Modern hashing for alignment-free sequence analysis

Part 2: Hashing, hash functions collision resolution

Jens Zentgraf & Sven Rahmann

GCB 2021

Hashing

Idea: Store several keys K in space slightly larger than necessary,
to get fast ("constant-time") access to each key
(check existence, retrieve associated value, …)

Ingredients:

- Set ("universe") U of possible keys
- Set of keys $K \subseteq U$ to be stored, $|K| = N$
- Hash table (array) with P slots
- Hash function $h\colon\ U \to \{0, \dots, P - 1\}$
- Collision resolution strategy (details later)

Hashing Example

- Universe U: all possible first names (of finite length)
- Set of keys $K = \{Anna, Franz, Bea, Fritz\}$
- Hash table with 6 slots
- Some function *h* mapping *U* to {0, 1, 2, 3, 4, 5}.

Hashing

h("Anna") = 3

h("Franz") = 5

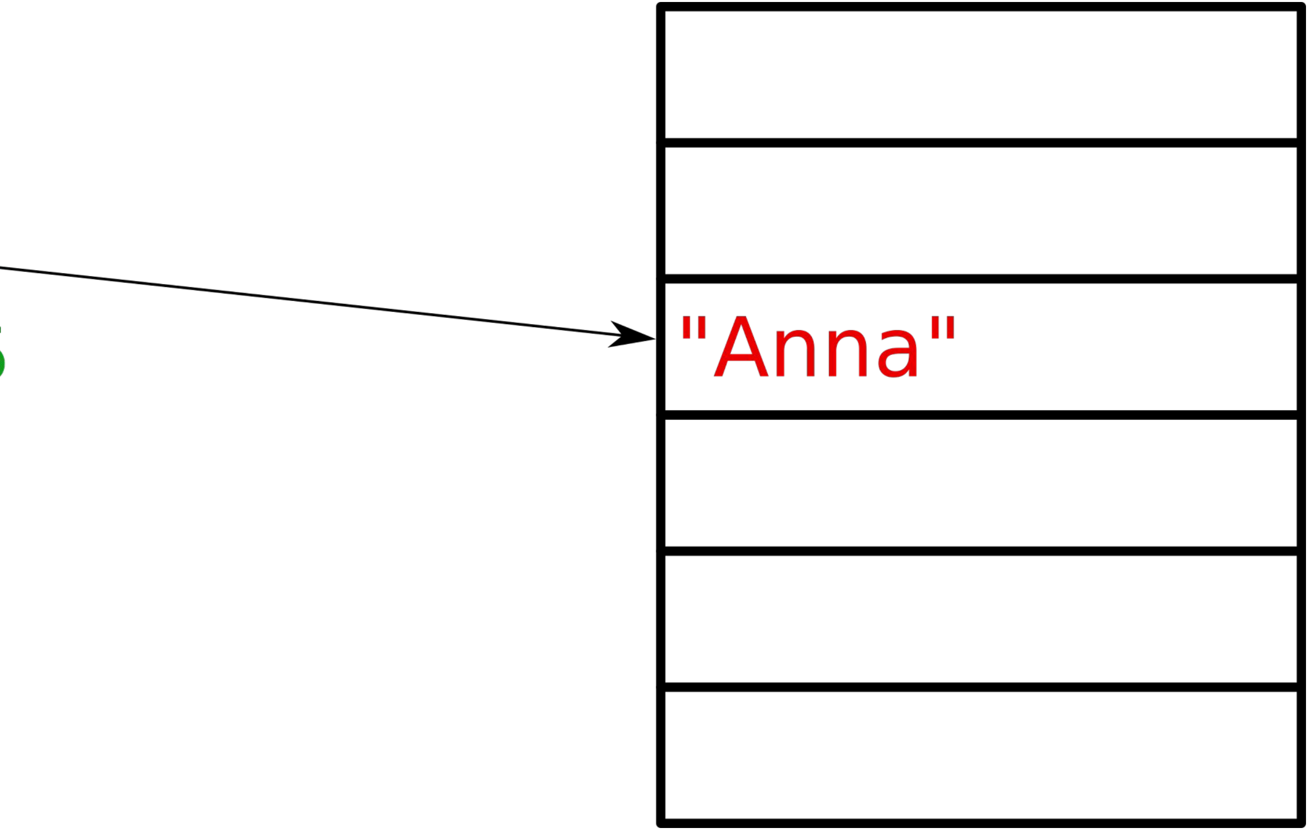
h("Bea") = 2

Hashing

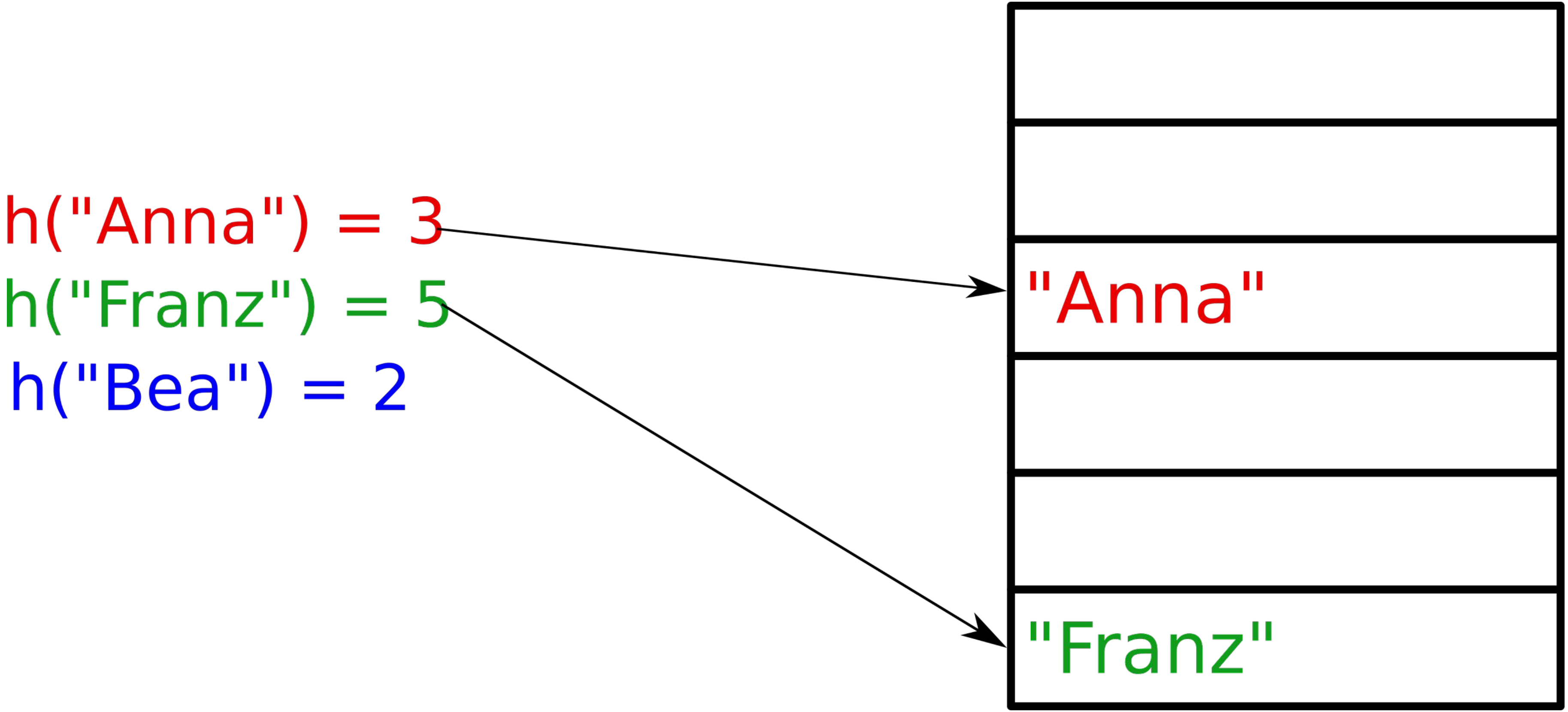
h("Anna") = 3

h("Franz") = 5

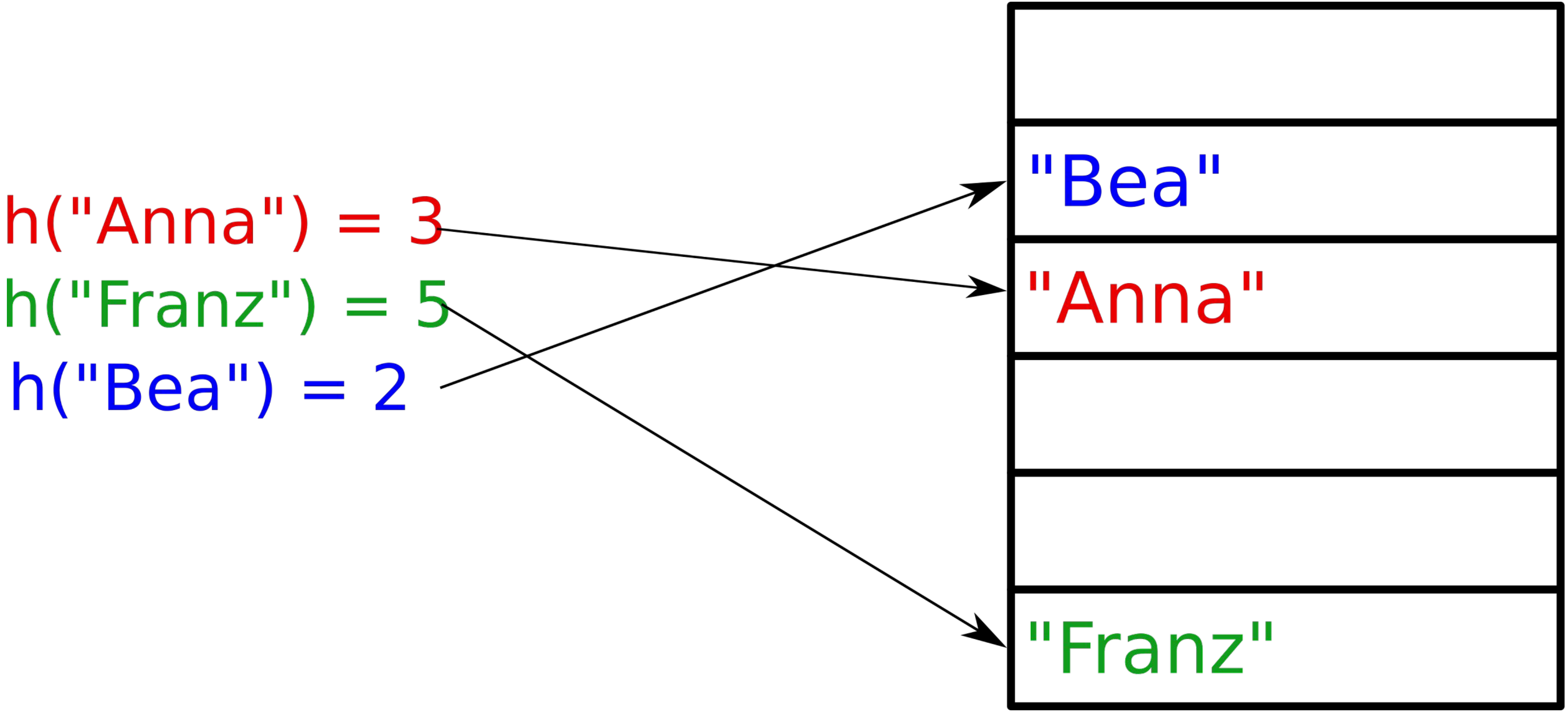
h("Bea") = 2



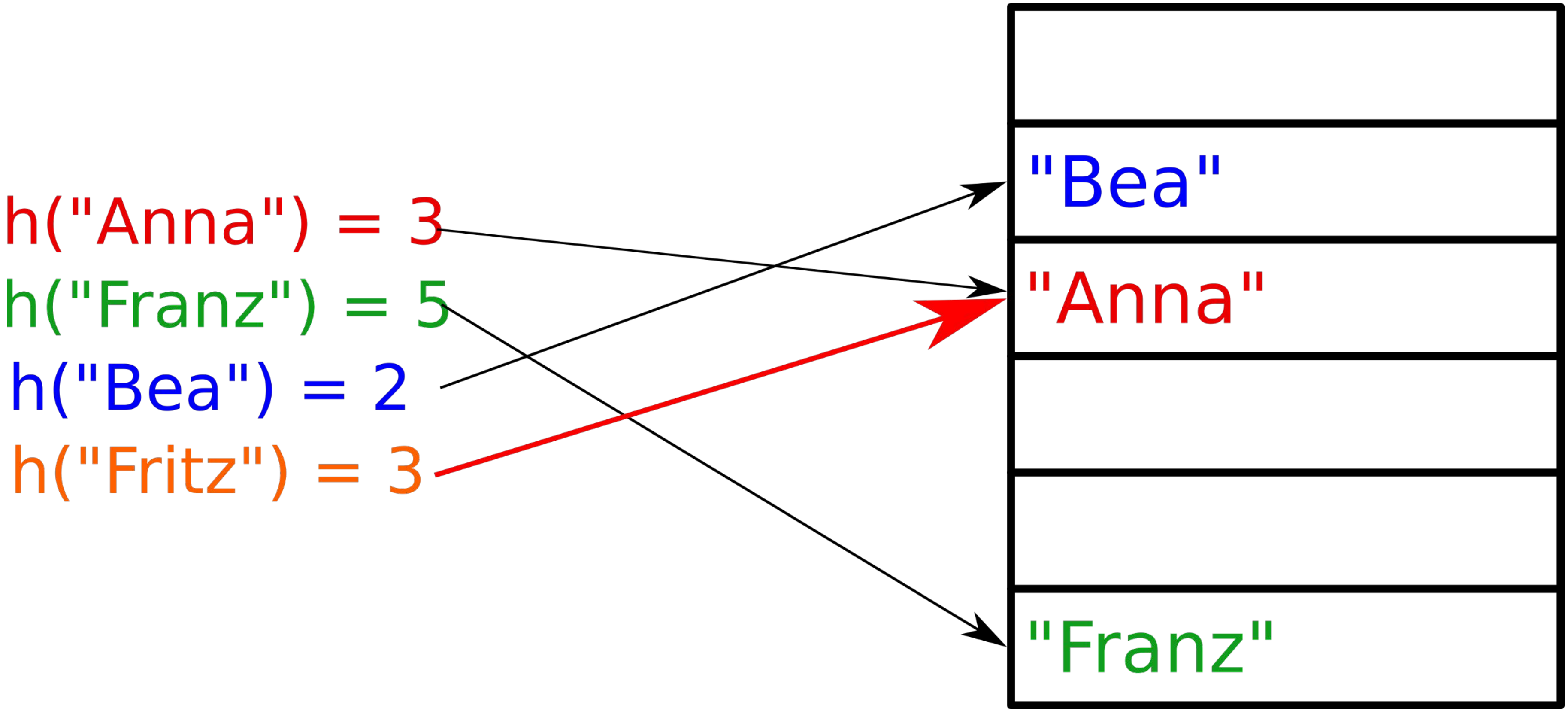
Hashing



Hashing



Hashing: Collision



Hash functions on DNA (and k-mers)

Definitions (hash function f on k-mers for a hash table of size P):

$$f : U \rightarrow \{0, 1, \ldots, P\text{-}1\}$$

- P: table (array) size
- U: universe of all possible keys (here: k-mers for fixed k)
- In concrete applications, f is restricted to actual key set $K \subseteq U$, written $f|_K$
- $f(x) = f(y)$ for $x \neq y$: collision occurs, x and y hash to same location (slot)
- $f|_K$ injective (no collisions on K): perfect hashing (usually when $P >> |K|$)
- $f|_K$ injective and $|K|=P$: minimal perfect hashing.

Encodings (codes) as hash functions ?

Observations:

- k-mer encoding, canonical code,
- any xor-ed (canonical) code with bit mask of 2k bits

are already hash functions of DNA k-mers into $\{0, 1, \ldots, 4^k-1\}$
(perfect hashing!).

However, requires a huge hash table with 4^k slots.
Typically, there are only $|K| = n \ll 4^k$ k-mers in an observed k-mer set K.

Assumption: Hash table size P with $n \leq P \ll 4^k$

Codes mod P as hash functions?

Assumption: Hash table size P with $|K| \leq P \ll 4^k$.

Proposal: $f(x) := \text{ccode}(x) \bmod P$
(remainder of canonical code after division by P)

Properties:

- good: same hash value for x and x's reverse complement
- bad: not flexible (no free parameters)
- bad: may show bias in distribution (non-uniform distribution across slots)

We want close-to-uniform distribution (few collisions),
even if K is an "adversarial" set of k-mers.

Using "standard" hash functions

Idea:

- Take a general-purpose hash function (for bytes/strings) from the internet
- Check that it outputs deterministic 64-bit values
- Take hash value mod P

Examples:

- MurmurHash2A (https://en.wikipedia.org/wiki/MurmurHash): 64 bits
- CityHash (google): on byte arrays (like tabulation hashing)
- FarmHash (google): on byte arrays (like tabulation hashing)

Note: Non-cryptographic (i.e easily invertible) hash functions are o.k here!

Tabulation Hashing

- Interpret $(2k)$-bit k-mer as vector of bytes (8-bit units)

  e.g. 23-mer = 46 bits = (almost) 6 bytes
- Write k-mer $x = (x_0, x_1, \ldots, x_5)$ as 6 bytes
- For each byte i, initialize a random table T_i of $2^8 = 256$ hash values (64 bits)
- Compute hash value $f(x) := (T_0[x_0] \oplus T_1[x_1] \oplus \ldots \oplus T_5[x_5]) \bmod P$

```
[[14442894551235957847,  6469421870737703710, 17438115425174121141, 16991427295943144545,  5025173426410057567,  2317480815800434420],
 [ 1075541066489945786, 16107346375557295288, 17156970515669299684, 10537353710669269530, 15753615170332459648, 10539149035937080374],
 [ 6352942000340010737,  9669198019791798240,  1767568531891784628,  3750932900196831156,  9556395753393875347,   977488244050155625],
 [16453330187979092062, 11455894856113951056,  2404176692238009439, 13446275190310080616,  7683026793024744608,  2566917295258172729],
 [ 4374697865087216574,  2948501028054143870, 15709718051506091967,   567219589563681675, 17558014467592015574, 11339424135516353160],
 [15543508203406612480, 17259474578117683833, 18184025842706544980,  6980811793600737347,  8184392962606199987,  8345167768415290135],
 ...,
 [ 7480629086903632978,  6612313070443330511, 17946777377357279890, 17810500917063643000, 14908903716162607610,   504214794497660276],
 [ 1776935294473424064, 15852374736013488866,  7297866075249847808,  4478248790474069837, 12275672329074345192,  8404072098603978824],
 [ 5882335407069488499,  2624558843767794939,  4675036349449086901, 10781893044502755034,  3795591906282680441, 13643704536747094467],
 [ 6875195240740202979,  7208988375847104790, 10160858921198161389,  8674721880753872424,  5612573330011873863,  3297829263140205588],
 [ 1976455990114768357,  4558317174298096864,  9674752687375864252,   776890991137020788, 10850188664737495916, 16956178566493365890],
 [13236883617738096169,  3170490034065544057,  5620456850890136315,  2738207822322483209,   427591245031707729, 14353683807222710284]]
```

Tabulation Hashing: Notes

```
[[14442894551235957847,  6469421870737703710, 17438115425174121141, 16991427295943144545,  5025173426410057567,  2317480815800434420],
 [ 10755410664899457 86, 16107346375557295288, 17156970515669299684, 10537353710669269530, 15753615170332459648, 10539149035937080374],
 [ 6352942000340010737,  9669198019791798240,  1767568531891784628,  3750932900196831156,  9556395753393875347,   977488244050155625],
 [16453330187979092062, 11455894856113951056,  2404176692238009439, 13446275190310080616,  7683026793024744608,  2566917295258172729],
 [ 4374697865087216574,  2948501028054143870, 15709718051506091967,   567219589563681675, 17558014467592015574, 11339424135516353160],
 [15543508203406612480, 17259474578117683833, 18184025842706544980,  6980811793600737347,  8184392962606199987,  8345167768415290135],
 ...,
 [ 7480629086903632978,  6612313070443330511, 17946777377357279890, 17810500917063643000, 14908903716162607610,   504214794497660276],
 [ 1776935294473424064, 15852374736013488866,  7297866075249847808,  4478248790474069837, 12275672329074345192,  8404072098603978824],
 [ 5882335407069488499,  2624558843767794939,  4675036349449086901, 10781893044502755034,  3795591906282680441, 13643704536747094467],
 [ 6875195240740202979,  7208988375847104790, 10160858921198161389,  8674721880753872424,  5612573330011873863,  3297829263140205588],
 [ 1976455990114768357,  4558317174298096864,  9674752687375864252,   776890991137020788, 10850188664737495916, 16956178566493365890],
 [13236883617738096169,  3170490034065544057,  5620456850890136315,  2738207822322483209,   427591245031707729, 14353683807222710284]]
```

- Compute hash value $f(x) := (T_0[x_0] \oplus T_1[x_1] \oplus \ldots \oplus T_5[x_5]) \bmod P$
- Hash values can have any number of bits (typically 64); operation "mod P" is finally applied to obtain range $\{0, \ldots, P-1\}$.
- Other units than bytes (8 bits) can be used; e.g. 4 bits or 16 bits; larger units mean much larger (but slightly fewer) tables.
- strong theoretical properties (3-independence).
- **Disadvantage:** Large space requirement for table

ntHash: specialized DNA hashing

- rolling hash function (like k-mer encoding):
  let x_1, x_2, … be the successive overlapping k-mers
  compute hash value $H(x_i)$ from: $H(x_{i-1})$, removed base, new base
  by updating in constant time instead of re-reading k basepairs.
- special form of tabulation hashing:
  one table with (specially crafted) "random" hash values for each basepair
- Update: $\boldsymbol{H(x_i) = \mathrm{rol}^1(H(x_{i-1})) \oplus \mathrm{rol}^k(h(s[\mathrm{i}-1])) \oplus h(s[i+k-1])}$
  "Hash value for x_i is hash value of x_{i-1}, rotated left by 1 bit,
  xor-ed with the tabulated value for the outgoing base s[i-1], rotated left by k bits,
  then xor-ed with the tabulated value for the incoming base $s[i+k-1]$ as is."

Hamid Mohamadi, Justin Chu, Benjamin P. Vandervalk, Inanc Birol, ntHash: recursive nucleotide hashing.
Bioinformatics, Volume 32, Issue 22, 15 November 2016, Pages 3492–3494. (https://doi.org/10.1093/bioinformatics/btw397)

Randomized Rotate-Multiply-Offset

Proposal (bit rotation, randomization): Pick two integers

- multiplier a odd in $\{1, 3, \ldots, 4^k\text{-}1\}$,
- offset b in $\{0, 1, 2, \ldots, 4^k\text{-}1\}$;

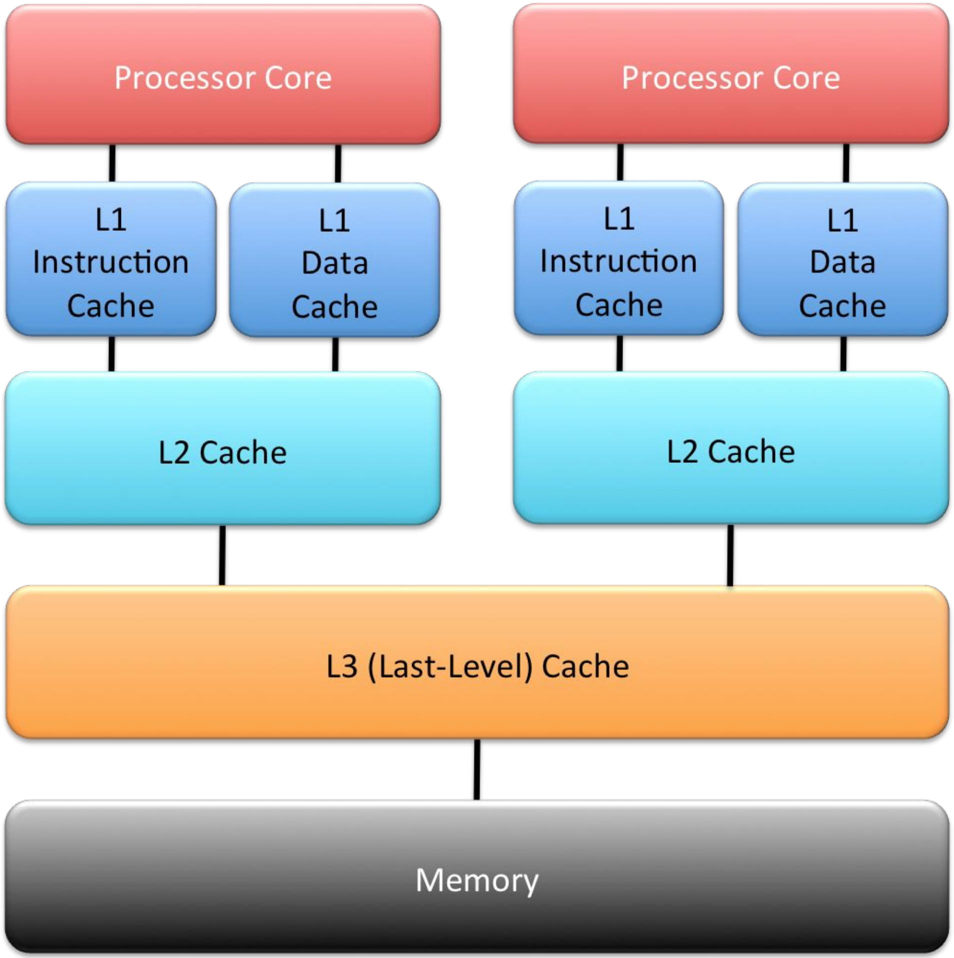
$$f(x) := [(a\ \mathrm{rot}_k(\mathrm{ccode}(x)) + b) \bmod 4^k\ ] \bmod P$$

- rot_k: cyclic rotation by k bits: inner bits outside, outer bits inside.

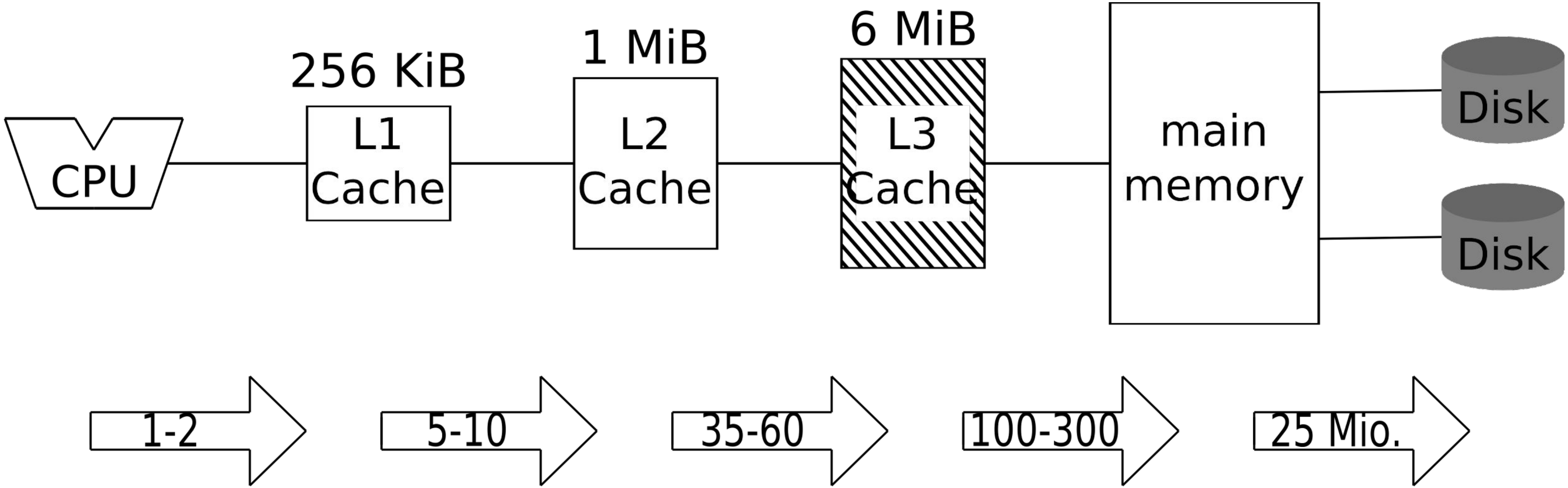
Good properties:

- same hash value for x and x's reverse complement
- The part in [...] is a random **bijection** on the universe U (if $|U|$ is a power of 2)
- If biased, just pick different random a, b.

CPU caches

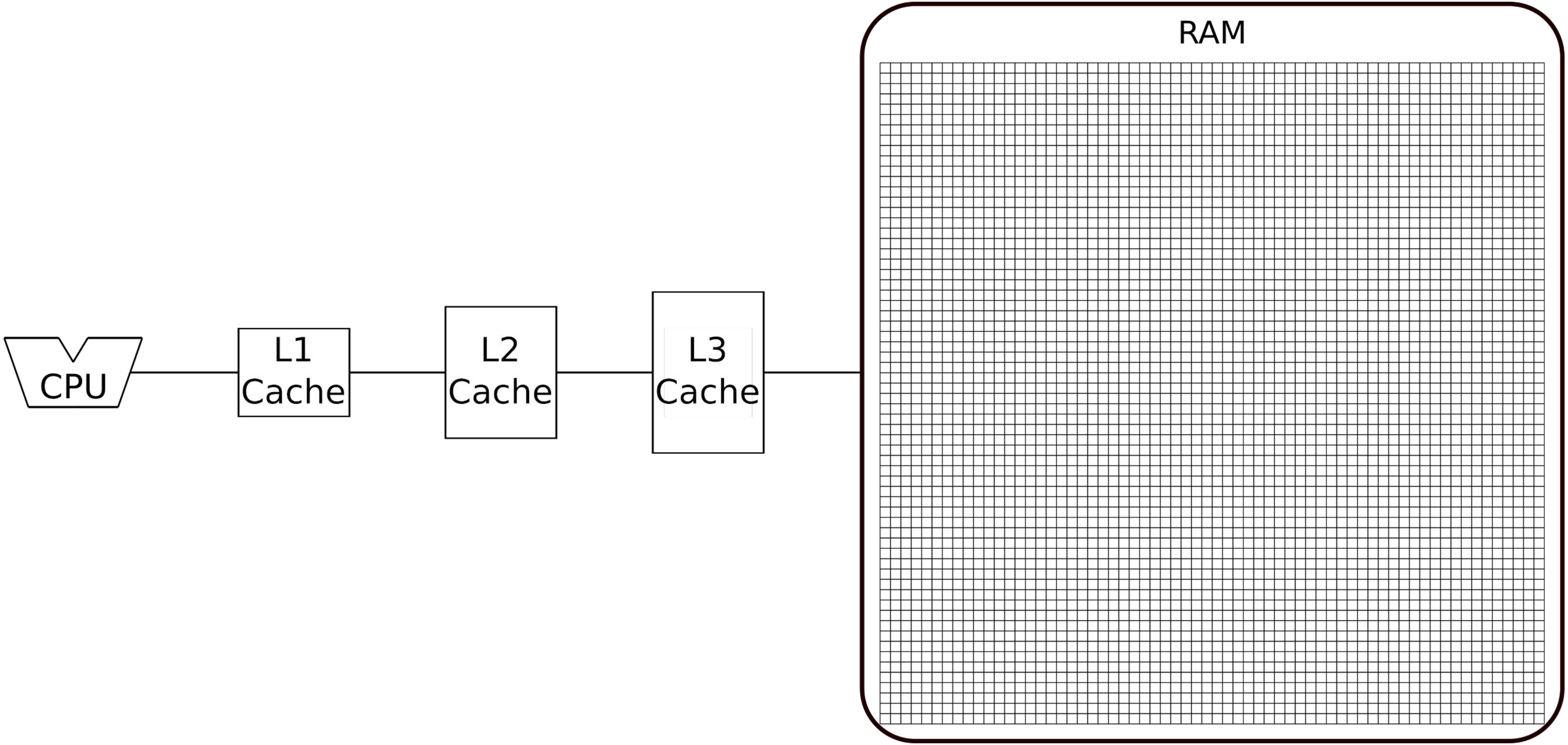

Processor Core
Processor Core
L1 Instruction Cache
L1 Data Cache
L1 Instruction Cache
L1 Data Cache
L2 Cache
L2 Cache
L3 (Last-Level) Cache
Memory

CPU caches

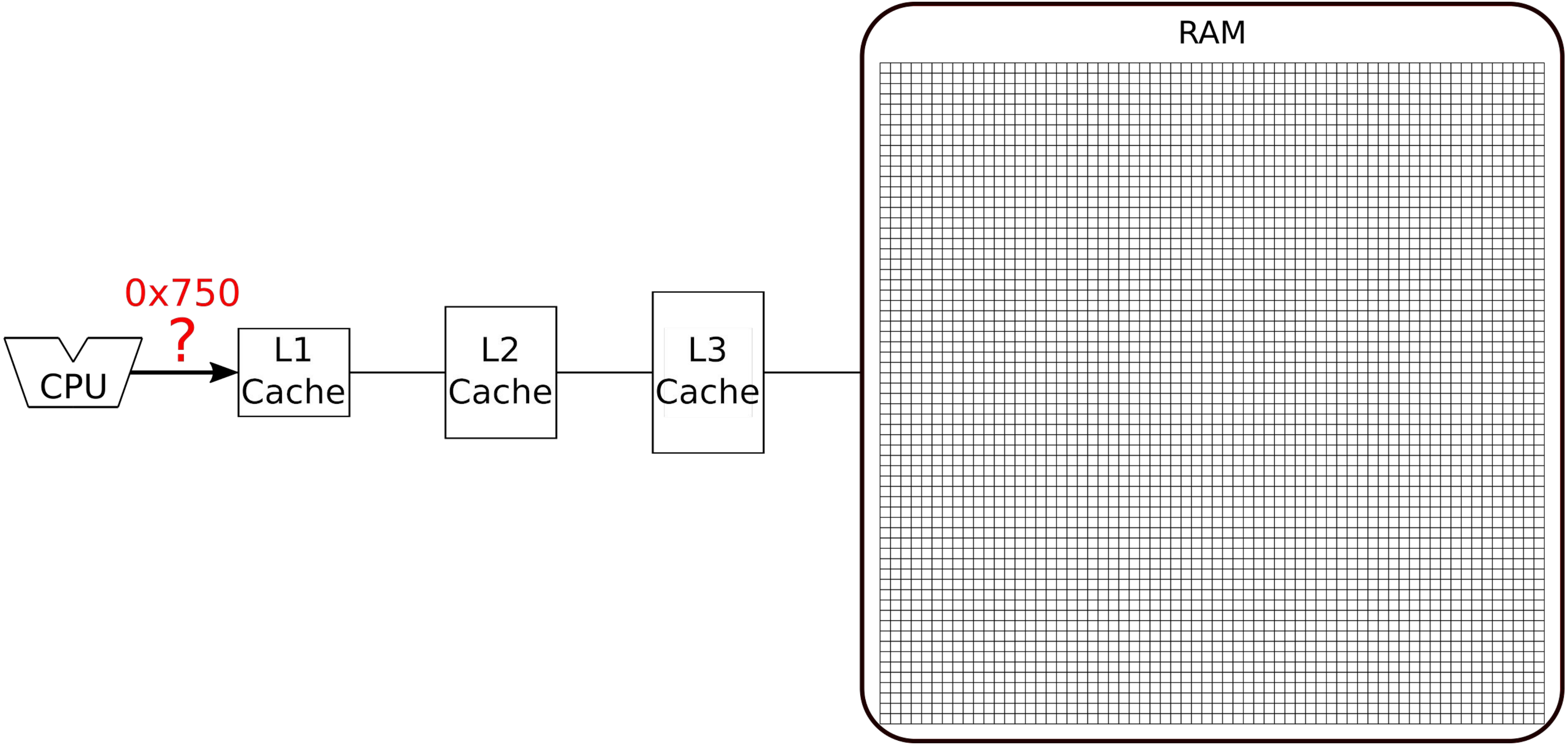


This and following illustrations by Uriel Elias Wiebelitz (TU Dortmund)

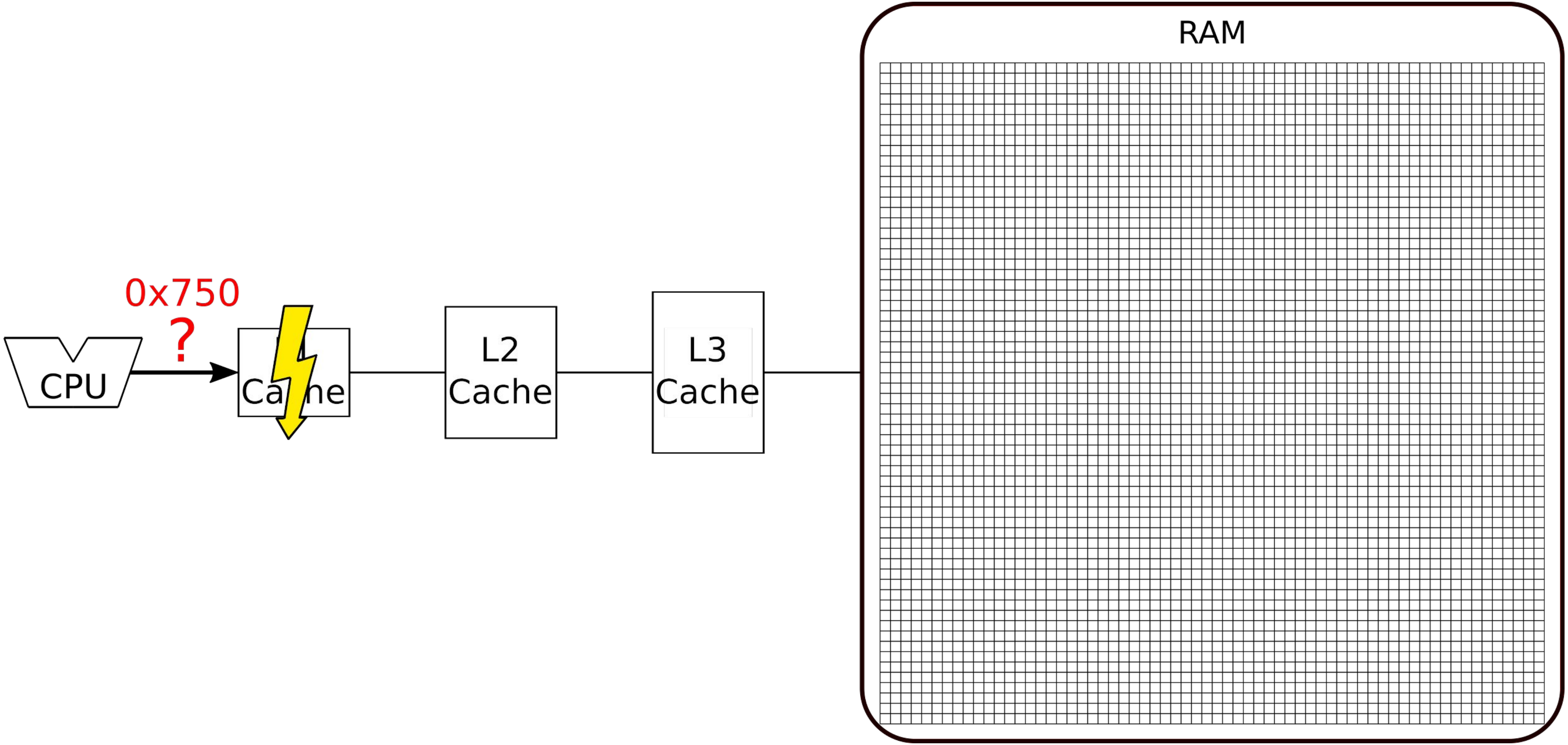
Cache line



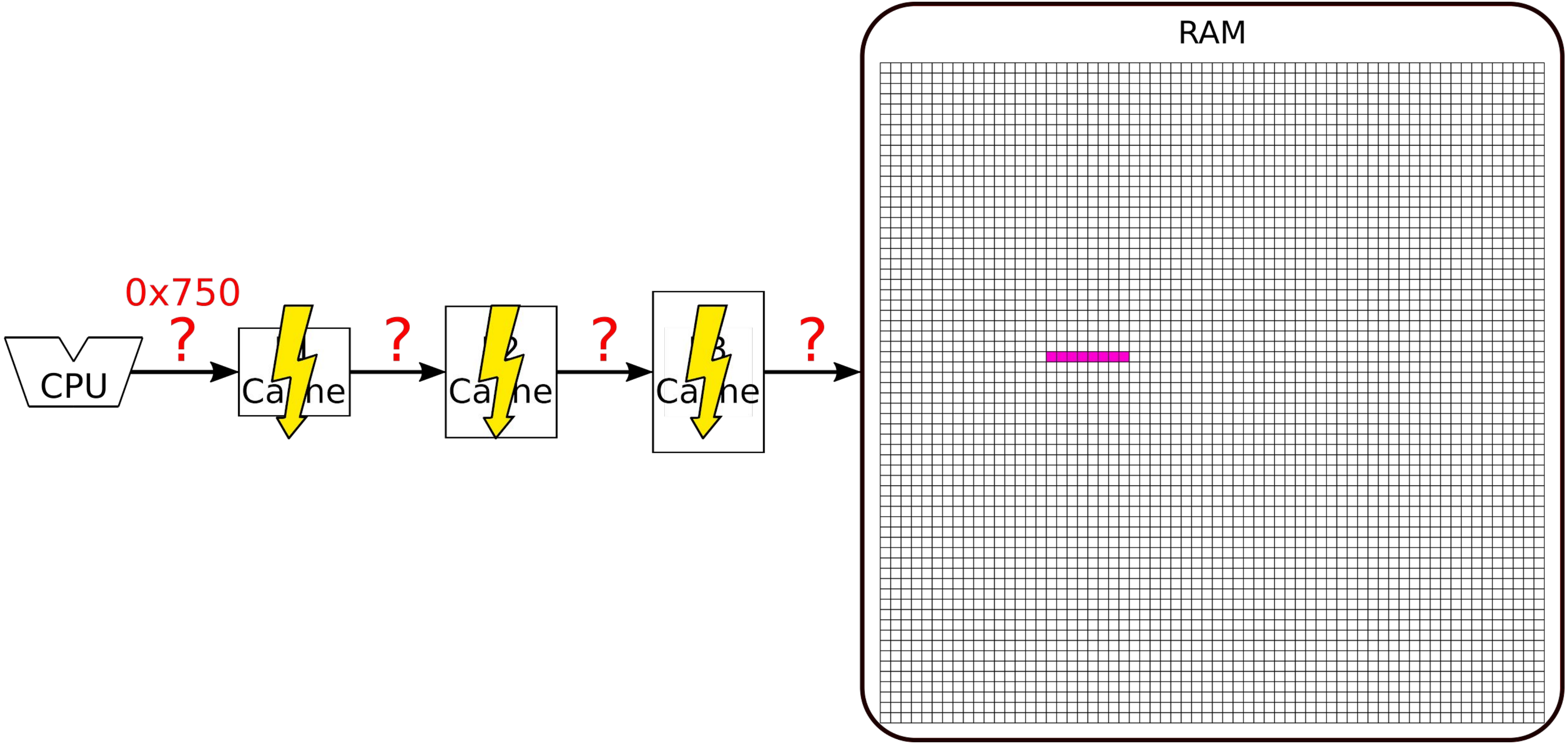
Cache line



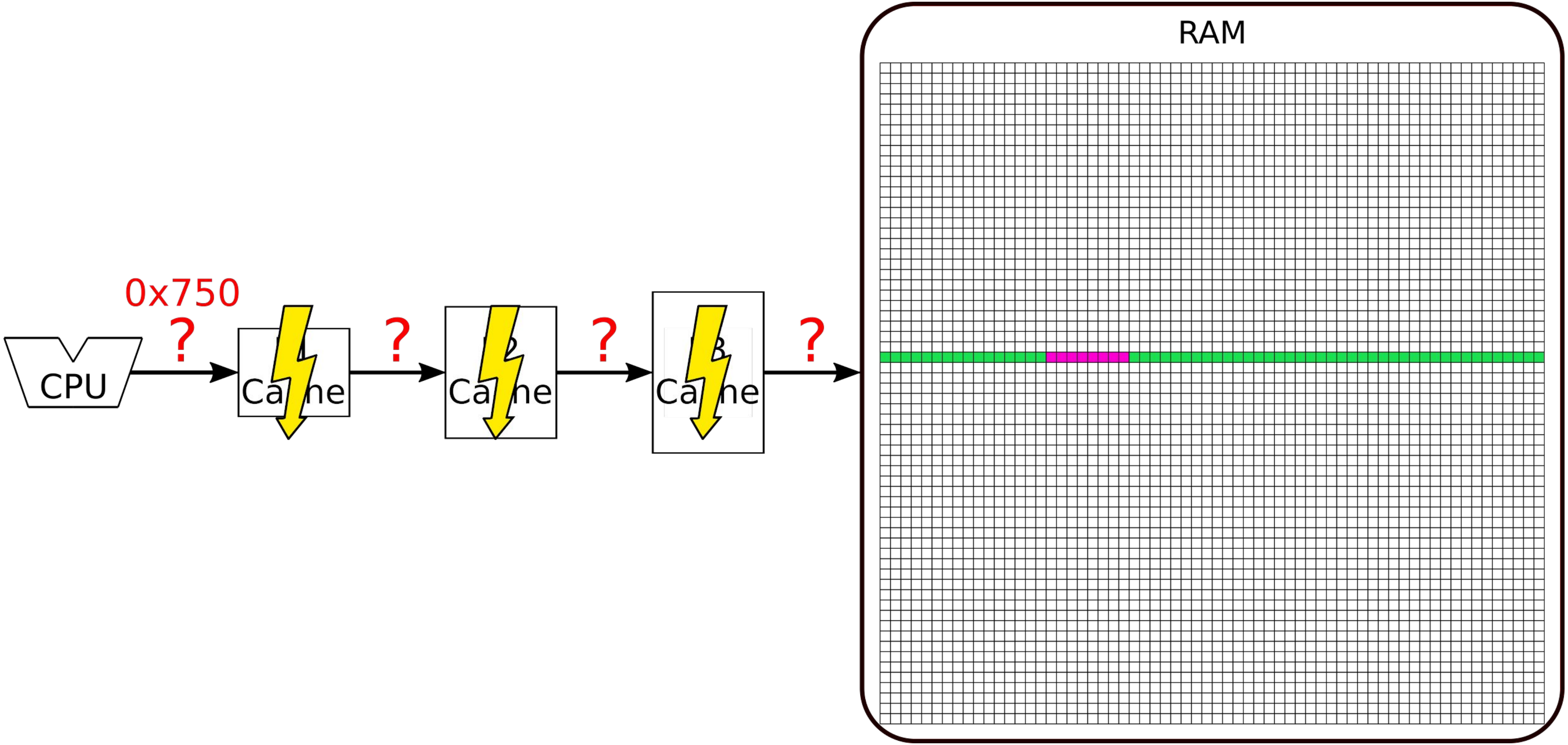
Cache line



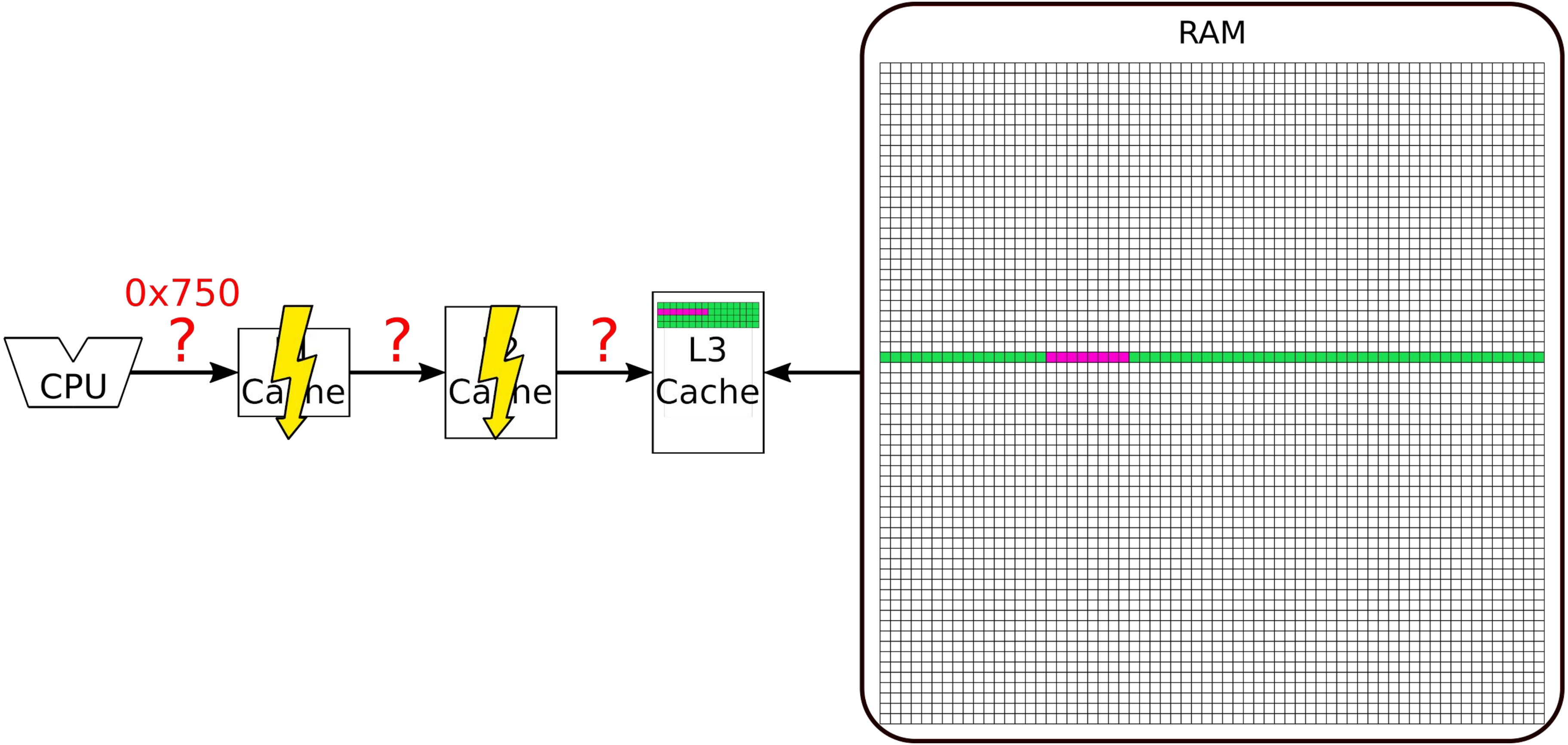
Cache line



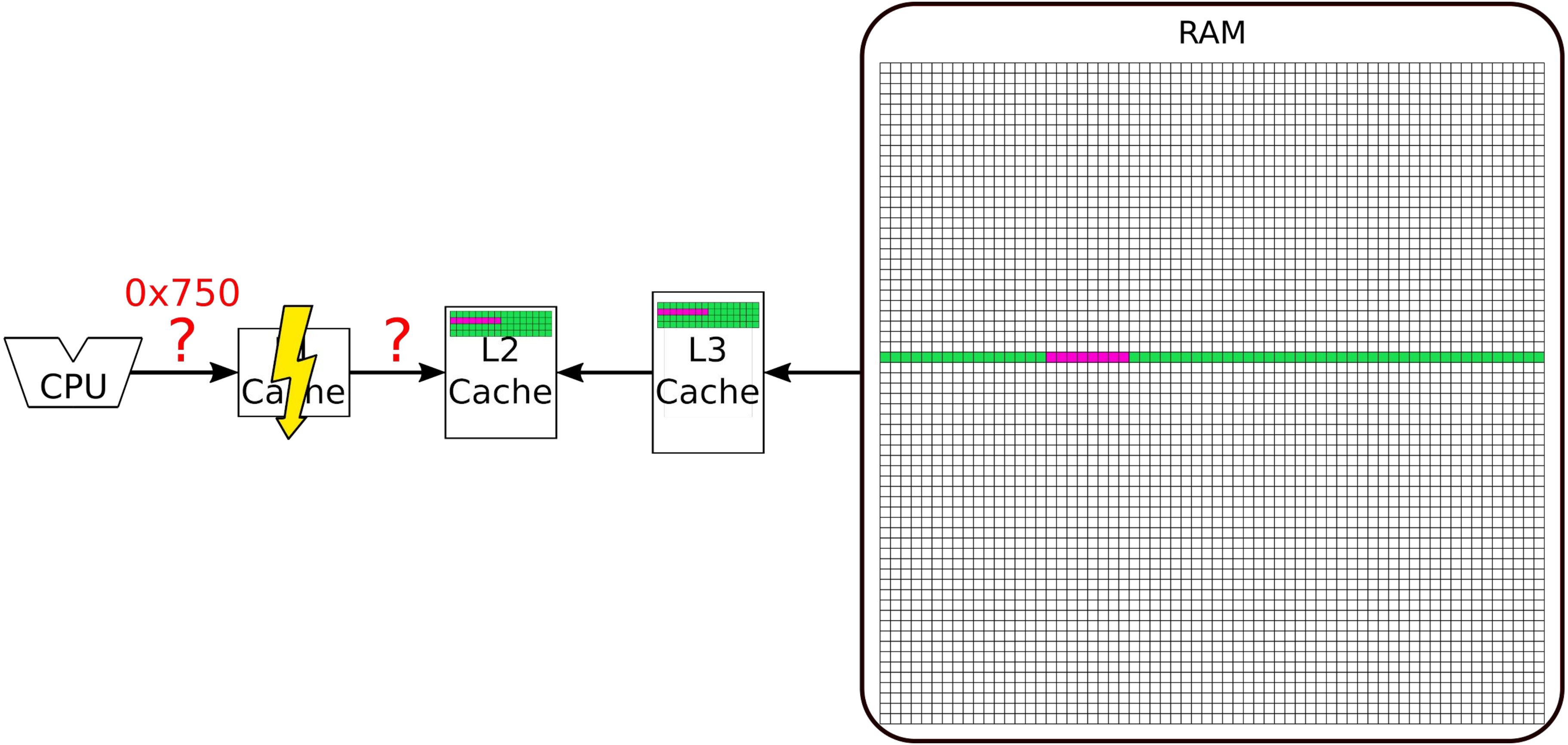
Cache line



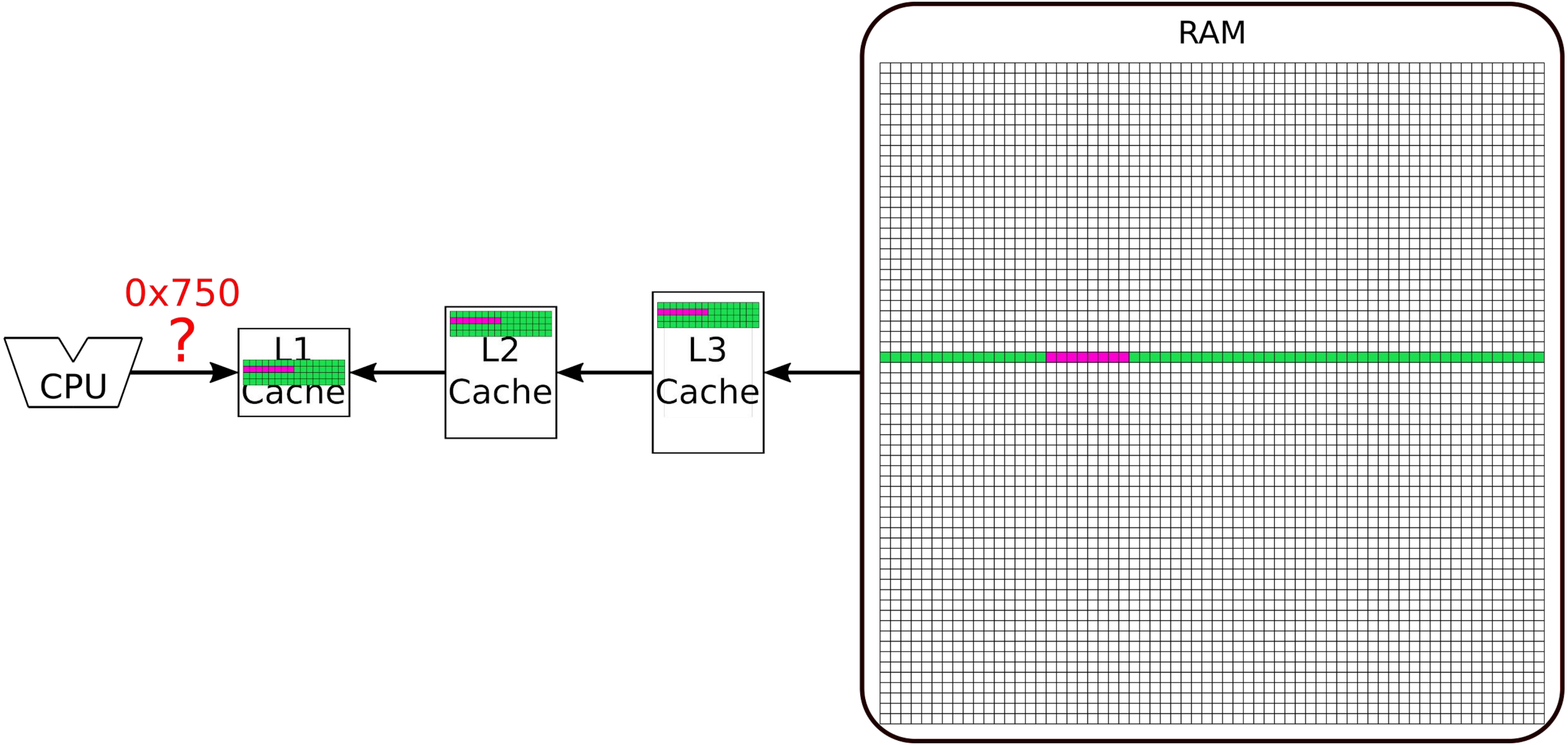
Cache line



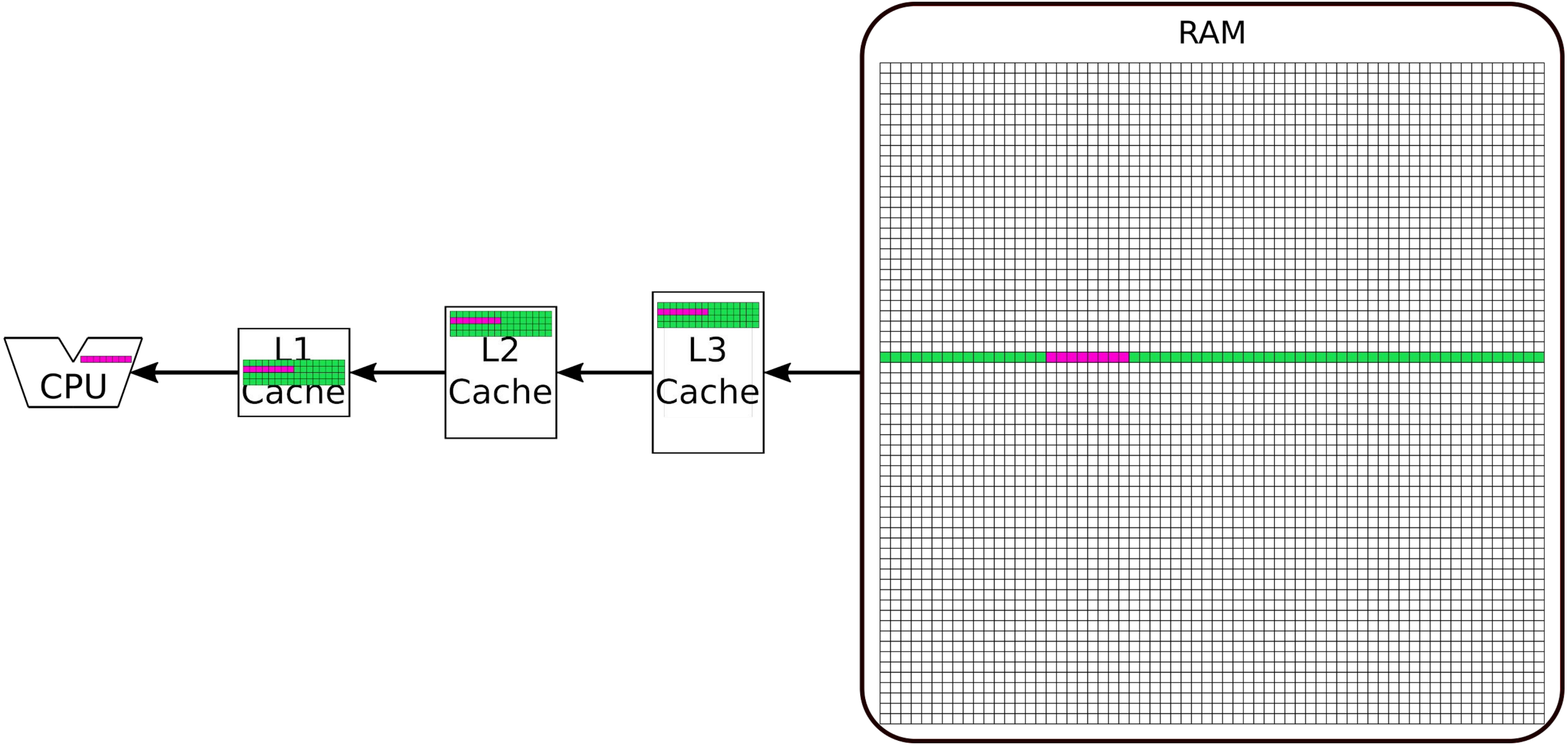
Cache line



Cache line



Cache line



Collisions and strategies of collision resolution

Definition: collision

- Two different elements are hashed to the same location
- $h(a) = h(b)$ for $a \neq b$

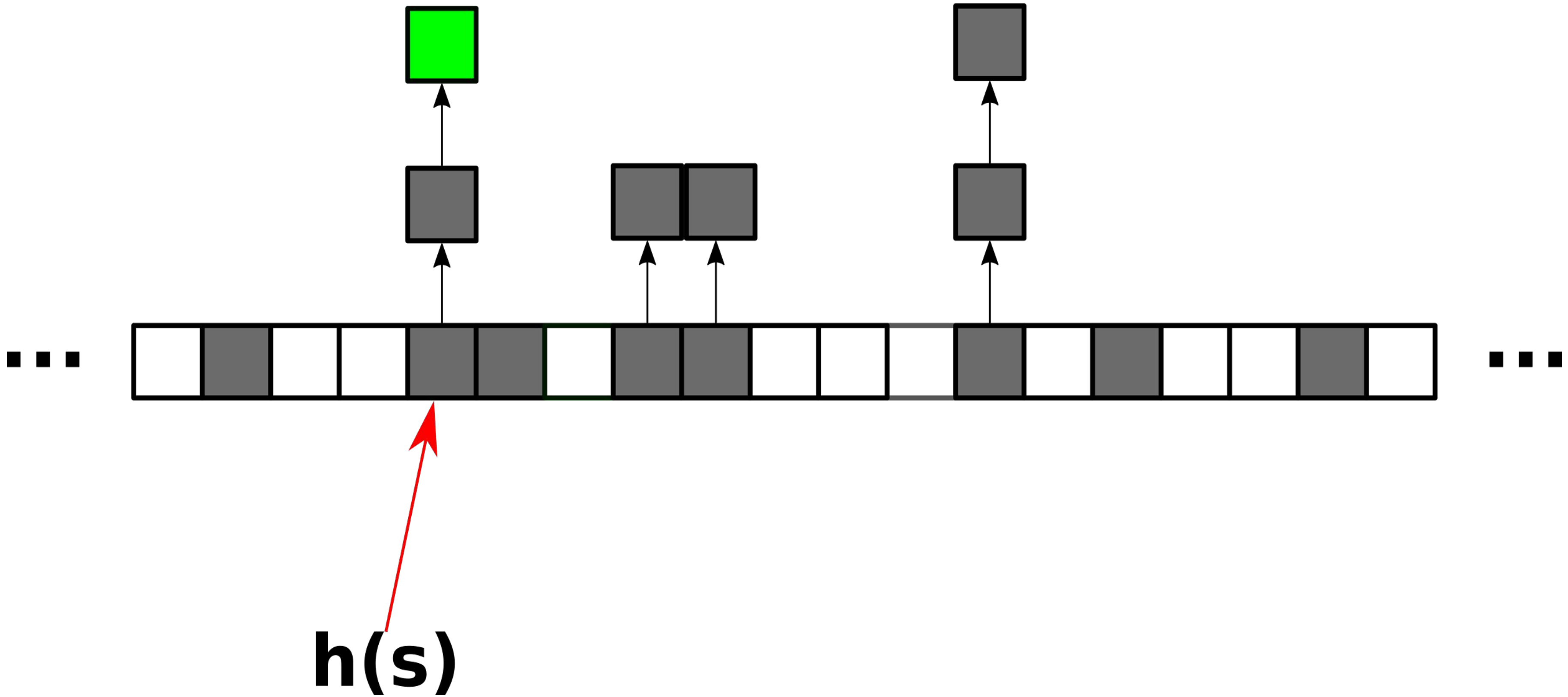
Strategies of collision resolution:

- Chaining (also: separate chaining)
- Open addressing (also: closed hashing)

Separate chaining

- Use one hash function
- Calculate hash position
- If a collision occurs:
    - Append new element to a (doubly) linked List

Separate chaining



Separate chaining

- Insert: O(1)
- Lookup: O(N)
  - Worst Case: Table degenerates to a single linked list

Open addressing / closed hashing

- Hash collision is resolved with probing
  - Linear probing
  - Quadratic probing
  - Double hashing

  - Cuckoo hashing
    - Standard
    - Multiple hash functions
    - Using buckets
    - (h,b) Cuckoo hashing

Linear probing

- One hash function
  h: $U \rightarrow \{0, \dots, P - 1\}$
- Distance c, often $c = 1$
- Hash function
  $h'_i(x) = (h(x) + c \cdot i) \mod P$

- Calculate hash position $h'_0(x)$
- Check if position is free
- If $h'_0(x)$ is occupied:
  - Calculate $h'_1(x)$
- Increase i until an empty slot is found

- Insert: O(N) worst case
- Lookup: O(N) worst case
- Expected case depends on table load
  (full slots / table size),
  fast if table is close to empty, $\ll 50\%$

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, 3rd Edition. MIT Press, 2009.

Linear probing

h'(s)

Linear probing

Linear probing

Linear probing

Linear probing

Linear probing

- Pros:
  - High performance for low to moderate loads (fill ratios), $\ll$ 50%

- Cons:
  - Worst case O(N) insertion and lookup time
  - In practice: slow if the table is loaded > 50%
  - Primary clustering (One collision causes more nearby collisions)

Quadratic probing

- One hash function
  $h: U \rightarrow \{0, \dots, P - 1\}$
- Distances c_1 and c_2

- Hash function
  $h'_i(x) = (h(x) + c_1 \cdot i + c_2 \cdot i^2) \mod P$

- Calculate hash position $h'_0(x)$
- Check if position is free
- If $h'_0(x)$ is occupied:
  - Calculate $h'_1(x)$
- Increase i until an empty slot is found

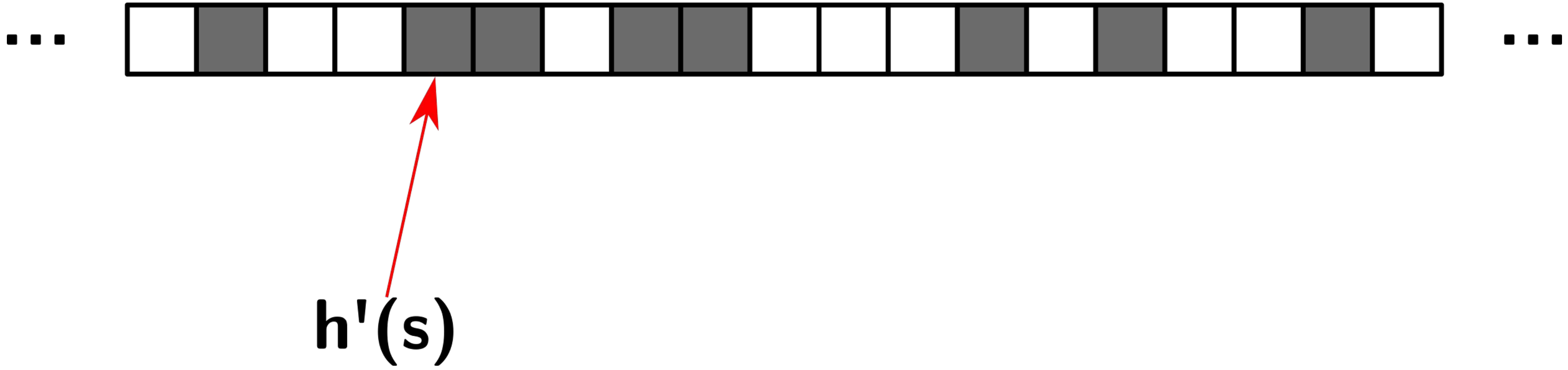
- Insert: O(N), where $N = |K|$
- Lookup: O(N)

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, 3rd Edition. MIT Press, 2009.

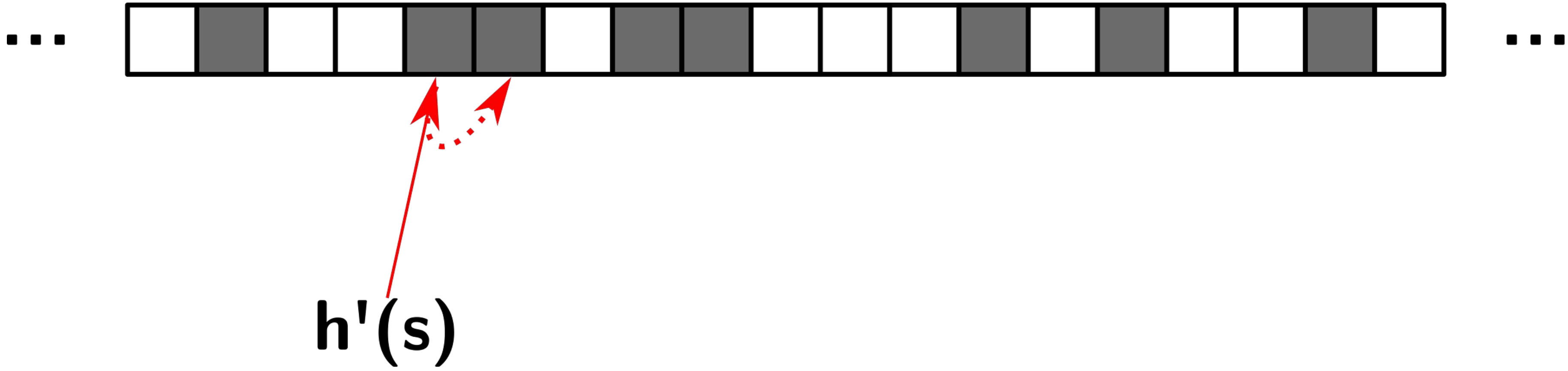
Quadratic probing

h'(s)

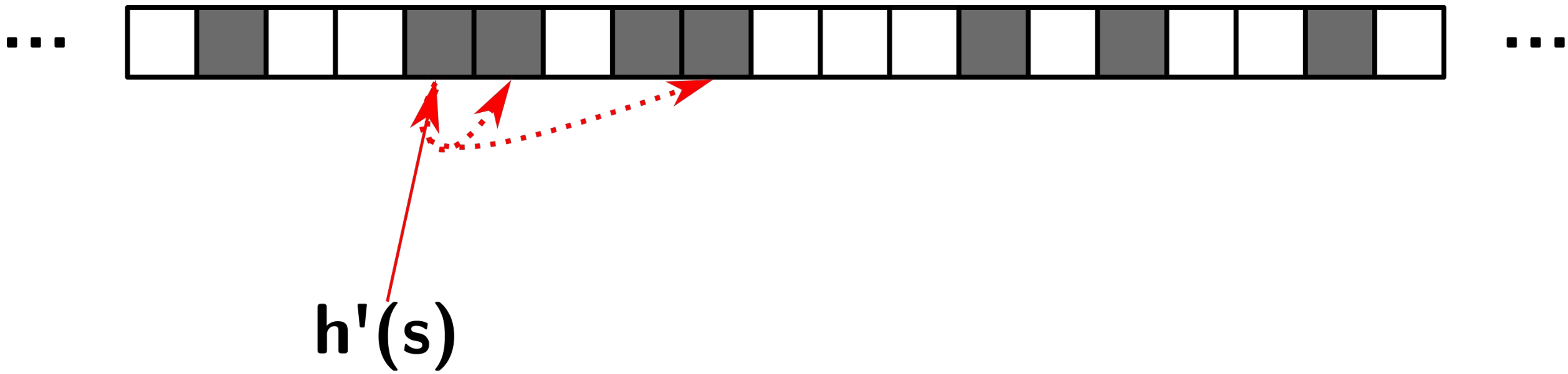
Quadratic probing



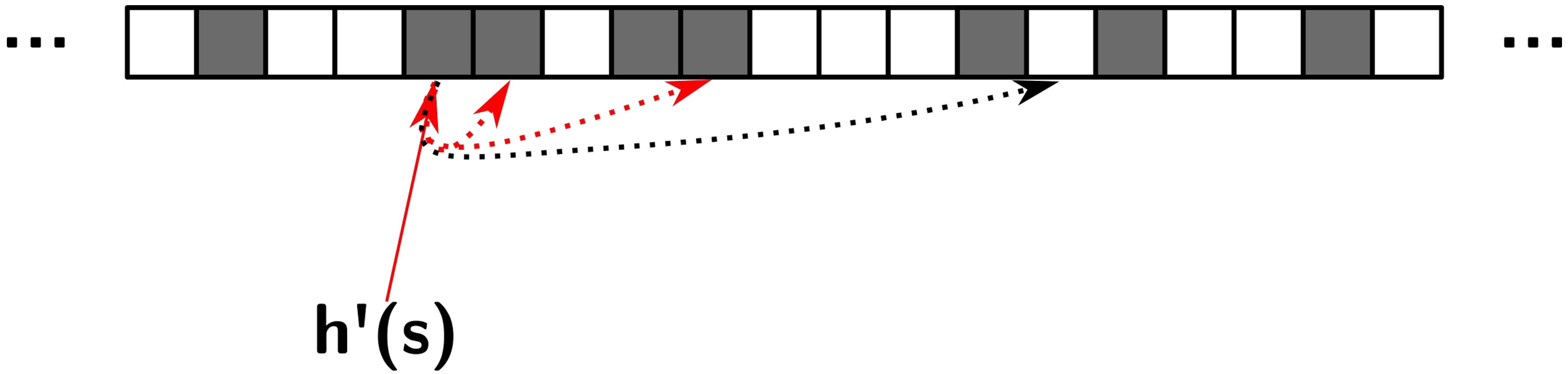
Quadratic probing



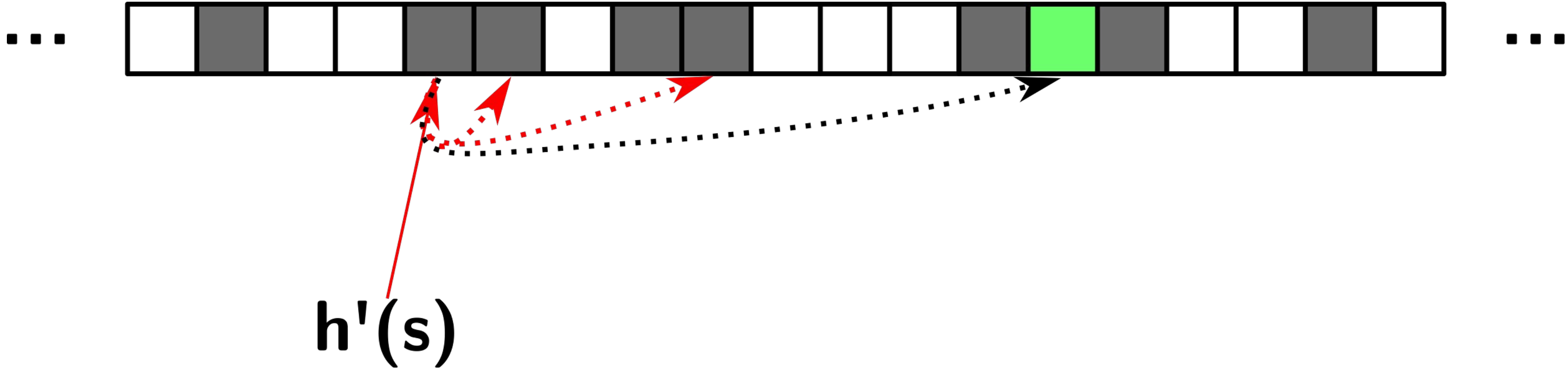
Quadratic probing



Quadratic probing



Quadratic probing



Double hashing

- Two hash functions
  $h_1(x)$ and $h_2(x)$
- Hash function
  $h'_i(x) = (h_1(x) + i \cdot h_2(x)) \mod P$

- Calculate hash position $h'_0(x)$
- Check if position is free
- If $h'_0(x)$ is occupied:
  - Calculate $h'_1(x)$
- Increase *i* until an empty slot is found

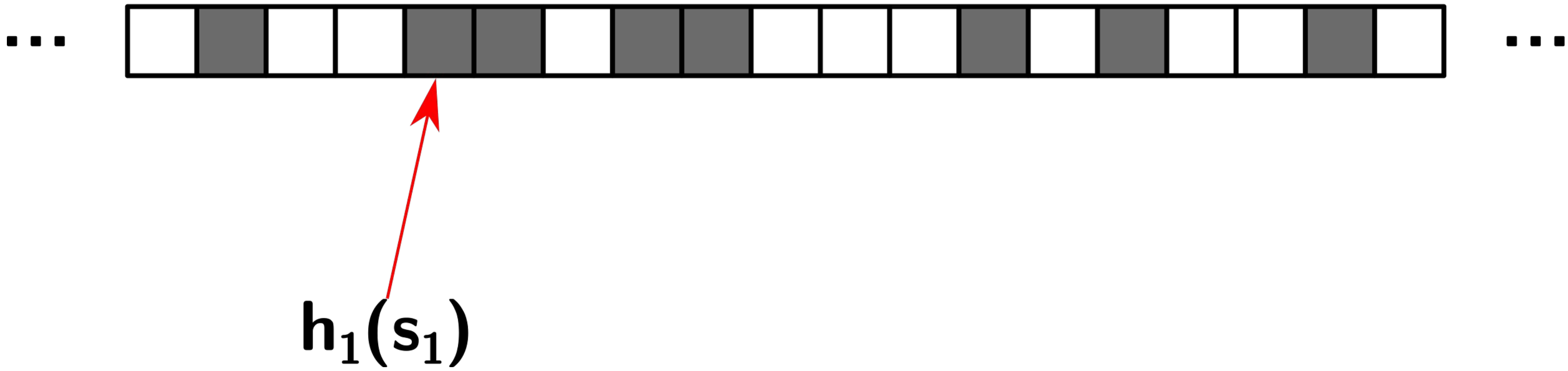
- Insert: O(N)
- Lookup: O(N)

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, 3rd Edition. MIT Press, 2009.

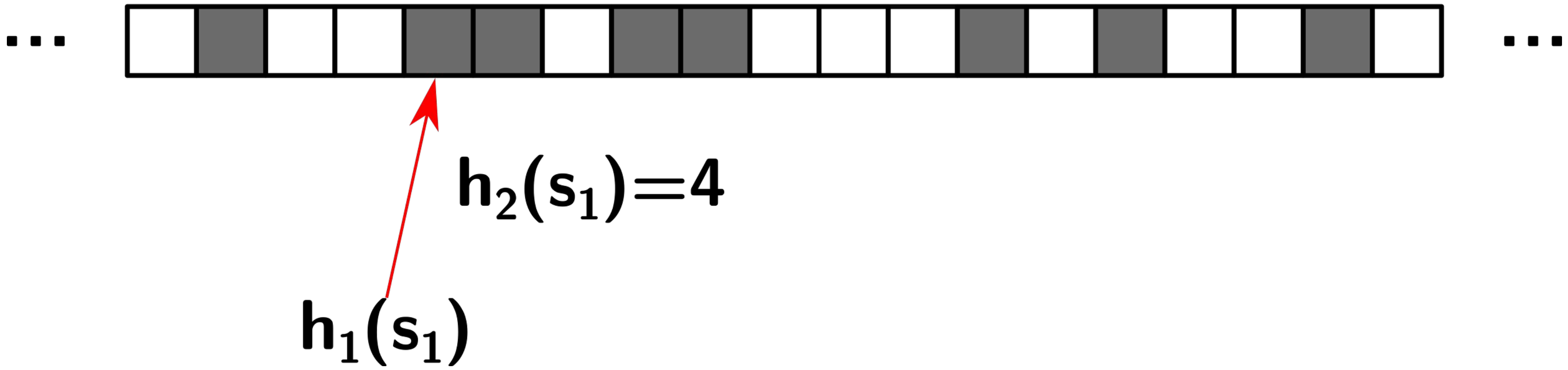
Double hashing

$$h_1(s_1)$$

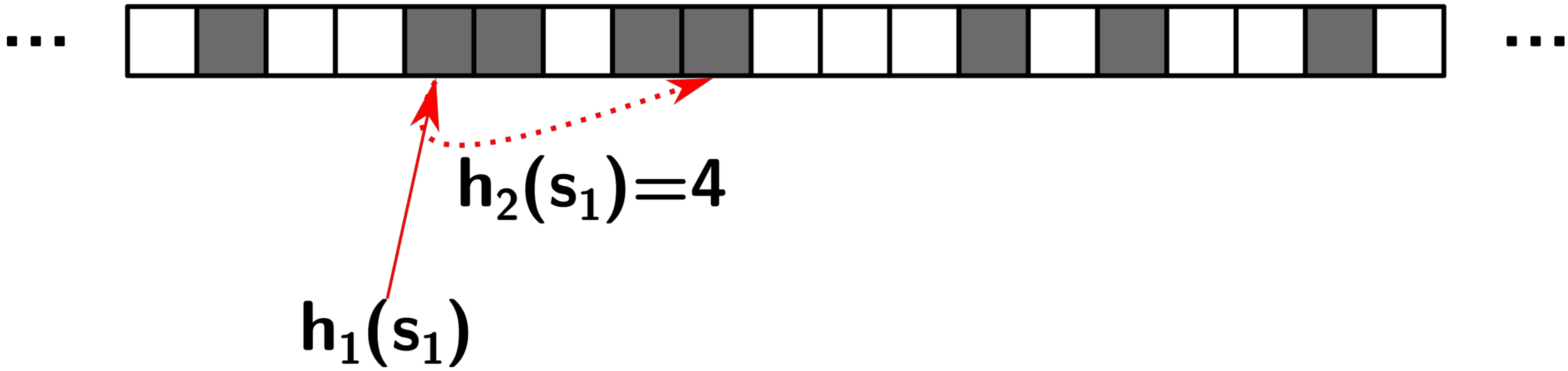
Double hashing



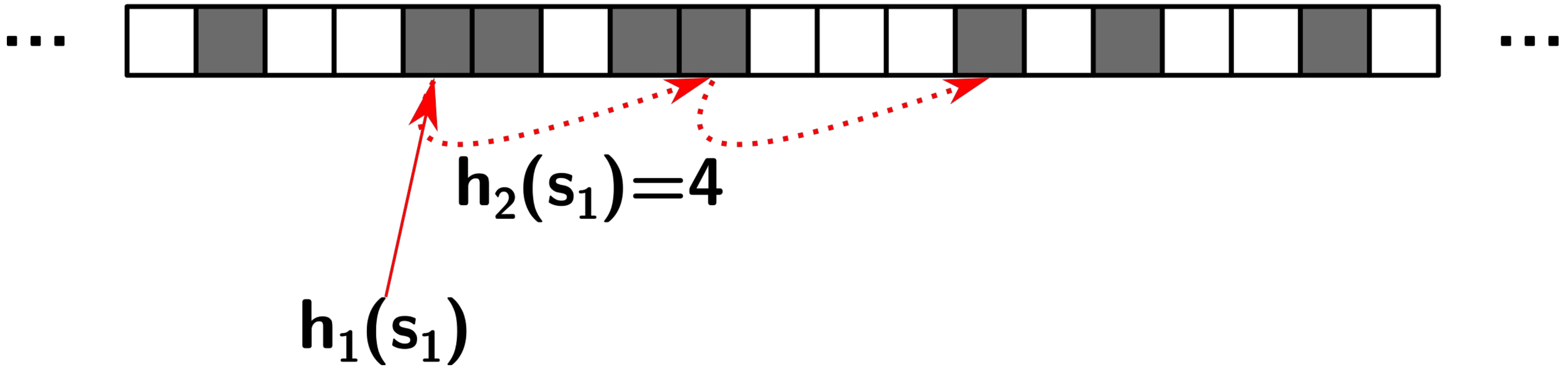
Double hashing



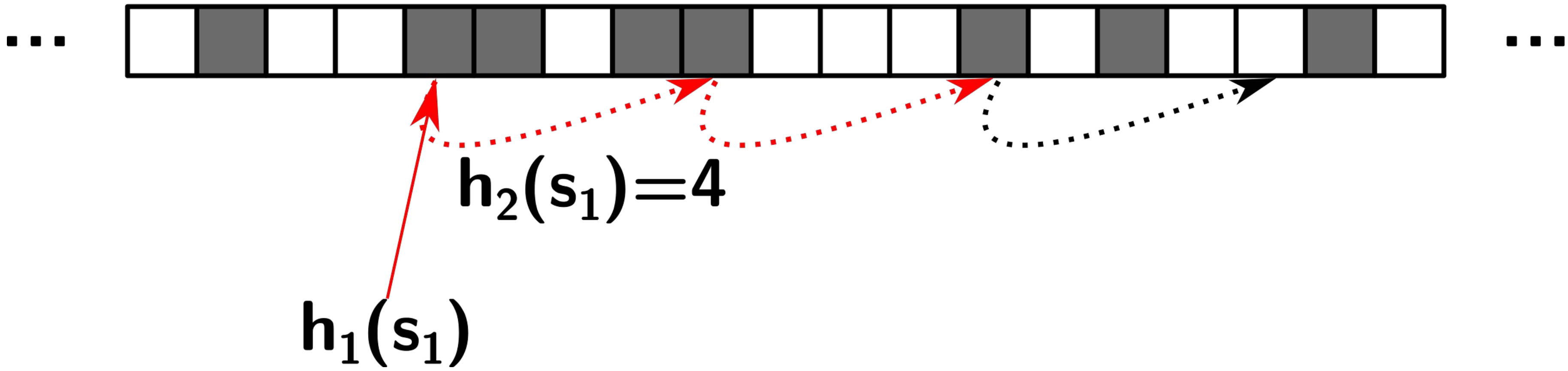
Double hashing



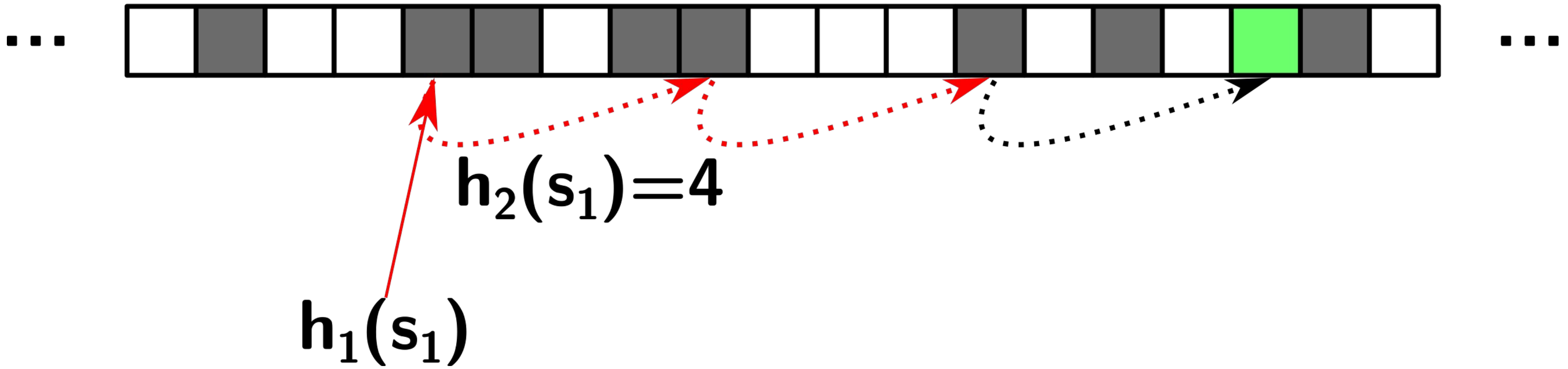
Double hashing



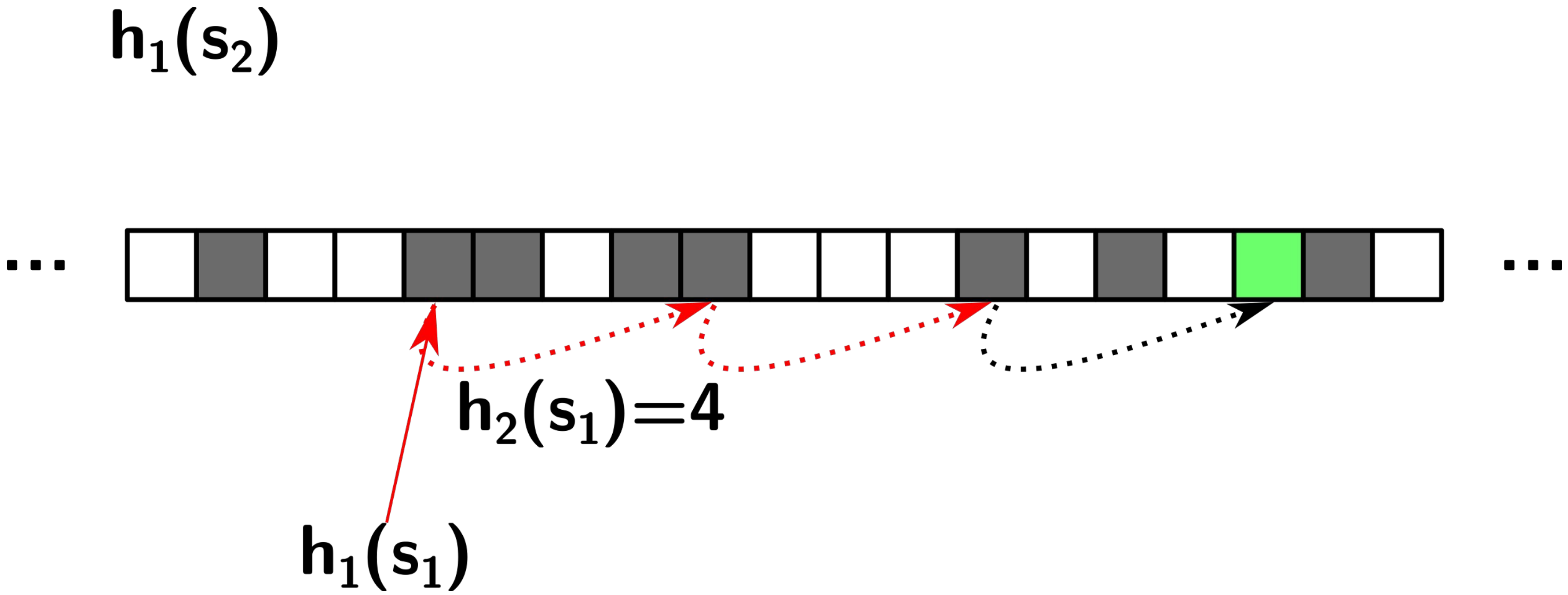
Double hashing



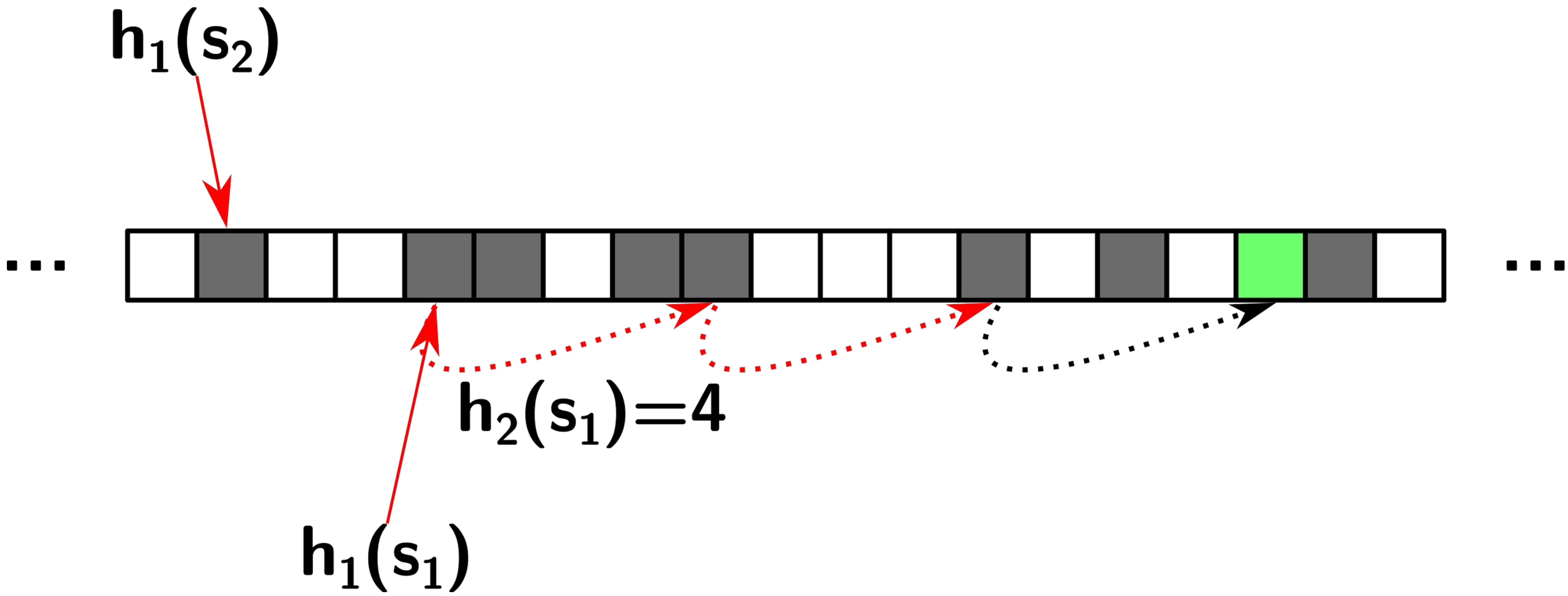
Double hashing



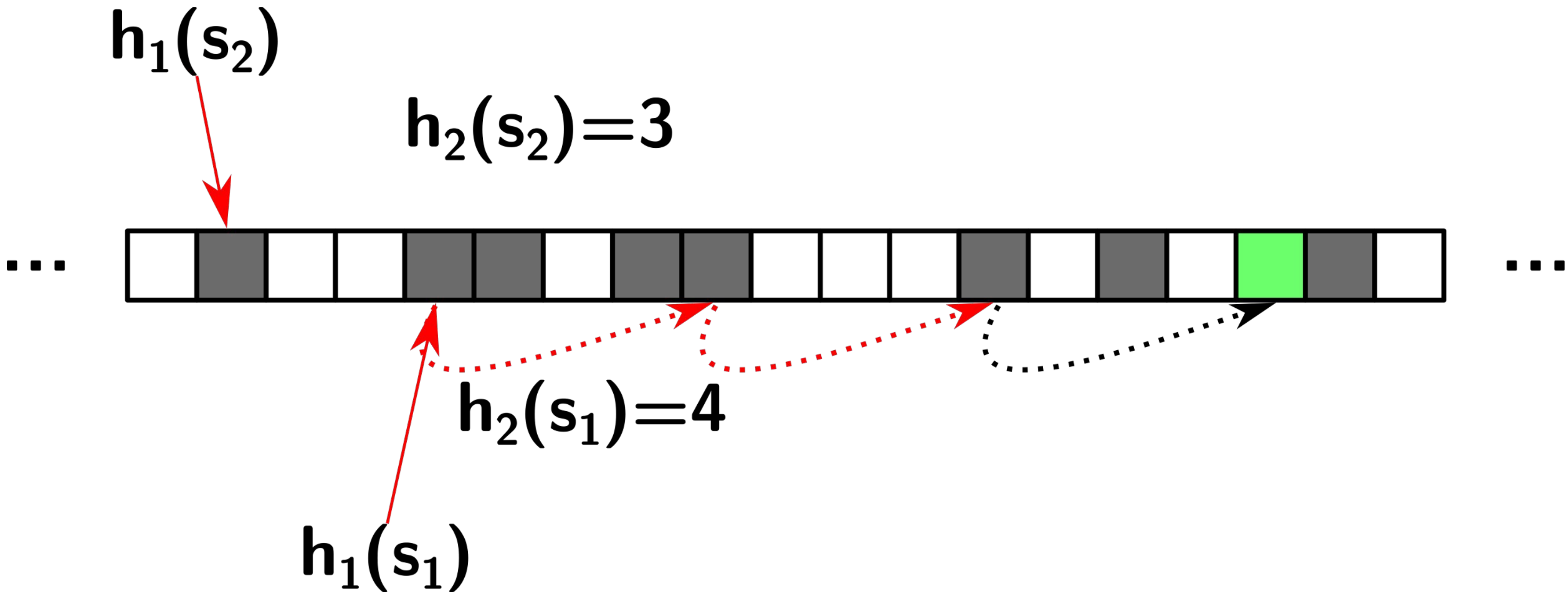
Double hashing



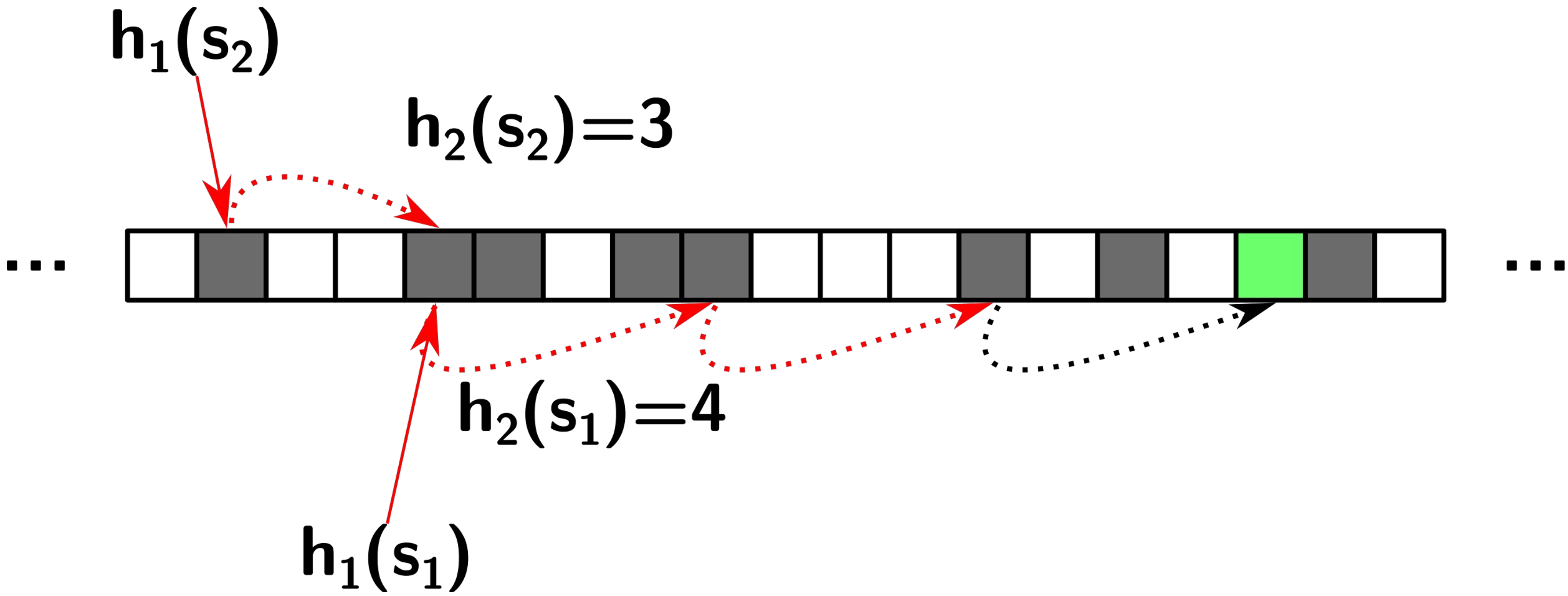
Double hashing



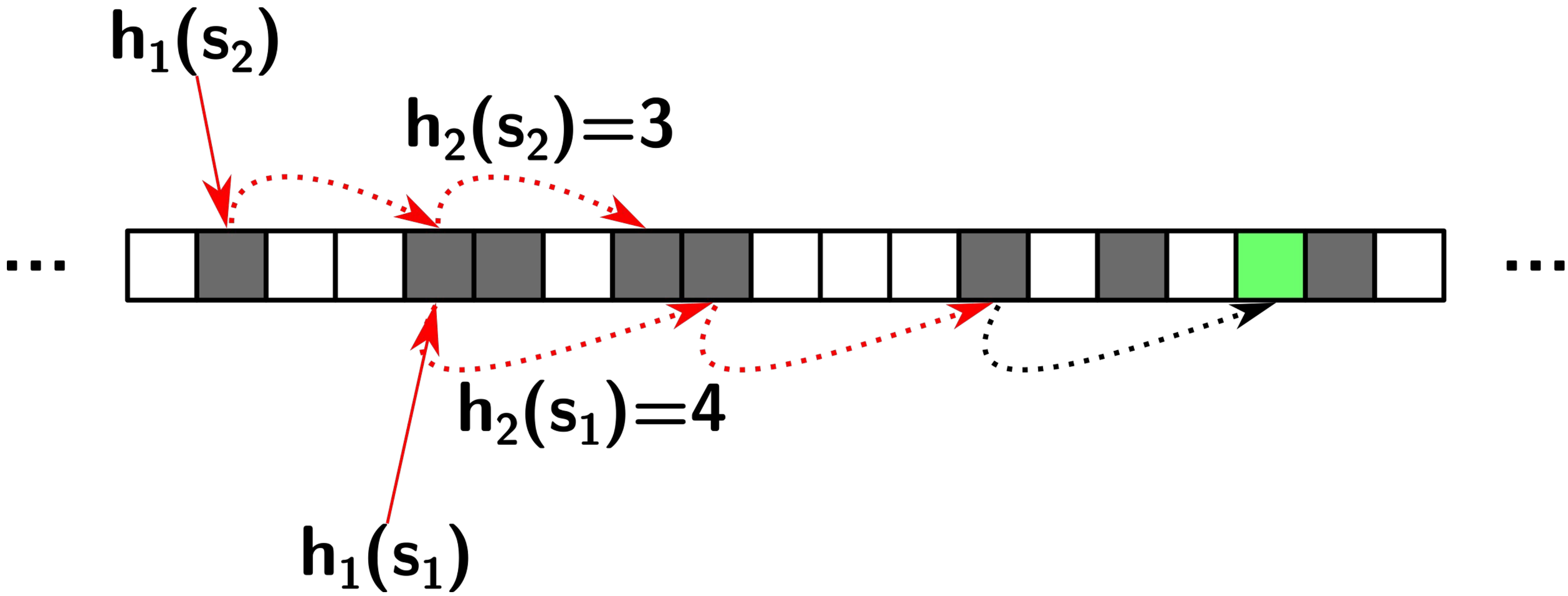
Double hashing



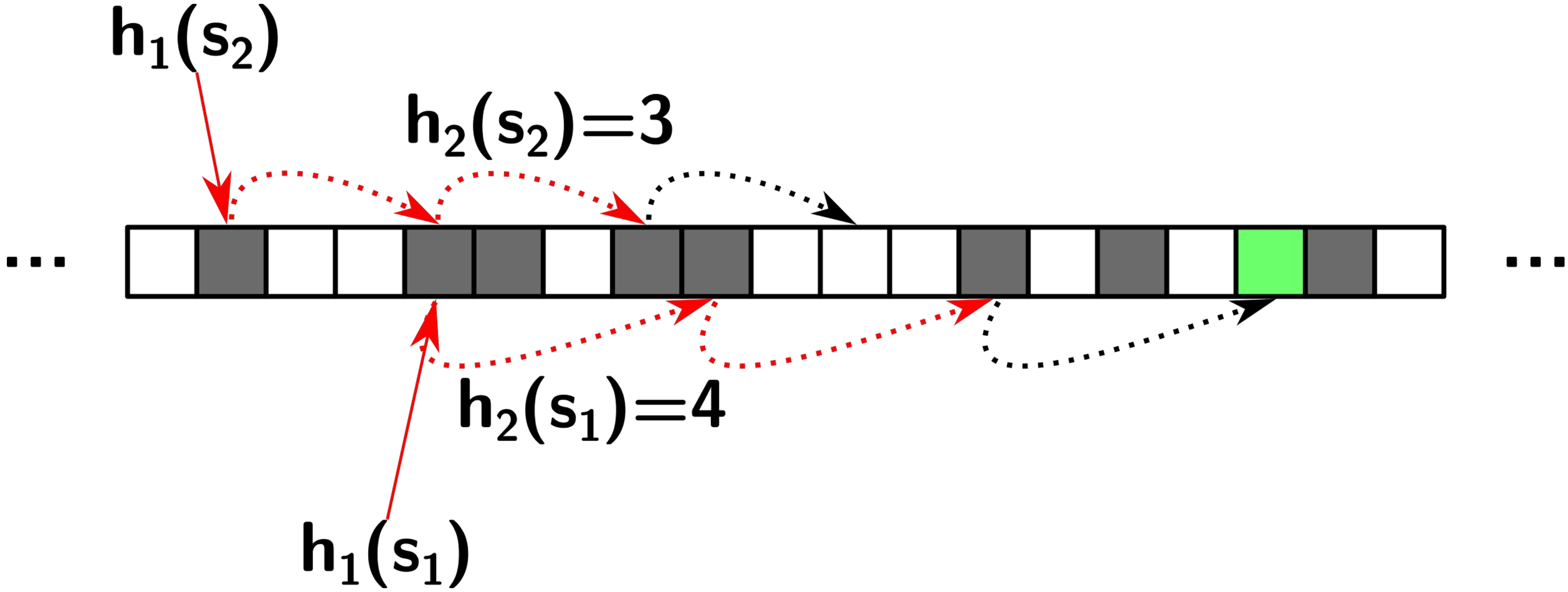
Double hashing



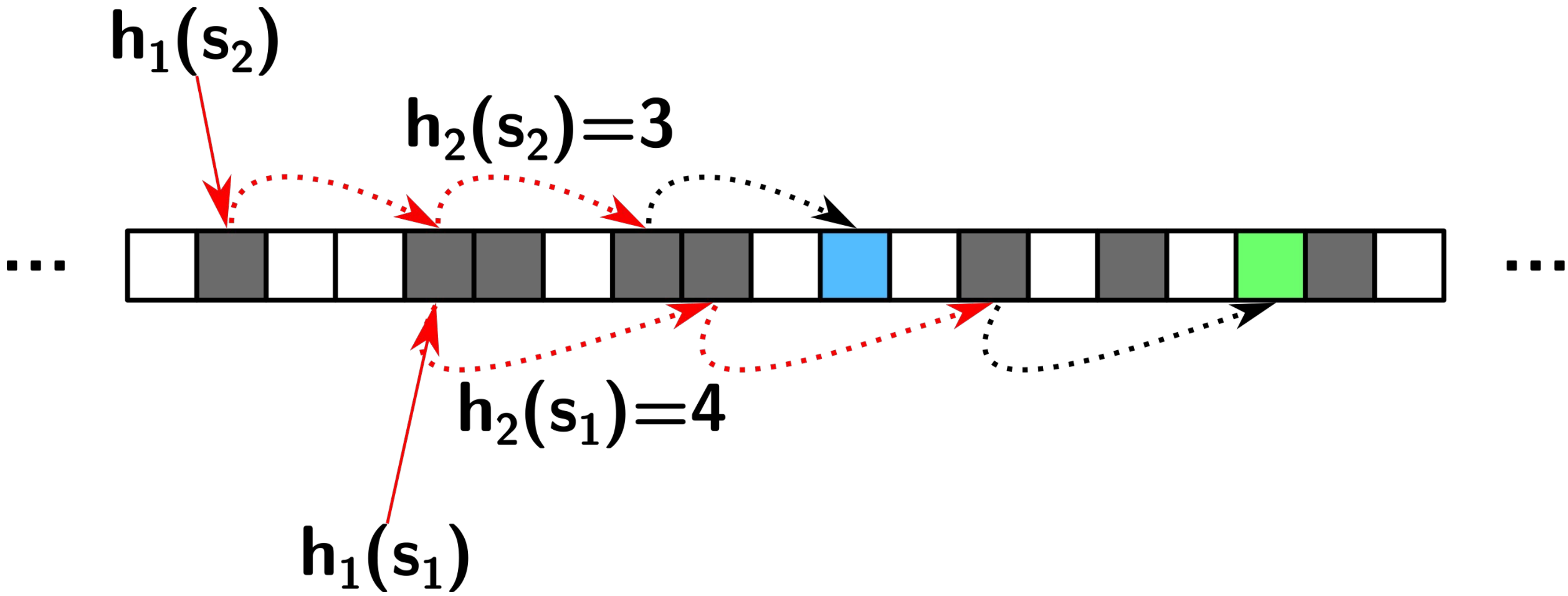
Double hashing



Double hashing



Double hashing



Quadratic probing and double hashing

- Pro:
  - No primary clustering
  - High performance for low to moderate loads (fill ratios), $\ll 50\%$
- Cons:
  - Worst case O(N) insertion and lookup time
  - In practice: slow if the table is loaded > 50%

Next part:
Multi-way bucketed cuckoo hashing for DNA k-mers