heap^{-1} and check which of them are necessary. It turns out that most of them are "cancelled" when build-heap is later invoked. Not executing such steps yields exactly the same output at a lower complexity.

Denote by $\{a_1, a_2, \ldots, a_h\}$ the indices of the nodes that reside on the path from the last leaf (i.e. the last node in the heap tree) to the root. The leaf is denoted a_1 and the root a_h, thus, i is the height of node a_i. Recall that the procedure build-heap^{-1} invokes first heapify^{-1} on the value at the root of the heap. This creates two well-formed sub-heaps and a value at the root, which is not necessarily in its proper position. Next, it applies recursively build-heap^{-1} on each of the sub-heaps. When it applies afterward build-heap it invokes first recursively build-heap on each of the root's children sub-trees. Next it applies heapify on the value at the root to create a well-formed heap. The major reduction is applying build-heap^{-1} and build-heap recursively only on the direction to the last leaf and do nothing on the other direction. The result is that extract-max-try-2 applies heapify^{-1} only on the nodes $(a_h, a_{h-1}, \ldots, a_1)$ (from top to bottom) and then heapify on a similar subset in a reverse order. We will prove that eventually this outputs the same heap as extract-max-try-1.

The procedure build-heap^{-1} uses randomness for choosing a descendant x_i for each visited vertex i. We denote the randomness by a vector $(x_1, \ldots, x_{n+1})$. But since we will only be interested in the vertices $(a_1, \ldots, a_h)$, we will use the notation $(x_{a_1}, x_{a_2}, \ldots, x_{a_h})$ to denote the sequence of random choices made for the vertices $(a_1, \ldots, a_h)$ that interest us. Thus, x_{a_j} is the random choice for vertex a_j.

Now, let us show how to reduce most of the steps in procedure extract-max-try-1. The heart of the matter is a procedure extract-recursive-try-2 that substitutes steps 2,3, and 4 in extract-max-try-1. Note that these steps require $O(n)$ time steps since build-heap^{-1} and build-heap are run. The idea is that instead of performing build-heap^{-1} on the heap, removing the last leaf and performing build-heap again (as in extract-max-try-1), it is enough to perform only parts of these two procedures: the parts that are relevant for the vertices $a_1, \ldots, a_h$.

Recall our notational convention from section 5.1. The heapify procedure gets 3 parameters (two sub-heaps and an index): heapify(i, H_L^i, H_R^i) and outputs a well-formed heap H^i, whereas the inverse function heapify^{-1} gets a well formed heap, and a choice x_j and it returns a tree containing two well formed sub-heaps H_L and H_R and the value v_{x_j} (of the input heap) at the root. The procedure extract-recursive-try-2 runs heapify^{-1} only on the vertices $a_h, \ldots, a_1$ (from root to leaf) instead of running it on all vertices. It then removes the value at the last leaf and reconstructs the heap by running heapify on the vertices $a_2, \ldots, a_h$ in a reverse order: from leaf to root (there is no need to run heapify on a_1 since the value at this node is extracted from the heap). To simplify the analysis later, we present procedure extract-recursive-try-2 in

```
procedure extract-recursive-try-2($H$: Heap, $(x_{a_1}, x_{a_2}, \ldots, x_{a_h})$: Random choices)
: Heap, value
begin
    if $H = v_i$ (i.e. $H$ is one node) return (empty heap, $v_i$). Otherwise:
        $(H_L, H_R, v_{x_{a_h}})$ = heapify$^{-1}(H, x_{a_h})$.
        If the path to the last leaf in $H$ is going to the left:
            ($H'_L$, last) $\leftarrow$extract-recursive-try-2$(H_L, \{x_{a_1}, x_{a_2}, \ldots, x_{a_h - 1}\})$
            $H'_R \leftarrow H_R$.
        Otherwise:
            ($H'_R$, last) $\leftarrow$ extract-recursive-try-2$(H_R, \{x_{a_1}, x_{a_2}, \ldots, x_{a_h - 1}\})$
            $H'_L \leftarrow H_L$.
    Return((heapify($H'_R, H'_L, v_{x_{a_h}}$), last)).
end
```

Fig. 7. The procedure extract-recursive-try-2

a recursive manner. First, heapify^{-1} is run on the root. In the bottom of the recursion, we have one vertex in the tree. In this case, this vertex is the last leaf, and it is removed. Otherwise, extract-recursive-try-2 is run recursively on the subtree that contains the path $(a_{h-1}, \ldots, a_1)$. Procedure extract-recursive-try-2 is assumed to return a well-formed heap from which the value v_i (that resides in the last leaf) has been removed. Finally, heapify is applied on the root (containing the value $v_{x_{a_j}}$ of the input heap at recursion level j), and the two sub-heaps: the one returned by the recursion and the one that was not modified (since it was not on the $(a_1, \ldots, a_h)$ path). Thus, extract-recursive-try-2 returns a well-formed heap. The procedure also returns the value of the last leaf (that was removed from the heap). The pseudo code appears in figure 7.

Procedure extract-max-try-2 is the procedure in which we switch steps 2,3, and 4 in extract-max-try-1 with the sub-procedure extract-recursive-try-2. The pseudo-code of this procedure is given in figure 9. We now claim that extract-max-try-1 and extract-max-try-2 have the same output distribution. The essence of the proof will be to show that lines 2,3,4 above (that will be denoted extract-recursive-try-1) output the same heap as extract-recursive-try-2 when they get the same input. We define extract-recursive-try-1 to be the sub-procedure that gets H and a proper random vector $(x_1, \ldots, x_{n+1})$ in the input. It performs lines 2,3, and 4 of extract-max-try-1 and returns the H' defined in line 4, and v_i, the value of the (removed) last leaf returned in step 3. For clarity we explicitly provide this procedure in figure 8.

To make the syntax equal, we let extract-recursive-try-2 take a full proper random vector $(x_1, \ldots, x_{n+1})$ although it uses only the values $(x_{a_1}, x_{a_2}, \ldots, x_{a_h})$ out of this vector. We now state and prove the claim.

Claim 24 *For any heap H of size $n+1$ and proper random vector $(x_1, \ldots, x_{n+1})$:*

```
procedure extract-recursive-try-1($H$: Heap, $(x_1, x_2, \ldots, x_{n+1})$: Random choices)
: Heap, value
begin
    $T$ = build-heap$^{-1}(H, (x_1, x_2, \ldots, x_{n+1}))$
    Let $T'$ be the tree obtained by removing the last node with value $v_i$ from $T$.
    $H$' = build-heap$(T')$
    Return $(H', v_i)$
end
```

Fig. 8. The procedure extract-recursive-try-1

```
procedure extract-max-try-2($H$: Heap) : Heap
begin
    Choose uniformly at random a proper randomization vector $(x_{a_1}, \ldots, x_{a_h})$
    for the procedure extract-recursive-try-2.
    $(H', v_i)$ = extract-recursive-try-2$(H, (x_{a_1}, x_{a_2}, \ldots, x_{a_h}))$
    if $v_i$ is the maximum return $(H')$. Otherwise:
        Modify the value at the root to $v_i$.
        $H''$ = heapify(H, 1) (i.e. apply heapify on the root)
        Return $(H'')$
end
```

Fig. 9. The procedure extract-max-try-2

extract-recursive-try-1$(H, (x_1, \ldots, x_{n+1}))$ = extract-recursive-try-2$(H, (x_1, \ldots, x_{n+1}))$

Proof: The proof is by induction on the height of the heap H.

Induction Base: When the heap is of height 1, there is no difference between the operation of extract-recursive-try-1 and extract-recursive-try-2. Therefore the claim holds.

Induction Step: Consider the first operation of heapify^{-1}, applied in the same manner in both extract-recursive-try-1 and extract-recursive-try-2. Let $(H'_L, H'_R, v_{x_{a_h}})$= heapify$^{-1}(H, x_{a_h})$. Assume without loss of generality that the first step from the root on the path to the last leaf goes left. Let $\vec{X}_{n+1}$ be a proper vector of size $n+1$, $(x_1, x_2, \ldots, x_{n+1})$. Now by reordering the operation of extract-recursive-try-1 we get that:

$$\begin{aligned}
&\textsf{extract-recursive-try-1}(H, \vec{X}_{n+1}) \\
&= \textsf{build-heap}(\text{remove last node}(\textsf{build-heap}^{-1}(H, \vec{X}_{n+1}))) \\
&= \textsf{heapify}(\textsf{extract-recursive-try-1}(H'_L, \vec{X}_{n+1}), \\
&\qquad\qquad \textsf{build-heap}(\textsf{build-heap}^{-1}(H'_R, \vec{X}_{n+1})), v_{x_{a_h}}) && (1) \\
&= \textsf{heapify}(\textsf{extract-recursive-try-1}(H'_L, \vec{X}_{n+1}), H'_R, v_{x_{a_h}}) && (2) \\
&= \textsf{heapify}(\textsf{extract-recursive-try-2}(H'_L, \vec{X}_{n+1}), H'_R, v_{x_{a_h}}) && (3) \\
&= \textsf{extract-recursive-try-2}(H, \vec{X}_{n+1}) && (4)
\end{aligned}$$

We remark that the output of extract-recursive-try-1 and extract-recursive-try-2 are two values: A new heap and a value extracted from the previous heap. The equation is only between the first parameter, the resulting heap. Since the resulting heaps are the same, the extracted value must also be the same. Equality 1 follows from reordering the operations of extract-recursive-try-1. To see that equality 1 holds, note that build-heap^{-1} applies heapify^{-1} on the root and then continues recursively to the sub-heaps of the root's children. The operation of build-heap^{-1} on each of the sub-heaps can be done independently of the other sub-heap. This is true since the operation of heapify^{-1} affects only the sub-tree it operates on. The same holds for the operation of build-heap, that can be done independently on both sub-heaps. Therefore, we can separate the operations done on the right child from the operations done on the left child. Equality 2 follows from removing the cancelling operations build-heap and build-heap^{-1}. Equality 3 follows from the induction hypothesis. The last equality is exactly the definition of extract-recursive-try-2. □

Corollary 25 *For any heap H* extract-max-try-1(H) *and* extract-max-try-2(H) *produce the same output distribution.*

Proof: The only difference between extract-max-try-1 and extract-max-try-2 is the use of extract-recursive-try-2 instead of extract-recursive-try-1. Thus, the corollary follows directly from claim 24 □

Though it is not needed for our final result, it is interesting to note that the worst time complexity of the procedure extract-recursive-try-2 and therefore the complexity of extract-max-try-2 is $O((\log n)^2)$. Each iteration of extract-recursive-try-2 has worst time complexity $O(h)$ and there are h such invocations, where $h = O(\log n)$. This bound also holds for the next (final) implementation of the extract-max operation, because the final implementation always executes less operations than the above implementation. However, for this final implementation, we will show that the expected time complexity is only $O(\log n)$.

We are now ready to provide the last improvement over the extract-max operation, which reduces the complexity of extract-max to $O(\log(n))$. We start with some intuition. Recall that the idea behind the first procedure extract-recursive-try-1 is to use build-heap^{-1} to get one of the possible permutations that could create the input heap H. This is done only in order to remove the value in the last leaf of the generated tree and build back the heap. When we build back the heap the value at the last leaf is removed and therefore it is possible that some of the operations that previously involved this value will change. In our improvement we try to determine which of the operations really involved the value at the last leaf. We then run the reversing and building only with these operations. We will show that in most cases there aren't many operations that involve the value of the last leaf. For instance, if the value is very small it probably stays at the last leaf and won't affect most of the operation in build-heap.

Practically, we will not change the procedure extract-recursive-try-2, but we will manipulate its input vector of random choices. Notice that when a node chooses to stay in its place and not replace another node during heapify^{-1} (i.e. when $x_{a_i} = a_i$) then the complexity of the heapify^{-1} is $O(1)$. We will manipulate the random choices so that most of the operations will become as efficient as that and we will show that the output remains the same.

Next we provide the sub-routine that manipulates a series of random choices $(x_{a_1}, x_{a_2}, \ldots, x_{a_h})$ and return instead a series $(y_{a_1}, y_{a_2}, \ldots, y_{a_h})$ that is cheaper to run. This is done in complexity $O(\log(n))$. We then prove that running extract-recursive-try-2 with the new series of random choices does not change the procedure's output.

The sub-routine produce-y that execute this manipulation appears in figure 10. Notice that it gets as input only the size of the heap tree and the random choices, and does not depend on the actual values in the heap. The procedure returns the new series $(y_{a_1}, y_{a_2}, \ldots, y_{a_h})$ plus an internal variable *leaf-h*. This value is not used by the calling routine but it will help us proving some properties about the functionality of the series. Informally, this variable contains the height of the value in the heap H that has been removed from the heap. The procedure produce-y works in a bottom-up manner. When it gets a vector $(x_{a_1}, x_{a_2}, \ldots, x_{a_h})$, it first manipulate the sub-series of its sub-heap and last manipulates the last value y_{a_h}. When the procedure manipulates the last value it can also increase the value of *leaf-h* by 1. This happens only when x_{a_h} is a location inside the sub-heap of node $a_{\text{leaf-h}}$, where *leaf-h* is the value been calculated for the sub-heap.

We now prove a few claims that shed light on the *leaf-h* index. The first claim asserts that when we apply extract-recursive-try-2 on a heap H of size $n + 1$ to get new heap H' of size n then the value that is removed from H is the value at node $a_{\text{leaf-h}}$ and the heaps H and H' are equal except for changes in

```
procedure produce-y($n$: **Heap size**, $(x_{a_1}, x_{a_2}, \ldots, x_{a_h})$: **Random choices**)
: $(y_{a_1}, y_{a_2}, \ldots, y_{a_h})$, *leaf-h*
begin
    if $n = 1$ (i.e. the heap is of size 1, and the vector is of size 1)
        Return($(x_{a_1}), 1$) (i.e. in this case $x_{a_1} = a_1$ always)
    $((y_{a_1}, y_{a_2}, \ldots, y_{a_{h-1}}), \text{leaf-h}) = \text{produce-y}(n', (x_{a_1}, x_{a_2}, \ldots, x_{a_{h-1}}))$
    where $n'$ is the size of the sub-heap to the direction of the last leaf.
    if $x_{a_h}$ is a location in the sub-heap of $a_{\text{leaf-h}}$,$H^{a_{\text{leaf-h}}}$ then:
        Return($(y_{a_1}, y_{a_2}, \ldots, y_{a_{h-1}}, x_{a_h}), \text{leaf-h} + 1$)
    Otherwise:
        Return($(y_{a_1}, y_{a_2}, \ldots, y_{a_{h-1}}, a_h), \text{leaf-h}$)
end
```

Fig. 10. The procedure produce-y

the sub-heap $H^{a_{\text{leaf-h}}}$.

Claim 26 *Let H be a heap with $n+1$ values. Let $a_1, a_2, \ldots, a_h$ be the path from the root to the last leaf. Let $(x_{a_1}, x_{a_2}, \ldots, x_{a_h})$ be a proper random choice for the nodes $(a_1, a_2, \ldots, a_h)$. Let $H' =$ extract-recursive-try-2$(H, (x_{a_1}, x_{a_2}, \ldots, x_{a_h}))$, and let* leaf-h *be the one returned by* produce-y(size$(H), x_{a_1}, x_{a_2}, \ldots, x_{a_h}$)). *Then:*

(1) The heaps H and H' are identical except for the sub-tree $H^{a_{\text{leaf-h}}}$ of the node $a_{\text{leaf-h}}$.

(2) The value of the node $a_{\text{leaf-h}}$ in H is the one that has been removed from H by extract-recursive-try-2.

Proof: The proof is by induction on the height of the heap.

Induction Base: If H is of size 1 then H contains one node and H' is empty. In this case *leaf-h* is always 1 and the claim holds trivially.

Induction Step: We consider a heap of height h and assume the claim holds for all heaps of height less than h. Assume without loss of generality that the first step on the path from the root to the last leaf goes left. Let $(H_L, H_R, v_{x_{a_h}})$ = heapify$^{-1}(H, x_{a_h})$, and let *leaf-h*' = produce-y(size(H_L), $(x_{a_1}, x_{a_2}, \ldots, x_{a_{h-1}})$). Consider the operation of extract-recursive-try-2 we may write:
extract-recursive-try-2$(H, (x_{a_1}, x_{a_2}, \ldots, x_{a_h})) =$
heapify(extract-recursive-try-2$(H_L, (x_{a_1}, x_{a_2}, \ldots, x_{a_{h-1}})), H_R, v_{x_{a_h}}$)

We partition the analysis into two cases according to whether *leaf-h* = *leaf-h*' or *leaf-h* = *leaf-h*' +1. Recall that the value of *leaf-h* on a series $(x_{a_1}, x_{a_2}, \ldots, x_{a_h})$ is at least the value of *leaf-h* on the sub-series $(x_{a_1}, x_{a_2}, \ldots, x_{a_{h-1}})$, and may be increased by at most by one in any recursive level of the procedure produce-y.

Case 1: *leaf-h* = *leaf-h*': By the operation of produce-y this means that x_{a_h} is a location not in the sub-heap of node $a_{\text{leaf-h}'}$. By the induction hypothesis H'_L=

extract-recursive-try-2$(H_L, (x_{a_1}, x_{a_2}, \ldots, x_{a_{h-1}}))$ is different from H_L only in the sub-heap of $a_{\text{leaf-h}'} = a_{\text{leaf-h}}$. The rest of H_L remains unchanged. The values in the sub-heap under $a_{\text{leaf-h}'}$ are all the values that were there in the heap H_L except for the value that was removed by extract-recursive-try-2. The removed value was at node $a_{\text{leaf-h}'}$ in H_L and thus was the maximum value among its sub-heap. Therefore, the value at location $a_{\text{leaf-h}'}$ in the modified heap H_L' is smaller than the value at the same location is H_L.

When we applied heapify^{-1} in step 2 of the procedure extract-recursive-try-2 the value at node x_{a_h} got to the root and shifted all the values on the path from the root to node x_{a_h} one step down. We claim that when we apply heapify in step 6 of extract-recursive-try-2 on the root location of the modified heap letting the value at the root 'float' down, the value 'floats' exactly along this previously shifted path returning all the values on this path to their original positions. This is true for two reasons: First, we know that this path does not intersect the changed sub-heap $H^{a_{\text{leaf-h}'}}$. Second, the value at location $a_{\text{leaf-h}'}$ is now smaller from the value that had been there before the operation of extract-recursive-try-2. By the definition of heapify a value that floats down exchange places with the maximal child so it will not change the floating route at the parent of node $a_{\text{leaf-h}}$ (that got smaller). For this reason the heaps H and H' remain different only in the sub-heap of $a_{\text{leaf-h}'} = a_{\text{leaf-h}}$. This proves the first part of the claim.

Moving to the second part we note that by the induction hypothesis, the value that is removed by extract-recursive-try-2 from H_L is the value at location $a_{\text{leaf-h}'} = a_{\text{leaf-h}}$ in H_L. Since x_{a_h} is a location not in the sub-heap of node $a_{\text{leaf-h}'}$ then the first operation of heapify^{-1} in step 2 does not change the value at that location. Thus, the removed value is at location $a_{\text{leaf-h}}$ also in H and we are done with the second part of the claim.

Case 2: *leaf-h* = *leaf-h*' +1. From the operation of produce-y this means that x_{a_h} is a location in the sub-heap of node $a_{\text{leaf-h}'}$.

By the induction hypothesis we know that the differences between H_L and H_L' are only in the sub-tree of node $a_{\text{leaf-h}'}$. When applying the first heapify^{-1} in step 2 of the procedure extract-recursive-try-2 the value at node x_{a_h} got to the root and shifted all the values on the path from the root to node x_{a_h} one step down. We claim that when we apply heapify in step 6 of the procedure extract-recursive-try-2 on the root location of the modified heap letting the value at the root 'float' down, the value 'floats' exactly along this previously shifted path at least until it reaches the location of the parent of node $a_{\text{leaf-h}'}$. This is true since H_L' is different from H_L only in the sub-heap of node $a_{\text{leaf-h}'}$. This claim implies that all the values on this sub-path return to their original positions. Thus, the heaps H and H' can now be different only in the sub-heap of the parent of location $a_{\text{leaf-h}'}$. Since *leaf-h* = *leaf-h'* +1, it is exactly node

$a_{\text{leaf-h}}$, and we are done with the first part of the claim.

By the induction hypothesis, the value that was removed from H is the value that was in node $a_{\text{leaf-h}'}$ in H_L. Remember that the operation $\mathsf{heapify}^{-1}$ moves the value at node x_{a_h} to the root and shift all the values on the path to location x_{a_h} one step down. Since x_{a_h} is a location in the sub-heap of node $a_{\text{leaf-h}'}$ this value is the value that was in location $a_{\text{leaf-h}}$ (the parent node of $a_{\text{leaf-h}'}$) in H and was shifted down one step by the first operation of $\mathsf{heapify}^{-1}$ in step 2. This proves the second part of the claim. □

We have shown that H' =extract-recursive-try-2$(H, (x_{a_1}, x_{a_2}, \dots, x_{a_h}))$ is different from H only inside the sub-heap of node $a_{\text{leaf-h}}$. We now prove that the changes in that sub-heap do not depend on the values in the rest of the heap. That is, if two heaps H_1, H_2 satisfy $H_1^{a_{\text{leaf-h}}} = H_2^{a_{\text{leaf-h}}}$ (but the rest of their values may be different) then the sub-heaps $H_1^{a_{\text{leaf-h}}}$ and $H_2^{a_{\text{leaf-h}}}$ remain equal also after applying extract-recursive-try-2 on both heaps.

Claim 27 *Let H_1, H_2 be two heaps of size n. Let $a_h, a_{h-1}, \dots, a_1$ be the nodes on the path from the root to the last leaf in both heaps. Let $(x_{a_1}, x_{a_2}, \dots, x_{a_h})$ be a proper random choices for the nodes $(a_1, a_2, \dots, a_h)$ respectively. Let H_1' =* extract-recursive-try-2$(H_1, (x_{a_1}, \dots, x_{a_h}))$ *and H_2' =* extract-recursive-try-2$(H_2, (x_{a_1}, \dots, x_{a_h}))$*, and let* leaf-h *be the one returned by* produce-y(size(H_1) = size(H_2), $(x_{a_1}, x_{a_2}, \dots, x_{a_h})$)*. Then:*
if $H_1^{a_{\textit{leaf-h}}} = H_2^{a_{\textit{leaf-h}}}$ then $H_1^{'a_{\textit{leaf-h}}} = H_2^{'a_{\textit{leaf-h}}}$.

Proof: The proof is by induction on the height of the heaps.

Induction Base: If The height of the heaps is 1 then *leaf-h* is always 1, and the claim holds trivially.

Induction Step: We consider two heaps of height h and assume the claim holds for every two heaps of height less than h. Assume without loss of generality that the first step on the path from the root to the last leaf goes left. This direction is the same in both heaps since they are of the same size. Let $(H_{1L}, H_{1R}, v_{x_{a_h}}) = \mathsf{heapify}^{-1}(H_1, x_{a_h})$ and $(H_{2L}, H_{2R}, v'_{x_{a_h}}) = \mathsf{heapify}^{-1}(H_2, x_{a_h})$. Let *leaf-h'* = produce-y(*size*(H_{1L})= *size*(H_{2L}) , $(x_{a_1}, x_{a_2}, \dots, x_{a_{h-1}})$). Looking at the operation of extract-recursive-try-2 we note that for both heaps:

$$\mathsf{extract\text{-}recursive\text{-}try\text{-}2}(H, (x_{a_1}, x_{a_2}, \dots, x_{a_h})) =$$

$$\mathsf{heapify}(\mathsf{extract\text{-}recursive\text{-}try\text{-}2}(H_L, (x_{a_1}, x_{a_2}, \dots, x_{a_{h-1}})), H_R, v_{x_{a_h}})$$

We partition the analysis into two cases according to whether *leaf-h* = *leaf-h'* or *leaf-h* = *leaf-h'* +1. Recall that the value of *leaf-h* on a series $(x_{a_1}, x_{a_2}, \dots, x_{a_h})$ is at least the value of *leaf-h* on the sub-series $(x_{a_1}, x_{a_2}, \dots, x_{a_{h-1}})$, and may increase by at most one in the top recursion level of the procedure produce-y.

Case 1: *leaf-h* = *leaf-h'*: By the operation of produce-y this means that x_{a_h} is a location not in the sub-heap of node $a_{\text{leaf-h}'}$ in both heaps. If before the operation of heapify^{-1} at step 2 of extract-recursive-try-2 $H_1^{a_{\text{leaf-h}}}$ equals $H_2^{a_{\text{leaf-h}}}$ then $H_{1L}^{a_{\text{leaf-h}}}$ must equal also $H_{2L}^{a_{\text{leaf-h}}}$, because the operation heapify^{-1} does not affect this sub-heap.

By claim 26, after applying recursively extract-recursive-try-2 on the sub-heaps H_{1L}, H_{2L}, the only change in the sub-heaps is in the sub-heaps of node $a_{\text{leaf-h}'}$. The new value at node $a_{\text{leaf-h}'}$ after applying extract-recursive-try-2 is the maximal value among the values in this sub-heap and it is smaller than the value that was located in $a_{\text{leaf-h}'}$ in H_L, because the value that was removed by extract-recursive-try-2 is the value at location $a_{\text{leaf-h}'}$ that contained the maximal value in this sub-heap.

We now return to the operation heapify^{-1} in step 2 in the procedure extract-recursive-try-2. On both heaps the value at node x_{a_h} got to the root and shifted all the values on the path from the root to node x_{a_h} one step down. We claim now that when we apply the last heapify (i.e in step 6 in extract-recursive-try-2) on the root location on the modified heaps letting the value at the root 'float' down, the value 'floats' exactly along this previously shifted path returning all the values on this path to their original positions. This is true for two reasons: First, we know that this path does not intersect the modified sub-heap $H^{a_{\text{leaf-h}'}}$. Second, the value at location $a_{\text{leaf-h}'}$ is now smaller then the value that had been there before the operation of extract-recursive-try-2. Therefore, the value of the root that 'floats' down exchanging places with its maximal child will not change its floating route at the parent of node $a_{\text{leaf-h}}$. For this reason the this 'float' does not change the sub-heaps of node $a_{\text{leaf-h}'}$ on both sub-heaps. We know from the induction hypothesis that the sub-heaps $H_{1L}^{a_{\text{leaf-h}'}}$ and $H_{2L}^{a_{\text{leaf-h}'}}$ were equal before the last operation of heapify in step 6, and that *leaf-h* = *leaf-h'*. Thus, in the end the sub-heap $H_1^{a_{\text{leaf-h}}}$ equals the sub-heap $H_2^{a_{\text{leaf-h}}}$ and we are done with case 1.

Case 2: *leaf-h* = *leaf-h'* +1. By the operation of produce-y this means x_{a_h} is a location in the sub-heap under $a_{\text{leaf-h}'}$. In this case, it is possible that H_{1L} is different from H_{2L} also inside the sub-heaps of node $a_{\text{leaf-h}}$, but we will show that eventually the sub-heaps of node $a_{\text{leaf-h}'}$ remain equal. This is true since the first operation of heapify^{-1} on the root in step 2 of procedure extract-recursive-try-2 applied both on H_1 and H_2 shifts the path to node x_{a_h} one step down, therefore can cause only the location $a_{\text{leaf-h}}$ to become different in H_{1L} and H_{2L}. Other then that both sub-heaps of node $a_{\text{leaf-h}}$ are equal in H_{1L} and H_{2L}.

By the induction hypothesis after applying recursively extract-recursive-try-2 on H_{1L} and H_{2L}, they remain the same in the sub-heap of node $a_{\text{leaf-h}'}$. Also by claim 26 the rest of the heap is not modified and thus, the other sub-heap of the child of node $a_{\text{leaf-h}}$ not on the path to the last leaf remains unchanged in H_{1L}' and H_{2L}' and therefore remains identical on both H_{1L}' and H_{2L}'.

Notice first that the value at the root of H_1 is the same as the value at the root of H_2. This is true since it is the value that got there via the operation of $\mathsf{heapify}^{-1}$ in step 2 that took the value at node x_{a_h} to the root. The location x_{a_h} is inside the identical sub-heap of H_1 and H_2 and therefore it is the same in both heaps.

When applying $\mathsf{heapify}^{-1}$ in step 2 on both heaps H_1 and H_2, the value at node x_{a_h} got to the root and shifted all the values on the path from the root to node x_{a_h} one step down. We claim that when we apply heapify in step 6 on the root location letting the value at the root 'float' down, the value 'floats' exactly along this previously shifted path at least until it reaches the location of the parent of node $a_{\text{leaf-h}'}$ (i.e. to node $a_{\text{leaf-h}}$). This returns all the values on this sub-path to their original positions. This is true since H'_{1L} is different from H_{1L} only in the sub-heap of node $a_{\text{leaf-h}'}$ and the same holds for H_{2L}.

From this point both sub-heaps of the children of node $a_{\text{leaf-h}}$ are the same in both heaps, therefore from this location the value continues to 'float' down in the same route in H_1' and H_2' resulting in equivalent sub-heaps under location $a_{\text{leaf-h}}$. Thus, in the end the sub-heap $H_1^{a_{\text{leaf-h}}}$ equals the sub-heap $H_2^{a_{\text{leaf-h}}}$ and we are done with claim 27 □

We are now ready to prove our main lemma regarding extract-max.

Lemma 28 *Let H be a heap with n values, let $a_h, a_{h-1}, \ldots, a_1$ be the nodes on the path from the root to the last leaf, let $(x_{a_1}, x_{a_2}, \ldots, x_{a_h})$ be a proper random choice for the nodes $(a_1, a_2, \ldots, a_h)$, let $((y_{a_1}, y_{a_2}, \ldots, y_{a_h}), \text{leaf-h}) =$ produce-y$(\text{size}(H), (x_{a_1}, x_{a_2}, \ldots, x_{a_h}))$. Then:*
extract-recursive-try-2$(H, (x_{a_1}, \ldots x_{a_h})) =$ extract-recursive-try-2$(H, (y_{a_1}, \ldots y_{a_h}))$

Proof: The proof is by induction on the height of the heap.

Induction Base: If the height of H is 1, then H contains only one node. In this case $y_{a_1} = x_{a_1} = a_1$ and the lemma holds.

Induction Step: We consider a heap of height h and assume the claim holds for all heaps of height less than h. Assume without loss of generality that the first step on the path from the root to the last leaf goes left. Let $(H_{Lx}, H_{Rx}, v_{x_{a_h}}) = \mathsf{heapify}^{-1}(H, x_{a_h})$ and $(H_{Ly}, H_{Ry}, v_{y_{a_h}}) = \mathsf{heapify}^{-1}(H, y_{a_h})$. These are the sub-heaps created after applying the first $\mathsf{heapify}^{-1}$ at step 2 of extract-recursive-try-2 with the random choice x_{a_h} and with y_{a_h}. Let *leaf-h'* = produce-y(size(H_{Lx}), $(x_{a_1}, x_{a_2}, \ldots, x_{a_{h-1}})$) Looking at the operation of extract-recursive-try-2 we may write:
extract-recursive-try-2$(H, (x_{a_1}, x_{a_2}, \ldots x_{a_h})) =$

$$\text{heapify}(\text{extract-recursive-try-2}(H_{Lx}, (x_{a_1}, x_{a_2}, \dots x_{a_{h-1}})), H_{Rx}, v_{x_{a_h}}).$$

We partition the analysis into two cases according to whether $y_{a_h} = x_{a_h}$ or $y_{a_h} \neq x_{a_h}$.

Case 1: $y_{a_h} = x_{a_h}$. If this is the case then $H_{Lx} = H_{Ly}$ and $H_{Rx} = H_{Ry}$. By the induction hypothesis we get that:
extract-recursive-try-2$(H_{Lx}, (x_{a_1}, x_{a_2}, \dots x_{a_{h-1}})) =$
extract-recursive-try-2$(H_{Ly} = H_{Lx}, (y_{a_1}, y_{a_2}, \dots y_{a_{h-1}}))$. Thus, just before applying heapify on the root at step 6. The value at the root, and both its sub-heaps are equal, this means that the heaps are also equal after executing heapify and we are done.

Case 2: $y_{a_h} \neq x_{a_h}$. From the operation of produce-y this means that at step 7 in produce-y y_{a_h} got the value a_h. This means that x_{a_h} is a location not in the sub-tree of node $a_{\text{leaf-h}'}$. In this case the following equalities hold:

$$\begin{aligned}
&\text{extract-recursive-try-2}(H, (x_{a_1}, x_{a_2}, \dots x_{a_h}))\\
&= \text{heapify}(\text{extract-recursive-try-2}(H_{Lx}, (x_{a_1}, x_{a_2}, \dots x_{a_{h-1}})), H_{Rx}, v_{x_{a_h}})\\
&= \text{heapify}(\text{extract-recursive-try-2}(H_{Ly}, (x_{a_1}, x_{a_2}, \dots x_{a_{h-1}})), H_{Ry}, v_{y_{a_h}}) \quad (5)\\
&= \text{heapify}(\text{extract-recursive-try-2}(H_{Ly}, (y_{a_1}, y_{a_2}, \dots y_{a_{h-1}})), H_{Ry}, v_{y_{a_h}}) \quad (6)\\
&= \text{extract-recursive-try-2}(H, (y_{a_1}, y_{a_2}, \dots y_{a_h}))
\end{aligned}$$

Equality 6 follows by the induction hypothesis. The main point here is equality 5. We first consider the changes happen in the two sub-heaps H_{Lx} and H_{Ly} when we apply extract-recursive-try-2 on them. Then we consider the changes after applying heapify on the root. Finally, we claim the result heap is the same.

H_{Lx} and H_{Ly} are not equal, but they are equal in the sub-heap of node $a_{\text{leaf-h}'}$. This is true since x_{a_h} is not a location in the sub-heap of $a_{\text{leaf-h}'}$. Thus, by claim 27 after applying extract-recursive-try-2 recursively on H_{Lx} and H_{Ly} the two sub-heaps remain equal in the sub-heap of node $a_{\text{leaf-h}'}$. Note also that by claim 26 the other parts node in the sub-heap of node $a_{\text{leaf-h}'}$ of both H_{Lx} and H_{Ly} remain unchanged. Consider now the operation of heapify at step 6 in extract-recursive-try-2 on H_{Lx} and H_{Ly}. In H_{Ly} the value at the root is the maximal value and therefore nothing happens. When we applied the first heapify^{-1} in step 2 in the procedure extract-recursive-try-2 with the value x_{a_h}, the value at node x_{a_h} got to the root and shifted all the values on the path from the root to node x_{a_h} one step down. We claim now that when we apply heapify in step 6 of extract-recursive-try-2 on the root location of the modified heap letting the value at the root 'float' down, the value 'floats' exactly along this previously shifted path returning all the values on this path to their original positions. This is true for two reasons: First, we know that this path does not

```
procedure extract-max-oblivious($H$: Heap) : Heap
begin
1.  Choose uniformly at random a proper randomization vector $(x_{a_1}, \ldots, x_{a_h})$
    for the procedure extract-recursive-try-2.
2.  $(y_{a_1}, y_{a_2}, \ldots, y_{a_h})$ = produce-y($size(H)$,$(x_{a_1}, x_{a_2}, \ldots, x_{a_h})$)
3.  $(H', v_i)$ = extract-recursive-try-2($H, (y_{a_1}, y_{a_2}, \ldots, y_{a_h})$)
4.  if $v_i$ is the maximum return $(H')$. Otherwise:
5.      Modify the value at the root to $v_i$.
6.      $H''$ = heapify($H, 1$) (i.e. apply heapify on the root)
7.      Return $(H'')$
end
```

Fig. 11. The procedure extract-max-oblivious

intersect the modified sub-heap $H^{a_{\text{leaf-h}'}}$. Second, the value at location $a_{\text{leaf-h}'}$ is now smaller then the value been there before the operation of extract-recursive-try-2, the value 'floats' down exchanging places with its maximal child so it will not change the floating route at the parent of node $a_{\text{leaf-h}}$. This shifts back all the values shifted by the operation heapify^{-1} applied at step 2.Thus, both heaps become equal both in the sub-heap of node $a_{\text{leaf-h}'}$ and the other parts of the heap after applying heapify at step 6 and we are done. □

We are now ready to provide the pseudo-code of extract-max-oblivious operation appears in figure 11. The only change in the algorithm from extract-max-try-2 is the use of the modified random vector produced by produce-y. We can now state the following corollary asserts that the procedure is history independent.

Corollary 29 *Let $v_1, \ldots, v_{n+1}$ be $n+1$ (distinct) values, let m denote the index of the maximum value (i.e., v_m is the maximum value), and let H be a uniformly distributed heap over all heaps with values $v_1, \ldots, v_{n+1}$. Then, invoking procedure* extract-max-oblivious *on H yields an output heap that is uniformly distributed over all possible heaps with content $\{v_1, \ldots, v_{n+1}\} \setminus \{v_m\}$.*

Proof: We only need to prove that for any heap H extract-max-oblivious and extract-max-try-2 produce the same output distribution. The only difference between extract-max-oblivious and extract-max-try-2 is the use of the new random series produced by the procedure produce-y. Thus, the corollary follows directly from lemma 28. □

It remains to analyze the complexity of extract-max-oblivious. We start with a useful claim. Informally, we claim that if we take a uniformly chosen permutation and build a heap from it then the last value of the permutation will not ascend too much. That is, the expected height of the value appears last in the permutation is $O(1)$. We will later relate this height to the complexity of extract-max-oblivious.

Claim 30 *Let $v_1, v_2, \ldots, v_n$ be n distinct values. Let $\pi \in_R \Pi(n)$ be a random permutation on these values specifying their order in an almost full tree, and let $h(H, v_{\pi(n)})$ be the height of $v_{\pi(n)}$ in the heap H. Then,*

$$E\left[\pi \in_R \Pi(n); H = \textsf{build-heap}(v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)}) : h(H, v_{\pi(n)})\right] \leq 4$$

Proof: By corollary 15 we can rephrase the above expected value as:

$$E\left[H \in_R \mathbb{H} \ : \ \pi = \textsf{build-heap}^{-1}(H) \ : \text{the height of } v_{\pi(n)} \text{ in the heap H}\right]$$

Where $\mathbb{H}$ is the set of all heaps of size n.

When applying build-heap^{-1} one of the values on the path from the last leaf to the root gets to be the last leaf (this is the opposite of build-heap in which the value at the last leaf can only ascend during the operation of build-heap). In extract-max-try-1, the value that got to the last leaf is removed. Thus, what we are looking for is the height of the value that was removed by extract-max-try-1, this is the value that got to the last leaf. Using claims 24 and 28 we know that extract-recursive-try-2 results in the same heap and removes the same value even when we apply the procedure with the values $(y_{a_1}, y_{a_2}, \ldots, y_{a_h})$ produced by produce-y instead of the original random series $(x_{a_1}, x_{a_2}, \ldots, x_{a_h})$. Therefore the value that is removed from the heap has the same distribution as in extract-max-try-1. Last, from the second part of claim 26 the value that got to the last leaf and was removed is the value in location $a_{\text{leaf-h}}$ in H with height *leaf-h*, where *leaf-h* is the value returned by produce-y($size(H)$,$(x_{a_1}, x_{a_2}, \ldots, x_{a_h})$). Notice that the location of the value that is removed does not depend on the actual values of the heap, but only on the vector $(x_{a_1}, x_{a_2}, \ldots, x_{a_h})$. Therefore, we can further rephrase the above expectation into:

$$E\left[\vec{X}_h \in_R \mathbb{X}; \textit{leaf-h} = \textsf{produce-y}(size(H), \vec{X}_h) : \textit{leaf-h}\right]$$

Where $\mathbb{X}$ is the set of all proper vectors $(x_{a_1}, x_{a_2}, \ldots, x_{a_h})$.

We analyze the expectation of *leaf-h* by looking at the inside operation of the procedure produce-y. The procedure has $h-1$ recursive calls. On the way back from each recursive call the procedure considers the random choice of the current sub-heap root. Let i be the current index of random choice (i.e the index that we consider after the recursive call with $i-1$ indices). We define $h-1$ random variables X_1, X_2, X_{h-1} where X_Δ is defined to be the number of returns from recursive calls of the procedure produce-y for which the difference between the index that is considered by the procedure and the value of *leaf-h* is Δ (i.e $\Delta = i-$ *leaf-h*). At the first return from a recursion call this difference is 1 (the index that is considered is 2 and the value *leaf-h* returned from the recursion base is 1). This difference can only grow throughout the procedure and get up to $h-1$ depend upon the exact values of the series

$(x_{a_1}, x_{a_2}, \ldots, x_{a_h})$. In each return from recursive call *leaf-h* may grow by one in step 5 or remain the same in step 7.

By the operation of produce-y if x_{a_i} is a location inside $H^{a_{\text{leaf-h}}}$ then *leaf-h* increases by one and the difference between the index that is considered by the procedure and the value of *leaf-h* remains the same (they both grow by 1). Otherwise, *leaf-h* does not increase and thus the difference increases by one. This means that if the difference between the iteration number and *leaf-h* remains steady for l iteration then *leaf-h* increases by $l-1$. Let the difference at the end of the procedure be k.

Using this notation, we can write the value *leaf-h* as the sum of increments of it during the procedure:

$$\textit{leaf-h} = \sum_{\Delta=1}^{k} (X_\Delta - 1) = \sum_{\Delta=1}^{k} X_\Delta - k \tag{7}$$

We now analyze the expected value of X_Δ and bound it from above. The analysis for X_1 is a little different from the others. We start by analyzing the expected value of X_1. The difference between the random index and *leaf-h* is 1. In order to increase *leaf-h* the random choice must be inside the sub-heap of the child in the direction to the last leaf. The 'worst' case is when this sub-heap is of maximal size and the other child is small. The heap is an almost full binary tree, therefore if the 'small' sub-heap of a child is of size a the other child's sub-heap can be at most of size $2a+1$. The probability of choosing the 'large' child is therefore always at most $\frac{2a+1}{3a+2} < \frac{2}{3}$ (i.e. choosing one of the $2a+1$ locations inside the 'large' sub-heap of size $2a+1$ and not the other smaller sub-heap of size a or the location of the root itself). This means that the probability that a random choice is a location not in the sub-heap to the last leaf and therefore increase the difference is at least $\frac{1}{3}$ in each iteration. Thus, $E(X_1) \leq 3$.

Extending this idea to $i > 1$ we may claim that if the difference is $\Delta > 1$ then the probability of choosing a location inside the 'bad' heap (that keeps the difference unchanged) is at most $\frac{2a+1}{(2^\Delta+1)a+2^\Delta} < \frac{2(a+1)}{2^\Delta(a+1)} = \frac{1}{2^{\Delta-1}}$. Therefore, the probability that the difference increases in the each time for which the difference is Δ is at least $1 - \frac{1}{2^{\Delta-1}} = \frac{2^{\Delta-1}-1}{2^{\Delta-1}}$. Thus $E(X_\Delta) \leq \frac{2^{\Delta-1}}{2^{\Delta-1}-1} \leq 1 + \frac{1}{2^{\Delta-2}}$

Plugging this result in 7 we get that:

$$E[\textit{leaf-h}] = E[\sum_{\Delta=1}^{k} X_\Delta - k] \leq 3 + \sum_{\Delta=2}^{k} (1 + \frac{1}{2^{\Delta-2}}) - k = 2 + \sum_{\Delta=2}^{k} \frac{1}{2^{\Delta-2}} \leq 4 = O(1)$$

□

We now ready to analyze the complexity time of extract-max-oblivious and prove the following claim:

Claim 31 *The expected time complexity of* extract-max-oblivious *operation is* $O(\log(n))$*. Where the expectation is over all random choices of the operation.*

Proof: Step 1 in the extract-max-oblivious chooses randomly a vector of size h Therefore works in worst time complexity of $O(h) = O(\log(n)$. In step 2 we operate produce-y on the random choices. This procedure manipulates the vector by one recursive call on each member in the vector. Each recursive call is done in $O(1)$. Thus, the time complexity of the procedure is $O(h) = O(\log(n))$. The complexity of steps 4-7 is just the complexity of one operation of heapify which is $O(h)$. The only problematic part is therefore the expected complexity of extract-recursive-try-2. This complexity depends on the random vector $(y_{a_1}, y_{a_2}, \dots y_{a_h})$ which depend on the previous random choice of $(x_{a_1}, x_{a_2}, \dots x_{a_h})$. Looking at the operation of extract-recursive-try-2, we see that in each recursive call if $x_{a_h} = a_h$ (i.e. the node 'decides' to stay in its place) then applying heapify^{-1} at step 2 is redundant. In that case the operation of heapify when we return from the recursive calls in step 6 is also redundant, since we only removed a value from the heap, and therefore the value at the root is still the maximum and the max-heap property is preserved. In real implementation we probably want to skip these two steps if this is the case. Therefore the complexity of extract-recursive-try-2 is only the complexity of the non trivial operations (i.e. where $y_{a_h} \neq a_h$) of heapify and heapify^{-1} inside it.

Since the complexity of both heapify and heapify^{-1} is $O(h)$, we only need to prove that the expected number of recursive calls for which $x_{a_h} \neq a_h$ is $O(1)$. Looking back at the procedure produce-y this number is exactly the value of *leaf-h* returned by produce-y. Therefore the claim follows directly from claim 30, and we are done. □

6.4 The insert operation

We start with a naive implementation of insert which we call insert-try-1. This implementation has complexity $O(n)$. This is of-course unacceptable for the insert operation but it allows a construction of a simple and useful implementation that will be improved later. The general goal is to get an input heap that is uniformly distributed and output a heap that is also uniformly distributed. The basic idea behind this implementation is as follows. Since we may assume we have a uniformly chosen heap, we can sample in the inverse of build-heap and get a uniformly chosen permutation (in $\Pi(n)$) of the heap values. Now, to insert the new value a and get a random heap on $n+1$ values,

```
procedure insert-try-1($H$: Heap, $a$: Value) : Heap
begin
1.  Think of the input value $a$ as being located in an additional node
    numbered $n+1$. (This is the first vacant place as in figure 5.)
    Choose uniformly at random a number $1 \leq i \leq n+1$, let the value
    of node $i$ by $v_i$.
2.  If $(i = n+1)$ then H'=H and skip to step 4. Otherwise:
3.      $H' \leftarrow$ increase-key-oblivious$(H, i, a)$
4.  Choose uniformly at random a proper randomization vector $(x_1, \ldots, x_n)$
    for the procedure build-heap$^{-1}$.
5.  Invoke $T =$ build-heap$^{-1}(H', (x_1, \ldots, x_n))$
6.  Let $T'$ be the tree obtained by adding to $T$ the next vacant node
    with value $v_i$.
7.  H = build-heap$(T')$
8.  Return $(H)$
end
```

Fig. 12. The procedure insert-try-1

we first choose a random location i, $1 \leq i \leq n+1$. If $i \leq n$ then we put a at location i and move the previous value of i to the end (which is now location $n+1$). If $i = n+1$ we just put the value a at the end. This yields a uniform permutation on the $n+1$ values. Now, invoking build-heap on these values, we get a uniform heap with the $n+1$ values.

The above procedure can be easily shown to yield a uniform heap but is so naively designed that it is difficult to improve it. We start with a little twist of this procedure, changing the order of operations and fixing the heap in between. The twisted procedure will allow improving its complexity as required. More specifically, 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 at 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, what we really do is applying 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.

The pseudo-code of the naive insert-try-1 appears in figure 12. Next we prove that this naive implementation is history independent.

Claim 32 *Let H be a heap of size n uniformly distributed among all heaps with the values $\{v_1, v_2, \ldots, v_n\}$, and let a be a new distinct value. Then:*

(1) The heap H' returned by insert-try-1 *in step 3 is distributed uniformly among all heaps with the values $\{v_1, v_2, \ldots, v_n\} \cup \{a\} \setminus \{v_i\}$.*

(2) T returned by insert-try-1 *in step 5 is distributed uniformly among all permutations with the values $\{v_1, \ldots, v_n\} \cup \{a\} \setminus \{v_i\}$.*

Proof: The first part of the claim follows directly from claim 21. The second part follows from corollary 15. □

Next we prove the history independence of insert-try-1.

Claim 33 *Let H be a heap of size n uniformly distributed among all heaps with the values $\{v_1, v_2, \ldots, v_n\}$, and let a be new distinct value. Then H' =* insert-try-1(H, a) *is distributed uniformly among all heaps with the values $\{v_1, \ldots, v_n\} \cup \{a\}$.*

Proof: Consider the value that is chosen in the first step of insert-try-1. We first claim that v_i is chosen uniformly among the values $\{v_1, \ldots, v_n\} \cup \{a\}$. This is true since insert-try-1 chooses random location i, $1 < i < n + 1$. Each random location i implies a unique value v_i. Thus, each value is selected with equal probability.

By the second part of claim 32 the tree T returned by insert-try-1 in step 5 is distributed uniformly among all permutations with the values $\{v_1, \ldots, v_n\} \cup \{a\} \setminus \{v_i\}$.Thus, we get that T' obtained in step 6 is distributed uniformly among all permutations with the values $\{v_1, \ldots, v_n\} \cup \{a\}$. Hence, by corollary 17 the heap returned by insert-try-1 is uniformly distributed among all heaps with the values $\{v_1, \ldots, v_n\} \cup \{a\}$ and we are done. □

Next we present insert-try-2. This is an essential step in the improvement of the insert operation. We will show that we can execute a small part of the operations of insert-try-1 and still get the same result. This improvement reduces the complexity of the insert operation from $O(n)$ to $O(\log^2(n))$, and will be the basis of the final version of insert-oblivious.

Denote by $\{a_1, a_2, \ldots, a_h\}$ the indices of the nodes that reside on the path from the first vacant place (i.e. the next free leaf in the heap tree) to the root (see figure 1). The first vacant place is denoted a_1 and the root is a_h, thus, i is the height of node a_i (in the heap of size $n + 1$). Recall that the procedure build-heap^{-1} invokes first heapify^{-1} on the value at the root of the heap. This creates two well-formed sub-heaps and a value at the root, which is not necessarily in its proper position. Next, we apply recursively build-heap^{-1} on each of the sub-heaps. When we apply afterward build-heap it invokes first recursively build-heap on each of the root's children sub-trees. Next it applies heapify on the value at the root to create a well-formed heap. The major reduction is applying build-heap^{-1} and build-heap recursively only on the direction to the first vacant place and do nothing on the other direction. The result is that

```
procedure recursive-insert-try-1($H$: Heap, $v$ : value, $(x_1, \ldots, x_n)$): Random choices) : Heap
begin
1.    $T = \text{build-heap}^{-1}(H, (x_1, \ldots, x_n))$
2.    Let $T'$ be the tree obtained by adding to $T$ the next vacant node
      with value $v$.
3.    Return ($H = \text{build-heap}(T')$)
end
```

Fig. 13. The procedure recursive-insert-try-1

insert-try-2 applies heapify^{-1} only on the nodes $(a_h, a_{h-1}, \ldots, a_1)$ (from top to bottom) and then heapify on a similar subset in a reverse order. We will prove that eventually its output is the same heap as insert-try-1. Notice that the path contains a node that is not in the heap, and therefore we really apply the operations of heapify^{-1} only until a_2. Though, on the way back, when applying heapify, we already insert the new value at the vacant node and it becomes part of the new heap tree.

The procedure build-heap^{-1} uses randomness for choosing a descendant x_i for each visited vertex i. We denote these random choices by a vector $(x_1, \ldots, x_n)$. But since we will only be interested in the vertices $(a_1, \ldots, a_h)$, we will use the notation $(x_{a_1}, x_{a_2}, \ldots, x_{a_h})$ to denote the sequence of random choices made for the vertices $(a_1, \ldots, a_h)$ that interest us. Thus, x_{a_j} is the random choice for vertex a_j.

The heart of our improvement is a new sub-procedure recursive-insert-try-2. This procedure substitutes steps 5,6,7 in insert-try-1. Note that these steps require $O(n)$ time since build-heap^{-1} and build-heap are run. Steps 5,6,7 in insert-try-1 take as input a heap H a value v_i and random vector of size n. Step 7 returns a new heap of size $n + 1$. We define recursive-insert-try-1 to be the sub-procedure that gets H and a value v_i and executes exactly these steps. For clarity we explicitly provide this procedure appears in figure 13. The new procedure, recursive-insert-try-2, is provided in figure 14. In this procedure we only invoke heapify and heapify^{-1} on the nodes $a_h, a_{h-1}, \ldots, a_1$.

To make syntax equal, we let recursive-insert-try-2 take full proper random vector $(x_1, x_2, \ldots x_n)$ although it uses only the values $(x_{a_1}, x_{a_2}, \ldots, x_{a_h})$ out of this vector. Notice that since a_1 is outside the heap there exist no x_{a_1}. Since this sub-heap is of size one, we may treat the value x_{a_1} as a_1, i.e. it stays in place. We now show that these two procedures produce the same output.

Claim 34 *For any heap H, new distinct value a, and a proper random vector $(x_1, x_2, \ldots x_n)$.*
recursive-insert-try-1$(H, a, (x_1, \ldots x_n))$ = recursive-insert-try-2$(H, a, (x_1, \ldots x_n))$

```
procedure recursive-insert-try-2($H$: Heap, $v$ : value, $(x_{a_1}, \dots, x_{a_h})$: Random choices) : Heap
begin
    if $height(H) = 0$ return $H = v$ (i.e put value $v$ in a new node). Otherwise:
        $(H_L, H_R, v_{x_{a_h}}) = \mathsf{heapify}^{-1}(H, x_{a_h})$.
        If the path to the first vacant place in $H$ is going to the left:
            $H'_L \leftarrow$recursive-insert-try-2$(H_L, v, (x_{a_1}, x_{a_2}, \dots, x_{a_{h-1}}))$
            $H'_R \leftarrow H_R$.
        Otherwise:
            $H'_R \leftarrow$ recursive-insert-try-2$(H_R, v, (x_{a_1}, x_{a_2}, \dots, x_{a_{h-1}}))$
            $H'_L \leftarrow H_L$.
        Return(heapify$(H_R, H_L, v_{x_{a_h}})$).
end
```

Fig. 14. The procedure recursive-insert-try-2

Proof: The proof is by induction on the height of the heap H.

Induction Base: When the height of the heap is 0, there is no difference between the operations of recursive-insert-try-1 and recursive-insert-try-2.

Induction Step: Assume without loss of generality that the first step in the path to the first vacant place goes left. Consider the first operation of heapify^{-1}, applied in the same manner in both recursive-insert-try-1 and recursive-insert-try-2. Let $(H'_L, H'_R, v_{x_{a_h}}) = \mathsf{heapify}^{-1}(H, x_{a_h})$. Let $\vec{X}_n = (x_1, x_2, \dots, x_n)$ be a proper vector of size n. Now by the operation of recursive-insert-try-1 we get that:

$$\begin{aligned}
&\mathsf{recursive\text{-}insert\text{-}try\text{-}1}(H, v, \vec{X}_n) \\
&= \mathsf{heapify}(\mathsf{recursive\text{-}insert\text{-}try\text{-}1}(H'_L, v, \vec{X}_n), \\
&\qquad\qquad \mathsf{build\text{-}heap}(\mathsf{build\text{-}heap}^{-1}(H'_R, \vec{X}_n)), v_{x_{a_h}}) && (8) \\
&= \mathsf{heapify}(\mathsf{recursive\text{-}insert\text{-}try\text{-}1}(H'_L, v, \vec{X}_n), H'_R, v_{x_{a_h}}) && (9) \\
&= \mathsf{heapify}(\mathsf{recursive\text{-}insert\text{-}try\text{-}2}(H'_L, v, \vec{X}_n), H'_R, v_{x_{a_h}}) && (10) \\
&= \mathsf{recursive\text{-}insert\text{-}try\text{-}2}(H, v, \vec{X}_n) && (11)
\end{aligned}$$

Where equality 8 follows from reordering the operations of recursive-insert-try-1. To see that equality 8 holds, note that build-heap^{-1} applies heapify^{-1} on the root and then continues recursively to the sub-heaps of the root's children. The operation of build-heap^{-1} on each of the sub-heaps can be done independently of the other sub-heap. This is true since the operation of heapify^{-1} affects only the sub-tree it operates on. The same holds for the operation of build-heap, that can be done independently on both sub-heaps. Therefore, we can separate the operations done on the right child from the operations done on the left

```
procedure insert-try-2($H$: Heap, $a$: Value) : Heap
begin
1.  Think of the input value $a$ as being located in an additional node
    numbered $n+1$. (This is the first vacant place as in figure 5.)
    Choose uniformly at random a number $1 \leq i \leq n+1$, let the value
    of node $i$ be $v_i$.
2.  If $(i = n+1)$ then H'=H and skip to step 5. Otherwise:
3.      $H' \leftarrow$ increase-key-oblivious$(H, i, a)$
4.  Choose uniformly at random a proper randomization vector $(x_{a_1}, \ldots, x_{a_h})$
    for the procedure recursive-insert-try-2.
5.  Return (recursive-insert-try-2$(H', v_i, (x_{a_1}, x_{a_2}, \ldots, x_{a_h}))$)
end
```

Fig. 15. The procedure insert-try-2

child.

Equality 9 follows from removing the cancelling operations build-heap and build-heap^{-1}. Equality 10 follows from the induction hypothesis. The last equality follows from the definition of recursive-insert-try-2. □

The code of insert-try-2 appears in figure 15. Let us prove our main claim about the operation of insert-try-2 .

Claim 35 *For any heap H and value a. The procedures* insert-try-1 *and* insert-try-2 *produce the same output distribution.*

Proof: Steps 1 to 3 in both procedures are the same. Thus, the claim follows from claim 34 □

It now follows that the worst time complexity of the procedure insert-try-2 is $O((\log n)^2)$. The complexity of insert-try-2 is dominated by the procedure recursive-insert2, which operates at the most $O(\log n)$ times the heapify and the heapify^{-1} operations, each costing $O(\log n)$ operations. This bound also holds for the next (final) implementation of the insert operation, because the final implementation always executes less operations than the above implementation. However, for this final implementation, we will show that the expected time complexity is only $O(\log n)$.

We now proceed to the last improvement that further reduces the complexity of insert to the desired $O(\log(n))$. We focus on improving over the procedure recursive-insert-try-2. The improved procedure, recursive-insert, gets one more parameter in its input. The input of recursive-insert-try-2 was H, v, and $(x_{a_1}, x_{a_2}, \ldots, x_{a_h})$. We add an input parameter j specifying the maximal i for which $v_{a_i} < v$. This value is a number between 1 and h, and is always larger or equal to 1 since the index a_1 is outside the heap, and therefore is considered

```
procedure recursive-insert($H$: Heap, $v$ : value, $\{x_{a_1}, x_{a_2}, \ldots, x_{a_h}\}$: Random choices, $j$ : index) : Heap
begin
1.  if height($H$) $= 0$ return $H = v$ (i.e put value $v$ in a new node). Otherwise:
2.      If $x_{a_h}$ is a location not in $H^{a_j}$ then $x_{a_h} \leftarrow a_h$, Otherwise: $j \leftarrow j - 1$
3.      $(H_L, H_R, v_{x_{a_h}}) =$ heapify$^{-1}(H, x_{a_h})$.
4.      If the path to the first vacant place in $H$ is going to the left:
5.          $H'_L \leftarrow$ recursive-insert$(H_L, v, (x_{a_1}, x_{a_2}, \ldots, x_{a_{h-1}}), j)$
            $H'_R \leftarrow H_R$.
        Otherwise:
6.          $H'_R \leftarrow$ recursive-insert$(H_R, v, (x_{a_1}, x_{a_2}, \ldots, x_{a_{h-1}}), j)$
            $H'_L \leftarrow H_L$.
7.      Return(heapify$(H'_R, H'_L, v_{x_{a_h}})$).
end
```

Fig. 16. The procedure recursive-insert

smaller than v.

Also since H is a well-formed heap then $v_{a_1} < v_{a_2} < \ldots, v_{a_h}$, and therefore j is the maximal index (i.e. the 'highest' node on the path) for which the value at node a_i in the heap is still smaller than the value v at the input of recursive-insert.

The intuition of this new input is that if the value v is small (This happens most of the time since v is chosen uniformly by insert-try-2) then it does not affect most of the calls of build-heap. The procedure tries to reverse only the operations for which the value of v may influence the building of the heap. We prove that at the end of the procedure the value v gets exactly to the height of j. The code of recursive-insertis provided in figure 16.

Notice that the only difference between recursive-insert and recursive-insert-try-2 is adding step 2. If $x_{a_h} = a_h$ it means that both heapify^{-1} in step 3 and heapify in step 7 are redundant. In a real implementation we would skip them both. Our main claim asserts that this modification has no affect on the output of the procedure. We prove now some useful claims that allow proving this main claim. Notice that some of the claims relate to the properties of recursive-insert-try-2 and not to recursive-insert.

Claim 36 *Let H be a heap with n values. Let $a_h, a_{h-1}, \ldots, a_1$ be the path from the root to the first vacant place. Let $(x_{a_1}, x_{a_2}, \ldots, x_{a_h})$ be a proper random choice for the nodes $(a_1, a_2, \ldots, a_h)$. Let v be new distinct value and let j be the maximal index for which $v_{a_j} < v$ in the heap H.*
Let $H' =$ recursive-insert-try-2$(H, v, (x_{a_1}, x_{a_2}, \ldots, x_{a_h}))$. Then,

(1) The heaps H and H' are identical except for the sub-heap of node a_j.

(2) The value v is at node a_j in the heap H'.

Proof: The proof is by induction on the height of the heap.

Induction Base: For heap of height 0 it is easy to verify the claim.

Induction Step: We consider a heap of height h and assume the claim holds for all heaps of height less than h. Assume without loss of generality that the first step on the path from the root to the first vacant place goes left.

Let $(H_L, H_R, v_{a_h}) = \mathsf{heapify}^{-1}(H, x_{a_h})$ be the first operation in step 2 of the procedure recursive-insert-try-2. If $j = h$ then the first part of the lemma trivially holds. It also means that the value v is larger than all the values in the heap H and therefore located at the root (location a_h) in H'. Otherwise: Let $H'_L = \mathsf{recursive\text{-}insert\text{-}try\text{-}2}(H_L, v, (x_{a_1}, x_{a_2}, \ldots, x_{a_h-1}))$. We partition the analysis into two cases according to whether x_{a_h} is a location in the sub-heap of node a_j or not.

Case 1: If x_{a_h} is a location in the sub-heap of node a_j then all the values on the path from the root to a_j are shifted one step down by the operation of heapify^{-1} in step 2 of procedure recursive-insert-try-2. If this is the case then when we apply recursive-insert-try-2 on H_L the maximal index j' for which $v_{a_j} < v$ in H_L equals now $j-1$. This is true because the parent of a_j that is larger than v moved one step down to the direction of the first vacant place. By the induction hypothesis we get that H'_L is different from H_L only in the sub-heap of node a_{j-1}.

When applying heapify^{-1} in step 2 of recursive-insert-try-2 the value at node x_{a_h} got to the root and shifted all the values on the path from the root to node x_{a_h} one step down. We claim now that when we apply heapify in step 6 on the root location in the modified heap letting the value at the root 'float' down, the value 'floats' exactly along this previously shifted path at least until it reaches the location of the parent of node $a_{j'}$ (that is node a_j). This returns all the values on this sub-path to their original positions. This is true since H'_L is different from H_L only in the sub-heap of node $a_{j'}$. Thus, the heaps H and H' can now be different only in the sub-heap of the parent of location $a_{j'}$. Since $j = j' + 1$, it is exactly node a_j, and we are done with the first part of the claim.

In addition, from the second part of the induction hypothesis, the value that is at node $a_{j'}$ in H'_L is the new value v. The value v is larger than the value at location a_j in H and hence larger than all the values in its sub-heap. This means that the value at the root, that was taken from this sub-heap, is strictly smaller than v. From the induction hypothesis the other sub-heap under location a_j (not to the direction of the next vacant place and v) is the same in H_L and H'_L. This means that the value at the top of this sub-heap in

H'_L is smaller than v, since it is can be at most the value located previously at node a_j in H. This means that in the operation of heapify at step 6 when the value at the root floats down and reaches node a_j it must choose to switch with v and not his sibling. Thus, moving v to location a_j. This proves the second part of the claim.

Case 2: If x_{a_h} is not a location in the sub-heap of a_j, then the maximal index in H_L for which $v_i < v$ is still j. Applying the induction hypothesis on the recursive call of recursive-insert-try-2 on H_L, we get that H'_L and H_L are only different in the sub-heap of node a_j. By the second part of the induction hypothesis the value at node a_j in H'_L is v which is strictly smaller than all the values of the ancestors of node a_j in H.

When we applied heapify^{-1} in step 2 in the procedure recursive-insert-try-2 the value at node x_{a_h} got to the root and shifted all the values on the path from the root to node x_{a_h} one step down. We claim now that when we apply heapify in step 6 of recursive-insert-try-2 on the root location of the modified heap letting the value at the root 'float' down, the value 'floats' exactly along this previously shifted path returning all the values on this path to their original positions. This path does not intersect the modified sub-heap of node a_j, thus if the path does not pass the parent of node a_j there is no problem. If the path passes through the parent of node a_j then we claim that the value never switches with the value v at node a_j (i.e does not change its route). This is true because the value that was previously at location of the parent of a_j in H is strictly larger than v. This value is either the value that floats down from the root (if x_{a_h} is the location of the parent of a_j), or it was shifted one step down to the direction of x_{a_h} and is now the sibling of node a_j and larger than v. Therefore, the value at node a_j remain the value v, and we are done with both parts of the claim. $\square$

We have shown that the heap H' = recursive-insert-try-2$(H, v, (x_{a_1}, \ldots, x_{a_h}))$ is different from H only inside the sub-heap of node a_j where j is the maximal index for which the value at node a_j in H is still smaller then the new value v. We now show that the modifications in that sub-heap do not depend on the rest of the heap. That is, if two sub-heaps H_1 and H_2 are equal in the sub-heap of node a_j, but the rest of their values may be different, then the sub-heaps $H_1^{a_j}$ and $H_2^{a_j}$ remain equal after applying recursive-insert-try-2 on both heaps.

Claim 37 *Let H_1 and H_2 be two heaps of size n. Let $a_h, a_{h-1}, \ldots, a_1$ be the path from the root to the first vacant place. Let $(x_{a_1}, x_{a_2}, \ldots, x_{a_h})$ be a proper random choice for the nodes $(a_1, a_2, \ldots, a_h)$. Let v be new distinct value and let j_1, j_2 be the maximal indices for which in H_1 and H_2 respectively $v_{a_{j_i}} < v$. Let H'_1 =* recursive-insert-try-2$(H_1, v, (x_{a_1}, x_{a_2}, \ldots, x_{a_h}))$ *H'_2 =* recursive-insert-try-2$(H_2, v, (x_{a_1}, \ldots, x_{a_h}))$ *Then, if $j_1 = j_2$ (and denote $j \triangleq j_1 = j_2$) and*

$H_1^{a_j} = H_2^{a_j}$ (i.e. the sub-heaps of node a_j are equal) then H_1' and H_2' are equal in the sub heap of node a_j.

Proof: The proof is by induction on the height of the heaps.

Induction Base: When H_1 and H_2 are of size 0, it is easy to verify that the claim holds.

Induction Step: We consider two heaps of height h and assume the claim holds for every two heaps of height less than h.

Assume without loss of generality that the first step on the path from the root to the first vacant place goes left. This direction is the same in both heaps since they are of the same size. Let $(H_{1L}, H_{1R}, v_{a_h}) = \mathsf{heapify}^{-1}(H_1, x_{a_h})$, and $(H_{2L}, H_{2R}, v_{a_h}) = \mathsf{heapify}^{-1}(H_2, x_{a_h})$.

We partition the analysis into two cases according to whether the location x_{a_h} is in the sub-heap of node a_j or not.

Case 1: If the location x_{a_h} is not in the sub-heap of node a_j (both in H_1 and H_2) then $H_{1L}^{a_j}$ is equal $H_{2L}^{a_j}$. By claim 36 we know that after applying recursive-insert-try-2 recursively on H_{1L} and on H_{2L} the heaps only change is in the sub-heaps of node a_j, and that the value at location a_j is the value v. Since $H_{1L}^{a_j}$ is equal $H_{2L}^{a_j}$ and the value j was not changed in both sub-heaps, we can apply the induction hypothesis on the recursive operation of recursive-insert-try-2. By the induction hypothesis $H_{1L}^{a_j}$ is equal $H_{2L}^{a_j}$ before applying heapify in step 6.

When we applied heapify^{-1} in step 2 in the procedure recursive-insert-try-2 the value at node x_{a_h} got to the root and shifted all the values on the path from the root to node x_{a_h} one step down. We claim now that when we apply heapify in step 6 of recursive-insert-try-2 on the root location of the modified heaps letting the value at the root 'float' down, the value 'floats' exactly along this previously shifted path returning all the values on this path to their original positions. This path does not intersect the modified sub-heap of node a_j, thus if the path do not pass the parent of node a_j there is no problem. If the path passes through the parent of node a_j then we claim that the value never switches with the value v at node a_j (i.e does not change its route). This is true because the value that was previously at location of the parent of a_j in H is strictly larger than v. This value is either the value that float down from the root (if x_{a_h} is the location of the parent of a_j), or it was shifted one step down to the direction of x_{a_h} and is now the sibling of node a_j and larger than v. From this reason the last float does not change the sub-heaps of node a_j on both heaps and thus these sub-heaps remain identical.

Case 2: The location x_{a_h} is inside the sub-heap of node a_j (in both H_1 and

H_2). This means that in the operation of heapify^{-1} the value at the node of the parent of node a_j is shifted down to the location of a_j. Thus, it is possible that H_{1L} and H_{2L} are not equal in the sub-heap of node a_j. Still, it is true that after this operation the two sub-heaps of the children nodes of node a_j are the same in H_{1L} and H_{2L}. The value that is shifted down to the node a_j is larger than v, therefore the maximal value that is still smaller than v in H_{1L} and H_{2L} is now at node a_{j-1}.

Consider now the two heaps H_{1L} and H_{2L} after inserting the value v (by recursive call of recursive-insert-try-2). By the induction hypothesis the two sub-heaps of node a_{j-1} are identical. By claim 36 the other parts of H_{1L} and H_{2L} not in the sub-heap of node a_{j-1} have not changed. In particular, notice that the sub-heap of the sibling of node a_{j-1} is the same in both modified heaps.

When applying heapify^{-1} in step 2 on both heaps H_1 and H_2 the value at node x_{a_h} got to the root and shifted all the values on the path from the root to node x_{a_h} one step down. We claim now that when we apply heapify in step 6 on the root location in the modified heaps letting the value at the root 'float' down, the value 'floats' exactly along this previously shifted path at least until it reaches the location of the parent of node a_{j-1}. This returns all the values on this sub-path to their original positions. This is true since the modified heaps are only different in the sub-heap of node a_{j-1}.

This means that the value at the root will get to node a_j. This value is the same in both sub-heaps because it is the value that was at node x_{a_h} in both heaps and got to the root by the heapify^{-1} at step 2. Since x_{a_h} is a node in the equal sub-heap this value is the same in both heaps.

From this point (where the value float and reached node a_j) both sub-heaps of the children of node a_j are the same in both heaps, therefore from this location the value continues to 'float' down the same in H_1' and H_2' resulting in the same sub-heap under node a_j. Thus, in the end the sub-heap of node a_j is the same in both H_1' and H_2'. □

We now ready to prove our main lemma regarding recursive-insert:

Lemma 38 *Let H be a heap of size n. Let $a_h, a_{h-1}, \ldots, a_1$ be the path from the root to the first vacant place. Let $(x_{a_1}, x_{a_2}, \ldots, x_{a_h})$ be a proper random choice for the nodes $(a_1, a_2, \ldots, a_h)$. Let v be a new distinct value and let j be the maximal index for which $v_{a_j} < v$ in H. Then:*
recursive-insert-try-2$(H, v, (x_{a_1}, \ldots, x_{a_h}))$= recursive-insert$(H, v, (x_{a_1}, \ldots, x_{a_h}), j)$

Proof: The proof is by induction on the height of the heap.

Induction Base: When the heap is of height 0, recursive-insert-try-2 and recursive-insert operate the same, therefore the lemma holds.

Induction Step: Consider a heap of height h and assume the claim holds for all heaps of height less than h. Assume without loss of generality that the first step on the path from the root to the first vacant place goes left. We partition the analysis into two cases according to whether recursive-insert changed x_{a_h} to a_h in step 2 or not. Notice that if x_{a_h} is changed to a_h then heapify in step 3 does not modify the heap.

Case 1: x_{a_h} remains unchanged in step 2 of recursive-insert. If this is the case then the lemma follows directly from the induction hypothesis. This is true because the recursive-insert and recursive-insert-try-2 are now applied recursively on the identical sub-heap H_L (in step 5). This sub-heap is of height less than h. Thus, H'_L returned is the same in both recursive-insert and recursive-insert-try-2 and we are done.

Case 2: x_{a_h} is changed to a_h in step 2 of recursive-insert. By the operation of recursive-insert this means that x_{a_h} is not in the sub-heap of node a_j.

Let $(H'_L, H'_R, v_{x_{a_h}})$ =heapify$^{-1}(H, x_{a_h})$ and let H_L, H_R be the original sub-heaps of the root of heap H. In this case recursive-insert is applied recursively H_L while recursive-insert-try-2 is applied on the heap H'_L. Let $H_L(end)$ = recursive-insert-try-2$(H_L, v, (x_{a_1}, \dots, x_{a_{h-1}}))$, $H'_L(end)$ = recursive-insert-try-2$(H'_L, v, (x_{a_1}, \dots, x_{a_{h-1}}))$.
Notice we operate recursive-insert-try-2 and not recursive-insert in both cases. In this case the following equalities hold:

$$\begin{aligned}
&\text{recursive-insert-try-2}(H, v, (x_{a_1}, x_{a_2}, \dots, x_{a_h}))\\
&= \text{heapify}(\text{recursive-insert-try-2}(H'_L, v, (x_{a_1}, x_{a_2}, \dots, x_{a_{h-1}})), H'_R, v_{x_{a_h}})\\
&= \text{heapify}(\text{recursive-insert-try-2}(H_L, v, (x_{a_1}, x_{a_2}, \dots, x_{a_{h-1}})), H_R, v_{a_h}) \quad (12)\\
&= \text{heapify}(\text{recursive-insert}(H_L, v, (x_{a_1}, x_{a_2}, \dots, x_{a_{h-1}}), j), H_R, v_{a_h}) \quad (13)\\
&= \text{recursive-insert}(H, v, (x_{a_1}, x_{a_2}, \dots, x_{a_h}), j)
\end{aligned}$$

Equality 13 follows by the induction hypothesis (note that the same heap H_L is used when applying the induction hypothesis). The main point here is equality 12. For this, we need to show that the result of recursive-insert-try-2 is not changed by setting $x_{a_h} = a_h$. First observe that the maximal index for which $v_{a_j} < v$ is the same in H_L and H'_L and that the sub-heap of node a_j is the same in H'_L and H_L, because the x_{a_h} is a location not in the sub-heap of a_j. Thus, by claim 37 $H'_L(end)$ and $H_L(end)$ remain equal in the sub-heap of node a_j. The value v appears in a_j by the end this routine. By claim 36 other parts of the heaps H_L and H'_L are not modified by recursive-insert-try-2.

When we applied heapify^{-1} in step 2 in the procedure recursive-insert-try-2

```
procedure insert-oblivious($H$: Heap, $a$: Value) : Heap
begin
1.  Think of the input value $a$ as being located in an additional node
    numbered $n+1$. (This is the first vacant place as in figure 5.)
    Choose uniformly at random a number $i$, $1 \leq i \leq n+1$,
    and denote the value of node $i$ by $v_i$.
2.  If ($i = n+1$) then H'=H and skip to step 5. Otherwise:
3.      $H' \leftarrow$ increase-key-oblivious($H, i, a$)
4.  Choose uniformly at random a proper randomization vector $(x_{a_1}, \ldots, x_{a_h})$
    for the procedure recursive-insert.
5.  Find the maximal index $j$ in $H$ for which $a_j < v_i$.
6.  Return (recursive-insert($H', v_i, (x_{a_1}, x_{a_2}, \ldots, x_{a_h}), j$))
end
```

Fig. 17. The procedure insert-oblivious

getting the heap H'_L the value at node x_{a_h} got to the root and shifted all the values on the path from the root to node x_{a_h} one step down. We claim that when we apply heapify in step 6 of recursive-insert-try-2 on the root location of the modified heap letting the value at the root 'float' down, the value 'floats' exactly along this previously shifted path returning all the values on this path to their original positions. This path does not intersect the modified sub-heap of node a_j, thus if the path does not pass the parent of node a_j there is no problem. If the path passes through the parent of node a_j then we claim that the value never switches with the value v at node a_j (i.e does not change its route). This is true because the value previously at location of the parent of a_j in H is strictly larger than v. This value is either the value that floats down from the root (if x_{a_h} is the location of the parent of a_j), or it was shifted one step down to the direction of x_{a_h} and is now the sibling of node a_j and larger than v. Thus, all the values on the shifted path return to their places. The sub-heap of node a_j remains the same. Thus, after applying heapify in step 6 the heaps $H_L(end)$ and $H'_L(end)$ are equal and we are done. □

We provide the pseudo code of inset-oblivious in figure 17, and state our last corollary proving its history independence.

Corollary 39 *Let H be a heap of size n that is uniformly distributed among all heaps with the values $\{v_1, v_2, \ldots, v_n\}$, and let a be new distinct value. Then H' = inset-oblivious(H, a) is distributed uniformly among all heaps with the values $\{v_1, \ldots, v_n\} \cup \{a\}$.*

Proof: We only need to prove that for any heap H and new distinct value a, the procedures insert-try-2 and insert-oblivious produce the same output distribution. The only difference in the procedures is the use of recursive-insertinstead of recursive-insert-try-2. Thus, the rest of the proof follows from claim 38. □

It remains to analyze the time complexity analysis of insert-oblivious.

Claim 40 *The expected time complexity of* insert-oblivious *is $O(\log(n))$, where the expectation is over all random choices of the operation.*

Proof: The complexity of increase-key-oblivious operation is no more than the height of the heap, since the value can float at most from the root to one of the leaves, or from one of the leaves to the root. Therefore the worst case complexity of steps 1 to 3 is $O(h) = O(\log(n))$. In step 4 we choose random vector of size $O(h)$ this takes $O(h)$ time. In step 5 we pass over the path from the root to the first vacant place finding the maximal index j for which $v_{a_j} < v_i$. This take $O(h)$ time. Thus, the only problematic part is the expected time complexity of recursive-insert. This complexity depend upon the random vector $(x_{a_1}, x_{a_2}, \ldots, x_{a_h})$, the random choice of the value v_i and the heap H.

Looking at the procedure recursive-insert we can see whenever x_{a_h} is not a location in the sub-heap under a_j we get that the operation of heapify^{-1} in step 3 does not change the heap. Therefore since the value v_i must be less than the value at the root (otherwise x_{a_h} is always in The sub-heap of node a_j) the operation of heapify at step 7 does not modify the heap as well. In fact, in a real implementation if this is the case, we probably skip both steps.

Next we observe that whenever x_{a_h} is a location inside the sub-heap of node a_j then $j \leftarrow j - 1$ (in step 2). When this happens the two operation heapify^{-1} in step 3 and heapify in step 7 are not redundant. Both operation work in worst time complexity of $O(h) = O(\log(n))$. This means that the complexity of recursive-insert is $O(j * h)$ where j is the starting value in the first call to recursive-insert. The rest of the proof analyzes the index j proving that its expected value is $O(1)$.

The value of j depends upon the heap H and the value of v_i inserted to the heap. The main key for the complexity proof are two observations: From the second part of claim 36 we know that j is the height of the value v_i at the end of the operation recursive-insert. From claim 38 and claim 34 we get that:

$$\begin{aligned} & \textsf{recursive-insert}(H, v_i, (x_{a_1}, x_{a_2}, \ldots, x_{a_h}), j) \\ = \; & \textsf{recursive-insert-try-2}(H, v_i, (x_{a_1}, \ldots, x_{a_h})) \\ = \; & \textsf{recursive-insert-try-1}(H, v_i, (x_1, x_2, \ldots, x_n)) \end{aligned}$$

Notice that again we treat recursive-insert and recursive-insert-try-2 as if they get full vector of size n, but uses only the random choices that they need for their operation.

Recall that the operation recursive-insert-try-1 consist of using build-heap^{-1} on H, putting v_i in the next vacant place in the tree and using build-heap to build back the tree. The value v_i is chosen uniformly from $\{v_1, v_2, \ldots, v_n\} \cup \{a\}$ where a is the new value that is inserted. Last, from the second part of claim 32 after applying $\textsf{build-heap}^{-1}$ we get uniform permutation over the values $\{v_1, v_2, \ldots, v_n\} \cup \{a\} \setminus \{v_i\}$. Combining these facts together we get that the expected value of j is:

$$E\left[\pi \in_R \Pi(n+1); \textsf{build-heap}(v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n+1)}) = H; h(H, v_{\pi(n)})\right]$$

Where $h(H, v_{\pi(n)})$ is the height of $v_{\pi(n)}$ in the heap H. From claim 30 this value is less or equal 4$= O(1)$, and we are done. □

7 Conclusion

In this work we showed a separation between the notion of weak and strong history independence in the comparison-based model. In this model we showed that implementing strong history independence requires a very high complexity penalty. A major open question is whether the two notions are different in the standard non-comparison-based model. We believe that achieving strong history independence is difficult even in the standard model.

A second interesting question is whether weak history independence imposes a complexity cost. That is, can one transform any data structure into being weakly history independent without paying any complexity penalty? Or by paying a small complexity penalty? Up to now, transformations with constant additional costs have been shown to several data structures: 2-3 trees, Hash-tables and Heaps. We believe that the general result is not possible. To show this, one must find a specific data structure and show that making it weakly history independent imposes a complexity cost. As a candidate for proving such a lower bound, we propose the Fibonacci heaps data structure.

References

[1] A. Andersson, T. Ottmann. Faster Uniquely Represented Dictionaries. Proc. 32nd IEEE Sympos. Foundations of Computer Science, pages 642–649, 1991

[2] M. Bellare, O. Goldreich and S. Goldwasser. Incremental Cryptography and Application to Virus Protection. Proc. of the 27th annual ACM Symp. on the theory of Computing. pp. 45-56, 1995.

[3] Niv Buchbinder, Erez Petrank. Lower and Upper Bounds on Obtaining History Independence. Proc. of the 23rd Annual Int. Cryptography Conference (CRYPTO 2003). pp. 445-462, 2003.

[4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. Introduction to Algorithms. MIT Press and McGraw-Hill Book Company, 6th edition, 1992.

[5] E.W.Dijkstra A note on two problems in connexion with graphs. Numerische Mathematik, 1:269-271, 1959

[6] Jason D. Hartline, Edwin S. Hong, Alexander E. Mohr, William R. Pentney, and Emiliy C. Rocke. Characterizing History independent Data Structures. ISAAC 2002 pp. 229-240, 2002.

[7] 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.

[8] Robert W. Floyd. Algorithm 245 (TREESORT). Communications of the ACM, 7:701, 1964.

[9] D. Micciancio. Oblivious data structures: Applications to cryptography. In Proc. 29th ACM Symp. on Theory of computing, pages 456-464, 1997.

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

[11] M.Naor and V.Teague. Anti-persistence: History Independent Data Structures. Proc. 33rd ACM Symp. on Theory of Computing, 2001.

[12] R.C.Prim. Shortest connection networks and some generalizations. Bell System Technical Journal, 36:1389-1401, 1957

[13] J.W.J Williams Algorithm 232 (HEAPSORT). Communication of the ACM, 7:347-348, 1964

A Proof that Strong history independence implies canonical representation

In this section we provide the proof of Lemma 3. As stated in the introduction, this lemma was proven in [6] and independently by us. The proof here slightly differs from the one in [6].

We say that a memory representation of a data-structure D is *reachable*, if there exists a sequence of operations (and a sequence of random choices for each of these operations) that yields D with the given implementation. Each content of the data structure may have several possible memory representations. An implementation of an abstract data structure can be viewed as a function mapping possible contents to memory representations and some algorithmic way of passing between these memory representations according to the content graph.

Lemma 41 *For any well-behaved data-structure, for any strongly history independent implementation of the data-structure, for any reachable memory representation D, and for any operation* Op *applied on D with some parameters $v_1, v_2, \ldots, v_k$. The operation yields only one memory representation.*

Note that the above lemma must hold even though the procedures implementing the data structure operations may be randomized.

Proof: Assume in a way of contradiction that there exists a reachable memory representation D an operation Op and additional parameters to Op, $v_1, \ldots v_k$ so that the operation may yield at least two memory representations D_1 and D_2 for $D_1 \neq D_2$. Since D is reachable, there exists a sequence S and a sequence of random coin-tosses that results in D. Let C be the content of the data structure. Let C' be the content of the data structure after applying Op with its parameters on C. Let S' be the sequence of operation on the path from C' to C in the content graph. The graph is strongly connected therefore there exist such path.

We define two sequences of operations starting from the empty data structure: $S_1 = S$ and $S_2 = (S_1, Op(\cdot, v_1, \ldots, v_k), S')$ ($\mathsf{Op}(\cdot, v_1, \ldots, v_k)$ means that we apply Op on the structure output by the previous steps of the sequence). Note that S_2 generates a structure with the same content as D after running S_1 and in the end. By strong history independence, we may choose stop-points in S_1 and S_2 when they contain the same content and get an equal distribution on memory representation tuples at those points. We choose two stop-points for each sequence. In both stop-points, the sequences result in the content of D. In S_1 both points are defined at the same location: the end of S_1. In S_2 one point is at the end of S_1 and the other one is at the end of S_2. Since the content of the data-structure is the same on both points, then the distribution of memory representation at the points must be identical. Since it is identical, it remains identical also when we condition on the first point being the memory representation D. We know that the conditioned event has positive probability since D is reachable. For S_1 the memory representation in both points (actually, the same point) is equal and must be (D, D). Thus, the memory representation in the points of S_2 (conditioned on the first being D) must also be (D, D). This means that for any $D_i = \mathsf{Op}(D, v_1, \ldots, v_k)$, it must hold that $S'(D_i) = D$ with probability 1, where S' means applying the sequence of operations in S' on D_i one by one.

Next we define two more sequences: $S_3 = (S_1,\ Op(\cdot, v_1, \ldots, v_k))$ and $S_4 = (S_3, S', Op(\cdot, v_1, \ldots, v_k)$. We choose two stop points for each of these sequences. For S_3 we choose both points at the end of S_3. For S_4 we choose the first point after S_3 and the second point at the end of S_4. Note that the content of the data structure in all these points the same, C'. By strong history independence the joint distribution on memory representations at the stop-points of S_3 (which is the same representation) must also be the joint distribution of the memory representations at the stop-points of S_4. Thus, the two points in S_4 must contain the same memory representation. Now, we already know that for any $D_i = Op(D, v_1, \ldots, v_k)$, it must hold that $S'(D_i) = D$. But here we get that for any such possible D_i, $\mathsf{Op}(S'(D_i), v_1, \ldots, v_k)$ must be D_i. Combining the two, we get that for any D_i, $\mathsf{Op}(D, v_1, \ldots, v_k)$ must be D_i for any i. This latter requirement results in a contradiction if there is more than one possible such D_i. Thus, there can only be one memory representation for $\mathsf{Op}(D, v_1, \ldots, v_k)$. □

Using the previous lemma we may now prove the lemma 3. We prove that 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.

Proof: Let C be any content of the data structure and let S_1 be any sequence of operation that yields this content. From lemma 41 each operation in the sequence yields only one possible memory representation, thus the content has only one possible memory representation. This is true for any sequence of operations that yield C. By the history independence of the data structure implementation (even not using strong history independence) the memory representation must be the same for each such sequence, and we are done. □

It worth nothing to say that lemma 3 does not hold when the data structure is not well behaved (i.e. when its content graph is not strongly connected). Consider for example a data structure that stores a set of elements and has only an insert operation and some other operations that do not change its content. It is not hard to see that the content graph of this data structure is a **DAG** and therefore not strongly connected. Indeed, an implementation that stores the values in an array, keeping the filled array uniformly distributed at any time regardless of the history led to this content is a strongly history independent implementation of this data structure, which is of-course not canonical.

In general, lemma 3 applies to any strongly connected part of the content graph which is of size strictly more than 1 (i.e. to any content that lie on a circle). In the opposite direction, each content that do not belong to such a strongly connected part may have multiple possible memory representations. Such a content may appear in any sequence S of operations only once. For this reason, it may have few possible memory representations, as long as the probabilities of these memory representations do not depend upon previous operations that lead to this content, the data structure remains strongly history independent, because the common distribution remains identical.