A Probabilistic Model Revealing Shortcomings in Lua's Hybrid Tables

Pablo Rotondo

LIGM, Université Gustave Eiffel

Joint work with

Conrado Martínez (UPC), Cyril Nicaud (LIGM)

COCOON 2022,

Online, 23 October, 2022.

Introduction

- Aim: study and model actual implementations
  - Engineers sometimes choose innovative implementations
    e.g., `TimSort` in `Python`.
  - Study choices in depth, make recommendations.

Introduction

- Aim: study and model actual implementations
  - Engineers sometimes choose innovative implementations
    e.g., `TimSort` in `Python`.
  - Study choices in depth, make recommendations.
- The `Lua` programming language
  - Scripting language widely used in the gaming industry,
  - Efficient, lightweight (few Kb of `C` code!), embeddable.

Introduction

- Aim: study and model actual implementations
  - Engineers sometimes choose innovative implementations
    e.g., `TimSort` in `Python`.
  - Study choices in depth, make recommendations.
- The `Lua` programming language
  - Scripting language widely used in the gaming industry,
  - Efficient, lightweight (few Kb of `C` code!), embeddable.

$\Rightarrow$ `Lua 5.0` introduced several innovations,
among them a new Table structure.

Introduction: table structure in Lua

- Only data-structuring mechanism in Lua
  - assignment H[x]=y, any types of x and y.

Introduction: table structure in Lua

- Only data-structuring mechanism in Lua
  - assignment `H[x]=y`, any types of `x` and `y`.
- Implementation
  - originally a simple hash-table up to `Lua 4`.

Introduction: table structure in Lua

- Only data-structuring mechanism in Lua
  - assignment H[x]=y, any types of x and y.
- Implementation
  - originally a simple hash-table up to Lua 4.
  - Lua 5 introduced a hybrid table-array,

Introduction: table structure in Lua

- Only data-structuring mechanism in Lua
  - assignment `H[x]=y`, any types of `x` and `y`.
- Implementation
  - originally a simple hash-table up to `Lua 4`.
  - `Lua 5` introduced a hybrid table-array,
  - integer keys form array $\{1, \ldots, n\}$ at least half-full.

Introduction: table structure in Lua

- Only data-structuring mechanism in Lua
  - assignment H[x]=y, any types of x and y.
- Implementation
  - originally a simple hash-table up to Lua 4.
  - Lua 5 introduced a hybrid table-array,
  - integer keys form array $\{1, \ldots, n\}$ at least half-full.

In our work we

- study the hash-table mechanism, [*main result*]

Introduction: table structure in Lua

- Only data-structuring mechanism in Lua
  - assignment `H[x]=y`, any types of `x` and `y`.
- Implementation
  - originally a simple hash-table up to `Lua 4`.
  - `Lua 5` introduced a hybrid table-array,
  - integer keys form array $\{1, \ldots, n\}$ at least half-full.

In our work we

- study the hash-table mechanism, [*main result*]
- in worst case, but most importantly,
  we introduce a reasonable probabilistic model.

Introduction: table structure in Lua

- Only data-structuring mechanism in Lua
  - assignment `H[x]=y`, any types of `x` and `y`.
- Implementation
  - originally a simple hash-table up to `Lua 4`.
  - `Lua 5` introduced a hybrid table-array,
  - integer keys form array $\{1, \ldots, n\}$ at least half-full.

In our work we

- study the hash-table mechanism, [*main result*]
- in worst case, but most importantly,
  we introduce a reasonable probabilistic model.
- present also an analysis of the hybrid table-array.

Introduction: worst case Lua hash-table

`Lua`'s hash-table aims for space-efficiency:

Introduction: worst case Lua hash-table

`Lua`'s hash-table aims for space-efficiency:

- insertions and lookups work in amortized $O(1)$

even if table is full.

Introduction: worst case Lua hash-table

`Lua`'s hash-table aims for space-efficiency:

- insertions and lookups work in amortized $O(1)$
  even if table is full.
- but we show there is a degradation if deletions are allowed.

Introduction: worst case Lua hash-table

`Lua`'s hash-table aims for space-efficiency:

- insertions and lookups work in amortized $O(1)$

  even if table is full.

- but we show there is a degradation if deletions are allowed.

Consider sequences of T insertions/deletions starting from an empty table

Introduction: worst case Lua hash-table

`Lua`'s hash-table aims for space-efficiency:

- insertions and lookups work in amortized $O(1)$
  even if table is full.
- but we show there is a degradation if deletions are allowed.

Consider sequences of T insertions/deletions starting from an empty table

Proposition: worst case

There is sequence of operations giving time $\Theta(T^2)$.

Introduction: worst case Lua hash-table

Lua's hash-table aims for space-efficiency:

- insertions and lookups work in amortized $O(1)$
  even if table is full.
- but we show there is a degradation if deletions are allowed.

Consider sequences of T insertions/deletions starting from an empty table

Proposition: worst case

There is sequence of operations giving time $\Theta(T^2)$.

- The example requires an unlikely cycle of delete-insert.

Introduction: worst case Lua hash-table

Lua's hash-table aims for space-efficiency:

- insertions and lookups work in amortized $O(1)$
  even if table is full.
- but we show there is a degradation if deletions are allowed.

Consider sequences of T insertions/deletions starting from an empty table

Proposition: worst case

There is sequence of operations giving time $\Theta(T^2)$.

- The example requires an unlikely cycle of delete-insert.
- Trouble for more realistic examples ?

Introduction: probabilistic model for the hash-table

Simple Probabilistic model

Consider $p > \frac{1}{2}$. A random sequence of T insertion/deletions:

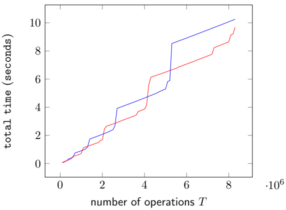
- with probability p insert a new element,
- with probability $1 - p$ delete an element.

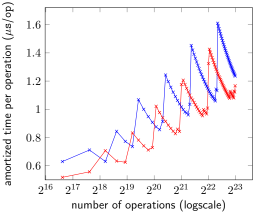
Introduction: probabilistic model for the hash-table

Simple Probabilistic model

Consider $p > \frac{1}{2}$. A random sequence of T insertion/deletions:

- with probability p insert a new element,
- with probability $1 - p$ delete an element.





Main result: `Lua` hash-table

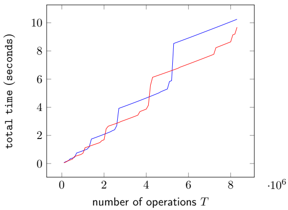
With high probability time is $\Omega(T \log T)$.

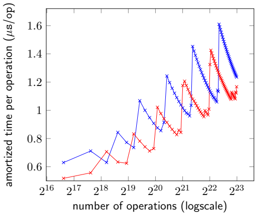
Introduction: probabilistic model for the hash-table

Simple Probabilistic model

Consider $p > \frac{1}{2}$. A random sequence of T insertion/deletions:

- with probability p insert a new element,
- with probability $1-p$ delete an element.





Main result: Lua hash-table

With high probability time is $\Omega(T \log T)$. ☹

Plan of the talk

1. The Lua hashmap
2. The probabilistic model
3. Conclusions and further work

The *pure* hashmap mechanism

`Lua`'s hashmap consists of

- an array H of size $M = 2^m$,

The *pure* hashmap mechanism

`Lua`'s hashmap consists of

- an array H of size $M = 2^m$,
- a hash function $h(x) = x \bmod M$, [*we do not discuss choice of* h]

The *pure* hashmap mechanism

`Lua`'s hashmap consists of

- an array H of size $M = 2^m$,
- a hash function $h(x) = x \bmod M$, [*we do not discuss choice of* h]
- entries: *key*, *value*, *index of next entry* in chain

The *pure* hashmap mechanism

Lua's hashmap consists of

- an array H of size $M = 2^m$,
- a hash function $h(x) = x \bmod M$, [*we do not discuss choice of* h]
- entries: *key*, *value*, *index of next entry* in chain

H	0	1	2	3	4	5	6	7
	120		a	d		b	e	c
	20		8	8		12	21	11
	7		5	nil		nil	nil	6

The *pure* hashmap mechanism: insertion

Insertions work as follows: key x

- if position $h(x)$ free $\Rightarrow$ insert

The *pure* hashmap mechanism: insertion

Insertions work as follows: key x

- if position $h(x)$ free $\Rightarrow$ insert
- else position $h(x)$ is occupied by key y,

  • if $h(y) = h(x) \Rightarrow$ put x into a free position, update chain $\mathtt{pos}(y) \to \mathtt{pos}(z) \to \dots$ to $\mathtt{pos}(y) \to \mathtt{pos}(x) \to \mathtt{pos}(z) \to \dots$

The *pure* hashmap mechanism: insertion

Insertions work as follows: key x

- if position $h(x)$ free $\Rightarrow$ insert
- else position $h(x)$ is occupied by key y,

  • if $h(y) = h(x) \Rightarrow$ put x into a free position, update chain $\texttt{pos}(y) \to \texttt{pos}(z) \to \ldots$ to $\texttt{pos}(y) \to \texttt{pos}(x) \to \texttt{pos}(z) \to \ldots$

  • if $h(y) \neq h(x) \Rightarrow$ we migrate y into a free position, updating its chain and put x at position $h(x)$.

	0	1	2	3	4	5	6	7
H					a 11 nil			

$h(\texttt{a}) = 4$, $h(\texttt{b}) = 4$,

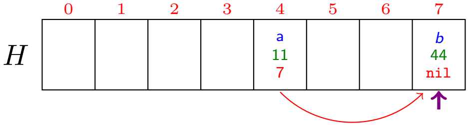
The *pure* hashmap mechanism: insertion

Insertions work as follows: key x

- if position $h(x)$ free $\Rightarrow$ insert
- else position $h(x)$ is occupied by key y,

  • if $h(y) = h(x) \Rightarrow$ put x into a free position, update chain $\texttt{pos}(y) \to \texttt{pos}(z) \to \ldots$ to $\texttt{pos}(y) \to \texttt{pos}(x) \to \texttt{pos}(z) \to \ldots$

  • if $h(y) \neq h(x) \Rightarrow$ we migrate y into a free position, updating its chain and put x at position $h(x)$.



$h(\texttt{a}) = 4$, $h(\texttt{b}) = 4$, $h(\texttt{d}) = 4$,

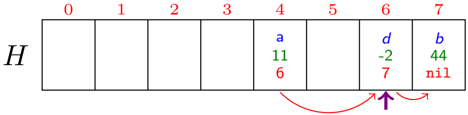
The *pure* hashmap mechanism: insertion

Insertions work as follows: key x

- if position $h(x)$ free $\Rightarrow$ insert
- else position $h(x)$ is occupied by key y,

  • if $h(y) = h(x)$ $\Rightarrow$ put x into a free position, update chain $\texttt{pos}(y) \to \texttt{pos}(z) \to \dots$ to $\texttt{pos}(y) \to \texttt{pos}(x) \to \texttt{pos}(z) \to \dots$

  • if $h(y) \neq h(x)$ $\Rightarrow$ we migrate y into a free position, updating its chain and put x at position $h(x)$.



$$h(\texttt{a}) = 4,\ h(\texttt{b}) = 4,\ h(\texttt{d}) = 4,\ h(\texttt{e}) = 7,$$

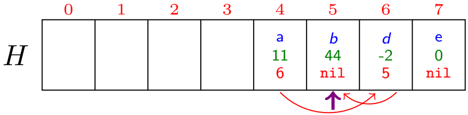
The *pure* hashmap mechanism: insertion

Insertions work as follows: key x

- if position $h(x)$ free $\Rightarrow$ insert
- else position $h(x)$ is occupied by key y,

  • if $h(y) = h(x) \Rightarrow$ put x into a free position, update chain $\texttt{pos}(y) \to \texttt{pos}(z) \to \ldots$ to $\texttt{pos}(y) \to \texttt{pos}(x) \to \texttt{pos}(z) \to \ldots$

  • if $h(y) \neq h(x) \Rightarrow$ we migrate y into a free position, updating its chain and put x at position $h(x)$.



$h(\texttt{a}) = 4,\ h(\texttt{b}) = 4,\ h(\texttt{d}) = 4,\ h(\texttt{e}) = 7,\ h(\texttt{f}) = 7$

The *pure* hashmap mechanism: insertion

Insertions work as follows: key x

- if position $h(x)$ free $\Rightarrow$ insert
- else position $h(x)$ is occupied by key y,

  • if $h(y) = h(x)$ $\Rightarrow$ put x into a free position, update chain $\texttt{pos}(y) \to \texttt{pos}(z) \to \ldots$ to $\texttt{pos}(y) \to \texttt{pos}(x) \to \texttt{pos}(z) \to \ldots$

  • if $h(y) \neq h(x)$ $\Rightarrow$ we migrate y into a free position, updating its chain and put x at position $h(x)$.

	0	1	2	3	4	5	6	7
H					a 11 6	*b* 44 nil	*d* -2 5	e 0 nil

$h(\texttt{a}) = 4$, $h(\texttt{b}) = 4$, $h(\texttt{d}) = 4$, $h(\texttt{e}) = 7$, $h(\texttt{f}) = 7$

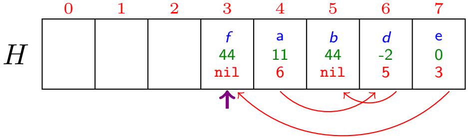
The *pure* hashmap mechanism: insertion

Insertions work as follows: key x

- if position $h(x)$ free $\Rightarrow$ insert
- else position $h(x)$ is occupied by key y,

  • if $h(y) = h(x) \Rightarrow$ put x into a free position, update chain $\texttt{pos}(y) \to \texttt{pos}(z) \to \ldots$ to $\texttt{pos}(y) \to \texttt{pos}(x) \to \texttt{pos}(z) \to \ldots$

  • if $h(y) \neq h(x) \Rightarrow$ we migrate y into a free position, updating its chain and put x at position $h(x)$.



$h(\texttt{a}) = 4,\ h(\texttt{b}) = 4,\ h(\texttt{d}) = 4,\ h(\texttt{e}) = 7,\ h(\texttt{f}) = 7$

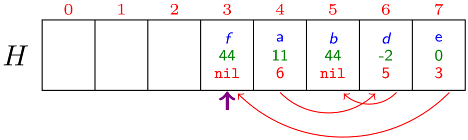
The *pure* hashmap mechanism: insertion

Insertions work as follows: key x

- if position $h(x)$ free $\Rightarrow$ insert
- else position $h(x)$ is occupied by key y,

  • if $h(y) = h(x) \Rightarrow$ put x into a free position, update chain $\texttt{pos}(y) \rightarrow \texttt{pos}(z) \rightarrow \ldots$ to $\texttt{pos}(y) \rightarrow \texttt{pos}(x) \rightarrow \texttt{pos}(z) \rightarrow \ldots$

  • if $h(y) \neq h(x) \Rightarrow$ we migrate y into a free position, updating its chain and put x at position $h(x)$.



Finding a free position: use pointer starting at end and moving to the left. If pointer exits, we rehash.

The *pure* hashmap mechanism: deletion

Deletions are simple:

- the *value* is marked as `nil`,

The *pure* hashmap mechanism: deletion

Deletions are simple:

- the *value* is marked as `nil`,
- chaining (*next cell*) kept intact.

The *pure* hashmap mechanism: deletion

Deletions are simple:

- the *value* is marked as `nil`,
- chaining (*next cell*) kept intact.

Deleted spot y

- can be reused to insert x when $h(x) = y$,

The *pure* hashmap mechanism: deletion

Deletions are simple:

- the *value* is marked as `nil`,
- chaining (*next cell*) kept intact.

Deleted spot y

- can be reused to insert x when $h(x) = y$,
- not taken into account by free position pointer
  $\Longrightarrow$ necessary to keep previous chaining

The *pure* hashmap mechanism: deletion

Deletions are simple:

- the *value* is marked as `nil`,
- chaining (*next cell*) kept intact.

Deleted spot y

- can be reused to insert x when $h(x) = y$,
- not taken into account by free position pointer
  $\Longrightarrow$ necessary to keep previous chaining

… deleted spots are cleaned up during rehashing

The *pure* hashmap mechanism: rehashing

Finding a free position: use pointer starting at end and moving to the left. If pointer exits, we rehash.

- Hashtable is then full, maybe with deleted cells.

The *pure* hashmap mechanism: rehashing

Finding a free position: use pointer starting at end and moving to the left. If pointer exits, we rehash.

- Hashtable is then full, maybe with deleted cells.
- Count actual used cells n,

The *pure* hashmap mechanism: rehashing

Finding a free position: use pointer starting at end and moving to the left. If pointer exits, we rehash.

- Hashtable is then full, maybe with deleted cells.
- Count actual used cells n,
- New hashtable of size $M = 2^m$, smallest m s.t. $n + 1 \leq 2^m$,
  $\implies +1$ for inserted element.

The *pure* hashmap mechanism: rehashing

Finding a free position: use pointer starting at end and moving to the left. If pointer exits, we rehash.

- Hashtable is then full, maybe with deleted cells.
- Count actual used cells n,
- New hashtable of size $M = 2^m$, smallest m s.t. $n + 1 \leq 2^m$,
  $\implies +1$ for inserted element.

Worst-case scenario

- insert until filling hashtable of size $M = 2^m$,
- alternate M deletion/insertions,
- insertions induce rehash unless deleted cell is picked,
- complexity $\Theta(M^2)$ for $3M$ operations.

The *pure* hashmap mechanism: rehashing

Finding a free position: use pointer starting at end and moving to the left. If pointer exits, we rehash.

- Hashtable is then full, maybe with deleted cells.
- Count actual used cells n,
- New hashtable of size $M = 2^m$, smallest m s.t. $n + 1 \leq 2^m$,
  $\Longrightarrow +1$ for inserted element.

Worst-case scenario

- insert until filling hashtable of size $M = 2^m$,
- alternate M deletion/insertions,
- insertions induce rehash unless deleted cell is picked,
- complexity $\Theta(M^2)$ for $3M$ operations.

... but it is not very realistic

The probabilistic model

We set a more interesting yet simple model

Probabilistic model

Fix $p > \frac{1}{2}$ and apply T insertion/deletions from an empty table:

- with probability p insert a new element,
- with probability $1 - p$ delete an element among present ones.

The probabilistic model

We set a more interesting yet simple model

Probabilistic model

Fix $p > \frac{1}{2}$ and apply T insertion/deletions from an empty table:

- with probability p insert a new element,
- with probability $1 - p$ delete an element among present ones.

Hashtable tends to grow: # keys $\approx pT - (1-p)T = (2p-1)T$

The probabilistic model

We set a more interesting yet simple model

Probabilistic model

Fix $p > \frac{1}{2}$ and apply T insertion/deletions from an empty table:

- with probability p insert a new element,
- with probability $1-p$ delete an element among present ones.

Hashtable tends to grow: $\#$ keys $\approx pT - (1-p)T = (2p-1)T$

Theorem (Martínez,Nicaud, R 2022)

With high probability, `Lua` uses $\Omega(T \log T)$ time for this process.

- Intuition: Large number of rehashes

  that serve only to remove few `nil` cells.

The probabilistic model

We set a more interesting yet simple model

Probabilistic model

Fix $p > \frac{1}{2}$ and apply T insertion/deletions from an empty table:

- with probability p insert a new element,
- with probability $1 - p$ delete an element among present ones.

Hashtable tends to grow: $\#$ keys $\approx pT - (1-p)T = (2p-1)T$

Theorem (Martínez,Nicaud, R 2022)

With high probability, `Lua` uses $\Omega(T \log T)$ time for this process.

- Intuition: Large number of rehashes
  that serve only to remove few `nil` cells.
- Each rehash costs linear time $\Theta(M)$. ☹

The probabilistic model: proof sketch

Theorem (Martínez,Nicaud, R 2022)

With high probability, `Lua` uses $\Omega(T \log T)$ time for this process.

- Number of keys in hashmap after t operations $\approx (2p-1)t$,
  with high probability size M only increases

The probabilistic model: proof sketch

Theorem (Martínez,Nicaud, R 2022)

With high probability, `Lua` uses $\Omega(T \log T)$ time for this process.

- Number of keys in hashmap after t operations $\approx (2p-1)t$,
  with high probability size M only increases
- First time we rehash into size $M = 2^m$ of order $\Theta(T)$:
  $\Longrightarrow \#\texttt{keys} = 2^{m-1} + 1$ and $2^{m-1} - 1$ free spots.

The probabilistic model: proof sketch

Theorem (Martínez,Nicaud, R 2022)

With high probability, `Lua` uses $\Omega(T \log T)$ time for this process.

- Number of keys in hashmap after t operations $\approx (2p-1)t$,
  with high probability size M only increases
- First time we rehash into size $M = 2^m$ of order $\Theta(T)$:
  $\implies \#\texttt{keys} = 2^{m-1} + 1$ and $2^{m-1} - 1$ free spots.
- But then random deletions slow down growth of M:
  $\implies$ from f free spots, we obtain γf after next rehash.

The probabilistic model: proof sketch

Theorem (Martínez,Nicaud, R 2022)

With high probability, Lua uses $\Omega(T \log T)$ time for this process.

- Number of keys in hashmap after t operations $\approx (2p-1)t$,
  with high probability size M only increases
- First time we rehash into size $M = 2^m$ of order $\Theta(T)$:
  $\implies \#\texttt{keys} = 2^{m-1} + 1$ and $2^{m-1} - 1$ free spots.
- But then random deletions slow down growth of M:
  $\implies$ from f free spots, we obtain γf after next rehash.

Lemma

If the hashmap has size M and just after a rehash it contains $f \gg \sqrt{M}$ free spots, then at the next rehash it still has size M and contains at least γf free spots (whp).

The probabilistic model: proof sketch

Theorem (Martínez,Nicaud, R 2022)

With high probability, Lua uses $\Omega(T \log T)$ time for this process.

- Number of keys in hashmap after t operations $\approx (2p-1)t$,
  with high probability size M only increases
- First time we rehash into size $M = 2^m$ of order $\Theta(T)$:
  $\Longrightarrow \#\texttt{keys} = 2^{m-1} + 1$ and $2^{m-1} - 1$ free spots.
- But then random deletions slow down growth of M:
  $\Longrightarrow$ from f free spots, we obtain γf after next rehash.

Lemma

If the hashmap has size M and just after a rehash it contains $f \gg \sqrt{M}$ free spots, then at the next rehash it still has size M and contains at least γf free spots (whp).

... at least $\log M$ rehashes to increase M

The probabilistic model: proof sketch

With (very) high probability:

- the hashtable is never empty after $t = 0$,
- we rehash at some point.

The probabilistic model: proof sketch

With (very) high probability:

- the hashtable is never empty after $t = 0$,
- we rehash at some point.

Under these conditions, between two rehashes, the number of deleted cells satisfies the recurrence (starting from $\delta_{t_0} = 0$)

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

The probabilistic model: proof sketch

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

- Equilibrium point at $\delta_t \approx \frac{1-p}{p} M$,

  ⊛ when $\delta_t < \frac{1-p}{p} M$ tendency to increase,

  ⊛ when $\delta_t > \frac{1-p}{p} M$ tendency to decrease,

The probabilistic model: proof sketch

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

- Equilibrium point at $\delta_t \approx \frac{1-p}{p} M$,

  ⊛ when $\delta_t < \frac{1-p}{p} M$ tendency to increase,

  ⊛ when $\delta_t > \frac{1-p}{p} M$ tendency to decrease,
- Rehash occurs a fraction before reaching equilibrium

  … sufficiently before to make δ_t increase linearly

The probabilistic model: proof sketch

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

- Equilibrium point at $\delta_t \approx \frac{1-p}{p} M$,

  ⊛ when $\delta_t < \frac{1-p}{p} M$ tendency to increase,

  ⊛ when $\delta_t > \frac{1-p}{p} M$ tendency to decrease,
- Rehash occurs a fraction before reaching equilibrium

  . . . sufficiently before to make δ_t increase linearly
- Tools: concentration inequalities.

Hybrid Tables and insertions

- Rehashes into same size hashtables pile up to $\Omega(T \log T)$.

Hybrid Tables and insertions

- Rehashes into same size hashtables pile up to $\Omega(T \log T)$.

And without deletions?

- Problem arises when considering effects of deletions.

Hybrid Tables and insertions

- Rehashes into same size hashtables pile up to $\Omega(T \log T)$.

And without deletions?

- Problem arises when considering effects of deletions.
- The hybrid data-structure presents a similar issue:
  using the array-part "simulates" deletions on the hash-part

Proposition [only insertions]

Inserting n elements into Lua's table takes $\Theta(n \log n)$ in the worst case.

Hybrid Tables and insertions

- Rehashes into same size hashtables pile up to $\Omega(T \log T)$.

And without deletions?

- Problem arises when considering effects of deletions.
- The hybrid data-structure presents a similar issue:
  using the array-part "simulates" deletions on the hash-part

Proposition [only insertions]

Inserting n elements into Lua's table takes $\Theta(n \log n)$ in the worst case.

Example: inserting $-(2^k - 1), -(2^k - 2), \ldots, -1, 0, 1, \ldots, 2^k$

Recap and conclusions

- Lua's hybrid data-structure is an interesting idea.
- We have presented a simple and natural probabilistic model revealing shortcomings in Lua's hashtables.

Recap and conclusions

- Lua's hybrid data-structure is an interesting idea.
- We have presented a simple and natural probabilistic model
  revealing shortcomings in Lua's hashtables.
- Issue can be fixed by ensuring more room when rehashing.
- This would also fix the hybrid part.

Recap and conclusions

- Lua's hybrid data-structure is an interesting idea.
- We have presented a simple and natural probabilistic model revealing shortcomings in Lua's hashtables.
- Issue can be fixed by ensuring more room when rehashing.
- This would also fix the hybrid part.

Conclusions

- Will Lua conceptors take this into account?
- Important to model and study algorithms implemented in practice.

Thank you!