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)

Séminaire LIGM,

Champs-Sur-Marne, 4 April, 2023.

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.

The Lua programming language[1]

❖ What is Lua?

Lua is a powerful, efficient, lightweight, embeddable scripting language. It supports procedural programming, object-oriented programming, functional programming, data-driven programming, and data description.

Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting bytecode with a register-based virtual machine, and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and rapid prototyping.

❖ Why choose Lua?

Lua is a proven, robust language

Lua has been used in many industrial applications (e.g., Adobe's Photoshop Lightroom), with an emphasis on embedded systems (e.g., the Ginga middleware for digital TV in Brazil) and games (e.g., World of Warcraft and Angry Birds). Lua is currently the leading scripting language in games. Lua has a solid reference manual and there are several books about it. Several versions of Lua have been released and used in real applications since its creation in 1993. Lua featured in HOPL III, the Third ACM SIGPLAN History of Programming Languages Conference, in 2007. Lua won the Front Line Award 2011 from the Game Developers Magazine.

The Lua programming language[1]

❖ What is Lua?

Lua is a powerful, efficient, lightweight, embeddable scripting language. It supports procedural programming, object-oriented programming, functional programming, data-driven programming, and data description.

Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting bytecode with a register-based virtual machine, and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and rapid prototyping.

❖ Why choose Lua?

Lua is a proven, robust language

Lua has been used in many industrial applications (e.g., Adobe's Photoshop Lightroom), with an emphasis on embedded systems (e.g., the Ginga middleware for digital TV in Brazil) and games (e.g., World of Warcraft and Angry Birds). Lua is currently the leading scripting language in games. Lua has a solid reference manual and there are several books about it. Several versions of Lua have been released and used in real applications since its creation in 1993. Lua featured in HOPL III, the Third ACM SIGPLAN History of Programming Languages Conference, in 2007. Lua won the Front Line Award 2011 from the Game Developers Magazine.

Lua is fast

Lua has a deserved reputation for performance. To claim to be "as fast as Lua" is an aspiration of other scripting languages. Several benchmarks show Lua as the fastest language in the realm of interpreted scripting languages. Lua is fast not only in fine-tuned benchmark programs, but in real life too. Substantial fractions of large applications have been written in Lua.

If you need even more speed, try LuaJIT, an independent implementation of Lua using a just-in-time compiler.

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 hash-array implementation,

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 hash-array implementation,

> The implementation of tables in Lua involves some clever algorithms. Every table in Lua has two parts: the *array* part and the *hash* part. The array part stores entries with integer keys in the range 1 to n, for some particular n. (We will discuss how this n is computed in a moment.) All other entries (including integer keys outside that range) go to the hash part.

Figure: Extract from the book Lua Programming Gems.

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 hash-array implementation,

> The implementation of tables in Lua involves some clever algorithms. Every table in Lua has two parts: the *array* part and the *hash* part. The array part stores entries with integer keys in the range 1 to n, for some particular n. (We will discuss how this n is computed in a moment.) All other entries (including integer keys outside that range) go to the hash part.

Figure: Extract from the book `Lua Programming Gems`.

- In our work we study this mechanism.

Introduction: worst case Lua hash-table

Running time:

Introduction: worst case Lua hash-table

Running time:

- insertions and lookups work in amortized $O(1)$
  even if table is full.

Introduction: worst case Lua hash-table

Running time:

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

Introduction: worst case Lua hash-table

Running time:

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

Consider sequences of T insertions/deletions from empty table

Introduction: worst case Lua hash-table

Running time:

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

Consider sequences of T insertions/deletions from empty table

Proposition: worst case

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

Introduction: worst case Lua hash-table

Running time:

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

Consider sequences of T insertions/deletions from empty table

Proposition: worst case

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

- Example requires an unlikely cycle of delete-insert.

Introduction: worst case Lua hash-table

Running time:

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

Consider sequences of T insertions/deletions from empty table

Proposition: worst case

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

- Example requires an unlikely cycle of delete-insert.
- A problem for more realistic scenarios ?

Introduction: probabilistic model for the hash-table

Our Simple Probabilistic model

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

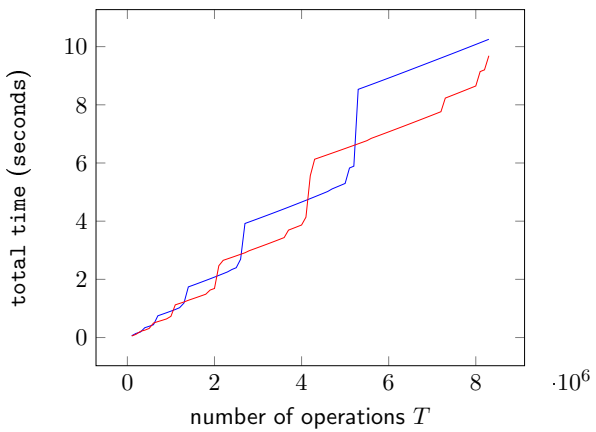
- with probability p insert a new element,
- with probability $1 - p$ delete an element.

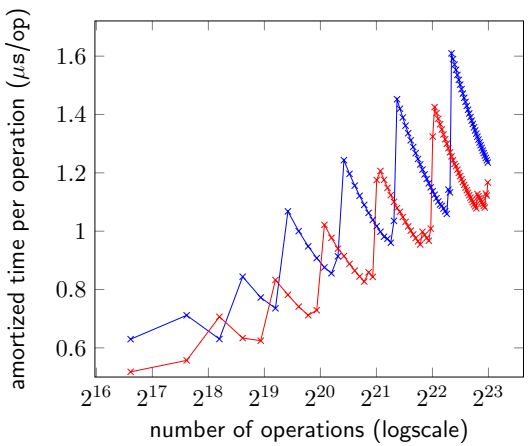
Introduction: probabilistic model for the hash-table

Our 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

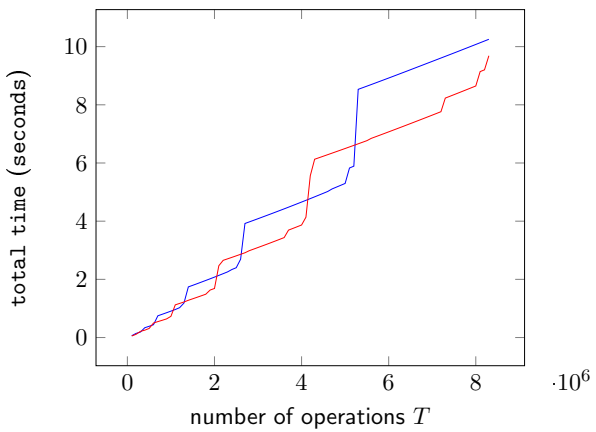
With high probability, complexity is $\Omega(T \log T)$.

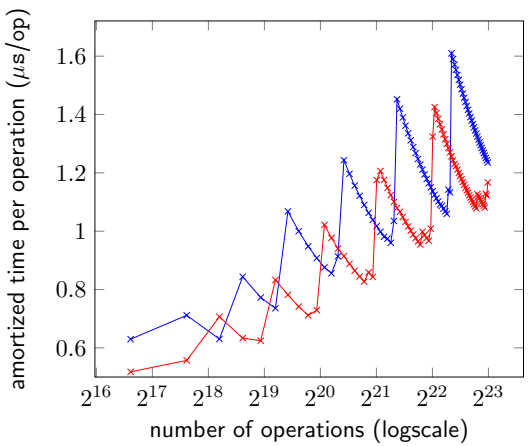
Introduction: probabilistic model for the hash-table

Our 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

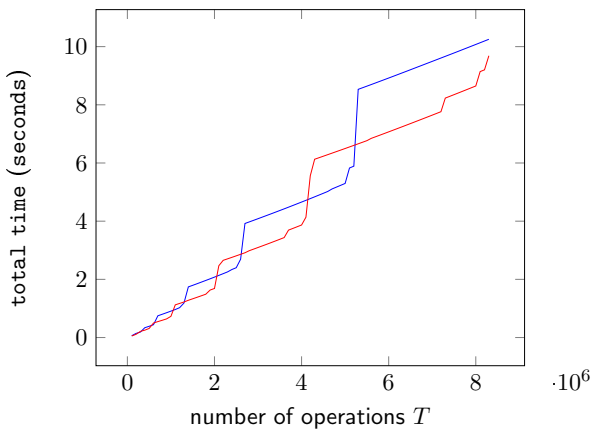
With high probability, complexity is $\Omega(T \log T)$. ☹

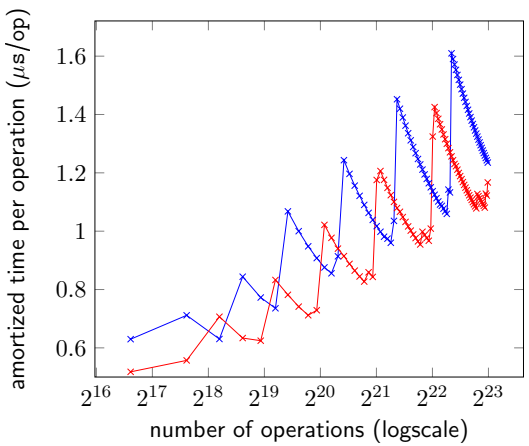
Introduction: probabilistic model for the hash-table

Our 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, complexity is $\Omega(T \log T)$. ☹

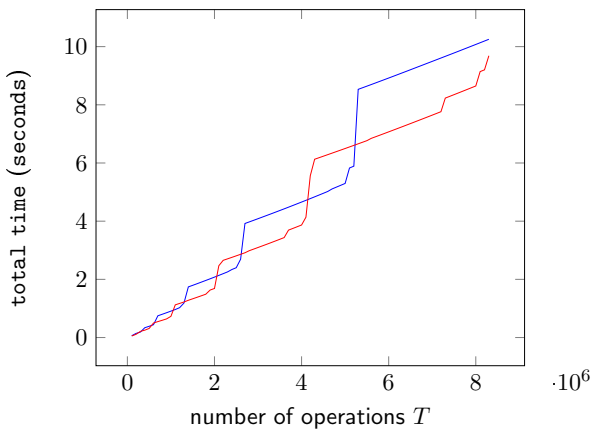
We propose potential fixes...

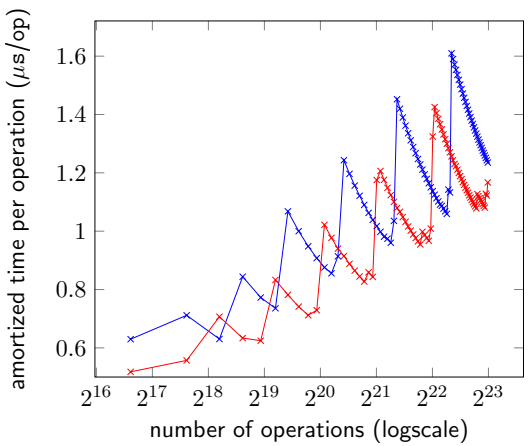
Introduction: probabilistic model for the hash-table

Our 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, complexity is $\Omega(T \log T)$. ☹

We propose potential fixes... which we will see and implement during the talk

Plan of the talk

1. The Lua hashmap
2. The probabilistic model
3. Hybrid Tables and insertions
4. Conclusions and further work

A (very) short intro to hashing

Problem: represent set from large universe $\mathcal{K}$ (e.g., IP addresses)

- Minimal interface: `init`, `insert`, `lookup`, `delete`

A (very) short intro to hashing

Problem: represent set from large universe $\mathcal{K}$ (e.g., IP addresses)

- Minimal interface: `init`, `insert`, `lookup`, `delete`
- Implementations:
  LinkedList (), Array (), balanced tree (),

A (very) short intro to hashing

Problem: represent set from large universe $\mathcal{K}$ (e.g., IP addresses)

- Minimal interface: `init`, `insert`, `lookup`, `delete`
- Implementations:
  LinkedList (), Array (), balanced tree (), hash-tables ($\star$)

A (very) short intro to hashing

Problem: represent set from large universe $\mathcal{K}$ (e.g., IP addresses)

- Minimal interface: `init`, `insert`, `lookup`, `delete`
- Implementations:
  LinkedList (), Array (), balanced tree (), hash-tables ($\star$)

Hash-tables solve the problem of the array "solution"

- array H of size M much smaller than $|\mathcal{K}|$,
- hash function $h: \mathcal{K} \to \mathbb{Z}_{\geq 0}$,
- store $x \in \mathcal{K}$ in slot $H[h(x) \bmod M]$.

A (very) short intro to hashing

Problem: represent set from large universe $\mathcal{K}$ (e.g., IP addresses)

- Minimal interface: `init`, `insert`, `lookup`, `delete`
- Implementations:
  LinkedList (), Array (), balanced tree (), hash-tables ($\star$)

Hash-tables solve the problem of the array "solution"

- array H of size M much smaller than $|\mathcal{K}|$,
- hash function $h: \mathcal{K} \to \mathbb{Z}_{\geq 0}$,
- store $x \in \mathcal{K}$ in slot $H[h(x) \bmod M]$.

... but there are surely many y with $h(x) \equiv h(y) \bmod M$?

A (very) short intro to hashing

Problem: represent set from large universe $\mathcal{K}$ (e.g., IP addresses)

- Minimal interface: `init`, `insert`, `lookup`, `delete`
- Implementations:
  LinkedList (), Array (), balanced tree (), hash-tables ($\star$)

Hash-tables solve the problem of the array "solution"

- array H of size M much smaller than $|\mathcal{K}|$,
- hash function $h: \mathcal{K} \to \mathbb{Z}_{\geq 0}$,
- store $x \in \mathcal{K}$ in slot $H[h(x) \bmod M]$.

... but there are surely many y with $h(x) \equiv h(y) \bmod M$? $\Rightarrow$ **collision**

A (very) short intro to hashing

Problem: represent set from large universe $\mathcal{K}$ (e.g., IP addresses)

- Minimal interface: `init`, `insert`, `lookup`, `delete`
- Implementations:
  LinkedList (), Array (), balanced tree (), hash-tables ($\star$)

Hash-tables solve the problem of the array "solution"

- array H of size M much smaller than $|\mathcal{K}|$,
- hash function $h: \mathcal{K} \to \mathbb{Z}_{\geq 0}$,
- store $x \in \mathcal{K}$ in slot $H[h(x) \bmod M]$.

... but there are surely many y with $h(x) \equiv h(y) \bmod M$? $\Rightarrow$ **collision**

Hash-functions must

- avoid collisions as much as possible,
- be fast to compute.

A (very) short intro to hashing

We still have to decide what to do in case of a collision:

- Separate chaining: use a linked-list at each slot $H[i]$,
- Internal chaining: put key somewhere else (where?) in H.

A (very) short intro to hashing

We still have to decide what to do in case of a collision:

- Separate chaining: use a linked-list at each slot $H[i]$,
- Internal chaining: put key somewhere else (where?) in H.

but what if

- Separate chaining: linked-lists are very long ?
- Internal chaining: array H is full ?

A (very) short intro to hashing

We still have to decide what to do in case of a collision:

- Separate chaining: use a linked-list at each slot $H[i]$,
- Internal chaining: put key somewhere else (where?) in H.

but what if

- Separate chaining: linked-lists are very long ?
- Internal chaining: array H is full ?

. . . rehash into larger array H'

Hashmap mechanism

Lua's hashmap consists of

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

Hashmap mechanism

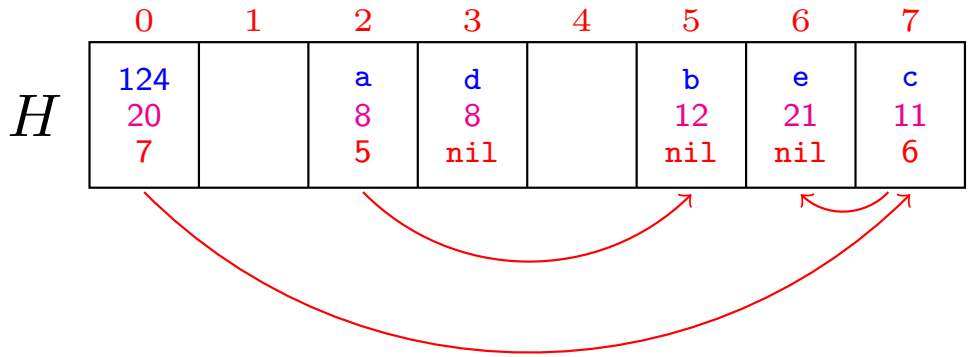
Lua's hashmap consists of

- an array H of size $M = 2^m$,
- entries: *key*, *value*, *index of next entry* in chain

Hashmap mechanism

Lua's hashmap consists of

- an array H of size $M = 2^m$,
- entries: *key*, *value*, *index of next entry* in chain

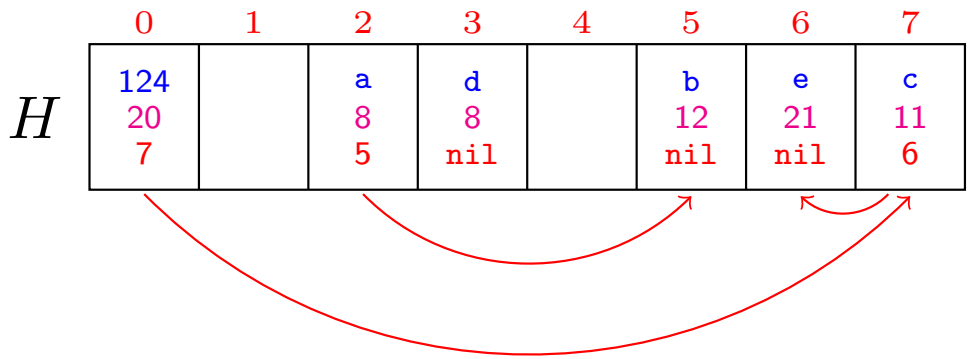


$\Rightarrow$ internal chaining

Hashmap mechanism

Lua's hashmap consists of

- an array H of size $M = 2^m$,
- entries: *key*, *value*, *index of next entry* in chain



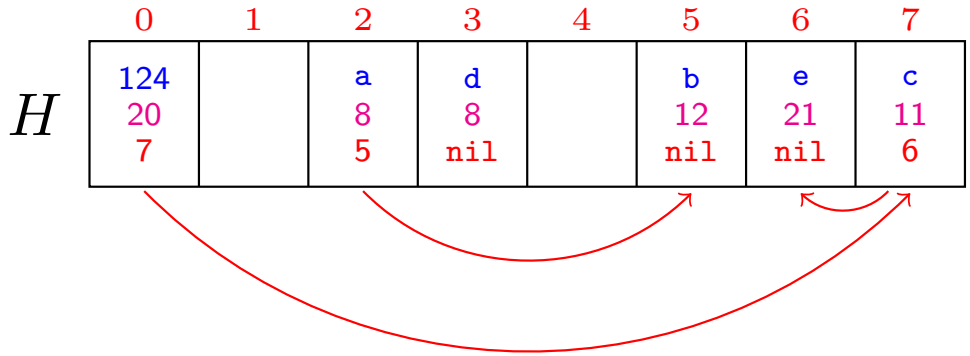
  $\Rightarrow$ internal chaining

- for x integer, hash function $h(x) = x \bmod M$,

Hashmap mechanism

Lua's hashmap consists of

- an array H of size $M = 2^m$,
- entries: *key*, *value*, *index of next entry* in chain



$\Rightarrow$ internal chaining

- for x integer, hash function $h(x) = x \bmod M$,

516 SEARCHING 6.4

when K is odd, and this will lead to a substantial bias in many files. It would be even worse to let M be a power of the radix of the computer, since K mod M would then be simply the least significant digits of K (independent of the other digits). Similarly we can argue that M probably shouldn't be a multiple of 3;

Figure: Extract from The Art of Computer Programming (vol. 3)

Hashmap mechanism: hash function

Not immediate that h may be a bad choice

- fast for integers: $x \bmod M = x\&(M-1)$,
- more involved for strings:

```
unsigned int luaS_hash (const char *str, size_t l,
     unsigned int seed) {
  unsigned int h = seed ^ cast_uint(l);
  for (; l > 0; l--)
    h ^= ((h<<5) + (h>>2) + cast_byte(str[l - 1]))
         ;
  return h;
}
```

Hashmap mechanism: hash function

Not immediate that h may be a bad choice

- fast for integers: $x \bmod M = x\&(M-1)$,
- more involved for strings:

```
unsigned int luaS_hash (const char *str, size_t l,
     unsigned int seed) {
  unsigned int h = seed ^ cast_uint(l);
  for (; l > 0; l--)
    h ^= ((h<<5) + (h>>2) + cast_byte(str[l - 1]))
         ;
  return h;
}
```

In this talk we do not discuss the choice of the hash-function h,

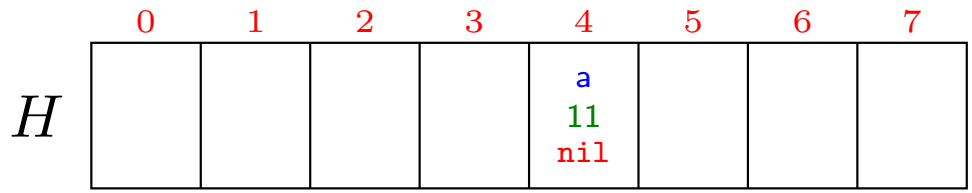
Base assumption

The function $x \mapsto h(x) \bmod M$ is roughly uniform

Hashmap mechanism: insertion

Insertions work as follows: we want to insert key x

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



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

Hashmap mechanism: insertion

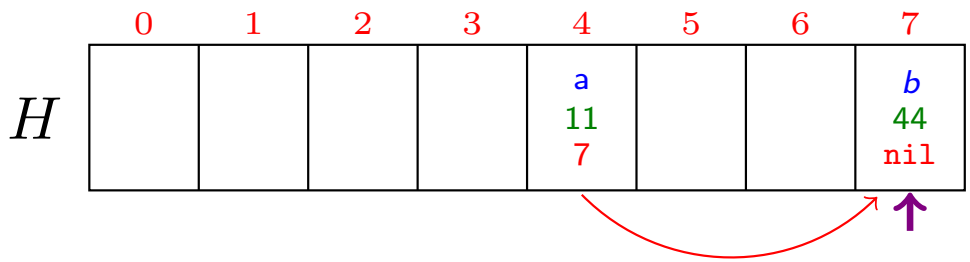
Insertions work as follows: we want to insert key x

- 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

$$\texttt{pos}(y) \to \texttt{pos}(z) \to \ldots \texttt{ to } \texttt{pos}(y) \to \texttt{pos}(x) \to \texttt{pos}(z) \to \ldots$$



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

Hashmap mechanism: insertion

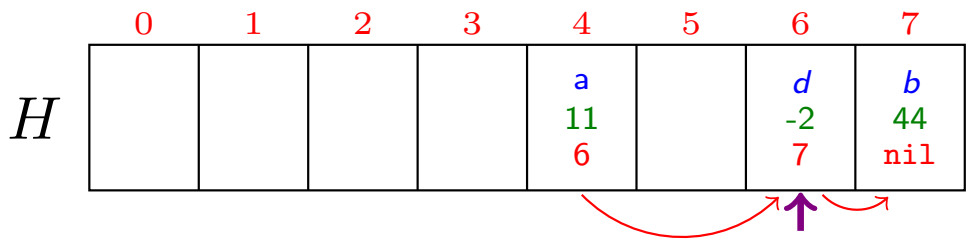
Insertions work as follows: we want to insert key x

- 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 $\texttt{pos}(y) \rightarrow \texttt{pos}(x) \rightarrow \texttt{pos}(z) \rightarrow \ldots$



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

Hashmap mechanism: insertion

Insertions work as follows: we want to insert key x

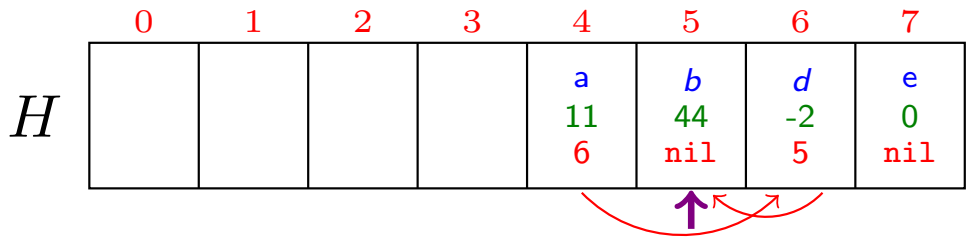
- if position $h(x)$ free $\Rightarrow$ insert

  ... else position $h(x)$ is occupied by key y,

- if $h(y) = h(x) \Rightarrow$ put x into a free position, update chain

  $$\texttt{pos}(y) \to \texttt{pos}(z) \to \ldots \texttt{ to } \texttt{pos}(y) \to \texttt{pos}(x) \to \texttt{pos}(z) \to \ldots$$

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



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

Hashmap mechanism: insertion

Insertions work as follows: we want to insert key x

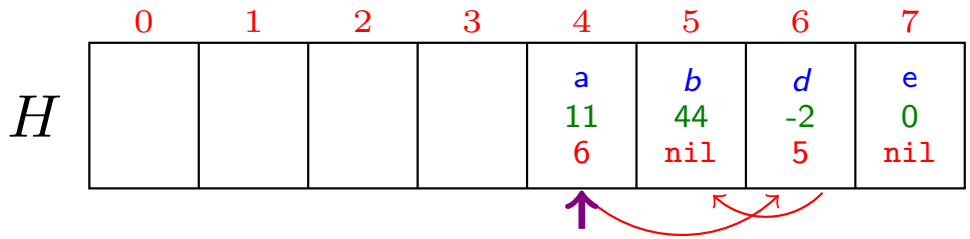
- if position $h(x)$ free $\Rightarrow$ insert

  ... else position $h(x)$ is occupied by key y,

- if $h(y) = h(x) \Rightarrow$ put x into a free position, update chain

  $$\texttt{pos}(y) \to \texttt{pos}(z) \to \ldots \texttt{ to } \texttt{pos}(y) \to \texttt{pos}(x) \to \texttt{pos}(z) \to \ldots$$

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



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

Hashmap mechanism: insertion

Insertions work as follows: we want to insert key x

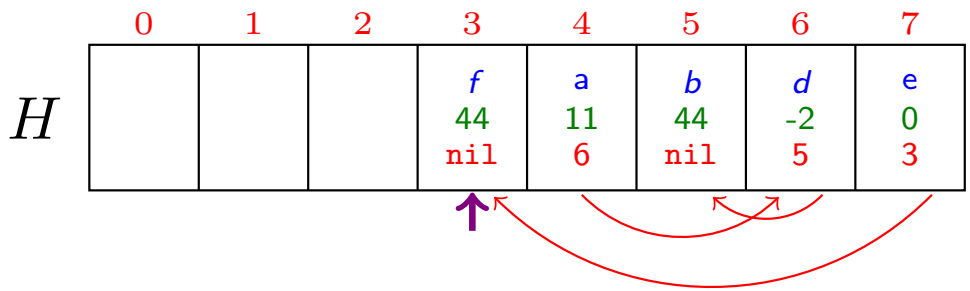
- if position $h(x)$ free $\Rightarrow$ insert

  ... else position $h(x)$ is occupied by key y,

- if $h(y) = h(x) \Rightarrow$ put x into a free position, update chain

  $$\texttt{pos}(y) \to \texttt{pos}(z) \to \ldots \texttt{ to } \texttt{pos}(y) \to \texttt{pos}(x) \to \texttt{pos}(z) \to \ldots$$

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



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

Hashmap mechanism: insertion

Insertions work as follows: we want to insert key x

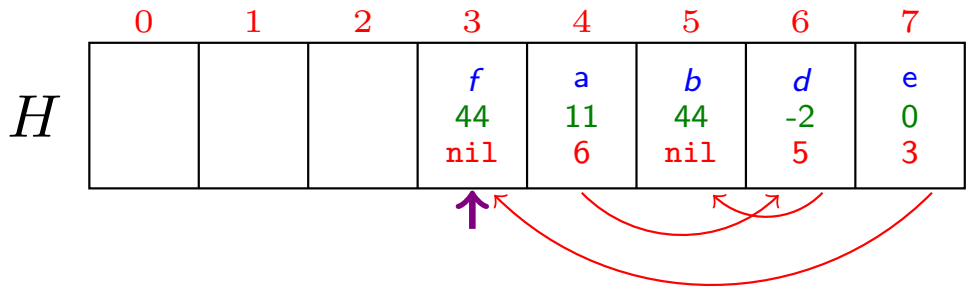
- if position $h(x)$ free $\Rightarrow$ insert

  ... else position $h(x)$ is occupied by key y,

- if $h(y) = h(x) \Rightarrow$ put x into a free position, update chain

  $$\texttt{pos}(y) \to \texttt{pos}(z) \to \ldots \texttt{ to } \texttt{pos}(y) \to \texttt{pos}(x) \to \texttt{pos}(z) \to \ldots$$

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



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

Finding free position: pointer from right to left.

Hashmap mechanism: insertion

Insertions work as follows: we want to insert key x

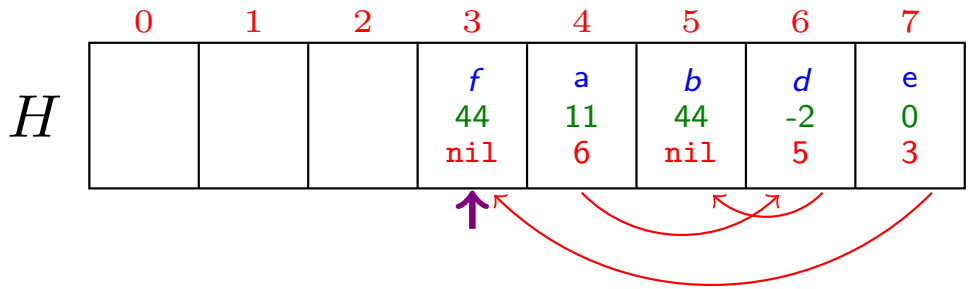
- if position $h(x)$ free $\Rightarrow$ insert

  ... else position $h(x)$ is occupied by key y,

- if $h(y) = h(x) \Rightarrow$ put x into a free position, update chain

  $$\texttt{pos}(y) \to \texttt{pos}(z) \to \ldots \texttt{ to } \texttt{pos}(y) \to \texttt{pos}(x) \to \texttt{pos}(z) \to \ldots$$

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



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

Finding free position: pointer from right to left. If pointer exits, rehash.

Hashmap mechanism: deletion

Deletions are simple

- *value* is marked as `nil` : equivalent to $H[\mathtt{x}] = \mathtt{nil}$,

Hashmap mechanism: deletion

Deletions are simple

- *value* is marked as `nil` : equivalent to $H[\mathtt{x}] = \mathtt{nil}$,
- coherent with semantics: if $\mathtt{x} \notin H \Rightarrow H[\mathtt{x}]$ evaluates `nil`,

Hashmap mechanism: deletion

Deletions are simple

- *value* is marked as `nil` : equivalent to $H[\mathtt{x}] = \mathtt{nil}$,
- coherent with semantics: if $\mathtt{x} \notin H \Rightarrow H[\mathtt{x}]$ evaluates `nil`,
- chaining (*next cell*) kept intact.

Hashmap mechanism: deletion

Deletions are simple

- *value* is marked as `nil` : equivalent to $H[\mathtt{x}] = \mathtt{nil}$,
- coherent with semantics: if $\mathtt{x} \notin H \Rightarrow H[\mathtt{x}]$ evaluates `nil`,
- chaining (*next cell*) kept intact.

Deleted position k

- can be reused to insert `x` when $h(\mathtt{x}) = k$,

Hashmap mechanism: deletion

Deletions are simple

- *value* is marked as `nil` : equivalent to $H[\mathtt{x}] = \mathtt{nil}$,
- coherent with semantics: if $\mathtt{x} \notin H \Rightarrow H[\mathtt{x}]$ evaluates `nil`,
- chaining (*next cell*) kept intact.

Deleted position k

- can be reused to insert x when $h(\mathtt{x}) = k$,
- not taken into account by free position pointer

$\Longrightarrow$ necessary to keep previous chaining

Hashmap mechanism: deletion

Deletions are simple

- *value* is marked as `nil` : equivalent to $H[\mathtt{x}] = \mathtt{nil}$,
- coherent with semantics: if $\mathtt{x} \notin H \Rightarrow H[\mathtt{x}]$ evaluates `nil`,
- chaining (*next cell*) kept intact.

Deleted position k

- can be reused to insert x when $h(\mathtt{x}) = k$,
- not taken into account by free position pointer

$\Longrightarrow$ necessary to keep previous chaining

. . . deleted spots are cleaned up during rehashing

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.

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 actually *used cells* n,

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 actually *used cells* n,
- Set size $M = 2^m$, with smallest m such that $n + 1 \leq 2^m$,
  $\Longrightarrow +1$ for inserted element.

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 actually *used cells* n,
- Set size $M = 2^m$, with smallest m such that $n + 1 \leq 2^m$,
  $\Longrightarrow$ $+1$ for inserted element.

Worst-case scenario construction

- insert until filling hashtable of size $M = 2^m$,
- do M iterations: (deletion+insertion),
- insertions induce rehash unless deleted cell is picked ($p = 1/M$)

Expected complexity $\Theta(M^2)$ for $3M$ operations.

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 actually *used cells* n,
- Set size $M = 2^m$, with smallest m such that $n + 1 \leq 2^m$,
  $\Longrightarrow +1$ for inserted element.

Worst-case scenario construction

- insert until filling hashtable of size $M = 2^m$,
- do M iterations: (deletion+insertion),
- insertions induce rehash unless deleted cell is picked ($p = 1/M$)

Expected complexity $\Theta(M^2)$ for $3M$ operations.

. . . but it is not very realistic, users do not behave this way (?)

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.

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 *useless* rehashes

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 *useless* rehashes
- Each rehash costs linear time $\Theta(M)$. ☹

Intuition: useless rehash

When rehashing table of size M

- free position pointer does not take into account deleted cells,

Intuition: useless rehash

When rehashing table of size M

- free position pointer does not take into account deleted cells,
- if $0 < \delta < M/2$ deleted cells remain,

Intuition: useless rehash

When rehashing table of size M

- free position pointer does not take into account deleted cells,
- if $0 < \delta < M/2$ deleted cells remain,

  ... new hashtable has same size,

Intuition: useless rehash

When rehashing table of size M

- free position pointer does not take into account deleted cells,
- if $0 < \delta < M/2$ deleted cells remain,

  ... new hashtable has same size, even if δ small

Intuition: useless rehash

When rehashing table of size M

- free position pointer does not take into account deleted cells,
- if $0 < \delta < M/2$ deleted cells remain,

  ... new hashtable has same size, even if δ small

  ... and we have $f = \delta - 1$ free cells after rehash

Intuition: useless rehash

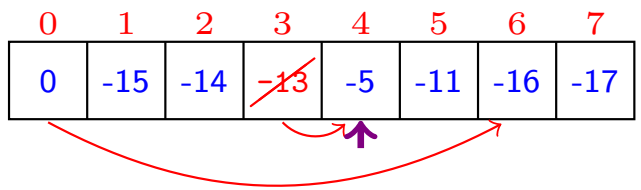
When rehashing table of size M

- free position pointer does not take into account deleted cells,
- if $0 < \delta < M/2$ deleted cells remain,

  ... new hashtable has same size, even if δ small

  ... and we have $f = \delta - 1$ free cells after rehash

Example: Before and after rehash (insertion -18), $\delta = 1$



Intuition: useless rehash

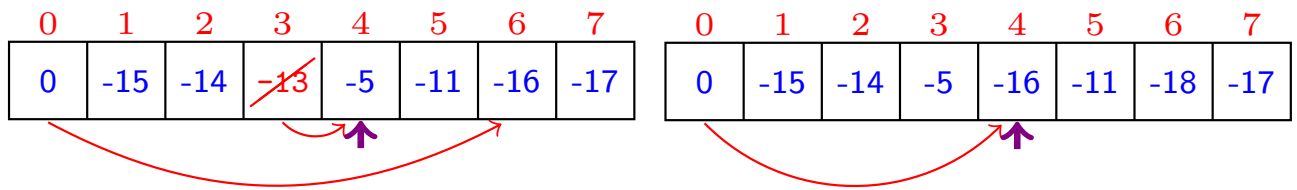
When rehashing table of size M

- free position pointer does not take into account deleted cells,
- if $0 < \delta < M/2$ deleted cells remain,

  ... new hashtable has same size, even if δ small

  ... and we have $f = \delta - 1$ free cells after rehash

Example: Before and after rehash (insertion -18), $\delta = 1$



Intuition: useless rehash

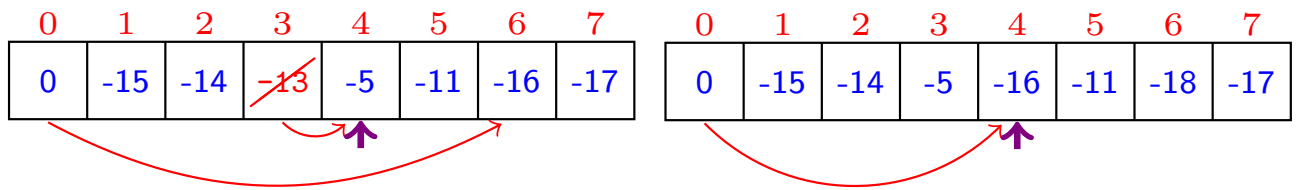
When rehashing table of size M

- free position pointer does not take into account deleted cells,
- if $0 < \delta < M/2$ deleted cells remain,

  ... new hashtable has same size, even if δ small

  ... and we have $f = \delta - 1$ free cells after rehash

  Example: Before and after rehash (insertion -18), $\delta = 1$



- such rehash is not of much use: only cleans (few) deleted cells

Intuition: useless rehash

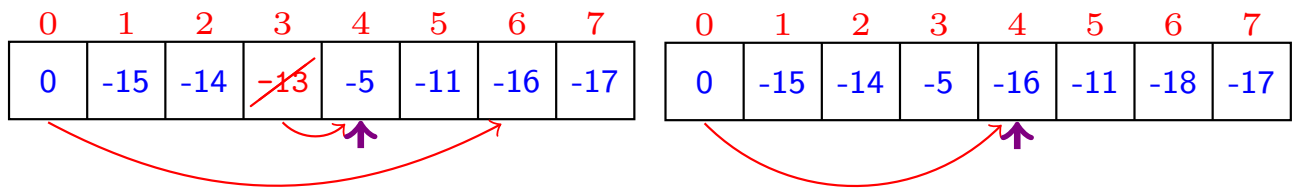
When rehashing table of size M

- free position pointer does not take into account deleted cells,
- if $0 < \delta < M/2$ deleted cells remain,

  ... new hashtable has same size, even if δ small

  ... and we have $f = \delta - 1$ free cells after rehash

  Example: Before and after rehash (insertion -18), $\delta = 1$



- such rehash is not of much use: only cleans (few) deleted cells

Intuition: useless rehash

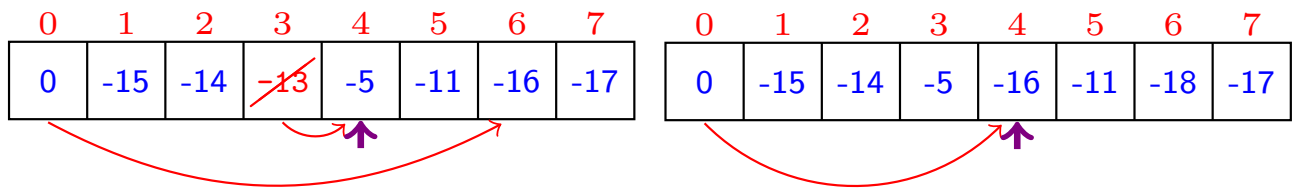
When rehashing table of size M

- free position pointer does not take into account deleted cells,
- if $0 < \delta < M/2$ deleted cells remain,

  ... new hashtable has same size, even if δ small

  ... and we have $f = \delta - 1$ free cells after rehash

  Example: Before and after rehash (insertion -18), $\delta = 1$



- such rehash is not of much use: only cleans (few) deleted cells

... useless rehashes go on until we hit a rehash with $\delta = 0$

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
- At some point 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
- At some point 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.

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
- At some point 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 hashmap has size M and just after a rehash it contains $f \gg \sqrt{M}$ free spots, 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
- At some point 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 hashmap has size M and just after a rehash it contains $f \gg \sqrt{M}$ free spots, 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 *simplifying assumptions*, 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

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

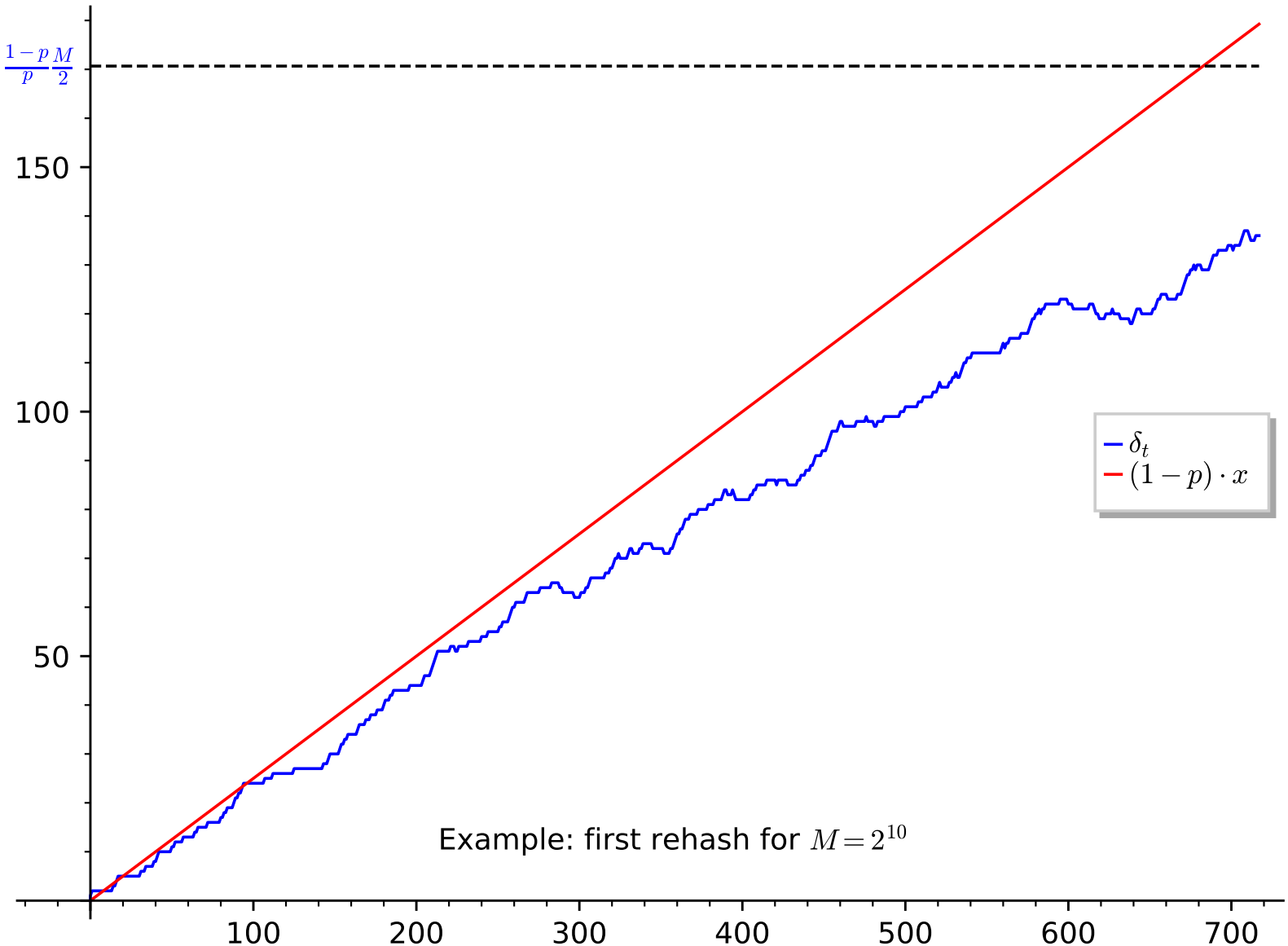
- Equilibrium point at $\delta_t \approx \frac{1-p}{p} M$,
  - ⊛ when $\delta_t < \frac{1-p}{p} M$ tendency to increase,
  - ⊛ when $\delta_t > \frac{1-p}{p} M$ tendency to decrease,

The probabilistic model: proof sketch

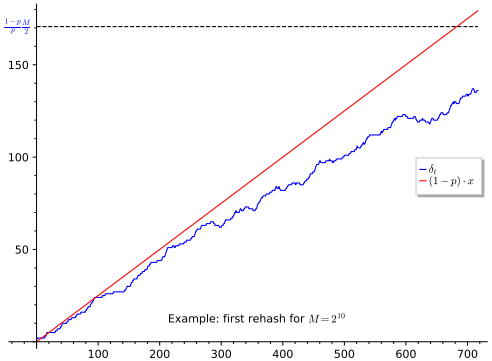
$$\delta_{t+1} = \begin{cases} \delta_t - 1 & \text{with probability } \frac{p\delta_t}{M} & [\textit{insertion at deleted key}], \\ \delta_t & \text{with probability } p\left(1 - \frac{\delta_t}{M}\right) & [\textit{insertion at free cell}], \\ \delta_t + 1 & \text{with probability } 1 - p & [\textit{deletion}].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

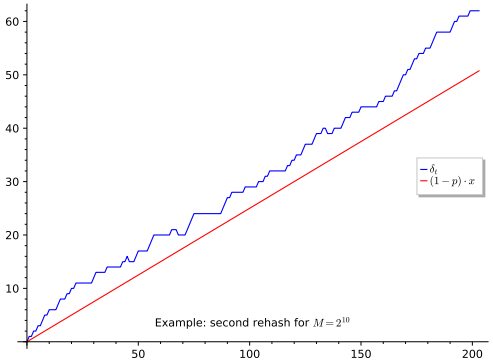
- Equilibrium point at $\delta_t \approx \frac{1-p}{p} M$,
  - ⊛ when $\delta_t < \frac{1-p}{p} M$ tendency to increase,
  - ⊛ when $\delta_t > \frac{1-p}{p} M$ tendency to decrease,
- Rehash occurs before reaching equilibrium
  - ... at the beginning δ_t increases linearly
  - ... as we approach equilibrium, increase weakens

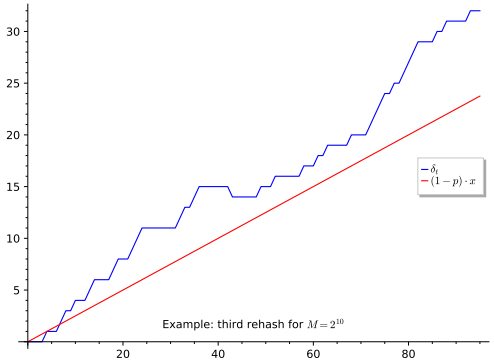
Evolution of number of deleted cells δ_t: linear increase



Evolution of number of deleted cells δ_t: linear increase

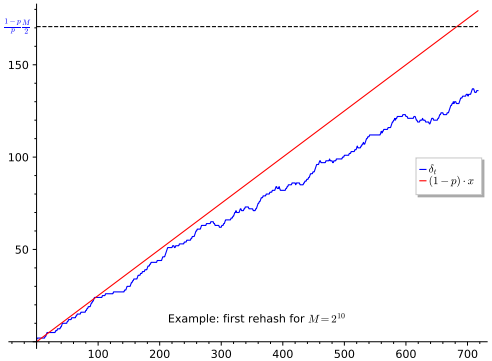


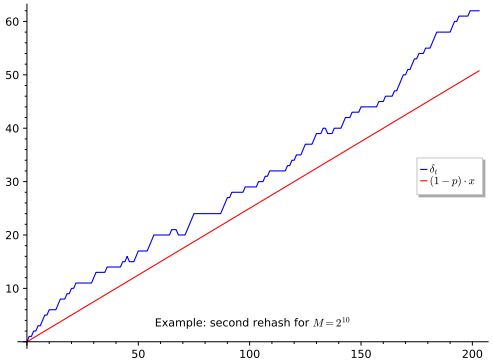


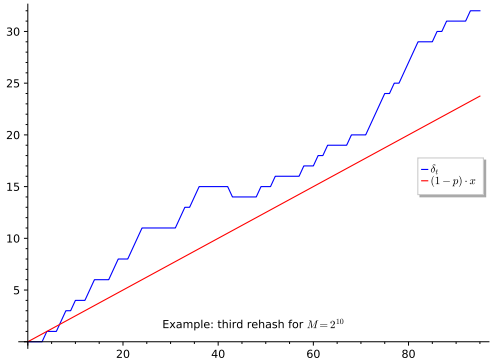


$$\delta_{t+1} = \begin{cases} \delta_t - 1 & \text{with probability } \frac{p\delta_t}{M} & [\textit{insertion at deleted key}], \\ \delta_t & \text{with probability } p\left(1 - \frac{\delta_t}{M}\right) & [\textit{insertion at free cell}], \\ \delta_t + 1 & \text{with probability } 1 - p & [\textit{deletion}].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

Evolution of number of deleted cells δ_t: linear increase







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

Let $\texttt{free}_0$ free (unused) cells after last rehash (set $t = 0$)

Remark: there is a linear regime for a proportion of time

After $t = \left\lfloor \frac{1-p}{p} \texttt{free}_0 \right\rfloor < \texttt{free}_0$ steps

$$\frac{p\delta_t}{M} \leq (1-p)\frac{\texttt{free}_0}{M} \leq \frac{1-p}{2},$$

$\Delta\delta_t = 1$ still twice as likely as $\Delta\delta_t = -1$.

Evolution of number of deleted cells δ_t: stopping time

- By time $t = \lfloor \frac{1-p}{p} \texttt{free}_0 \rfloor$, number of deleted cells δ_t increased linearly.
- It remains to show that it will not decrease

Evolution of number of deleted cells δ_t: stopping time

- By time $t = \lfloor \frac{1-p}{p} \mathtt{free}_0 \rfloor$, number of deleted cells δ_t increased linearly.
- It remains to show that it will not decrease

Lemma

If $c > \frac{1}{2p-1}$, number of operations before next rehash τ satisfies

$$\tau \leq c \cdot \mathtt{free}_0$$

with high probability.

. . . and so only a linear number of steps remain

Evolution of number of deleted cells δ_t: stopping time

- By time $t = \lfloor \frac{1-p}{p} \mathtt{free}_0 \rfloor$, number of deleted cells δ_t increased linearly.
- It remains to show that it will not decrease

Lemma

If $c > \frac{1}{2p-1}$, number of operations before next rehash τ satisfies

$$\tau \leq c \cdot \mathtt{free}_0$$

with high probability.

. . . and so only a linear number of steps remain

Lemma

With high probability equilibrium has never been reached by time τ.

. . . and at worse δ_t looks like a $\frac{1}{2} - \frac{1}{2}$ random walk (close to τ)

Fixing the Lua hash table

Potential solutions:

- implement real deletions

Fixing the Lua hash table

Potential solutions:

- implement real deletions

  . . . requires several changes (update chains, list of available cells)

Fixing the Lua hash table

Potential solutions:

- implement real deletions

  . . . requires several changes (update chains, list of available cells)

- keep a minimum proportion of free cells after rehashing

Fixing the Lua hash table

Potential solutions:

- implement real deletions

  . . . requires several changes (update chains, list of available cells)

- keep a minimum proportion of free cells after rehashing

  . . . we will implement it now in 2 minutes for $\geq 20\%$

Fixing the Lua hash table

Potential solutions:

- implement real deletions

  . . . requires several changes (update chains, list of available cells)

- keep a minimum proportion of free cells after rehashing

  . . . we will implement it now in 2 minutes for $\geq 20\%$

Fixing the Lua hash table

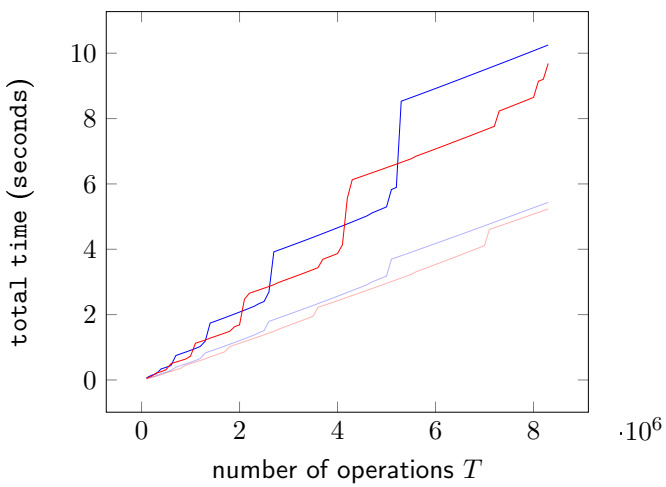
Potential solutions:

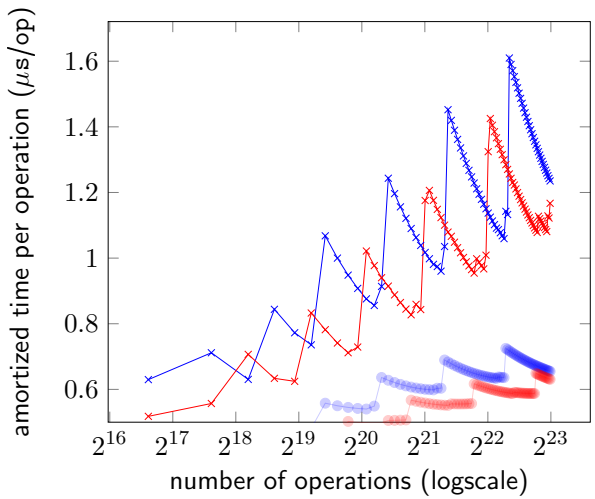
- implement real deletions

  . . . requires several changes (update chains, list of available cells)

- keep a minimum proportion of free cells after rehashing

  . . . we will implement it now in 2 minutes for ≥ 20%





Fixing the Lua hash table

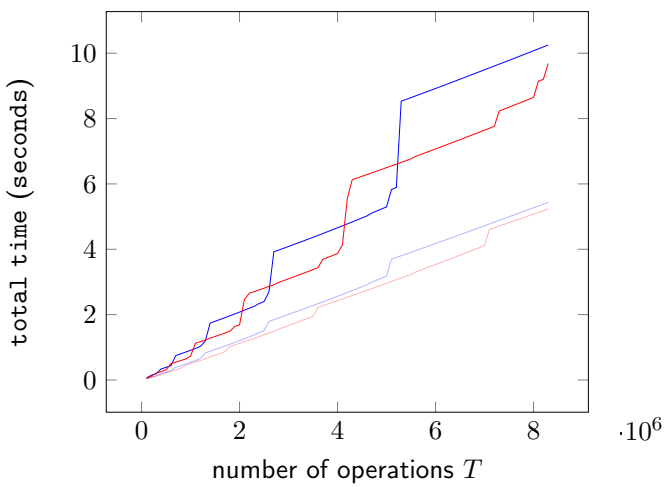
Potential solutions:

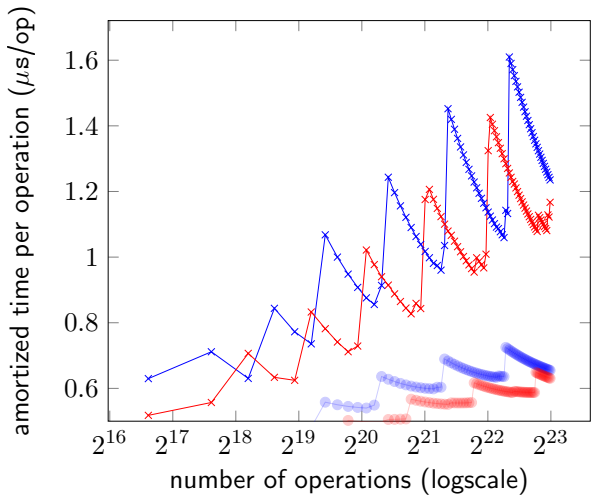
- implement real deletions

  . . . requires several changes (update chains, list of available cells)

- keep a minimum proportion of free cells after rehashing

  . . . we will implement it now in 2 minutes for ≥ 20%





Ensuring a proportion $\beta \in (0, 1)$ of empty cells

- we require at least βM operations before rehashing again

Fixing the Lua hash table

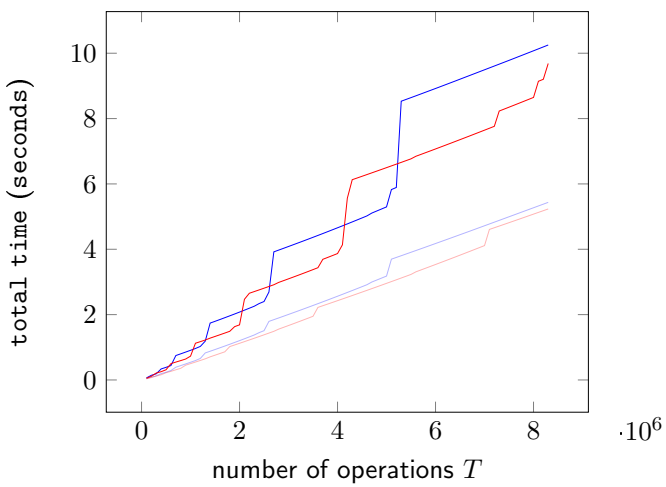
Potential solutions:

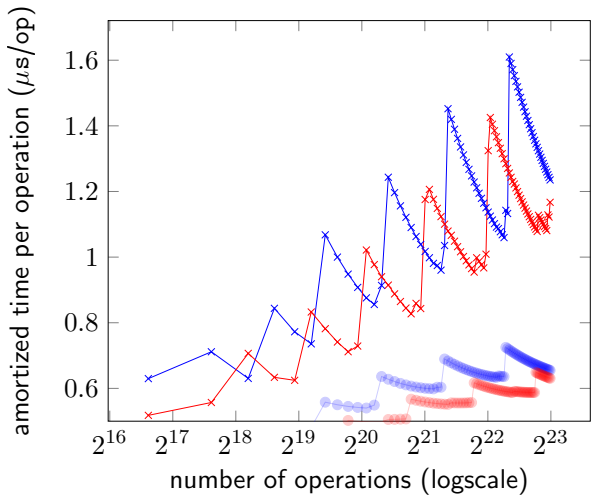
- implement real deletions

  . . . requires several changes (update chains, list of available cells)

- keep a minimum proportion of free cells after rehashing

  . . . we will implement it now in 2 minutes for ≥ 20%





Ensuring a proportion $\beta \in (0, 1)$ of empty cells

- we require at least βM operations before rehashing again
- amortized #insertions per operation $\leq (M + \beta M)/(\beta M) = 1 + \beta^{-1}$

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 $\Omega(T \log T)$,
- More precisely it is $\Theta(T \log T)$.

Hybrid Tables and insertions

- Rehashes into same size hashtables pile up to $\Omega(T \log T)$,
- More precisely it is $\Theta(T \log T)$.

And without deletions?

- Problem arises when considering effects of deletions.

Hybrid Tables and insertions

- Rehashes into same size hashtables pile up to $\Omega(T \log T)$,
- More precisely it is $\Theta(T \log T)$.

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

Hybrid Tables and insertions

- Rehashes into same size hashtables pile up to $\Omega(T \log T)$,
- More precisely it is $\Theta(T \log T)$.

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 $\Omega(T \log T)$,
- More precisely it is $\Theta(T \log T)$.

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

Hybrid Tabes mechanism: principle

- array-part corresponds to interval $[1, n]$, $n = 2^j$,

Hybrid Tabes mechanism: principle

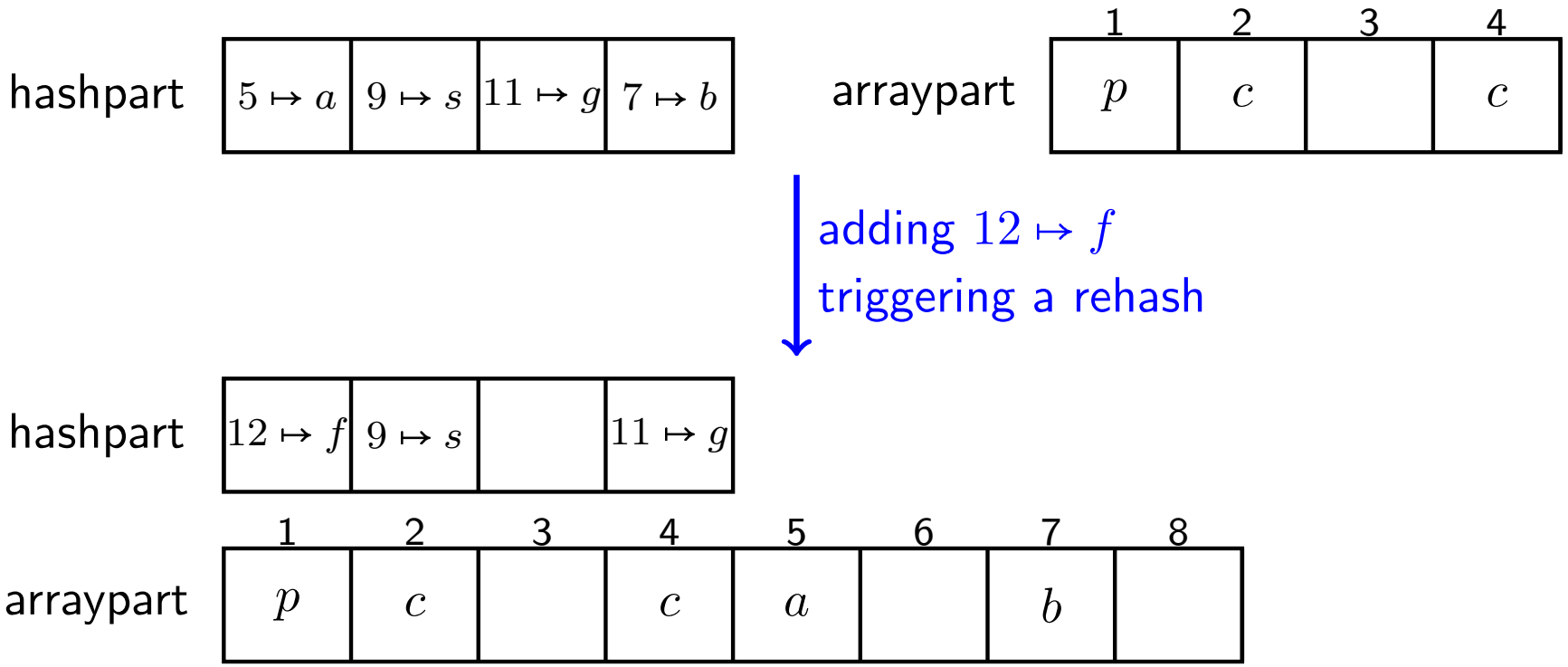
- array-part corresponds to interval $[1, n]$, $n = 2^j$,
- the interval must be more than *half-full*, $> n/2$ elements,

Hybrid Tabes mechanism: principle

- array-part corresponds to interval $[1, n]$, $n = 2^j$,
- the interval must be more than *half-full*, $> n/2$ elements,
- maximum $n = 2^j$ chosen during rehash.

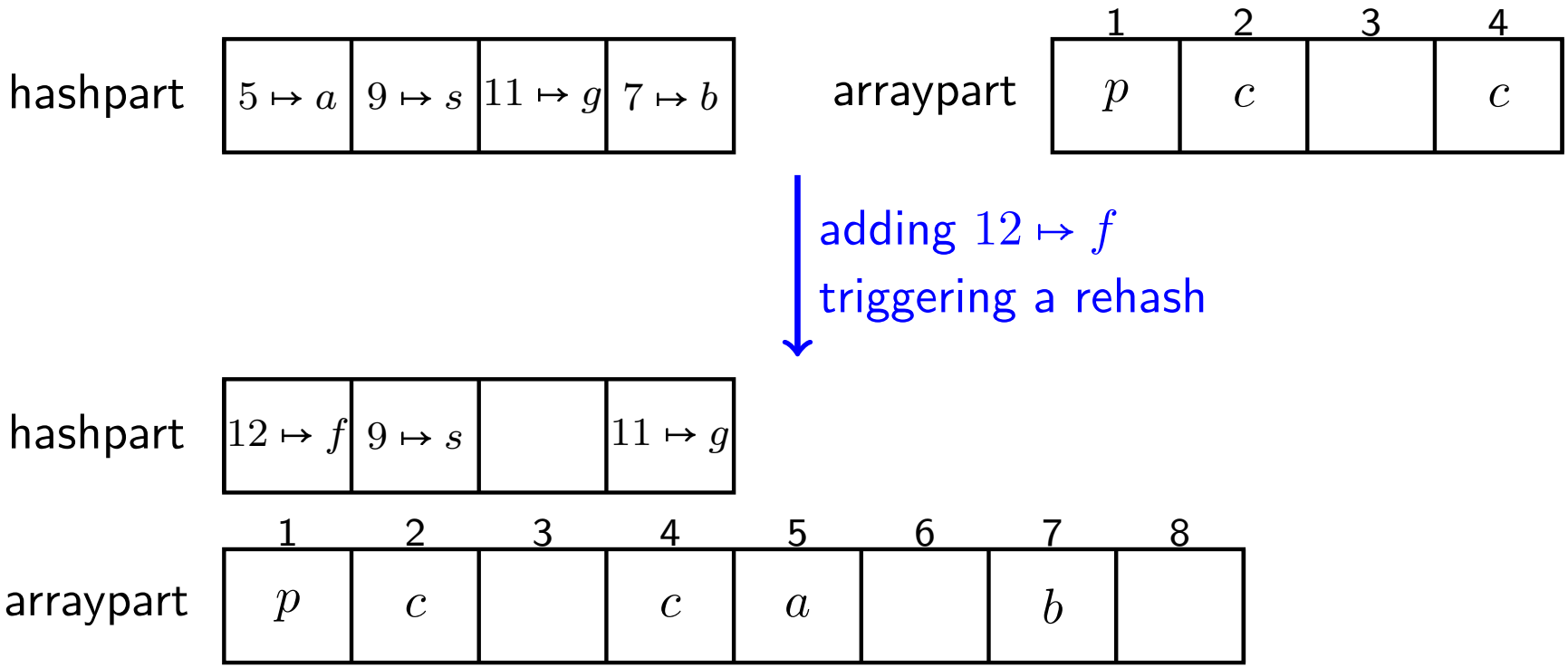
Hybrid Tabes mechanism: principle

- array-part corresponds to interval $[1, n]$, $n = 2^j$,
- the interval must be more than *half-full*, $> n/2$ elements,
- maximum $n = 2^j$ chosen during rehash.



Hybrid Tabes mechanism: principle

- array-part corresponds to interval $[1, n]$, $n = 2^j$,
- the interval must be more than *half-full*, $> n/2$ elements,
- maximum $n = 2^j$ chosen during rehash.



. . . rehash just *emulated* a deletion in the hash-part !

Hybrid Tables: $n \log n$ example 1

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

- keys $-(2^k - 1), -(2^k - 2), \ldots, 0$ go into hash-part

Hybrid Tables: $n \log n$ example 1

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

- keys $-(2^k - 1), -(2^k - 2), \ldots, 0$ go into hash-part $\Rightarrow$ $M = 2^k$ full

Hybrid Tables: $n \log n$ example 1

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

- keys $-(2^k - 1), -(2^k - 2), \ldots, 0$ go into hash-part $\Rightarrow$ $M = 2^k$ full
- key 1 induces rehash

Hybrid Tables: $n \log n$ example 1

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

- keys $-(2^k - 1), -(2^k - 2), \ldots, 0$ go into hash-part $\Rightarrow$ $M = 2^k$ full
- key 1 induces rehash ... but goes into the array-part

Hybrid Tables: $n \log n$ example 1

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

- keys $-(2^k - 1), -(2^k - 2), \ldots, 0$ go into hash-part $\Rightarrow$ $M = 2^k$ full
- key 1 induces rehash ... but goes into the array-part
  $\Rightarrow$ array-part $A = 2^0$, hash-part $M = 2^k$.

Hybrid Tables: $n \log n$ example 1

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

- keys $-(2^k - 1), -(2^k - 2), \ldots, 0$ go into hash-part $\Rightarrow$ $M = 2^k$ full
- key 1 induces rehash ... but goes into the array-part
  $\Rightarrow$ array-part $A = 2^0$, hash-part $M = 2^k$.
- key 2 induces rehash

Hybrid Tables: $n \log n$ example 1

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

- keys $-(2^k - 1), -(2^k - 2), \ldots, 0$ go into hash-part $\Rightarrow$ $M = 2^k$ full
- key 1 induces rehash ... but goes into the array-part
  $\Rightarrow$ array-part $A = 2^0$, hash-part $M = 2^k$.
- key 2 induces rehash ... and goes into the array-part

Hybrid Tables: $n \log n$ example 1

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

- keys $-(2^k - 1), -(2^k - 2), \ldots, 0$ go into hash-part $\Rightarrow$ $M = 2^k$ full
- key 1 induces rehash ... but goes into the array-part
  $\Rightarrow$ array-part $A = 2^0$, hash-part $M = 2^k$.
- key 2 induces rehash ... and goes into the array-part
  $\Rightarrow$ array-part $A = 2^1$, hash-part $M = 2^k$.

Hybrid Tables: $n \log n$ example 1

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

- keys $-(2^k - 1), -(2^k - 2), \ldots, 0$ go into hash-part $\Rightarrow$ $M = 2^k$ full
- key 1 induces rehash ... but goes into the array-part
  $\Rightarrow$ array-part $A = 2^0$, hash-part $M = 2^k$.
- key 2 induces rehash ... and goes into the array-part
  $\Rightarrow$ array-part $A = 2^1$, hash-part $M = 2^k$.
- key 3 induces rehash

Hybrid Tables: $n \log n$ example 1

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

- keys $-(2^k - 1), -(2^k - 2), \ldots, 0$ go into hash-part $\Rightarrow$ $M = 2^k$ full
- key 1 induces rehash ... but goes into the array-part
  $\Rightarrow$ array-part $A = 2^0$, hash-part $M = 2^k$.
- key 2 induces rehash ... and goes into the array-part
  $\Rightarrow$ array-part $A = 2^1$, hash-part $M = 2^k$.
- key 3 induces rehash ... and goes into the array-part

Hybrid Tables: $n \log n$ example 1

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

- keys $-(2^k - 1), -(2^k - 2), \ldots, 0$ go into hash-part $\Rightarrow M = 2^k$ full
- key 1 induces rehash ... but goes into the array-part
  $\Rightarrow$ array-part $A = 2^0$, hash-part $M = 2^k$.
- key 2 induces rehash ... and goes into the array-part
  $\Rightarrow$ array-part $A = 2^1$, hash-part $M = 2^k$.
- key 3 induces rehash ... and goes into the array-part
  $\Rightarrow$ array-part $A = 2^2$, hash-part $M = 2^k$.

Hybrid Tables: $n \log n$ example 1

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

- keys $-(2^k - 1), -(2^k - 2), \ldots, 0$ go into hash-part $\Rightarrow$ $M = 2^k$ full
- key 1 induces rehash ... but goes into the array-part
  $\Rightarrow$ array-part $A = 2^0$, hash-part $M = 2^k$.
- key 2 induces rehash ... and goes into the array-part
  $\Rightarrow$ array-part $A = 2^1$, hash-part $M = 2^k$.
- key 3 induces rehash ... and goes into the array-part
  $\Rightarrow$ array-part $A = 2^2$, hash-part $M = 2^k$.
- key 4 is inserted directly into array-part $[1, 2^2]$.

Hybrid Tables: $n \log n$ example 1

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

- keys $-(2^k - 1), -(2^k - 2), \ldots, 0$ go into hash-part $\Rightarrow$ $M = 2^k$ full
- key 1 induces rehash ... but goes into the array-part
  $\Rightarrow$ array-part $A = 2^0$, hash-part $M = 2^k$.
- key 2 induces rehash ... and goes into the array-part
  $\Rightarrow$ array-part $A = 2^1$, hash-part $M = 2^k$.
- key 3 induces rehash ... and goes into the array-part
  $\Rightarrow$ array-part $A = 2^2$, hash-part $M = 2^k$.
- key 4 is inserted directly into array-part $[1, 2^2]$.
- key 5 induces rehash

Hybrid Tables: $n \log n$ example 1

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

- keys $-(2^k-1), -(2^k-2), \ldots, 0$ go into hash-part $\Rightarrow$ $M = 2^k$ full
- key 1 induces rehash ... but goes into the array-part
  $\Rightarrow$ array-part $A = 2^0$, hash-part $M = 2^k$.
- key 2 induces rehash ... and goes into the array-part
  $\Rightarrow$ array-part $A = 2^1$, hash-part $M = 2^k$.
- key 3 induces rehash ... and goes into the array-part
  $\Rightarrow$ array-part $A = 2^2$, hash-part $M = 2^k$.
- key 4 is inserted directly into array-part $[1, 2^2]$.
- key 5 induces rehash ... and goes into the array-part

Hybrid Tables: $n \log n$ example 1

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

- keys $-(2^k-1), -(2^k-2), \ldots, 0$ go into hash-part $\Rightarrow$ $M = 2^k$ full
- key 1 induces rehash ... but goes into the array-part
  $\Rightarrow$ array-part $A = 2^0$, hash-part $M = 2^k$.
- key 2 induces rehash ... and goes into the array-part
  $\Rightarrow$ array-part $A = 2^1$, hash-part $M = 2^k$.
- key 3 induces rehash ... and goes into the array-part
  $\Rightarrow$ array-part $A = 2^2$, hash-part $M = 2^k$.
- key 4 is inserted directly into array-part $[1, 2^2]$.
- key 5 induces rehash ... and goes into the array-part
  $\Rightarrow$ array-part $A = 2^3$, hash-part $M = 2^k$.

Hybrid Tables: $n \log n$ example 1

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

- keys $-(2^k-1), -(2^k-2), \ldots, 0$ go into hash-part $\Rightarrow$ $M = 2^k$ full
- key 1 induces rehash ... but goes into the array-part
  $\Rightarrow$ array-part $A = 2^0$, hash-part $M = 2^k$.
- key 2 induces rehash ... and goes into the array-part
  $\Rightarrow$ array-part $A = 2^1$, hash-part $M = 2^k$.
- key 3 induces rehash ... and goes into the array-part
  $\Rightarrow$ array-part $A = 2^2$, hash-part $M = 2^k$.
- key 4 is inserted directly into array-part $[1, 2^2]$.
- key 5 induces rehash ... and goes into the array-part
  $\Rightarrow$ array-part $A = 2^3$, hash-part $M = 2^k$.

  ...
- keys $2^j + 1, j \geq 0$, induce rehash, rest inserted in array-part directly

Hybrid Tables: $n \log n$ example 1

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

- keys $-(2^k-1), -(2^k-2), \ldots, 0$ go into hash-part $\Rightarrow$ $M = 2^k$ full
- key 1 induces rehash ... but goes into the array-part
  $\Rightarrow$ array-part $A = 2^0$, hash-part $M = 2^k$.
- key 2 induces rehash ... and goes into the array-part
  $\Rightarrow$ array-part $A = 2^1$, hash-part $M = 2^k$.
- key 3 induces rehash ... and goes into the array-part
  $\Rightarrow$ array-part $A = 2^2$, hash-part $M = 2^k$.
- key 4 is inserted directly into array-part $[1, 2^2]$.
- key 5 induces rehash ... and goes into the array-part
  $\Rightarrow$ array-part $A = 2^3$, hash-part $M = 2^k$.

  ...
- keys $2^j + 1, j \geq 0$, induce rehash, rest inserted in array-part directly

$\Longrightarrow$ time $\Omega(k \cdot 2^k)$

Hybrid Tables: $n \log n$ example 2

A permutation of $[1, \ldots, n]$

Permutations of $[1, \ldots, n]$ are a natural choice:

- they come up in practice in many contexts,
- they give the array-part a good shot at being used, yet ...

Hybrid Tables: $n \log n$ example 2

A permutation of $[1, \ldots, n]$

Permutations of $[1, \ldots, n]$ are a natural choice:

- they come up in practice in many contexts,
- they give the array-part a good shot at being used, yet ...

Example: inserting $2 \cdot 2^k + 1, 2 \cdot 2^k + 2, \ldots, 3 \cdot 2^k, 1, 2, \ldots, 2^k$

- mechanism is exactly as in previous example,
- keys $2 \cdot 2^k + 1, 2 \cdot 2^k + 2, \ldots, 3 \cdot 2^k$ go into hash-part

Hybrid Tables: $n \log n$ example 2

A permutation of $[1, \ldots, n]$

Permutations of $[1, \ldots, n]$ are a natural choice:

- they come up in practice in many contexts,
- they give the array-part a good shot at being used, yet ...

Example: inserting $2 \cdot 2^k + 1, 2 \cdot 2^k + 2, \ldots, 3 \cdot 2^k, 1, 2, \ldots, 2^k$

- mechanism is exactly as in previous example,
- keys $2 \cdot 2^k + 1, 2 \cdot 2^k + 2, \ldots, 3 \cdot 2^k$ go into hash-part $\Rightarrow M = 2^k$ full,

Hybrid Tables: $n \log n$ example 2

A permutation of $[1, \ldots, n]$

Permutations of $[1, \ldots, n]$ are a natural choice:

- they come up in practice in many contexts,
- they give the array-part a good shot at being used, yet ...

Example: inserting $2 \cdot 2^k + 1, 2 \cdot 2^k + 2, \ldots, 3 \cdot 2^k, 1, 2, \ldots, 2^k$

- mechanism is exactly as in previous example,
- keys $2 \cdot 2^k + 1, 2 \cdot 2^k + 2, \ldots, 3 \cdot 2^k$ go into hash-part $\Rightarrow M = 2^k$ full,
- keys $1, 2, 3, 5, 9, \ldots$ induce rehash, while rest inserted into array-part

Hybrid Tables: $n \log n$ example 2

A permutation of $[1, \ldots, n]$

Permutations of $[1, \ldots, n]$ are a natural choice:

- they come up in practice in many contexts,
- they give the array-part a good shot at being used, yet ...

Example: inserting $2 \cdot 2^k + 1, 2 \cdot 2^k + 2, \ldots, 3 \cdot 2^k, 1, 2, \ldots, 2^k$

- mechanism is exactly as in previous example,
- keys $2 \cdot 2^k + 1, 2 \cdot 2^k + 2, \ldots, 3 \cdot 2^k$ go into hash-part $\Rightarrow$ $M = 2^k$ full,
- keys $1, 2, 3, 5, 9, \ldots$ induce rehash, while rest inserted into array-part

... but this is rather unlikely for a permutation

Hybrid Tables: random permutation

Random model starting from an empty table:

- Consider random permutation π of $[n]$

Hybrid Tables: random permutation

Random model starting from an empty table:

- Consider random permutation π of $[n]$
- Insert elements in order $\pi(1), \pi(2) \ldots, \pi(n)$.

Hybrid Tables: random permutation

Random model starting from an empty table:

- Consider random permutation π of $[n]$
- Insert elements in order $\pi(1), \pi(2) \ldots, \pi(n)$.

Hybrid Tables: random permutation

Random model starting from an empty table:

- Consider random permutation π of $[n]$
- Insert elements in order $\pi(1), \pi(2) \ldots, \pi(n)$.

Fix $c < \frac{1}{2}$ and consider an arbitrary $g(n) \to \infty$, e.g., $g(n) = \log^*(n)$.

Lemma

For every time $t \leq cn$, none of $S_j = [1, 2^j]$ for $2^j \geq g(n)$ is half-full with probability tending to one.

Hybrid Tables: random permutation

Random model starting from an empty table:

- Consider random permutation π of $[n]$
- Insert elements in order $\pi(1), \pi(2) \ldots, \pi(n)$.

Fix $c < \frac{1}{2}$ and consider an arbitrary $g(n) \to \infty$, e.g., $g(n) = \log^*(n)$.

Lemma

For every time $t \leq cn$, none of $S_j = [1, 2^j]$ for $2^j \geq g(n)$ is half-full with probability tending to one.

$\Longrightarrow$ array part not really used

Hybrid Tables: random permutation

Random model starting from an empty table:

- Consider random permutation π of $[n]$
- Insert elements in order $\pi(1), \pi(2) \ldots, \pi(n)$.

Fix $c < \frac{1}{2}$ and consider an arbitrary $g(n) \to \infty$, e.g., $g(n) = \log^*(n)$.

Lemma

For every time $t \leq cn$, none of $S_j = [1, 2^j]$ for $2^j \geq g(n)$ is half-full with probability tending to one.

$\Longrightarrow$ array part not really used and complexity essentially linear

Theorem (Martínez,Nicaud,R 2022)

Inserting random permutation takes time $\mathcal{O}(n\, g(n))$ provided that n does not approximate powers of two from above[a].

[a] Fix $b \in (1,2)$, $n \in \mathbb{N}_b := \bigcup_{j \geq 0} \{k : 2^j b < k \leq 2^{j+1}\}$.

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!