Dynamic Filters (Quotient & InfiniFilter)

Research Topics in Database Management

Niv Dayan

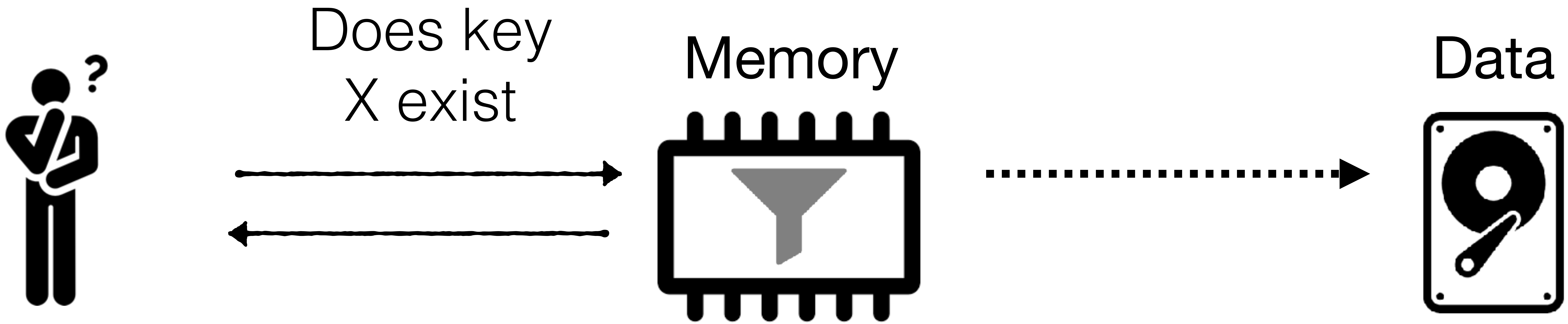
What is a Filter?

Does X exist? →

Set

X Y Z

If key X does not exist

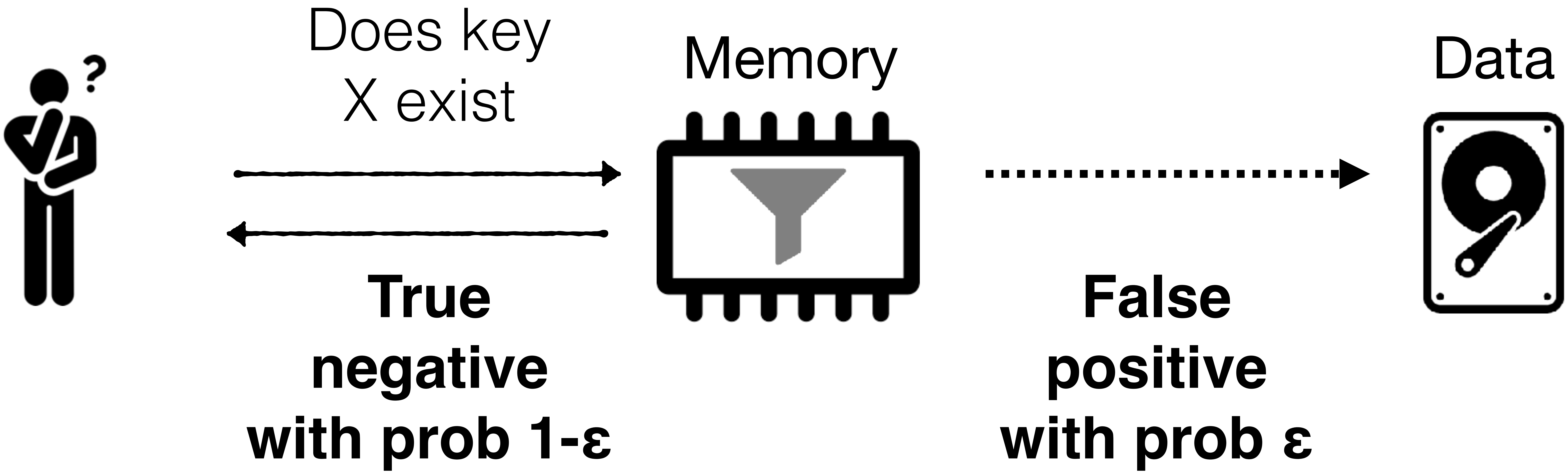


Does key X exist

Memory

Data

If key X does not exist



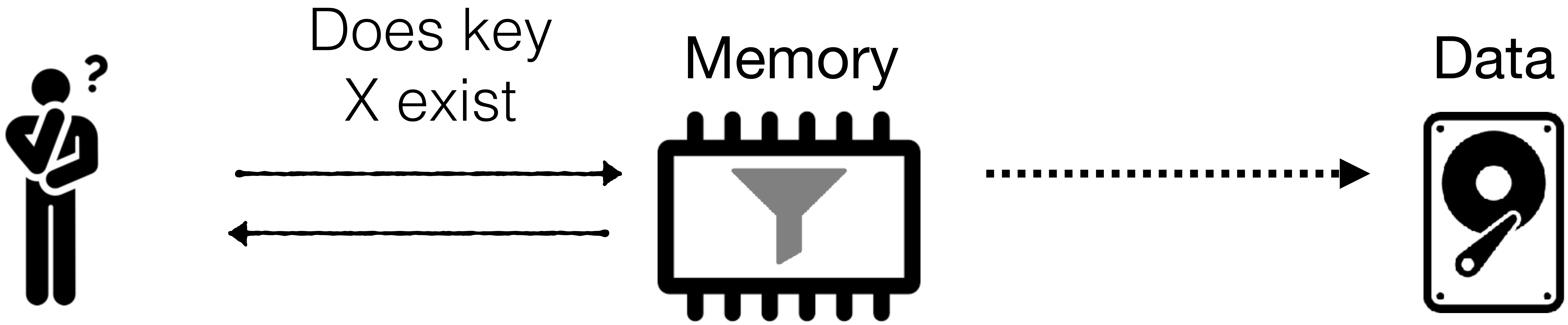
Does key X exist

Memory

Data

True negative with prob 1-ε

False positive with prob ε

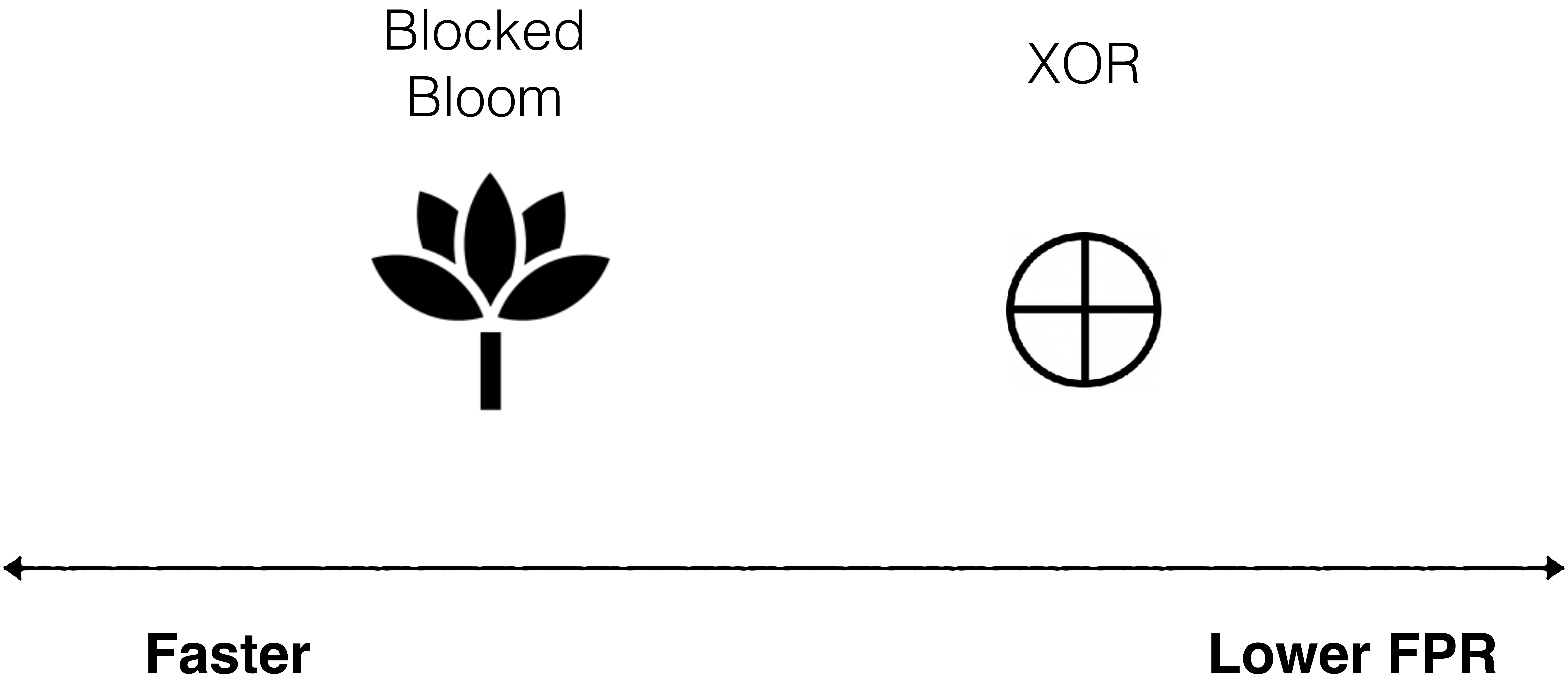


Does key X exist

Memory

Data

Saves storage accesses & network hops



Blocked Bloom

XOR

Faster

Lower FPR

Blocked
Bloom

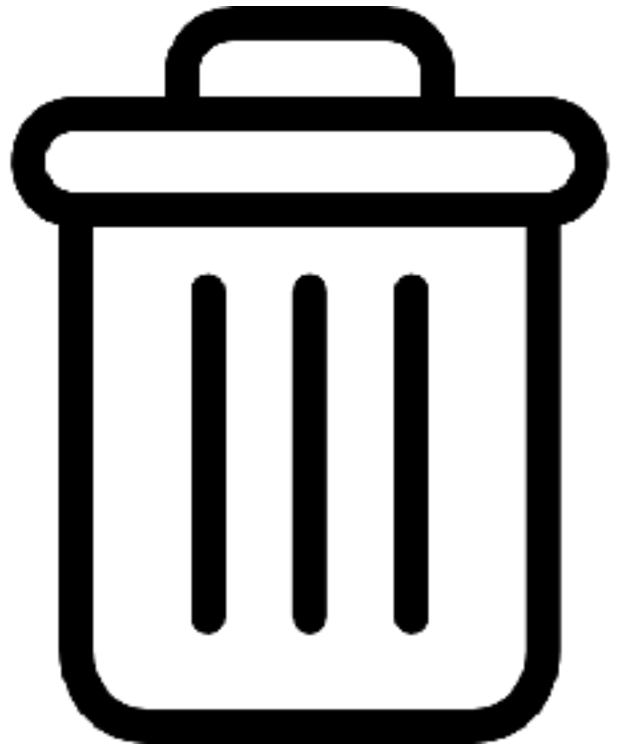
XOR

⊕

Static - no deletes or resizing

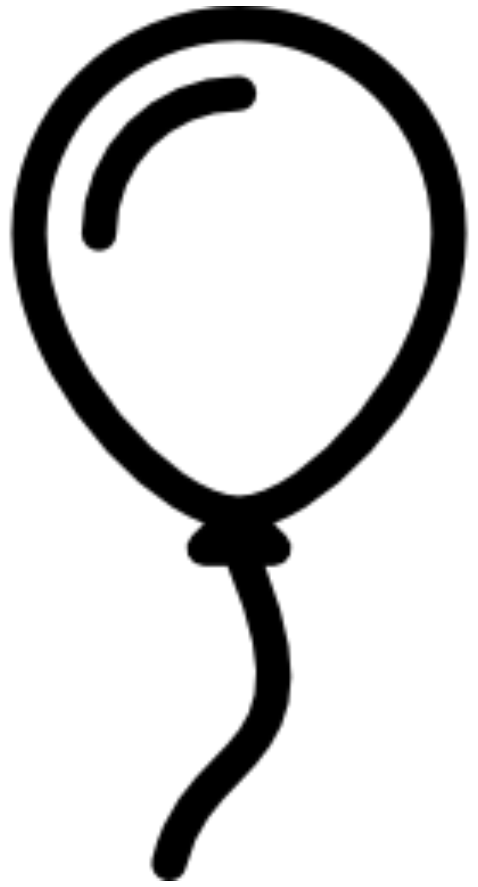
Supporting Dynamic Data

Supporting Dynamic Data



Deletes

(First hour)



Resizing

(Second hour)

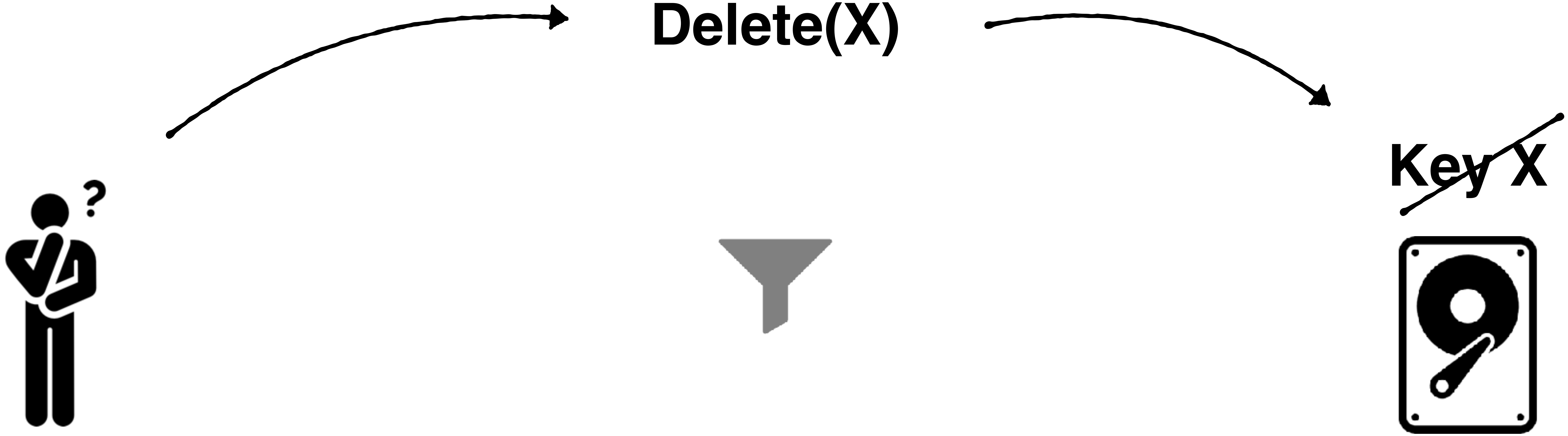
Why Support Deletes?

Why Support Deletes?

Query(X)

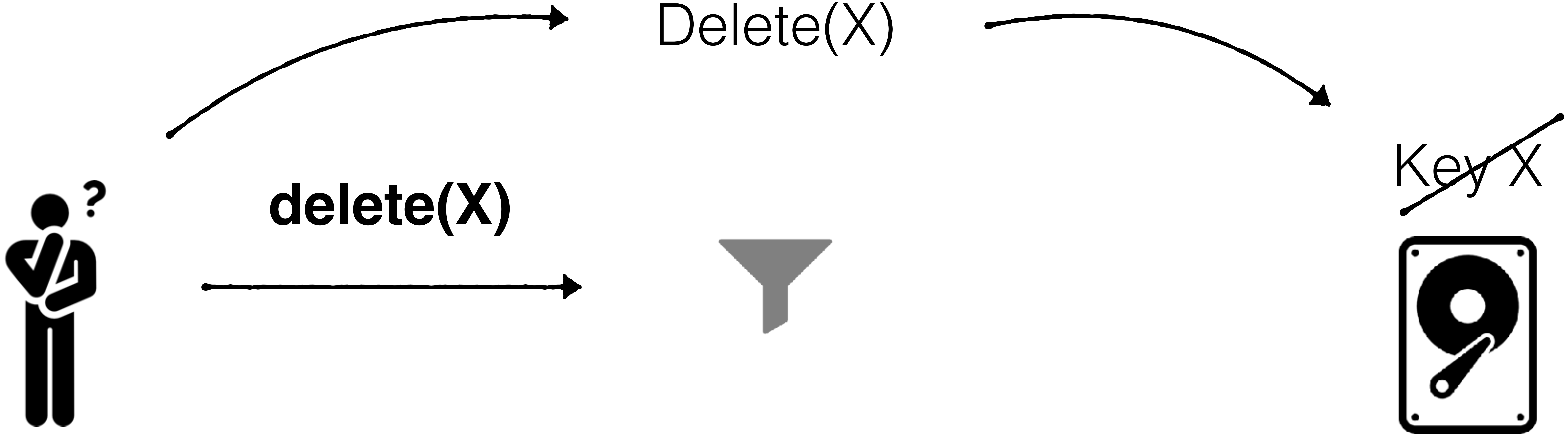
True positive

Key X



Delete(X)

~~**Key X**~~



Delete(X)

~~Key X~~

delete(X)

Why should we also delete from filter?

Desired Outcome

Query(X)

Negative

~~Key X~~

Desired Outcome

Query(X)

Negative

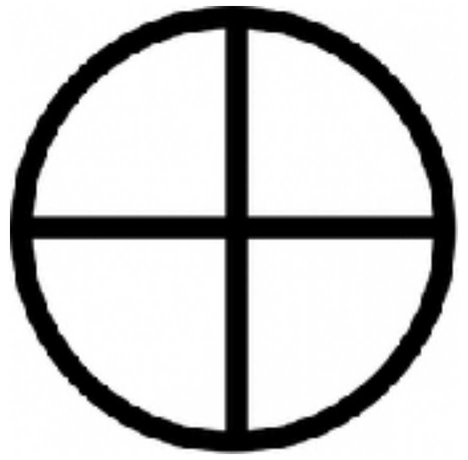
~~Key X~~

Only possible if filter supports deletes

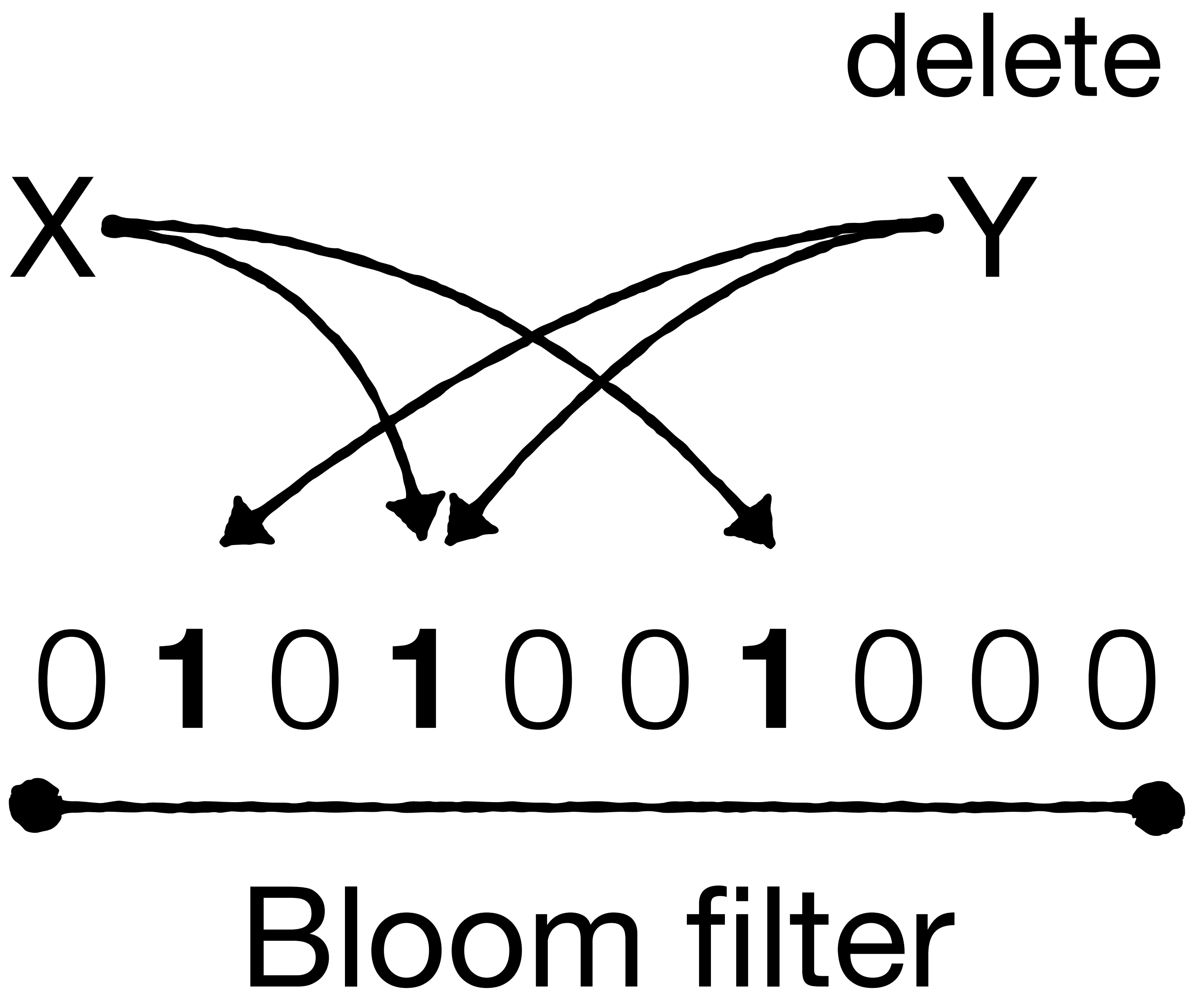
Why do last week's filters not support deletes?

Bloom

XOR

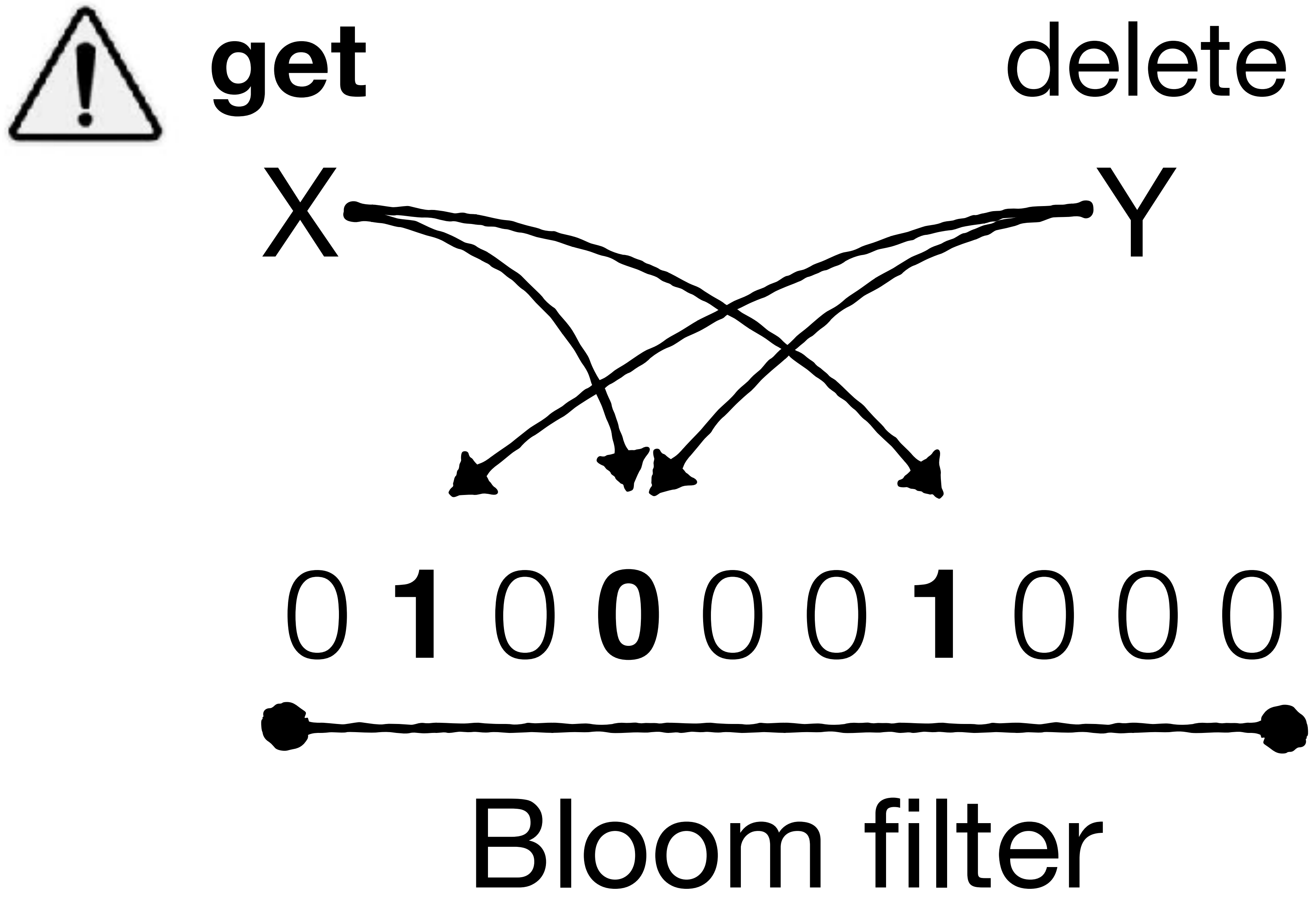


Multiple keys may map to each bit



Multiple keys may map to each bit

Setting bits back to 0s can lead to false negatives



⚠ **get** delete

X Y

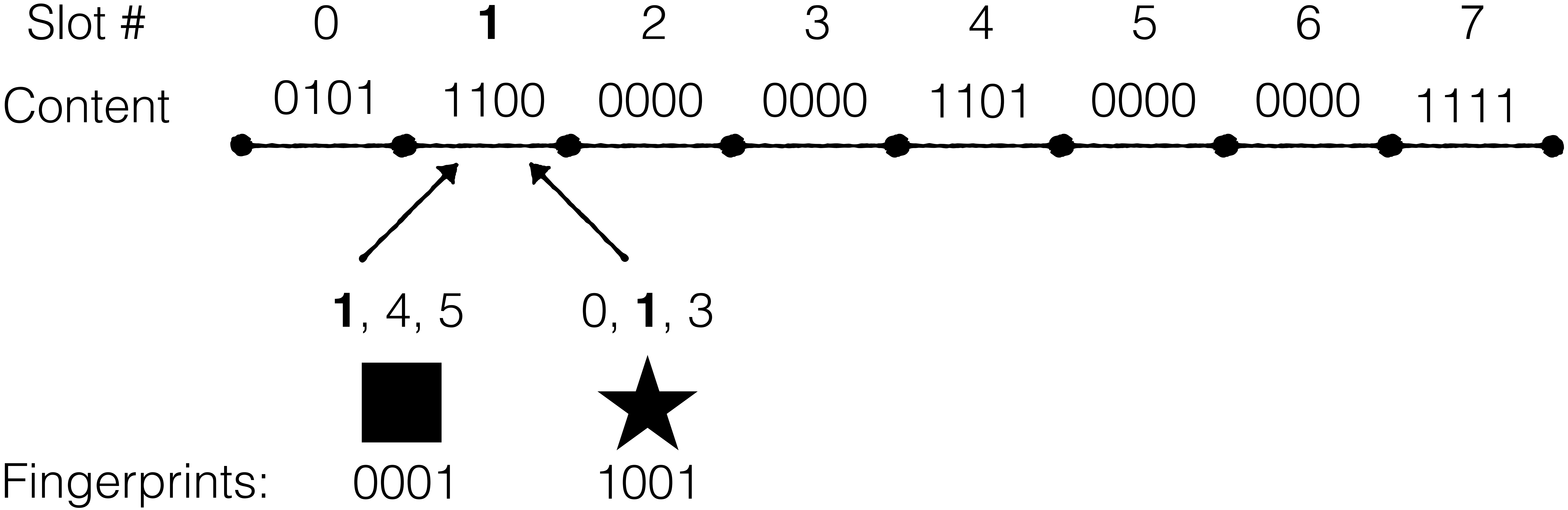
0 **1** 0 **0** 0 0 **1** 0 0 0

Bloom filter

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

XOR filter

Multiple keys share slots



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

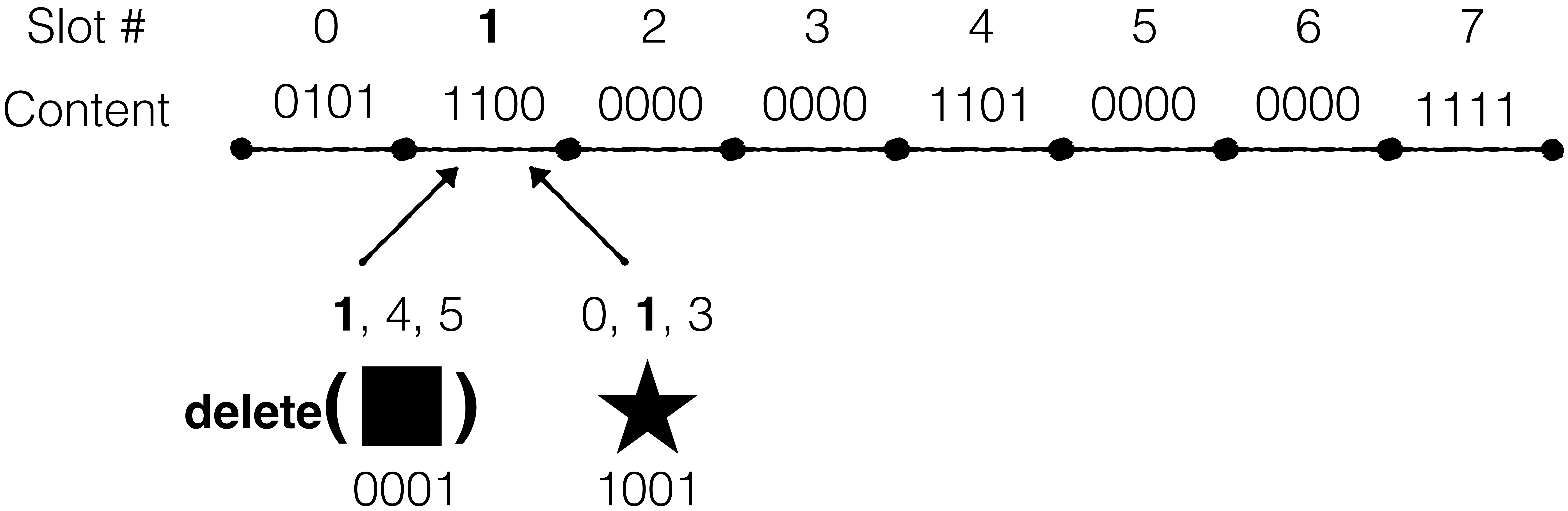
1, 4, 5

0, **1**, 3

■ ★

Fingerprints: 0001 1001

Multiple keys share slots



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

1, 4, 5

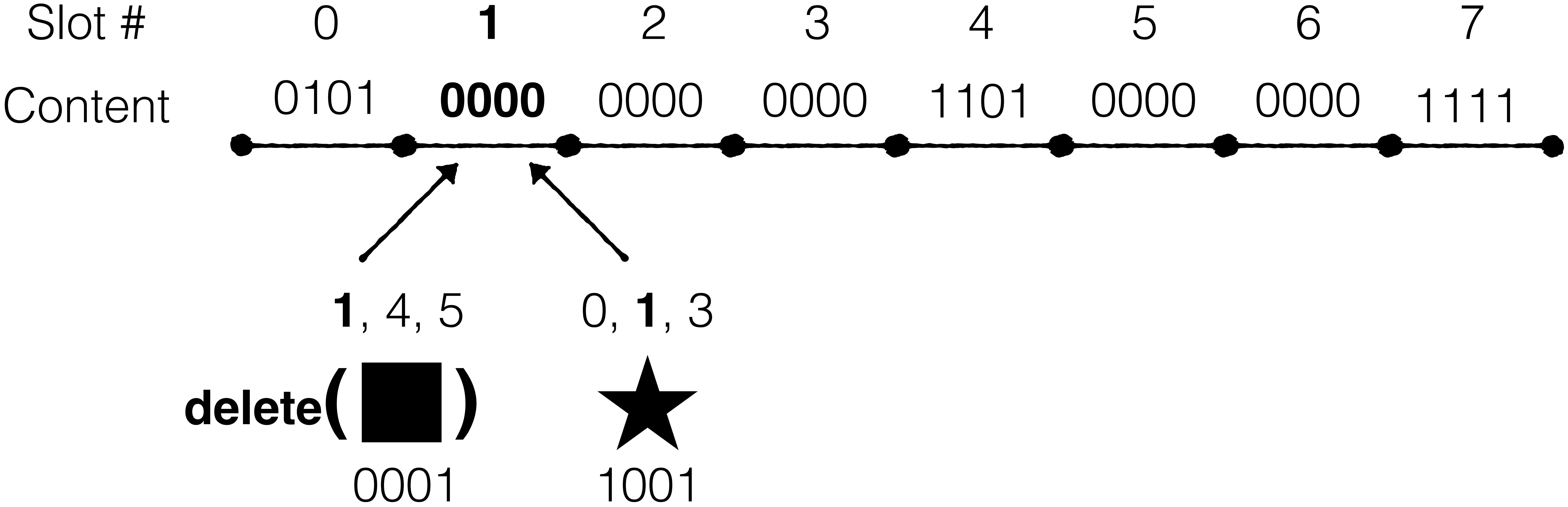
delete(■)

0001

0, **1**, 3

★

1001



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

1, 4, 5

0, **1**, 3

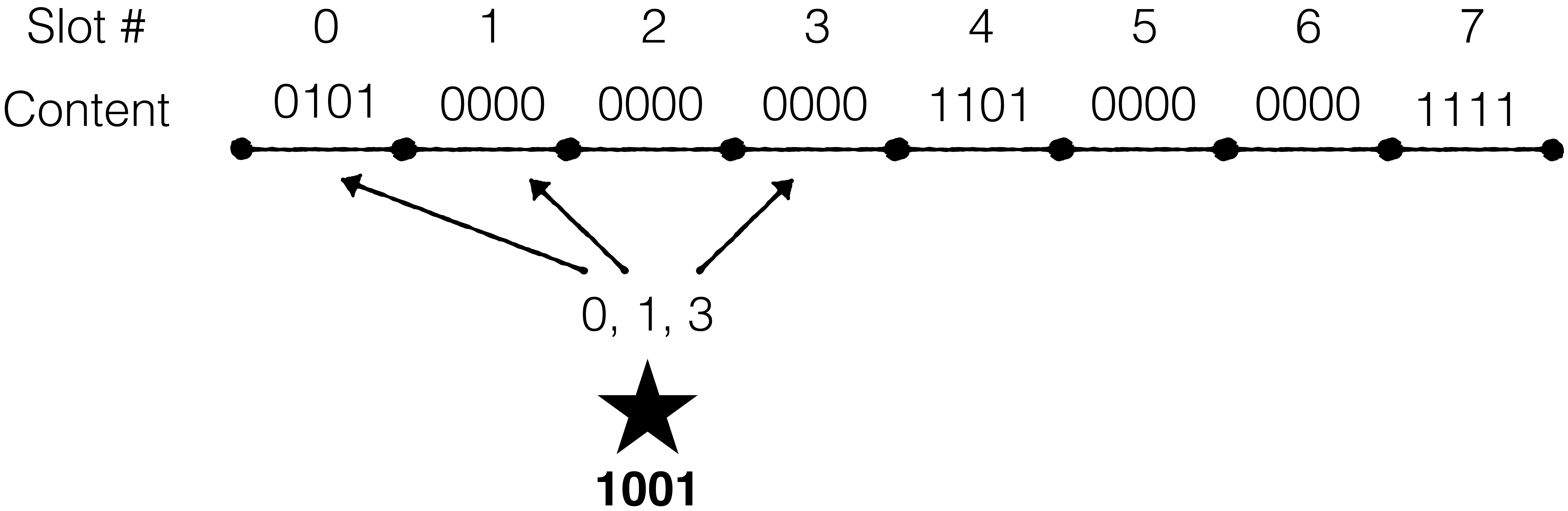
delete(■)

★

0001

1001

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



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

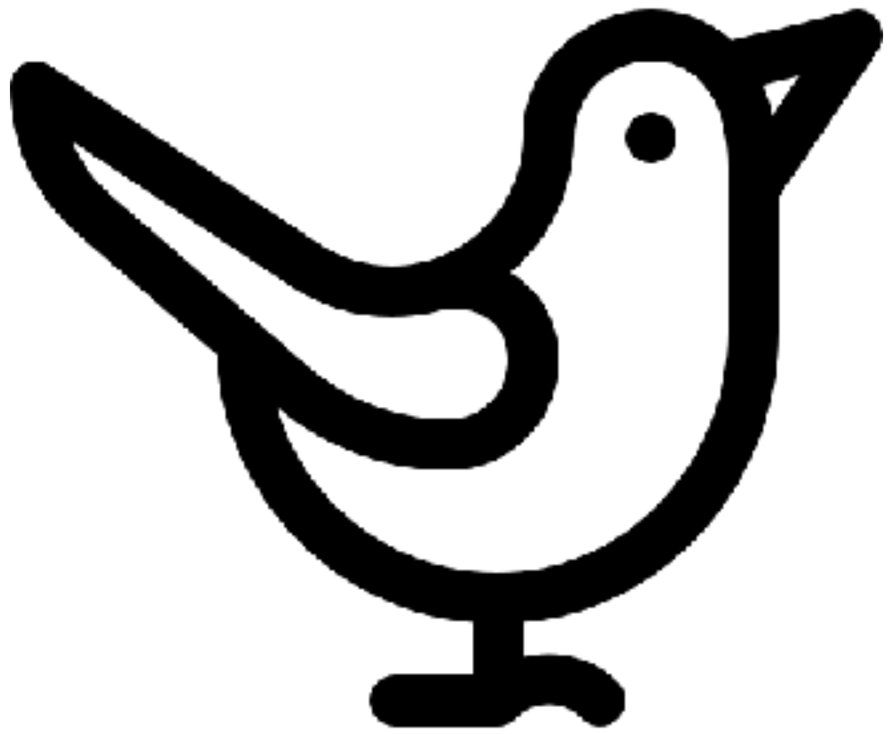
0, 1, 3

★

1001

How to support deletes without false negatives?

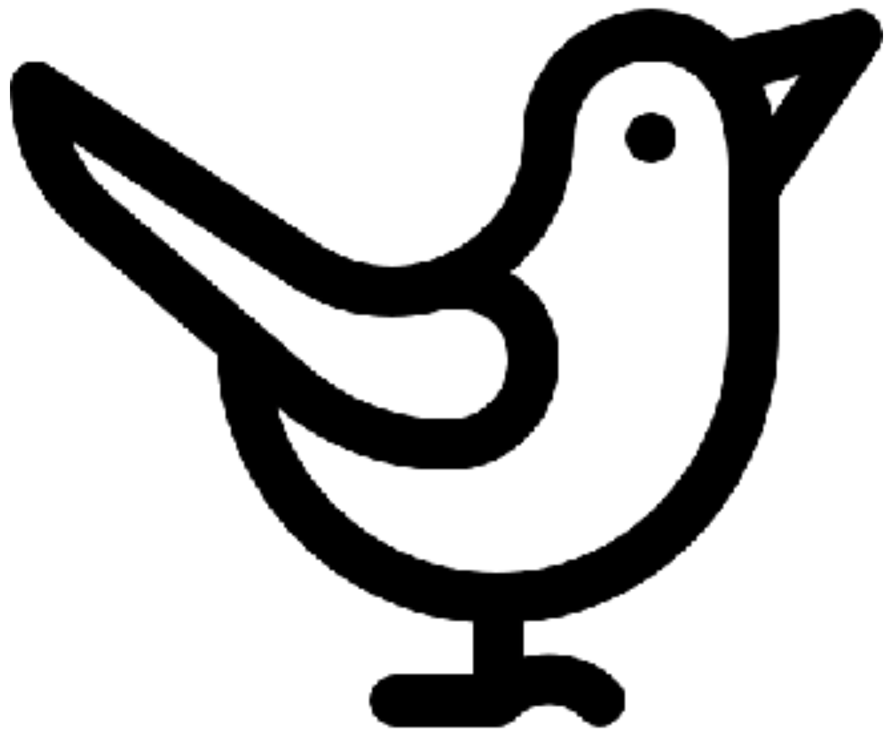
How to support deletes without false negatives?



Cuckoo Filters

(Last semester)

How to support deletes without false negatives?



Cuckoo Filters

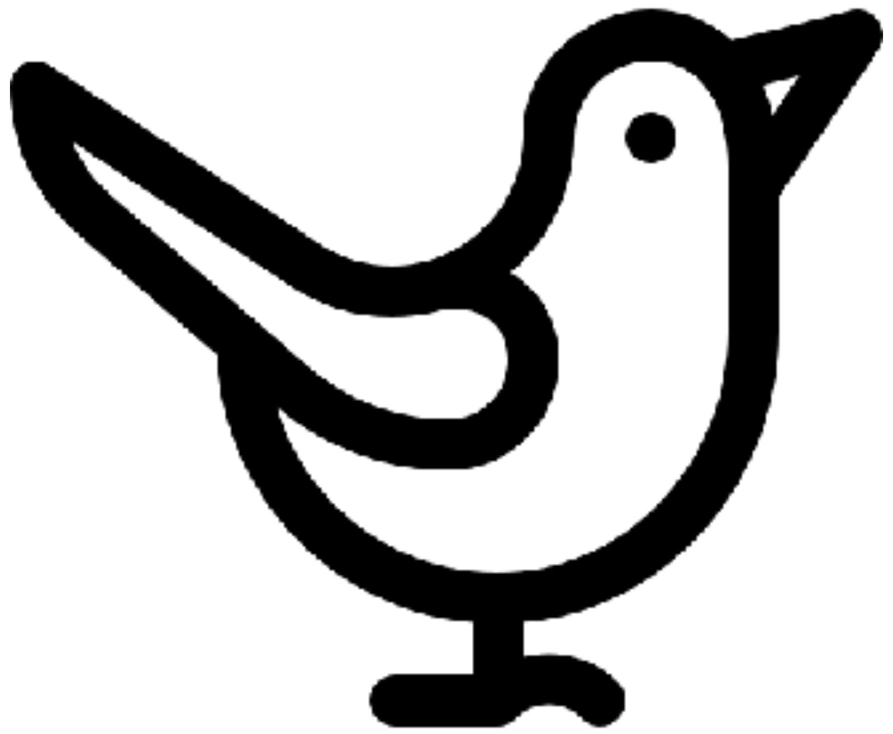
(Last semester)

%

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.

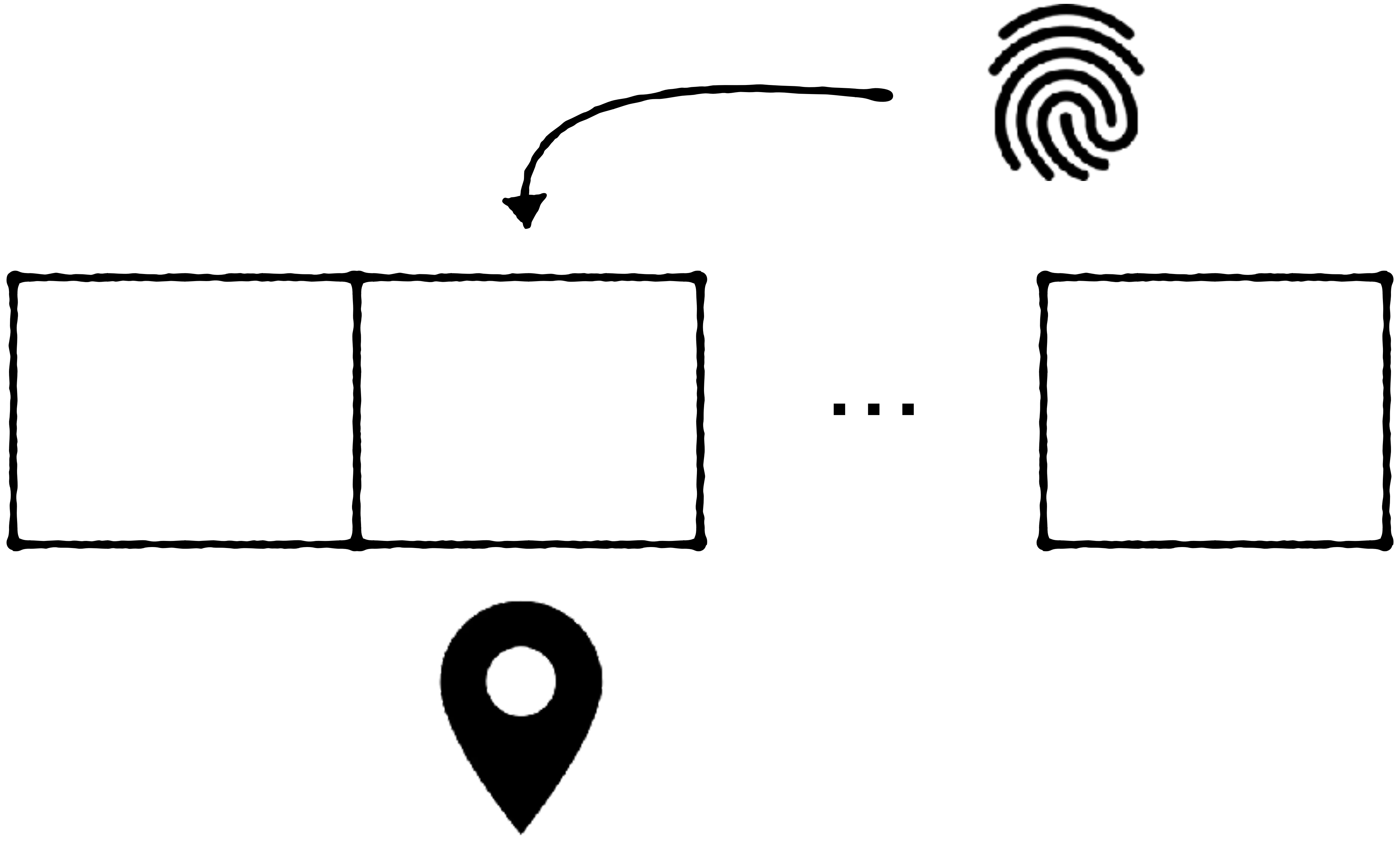
Vector Quotient Filters: Overcoming the Time/Space Trade-Off in Filter Design. SIGMOD 2021.

Uses SIMD & less tunable

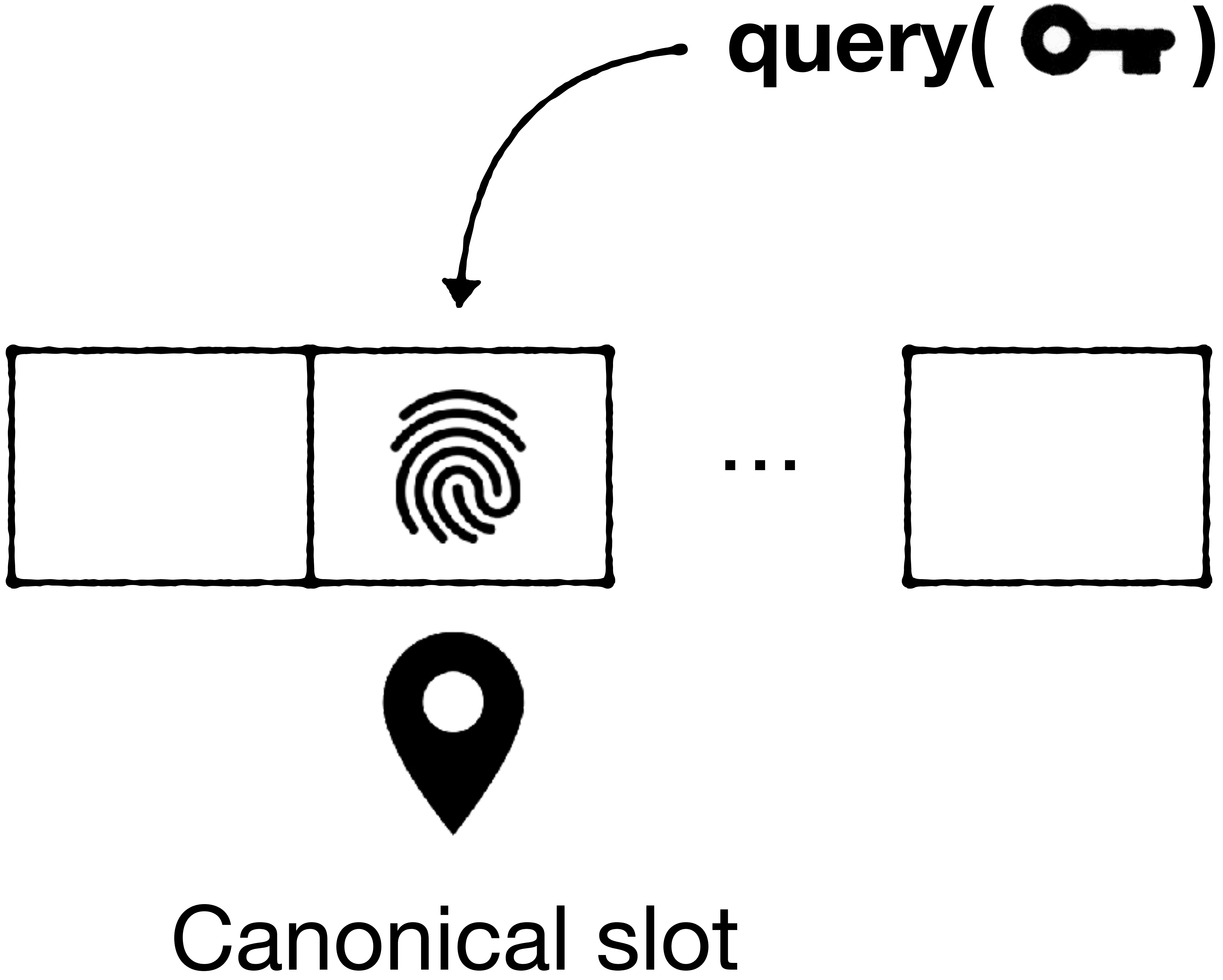
Our focus

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

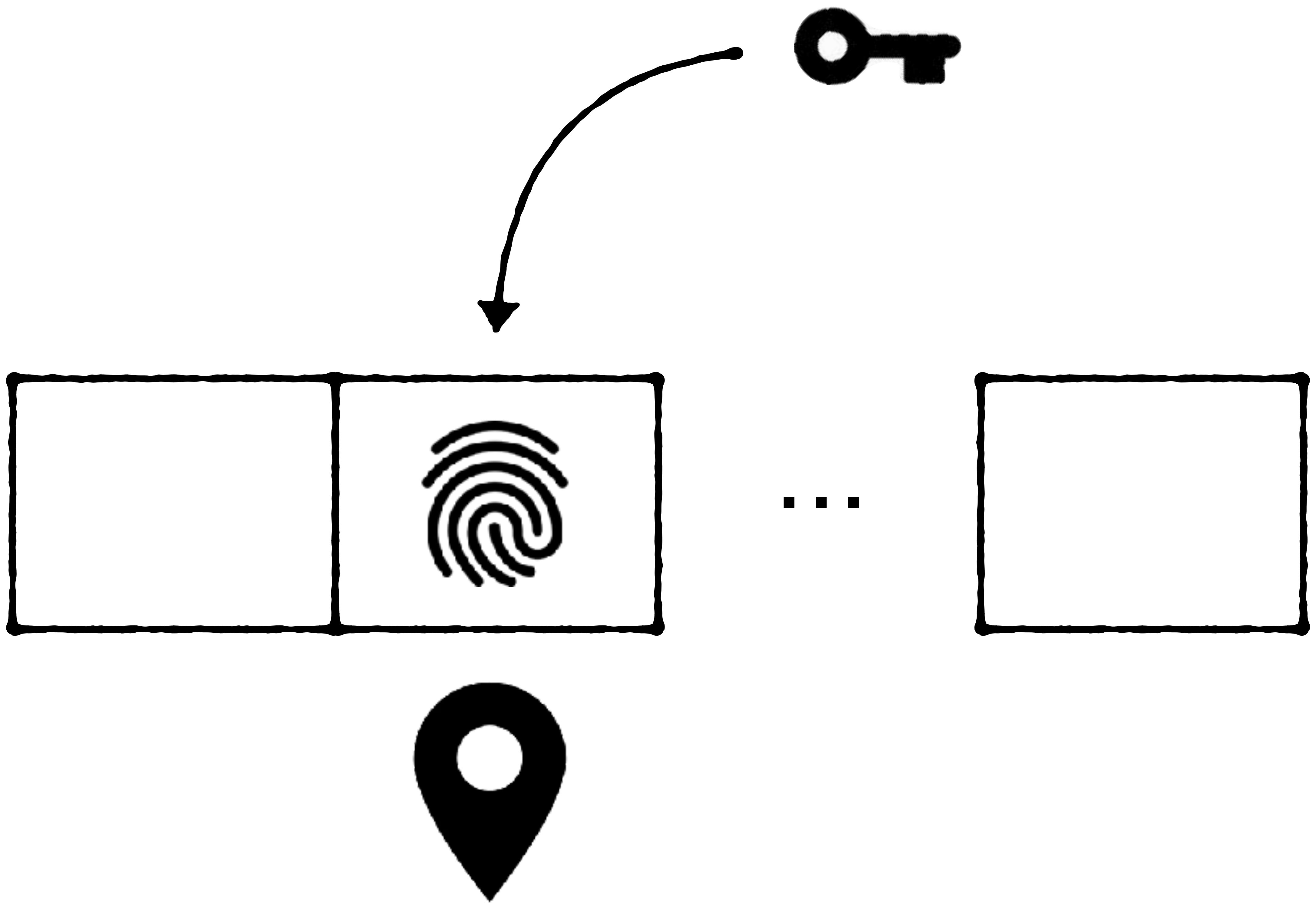


query(🔑)

…

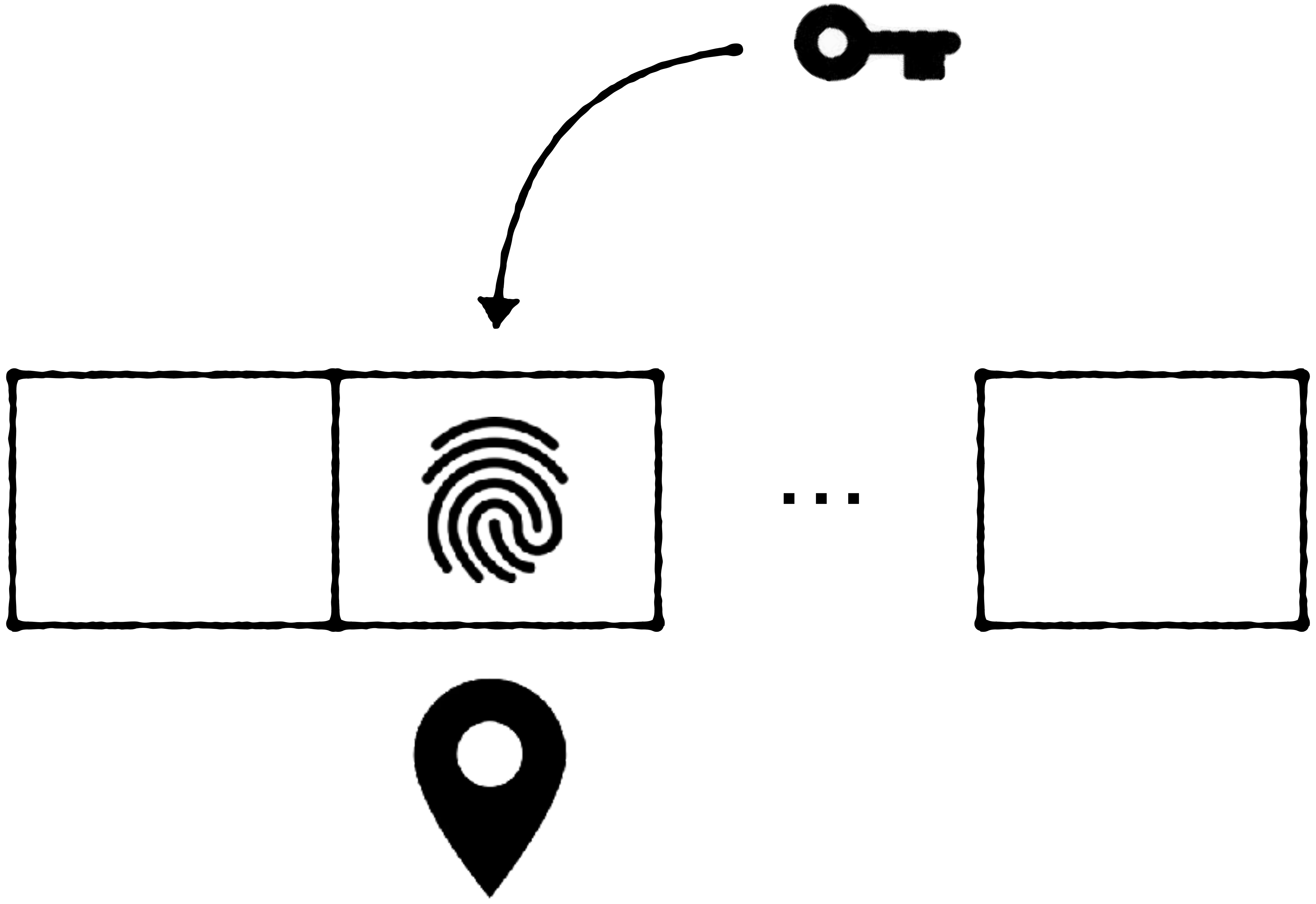
Canonical slot

Each inserted fingerprint corresponds to exactly one entry

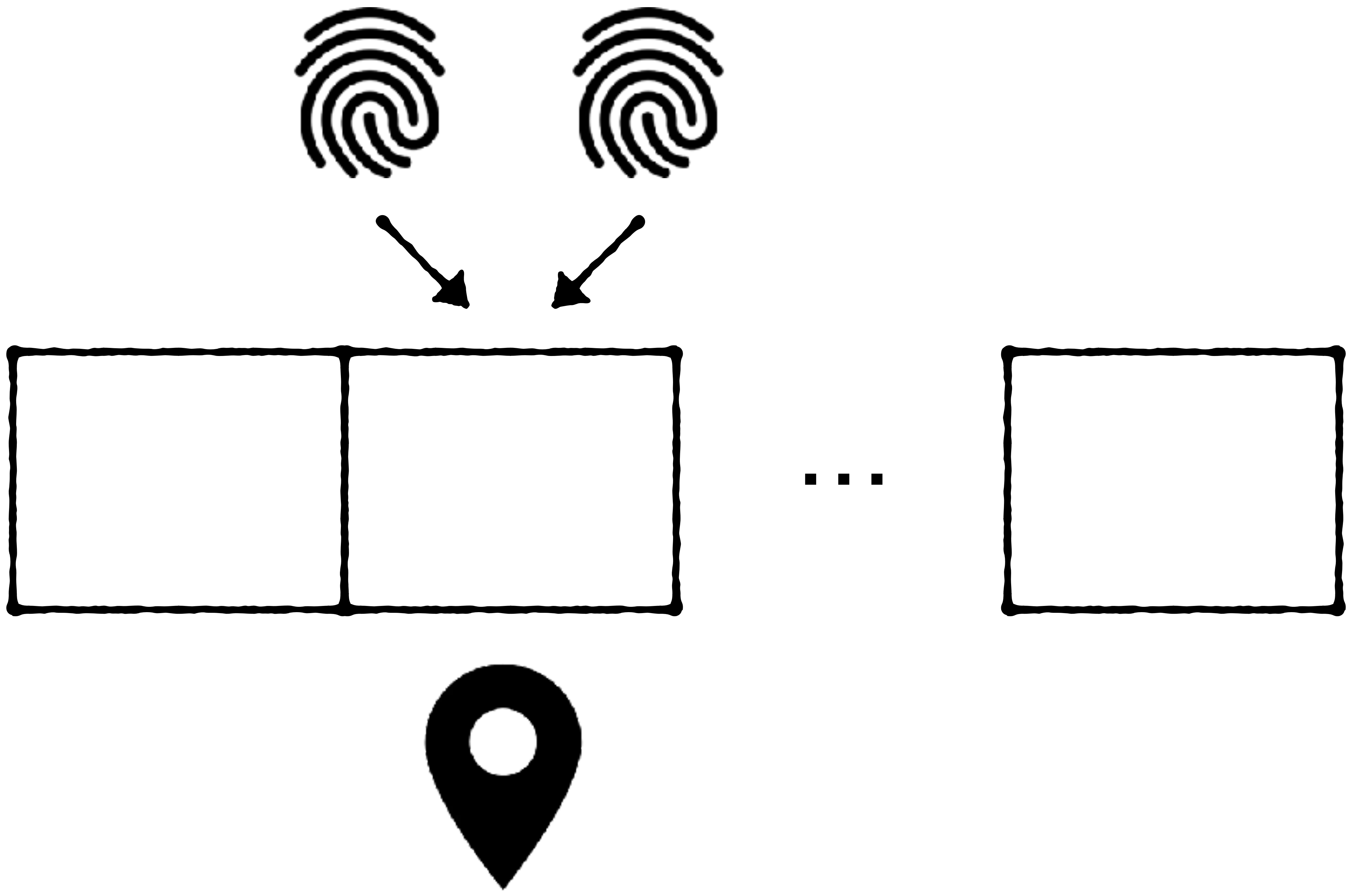


Each inserted fingerprint corresponds to exactly one entry

Removing it won’t introduce false negatives for other entries

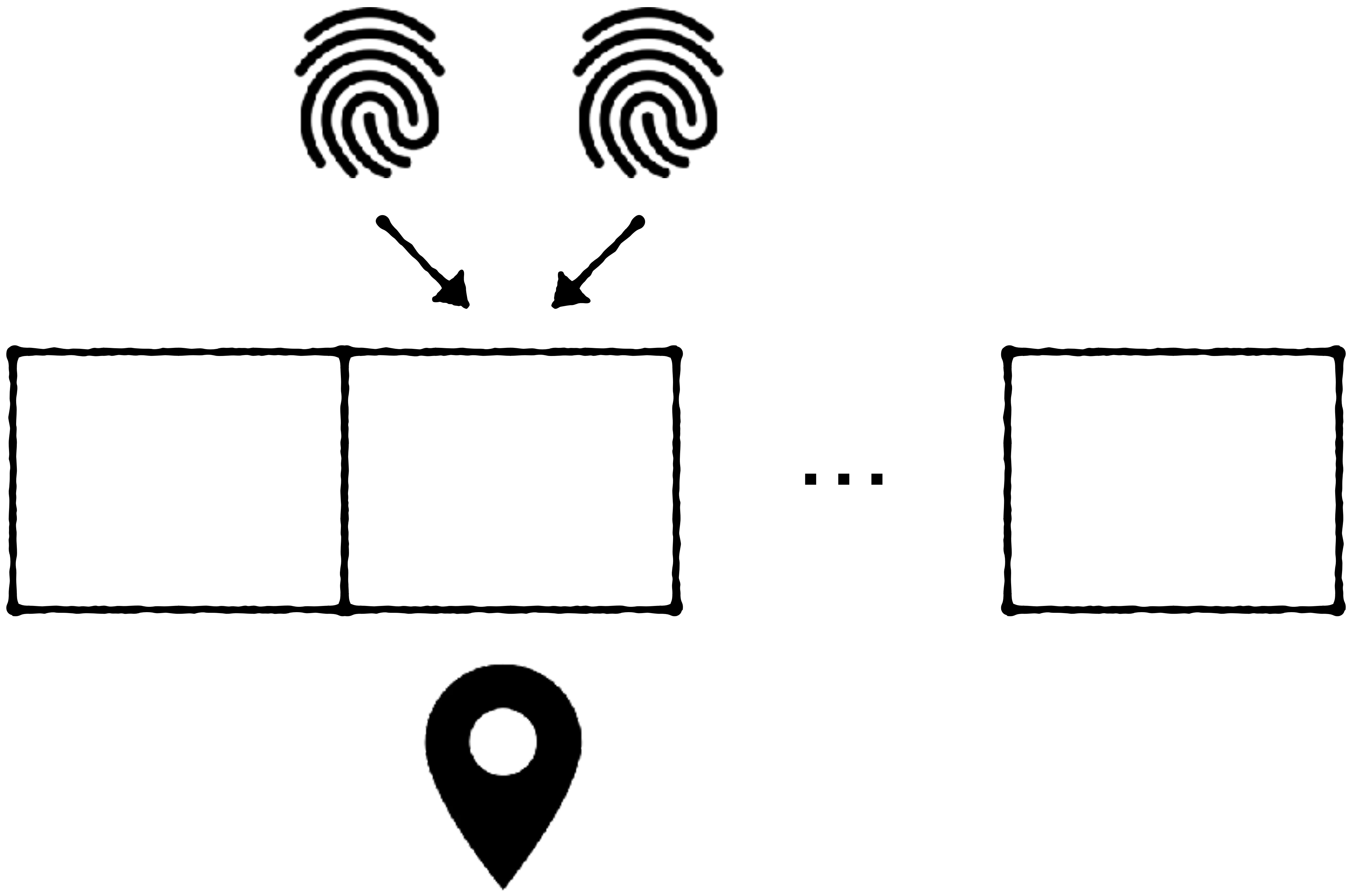


Hash collisions - multiple fingerprints map to same canonical slot



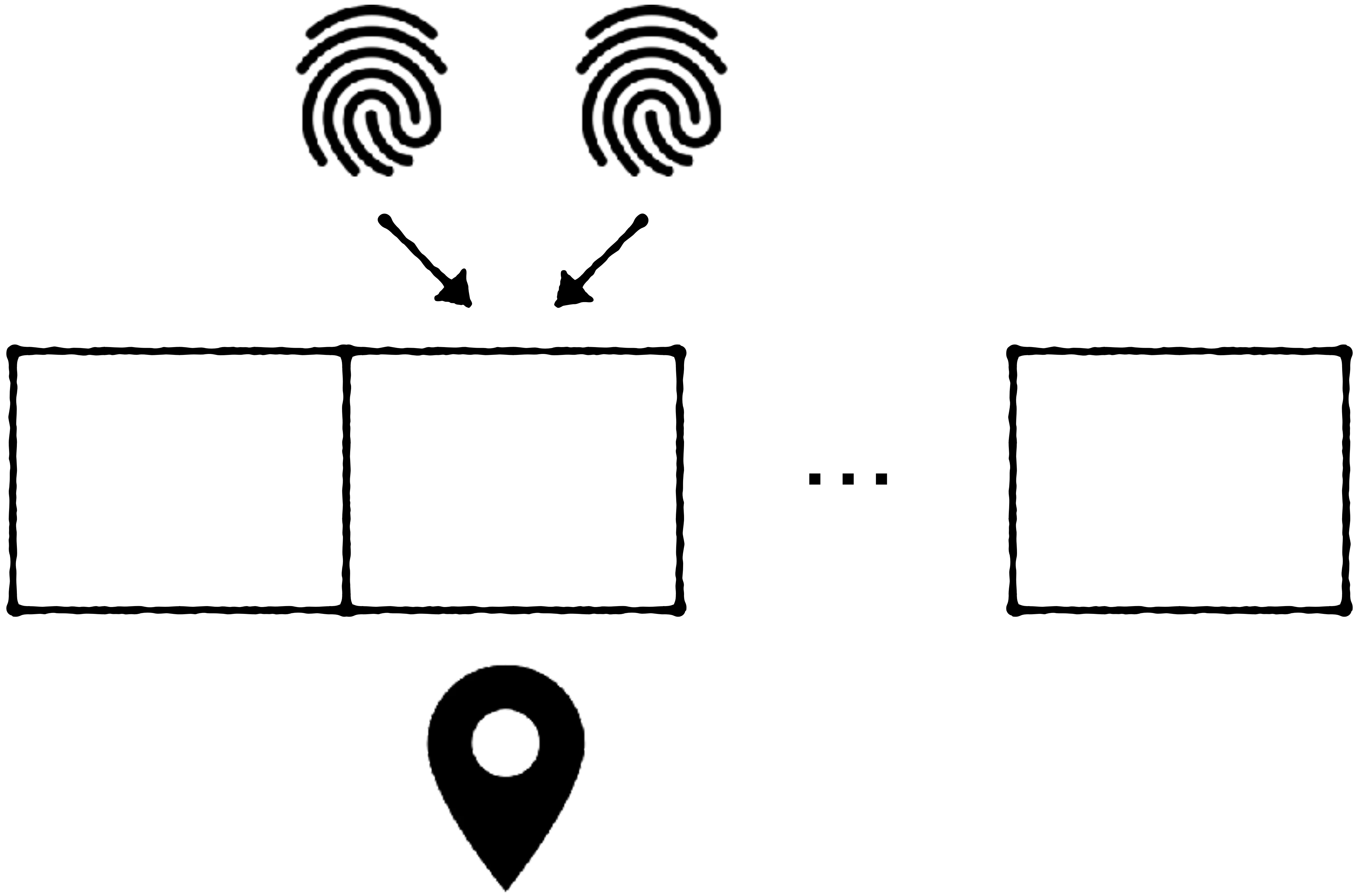
Canonical slot

Address using Robin Hood Hashing

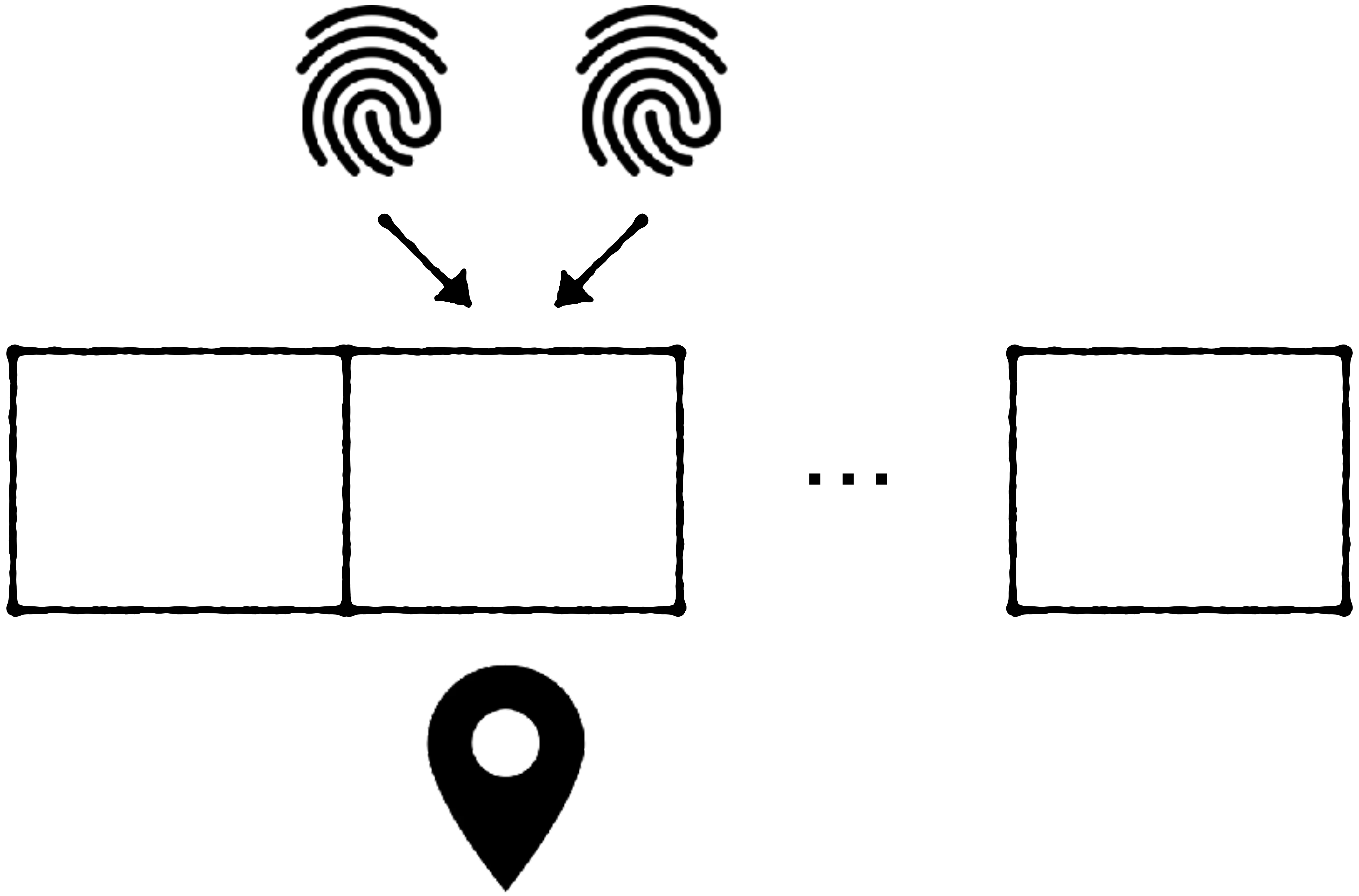


Address using Robin Hood Hashing

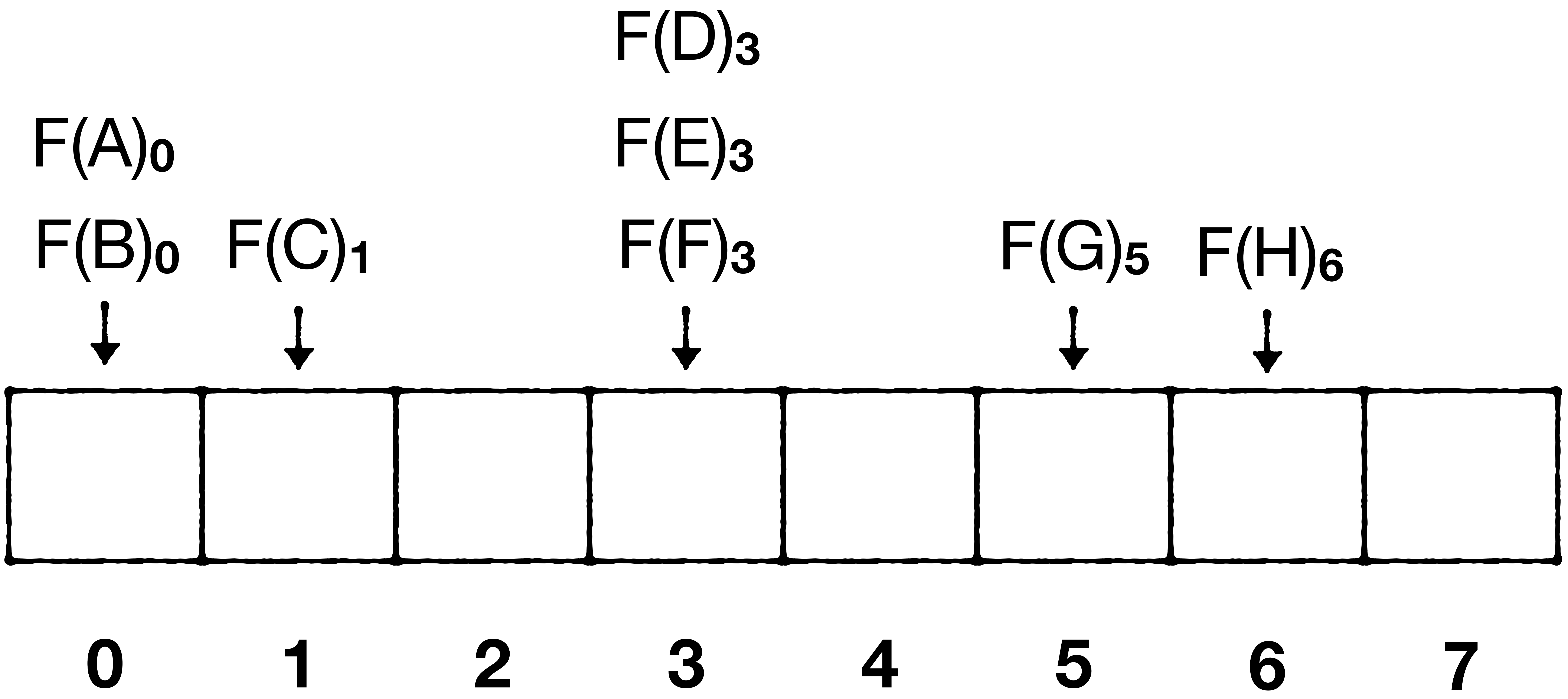
Variant of linear probing



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”

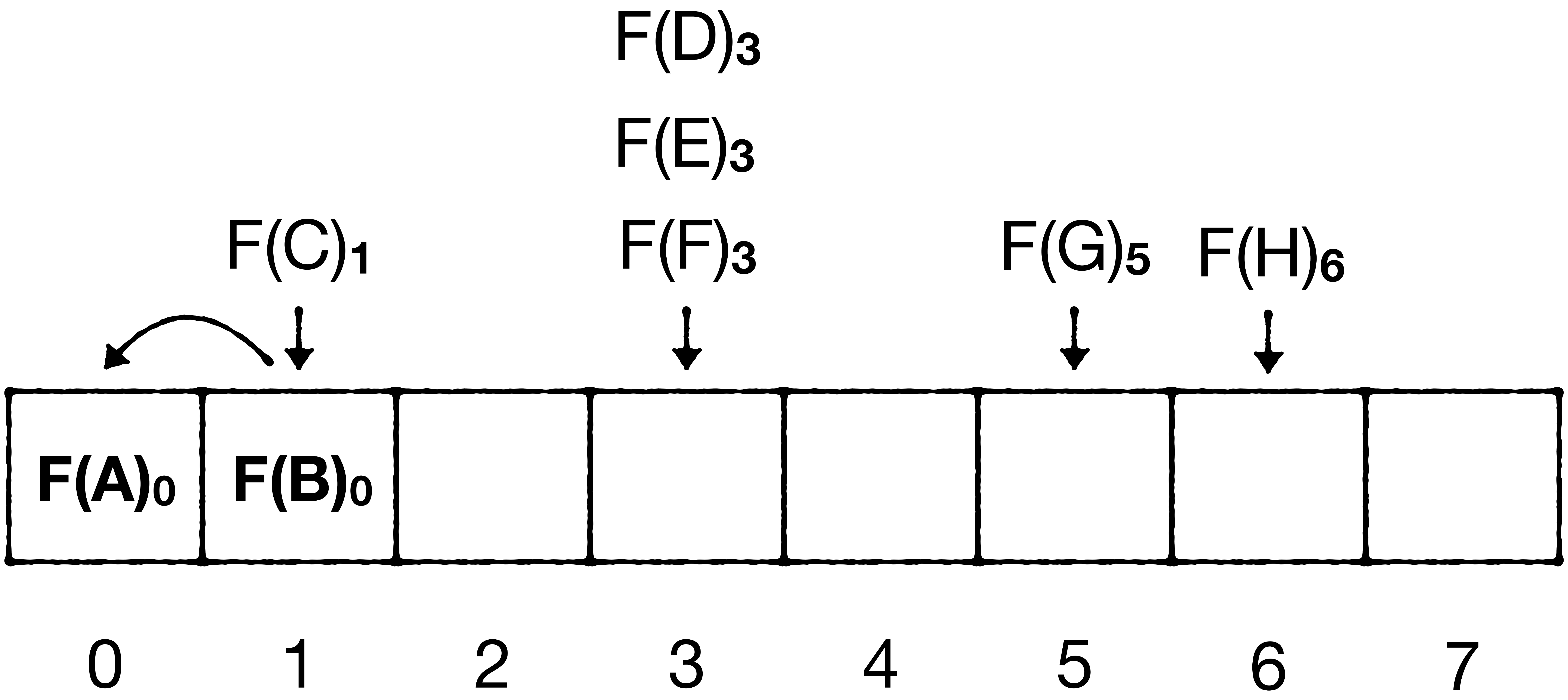


$F(D)_3$

$F(A)_0$ $F(E)_3$

$F(B)_0$ $F(C)_1$ $F(F)_3$ $F(G)_5$ $F(H)_6$

0 1 2 3 4 5 6 7

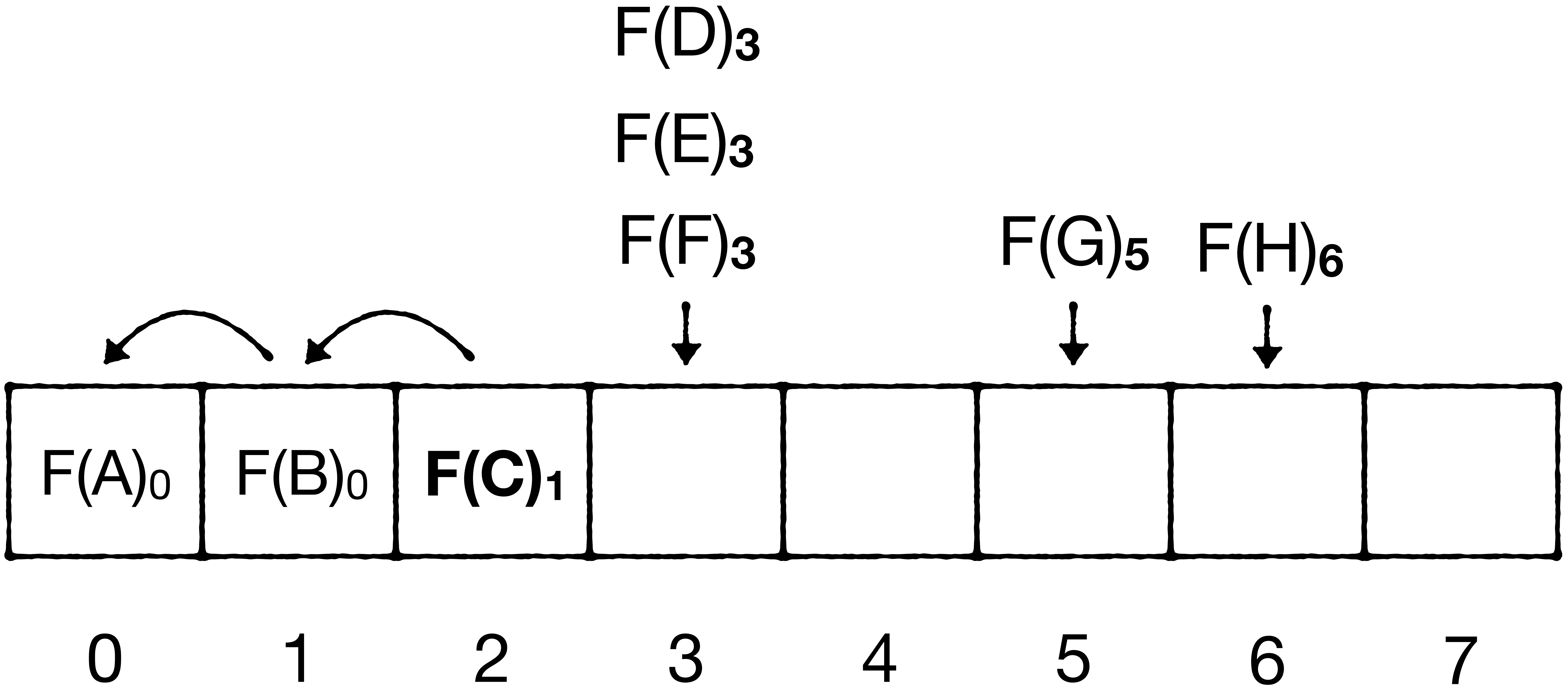


$F(D)_3$

$F(E)_3$

$F(C)_1$ $F(F)_3$ $F(G)_5$ $F(H)_6$

$F(A)_0$	**$F(B)_0$**						
0	1	2	3	4	5	6	7



$F(D)_3$

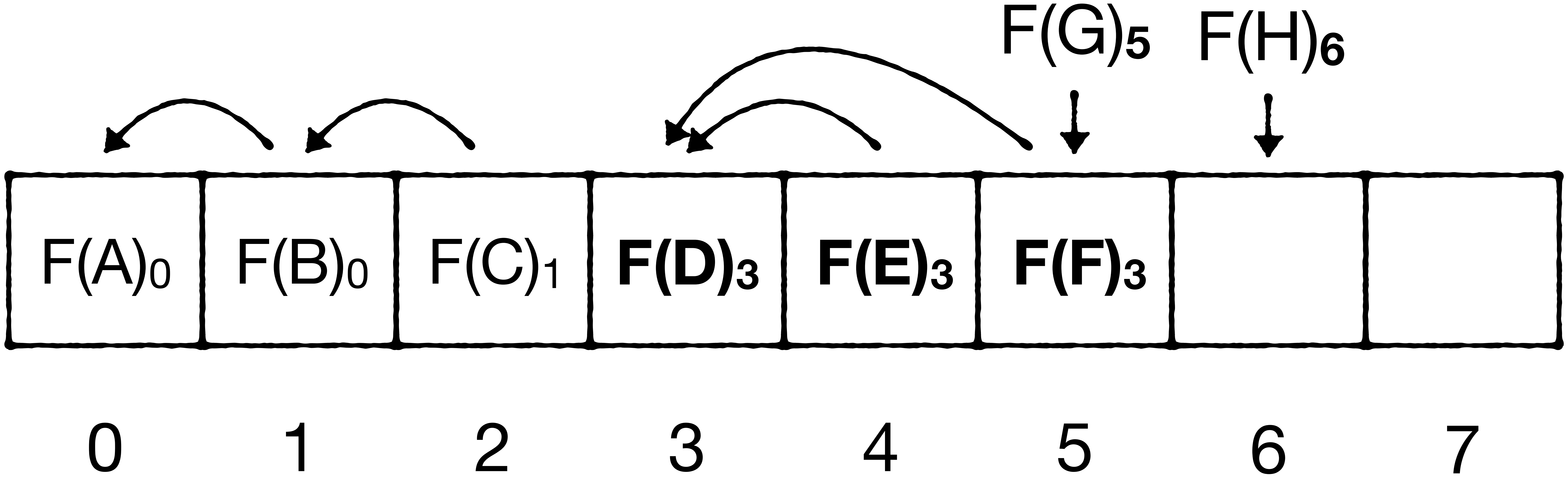
$F(E)_3$

$F(F)_3$

$F(G)_5$

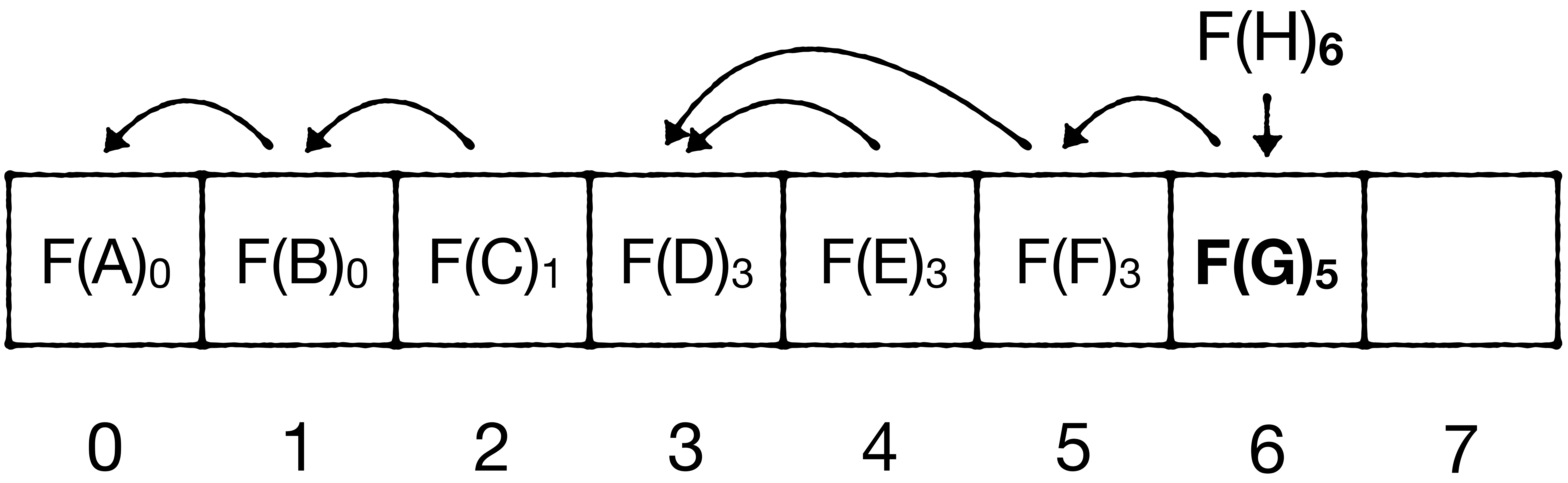
$F(H)_6$

$F(A)_0$	$F(B)_0$	**$F(C)_1$**					
0	1	2	3	4	5	6	7



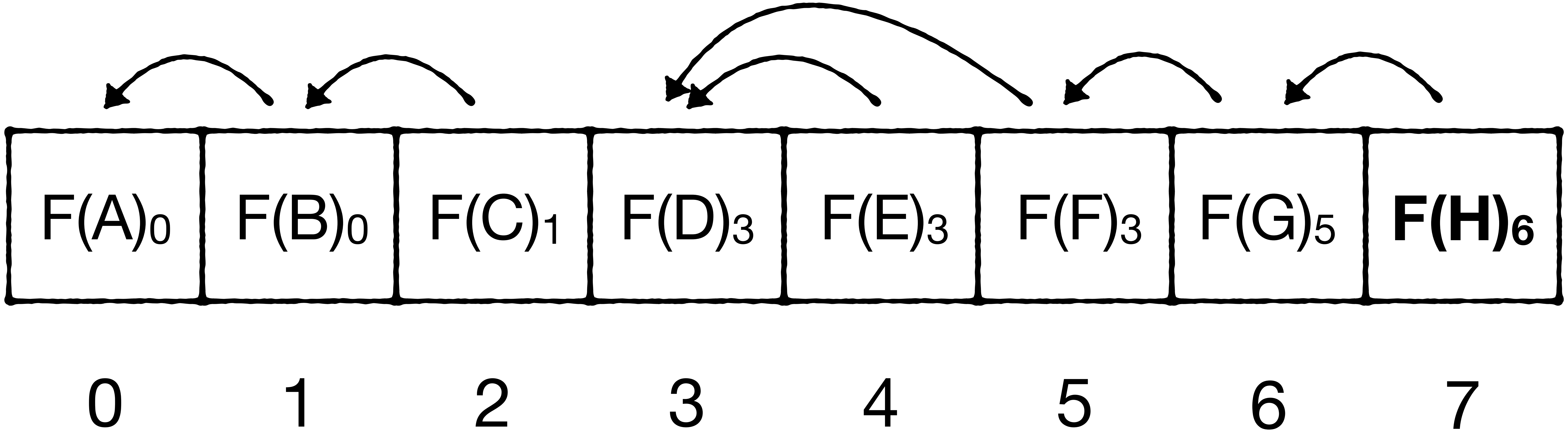
$F(G)_{\mathbf{5}}$ $F(H)_{\mathbf{6}}$

$F(A)_0$	$F(B)_0$	$F(C)_1$	$\mathbf{F(D)_3}$	$\mathbf{F(E)_3}$	$\mathbf{F(F)_3}$		
0	1	2	3	4	5	6	7

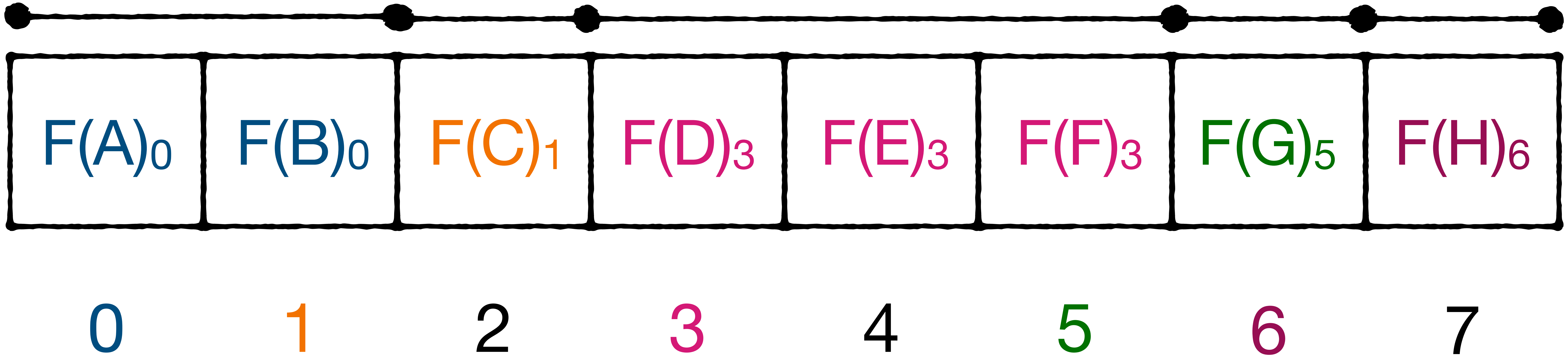


$F(H)_{\mathbf{6}}$

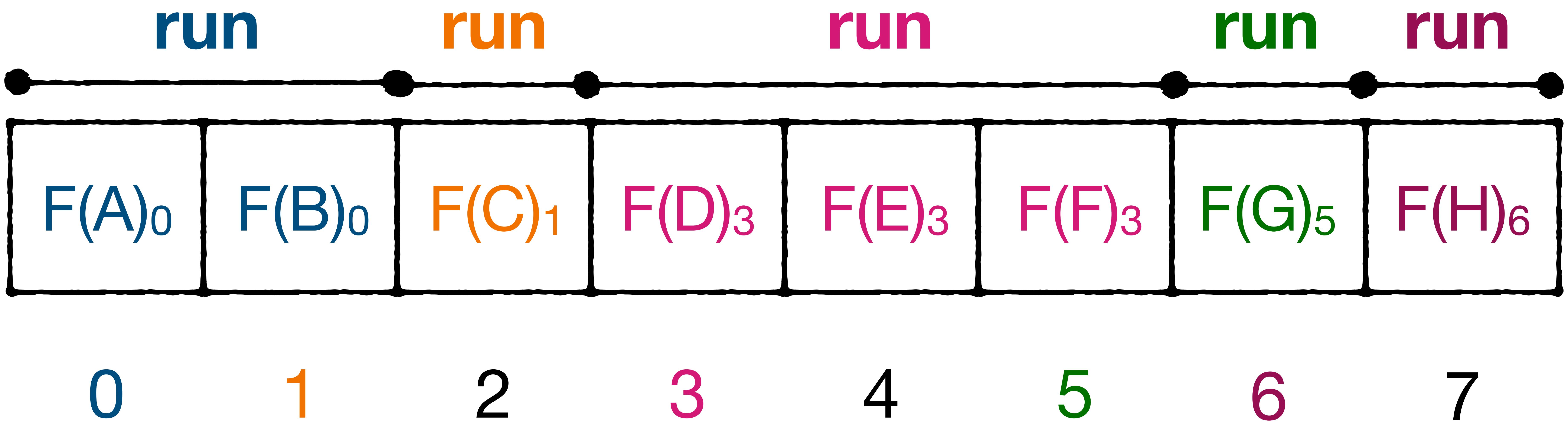
$F(A)_0$	$F(B)_0$	$F(C)_1$	$F(D)_3$	$F(E)_3$	$F(F)_3$	$\mathbf{F(G)_5}$	
0	1	2	3	4	5	6	7



Note: fingerprints belonging to same canonical slot are contiguous



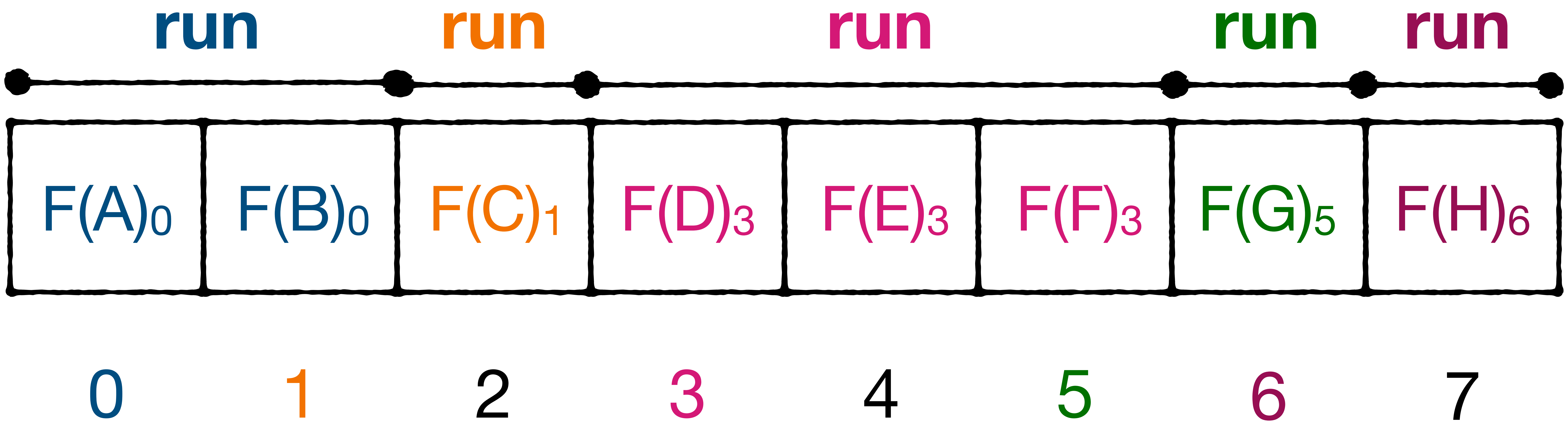
Note: fingerprints belonging to same canonical slot are contiguous



run run run run run

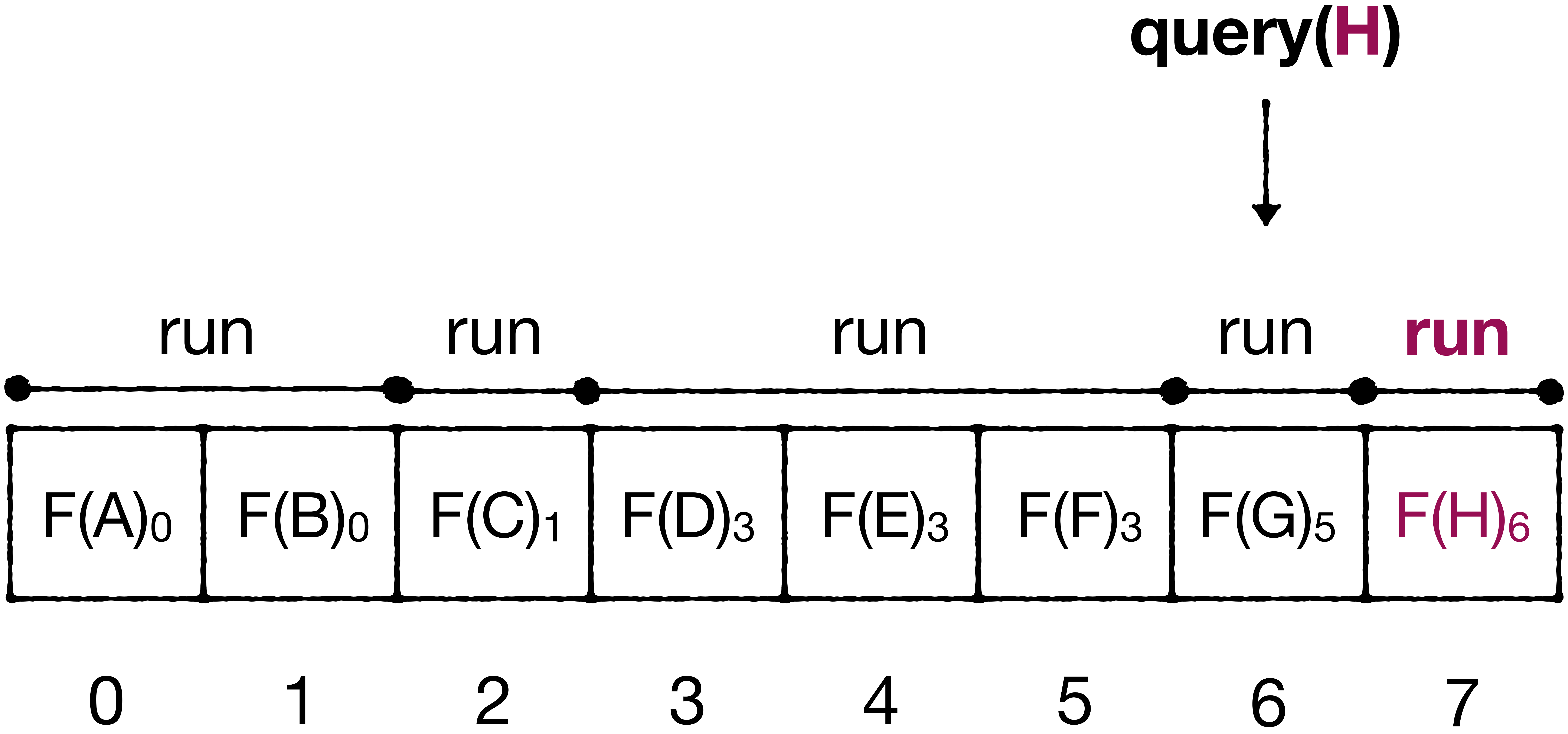
$F(A)_0$	$F(B)_0$	$F(C)_1$	$F(D)_3$	$F(E)_3$	$F(F)_3$	$F(G)_5$	$F(H)_6$
0	1	2	3	4	5	6	7

On average, each run consists of ≈ 1 slot



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

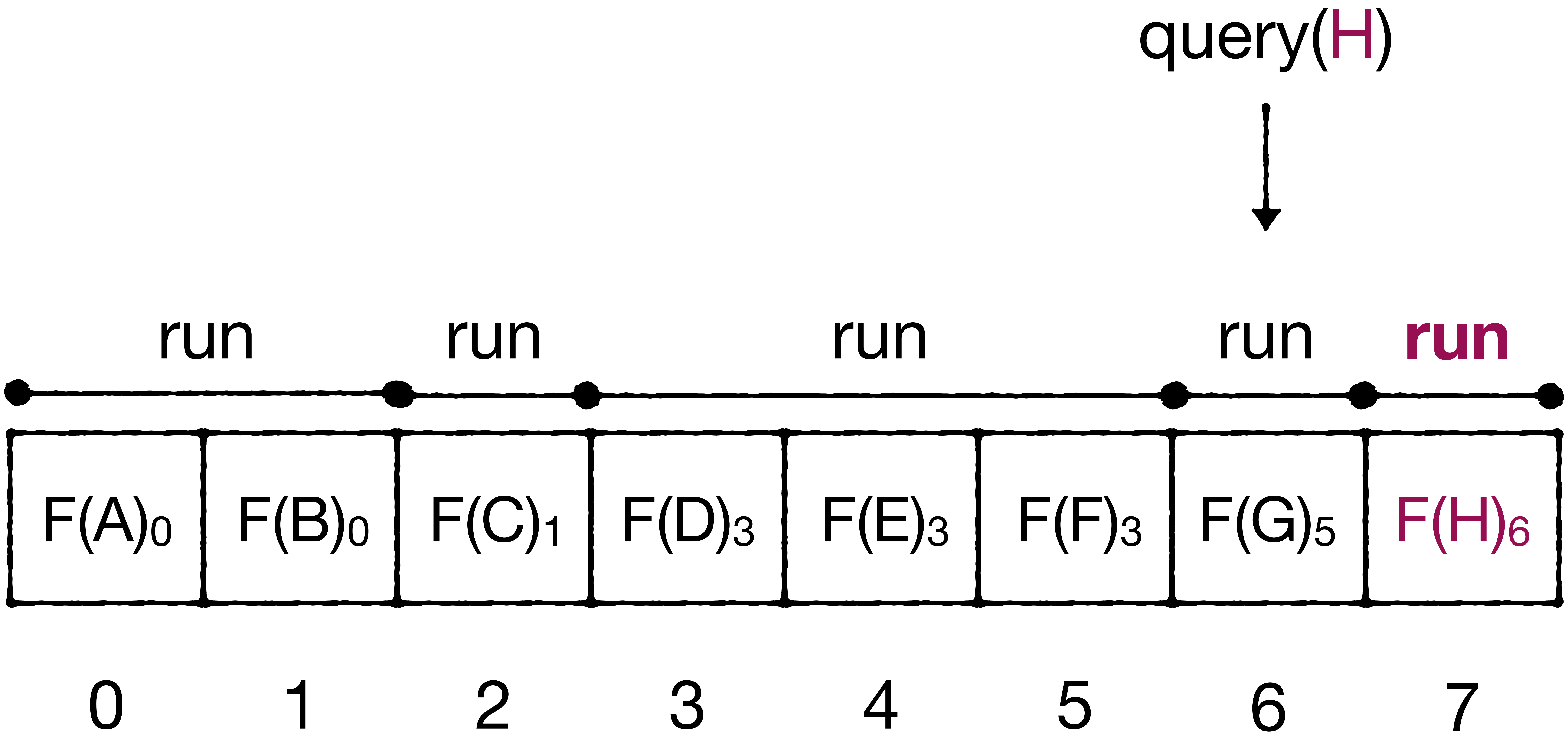


query(H)

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?

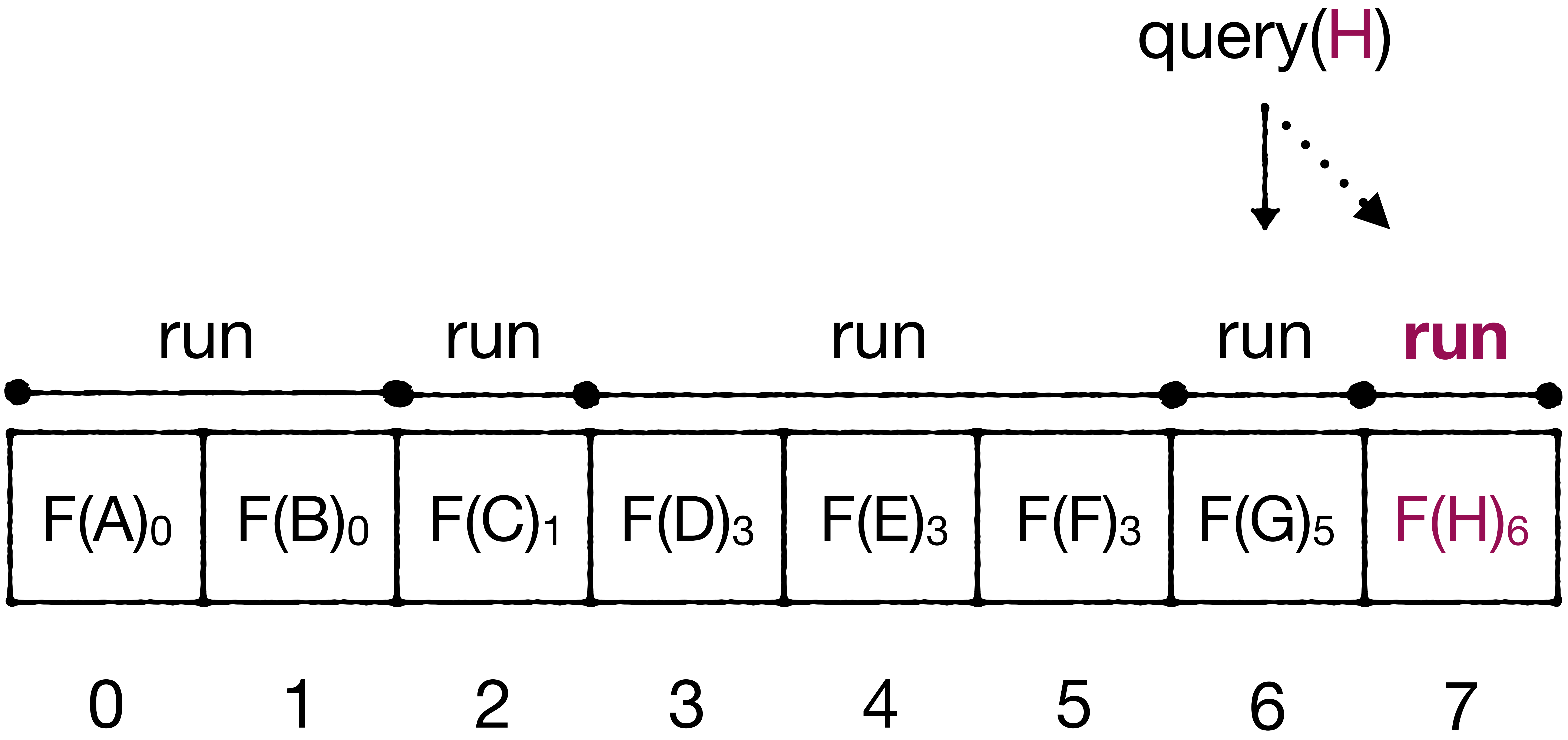


query(H)

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**



query(H)

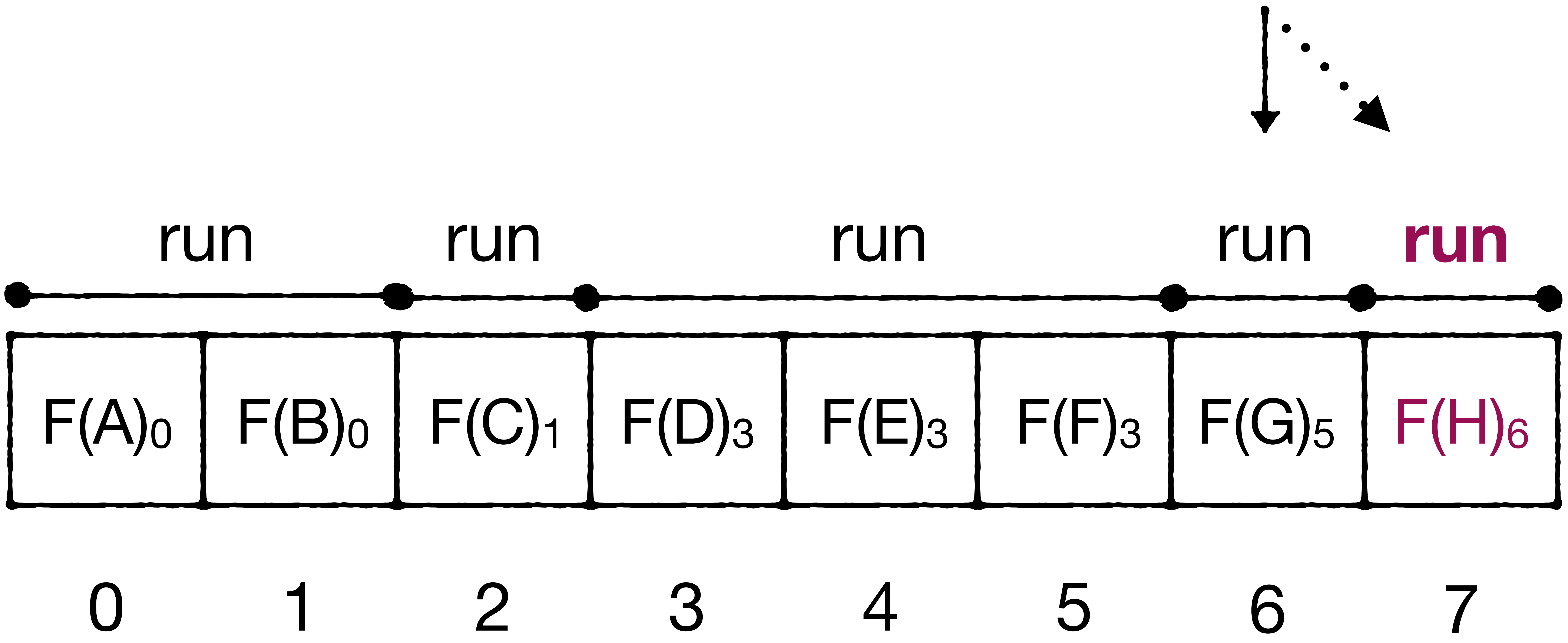
run run run run **run**

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

Problem? Fingerprint might have shifted to the right

Solution?

query(H)

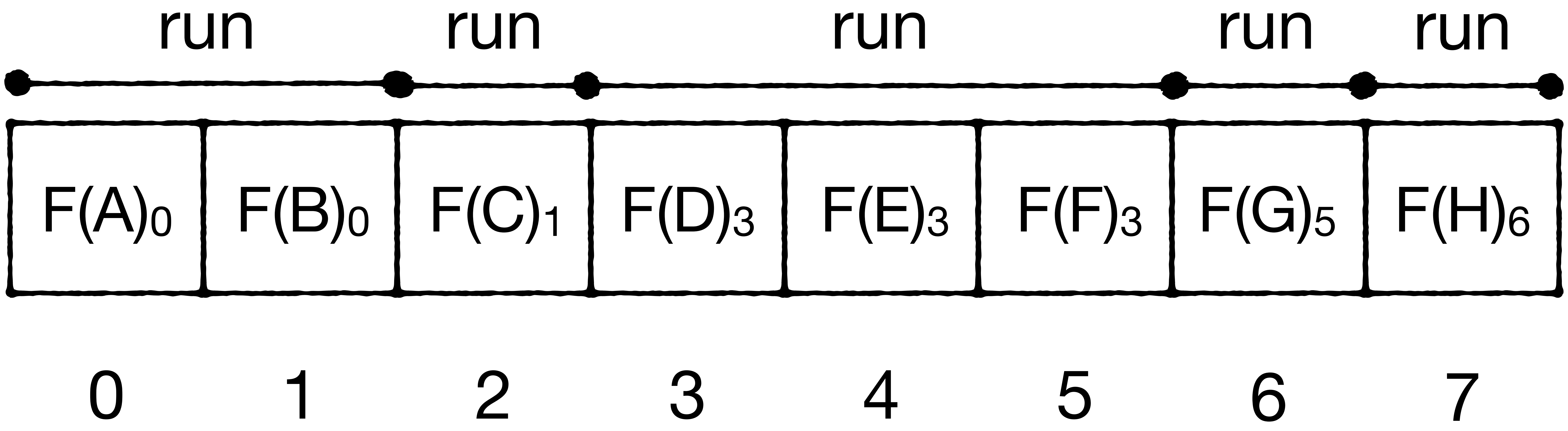


$F(A)_0$	$F(B)_0$	$F(C)_1$	$F(D)_3$	$F(E)_3$	$F(F)_3$	$F(G)_5$	$F(H)_6$
0	1	2	3	4	5	6	7

Solution: delineate runs using 2 bitmaps

Occupied:

End:

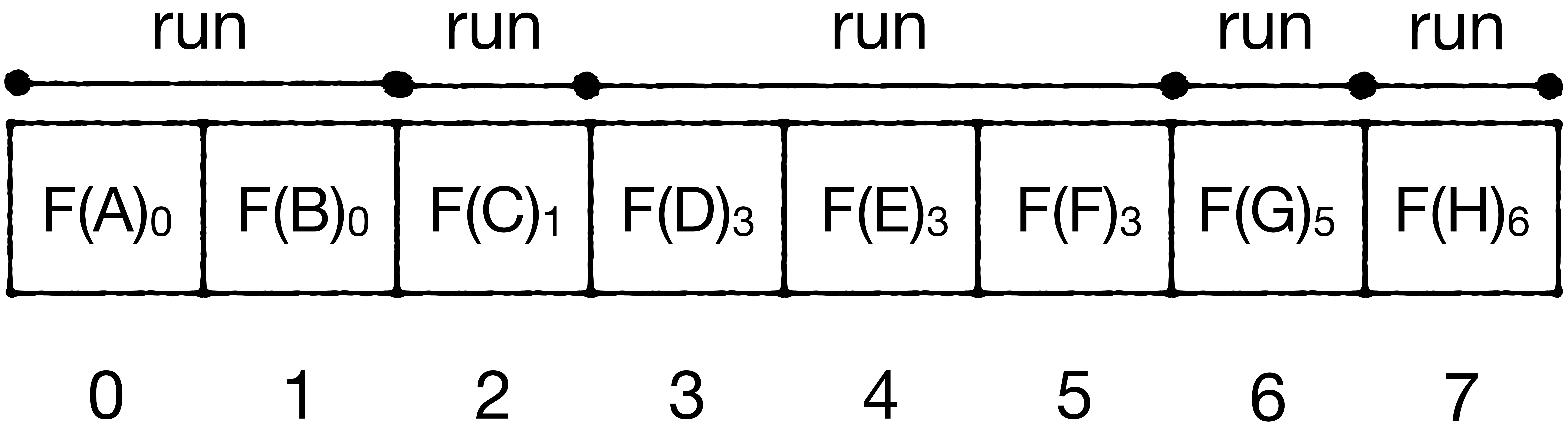


run run run run run

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

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

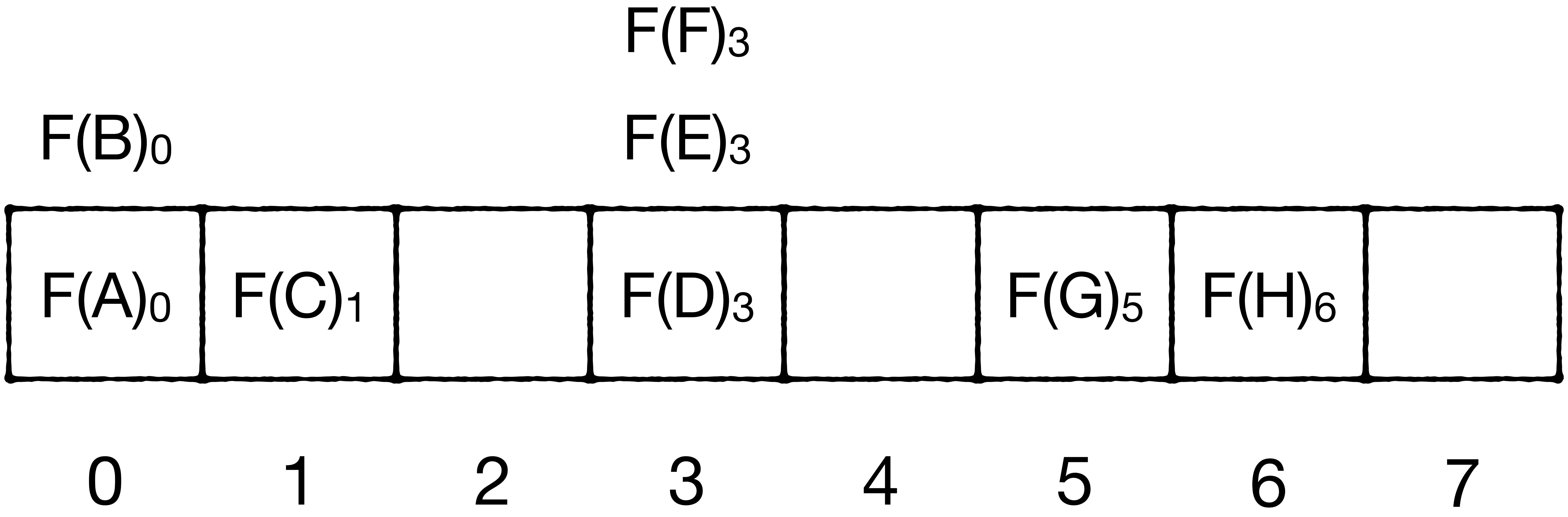
End:



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

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

End:



$F(F)_3$

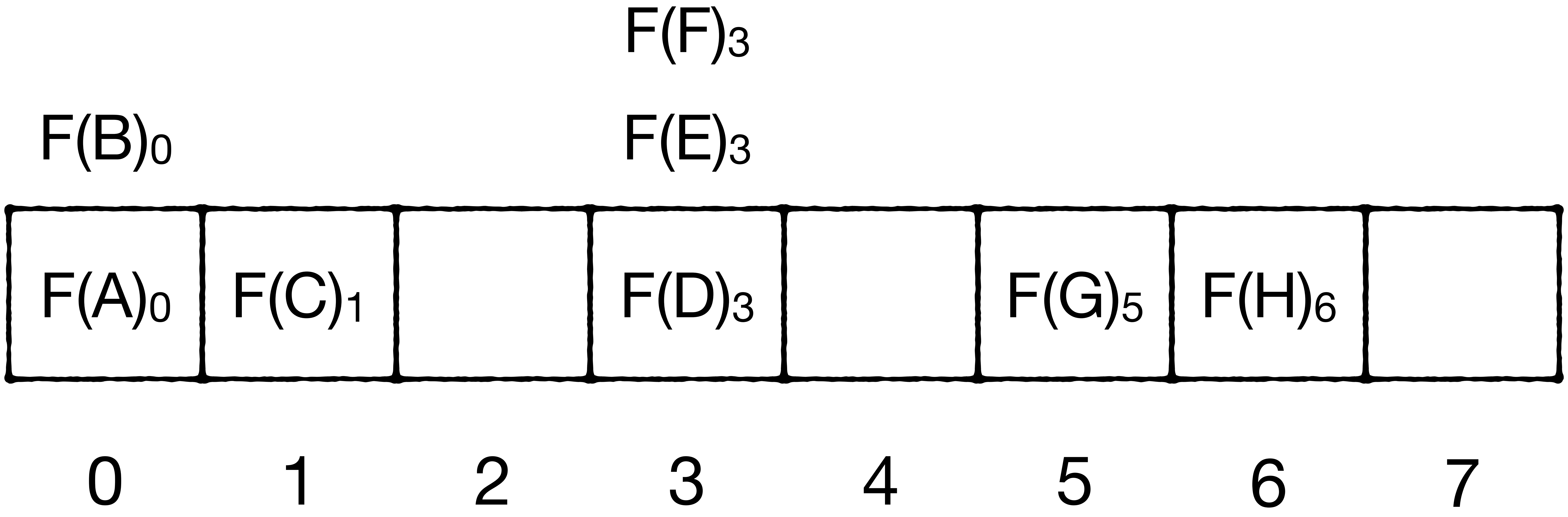
$F(B)_0$ $F(E)_3$

$F(A)_0$ $F(C)_1$ $F(D)_3$ $F(G)_5$ $F(H)_6$

0 1 2 3 4 5 6 7

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

End:



F(F)$_3$

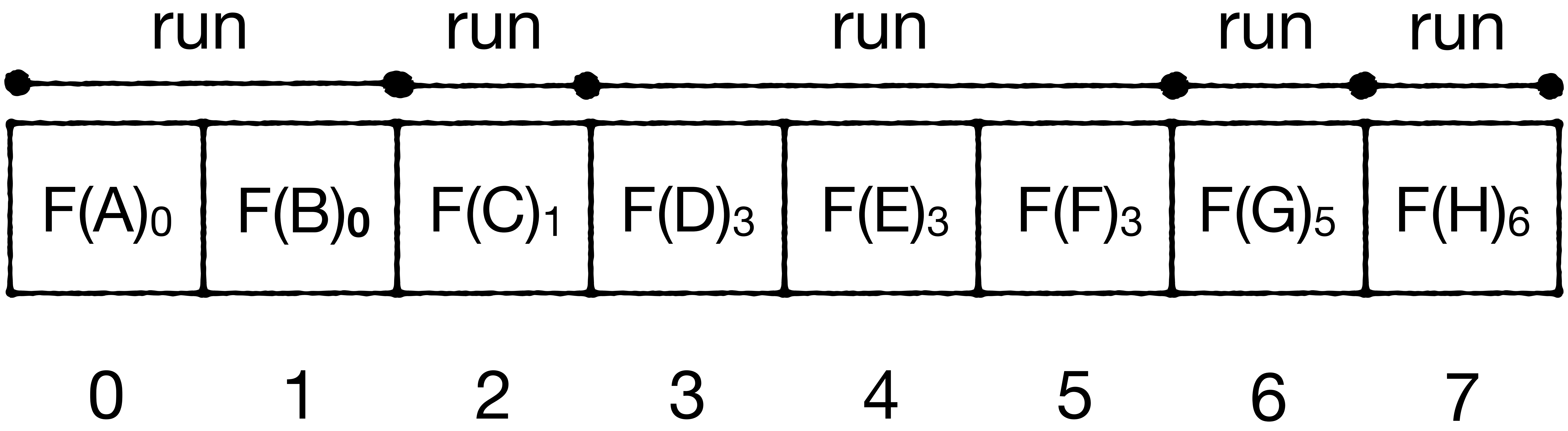
F(B)$_0$ F(E)$_3$

F(A)$_0$ F(C)$_1$ F(D)$_3$ F(G)$_5$ F(H)$_6$

0 1 2 3 4 5 6 7

Occupied: 1 1 0 1 0 1 1 0

End:

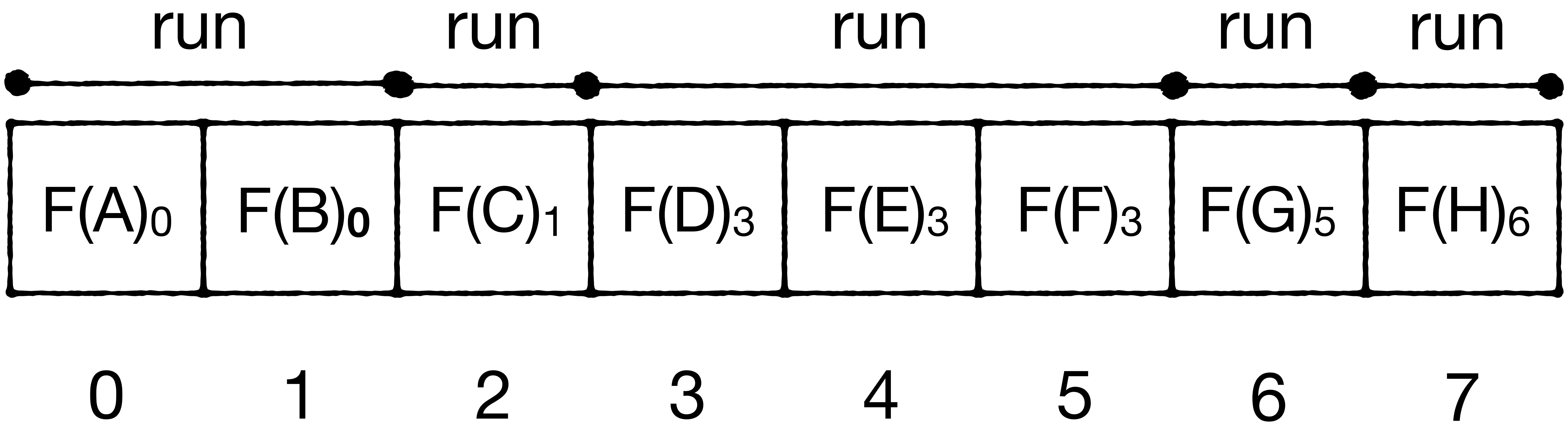


run run run run run

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

Occupied:	1	1	0	1	0	1	1	0

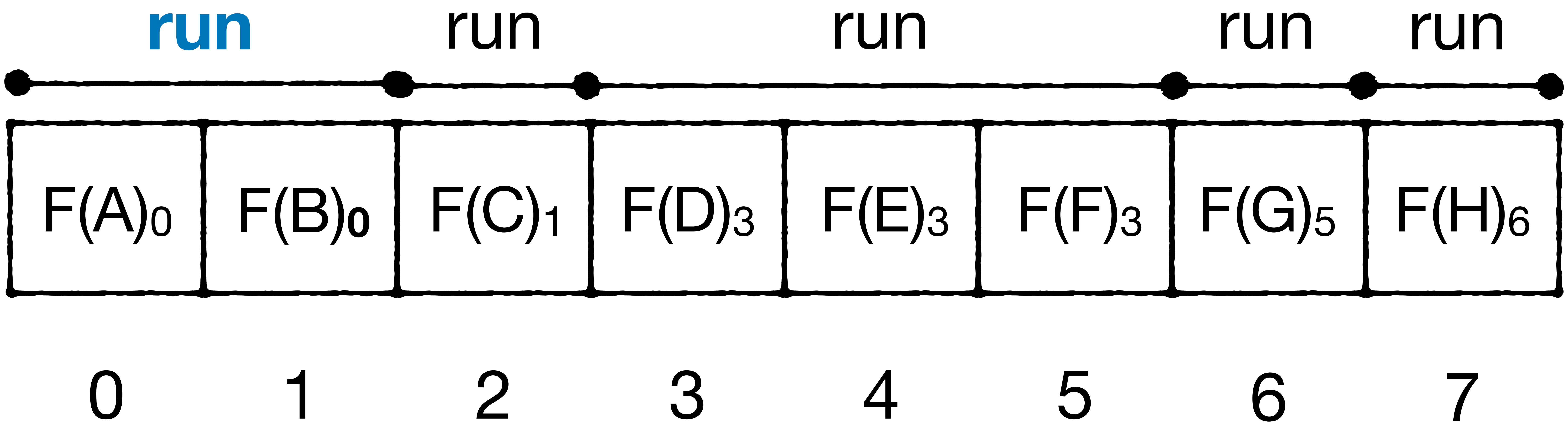
End: **1 for each slot where a run ends**



run run run run run

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

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

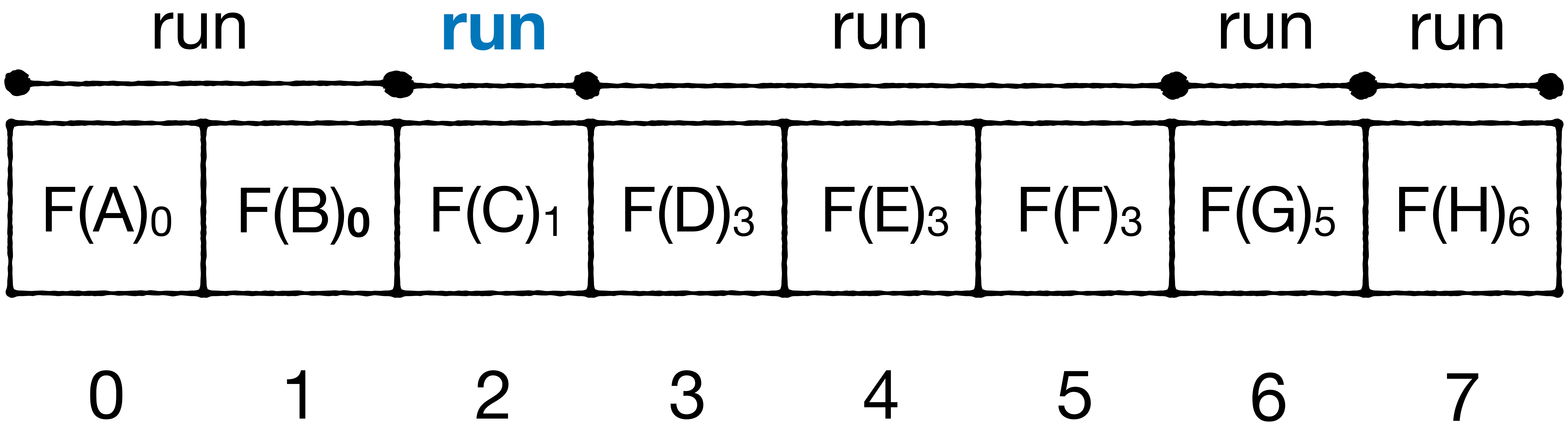


run run run run run

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

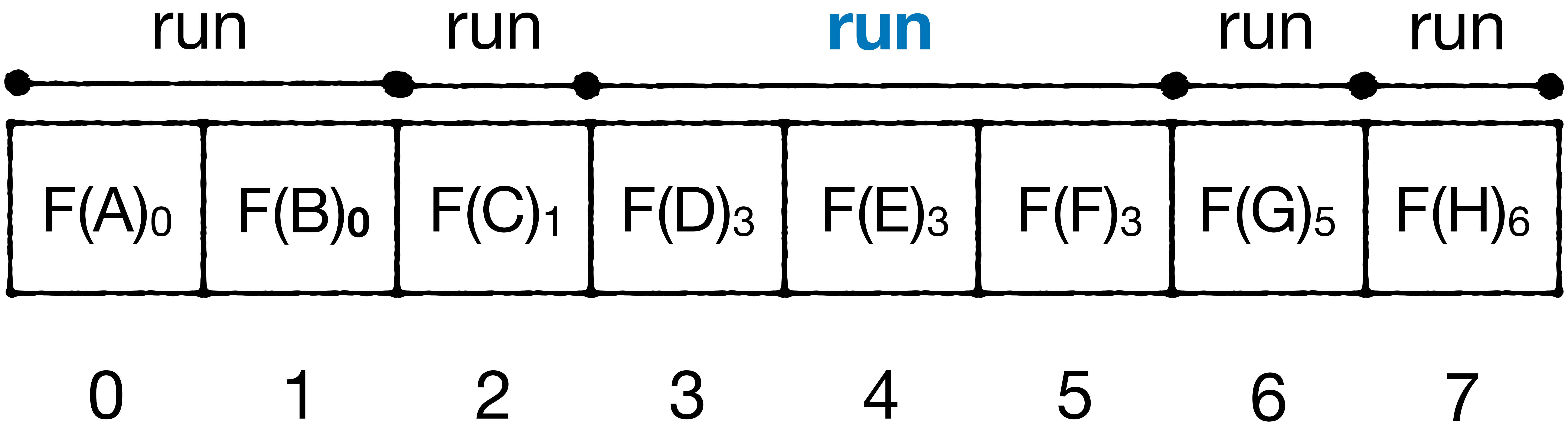
Occupied: 1 1 0 1 0 1 1 0

End: 0 1 **1**



F(A)$_0$	F(B)$_\mathbf{0}$	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

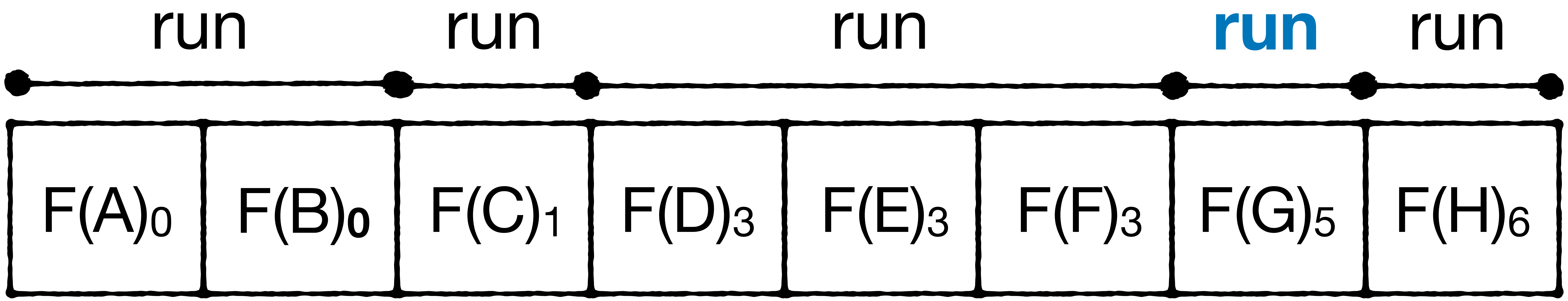
Occupied:	1	1	0	1	0	1	1	0
End:	0	1	1	**0**	**0**	**1**		



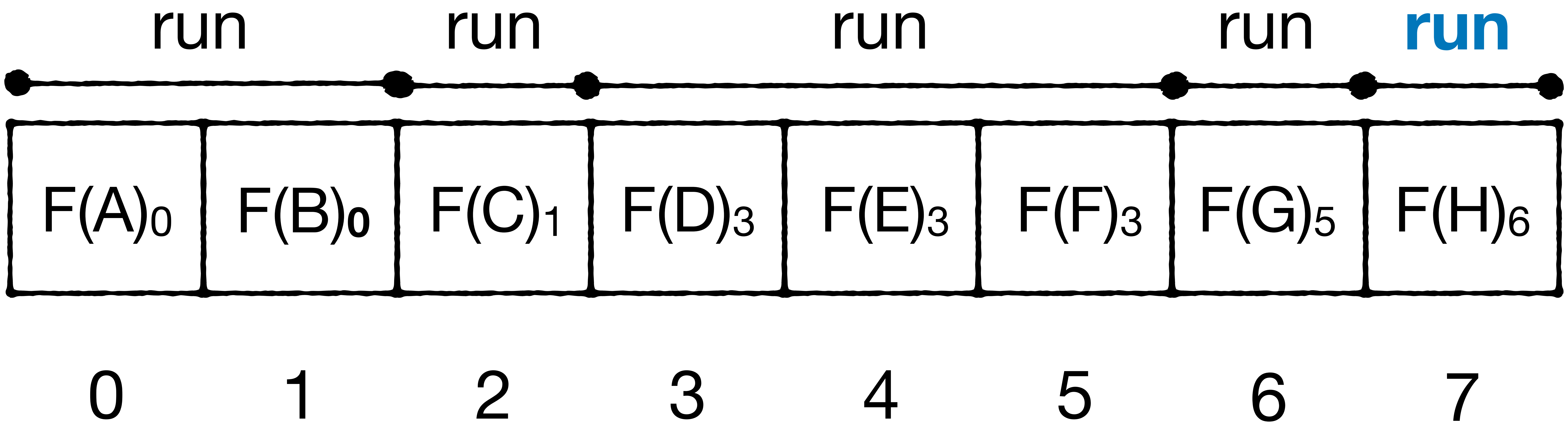
run run **run** run run

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

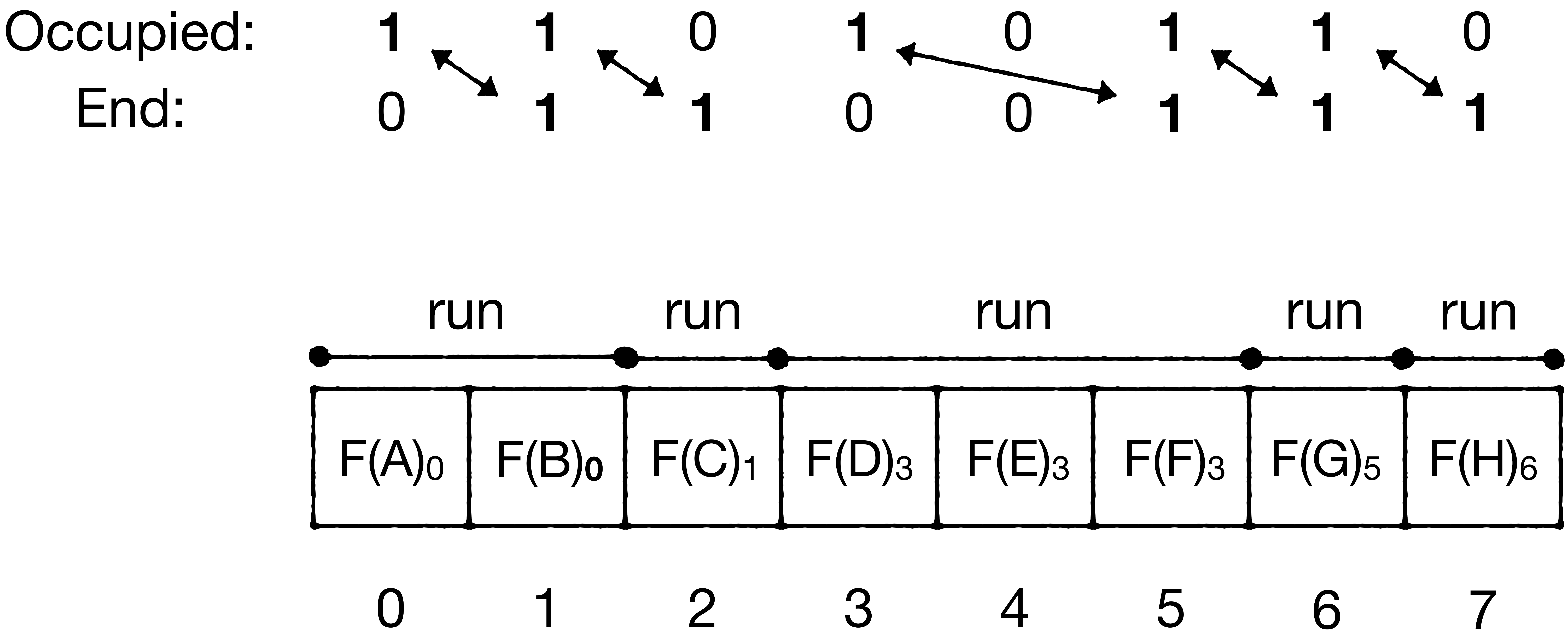
Occupied:	1	1	0	1	0	1	1	0
End:	0	1	1	0	0	1	**1**	
	$F(A)_0$	$F(B)_{\mathbf{0}}$	$F(C)_1$	$F(D)_3$	$F(E)_3$	$F(F)_3$	$F(G)_5$	$F(H)_6$
	0	1	2	3	4	5	6	7



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

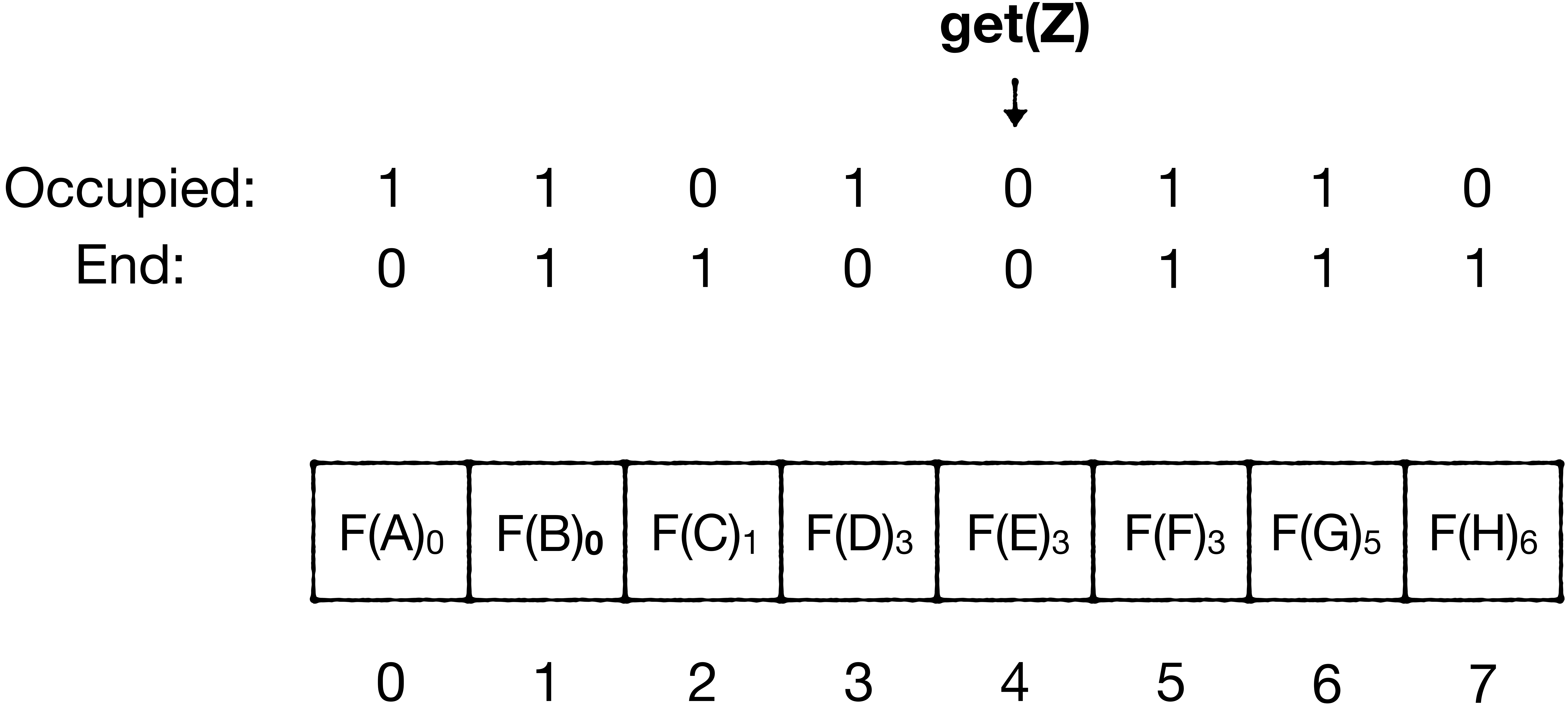


Occupied:	**1**	**1**	0	**1**	0	**1**	**1**	0
End:	0	**1**	**1**	0	0	**1**	**1**	**1**

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

How to query?

Occupied:	1	1	0	1	0	1	1	0
End:	0	1	1	0	0	1	1	1
	$F(A)_0$	$F(B)_{\mathbf{0}}$	$F(C)_1$	$F(D)_3$	$F(E)_3$	$F(F)_3$	$F(G)_5$	$F(H)_6$
	0	1	2	3	4	5	6	7



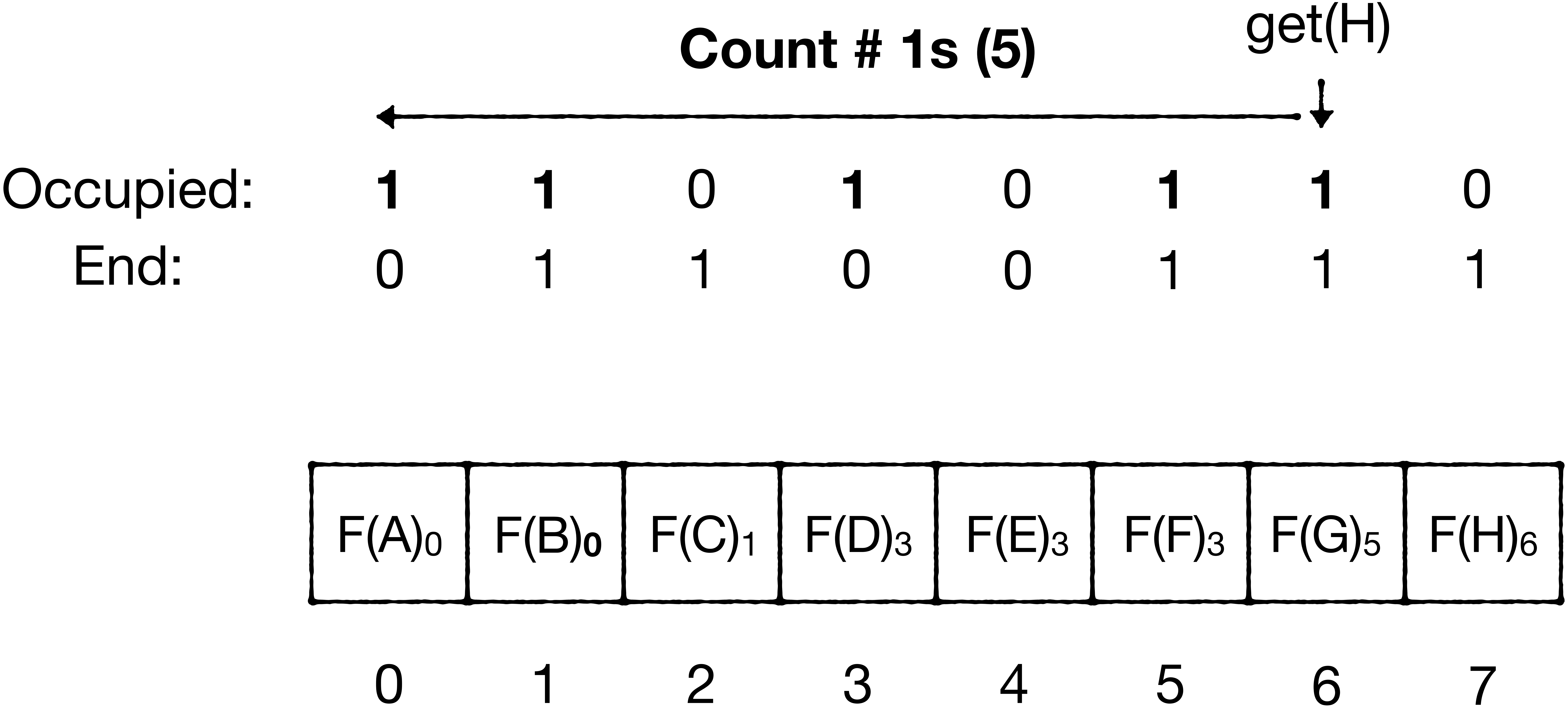
					get(Z) ↓			
Occupied:	1	1	0	1	0	1	1	0
End:	0	1	1	0	0	1	1	1
	$F(A)_0$	$F(B)_{\mathbf{0}}$	$F(C)_1$	$F(D)_3$	$F(E)_3$	$F(F)_3$	$F(G)_5$	$F(H)_6$
	0	1	2	3	4	5	6	7

get(Z) **return negative**

↓

Occupied:	1	1	0	1	**0**	1	1	0
End:	0	1	1	0	0	1	1	1
	$F(A)_0$	$F(B)_{\mathbf{0}}$	$F(C)_1$	$F(D)_3$	$F(E)_3$	$F(F)_3$	$F(G)_5$	$F(H)_6$
	0	1	2	3	4	5	6	7

							get(H)	
							↓	
Occupied:	1	1	0	1	0	1	**1**	0
End:	0	1	1	0	0	1	1	1
	$F(A)_0$	$F(B)_{\mathbf{0}}$	$F(C)_1$	$F(D)_3$	$F(E)_3$	$F(F)_3$	$F(G)_5$	$F(H)_6$
	0	1	2	3	4	5	6	7

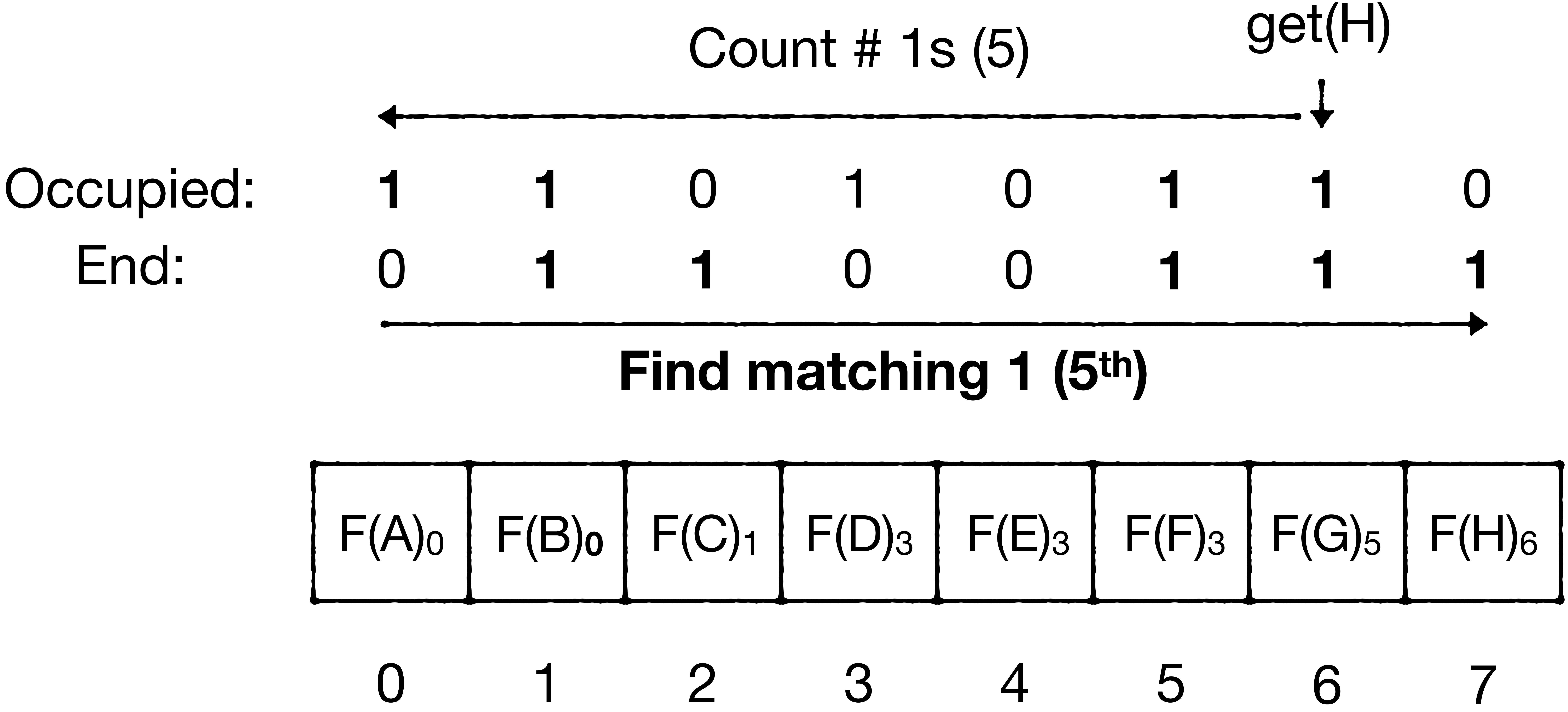


Count # 1s (5)

get(H)

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

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



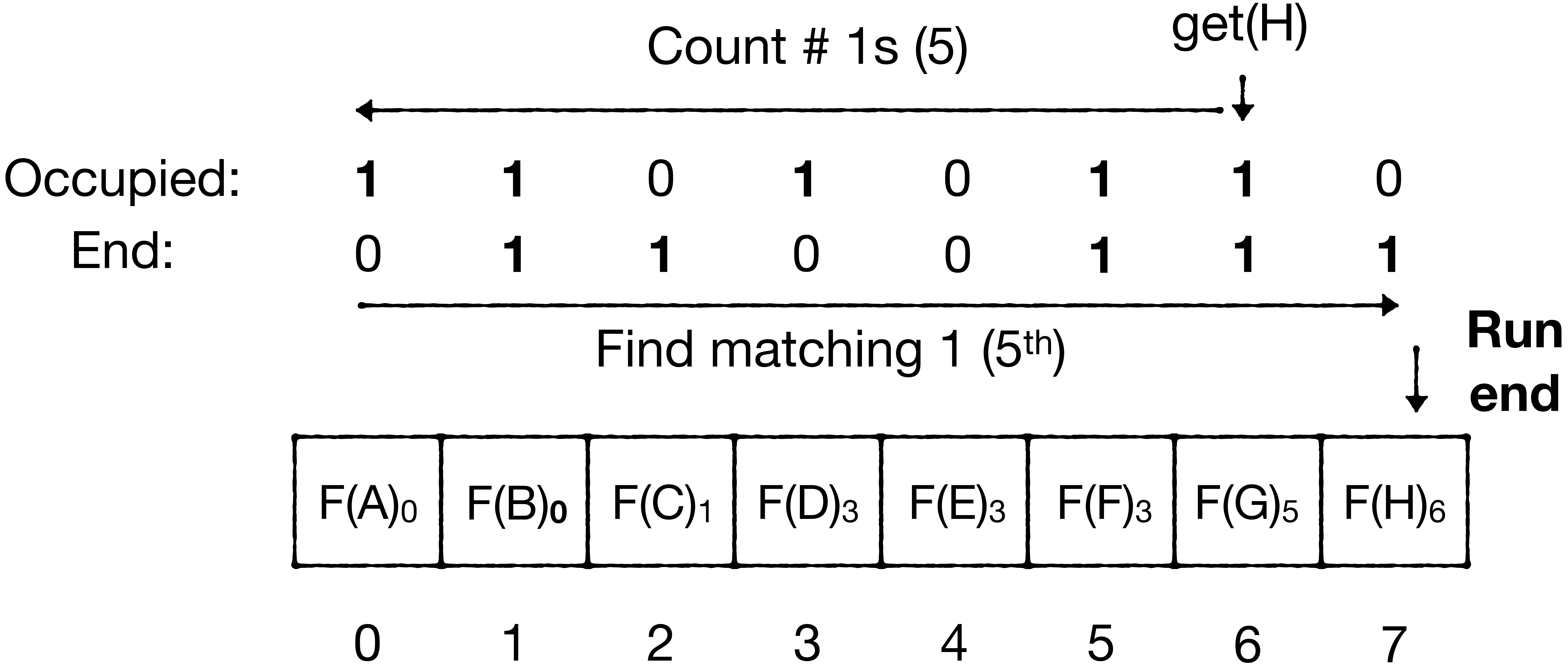
Count # 1s (5)

get(H)

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

Find matching 1 (5th)

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



get(H)

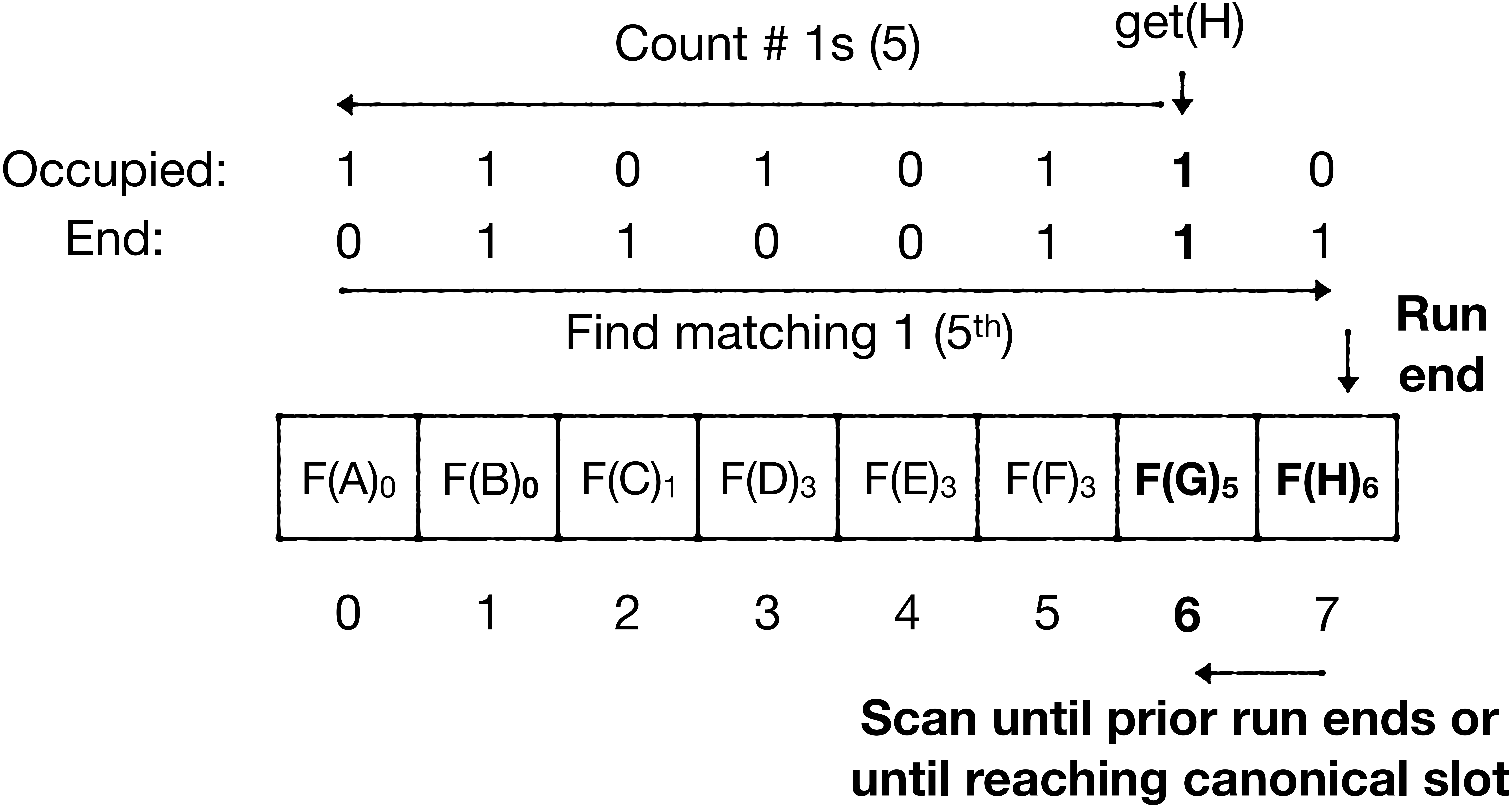
Count # 1s (5)

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

Find matching 1 (5^{th})

Run end

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



get(H)

Count # 1s (5)

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

Find matching 1 (5th)

Run end

Scan until prior run ends 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
	$F(A)_0$	$F(B)_{\mathbf{0}}$	$F(C)_1$	$F(D)_3$	$F(E)_3$	$F(F)_3$	$F(G)_5$	$F(H)_6$
	0	1	2	3	4	5	6	7

Can handle queries :) **problem?**

Occupied:	1	1	0	1	0	1	1	0
End:	0	1	1	0	0	1	1	1
	$F(A)_0$	$F(B)_{\mathbf{0}}$	$F(C)_1$	$F(D)_3$	$F(E)_3$	$F(F)_3$	$F(G)_5$	$F(H)_6$
	0	1	2	3	4	5	6	7

Scanning bitmaps takes O(N)

Occupied:	1	1	0	1	0	1	1	0
End:	0	1	1	0	0	1	1	1
	$F(A)_0$	$F(B)_{\mathbf{0}}$	$F(C)_1$	$F(D)_3$	$F(E)_3$	$F(F)_3$	$F(G)_5$	$F(H)_6$
	0	1	2	3	4	5	6	7

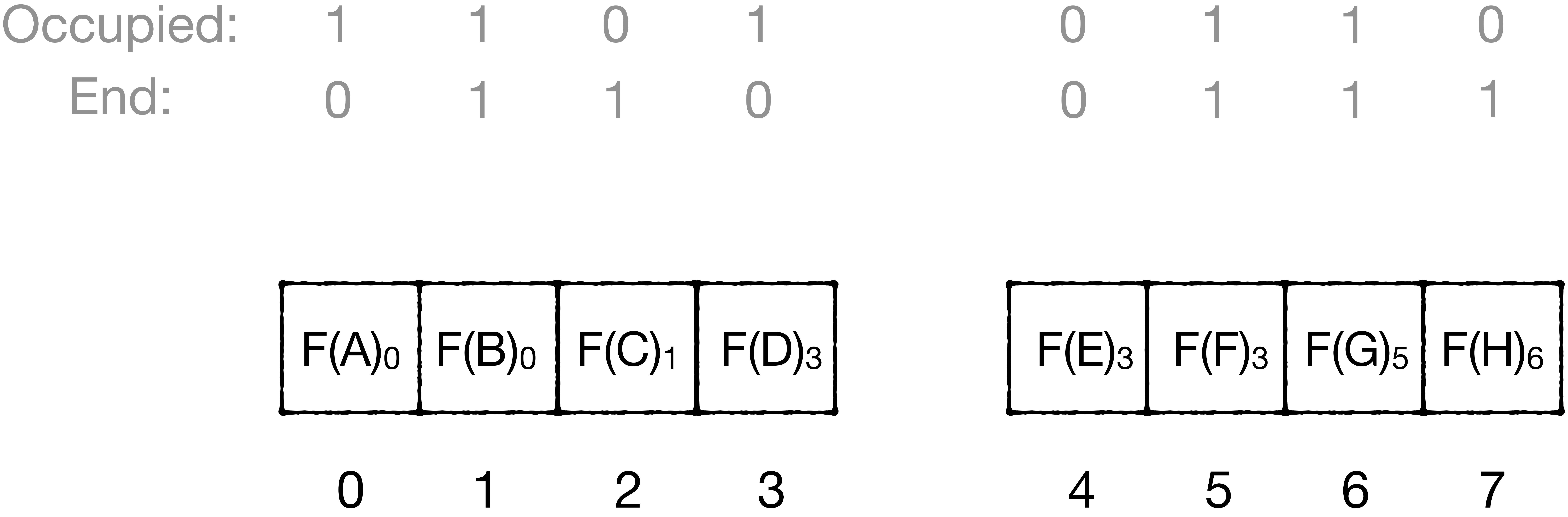
Scanning bitmaps takes O(N)

Ideas?

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

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

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

Split filter into chunks (64 slots in practice)

≈ 1-2 cache lines

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

Each chunk has offset field (8 bits)

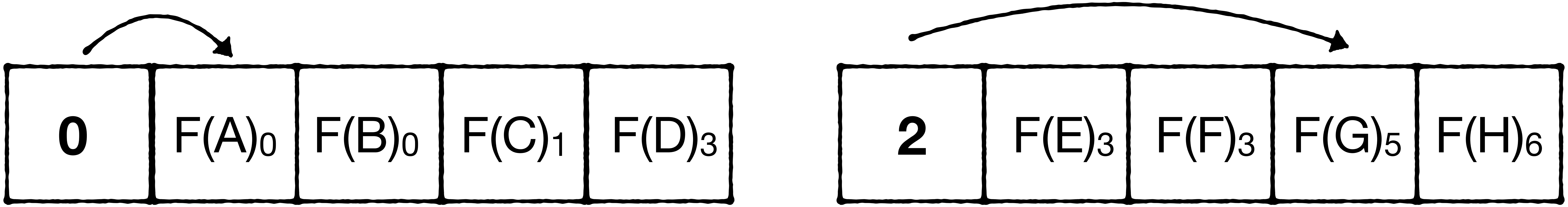
Occupied:	1	1	0	1
End:	0	1	1	0
	$F(A)_0$	$F(B)_0$	$F(C)_1$	$F(D)_3$
Offset	0	1	2	3

	0	1	1	0
	0	1	1	1
	$F(E)_3$	$F(F)_3$	$F(G)_5$	$F(H)_6$
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



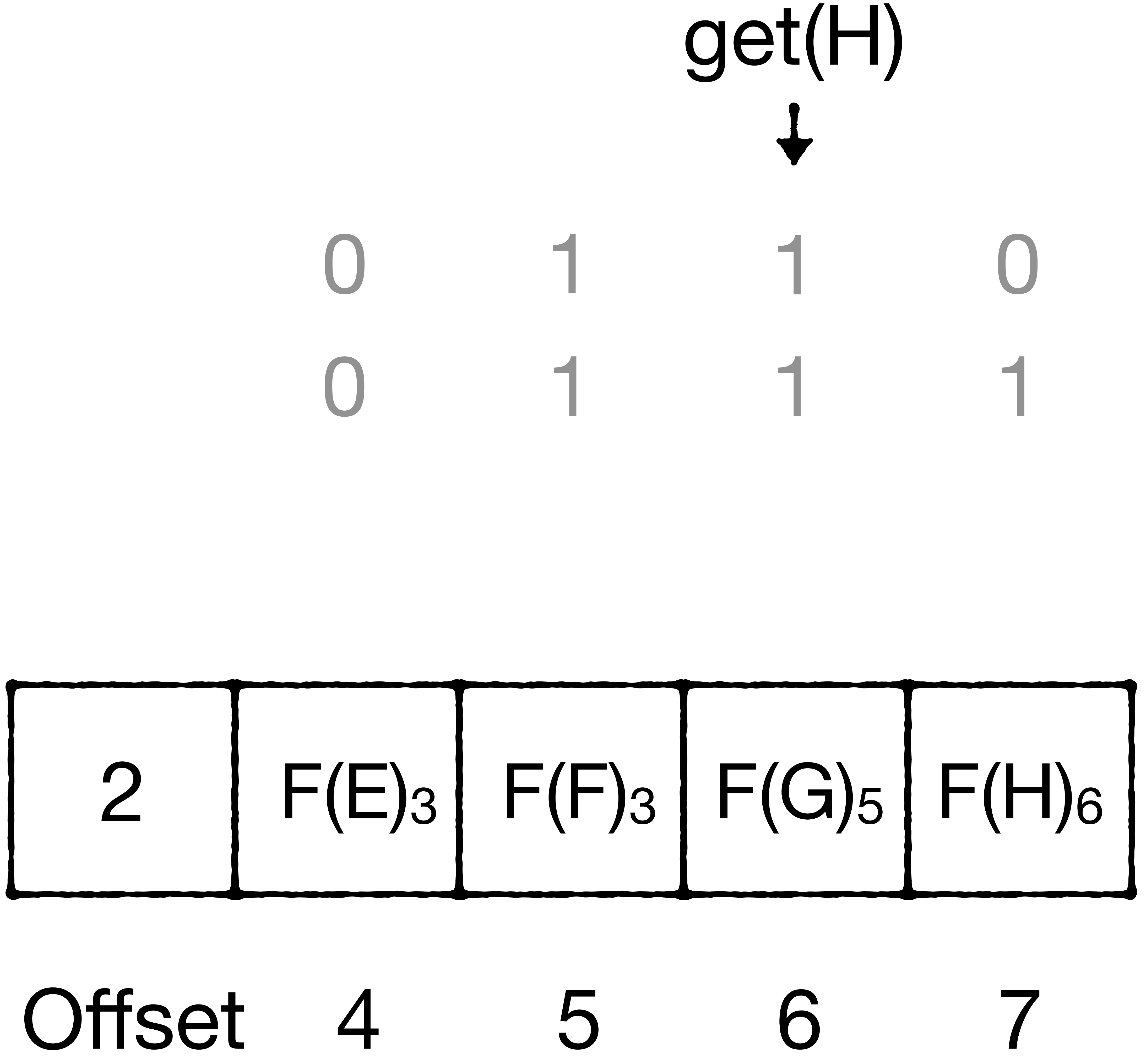
0	$F(A)_0$	$F(B)_0$	$F(C)_1$	$F(D)_3$	**2**	$F(E)_3$	$F(F)_3$	$F(G)_5$	$F(H)_6$
Offset	0	1	2	3	**Offset**	4	5	6	7

Back to Example

get(H)

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

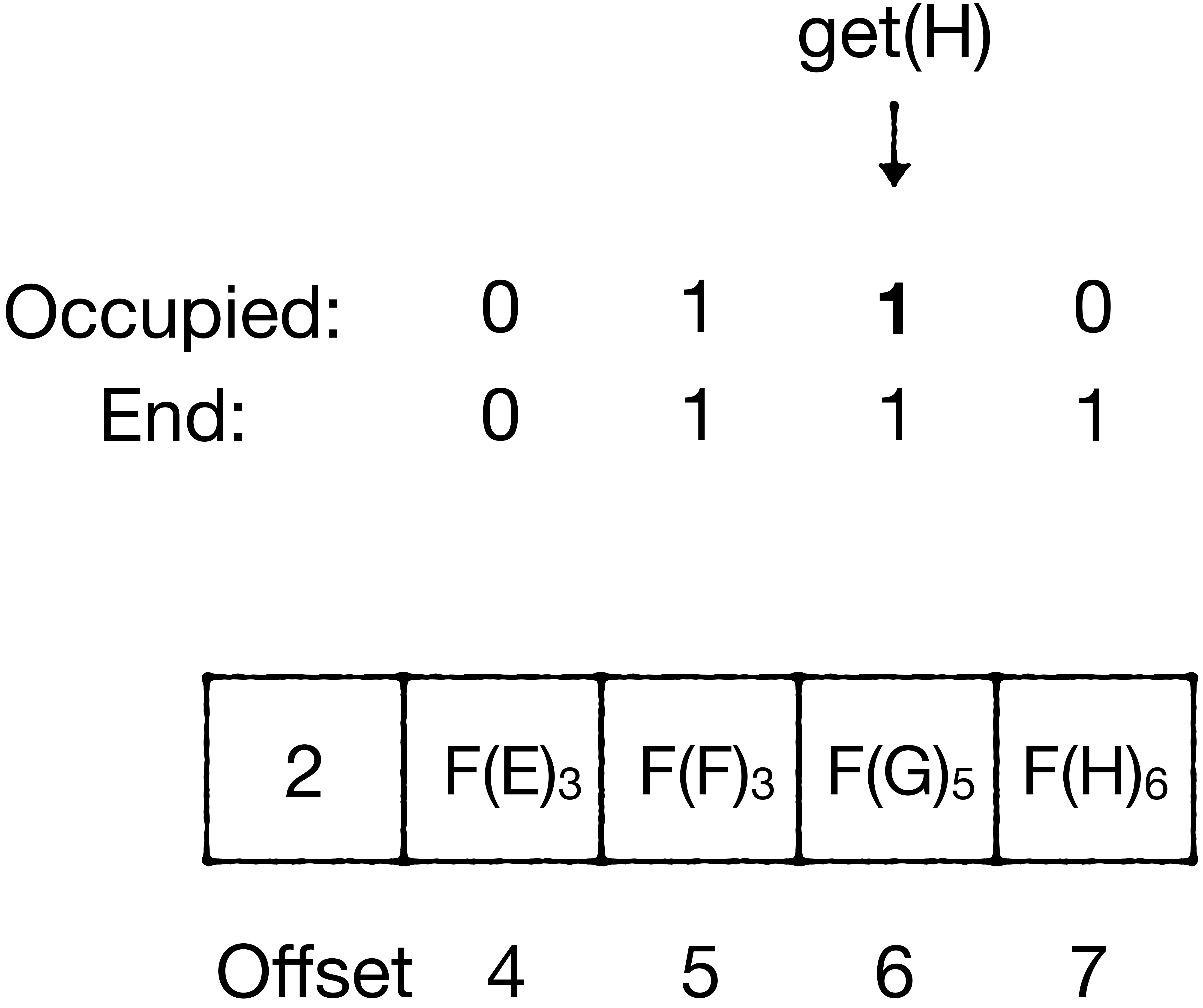
Back to Example



get(H)

↓

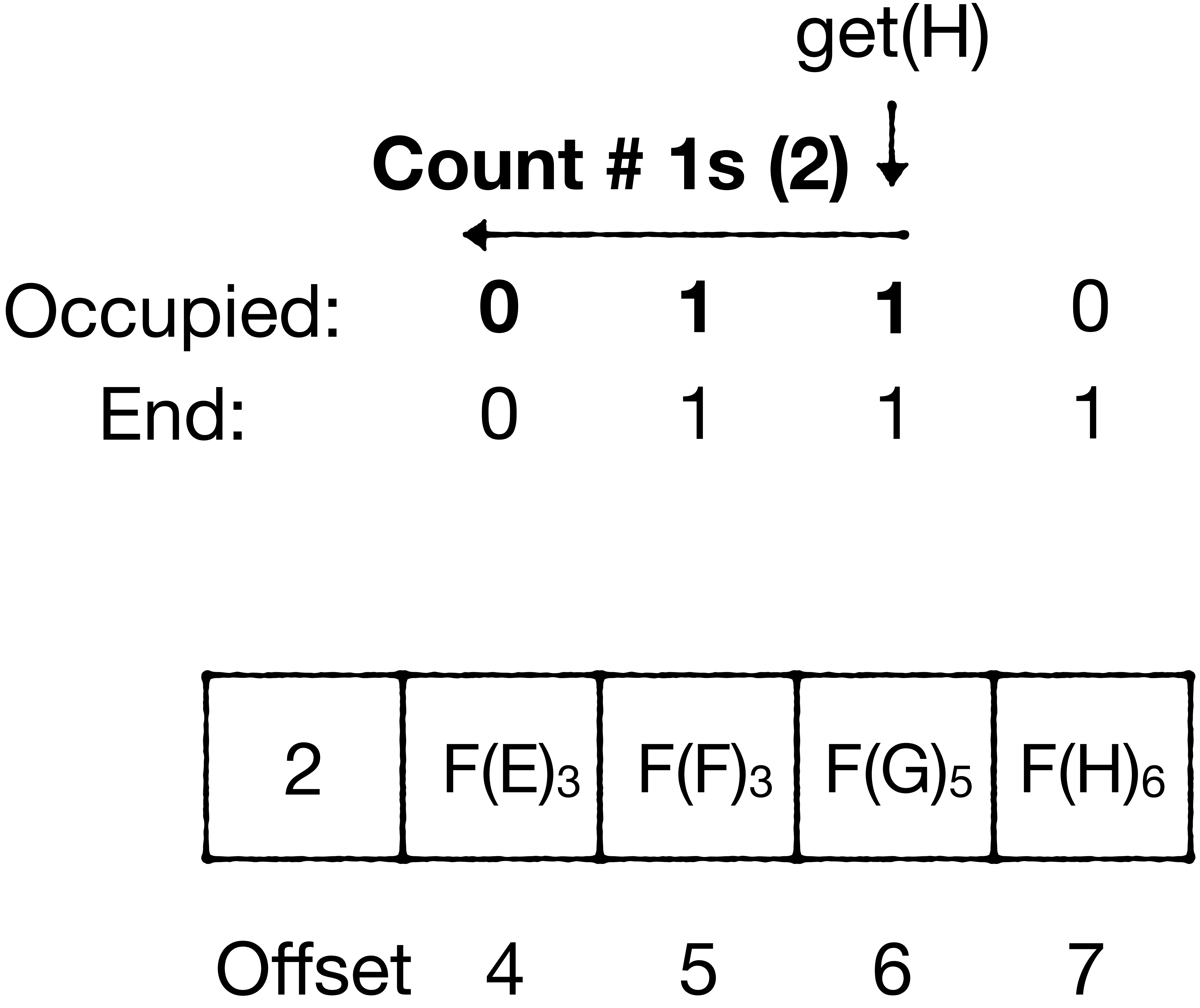
	0	1	1	0
	0	1	1	1
2	$F(E)_3$	$F(F)_3$	$F(G)_5$	$F(H)_6$
Offset	4	5	6	7



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

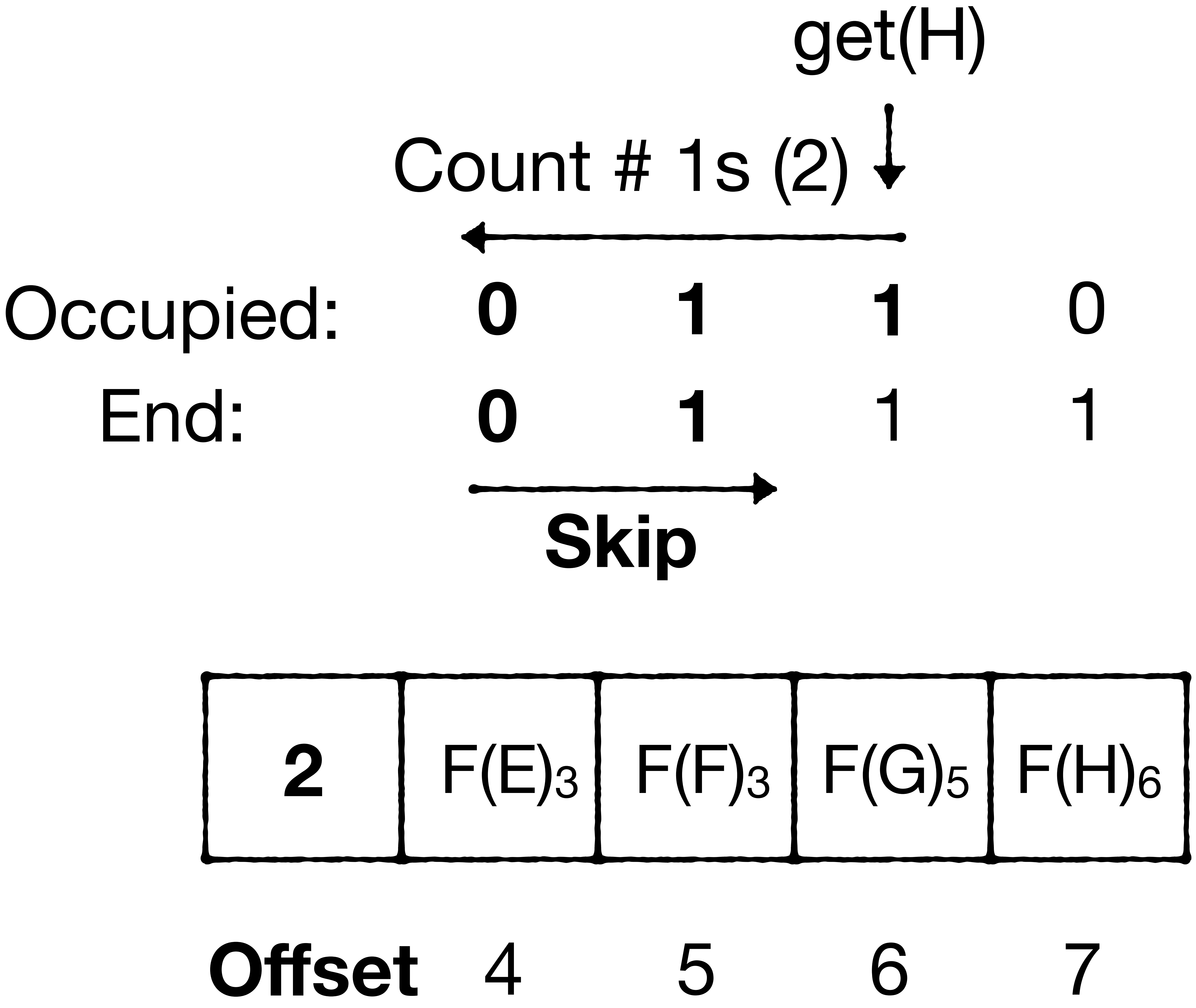


get(H)

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



get(H)

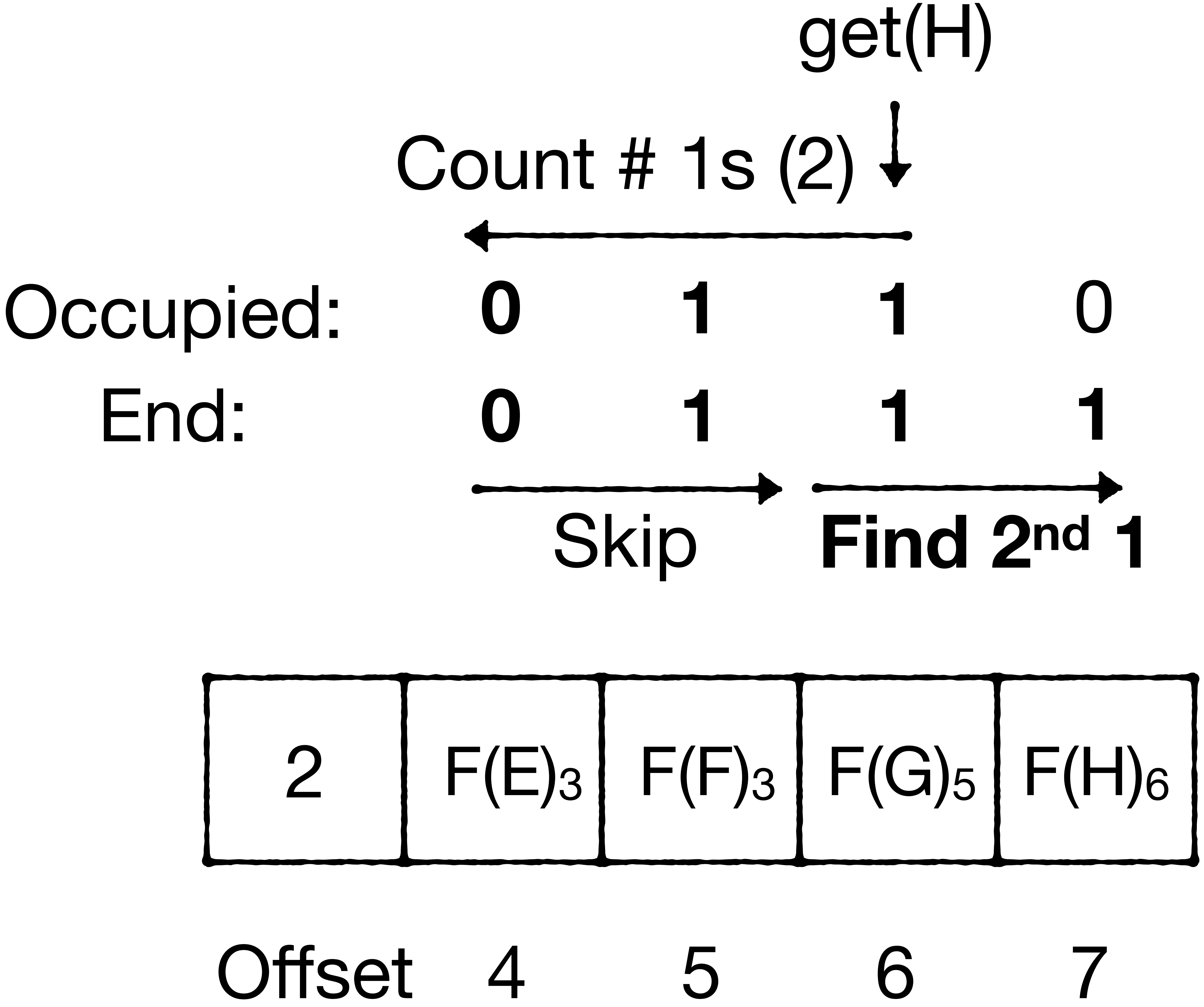
Count # 1s (2)

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

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

Skip

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



get(H)

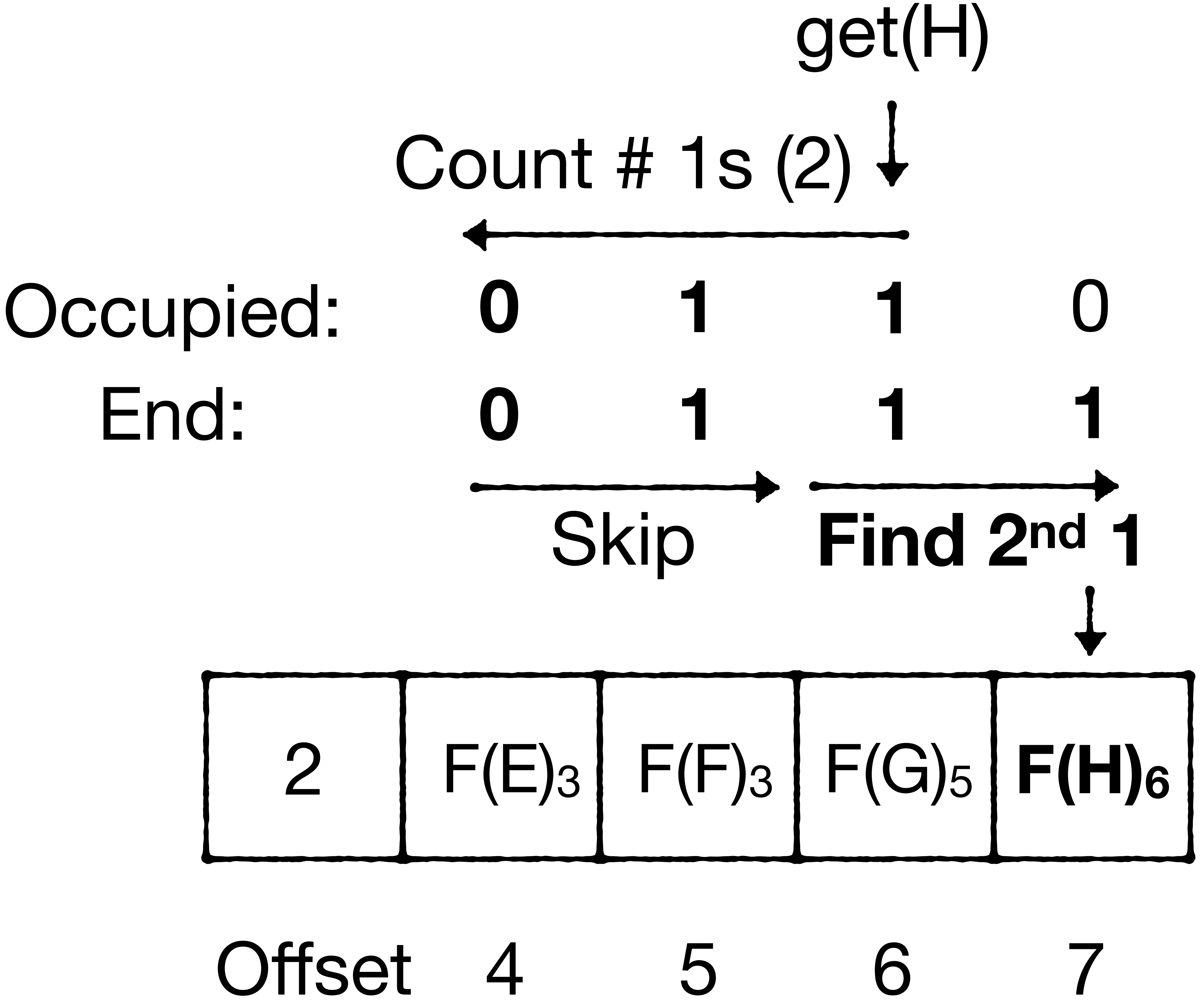
Count # 1s (2)

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)

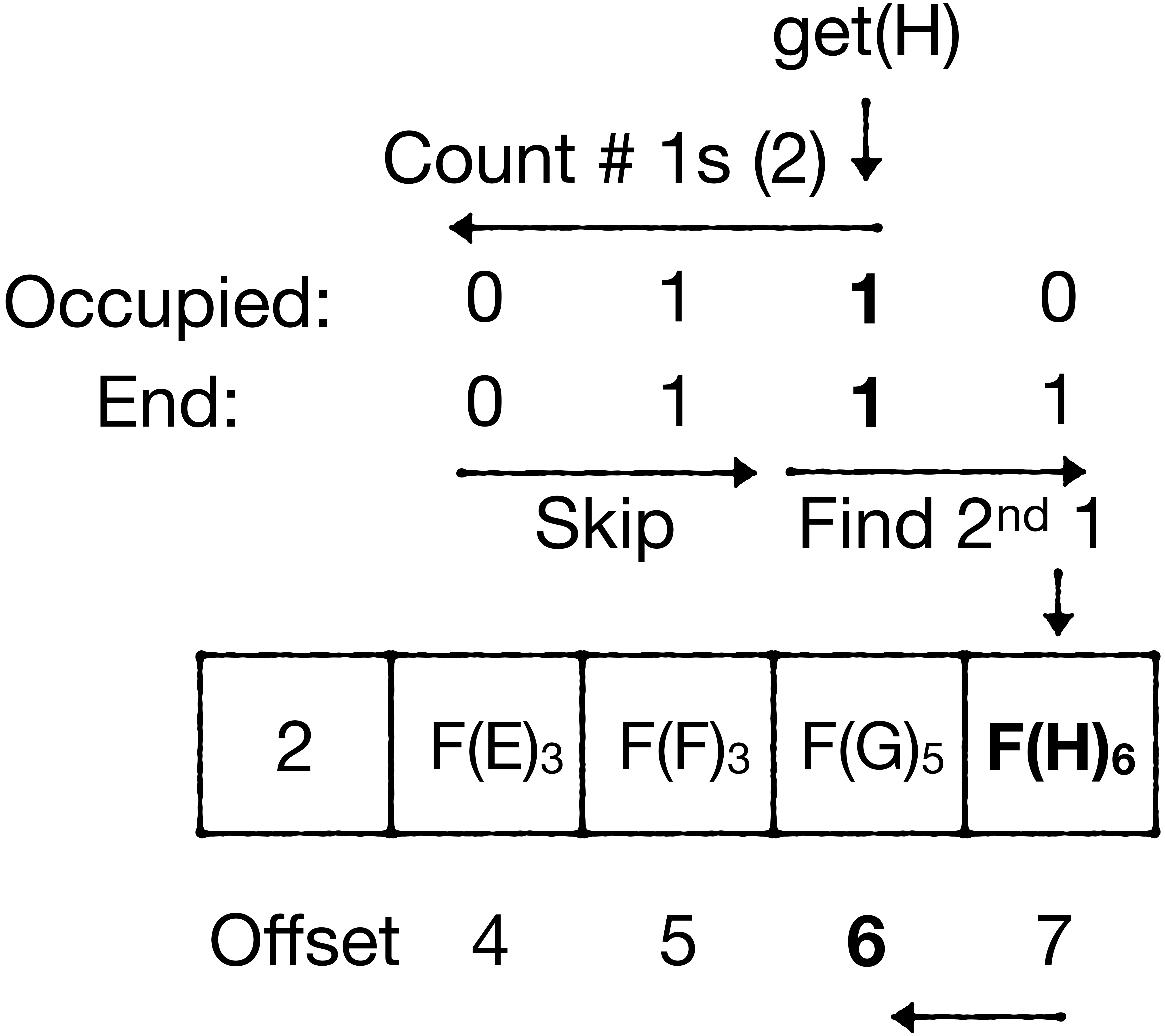
Count # 1s (2)

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 run ends or canonical slot

Target run may have been pushed to next chunk

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

Target run may have been pushed to next chunk

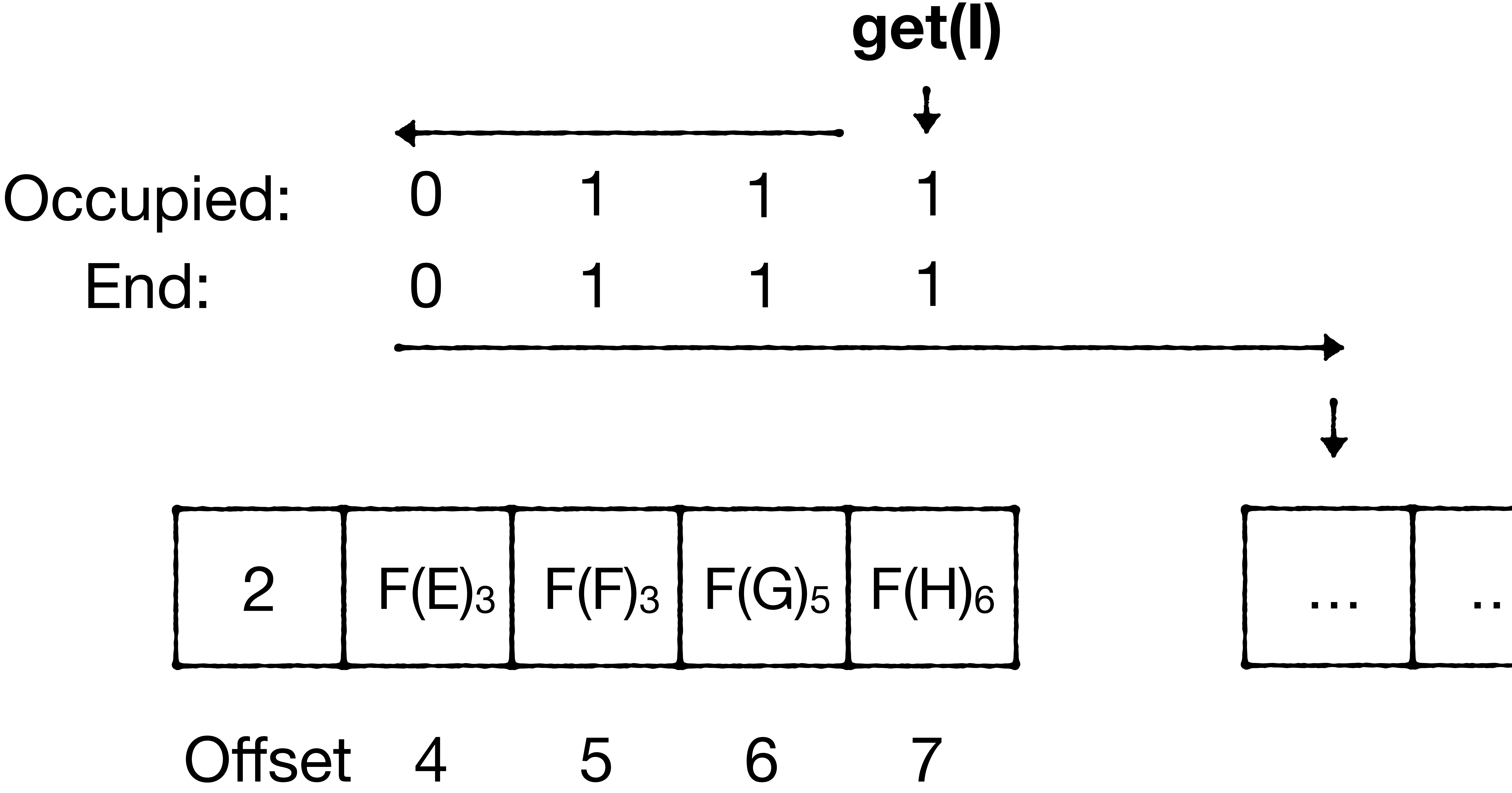
get(I)

↓

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

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

Target run may have been pushed to next chunk



get(I)

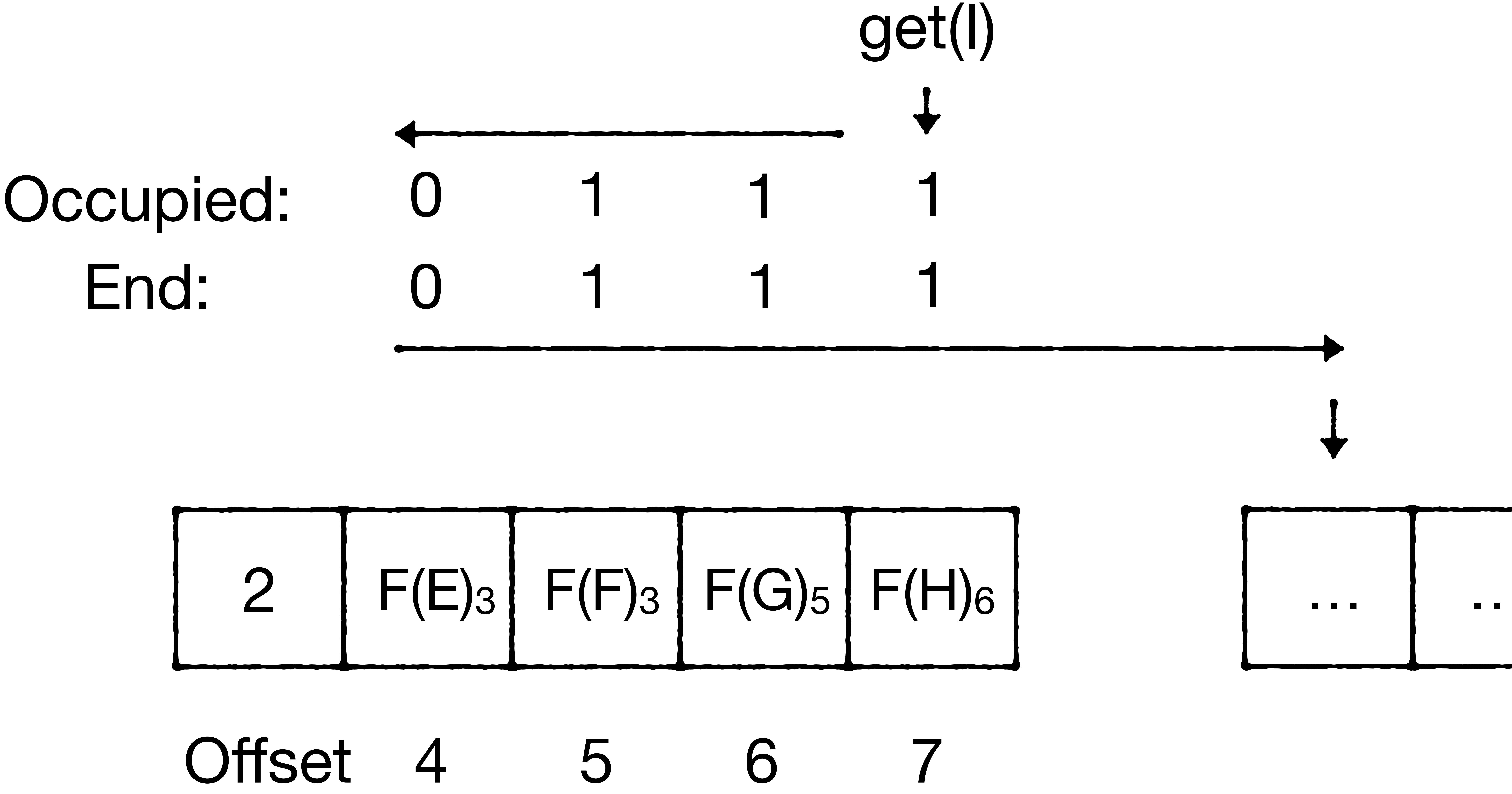
Occupied: 0 1 1 1

End: 0 1 1 1

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

Sequential cache misses, since chunks are adjacent

(not too shabby)



get(I)

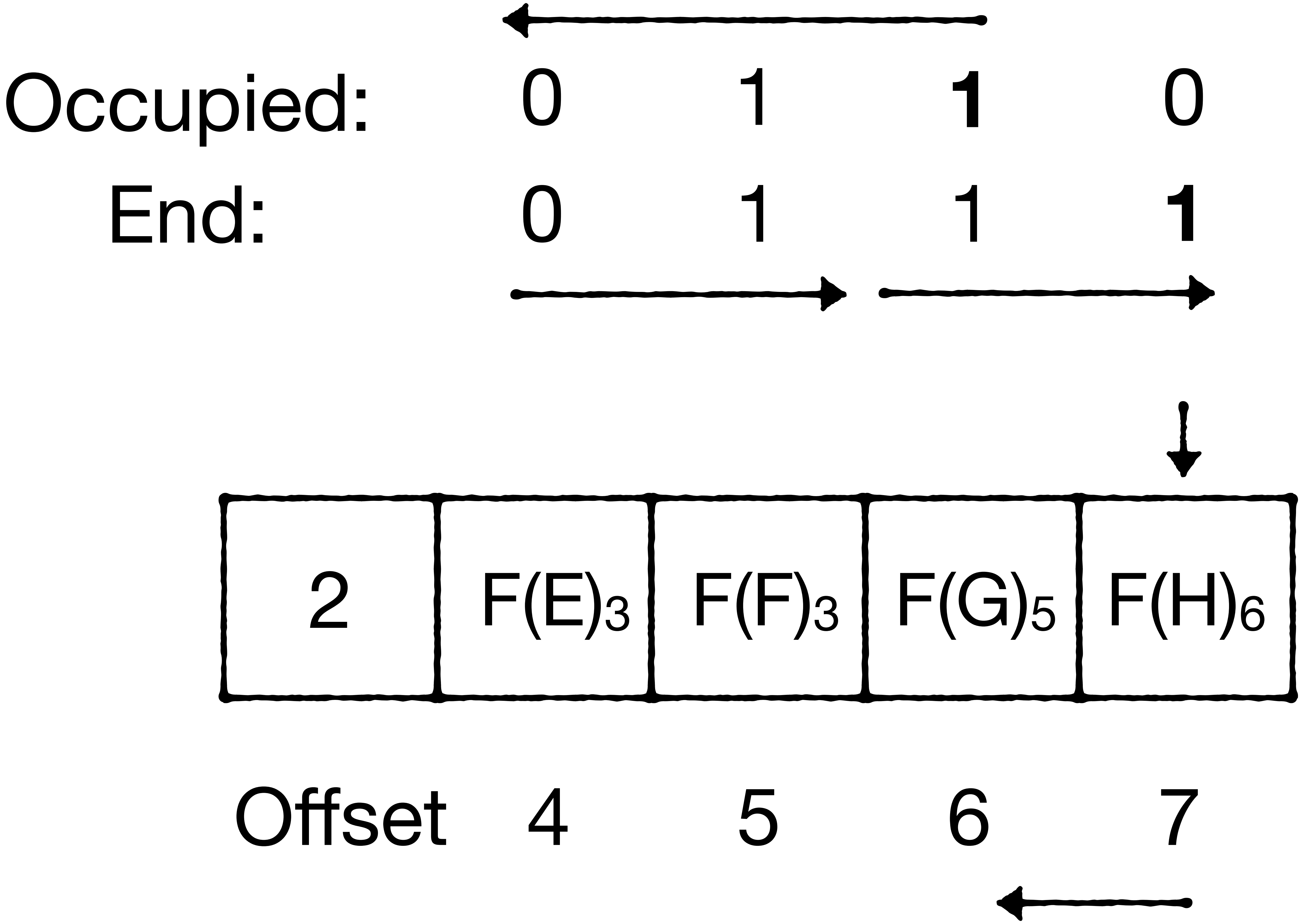
Occupied: 0 1 1 1

End: 0 1 1 1

2 | $F(E)_3$ | $F(F)_3$ | $F(G)_5$ | $F(H)_6$ … ..

Offset 4 5 6 7

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



Occupied: 0 1 **1** 0

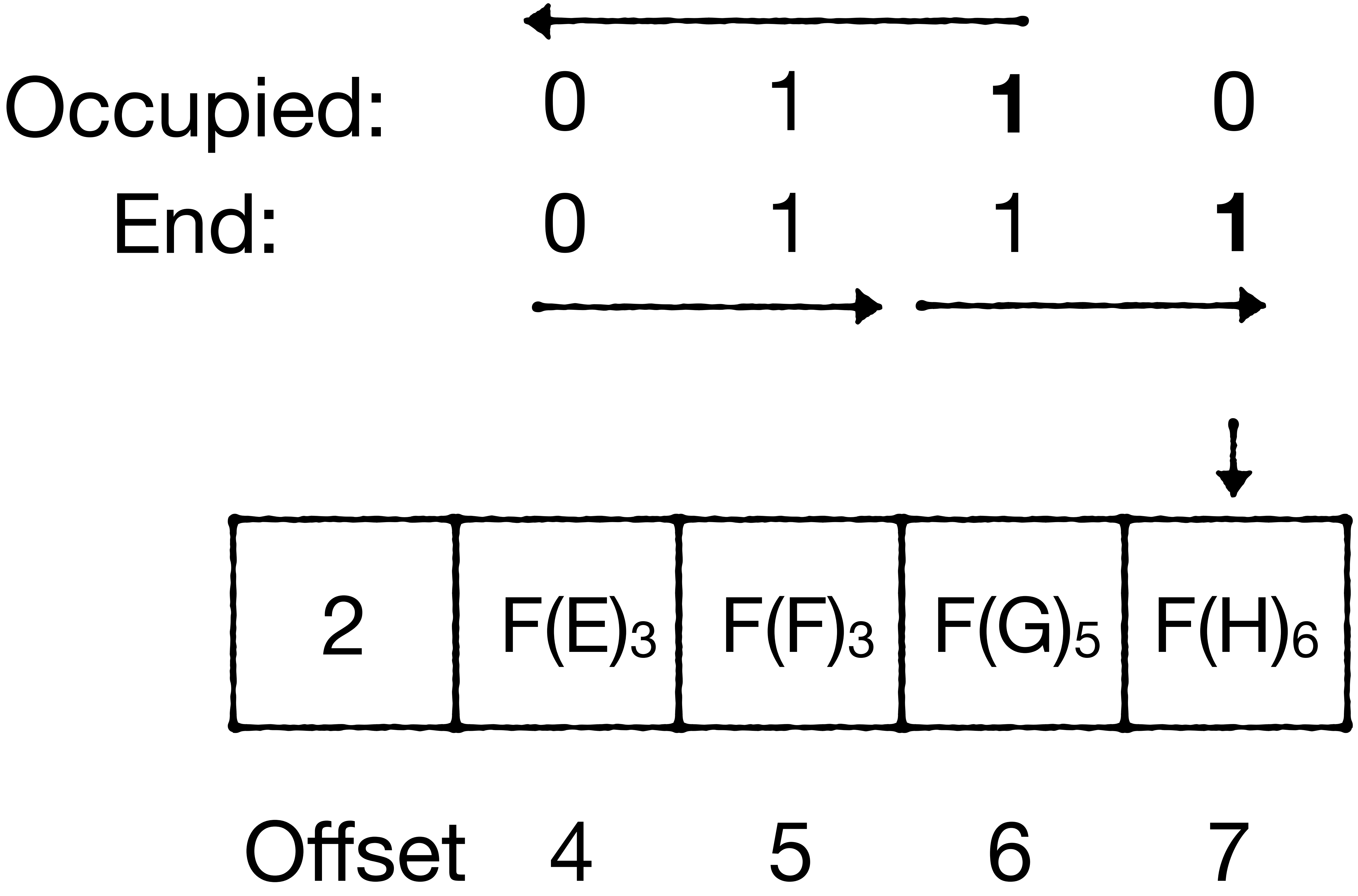
End: 0 1 1 **1**

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

Offset 4 5 6 7

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

Can we do O(1)?

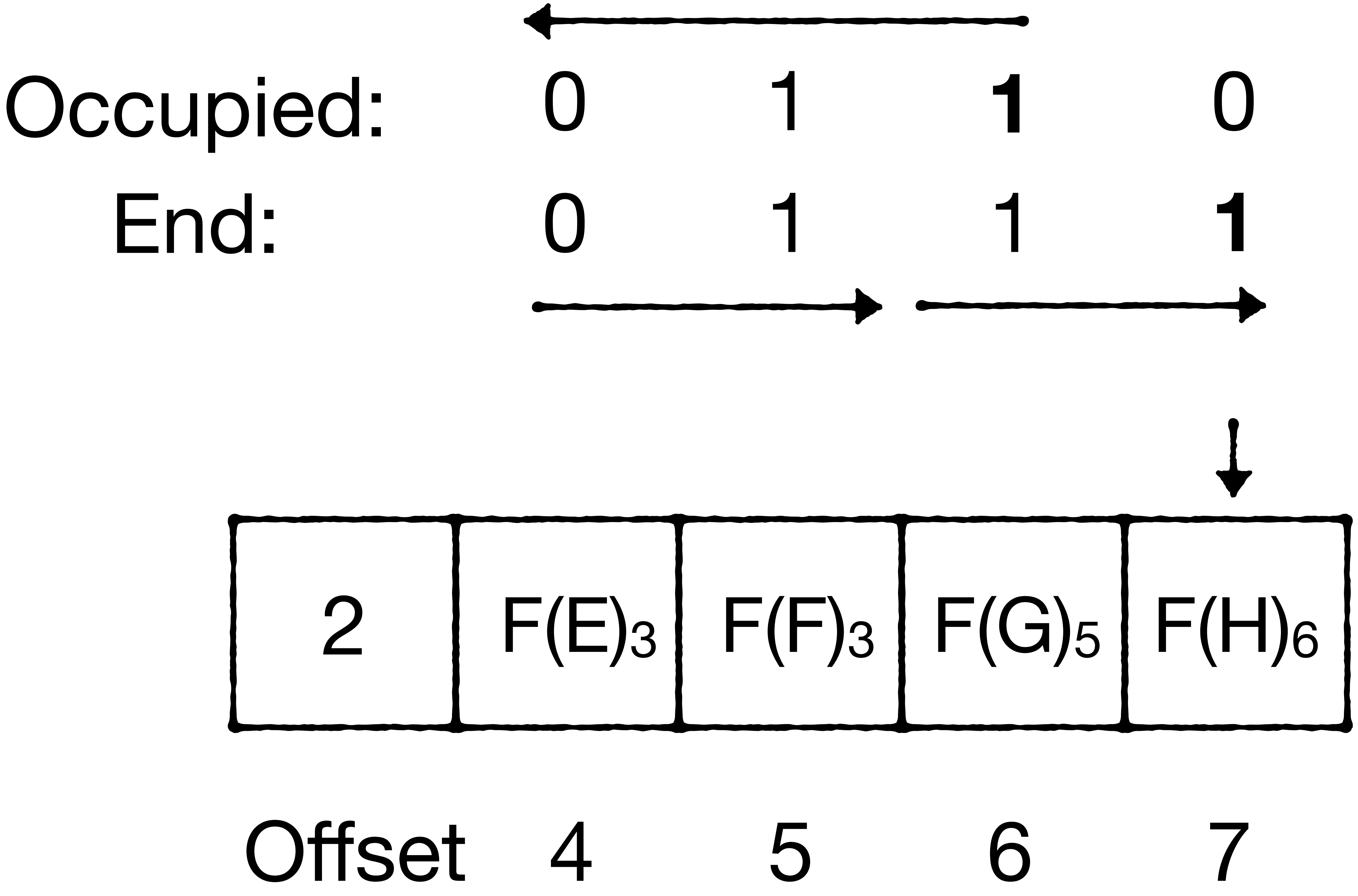


Occupied: 0 1 **1** 0

End: 0 1 1 **1**

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

Rank & Select



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

Rank & Select

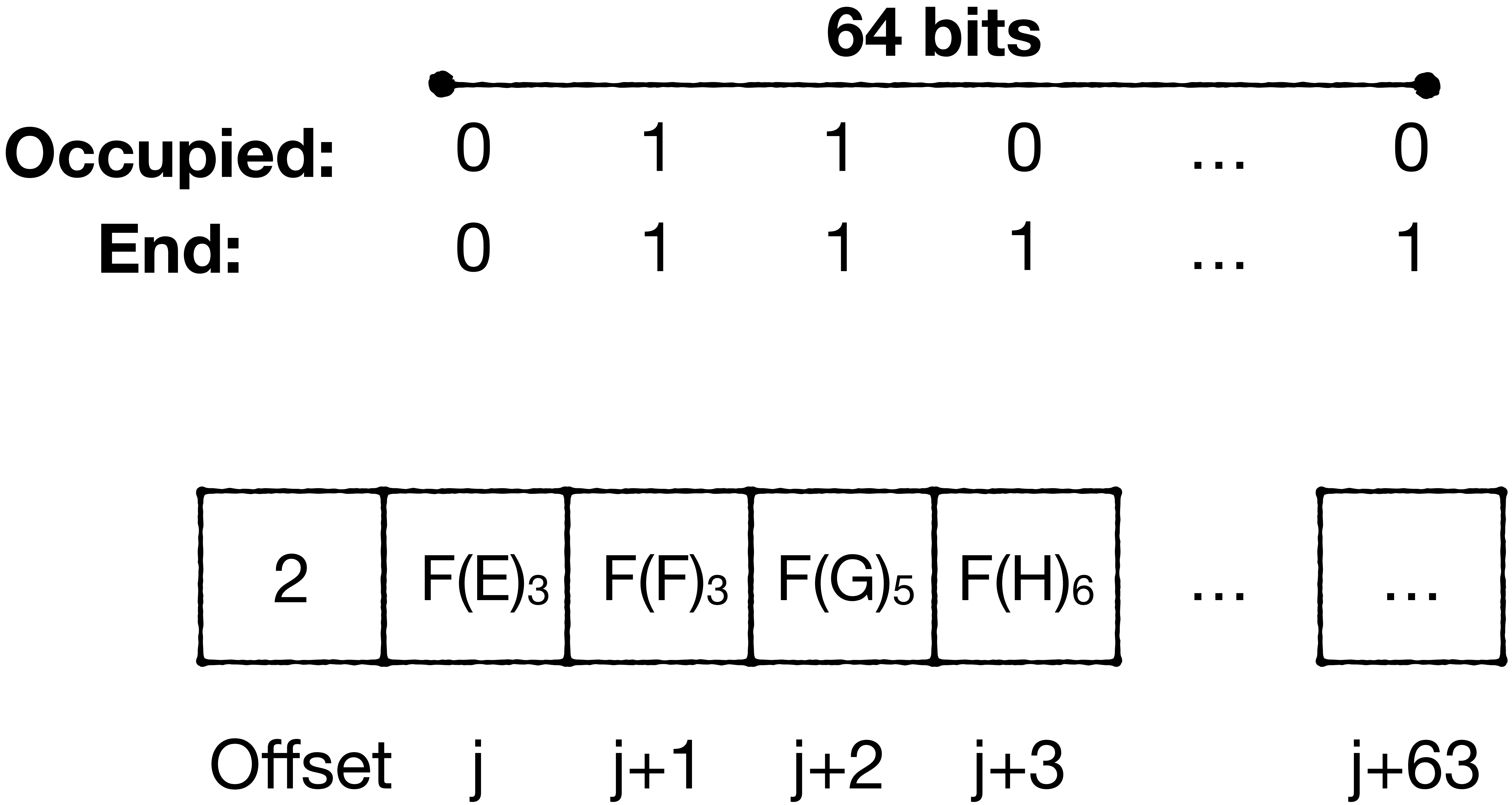
Can parse a 64-bit bitmap in constant time

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

Rank & Select

Can parse a 64-bit bitmap in constant time



64 bits

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

2	$F(E)_3$	$F(F)_3$	$F(G)_5$	$F(H)_6$	…	…
Offset	j	j+1	j+2	j+3		j+63

Rank(i) counts # 1s before the ith bit

Select (i)

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

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

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

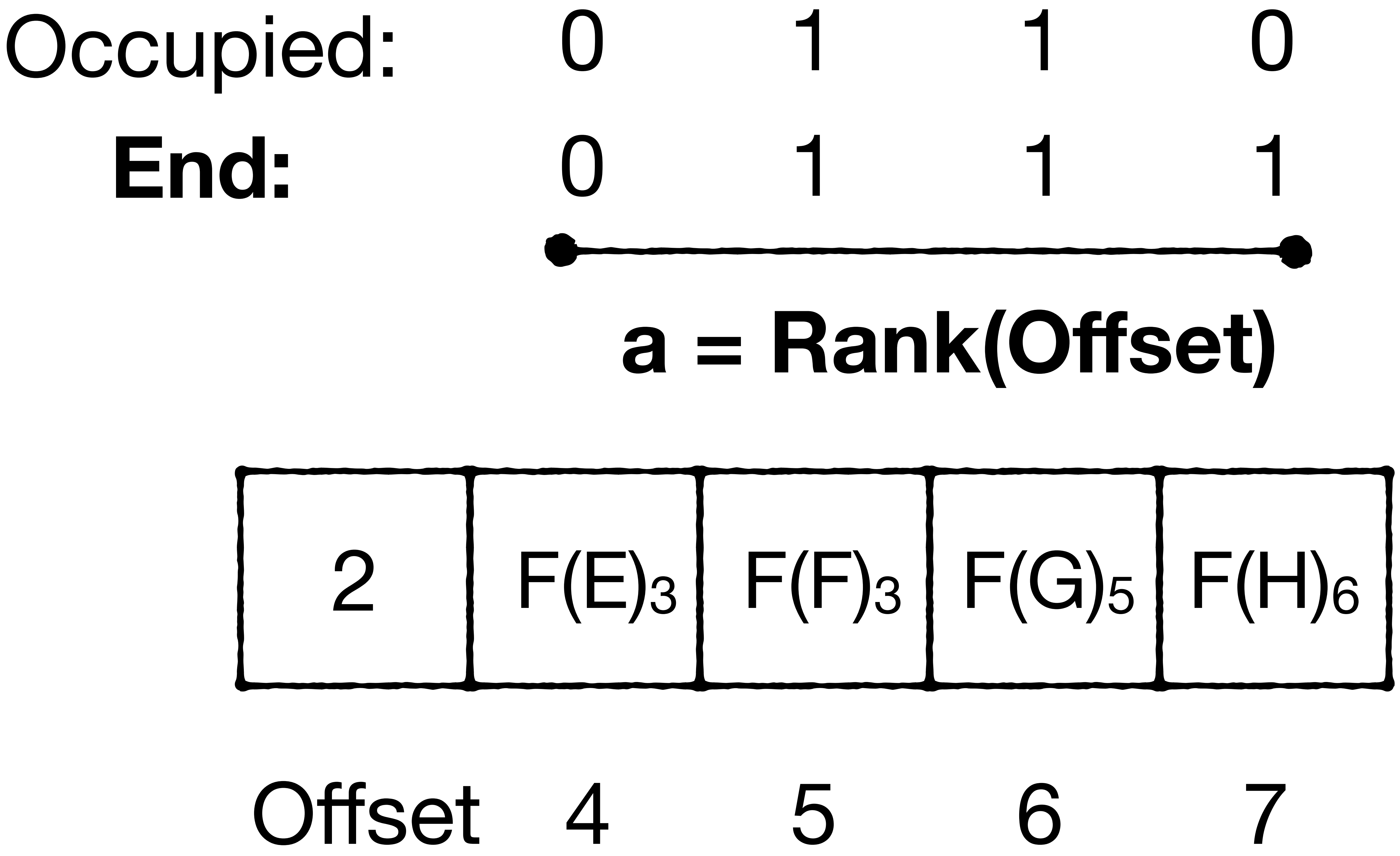
2	$F(E)_3$	$F(F)_3$	$F(G)_5$	$F(H)_6$
Offset	4	5	6	7

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

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



Occupied: 0 1 1 0

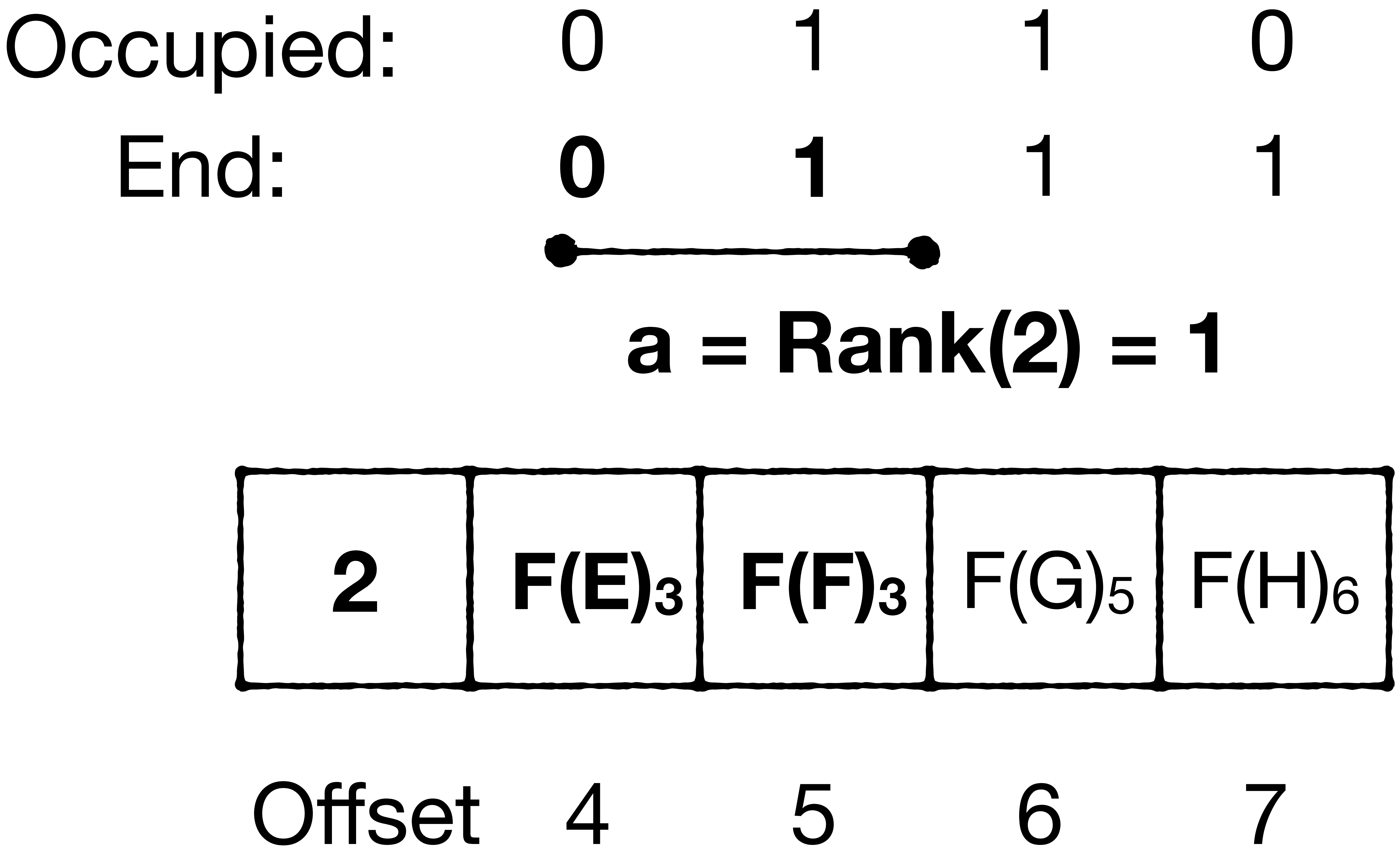
End: 0 1 1 1

a = Rank(Offset)

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

Offset 4 5 6 7

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



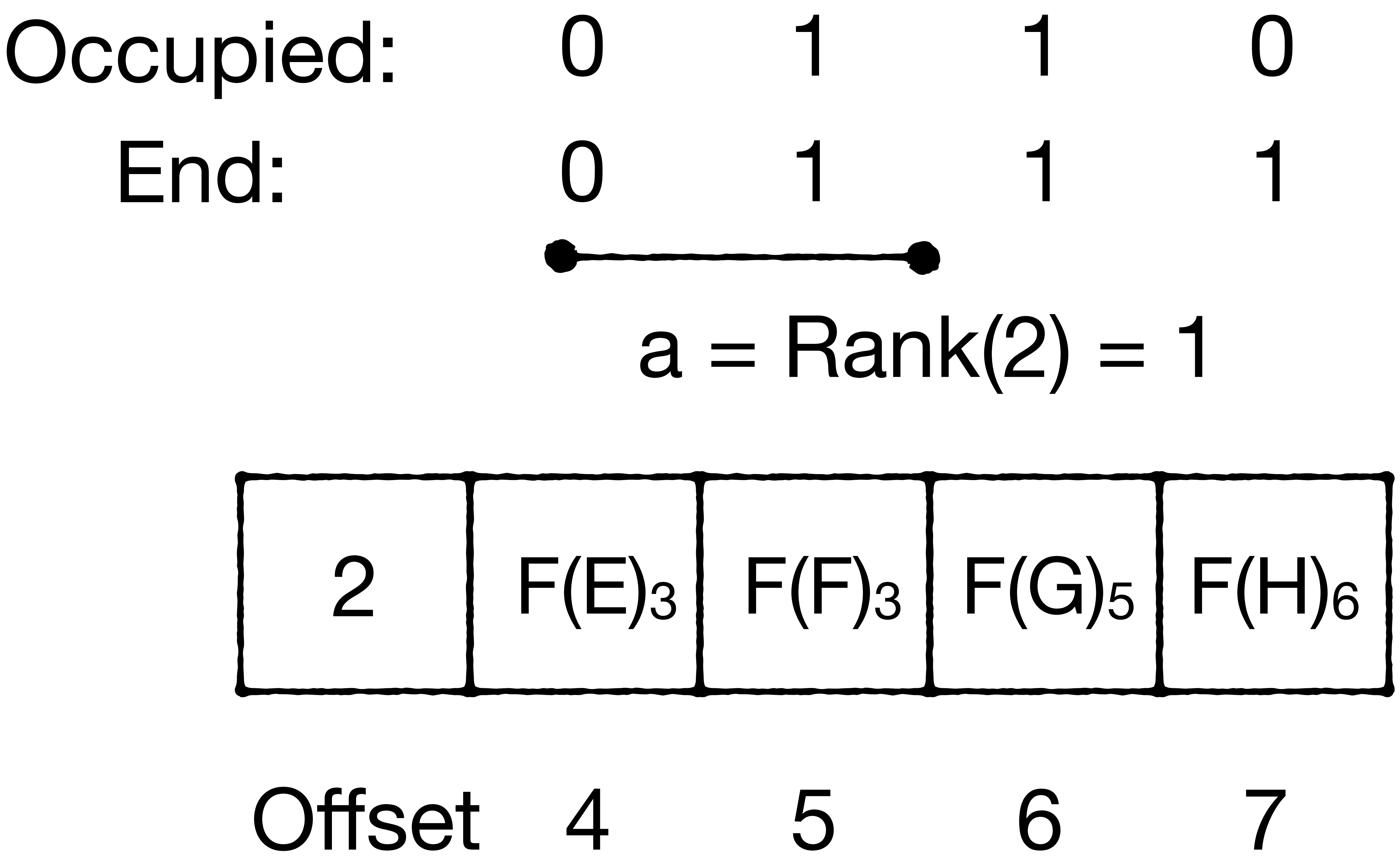
Occupied: 0 1 1 0

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

a = Rank(2) = 1

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

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



Occupied: 0 1 1 0

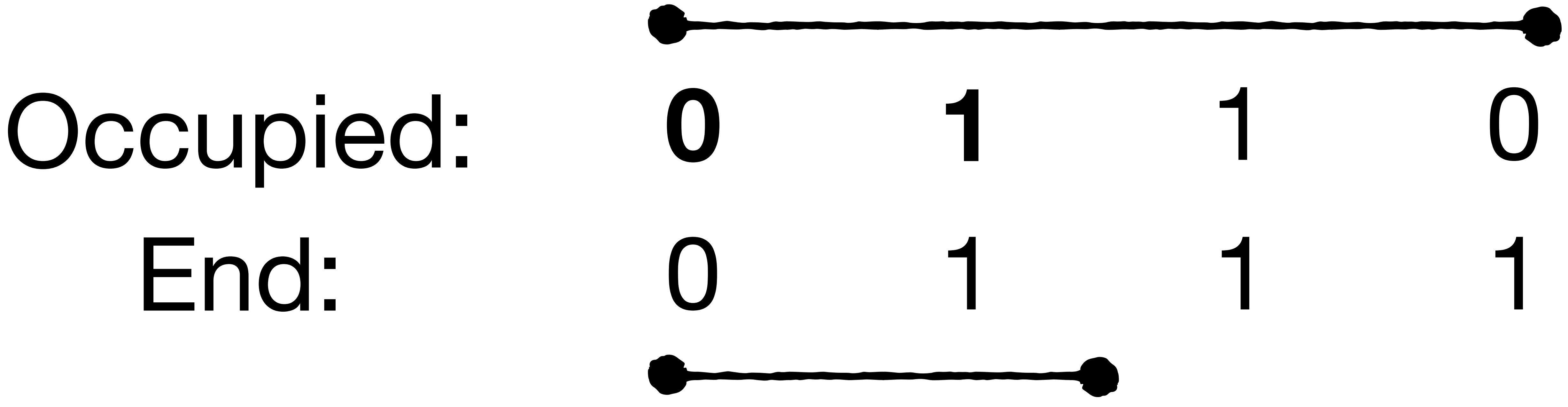
End: 0 1 1 1

a = Rank(2) = 1

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

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

b = Rank(*targetSlot* **-** *firstChunkSlot***)**

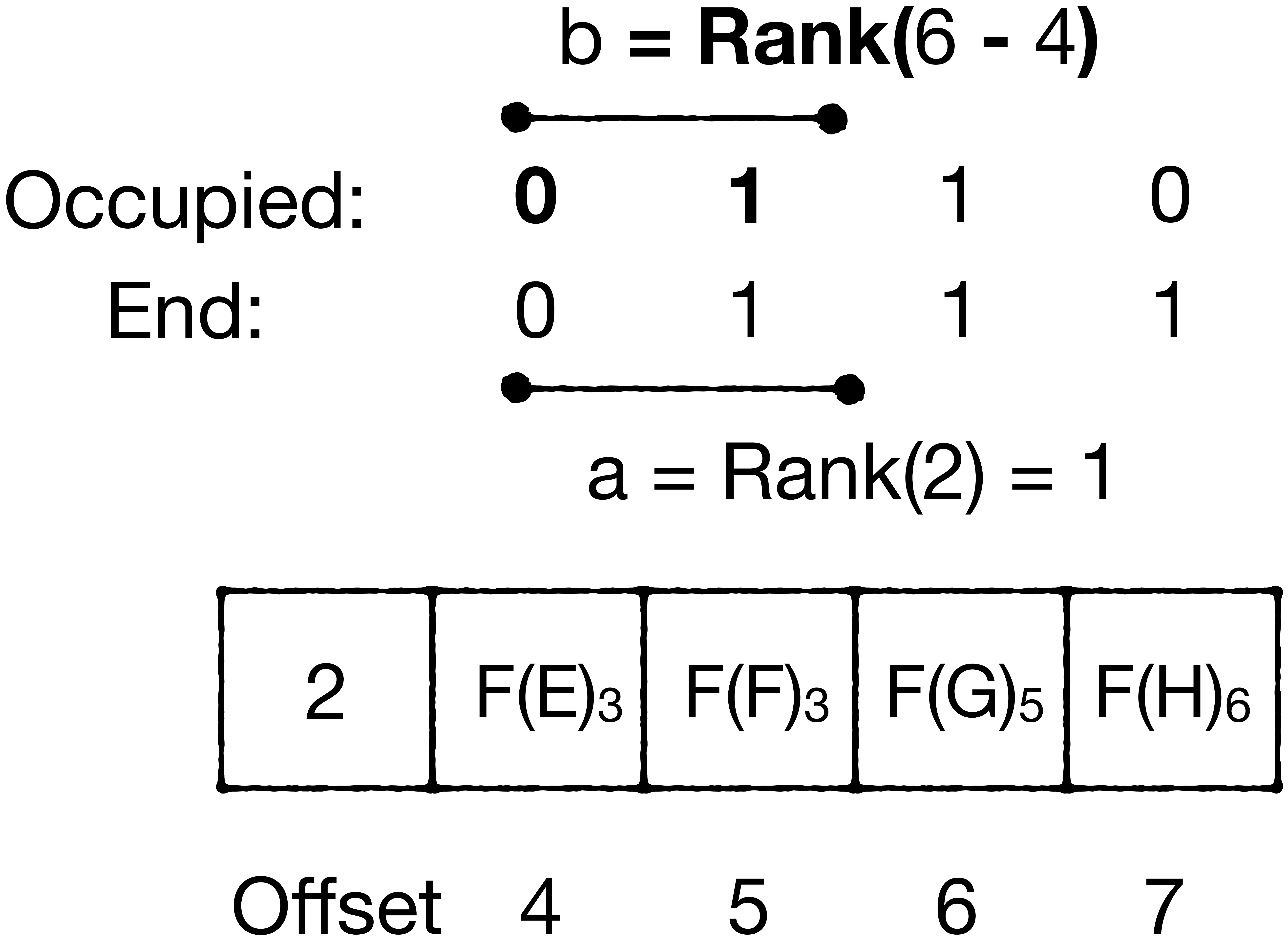


Occupied:	**0**	**1**	1	0
End:	0	1	1	1

a = Rank(2) = 1

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

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



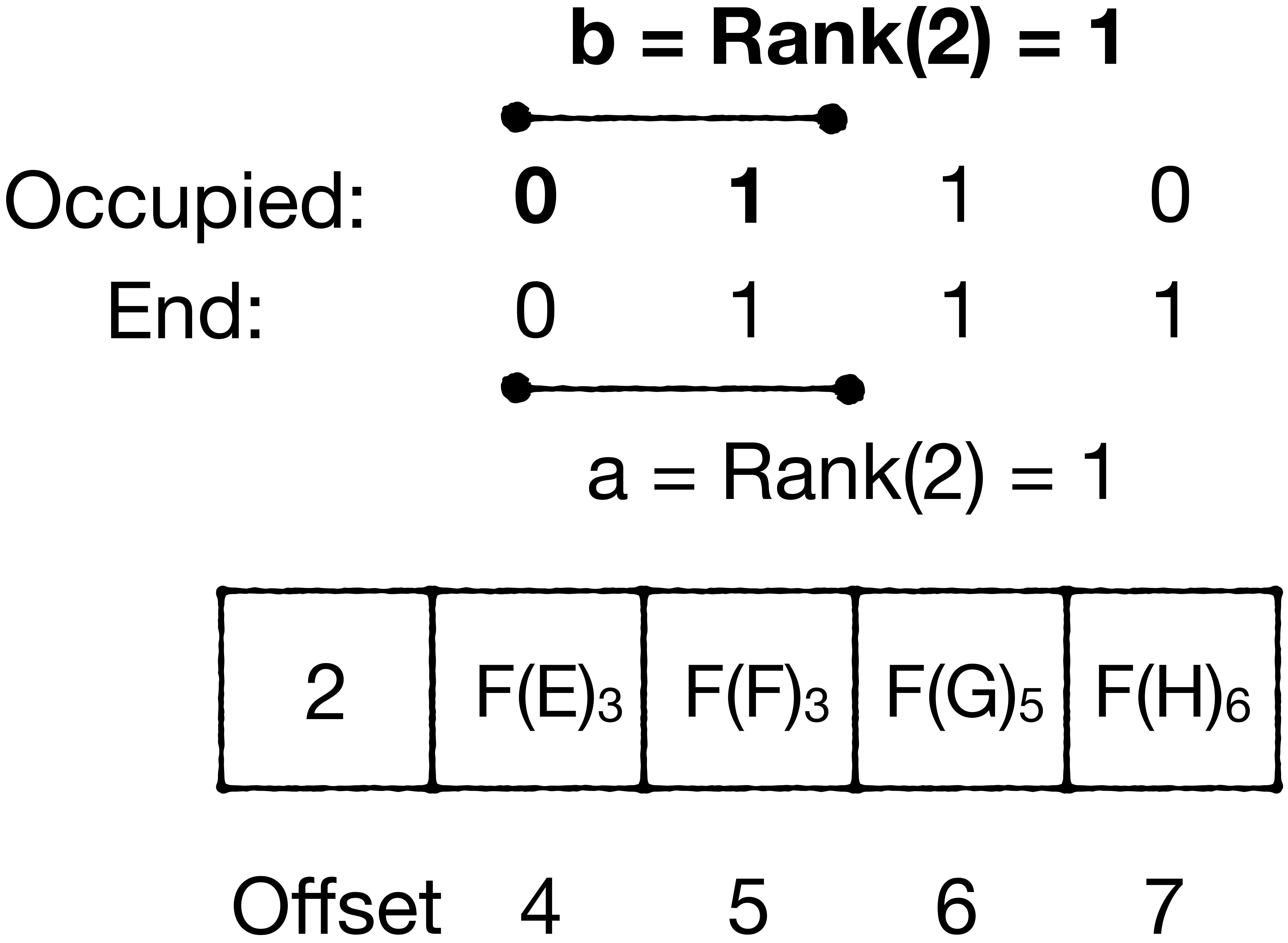
b = **Rank(**6 - 4**)**

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

a = Rank(2) = 1

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

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



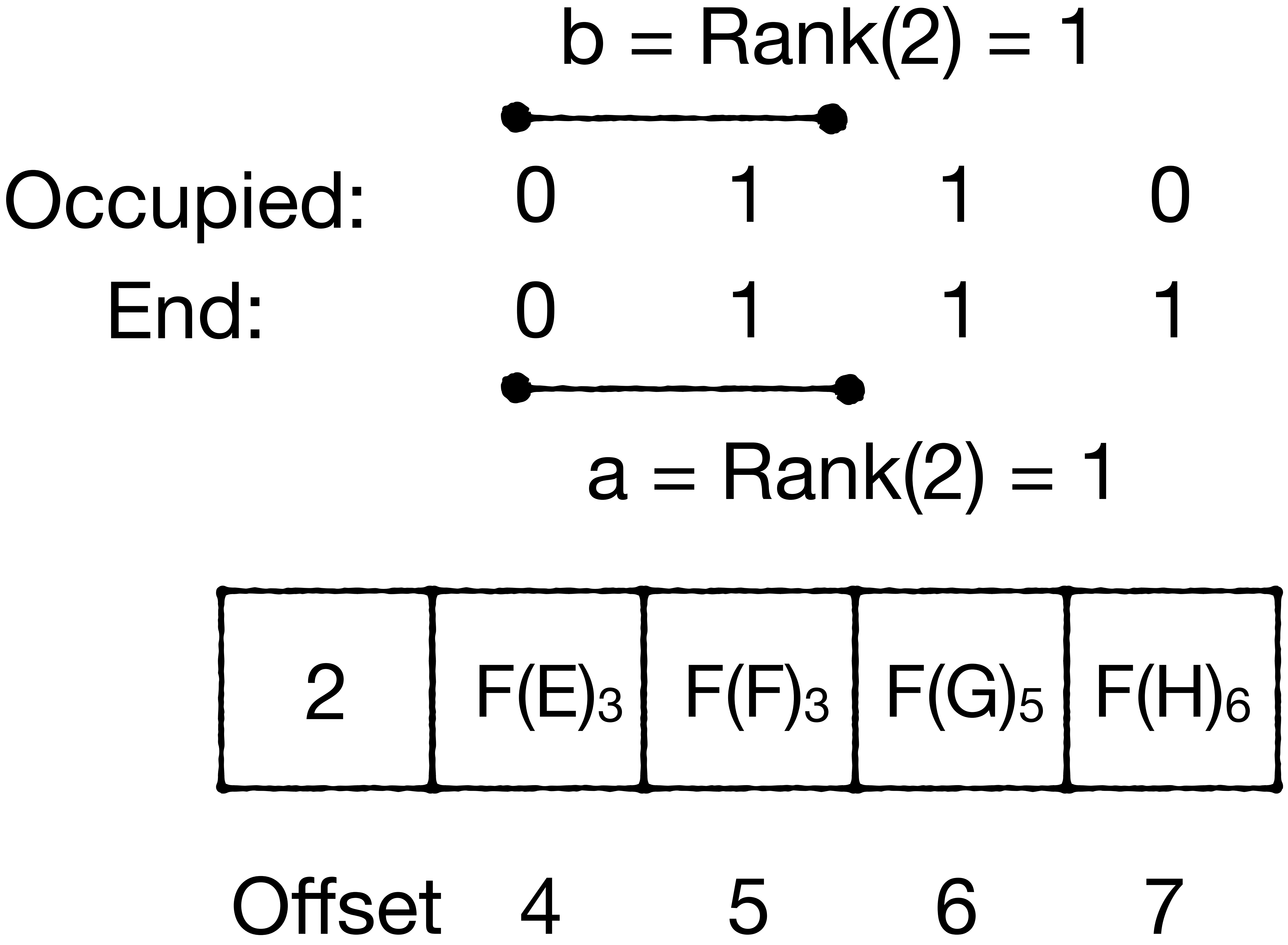
b = Rank(2) = 1

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

a = Rank(2) = 1

	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



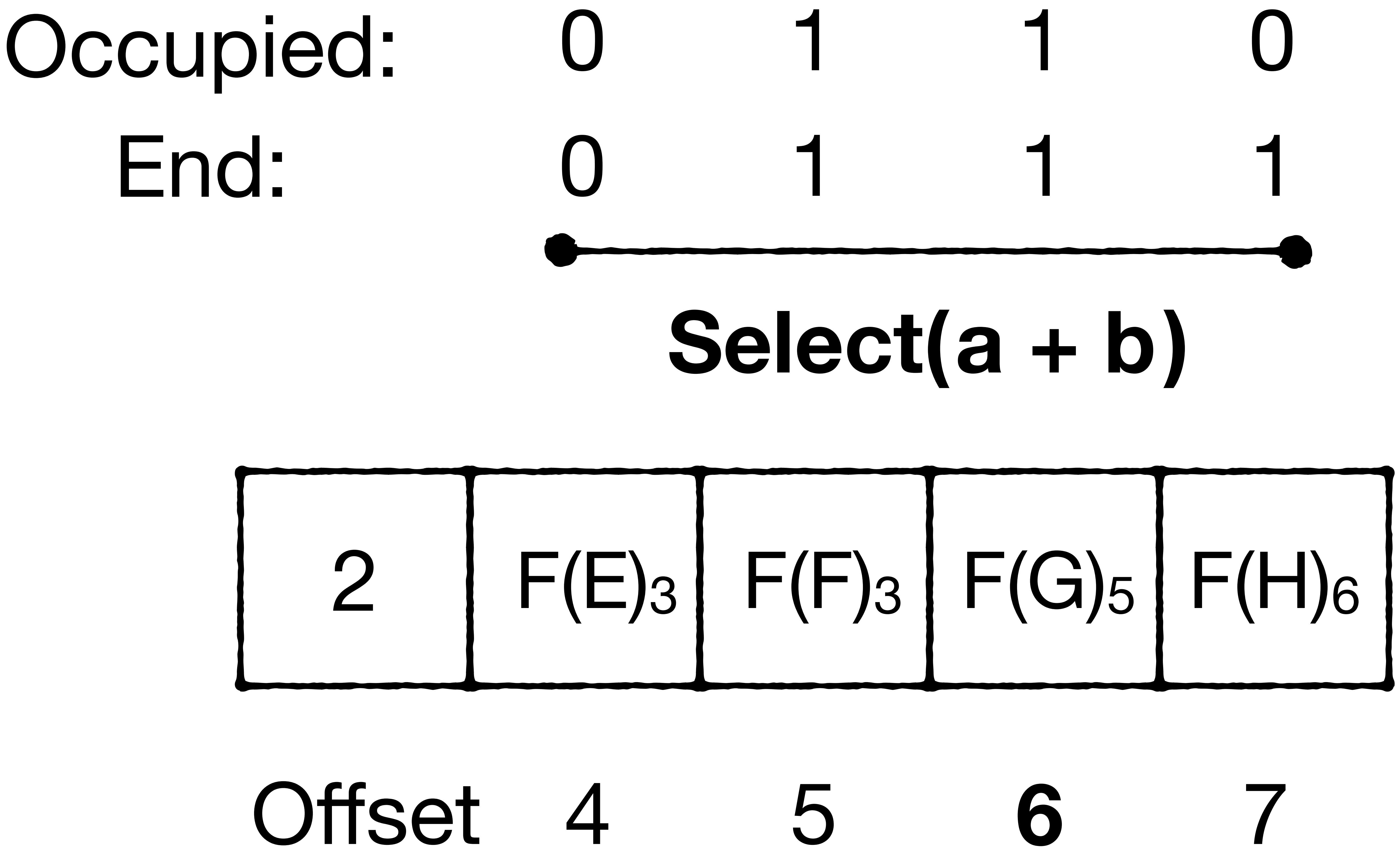
b = Rank(2) = 1

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

a = Rank(2) = 1

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

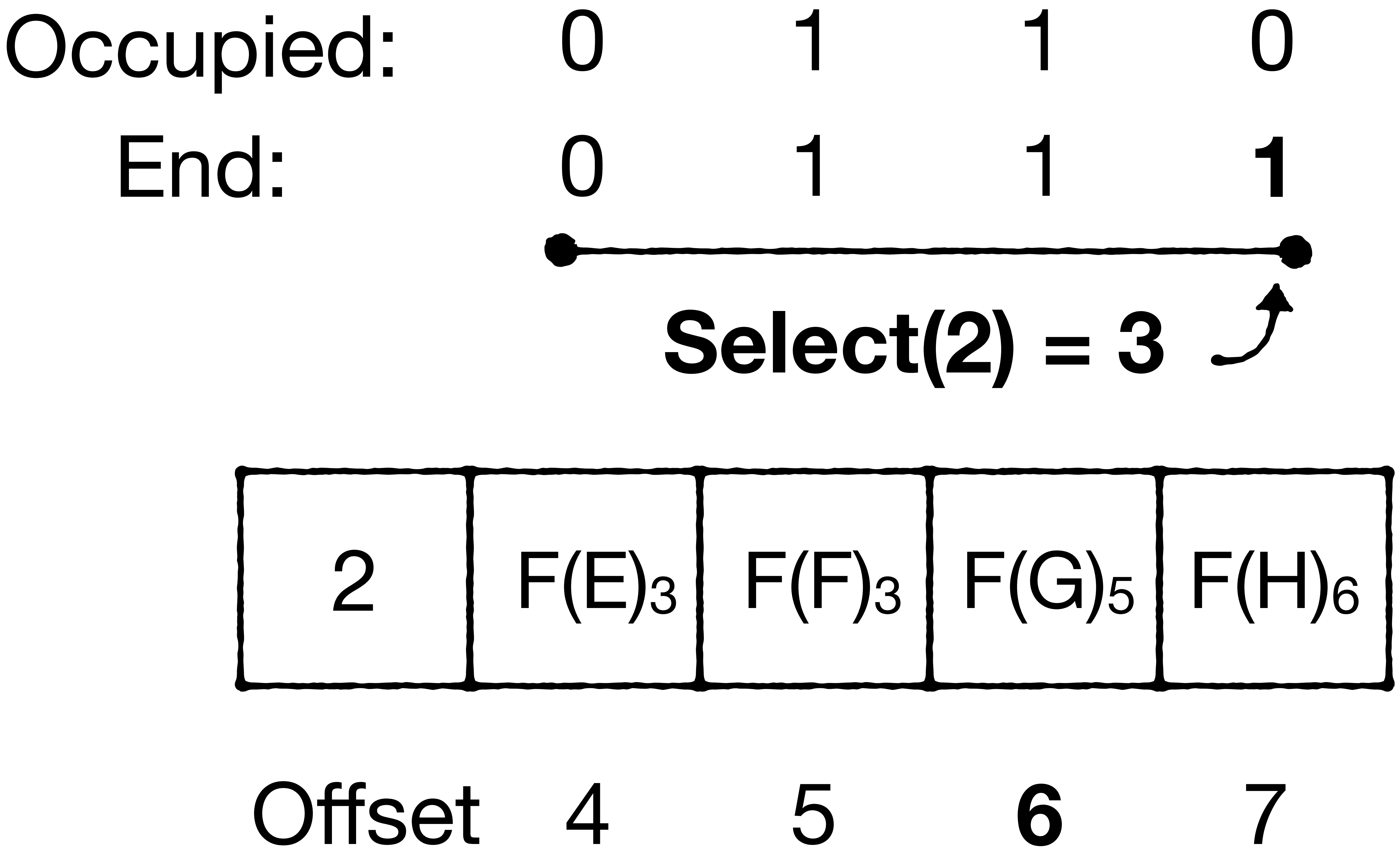


Occupied:	0	1	1	0
End:	0	1	1	1

Select(a + b)

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



Occupied: 0 1 1 0

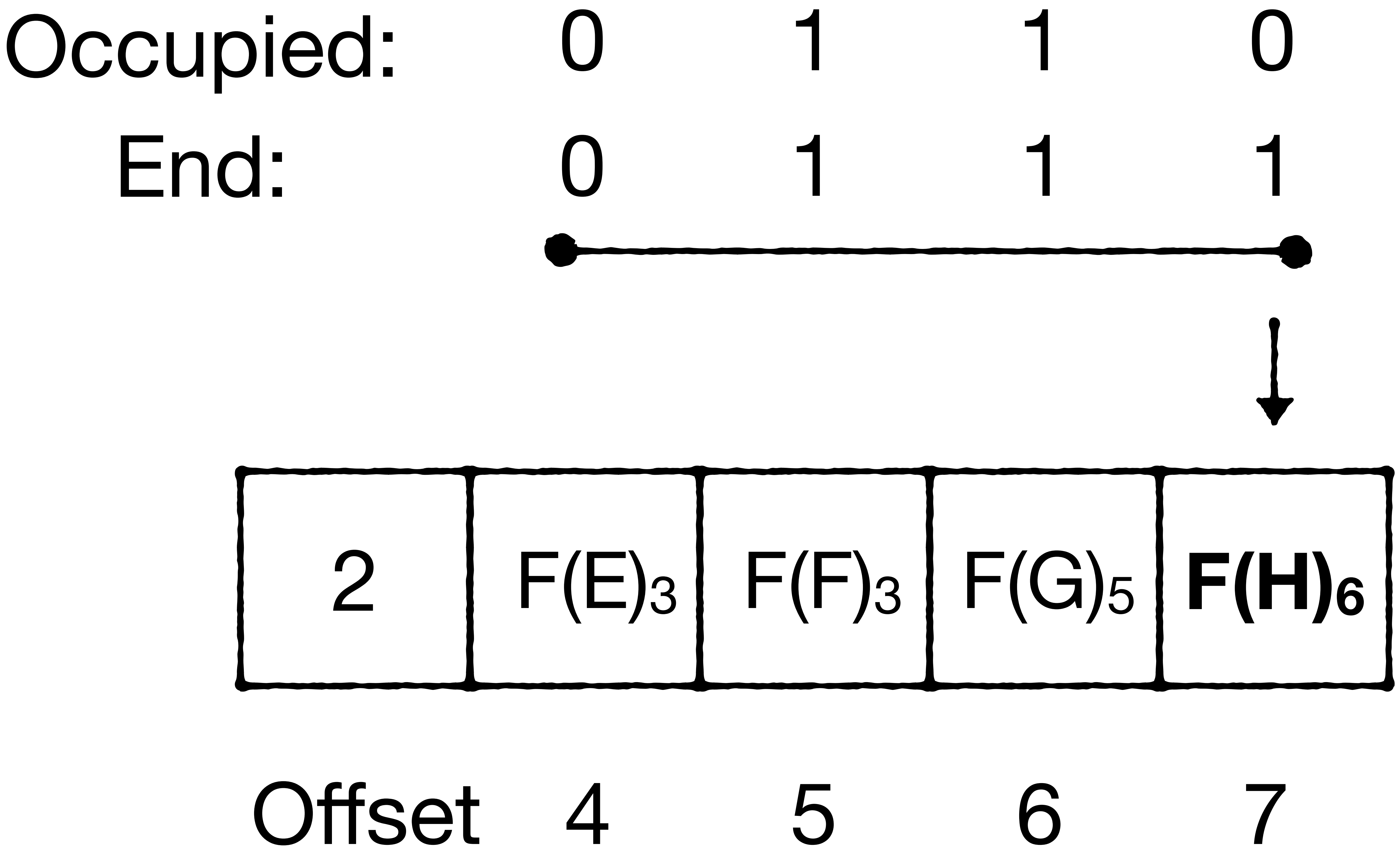
End: 0 1 1 **1**

Select(2) = 3

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

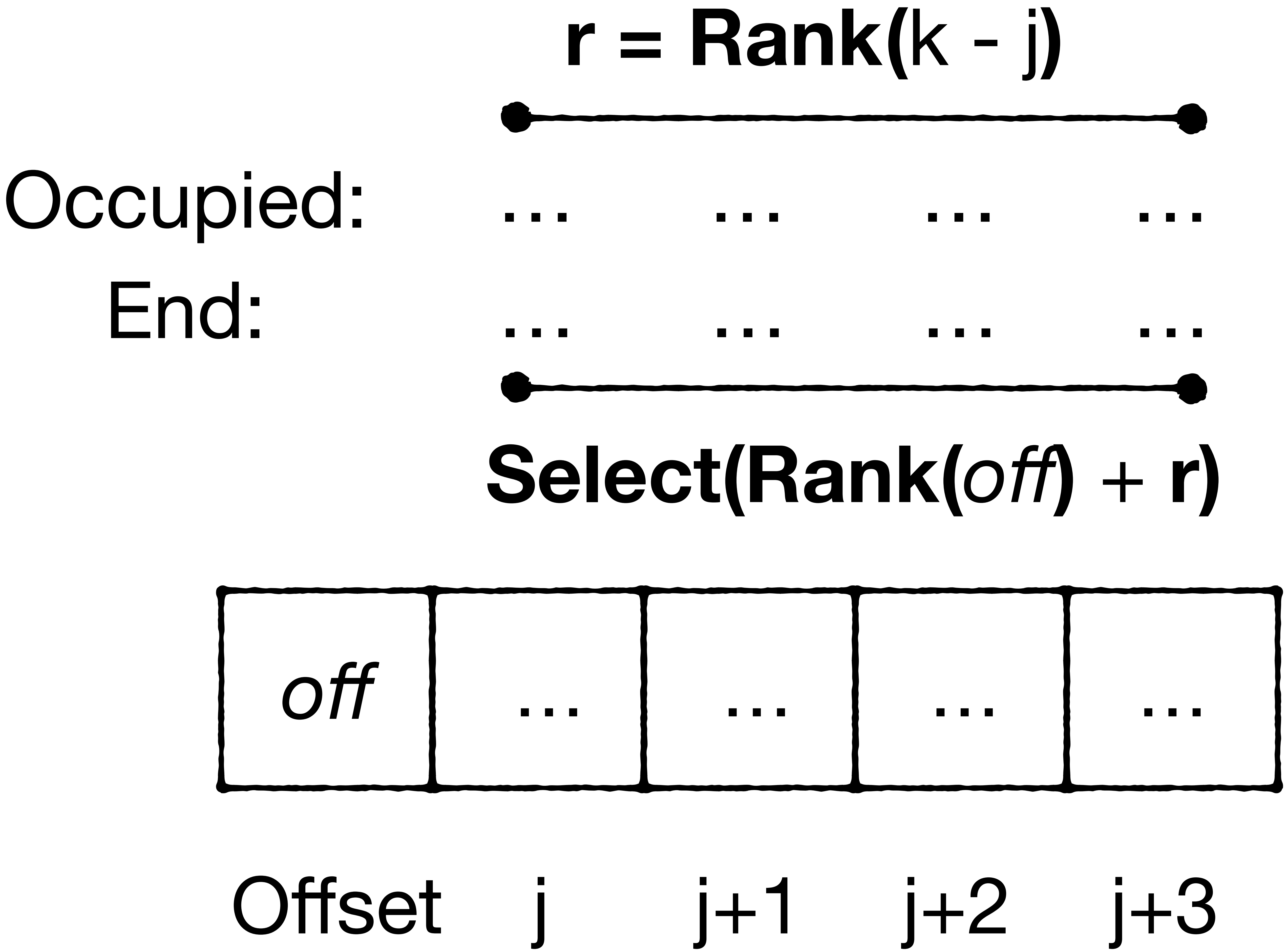
Offset 4 5 **6** 7

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



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

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



r = Rank(k - j**)**

Occupied: … … … …

End: … … … …

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

off	…	…	…	…
Offset	j	j+1	j+2	j+3

Implementing Rank and Select Efficiently

Implementing Rank and Select Efficiently

No looping

Implementing Rank Efficiently

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

Implementing Rank Efficiently

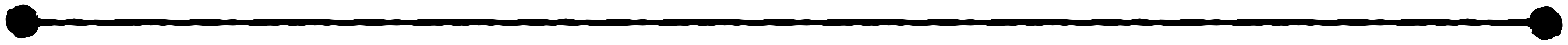
$$\text{rank}(i) = \text{popcount}(\mathbf{B}\ \&\ (2^{i} - 1))$$

Bitmap (64 bits long)

Implementing Rank Efficiently

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

Bitmap (64 bits long)



Least significant bits

Most significant bits

Implementing Rank Efficiently

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

↑

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**

rank(i) = 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

rank(i) = 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

rank(i) = 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

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

Implementing Select Efficiently

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

↑

Bitmap (64 bits long)

Implementing Select Efficiently

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

Count trailing zeros

Implementing Select Efficiently

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

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

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

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

Implementing Select Efficiently

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

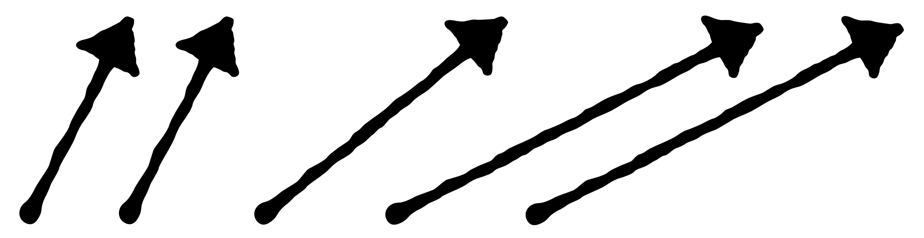
e.g., B = 0 1 1 0 1 0 1 1 Select(2) = 4

$\mathbf{2}^{2}$ = 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

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

↑

Only the 1 at relevant position is now set

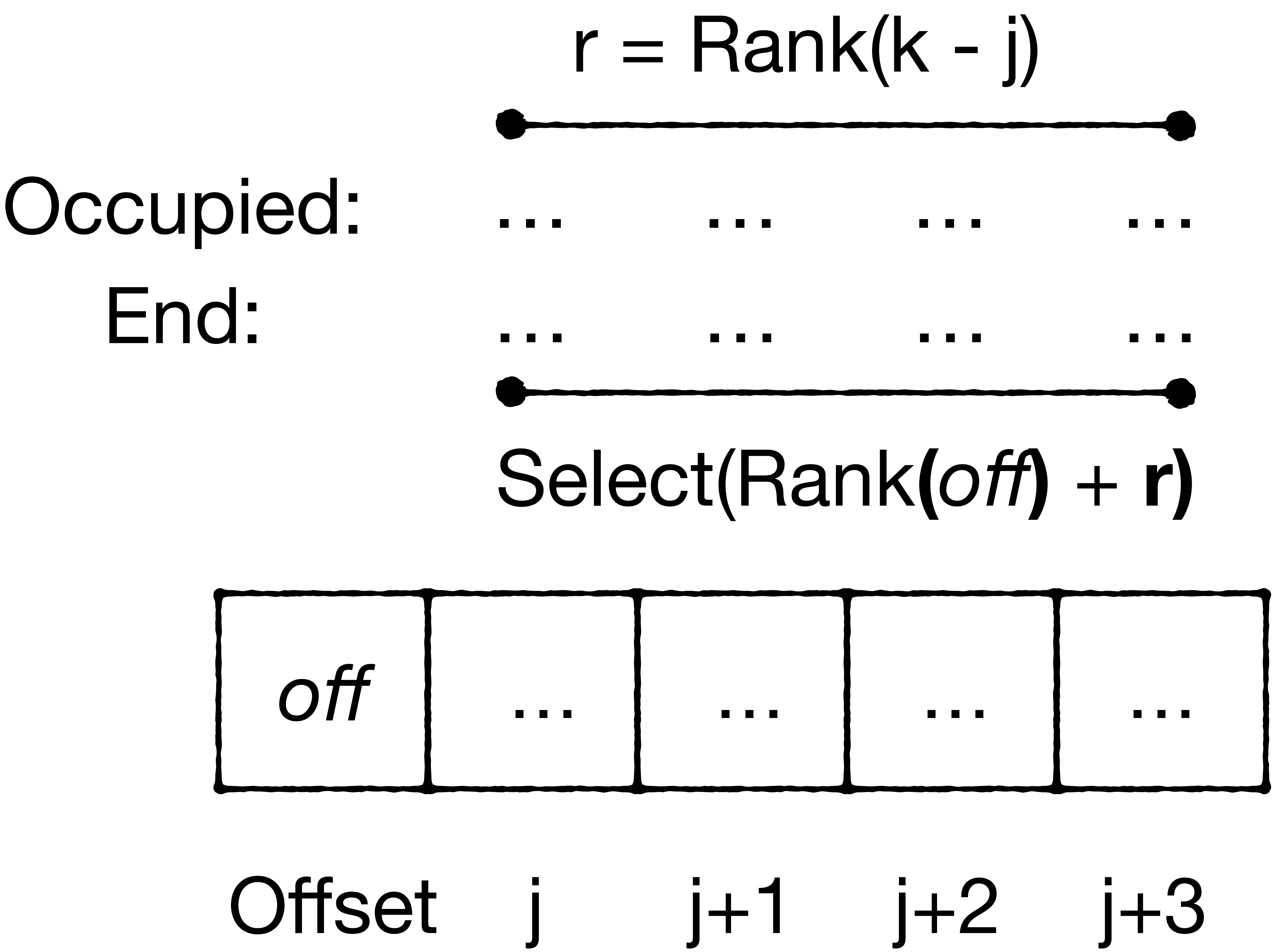
Implementing Select Efficiently

select(i) = tzcnt(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?

off	…	…	…	…
Offset	j	j+1	j+2	j+3

Insertions

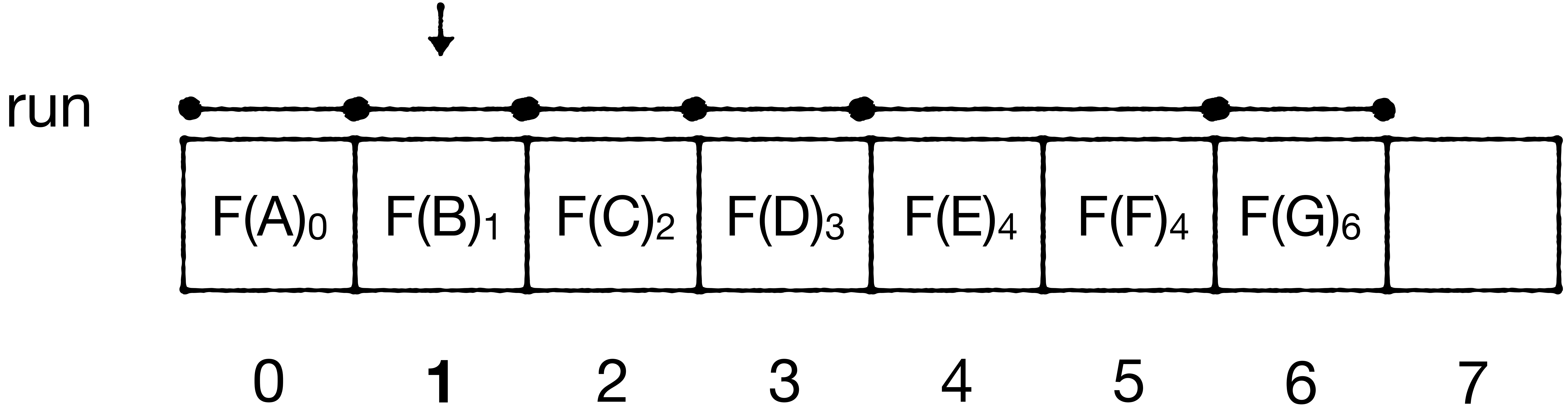
Find target run, push colliding entries to right, insert

off	…	…	…	…
Offset	j	j+1	j+2	j+3

Insertions

Find target run, push colliding entries to right, insert

Insert $F(X)_1$

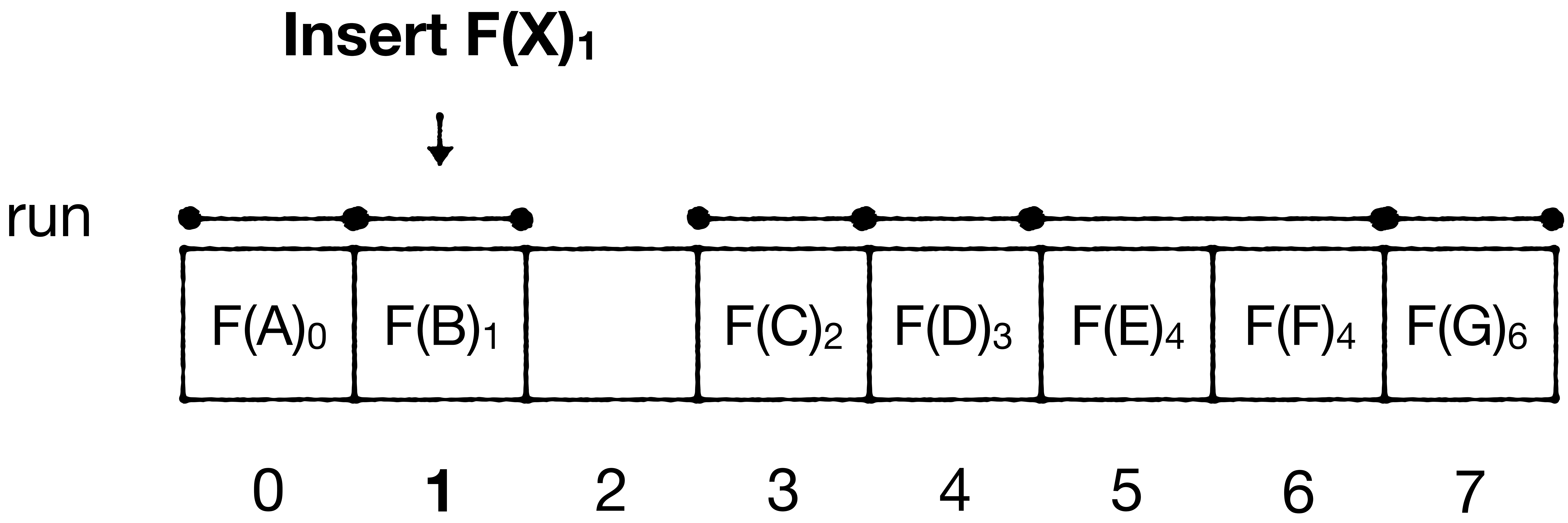


run

$F(A)_0$	$F(B)_1$	$F(C)_2$	$F(D)_3$	$F(E)_4$	$F(F)_4$	$F(G)_6$	
0	**1**	2	3	4	5	6	7

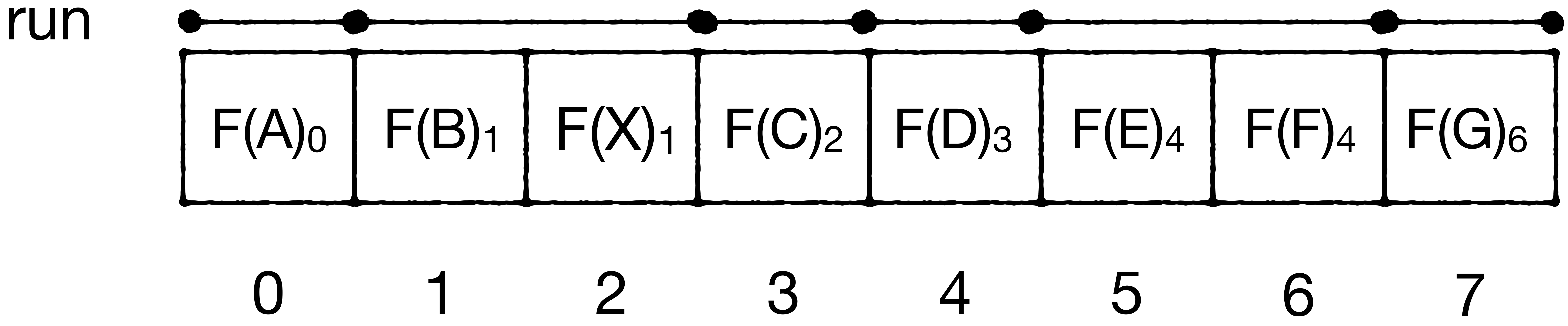
Insertions

Find target run, push colliding entries to right, insert



Insertions

Find target run, push colliding entries to right, insert

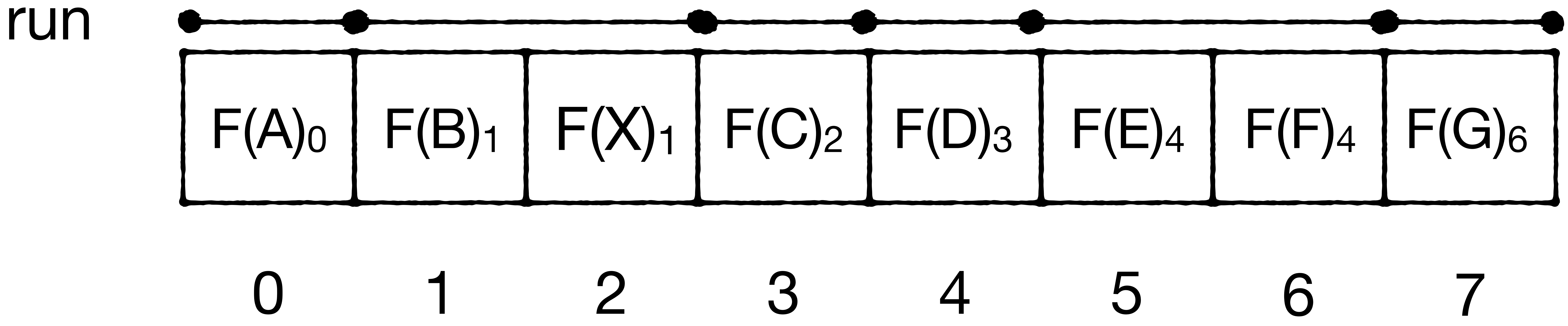


$F(A)_0$	$F(B)_1$	$F(X)_1$	$F(C)_2$	$F(D)_3$	$F(E)_4$	$F(F)_4$	$F(G)_6$
0	1	2	3	4	5	6	7

Insertions

Find target run, push colliding entries to right, insert

Problem?



run

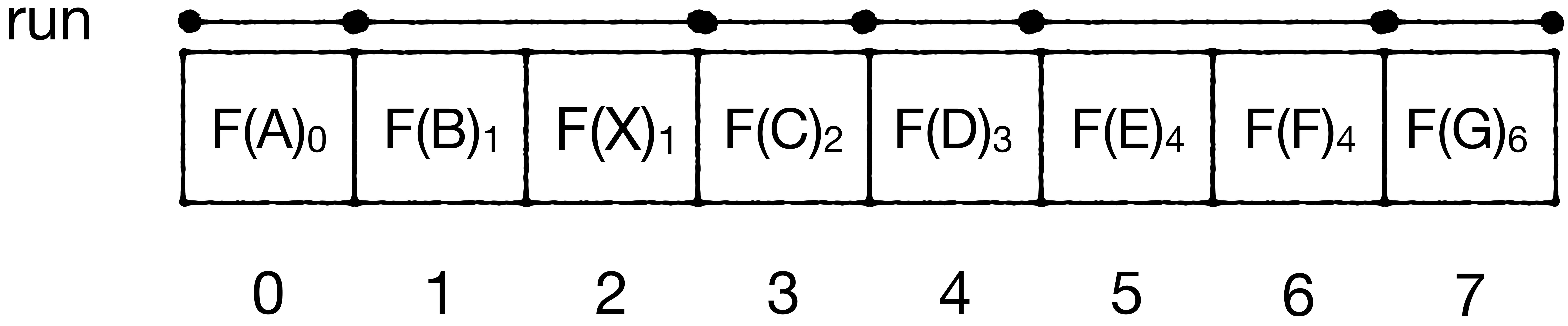
F(A)$_0$	F(B)$_1$	F(X)$_1$	F(C)$_2$	F(D)$_3$	F(E)$_4$	F(F)$_4$	F(G)$_6$
0	1	2	3	4	5	6	7

Insertions

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

↑

potentially O(N)

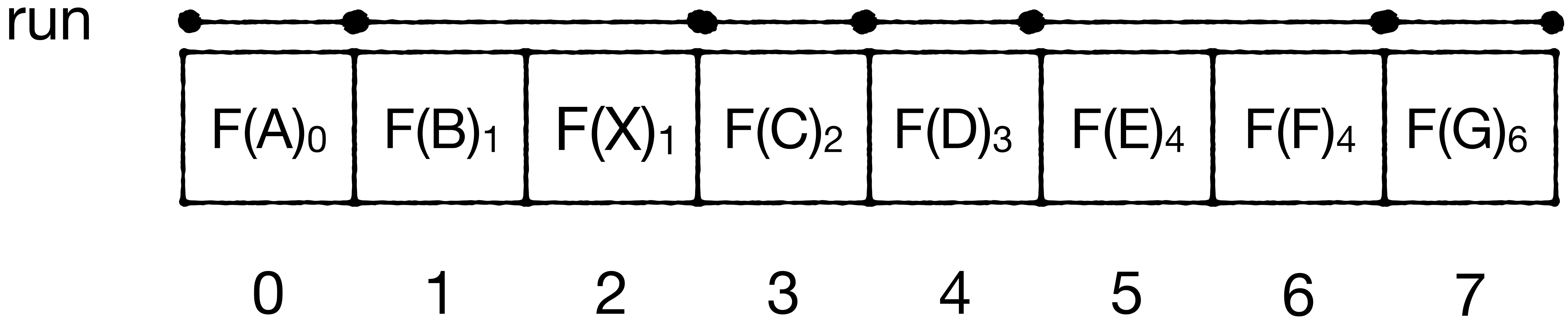


$F(A)_0$	$F(B)_1$	$F(X)_1$	$F(C)_2$	$F(D)_3$	$F(E)_4$	$F(F)_4$	$F(G)_6$
0	1	2	3	4	5	6	7

Insertions

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

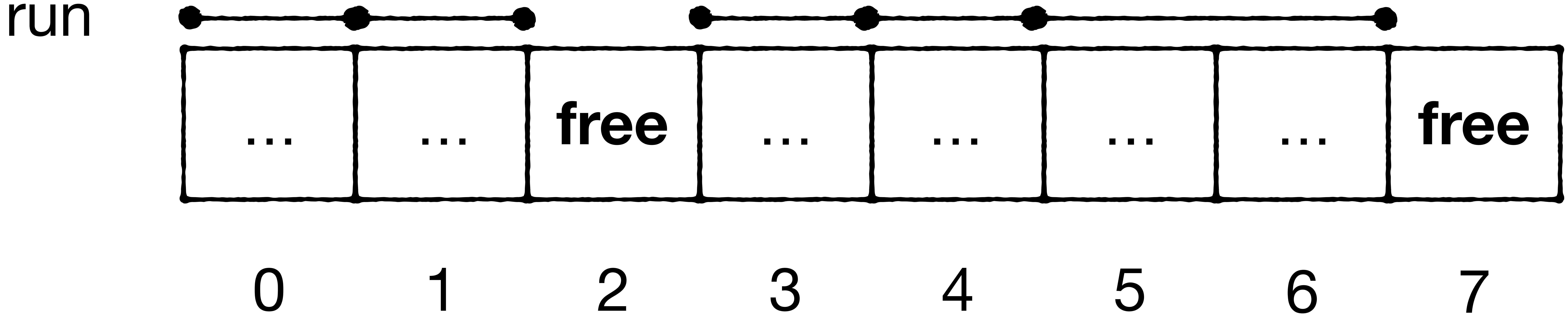
potentially O(N) - **solution**?



run								
	$F(A)_0$	$F(B)_1$	$F(X)_1$	$F(C)_2$	$F(D)_3$	$F(E)_4$	$F(F)_4$	$F(G)_6$
	0	1	2	3	4	5	6	7

Insertions

Keep at least 5% spare capacity



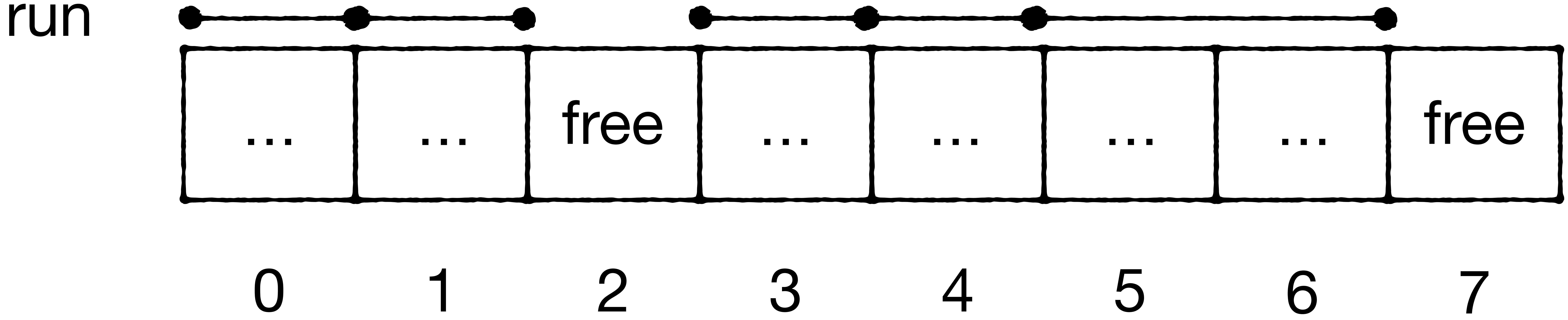
run

…	…	**free**	…	…	…	…	**free**
0	1	2	3	4	5	6	7

Insertions

Keep at least 5% spare capacity

Push on avg. 20 entries on avg due to hashing

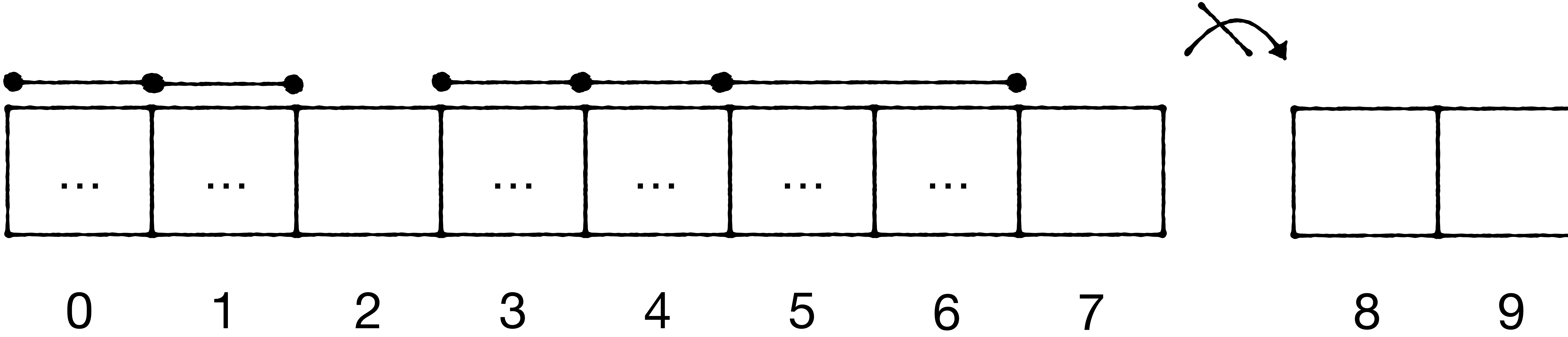


Insertions

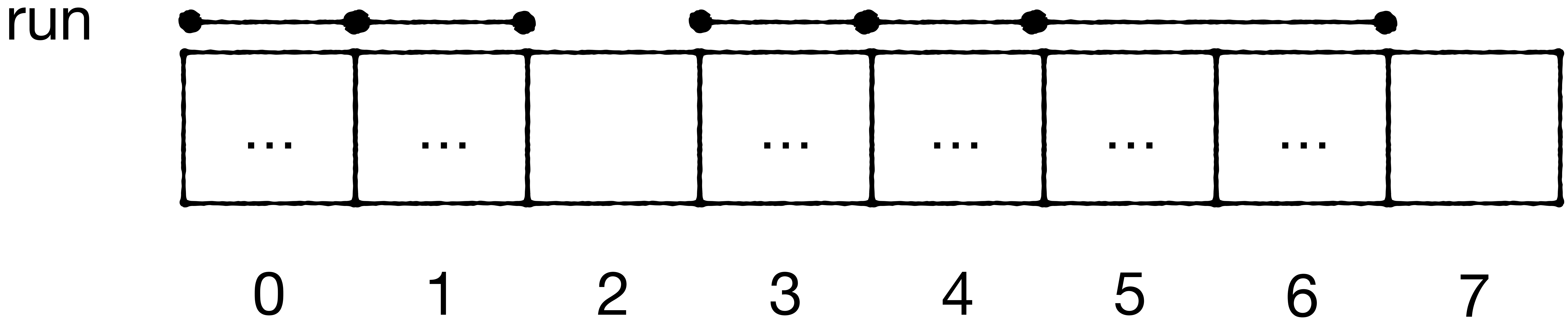
Keep at least 5% spare capacity

Push on avg. 20 entries on avg due to hashing

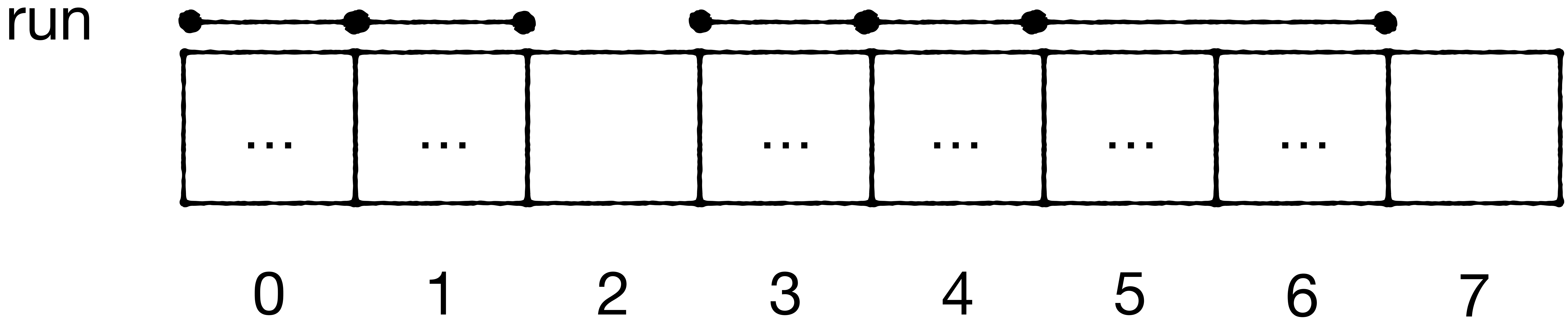
Most insertions don't spill to the next chunk



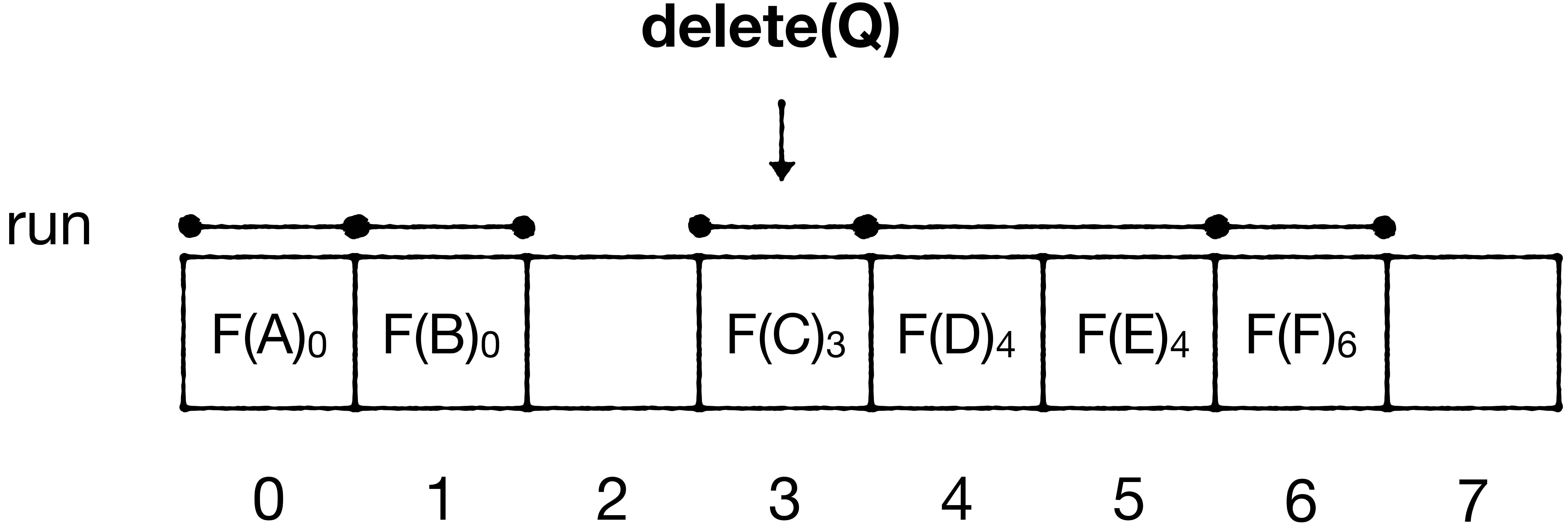
deletes?



Can only delete entry we know exists. Why?



Can only delete entry we know exists. Why?



delete(Q)

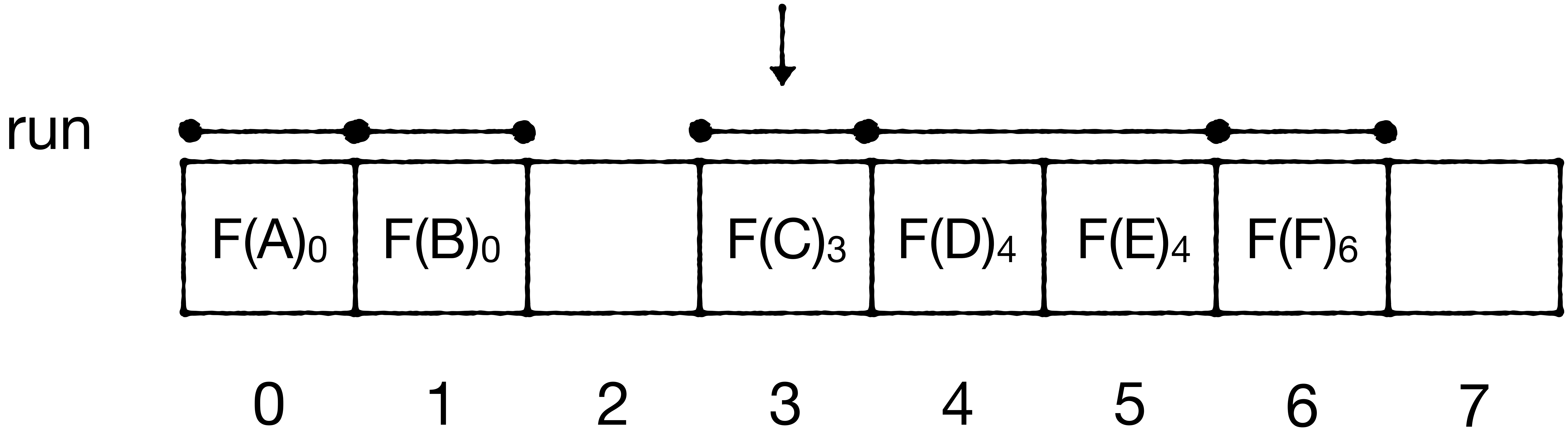
run

$F(A)_0$ $F(B)_0$ $F(C)_3$ $F(D)_4$ $F(E)_4$ $F(F)_6$

0 1 2 3 4 5 6 7

Can only delete entry we know exists. Why?

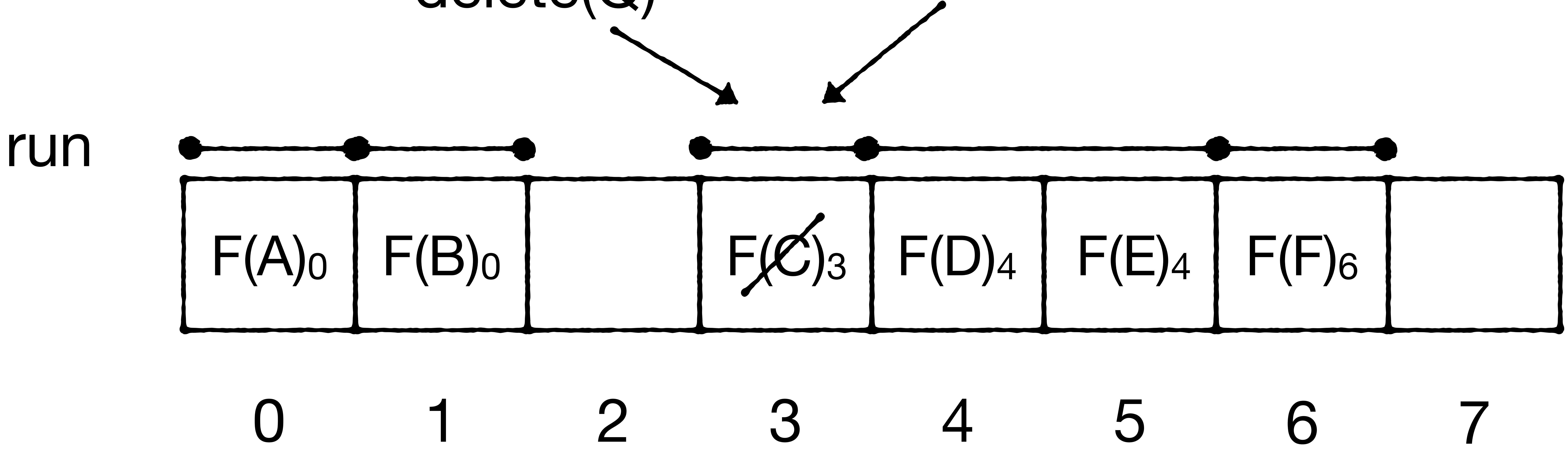
delete(Q) - matches C’s FP at slot 3



Can only delete entry we know exists. Why?

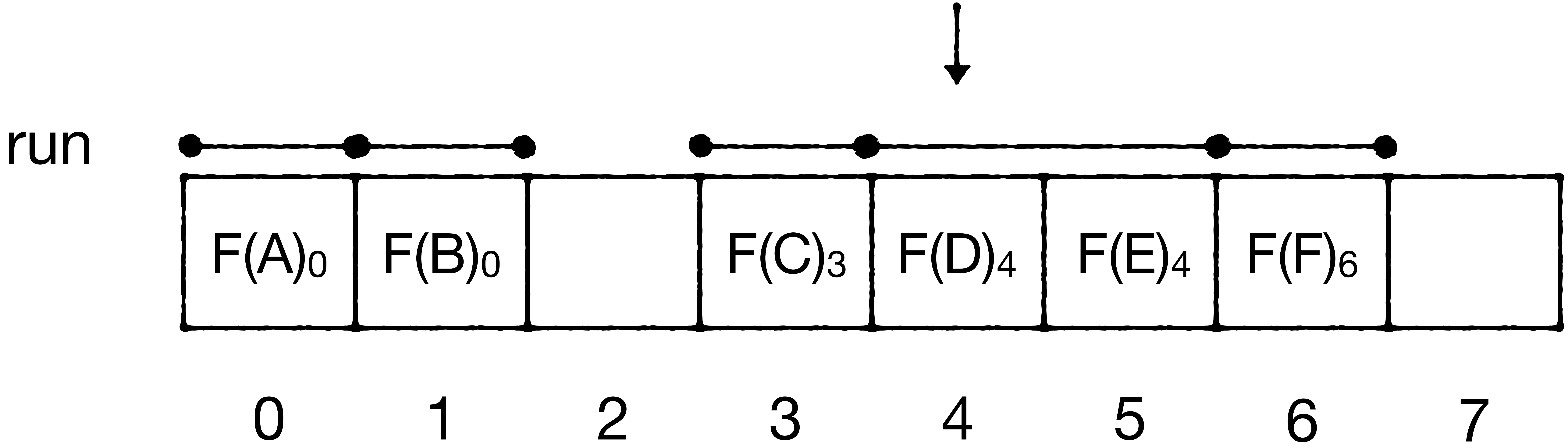
Subsequent get(C) return false negatives

delete(Q)



How to delete an entry we know exists?

delete(D)

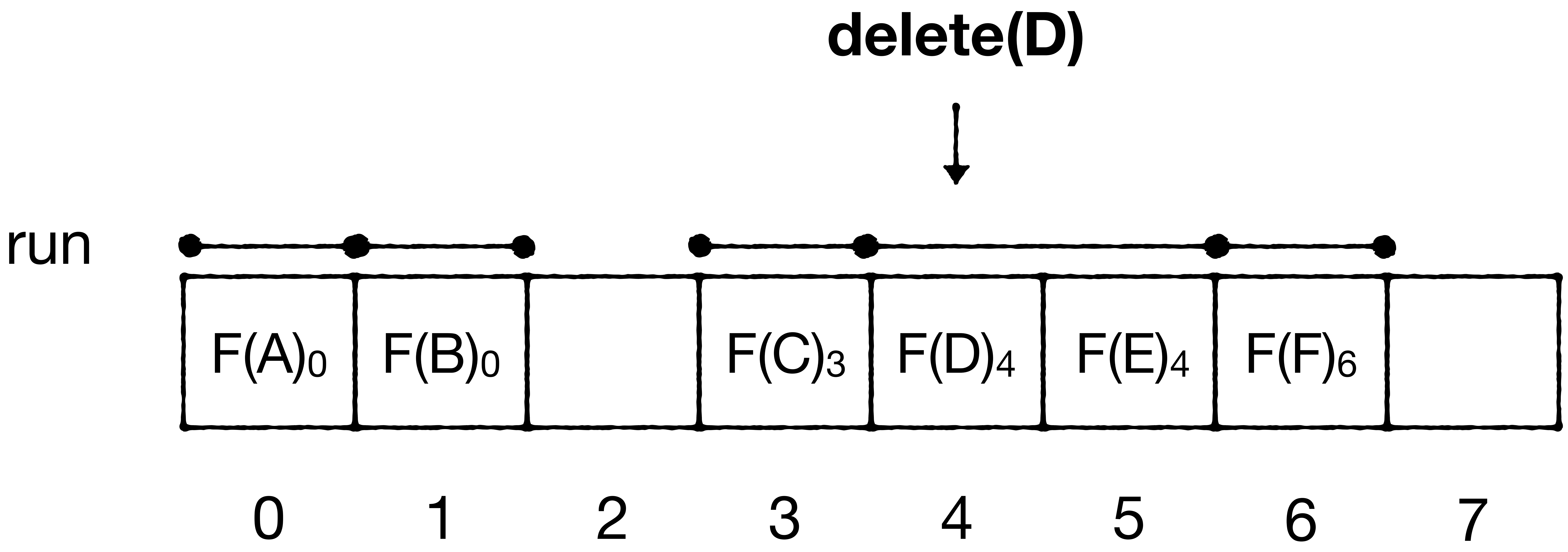


run

$F(A)_0$	$F(B)_0$		$F(C)_3$	$F(D)_4$	$F(E)_4$	$F(F)_6$	
0	1	2	3	4	5	6	7

How to delete an entry we know exists?

(1) Find run, remove matching fingerprint



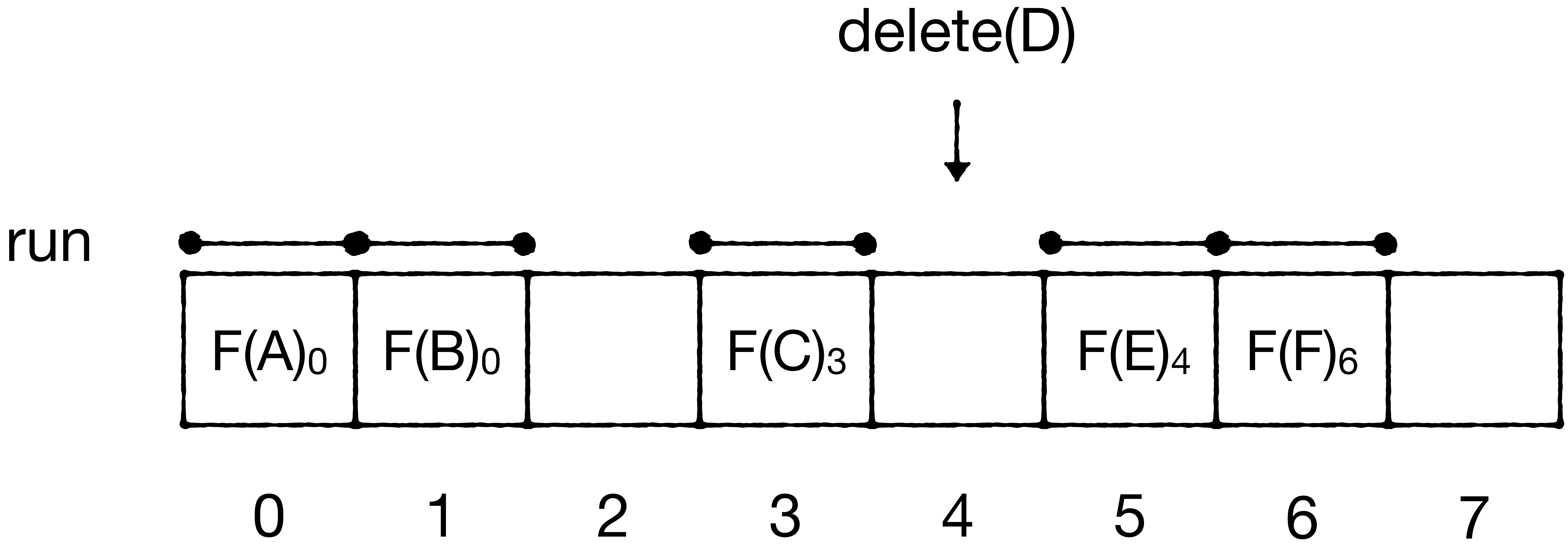
delete(D)

run

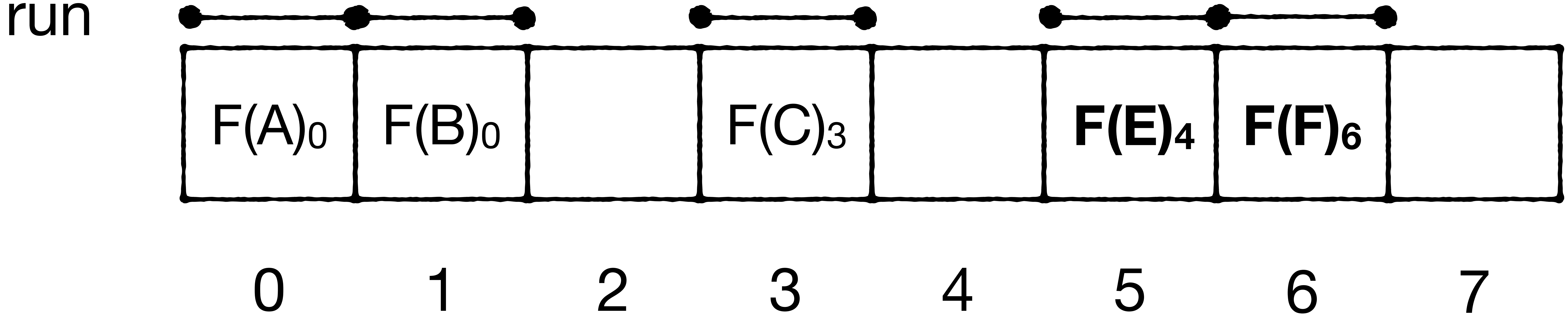
$F(A)_0$	$F(B)_0$		$F(C)_3$	$F(D)_4$	$F(E)_4$	$F(F)_6$	
0	1	2	3	4	5	6	7

How to delete an entry we know exists?

(1) Find run, remove matching fingerprint

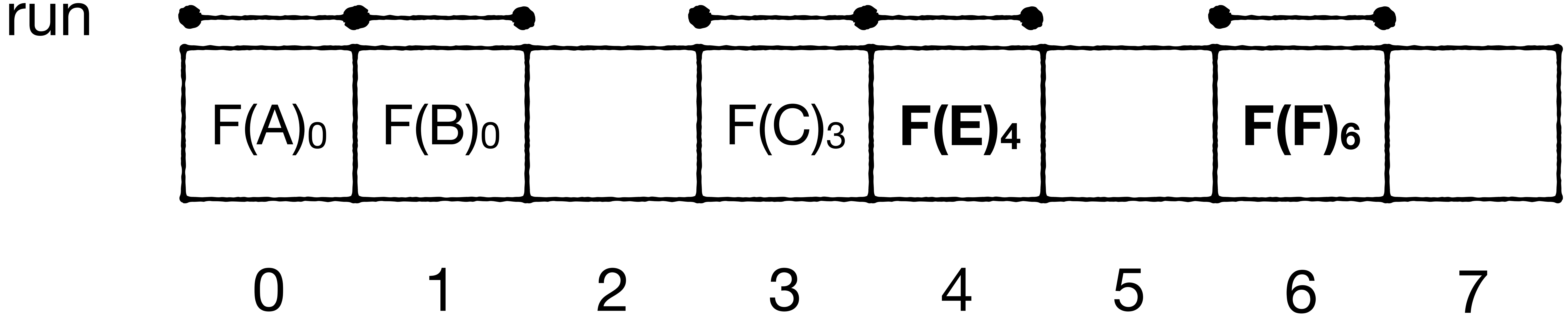


(2) shift entries leftwards if needed to maintain contiguous runs as close as possible to their canonical slot



$F(A)_0$	$F(B)_0$		$F(C)_3$		**$F(E)_4$**	**$F(F)_6$**	
0	1	2	3	4	5	6	7

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



	$F(A)_0$	$F(B)_0$		$F(C)_3$	**$F(E)_4$**		**$F(F)_6$**	
	0	1	2	3	4	5	6	7

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

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

Load factor, α < 0.95

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

$\uparrow$

Avg run length

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

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

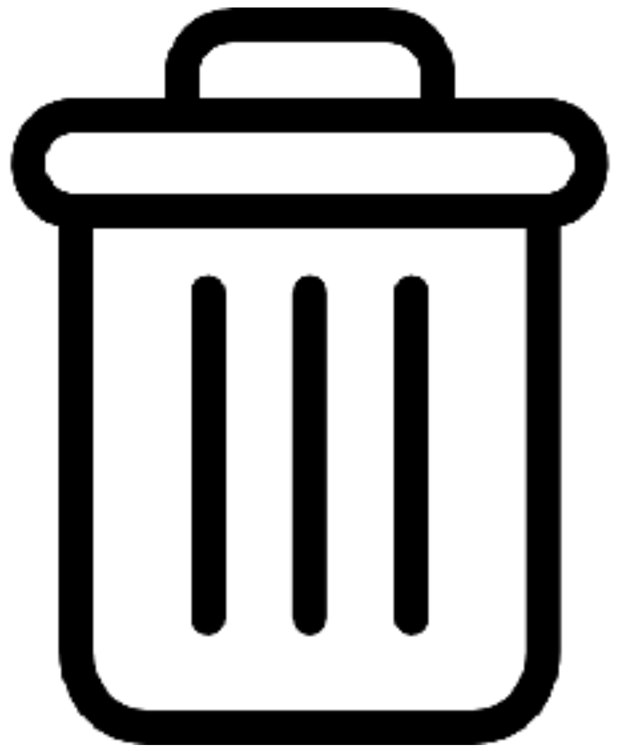
Bloom	Quotient	**XOR**	Idealized
$\approx 2^{-M/N \cdot 0.69}$	$\approx \alpha \cdot 2^{-(M/N - 2.125)/\alpha}$	$\approx 2^{-M/N \cdot 0.81}$	$\approx 2^{-M/N}$

Lower than Bloom for $M/N > 10$

Supports deletes :)

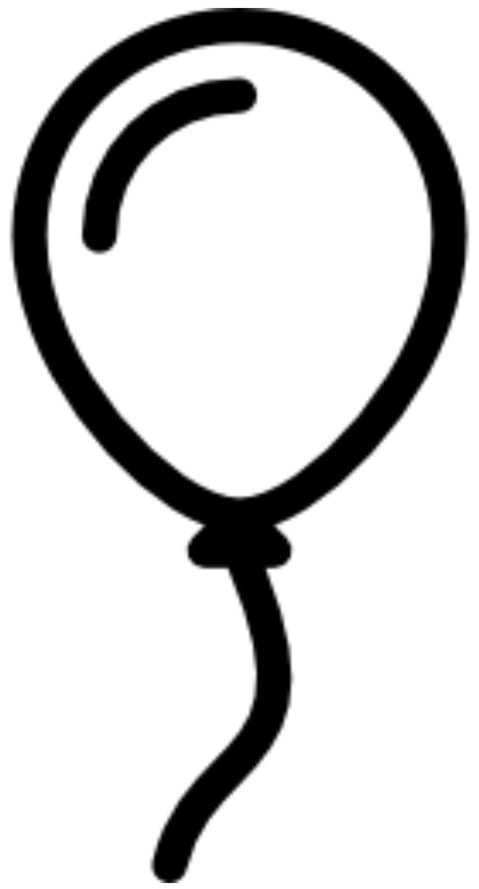
Performance (cache misses)

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

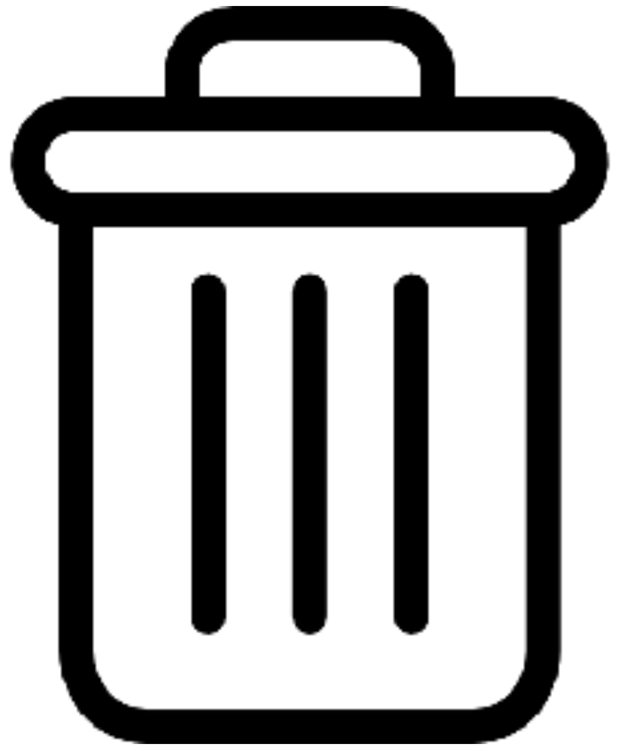


Deletes

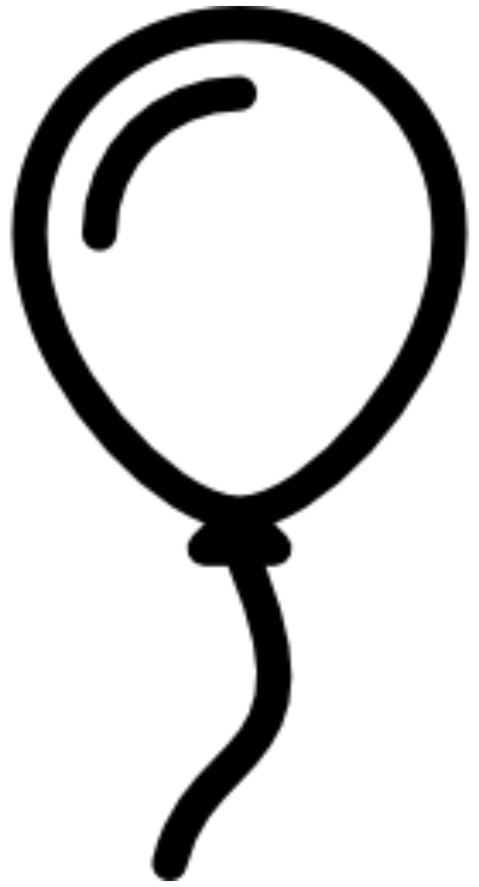
↑



Resizing



Deletes

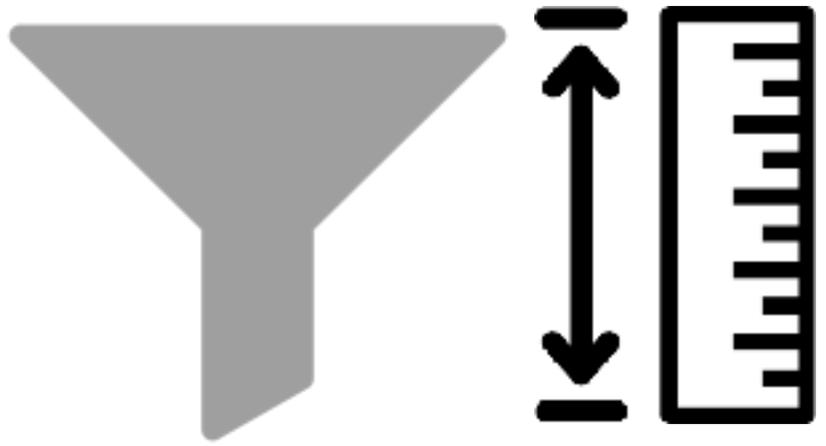


Resizing

↑

Break

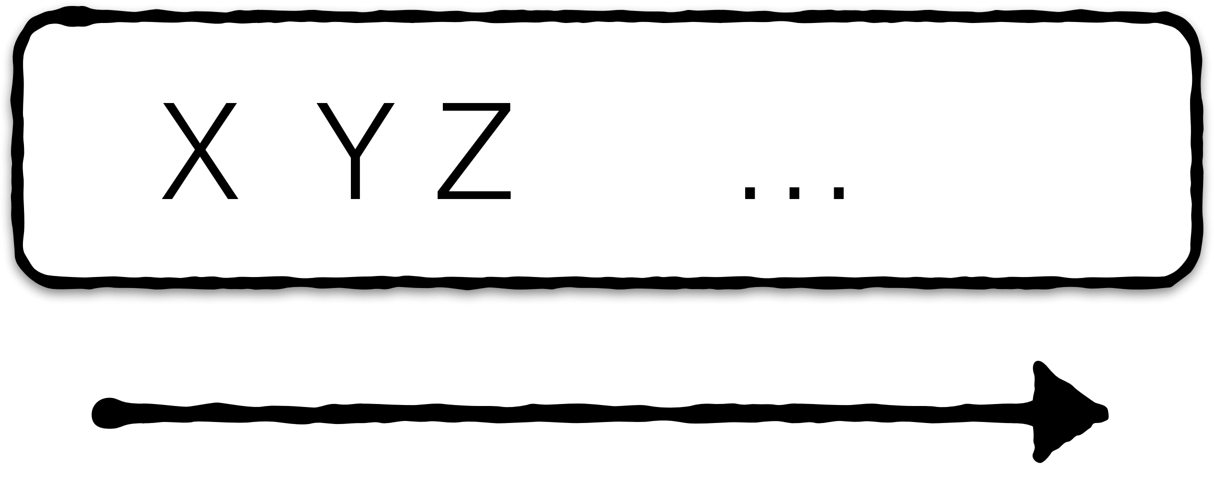
Allocated with fixed capacity



Allocated with fixed capacity

⤓ **False positive rate**

⤓ **Insertion/query/delete cost**

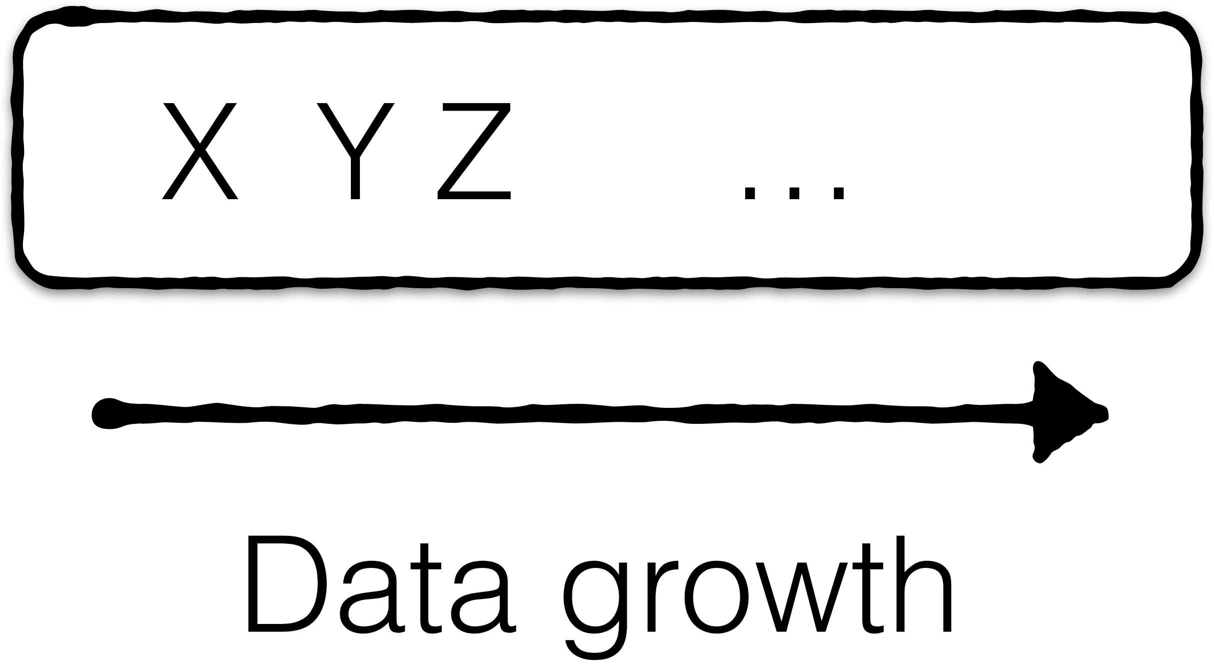


X Y Z ...

Data growth

How to Expand Filters Efficiently?

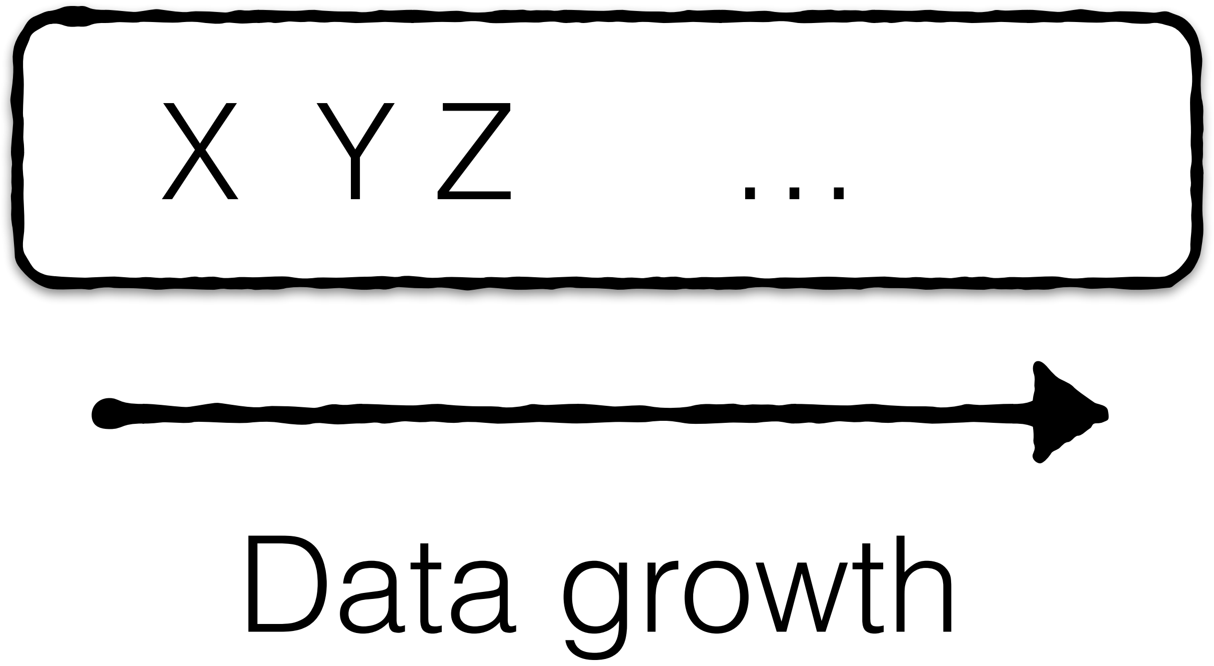
?



X Y Z …

Data growth

How to Expand Filters Efficiently?



Without rereading the original data

Bloom Filters: unexpandable

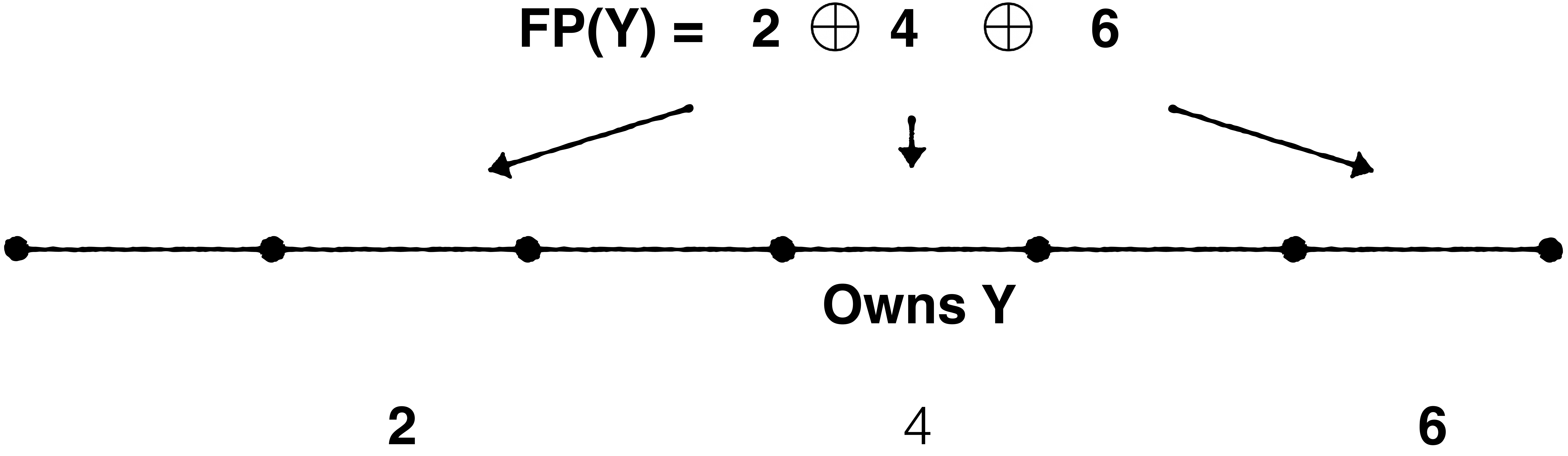
?

0 0 1 0 0 0 0 1 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

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

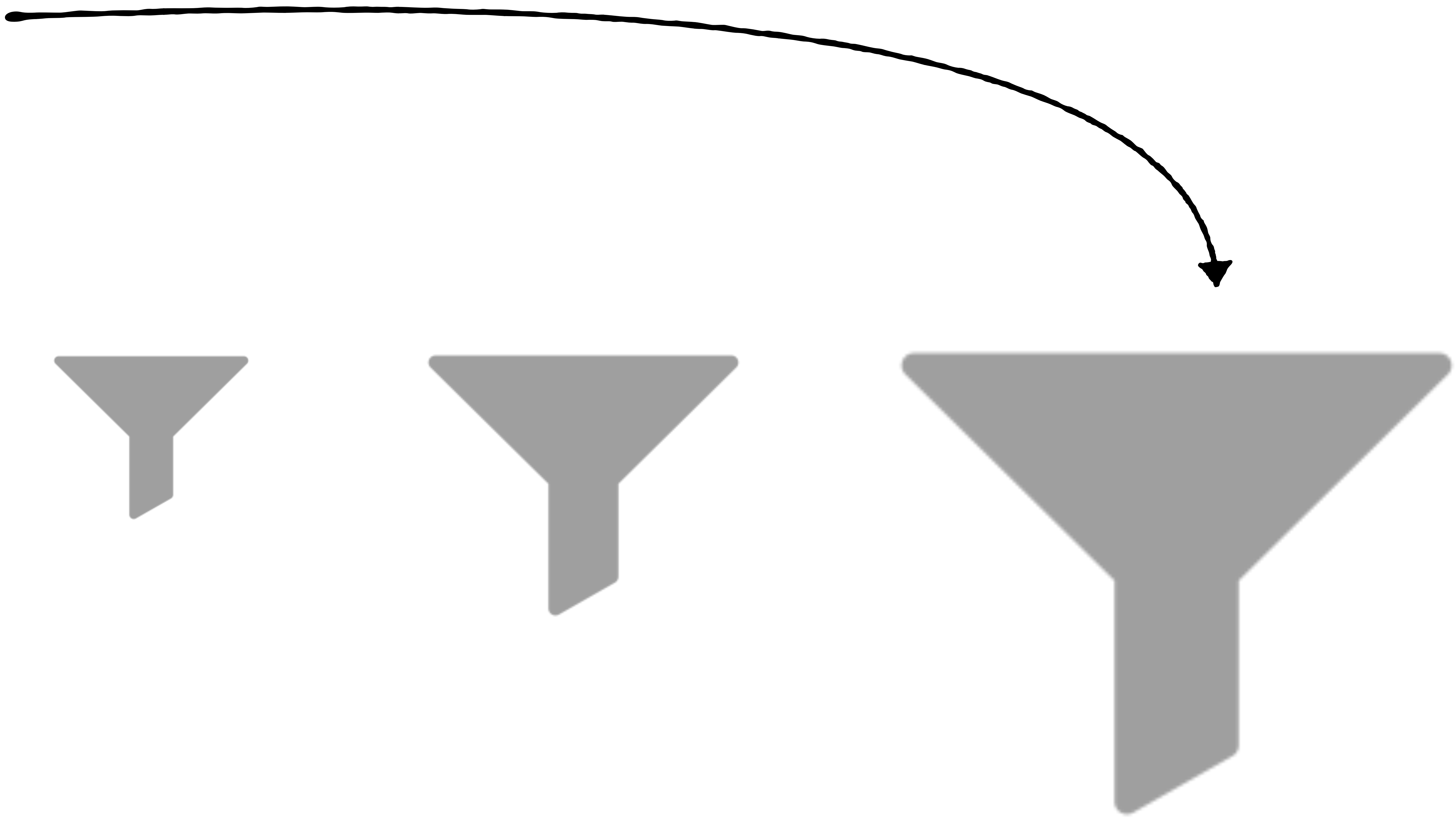
Quotient Filter

InfiniFilter &
Aleph Filter

Chaining

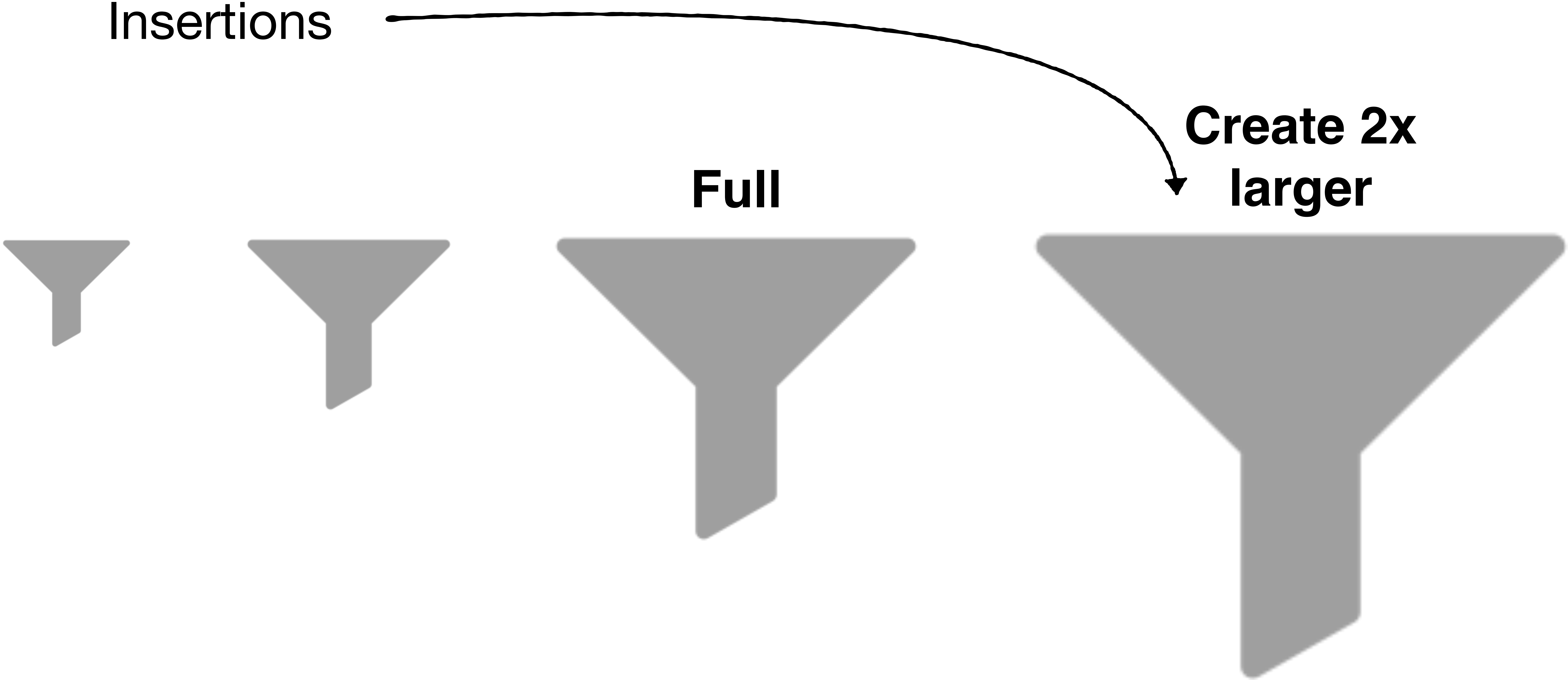
Create 2x larger filter when former reaches capacity

Chaining



Insertions

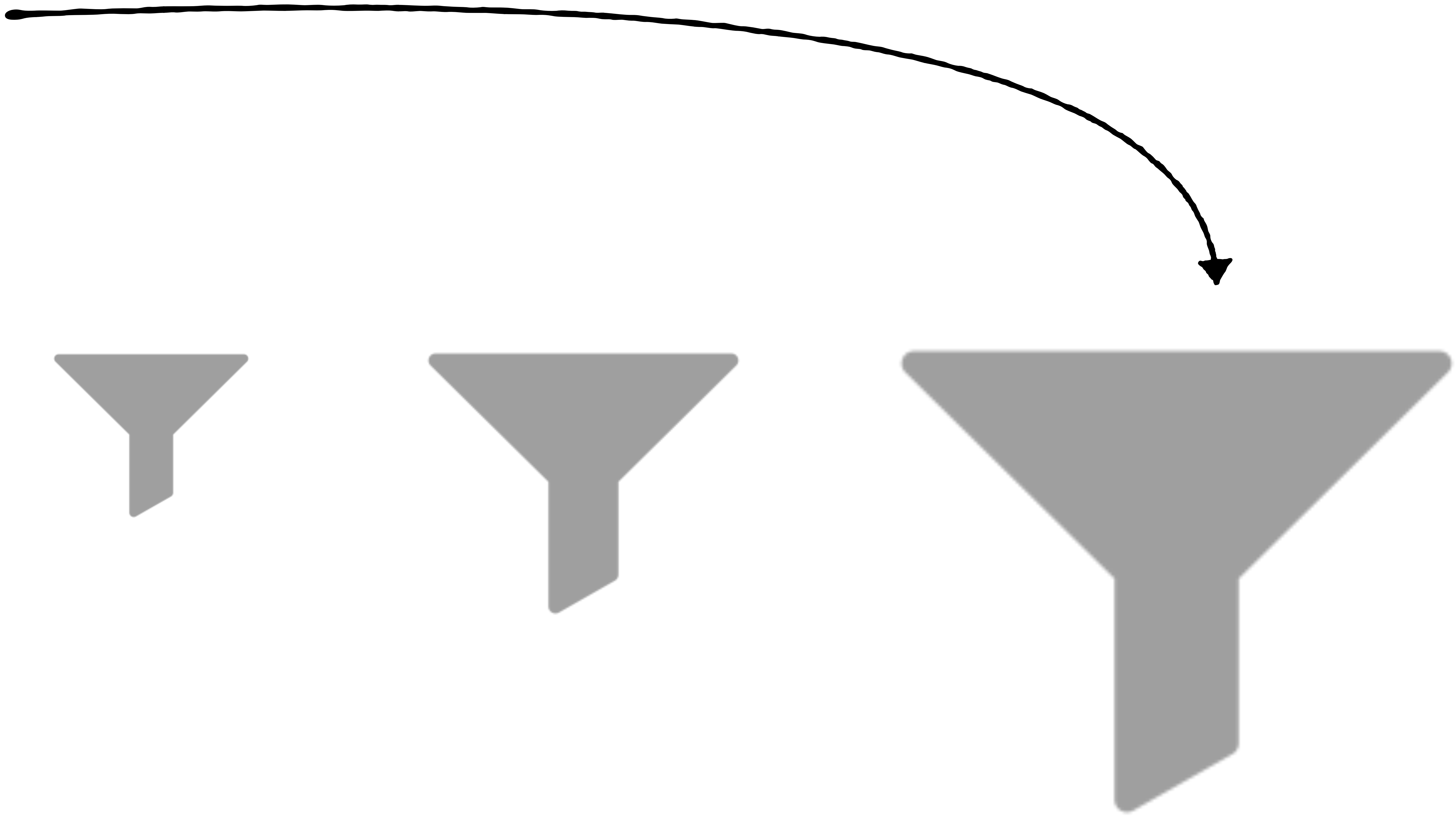
Chaining



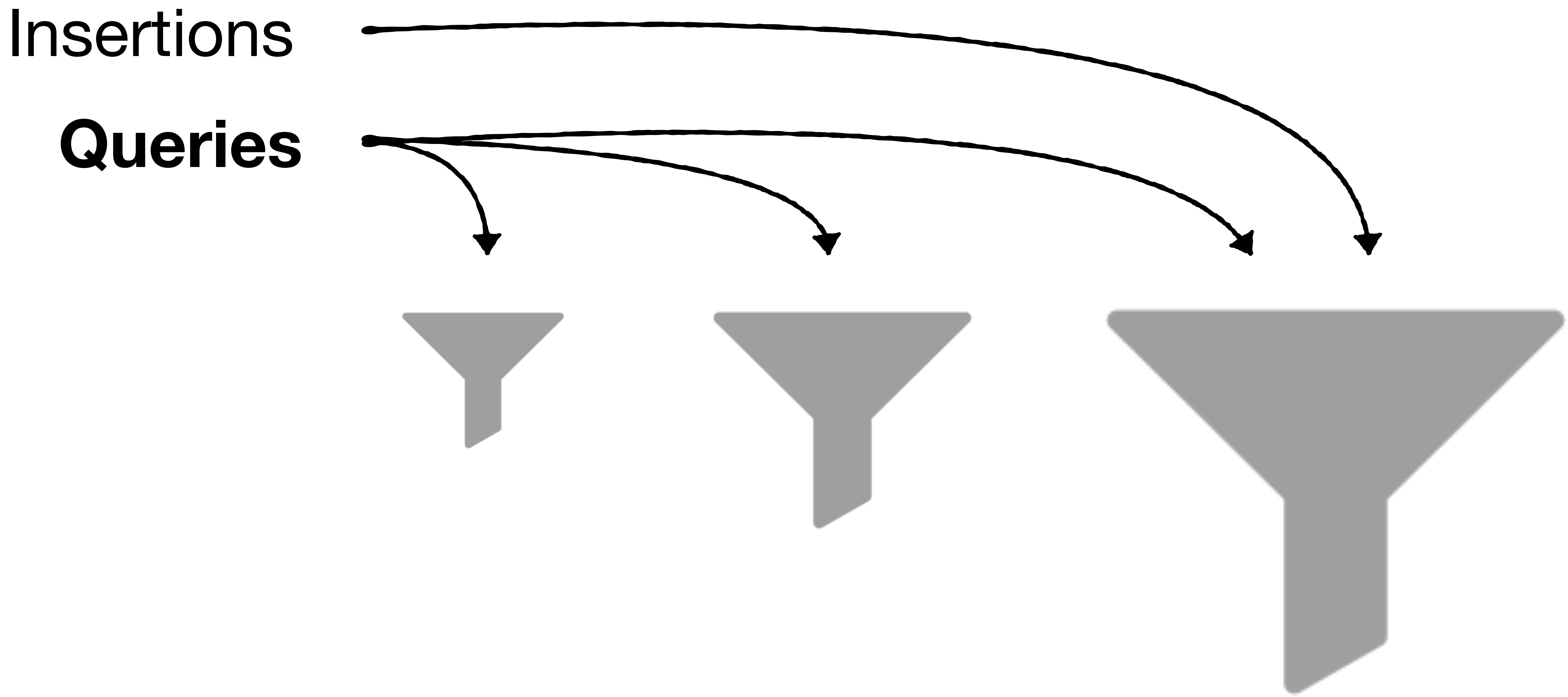
Insertions

Full

Create 2x larger



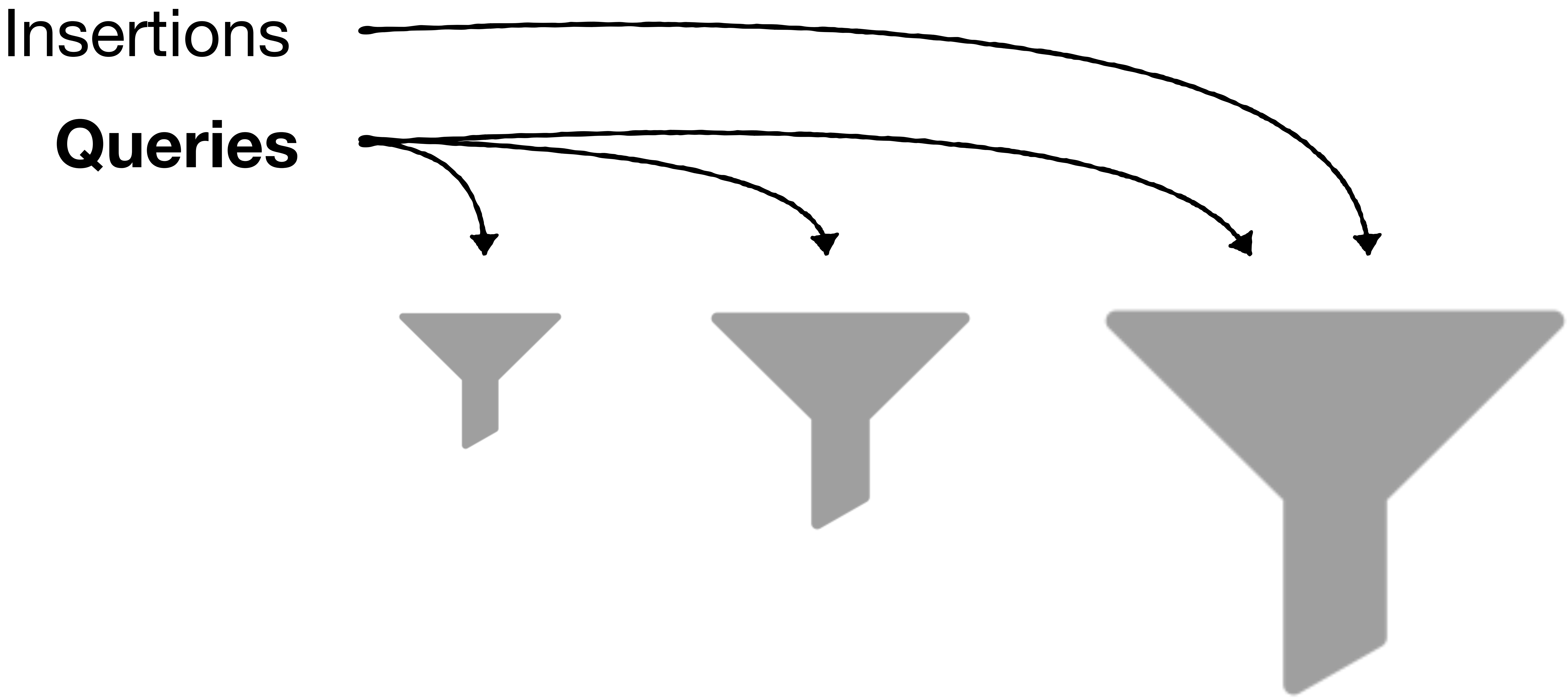
Insertions



Insertions

Queries

Works with any filter

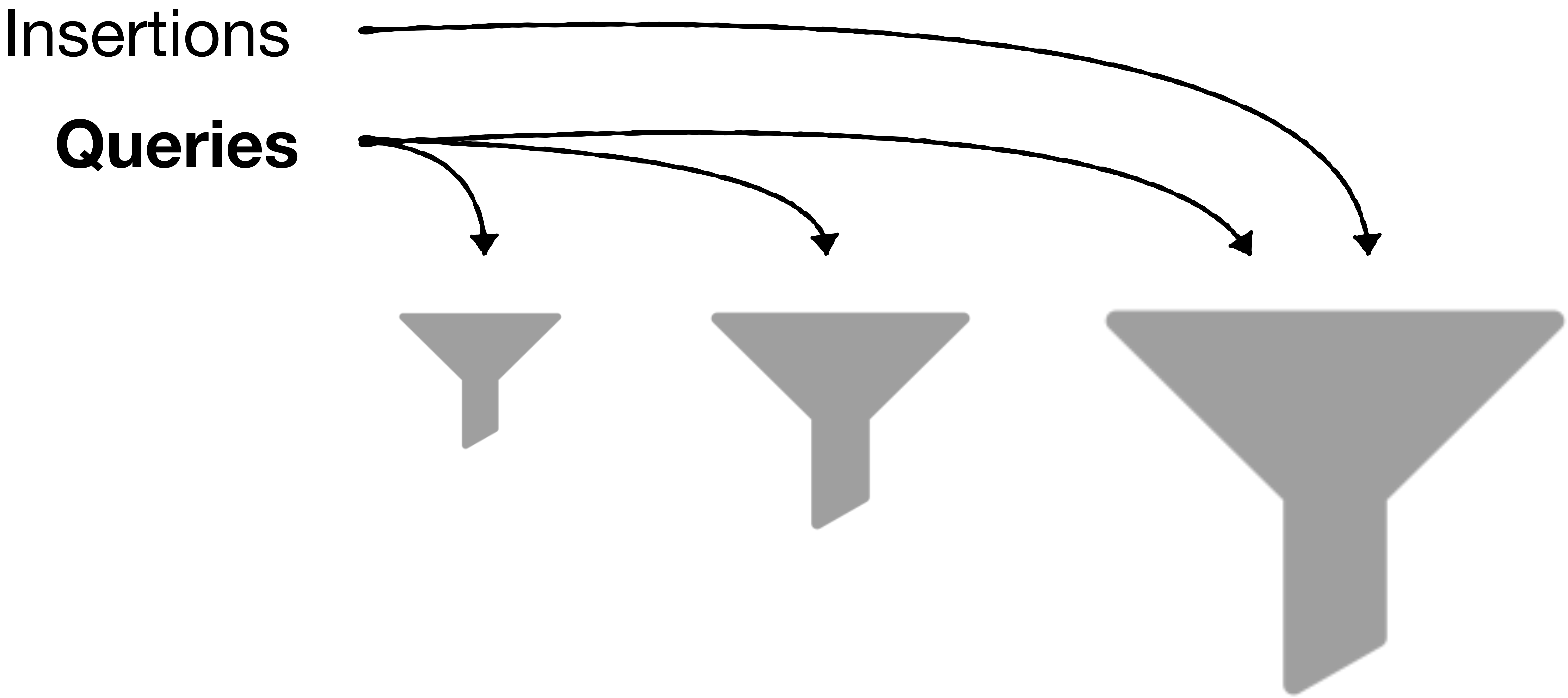


Insertions

Queries

Works with any filter

Downsides?

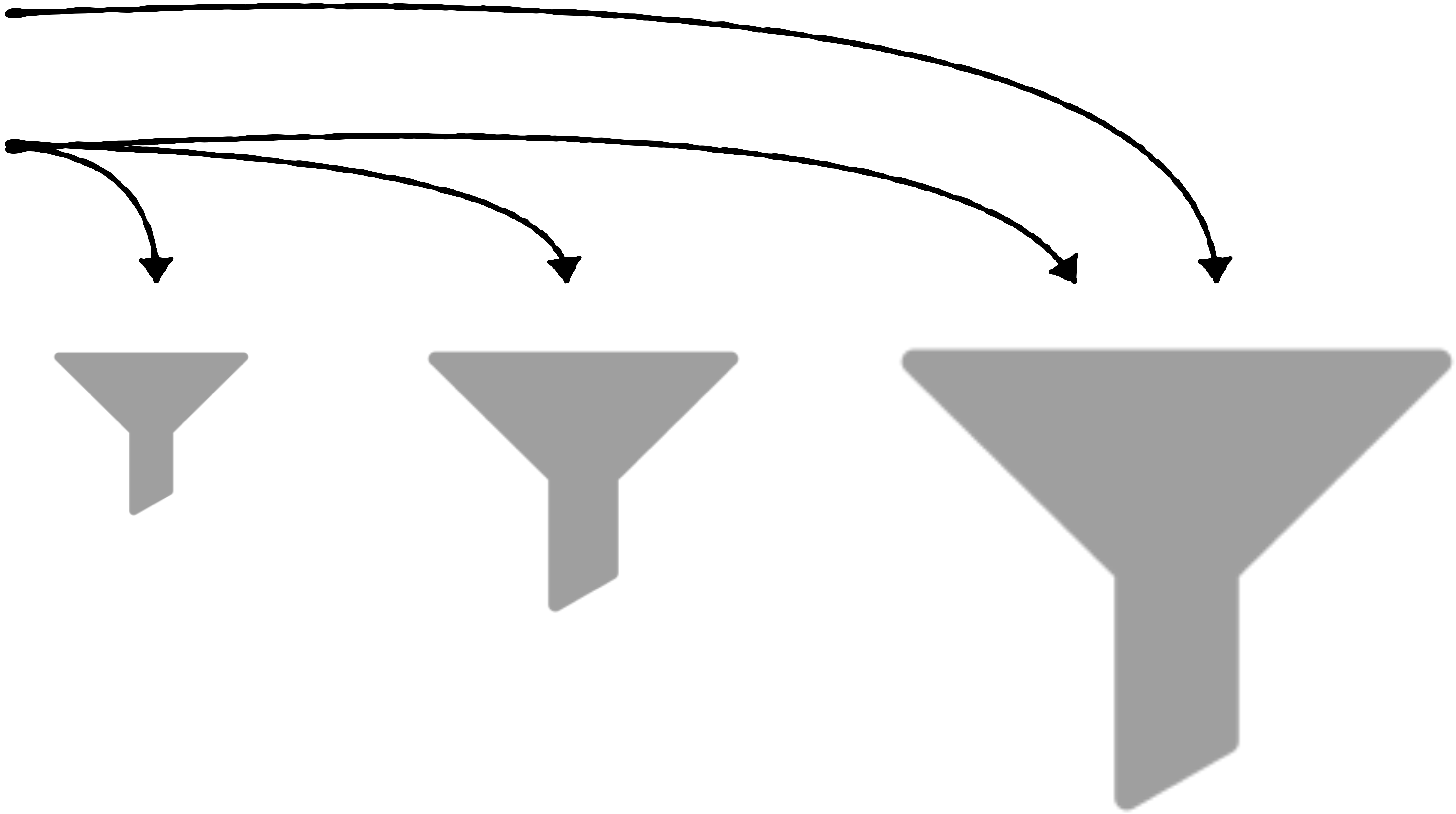


Insertions

Queries

Works with any filter

Downsides?



Insertions

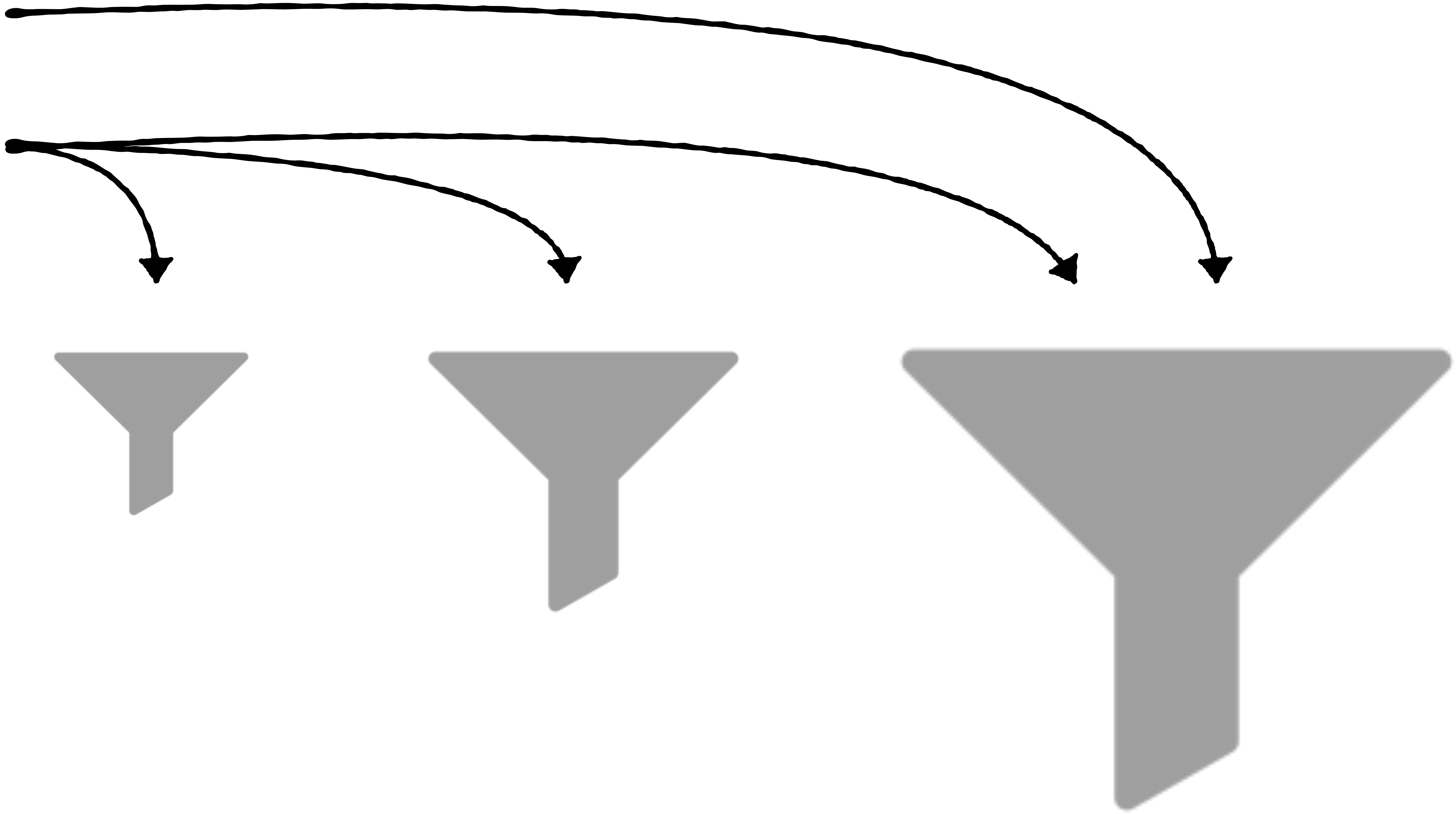
Queries

O($\log_2$ N)

👎

Works with any filter

Downsides?



Insertions

Queries

$O(\log_2 N)$

FPR?

$$\mathbf{FPR} \lesssim \varepsilon + \varepsilon + \varepsilon = O(\varepsilon \cdot \log_2 N)$$

👎

Suppose we want to keep it ε?

$$\text{FPR} \lesssim \varepsilon + \varepsilon + \varepsilon = O(\varepsilon \cdot \log_2 N)$$

Suppose we want to keep it ε?

$$\textbf{FPR} \lesssim \underrightarrow{\boldsymbol{\varepsilon} + \boldsymbol{\varepsilon} + \boldsymbol{\varepsilon}} = O(\boldsymbol{\varepsilon} \cdot \log_2 N)$$

Set lower FPRs for newer filters

Geometrically decreasing. Any issue?

$$\text{FPR} \lesssim \varepsilon + \varepsilon/2 + \varepsilon/4 = O(\varepsilon \cdot \cancel{\log_2} N)$$

$$\text{FPR} \lesssim \varepsilon + \varepsilon/2 + \boldsymbol{\varepsilon/4} = O(\varepsilon)$$

↑

Most Memory

Most data, lowest FPR

$$\text{FPR} \lesssim \varepsilon + \varepsilon/2 + \varepsilon/4 = O(\varepsilon)$$

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

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

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

Can we better scale memory?

$$\text{FPR} \lesssim \varepsilon + \varepsilon/2 + \varepsilon/4 = O(\varepsilon)$$

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

The FPRs should decrease more slowly but still converge

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

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

Reciprocal of square numbers

$$1/\mathbf{1}^2 \quad + \quad 1/\mathbf{2}^2 \quad + \quad 1/\mathbf{3}^2 \quad + \quad \ldots \quad = \quad ?$$

Solved by Euler in 1734

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

Solved by Euler in 1734

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

$$= \quad 1.645$$

Solved by Euler
in 1734

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

Polynomially decreasing yet still convergent

$$\text{FPR} \lesssim \varepsilon/1^2 + \varepsilon/2^2 + \varepsilon/3^2 = \varepsilon \cdot \pi^2/6$$

$\text{FPR} \lesssim \varepsilon/1^2 + \varepsilon/2^2 + \varepsilon/3^2 = \varepsilon \cdot \pi^2/6$

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

$$\text{FPR} \lesssim \varepsilon/1^2 + \varepsilon/2^2 + \varepsilon/3^2 = \varepsilon \cdot \pi^2/6$$

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

$\text{FPR} \lesssim \quad \varepsilon/1^2 \quad + \quad \varepsilon/2^2 \quad + \quad \varepsilon/3^2 \quad = \quad \varepsilon \cdot \pi^2/6$

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

FPR $\lesssim \varepsilon$

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

FPR $\lesssim$ ε

Bits / entry: $\underbrace{2 \ \log_2 \log_2 (N) + \log(1/\varepsilon)}_{\textbf{Close to lower bound}}$

How to Approximate A Set Without Knowing Its Size In Advance

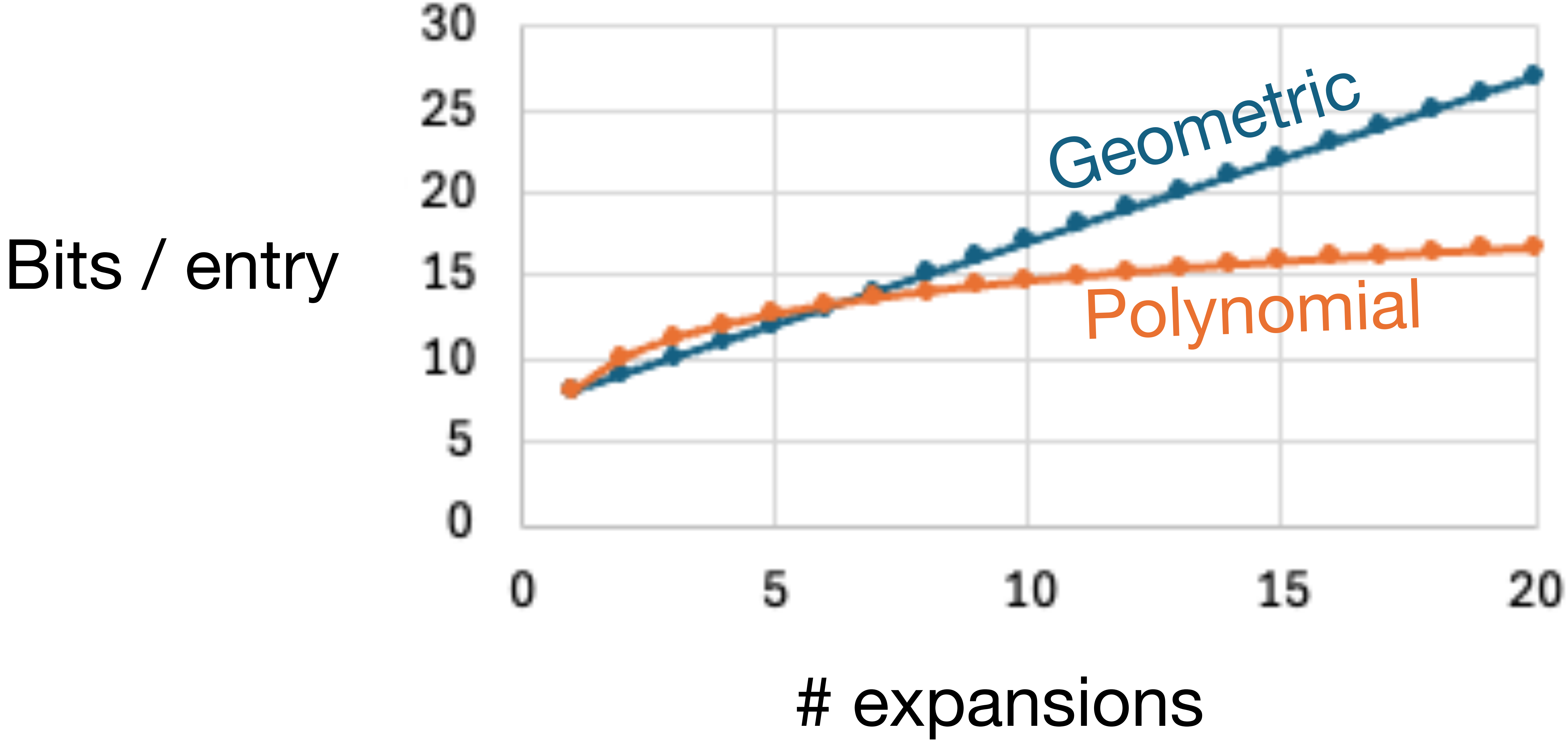
Rasmus Pagh, Gil Segev, Udi Wieder. **FOCS 2013.**

FPR $\lesssim$ ε

Bits / entry: $\underbrace{2\ \log_2 \log_2 (N) + \log(1/\varepsilon)}_{\text{Close to lower bound}}$ 👌

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
30
25
20
15
10
5
0
0
5
10
15
20
expansions
Geometric
Polynomial

Chaining

👎 queries

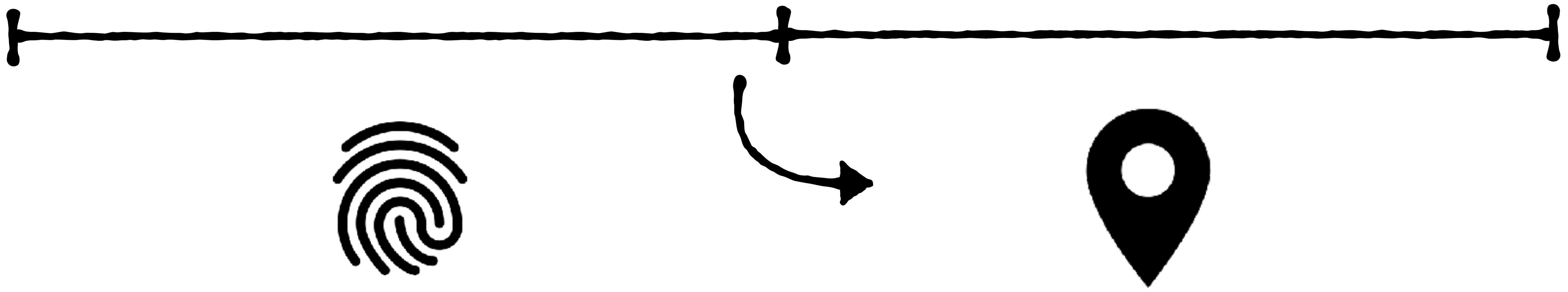
Quotient Filters

InfiniFilter &
Aleph Filter

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

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

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

Lose 1 fingerprint bit in each expansion

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

Remove constants

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

Simplify

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

Supports up to *M/N* expansions 👎

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

Supports up to *M/N* expansions 👎

O(1) operations 👍

Chaining

👎 queries

Quotient Filters

👎 FPR

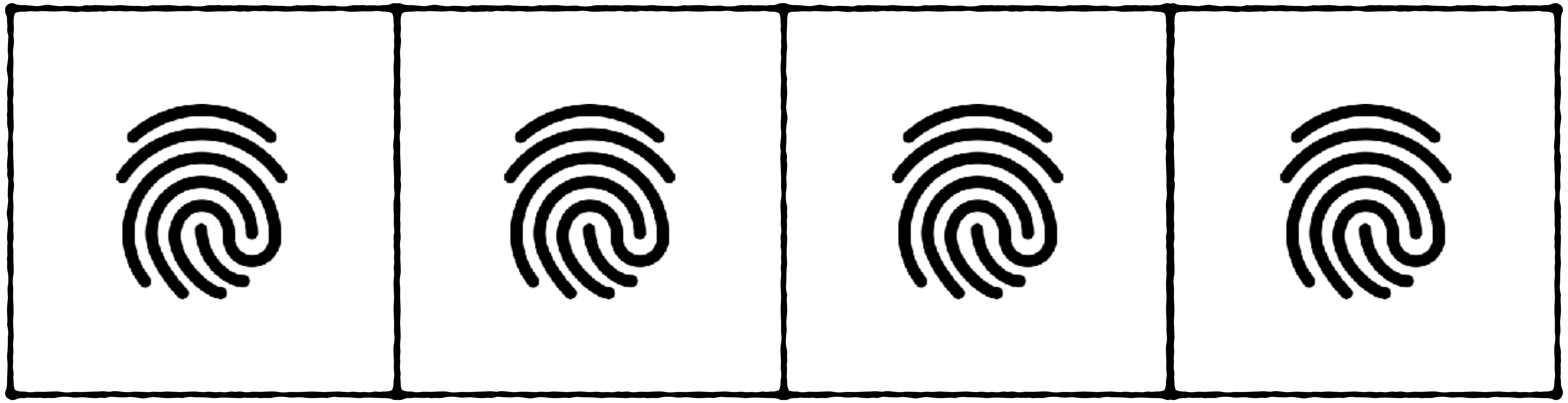
\# expansions

InfiniFilter &
Aleph Filter

InfiniFilter: Expanding Filters to Infinity and Beyond

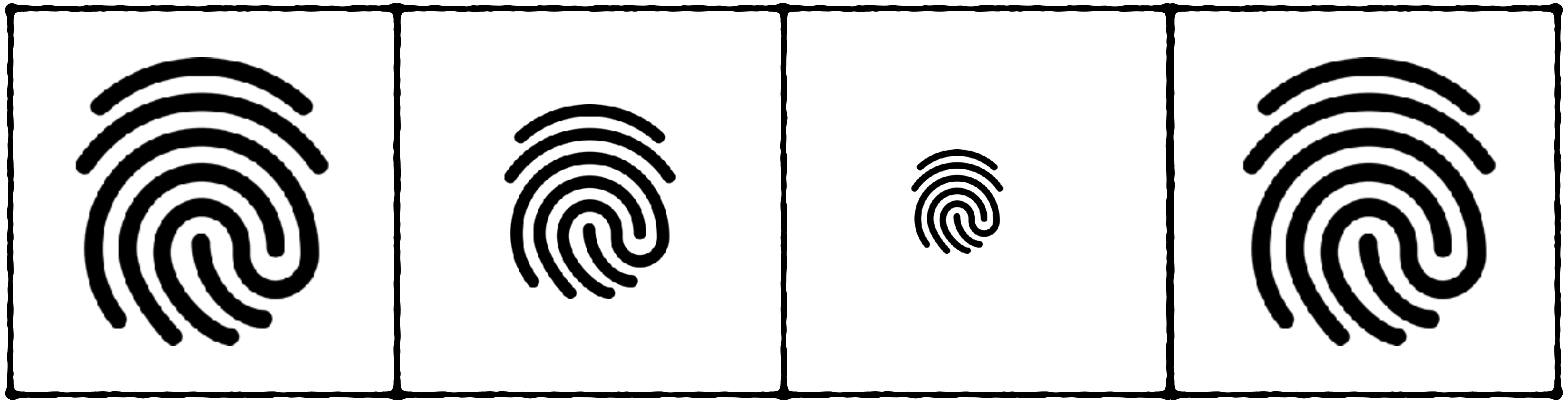
Niv Dayan, Ioana Bercea, Pedro Reviriego, Rasmus Pagh. SIGMOD 2023

InfiniFilter



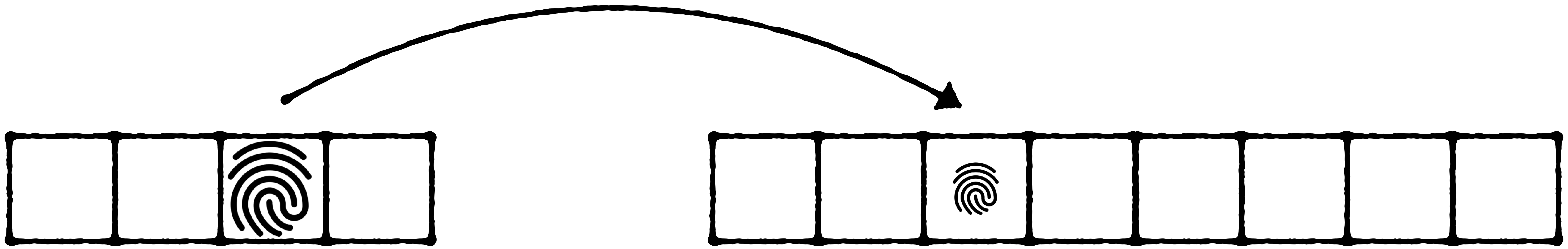
Quotient filter

InfiniFilter



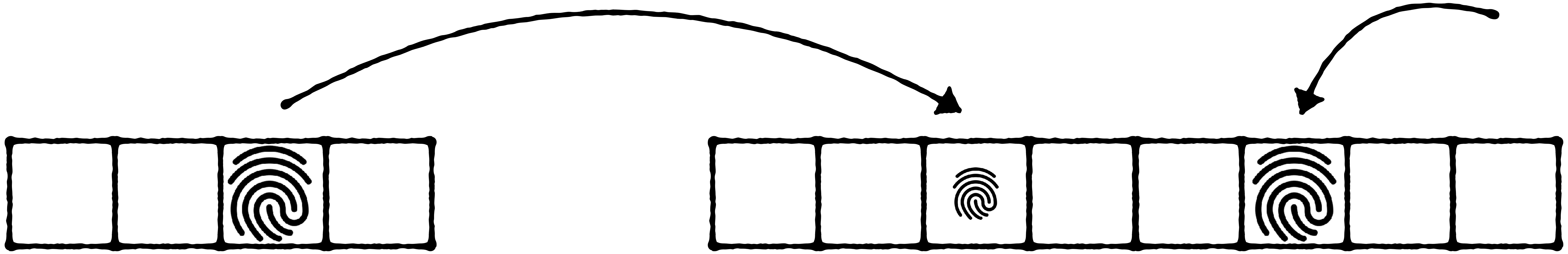
Variable-sized fingerprints

InfiniFilter



(1) sacrifice one bit during expansion

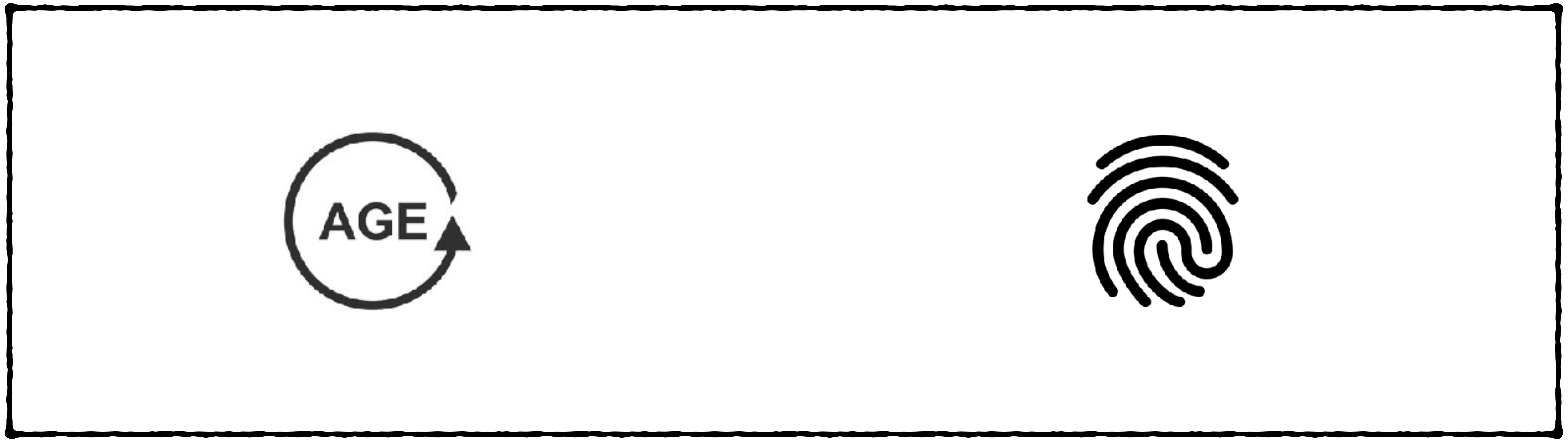
InfiniFilter



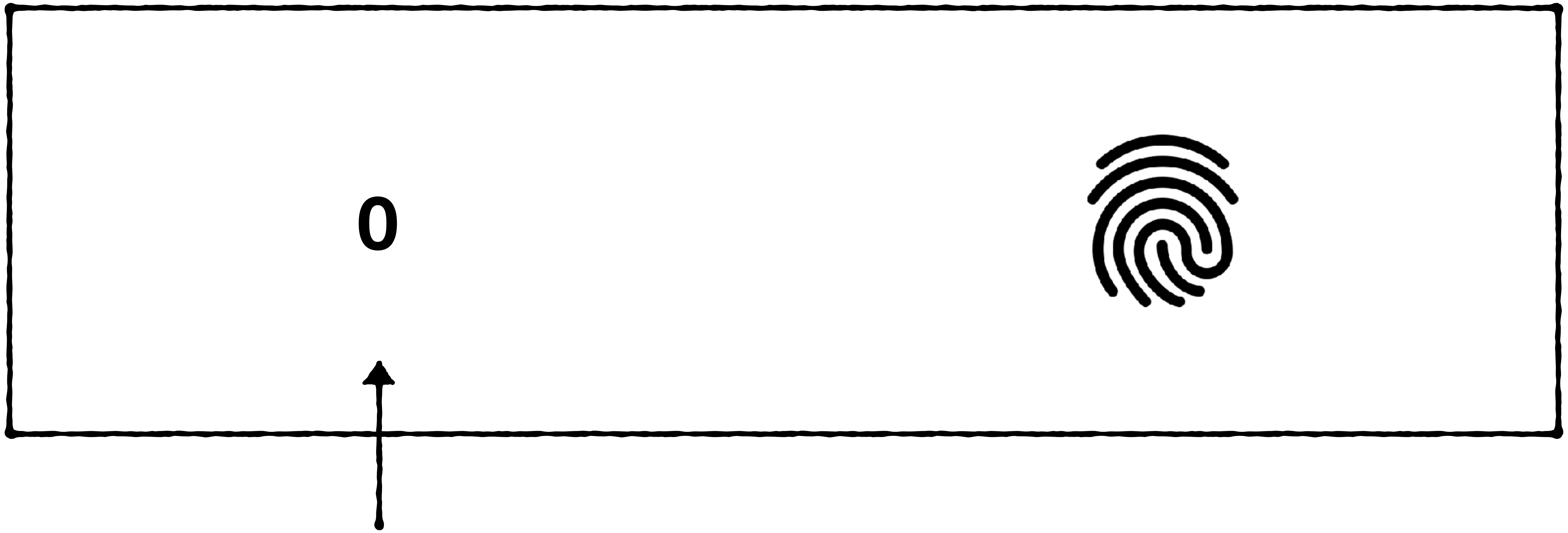
(1) sacrifice one bit
during expansion

(2) Newer entries get
longer fingerprints

Unary age counter Fingerprint

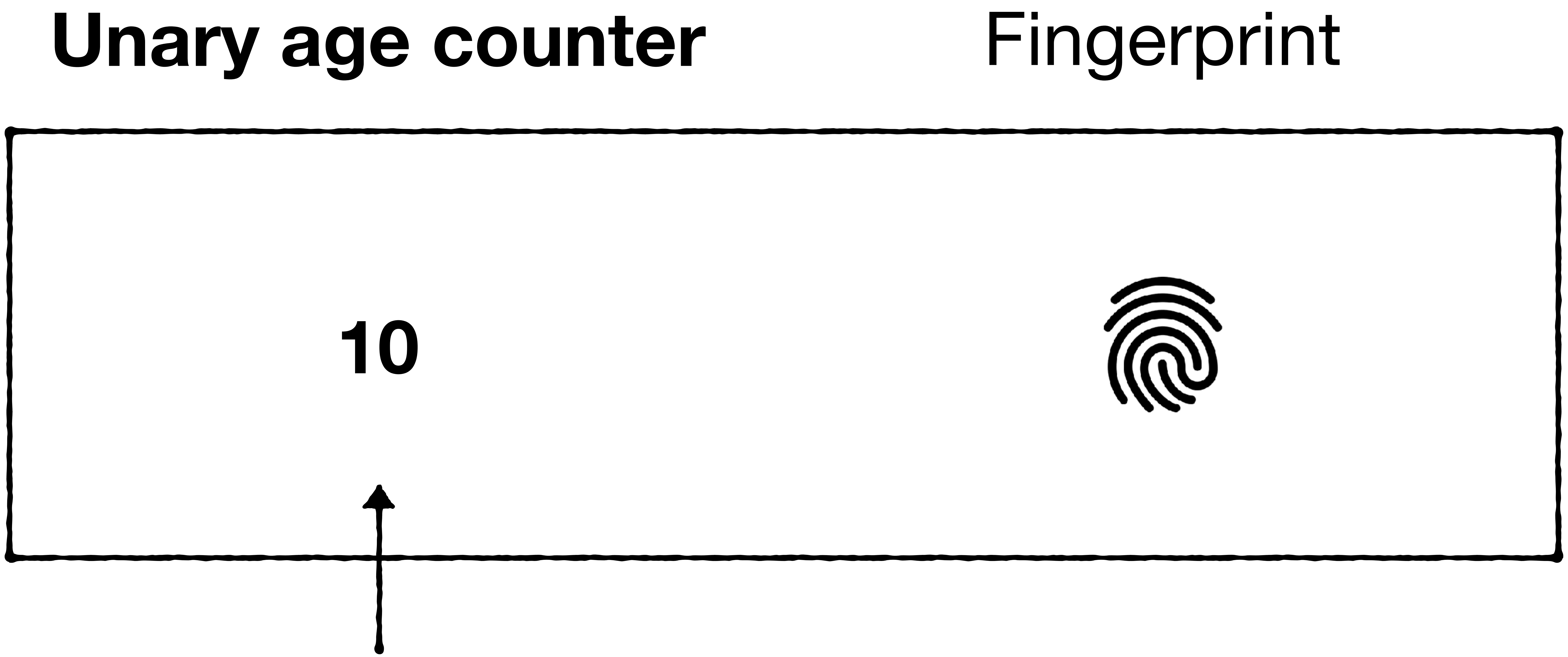


Unary age counter Fingerprint



0

0 expansions ago

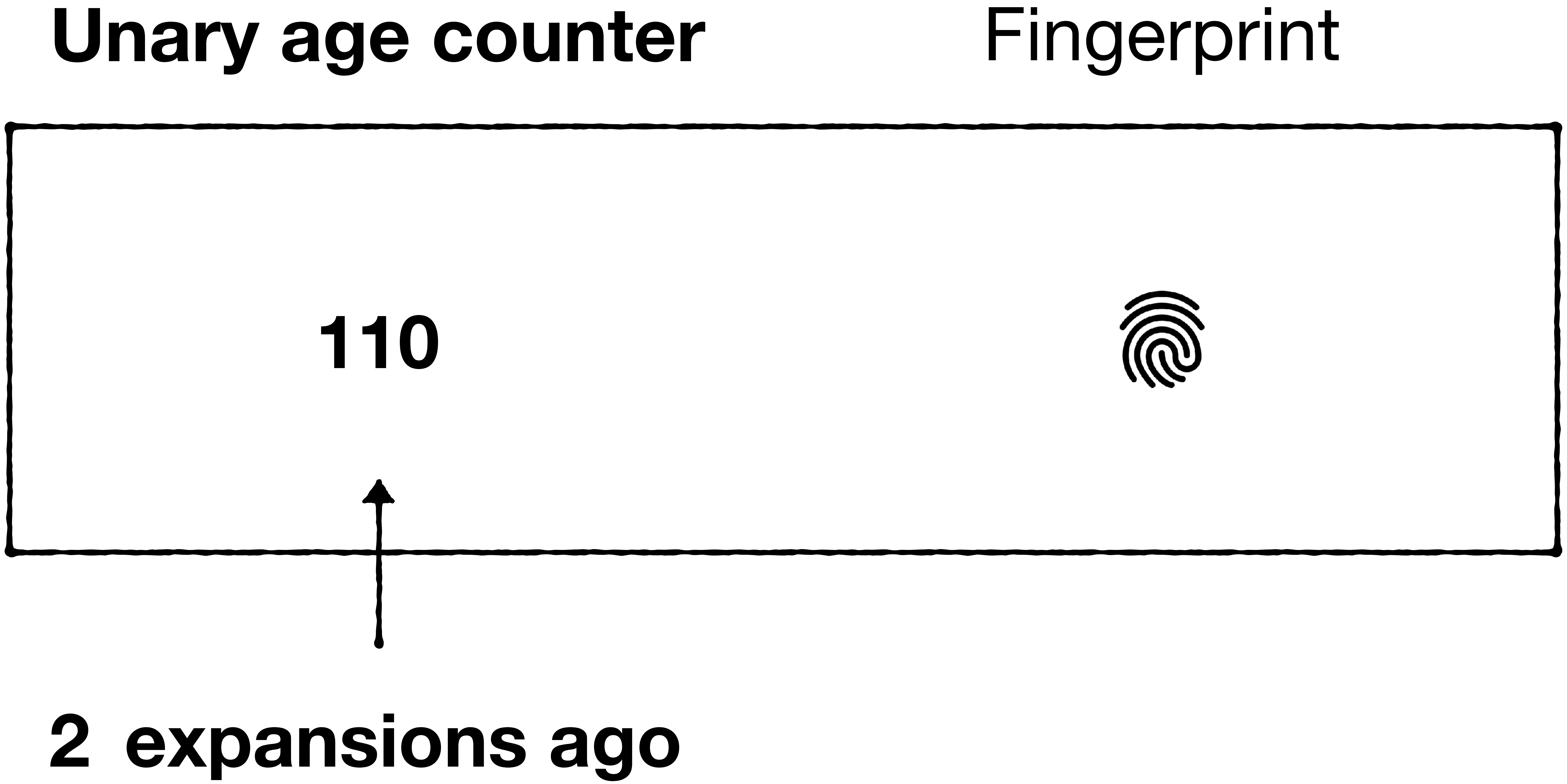


Unary age counter

Fingerprint

10

1 expansions ago

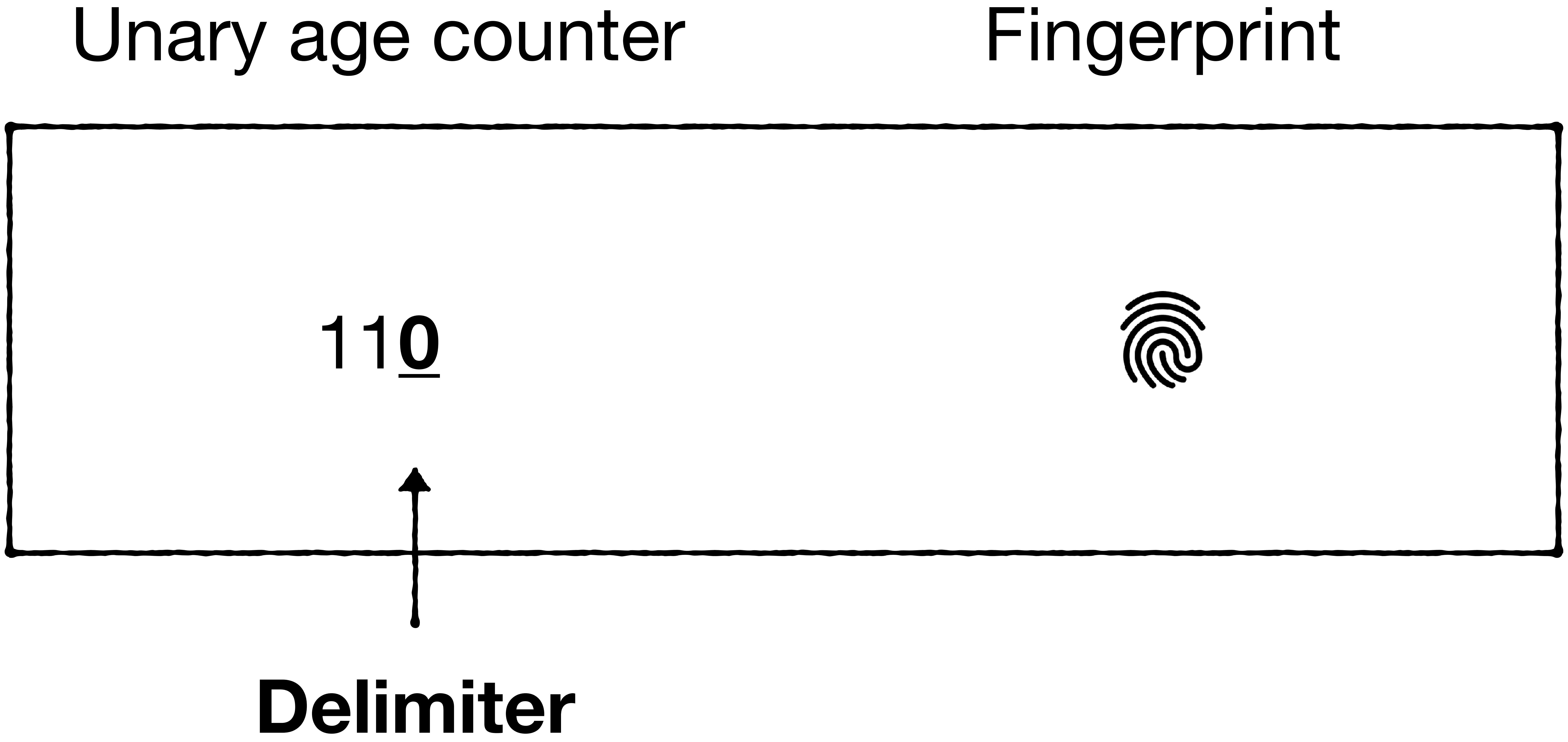


Unary age counter

Fingerprint

110

2 expansions ago

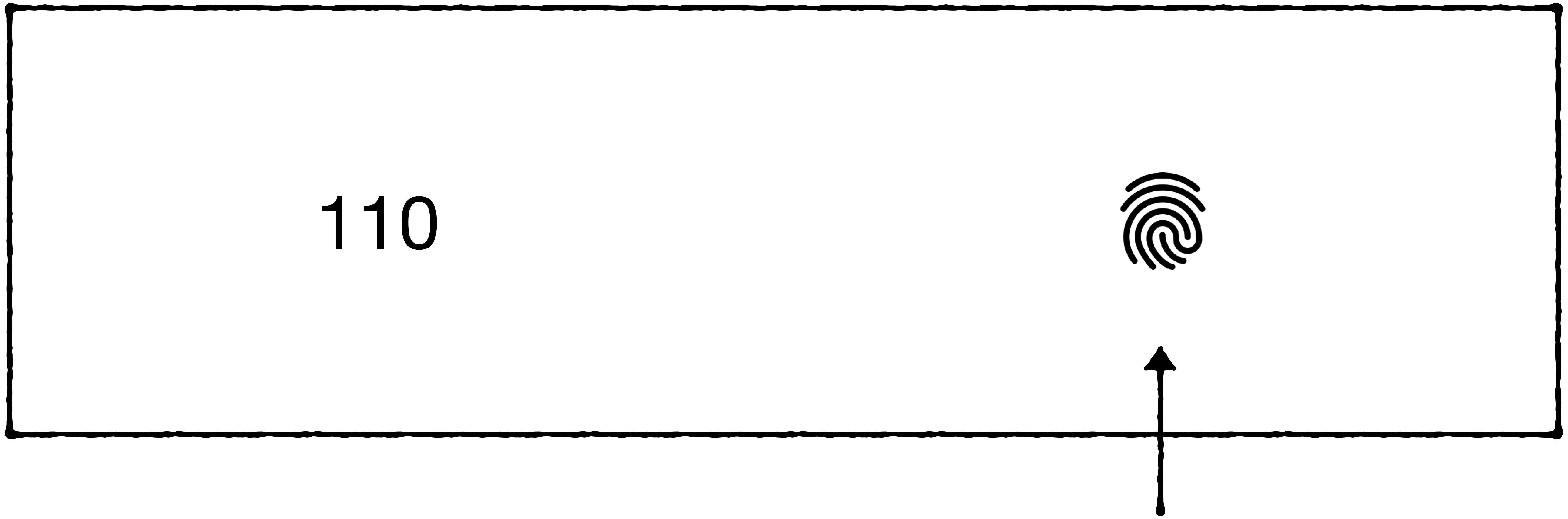


Unary age counter

Fingerprint

11**0**

Delimiter



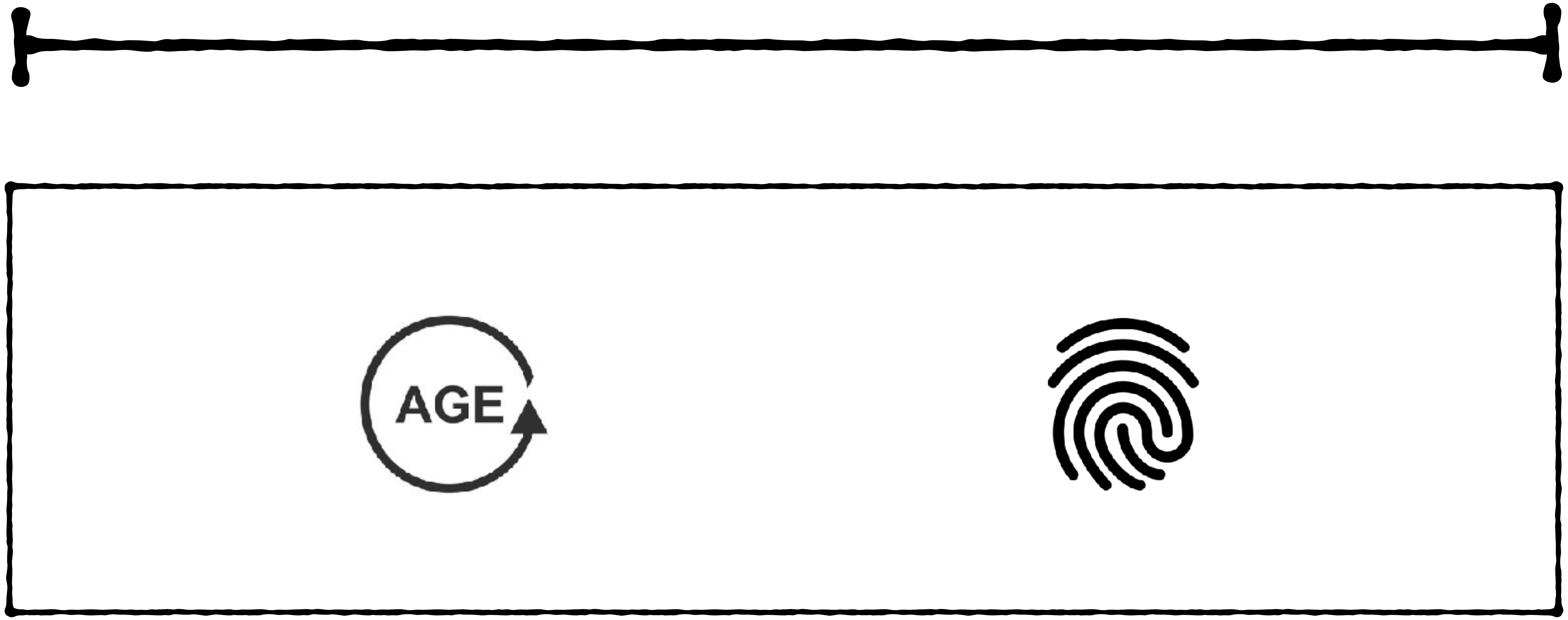
Unary age counter

Fingerprint

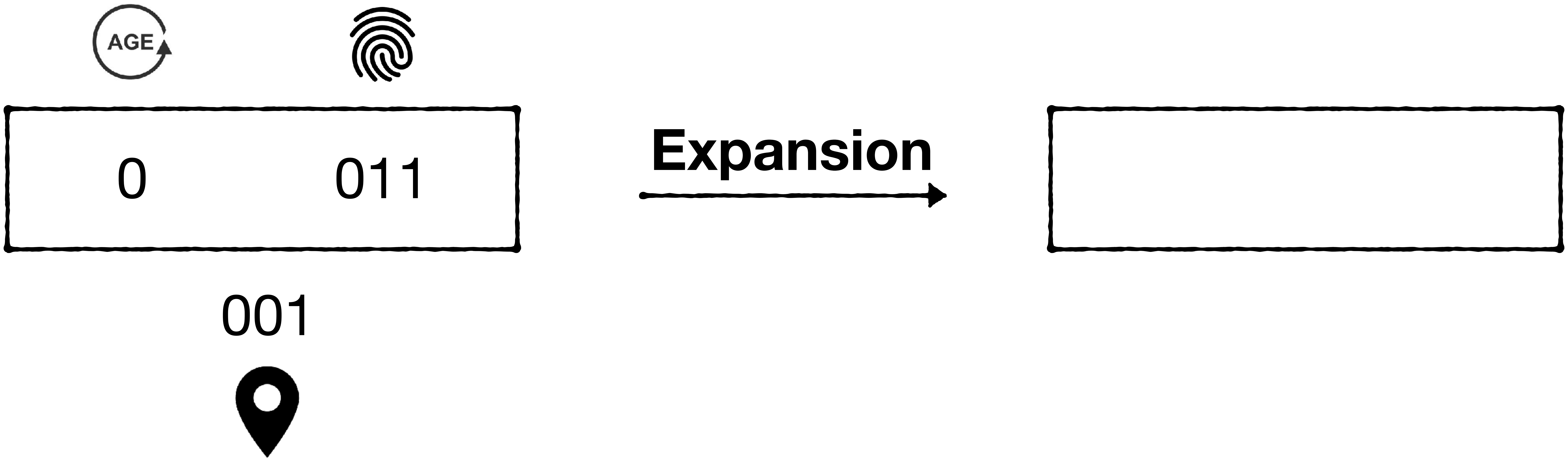
110

All remaining slot bits

Fixed-length



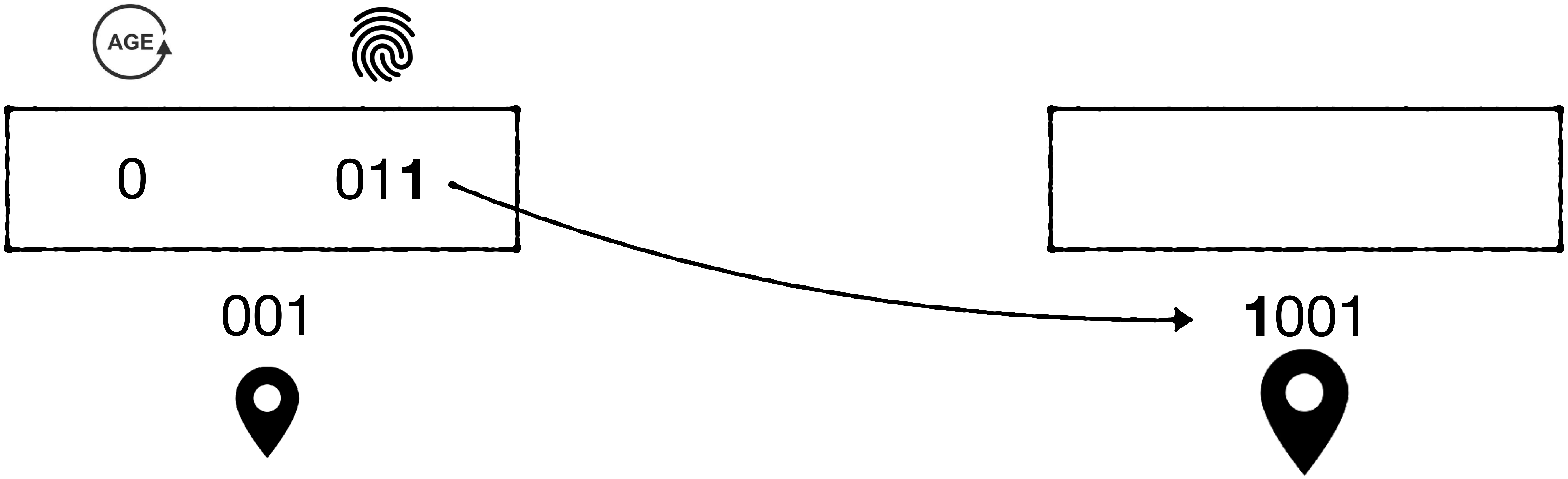
Expansion



0 011

Expansion

001

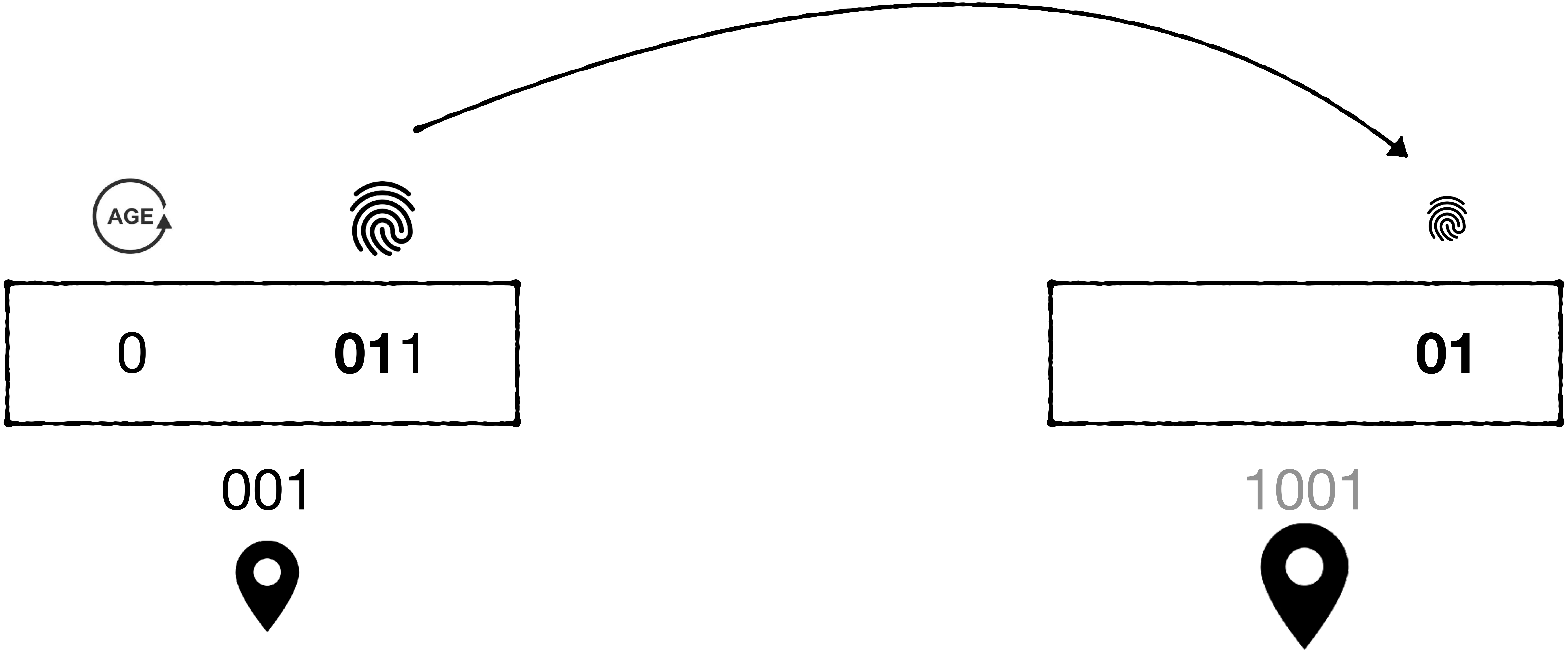


0

01**1**

001

1001

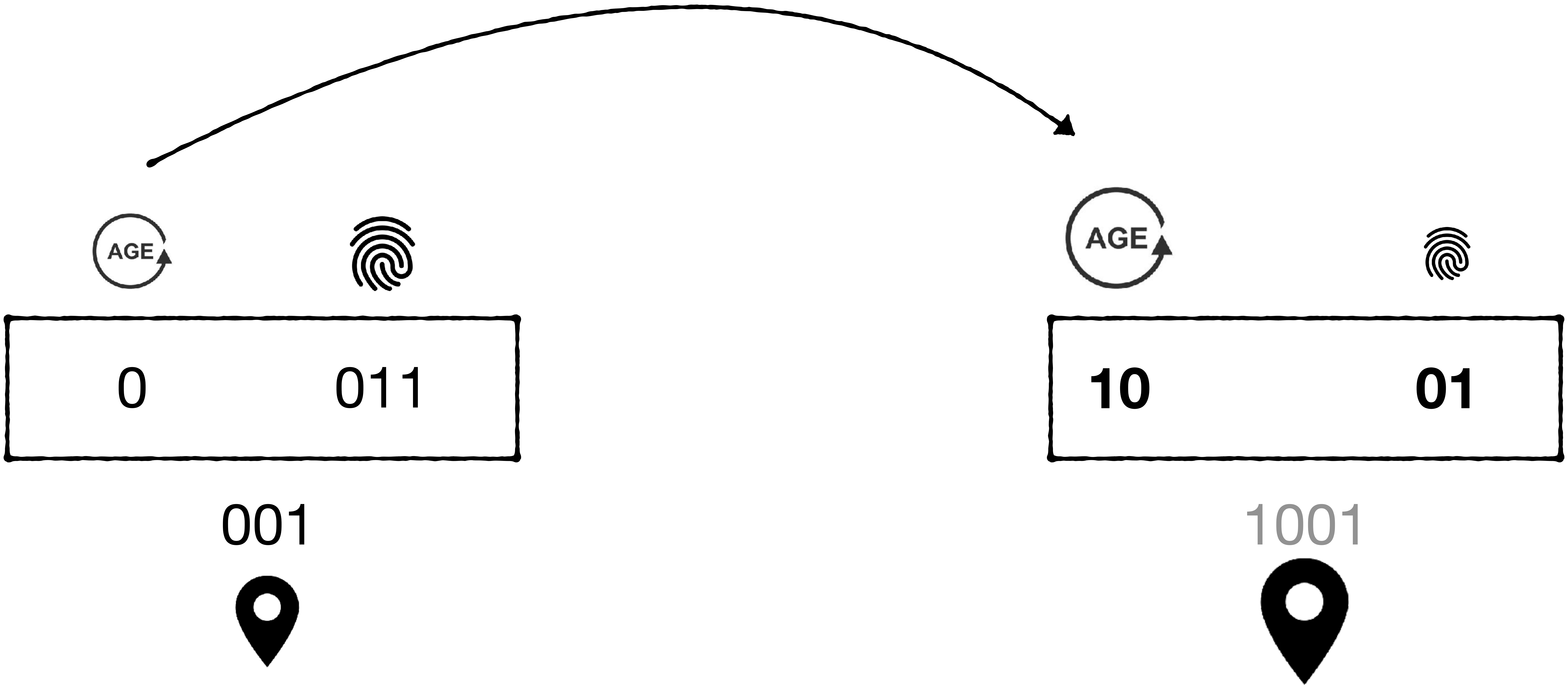


0 **01**1

01

001

1001



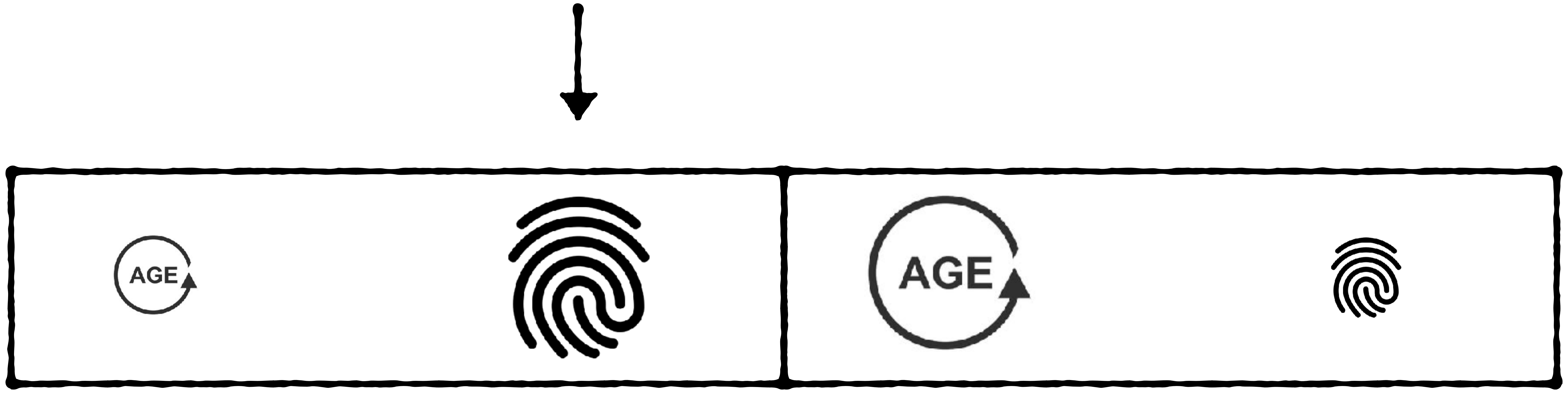
0 011

001

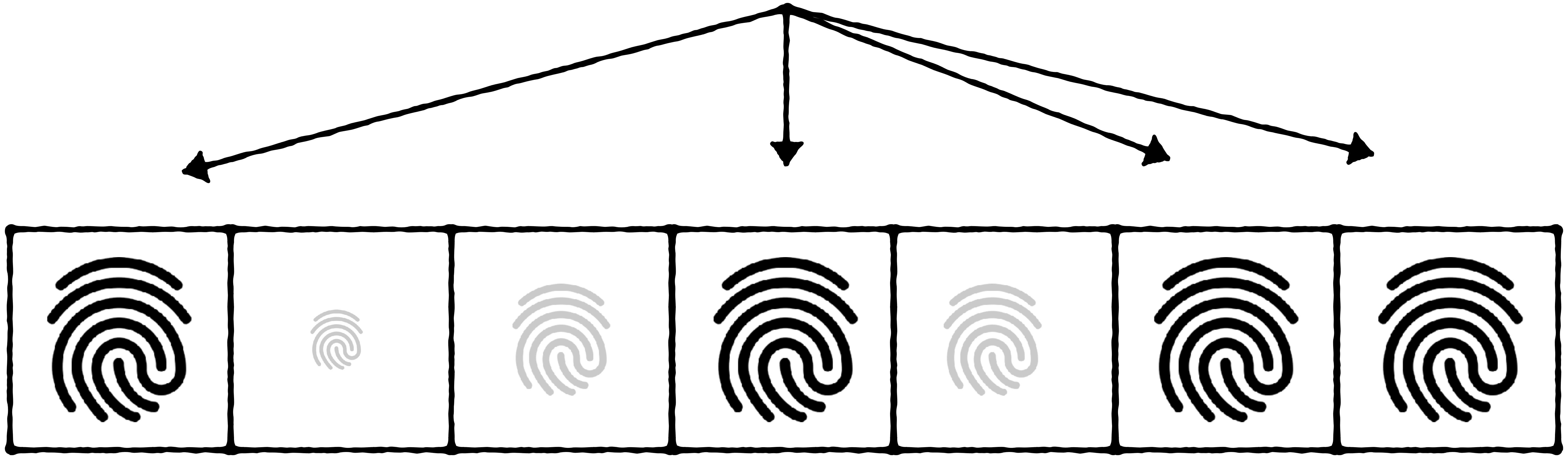
10 **01**

1001

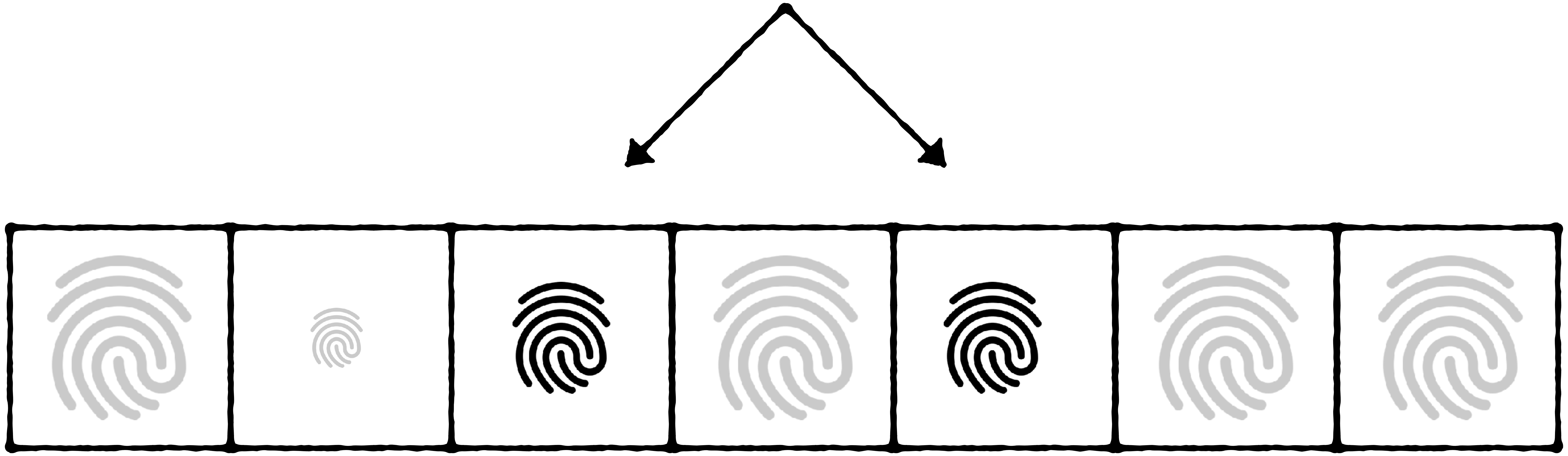
Longer fingerprints can be inserted after expansion



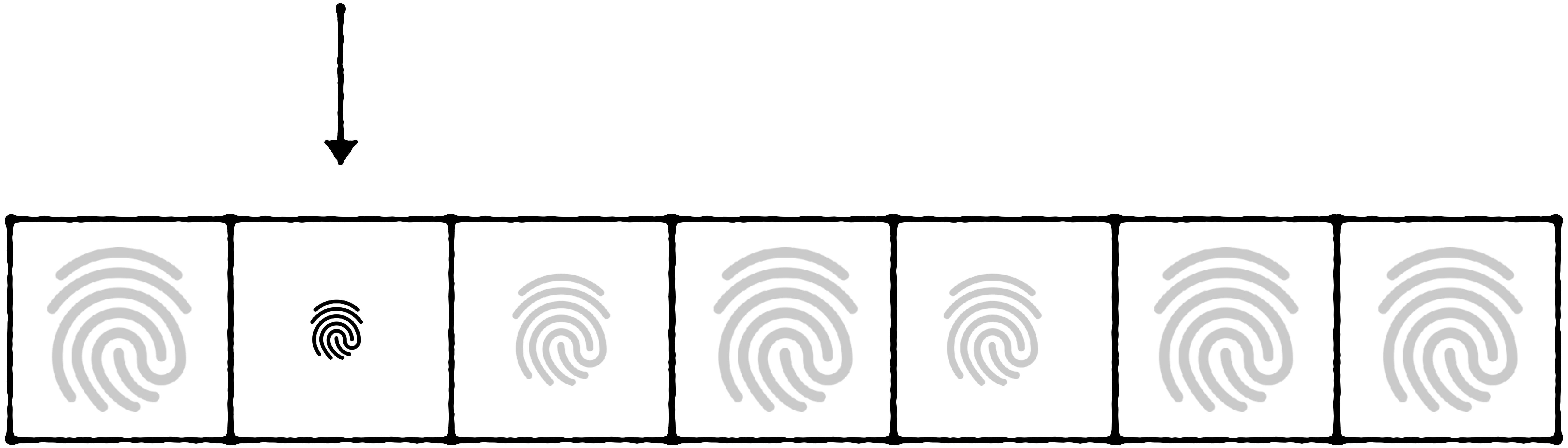
Half of entries have *F* bit fingerprints



Quarter have *F*-1 bit fingerprints



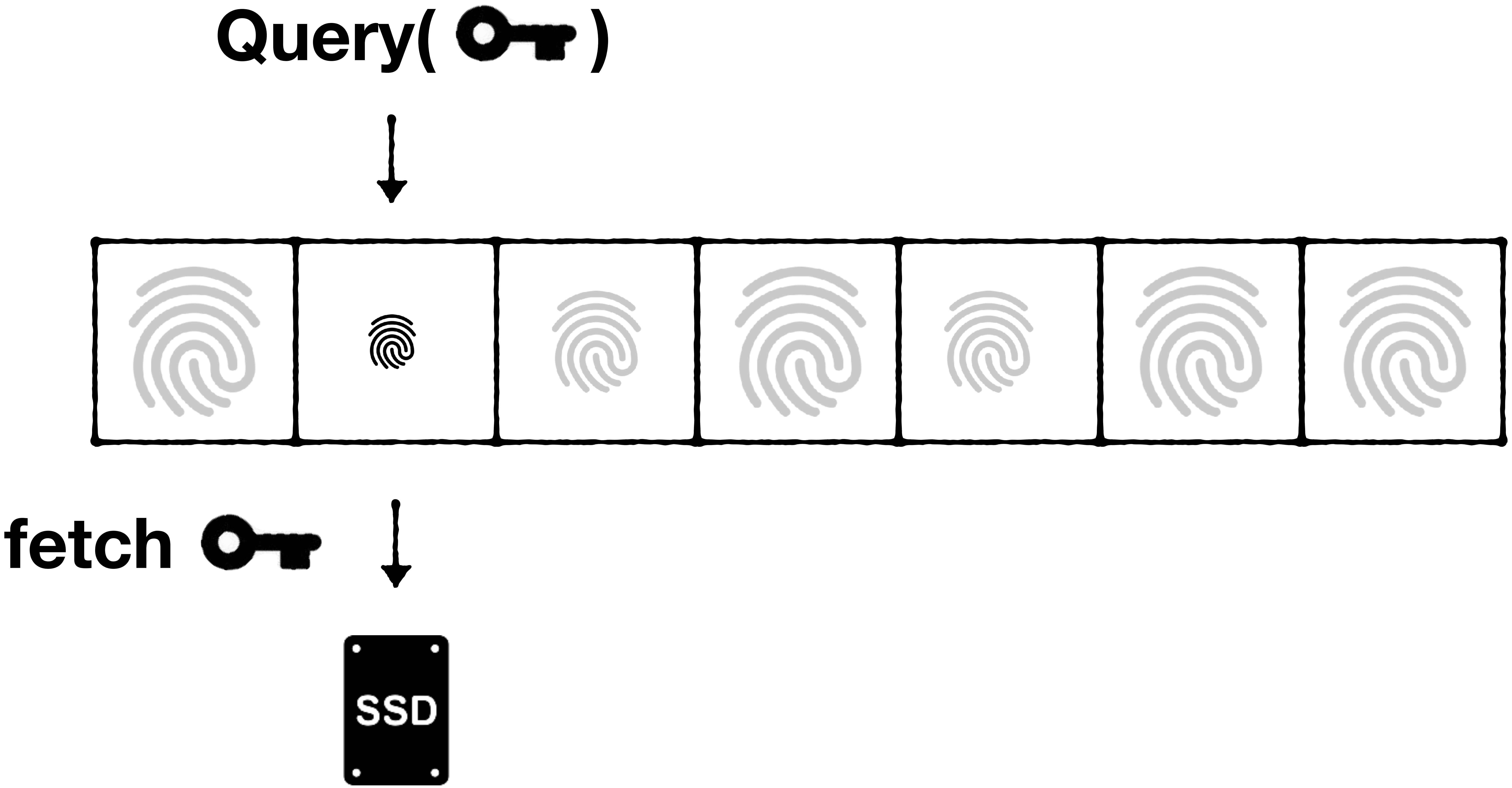
Eighth have *F*-2 bit fingerprints



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

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

with quotient
filter

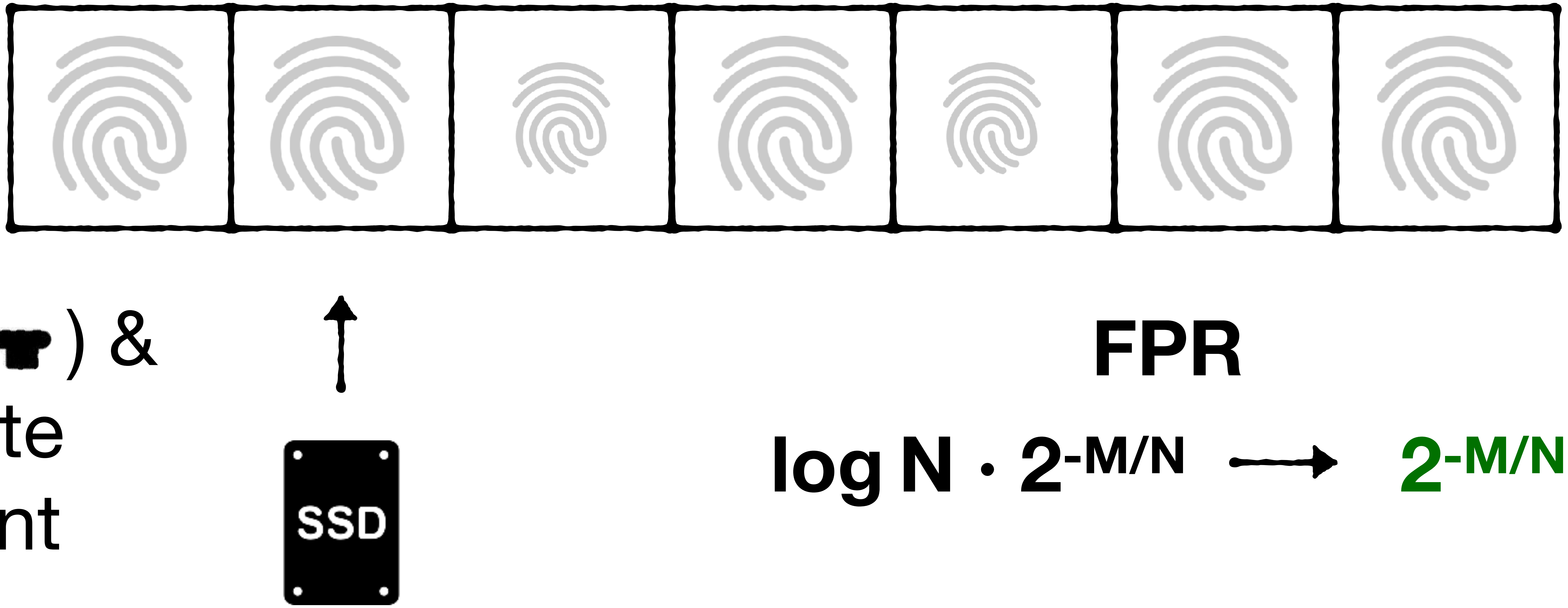


Query(🔑)

fetch 🔑

SSD

Rehash(🔑) &
rejuvenate
fingerprint



Rehash(🔑) &
rejuvenate
fingerprint

FPR

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

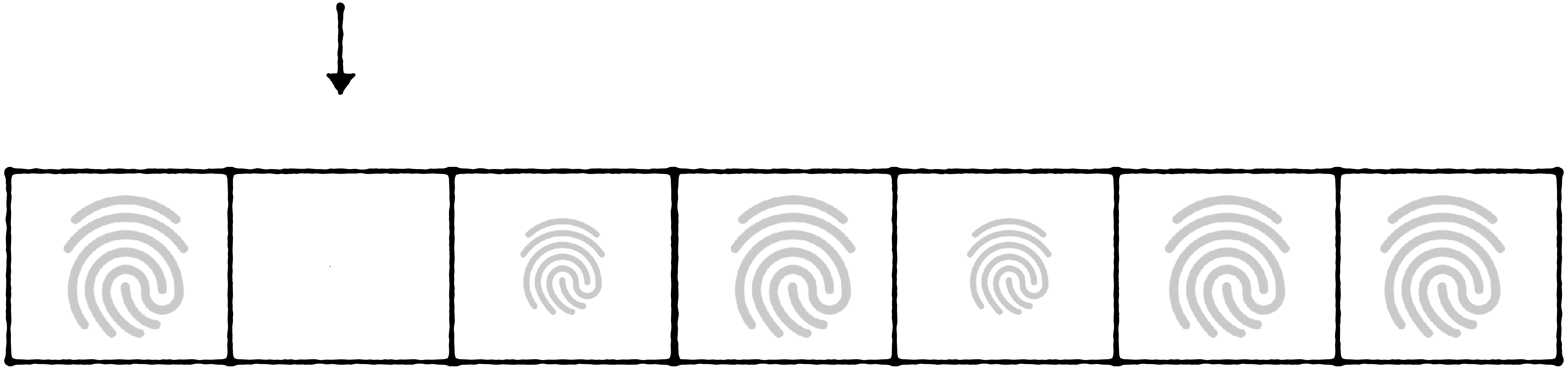
Increase slot width at rate of $\approx 2 \log_2 \log_2 N$

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

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

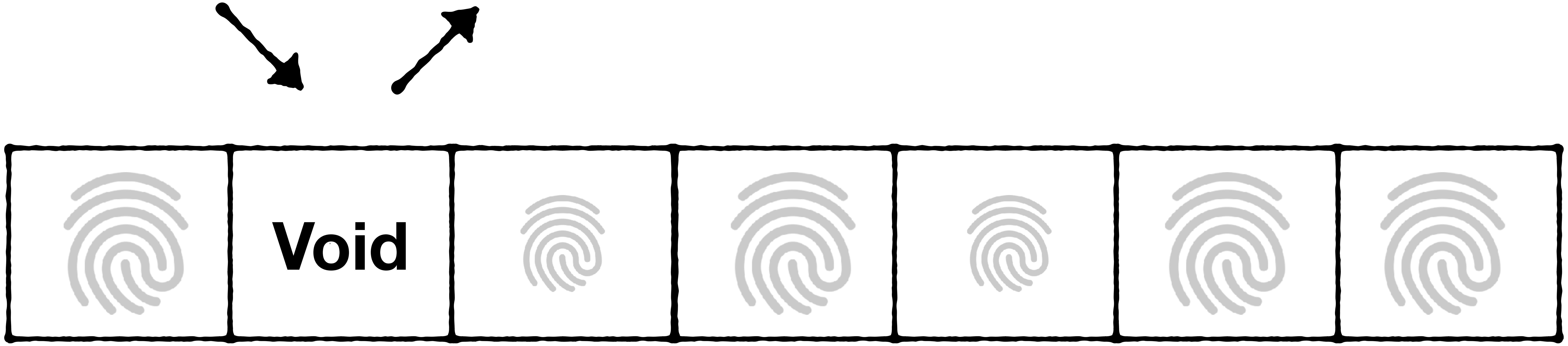
$$\text{FPR} \approx 2^{-M/N}$$

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



Unary padding occupies whole slot

Unary padding occupies whole slot

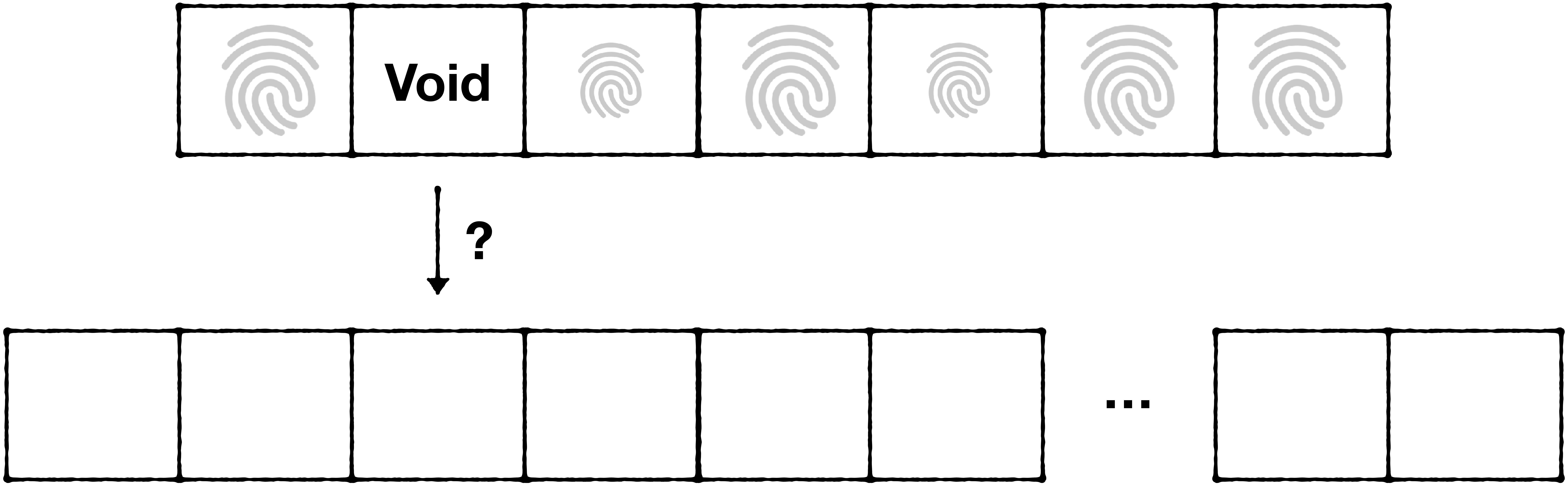


Any query

Positive

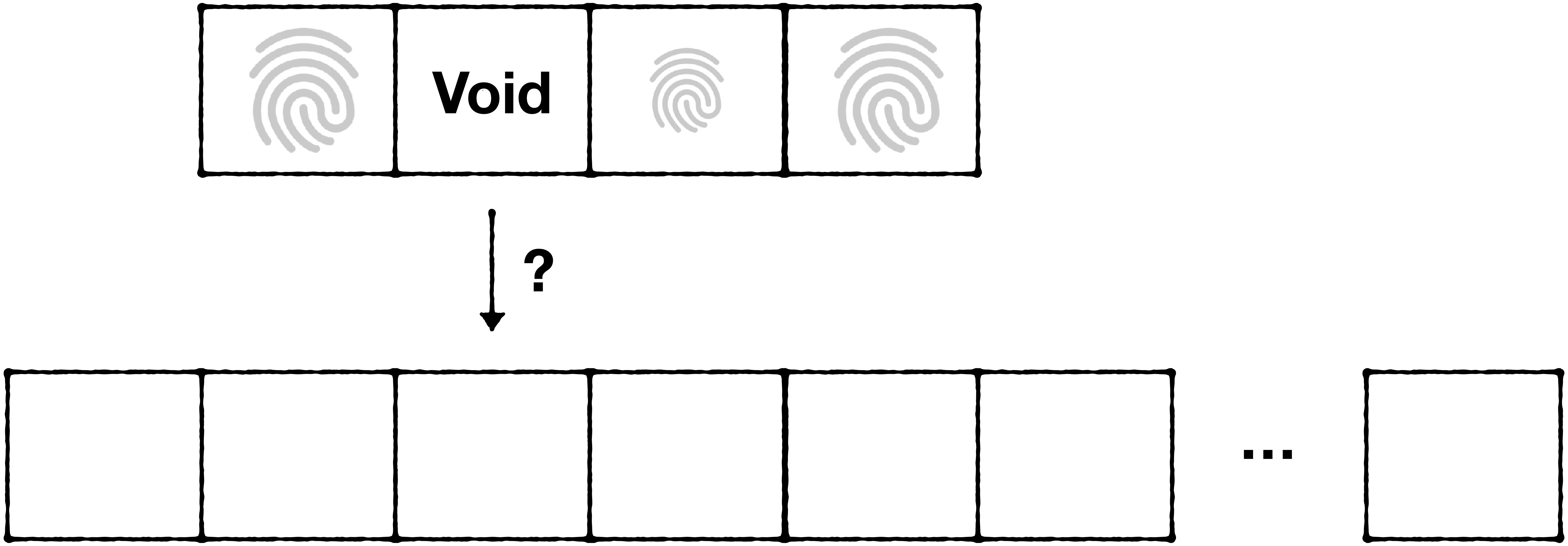
Void

How to continue expanding?

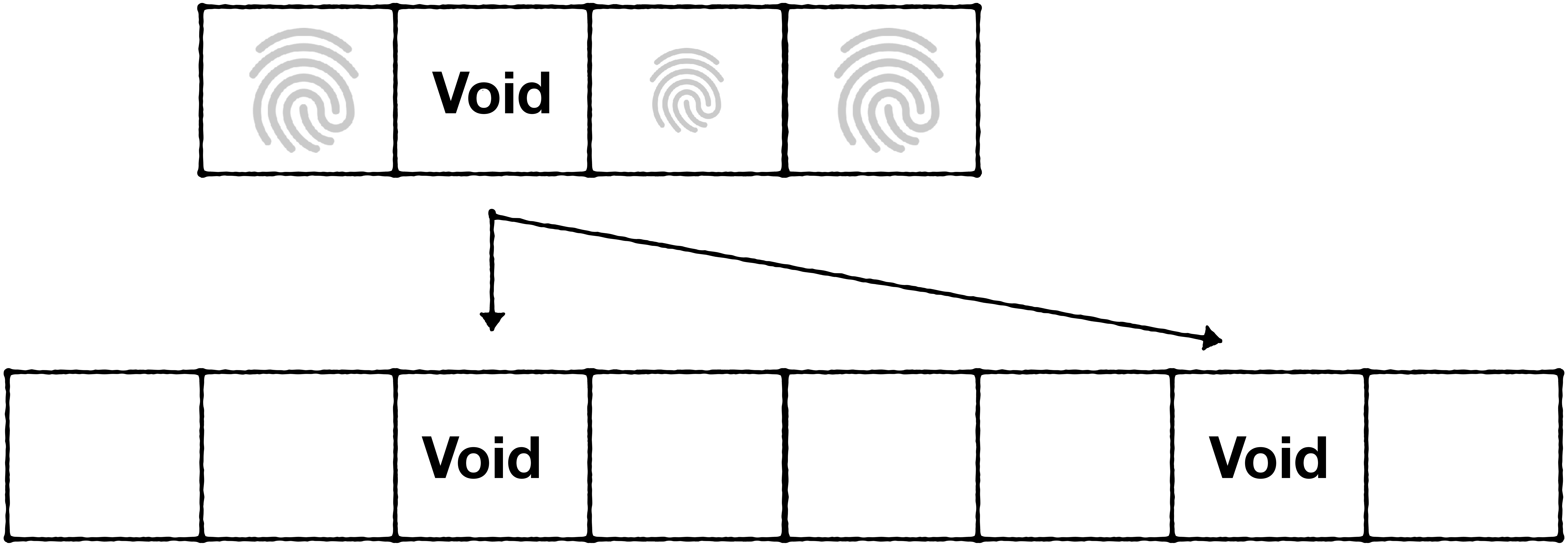


Aleph Filter: To Infinity in Constant Time

Niv Dayan, Ioana Bercea, Rasmus Pagh.   VLDB 2024



Duplicate



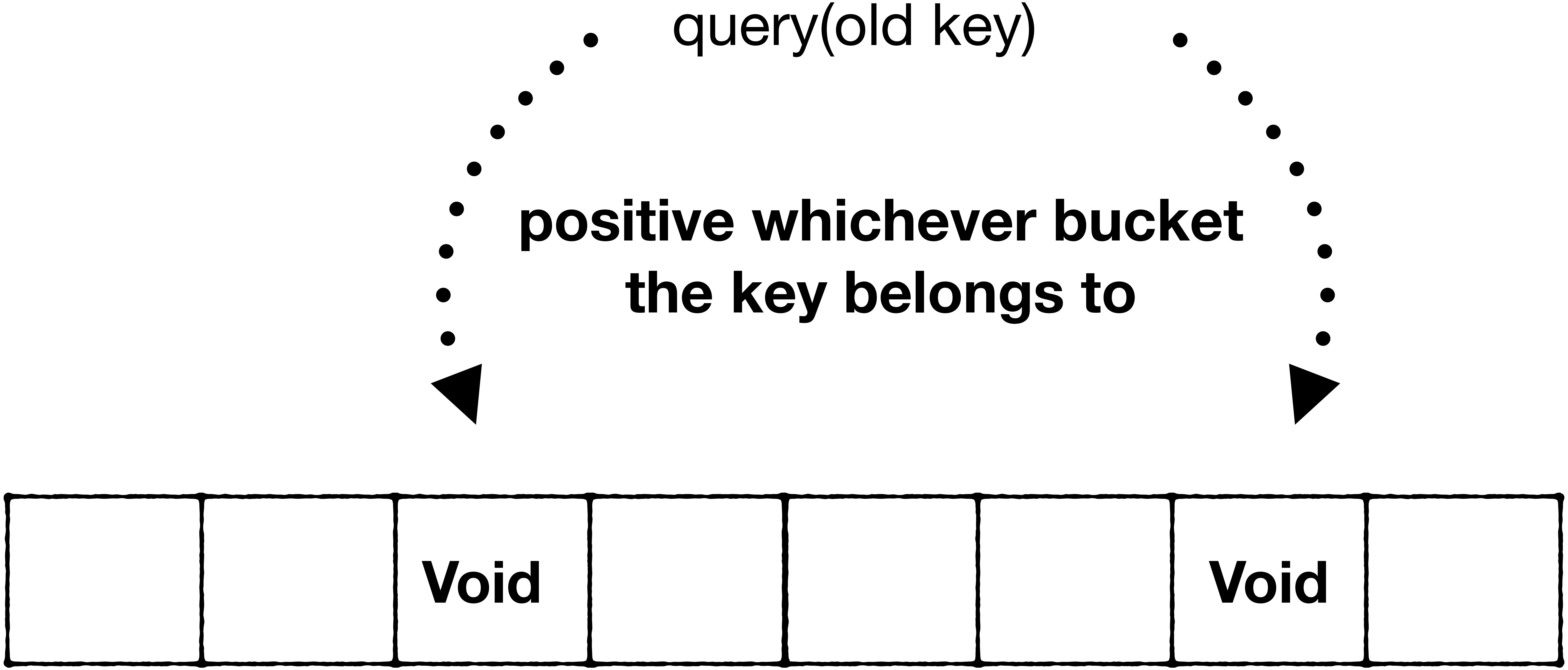
Void

Void

Void

query(old key)

		Void				**Void**	



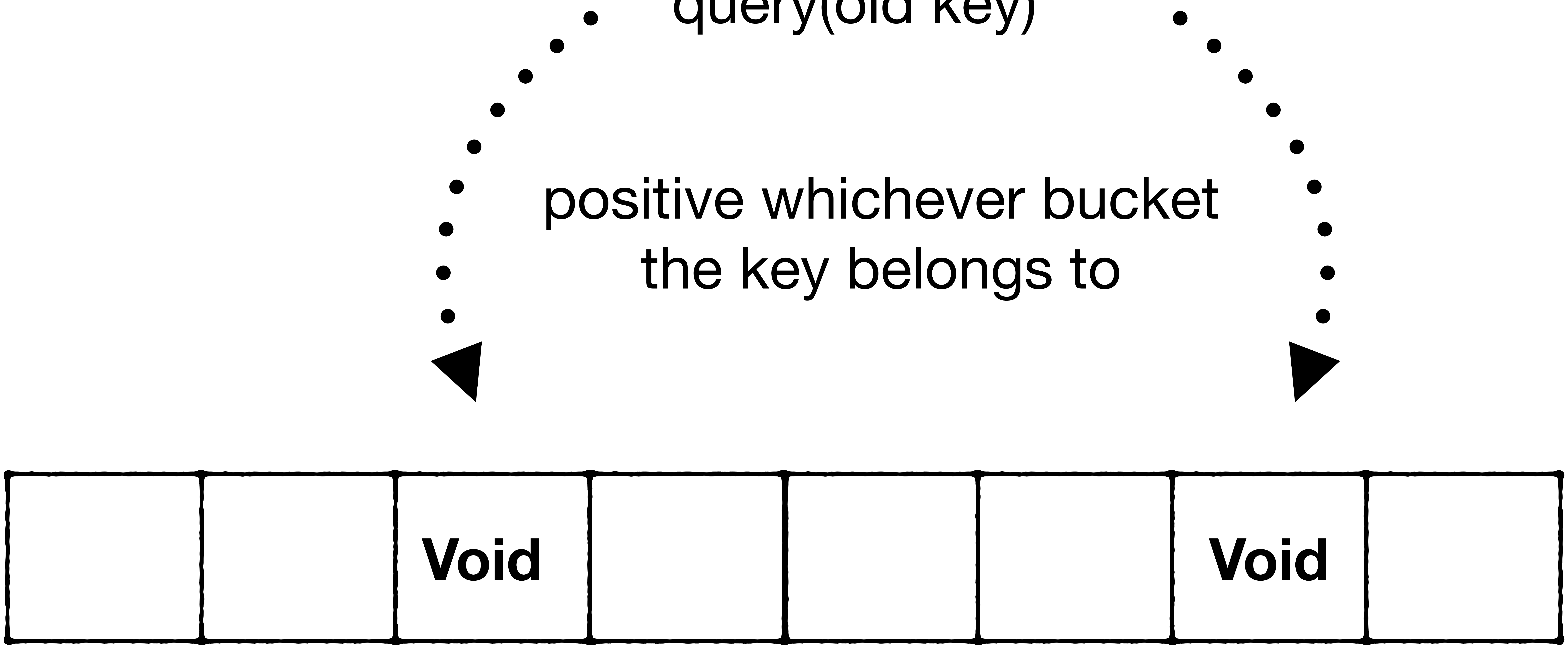
query(old key)

positive whichever bucket the key belongs to

Void

Void

Expand Indefinitely with O(1) performance



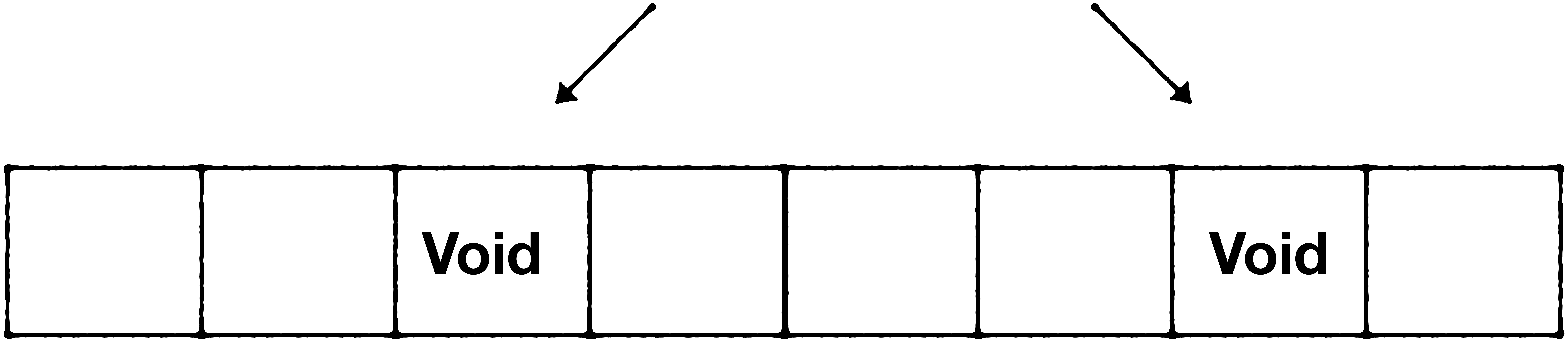
query(old key)

positive whichever bucket
the key belongs to

		Void				**Void**	

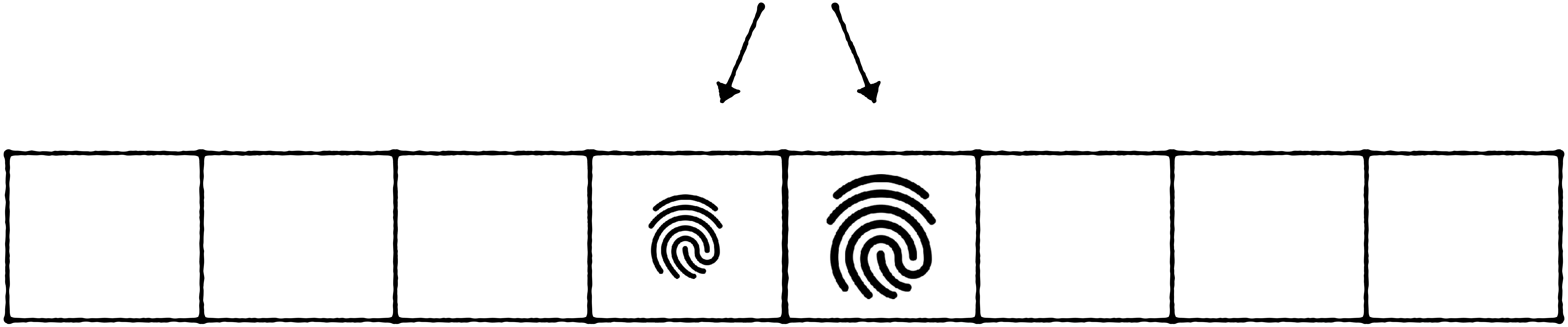
Expandable Filters Complicate Deletes

Identify how many void entries to remove



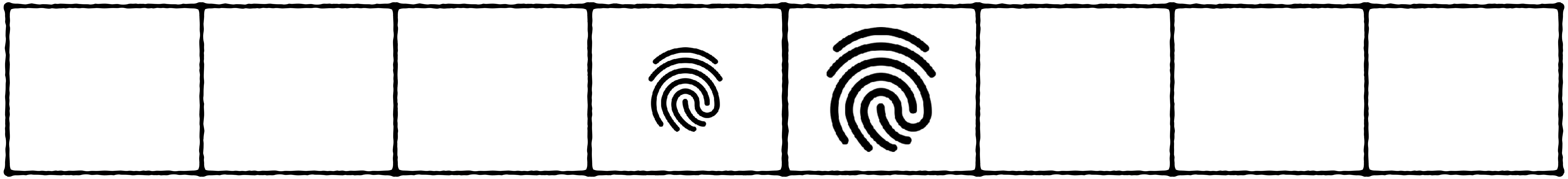
Expandable Filters Complicate Deletes

Multiple fingerprints of diff lengths may match key to delete



Expandable Filters Complicate Deletes

Solutions exist in the papers :)



Thank you!