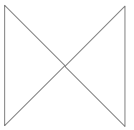
Hashing with SIMD: Linear Hashing and Double Hashing

Jakob Huber
05.11.18

Motivation

R

key	payload
1	199435
2	23645
3	576476
4	34534
5	786367
6	23974
7	89421
8	2842641
9	7883448
10	45483248
11	45678134
12	9435933



S

key	payload
7	7891523
9	2973758
12	1379312
12	753257
3	2793423
8	7855
1	14856486

Hashing with Linear Probing

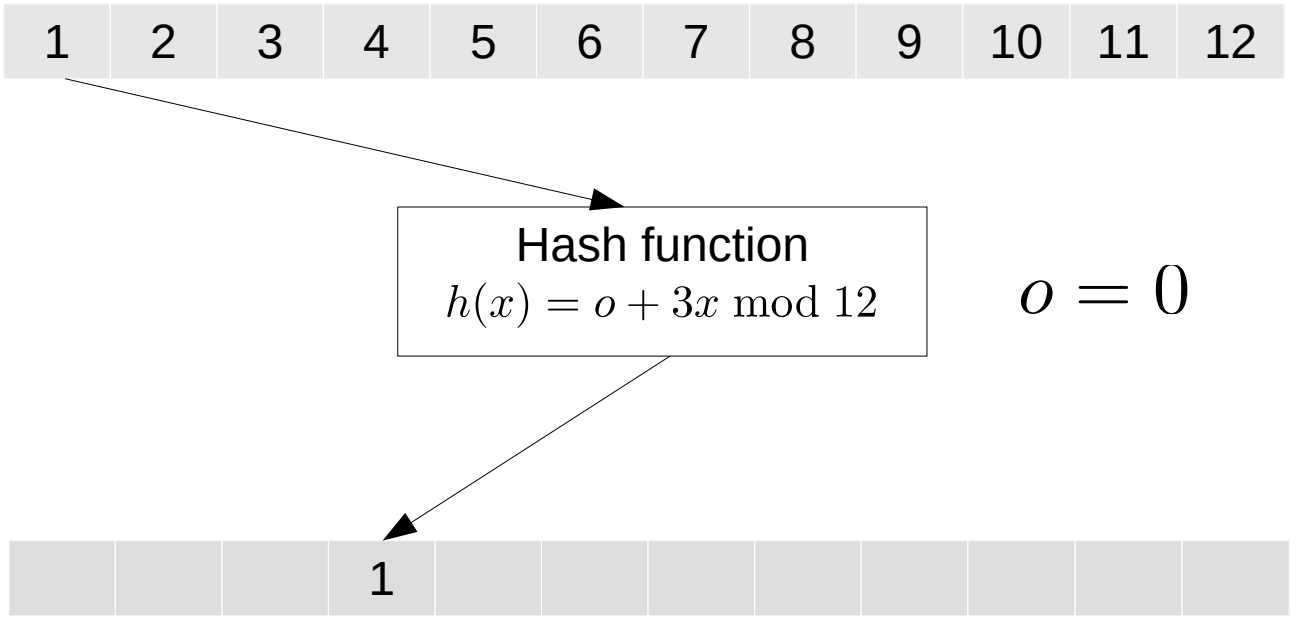
1	2	3	4	5	6	7	8	9	10	11	12

Hash function

$h(x) = o + 3x \bmod 12$

$o = 0$

Hashing with Linear Probing



Hashing with Linear Probing

				5	6	7	8	9	10	11	12

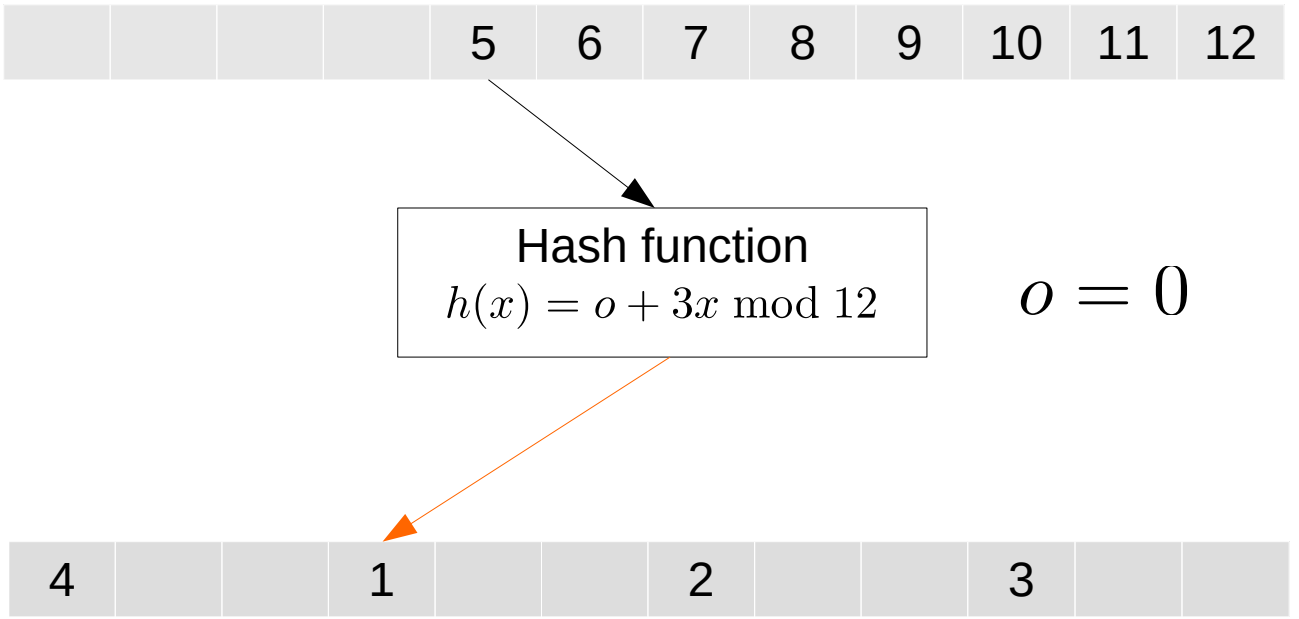
Hash function

$h(x) = o + 3x \bmod 12$

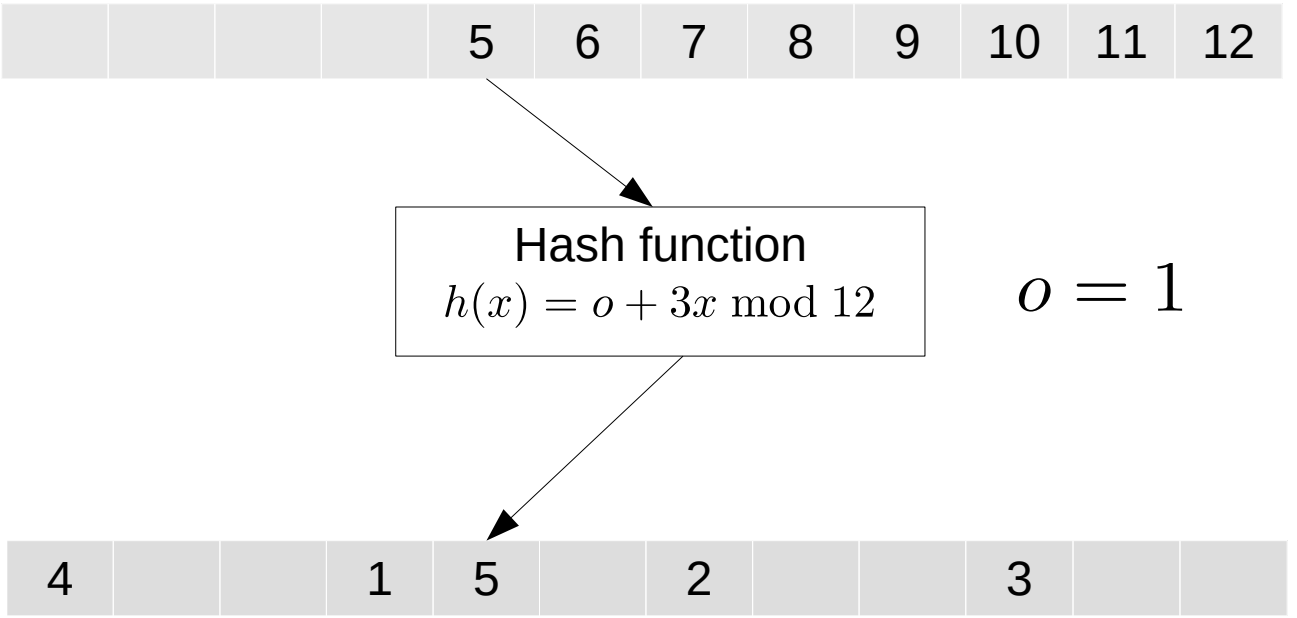
$o = 0$

4			1			2			3		

Hashing with Linear Probing



Hashing with Linear Probing



Double Hashing

1	2	3	4	5	6	7	8	9	10	11	12

Hash functions

$$h_1(x) = 3x \bmod 12$$

$$h_2(x) = 3x \bmod 11 + 1$$

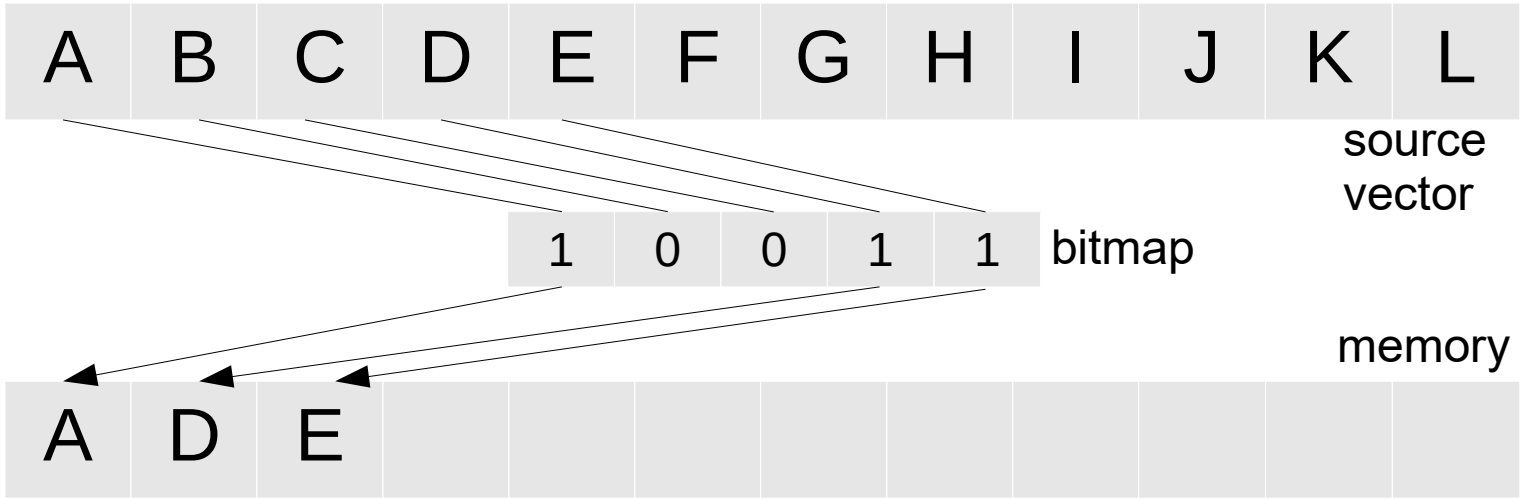
$$h(x) = (h_1(x) + n \cdot h_2(x)) \bmod 12$$

4			1	5		2			3		

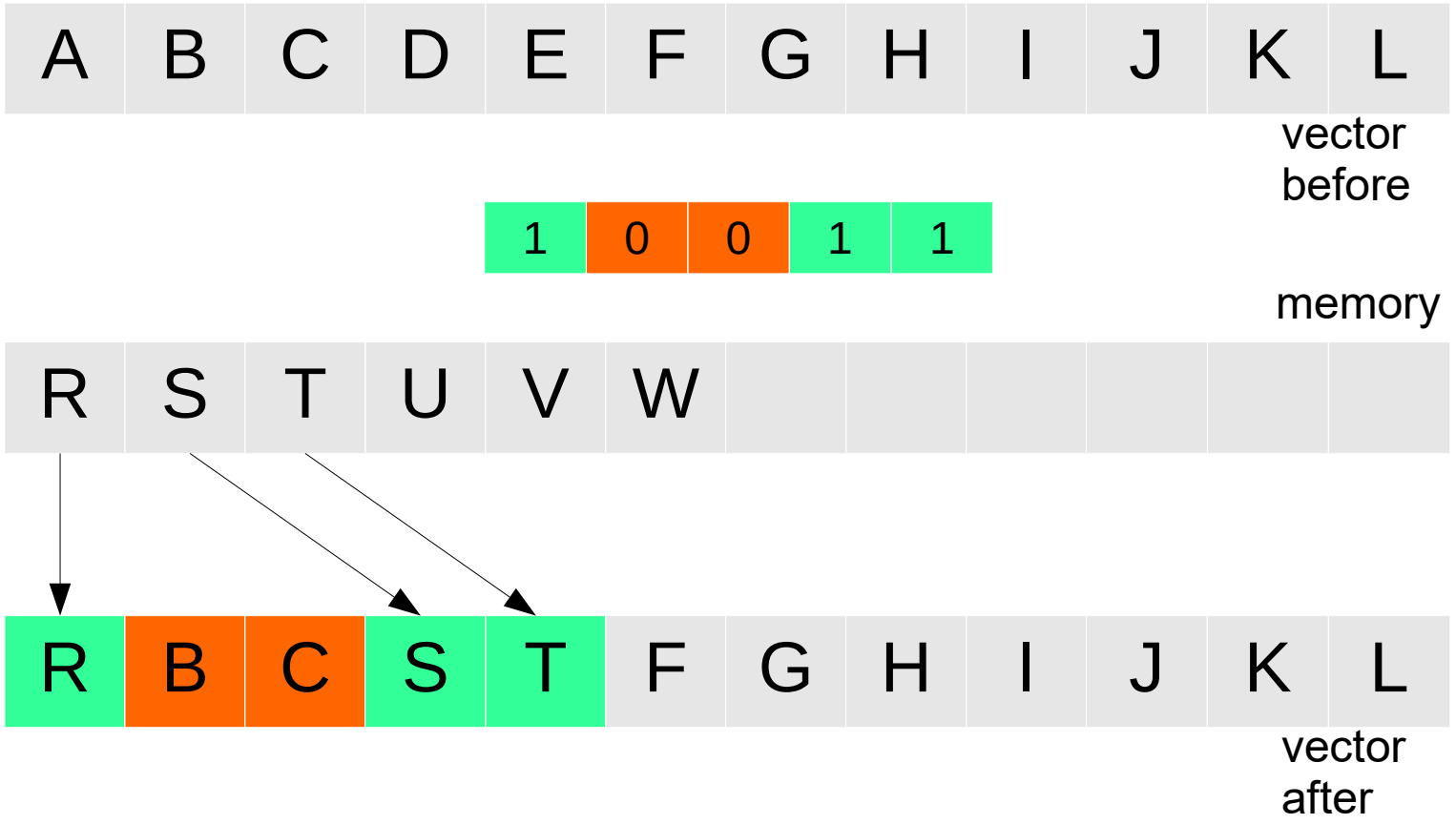
SIMD

- Single Instruction, Multiple Data
- Allows to perform arithmetic/logical operations on multiple values in one instruction
- Data can be stored in vectors
- Supported from Haswell Generation (mainstream CPUs)
- More advanced instructions available on MIC CPUs

SIMD: Selective Store

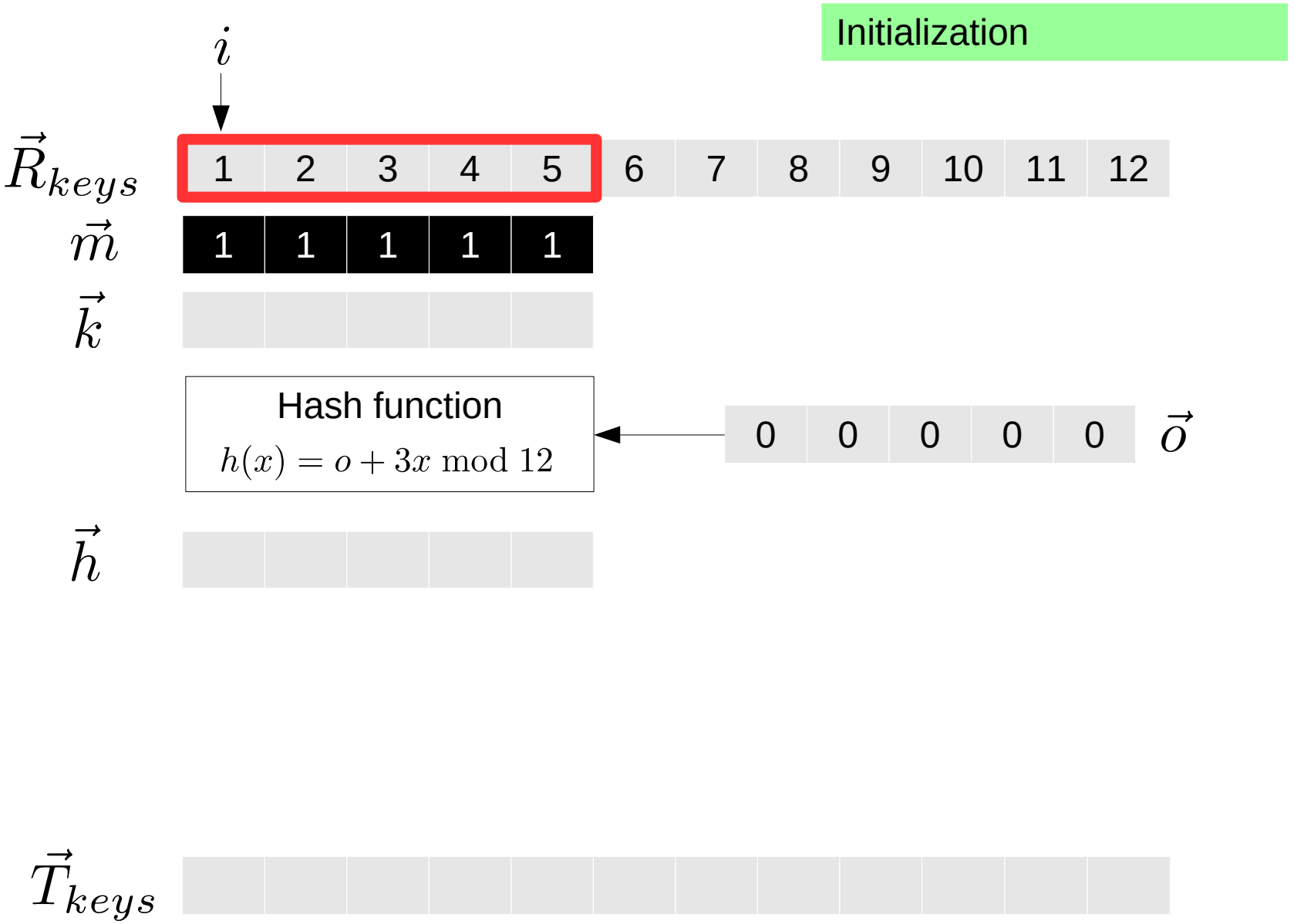


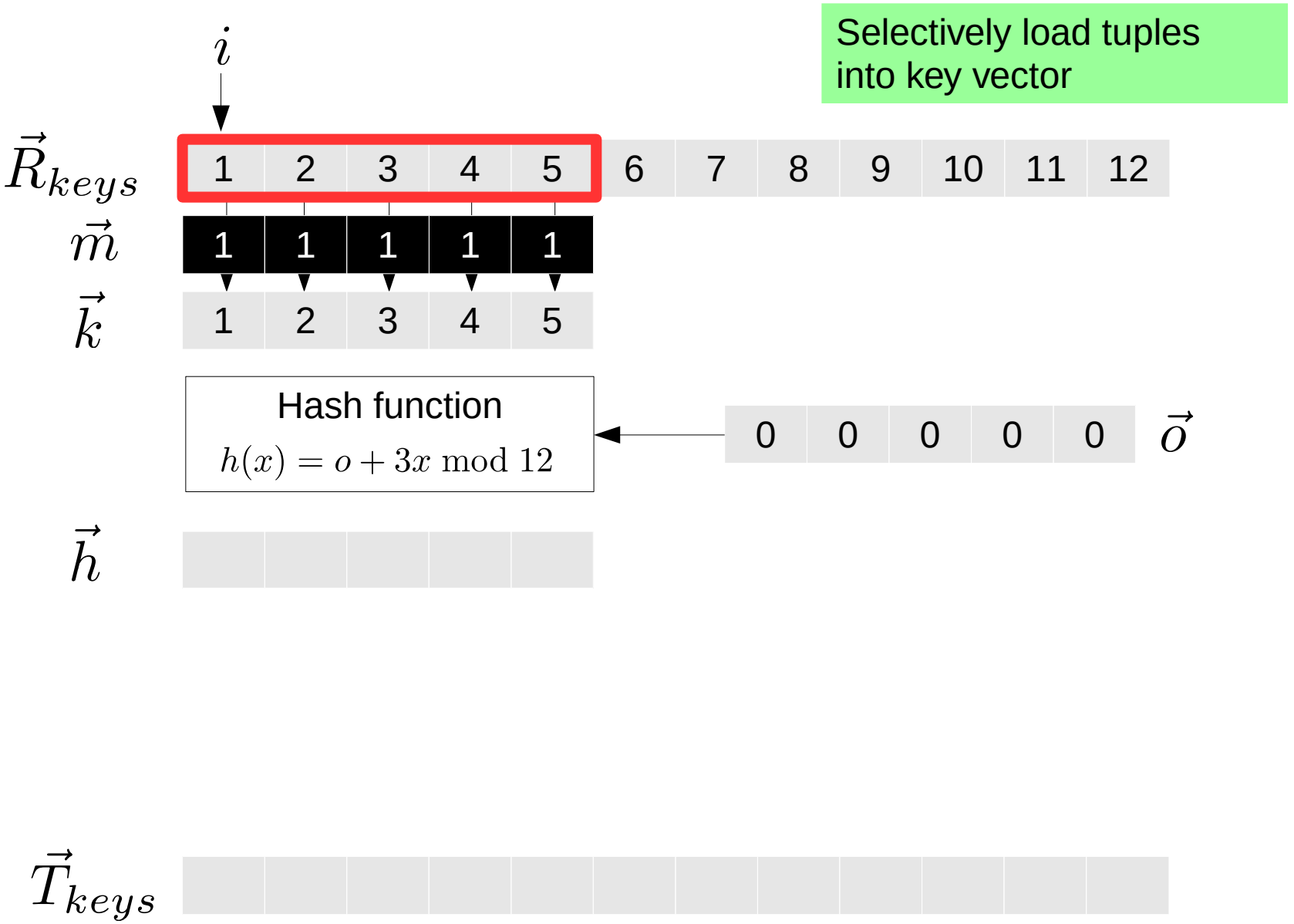
SIMD: Selective Load



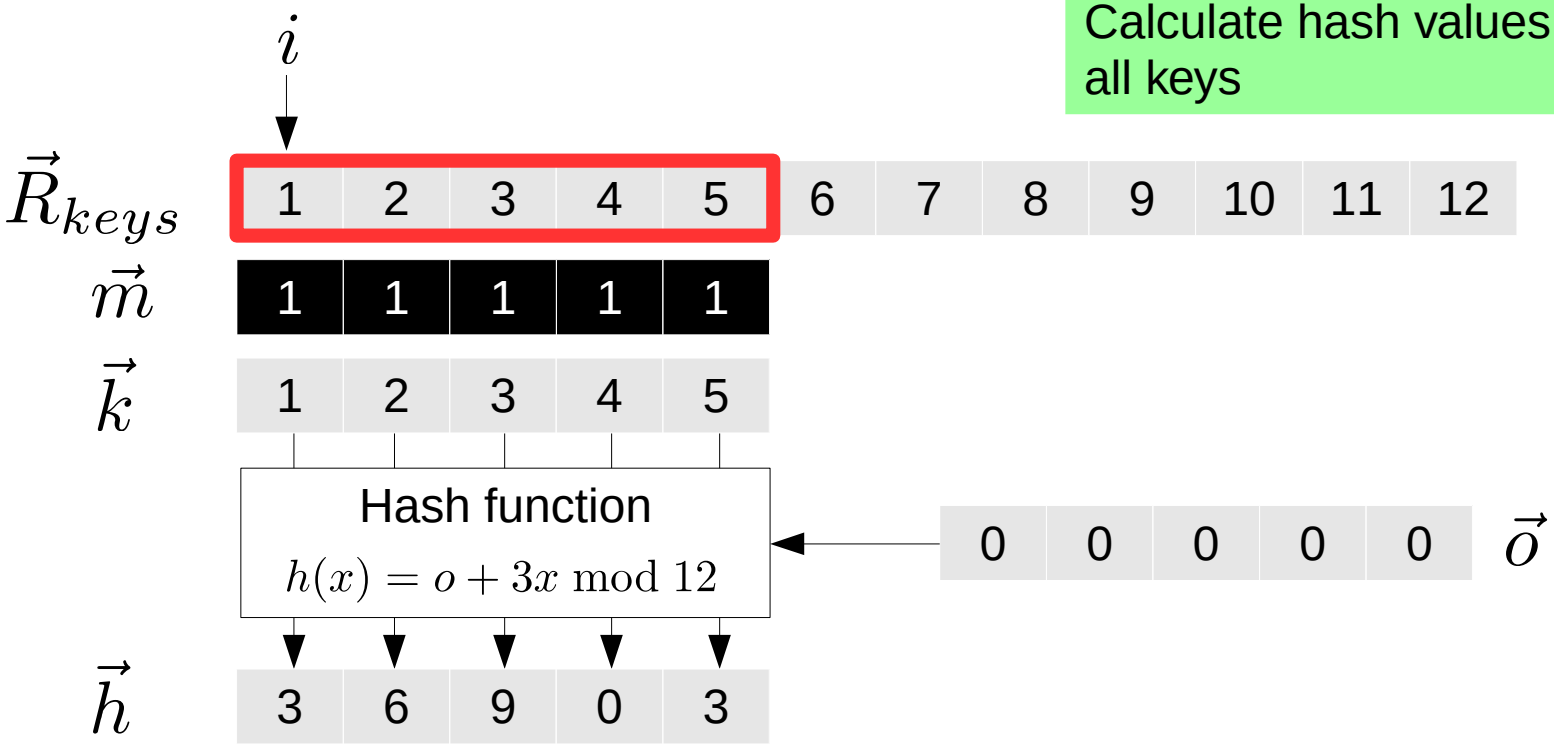
SIMD: Gather

SIMD: Scatter

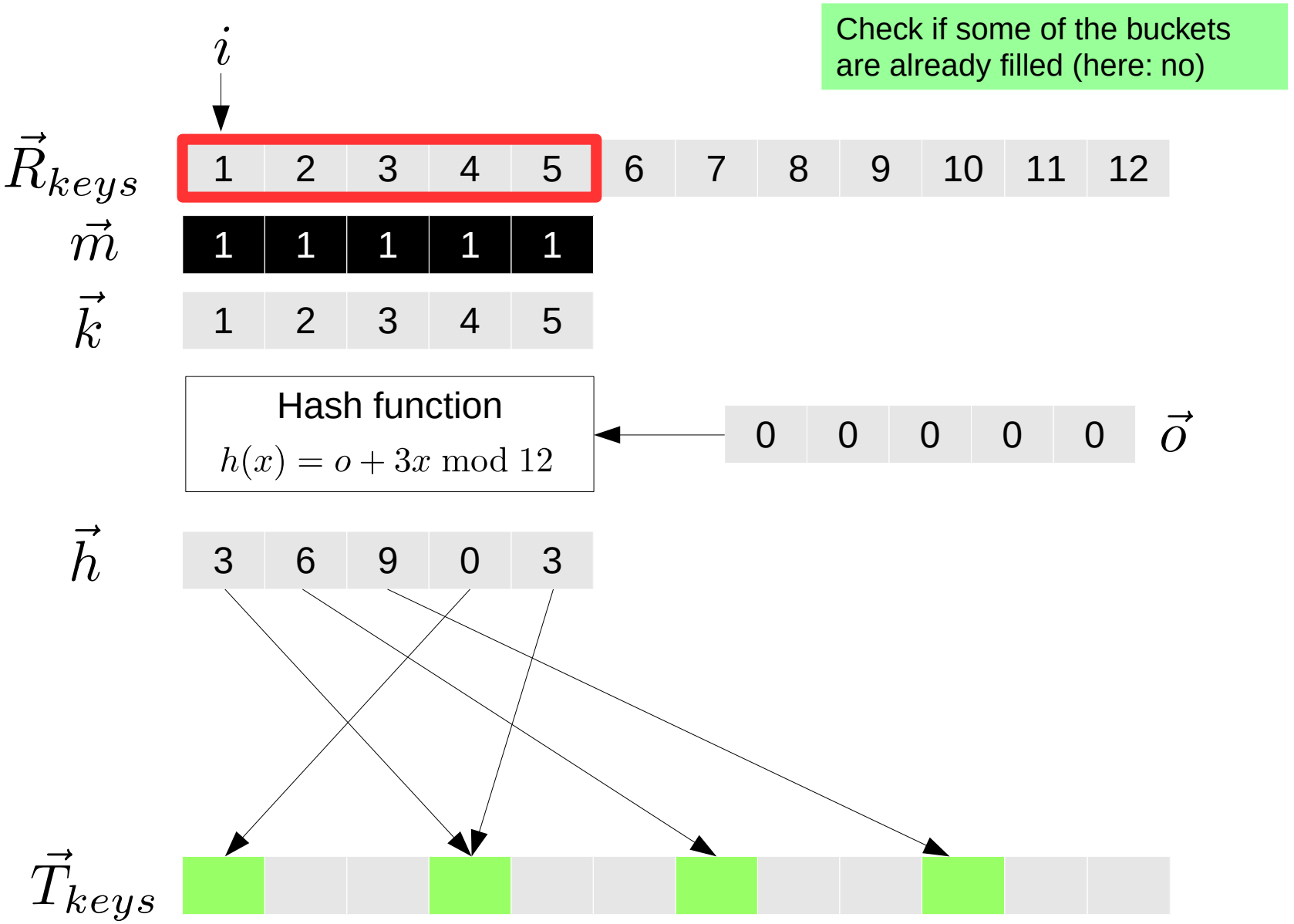

Initialization
i
$\vec{R}_{keys}$
1 2 3 4 5 6 7 8 9 10 11 12
$\vec{m}$
1 1 1 1 1
$\vec{k}$
Hash function
$h(x) = o + 3x \bmod 12$
0 0 0 0 0
$\vec{o}$
$\vec{h}$
$\vec{T}_{keys}$

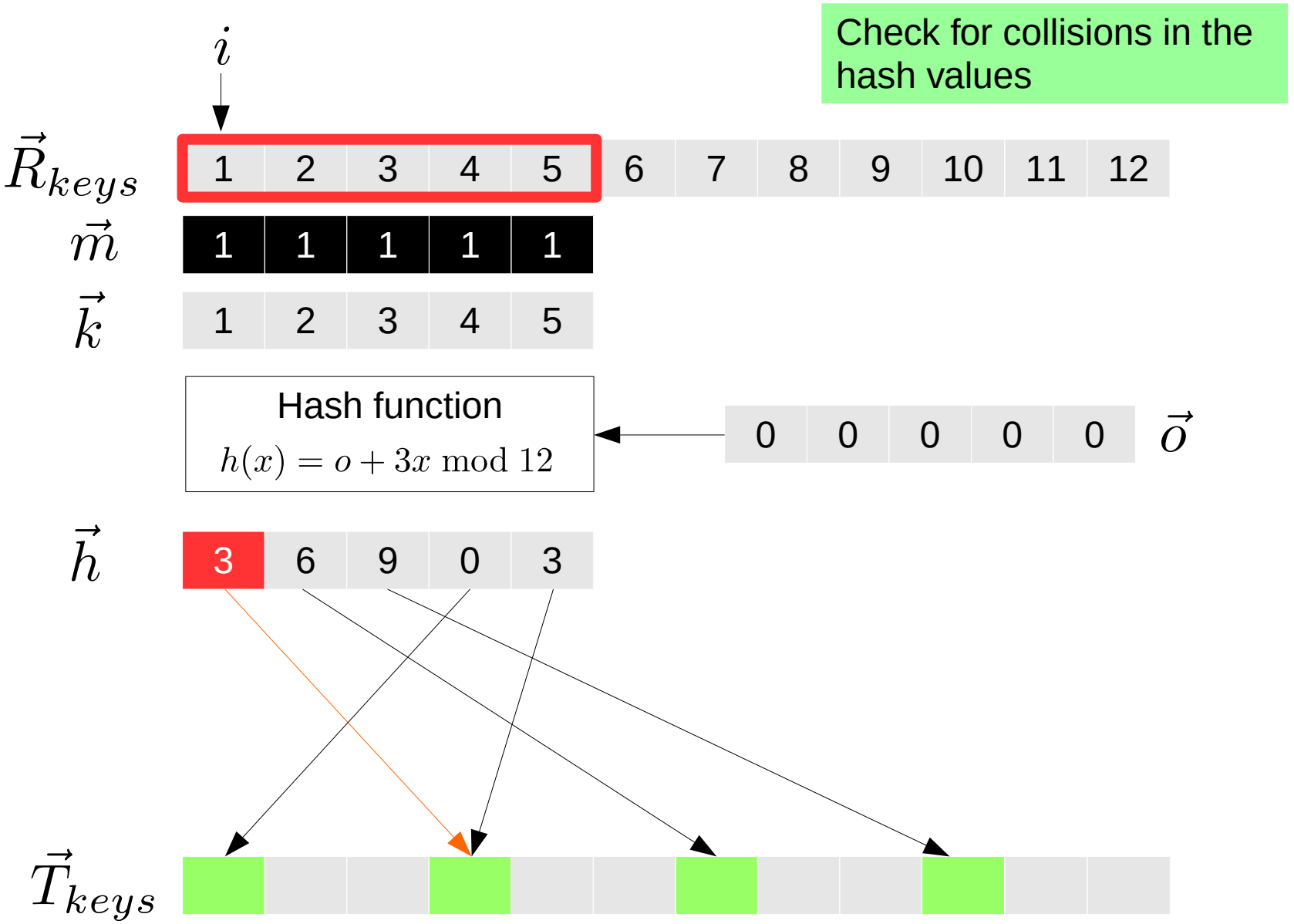

Selectively load tuples into key vector
i
$\vec{R}_{keys}$
1 2 3 4 5 6 7 8 9 10 11 12
$\vec{m}$
1 1 1 1 1
$\vec{k}$
1 2 3 4 5
Hash function
$h(x) = o + 3x \bmod 12$
0 0 0 0 0
$\vec{o}$
$\vec{h}$
$\vec{T}_{keys}$

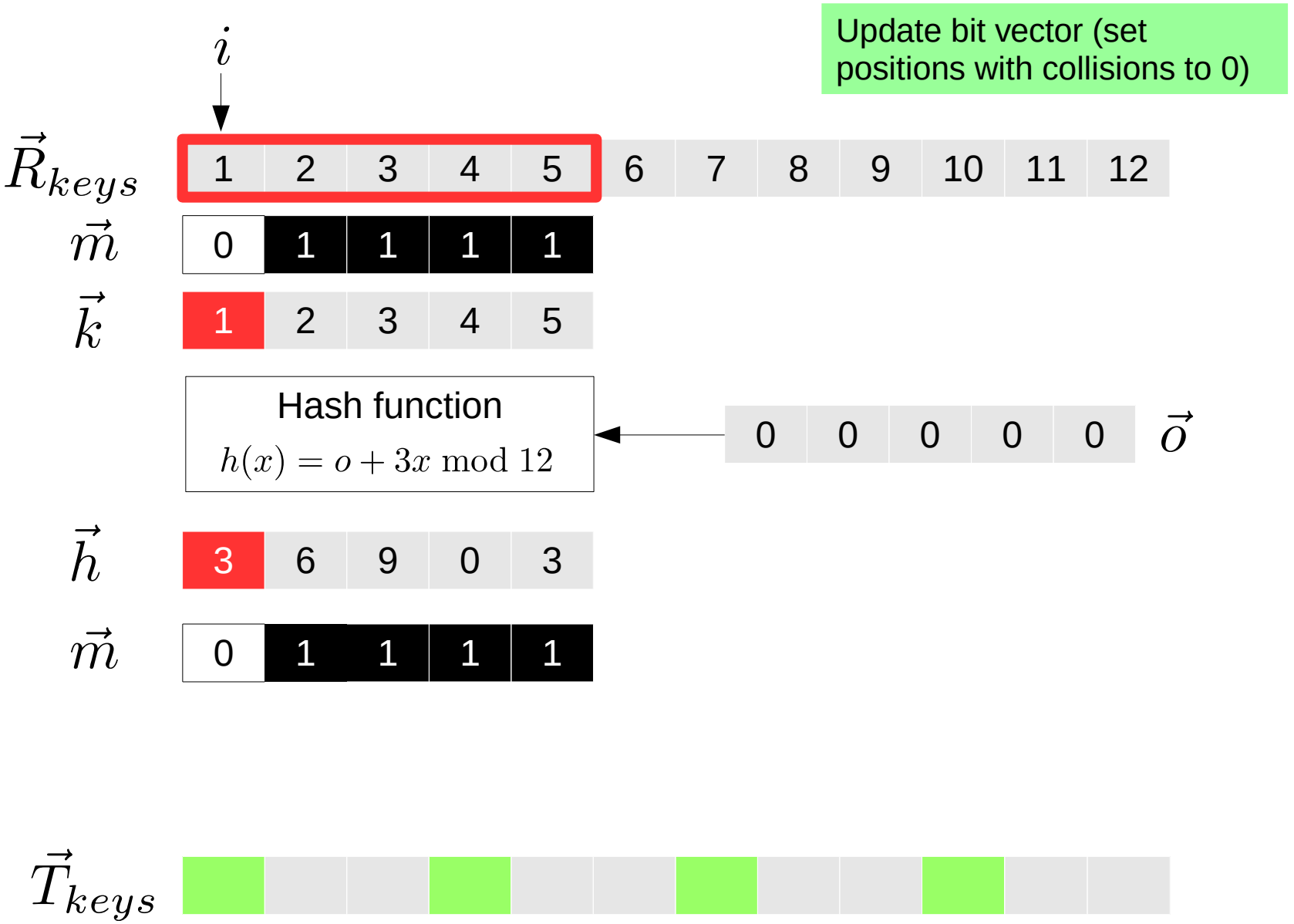
Calculate hash values for all keys

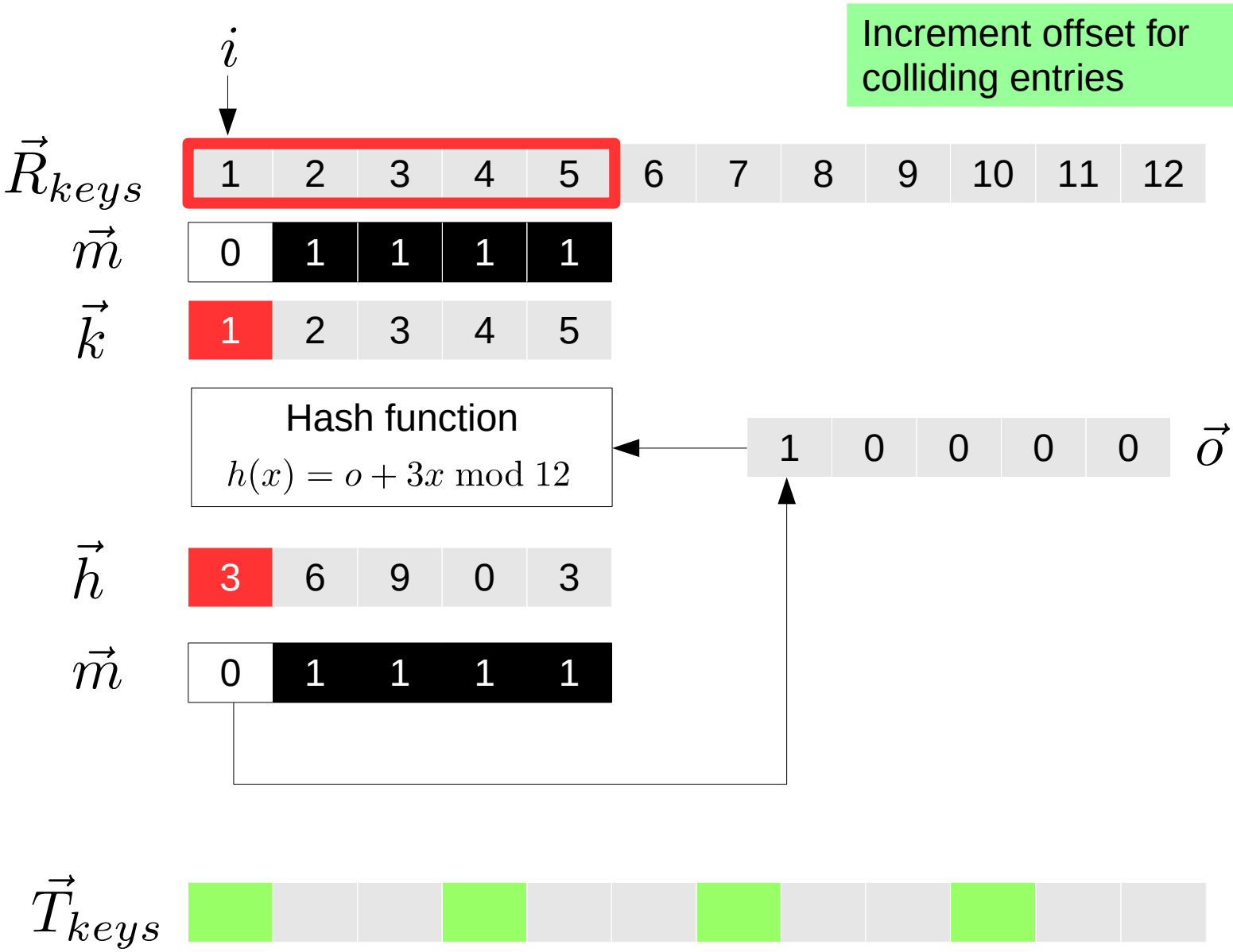


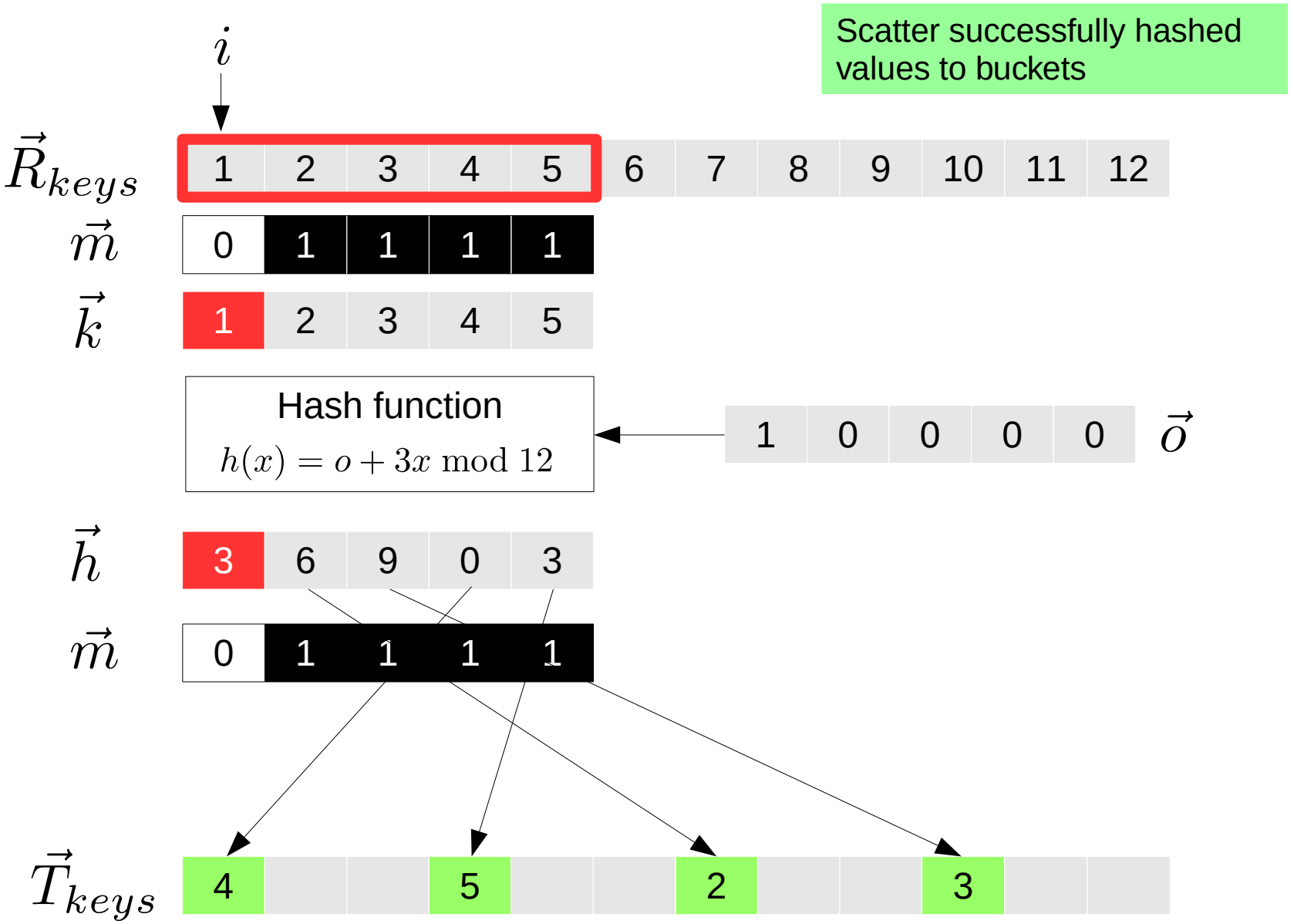
$\vec{T}_{keys}$

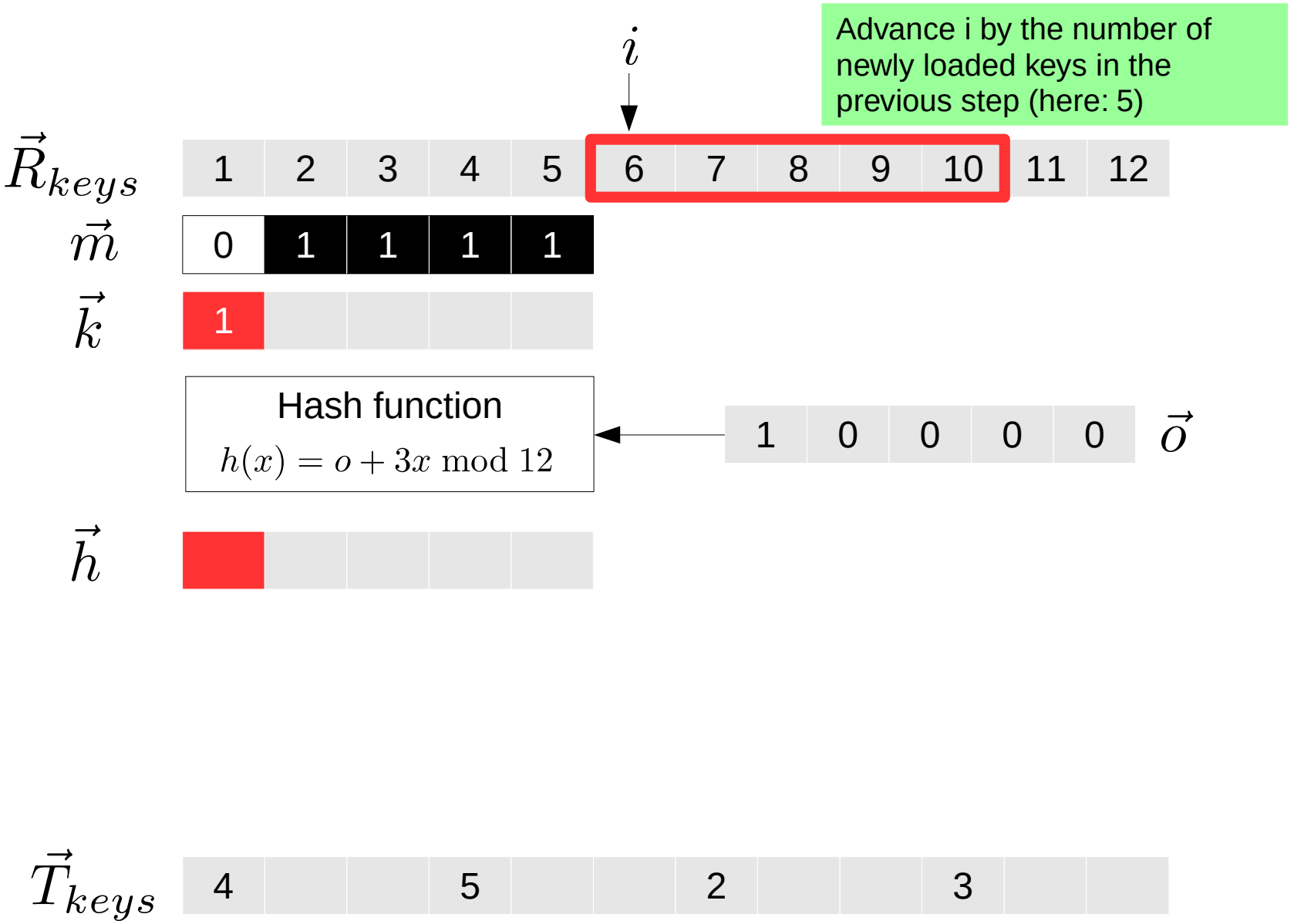

Check if some of the buckets are already filled (here: no)
i
$\vec{R}_{keys}$
1 2 3 4 5 6 7 8 9 10 11 12
$\vec{m}$
1 1 1 1 1
$\vec{k}$
1 2 3 4 5
Hash function
$h(x) = o + 3x \bmod 12$
0 0 0 0 0
$\vec{o}$
$\vec{h}$
3 6 9 0 3
$\vec{T}_{keys}$

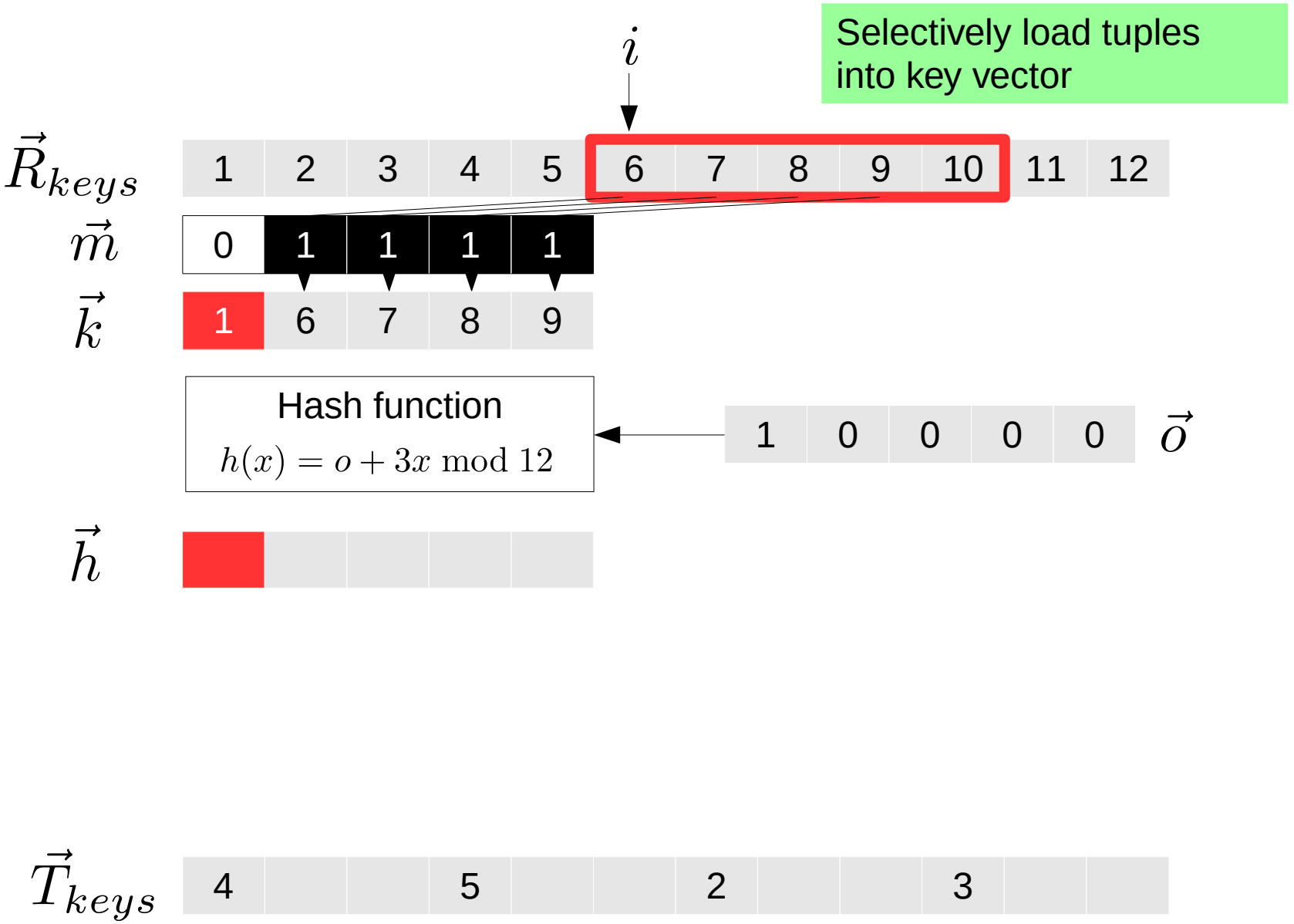

Check for collisions in the hash values
i
$\vec{R}_{keys}$
1 2 3 4 5 6 7 8 9 10 11 12
$\vec{m}$
1 1 1 1 1
$\vec{k}$
1 2 3 4 5
Hash function
$h(x) = o + 3x \bmod 12$
0 0 0 0 0 $\vec{o}$
$\vec{h}$
3 6 9 0 3
$\vec{T}_{keys}$


Update bit vector (set positions with collisions to 0)
i
$\vec{R}_{keys}$
1 2 3 4 5 6 7 8 9 10 11 12
$\vec{m}$
0 1 1 1 1
$\vec{k}$
1 2 3 4 5
Hash function
$h(x) = o + 3x \bmod 12$
0 0 0 0 0 $\vec{o}$
$\vec{h}$
3 6 9 0 3
$\vec{m}$
0 1 1 1 1
$\vec{T}_{keys}$

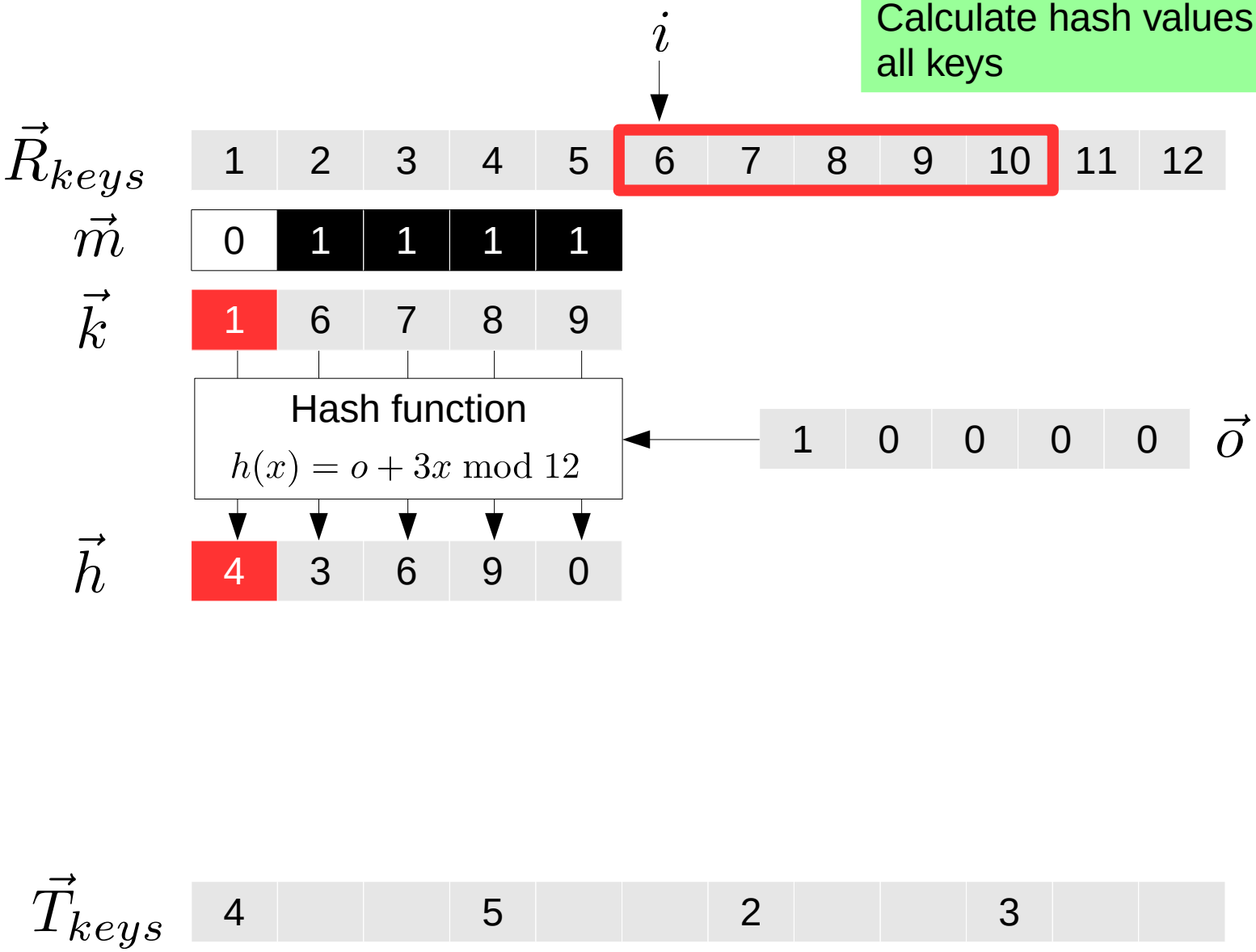

Increment offset for colliding entries
i
$\vec{R}_{keys}$
1 2 3 4 5 6 7 8 9 10 11 12
$\vec{m}$
0 1 1 1 1
$\vec{k}$
1 2 3 4 5
Hash function
$h(x) = o + 3x \bmod 12$
1 0 0 0 0 $\vec{o}$
$\vec{h}$
3 6 9 0 3
$\vec{m}$
0 1 1 1 1
$\vec{T}_{keys}$


Scatter successfully hashed values to buckets
i
$\vec{R}_{keys}$
1 2 3 4 5 6 7 8 9 10 11 12
$\vec{m}$
0 1 1 1 1
$\vec{k}$
1 2 3 4 5
Hash function
$h(x) = o + 3x \bmod 12$
1 0 0 0 0 $\vec{o}$
$\vec{h}$
3 6 9 0 3
$\vec{m}$
0 1 1 1 1
$\vec{T}_{keys}$
4 5 2 3

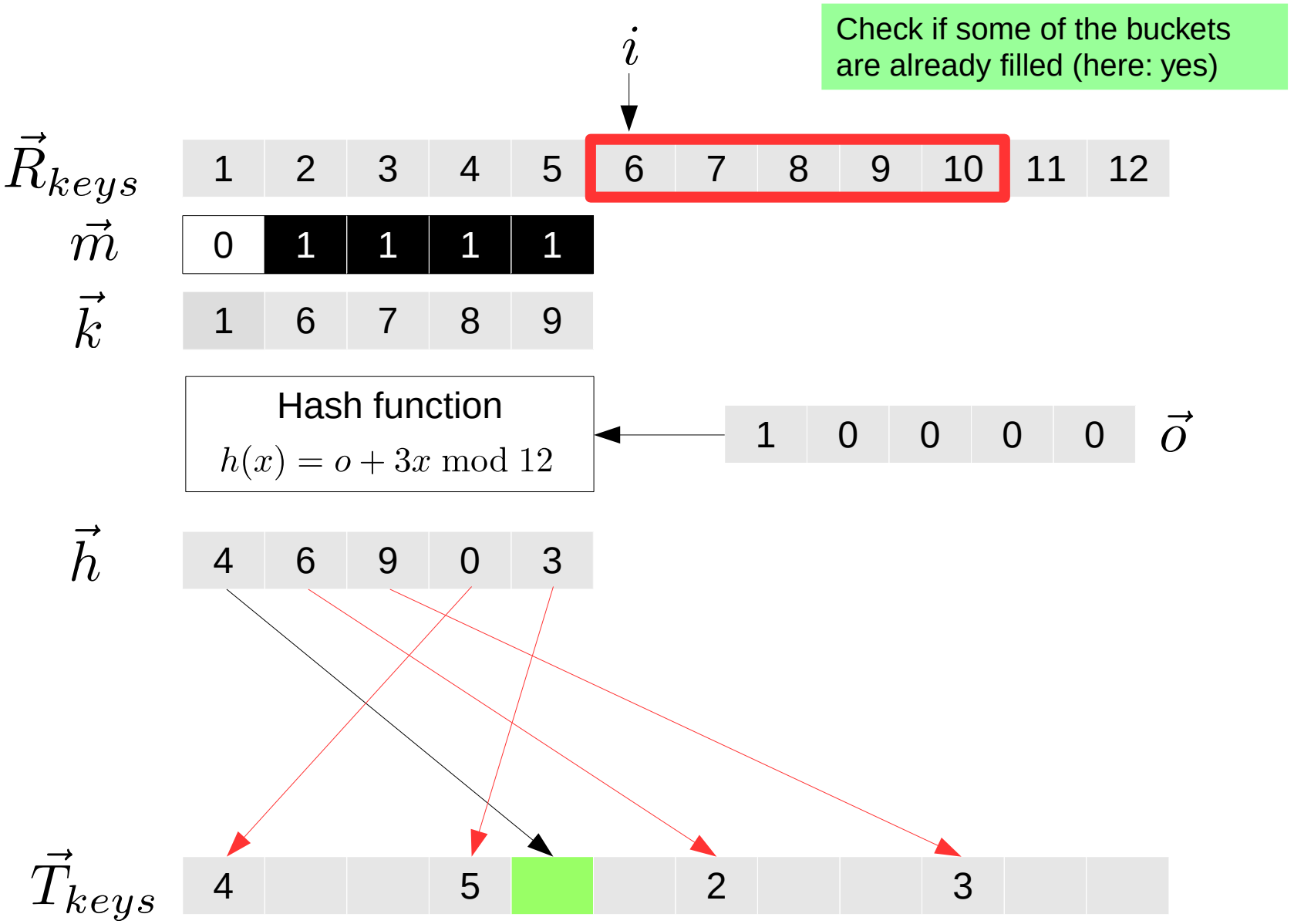

Advance i by the number of newly loaded keys in the previous step (here: 5)
i
$\vec{R}_{keys}$
1 2 3 4 5 6 7 8 9 10 11 12
$\vec{m}$
0 1 1 1 1
$\vec{k}$
1
Hash function
$h(x) = o + 3x \bmod 12$
1 0 0 0 0
$\vec{o}$
$\vec{h}$
$\vec{T}_{keys}$
4 5 2 3

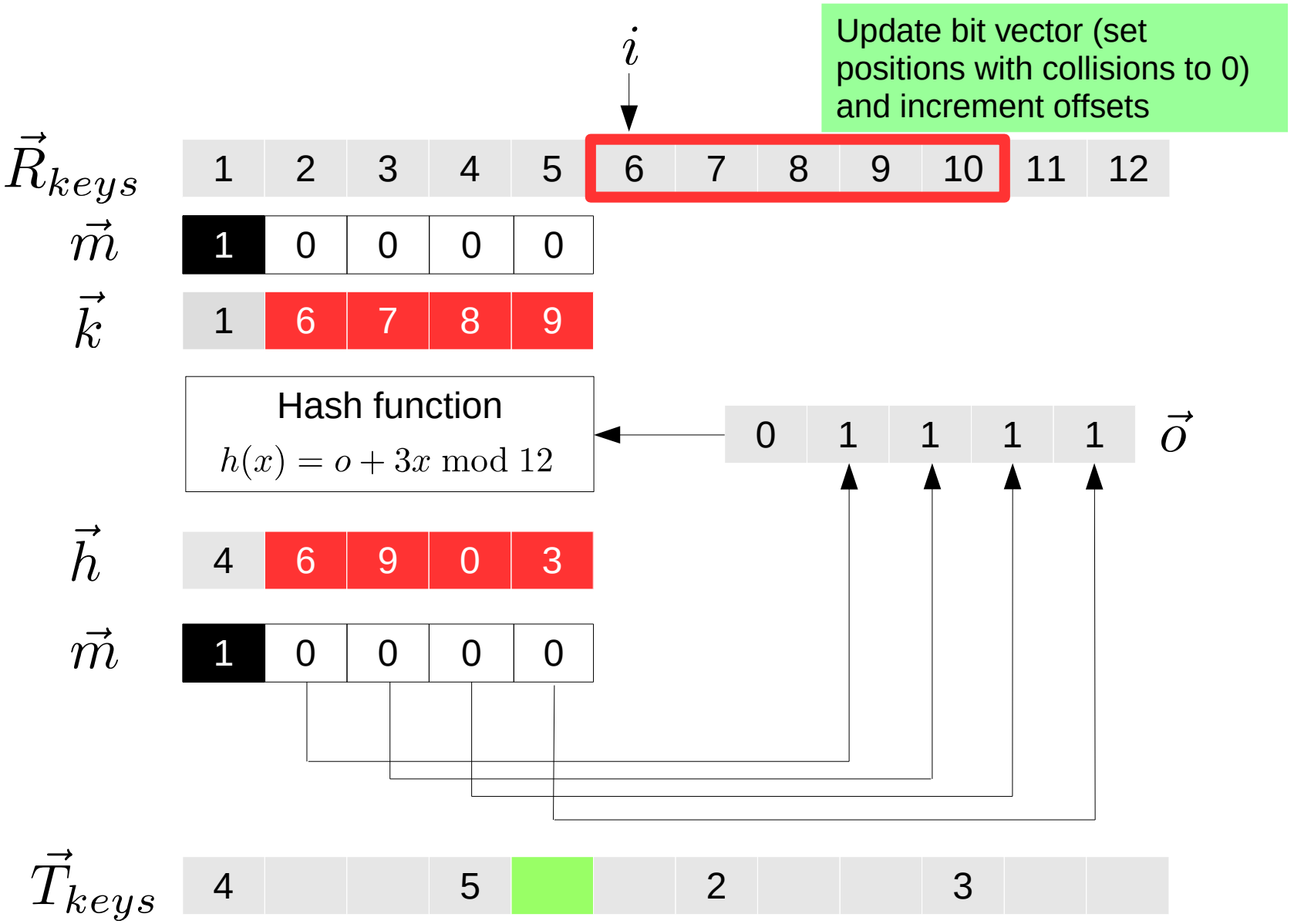

Selectively load tuples into key vector
i
$\vec{R}_{keys}$
1 2 3 4 5 6 7 8 9 10 11 12
$\vec{m}$
0 1 1 1 1
$\vec{k}$
1 6 7 8 9
Hash function
$h(x) = o + 3x \bmod 12$
1 0 0 0 0 $\vec{o}$
$\vec{h}$
$\vec{T}_{keys}$
4 5 2 3

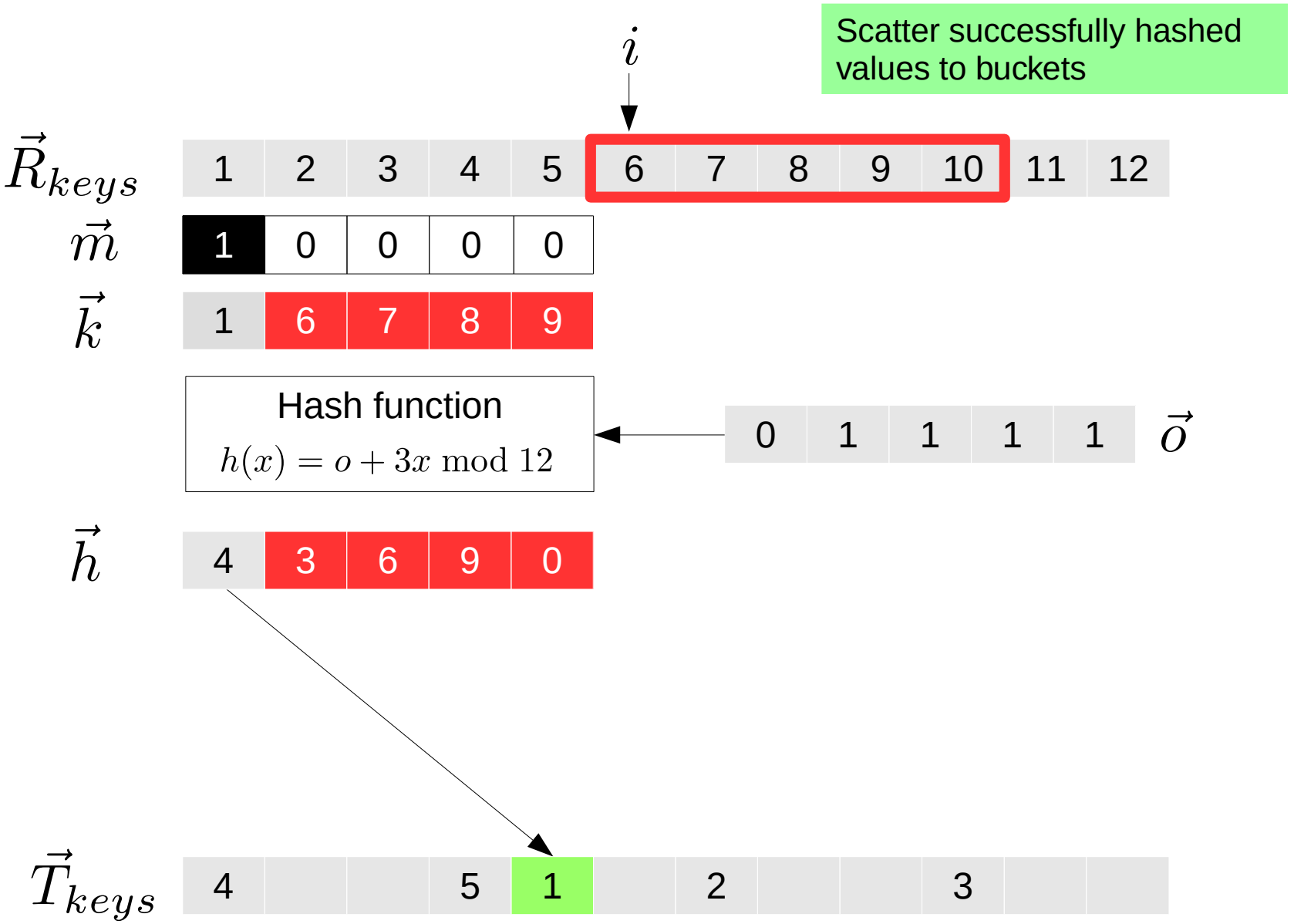
Calculate hash values for all keys

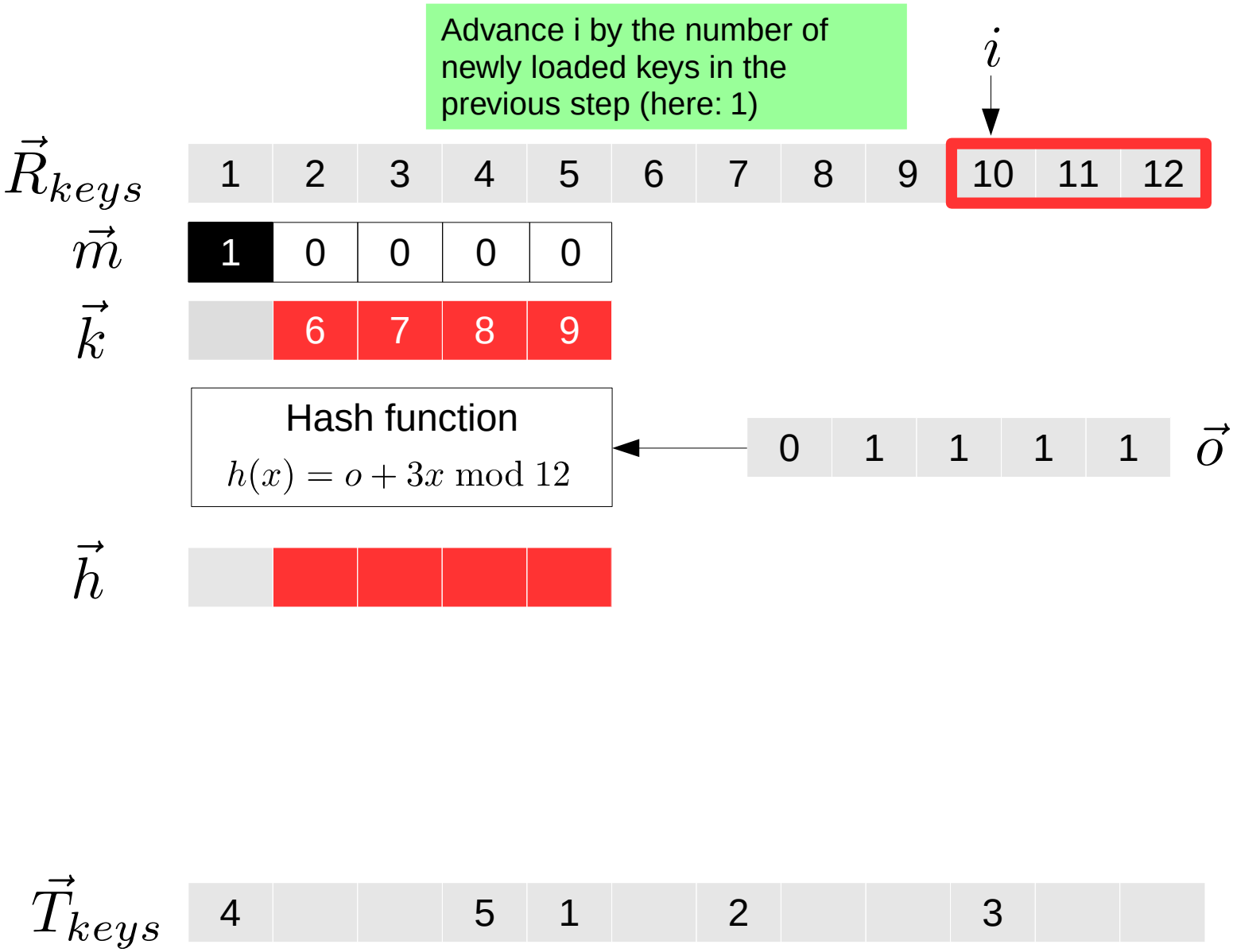


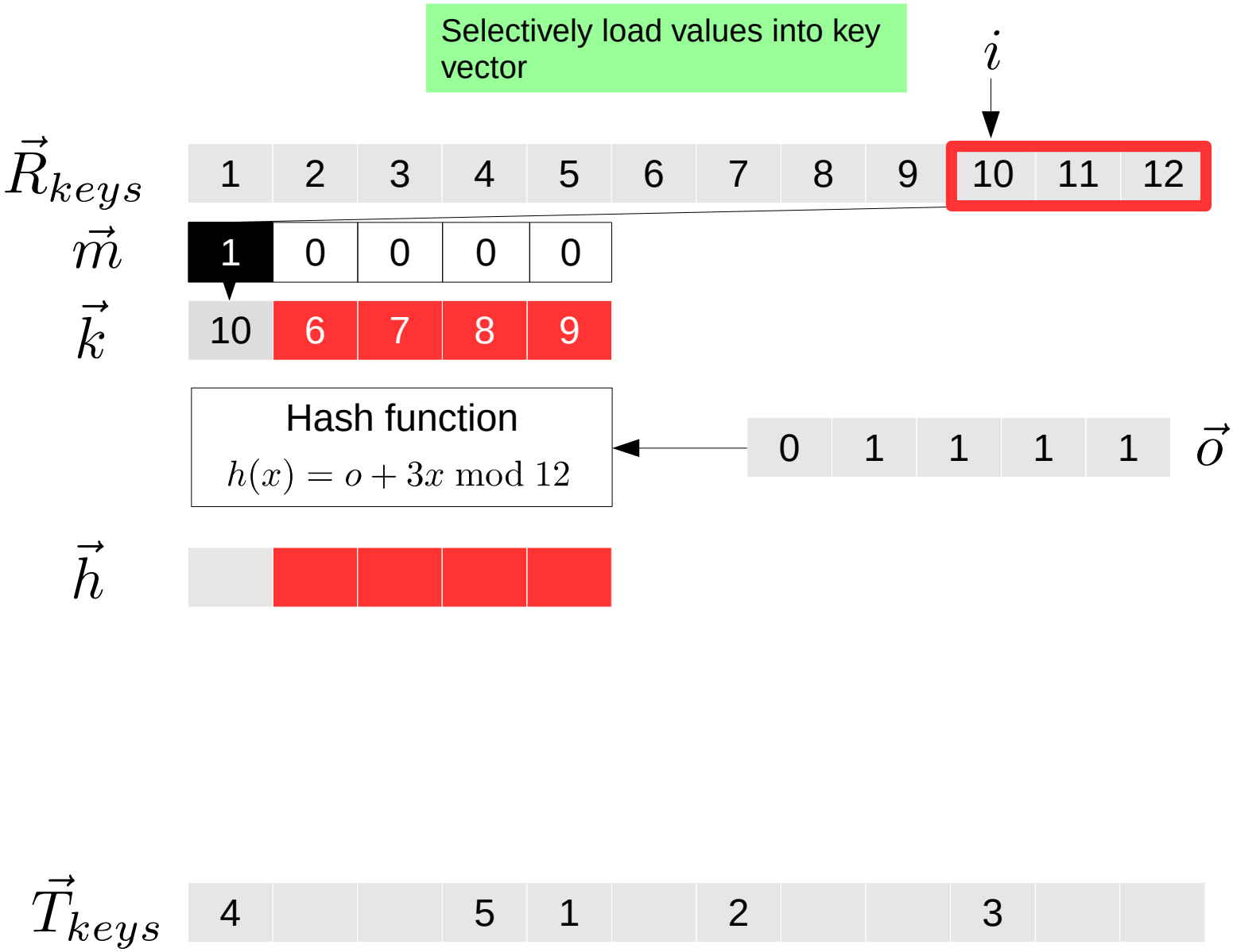
$\vec{T}_{keys}$ 4 5 2 3

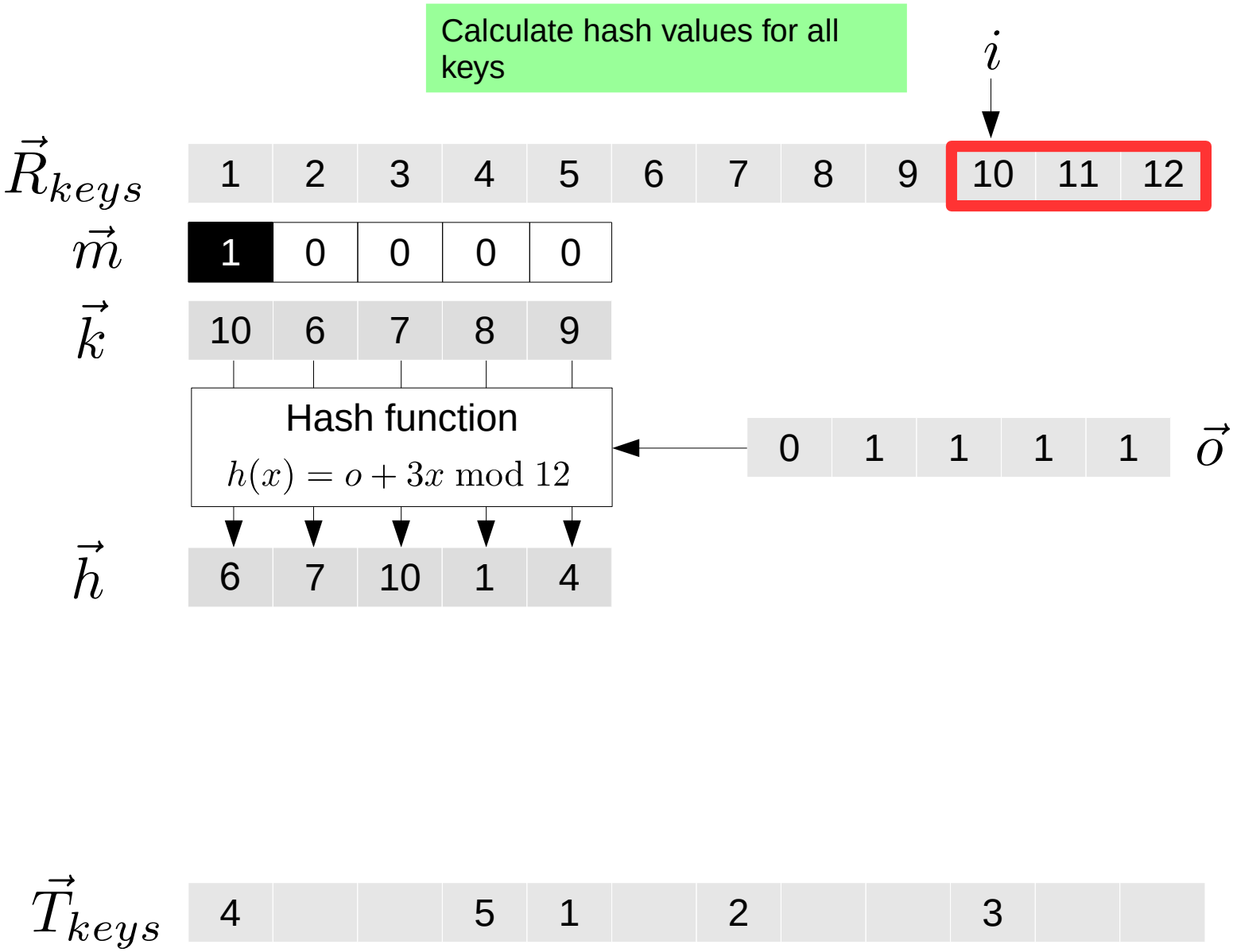

Check if some of the buckets are already filled (here: yes)
i
$\vec{R}_{keys}$
1 2 3 4 5 6 7 8 9 10 11 12
$\vec{m}$
0 1 1 1 1
$\vec{k}$
1 6 7 8 9
Hash function
$h(x) = o + 3x \bmod 12$
1 0 0 0 0 $\vec{o}$
$\vec{h}$
4 6 9 0 3
$\vec{T}_{keys}$
4 5 2 3

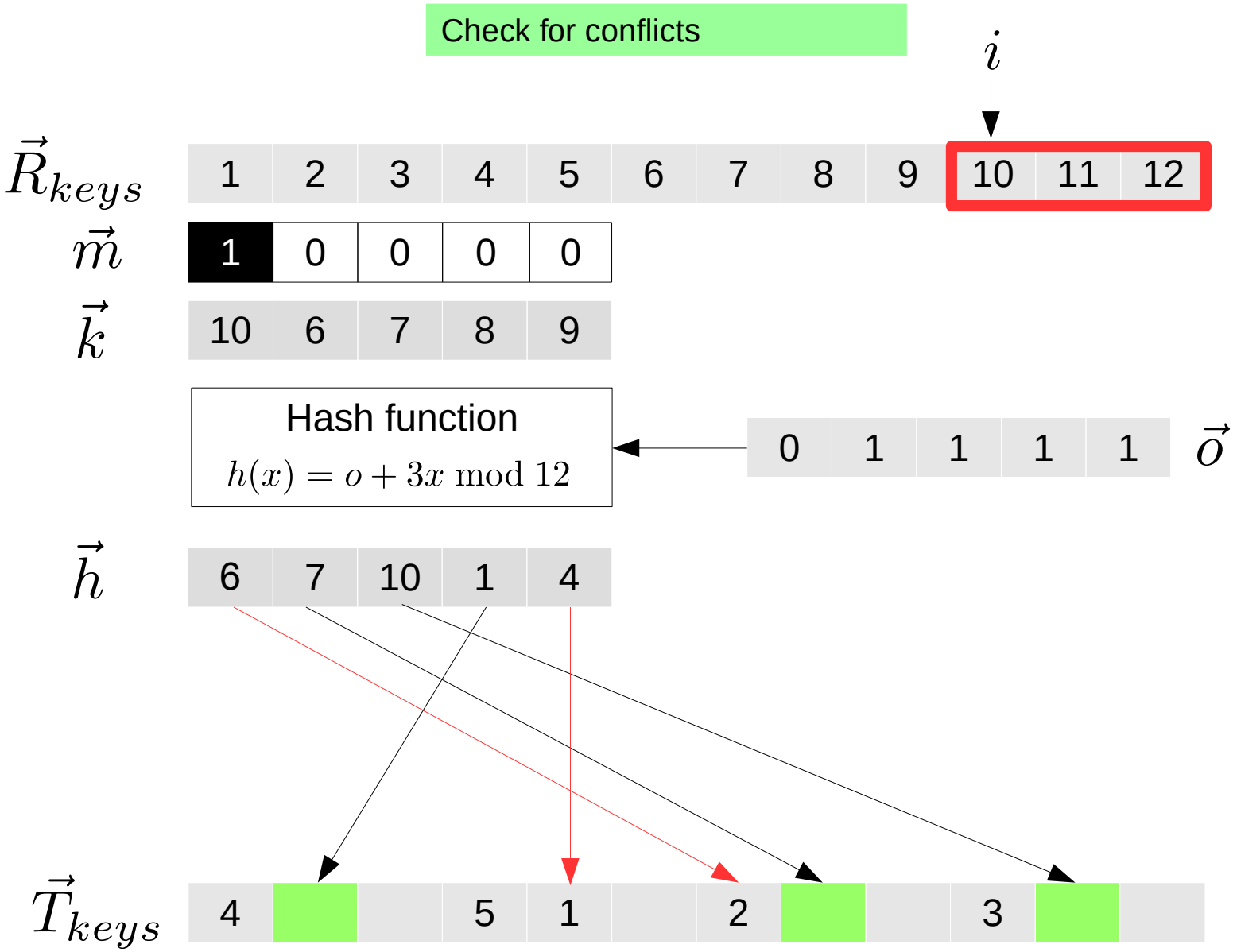

Update bit vector (set positions with collisions to 0) and increment offsets
i
$\vec{R}_{keys}$
1 2 3 4 5 6 7 8 9 10 11 12
$\vec{m}$
1 0 0 0 0
$\vec{k}$
1 6 7 8 9
Hash function
$h(x) = o + 3x \bmod 12$
0 1 1 1 1 $\vec{o}$
$\vec{h}$
4 6 9 0 3
$\vec{m}$
1 0 0 0 0
$\vec{T}_{keys}$
4 5 2 3

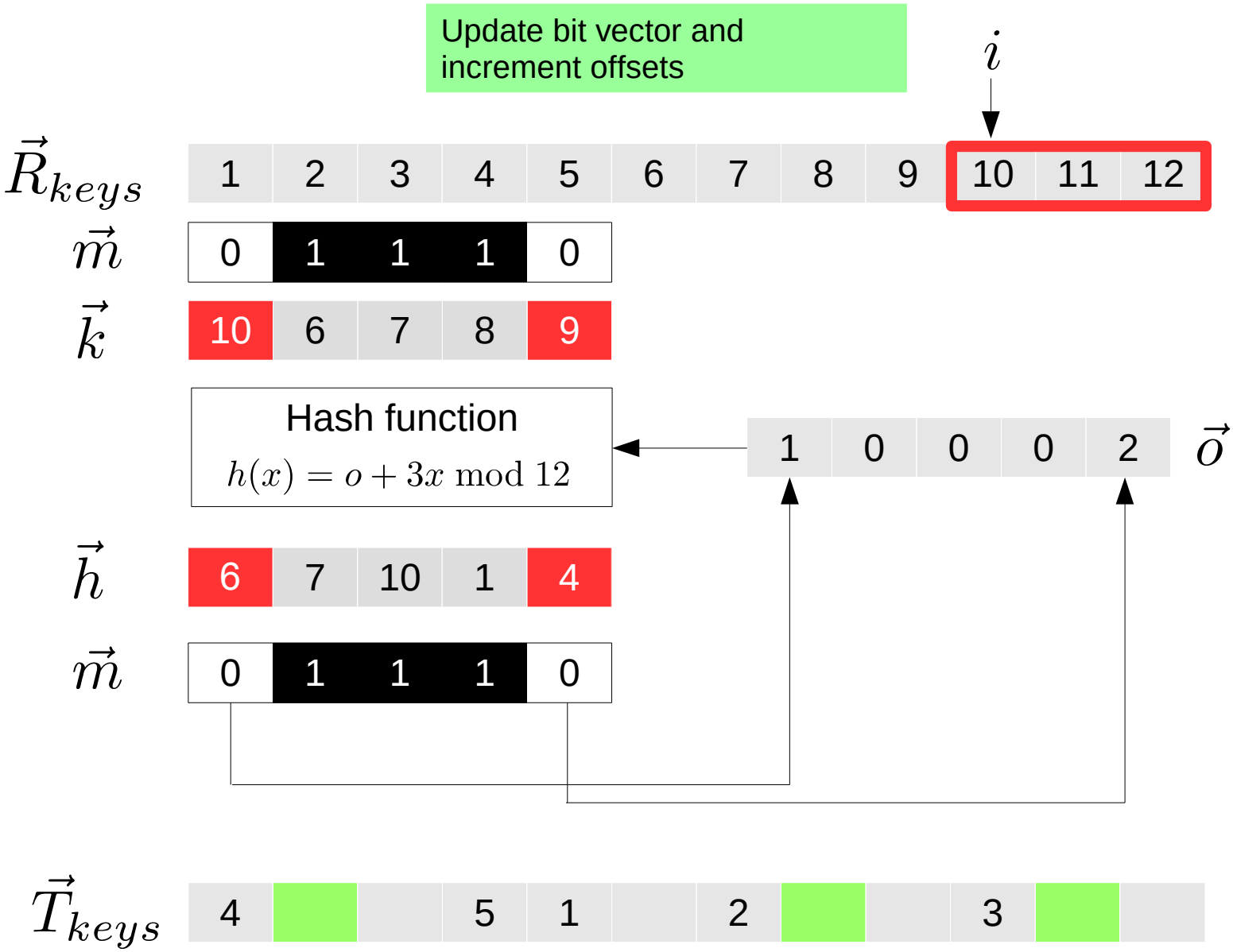

Scatter successfully hashed values to buckets
i
$\vec{R}_{keys}$
1 2 3 4 5 6 7 8 9 10 11 12
$\vec{m}$
1 0 0 0 0
$\vec{k}$
1 6 7 8 9
Hash function
$h(x) = o + 3x \bmod 12$
0 1 1 1 1
$\vec{o}$
$\vec{h}$
4 3 6 9 0
$\vec{T}_{keys}$
4 5 1 2 3

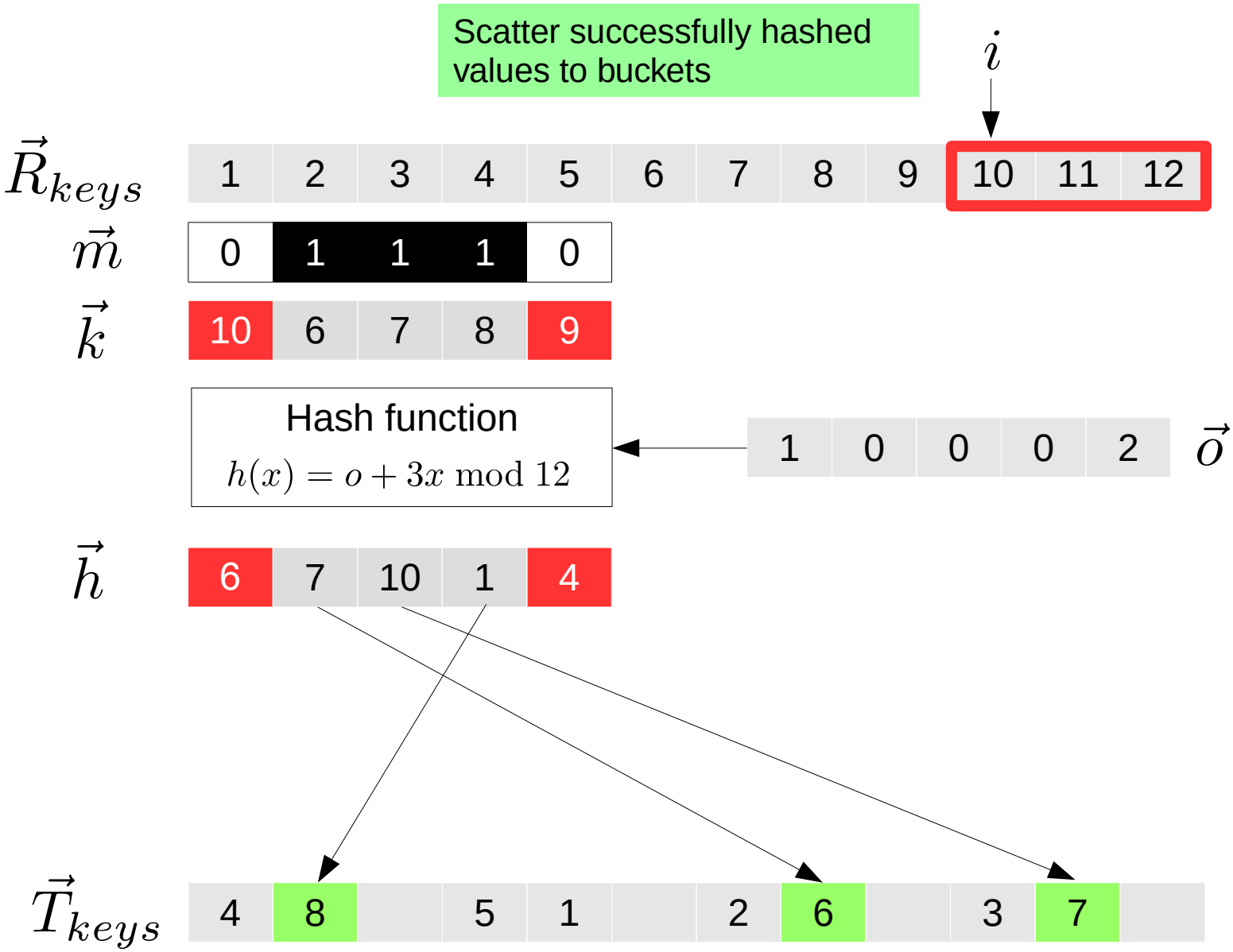

Advance i by the number of newly loaded keys in the previous step (here: 1)
i
$\vec{R}_{keys}$
1 2 3 4 5 6 7 8 9 10 11 12
$\vec{m}$
1 0 0 0 0
$\vec{k}$
6 7 8 9
Hash function
$h(x) = o + 3x \bmod 12$
0 1 1 1 1
$\vec{o}$
$\vec{h}$
$\vec{T}_{keys}$
4 5 1 2 3

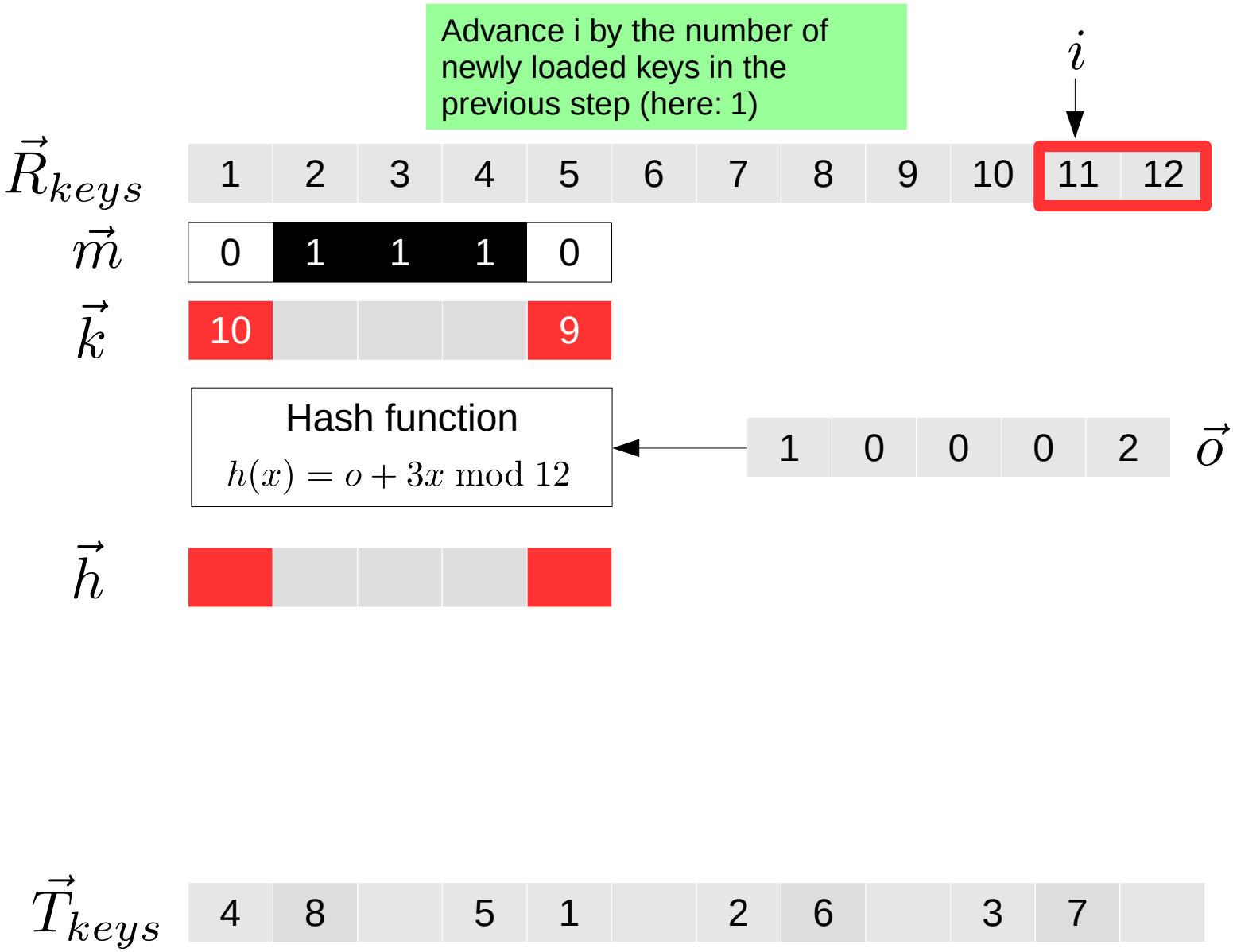

Selectively load values into key vector
i
$\vec{R}_{keys}$
1 2 3 4 5 6 7 8 9 10 11 12
$\vec{m}$
1 0 0 0 0
$\vec{k}$
10 6 7 8 9
Hash function
$h(x) = o + 3x \bmod 12$
0 1 1 1 1
$\vec{o}$
$\vec{h}$
$\vec{T}_{keys}$
4 5 1 2 3

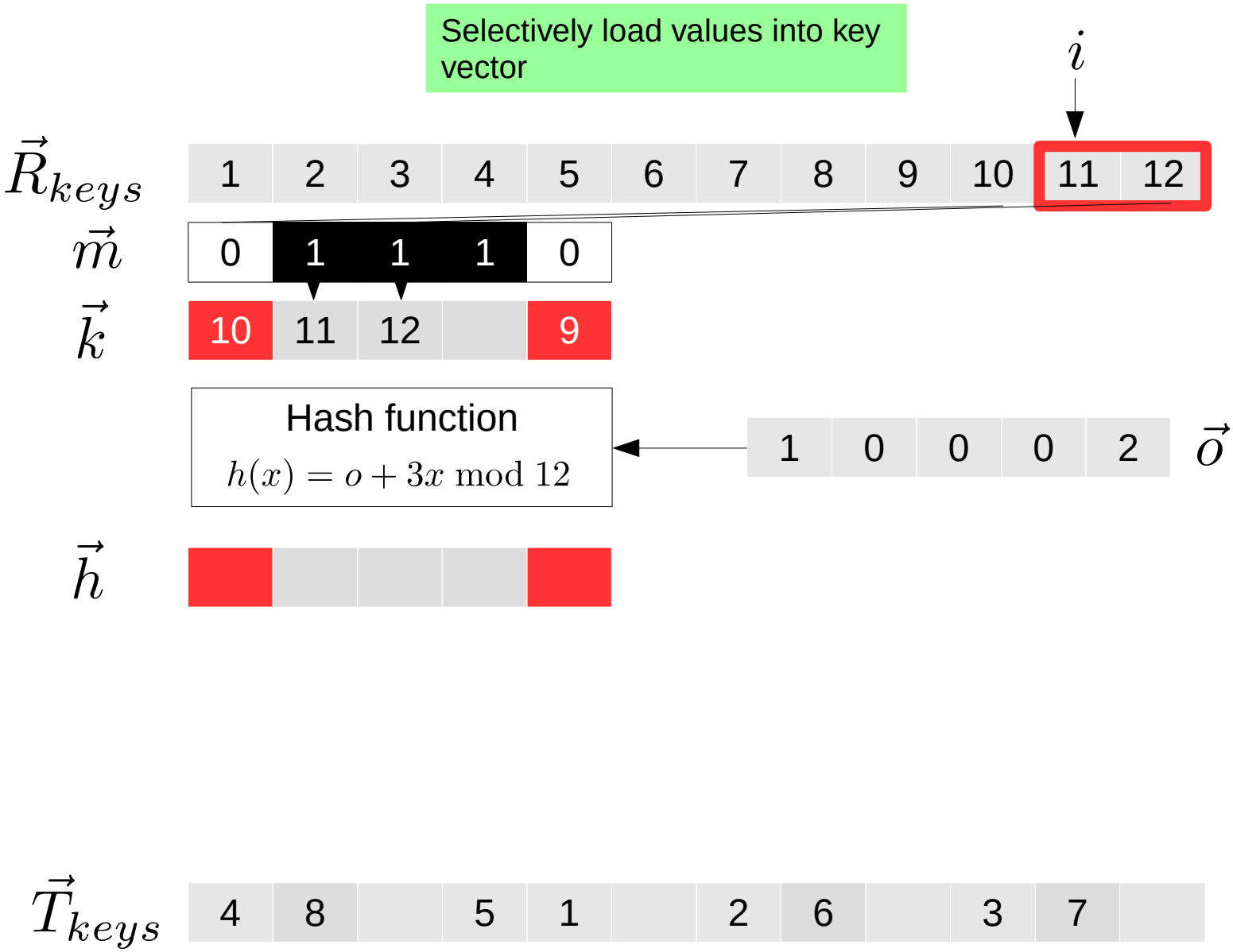

Calculate hash values for all keys
i
$\vec{R}_{keys}$
1 2 3 4 5 6 7 8 9 10 11 12
$\vec{m}$
1 0 0 0 0
$\vec{k}$
10 6 7 8 9
Hash function
$h(x) = o + 3x \bmod 12$
0 1 1 1 1 $\vec{o}$
$\vec{h}$
6 7 10 1 4
$\vec{T}_{keys}$
4 5 1 2 3

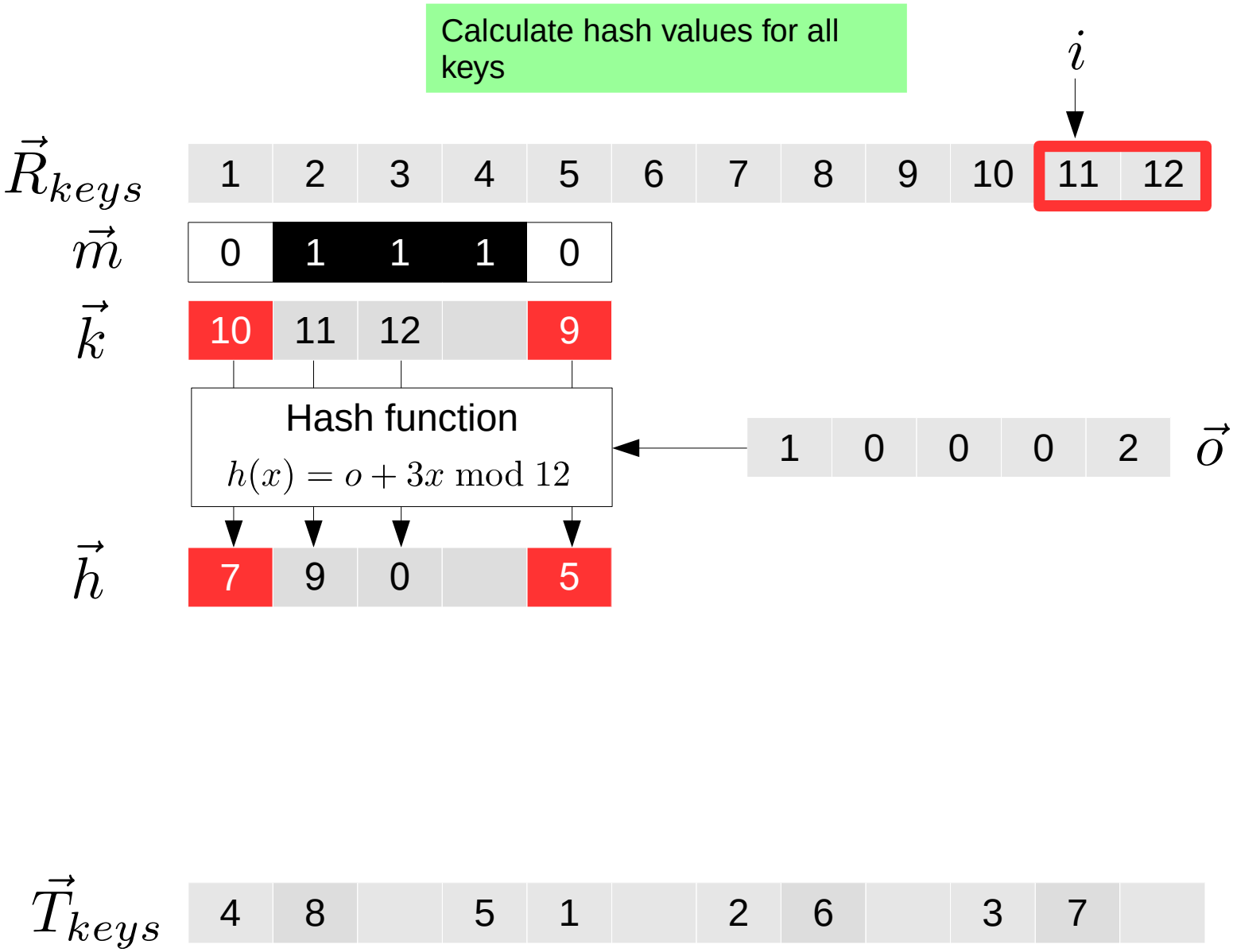

Check for conflicts
i
$\vec{R}_{keys}$
1 2 3 4 5 6 7 8 9 10 11 12
$\vec{m}$
1 0 0 0 0
$\vec{k}$
10 6 7 8 9
Hash function
$h(x) = o + 3x \bmod 12$
0 1 1 1 1 $\vec{o}$
$\vec{h}$
6 7 10 1 4
$\vec{T}_{keys}$
4 5 1 2 3

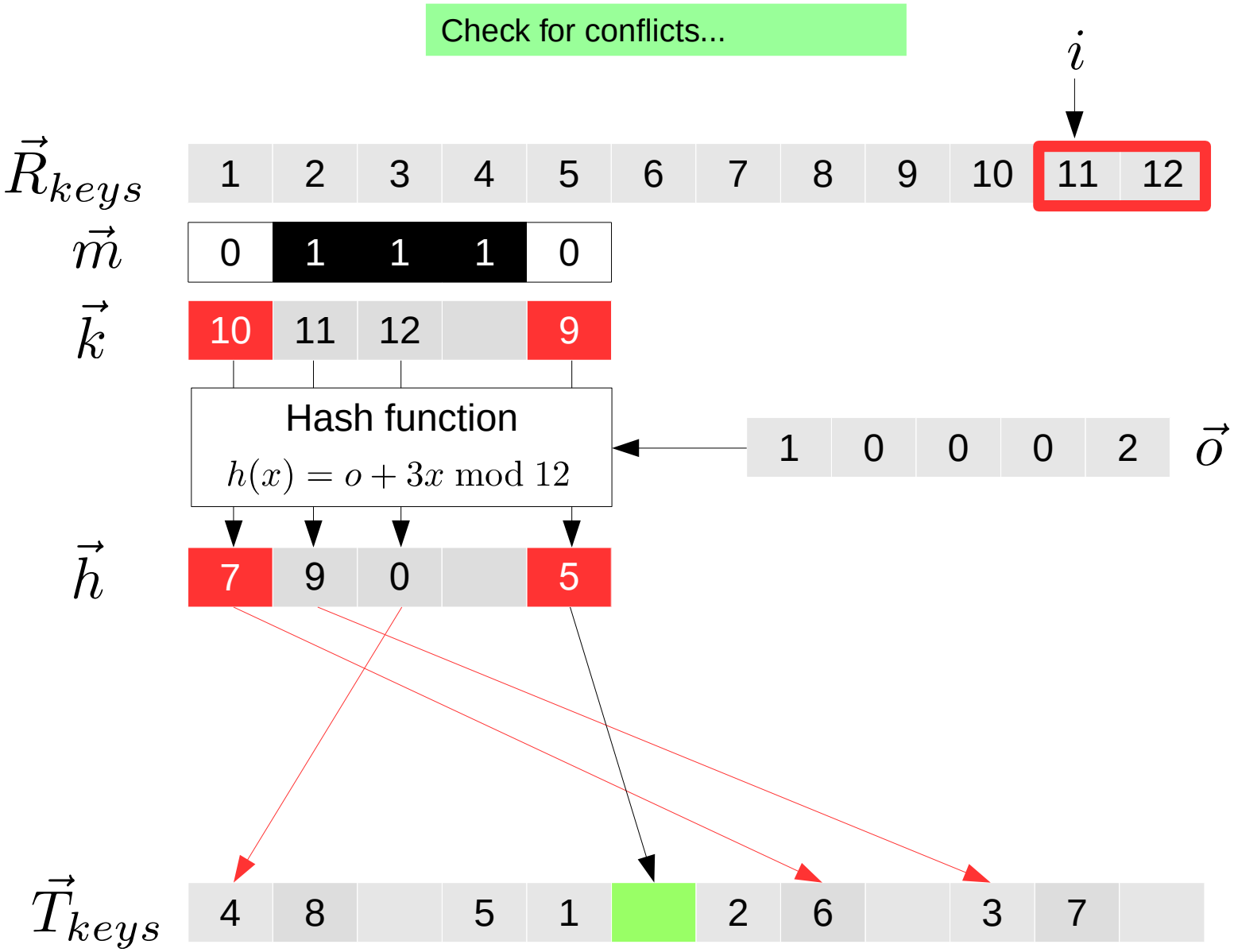

Update bit vector and increment offsets
i
$\vec{R}_{keys}$
1 2 3 4 5 6 7 8 9 10 11 12
$\vec{m}$
0 1 1 1 0
$\vec{k}$
10 6 7 8 9
Hash function
$h(x) = o + 3x \bmod 12$
1 0 0 0 2
$\vec{o}$
$\vec{h}$
6 7 10 1 4
$\vec{m}$
0 1 1 1 0
$\vec{T}_{keys}$
4 5 1 2 3


Scatter successfully hashed values to buckets
i
$\vec{R}_{keys}$
1 2 3 4 5 6 7 8 9 10 11 12
$\vec{m}$
0 1 1 1 0
$\vec{k}$
10 6 7 8 9
Hash function
$h(x) = o + 3x \bmod 12$
1 0 0 0 2
$\vec{o}$
$\vec{h}$
6 7 10 1 4
$\vec{T}_{keys}$
4 8 5 1 2 6 3 7


Advance i by the number of newly loaded keys in the previous step (here: 1)
i
$\vec{R}_{keys}$
1 2 3 4 5 6 7 8 9 10 11 12
$\vec{m}$
0 1 1 1 0
$\vec{k}$
10 9
Hash function
$h(x) = o + 3x \bmod 12$
1 0 0 0 2
$\vec{o}$
$\vec{h}$
$\vec{T}_{keys}$
4 8 5 1 2 6 3 7

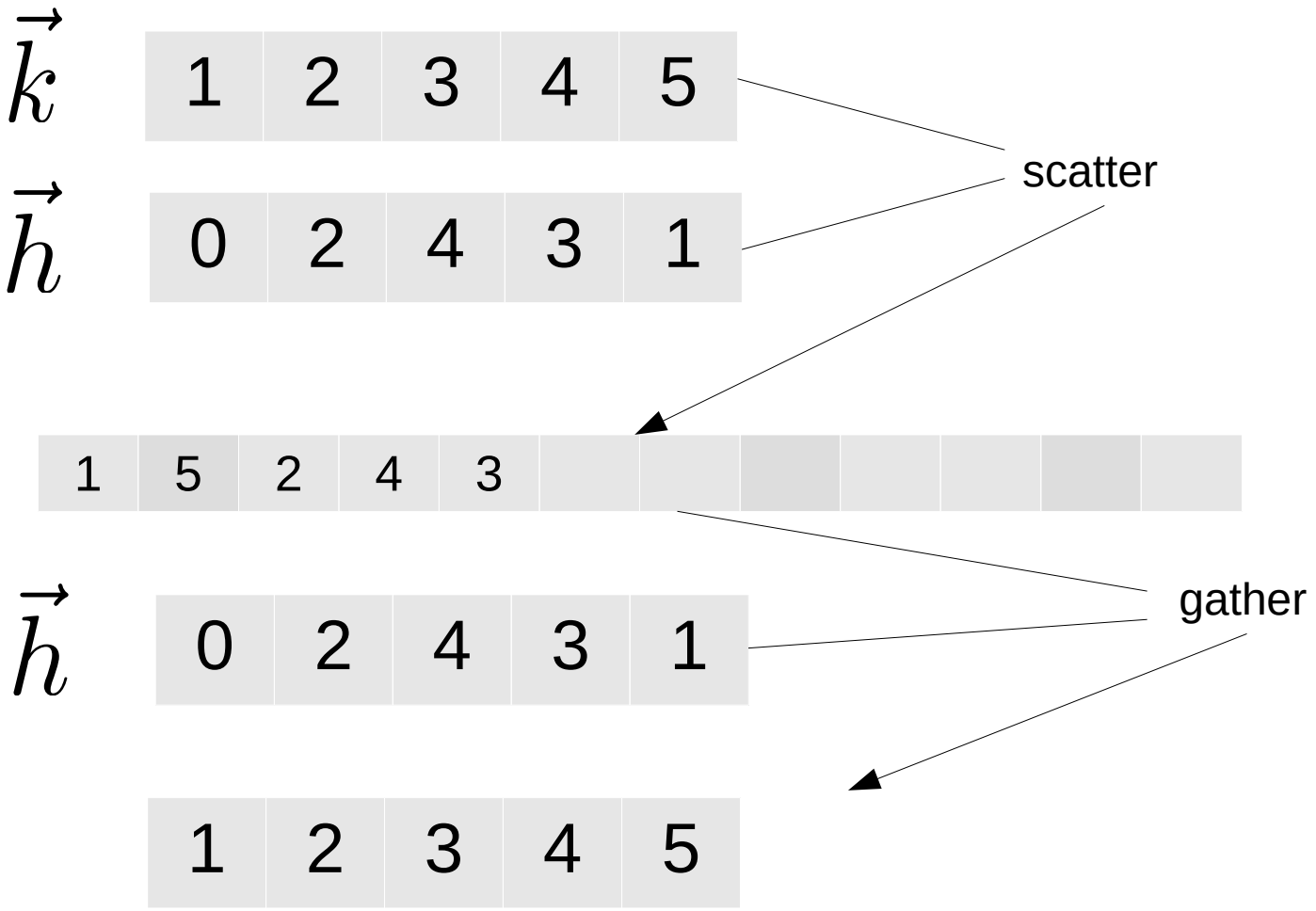

Selectively load values into key vector
i
$\vec{R}_{keys}$
1 2 3 4 5 6 7 8 9 10 11 12
$\vec{m}$
0 1 1 1 0
$\vec{k}$
10 11 12 9
Hash function
$h(x) = o + 3x \bmod 12$
1 0 0 0 2
$\vec{o}$
$\vec{h}$
$\vec{T}_{keys}$
4 8 5 1 2 6 3 7


Calculate hash values for all keys
i
$\vec{R}_{keys}$
1 2 3 4 5 6 7 8 9 10 11 12
$\vec{m}$
0 1 1 1 0
$\vec{k}$
10 11 12 9
Hash function
$h(x) = o + 3x \bmod 12$
1 0 0 0 2
$\vec{o}$
$\vec{h}$
7 9 0 5
$\vec{T}_{keys}$
4 8 5 1 2 6 3 7


Check for conflicts...
i
$\vec{R}_{keys}$
1 2 3 4 5 6 7 8 9 10 11 12
$\vec{m}$
0 1 1 1 0
$\vec{k}$
10 11 12 9
Hash function
$h(x) = o + 3x \bmod 12$
1 0 0 0 2
$\vec{o}$
$\vec{h}$
7 9 0 5
$\vec{T}_{keys}$
4 8 5 1 2 6 3 7

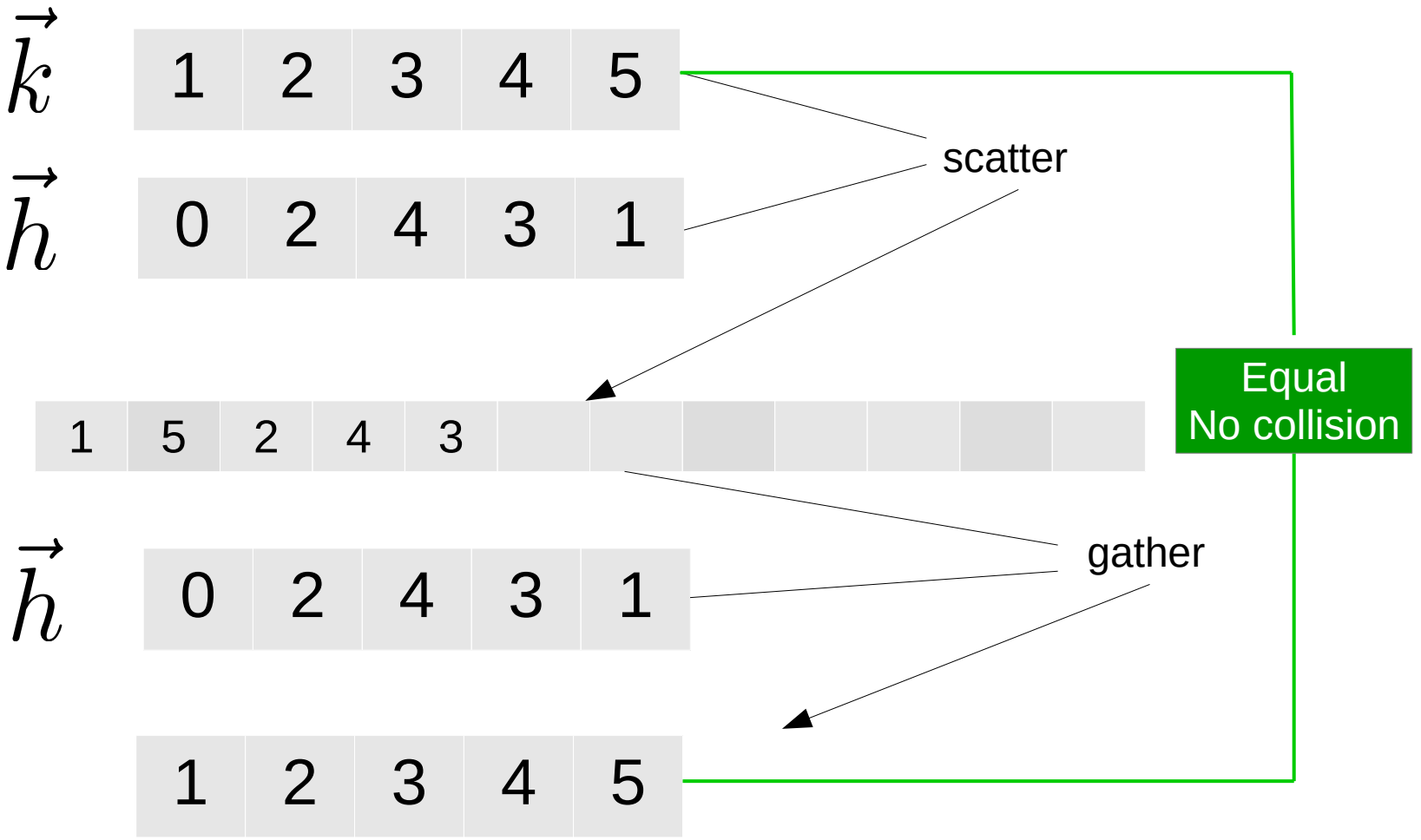
Collision Detection

Scatter + Gather operations on a vector containing unique values.



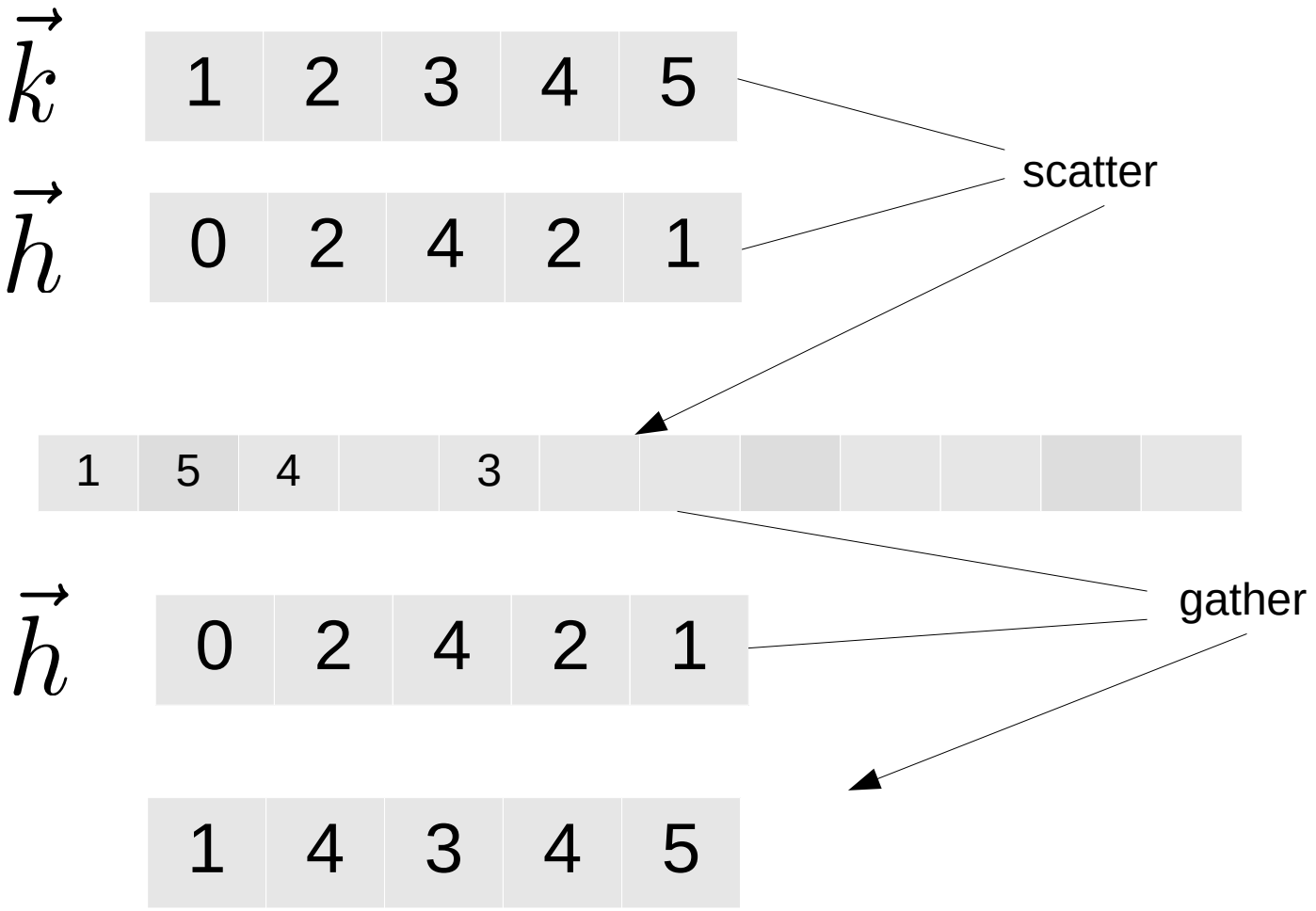
Collision Detection

Scatter + Gather operations on a vector containing unique values.



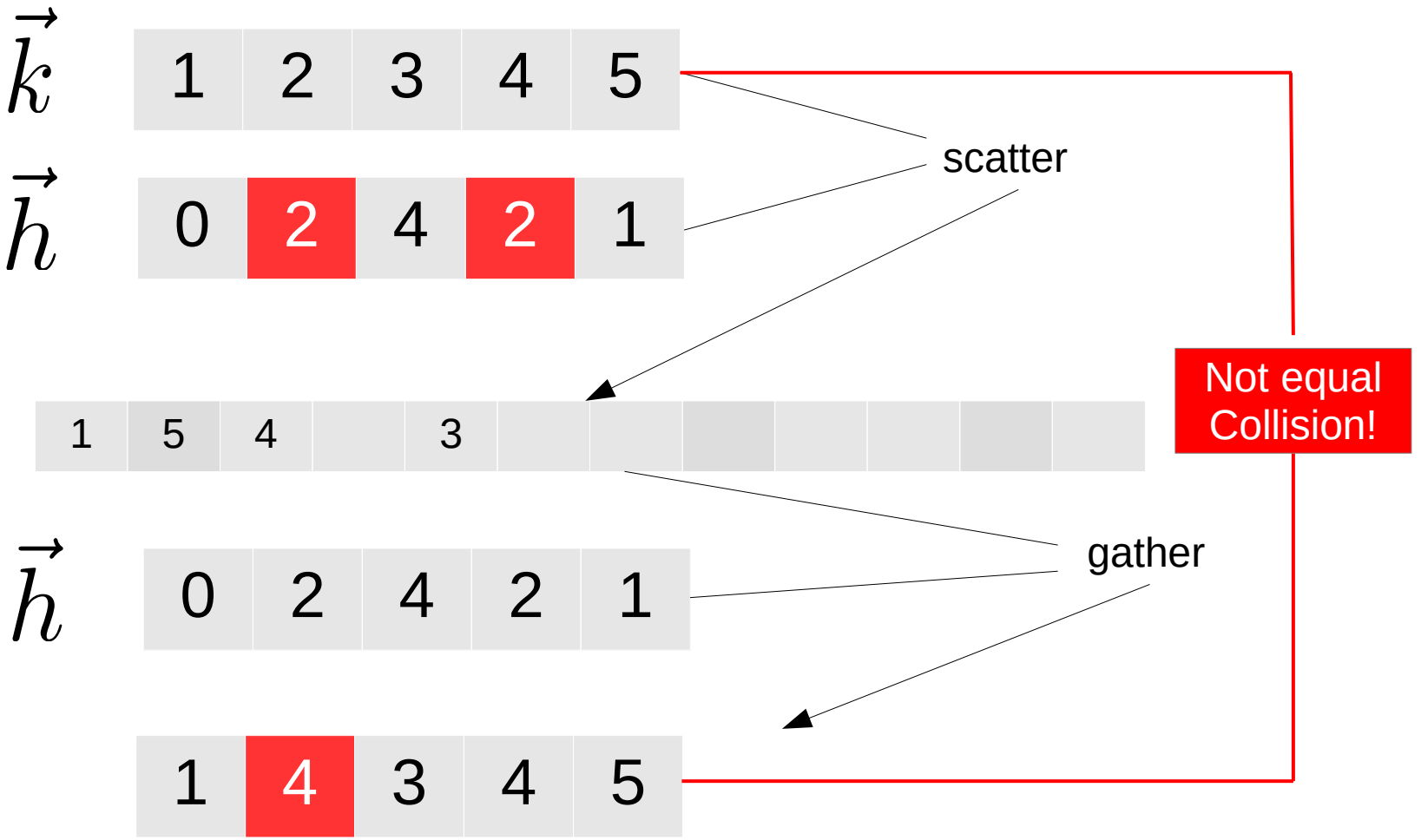
Collision Detection

Scatter + Gather operations on a vector containing unique values.



Collision Detection

Scatter + Gather operations on a vector containing unique values.



Linear Probing

$\vec{S}_{keys}$

7	4	10	5	8	19	15					

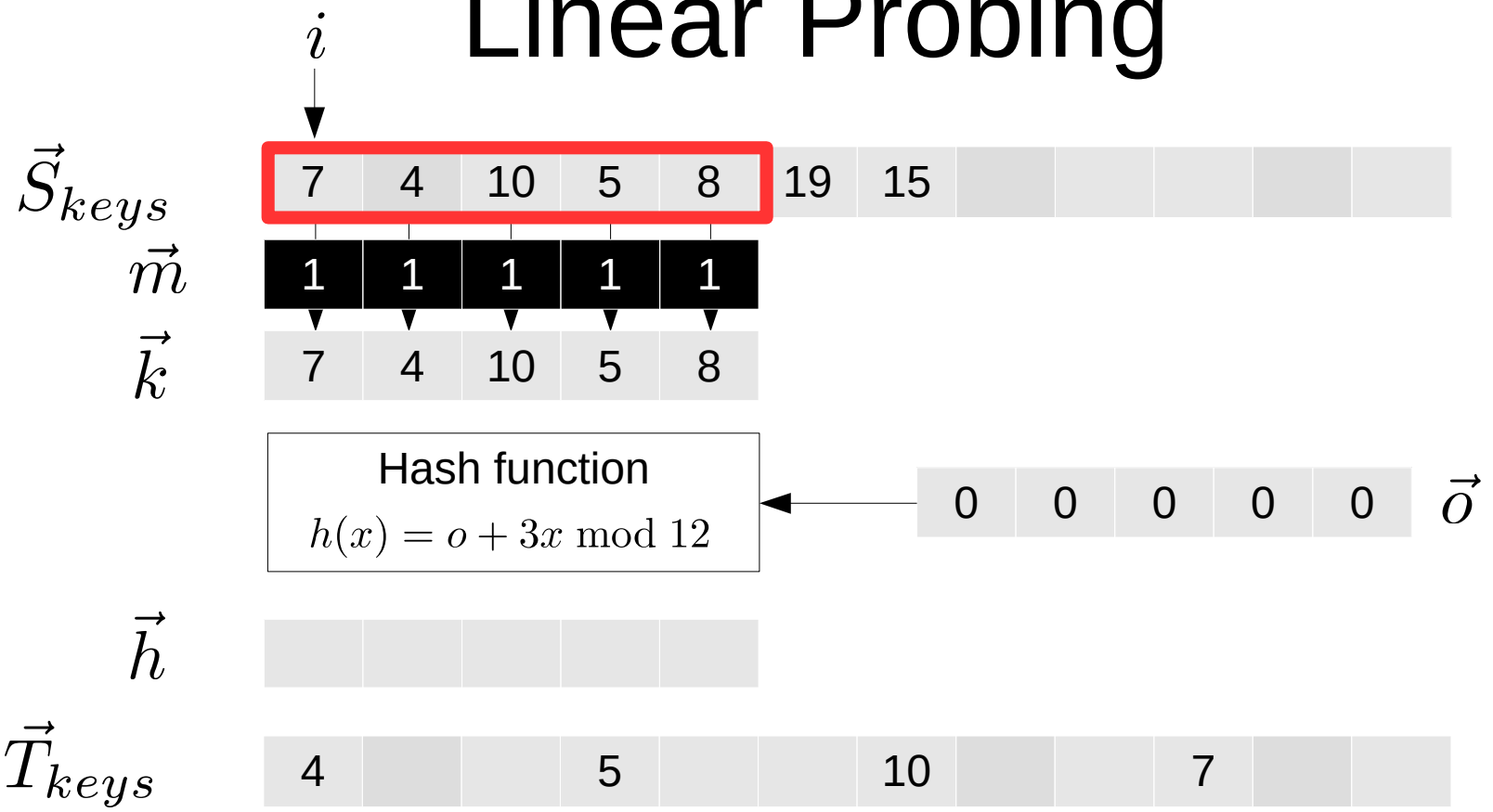
$\vec{T}_{keys}$

4			5			10			7		

R.key	R.payload	S.key	S.payload
...	...	...	...

Selectively load keys into vector

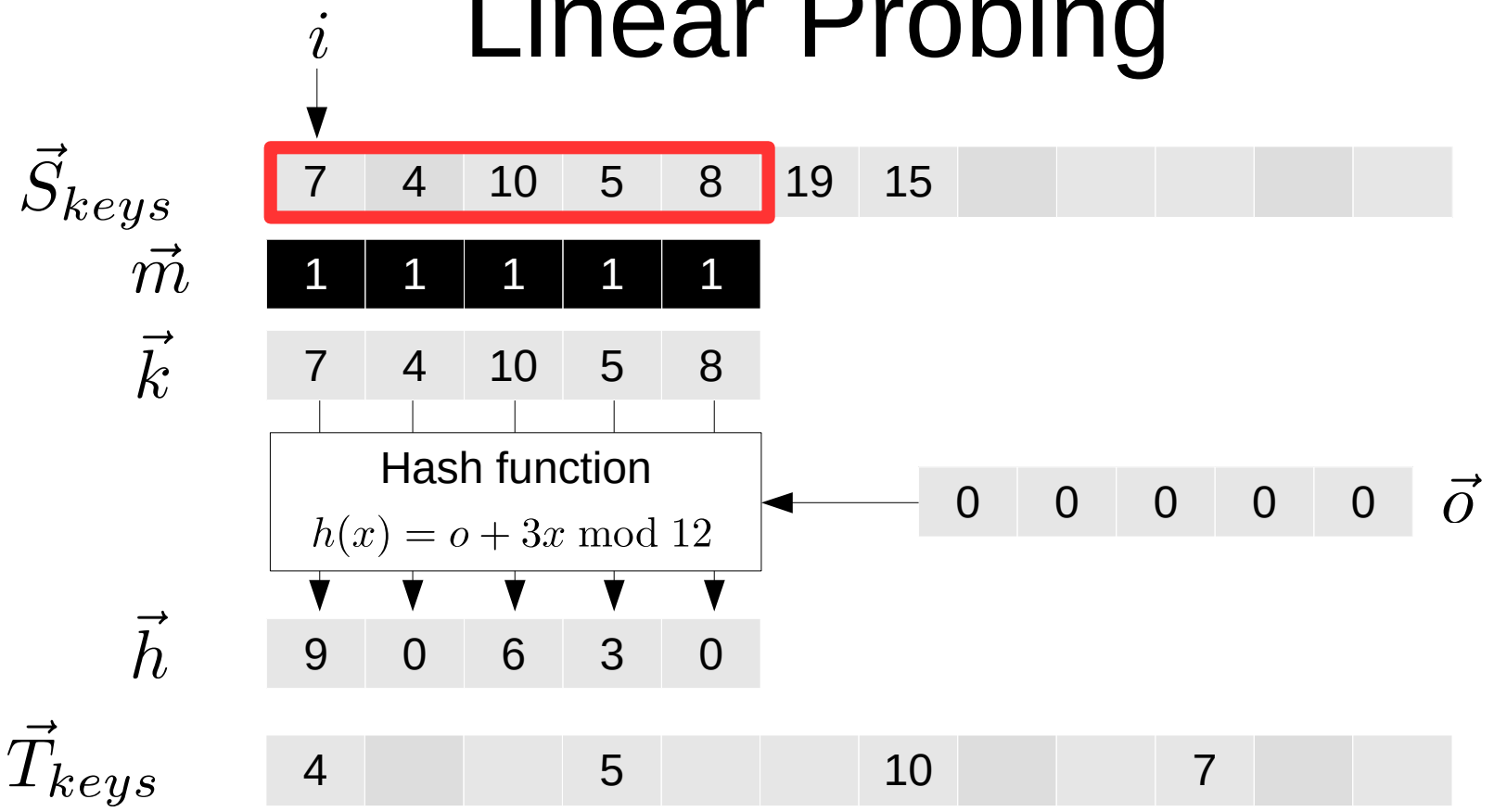
Linear Probing



R.key	R.payload	S.key	S.payload
...	...	...	...

Calculate hash values for keys

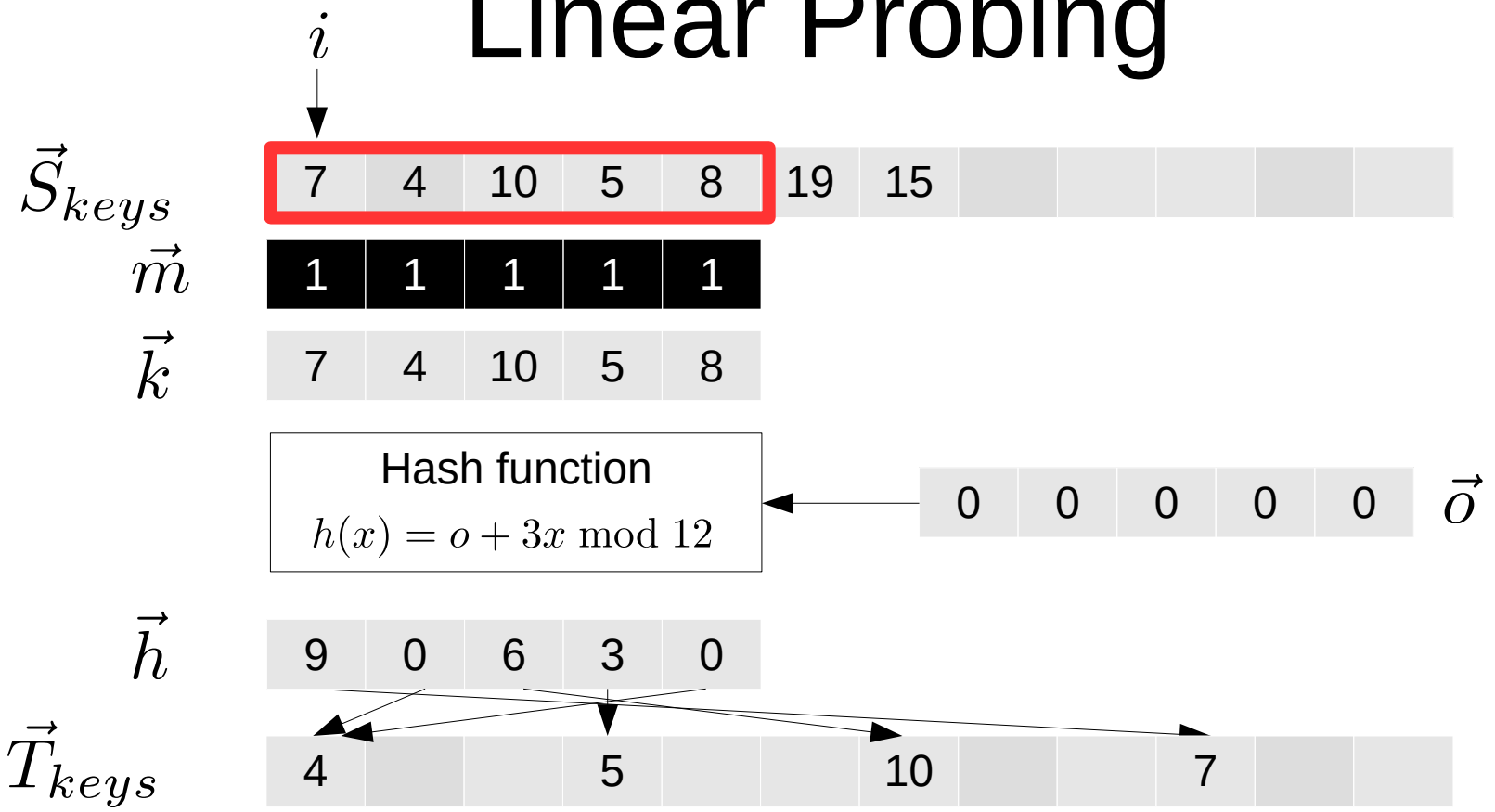
Linear Probing



R.key	R.payload	S.key	S.payload
...	...	...	...

Match hash values with existing buckets

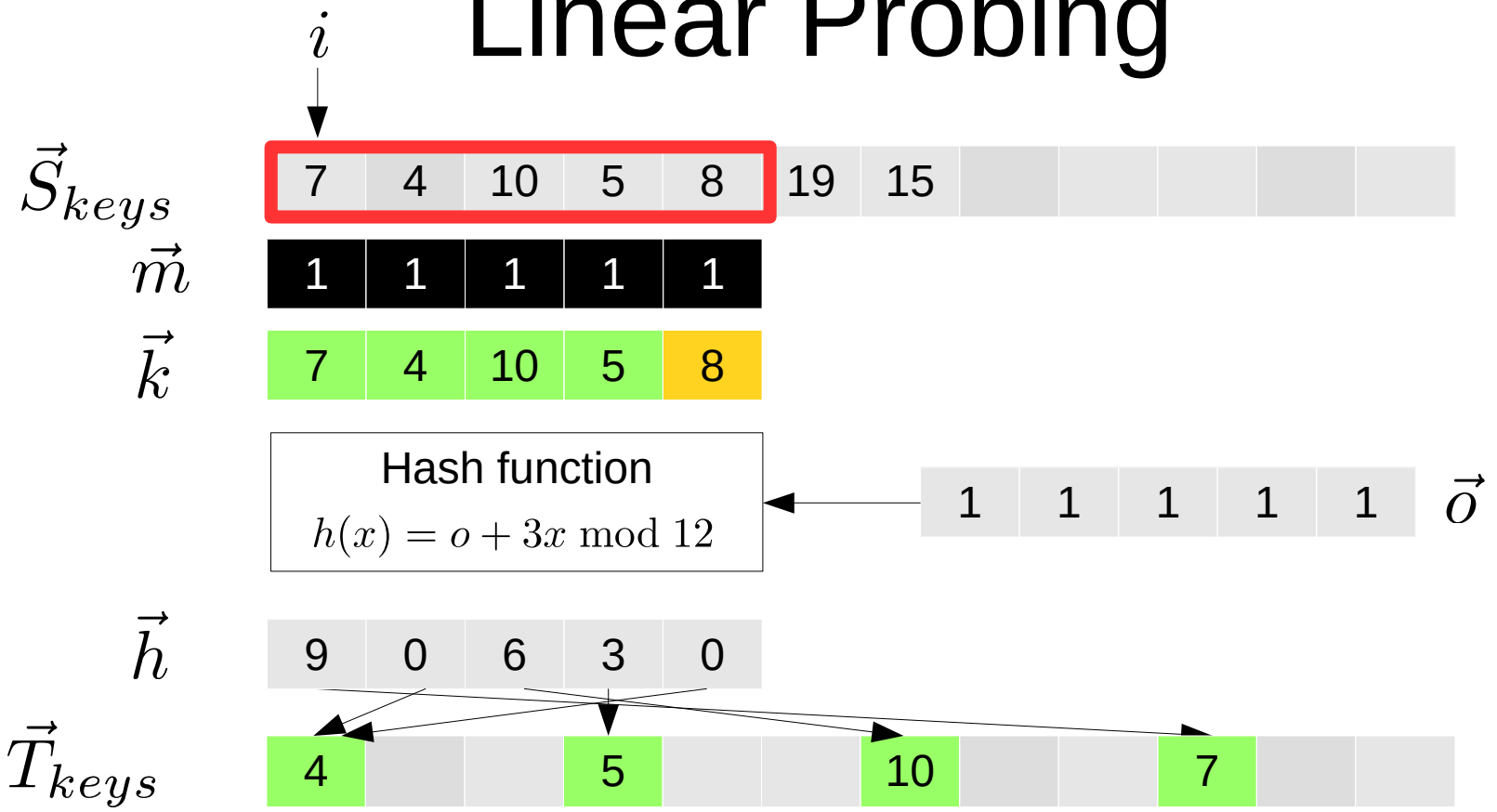
Linear Probing



R.key	R.payload	S.key	S.payload
...	...	...	...

Check which values have matches (here: all)

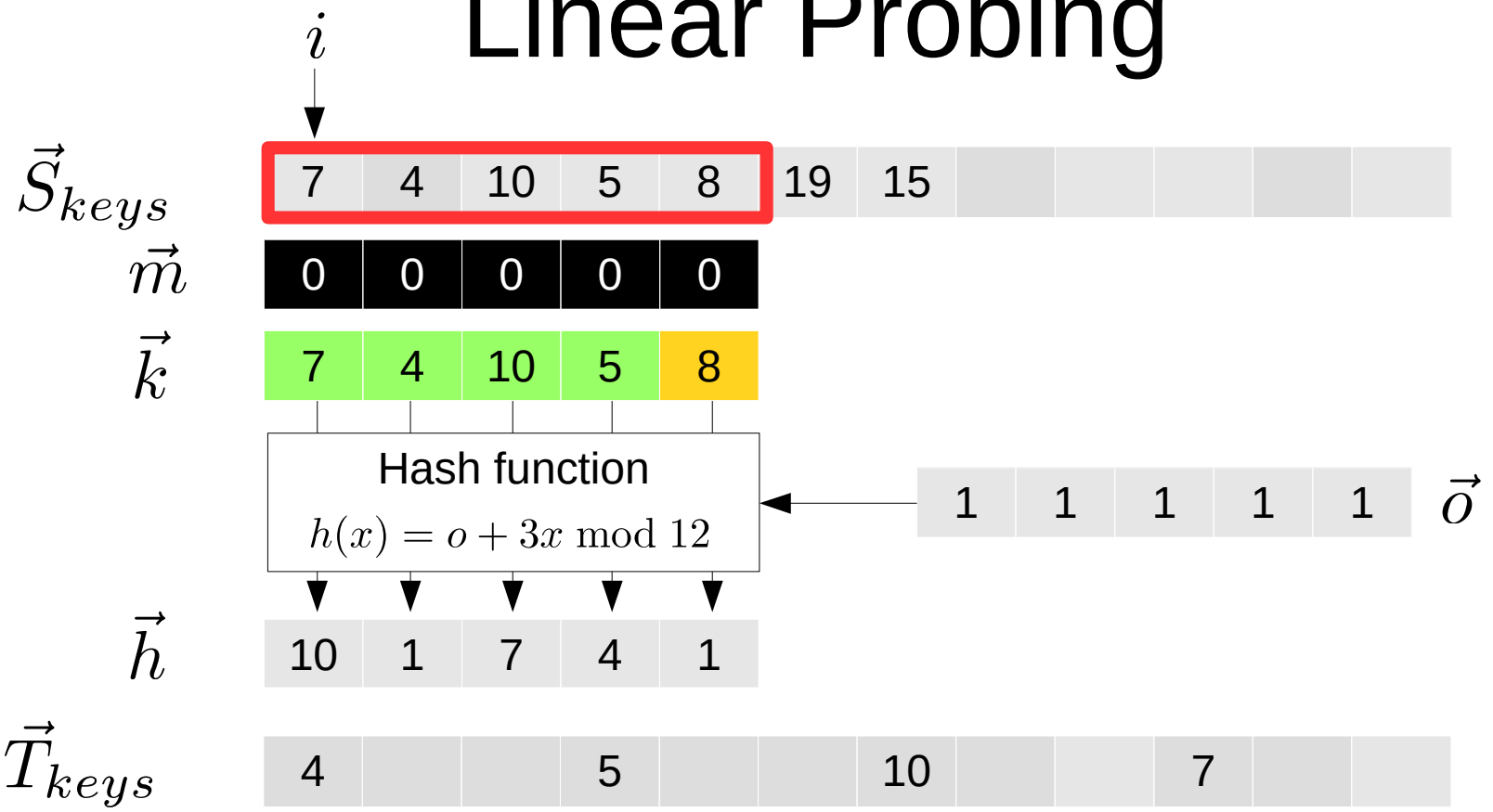
Linear Probing



R.key	R.payload	S.key	S.payload
7	5643	7	7861348
10	864	10	8615348
4	15645234	4	67453678
5	45345	5	453453453
...	...	...	...

Calculate hash values for keys

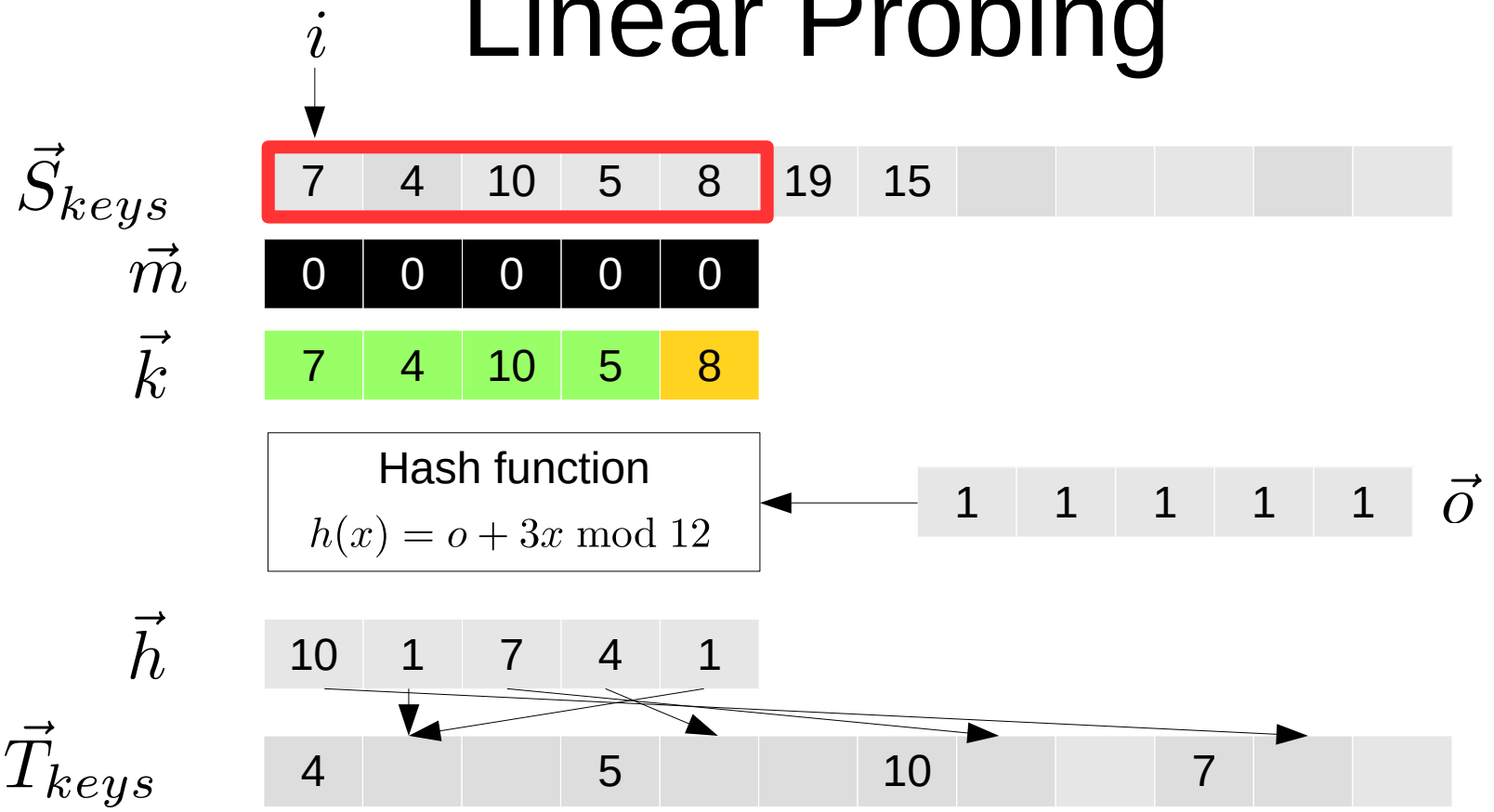
Linear Probing



R.key	R.payload	S.key	S.payload
7	5643	7	7861348
10	864	10	8615348
4	15645234	4	67453678
5	45345	5	453453453
...	...	...	...

Calculate hash values for keys

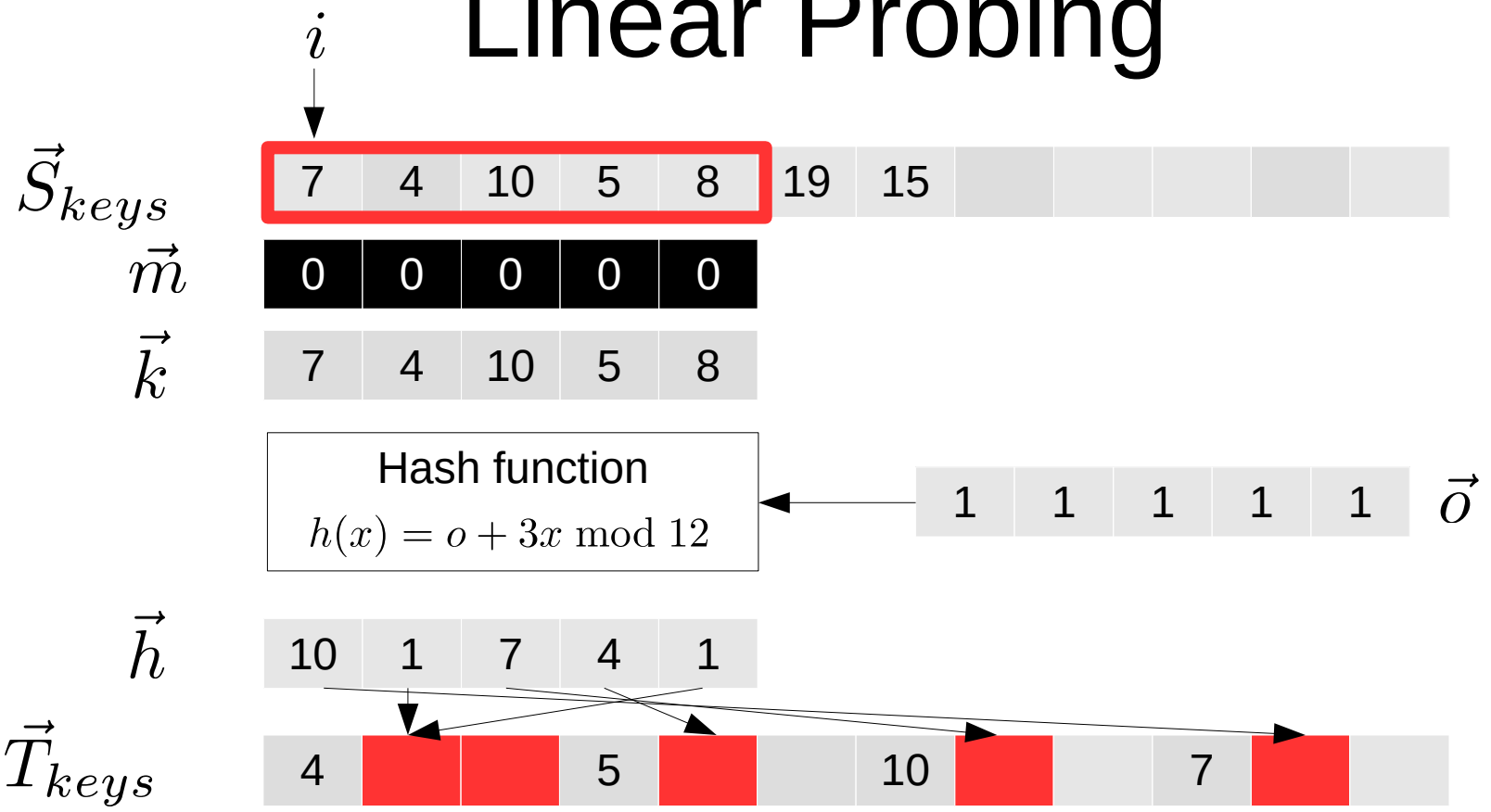
Linear Probing



R.key	R.payload	S.key	S.payload
7	5643	7	7861348
10	864	10	8615348
4	15645234	4	67453678
5	45345	5	453453453
...	...	...	...

All calculated buckets are empty →done.

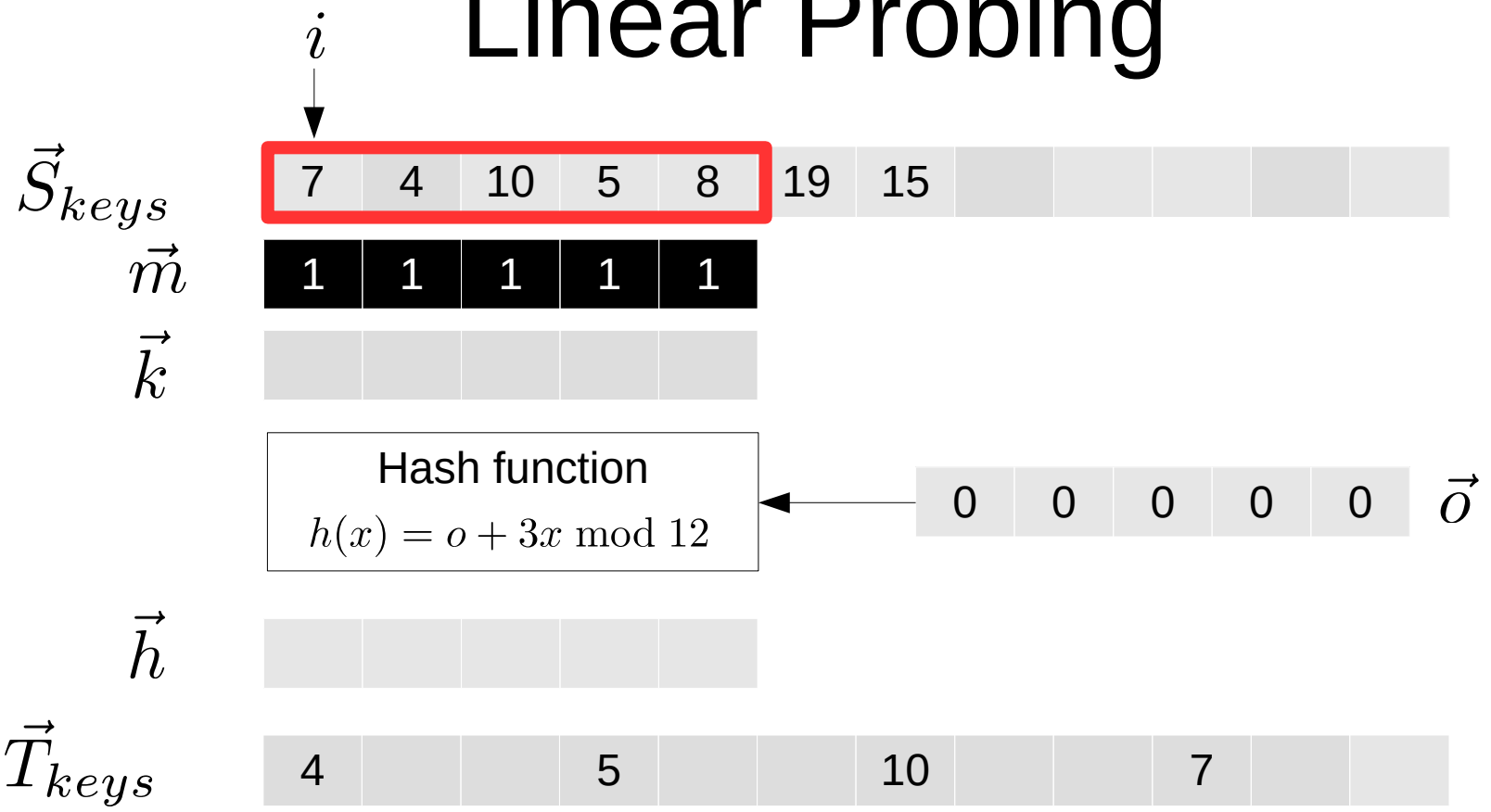
Linear Probing



R.key	R.payload	S.key	S.payload
7	5643	7	7861348
10	864	10	8615348
4	15645234	4	67453678
5	45345	5	453453453
...	...	...	...

All calculated buckets are empty →done.

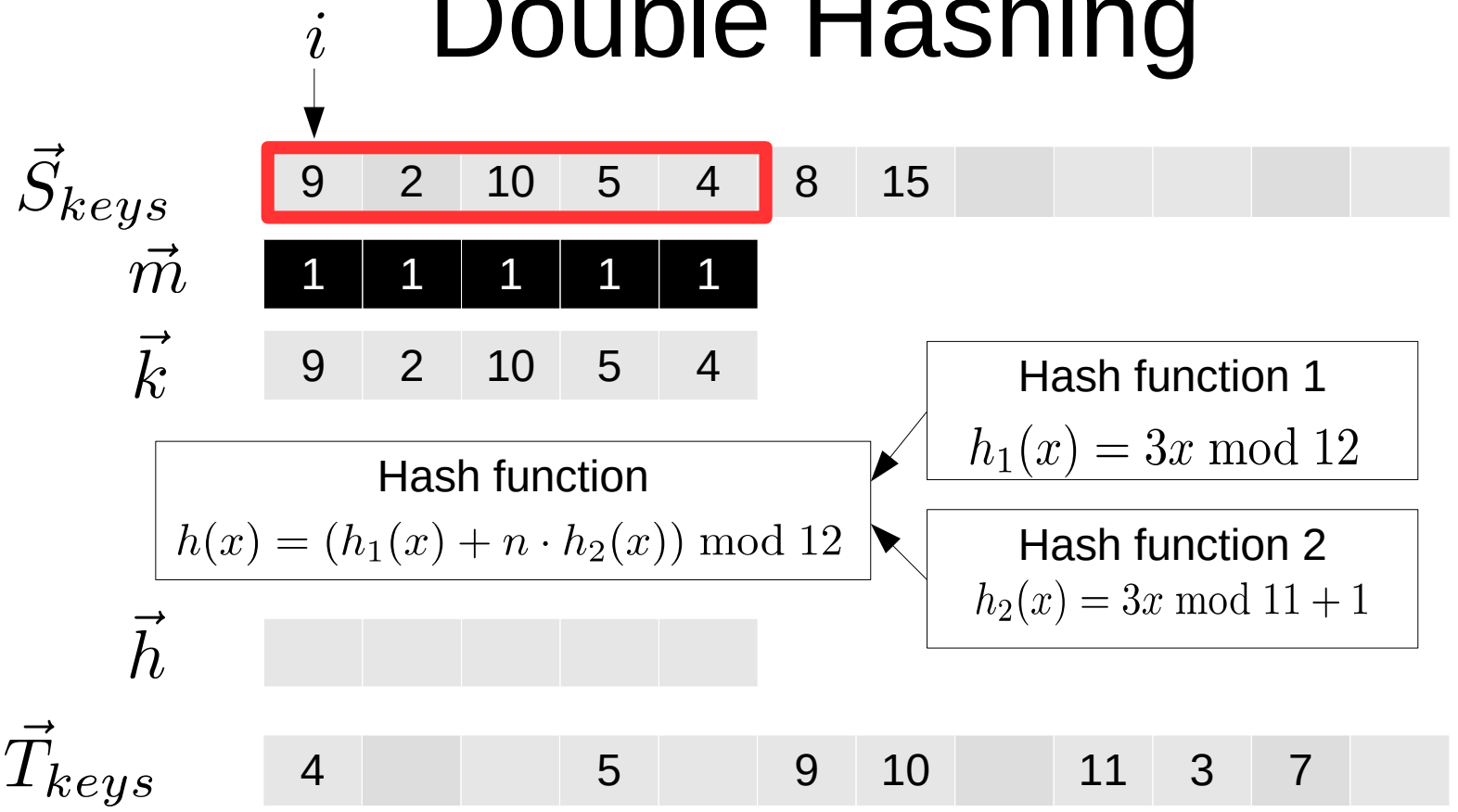
Linear Probing



R.key	R.payload	S.key	S.payload
7	5643	7	7861348
10	864	10	8615348
4	15645234	4	67453678
5	45345	5	453453453
...	...	...	...

Increase n with each collision.

Double Hashing



Duplicate values

- Store them in a separate hash table
  good for many repetitions among matching keys

- Repeat keys
  good for mostly unique keys
  bad if clustered values appear

Implementation

Arithmetic Intrinsics

Intel® Streaming SIMD Extensions 2 (Intel® SSE2) intrinsics for integer arithmetic operations are listed in thi topic. The prototypes for Intel® SSE2 intrinsics are in the `emmintrin.h` header file.

The results of each intrinsic operation are placed in registers. The information about what is placed in each register appears in the tables below, in the detailed explanation of each intrinsic. `R`, `R0`, `R1`, ..., `R15` represent the registers in which results are placed.

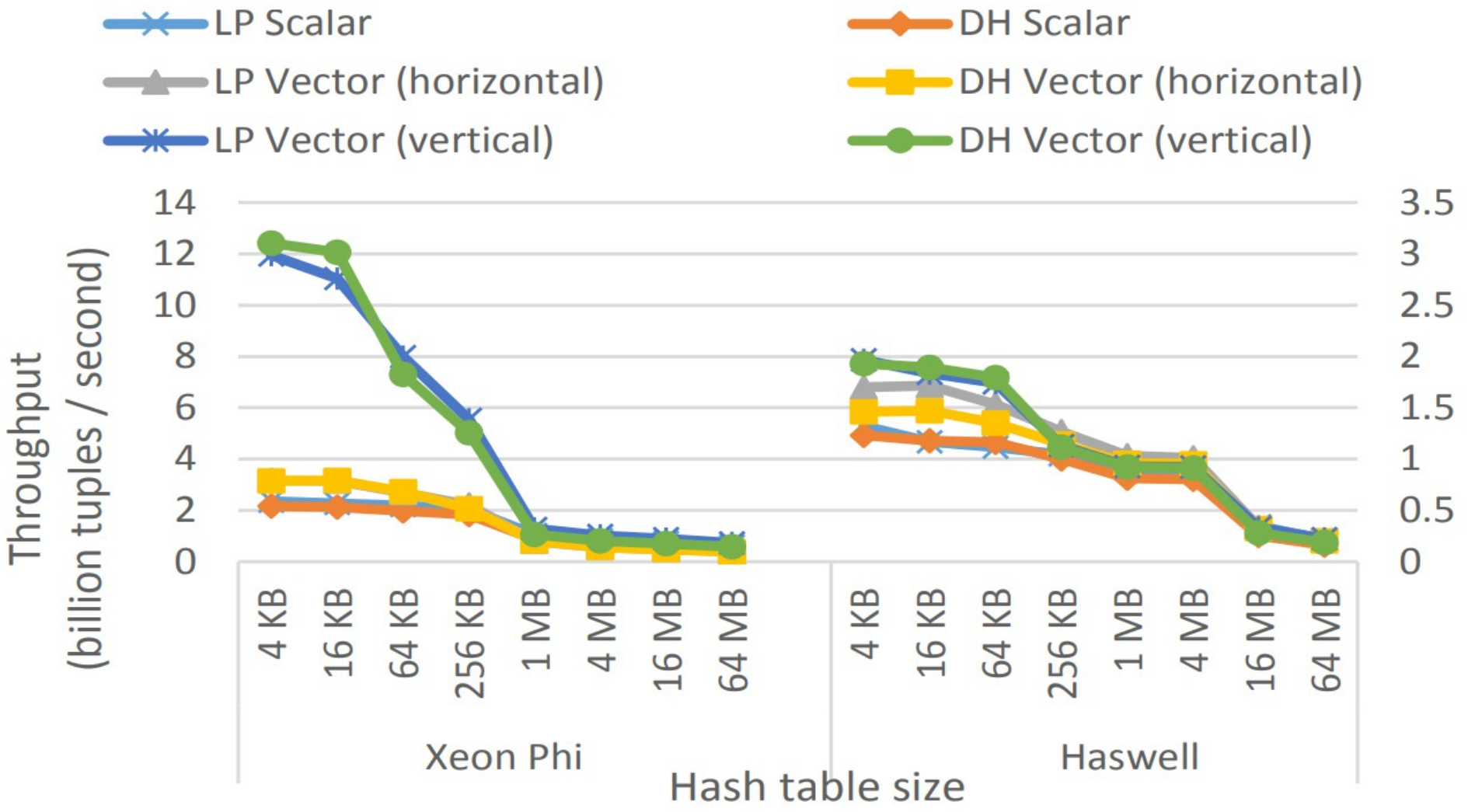
Intrinsic Name	Operation	Corresponding Intel® SSE2 Instruction
`_mm_add_epi8`	Addition	`PADDB`
`_mm_add_epi16`	Addition	`PADDW`
`_mm_add_epi32`	Addition	`PADDD`
`_mm_add_si64`	Addition	`PADDQ`
`_mm_add_epi64`	Addition	`PADDQ`
`_mm_adds_epi8`	Addition	`PADDSB`
`_mm_adds_epi16`	Addition	`PADDSW`

Source: https://software.intel.com/en-us/node/524236

Implementation

- 32 bit integer keys and payloads
- Xeon Phi (MIC):
  - 512-bit vector lanes
  - Advanced instructions for collisions, selective load etc.
- Haswell:
  - Advanced Vector Extensions
  - 256-bit vector lanes
  - Selective load implemented using vector permutations
  - Scatter+gather for collision detection

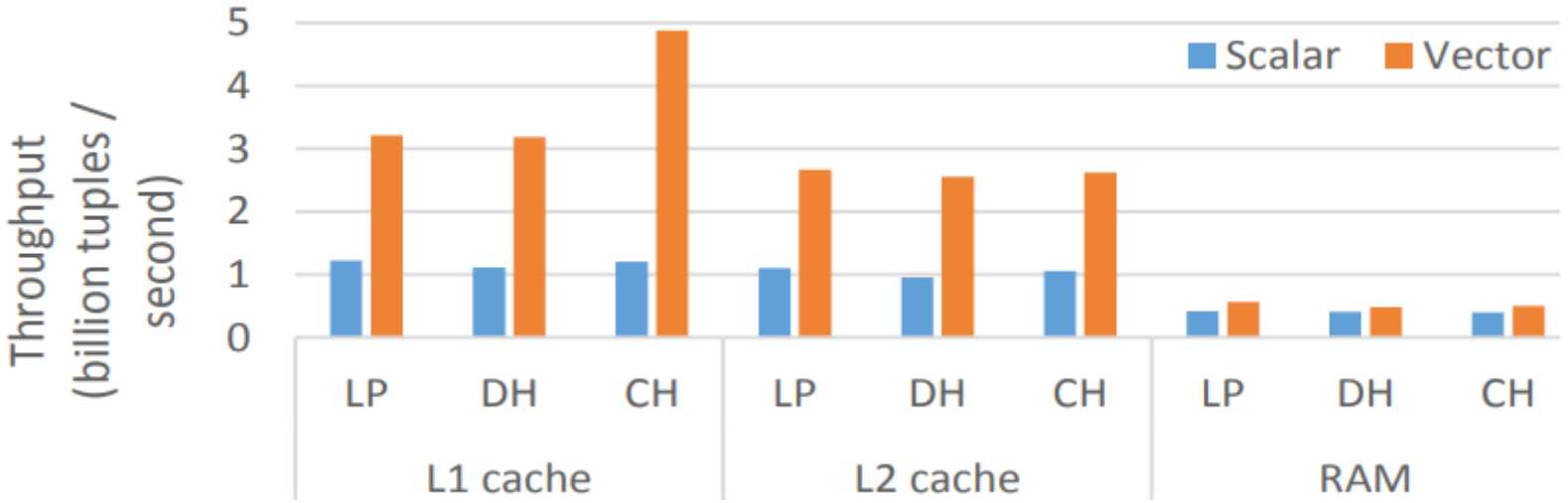
Benchmark - 1:1



Throughput: $\frac{|R| + |S|}{t}$

Source: O. Polychroniou, A. Raghavan, K. Ross: Rethinking SIMD Vectorization for In-Memory Databases. Columbia University, 2015.

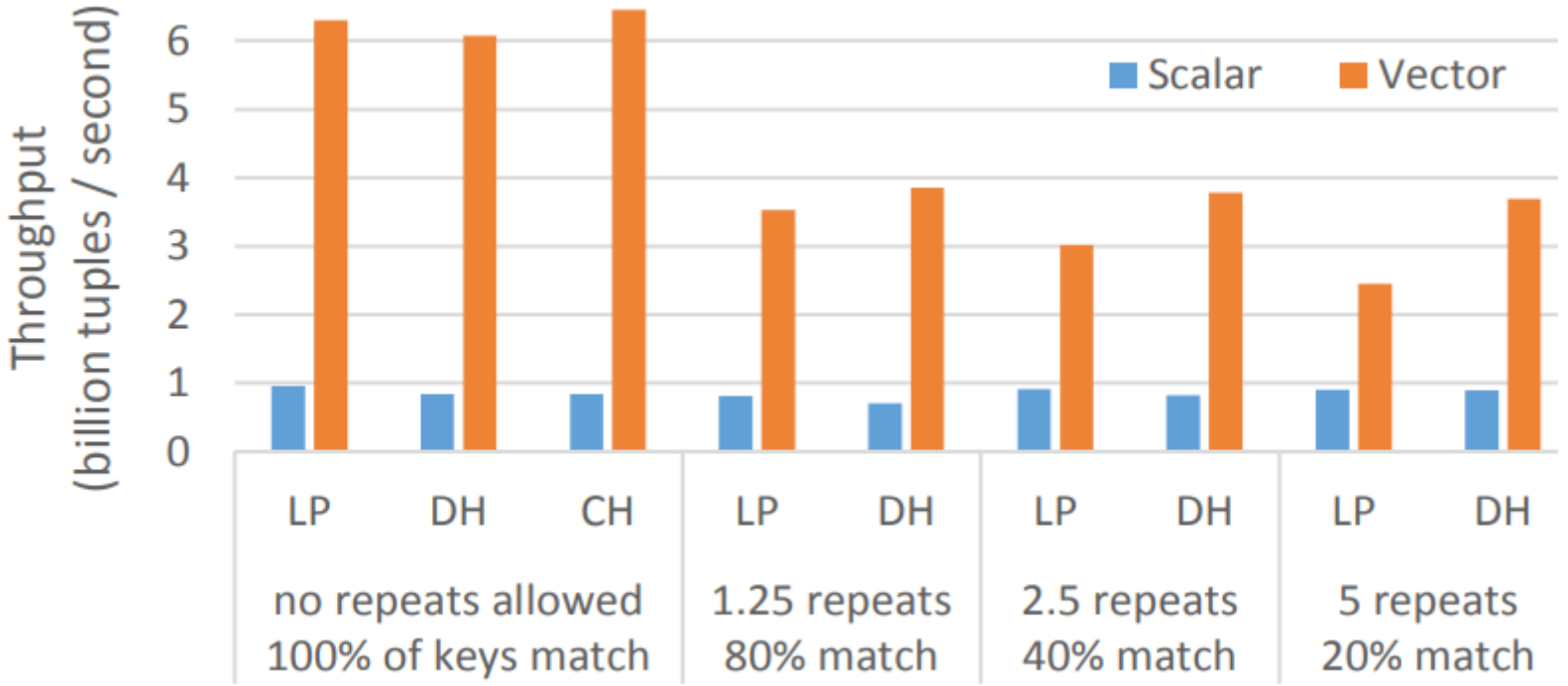
Benchmark – 1:1



Throughput: $\frac{|R| + |S|}{t}$

Source: O. Polychroniou, A. Raghavan, K. Ross: Rethinking SIMD Vectorization for In-Memory Databases. Columbia University, 2015.

Benchmark – 1:10



Throughput: $\frac{|R| + |S|}{t}$

Source: O. Polychroniou, A. Raghavan, K. Ross: Rethinking SIMD Vectorization for In-Memory Databases. Columbia University, 2015.

Sources

- O. Polychroniou, A. Raghavan, K. Ross: Rethinking SIMD Vectorization for In-Memory Databases. Columbia University, 2015.
- Intel Corporation: Compare Intrinsics, https://software.intel.com/en-us/node/524239, last access: October 2018.