Probabilistic Skipping-Based Data Structures with Robust Efficiency Guarantees*

Marc Fischlin
Technische Universität Darmstadt
Darmstadt, Germany
marc.fischlin@tu-darmstadt.de

Moritz Huppert
Technische Universität Darmstadt
Darmstadt, Germany
moritz.huppert@tu-darmstadt.de

Sam A. Markelon
University of Florida
Gainesville, USA
smarkelon@ufl.edu

ABSTRACT

Probabilistic data structures like hash tables, skip lists, and treaps support efficient operations through randomized hierarchies that enable "skipping" elements, achieving sub-linear query complexity on average for perfectly correct responses. They serve as critical components in performance-sensitive systems where correctness is essential and efficiency is highly desirable. While simpler than deterministic alternatives like balanced search trees, these structures traditionally assume that input data are independent of the structure's internal randomness and state – an assumption questionable in malicious environments – potentially leading to a significantly increased query complexity. We present adaptive attacks on all three aforementioned structures that, in the case of hash tables and skip lists, cause exponential degradation compared to the input-independent setting. While efficiency-targeting attacks on hash tables are well-studied, our attacks on skip lists and treaps provide new insights into vulnerabilities of skipping-based probabilistic data structures. Next, we propose simple and efficient modifications to the original designs of these data structures to provide provable security against adaptive adversaries. Our approach is formalized through Adaptive Adversary Property Conservation (AAPC), a general security notion that captures deviation from the expected efficiency guarantees in adversarial scenarios. We use this notion to present rigorous robustness proofs for our versions of the data structures. Lastly, we perform experiments whose empirical results closely agree with our analytical results.

KEYWORDS

Adversial Environment, Probabilistic Data Structures, Denial of Service, Complexity Attacks, Hash Table, Skip List, Treap

1 INTRODUCTION

Probabilistic data structures (PDS), such as hash tables, Bloom filters, and skip lists, are widely implemented due to their memory efficiency and favorable operational time complexity, making them essential tools for cost-effective, scalable data processing in resource-constrained environments. Industry adoption of these structures is extensive and growing: Redis, an open-source in-memory data structure store, leverages HyperLogLog for cardinality estimation and Bloom filters for membership testing, achieving superior performance in high-volume deployments at Twitter, Pinterest, and other large internet companies [40]. Similarly, Nowack et al. demonstrated enhanced scalability in Discord's server member management through skip list-based implementations [41], Prout et al. utilized skip lists for relational database indexing [45], while Schanck et al. reported significant cost reductions in Mozilla Firefox's certificate revocation checking using the novel Clubcard PDS [48]. Despite their widespread application, analyses of these structures' behavior under adversarial conditions remain notably sparse in the literature. This gap is significant because these data structures frequently operate in contexts where malicious actors might deliberately manipulate inputs to induce erroneous outputs or degrade the performance of these structures.

Compressing probabilistic data structures (CPDS), such as Bloom filters, HyperLogLog, count-min sketch, etc., provide compact (sub-linear) representations of potentially large collections of data and support a small set of queries that can be answered efficiently. These space and (by extension) performance gains come at the expense of correctness. Specifically, CPDS query responses are computed over the compact representation of the data, as opposed to the complete data. As a result, CPDS query responses are only guaranteed to be *close* to the true answer with *large* probability, where *close* and *large* are typically functions of structure parameters (e.g., the representation size) and properties of the data. These guarantees are stated under the assumption that the data and the internal randomness of the PDS are independent. Informally, this is tantamount to assuming that the entire collection of data is (or can be) determined *before* the PDS makes any random choices. For many PDS, this amounts to the sampling of hash functions, as the PDS operates deterministically afterward.

Recent research [17, 23–25, 35, 39, 43] shows that, under adversarial conditions, these errors for a variety of CPDS can be exacerbated significantly, potentially undermining the reliability of systems that rely on these structures in critical or adversarial contexts. These attacks exploit fundamental structural vulnerabilities where adversaries can strategically craft input sets that cause approximations to deviate far from the non-adaptive guarantee. The threat model extends beyond intentional adversarial exploitation to encompass unintentional system misconfigurations and client malfunctions that violate the critical independence assumption between input data and internal randomness, such as predictable hash function seeds or correlated input distributions. Some of these works also explore applying provable security techniques to these structures, in turn providing robust versions of these structures with respect to correctness in adversarial environments.

A distinct subset of probabilistic data structures, which ensures correctness (and hence are not compressing) while offering fast probabilistic runtime guarantees, have received considerably less attention in the literature. Existing security analyses, such as those addressing the robustness of hash tables [9, 10, 13, 19, 22, 33] and skip lists [42], provide valuable insights but lack formal adversarial

*Please cite the conference version of this paper published at ACM CCS 2025 [26].

models and rigorous security analyses. Due to their runtime properties, we refer to these as *probabilistic skipping-based data structures* (PSDS), as they inherently "skip" over parts of their internal structure to accelerate lookup operations. The lack of research in this area is particularly concerning given that the studies on hash tables have already uncovered practical attacks, including methods to mount denial-of-service attacks against intrusion detection systems [10], web application servers [33], and the QUIC protocol [13].

To address these shortcomings:

- We develop a novel security model for PSDS that captures how their probabilistic runtime guarantees can be preserved under completely adversarial conditions. Our model addresses the unavoidable side-channel vulnerabilities created by the intrinsic correlation between observed runtime and the internal representation of PSDS.
- We analyze three popular probabilistic data structures (hash tables, skip lists, and treaps) using a novel framework that models their data representation as sequences of identically distributed random variables. This perspective demonstrates a powerful new attack against skip lists (the gap attack) and refines our understanding of deletion handling in robustness analysis. Our attacks exploit structural vulnerabilities where adversaries craft inputs to force worst-case performance degradation from expected logarithmic or constant time to linear complexity. Critically, both the well-known collision attack on hash tables and the gap attack require only standard insertion capabilities, making them accessible to external attackers without privileged operations like deletions. Surprisingly, treaps exhibit inherent robustness against this adversary model.
- We present provably secure and efficient constructions for hash tables, skip lists, and treaps. These constructions maintain security even when adaptive adversaries can observe the internal state of the particular data structure during their entire execution.
- We validate our theoretical contributions through experimental evaluations and release our implementation code as a public resource accompanying this paper*.

2 RELATED WORK

2.1 Self-Balancing and Self-Organizing Data Structures

Although PSDS share conceptual similarities with *self-balancing* and *self-organizing* data structures, they differ fundamentally in their guarantees and methodological approaches.

Self-organizing data structures [2], whether randomized or deterministic, dynamically adjust their internal ordering of elements to optimize performance based on a given (potentially adversarial) sequence of input requests. For instance, self-organizing lists may employ the move-to-front heuristic, where accessed elements are relocated to the front of the list, or the transpose method, where elements swap positions with their predecessors when accessed. Similarly, splay trees [50] rotate frequently accessed nodes closer to the root to reduce future access times. Self-organizing data structures have been extensively analyzed under adversarial models, with (randomized) self-organizing lists incurring a cost at least three times that of the optimal reordering strategy [47].

Self-balancing data structures, such as Red-Black trees [11] and AVL trees [1], *deterministically* ensure an upper-bound on node depth, thereby providing worst-case performance guarantees for search operations. This deterministic approach is also exemplified by the deterministic skip list [38], which enforces an optimal structure by carefully promoting inserted nodes and their neighborhoods to appropriate levels. While these structures guarantee bounded search path lengths (even in adversarial settings), they require complex re-balancing mechanisms. In stark contrast, PSDS, such as the treap [49] and the original skip list [46], offer comparable expected performance, achieved through simple, probabilistic updating mechanisms. This presents a clear trade-off: deterministic structures provide absolute performance guarantees at the cost of implementation complexity, while probabilistic alternatives offer simplicity, albeit with only probabilistic guarantees. In this work, we investigate whether we can maintain the implementation simplicity of probabilistic data structures while preserving their performance guarantees even in adversarial settings.

While both self-organizing and self-balancing data structures have been analyzed under adversarial and worst-case models, respectively, the corresponding analysis for PSDS against adaptive adversaries remains an open problem.

2.2 Performance Attacks Against Probabilistic Skipping-Based Data Structures

This section examines performance attacks, also termed *complexity attacks* in the literature, which exploit the probabilistic nature of PSDS to systematically degrade their expected performance guarantees. Previous research has identified vulnerabilities in hash tables and skip lists, but these works lack formal security analysis and rigorous proofs of security when potential mitigations are put forth. Hash tables have received the most attention, while skip lists have been addressed (to our knowledge) in only a single paper in this context. Further, to our knowledge, no prior work has examined complexity attacks against treaps. This absence is consistent with our finding that treaps possess inherent resistance to such attacks.

2.2.1 Hash Tables. Assuming a hash table's internal hash function has "good" collision-resistance properties, the amortized average-case complexity of insertions, deletions, and look-ups is $O(1)$. For these efficiency reasons, hash tables are widely used in many applications such as implementing associative arrays [37] and sets [12] in many programming languages, in cache systems [30], as well as for database indexing [54].

However, the average-case performance relies on a critical assumption: that the data inserted into a hash table is independent of the (potentially randomly selected) hash function used to map key-value pairs to buckets. This assumption fundamentally breaks down in adversarial scenarios where an attacker can deliberately craft insertions that exploit knowledge of the hash function or its outputs. Given the ubiquity of hash tables in modern computing systems, numerous researchers [9, 10, 13, 19, 22, 33, 44] have investigated techniques to compromise the data structure, forcing operations to degrade from expected $O(1)$ to worst-case $O(s)$ time complexity, where s represents the total number of elements in the structure.

*https://github.com/MoritzHuppert/robust-PSDS

These adversarial approaches typically constitute complexity attacks that strategically engineer inputs, causing multi-collisions, deliberately exploiting hash function properties to force numerous distinct keys into identical buckets.

Crosby and Wallach [19] demonstrated denial-of-service attacks via complexity attacks in applications using hash tables, such as the Bro intrusion detection system [44], by forcing collisions with weak, fixed hash functions. They suggested universal hashing [16] as a mitigation, though without any formal guarantees. Klink and Walde [33] showed similar CPU exhaustion attacks on web servers (e.g., PHP, ASP.NET, Java), only using a single carefully crafted HTTP request. Aumasson et al.[9] further revealed vulnerabilities in hash tables using non-cryptographic hash functions (like MurmurHash and CityHash[5]), proposing SipHash [8] as a secure alternative. While SipHash was demonstrated to be an effective pseudorandom function (PRF), the authors lacked a rigorous security framework for analyzing hash tables employing keyed PRFs in adversarial settings. Our work addresses this gap by providing the first formal security analysis of keyed hash tables under adaptive adversaries, establishing provable bounds on adversarial runtime degradation. Complexity attacks have also been shown to be effective in causing denial-of-service against flow-monitoring systems [22]. Further, the use of salting was undermined by remote timing attacks [10]. Recently, Bottinelli et al. [13] found nearly a third of QUIC implementations vulnerable to similar attacks. Despite these works and many proposed defenses, no formal framework exists for the provable security of (keyed) hash tables against adaptive adversaries.

2.2.2 Skip Lists. In the original skip list paper [46], it is noted that it is imperative to keep the internal structure of the skip list hidden. Otherwise, adversarial users could observe the levels of individual elements and delete any element at a level greater than zero (the bottom layer). This would degenerate the structure to a simple linked list and force worst-case runtime $\Omega(s)$ on subsequent operations after these deletions occur.

Nussbaum and Segal [42] demonstrate that private internal structure alone fails to protect skip lists against timing attacks. They present a remote timing attack that correlates query response times with element heights, ultimately forcing all elements to the lowest level and degrading performance to linear time. However, their work has significant limitations: their adversarial model assumes non-adversarial initial data collection, requires preserving the original data during attacks, and restricts adversaries from accessing the internal skip list structure. While they propose a *splay skip list* countermeasure that swaps node heights with random elements during search operations, their solution lacks formal security analysis and, as our analysis demonstrates, fails to prevent structural information leakage despite introducing logarithmic overhead.

Our work addresses these limitations by presenting a significantly stronger adversarial model with formal security guarantees. We introduce a novel *gap attack* on skip lists that demonstrates how adversaries can force worst-case performance degradation from expected logarithmic time to linear complexity using only standard insertion operations — a more practical threat than the deletion-based attacks in [42]. Our proposed defense mechanism uses deliberate height swapping within small intervals to probabilistically enforce expected structural properties under adaptive adversaries, operating in constant time with provable security guarantees. For an extensive commentary on [42] and vulnerabilities in their construction, see the full version.

2.2.3 Treaps. To our knowledge, no prior work has systematically analyzed complexity attacks against treaps in adversarial settings. We provide formal proofs demonstrating that treaps with a modified deletion mechanism maintain their expected logarithmic performance guarantees even under adaptive adversaries capable of observing structural information and crafting malicious inputs.

3 PRELIMINARIES

3.1 Notations

For integers $a \le b$, we denote $[a..b] = \{a, a+1, \ldots, b\}$ as the set of integers from a to b, and $[b] = \{1, 2, \ldots, b\}$ when $a = 1$. For arrays, we write $A \leftarrow \mathsf{new}\ S$ to initialize an empty array indexed by set S (typically $S = [0..b]$ or $S = [b]$), access entries via $A[i]$ for $i \in S$, and construct sub-arrays $A[i\colon j]$ containing entries from indices i through j where $i, j \in S$ and $i \le j$. Multi-dimensional arrays can be accessed recursively as $A[i][j]$. For other data structures, such as linked lists L, we write $A \leftarrow \mathsf{new}\ \mathsf{L}$ to denote that A is assigned a new linked list, initialized to empty.

For two sets $\mathcal{A}$ and $\mathcal{B}$ we let $\mathrm{Func}(\mathcal{A}, \mathcal{B})$ be the set of all functions $f : \mathcal{A} \to \mathcal{B}$. We denote by $\mathsf{U}(S)$ the uniform distribution on the (finite or uncountable) set $S \neq \emptyset$, and by $\mathsf{G}(p)$ the geometric distribution for success probability p. In general, we write $X \sim \mathsf{D}(\cdot)$ if the random variable X is distributed according to distribution D, where D may take some parameter. We usally denote the uniform distribution on some finite set S as $\mathsf{U}(S)$. We denote by $r \twoheadleftarrow \mathsf{D}(\cdot)$ the process of randomly sampling r from the distribution $\mathsf{D}(\cdot)$.

3.2 A Syntax for Data Structures

We use the syntax for data structures first provided by [17]. Below, we give a brief overview and introduce the formal definition in Definition 3.1. We start by fixing three non-empty sets $\mathcal{D}, \mathcal{R}, \mathcal{K}$ of *data objects*, *responses*, and *keys*, respectively. Let $\mathcal{Q} \subseteq \mathrm{Func}(\mathcal{D}, \mathcal{R})$ be a set of allowed *queries*, and let $\mathcal{U} \subseteq \mathrm{Func}(\mathcal{D}, \mathcal{D})$ be a set of allowed data-object *updates*. For example, in a skip list, the data objects are subsets of the set of all admissible elements, representing the elements currently in the list. The search query qry_x maps the data objects to possible responses 0 or 1, indicating if the element x is in the list (or more generically, some space of values if the skip list is being used as an index-value store). The update operation up_x corresponds to the insertion (ins_x) or deletion (del_x) of element x in the skip list.

For data structures, we operate on the above sets. However, such structures have a specific representation, e.g., the height of elements in skip lists. This representation influences the algorithm's runtime and, in our adversarial setting, is thus made available to the adversary via the mapping Rep. The query algorithm Qry and the update algorithm Up also operate on representations. We implement secretly keyed data structures by introducing a key from the set $\mathcal{K}$ for each algorithm. Allowing each algorithm to take an individual key permits one to separate any secret randomness used across

data structure operations from per-operation randomness (e.g., salt generation or per-insertion coin flips). Note that this syntax admits the case of *unkeyed* data structures by setting $\mathcal{K} = \{\varepsilon\}$.

Definition 3.1 (Data Structure). A *Data Structure* is a tuple $\Pi = (\textsc{Rep}, \textsc{Qry}, \textsc{Up})$, where:

- $\textsc{Rep}: \mathcal{K} \times \mathcal{D} \to \{0,1\}^* \cup \{\perp\}$ is a (possibly) randomized *representation algorithm*, taking as input a key $\mathsf{k} \in \mathcal{K}$ and data object $D \in \mathcal{D}$, and outputting the representation $\mathsf{repr} \in \{0,1\}^*$ of D, or $\perp$ in the case of a failure. We write this as $\mathsf{repr} \leftarrow \textsc{Rep}_{\mathsf{k}}(D)$.
- $\textsc{Qry}: \mathcal{K} \times \{0,1\}^* \times \mathcal{Q} \to \mathcal{R} \cup \{\perp\}$ is a deterministic *query-evaluation algorithm*, taking as input $\mathsf{k} \in \mathcal{K}$, $\mathsf{repr} \in \{0,1\}^*$, and $\mathsf{qry} \in \mathcal{Q}$, and outputting an answer $\mathsf{answ} \in \mathcal{R}$, or $\mathsf{answ} = \perp$ in the case of a failure. We write this as $a \leftarrow \textsc{Qry}_{\mathsf{k}}(\mathsf{repr}, \mathsf{qry})$.
- $\textsc{Up}: \mathcal{K} \times \{0,1\}^* \times \mathcal{U} \to \{0,1\}^* \cup \{\perp\}$ is a (possibly) randomized *update algorithm*, taking as input $\mathsf{k} \in \mathcal{K}$, $\mathsf{repr} \in \{0,1\}^*$, and $\mathsf{up} \in \mathcal{U}$, and outputting an updated representation repr', or $\mathsf{repr}' = \perp$ in the case of a failure. We write this as $\mathsf{repr}' \leftarrow \textsc{Up}_{\mathsf{k}}(\mathsf{repr}, \mathsf{up})$.

Note that Up takes a function operating on data objects as an argument, even though Up itself operates on *representations* of data objects. This is intentional to match the way these data structures generally operate. In a data structure representing a set or multiset, we often think of performing operations such as 'insert x' or 'delete y'. When the set or multiset is not being stored but instead modeled via a representation, the representation must transform these operations into operations on the data structure it uses for storage. This is common for operations on compressing probabilistic data structures. We also note that the query algorithm Qry is deterministic, which reflects the overwhelming majority of data structures in practice.

3.3 Hash Tables

Hash table (HT) elements consist of pairs of entries (x, v) of a unique index value x and the value v. An instance of HT consists of b buckets, each containing an (initially empty) linked list L, and a mapping $\mathsf{H}(\mathsf{k}, \cdot)$ from index value x to bucket number in $[b]$.

An index-value pair (x, v) is inserted into the HT representation by computing $\mathsf{H}(\mathsf{k}, x)$=$i$ and traversing it to the i-th bucket. We then check if the pair is already in the linked list $\mathsf{L}[i]$ stored and delete the prior mapping if this is the case. This is necessary since we insert elements according to the index key x, and the value entry v may have changed in the new request. Finally, we insert the new pair into $\mathsf{L}[i]$. Likewise, a key is deleted by searching in the bucket to which it is assigned and removing the key and its associated value from the linked list $\mathsf{L}[i]$ in the bucket if this pair exists there. Traditionally, it is assumed that $i = \mathsf{H}(\mathsf{k}, x)$, where H is a fast-to-compute hash function with good (enough) collision resistance properties. However, we generalize here to make the exposition cleaner and allow the mapping to depend upon secret randomness (i.e., a key k). To query a key for its associated value, the algorithm $\textsc{Qry}(\mathsf{qry}_x)$ searches the bucket x maps to and returns the index-value pair if it exists there; otherwise, we return the distinguished null symbol $\star$. We give a formal pseudocode description of a hash table in the syntax of [17] in the full version.

3.4 Skip Lists

A skip list (SL) maintains an ordered collection of data that allows for average-case runtime $O(\log s)$ for search, insertions, and deletions (where the size of the represented collection is s). The structure is maintained as a hierarchy of linked lists, with the first level[†] containing all the elements of the collection, and each higher level in the structure skipping over an increasing number of elements. Searching (as well as insertions and deletions) starts at the highest level, only moving down to lower levels as necessary. The specific elements that are skipped at each level are determined either probabilistically or deterministically (using, say, a PRF) at insertion time – we focus on the probabilistic version of this structure in this paper. For a full structure description, we point to the original paper [46]. We give a formal pseudocode description of a skip list in the syntax of [17] in the full version.

Skip lists provide an elegant probabilistic alternative to balanced binary search trees. They are widely deployed in industry applications, managing millions of Discord server members [41], storing data in Apache Web Servers [4], and indexing SingleStore databases [45]. Unlike hash tables, skip lists efficiently support range queries, ordered traversals, and predecessor/successor operations, making them valuable for various applications [3, 32, 53].

3.5 Treaps

A treap [6] (TR) combines the algorithms of a binary search tree (BST) and a heap, and achieves an expected height of $O(\log s)$ for a structure containing s elements [6]. Inserting a node into a treap works analogously to a BST, but the node gets assigned an additional random priority. Subsequently, the algorithm rotates the tree to maintain a heap order among the priority values without affecting the key ordering. For instance, in a MIN heap, the parent nodes are guaranteed lower priority values than their children. Intuitively, when interpreting the priority values as timestamps, the resulting treap will correspond to a binary search tree in which all nodes have been inserted in random order (i.e., a randomized binary search tree). Deletion first rotates a node down the treap without affecting the key ordering and removes it once it reaches a leaf position. We give a formal pseudocode description of a treap in the syntax of [17] in the full version.

Treaps efficiently support the full spectrum of binary tree operations, including range queries, predecessor/successor look-ups, in-order traversals, and advanced tree operations like join, split, and union. This versatility has made treaps valuable in applications where search efficiency and ordered operations are critical requirements, such as implementing retroactive data structures [21].

4 UNIFYING PROBABILISTIC SKIPPING-BASED DATA STRUCTURES

We discuss the common property of probabilistic skipping-based data structures that the essential features of their (run time) efficiency can be described by the random choices made for each element currently in the list. This stochastic decomposition is a lossy representation of the state of the data structure and consists of a multi-set of chosen bins in the case of hash tables and a vector

[†]We consistently index skip list levels starting from 0 at the bottom level, which aligns naturally with the geometric random variables used in our later analysis.

of heights for skip lists, respectively, priorities for treaps. We then use the decomposition to argue that lazy deletion, in which one only marks deleted elements but keeps them until a reinitialization occurs or a new element retakes their place, is a valid strategy to confine the power of deletions in attacks. That is, we show that with lazy deletion, any sequence of insertion and deletion operations on our data structures can be turned into an insertion-only sequence with the same decomposition. Hence, adversaries without using deletions could cause the same (bad) run-time behavior.

4.1 Data Structures and their Stochastic Decomposition

Informally, one can think of a probabilistic skipping-based data structure as a data structure that uses some form of randomness (either fixed at initialization time or freshly sampled per operation) to distribute the underlying collection within its representation. This randomized representation is to (generally) allow for efficient search by "skipping" over some elements, such that the resulting expected runtime is sublinear with high probability.

For instance, hash tables employ a hash function to "randomly" map elements to buckets, and therefore, one only has to search in this bucket for a desired element. Likewise, skip lists randomly assign heights to elements to facilitate "skipping" over a sequence of elements while performing a search. While the treap randomly assigns priority values to maintain an (approximately) balanced tree representation. In turn, the hash table achieves non-adaptive adversarial expected runtime $O(1)$ for insertions, deletions, and search; similarly, the skip list and treap achieve non-adversarial expected runtime $O(\log s)$ for these operations on an ordered collection (but have other advantages such as supporting range queries).

In contrast to compressing probabilistic data structures (e.g., Bloom filters, count-min sketches, HyperLogLogs, etc.), PSDS always return a correct Qry response. Further, unlike self-balancing data structures (e.g., splay trees, red-black trees, sorted arrays, etc.), skipping data structures do not require complex update mechanisms to maintain favorable representations. That is, under non-adversarial conditions, using randomness is sufficient to facilitate efficient operational runtimes (with high probability) without the overhead of complex and potentially expensive rebalancing.

While this provides an intuitive notion of a skipping-based data structure, it fails to provide a formal or constructive definition. Therefore, let us consider the following. Take a hash table, whose representations are built over a size s set of elements (index keys) from the domain $\{0,1\}^\lambda$ by running them each through a hash function and putting them into a bucket depending on the output of this hash function. Under the assumption that the hash function behaves like a random function, the elements in the table can be viewed as an (unordered) sequence of i.i.d. random variables. That is, we can decompose a hash table's representation as $B_1, B_2, \ldots, B_s$ where $\forall i \in [s] : B_i \sim \mathsf{U}(\{1, 2, \ldots, b\})$, where b is the number of buckets for the particular structure. More generally, we can define the random variable decomposition for an unordered skipping-based data structure as follows.

Definition 4.1 (Unordered PSDS Random Variable Decomposition). Let $\Pi = (\textsc{Rep}, \textsc{Qry}, \textsc{Up})$ denote an unordered skipping-based probabilistic data structure using a key $K \in \mathcal{K}$. Let $M = \{x_1, x_2, \ldots, x_s\}$ denote a set and repr $\twoheadleftarrow \textsc{Rep}_K(M)$. The unordered random variable decomposition $\mathcal{D}(\text{repr}) = \{X_{x_i} : m(X_{x_i}) \mid i = 1, 2, \ldots, s\}$ represents the multiset of random variables associated with the contained elements, where $m(X_{x_i})$ denotes the multiplicity of random variable X_{x_i} and each X_{x_i} captures the probabilistic structural information for element $x_i \in M$.

For a skip list, we can take a similar view. Here, we again assume that a skip list represents a size s set of elements from the domain $\{0,1\}^\lambda$. Additionally, we assumed that the set is well-ordered. Under the non-adversarial assumption that all updates are made uniformly at random from the universe of all elements, the representation can be viewed as a sequence of ordered i.i.d. random variables (again, in the adaptive adversarial setting, independence of these random variables does not necessarily hold). We can decompose the skip list representation as the randomly assigned heights $H_1, H_2, \ldots, H_s$, where $\forall i \in [s] : H_i \sim \mathsf{G}(p)$ for the geometric distribution, where p is the probability parameter of the structure. That is, a skip list can be viewed as the ordered sequence of its elements' heights. The sequence of random variables $H_1, H_2, \ldots, H_s$ is sorted according to the order of the keys in the representation. Similarly, one can decompose the treap representation as $P_1, P_2, \ldots, P_s$ where $\forall i \in [n] : P_i \sim \mathsf{U}([0,1])$. This sequence of random variables represents the priority of elements in the treap, and the sequence is again ordered by the keys in the representation. That is, treaps can be viewed as a binary search tree where the order of insertion is determined by the randomly sampled priorities [49]. More generally, we can define the random variable decomposition for an ordered skipping-based data structure as follows.

Definition 4.2 (Ordered PSDS Random Variable Decomposition). Let $\Pi = (\textsc{Rep}, \textsc{Qry}, \textsc{Up})$ denote an unordered skipping-based probabilistic data structure using a key $K \in \mathcal{K}$. Let $M = \{x_1, x_2, \ldots, x_s\}$ denote a set and repr $\twoheadleftarrow \textsc{Rep}_K(M)$. The ordered random variable decomposition $\mathcal{D}(\text{repr}) = (X_{x_1}, X_{x_2}, \ldots, X_{x_s})$ represents the vector of random variables associated with the contained elements in sorted order, where each X_{x_i} captures the probabilistic structural information for element $x_i \in M$.

With this intuition built, we arrive at our definition for probabilistic skipping-based data structures.

Definition 4.3 (Probabilistic Skipping-Based Data Structure). A probabilistic skipping-based data structure that represents a size s collection of elements from the domain $\{0,1\}^\lambda$ is a data structure whose representation can be decomposed as a sequence of identically distributed random variables from some distributions $\mathcal{X}$. This sequence is either unordered (for data structures representing unordered data, like hash tables) or implicitly ordered by some well-defined ordering over the domain of the underlying collection (as is the case for ordered data structures, like skip lists and treaps).

This definition offers a few key benefits. First, from an attacker's perspective, it helps us formally specify the necessary conditions for an adversary to succeed in our security game. For hash table attacks, this means forcing a large portion of the discrete uniform random variables $B_1, B_2, \ldots, B_s$ to be equal – a condition any successful attack strategy must achieve to degenerate the data structure. Additionally, it allows us to precisely differentiate between adaptive and

non-adaptive adversarial capabilities. When decomposing a skip list into geometric random variables $H_1, H_2, \ldots, H_s$ (sorted according to index order), an adaptive adversary can observe previous outcomes and strategically insert a new H_i at any position in the sequence, thereby creating dependencies among the variables. In contrast, a non-adaptive adversary cannot observe previous geometric random variable outcomes, resulting in a final sequence $H_1, H_2, \ldots, H_s$ that maintains independence among the sequence of random variables.

Second, the stochastic formalization enables the application of well-established probabilistic techniques to derive tight bounds on adversarial success probabilities: balls-and-bins analysis for hash tables as well as martingale-based arguments for skip lists and treaps. Finally, for researchers looking at PSDS different from the ones we consider, it allows for the generalization of our robust data structures: proving security for one structure characterized by a particular sequence of identically distributed random variables allows us to transfer robustness techniques to other structures of the same type. Though specific structural details may prevent exact technique transfer, this approach should inform effective general strategies.

4.2 Towards Robust PSDS

We observe that two abilities allow an adaptive adversary to shape the distribution of data in a PSDS such that subsequent operations on the structures are degraded with high probability. The first is the ability to *delete* elements. This allows an adversary to degenerate a structure after a series of insertions by deleting unfavorably (w.r.t. to the adversary's goal) elements. The second is the ability of the adversary to influence *where* a particular element gets placed in the structure upon insertion. This is akin to knowing in advance which bucket an element will be inserted into in a hash table, at what height an element will be inserted in a skip list, or the priority an element will receive upon insertion into a treap. Therefore, we propose two inexpensive and general modifications to the base PSDS to make them robust in an adversarial setting. We will later prove these modified structures secure.

Lazy Deletion. Deletion operations have long posed fundamental challenges in the probabilistic data structure (PDS) domain, as demonstrated by the difficulties encountered in provably secure compact frequency estimators [17, 35]. Even under restrictive assumptions, such as limiting adversarial access to the data structure's internal state, theoretical bounds reveal the substantial vulnerabilities that deletions introduce [24].

In our threat model, where adversaries possess complete internal access to the data structure's state, deletions become particularly devastating. They enable adversaries to fundamentally alter the distribution of the underlying random variables that govern the probabilistic data structure's behavior. The core vulnerability is that deletions effectively allow an adversary to repeatedly "roll the dice" on random experiments until achieving an advantageous outcome, thereby completely subverting the randomness properties upon which the data structure's security guarantees depend. For instance, in skip lists, the security relies on the probabilistic property that approximately every p^{th} element resides on a higher layer; through strategic deletions, an adversary can manipulate the structure to violate this fundamental assumption.

To obtain provable guarantees, we must maintain an invariant on the data: after deletion operations, the underlying random variables that govern the probabilistic data structure's behavior must retain distributions that support provable runtime guarantees. This requirement can be satisfied through two distinct approaches: either deletions preserve the original distribution with minimal perturbation, or they induce controlled distributional changes that, while significant, still permit theoretical analysis and provable bounds. We focus on a generic method that preserves the original distribution with minimal perturbation at the potential cost of additional space complexity via an approach called *lazy deletion*. It remains an open question for which data structures there exists a space-efficient deletion mechanism specifically tailored to the underlying random variables of the respective probabilistic data structures.

Lazy deletion enables element removal without modifying the underlying probabilistic structure imposed by previous insertions in a PSDS. This preservation of the insertion-imposed distribution is crucial because it maintains the foundation for our provable guarantees. The approach operates by initially marking an element as "deleted" rather than removing it, then performing actual deletion only when the data structure reinitializes or when a fresh element is inserted at the same relative position (i.e., the same bucket in a hash table or in between the same elements in a skip list), effectively replacing the deleted element while preserving the same randomness properties. Note that this means the marked element remains in the data structure until replacement or re-initialization, though the data structure continues to function correctly due to the element's marking. The deletion implementation follows a consistent approach across all data structures under consideration: rather than immediately removing elements, we perform a logical deletion by replacing the target element's value with a distinguished sentinel value (denoted $\diamond$) while maintaining the original element key.

Formally, lazy deletion preserves the underlying probabilistic structure established by prior insertions. Specifically, given a fixed random tape, any sequence of update queries (including deletions) can be reduced to a sequence of insert-only queries of equal or shorter length that yields an identical probabilistic structure. This reduction demonstrates that deletions confer no strategic advantage to an adversary; they merely consume computational resources from the attack budget without altering the data structure's fundamental properties. We term this property of skip-list-based probabilistic data structures *deletion invariance*.

Definition 4.4 (PSDS Deletion Invariance). Let $\Pi = (\textsc{Rep}, \textsc{Qry}, \textsc{Up})$ denote a skipping-based probabilistic data structure using a key $K \in \mathcal{K}$. Π is *deletion invariant* under lazy deletion if for any sequence of update queries $S \in \mathcal{U}^m$, there exists an insert-only sequence $\widetilde{S} = S_I$ for an index set $I \subseteq \{1, \ldots, m\}$ such that when both sequences are executed for a fixed random tape r, the resulting representations repr, $\widetilde{\text{repr}}$ satisfy $\mathcal{D}(\text{repr}) = \mathcal{D}(\widetilde{\text{repr}})$, where $\mathcal{D}(\cdot)$ denotes the random variable decomposition of contained elements as defined in Definition 4.1 and Definition 4.2, respectively.

Lemma 4.5 (HT Deletion Invariance). *Let* Π *be a hash table. Then* Π *is* deletion invariant *under lazy deletion.*

The proof and the formal hash table specification can be found in the full version. The sequence $\widetilde{S}$ can be constructed from S in time $O(|S|^2)$.

Lemma 4.6 (SL/TR Deletion Invariance). *Let* Π *be a skip list or treap, respectively. Then* Π *is* deletion invariant *under lazy deletion.*

The proof and the formal data structure specifications can be found in the full version. The sequence $\widetilde{S}$ can be constructed from S in time $O(|S|^2)$.

Two PSDS D_1 and D_2 with identical decompositions exhibit equivalent runtime behavior. Specifically, for every element requiring t time steps to process in D_1, there exists a corresponding element in D_2 that also requires exactly t time steps. This equivalence ensures that any upper bound on the maximum runtime established for one structure applies directly to the other. This property will be leveraged throughout our subsequent proofs.

Lazy deletion is a fundamental design choice in probabilistic data structures where deleted elements remain as inactive nodes rather than being removed. This approach ensures that the random choices made during an element's initial insertion stay fixed in the structure, providing crucial protection against adversarial attacks.

Without lazy deletion, adaptive adversaries can exploit a simple attack. They insert many elements, then selectively delete those with unfavorable random outcomes. In hash tables, they remove elements that fail to map to target buckets. In skip lists, they delete nodes on higher levels. In treaps, they eliminate nodes with unwanted priorities. This selective deletion forces the structure into degenerate configurations, degrading search time from the expected $O(\log s)$ to $\Theta(s)$.

Lazy deletion prevents this attack. By preserving the random choices from the first at most n insertions (where n is the capacity of the structure), the data structure maintains its probabilistic guarantees under any deletion sequence. This property allows us to prove that search time remains $O(\log s)$ for a represented collection of size $s \leq n$ regardless of the adversary's deletion strategy.

The space overhead from deleted elements can be managed through two mechanisms. First, under non-adversarial conditions, new insertions can reuse positions of deleted entries with probability approximately k/s, where k is the number of deleted elements among s total positions. Second, periodic rebuilding removes all deleted entries in $O(s)$ time while reclaiming space and restoring $O(\log s')$ performance for the remaining s' elements. This rebuilding process is already standard practice for maintaining hash table load factors and skip list height bounds.

Adversarial Robustness. While lazy deletion is necessary for robustness, it alone is insufficient in most cases. The second requirement is minimizing an adversary's ability to predict or influence element placement. Each data structure requires tailored security mechanisms to achieve this goal.

The fundamental challenge differs between hash tables and ordered structures (skip lists and treaps). Hash tables determine bucket placement through hash function outputs rather than element keys, requiring identical hash results during both insertion and search. Conventional implementations use public hash functions, creating a vulnerability: adversaries can precalculate hash values and execute complexity attacks by targeting specific buckets. To counter this, we replace public hash functions with secretly keyed primitives that behave like truly random functions (Section 6), preventing adversarial precalculation. While this approach aligns with previous work, no formal analysis has been conducted.

Skip lists and treaps face different challenges. Both employ per-insertion randomness while maintaining key-based ordering, enabling range queries and order-dependent operations. However, simply applying secretly-keyed random functions would destroy this ordering property. We therefore develop order-preserving security mechanisms.

For skip lists, we introduce an unkeyed algorithmic approach. Although adversaries cannot precalculate coin flip outcomes determining element heights, they can still manipulate the structure by shifting unfavorable random outcomes to one side, effectively placing elements at chosen positions (Section 7.1). We counter this by enforcing *local* balance through constant-overhead swap operations, making such attacks exponentially more difficult (Section 7).

Treaps inherently resist manipulation better than skip lists. They automatically rebalance their entire structure based on the priorities of all inserted elements. As we demonstrate in Section 8, this global rebalancing property substantially reduces an adversary's control over element placement, requiring minimal additional security mechanisms.

5 A SECURITY MODEL FOR PROBABILISTIC SKIPPING-BASED STRUCTURES

Timing Side Channels. PSDS share a critical vulnerability: their runtime variation for distinct queries directly reveals information about their internal structure. This inherent timing side-channel has been successfully exploited in attacks against hash tables with (secret) salts [10] and skip lists [42]. For treaps, this vulnerability also manifests, as runtime correlates with node depth, potentially exposing the complete internal structure when combined with the ordering of the inserted elements. While remote attackers might face challenges like network latency in precisely measuring timing differences, recent research demonstrates that timing side-channels can be exploited with remarkable precision – as shown in [31], where researchers recovered an AES key from a Bluetooth chip's hardware accelerator.

Implementing enforced constant-time operations fails as a solution, as this would require the data structure to always operate at a worst-case (linear) time, defeating the purpose of using these efficient structures. Similarly, making the data structure oblivious to prevent information leakage has significant limitations. Such approaches are inherently fragile – once an adversary learns anything about the internal structure, the security guarantees collapse. Previous attempts to prevent information leakage in skip lists [42] by randomly swapping elements have proven unsuccessful.

Given these considerations, we adopt a more realistic approach by considering a very strong adversarial model. We grant the adversary full access to the internal structure of the PSDS, then prove that even with this knowledge, they cannot successfully degenerate the structure. This robust security model acknowledges that side channels inevitably exist in practical implementations and builds defenses that remain effective despite full information leakage. While this represents a strong adversarial capability, we argue it better reflects real-world threat scenarios than a model that assumes perfect or partial information hiding.

Informal Security for Probabilistic Skipping-based Structures. Our goal is to capture the average-case runtime of operations PSDS being *conserved* in the face of an adaptive adversary that can control the data represented by the structure. Loosely, the average-case runtime of PSDS relates to how data is "distributed" in the representation. For instance, an ideal hash table would distribute the elements it represents equally among the buckets. Analogously, ideal ordered structures (e.g., a skip list or a treap) would be isomorphic to a balanced tree. If a data collection were fixed, and we ignored a desire for efficiency, one could always craft an ideal representation with respect to the runtime of queries. For a hash table, one could find a hash function that equally distributes the fixed collection to its buckets. For a fixed-ordered structure, one could simply assign the heights (depths) of elements such that the shortest possible search paths are guaranteed, as with a perfectly balanced tree structure.

However, PSDS are used in mutable settings. For this reason (and for efficiency), PSDS use some form of randomness to process updates dynamically and update their representation. Hash tables select a random hash function to map elements to buckets, and ordered PSDS employ per-operation randomization during insertion to determine an element's position in the structure — typically through coin flips for skip lists or random priority assignments for treaps. These processes have been shown (with high probability) to yield representations of a dynamic data collection that are "close" to the ideal representations. Hash tables are analyzed using standard ball-and-bin arguments. Assuming a collision-resistant hash function and a load factor such that $s \leq \frac{2}{e} b \ln b$ (i.e., the size s of the data collection stored is less than or equal to $\frac{2}{e} b \ln b$, where b is the number of buckets), it is known [20] that with probability $p = 1 - \frac{1}{b}$ at any point in time no bucket has more than $\frac{4 \ln b}{\ln(\frac{2b}{se} \ln b)}$ entries. This maximum bucket population corresponds to a subsequent operation's maximum insertion, deletion, or query time. Likewise, the maximum search cost path of any element queried to a skip list or treap containing s elements has been shown to not exceed $O(\log s)$ with high probability (where the exact constants are functions of the parameters of the structure).

The above analyses are done under a strictly *non-adaptive* adversarial assumption. That is, these probabilistic bounds on the "distribution" of elements are done under the assumption that the updates and queries made to the structure do not depend on the internal randomness of the structure, the results of past operations, or the state of the representation. In the adaptive adversarial setting, this cannot be assumed. This is seen in both the hash flooding attack and the skip list degeneration attack [10, 19, 33, 42]. Therefore, intuitively, a robust PSDS would conserve the desired element distribution property of the structure with high probability, even in the face of an adaptive adversary. This is what we aim to capture with our formal security model.

Formal Security Model. Let $\Pi = (\textsc{Rep}, \textsc{Up}, \textsc{Qry})$ be a probabilistic skipping-based data structure. We define a notion of adversarial property conservation involving Π, a property function $\phi : \mathcal{D} \times \{0,1\}^* \to \mathbb{R}$, a target bound $\beta : \mathcal{P} \times \mathbb{Z}^+ \to \mathbb{R}$, a threshold $\epsilon \in \mathbb{R}, \epsilon > 0$, and a capacity parameter n.

$\mathbf{Exp}^{\text{aapc}}_{\Pi,\phi,\beta,\epsilon,n}(\mathcal{A})$

1 : $r \leftarrow 0; K \twoheadleftarrow \mathcal{K}$
2 : done $\twoheadleftarrow \mathcal{A}^{\mathbf{Rep},\mathbf{Up},\mathbf{Qry}}$
3 : **if** $|D| > n$: **return** 0
4 : **return** $\left\lfloor \frac{\phi(D, \text{repr})}{\beta(\mathcal{P}, |D|)} \geq \epsilon \right\rfloor$

$\mathbf{Hash}(X)$

1 : **if** $X \notin \mathcal{X}$: **return** $\perp$
2 : **if** $H[X] = \perp$
3 : $\quad X[X] \twoheadleftarrow \mathcal{Y}$
4 : **return** $H[X]$

$\mathbf{Rep}(C)$

1 : **if** $r = 1$: **return** $\perp$
2 : $r \leftarrow 1$
3 : repr $\twoheadleftarrow \textsc{Rep}_K(C)$
4 : $D \leftarrow C$
5 : **return** repr

$\mathbf{Up}(\text{up})$

1 : repr $\twoheadleftarrow \textsc{Up}_K(\text{repr}, \text{up})$
2 : $D \leftarrow \text{up}(D)$
3 : **return** repr

$\mathbf{Qry}(\text{qry})$

1 : **return** $\textsc{Qry}_K(\text{repr}, \text{qry})$

Figure 1: The Adaptive Adversary Property Conservation (AAPC) security game. The experiment enforces that the adversary is only able to call Rep once. The experiment returns the output of a predicate that returns 1 iff the property function $\phi(D, \text{repr})$ computed over the representation the adversary interacts with is greater than ϵ-times (for some $\epsilon > 0$) larger than some target bound β (that only depends on the parameters of the structure $\mathcal{P}$ and the size of the represented data object $|D|$). The experiment always returns 0 if the size of the adversarially crafted representation (in terms of the number of elements) is greater than the capacity parameter n. The Hash oracle computes a random mapping $\{X\} \to \{Y\}$ (i.e., a random oracle), and is implicitly provided to Rep, Up and Qry as needed.

A property function ϕ takes as input the data object $D \subseteq \mathcal{D}$ represented by repr (the representation the adversary produces during its execution) and the representation repr itself and outputs a value that indicates the concrete property for the given adversarially chosen data collection and corresponding representation. This function represents the desired property one would like to conserve. For all structures of interest, this is the maximum search path cost over all elements $d \in D$. [‡] The intuition is that a complexity attack is deemed successful precisely when it significantly increases the maximum search path cost; therefore, a robust data structure must maintain nearly equivalent worst-case performance (with high probability) regardless of adversarial manipulation. Due to space constraints, we give the formal specification of the property function for hash tables, skip lists, and treaps in the full version.

A target bound β takes as input the structure parameters $\mathcal{P}$ (e.g., the number of buckets for a given hash table) and the size of the represented data object $|\mathcal{D}|$ in terms of the number of elements in the representation (denoted s below and throughout the rest of this work), and outputs the resulting bound value. We choose a target bound such that it corresponds to the known non-adaptive

[‡]This property sufficiently captures the search path cost of any d in the universe of all possible elements, as a search for an element not in the representation terminates with at most one more pointer traversal compared to any element in the representation.

bound for the property we want to conserve. For the hash table maximum search path cost, this is $\beta(\langle b\rangle, s) = \frac{4 \ln b}{\ln(\frac{2b}{se} \ln b)}$. For the skip list and treap maximum search path, we chose $\beta(\langle p, m\rangle, n) = c \log_{1/p}(s)$ (for a small constant c), and $\beta(\langle\rangle, s) = 2 \lg s) + 1$, respectively, as these are the (blunt) non-adversarial expected search path lengths [46, 47].

We also have a capacity parameter n. We enforce that the size s of the adversarially crafted representation (in terms of the number of elements in the representation) must be less than or equal to n. This reflects the usual behavior of PSDS, as they usually have an explicit capacity based on other structural parameters or operational concerns. For instance, a hash table is initiated with a load factor: the number of elements in a hash table must not exceed some value, or resizing and reconstruction are necessary. A skip list is limited to having less than or equal to $n = \frac{1}{p}^m$ elements (where p, m are parameters of the structure), else its probabilistic runtime guarantees become vacuous. A treap has no inherent capacity, but in practice may be limited due to hardware constraints or a targeted threshold for lookup times.

We give this notion of adversarial property conservation in Figure 1. The *Adaptive Adversary Property Conservation* (AAPC) experiment aims to capture an adversary's ability to adaptively craft a representation repr of some dynamic and adversarially decided data object D, such that when the property function ϕ is computed, the ratio of its output to the target bound's output is large (to win the experiment this ratio needs to exceed ϵ). If the adversarially decide data object D's size exceeds the capacity n, then 0 is output. As the properties (and their accompanying target bounds) measure how data elements are distributed in a particular representation (and bound how they are distributed in the non-adaptive setting), this notion directly translates to an adversary's ability to disrupt the expected runtime of a data structure's operations.

The AAPC experiment begins by setting a parameter $r = 0$ and selecting a key K from the key space $\mathcal{K}$. For unkeyed hash tables (insecure) and non-deterministic versions of the ordered PSDS, the key space is the empty set. The adversary is then allowed to instantiate the data structure with any initial data object C (including the empty data object) via the **Rep** oracle and receives back the resulting representation. We enforce that the adversary is only allowed to call **Rep** once via the parameter r. This is to disallow the adversary from leveraging past information from a data structure that is keyed with the same key K to trivially win the game. That is, keyed hash tables and deterministic PSDS must sample a fresh random key to guarantee security.

The adversary is then allowed to make any sequence of **Up** and **Qry** calls. Upon each update, we also update the internal data object D kept by the experiment, as this is used for computing ϕ and β. After each update, the updated representation repr is returned to the adversary. Thus, the notion of security we propose is quite strong in that it allows an adversary to have complete access to the structure's internals during its execution (as discussed in Section 5). The only information kept from the adversary is the secret key (in the case that the structure relies on one). This further makes calls to **Qry** unnecessary, as the adversary entirely determines the underlying collection represented by the structure and has access to the internal representation at all times.

The adversary ends its execution by announcing **done** or is implicitly done when it exhausts its **Up** budget (the number of updates they are allowed to make). The experiment concludes by outputting a bit that determines whether the adversary has successfully met the winning condition. With this intuition built, we give our succinct formal definition of security.

Definition 5.1 ($(\phi, \beta, \epsilon, \delta, n, t)$-Conserved). We say a skipping-based probabilistic data structure Π is $(\phi, \beta, \epsilon, \delta, n, t)$-conserved if the advantage of an AAPC-adversary $\mathcal{A}$ running in time t is less-than-or-equal to δ for some property function ϕ, some target bound β, some $\epsilon \in \mathbb{R}, \epsilon > 0$, some $\delta \in [0, 1)$, and some capacity $n > 0$. More precisely, we say the structure is $(\phi, \epsilon, \beta, \delta, n, t)$-conserved iff,

$$\mathbf{Adv}^{\text{aapc}}_{\Pi,\phi,\beta,\epsilon,n}(\mathcal{A}) = \Pr[\mathbf{Exp}^{\text{aapc}}_{\Pi,\phi,\beta,\epsilon}(\mathcal{A}) = 1] \leq \delta$$

and write $\mathbf{Adv}^{\text{aapc[u,v]}}_{\Pi,\phi,\beta,\epsilon}(t, q_Q, q_U, q_H)$ as the maximum advantage of any AAPC-adversary running in t time steps and allowed to make at most q_Q calls to **Qry**, q_U calls to **Up**, and q_H calls to **Hash** in the ROM. We are interested in ensuring $\mathbf{Adv}^{\text{aapc}}_{\Pi,\phi,\beta,\epsilon,n}(t, q_Q, q_U, q_H) \leq \delta$.

6 ROBUST HASH TABLES

6.1 Insecurity Of Standard Hash Tables

Unkeyed Hash Tables. Consider a standard hash table instantiated with a fixed and publicly known hash function. A simple pre-computation attack will trivially win our security experiment with probability one (assuming the ability to make sufficiently many local hash computations). An adversary can sample index keys from the universe and compute the bucket they will map to by using the public hash function (assuming the parameters of the structure are known). The adversary can select a target bucket and insert index keys (with some arbitrary value) iff they map to this target bucket. In this way, an adversary can ensure that all elements go to a single bucket, causing a linear overhead when searching for an element in this bucket.

Keyed Hash Tables with Deletions. Consider a hash table where we replace a public hash function with a secretly keyed primitive, like a PRF. Our security game also yields a simple strategy for an adversary to win our game with a high probability if the hash tables support deletions in the usual way. The adversary selects a target bucket. Then it samples keys from the universe (along with arbitrary values for these keys) and inserts them into the table. Observing the state of the table after each insertion, the adversary deletes the element unless it has been inserted in the target bucket. At the end of the adversary's execution, the hash table will only have elements that reside in a single bucket. For this reason, we do not allow adversaries to make deletions that actually remove elements from the hash table and compare our adversarial results to a standard ball-in-bins result that assumes no deletions.

While an attack of this nature may seem vacuous and an artifact of our security experiment, it is designed to capture something more complex. Consider if you could guarantee that the state of the hash table could remain hidden during the adversary's execution. Then, it seems intuitive that just keying the structure would result in robust construction per our security definition. However, as evidenced by side-channel attacks against hash tables [10], it is

nearly impossible to guarantee that the internal structure of the hash table remains entirely hidden. Therefore, we continually leak the entire state of the structure to the adversary during its execution to emulate the best possible side channel (as detailed in Section 5).

6.2 A Robust Construction

The robust hash table requires that a keyed mapping function R is used. Concretely, this can be instantiated as PRF that is then mapped to b (by, say, taking the output of the PRF modulo b). In particular, SipHash [8] provides performance that is comparable to traditionally used non-cryptographic hash functions [43]. We also use our modified deletion scheme. The deletion functionality simply replaces the value of the respective key-value pair with a distinguished symbol $\diamond$. The insertion functionality changes such that if an element to be inserted can overwrite a linked list node containing $(z, \diamond)$ for any key z, it does; otherwise, a normal insertion occurs. The query functionality remains unchanged. We give a formal pseudocode description of the robust hash table construction in the full version.

We will now start a formal security theorem and prove the robust hash table construction secure in the AAPC model. Due to space constraints, we provide a proof sketch here and give the full proof in the full version.

Theorem 6.1 (Robust Hash Table AAPC Security Result). *Let Π be our robust hash table using PRF F to map elements to buckets. For integers $q_U, q_Q, q_H, n, t \geq 0$ such that $q_U \geq n \leq \frac{2}{e} b \ln b$ (where b is the number of buckets in the hash table Π), it holds that Π is $(\phi, \beta, \epsilon, \delta, n, t)$-conserved with ϕ being the HT Maximum Bucket Population function, $\beta = \frac{4 \ln b}{\ln(\frac{2b}{se} \ln b)}$, $\epsilon = 1$, and $\delta = (\frac{1}{b} + \mathbf{Adv}_F^{\text{prf}}(O(t), n + q_Q))$, with $s \leq n$ denoting the number of elements contained in the data structure.*

Proof Sketch. We first observe a key property of our construction: the modified insertion and deletion procedures ensure elements are never truly removed, only marked as deleted. By Lemma 4.5, there exists an insert-only adversary that produces an equivalent decomposition (and runtime behavior). Therefore, it suffices to upper bound the maximum runtime of any insert-only adversary who can maximally make up to $s \leq n$ fresh insertions.

Then our proof proceeds via a hybrid argument:

1. Let $\mathbf{G}_0$ be the original AAPC security game with our robust hash table Π using PRF F, property function ϕ (the Maximum Bucket Population function), and target bound $\beta = \frac{4 \ln b}{\ln(\frac{2b}{se} \ln b)}$.

2. Let $\mathbf{G}_1$ be identical to $\mathbf{G}_0$ except we replace the PRF with truly random function. We build a PRF distinguisher $\mathcal{B}$ such that:

$$\mathbf{Adv}_F^{\text{prf}}(\mathcal{B}) = \Pr[\mathbf{G}_0(\mathcal{A}) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}) = 1] \tag{1}$$

$\mathcal{B}^F$ works by executing $\mathcal{A}$ in $\mathbf{G}_1$, using its own oracle whenever F is called, and outputs 1 if $\mathcal{A}$ wins, 0 otherwise.

3. In $\mathbf{G}_1$, we now have a standard balls-and-bins problem with $\leq n = \frac{2}{e} b \ln b$ balls randomly thrown into b bins. By applying the standard bound for this problem, we conclude that $\phi(\cdot) \leq \beta(\cdot)$ with probability $1 - \delta$ where $\delta = \frac{1}{b} + \mathbf{Adv}_F^{\text{prf}}(O(t), n + q_Q)$. □

To give a concrete illustration of this bound, suppose we had $s = n = b = 2^{32}$ and $\epsilon = 1$. Leveraging our results from Theorem 6.1, the probability our maximum search cost path is greater than $M = \frac{4 \ln(2^{32})}{\ln(\frac{2^{33}}{2^{32}e} \ln 2^{32})} \approx 31.77$ is less than or equal to $\delta = \frac{1}{2^{32}} + \mathbf{Adv}_F^{\text{prf}}(O(t), b + q_Q) \approx 2.33 \cdot 10^{-10} + \mathbf{Adv}_F^{\text{prf}}(O(t), b + q_Q)$.

6.3 Limitations and Real World Deployments

Our robust hash table construction faces two primary limitations that impose practical constraints on its deployment.

Performance Overhead. The transition to cryptographic primitives introduces reasonable computational costs in our robust constructions. Paterson and Raynal [43] demonstrate that cryptographic hash functions like SipHash(2,4) maintain competitive performance, requiring only 102 ns per evaluation on 90-byte inputs versus 60 ns for MurmurHash3, while HMAC takes 705 ns. This shows that the computational overhead of switching to secure hash functions remains modest relative to the security benefits gained.

Space Overhead. When initializing a hash table, there's an implicit promise to allocate enough memory for a pre-defined number of elements. If the collection grows too large and exceeds this capacity, the structure must be resized, typically by doubling the number of buckets. For an index-value pair where indexes are x bits and values are v bits, we expect to allocate up to $\alpha \cdot b \cdot x \cdot v$ bits of memory, where α is the load factor defined as $\alpha = \frac{s}{b}$, with s being the number of elements and b the number of buckets [18]. If the load factor exceeds a set limit, resizing is required. In our security experiment, we implicitly specify a load factor by defining a capacity n for our structure, and in turn, never allow this load factor to be exceeded. That is, we do not consider attacks that would trigger resizing. Hence, we discuss the consequences of our robust construction in real-world deployments below by analyzing how our modifications change the frequency of required resizing.

Consider a standard hash table with I successful insertions and D successful deletions. For resizing to be necessary, it must be that $\frac{I-D}{b}$ has exceeded α. At some point, before resizing is triggered, if the rate of insertions and deletions is roughly equal, a structure could persist indefinitely without resizing.

Now consider a modification where deletions merely mark elements as deleted without allowing for the possibility of being replaced by fresh insertions. That is, we do not modify the insertion procedure to replace previously deleted elements. In this scenario, resizing occurs when $\frac{I}{b}$ has surpassed α, even if only a few elements are actually represented in the structure. This could occur when an adversary inserts $\approx \alpha \cdot b$ elements, then deletes nearly all of them[§], and finally triggers a resizing with a few subsequent fresh insertions. Although this seems wasteful, it aligns with the resizing logic since the total insertions exceed the threshold. That is, a resizing is triggered only after the total number of insertions exceeds the threshold set by α (regardless of any deletions).

We would like our robust hash table to conserve the property where deletions free space, such that $\frac{I}{b} > \alpha$ does not necessarily trigger a resizing. Thus, in addition to marking deleted elements, we also replace deleted elements with new insertions. This is desirable in the non-adversarial case (where insertions, deletions, and queries

[§]Of course, if an adversary deleted all elements, it would be trivial to flush the table and reinitialize the structure.

do not depend on the internal randomness of the structure, the internal state of the structure, or past operations), as one expects freshly inserted elements will eventually replace deleted elements.

Adversarial strategies can still trigger resizing with few non-deleted elements. For example, an adversary could insert $I = \alpha \cdot b - 1$ elements, delete all but those in the least populated bucket, and with a $1/b$ probability, trigger resizing with only those elements in that bucket remaining. While this requires the adversary to exceed the threshold number of insertions, making it marginally problematic in practice, the collection size at the time of resizing may be smaller than the non-adversarial threshold. In sum, while adversaries can still trigger small collection resizing under certain conditions, our approach ensures the hash table is provably robust and allows it to persist for extended periods without resizing if insertions and deletions are balanced in the non-adversarial setting.

7 ROBUST SKIP LISTS

7.1 Insecurity of Standard Skip Lists

As noted in [46], the heights of the elements in the skip list must be kept secret, or otherwise, the skip list can be degenerated by simply deleting all elements in the list that are not at height zero. In our security model, this attack is trivial. However, even when disallowing deletions (or using our modified deletion), an adaptive adversary can still degenerate the skip list using a powerful but subtle strategy. We call this the *gap attack* and detail it next, but use an intuitive rather than a formal description.

We assume the skip list takes values from $\{0,1\}^n$, which we interpret as integers between 0 and $2^n - 1$. The gap attacker proceeds as in Figure 2. It starts by inserting an element in the middle $M = 2^{n-1}$ of the interval $[L, R] = [0, 2^n]$. If this element gets assigned a height of 0 in the skip list, i.e., is only inserted in the bottom list, then the attacker secures it by shifting the left bound L of the interval to M, moving to that gap. In the other case, if the height is large, it "gives up" this part of the skip list and moves the right bound R of the interval to M. Continue with the new interval $[L', R']$ of half the size until n elements have been inserted.

Gap attack on skip list

```
1 : L ← 0, R ← 2^n
2 : for i ← 1 to n
3 :     insert element M ← (R + L)/2
4 :     if height(M) = 0 then L ← M else R ← M
```

Figure 2: The gap attack on skip lists, inserting n elements from $\{0,1\}^n$.

By construction, the value M in each iteration is always an integer between L and R. Moreover, at the end of each iteration, there are only elements of height 0 in the interval $[0, L]$ (if any), and all elements of larger height in $[R, 2^n]$ (if any), since we set the left resp. right bound accordingly in each iteration. Hence, after n iterations and $R - L = 1$ we have all elements of height 0 in $[0, L]$, and elements of larger height in $[L + 1, 2^n]$. In each iteration, we insert an element of height 0 with constant probability $1 - p$, which will eventually lie in the interval $[0, L]$. Therefore, the expected number of elements in $[0, L]$ is $(1-p)n$. The resulting skip list is now highly degenerated in the interval $[0, L]$. Specifically, it corresponds to a linked list of average length $(1 - p)n$. Hence, the search for the element L takes linear time on average, whereas a regular skip list would yield a logarithmic average time. This is an exponential blow-up in running time, which the gap attacker enforces.

7.2 A Robust Construction

In our robust skip list construction, elements can be marked as deleted by setting the value of a key-value pair to a distinguished symbol $\diamond$. We highlight that deleting and reinserting an element does not change the associated height, as only the value is replaced.

We use a simple swapping mechanism to make the skip list robust (a cutout of the complete insert algorithm picturing only the novel swapping mechanism is depicted in Figure 3). When inserting element x, the algorithm first performs standard insertion by searching down from the top level to find the correct position, storing in $u[i]$ the rightmost node at each level i with key less than x. If the key exists, the algorithm updates the value; otherwise, it generates a random level ℓ, creates a new node, and splices it into the structure by updating forward pointers of predecessor nodes in u. Subsequently, the swapping procedure uses the update vector u from insertion. After x is inserted at level ℓ, the mechanism counts nodes at level $\ell - 1$ between x's predecessor $u[\ell]$ and x's successor $x[\ell]$, both at level ℓ. Then it identifies the middle element using the tortoise and hare algorithm [34]. This is done by moving a slow pointer middle and a fast pointer fast with twice the speed on the nodes on level $\ell - 1$, starting with the node $u[\ell]$ pointing to the node left of x on the level ℓ. We end the search if fast has reached the successor of x or points to it. If the middle element is not x itself (verified by checking $\ell <$ middle.level), a height swap occurs: the middle element's height increases to level ℓ while x's height decreases to level $\ell - 1$, effectively exchanging their heights.

Swapping mechanism for robust skip lists

```
1 :  / find layer ℓ − 1 middle element using tortoise and hare
2 :  if ℓ = 0 then return ⊥
3 :  middle ← u[ℓ], fast ← u[ℓ]
4 :  / ensure that the fast pointer doesn't go past the successor x[ℓ]
5 :  while fast ≠ x[ℓ] and fast[ℓ − 1] ≠ x[ℓ] do
6 :      middle ← middle[ℓ − 1], fast ← fast[ℓ − 1][ℓ − 1]
7 :  / swapping logic
8 :  if ℓ > middle.level then
9 :      middle.append(x[ℓ]), x ← x[0 : ℓ − 1] || x[ℓ + 1]
10 :     u[ℓ][ℓ] ← middle
```

Figure 3: The swapping mechanism for robust skip lists, which is invoked after a node x has been inserted on layer ℓ, and the update vector u has been constructed during this process. This is only a cutout of the complete insert procedure as depicted in the full version.

This mechanism locally balances the skip list, preventing adversaries from creating large sequences of same-height elements that would result in a search path blowup. The gap attack specifically becomes highly infeasible, as elements of a fixed height can no longer be shifted toward one side of the data structure. Instead, heights are immediately swapped at the interval's midpoint, halving long sequences of elements on level $\ell - 1$. Note, this approach effectively handles corner cases where ℓ = list.header or $n[\ell]$ = null. Moreover, the interval typically contains a constant, denoted a, number of nodes with overwhelming probability (for obtaining this value a see Theorem 7.1), ensuring the mechanism operates in constant time with high probability. We give a formal pseudocode description of the robust skip list construction in the full version.

As we will see, this surprisingly simple mechanism allows us to prove the robust skip list construction secure in the AAPC model using the following formal security theorem. Due to space constraints, we provide a proof sketch here and give the full proof (with all intermediary lemmas) in the full version.

Theorem 7.1 (Robust Skip List AAPC Security Result). *Let Π be our robust skip list with parameters $p \in [0,1]$ and $m \geq 0$. For integers $q_U, q_Q, n, t \geq 0$, such that $q_U \geq n = \frac{1}{p}^m$, it holds that Π is $(\phi, \beta, \epsilon, \delta, n, t)$-conserved with ϕ being the Maximum Search Path Cost function, $\beta = c\log_{1/p}(s)$, with $s \leq n$ denoting the number of elements contained in the data structure, $\epsilon > 0$, and*

$$\delta = e^{(\lambda^* a) - (\epsilon\lambda^* a)} + e^{-\frac{((1-p)a\log_{1/p}(n)-1)^2}{(1-p)(2+(1-p)a\log_{1/p}(n)-1)}},$$

where $a = \frac{2(1+p)}{p}$, $c = a(1+\epsilon)$, and λ^* is the maximal solution $\lambda > 0$ to

$$(1-p)e^{\lambda} + p(1-p)e^{-\lambda\left(\frac{1}{p}+\frac{a}{2}\right)} + p^2 \leq 1.$$

Proof Sketch. We first observe a key property of our construction: the modified insertion and deletion procedures ensure elements are never truly removed, only marked as deleted. By Lemma 4.6, there exists an insert-only adversary that produces an equivalent decomposition (and runtime behavior). Therefore, it suffices to upper bound the maximum runtime of any insert-only adversary who can maximally make up to $s \leq n$ fresh insertions.

We show that our robust skip list is secure by analyzing the necessary conditions for degrading its performance. Any attack must either (1) create wide gaps on some level below $L(n) = \log_{1/p}(n)$, or (2) rely on many elements being assigned heights above $L(n)$.

We first prove that if on all levels $i < L(n)$ the number of level i elements between any pair of level $i+1$ elements are at most w, then any search path is of length at most $2w\log_{1/p} s$, unless the total number of elements above level $L(n)$ exceeds $w\log_{1/p} s$, where s is the size of the adversarially crafted representation.

Our robust deletion method ensures the adversary cannot directly influence element heights. Hence, the adversary's ability to create wide gaps reduces to a coin-flipping game where each element's height is determined by flipping a coin (heads with probability p) until tails appears. We track a random variable X_t representing gap width at time t, which updates to X_{t+1} as: $X_t + 1$ with probability $1 - p$ (i.e., extends the gap), $\lceil X_t/2 \rceil$ with probability $p(1-p)$ (i.e., gap is halved due to swapping), X_t with the remaining probability p^2 (i.e., gap is unchanged).

For the base layer (layer 0) of the Skip List, the expected value of this stochastic process is a fixed point $a = \frac{2(1+p)}{p}$. We then prove this process is a supermartingale and lift this result to the maximum over all layers using a chaining argument. We show that, for any $\epsilon > 0$, the probability of any level having gaps exceeding $a(1+\epsilon)$ is at most $e^{(\lambda^* a) - (\epsilon\lambda^* a)}$, where λ^* is the maximal solution to $(1-p)e^{\lambda} + p(1-p)e^{-\lambda(\frac{1}{p}+\frac{a}{2})} + p^2 \leq 1$. Further, we prove that the expected number of elements above level $L(n)$ is $\frac{1}{1-p}$, and the probability that this exceeds $a\log_{1/p}(s)$ is at most

$$e^{-\frac{((1-p)a\log_{1/p}(n)-1)^2}{(1-p)(2+(1-p)a\log_{1/p}(n)-1)}}.$$

Combining these bounds yields our final result. □

To give a concrete illustration of this bound, suppose we had $s = n = 2^{32}$, $p = \frac{1}{2}$. Then our fixed point $a = 6$, and solving for λ^* numerically yields $\lambda^* = 0.34$. Then, choosing $\epsilon = 8$ (hence, $c = 54$), the probability that the maximum search cost path exceeds $2 * c\log_{1/p}(s) = 108\log_{1/p}(s)$ is less than or equal to $\delta \approx 6.28 \times 10^{-7}$. While a constant $2c = 108$ may appear large, consider that, λ^* solely depends on p. In turn, for any fixed ϵ, this bound is constant as $s \to \infty$, showing that our adaptive search path is indeed $O(\log s)$. We further remark that this constant is likely "artificially" large, in the sense that the stochastic process we bound is complex, leaving us only to be able to use blunt Markov-like concentration bounds.

7.3 Limitations and Real World Deployments

Our robust skip list construction faces two primary limitations that impose practical constraints on its deployment.

Performance Overhead. The proposed swapping mechanism runs in time proportional to the interval length, which is constant with overwhelming probability (see the full version for a formal analysis). Here, we use the fact that in our AAPC proof for robust skip lists, we establish bounds on the number of sequential elements within or above a layer in the robust construction. Our swapping mechanism is dominated by a single pass over this bounded interval, enabling it to run in time proportional to the interval length, which is a constant with overwhelming probability.

Space Overhead. Skip lists, like hash tables, have an explicit capacity for a set number of elements and require resizing when exceeded. While skip lists do not require upfront memory allocation, they require setting a maximum node height of $m = \log_{\frac{1}{p}} n$ for a maximal n insertions. Exceeding n insertions necessitates resizing as the probabilistic guarantees otherwise deteriorate [46]. Our security analysis avoids considering attacks that trigger reinitialization by enforcing such a capacity. We now evaluate how our modifications affect resizing frequency in practice.

Standard skip lists with I successful insertions and D successful deletions require resizing when $\log_{\frac{1}{p}}(I - D) > m$. Previously, structures could operate indefinitely without resizing if insertion and deletion rates remained balanced. Our modified structure, employing lazy deletion, requires resizing when approximately $\log_{\frac{1}{p}}(I) > m$ regardless of remaining elements. This allows adversaries to

trigger early resizing by inserting approximately $\left(\frac{1}{p}\right)^m$ elements, deleting most, then forcing a resize with few additional insertions.

Note that, similar to hash tables, we can replace deleted nodes with new insertions when the new element maintains the same relative position within the data structure's ordering. In a non-adversarial setting, this approach preserves the structural invariants while potentially deferring costly re-initialization operations, thereby improving overall efficiency. In an adversarial setting, the adversary can simply not insert in the same relative position as a deleted element.

8 ROBUST TREAPS

8.1 (In)Security of the Standard Treap

Unlike other probabilistic data structures in this study, treaps (without deletions) already demonstrate intrinsic security against search path cost blow-up. However, adaptive adversaries can still mount attacks that force certain elements to be near the root with high probability. We discuss such an attack in the full version.

Importantly, for our purposes, treaps maintain their expected $O(\log s)$ operational complexity against adaptive adversaries only when our modified deletion procedure is applied. Without it, an adversary could simply re-insert (and delete) an element until obtaining a favorable priority, making degeneration attacks trivial.

This resistance to performance degradation attacks under lazy deletion represents a significant finding, as all other PSDS examined proved vulnerable. The treap's rebalancing mechanism, based on previously sampled priorities, provides a natural defense against malicious attempts to create operation sequences that would otherwise lead to worst-case runtime scenarios.

8.2 A Robust Construction

In our robust treap construction, elements can be marked as deleted by setting a "deleted" bit d, which is stored in the node as $\top$. It is important to point out that deleting and reinserting an element does not change the associated priority, as only the d bit is flipped to $\bot$. We give a formal pseudocode description of the robust treap construction in the full version.

This technique alone is sufficient to prove the robust treap construction secure in the AAPC model using the following formal security theorem. Due to space constraints, we provide a proof sketch here and give the full proof (with all intermediary lemmas) in the full version.

Theorem 8.1 (Treap AAPC Result). *Let Π be our robust treap. For integers $q_U, q_Q, n, t \geq 0$, such that $q_U \geq n$, it holds that Π is $(\phi, \beta, \epsilon, \delta, n, t)$-conserved with ϕ being the Maximum Search Path Cost function, $\beta = 2\ln s + 1$, with $s \leq n$ denoting the number of elements contained in the data structure, any $\epsilon > 0$ and*

$$\delta = se^{-\frac{\epsilon^2 H_s}{2(1+\epsilon)}},$$

where H_s is the s^{th} harmonic number.

Proof Sketch. We first observe a key property of our construction: the modified insertion and deletion procedures ensure elements are never truly removed, only marked as deleted. By Lemma 4.6, there exists an insert-only adversary that produces an equivalent decomposition (and runtime behavior). Therefore, it suffices to upper bound the maximum runtime of any insert-only adversary who can maximally make up to $s \leq n$ fresh elements.

For any element x_i in the treap, its search path consists of all ancestors. An element x_j is an ancestor of x_i if and only if it has the minimum priority among elements between x_i and x_j (inclusively). This means the search path length $S^s_{x_i}$ equals the sum of "records" (priority minima) in two sequences extending left and right from x_i, minus 1.

We first show that even under adaptive insertions, the probability that with respect to any element x, a freshly inserted element $\tilde{x}$ with j elements between them (inclusive) forms a record remains exactly $\frac{1}{j}$.

Let X_s denote the total number of records over all s fresh insertions for any particular element x. We have $\mathbb{E}[X_s] = \sum_{j=1}^{s} \frac{1}{j} = H_s$ (the s^{th} harmonic number). Since the search path length $S^s_x \leq 2X_s - 1$, we need to show X_s concentrates around its expectation.

We construct a Doob martingale where: $M_j = \sum_{i=1}^{j} (I_i - \mathbb{E}[I_i|\mathcal{F}_{i-1}])$ with $M_0 = 0$, where $\mathcal{F}_{i-1}$ is a σ-algebra containing the ordered priority values and all adversarial decisions regarding the first $j-1$ insertions. The martingale difference is $|D_j| \leq 1$, and the expectation of the predictable quadratic variation is $\mathbb{E}[V_s] \leq H_s$. Applying Freedman's inequality with $a = \epsilon H_s$ and $b = H_s$, we get:

$$\Pr(M_s \geq \epsilon H_s) \leq e^{-\frac{\epsilon^2 H_s}{2(1+\epsilon)}}$$

This shows that with high probability, $X_s \leq (1+\epsilon)H_s$, which implies $S^s_x \leq 2(1+\epsilon)H_s - 1$. Using $H_s \leq \ln(s) + 1$ and a union bound over all s elements in the treap yields our final result. □

To give a concrete illustration of this bound, suppose we had $s = n = 2^{32}$ and select $\epsilon = 5$. Our expected search path cost is $2\ln(2^{32}) + 1 \approx 45.36$, and leveraging our results from Theorem 8.1 the probability the maximum search cost path exceeds this by five times is $\leq \delta = 2^{32} \cdot e^{-\frac{25 H_{2^{32}}}{12}} \approx 6.65 \times 10^{-12}$.

8.3 Limitations and Real World Deployments

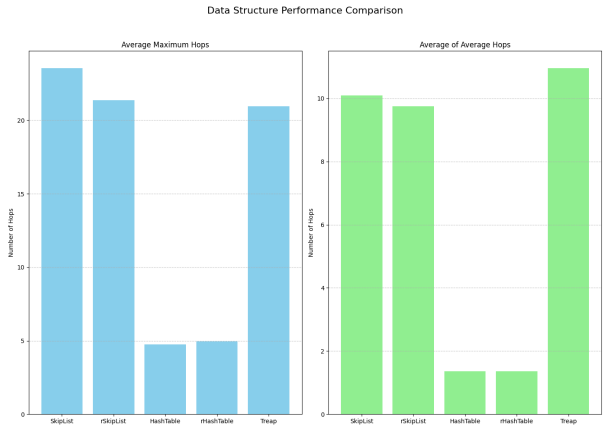
Our robust treap construction faces one primary limitation that imposes practical constraints on its deployment.

Space Overhead. Unlike hash tables and skip lists, treaps operate without an implicit maximum capacity and typically don't require resizing operations. However, our modified structure, employing lazy deletion, may still necessitate periodic treap re-initialization to reclaim memory occupied by deleted elements.

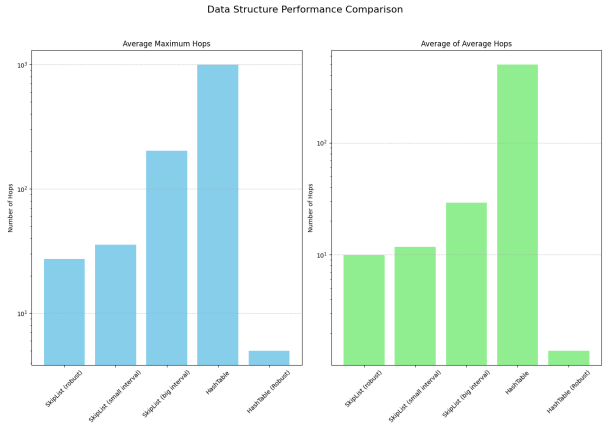
Note that, similar to hash tables, we can replace deleted nodes with new insertions when the new element maintains the same relative position within the data structure's ordering. In a non-adversarial setting, this approach preserves the structural invariants while potentially deferring costly re-initialization operations, thereby improving overall efficiency. In an adversarial setting, the adversary can simply not insert in the same relative position as a deleted element.

9 EXPERIMENTS

We conducted experiments to empirically validate our analytical results. The implementation will be published beside this paper.



(a) Maximum and average hop count in the non-adaptive setting, displayed on a linear scale.



(b) Maximum and average hop count in the adaptive setting, displayed on a logarithmic scale.

Figure 4: Maximum and average hop count for (robust) data structures in both the non-adaptive and adaptive settings.

Our first experiment tested whether robust data structures offer benefits in non-adversarial settings. Using a dataset of 10 million usernames [15], we randomly inserted 1,000 usernames into each data structure. We measured performance by counting hops (forward movements between nodes). The hash table's load factor was limited to 0.7 [36], and the skip list's maximum height was set to $\log_2 n$. We used Python's built-in hash function, which is vulnerable to multi-collision attacks. Results were averaged over 100 trials.

As shown in Figure 4a, the robust skip list consistently required fewer mean and maximum hops than its standard counterpart, demonstrating benefits even in non-adversarial settings with only constant overhead. The robust hash table showed comparable performance to the original structure. We benchmarked an unmodified treap implementation given its inherent adversarial robustness.

Our second experiment evaluated performance under adaptive adversarial conditions. We implemented a hash collision attack on hash tables and a gap attack on skip lists, averaging results over 100 trials. Treaps were excluded due to their established robustness.

For the hash collision attack, we pre-calculated bucket values to deliberately insert all elements into a single bucket, creating worst-case conditions for standard hash tables. For the robust implementation, we used a random key for pre-calculation, since the actual secret key would be unknown to an attacker.

For the gap attack against skip lists, we tested two variants: a restricted version using the same username dataset and an unrestricted version using integers within the range $[0, 10^{100}]$.[¶] We report the top 1% of outcomes with respect to the maximum hops.

Results in Figure 4b confirm that adversarial attacks significantly degrade standard implementations, while robust counterparts maintain consistent performance. The robust skip list maintained an average maximum hop count of 27.36, compared to 33.17 for the non-robust implementation under non-adaptive conditions. Under adaptive settings, the non-robust implementation degraded to 35.61 maximum average hops, and further to 202.71 hops when using the larger integer range. This validation confirms our theoretical findings on adversarial robustness, and also suggests that our remark regarding the artificial "looseness" of the bound carries weight.

10 CONCLUSION AND FUTURE WORK

In this work, we conducted the first systematic analysis of probabilistic skipping-based data structures – specifically, hash tables, skip lists, and treaps – in adaptive adversarial settings. Further, we established formal security notions and provided provably secure variants of each structure. Moreover, we uncovered innate vulnerabilities in the standard constructions of hash tables and skip lists that allowed adversaries to carry out attacks that degraded the expected performance of these structures exponentially. Remarkably, we found that (insertion-only) treaps demonstrated inherent resilience against adaptive adversarial manipulation.

While our theoretical bounds provide rigorous security guarantees, there remains scope for developing tighter bounds. Further future research directions include extending our analysis to related data structures such as zip trees [51], zip-zip trees [29], skip graphs [7], and randomized meldable heaps [28]. Moreover, a more sophisticated approach to deletions could potentially eliminate memory overhead while maintaining security guarantees. One promising direction involves localized re-initializations that would allow portions of the data structure to be safely rebuilt without compromising robustness. Additionally, it may be of interest to apply our AAPC framework to more classes of data structures and properties, as it is inherently quite extensible.

REFERENCES

[1] Georgii M Adel'son-Vel'skii. 1962. An algorithm for the organization of information. *Soviet Math.* 3 (1962), 1259–1263.
[2] Susanne Albers and Jeffery Westbrook. 2005. Self-organizing data structures. *Online Algorithms: The state of the art* (2005), 13–51.
[3] A. Ambainis. 2004. Quantum walk algorithm for element distinctness. In *45th Annual IEEE Symposium on Foundations of Computer Science*. 22–31. https://doi.org/10.1109/FOCS.2004.54
[4] Apache. 2023. https://vovkos.github.io/doxyrest/samples/apr-sphinxdoc/group_apr_skiplist.html.
[5] Austin Appleby. 2016. SMHasher. https://github.com/aappleby/smhasher.

[¶]While this serves primarily as a proof of concept, such a vast interval could realistically be achieved using a 20-character limit with Unicode encoding. We emphasize that significant runtime degradation can be observed even with substantially smaller intervals.

[6] Cecilia R Aragon and Raimund Seidel. 1989. Randomized search trees. In *FOCS*, Vol. 30. 540–545.

[7] James Aspnes and Gauri Shah. 2007. Skip graphs. *Acm transactions on algorithms (talg)* 3, 4 (2007), 37–es.

[8] Jean-Philippe Aumasson and Daniel J Bernstein. 2012. SipHash: a fast short-input PRF. In *International Conference on Cryptology in India*. Springer, 489–508.

[9] Jean-Philippe Aumasson, Martin Boßlet, and Daniel J Bernstein. 2011. Hash-flooding DoS reloaded:attacks and defenses. Slides presented at the 2011 Application Security Forum – Western Switzerland. https://web.archive.org/web/20130913185247/https://131002.net/siphash/siphashdos_appsec12_slides.pdf

[10] Noa Bar-Yosef and Avishai Wool. 2007. Remote algorithmic complexity attacks against randomized hash tables. In *International Conference on E-Business and Telecommunications*. Springer, 162–174.

[11] Rudolf Bayer. 1972. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta informatica* 1, 4 (1972), 290–306.

[12] Jim Blandy, Jason Orendorff, and Leonora FS Tindall. 2021. *Programming Rust*. " O'Reilly Media, Inc.".

[13] Paul Bottinelli. 2025. Technical Advisory – Hash Denial-of-Service Attack in Multiple QUIC Implementations. https://www.nccgroup.com/us/research-blog/technical-advisory-hash-denial-of-service-attack-in-multiple-quic-implementations/. NCC Group Research Blog.

[14] Stéphane Boucheron, Gábor Lugosi, and Pascal Massart. 2013. *Concentration Inequalities: A Nonasymptotic Theory of Independence*. Oxford University Press. https://doi.org/10.1093/acprof:oso/9780199535255.001.0001

[15] Mark Burnett. 2015. https://medium.com/xato-security/today-i-am-releasing-ten-million-passwords-b6278bbe7495.

[16] J Lawrence Carter and Mark N Wegman. 1977. Universal classes of hash functions. In *Proceedings of the ninth annual ACM symposium on Theory of computing*. 106–112.

[17] David Clayton, Christopher Patton, and Thomas Shrimpton. 2019. Probabilistic data structures in adversarial environments. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1317–1334.

[18] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (third edition ed.). MIT Press, Cambridge, MA. 228–355 pages.

[19] Scott A Crosby and Dan S Wallach. 2003. Denial of service via algorithmic complexity attacks. In *12th USENIX Security Symposium (USENIX Security 03)*.

[20] Artur Czumaj. 2004. CIS 786: Approximation and Randomized Algorithms, Lecture 4: Balls and bins games. https://www.ic.unicamp.br/~celio/peer2peer/math/czumaj-balls-into-bins.pdf

[21] Erik D Demaine, John Iacono, and Stefan Langerman. 2007. Retroactive data structures. *ACM Transactions on Algorithms (TALG)* 3, 2 (2007), 13–es.

[22] David Eckhoff, Tobias Limmer, and Falko Dressler. 2009. Hash tables for efficient flow monitoring: Vulnerabilities and countermeasures. In *2009 IEEE 34th Conference on Local Computer Networks*. IEEE, 1087–1094.

[23] Mia Filić, Jonas Hofmann, Sam A Markelon, Kenneth G Paterson, and Anupama Unnikrishnan. 2024. Probabilistic Data Structures in the Wild: A Security Analysis of Redis. *Cryptology ePrint Archive* (2024).

[24] Mia Filić, Keran Kocher, Ella Kummer, and Anupama Unnikrishnan. 2025. Deletions and Dishonesty: Probabilistic Data Structures in Adversarial Settings. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 137–168.

[25] Mia Filic, Kenneth G Paterson, Anupama Unnikrishnan, and Fernando Virdia. 2022. Adversarial correctness and privacy for probabilistic data structures. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1037–1050.

[26] Marc Fischlin, Moritz Huppert, and Sam A. Markelon. 2025. Probabilistic Skipping-Based Data Structures with Robust Efficiency Guarantees. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25)*. ACM, Taipei, Taiwan, 15 pages. https://doi.org/10.1145/3719027.3765149

[27] David A Freedman. 1975. On tail probabilities for martingales. *the Annals of Probability* (1975), 100–118.

[28] Anna Gambin and Adam Malinowski. 1998. Randomized meldable priority queues. In *International Conference on Current Trends in Theory and Practice of Computer Science*. Springer, 344–349.

[29] Ofek Gila, Michael T Goodrich, and Robert E Tarjan. 2023. Zip-Zip Trees: Making Zip Trees More Balanced, Biased, Compact, or Persistent. In *Algorithms and Data Structures Symposium*. Springer, 474–492.

[30] Zsolt István, Gustavo Alonso, Michaela Blott, and Kees Vissers. 2015. A hash table for line-rate data processing. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 8, 2 (2015), 1–15.

[31] Yanning Ji, Elena Dubrova, and Ruize Wang. 2025. Is Your Bluetooth Chip Leaking Secrets via RF Signals? *Cryptology ePrint Archive* (2025).

[32] Philipp Kisters, Heiko Bornholdt, and Janick Edinger. 2023. SkABNet: A Data Structure for Efficient Discovery of Streaming Data for IoT. In *2023 32nd International Conference on Computer Communications and Networks (ICCCN)*. 1–10. https://doi.org/10.1109/ICCCN58024.2023.10230169

[33] Alexander Klink and Julian Walde. 2011. Efficient denial of service attacks on web application platforms. In *28th Chaos Communication Congress*.

[34] Donald Ervin Knuth. 1971. *The art of computer programming. 2. Seminumerical algorithms*. Addison-Wesley.

[35] Sam A Markelon, Mia Filic, and Thomas Shrimpton. 2023. Compact Frequency Estimators in Adversarial Environments. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 3254–3268.

[36] Michael T McClellan and Jack Minker. 1974. The art of computer programming, vol. 3: sorting and searching.

[37] Kurt Mehlhorn and Peter Sanders. 2008. Hash tables and associative arrays. *Algorithms and Data Structures: The Basic Toolbox* (2008), 81–98.

[38] J Ian Munro, Thomas Papadakis, and Robert Sedgewick. 1992. Deterministic skip lists. In *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*. 367–375.

[39] Moni Naor and Eylon Yogev. 2015. Bloom Filters in Adversarial Environments. In *Advances in Cryptology – CRYPTO 2015, Part II (Lecture Notes in Computer Science, Vol. 9216)*, Rosario Gennaro and Matthew J. B. Robshaw (Eds.). 565–584. https://doi.org/10.1007/978-3-662-48000-7_28

[40] Savannah Norem. 2022. https://redis.io/blog/streaming-analytics-with-probabilistic-data-structures/.

[41] Matt Nowack. 2019. https://discord.com/blog/using-rust-to-scale-elixir-for-11-million-concurrent-users.

[42] Eyal Nussbaum and Michael Segal. 2019. Skiplist Timing Attack Vulnerability. In *International Workshop on Data Privacy Management*. Springer, 49–58.

[43] Kenneth G Paterson and Mathilde Raynal. 2022. Hyperloglog: Exponentially bad in adversarial settings. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 154–170.

[44] Vern Paxson. 1999. Bro: a system for detecting network intruders in real-time. *Computer networks* 31, 23-24 (1999), 2435–2463.

[45] Adam Prout. 2019. https://www.singlestore.com/blog/what-is-skiplist-why-skiplist-index-for-memsql/.

[46] William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (jun 1990), 668–676. https://doi.org/10.1145/78973.78977

[47] Nick Reingold, Jeffery Westbrook, and Daniel D Sleator. 1994. Randomized competitive algorithms for the list update problem. *Algorithmica* 11, 1 (1994), 15–32.

[48] John M Schanck. 2025. Clubcards for the WebPKI: smaller certificate revocation tests in theory and practice. *Cryptology ePrint Archive* (2025).

[49] Raimund Seidel and Cecilia R Aragon. 1996. Randomized search trees. *Algorithmica* 16, 4 (1996), 464–497.

[50] Daniel Dominic Sleator and Robert Endre Tarjan. 1985. Self-adjusting binary search trees. *Journal of the ACM (JACM)* 32, 3 (1985), 652–686.

[51] Robert E Tarjan, Caleb Levy, and Stephen Timmel. 2021. Zip trees. *ACM Transactions on Algorithms (TALG)* 17, 4 (2021), 1–12.

[52] Jean Ville. 1939. *Etude critique de la notion de collectif*. Vol. 3. Gauthier-Villars Paris.

[53] Jianing Zhao, Yubiao Pan, Huizhen Zhang, Mingwei Lin, Xin Luo, and Zeshui Xu. 2023. InPlaceKV: in-place update scheme for SSD-based KV storage systems under update-intensive Worklaods. *Cluster Computing* (05 2023). https://doi.org/10.1007/s10586-023-04031-9

[54] Justin Zobel, Steffen Heinz, and Hugh E Williams. 2001. In-memory hash tables for accumulating text vocabularies. *Inform. Process. Lett.* 80, 6 (2001), 271–277.

A COMMENTARY ON [42]

Nussbaum and Segal [42] show that keeping the internal structure of the skip list private is insufficient to protect against complexity attacks. We discuss their attack in more detail because it is instructive in light of how to model attacks and prove the properties of robust alternatives. Nussbaum and Segal present a timing attack that allows an adversary to discover the levels at which specific elements reside through a series of queries and, in turn, correlate the time it takes to answer a query on a given element with the height of that element. After the heights of the elements are discovered, the simple deletion attack can be mounted.

The specific attack they present includes several assumptions.

- The size of the collection represented by the structure, n, is known to the adversary, $\mathcal{A}$.
- Each node in the structure holds a unique value.
- The well-ordered universe $\mathcal{U}$ is known and is of size $O(n)$.

- The runtime of the search algorithm in the structure is consistent. That is, a search for the same value will yield the same runtime each time the search is executed.

Further, their adversarial model is the following.

- $\mathcal{A}$ is given a skip list containing some collection of data, D, that was selected by some (non-adversarial) process.
- The adversary, $\mathcal{A}$ does not have access to the internal structure of the skip list at any point. $\mathcal{A}$ can only interact with the structure through oracles that provide search, insertion, and deletion functionality to the structure that is under attack.
- After the completion of the attack, $\mathcal{A}$ is required to have altered the skip list it interacts with such that it contains the original D represented by the structure (before any adversarial interaction occurs), and the level that all (or nearly all) the elements reside at is the first.

The attack in this setting works by first running the timing attack to discover the level at which the elements in the structure exist (and, on the first iteration, which elements from $\mathcal{U}$ are present in the structure). Then all elements with a level greater than zero (exist at a higher level than the initial level) are removed. This set of removed elements is reinserted. These steps are repeated until (nearly) all the elements in the structure reside at level zero, and the original collection represented by the structure is conserved – thereby degrading the representation of this collection to (nearly) a flat singly-linked list.

As a countermeasure, the splay skip list structure is presented [42]. The approach is to swap the levels of certain elements during a search query, thereby preventing the adversary from discovering information about the level where any particular element resides (as they are not fixed). The structure is believed to prevent the timing attack from being effective, but no formal analysis of the security of the structure is given.

We again note that the adversarial setting that is given in [42] is rather limited. It assumes the adversary does not have access to the internal structure of the skip list, nor the ability to control the initial collection of data the skip represents. Further, it requires the adversary to conserve the initial data collection D that the skip list represents before any adversarial interaction occurs. We present a much stronger adversarial model in our work and a construction that satisfies this definition in Section 7.

The authors propose a new structure that is believed to prevent the timing attack they present; however, as previously stated, no formal security analysis is given. Indeed, the splay skip list is still vulnerable to attacks, as demonstrated by the following scenario. Consider a collection D of elements represented by a splay skip list, where a total order is defined on the universe in which D resides. Suppose there exists an element d such that $x_1 \leq d \leq x_2$ for every pair of elements $x_1, x_2 \in D$, where $x_1 \neq x_2$. For a specific order, $x_1 \leq d_1 \leq x_2 \leq d_2 \leq \ldots$ for $x_i \in D$ and $d_i \notin D$, an adversary can exploit this by conducting search queries for the intermediary elements d_i.

Unlike searches for elements $x_i \in D$, which would trigger the splay mechanism, searches for these intermediary elements $d_i \notin D$ bypass the splay security mechanism. The runtimes required to (not) find these intermediate nodes, however, still uniquely determine the height of elements contained in D.[‖] After the discovery of the heights of the elements contained in D, the trivial deletion attack could be carried out as before.

B FULL PROOFS

B.1 Robust Hash Table

Theorem B.1 (Robust Hash Table AAPC Security Result). *Let Π be our robust hash table from Figure 7, using PRF F to map elements to buckets. For integers $q_U, q_Q, q_H, n, t \geq 0$ such that $q_U \geq n \leq \frac{2}{e} b \ln b$ (where b is the number of buckets in the hash table Π), it holds that Π is $(\phi, \beta, \epsilon, \delta, n, t)$-conserved with ϕ being the HT Maximum Bucket Population function (Figure 5a), $\beta = \frac{4 \ln b}{\ln(\frac{2b}{se} \ln b)}$, $\epsilon = 1$, and $\delta = (\frac{1}{b} + \mathbf{Adv}^{\text{prf}}_F(O(t), n + q_Q))$, with $s \leq n$ denoting the number of elements contained in the data structure.*

Proof. We first observe a key property of our construction: the modified insertion and deletion procedures ensure elements are never truly removed from the hash table, only marked as deleted. By Lemma B.9, there exists an insert-only adversary that produces an equivalent decomposition (and runtime behavior). Therefore, it suffices to upper bound the maximum runtime of any insert-only adversary who can maximally make up to $s \leq n$ fresh insertions.

We start with a game $\mathbf{G}_0$ that is that the AAPC security game instantiated with our robust hash table Π using PRF F, property function ϕ as the HT Maximum Bucket Population function (Figure 5a), and target bound $\beta = \frac{4 \ln b}{\ln(\frac{2b}{se} \ln b)}$. In this game, the number of times F is evaluated on distinct inputs bounded by the adversary's resource budget. Calls to **Up** (also implicitly used by **Rep**) call F once. Calls to **Qry** also call F once. Thus, when executed with $\mathcal{A}$, game $\mathbf{G}_0$ makes at most $Q = n + q_Q$ queries to F.

Let $\mathbf{G}_1$ be identical to $\mathbf{G}_0$ except we use truly random sampling (modeled in the ROM) in place of the PRF. If $\mathcal{A}$ cannot distinguish F from a random function. Then, these games are indistinguishable from the adversary's perspective. We build a $O(t)$-time PRF distinguishing adversary $\mathcal{B}$ making at most Q queries to its oracle such that

$$\mathbf{Adv}^{\text{prf}}_F(\mathcal{B}) = \Pr[\mathbf{G}_0(\mathcal{A}) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}) = 1]. \tag{2}$$

Adversary $\mathcal{B}^F$ works by executing $\mathcal{A}$ in $\mathbf{G}_1$. Whenever $\mathbf{G}_1$ calls F, adversary $\mathcal{B}$ computes the response using its own oracle. When $\mathcal{A}$ halts, if the winning condition of $\mathbf{G}_1$ is satisfied, then $\mathcal{B}$ outputs 1; otherwise it outputs 0. Conditioning on the outcome of the coin flip z in $\mathcal{B}$'s game, we have the following:

$$\begin{aligned}
\mathbf{Adv}^{\text{prf}}_F(\mathcal{B}) &= 2\Pr[\mathbf{Exp}^{\text{prf}}_F(\mathcal{B} = 1)] - 1 \\
&= 2(\frac{1}{2}\Pr[\mathbf{Exp}^{\text{prf}}_F(\mathcal{B} = 1) | z = 1] \\
&+ \frac{1}{2}\Pr[\mathbf{Exp}^{\text{prf}}_F(\mathcal{B} = 1) | z = 0]) - 1 \\
&= \Pr[\mathbf{Exp}^{\text{prf}}_F(\mathcal{B} = 1) | z = 1] + \Pr[\mathbf{Exp}^{\text{prf}}_F(\mathcal{B} = 1) | z = 0] - 1 \\
&= \Pr[\mathbf{G}_0(\mathcal{A}) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}) = 1].
\end{aligned}$$

[‖] Compared to searching for elements $x_1, x_2, \ldots$ as described in the original attack, the runtimes for searching $d_1, d_2, \ldots$ only change by a constant factor (one extra step to find that the $d_i \notin S$.

HT Maximum Search Path: $\phi(D, \text{repr})$

```
1 : e ← 0
2 : for i ← 1 to m
3 :     ℓ ← length(T[i])
4 :     if ℓ > e
5 :         e ← ℓ
6 : return e
```

(a) The HT Maximum Search Path function $\phi : \mathcal{D} \times \{0,1\}^* \to \mathbb{R}$. The function iterates through all m buckets, returning the bucket with the greatest population, which is equivalent to the longest search path in the table.

TR Maximum Search Path: $\phi(D, \text{repr})$

```
1 : return φ^rec(T.root, 0)
```

$\phi^{\text{rec}}(n, e)$

```
1 : if n = null then
2 :     return
3 : e_1 ← φ^rec(n[2], e + 1)
4 : e_2 ← φ^rec(n[3], e + 1)
5 : return max(e_1, e_2)
```

(b) The TR Maximum Search Path function $\phi : \mathcal{D} \times \{0,1\}^* \to \mathbb{R}$. The function performs an in-order traversal for all elements $d \in D$, returning the longest search path cost among them.

SL Maximum Search Path: $\phi(D, \text{repr})$

```
1 : m ← 0
2 : for d ∈ D
3 :     ℓ ← 0, c ← L.header
4 :     for i ← L.level downto 1 do
5 :         while c[i] ≠ null and c[i][0].key < d    do
6 :             c ← c[i], ℓ ← ℓ + 1
7 :     c ← c[1], ℓ ← ℓ + 1
8 :     if c ≠ null and c[0].key = d then
9 :         if ℓ > m then
10 :            m ← ℓ
11 : return m
```

(c) The SL Maximum Search Path functions $\phi : \mathcal{D} \times \{0,1\}^* \to \mathbb{R}$. The function iterates through all elements $d \in D$, returning the longest search path cost among them. Our function only computes rightward pointer traversals, as downward movements equate to a simple array lookup.

Figure 5: The Maximum Search Path functions $\phi : \mathcal{D} \times \{0,1\}^* \to \mathbb{R}$ for hash tables, treaps, and skip lists. Observe that for any element $e \in \mathbb{U}, e \notin D$, the cost of a lookup is *at most* one more than for some element $d \in D$.

Now, with $\mathbf{G}_1$, we immediately have a standard insertion-only truly random balls-and-bins problem with $\leq n = \frac{2}{e} b \ln b$ balls being randomly thrown into b bins. We can apply the standard bound and conclude $\phi(\cdot) \leq \beta(\cdot)$ (that is $\frac{\phi(\cdot)}{\beta(\cdot)} \leq \epsilon = 1$) with probability $1 - \delta$ where $\delta = (\frac{1}{b} + \mathbf{Adv}^{\text{prf}}_F(O(t), Q))$. The first term comes from the standard bound, and the second results from the hybrid we showed above. □

B.2 Robust Skip List

We will formally show that our robust skip list is secure via a number of intermediary lemmas. The first of which proves a necessary condition for degenerating a skip list (including our robust version). We specifically analyze the skip list from the point of view of being able to have infinite height (we rectify this with reality before delivering our final result). One examines the set of elements that appear in the skip list strictly below level $L(n) = \log_{1/p}(n)$, where $n = \frac{1}{p}^m$ is the capacity of the skip list, and the set of elements that appear at or above level $L(n)$.

We first relate the length of a search path (i.e., the number of nodes to be visited) to the maximal width w on each level below $L(n)$, where the maximal width describes the maximal number of level i elements between level $i+1$ elements in the skip list over all levels $i = 0, 1, \ldots, L(n) - 1$. Here, we call an element a *level i element* if the node's height is at least i. In particular, any level j element is also a level i element for $i \leq j$. We say that the element is a *max-level i element* if it is a level i element but not a level $i+1$ element.

Lemma B.2 (Necessary Condition for Degenerating a Skip List). *Consider a skip list for parameter $p \in (0,1)$ holding at most $s \leq n$ elements (possibly inserted by an adaptive adversary). If on all levels $i \in \{0, 1, \ldots, L(n) - 1\}$ the number of level i elements between any pair of level $i+1$ elements is at most w, then any search path is*

of length at most $2w \log_{1/p} s$*, or the total number of elements on or above level* $L(n)$ *exceeds* $w \log_{1/p} s$.

The lemma states that, for the adversary to create a bad skip-list representation, it may either hope that many elements are assigned a height beyond $L(n)$—which is very unlikely since the heights are determined faithfully by the data structure—or it must ensure that there is a "degenerated" sub-lists exceeding the width w on some level. The latter matches our gap attack in Section 7.1, where we followed this strategy, and the lemma states that this is indeed the only valid attack strategy.

Proof. Assume that on all levels i, there exists at most w elements between any two elements on level $i+1$, and that the total size of the skip list on or above level $L(n)$ is at most $w \log_{1/p} s$. Then the search path below level $L(n)$ is at most $w \log_{1/p} s$ because whenever we descend to a level i (and the index to be searched is thus between the indexes of both level $i+1$ elements), we make at most w steps on the level i. This bounds the total number of steps on all levels i below $L(n)$ by $w \cdot L(n) = w \cdot \log_{1/p} s$. In addition, on level $L(n)$ or above, the total number of elements is bounded by $w \log_{1/p} s$, such that even searching all these elements cannot increase the overall number of inspected elements by more than $w \log_{1/p} s$. This yields an overall length of the search part of $2w \log_{1/p} s$. □

We formulate the following game to bound the number of elements on max-level i between two level $i+1$ elements for the robust skip list. We assume that the adversary can insert as many fresh elements as they like (up to the capacity n) and that the adversary can insert into any gap arbitrarily many times. Further, observe that the adversary cannot influence the height of any particular element or alter the heights that were chosen by deletion due to our special deletion method. Then, the ability of the adversary to accrue elements that exist on level i between two level $i+1$ elements boils down to a coin-flipping game.

Given the maximum number of individual trials n and probability p, the game is as follows. For each individual trial, a coin (that is *heads* with probability p and is *tails* with probability $1-p$) is flipped until a tail appears, at which point the particular trial is concluded. The outcome of a trial is the total number of *heads* that occurred during a particular trial. For instance, the outcome *tails* maps to 0, while the outcome *heads*, *heads*, *tails* maps to 2.

The game keeps a sequence of all the outcomes. Say the sequence at a point in time $t-1$ is $o_1, o_2, \ldots, o_{t-1}$. The adversary is allowed to run the next trial t anywhere within the sequence. That is, they could dictate the outcome of trial t (the result of which they do not control, as coin flips determine this) at the beginning of the sequence (before o_1), at the end of the sequence (after o_{t-1}), or anywhere in between two adjacent o_{i-1}, o_i, $i \le t-1$. The trial is then run, the outcome recorded in the sequence, and the sequence relabeled (depending on where the adversary decided to place the outcome of the most recently run trial).

For each possible trial outcome, we have the following "halving" behavior concerning runs (consecutive subsequences) of outcome i for each $i \in \{0, 1, \ldots \log_{1/p}(n)\}$. If the adversary is trying to extend a particular run, they always insert it at the beginning or end of the run. By inspection of our robust skip list structure, this strategy is optimal, as it maximizes the probability of extending a particular run (by minimizing the probability of halving). Given this, when an adversary tries to extend a run of outcome i, three possible outcomes can occur:

(1) if the outcome of this fresh trial is i, then the run extends by length 1;
(2) if the outcome of this trial is $i+1$; the length of the run is halved (or more precisely, the updated run length is the ceiling of dividing the previous run length by 2);
(3) if the outcome of the fresh trial is any other outcome, the run length remains the same.

Observe that this is precisely equivalent to the procedure for an adaptive adversary inserting at most n fresh elements into our robust skip list with probability parameter p. We specifically consider the scenario where the adversary tries to accrue elements that exist on level 0 between two level 1 elements. This is because the probability of the accruing elements on this level is maximized. Looking ahead, we will cast this run width accruing game as a stochastic process that is supermartingale, generalize our result for level 0 to all levels $i \in \{0, 1, \ldots, \log_{1/p}(n) - 1\}$, and combine them to get a bound on the maximum search path cost over the entire robust skip list.

Lemma B.3 (Bounding Layer Sequential Elements for the Robust Skip List). *Denote* W_i *the random variable describing the maximum sequence of elements that exist on max-layer* i *between two level* $i+1$ *elements for* $i \in \{0, 1, \ldots, \log 1/p(n) - 1\}$ *for a robust skip list. For* $\epsilon > 0$ *and* $a = \frac{2(1+p)}{p}$ *let* W *be the event that there exists* $W_i > a(1+\epsilon)$ *for some* $i \in \{0, 1, \ldots, L(n)-1\}$*, then*

$$\Pr[W] \le e^{(\lambda^* a) - (\epsilon \lambda^* a)},$$

where λ^* *is the maximal solution* $\lambda > 0$ *to*

$$(1-p)e^{\lambda} + p(1-p)e^{-\lambda\left(\frac{1}{p} + \frac{a}{2}\right)} + p^2 \le 1.$$

Proof. *Defining the Probabilistic Process.* We begin by considering the adversary trying to accrue a run of outcome 0. Given the adversary can play many independent trials, indexed by integer t (and in reality bounded by n), we define a counter tracking the run length X_t, initialized to 0, that is updated as follows:

$$X_{t+1} = \begin{cases} X_t + 1 & \text{with probability } 1-p \\ \left\lceil \frac{X_t}{2} \right\rceil & \text{with probability } p(1-p) \\ X_t & \text{with probability } 1 - ((1-p) + p(1-p)) = p^2 \end{cases}$$

For $X_t = x$,

$$\begin{aligned} \mathbb{E}[X_{t+1} | X_t = x] &= (1-p)(x+1) + p(1-p)\left(\frac{x}{2} + 1\right) + p^2 x \\ &= \left((1-p) + \frac{p(1-p)}{2} + p^2\right) x + (1-p) + p(1-p) \\ &= \left(1 - \frac{p}{2} + \frac{p^2}{2}\right) x + 1 - p^2, \end{aligned}$$

where we approximate $\left\lceil \frac{x}{2} \right\rceil$ as $\frac{x}{2} + 1$.

Setting $A = \left((1-p) + \frac{p(1-p)}{2} + p^2\right)$ and $D = 1 - p^2$, we can solve for a fix point (i.e., the steady-state solution where the drift is zero) by solving $a = Aa + D = a(1-A) = D$:

$$a = \frac{D}{1-A} = \frac{1-p^2}{\frac{p(1-p)}{2}} = \frac{2(1+p)}{p}.$$

Therefore, for any, p the fixed point is $a = \frac{2(1+p)}{p}$.

Bounding the Process for Outcome 0. We next cast this process as a martingale to be able to apply a concentration bound. Specifically, for trying to accrue a run of outcome 0, we define the process

$$M_t = e^{(\lambda(X_t - a))},$$

where $\lambda > 0$ is a parameter to be selected. Our goal is to show that when X_t exceeds a certain threshold (say $x \geq a + B$ for some constant $B > 0$), the process M_t is a supermartingale. That is for all $x \geq a + B, \mathbb{E}[M_{t+1}|X_t = x] \leq M_t$.

Using the update rule for our process, we have for $X_t = x$:

$$\begin{aligned}\mathbb{E}[M_{t+1}|X_t = x] &= (1-p)e^{\lambda((x+1)-a)} + p(1-p)e^{\lambda((x/2+1)-a)} + p^2 e^{\lambda(x-a)} \\ &= e^{\lambda(x-a)}\left((1-p)e^{\lambda} + p(1-p)e^{\lambda(1-x/2)} + p^2\right).\end{aligned}$$

Observe that since the term $e^{\lambda 1 - x/2}$ decreases in x, the worst case for $x \geq a + B$ is exactly at $x = a + B$. Hence, it suffices to have

$$(1-p)e^{\lambda} + p(1-p)e^{\lambda(1-a+B/2)} + p^2 \leq 1$$

for our stochastic process to satisfy the supermartingale condition.

Further, we have $\frac{a+b}{2} = \frac{1+p}{p} + \frac{B}{2}$ and $1 - \frac{a+B}{2} = -\frac{1}{p} - \frac{B}{2}$, thus, our condition simplifies to

$$(1-p)e^{\lambda} + p(1-p)e^{-\lambda\left(\frac{1}{p} - \frac{B}{2}\right)} + p^2 \leq 1.$$

For any fixed $p \in (0,1)$ and chosen $B > 0$ (in practice we chose B to be a small constant that is $\approx a$), one can solve for that largest λ that satisfies this inequality; denote this $\lambda^* = \lambda(p, B)$.

Now, define a stopping time

$$t_0 = \min\{t \geq 0 : X_t \geq a + B\}.$$

At this stopping time, we have

$$M_{t_0} = e^{\lambda^*(X_{t_0} - a)} \leq e^{\lambda^* B},$$

as $X_{t_0} \geq a + B$. We then work with the stopped process $M_{t \wedge t_0}$ (or more precisely with the process from time t_0 onward) and apply Ville's inequality [52]. This yields for all $k \geq 0$

$$\Pr\left[\max_{0 \leq t_0 \leq n} X_t \geq a + k\right] \leq \frac{\mathbb{E}[M_{t_0}]}{e^{\lambda^* k}} \leq e^{\lambda^* B} e^{-\lambda^* k}.$$

Define $C = e^{\lambda^* B}$, we then obtain the concentration bound for outcome 0 for all $k \geq 0$:

$$\Pr[X_n \geq a + k] \leq Ce^{-\lambda^* k}.$$

Recasting in the multiplicative form (as $(1+\epsilon)a = a + \epsilon a$), for any $\epsilon > 0$ we have

$$\Pr[X_t \geq (1+\epsilon)a] \leq Ce^{-\lambda^* \epsilon a}.$$

Lifting Result to All Outcomes. Next, we use a chaining argument to lift our result to the maximum over all outcomes $j \in \{0, 1, \ldots, \log_{1/p}(n) - 1\}$.

Let $X_n^{(j)}$ denote the run length process for outcome $j \in \{0, 1, \ldots, \log_{1/p}(n) - 1\}$. Observe that the probability of outcome j is $p^j(1-p)$ and the outcome $j+1$ is $p^{j+1}(1-p)$. Thus, the ratio is

$$\frac{p^j(1-p)}{p^{j+1}(1-p)} = \frac{1}{p},$$

which is independent of j. In turn, the fixed point $a = \frac{2(1+p)}{p}$ is identical (up to a constant additive error) for every outcome j. Therefore, for each j, we have

$$\Pr\left[\max_{0 \leq t_0 \leq n} X_t^{(j)} \geq a + k\right] \leq Ce^{-\lambda^* k}.$$

A naive union bound would suggest

$$\Pr\left[\max_{j=0}^{\log_{1/p}(n)-1} \max_{0 \leq t_0 \leq n} X_t^{(j)} \geq a + k\right] \leq \log_{1/p}(n) Ce^{-\lambda^* k}.$$

However, using a standard chaining and peeling argument [14] we can show that in fact, there exist constants $C', \lambda' > 0$ (depending only on p) such that

$$\Pr\left[\max_{j=0}^{\log_{1/p}(n)-1} \max_{0 \leq t_0 \leq n} X_t^{(j)} \geq a + k\right] \leq C'e^{-\lambda' k},$$

or equivalently, for any $\epsilon > 0$,

$$\Pr\left[\max_{j=0}^{\log_{1/p}(n)-1} \max_{0 \leq t_0 \leq n} X_t^{(j)} \geq (1+\epsilon)a\right] \leq C'e^{-\lambda' \epsilon a}.$$

In practice, we simply take $C' \approx C$ and $\lambda' \approx \lambda^* = \lambda(p, B)$ (the same constants as above for outcome 0), as there is at most a negligible difference between the bound for different outcomes in $\{0, 1, \ldots, \log_{1/p}(n) - 1\}$ and the bound is maximized at outcome 0. Then, by choosing $B = a$, we obtain our result in the lemma. □

Lemma B.4 (Overall Robust Skip List Search Path Cost). *For $\epsilon > 0$ and $a = \frac{2(1+p)}{p}$, let S be the total search path cost of the robust skip list, , then*

$$\Pr[S \geq a(1+\epsilon)\log_{1/p}(s)] \leq e^{(\lambda^* a) - (\epsilon \lambda^* a)},$$

where λ^* is the maximal solution $\lambda > 0$ to

$$(1-p)e^{\lambda} + p(1-p)e^{-\lambda\left(\frac{1}{p} + \frac{a}{2}\right)} + p^2 \leq 1.$$

Proof. The lemma directly follows from Lemma B.3, and the fact that

$$\begin{aligned}S &= \sum_{j=0}^{\log_{1/p}(n)} X^{(j)} \\ &\leq \sum_{j=0}^{\log_{1/p}(n)} max_{j=0}^{\log_{1/p}(n)-1} \max_{0 \leq t_0 \leq n} X_t^{(j)}.\end{aligned}$$

□

Next, we bound the number of elements in a skip list above level $L(n) = \log_{1/p}(n)$, addressing the second point in Lemma B.2.

Lemma B.5 (Bound on the Size of the list Above Level $L(n)$). *Given a (robust) skip list with probability parameter p, let H be the number of elements that appear at heights $\geq L(n) = \log_{1/p}(n)$. That is, H counts the total occurrences of elements at or above height $L(n)$ resp. the total size of the skip list at or above height $L(n)$. Then,*

$$\mathbb{E}[H] = \frac{1}{1-p},$$

and

$$\Pr\left[H \geq w \cdot \log_{1/p}(s)\right] \leq e^{-\frac{((1-p)w\log_{1/p}(n)-1)^2}{(1-p)(2+(1-p)w\log_{1/p}(n)-1)}}.$$

Proof. Let $H_i \sim \text{Bin}(s, p^i)$ denote the random variable describing the number of elements on level i among the $s \leq n$ elements, such that $\mathbb{E}[H_i] = sp^i$. Let $H = \sum_{i \geq L(n)} H_i$ be the total number of times an element appears at some level in all levels above level $L(n) = \log_{1/p}(n)$. This expected value is maximized when $s = n$, so for the rest of the analysis we assume that $s = n$. Then,

$$\mathbb{E}[H] = \sum_{i \geq L(n)} np^i = np^{L(n)} \sum_{j \geq 0} p^j = np^{L(n)} \frac{1}{1-p}.$$

Now, observe that $p^{L(n)} = p^{\log_{1/p}(n)} = n^{\log_{1/p} p} = n^{-1}$, in turn $np^{L(n)} = n \cdot p^{L(n)} = n \cdot \frac{1}{n} = 1$. Therefore, the expected number of elements that appear at any level on or above level $L(n)$ is

$$\mathbb{E}[H] = \frac{1}{1-p}.$$

We then obtain a tail bound via the standard Chernoff bound, solving for a value δ** such that $(1+\delta)\left(\frac{1}{1-p}\right) = w\log_{1/p}(n)$. This completes the proof. The value $\delta = (1-p)w\log_{1/p}(n) - 1$ works, and we get the upper bound of $\exp\left(-\frac{\delta^2}{(1-p)(2+\delta)}\right)$. □

In our robust skip list (and in all practical skip list constructions), we define a maximum level m. This, in turn, defines the capacity of the list n by solving $m = \log_{1/p}(n) = L(n)$. So, in reality, this result actually reflects the maximum number of elements on level $L(n)$. To win in our game, the adversary must either craft a structure such that the total search path cost below level $L(n)$ exceeds $w\log_{1/p}(s)$ or the above "bad" event happens where there exists more than $w\log_{1/p}(s)$ elements on level $L(n)$. Combining these results gives us the following theorem.

Theorem B.6 (Robust Skip List AAPC Security Result). *Let Π be the robust skip list from Figure 9 with parameters $p \in [0,1]$ and $m \geq 0$. For integers $q_U, q_Q, n, t \geq 0$, such that $q_U \geq n = \frac{1}{p}^m$, it holds that Π is $(\phi, \beta, \epsilon, \delta, n, t)$-conserved with ϕ being the Maximum Search Path Cost function (Figure 5c), $\beta = c\log_{1/p}(s)$, with $s \leq n$ denoting the number of elements contained in the data structure, $\epsilon > 0$, and*

$$\delta = e^{(\lambda^* a)-(\epsilon\lambda^* a)} + e^{-\frac{((1-p)a\log_{1/p}(n)-1)^2}{(1-p)(2+(1-p)a\log_{1/p}(n)-1)}},$$

where $a = \frac{2(1+p)}{p}$, $c = a(\epsilon+1)$, and λ^* is the maximal solution $\lambda > 0$ to

$$(1-p)e^{\lambda} + p(1-p)e^{-\lambda\left(\frac{1}{p}+\frac{a}{2}\right)} + p^2 \leq 1.$$

**Here δ refers to the usual difference from the mean in the Chernoff bound, not the parameter of the AAPC security notion.

Proof. We first observe a key property of our construction: the modified insertion and deletion procedures ensure elements are never truly removed from the skip list, only marked as deleted. By Lemma B.11, there exists an insert-only adversary that produces an equivalent decomposition (and runtime behavior). Therefore, it suffices to upper bound the maximum runtime of any insert-only adversary who can maximally make up to $s \leq n$ fresh insertions.

The theorem then directly follows from Lemma B.2, Lemma B.4, and Lemma B.5. □

B.3 Robust Treap

We will formally show the security (with regard to the maximal search path) of our modified-deletion treap. However, we first formalize a view of the treap's representation via a stochastic process. We start by analyzing the representation formed by a non-adaptive adversary and the subsequent maximum search path cost.

Consider a treap containing s elements inserted by a non-adaptive adversary, i.e., selected uniformly at random from the universe of all possible elements. Consider all inserted elements in the sorted order of their key value $x_1 \leq x_2 \leq \ldots \leq x_s$. Each key x_i is assigned a random priority r^{x_i} drawn independently from the uniform distribution on $[0,1]$.

Let $S^s_{x_i}$ denote the random variable representing the search path length for a fixed element x_i. The search path to element x_i consists of all ancestors of x_i in the treap structure. From Aragon and Seidel [6], x_j is an ancestor of x_i if and only if x_j has the lowest priority among all elements between x_i and x_j (inclusive). Specifically:

- If $j > i$, then x_j is an ancestor of x_i if and only if $r^{x_j} = \min\{r^{x_i}, r^{x_{i+1}}, \ldots, r^{x_j}\}$
- If $j < i$, then x_j is an ancestor of x_i if and only if $r^{x_j} = \min\{r^{x_j}, r^{x_{j+1}}, \ldots, r^{x_i}\}$

This means that an element is an ancestor of x_i precisely when its priority is a minimum value – what we will refer to as a "record" – in one of two sequences extending from x_i. Hence, we can interpret $S^s_{x_i}$ as:

$$S^s_{x_i} = |\{\text{records in sequence } r^{x_i}, r^{x_{i-1}}, \ldots, r^{x_1}\}| + |\{\text{records in sequence } r^{x_i}, r^{x_{i+1}}, \ldots, r^{x_s}\}| - 1,$$

where the subtraction of 1 accounts for x_i being counted in both sequences.

A classical fact about random permutations is the behavior of records. For a sequence of k i.i.d. uniformly distributed random variables, the probability that the k^{th} element is a record (i.e., it is less all $k-1$ preceding values is exactly $\frac{1}{k}$). More precisely, define the following indicator variables for a given sequence:

$$I_= \begin{cases} 1, & \text{if the } j^{th} \text{ element is a record,} \\ 0, & \text{otherwise.} \end{cases}$$

Then we have $\mathbb{E}[I_j] = \frac{1}{j}$. For a sequence of length k, the total number of records is $R_k = \sum_{j=1}^k I_j$ and its expectation is $\mathbb{E}[R_k] = \sum_{j=1}^k \frac{1}{j} = H_k$, where H_k is the k-th harmonic number. In our context, when considering the "leftward" sequence of priority values L_i (of length i) and the "rightward" sequence if priority R_i (of

length $n - i + 1$) with respect to key at index i, we have $\mathbb{E}[L_i] = H_i$ and $\mathbb{E}[R_i] = H_{n-i+1}$.

Thus, for a fixed x_i,

$$\mathbb{E}[S^s_{x_i}] = \mathbb{E}[L_i + R_i - 1] = H_i + H_{s-i+1} - 1.$$

Nothing, that for any i it must be that $H_i \le H_n$ and $H_{s-i+1} \le H_s$, and the well known fact $H_s \le \ln(s) + 1$, we have

$$\begin{aligned}\mathbb{E}[S^s_{x_i}] &\le 2H_s - 1\\ &\le 2\ln(s) + 1.\end{aligned}$$

We next argue that even when an adaptive adversary determines the insertions into a treap, the probability of forming a record with any new insertion $\tilde{x}$ with respect to any element x_i (that is $\tilde{x}$ becomes an ancestor of x_i) remains exactly $\frac{1}{j}$ when there are j elements between $\tilde{x}$ and x_i (inclusive). Even though an adaptive adversary can observe all previous outcomes and choose the next element adaptively (that is, select the key value so it falls into any "gap" of existing key values, like in the case of the skip list in Section 7, the new priority is still drawn uniformly and independently from $[0, 1]$). Therefore, the joint distribution of the prior priorities is unchanged. We formalize this idea in the following lemma. We ignore the trivial base case when $\tilde{x}$ is the first element inserted into the treap.

Lemma B.7 (Invariant Record Probability under Adaptive Insertion). *Consider any previously inserted reference element x_i. Further, let $x_1, x_2, \ldots, x_i, x_{i+1}, \ldots, x_{s-1}$ be the existing keys in sorted key order with associated priorities*

$$r^{x_1}, r^{x_2}, \ldots, r^{x_i}, r^{x_{i+1}}, \ldots, r^{x_{s-1}},$$

drawn independently from the uniform distribution on $[0, 1]$.

An adaptive adversary chooses a gap (i.e., a position between any two or before or after these keys) into which to insert a new key $\tilde{x}$. WLOG assume $\tilde{x} > x_i$ (the other case is symmetric). Let the number of keys (by key ordering) between x_i and $\tilde{x}$ (inclusive) be j. Then, the new key $\tilde{x}$ receives an independent priority $r^{\tilde{x}} \sim \mathsf{U}[0, 1]$. After relabeling the keys according to their inherent order, let the ordered subsequence of priorities of keys between x_i and $\tilde{x}$ (inclusive) be

$$r_{(\tilde{1})} \le r_{(\tilde{2})} \le \cdots \le r_{(\tilde{j})}.$$

Then, even conditioned on the past σ-algebra $\mathcal{F}_{s-1}$ (which contains the ordered priority values and all adversarial decisions regarding the first $s - 1$ insertions), we have

$$\Pr\left(r^{\tilde{x}} = r_{(1)} \,\middle|\, \mathcal{F}_s\right) = \frac{1}{j}.$$

Proof. Condition on the σ-algebra $\mathcal{F}_{s-1}$; that is, assume the priorities

$$r^{x_1}, r^{x_2}, \ldots, r^{x_i}, r^{x_{i+1}}, \ldots, r^{x_{s-1}}$$

are fixed and rearranged in increasing order:

$$r_{(1)} \le r_{(2)} \le \cdots \le r_{(s-1)}.$$

An adaptive adversary may insert the new key x_j in any gap between any two keys in the current sequence (or before the smallest or after the largest). Still, such a decision affects only the position of the key in the *key order* and does not alter the statistical properties of the newly drawn priority.

The new priority $r^{\tilde{x}}$ is drawn independently from $\mathsf{U}[0, 1]$. Thus, when the new key is inserted, the complete set of s priorities is

$$\{r^{\tilde{x}}, r_{(1)}, r_{(2)}, \ldots, r_{(s-1)}\}.$$

With respect to any element x_i (for the case $\tilde{x} > x_i$)) denote

$$r_{(\tilde{1})} \le r_{(\tilde{2})} \le \cdots \le r_{(\widetilde{j-1})}$$

as the ordered priorities for the j elements between x_i and $\tilde{x}$ (inclusive).

Since the first $j-1$ values are already fixed and $r^{\tilde{x}}$ is independent and uniformly distributed over $[0, 1]$, the resulting set of j priorities is exactly equivalent to a set of j independent uniform samples upon relabeling.

In any sequence of j i.i.d. $\mathsf{U}[0, 1]$ random variables, symmetry implies that the probability that any particular one (here, the newly inserted element $\tilde{x}$) is the minimum is exactly $1/j$. Formally, we have

$$\Pr\left(r^{\tilde{x}} = \min\{r^{\tilde{x}}, r_{(1)}, r_{(2)}, \ldots, r_{(j-1)}\} \,\middle|\, \mathcal{F}_{s-1}\right) = \frac{1}{j}.$$

Thus, regardless of where the adversary chooses to insert $\tilde{x}$, the probability that $\tilde{x}$ forms a record with respect to reference element x_i (i.e., its priority is the smallest among the j keys between x_i and $\tilde{x}$ inclusive) remains equal to $1/j$, as in the non-adaptive setting. Note the case where $x_i > \tilde{x}$ is symmetric. □

Theorem B.8 (Treap AAPC Result). *Let Π be the robust treap from Figure 11. For integers $q_U, q_Q, n, t \ge 0$, such that $q_U \ge n$, it holds that Π is $(\phi, \beta, \epsilon, \delta, n, t)$-conserved with ϕ being the Maximum Search Path Cost function (Figure 5b), $\beta = 2\ln s + 1$, with $s \le n$ denoting the number of elements contained in the data structure, any $\epsilon > 0$ and*

$$\delta = se^{-\frac{\epsilon^2 H_s}{2(1+\epsilon)}},$$

where H_s is the s^{th} harmonic number.

Proof. We first observe a key property of our construction: the modified insertion and deletion procedures ensure elements are never truly removed from the treap, only marked as deleted. By Lemma B.11, there exists an insert-only adversary that produces an equivalent decomposition (and runtime behavior). Therefore, it suffices to upper bound the maximum runtime of any insert-only adversary who can maximally make up to s fresh insertions.

Casting the Insertion Process as a Doob Martingale.
Consider any key x in the treap. For a fresh key $\tilde{x}$ inserted after $s - 1 > 1$ elements have been inserted and that has j keys between x and itself (inclusive), take the indicator variable I_j as defined above. Then, conditioned on the past σ-algebra $\mathcal{F}_{s-1}$ (which contains the ordered priority values and all adversarial decisions regarding the first $s - 1$ insertions), and letting

$$m_{j-1} := \min\{r^x, \ldots, r^{x_{j-1}}\} \quad (\text{with } m_0 = 0).$$

It is easy to see that

$$\Pr(I_j = 1 | \mathcal{F}_{s-1}) = \Pr(r^{\tilde{x}} < m_{j-1} | \mathcal{F}_{j-1}) = m_{j-1}.$$

From Lemma B.7, we have that even under adaptive insertions, the unconditional expectation remains

$$\mathbb{E}[m_{j-1}] = \frac{1}{j}.$$

Then, if letting X_s denote the total number of records over all s insertions with respect to any element x, the unconditional expected number of records is

$$\mathbb{E}[X_s] = \sum_{j=1}^{s} \mathbb{E}[I_j] = \sum_{j=1}^{s} \frac{1}{j} = H_s,$$

where H_s is the s^{th} harmonic number.

From our above analysis, we have that the search path length S_x^s for any key x is bounded in terms of the number of records X_s by

$$S_s^x \leq 2X_s - 1.$$

Thus, if we can show that X_s is concentrated around H_s, we also have a bound on the search cost for any particular element. To do this, define the Doob martingale

$$M_j = \sum_{i=1}^{j} (I_i - \mathbb{E}[I_i|\mathcal{F}_{i-1}]), \quad j = 0, 1, 2, \ldots, s$$

with $M_0 = 0$. By construction, $\{M_j\}$ is a martingale relative to the filtration $\{F_j\}$.

Next, observe that the martingale difference satisfies

$$D_j = M_j - M_{j-1} = I_j - \mathbb{E}[I_i|\mathcal{F}_{i-1}].$$

Since $I_j \in \{0, 1\}$ and $\mathbb{E}[I_i|\mathcal{F}_{i-1} \in [0, 1]$, we have $|D_j| \leq 1$.

Since $I_j \in \{0, 1\}$ is a Bernoulli random variable with parameter m_{j-1}, its conditional expectation is

$$\mathbb{E}[I_j \mid \mathcal{F}_{j-1}] = m_{j-1},$$

and the conditional variance is computed as:

$$\begin{aligned}\text{Var}(I_j \mid \mathcal{F}_{j-1}) &= \mathbb{E}\Big[(I_j - m_{j-1})^2 \,\Big|\, \mathcal{F}_{j-1}\Big] \\ &= m_{j-1}(1 - m_{j-1}).\end{aligned}$$

Now, note that subtracting the constant $\mathbb{E}[I_j \mid \mathcal{F}_{j-1}]$ does not change the variance. That is,

$$\begin{aligned}\text{Var}(D_j \mid \mathcal{F}_{j-1}) &= \text{Var}(I_j - \mathbb{E}[I_j \mid \mathcal{F}_{j-1}] \mid \mathcal{F}_{j-1}) \\ &= \text{Var}(I_j \mid \mathcal{F}_{j-1}) \\ &= m_{j-1}(1 - m_{j-1}).\end{aligned}$$

Thus, computing the predictable quadratic variation, we have

$$V_s = \sum_{j=1}^{s} \text{Var}(D_j \mid \mathcal{F}_{j-1}) = \sum_{j=1}^{s} m_{j-1}(1 - m_{j-1}).$$

Since $m_{j-1}(1 - m_{j-1}) \leq m_{j-1}$ (because $1 - m_{j-1} \leq 1$ for all $m_{j-1} \in [0, 1]$), we obtain

$$V_s \leq \sum_{j=1}^{s} m_{j-1}.$$

Further, as $E[m_{j-1}] = \frac{1}{j}$, we have

$$\mathbb{E}[V_s] \leq \sum_{j=1}^{s} \frac{1}{j} = H_s.$$

Applying A Concentration Bound.

Freedman's inequality [27] states that if $\{M_j\}$ is a martingale with a difference bounded by 1 with predictable quadratic variation V_s, then for any $a, b > 0$,

$$\Pr(M_s \geq a \text{ and } V_s \leq b) \leq e^{-\frac{a^2}{2(a+b)}}.$$

We set $b = H_s$ (as typically V_s will not exceed H_s by much) and chose $a = \epsilon H_s$, where $\epsilon > 0$ is our parameter from our security statement.

Then, Freedman's inequality gives us

$$\Pr(M_s \geq \epsilon H_s) \leq e^{-\frac{\epsilon^2 H_s^2}{2(\epsilon H_s + H_s)}} = e^{-\frac{\epsilon^2 H_s}{2(1+\epsilon)}}.$$

Since

$$X_s = \sum_{j=1}^{s} I_j = M_s + \sum_{j=1}^{s} \mathbb{E}[I_j|\mathcal{F}_{j-1}],$$

and $\sum_{j=1}^{s} \mathbb{E}[I_j|\mathcal{F}_{j-1}]$ has expectation H_s, the above inequality shows that with probability at least $1 - e^{-\frac{\epsilon^2 H_s}{2(1+\epsilon)}}$ we have $X_s \leq (1 + \epsilon)H_s$. Further, recalling that $S_x^s \leq 2X_s - 1$, this implies with the same probability, $S_x^s \leq 2(1 + \epsilon)H_s - 1$.

Bounding the Search Cost Path Over All Elements.

Let E_x be the event that the search path cost for a fixed element x exceeds the threshold $T = 2(1 + \epsilon)\ln(s) + 1$.

From the above, we have that

$$\Pr(E_x) \leq e^{-\frac{\epsilon^2 H_s}{2(1+\epsilon)}},$$

as $H_s \leq ln(s) + 1$.

Then, applying a standard union bound over all the s elements in the treap, the event that there exists some element with a search path cost exceeding T is bounded by

$$\Pr\left(\bigcup_{x \in \{x_1, \ldots, x_s\}} E_x\right) \leq se^{-\frac{\epsilon^2 H_s}{2(1+\epsilon)}}.$$

□

B.4 Lazy Deletion

Lemma B.9 (Hash Table Deletion Invariance). *Let* Π *be the hash table from Figure 6. Then* Π *is* deletion invariant *under lazy deletion.*

Proof. Let S denote a sequence of update queries. Without loss of generality, assume S contains only successful operations (unsuccessful deletions can be removed without affecting the analysis).

Setup and Eager Sampling. Using an identical random tape, we employ eager sampling to generate $B_{x_i} \sim \mathsf{U}(\{1, 2, \ldots, b\})$ for each insertion ins_{x_i} appearing in S. Our goal is to construct an insert-only sequence $\widetilde{S}$ that produces the same random variable decomposition as S.

Replacement Chains. Under lazy deletion, when an element is deleted, it remains in the data structure but is marked as deleted. A subsequent insertion to the same bucket position overwrites the deleted element. This motivates the following definition:

Definition B.10 (Replacement Chain). A *replacement chain* is a maximal sequence of operations of the form

$$(\mathsf{ins}_{x_1}, \mathsf{del}_{x_1}, \mathsf{ins}_{x_2}, \mathsf{del}_{x_2}, \ldots, \mathsf{ins}_{x_k}, \mathsf{del}_{x_k})$$

or

$$(\mathsf{ins}_{x_1}, \mathsf{del}_{x_1}, \mathsf{ins}_{x_2}, \mathsf{del}_{x_2}, \ldots, \mathsf{ins}_{x_k})$$

where $B_{x_i} = B_{x_j}$ for all i, j, and the operations appear in this order in S.

Each replacement chain represents a sequence of operations where elements successively replace each other in the same bucket position. The final state of each chain contributes exactly one element to the bucket (also if the chain ends with a deletion, as lazy deletion solely marks the element as deleted).

Chain Extraction Algorithm. We construct $\widetilde{S}$ by extracting the first insertion from each replacement chain:

(1) Initialize $\widetilde{S} = \emptyset$ and mark all operations in S as unprocessed.
(2) For each unprocessed insertion ins_x in S (in order):
   (a) Add ins_x to $\widetilde{S}$ and mark it as processed.
   (b) Set $c \leftarrow x$.
   (c) **Chain extension:**
      (i) Search for the next unprocessed deletion del_c after the current position. If not found, break the chain extension.
      (ii) Mark del_c as processed.
      (iii) Search for the next unprocessed insertion ins_y (after del_c) such that $B_y = B_c$. If not found, break the chain extension.
      (iv) Mark ins_y as processed and set $c \leftarrow y$.

Algorithm Correctness. The algorithm partitions S into disjoint replacement chains. Each chain contributes exactly its first insertion to $\widetilde{S}$. The algorithm runs in $O(|S|^2)$ time.

Identical Random Variable Decompositions. We now prove that S and $\widetilde{S}$ yield identical random variable decompositions under the same eager sampling.

Consider any replacement chain $C = (\mathsf{ins}_{x_1}, \mathsf{del}_{x_1}, \mathsf{ins}_{x_2}, \mathsf{del}_{x_2}, \ldots)$ identified by the algorithm, where all elements have the same bucket assignment $B_{x_1} = B_{x_2} = \cdots = \beta$.

Execution of S: Under lazy deletion, the execution proceeds as follows:

- ins_{x_1}: Adds random variable $B_{x_1} = \beta$ to the multiset decomposition.
- del_{x_1}: Marks x_1 as deleted but preserves the random variable β in the decomposition.
- ins_{x_2}: Overwrites the deleted x_1 with x_2, replacing random variable B_{x_1} with $B_{x_2} = \beta$ in the decomposition.
- This pattern continues for the entire chain.

Throughout this process, exactly one instance of β remains in the decomposition for that specific replacement chain. The identity of the element changes, but the random variable contribution is constant.

Execution of $\widetilde{S}$: The sequence $\widetilde{S}$ contains only ins_{x_1} from chain C, contributing exactly one copy of random variable $B_{x_1} = \beta$ to the decomposition.

Since every operation in S belongs to exactly one replacement chain, and each chain contributes the same random variable to both decompositions, we conclude that S and $\widetilde{S}$ produce identical multisets of random variables.

Therefore, Π is deletion-invariant under lazy deletion. □

Lemma B.11 (Skip List/Treap Deletion Invariance). *Let Π be the skip list or treap from Figure 8 and Figure 10, respectively. Then Π is* deletion-invariant *under lazy deletion.*

Proof. Let S denote a sequence of update queries. Without loss of generality, assume S contains only successful operations. We fix the random tape (sequential coin flips for skip lists, priority sampling for treaps) to compare both sequences under identical randomness.

Setup and Identical Randomness. We fix the random tape (sequential coin flips for skip lists, priority sampling for treaps) to compare both sequences under identical randomness. Our goal is to construct an insert-only sequence $\widetilde{S}$ that produces the same random variable decomposition as S.

Replacement Chains. For skip lists and treaps, replacement chains are defined by structural position rather than hash bucket assignment:

Definition B.12 (Structural Replacement Chain). A *structural replacement chain* is a maximal sequence of operations of the form

$$(\mathsf{ins}_{x_1}, \mathsf{del}_{x_1}, \mathsf{ins}_{x_2}, \mathsf{del}_{x_2}, \ldots, \mathsf{ins}_{x_k}, \mathsf{del}_{x_k})$$

or

$$(\mathsf{ins}_{x_1}, \mathsf{del}_{x_1}, \mathsf{ins}_{x_2}, \mathsf{del}_{x_2}, \ldots, \mathsf{ins}_{x_k})$$

where each $\mathsf{ins}_{x_{i+1}}$ (after del_{x_i}) satisfies the *neighbor replacement condition*: x_{i+1} becomes an immediate neighbor of the (deleted) position of x_i in the sorted order.

Specifically, $\mathsf{ins}_{x_{i+1}}$ can extend the chain if and only if x_{i+1} takes the exact same structural position that x_i occupied at the time of deletion.

Chain Extraction Algorithm. We construct $\widetilde{S}$ using the same chain extraction algorithm as in the hash table case, with the neighbor replacement condition substituted for the bucket assignment condition:

(1) Initialize $\widetilde{S} = \emptyset$ and mark all operations in S as unprocessed.
(2) For each unprocessed insertion ins_x in S (in order):
   (a) Add ins_x to $\widetilde{S}$ and mark it as processed.
   (b) Set $c \leftarrow x$.
   (c) **Chain extension:**
      (i) Search for the next unprocessed deletion del_c after the current position. If not found, break.
      (ii) Mark del_c as processed and determine the immediate neighbors n_1, n_2, such that $\nexists n_3 : n_1 \leq n_3 \leq c \vee c \leq n_3 \leq n_2$.
      (iii) Search for the next unprocessed insertion ins_y (after del_c) such that $n_1 \leq y \leq c$ or $c \leq y \leq n_2$.
      (iv) Mark ins_y as processed and set $c \leftarrow y$.

The algorithm runs in $O(|S|^2)$ time and partitions S into disjoint structural replacement chains.

Identical Random Variable Decompositions. We prove that S and $\widetilde{S}$ yield identical random variable decompositions under the same random tape.

Consider any replacement chain $C = (\mathsf{ins}_{x_1}, \mathsf{del}_{x_1}, \mathsf{ins}_{x_2}, \mathsf{del}_{x_2}, \ldots)$ identified by the algorithm.

Execution of S: Under lazy deletion, the execution proceeds as follows:

- ins_{x_1}: Samples random variable X_{x_1} and adds it to the decomposition at the position corresponding to the rank of x_1 among all contained elements.
- del_{x_1}: Marks x_1 as deleted but preserves it in the data structure. The random variable X_{x_1} remains in the same position in the decomposition.
- ins_{x_2}: Since x_2 takes the exact same structural position as the deleted x_1 (having identical immediate neighbors), it structurally replaces x_1 without new random sampling. The random variable is relabeled from X_{x_1} to X_{x_2} in the decomposition.
- Since x_2 has the same rank among contained elements as x_1 had at the time of deletion, the random variable remains at the same position in the decomposition.

This pattern continues throughout the chain. At each step, exactly one random variable occupies the structural position, regardless of which element currently holds it.

Execution of $\widetilde{S}$: The sequence $\widetilde{S}$ contains only ins_{x_1} from chain C, contributing exactly the random variable X_{x_1} to the decomposition at the same structural position.

Since the random variables are sampled in the same order for both sequences (by construction, $\widetilde{S}$ preserves the order of first insertions), and each replacement chain contributes the same random variable to both decompositions, we conclude that S and $\widetilde{S}$ produce identical random variable decompositions.

Therefore, Π is deletion-invariant under lazy deletion. □

C ADAPTIVE ATTACK ON TREAPS

Consider a real-time task scheduling system that uses a treap data structure to manage job priorities in a multi-user environment. Tasks are stored with their priority values as keys (lower numbers indicating higher priority), and each task receives a random treap priority for balancing upon insertion. The scheduler extracts the minimum priority task through depth-limited breadth-first search for parallel execution.

However, this implementation contains a critical security vulnerability against adaptive adversaries. While attackers cannot directly manipulate the random priority values assigned to entries, they can execute a more sophisticated attack by strategically inserting elements with carefully chosen keys positioned adjacent to a target element. By continuing this insertion pattern until placing an element with exceptionally high priority, they force the treap to perform rotation operations that position the target element high in the treap, despite a possibly low priority.

Concretely, let $x_1, x2, \ldots, xj-1$ be the tasks inserted in sorted order with associated priorities $r^{x_1}, r^{x_2}, \ldots, r^{x_{j-1}}$ drawn independently from the uniform distribution on $[0, 1]$. An adaptive adversary selects an arbitrary target element x_i. The adversary then repeatedly inserts new elements into the gaps between x_i and x_{i+1} and between x_{i-1} and x_i until obtaining exceptionally low priority values. This is expected to occur after a constant number of insertions. After inserting, let $S^n_{x_i}$ denote the search path length to x_i. Since

$$S^s_{x_i} = |\{\text{records in sequence } r^{x_i}, r^{x_{i-1}}, \ldots, r^{x_1}\}| + |\{\text{records in sequence } r^{x_i}, r^{x_{i+1}}, \ldots, r^{x_s}\}| - 1,$$

and the number of records in these intervals is constant (as the neighboring elements to x_i have exceptionally low priorities for which there exists only a constant number of nodes with even lower priorities), x_i now resides near the top of the treap with high probability.

D ADDITIONAL ANALYSIS OF THE SWAPPING MECHANISM FOR THE ROBUST SKIP LIST

We now show that the runtime of the proposed swapping mechanism is constant with overwhelming probability.

Theorem D.1 (Swapping Mechanism Overhead). *Let Π be the robust skip list from Figure 9 with parameters $p \in (0, 1)$ and $m = \log_{1/p}(q_U)$. For integers $q_U, q_Q, t \geq 0$, the overhead of the swapping mechanism from Figure 3 satisfies the following bounds:*

(1) ***Lower layers*** *($i \in \{0, 1, \ldots, \log_{1/p}(n) - 1\}$): For any $\epsilon > 0$:*

$$\Pr[S_{i+1} > (1+\epsilon)\frac{2(1+p)}{p}] \leq e^{\lambda^* a(1-\epsilon)},$$

where λ^* is the maximal solution $\lambda > 0$ to

$$(1-p)e^{\lambda} + p(1-p)e^{-\lambda\left(\frac{1}{p}+\frac{a}{2}\right)} + p^2 \leq 1.$$

(2) ***Higher layers*** *($i + 1 \geq \log_{1/p}(n)$): For any $\delta \geq 0$,*

$$\Pr\left[S_{i+1} \geq (1+\delta)\frac{1}{1-p}\right] \leq \exp\left(-\frac{\delta^2}{(2+\delta)(1-p)}\right).$$

Here, S_{i+1} denotes the maximum overhead of the swapping mechanism when inserting an element on layer $i + 1$ (hence swapping on layer i), and n represents the total number of elements in the data structure.

Proof. Our swapping mechanism identifies the middle element at layer i that lies between two consecutive layer $i+1$ elements, then performs a constant-time height swap. Consequently, the worst-case runtime is determined by the maximum number of layer i elements that can exist between any two consecutive layer $i + 1$ elements. Fortunately, we have already established bounds for this quantity. Let S_{i+1} denote the maximum overhead of the swapping mechanism when inserting an element on layer $i + 1$.

Let W_i denote the maximum sequence of elements that exist on max-layer i between two level $i + 1$ elements. Further, let $\epsilon > 0$ and $a = \frac{2(1+p)}{p}$. Using Lemma B.3, we immediately get that for any layer $i \in \{0, 1, \ldots, \log 1/p(n) - 1\}$, it holds that

$$\Pr[W_i > a(1+\epsilon)] \leq e^{(\lambda^* a) - (\epsilon\lambda^* a)},$$

where λ^* is the maximal solution $\lambda > 0$ to

$$(1-p)e^{\lambda} + p(1-p)e^{-\lambda\left(\frac{1}{p}+\frac{a}{2}\right)} + p^2 \leq 1.$$

This immediately translates to the number of steps the swapping mechanism has to perform for layers $i \in \{0, 1, \ldots, \log 1/p(n) - 1\}$, that is,

$$\Pr[S_{i+1} > a(1+\epsilon)] \leq e^{(\lambda^* a) - (\epsilon\lambda^* a)},$$

where λ^* is the maximal solution $\lambda > 0$ to

$$(1-p)e^{\lambda} + p(1-p)e^{-\lambda\left(\frac{1}{p}+\frac{a}{2}\right)} + p^2 \leq 1.$$

We still need to bound the number of steps for insertions above layer $\log_{1/p}(n)$. Let H be the number of elements that appear at heights $\geq L(n) = \log_{1/p}(n)$. Equivalently to the proof for Lemma B.5: Let $H_i \sim \text{Bin}(s, p^i)$ denote the random variable describing the number of elements on level i among the s elements, such that $\mathbb{E}[H_i] = sp^i$. Let $H = \sum_{i \geq L(n)} H_i$ be the total number of times an element appears at some level in all levels above level $L(n) = \log_{1/p}(n)$. Then, noting that this value is maximized when $s = n$,

$$\mathbb{E}[H] = \sum_{i \geq L(n)} np^i = np^{L(n)} \sum_{j \geq 0} p^j = np^{L(n)} \frac{1}{1-p}.$$

Now, observe that $p^{L(n)} = p^{\log_{1/p}(n)} = n^{\log_{1/p} p} = n^{-1}$, in turn $np^{L(n)} = n \cdot p^{L(n)} = n \cdot \frac{1}{n} = 1$. Therefore, the expected number of elements that appear at any level on or above level $L(n)$ is

$$\mathbb{E}[H] = \frac{1}{1-p}.$$

We then obtain a tail bound via the standard Chernoff bound (for $\delta \geq 0$):

$$\Pr[X \geq (1+\delta)\mu] \leq \exp\left(-\frac{\delta^2 \mu}{2+\delta}\right).$$

We get the upper bound of

$$\Pr\left[H \geq (1+\delta)\frac{1}{1-p}\right] \leq \exp\left(-\frac{\delta^2}{(2+\delta)(1-p)}\right).$$

Once again, this immediately translates to the number of steps the swapping mechanism has to perform for layers $i + 1 \geq \log_{1/p}(n)$, that is,

$$\Pr\left[S_{i+1} \geq (1+\delta)\frac{1}{1-p}\right] \leq \exp\left(-\frac{\delta^2}{(2+\delta)(1-p)}\right).$$

□

E PSEUDOCODE FOR SKIPPING-BASED PROBABILISTIC DATA STRUCTURES

We present full pseudocode for the various original versions of the hash table, skip list, and treap structures (Figure 6, Figure 8, Figure 10) and the robust hash table, skip list, and treap versions (Figure 7, Figure 9, Figure 11).

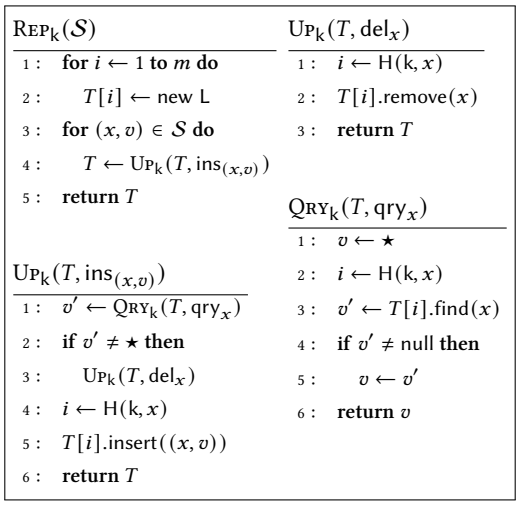


Figure 6: A possibly keyed hash-table structure $\text{HT}[\mathsf{H}_{\mathsf{k}}, b]$ admitting insertions, deletions, and queries for any $k \in \mathcal{U}_\kappa$ and its associated value v. The parameters are an integer $b \geq 1$, and a keyed function $\mathsf{H} : \mathcal{K} \times \mathcal{U}_\kappa \to [b]$ that maps the key part of key-value pair data-object elements (encoded as strings) to a position in the one of the table buckets $v.T$. A particular choice of parameters gives a concrete scheme. Each bucket contains a simple linked list L equipped with its usual operations insert, find, **and** remove **for insertion, searching, and deletion. If an item is not contained in the map, the distinguished symbol $\star$ is returned.**

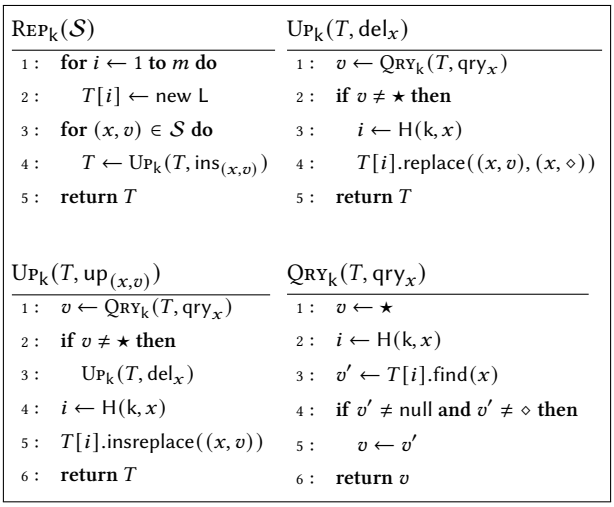
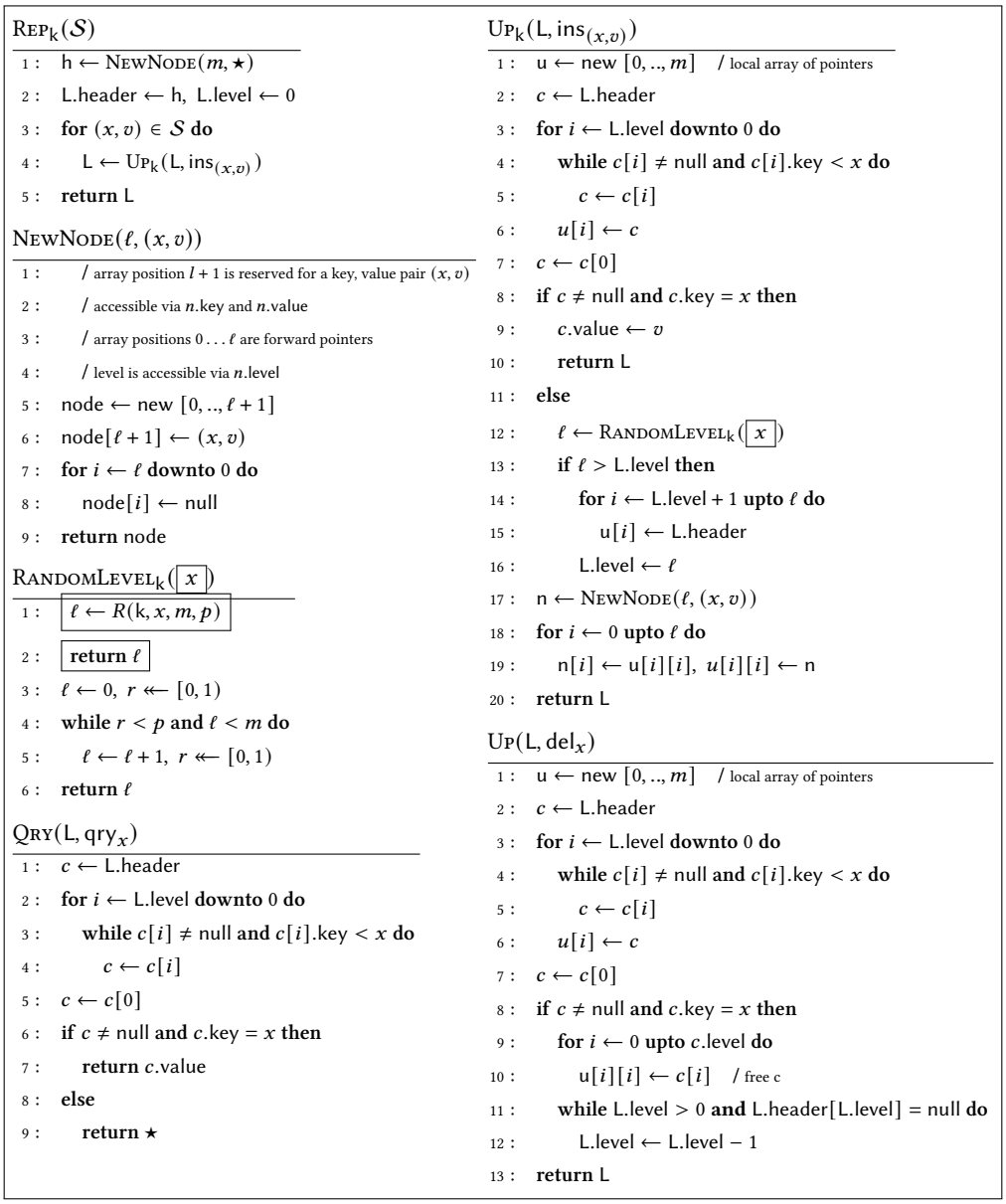
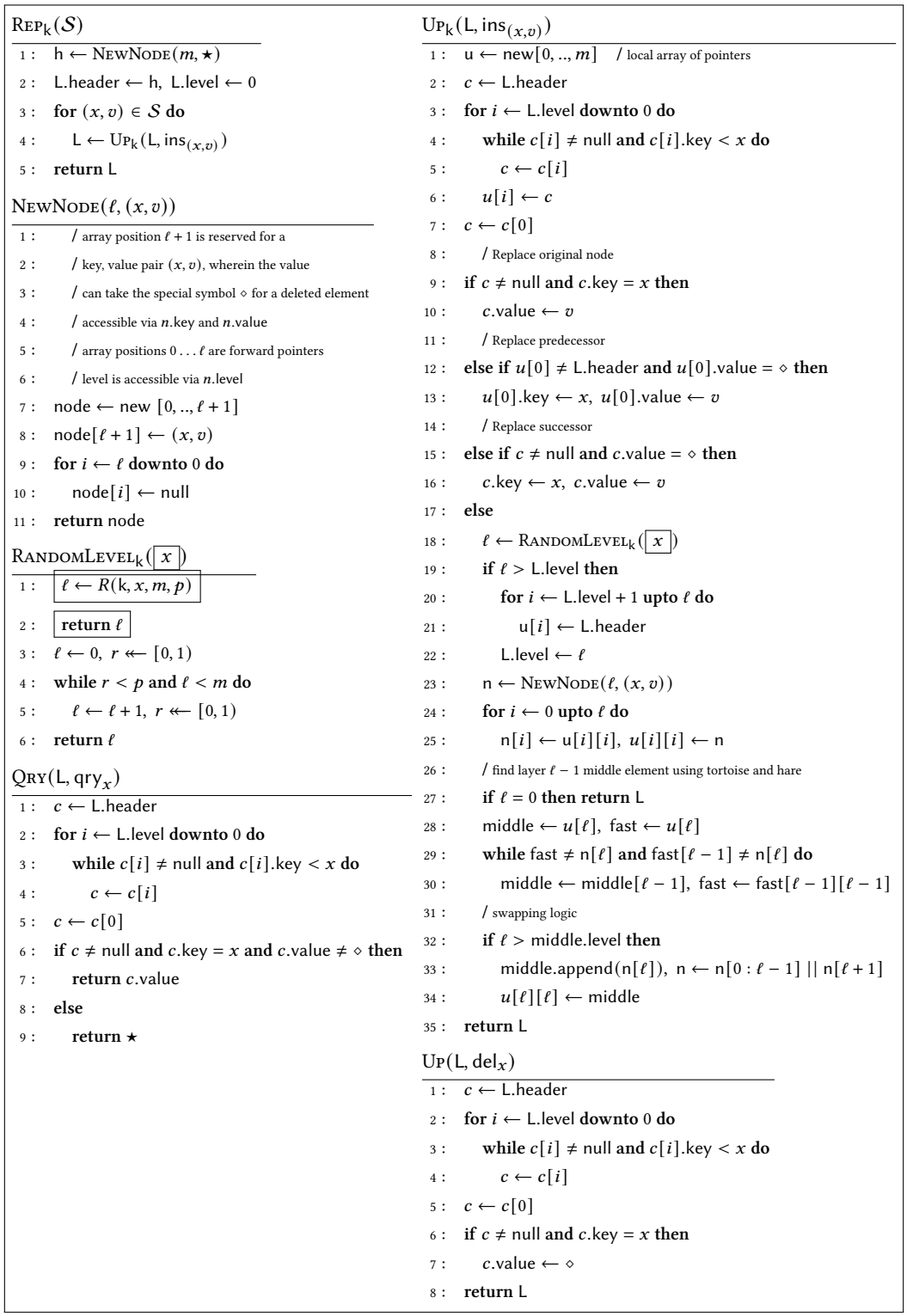


Figure 7: A robust hash table in the AAPC security model. It is an explicitly keyed hash-table structure $\text{RHT}[\mathsf{H}, b]$ admitting insertions, modified deletions, and queries for any $k \in \mathcal{U}_\kappa$ and its associated value v. The parameters are an integer $b \geq 1$, and a keyed function $\mathsf{H} : \mathcal{K} \times \mathcal{U}_\kappa \to [b]$ that maps the key part of key-value pair data-object elements (encoded as strings) to a position in one of the table buckets $v.T$. A particular choice of parameters gives a concrete scheme. Each bucket contains a simple linked list L equipped with its usual operations. We define the insreplace **operation of L, such that if it finds an item with $(k, \diamond)$ for any $k \in \mathcal{U}_\kappa$ during its internal search, the item to be inserted is written in this location; otherwise, a regular insertion occurs. If an item is not contained in the map, the distinguished value $\star$ is returned. Similarly, the distinguished symbol $\diamond$ marks an element as deleted.**



```
REP_k(S)
1 : h ← NEWNODE(m, ⋆)
2 : L.header ← h, L.level ← 0
3 : for (x, v) ∈ S do
4 :     L ← UP_k(L, ins_(x,v))
5 : return L

NEWNODE(ℓ, (x, v))
1 :     / array position l + 1 is reserved for a key, value pair (x, v)
2 :     / accessible via n.key and n.value
3 :     / array positions 0 . . . ℓ are forward pointers
4 :     / level is accessible via n.level
5 : node ← new [0, .., ℓ + 1]
6 : node[ℓ + 1] ← (x, v)
7 : for i ← ℓ downto 0 do
8 :     node[i] ← null
9 : return node

RANDOMLEVEL_k(x)
1 : ℓ ← R(k, x, m, p)
2 : return ℓ
3 : ℓ ← 0, r ↞ [0, 1)
4 : while r < p and ℓ < m do
5 :     ℓ ← ℓ + 1, r ↞ [0, 1)
6 : return ℓ

QRY(L, qry_x)
1 : c ← L.header
2 : for i ← L.level downto 0 do
3 :     while c[i] ≠ null and c[i].key < x do
4 :         c ← c[i]
5 : c ← c[0]
6 : if c ≠ null and c.key = x then
7 :     return c.value
8 : else
9 :     return ⋆

UP_k(L, ins_(x,v))
1 : u ← new [0, .., m]    / local array of pointers
2 : c ← L.header
3 : for i ← L.level downto 0 do
4 :     while c[i] ≠ null and c[i].key < x do
5 :         c ← c[i]
6 :     u[i] ← c
7 : c ← c[0]
8 : if c ≠ null and c.key = x then
9 :     c.value ← v
10 :    return L
11 : else
12 :    ℓ ← RANDOMLEVEL_k(x)
13 :    if ℓ > L.level then
14 :        for i ← L.level + 1 upto ℓ do
15 :            u[i] ← L.header
16 :        L.level ← ℓ
17 :    n ← NEWNODE(ℓ, (x, v))
18 :    for i ← 0 upto ℓ do
19 :        n[i] ← u[i][i], u[i][i] ← n
20 :    return L

UP(L, del_x)
1 : u ← new [0, .., m]    / local array of pointers
2 : c ← L.header
3 : for i ← L.level downto 0 do
4 :     while c[i] ≠ null and c[i].key < x do
5 :         c ← c[i]
6 :     u[i] ← c
7 : c ← c[0]
8 : if c ≠ null and c.key = x then
9 :     for i ← 0 upto c.level do
10 :        u[i][i] ← c[i]    / free c
11 :    while L.level > 0 and L.header[L.level] = null do
12 :        L.level ← L.level − 1
13 : return L
```

Figure 8: A possibly "deterministic" (and keyed) skip list structure $\mathrm{SL}[\boxed{R}, m, p]$ admitting insertions, deletions, and queries for any $x \in \mathcal{U}$ for some well-ordered universe $\mathcal{U}$. The parameters are an integer $m \geq 0$ representing the maximum level of the structure, a fraction $p \in (0, 1)$ used for determining an element's random level, and, if using the deterministic version of the structure, a keyed function $R : \mathcal{K} \times \mathcal{U} \times \mathbb{Z}^+ \times (0, 1) \to [m]$ that maps an element to a level in accordance with the distribution imposed by m and p. A concrete scheme is given by a particular choice of parameters. Subroutines used by the deterministic version of the structure appear in the boxed environment. If an item is not contained in the map, the distinguished symbol $\star$ is returned.



$\textsc{Rep}_{\mathsf{k}}(\mathcal{S})$

```
h ← NewNode(m, ⋆)
L.header ← h, L.level ← 0
for (x, v) ∈ S do
    L ← Up_k(L, ins_(x,v))
return L
```

$\textsc{NewNode}(\ell, (x, v))$

```
/ array position ℓ + 1 is reserved for a
/ key, value pair (x, v), wherein the value
/ can take the special symbol ⋄ for a deleted element
/ accessible via n.key and n.value
/ array positions 0 . . . ℓ are forward pointers
/ level is accessible via n.level
node ← new [0, .., ℓ + 1]
node[ℓ + 1] ← (x, v)
for i ← ℓ downto 0 do
    node[i] ← null
return node
```

$\textsc{RandomLevel}_{\mathsf{k}}(\boxed{x})$

```
ℓ ← R(k, x, m, p)
return ℓ
ℓ ← 0, r ↞ [0, 1)
while r < p and ℓ < m do
    ℓ ← ℓ + 1, r ↞ [0, 1)
return ℓ
```

$\textsc{Qry}(\mathsf{L}, \mathsf{qry}_x)$

```
c ← L.header
for i ← L.level downto 0 do
    while c[i] ≠ null and c[i].key < x do
        c ← c[i]
c ← c[0]
if c ≠ null and c.key = x and c.value ≠ ⋄ then
    return c.value
else
    return ⋆
```

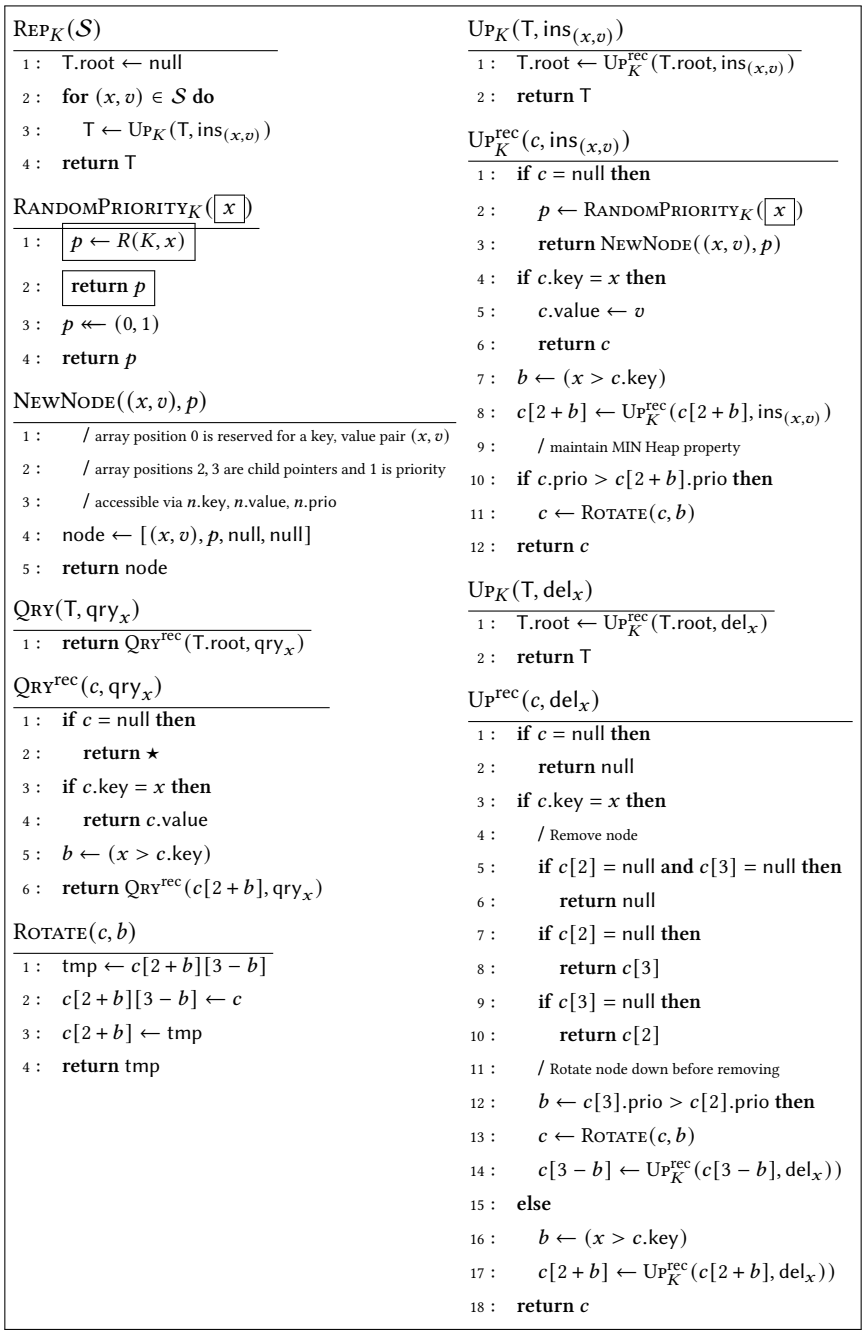
$\textsc{Up}_{\mathsf{k}}(\mathsf{L}, \mathsf{ins}_{(x,v)})$

```
u ← new[0, .., m]   / local array of pointers
c ← L.header
for i ← L.level downto 0 do
    while c[i] ≠ null and c[i].key < x do
        c ← c[i]
    u[i] ← c
c ← c[0]
    / Replace original node
if c ≠ null and c.key = x then
    c.value ← v
    / Replace predecessor
else if u[0] ≠ L.header and u[0].value = ⋄ then
    u[0].key ← x, u[0].value ← v
    / Replace successor
else if c ≠ null and c.value = ⋄ then
    c.key ← x, c.value ← v
else
    ℓ ← RandomLevel_k(x)
    if ℓ > L.level then
        for i ← L.level + 1 upto ℓ do
            u[i] ← L.header
        L.level ← ℓ
    n ← NewNode(ℓ, (x, v))
    for i ← 0 upto ℓ do
        n[i] ← u[i][i], u[i][i] ← n
    / find layer ℓ − 1 middle element using tortoise and hare
    if ℓ = 0 then return L
    middle ← u[ℓ], fast ← u[ℓ]
    while fast ≠ n[ℓ] and fast[ℓ − 1] ≠ n[ℓ] do
        middle ← middle[ℓ − 1], fast ← fast[ℓ − 1][ℓ − 1]
    / swapping logic
    if ℓ > middle.level then
        middle.append(n[ℓ]), n ← n[0 : ℓ − 1] || n[ℓ + 1]
        u[ℓ][ℓ] ← middle
return L
```

$\textsc{Up}(\mathsf{L}, \mathsf{del}_x)$

```
c ← L.header
for i ← L.level downto 0 do
    while c[i] ≠ null and c[i].key < x do
        c ← c[i]
c ← c[0]
if c ≠ null and c.key = x then
    c.value ← ⋄
return L
```

Figure 9: A robust, possibly "deterministic" (and keyed) skip list structure $\mathrm{SL}[\boxed{R}, m, p]$ admitting insertions, deletions, and queries for any $x \in \mathcal{U}$ for some well-ordered universe $\mathcal{U}$. The parameters are an integer $m \geq 0$ representing the maximum level of the structure, a fraction $p \in (0, 1)$ used for determining an element's random level, and, if using the deterministic version of the structure, a keyed function $R : \mathcal{K} \times \mathcal{U} \times \mathbb{Z}^+ \times (0, 1) \to [m]$ that maps an element to a level in accordance with the distribution imposed by m and p. A concrete scheme is given by a particular choice of parameters. Subroutines used by the deterministic version of the structure appear in the boxed environment. We define the append **operation, such that it inserts an element in the node array in between entry ℓ and $\ell + 1$. If an item is not contained in the map, the distinguished value $\star$ is returned. Similarly, the distinguished symbol $\diamond$ marks an element as deleted.**



```
REP_K(S)
1 : T.root ← null
2 : for (x, v) ∈ S do
3 :     T ← UP_K(T, ins_(x,v))
4 : return T

RANDOMPRIORITY_K([x])
1 : [p ← R(K, x)]
2 : [return p]
3 : p ↞ (0, 1)
4 : return p

NEWNODE((x, v), p)
1 :     / array position 0 is reserved for a key, value pair (x, v)
2 :     / array positions 2, 3 are child pointers and 1 is priority
3 :     / accessible via n.key, n.value, n.prio
4 : node ← [(x, v), p, null, null]
5 : return node

QRY(T, qry_x)
1 : return QRY^rec(T.root, qry_x)

QRY^rec(c, qry_x)
1 : if c = null then
2 :     return ⋆
3 : if c.key = x then
4 :     return c.value
5 : b ← (x > c.key)
6 : return QRY^rec(c[2 + b], qry_x)

ROTATE(c, b)
1 : tmp ← c[2 + b][3 − b]
2 : c[2 + b][3 − b] ← c
3 : c[2 + b] ← tmp
4 : return tmp

UP_K(T, ins_(x,v))
1 : T.root ← UP_K^rec(T.root, ins_(x,v))
2 : return T

UP_K^rec(c, ins_(x,v))
1 : if c = null then
2 :     p ← RANDOMPRIORITY_K([x])
3 :     return NEWNODE((x, v), p)
4 : if c.key = x then
5 :     c.value ← v
6 :     return c
7 : b ← (x > c.key)
8 : c[2 + b] ← UP_K^rec(c[2 + b], ins_(x,v))
9 :     / maintain MIN Heap property
10 : if c.prio > c[2 + b].prio then
11 :     c ← ROTATE(c, b)
12 : return c

UP_K(T, del_x)
1 : T.root ← UP_K^rec(T.root, del_x)
2 : return T

UP^rec(c, del_x)
1 : if c = null then
2 :     return null
3 : if c.key = x then
4 :     / Remove node
5 :     if c[2] = null and c[3] = null then
6 :         return null
7 :     if c[2] = null then
8 :         return c[3]
9 :     if c[3] = null then
10 :        return c[2]
11 :    / Rotate node down before removing
12 :    b ← c[3].prio > c[2].prio then
13 :    c ← ROTATE(c, b)
14 :    c[3 − b] ← UP_K^rec(c[3 − b], del_x))
15 : else
16 :    b ← (x > c.key)
17 :    c[2 + b] ← UP_K^rec(c[2 + b], del_x))
18 : return c
```

Figure 10: A possibly "deterministic" (and keyed) MIN treap structure $\mathrm{TR}[\boxed{R}]$ admitting insertions, deletions, and queries for any $x \in \mathcal{U}$ for some well-ordered universe $\mathcal{U}$. The parameter is a keyed function $R : \mathcal{K} \times \mathcal{U} \to (0, 1))$ that assigns an element a random priority. Subroutines used by the deterministic version of the structure appear in the boxed environment. If an item is not contained in the map, the distinguished value $\star$ is returned.

```
REP_K(S)
1 : T.root ← null
2 : for (x, v) ∈ S do
3 :     T ← UP_K(T, ins_(x,v))
4 : return T

RANDOMPRIORITY_K(x)
1 : p ← R(K, x)
2 : return p
3 : p ↞ (0, 1)
4 : return p

NEWNODE((x, v), p)
1 :     / array position 0 is reserved for a key, value pair (x, v), wherein
2 :     / the value can take the special symbol ⋄ for a deleted element
3 :     / array positions 2, 3 are child pointers and 1 is priority
4 :     / accessible via n.key, n.value, n.prio
5 : node ← [(⊥, x, v), p, null, null]
6 : return node

QRY(T, qry_x)
1 : return QRY^rec(T.root, qry_x)

QRY^rec(c, qry_x)
1 : if c = null then
2 :     return ⋆
3 : if c.key = x and c.value ≠ ⋄ then
4 :     return c.value
5 : if c.key = x and c.value = ⋄ then
6 :     return ⋆
7 : b ← (x > c.key)
8 : return QRY^rec(c[2 + b], qry_x)

ROTATE(c, b)
1 : tmp ← c[2 + b][3 − b]
2 : c[2 + b][3 − b] ← c
3 : c[2 + b] ← tmp
4 : return tmp

UP_K(T, ins_(x,v))
1 : T.root ← UP_K^rec(T.root, ins_(x,v), null, null)
2 : return T

UP_K^rec(c, ins_(x,v), pred, succ)
1 : if c = null then
2 :     / replace predecessor
3 :     if pred ≠ null and pred.value = ⋄ then
4 :         pred.key ← x, pred.value ← v
5 :         return null
6 :     / replace successor
7 :     if succ ≠ null and succ.value = ⋄ then
8 :         succ.key ← x, succ.value ← v
9 :         return null
10 :    p ← RANDOMPRIORITY_K(x)
11 :    return NEWNODE((x, v), p)
12 :    / replace original node
13 : if c.key = x then
14 :    c.value ← v
15 :    return c
16 : b ← (x > c.key)
17 : if b = 0 then succ ← c else pred ← c
18 : c[2 + b] ← UP_K^rec(c[2 + b], ins_(x,v), pred, succ)
19 :    / maintain MIN Heap property
20 : if c[2 + b] ≠ null and c.prio > c[2 + b].prio then
21 :    c ← ROTATE(c, b)
22 : return c

UP_K(T, del_x)
1 : UP_K^rec(T.root, del_x)
2 : return T

UP^rec(c, del_x)
1 : if c = null then
2 :     return
3 : if c.key = x then
4 :     / Remove node
5 :     c.value ← ⋄
6 : else
7 :     b ← (x > c.key)
8 :     UP_K^rec(c[2 + b], del_x)
9 : return
```

Figure 11: A robust, possibly "deterministic" (and keyed) robust MIN treap structure $\mathrm{TR}[\boxed{R}]$ admitting insertions, deletions, and queries for any $x \in \mathcal{U}$ for some well-ordered universe $\mathcal{U}$. The parameter is a keyed function $R : \mathcal{K} \times \mathcal{U} \to (0, 1))$ that assigns an element a random priority. Subroutines used by the deterministic version of the structure appear in the boxed environment. If an item is not contained in the map, the distinguished value ⋆ is returned. Similarly, the distinguished symbol ⋄ marks an element as deleted.