Mathematical Models to Analyze Lua Hybrid Tables and Why They Need a Fix

Conrado Martínez, Cyril Nicaud, Pablo Rotondo

1Dept. of Computer Science, Universitat Politècnica de Catalunya, Campus Nord, Omega-241, Jordi Girona, 1–3, Barcelona, 08034, Spain.
2Laboratoire d’Informatique Gaspard-Monge (LIGM) UMR 8049, Université Gustave Eiffel, Bâtiment Copernic, 5, boulevard Descartes, Cité Descartes, Champs-sur-Marne, 77454, France.

*Corresponding author(s), E-mail(s): pablo.rotondo@univ-eiffel.fr; Cyril.Nicaud@univ-eiffel.fr; Contributing authors: conrado@cs.upc.edu; cyril.nicaud@univ-eiffel.fr;

Abstract
Lua (Ierusalimschy et al., 1996) is a well-known scripting language, popular among many programmers, most notably in the gaming industry. Remarkably, the only data-structuring mechanism in Lua are associative arrays, called tables. With Lua 5.0, the reference implementation of Lua introduced hybrid tables to implement tables using both a hashmap and a dynamically growing array combined together: the values associated with integer keys are stored in the array part, when suitable, everything else is stored in the hashmap. All this is transparent to the user, who gets a unique simple interface to handle tables. In this paper we carry out a theoretical analysis of the performance of Lua’s tables, by considering various worst-case and probabilistic scenarios. In particular, we uncover some problematic situations for the simple probabilistic model where we add a new key with some fixed probability $p > \frac{1}{2}$ and delete a key with probability $1 - p$: the cost of performing $T$ such operations is proved to be $\Omega(T \log T)$ with high probability, where linear complexity is expected instead.

Keywords: Algorithm engineering, Data structures, Probabilistic analysis of algorithms, Hash tables, Lua
1 Introduction

When implementing the standard algorithms and data structures of a new programming language, engineers usually follow the classical solutions that have been validated by both practice and theory. Sometimes, however, they innovate and propose new ideas that fit best with the internal implementation of the language or that behave better with the typical data of their intended audience. For example, this was the case for the sorting algorithm TimSort used in the main implementation of Python.¹ This new, elegant and powerful sorting algorithm was quickly adopted by several other languages, including Java.² After a decade of existence, computer scientists started analyzing its efficiency, helped to fix some issues and confirmed its excellent performances at a theoretical level [1–3]. The actual principal implementation of the language Lua revisits the way maps, i.e., associative arrays, are structured internally, in a new and innovative structure named table. Studying this novel way of structuring data from a theoretical point of view is the main purpose of this article.

Lua³ is a scripting programming language [4] created in the early nineties and adopted by many programmers, especially for the development of gaming applications; it keeps a base of tens of thousands of users worldwide. Like many scripting languages, Lua is characterized by its simplicity and extensibility, with the aim to help integrate code written in different programming languages.

The only data-structuring mechanism in Lua are so-called tables: this was a design decision which made the language simple, yet flexible and powerful. If $H$ is a Lua table then the assignment $H[x] = y$ associates the value $y$ to the key $x$, for whatever $x$ and $y$, and regardless of their types. If $x$ was already a key in $H$, then the value associated to $H$ is updated and changed to $y$. If $x$ is not present, then the instruction inserts the pair $(x, y)$ in the table $H$. The expression $H[x]$ actually returns a reference to the place where the value associated to $x$ is stored or the special value nil if the key is not present in the table; the reference can be then used to obtain the sought value or to assign a new value. To “delete” a pair $(x, y)$ from $H$ it is enough to assign nil to $H[x]$. Everything is transparent to the user, including how the table grows to accommodate more and more pairs, or how unused space is restored back to the memory heap for future use.

Until Lua 4.0, tables were implemented strictly as hashmaps: all pairs $(x, y)$ were explicitly stored in a single hashmap, irrespective of the type of the keys $x$. Lua 5.0 brought on a new implementation of the tables in order to optimize their use as arrays: pairs with integer keys are stored in a separate actual array, without storing their keys, provided that the index (the integer key) falls within the current range $[1, \ldots, n]$ of the array [5]. The value $n$ changes dynamically so that the array always contains $> n/2$ non-nil values. All other pairs, when the key is not an integer, or it is outside the current range of the array, are stored in the hashmap as usual. The new Lua (hybrid) tables thus have two parts, called the hash-part or hashmap, and the array-part or array.

¹ https://github.com/python/cpython/blob/main/Objects/listsort.txt
² https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html
³ From the Portuguese “lua”, meaning moon.
Main contributions & Plan of the paper. In this paper, we provide a theoretical analysis covering some aspects of the performance of Lua hybrid tables. We first consider the implementation of hashmaps (Section 2). It is quite direct to establish that, in presence of both insertions and deletions, Lua’s hashmap is suboptimal in the worst case (amortized). This is due to incorrect resizing parameter settings of the hashmap, which can be exploited to construct a sequence of insertions and deletions that frequently trigger a recalculation of the entire hashmap.

However, this worst case instance is very specific, and one can legitimately argue that it is very unlikely to happen in practice. Our main contribution is to propose a simple yet reasonable probabilistic scenario for insertions and deletions in a data structure, then establish that under this model, an operation in a Lua hashmap takes $\Omega(\log n)$ time. This is done in Section 3.4, and this strongly advocates for changes in the implementations, which are discussed in the conclusion.

After that, in Section 4, we investigate the performance of the hybrid table as a whole, this time focusing on sequences of insertions involving integer keys, which should exploit, as much as possible, the array component of Lua’s hybrid tables. We show first that a carefully crafted sequence of $n$ insertions will require super-linear cost (Example 2 and Proposition 8). We also show that less adverse scenarios, in particular, some that may arise naturally in practice, will need expected constant amortized time per insertion, however the array part will be empty most of the time (Lemma 10), and thus the advantages of the hybrid scheme can become blurred (Theorem 9).

Finally, in the conclusions (Section 5), we present some striking experiments, comparing the performance of Lua with a Lua modified according to a classical solution to avoid the drawback of the hybrid table’s implementation.

A preliminary version of this paper was presented in a conference extended abstract [6].

2 Hashmaps in Lua

2.1 Description of the hashmap algorithms

In this section, we give a precise description of the hashmap algorithms in Lua, corresponding to the version 5.4.5.

When non-empty, a hashmap in Lua consists of an array of size $M = 2^m$ for some $m \geq 0$. Each slot contains a key, a value and the index next of the next location to probe in the search sequence if the sought key is not the one at that slot (next is nil if there is no successor). When both the key and the value are nil, the slot is empty; for slots that have been deleted the value has been set to nil, but the key is retained; the slots that contain the actual elements of the table have both their keys and their value fields non-nil. Throughout the article, a slot is said to be used, deleted or free when it contains a pair (key, value), a pair (key, nil) or a pair (nil, nil), respectively.

---

4 All detailed descriptions in the remaining of the paper and our analysis refer to version 5.4.4 of Lua (the most recent).
5 The source code can be found in the file https://www.lua.org/source/5.4/ltable.c.html
In addition to the array, the data structure also keeps an index `last_free` pointing to the first slot that must be checked when looking for a free slot in a downwards scanning of the array. Initially, we set `last_free` to `M − 1`, and it can only decrease.

**Search.** The search for a key `x` simply consists in computing its hash value then following the `next` links until we find the key `x` and return the associated value (success if it is not `nil`) or the end of the list (failure). Notice that returning the value of the last probed slot, we will return `nil` whenever `x` is not in the table (either because `x` was deleted or because we reach a free slot).

**Deletion.** To delete a key `x`, we search for it. The associated value of the slot where the search ends is set to `nil` (it might already be `nil` if `x` was not in the table). The `next` field remains unaltered to maintain the chaining.

**Insertion.** If one wants to set `x ↦→ y` and the key `x` is already in the table, even with a deleted status, we just update its associated value to `y`. If `x` is not already there, let `i` be the position corresponding to the hash value of `x` (the main position of `x`, in Lua parlance). If the slot `i` is free or deleted, the key `x` and its value `y` are put there, with a `nil` `next`-link when the slot is free and keeping the `next`-link value if it is deleted. Otherwise, there is a collision with another pair `{x', y'} (y' ≠ nil) at the position `i`. There are two distinct cases when it happens. If `x'` is at its main position, a free slot is found for `{x, y}` and the chaining is updated so that `{x, y}` is on the second position of the linked list starting at index `i`. This is easily done by updating the two `next`-links at index `i` and at the free slot. If `x'` is not at its main position, it is moved to a free slot, which requires a list scan starting from the hash value of `x'` to find its predecessor in its chaining. Then `{x, y}` is set at index `i`, with a `nil` `next`-link, starting a new chain within the table.

Looking for a free slot is accomplished by scanning the table from right to left, starting at position `last_free`, until a free slot is found. Importantly, deleted slots are ignored during this process in Lua, to avoid problems with the chaining. If no free slot is found, i.e., `last_free` exits the left boundary of the array, then a rehash occurs: the number of used keys `n` is determined, then a new hashmap of size `2^m` is allocated, where `m` is the smallest integer such that `n + 1 ≤ 2^m`. All pairs `{keys, value}` and the pair `{x, y}` are then inserted into the new map.

So in any case, just after its insertion, `{x, y}` is at the first or second place in its chain. The algorithms are depicted in Fig. 1 and examples of insertions in a Lua hashmap are depicted in Fig. 2.
Algorithm 1: Insertion($T, x, y$)

1. $i \leftarrow \text{Hash}(x)$
2. if $T[i].value = \text{nil}$ then $(T[i].key, T[i].value) \leftarrow (x, y)$
3. else
   4. $f \leftarrow \text{GetFreePos}(T)$
5. if $f = \text{nil}$ then $T \leftarrow \text{ReHash}(T, x, y)$
6. else
   7. $j \leftarrow \text{Hash}(T[i].key)$
8. if $i = j$ then
   9. $(T[f].key, T[f].value) \leftarrow (x, y)$
10. $T[f].next \leftarrow T[i].next$
11. $T[i].next \leftarrow f$
   else
   12. $p \leftarrow \text{Predecessor}(T, i)$
13. $T[f] \leftarrow T[i]$
14. $T[p].next \leftarrow f$
15. $(T[i].key, T[i].value, T[i].next) \leftarrow (x, y, \text{nil})$

Algorithm 2: ReHash($T, x, y$)

1. $n, M \leftarrow 0, 1$
2. for $i \text{ index of } T$ do
3. if $T[i].value \neq \text{nil}$ then $n \leftarrow n + 1$
4. while $M < n$ do $M \leftarrow 2M$
5. $T' \leftarrow \text{Allocate}(M)$
6. for $i \text{ index of } T$ do
7. if $T[i].value \neq \text{nil}$ then Insertion($T', T[i].key, T[i].value$)
8. Insertion($T', x, y$)
9. return $T'$

Algorithm 3: GetFreePos($T$)

1. while $T\.last\.free \geq 0$ and $T[T\.last\.free].key \neq \text{nil}$ do
2. $T\.last\.free \leftarrow T\.last\.free - 1$
3. if $T\.last\.free = -1$ then return nil
4. else return $T\.last\.free$

Fig. 1: Algorithm for the insertion of a pair key/value $(x, y)$ in a Lua hashmap (instruction $T[x]=y$ in Lua), when $x$ is not already a key in $T$. Predecessor just iterates through the chaining of $T[i]$ to find its predecessor.
Fig. 2: If there is no deleted value and if there is a collision with a key that is in its main position, the new inserted element is placed in a free spot and is at the second position in its chaining. If the colliding key is not at its main position, the new element is put there, and the colliding element is placed in the free spot.

Fig. 3: If the main position of the newly inserted key has a deleted value, it is just put there. If it is used, then the insertion proceeds as previously, considering the deleted spot as occupied for the search of a free spot. Observe that at the end, the hash values of 2 and 7 share the same chain, which could not happen with no deletion.

The algorithms of Figure 1 correspond to the C implementation as follows: the algorithm Insertion corresponds to luaH_newkey, the algorithm GetFreePos corresponds to getfreepos, and ReHash to rehash. More precisely, luaH_newkey performs the insertion of a new element, while the function luaH_set corresponds to an assignment $H[x] = y$, where the key $x$ could already be present in the table $H$ (the value is updated in this case).
2.2 Settings and analysis when there is no deletion

Designing accurate hash functions is a whole field on its own, and it is not the topic of this paper. So, throughout the article, we consider that for a hashmap of length \( M \), the hash values of different keys are independent uniform random integers of \( \{0, \ldots, M - 1\} \). They are sampled again when a rehash occurs. This is the standard assumption for such analysis that do not go into the details of specific hash functions [7]. This model is called the simple uniform hashing.

In the absence of deletions, Lua’s hashmaps behave as separate chaining hashing [7]. If \( N \) is the number of elements in the table and \( M \) the size of the table, the load factor is classically defined as \( \alpha = N/M \leq 1 \). The average cost (measured as the number of slots inspected) of successful searches (\( S_N \)) and unsuccessful searches (\( U_N \)) is [7, §6.4, p. 525], as \( N \) and \( M \) tends to infinity: \( S_N \approx 1 + \frac{\alpha}{2} \) and \( U_N \approx 1 + \frac{\alpha^2}{2} \). Note that it may seem strange that \( U_N \leq S_N \), but it is because of the implicit conditioning that successful searches do not consider free slots. It is worth mentioning here that if we used coalescing chaining, that is, adding a new pair \( \langle x, y \rangle \) always as successor of a conflicting pair \( \langle x', y' \rangle \), whether that pair sits at its main location or not, then scanning the table to look for the Predecessor of \( \langle x', y' \rangle \) would be unnecessary and the performance of successful and unsuccessful searches would degrade only slightly, namely to \( S_N \approx 1 + \frac{2\alpha - 1 - 2\alpha^2}{2\alpha} \) and \( U_N \approx 1 + \frac{2\alpha^2 - 1 - 2\alpha^3}{4\alpha} \) (see [7, §6.4, p. 524]). The improvement granted by pure separate chaining “is not enough of an improvement over [coalesced chaining] to warrant changing the algorithm [=using separate chaining instead of coalesced chaining]” [7].

Classically, the rehashing procedure by doubling the capacity has a constant amortized cost, as the pointer last_free cannot be decreased more than \( M \) times. In conclusion, everything is well known when there is no deletion, and the expected amortized cost of an insertion is \( O(1) \).

2.3 An Unlikely Worst-Case Scenario When Considering Deletions

In this section, we establish that the situation changes significantly when we consider deletions, and that the expected amortized cost in not constant anymore. We estimate the time taken by the whole process by counting the number of times the function to insert a key is called: it is called once when an insertion is performed, unless a rehash occurs, in which case it is also called once for every key having an assigned value. Clearly, this number of calls is a lower bound for the complexity of the whole process. We suppose that keys are not integers, so only the hash-part is studied in this section.

**Example 1** (deletions-insertions in a full hashmap). If we delete an element from a full hashmap of size \( M = 2^m \) and then perform an insertion of a new element, we rehash the whole table into the same size and the hash is going to be full again. The only case it does not happen if when the hash value of the newly inserted key is equal to the position of the previously deleted key, but this is very unlikely (probability \( 1/M \)). Each rehash costs \( \Theta(M) \) calls to the insertion function of Lua. If we keep this alternation of delete-insert for \( M \) times, the cost is huge: for \( \Theta(M) \) operations, we obtain a quadratic cost \( \Theta(M^2) \).
One can legitimately object that this scenario is too unlikely to question the implementation. Users normally alternate a number of insertions with deletions in a more complex pattern, and there is no reason it happens exactly when the table is full. In the next section we present a simple, yet natural, probabilistic model for insertions and deletions.

3 Analysis Under a Probabilistic Model for Insertions/Deletions in Hashmaps

Recall first that we use the classical simple uniform hashing assumption [7] to model the behavior of the hash function, in both our worst case and probabilistic analysis. Hence, there is a layer of randomness within the algorithms, induced by this model, as it is classical in the field of randomized algorithm. In this section, we add probabilities on the inputs of the algorithms, which is the first step to perform the average case analysis. This randomness is not within the algorithm, it is used to model the input. Hence, in some sense, we are proposing in the following an average case analysis of a randomized algorithm: there are two layers of randomness, within the algorithm and on the inputs.

Usually, such analysis is not necessary, as hashmaps behave efficiently in the usual worst case amortized analysis of randomized algorithms (the randomness is just used to model the hash function). Which trivially implies the efficiency on average for any distribution on the inputs.

The situation is different here, as we established the inefficiency for the worst-case amortized setting, and want to improve on this result by proving that the algorithm still have an expected \( \Omega(\log n) \) running time by elementary operation in a reasonable probabilistic model on the inputs.

3.1 Description of the Probabilistic Model

For any fixed \( p \in (\frac{1}{2}, 1) \), our probabilistic model is the following:

- We perform \( T \) operations starting from an empty hashmap.
- Each operation is an insertion with probability \( p \) and a deletion with probability \( 1 - p \), independently of the previous operations. There is one exception when the hashmap is empty, in which case an insertion is always performed (with probability 1).
- An insertion consists in adding a new key. As we use the simple uniform hashing assumption, its associated hash value is a uniform random integer between 0 and \( M - 1 \), where \( M \) is the current size of the hashmap. We only consider keys that are not integer, so we do not trigger the use of the array-part: we only study the hashmap in this section.
- A deletion consists in deleting a key already present in the hashmap, taken uniformly at random amongst the keys currently present in the hash-map. Recall that in Lua, only the value associated to the key is deleted during this process.

When considering the \( t \)-th operation, for \( t \in \{0, \ldots, T\} \), we will often write that we are "at time \( t \)", and freely use the subscript \( t \) to design the current value of a parameter.
of the hashmap at time $t$. For instance, $M_t$ denote the size of the hashmap obtained after $t$ operations. Observe that in our settings, we have a probabilistic process, and $M_t$ is a random variable.

The parameter $p$ is chosen greater than $\frac{1}{2}$ so that the expected number of keys and the expected size of the hashmap increase linearly with the time. Taking $p < \frac{1}{2}$ yields non-interesting cases, and $p = \frac{1}{2}$ is a very specific, singular case, which could be studied using similar techniques.

3.2 Technical tools

In this section, we introduce the main tool we use for our proofs: Hoeffding's inequality for binomial random variables. For $\theta \in (0, 1)$ and $s \in \mathbb{Z}_{>0}$, we denote by $\text{Bin}(s, \theta)$ the binomial law defined for all $i \in \{0, \ldots, s\}$ by:

$$\Pr(\text{Bin}(s, \theta) = i) = \binom{s}{i} \theta^i (1 - \theta)^{s-i}.$$  

The following is a particular case of a classical result in Probability Theory, which gives exponential bounds for the probability of $\text{Bin}(s, \theta)$ deviating from its expected value $s\theta$. See for instance [8, Thm 2.8]

**Proposition 1** (Hoeffding's inequality). Let $s \in \mathbb{Z}_{>0}$ and $\theta \in (0, 1)$. Then for $t \in \mathbb{R}_{>0}$ we have the bounds:

$$\Pr(|\text{Bin}(s, \theta) - s\theta| \geq t) \leq 2 \exp\left(-\frac{2t^2}{s}\right),$$

and

$$\Pr(\text{Bin}(s, \theta) - s\theta \geq t) \leq \exp\left(-\frac{2t^2}{s}\right),$$

(1)

3.3 Main result and proof strategy

Let us state our main result, which establishes the inefficiency of the Lua's implementation of hashmap for our probabilistic scenario. We use the following definition: a property holds with exponentially (resp. super-polynomially) high probability in $T$ when the probability that it does not hold is less than $\exp(-cT)$ (resp. $\exp(-T^c)$) for some positive constant $c$ and $T$ sufficiently large.

**Theorem 2.** Let $p \in (\frac{1}{2}, 1)$. Starting from an empty hashmap, if $T$ operations of insertions with probability $p$ and deletions with probability $1-p$ are performed, then with super-polynomially high probability in $T$, the insertion function of Lua is called $\Omega(T \log T)$ times. As a consequence, the expected running time of the whole process is in $\Omega(T \log T)$.

As we will see, this is mostly because Lua spends a lot of time rehashing almost full hashmaps without increasing their size, impairing the efficiency of the data structure. This shows on our simulation, as depicted in Fig. 4.

Informally, the proof strategy of Theorem 2 is the following. With high probability, the number of keys in the hashmap after $T$ operation is linear in $T$. We can further
prove that in the process, for a well-chosen \( M = \Theta(T) \), with high probability, we allocate a hashmap of size \( M \) for the first time at some time \( t_M \) and of size \( 2M \) for the first time at some time \( t_{2M} \) with \( t_M \leq t_{2M} \leq T \).

Since keys are added one by one, at time \( t_M \) the hashmap contains exactly \( M/2 - 1 \) empty slots. We will establish that for some positive constant \( \gamma \), with high probability, if there are \( m \) empty slots after a rehash of size \( M \), then there are at least \( \gamma m \) empty slots at the next rehash (the hashmap’s size is therefore still \( M \) after the rehash). This holds with super-polynomially high probability for \( m \geq \sqrt{M} \). The intuitive reason behind this phenomenon is that during the process of adding \( m \) new keys, some are deleted and sufficiently many slots they occupied are not reused by the insertion process.

Hence, with super-polynomially high probability, at the rehash times starting from time \( t_M \), the hashmap contains \( M/2 - 1 \) empty slots, then at least \( \gamma(M/2 - 1) \) empty slots, then at least \( \gamma^2(M/2 - 1) \) empty slots, etc. So we need a logarithmic number of rehashes before the number of empty slots becomes smaller than \( \sqrt{M} \). Since each rehash costs \( \Theta(M) \) re-insertions, the total cost is \( \Omega(M \log M) \) time, which is \( \Omega(T \log T) \).

To formalize this proof sketch, we have to study the process and evaluate finely the error term, that is, the probability that the typical scenario described above does not hold.

### 3.4 Proof of the main result

This section is devoted to the proof of Theorem 2, which follows the general strategy described in the previous section. Let us first introduce some useful notations. At any time \( t \in \{0, \ldots, T\} \), the hashmap has size \( M_t \), and contains \( \varepsilon_t \) free cells, \( \delta_t \) deleted cells (keys with no values), and \( \nu_t \) used cells\(^6\) (keys with values). As every slot is in one of the three states, at any time \( t \), we have the identity \( \varepsilon_t + \delta_t + \nu_t = M_t \). Also, at time \( t = 0 \) the hashmap is empty, and thus \( M_0 = \varepsilon_0 = \delta_0 = \nu_0 = 0 \).

As an insertion is performed with probability \( p > \frac{1}{2} \) and a deletion with probability \( 1 - p < \frac{1}{2} \), the number of used keys in the table increases by \( 2p - 1 > 0 \) in expectation with each operation. This can be turned into a statement with exponentially high

---

\(^6\)We use the Greek letter \( \nu \) for consistency with the other notations in this section, \( \nu_t \) corresponds to \( N \) when no deletion occurred.
probability using Hoeffding’s inequality (Proposition 1), as stated in the following lemma.

**Lemma 3.** For any real constant $c$ such that $0 < c < 2p - 1$, if at a given time $s$ there are $\nu_s$ used slots in the hashmap, then after $t$ more operations $\nu_{s+t} \geq \nu_s + ct$ with exponentially high probability in $t$.

*Proof.* Let $I_t$ denote the number of insertions made during the first $t$ operations. If the hashmap is never empty, one has $I_t \sim \text{Bin}(t, p)$. Since operations that are not insertions are deletions in our model, the number of used slots at time $t$ is $2I_t - t$. By Hoeffding Inequality, Equation (2), this quantity is strongly concentrated around its expectation $2pt - t = (2p - 1)t$. We deal with a possibly empty hashmap by observing that the real $I_t$ is stochastically lower bounded by $\text{Bin}(t, p)$, yielding the result. \(\square\)

Set $\beta = \frac{2p-1}{M}$. As we want to establish an asymptotic result in $T$, we assume in the sequel that $T$ is sufficiently large to establish the required properties. A first consequence of Lemma 3 is that, with exponentially high probability in $T$, at some time the hashmap will have size $M$ with $\frac{2}{\beta}T < M \leq \beta T$, and some time after that the hashmap will reach size $2M$.

Let $t_h$ be any time when we just rehash into a hashmap of size $M$, where $M$ is the unique power of two such that $\frac{2}{\beta}T < M \leq \beta T$. It is not necessarily the first time we rehash into a hashmap of size $M$. As we just rehashed, we have $\delta_h = 0$, and $M = M_h = \nu_{t_h} + \varepsilon_{t_h}$. As long as the hashmap is not empty and that there is no rehash we have for $t \geq t_h$:

\[
\delta_{t+1} = \begin{cases} 
\delta_t - 1 & \text{with probability } \frac{\rho \delta_t}{M} \\
\delta_t & \text{with probability } p \left(1 - \frac{\delta_t}{M}\right) \\
\delta_t + 1 & \text{with probability } 1 - p
\end{cases} \quad \text{[insertion in a deleted slot],}
\]

\[
\delta_{t+1} = \begin{cases} 
\delta_t - 1 & \text{with probability } \frac{\rho \delta_t}{M} \\
\delta_t & \text{with probability } p \left(1 - \frac{\delta_t}{M}\right) \\
\delta_t + 1 & \text{with probability } 1 - p
\end{cases} \quad \text{[insertion in a free slot],} \quad (3)
\]

Intuitively, by comparing $\frac{\rho \delta_t}{M}$ and $1 - p$ in Equation (3), one can see that $\delta_t$ tends to increase when $\rho \delta_t/M < 1 - p$ and that it tends to decrease when $\rho \delta_t/M > 1 - p$. So the equilibrium point is at $\delta_t = \frac{1 - p}{\rho} M$. Fortunately for the analysis, we show that a rehash occurs before $\delta_t$ reaches this value, with exponentially high probability, so that at any step before the rehash $\delta_t$ is more likely to increase than to decrease. Let $t_h'$ denote the time of the next rehash. We can prove the following lemma.

**Lemma 4.** For any positive $d > \frac{1 - p}{\rho}$, with exponentially high probability in $\varepsilon_{t_h}$, we have (i) $t_h' \leq t_h + [d\varepsilon_{t_h}]$, and (ii) at any time $t$ between $t_h$ and $t_h'$ we have $\frac{\rho \delta_t}{M} \leq 1 - p$.

*Proof.* The first statement is a direct consequence of Lemma 3, taking $s = t_h$, $t = [d\varepsilon_{t_h}]$ and $c$ such that $\frac{1}{2} < c < 2p - 1$: with exponentially high probability in $\varepsilon_{t_h}$, there are more than $\nu_{t_h} + c[d\varepsilon_{t_h}] > \nu_{t_h} + \varepsilon_{t_h} = M$ used keys in the hashmap; they do not fit in a table of size $M$, hence a rehash already occurred and therefore $t_h' \leq t_h + [d\varepsilon_{t_h}]$.

For the second part we consider the processes $\bar{\delta}_t$ and $\bar{\nu}_t$ defined exactly as $\delta_t$ and $\nu_t$ except that there are no border condition (rehash or empty table), so that they can continue indefinitely. Observe that $\bar{\delta}_t$ and $\bar{\delta}_t$ (resp. $\bar{\nu}_t$ and $\bar{\nu}_t$) coincide as long as no rehash is triggered and as long as the hashmap is not empty. We want to bound from above the probability that $\bar{\delta}_t > \frac{1 - p}{\rho} M$ for some $t$ between $t_h$ and $t_h'$. By the union
bound, this probability is upper bounded by the probability that $\tilde{\delta}_t > \frac{1-p}{p} M$ for some $t$ between $t_h$ and $t_h + \lceil d\varepsilon_{t_h} \rceil$, plus some exponentially small probability for the cases where $\tilde{\delta}_t$ and $\hat{\delta}_t$ cease to coincide. We can therefore focus on finding an upper bound for the quantity $P$, defined as follows:

$$P := \sum_{t=t_h}^{t_h + \lceil d\varepsilon_{t_h} \rceil} \Pr\left(\hat{\delta}_t = \left\lfloor \frac{1-p}{p} M \right\rfloor + 1 \text{ and } t \leq t'_h\right),$$

using the fact that since $\hat{\delta}_t$ increases by at most one at each step, $\hat{\delta}_t > \frac{1-p}{p} M$ implies that $\tilde{\delta}_s = \left\lfloor \frac{1-p}{p} M \right\rfloor + 1$ for some $s \leq t$. Clearly, $P$ is an upper bound for the probability that (ii) does not hold for $\tilde{\delta}_t$.

Moreover, as we just rehashed at time $t_h$, we have $\tilde{\delta}_{t_h} = 0$. Hence, the probabilities for $t \leq t_h + \left\lfloor \frac{1-p}{p} M \right\rfloor$ in $P$ contribute to zero to the sum: not sufficiently many keys can have been inserted. Let $\kappa = \left\lfloor \frac{1-p}{p} M \right\rfloor + 1$, our expression for $P$ simplifies to

$$P = \sum_{t=t_h+\kappa}^{t_h + \lceil d\varepsilon_{t_h} \rceil} \Pr\left(\hat{\delta}_t = \kappa \text{ and } t \leq t'_h\right).$$

Let $f \in (0,1)$ be a constant that only depends on $p$ to be chosen later on. We split $P$ in two parts $P_1$ and $P_2$ the following way, with the convention that the second sum is 0 if $\lceil fM/p \rceil + 1 > \lceil d\varepsilon_{t_h} \rceil$:

$$P \leq \sum_{t=t_h+\kappa}^{t_h + \lceil fM/p \rceil} \Pr\left(\hat{\delta}_t = \kappa \text{ and } t \leq t'_h\right) + \sum_{t=t_h + \lceil fM/p \rceil + 1}^{t_h + \lceil d\varepsilon_{t_h} \rceil} \Pr\left(\hat{\delta}_t = \kappa \text{ and } t \leq t'_h\right).$$

A deletion is required for $\hat{\delta}_t$ to increases, hence $\hat{\delta}_t \leq D_{t-t_h}$ where $D_k \sim \text{Bin}(k,1-p)$. For any $t$ such that $t_h + \kappa \leq t \leq t_h + fM/p$ we have to perform at most $fM/p$ operations since time $t_h$, and we intuitively expect to have at most $\approx (1-p)fM/p$ deletions, which is smaller than $\kappa$ for sufficiently large $M$, as $f < 1$. Formally, for sufficiently large $T$, $M$ is sufficiently large so that $\kappa = \left\lfloor \frac{1-p}{p} M \right\rfloor + 1 > \frac{1-p}{p} \cdot \frac{1+f}{1-p} \cdot M$. Hence, for any $t$ such that $t_h + \kappa \leq t \leq t_h + fM/p$,

$$\Pr\left(\hat{\delta}_t = \kappa \text{ and } t \leq t'_h\right) \leq \Pr\left(\hat{\delta}_t = \kappa\right) \leq \Pr\left(\hat{\delta}_t > \frac{(1+f)(1-p)M}{2p}\right) \leq \Pr\left(\text{Bin}(t-t_h,1-p) > \frac{(1+f)(1-p)M}{2p}\right) \leq \Pr\left(\text{Bin}([fM/p],1-p) > \frac{(1+f)(1-p)M}{2p}\right).$$
This quantity is exponentially small in $M$ by Hoeffding inequality (Proposition 1). Hence, $P_1$ is exponentially small in $M$, hence in $\delta_{t_h}$, as a sum of a linear number of exponentially small quantities.

For $P_2$ we use the fact that if $\delta_t + \nu_t > M$ then a rehash was triggered and $t'_h > t$, which is therefore not possible. Since we are looking for a time $t$ where $\delta_t = \kappa$, we weaken the condition to obtain

$$P_2 \leq \sum_{t=t_h + \lfloor fM/p \rfloor + 1}^{t_h + \lfloor fM/p \rfloor} \Pr (\nu_t \leq M - \kappa) \leq \sum_{t=t_h + \lfloor fM/p \rfloor + 1}^{t_h + \lfloor fM/p \rfloor} \Pr \left( \nu_t \leq \frac{2p-1}{p} M \right).$$

Recall that $\nu_t = \nu'_t > M/2$. By Lemma 3, after more than $fM/p$ additional operations, we have $\nu_t \geq M/2 + c \cdot fM/p$ with exponentially high probability in $M$, for any $c \in (0, 2p - 1)$. But if $\nu_t \geq M/2 + c \cdot fM/p$ then

$$\nu_t - \frac{2p-1}{p} M \geq M \left( \frac{1}{2} + \frac{c \cdot f}{p} - \frac{2p-1}{p} \right).$$

As $c$ can be as close to $2p - 1$ as needed and $f$ as close to 1 as needed, we can choose them such that $cf - (2p - 1) > -p/4$. Therefore, for such a choice of $c$ and $f$, we have that $\Pr \left( \nu_t \leq \frac{2p-1}{p} M \right)$ is exponentially small in $M$. Summing a linear number of an exponentially small probabilities yields an exponentially small probability, hence $P_2$ is exponentially small in $M$, hence in $\varepsilon_{t_h}$, concluding the proof.

Let $t_0 = \lfloor \frac{1-p}{p} \varepsilon_{t_h} \rfloor < \varepsilon_{t_h}$. Between time $t_h + 1$ and $t_h + t_0$, there are not enough operations to trigger a rehash or to empty the hashmap. As $\delta_t$ increases by at most 1 at each operation, we have $\delta_t \leq \frac{1-p}{p} \varepsilon_{t_h}$ for any $t$ such that $t_h \leq t \leq t_h + t_0$. Since the hashmap is more than half filled just after a rehash, we have $\varepsilon_{t_h} < \frac{M}{2}$ and $\frac{\varepsilon_{t_h}^2}{23} \leq \frac{1-p}{p}$, for $t_h \leq t \leq t_h + t_0$. Hence, until time $t_h + t_0$, we can bound from below the process $\delta_t$ by a process that increases with probability $1 - p$, decreases with probability $\frac{1}{4}(1 - p)$ and does not change otherwise. This observation yields the following result.

**Lemma 5.** Let $t_0 = \lfloor \frac{1-p}{p} \varepsilon_{t_h} \rfloor$. With exponentially high probability in $t_0$, we have $\delta_{t_h + t_0} \geq \frac{(1-p)^2}{2p} \varepsilon_{t_h}$.

**Proof.** Using the observations made before stating the lemma, the process of Equation (3) is stochastically lower bounded by the process $\delta^-_t$ defined by $\delta^-_{t_h} = \delta_{t_0} = 0$ and for all $t \in \{t_h, \ldots, t_h + t_0\}$:

$$\delta^-_{t + 1} = \begin{cases} 
\delta^-_t - 1 & \text{with probability } \frac{1}{4}(1 - p) \\
\delta^-_t + 1 & \text{with probability } \frac{3}{4}(3p - 1) \\
\delta^-_t + 1 & \text{with probability } 1 - p.
\end{cases}$$

(4)

The random variable $\delta^-_{t + 1}$ is the difference of two (dependent) random variables, both following a binomial law. Since the expected increase at each operation is $\frac{1}{4}(1 - p)$ and there are $\lfloor \frac{1-p}{p} \varepsilon_{t_h} \rfloor$ operations, we get the result by applying Hoeffding Inequality.
Proposition 1: if the difference deviates from the expectation of more than $\alpha$ times the expectation, at least one of the two random variables deviates from its expectation of more than $\alpha/2$ times its expectation, which is exponentially unlikely.

From Lemma 4, with exponentially high probability the probability that the number of deleted elements decreases at any given step is smaller than the probability it increases. So $\delta_t$ remains linear in $\varepsilon_{t_h}$ after time $t_h + t_0$ and until a rehash occurs, as formalized in the following statement.

Lemma 6. For any $d > \frac{2}{p^2 - p}$, with exponentially high probability w.r.t. $\varepsilon_{t_h}$ there is a rehash at some time $t'_h \leq t_h + d\varepsilon_{t_h}$. Moreover, there exists some constant $\gamma \in (0, 1)$ such that, just before the rehash, $\delta_{t'_h - 1} \geq \gamma \varepsilon_{t_h}$.

Proof. The evolution of $\delta_t$ is lower bounded by a process that increases and decreases with the same probability, namely, $1 - p$. So after $t'_h - t_0 - t_h$ new operations made after time $t_0$, $\delta_t - \delta_{t_0 + t_h}$ is stochastically lower bounded by the difference of two dependent random variable that both follow a binomial distribution of parameters $t'_h - t_0 - t_h$ and $1 - p$. The expectation is 0, so we conclude using Hoeffding Inequality (Proposition 1) and take $\gamma = \left(1 - p\right)^2$ to get some linear room and have an exponentially small probability that the property does not hold.

We now have all the ingredients to prove our main theorem.

Proof of Theorem 2. with exponentially high probability in $T$, the hashmap reaches the capacity $M$. When it is the first time it reaches this size, and as we only insert elements one at a time, the hashmap contains $M/2 + 1$ used cells and $\varepsilon_0 := M/2 - 1$ free cells. By Lemma 6, after some time another rehash occurs, with at least $\gamma \varepsilon_0$ deleted cells just before rehashing. So the newly created hashmap has capacity $M$ (it is exponentially unlikely to decrease) and contains at least $\gamma \varepsilon_0$ empty cells. Then we apply Lemma 6 again, and there is another rehash into a hashmap of capacity $M$, with at least $\gamma^2 \varepsilon_0$ empty cells, etc. We continue while $\gamma^t \varepsilon_0 \geq \sqrt{T}$, that is, a logarithmic (in $T$) number of times. At each rehash we have to insert all the used keys, and there are a linear number of them, so it globally costs $\Omega(T \log T)$ calls to the insertion function: $\Omega(\log T)$ rehashes each costing $\Theta(T)$ calls. It is very likely that such a sequence of rehashes will occur, since at some point we reach a capacity of $2M$, with exponentially high probability, by Lemma 3. When we sum the probabilities of error using the union bound, we sum a logarithmic number of error terms, which are all in $O\left(\exp\left(-c\sqrt{T}\right)\right)$, so it is super-polynomially unlikely that this scenario does not happen. This concludes the proof.

4 Hybrid tables in Lua

Recall that when keys are integers, Lua stores their values in the array part of the hybrid table [5, 9]. The array-part corresponds to a range $[1, 2^n]$ of keys, or $\emptyset$ at the beginning. To avoid wasting memory, Lua makes sure that more than half of the keys are being used at any one time. When associating a value to a key into the table, if the key is an integer within the range of the array-part, the value is simply inserted there. Otherwise, the pair key/value is inserted into the hashmap as explained in
Section 2.1. If the insertion into the hashmap provokes a rehash, we first compute the largest $a' \geq 0$ such that $[1, 2^{a'}]$ contains at least $2^{a' - 1} + 1$ keys from the hybrid table. Then $A' = 2^{a'}$ will be the new size of the array-part, and the values of the keys within its range are placed there. The rest of the keys are placed in the hashmap, which has size $M = 2^m$, where $m$ is chosen so that the total number of elements it contains is between $2^{m-1}$ (strictly) and $2^m$. The insertions after the rehash are performed in the order of their position in the previous hashmap, each key going either to the hash-part or to the array-part if it is an integer smaller than or equal to $A'$. This process is exemplified in Figure 5.

4.1 Settings for the analysis

In all the following analysis of Lua hybrid tables, we consider that only insertions of pairs key/value are performed. This setting is sufficient to exhibit some unfortunate behavior in natural models, and it can only become worse if we also allow deletions. Rather than being interested in what happens between two rehashes, as in the previous section, we study what happens when a rehash occurs.

We consider a sequence of $n$ insertions $y = (y_1, \ldots, y_n)$ of integers (keys) into an initially empty Lua table. We write $t_0, t_1, t_2, \ldots, t_\ell$ for the sequence of rehash times, setting $t_0 = 0$, letting $t_i$ be the time of the $i$-th rehash. Let $\ell = \ell(y)$ be the total number of rehashes. More precisely, the insertion of $y_i$ induces the $i$-th rehash. Denote by $\beta_i$ the size of the hashmap after the $i$-th rehash.

In the process, the cost introduced by the insertions of elements in the array-part is small: as we only consider insertions in this section, the size of the array-part can only increase, and the amortized cost of an insertion in such a dynamic array is $O(1)$.

---

7This is done in linear time by counting the number of integer keys between $2^{l-1}$ and $2^l$ for each $l$, for $1 \leq 2^l \leq M$. 

Fig. 5: Example of a Lua table, with its hash-part and array-part. On the top, we have the initial state of the table, with the hash-part full. This is a valid configuration, it can be obtained in Lua from an empty table if we insert the keys in the order 1, 2, 4, 11, 9, 7, 5. Inserting $12 \mapsto f$ induces a rehash. Then the size of the array-part is recomputed, choosing the largest interval of keys $[1, 2^a]$ so that it will be more than half-full. The situation after the rehash can be seen at the bottom.
so the overall cost is $O(n)$. Hence, the cost of interest for us is the one induced by the rehashes, and we denote by $C$ the cost defined by $C := \sum \beta_i$. This exactly estimates the over-cost induced by rehashes, and is our main parameter of study in this section. It is an accurate estimation of the total number of calls to the Lua insertion function, up to a multiplicative constant.

### 4.2 Rehashing into the array-part

In this section, we show a simple example of how the hybrid mechanism may lead to super-linear costs $C = \Omega(n \log n)$. Moreover, we prove that $O(n \log n)$ is the worst case possible for a sequence $n$ insertions into an initially empty Lua table.

**Example 2.** Consider inserting $(-(2^k-1), -(2^k-2), \ldots, 0, 1, 2, \ldots, 2^k)$ into an empty Lua table. We claim that this induces exactly $k$ rehashes, in which we systematically reinsert $2^k$ entries into a renewed hash-part of size $2^k$.

This is proved as follows. Non-positive integers go into the hash-part of the table. Thus $(-(2^k-1), -(2^k-2), \ldots, 0)$ go into the hashmap, which is then full and of size $M = 2^k$. Then inserting 1, induces a rehash. However, the key 1 goes immediately to the array-part. Continuing, we rehash and double the size of the array-part when inserting 1, 2, 3, 5, 9, \ldots, in short for 1 and $(2^i + 1)_{i=0}^{k-1}$. The remaining positive keys go directly into a free spot of the array-part and do not induce a rehash.

It is therefore possible for a sequence of $n = 2^{k+1}$ integers to yield a cost $C = \Omega(n \log n)$. This is the worst possible case:

**Proposition 7.** When performing $n$ insertions into an initially empty Lua table, the insertion function is called $O(n \log n)$ times in the worst-case.

We recall that we denote by $t_0, t_1, t_2, \ldots, t_\ell$ the sequence of rehash times, setting $t_0 = 0$, letting $t_i$ be the time of the $i$-th rehash, and $\ell = \ell(y)$ be the total number of rehashes. Thus, the insertion of $y_i$ induces the $i$-th rehash. We denote by $\beta_i$ the size of the hashmap after the $i$-th rehash, and let $\alpha_j$ be the number of elements in the array-part right before performing the $j$-th rehash. We also denote by $\alpha_{\ell+1}$ the final number of elements in the array-part, as there is no $(\ell+1)$-th rehash, and we set $t_{\ell+1} := \alpha_{\ell+1} + \beta_{\ell+1}$.

**Remark 1.** With this notation, the rehash times satisfy $\alpha_{j+1} + \beta_{j+1} = t_{j+1}$, for all $j$ such that $0 \leq j \leq \ell$.

**Remark 2.** Note that when considering only insertions, at each rehash, either the size (capacity) of the array-part increases or of the hash-part increases. Indeed, when an element triggers a rehash, it cannot go into the array-part (else there would be no rehash) and the hash-part must be full. If the hash-part does not increase in size, then the array-part has to.

**Proof of Proposition 7.** We want to show that the cost $C := \sum_{i=0}^{\ell} \beta_i$ is $O(n \log n)$. For this, we distinguish the $\beta_i$’s that correspond to an increase of the array-part size from the other ones.

When the array-part increases in size, at the very least it doubles in size. Hence, the total number of rehashes in which the array-part increases is $O(\log n)$. The hash-part does not increase when the array-part does, as elements are added one at a time. Since $\beta_i \leq 2n$ for all $i$, the overall contribution of these $\beta_i$’s to $C$ is at most $O(n \log n)$. 

16
We now show that the remaining rehashes only contribute a total of $O(n)$. Consider the values of $i$ for which the hash-part doubles (with $1 \leq i \leq t$), that is, such that $\beta_i = 2\beta_{i-1}$. Then $\beta_i = 2(\beta_i - \beta_{i-1})$ we have $\sum_{i: \beta_i > \beta_{i-1}} \beta_i = 2 \sum_{j: \beta_{i+1} > \beta_j} (\beta_{j+1} - \beta_j)$. Observe that $\sum_{j: \beta_{j+1} > \beta_j} (\beta_{j+1} - \beta_j) = \sum_{j=0}^{t-1} (\beta_{j+1} - \beta_j) - \sum_{j: \beta_{j+1} \leq \beta_j} (\beta_{j+1} - \beta_j)$, and $\sum_{j=0}^{t-1} (\beta_{j+1} - \beta_j) = \beta_t \leq 2n$.

To conclude, we remark that $\sum_{j: \beta_{j+1} \leq \beta_j} |\beta_{j+1} - \beta_j| \leq n$. This is because the keys leaving the hash-part go into the array-part once and never come back. We make this connection formal. When rehashing we have $\alpha_{j+1} + \beta_j + 1 = t_j + 1$ (see Remark 1), thus $\beta_j - \beta_{j+1} = \alpha_{j+2} - \alpha_{j+1} + (t_{j+1} - t_{j+2}) < \alpha_{j+2} - \alpha_{j+1}$. Then, as $\alpha_j$ is non-decreasing, $|\beta_{j+1} - \beta_j| \leq \alpha_{j+2} - \alpha_{j+1}$ when $\beta_{j+1} \leq \beta_j$. Again using that $\alpha_j$ is non-decreasing we obtain

$$\sum_{j: \beta_{j+1} \leq \beta_j} |\beta_{j+1} - \beta_j| \leq \sum_{j: \beta_{j+1} \leq \beta_j} (\alpha_{j+2} - \alpha_{j+1}) \leq \sum_{j=0}^{t-1} (\alpha_{j+2} - \alpha_{j+1}) = \alpha_t + 1 - \alpha_1 \leq n.$$

Therefore $\sum_{i: \beta_i > \beta_{i-1}} \beta_i \leq 6n$, concluding the proof.

\begin{remark}
The proof of Proposition 7 shows that the rehashes in which the hash-part increases have a global contribution of only $O(n)$, while the others contribute for at most $O(n \cdot \log(1 + m))$, where $m$ is the maximum size of the array part.
\end{remark}

\begin{remark}
If some of the $n$ inserted keys are not positive integers, we can refine the result. Let $n'$ be the total number of positive integer keys. Then the worst case for $C$ is $O(n + n \log(1 + n'))$, as long as only insertions are performed.
\end{remark}

### 4.3 Inserting permutations in Lua Tables

In Example 2 we showed a sequence of $n$ insertions of integers yielding a cost $C = \Theta(n \log n)$. Some of the keys were negative integers, so they could only ever be stored in the hash-part. In this subsection, we show that this worst case is still attainable on a very natural setting involving only positive integers: the sequence $\mathbf{y}$ is a permutation of $[n] := \{1, \ldots, n\}$. This setting, a priori, gives the array-part the best possible chance of being exploited, while not repeating keys. We present both the worst-case (this subsection) and the case of a random permutation (Subsection 4.4). Though described in terms of permutations, these settings apply whenever the keys are consecutive integers that do not necessarily appear in increasing order. It is the case for instance during the marking of the transversal of a graph of vertex set $[n]$.

\begin{proposition}
Inserting $n$ elements given by the order of a permutation $\pi$ of $[n]$ requires $\Omega(n \log n)$ calls to the insertion function of Lua in worst-case.
\end{proposition}

\begin{proof}
Consider first $n = 3 \times 2^k$ for some $k > 0$. Define the permutation

$$\pi = \begin{pmatrix}
1 & 2 & \cdots & 3^k \\
2 & 2^2 & \cdots & 3 \cdot 2^k \\
\vdots & \vdots & \ddots & \vdots \\
2^k & 2^k & \cdots & 2^k
\end{pmatrix}.
$$

Notice that $2 \cdot 2^k + 1, \ldots, 3 \cdot 2^k$ cannot be on the array-part, unless the keys of the array-part contain the range $[1, \ldots, 4 \cdot 2^k]$. For this to happen more than half of the
elements of that range must have been inserted, and this only happens after a time \( t \) such that \( \pi(t) = 2^k \), so at time \( t = 2^{k+1} \). Thus, after inserting \( \pi(1), \ldots, \pi(2^k) \), the hash-part has size \( M = 2^k \) and it is full. Inserting \( \pi(2^k + 1) = 1 \) induces a rehash, but 1 goes into the array-part. Similarly, for \( \pi(2^k + 2) \), and more generally the same is true for the insertion of \( \pi(2^k + 2^j + 1) \) as long as \( j < k \). The rest are inserted directly into the array-part. For example, the insertion of 4 = \( \pi(3) \) does not produce a rehash as 4 goes into the array-part of range \([1, 4]\). Each of the \( k \) rehashes we have just described costs at least \( 2^k \) because all the elements \( \pi(1), \ldots, \pi(2^k) \) are still in the hash-part. Hence, we obtain a cost that is \( \Omega(2^k \times k) = \Omega(n \log n) \).

For general \( n \), we pick the largest integer \( k \) such that \( m = 3 \times 2^k \leq n \), and complete the permutation \( \pi \) above by setting \( \pi(i) = i \) for \( i \geq m + 1 \).

### 4.4 Average case for insertion of permutations

The average case for permutations is almost linear, but not because of the wrong reasons. Our main result of this section, Theorem 9, tells us that, essentially for any \textit{super-linear} function \( h(n) \), the cost \( C \) of a random permutation is \( \mathcal{O}(h(n)) \) with probability tending to 1. There is a caveat with the choice of \( n \): the theorem does not apply if \( n \to \infty \) approximates powers of two from above. Theorem 9 enforces this restriction by introducing subsets \( \mathbb{N}_b := \bigcup_{j=0}^{\infty} \{ n : 2^jb < n \leq 2^{j+1} \} \), parameterized by \( b \in (1, 2) \). This restriction is technical, but not restrictive for our purpose as it still applies to most natural choices of \( n \), e.g., powers of two \( n = 2^k \). We suspect that such a restriction might be necessary; we are not sure if the conclusions apply to the sub-sequence \( n = 2^k + 1 \), and the arguments become too involved for the main topic of this article to justify an in-depth study here.

Even though inserting the keys of \([n]\) in a permuted order is essentially done almost linear time, we prove that the array-part is not really exploited as it should. For a uniform random permutation, with high probability, the corresponding Lua table does not have an array-part until the very end. This can be seen in Lemma 10 below. Thus, in this scenario, we do not really take advantage of the hybrid data structure.

**Theorem 9.** Let \( g : \mathbb{N} \to \mathbb{R}_{>0} \) be an arbitrary function satisfying \( g(n) \to \infty \), with \( g(n) = o(n) \) as \( n \to \infty \). Fix an arbitrary \( b \in (1, 2) \) and let \( n \to \infty \) over \( \mathbb{N}_b \). Then, with probability tending to 1 as \( n \to \infty \), the number of calls to the insertion function, starting from an empty table, to build the Lua table for a uniform random permutation of \([n]\) is \( \mathcal{O}(ng(n)) \).

Fixing \( S \subseteq [n] \), let \( S_t = S_t(\pi) := \{ \pi(k) : k \leq t \} \cap S \), in other words, the set of keys from \( S \) revealed up to time \( t \). The proof of Theorem 9 is based on a key lemma and its corollary. Consider a time \( t \leq cn \) with \( c < 1/2 \). We show in Lemma 10 that, if \( |S| \) is large, then time \( t \) will not be enough to have revealed half of the elements of \( S \), i.e., \( |S_t| > |S|/2 \) is unlikely. Lemma 10 and its corollary, Corollary 11, apply to all \( n \in \mathbb{N} \). In particular, the latter implies that array-part is not really used at any time \( t \leq cn \).

**Lemma 10.** Let \( \theta = \frac{c}{n} < \frac{1}{2} \) and \( s = |S| \), then

\[
\Pr (|S_t| > s/2) \leq \frac{\theta}{(1/2 - \theta)^2} s^{-1}.
\]
Corollary 11. Fix \( c < 1/2 \) and let \( g(n) \to \infty \). With probability tending to one, none of the sets \( A_j = [1 \ldots 2^j] \), for \( 2^j \geq g(n) \), is half-full at time \( t \leq cn \).

Proof. The probability that \( A_j \) is more than half-full is at most \( c (\frac{1}{2} - c)^{-2} 2^{-j} \) by Lemma 10. Thus, by the union bound, the probability that at least one of the \( A_j \)'s is more than half-full is bounded by \( \sum_{j \geq \log_2 g(n)} c (\frac{1}{2} - c)^{-2} 2^{-j} = c (\frac{1}{2} - c)^{-2} 2^{-\lceil \log_2 g(n) \rceil} \). As \( g(n) \to \infty \), the bound tends to 0. \( \square \)

Remark 5. One can directly obtain \( \Pr(\{|S_k| = i\}) = \binom{n}{i} \binom{n-|S|}{i} / \binom{n}{i} \) using a simple counting argument. This is a hypergeometric distribution. More precise tail inequalities are known for this distribution [10]. For our purposes, the bound in Lemma 10 is enough, as we only consider \( S = [1,2^j] \) for \( j \geq 1 \).

4.4.1 Proof of Lemma 10

We first recall the notation we use for the proof. Given a time \( t \leq n \), let \( \theta = \frac{t}{n} \). Fixing \( S \subseteq [n] \), for a permutation \( \pi \) of \( \{1, \ldots, n\} \), we write \( S_t = S_t(\pi) := \{\pi(k) : k \leq t\} \cap S \), \( s = |S| \) and \( s_t = s_t(\pi) := |S_t(\pi)| \).

We start with the proof of Lemma 10. For the proof, we apply Chebyshev’s inequality. Write
\[
s_t(\pi) = \sum_{i \in S} X_i^{(t)}(\pi)
\]
where \( X_i^{(t)}(\pi) = 1 \) if \( i \) belongs to \( \{\pi(1), \ldots, \pi(t)\} \) and 0 otherwise. For simplicity, we write \( X_i = X_i^{(t)} \) omitting the time \( t \), as it will be fixed.

Lemma 12. For \( A \subseteq \{1, \ldots, n\} \), let \( X_A = \prod_{i \in A} X_i \). Then \( \mathbb{E}[X_A] = \binom{n-|A|}{t-|A|} / \binom{n}{t} \).

Moreover, the inequality \( \mathbb{E}[X_A] \leq \left( \frac{1}{2} \right)^{|A|} \) holds.

Proof. The variable \( X_A \) is 1 if \( S \subseteq \{\pi(1), \ldots, \pi(t)\} \) and 0 otherwise. Disregarding the order of \( \pi(1), \ldots, \pi(t) \), as it does not affect the sets, the number of sets of size \( t \) having this property is \( \binom{n-|A|}{t-|A|} \) as we must enforce that the elements of \( A \) are present. Hence, the first equality.

For the inequality, we note that if \( |A| \leq t \), \( \binom{n-|A|}{t-|A|} / \binom{n}{t} = \frac{t-|A|}{n-|A|} \), and \( \frac{t-1}{n-1} \leq \frac{t}{n} \) for \( i < n \). If \( |A| > t \) the inequality is trivial. \( \square \)

By symmetry and linearity of the expectation, \( \mathbb{E}[s_t] = s \times \mathbb{E}[X_1] = s \times \binom{n-1}{t-1} / \binom{n}{t} = s \times \theta^t \), yielding the expected value of \( s_t \).

For the second moment of \( s_t \), we observe that classically
\[
\mathbb{E}[s_t^2] = \sum_{i \in S} \mathbb{E}[X_i^2] + 2 \sum_{i,j \in S, i < j} \mathbb{E}[X_i X_j] = \mathbb{E}[|S_t|] + 2 \sum_{i,j \in S, i < j} \mathbb{E}[X_i X_j] .
\]

By Lemma 12, \( \mathbb{E}[X_i X_j] = \binom{n-2}{t-2} / \binom{n}{t} \leq \theta^2 \) where \( \theta = t/n \). Thus, by symmetry again and \( \text{Var}(s_t) = \mathbb{E}[s_t^2] - \mathbb{E}[s_t]^2 \), we obtain
\[
\mathbb{E}[s_t^2] \leq s \theta + s^2 \theta^2, \quad \text{Var}(s_t) \leq s \theta .
\]
Hence, by Chebyshev’s inequality, we have proved Lemma 10.

### 4.4.2 Proof of Theorem 9

We are now ready to prove the theorem. Recall that when we say at time \( t \), we mean after the \( t \)-th insertion. The notation \( a(n) \ll b(n) \) and \( b(n) \gg a(n) \) means \( a(n) < b(n) \) for all large enough \( n \).

The following simple remark will be important in the proof.

**Remark 6.** Suppose we only insert elements into an initially empty hybrid table. If, after the \( t_0 \)-th insertion, the hash-part has capacity \( M = 2^k \) and the array-part \( A = 2^a \), then \( t_0 \leq M + A \) and the next rehash is triggered by an insertion between \( t = M + 1 \) and \( t = M + A + 1 \) (inclusive).

**Proof of Theorem 9.** Write \( 2^k b < n \leq 2^{k+1} \), where \( b \in (1, 2) \).

The proof consists of three steps:

1. By Corollary 11, at time \( t = 2g(n) + 2^{k-1} \) the array-part has size less than \( g(n) \), with probability tending to 1. Indeed, if \( b' \in (1, b) \), since \( g(n) = o(n) \) we have \( t = 2g(n) + 2^{k-1} \ll n/(2b') \), and we apply Corollary 11 with \( c = 1/(2b') \). Thus, the hash-part must have size \( M = 2^k \), because \( 2^{k-1} + g(n) \ll t \) and so \( M \leq 2^{k-1} \) is impossible by Remark 6, while \( M = 2^{k+1} \) would be absurd as \( M/2 = 2^k \gg t \), by our hypothesis \( g(n) = o(n) \), and the hash-part would not be half-full.

2. By Remark 6, the next rehash occurs at \( t > 2^k \). Then the array-part takes up the whole range \([1, 2^{k+1}]\). Thus, the total number of rehashes in which the array-part can increase is at most \( 2 + \log_2 g(n) = O(g(n)) \).

3. Then we conclude by Remark 3: the rehashes in which the array-part increases contribute at most \( O(n g(n)) \) to the cost \( C \), while the rest of the rehashes contribute only \( O(n) \).

### 5 Conclusions and final remarks

The only data-structuring mechanism in Lua are tables, so it is of the upmost importance that they are extremely efficient in time and space usage. Lua hybrid tables work very well in many practical use-cases, for example, to create an array of \( n \) elements filling \( A[1], \ldots, A[n] \) sequentially, or to build a dictionary in which we alternate insertions and searches but have no deletions, or when we fill a table and then process and remove one by one its elements. But there are some situations which may also arise in practice rather naturally where there are noticeable inefficiencies or suboptimal performance of the Lua hybrid tables, as our theoretical analysis has revealed. Fig.6 illustrates that this unwanted behavior shows in practice on our simulations.

In Section 4, we have shown that the hybrid structure introduces similar issues (see Prop. 8) when considering only insertions. The effect is more limited than in the case of both insertions and deletions (see Prop. 7 and Thm. 9), yet the array-part might not be exploited to reduce memory consumption as much as would be expected.

These problems seem to have easy fixes, the most immediate one being to allow more room when rehashing, to avoid restarting with a new full or almost full table. A second solution would be to implement true deletions instead of just marking the
Fig. 6: Experimental plots for time against number of operations. The plot on the right shows the average microseconds/operation made on a personal computer. Blue plots correspond to a probability of insertion $p = 0.9$, while red plots correspond to $p = 0.75$. The solid plots correspond to the original Lua code, while the dashed plots correspond to the modified Lua, in order to ensure free room when rehashing as described below (our fix proposal). The Lua code for the experiments can be found at https://gist.github.com/PRotondo/59036a5a3dd53b30d9555a3a748cb7c.

deleted elements by setting their values to nil. Both solutions are very classical and details can be found, for instance, in [7].

We conclude by briefly considering the first fix. In practice, it is enough to change just one line of code. The function setnodesize in ltable.c creates a new hash-part having at least size elements, which is passed on as an argument. There, the new exponent $m$ of the size of the hash-part, called $lsize$ in the C code, is chosen to be $\lceil \log_2 \text{size} \rceil$, in C code luaOceillog2(size). To enforce extra space, it suffices to set $lsize$ to $\text{luaOceillog2}(\text{size}+(\text{size}>>2))$. Increasing the capacity of the new hash-part this way ensures that at least 20% of its cells are free after reinserting the old elements. The effects of this new rehash policy can be seen in Figure 6. This fix guarantees amortized constant time per operation: indeed, between two consecutive rehashes we must perform at least $M/5$ insertions$^9$.

m = 1<<24
for i =(-m+1),m do
  tab[i] = 1
end

Fig. 7: Lua code for the worst-case (insertions) in Example 2.

We also considered the effect of this modification on the worst-case scenarios described previously. For the situation in Example 1 with $M = 2^{15}$, our simulations in a personal computer$^{10}$ yield a time of approximately 32 seconds for the original Lua, while the modified Lua takes just 16 milliseconds. Considering only insertions, Lua takes 21 seconds for Example 2 with $M = 2^{24}$, whose five-line code can be seen in Figure 7 on the left, against 3.5 seconds for the modified version.

$^8$More precisely, as $\text{size}>>2$ corresponds to $\lfloor \text{size}/4 \rfloor$ rather than $\lceil \text{size}/4 \rceil$, the number of free cells is guaranteed to be at least 20% of the total, minus one extra cell due to the floor function.

$^9$Not $M/5 - 1$ because we need to perform one extra insertion to trigger the rehash.

$^{10}$Processor AMD Ryzen 5 PRO 2500U, 16 GB RAM, running LinuxMint. The Lua code for the tests can be found in https://gist.github.com/PRotondo/275a1292c0d04cc8064211f2e600dc1
References

[1] Auger, N., Jugé, V., Nicaud, C., Pivoteau, C.: On the Worst-Case Complexity of TimSort. In: Azar, Y., Bast, H., Herman, G. (eds.) 26th Annual European Symposium on Algorithms (ESA 2018). Leibniz International Proceedings in Informatics (LIPIcs), vol. 112, pp. 4–1413. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2018). https://doi.org/10.4230/LIPIcs.ESA.2018.4 . http://drops.dagstuhl.de/opus/volltexte/2018/9467

[2] Buss, S., Knop, A.: Strategies for stable merge sorting. In: Chan, T.M. (ed.) Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019, pp. 1272–1290. SIAM, Philadelphia (2019). https://doi.org/10.1137/1.9781611975482.78 . https://doi.org/10.1137/1.9781611975482.78

[3] De Gouw, S., Rot, J., Boer, F.S., Bubel, R., Hähnle, R.: OpenJDK’s Java.utils.Collection.sort() is broken: The good, the bad and the worst case. In: Int. Conf. on Computer Aided Verification, Heidelberg, pp. 273–289 (2015). Springer

[4] Ierusalimschy, R., Figueiredo, L.H., Filho, W.C.: Lua—an extensible extension language. Software: Practice & Experience 26, 635–652 (1996)

[5] Ierusalimschy, R., Figueiredo, L.H., Filho, W.C.: The implementation of Lua 5.0. J. Univers. Comput. Sci. 11(7), 1159–1176 (2005)

[6] Martínez, C., Nicaud, C., Rotondo, P.: A probabilistic model revealing shortcomings in Lua’s hybrid tables. In: Zhang, Y., Möhrig, R., Miao, D. (eds.) Proc. 28th International Computing and Combinatorics Conference (COCOON). Lect. Notes in Comp. Sci., vol. 13595, pp. 381–393. Springer, Heidelberg (2022)

[7] Knuth, D.E.: The Art of Computer Programming: Sorting and Searching vol. 3, 2nd edn. Addison-Wesley, Boston (1998)

[8] Boucheron, S., Lugosi, G., Massart, P.: Concentration Inequalities. A Nonasymptotic Theory of Independence, p. 481. Oxford University Press, ??? (2013)

[9] Ierusalimschy, R., Figueiredo, L.H., Filho, W.C.: Lua Programming Gems. Lua.Org, Rio de Janeiro (2008)

[10] Chvátal, V.: The tail of the hypergeometric distribution. Discrete Mathematics 25(3), 285–287 (1979) https://doi.org/10.1016/0012-365X(79)90084-0