Checking and Spot-Checking the Correctness of Priority Queues

Matthew Chu[1], Sampath Kannan[1], and Andrew McGregor[2]

[1] Dept. of Computer and Information Science, University of Pennsylvania
{mdchu,kannan}@cis.upenn.edu
[2] ITA Center, University of California, San Diego
andrewm@ucsd.edu

Abstract. We revisit the problem of memory checking considered by Blum et al. [3]. In this model, a checker monitors the behavior of a data structure residing in unreliable memory given an arbitrary sequence of user defined operations. The checker is permitted a small amount of separate reliable memory and must fail a data structure if it is not behaving as specified and pass it otherwise. How much additional reliable memory is required by the checker? First, we present a checker for an implementation of a priority queue. The checker uses $O(\sqrt{n}\log n)$ space where n is the number of operations performed. We then present a *spot-checker* using only $O(\epsilon^{-1}\log\delta^{-1}\log n)$ space, that, with probability at least $1-\delta$, will fail the priority queue if it is ϵ-far (defined appropriately) from operating like a priority queue and pass the priority queue if it operates correctly. Finally, we then prove a range of lower bounds that complement our checkers.

1 Introduction

Program checking [4] is a paradigm for gaining confidence at run-time in the output produced by a program by running an auxiliary program called the *checker* to verify the correctness of the output on the current input. Checkers are allowed to be probabilistic and have a specifiably small probability of themselves making a mistake; however, this probability of error depends only on the checker's internal coin tosses and not on the presence or absence of any particular bug in the program being checked. Checkers may also query the program being checked on additional inputs and use the self-consistency of the outputs to determine correctness. Checkers designed for a particular computational problem can check any program that claims to solve the problem.

Since checkers are run on-line, they should be efficient and should not introduce a significant overhead in the program's running time and resource use. Sometimes it is possible and desirable to design highly efficient checkers that only verify that the program output is *close* to the true output in a quantifiable way. Such checkers, known as *spot-checkers* [6] do not even look at the entire input and output of the program. Spot-checking is related to the idea of *property testing* [10] which seeks to examine small portions of the input to determine if it is close to having a specified property.

L. Arge et al. (Eds.): ICALP 2007, LNCS 4596, pp. 728–739, 2007.

The problem of checking the correctness of the behavior of data structures residing in cheap but potentially unreliable memory using a small amount of reliable memory was first considered by Blum et al. [3]. In contrast to program checking, we emphasize that we are less concerned with ensuring that the data structure has been correctly coded but rather wish to ensure that the use of unreliable memory has not caused incorrect behavior. However, in the spirit of the program checking framework, we will assume a fully general scenario in which an adversary controls both the sequence of queries and updates to the data structure and the exact state of the data structure at each point.

A checker equipped with a small amount of reliable (but not necessarily secret) memory observes the sequence of operations and the results produced by the data structure and must decide whether it behaved correctly. Two flavors of checkers are considered in [3] — an *off-line* checker that makes its determination after an entire sequence of accesses have been made to the data structure, and an *on-line* checker that must immediately report an error when it occurs. These models were further explored in the context of simple linked data structures such as linked lists, trees, and graphs in a paper by Amato and Loui [2]. However, to date, no efficient checkers have been designed for data structures such as priority queues and search trees that maintain global order properties amongst the keys they store. We note that there has been some work done on designing new data structures that are resilient to memory faults, e.g., Finocchi et al. [9], but stress that this is a different problem than checking correctness.

The memory checkers of Blum et al. [3] and Amato and Loui [2] fit in a model of computation called the streaming model [11,1,7]. In this model a computer with a small amount of memory observes an adversarially generated stream and determines (with high probability) whether the stream satisfies a specified property. The notion of spot-checking in such a model [8] is a little different from the standard notion — rather than requiring the spot–checker to sample only a few places in the input and output, we allow the spot-checker to observe the entire stream of inputs and outputs, but require that it operate with the space constraints imposed by the streaming model.

A common implementation of priority queues is as binary heaps. A heap is a data structure that is used to easily remove the minimum key (min-heaps) or the maximum (max-heaps). For the purposes of this paper, we will consider min-heaps, but all conclusions can easily be applied to max-heaps as well. Heaps are usually implemented as binary trees, explicitly or implicitly as an array. Each node has a key, a right child, and a left child. There are two main properties of all heaps. The first is that all descendants of a certain node have a key that is greater than the node's key. The second property is that the tree is filled in order from left to right, level by level. All add and delete operations run in $O(\log n)$ time, where n is the number of nodes in the tree.

1.1 Our Results

In this paper we present results on the checking and spot-checking of priority queues. We will assume that the priority queue is implemented in unreliable

memory and that the checker observes the values inserted into the priority queue and the result of each extract operation. We will present our results by referring to a binary-heap used to implement the priority queue but our checkers will be able to check any implementation. Note that our goal is only to verify that the input/output behavior of the implementation is correct. We do not (and in this model, cannot) actually verify that the priority queue is implemented as a heap and that this heap satisfies the structure property.

Our checkers are all off-line and assume that the sequence of operations start and end with the empty heap. We present a $O(\sqrt{n}\log n)$-space checker and an $O(\epsilon^{-1}\log\delta^{-1}\log n)$-space spot-checker in Section 3 and Section 4 respectively. In Section 5 we present lower bounds that show that any on-line checker requires $\Omega(n/\log n)$ space, that any deterministic checker requires $\Omega(n)$ space, and that our checker is near-optimal among checkers of a certain type.

2 Preliminaries

We start by adapting the definition of a memory checker from [3] for the checking of heaps. See Fig. 1 for an accompanying figure.

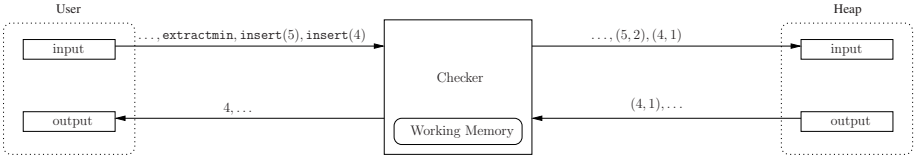

Fig. 1. Memory Checker interaction with User and Data Structure

Definition 1 (Heap Checker). *A* heap checker *for heap $\mathcal{H}$ is an (probabilistic) algorithm C with access to the following four tapes:*

1. *A "user" input tape from which the checker reads user specified operations. Each operation is either an* `insert`$(\cdot)$ *where the argument is a value to be inserted or an* `extractmin`*.*
2. *A "user" output tape on which the checker writes a value for each* `extractmin` *operation or alternatively writes* `FAIL` *and terminates if the checker has determined that the heap checker is not operating correctly. If the checker never outputs* `FAIL` *during the processing of the user specified operations, then the checker concludes by writing* `PASS` *on this tape.*
3. *A "heap" input tape on which the checker specifies operations to $\mathcal{H}$. On a user operation* `insert`(u)*, the checker requests the insertion of the tuple (u, t) into the heap where t is the index of the user operation*[1]*, referred to as the* time-stamp*. On a user* `extractmin` *operation, the checker requests that the heap extracts the smallest value presently stored in the heap.*

[1] In the actual implementation of the heap the t is encoded as lower order bits and u is encoded as higher order bits. A consequence of this is that if (u, t) and (u, t') are concurrently in memory, then (u, t) should be extracted before (u, t') if $t < t'$.

4. *A "heap" output tape from which the checker reads the output of each operation. The user* `extractmin` *operation will correspond to a tuple* (u, t) *being read from the heap.*

In addition, the checker has a limited amount of working memory.

We can abstract the input of the heap checker as a sequence that encodes both the user specified operations and the output of the heap. We define this sequence and what it means for the sequence to be "heap-like" as follows.

Definition 2 (Interaction Sequence and Heap-like). *An* interaction sequence *is a length* $2n$ *sequence of tuples* $S = c_1 \dots c_{2n}$ *where each tuple* c_j *is either an* `insert` *of some value* u*, denoted* $\langle \texttt{insert}, u \rangle$*, or an* `extractmin` *operation that returns some value* v *that purports to have inserted at time* t*, denoted* $\langle \texttt{extractmin}, (v, t)) \rangle$*. Furthermore, we require that for any* $k \in [2n]$*, the number of* `extractmin` *operations among* $c_1 \dots c_k$ *is at most the number of* `insert` *operations. We define an ordering of the tuples in the natural way:* $(u, t) < (u', t')$ *iff* $u < u'$ *or* $(u = u'$ *and* $t < t')$*. We say an interaction sequence is* heap-like *if for all* `extractmin` *operations* c_j

$$c_j = \langle \texttt{extractmin}, (u, t) \rangle \Rightarrow (u, t) = \min M_{j-1} \ , \tag{C}$$

where,

$$M_0 = \emptyset \text{ and } M_j = \begin{cases} M_{j-1} \setminus \min M_{j-1} & \text{if } c_j = \langle \texttt{extractmin}, (\cdot, \cdot) \rangle \\ M_{j-1} \cup \{(u, j)\} & \text{if } c_j = \langle \texttt{insert}, u \rangle \end{cases} .$$

Conceptually, M_j *is the set of (value, time-stamp) tuples still to be extracted from the heap if the heap were operating correctly.*

It will be convenient for us to decompose the heap-like condition into three separate conditions in the case that we start and end with an empty heap. We prove these three condition are together equivalent in the following lemma.

Lemma 1 (Equivalent Definition for Heap-like). *An interaction sequence* S *is heap-like iff* S *satisfies the following three conditions,*

$$\{(u, t) : c_t = \langle \texttt{insert}, u \rangle\} = \{(u, t) : \langle \texttt{extractmin}, (u, t) \rangle \in S\} \ , \tag{C1}$$

$$\forall c_{t_b} = \langle \texttt{extractmin}, (\cdot, t_a) \rangle, t_a < t_b \ , \text{ and} \tag{C2}$$

$$\forall c_{t_b} = \langle \texttt{extractmin}, (u, t_a) \rangle, c_{t_{b'}} = \langle \texttt{extractmin}, (u', t_{a'}) \rangle, \\ (u, t_a) < (u', t_{a'}) \Rightarrow (t_{b'} < t_a \text{ or } t_b < t_{b'}) \ . \tag{C3}$$

These conditions correspond to the fact that 1) the set of (value, time-stamp) pairs inserted should match the (value, time-stamp) pairs extracted, 2) a (value, time-stamp) should only be extracted after it has been inserted and 3) a pair (u', t') *should not be extracted between the insert and extraction of a* (u, t) *pair if* $(u, t) < (u', t')$*.*

Proof. If S satisfies (C) then clearly (C3), (C2), and (C1) are satisfied. Conversely assume S satisfies (C3), (C2), and (C1). For the sake of contradiction, assume that there exists a $t_{b'}$ such that, $c_{t_{b'}} = \langle \texttt{extractmin}, (u', t_{a'}) \rangle$ and $(u', t_{a'}) \neq \min M_{j-1}$. and assume $t_{b'}$ is the smallest such value. But then there exists $(u, t_a) \in M_{j-1}$ with $(u, t_a) < (u', t_{a'})$. By the minimality of $t_{b'}$ and (C1), $t_a < t_{b'} < t_b$ for some t_b such that $c_{t_b} = \langle \texttt{extractmin}, (u, t_a) \rangle$. Hence S violates (C3) which is a contradiction. Hence S satisfies (C) after all.

Next we define what it means for a sequence to be far from heap-like. It will turn out that it is relatively straight-forward to ensure that a sequence satisfies (C2) and (C1). Hence, we only define distances between sequences that satisfy these two properties. Intuitively the distance between two interaction sequences S and S' will be the least number of moves that are necessary to transform S into S'. However, the exact definition is a little more awkward because of the (value, time-stamp) pairs in the `extractmin` operations.

Definition 3 (ϵ-far from Heap-like). *Consider an interaction sequence $S = c_1 c_2 \dots c_{2n}$ and let σ be a permutation on $[2n]$. Let $S_\sigma = c'_1 c'_2 \dots c'_{2n}$ be the sequence where,*

$$c'_i = \begin{cases} \langle \texttt{insert}, u \rangle & \text{if } c_{\sigma^{-1}(i)} = \langle \texttt{insert}, u \rangle \\ \langle \texttt{extractmin}, (u, \sigma(t)) \rangle & \text{if } c_{\sigma^{-1}(i)} = \langle \texttt{extractmin}, (u, t) \rangle \end{cases}.$$

We define $\mathrm{dist}(S, S_\sigma)$ *as the number of edits required to sort* $(\sigma(1), \dots, \sigma(2n))$ *where an edit is of the form "move the value in position k and insert it at position j." In particular we say S is ϵ-far from being heap-like if for all σ such that S_σ is heap-like,* $\mathrm{dist}(S, S_\sigma) \geq \epsilon n$.

So, for example, for $\sigma = (1 \to 1, 2 \to 3, 3 \to 4, 4 \to 2)$ and

$$S = \langle \texttt{insert}, 15 \rangle \langle \texttt{insert}, 16 \rangle \langle \texttt{extractmin}, (16, 2) \rangle \langle \texttt{extractmin}, (15, 1) \rangle$$

then, $S_\sigma = \langle \texttt{insert}, 15 \rangle \langle \texttt{extractmin}, (15, 1) \rangle \langle \texttt{insert}, 16 \rangle \langle \texttt{extractmin}, (16, 3) \rangle$. In this case, $\mathrm{dist}(S, S_\sigma) = 1$ and since S_σ is heap-like, S is at most $1/2$-far from being heap-like.

3 An $O(\sqrt{n} \log n)$-Space Checker for Heaps

In this section we present a memory checker that accepts an interaction sequence that is heap-like and, with probability at least $1 - \delta$, rejects an interaction sequence that is not heap-like. The algorithm uses $O(\sqrt{n} \log n + \log(1/\delta) \log n)$ space and processes each term of the interaction sequence in $O(\log n + \log(1/\delta))$ time. Hence forth we assume that $\delta > 1/n$ and therefore omit the δ dependencies.

To ensure that the interaction sequence satisfies (C1), we use the ϵ-biased hash function construction of Naor and Naor [12]. Their relevant properties are presented in the following theorem.

Theorem 1 (ϵ-biased Hash Function [12]). *Consider two n-bit binary strings x and y. There exists a randomized scheme using $O(\log n + \log \delta^{-1})$ random bits that constructs a hash function h such that $\Pr(h(x) = h(y)) \leq \delta$ if $x \neq y$. Furthermore $h(\cdot)$ can be computed in $O(\log n + \log \delta^{-1})$ space even if the string to be hashed is revealed bit by bit in some arbitrary order.*

Using such a function we hash (value, time-stamp) pairs inserted and extracted to ensure that, with probability at least $1 - \delta$, condition (C1) is satisfied. It is easy to check that condition (C2) is also satisfied as for each operation $c_{t_b} = \langle \texttt{extractmin}, (\cdot, t_a) \rangle$ it is sufficient to check that $t_a < t_b$.

To ensure condition (C3), the algorithm maintains two lists

$$E = \{(v_1, t_1), \ldots, (v_{|E|}, t_{|E|})\},$$

a list of (value, time) pairs, and B, a list of recently inserted values and their time-stamps. The list E will define a series of *epochs* and B will be used to buffer (value, time) between the creation of new epochs. Specifically, the list E will be sorted such that $t_1 < t_2 < \ldots < t_{|E|}$. We then refer to the period of time $T_i = \{t : t_{i-1} < t \leq t_i\}$ as the ith epoch (where $t_0 = 0$). We define an epoch to the period $T_{|E|+1} = \{t : t > t_E\}$ as the *current epoch.*

The list B will contain all the values that have been inserted into the heap since the last pair was added to E not including those that have been returned by an `extractmin` operation. Together, the state of B and E at any point in the algorithm define the function,

$$f(t) = \begin{cases} (v_i, t_i) \text{ if } t \in T_i \text{ for } i \leq |E| \\ \min B \text{ if } t \in T_{|E|+1} \end{cases}.$$

The semantics of f is such that at any time, if (u, t) has been inserted but not extracted, then, $f(t) \leq (u, t)$ if the heap is performing correctly. f is potentially updated in every iteration of the algorithm. See Figure 2 for a schematic of the update and utility of the function f. The algorithm maintains B and E such that there are at most $\sqrt{n}$ tuples in both sets. This is achieved by using the set B to, in effect, buffer inserted tuples such that at least $\sqrt{n}$ tuples must be inserted for $|E|$ to increase by 1. The algorithm is presented in Figure 3.

Theorem 2. *Algorithm Heap-Checker is a checker for the correctness of heaps. It uses $O(\sqrt{n} \log n)$ memory and runs in $O(\log n)$ time per term of the interaction sequence.*

Proof. We first prove the correctness of the *Heap-Checker* algorithm and then bound the space and time complexity.

Correctness: If S is heap-like then, the tester will `PASS` the sequence since property (C1) ensures that the algorithm does not fail in Line 1 and properties (C2) and (C3) ensure that the algorithm does not fail in Line 9. Conversely, if S does not satisfy property (C1) or (C2), the checker will fail the sequence at Line 1 or 9. Assuming that S has properties (C1) and (C2), we now show

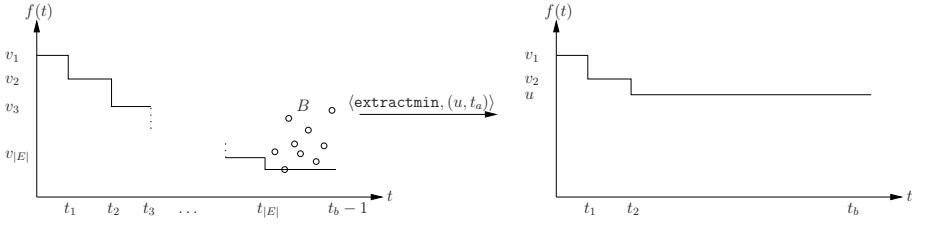

Fig. 2. A schematic depicting the part of the behavior of the algorithm *Heap-Checker* when processing item $c_{t_b} = \langle \texttt{extractmin}, (u, t_a) \rangle$ where $t_2 < t_a < t_3$. First it is checked that $f(t_a) \leq (u, t_a)$ and, if so, $\{(v_i, t_i) \in E : v_i < u\}$ is removed from E and (u, t_b) is added. Furthermore B is emptied.

that the checker will fail at Line 9 at some point during the processing of the interaction sequence. If property (C3) is not satisfied then consider the smallest t_b such that $c_{t_b} = \langle \texttt{extractmin}, (u, t_a) \rangle$ and there exists a $t_{b'}$ with $c_{t_{b'}} = \langle \texttt{extractmin}, (u', t_{a'}) \rangle$ with $(u, t_a) < (u', t_{a'})$ and $t_a < t_{b'} < t_b$. We consider two cases:

1. At time $t_{b'}$ assume t_a is in the current epoch. Therefore $(u, t) \in B$ and hence $(u', t') > (u, t) \geq \min B$. Therefore the algorithm fails in iteration $t_{b'}$ at line 9.
2. Otherwise assume that, at the start of iteration $t_{b'}$, t_a is not in the current epoch, i.e. $t_a < t_{|E|}$. Therefore, at the end of the iteration $v_{|E|} \geq u'$. But then at the start of iteration t_b, $f(t_a)$ is also at least u'. Since at iteration t_b we have $f(t_a) \geq u' > u$, the algorithm fails at this iteration during Line 9.

Space Use: It is clear that there are never more than $\sqrt{n}$ values stored in B. To bound the size of E we consider the following two ways in which a pair is added to E.

1. Line 7: Note that there can be at most $\sqrt{n}$ such addition since in between each such addition there must be at least $\sqrt{n}$ `insert` operations.
2. Line 15: Note that there is no net increase in the size of E in this step.

Therefore, the maximum number of pairs stored in E is $\sqrt{n}$. Hence the space use of the algorithm is $O(\sqrt{n} \log n)$ as claimed.

Running Time: For keeping track of the current epoch, the checker can keep a heap of the values in B in its own reliable memory. This means that all insert and delete operations can be done in $O(\log n)$ time. When updating the tuples in E the checker can do a binary search ($O(\log n)$ time) through the values in each tuple since these are in sorted order. Since these are the only operations the checker must perform, it runs in $O(\log n)$ time.

4 Spot-Checker

In this section we present a memory checker that accepts an interaction sequence that is heap-like and with probability at least $1-\delta$ rejects an interaction sequence

Algorithm *Heap-Checker*$(S = c_1 c_2 \ldots c_{2n})$

1. Use ϵ-biased hashing to ensure that,

$$\{(u,t) : c_t = \langle \texttt{insert}, u\rangle\} = \{(u,t) : \langle \texttt{extractmin}, (u,t)\rangle \in S\}$$

   and if not **return** `FAIL`
2. Maintain two lists, initially empty,

$$E = \{(v_1,t_1),\ldots,(v_{|E|},t_{|E|})\} \text{ and } B = \{(u_1,t'_1),\ldots,(u_{|B|},t'_{|B|})\}$$

3. **for** $t \in [2n]$
4. **do if** $c_t = \langle \texttt{insert}, u\rangle$
5. **then if** $|B| < \sqrt{n}$
6. **then** $B \leftarrow B \cup \{(u,t)\}$
7. **else** $B \leftarrow \emptyset$ and $E \leftarrow E \cup \{(-\infty, t)\}$
8. **if** $c_t = \langle \texttt{extractmin}, (u,t')\rangle$
9. **then if** $t' \geq t$ or $(u,t') < f(t')$ **then return** `FAIL`
10. **if** $t' > t_{|E|}$
11. **then** $B \leftarrow B \setminus \{(u,t')\}$
12. $E \leftarrow (E \setminus \{(v_i,t_i) \in E : v_i < u\}) \cup \{(\min\{v_{|E|}, u\}, t_{|E|})\}$
13. **if** $t' \leq t_{|E|}$
14. **then** $B \leftarrow \emptyset$
15. $E \leftarrow (E \setminus \{(v_i,t_i) \in E : v_i < u\}) \cup \{(u,t)\}$
16. **return** `PASS`

Fig. 3. The *Heap-Checker* Algorithm

that is ϵ-far from being heap-like. The algorithm uses $O(\epsilon^{-1}\log(1/\delta)\log n)$ space and processes each term of the interaction sequence in $O(\epsilon^{-1}\log(1/\delta))$ time.

As before, we use the hashing techniques described in Section 3 to ascertain (with probability at least $1-\delta/2$) whether the sequence has property (C1) and can simply check for property (C2) by checking that the time-stamp of each extracted value does not exceed the current time. To check for property (C3), the algorithm stores a set of $p = \ln(2/\delta)/\epsilon$ (value, time-stamp) pairs that are chosen at random from all the n such pairs. The hope is that one of the pairs stored will reveal that an interaction is un-heaplike if this is indeed the case. We make the following definition to clarify this notion.

Definition 4 (Revealing Tuples). *We call a tuple* (u,t_a) *a* revealing tuple *if there exists* $t_b, t_{a'}, t_{b'}, u'$ *such that,* $c_{t_a} = \langle \texttt{insert}, u\rangle$, $c_{t_b} = \langle \texttt{extractmin}, (u,t_a)\rangle$, $c_{t_{a'}} = \langle \texttt{insert}, u'\rangle$, $c_{t_{b'}} = \langle \texttt{extractmin}, (u',t_{a'}\rangle$, $t_a < t_{b'} < t_b$, *and* $(u,t_a) < (u',t_{a'}))$.

In the time between the insertion and the extraction of a tuple (u,t_a), the algorithm checks that no extraction returns a value $u' > u$. If this ever occurs then the sequence is not heap-like because the sequence violates property (C3). The crux of the proof of correctness will be that interaction sequence has many revealing tuples if it is far from being heap-like. The following lemma asserts that this is indeed the case. The algorithm is presented in Figure 4.

Algorithm *Heap-Spot-Checker*$(S = c_1 c_2 \dots c_{2n})$

1. Use ϵ-biased hashing to ensure

$$\{(u,t) : c_t = \langle \texttt{insert}, u\rangle\} = \{(u,t) : \langle \texttt{extractmin}, (u,t')\rangle \in S\}$$

2. Let R be a set of $p = \ln(2/\delta)/\epsilon$ values chosen randomly from the set $[n]$
3. Maintain P, a list of at most p (value, insertion time) tuples
4. $i \leftarrow 1$
5. **for** $t \in [2n]$
6. **if** $c_t = \langle \texttt{insert}, u\rangle$ for some u
7. **then if** $i \in R$ **then** $P \leftarrow P \cup \{(u,t)\}$
8. $i \leftarrow i + 1$
9. **if** $c_t = \langle \texttt{extractmin}, (u,t')\rangle$ for some u
10. **then if** $t' > t$ **then return** `FAIL`
11. **if** $\exists\ (v,t_a) \in P$ with $(v,t_a) < (u,t')$ **then return** `FAIL`
12. **if** $(u,t') \in P$ **then** $P \leftarrow P \setminus \{(u,t')\}$
13. **return** `PASS`

Fig. 4. The *Heap-Spot-Checker* Algorithm

Lemma 2. *Assume S satisfies, (C1) and (C2). Then, if S is ϵ-far from being heap-like, there are at least ϵn revealing tuples.*

Proof. Assume S is ϵ-far from being heap-like. Let r be the number of revealing tuples in S. Consider the interaction sequence S and let $t_{b'}$ be the smallest value such that $c_{t_{b'}}$ does not satisfy the heap-condition (C), i.e., $c_{t_{b'}} = \langle \texttt{extractmin}, (u', t_{a'})\rangle$ but $(u', t_{a'}) \geq (u, t_a)$ where $(u, t_a) = \min M_{t_{b'-1}}$. Let t_b be such that $c_{t_b} = \langle \texttt{extractmin}, (u, t_a)\rangle$. But then (u, t_a) is a revealing tuple. Consider rearranging the terms of S by bringing c_{t_b} up to position $t_{b'}$ (and adjusting the position of the other terms and the time-stamps according. We claim this reduces the number of revealing tuples by at least one. To see this note that (u, t_a) is no longer a revealing tuple. Furthermore, note that no other tuple has become revealing.

We repeat this process until there are no operations that do not satisfy the heap-condition. Note that we can do this at most r times since we decrease the number of revealing tuples by one each time. Hence S is at most distance r from being heap-like and therefore $r \geq \epsilon n$.

Theorem 3. *Algorithm Heap-Spot-Checker* `PASS`*es heap-like sequences and* `FAIL`*s sequences that are $\epsilon-$far from being heap-like with probability at least $1 - \delta$. It uses $O(\epsilon^{-1} \log(\delta^{-1}) \log n)$ memory and runs in $O(\epsilon^{-1} \log(\delta^{-1}))$ time.*

Proof. If a sequence is heap-like, the tester returns `PASS` because there will be no revealing tuples. From Lemma 2, we know that if a heap is ϵ-far from being heaplike, there are at least ϵn revealing tuples. Furthermore, if any revealing insertion is sampled then the spot-checker will `FAIL` the sequence. The probability that a revealing insertion is stored by the algorithm is at least,

$$1 - (1 - \epsilon)^{\epsilon^{-1} \ln(2/\delta)} \geq 1 - \delta/2 \ .$$

The space requirement is obvious since the algorithm samples at most $\epsilon^{-1}\ln(2/\delta)$ triples. The running time per-element is also $O(\epsilon^{-1}\ln(1/\delta))$ since at each `extractmin` operation, at most $\epsilon^{-1}\ln(1/\delta)$ tuples need to be checked for a potential violation.

5 Lower Bounds

The checker presented in Section 3 was off-line and required randomization. It would be preferable if the checker could identify any error as soon as the error occurred and, ideally, the checker would be deterministic. We start this section by showing that any checker that had either of these properties would need almost as much reliable working space as the space used to store the data structure.

Theorem 4. *Any on-line checker that is correct with probability at least* $3/4$ *requires* $\Omega(n/\log n)$ *working space. Any deterministic off-line checker requires* $\Omega(n)$ *working space.*

Proof. The proofs will be by reductions from the one-round communication complexity of $\textsc{Prefix} : \{0,1\}^n \times \{0,1\}^n \times [n] \to \{0,1\}$ and $\textsc{Equality} : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ where

$$\textsc{Prefix}(x,y,j) = \begin{cases} 1 \text{ if } \forall i \in [j], x_i = y_i \\ 0 \text{ otherwise} \end{cases}$$

and $\textsc{Equality}(x,y) = \textsc{Prefix}(x,y,n)$.

Suppose there exists an on-line checker $\mathcal{C}$ that correctly identifies the first error in the operation of the heap with probability at least $3/4$. Consider an instance of Prefix where Alice has a binary string $x \in \{0,1\}^n$ and Bob has a binary string $y \in \{0,1\}^n$ and an index $j \in [n]$. Then let Alice run the $\mathcal{C}$ on the sequence $\langle \texttt{insert}, x_1\rangle\langle \texttt{insert}, x_2 + 2\rangle \ldots \langle \texttt{insert}, x_n + 2(n-1)\rangle$ and then communicate the memory state of $\mathcal{C}$ to Bob. Bob instantiates $\mathcal{C}$ with this memory state and continues running $\mathcal{C}$ on the sequence

$$\langle \texttt{extractmin}, (y_1, 1)\rangle\langle \texttt{extractmin}, (y_2, 2)\rangle \ldots \langle \texttt{extractmin}, (y_n + 2(n-1), n)\rangle \ .$$

Then $\textsc{Prefix}(x,y,j) = 1$ iff the $\mathcal{C}$ does not fail until after the jth `extractmin` operation. But it was shown by Chakribatri et al. [5] that Alice needs to send $\Omega(n/\log n)$ bits if Bob is to determine the value of $\textsc{Prefix}(x,y,j)$ with probability at least $3/4$. Hence, the checker requires $\Omega(n/\log n)$ bits.

The proof for the second part of the theorem is similar: Alice has a binary string $x \in \{0,1\}^n$ and Bob has a binary string $y \in \{0,1\}^n$ and wishes to learn $\textsc{Equality}(x,y)$. Alice and Bob create sequences and use a deterministic off-line checker as before. Then $\textsc{Equality}(x,y) = 1$ iff the checker does not fail. But it is known (e.g. Kushilevitz and Nisan [13]) that Alice needs to send a message of length $\Omega(n)$ if Bob is to determine the value of $\textsc{Equality}(x,y)$ with zero error. Hence the memory state of the checker must require $\Omega(n)$ bits.

In the remainder of this section we argue that any checker that operates by storing (value, time-stamp) pairs and their extraction times must use $\Omega(\sqrt{n})$ space if it is to succeed with at least constant probability. This is even the case if properties (C1) and (C2) are guaranteed. Let $k = \sqrt{n}$. Consider the following probabilistic construction of an interaction sequence S.

1. For $i \in [k]$, let R_i be the range $[100(i-1)k, 100ik]$ and let $S_i = \{u_1^i, \ldots, u_k^i\}$ be k random elements in range R_i. We order the elements of each S_i such that,
$$u_1^1 \leq u_2^1 \leq \ldots \leq u_k^1 < u_1^2 \leq \ldots \leq u_k^k \ .$$
2. Let $j_1, \ldots, j_k$ be random elements in the range $[k]$.
3. Consider the sequence S_{good} described as follows. First we insert S_k in a random order and then we extract j_k values from the heap. We call these deletions the *immediate deletes at stage k*. We then insert S_{k-1} and extract j_{k-1} values. We continue in this way until we insert S_1 and then we extract all the remaining values until the heap is empty. Let S_{bad} be the sequence constructed from S_{good} by choosing $i \in [k]$ at random and swapping the last value extracted in the immediate deletes at stage i with the extraction of the $(j_i + 1)$th smallest element of S_i. Call these values u and v respectively. By construction $u < v$.
4. Let $S = S_{\text{good}}$ with probability $1/2$ and $S = S_{\text{bad}}$ otherwise.

Note that by definition S_{good} is heap-like and, while S_{bad} satisfies (C1) and (C2), it violates (C3). Hence the only way for an algorithm (that only stores and compares (value, time-stamp) pairs along with their extraction times) to recognize if $S = S_{\text{bad}}$ is to either a) have $(u, \cdot)$ stored in memory when $(v, \cdot)$ is extracted or b) have memory of the extraction time of $(v, \cdot)$ when $(u, \cdot)$ is extracted. Unfortunately since i and j_i are chosen at random, unless the algorithm can store $O(k)$ pairs and deletion times then the probability of this is $o(1)$.

6 Conclusions and an Open Question

In this paper we presented a checker and an spot-checker for a priority queue. Both of are very practical and could be used as a guarantee of correct memory behavior when attempting to utilize cheap memory that may be unreliable.

We complemented the checkers with space lower bounds that showed that on-line checking and deterministic checking were infeasible. We also showed that off-line, randomized checkers of a specific class, that included the checker presented, required almost as much space as that required by the checker presented. However, it is conceivable that a better checker may exist that did not belong to this class. We conjecture that this is not the case. Also, a general proof would be very interesting because it appears that the such a proof is not possible with known techniques such as a reducing from communication complexity results. The reason for this is that consecutive subsequences of the interaction sequence can not be generated independently if the interaction sequence is to be heap-like.

References

1. Alon, N., Matias, Y., Szegedy, M.: The space complexity of approximating the frequency moments. Journal of Computer and System Sciences 58(1), 137–147 (1999)
2. Amato, N.M., Loui, M.C.: Checking linked data structures. In: FTCS, pp. 164–173 (1994)
3. Blum, M., Evans, W.S., Gemmell, P., Kannan, S., Naor, M.: Checking the correctness of memories. Algorithmica 12(2/3), 225–244 (1994)
4. Blum, M., Kannan, S.: Designing programs that check their work. J. ACM 42(1), 269–291 (1995)
5. Chakrabarti, A., Cormode, G., McGregor, A.: A near-optimal algorithm for computing the entropy of a stream. In: ACM-SIAM Symposium on Discrete Algorithms (2007)
6. Ergün, F., Kannan, S., Kumar, R., Rubinfeld, R., Viswanathan, M.: Spot-checkers. J. Comput. Syst. Sci. 60(3), 717–751 (2000)
7. Feigenbaum, J., Kannan, S., Strauss, M., Viswanathan, M.: An approximate L^1 difference algorithm for massive data streams. SIAM Journal on Computing 32(1), 131–151 (2002)
8. Feigenbaum, J., Kannan, S., Strauss, M., Viswanathan, M.: Testing and spot-checking of data streams. Algorithmica 34(1), 67–80 (2002)
9. Finocchi, I., Grandoni, F., Italiano, G.F.: Resilient Search Trees. In: ACM-SIAM Symposium on Discrete Algorithms, ACM Press, New York (2007)
10. Goldreich, O., Goldwasser, S., Ron, D.: Property testing and its connection to learning and approximation. J. ACM 45(4), 653–750 (1998)
11. Henzinger, M.R., Raghavan, P., Rajagopalan, S.: Computing on data streams. Technical Report 1998-001, DEC Systems Research Center (1998)
12. Naor, J., Naor, M.: Small-bias probability spaces: Efficient constructions and applications. SIAM J. Comput. 22(4), 838–856 (1993)
13. Kushilevitz, E., Nisan, N.: Communication Complexity. Cambridge University Press, Cambridge (1997)