PRAC: Round-Efficient 3-Party MPC for Dynamic Data Structures

Sajin Sasy
University of Waterloo
Waterloo, ON, Canada
ssasy@uwaterloo.ca

Adithya Vadapalli
IIT Kanpur
Kanpur, Uttar Pradesh, India
avadapalli@cse.iitk.ac.in

Ian Goldberg
University of Waterloo
Waterloo, ON, Canada
iang@uwaterloo.ca

ABSTRACT

We present Private Random Access Computations (PRAC), a 3-party Secure Multi-Party Computation (MPC) framework to support random-access data structure algorithms for MPC with efficient communication in terms of rounds and bandwidth. PRAC extends the state-of-the-art DORAM Duoram with a new implementation, more flexibility in how the DORAM memory is shared, and support for Incremental and Wide DPFs. We then use these DPF extensions to achieve algorithmic improvements in three novel oblivious data structure protocols for MPC. PRAC exploits the observation that a secure protocol for an algorithm can gain efficiency if the protocol explicitly reveals information leaked by the algorithm inherently. We first present an optimized binary search protocol that reduces the bandwidth from $O(\lg^2 n)$ to $O(\lg n)$ for obliviously searching over n items. We then present an oblivious heap protocol with rounds reduced from $O(\lg n)$ to $O(\lg \lg n)$ for insertions, and bandwidth reduced from $O(\lg^2 n)$ to $O(\lg n)$ for extractions. Finally, we also present the first oblivious AVL tree protocol for MPC where no party learns the data or the structure of the AVL tree, and can support arbitrary insertions and deletions with $O(\lg n)$ rounds and bandwidth. We experimentally evaluate our protocols with realistic network settings for a wide range of memory sizes to demonstrate their efficiency. For instance, we observe our binary search protocol provides $> 27\times$ and $> 3\times$ improvements in wall-clock time and bandwidth respectively over other approaches for a memory with 2^{26} items; for the same setting our heap's extract-min protocol achieves $> 31\times$ speedup in wall-clock time and $> 13\times$ reduction in bandwidth.

KEYWORDS

Oblivious data structures, Secure multi-party computation, Oblivious RAMs, Distributed privacy

1 INTRODUCTION

The modern Internet is riddled with problems. As more activities become digital, users share more personal information online, and consequently, the threats of cyber-attacks and data breaches loom large. One way of protecting data is building systems so no unauthorized entity can access the data they are not supposed to; building firewalls is one such direction, but the prevalence of successful attacks shows that it is not sufficiently effective. A more robust approach to tackling this problem is making systems that are *databreach proof*; i.e., even if an unauthorized entity gets access to data they are not supposed to, they can make no sense of it. There are three main ways to build databreach-proof systems: (i) trusted hardware, (ii) fully homomorphic encryption, and (iii) distributed trust. In this paper, we focus on the distributed-trust setting, where operations on data are performed via secure Multi-Party Computation (MPC). MPC has been studied for several decades. However, many computations one might wish to perform with MPC use dynamic data structures [9, 23]. Such computations require oblivious random access memory (ORAM), where the *indices* at which memory is read or written must themselves be kept private. These computations remain challenging to do efficiently in MPC, especially in deployments with typical Internet latencies and throughputs. This paper presents PRAC (Private Random Access Computations), a general MPC framework designed to support random-access data structure algorithms with efficient communication costs, both in terms of rounds and bandwidth used.

PRAC provides both generic optimizations, and ones specific to particular data structures and algorithms. PRAC exploits the fact that if an algorithm innately leaks some information, a cryptographic protocol implementing such an algorithm can be made more efficient if the protocol explicitly reveals this leakage. Broadly speaking, data structures can be (i) static, where no insertions or deletions can be made to the data, (ii) restrictively dynamic, where some specific types of insertions or deletions can be made, and (iii) fully dynamic, where arbitrary insertions or deletions can be performed. In this work, we provide an example of each of these data structure types.

We start with a simple example, by presenting a novel oblivious binary search protocol for static data. Existing works that design binary search protocols for MPC stem from Distributed ORAMs (DORAMs) [8, 12, 22, 33]. The high-level idea is to initialize a DORAM with all the items to be searched already sorted, and then perform a logarithmic number of DORAM accesses to perform a binary search in the straightforward manner. We observe that the binary search protocol inherently leaks the lists of indices potentially accessed in each ORAM operation, and so by using the Incremental Distributed Point Function [5] cryptographic primitive, the algorithm can explicitly leak this information and achieve performance gains.

Next, we present oblivious heaps, which fall in the category of "restrictively dynamic" data structures. Heaps are "restrictive" in that, while any item can be inserted into the heap, only the minimal element of the heap can be deleted. The "heapification" process requires DORAM update operations. We observe that the indices at which these updates happen are related. Our protocols explicitly reveal this relationship and reduce the computation and communication costs by a factor of 3. Heaps also leverage our optimized binary search protocol to reduce the communication rounds of an insert operation from $O(\lg n)$ to $O(\lg \lg n)$.

Finally, we present oblivious AVL trees, an example of a fully dynamic data structure supporting arbitrary insertions and deletions. AVL trees differ from the binary search and heap in how the data is laid out to represent them; those protocols' memory layout have an *implicit structure*. In other words, the memory indices dictate the tree structure in both binary search and heaps. In AVL trees, the position of items in the tree is dynamic, as items change position when the tree is rebalanced. This rebalancing operation (which must itself be oblivious) would have prohibitive cost if the tree structure were determined by the memory indices. Therefore, AVL trees require pointers to maintain parent-child relationships and correctly traverse the tree structure; we refer to such protocols as *explicit-structure* protocols. These protocols take advantage of PRAC's extremely efficient oblivious RAM operations to keep the costs of oblivious searches, insertions, and deletions (including rebalancing) low.

The contributions of our work are:

1. We extend the state-of-the-art DORAM Duoram [33] with *implementation* improvements (Section 3) and support for Incremental DPFs (IDPFs) and Wide DPFs (WDPFs). We use the latter for *algorithmic* improvements in three novel oblivious data structure protocols:
   (a) Our **binary search** protocol (Section 4) uses incremental DPFs to reduce bandwidth, as the $\lg n$ DPFs required can be replaced with 1 IDPF.
   (b) Our **heap** protocols (Section 5) reduce rounds of insertion from $O(\lg n)$ to $O(\lg \lg n)$ by leveraging our binary search protocol. Our extract-min protocol reduces bandwidth from $O(\lg^2 n)$ to $O(\lg n)$ with WDPFs.
   (c) Finally, we present the first **AVL tree** (Section 6) construction in a distributed trust setting that enables oblivious insertion and deletion of data items.
2. To implement the above, we present PRAC, a three-party MPC framework to implement oblivious data structures with low communication costs (both rounds and bandwidth). We use PRAC to provide an open-source implementation of all the above contributions. We experimentally compare our protocols against other DORAM-based binary search and heap protocols (Section 8) with realistic network settings to demonstrate their efficiency. For instance, with a memory of 2^{26} items, we observe: (i) $> 18\times$ and $> 3\times$ improvements in wall-clock time and bandwidth respectively for our binary search protocols, and (ii) $> 16\times$ and $> 7\times$ improvements in wall-clock time and bandwidth for our heap extract-min protocol.

Applications to privacy enhancing technologies. Data structures such as heaps and AVL trees can be crucial in designing various privacy enhancing technologies. For instance, priority queues (which can be realized using heaps) enable the implementation of sampling algorithms [28], widely utilized in differential privacy. Priority queues are also used by Mazloom et al. [23] and Cartlidge et al. [9] in implementing dark pools. Similarly, Nearest Neighbor Search is fundamental in many machine learning applications, including targeted advertising [27], pattern recognition [24], recommendation systems [1], and DNA sequencing. One way to implement nearest neighbor search is using priority queues, which in turn can be implemented using heaps. Therefore, a privacy-preserving heap implementation would enable the implementation of the aforementioned applications while maintaining privacy.

Maintaining efficient search and manipulation operations in databases is crucial for handling large volumes of data effectively. AVL trees enable quick search for specific records and provide efficient *worst-case* insertion and deletion of data, making them valuable in data-oblivious MPC settings.

2 BACKGROUND

2.1 Secret Sharing

A method by which two or more parties distribute a secret among themselves is known as secret sharing. Secret sharing is a critical component in establishing distributed trust. Definition 2.1 formalizes the notion of secret sharing.

Definition 2.1. An (n, t) secret sharing scheme allows a *dealer* to distribute a secret among n parties such that only a subset of size at least t can reconstruct the secret and any subset of size less than t can learn nothing about the secret.

We only use $(2, 2)$ secret sharing in this work and omit "$(2, 2)$" henceforth. PRAC uses three types of secret sharing:, (i) additive shares, modulo 2^r unless otherwise specified, (ii) r-bit XOR shares, and (iii) single-bit boolean shares. (We typically use $r = 64$.) When two parties additively secret-share an integer R, we denote it as R^{AS}, and the two shares add to $\mathsf{R} \bmod 2^r$. Similarly, when two parties XOR secret-share an integer R, we denote it as R^{XS}, and the two shares XOR to R. We use the notation b^{BS} to denote the boolean sharing of a bit b, where the two (single-bit) shares XOR to b. The shares held by parties P_0 and P_1 are denoted as $\mathsf{R}_0^{\mathrm{GS}}$ and $\mathsf{R}_1^{\mathrm{GS}}$ respectively, where $\mathrm{G} \in \{\mathrm{A}, \mathrm{X}, \mathrm{B}\}$. If G is B, then R is a single bit; otherwise, it is (typically) an r-bit word.

2.2 Distributed Point Functions

A 1-hot vector is a vector with size n that has $n - 1$ zeros and exactly one non-zero value. A point function is a function over a domain that evaluates to 0 at every point in the domain (of the point function) except at one unique point where it evaluates to a non-zero value. The outputs of a point function over its entire domain result in a 1-hot vector.

Secret shares of *the outputs of* point functions have been used in many cryptographic constructions, ranging from private information retrieval protocols (to select elements from vectors or matrices obliviously) [3, 15, 17, 18] through ring signature schemes (where they choose public verification keys from the set) [25], to secure MPC (where they allow random read and write accesses into arrays) [12, 32, 33]. Gilboa and Ishai introduced Distributed Point Functions (DPFs) [14], with further improvements by Boyle et al. [6, 7]. DPFs are a concise way to share a point function among two or more parties. In this paper, we only concern ourselves with DPFs that share a point function between exactly two parties. Definition 2.2 (a restatement of Vadapalli et al. [34, Def 4]) formally defines $(2, 2)$-DPFs.

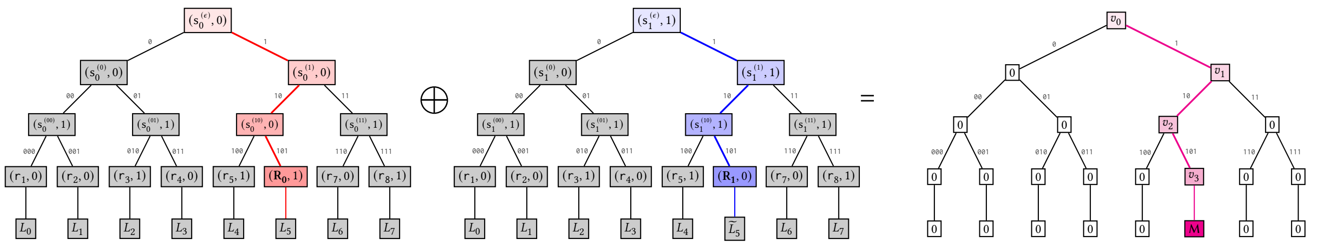


Figure 1: Distributed Point Functions. The two trees on the left hand side, are the DPF trees held by P_0 and P_1 respectively. Each node in the tree is a (label, advice bit) tuple $(s^{(x)}, b) \in \{0,1\}^{\lambda} \times \{0,1\}$. The two children of $(s^{(x)}, b)$ are $(s^{(x\|d)}, b') \leftarrow G_d(s^{(x)}, 0) \oplus (b \times cw)$ for $d \in \{0,1\}$, where G_0 and G_1 are the two halves of a length-doubling PRG, and cw is the correction word. The tuples $(r_i, b) \in \{0,1\}^{\lambda} \times \{0,1\}$ are nodes that XOR to 0 before the final correction word is applied. The tuple (R_b, b) is node that XORs to a random value before the final correction word is applied. $L_i \in \{0,1\}^{\lambda \times w}$ are the leaf nodes. Two corresponding nodes in the two trees having the same color are equal; for instance, $s_0^{(0)} = s_1^{(0)}$. If the DPF is a "wide DPF" of width w, leaves are $\in \{0,1\}^{\lambda \times w}$.

Definition 2.2. A $(2,2)$-*distributed point function*, or $(2,2)$-*DPF*, is a pair of PPT algorithms $(\textsc{Gen}, \textsc{Eval})$ defining secret-shared representations of point functions (with domain $\mathtt{i} \in [0,n)$); that is, given (i) a security parameter $\lambda \in \mathbb{N}$, (ii) a target point $\mathtt{i}^*$, and (iii) a target value M, we have

1. **Correctness:** If $(\mathsf{k}_0, \mathsf{k}_1) \leftarrow \textsc{Gen}(1^{\lambda}, \mathtt{i}^*, \mathsf{M})$, then, for all $\mathtt{i} \in [0,n)$,
$$\textsc{Eval}(\mathsf{k}_0, \mathtt{i}) \oplus \textsc{Eval}(\mathsf{k}_1, \mathtt{i}) = \begin{cases} \mathsf{M} & \text{if } \mathtt{i} = \mathtt{i}^*, \text{and} \\ 0 & \text{otherwise.} \end{cases}$$
2. **Simulatability:** There exists a PPT simulator $\mathcal{S}$ such that, for a tuple of target index and a target value, $(\mathtt{i}^*, \mathsf{M})$, and bit $b \in \{0,1\}$, the distribution ensembles $\{\mathcal{S}(1^{\lambda}, b)\}_{\lambda \in \mathbb{N}}$ and $\{\mathsf{k}_b \mid (\mathsf{k}_0, \mathsf{k}_1) \leftarrow \textsc{Gen}(1^{\lambda}, \mathtt{i}^*, \mathsf{M})\}_{\lambda \in \mathbb{N}}$ are computationally indistinguishable.

The k_b output by $\textsc{Gen}$ are called $(2,2)$-*DPF keys*.

2.2.1 *DPF constuction.* For a detailed description of the construction of DPFs, we refer the reader to the original DPF construction [7] and some of the subsequent works [5, 32–34] that used DPFs. In brief, a DPF is represented as two complete binary trees of height $h = \lg n$; each party will learn one of the trees. Each node of each tree contains a λ-bit *label* and a single *advice bit* (called in some works the *flag bit*). The construction begins with the roots of the two trees having random λ-bit labels, and the root advice bit in tree b being b itself ($b \in \{0,1\}$).

The invariant is that for each $0 \le j \le h$, the corresponding nodes in level j of the two trees will be identical, except for one, which will have different labels and different advice bits in the two trees. This invariant is maintained through the use of h *correction words*, one for each non-leaf level. The labels and advice bits of the two children of a node are then a deterministic pseudo-random function (typically based on AES) of the label on the node, XORed with the *correction word* for that level if the node's advice bit is 1. Importantly, the nodes that are the same in the two trees will always have each of their children the same in the two trees. Further, the correction word is chosen exactly so that for the one node in each (non-leaf) level that is different in the two trees, exactly one of its children will be the same in the two trees, and the other will have different labels and advice bits. Which child will be the same and which will be different in level j will depend on the jth bit of the DPF target index $\mathtt{i}^*$.

Then by the invariant, the leaf layer (before the application of the final correction word) of the two trees will be the same, except for one node, which will have different (pseudo-random) labels and different advice bits. Then there is a *final correction word* that is XORed into the label of each leaf with advice bit equal to 1. In this way, the final correction word can be set so that the XOR of the two unequal leaf nodes is the desired target value M.

The DPF keys $(\mathsf{k}_0, \mathsf{k}_1)$ output by $\textsc{Gen}$ are then the random root labels (different in the two output keys), the h correction words, and the final correction word (the latter two the same in the two output keys). Computing $\textsc{Eval}(\mathsf{k}_b, \mathtt{i})$ involves evaluating the path from the root to leaf $\mathtt{i}$ (including all the correction words), and outputting the label on that leaf. Figure 1 illustrates the process.

2.2.2 *Incremental DPFs.* The tree-based DPF construction we use has the property that the XOR of the two parties' trees is zero everywhere except on a single path from the root to the target leaf index $\mathtt{i}^*$. Boneh et al. [5] observe that this construction can be leveraged to yield what they call an *incremental* DPF, or IDPF. In an IDPF, the parties can evaluate their DPFs at bitstrings $\mathtt{i}$ of any length $1 \le j \le \lg n$. The parties will produce different outputs exactly when $\mathtt{i}$ is a prefix of the target index $\mathtt{i}^*$, and the same output otherwise.

The cost to construct an IDPF is about 50% more than a regular DPF, but you end up with effectively $\lg n$ DPFs (one each of sizes $2, 4, 8, 16, \ldots, n$), whose target indices are the j-bit prefixes of a single $\mathtt{i}^*$.

2.2.3 *Wide DPFs.* Wide DPFs (WDPFs) are DPFs with leaves wider than internal nodes. In particular, for a WDPF of width, w, the leaf nodes are w times longer than the internal nodes. The WDPF construction differs from the traditional DPF construction in that the final layer uses a "length-w-stretching" PRG before applying the final correction word (which is also w times longer than the internal correction words). Note that creating a wide DPF with width w requires $n \cdot (w-1)$ more AES evaluations than a regular DPF. For instance, in Figure 1, if the leaves L_i are w times longer than the internal nodes, then we say that it is a width-w DPF. We will see in Section 5.2 that WDPFs are particularly useful to efficiently do reads and updates of related indices in a DORAM.

2.3 Secure Multi-party Computation

Secure Multi-Party Computation (MPC) is a cryptographic primitive that allows parties holding some secret inputs to compute functions on those secret inputs while revealing no other information. A *3-PC* comprises three parties where some or all the three parties have secret inputs, and all three parties are interactive in the protocol. A *(2+1)-PC* is a three-party protocol where one of the parties does not hold any secret and only sends correlated randomness to the other two parties to facilitate their MPC. (2+1)-PC is also called server-aided 2-PC.

Evaluating the cost of an MPC protocol. There are three main parameters that are used to evaluate the cost of an MPC protocol, namely, (i) local computation cost, (ii) bandwidth cost (the amount of data that needs to be sent to different parties), and (iii) round cost (the number of sequential messages sent).[1]

Secure MPC to generate DPFs. Doerner and shelat [12] introduced a technique to generate DPFs, which has been used in the DORAM literature [12, 33]. The technique involves computing parties performing PRG evaluations outside MPC locally. The cost is (i) $O(n)$ computational cost, (ii) $O(\lg n)$ bandwidth, and (iii) $O(\lg n)$ sequential messages. We use Doerner and shelat's DPF generation algorithm.

2.4 Distributed Oblivious RAMs (DORAMs)

Oblivious RAM (ORAM) [16] is a cryptographic primitive that enables a client to outsource data storage to an untrusted server. The untrusted server is oblivious to memory contents and access patterns, while the client has full visibility into them. Distributed ORAM (DORAM) introduced by Lu and Ostrovsky [22] is a variant of the traditional ORAM, with no distinct notion of a client and a server. Instead, the computing parties act as both clients and servers. As such, *no party* knows the contents of the DORAM or the access patterns. Doerner and shelat, with Floram [12], identify bandwidth and round complexity as the bottleneck in DORAM settings, rather than local computation. Floram uses different memory layouts for reading and writing, and incurs an amortized $O(\sqrt{n})$ cost to switch between them. Vadapalli et al. took this a step forward and presented Duoram [33]. Duoram stores the memory as secret shares for both read and write operations, thus avoiding the amortized $O(\sqrt{n})$ cost incurred in Floram. Additionally, Duoram moves most of the expensive work to a preprocessing phase and only requires constant rounds and bandwidth in the online phase.

For each read and update operation, Duoram requires the generation of three DPFs with a size of n, at the index $\mathtt{i}^*$. However, it is worth noting that these DPFs can be generated in advance during a *preprocessing phase* at a random location $\mathtt{ri}$. In the online phase, the target index of these DPFs can be moved to $\mathtt{i}^*$ by exchanging a single word over the network. Duoram requires a constant number of rounds and a constant bandwidth during the online phase. In the preprocessing phase, the protocol incurs a bandwidth cost of $O(\lg n)$, and $O(\lg n)$ rounds, to generate a DPF triple (three different DPFs with the same target index and value) for an ORAM operation. Note that the number of rounds (unlike the bandwidth) is independent of the number of DPFs created, since they can all be done in parallel. Duoram incurs a linear local computation cost in both the preprocessing and online phases. Of course, scaling local computation is much easier than reducing latency or increasing bandwidth between the (non-colluding) parties. Therefore, it is typically preferable to have a DORAM with a lower communication cost, particularly in terms of the number of rounds, than a lower computation cost but a higher number of rounds.

3 PRAC

PRAC has three servers P_0, P_1, and P_2. P_0 and P_1 hold the shares of the data; P_2 does not hold any input. There are two main phases for all PRAC protocols: (i) the preprocessing phase, and (ii) the online phase. In the preprocessing phase, P_2 sends different types of correlated randomness (for instance, AND and Multiplicative Triples [33]) to parties P_0 and P_1 to facilitate the MPC. The DPF generation (discussed in Section 2.3) also occurs in the preprocessing phase in which P_2 sends correlated randomness to do the MPC to produce the DPFs.

PRAC uses a DORAM to perform its oblivious memory accesses. We denote access to a (for example) XOR-shared index $\mathtt{i}$ of a DORAM $\mathbf{D}$ as $\mathbf{D}[\mathtt{i}^{\mathtt{XS}}]$. This paper uses Duoram [33] as the underlying DORAM. The choice of picking Duoram as our underlying DORAM is because PRAC protocols require several DORAMs to be initialized, and the initialization cost in Duoram is almost zero (each party initializes local memory to 0). In the most efficient instantiation of Duoram, P_2 not only sends correlated randomness to P_0 and P_1 but also actively participates in the DORAM read and update protocols in the online phase [33]; therefore, PRAC is a 3-party protocol.[2]

As compared to previous work, PRAC includes both *algorithmic* improvements to MPC protocols to reduce the number and complexity of MPC operations required to implement them (described in the following sections), as well as *implementation* improvements to substantially lower the computation and communication costs of executing those operations (as we will see in Section 8). For example, as PRAC aims to minimize the number of rounds required by its oblivious protocols, all computations are implemented as coroutines. In this way, each party can easily perform multiple subcomputations with minimal latency: each subcomputation will run, one at a time, until it emits a message to send. When all subcomputations have emitted a message (or terminated), the batch of messages is flushed. Each non-terminated subcomputation will then receive its response and continue, with only a single message latency having been incurred. Thus, PRAC computations can be written modularly, with each interactive MPC operation unaware of any other operations happening simultaneously, but the resulting messages are automatically optimally batched to minimize latency. In addition, PRAC also improves upon the original Duoram implementation. The original Duoram protocol stores the memory and

[1]A protocol uses m sequential messages if the time spent on Internet latency is m times the one-way latency.

[2]If we were to replace Duoram with a $(2+1)$-party or a 2-party DORAM, all the PRAC protocols would also involve the same number of parties. In other words, the number parties in the DORAM determine the number of parties required in PRAC.

the indices into which memory access must be performed as additive shares. PRAC extends the Duoram construction (and provides an entirely fresh implementation) to support XOR- or additive-shared indices, and XOR- or additive-shared memory (or even a memory whose elements are structures of some XOR-shared and some additive-shared fields), in any combination. PRAC works in a semi-honest model. The security assumptions are (i) Existence of PRGs (for the construction of DPFs), (ii) Existence of a secure communication channel, and (iii) None of the three parties, P_0, P_1, and P_2 collude with each other.

3.1 MPC Operations used in PRAC

All of the non-DORAM MPC operations in PRAC we list below involve only P_0 and P_1 in the online phase. For each operation, we list the bandwidth and number of rounds in the online phase. In the preprocessing phase, P_2 sends all the correlated randomness required for arbitrarily many of these operations to P_0 and P_1 in a single message to each player.

Definition 3.1 (MPC AND). P_0 and P_1 hold boolean shares of $x \in \{0,1\}$ and $y \in \{0,1\}$, and wish to jointly compute boolean shares of the AND of x and y. We denote the operation as $z^{\text{BS}} \leftarrow x^{\text{BS}} \wedge y^{\text{BS}}$. MPC AND requires one word being sent in one round in the online phase.

Definition 3.2 (MPC Flag-Word (FW) Multiplication). P_0 and P_1 hold additive shares of $y \in \mathbb{Z}_{2^r}$ and boolean shares of $f \in \{0,1\}$. The two parties compute additive shares of $z \in \mathbb{Z}_{2^r}$, such that $z = f \cdot y$. We denote this as $z^{\text{AS}} \leftarrow f^{\text{BS}} \cdot y^{\text{AS}}$. MPC FW-multiplication requires two words being sent in one round in the online phase.

Definition 3.3 (Oblivious Swap). P_0 and P_1 hold (additive or XOR) shares of both x and y. They also hold boolean shares of a bit b. After the Oblivious Swap protocol, denoted as $\mathsf{Oswap}(x^{\text{GS}}, y^{\text{GS}}, b^{\text{BS}})$ If $\mathsf{b} = 0$, the two parties' shares of x and y are replaced by fresh sharings of x and y respectively. If $b = 1$, they are replaced by fresh sharings of y and x respectively. Oswap requires two words being sent in one round in the online phase.

Definition 3.4 (Oblivious Select). P_0 and P_1 hold (additive or XOR) shares of both x and y. Oblivious Select, denoted as Oselect, is defined as: $z^{\text{GS}} \leftarrow \mathsf{Oselect}(x^{\text{GS}}, y^{\text{GS}}, \mathsf{b}^{\text{BS}})$. If $\mathsf{b} = 1$, then $z = y$ and if $\mathsf{b} = 0$, then $z = x$. Oselect requires one word being sent in one round in the online phase.

Definition 3.5 (Oblivious Compare). P_0 and P_1 hold additive shares of both x and y. The comparison protocol[3] outputs boolean shares of the possible results of the comparison. We denote the compare protocol as $(\mathtt{lt}^{\text{BS}}, \mathsf{eq}^{\text{BS}}, \mathtt{gt}^{\text{BS}}) \leftarrow \mathsf{Ocompare}(x^{\text{AS}}, y^{\text{AS}})$. The property that holds (i) if $x < y$, then $\mathtt{lt} = 1, \mathsf{eq} = 0, \mathtt{gt} = 0$, (ii) if $x > y$, then $\mathtt{lt} = 0, \mathsf{eq} = 0, \mathtt{gt} = 1$, and (iii) if $x = y$, then $\mathtt{lt} = 0, \mathsf{eq} = 1, \mathtt{gt} = 0$. Ocompare requires one word being sent in one round in the online phase.

4 OBLIVIOUS BINARY SEARCH

As a warmup (and because we use this functionality in our heap implementation later), we will look at PRAC's two implementations of binary search: a *basic* version that uses DORAM in the obvious way, and an *optimized* version, which uses PRAC's improved MPC operations (in this case, IDPF-based related reads). The former will be used for head-to-head comparisons with previous work to show our *implementation* improvements, while the latter will show the benefits specifically due to our *algorithmic* improvements.

Memory layout. For simplicity, assume the memory to be searched is a power of 2 in size; that is, there are $n = 2^h$ items. (PRAC handles non-power-of-2 sizes by virtually extending the memory to the next power of 2, but in a manner that only actually consumes a small constant amount of actual memory.) Binary search operates over an additive-shared DORAM $\mathbf{D}^{\text{AS}}$, which is required to be sorted. (PRAC also provides an implementation of bitonic sort, which will obliviously sort n items with $\lceil \lg n \rceil (\lceil \lg n \rceil + 1)$ rounds.) The DORAM is laid out linearly. In other words, P_0 and P_1 hold additive DORAM shares $\mathbf{D}_0^{\text{AS}}$ and $\mathbf{D}_1^{\text{AS}}$ respectively, such that, for any $\mathtt{i}, \mathtt{j}$ with $\mathtt{i} < \mathtt{j}$, we have $(\mathbf{D}_0^{\text{AS}}[\mathtt{i}] + \mathbf{D}_1^{\text{AS}}[\mathtt{i}] \bmod 2^r) \leq (\mathbf{D}_0^{\text{AS}}[\mathtt{j}] + \mathbf{D}_1^{\text{AS}}[\mathtt{j}] \bmod 2^r)$. Their goal is to search for an item M in $\mathbf{D}$, for which they hold shares M^{AS}. Our protocol returns shares of the smallest memory index containing a value at least M. (This is the index at which one would insert the value M to maintain the sorted order, which will be important in Section 5.2.)

4.1 Basic Oblivious Binary Search Protocol

We denote by $\mathtt{CurInd}$, the *current pointer* of the binary search, and $\mathtt{CurVal}^{\text{AS}} = \mathbf{D}^{\text{AS}}[\mathtt{CurInd}^{\text{AS}}]$. Suppose that we are obliviously searching for M. We start at depth $\mathsf{d} = h$, and first set the $\mathtt{CurInd}$ to the middle of the memory, i.e., index $2^{h-1} - 1$. The invariant is that the result we are looking for is in a range of width 2^{d}, and $\mathtt{CurInd}$ is the rightmost element of the left half of that range. We begin by an oblivious comparison of M and $\mathbf{D}[\mathtt{CurInd}]$. If $\mathbf{D}[\mathtt{CurInd}] \geq \mathsf{M}$, then the desired result index is at $\mathtt{CurInd}$ or to the left, and therefore, we subtract $2^{\mathsf{d}-2}$ from the current pointer. Otherwise, the desired result index is to the right, and therefore, we add $2^{\mathsf{d}-2}$ to the current pointer. If $(\mathtt{lt}, \mathtt{eq}, \mathtt{gt}) \leftarrow \mathsf{Compare}(\mathsf{M}, \mathtt{CurVal})$, then the above operations can be done obliviously with $\mathtt{CurInd} \leftarrow \mathtt{CurInd} - 2^{\mathsf{d}-2} + \underbrace{\mathtt{lt} \cdot 2^{\mathsf{d}-1}}_{\text{FW-Mult}}$. We then decrement d and loop until $\mathsf{d} = 0$. Note that the basic binary search protocol requires $\lg n$ DORAM read operations, thus requiring $\lg n$ DPFs. Figure 2 illustrates this protocol. The detailed protocol description appears in Appendix A.

We make a few observations about the basic binary search protocol. The DORAM calls in the basic binary search completely hide the indices that are being accessed. This hides more information than required; since we are doing a binary search, it is public knowledge that after an access is made at index $\mathtt{CurInd}$, the next access would be made at index either $\mathtt{CurInd} - 2^{\mathsf{d}-2}$ or $\mathtt{CurInd} + 2^{\mathsf{d}-2}$. Therefore, using a special-purpose DORAM specifically for such queries could be more efficient than a general-purpose DORAM. In the next section, we will explicitly reveal this information to our DORAM calls in exchange for efficiency.

4.2 Optimized Binary Search

This section details PRAC's innovative method for binary search, which reduces the number of DPFs required from $\lg n$ to exactly

[3]PRAC uses the oblivious compare due to Storrier, Vadapalli, Lyons, and Henry [30].

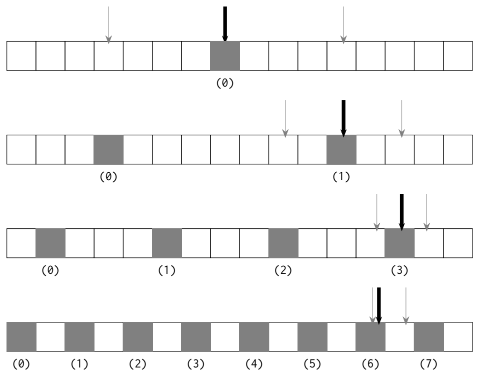


Figure 2: The binary search protocol. The shaded elements are all the *accessible* items (the items the binary search protocol could possibly access) at that depth. The thick arrow represents the current pointer. The two gray arrows are the options the current pointer has to move to the next level. The pointers, memory contents, and the item to search for are all secret shared and not known to any party.

one IDPF to do the $\lg n$ ORAM operations. Observe that a binary search on a memory of size n requires $\lg n$ read operations. Next, observe that the number of items that could possibly be read in each of these $\lg n$ read operations are $1, 2, 4, \ldots, n/2$ respectively as seen in Figure 2. We call these the *accessible items* for that particular read operation, denoted as $\mathbf{D}^{(1)}, \ldots, \mathbf{D}^{(\lg n)}$ respectively. The final observation about binary search is that, if we have made access into index $\mathtt{i}$ of memory $\mathbf{D}^{(i)}$, we would have made access into index $\lfloor \mathtt{i}/2 \rfloor$ into memory $\mathbf{D}^{(i-1)}$. In other words, if the binary representation of the index accessed is $\mathbf{b}$, then in the next iteration, the binary representation of the index that will be accessed by either $\mathbf{b}\|0$ or $\mathbf{b}\|1$, which is the structure of an IDPF.

From this observation, at each depth, the index (within the accessible elements for that level) that we access is the previous index appended with the bit $\mathtt{lt}$. Accessing indices with prefixes being incrementally appended with a bit in this way, can be done with a single incremental DPF. A formal protocol describing the IDPF-based binary search appears in Appendix A. Note that, unlike the simple binary search protocol, the IDPF-based binary search protocol does not require any Flag-Word MPC multiplications.

We remark here that a recent work by Blaton and Yuan [4] presented a similar optimization for a generic DORAM-based binary search where the read operations can be done on $\lg(n/2)$ ORAMs of sizes $2, 4, \ldots, n/2$, respectively. In contrast, our optimization exploits the observation that IDPFs produce the shares of the standard basis vectors that are required to do precisely those $\lg(n/2)$ operations. This allows us to do a binary search with exactly one IDPF producing $\lg(n/2)$ shares of the required standard basis vectors, each operating on subsets $\mathbf{D}^{(1)}, \ldots, \mathbf{D}^{(\lg n)}$ of the underlying DORAM, thus avoiding the cost of initializing and working on $\lg(n/2)$ different DORAMs.

PRAC's key advantage here is in recognizing that it is public information that the sequence of indices that will be accessed will be prefixes each of the next. We can therefore gain efficiency by *explicitly* leaking that information through the use of a single IDPF, as opposed to performing logarithmically many *independent* oblivious memory accesses.

5 OBLIVIOUS HEAPS

In this section we describe our oblivious heap data structure. In particular, PRAC supports the HeapInsert and ExtractMin operations.

Memory layout. The root of the heap lies in $\mathbf{D}[1]$. The left child of a node at $\mathbf{D}[\mathtt{i}]$ is $\mathbf{D}[2 \cdot \mathtt{i}]$, and the right child is $\mathbf{D}[2 \cdot \mathtt{i} + 1]$. The memory at index 0 holds no value. A heap can be initialized as empty, and have elements added to it one at a time using HeapInsert, or it can be initialized with pre-existing data by obliviously sorting the data (as a sorted array is automatically a heap).

The main heap algorithms. Heaps have two main protocols (i) ExtractMin, and (ii) HeapInsert. The ExtractMin algorithm returns the smallest element in a heap, which will be at the root, and replaces the root with the last element in the heap. The heap property no longer holding true, it runs a *heapify* algorithm to restore the heap property. The HeapInsert protocol adds a new element into the heap at the next available memory location, and then restores the heap property.

5.1 Basic Oblivious Heap Protocols

Basic Oblivious HeapInsert. Consider a heap with n items. Suppose we want to insert a new item M into the heap. We first assign the item M to the $(n+1)^{\text{th}}$ position in the memory (not a DORAM operation). Next, we need to restore the heap property. We obliviously compare M with its parent. If M is smaller than the parent, we swap them. This process is continued up to the root of the heap, at which point the heap property is restored, and M is in its correct position. Therefore, the basic oblivious HeapInsert protocol requires one oblivious comparison and swap at each level, without any DORAM operations at all. A formal description of the basic HeapInsert protocol appears in Appendix B

Basic Oblivious ExtractMin. The ExtractMin protocol begins with saving a copy of the root (i.e., the item situated at index 1) and replacing it with the last leaf node (i.e., the item at $\mathbf{D}[n]$) and decrementing n. After that, we have to restore the heap property, which is done by the Heapify protocol. The Heapify algorithm begins by comparing the new root with the smaller child. If the root is larger, then we swap the root with the smaller child. This process is repeated down the path following the smaller child. For some random index $\mathtt{i}$, suppose that (i) $\mathbf{D}[\mathtt{i}] = x$, (ii) $\mathbf{D}[2 \cdot \mathtt{i}] = y$, and (iii) $\mathbf{D}[2 \cdot \mathtt{i} + 1] = z$. There are two comparison operations that we do, namely,

(1) $(\mathtt{lt}^{\mathrm{c}}, \mathtt{eq}^{\mathrm{c}}, \mathtt{gt}^{\mathrm{c}}) \leftarrow \mathsf{Compare}(y, z)$
- Let $s \leftarrow \mathsf{Oselect}(z, y, \mathtt{lt}^{\mathrm{c}}) = \min(y, z)$

(2) $(\mathtt{lt}^{\mathrm{p}}, \mathtt{eq}^{\mathrm{p}}, \mathtt{gt}^{\mathrm{p}}) \leftarrow \mathsf{Compare}(x, s)$

Thus, we have the following truth table for the desired new values for the three cells in question:

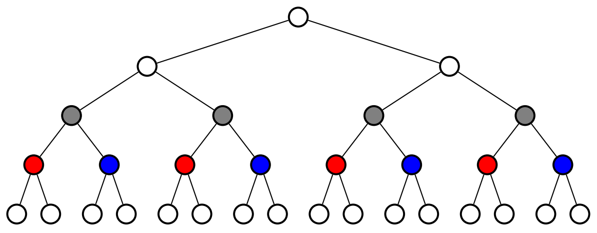


Figure 3: Accessible elements during the optimized ExtractMin. The elements shaded in gray are P, elements in red are L, and the ones in blue are R.

$\mathtt{lt^c}$	$\mathtt{lt^p}$	$\mathbf{D}'[\mathtt{i}]$	$\mathbf{D}'[2\cdot\mathtt{i}]$	$\mathbf{D}'[2\cdot\mathtt{i}+1]$
0	0	z	y	x
0	1	x	y	z
1	0	y	x	z
1	1	x	y	z

From the above truth table we infer that, to restore the heap property, we define $\mathsf{P} \leftarrow (s-x)\cdot(1-\mathtt{lt^p})$ and $\mathsf{L} \leftarrow (x-y)\cdot(\mathtt{lt^c})\cdot(1-\mathtt{lt^p})$, and update: (i) $\mathbf{D}'[\mathtt{i}] \leftarrow \mathbf{D}[\mathtt{i}] + \mathsf{P}$, (ii) $\mathbf{D}'[2\cdot\mathtt{i}] \leftarrow \mathbf{D}[2\cdot\mathtt{i}] + \mathsf{L}$, and (iii) $\mathbf{D}'[2\cdot\mathtt{i}+1] \leftarrow \mathbf{D}[2\cdot\mathtt{i}+1] - (\mathsf{P}+\mathsf{L})$. Therefore to do this process in an oblivious manner, we require two oblivious comparisons, three oblivious reads, and three oblivious updates at each level of the heap. The formal description of the basic ExtractMin appears in Appendix B.

Like the basic binary search, the basic heap protocol hides more information than is needed. The DORAMs in the basic heap operation hide the indices being read and the indices into which an update occurs. Since the heap algorithm inherently leaks part of this information, having general-purpose DORAMs is unnecessary. Next, we will see that using a "heap-specific" DORAM is beneficial.

5.2 Optimized Heap Protocol

Optimized ExtractMin. The main underlying observation to optimize ExtractMin is that, at each level, the algorithm is doing three reads and updates at *related* indices $\mathtt{i}$, $2\cdot\mathtt{i}$, and $2\cdot\mathtt{i}+1$. This fact is public knowledge, even though the value of $\mathtt{i}$ itself is not. We can perform these related-index reads and updates more efficiently than doing three arbitrary reads and updates.

Similarly to the binary search case, given the level d of the heap we are currently working on, only a subset of elements of **D** are *accessible*; i.e., are possible candidates of elements we might be accessing. Let **P** be the subset of **D** consisting of the nodes in level d, **L** be the left children of those nodes, and **R** be the right children of those nodes. Note that **P**, **L**, and **R** all have the same size. (See Figure 3.) Then we can use a single DPF to read index $\mathtt{i}^* = \mathtt{i} - 2^{\mathsf{d}}$ from each of the three accessible sets, yielding the shares of the parent and the two children at the same time. Updating the values is slightly trickier; the Duoram update protocol [33, §4.2.3] exposes the difference between the update value M and the random leaf value at the target index of the DPF (see Section 2.2.1). We therefore cannot reuse these random leaf values with multiple updates, even if we can reuse the DPF itself because the *index* being updated is fixed. The solution is to use wide DPFs (see Section 2.2.3), where each leaf stores three random values instead of one. The updates of $\mathbf{P}[\mathtt{i}^*]$, $\mathbf{L}[\mathtt{i}^*]$, and $\mathbf{R}[\mathtt{i}^*]$ can then all use the same (wide) DPF, and indeed the very same one used for reading. Furthermore, this operation can be performed for all levels with a single incremental wide DPF.

Optimized HeapInsert. We optimized the HeapInsert protocol using our binary search from Section 4.2. Recall that the objective of the procedure is to insert a value M into the heap. The protocol begins by adding an empty node at the end of the heap array. The key observation is that after HeapInsert is complete, the only entries that might change are the ones on the path from the root to this new node. Further, since the number of entries in the heap is public, *which* entries in **D** form this path is also public. We form the accessible set **P** of the nodes from the root to the newly added (empty) node. The next observation is that this path (from root to leaf) starts off sorted, and will end up with the new element M inserted into the correct position so as to keep the path sorted. The path is of length $\lg n$, so we use our binary search to find the appropriate insertion position with a single IDPF of height $\lg\lg n$.

The advice bits of that IDPF will then be bit shares of a vector $\mathbf{t} = [0,0,1,0,\ldots,0]$ with the 1 indicating the position at which the new value must be inserted. The shares of **t** are (locally) converted to shares of $\mathbf{u} = [0,0,1,1,\ldots,1]$ by taking running XORs. The bits of **t** and **u** are used in $2\lg n$ parallel Flag-Word multiplications to shift the elements greater than M down one position, and to write M into the resulting hole, with a single message of communication. Figure 4 illustrates the optimized insert protocol.

6 AVL TREES

In the previous section, we presented oblivious heaps, which admit arbitrary dynamic insertions, but only deletions of the minimum element. In this paper, we do not give a detailed description on how AVL Trees work. We refer the reader to any Algorithms textbook [11] for a refresher on AVL trees. The binary search tree (BST) is a classic data structure that supports arbitrary insertions and deletions. Unfortunately, BSTs have $O(n)$ worst-case costs for all its operations (insert, delete, and lookup). Worse, when working in an oblivious setting, *every* BST operation has to necessarily incur this worst-case $O(n)$ cost lest it reveal the structure of the BST. We therefore look at AVL trees, which are *balanced* binary search trees that support a worst-case $O(\lg n)$ cost for their operations, as their depths are guaranteed to be bounded by $1.44\lg n$, and thus are far more appealing than BSTs under the obliviousness constraint. In AVL trees, an *imbalance* at an element happens when the difference between the heights of its subtrees (the subtrees rooted at its children) is greater than one. Any operation that results in an imbalance is fixed by *rotations* which modify the positions of the imbalanced element and its children. Both insertions and deletions can result in potential imbalances.

Oblivious AVL trees have been proposed in the client-server model [37]. In such a setting, the structure of the AVL tree is known to the client but hidden from the server. Our work is the first oblivious AVL tree design in the model where all the parties involved[4] are oblivious to the underlying AVL tree structure that is being

[4]Recall from Section 2.4 that in our setting the computing parties play the role of both clients and servers.

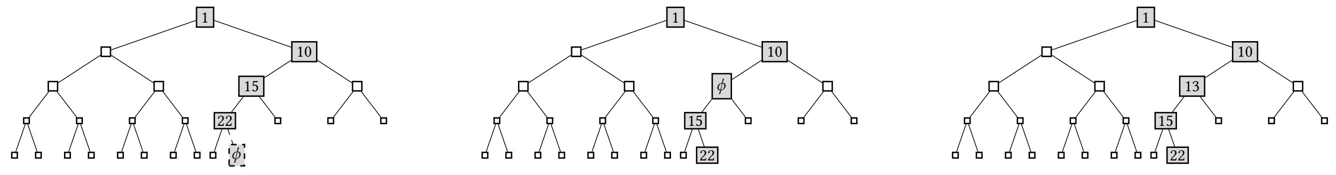
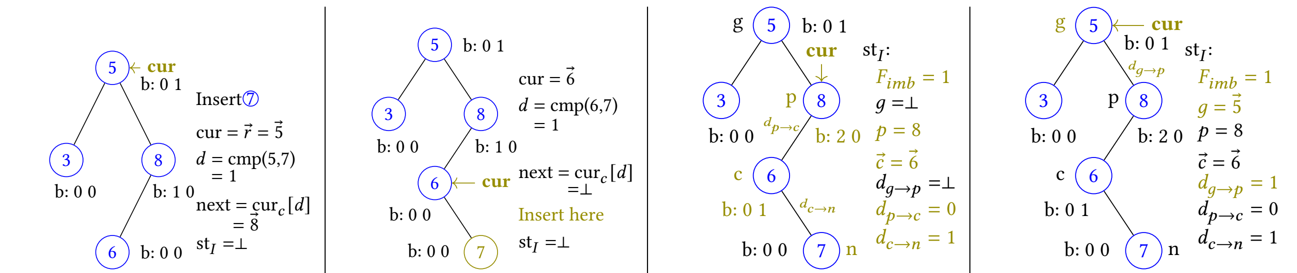


Figure 4: Binary search-based HeapInsert protocol. In the first step *(left)*, a new empty node is created, and an oblivious binary search is done to find the location where M = 13 should be inserted in the (sorted) path from the root to the new node. In the second step *(center)*, an oblivious trickle-down is performed using the output of the binary search. In the last step *(right)*, the desired value is obliviously written into the hole. Note that the gray path is the set of accessible items in the heap. The heap contents and M are secret shared and not known to any party, but which heap *locations* form the accessible set is public knowledge.



(a) **Insertion Phase 1: BST insertion. Compare with key in current (root) node and compute direction bit d. Set the next pointer to traverse to using d. st_I remains empty.**

(b) **Insertion Phase 1: BST insertion. Compare with key in current node and compute direction bit d. Since the next pointer is ⊥, we insert node 7 here. st_I remains empty in this**

(c) **Insertion Phase 2: Gather st_I. At node 8, we have an imbalance so obliviously update st_I to store all the imbalance information at this and lower levels, namely F^{BS}_{imb}, $\vec{p}^{XS}$, $\vec{c}^{XS}$, d_{pc}, d_{cn}.**

(d) **Insertion Phase 2: Gather st_I. At node 8's parent, since it hit an imbalance at the lower level, obliviously update st_I to store the parent to imbalance node $\vec{g}^{XS}$, and the direction from parent d_{gp}.**

Figure 5: Illustration of AVL insertion. (a),(b): Phase 1 - BST Insertion. (c),(d): Phase 2 - Gather insertion state st_I. The insertion is completed by Phase 3 (not shown here) - $\texttt{FixImb}_I$ that uses the state st_I to complete the balancing procedure obliviously. The imbalance chain is $g \to p \to c \to n$, with g - grandparent, p - parent / imbalance point, c - child, and n - c's child on the insertion path. The d_x are direction bit flags, with $x \in \{g \to p, p \to c, c \to n\}$. The balance values at each node are b.

maintained. To this end we design novel *oblivious algorithms* for AVL imbalance detection and balancing rotations.

Our binary search and heap protocols leveraged an *implicit structure*; i.e., the indices of the memory dictate the tree structure. In AVL trees, the position of items in the tree are ephemeral due to the imbalance-fixing rotations. These rotations incur significant overheads if we try to leverage implicit structure as every operation potentially requires reorganizing portions of the DORAM. Obliviousness worsens this overhead as it requires padding each imbalance rotation to the worst case of repositioning all elements in the tree. Hence, AVL trees cannot exploit an implicit structure (and the optimizations it enables).

We therefore consider the DORAM **D** to be composed of items that are *nodes* of an AVL tree. A node is a structure containing key, value, the left and right child pointers (DORAM indices), and left and right balance bits, and as before each server holds a share of this DORAM composed of nodes. We say such types of data structures have an explicit structure, as the locations of nodes in the DORAM **D** are independent of the tree structure. The AVL tree structure is maintained by child pointers of each node in the tree and an external root pointer. We refer to the key, value, children, and balance bits of this structure for a node x as x_k^{AS}, x_v^{XS}, $x_c^{XS}[2]$, and $x_b^{XS}[2]$ respectively.

Memory Layout. Database entry $\mathbf{D}[0]$ is a special entry containing a 0 entry for all fields of the node. The rest of **D** will store nodes as they are inserted without any tree-specific layout (unlike our heap). The child pointers of each node store shares of indices of its children nodes in the DORAM. A child pointer with the value 0 denotes a NULL pointer (the child does not exist). A child node is retrieved by a DORAM read on its shared index. For a node x, we refer to its pointer (index) in the DORAM as $\vec{x}$. Additionally, AVL trees also maintain a global root which stores shares of the index in the DORAM that correspond to the current root of the AVL tree $\vec{\mathbf{r}}^{XS}$, and the number of items currently in the tree **n**. As with heaps, an oblivious AVL tree can be initialized as empty, or from an existing dataset by sorting it; the pointers and balance bits can then be added entirely with local operations.

6.1 Insertion

Inserting a new node x into an AVL tree establishes an *insertion path*, which starts at the root and terminates at a leaf node ℓ, and x is inserted as ℓ's left or right child. Since an insertion can result in an imbalance at any node along the insertion path, naively converting insertion to be oblivious would require us to perform an oblivious rotation step at each level of the AVL tree to prevent leaking the level at which an imbalance occurs. This would incur significant performance penalties. However, for an insert operation we observe that at most one rotation is required to restore balance. Instead of performing oblivious rotations at all levels of the tree, then, we design a novel recursive insert protocol which returns an *insertion state*, st_I, that captures the imbalance point (if any) during the insertion. To restore balance, we introduce a $\mathtt{FixImb}_I$ protocol which takes in st_I as its only parameter. This insertion state st_I, contains shares of (i) $\mathrm{F}^{\mathrm{BS}}_{imb}$ a flag indicating if an imbalance has occured, (ii) index of the imbalanced node $\vec{p}^{\mathrm{XS}}$, (iii) parent of the imbalanced node $\vec{g}^{\mathrm{XS}}$, (iv) the child of the imbalanced node $\vec{c}^{\mathrm{XS}}$, and (v) three bit shares $\mathrm{d}^{\mathrm{BS}}_{gp}$, $\mathrm{d}^{\mathrm{BS}}_{pc}$, and $\mathrm{d}^{\mathrm{BS}}_{cn}$ that capture the direction bits between these nodes (whether p is the left or right child of g, for example), where n is c's child on the insertion path.

To insert an item into the AVL tree, the corresponding node x is inserted into the next available free index $\vec{x}$ in the DORAM. The AVL insert effectively has three phases. The first phase is identical to a classic BST insert. Starting with the node stored at the root ($\vec{\mathrm{r}}^{\mathrm{XS}}$) as the current node, the insertion node's key x_k is compared with the current node's key to get a direction bit d which dictates the direction of traversal for the next level. This process is repeated until we arrive at a node with no children in the traversal direction from it. The new node is obliviously inserted here by updating the children pointers of the current node. In the second phase the imbalance (if any) is captured in st_I, through oblivious flag manipulations on the return path of the recursive insert. Figure 5 illustrates both these phases. The state st_I is used in the third phase, where the imbalance (if any) is fixed with the procedure $\mathtt{FixImb}_I$. $\mathtt{FixImb}_I$ performs the rotations (if required) obliviously and updates the balance bits of the nodes involved in the imbalance, to arrive at a balanced state after the insertion. The insertion protocol in its entirety is detailed in Protocols 8–15 provided in Appendix C.

6.2 Deletion

The deletion procedure of AVL tree begins identical to that of standard BST; i.e., (i) if the node to be deleted y has no children, it can be deleted directly, (ii) if it has one child, then y is deleted and y's parent is updated to point to y's only child, and (iii) if y has both children, then y is swapped with its successor first and then deleted from the tree. Of course, in our setting all of these cases have to be handled obliviously; the computing parties will not be aware of which case they are in. Deletions may result in an imbalance, but in contrast to insertion, deleting a node can require imbalance-fixing rotations at *each level* on the deletion path. Hence unlike our AVL insert, the fix imbalance operation cannot be deferred to the end. However, we can still perform deletion in $O(\log n)$ rounds and bandwidth by performing the fix-imbalance operation $\mathtt{FixImb}_D$, at every level of the AVL tree. Protocols 16–21 in Appendix C detail the complete deletion protocol.

7 APPLICATION: OBLIVIOUS STREAM SAMPLER

An example application of oblivious data structures is oblivious *stream sampling*. A stream sampler of size k receives items one at a time, in a stream of length unknown in advance. At the end of the stream, the stream sampler outputs a random subset of size k of the streamed items, while using only $O(k)$ memory.

A stream sampler provides the key functionality to take advantage of an important result in differential privacy (DP), particularly as used in federated learning: roughly, if you first sample your dataset of size n down to a random sample of size k, and then apply an ϵ-DP algorithm to it, you end up with an $O(k\epsilon/n)$-DP algorithm [2].

Crucially, however, this result requires that the choice of elements in the sample be *unknown*. Previous work [26] has implemented sampling in this fashion in the setting of trusted execution environments (TEEs), like Intel SGX. If one does not wish to trust the security of SGX, however, one can use MPC.

Shi [28, §V-B] provided an oblivious heap-based stream sampler for this application in the client-server MPC model, where the single client *can* see all of the data, but the data is oblivious to the server. Using PRAC, we port over this application to the secret-shared server MPC model, where the data is oblivious to *all* of the players participating in the computation. In our setting, the items may arrive, for example, by having many clients across the Internet asynchronously submit items by each splitting their item into secret shares, and submitting one share to each player.

We, like Shi, use a heap with random labels to implement the reservoir sampling algorithm by Vitter [35], which works as follows: (i) store the first k items, and (ii) for item number $m > k$, select an already stored item at random and replace it with the arrived item with probability k/m. (We correct here a typo of Shi [28], where this value is written as $1/m$.) Our implementation replaces Shi's client-server Path Oblivious Heap with our optimized heap oblivious to all parties, and corrects the typo noted above, but otherwise follows Shi's algorithm faithfully.

We evaluate our implementation in Section 8.2.3.

8 EVALUATION

8.1 Analytical Evaluation

This section analytically evaluates PRAC's algorithms with the state of the art. We start with Table 1, which analytically compares PRAC's binary search (basic and optimized) with (i) Floram's binary search, and (ii) Blanton and Yuan [4]. We remark that the latter works in either the malicious or semi-honest model, but with the same asymptotic complexities [4, §VIII-A]. Floram's amortized bandwidth cost for a DORAM operation is $O(\sqrt{n})$, from its linear-cost refresh operation, which needs to be performed before every $\sqrt{n}/8$ write operations. However for binary search, we need to only perform DORAM reads, which can use Florom (Floram optimized for only read operations) with an $O(\lg n)$ bandwidth cost [12, Table

Table 1: Analytical comparison of MPC binary search. The protocol marked as * works in either the semi-honest or the malicious security model (with the same asymptotic complexity [4, §VIII-A]).

Protocol	Communication		Computation
	Bandwidth	Rounds	
Floram [12]	$O(\lg^2 n)$	$O(\lg^2 n)$	$O(n \lg n)$
Blanton & Yuan [4] *	$O(\sqrt{n})$	$O(\lg n)$	$O(n)$
PRAC (Basic)	$O(\lg^2 n)$	$O(\lg n)$	$O(n \lg n)$
PRAC (Optimized)	$O(\lg n)$	$O(\lg n)$	$O(n)$

Table 2: Analytical comparison of MPC Heap protocols. The protocol marked as * works in the malicious security model with an honest majority.

Protocol	HeapInsert			ExtractMin		
	Communication		Computation	Communication		Computation
	Bandwidth	Rounds		Bandwidth	Rounds	
Mazloom et al. [23] *	$O(\lg^2 n)$	$O(\lg^2 n)$	$O(\lg^2 n)$	$O(1)$	$O(1)$	$O(n)$
PRAC's Basic Heap	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$	$O(\lg^2 n)$	$O(\lg n)$	$O(n \lg n)$
PRAC's Optimized Heap	$O(\lg n)$	$O(\lg \lg n)$	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$	$O(n)$

1]. Performing naive binary search with Floram would require $\lg n$ ORAM read operation each incurring $\lg n$ bandwidth and rounds. Note that the reduction in bandwidth for optimized PRAC is from using a single IDPF.

PRAC's heap protocols (basic and optimized) are compared against Mazloom et al. [23] in Table 2. Mazloom et al.'s heaps work in the malicious model with an honest majority. Mazloom et al.'s ExtractMin operations have an amortized communication cost of $O(1)$, compared to PRAC's worst-case $O(\lg n)$ cost. However, PRAC's HeapInsert has a bandwidth cost of $O(\lg n)$ sent in $O(\lg \lg n)$ rounds, in contrast to Mazloom et al.'s amortized $O(\lg^2 n)$ cost for bandwidth, rounds, and computation. We remark here that a cheaper HeapInsert is preferable since the number of insert operations has to be at least the number of extract operations.

PRAC is the only framework to support AVL trees in a distributed trust setting, hence we do not provide any comparisons. Our AVL algorithms incur an $O(\lg n)$ bandwidth and round cost, with an $O(n)$ computation cost. We present a breakdown of the number of MPC operations in the various PRAC algorithms in Table 6 in Appendix D.

8.2 Empirical Evaluation

Experimental Setup. We implemented[5] and benchmarked PRAC with a reference implementation in C++, using Boost Asio for asynchronous network communications. All of our experiments except those in Section 8.2.3 were performed on a single machine with dual Intel 8380 2.3 GHz 40-core processors. We ran each party in its own docker container, each pinned to a dedicated 20 cores, and used netem to induce 100 Mbps bandwidth and 30 ms each-way latency between the containers. For the comparator systems, we ran their available code in this same setup. To test our system across a live network, in Section 8.2.3 we also include experiments performed on Amazon EC2 instances located at us-east, ca-central, and us-west. The PRGs in DPFs are implemented using AES. Our experiments use DORAMs with 64-bit words.

8.2.1 Evaluating PRAC's DORAM. Figure 6 compares the performance (both wall-clock time and bandwidth) of PRAC's improved Duoram, under the typical Internet conditions used by prior work [8, 33] (30 ms latency and 100 Mbps bandwidth) with prior work, namely (i) Ramen [8], (ii) Duoram [33], (iii) Floram [12], and (iv) 3P-Circuit ORAM [20]. To compare, we (i) fix the DORAM size at 2^{20} and vary the number of Read operations, and (ii) fix the number of Read operations at 10 and vary the DORAM sizes from 2^{16} to 2^{30}. Figure 6 shows that for a DORAM of size 2^{20}, PRAC is the best performing DORAM. PRAC's implementation-level improvements as mentioned in Section 3 makes its performance better than Duoram. Our benchmarks also include the initialization costs. Observe that for DORAM sizes of 2^{30}, 3P-Circuit ORAM outperforms PRAC as PRAC's linear computational cost becomes the bottleneck [33, Table 1]. We observe that Ramen, PRAC, and Duoram have the least bandwidth consumption, but Ramen must perform a refresh process with an $O(n)$ bandwidth cost every $\sqrt{n}$ operations, not accounted for in Figure 6.

8.2.2 Evaluating Oblivious Data Structures.

Baseline setups for evaluating data structures. Since most DORAM schemes do not implement data structures (except Floram, which implements binary search), we estimate costs for other schemes based on their underlying DORAM costs. We estimate these costs as follows. Table 6 (Appendix D) presents a detailed accounting for all the operations involved in our oblivious data structures. However, we limit ourselves to the underlying DORAM costs for estimated costs. In particular we use (i) the DORAM preprocessing cost for the n in consideration, (ii) the DORAM cost per access for this value of n, and (iii) and the number of such operations required for the data structure, to arrive at just the DORAM costs of the data structure and project this as the total data structure cost. The overheads of computation, communication latency, and bandwidth from all the other MPC operations (see Table 6) are not accounted for in the estimates of PRAC's comparators, but they are for PRAC. Consequently, the performance benefits we claim throughout this section are conservative. Also, note that the estimates of Ramen and 3P-Circuit ORAM are for the basic binary search and heap versions; these DORAMs cannot leverage our optimizations since the underlying Square-root ORAM and Circuit ORAM (of Ramen and 3P-Circuit ORAM, respectively) preclude them from operating over a subset (of accessible elements) of the ORAM. From Section 8.2.1, we observe that among the comparators of PRAC, Ramen performs best for small DORAMs, and 3P-Circuit ORAM performs best for large DORAMs. Therefore, to evaluate PRAC's data structures, we benchmark them against estimates for Ramen and 3P-Circuit ORAM.

Binary Search. Figure 7 compares PRAC's optimized and basic binary search with (i) Floram's binary search implementation, (ii) Ramen (estimate), and (iii) 3P-Circuit ORAM (estimate). For a constant DORAM size, we observe that as the number of searches increases linearly, so does the time taken. Next, we consider the case of doing one binary search and varying the DORAM size. Recall

[5]Code available at https://git-crysp.uwaterloo.ca/iang/prac/.

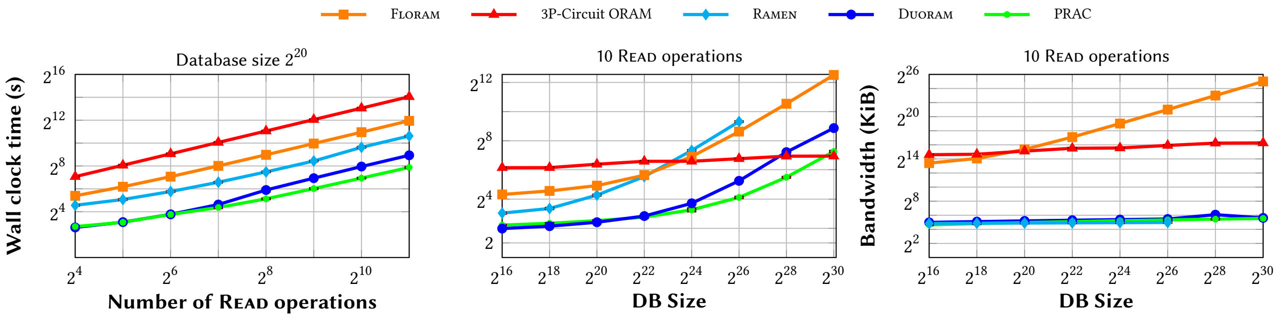


Figure 6: Comparing PRAC's underlying DORAM with Floram, 3P-Circuit ORAM, Ramen, and Duoram. The network settings are 30 ms latency and 100 Mbps throughput. Note that the plots for some systems stop early when they support only up to that size. The bandwidth lines for Duoram, Ramen, and PRAC nearly coincide. The error bars are too small and thus not visible.

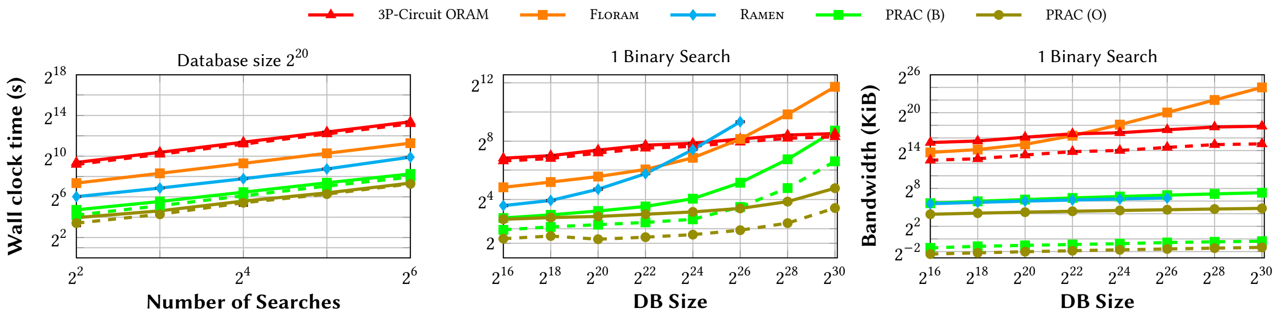


Figure 7: Comparing PRAC's Binary Search with 3P-Circuit ORAM, Floram, and Ramen. The network settings are 30 ms latency and 100 Mbps throughput. Note that some implementations stop early since they support only up to that size. PRAC (B) and PRAC (O) are the basic and optimized versions of PRAC respectively. The dashed lines represent the online portion of the cost. The error bars are too small and thus not visible.

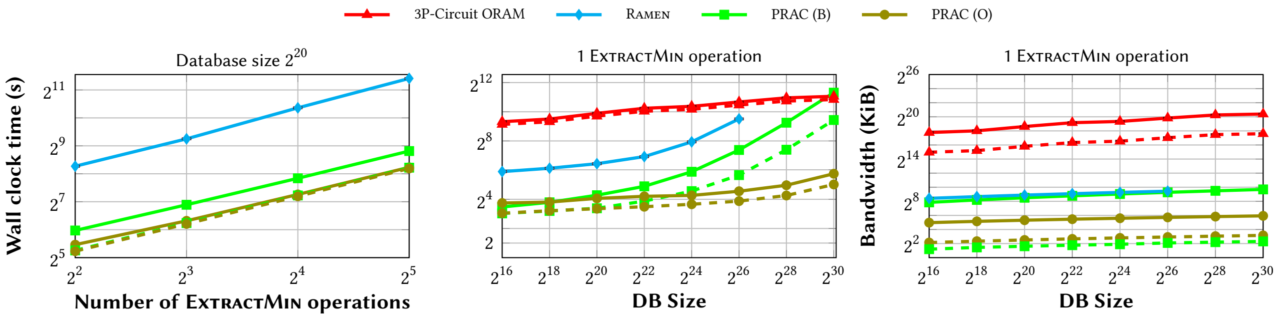


Figure 8: Comparing PRAC's ExtractMin with Ramen and 3P-Circuit ORAM. The network settings are 30 ms latency and 100 Mbps throughput. Note that some implementations stop early since they support only up to that size. For a constant DORAM size of 2^{20}, we only compare against Ramen since it is the best performing DORAM at this DORAM size. PRAC (B) and PRAC (O) are the basic and optimized versions of PRAC respectively. The dashed lines represent the online portion of the cost. The error bars are too small and thus not visible.

that the crossover point between PRAC and 3P-Circuit ORAM is 2^{30} for performing DORAM reads. Thus, we observe similar behaviour for basic binary search. However, our optimized IDPF-based binary search outperforms 3P-Circuit ORAM for even DORAM sizes exceeding the crossover point, despite our underlying DORAM being costlier at these sizes, highlighting the benefits of our optimizations.

We also observe that using one IDPF rather than lg n DPFs reduces bandwidth consumption.

Concretely, for 2^{26} items[6] PRAC (optimized) presents a 3.4× and 5.0× improvement over PRAC (basic) for binary search in wall-clock

[6]We pick 2^{26} as the comparison point, since Ramen only supports DORAM sizes of up to 2^{26}.

Table 3: HeapInsert evaluation, comparing PRAC Basic (B) and PRAC Optimized (O) in terms of wall-clock time, number of rounds, and bandwidth to do one insert on DORAM of sizes 2^{16}, 2^{20}, and 2^{24}. The preprocessing numbers are shaded in gray.

	Size	Wall-Clock Time (s)	Rounds	BW (KiB)
Insert (B)	2^{16}	0.183 ± 0.001 + 3.747 ± 0.001	1 + 61	9.7 + 0.6
	2^{20}	0.184 ± 0.001 + 4.716 ± 0.005	1 + 77	12.3 + 0.7
	2^{24}	0.184 ± 0.001 + 5.667 ± 0.003	1 + 93	14.9 + 0.8
Insert (O)	2^{16}	1.525 ± 0.001 + 0.866 ± 0.004	17 + 17	5.6 + 0.6
	2^{20}	1.525 ± 0.001 + 0.866 ± 0.003	17 + 17	5.7 + 0.7
	2^{24}	1.526 ± 0.001 + 0.860 ± 0.006	17 + 17	5.9 + 0.8

Table 4: AVL tree evaluation, showing wall-clock time, number of rounds, and bandwidth to do one insert or delete on DORAM of sizes 2^{16}, 2^{20}, and 2^{24}. The preprocessing numbers are shaded in gray.

	Size	Wall-Clock Time (s)	Rounds	BW (KiB)
Insert	2^{16}	3.08 ± 0.04 + 25.4 ± 0.1	37 + 440	107.6 + 2.7
	2^{20}	5.5 ± 0.3 + 34.5 ± 0.2	45 + 525	147.8 + 3.2
	2^{24}	21.2 ± 0.3 + 70 ± 2	53 + 627	199.9 + 3.8
Delete	2^{16}	3.12 ± 0.01 + 67.46 ± 0.03	37 + 1182	239.3 + 9.2
	2^{20}	8.88 ± 0.08 + 86.5 ± 0.2	45 + 1427	343.7 + 11.3
	2^{24}	29.88 ± 0.08 + 147 ± 2	53 + 1721	479.2 + 13.6

time and bandwidth, respectively, showcasing our algorithmic improvements. For the same 2^{26} items, when comparing with Ramen, PRAC (optimized) provides 61× and 3.6× improvements in wall-clock time and bandwidth, respectively. Compared with 3P-Circuit ORAM, PRAC (optimized) improves wall-clock time by 27.1× and reduces bandwidth by more than three orders of magnitude (> 6400×). While there is a recent work by Blanton and Yuan [4], we do not compare against them since their implementation is not publicly available.

Heaps. Figure 8 compares basic and optimized PRAC's ExtractMin with Ramen and 3P-Circuit-ORAM. The gap between Ramen and (basic) PRAC is lower in Figure 8 than in Figure 6 since the initialization cost of Ramen is amortized over many DORAM operations. Like in the case of binary search, the basic PRAC implementation crosses over 3P-Circuit-ORAM at DORAM sizes of 2^{30}. However, our wide IDPF helps optimized PRAC perform better even at DORAM sizes where our underlying DORAM is inferior.

The improvements that we see in optimized ExtractMin protocol are because (i) the preprocessing is much cheaper since it requires the generation of a single wide IDPF, and (ii) the online phase is cheaper because the ORAM operations are performed on smaller logical DORAM rather than the full DORAM. We see from the plot that for smaller DORAM sizes, PRAC's basic and optimized ExtractMin perform almost the same in the online phase. This is because the latency cost is the bottleneck at this point, which is similar for both our versions. As the DORAM size increases, the advantage of operating on smaller DORAM for ORAM reads starts bearing fruit. Concretely, for 2^{26} items, PRAC (optimized) improves over PRAC (basic) 7.1× and 11.7× in terms of wall-clock time and bandwidth respectively, highlighting our algorithmic improvements. Against Ramen, PRAC (optimized) provides 31× and 13× improvements in terms of time and bandwidth for the same size of 2^{26} items; in comparison to 3P-Circuit ORAM, PRAC (optimized) presents a 69× improvement in time and over four magnitudes improvements (> 17500×) in terms of bandwidth.

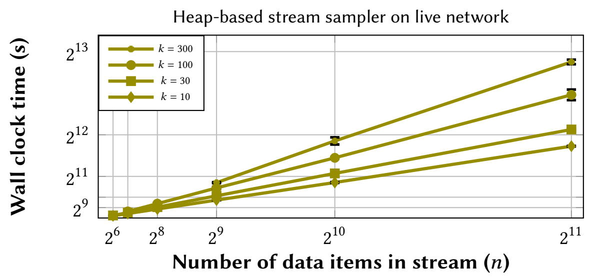


Figure 9: Timings for running the end-to-end heap-based stream sampler with players separated across the Internet, not simulated in dockers. n is the total number of data items in the stream; k is the number of items in the resulting sample.

Since HeapInsert has no DORAM operations, we do not provide comparisons with other works and only compare our basic and optimized implementations. Table 3 compares PRAC's basic HeapInsert with our binary search-based optimization. Since the basic HeapInsert does not require DPF generation, its preprocessing cost is over 8× cheaper than the optimized version. The number of rounds for our optimized HeapInsert for a heap of size n is $3\lceil \lg \lg(n+1) \rceil + 2$; this is nearly a constant for all practical purposes.

AVL Trees. Since PRAC is the first framework to support AVL trees in a distributed trust setting, we do not have any direct comparisons. Furthermore, since DORAM operations do not constitute the bulk of the AVL cost, there is no reasonable way to estimate the performance of an AVL implementation using other DORAMs. Table 4 presents the evaluation of PRAC's AVL implementation.

8.2.3 *Evaluating the Stream Sampler Application.*

We next evaluate our implementation of the heap-based stream sampling application described in Section 7. In this evaluation, we also demonstrate running PRAC on machines separated over the Internet, as opposed to in docker containers on a single machine with artificial latency.

For the experiments in this section, we used three Amazon EC2 t2.large instances, in the us-west, us-east, and ca-central regions. The one-way latencies between them ranged from 10 to 35 ms, which is comparable to the latencies in the docker experiments.

The stream sampler has two parameters: n is the number of data items in the stream; the parties learn the data items one at a time in secret-shared form. k is the size of the sample; the stream sampler will keep a random subset of size k of the data items (without knowing n in advance).

We ran our experiment using PRAC for multiple values of n and k (always with $n > k$ of course); the resulting end-to-end timings are shown in Figure 9. Unlike for the docker experiments above, the error bars for the results of this experiment are visible (if barely), owing to the variability in the real-life latency between the parties.

For the largest configuration we tested ($(n, k) = (2^{11}, 300)$), PRAC's total time was about 2.5 hours. We also compared this configuration to 3P-Circuit ORAM and Ramen. For the comparator systems, as in Figure 8, we conservatively estimate the time those systems would take by only taking into account the DORAM operations required for only the heap extraction component of the heap-based stream sampler protocol. (We emphasize that PRAC's timings, on the other hand, are a complete end-to-end time of the entire stream sampler protocol.) We ran the comparator schemes on the same EC2 instances as we used for the PRAC experiments in this section. We found that where PRAC took about 2.5 hours to evaluate the largest configuration we tested, Ramen would require at least 15 hours, and 3P-Circuit ORAM would require over 3.5 *days*.

9 RELATED WORK

Distributed ORAMs. The distributed ORAM literature can be classified into two categories on a broad level: (i) Communication-efficient, and (ii) Computation-efficient. Communication-efficient DORAMs typically have linear local computation costs. These ORAM schemes optimize for round complexity and bandwidth cost. Such DORAMs are built on top of DPFs, and the evaluation of the DPFs incurs the linear cost. Floram [12] and Duoram [33] fall in this category. Computation-efficient DORAMs optimize for the local computation cost, in exchange for higher bandwidth and rounds. Jarecki and Wei's 3P-Circuit ORAM [20] derived from tree-based ORAMs [36] and Braun et al.'s Ramen [8] based on the square-root ORAM [16] fall in this category. A recent work by Falk et al. [13] presents GigaDORAM. Their work targets the setting where the three mutually distrusting parties have their servers colocated in the same datacentre, while PRAC allows for typical Internet latencies between the servers.

Oblivious Binary Search. Floram performs binary search with the basic oblivious binary search protocol from Section 4.1 with the difference that the $\lg n$ DPFs are produced in the online phase. Blanton and Yuan [4] present a collection of binary search protocols for MPC without DORAMs. Their best construction achieves a $O(\sqrt{n})$ communication and $O(n)$ computation for performing a binary search. Their work also details extensions for binary search for updating the items being searched upon or inserting (similarly deleting) items from the search set; these extensions have an $O(n)$ local computation and bandwidth cost.

Oblivious Heaps. Oblivious heaps have been studied in MPC for the client-server model [19, 28, 37], a different setting from our work. In such protocols, the client knows the state of the oblivious data structure; clients perform computations over subsets of the data in private unobservable memory on the client side, and the single untrusted server stores the entire data structure. In contrast, in our work, the computing servers play the role of servers and clients, and the data structure state is oblivious to all parties.

Keller and Scholl [21] present an oblivious priority queue with $O(\log^5 n)$ computation and $O(\log^3 n)$ rounds per operation. Their work ports the first tree-based ORAM, SCSL ORAM [29], to the MPC setting, and designs a priority queue in the basic fashion (Section 5.1). We compare our heap protocols against Keller and Scholl's approach with 3P-Circuit ORAM, the current state-of-the-art version of tree-based MPC ORAMs, resulting in a heap with $O(\log^4 n)$ computation and $O(\log^2 n)$ rounds per operation. Mazloom et al. [23] presents a *data-independent* priority queue that improves upon work by Toft [31]. Their construction has an $O(\log^2 n)$ amortized insertion cost and $O(1)$ amortized extract cost, where the cost is in terms of comparison operations. Not included in this $O(1)$ amortized extract cost is a 'pack' operation which incurs an $O(n)$ local computation. In comparison, our heaps support an $O(\log n)$ worst-case insertion cost in terms of bandwidth and rounds with a comparison cost of $O(\log \log n)$, and extract with worst-case bandwidth, round, and comparison cost of $O(\log n)$. Our lower insertion cost is favorable in practice as the number of delete operations cannot be greater than the number of inserts. However, their priority queue works in a stronger adversarial model (one of the three parties may be malicious), and is designed to support extra functionality, namely Read-Front(x), meant to read (but not delete) the front x items in the queue with an $O(1)$ worst-case cost. Our algorithms can easily support a Read-Front(1) with $O(1)$ worst-case cost, but reading more than the minimum item is non-trivial. However this functionality is extraneous to the definition of a heap, and they introduce it to facilitate their dark pool application.

AVL Trees. While ours is the first oblivious AVL tree protocol in the distributed trust setting, Wang et al. put forth Oblivious AVL trees [37] in the client-server model. In our work, the data structure state is oblivious to all parties. In contrast, Wang et al.'s protocol takes advantage of private unobservable client-side memory (just like the client-server model oblivious heaps discussed earlier), and the client knows the data structure state in their protocol.

10 CONCLUSION

In this work, we present PRAC, a general MPC framework designed for oblivious data structures in MPC with efficient communication costs, in terms of both rounds and bandwidth overheads. We present three different oblivious data structures exemplifying different functionalities within PRAC, namely binary search (for static data), heaps (restrictively dynamic), and AVL trees (fully dynamic). PRAC presents algorithmic optimizations for these data structures that exploit different types of DPFs, and the ability of DORAMs like Duoram to selectively access a subset of the ORAM efficiently. These improvements result in asymptotic as well as significant concrete performance improvements.

For binary search, our optimizations reduce computation from $O(n \lg n)$ to $O(n)$ and communication from $O(\lg^2 n)$ to $O(\lg n)$, and concretely improves the wall-clock time by more than an order of magnitude. Similarly for heaps, our techniques reduce the round trips from $O(\lg n)$ to $O(\lg \lg n)$ for insertion, and reduces the bandwidth from $O(\lg^2 n)$ to $O(\lg n)$ for extracting the min item; our extract algorithm improves wall-clock time by more than an order of magnitude. PRAC presents the first AVL tree construction in a distributed trust setting as a means to demonstrate fully dynamic data structures for MPC. We hope that the flexibility of PRAC will enable the exploration and design of more efficient fully dynamic data structures for MPC in the future.

ACKNOWLEDGMENTS

This work benefited from the use of the CrySP RIPPLE Facility at the University of Waterloo. We thank NSERC (RGPIN-2023-03260) and the Royal Bank of Canada for funding this work. This research was undertaken, in part, thanks to funding from the Canada Research Chairs program, award CRC-2018-00135. This work was also supported by the University of Waterloo Cybersecurity and Privacy Institute with an Excellence Graduate Scholarship sponsored by MasterCard Canada, and by an Ontario Graduate Scholarship.

REFERENCES

[1] D.A. Adeniyi, Z. Wei, and Y. Yongquan. Automated Web Usage Data Mining and Recommendation System Using K-Nearest Neighbor (KNN) Classification Method. *Applied Computing and Informatics*, 2016.

[2] Borja Balle, Gilles Barthe, and Marco Gaboardi. Privacy Amplification by Subsampling: Tight Analyses via Couplings and Divergences. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[3] Amos Beimel and Yuval Ishai. Information-Theoretic Private Information Retrieval: A Unified Construction. In *ICALP*. Springer, 2001.

[4] Marina Blanton and Chen Yuan. Binary Search in Secure Computation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2022.

[5] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight Techniques for Private Heavy Hitters. In *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[6] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function Secret Sharing. In *Advances in Cryptology - EUROCRYPT*, 2015.

[7] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function Secret Sharing: Improvements and Extensions. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[8] Lennart Braun, Mahak Pancholi, Rahul Rachuri, and Mark Simkin. Ramen: Souper Fast Three-Party Computation for RAM Programs. In *Proceedings of the 30th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023.

[9] John Cartlidge, Nigel P. Smart, and Younes Talibi Alaoui. MPC Joins The Dark Side. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. Association for Computing Machinery, 2019.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* 2009.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009.

[12] Jack Doerner and abhi shelat. Scaling ORAM for Secure Computation. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2017.

[13] Brett Hemenway Falk, Rafail Ostrovsky, Matan Shtepel, and Jacob Zhang. GigaDORAM: Breaking the Billion Address Barrier. USENIX Association, 2023.

[14] Niv Gilboa and Yuval Ishai. Distributed Point Functions and Their Applications. In *Advances in Cryptology - EUROCRYPT 2014*, 2014.

[15] Ian Goldberg. Improving the Robustness of Private Information Retrieval. In *IEEE Symposium on Security and Privacy*, 2007.

[16] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[17] Syed Mahbub Hafiz and Ryan Henry. Querying for Queries: Indexes of Queries for Efficient and Expressive IT-PIR. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[18] Syed Mahbub Hafiz and Ryan Henry. A bit more than a bit is more than a bit better: Faster (essentially) optimal-rate many-server PIR. *Proceedings of Privacy Enhancing Technologies (PoPETs)*, 2019.

[19] Zahra Jafargholi, Kasper Green Larsen, and Mark Simkin. Optimal Oblivious Priority Queues. In *SODA*. SIAM, 2021.

[20] Stanislaw Jarecki and Boyang Wei. 3PC ORAM with Low Latency, Low Bandwidth, and Fast Batch Retrieval. In *Applied Cryptography and Network Security (ACNS)*, 2018.

[21] Marcel Keller and Peter Scholl. Efficient, Oblivious Data Structures for MPC. In *Advances in Cryptology–ASIACRYPT 2014*, 2014.

[22] Steve Lu and Rafail Ostrovsky. Distributed Oblivious RAM for Secure Two-Party Computation. In *Theory of Cryptography*, 2013.

[23] Sahar Mazloom, Benjamin Diamond, Antigoni Polychroniadou, and Tucker Balch. An Efficient Data-Independent Priority Queue and its Application to Dark Pools. In *Proceedings of Privacy Enhancing Technologies*, 2023.

[24] M. Narasimha Murty and V. Susheela Devi. *Nearest Neighbour Based Classifiers.* 2011.

[25] Ronald L. Rivest, Adi Shamir, and Yael Tauman. How to Leak a Secret. In *Advances in Cryptology — ASIACRYPT*, 2001.

[26] Sajin Sasy and Olga Ohrimenko. Oblivious Sampling Algorithms for Private Data Analysis. In *33rd Conference on Neural Information Processing Systems (NeurIPS 2019)*, 2019.

[27] Sacha Servan-Schreiber, Kyle Hogan, and Srinivas Devadas. AdVeil: A Private Targeted Advertising Ecosystem. IACR Cryptol. ePrint Arch, Report 2023/1032, 2021.

[28] Elaine Shi. Path Oblivious Heap: Optimal and Practical Oblivious Priority Queue. In *IEEE Symposium on Security and Privacy*, 2020.

[29] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log n)^3)$ Worst-Case Cost. In *Asiacrypt*, 2011.

[30] Kyle Storrier, Adithya Vadapalli, Allan Lyons, and Ryan Henry. Grotto: Screaming Fast $(2+1)$-PC for $\mathbb{Z}_{2^n}$ via (2, 2)-DPFs. In *Proceedings of the 30th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023.

[31] Tomas Toft. Secure Data Structures Based on Multi-Party Computation. In *PODC*. ACM, 2011.

[32] Adithya Vadapalli, Fattaneh Bayatbabolghani, and Ryan Henry. You May Also Like... Privacy: Recommendation Systems Meet PIR. *Proceedings of Privacy Enhancing Technologies (PoPETs)*, 2021.

[33] Adithya Vadapalli, Ryan Henry, and Ian Goldberg. Duoram: A Bandwidth-Efficient Distributed ORAM for 2- and 3-Party Computation. In *32nd USENIX Security Symposium*, 2023.

[34] Adithya Vadapalli, Kyle Storrier, and Ryan Henry. Sabre: Sender-Anonymous Messaging with Fast Audits. In *IEEE Symposium on Security and Privacy (SP)*, 2022.

[35] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Transaction Math. Softw.*, 11:37–57, 1985.

[36] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[37] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious Data Structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

A OBLIVIOUS BINARY SEARCH PROTOCOLS

In this section, we present the basic and optimized binary search protocols we described in Section 4. Protocol 1 describes the basic oblivious binary search protocol, and Protocol 2 describes the optimized version.

From Section 4.2, recall that the *accessible elements* are the elements of the DORAM that could possibly be accessed in a particular DORAM operation. In binary search, these elements are separated by a fixed distance (see Figure 2). We formalize this notion of equally spaced elements forming an accessible set as a Stride:

Definition A.1 (Stride)**.** Stride refers to the elements in the DORAM that are separated by a fixed distance. More formally, $\mathbf{D}' \leftarrow \mathsf{Stride}(\mathbf{D}, \sigma, \mathtt{offset})$, such that $\mathbf{D}'[\mathtt{i}] = \mathbf{D}[\mathtt{offset} + \mathtt{i} \cdot \sigma]$.

Importantly, PRAC allows DORAM operations on accessible sets such as a Stride with a cost that depends only on the size of the accessible set, not on the size of the entire DORAM $\mathbf{D}$.

An IDPF allows access to these accessible sets via an index with a incrementally growing prefix. IDPFs also allow appending a bit to this prefix. This reveals publicly that the subsequent index being retrieved is either twice the previous one, or one more than twice the previous one, while hiding (even from the computational parties themselves) which case they are in, or what the previous or new indices are. Since this information is clear from the structure of binary search anyway, it is safe to leak, and yields our performance improvements.

B OBLIVIOUS MIN-HEAP PROTOCOLS

In this section we present the basic and optimized heap protocols we described in Section 5. Protocol 3 describes the basic HeapInsert

Protocol 1 Basic Oblivious Binary Search Protocol, searching for M^{AS} in a sorted array $\mathbf{D}^{AS}$ with numitems elements. The protocol returns the shares of the smallest memory index containing the value at least M.

```
1:  Find d such that $2^{d} >$ numitems
    // $\mathtt{CurInd}^{AS}$ initially holds shares of $2^{d-1} - 1$
2:  $\mathtt{CurInd}^{AS} \leftarrow \mathsf{party} \cdot (2^{d-1} - 1)$, for party $\in \{0, 1\}$
3:  while $\mathsf{d} > 0$ do
4:      $\mathtt{CurVal}^{AS} \leftarrow \mathbf{D}[\mathtt{CurInd}^{AS}]$
5:      $(\mathtt{lt}^{BS}, \mathtt{eq}^{BS}, \mathtt{gt}^{BS}) \leftarrow \mathsf{Ocompare}(\mathsf{M}^{AS}, \mathtt{CurVal}^{AS})$
6:      if $\mathsf{d} > 1$ then
    // $P_0$ sets its share of uncd to 0
    // $P_1$ sets it to $2^{d-2}$
7:          $\mathsf{uncd}^{AS} \leftarrow \mathsf{party} \cdot 2^{d-2}$
    // $P_0$ sets its share of cond to 0
    // $P_1$ sets it to $2^{d-1}$
8:          $\mathsf{cond}^{AS} \leftarrow \mathsf{party} \cdot 2^{d-1}$
    // A Flag-Word Multiplication
9:          $\mathsf{condprod}^{AS} \leftarrow \mathtt{lt}^{BS} \cdot \mathsf{cond}^{AS}$
    // Conditionally add or subtract $2^{d-2}$
10:         $\mathtt{CurInd}^{AS} \leftarrow \mathtt{CurInd}^{AS} - \mathsf{uncd}^{AS} + \mathsf{condprod}^{AS}$
11:     else
    // $P_0$ and $P_1$ get shares of 1
12:         $\mathsf{cond}^{AS} \leftarrow \mathsf{party}$
    // A Flag-Word Multiplication
13:         $\mathsf{condprod}^{AS} \leftarrow \mathtt{lt}^{BS} \cdot \mathsf{cond}^{AS}$
14:         $\mathtt{CurInd}^{AS} \leftarrow \mathtt{CurInd}^{AS} + \mathsf{condprod}^{AS}$
15:     end if
16:     $\mathsf{d} \leftarrow \mathsf{d} - 1$
17: end while
18: return $\mathtt{CurInd}^{AS}$
```

protocol. This protocol adds the element M to the end of the heap. Note that the *location* of the added item is public, but its *value* is not. It then obliviously swaps the item in that location with its parent, so that the smaller value becomes the parent. It then obliviously swaps the parent with the parent's parent, and so on, until it hits the root of the heap.

Protocol 4 describes the optimized HeapInsert protocol. Recall from Section 5.2 that the accessible items in a HeapInsert protocol lie on the path from the root to the newly inserted item. This notion is formalized as follows.

Definition B.1 (Path)**.** The notion of Path exists when the DORAM represents an implicit tree structure, such that $\mathbf{D}[2\cdot\mathtt{i}]$ and $\mathbf{D}[2\cdot\mathtt{i}+1]$ are left and right children of $\mathbf{D}[\mathtt{i}]$, and $\mathbf{D}[1]$ is the root. Path refers to elements in the ORAM that form a path from the root to a particular node. More formally, $\mathbf{D}' \leftarrow \mathsf{Path}(\mathbf{D}, \mathsf{node})$, such that $\mathbf{D}'[k] = \mathbf{D}[\lfloor \mathsf{node}/2^{n-k-1} \rfloor]$ for a path with $n = \lfloor \lg \mathsf{node} \rfloor + 1$ elements.

By the heap property, elements in the path from the root to the last value in the heap are sorted. Suppose we want to write a value, M, into the heap. PRAC's optimized heap insert does a binary search on this path to find the position where M belongs. It then obliviously trickles down the elements in the path, starting

Protocol 2 Optimized Oblivious Binary Search Protocol, searching for M^{AS} in a sorted array $\mathbf{D}^{AS}$ with numitems elements. The protocol returns the shares of the smallest memory index containing the value at least M.

```
1:  Find d such that $2^{d} >$ numitems
2:  $\mathsf{mid} \leftarrow 2^{d-1} - 1$
    // Explicit, not DORAM, read
3:  $\mathsf{val}^{AS} \leftarrow \mathbf{D}[\mathsf{mid}]$
4:  $(\mathtt{lt}^{BS}, \mathtt{eq}^{BS}, \mathtt{gt}^{BS}) \leftarrow \mathsf{Ocompare}(\mathsf{M}^{AS}, \mathsf{val}^{AS})$
5:  if $\mathsf{d} = 1$ then
6:      return $\mathtt{lt}^{BS}$
7:  end if
    // Append lt to empty IDPF prefix
8:  $\mathtt{i}^{XS,(p)} \leftarrow \mathtt{i}^{XS,(\epsilon \| \mathtt{lt}^{BS})}$
9:  $\mathsf{d} \leftarrow \mathsf{d} - 1$
10: while $\mathsf{d} > 0$ do
    // $\mathbf{D}^{(d)}$ is the array of accessible items at the current depth
11:     $\sigma \leftarrow 2^{d}$, $\mathsf{offset} \leftarrow 2^{d-1} - 1$
12:     $\mathbf{D}^{(d)} \leftarrow \mathsf{Stride}(\mathbf{D}, \sigma, \mathsf{offset})$
    // read is performed using IDPF
13:     $\mathsf{val}^{AS} \leftarrow \mathbf{D}^{(d)}[\mathtt{i}^{XS,(p)}]$
14:     $(\mathtt{lt}^{BS}, \mathtt{eq}^{BS}, \mathtt{gt}^{BS}) \leftarrow \mathsf{Ocompare}(\mathsf{M}^{AS}, \mathsf{val}^{AS})$
    // Append lt to current prefix
15:     $\mathtt{i}^{XS,(p)} \leftarrow \mathtt{i}^{XS,(p \| \mathtt{lt}^{BS})}$
16:     $\mathsf{d} \leftarrow \mathsf{d} - 1$
17: end while
18: return $\mathtt{i}^{XS}$
```

that position and thus creating room for M at the desired location. Finally, M is obliviously written to the correct location.

Protocol 5 describes ExtractMin protocol. The ExtractMin protocol saves the root (which is the minimum element), replaces it with the last element of the heap, and then restores the heap property by running a Heapify protocol at each level of the heap from the root down. Heapify obliviously swaps the node at the index it is given with its smaller child (if that child is smaller than the node itself), and returns the share of the index of the smaller child. In this way, repeatedly calling Heapify on the output of the previous call will restore the heap property.

The basic version of the Heapify protocol is described in Protocol 6. The optimized Heapify protocol, described in Protocol 7, uses a wide incremental DPF to perform the read and update operations. It exploits the fact that each level the accessible elements for the three update operations are (i) all the nodes in that level (a Stride with $\sigma = 1$), (ii) all the left children in the next level (a Stride with $\sigma = 2$), and (iii) all the right children in the next level (a Stride with $\sigma = 2$), and the *same* index into these three Strides will yield the desired node and its two children.

C OBLIVIOUS AVL TREE PROTOCOLS

In Section 6, we presented the high-level idea of our oblivious AVL tree protocols, and here we present the protocols in depth. For AVL trees, the maximum depth of a tree with n items inserted is upper bounded by $1.44 \cdot \lg n$ [10]. In order to traverse the tree for any operation, starting with the root as the current node, at

Protocol 3 Basic HEAPINSERT protocol, inserting M^{AS} into the heap $\mathbf{D}^{AS}$ with n elements, preserving the heap property.

```
1: $n \leftarrow n + 1$
2: $\mathbf{D}^{AS}[n] = M^{AS}$
   // pi is the parent index of ci
3: $ci = n; pi = ci/2$
4: while $pi \neq 0$ do
     // Note these are accesses at public indices, not DORAM operations
5:   $(lt^{BS}, eq^{BS}, gt^{BS}) \leftarrow \text{Ocompare}(\mathbf{D}^{AS}[ci], \mathbf{D}^{AS}[pi])$
6:   $\text{Oswap}(\mathbf{D}^{AS}[ci], \mathbf{D}^{AS}[pi], lt^{BS})$
7:   $ci = pi$
8:   $pi = pi/2$
9: end while
```

Protocol 4 Optimized HEAPINSERT protocol, inserting M^{AS} into the heap $\mathbf{D}^{AS}$ with n elements, preserving the heap property.

```
1: $n \leftarrow n + 1$
   // The accessible set is the path from the root to the new node
2: $\mathbf{P} \leftarrow \text{Path}(\mathbf{D}, n)$
   // Do a binary search along P to locate the position where we need to write M
3: $i_F^{XS} \leftarrow \text{BINARYSEARCH}(\mathbf{P}^{AS}, M^{AS})$
   // Evaluate a DPF at the target location $i_F$, yielding shares of a vector like [0, 0, 1, 0, 0, 0] with the 1 in position $i_F$.
4: $\mathbf{t}^{BS} \leftarrow \text{EVALDPF}(i_F)$
   // Local computation yielding a vector like [0, 0, 1, 1, 1, 1]
5: $\mathbf{u}^{BS}[0] \leftarrow \mathbf{t}^{BS}[0]$
6: $\forall i > 0 : \mathbf{u}^{BS}[i] \leftarrow \mathbf{t}^{BS}[i] \oplus \mathbf{u}^{BS}[i-1]$
   // Flag-Word multiplications (parallel) to obliviously trickle down the entries in the path from $i_F$ on
7: $\mathbf{w}^{AS}[0] \leftarrow 0$
8: $\forall i > 0 : \mathbf{w}^{AS}[i] \leftarrow \mathbf{u}_b^{BS}[i-1] \cdot (\mathbf{P}^{AS}[i-1] - \mathbf{P}^{AS}[i])$
   // Flag-Word multiplications (also parallel) to obliviously write M into position $i_F$ of the path
9: $\forall i : \mathbf{v}[i]^{AS} \leftarrow \mathbf{t}_b^{BS}[i] \cdot (M^{AS} - \mathbf{P}^{AS}[i])$
   // Local operations to update the path entries
10: $\forall i : \mathbf{P}^{AS}[i] \leftarrow \mathbf{P}^{AS}[i] + (\mathbf{w}^{AS}[i] + \mathbf{v}^{AS}[i])$
```

each level we compare the current node's key with the operation (lookup/insert/delete) key, and choose the next node index from the left or right child pointers depending on the comparison result. All traversals on the tree have to be padded up to the maximum depth of the tree. (This is why a balanced tree is beneficial: a naive binary search tree has maximum depth n and so all traversals would have to take n steps because the computing parties do not know the depth of the tree.)

Notation. When the servers hold shares of an index in the AVL tree i^{XS}, by performing an ORAM read: $x \leftarrow \mathbf{D}[i^{XS}]$, the servers retrieve shares of the node structure x; Note that x is a structure composed of several shared variables. We refer to the key, value, children, and balance bits of this structure as $x_k^{AS}, x_v^{XS}, x_c^{XS}[]$, and $x_b^{XS}[]$ respectively. For ease of exposition, we will overload the notation to refer to both the children indices as x_c^{XS}, and similarly both the balance bits as x_b^{BS}. The index (or pointer) to the node x, we represent as $\vec{x}$, so $x \leftarrow \mathbf{D}[\vec{x}^{XS}]$ The balance bits for any node x, namely $x_b[0]$ ($x_b[1]$) indicate if the left (right) subtree has a greater depth than the other subtree or not. Note that $x_b[0]$ and $x_b[1]$ can never be 1 at the same time. At several points in our AVL algorithms, we read and write just the children pointers and balance fields of a node. For convenience we overload the read and write operations to represent this by just modifying the output (input) to capture the fields being read (written); for instance, reading just the balance bits and children pointers of a node we express as $(p_c^{XS}, p_b^{BS}) \leftarrow \mathbf{D}[\vec{p}^{XS}]$. Writes follow similarly, $\mathbf{D}[\vec{p}^{XS}] \leftarrow (p_c'^{XS}, p_b'^{BS})$. Additionally for better readability,

Protocol 5 Basic EXTRACTMIN protocol for the heap $\mathbf{D}^{AS}$ with n elements. The protocol returns shares of the minimum value in heap, deletes that value, and restores the heap property. The optimized EXTRACTMIN protocol is the same, except it uses an IDPF index $i^{XS,(p)}$ in place of smallerchild^{XS}, and calls the optimized HEAPIFY (Protocol 7).

```
1: $\text{min}^{AS} \leftarrow \mathbf{D}^{AS}[1]; \mathbf{D}^{AS}[1] \leftarrow \mathbf{D}^{AS}[n]; \mathbf{D}^{AS}[n] \leftarrow 0$
2: $n \leftarrow n - 1$
   // Restores the heap property at the root
3: $\text{smallerchild}^{XS} \leftarrow \text{HEAPIFY}(1, 0)$
4: $\text{depth} \leftarrow 1$
5: while $\text{depth} < \lfloor \lg n \rfloor$ do
     // HEAPIFY restores the heap property at that level
6:   $\text{smallerchild}^{XS} \leftarrow \text{HEAPIFY}(\text{smallerchild}^{XS}, \text{depth})$
7:   $\text{depth} \leftarrow \text{depth} + 1$
8: end while
   return $\text{min}^{AS}$
```

Protocol 6 Basic HEAPIFY protocol, taking as input an index share i^{XS} and the current depth d. Restores the heap property at that depth by obliviously swapping the node at that index with its smaller child (if that child is smaller than the node itself) and outputs the share of the index of the smaller child.

```
1: $pi^{XS} \leftarrow i^{XS}, li^{XS} \leftarrow 2 \cdot i^{XS}, ri^{XS} \leftarrow li^{XS} \oplus \text{party}$
   // Three DORAM reads in parallel
2: $\text{parent}^{AS} \leftarrow \mathbf{D}^{AS}[pi^{XS}]$
3: $\text{leftchild}^{AS} \leftarrow \mathbf{D}^{AS}[li^{XS}]$
4: $\text{rightchild}^{AS} \leftarrow \mathbf{D}^{AS}[ri^{XS}]$
5: $(lt_c^{BS}, eq_c^{BS}, gt_c^{BS}) \leftarrow \text{Ocompare}(\text{leftchild}^{AS}, \text{rightchild}^{AS})$
   // Compute the smaller child and its index in parallel
6: $\text{smaller}^{AS} \leftarrow \text{Oselect}(\text{rightchild}^{AS}, \text{leftchild}^{AS}, lt_c^{BS})$
7: $si^{XS} \leftarrow \text{Oselect}(ri^{XS}, li^{XS}, lt_c^{BS})$
8: $(lt_p^{BS}, eq_p^{BS}, gt_p^{BS}) \leftarrow \text{Ocompare}(\text{smaller}^{AS}, \text{parent}^{AS})$
   // MPC AND operation
9: $lt_{c,p}^{BS} \leftarrow lt_c^{BS} \wedge lt_p^{BS}$
   // Compute parent and left child offsets in parallel
10: $L \leftarrow (\text{parent} - \text{leftchild})^{AS} \cdot lt_{c,p}^{BS}$
11: $P \leftarrow (\text{smaller} - \text{parent})^{AS} \cdot lt_p^{BS}$
   // Three DORAM updates in parallel
12: $\mathbf{D}[i^{XS}] \leftarrow \mathbf{D}[i^{XS}] + P$
13: $\mathbf{D}[li^{XS}] \leftarrow \mathbf{D}[li^{XS}] + L$
14: $\mathbf{D}[ri^{XS}] \leftarrow \mathbf{D}[ri^{XS}] - (P + L)$
   return $si^{XS}$
```

Protocol 7 Optimized HEAPIFY protocol, taking as input a wide-IDPF index $\mathtt{i}^{\mathrm{XS},(\mathrm{p})}$ and the current depth d. Restores the heap property at that depth by obliviously swapping the node at that index with its smaller child (if that child is smaller than the node itself) and outputs the share of the index of the smaller child.

```
    // P contains all the nodes depth d
1:  $\mathbf{P} \leftarrow \mathsf{Stride}(\mathbf{D}, 1, 2^{\mathrm{d}})$
    // L contains all the left children at depth d + 1
2:  $\mathbf{L} \leftarrow \mathsf{Stride}(\mathbf{C}, 2, 2^{\mathrm{d}+1})$
    // R contains all the right children at depth d + 1
3:  $\mathbf{R} \leftarrow \mathsf{Stride}(\mathbf{C}, 2, 2^{\mathrm{d}+1} + 1)$
    // Three DORAM reads in parallel using a wide IDPF
4:  $\mathsf{parent}^{\mathrm{AS}} \leftarrow \mathbf{P}^{\mathrm{AS}}[\,\mathtt{i}^{\mathrm{XS},(\mathrm{p})}\,]$
5:  $\mathsf{leftchild}^{\mathrm{AS}} \leftarrow \mathbf{L}^{\mathrm{AS}}[\,\mathtt{i}^{\mathrm{XS},(\mathrm{p})}\,]$
6:  $\mathsf{rightchild}^{\mathrm{AS}} \leftarrow \mathbf{R}^{\mathrm{AS}}[\,\mathtt{i}^{\mathrm{XS},(\mathrm{p})}\,]$
7:  $(\mathtt{lt}_c^{\mathrm{BS}}, \mathsf{eq}_c^{\mathrm{BS}}, \mathsf{gt}_c^{\mathrm{BS}}) \leftarrow \mathsf{Ocompare}(\mathsf{leftchild}^{\mathrm{AS}}, \mathsf{rightchild}^{\mathrm{AS}})$
    // Compute smaller child
8:  $\mathsf{smaller}^{\mathrm{AS}} \leftarrow \mathsf{Oselect}(\mathsf{rightchild}^{\mathrm{AS}}, \mathsf{leftchild}^{\mathrm{AS}}, \mathtt{lt}^{\mathrm{BS}})$
9:  $(\mathtt{lt}_p^{\mathrm{BS}}, \mathsf{eq}_p^{\mathrm{BS}}, \mathsf{gt}_p^{\mathrm{BS}}) \leftarrow \mathsf{Ocompare}(\mathsf{smaller}^{\mathrm{AS}}, \mathsf{parent}^{\mathrm{AS}})$
    // MPC AND operation
10: $\mathtt{lt}_{c,p}^{\mathrm{BS}} \leftarrow \mathtt{lt}_c^{\mathrm{BS}} \land \mathtt{lt}_p^{\mathrm{BS}}$
    // Compute parent and left child offsets in parallel
11: $L \leftarrow (\mathsf{parent} - \mathsf{leftchild})^{\mathrm{AS}} \cdot \mathtt{lt}_{c,p}^{\mathrm{BS}}$
12: $P \leftarrow (\mathsf{smaller} - \mathsf{parent})^{\mathrm{AS}} \cdot \mathtt{lt}_p^{\mathrm{BS}}$
    // Three DORAM updates in parallel using a wide IDPF
13: $\mathbf{P}[\,\mathtt{i}^{\mathrm{XS},(\mathrm{p})}\,] \leftarrow \mathbf{P}[\,\mathtt{i}^{\mathrm{XS},(\mathrm{p})}\,] + P$
14: $\mathbf{L}[\,\mathtt{i}^{\mathrm{XS},(\mathrm{p})}\,] \leftarrow \mathbf{L}[\,\mathtt{i}^{\mathrm{XS},(\mathrm{p})}\,] + L$
15: $\mathbf{R}[\,\mathtt{i}^{\mathrm{XS},(\mathrm{p})}\,] \leftarrow \mathbf{R}[\,\mathtt{i}^{\mathrm{XS},(\mathrm{p})}\,] - (P + L)$
    // Append ($\mathtt{gt}_c \oplus \mathsf{eq}_c$) to the IDPF index
16: return $\mathtt{i}^{\mathrm{XS},(\mathrm{p}\|(\mathtt{gt}_c \oplus \mathsf{eq}_c))}$
```

we contract array dereferencing using shared flags with the short hand notation $\mathrm{x}_j^{\mathrm{XS}}[\mathrm{d}^{\mathrm{BS}}]$, which stands for Oselect ($\mathrm{x}_j^{\mathrm{XS}}[0]$, $\mathrm{x}_j^{\mathrm{XS}}[1]$, d^{BS}), where $j \in \{c, b\}$ (child pointer/balance bits of node x). Table 5 provides a succinct summary of the notations used for AVL trees.

C.1 Insertion

Protocol 8 details the main insert function Insert(), which internally calls a recursive insert RInsert(). To insert a new node x, x is first stored into the next available slot in the DORAM, and Insert() is invoked with x's index (i^{XS}) in the DORAM, its key ($\mathrm{x}_k^{\mathrm{AS}}$), and index of the current root of the AVL tree $\vec{\mathrm{r}}^{\mathrm{XS}}$. In an AVL insertion that results in an imbalance, a single rotation at the imbalance point on the path from the root to the newly inserted node resolves the imbalance. To insert obliviously, one might think a rotation (or dummy rotation) must be performed at every level of the traversal to hide the depth of the imbalance point. Instead one can 'lift' the rotation into a one-time operation after the insertion by storing the imbalance point during the traversal. If the insertion did not result in an imbalance, no rotation is needed. However to maintain obliviousness a dummy rotation is performed which operates over just the special $\mathbf{D}[0]$ node, emulating the memory accesses and computation of a real rotation.

Protocol 8 Oblivious AVL insert. $\mathtt{Insert}(\vec{\mathrm{r}}^{\mathrm{XS}}, \mathrm{i}^{\mathrm{XS}}, \mathrm{k}^{\mathrm{AS}}) \rightarrow \vec{\mathrm{r}}^{\mathrm{XS}}$. **Inputs**: i^{XS}: insertion index, and k^{AS}: insertion key.
Outputs: new root $\vec{\mathrm{r}}^{\mathrm{XS}}$.

```
1:  if $\mathbf{n} == 0$ then
2:      $\vec{\mathrm{r}}^{\mathrm{XS}} \leftarrow \mathrm{i}^{\mathrm{XS}}$
3:      $\mathbf{n}{+}{+}$
4:      return $\vec{\mathrm{r}}^{\mathrm{XS}}$
5:  else if $\mathbf{n} == 1$ then
6:      TTL $\leftarrow 1$
7:  else
8:      TTL $\leftarrow \lceil 1.44 \lg(\mathbf{n}) \rceil$
9:  end if
    // state $\mathrm{st}_I$: $\{\vec{g}^{\mathrm{XS}}, \vec{p}^{\mathrm{XS}}, \vec{c}^{\mathrm{XS}}, \mathrm{d}_{gp}^{\mathrm{BS}}, \mathrm{d}_{pc}^{\mathrm{BS}}, \mathrm{d}_{cn}^{\mathrm{BS}}, \mathrm{F}_{imb}^{\mathrm{BS}}, \mathrm{F}_g^{\mathrm{BS}}\}$
10: $\mathrm{st}_I \leftarrow \perp$
11: $(u^{\mathrm{BS}}, \perp, \mathrm{st}_I) \leftarrow \mathtt{RInsert}(\vec{\mathrm{r}}^{\mathrm{XS}}, \mathrm{i}^{\mathrm{XS}}, \mathrm{k}^{\mathrm{AS}}, \mathrm{TTL}, 0^{\mathrm{BS}}, \mathrm{st}_I)$
12: $\vec{\mathrm{r}}^{\mathrm{XS}} \leftarrow \mathtt{FixImb}_I(\mathrm{st}_I)$
13: $\mathbf{n}{+}{+}$
14: return $\vec{\mathrm{r}}^{\mathrm{XS}}$
```

Protocol 9 Oblivious recursive AVL insert function.
$\mathtt{RInsert}(\ell^{\mathrm{XS}}, \mathrm{i}^{\mathrm{XS}}, \mathrm{k}^{\mathrm{AS}}, \mathrm{TTL}, \mathrm{F}_d^{\mathrm{BS}}, \mathrm{st}_I) \rightarrow (\mathrm{u}^{\mathrm{BS}}, \mathrm{d}_r^{\mathrm{BS}}, \mathrm{st}_I)$
Inputs: ℓ^{XS}: index for this level, i^{XS}: insertion index, k^{AS}: insertion key, TTL: Time-To-Live, $\mathrm{F}_d^{\mathrm{BS}}$: dummy flag, st_I: insertion state structure.
Outputs: u^{BS}: update bit, $\mathrm{d}_r^{\mathrm{BS}}$: direction bit returned from recursion, st_I: updated state.

```
1:  if TTL = 0 then
2:      return $(0^{\mathrm{BS}}, 0^{\mathrm{BS}}, \mathrm{st}_I)$
3:  end if
4:  $nl \leftarrow \mathbf{D}[\ell^{\mathrm{XS}}]$                    ▷ nl: node at this level
5:  $(\mathrm{lt}^{\mathrm{BS}}, \mathrm{eq}^{\mathrm{BS}}, \mathrm{gt}^{\mathrm{BS}}) \leftarrow$ Ocompare $(\mathrm{nl}_k^{\mathrm{AS}}, \mathrm{k}^{\mathrm{AS}})$
6:  $\mathrm{np}^{\mathrm{XS}} \leftarrow$ Oselect $(nl_c{}^{\mathrm{XS}}[0], nl_c{}^{\mathrm{XS}}[1], \mathrm{gt}^{\mathrm{BS}})$
7:  $\mathrm{F}_i^{\mathrm{BS}} \leftarrow ((!\mathrm{F}_d^{\mathrm{BS}}) \cdot (\mathtt{IsZero}(\mathrm{np}^{\mathrm{XS}})))$
8:  $\mathrm{F}_d^{\mathrm{BS}} \leftarrow \mathrm{F}_d^{\mathrm{BS}} \oplus \mathrm{F}_i^{\mathrm{BS}}$
9:  $(\mathrm{u}^{\mathrm{BS}}, \mathrm{d}_r^{\mathrm{BS}}, \mathrm{st}_I) \leftarrow \mathtt{RInsert}(\mathrm{np}^{\mathrm{XS}}, \mathrm{i}^{\mathrm{XS}}, \mathrm{k}^{\mathrm{AS}}, \text{TTL-1}, \mathrm{F}_d^{\mathrm{BS}}, \mathrm{st}_I)$
    // If $F_i$, insert the node here, and set $u$ to 1
10: $\mathrm{nl}_c^{\mathrm{XS}} \leftarrow \mathtt{UpdCPtrs}(\mathrm{nl}_c^{\mathrm{XS}}, \mathrm{gt}^{\mathrm{BS}}, \mathrm{i}^{\mathrm{XS}}, \mathrm{F}_i^{\mathrm{BS}})$
11: $\mathrm{u}^{\mathrm{BS}} \leftarrow$ Oselect $(\mathrm{u}^{\mathrm{BS}}, 1^{\mathrm{BS}}, \mathrm{F}_i^{\mathrm{BS}})$
12: $(\mathrm{nl}_b^{\mathrm{XS}}, \mathrm{u}^{\mathrm{BS}}, \mathrm{F}_{imb}^{\mathrm{BS}}) \leftarrow \mathtt{UpdBal}_I(\mathrm{nl}_b^{\mathrm{BS}}, \mathrm{u}^{\mathrm{BS}}, \mathrm{gt}^{\mathrm{BS}})$
13: $\mathbf{D}[\ell^{\mathrm{XS}}] \leftarrow (\mathrm{nl}_b^{\mathrm{XS}}, \mathrm{nl}_c^{\mathrm{XS}})$
    // Lines 14-21 update $\mathrm{st}_I{}'$ to store the imbalance chain.
14: $\mathrm{st}_I.\mathrm{F}_{imb}^{\mathrm{BS}} \leftarrow \mathrm{st}_I.\mathrm{F}_{imb}^{\mathrm{BS}} \oplus \mathrm{F}_{imb}^{\mathrm{BS}}$
15: $\mathrm{st}_I.\vec{p}^{\mathrm{XS}} \leftarrow$ Oselect $(\mathrm{st}_I.\vec{p}^{\mathrm{XS}}, \ell^{\mathrm{XS}}, \mathrm{F}_{imb}^{\mathrm{BS}})$
16: $\mathrm{st}_I.\mathrm{d}_{pc}^{\mathrm{BS}} \leftarrow$ Oselect $(\mathrm{st}_I.\mathrm{d}_{pc}^{\mathrm{BS}}, \mathrm{gt}^{\mathrm{BS}}, \mathrm{F}_{imb}^{\mathrm{BS}})$
17: $\mathrm{st}_I.\vec{c}^{\mathrm{XS}} \leftarrow$ Oselect $(\mathrm{st}_I.\vec{c}^{\mathrm{XS}}, \mathrm{np}^{\mathrm{XS}}, \mathrm{F}_{imb}^{\mathrm{BS}})$
18: $\mathrm{st}_I.\mathrm{d}_{cn}^{\mathrm{BS}} \leftarrow$ Oselect $(\mathrm{st}_I.\mathrm{d}_{cn}^{\mathrm{BS}}, \mathrm{d}_r^{\mathrm{BS}}, \mathrm{F}_{imb}^{\mathrm{BS}})$
    // If $F_g$ (imbalance at lower level), update $\vec{g}$ and $d_{gp}$.
19: $\mathrm{st}_I.\vec{g}^{\mathrm{XS}} \leftarrow$ Oselect $(\mathrm{st}_I.\vec{g}^{\mathrm{XS}}, \ell^{\mathrm{XS}}, \mathrm{F}_g^{\mathrm{BS}})$
20: $\mathrm{st}_I.\mathrm{d}_{gp}^{\mathrm{BS}} \leftarrow$ Oselect $(\mathrm{st}_I.\mathrm{d}_{gp}^{\mathrm{BS}}, \mathrm{gt}^{\mathrm{BS}}, \mathrm{F}_g^{\mathrm{BS}})$
    // Set $F_g$ if the imbalance was at this level.
21: $\mathrm{st}_I.\mathrm{F}_g^{\mathrm{BS}} \leftarrow \mathrm{F}_{imb}^{\mathrm{BS}}$
22: return $(\mathrm{u}^{\mathrm{BS}}, \mathrm{d}_r^{\mathrm{BS}}, \mathrm{st}_I)$
```

Table 5: Table summarizing variables and notations used in Protocols 8 - 21.

Variables	Definition
TTL	Time-To-Live
F_x	Flag for x
st	a state object used to capture stateful variables
x	Node x
$\vec{x}$	Pointer to (index of) node x
x_c	children pointers for node x. $x_c[0]$: left child, $x_c[1]$: right child
x_b	balance bits of node x. $x_b[0]$: left balance, $x_b[1]$: right balance
g, p	grandparent node, parent node
c, n	child node, next node after child on path $g \to p \to c \to n$ represents a potential imbalance chain, with imbalance rooted at p
u	update bit. Semantically, for insertion $u \in \{0, 1\}$, and for deletion $u \in \{0, -1\}$
nl	node at this level of recursion
np	next pointer for traversing the tree
d_{xy}	The direction bit from $x \to y$. $xy \in \{gp, pc, cn\}$
d_r	The direction bit returned from recursion, i.e., direction for np in the recursive level

Global variables	Definition
$\mathbf{n}$	Number of items in the tree
$\vec{\mathbf{r}}$	Pointer to (index of) the tree root in $\mathbf{D}$

Protocol 10 `FixImb`$_I$. Fix the imbalance during insertion (if any). `FixImb`$_I(\mathrm{st}_I) \to (\vec{r}'^{\,\mathrm{XS}})$.
Inputs: st_I: state structure after `RInsert` that stores the imbalance chain (if any).
Outputs: $\vec{\mathbf{r}}^{\mathrm{XS}}$: new AVL root,

```
    // $F_{dr}$: flag for double rotation case (LR/RL).
 1: $\mathrm{F}^{\mathrm{BS}}_{dr} \leftarrow (\mathrm{st}_I.\mathrm{d}^{\mathrm{BS}}_{pc} \oplus \mathrm{st}_I.\mathrm{d}^{\mathrm{BS}}_{cn}) \cdot \mathrm{st}.\mathrm{F}^{\mathrm{BS}}_{imb}$
 2: $(\mathrm{g}^{\mathrm{XS}}_c, \mathrm{g}^{\mathrm{BS}}_b) \leftarrow \mathbf{D}[\mathrm{st}_I.\vec{g}^{\mathrm{XS}}], (\mathrm{p}^{\mathrm{XS}}_c, \mathrm{p}^{\mathrm{BS}}_b) \leftarrow \mathbf{D}[\mathrm{st}_I.\vec{p}^{\mathrm{XS}}]$
 3: $(\mathrm{c}^{\mathrm{XS}}_c, \mathrm{c}^{\mathrm{BS}}_b) \leftarrow \mathbf{D}[\mathrm{st}_I.\vec{c}^{\mathrm{XS}}]$
 4: $\vec{n}^{\mathrm{XS}} \leftarrow$ Oselect $(\mathrm{c}^{\mathrm{XS}}_c[0], \mathrm{c}^{\mathrm{XS}}_c[1], \mathrm{st}_I.\mathrm{d}^{\mathrm{BS}}_{cn})$
 5: $(\mathrm{n}^{\mathrm{XS}}_c, \mathrm{n}^{\mathrm{BS}}_b) \leftarrow \mathbf{D}[\vec{n}^{\mathrm{XS}}]$
 6: $(\vec{q}^{\mathrm{XS}}, \mathrm{c}^{\mathrm{XS}}_c, \mathrm{n}^{\mathrm{XS}}_c) \leftarrow$ ORotate$(\vec{c}^{\mathrm{XS}}, \mathrm{c}^{\mathrm{XS}}_c, \vec{n}^{\mathrm{XS}}, \mathrm{n}^{\mathrm{XS}}_c, \mathrm{st}_I.\mathrm{d}^{\mathrm{BS}}_{cn}, \mathrm{F}^{\mathrm{BS}}_{dr})$
 7: $\mathrm{p}^{\mathrm{XS}}_c \leftarrow$ UpdCPtrs$(\mathrm{p}^{\mathrm{XS}}_c, \mathrm{st}_I.\mathrm{d}^{\mathrm{BS}}_{pc}, \vec{q}^{\mathrm{XS}}, \mathrm{F}^{\mathrm{BS}}_{dr})$
    // If $F_{dr}$: Switch c and n before the next ORotate().
 8: $tmp\_c_c{}^{\mathrm{XS}} \leftarrow$ Oselect $(\mathrm{c}^{\mathrm{XS}}_c, n'^{\,\mathrm{XS}}_c, \mathrm{F}^{\mathrm{BS}}_{dr})$
 9: $tmp\_\vec{c}^{\mathrm{XS}} \leftarrow$ Oselect $(\vec{c}^{\mathrm{XS}}, \vec{n}^{\mathrm{XS}}, \mathrm{F}^{\mathrm{BS}}_{dr})$
10: $(\vec{m}^{\mathrm{XS}}, \mathrm{p}^{\mathrm{XS}}_c, tmp\_c_c{}^{\mathrm{XS}}) \leftarrow$ ORotate$(\vec{p}^{\mathrm{XS}}, \mathrm{p}^{\mathrm{XS}}_c, tmp\_\vec{c}^{\mathrm{XS}}, tmp\_c_c{}^{\mathrm{XS}},$
    $\mathrm{st}_I.\mathrm{d}^{\mathrm{BS}}_{pc}, \mathrm{F}^{\mathrm{BS}}_{imb})$
11: $\mathrm{g}^{\mathrm{XS}}_c \leftarrow$ UpdCPtrs$(\mathrm{g}^{\mathrm{XS}}_c, \mathrm{st}_I.\mathrm{d}^{\mathrm{BS}}_{pc}, \vec{m}^{\mathrm{XS}}, (!\mathrm{F}^{\mathrm{BS}}_g).\mathrm{F}^{\mathrm{BS}}_{imb})$
    // Return c and n's children pointers back correctly.
12: $\mathrm{n}^{\mathrm{XS}}_c \leftarrow$ Oselect $(\mathrm{n}^{\mathrm{XS}}_c, tmp\_c_c{}^{\mathrm{XS}}, \mathrm{F}^{\mathrm{BS}}_{dr})$
13: $\mathrm{c}^{\mathrm{XS}}_c \leftarrow$ Oselect $(tmp\_c'^{\,\mathrm{XS}}_c, \mathrm{c}^{\mathrm{XS}}_c, \mathrm{F}^{\mathrm{BS}}_{dr})$
14: $(\mathrm{p}^{\mathrm{BS}}_b, \mathrm{c}^{\mathrm{BS}}_b, \mathrm{n}^{\mathrm{BS}}_b) \leftarrow$ FixBal$_I(\mathrm{p}^{\mathrm{BS}}_b, \mathrm{c}^{\mathrm{BS}}_b, \mathrm{n}^{\mathrm{BS}}_b, \mathrm{st}_I.\mathrm{d}^{\mathrm{BS}}_{pc}, \mathrm{F}^{\mathrm{BS}}_{imb}, \mathrm{F}^{\mathrm{BS}}_{dr})$
15: $\mathbf{D}[\mathrm{st}_I.\vec{g}^{\mathrm{XS}}] \leftarrow \mathrm{g}^{\mathrm{XS}}_c, \mathbf{D}[\mathrm{st}_I.\vec{p}^{\mathrm{XS}}] \leftarrow (\mathrm{p}^{\mathrm{XS}}_c, \mathrm{p}^{\mathrm{BS}}_b)$
16: $\mathbf{D}[\mathrm{st}_I.\vec{c}^{\mathrm{XS}}] \leftarrow (\mathrm{c}^{\mathrm{XS}}_c, \mathrm{c}^{\mathrm{BS}}_b), \mathbf{D}[\vec{n}^{\mathrm{XS}}] \leftarrow (\mathrm{n}^{\mathrm{XS}}_c, \mathrm{n}^{\mathrm{BS}}_b)$
17: $\mathrm{t}^{\mathrm{XS}} \leftarrow$ Oselect $(\vec{c}^{\mathrm{XS}}, \vec{n}^{\mathrm{XS}}, \mathrm{F}^{\mathrm{BS}}_{dr})$
    // If $F_g$, update root
18: $\vec{\mathbf{r}}^{\mathrm{XS}} \leftarrow$ Oselect $(\vec{\mathbf{r}}^{\mathrm{XS}}, \mathrm{t}^{\mathrm{XS}}, \mathrm{st}_I.\mathrm{F}^{\mathrm{BS}}_g)$
19: return $(\vec{\mathbf{r}}^{\mathrm{XS}})$
```

Protocol 11 `ORotate`. Performs a single parent-child rotation as illustrated in Figure 11, if flag $\mathrm{F}^{\mathrm{BS}}_{imb} = 1$. If $\mathrm{F}^{\mathrm{BS}}_{imb} = 0$, `ORotate` has no effect. `ORotate`$(\vec{p}^{\mathrm{XS}}, \mathrm{p}^{\mathrm{XS}}_c, \vec{c}^{\mathrm{XS}}, \mathrm{c}^{\mathrm{XS}}_c, \mathrm{d}^{\mathrm{BS}}_{pc}, \mathrm{F}^{\mathrm{BS}}_{imb}) \to (\vec{q}^{\mathrm{XS}}, p'^{\,\mathrm{XS}}_c, c'^{\,\mathrm{XS}}_c)$.
Inputs: $\vec{p}^{\mathrm{XS}}$: parent (p) pointer, $\mathrm{p}^{\mathrm{XS}}_c$: parent's children pointers, $\vec{c}^{\mathrm{XS}}$: child (c) pointer, $\mathrm{c}^{\mathrm{XS}}_c$: c's children pointers, $\mathrm{d}^{\mathrm{BS}}_{pc}$: direction bit from p to c, $\mathrm{F}^{\mathrm{BS}}_{imb}$: imbalance flag.
Outputs: $\vec{q}^{\mathrm{XS}}$: pointer to the node in p's position after rotation, $\mathrm{p}^{\mathrm{XS}}_c$: updated children pointers for p, $\mathrm{c}^{\mathrm{XS}}_c$: updated children pointers for c.

```
1: $\mathrm{p}^{\mathrm{XS}}_c[\mathrm{d}^{\mathrm{BS}}_{pc}] \leftarrow$ Oselect $(\mathrm{p}^{\mathrm{XS}}_c[\mathrm{d}^{\mathrm{BS}}_{pc}], \mathrm{c}^{\mathrm{XS}}_c[!\mathrm{d}^{\mathrm{BS}}_{pc}], \mathrm{F}^{\mathrm{BS}}_{imb})$
2: $\mathrm{c}^{\mathrm{XS}}_c[!\mathrm{d}^{\mathrm{BS}}_{pc}] \leftarrow$ Oselect $(\mathrm{c}^{\mathrm{XS}}_c[!\mathrm{d}^{\mathrm{BS}}_{pc}], \vec{p}^{\mathrm{XS}}, \mathrm{F}^{\mathrm{BS}}_{imb})$
3: $\vec{q}^{\mathrm{XS}} \leftarrow$ Oselect $(\vec{p}^{\mathrm{XS}}, \vec{c}^{\mathrm{XS}}, \mathrm{F}^{\mathrm{BS}}_{imb})$
4: return $(\vec{q}^{\mathrm{XS}}, \mathrm{p}^{\mathrm{XS}}_c, \mathrm{c}^{\mathrm{XS}}_c)$
```

Protocol 12 `UpdCPtrs`. Update Child Pointers of a node.
`UpdCPtrs`$(\mathrm{nl}^{\mathrm{XS}}_c, \mathrm{d}^{\mathrm{BS}}, \vec{q}^{\mathrm{XS}}, F_r = 1^{\mathrm{BS}}) \to \mathrm{nl}^{\mathrm{XS}}_c$
Inputs: $\mathrm{nl}^{\mathrm{XS}}_c$: current child pointers of nl, d^{BS}: direction of next node, $\vec{q}^{\mathrm{XS}}$: new pointers to update $\mathrm{nl}^{\mathrm{XS}}_c[d]$ with, F_r: optional real/dummy (1/0) flag
Outputs: $\mathrm{nl}^{\mathrm{XS}}_c$: updated child pointers

```
1: $\mathrm{nl}^{\mathrm{XS}}_c[1] \leftarrow$ Oselect $(\mathrm{nl}^{\mathrm{XS}}_c[1], \vec{q}^{\mathrm{XS}}, \mathrm{d}^{\mathrm{BS}} \cdot F_r)$
2: $\mathrm{nl}^{\mathrm{XS}}_c[0] \leftarrow$ Oselect $(\mathrm{nl}^{\mathrm{XS}}_c[0], \vec{q}^{\mathrm{XS}}, (!\mathrm{d}^{\mathrm{BS}}) \cdot F_r)$
3: return $(\mathrm{nl}^{\mathrm{XS}}_c)$
```

Protocol 13 `RSBal` (Right Shift Balances) and `LSBal` (Shift Balances).
`RSBal`$(\mathrm{p}^{\mathrm{BS}}_b, \mathrm{F}^{\mathrm{BS}}_b, \mathrm{F}^{\mathrm{BS}}_{imb}, \mathrm{F}^{\mathrm{BS}}_{rs}) \to (\mathrm{p}^{\mathrm{BS}}_b, \mathrm{F}^{\mathrm{BS}}_{imb})$
`LSBal`$(\mathrm{p}^{\mathrm{BS}}_b, \mathrm{F}^{\mathrm{BS}}_b, \mathrm{F}^{\mathrm{BS}}_{imb}, \mathrm{F}^{\mathrm{BS}}_{ls}) \to (p'^{\,\mathrm{BS}}_b, \mathrm{F}^{\mathrm{BS}}_{imb})$
Inputs: $\mathrm{p}^{\mathrm{BS}}_b$: balance bits of node p, $\mathrm{F}^{\mathrm{BS}}_b$: flag for if this node is balanced, $\mathrm{F}^{\mathrm{BS}}_{imb}$: current imbalance flag, $\mathrm{F}^{\mathrm{BS}}_{rs}$: flag to right shift balances (or $\mathrm{F}^{\mathrm{BS}}_{ls}$: flag to left shift balances for `LSBal`)
Output: $\mathrm{p}^{\mathrm{BS}}_b$: new balance bits for node p, $\mathrm{F}^{\mathrm{BS}}_{imb}$: updated imbalance flag.

```
    // If $F_{rs}$, RSBal() performs a right shift:
    // Right shift chain: $p_b[0] \to F_b \to p_b[1] \to F_{imb}$
1: RSBal$(\mathrm{p}^{\mathrm{BS}}_b, \mathrm{F}^{\mathrm{BS}}_b, \mathrm{F}^{\mathrm{BS}}_{rs}, \mathrm{F}^{\mathrm{BS}}_{imb}) \to (\mathrm{p}^{\mathrm{BS}}_b, \mathrm{F}^{\mathrm{BS}}_{imb})$:
2:     $\mathrm{F}^{\mathrm{BS}}_{imb} \leftarrow$ Oselect $(\mathrm{F}^{\mathrm{BS}}_{imb}, \mathrm{p}^{\mathrm{BS}}_b[1], \mathrm{F}^{\mathrm{BS}}_{rs})$
3:     $\mathrm{p}^{\mathrm{BS}}_b[1] \leftarrow$ Oselect $(\mathrm{p}^{\mathrm{BS}}_b[1], \mathrm{F}^{\mathrm{BS}}_b, \mathrm{F}^{\mathrm{BS}}_{rs})$
4:     $\mathrm{F}^{\mathrm{BS}}_b \leftarrow$ Oselect $(\mathrm{F}^{\mathrm{BS}}_b, \mathrm{p}^{\mathrm{BS}}_b[0], \mathrm{F}^{\mathrm{BS}}_{rs})$
5:     $\mathrm{p}^{\mathrm{BS}}_b[0] \leftarrow$ Oselect $(\mathrm{p}^{\mathrm{BS}}_b[0], 0^{\mathrm{BS}}, \mathrm{F}^{\mathrm{BS}}_{rs})$
6:     return $(\mathrm{p}^{\mathrm{BS}}_b, \mathrm{F}^{\mathrm{BS}}_{imb})$

    // LSBal$(\mathrm{p}^{\mathrm{BS}}_b, \mathrm{F}^{\mathrm{BS}}_b, \mathrm{F}^{\mathrm{BS}}_{ls}, \mathrm{F}^{\mathrm{BS}}_{imb}) \to (\mathrm{p}^{\mathrm{BS}}_b, \mathrm{F}^{\mathrm{BS}}_{imb})$
    // Similar to RSBal, except left shift if $\mathrm{F}^{\mathrm{BS}}_{ls}$:
    // Left shift chain: $F_{imb} \leftarrow p_b[0] \leftarrow F_b \leftarrow p_b[1]$
```

To store the imbalance point, let *state* st_Ibe a structure containing index shares $\vec{g}^{\mathrm{XS}}, \vec{p}^{\mathrm{XS}}, \vec{c}^{\mathrm{XS}}$(grandparent, parent, and child respectively of the imbalance chain, with the imbalance rooted at p), $\mathrm{d}^{\mathrm{BS}}_{gp}, \mathrm{d}^{\mathrm{BS}}_{pc}$(the shares of bit flags of direction from g to p, and p to c), $\mathrm{d}^{\mathrm{BS}}_{cn}$(shares of direction from the child to next node on traversal path) and $\mathrm{F}^{\mathrm{BS}}_{imb}$ a shared bit flag indicating if the insertion resulted

Protocol 14 Update Balance for AVL Insert function. $\texttt{UpdBal}_I(\mathrm{p}_b^{\mathrm{BS}}, \mathrm{u}^{\mathrm{BS}}, \mathrm{gt}^{\mathrm{BS}}) \rightarrow (\mathrm{p}_b^{\mathrm{BS}}, \mathrm{u}^{\mathrm{BS}}, \mathrm{F}_{imb}^{\mathrm{BS}})$
Inputs: $\mathrm{p}_b^{\mathrm{BS}}$: balance bits of current node p, u^{BS}: update bit, $\mathrm{gt}^{\mathrm{BS}}$: direction of update bit, i.e., is the update coming from the left or right child
Outputs: $\mathrm{p}_b^{\mathrm{BS}}$: updated balance bits for p, u^{BS}: new update bit, $\mathrm{F}_{imb}^{\mathrm{BS}}$: imbalance flag.

```
    // Updates balances for the current node $p$ on the path from root to inserted node. In case of imbalance $\texttt{FixBal}_I$ completes the balance update resulting from rotations.
1:  $\mathrm{F}_{rs}^{\mathrm{BS}} \leftarrow u^{\mathrm{BS}} \cdot gt^{\mathrm{BS}}$
2:  $\mathrm{F}_{ls}^{\mathrm{BS}} \leftarrow u^{\mathrm{BS}} \cdot !gt^{\mathrm{BS}}$
3:  $\mathrm{F}_b^{\mathrm{BS}} \leftarrow \mathrm{p}_b^{\mathrm{BS}}[0] \oplus \mathrm{p}_b^{\mathrm{BS}}[1]$
4:  $(\mathrm{p}_b^{\mathrm{BS}}, \mathrm{F}_{imb}^{\mathrm{BS}}) \leftarrow \texttt{RSBal}(\mathrm{p}_b^{\mathrm{BS}}, \mathrm{F}_b^{\mathrm{BS}}, 0^{\mathrm{BS}}, \mathrm{F}_{rs}^{\mathrm{BS}})$
5:  $(\mathrm{p}_b^{\mathrm{BS}}, \mathrm{F}_{imb}^{\mathrm{BS}}) \leftarrow \texttt{LSBal}(\mathrm{p}_b^{\mathrm{BS}}, \mathrm{F}_b^{\mathrm{BS}}, \mathrm{F}_{imb}^{\mathrm{BS}}, \mathrm{F}_{ls}^{\mathrm{BS}})$
6:  $\mathrm{F}_b^{\mathrm{BS}} \leftarrow \mathrm{p}_b^{\mathrm{BS}}[0]^{\mathrm{BS}} \oplus \mathrm{p}_b^{\mathrm{BS}}[1]$
7:  $\mathrm{F}_{bu}^{\mathrm{BS}} \leftarrow (\mathrm{F}_b^{\mathrm{BS}} \cdot u^{\mathrm{BS}})$
    // If update results in this node being balanced ($\mathrm{F}_{bu}^{\mathrm{BS}}$), or an imbalance (${F'_{imb}}^{\mathrm{BS}}$). Then $u \leftarrow 0$; no more height changes to propagate in either case.
8:  $\mathrm{u}^{\mathrm{BS}} \leftarrow$ Oselect $(\mathrm{u}^{\mathrm{BS}}, 0^{\mathrm{BS}}, \mathrm{F}_{bu}^{\mathrm{BS}})$
9:  $\mathrm{u}^{\mathrm{BS}} \leftarrow$ Oselect $(u^{\mathrm{BS}}, 0^{\mathrm{BS}}, \mathrm{F}_{imb}^{\mathrm{BS}})$
10: **return** $(\mathrm{p}_b^{\mathrm{BS}}, \mathrm{u}^{\mathrm{BS}}, \mathrm{F}_{imb}^{\mathrm{BS}})$
```

Protocol 15 Fix Balance for AVL Insert function. $\texttt{FixBal}_I(\mathrm{p}_b^{\mathrm{BS}}, \mathrm{c}_b^{\mathrm{BS}}, \mathrm{n}_b^{\mathrm{BS}}, \mathrm{d}_{pc}^{\mathrm{BS}}, \mathrm{F}_{imb}^{\mathrm{BS}}, \mathrm{F}_{dr}^{\mathrm{BS}}) \rightarrow (\mathrm{p}_b^{\mathrm{BS}}, \mathrm{c}_b^{\mathrm{BS}}, \mathrm{n}_b^{\mathrm{BS}})$
Inputs: $\mathrm{p}_b^{\mathrm{BS}}$: parent($p$)'s balance bits, $\mathrm{c}_b^{\mathrm{BS}}$: child($c$)'s balance bits, $\mathrm{n}_b^{\mathrm{BS}}$: balance bits for next node after child on the insertion path (n), $\mathrm{d}_{pc}^{\mathrm{BS}}$: direction bit from p to c, $\mathrm{F}_{imb}^{\mathrm{BS}}$: imbalance flag, $\mathrm{F}_{dr}^{\mathrm{BS}}$: double rotation flag.
Outputs: $\mathrm{p}_b^{\mathrm{BS}}, \mathrm{c}_b^{\mathrm{BS}}, \mathrm{n}_b^{\mathrm{BS}}$: Updated balance bits for p, c, and n.

```
    // Balance updates from rotation result in $p_b$ and $c_b$being set to zeroes. (Note $p_b$ is already set to 0 by $\texttt{UpdBal}_I$ if there was an imbalance.)
1:  $\mathrm{c}_b^{\mathrm{BS}} \leftarrow$ Oselect $(\mathrm{c}_b^{\mathrm{BS}}, 0^{\mathrm{BS}}, \mathrm{F}_{imb}^{\mathrm{BS}})$
    // $n_b$ stays same, except in a double rotation case $n_b \leftarrow 0$.
2:  $\mathrm{n}_b^{\mathrm{BS}} \leftarrow$ Oselect $(\mathrm{n}_b^{\mathrm{BS}}, 0^{\mathrm{BS}}, \mathrm{F}_{dr}^{\mathrm{BS}})$
3:  ${F_{nl0}}^{\mathrm{BS}} \leftarrow \texttt{IsZero}(\mathrm{n}_c^{\mathrm{XS}}[0])$
4:  ${F_{nr0}}^{\mathrm{BS}} \leftarrow \texttt{IsZero}(\mathrm{n}_c^{\mathrm{XS}}[1])$
    // $F_{nc}$: does n have children. If $F_{nc}$ and $F_{dr}$, $p_b$ and $c_b$ have their balances tweaked to account for $n$'s children.
5:  $\mathrm{F}_{nc}^{\mathrm{BS}} \leftarrow !({F_{nl0}}^{\mathrm{BS}} \cdot {F_{nr0}}^{\mathrm{BS}})$
6:  $\mathrm{F}_{pcu}^{\mathrm{BS}} \leftarrow \mathrm{F}_{dr}^{\mathrm{BS}} \cdot \mathrm{F}_{nc}^{\mathrm{BS}}$
7:  $\mathrm{p}_b^{\mathrm{BS}}[!\mathrm{d}_{pc}^{\mathrm{BS}}] \leftarrow$ Oselect $(\mathrm{p}_b^{\mathrm{BS}}[!\mathrm{d}_{pc}^{\mathrm{BS}}], !(\mathrm{n}_b^{\mathrm{BS}}[!\mathrm{d}_{pc}^{\mathrm{BS}}]), \mathrm{F}_{pcu}^{\mathrm{BS}})$
8:  $\mathrm{c}_b^{\mathrm{BS}}[\mathrm{d}_{pc}^{\mathrm{BS}}] \leftarrow$ Oselect $(\mathrm{c}_b^{\mathrm{BS}}[\mathrm{d}_{pc}^{\mathrm{BS}}], !(\mathrm{n}_b^{\mathrm{BS}}[\mathrm{d}_{pc}^{\mathrm{BS}}]), \mathrm{F}_{pcu}^{\mathrm{BS}})$
```

in an AVL imbalance. The imbalance point is the parent p, and the rotation happens over the chain $p \rightarrow c \rightarrow c_c[d_{cn}]$, but we need to track the grandparent node as well, since any rotation on the $p \rightarrow c \rightarrow c_c[d_{cn}]$, requires updating $g_c[d_{gp}]$ link to point to the new node occupying the former parent node's position after the rotation.

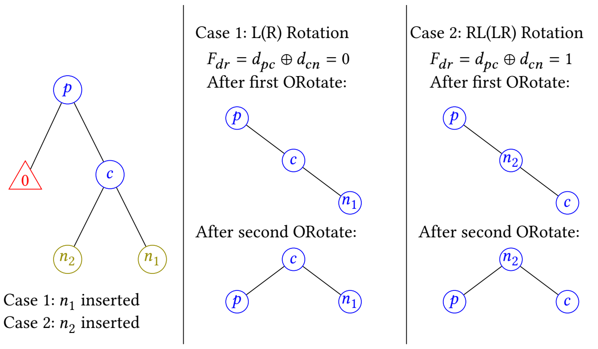


Figure 10: AVL insertion imbalance rotations. The figure shows the L and RL rotation case. The mirror images of L and RL rotations correspond to R and LR rotation cases respectively.

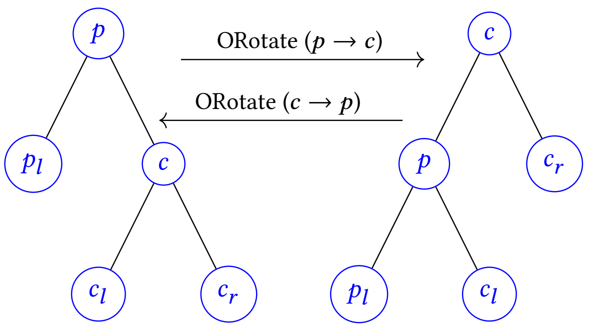


Figure 11: The single AVL rotation operation performed by ORotate().

We first look at the recursive insert detailed in Protocol 9. At each level of the tree, we compare the insertion key with the key of the current node at this level and pick the next index to iterate to depending on the comparison result (lines 4–6), just like how an item lookup is performed. Line 7 sets the flag $\mathrm{F}_i^{\mathrm{BS}}$, which dictates if the insertion happens at this level or not. $\mathrm{F}_i^{\mathrm{BS}}$ checks if the operation is not a dummy and that the next index is 0; i.e., no more real nodes along the path in this direction. Line 8 toggles the operation to a dummy if we find the point to insert at, and in line 9, the function recurses. On the path back from the recursion, line 10 completes the insertion by updating the child pointers at the level where $\mathrm{F}_i^{\mathrm{BS}}$ is true, and line 12 triggers the update bit propagation as the height has increased by one at this node. Finally, lines 16–21 obliviously store the chain of nodes that has an imbalance (if any arises from this insertion). This recorded state is then used to complete imbalance rotation to restore the AVL property of the tree in lines 8 to 24 of `Insert()`, and Figure 10 illustrates the the insertion imbalance rotations.

C.1.1 Fixing Imbalance in `Insert()`*.* In the case of any imbalance (during insertion or deletion), imbalances are fixed via one of four possible rotations referred to as L, R, LR, or RL rotations, where the former two have exactly one rotation, while the latter two have two rotations. Figure 10 illustrates the L and RL rotations; the R and LR rotations are mirror images of the rotations shown in the figure. Hence, in the context of oblivious AVL trees, all imbalance fix procedures perform two rotations (padded up with dummy rotations if only one or no rotations are needed) to hide if a rotation happened and the rotation type. Each of these rotations transforms the chain $p \longrightarrow c$ to $c \longrightarrow p$ in the tree, i.e., rotating the parent and child nodes such that their parent-child relationship gets swapped by modifying their children pointers. Figure 11 illustrates the single rotation performed by an `ORotate`.

In line 8 in `Insert()`, once `RInsert()` completes, `Insert()` invokes $\texttt{FixImb}_I()$ in Protocol 10 to fix the imbalance if any. $\texttt{FixImb}_I$ first computes if the imbalance (if any) requires a double rotation (LR/RL) in line 1. Lines 2–5 read all the child pointers and balances of the nodes involved in the imbalance chain; note that this read does not require reading the keys and the values of each node in this chain. As explained earlier, any imbalance is resolved with up to two rotations, and each rotation is performed by invoking the `ORotate` protocol. `ORotate` operates over shares of the children pointers (p_c, c_c), the node pointers ($\vec{p}$ and $\vec{c}$), the direction bits (d_{gp}, d_{pc}), and flag $\mathrm{F}^{\mathrm{BS}}_{imb}$. The internals of `ORotate` are detailed in Protocol 11, and perform the operations depicted in Figure 11. Lines 8–9 and 12–13 in $\texttt{FixImb}_I$ bridge the two rotations together in the case of a double rotation. In the case of a double rotation after the first rotations, the chain turns from ($p \longrightarrow c \longrightarrow n$) to ($p \longrightarrow n \longrightarrow c$), so we swap the node and child pointers for c and n before the second rotation in lines 8–9 and then update n and c's final child pointers correctly in lines 12–13. To complete the rotation, we need to update the balances of p, c, and n to account for their final position. This is handled by $\texttt{FixBal}_I$ detailed in Algorithm 15, and these updated balances and children pointers are all written back into the DORAM in lines 15–16. Finally, lines 17–18 update and return this AVL tree's new root if the imbalance chain began at the root.

C.2 Deletion

Since deleting a node can lead to imbalances at possibly every level on the path from the root to the deletion node (or more precisely until its successor), making it oblivious requires that we perform rotations at all nodes along this path lest we leak the depth of the node deleted. So unlike insertion where we could lift the imbalance-fixing rotation into a one-time operation, here it has to be performed at every level. Protocol 16 presents the main `Delete` function, which internally invokes the recursive delete operation `RDelete`in Protocol 17. Unlike `RInsert` where the traversal direction is dictated by just the comparison of keys, in `RDelete` there are four cases (F_{C1}–F_{C4}) that modify this traversal direction captured by lines 6–12.

- F_{C1}: Node to delete was found ∧ current node has only one child. In this case, to delete we simply skip the node at this level.
- F_{C2}: Node to delete was found ∧ current node has both children. This case corresponds to finding the successor and swapping the successor node with the deletion node, and removing the successor from the tree. To find the successor, we need to find the smallest item greater than this deletion key, so we go to the right subtree and then traverse to find the leftmost (smallest) item in it.

The latter two cases are really subcases of F_{C2} since they deal with handling the successor search.

- F_{C3}: Finding successor ∧ current node has a left child. In this case we traverse left, since the objective is to find the node with smallest key greater than the deletion key.
- F_{C4}: Finding successor ∧ current node has no left child. This means we are at the successor node. Since the successor node would be deleted (by skipping it, like case F_{C1}), we traverse down the only (right) child path or just dummy traversal if successor was a leaf node.

Protocol 16 Oblivious AVL delete. As convention, `Delete` deletes the first node in the tree (in case of repeated entries) that matches the deletion key. $\texttt{Delete}(\vec{\mathrm{r}}^{\mathrm{XS}}, k^{\mathrm{AS}}) \rightarrow (\vec{\mathrm{r}}^{\mathrm{XS}}, F_f)$
Input: k^{AS}: deletion key.
Outputs: F_f: flag for if deletion was successful, $\vec{\mathrm{r}}^{\mathrm{XS}}$: new root

```
1:  if $\mathbf{n} == 0$ then
2:      return $(\perp, 0)$
3:  else if $\mathbf{n} == 1$ then
4:      $nl \leftarrow \mathbf{D}[\vec{\mathrm{r}}^{\mathrm{XS}}]$
5:      $(lt^{\mathrm{BS}}, eq^{\mathrm{BS}}, gt^{\mathrm{BS}}) \leftarrow$ Ocompare $(\mathrm{nl}^{\mathrm{AS}}_k, k^{\mathrm{AS}})$
6:      $F_f \leftarrow$ Reconstruct$(eq^{\mathrm{BS}})$
7:      if $F_f == 1$ then
8:          $\mathbf{n} \leftarrow 0$
9:          return $(\perp, F_f)$
10:     else
11:         return $(\vec{\mathrm{r}}^{\mathrm{XS}}, F_f)$
12:     end if
13: else
14:     $\mathrm{TTL} \leftarrow \lceil 1.44 \lg(\mathbf{n}) \rceil$;
15: end if
    // state $\mathrm{st}_D$: $\{\mathrm{F}^{\mathrm{BS}}_f, \mathrm{F}^{\mathrm{BS}}_{fs}, \mathrm{F}^{\mathrm{BS}}_{ss}, \vec{nd}^{\mathrm{XS}}, \vec{ns}^{\mathrm{XS}}, \mathrm{F}^{\mathrm{BS}}_r, \mathrm{ri}^{\mathrm{XS}}\}$
16: $\mathrm{st}_D \leftarrow \perp$
17: $(F_f, \vec{\mathrm{r}}^{\mathrm{XS}}, \mathrm{st}_D) \leftarrow$ RDelete$(\vec{\mathrm{r}}^{\mathrm{XS}}, k^{\mathrm{AS}}, \mathrm{TTL}, \mathrm{st}_D)$
18: if $F_f == 0$ then
19:     return $(F_f, \vec{\mathrm{r}}^{\mathrm{XS}})$
20: else
21:     $\mathbf{n} - -$
    // Swap successor and node to delete (if applicable)
22:     $(\mathrm{nd}^{\mathrm{XS}}_k, \mathrm{nd}^{\mathrm{XS}}_v) \leftarrow \mathbf{D}[\mathrm{st}_D.\vec{nd}^{\mathrm{XS}}]$
23:     $(\mathrm{ns}^{\mathrm{XS}}_k, \mathrm{ns}^{\mathrm{XS}}_v) \leftarrow \mathbf{D}[\mathrm{st}_D.\vec{ns}^{\mathrm{XS}}]$
    // If $(F_{ss})$: $\mathbf{D}[\mathrm{st}_D.\vec{nd}^{\mathrm{XS}}] \leftarrow \perp$; else $\mathbf{D}[\mathrm{st}_D.\vec{ns}^{\mathrm{XS}}] \leftarrow \perp$
24:     $\mathrm{nd}^{\mathrm{XS}}_k \leftarrow$ Oselect $(\mathrm{nd}^{\mathrm{XS}}_k, \mathrm{ns}^{\mathrm{XS}}_k, \mathrm{F}^{\mathrm{BS}}_{ss})$
25:     $\mathrm{nd}^{\mathrm{XS}}_v \leftarrow$ Oselect $(\mathrm{nd}^{\mathrm{XS}}_v, \mathrm{ns}^{\mathrm{XS}}_v, \mathrm{F}^{\mathrm{BS}}_{ss})$
26:     $\mathbf{D}[\mathrm{st}_D.\vec{nd}^{\mathrm{XS}}] \leftarrow (\mathrm{nd}^{\mathrm{XS}}_k, \mathrm{nd}^{\mathrm{XS}}_v)$
    // The index shares $\mathrm{st}_D.\vec{nd}^{\mathrm{XS}}$or $\mathrm{st}_D.\vec{ns}^{\mathrm{XS}}$is obliviously
    // added to a queue of empty locations for future inserts.
27:     $\vec{\mathrm{r}}^{\mathrm{XS}} \leftarrow$ Oselect $(\vec{\mathrm{r}}^{\mathrm{XS}}, ri, \mathrm{st}_D.\mathrm{F}^{\mathrm{BS}}_r)$
28:     return $(F_f, \vec{\mathrm{r}}^{\mathrm{XS}})$
29: end if
```

Protocol 17 Oblivious AVL recursive delete.
`RDelete`$(\ell^{\mathrm{XS}}, \mathrm{k}^{\mathrm{AS}}, \mathrm{TTL}, \mathrm{st}_D) \rightarrow (F_f, \mathrm{u}^{\mathrm{BS}}, \mathrm{st}_D)$
Inputs: ℓ^{XS}: current index, k^{AS}: deletion key, TTL: Time-To-Live, st_D: deletion state.
Outputs: F_f: flag for deletion success, u^{BS}: balance update bit, st_D: updated state

```
1:  if TTL = 0 then
2:      return (RECONSTRUCT($\mathrm{st}_D.\mathrm{F}_f^{\mathrm{BS}}$), $0^{\mathrm{BS}}$, $\mathrm{st}_D$)
3:  end if
4:  $nl \leftarrow \mathbf{D}[\ell^{\mathrm{XS}}]$                                ▷nl: node at this level
5:  $(lt^{\mathrm{BS}}, eq^{\mathrm{BS}}, gt^{\mathrm{BS}}) \leftarrow$ Ocompare $(\mathrm{nl}_k^{\mathrm{AS}}, \mathrm{k}^{\mathrm{AS}})$
    // Unlike RInsert, the direction to traverse is altered by 4 possible special cases (C1-C4) in RDelete.
    // $F_{lf}$: is this the first instance of deletion key
6:  $\mathrm{F}_{lf}^{\mathrm{BS}} \leftarrow eq^{\mathrm{BS}} \cdot !(\mathrm{st}_D.\mathrm{F}_f^{\mathrm{BS}})$
    // $l0$: is left child empty, $r0$: is right child empty
7:  $\mathrm{l0}^{\mathrm{BS}} \leftarrow$ IsZero$(nl_c[0]^{\mathrm{XS}})$, $\mathrm{r0}^{\mathrm{BS}} \leftarrow$ IsZero$(nl_c[1]^{\mathrm{XS}})$
    // The number of children at this level ($F_0$, $F_1$, $F_2$).
8:  $\mathrm{F}_0^{\mathrm{BS}} \leftarrow \mathrm{l0}^{\mathrm{BS}} \cdot \mathrm{r0}^{\mathrm{BS}}$, $\mathrm{F}_1^{\mathrm{BS}} \leftarrow \mathrm{l0}^{\mathrm{BS}} \oplus \mathrm{r0}^{\mathrm{BS}}$, $\mathrm{F}_{dh}^{\mathrm{BS}} \leftarrow \mathrm{F}_0^{\mathrm{BS}} \cdot \mathrm{F}_{lf}^{\mathrm{BS}}$
9:  $\mathrm{F}_2^{\mathrm{BS}} \leftarrow !(\mathrm{F}_0^{\mathrm{BS}} \oplus \mathrm{F}_1^{\mathrm{BS}})$
    // C1: Found key ∧ one child. Traverse lone child.
    // C2: Found key ∧ both children. Find successor: go right (then find leftmost node in this subtree)
10: $\mathrm{F}_{C1}^{\mathrm{BS}} \leftarrow \mathrm{F}_{lf}^{\mathrm{BS}} \cdot \mathrm{F}_1^{\mathrm{BS}}$, $\mathrm{F}_{C2}^{\mathrm{BS}} \leftarrow \mathrm{F}_{lf}^{\mathrm{BS}} \cdot \mathrm{F}_2^{\mathrm{BS}}$
    // C3: Finding successor ∧ left child. Go left.
    // C4: Finding successor ∧ no left child. This node is the successor, go right.
11: $\mathrm{F}_{C3}^{\mathrm{BS}} \leftarrow \mathrm{st}_D.\mathrm{F}_{fs}^{\mathrm{BS}} \cdot (!\mathrm{l0}^{\mathrm{BS}})$, $\mathrm{F}_{C4}^{\mathrm{BS}} \leftarrow \mathrm{st}_D.\mathrm{F}_{fs}^{\mathrm{BS}} \cdot \mathrm{l0}^{\mathrm{BS}}$
    // Direction bit computed by accounting for C1-C4 cases altogether.
12: $\mathrm{d}^{\mathrm{BS}} \leftarrow ((\mathrm{F}_{C2}^{\mathrm{BS}} | \mathrm{F}_{C4}^{\mathrm{BS}}) \cdot (!\mathrm{F}_{C3}^{\mathrm{BS}})) | (\mathrm{F}_{C1}^{\mathrm{BS}} \cdot \mathrm{l0}^{\mathrm{BS}})$
13: $\mathrm{np}^{\mathrm{XS}} \leftarrow$ Oselect $(\mathrm{nl}_c^{\mathrm{XS}}[0], \mathrm{nl}_c^{\mathrm{XS}}[1], \mathrm{d}^{\mathrm{BS}})$
14: $\mathrm{st}_D.F_f \leftarrow$ Oselect $(\mathrm{st}_D.F_f, 1^{\mathrm{BS}}, \mathrm{F}_{lf}^{\mathrm{BS}})$
15: $\mathrm{st}_D.\mathrm{F}_{fs}^{\mathrm{BS}} \leftarrow \mathrm{st}_D.\mathrm{F}_{fs}^{\mathrm{BS}} \oplus \mathrm{F}_{C2}^{\mathrm{BS}}$
16: $\mathrm{st}_D.\mathrm{F}_{ss}^{\mathrm{BS}} \leftarrow \mathrm{st}_D.\mathrm{F}_{ss}^{\mathrm{BS}} \oplus \mathrm{F}_{C2}^{\mathrm{BS}}$
17: $\mathrm{st}_D.\mathrm{F}_{fs}^{\mathrm{BS}} \leftarrow$ Oselect $(\mathrm{st}_D.\mathrm{F}_{fs}^{\mathrm{BS}}, 0^{\mathrm{BS}}, \mathrm{F}_{C4}^{\mathrm{BS}})$
18: $(F_f, \mathrm{u}^{\mathrm{BS}}, \mathrm{st}_D) \leftarrow$ RDelete$(\mathrm{np}^{\mathrm{XS}}, \mathrm{k}^{\mathrm{AS}}, \mathrm{TTL\text{-}1}, \mathrm{st}_D)$
    // Abort if the deletion key was not found.
19: if $F_f$==0 then
20:     return (0, ⊥, ⊥)
21: end if
22: $\mathrm{nl}_c^{\mathrm{XS}} \leftarrow$ UpdCPtrs$(\mathrm{nl}_c^{\mathrm{XS}}, \mathrm{d}^{\mathrm{BS}}, \mathrm{st}_D.ri, \mathrm{st}_D.\mathrm{F}_r^{\mathrm{BS}})$
23: $(\mathrm{nl}_b^{\mathrm{XS}}, \mathrm{u}^{\mathrm{BS}}, \mathrm{F}_{imb}^{\mathrm{BS}}) \leftarrow$ UpdBal$_D(\mathrm{nl}_b^{\mathrm{XS}}, \mathrm{u}^{\mathrm{BS}}, \mathrm{d}^{\mathrm{BS}})$
24: $(\ell^{\mathrm{XS}}, \mathrm{u}^{\mathrm{BS}}) \leftarrow$ FixImb$_D(\ell^{\mathrm{XS}}, \mathrm{d}^{\mathrm{BS}}, \mathrm{u}^{\mathrm{BS}}, \mathrm{F}_{imb}^{\mathrm{BS}}, \mathrm{st}_D)$
    // $\mathrm{F}_{rs}^{\mathrm{BS}}$: return skip flag. Handles the two cases where we delete (skip) the next child on path.
25: $\mathrm{F}_{rs}^{\mathrm{BS}} \leftarrow (\mathrm{F}_{C2}^{\mathrm{BS}} | \mathrm{F}_{C4}^{\mathrm{BS}})$
    // If $\mathrm{F}_{rs}^{\mathrm{BS}}$: we skipped a node, so balance update u = 1
26: $\mathrm{u}^{\mathrm{BS}} \leftarrow$ Oselect $(\mathrm{u}^{\mathrm{BS}}, 1^{\mathrm{BS}}, \mathrm{F}_{rs}^{\mathrm{BS}})$
27: $\mathrm{st}_D \leftarrow$ UpdState$(\mathrm{F}_{rs}^{\mathrm{BS}}, \mathrm{F}_{imb}^{\mathrm{BS}}, \ell^{\mathrm{XS}}, \mathrm{F}_{C2}^{\mathrm{BS}}, \mathrm{F}_{C4}^{\mathrm{BS}}, \mathrm{F}_{dh}^{\mathrm{BS}}, \mathrm{st}_D)$
28: return $(1, \mathrm{u}^{\mathrm{BS}}, \mathrm{st}_D)$
```

Protocol 18 `UpdBal`$_D$: Update Balance for Delete.
`UpdBal`$_D(\mathrm{p}_b^{\mathrm{BS}}, \mathrm{u}^{\mathrm{BS}}, \mathrm{d}^{\mathrm{BS}}) \rightarrow (\mathrm{p}_b^{\mathrm{BS}}, \mathrm{u}^{\mathrm{BS}}, \mathrm{F}_{imb}^{\mathrm{BS}})$
Inputs: $\mathrm{p}_b^{\mathrm{BS}}$: current balances of the node, u^{BS}: update bit, d^{BS}: direction of child responsible for update
Outputs: $\mathrm{p}_b^{\mathrm{BS}}$: updated balances, u^{BS}: new update bit to propogate upward, $\mathrm{F}_{imb}^{\mathrm{BS}}$: imbalance flag
Note in case of an imbalance, this function sets $\mathrm{p}_b^{\mathrm{BS}}$to zeroes.

```
1:  $\mathrm{F}_{ls}^{\mathrm{BS}} \leftarrow u^{\mathrm{BS}} \cdot gt^{\mathrm{BS}}$
2:  $\mathrm{F}_{rs}^{\mathrm{BS}} \leftarrow u^{\mathrm{BS}} \cdot !gt^{\mathrm{BS}}$
3:  $\mathrm{F}_b^{\mathrm{BS}} \leftarrow \mathrm{p}_b^{\mathrm{BS}}[0] \oplus \mathrm{p}_b^{\mathrm{BS}}[1]$
4:  $(\mathrm{p}_b^{\mathrm{BS}}, \mathrm{F}_{imb}^{\mathrm{BS}}) \leftarrow$ RSBal$(\mathrm{p}_b^{\mathrm{BS}}, \mathrm{F}_b^{\mathrm{BS}}, 0^{\mathrm{BS}}, \mathrm{F}_{rs}^{\mathrm{BS}})$
5:  $(\mathrm{p}_b^{\mathrm{BS}}, \mathrm{F}_{imb}^{\mathrm{BS}}) \leftarrow$ LSBal$(\mathrm{p}_b^{\mathrm{BS}}, \mathrm{F}_b^{\mathrm{BS}}, \mathrm{F}_{imb}^{\mathrm{BS}}, \mathrm{F}_{ls}^{\mathrm{BS}})$
    // If update results in this node becoming left or right heavy: the height of the subtree has not changed. Set $u^{\mathrm{BS}}$ to 0.
6:  $\mathrm{F}_b^{\mathrm{BS}} \leftarrow \mathrm{p}_b^{\mathrm{BS}}[0] \oplus \mathrm{p}_b^{\mathrm{BS}}[1]$
7:  $\mathrm{F}_{bu}^{\mathrm{BS}} \leftarrow \mathrm{F}_b^{\mathrm{BS}} \cdot u^{\mathrm{BS}}$
8:  $\mathrm{u}^{\mathrm{BS}} \leftarrow$ Oselect $(\mathrm{u}^{\mathrm{BS}}, 0^{\mathrm{BS}}, \mathrm{F}_{bu}^{\mathrm{BS}})$
    // If update results in this node becoming balanced, height of this subtree has decreased, so continue propogating $u^{\mathrm{BS}}$.
    // FixBal$_D$ handles the balance updates in case of an imbalance.
9:  return $(\mathrm{p}_b^{\mathrm{BS}}, \mathrm{u}^{\mathrm{BS}}, \mathrm{F}_{imb}^{\mathrm{BS}})$
```

Protocol 19 `FixImb`$_D$. Fix the imbalance (if there is one) at a level during deletion.
`FixImb`$_D(\ell^{\mathrm{XS}}, \mathrm{p}_c^{\mathrm{XS}}, \mathrm{p}_b^{\mathrm{BS}}, \mathrm{d}^{\mathrm{BS}}, \mathrm{u}^{\mathrm{BS}}, \mathrm{F}_{imb}^{\mathrm{BS}}, \mathrm{st}_D) \rightarrow (\ell^{\mathrm{XS}}, \mathrm{u}^{\mathrm{BS}})$
Inputs: ℓ^{XS}: current subtree root, $\mathrm{p}_c^{\mathrm{XS}}$: children pointers of subtree root, $\mathrm{p}_b^{\mathrm{BS}}$: balance bits of subtree root, d^{BS}: traversal direction, u^{BS}: update bit shares, $\mathrm{F}_{imb}^{\mathrm{BS}}$: imbalance flag, st_D: state
Outputs: u^{BS}: new update bit, ℓ^{XS}: new root for current subtree.

```
    // $\vec{cs}$: pointer to the child's sibling, i.e. $p_c[!d]$
1:  $\vec{cs}^{\mathrm{XS}} \leftarrow$ Oselect $(\mathrm{p}_c^{\mathrm{XS}}[1], \mathrm{p}_c^{\mathrm{XS}}[0], \mathrm{d}^{\mathrm{BS}})$
2:  $(\mathrm{cs}_c^{\mathrm{XS}}, \mathrm{cs}_b^{\mathrm{BS}}) \leftarrow \mathbf{D}[\vec{cs}^{\mathrm{XS}}]$
    // $n$ is the node $cs[d]$, $n_b$ ($n$'s balance) dictates if the imbalance requires a LRRL rotation.
3:  $\vec{n}^{\mathrm{XS}} \leftarrow \mathrm{cs}_c^{\mathrm{XS}}[d^{\mathrm{BS}}]$
4:  $(\mathrm{n}_c^{\mathrm{XS}}, \mathrm{n}_b^{\mathrm{BS}}) \leftarrow \mathbf{D}[\vec{n}^{\mathrm{XS}}]$
5:  $\mathrm{F}_{dr}^{\mathrm{BS}} \leftarrow \mathrm{F}_{imb}^{\mathrm{BS}} \cdot \mathrm{cs}_b^{\mathrm{BS}}[d^{\mathrm{BS}}]$
6:  $(\vec{q}^{\mathrm{XS}}, \mathrm{cs}_c^{\mathrm{XS}}, \mathrm{n}_c^{\mathrm{XS}}) \leftarrow$ ORotate$(\vec{cs}^{\mathrm{XS}}, \mathrm{cs}_c^{\mathrm{XS}}, \vec{n}^{\mathrm{XS}}, \mathrm{n}_c^{\mathrm{XS}}, \mathrm{st.d}_{cn}^{\mathrm{BS}}, \mathrm{F}_{dr}^{\mathrm{BS}})$
7:  $\mathrm{p}_c^{\mathrm{XS}} \leftarrow$ UpdCPtrs$(\mathrm{p}_c^{\mathrm{XS}}, !\mathrm{d}^{\mathrm{BS}}, \vec{q}^{\mathrm{XS}}, \mathrm{F}_{dr}^{\mathrm{BS}})$
    // If $\mathrm{F}_{dr}^{\mathrm{BS}}$: Switch $cs$ and $n$ before the next ORotate().
8:  $tmp\_cs_c{}^{\mathrm{XS}} \leftarrow$ Oselect $(\mathrm{cs}_c^{\mathrm{XS}}, \mathrm{n}_c^{\mathrm{XS}}, \mathrm{F}_{dr}^{\mathrm{BS}})$
9:  $tmp\_cs'^{\mathrm{XS}} \leftarrow$ Oselect $(\vec{cs}^{\mathrm{XS}}, \vec{n}^{\mathrm{XS}}, \mathrm{F}_{dr}^{\mathrm{BS}})$
10: $(\ell^{\mathrm{XS}}, \mathrm{p}_c^{\mathrm{XS}}, tmp\_cs_c{}^{\mathrm{XS}}) \leftarrow$ ORotate$(\vec{p}^{\mathrm{XS}}, \mathrm{p}_c^{\mathrm{XS}}, tmp\_cs'^{\mathrm{XS}}, tmp\_cs_c{}^{\mathrm{XS}}, \mathrm{st.d}_{pc}^{\mathrm{BS}}, \mathrm{F}_{imb}^{\mathrm{BS}})$
    // Return cs and n's children pointers back correctly.
11: $\mathrm{n}_c^{\mathrm{XS}} \leftarrow$ Oselect $(\mathrm{n}_c^{\mathrm{XS}}, tmp\_cs_c{}^{\mathrm{XS}}, \mathrm{F}_{dr}^{\mathrm{BS}})$
12: $\mathrm{cs}_c^{\mathrm{XS}} \leftarrow$ Oselect $(tmp\_cs_c{}^{\mathrm{XS}}, c_c'{}^{\mathrm{XS}}, \mathrm{F}_{dr}^{\mathrm{BS}})$
    // FixBal$_D$ updates balances of $\mathrm{p}_b^{\mathrm{BS}}$, $\mathrm{cs}_b^{\mathrm{BS}}$, $\mathrm{n}_b^{\mathrm{BS}}$ to account for rotations performed.
13: $(\mathrm{u}^{\mathrm{BS}}, \mathrm{p}_b^{\mathrm{BS}}, \mathrm{cs}_b^{\mathrm{BS}}, \mathrm{n}_b^{\mathrm{BS}}) \leftarrow$ FixBal$_D(\mathrm{p}_b^{\mathrm{BS}}, \mathrm{cs}_b^{\mathrm{BS}}, \mathrm{n}_b^{\mathrm{BS}}, \mathrm{d}^{\mathrm{BS}}, \mathrm{u}^{\mathrm{BS}}, \mathrm{F}_{imb}^{\mathrm{BS}})$
14: $\mathbf{D}[\ell^{\mathrm{XS}}] \leftarrow (\mathrm{p}_b^{\mathrm{BS}}, \mathrm{p}_c^{\mathrm{XS}})$, $\mathbf{D}[\vec{cs}^{\mathrm{XS}}] \leftarrow (\mathrm{cs}_b^{\mathrm{BS}}, \mathrm{cs}_c^{\mathrm{XS}})$,
15: $\mathbf{D}[\vec{n}^{\mathrm{XS}}] \leftarrow (\mathrm{n}_b^{\mathrm{BS}}, \mathrm{n}_c^{\mathrm{XS}})$
16: return $(\ell^{\mathrm{XS}}, \mathrm{u}^{\mathrm{BS}})$
```

Protocol 20 `FixBal`$_D$. Fixes the balances of nodes affected by an imbalance (if there is one) at the current level.
`FixBal`$_D(\mathrm{p}_b^{\mathrm{BS}}, \mathrm{cs}_b^{\mathrm{BS}}, \mathrm{n}_b^{\mathrm{BS}}, \mathrm{d}^{\mathrm{BS}}, \mathrm{u}^{\mathrm{BS}}, \mathrm{F}_{imb}^{\mathrm{BS}}) \rightarrow (\mathrm{u}^{\mathrm{BS}}, \mathrm{p}_b^{\mathrm{BS}}, \mathrm{cs}_b^{\mathrm{BS}}, \mathrm{n}_b^{\mathrm{BS}})$
Inputs: $\mathrm{p}_b^{\mathrm{BS}}$: parent($p$)'s balance bits, $\mathrm{cs}_b^{\mathrm{BS}}$: child's sibling($cs$)'s balance bits, $\mathrm{n}_b^{\mathrm{BS}}$: balance bits of next node in the direction $\mathrm{d}^{\mathrm{BS}}$from cs, d^{BS}: direction bit from p to c (and also from cs to n), u^{BS}: update bit, $\mathrm{F}_{imb}^{\mathrm{BS}}$: imbalance flag
Outputs: u^{BS}: new update bit, $\mathrm{p}_b^{\mathrm{BS}}$: updated balance bits for parent (p), $\mathrm{cs}_b^{\mathrm{BS}}$: updated balance bits for child's sibling (cs), $\mathrm{n}_b^{\mathrm{BS}}$: updated balance bits for n.

```
    // IC(1-3) correspond to three possible imbalance cases.
 1: $\mathrm{F}_{IC1}^{\mathrm{BS}} \leftarrow \mathrm{F}_{imb}^{\mathrm{BS}} \cdot \mathrm{cs}_b^{\mathrm{BS}}[!\mathrm{d}^{\mathrm{BS}}]$
 2: $\mathrm{F}_{IC3}^{\mathrm{BS}} \leftarrow \mathrm{F}_{imb}^{\mathrm{BS}} \cdot \mathrm{cs}_b^{\mathrm{BS}}[\mathrm{d}^{\mathrm{BS}}]$
 3: $\mathrm{F}_{IC2}^{\mathrm{BS}} \leftarrow \mathrm{F}_{imb}^{\mathrm{BS}} \cdot !(\mathrm{F}_{IC1}^{\mathrm{BS}} \oplus \mathrm{F}_{IC3}^{\mathrm{BS}})$
    // IC1 case: set $cs_b[!d]$ to 0.
 4: $\mathrm{cs}_b^{\mathrm{BS}}[!\mathrm{d}^{\mathrm{BS}}] \leftarrow$ Oselect $(0^{\mathrm{BS}}, \mathrm{cs}_b^{\mathrm{BS}}[!\mathrm{d}^{\mathrm{BS}}], \mathrm{F}_{IC1}^{\mathrm{BS}})$
    // IC2 case: set $cs_b[d]$ to 1, and $p_b[!d]$ to 1.
 5: $\mathrm{cs}_b^{\mathrm{BS}}[\mathrm{d}^{\mathrm{BS}}] \leftarrow$ Oselect $(\mathrm{cs}_b^{\mathrm{BS}}[\mathrm{d}^{\mathrm{BS}}], 1^{\mathrm{BS}}, \mathrm{F}_{IC2}^{\mathrm{BS}})$
 6: $\mathrm{p}_b^{\mathrm{BS}}[!\mathrm{d}^{\mathrm{BS}}] \leftarrow$ Oselect $(\mathrm{p}_b^{\mathrm{BS}}[!\mathrm{d}^{\mathrm{BS}}], 1^{\mathrm{BS}}, \mathrm{F}_{IC2}^{\mathrm{BS}})$
    // IC2 rotation does not decrease subtree height, so $u \leftarrow 0$.
 7: $\mathrm{u}^{\mathrm{BS}} \leftarrow$ Oselect $(\mathrm{u}^{\mathrm{BS}}, 0^{\mathrm{BS}}, \mathrm{F}_{IC2}^{\mathrm{BS}})$
    // IC3 has 3 subcases. IC3-S1 : $n_b[!d] = 1$, IC3-S2 : $n_b[d] = 1$,
    IC3-S3: $n_b[d] = n_b[!d] = 0$
 8: $\mathrm{F}_{IC3-S1}^{\mathrm{BS}} \leftarrow \mathrm{n}_b^{\mathrm{BS}}[!\mathrm{d}^{\mathrm{BS}}]$, $\mathrm{F}_{IC3-S2}^{\mathrm{BS}} \leftarrow \mathrm{n}_b^{\mathrm{BS}}[\mathrm{d}^{\mathrm{BS}}]$
 9: $\mathrm{F}_{IC3-S3}^{\mathrm{BS}} \leftarrow !(\mathrm{F}_{IC3-S1}^{\mathrm{BS}} \oplus \mathrm{F}_{IC3-S2}^{\mathrm{BS}})$
    // IC3-S1: set $p_b[d]$ to 1, and $cs_b[d]$ to 0.
10: $\mathrm{p}_b^{\mathrm{BS}}[\mathrm{d}^{\mathrm{BS}}] \leftarrow$ Oselect $(1^{\mathrm{BS}}, \mathrm{p}_b^{\mathrm{BS}}[\mathrm{d}^{\mathrm{BS}}], \mathrm{F}_{IC3-S1}^{\mathrm{BS}})$
11: $\mathrm{cs}_b^{\mathrm{BS}}[\mathrm{d}^{\mathrm{BS}}] \leftarrow$ Oselect $(0^{\mathrm{BS}}, \mathrm{cs}_b^{\mathrm{BS}}[\mathrm{d}^{\mathrm{BS}}], \mathrm{F}_{IC3-S1}^{\mathrm{BS}})$
    // IC3-S2: swap $cs_b[d]$ and $cs_b[!d]$. Set $n_b[d]$ to 0.
12: Oswap $(\mathrm{cs}_b^{\mathrm{BS}}[\mathrm{d}^{\mathrm{BS}}], \mathrm{cs}_b^{\mathrm{BS}}[!\mathrm{d}^{\mathrm{BS}}], \mathrm{F}_{IC3-S2}^{\mathrm{BS}})$
13: $\mathrm{n}_b^{\mathrm{BS}}[\mathrm{d}^{\mathrm{BS}}] \leftarrow$ Oselect $(0^{\mathrm{BS}}, \mathrm{n}_b^{\mathrm{BS}}[\mathrm{d}^{\mathrm{BS}}], \mathrm{F}_{IC3-S2}^{\mathrm{BS}})$
    // IC3-S3: set $cs_b[d]$ to 0.
14: $\mathrm{cs}_b^{\mathrm{BS}}[\mathrm{d}^{\mathrm{BS}}] \leftarrow$ Oselect $(0^{\mathrm{BS}}, \mathrm{cs}_b^{\mathrm{BS}}[\mathrm{d}^{\mathrm{BS}}], \mathrm{F}_{IC3-S3}^{\mathrm{BS}})$
15: return $(\mathrm{u}^{\mathrm{BS}}, \mathrm{p}_b^{\mathrm{BS}}, \mathrm{cs}_b^{\mathrm{BS}}, \mathrm{n}_b^{\mathrm{BS}})$
```

When a leaf node is reached, the protocol does dummy recursions until the maximum height of the AVL tree.

Once the next node pointer np from this level is selected (line 12), lines 13 to 15 update the flags keeping the state of if the node was found (st.F_f), and if we are finding successor (st.F_{fs}), and then we recurse to the next level. In deletion, imbalances arise on the sibling path (rather than the traversal path). Since we remove a node along the traversal path, the relative height of the sibling subtree may increase, leading to an imbalance. Hence cs corresponds to the child's sibling (with the child being the node at the next pointer np we arrived at after the four cases). The relevant node for handling imbalances during deletion is cs (and not c which was the case during insert), and lines 20–21 extract this child's sibling node. Line 22 handles updating the child pointers at the current level based on the state st′ returned by the recursive `RDelete` invocation. There are three cases that result in updating child pointers:

- Skip deletion node: At the deletion node level, we return the return index ri returned from the recursion.
- Skip successor node: Same as above. Both cases are handled by the `UpdState` protocol detailed in Protocol 21.

Protocol 21 `UpdState`$(\mathrm{F}_{rs}^{\mathrm{BS}}, \mathrm{F}_{imb}^{\mathrm{BS}}, \ell^{\mathrm{XS}}, \mathrm{F}_{C2}^{\mathrm{BS}}, \mathrm{F}_{C4}^{\mathrm{BS}}, \mathrm{F}_{dh}^{\mathrm{BS}}, \mathrm{st}_D) \rightarrow (\mathrm{st}_D)$
Inputs: $\mathrm{F}_{rs}^{\mathrm{BS}}$: return skip flag, $\mathrm{F}_{imb}^{\mathrm{BS}}$: imbalance flag, ℓ^{XS}: pointer to current subtree root after imbalance fix, $\mathrm{F}_{C2}^{\mathrm{BS}}$: flag to delete node (with both childen) at this level, $\mathrm{F}_{C4}^{\mathrm{BS}}$: flag to indicate successor found, $\mathrm{F}_{dh}^{\mathrm{BS}}$: flag to delete leaf node, st_D: current state
Outputs: st_D: updated state

```
   // If we skip a level or had an imbalance, child pointers of the
   node at the higher level needs to be updated.
1: $\mathrm{st}_D.\mathrm{F}_r^{\mathrm{BS}} \leftarrow \mathrm{F}_{rs}^{\mathrm{BS}} | \mathrm{F}_{imb}^{\mathrm{BS}} | \mathrm{F}_{dh}^{\mathrm{BS}}$
2: $\mathrm{st}_D.\mathrm{ri}^{\mathrm{XS}} \leftarrow$ Oselect $(\ell^{\mathrm{XS}}, \mathrm{st}_D.\mathrm{ri}^{\mathrm{XS}}, \mathrm{st}_D.\mathrm{F}_r^{\mathrm{BS}})$
   Store deletion and successor node
3: $\mathrm{st}_D.\vec{nd}^{\mathrm{XS}} \leftarrow (\mathrm{st}_D.\vec{nd}^{\mathrm{XS}}, \ell^{\mathrm{XS}}, \mathrm{F}_{C2}^{\mathrm{BS}})$
4: $\mathrm{st}_D.\vec{ns}^{\mathrm{XS}} \leftarrow (\mathrm{st}_D.\vec{ns}^{\mathrm{XS}}, \ell^{\mathrm{XS}}, \mathrm{F}_{C4}^{\mathrm{BS}})$
5: return $\mathrm{st}_D$
```

- Update child subtree root with new node from imbalance-fixing rotations: `FixImb`$_D$ detailed in Protocol 19 returns the index for the new root of this subtree ℓ' (either index of the present root itself, or in the case of an imbalance either $\vec{cs}$ or $cs_c[d]$.)

Similar to insert, line 23 updates balances to propagate the potential decrease in the height of a subtree and this process is detailed in Protocol 18. As we mentioned earlier, since there could be an imbalance at possible every level on the traversal path in the case of deletion, we perform the imbalance fix procedure, similar to the one-time imbalance-fix operation we detailed at the end of `Insert`, at every level by invoking `FixImb`$_D$. The difference is that the balance bit updates have a different semantic meaning in deletion (decrease by one instead of increase by one). Finally, line 27 updates the state to ensure: (i) the correct return index ri is returned by the recursion depending on the flag bits that manipulate if a skip or imbalance has happened (F_r, $\mathrm{F}_{rs}^{\mathrm{BS}}$, $\mathrm{F}_{imb}^{\mathrm{BS}}$), and (ii) the deletion and successor node index (when applicable) are stored in st.

D MPC OPERATION BREAKDOWN OF PRAC PROTOCOLS

Table 6 gives a granular breakdown of the number of MPC operations required in data structures implemented in PRAC. We denote the DORAM operations as tuple $(n, \mathcal{S})$, where n is the number of ORAM operations and (a size n) $\mathcal{S}$ is a set of sizes on which the n ORAM operations are performed.

Table 6: Presenting MPC and ORAM operations for our optimized data structure algorithms. $N = [n, n, \ldots, n]$, $A = [2, 4, \ldots, n/2]$, $B = [2, 4, \ldots, \lg n/2]$. **We denote the change in the number of operations from basic to optimized counts as** (**basic counts** $\longrightarrow$ **optimized counts**). **Green indicates a decrease in the count and Red indicates an increase in the count.**

	BinarySearch	HeapInsert	ExtractMin	AVLInsert	AVLDelete
ORAM Read	$(\lg n, N) \longrightarrow (\lg n, A)$	$0 \longrightarrow (\lg \lg n, B)$	$3 \cdot \lg n$ (r,p)	$(4 + 1.44 \cdot \lg n, N)$	$2 + 3 \cdot 1.44 \cdot \lg n$
ORAM Update	0	0	$3 \cdot \lg n$ (r,p)	$(4 + 1.44 \cdot \lg n, N)$	$2{+}3 \cdot 1.44 \cdot \lg n$
# ORAM DPFs	$\lg n \longrightarrow 1$	$0 \longrightarrow 1$	$3 \cdot \lg n$ (r)	$4 + 1.44 \cdot \lg n$	$2 + 3 \cdot 1.44 \cdot \lg n$
Ocompare	$\lg n$	$\lg n \longrightarrow \lg(\lg n)$	$2 \cdot \lg n$	$1.44 \cdot \lg n$	$2 + 3 \cdot 1.44 \lg n$
Oselect	0	0	$2 \lg n$	$40 + \lg n$	$4 + 69 \cdot 1.44 \cdot \lg n$
Oswap	0	$\lg n \longrightarrow 0$	0	0	0
MPC-ANDs	0	0	$\lg n$	$5 + 8 \cdot 1.44 \cdot \lg n$	$40 \cdot 1.44 \cdot \lg n$
MPC-FW-Mults	$\lg n \longrightarrow 0$	0	$2 \cdot \lg n(2p)$	0	0