A Probabilistic Model Revealing Shortcomings in Lua's Hybrid Tables

Conrado Martínez, Cyril Nicaud, Pablo Rotondo

A Probabilistic Model Revealing Shortcomings in Lua's Hybrid Tables*

Conrado Martínez[1][0000−0003−1302−9067], Cyril Nicaud[2][0000−0002−8770−0119], and Pablo Rotondo[2][0000−0001−8777−1278]

[1] Dept. of Computer Science. Universitat Politècnica de Catalunya. conrado@cs.upc.edu

[2] Laboratoire d'Informatique Gaspard-Monge (LIGM) UMR 8049. Université Gustave Eiffel. {cyril.nicaud,pablo.rotondo}@u-pem.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, is an associative array called a table. With Lua 5.0, the reference implementation of Lua introduced *hybrid tables* to implement tables using both a hash table and a dynamically growing array combined together: the values associated with integer keys are stored in the array part, when suitable. All this is transparent to the user, which has 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, instead of linear in T.

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.[3] This new, elegant and powerful sorting algorithm was quickly adopted by several other languages, including Java.[4] After a decade of existence, computer scientists started analyzing its efficiency,

* The work of the first author has been supported by funds from project MOTION (Project PID2020-112581GB-C21) of the Spanish Ministry of Science & Innovation MCIN/AEI/10.13039/501100011033.

[3] https://github.com/python/cpython/blob/main/Objects/listsort.txt

[4] https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html

helped fixing some issues and confirmed its excellent performances at a theoretical level [1,2,3]. 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.

Lua[5] is a scripting programming language [4] created in the early nineties 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 was not present then the instruction inserts the pair $x \mapsto y$ 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 the used to obtain the sought value or to assign a new value. To delete a pair $x \mapsto y$ 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 returned back to the memory heap for future use.

Until Lua 4.0, tables were implemented strictly as hash tables: all pairs $x \mapsto y$ were explicitly stored in a single hash table, 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 (=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 $\geq n/2$ 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 hash table as usual. The new Lua (hybrid) tables thus have two parts called the hash-part or *hashmap*[6] and the array-part or *array*.

Main contributions & Plan of the paper. In this paper we provide a theoretical analysis covering some aspects of the performance of Lua hybrid tables[7]. First we consider the hashmaps (Section 2). We review how Lua hashmaps work, and discuss their performance in the absence of deletions. Our main result is in Subsection 2.4 where we develop a simple probabilistic model which applies in many reasonable practical scenarios and show that Lua might significantly deviate from the desirable expected constant amortized time per insertion (Theorem 1). It is important to remark that in our analysis we are not assuming

[5] From the Portuguese "*lua*", meaning moon.

[6] We will sometimes also refer to the hashmap as the *hash table*.

[7] All detailed descriptions in the remaining of the paper and our analysis refer to version 5.4.4 of Lua (the most recent).

worst-case hypothesis such as a bad hash function or that we have to perform a sequence of operations designed by an adversary.

After that, in Section 3, we investigate the performance of the hybrid table as a whole, this time focusing in 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 2). 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 for most part of the time (Theorem 2), and thus the advantages of the hybrid scheme can become blurred. Finally, in the conclusions (Section 4), we present some experiments and propose some easy and classical solutions to avoid the drawback of the hybrid table's implementation.

2 Hashmaps in Lua

2.1 Description of the hashmap algorithms

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 element (**nil** if there is no successor in the search sequence). 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.

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` $\leftarrow M - 1$, and it can only decrease.

Search. The search of 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) or the end of the list (failure).

Deletion. To delete a key x, its position in the table, if it exists, is found as for the search and then its associated value is set to **nil**. The `next` field remains unaltered to maintain the chaining.

Insertion. If one wants to set $x \mapsto y$ 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. Otherwise, there is a collision with another pair $x' \mapsto y'$ at the position i. We distinguish two cases. If x' is at its main position, a free slot is found for $x \mapsto y$ and the chaining is updated so that $x \mapsto y$ is on the second position in 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 $x \mapsto y$ is set at index i, with a **nil** `next`-link.

Looking for a free slot is accomplished by scanning the table from right to left starting at position `last_free` until a free slot. Importantly, *deleted slot are ignored during this process* in Lua (to avoid problems with the chainings). 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 $n + 1 \leq 2^m$. All pairs keys↦values and the pair $x \mapsto y$ are then inserted into the new map.

So in any case, just after its insertion, $x \mapsto y$ is at the first or second place in its chain. Examples of insertions in a Lua hashmap are depicted in Fig. 1.

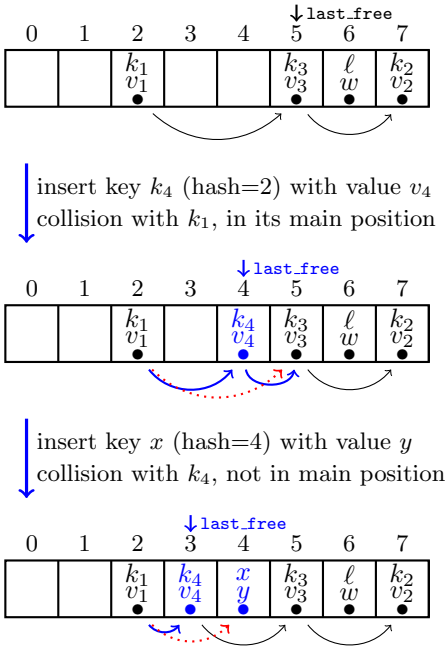

Fig. 1: If there is no deleted value and if there is a collision with a key that is in its main position, the new inserted element is placed in a free spot and is at the second position in its chaining. If the colliding key is not at its main position, the new element is put there, and the colliding element is placed in the free spot.

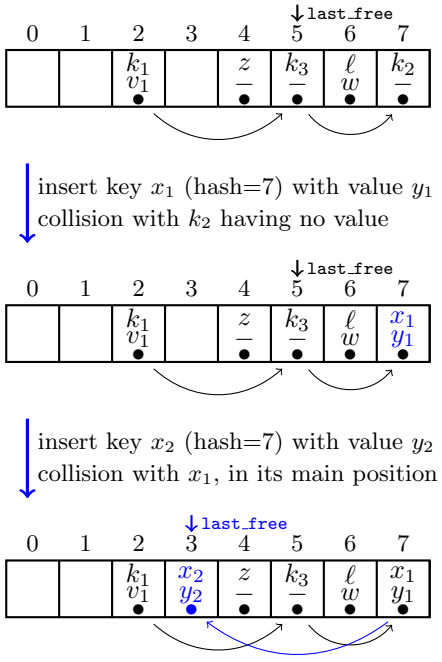

Fig. 2: If the main position of the newly inserted key has a deleted value, it is just put there. If it is used, then the insertion proceeds as previously, considering the deleted spot as occupied for the search of a free spot. Observe that at the end, the hash values of 2 and 7 share the same chain,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

The insertion algorithm and other auxiliary functions in pseudo-code can be found in Appendix **??**.

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 $\{0, \dots, M-1\}$. 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] (it is called *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 $\alpha = N/M \leq 1$. 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.

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 $\mathcal{O}(1)$.

2.3 General Analysis: an Unlikely Worst-Case Scenario

When we insert N elements into a Lua table, we have seen that the total expected amortized cost is $\mathcal{O}(N)$. We show that the situation changes significantly when we consider deletions. 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. Keys are not integers, only the hash-part is studied in this section.

Example 1 (Alternation of deletion-insertion on 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. 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. In the next section we present a simple, yet natural, probabilistic model for insertions-deletions.

2.4 General Analysis: An Average-Case Scenario

In this section we consider a simple probabilistic model in which we start with an empty hashmap, and then perform a large number T of operations. Each

operation consists in inserting a new key to the hash table, with probability $p \in (\frac{1}{2}, 1)$, or deleting the value associated with a key in the table with probability $1 - p$ (unless the hash table contains no value, in which case only an insertion is possible). The parameter p is fixed, and chosen to be greater than $\frac{1}{2}$ so that the hash table tends to grow linearly in size with T. When a key is deleted, it is taken uniformly at random amongst the assigned keys currently in the hash table. We choose this model as it is a simple yet natural way to describe a mix of insertions and deletions in a data structure. Keys are not integers, only the hash-part is studied in this section.

In the following, we say that 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. $\exp(-cT^d)$) for some positive constants c (and d).

Our main result of this section is a negative result, emphasizing that Lua's hash table behaves badly for this simple and natural probabilistic model.

Theorem 1. *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 time complexity of the process is in $\Omega(T \log T)$.*

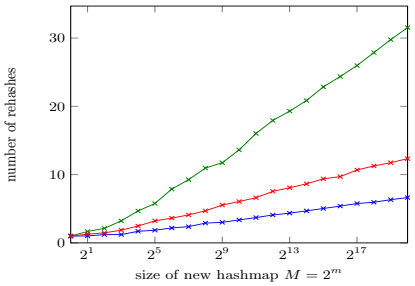

Fig. 3: Total number of rehashes (y-axis) producing a hashmap of a given size M (x-axis) during a run of $T = 10^7$ operations. The plot is logarithmic in M(x-axis). Plots correspond to $p = 0.6$ (green), $p = 0.75$ (red) and $p = 0.9$ (blue). Each point is the average result of 100 simulations directly using Lua.

See Appendix ?? for the proofs of the lemmas.

In our proof, we establish that Lua spends a lot of time rehashing almost full hashmaps without increasing the size, impairing the efficiency of the data structure. This also shows on our simulation, see Fig. 3.

Before going on with the proof of Theorem 1, let us clarify our notations for this section: 1) T denotes the number of operations performed in total; at a given moment $t \in \{0, \dots, T\}$ operations have been performed; t is often referred to as *time*; 2) at any time t, the hash table has size M_t, and contains ε_t free cells, δ_t deleted cells (keys with no values), and ν_t used cells[8] (keys with values). Notice that, at any time t, $\varepsilon_t + \delta_t + \nu_t = M_t$: every cell is in one of the three states; 3) at time $t = 0$, the hash table is empty: $M_0 = \varepsilon_0 = \delta_0 = \nu_0 = 0$.

First observe that since an insertion is performed with probability p and a deletion with probability $1 - p$, the number of used keys in the table increases by $2p - 1 > 0$ in expectation with each operation. This can be turned into

[8] We use the Greek letter ν for consistency with the other notation in this section, ν_t corresponds to N when no deletions occur.

a statement with exponentially high probability using classical concentration inequalities, yielding the following lemma.

Lemma 1. *Let c be a constant with $0 < c < 2p-1$. If at a given moment s there are ν_s used cells in the hash table, then after t more operations, the hashmap contains more than $\nu_s + ct$ keys with exponentially high probability in t.*

Set $\beta = \frac{2p-1}{3}$ and assume that T is sufficiently large. A first consequence of Lemma 1 is that, with exponentially high probability in T, at some moment the hashmap will have size M with $\frac{\beta}{2}T < M \leq \beta T$, and some time after the hashmap will reach size $2M$. We will establish that between these two moments, the insert function is called $\Omega(T \log T)$ times.

Let t_h be any time in which we just rehash into a hashmap of size M, where M is the unique power of two such that $\frac{\beta}{2}T < M \leq \beta T$. It is not necessarily the first time we rehash into a hash table of size M. As we just rehashed, we have $\delta_{t_h} = 0$, and $M = \nu_{t_h} + \varepsilon_{t_h}$. As long as the hash table is not empty (this would force an insertion; but it is exponentially unlikely that we reach an empty table) and that there is no rehash (a rehash will happen at some point with exponentially high probability), we have for $t \geq t_h$:

$$\delta_{t+1} = \begin{cases} \delta_t - 1 & \text{with probability } \frac{p\delta_t}{M} & [\textit{insertion in a deleted key}], \\ \delta_t & \text{with probability } p\left(1 - \frac{\delta_t}{M}\right) & [\textit{insertion in a free cell}], \\ \delta_t + 1 & \text{with probability } 1-p & [\textit{deletion}]. \end{cases} \quad (1)$$

In Eq. (1), we see that δ_t tends to increase when $p\delta_t/M < 1-p$ and it tends to decrease when $p\delta_t/M > 1-p$. So the equilibrium point is at $\delta_t \approx \frac{1-p}{p}M$. Fortunately for the analysis, we show that a rehash occurs before δ_t reaches this value, with exponentially high probability. Let τ denote the time of the next rehash. We can prove the following lemma.

Lemma 2. *For any positive $d > \frac{1}{2p-1}$, with exponentially high probability in ε_{t_h}, we have $\tau \leq t_h + d\varepsilon_{t_h}$ and at any time t between t_h and τ we have $\frac{p\delta_t}{M} \leq 1-p$.*

Let $t_0 = \lfloor \frac{1-p}{p}\varepsilon_{t_h} \rfloor < \varepsilon_{t_h}$. It is easy to check that no rehash could have occurred and that the hashtable was never empty when we reach time $t_h + t_0$. We are now interested in estimating the value of $\delta_{t_h+t_0}$. As δ_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 hash table is more than half filled just after a rehash, we have $\varepsilon_{t_h} < \frac{M}{2}$ and $\frac{p\delta_t}{M} \leq \frac{1-p}{2}$, for $t_h \leq t \leq t_h + t_0$. Hence, until time $t_h + t_0$, we can bound from below the process δ_t by a process that increases with probability $1-p$, decreases with probability $\frac{1}{2}(1-p)$ and does not change otherwise.

Lemma 3. *At time $t_h + t_0$, with $t_0 = \lfloor \frac{1-p}{p}\varepsilon_{t_h} \rfloor$, with exponentially high probability in t_0 we have $\delta_{t_h+t_0} \geq \frac{(1-p)^2}{3p}\varepsilon_{t_h}$.*

Thanks to Lemma 2, we know that, 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 δ_t remains linear in ε_{t_h} after time $t_h + t_0$ and until a rehash occurs, as formalized in the following statement.

Lemma 4. *With exponentially high probability in ε_{t_h} there is a rehash at some time $\tau \leq t_h + d\varepsilon_{t_h}$, for any $d > \frac{1}{2p-1}$. Moreover, there exists some constant $\gamma \in (0,1)$ such that, just before the rehash, $\delta_{\tau-1} \geq \gamma\, \varepsilon_{t_h}$.*

We can now conclude the proof of Theorem 1: 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 hash table contains $M/2+1$ used cells and $\varepsilon_0 := M/2-1$ free cells. By Lemma 4, after some time another rehash occurs, with at least $\gamma\, \varepsilon_0$ deleted cells just before rehashing. So the newly created hash table has capacity M (it is exponentially unlikely to decrease) and contains at least $\gamma\, \varepsilon_0$ empty cells. Then we apply Lemma 4 again, and there is another rehash into a hash table of capacity M, with at least $\gamma^2\, \varepsilon_0$ empty cells, etc. We continue while $\gamma^i\, \varepsilon_0 \geq \sqrt{T}$, 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 1. To conclude the proof, we just have to observe that when we sum the probabilities of error (by the union bound), we sum a logarithmic number of error terms, which are all in $\mathcal{O}(\exp(-c\sqrt{T}))$, so it is super-polynomially unlikely it does not happen.

3 Hybrid tables in Lua

Recall that when keys are integers, Lua stores their values in the array part of the hybrid table [5,6]. The array-part corresponds to a range $[1, 2^a]$ 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.[9] 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 hashtable, each key going either to the hash-part or to the array-part if it is an integer smaller than or equal to A'.

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

3.1 Settings for the analysis

In all the following analysis of Lua hybrid tables, we consider that only insertions of pairs key/values are performed. This setting is sufficient to exhibit some unfortunate behavior in natural models, and it can only become worse if we also have 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 $\boldsymbol{y} = (y_1, \dots, y_n)$ of integers (keys) into an initially empty Lua table. We write $t_0, t_1, t_2, \dots, t_\ell$ for the sequence of rehash times, setting $t_0 = 0$, letting t_i be the time of the i-th rehash. Let $\ell = \ell(\boldsymbol{y})$ be the total number of rehashes. More precisely, the insertion of y_{t_i} induces the i-th rehash. Denote by β_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 $\mathcal{O}(1)$, so the overall cost is $\mathcal{O}(n)$. 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.

3.2 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 $\mathcal{O}(n \log n)$ is the worst case possible for a sequence n insertions into an initially empty Lua table.

Example 2. Consider inserting $(-(2^k - 1), -(2^k - 2), \dots, 0, 1, 2, \dots, 2^k)$ 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.

Non-positive integers go into the hash-part of the table. Thus $(-(2^k-1), -(2^k-2), \dots, 0$ go into the hashmap, which is going to be full and of size $M = 2^k$. Then inserting 1, induces a rehash. However 1 goes immediately to the array-part. Continuing, we rehash and double the size of the array-part when inserting $1, 2, 3, 5, 9, \dots$, in short for 1 and $(2^i + 1)_{i=0}^{i=k-1}$. 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 1. *When performing n insertions into an initially empty Lua table, the insertion function is called $\mathcal{O}(n \log n)$ times in the worst-case.*

Proof (Proof sketch.). We want to show that the cost $C := \sum_{i=0}^{\ell} \beta_i$ is $\mathcal{O}(n \log n)$. For this, we distinguish the β_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 $\mathcal{O}(\log n)$. The hashmap does not increase when the array-part

See Appendix ?? for the full proof.

does, as elements are added one at a time. Since $\beta_i \leq 2n$ for all i, the overall contribution of these β_i's to C is at most $\mathcal{O}(n \log n)$.

At this point we show that the remaining rehashes, in which the hash-part grows, only contribute to a total of $\mathcal{O}(n)$, which concludes the proof.

Remark 1. If not all of the n insertions are positive integers, we may refine the result. Let n' be the total number of positive integers. Then the worst case for C is $\mathcal{O}(n + n \log(1 + n'))$, as long as only insertions are performed.

3.3 Inserting permutations in Lua Tables

In Example 2 we showed a sequence of n insertions of integers leading to a cost $C = \Theta(n \log n)$. Some of the integers were negative, so they could only ever be in the hashmap. In this subsection we show that this worst case is still attainable on a very natural setting involving only positive integers: the sequence $\boldsymbol{y}$ is a permutation of $[n] := \{1, \ldots, n\}$. This setting, a priori, gives the array-part the best chance possible of being exploited, while not repeating keys. We present both the worst-case (this subsection) and the case of a random permutation (Subsection 3.4). Though presented in terms of permutations, these settings apply whenever the keys are consecutive integers but do not appear in increasing order, for instance during the marking of the transversal of a graph of vertex set $[n]$.

Proposition 2. *Inserting n elements given by the order of a permutation π of $[n]$ requires $\Omega(n \log n)$ calls to the insertion function of Lua in worst-case.*

The full proof is available in Appendix **??**.

Proof (Proof sketch.). Consider first $n = 3 \times 2^k$ for some $k > 0$. Define the permutation

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

The integer keys $2 \cdot 2^k + 1, \ldots 3 \cdot 2^k$ cannot be on the array-part, unless we have inserted more than $t = 2^{k+1}$ elements of π. Thus, after inserting $\pi(1), \ldots, \pi(2^k)$, the hash-part has size $M = 2^k$ and it is full. Then, the insertion of $\pi(2^k + 1)$ and $\pi(2^k + 2^j + 1)$, as long as $j < k$, induces a rehash that increases the size of the array. Each of the k rehashes we have just described has a cost of at least 2^k. Hence we obtain a cost that is $\Omega(2^k \times k) = \Omega(n \times \log n)$. For general n, pick the largest k with $3 \cdot 2^k \leq n$, and complete the permutation π above.

3.4 Average case for insertion of permutations

Fortunately, the average case for permutations is almost linear: our main result of this section, Theorem 2, states that, for any *super-linear* function $h(n)$, with probability tending to 1 the cost C of a random permutation is $\mathcal{O}(h(n))$.

However, we will see that the array-part is not really exploited as it should. For a random permutation, with high probability, the corresponding Lua table does not have an array-part until the very end: for this scenario we do not really take advantage of the hybrid data structure.

Theorem 2. *Let $g\colon \mathbf{N} \to \mathbb{R}_{>0}$ be an arbitrary non-decreasing function satisfying $g(n) \to \infty$ and $g(n) = o(n)$ as $n \to \infty$. 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 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 subset of keys from S revealed up to time t. Consider a time $t \leq cn$ with $c < 1/2$. The following lemma tells us 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 highly unlikely. Note that the result only depends on the cardinality of S.

Lemma 5. *There exist $D > 0$ such that if $t \leq c\,n$, for $c < 1/2$, then*

$$\Pr_n\left(|S_t(\pi)| > |S|/2\right) \leq D \cdot \exp\left(-\tfrac{(1-2c)^2}{2} \times |S|\right), \text{ for all } S \subseteq [n] \text{ with } |S| \leq n/4.$$

We give here a sketch of the proof of Theorem 2. First, it is enough to assume $n = 2^k$. Otherwise we may round down to a power of two, as the integers $[2^k + 1, n]$ can only go into the hash-part. By Lemma 5, at time $t \approx n/3$ the array-part is very small (actually $\mathcal{O}(g(n))$), because the subsets $[1, \ldots, 2^a] \subset [n]$ are unlikely to be half-full if $t/n \leq 1/3$. Thus the hash-part must have size $M = 2^{k-1} = n/2$, as $n/3$ is much larger than $2^{k-1}/2 = n/4$. We do not rehash again until $t > M = 2^{k-1}$, but then the array-part takes up the whole size of the permutation. Thus, with high probability, the array-part goes from being very small $A = \mathcal{O}(g(n))$ to being of size $A = n$ on a single rehash. The array-part can only grow, as there are no deletions, hence the number of rehashes in which it actually increases is very small. These rehashes account for $\mathcal{O}(ng(n))$ calls to the insertion function. Finally we show that the rest of the rehashes, in which the hash-part grows, can only account for $\mathcal{O}(n)$ calls, concluding the proof.

See Appendix ?? for the complete proof with all of the technical lemmas.

4 Conclusions & final remarks

The only data-structuring mechanism in Lua are tables, so it is of the upmost importance that they be 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 creating 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 sub-optimal performance of the Lua hybrid tables, as our theoretical analysis has shown. Fig.4 illustrates that this unwanted behavior shows in practice on our simulations.

In Section 3, we have shown that the hybrid structure introduces similar issues (see Prop. 2), even when considering only insertions. The effect is more limited than in the case of deletions (see Prop. 1 and Thm. 2), yet the array-part might not be exploited (to reduce memory consumption) as much as would be expected.

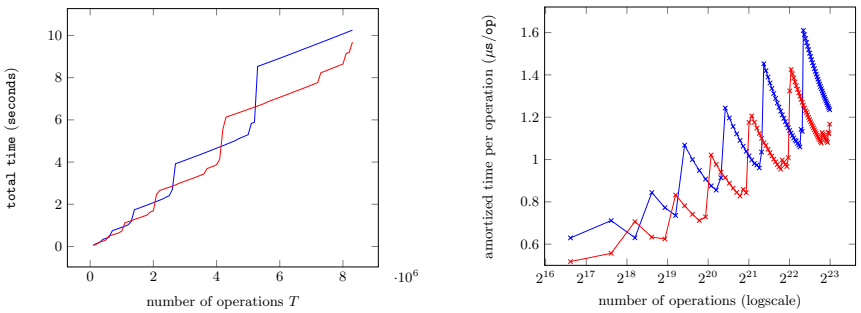

Fig. 4: Experimental plots for time against number of operations. The plot on the right shows the average microseconds/operation. Color blue corresponds to a probability of insertion $p = 0.9$, while red corresponds to $p = 0.75$. Each point is the average of 100 simulations in Lua.

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

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, August 20-22, 2018, Helsinki, Finland. LIPIcs, vol. 112, pp. 4:1–4:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018)
2. Buss, S., Knop, A.: Strategies for stable merge sorting. In: Chan, T.M. (ed.) Proceedings of the 30th ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019. pp. 1272–1290. SIAM (2019)
3. De Gouw, S., Rot, J., de 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. pp. 273–289. Springer (2015)
4. Ierusalimschy, R., de Figueiredo, L.H., Filho, W.C.: Lua-an extensible extension language. Software: Practice & Experience **26**, 635–652 (1996)
5. Ierusalimschy, R., de Figueiredo, L.H., Filho, W.C.: The implementation of Lua 5.0. J. Univers. Comput. Sci. **11**(7), 1159–1176 (2005)
6. Ierusalimschy, R., de Figueiredo, L.H., Filho, W.C.: Lua Programming Gems. Lua.Org (2008)
7. Knuth, D.E.: The Art of Computer Programming: Sorting and Searching, vol. 3. Addison-Wesley, 2nd edn. (1998)