Optimal Oblivious Priority Queues

Zahra Jafargholi *
Aarhus University

Kasper Green Larsen †
Aarhus University

Mark Simkin ‡
Aarhus University

Abstract

In this work, we present the first asymptotically optimal oblivious priority queue, which matches the lower bound of Jacob, Larsen, and Nielsen (SODA'19). Our construction is conceptually simple, statistically secure, and has small hidden constants. We illustrate the power of our optimal oblivious priority queue by presenting a conceptually equally simple construction of statistically secure offline ORAMs with $O(\log n)$ bandwidth overhead.

1 Introduction

Oblivious RAM (ORAM) is a cryptographic primitive, which allows a client to outsource its private data from an untrusted storage provider while maintaining strong privacy guarantees. The client can perform read and write operations on this outsourced data set without revealing the data, the operation that was performed, or the location that was accessed. ORAM was first introduced by Goldreich and Ostrovsky [Gol87, Ost90, GO96] and has since then be the topic of a long line of research works [OS97, GMOT11, SCSL11, DMN11, SSS12, SvS+13, BN16, LN18, CNS18, PPRY18, AKL+18]. The main measure of efficiency for ORAM schemes is the bandwidth cost per access, meaning how many data blocks the client has to access in order to perform the desired operation and maintain its privacy. The bandwidth costs can be measured in either the worst-case or the amortized sense. The worst-case bandwidth cost is the maximum amount of data blocks that the client has to retrieve from the storage provider in any access from a sequence of accesses. The amortized bandwidth cost is the average amount of data blocks that the client has to retrieve per access. In the following discussion as well as the remainder of the paper, we always assume that n is the number of data elements, m is the client-side memory in bits, w is the server-side block size in bits, and r is the size of a single data element in bits. Furthermore we assume that $m \geq w \geq r$, and unless otherwise stated, we also assume $m, w, r = \Theta(\log n)$.

In their seminal work, Goldreich and Ostrovsky [GO96] proved that any ORAM scheme that behaves in a "balls-and-bins" fashion and is statistically secure has to have an amortized bandwidth cost of $\Omega(\log_{m/r} n)$ data blocks. Their lower bound even holds for so called offline ORAMs, which are given the access pattern of the access sequence that will be executed upfront. Boyle and

*Email: zahra@cs.au.dk. Supported by the European Unions's Horizon 2020 research and innovation program under grant agreement No 669255 (MPCPRO) and No 731583 (SODA).

†Email: larsen@cs.au.dk. Supported by a Villum Young Investigator grant and an AUFF starting grant. Part of the work was done while being a long term visitor at the Simons Institute for Theory of Computing.

‡Email: simkin@cs.au.dk. Supported by the European Unions's Horizon 2020 research and innovation program under grant agreement No 669255 (MPCPRO) and No 731583 (SODA).

Naor [BN16] show that a lower bound proof for offline ORAM[1] without the balls-and-bins restriction would imply a solution to a long standing open problem in circuit complexity. As a way around this barrier, the authors suggest to prove a lower bound for a stronger notion called online ORAM, where the access pattern is not known upfront. Larsen and Nielsen [LN18] prove a lower bound of $\Omega(\log(nr/m))$ blocks on the amortized bandwidth cost of any computationally secure online ORAM scheme without the balls-and-bins restriction.

Apart from their lower bound, Goldreich and Ostrovsky [GO96] also provided the first upper bound by presenting a hierarchical ORAM construction with amortized bandwidth cost of $\mathrm{O}(\log^3 n)$ blocks and computational security. Their hierarchical approach was the basis for many follow-up works [GMOT11, DMN11, GM11, CNS18], which recently led to the first computationally secure ORAMs with $\mathrm{O}(\log n \log\log n)$ bandwidth costs in the offline [MZ14] and the online [PPRY18] setting. An asymptotically optimal online ORAM was presented by Asharov et al. [AKL+18]. The construction of Asharov et al. matches the lower bound of Larsen and Nielsen, but is conceptually quite complex relying on advanced building blocks like multi-array oblivious shuffles, oblivious cuckoo hashing, and expander graphs. The construction does also not provide a matching upper bound to the lower bound of Goldreich and Ostrovsky, since it is only computationally secure. A very different and conceptually simple approach to constructing ORAM was introduced by Stefanov et al. [SCSL11, SvS+13], who present a tree-based statistically secure construction with $\mathrm{O}(\log^2 n)$ worst-case bandwidth cost. The authors of that work also show that their construction has an asymptotically optimal bandwidth *overhead* of $\mathrm{O}(\log n)$ blocks if one assumes large data elements of $r = \Omega(\log^2 n)$ bits, but small server blocks of $w = O(\log n)$ bits.

Oblivious Priority Queues. Several works [Tof11, MZ14, WNL+14, KS14] focus on constructing special-purpose oblivious data structures aiming for concrete efficiency instead of generality. The data structures that we are particularly interested in, in this work, are oblivious priority queues (OPQs). A priority queue stores data elements along with associated priorities in a "sorted order" and supports efficient insertion of new data priority pairs and efficient extraction of the data element with the smallest priority. In the following we refer to these two procedures as Insert and ExtractMin. In addition, some priority queues support a so called DecreaseKey operation, which allows one to decrease the priority of an element that is already stored in the queue. Priority queues are some of the most fundamental and well studied data structures.

Before examining existing OPQs, let us first recall what is known about non-oblivious counterparts. The classic Fibonacci heap [FT87] can support the Insert operation in worst case $O(1)$ time, ExtractMin in amortized $O(\log n)$ time and DecreaseKey in amortized $O(1)$ time. When the priorities are only comparable, this time complexity is optimal because a priority queue can be used to implement comparison-based sorting which has a lower bound of $\Omega(n \log n)$. If we assume priorities are w-bit integers, where w is the machine word size, then a priority queue that uses the reduction to integer sorting [Tho07] can support each operation in deterministic amortized $O(\log\log n)$ time [Han02] or expected amortized $O(\sqrt{\log\log n})$ time [HT02]. A different line of works considers priority queues in the external memory setting. In this setting, we have a small internal memory of size m bits and an external memory consisting of blocks/words of w bits each (similar to the oblivious setting). If we assume that each element inserted in a priority queue can be stored in r bits, then Kumar and Schwabe [KS96] proposed an external memory

[1] To be precise, the authors actually consider a even stronger notion of offline ORAM, where not only the access pattern, but also the operations that will be performed are given upfront.

tournament tree that supports the Insert, Delete, ExtractMin and DecreaseKey operations of a priority queue in amortized $O((r/w)\log(nr/w))$ memory accesses. This was recently improved to $O((r/w)\log(nr/w)/\log\log n)$ [JL19] which is known to be almost optimal due to a lower bound of $\Omega((r/w)\log(w)/\log\log n)$ proved in [ELY17]. It is also known that if DecreaseKeys do not need to be supported, then there exist more efficient priority queues that make an amortized $O((r/w)\log(nr/w)/\log(m/w))$ memory accesses [FJKT99, Arg03]

The first OPQ is due to Toft [Tof11], who presents a construction with a bandwidth cost of $\mathrm{O}(\log^2 n)$ blocks. However, his construction reveals the type of operation that is performed and it does not support DecreaseKey operations, which are used by popular algorithms like Dijkstra's shortest path algorithm. Wang et al. [WNL+14] propose an OPQ with the same bandwidth cost, which does hide the operation that is performed, but does not support DecreaseKey operations. Concurrently, Keller and Scholl [KS14] present a construction similar to the one of Wang et al. that is tailored to secure multiparty computation and supports DecreaseKey operations while also having a bandwidth cost of $\mathrm{O}(\log^2 n)$ blocks. Mitchell and Zimmerman [MZ14] also independently propose an oblivious priority with bandwidth costs $\mathrm{O}(\log^2 n)$. Jacob, Larsen, and Nielsen [JLN19] proved that any OPQ has to have a bandwidth cost of $\Omega((r/w)\log(nr/m))$ blocks, even if it is allowed to return an incorrect result on any ExtractMin operation with constant probability. To date, no matching upper bound was known.

Oblivious Sorting and Shuffling. A different line of works [AKS83, LP98, Bat68, Goo10, Goo14, OGTU14, PPY18] focuses on obliviously sorting and shuffling a remote data array. A sorting algorithm is said to be oblivious if the memory access pattern is independent of the memory contents that it sorts or shuffles. Oblivious sorting algorithms fall in two categories. Deterministic algorithms [AKS83, Bat68, Goo14] that always output a correctly sorted sequence and randomized algorithms [Goo10] that correctly sort a given input with high probability. In a celebrated result due to Ajtai, Komlós, and Szemerédi [AKS83], the authors propose a deterministic asymptotically optimal oblivious sorting algorithm with bandwidth cost $\mathrm{O}(n\log n)$, which when executed in parallel only requires $\mathrm{O}(\log n)$ steps. Unfortunately, the algorithm is merely of theoretical interest, since the hidden constant behind the big-O notation is currently at least 6100 [Pat90]. Goodrich [Goo14] proposes Zig-Zag sort, which is also deterministic and asymptotically optimal, but still has a hidden constant of around 2700 and is not parallelizable. The most popular algorithm in practice is Batcher's bitonic sort [Bat68], which is deterministic and has a bandwidth cost of $\mathrm{O}(n\log^2 n)$ with a hidden constant of approximately 1/2 [PR10]. The first asymptotically optimal randomized oblivious sorting algorithm with better hidden constants is due to Leighton and Plaxton [LP98]. In a subsequent work, Goodrich [Goo10] proposes a simpler oblivious sorting algorithm with even better hidden constants. Lin, Shi, and Xie [LSX19] prove, roughly speaking, that any (randomized) oblivious sorting algorithm that behaves in a balls-and-bins fashion and uses constant client memory has to have a bandwidth cost of $\Omega(n\log n)$.

Any comparison based oblivious sorting algorithm can also be used as an oblivious shuffling algorithm. The first oblivious shuffling algorithm not based on sorting was proposed by Ohrimenko et al. [OGTU14], who present a randomized construction, which requires $\mathrm{O}(\sqrt{n})$ clients-side storage and has a total bandwidth cost of $\mathrm{O}(n)$. The constants in their construction were improved upon by Patel et al. [PPY18], who present a construction that uses 4 times less bandwidth.

1.1 Our Contribution

In this work, we present the *first* statistically-secure construction of OPQs with asymptotically optimal amortized bandwidth costs, which matches the lower bound of Jacob, Larsen, and Nielsen [JLN19] for virtually all parameter ranges. More concretely, we show

Theorem 1 (Informal)**.** *There exists an OPQ with an amortized bandwidth cost of* $\mathrm{O}((r/w)\log(nr/m))$ *blocks for any* $m = \Omega(w + r\log n)$ *and* $w \geq r = \Omega(\log n)$*.*

From a conceptual point of view our construction can be seen as a hybrid between the hierarchical ORAM construction of Goldreich and Ostrovsky [GO96] and the tree-based approach of Stefanov et al. [SCSL11, SvS+13]. Apart from being asymptotically optimal, our construction is conceptually simple, has small hidden constants, and supports DecreaseKey operations. For the natural setting of parameters $r, w = \Theta(\log n)$, our priority queue achieves a bandwidth cost of $O(\log n)$ which is a $\log n$ improvement over previous constructions. Moreover, our priority queue demonstrates that obliviousness comes *for free* for comparison based priority queues!

We illustrate the power of our asymptotically optimal OPQ by using it to construct other popular oblivious primitives. We present a conceptually simple offline ORAM with bandwidth cost $\mathrm{O}(\log n)$ for memory size $m = \mathrm{O}(\log^2 n)$ bits and $r, w = \Theta(\log n)$, matching the $\Omega(\log_{m/r} n) = \Omega(\log n/\log\log n)$ lower bound by Goldreich and Ostrovsky up to a $\log\log n$ factor. That is, we present the first statistically secure construction of offline ORAM, which behaves in a balls-and-bins fashion, with an amortized bandwidth cost of $\mathrm{O}(\log n)$ and a client memory of $\mathrm{O}(\log n)$ blocks.

Theorem 2 (Informal)**.** *There exists a statistically secure offline ORAM with an amortized bandwidth cost of* $\mathrm{O}(\log n)$ *blocks using* $\mathrm{O}(\log n)$ *client memory blocks.*

Lastly, we use our OPQ to construct simple, yet optimal, randomized oblivious sorting and shuffling algorithms. Our constructions, are essentially as efficient as the underlying OPQ and we believe that our sorting and shuffling algorithms could be competitive with existing solutions.

Subsequent Work. After this manuscript appeared online, Shi [Shi19] presented an alternative construction of oblivious priority queues. Her construction is only for the setting of $r = w = \Theta(\log n)$. It achieves an amortized bandwidth cost of $\mathrm{O}(\log n)$ and uses $\mathrm{O}(1)$ client memory blocks. Her solution is thus better for small client memory ($m = o(\log^2 n)$), but it cannot handle blocks of $w = \omega(r)$ bits cheaper than $\mathrm{O}(\log n)$ block accesses per operation. If one can afford a client memory of a couple of megabytes in practice, i.e. $w \gg r$, then we believe our solution performs much better, as the number of times the server needs to be accessed for our solution grows as $\mathrm{O}(r/w) \cdot \log n$. Thus even though the two priority queues transfer the same number of bits, ours transfers data in much larger chunks at a time and avoids the many expensive requests back and forth between client and server. In fact, if $w = \omega(r\log n)$, then our solution makes $o(1)$ accesses to the server amortized, i.e. on most operations, the data needed is already in client memory and incur no network traffic.

2 Oblivious Priority Queue

A priority queue stores elements that consist of a (key,priority)-pair. It supports the following operations:

- Insert(k, p): Insert an element with key k and priority p. If there is already an element present with key k and priority p', then update the priority of that element to $\min\{p, p'\}$.
- DecreaseKey(k, p): Same as Insert(k, p).
- ExtractMin: Return the element with smallest priority among all elements in the priority queue.
- Delete(k): Delete the element with key k.

Throughout the paper, we assume that r is an upper bound on the number of bits needed to store an element. We assume that the priorities are comparable and that the keys are integers from a universe $[U]$ with $U < 2^r$ and $r = \Omega(\log n)$.

2.1 Simple Solution

In this section, we describe a simple version of our optimal priority queue, which supports only the Insert(k, p) and ExtractMin operations. Furthermore, this solution assumes that we never perform an Insert with a key k that is already present in the priority queue. For simplicity, we also assume that the server block size w satisfies $w = \Theta(r) = \Theta(\log n)$. The solution will also use $\Theta(n)$ space where n is the number of operations performed on the priority queue, not the maximum number of elements stored in it. We show how to remove all these assumption in Section 2.2.

Our priority queue is a perfect binary tree $\mathcal{T}$ with initial height 1, i.e. it consists of a single root node. We say that the depth of the root is 0. After sufficiently many operations on the priority queue, we append another layer to $\mathcal{T}$ (by allocating more storage). We will describe later precisely how often we append a new layer.

Each node of $\mathcal{T}$ has a buffer for storing elements. The buffer can store up to B elements. We require $B \geq c \ln n$ where n is an upper bound on the number of operations to be performed on the priority queue and $c > 1$ is a sufficiently large constant. We will furthermore assume that B is a multiple of 4. This can always be ensured by a constant factor increase in c.

The high level idea in our solution is that the buffers will move elements up towards the root as they get closer to being the smallest elements in the priority queue. Newly inserted elements start at the root's buffer and initially move down the tree until they reach a level where their priority is small compared to the other elements. They will then start ascending towards the root again.

We will argue that most of the time, we only need to work on data in the root: Elements that need to be returned by ExtractMin will have moved up through the buffers, and thus are stored in the root when needed. Similarly, when we insert new elements, we simply insert them in the root's buffer. Of course the buffers in the tree might become either full or empty. Therefore, we occasionally visit all the top i levels of $\mathcal{T}$ and move elements up and down to avoid overflowing or underflowing buffers. To avoid overflowing buffers, we need that the elements are distributed evenly down the tree. This is done using randomness: when an element (k, p) is inserted into the priority queue, we generate a uniform random bit string $h(k, p)$ in $\{0, 1\}^{\log n}$ and store it together with the element. The element (k, p) will be stored somewhere along the root-to-leaf path obtained as follows: Start at the root and descend down the tree as follows: When at an internal node v of depth i, if the i'th bit of $h(k, p)$ is 0, descend into v's left child, otherwise descend into its right child. The element will always be stored in a buffer along this path.

We will always keep the root in client memory, while other nodes of $\mathcal{T}$ will be stored at the server. We handle the two procedures as follows:

Insert(k, p): Sample a random $h(k, p) \in \{0, 1\}^{\log n}$ and store it together with the element (k, p) in the root's buffer.

ExtractMin: Return and remove the element (k, p) with smallest p among all elements in the root's buffer.

In addition to the above very simple procedures for handling Insert and ExtractMin, we need a procedure for clearing and filling the buffers, as otherwise the root buffer would quickly overflow or underflow. This is handled via two procedures PushDown(i) and PullUp(i). These procedures are invoked as follows: Maintain a counter C that keeps track of the number of operations performed on the priority queue. After every operation, we increment C and check whether C is a multiple of $B/4$. If it is, we compute the largest integer i such that C is also a multiple of $2^i B/4$. We then invoke PushDown(0), PushDown(1), ..., PushDown(i), PullUp(i),PullUp($i-1$),...,PullUp(0) in that order. We now describe the two procedures:

PushDown(i): If this is the first time PushDown(i) is called, we start by adding a new layer to $\mathcal{T}$ at depth $i+1$ (by allocating more storage). We now examine the nodes at depth i one by one, from left to right. When examining a node v, we load v and its two children into client memory. We take all elements in v's buffer and distribute them to the buffers of v's children. Each element (k, p) is placed in the child consistent with $h(k, p)$.

PullUp(i): This operation assumes all buffers at depth i are empty. We examine the nodes at depth i one by one, from left to right. When examining a node v, we load v and its two children into client memory. We take the $B/2$ elements with smallest priority amongst all elements in both childrens' buffers and put them in v's buffer. We leave the remaining where they are. If there are less than $B/2$ elements in total in all the buffers of v's children, we simply move all of them to v's buffer.

Observe that the memory access pattern of the priority queue is completely fixed and depends only on the number of operations performed. Hence the priority queue is oblivious. The only thing that could go wrong, is that the priority queue returns an incorrect element or that a buffer needs to store more than B elements. In the latter case, we will simply delete the extra elements in a buffer. This guarantees that the access pattern remains the same, but of course means that the priority queue will return incorrect results. The remainder of this section proves that this happens very rarely and that the priority queue makes few memory accesses:

Theorem 3. *The simple priority queue processes a sequence of n operations with an amortized bandwidth cost of $O(\log n)$ blocks and is correct with probability at least $1 - n^2/e^{B/12}$. It uses $m = O(B \log n)$ bits of client memory and assumes that $r, w = \Theta(\log n)$.*

To prove correctness of the above priority queue, we need to show two things: 1) The element with smallest priority is always in the root's buffer and 2) No buffer ever stores more than B elements. The first property ensures that ExtractMin always returns the correct element and the second property simply ensures that there is enough space in the buffers to hold the elements to be stored there.

We prove that the priority queue satisfies both 1) and 2) as long as the randomness $h(k, p)$ distributes the elements sufficiently evenly down the tree. We define this formally in the following:

Definition 1. *Let $\mathcal{S} := \mathrm{op}_1, \ldots, \mathrm{op}_N$ be a sequence of operations on a priority queue. For each $t = 1, \ldots, N$, define $\mathcal{L}(\mathcal{S}, t)$ as the set of elements (k, p) such that the key k has priority p after operations $\mathrm{op}_1, \ldots, \mathrm{op}_t$. For notational convenience, we define $\mathcal{L}(\mathcal{S}, t) = \emptyset$ for $t \leq 0$.*

The set $\mathcal{L}(\mathcal{S}, t)$ thus denotes the set of elements that should be present in the priority queue after operations $\mathrm{op}_1, \ldots, \mathrm{op}_t$ (on a correct execution). We also need a set describing the elements involved in a subsequence of operations:

Definition 2. *Let $\mathcal{S} := \mathrm{op}_1, \ldots, \mathrm{op}_N$ be a sequence of operations and let $\mathrm{op}_i, \ldots, \mathrm{op}_j$ be a subsequence. Define the set $\mathcal{K}\int(\mathcal{S}, i, j)$ containing all elements (k, p) such that at least one of the operations $\mathrm{op}_i, \ldots, \mathrm{op}_j$ is either an Insert(k, p) or an ExtractMin where the correct answer is (k, p). For notational convenience, we define $\mathcal{K}\int(\mathcal{S}, i, j) = \mathcal{K}\int(\mathcal{S}, 1, j)$ if $i \leq 0$.*

With the above two definitions, we can define what we mean by distributing keys evenly:

Definition 3. *For a node $u \in \mathcal{T}$ of depth d and a sequence of operations $\mathcal{S} := \mathrm{op}_1, \ldots, \mathrm{op}_N$, we say for every $t = 1, \ldots, N$, that u is* s-overloaded *at op_t if there are $B/2$ or more elements (k, p), such that $(k, p) \in \mathcal{K}\int(t - 2^d B/4 + 1, t)$ and $h(k, p)$ describes a path into u's subtree.*

Here we used the s- in **s**-overloaded just to distinguish the definition of overloaded from the one we give in the full solution (Section 2.2). s- should be thought of as refering to the **s**imple solution.

We prove that as long as no node becomes s-overloaded, then the simple priority queue is correct:

Lemma 1. *For a sequence $\mathcal{S}$ of up to n operations on the priority queue, if there is no node u that is s-overloaded after op_t for any t, then the priority queue returns the correct element on every ExtractMin and never stores more than B elements in a buffer.*

Proof. We prove the lemma by induction on the operations. Let $\mathcal{S} = \mathrm{op}_1, \ldots, \mathrm{op}_N$ for an $N \leq n$. The lemma clearly holds prior to the first operation being performed. Assume that there are no nodes that are s-overloaded after op_i for any i. Assume also that the priority queue returns the correct element on every ExtractMin during $\mathrm{op}_1, \ldots, \mathrm{op}_{t-1}$ and that no buffer stores more than B elements during $\mathrm{op}_1, \ldots, \mathrm{op}_{t-1}$. We prove that the same holds for op_t.

We start by proving that op_t returns the correct element if it is an ExtractMin. By induction, the priority queue contained exactly the elements $\mathcal{L}(\mathcal{S}, t-1)$ after op_{t-1}. Hence the smallest element in $\mathcal{T}$ is the correct smallest element to return. We thus need to argue that the smallest element (k, p) is stored in the root when op_t begins (as this causes our ExtractMin procedure to return the correct result). To see this, assume for the sake of contradiction that (k, p) is stored in some node u of depth $d \geq 1$ when op_{t-1} ends. Let $t' < t$ be the last time a PullUp$(d-1)$ operation was performed. We must have $t' \geq t - 2^{d-1}B/4$. It must have been the case that when PullUp$(d-1)$ visited u's parent v during $\mathrm{op}_{t'}$, the element (k, p) remained in u. This must hold since all operations after that PullUp$(d-1)$ and until the end of op_{t-1} access only nodes of depth at most $d-1$. Since PullUp$(d-1)$ failed to move (k, p) to v, there must have been at least $B/2$ other elements (k', p') with $p' < p$ in v's subtree. Since the priority queue was correct after $\mathrm{op}_{t'}$ (by induction), all these elements must be in $\mathcal{L}(\mathcal{S}, t')$. Since (k, p) is the smallest element after op_{t-1} and the priority queue was correct up to op_{t-1}, these elements must all be extracted by ExtractMins during $\mathrm{op}_{t'+1}, \ldots, \mathrm{op}_{t-1}$ which puts the elements in $\mathcal{K}\int(\mathcal{S}, t'+1, t) \subseteq \mathcal{K}\int(\mathcal{S}, t - 2^{d-1}B/4 + 1, t)$. Moreover, $\mathcal{K}\int(\mathcal{S}, t - 2^{d-1}B/4 + 1, t)$ also contains (k, p) since (k, p) is the correct result of the ExtractMin

during op_t. This means that there are more than $B/2$ elements (k', p') in $\mathcal{K}\!\int(\mathcal{S}, t - 2^{d-1}B/4 + 1, t)$ that all have $h(k', p')$ describing a path into v's subtree. This contradicts that v is not s-overloaded at op_t.

We now prove that no buffer stores more than B elements. Observe that a buffer can only overflow either during an Insert at the root's buffer, or when a PushDown visits the node's parent. For the root node, notice that we do a PushDown(0) every $B/4$ operations. This empties the root's buffer. The following PullUp(0) brings at most $B/2$ elements into the root's buffer. Hence the root never stores more than $B/4+B/2 < B$ elements. Now let u be a node of depth $d \geq 1$ and assume its buffer stores more than B elements during a PushDown$(d-1)$ in some operation op_t. Let $t' < t$ be the last time a PushDown(d) was performed prior to op_t (or let $t' = 0$ if no such PushDown(d) has been performed yet). We have $t' \geq t - 2^d B/4$. When the PushDown operations during $\mathrm{op}_{t'}$ ended, all buffers in levels $0, \ldots, d$ were empty. The following PullUp(d) operation brings at most $B/2$ elements into u's buffer and these are the only elements where h describes a path into u's subtree that are stored in depth $0, \ldots, d$ at the end of $\mathrm{op}_{t'}$. Hence for u's buffer to overflow during op_t, there must be more than $B/2$ new elements inserted during $\mathrm{op}_{t'+1}, \ldots, \mathrm{op}_t$ that all have h describing a path into u's subtree. All these elements are in $\mathcal{K}\!\int(\mathcal{S}, t' + 1, t) \subseteq \mathcal{K}\!\int(\mathcal{S}, t - 2^d B/4 + 1, t)$. This contradicts that the number of elements (k, p) from $\mathcal{K}\!\int(\mathcal{S}, t - 2^d B/4 + 1, t)$ with $h(k, p)$ describing a path into u's subtree is bounded by $B/2$ when u is not s-overloaded. □

Thus all we need to show is that there is a small probability that a node is s-overloaded after op_t:

Lemma 2. *For a sequence $\mathcal{S}$ of up to n operations on the priority queue, it holds with probability at least $1 - n^2/e^{B/12}$ that there are no nodes u that are s-overloaded at op_t for any t.*

Proof. Define the event $E_{u,t}$ that happens if node u is s-overloaded at op_t. Let d be the depth of u. Since all elements (k, p) in $\mathcal{K}\!\int(\mathcal{S}, t - 2^d B/4 + 1, t)$ are assigned uniform random and independently chosen bit strings $h(k, p)$, it follows that $h(k, p)$ describes a path into u's subtree with probability 2^{-d}. Moreover, there are only $2^d B/4$ elements involved in the operations $\mathrm{op}_{t-2^d B/4+1}, \ldots, \mathrm{op}_t$, thus $|\mathcal{K}\!\int(\mathcal{S}, t - 2^d B/4 + 1, t)| \leq 2^d B/4$. The expected number of elements that are distributed to u's subtree is thus $\mu \leq B/4$. By a Chernoff bound ($\Pr[X \geq (1+\delta)\mu] \leq e^{-\delta\mu/3}$ for $\delta \geq 1$), we get that the probability that there are at least $B/2$ elements that are distributed to u's subtree is no more than $e^{-(B/(2\mu)-1)\mu/3} = e^{-(B/6-\mu/3)} \leq e^{-B/12}$. A union bound over all nodes u and all operations op_t concludes the proof. □

What remains is to analyse the number of server blocks accessed while processing a sequence of $N \leq n$ operations. Since the root is stored in client memory, the operations on it cause no accesses of server blocks. For a node u of depth $d \geq 1$, we read its B elements into client memory when we access the node. It is accessed during PushDown$(d-1)$, PullUp$(d-1)$, PushDown(d) and PullUp(d), hence u causes us to access $O(B)$ server blocks for every $O(2^d B)$ operations. There are 2^d nodes at depth d. Summing over all nodes (max depth is $\log n$) gives:

$$\sum_{i=1}^{\log n} (N/2^d B) \cdot 2^d \cdot O(B) = O(N \log n).$$

Thus the amortized cost is $O(\log n)$.

We have thus proved Theorem 3.

2.2 Full Solution

In this section, we extend the simple solution from Section 2.1 such that it supports all four priority queue operations Insert(k, p), ExtractMin, DecreaseKey(k, p) and Delete(k). Moreover, we generalize it such that it works for any $w \geq r = \Omega(\log n)$, and such that the space usage is $O(nr)$ bits where n is the maximum number of elements to be simultaneously stored in the priority queue. This is linear space since each element takes r bits to store. We show that the amortized running time per operation is $O(\log n)$ and that the amortized bandwidth cost is $O((r/w)\log(nr/m))$ blocks. These bounds hold if the number of operations performed on the priority queue is $N \leq \text{poly}(n)$.

As in the simple solution, our priority queue consists of a perfect binary tree $\mathcal{T}$, where nodes store buffers that can hold up to B elements. Here we set $B := cm/r$ for any constant $c > 0$ small enough that a constant number of buffers fit in client memory (which is enough to support the operations below). We assume B is a multiple of 8. This can always be ensured by constant factor changes in c. The height of $\mathcal{T}$ is set to $\lceil \log(16n/B) \rceil$ and thus the total number of elements that can be stored in the buffers of $\mathcal{T}$ is $O(2^{\log(16n/B)}B) = O(n)$. Hence $\mathcal{T}$ uses linear space. We always keep the root of $\mathcal{T}$ in client memory.

Each internal node of $\mathcal{T}$ has two buffers, an *element buffer* and a *delete buffer*. The element buffers store elements (k, p) and the delete buffer stores just keys k. The intuition to have in mind is that an element (k, p) stored in the element list of a node u should be deleted if the key k occurs in a delete buffer of a proper ancestor of u. All buffers can hold up to B elements/keys. The leaves of $\mathcal{T}$ only have an element buffer. We maintain the invariant that any element buffer stores at most one element with any given key k, and there are no duplicates in the delete buffers.

As in the simple solution, we will distribute elements down the tree based on randomly chosen values. However, in this full solution, we need to handle multiple insertions of the same key, DecreaseKeys and Delete operations. This means that multiple occurrences of the same element should "meet" in $\mathcal{T}$ such that the one with minimum priority can "remove" the other occurrences. For this reason, we distribute the elements based on a truly random hash of their keys $h : \{0,1\}^r \to \{0,1\}^{\lceil \log(16n/B) \rceil - 1}$. That is, each element (k, p) in an element buffer, and any key k in a delete buffer, will be stored somewhere along the root-to-leaf path specified by $h(k)$ as follows: The path starts at the root, and at the i'th level, it descends into the left child if the i'th bit of $h(k)$ is 0 and into the right child otherwise. We will argue later that h need not be truly random, but only a q-wise independent hash function for a $q \leq B/2 = O(m/r)$. Such hash functions can be stored in $O(q(r + \log n)) = O(qr)$ bits and will fit in the client memory (using e.g. the standard polynomial hash function $h(x) = (((\sum_{i=0}^{q} \alpha_q x^q) \mod p) \mod 2^{\lceil \log(16n/B) \rceil - 1})$ for uniform random coefficients $\alpha_i \in \{0, \ldots, p-1\}$, where $p > 2^{\lceil \log(16n/B) \rceil - 1}$ is a prime). Thus elements with the same key are on the same root-to-leaf path.

In the description of our priority queue, we assume all priorities are distinct. This can be ensured e.g. by breaking ties via a comparison on keys. Moreover, our priority queue implementation will never need to break ties on identical copies of the same element (k, p) as we always remove one of the identical copies if they "meet" in $\mathcal{T}$.

We handle operations as follows:

Insert(k, p): Call DecreaseKey(k, p).

DecreaseKey(k, p): Check if the root's element buffer already has an element (k, p') with key k. If so, update p' to $\min\{p, p'\}$ and return. Otherwise, compute $h(k)$ and store it together with the

element (k, p) in the root's element buffer.

Delete(k): Remove any elements (k, p) with key k from the root's element buffer. If the root's delete buffer does not already have the key k, then add k to the delete buffer.

ExtractMin: Return and remove the element (k, p) with smallest p among all elements in the root's element buffer. We then add the key k to the root's delete buffer if it is not already there. If there is no element in the root, then return "Empty".

As in the simple solution, we also have PushDown and PullUp operations for avoiding buffer underflows and overflows. We use these as follows: When the priority queue is initialized, we set a counter C to 0. We increment C after every Insert, DecreaseKey, Delete and ExtractMin operation on the priority queue. If C becomes an integer multiple of $B/8$, we compute the largest integer i such that C is a multiple of $2^i B/8$. We then let $i^* = \min\{i, \lceil \log(16n/B) \rceil - 1\}$ and execute PushDown(0),...,PushDown(i^*),PullUp(i^*),...,PullUp(0) in that order. We implement PushDown and PullUp as follows:

PushDown(i): Examine the nodes at depth i one by one, from left to right. When examining a node v, we load v and its two children into client memory. We first take all keys in v's delete buffer and distribute them to the delete buffers of v's children. A key k is distributed to the child consistent with $h(k)$. When inserting a key k into a child w's delete buffer, we remove any element (k, p) from w's element buffer. If the key k is already stored in w's delete buffer, we do not add an additional copy. Next, we distribute all elements from v's element buffer to the element buffers of v's children. Each element (k, p) is placed in the child consistent with $h(k)$. When adding the element (k, p) to the element buffer of a child w, we first check whether there is already an element (k, p') stored in w's element buffer. If so, we update p' to $\min\{p, p'\}$ instead of inserting (k, p) in the buffer.

If the children of v are leaves, we do not place the keys from v's delete buffer into the children's delete buffers (they don't have any). Thus we simply remove each such key k from the priority queue after having removed elements (k, p) from the child's element buffer.

PullUp(i): This operation assumes that all buffers at level i are empty. We examine the nodes at depth i one by one, from left to right. When examining a node v, we load v and its two children into client memory. We take the $B/2$ elements with smallest priority amongst all elements in the element buffers of v's children and put them into v's element buffer. We leave the remaining where they are. If there are less than $B/2$ elements in total in the childrens' element buffers, we simply move all of them to v's element buffer.

The above priority queue has a deterministic server memory access pattern that depends only on the number of operations N performed and the maximum number of elements n that can be stored in the priority queue. Hence it is oblivious. If a buffer ever stores more than B elements, we simply remove the extra elements to ensure that the memory access pattern remains the same. This of course causes the priority queue to return incorrect results. We will show in the remainder of this section that the priority queue has the following guarantees:

Theorem 4. *For any $q \leq B/2 = O(m/r)$, we can implement the priority queue such that it processes a sequence of N operations with amortized bandwidth cost $O((r/w)\log(nr/m))$ blocks, has an amortized running time of $O(q + \log n)$ and is correct with probability at least $1 - Nn/e^{q/4}$. It works for any number of client bits of memory $m \geq w$ satisfying $m = \Omega(r \log n)$.*

2.2.1 Correctness

We will prove that the above implementation of the priority queue is correct, provided that the hash function h distributes the keys *evenly enough* down the tree $\mathcal{T}$. We will then argue that this happens with good probability. Putting it all together, we obtain the following result:

Lemma 3. *For a sequence of operations $\mathcal{S} := \mathrm{op}_1, \ldots, \mathrm{op}_N$, if we use a q-wise independent hash function h for any $q \leq B/2$, then it holds with probability at least $1 - Nn/e^{q/4}$ that the priority queue answers every ExtractMin in $\mathcal{S}$ correctly and no buffer ever stores more than B elements.*

To prove Lemma 3, we need to argue that the delayed removal of elements (k, p) and duplicate occurrences of a key k does not lead to problems for our priority queue. For this, we start by introducing some terminology. Consider a sequence of operations $\mathcal{S} := \mathrm{op}_1, \ldots, \mathrm{op}_n$. Let (k, p) be an element stored in an element buffer of a node u in the priority queue. We say that there is a *pending delete* for (k, p) if the key k is stored in a delete buffer of a proper ancestor of u (proper meaning not including u itself). We will prove the following:

Lemma 4. *Let $\mathcal{S} := \mathrm{op}_1, \ldots, \mathrm{op}_t$ be a sequence of operations on a priority queue and assume the priority queue returns the correct smallest element on every ExtractMin during $\mathrm{op}_1, \ldots, \mathrm{op}_t$ and that no buffer stores more than B elements during $\mathrm{op}_1, \ldots, \mathrm{op}_t$. Then after operation op_t, it holds that:*

1. *For all elements $(k, p) \in \mathcal{L}(\mathcal{S}, t)$ we have* all *of the following:*
   (a) *(k, p) is stored in the priority queue.*
   (b) *There is no pending delete for (k, p).*
   (c) *There is no element (k, p') with $p' < p$ that is stored in the priority queue without a pending delete for it.*
2. *For all elements $(k, p) \notin \mathcal{L}(\mathcal{S}, t)$, we have* at least one *of the following:*
   (a) *(k, p) is not stored in the priority queue.*
   (b) *There is a pending delete for (k, p).*
   (c) *There is an element $(k, p') \in \mathcal{L}(\mathcal{S}, t)$ with $p' < p$.*

Proof. We prove the lemma by induction on t. The lemma clearly holds before the first operation. So consider some operation op_t and assume the lemma is true after every operation $\mathrm{op}_1, \ldots, \mathrm{op}_{t-1}$.

First let $(k, p) \in \mathcal{L}(\mathcal{S}, t)$. We show that 1.a-c holds for (k, p) after op_t. Let op_i with $i \leq t$ be the operation where $(k, p) \in \mathcal{L}(\mathcal{S}, j)$ for all $i \leq j \leq t$ and $(k, p) \notin \mathcal{L}(\mathcal{S}, i-1)$. By definition of $\mathcal{L}(\mathcal{S}, i)$, the operation op_i is either Insert(k, p) or DecreaseKey(k, p) and there is no (k, p') with $p' < p$ in $\mathcal{L}(\mathcal{S}, i-1)$. Moreover, no Insert$(k, p')$ or DecreaseKey(k, p') with $p' < p$ is executed during $\mathrm{op}_{i+1}, \ldots, \mathrm{op}_t$ (as otherwise there would be a j with $i < j \leq t$ such that $(k, p) \notin \mathcal{L}(\mathcal{S}, j)$) and no Delete$(k)$ is executed during $\mathrm{op}_{i+1}, \ldots, \mathrm{op}_t$ (again, this would give an index j with $i < j \leq t$ for

which $(k,p) \notin \mathcal{L}(\mathcal{S}, j)$). During op_i, we insert (k,p) in the root's element list. Any delete k already stored in the priority queue cannot remove (k,p) during operations $\text{op}_i, \ldots, \text{op}_t$ and cannot become a pending delete for (k,p) (an element never "overtakes" a delete in our PushDown procedure). Any (k,p') with $p' < p$ that was stored in the priority queue after op_{i-1} must have had a pending delete by induction (since $(k,p'') \notin \mathcal{L}(\mathcal{S}, i-1)$ for any $p'' < p$ and $p' < p$, such a (k,p') could not have satisfied 2.c after op_{i-1}, and if (k,p') was stored in the priority queue, it also did not satisfy 2.a. Hence it must have satisfied 2.b). Thus such an element (k,p') cannot cause (k,p) to be removed (the PushDown procedure will remove the element due to the k in a delete buffer, before it can cause (k,p) to be removed). Since the priority queue returns the correct smallest element on every ExtractMin during $\text{op}_i, \ldots, \text{op}_t$ by assumption, we conclude that no element (k,p') is returned. Since only Delete and ExtractMin can cause the key k to enter the root's delete buffer, we conclude that (k,p) is still stored in the priority queue after op_t and there is no pending delete for it, i.e. 1.a and 1.b holds. For 1.c, we already argued that all (k,p') with $p' < p$ that were present at op_i had a pending delete. Furthermore, no (k,p') with $p' < p$ is inserted after op_i, thus 1.c also holds.

Next, let $(k,p) \notin \mathcal{L}(\mathcal{S}, t)$. We assume (k,p) is stored in the priority queue after op_t and prove that this implies that either 2.b or 2.c is satisfied. Since (k,p) is in the priority queue, there must be some (last) operation op_i with $i \leq t$ that was either an Insert(k,p) or DecreaseKey(k,p) which caused (k,p) to be inserted into the root's element list. We split the proof in two cases:

- If $(k,p) \in \mathcal{L}(\mathcal{S}, i)$, then since $(k,p) \notin \mathcal{L}(\mathcal{S}, t)$, (k,p) must either 1) be the correct smallest element to return on an ExtractMin during $\text{op}_{i+1}, \ldots, \text{op}_t$, 2) there is be a Delete(k) during $\text{op}_{i+1}, \ldots, \text{op}_t$ or 3) there is an Insert(k,p') or DecreaseKey(k,p') with $p' < p$ during $\text{op}_{i+1}, \ldots, \text{op}_t$. Since the priority queue returns the correct smallest element on every operation, we conclude that either 2.b or 2.c holds after op_t.

- If $(k,p) \notin \mathcal{L}(\mathcal{S}, i)$, then there must have been another element $(k,p') \in \mathcal{L}(\mathcal{S}, i)$ with $p' < p$. There are three cases: Either 1) there is an ExtractMin during $\text{op}_{i+1}, \ldots, \text{op}_t$ which returns an element (k,p'') with $p'' \leq p' < p$, 2) there is a Delete(k) during $\text{op}_{i+1}, \ldots, \text{op}_t$, or 3), $(k,p'') \in \mathcal{L}(\mathcal{S}, t)$ for some $p'' \leq p' < p$. In the first case, a delete k is inserted in the root's delete buffer during the ExtractMin and becomes a pending delete for (k,p), i.e. 2.b holds. In the second case, k is inserted into the root's delete buffer during the Delete(k) operation and 2.b holds. In the third case, 2.c holds.

□

We are now ready to prove that when keys are distributed evenly, the priority queue behaves as intended. For this, we formally define what we mean by distributing keys evenly. We re-use the definition of $\mathcal{L}(\mathcal{S}, t)$ from Section 2.1 (Definition 1) that gives the set of elements (k,p) that should be stored in the priority queue after operations $\text{op}_1, \ldots, \text{op}_t$ in the sequence of operations $\mathcal{S} := \text{op}_1, \ldots, \text{op}_N$. We also need the following definition:

Definition 4. *Let $\mathcal{S} := \text{op}_1, \ldots, \text{op}_N$ be a sequence of operations and let $\text{op}_i, \ldots, \text{op}_j$ be a subsequence. Define the set $\mathcal{K}(\mathcal{S}, i, j)$ containing all keys k such that at least one of the operations $\text{op}_i, \ldots, \text{op}_j$ is either an Insert(k,p), DecreaseKey(k,p), Delete(k) or an ExtractMin where the correct answer is an element (k,p). For notational convenience, we define $\mathcal{K}(\mathcal{S}, i, j) = \mathcal{K}(\mathcal{S}, 1, j)$ if $i \leq 0$.*

With the above definition, we can define what we mean by distributing keys evenly:

Definition 5. *For a node $u \in \mathcal{T}$ of depth d and a sequence of operations $\mathcal{S} := \mathrm{op}_1, \ldots, \mathrm{op}_N$, we say for every $t = 1, \ldots, N$, that u is* overloaded *at op_t if:*

- *u is an internal node and there are $B/2$ or more keys k such that k is in $\mathcal{K}(t - 2^d B/8 + 1, t)$ and $h(k)$ describes a path into u's subtree.*
- *u is a leaf and there are $B/2$ or more keys k such that k is in $\mathcal{K}(t-2^d B/8+1, t) \cup \mathcal{L}(\mathcal{S}, t-2^d B/8)$ and $h(k)$ describes a path into u's subtree.*

We will show that our priority queue is correct if no node ever becomes overloaded:

Lemma 5. *Let $\mathcal{S} := \mathrm{op}_1, \ldots, \mathrm{op}_N$ be a sequence of operations on a priority queue and assume $|\mathcal{L}(\mathcal{S}, t)| \leq n$ for all t. If it holds for all operations op_t that no node u is overloaded at op_t, then the priority queue returns the correct smallest element on every ExtractMin and no buffer ever stores more than B elements.*

Proof. We prove the lemma by induction on t. The lemma clearly holds before the first operation. So consider some operation op_t and assume the lemma is true after every operation $\mathrm{op}_1, \ldots, \mathrm{op}_{t-1}$. The properties in Lemma 4 hold after op_{t-1} by induction.

First we prove that if op_t is an ExtractMin, then it returns the correct smallest element. Observe that the correct element to return, is the element (k, p) with smallest p among all elements $(k', p') \in \mathcal{L}(\mathcal{S}, t-1)$. We need to show that (k, p) is stored in the root's element buffer after op_{t-1} and that no element (k', p') with $p' < p$ is stored in the root's element buffer. We start with the second part as it is easiest to prove. Assume for the sake of contradiction that an element (k', p') with $p' < p$ was stored in the root's element buffer. By definition, there is no pending delete for (k', p') (a pending delete requires the key k' to be stored in a proper ancestor's delete buffer). Moreover, $(k', p') \notin \mathcal{L}(\mathcal{S}, t-1)$ since (k, p) was the element with smallest priority in $\mathcal{L}(\mathcal{S}, t-1)$. Hence by Lemma 4, (k', p') must satisfy 2.c, i.e. it must be the case that $(k', p'') \in \mathcal{L}(\mathcal{S}, t-1)$ for some $p'' < p'$. This contradicts that (k, p) has the smallest priority among elements in $\mathcal{L}(\mathcal{S}, t-1)$. Thus such an element (k', p') cannot exist. We now turn to proving that (k, p) is stored in the root's element buffer after op_{t-1}. First, Lemma 4 1.a gives us that (k, p) is stored somewhere in the priority queue. So assume for the sake of contradiction that it is stored in a node u of depth $d \geq 1$ after op_{t-1}. Let $t' \leq t-1$ be the last time a PullUp$(d-1)$ was performed. We have $t' > t - 1 - 2^{d-1}B/8 \Rightarrow t' \geq t - 2^{d-1}B/8$. It must have been the case that (k, p) remained in u's element buffer when PullUp$(d-1)$ visited u's parent v. This has to be the case as all operations performed after the PullUp$(d-1)$ and until the end of op_{t-1} only visit nodes of depth $< d$. Since (k, p) remained in u, there must have been at least $B/2$ elements (k', p') in v's subtree that have $p' < p$. These must all have distinct keys since we never have two elements with the same key in a node's element buffer. Notice that for all these element (k', p'), there was no k' in a delete buffer in v or an ancestor of v since all delete buffers in levels $0, \ldots, d-1$ were empty due to the preceding PushDowns executed during $\mathrm{op}_{t'}$. Hence there was no pending delete for any such (k', p') at the end of $\mathrm{op}_{t'}$. By Lemma 4, it must hold for each such (k', p') that either $(k', p') \in \mathcal{L}(\mathcal{S}, t')$ or 2.c gives us that $(k', p'') \in \mathcal{L}(\mathcal{S}, t')$ for some $p'' < p'$. We thus conclude that we have at least $B/2$ distinct keys k' such that $(k', p') \in \mathcal{L}(\mathcal{S}, t')$ and $p' < p$. But (k, p) is the element with smallest priority in $\mathcal{L}(\mathcal{S}, t-1)$, hence each such (k', p') must either 1) be the result of an ExtractMin during $\mathrm{op}_{t'+1}, \ldots, \mathrm{op}_{t-1}$, 2) there is a Delete$(k')$ during $\mathrm{op}_{t'+1}, \ldots, \mathrm{op}_{t-1}$, or 3), there is an Insert(k', p'') or DecreaseKey(k', p'') with a $p'' < p'$ during $\mathrm{op}_{t'+1}, \ldots, \mathrm{op}_{t-1}$. Any of these conditions puts k' in

$\mathcal{K}(\mathcal{S}, t-2^{d-1}B/8, t-1)$ by definition. Thus $|\mathcal{K}(\mathcal{S}, t-2^{d-1}B/8, t-1)| \geq B/2$ which contradicts that v is not overloaded at op_{t-1}.

Next we prove that no buffer stores more than B elements during op_t. For the root, notice that its buffers are both emptied after every PushDown(0) and the following PullUp(0) moves at most $B/2$ elements into the root's element buffer. Since PushDown(0) is executed every $B/8$ operations, the root's buffers never store more than $B/2 + B/8 < B$ elements. For an internal node u of depth d with $1 \leq d < \lceil \log(16n/B) \rceil$, observe that u's element buffer can overflow only during a PushDown$(d-1)$ when visiting u's parent v. Let $t' < t$ be the last time a PushDown(d) was performed (let $t' = 0$ if no such PushDown has been performed yet) prior to op_t. We have $t' \geq t - 2^d B/8$. After the PushDown operations during $\mathrm{op}_{t'}$, it held that both of u's buffers were empty. The following PullUp operations during $\mathrm{op}_{t'}$ moves at most $B/2$ elements (k,p), with $h(k)$ describing a path into u's subtree, into buffers of levels $0, \ldots, d$. Thus for u's element buffer to overflow, there must be at least $B/2$ new elements (k,p) inserted during $\mathrm{op}_{t'+1}, \ldots, \mathrm{op}_t$ such that $h(k)$ describes a path into u's subtree ($B/2$ elements with distinct keys as we never have two elements with the same key in a node's element buffer). This puts the key k of all these elements into $\mathcal{K}(\mathcal{S}, t'+1, t) \subseteq \mathcal{K}(\mathcal{S}, t-2^d B/8+1, t)$, contradicting that u is not overloaded at op_t. For u's delete buffer to overflow, we similarly observe that there must have been more than B distinct keys k that enter the root's delete buffer during $\mathrm{op}_{t'+1}, \ldots, \mathrm{op}_t$ and have $h(k)$ describing a path into u's subtree. By induction, all ExtractMins were answered correctly, thus any key k entering the root's delete buffer must correspond either to a Delete(k) or an ExtractMin returning a correct element (k,p). Both put k in $\mathcal{K}(\mathcal{S}, t'+1, t) \subseteq \mathcal{K}(\mathcal{S}, t-2^d B/8+1, t)$ contradicting that u is not overloaded at op_t. Finally, for a leaf node u of depth $d = \lceil \log(16n/B) \rceil$, let $t' \geq t - 2^{d-1}B/8$ be the last time prior to op_t that a PushDown$(d-1)$ was performed. After the PushDown operations during $\mathrm{op}_{t'}$, all buffers above the leaves were empty, hence there were no keys in any delete buffers (the leaves have no delete buffers). Since only the element buffers in the leaves had elements, it follows by induction and Lemma 4 that the set of elements (k,p) in the priority queue was precisely the set of elements in $\mathcal{L}(\mathcal{S}, t')$ (we have no two elements with the same key since we always remove one of the elements if they meet in a buffer). Thus the only elements that can enter u's element buffer during op_t are elements (k,p) that were in $\mathcal{L}(\mathcal{S}, t')$ or where an Insert(k,p) or DecreaseKey(k,p) was performed during $\mathrm{op}_{t'+1}, \ldots, \mathrm{op}_t$. Thus for u's buffer to overflow, we must have at least B elements in $\mathcal{L}(\mathcal{S}, t') \cup \mathcal{K}(\mathcal{S}, t'+1, t)$. Since $(\mathcal{L}(\mathcal{S}, t') \cup \mathcal{K}(\mathcal{S}, t'+1, t)) \subseteq (\mathcal{L}(\mathcal{S}, t-2^d B/8+1, t) \cup \mathcal{K}(\mathcal{S}, t-2^d B/8+1, t))$ this contradicts that u is not overloaded at op_t. □

Finally, what remains to be proved is that no node becomes overloaded with good probability:

Lemma 6. *Let $\mathcal{S} := \mathrm{op}_1, \ldots, \mathrm{op}_N$ be a sequence of operations on a priority queue, satisfying that there are at most n elements in $\mathcal{L}(\mathcal{S}, t)$ for any op_t. If we use a q-wise independent hash function h for any $q \leq B/2$, then it holds with probability at least $1 - Nn/e^{q/4}$ that no node $u \in \mathcal{T}$ is overloaded at any operation op_t.*

Proof. Define an event $E_{u,t}$ that happens if u is overloaded at op_t. Assume first u is an internal node. The set $\mathcal{K}(t-2^d B/8+1, t)$ has size at most $2^d B/8$. Each key $k \in \mathcal{K}(t-2^d B/8+1, t)$ hashes to u's subtree independently with probability 2^{-d}. Thus the expected number of keys k where $h(k)$ describes a path into u's subtree is $\mu \leq B/8$. By a Chernoff bound ($\Pr[X \geq (1+\delta)\mu] \leq e^{-\delta\mu/3}$ for $\delta \geq 1$), we get that the probability that there are at least $B/2$ elements that are distributed to v's subtree is no more than $e^{-(B/(2\mu)-1)\mu/3} = e^{-(B/6-\mu/3)} \leq e^{-B/12}$. If u is a leaf, then $d = \lceil \log(16n/B) \rceil$ and we have $|\mathcal{K}(t-2^d B/8+1, t) \cup \mathcal{L}(\mathcal{S}, t-2^d B/8)| \leq 2^d B/8 + n \leq 2^d B/8 + 2^d B/16 =$

$(3/16)2^d B$. Each key $k \in \mathcal{K}(t-2^d B/8+1, t) \cup \mathcal{L}(\mathcal{S}, t-2^d B/8)$ hashes to u's subtree with probability 2^{-d} and we get that the expected number of keys with $h(k)$ describing a path into u's subtree is at most $\mu \leq (3/16)B$. A Chernoff bound again gives $\Pr[E_{u,t}] \leq e^{-(B/6-\mu/3)} \leq e^{-B/12}$. A union bound over all $\leq n$ nodes $u \in \mathcal{T}$ and all N operations gives the claimed result.

If we use a q-wise independent hash function h instead of a truly random hash function, then using a Chernoff bound with q-wise independent random variables, we can still conclude that $\Pr[E_{u,t}] \leq e^{-q/4}$. To see this, note that for q-wise independent indicator random variables $X_1, \ldots, X_n$ with $X = \sum_i X_i$ and $p = \Pr[X_i = 1]$ for all i, if $\ell \geq 2q$, then $\Pr[X > \ell] \leq (2\mathbb{E}[X]/\ell)^q$. Setting $\ell = B/2$ and using that $\mathbb{E}[X] \leq (3/16)B$, we get that $\Pr[X > B/2] \leq (3/4)^q = e^{-q\ln(4/3)} < e^{-q/4}$ provided that $B \geq 2q$. □

2.2.2 Performance

Here we analyse the performance of the priority queue. First we analyse the number of server blocks that are accessed while processing a sequence of N operations. We observe that all operations on the root incur no accesses at the server (since the root is stored in client memory). We note that it is important that the root remains in client memory between operations and isn't fetched during every operation as this would require reading $O(m/w)$ blocks for each operation. We thus focus on bounding the number of accesses to nodes u of depth $d \geq 1$. A node of depth d is accessed during PushDown$(d-1)$, PullUp$(d-1)$, PushDown(d) and PullUp(d). Each of these causes us to read u into client memory. Since u stores two buffers that can hold B elements of r bits each, this accesses $O(Br/w)$ server blocks. There are 2^d nodes at depth d and they are accessed $O(1)$ times for every $2^{d-1}B/8$ operations. The height of $\mathcal{T}$ is $\lceil \log(16n/B) \rceil$, hence the total number of blocks accessed at the server is

$$\begin{aligned}
\sum_{i=1}^{\lceil \log(16n/B) \rceil} \frac{N}{2^{d-1}B/8} \cdot 2^d \cdot O(Br/w) &= \\
\sum_{i=1}^{\lceil \log(16n/B) \rceil} O(Nr/w) &= \\
O(\log(16n/B) \cdot Nr/w) &= \\
O((Nr/w) \cdot \log(nr/m)).
\end{aligned}$$

That is, the amortized bandwidth cost is $O((r/w)\log(nr/m))$ blocks.

Running Time. We also argue about the running time, which is different from the number of server blocks accessed. In particular, the running time includes the work performed on data in client memory. To obtain fast $O(\log n)$ amortized time per operation, we need to choose concrete representations of the buffers. These are: In the root node, we store the elements of the element buffer in a standard comparison based priority queue (e.g. a binary heap) with $O(\log B)$ time per operation, where B is the number of elements stored in it. The root also stores a hash table on the priority queue, mapping keys to the positions of the elements in the queue (this is necessary to support Delete and DecreaseKey in $O(\log B)$ time when one is not given a pointer to an element's position in the queue but only the element's key k). All nodes of the tree, including the root, store the set of keys in the delete buffer in a hash table. For all the hash tables, we use a standard linear

space solution with $O(1)$ expected time operations, like e.g. a linear probing hash table. For nodes other than the root, we always store the element buffer in sorted order of priority.

We argue that we can maintain the above representations efficiently during our procedures above.

- The Insert and DecreaseKey procedures need a look-up in the hash tables on the set of keys in the element buffer of the root. It might also need to perform a DecreaseKey on the heap representing the element buffer. Finally, it may also need to perform an Insert in the heap. All in all, this costs $O(\log B)$ expected time since the root's buffers can hold B elements.

- The Delete procedure needs a look-up in the hash tables on the set of keys in the root's element buffer. It might also need to perform a Delete on the heap representing the buffer and potentially an Insert into the hash table representing the delete buffer. This costs $O(\log B)$ expected time.

- The ExtractMin procedure needs to make an ExtractMin on the heap representing the element buffer. It also needs to insert a key into the hash table storing the delete buffer. This costs $O(\log B)$ expected time in total.

- When PushDown(i) visits a node v, it takes $O(Br/w)$ time to load v and its children into client memory. When distributing keys to the children, we scan the element buffers of the children, and for each element (k, p) we ask the hash-table on v's delete-buffer whether k is there. In this way, we can remove all elements that need to be deleted from the childrens' element buffers in $O(B)$ time. Inserting the keys from v's delete buffer into the hash tables representing the childrens' delete buffers also takes expected $O(B)$ time. When distributing elements from v's element buffer to the children, we start by sorting the elements in v's element buffer by priority in case v is the root. This takes $O(B \log B)$ time using a comparison-based sorting algorithm. If v is not the root, they are already sorted and we don't need to perform this extra work. We then create a hash table H mapping the keys k of the elements (k, p) in v's element buffer to the element (k, p). We then scan the element buffers of the children, and for each element (k', p'), we query the hash table H to see if (k', p) is stored in v's element buffer for some p. If it is, we compare p and p'. If $p < p'$, we remove (k', p') from the child's element buffer. Otherwise, we remove (k, p) from H. By moving elements to the front of the child's element buffer during the scan, we can do this in $O(B)$ time while maintaining the sorted order. Finally, we can compute the new sorted list of elements in the childrens' element buffers by doing a standard merge of the sorted list representing v's element buffer and the sorted lists representing the childrens' element buffers. While doing so, we check for each element (k, p) in v's element buffer whether k is still in H. If not, we simply remove (k, p). The distribution of the elements in the element buffers thus takes $O(B)$ time.

- When PullUp(i) visits a node v, it takes $O(Br/w)$ time to load v and its children into client memory. Since the elements in v's childrens' element buffers are stored in sorted order, we can do a merge to find the $B/2$ smallest elements in $O(B)$ time. Removing them from the childrens' element buffers and inserting them in v's element buffer also takes $O(B)$ time.

In addition to the above work, we need to evaluate the q-wise independent hash function h once per Insert, Delete and DecreaseKey. Using a standard degree $q - 1$ polynomial allows this to be done in $O(q)$ time. We now analyse the total amount of work done. For the root node, each operation

costs $O(\log B + q)$ time, and for every $B/8$ operations, we do an additional $O(B \log B)$ work, giving an amortized cost at the root of $O(\log B + q)$ per operation. For each node at depth d, we do $O(B)$ work once for every $2^d B/8$ operations. There are 2^d nodes of depth d, hence the total work over a sequence of N operations is $\sum_{d=1}^{\lceil \log(16n/B) \rceil} (8N/2^d B) \cdot 2^d \cdot O(B) = O(N \log(16n/B))$. Hence the amortized cost per operation is $O(q + \log B + \log(16n/B)) = O(q + \log n)$. We can set q as large as $B/2 = \Theta(m/r)$ while still storing the hash function in client memory, which reduces the failure probability to $1 - Nn/e^{q/4}$.

3 Offline Oblivious RAM

We construct an efficient offline oblivious RAM from oblivious priority queues that only support the operations ExtractMin and Insert. The resulting ORAM construction has the same asymptotic bandwidth overhead as the underlying oblivious priority queue and thus by instantiating our construction here with our oblivious priority queue construction from Section 2.1, we obtain an asymptotically optimal offline ORAM.

Notation Following the notation of [PPRY18], we use $\mathsf{Addrs}[\mathsf{Alg}]$ to denote the memory access pattern of an algorithm that uses external memory, which consist of all the accessed addresses in the external memory.

Definition 6 (Offline Oblivious RAM)**.** *An Offline Oblivious RAM scheme* $\mathsf{ORAM} = (\mathsf{Init}, \mathsf{Access})$ *consists of the following algorithms:*

- $(\widetilde{D}, \mathrm{st}) \leftarrow \mathsf{Init}(1^\lambda, D, \mathrm{P})$*: The initialization algorithm takes the database* D *and access pattern* $\mathrm{P} = (\mathrm{addr}_1, \ldots, \mathrm{addr}_n)$ *as input and outputs a state* st *and a memory data structure* $\widetilde{D}$*.*
- $(v, \mathrm{st}') \leftarrow \mathsf{Access}(\mathrm{st}, \widetilde{D}, \mathrm{I})$*: The access algorithm takes as input the memory data structure* $\widetilde{D}$*, the current state* st*, and an instruction* $\mathrm{I} = (\mathrm{op}, \mathrm{addr}, \mathrm{data})$*, where* $\mathrm{op} \in \{\mathrm{read}, \mathrm{write}\}$*. If* $\mathrm{op} = \mathrm{read}$*, then* $\mathrm{data} = \perp$ *and the algorithm sets* v *to be the data block stored in* $\widetilde{D}$ *at position* addr*. If* $\mathrm{op} = \mathrm{write}$*, then the algorithm writes* data *into the memory* $\widetilde{D}$ *at location* addr *and sets* $v = \perp$*. The algorithm returns* v *and a updated state* st'*.*

An offline ORAM is **correct** *if for every* $i \in [|P|]$*,* $v_i = v_i'$ *where* v_i *and* v_i' *are the values returned by the* i*-th access operation of the Offline ORAM (on* $\widetilde{D}$*) and RAM (on* D*) respectively.*
An offline ORAM is **secure** *if there exists a simulator* Sim *such that for every PPT adversary* $\mathcal{A}$*, and any* $n = \mathrm{poly}(\lambda)$*,*

$$\left| \Pr\left[\mathbf{Exp}_{\mathcal{A}}^{\mathsf{Real}}(n, \lambda) = 1 \right] - \Pr\left[\mathbf{Exp}_{\mathcal{A}}^{\mathsf{Sim}}(n, \lambda) = 1 \right] \right| \leq \mathsf{negl}(\lambda)$$

where $\mathbf{Exp}_{\mathcal{A}}^{\mathsf{Real}}$ *and* $\mathbf{Exp}_{\mathcal{A}}^{\mathsf{Sim}}$ *are defined below.*

Notice that the goal of an oblivious RAM is to hide any information revealed by the access pattern during a RAM execution. Therefore it's natural to assume that the memory is already encrypted and that every cell accessed, is re-encrypted and rewritten in the memory (in RAM), otherwise hiding the addresses accessed is the least of anyone's worries! Consequently the simulator uses the same distribution to sample the content of the simulated $\widetilde{D}$; what is the focus of our attention here is that the outputs of $\mathsf{Addrs}[\cdot]$ produced in the two experiments are indistinguishable from each other.

$\mathbf{Exp}^{\mathsf{Real}}_{\mathcal{A}}(n,\lambda)$

$(D,P) \leftarrow \mathcal{A}(1^\lambda)$

$\mathcal{X} \leftarrow \mathsf{Addrs}\Big[(\widetilde{D},\mathrm{st}) \leftarrow \mathsf{Init}(1^\lambda, D, P)\Big]$

for $j = 1$ to n

  $I_j \leftarrow \mathcal{A}(\widetilde{D}, \mathcal{X})$

  If $\mathsf{addr}_j \neq P[j]$ **return** 0.

  $\mathcal{X}^* \leftarrow \mathsf{Addrs}\Big[(v,\mathrm{st}) \leftarrow \mathsf{Access}(\mathrm{st}, \widetilde{D}, I_j)\Big]$

  $\mathcal{X} \leftarrow \mathcal{X}^* \cup \mathcal{X}$

return $b \leftarrow \mathcal{A}(\widetilde{D}, \mathcal{X})$.

$\mathbf{Exp}^{\mathsf{Sim}}_{\mathcal{A}}(n,\lambda)$

$(D,P) \leftarrow \mathcal{A}(1^\lambda)$

$\mathcal{X} \leftarrow \mathsf{Addrs}\Big[(\widetilde{D},\mathrm{st}) \leftarrow \mathrm{Sim}(1^\lambda, |D|, |P|)\Big]$

for $j = 1$ to n

$I_j \leftarrow \mathcal{A}(\widetilde{D}, \mathcal{X})$

If $\mathsf{addr}_j \neq P[j]$ **return** 0.

$\mathcal{X}^* \leftarrow \mathsf{Addrs}\Big[(v,\mathrm{st}) \leftarrow \mathrm{Sim}(\mathrm{st}, \widetilde{D}, acc)\Big]$

$\mathcal{X} \leftarrow \mathcal{X}^* \cup \mathcal{X}$

return $b \leftarrow \mathcal{A}(\widetilde{D}, \mathcal{X})$.

Figure 1: Real and Simulated Experiments for defining the security of the Offline ORAM

3.1 Construction of Offline ORAM

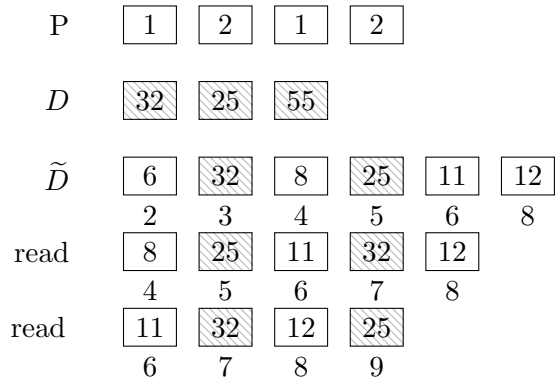

Figure 2: P is a arbitrary, but fixed access pattern represented by an ordered sequence of addresses, D is the initial RAM state, and the initial state of the priority queue representing the encoded oblivious RAM is $\widetilde{D}$. For the priority queue $\widetilde{D}$, the keys are written inside the box and the priorities are written below. The last two rows represent the state of the encoded memory after two read operations have been performed.

In an offline ORAM, the access pattern $\mathrm{P} = (\mathrm{addr}_1, \ldots, \mathrm{addr}_n)$ is fixed a-priori and known during the initialization of the ORAM data structure $\widetilde{D}$. Knowing this, it is possible to create an ordered list for each memory cell addr, which consists of the time-steps when that cell is accessed, in other words $\mathsf{AcTi}_{\mathsf{addr}} = \{i \in \{1,\ldots,n\}\, s.t.\ \ P[i] = \mathsf{addr}\}$. Let $s_{\mathsf{addr}} = |\mathsf{AcTi}_{\mathsf{addr}}|$. Now imagine we create a priority queue and store every memory cell addr by inserting $(D[\mathsf{addr}], \mathsf{AcTi}_{\mathsf{addr}}[j])$ for all $j \in \{1,\ldots,s_{\mathsf{addr}}\}$ in the priority queue. Then cell addr is extracted from the priority queue at time $\mathsf{AcTi}_{\mathsf{addr}}[1]$ and then at time $\mathsf{AcTi}_{\mathsf{addr}}[2]$ and so on and so forth for all $\mathsf{addr} \in P$. However we do not know what the memory cell addr contains *at all times* during preprocessing. We only know

their initial values[2]. Therefore we cannot actually save the memory cell's content for all future time-steps when creating the priority queue. What we can do is to separate the two components "data" and"access time" that were saved together in our previous approach and save them into the queue with two consecutive priorities p and $p+1$. And make sure that each operation extracts two elements from the priority queue. As a result, the priority for the operation i is no longer i but $2i$[3]. If the ith operation is the jth time addr is accessed (i.e $P[i] = \mathsf{addr}$ and $\mathsf{AcTi}_{\mathsf{addr}}[j] = i$), then we add $(2\mathsf{AcTi}_{\mathsf{addr}}[j+1], p)$ to the queue which saves the next "access time" for addr; this can be done during the initialization for all operations in P. And we would like to add $(D[\mathsf{addr}], p+1)$ to save the correct content for the ith access operation. If the ith access is the first time addr is accessed then the correct value for $D[\mathsf{addr}]$ is known during the initialization and can be added to the queue. If $j > 1$, we need to make sure that at time $\mathsf{AcTi}_{\mathsf{addr}}[j-1]$ the updated value of $D[\mathsf{addr}]$ is inserted into the queue with priority $p+1 = 2\mathsf{AcTi}_{\mathsf{addr}}[j]+1$. Consequently, each RAM operation is translated into 2 ExtractMin and one Insert operation on the priority queue. Finally using an oblivious priority queue will ensure that no information about the priorities and operations performed on the queue is revealed. We now provide a formal description of our construction and illustrate how it works in Figure 2.

Notation We index lists and arrays starting from 1. Let $\mathrm{P} = (\mathrm{addr}_1, \ldots, \mathrm{addr}_n)$ be the ordered sequence of memory cells that will be accessed and let D be the initial memory content. Let A be the set of all addresses accessed in P. Let s_{addr} be the number of times addr is repeated in P. Finally let $\mathsf{AcTi}_{\mathsf{addr}}$ be as defined above. We define our offline oblivious RAM scheme $\Pi = (\mathsf{Init}, \mathsf{Access})$ as follows

Init$(1^\lambda, D, \mathrm{P})$**:**

1. Create $\mathsf{AcTi}_{\mathsf{addr}}$ for each $\mathsf{addr} \in D$. Then add one more auxiliary element to $\mathsf{AcTi}_{\mathsf{addr}} := \mathsf{AcTi}_{\mathsf{addr}} \cup \{n + \mathsf{addr}\}$. // this is a dummy element added to list of time-steps.
2. Initialize an empty oblivious priority queue Q.
3. For each $\mathsf{addr} \in D$ insert $(D[\mathsf{addr}], 2\mathsf{AcTi}_{\mathsf{addr}}[1]+1)$ to Q.
4. For each $\mathsf{addr} \in \mathsf{A}$ and for $j = 1, \ldots, s_{\mathsf{addr}}$, insert $(2\mathsf{AcTi}_{\mathsf{addr}}[j+1], 2\mathsf{AcTi}_{\mathsf{addr}}[j])$ to Q.
5. Return Q as the encoded memory $\widetilde{D}$.

Access$(\mathrm{st}, \widetilde{D}, \mathrm{I} = (\mathrm{op}, \mathrm{addr}, \mathrm{data}))$**:**

1. ExtractMin to obtain the lowest priority element in $\widetilde{D}$ and call it p.
2. ExtractMin again and call the returned value v.
3. If op = write, then set $v :=$ data.
4. (Irrespective of which operation is performed,) insert $(v, p+1)$ into the priority queue
5. If op = read, then we return v else return $\perp$.

[2]We assume all memory cells have some initial value, without loss of generality.

[3]for simplicity we consider $p = 2i$ for the ith operation and therefore the smallest priority in the queue is 2.

Theorem 5. *Assume the simple oblivious priority queue processes a sequence of $n^* = 4n + d$ operations using amortized x memory accesses at the server and uses m bits of client memory, with element size r and server block size w and is correct with probability p and is statistically secure. Then Offline Oblivious RAM Π described above is correct with the same correctness and security and efficiency parameters as the oblivious priority queue if the offline RAM processes $n = \text{poly}(d)$ operations and has memory D of size $|D| = d$ and the same element size and server block size as the OPQ.*

Corollary 1. *For any $m = \Omega(\log^2 n)$, there exists an offline oblivious RAM which can process n operations with amortized bandwidth cost $O(\log n)$ blocks and using m bits of client memory, with element size $\Theta(\log n)$ and server block size $\Theta(\log n)$ which is correct with probability at least $1 - e^{-\Omega(m/\log n)}$ and is statistically secure.*

Proof of Security. Consider a simulator that is given $|D|$ and $|P|$. Then the simulator initializes an oblivious priority queue and inserts $|D| + |P|$ elements (of correct size) in it and returns it as $\widetilde{D}$ then every time the simulator is called to access $\widetilde{D}$ it will perform two ExtractMin operations and one insert operation. Therefore by the security of Oblivious Priority Queue the two experiments are indistinguishable.

Proof of Correctness. We show that the following statement is true, from which follows that the Offline ORAM is correct assuming the OPQ is correct.

For $i \in \{1, \ldots, n\}$, at time i (when the i-th RAM operation is performed),

a) the two elements in the OPQ with the smallest priorities are the following, assuming i is the jth operation accessing address r, or in other words $\mathsf{AcTi}_r[j] = i$:

$$(2\mathsf{AcTi}_r j + 1], 2\mathsf{AcTi}_r[j] = 2i), \quad (D[r, j-1], 2\mathsf{AcTi}_r[j] + 1 = 2i + 1)$$

where $D[r, j]$ represents the value of $D[r]$ after the jth accesses to cell r.

b) and element $(D[r, j], 2\mathsf{AcTi}_r[j+1] + 1)$ is added to the queue.

Proof. First notice that at the initialization stage $|D| + n$ elements are inserted in an empty OPQ; Of which, n have even number priorities $2i$ (one for each access operation) and $|D|$ have odd number priorities, and only for addresses accessed in P these odd priorities are of the form $2i + 1$ for some $i \in \{1, \ldots, n\}$, the rest are priorities too large to be ever extracted from the queue (we will show this more formally shortly). Also note that the even number priorities $(k, 2i)$ for $i \in \{1, \ldots, n\}$ have a time-step as their keys, more specifically the next time step when the same address is going to be accessed; while the odd number priorities have that cell's content as their key.

Base Cases

1. At time 1, $D[r, 0] = D[r]$ and $j = 1$. We can see immediately that the two elements $(2\mathsf{AcTi}_r[2], 2)$ and $(D[r, 0], 3)$ are indeed in the queue (added during the initialization) and since 2 and 3 are the smallest priorities possible, they clearly satisfy Part a of the statement. Therefore once operation $\mathsf{Access}(\text{st}, \widetilde{D}, \text{I}_1 = (\text{op}, r, \text{data}))$ is performed, $p = 2\mathsf{AcTi}_r[2]$ and $v = D[r, 0]$. If op = read then $D[r, 1] := v$ is returned if op = write $D[r, 1] :=$ data and in both cases $(D[r, 1], 2\mathsf{AcTi}_r[2] + 1)$ is inserted into the queue. Therefore the second part of the statements holds too.

2. At time $i = 2$

   (a) if $r = P[i]$ is accessed for the first time (i.e $j = 1$, and $\mathsf{AcTi}_r[1] = i$), then we know (according to the initialization phase) that $(2\mathsf{AcTi}_r[2], 2i = 4)$ and $(D[r], 2i + 1 = 5)$ are in the priority queue. And since 4 and 5 are the next smallest numbers (after removal of 2 and 3 at time step 1,) the first part of the statement holds and following the same reasoning as before, the second part holds as well.

   (b) if r is accessed for the second time, then $\mathsf{AcTi}_r[2] = 2$ and therefore since $(2\mathsf{AcTi}_r[3], 2\mathsf{AcTi}_r[2])$ was added during the initialization and we added $(v = D[r, 1], 2\mathsf{AcTi}_r[2] + 1)$ at time $\mathsf{AcTi}_r[1] = 1$, both elements exists and have the smallest priorities. Thus they are removed during the ith/second operation and we have $p = 2\mathsf{AcTi}_r[3]$ and $v = D[r, 1]$. If the operation is read, $D[r, 2] := v$ is returned, if op = write $D[r, 2] :=$ data and in both cases $(D[r, 2], 2\mathsf{AcTi}_r[3] + 1)$ is inserted into the queue. Therefore the second part of the statements holds too

Induction Step Assume that for any $i \leq \ell < n - 1$ and for $r = P[i]$ and $\mathsf{AcTi}_r[j] = i$ it holds that at time i, $(2\mathsf{AcTi}_r[j + 1], 2\mathsf{AcTi}_r[j] = 2i)$ and $(D[r, j - 1], 2\mathsf{AcTi}_r[j] + 1 = 2i + 1)$ are in the queue and have the two smallest priorities and that $(D[r, j], 2\mathsf{AcTi}_r[j+1]+1)$ is added to the queue at the end of ith operation (induction hypothesis)

Since at every step i before $\ell+1$ the two elements removed from the queue have priorities $2i$ and $2i+1$, if we show that two elements with priorities $2(\ell+1)$ and $2(\ell+1)+1 = 2\ell+3$ exist in the queue, it follows that they also have the smallest priorities (all smaller ones are removed in the previous steps) and will be removed from it during the $\ell+1$st operation. Let $P[\ell+1] = r'$ and $\mathsf{AcTi}_{r'}[j] = \ell+1$ for some $j \in \{1, \ldots, s_{r'}\}$. We know that $(2\mathsf{AcTi}_{r'}[j+1], 2\mathsf{AcTi}_{r'}[j] = 2\ell+2)$ was added to the queue in the initialization phase. If $j = 1$ then, it is the first time that r' is accessed and therefore $(D[r', 0], 2\mathsf{AcTi}_{r'}[j] + 1 = 2\ell + 3)$ was also added during the initialization. If however $j > 1$ then according to the induction hypothesis at time $\mathsf{AcTi}_{r'}[j - 1] \leq \ell$ $(D[r', j - 1], 2\mathsf{AcTi}'_r[j] + 1 = 2\ell + 3)$ was added to the queue. Therefore the correct two elements with priorities $2\ell + 2$ and $2\ell + 3$ are in the queue. Once operation $\mathsf{Access}(\mathrm{st}, \widetilde{D}, \mathrm{I}_{\ell+1} = (\mathrm{op}, r', \mathrm{data}))$ is performed, $p = 2\mathsf{AcTi}_{r'}[j + 1]$ and $v = D[r, j - 1]$. If op = read then $D[r, j] := v$ is returned, if op = write, $v = D[r, j] :=$ data and in both cases $(v, 2\mathsf{AcTi}_r[j + 1] + 1)$ is inserted into the queue.

This shows that at every step the value returned by the offline oblivious RAM is the same as the output of a RAM executing the same accesses P. Furthermore since only $n = |P|$ operations are performed and during those operations elements with priorities $\{2, 3, \ldots, 2n + 1\}$ are removed, none of the elements with higher priorities ($p \geq 2(n+1)$) are removed from the queue. Therefore the correctness of the scheme follows directly from the correctness of the oblivious priority queue. □

4 Offline Oblivious RAM

Our construction of offline oblivious RAM is deferred to Appendix A.

5 Oblivious Sorting and Shuffling

In this section we outline how our OPQ leads to optimal oblivious sorting and shuffling algorithms. Recall, that a sorting or shuffling algorithm is called oblivious, if the access pattern is independent

of the memory cell's contents that are being sorted or shuffled. Let $\mathcal{X} = x_1, \ldots, x_N$ be the memory contents that we want to sort or shuffle. For the sake of simplicity, we assume that both data elements and priorities come from the same universe and that all memory cells contain distinct entries. Let π be a function, which defines how the memory cells should be rearranged. For the case of sorting, π maps each index $i \in [N]$ to x_i. For the case of shuffling, π is a uniformly random permutation on $[N]$. For our algorithm here we only need the simple version of our OPQ from Section 2.1.

Our algorithm works as follows. We initialize an empty priority queue. For each $i \in [N]$, we read x_i from $\mathcal{X}$ and then insert x_i with priority $\pi(i)$ into the oblivious priority queue Once all N elements are inserted, we then use ExtractMin N times to obtain the permuted memory cells $\mathcal{X}' = x'_1, \ldots, x'_N$, where each x'_i is the output of the i-th ExtractMin call.

It is easy to see that the above algorithm is oblivious, since it performs one data-independent iteration over $\mathcal{X}$ and only uses a fixed number of operations on the OPQ afterwards. The correctness of our sorting/shuffling algorithm follows from the correctness of the underlying OPQ. For example, for the case of sorting, the elements are inserted with their priorities being their actual values. Since the OPQ returns the elements sorted according to their priorities, it follows that the elements are sorted according to their values.

In terms of efficiency, we perform $2N$ OPQ accesses and N direct accesses to $\mathcal{X}$, which means that our amortized overhead is is a constant factor larger than the overhead of the underlying OPQ.

6 Offline Oblivious RAM

We construct an efficient offline oblivious RAM from oblivious priority queues that only support the operations ExtractMin and Insert. The resulting ORAM construction has the same asymptotic bandwidth overhead as the underlying oblivious priority queue and thus by instantiating our construction here with our oblivious priority queue construction from Section 2.1, we obtain an asymptotically optimal offline ORAM.

Notation Following the notation of [PPRY18], we use $\mathsf{Addrs}[\mathsf{Alg}]$ to denote the memory access pattern of an algorithm that uses external memory, which consist of all the accessed addresses in the external memory.

Definition 7 (Offline Oblivious RAM)**.** *An Offline Oblivious RAM scheme* $\mathsf{ORAM} = (\mathsf{Init}, \mathsf{Access})$ *consists of the following algorithms:*

- $(\widetilde{D}, \mathrm{st}) \leftarrow \mathsf{Init}(1^\lambda, D, \mathrm{P})$*: The initialization algorithm takes the database* D *and access pattern* $\mathrm{P} = (\mathrm{addr}_1, \ldots, \mathrm{addr}_n)$ *as input and outputs a state* st *and a memory data structure* $\widetilde{D}$*.*
- $(v, \mathrm{st}') \leftarrow \mathsf{Access}(\mathrm{st}, \widetilde{D}, \mathrm{I})$*: The access algorithm takes as input the memory data structure* $\widetilde{D}$*, the current state* st*, and an instruction* $\mathrm{I} = (\mathrm{op}, \mathrm{addr}, \mathrm{data})$*, where* $\mathrm{op} \in \{\mathrm{read}, \mathrm{write}\}$*. If* $\mathrm{op} = \mathrm{read}$*, then* $\mathrm{data} = \perp$ *and the algorithm sets* v *to be the data block stored in* $\widetilde{D}$ *at position* addr*. If* $\mathrm{op} = \mathrm{write}$*, then the algorithm writes* data *into the memory* $\widetilde{D}$ *at location* addr *and sets* $v = \perp$*. The algorithm returns* v *and a updated state* st'*.*

An offline ORAM is **correct** *if for every* $i \in [|P|]$*,* $v_i = v'_i$ *where* v_i *and* v'_i *are the values returned by the* i*-th access operation of the Offline ORAM (on* $\widetilde{D}$*) and RAM (on* D*) respectively.*

An offline ORAM is **secure** *if there exists a simulator* Sim *such that for every PPT adversary* $\mathcal{A}$*, and any* $n = \text{poly}(\lambda)$*,*

$$\left|\Pr\left[\mathbf{Exp}_{\mathcal{A}}^{\mathsf{Real}}(n,\lambda)=1\right]-\Pr\left[\mathbf{Exp}_{\mathcal{A}}^{\mathsf{Sim}}(n,\lambda)=1\right]\right|\leq \mathsf{negl}(\lambda)$$

where $\mathbf{Exp}_{\mathcal{A}}^{\mathsf{Real}}$ *and* $\mathbf{Exp}_{\mathcal{A}}^{\mathsf{Sim}}$ *are defined below.*

$\mathbf{Exp}_{\mathcal{A}}^{\mathsf{Real}}(n,\lambda)$

$(D,P)\leftarrow\mathcal{A}(1^\lambda)$

$\mathcal{X}\leftarrow\mathsf{Addrs}\Big[(\widetilde{D},\text{st})\leftarrow\mathsf{Init}(1^\lambda,D,P)\Big]$

for $j=1$ to n

  $I_j\leftarrow\mathcal{A}(\widetilde{D},\mathcal{X})$

  If $\mathsf{addr}_j\neq P[j]$ **return** 0.

  $\mathcal{X}^*\leftarrow\mathsf{Addrs}\Big[(v,\text{st})\leftarrow\mathsf{Access}(\text{st},\widetilde{D},I_j)\Big]$

  $\mathcal{X}\leftarrow\mathcal{X}^*\cup\mathcal{X}$

return $b\leftarrow\mathcal{A}(\widetilde{D},\mathcal{X})$.

$\mathbf{Exp}_{\mathcal{A}}^{\mathsf{Sim}}(n,\lambda)$

$(D,P)\leftarrow\mathcal{A}(1^\lambda)$

$\mathcal{X}\leftarrow\mathsf{Addrs}\Big[(\widetilde{D},\text{st})\leftarrow\text{Sim}(1^\lambda,|D|,|P|)\Big]$

for $j=1$ to n

  $I_j\leftarrow\mathcal{A}(\widetilde{D},\mathcal{X})$

  If $\mathsf{addr}_j\neq P[j]$ **return** 0.

  $\mathcal{X}^*\leftarrow\mathsf{Addrs}\Big[(v,\text{st})\leftarrow\text{Sim}(\text{st},\widetilde{D},acc)\Big]$

  $\mathcal{X}\leftarrow\mathcal{X}^*\cup\mathcal{X}$

return $b\leftarrow\mathcal{A}(\widetilde{D},\mathcal{X})$.

Figure 3: Real and Simulated Experiments for defining the security of the Offline ORAM

Notice that the goal of an oblivious RAM is to hide any information revealed by the access pattern during a RAM execution. Therefore it's natural to assume that the memory is already encrypted and that every cell accessed, is re-encrypted and rewritten in the memory (in RAM), otherwise hiding the addresses accessed is the least of anyone's worries! Consequently the simulator uses the same distribution to sample the content of the simulated $\widetilde{D}$; what is the focus of our attention here is that the outputs of $\mathsf{Addrs}[\cdot]$ produced in the two experiments are indistinguishable from each other.

6.1 Construction of Offline ORAM

In an offline ORAM, the access pattern $\text{P} = (\text{addr}_1, \ldots, \text{addr}_n)$ is fixed a-priori and known during the initialization of the ORAM data structure $\widetilde{D}$. Knowing this, it is possible to create an ordered list for each memory cell addr, which consists of the time-steps when that cell is accessed, in other words $\mathsf{AcTi}_{\mathsf{addr}} = \{i \in \{1,\ldots,n\}\ s.t.\ \ P[i] = \mathsf{addr}\}$. Let $s_{\mathsf{addr}} = |\mathsf{AcTi}_{\mathsf{addr}}|$. Now imagine we create a priority queue and store every memory cell addr by inserting $(D[\mathsf{addr}], \mathsf{AcTi}_{\mathsf{addr}}[j])$ for all $j \in \{1,\ldots,s_{\mathsf{addr}}\}$ in the priority queue. Then cell addr is extracted from the priority queue at time $\mathsf{AcTi}_{\mathsf{addr}}[1]$ and then at time $\mathsf{AcTi}_{\mathsf{addr}}[2]$ and so on and so forth for all $\mathsf{addr} \in P$. However we do not know what the memory cell addr contains *at all times* during preprocessing. We only know their initial values[4]. Therefore we cannot actually save the memory cell's content for all future time-steps when creating the priority queue. What we can do is to separate the two components "data" and"access time" that were saved together in our previous approach and save them into the

[4]We assume all memory cells have some initial value, without loss of generality.

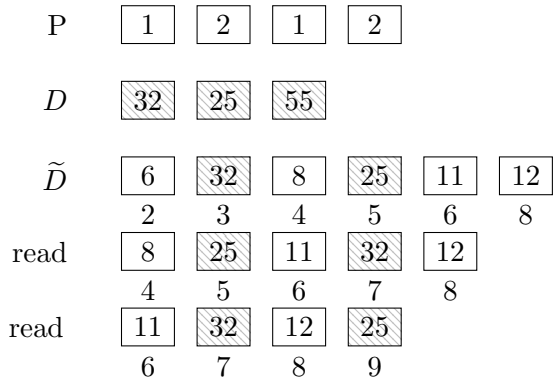

Figure 4: P is a arbitrary, but fixed access pattern represented by an ordered sequence of addresses, D is the initial RAM state, and the initial state of the priority queue representing the encoded oblivious RAM is $\widetilde{D}$. For the priority queue $\widetilde{D}$, the keys are written inside the box and the priorities are written below. The last two rows represent the state of the encoded memory after two read operations have been performed.

queue with two consecutive priorities p and $p+1$. And make sure that each operation extracts two elements from the priority queue. As a result, the priority for the operation i is no longer i but $2i$[5]. If the ith operation is the jth time addr is accessed (i.e $P[i] = \mathsf{addr}$ and $\mathsf{AcTi}_{\mathsf{addr}}[j] = i$), then we add $(2\mathsf{AcTi}_{\mathsf{addr}}[j+1], p)$ to the queue which saves the next "access time" for addr; this can be done during the initialization for all operations in P. And we would like to add $(D[\mathsf{addr}], p+1)$ to save the correct content for the ith access operation. If the ith access is the first time addr is accessed then the correct value for $D[\mathsf{addr}]$ is known during the initialization and can be added to the queue. If $j > 1$, we need to make sure that at time $\mathsf{AcTi}_{\mathsf{addr}}[j-1]$ the updated value of $D[\mathsf{addr}]$ is inserted into the queue with priority $p+1 = 2\mathsf{AcTi}_{\mathsf{addr}}[j]+1$. Consequently, each RAM operation is translated into 2 ExtractMin and one Insert operation on the priority queue. Finally using an oblivious priority queue will ensure that no information about the priorities and operations performed on the queue is revealed. We now provide a formal description of our construction and illustrate how it works in Figure 2.

Notation We index lists and arrays starting from 1. Let $\mathrm{P} = (\mathrm{addr}_1, \ldots, \mathrm{addr}_n)$ be the ordered sequence of memory cells that will be accessed and let D be the initial memory content. Let A be the set of all addresses accessed in P. Let s_{addr} be the number of times addr is repeated in P. Finally let $\mathsf{AcTi}_{\mathsf{addr}}$ be as defined above. We define our offline oblivious RAM scheme $\Pi = (\mathsf{Init}, \mathsf{Access})$ as follows

Init$(1^\lambda, D, \mathrm{P})$**:**

1. Create $\mathsf{AcTi}_{\mathsf{addr}}$ for each $\mathsf{addr} \in D$. Then add one more auxiliary element to $\mathsf{AcTi}_{\mathsf{addr}} := \mathsf{AcTi}_{\mathsf{addr}} \cup \{n + \mathsf{addr}\}$. // this is a dummy element added to list of time-steps.
2. Initialize an empty oblivious priority queue Q.

[5]for simplicity we consider $p = 2i$ for the ith operation and therefore the smallest priority in the queue is 2.

3. For each $\mathsf{addr} \in D$ insert $(D[\mathsf{addr}], 2\mathsf{AcTi}_{\mathsf{addr}}[1] + 1)$ to Q.

4. For each $\mathsf{addr} \in \mathsf{A}$ and for $j = 1, \ldots, s_{\mathsf{addr}}$, insert $(2\mathsf{AcTi}_{\mathsf{addr}}[j+1], 2\mathsf{AcTi}_{\mathsf{addr}}[j])$ to Q.

5. Return Q as the encoded memory $\widetilde{D}$.

Access$(\mathrm{st}, \widetilde{D}, \mathrm{I} = (\mathrm{op}, \mathrm{addr}, \mathrm{data}))$**:**

1. ExtractMin to obtain the lowest priority element in $\widetilde{D}$ and call it p.

2. ExtractMin again and call the returned value v.

3. If op = write, then set $v :=$ data.

4. (Irrespective of which operation is performed,) insert $(v, p+1)$ into the priority queue

5. If op = read, then we return v else return $\perp$.

Theorem 6. *Assume the simple oblivious priority queue processes a sequence of $n^* = 4n + d$ operations using amortized x memory accesses at the server and uses m bits of client memory, with element size r and server block size w and is correct with probability p and is statistically secure. Then Offline Oblivious RAM Π described above is correct with the same correctness and security and efficiency parameters as the oblivious priority queue if the offline RAM processes $n = \mathrm{poly}(d)$ operations and has memory D of size $|D| = d$ and the same element size and server block size as the OPQ.*

Corollary 2. *For any $m = \Omega(\log^2 n)$, there exists an offline oblivious RAM which can process n operations with amortized bandwidth cost $O(\log n)$ blocks and using m bits of client memory, with element size $\Theta(\log n)$ and server block size $\Theta(\log n)$ which is correct with probability at least $1 - e^{-\Omega(m/\log n)}$ and is statistically secure.*

Proof of Security. Consider a simulator that is given $|D|$ and $|P|$. Then the simulator initializes an oblivious priority queue and inserts $|D| + |P|$ elements (of correct size) in it and returns it as $\widetilde{D}$ then every time the simulator is called to access $\widetilde{D}$ it will perform two ExtractMin operations and one insert operation. Therefore by the security of Oblivious Priority Queue the two experiments are indistinguishable.

Proof of Correctness. We show that the following statement is true, from which follows that the Offline ORAM is correct assuming the OPQ is correct.

For $i \in \{1, \ldots, n\}$, at time i (when the i-th RAM operation is performed),

a) the two elements in the OPQ with the smallest priorities are the following, assuming i is the jth operation accessing address r, or in other words $\mathsf{AcTi}_r[j] = i$:

$$(2\mathsf{AcTi}_r j + 1], 2\mathsf{AcTi}_r[j] = 2i), \quad (D[r, j-1], 2\mathsf{AcTi}_r[j] + 1 = 2i + 1)$$

where $D[r, j]$ represents the value of $D[r]$ after the jth accesses to cell r.

b) and element $(D[r, j], 2\mathsf{AcTi}_r[j+1] + 1)$ is added to the queue.

Proof. First notice that at the initialization stage $|D|+n$ elements are inserted in an empty OPQ; Of which, n have even number priorities $2i$ (one for each access operation) and $|D|$ have odd number priorities, and only for addresses accessed in P these odd priorities are of the form $2i+1$ for some $i \in \{1,\ldots,n\}$, the rest are priorities too large to be ever extracted from the queue (we will show this more formally shortly). Also note that the even number priorities $(k, 2i)$ for $i \in \{1,\ldots,n\}$ have a time-step as their keys, more specifically the next time step when the same address is going to be accessed; while the odd number priorities have that cell's content as their key.

Base Cases

1. At time 1, $D[r,0] = D[r]$ and $j = 1$. We can see immediately that the two elements $(2\mathsf{AcTi}_r[2], 2)$ and $(D[r,0], 3)$ are indeed in the queue (added during the initialization) and since 2 and 3 are the smallest priorities possible, they clearly satisfy Part a of the statement. Therefore once operation $\mathsf{Access}(\mathrm{st}, \widetilde{D}, \mathrm{I}_1 = (\mathrm{op}, r, \mathrm{data}))$ is performed, $p = 2\mathsf{AcTi}_r[2]$ and $v = D[r,0]$. If op = read then $D[r,1] := v$ is returned if op = write $D[r,1] :=$ data and in both cases $(D[r,1], 2\mathsf{AcTi}_r[2]+1)$ is inserted into the queue. Therefore the second part of the statements holds too.

2. At time $i = 2$

   (a) if $r = P[i]$ is accessed for the first time (i.e $j = 1$, and $\mathsf{AcTi}_r[1] = i$), then we know (according to the initialization phase) that $(2\mathsf{AcTi}_r[2], 2i = 4)$ and $(D[r], 2i+1 = 5)$ are in the priority queue. And since 4 and 5 are the next smallest numbers (after removal of 2 and 3 at time step 1,) the first part of the statement holds and following the same reasoning as before, the second part holds as well.

   (b) if r is accessed for the second time, then $\mathsf{AcTi}_r[2] = 2$ and therefore since $(2\mathsf{AcTi}_r[3], 2\mathsf{AcTi}_r[2])$ was added during the initialization and we added $(v = D[r,1], 2\mathsf{AcTi}_r[2]+1)$ at time $\mathsf{AcTi}_r[1] = 1$, both elements exists and have the smallest priorities. Thus they are removed during the ith/second operation and we have $p = 2\mathsf{AcTi}_r[3]$ and $v = D[r,1]$. If the operation is read, $D[r,2] := v$ is returned, if op = write $D[r,2] :=$ data and in both cases $(D[r,2], 2\mathsf{AcTi}_r[3]+1)$ is inserted into the queue. Therefore the second part of the statements holds too

Induction Step Assume that for any $i \le \ell < n-1$ and for $r = P[i]$ and $\mathsf{AcTi}_r[j] = i$ it holds that at time i, $(2\mathsf{AcTi}_r[j+1], 2\mathsf{AcTi}_r[j] = 2i)$ and $(D[r,j-1], 2\mathsf{AcTi}_r[j]+1 = 2i+1)$ are in the queue and have the two smallest priorities and that $(D[r,j], 2\mathsf{AcTi}_r[j+1]+1)$ is added to the queue at the end of ith operation (induction hypothesis)

Since at every step i before $\ell+1$ the two elements removed from the queue have priorities $2i$ and $2i+1$, if we show that two elements with priorities $2(\ell+1)$ and $2(\ell+1)+1 = 2\ell+3$ exist in the queue, it follows that they also have the smallest priorities (all smaller ones are removed in the previous steps) and will be removed from it during the $\ell+1$st operation. Let $P[\ell+1] = r'$ and $\mathsf{AcTi}_{r'}[j] = \ell+1$ for some $j \in \{1,\ldots,s_{r'}\}$. We know that $(2\mathsf{AcTi}_{r'}[j+1], 2\mathsf{AcTi}_{r'}[j] = 2\ell+2)$ was added to the queue in the initialization phase. If $j = 1$ then, it is the first time that r' is accessed and therefore $(D[r',0], 2\mathsf{AcTi}_{r'}[j]+1 = 2\ell+3)$ was also added during the initialization. If however $j > 1$ then according to the induction hypothesis at time $\mathsf{AcTi}_{r'}[j-1] \le \ell$ $(D[r', j-1], 2\mathsf{AcTi}'_r[j]+1 = 2\ell+3)$ was added to the queue. Therefore the correct two elements with priorities $2\ell+2$ and $2\ell+3$ are in

the queue. Once operation $\mathsf{Access}(\mathrm{st}, \widetilde{D}, \mathrm{I}_{\ell+1} = (\mathrm{op}, r', \mathrm{data}))$ is performed, $p = 2\mathsf{AcTi}_{r'}[j+1]$ and $v = D[r, j-1]$. If $\mathrm{op} = \mathrm{read}$ then $D[r, j] := v$ is returned, if $\mathrm{op} = \mathrm{write}, v = D[r, j] := \mathrm{data}$ and in both cases $(v, 2\mathsf{AcTi}_r[j+1]+1)$ is inserted into the queue.

This shows that at every step the value returned by the offline oblivious RAM is the same as the output of a RAM executing the same accesses P. Furthermore since only $n = |P|$ operations are performed and during those operations elements with priorities $\{2, 3, \ldots, 2n+1\}$ are removed, none of the elements with higher priorities ($p \geq 2(n+1)$) are removed from the queue. Therefore the correctness of the scheme follows directly from the correctness of the oblivious priority queue. □

References

[AKL+18] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. Optorama: Optimal oblivious ram. Cryptology ePrint Archive, Report 2018/892, 2018. https://eprint.iacr.org/2018/892.

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. An 0(n log n) sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 1–9, New York, NY, USA, 1983. ACM.

[Arg03] Lars Arge. The buffer tree: a technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.

[BN16] Elette Boyle and Moni Naor. Is there an oblivious RAM lower bound? In Madhu Sudan, editor, *ITCS 2016: 7th Conference on Innovations in Theoretical Computer Science*, pages 357–368, Cambridge, MA, USA, January 14–16, 2016. Association for Computing Machinery.

[CNS18] T.-H. Hubert Chan, Kartik Nayak, and Elaine Shi. Perfectly secure oblivious parallel RAM. In Amos Beimel and Stefan Dziembowski, editors, *TCC 2018: 16th Theory of Cryptography Conference, Part II*, volume 11240 of *Lecture Notes in Computer Science*, pages 636–668, Panaji, India, November 11–14, 2018. Springer, Heidelberg, Germany.

[DMN11] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In Yuval Ishai, editor, *TCC 2011: 8th Theory of Cryptography Conference*, volume 6597 of *Lecture Notes in Computer Science*, pages 144–163, Providence, RI, USA, March 28–30, 2011. Springer, Heidelberg, Germany.

[ELY17] Kasper Eenberg, Kasper Green Larsen, and Huacheng Yu. DecreaseKeys are expensive for external memory priority queues. In *STOC'17—Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1081–1093. ACM, New York, 2017.

[FJKT99] R. Fadel, K. V. Jakobsen, J. Katajainen, and J. Teuhola. Heaps and heapsort on secondary storage. *Theoret. Comput. Sci.*, 220(2):345–362, 1999.

[FT87] Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[GM11] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *ICALP 2011: 38th International Colloquium on Automata, Languages and Programming, Part II*, volume 6756 of *Lecture Notes in Computer Science*, pages 576–587, Zurich, Switzerland, July 4–8, 2011. Springer, Heidelberg, Germany.

[GMOT11] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious ram simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 95–100. ACM, 2011.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[Gol87] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In Alfred Aho, editor, *19th Annual ACM Symposium on Theory of Computing*, pages 182–194, New York City, NY, USA, May 25–27, 1987. ACM Press.

[Goo10] Michael T. Goodrich. Randomized shellsort: A simple oblivious sorting algorithm. In Moses Charika, editor, *21st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1262–1277, Austin, TX, USA, January 17–19, 2010. ACM-SIAM.

[Goo14] Michael T. Goodrich. Zig-zag sort: a simple deterministic data-oblivious sorting algorithm running in $O(n \log n)$ time. In David B. Shmoys, editor, *46th Annual ACM Symposium on Theory of Computing*, pages 684–693, New York, NY, USA, May 31 – June 3, 2014. ACM Press.

[Han02] Yijie Han. Deterministic sorting in $O(n \log \log n)$ time and linear space. In *Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing*, pages 602–608. ACM, New York, 2002.

[HT02] Yijie Han and Mikkel Thorup. Integer sorting in $O(n\sqrt{\log \log n})$ expected time and linear space. In *Foundations of Computer Science, 2002. Proceedings. The 43rd Annual IEEE Symposium on*, pages 135–144. IEEE, 2002.

[JL19] Shunhua Jiang and Kasper Green Larsen. A faster external memory priority queue with decreasekeys. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1331–1343, 2019.

[JLN19] Riko Jacob, Kasper Green Larsen, and Jesper Buus Nielsen. Lower bounds for oblivious data structures. In Timothy M. Chan, editor, *30th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2439–2447, San Diego, California, USA, January 6–9, 2019. ACM-SIAM.

[KS96] Vijay Kumar and Eric J Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Parallel and Distributed Processing, 1996., Eighth IEEE Symposium on*, pages 169–176. IEEE, 1996.

[KS14] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014, Part II*, volume 8874 of *Lecture Notes in Computer Science*, pages 506–525, Kaoshiung, Taiwan, R.O.C., December 7–11, 2014. Springer, Heidelberg, Germany.

[LN18] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 523–542, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

[LP98] Tom Leighton and C Greg Plaxton. Hypercubic sorting networks. *SIAM Journal on Computing*, 27(1):1–47, 1998.

[LSX19] Wei-Kai Lin, Elaine Shi, and Tiancheng Xie. Can we overcome the n log n barrier for oblivious sorting? In Timothy M. Chan, editor, *30th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2419–2438, San Diego, California, USA, January 6–9, 2019. ACM-SIAM.

[MZ14] John C. Mitchell and Joe Zimmerman. Data-oblivious data structures. In *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014), STACS 2014, March 5-8, 2014, Lyon, France*, pages 554–565, 2014.

[OGTU14] Olga Ohrimenko, Michael T. Goodrich, Roberto Tamassia, and Eli Upfal. The melbourne shuffle: Improving oblivious storage in the cloud. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *ICALP 2014: 41st International Colloquium on Automata, Languages and Programming, Part II*, volume 8573 of *Lecture Notes in Computer Science*, pages 556–567, Copenhagen, Denmark, July 8–11, 2014. Springer, Heidelberg, Germany.

[OS97] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *29th Annual ACM Symposium on Theory of Computing*, pages 294–303, El Paso, TX, USA, May 4–6, 1997. ACM Press.

[Ost90] Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *22nd Annual ACM Symposium on Theory of Computing*, pages 514–523, Baltimore, MD, USA, May 14–16, 1990. ACM Press.

[Pat90] Mike Paterson. Improved sorting networks with o(log N) depth. *Algorithmica*, 5(1):65–92, 1990.

[PPRY18] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. PanORAMa: Oblivious RAM with logarithmic overhead. In Mikkel Thorup, editor, *59th Annual Symposium on Foundations of Computer Science*, pages 871–882, Paris, France, October 7–9, 2018. IEEE Computer Society Press.

[PPY18] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. CacheShuffle: A family of oblivious shuffles. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *ICALP 2018: 45th International Colloquium on Automata, Languages and Programming*, volume 107 of *LIPIcs*, pages 161:1–161:13, Prague, Czech Republic, July 9–13, 2018. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[PR10] Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 502–519, Santa Barbara, CA, USA, August 15–19, 2010. Springer, Heidelberg, Germany.

[SCSL11] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 197–214, Seoul, South Korea, December 4–8, 2011. Springer, Heidelberg, Germany.

[Shi19] Elaine Shi. Path oblivious heap. *IACR Cryptology ePrint Archive*, 2019:274, 2019.

[SSS12] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. Towards practical oblivious RAM. In *ISOC Network and Distributed System Security Symposium – NDSS 2012*, San Diego, CA, USA, February 5–8, 2012. The Internet Society.

[SvS+13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13: 20th Conference on Computer and Communications Security*, pages 299–310, Berlin, Germany, November 4–8, 2013. ACM Press.

[Tho07] Mikkel Thorup. Equivalence between priority queues and sorting. *Journal of the ACM (JACM)*, 54(6):28, 2007.

[Tof11] Tomas Toft. Secure datastructures based on multiparty computation. Cryptology ePrint Archive, Report 2011/081, 2011. http://eprint.iacr.org/2011/081.

[WNL+14] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14: 21st Conference on Computer and Communications Security*, pages 215–226, Scottsdale, AZ, USA, November 3–7, 2014. ACM Press.