Dynamic Filters (Quotient & InfiniFilter)

Research Topics in Database Management

Niv Dayan

Two volunteers needed for next week's presentations

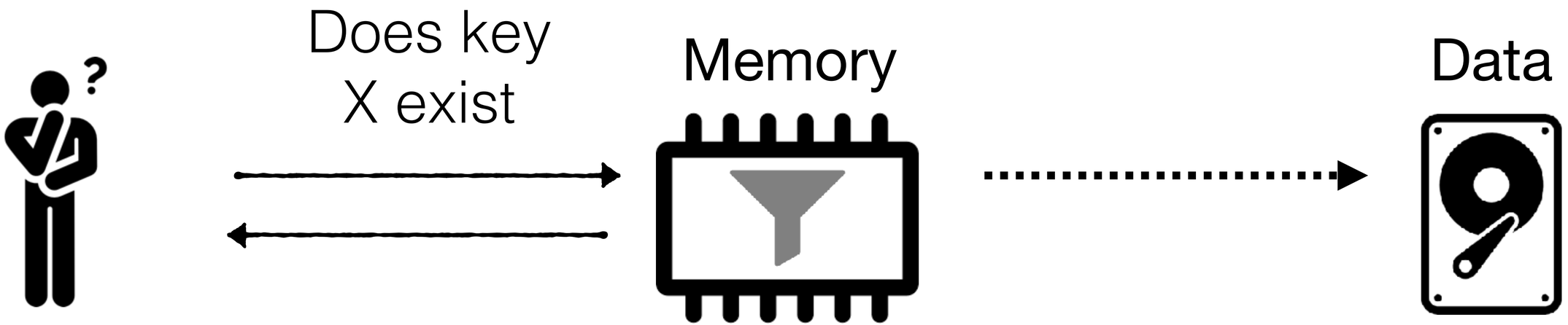
What is a Filter?

Does X exist? → 

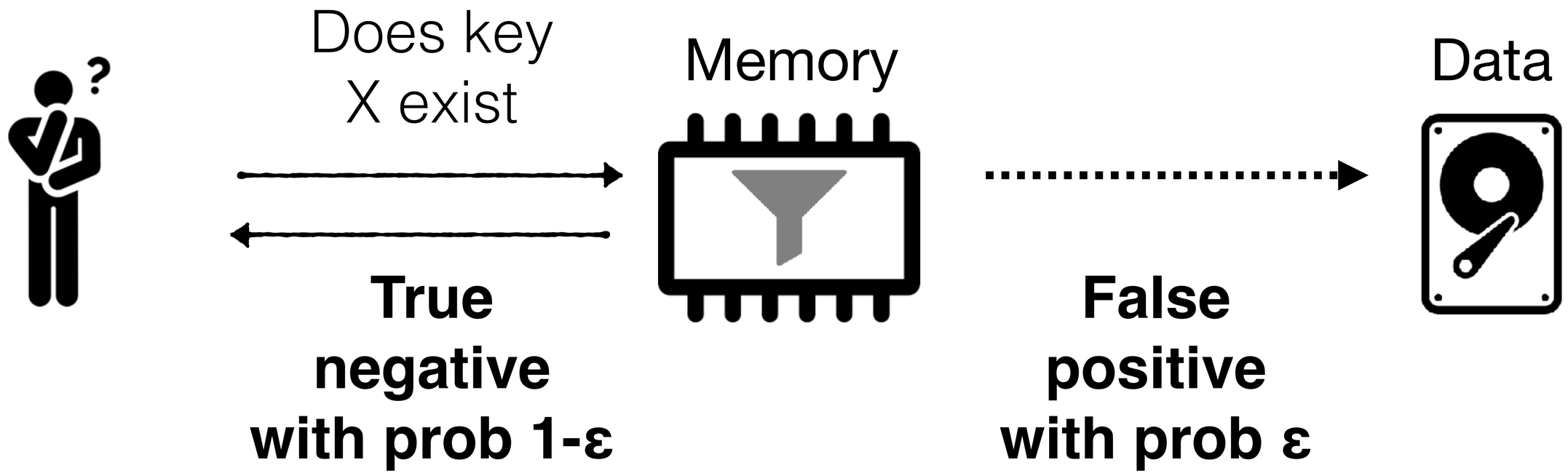
Set

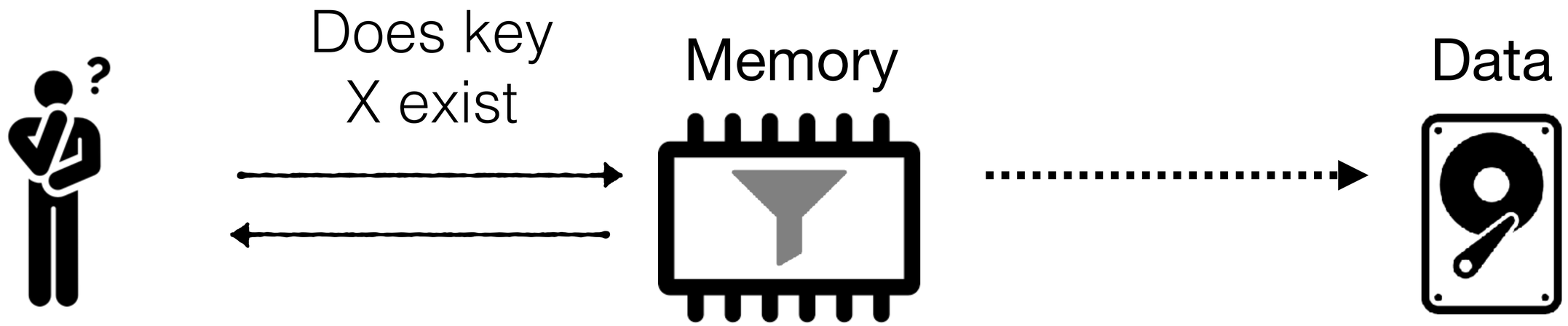
X Y Z

If key X does not exist



If key X does not exist

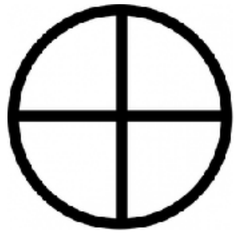




Saves storage accesses & network hops

Blocked
Bloom

XOR

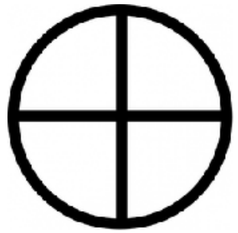


Faster

Lower FPR

Blocked
Bloom

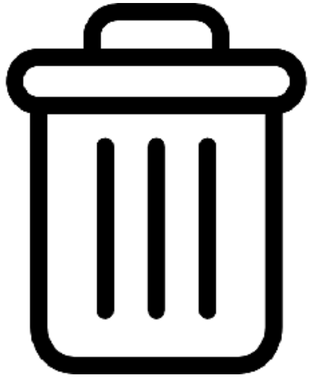
XOR



Static - no deletes or resizing

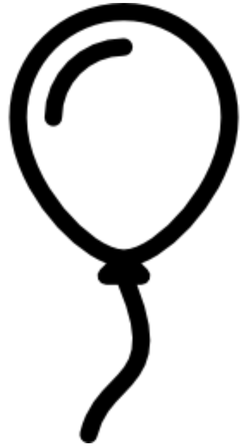
Supporting Dynamic Data

Supporting Dynamic Data



Deletes

(First hour)

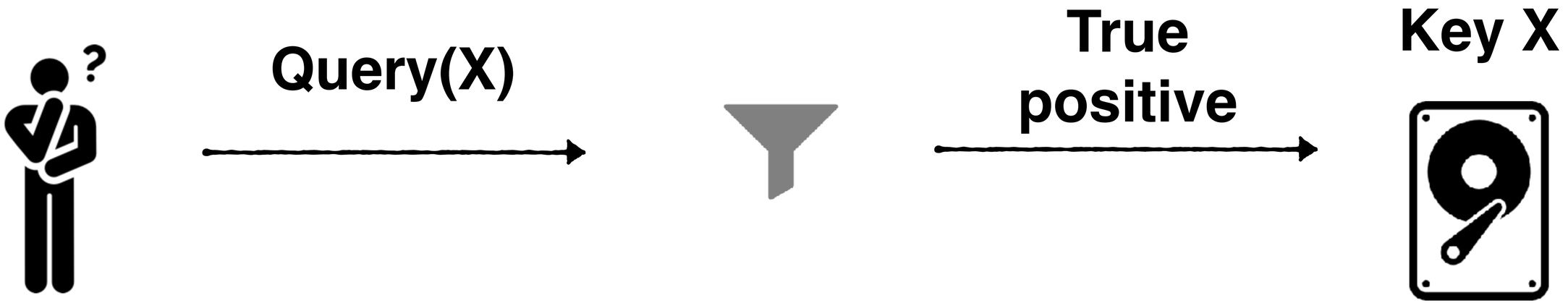


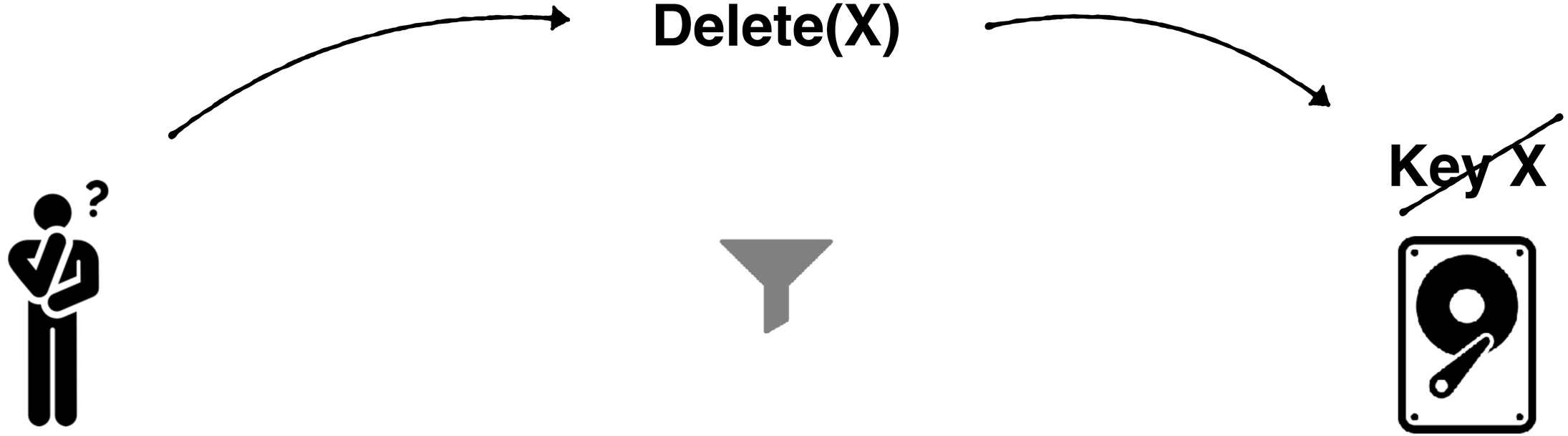
Resizing

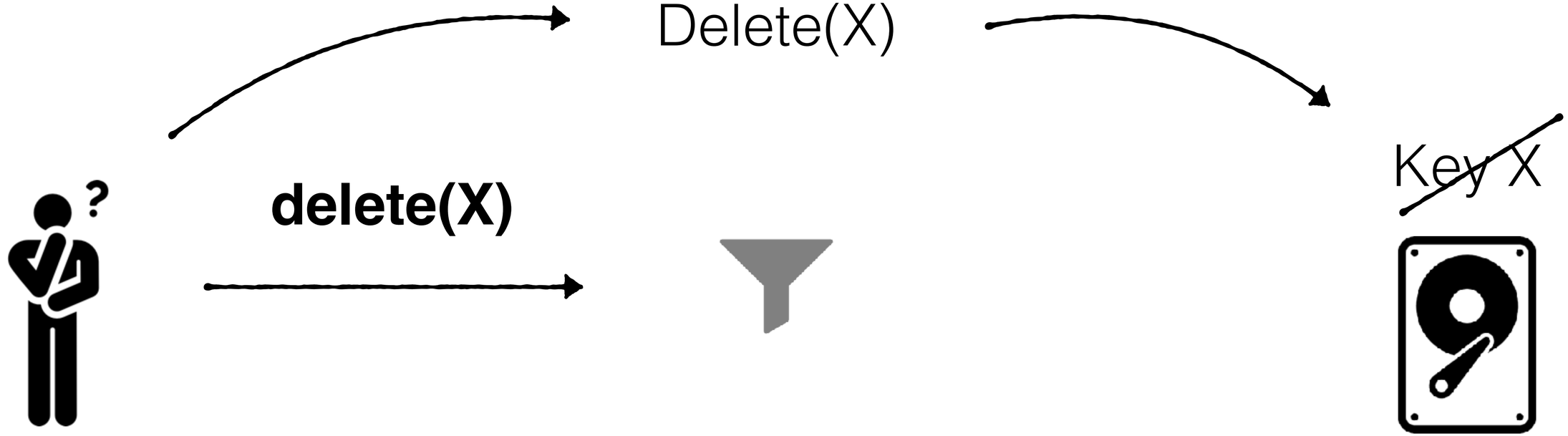
(Second hour)

Why Support Deletes?

Why Support Deletes?




Delete(X)
?
Key X



Why should we also delete from filter?

Desired Outcome

Query(X)

Negative

~~Key X~~

Desired Outcome

Query(X)

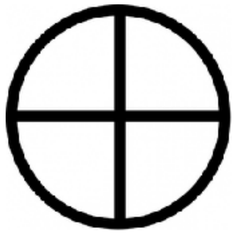
Negative

Only possible if filter supports deletes

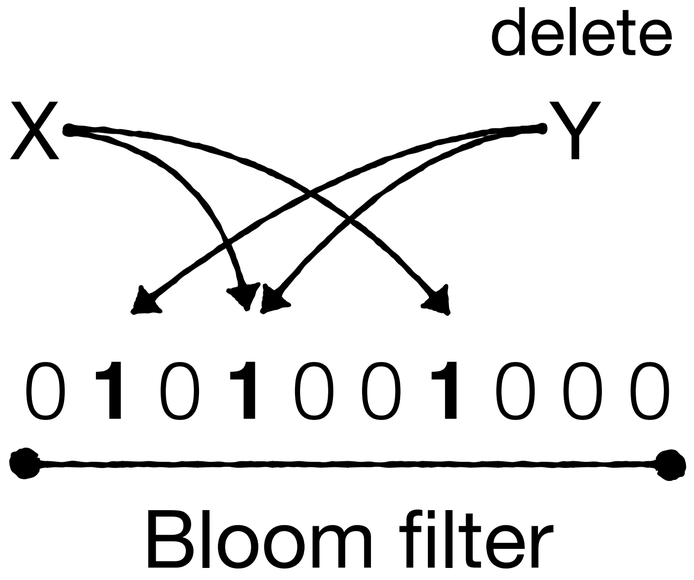
Why do last week's filters not support deletes?

Bloom

XOR

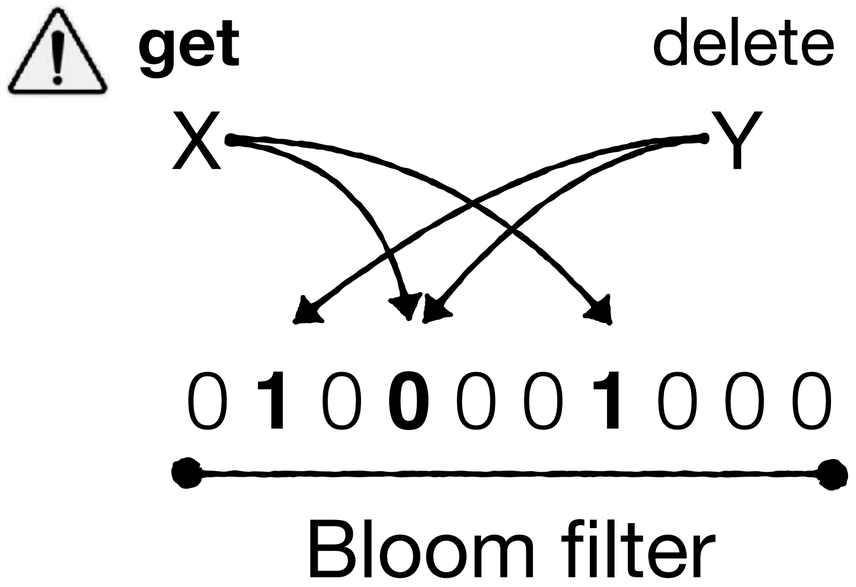


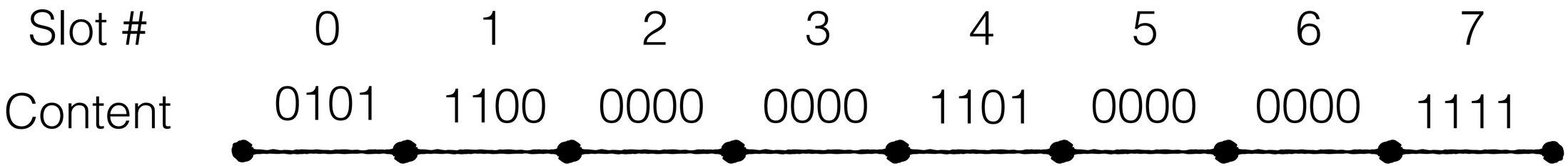
Multiple keys may map to each bit



Multiple keys may map to each bit

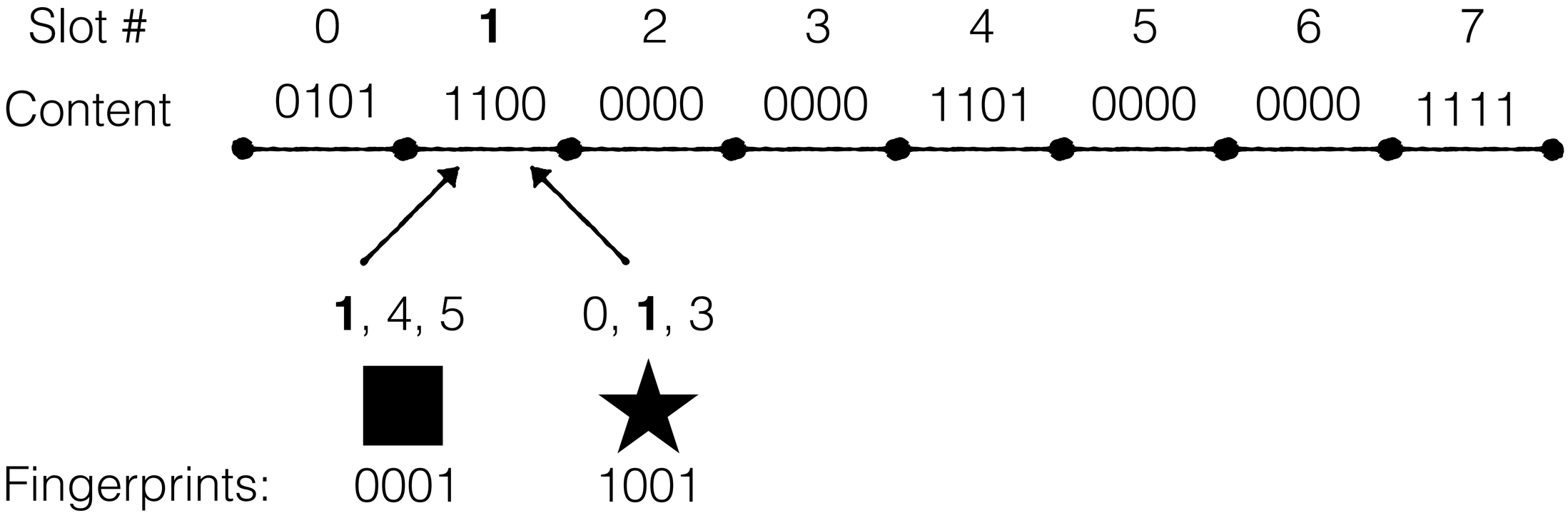
Setting bits back to 0s can lead to false negatives



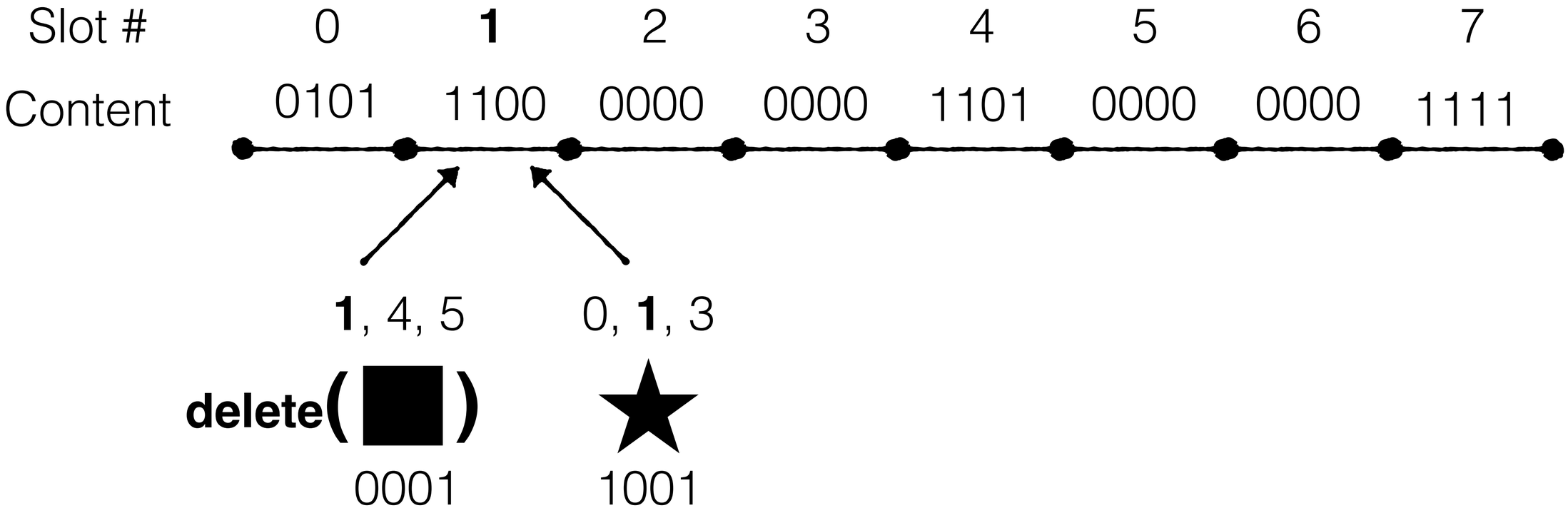

Slot #
0
1
2
3
4
5
6
7
Content
0101
1100
0000
0000
1101
0000
0000
1111

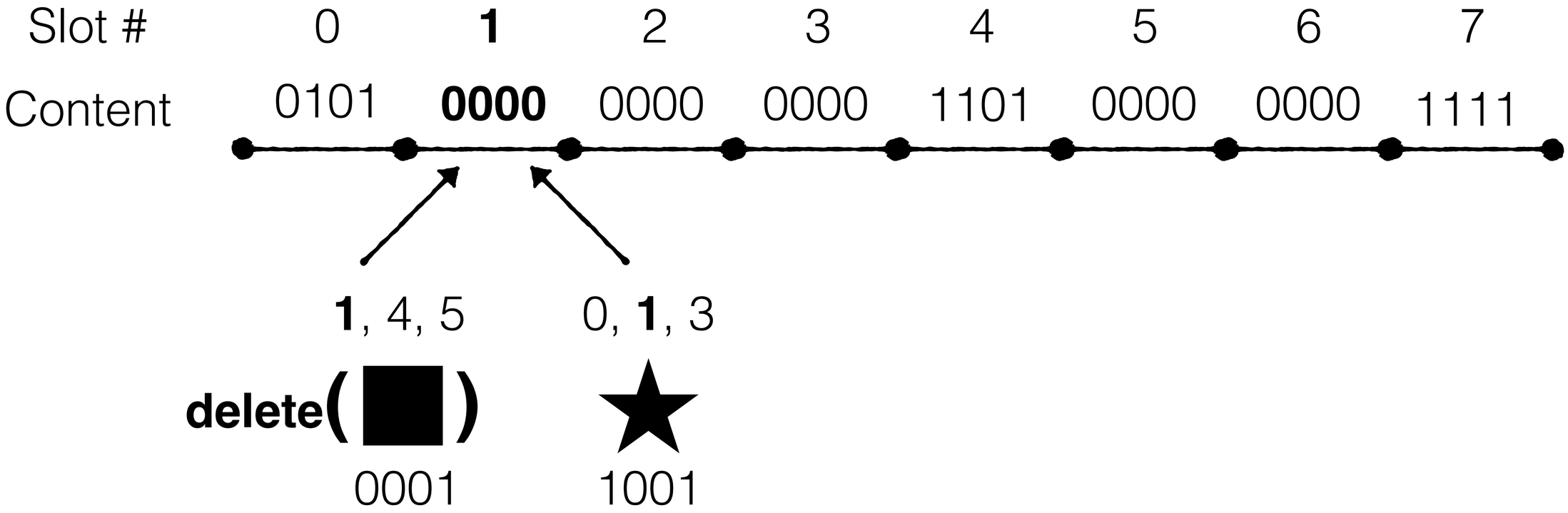
XOR filter

Multiple keys share slots

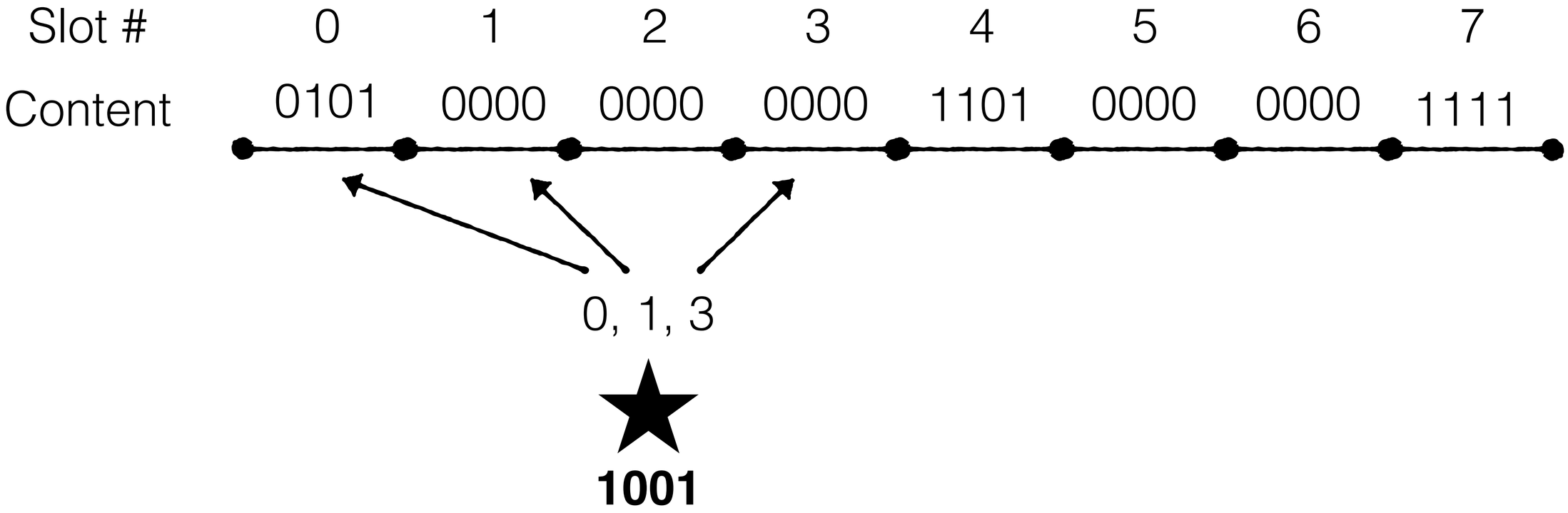


Multiple keys share slots



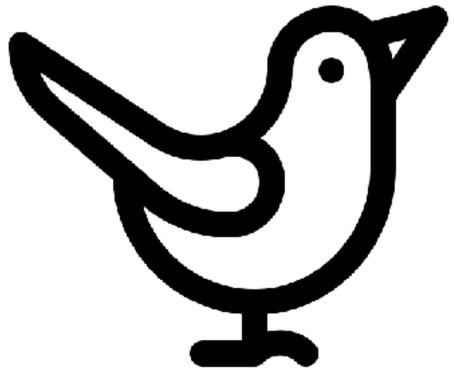

Slot #
0
1
2
3
4
5
6
7
Content
0101
0000
0000
0000
1101
0000
0000
1111
1, 4, 5
0, 1, 3
delete( )
0001
1001

Resetting a slot for one entry will cause false negatives over other entries



How to support deletes without false negatives?

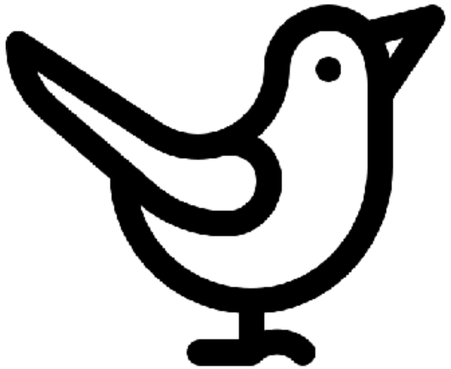
How to support deletes without false negatives?



Cuckoo Filters

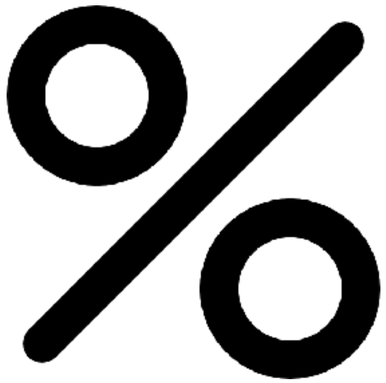
(Last semester)

How to support deletes without false negatives?



Cuckoo Filters

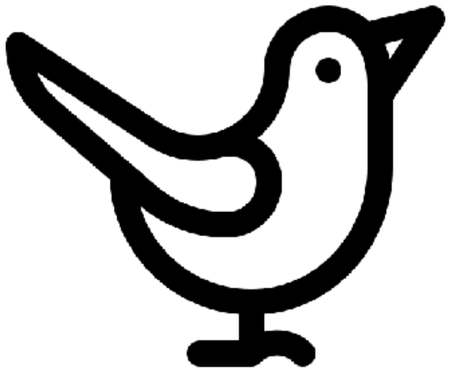
(Last semester)



Quotient Filters

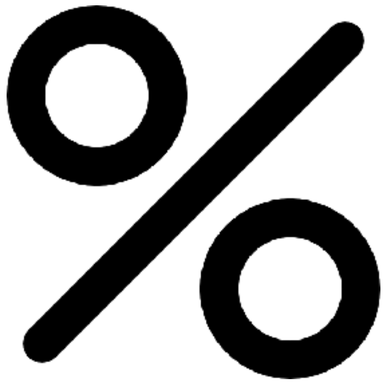
(Today)

Why cover another filter that supports deletes?



Cuckoo Filters

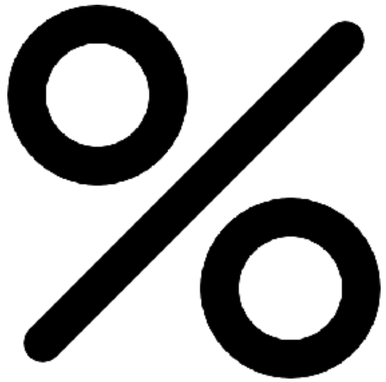
(Last semester)



Quotient Filters

(Today)

Showcase cool encoding/decoding techniques



Quotient Filters

Quotient Filters

Don’t Thrash: How to Cache Your Hash on Flash. VLDB 2012.

Michael A Bender, Martin Farach-Colton, Rob Johnson, Bradley C Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P Spillane, Erez Zadok.

A General-Purpose Counting Filter: Making Every Bit Count. SIGMOD 2017.

Prashant Pandey, Michael A Bender, Rob Johnson, Rob Patro.

Vector Quotient Filters: Overcoming the Time/Space Trade-Off in Filter Design. SIGMOD 2021.

Prashant Pandey, Alex Conway, Joe Durie, Michael A. Bender, Martin Farach-Colton, Rob Johnson.

Which to focus on?

Don’t Thrash: How to Cache Your Hash on Flash. VLDB 2012.

A General-Purpose Counting Filter: Making Every Bit Count. SIGMOD 2017.

Vector Quotient Filters: Overcoming the Time/Space Trade-Off in Filter Design. SIGMOD 2021.

Don’t Thrash: How to Cache Your Hash on Flash. VLDB 2012.
Worse memory & query efficiency

A General-Purpose Counting Filter: Making Every Bit Count. SIGMOD 2017.

Vector Quotient Filters: Overcoming the Time/Space Trade-Off in Filter Design. SIGMOD 2021.

Don’t Thrash: How to Cache Your Hash on Flash. VLDB 2012.

Worse memory & query efficiency

A General-Purpose Counting Filter: Making Every Bit Count. SIGMOD 2017.

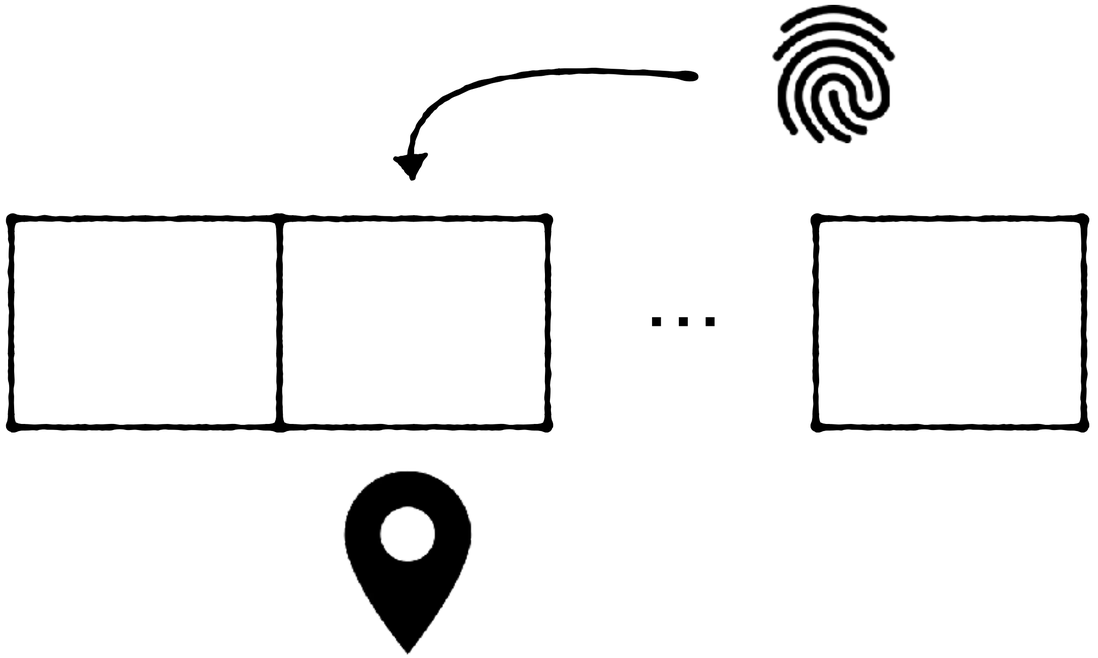
Vector Quotient Filters: Overcoming the Time/Space Trade-Off in Filter Design. SIGMOD 2021.

Uses SIMD - less tunable

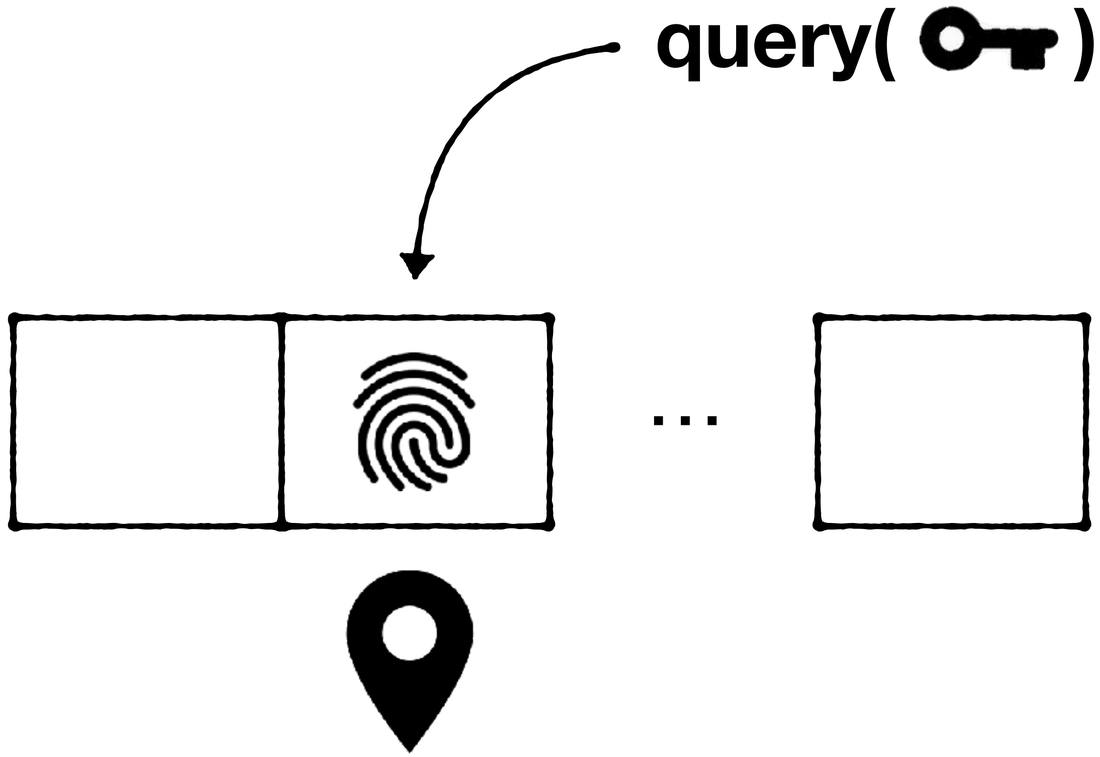
Our focus

A General-Purpose Counting Filter: Making Every Bit Count. 2017.

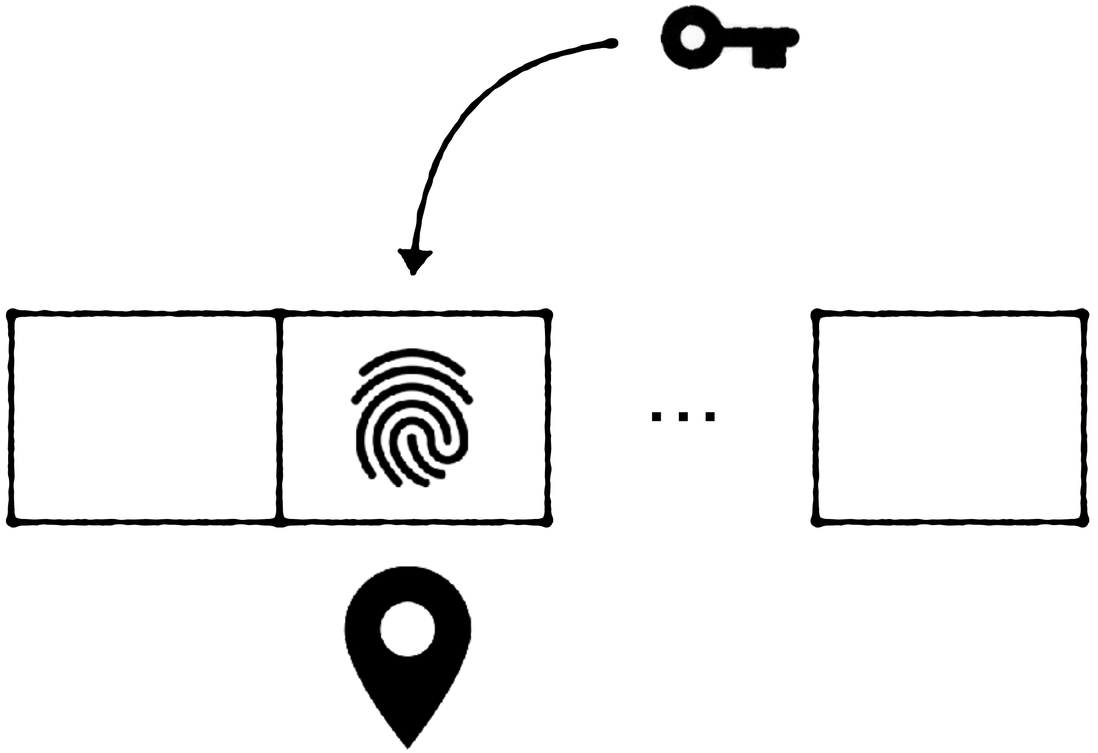

hash( )
0 1 0 1 0 0 1 1 0 1 0 1 1 0 1 1 0 0



Canonical slot

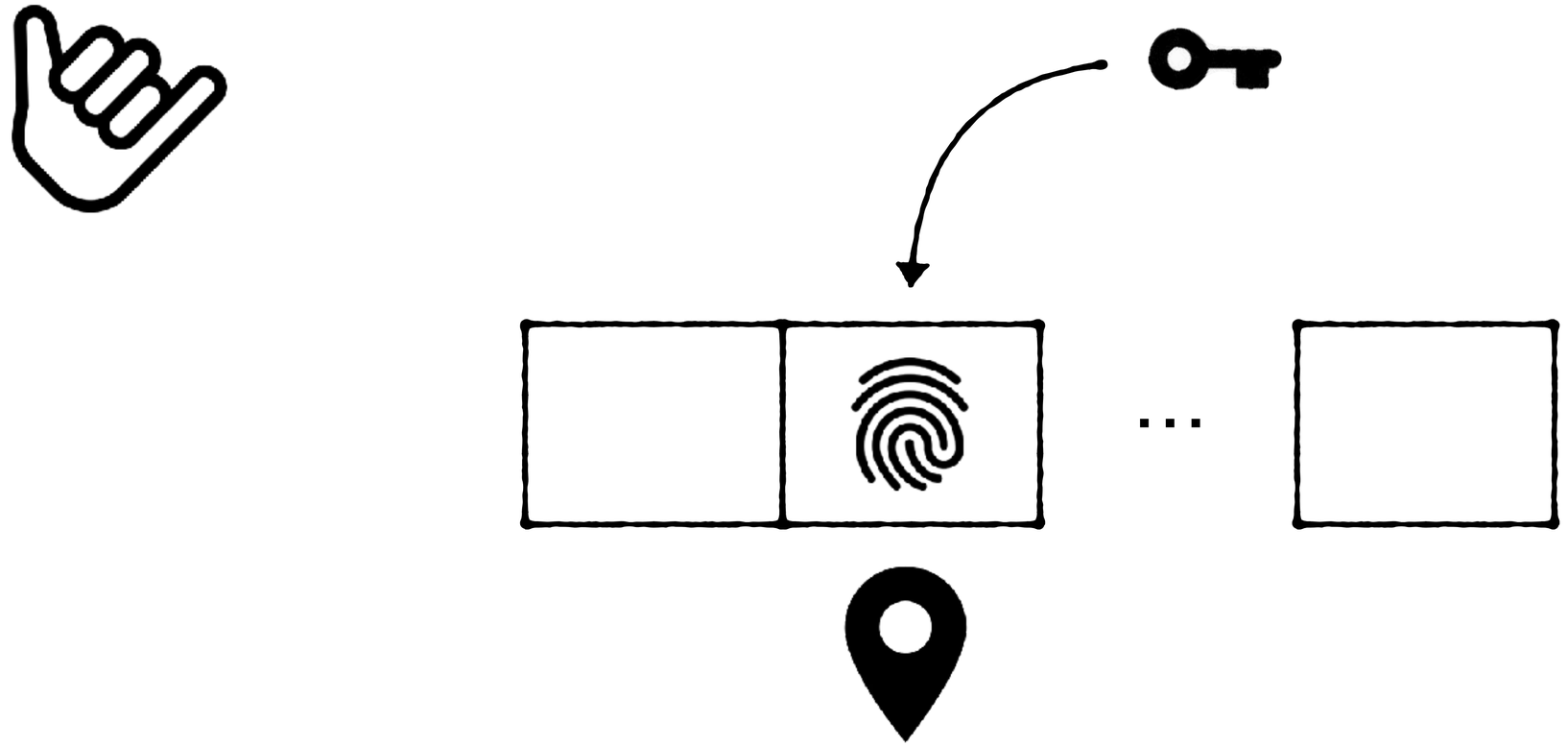

query( )
...
Canonical slot

Each inserted key corresponds to exactly one entry

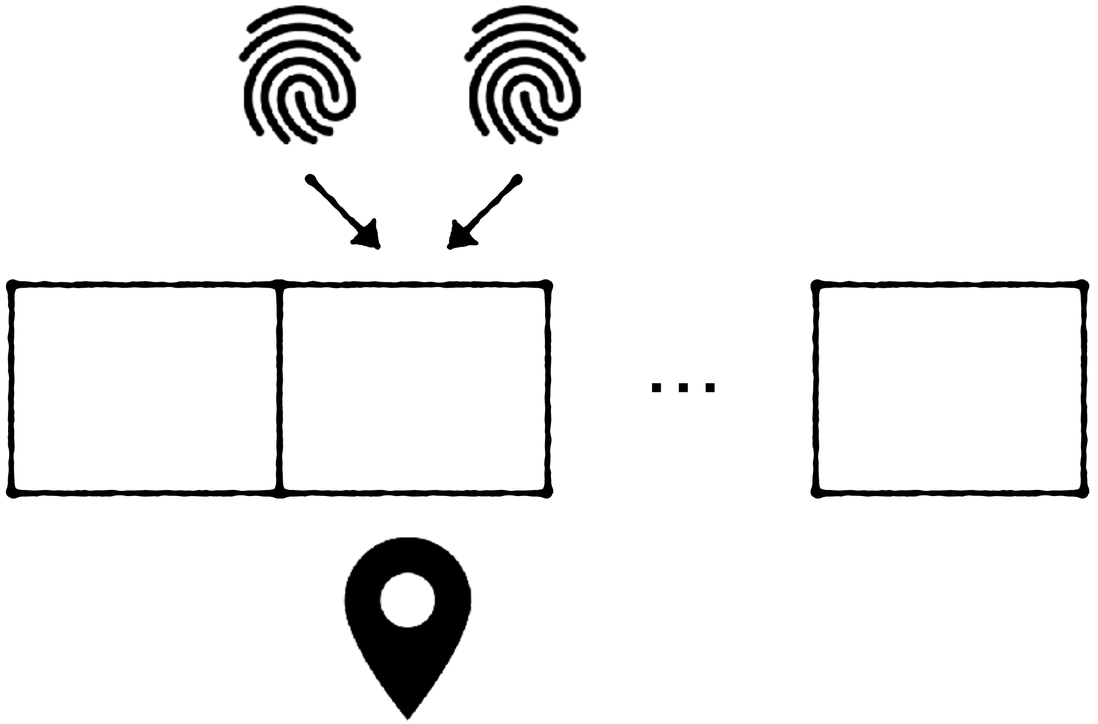


Each inserted key corresponds to exactly one entry

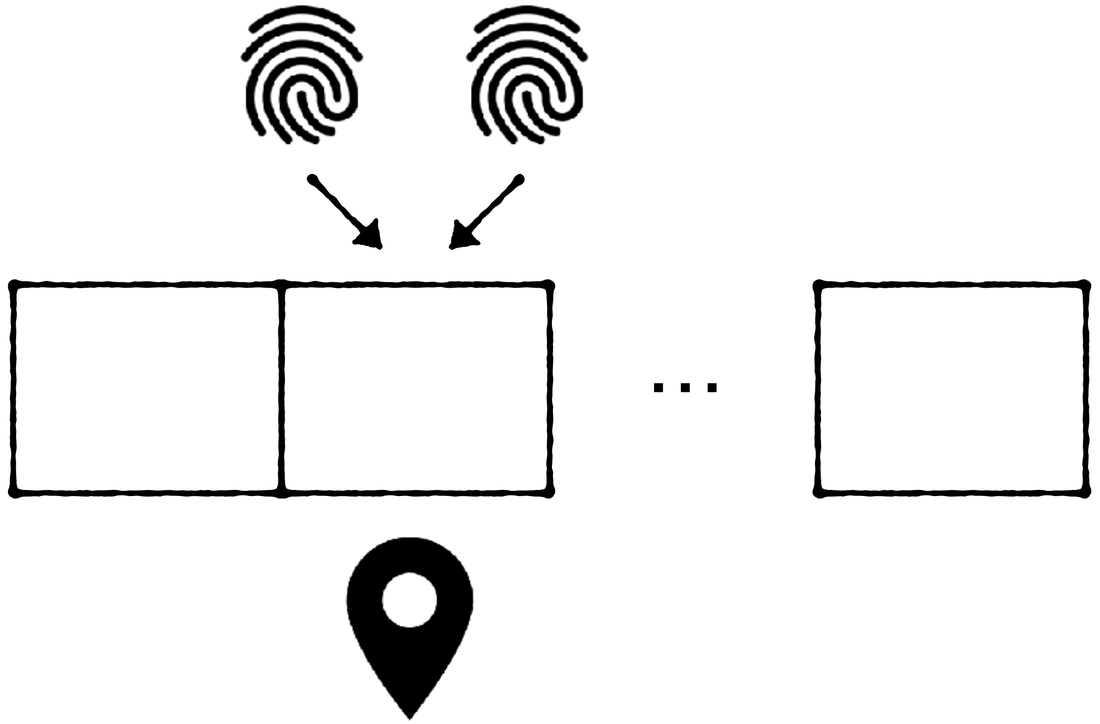
Removing it won’t introduce false negatives for other entries



Hash collisions - multiple fingerprints map to same canonical slot

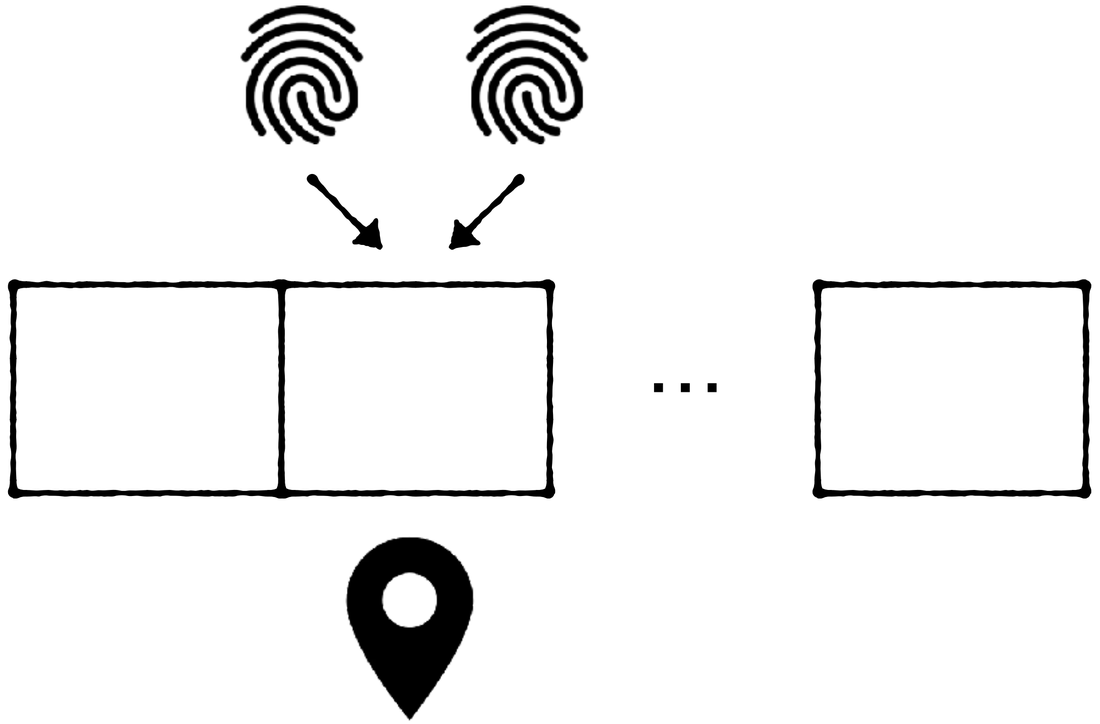


Address using Robin Hood Hashing

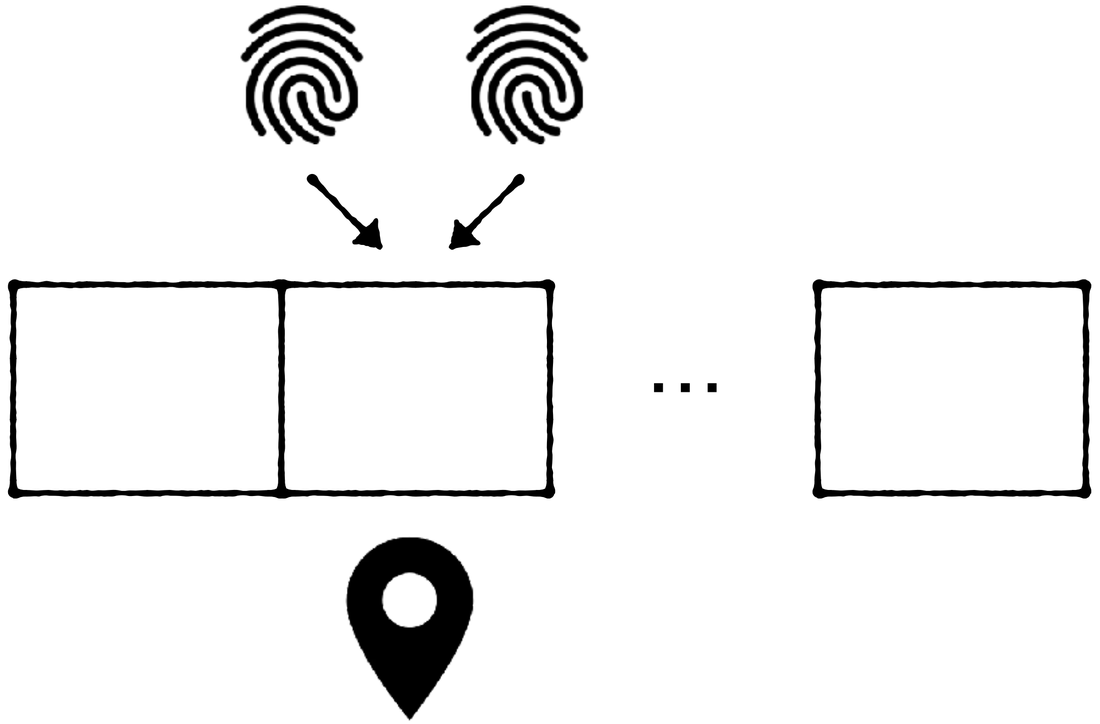


Address using Robin Hood Hashing

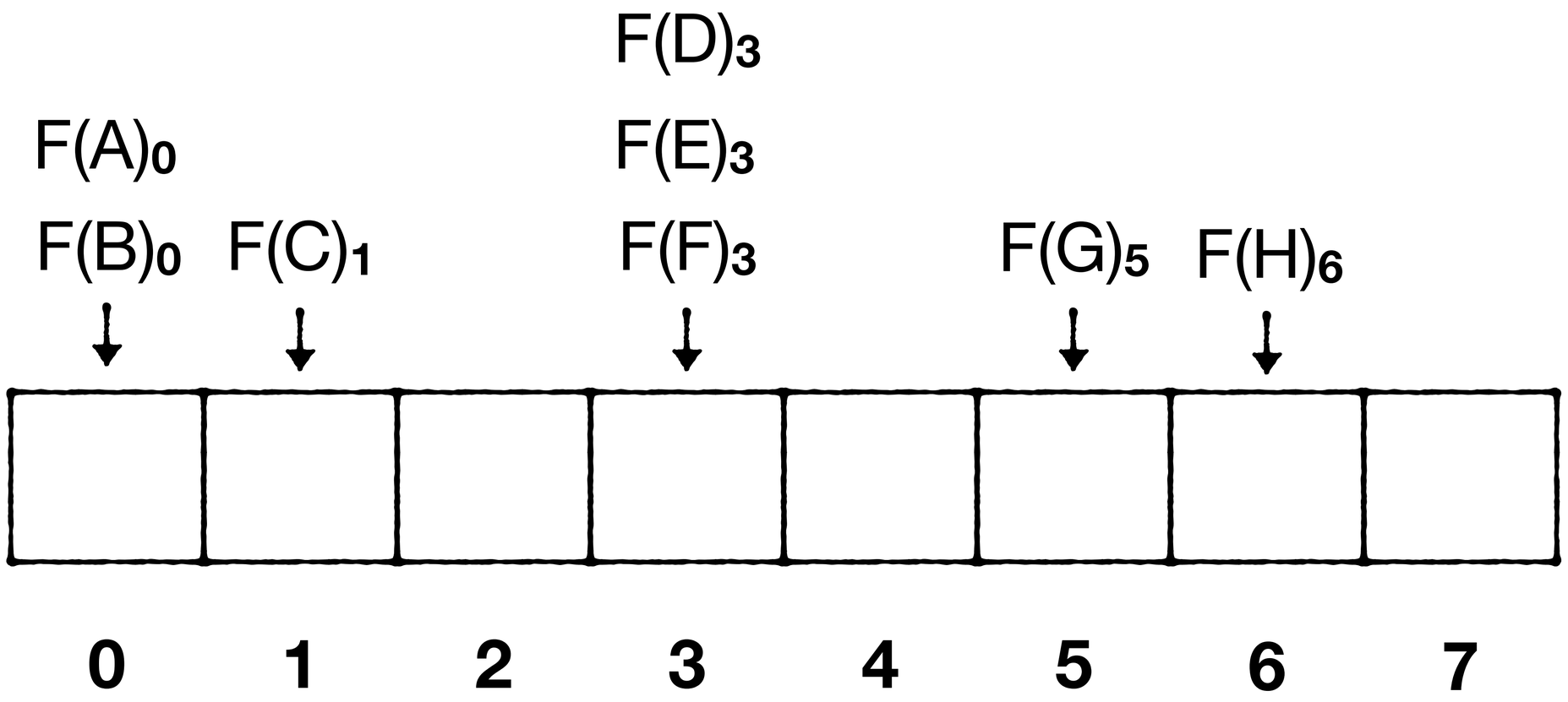
Variant of linear probing

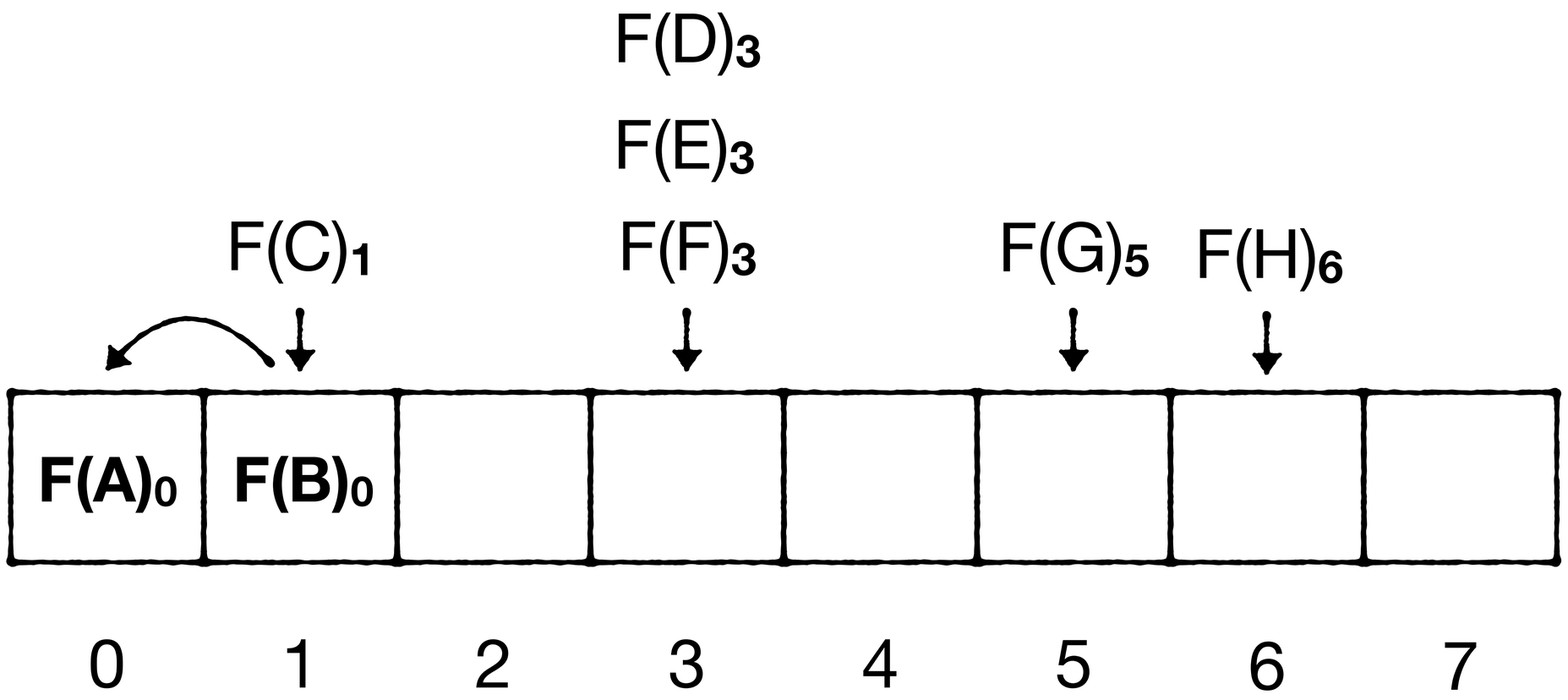


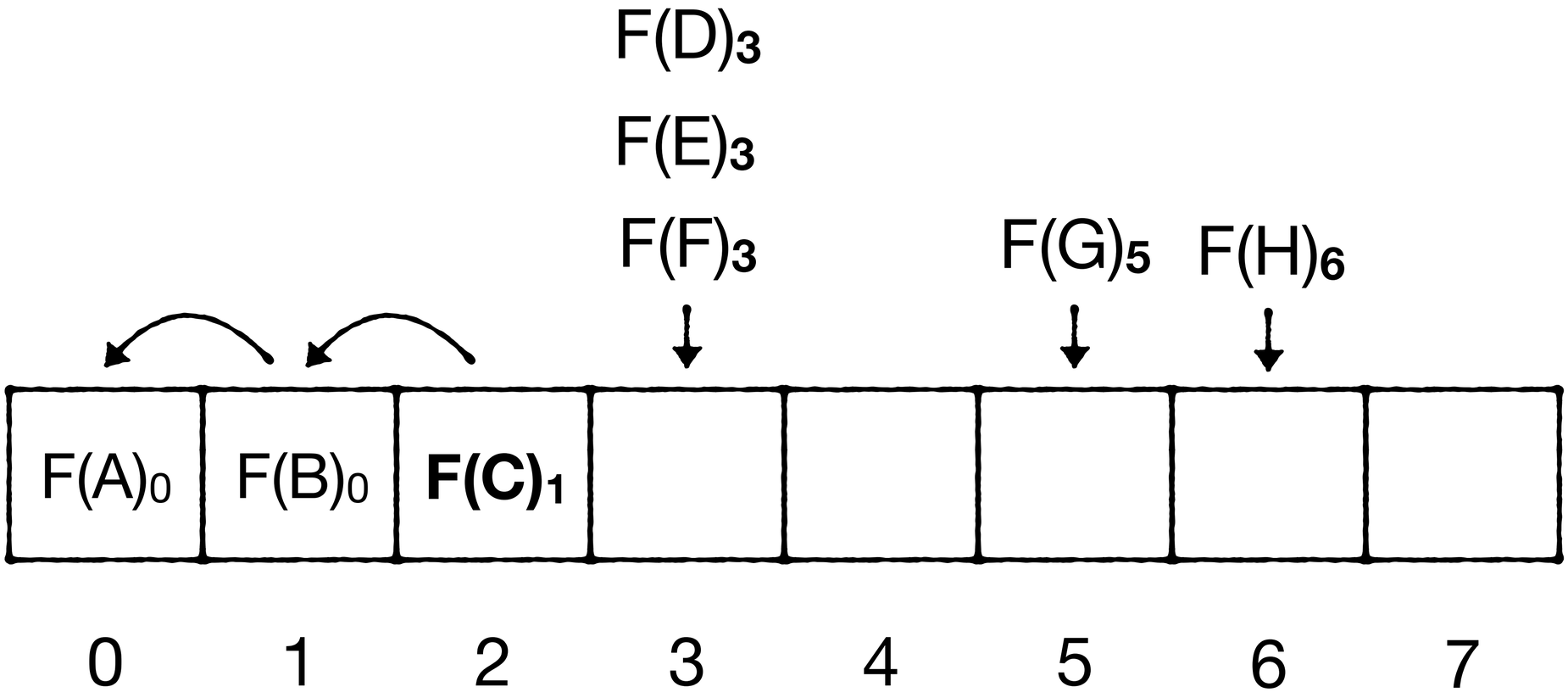
Each fingerprint is pushed rightwards yet stays as close as possible to its “canonical slot”

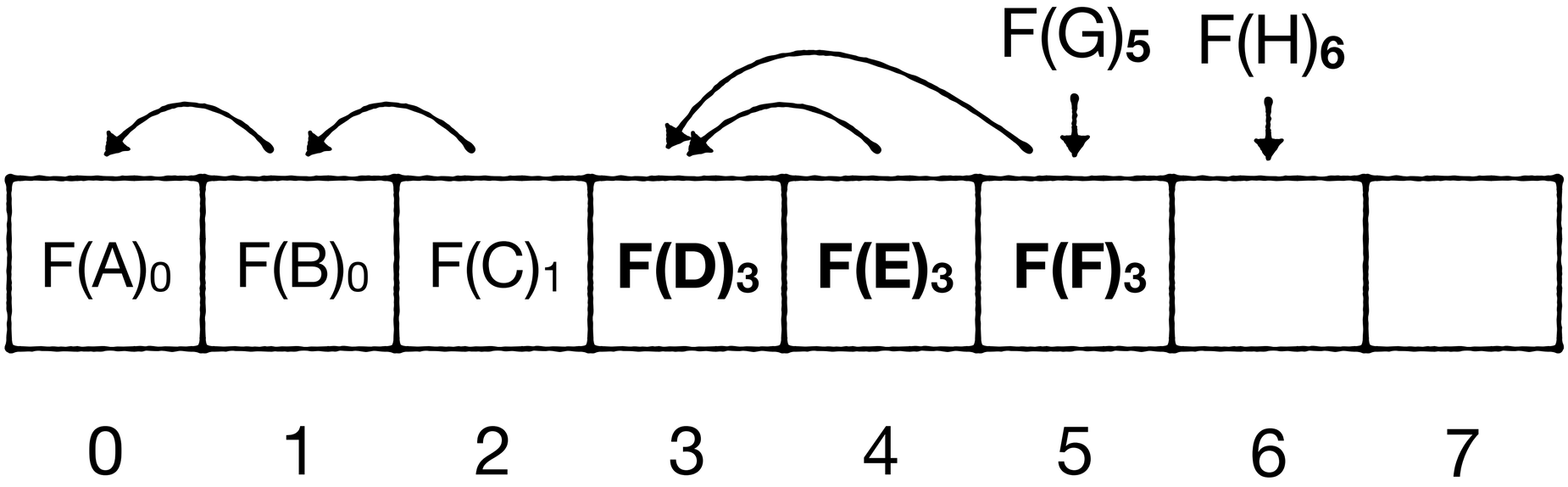


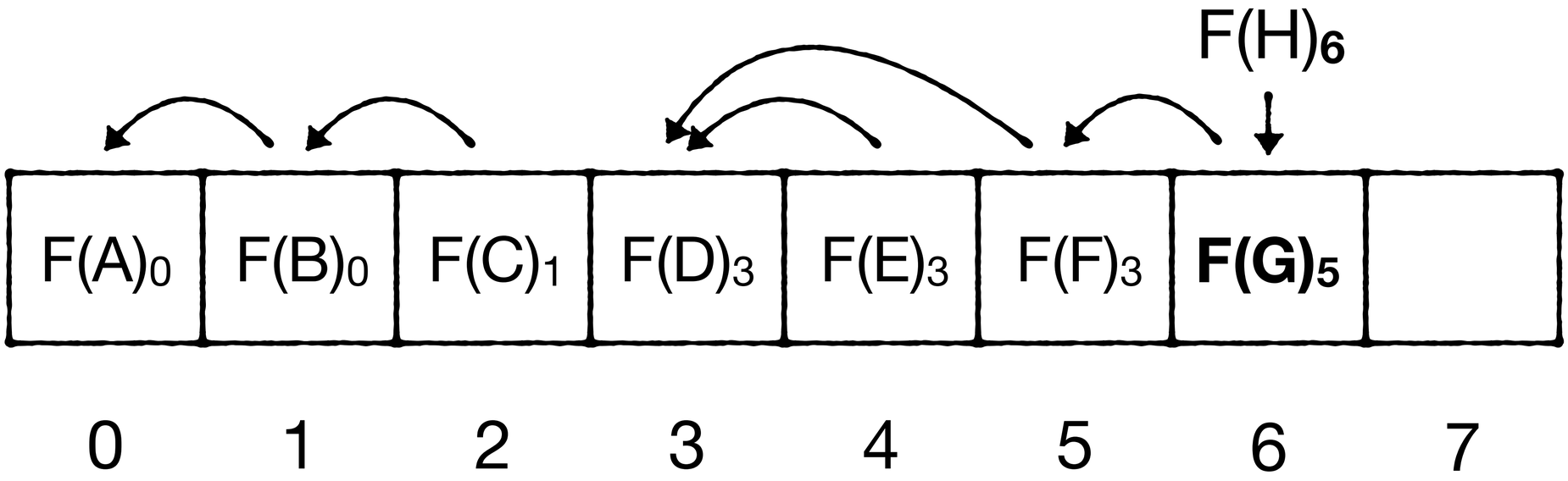
Each fingerprint is pushed rightwards yet stays as close as possible to its “canonical slot”

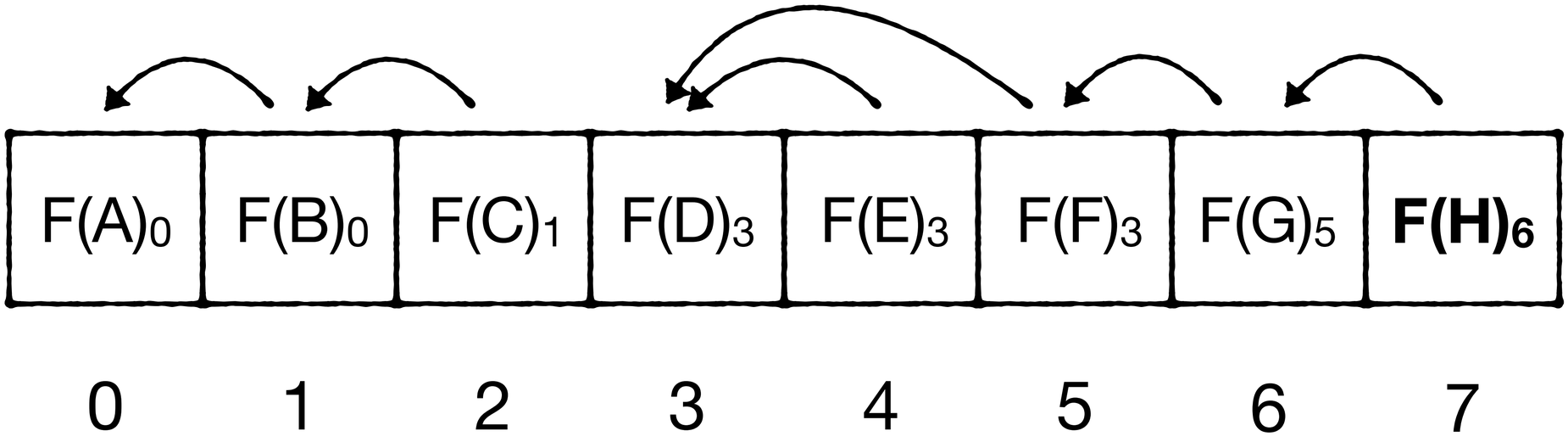



F(D)3
F(E)3
F(C)1
F(F)3
F(G)5
F(H)6
F(A)0
F(B)0
0
1
2
3
4
5
6
7

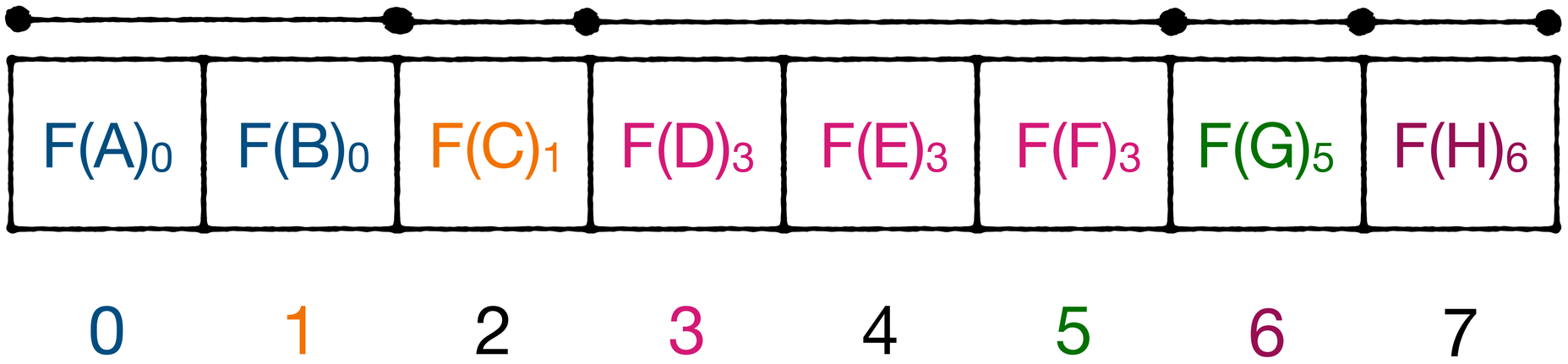

F(D)3
F(E)3
F(F)3
F(G)5
F(H)6
F(A)0
F(B)0
F(C)1
0
1
2
3
4
5
6
7


F(G)5
F(H)6
F(A)0
F(B)0
F(C)1
F(D)3
F(E)3
F(F)3
0
1
2
3
4
5
6
7

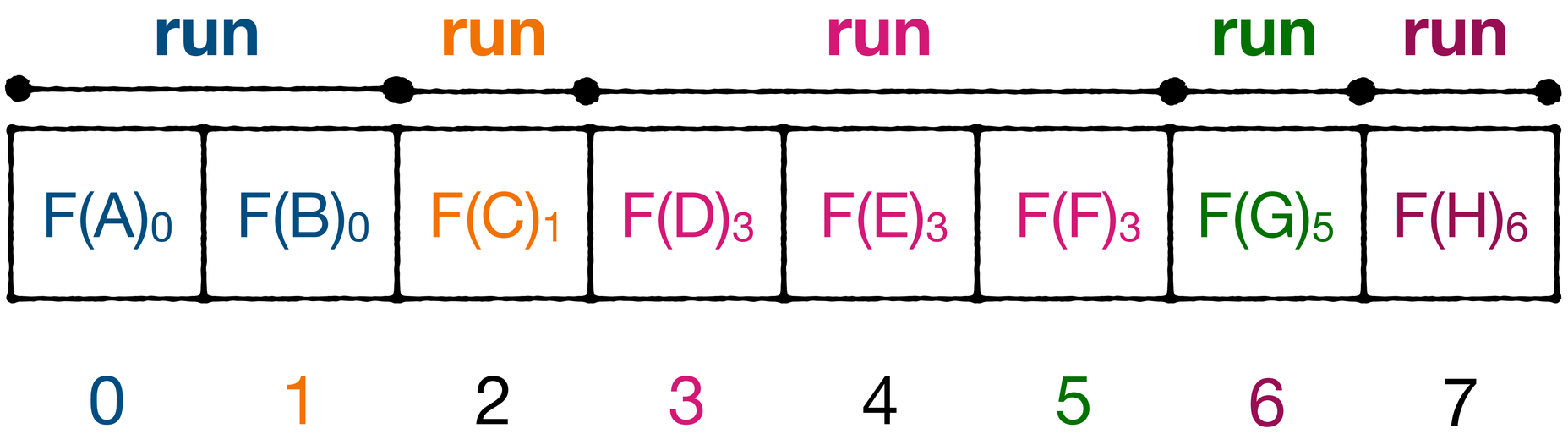

F(H)6
F(A)0
F(B)0
F(C)1
F(D)3
F(E)3
F(F)3
F(G)5
0
1
2
3
4
5
6
7


F(A)0
F(B)0
F(C)1
F(D)3
F(E)3
F(F)3
F(G)5
F(H)6
0
1
2
3
4
5
6
7

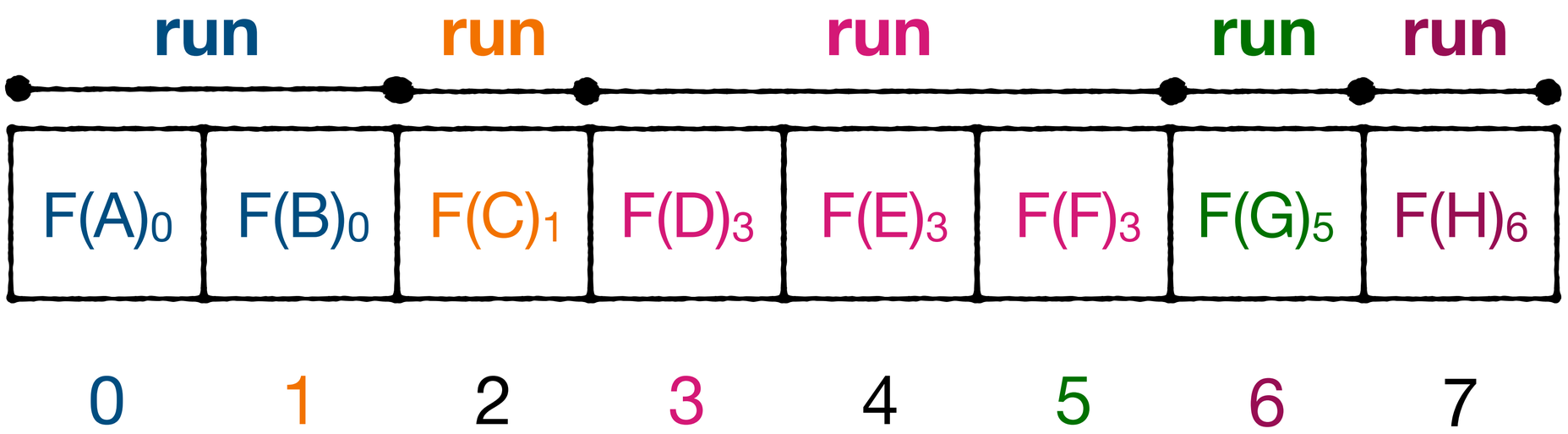
Note: fingerprints belonging to same canonical slot are contiguous

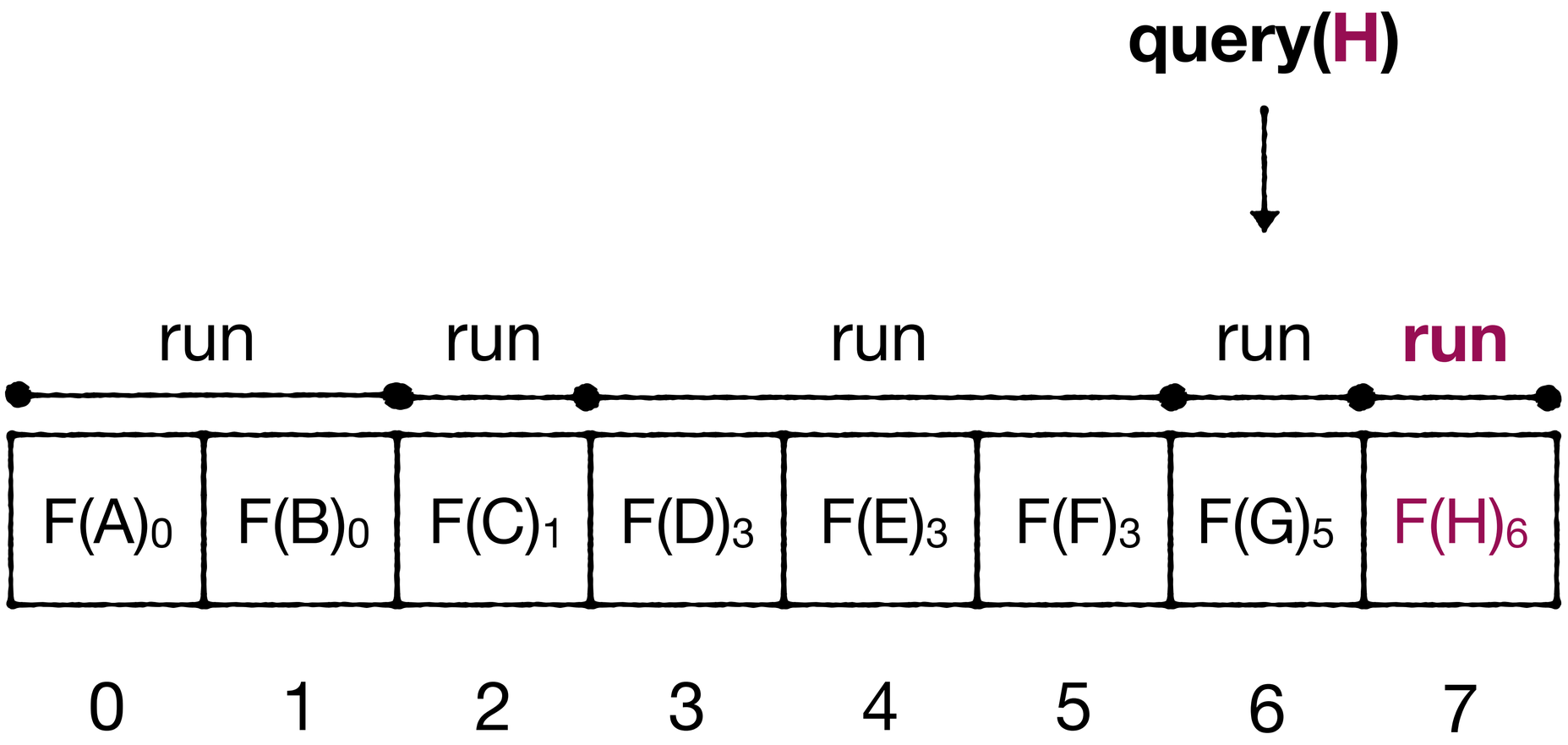


Note: fingerprints belonging to same canonical slot are contiguous

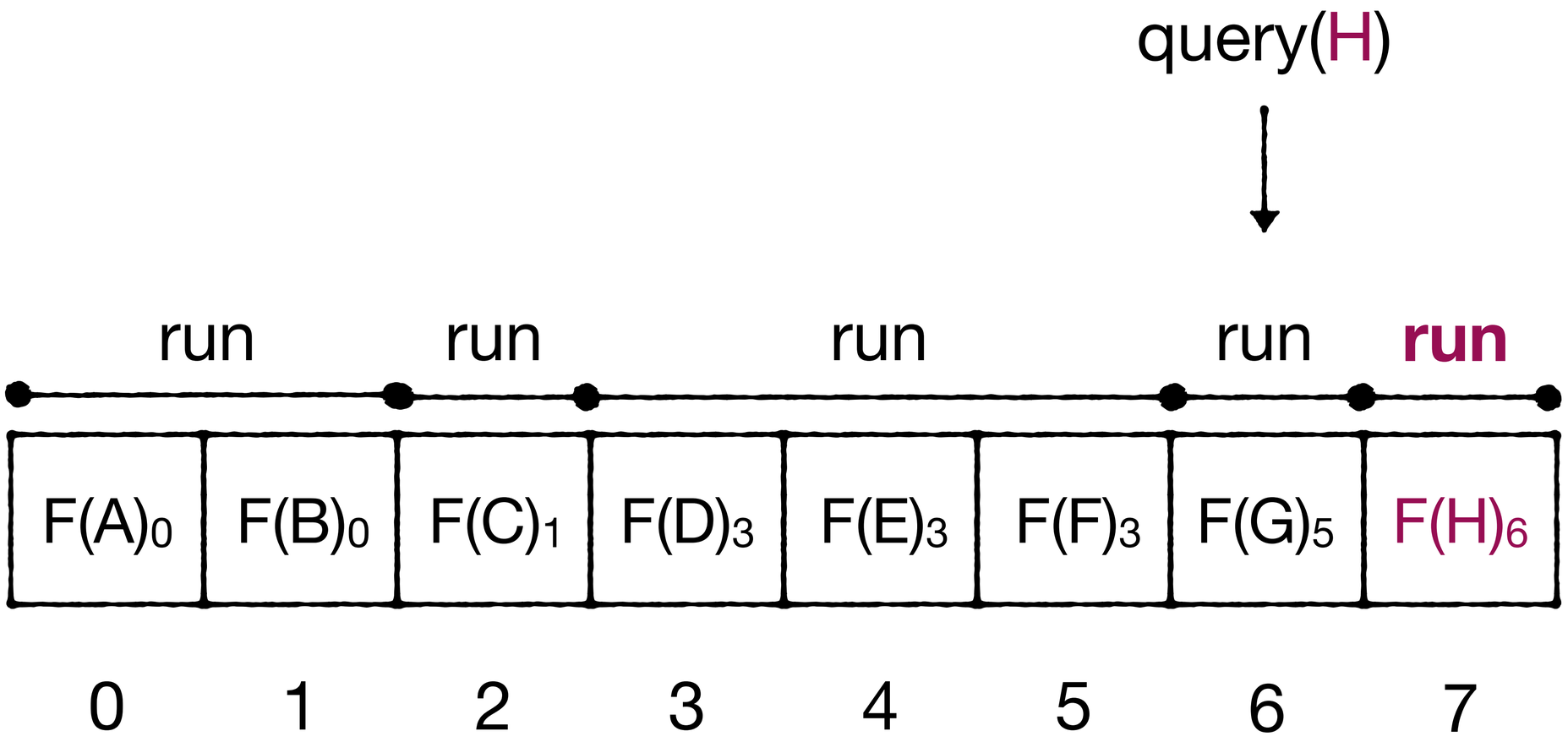


≈ On average, each run consists of 1 slot

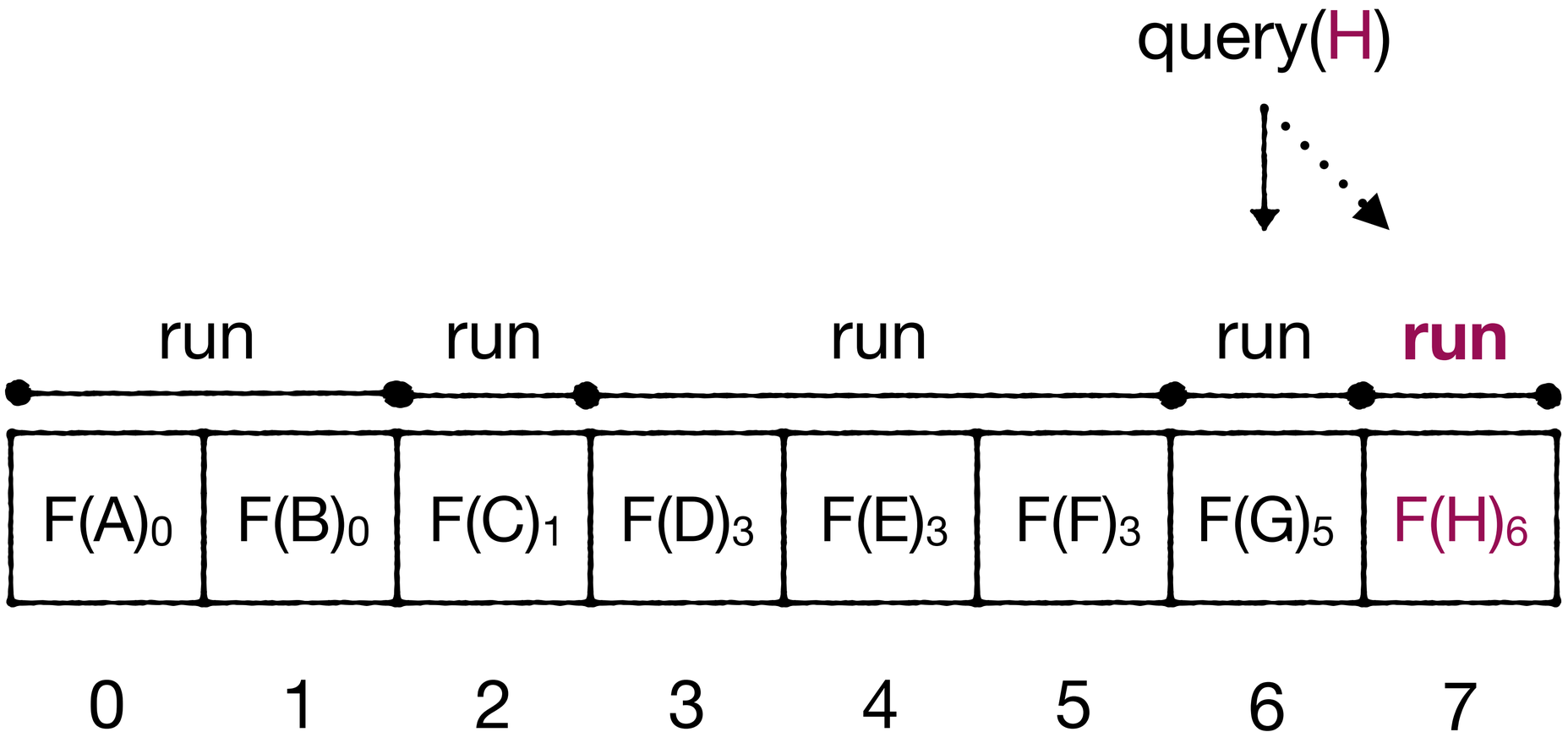



query(H)
run
run
run
run
run
F(A)0
F(B)0
F(C)1
F(D)3
F(E)3
F(F)3
F(G)5
F(H)6
0
1
2
3
4
5
6
7

Problem?

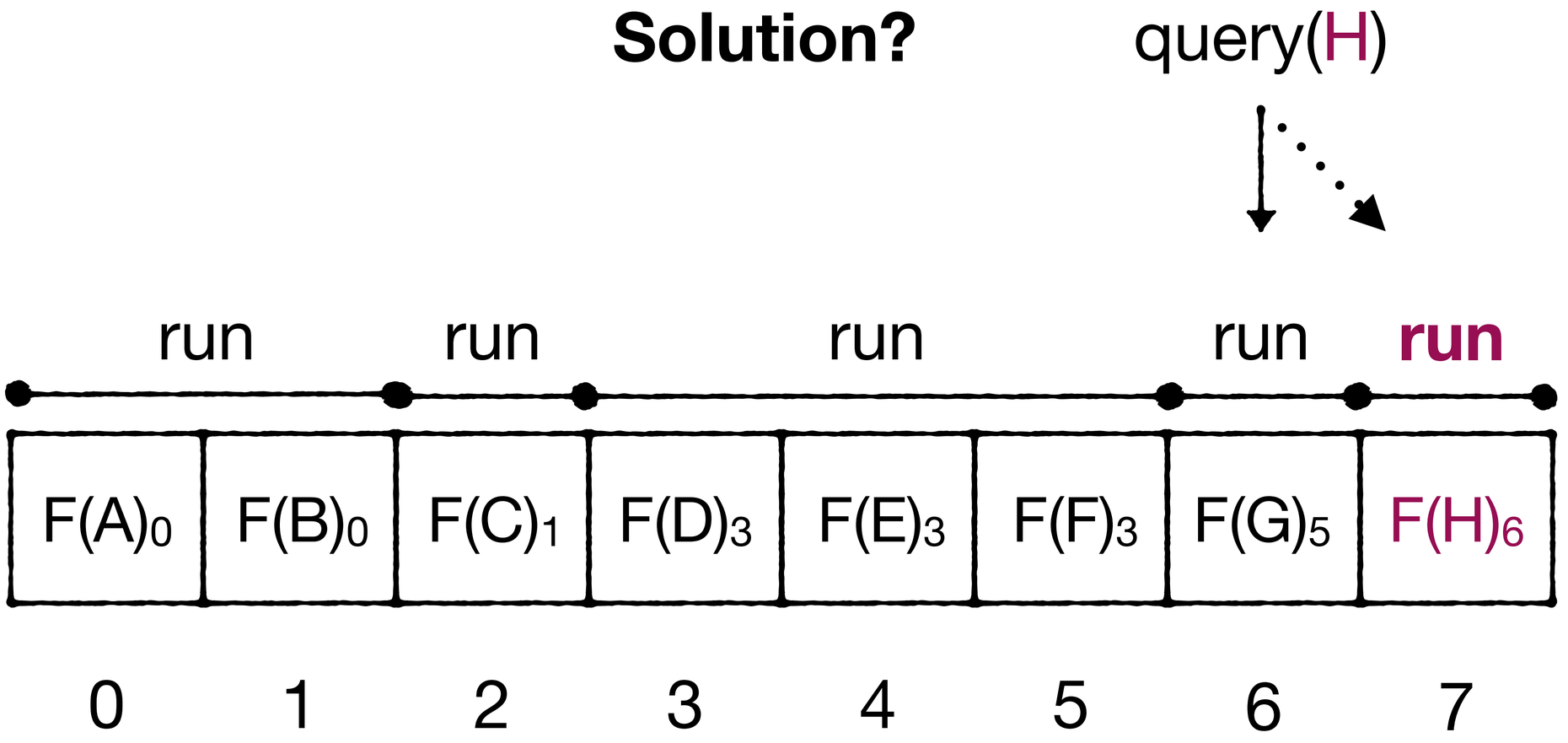

query(H)
run
run
run
run
run
F(A)0
F(B)0
F(C)1
F(D)3
F(E)3
F(F)3
F(G)5
F(H)6
0
1
2
3
4
5
6
7

Problem? **Fingerprint might have shifted to the right**



Problem? Fingerprint might have shifted to the right

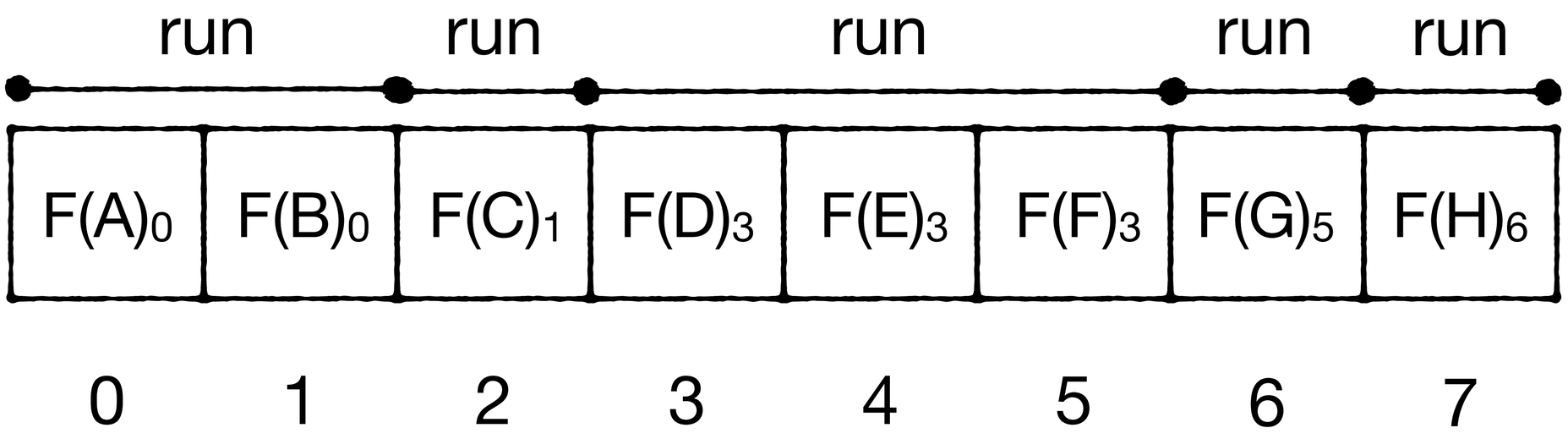
Solution?



Solution: delineate runs using 2 bitmaps

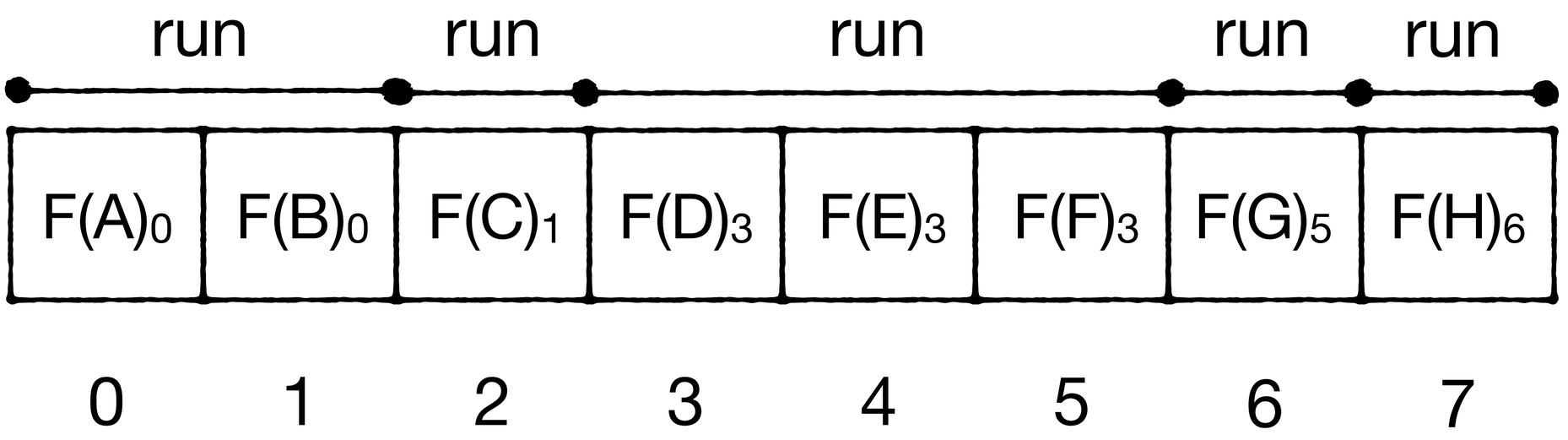
Occupied:

End:



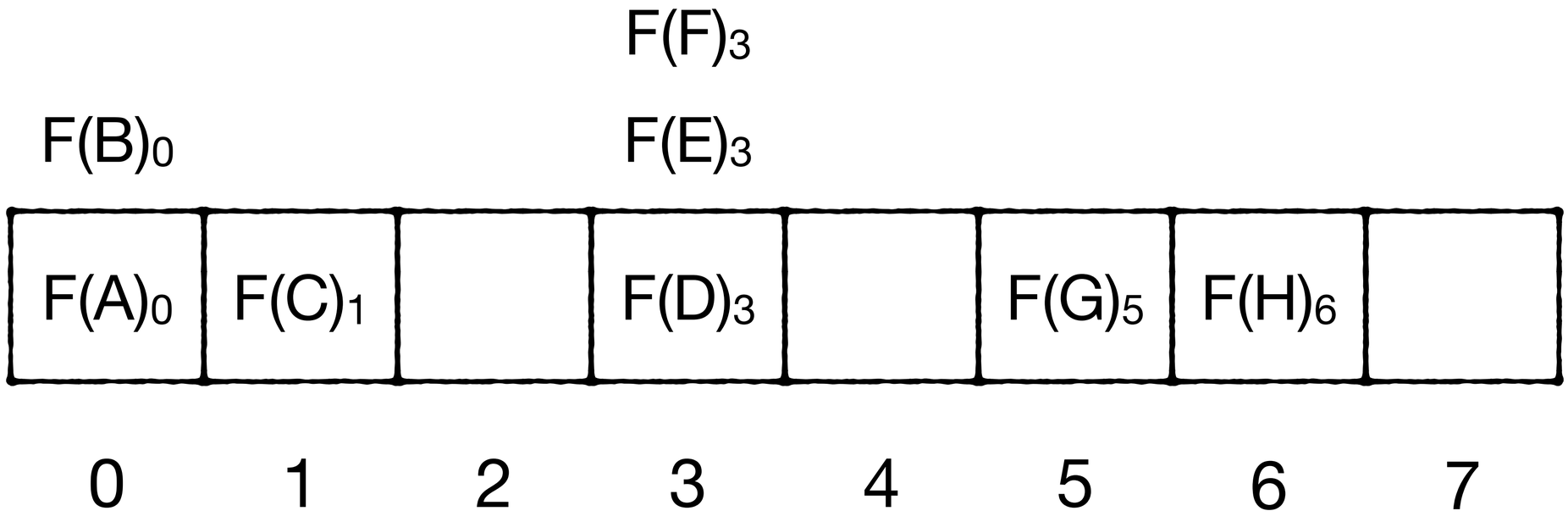
Occupied: **1 if there is a run belonging to this slot**

End:

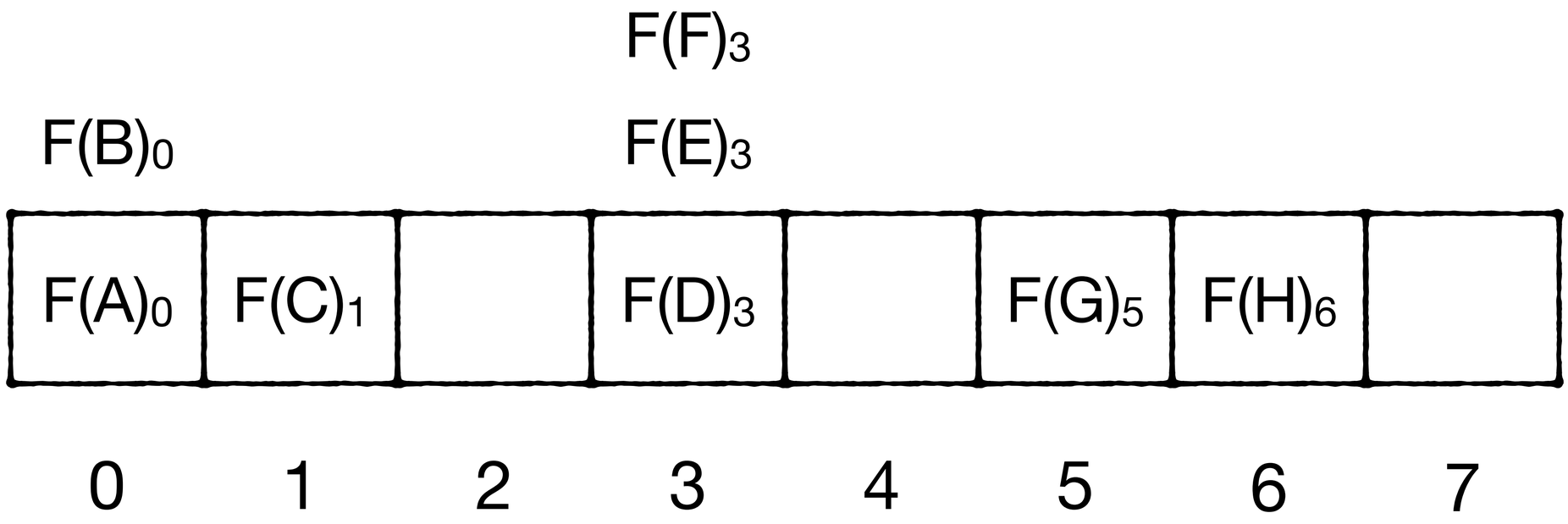


Occupied: **1 if there is a run belonging to this slot**

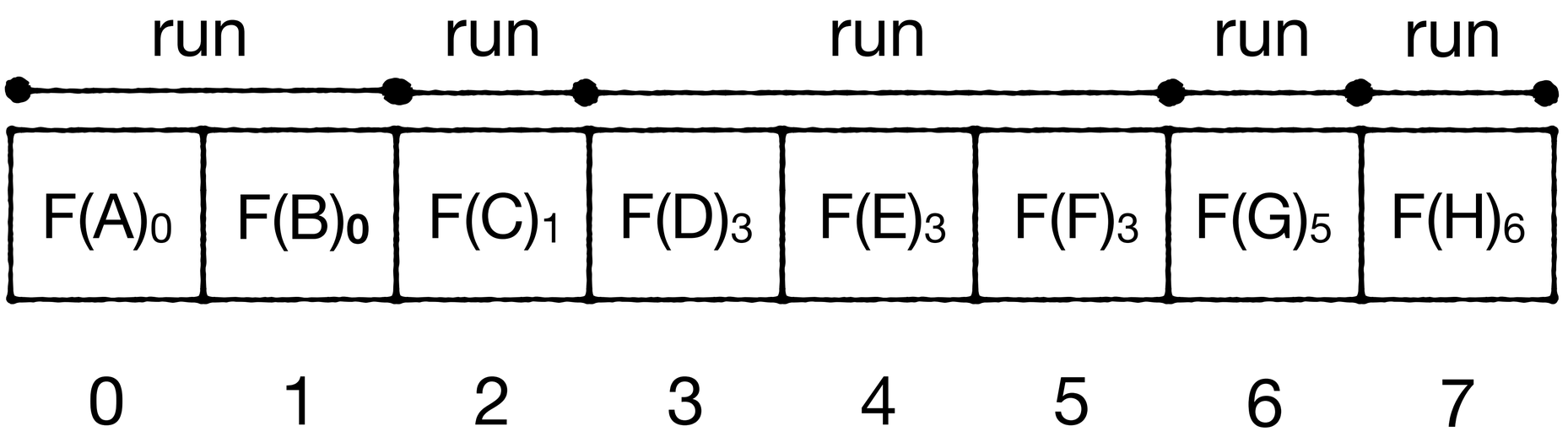
End:



Occupied:	**1**	**1**	0	**1**	0	**1**	**1**	0
End:								

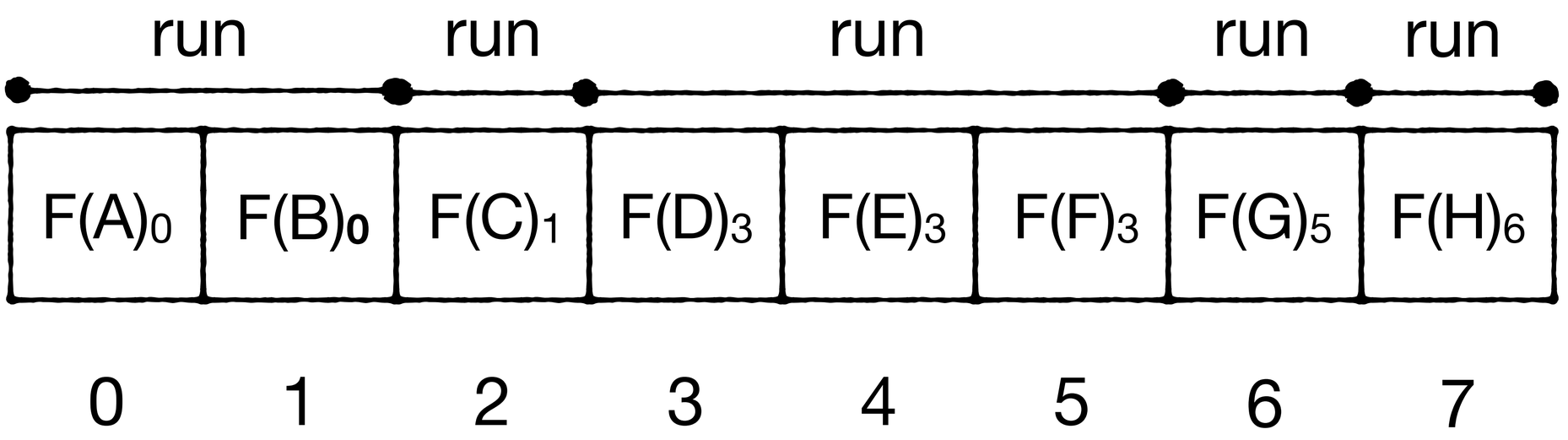


Occupied:	1	1	0	1	0	1	1	0
End:								

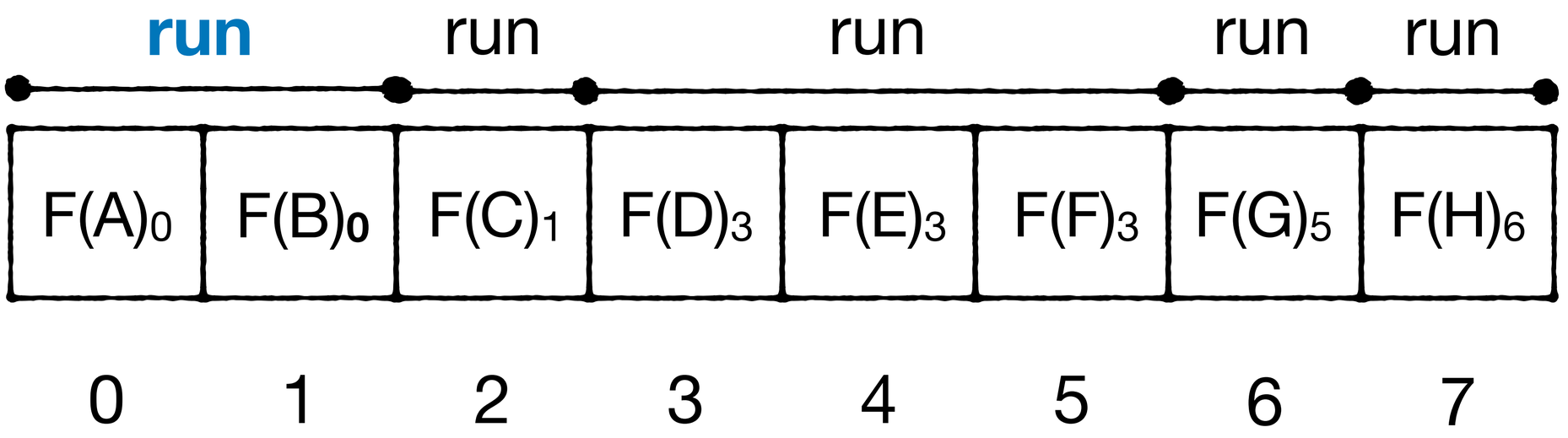


Occupied: 1 1 0 1 0 1 1 0

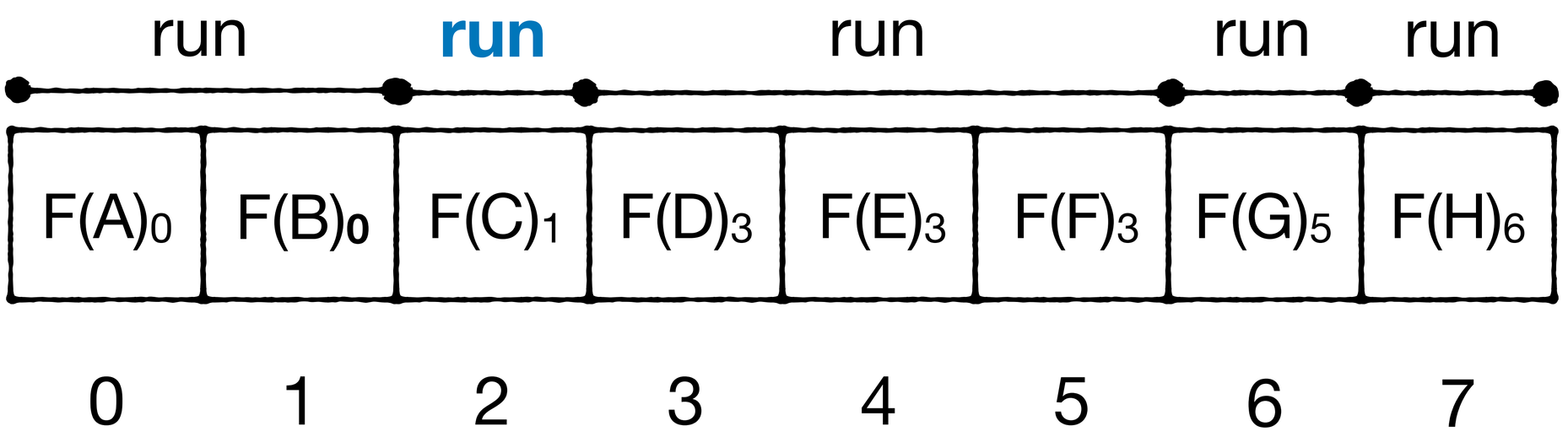
End: **1 for each slot where a run ends**



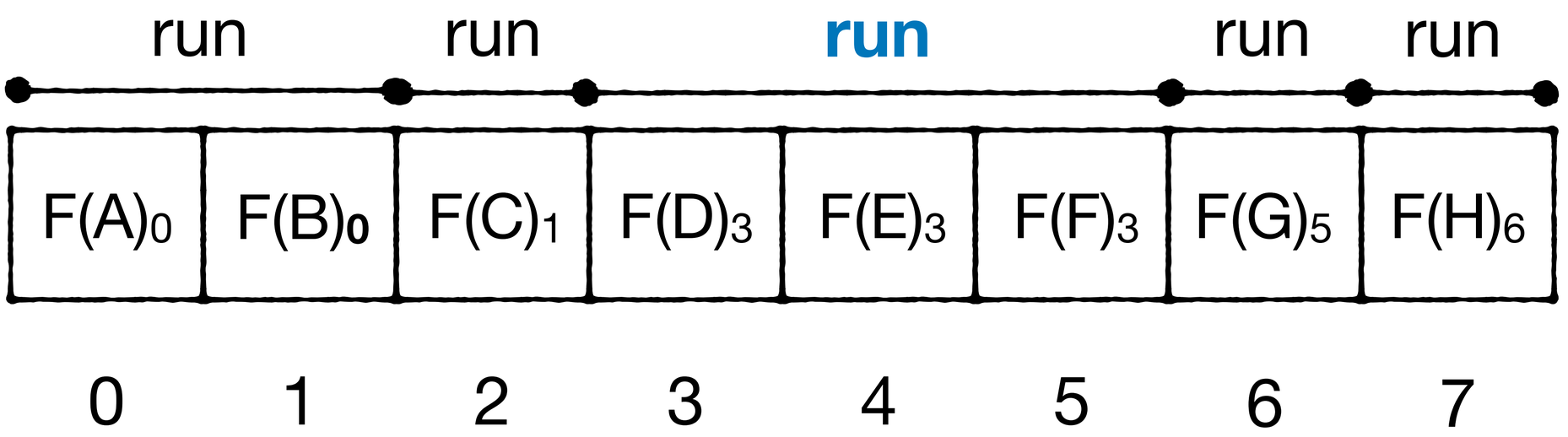
Occupied:	1	1	0	1	0	1	1	0
End:	**0**	**1**						



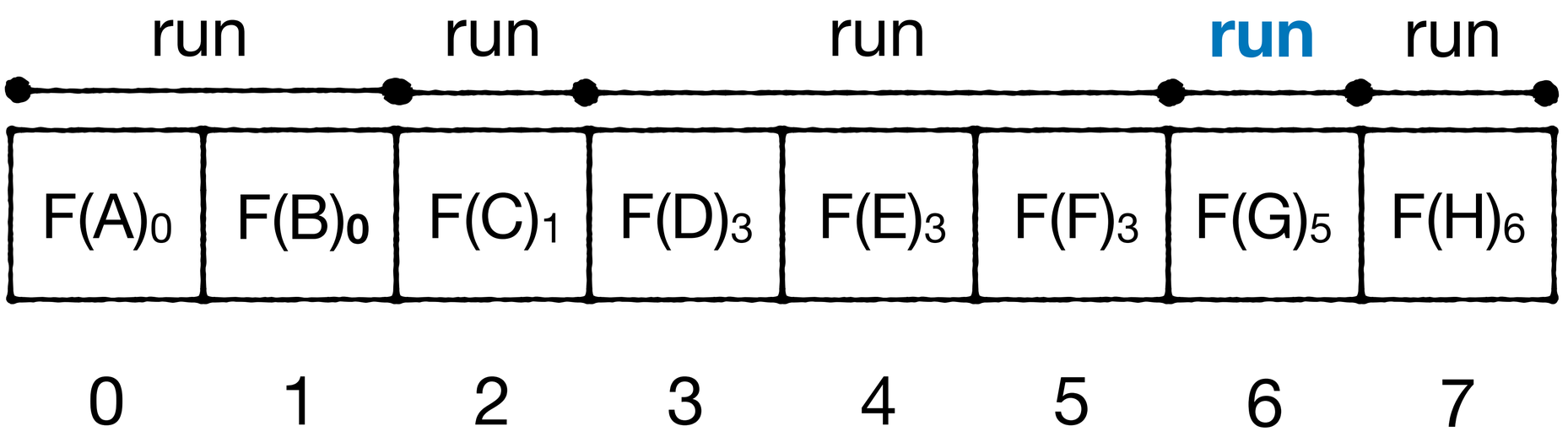
Occupied:	1	1	0	1	0	1	1	0
End:	0	1	**1**					



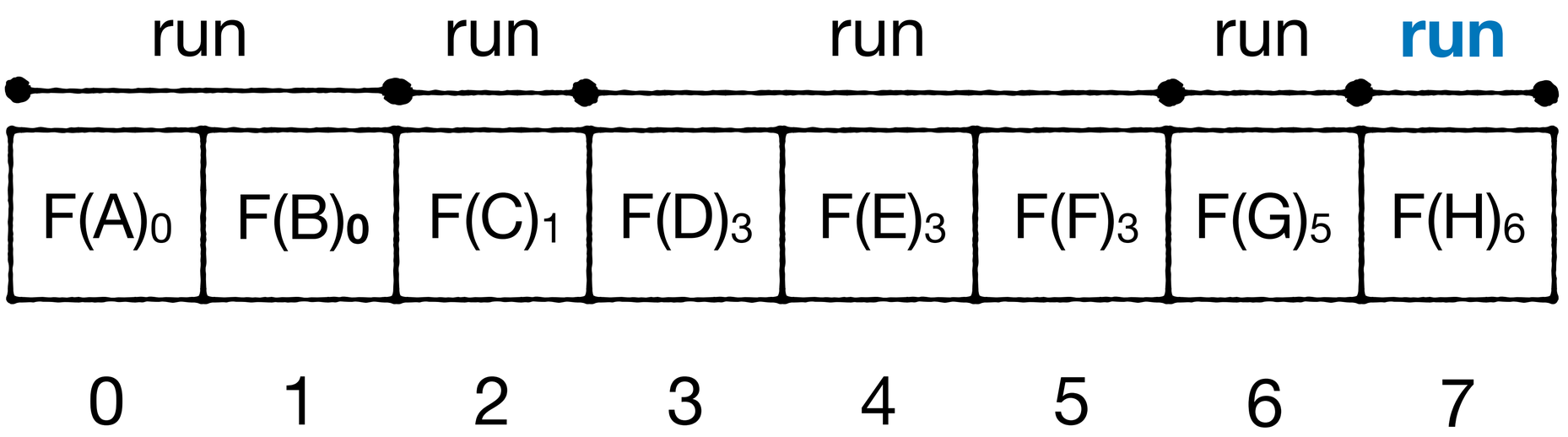
Occupied:	1	1	0	1	0	1	1	0
End:	0	1	1	**0**	**0**	**1**		



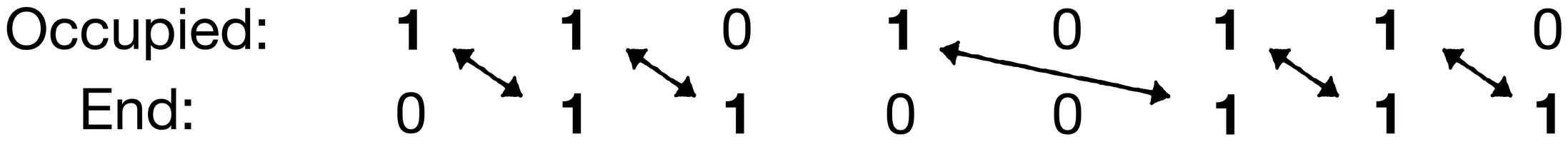
Occupied:	1	1	0	1	0	1	1	0
End:	0	1	1	0	0	1	**1**	

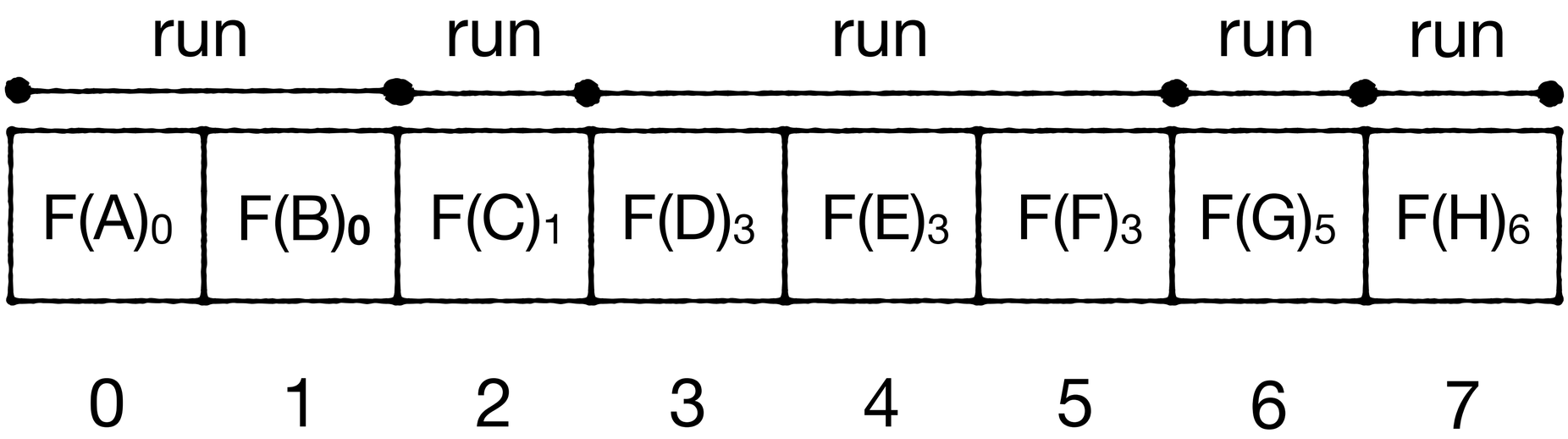


Occupied:	1	1	0	1	0	1	1	0
End:	0	1	1	0	0	1	1	**1**



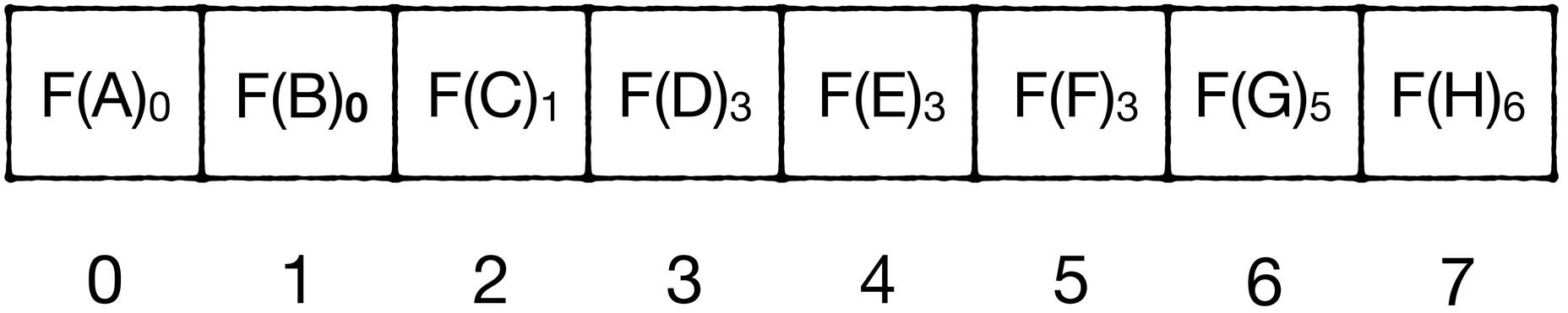
i^{th} set occupied bit corresponds to i^{th} set end bit





How to query?

Occupied:	1	1	0	1	0	1	1	0
End:	0	1	1	0	0	1	1	1



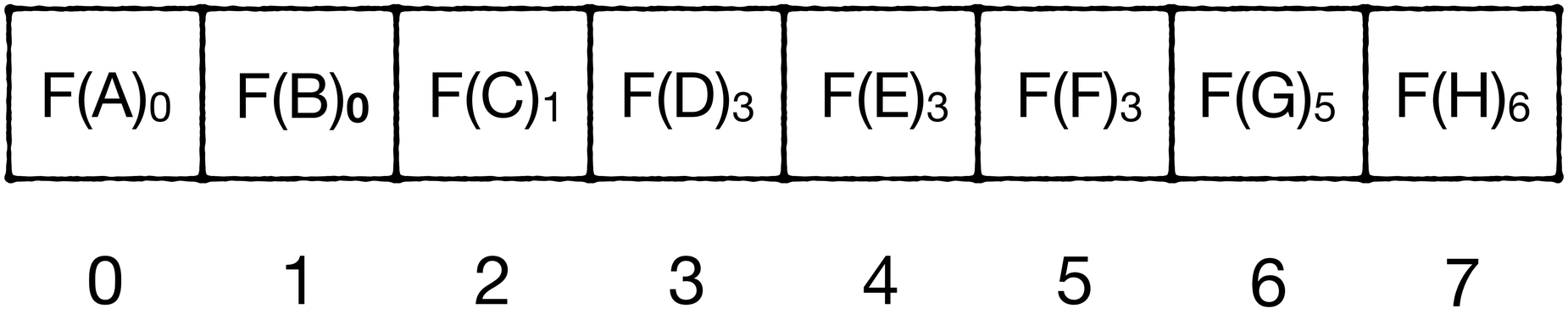
get(Z)

Occupied:	1	1	0	1	0	1	1	0
End:	0	1	1	0	0	1	1	1

$F(A)_0$	$F(B)_0$	$F(C)_1$	$F(D)_3$	$F(E)_3$	$F(F)_3$	$F(G)_5$	$F(H)_6$
0	1	2	3	4	5	6	7

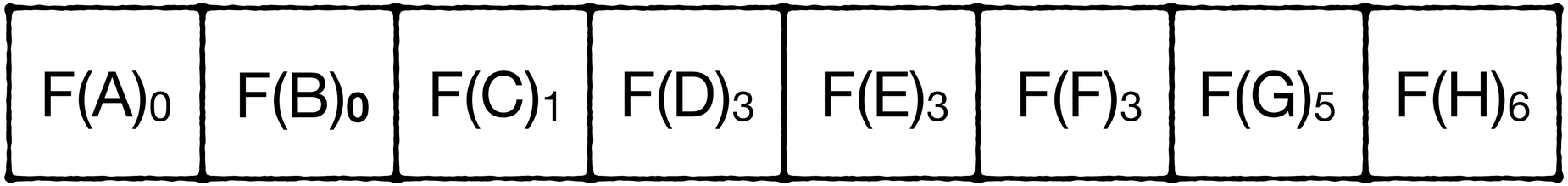
get(Z) **return negative**

Occupied:	1	1	0	1	**0**	1	1	0
End:	0	1	1	0	0	1	1	1

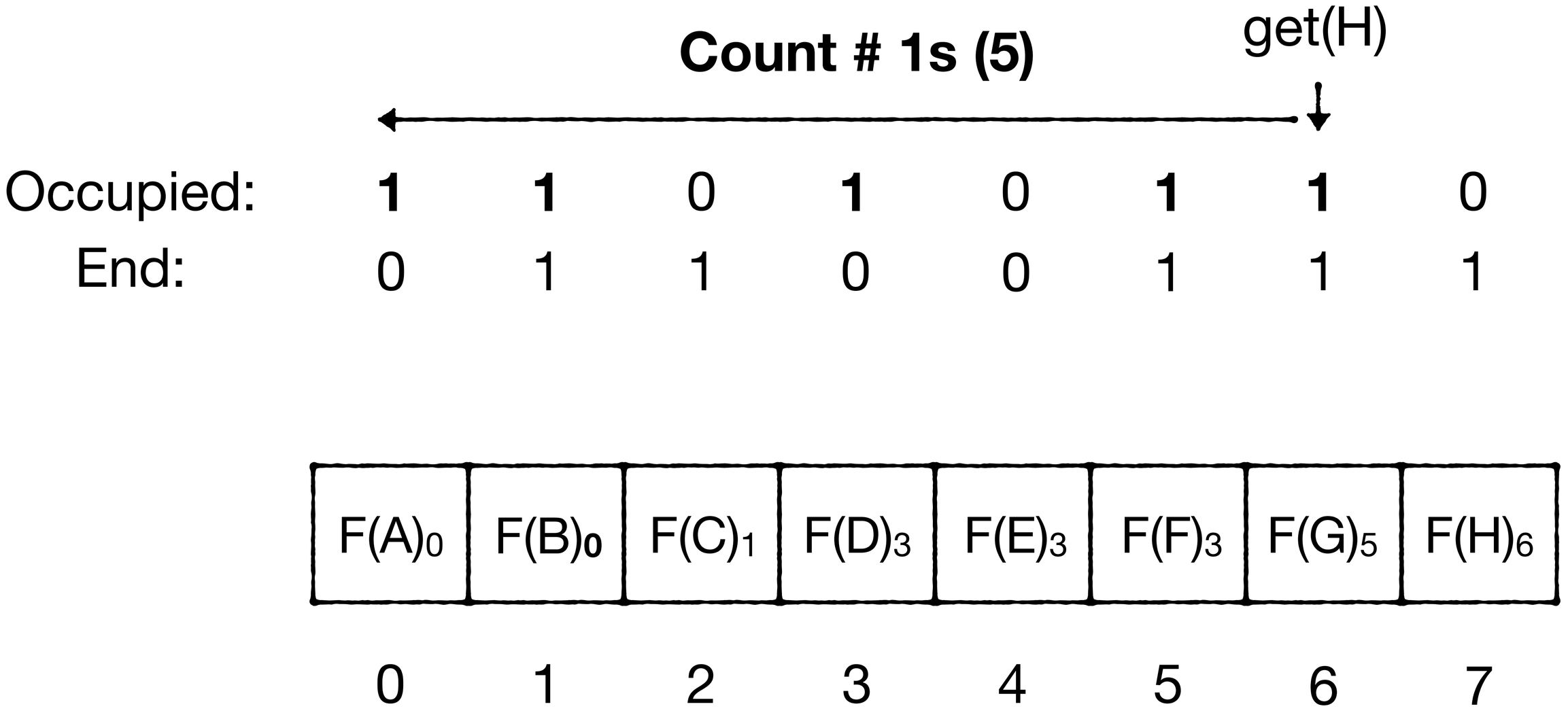


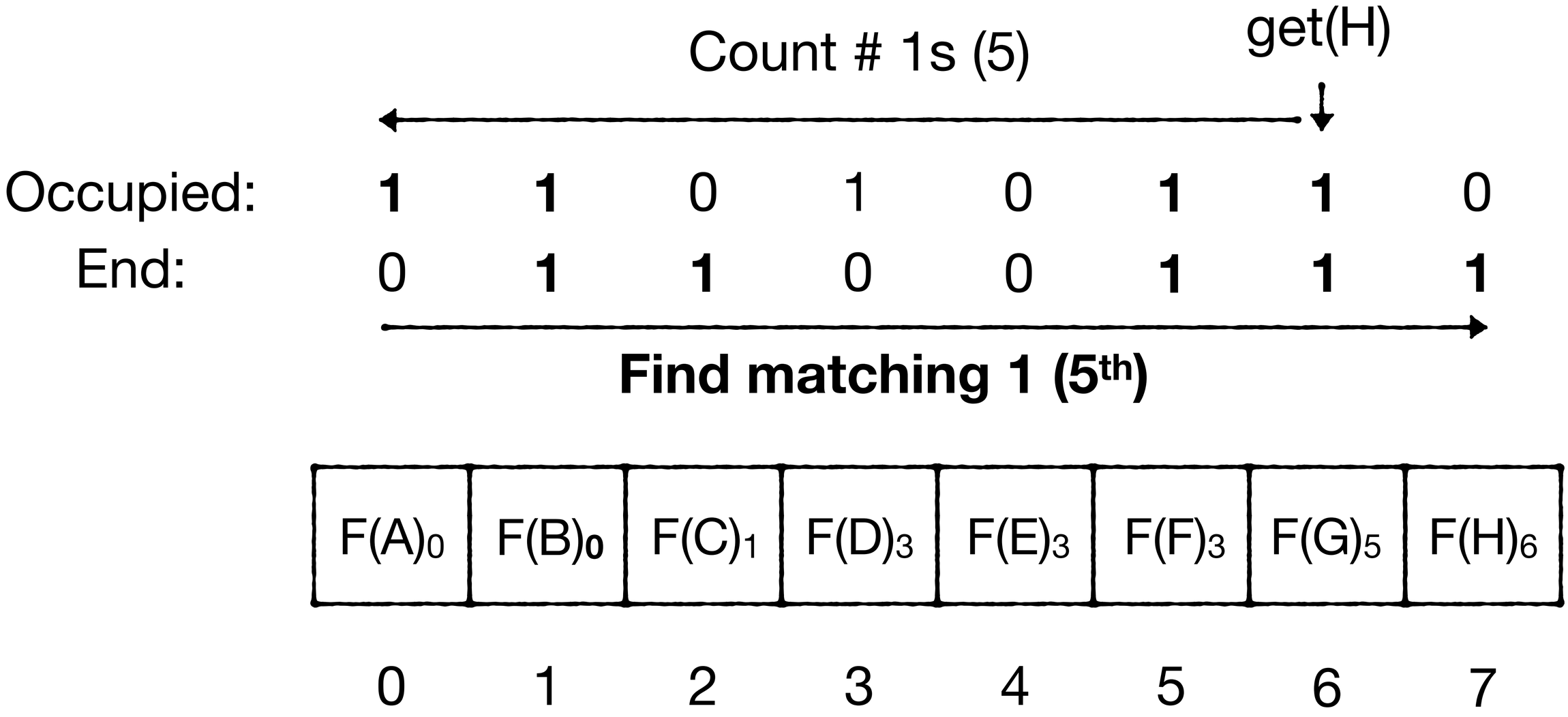
get(H)

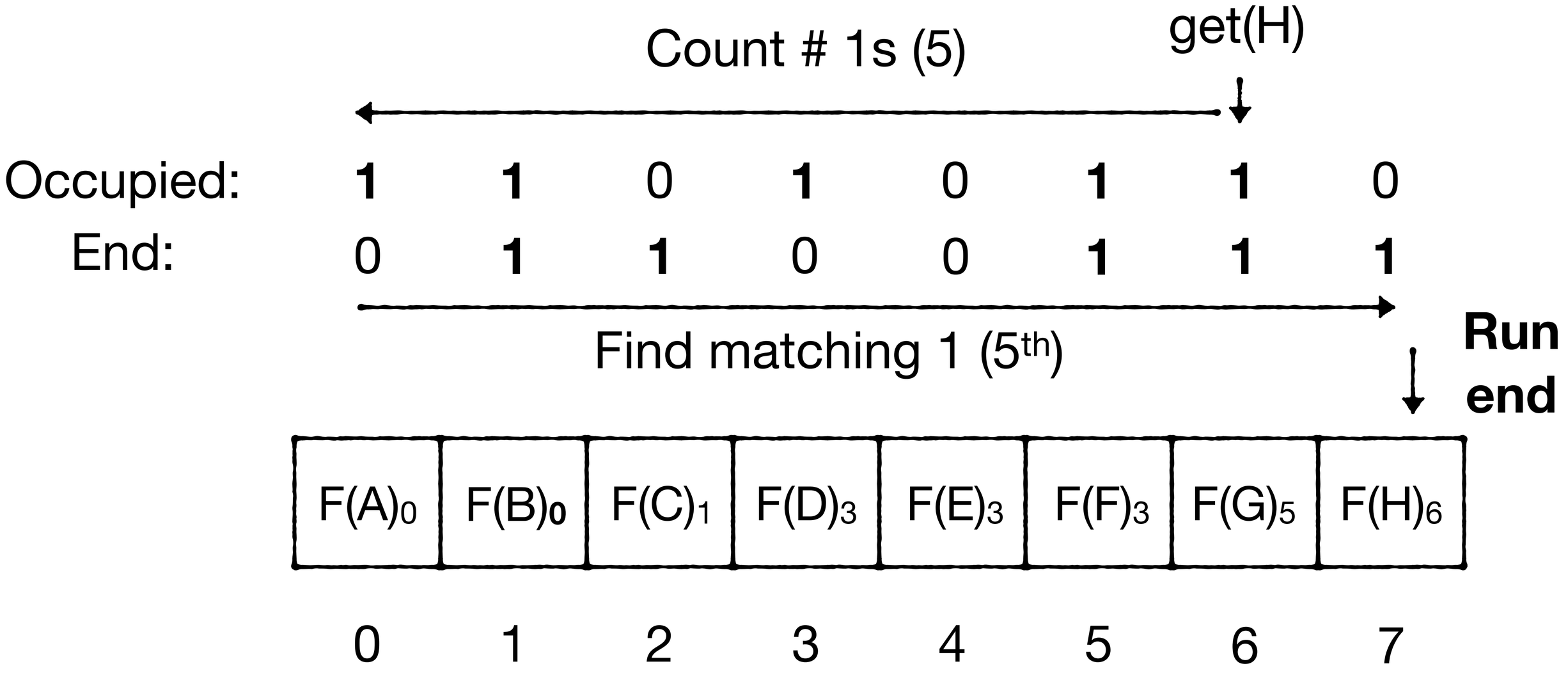
Occupied:	1	1	0	1	0	1	**1**	0
End:	0	1	1	0	0	1	1	1

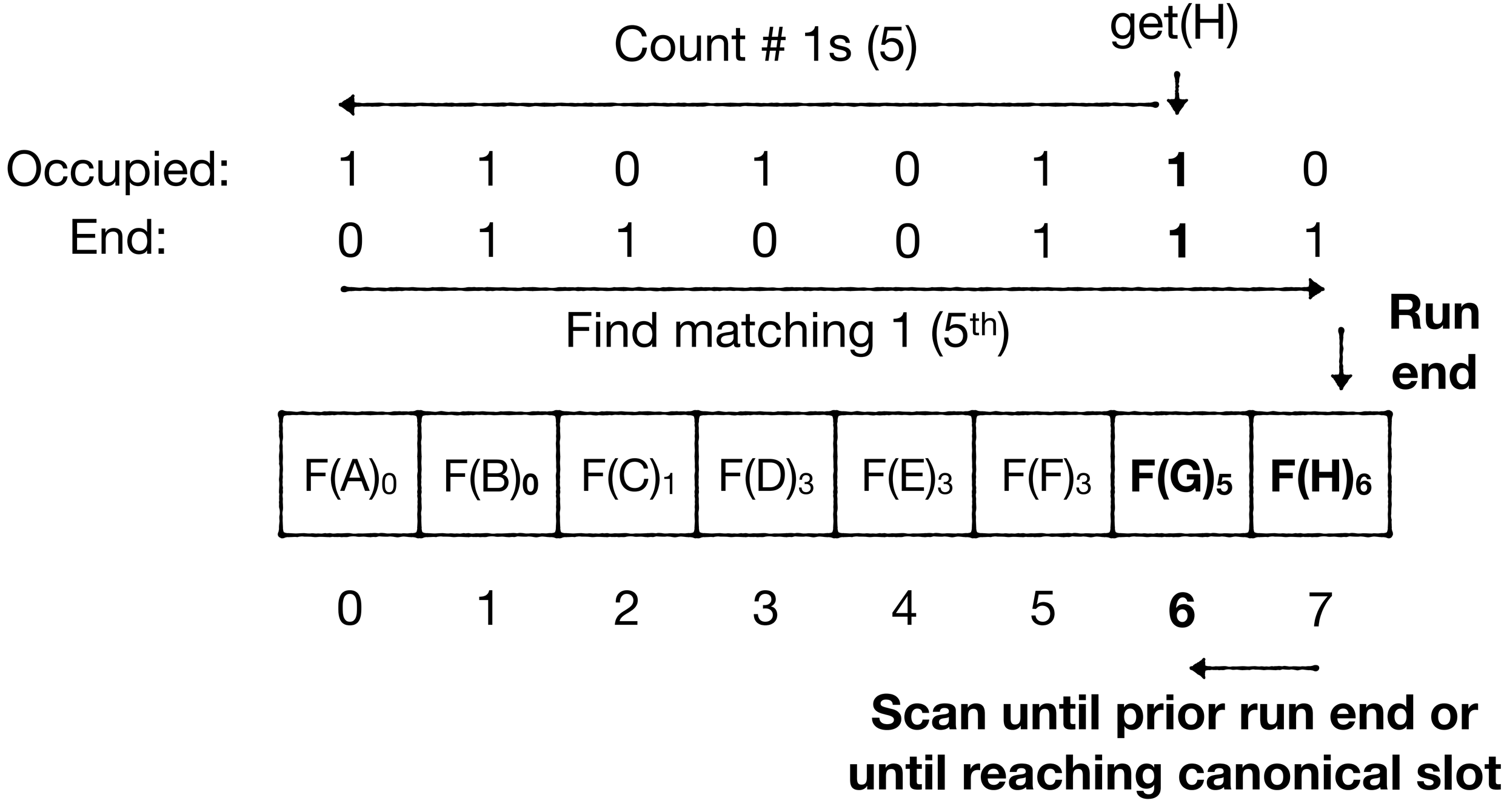


0	1	2	3	4	5	6	7


Count # 1s (5)
get(H)
Occupied: 1 1 0 1 0 1 1 0
End: 0 1 1 0 0 1 1 1
F(A)0
F(B)0
F(C)1
F(D)3
F(E)3
F(F)3
F(G)5
F(H)6
0 1 2 3 4 5 6 7

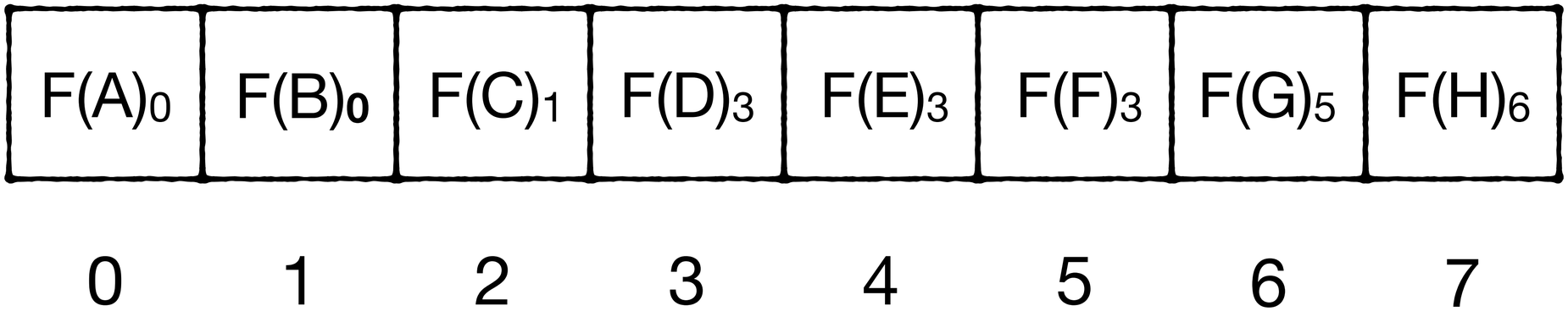

Count # 1s (5)
get(H)
Occupied:
1 1 0 1 0 1 1 0
End:
0 1 1 0 0 1 1 1
Find matching 1 (5th)
F(A)0
F(B)0
F(C)1
F(D)3
F(E)3
F(F)3
F(G)5
F(H)6
0 1 2 3 4 5 6 7


Count # 1s (5)
get(H)
Occupied:
1 1 0 1 0 1 1 0
End:
0 1 1 0 0 1 1 1
Find matching 1 (5th)
Run end
F(A)0
F(B)0
F(C)1
F(D)3
F(E)3
F(F)3
F(G)5
F(H)6
0 1 2 3 4 5 6 7


Count # 1s (5)
get(H)
Occupied: 1 1 0 1 0 1 1 0
End: 0 1 1 0 0 1 1 1
Find matching 1 (5th)
Run end
F(A)0 F(B)0 F(C)1 F(D)3 F(E)3 F(F)3 F(G)5 F(H)6
0 1 2 3 4 5 6 7
Scan until prior run end or until reaching canonical slot

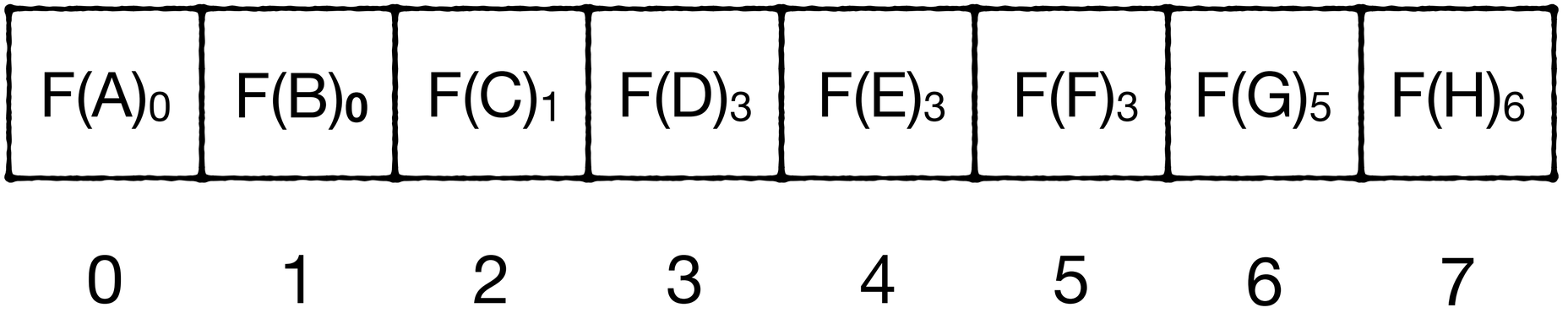
Can handle queries :)

Occupied:	1	1	0	1	0	1	1	0
End:	0	1	1	0	0	1	1	1



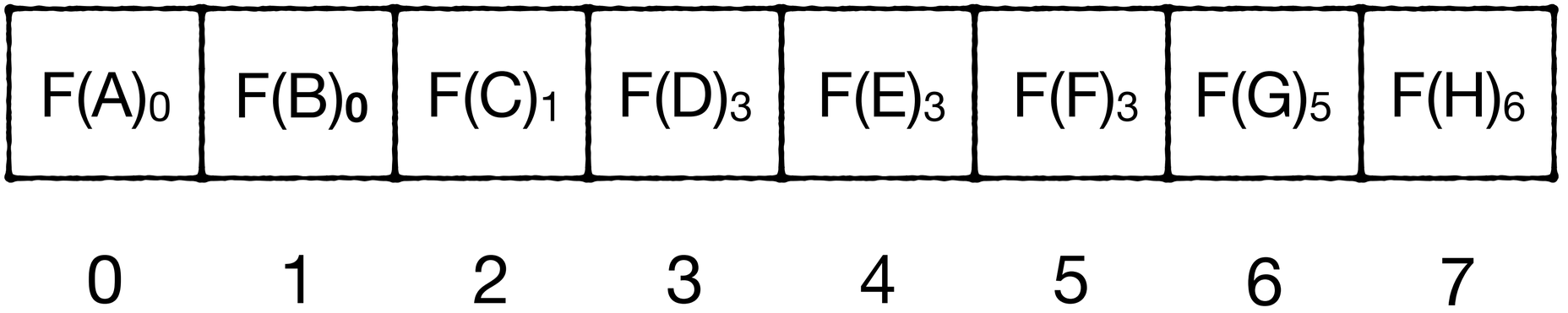
Can handle queries :) **problem?**

Occupied:	1	1	0	1	0	1	1	0
End:	0	1	1	0	0	1	1	1



Scanning bitmaps takes O(N)

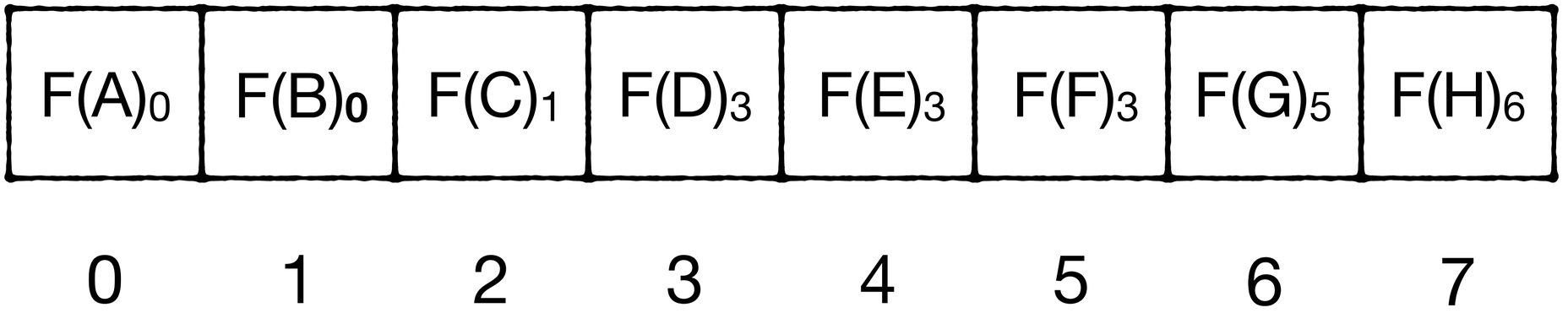
Occupied:	1	1	0	1	0	1	1	0
End:	0	1	1	0	0	1	1	1



Scanning bitmaps takes O(N)

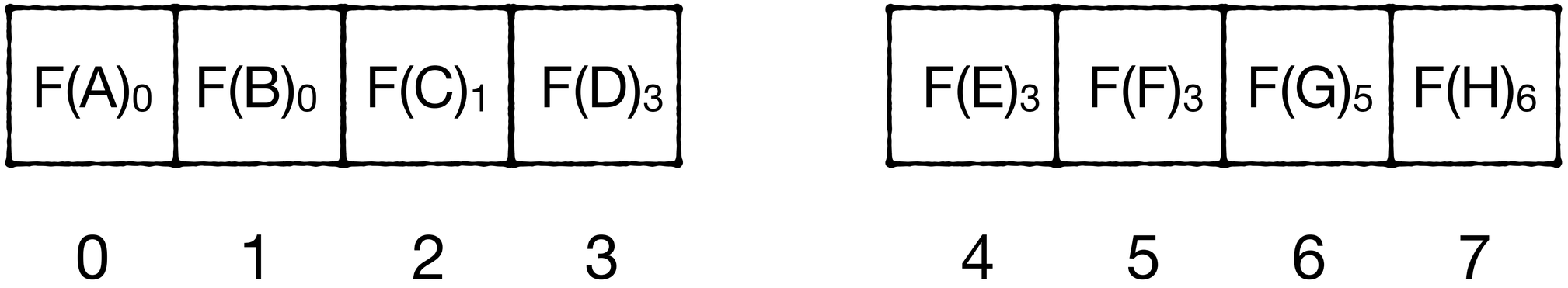
Ideas?

Occupied:	1	1	0	1	0	1	1	0
End:	0	1	1	0	0	1	1	1



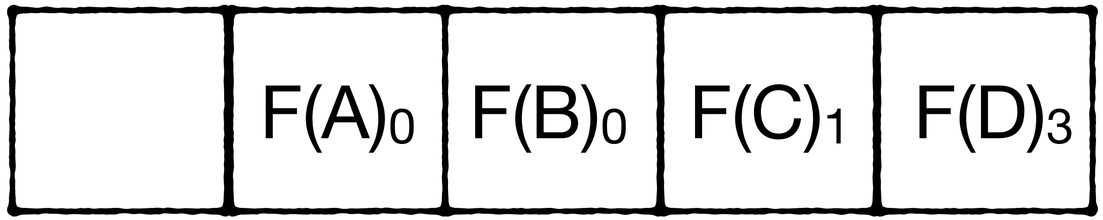
Split filter into chunks (64 slots in practice)

Occupied:	1	1	0	1	0	1	1	0
End:	0	1	1	0	0	1	1	1

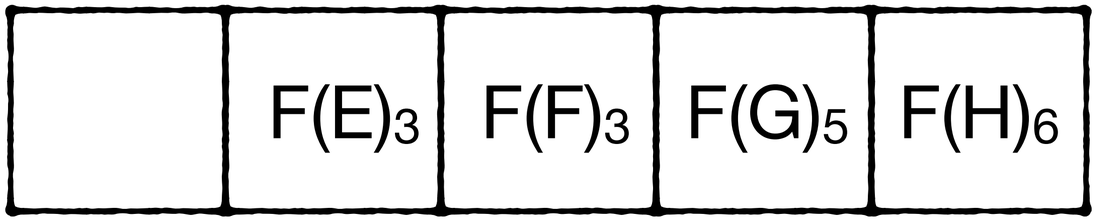


Each chunk has offset field (8 bits)

Occupied:	1	1	0	1	0	1	1	0
End:	0	1	1	0	0	1	1	1



Offset 0 1 2 3

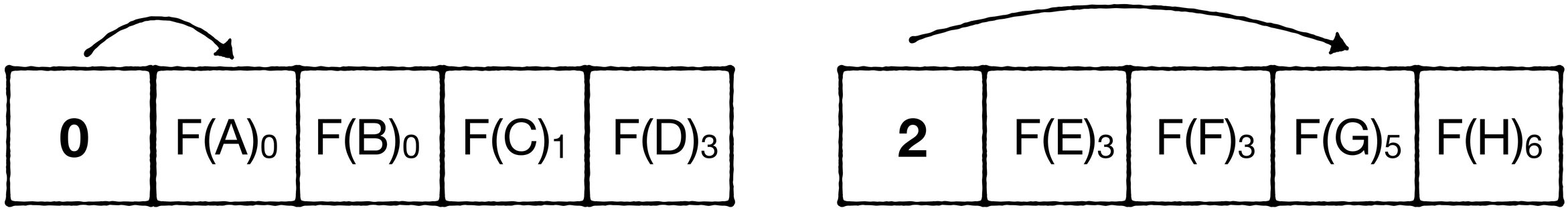


Offset 4 5 6 7

Each chunk has offset field

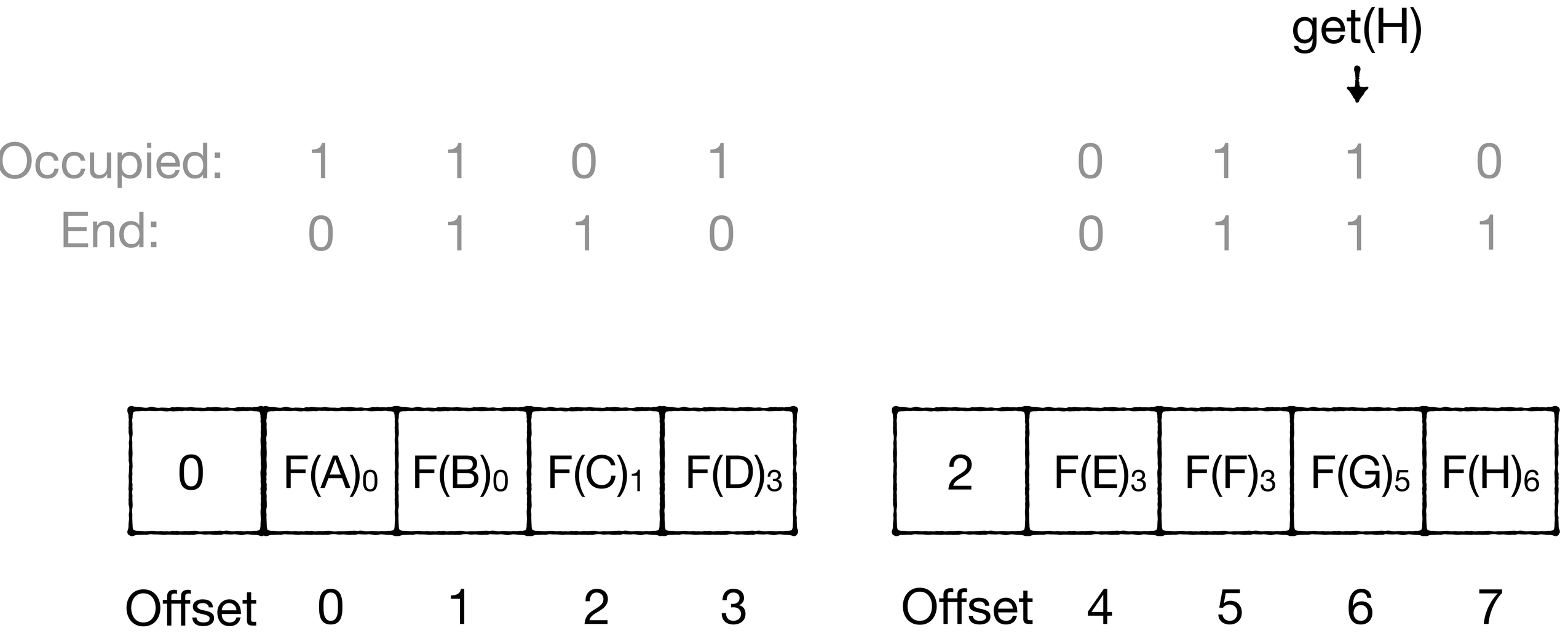
Measures distance to first entry of chunk

Occupied:	1	1	0	1		0	1	1	0
End:	0	1	1	0		0	1	1	1

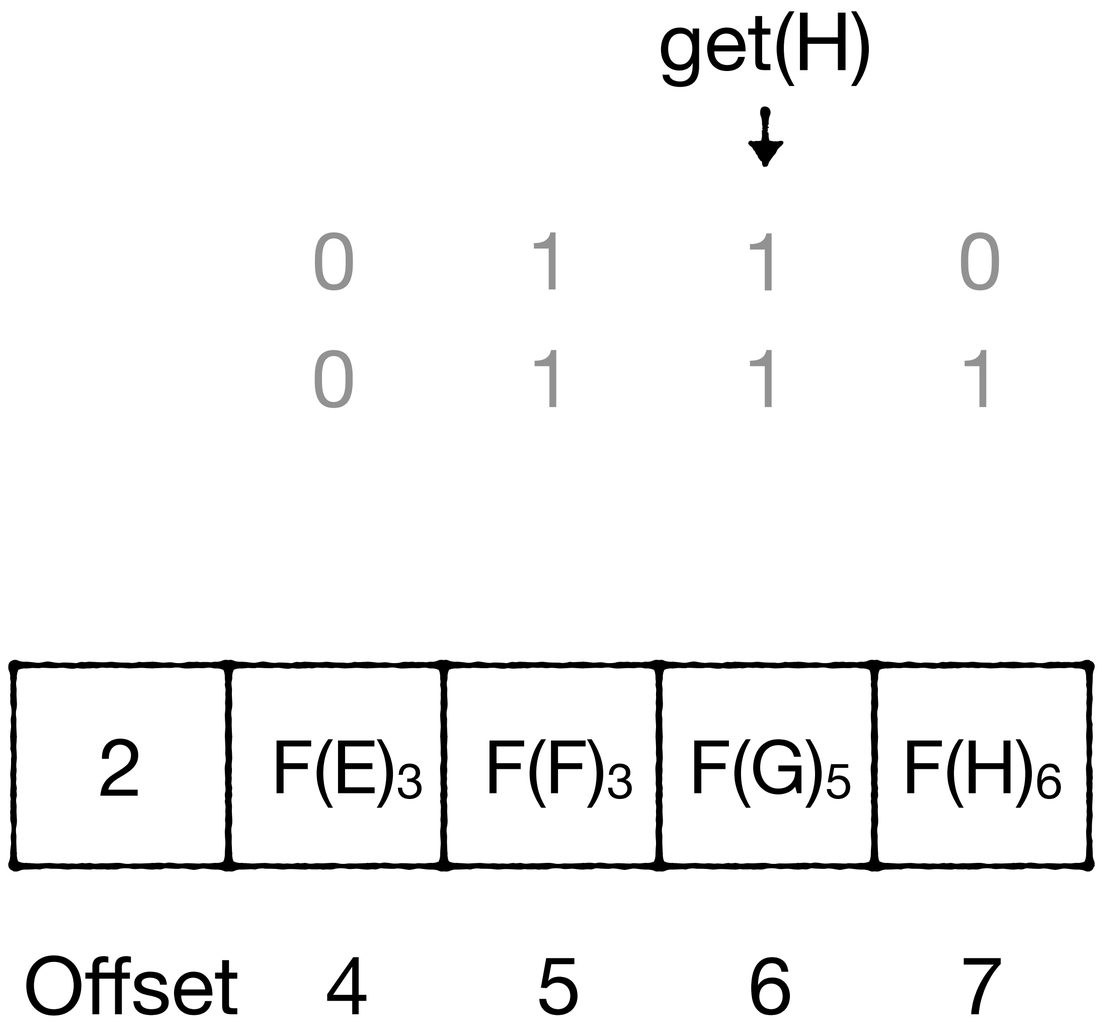


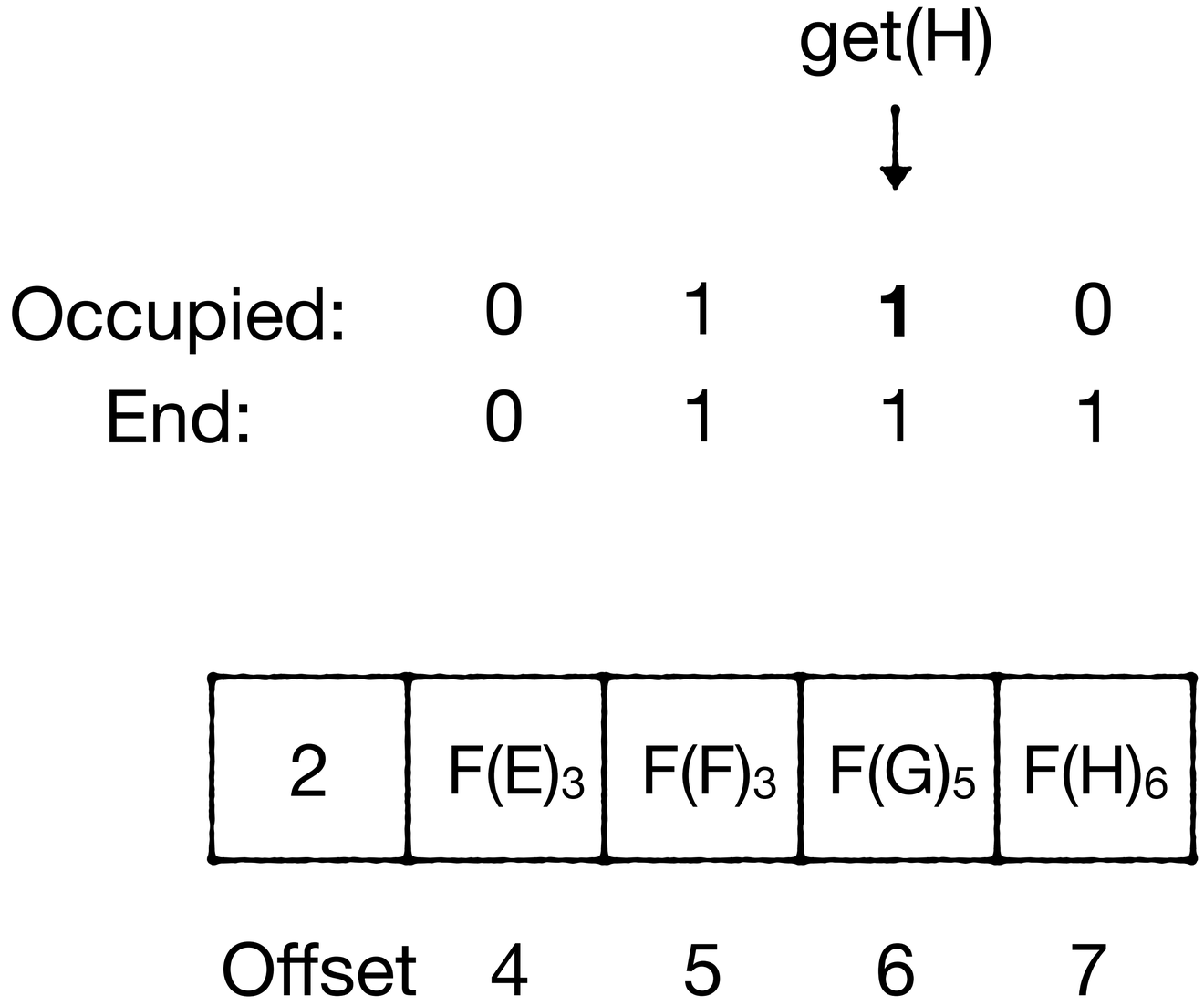
Offset	0	1	2	3	**Offset**	4	5	6	7

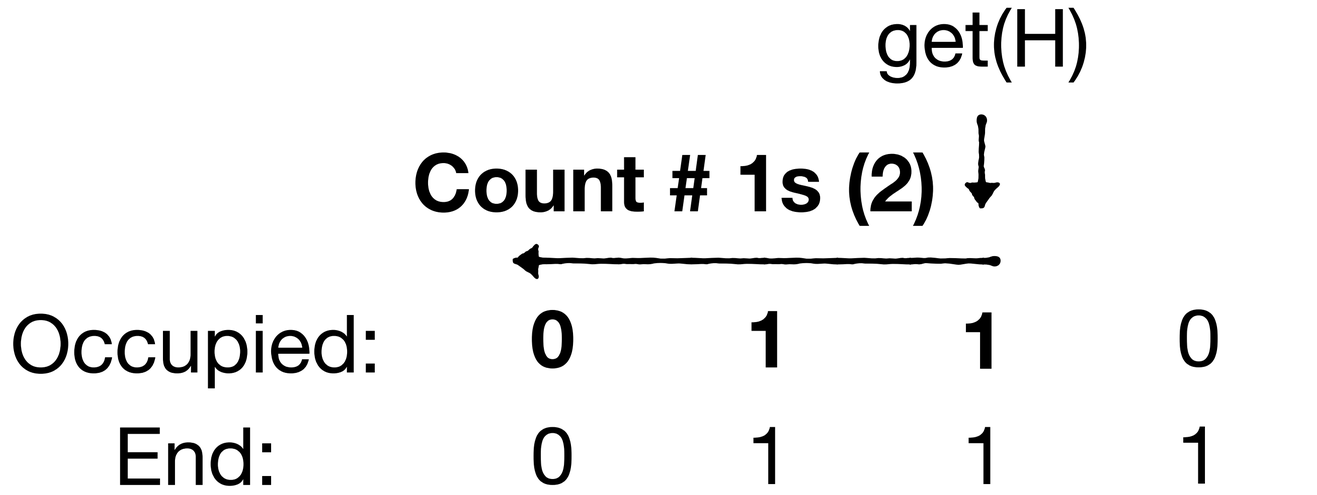
Back to Example

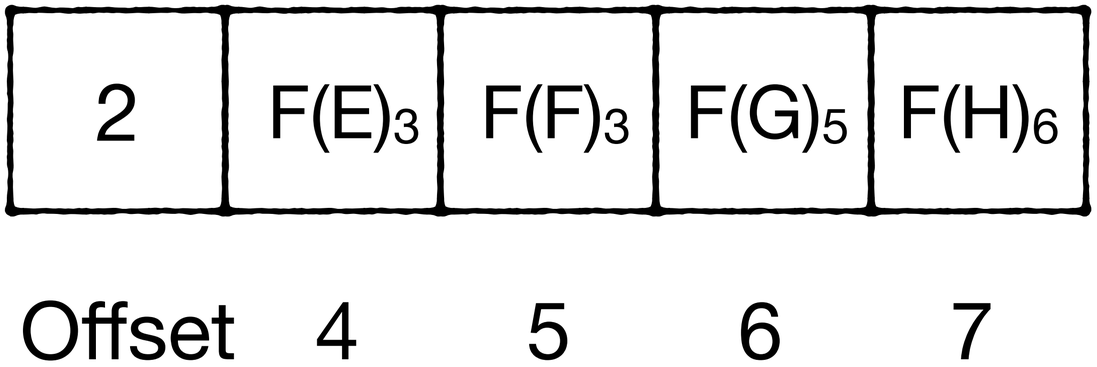


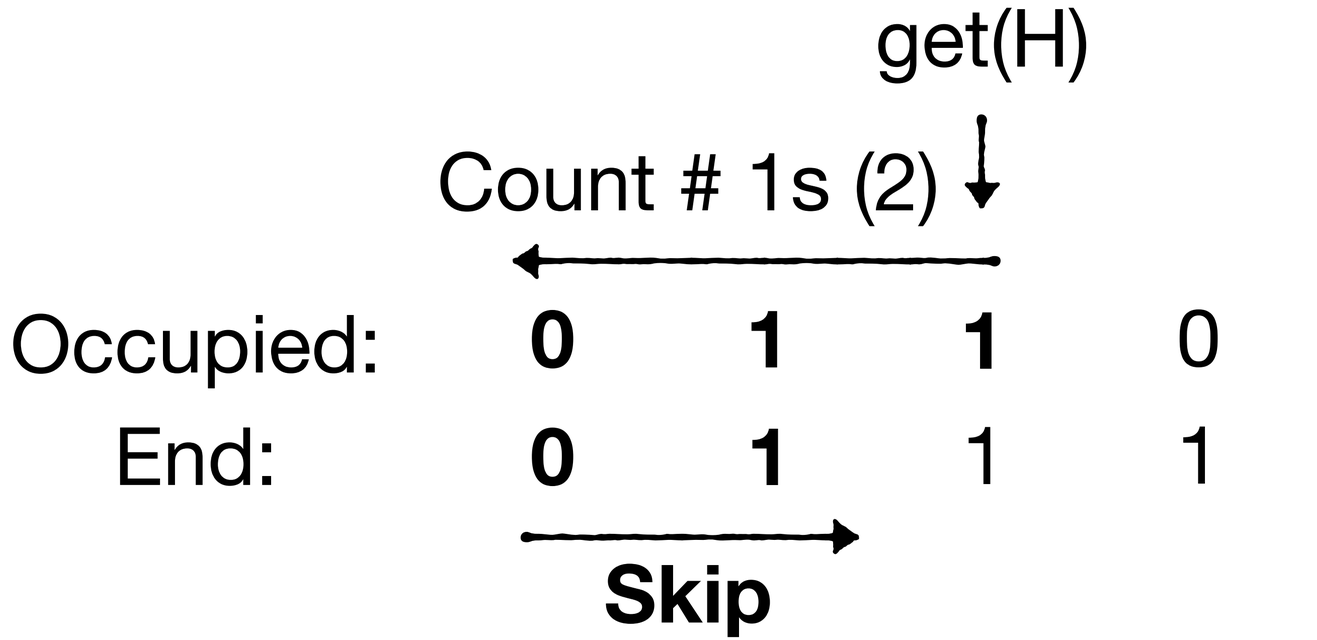
Back to Example




get(H)
Occupied: 0 1 1 0
End: 0 1 1 1
2
F(E)3
F(F)3
F(G)5
F(H)6
Offset 4 5 6 7

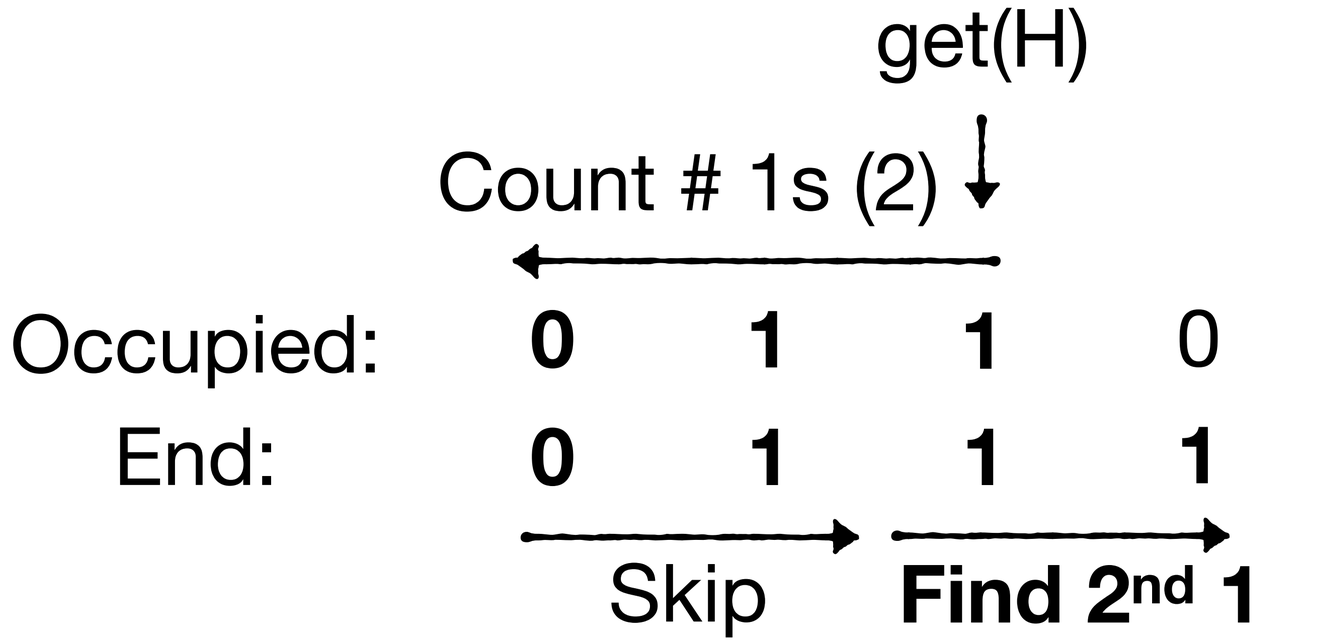

get(H)
Count # 1s (2)
Occupied: 0 1 1 0
End: 0 1 1 1

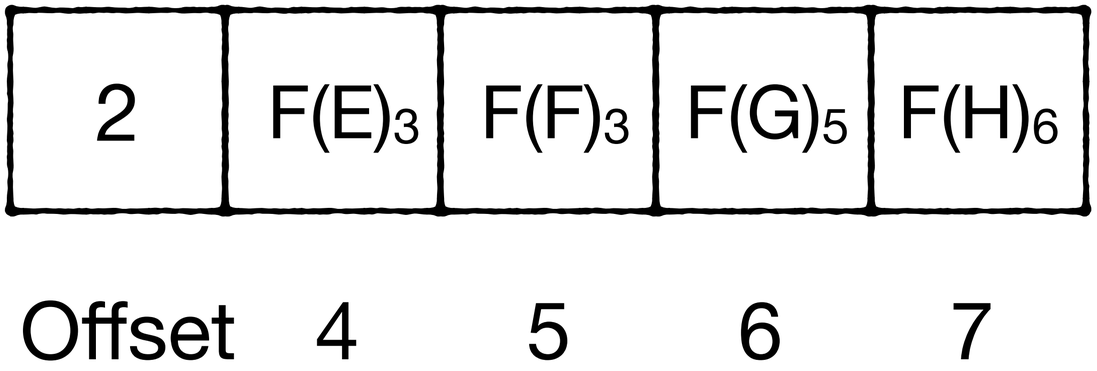

2
F(E)3
F(F)3
F(G)5
F(H)6
Offset 4 5 6 7

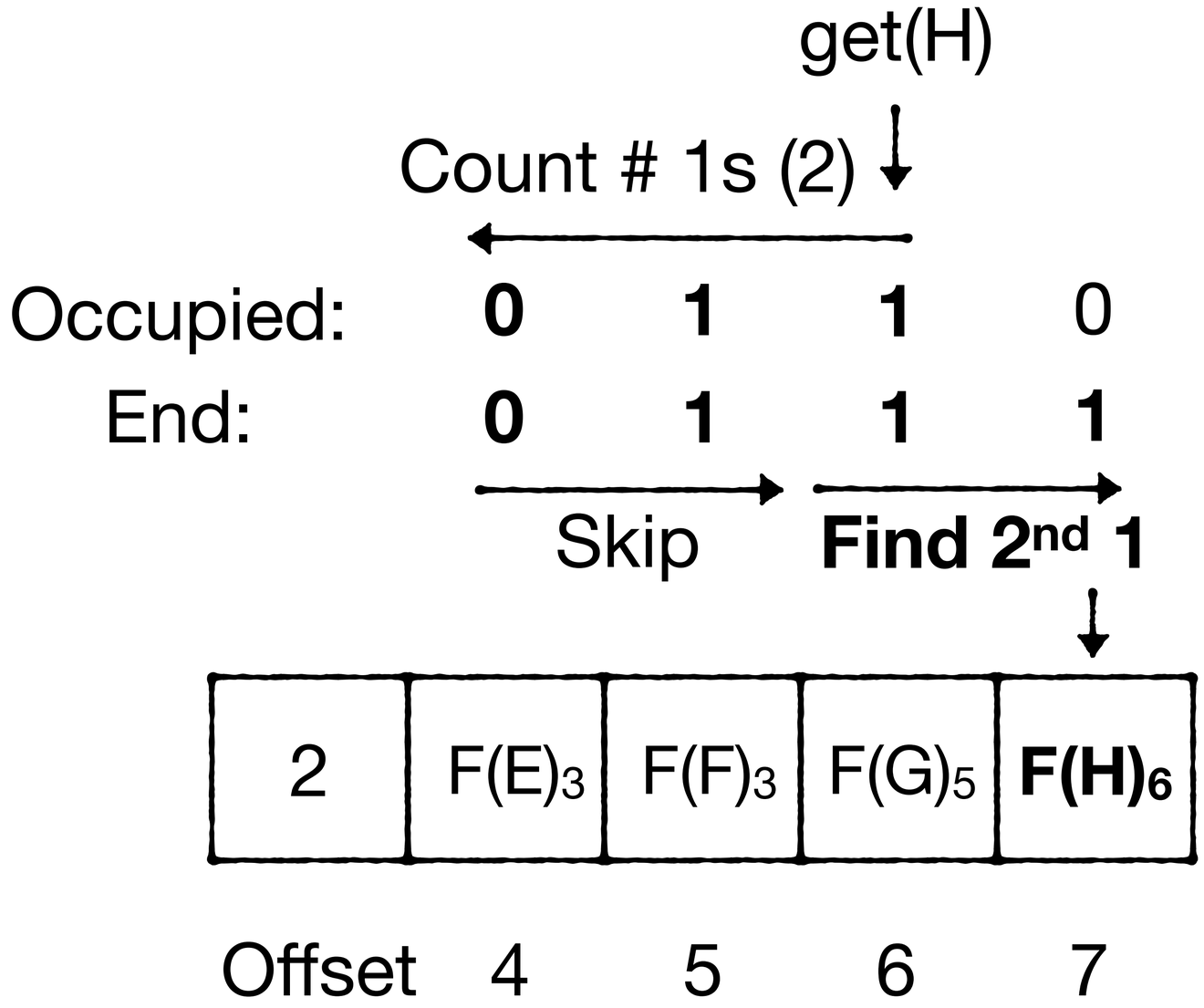


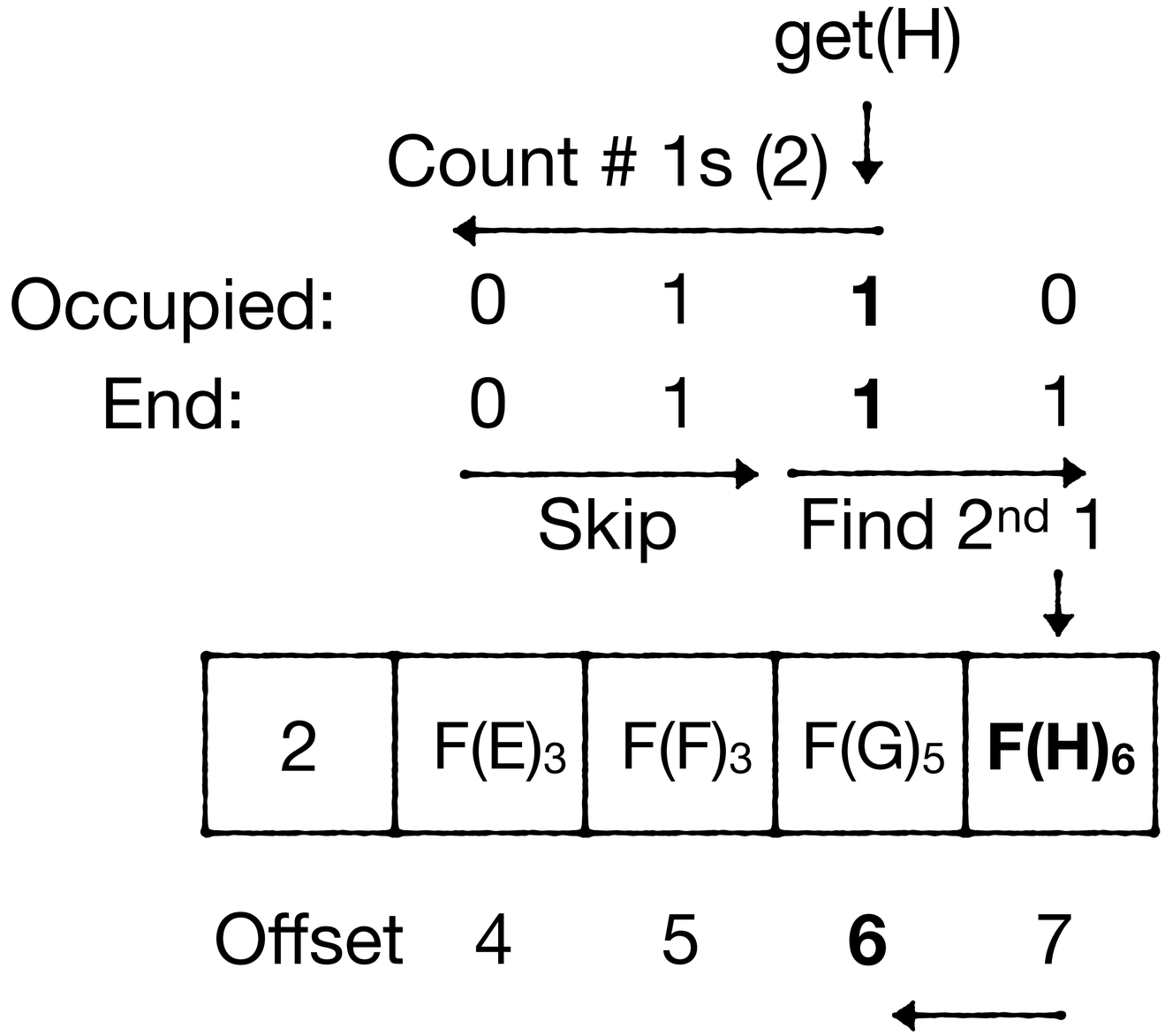
2	$F(E)_3$	$F(F)_3$	$F(G)_5$	$F(H)_6$

Offset 4 5 6 7


get(H)
Count # 1s (2)
Occupied:
0
1
1
0
End:
0
1
1
1
Skip
Find 2nd 1

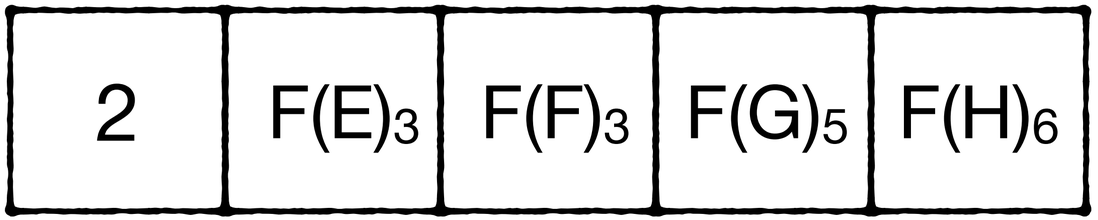

2
F(E)3
F(F)3
F(G)5
F(H)6
Offset
4
5
6
7


get(H)
Count # 1s (2)
Occupied: 0 1 1 0
End: 0 1 1 1
Skip
Find 2nd 1
2
F(E)3
F(F)3
F(G)5
F(H)6
Offset 4 5 6 7


get(H)
Count # 1s (2)
Occupied: 0 1 1 0
End: 0 1 1 1
Skip
Find 2nd 1
2
F(E)3
F(F)3
F(G)5
F(H)6
Offset 4 5 6 7
Scan until next run end or canonical slot

Target run may have been pushed to next chunk

Occupied:	0	1	1	0
End:	0	1	1	1

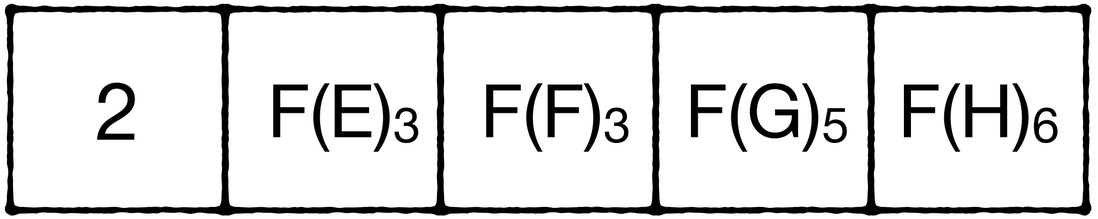


Offset	4	5	6	7

Target run may have been pushed to next chunk

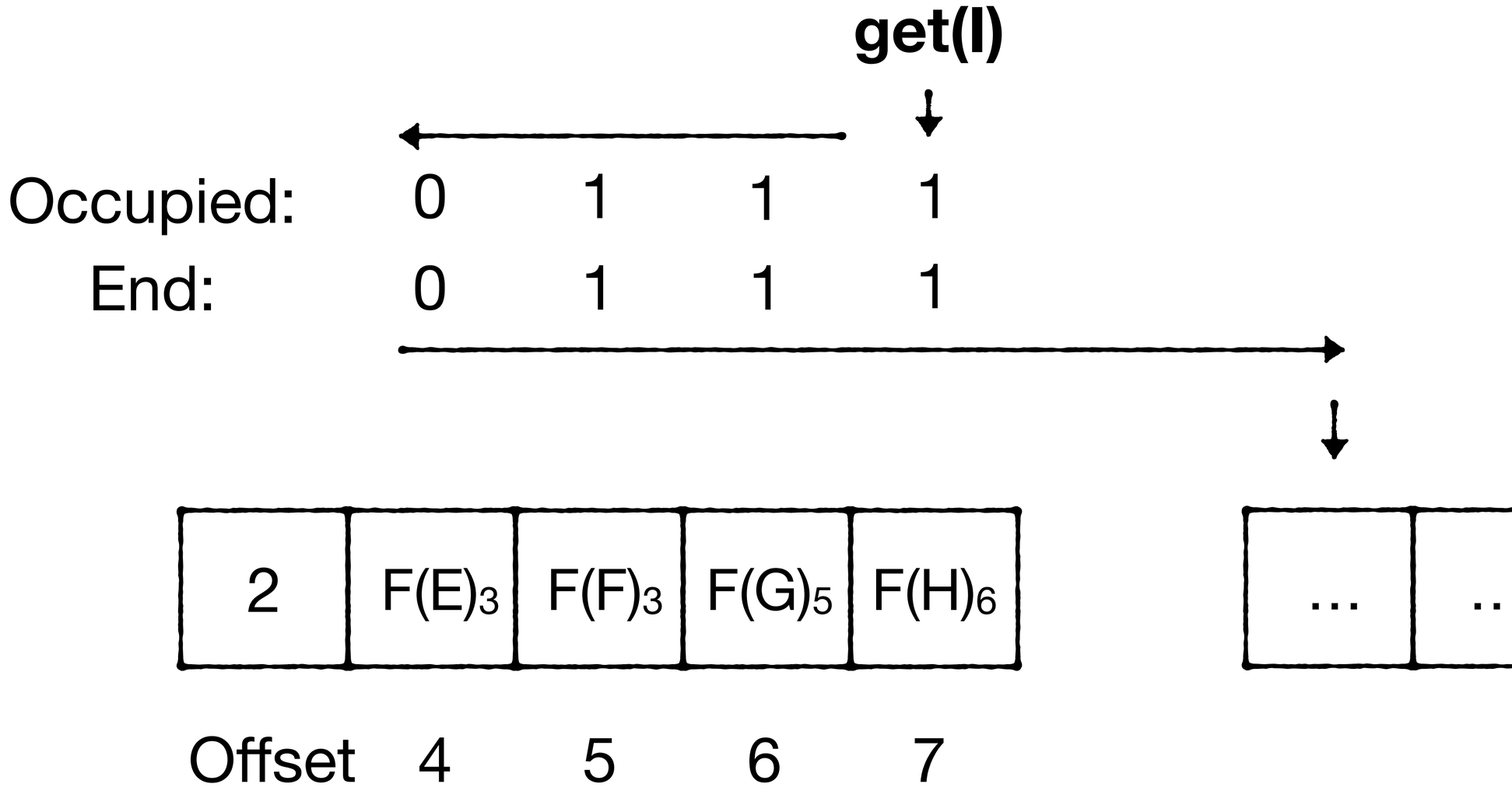
get(l)

Occupied:	0	1	1	1
End:	0	1	1	1

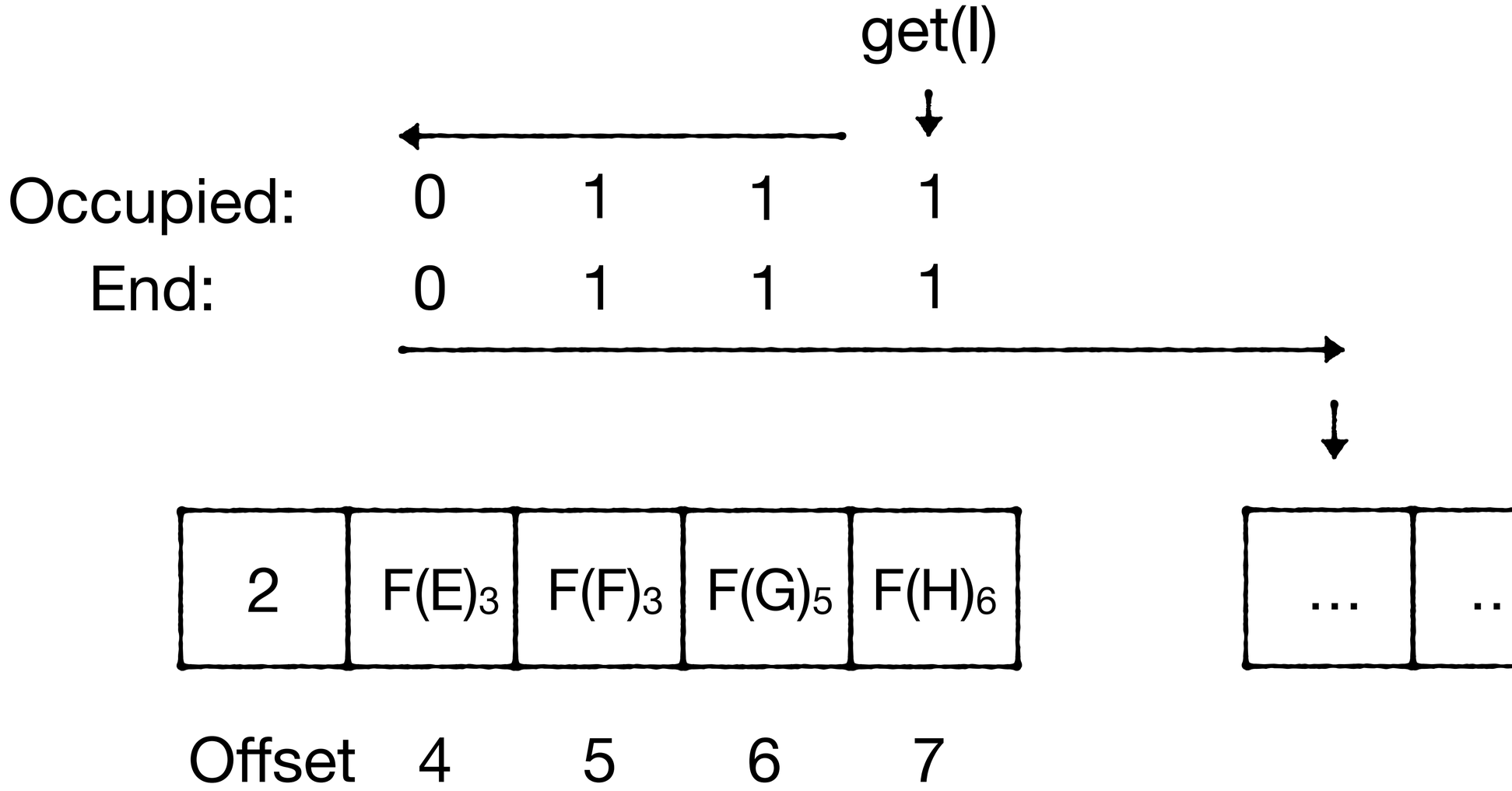


Offset	4	5	6	7

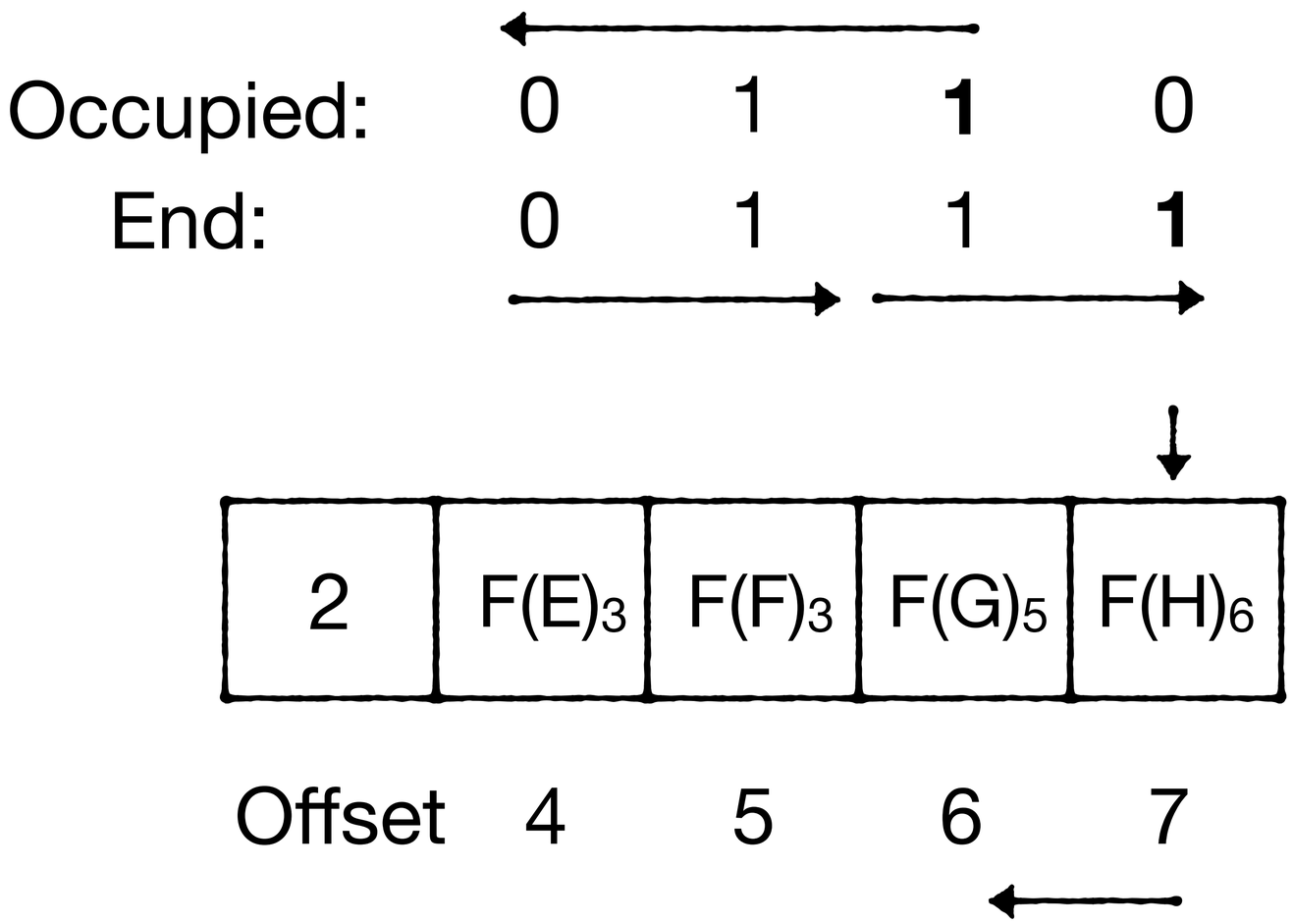
Target run may have been pushed to next chunk



Sequential cache misses

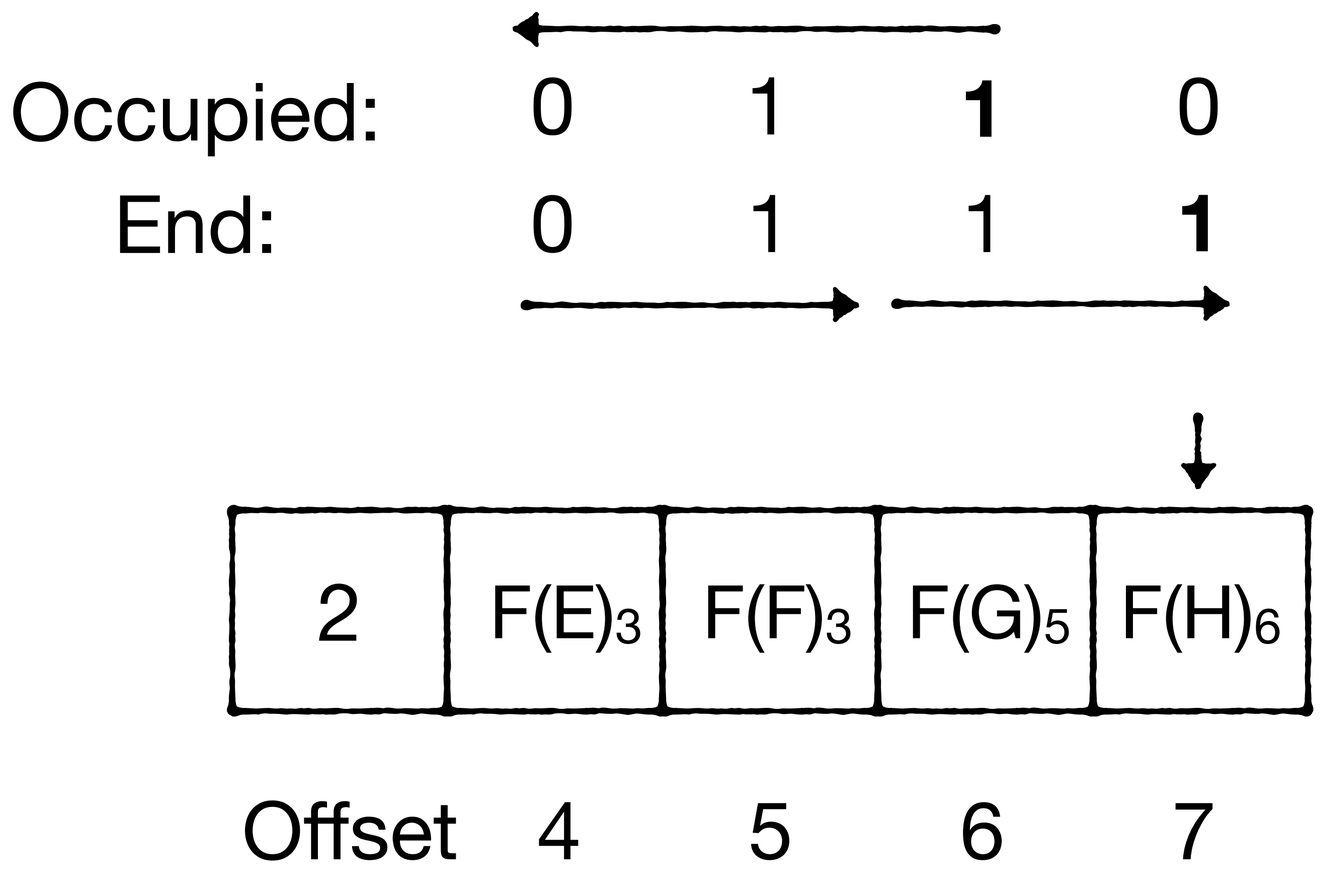


Queries in O(C), where C=64 is chunk size

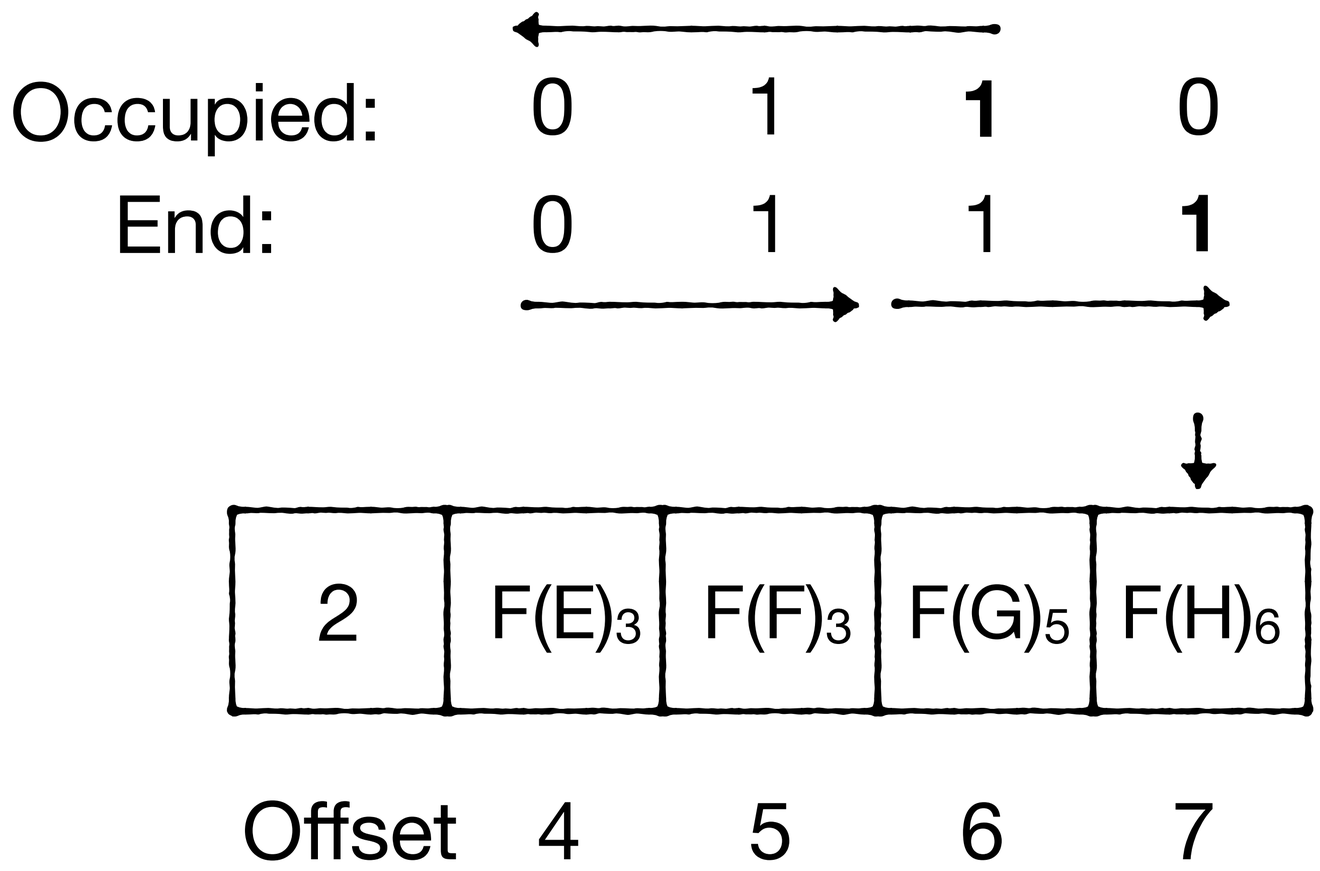


Queries in O(C), where C=64 is chunk size

Can we do O(1)?



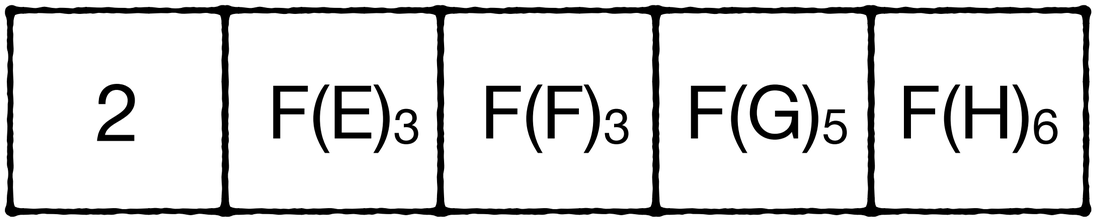
Rank & Select



Rank & Select

Can parse a 64-bit bitmap in constant time

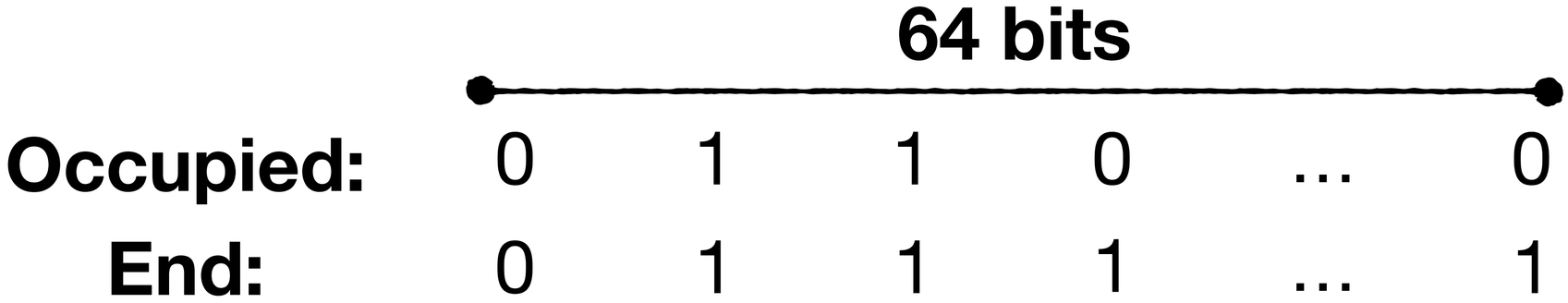
Occupied:	0	1	1	0
End:	0	1	1	1

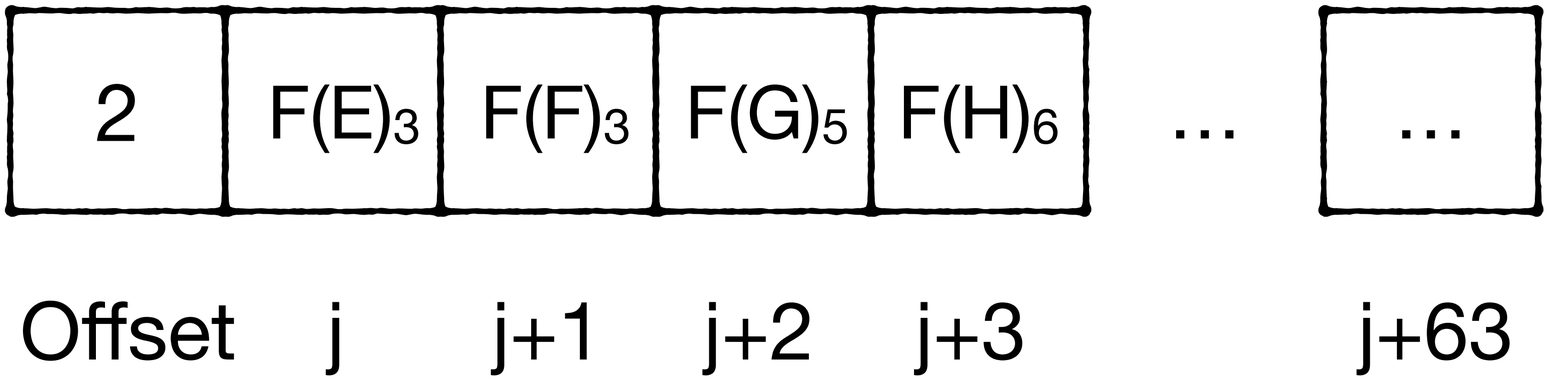


Offset	4	5	6	7

Rank & Select

Can parse a 64-bit bitmap in constant time

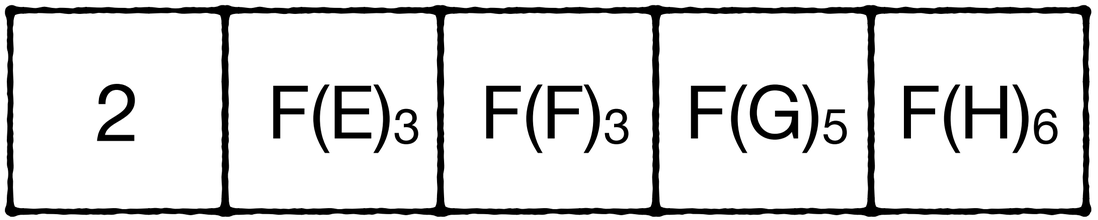




Rank(i) counts # 1s before the i^{th} bit

Select(i)

Occupied:	0	1	1	0
End:	0	1	1	1

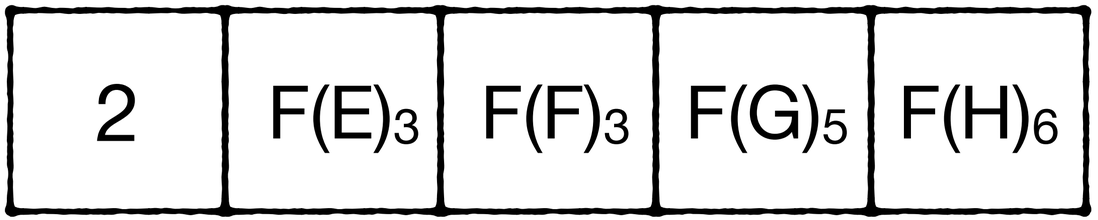


Offset	4	5	6	7

Rank (i) counts # 1s before the i^{th} bit

Select(i) returns the offset of the i^{th} 1

Occupied:	0	1	1	0
End:	0	1	1	1



Offset	4	5	6	7

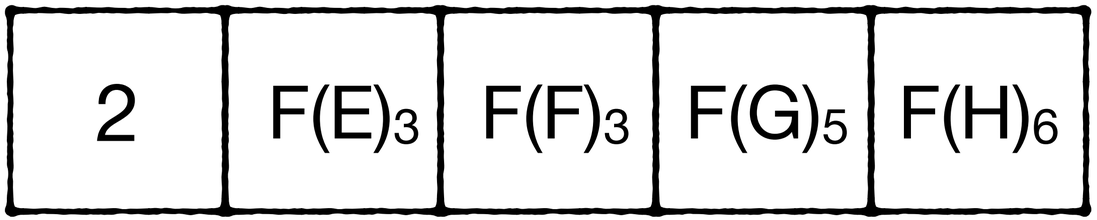
Rank (i) counts # 1s before the i^{th} bit

Select (i) returns the offset of the i^{th} 1

(1) How to use?

(2) How to implement?

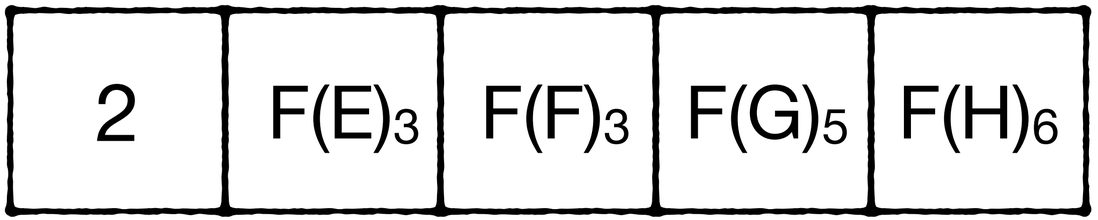
Occupied:	0	1	1	0
End:	0	1	1	1



Offset	4	5	6	7

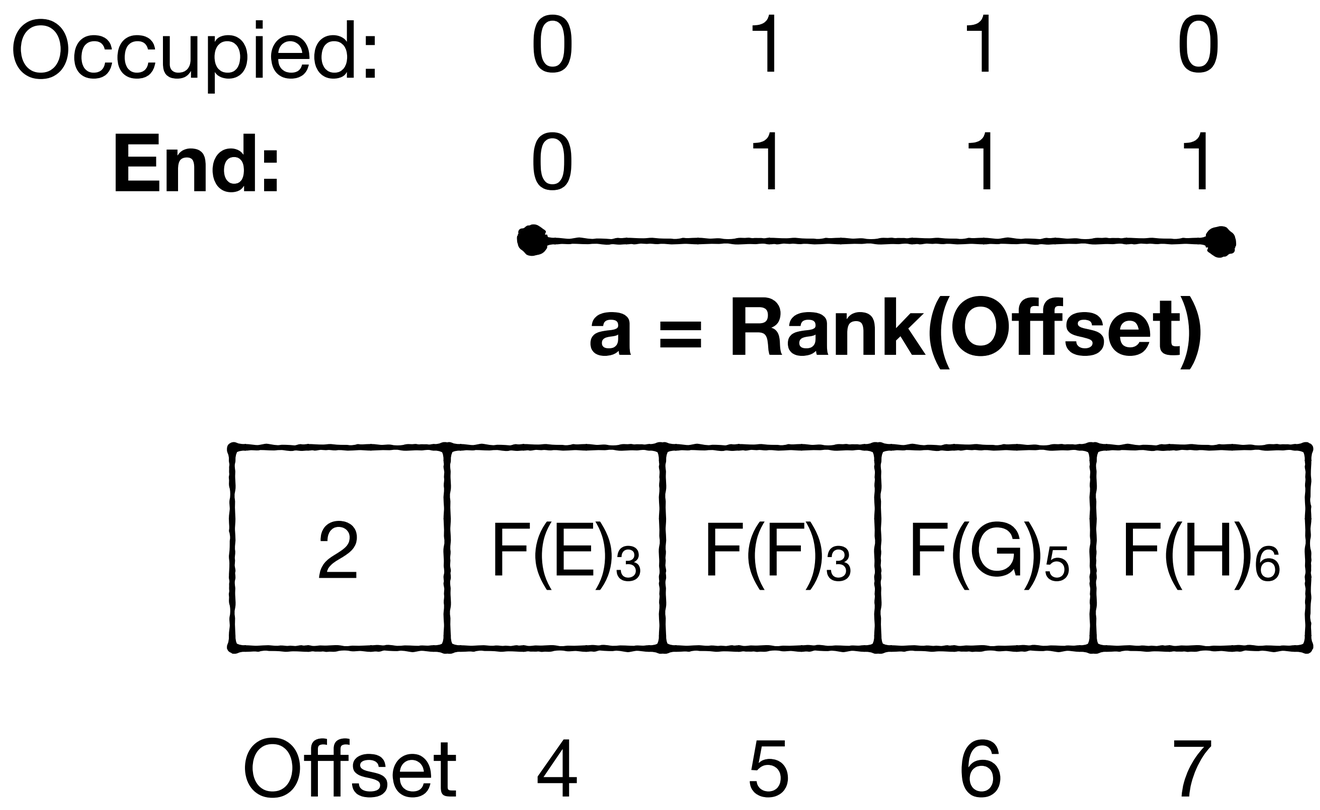
Back to example: get(H)

Occupied:	0	1	1	0
End:	0	1	1	1

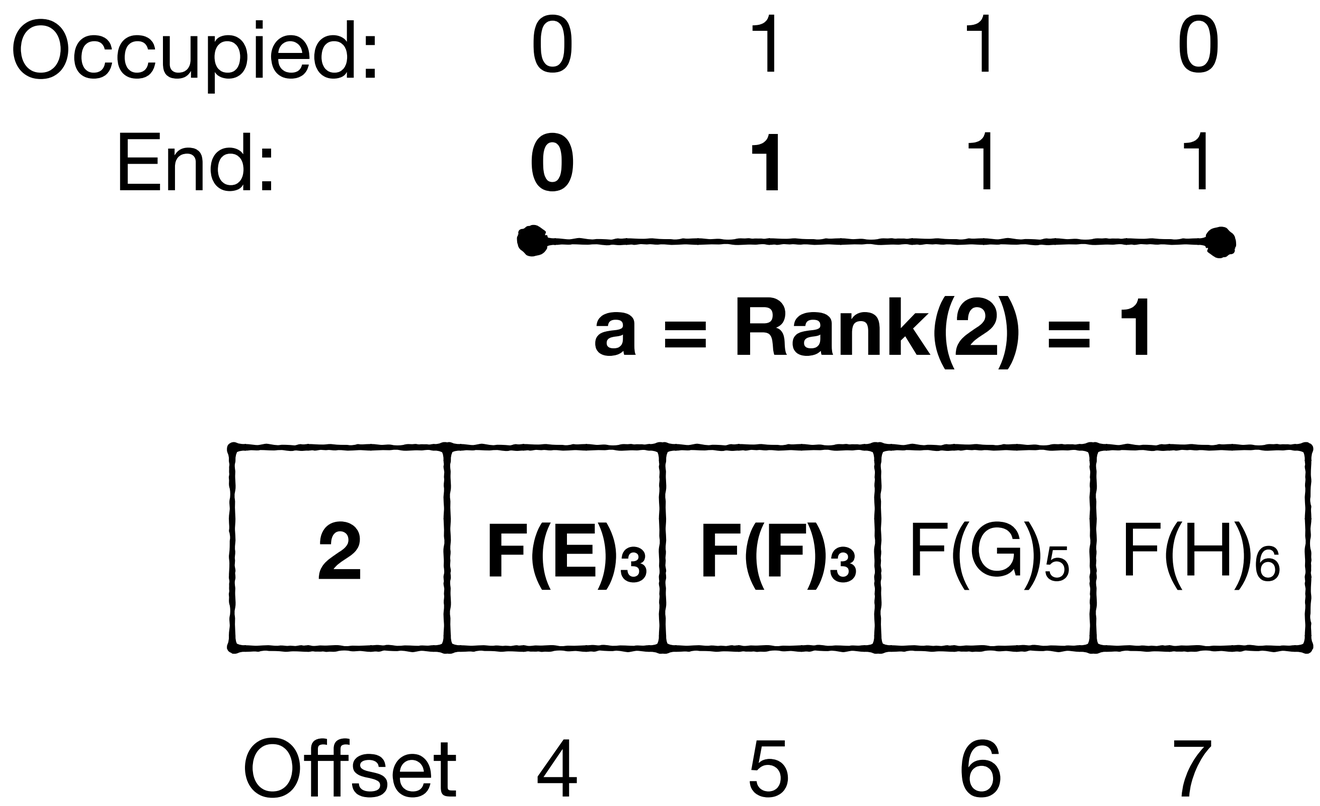


Offset	4	5	6	7

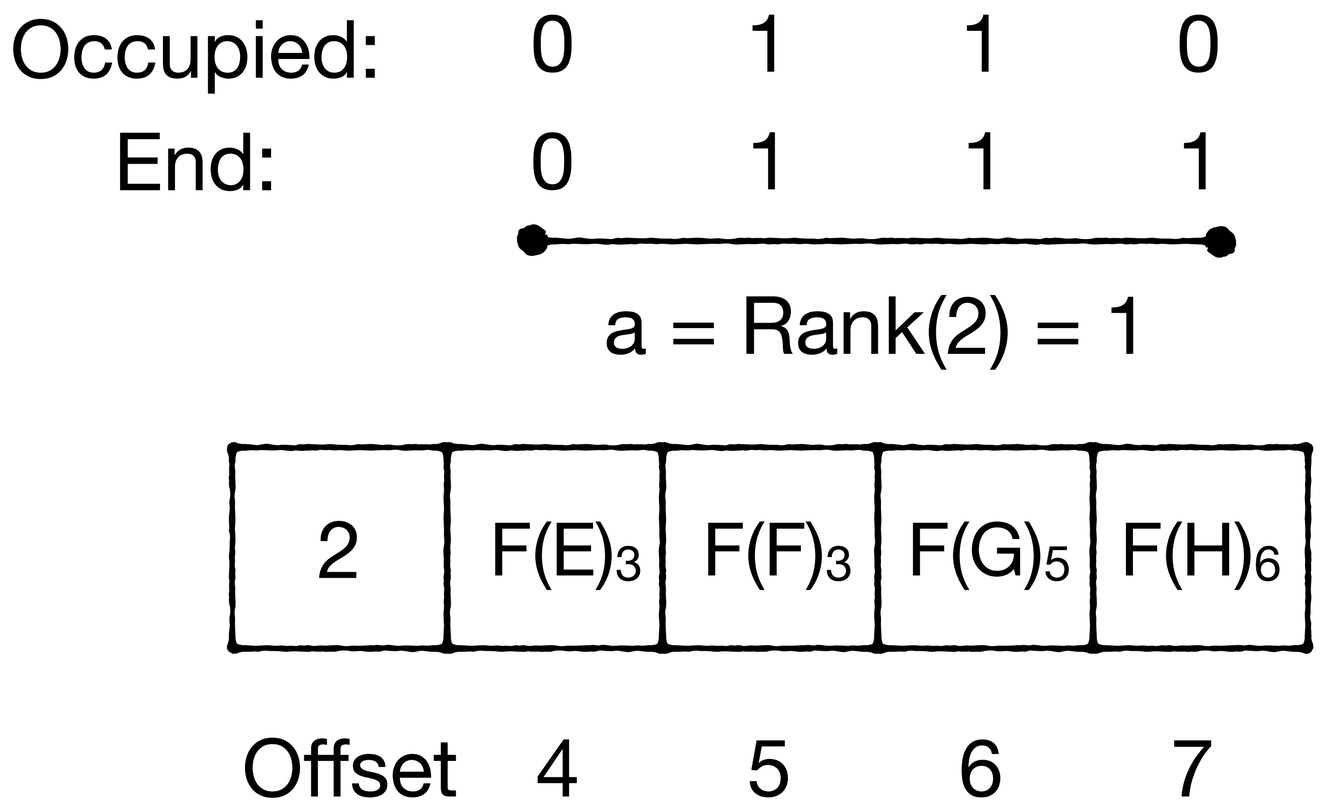
(A) Count # of runs ends belonging to previous chunks



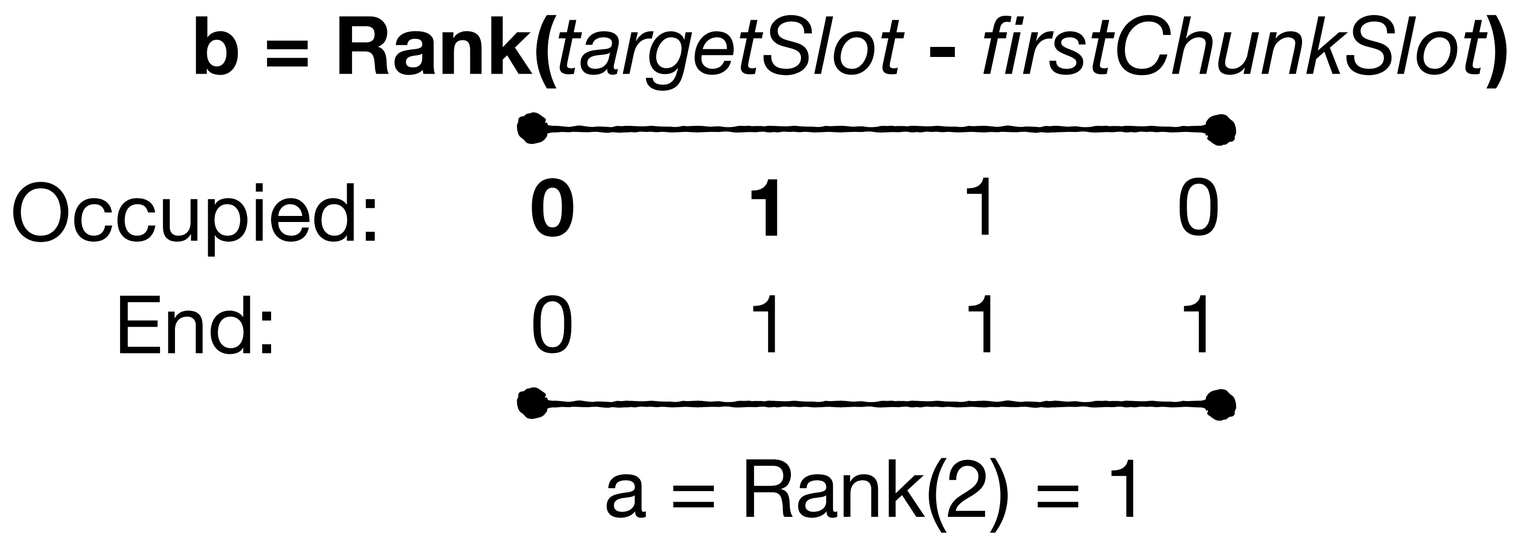
(A) Count # of runs ends belonging to previous chunks



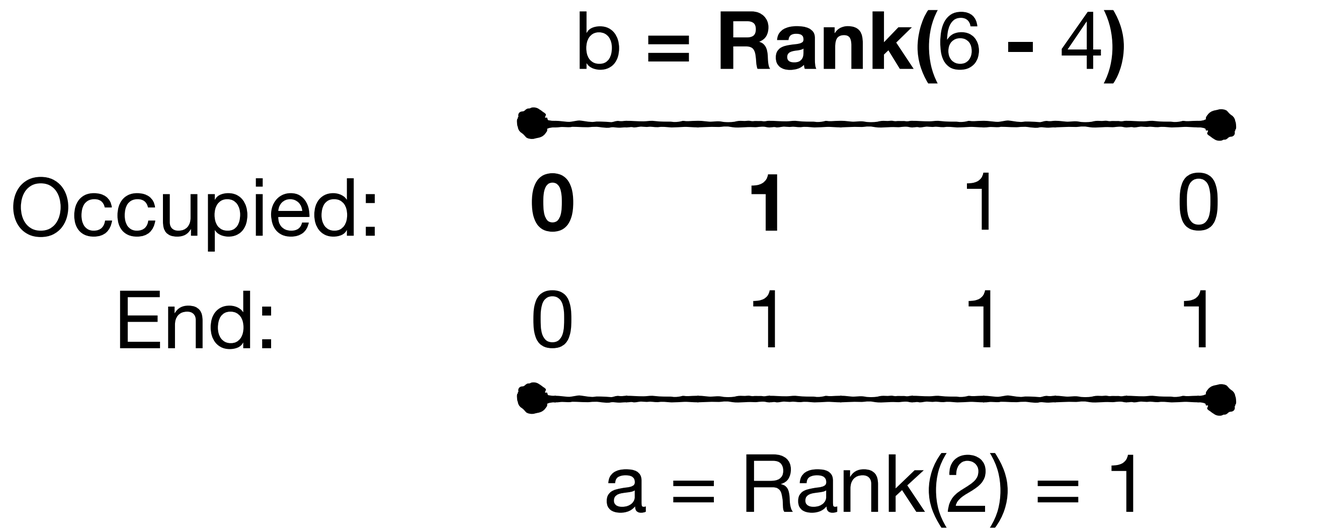
(B) Count # of run ends belonging to this chunk before target



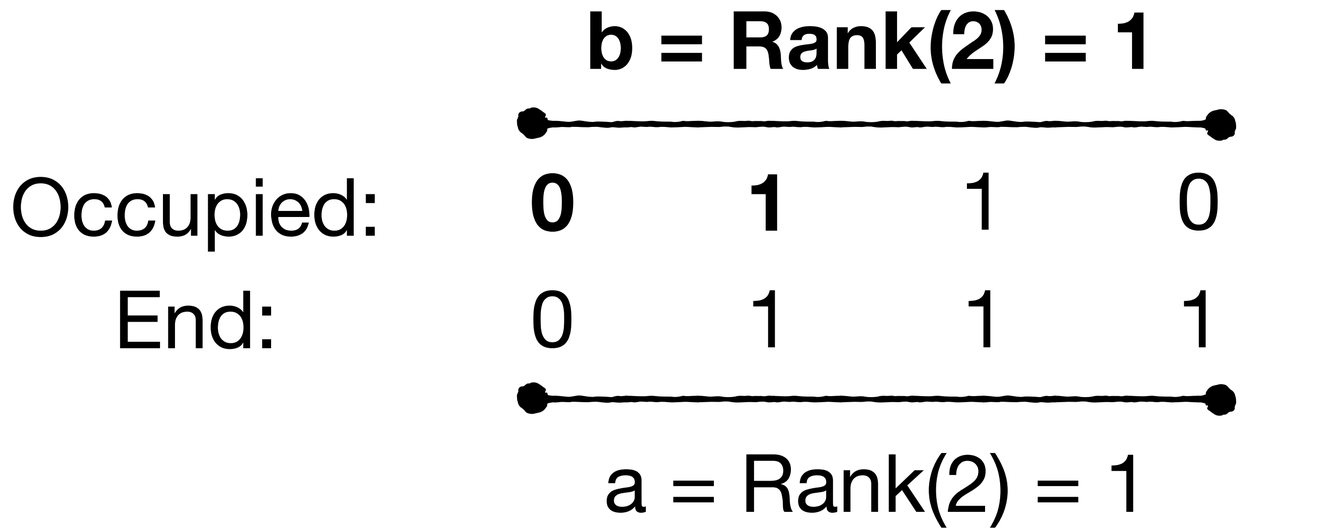
(B) Count # of run ends belonging to this chunk before target



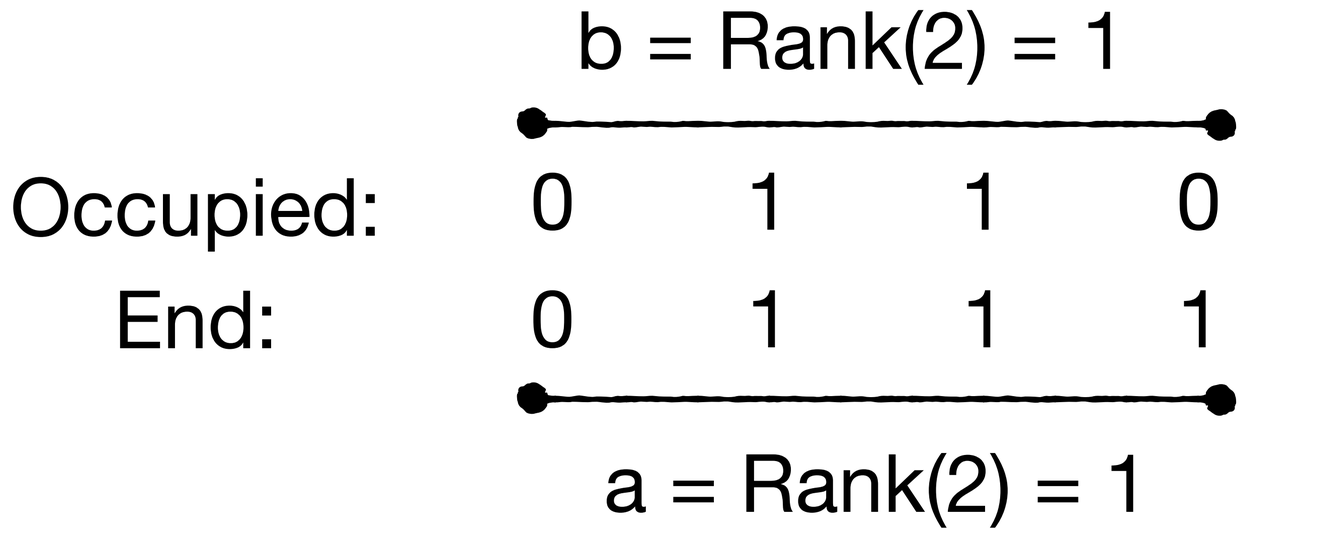
(B) Count # of run ends belonging to this chunk before target



(B) Count # of run ends belonging to this chunk before target

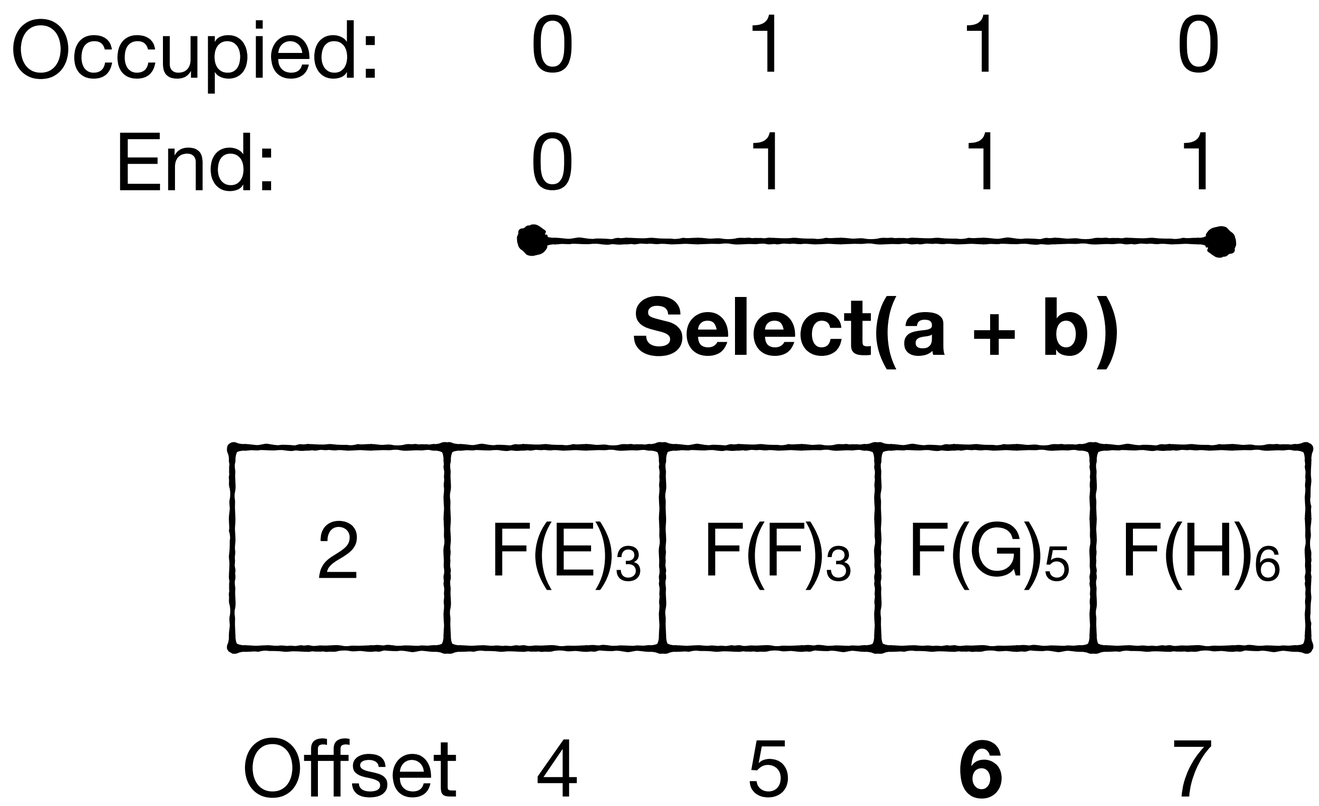


(C) skip to the (a+b)th run end



2	$F(E)_3$	$F(F)_3$	$F(G)_5$	$F(H)_6$
Offset	4	5	6	7

(C) skip to the $(a+b)^{th}$ run end

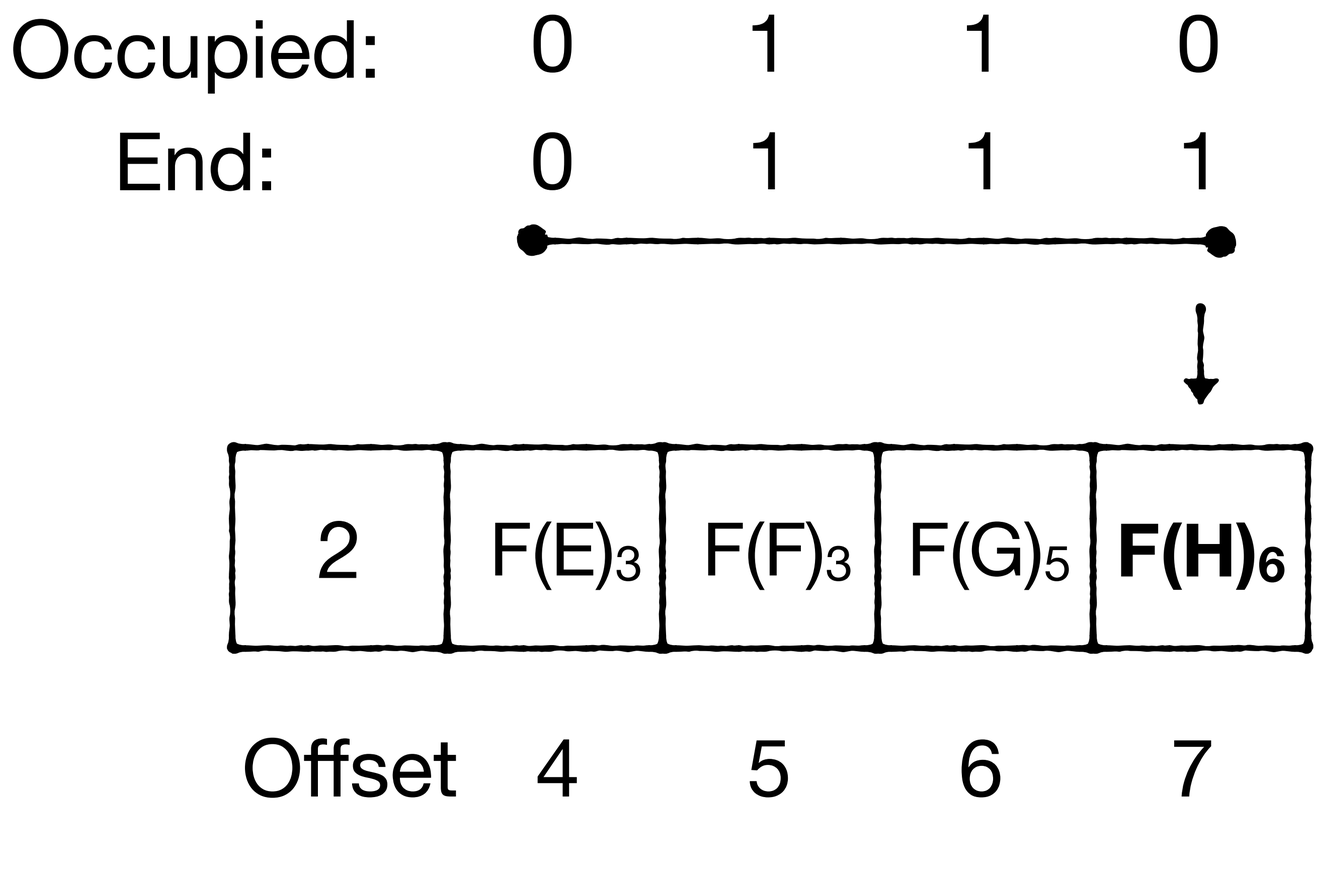


(C) skip to the $(a+b)^{th}$ run end

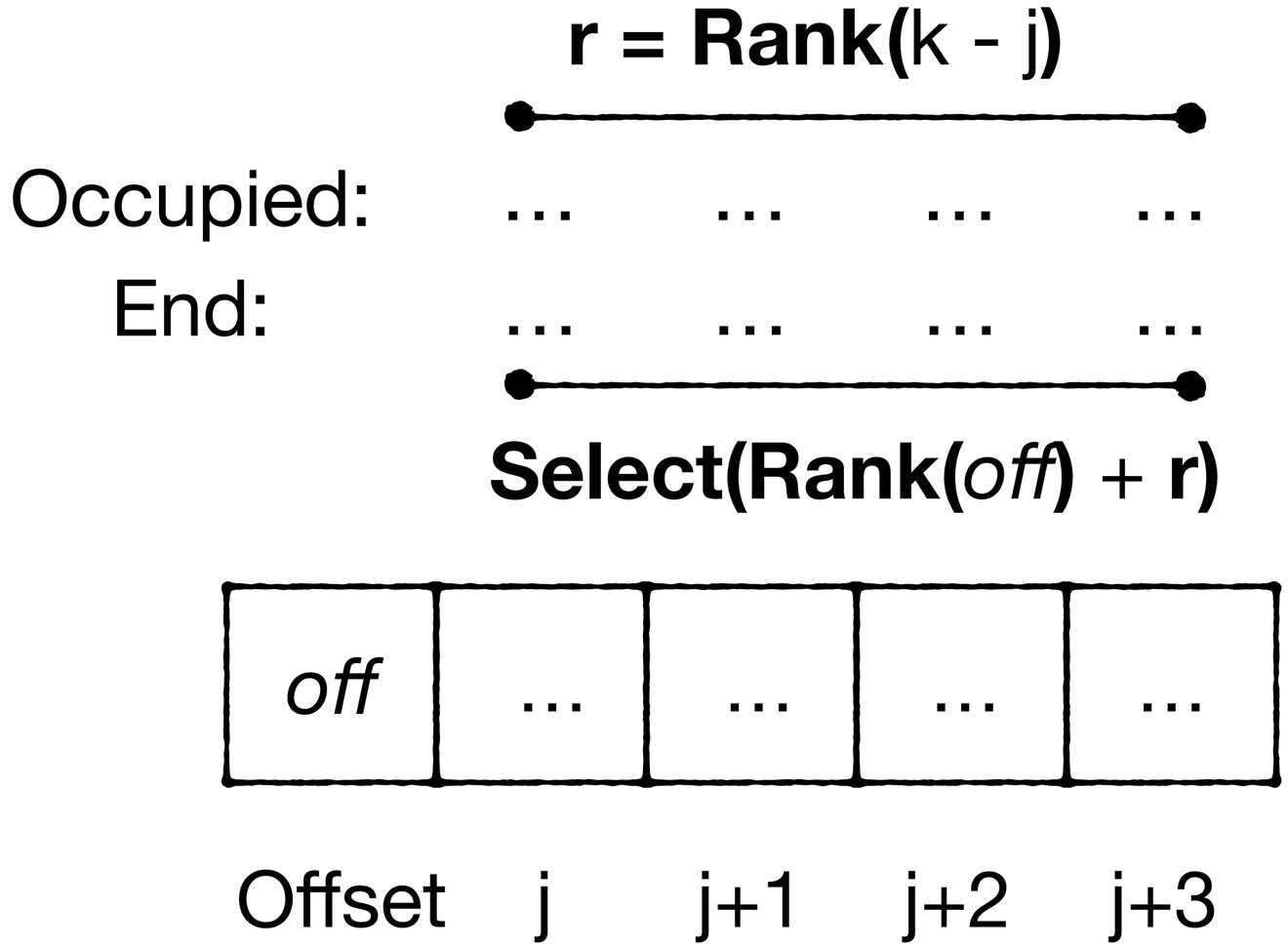
2	$F(E)_3$	$F(F)_3$	$F(G)_5$	$F(H)_6$

Offset 4 5 **6** 7

(C) skip to the $(a+b)^{th}$ run end



General algorithm to bring us to end of slot k's run



Implementing Rank and Select Efficiently

Implementing Rank and Select Efficiently

No looping

Implementing Rank Efficiently

$rank(i) = popcount(B \,\&\, (2^i - 1))$

Implementing Rank Efficiently

$$rank(i) = popcount(\mathbf{B} \;\&\; (2^i - 1))$$

Bitmap (64 bits long)

Implementing Rank Efficiently

rank(i) = popcount(**B** & (2^i - 1))

Bitmap (64 bits long)

Least significant bits

Most significant bits

Implementing Rank Efficiently

$$\text{rank}(i) = \textbf{popcount}(B \,\&\, (2^i - 1))$$

↑

Total # of 1s

Implementing Rank Efficiently

rank(i) = popcount(B **& $(2^i - 1)$**)

Mask out irrelevant more significant bits

$$\text{rank}(i) = \text{popcount}(B \,\&\, (2^i - 1))$$

e.g., **B = 0 1 1 0 1 0 1 1**

rank(i) = popcount(B & (2^i - 1))

e.g., B = 0 **1 1** 0 **1** 0 1 1 **rank(6) = 3**

$$\text{rank}(i) = \text{popcount}(B \,\&\, (2^i - 1))$$

e.g., B = 0 1 1 0 1 0 1 1 **rank(6) = 3**

mask: $2^6 - 1$ = **1 1 1 1 1 1** 0 0

$$\text{rank}(i) = \text{popcount}(B \,\&\, (2^i - 1))$$

e.g., B = 0 1 1 0 1 0 1 1 **rank(6) = 3**

&

1 1 1 1 1 1 0 0

=

0 1 1 0 1 0 0 0

$$\text{rank}(i) = \text{popcount}(B \,\&\, (2^i - 1))$$

e.g., B = 0 1 1 0 1 0 1 1 **rank(6) = 3**

popcount(0 **1 1** 0 **1** 0 0 0 **) = 3**

Implementing Select Efficiently

Implementing Select Efficiently

select(i) = tzcnt(pdep(2^i, B))

Implementing Select Efficiently

$$select(i) = tzcnt(pdep(2^i, \mathbf{B}))$$

Bitmap (64 bits long)

Implementing Select Efficiently

$$\text{select}(i) = \text{tzcnt}(\text{pdep}(2^i, \mathbf{B}))$$

↑

Count trailing zeros

Implementing Select Efficiently

select(i) = tzcnt(pdep(2^i, **B**))

Count trailing zeros

tzcnt(*000*11101) = 3

Implementing Select Efficiently

select(i) = tzcnt(**pdep**(2^i, B))

Scatter bits in first operand at 1s in second operand

Implementing Select Efficiently

select(i) = **tzcnt**(**pdep**(2^i, B))

Available on x86

https://www.felixcloutier.com/x86/

Implementing Select Efficiently

$$\text{select}(i) = \text{tzcnt}(\text{pdep}(2^i, B))$$

e.g., **B = 0 1 1 0 1 0 1 1**

Implementing Select Efficiently

$$\text{select}(i) = \text{tzcnt}(\text{pdep}(2^i, B))$$

e.g., B = 0 1 1 0 **1** 0 1 1 **Select(2) = 4**

Implementing Select Efficiently

select(i) = tzcnt(pdep($\mathbf{2^i}$, B))

e.g., B = 0 1 1 0 1 0 1 1 Select(2) = 4

$\mathbf{2^2}$ = 0 0 **1** 0 0 0 0 0

Implementing Select Efficiently

select(i) = tzcnt(pdep(2^i, B))

e.g., B = 0 1 1 0 1 0 1 1 Select(2) = 4

pdep(0 0 **1** 0 0 0 0 0 , B) = 0 0 0 0 **1** 0 0 0

Scatter bits in first operand at 1s in second operand

Implementing Select Efficiently

$$\text{select}(i) = \text{tzcnt}(\text{pdep}(2^i, B))$$

e.g., B = 0 1 1 0 1 0 1 1 Select(2) = 4

pdep(0 0 1 0 0 0 0 0 , B) = 0 0 0 0 **1** 0 0 0

Only the 1 at relevant position is now set

Implementing Select Efficiently

$$\text{select}(i) = \text{tzcnt}(\text{pdep}(2^i, B))$$

e.g., B = 0 1 1 0 1 0 1 1 Select(2) = 4

tzcnt(0 0 0 0 1 0 0 0) = **4**

queries in O(1) due to fast rank and select

r = Rank(k - j)

Occupied: … … … …

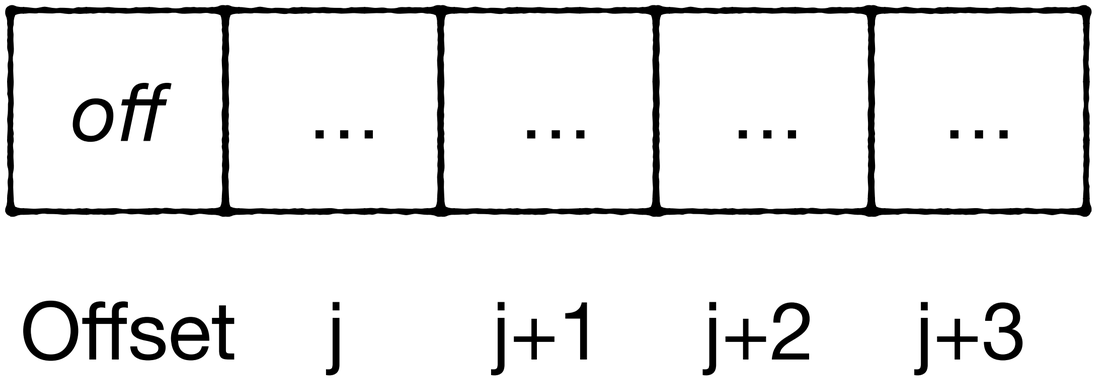
End: … … … …

Select(Rank(*off*) + **r**)

off	…	…	…	…

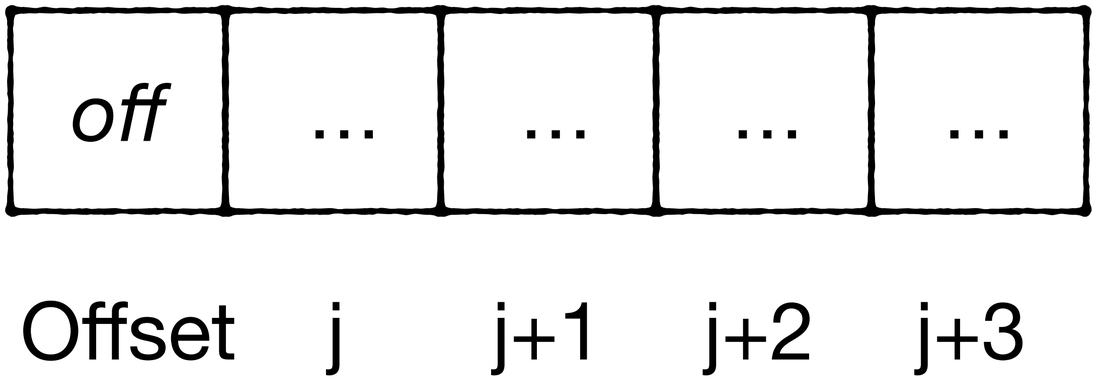
Offset j j+1 j+2 j+3

Insertions?



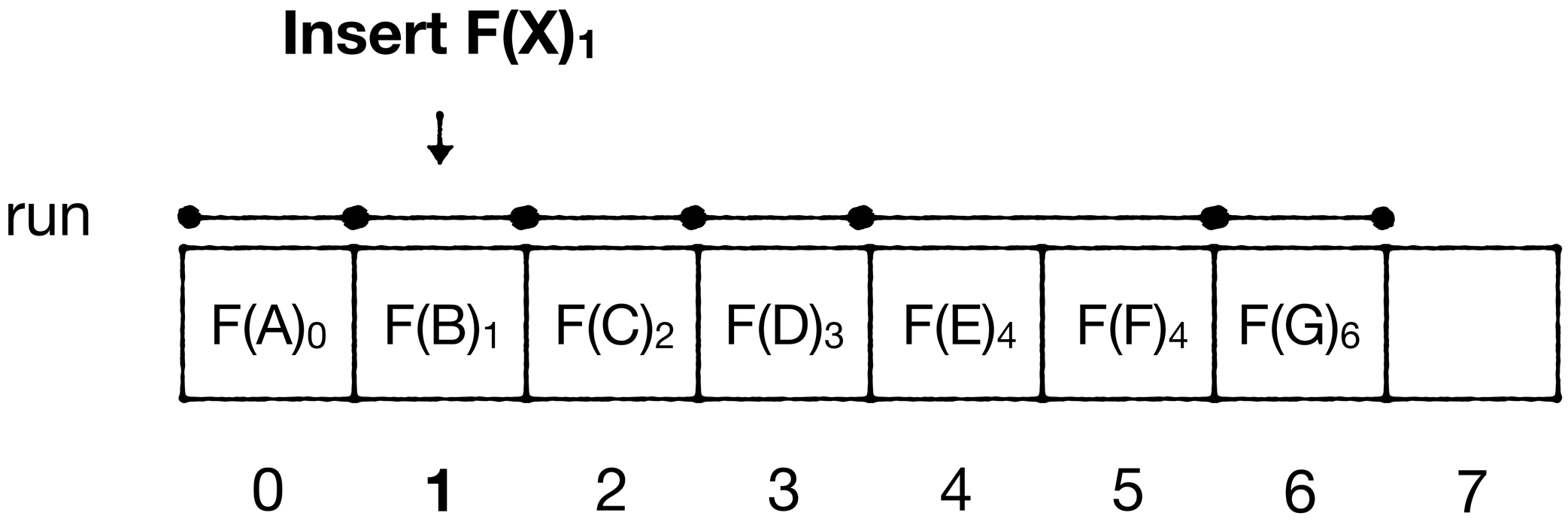
Insertions

Find target run, push colliding entries to right, insert



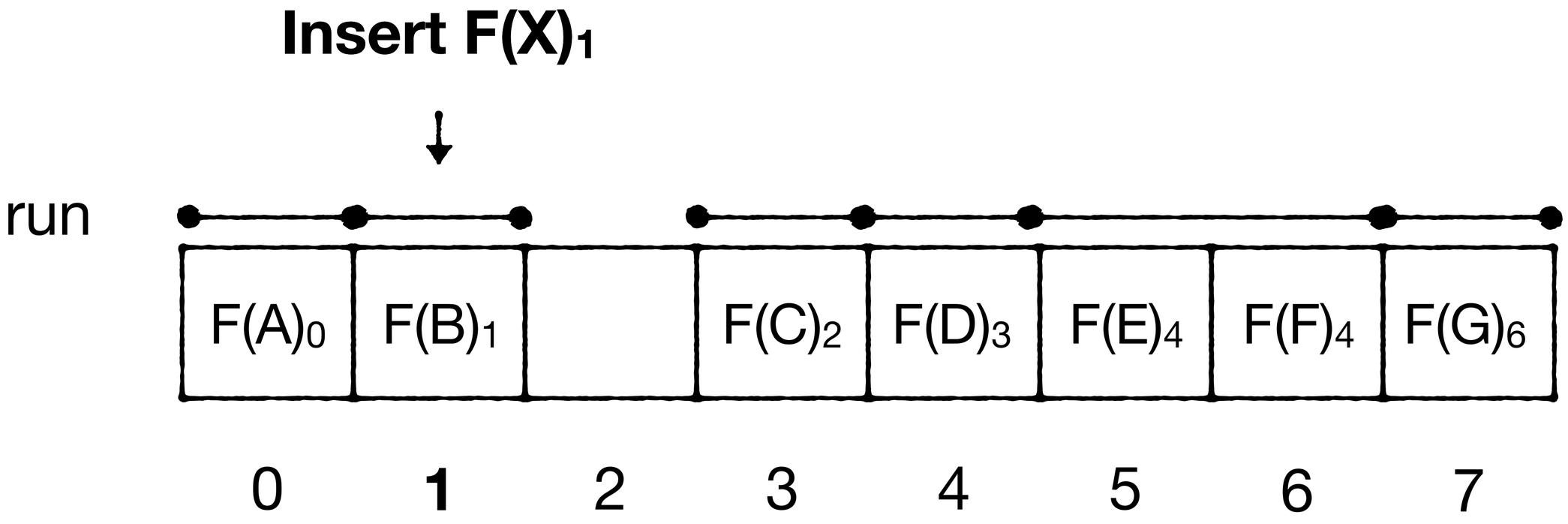
Insertions

Find target run, push colliding entries to right, insert



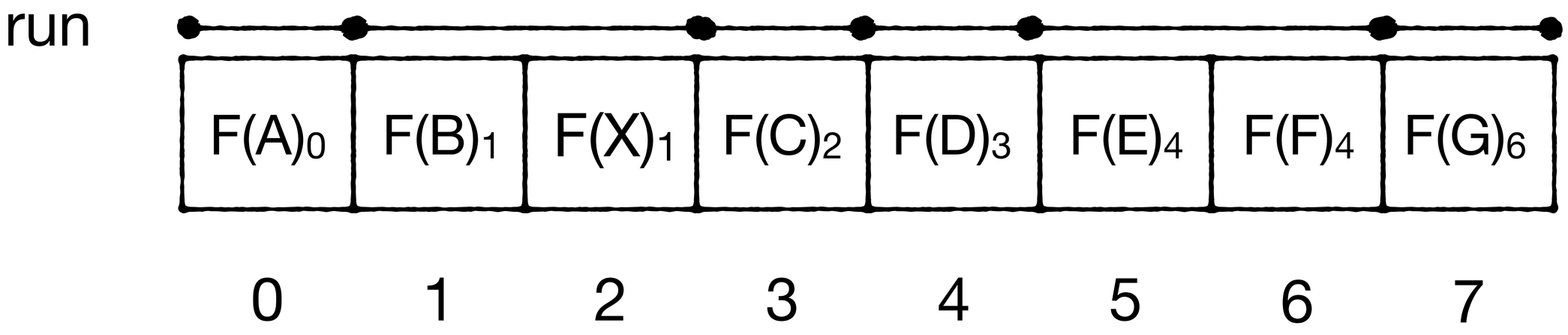
Insertions

Find target run, push colliding entries to right, insert



Insertions

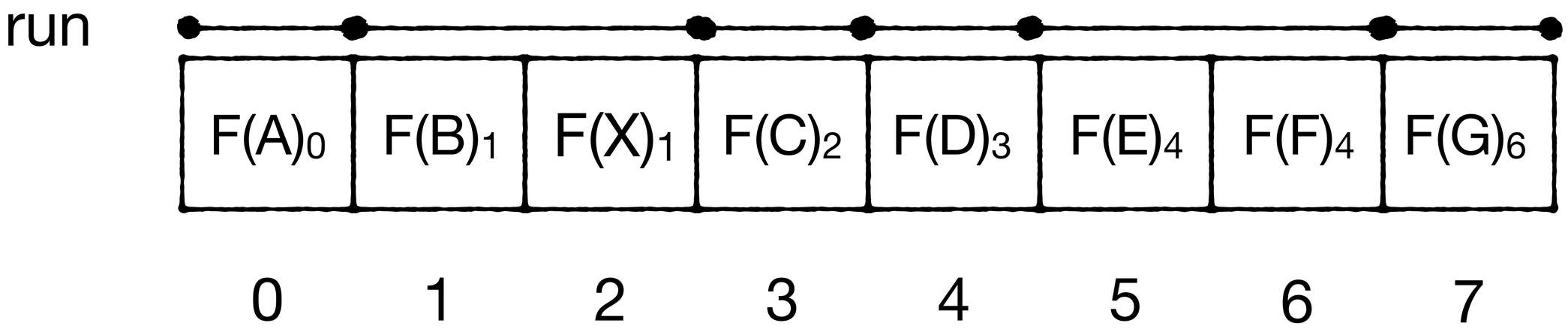
Find target run, push colliding entries to right, insert



Insertions

Find target run, push colliding entries to right, insert

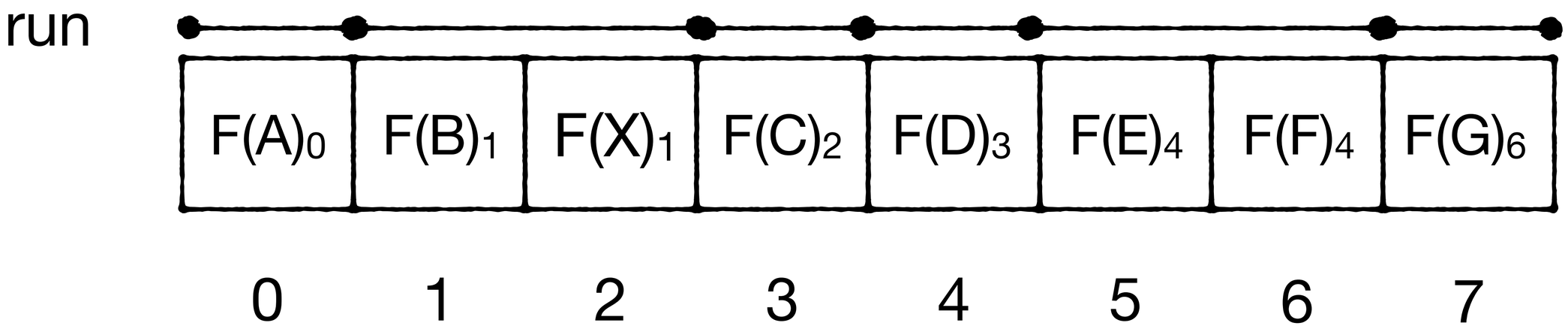
Problem?



Insertions

Find target run, **push colliding entries to right**, insert

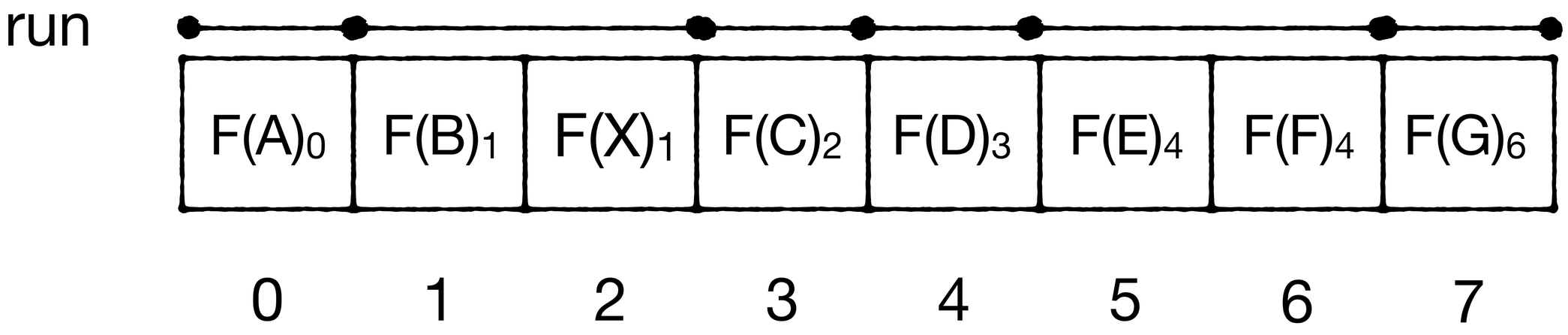
potentially O(N)



Insertions

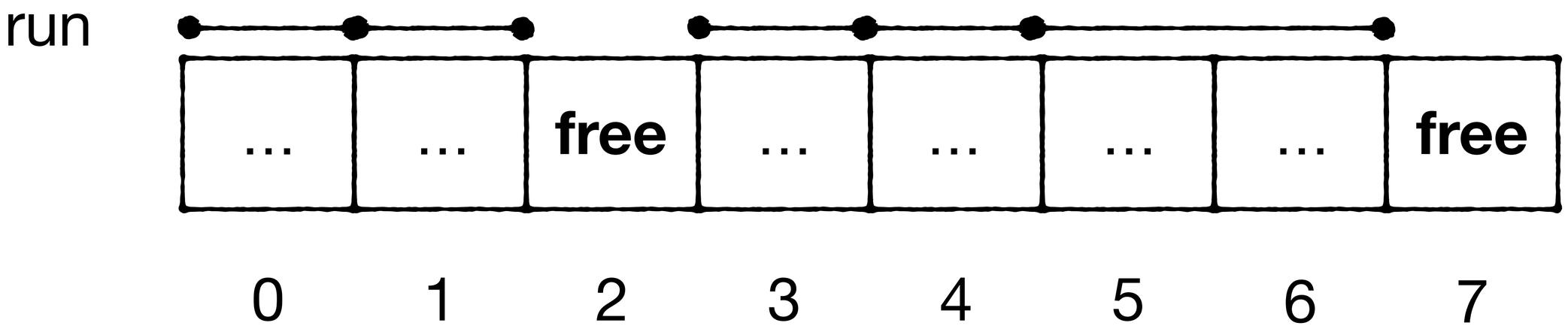
Find target run, **push colliding entries to right**, insert

potentially O(N) - **solution**?



Insertions

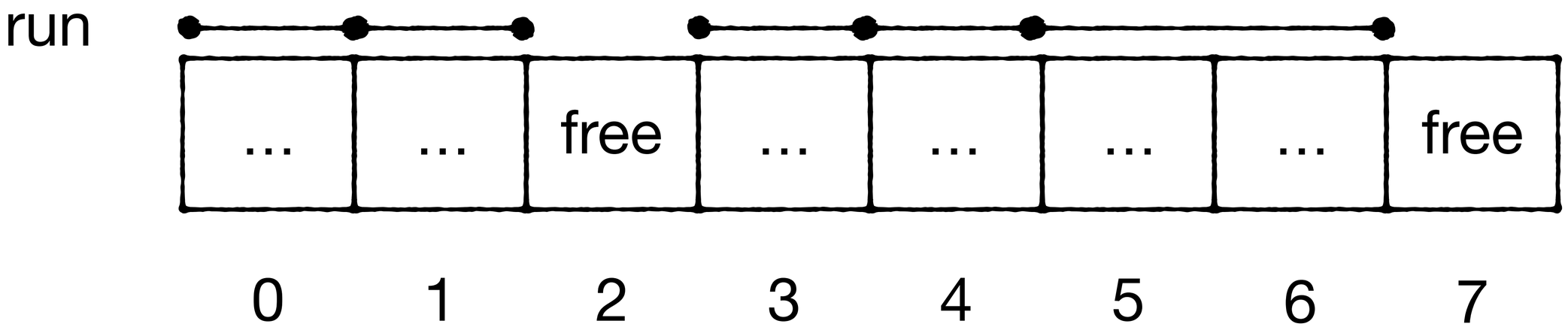
Keep at least 5% spare capacity



Insertions

Keep at least 5% spare capacity

Push at most 20 entries on avg due to hashing

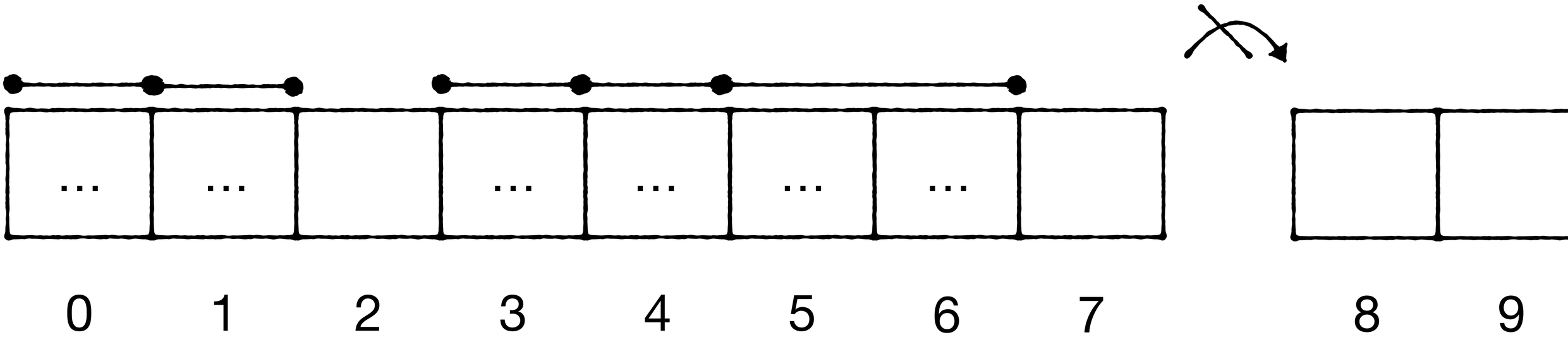


Insertions

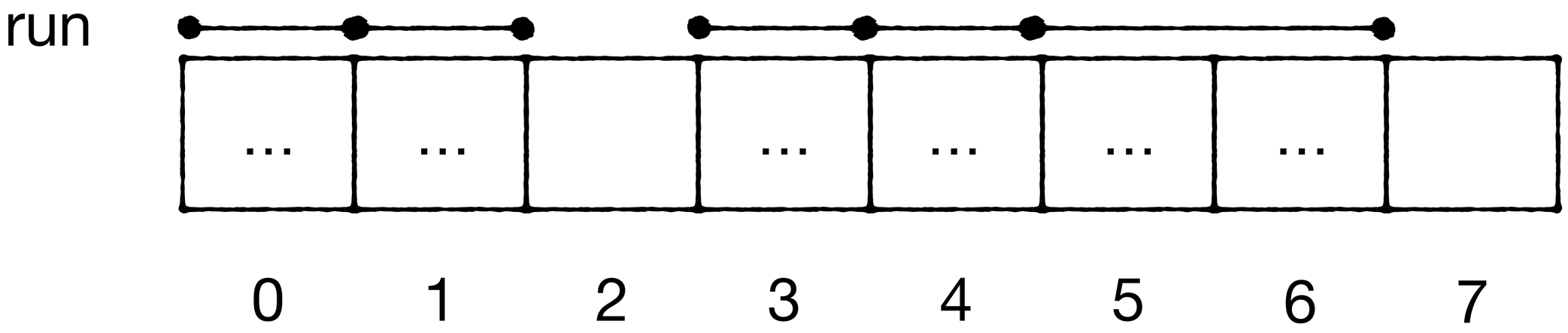
Keep at least 5% spare capacity

Push at most 20 entries on avg due to hashing

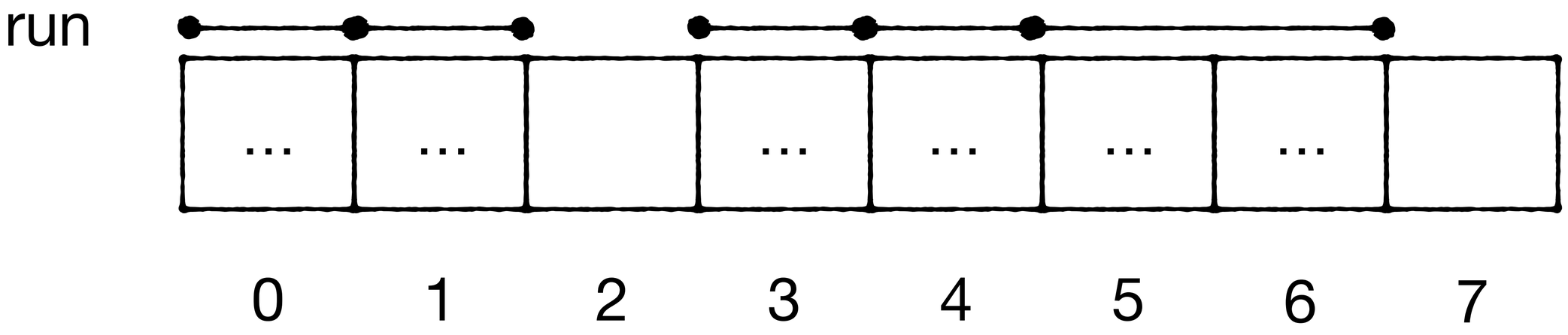
Most insertions don’t spill to the next chunk



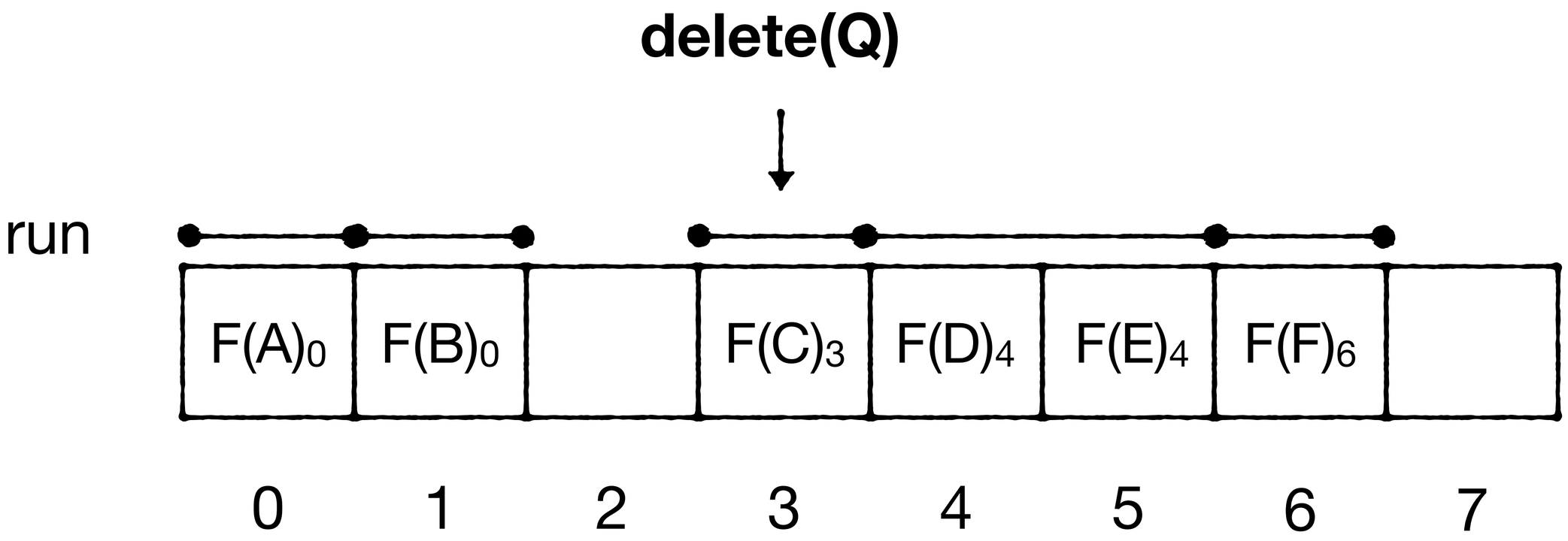
deletes?



Can only delete entry we know exists. Why?

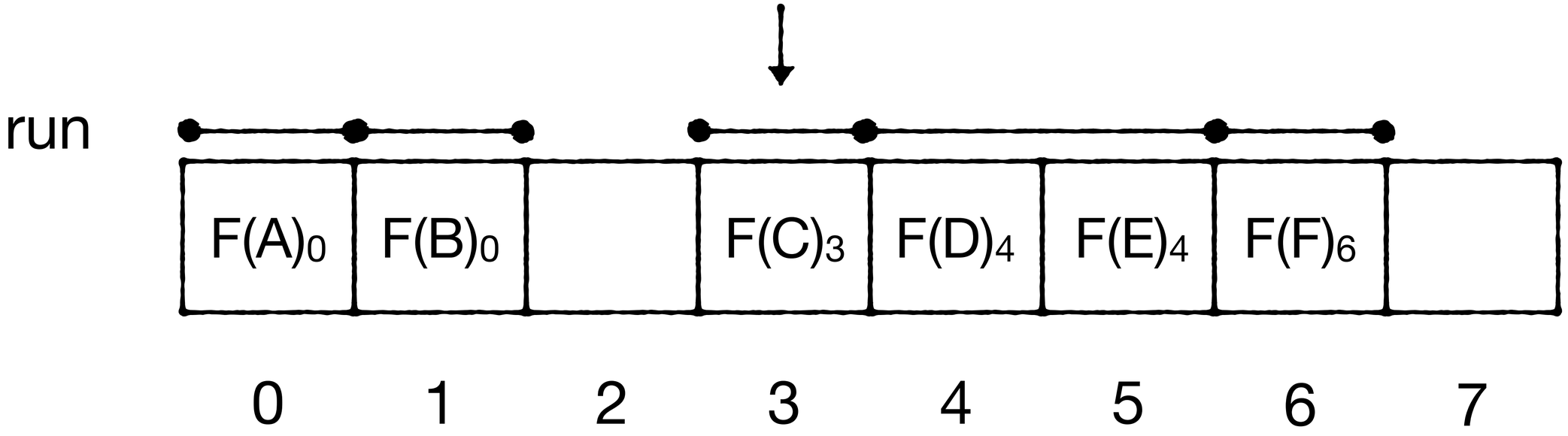


Can only delete entry we know exists. Why?



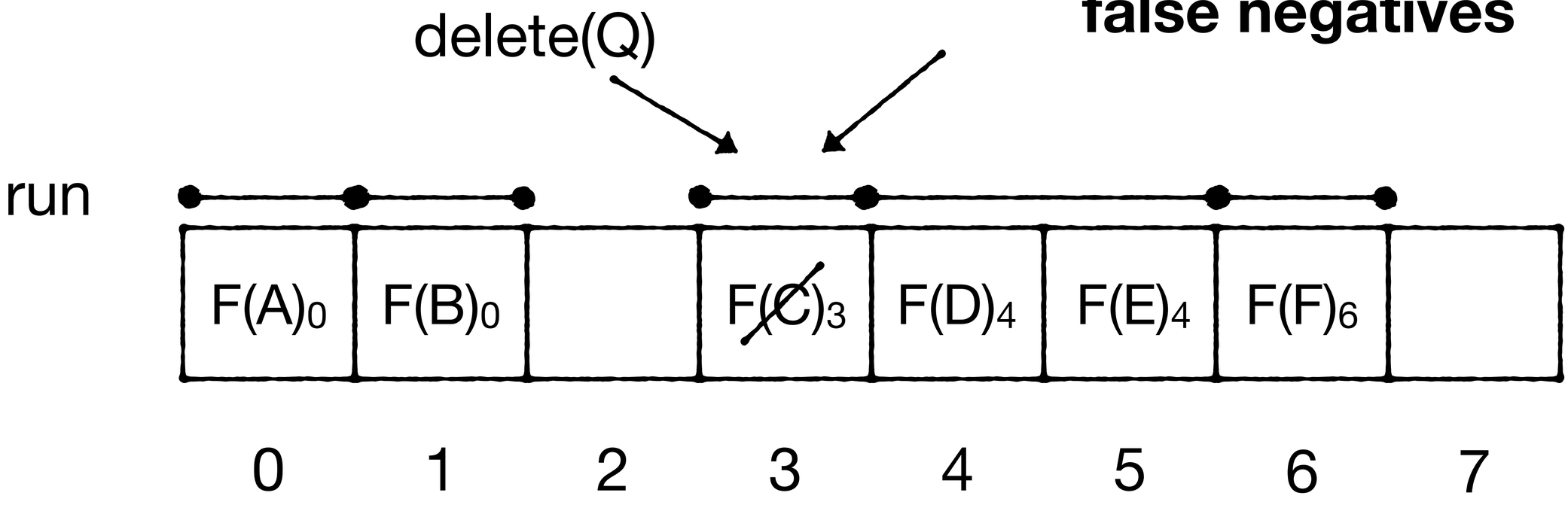
Can only delete entry we know exists. Why?

delete(Q) - matches C's FP at slot 3

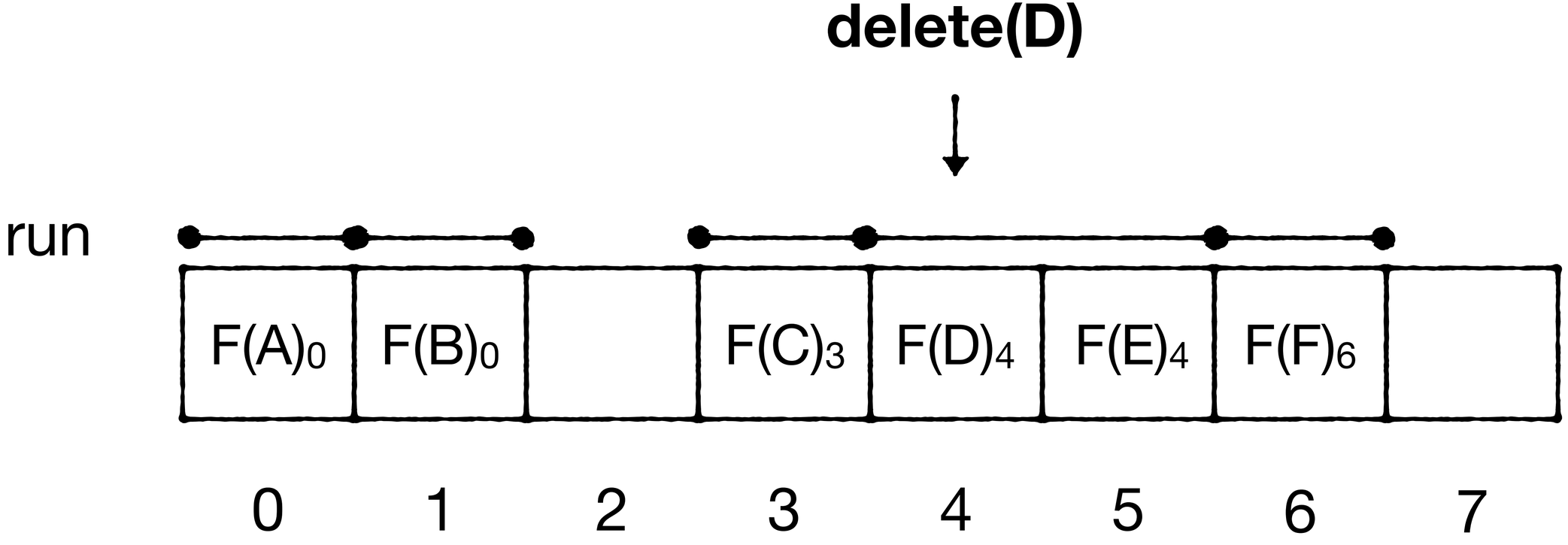


Can only delete entry we know exists. Why?

Subsequent get(C) return false negatives

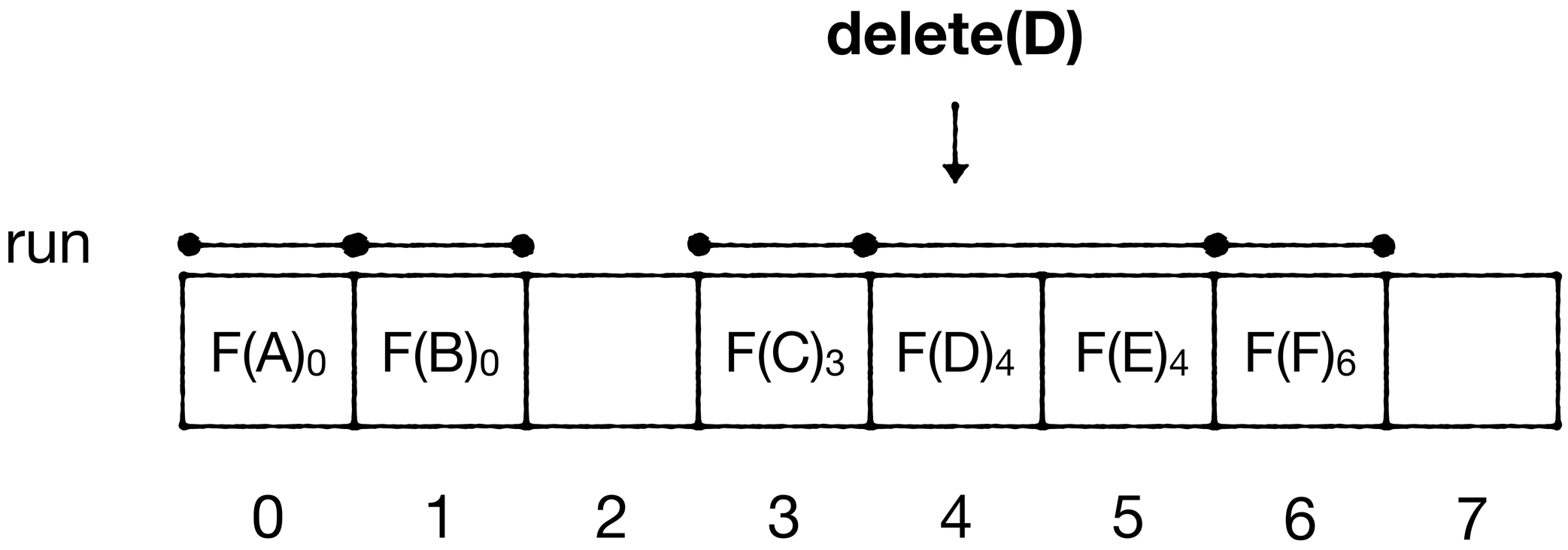


How to delete an entry we know exists?



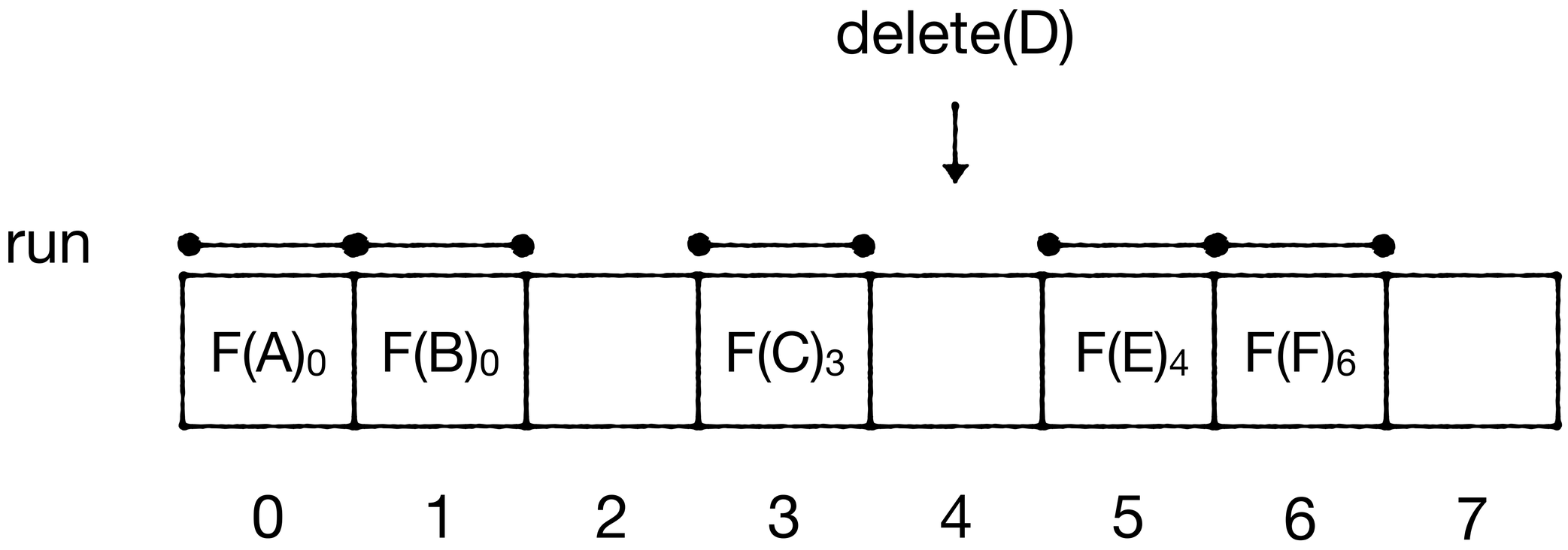
How to delete an entry we know exists?

(1) Find run, remove matching fingerprint

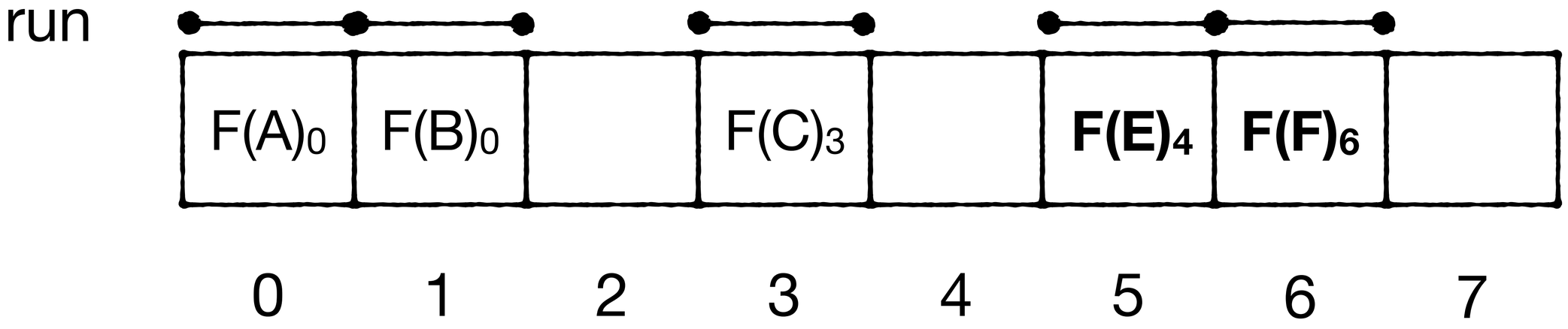


How to delete an entry we know exists?

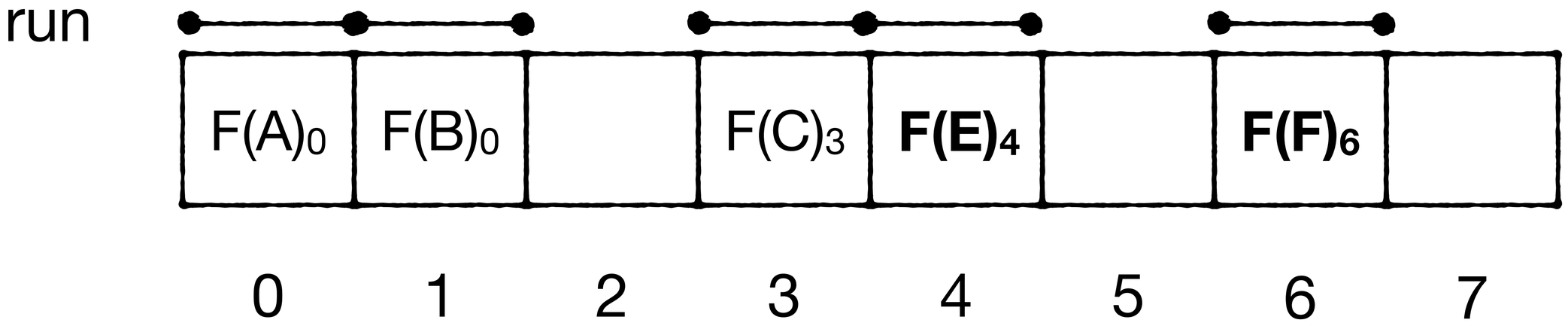
(1) Find run, remove matching fingerprint



(2) shift entries leftwards if needed to maintain contiguous runs as close as possible to their canonical slot



(2) shift entries leftwards if needed to maintain contiguous runs as close as possible to their canonical slot



Analysis

Query/insert/delete

False positive rate

Analysis

Query/insert/delete **O(1)** **expected time**

False positive rate

Analysis

Query/insert/delete O(1) expected time

False positive rate $\approx \alpha \cdot 2^{-(M/N - 2.125)/\alpha}$

Analysis

Query/insert/delete — O(1)

False positive rate — $\approx \alpha \cdot 2^{-(M/N - 2.125)/\alpha}$

Bits / entry budget

	Analysis
Query/insert/delete	O(1)
False positive rate	$\approx \alpha \cdot 2^{-(M/N - \mathbf{2.125})/\alpha}$

Metadata bits
(2 bitmaps and offsets field)

Analysis

Query/insert/delete O(1)

False positive rate $\approx \alpha \cdot 2^{-(M/N - 2.125)/\alpha}$

Load factor, $\alpha < 0.95$

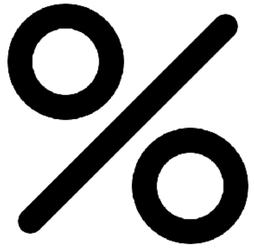
	Analysis
Query/insert/delete	O(1)
False positive rate	$\approx \alpha \cdot 2^{-(M/N - 2.125)/\alpha}$

Avg run length

Bloom

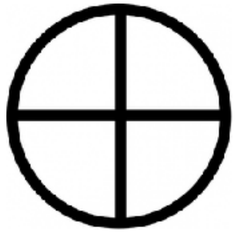
$\approx 2^{-M/N} \cdot 0.69$

Quotient



$\approx \alpha \cdot 2^{-(M/N - 2.125)/\alpha}$

XOR



$\approx 2^{-M/N} \cdot 0.81$

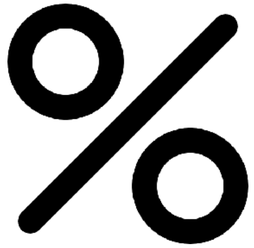
Idealized

$\approx 2^{-M/N}$

Bloom

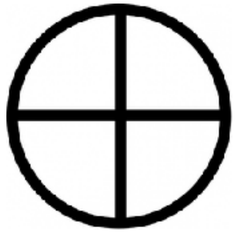
$\approx 2^{-M/N} \cdot 0.69$

Quotient



$\approx \alpha \cdot 2^{-(M/N - 2.125)/\alpha}$

XOR

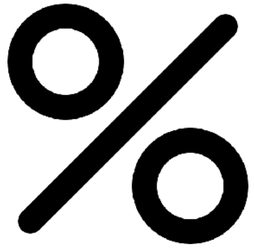
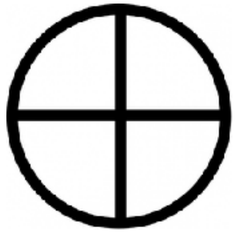


$\approx 2^{-M/N} \cdot 0.81$

Idealized

$\approx 2^{-M/N}$

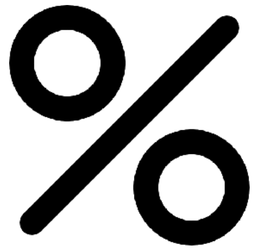
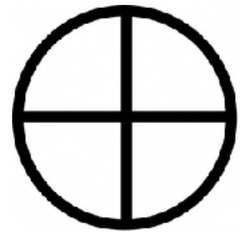
Lower than Bloom for M/N > 10

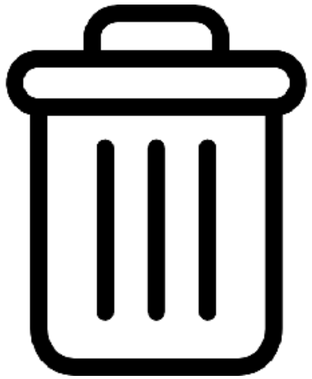
Bloom	Quotient	**XOR**	Idealized
			
$\approx 2^{-M/N} \cdot 0.69$	$\approx \alpha \cdot 2^{-(M/N - 2.125)/\alpha}$	$\approx 2^{-M/N} \cdot 0.81$	$\approx 2^{-M/N}$

Lower than Bloom for M/N > 10

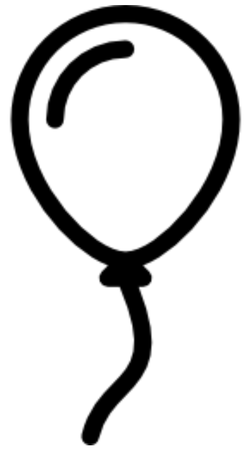
Supports deletes :)

Performance (cache misses)

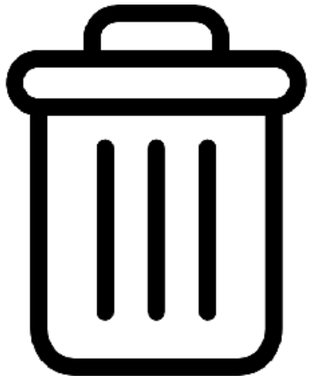
Blocked Bloom	Quotient	XOR
		
1	**≈1-2 on avg** *sequential*	*3* *random*



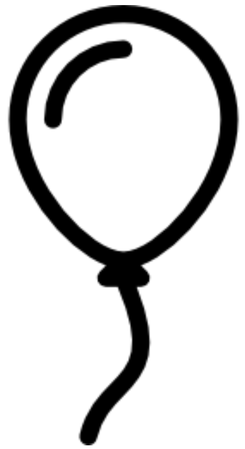
Deletes



Resizing



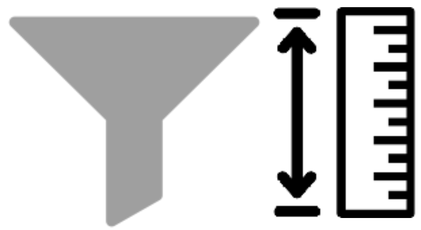
Deletes



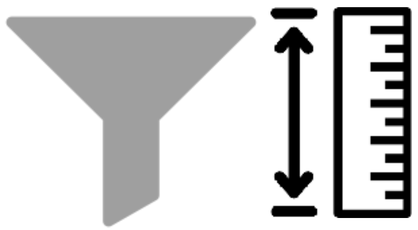
Resizing

Break

Allocated with fixed capacity

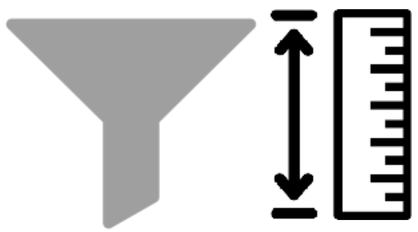


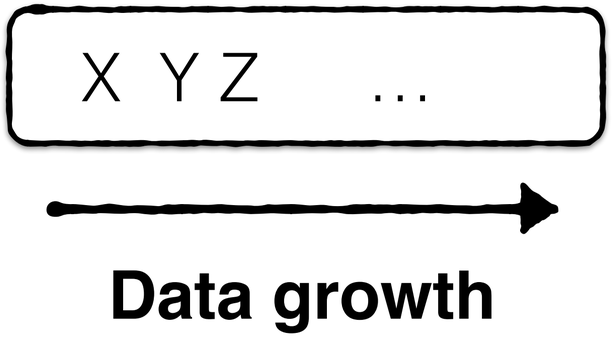
Allocated with fixed capacity



False positive rate

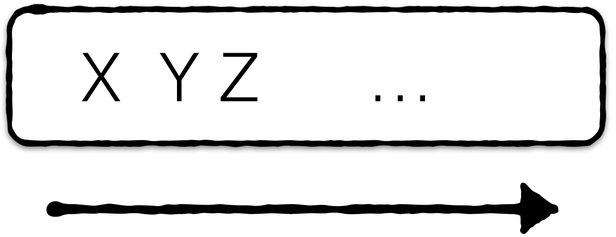
Insertion/query/ delete cost




X Y Z ...
Data growth

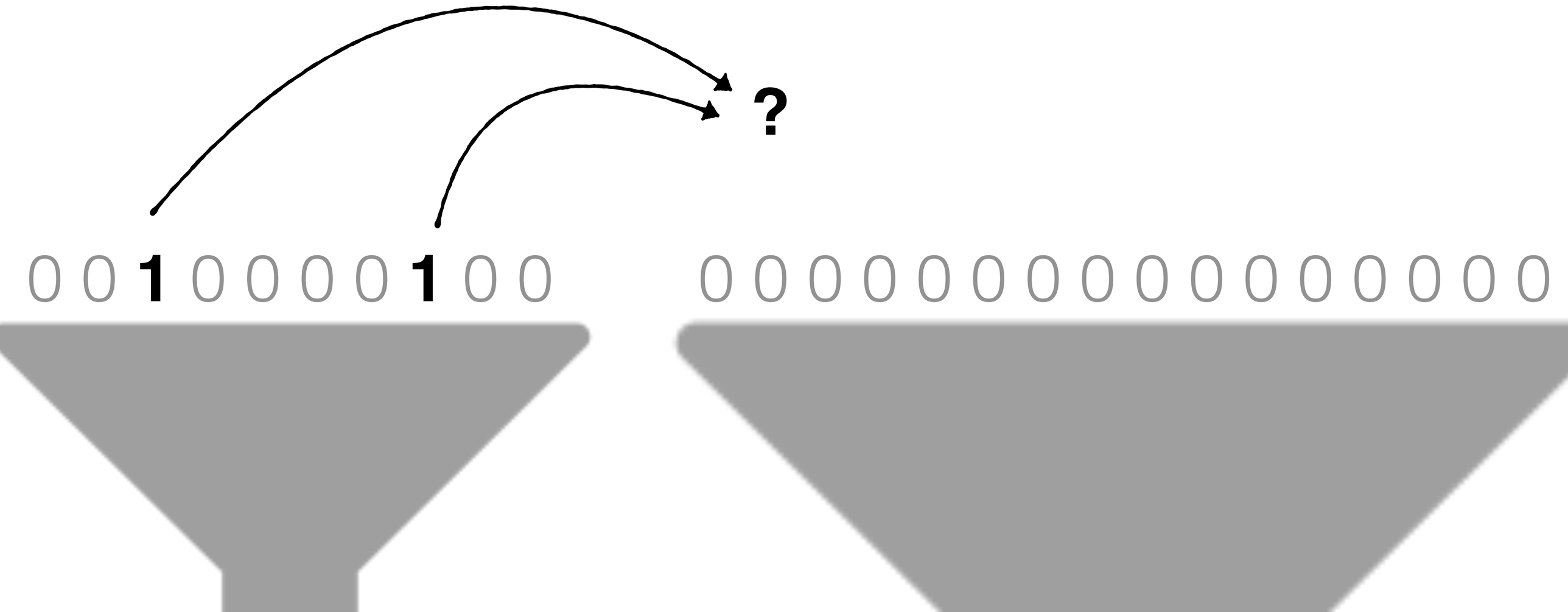
How to Expand Filters Efficiently?

?



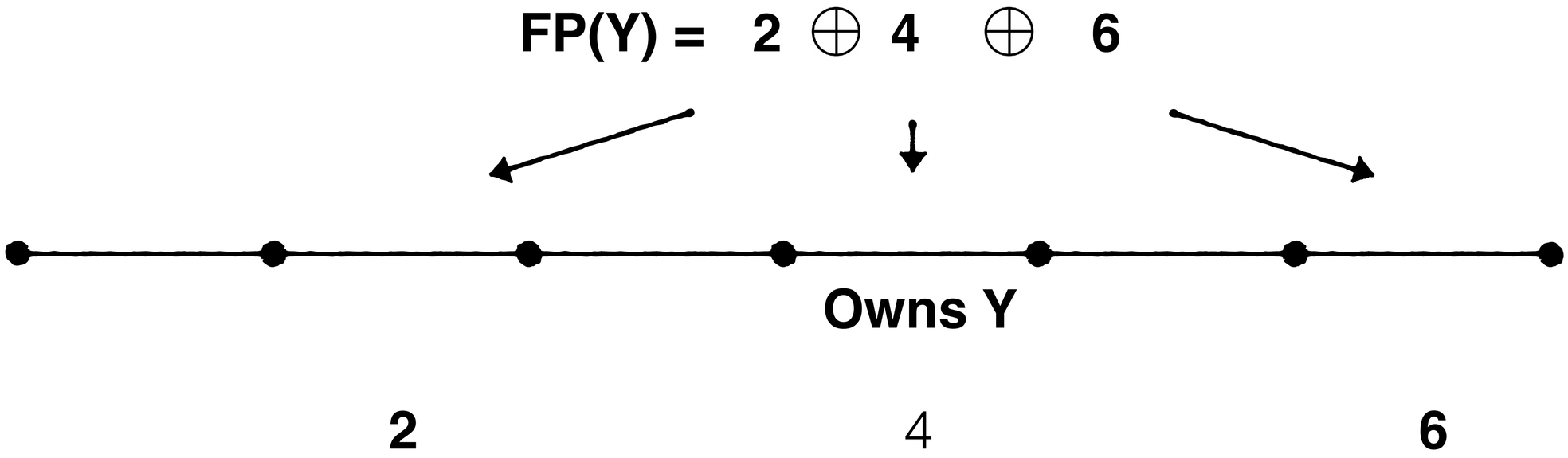
Data growth

Bloom Filters: unexpandable



XOR Filters: unexpandable

Can’t recover original fingerprints without accessing the original data



Expansion Workarounds

Expansion Workarounds

Pre-Allocation

Memory

Expansion Workarounds

Pre-Allocation

Memory

Reconstruction

Full scan

Agenda

Chaining

Quotient Filter

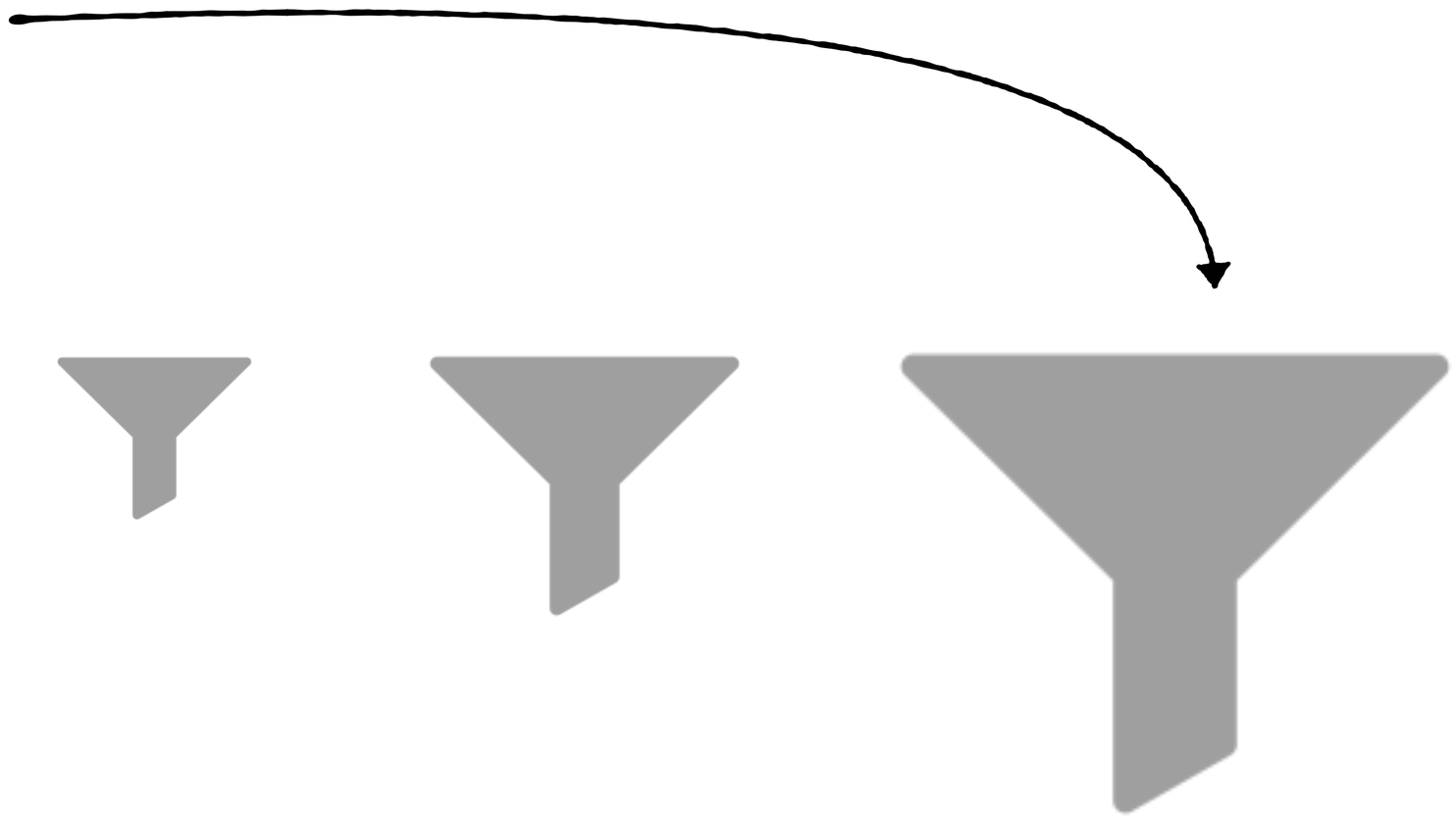
InfiniFilter &
Aleph Filter

Chaining

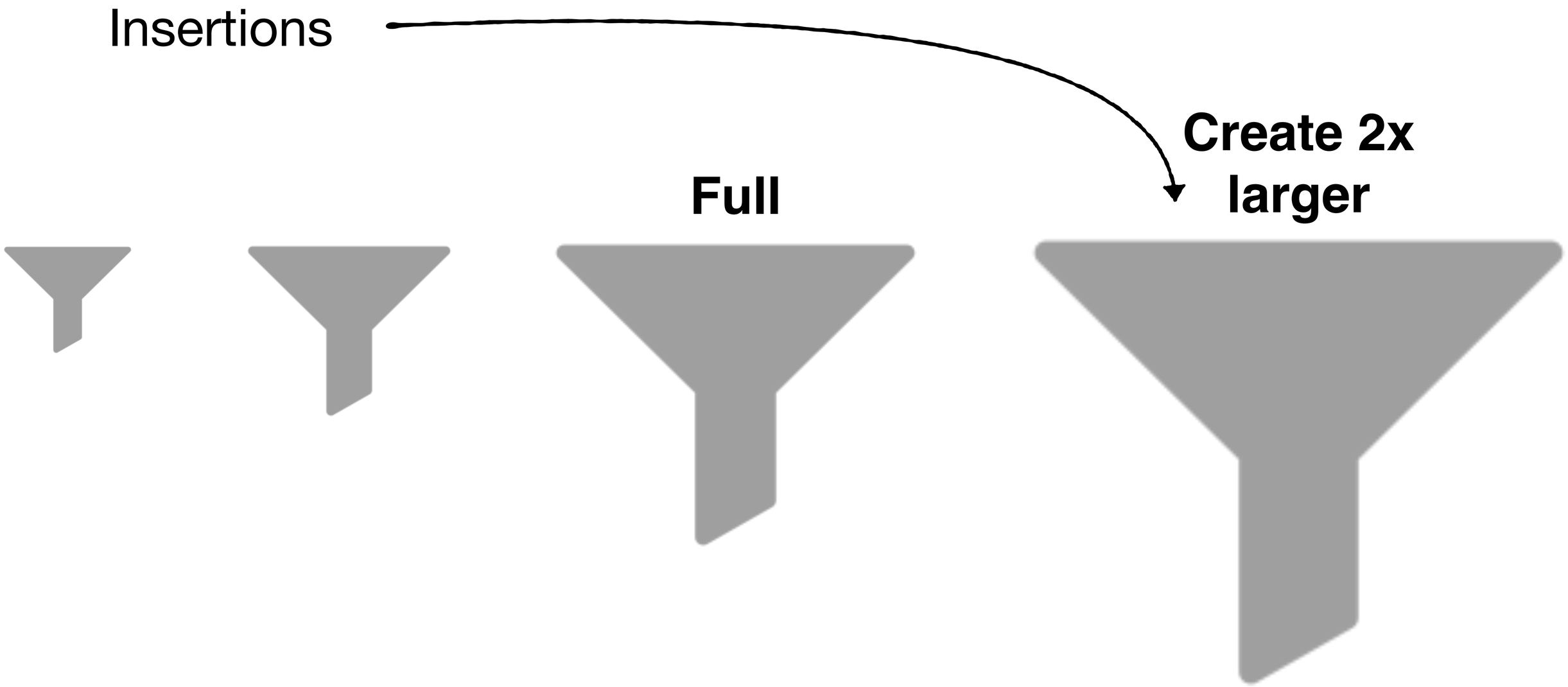
Create 2x larger filter when former reaches capacity

Chaining

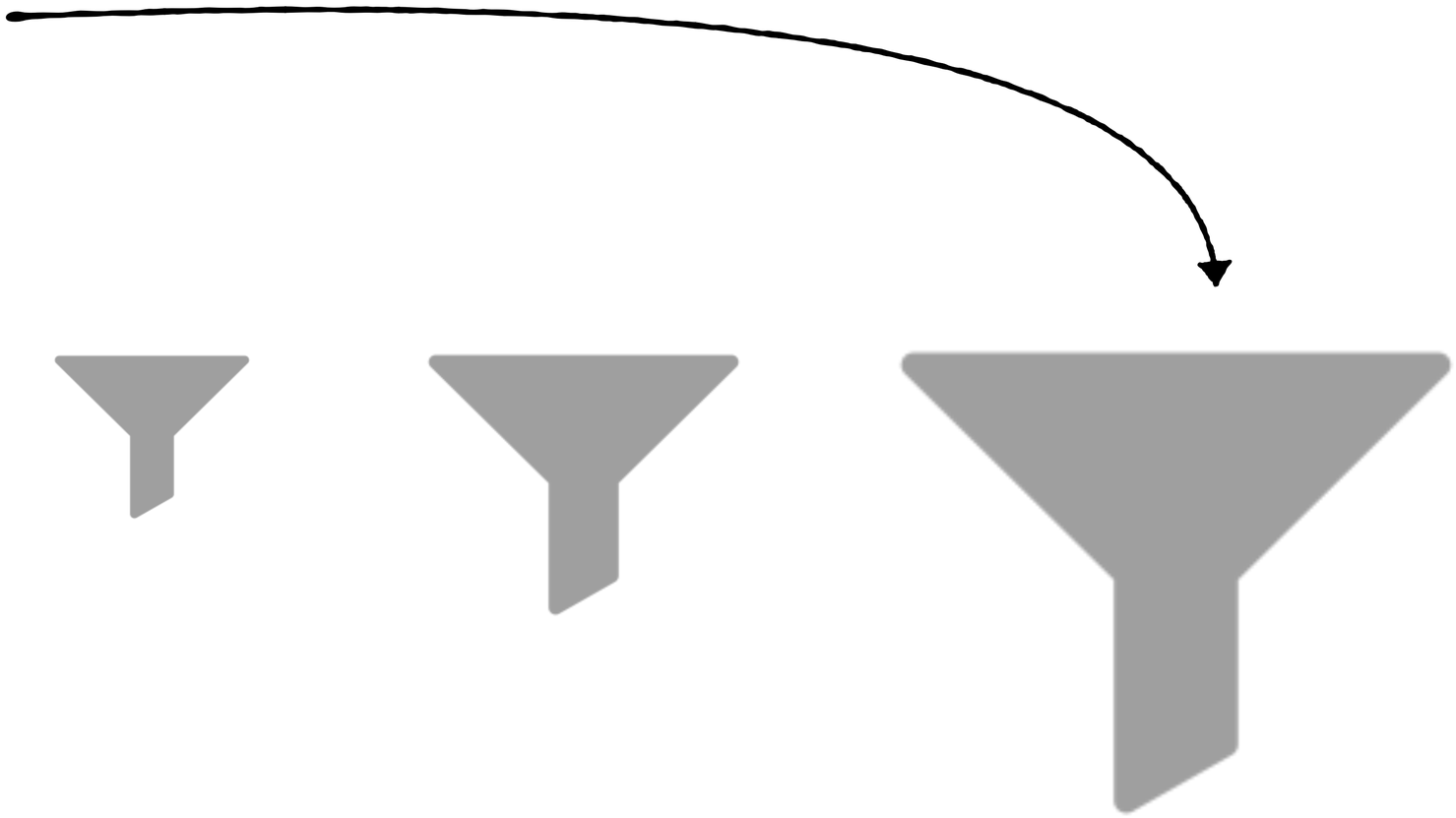
Insertions

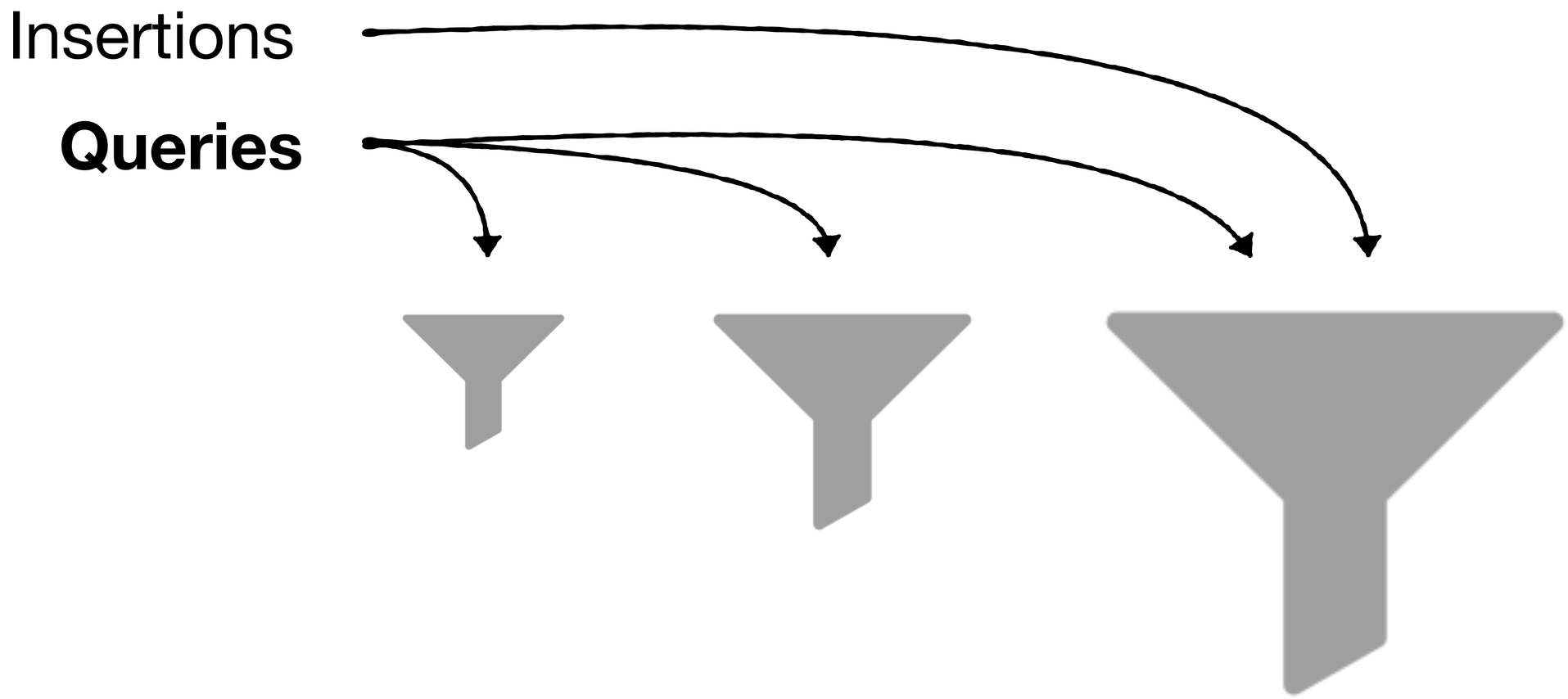


Chaining

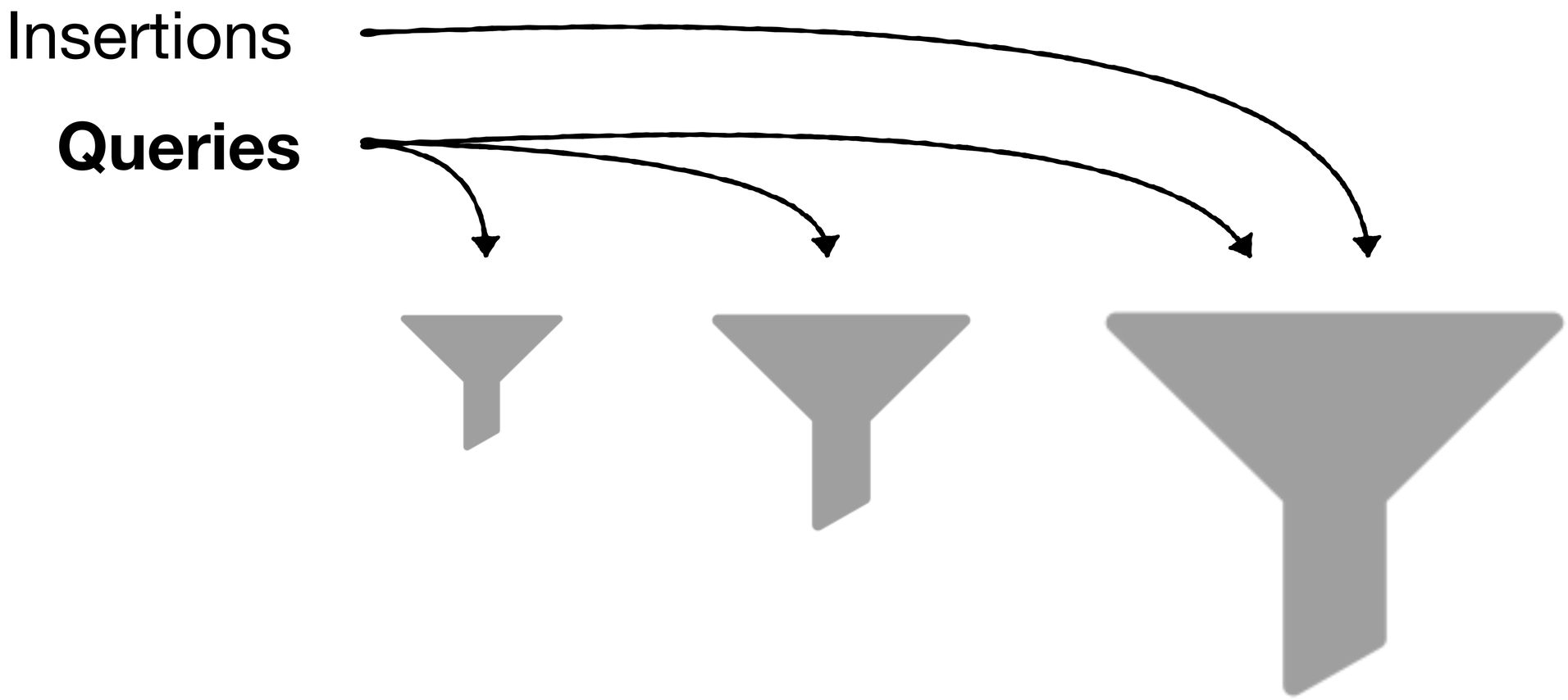


Insertions



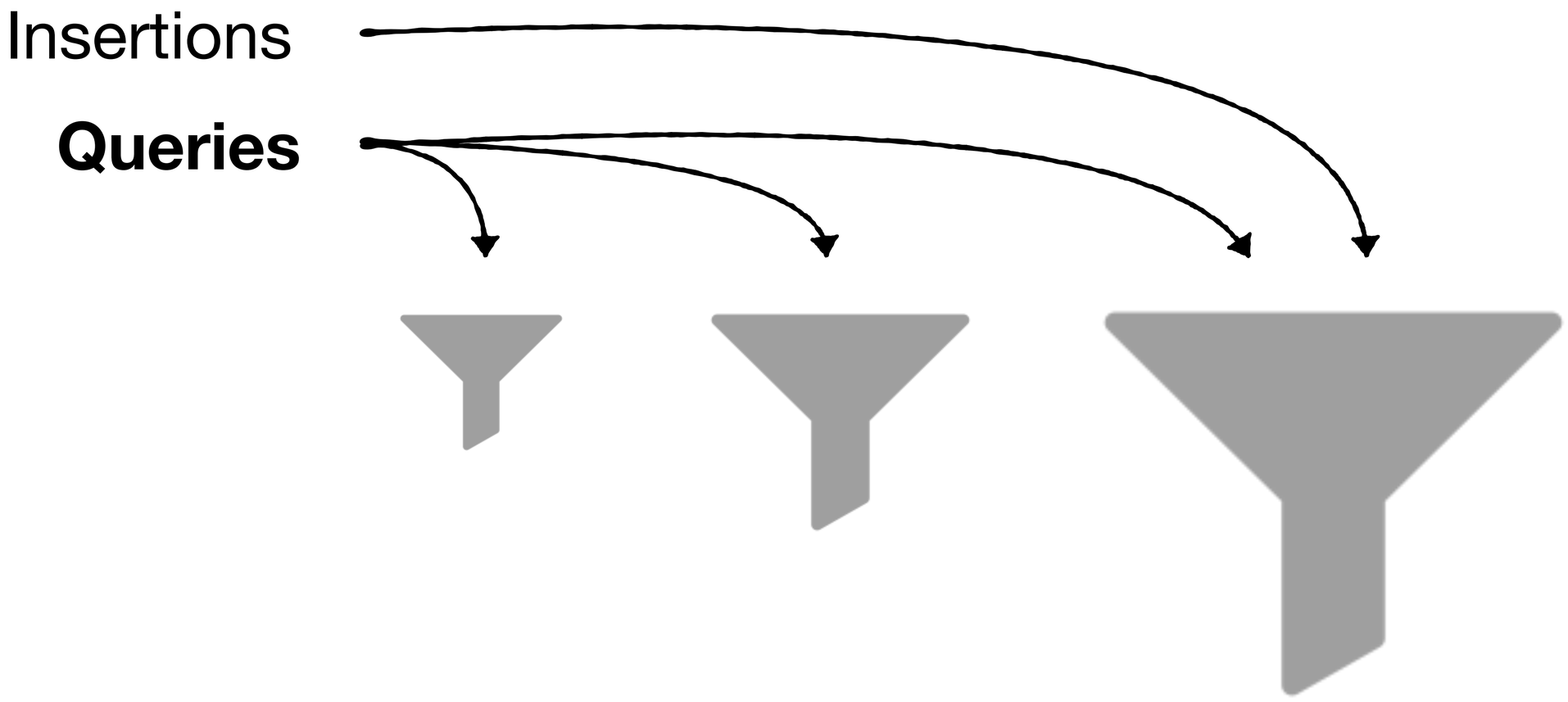

Insertions
Queries

Works with any filter



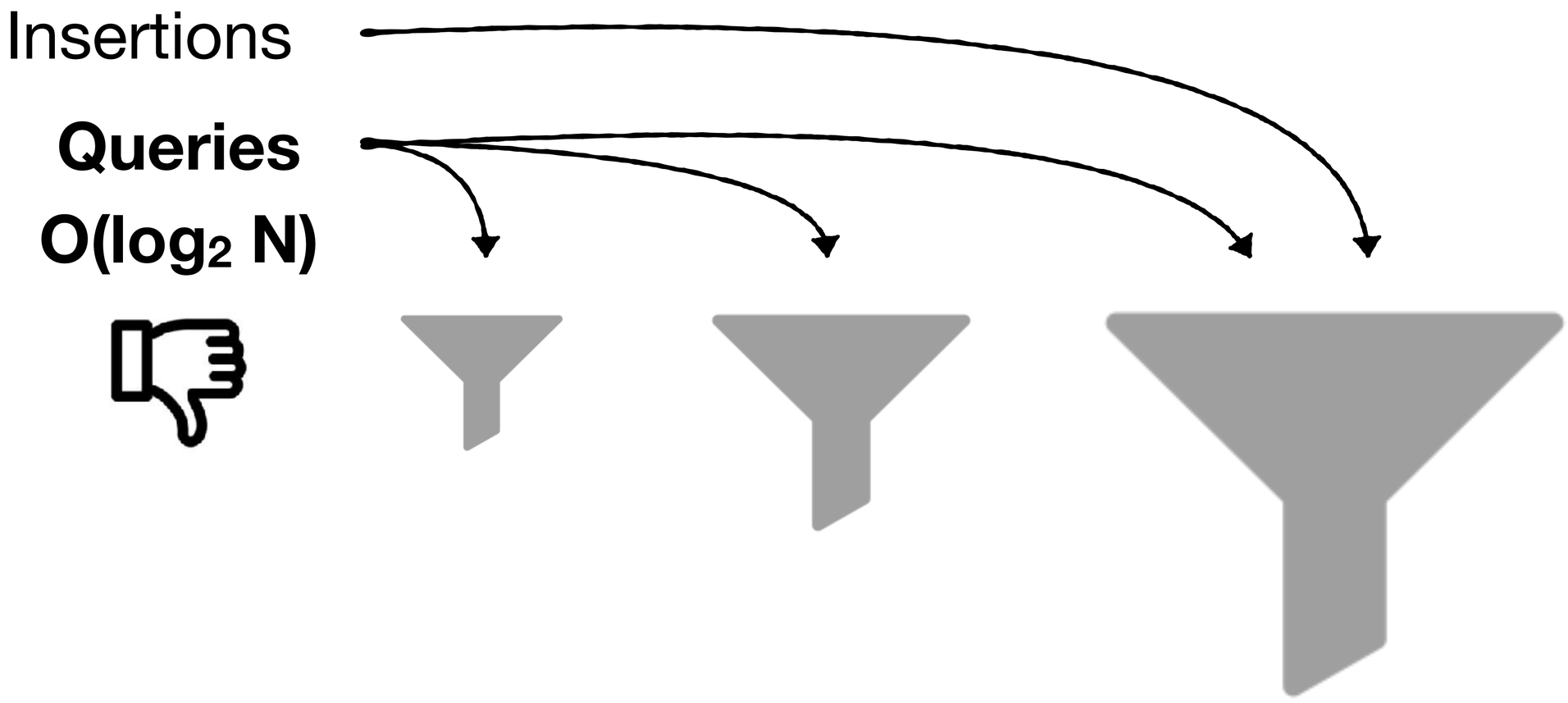
Works with any filter

Downsides?



Works with any filter

Downsides?



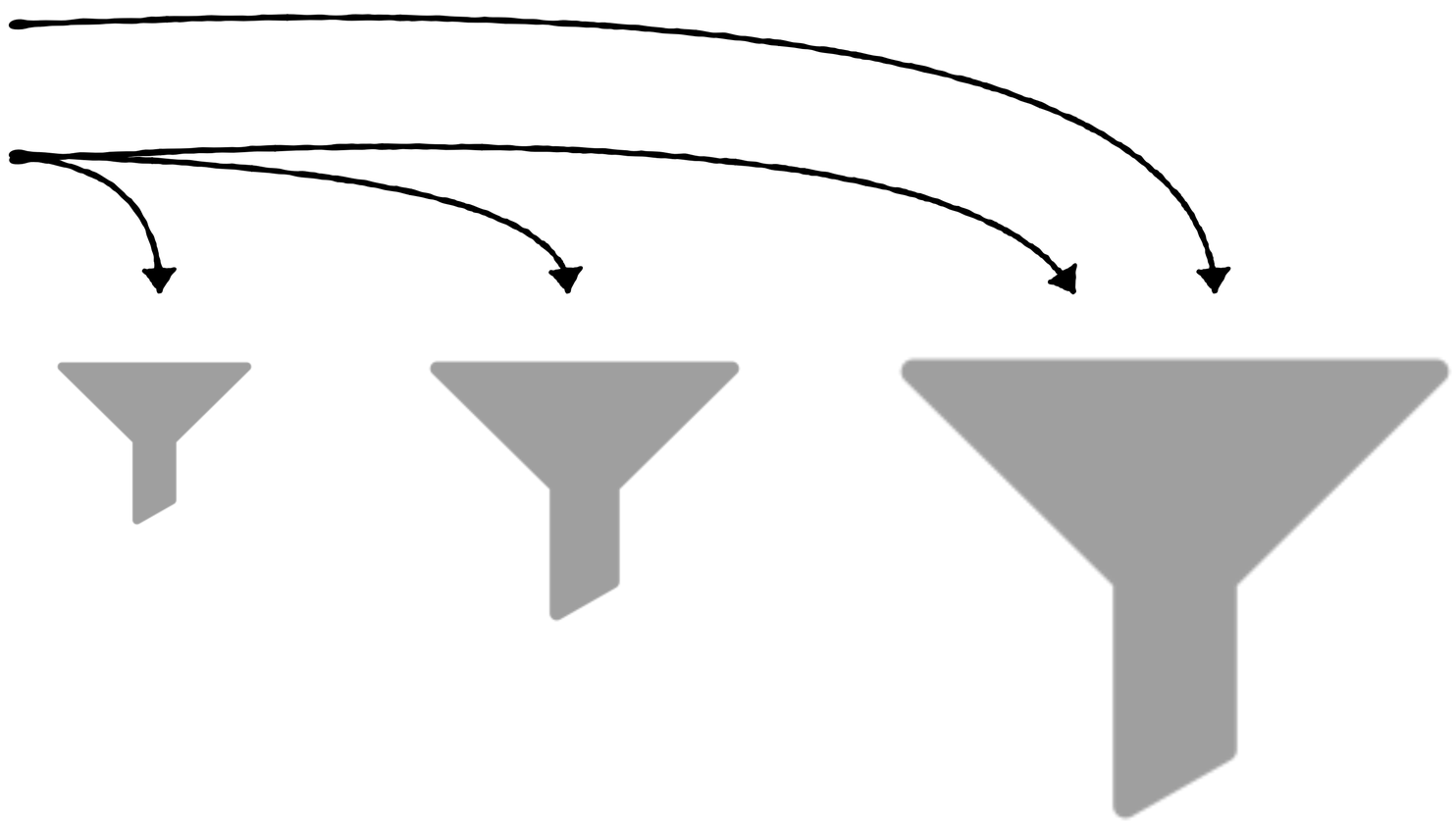
Works with any filter

Downsides?

Insertions

Queries

$O(\log_2 N)$



FPR?

FPR: $\varepsilon + \varepsilon + \varepsilon = O(\varepsilon \cdot \log_2 N)$

Suppose we want to keep it ε?

FPR: $\varepsilon + \varepsilon + \varepsilon = O(\varepsilon \cdot \log_2 N)$

Suppose we want to keep it ε?

FPR: $\varepsilon + \varepsilon + \varepsilon = O(\varepsilon \cdot \log_2 N)$

Set lower FPRs for newer filters

Geometrically decreasing. Any issue?

FPR: $\varepsilon + \varepsilon/2 + \varepsilon/4 = O(\varepsilon \cdot \log_2 N)$

FPR: ε + ε/2 + **ε/4** = O(ε)

Most Memory

Most data, lowest FPR

FPR: $\varepsilon + \varepsilon/2 + \varepsilon/4 = O(\varepsilon)$

Bits / entry: $\log(4/\varepsilon)$

FPR: $\varepsilon + \varepsilon/2 + \varepsilon/4 = O(\varepsilon)$

Bits / entry: $\log(\mathbf{2}^{\log \mathbf{N}}/\varepsilon)$

Can we better scale memory?

FPR: $\varepsilon + \varepsilon/2 + \varepsilon/4 = O(\varepsilon)$

Bits / entry:  $\mathbf{\log_2 N} + \log(1/\varepsilon)$

The FPRs should decrease more slowly but still converge

FPR: $\varepsilon + \varepsilon/2 + \varepsilon/4 = O(\varepsilon)$

Bits / entry: 👎 $\log_2 N + \log(1/\varepsilon)$

Reciprocal of square numbers

$$1/1^2 + 1/2^2 + 1/3^2 = ?$$

Solved by Euler in 1734

$$1/1^2 + 1/2^2 + 1/3^2 = \pi^2/6$$

Solved by Euler in 1734

$$1/1^2 + 1/2^2 + 1/3^2 = \boldsymbol{\pi^2/6}$$

$$= 1.645$$

Solved by Euler
in 1734

$$1/1^2 + 1/2^2 + 1/3^2 = \pi^2/6$$

Polynomially decreasing yet still convergent

FPR: $\varepsilon/1^2 + \varepsilon/2^2 + \varepsilon/3^2 = \varepsilon \cdot \pi^2/6$

FPR: $\varepsilon/1^2 + \varepsilon/2^2 + \varepsilon/3^2 = \varepsilon \cdot \pi^2/6$

Bits / entry: $\log(3^2/\varepsilon)$

FPR: $\varepsilon/1^2 + \varepsilon/2^2 + \varepsilon/3^2 = \varepsilon \cdot \pi^2/6$

Bits / entry: $\log(\log(N)^2/\varepsilon)$

FPR: $\varepsilon/1^2 + \varepsilon/2^2 + \varepsilon/3^2 = \varepsilon \cdot \pi^2/6$

Bits / entry: $2 \log_2 \log_2(N) + \log(/\varepsilon)$

FPR: $\approx \varepsilon$

Bits / entry: $\mathbf{2\ \log_2 \log_2 (N) + \log(/\varepsilon)} \quad < \quad \mathbf{\log N + \log(1/\varepsilon)}$

FPR: $\approx \varepsilon$

Bits / entry: $2 \log_2 \log_2 (N) + \log(1/\varepsilon)$

Close to lower bound

How to Approximate A Set Without Knowing Its Size In Advance

Rasmus Pagh, Gil Segev, Udi Wieder. **FOCS 2013.**

FPR: $\approx \varepsilon$

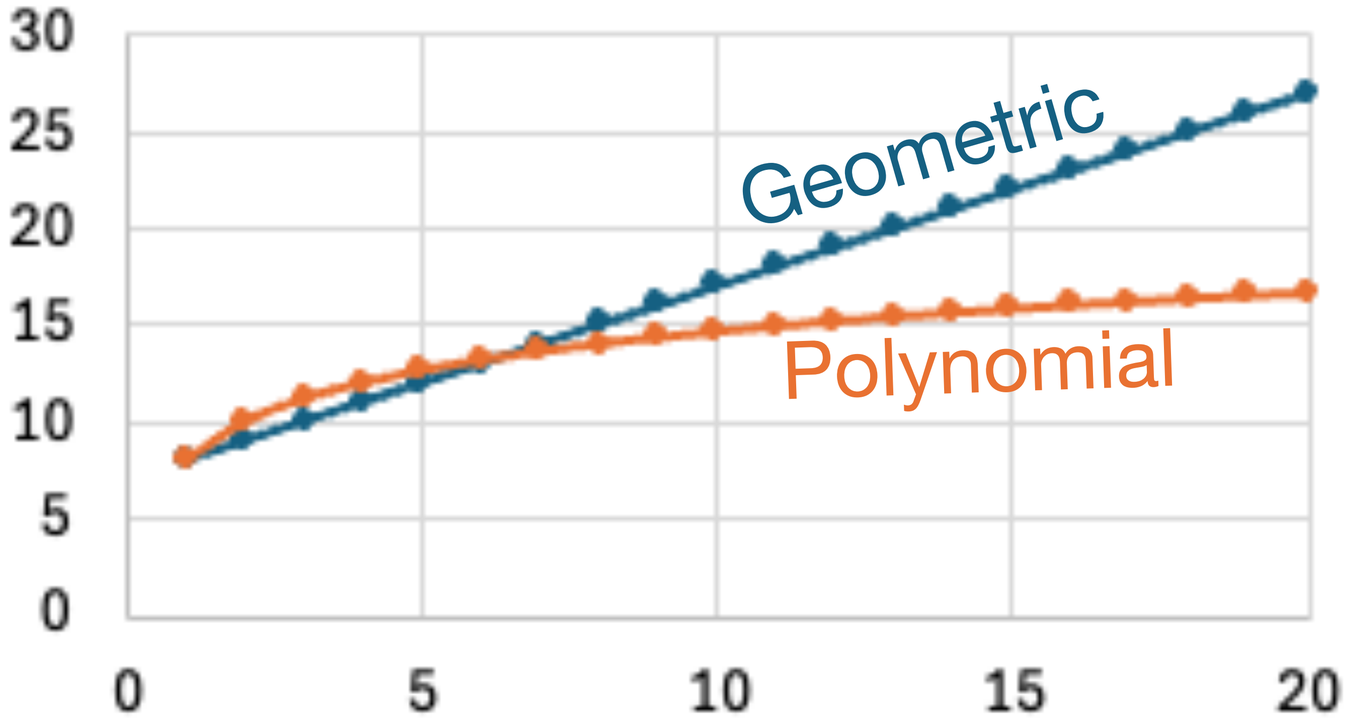
Bits / entry: $2 \log_2 \log_2 (N) + \log(1/\varepsilon)$

Close to lower bound

How to Approximate A Set Without Knowing Its Size In Advance
Rasmus Pagh, Gil Segev, Udi Wieder. FOCS 2013.

Much of what follows originates from here :)

Bits / entry



expansions

Chaining

Quotient Filters

InfiniFilter &
Aleph Filter

queries

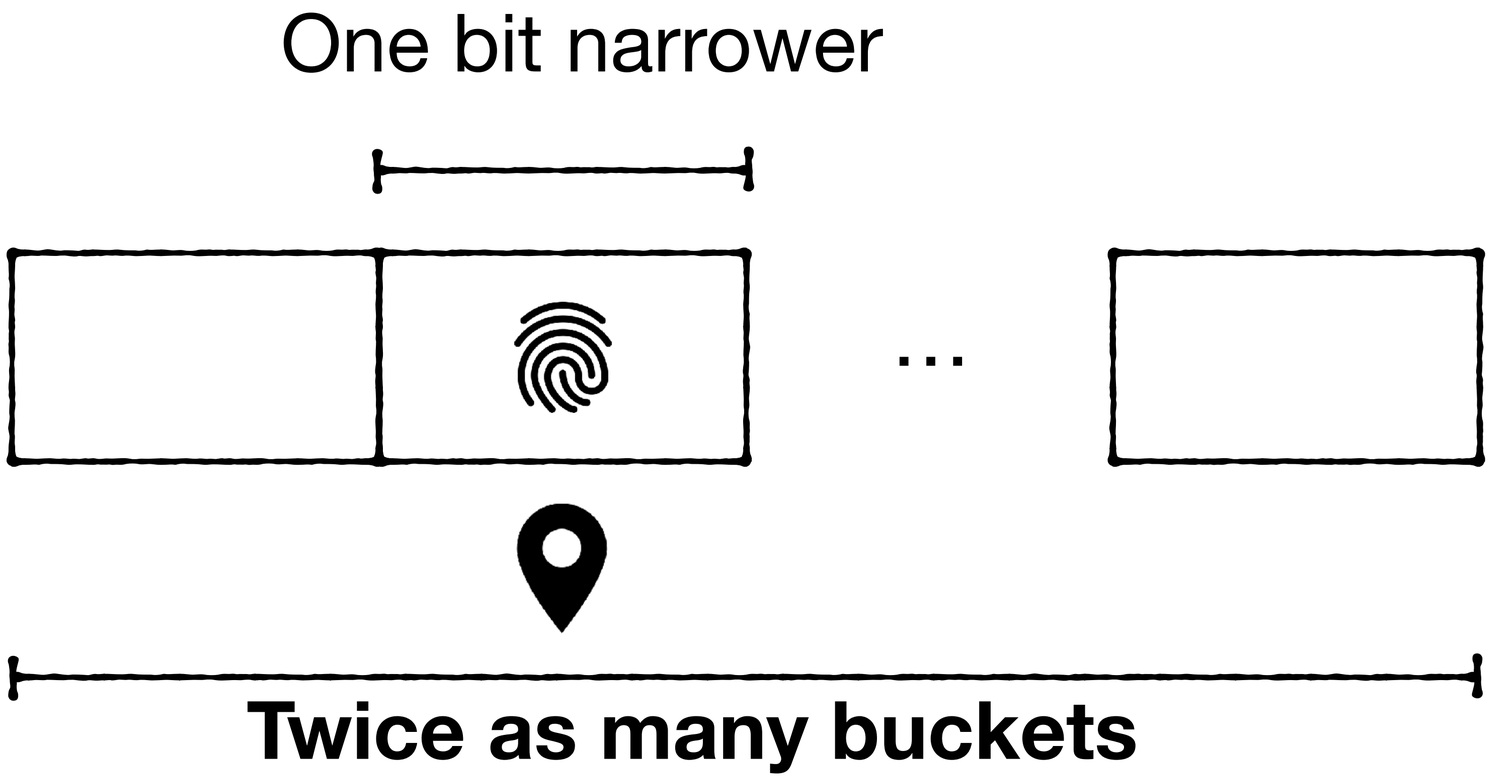
Quotient Filters are Semi-Expandable

Semi-Expandable


hash( )
0 1 0 1 0 0 1 1 0 1 0 1 1 0 1 1 0 0


hash( )
0 1 0 1 0 0 1 1 0 1 0 1 1 0 1 1 0 0

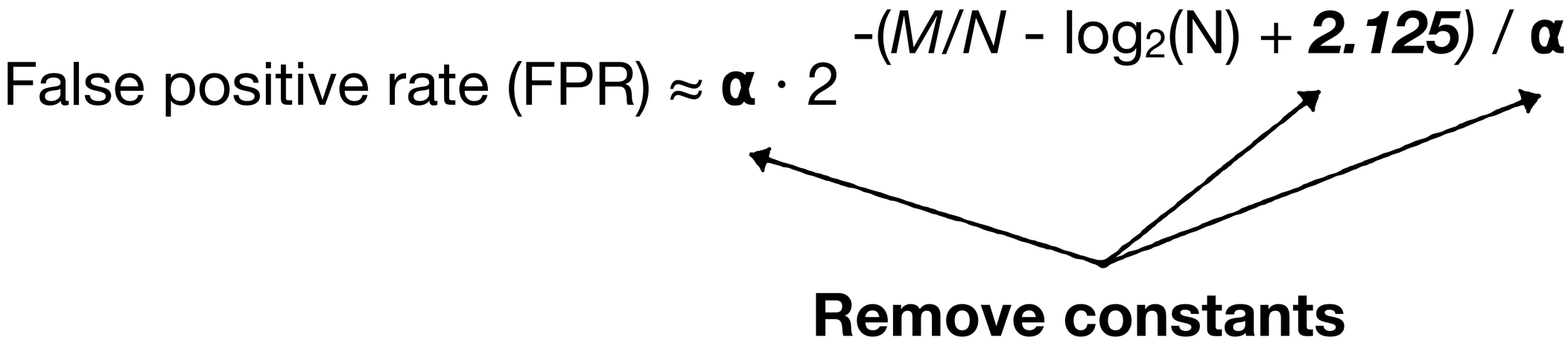
One bit narrower


One bit narrower
...
Twice as many buckets

$$\textbf{False positive rate (FPR)} \approx \alpha \cdot 2^{-(M/N + 2.125) / \alpha}$$

$$\text{False positive rate (FPR)} \approx \alpha \cdot 2^{-(M/N - \mathbf{\log_2(N)} + 2.125) / \alpha}$$

Lose 1 fingerprint bit in each expansion


False positive rate (FPR) ≈ **α** · 2
-(*M*/*N* - log$_2$(N) + ***2.125***) / **α**
Remove constants

False positive rate (FPR) $\approx 2^{-(M/N - \mathbf{\log_2(N)})}$

Simplify

$$\text{False positive rate (FPR)} \approx \mathbf{N} \cdot 2^{-M/N}$$

Linear increase with data size

False positive rate (FPR) ≈ $N \cdot 2^{-M/N}$

Supports up to *M/N* expansions

False positive rate (FPR) ≈ $N \cdot 2^{-M/N}$

Supports up to *M/N* expansions

O(1) operations

Chaining

queries

Quotient Filters

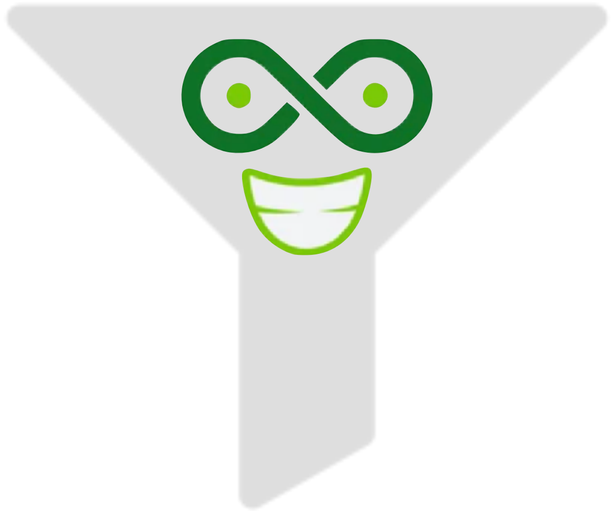
FPR

expansions

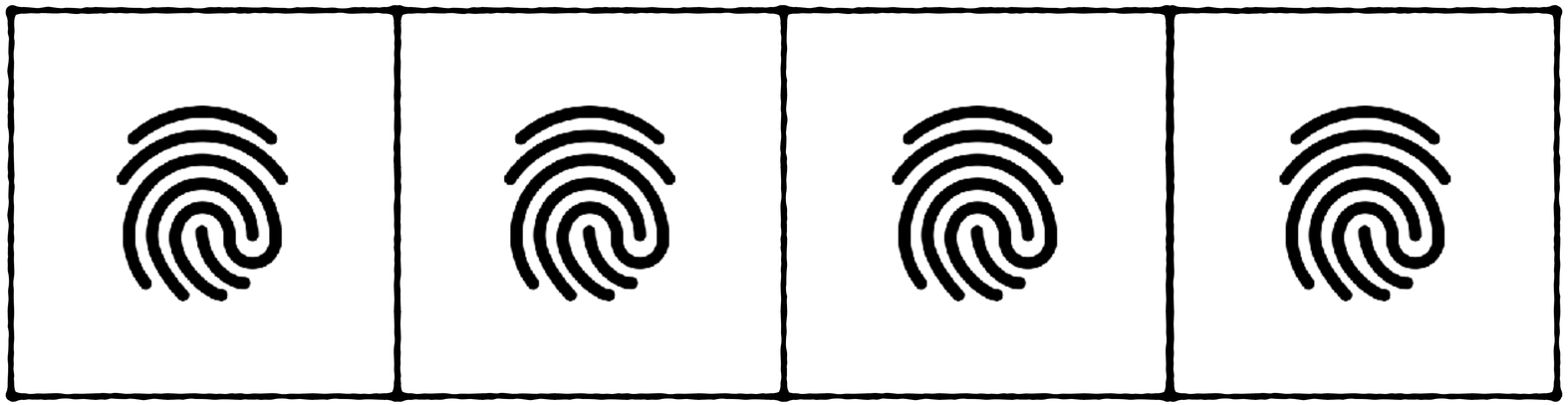
InfiniFilter & Aleph Filter

InfiniFilter: Expanding Filters to Infinity and Beyond

Niv Dayan, Ioana Bercea, Pedro Reviriego, Rasmus Pagh. SIGMOD 2023

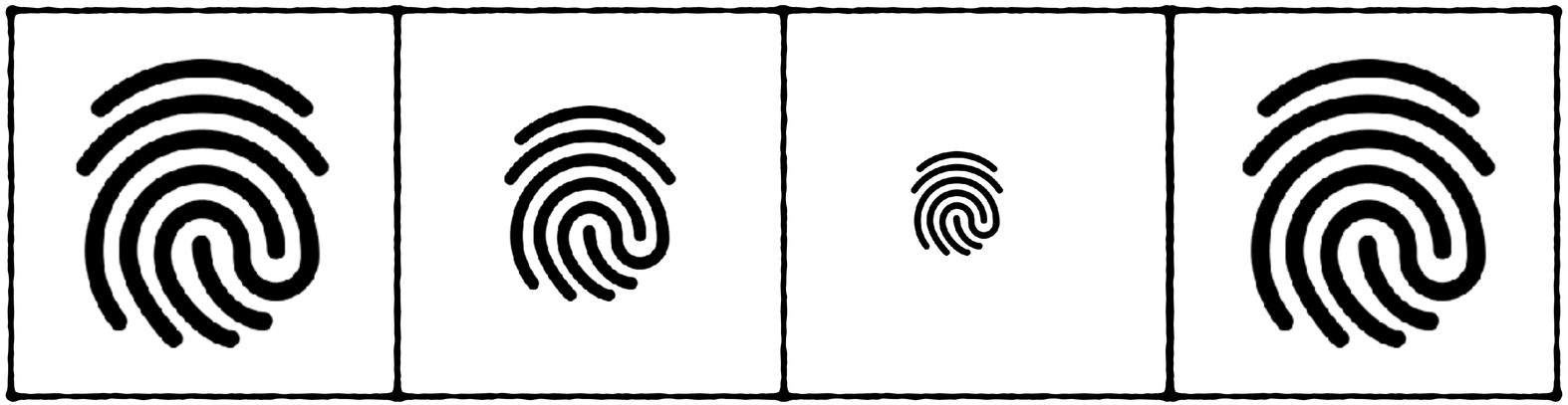


InfiniFilter



Quotient filter

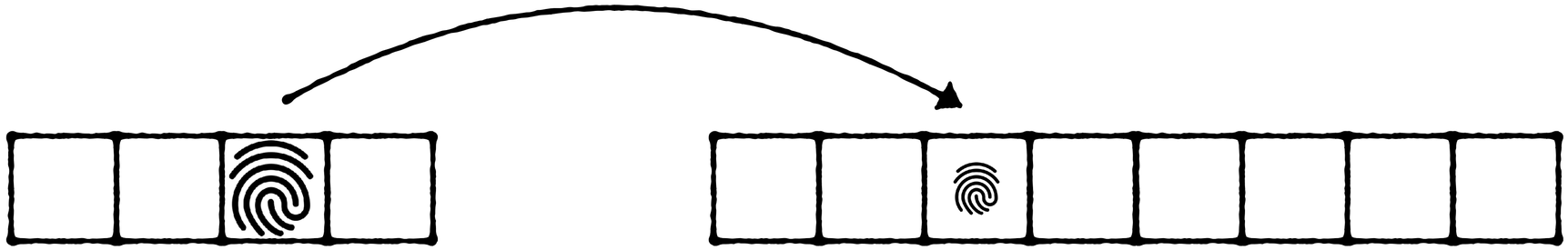
InfiniFilter



Variable-sized fingerprints

InfiniFilter

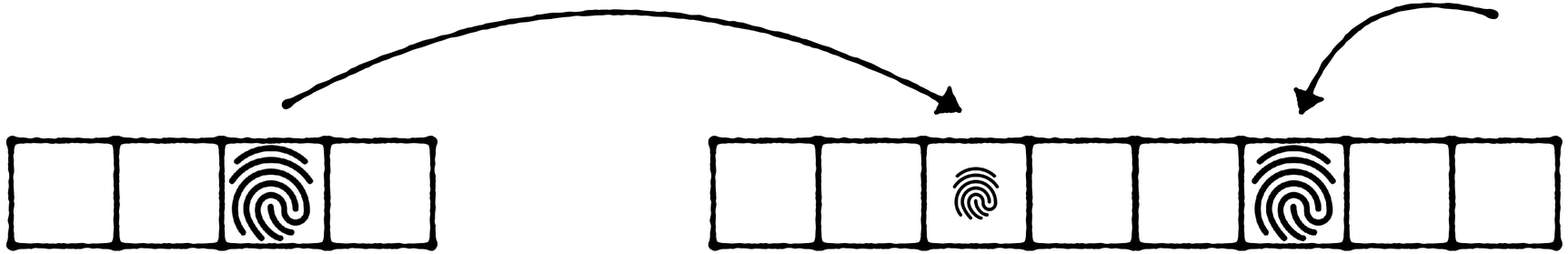
(1) sacrifice one bit during expansion



InfiniFilter

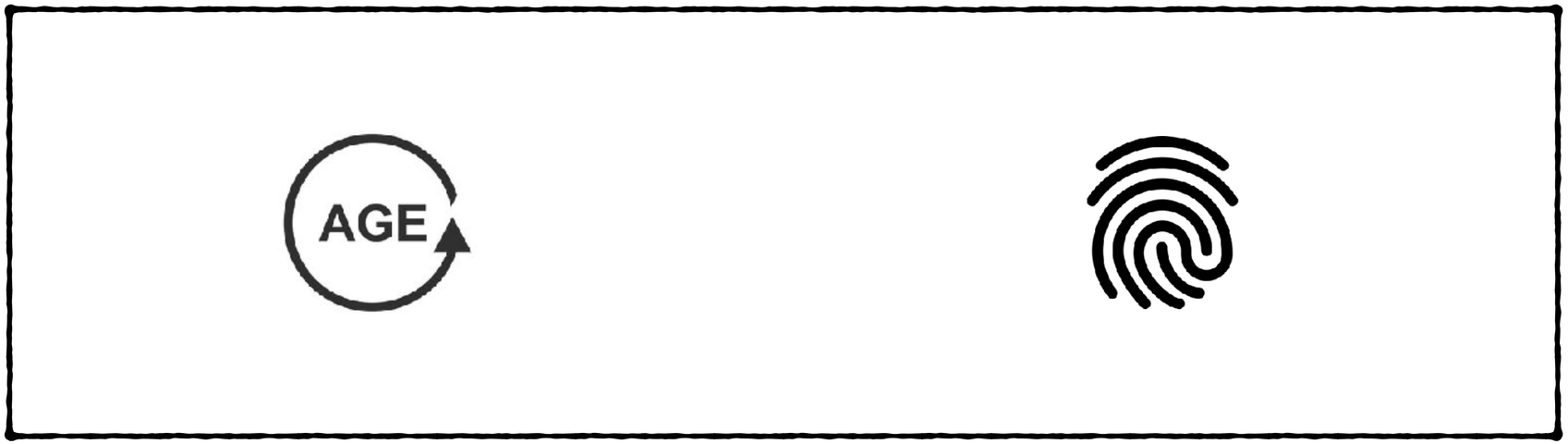
(1) sacrifice one bit
during expansion

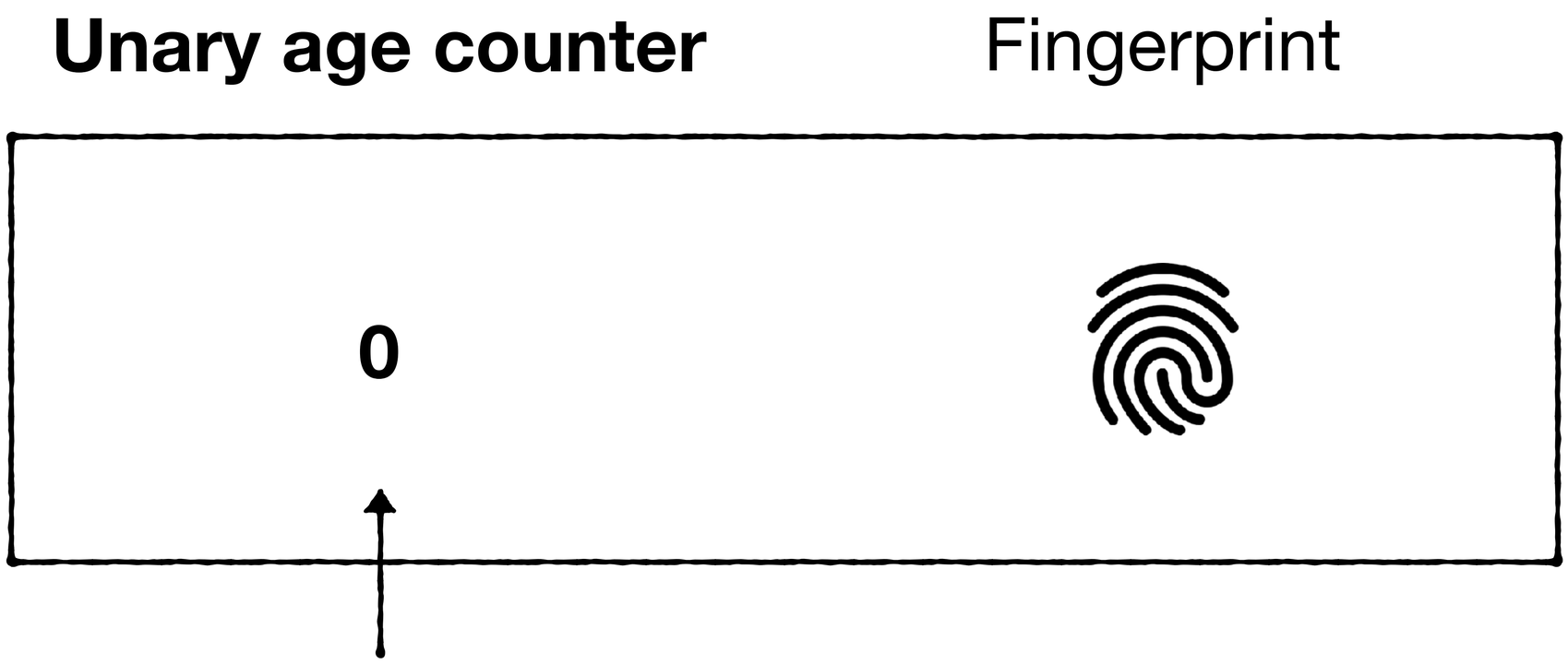
(2) Newer entries get longer fingerprints

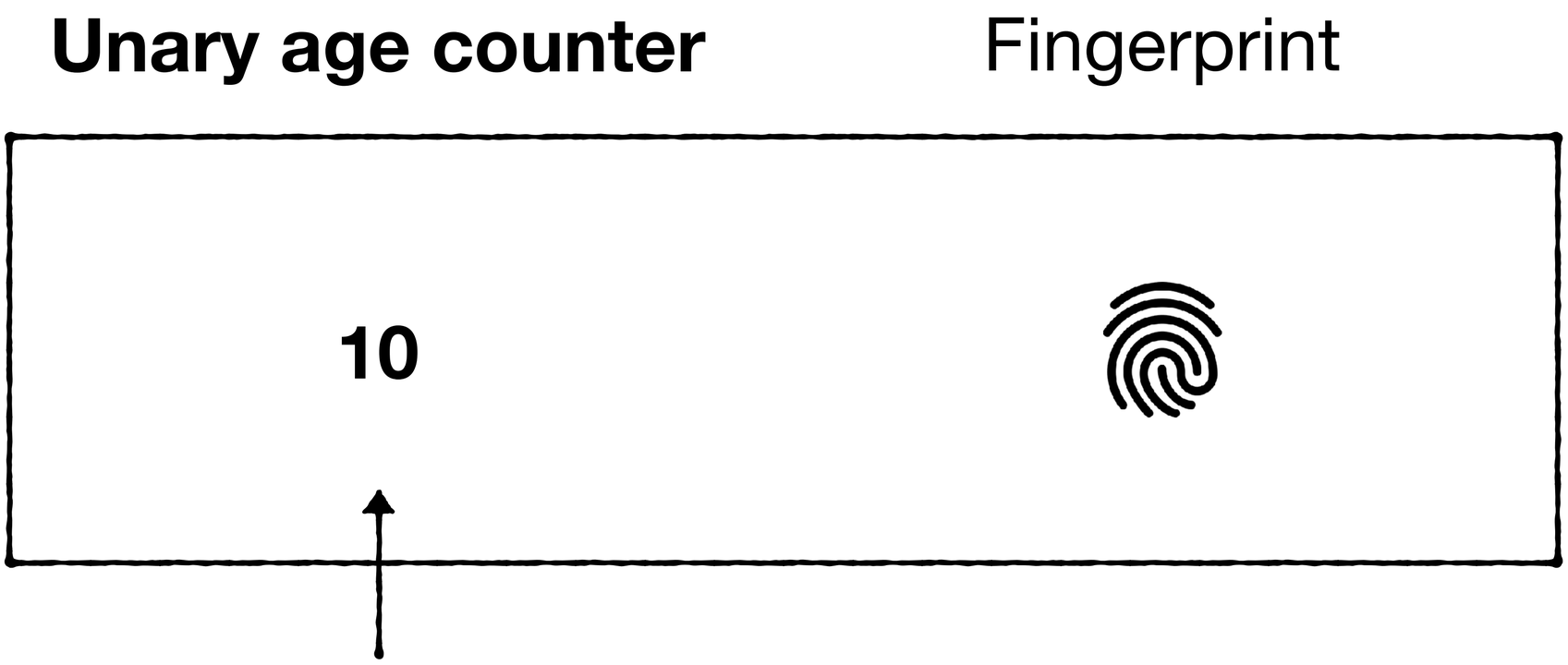


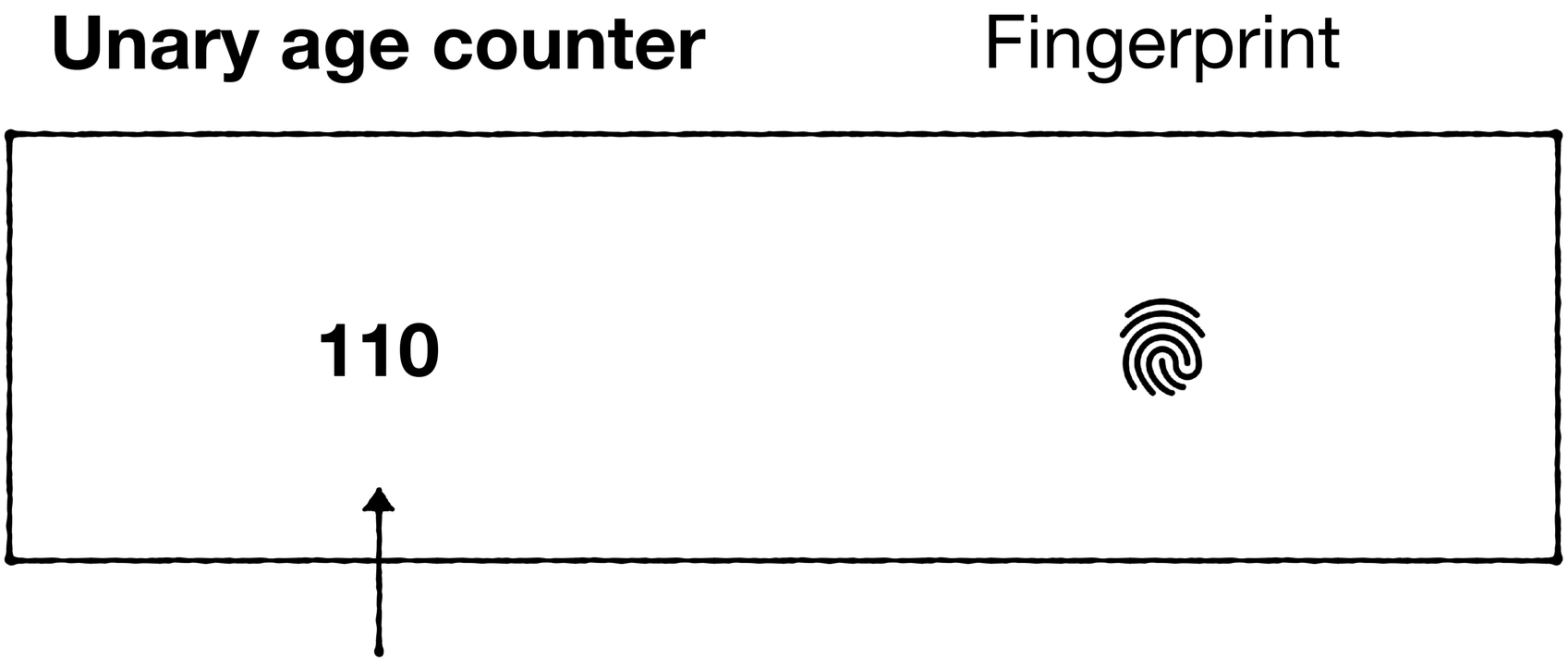
Unary age counter

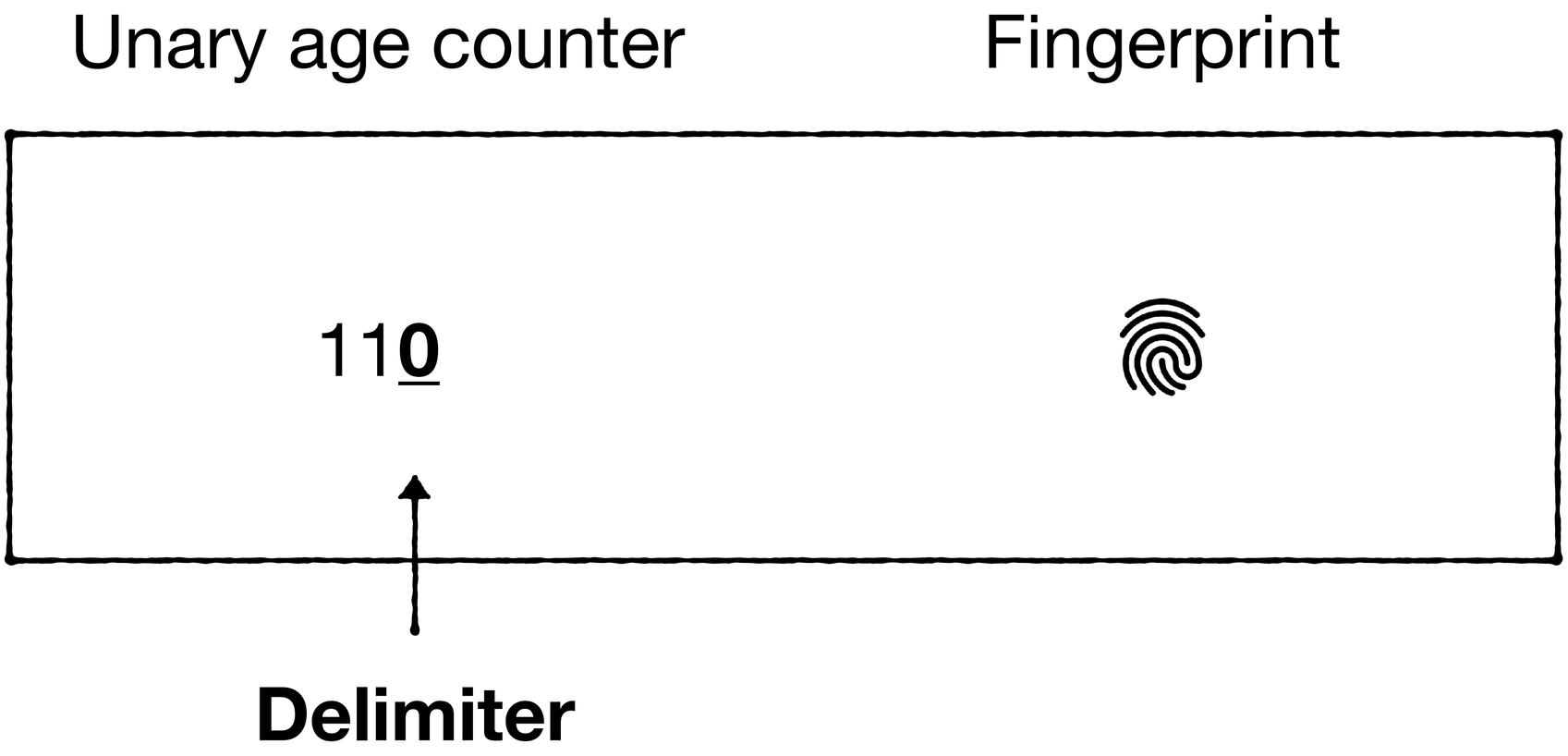
Fingerprint

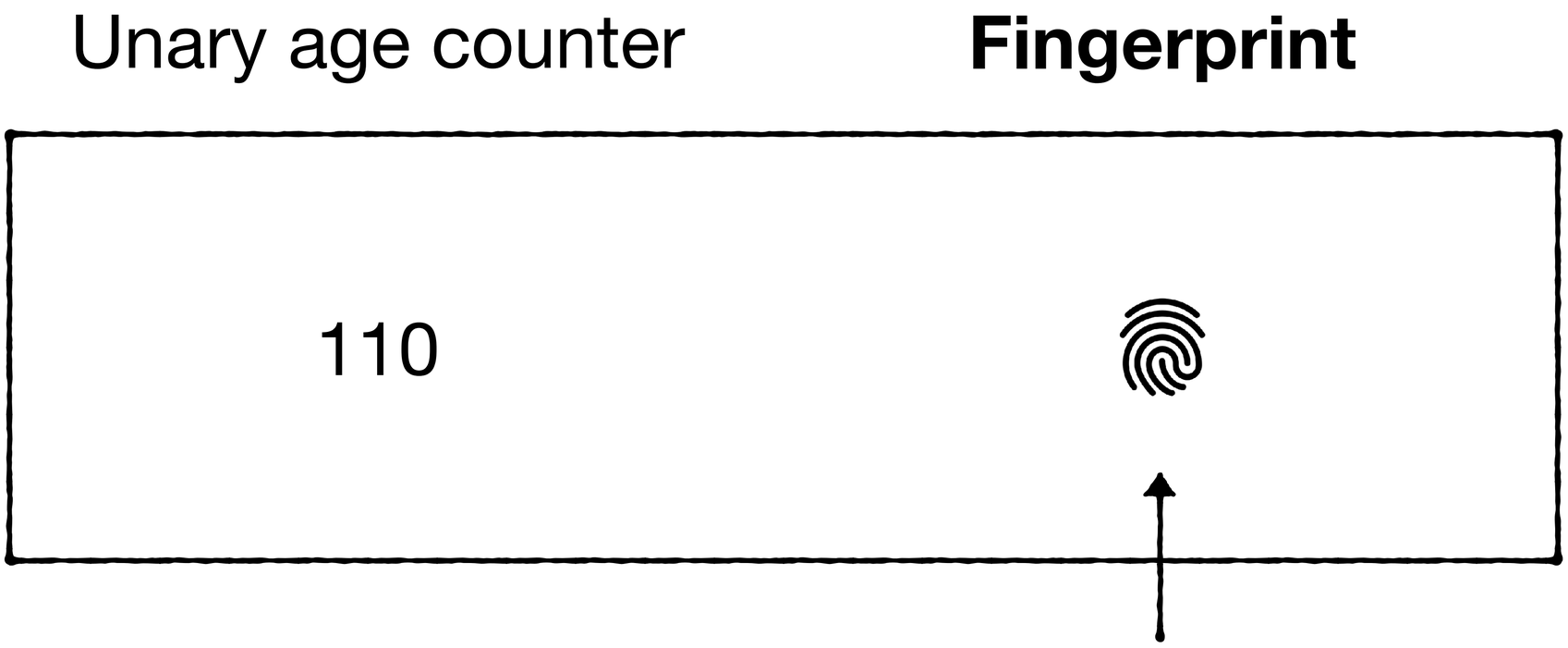



Unary age counter
Fingerprint
0
0 expansions ago

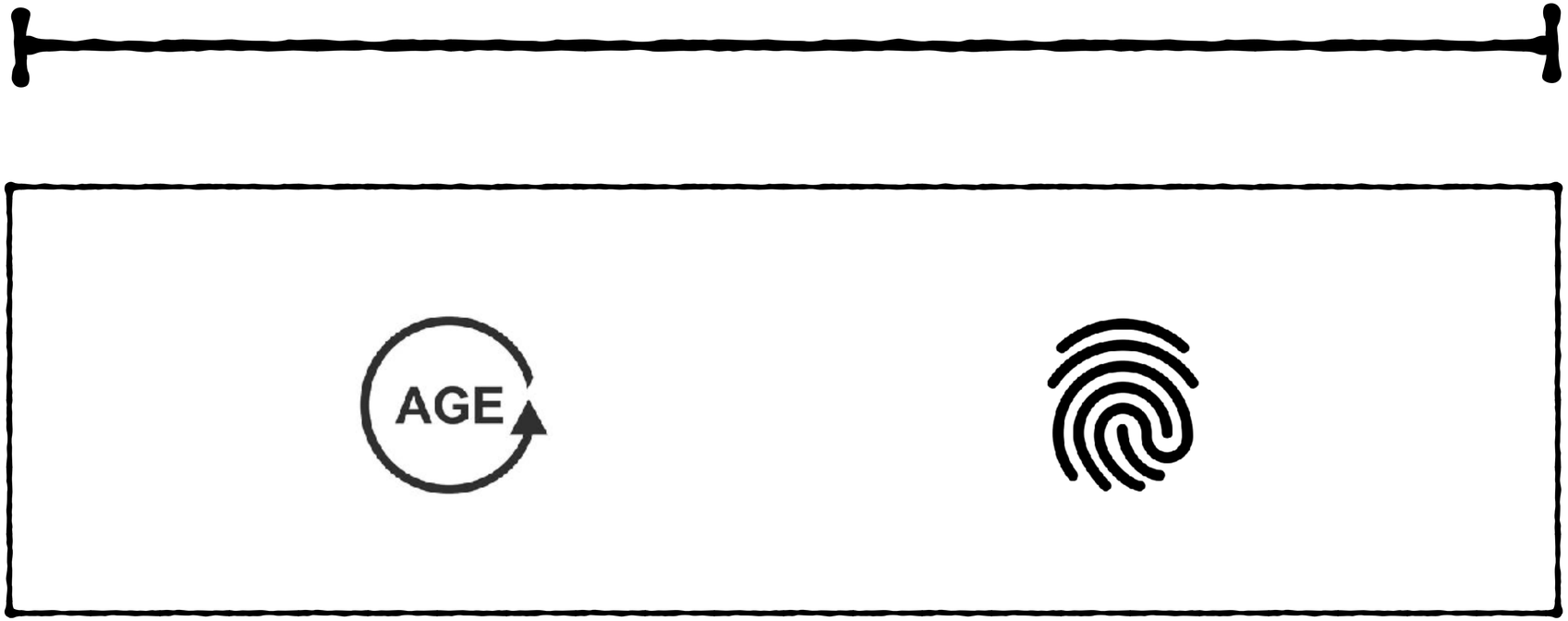

Unary age counter
Fingerprint
10
1 expansions ago


Unary age counter
Fingerprint
110
2 expansions ago

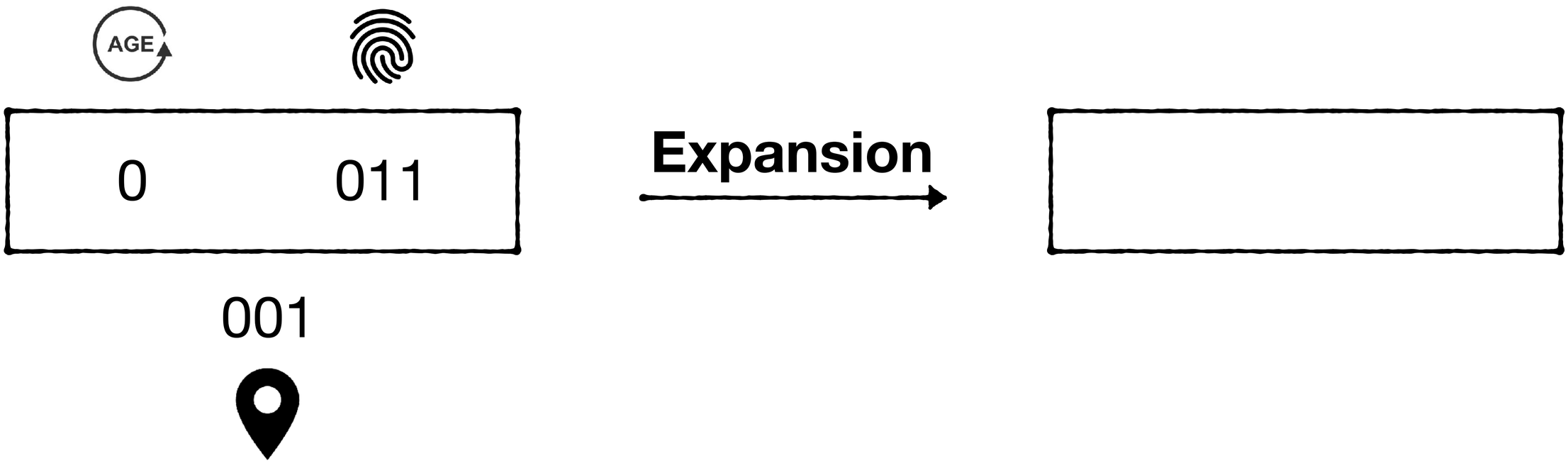

Unary age counter
Fingerprint
110
Delimiter

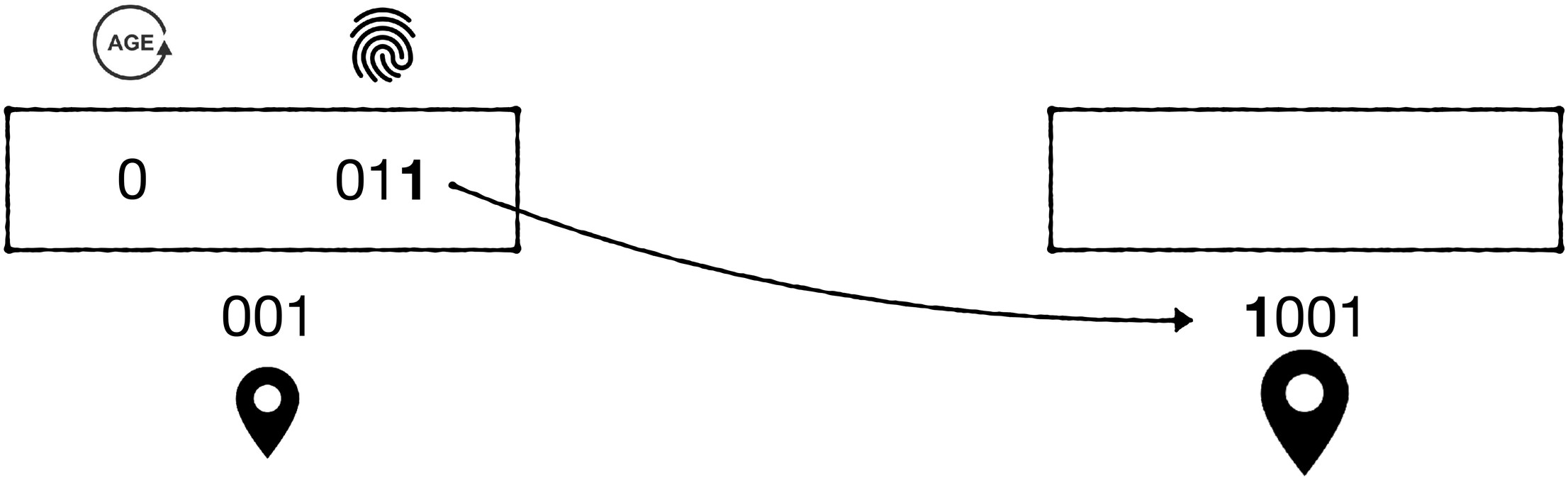

Unary age counter
Fingerprint
110
All remaining slot bits

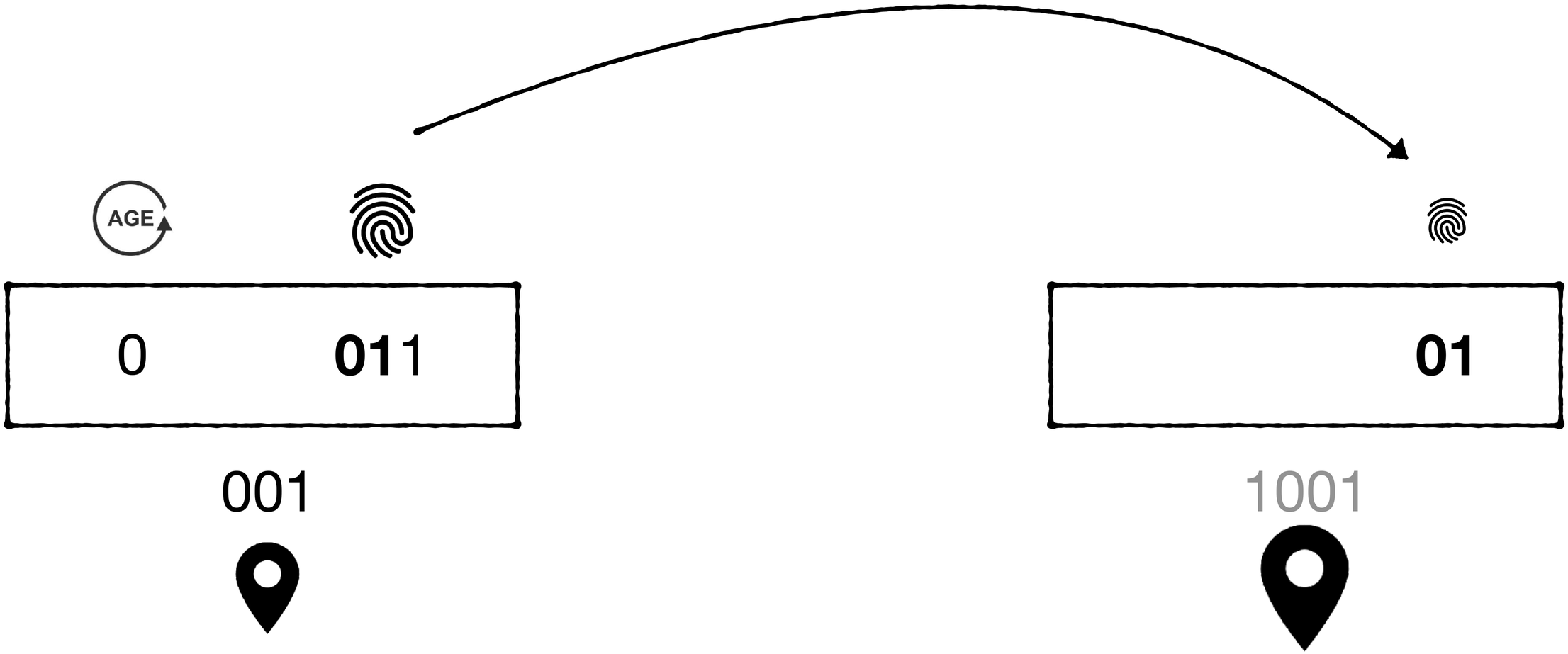
Fixed-length

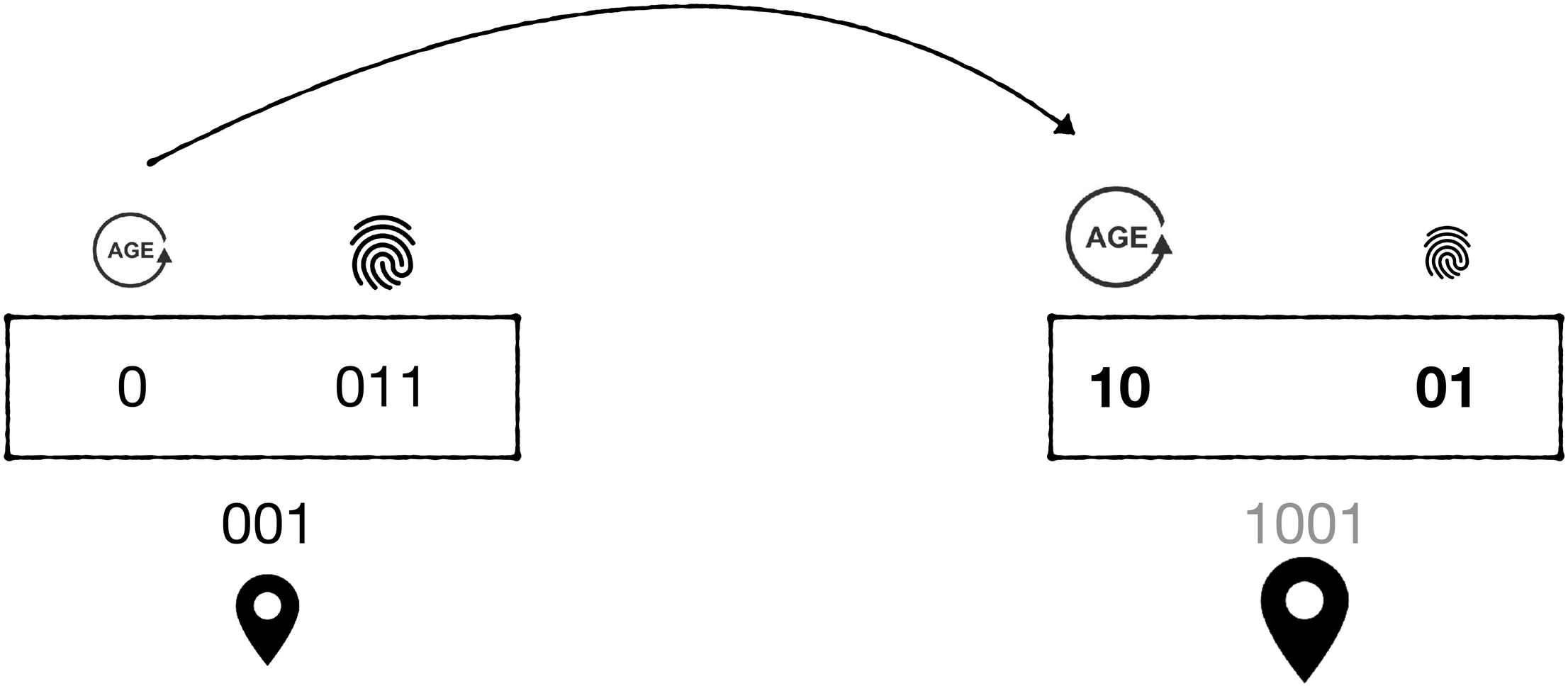


Expansion

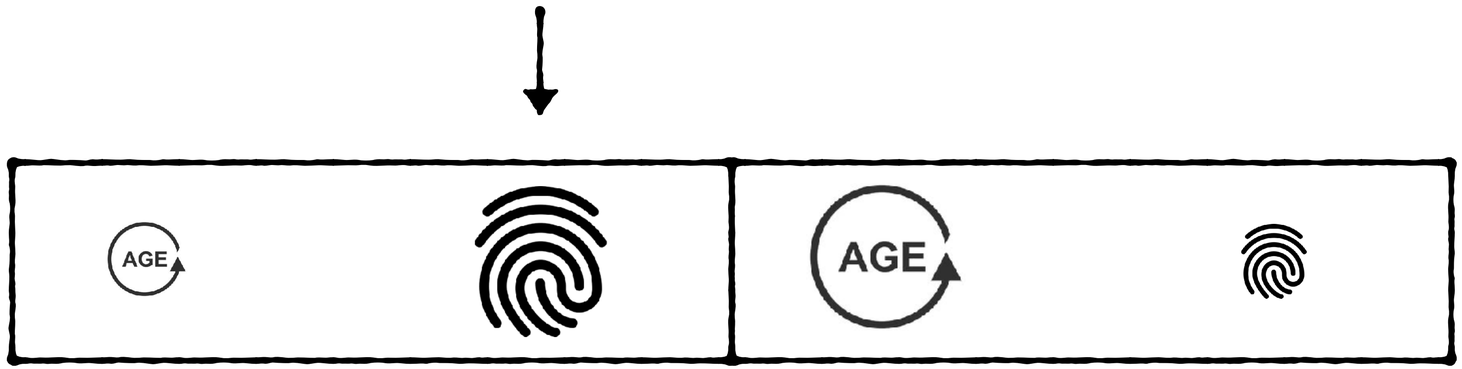

AGE
0
011
Expansion
001


AGE
0
011
001
1001

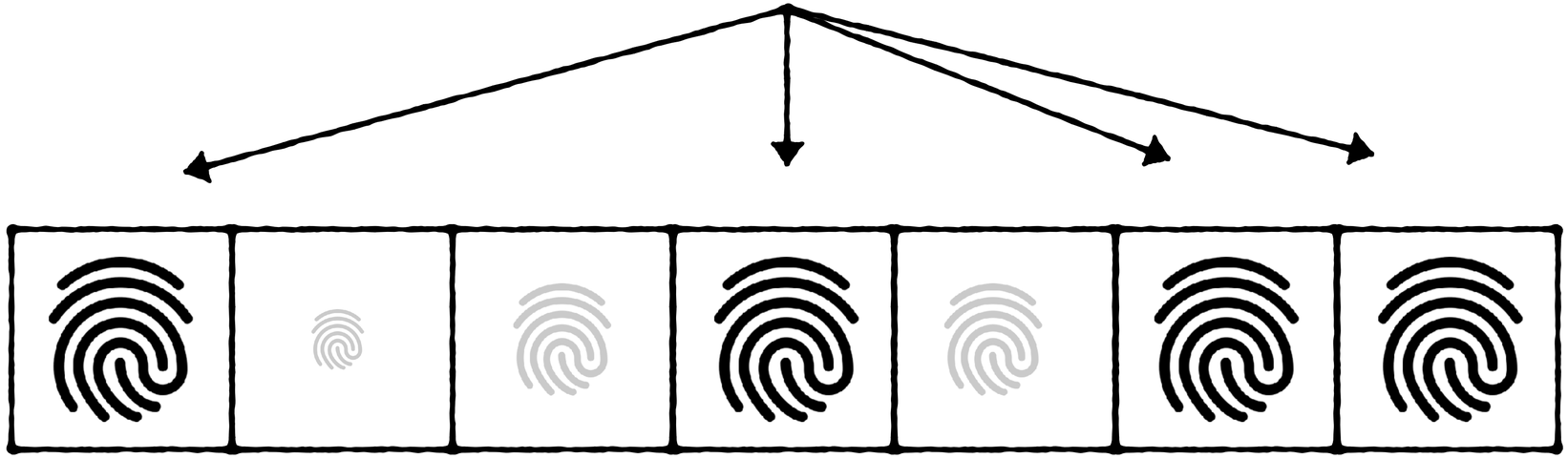

AGE
0 011
01
001
1001


AGE
0
011
001
AGE
10
01
1001

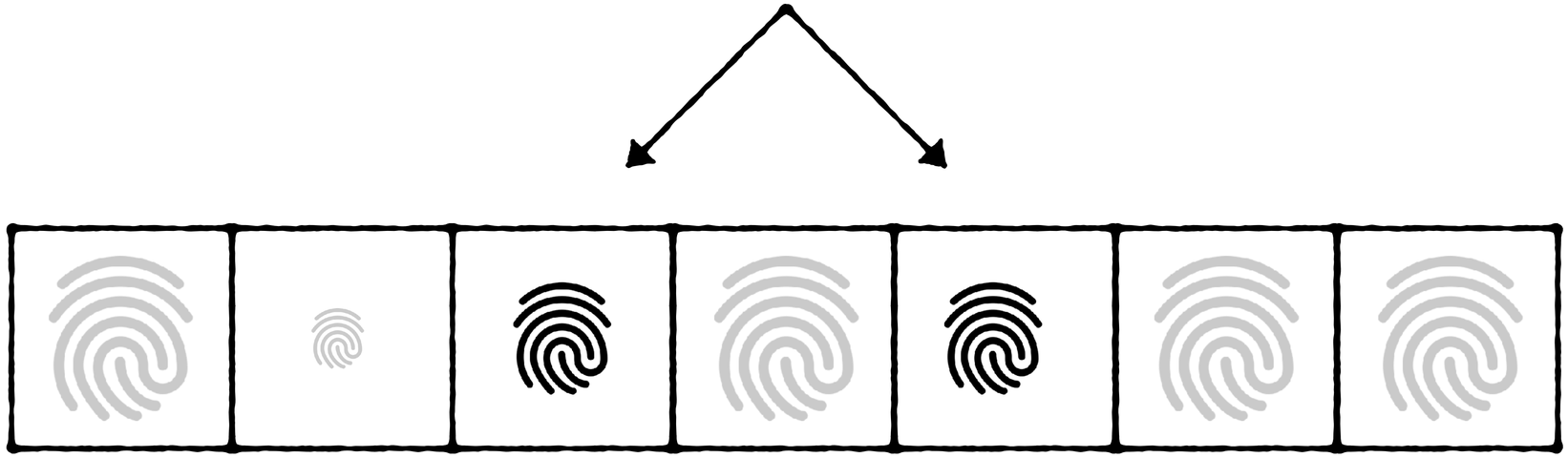
Longer fingerprints can be inserted after expansion



Half of entries have *F* bit fingerprints



Quarter have *F*-1 bit fingerprints

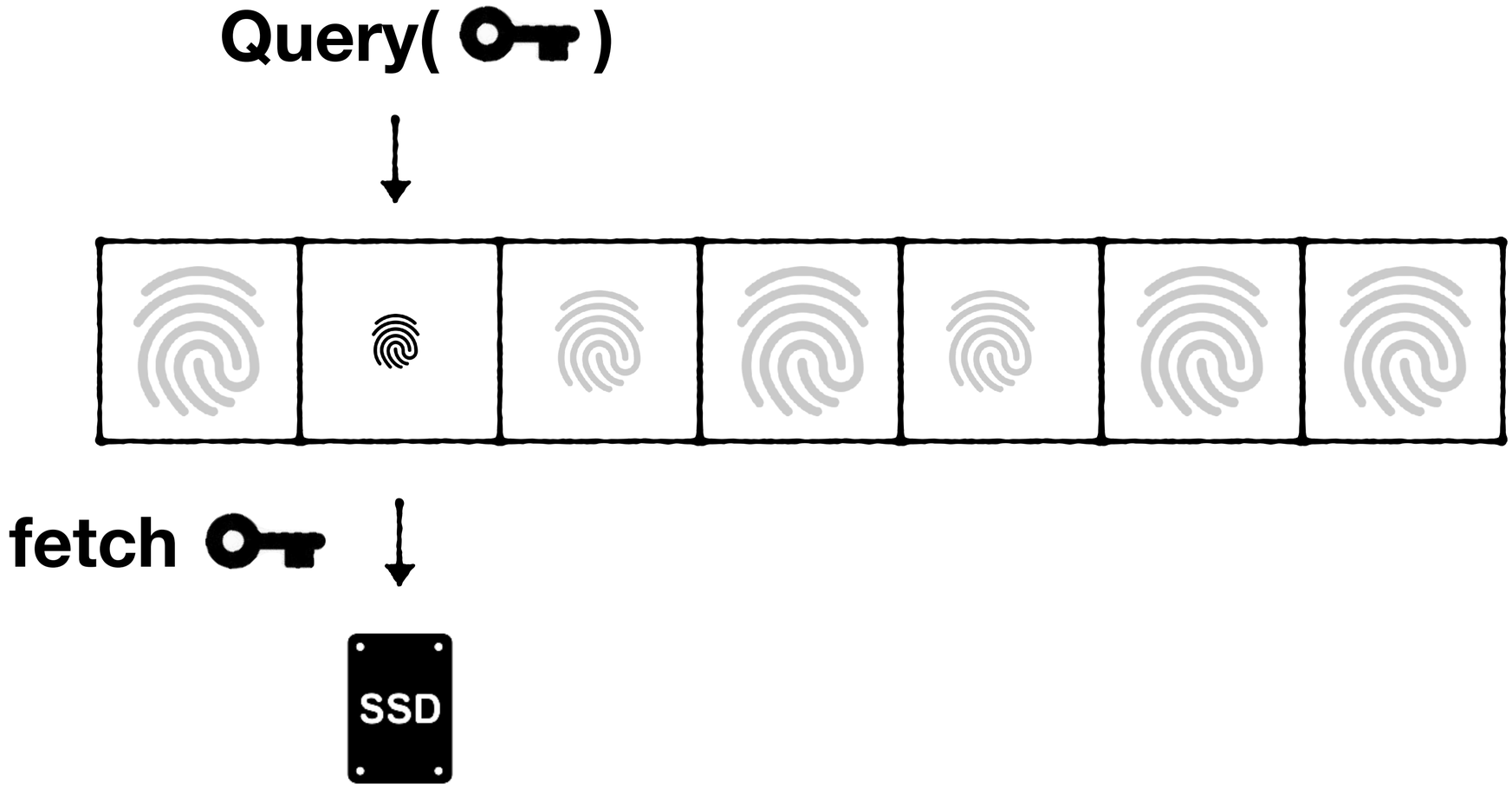


Eighth have *F*-2 bit fingerprints

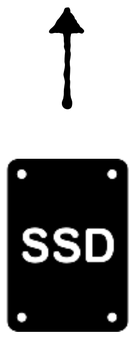
weighted false positive rate $\approx \log_2(N) \cdot 2^{-F}$

false positive rate = $\log_2(N) \cdot 2^{-M/N}$ < $N \cdot 2^{-M/N}$

with quotient filter


Query( )
fetch
SSD

Rehash(🔑) &
rejuvenate
fingerprint



Rehash(🔑) &
rejuvenate
fingerprint

SSD

FPR

$$\log N \cdot 2^{-M/N} \longrightarrow 2^{-M/N}$$

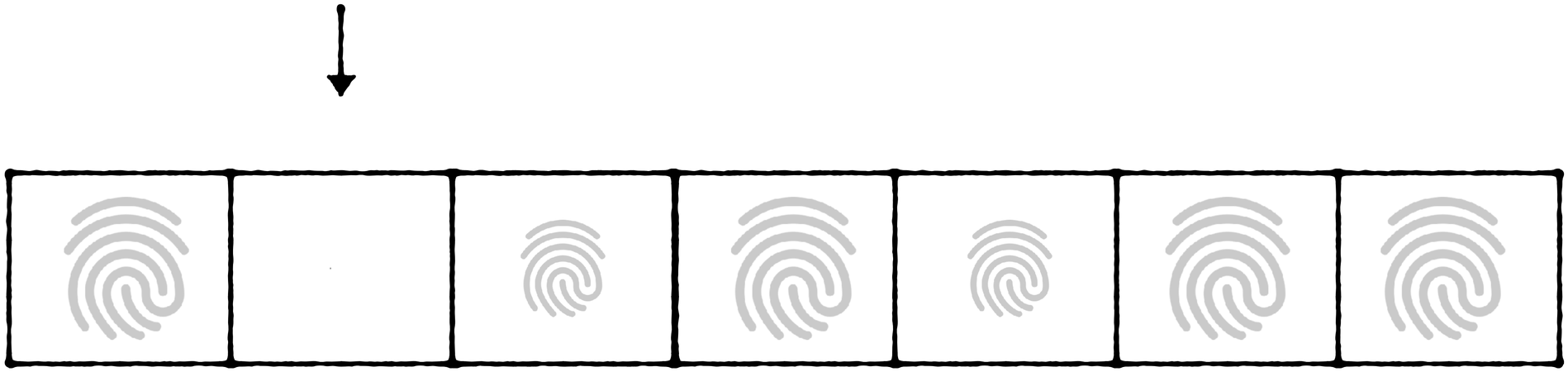
Increase slot width at rate of $\approx 2 \log_2 \log_2 N$

$$\text{FPR} \approx \log N \cdot 2^{-M/N}$$

$$\text{FPR} \approx \log N \cdot 2^{-M/N} - 2 \log_2 \log_2 N$$

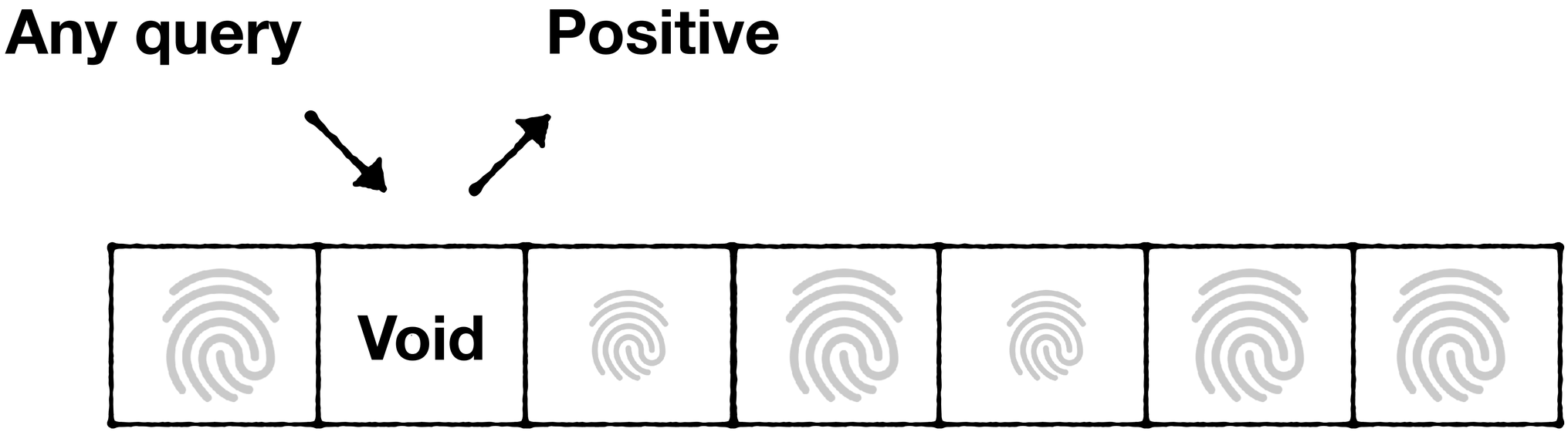
$$FPR \approx 2^{-M/N}$$

After *F* expansions, oldest fingerprints run out of bits

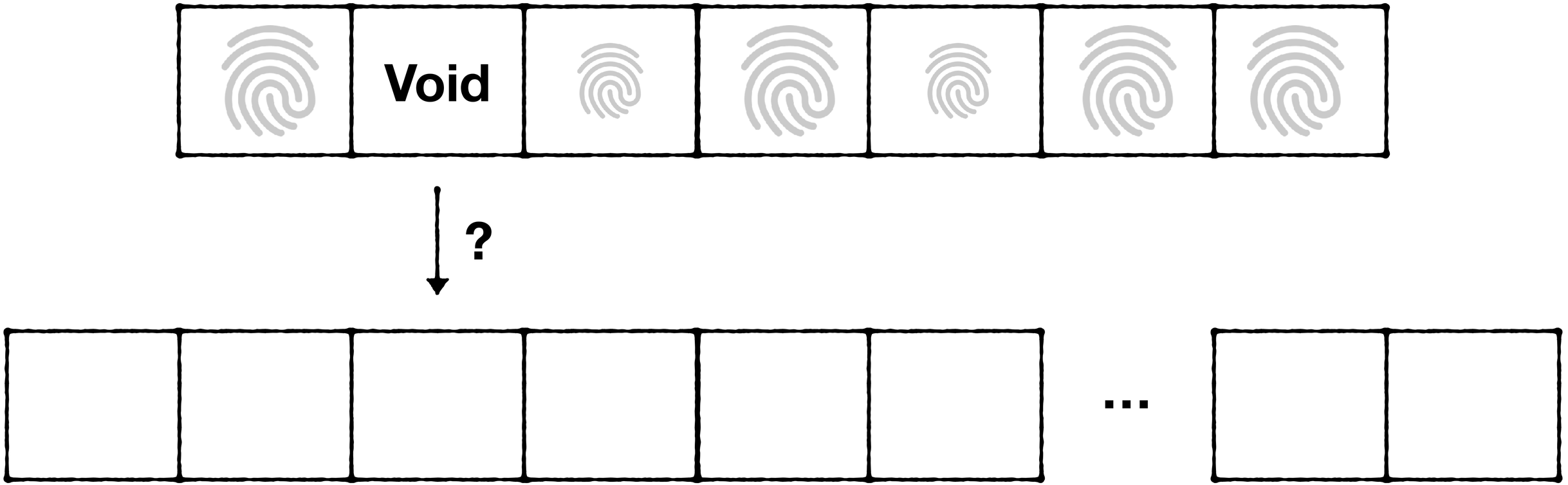


Unary padding occupies whole slot

Unary padding occupies whole slot

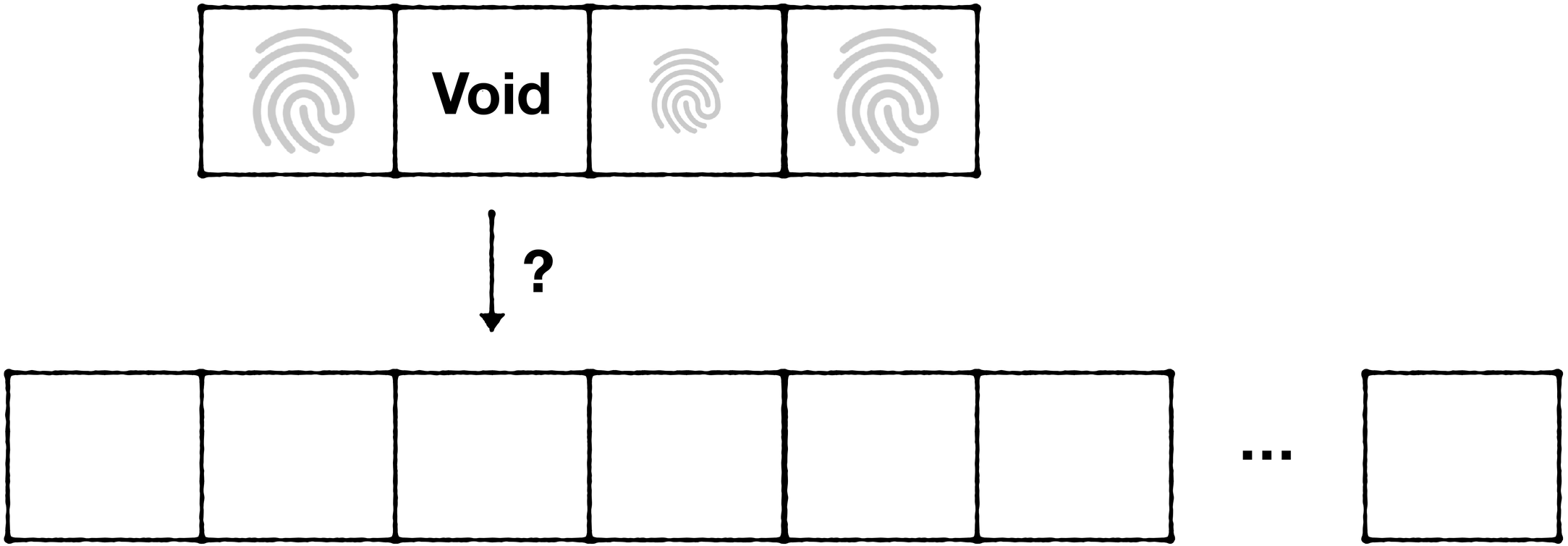

Any query
Positive
Void

How to continue expanding?

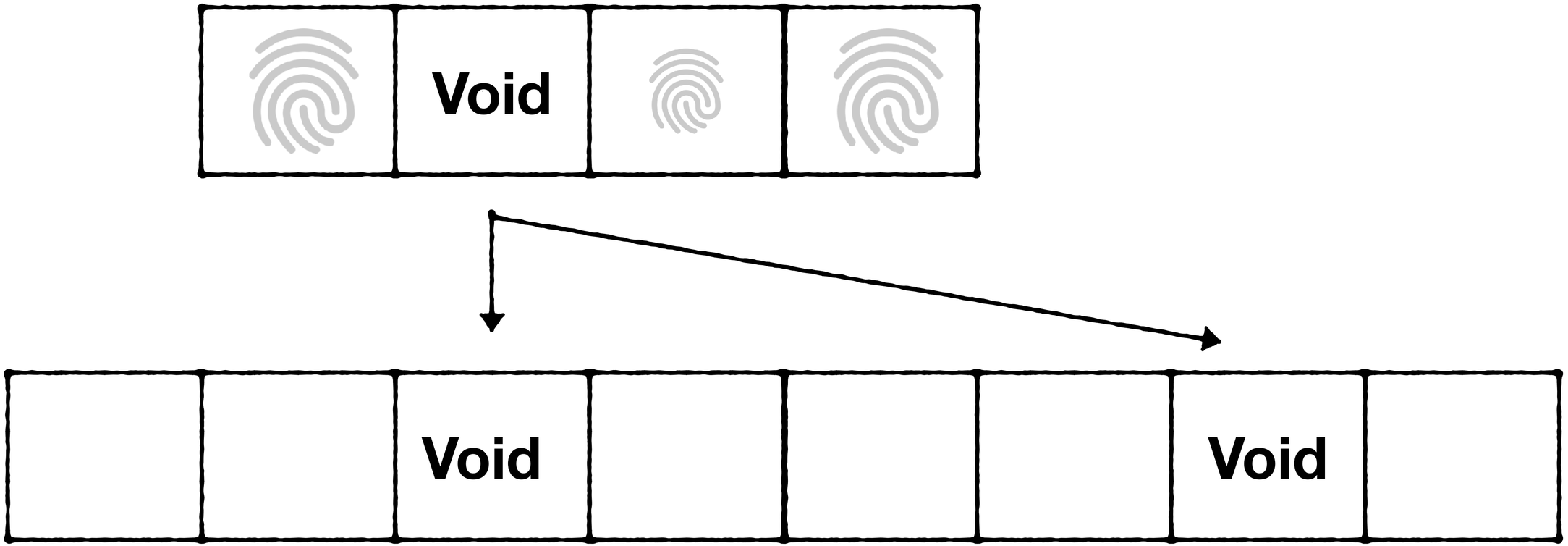


Aleph Filter: To Infinity in Constant Time

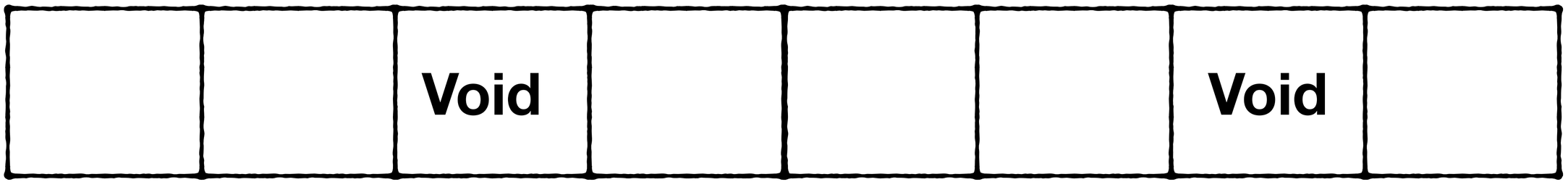
Niv Dayan, Ioana Bercea, Rasmus Pagh. VLDB 2024

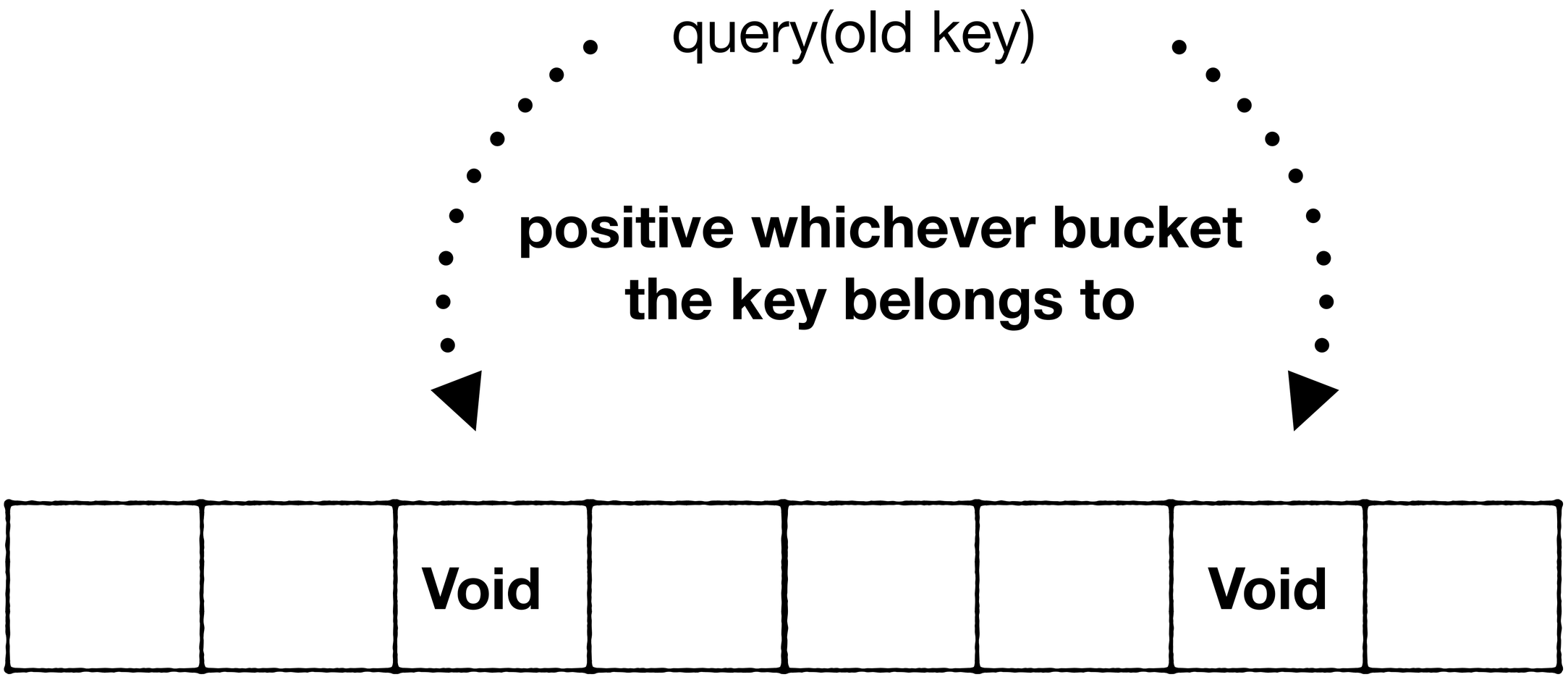


Duplicate

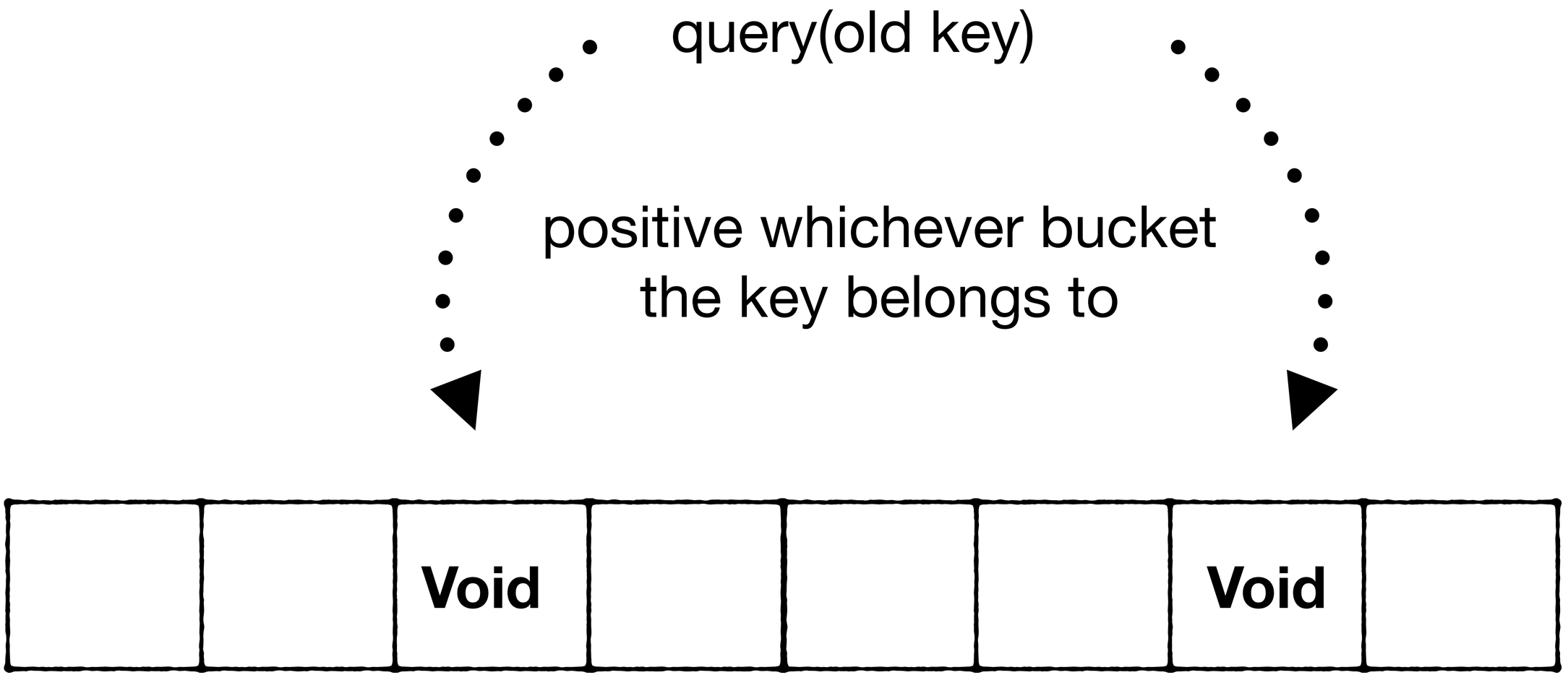


query(old key)



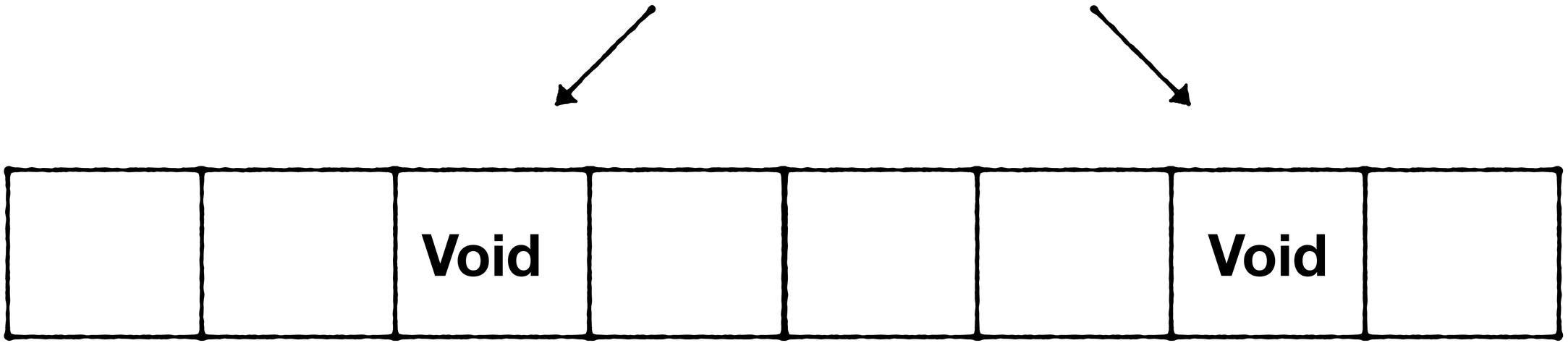

query(old key)
positive whichever bucket
the key belongs to
Void
Void

Expand Indefinitely with O(1) performance



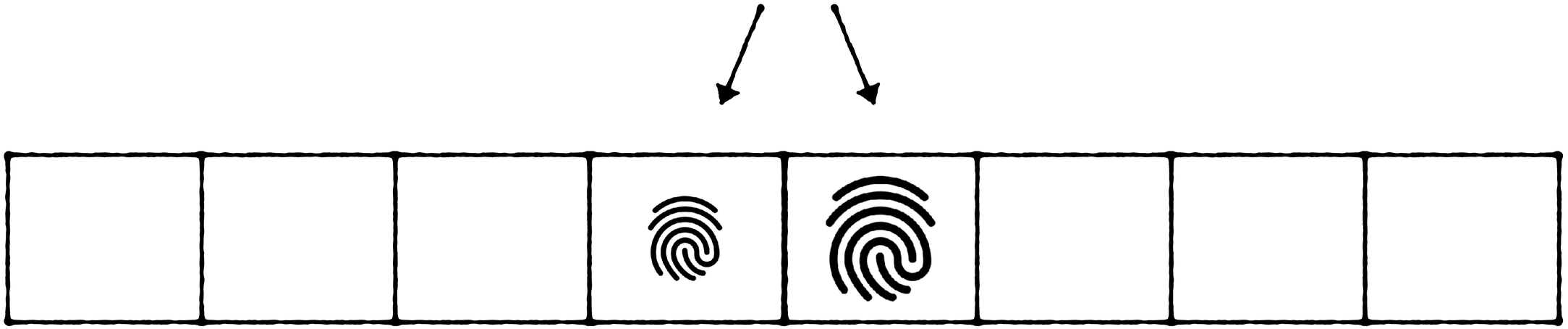
Expandable Filters Complicate Deletes

Identify how many void entries to remove



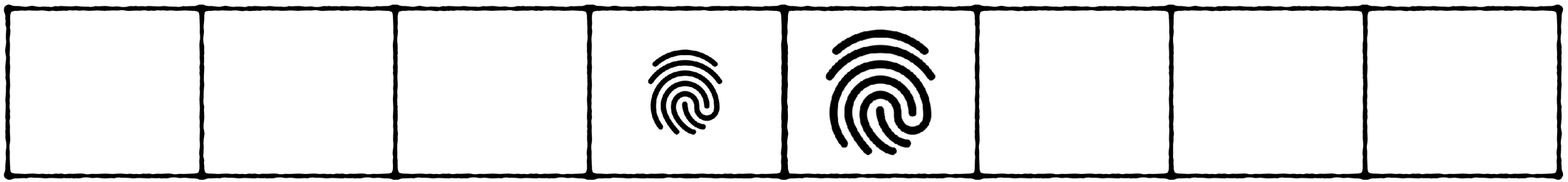
Expandable Filters Complicate Deletes

Multiple fingerprints of diff lengths may match key to delete



Expandable Filters Complicate Deletes

Solutions exist in the papers :)



Two volunteers needed for next week's presentations

Thank you!