History-Independent Concurrent Hash Tables

Hagit Attiya
Technion
Haifa, Israel
hagit@cs.technion.ac.il

Michael A. Bender
Stony Brook University
Stony Brook, USA
bender@cs.stonybrook.edu

Martín Farach-Colton
New York University
Brooklyn, USA
martin@farach-colton.com

Rotem Oshman
Tel Aviv University
Tel Aviv, Israel
roshman@tau.ac.il

Noa Schiller
Tel Aviv University
Tel Aviv, Israel
noaschiller@mail.tau.ac.il

Abstract

A *history-independent* data structure does not reveal the history of operations applied to it, only its current logical state, even if its internal state is examined. This paper studies history-independent concurrent *dictionaries*, in particular, hash tables, and establishes inherent bounds on their space requirements.

This paper shows that there is a lock-free history-independent concurrent hash table, in which each memory cell stores two elements and two bits, based on Robin Hood hashing. Our implementation is linearizable, and uses the shared memory primitive LL/SC. The expected amortized step complexity of the hash table is $O(c)$, where c is an upper bound on the number of concurrent operations that access the same element, assuming the hash table is not overpopulated. We complement this positive result by showing that even if we have only two concurrent processes, no history-independent concurrent dictionary that supports sets of any size, with *wait-free* membership queries and *obstruction-free* insertions and deletions, can store only two elements of the set and a constant number of bits in each memory cell. This holds even if the step complexity of operations on the dictionary is unbounded.

CCS Concepts

• **Theory of computation** → **Concurrent algorithms**; **Data structures design and analysis**; • **Computing methodologies** → **Concurrent algorithms**.

Keywords

concurrent data structures, history independence, linear hashing, robin-hood hashing

ACM Reference Format:
Hagit Attiya, Michael A. Bender, Martín Farach-Colton, Rotem Oshman, and Noa Schiller. 2025. History-Independent Concurrent Hash Tables. In *Proceedings of the 57th Annual ACM Symposium on Theory of Computing (STOC '25), June 23–27, 2025, Prague, Czechia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3717823.3718283

This work is licensed under a Creative Commons Attribution-NoDerivatives 4.0 International License.
STOC '25, Prague, Czechia
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1510-5/25/06
https://doi.org/10.1145/3717823.3718283

1 Introduction

A data structure is said to be *history independent* (HI) [42, 46] if its representation (in memory or on disk) depends only on the current logical state of the data structure and does not reveal the history of operations that led to this state. History independence is a privacy property: for example, a history-independent file system does not reveal that a file previously existed in the system but was then deleted, or that one file was created before another. The notion of history independence was introduced by Micciancio [42], and Naor and Teague [46] then formalized two now-classical notions of history independence: a data structure is *weakly history independent* (WHI) if it leaks no information to an observer who sees the representation a single time; it is *strongly history independent* (SHI) if it leaks no information even to an observer who sees the representation at multiple points in the execution. History independence has been extensively studied in sequential data structures [1, 14, 17, 18, 24, 25, 28, 29, 42, 45, 46] (see Section 6), and the foundational algorithmic work on history independence has found its way into secure storage systems and voting machines [9–12, 16, 20, 26, 48, 53]. Surprisingly, despite the success of history independence in the sequential setting, there are very few results on history independence in *concurrent* data structures [7, 55]. In this work we initiate the study of *history-independent concurrent hash tables.* A hash table implements a *dictionary*, an abstract data type that supports insertions, deletions and lookups of items in a set. Hash tables make for an interesting test case for history-independent concurrent data structures: on the one hand, they naturally disperse contention between threads and allow for true parallelism [21, 30, 31, 43, 51, 54]. On the other hand, many techniques that are useful in concurrent data structures are by nature not history independent: for example, *timestamps* and *process identifiers* are often encoded into the data structure to help manage contention between threads, but they reveal information about the order of operations invoked in the past and which process invoked them; and *tombstones* are sometimes used to mark an element as deleted without physically removing it, decreasing contention, but they disclose information about deleted elements that are no longer in the dictionary. In addition to not being history independent, these standard tools of the trade require *large memory cells*, often impractically so, whereas hash tables are meant to serve as a space-efficient, lightweight dictionary. Ideally, if the universe size is $|\mathcal{U}| = u$, then each memory cell should be of width $O(\log u)$, storing a single element in the set and not much more.

Our goal. It is tricky to define history independence in a concurrent setting [7]. The bare minimum that one could ask for is *correctness* (i.e., linearizability [32]) in concurrent executions, and *history independence* when restricted to sequential executions. To date there has been no hash table satisfying even this minimal notion. A slightly stronger definition was proposed in [7]: that the data structure be linearizable in concurrent executions, and be history independent when no state-changing operation is in progress. This is called *state-quiescent history independence (SQHI)*. The open problem that we consider is:

> Is there a linearizable lock-free SQHI hash table that uses $O(m)$ memory cells, of width $O(\log u)$ each, to represent a set of m elements from a domain of size u, independent of the number of concurrent operations?

Our results. Our main result is a positive one:

Theorem 1. *There is a lock-free*[1] *SQHI concurrent hash table using the shared memory primitive LL/SC,*[2] *where each memory cell stores two elements and two bits (i.e.,* $2\lceil\log u\rceil + 2$ *bits). The expected amortized step complexity of the hash table is* $O(c)$*, where* c *is an upper bound on the number of concurrent operations that access the same element, assuming that the load*[3] *on the hash table is bounded away from 1.*

Our construction shows that issues of concurrency that are often handled by heavyweight mechanisms such as global sequence numbers, maintaining a list of all ongoing operations, and so on, can be resolved by maintaining in each memory cell a *lookahead* that stores the element in the next cell, and two bits indicating whether an insertion (resp. deletion) is ongoing in this cell.

Our implementation is based on Robin Hood hashing [19], a linear-probing hashing scheme where keys that compete for the same location i in the hash table are sorted by the distance of i from their hash value (see Section 3). We call the distance of cell i from the key's hash value the *priority* of the key in cell i. Priority-based linear-probing hash tables have been used for history independence in a sequential [17, 46] or partially-concurrent [55] setting,[4] but these constructions are insensitive to the choice of priority mechanism. We show that among all linear probing hash tables based on priority mechanisms with a lookahead, *only* the priority mechanism used in Robin Hood hashing has the property that we can conclude that an element is not in the table by reading just a single cell—a key property that our algorithm relies on.

As we said above, our hash table stores two keys and two extra bits in each cell. We complement our construction with a negative result showing that it is *necessary* to store extra information in the cells of the table beyond just a single key, in order to have both SQHI and wait freedom:

Theorem 2. *[Informal] There is no concurrent SQHI hash table representing sets of up to* $m < u$ *elements using* m *memory cells, each of which stores either a key that is in the set or* $\perp$*, and supporting wait-free lookups and obstruction-free insertions and deletions.*[5] *Furthermore, if* u *is sufficiently large and* $m < \sqrt{u}$*, then the impossibility result holds even if each cell also includes* $\lceil\log u\rceil + O(1)$ *bits.*

This result separates sequential hash tables from concurrent ones: sequential hash tables *can* be made history independent while storing one element per cell [17, 46]. Moreover, the impossibility result makes no assumptions about the step complexity of the hash table, so it applies even to highly inefficient implementations (e.g., implementations where individual operations are allowed to read or modify the entire hash table). The assumption that the dictionary can store up to m elements using m cells holds for our construction from Theorem 1, and it is necessary in Theorem 2 to avoid empty cells that are used *only* for synchronization, not for storing values from the set. We also prove that Theorem 1 is tight in the sense that we cannot achieve a *wait-free* history-independent implementation using a one-cell lookahead plus $O(1)$ bits of metadata (only lock-freedom is possible).

In the remainder of the paper we provide an overview of our main results. Additional details and proofs are available in the full version of this paper [8].

2 Preliminaries

The asynchronous shared-memory model. We use the standard model, in which a finite number of concurrent processes communicate through shared memory consisting of m *memory cells*. The shared memory is accessed by executing *primitive operations*. Our implementation uses the primitive load-link/store-conditional (LL/SC), which supports the following operations: $\mathsf{LL}(x)$ returns the value stored in memory cell x, and $\mathsf{SC}(x, \mathit{new})$ writes the value *new* to x, if it was not written to since the last time the process performed an $\mathsf{LL}(x)$ operation (otherwise, x is not modified). SC returns *true* if it writes successfully, and *false* otherwise. We also use the *validate* instruction, $\mathsf{VL}(x)$, which checks whether x has been written to since the last time the process performed $\mathsf{LL}(x)$.

An implementation of an abstract data type (ADT) specifies a program for each process and operation of the ADT; when receiving an *invocation* of an operation, the process takes *steps* according to this program. The process may change its local state or the value of a memory cell after a step, and it may return a *response* to the operation of the ADT. We focus on implementations of a *dictionary*, representing an unordered set of elements.[6] It supports the operations $\mathsf{insert}(v)$, $\mathsf{delete}(v)$ and $\mathsf{lookup}(v)$, each operation takes an input element v and returns *true* or *false*, indicating whether the element is present in the set based on the specific operation.

The *memory representation* is a vector specifying the state of each memory cell; this does not include local private variables held by each process, only the shared memory. A sequence of steps, starting from an initial configuration that specifies the state of every process and the memory representation, defines an *execution*. The system may have several initial configurations. We say that an

[1] *Lock-freedom* requires that at any point in time, if we wait long enough, *some* operation will complete.
[2] See Section 2 for the definition of LL/SC.
[3] The *load* is the ratio of the number of elements stored in the hash table to the number of memory cells.
[4] The hash table constructed in [55] is *phase-concurrent*: it can handle concurrent operations of the same type (insert, delete or lookup), but not concurrent operations of different types (e.g., inserts and lookups).

[5] *Wait-freedom* requires that every operation must terminate in a finite number of steps by the executing thread, regardless of concurrency and contention from other threads. *Obstruction-freedom* only requires an operation to terminate if the process executing it runs by itself for sufficiently long.
[6] For simplicity, we focus on storing the keys and ignore the values that are sometimes associated with the keys.

operation *completes* in execution α if α includes both the invocation and response of the operation. If α includes the invocation of an operation, but no matching response, then the operation is *pending*. We say that op_1 *precedes* op_2 in execution α, if op_1 response precedes op_2 invocation in α.

The standard correctness condition for concurrent implementations is *linearizability* [32]: intuitively, it requires that each operation appears to take place instantaneously at some point between its invocation and its return. Formally, an execution α is *linearizable* if there is a sequential permutation π of the completed, and possibly some uncompleted, operations in α, that matches the sequential specification of the ADT, with uncompleted operations assigned some output value. Additionally, if op_1 precedes op_2 in α, then op_1 also precedes op_2 in π; namely, the permutation π respects the real-time order of non-overlapping operations in α. We call this permutation a *linearization* of α. An implementation is linearizable if all of its executions are linearizable.

An implementation is *lock-free* if whenever there is a pending operation, some operation returns in a finite number of steps of all processes.

History independence. In this paper we consider implementations where the randomness is fixed up-front; after initialization, the implementation is deterministic. An example of such an initialization is choosing a hash function. It is known that in this setting, strong history independence is equivalent to requiring, for an object with state space Q, that every logical state $q \in Q$ corresponds to a unique *canonical memory representation*, fixed at initialization [28, 29].

Several notions of history independence for concurrent implementations were explored in [7]. In this paper we adopt the notion of *state-quiescent history independence* (SQHI), which requires that the memory representation be in its canonical representation at any point in the execution where *no state-changing operation* (in our case, insert or delete) is pending. The logical state of the object is determined using a linearization of the execution up to the point in which the observer inspects the memory representation.

3 Overview of the Hash Table Construction

In this section we describe our main result, the construction of a concurrent history-independent lock-free hash table where each memory cell stores two elements plus two bits. We give a high-level overview, glossing over many subtle details in the implementation.

Our hash table is based on *Robin Hood hashing* [19], a type of linear-probing hash table. In Robin Hood hashing, each element x is inserted as close to its hash location $h(x)$ as possible, subject to the following priority mechanism: for any cell i and elements $x \neq y$, element x has *higher priority in cell i* than y, denoted $x >_{p_i} y$, if cell i is farther from x's hash location than it is from y's: $i-h(x) > i-h(y)$ (or, to break ties, if $i - h(x) = i - h(y)$ and $x > y$).[7]

Robin Hood hash tables maintain the following invariant: [8]

Invariant 1 (Ordering Invariant [55])**.** *If an element v is stored in cell i, then for any cell j between $h(v)$ and i, the element v' stored in cell j has higher than or equal to priority than v (that is, $v' \geq_{p_j} v$).*

[7]Here and throughout, we assume modular arithmetic that wraps around the m-th cell.

[8]We assume that in all cells, $\perp$, indicating an empty cell, has lower priority than all other elements.

Note that we use 'higher than or equal to' instead of 'higher than', even though all elements are distinct. This is because we later use this invariant in a table that includes duplicates of the same element (see Section 4). The invariant is achieved as follows.

Let $A[0, \ldots, m-1]$ be the hash table. To insert an element v, we first try to insert it at its hash location, $h(v)$. If $A[h(v)]$ is occupied by an element v', then whichever element has lower priority in cell $h(v)$ is *displaced*, that is, pushed into the next cell. If the next cell is not empty, then the displaced element either displaces the element stored there or is displaced again,creating a chain of displacements that ends when we reach the *end of the run*: the first empty cell encountered in the probe sequence. (A *run* is a maximal consecutive sequence of occupied cells.) We place the last displaced element in the empty cell, and return. We refer to the process of shifting elements forward as *propagating the insertion.*

Deletions are also handled in a way that preserves the invariant: after deleting an element v from cell i of the hash table, we check whether the element v' in cell $i+1$ "prefers" to shift backwards, that is, whether cell i is closer to $h(v')$ than cell $i+1$. If so, we shift v' backwards into cell i and examine the next element. We continue shifting elements backwards until we reach either the end of the run (an empty cell), or an element that is already at its hash location. This is referred to as *propagating the deletion.*

Finally, to perform a lookup(v) operation, we scan the hash table starting at location $h(v)$, until we either find v and return *true*, reach the end of the run and return *false*, or find an element with lower priority than v (that is, reach a cell j such that $A[j] <_{p_j} v$). In the latter case, the invariant allows us to conclude that v is not in the hash table, and we return *false.*

The crucial property of Robin Hood hashing that makes it suitable for our concurrent implementation is that following an insertion or deletion, *no element in the hash table moves by more than one cell.* In Section 3.1, we discuss why, among all priority mechanisms, only Robin Hood hashing is suitable for our implementation.

As we mentioned in Section 1, several traditional mechanisms for ensuring progress in the face of contention are unavailable to us, because they compromise history independence or increase the cell size too much (or both). We begin by outlining the challenges involved in constructing a concurrent history-independent Robin Hood hash table, and then describe our implementation and how it overcomes them.

Handling displaced elements. In Robin Hood hashing, insertions may result in a chain of displacements, with multiple higher-priority elements "bumping" (displacing) lower-priority elements to make room for a newly-inserted element. In a sequential implementation, if we wish to "bump" a lower-priority element y in favor of a higher-priority element x, we simply store y in local memory, overwrite the cell containing y in the hash table to place x there instead, and then proceed to find a place for y in the hash table. While a new place for y is sought, it exists only in the local memory of the operation performing the insertion. In a *concurrent* implementation this approach is dangerous: if a displaced element y does not physically exist in the hash table, then a concurrent lookup(y) operation may mistakenly report that y is not in the hash table. This is not allowed in a linearizable implementation.[9]

Avoiding moves that happen behind a lookup's back. In sequential Robin Hood hashing, if a $\mathsf{lookup}(x)$ operation is invoked when x is not in the set, the operation traverses the table starting from location $h(x)$, until it reaches an empty cell or a cell i with a value $y <_{p_i} x$ and returns *false*. However, in a concurrent implementation this is risky: a $\mathsf{lookup}(x)$ operation might scan from location $h(x)$ until it reaches a cell i with a value $y <_{p_i} x$, never seeing element x, *even though x is in the set the entire time*. This can happen if x is initially ahead of the location currently being examined by the $\mathsf{lookup}(x)$ operation, but then, between steps of the $\mathsf{lookup}(x)$ operation, multiple elements stored between x and location $h(x)$ are deleted, pulling x backwards to a location that the $\mathsf{lookup}(x)$ operation has already passed.

Synchronization without timestamps, process IDs, and operation announcements. In our implementation we wish to store as little information as possible in each cell, both for the sake of memory-efficiency and also to facilitate history independence (extraneous information must be eventually wiped clear, and this can be challenging). To ensure progress, operations must be able to "clear the way" for themselves if another operation is blocking the way, without *explicitly knowing* what operation is blocking them, or which process invoked that operation, as we do not wish to store that information in the hash table.

3.1 Robin Hood Hashing with 1-Lookahead and Lightweight Helping

To overcome the challenges described above we introduce a *lookahead* mechanism. Each cell i has the form $A[i] = \langle value, lookahead, mark\rangle$, consisting of three slots: the *value* slot stores the element currently occupying cell i, or $\perp$ for an empty cell; the *lookahead* slot is intended to store the element occupying cell $i+1$, when no operation is in progress; and the *mark* slot indicates the status of the cell, and can take on three values, $mark \in \{S, I, D\}$, with S standing for "stable", indicating that no operation is working on this cell, and I, D indicating that an insertion or deletion (resp.) are working on this cell. If the cell is stable (S), then its *lookahead* is consistent with the next cell: if $A[i] = \langle a, b, S\rangle$ and $A[i+1] = \langle c, d, M\rangle$, then $b = c$. If a cell is marked with I or D, its *lookahead* might be inconsistent.

The role of the lookahead. Adding a lookahead serves two purposes. First, it allows lookups to safely conclude that an element is not in the hash table by reading a single cell: if we are looking for element x, and we reach a cell $A[i] = \langle v, v', *\rangle$ such that either $i = h(x)$ and $x >_{p_i} v$, or $v >_{p_i} x >_{p_{i+1}} v'$, then x can safely be declared to not be in the hash table. This resolves the concern that an element may be moved "back and forth" behind a lookup's back, so that the lookup can never safely decide that the element is not in the hash table. Second, the lookahead serves as temporary storage for elements undergoing displacement due to an insertion: instead of completely erasing a lower-priority element y and writing a higher-priority element x in its place, we temporarily shift y from the *value* slot to the *lookahead* slot of the cell, and store x in the *value* slot. Later on, we move y into the *value* slot of the next cell, possibly displacing a different element by shifting it into the *lookahead* slot, and so on. At any time, all elements that are in the table are physically stored in shared memory, either in the *value* slot or the *lookahead* slot of some cell — ideally, in the *value* slot of some cell and also in the *lookahead* slot of the preceding cell.

The choice of Robin Hood hashing. Any priority-based linear-probing hash table satisfies the ordering invariant (Invariant 1). However, Robin Hood hashing also satisfies the following additional invariant: If an element v' stored in cell i has higher priority than v, then for any cell j between $h(v)$ and i, the element v'' stored in cell j also has higher priority than v. This invariant allows us to determine the following by reading only two consecutive cells, cell i which stores element v' and cell $i+1$ which stores element v'': (1) v is in the table ($v = v'$ or $v = v''$), (2) v is not in the table and should be inserted to cell $i+1$ ($i+1 = h(v)$ and $v >_{p_{i+1}} v''$, or $v' >_{p_i} v >_{p_{i+1}} v''$), (3) if v is in the table, it can only be in a cell before cell i ($v' <_{p_i} v$), or (4) if v is in the table, it can only be in a cell after cell $i+1$ ($v'' >_{p_{i+1}} v$). This allows the algorithm to make decisions based on a single cell with a *lookahead*, without depending on the entire sequence of values read starting from the initial hash location, which may have changed since they were last read. We show that it is not possible to determine that element v is not in the hash table using any other priority mechanism by reading only two consecutive cells. We do so by showing that every two consecutive cells in the canonical memory representation of a set that does not include v is equal to the same cells in a different canonical memory representation of a set that does include v, where the priority mechanism determines the canonical memory representations. (See the full version [8].)

The role of the mark. The *mark* slot can be viewed as a *lock*, except that cells are locked by *operations*, not by processes; to release the lock, a process must *help* the operation that placed the mark, completing whatever steps are necessary to propagate the operation one step forward, from the "locked" cell into the next cell (which is then "locked" by the operation). Operations move forward in a manner that resembles hand-over-hand locking [13]; we explain this in detail below.

Life cycle of an operation. Each operation — $\mathsf{lookup}(x)$, $\mathsf{insert}(x)$ or $\mathsf{delete}(x)$ — goes through some or all of the following stages:

(1) Finding the "correct position" for element x: starting from position $h(x) - 1$,[10] we scan each cell, until we either find element x in the hash table, or we find the cell where element x *should have been* if it were in the hash table, and conclude that x is not in the hash table. As discussed above, the choice of Robin Hood hashing guarantees that if element x is not in the table (and there are no ongoing operations), we can find such a cell. At this point, $\mathsf{lookup}(x)$ returns the appropriate answer, and $\mathsf{insert}(x)$, $\mathsf{delete}(x)$ may also return, if x is already in the hash table (for $\mathsf{insert}(x)$) or is not in the hash table (for $\mathsf{delete}(x)$). Note that if x is found in a *lookahead* slot of a cell with a delete operation in progress, we may

[9]Unless y itself is concurrently being inserted or deleted; however, Robin Hood hashing displaces elements upon insertion or deletion of *other* elements, so we are not guaranteed this.

[10]To detect concurrent operations on element x, we need to start the scan in location $h(x) - 1$, not $h(x)$; this will become clear below.

not determine that v is in the table. This is to ensure the implementation is linearizable.

Along the way, operations *help* advance any other operation that they encounter: if an operation encounters a non-stable cell, it performs whatever steps are necessary to make the cell stable, and only then is allowed to proceed.

(2) Let $i-1$ be the cell reached, where cell i is the "correct position" for element x. The *lookahead* of cell $i-1$ should reflect the absence (for insert) or presence (for delete) of element x.[11] Then the operation makes its initial write into the table:
- insert(x) writes x into the *lookahead* slot of cell $i-1$: if $A[i-1] = \langle v, v', S\rangle$, then the insert writes $A[i-1] = \langle v, x, I\rangle$, "locking" the cell. At this point, the newly-inserted element x is considered to be displaced, as it is stored only in the *lookahead* slot of cell $i-1$, immediately preceding its correct location (cell i).
- delete(x) logically erases x from the *lookahead* slot of cell $i-1$: if $A[i-1] = \langle v, x, S\rangle$, then the delete writes $A[i-1] = \langle v, x, D\rangle$, "locking" the cell. The cell now contains a logical "hole" (the *lookahead* slot containing the value v), which may need to be filled by moving backwards elements from cells $A[i+1], A[i+2], \ldots$ until we reach either the end of the run or an element that is already in its hash position.

(3) Following the initial write, operations continue moving forward until the end of the run, *propagating* any operations that they encounter, to resolve the chain of displacements that may occur. In some sense, processes "lose their identity" in this stage: because we do not use process IDs, timestamps or sequence numbers, processes can no longer keep track of their own operation and distinguish it from other operations, and must instead help propagate all operations equally. By proceeding all the way to the end of the run and helping all operations along the way, a process ensures that by the time it returns, its own operation has been completed, either by itself or some other process, or a different process becomes responsible for ensuring the operation completes (see more details below).

3.2 Propagating an Operation

Consider consecutive cells $A[i] = \langle a, b, M\rangle, A[i+1] = \langle c, d, M'\rangle$, with the mark $M \in \{I, D\}$ in cell $A[i]$ indicating that an operation is in progress. Operations are not allowed to overtake one another, so if cell $A[i+1]$ is not stable ($M' \neq S$), we *help* whatever operation is in progress there by propagating it forward one step, clearing the way for the current operation. Note that helping an operation in cell $A[i+1]$ may require us to first clear the way by helping an operation in cell $A[i+2]$, which may in turn require helping an operation in cell $A[i+3]$, and so on.

Now suppose that cell $A[i+1]$ is stable. Propagating the operation that is currently in cell $A[i]$ one step forward into cell $A[i+1]$ involves modifying both $A[i]$ and $A[i+1]$; however, we cannot modify both cells in one atomic step, as we are using single-word memory primitives. Thus, the propagation is done by a careful interleaving of LL, VL and SC steps, in a manner that is similar to hand-over-hand locking [13], except that the "locks", represented by the *mark* slot, are owned by *operations* rather than *processes*: this allows any process to take over the propagation of an operation that stands in its way, even if it did not invoke that operation. To modify cells $A[i]$ and $A[i+1]$ when cell $A[i]$ is already "locked", an operation first "locks" cell $A[i+1]$, setting its *mark* appropriately (and also changing its contents, that is, the *value* and *lookahead* slots); then it releases the "lock" on cell $A[i]$, setting the *mark* to S (and possibly also changing its contents).

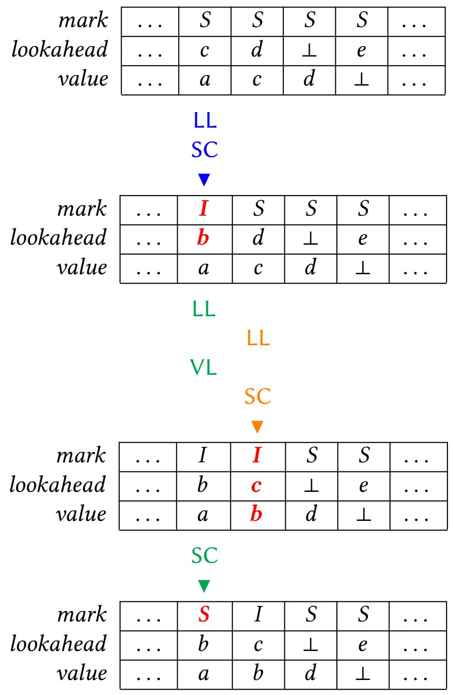
In the sequel, we make distinguish between *operations* and *processes.* An *operation* that is in mid-propagation (i.e., at any point following the initial write and before the operation is complete) and is currently located in cell $A[i]$ can be propagated by any *process* that reaches $A[i]$. As noted above, following their initial write (and also before it, on their way to their target location), processes are in some sense nameless workers that simply propagate all the operations they encounter, until they reach the end of the run. However, prior to the initial write of an operation, only the process that invoked the operation knows of it, so at this point the operation is synonymous with the process that invoked it.

Propagating an insertion (see Fig. 1). Suppose that $A[i] = \langle a, b, I\rangle$ and $A[i+1] = \langle c, d, S\rangle$, with cell $A[i]$ "locked" by a propagating insertion. Element b has been displaced by the insertion: it is temporarily stored in the *lookahead* slot of $A[i]$, and we need to move it forward into the *value* slot of $A[i+1]$, displacing element c instead (unless $c = \bot$, in which case $A[i+1]$ is an empty cell, and no further elements need to be displaced). The target state following the propagation step is $A[i] = \langle a, b, S\rangle, A[i+1] = \langle b, c, I\rangle$, if $c \neq \bot$ (element c is now displaced), and $A[i] = \langle a, b, S\rangle, A[i+1] = \langle b, d, S\rangle$ if $c = \bot$ (no element is displaced, and the insertion is done propagating). We describe here the case where $c \neq \bot$ (the other case is very similar).

We move from the current state, $A[i] = \langle a, b, I\rangle, A[i+1] = \langle c, d, S\rangle$, to the target state, $A[i] = \langle a, b, S\rangle, A[i+1] = \langle b, c, I\rangle$, in several steps: we begin by performing an LL on cell $A[i]$, an LL on cell $A[i+1]$, and then a VL on cell $A[i]$, to ensure that it has not been altered while we were reading $A[i+1]$. Then we "lock" $A[i+1]$ and at the same time modify its contents, by performing an SC to set $A[i+1] \leftarrow \langle b, c, I\rangle$. This SC may fail, but since $A[i]$ is already "locked" (it is marked with I), no other operation can overtake the current insertion without helping it move forward. Thus, there are only two possible reasons for a failed SC on $A[i+1]$: either some other process already performed the same SC, in which case we can proceed to the next step; or a new operation performed its initial write into $A[i+1]$, thereby "locking" it for itself. In this case we must first help propagate the new operation to clear the way, and then we can resume trying to propagate the current insertion.

Suppose the SC on $A[i+1]$ succeeds (either the current process succeeded, or some other process did). At this point we have $A[i] = \langle a, b, I\rangle, A[i+1] = \langle b, c, I\rangle$, with both cells "locked" by the insertion. We now "release the lock" on $A[i]$ by performing an SC to set $A[i] \leftarrow \langle a, b, S\rangle$. This SC may also fail, but in this case, since cell $A[i]$ itself is "locked" prior to the SC, the only possible reason is that some other process performed the same SC successfully. Thus,

[11] This is the reason that scans for element x begin in location $h(x)-1$: it is possible that x is currently being inserted or deleted, but the change is reflected only in location $h(x)-1$ at this point.



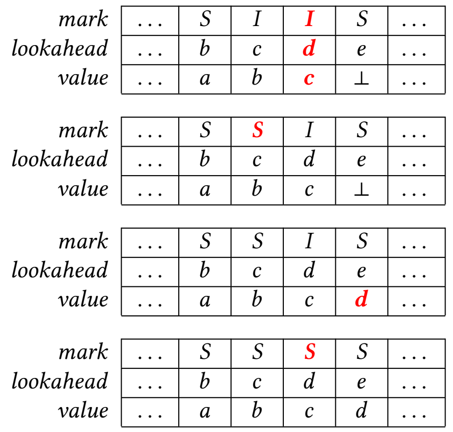
mark	...	S	S	S	S	...
lookahead	...	c	d	$\perp$	e	...
value	...	a	c	d	$\perp$	...

mark	...	I	S	S	S	...
lookahead	...	b	d	$\perp$	e	...
value	...	a	c	d	$\perp$	...

mark	...	I	I	S	S	...
lookahead	...	b	c	$\perp$	e	...
value	...	a	b	d	$\perp$	...

mark	...	S	I	S	S	...
lookahead	...	b	c	$\perp$	e	...
value	...	a	b	d	$\perp$	...

(a) The initial write of insert(b) into a cell $A[i]$, followed by its first propagation step. The sequence of LL, VL and SC operations is depicted from top to bottom.



mark	...	S	I	I	S	...
lookahead	...	b	c	d	e	...
value	...	a	b	c	$\perp$	...

mark	...	S	S	I	S	...
lookahead	...	b	c	d	e	...
value	...	a	b	c	$\perp$	...

mark	...	S	S	I	S	...
lookahead	...	b	c	d	e	...
value	...	a	b	c	d	...

mark	...	S	S	S	S	...
lookahead	...	b	c	d	e	...
value	...	a	b	c	d	...

(b) The rest of the propagation steps, omitting the LL, VL and SC operations.

Figure 1: Depiction of the insertion process of element b, initiated by operation insert(b), where b is inserted into a cell $A[i] = \langle a, c, S\rangle$ such that $a >_{p_i} b >_{p_{i+1}} c$.

there is no need to even check if the SC succeeded; we simply move on to the next cell.

Propagating a deletion (see Fig. 2). Recall that the initial write of a delete(x) operation targets the cell preceding the location of x, "locking" it by marking it with D. Deletions maintain the following invariant as they propagate: if cell $A[i] = \langle a, b, D\rangle$ is "locked" by a deletion (i.e., marked with D), and the subsequent cell is stable, $A[i+1] = \langle c, d, S\rangle$, then the *value* slot of $A[i+1]$ (i.e., the value c) is either $\perp$ or redundant: in the latter case, it is either the target of the deletion (immediately following the initial write), or we have already copied it into cell $A[i]$, so that $a = c$ (following subsequent propagation steps). Since cell $A[i+1]$ is stable, its *lookahead* matches the *value* slot of the next cell, $A[i+2] = \langle d, e, *\rangle$. Now there are three cases:

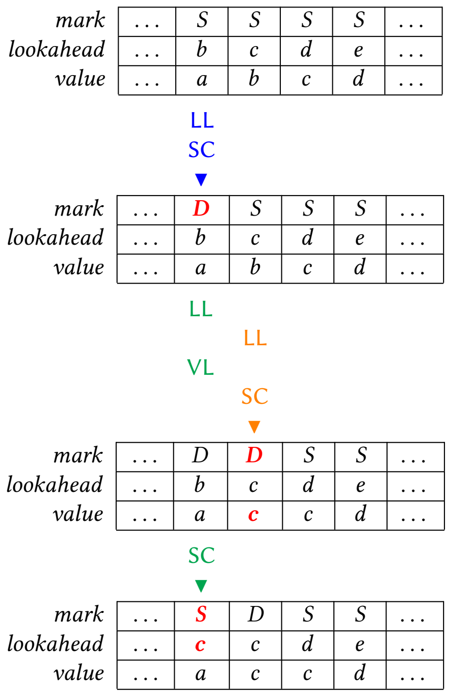
- If $d = \perp$, then we have reached the end of the run, and we need to set $A[i] = \langle a, \perp, S\rangle, A[i+1] = \langle \perp, \perp, S\rangle$. The delete operation is then done propagating.
- If $d \neq \perp$ and $h(d) = i+2$, then element d is already in its hash location, $A[i+2]$, and does not need to be shifted backwards. In this case we need to set $A[i] = \langle a, \perp, S\rangle, A[i+1] = \langle \perp, d, S\rangle$, "puncturing" the run and completing the propagation of the delete operation. However, the process that punctured the run is not allowed to return immediately, for reasons that we explain below.
- Finally, if $d \neq \perp$ and $h(d) \neq i+2$, then element d should be shifted one step back (closer to $h(d)$), from $A[i+2]$ to $A[i+1]$. In this case we need to set $A[i] = \langle a, d, S\rangle, A[i+1] = \langle d, d, D\rangle$, duplicating element d. Notice that the invariant is maintained: following the update, the *value* slot of cell $A[i+2] = \langle d, e, *\rangle$, which remains untouched,[12] is redundant, because we copied d backwards into cell $A[i+1]$. We duplicate element d instead of deleting it from the *lookahead* slot to preserve the next invariant: The element in the *lookahead* slot is either consistent with the next cell, or it directly precedes the element in the *value* slot of the next cell in a table constructed according to Robin Hood hashing, containing all the elements in A.
  In this final case the deletion is not done propagating: we still need to delete d from cell $A[i+2]$, and we may also need to shift subsequent elements one step back, closer to their hash locations.

The changes to cells $A[i]$ and $A[i+1]$ are done in a manner similar to the way insertions are propagated, by interleaving LL, VL and SC operations so that we first "lock" cell $A[i+1]$ and update its contents at the same time, then "release" cell $A[i]$ and update its contents at the same time.

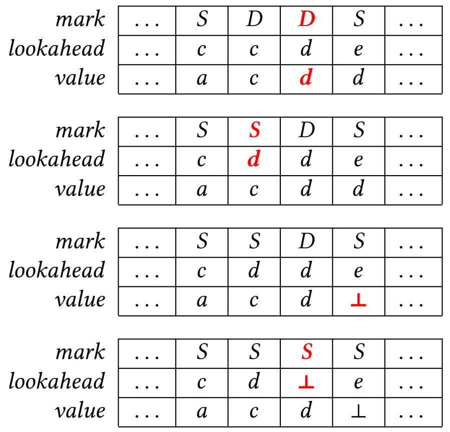
Puncturing a run. One delicate point is that if a process p punctures a run R by splitting it into two runs R_1, R_2 with an empty cell between them, and then p returns (upon "reaching the end of the run" R_1), then some operations may become stranded in the second part, R_2, with no process working to propagate them. This can happen, for example, if prior to the puncture a lookup operation helped another operation o and pushed it into R_2, but then found its target and returned prior to completing operation o. The process q that originally invoked o may lag behind, so that it reaches the end of R_1 only after the run is punctured. In this case, if q returns, operation o will be stranded with no process working in R_2.

To avoid this scenario, it suffices to have any process p that punctures a run continue into the second part and propagate all operations it finds there. However, this is also problematic, because

[12]By the current operation. Other operations may make their initial write into cell $i+2$, but if they do, they update the *lookahead* slot, not the *value* slot.



(a) The initial write of delete(b) into a cell $A[i] = \langle a, b, S\rangle$, followed by one propagation step. The sequence of LL, VL and SC operations is depicted from top to bottom.



(b) The rest of the propagation steps, omitting the LL, VL and SC operations. The propagation of the delete operation completes upon encountering element e where $h(e) = i + 4$, that is, element e is already in its hash location.

Figure 2: Depiction of the deletion process of element b, initiated by operation delete(b).

we cannot allow a situation where some operations are incomplete, and a process p that invoked a lookup operation is the *only* process propagating them: this violates SQHI. Therefore we distinguish between two cases: if a process q whose original operation is an insert or delete punctures a run, it continues into the second part. On the other hand, if a process q whose original operation is a lookup encounters a location where it needs to puncture a run in order to help propagate a delete operation, it does not do so. This can cause the indication that an element is not in the table to be "split" across two cells, rather than just one, by an ongoing insert operation that cannot overtake an ongoing delete operation that needs to puncture a run to complete. To address this, we design a mechanism that allows lookup operations to identify that an element is not in the table based on the values of two cells instead of just one (Lines 50–53).

Restarting an operation. There are two scenarios that cause an operation $o(x)$ to restart, and both occur prior to a successful initial write into the table. The first is if o is an insertion or deletion that has found the "correct" location for element x, but the initial write into the table, which is done using SC, fails. This indicates that the operation may need to re-position itself, as the "correct" location for element x may have changed, and we do this by restarting the operation. The second scenario occurs during the lookup stage, when the process searches for element x: if the process reads a cell $A[i] = \langle a, b, *\rangle$ such that $b >_{p_i} x$, but the next cell is $A[i+1] = \langle c, d, *\rangle$ such that $c <_{p_{i+1}} x$, then there is a chance that element x was ahead of location i when $A[i]$ was read, but due to concurrent deletions, it was moved behind cell i by the time $A[i+1]$ was read. To avoid false negatives, the process restarts its lookup from position $h(x) - 1$.

We note that restarting from scratch is only for simplicity; instead, we can move backwards cell-by-cell until reaching a cell $A[j] = \langle a, b, *\rangle$ such that either $j = h(x) - 1$ or $a >_{p_j} x$, and then resume forward movement. Each step back can be blamed on a concurrent deletion that moved element x one step backwards, so this is more efficient in workloads with low contention of deletes.

4 Code and Proof Outline for the History-Independent Hash Table

The pseudocode for the insert, delete and lookup operations is presented in Algorithm 1, Algorithm 2, and Algorithm 3, respectively. Procedure help_op, presented in Algorithm 4, describes the code for helping propagate an operation to the next cell by some process. Procedure propagate, presented in Algorithm 5, describes the code for ensuring that an operation completes its propagation.

The hash table A is of size m, and is initialized to $A[i] = \langle \perp, \perp, S\rangle$, for every $0 \le i < m$. An operation may abort if the table is full and there is no vacant cell to insert a new element. This can happen when a process initiating an insert operation does not find a vacant cell for the new element (Line 16), or when an insert operation cannot be propagated to the next cell without violating the ordering invariant, indicating there is no vacant cell for the displaced element (Line 72).

Next, we sketch the proof that the hash table linearizable, SQHI and lock-free; see details in the full version [8]. We prove correctness assuming there is always an empty stable cell available to insert a new element. This guarantees that no operation aborts and that some operation can always propagate to the next cell.

The proof relies on the following key invariant that extends the ordering invariant to also account for the *lookahead* slot and element repetition. For a cell $A[i]$, $0 \le i < m$, that contains the value $\langle a, b, M\rangle$, let $A[i].val = a$, $A[i].next = b$ and $A[i].mark = M$.

Algorithm 1 Pseudocode for insert

```
    insert($v$):
 1: $i \leftarrow h(v) - 1$
 2: $\langle a, b, M\rangle \leftarrow$ LL($A[i]$)
 3: $first \leftarrow true$
 4: repeat
 5:     if $a = v$ or ($b = v$ and ($M \neq D$ or $h(b) \neq i+1$)) then
 6:         return $false$                          ▷ $v$ is already in the table
 7:     if $M \neq S$ then                          ▷ $A[i]$ is unstable
 8:         help_op($i$)
 9:         goto Line 18
10:     if $b = \perp$ or $v >_{p_{i+1}} b$ then     ▷ Insert position found
11:         if SC($A[i], \langle a, v, I\rangle$) then
12:             propagate($h(v), i, \{I\}$)
13:             return $true$
14:         goto Line 1
15:     $i \leftarrow i + 1$
16:     if $\neg first$ and $i = h(v)$ then abort()  ▷ No empty cell to insert $v$
17:     $first \leftarrow false$
18:     $\langle a, b, M\rangle \leftarrow$ LL($A[i]$)
19: until $v >_{p_i} a$
20: goto Line 1
```

Algorithm 2 Pseudocode for delete

```
    delete($v$):
21: $i \leftarrow h(v) - 1$
22: $\langle a, b, M\rangle \leftarrow$ LL($A[i]$)
23: $first \leftarrow true$
24: repeat
25:     if ($i = h(v)$ and $v >_{p_i} a$) or ($a >_{p_i} v >_{p_{i+1}} b$
            and ($M = S$ or $h(b) \neq i+1$)) then return $false$
                                                ▷ $v$ is not in the table
26:     if $M \neq S$ then                          ▷ $A[i]$ is unstable
27:         help_op($i$)
28:         goto Line 40
29:     if $a = v$ then ▷ Step back one cell to locate $v$ in the *lookahead* slot
30:         $i \leftarrow i - 1$
31:         goto Line 40
32:     if $b = v$ then                             ▷ Delete position found
33:         if SC($A[i], \langle a, v, D\rangle$) then
34:             propagate($h(v), i, \{D\}$)
35:             return $true$
36:         goto Line 21
37:     $i \leftarrow i + 1$
38:     if $\neg first$ and $i = h(v)$ then return $false$
                        ▷ The loop iterated through all cells in the table
39:     $first \leftarrow false$
40:     $\langle a, b, M\rangle \leftarrow$ LL($A[i]$)
41: until $i \neq h(v)$ and $v >_{p_i} a$
42: goto Line 21
```

Invariant 2 (Extended Ordering Invariant). *For every $0 \leq i < m$:*

(a) If $A[i].val = v$, then for any $h(v) \leq j < i$, $A[j].val \geq_{p_j} v$.
(b) $A[i].next \geq_{p_{i+1}} A[i+1].val$ or $h(A[i].next) = i + 1$.
(c) Either $A[i].val \geq_{p_i} A[i].next$ or $h(A[i].next) = i + 1$ and $A[i].next \geq_{p_{i+1}} A[i+1].val$.

Algorithm 3 Pseudocode for lookup

```
    lookup($v$):
43: $i \leftarrow h(v) - 1$
44: $\langle a, b, M\rangle \leftarrow$ LL($A[i]$)
45: $first \leftarrow true$
46: repeat
47:     if $a = v$ or ($b = v$ and ($M \neq D$ or $h(b) \neq i+1$)) then
48:         return $true$                            ▷ $v$ is in the table
49:     if ($i = h(v)$ and $v >_{p_i} a$) or ($a >_{p_i} v >_{p_{i+1}} b$
            and ($M = S$ or $h(b) \neq i+1$)) then return $false$
                                                ▷ $v$ is not in the table
50:     if $M = I$ then ▷ Check if the indication that $v$ is not in the table is
                                                split across two cells
51:         $\langle c, *, *\rangle \leftarrow$ LL($A[i+1]$)
52:         if $b >_{p_i} v >_{p_{i+1}} c$ and $h(b) \neq i+1$ and VL($A[i]$) then
53:             return $false$
54:     if $M \neq S$ then help_op($i$)
55:     $i \leftarrow i + 1$
56:     if $\neg first$ and $i = h(v)$ then return $false$
                        ▷ The loop iterated through all cells in the table
57:     $first \leftarrow false$
58:     $\langle a, b, M\rangle \leftarrow$ LL($A[i]$)
59: until $i \neq h(v)$ and $v >_{p_i} a$
60: goto Line 43
```

Invariant 2(a) is the ordering invariant for the *value* slots in a table that can store multiple copies of the same element (recall that deletions create duplications). It was shown that this ordering invariant guarantees a unique canonical representation for every *multiset* of elements [55]. Specifically, in Robin Hood hashing, duplicates of the same element are stored in consecutive cells in the canonical representation. Invariant 2(b) and Invariant 2(c) ensure that the "correct position" of the element in the *lookahead* slot is the next cell, unless it has lower priority than the *value* slot in the same cell and the next cell is its initial hash location, which indicate the table is full. Invariant 2 allows us to use the properties of Robin Hood described earlier.

First, we show that the propagation of operations follows a pattern similar to hand-over-hand locking. Assuming that $A[i]$ is locked (i.e., it contains an ongoing insert or delete operation), we then lock $A[i+1]$, release $A[i]$, lock $A[i+2]$, release $A[i+1]$, and so on. Then, relying on the correct propagation of operations, we can show that Invariant 2 holds at all points during the execution of the algorithm. In addition, we show that if a cell is stable (i.e., cell i such that $A[i].mark = S$), then its *lookahead* slot is consistent with the next cell, and there are no duplicate elements in this cell.

Linearizability. We order the insert and delete operations that actually insert or delete elements from the table and change the state of the dictionary according to the order of their initial writes. In most cases, we can show that a lookup(v) operation returns *true* if and only if v is present in the table A, either in a *lookahead* or *value* slot, during its last read of A, and v is present in the table A if and only if v is in the set by the order of operations defined above. (This also applies for insert and delete operations that return *false*.) The only exception is when the initial write by an insert(v) or delete(v) operation has occurred, but the first propagation of the

Algorithm 4 Pseudocode for help_op

```
    help_op($i$):
61: $\langle a, b, M_1 \rangle \leftarrow$ LL($A[i]$)
62: $\langle c, d, M_2 \rangle \leftarrow$ LL($A[i+1]$)
63: if $M_1 \neq S$ then
64:     while $M_2 \neq S$ and ($M_1 \neq D$ or $M_2 \neq D$ or $b = c$) and
            ($M_1 \neq I$ or $b \neq c$) and ($M_1 \neq D$ or $c \neq \bot$) do
65:         $\langle a, b, M_1 \rangle \leftarrow \langle c, d, M_2 \rangle$
66:         $i \leftarrow i + 1$
67:         $\langle c, d, M_2 \rangle \leftarrow$ LL($A[i+1]$)
                ▷ ($M_1 \neq S$ and $M_2 = S$) or ($M_1 = M_2 = D$ and $b \neq c$)
                  or ($M_1 = I$ and $b = c$) or ($M_1 = D$ and $c = \bot$)
68:     if $M_1 = I$ and VL($A[i]$) then
69:         $\langle x, y, M_3 \rangle \leftarrow$ LL($A[i-1]$)
70:         if $M_3 = I$ and $y = a$ and VL($A[i]$) then
                ▷ Ensure $A[i-1]$ is unlocked
71:             SC($A[i-1], \langle x, y, S \rangle$)
72:         if $c >_{p_{i+1}} b$ and not a lookup operation then abort()
                ▷ Table is full
73:         else if $b = c$ then SC($A[i], \langle a, b, S \rangle$)
                ▷ $A[i+1]$ is already locked or propagation ended
74:         else if $c = \bot$ then DSC($i+1, \langle b, d, S \rangle, i, \langle a, b, S \rangle$)
                ▷ End of the propagation
75:         else DSC($i+1, \langle b, c, I \rangle, i, \langle a, b, S \rangle$)
                ▷ Propagate insert to $A[i+1]$ and unlock $A[i]$
76:     else if $s_1 = D$ and VL($A[i]$) then
77:         $\langle x, y, M_3 \rangle \leftarrow$ LL($A[i-1]$)
78:         if $M_3 = D$ and $y \neq a$ and VL($A[i]$) then
                ▷ Ensure $A[i-1]$ is unlocked
79:             SC($A[i-1], \langle x, a, S \rangle$)
80:         if $c = \bot$ or $M_2 = D$ then SC($A[i], \langle a, c, S \rangle$)
                ▷ $A[i+1]$ is already locked or propagation ended
81:         else if $M_2 = S$ then
                ▷ Propagate delete to $A[i+1]$ and unlock $A[i]$
82:             if $d \neq \bot$ and $h(d) \neq i+2$ then
83:                 DSC($i+1, \langle d, d, D \rangle, i, \langle a, d, S \rangle$)
84:             else if not a lookup operation then
85:                 if DSC($i+1, \langle \bot, d, S \rangle, i, \langle a, \bot, S \rangle$) then
                        ▷ End of the propagation
86:                     if $d \neq \bot$ then propagate($i+1, i+2, \{I, D\}$)
                            ▷ Punctured a run

    DSC($i, tup_1, j, tup_2$):
        ▷ Write $tup_1$ to $A[i]$ and if successful, write $tup_2$ to $A[j]$
87: if SC($A[i], tup_1$) then
88:     SC($A[j], tup_2$)
89:     return $true$
90: else
91:     $tup \leftarrow$ LL($A[i]$)
92:     if $tup.val = tup_1.val$ then SC($A[j], tup_2$)
```

operation has not yet completed. An operation on element v which starts before the initial write may be unaware of such insertion or deletion. As a result, it can conclude that v is not in the set (resp. in the set) when it is present (resp. not present) in the table, before the first propagation completes. However, since this operation must be concurrent with the uncompleted insert or delete operation, we can simply place this operation before the uncompleted operation in the order, ensuring that the operation returns the correct response according to the order of operations.

Algorithm 5 Pseudocode for propagate

```
     propagate($i, j, A$):                                  ▷ $A \subseteq \{I, D\}$
93:  repeat
94:      $\langle a, b, M \rangle, prev \leftarrow A[j]$
95:      while $\langle a, b, M \rangle = prev$ and $M \in A$ do   ▷ Continue while $A[j]$
                 does not change and contains a delete or insert operation
96:          help_op($j$)
97:          $prev \leftarrow \langle a, b, M \rangle$
98:          $\langle a, b, M \rangle \leftarrow A[j]$
99:      $j \leftarrow j + 1$
100: until ($a = \bot$ or ($M = S$ and $b = \bot$)) or $j = i$
             ▷ Stop upon reaching an empty cell or back to the start
```

History independence. When all cells are stable, there are no duplicate elements in the table, and the *lookahead* slot in each cell is equal to the *value* slot of the next cell. Together with Invariant 2, this implies that if no operation is propagating, the table is in the canonical representation of the set of elements stored in the table, as defined by Robin Hood hashing. As discussed in Section 3.2, we show that when an operation returns, it either completes its propagating, or a different ongoing insert or delete operation takes over responsibility for finishing the propagation. This ensures that when there are no ongoing insert or delete operations, the table is in its canonical memory representation.

Lock-freedom. A failed SC operation indicates that there was either a successful initial write by some operation or a successful propagation of some operation. We show that an operation can propagate to each cell at most once before it completes. Since there is a finite number of processes, there can only be a finite number of initial writes before an operation finishes propagating. Finally, we show that eventually, some process must return; otherwise, after some time, there will be no new initial writes, and all operations must complete propagating. In this case, the table becomes constant with only stable cells, and some process must eventually identify this and return. The only exception is when only lookup operations take steps. In this case, we show that even if operations cannot finish propagating, since lookup operations do not puncture runs, some operation can still find the element it is looking for or detect its absence from the table.

Amortized step complexity. Consider a batch of n operations, each invoked by a different process,[13] and let c be an upper bound on the number of operations that access the same element (whether they be insertions, deletions or lookups). Assume that the ratio between the number of insertions in the batch and the size of the hash table is bounded by some constant $\alpha \in (0, 1)$.[14] We prove that in expectation over the hash function, even under a *worst-case scheduler* that knows the hash function, the total number of steps required to complete all operations is $O(n \cdot c)$. We sketch the proof; see the full version [8] for details.

Fix a process p that invokes an operation o on element x (i.e., $o \in \{\text{insert}(x), \text{delete}(x), \text{lookup}(x)\}$). Let N be a random variable representing the length of the run that contains $h(x)$, if we schedule

[13]This assumption is made to simplify the modeling and analysis, but it is not essential.
[14]Here, too, a more fine-grained analysis is possible, taking into consideration only elements that are in the hash table at the same time, but for simplicity we take the total number of insertions as an upper bound.

all insertions from the batch prior to scheduling operation o. This is an upper bound on the distance from $h(x)$ to the "correct location" for element x (the cell where x is stored, or if x is not in the set, the cell where x *would* be stored if it were in the set). Also, let $P \leq N \cdot c$ be the number of processes working on elements in the same run as x. We design a *charging scheme* that charges each step taken by process p to some operation — either its own operation o, or an operation working in the same run as element x — in such a way that the total charge for any operation is at most $O(N^3 \cdot c)$. We then prove that if the load on the hash table is bounded away from 1, then $\mathbb{E}[N^3] = O(1)$, where the expectation is over the choice of the hash function. By linearity of expectation, the expected number of steps required for all operations to complete is $O(n \cdot c)$.

The charging scheme assigns to each operation all *successful* steps performed "on its behalf", regardless of which process performed them. Propagating an operation one position forward requires only a constant number of successful steps, so the total charge here is $O(N)$. In contrast, *failed* steps are charged to the operation whose successful SC caused the failure. For example, if a process p attempts to propagate an operation o from position i to position $i + 1$, but some other process q succeeds in doing so faster than p, then the $O(1)$ steps that process p "wasted" are charged to operation o; but if q's failure is instead caused by some operation $o' \neq o$ making its initial write into position $i + 1$, then the wasted steps are charged to operation o'. The largest charge is for causing a process to restart its operation prior to its initial write into the table; the process may have wasted up to N steps, and all are charged to the operation whose successful SC caused the restart. Each operation is propagated at most once across each position in the run (at most N positions), and each propagation step involves a constant number of successful SCs, which each fail up to P processes, possibly causing them to restart and costing N wasted steps. Thus, the total charge to an operation is bounded by $O(N \cdot N \cdot P) = O(N^3 \cdot c)$.

5 Overview of the Lower Bound

We give a very brief overview of the impossibility result of Theorem 2. The full details appear in [8]. The idea is to show that there exists some element $v \in \mathcal{U}$ for which both of the following hold:

(1) For every memory cell ℓ, there exist two sets $S, S' \subseteq \mathcal{U}$ where $v \in S$ and $v \notin S'$, but in the canonical representation of S and S', memory cell ℓ has the same value. Thus, by reading location ℓ, a lookup cannot unambiguously decide whether v is in the set.

(2) For every memory cell ℓ, there exist two pairs of sets, $S_0, S'_0 \subseteq \mathcal{U}$ and $S_1, S'_1 \subseteq \mathcal{U}$, such that $v \notin S_0, S'_0$ (resp. $v \in S_1, S'_1$) but cell ℓ has a different value in the canonical representation of S_0 than it does in S'_0 (resp. in S_1 and in S'_1).

We then construct two executions α_0, α_1, such that in α_0, element v is not in the dictionary throughout, and in α_1, element v *is* in the dictionary throughout; however, there is a lookup(v) operation that cannot distinguish α_0 from α_1, and therefore cannot return. To "confuse" the lookup operation, every time it accesses a cell ℓ using an operation whose behavior depends only on the cell's current value (such as read/write or compare-and-swap), we use property (1) to extend executions α_0, α_1 in a way that the reader cannot distinguish; and if the access is using VL or SC, we use property (2) to first "overwrite" the value of cell ℓ, ensuring the VL or SC returns *false* in both executions. We remark that property (1) suffices to rule out implementations from memory primitives whose behavior depends only on the current value of the memory cell and not the history, and if we use only such primitives, we do not need to assume that the size of the dictionary matches the size of the largest set it can store; the dictionary can have any size $m < u$.

We show that any *natural assignment*—which maps each set to a canonical memory representation where each memory cell can hold either an element currently in the set or $\perp$ (indicating vacancy), along with additional metadata bits — satisfies property 1, subject to a restriction on the number of memory cells, which depends on the universe size, maximal size of the set and number of metadata bits. The definition of natural assignments allows for multiple copies of the same element and ensures that every element in the dictionary appears in at least one memory cell. For a dictionary from m memory cell that can store at most m elements, a natural assignment with any number of metadata bits that satisfies property (1) also satisfies property (2) for the same element, which implies Theorem 2.

6 Related Work

History-independent data structures. Efficient sequential history-independent data structure constructions include fast HI constructions for cuckoo hash tables [45], linear-probing hash tables [17, 27], other hash tables [17, 46], trees [1, 42], memory allocators [27, 46], write-once memories [44], priority queues [18], union-find data structures [46], external-memory dictionaries [14, 23–25], file systems [9, 11, 12, 52], cache-oblivious dictionaries [14], order-maintenance data structures [17], packed-memory arrays/list-labeling data structures [14, 15], and geometric data structures [59]. Given the strong connection between history independence and unique representability [28, 29], some earlier data structures can also be made history independent, including hashing variants [2, 19], skip lists [49], treaps [5], and other less well-known deterministic data structures with canonical representations [3, 4, 50, 56, 58]. There is work on the time-space tradeoff for strongly history-independent hash tables, see e.g., [38–40] and references therein.

The algorithmic work on history independence has found its way into systems. There are now voting machines [16], file systems [9, 10, 12], databases [11, 48, 53], and other storage systems [20] that support history independence as an essential feature.

History independence for concurrent data structures. To our knowledge, the only work to study history independence in a fully concurrent setting is [7], which investigated several possible definitions for history independence in a concurrent system that might never quiesce. Several lower bounds and impossibility results are proven in [7]. In particular, it is shown that if the observer is allowed to examine the memory at any point in the execution, then in any concurrent implementation that is obstruction free, for any two logical states q, q' of the ADT such that some operation can cause a transition from q to q', the memory representations of q, q' in the implementation must differ by exactly one memory cell. This rules out open-addressing hash tables, unless their size is $|\mathcal{U}|$, because it does not allow us to move elements around.

Sequential hash tables. There are several popular design paradigms for hash tables, including probing, both linear and other probing patterns (e.g., [2, 19, 33, 36, 41]); cuckoo hashing [47]; and chained hashing [37]. These can all be made history independent [17, 27, 45, 46], typically by imposing a canonical order on the elements in the hash table. This involves moving elements around, which can be challenging for concurrent implementations. Notably, for sequential history-independent hash tables, history independence does not come at the cost of increasing the width of memory cells—it suffices to store one element per cell; our work shows that in the concurrent setting, storing one element per cell is not enough.

Concurrent hash tables. There is a wealth of literature on non-history-independent concurrent hash tables; we discuss here only the work directly related to ours, which includes concurrent implementations of linear probing hash tables in general, and Robin Hood hashing in particular. We note that while [7] gives a universal wait-free history-independent construction that can be used to implement any abstract data type, including a dictionary, the construction does not allow for true concurrency: processes compete to perform operations, with only one operation going through at a time. In addition, the universal construction of [7] uses memory cells whose width is linear in the state of the data type (in the case of a dictionary, the size of the set that it represents), and in the number of processes.

There are several implementations of concurrent linear-probing hash tables, e.g., [22, 57]. It is easier to implement a concurrent linear-probing hash table if one is not concerned with history independence: since the elements do not need to be stored in a canonical order, we can simply place newly-inserted elements in the first empty cell we find, and we do not need to move elements around. Tombstones can be used to mark deleted elements, thus avoiding the need to move elements once inserted.

As for Robin Hood hashing, which gained popularity due to its use in the Rust programming language, there are not many concurrent implementations. Kelly et al. [35] present a lock-free concurrent Robin Hood hash table that is not history independent. Their design is based on a k-CAS primitive, which is then replaced with an optimized implementation from CAS [6]. This implementation uses standard synchronization techniques that require large memory cells: timestamps, operation descriptors, and so on. Bolt [34] is a concurrent resizable Robin Hood hash table, with a lock-free fast path and a lock-based slow path. This implementation is not lock-free, although it does avoid the use of locks in lightly-contended cases (the fast path).

Shun and Blelloch [55] present a *phase-concurrent* deterministic hash table based on linear probing, which supports one type of operation within a synchronized phase. Different types of operations (inserts, deletes and lookups) may not overlap. The hash table of [55] is based on the same history-independent scheme used in [17, 46]. In this scheme, each memory cell stores a single element; this does not contradict our impossibility result, because none of these implementations support overlapping lookups and operations that modify the hash table (inserts and/or deletes).

Acknowledgments

Hagit Attiya is partially supported by the Israel Science Foundation (grant number 22/1425). Rotem Oshman and Noa Schiller are funded by NSF-BSF Grant No. 2022699. Michael Bender and Martín Farach-Colton are funded by NSF Grants CCF-2106999, CCF-2118620, CNS-1938180, CCF-2118832, CCF-2106827, CNS-1938709, CCF-2247577.

References

[1] Umut A Acar, Guy E Blelloch, Robert Harper, Jorge L Vittes, and Shan Leung Maverick Woo. 2004. Dynamizing static algorithms, with applications to dynamic trees and history independence. In *Proc. of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA).* Society for Industrial and Applied Mathematics, USA, 531–540. https://doi.org/10.5555/982792.982871

[2] Ole Amble and Donald Ervin Knuth. 1974. Ordered hash tables. *Comput. J.* 17, 2 (Jan. 1974), 135–142. https://doi.org/10.1093/comjnl/17.2.135

[3] Arne Andersson and Thomas Ottmann. 1991. Faster uniquely represented dictionaries. In *Proc. of the 32nd Annual IEEE Symposium on Foundations of Computer Science (FOCS).* IEEE Computer Society, Los Alamitos, CA, USA, 642–649. https://doi.org/10.1109/SFCS.1991.185430

[4] Arne Andersson and Thomas Ottmann. 1995. New tight bounds on uniquely represented dictionaries. *SIAM J. Comput.* 24, 5 (1995), 1091–1103. https://doi.org/10.1137/S0097539792241102

[5] Cecilia R Aragon and Raimund G Seidel. 1989. Randomized search trees. In *Proc. of the 30th Annual IEEE Symposium on Foundations of Computer Science (FOCS).* IEEE Computer Society, Los Alamitos, CA, USA, 540–545. https://doi.org/10.1109/SFCS.1989.63531

[6] Maya Arbel-Raviv and Trevor Brown. 2017. Reuse, Don't Recycle: Transforming Lock-Free Algorithms That Throw Away Descriptors. In *31st International Symposium on Distributed Computing (DISC).* Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 4:1–4:16. https://doi.org/10.4230/LIPIcs.DISC.2017.4

[7] Hagit Attiya, Michael A. Bender, Martín Farach-Colton, Rotem Oshman, and Noa Schiller. 2024. History-Independent Concurrent Objects. In *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing (PODC).* Association for Computing Machinery, New York, NY, USA, 14–24. https://doi.org/10.1145/3662158.3662814

[8] Hagit Attiya, Michael A. Bender, Martín Farach-Colton, Rotem Oshman, and Noa Schiller. 2025. History-Independent Concurrent Hash Tables. arXiv:2503.21016 [cs.DC] Full version of this paper.

[9] Sumeet Bajaj, Anrin Chakraborti, and Radu Sion. 2015. Practical Foundations of History Independence. arXiv:1501.06508 [cs.CR] https://arxiv.org/abs/1501.06508

[10] Sumeet Bajaj, Anrin Chakraborti, and Radu Sion. 2016. Practical Foundations of History Independence. *IEEE Trans. Inf. Forensics Secur.* 11, 2 (2016), 303–312. https://doi.org/10.1109/TIFS.2015.2491309

[11] Sumit Bajaj and Radu Sion. 2013. Ficklebase: Looking into the future to erase the past. In *Proc. of the 29th IEEE International Conference on Data Engineering (ICDE).* IEEE Computer Society, Los Alamitos, CA, USA, 86–97. https://doi.org/10.1109/ICDE.2013.6544816

[12] Sumeet Bajaj and Radu Sion. 2013. HIFS: History independence for file systems. In *Proc. of the ACM SIGSAC Conference on Computer & Communications Security (CCS).* Association for Computing Machinery, New York, NY, USA, 1285–1296. https://doi.org/10.1145/2508859.2516724

[13] Rudolf Bayer and Mario Schkolnick. 1977. Concurrency of operations on B-trees. *Acta informatica* 9 (1977), 1–21. https://doi.org/10.1007/BF00263762

[14] Michael A. Bender, Jon Berry, Rob Johnson, Thomas M. Kroeger, Samuel McCauley, Cynthia A. Phillips, Bertrand Simon, Shikha Singh, and David Zage. 2016. Anti-Persistence on Persistent Storage: History-Independent Sparse Tables and Dictionaries. In *Proc. 35th ACM Symposium on Principles of Database Systems (PODS).* Association for Computing Machinery, New York, NY, USA, 289–302. https://doi.org/10.1145/2902251.2902276

[15] Michael A. Bender, Alex Conway, Martín Farach-Colton, Hanna Komlós, William Kuszmaul, and Nicole Wein. 2022. Online List Labeling: Breaking the $\log^2 n$ Barrier. In *Proc. 63rd IEEE Annual Symposium on Foundations of Computer Science (FOCS).* IEEE Computer Society, Los Alamitos, CA, USA, 980–990. https://doi.org/10.1109/FOCS54457.2022.00096

[16] John Bethencourt, Dan Boneh, and Brent Waters. 2007. Cryptographic methods for storing ballots on a voting machine. In *Proc. of the 14th Network and Distributed System Security Symposium (NDSS).* 209–222. https://www.ndss-symposium.org/ndss2007/cryptographic-methods-storing-ballots-voting-machine/

[17] Guy E Blelloch and Daniel Golovin. 2007. Strongly history-independent hashing with applications. In *Proc. of the 48th Annual IEEE Symposium on Foundations of Computer Science (FOCS).* IEEE Computer Society, Los Alamitos, CA, USA, 272–282. https://doi.org/10.1109/FOCS.2007.36

[18] Niv Buchbinder and Erez Petrank. 2003. Lower and Upper Bounds on Obtaining History Independence. In *Advances in Cryptology - CRYPTO 2003*, Dan Boneh (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 445–462. https://doi.org/10.1007/978-3-540-45146-4_26
[19] Pedro Celis, Per-Åke Larson, and J. Ian Munro. 1985. Robin Hood Hashing (Preliminary Report). In *26th Annual Symposium on Foundations of Computer Science (FOCS'85)*. IEEE Computer Society, Los Alamitos, CA, USA, 281–288. https://doi.org/10.1109/SFCS.1985.48
[20] Bo Chen and Radu Sion. 2015. HiFlash: A History Independent Flash Device. arXiv:1511.05180 [cs.CR] https://arxiv.org/abs/1511.05180
[21] Cliff Click. 2007. A Lock-Free Wait-Free Hash Table. http://www.stanford.edu/class/ee380/Abstracts/070221 Accessed on 2024-07-04.
[22] H. Gao, J. F. Groote, and W. H. Hesselink. 2005. Lock-free dynamic hash tables with open addressing. *Distrib. Comput.* 18, 1 (2005), 21–42. https://doi.org/10.1007/s00446-004-0115-2
[23] Daniel Golovin. 2008. *Uniquely Represented Data Structures with Applications to Privacy.* Ph. D. Dissertation. Carnegie Mellon University, Pittsburgh, PA, 2008.
[24] Daniel Golovin. 2009. B-treaps: A uniquely represented alternative to B-Trees. In *Proc. of the 36th Annual International Colloquium on Automata, Languages, and Programming (ICALP).* Springer Berlin Heidelberg, Berlin, Heidelberg, 487–499. https://doi.org/10.1007/978-3-642-02927-1_41
[25] Daniel Golovin. 2010. The B-Skip-List: A Simpler Uniquely Represented Alternative to B-Trees. arXiv:1005.0662 [cs.DS]
[26] Michael T. Goodrich, Evgenios M. Kornaropoulos, Michael Mitzenmacher, and Roberto Tamassia. 2016. More Practical and Secure History-Independent Hash Tables. In *Computer Security – ESORICS 2016 (Lecture Notes in Computer Science, Vol. 9879).* Springer, Cham, 20–38. https://doi.org/10.1007/978-3-319-45741-3_2
[27] Michael T. Goodrich, Evgenios M. Kornaropoulos, Michael Mitzenmacher, and Roberto Tamassia. 2017. Auditable Data Structures. In *Proc. IEEE European Symposium on Security and Privacy (EuroS&P).* IEEE Computer Society, Los Alamitos, CA, USA, 285–300. https://doi.org/10.1109/EuroSP.2017.46
[28] Jason D. Hartline, Edwin S. Hong, Alexander E. Mohr, William R. Pentney, and Emily Rocke. 2002. Characterizing History Independent Data Structures. In *Proceedings of the Algorithms and Computation, 13th International Symposium (ISAAC).* Springer Berlin Heidelberg, Berlin, Heidelberg, 229–240. https://doi.org/10.1007/3-540-36136-7_21
[29] Jason D Hartline, Edwin S Hong, Alexander E Mohr, William R Pentney, and Emily C Rocke. 2005. Characterizing history independent data structures. *Algorithmica* 42, 1 (2005), 57–74. https://doi.org/10.1007/s00453-004-1140-z
[30] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. 2020. *The Art of Multiprocessor Programming, Second Edition.* Elsevier. https://doi.org/10.1016/C2011-0-06993-4
[31] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. 2008. Hopscotch Hashing. In *Proceedings of the 22nd International Symposium on Distributed Computing* (Arcachon, France) *(DISC '08).* Springer-Verlag, Berlin, Heidelberg, 350–364. https://doi.org/10.1007/978-3-540-87779-0_24
[32] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (jul 1990), 463–492. https://doi.org/10.1145/78969.78972
[33] F. R. A. Hopgood and J. Davenport. 1972. The quadratic hash method when the table size is a power of 2. *Comput. J.* 15, 4 (1972), 314–315. https://doi.org/10.1093/comjnl/15.4.314
[34] Endrias Kahssay. 2021. *A fast concurrent and resizable Robin-Hood hash table.* Master's thesis. Massachusetts Institute of Technology.
[35] Robert Kelly, Barak A. Pearlmutter, and Phil Maguire. 2019. Concurrent Robin Hood Hashing. In *22nd International Conference on Principles of Distributed Systems (OPODIS) (Leibniz International Proceedings in Informatics (LIPIcs)).* Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 10:1–10:16. https://doi.org/10.4230/LIPIcs.OPODIS.2018.10
[36] Don Knuth. 1963. Notes on "Open" Addressing.
[37] Robert L. Kruse. 1984. *Data Structures and Program Design.* Prentice-Hall Inc, Englewood Cliffs, New Jersey, USA.
[38] William Kuszmaul. 2023. Strongly History-Independent Storage Allocation: New Upper and Lower Bounds. In *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS).* IEEE Computer Society, Los Alamitos, CA, USA, 1822–1841. https://doi.org/10.1109/FOCS57990.2023.00111
[39] Tianxiao Li, Jingxun Liang, Huacheng Yu, and Renfei Zhou. 2023. Tight Cell-Probe Lower Bounds for Dynamic Succinct Dictionaries . In *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS).* IEEE Computer Society, Los Alamitos, CA, USA, 1842–1862. https://doi.org/10.1109/FOCS57990.2023.00112
[40] Tianxiao Li, Jingxun Liang, Huacheng Yu, and Renfei Zhou. 2024. Dynamic Dictionary with Subconstant Wasted Bits per Key. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA).* Society for Industrial and Applied Mathematics, 171–207. https://doi.org/10.1137/1.9781611977912.9
[41] W. D. Maurer. 1968. An Improved Hash Code for Scatter Storage. *Commun. ACM* 11, 1 (1968), 35–38. https://doi.org/10.1145/362851.362880
[42] Daniele Micciancio. 1997. Oblivious data structures: applications to cryptography. In *Proc. of the 29th Annual ACM Symposium on Theory of Computing (STOC).* Association for Computing Machinery, New York, NY, USA, 456–464. https://doi.org/10.1145/258533.258638
[43] Maged M. Michael. 2002. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA).* Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/564870.564881
[44] Tal Moran, Moni Naor, and Gil Segev. 2007. Deterministic history-independent strategies for storing information on write-once memories. In *Proc. 34th International Colloquium on Automata, Languages and Programming (ICALP).* Springer Berlin Heidelberg, Berlin, Heidelberg, 303–315. https://doi.org/10.1007/978-3-540-73420-8_28
[45] Moni Naor, Gil Segev, and Udi Wieder. 2008. History-independent cuckoo hashing. In *Proc. of the 35th International Colloquium on Automata, Languages and Programming (ICALP).* Springer, 631–642. https://doi.org/10.1007/978-3-540-70583-3_51
[46] Moni Naor and Vanessa Teague. 2001. Anti-persistence: history independent data structures. In *Proc. of the 33rd Annual ACM Symposium on Theory of Computing (STOC).* Association for Computing Machinery, New York, NY, USA, 492–501. https://doi.org/10.1145/380752.380844
[47] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122 – 144. https://doi.org/10.1016/j.jalgor.2003.12.002
[48] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. 2016. Arx: An Encrypted Database using Semantically Secure Encryption. Cryptology ePrint Archive, Paper 2016/591. https://doi.org/10.14778/3342263.3342641
[49] William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (1990), 668–676. https://doi.org/10.1145/78973.78977
[50] William Pugh and Tim Teitelbaum. 1989. Incremental computation via function caching. In *Proc. of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).* Association for Computing Machinery, New York, NY, USA, 315–328. https://doi.org/10.1145/75277.75305
[51] Chris Purcell and Tim Harris. 2005. Non-blocking Hashtables with Open Addressing. In *Distributed Computing, 19th International Conference, DISC (Lecture Notes in Computer Science, Vol. 3724).* Springer-Verlag, Berlin, Heidelberg, 108–121. https://doi.org/10.1007/11561927_10
[52] Daniel S. Roche, Adam J. Aviv, and Seung Geol Choi. 2015. A Practical Oblivious Map Data Structure with Secure Deletion and History Independence. arXiv:1505.07391 [cs.CR] https://arxiv.org/abs/1505.07391
[53] Daniel S. Roche, Adam J. Aviv, and Seung Geol Choi. 2016. A Practical Oblivious Map Data Structure with Secure Deletion and History Independence. In *IEEE Symposium on Security and Privacy (SP).* IEEE Computer Society, Los Alamitos, CA, USA, 178–197. https://doi.org/10.1109/SP.2016.19
[54] Ori Shalev and Nir Shavit. 2003. Split-ordered lists: lock-free extensible hash tables. In *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing (PODC).* Association for Computing Machinery, New York, NY, USA, 102–111. https://doi.org/10.1145/1147954.1147958
[55] Julian Shun and Guy E. Blelloch. 2014. Phase-Concurrent Hash Tables for Determinism. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures* (Prague, Czech Republic) *(SPAA '14).* Association for Computing Machinery, New York, NY, USA, 96–107. https://doi.org/10.1145/2612669.2612687
[56] Lawrence Snyder. 1977. On uniquely represented data structures. In *Proc. of the 18th Annual IEEE Symposium on Foundations of Computer Science (FOCS).* IEEE Computer Society, Los Alamitos, CA, USA, 142–146. https://doi.org/10.1109/SFCS.1977.22
[57] Alex Stivala, Peter J. Stuckey, Maria Garcia de la Banda, Manuel Hermenegildo, and Anthony Wirth. 2010. Lock-free parallel dynamic programming. *J. Parallel and Distrib. Comput.* 70, 8 (2010), 839–848. https://doi.org/10.1016/j.jpdc.2010.01.004
[58] Rajamani Sundar and Robert Endre Tarjan. 1990. Unique binary search tree representations and equality-testing of sets and sequences. In *Proc. of the 22nd Annual ACM Symposium on Theory of Computing (STOC).* Association for Computing Machinery, New York, NY, USA, 18–25. https://doi.org/10.1145/100216.100219
[59] Theodoros Tzouramanis. 2012. History-independence: a fresh look at the case of R-trees. In *Proc. 27th Annual ACM Symposium on Applied Computing (SAC).* Association for Computing Machinery, New York, NY, USA, 7–12. https://doi.org/10.1145/2245276.2245279

Received 2024-11-04; accepted 2025-02-01