# A Secure Priority Queue; Or: On Secure Datastructures from Multiparty Computation

Tomas Toft[(✉)]

Department of CS, Aarhus University, Aarhus, Denmark
ttoft@cs.au.dk

**Abstract.** Secure multiparty computation (MPC) – computation on distributed, private inputs – has been studied for thirty years. This includes "one shot" applications as well as reactive tasks, where the exact computation is not known in advance. We extend this line of work by exploring efficient datastructures based on MPC primitives. The oblivious RAM (ORAM) provides a completeness theorem. However, implementing the ORAM-CPU using MPC-primitives is costly; current IT-secure constructions incur a poly-log overhead on computation and memory, while computationally secure constructions require MPC-evaluation of one-way functions, which introduces considerable overhead. Using ideas radically different from those in ORAM's, we propose a secure priority queue. Data accesses are *deterministic*, whereas ORAM's hide the access pattern through *randomization*. $n$ priority queue operations – insertion and deletion of the minimal element – require $O(n \log^2 n)$ invocations of the cryptographic primitives in $O(n)$ rounds. The amortized cost of each operation is low, thus demonstrating feasibility.

**Keywords:** MPC · Reactive functionalities · Datastructures

## 1 Introduction

*Secure function evaluation* considers the problem of evaluating a function $f$ on data held by $N$ parties in a distributed manner. The goal is *privacy*: The parties learn $f(x_1, \ldots, x_N)$, but do so without revealing additional information about the $x_i$. This problem has been rigorously studied in the cryptographic community since it was proposed by Yao more than thirty years ago, [Yao82]. The notion can be extended to secure multiparty computation (MPC), which considers reactive tasks: An MPC protocol may consist of multiple sequential function evaluations, where each one depends on – and potentially updates – a secret state.

Different notions of security have been proposed, e.g., protocols can provide passive or active security. In the former, all parties follow the protocol, but may

collude in an attempt to break the privacy of others. For active security, an adversary controls all corrupt parties who not only pool information, but can misbehave arbitrarily in a coordinated manner. Classic results demonstrate that any function can be computed with active security and polynomial overhead given a fully connected, synchronous network with authenticated channels (authenticated and secure channels when the adversary is computationally unbounded, i.e., the information theoretic (IT) case) [GMW87, BGW88, CCD88].

Many specialized protocols for specific, well-motivated problems have also been proposed – auctions and data mining are two popular examples. Utilizing domain specific knowledge and focusing solely on the task at hand may allow considerable efficiency gains. Though solutions may be reactive, this is rarely the case for the tasks themselves. Put differently: the topic of explicit datastructures based on MPC primitives has received surprisingly little attention.

*Contribution.* With the exception of realizations of the oblivious RAM (ORAM; see related work below), to our knowledge, we consider the first datastructure based on MPC. We construct an efficient priority queue (PQ) based on protocols providing secure storage and arithmetic over a ring, $\mathbb{Z}_M$, and inherit their security guarantees. Formally, protocols will be presented in a hybrid model providing secure black-box arithmetic; this can, e.g., be based on secret sharing.

Our PQ is inspired by the bucket heap of Brodal et al. [BFMZ04] and allows two operations: INSERT$(p, x)$ which inserts a secret element, $x$, into the queue with secret priority $p$; and GETMIN() which deletes and returns (in secret form) the element with minimal priority. Each operations use $O(\log^2 n)$ primitive operations – arithmetic and comparisons – in $O(1)$ rounds (both amortized).

The overall approach taken in this paper is to construct a datastructure where the actions performed are completely independent of the inputs. From there it is merely a matter of implementing the operations using MPC primitives. This strategy presents an immediate path to the present goal, however, it is not at all clear that it is the only one, or indeed the best one.

*Related Work.* We find three areas of related work: *incremental cryptography (IC)* of Bellare et al. [BGG94, BGG95]; *history independent (HI) datastructures* introduced by Naor and Teague building on Micciancio's oblivious datastructures [NT01, Mic97]; and the *Oblivious RAM* due to Goldreich and Ostrovky [GO96].

IC considers evaluating some cryptographic function – e.g. a digital signature – on *known*, changing data *without* recomputing that function from scratch every time. HI datastructures on the other hand focus the problem of eliminating unintentional information leakages when datastructures containing *known* data are passed on to other parties. E.g., the shape of the structure itself may reveal information on the operations performed. Both consider security and structuring data, but are fundamentally different as the data is known to some party.

The closest related concept is the ORAM, where a CPU (with $O(1)$ private memory) runs a program residing in main memory. An adversary observes the memory access pattern (but not the data/instructions retrieved) and attempts to extract information. Damgård et al. observed (as hinted by Goldreich and

Ostrovky) that implementing the CPU using MPC primitives provides a secure RAM, i.e., allows arbitrary datastructures to be used in MPC.

Oblivious RAMs hide the access pattern by randomizing it. [GO96] achieved this using a random oracle instantiated by a one-way function. In a recent result (with security issues fixed in subsequent papers), Pinkas and Reinman brought the computational overhead down to $O(\log^2 n)$ and the memory overhead down to $O(1)$ [PR10]; this was further reduced to $O(\log^2 n/\log\log n)$ by Kushilevitz et al. [KLO12]. The approach used has two drawbacks when considering datastructures in MPC: The use of one-way functions implies that the solution cannot be IT-secure. Moreover, the one-way function must be evaluated using MPC; this can be done but will most likely be costly in terms of secure computation. Independently, Ajtai [Ajt10], and Damgård et al. [DMN10] have proposed information theoretic ORAM's. Though the solutions are different, both have poly-logarithmic overhead on both (secure) computation and memory usage. Recently Lu and Ostrovsky have proposed a much more efficient ORAM solution for the two party setting [LO11]. Combining their ideas with a heap matches the theoretic complexity of the present solution. However, recent advances by Damgård et al. allow *highly efficient* IT-secure two-party arithmetic, given a preprocessing phase [DPSZ12]. These are among the fastest MPC protocols presently known, and it is unclear how to implement the shuffles needed for the ORAM in that setting without using super-linear preprocessing or online communication, i.e., without incurring an overhead.

Where the ORAM provides a completeness theorem, the present work focuses on whether different strategies may provide more efficient means of reaching specific goals. Indeed, the present approach is radically different than those used when constructing ORAM's: in stark contrast to the above, the access pattern of the PQ solution presented is *completely deterministic*, whereas *any* IT secure realization of the ORAM require at least $\log n$ bits of randomness per operation, where $n$ is the overall size of the memory, [DMN10]. This is possible since the overall "program" is known: Actions may depend on the task at hand.

Despite the common ground, ORAM's do not provide all answers regarding MPC datastructures, at least not presently. In addition to the above, using MPC to implement present ORAM solutions (other than [LO11]) incurs at least an overhead of $O(\log^2 n)$ on every read/write operation – this *equals* the cost of our PQ operations. The sequential nature of the ORAM also implies that it cannot provide round-efficient solutions. Further, both IT secure ORAM's have a poly-logarithmic overhead on memory usage, whereas the present construction does not, thus, to our knowledge the present work contains the first IT secure datastructure with constant memory overhead. Finally, there are no obvious reasons why the secure PQ could not be improved, while an IT secure ORAM with constant overhead seems less plausible.

## 2   The Basic Model of Secure Computation

We consider a setting where $N$ parties, $P_1, \ldots, P_N$, are pairwise connected by authenticated channels in a synchronous network, and focus on MPC protocols

based on linear primitives over a ring $\mathbb{Z}_M$. Secure storage and arithmetic is modeled using an ideal functionality – the arithmetic black-box (ABB), $\mathcal{F}_{\mathsf{ABB}}$ – and protocols are constructed in the $\mathcal{F}_{\mathsf{ABB}}$-hybrid model. This functionality was introduced by Damgård and Nielsen [DN03]; the benefits include abstracting away irrelevant low-level details as well as simplifying security proofs: The underlying primitives provides security for any the application, thus privacy can only be lost if we *explicitly* output a value.

## 2.1   The Arithmetic Black-Box

Reference [DN03] presents $\mathcal{F}_{\mathsf{ABB}}$ in the UC framework of Canetti [Can00] and realizes it efficiently based on Paillier encryption [Pai99]. The protocols are shown secure against an active, adaptive adversary corrupting a minority of the parties. For simplicity, we present a modified $\mathcal{F}_{\mathsf{ABB}}$ focusing on passive corruption only:

– **Input:** If party $P_i$ sends "$P_i : x \leftarrow v$" and all other send "$P_i : x \leftarrow ?$", $\mathcal{F}_{\mathsf{ABB}}$ stores $v$ under the variable name $x$,[1] and sends "$P_i : x \leftarrow ?$" to everyone.
– **Output:** If all parties send "$\mathtt{output}(x)$", then assuming that value $v$ was stored under $x$, $\mathcal{F}_{\mathsf{ABB}}$ sends "$x = v$" to everyone as well as the adversary.
– **Arithmetic:** Upon receiving "$x \leftarrow y + z$" from all parties, $\mathcal{F}_{\mathsf{ABB}}$ computes the sum of the values stored under $y$ and $z$ and stores the result as $x$. Similarly, upon receiving "$x \leftarrow y \cdot z$" from all parties, the product is stored under $x$.

Input/output can be though of as secret sharing/reconstruction, in which case linear primitives implies that addition of shares is addition of secrets; multiplication then requires interaction. Shamir sharing along with the protocols of Ben-Or et al. fit this description [Sha79, BGW88], see Appendix A, though we can equally well instantiate $\mathcal{F}_{\mathsf{ABB}}$ using homomorphic encryption, e.g., [Pai99, CDN01].

In case of active adversaries, minor alterations must be made to $\mathcal{F}_{\mathsf{ABB}}$ to ensure that it exactly captures the possible behavior. It is stressed that such change do not invalidate our construction below. Consider, e.g., the case of honest majority and guaranteed termination: Adversarial parties are allowed to abort, hence $\mathcal{F}_{\mathsf{ABB}}$ should only receive $\lfloor (N+1)/2 \rfloor$ output messages before sending "$x = v$". In a similar vein, when fairness is not ensured, the adversary receives "$x = v$" first, and can decide if the honest parties should receive it as well.

## 2.2   Complexity

As abstract primitives are used, one can merely count the number of operations performed by $\mathcal{F}_{\mathsf{ABB}}$. These correspond directly to the computation and communication of the underlying primitives. We focus on communication complexity of such operations; since linear primitives are assumed, this implies that addition (and multiplication by public values) is costless. We will not distinguish between the complexities of the remaining operations, but remark that multiplication is generally both the most used and the most costly one.

---

[1] For simplicity, consider these distinct, i.e., variables are never overwritten.

Regarding instantiations of $\mathcal{F}_{\text{ABB}}$, the basic operations are typically reasonably cheap. For passive adversaries, typically only $O(1)$ ring elements are communicated per player or pair of players. E.g., performing a passively secure multiplication of Shamir shared values can be done by having each party reshare the product of its shares (plus local computation), i.e., two field elements per pair. The dominating term of the Paillier based protocols of [DN03] – and in other actively secure constructions – is $O(N)$ *Byzantine agreements*on ring elements (e.g., encryptions) per (non-costless) operation, i.e., $O(1)$ Byzantine agreements per player. Unless a broadcast channel is assumed, such an overhead is required to guarantee robustness against actively malicious adversaries.

A second measure of complexity of protocols is the number of rounds required (the number of message exchanges). For clarity this was left out of the presentation above, however, it is easily incorporated: Assume that all operations take the same, constant number of rounds. Now, rather than receiving *one* instruction from each party, parties send *lists of independent instructions* to be performed by the functionality. Each invocation of $\mathcal{F}_{\text{ABB}}$ then refers to one round of operations, which in turn translates to one or more rounds of communication.

The straightline program notation used below improves readability, but has a drawback: The description of the protocols is detached from the actual execution in the $\mathcal{F}_{\text{ABB}}$ hybrid model. Hence, complexity analysis becomes slightly more complicated, as the description does not explicitly state which operations can be performed in parallel. Clearer descriptions easily makes up for this, though.

# 3 Extending the Arithmetic Black-Box

The secure priority queue is not constructed directly based on $\mathcal{F}_{\text{ABB}}$. We extend that functionality with additional operations. These are realized using nothing more than the basic operations of $\mathcal{F}_{\text{ABB}}$. This section can be viewed as containing preliminaries in the sense that it introduces a number of known constructions.

## 3.1   Secure Comparison

Having priorities implies some notion of order with respect to the stored elements. Further, $\mathcal{F}_{\text{ABB}}$ must allow us to compare priorities to determine which is larger. Extending the functionality with such an operation is straightforward:

- **Comparison:** Upon receiving "$x \leftarrow y \overset{?}{>} z$" from all parties, $\mathcal{F}_{\text{ABB}}$ determines if $y$ is larger than $z$, and stores the result as $x$; 1 for true and 0 for false.

As an example, consider the "integer ordering" of $\mathbb{Z}_M$-elements. For prime $M$, this can be implemented using $O(\log M)$ non-costless operations in $O(1)$ rounds, e.g. [NO07]. When $M$ is an RSA modulus – e.g., a public Paillier key – complexity is increased to $O(N \log M)$ due to more expensive sub-protocols. In specific settings other solutions may be preferable, e.g., [Tof11,LT13]. It is stressed that these are merely options; *any* secure computation and *any* ordering works.

For simplicity of the analysis, we assume that the comparison requires only a constant number of rounds, and count the number of comparison invocations separately from the basic operations due to its (in general) much higher cost. Given a specific protocol one can determine the actual cost.

## 3.2   Secure Conditional Swap

Based on the ability to compare, it is possible to perform conditional swaps: Given two values, swap them if the latter is larger than the former. This can be viewed as sorting lists of length two, and is easily constructed within the ABB by simply computing the maximal and minimal of the two.

$$max \leftarrow \left(a >^? b\right)(a - b) + b; \quad min \leftarrow a + b - max$$

These expressions easily translate to messages from parties to $\mathcal{F}_{\mathtt{ABB}}$; work is constant – $O(1)$ basic operations and a single comparison – and multiple swaps may be executed in parallel. The swap computation can be generalized to multi-element values, say pairs consisting of a priority and a data element. It is simply a question of having a well-defined comparison operator and using its output to choose between the two candidates on a single element basis.

## 3.3   Secure Merging

The main, large-scale primitive is the ability to merge sorted lists of length $\ell$ stored within $\mathcal{F}_{\mathtt{ABB}}$. This is written $\mathtt{MERGE}\,(X, Y)$, where $X$ and $Y$ refer to lists of stored values. A solution is obtained from sorting networks – sorting algorithms created directly based on conditional swaps. No branching is performed, hence they are deterministic and oblivious to the inputs, except the problem size, $\ell$.

Any sorting network can be utilized to merge, by simply viewing the whole input as a single unsorted list. However, for efficiency, we take the inner workings of Batcher's odd-even mergesort [Bat68]. The whole sorting network requires $O(\ell \log^2 \ell)$ conditional swaps, but merging alone requires only $O(\ell \log \ell)$ conditional swaps in $O(\log \ell)$ rounds, and constants are low.

A primitive for merging lists of *differing* lengths, $\ell \neq \ell'$, is also required. The shorter list is simply padded – assume that some element, $e_\infty$, which is greater than all others is reserved for this – such that they become of equal length. Now merge the lists using the above solution and remove the padding; since these elements are greater than any valid ones, all such elements are pushed to one side. The size of the padding is known, so those elements can be removed by truncating the list. Complexity is $O(\max(\ell \log \ell; \ell' \log \ell'))$ operations in $O(\max(\log \ell, \log \ell'))$ rounds. We overload $\mathtt{MERGE}\,(\cdot, \cdot)$ to avoid introducing additional notation.

We present a final, needed primitive which is highly related to merging: *merge-split*. This operation, denoted $\mathtt{MERGESPLIT}\,(X, Y)$, takes two lists as input as above. As the name suggests, the goal is to merge two lists into one, which is then split (cut into two parts whose concatenation is the sorted list). The only requirement is that lengths of the new lists must equal the lengths of the old ones.

The effect of a merge-split is that the most significant elements end up in one of the lists, while the least significant ones end up in the other. Naturally, both new lists are still sorted. Clearly this operation is equivalent to a merge, as the split merely renames variables. Hence, its complexity is the same as merging.

## 4  The Goal: A Secure Priority Queue

We are now ready to present the desired goal, an ideal functionality for a priority queue, $\mathcal{F}_{\texttt{PQ}}$. However, the data of a datastructure is not separated from the rest of the world in general and inputs to the datastructure may not originate from some party, but could be the result of previous computation. Thus, the goal is to *further extend* the arithmetic black-box with a priority queue. As with the introduction of a comparison operator, we simply list all operations needed. I.e., $\mathcal{F}_{\texttt{PQ}}$ contains the operations of the extended $\mathcal{F}_{\texttt{ABB}}$ in addition to the following:

– INSERT$(p, x)$:[2] Upon receiving "PQinsert$(p, x)$" from all parties, where $p$ and $x$ are variables, $\mathcal{F}_{\texttt{PQ}}$ stores the values associated with the pair $(p, x)$ in an internal, initially empty list, $L$. All parties then receive "PQinsert$(p, x)$".
– GETMIN(): Upon receiving "$y \leftarrow$PQgetmin()" from all parties, $\mathcal{F}_{\texttt{PQ}}$ determines and deletes from $L$ the pair with the lowest $p$-value. The corresponding $x$-value is stored as $y$, and all parties receive "$y \leftarrow$PQgetmin()" from $\mathcal{F}_{\texttt{PQ}}$.

Naturally, parties engaging in a protocol may interleave these two operations arbitrarily with other computation. This could even contain operations for other priority queues. Note, however, that $\mathcal{F}_{\texttt{PQ}}$ must treat the operations on a given PQ as atomic with respect to each other. There is a small issue with the above description: The behavior of $\mathcal{F}_{\texttt{PQ}}$ is not specified if GETMIN() is executed on an empty queue. In this case, $\mathcal{F}_{\texttt{PQ}}$ may simply discard the operation. All parties always know the exact number of elements in the queue, as they are notified whenever operations occur, hence this has no consequences.

## 5  The Secure Bucket Heap

A standard binary heap is not directly implementable using MPC primitives as one cannot traverse a tree from root to leaf by a path depending on secret data. The realization of $\mathcal{F}_{\texttt{PQ}}$ is instead based off of the bucket heap of Brodal et al. [BFMZ04], though a few significant changes are made. Jumping ahead, the original solution merges sorted lists using linear scans – we must employ Batcher's solution from Sect. 3.3. Secondly, we impose a rigid structure (with respect to the priorities) of the elements of each bucket. This actually causes the name *bucket heap* to be slightly misleading. Finally, we consider a simple problem than [BFMZ04] – the decrease-key operation has been eliminated, which implies that the actual content can be ignored.

---

[2] This is referred to as INSERT$(p)$ below; $x$, is left implicit to avoid clutter.

## 5.1 The Intuition of the Secure Bucket Heap

We stress that this section is *not*, strictly speaking, correct. However, it explains the core ideas nicely: Store a list, $D$, containing all the data in sorted order. Doing so naively makes inserts too costly, as a newly inserted element can end up anywhere. Thus, rather than inserting directly into that list, elements are placed in buffers until sufficiently many have arrived to pay for the combined cost of all insertions. More formally, the data is split into sub-lists (buckets), $D_0, D_1, D_2, \ldots$, where the elements of $D_i$ are less than those of $D_{i+1}$. The size of the $D_i$ double with each step (or level) – $|D_i| = 2^i$. In addition to this, at each level, $i$, there is a buffer, $B_i$, of the same length as the data; see Fig. 1.
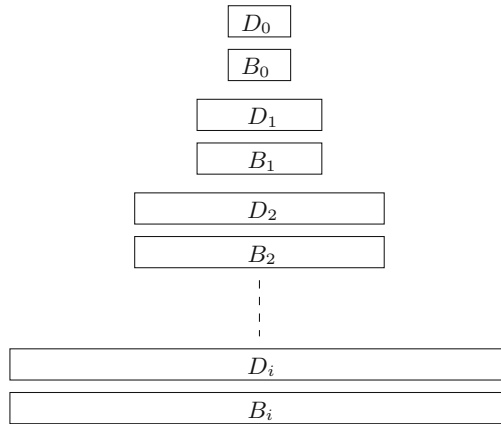


**Fig. 1.** The structure of the bucket heap

Inserting new data means placing it in the uppermost buffer, $B_0$, the intuition being, that whenever a buffer $B_i$ is full, its contents are processed. The elements that "belong at this level" are moved to $D_i$, while the rest are pushed down to $B_{i+1}$. The $D_i$ can be viewed as a sorted list of "buckets" of elements, where elements increase with each step. Thus, "belong at" means that an element is smaller than some $p \in D_i$. The minimal is obtained by returning the contents of $D_0$. Subsequent GETMIN()'s will find, $D_0$ empty, but the desired element is found in the top-most, non-empty bucket. The remainder of the content of its bucket is then placed in the buckets above.

## 5.2 Invariants

Data is stored as specified above, but with a few additional requirements. Bucket $D_i$ is either completely full or completely empty, $|D_i| \in \{0, 2^i\}$. Buffers are slightly different as the $B_i$ must contain strictly less than $2^i$ elements. They may temporarily exceed this limit – denoted that the buffer is full – at which point

---

**Protocol 1.** FLUSH$(i)$ – flushing buffer $B_i$ at level $i$

---

**Require:** Full buffer, $B_i$, at level $i$.
**Operation:** Flush $B_i$, moving the elements contained into data or subsequent buffers.
    **if** $|D_i| = 0$ and $i$ is the lowest level **then**
        $D_i \leftarrow B_i(1..2^i)$
        $B_i \leftarrow B_i((2^i + 1)..|B_i|)$
        **if** $|B_i| \geq 2^i$ **then**
5:        FLUSH$(i)$
        **end if**
    **else**
        $(D_i, B_i) \leftarrow$ MERGESPLIT $(D_i, B_i)$
        $B_{i+1} \leftarrow$ MERGE $(B_i, B_{i+1})$
10:    Set $B_i$ empty
        **if** $|B_{i+1}| \geq 2^{i+1}$ **then**
            FLUSH$(i + 1)$
        **end if**
    **end if**

---

the contents will be processed. Finally, the elements of buffer $B_i$ are greater than (have higher priority than) the elements of the higher-lying buckets, $D_j$, $j < i$. In difference to the original bucket heap, the contents of the buckets and buffers are stored sorted by priority. This is the rigid structure referred to above. Note that the concatenation of the $D_i$ can be viewed as one long, sorted list.

### 5.3   The Operations

The datastructure must be maintained using only $\mathcal{F}_{\mathsf{ABB}}$-operations. The two operations needed are the insertion of a new value and the extraction of the present minimal. The main parts of these operations are seen as Protocols 1 and 2.

The insert operation, INSERT$(p)$, is performed by placing $p$ in the top buffer, $B_0$. This fills it and it must be flushed using Protocol 1. The GETMIN() operation is realized by the (attempted) extraction the element stored in the top-level bucket. This is done by executing DELMIN $(0)$; the details are seen as Protocol 2.

### 5.4   Correctness

To show correctness, it suffices to show that the invariants hold and that these imply the desired behavior. It is clear that for the starting position – an empty priority queue – all invariants hold. All buckets are empty which is acceptable; further, there are no elements so the required ordering between elements of different buckets and buffers as well as the internal ordering are clearly satisfied.

An INSERT$(p)$ operation places $p$ in $B_0$. Note that all invariants holds except that $B_0$ is full – no relationship to other elements is required of the sole element in $B_0$. After this, the buffer is flushed. There are two possible states, as seen from the "outer" if-statement of Protocol 1: either this is the lowest level and $D_i$ is empty; *or* there is data here or below. At the bottom we simply move the

**Protocol 2.** DELMIN($i$) – return the $2^i$ smallest elements from level $i$ and below (or everything if there are fewer than $2^i$ elements)

---

**Require:** Non-empty bucket heap; all levels above the $i$'th are completely empty.

**Operation:** DELMIN($i$) – determine and return the $2^i$ minimal elements

  **if** $|D_i| = 2^i$ **then**

   $(D_i, B_i) \leftarrow$ MERGESPLIT $(D_i, B_i)$

   Return $D_i$ and set it empty

  **else if** $i$ is the lowest level **then**

5:   Return $B_i$ and set it empty

  **else**

   $B_{i+1} \leftarrow$ MERGE $(B_i, B_{i+1})$

   Set $B_i$ empty

   **if** $|B_{i+1}| \geq 2^{i+1}$ **then**

10:    FLUSH($i+1$)

   **end if**

   $\tilde{D} \leftarrow$ DELMIN $(i+1)$

   **if** $|\tilde{D}| = 2^{i+1}$ **then**

    $D_i \leftarrow \tilde{D}(2^i + 1..2^{i+1})$

15:    Return $\tilde{D}(1..2^i))$

   **else if** $|\tilde{D}| > 2^i$ **then**

    $B_i \leftarrow \tilde{D}(2^i + 1..|\tilde{D}|)$

    Return $\tilde{D}(1..2^i))$

   **else**

20:    Return $\tilde{D}$

   **end if**

  **end if**

---

$2^i$ smallest elements into the bucket (buffers are only flushed when they contain $2^i$ elements). As *all* the elements in the buffer are bigger than the elements in the buckets above, then the new relationship with all buckets hold.

Alternatively, there is data in the present bucket, $D_i$, or below. By the invariant, all elements are greater than the elements of the buckets above. Thus, performing the merge-split, line 8, does not violate invariants. This step ensures that the smallest elements of the level end in $D_i$; these are at most as big as the previous largest element of $D_i$, and must therefore be smaller than the elements of the levels below. Additionally, it is guaranteed that the elements of $D_i$ are smaller than those of $B_i$, so the latter can be pushed into the buffer below. All invariant still hold, except that $B_{i+1}$ may now have become full; if so, flush it.

The minimal element is obtained using DELMIN($0$). The intuition behind Protocol 2 is that the minimal element must come from a bucket. Only when *no* such elements exist will a buffer-element be taken, line 5. The invariant implies that the minimal element will be in the top-most, non-empty bucket *or* in a buffer above. Starting with $B_0$, buffers are flushed until a non-empty bucket is found, lines 7 and 12. Note that these buffer merges do not affect the invariant.

Once a non-empty bucket is found, it is merge-split with its buffer to ensure that it contains the $2^i$ smallest elements, not only at this level, but *overall*:

buckets and buffers above are empty, and any element in the bucket is less significant than any at a level below. The bucket is then emptied into the buckets above, filling them and leaving one element to be returned – this task is trivial as all buckets (and their concatenation) are sorted. It is easily verified that the invariants hold at this point.

If all buckets are empty, then all buffers are merged until only a single non-empty one exists (at the lowest level, $i$). Viewing $B_i$ as a sorted list, its contents may be distributed to the top buckets above, exactly as with the emptying of a bucket above, *except* that there may be "excess elements." For $|B_i| = 2^j + k$, with $k < 2^j$, the minimal element can be returned and the $j$ top-most buckets filled. This leaves the $k$ largest elements; these are placed in the buffer $B_{j+1}$. The elements of $B_i$ are easily distributed such that the invariant holds.

## 5.5   Complexity

Complexity of both INSERT($p$) and DELMIN(0) is $O(\log^2 n)$ amortized, where $n$ is the overall number of operations. This follows from a coin argument, where each coin pays for a conditional swap.

When inserting an element into $B_0$, $\Theta(\log^2 n)$ coins are placed on it. The invariant is that every element in $B_i$ has $\Theta(((\log n) - i) \log n)$ coins, which is clearly satisfied for both the initial (empty) datastructure and for the newly inserted element. These coins pay for the flushes caused by full buffers, Protocol 1.

Moving elements from the buffer to the empty bucket at the lowest level is costless. In the other case, the buffer $B_i$ is merged with bucket, $D_i$, (in the merge-split) and with buffer $B_{i+1}$ below. Both merges require $O(2^i \log 2^i)$ conditional swaps – the lists are at most a constant factor longer than $2^i$. This cost is paid using $\Theta(2^i \log n)$ coins from the elements of $B_i$. The merge-split potentially moves elements between the buffer and bucket, however, the number of elements in the buffer remains the same. The second merge moves the contents to the level below. As $B_i$ was full, it contained at least $2^i$ elements; thus, it suffices if each one moved pays $\Theta(\log n)$ coins. As the entire contents of the buffer is pushed one level down, the elements only require $\Theta(((\log n) - (i + 1)) \log n)$ to ensure that the invariant holds. Hence, the invariant holds after each element has paid the coins needed for the flush. This implies the stated complexity for INSERT($p$).

A similar argument is needed for deletion, DELMIN(0). However, rather than placing coins on the elements themselves, the deletion coins are placed on the buffers. Each operation places $\Theta(\log n)$ coins on each of the buffers, $B_i$; this requires $\Theta(\log^2 n)$ coins overall. The invariant is, that $B_i$ has $\Omega(k \log n)$ coins, where $k$ is the combined size of the empty buckets above, i.e. $k = \sum_{j=0; |D_j|=0}^{i-1} 2^j$. Whenever DELMIN($i$) is called, it implies that the buckets of all levels $j < i$ above are empty. Hence, the buffer $B_i$ has $\Omega((2^i - 1) \log n)$ coins allowing it to pay for a merge at level $i$, either with the contents of bucket $D_i$ or the buffer below.

Either way, all buckets above are filled,[3] implying that $B_i$ no longer needs coins to satisfy the invariant. Thus, earlier delete operations pay for the required merge.

Regarding round complexity, the operations require at most a constant number of merges per level, so worst-case complexity is $O(\log^2 n)$. Amortized complexity is only constant, though. Lower levels are rarely processed ($\Omega(2^i)$ operations occur between the ones "touching" level $i$) and upper levels are cheap (only $O(i)$ rounds are required to merge at level $i$); for $n$ operations, $\sum_{i=0}^{\log n} \frac{n}{2^i} i^2$ rounds are needed overall implying $O(1)$ rounds on average.

## 5.6   Security

Intuitively, security of the bucket heap follows directly from the security of $\mathcal{F}_{\text{ABB}}$: An adversary, $\mathcal{A}$, can only learn information when the ideal functionality outputs a value, i.e., when the underlying primitives explicitly reveal information. However, at no point in the present computation is an output command given by any of the honest parties. Hence, as $\mathcal{A}$ does not control what amounts to a qualified set, it cannot make $\mathcal{F}_{\text{ABB}}$ perform an output operation. By similar reasoning, it can be seen that no adversary – i.e., set of parties behaving incorrectly – can influence the computation resulting in incorrect values stored in $\mathcal{F}_{\text{ABB}}$.

The above is of course only the intuitive explanation. Formally, the view of $\mathcal{A}$ must be simulated in the $\mathcal{F}_{\text{ABB}}$-hybrid model. The required simulator, however, is trivial. It simply "executes" the realizing PQ computation, except that for every operation that the basic $\mathcal{F}_{\text{ABB}}$ should be instructed to perform, the simulator will simply play the role of $\mathcal{F}_{\text{ABB}}$ towards the corrupt players. It will receive their commands and send the messages (acknowledgments) to the corrupt players that they expect to receive. This is clearly indistinguishable from the point of view of any adversary. For each PQ operation, it simply sees a fixed set of messages, namely the ones corresponding to the secure computation implementing the operation, which it "*knows*" is being executed.

## 5.7   Hiding Whether an Operation Is Performed

A simple variation consists of conditional operations, i.e., operations based on secret bit, $b$. To achieve this, we add an additional key, $e_{-\infty}$, smaller than any real key and implement conditional INSERT($p$) as INSERT $(b \cdot (p - e_\infty) + e_\infty)$ – this inserts $p$ or $e_\infty$ depending on $b$. Similarly, we can implement a conditional GETMIN() as INSERT $(b \cdot (e_\infty - e_{-\infty}) + e_{-\infty})$; GETMIN (). If $b = 0$ $e_{-\infty}$ is inserted and immediately removed. Otherwise $e_\infty$ is inserted and the minimal removed.

Note that we no longer know the number of real keys in the PQ. This is unavoidable – a conditional GETMIN() cannot decrease the number of elements stored, while INSERT($\cdot$) must always add an element. If desired, one can keep count of the actual size, adding (subtracting) $b$ for every INSERT($\cdot$) (GETMIN()).

---

[3] The only possible exception occurs when all buckets are empty and the buffers contain too few elements to fill them all. In this case a "completely full" structure is constructed from scratch so no coins are needed.

# A   An ABB Realization

Consider a *passive* adversary and Shamir's secret sharing scheme over $\mathbb{Z}_M = \mathbb{F}_M$ for prime $M$, [Sha79]. Secret sharing allows one party to store a value privately and robustly among multiple others. If *and only if* sufficiently many agree, the value will be revealed. Input (respectively output) simply refers to secret sharing a value (respectively reconstructing a secret shared value). To implement arithmetic, note that Shamir's scheme is linear, so addition is simply addition of shares, while secure multiplication can be obtained through the protocols of Ben-Or et al. when less than $N/2$ parties are corrupt [BGW88]. It can be shown (given secure communication between all pairs of players, and assuming that all parties agree on the secure computation being performed) that these protocols realize $\mathcal{F}_{\texttt{ABB}}$ with perfect security in the presence of passive adversaries. Further, the protocols of [BGW88] even realize (a variation of) the presented $\mathcal{F}_{\texttt{ABB}}$ in the presence of active adversaries if the corruption threshold is reduced to $N/3$ – this solution guarantees termination.

# References

[Ajt10]   Ajtai, M.: Oblivious rams without cryptogrpahic assumptions. In: 42nd Annual ACM Symposium on Theory of Computing, pp. 181–190. ACM Press (2010)

[Bat68]   Batcher, K.E.: Sorting networks and their applications. In: AFIPS Spring Joint Computing Conference, pp. 307–314 (1968)

[BFMZ04] Brodal, G.S., Fagerberg, R., Meyer, U., Zeh, N.: Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. In: Hagerup, T., Katajainen, J. (eds.) SWAT 2004. LNCS, vol. 3111, pp. 480–492. Springer, Heidelberg (2004)

[BGG94]   Bellare, M., Goldreich, O., Goldwasser, S.: Incremental cryptography: the case of hashing and signing. In: Desmedt, Y.G. (ed.) CRYPTO 1994. LNCS, vol. 839, pp. 216–233. Springer, Heidelberg (1994)

[BGG95]   Bellare, M., Goldreich, O., Goldwasser, S.: Incremental cryptography and application to virus protection. In: 27th Annual ACM Symposium on Theory of Computing, pp. 45–56. ACM Press (1995)

[BGW88]   Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for noncryptographic fault-tolerant distributed computations. In: 20th Annual ACM Symposium on Theory of Computing, pp. 1–10. ACM Press (1988)

[Can00]   Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067 (2000). http://eprint.iacr.org/

[CCD88]   Chaum, D., Crépeau, C., Damgård, I.: Multiparty unconditionally secure protocols. In: 20th Annual ACM Symposium on Theory of Computing, pp. 11–19. ACM Press (1988)

[CDN01]   Cramer, R., Damgård, I.B., Nielsen, J.B.: Multiparty computation from threshold homomorphic encryption. In: Pfitzmann, B. (ed.) EUROCRYPT 2001. LNCS, vol. 2045, pp. 280–300. Springer, Heidelberg (2001)

[DMN10]   Damgård, I., Meldgaard, S., Nielsen, J.B.: Perfectly secure oblivious ram without random oracles. Cryptology ePrint Archive, Report 2010/108 (2010). http://eprint.iacr.org/. (conference version to appear at TCC 2011)

[DN03] Damgård, I.B., Nielsen, J.B.: Universally composable efficient multiparty computation from threshold homomorphic encryption. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 247–264. Springer, Heidelberg (2003)

[DPSZ12] Damgård, I., Pastro, V., Smart, N., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 643–662. Springer, Heidelberg (2012)

[GMW87] Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game. In: STOC '87: Proceedings of the Nineteenth Annual ACM Conference on Theory of Computing, pp. 218–229. ACM Press, New York (1987)

[GO96] Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. J. ACM **43**(3), 431–473 (1996)

[KLO12] Kushilevitz, E., Lu, S., Ostrovsky, R.: On the (in)security of hash-based oblivious ram and a new balancing scheme. In: Rabani, Y. (ed.) SODA, pp. 143–156. SIAM (2012)

[LO11] Lu, S., Ostrovsky, R.: Distributed oblivious ram for secure two-party computation. Cryptology ePrint Archive, Report 2011/384 (2011). http://eprint.iacr.org/

[LT13] Lipmaa, H., Toft, T.: Secure equality and greater-than tests with sublinear online complexity. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M., Peleg, D. (eds.) ICALP 2013, Part II. LNCS, vol. 7966, pp. 645–656. Springer, Heidelberg (2013)

[Mic97] Micciancio, D.: Oblivious data structures: applications to cryptography. In: STOC, pp. 456–464 (1997)

[NO07] Nishide, T., Ohta, K.: Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In: Okamoto, T., Wang, X. (eds.) PKC 2007. LNCS, vol. 4450, pp. 343–360. Springer, Heidelberg (2007)

[NT01] Naor, M., Teague, V.: Anti-persistence: history independent data structures. In: STOC, pp. 492–501 (2001)

[Pai99] Paillier, P.: Public-Key cryptosystems based on composite degree residuosity classes. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, p. 223. Springer, Heidelberg (1999)

[PR10] Pinkas, B., Reinman, T.: Oblivious RAM revisited. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 502–519. Springer, Heidelberg (2010)

[Sha79] Shamir, A.: How to share a secret. Commun. ACM **22**(11), 612–613 (1979)

[Tof11] Toft, T.: Sub-linear, secure comparison with two non-colluding parties. In: Catalano, D., Fazio, N., Gennaro, R., Nicolosi, A. (eds.) PKC 2011. LNCS, vol. 6571, pp. 174–191. Springer, Heidelberg (2011)

[Yao82] Yao, A.: Protocols for secure computations (extended abstract). In: 23th Annual Symposium on Foundations of Computer Science (FOCS '82), pp. 160–164. IEEE Computer Society Press (1982)

![Springer]