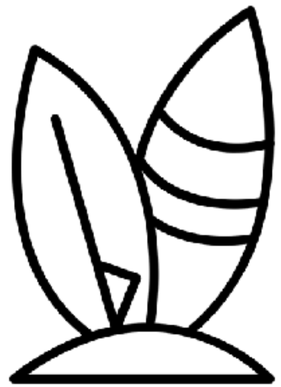
Range Filtering at Orca Lab

Navid Eslami, Niv Dayan

CSC2525: Research Topics
in Database Management

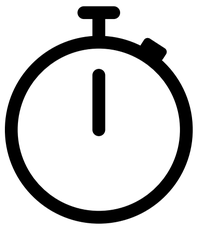
Problems?



SuRF

Robustness

Dynamicity

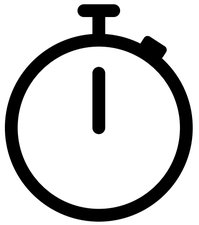


Speed

No range filter achieved all at the same time

Robustness

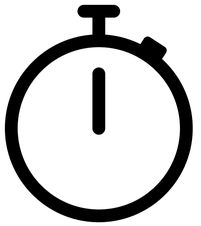
Dynamicity



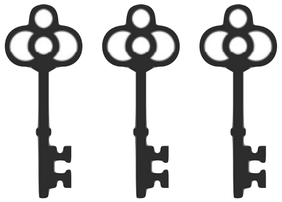
Speed

Robustness

Dynamicity



Speed



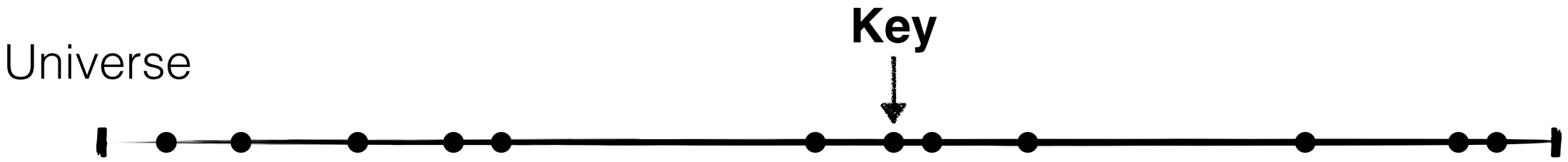
Finite universe (i.e., fixed-length keys)

Range queries of length ≤ R

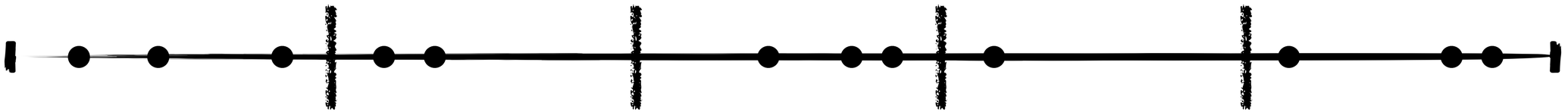
Memento Filter

Memento Filter

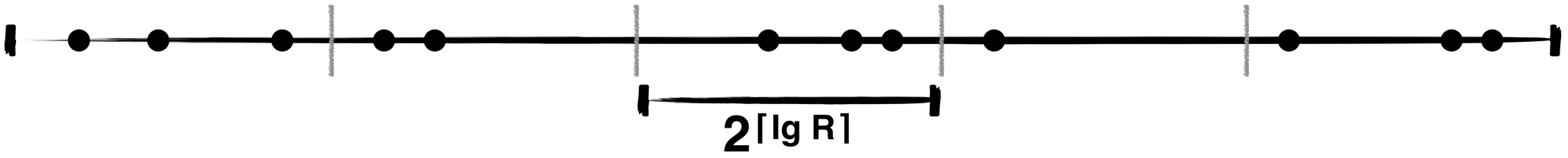
Universe


Key
Universe

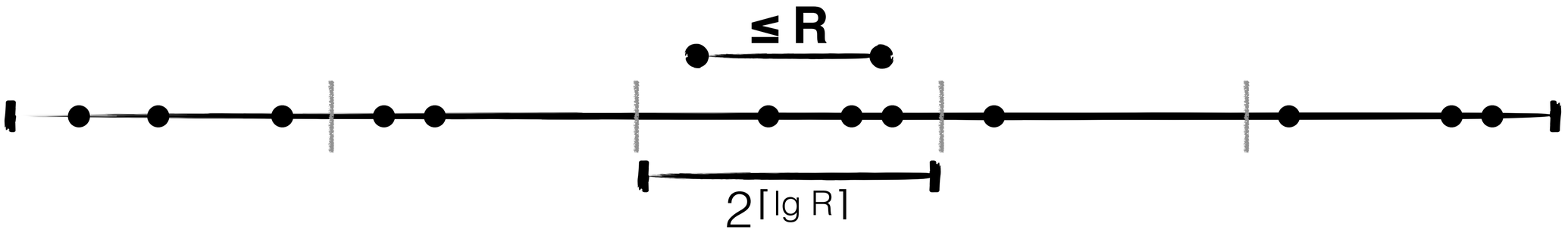
Key idea: partition universe



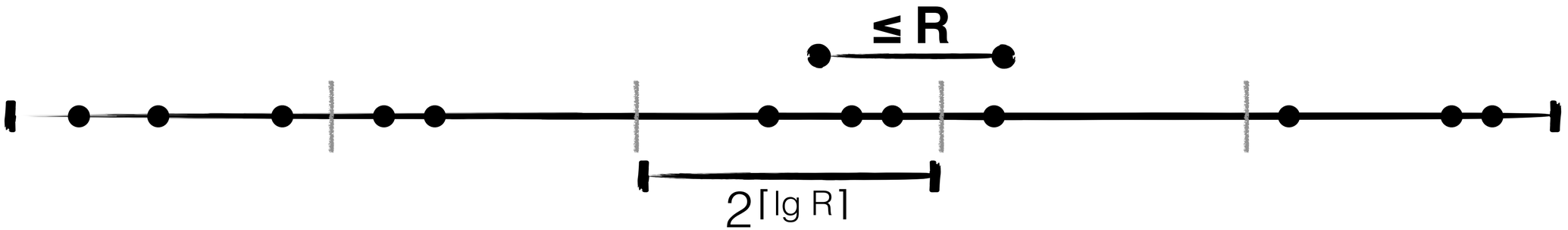
Key idea: partition universe



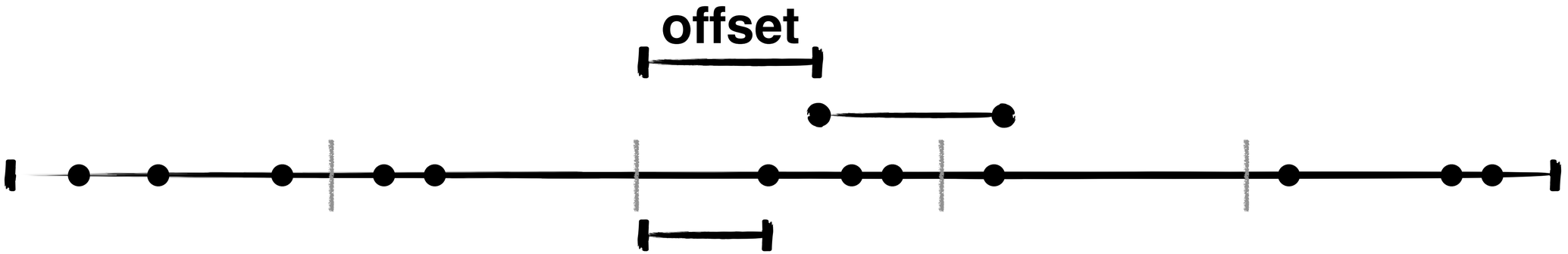
Queries intersect at most two adjacent partitions

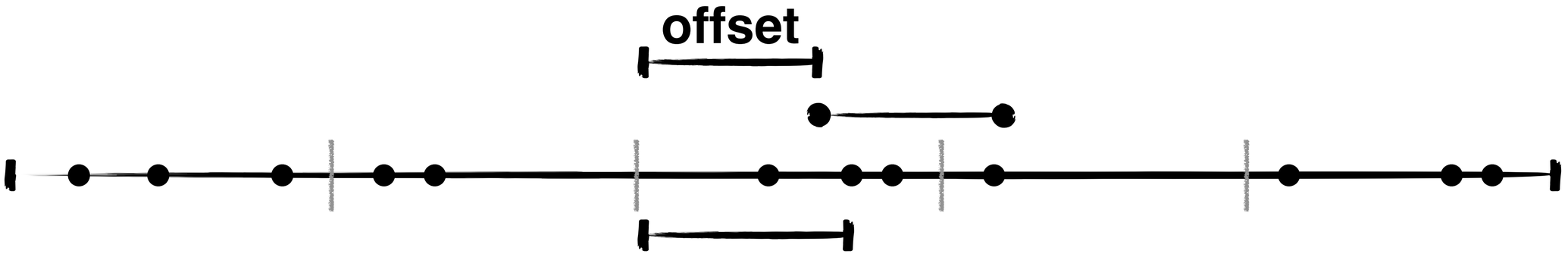


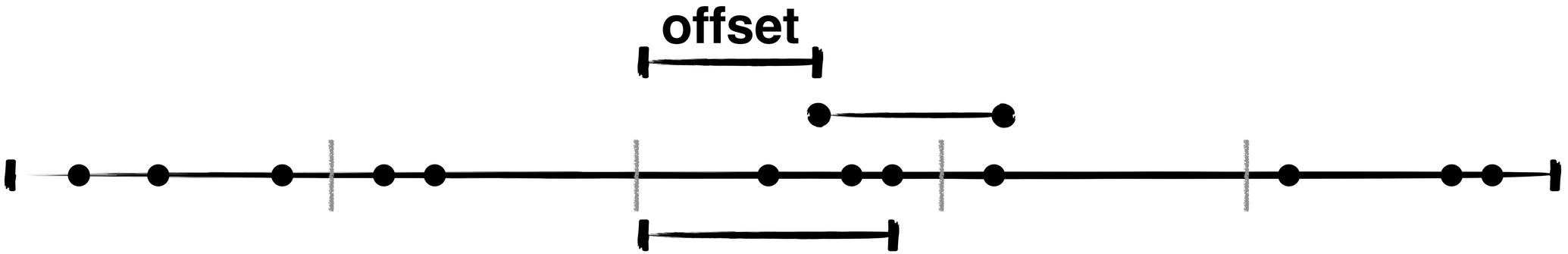
Queries intersect at most two adjacent partitions



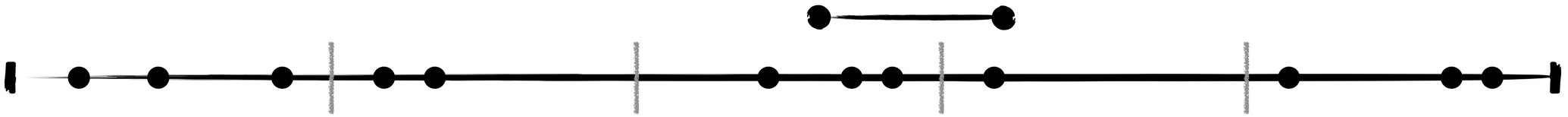
Queries intersect at most two adjacent partitions




offset

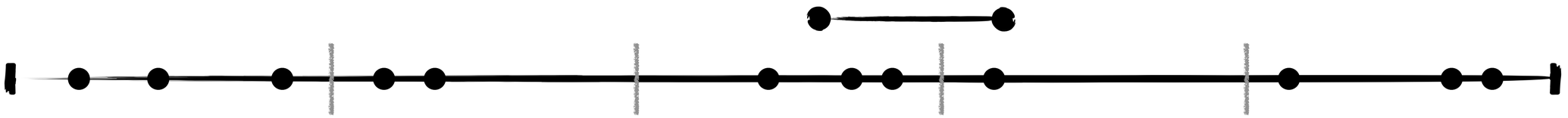

offset

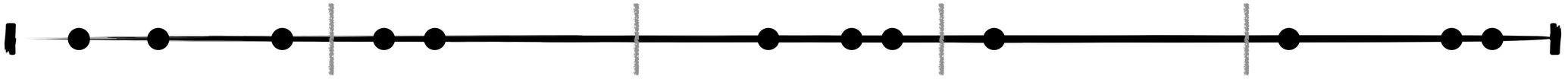
Compare offsets of keys and query endpoints

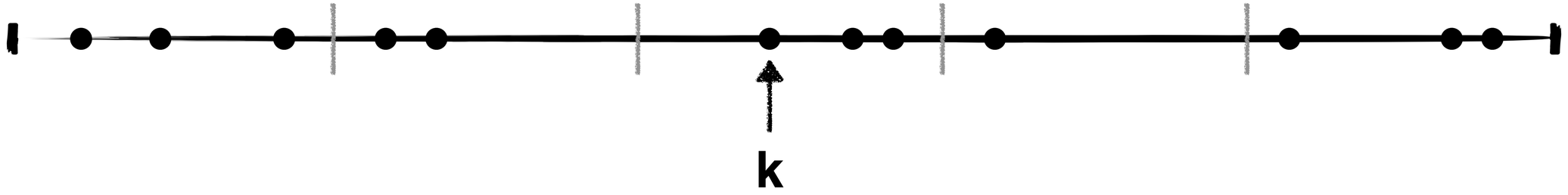


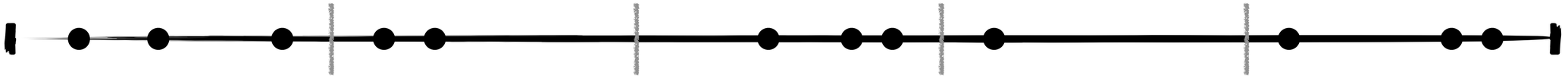
%

Store offsets in a Quotient filter (week 3)

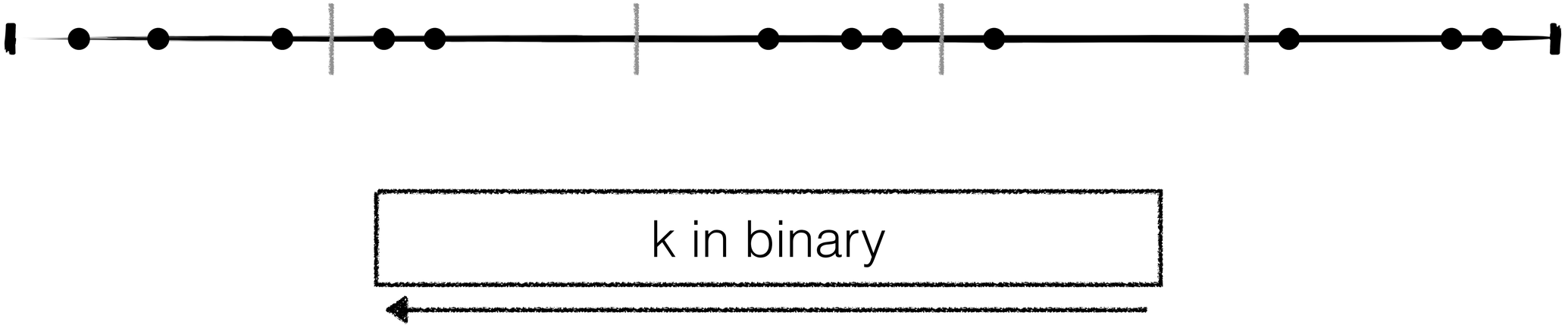


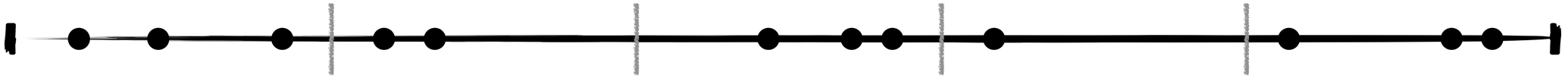



k

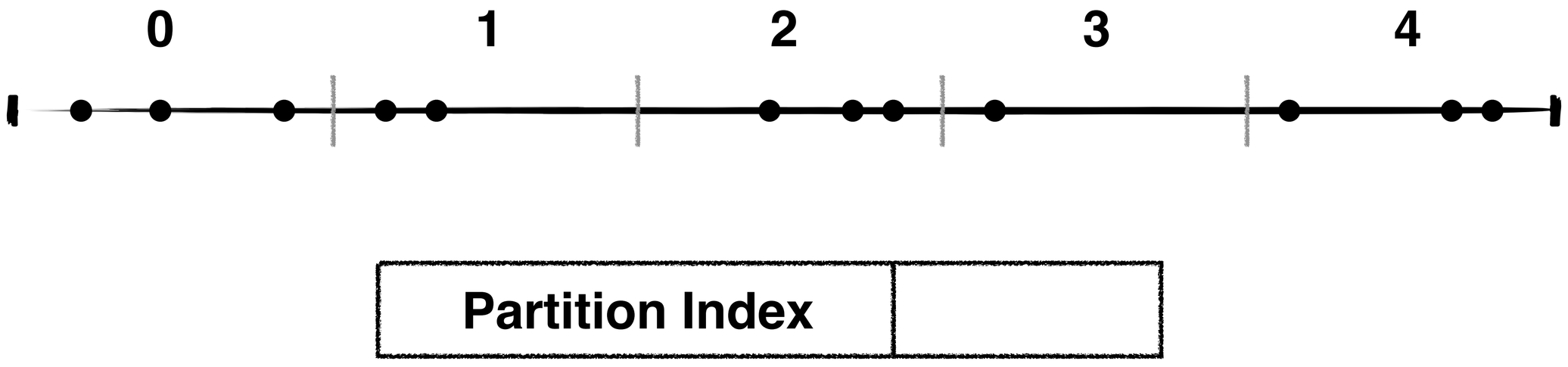


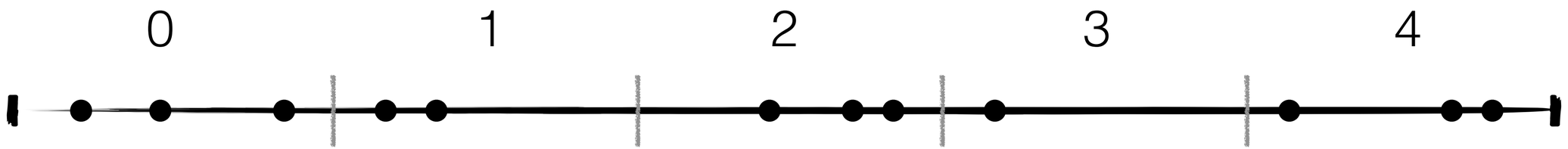
k in binary


k in binary
Higher-order bits
Lower-order bits

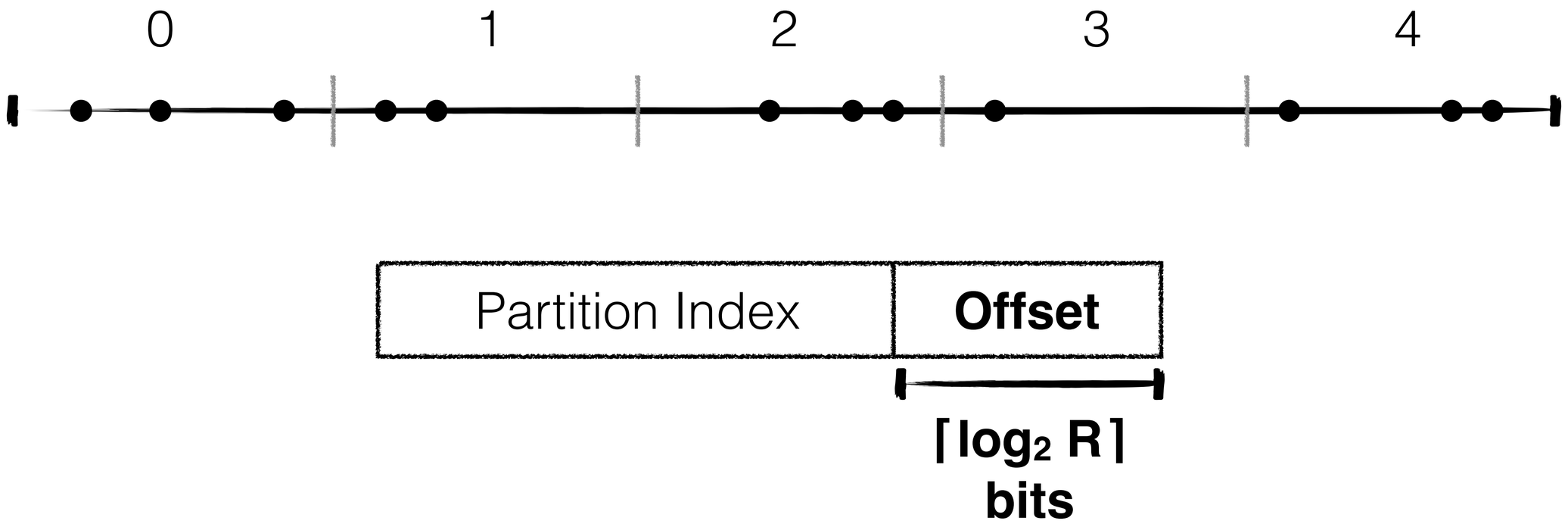


Partition Index	

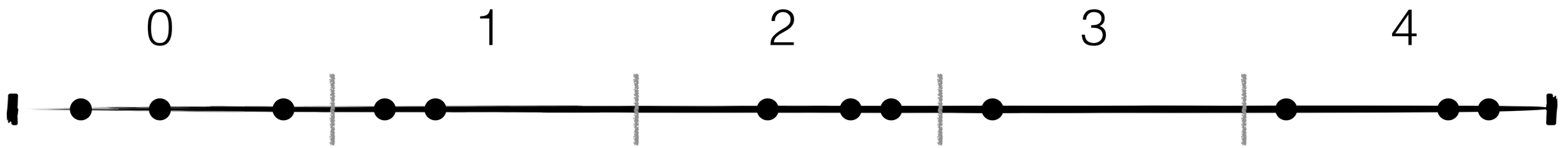

0
1
2
3
4
Partition Index

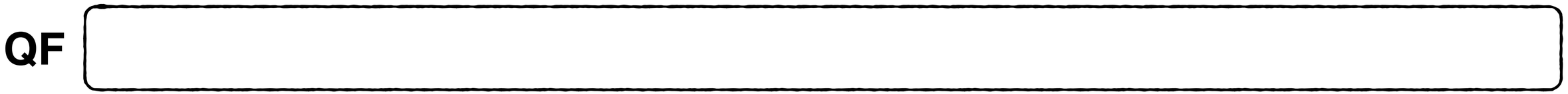


Partition Index	**Offset**

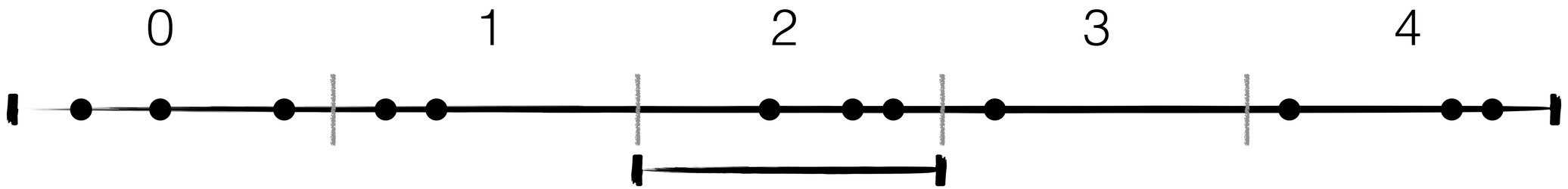

0
1
2
3
4
Partition Index
Offset
⌈log2 R⌉
bits

Storage layout

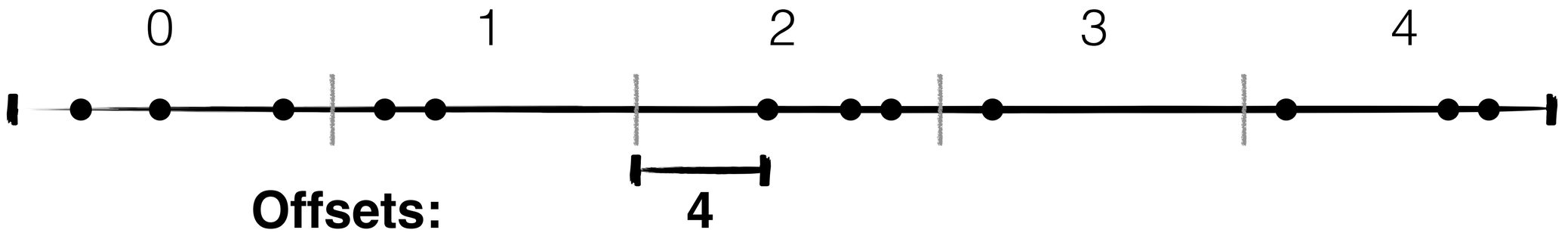




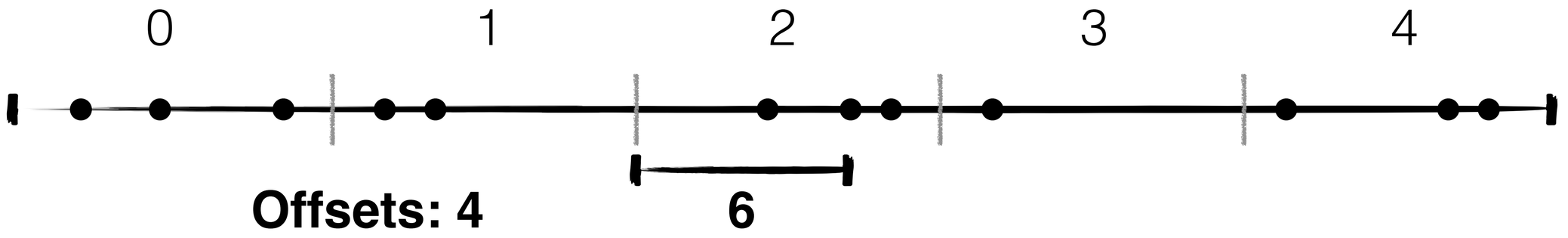
Storage layout

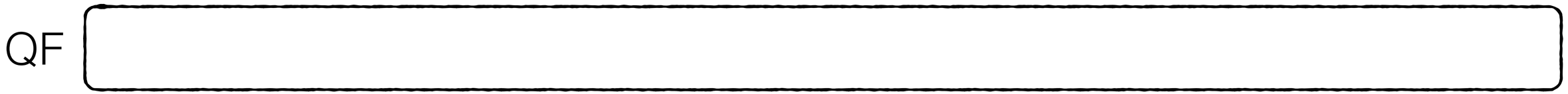


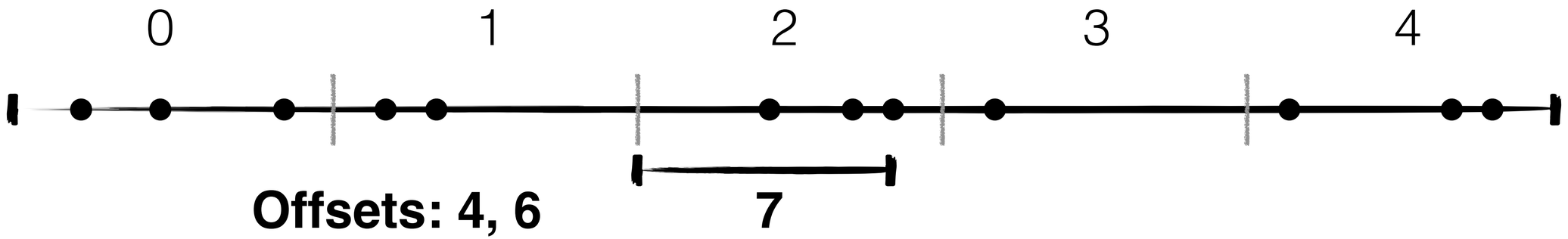
QF

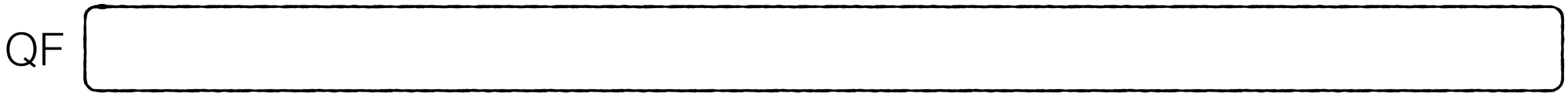


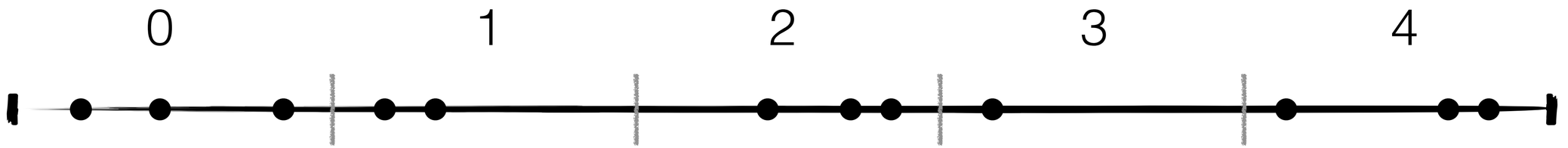
QF


0
1
2
3
4
Offsets: 4
6

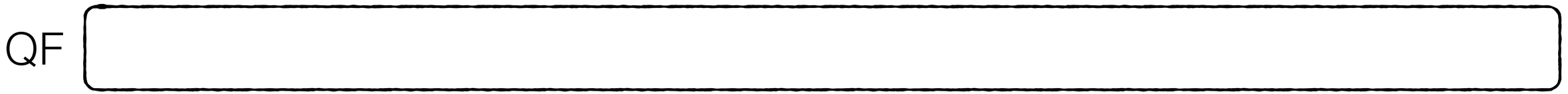

QF


0
1
2
3
4
Offsets: 4, 6
7

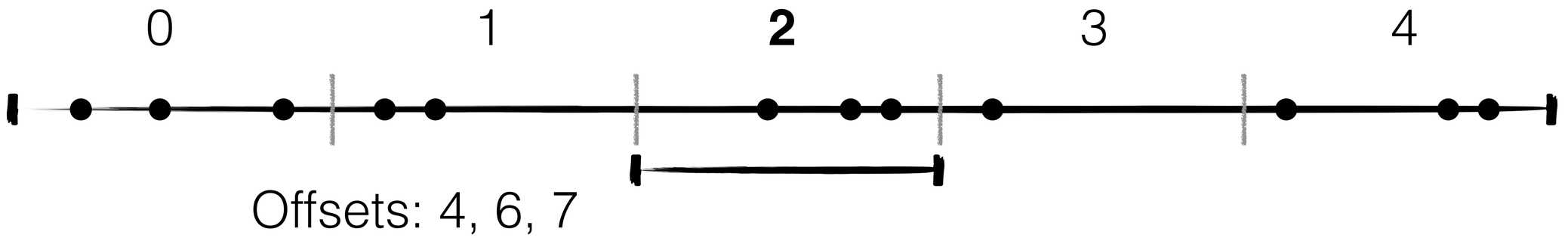

QF



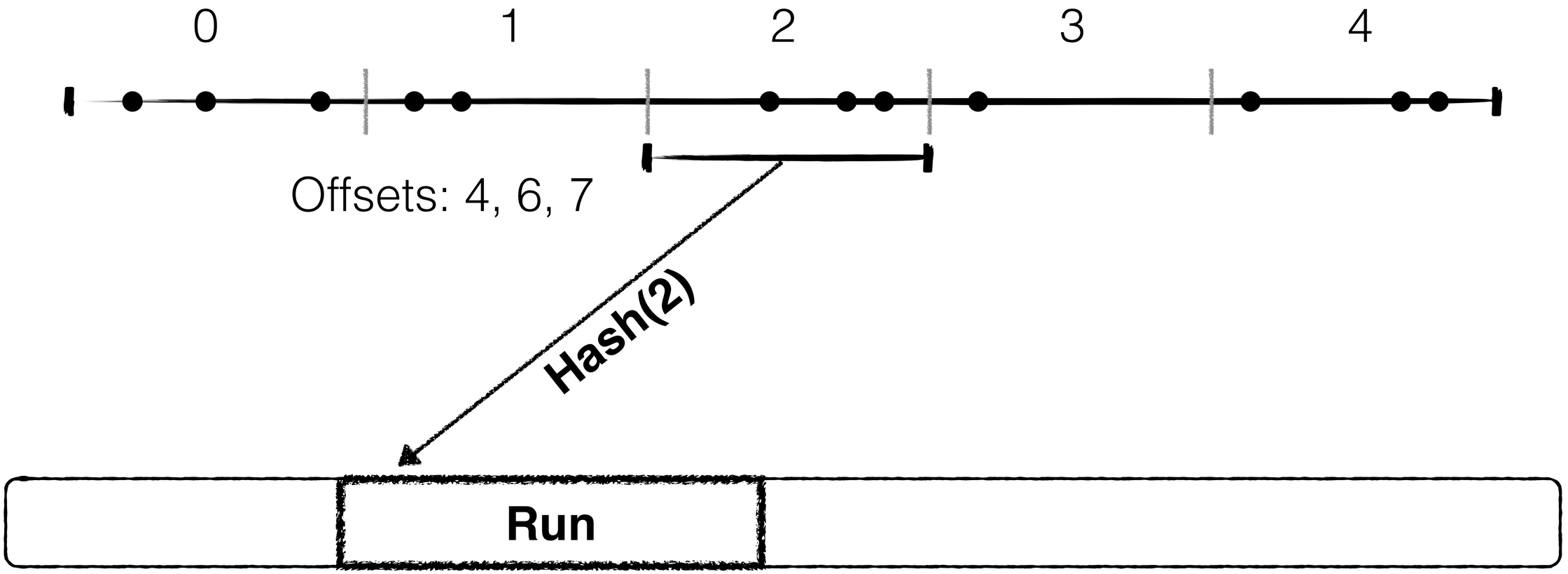
Offsets: 4, 6, 7

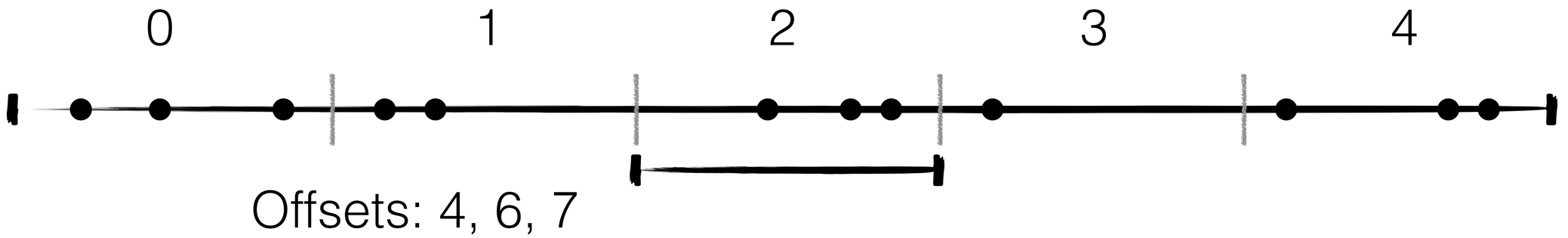


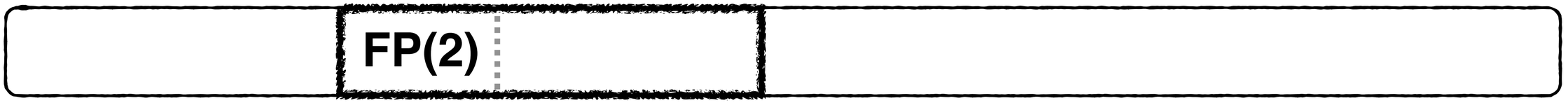
Hash partition number

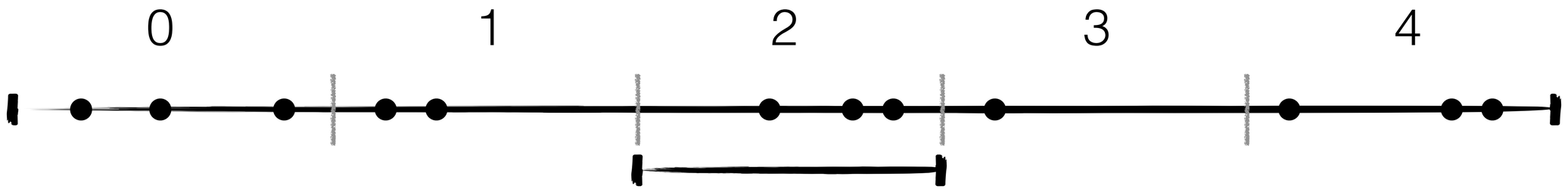


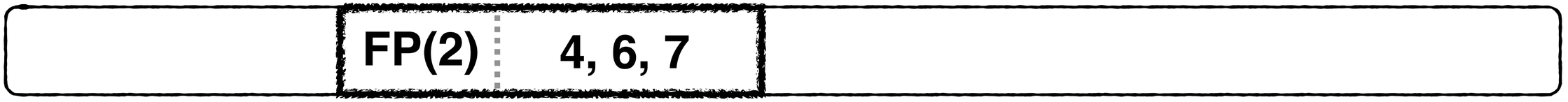
QF

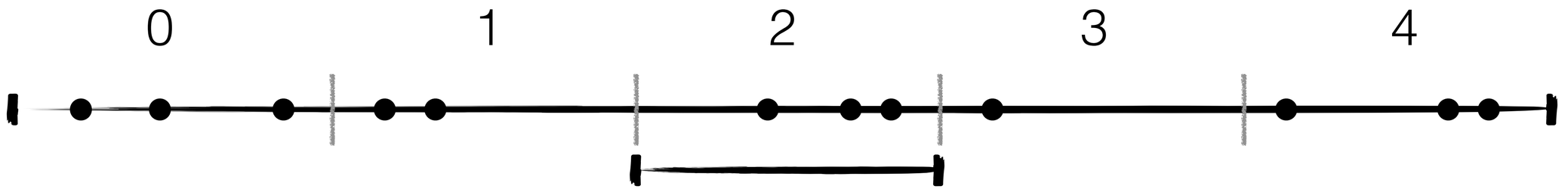

0
1
2
3
4
Offsets: 4, 6, 7
Hash(2)
Run

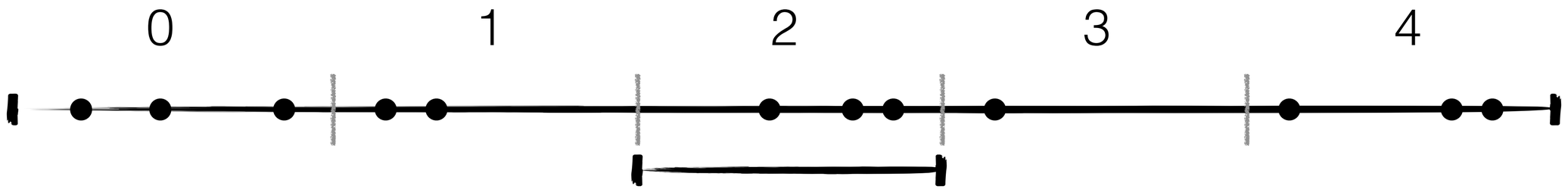

0
1
2
3
4
Offsets: 4, 6, 7

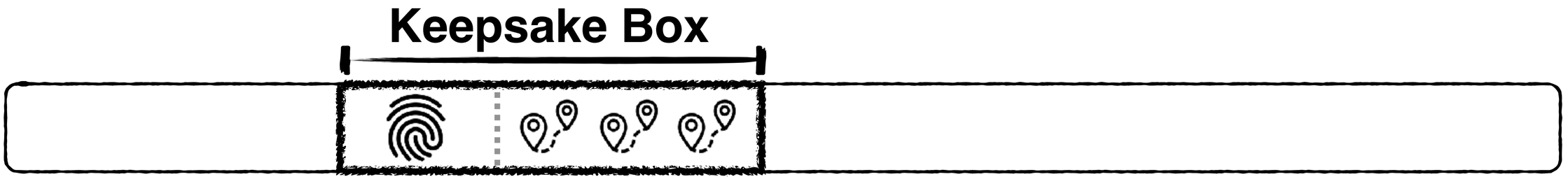

FP(2)

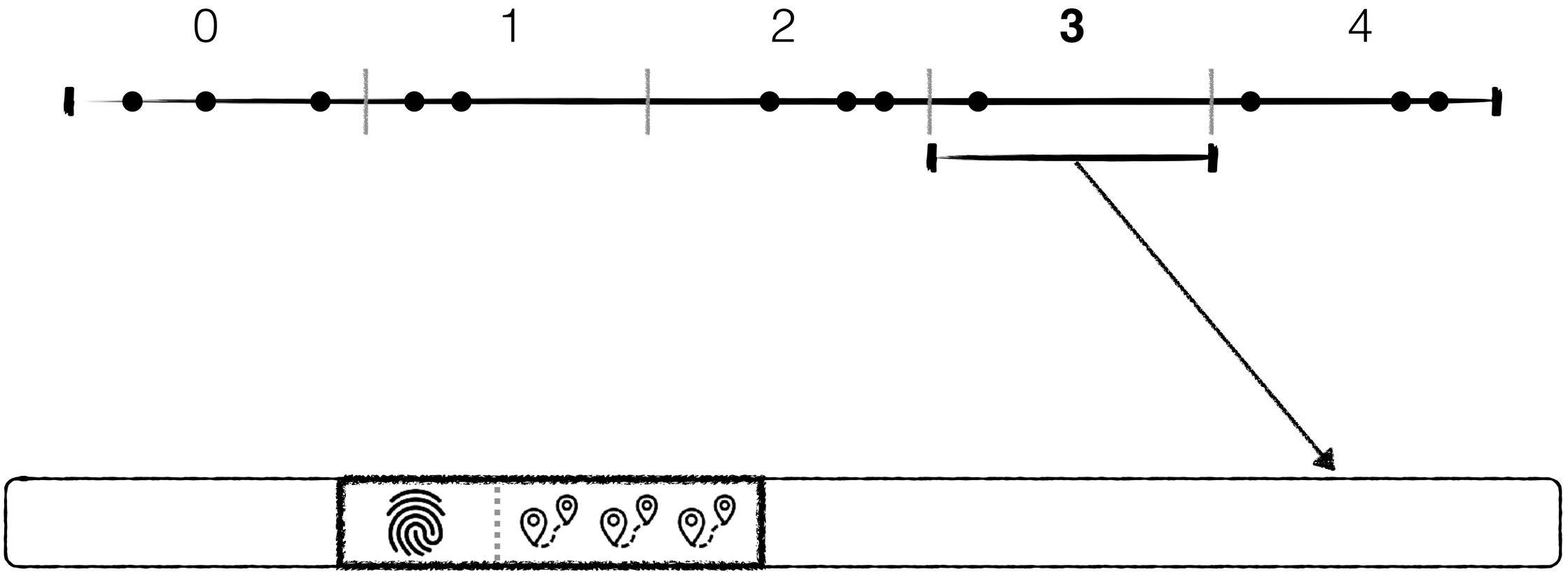

0
1
2
3
4

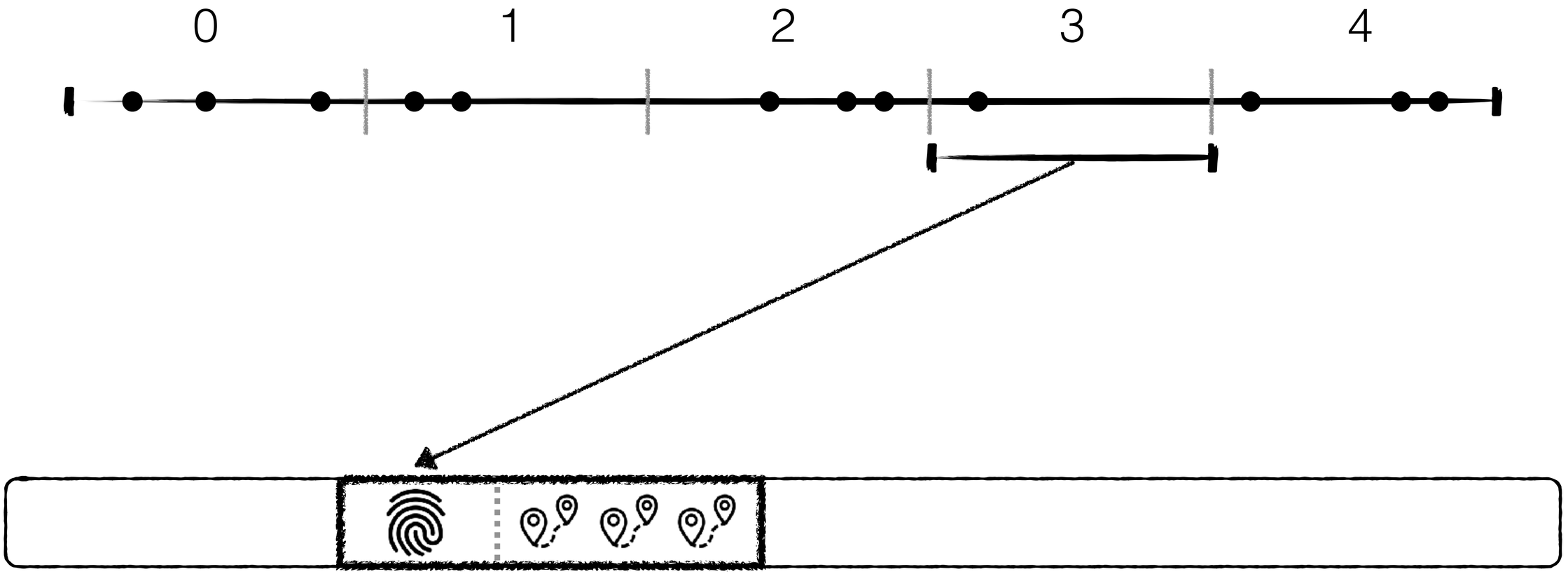

FP(2)
4, 6, 7

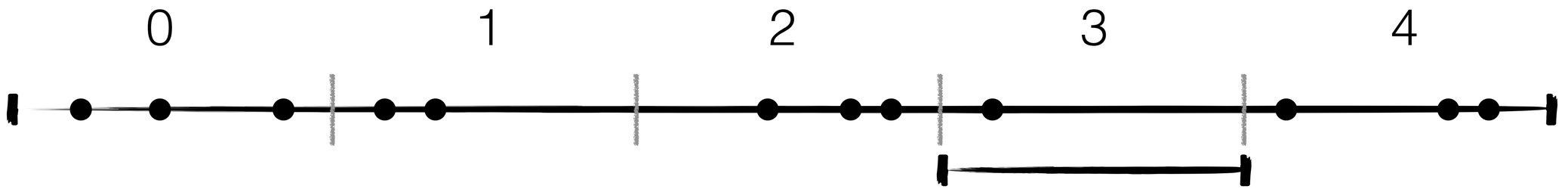

0
1
2
3
4

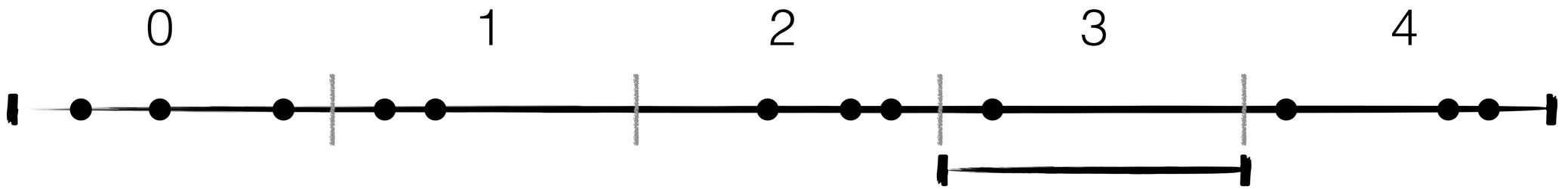

0
1
2
3
4

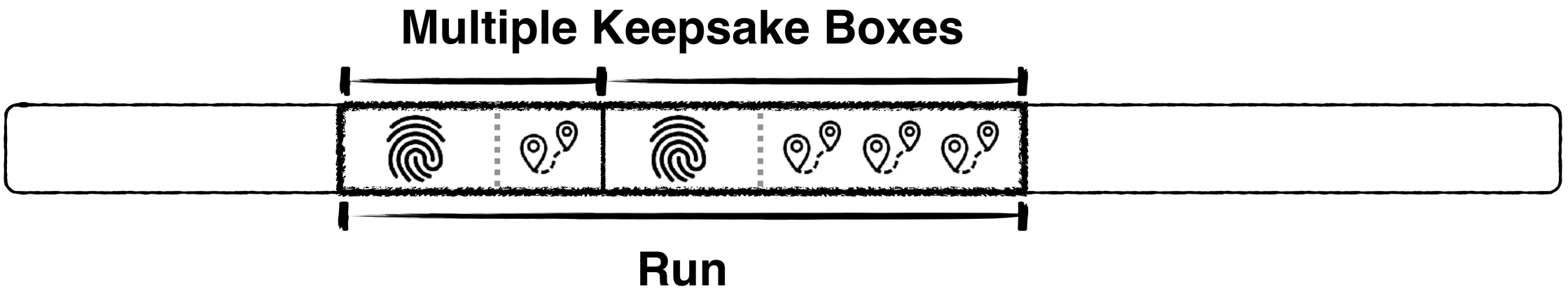

Keepsake Box

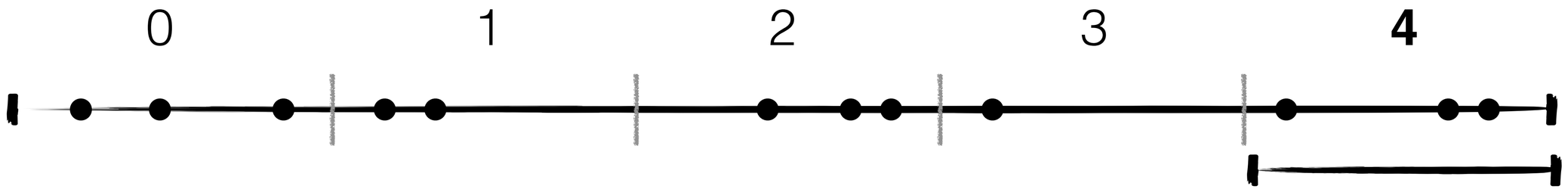

0
1
2
3
4

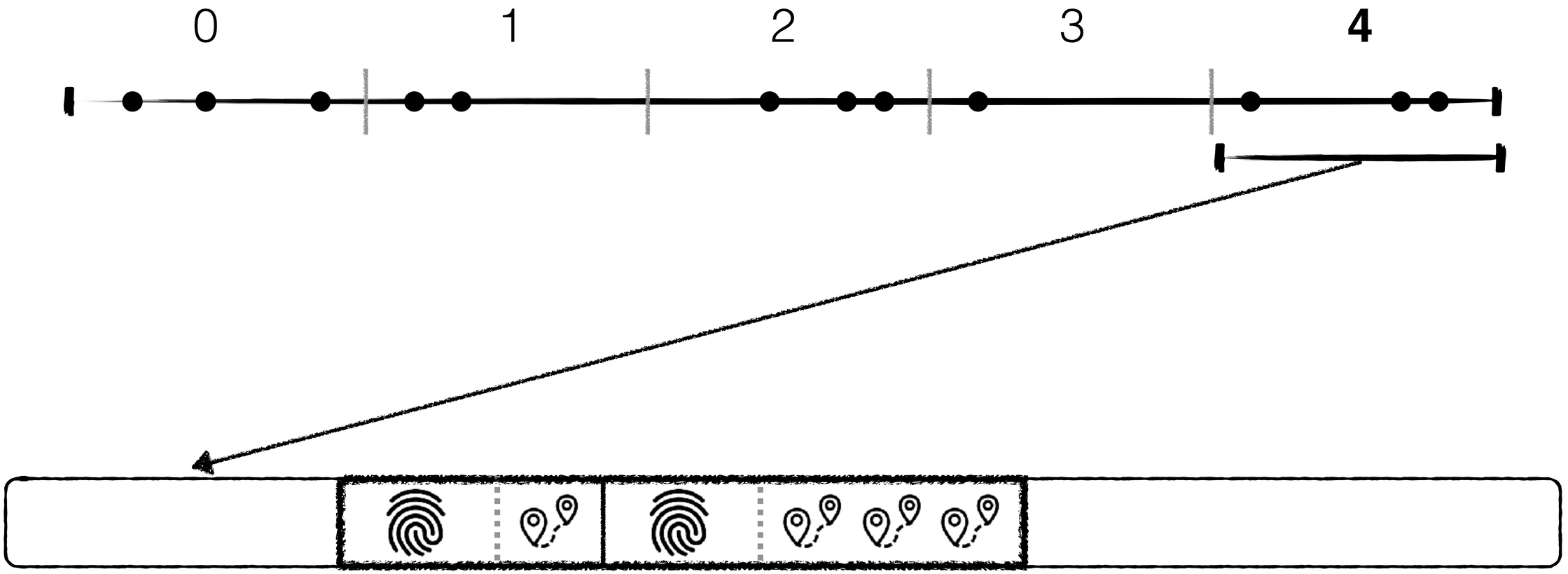

0
1
2
3
4

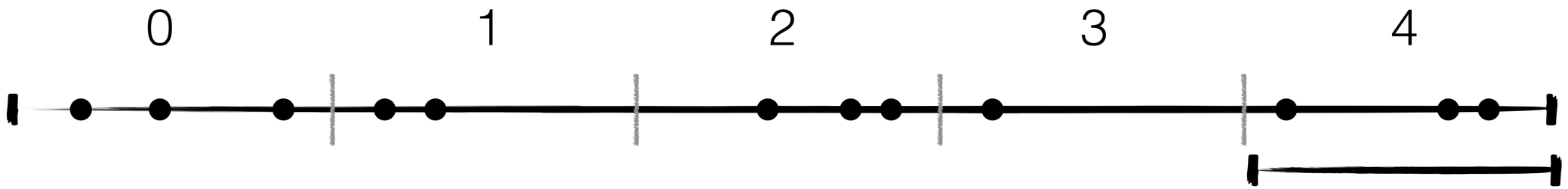

0
1
2
3
4

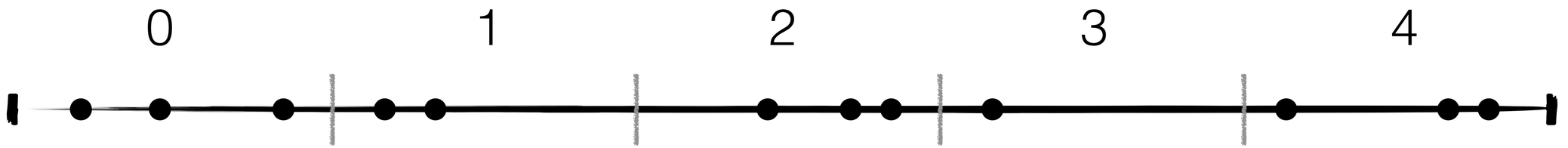

0
1
2
3
4

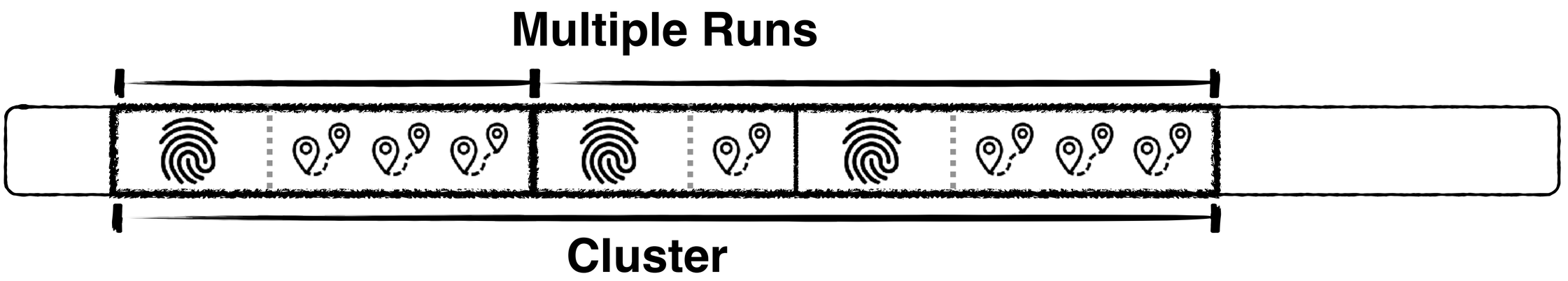

Multiple Keepsake Boxes
Run


0
1
2
3
4

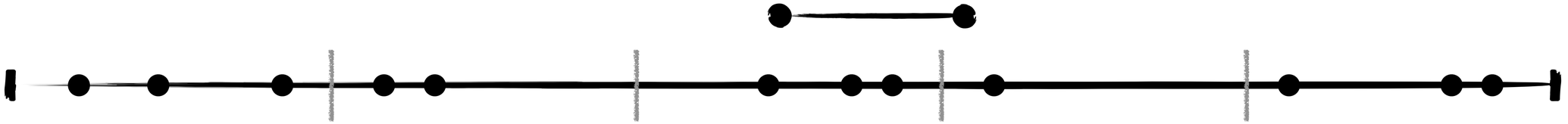

0
1
2
3
4


0
1
2
3
4

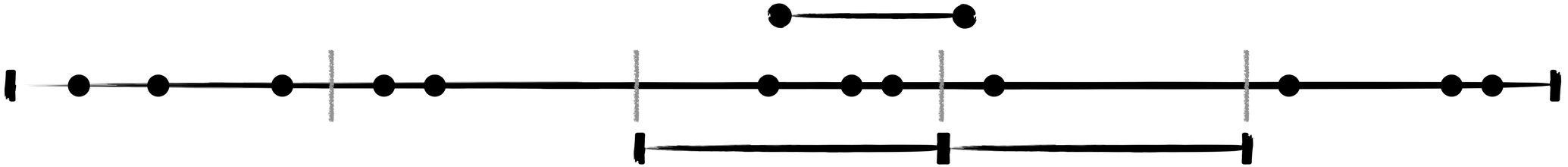

0
1
2
3
4

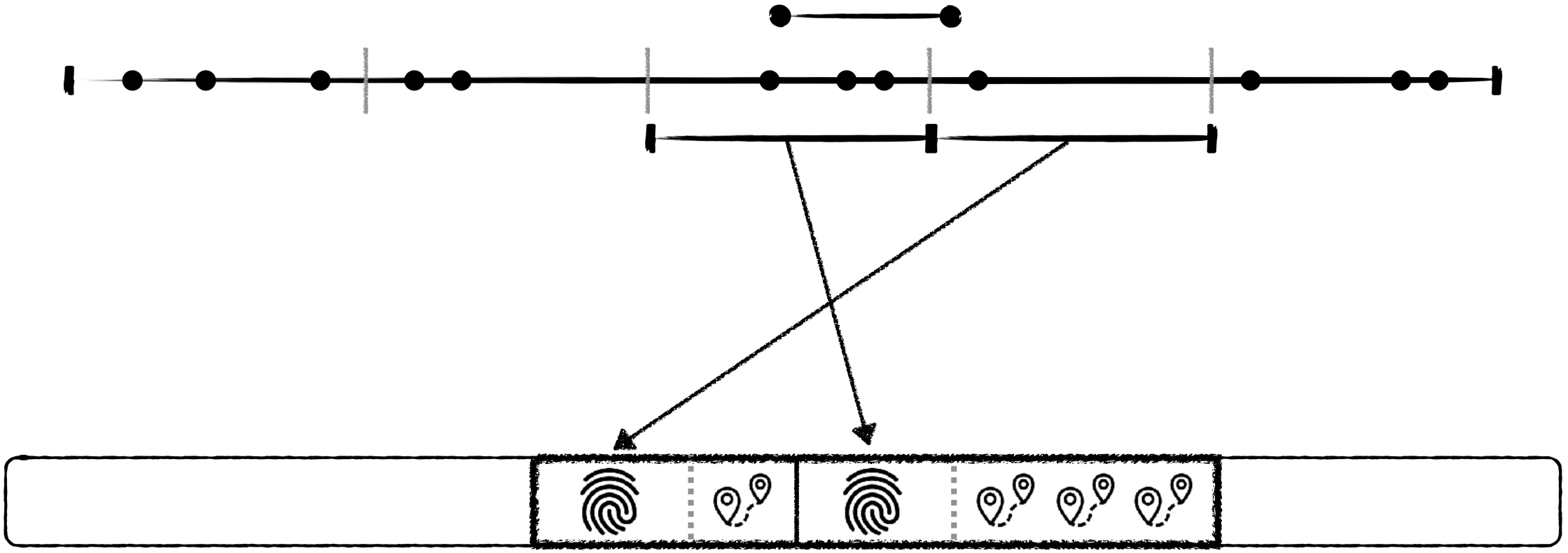

Multiple Runs
Cluster

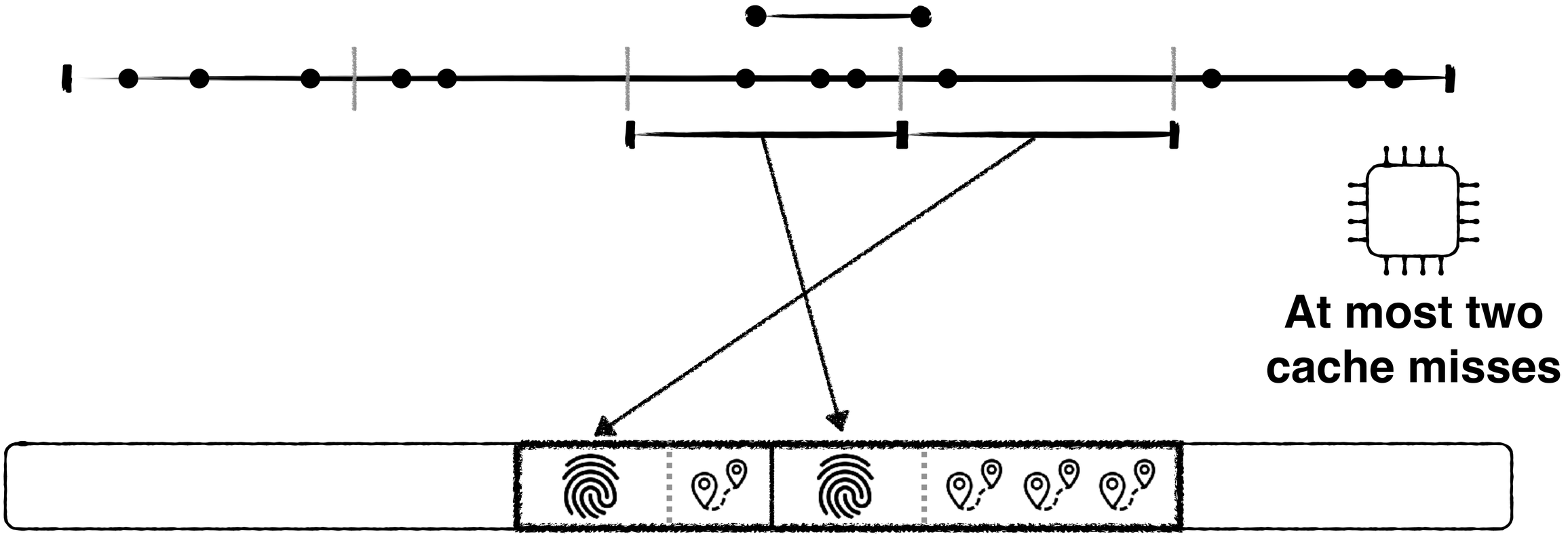
Queries

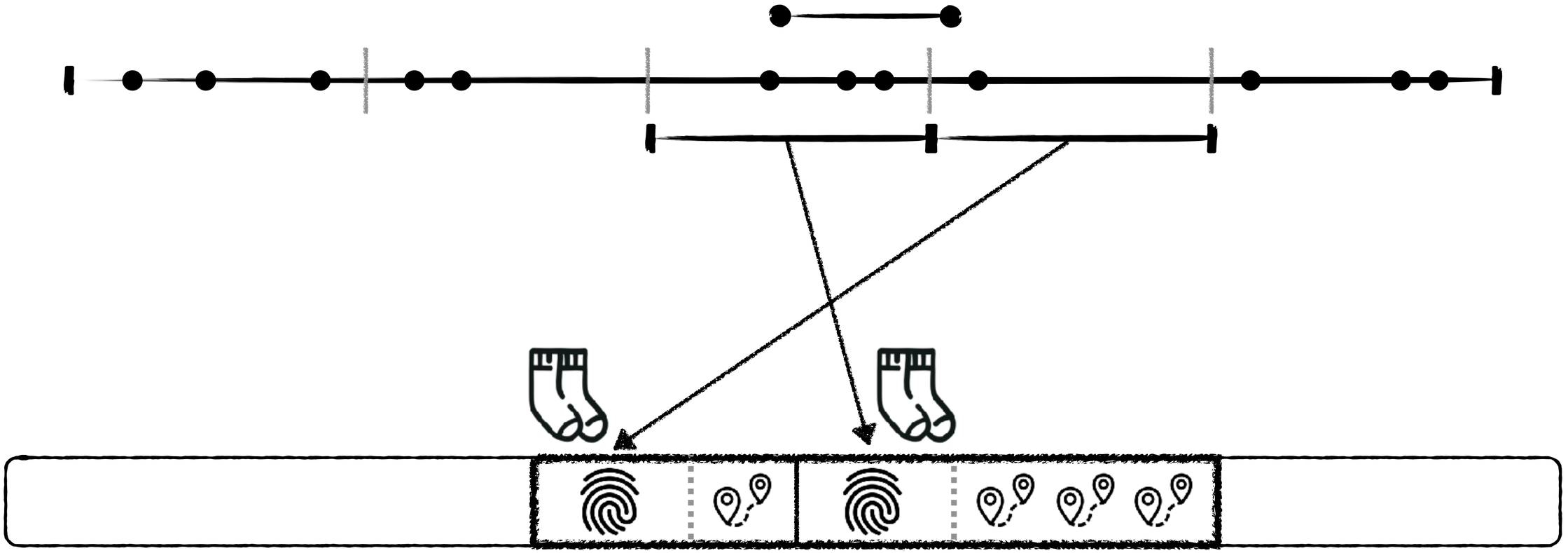


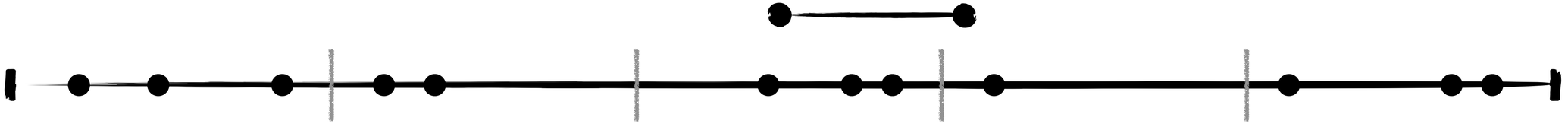
Queries

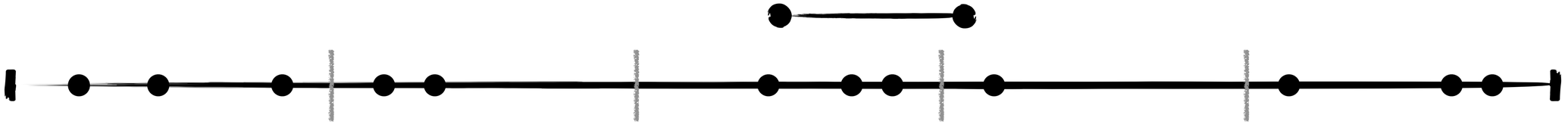




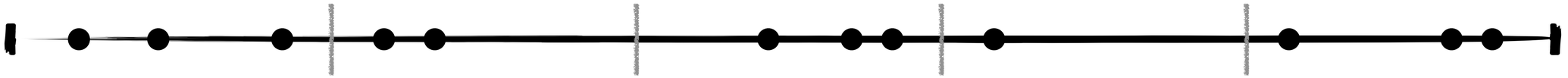

At most two
cache misses



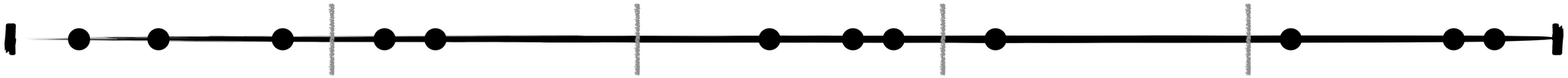




False positives due to fingerprint collisions



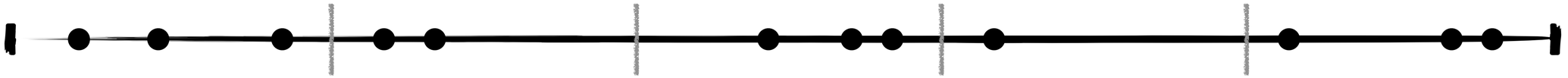
$$FPR \leq 2 \cdot 2^{-|\text{fingerprint}|}$$



$$FPR \leq 2 \cdot 2^{-|\text{fingerprint}|}$$

↓

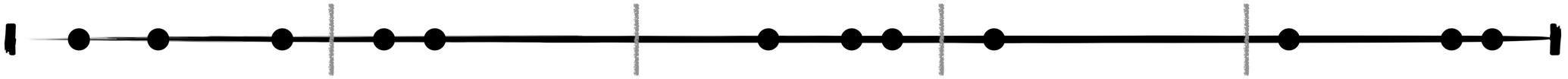
Collision probability



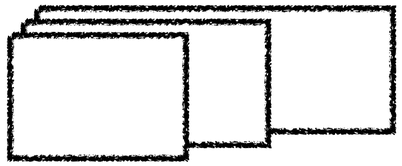
$$FPR \leq 2 \cdot 2^{-|\text{fingerprint}|}$$

↓

Two checks

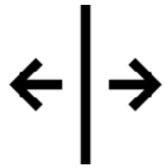


Workload independent



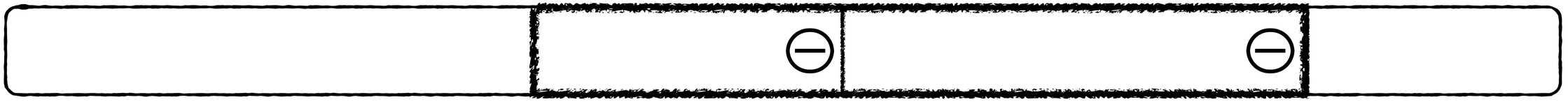
Var-len keepsake boxes in a run

Var-len keepsake boxes in a run



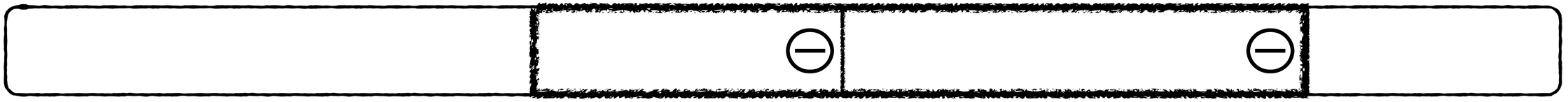
Delimit to decode unambiguously

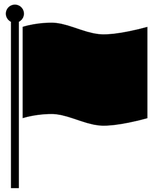
Simple solution: special delimiter character



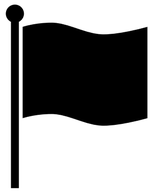
Simple solution: special delimiter character

Expensive

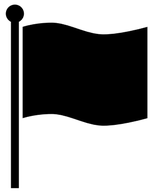




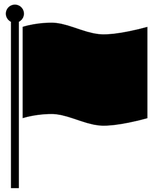
Self-delimiting encoding: minimize metadata overhead



Self-delimiting encoding:
minimize metadata overhead



Self-delimiting encoding:
minimize metadata overhead

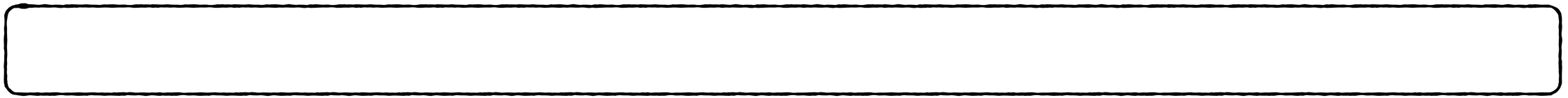


Self-delimiting encoding:
minimize metadata overhead

Removes need for delimiters!

Increasing fingerprint order

Slot structure

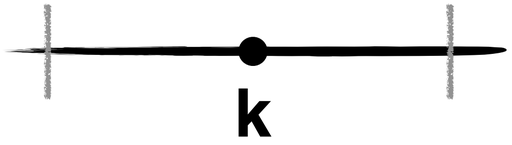


Slot

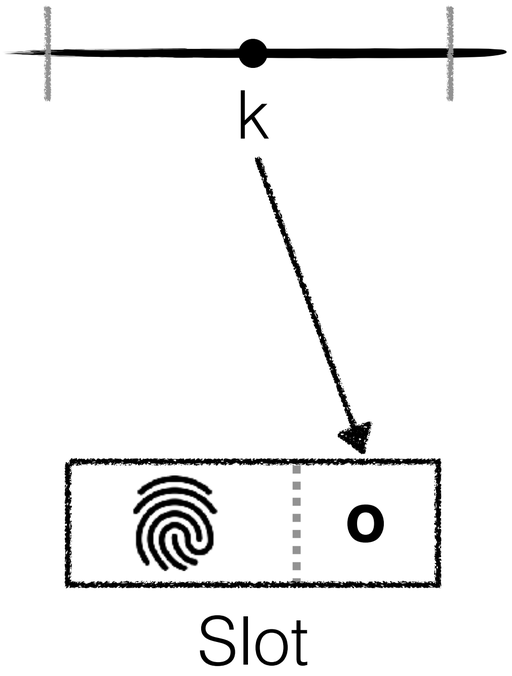
Slot

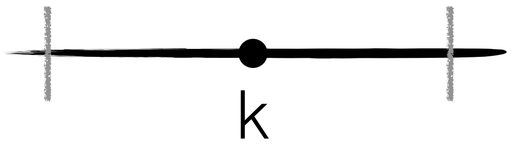
Encoding keepsake boxes

Slot

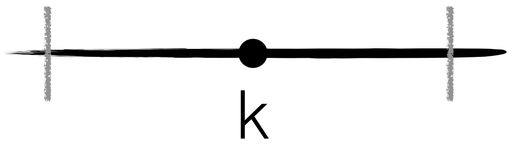


Slot

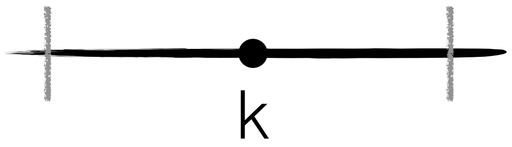

k
o
Slot



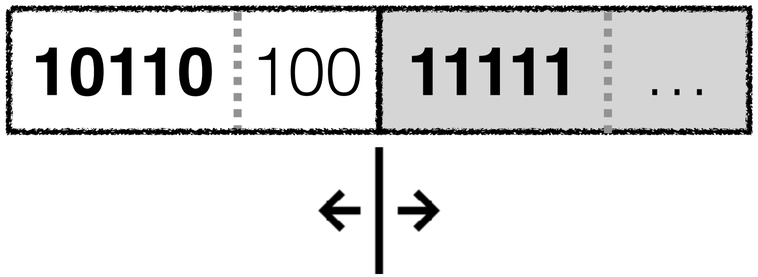
Example **10110** 100

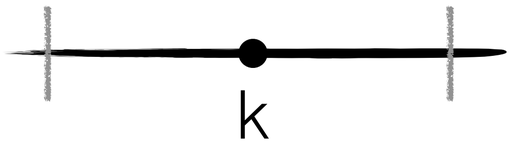


Example



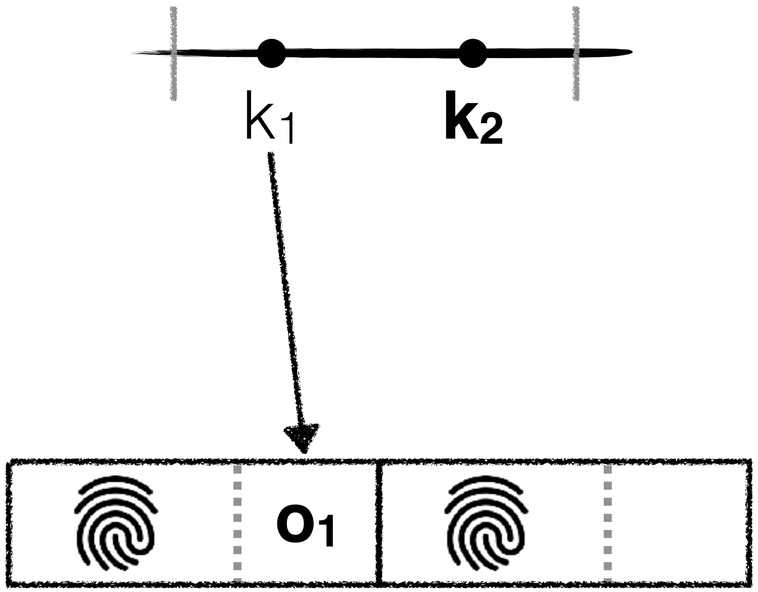
Example

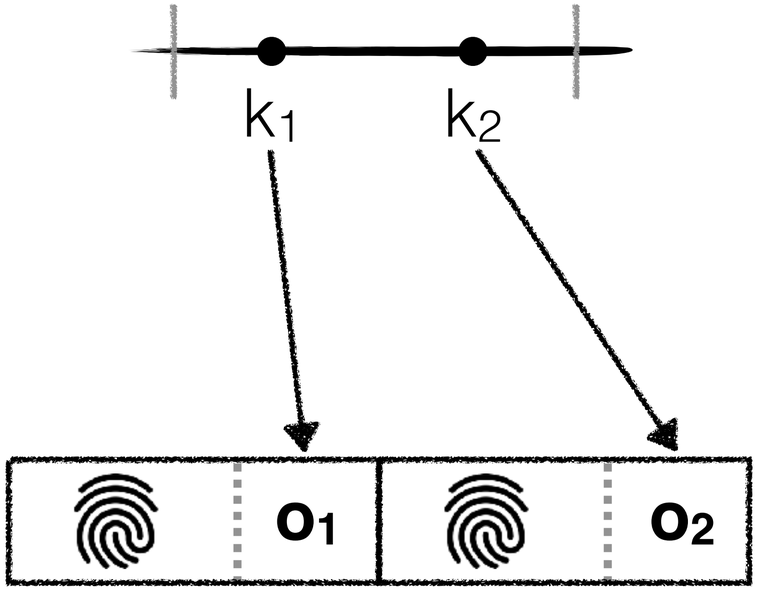



k

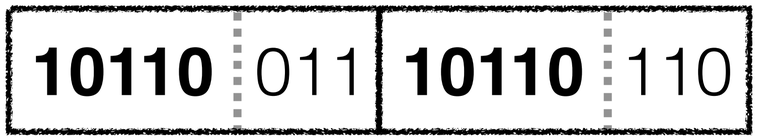

k_1
k_2


k_1
k_2


k1
k2
o1

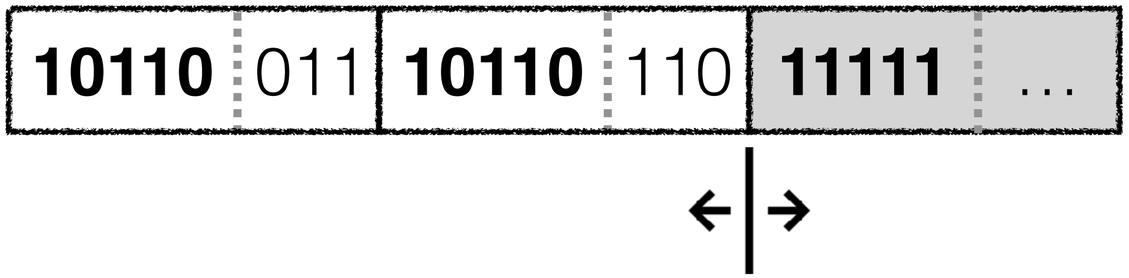

k1
k2
o1
o2

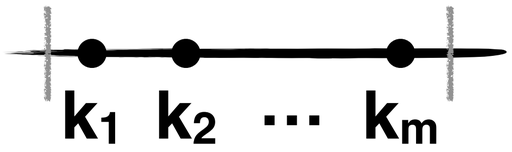
Example

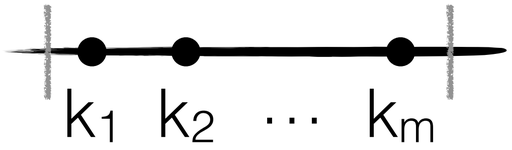


Example

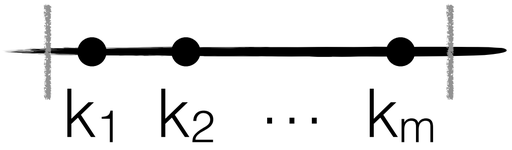
Example




k_1 k_2 ⋯ k_m

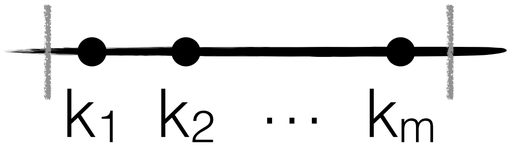


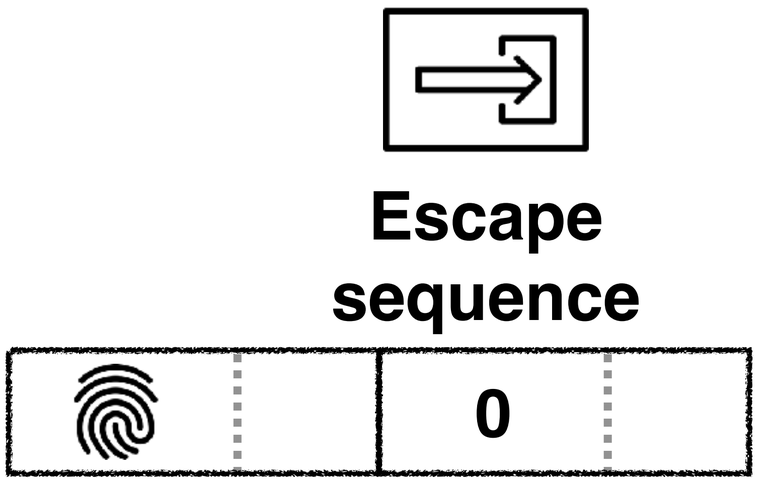
Cut down on encoding length

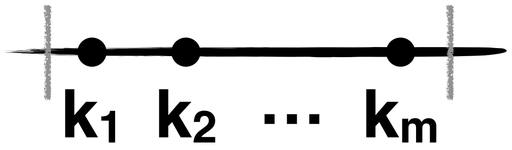


Cut down on encoding length

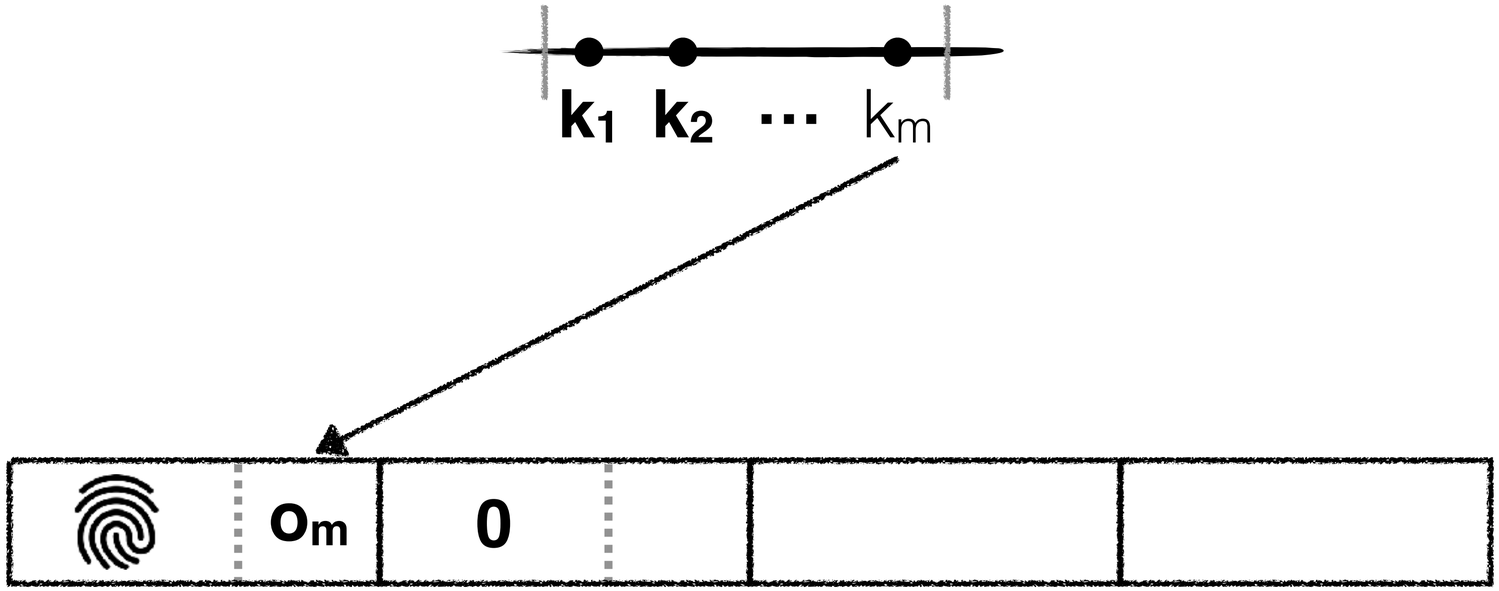
No duplicates

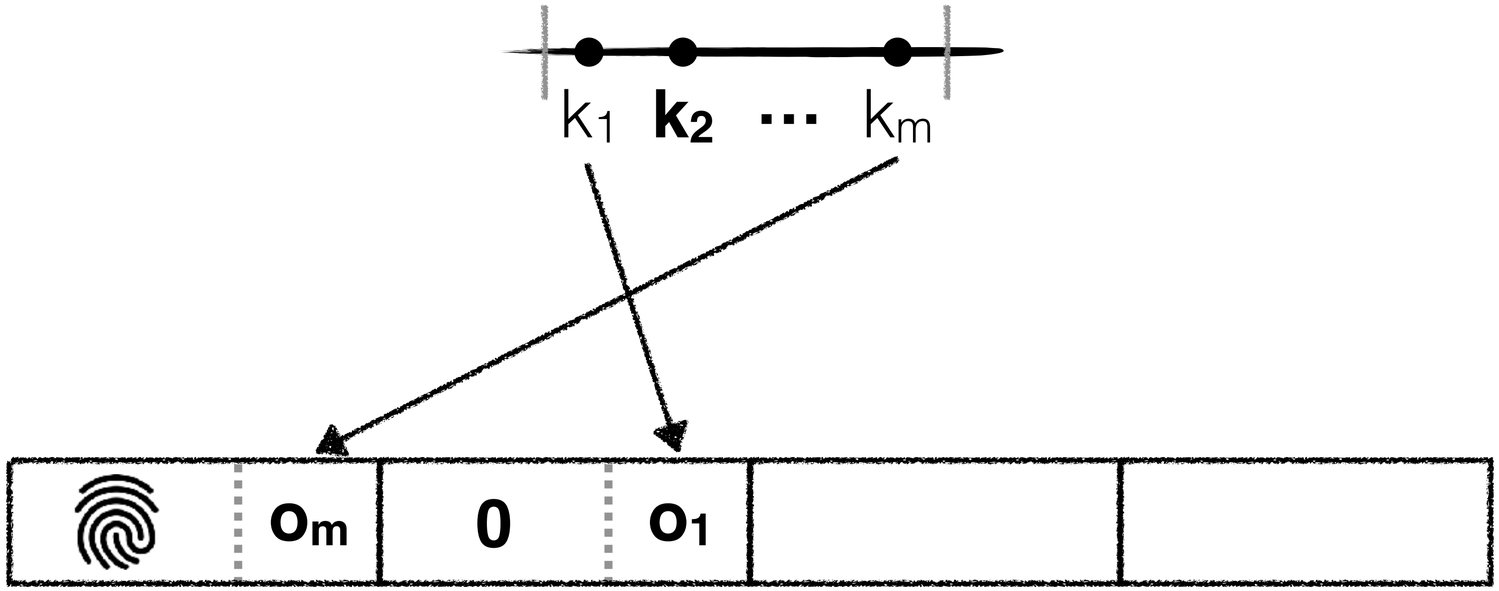

k_1 k_2 ⋯ k_m

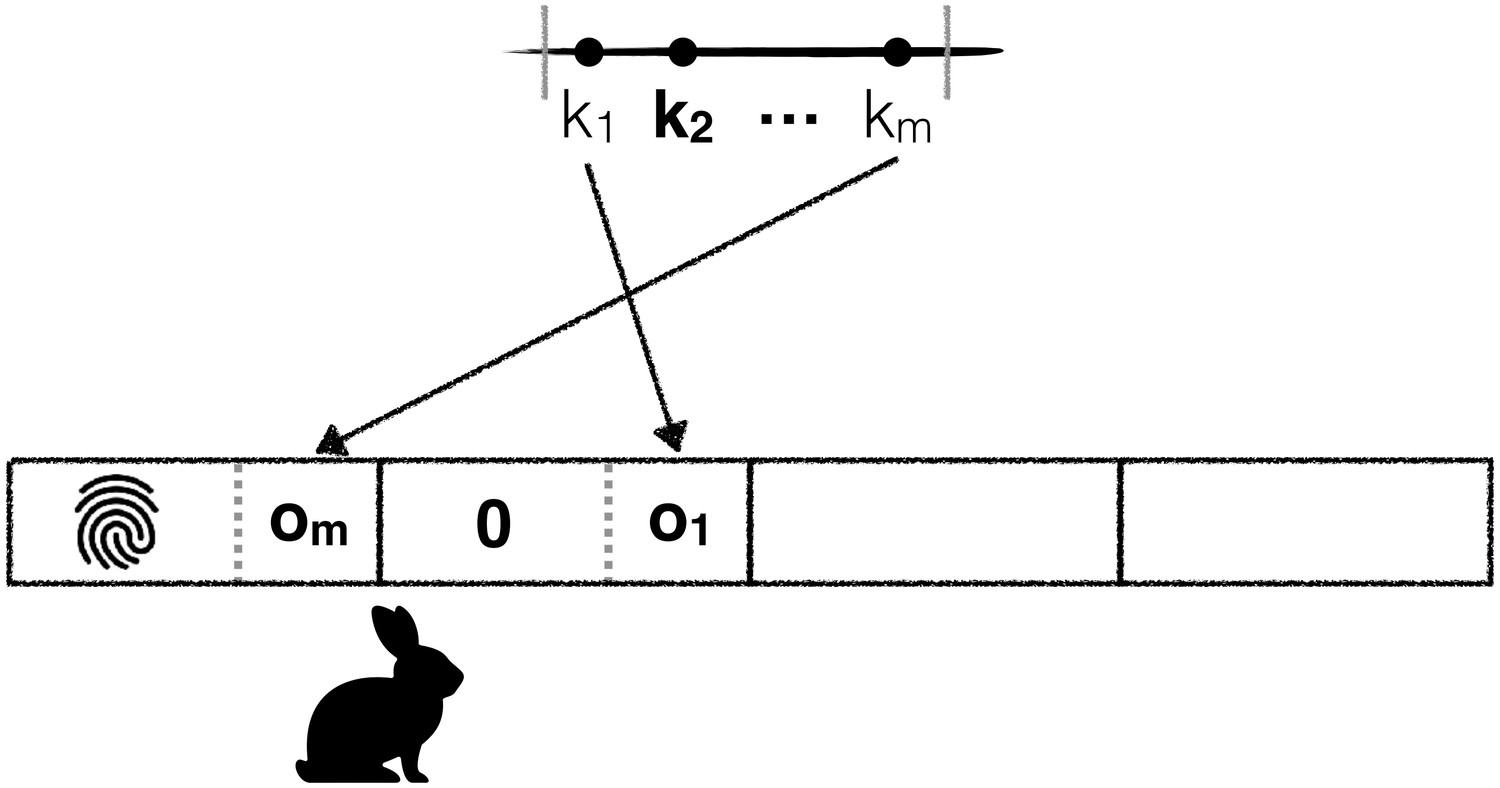

Escape sequence
0


k_1 k_2 ⋯ k_m

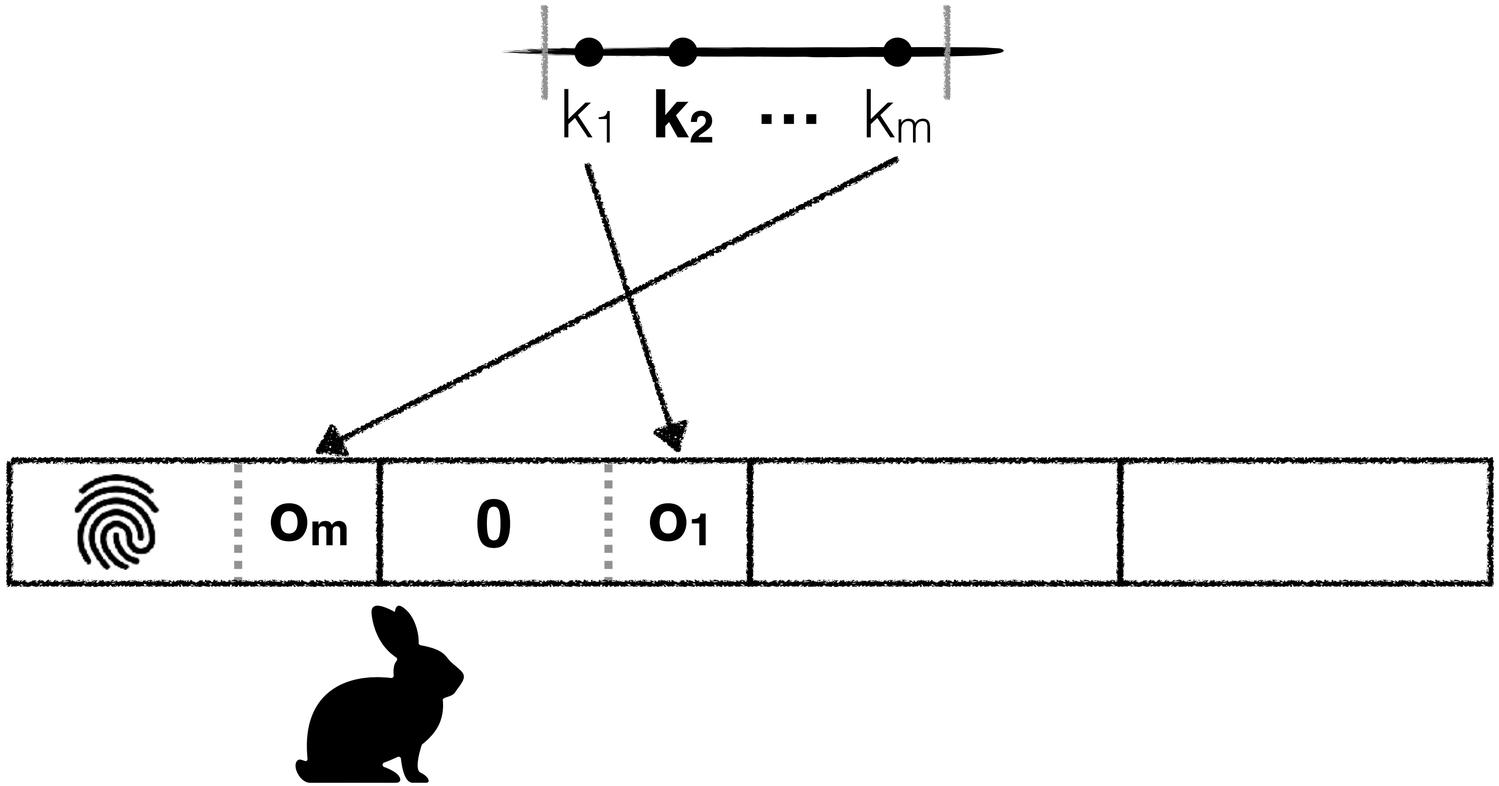

0


k1 k2 ··· km
om
0

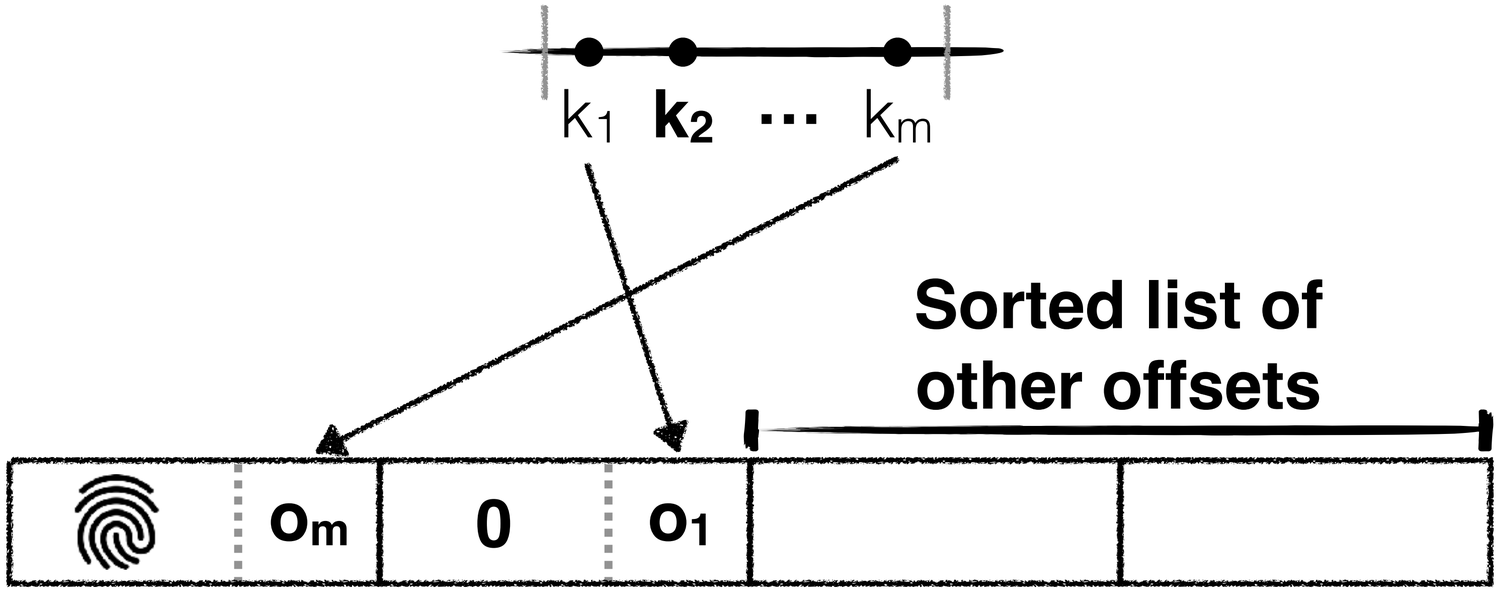

k1 k2 ··· km
om
0
o1

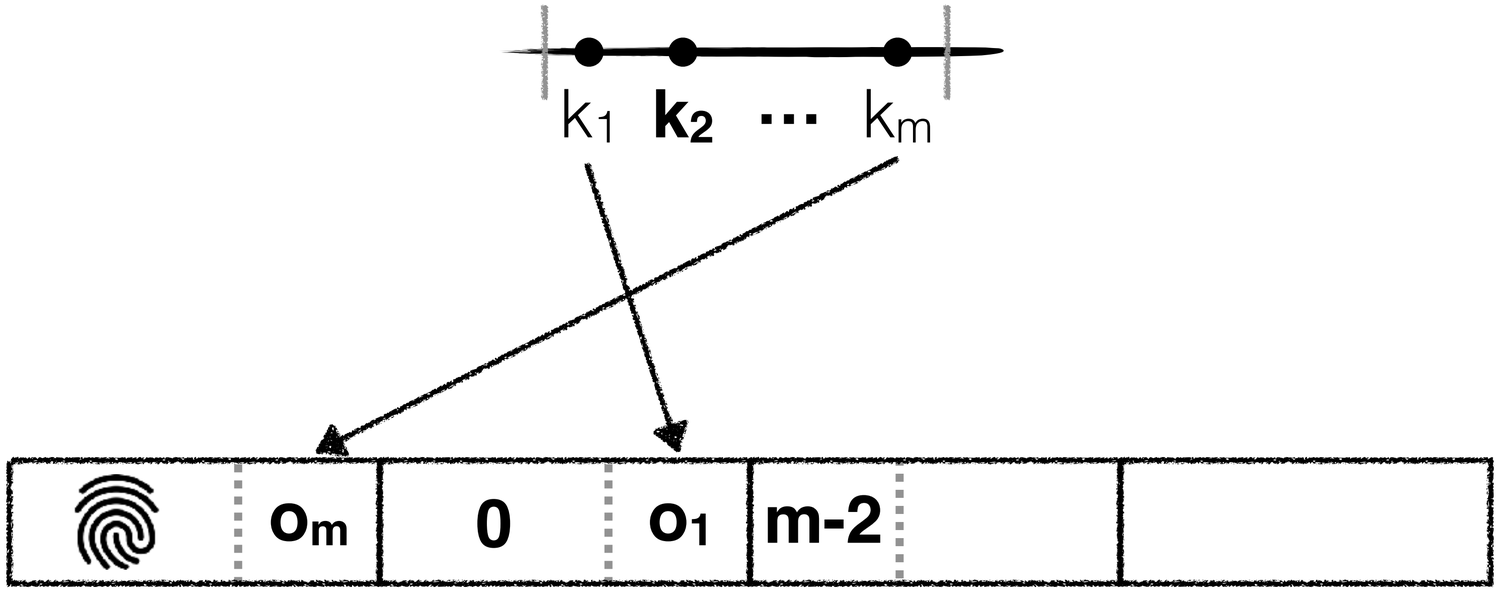


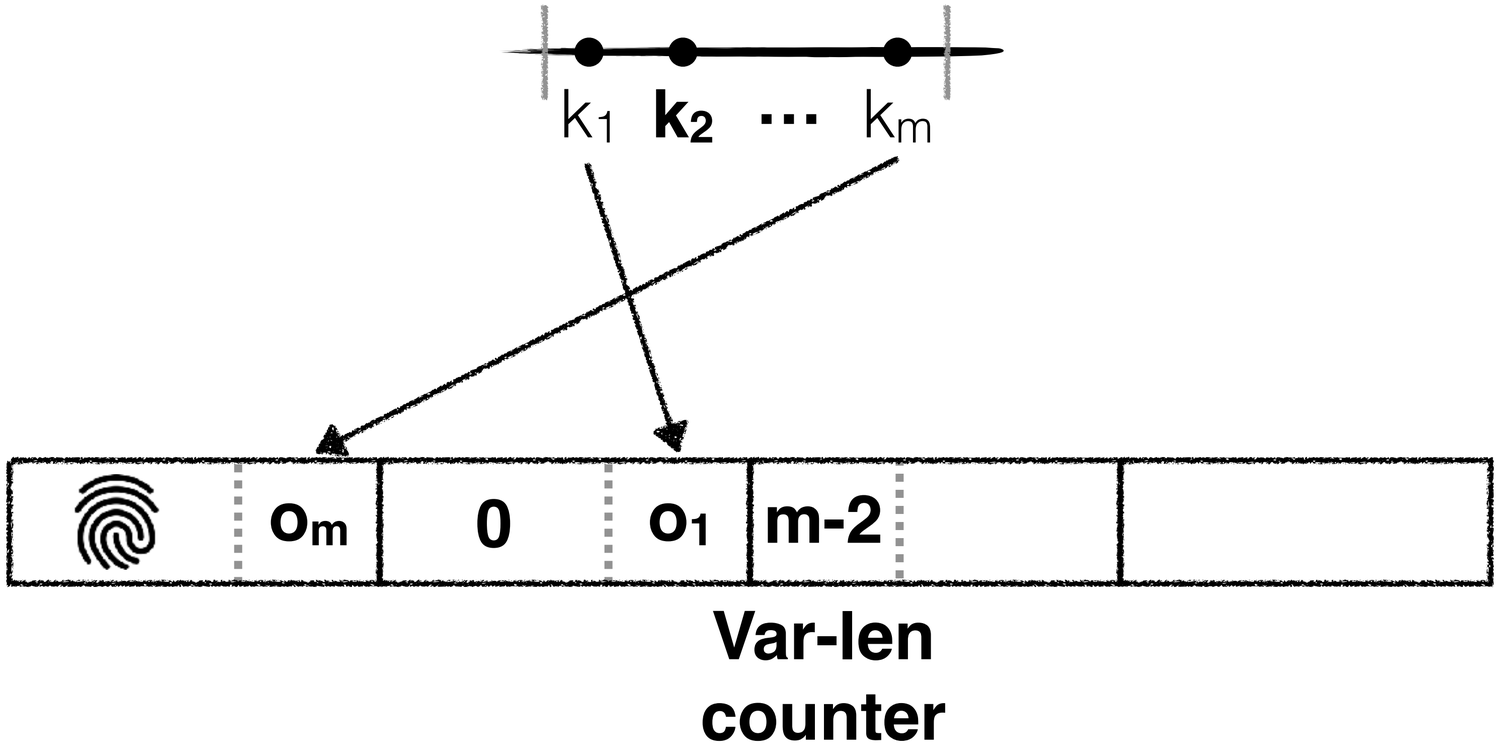
Fast access to max and min

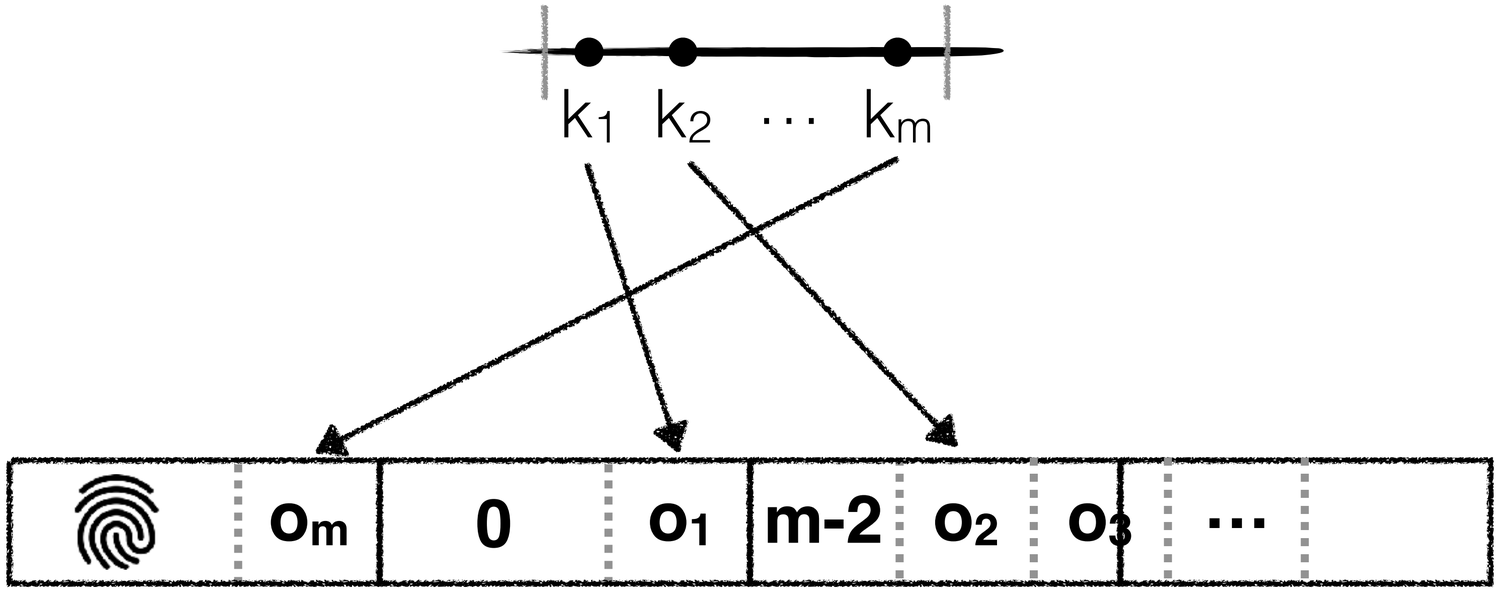


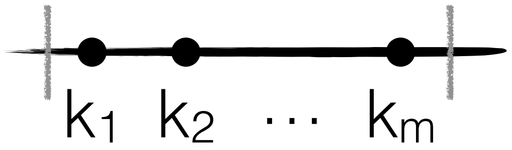
Speed up queries

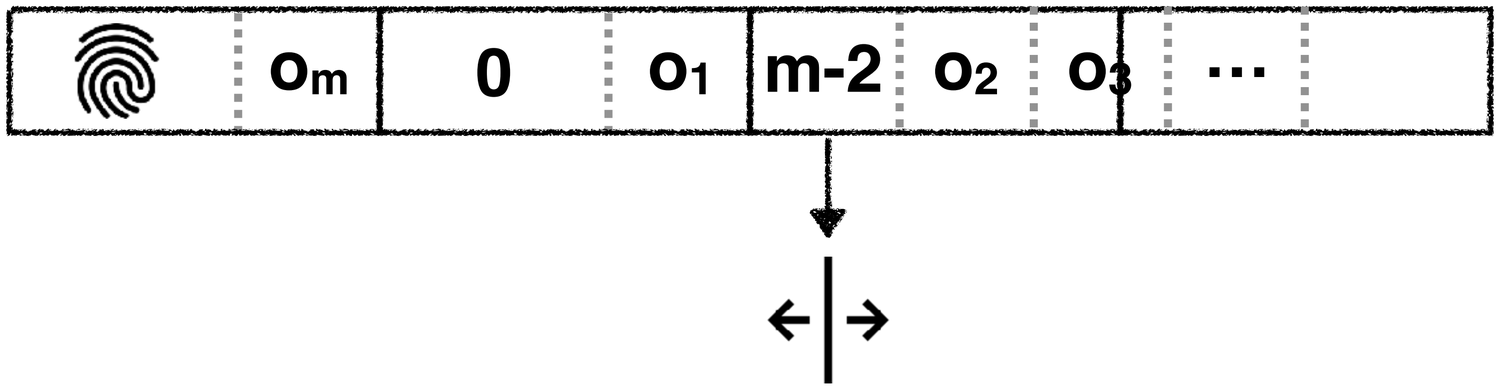

k1 k2 ··· km
Sorted list of other offsets
om
0
o1


k1 k2 ··· km
om
0
o1
m-2

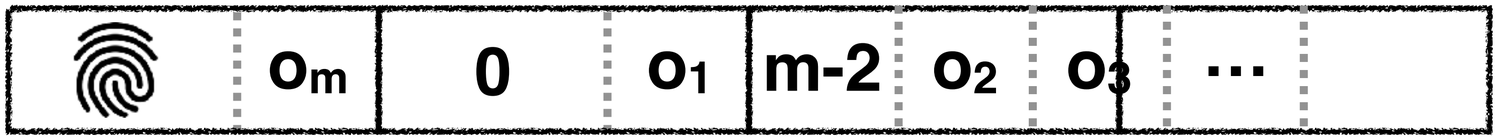

k_1 $\mathbf{k_2}$ ⋯ k_m
o_m
0
o_1
m-2
Var-len
counter


k_1 k_2 ⋯ k_m
o_m
0
o_1
m-2
o_2
o_3
⋯

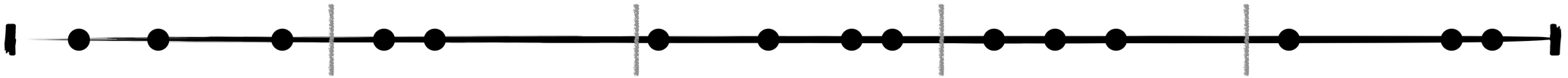

k_1 k_2 ⋯ k_m


o_m
0
o_1
m-2
o_2
o_3
⋯

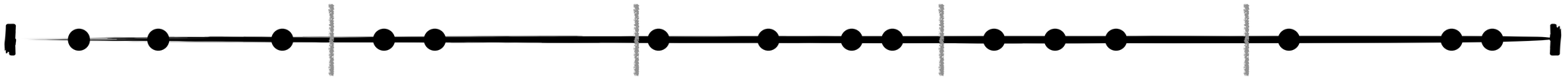
Maintain encoding with updates



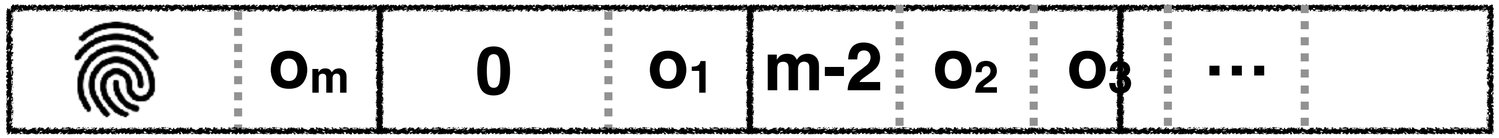
Range queries: revisited



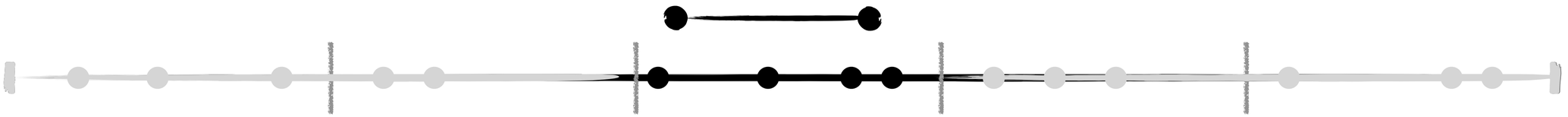
Range queries: revisited

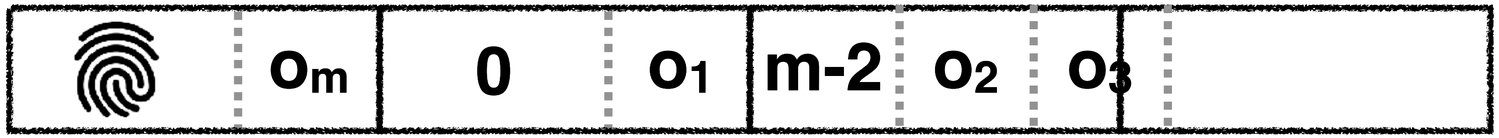


Focus on the case of long keepsake boxes

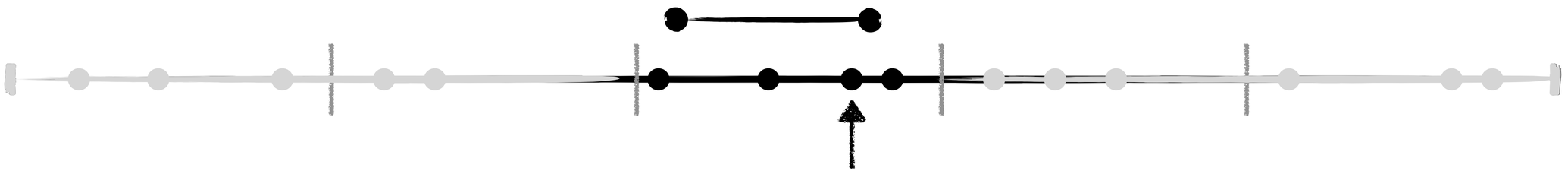


Case 1: single partition

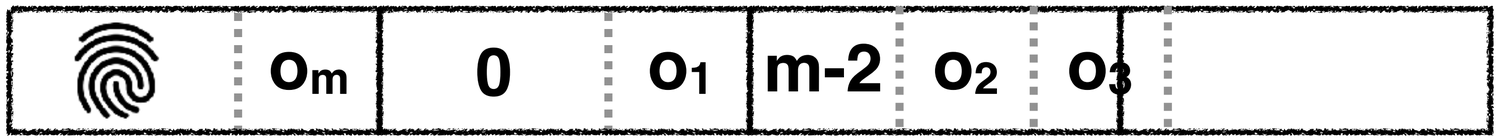


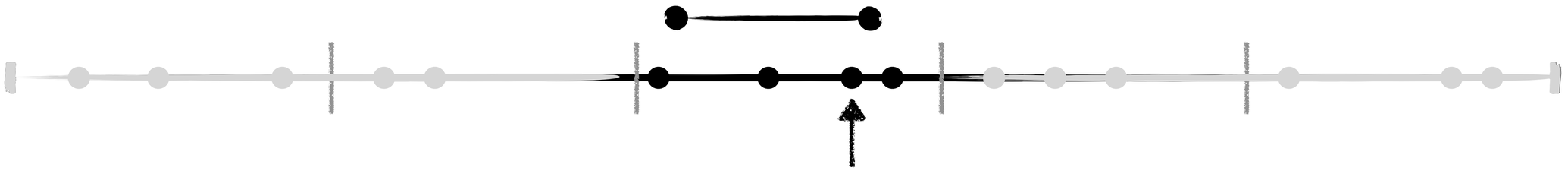


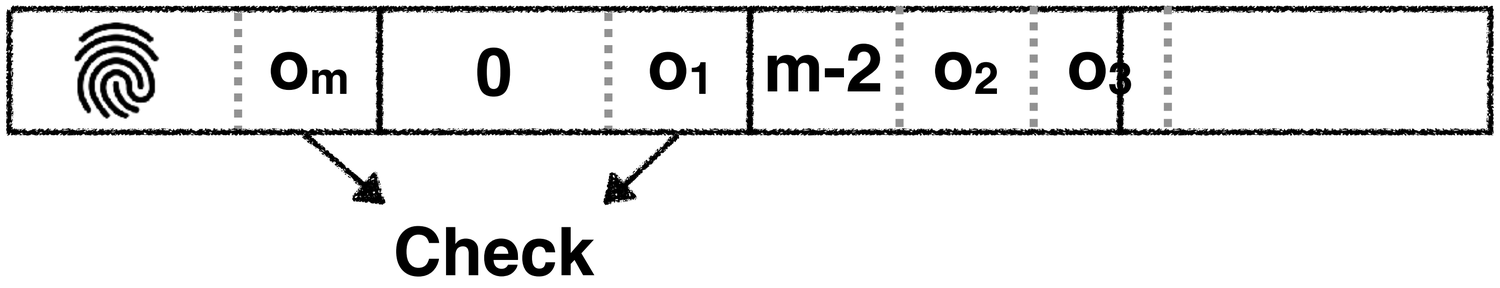
Case 1: single partition

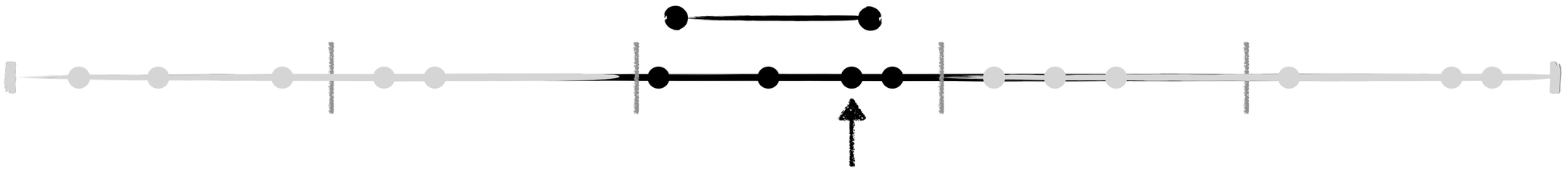


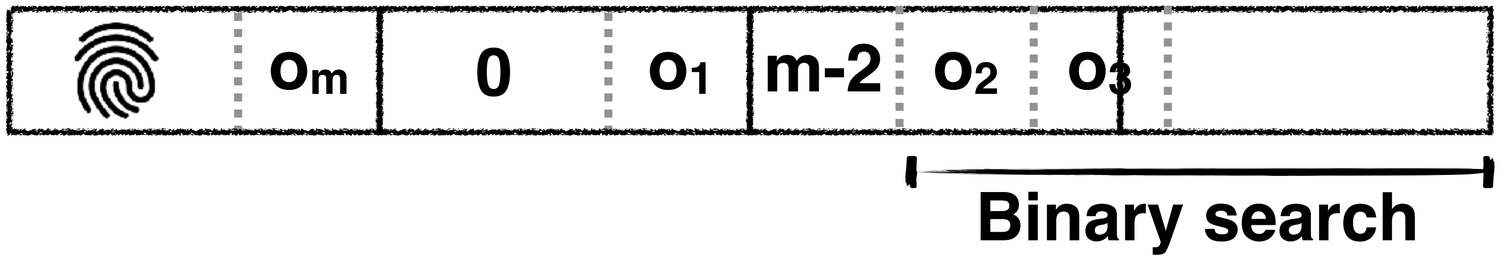
Predecessor of right endpoint



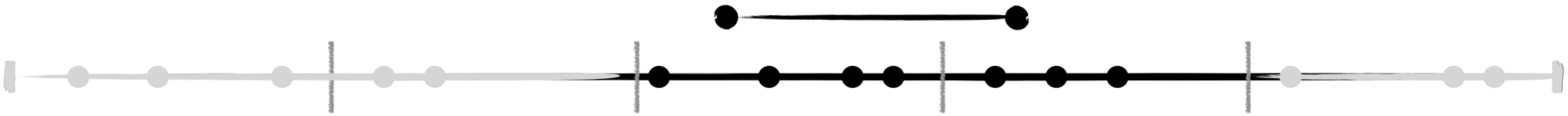

Predecessor of
right endpoint

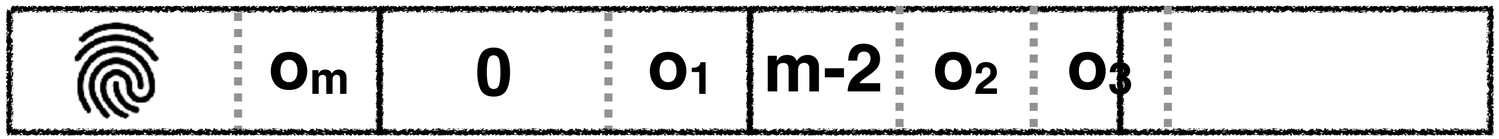

o_m
0
o_1
m-2
o_2
o_3
Check


Predecessor of
right endpoint

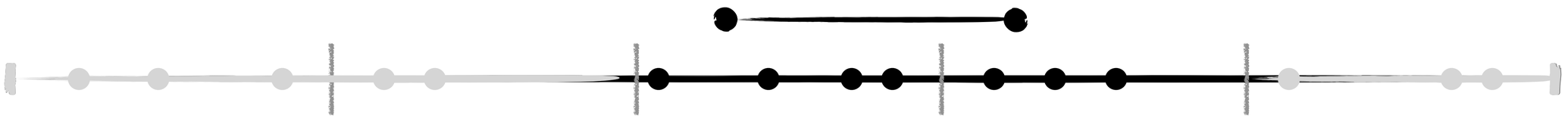

o_m
0
o_1
m-2
o_2
o_3
Binary search

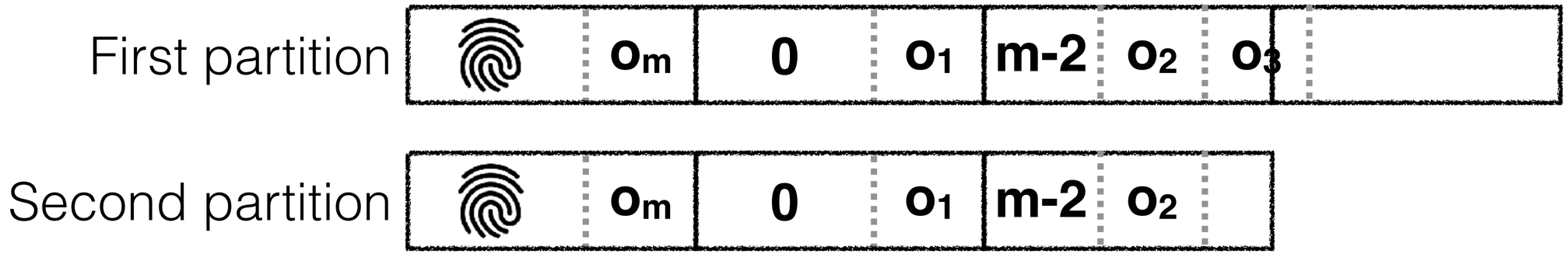
Case 2: two partitions

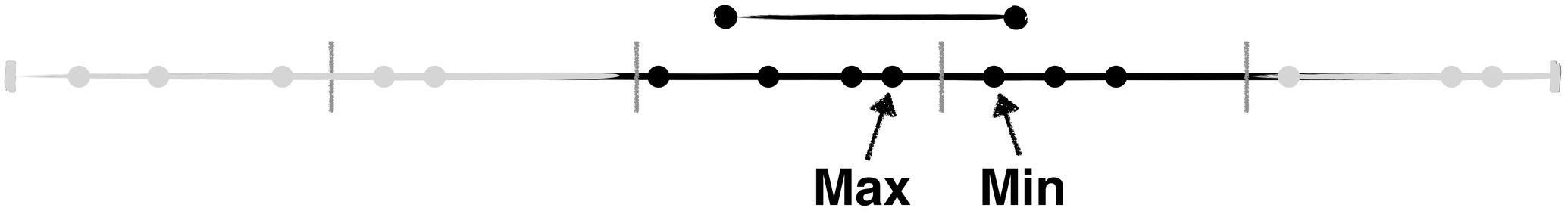


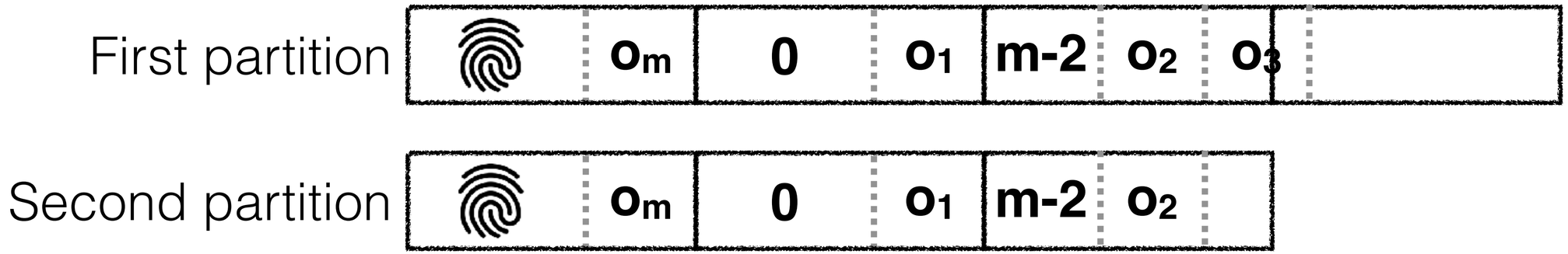


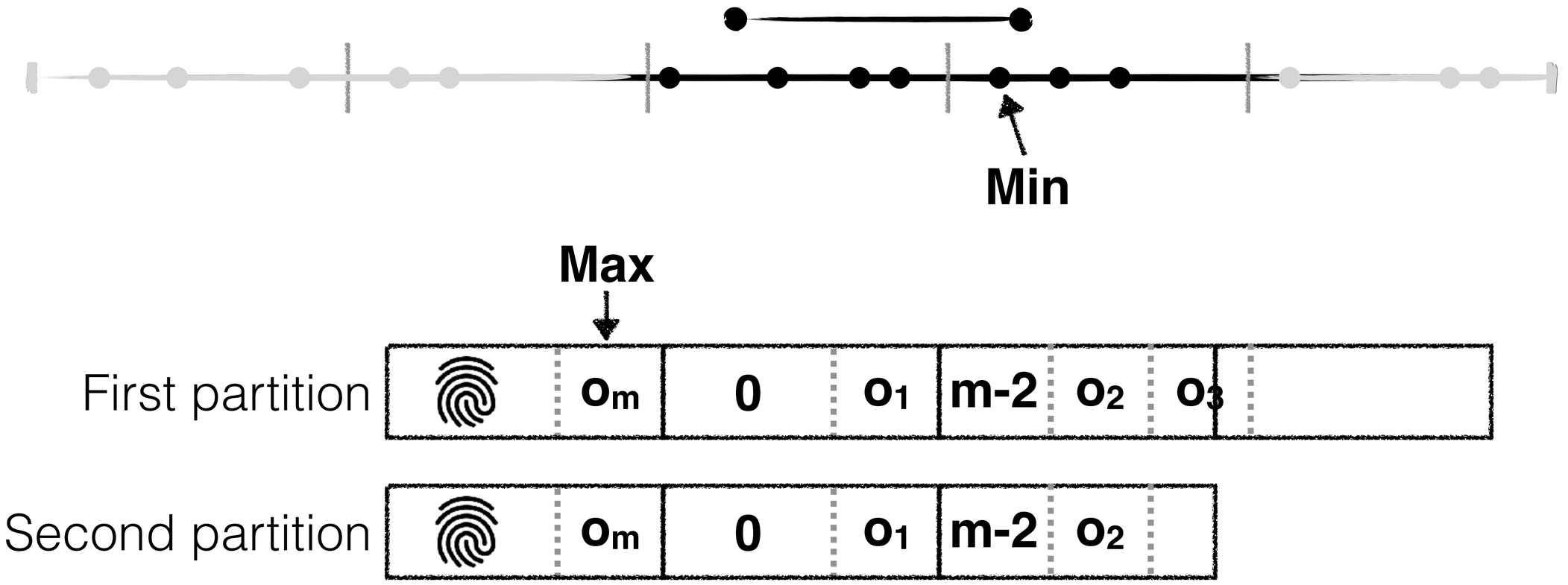
Case 2: two partitions

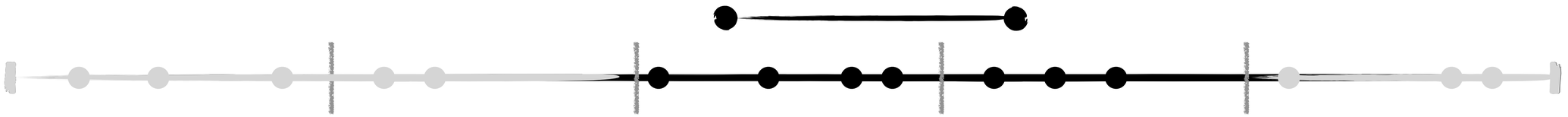


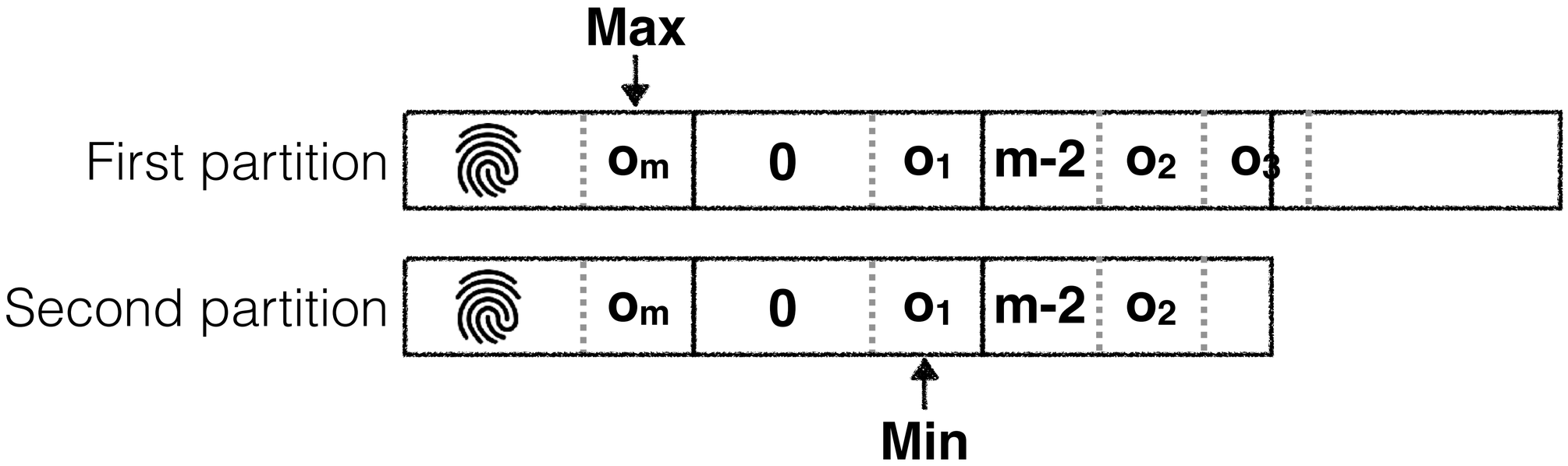



Max
Min


First partition
o_m
0
o_1
m-2
o_2
o_3
Second partition
o_m
0
o_1
m-2
o_2

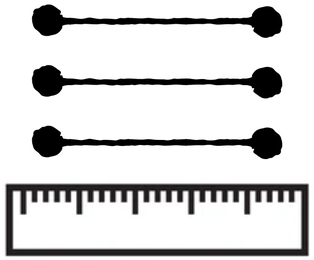

Min
Max
First partition
o_m
0
o_1
m-2
o_2
o_3
Second partition
o_m
0
o_1
m-2
o_2



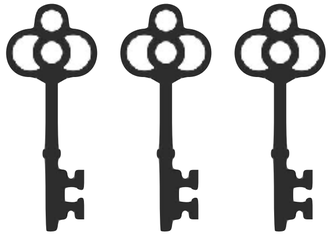

Max
First partition
o_m
0
o_1
m-2
o_2
o_3
Second partition
o_m
0
o_1
m-2
o_2
Min

Problems?

Problems?



Bounded queries



Fixed-length keys

Can we design a robust and dynamic range filter that resolves these issues?

Can we design a robust and dynamic range filter that resolves these issues?

No :(

Theorem [GGLP'14]: Robust range filters need at least $\log_2(R / \varepsilon)$ bits per key

Theorem [GGLP'14]: Robust range filters need at least $\log_2(R / \varepsilon)$ bits per key

↓

Max range length

Theorem [GGLP'14]: Robust range filters need at least $\log_2(R / \varepsilon)$ bits per key

↓

FPR

$$\log_2 (R / \varepsilon)$$

$$\log_2 R + \log_2 (1 / \varepsilon)$$

$\log_2 R + \log_2 (1 / \varepsilon)$

Memento Filter

$\log_2 R + \log_2 (1 / \varepsilon)$

Memento Filter

$\log_2 R$

Memento Filter

$\log_2 (1 / \varepsilon)$

$\log_2 R$

Memento Filter

$|\text{Offset}|$

$\log_2 (1 / \varepsilon)$

Memento Filter

|Offset|

$\log_2 (1 / \varepsilon)$

$\log_2 R$

Memento Filter

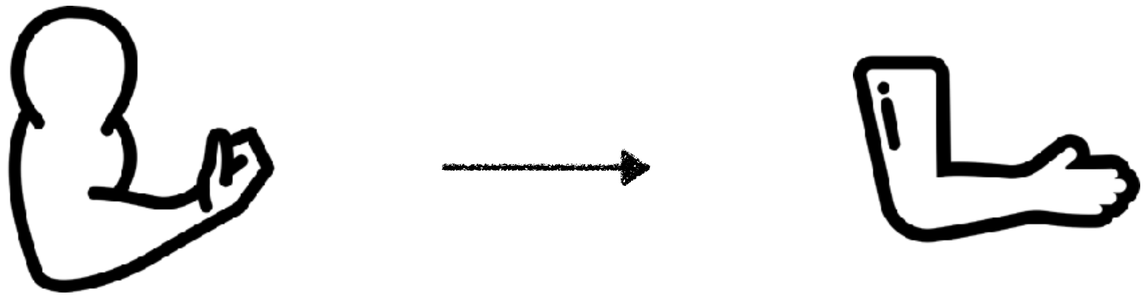
|Offset|

$\log_2 (1 / \varepsilon)$

$\log_2 R$

Any way out?

Robustness



Robustness **Semi-Robustness**

Semi-Robustness

Guarantee an FPR of ε for “common” datasets

Dynamicity, var-length

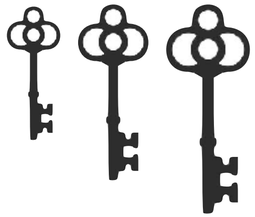
Diva

Diva
VLDB 2025 Best Paper

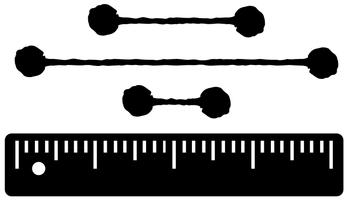
Diva

Semi-Robustness

Dynamicity

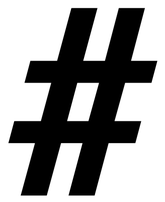


Var-length keys



Var-length queries

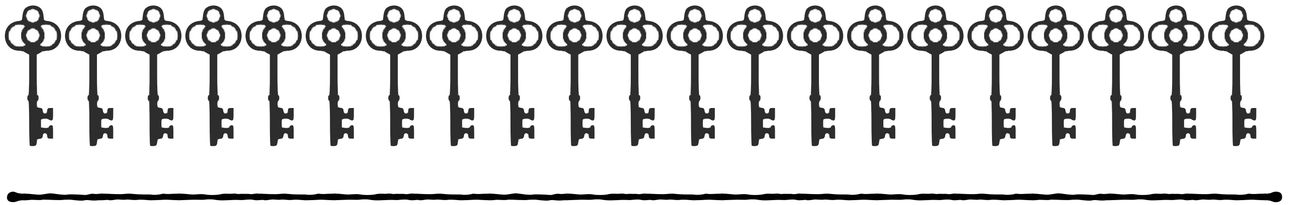
Speed



No hashing!

A…

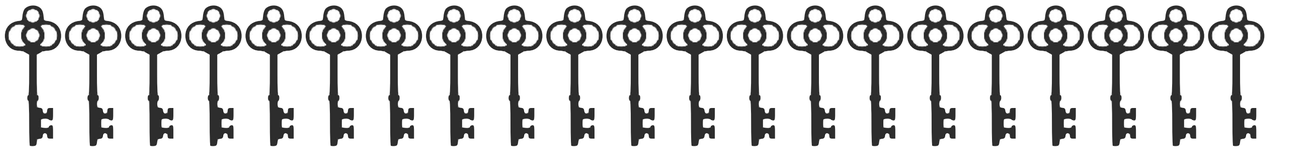
…Z



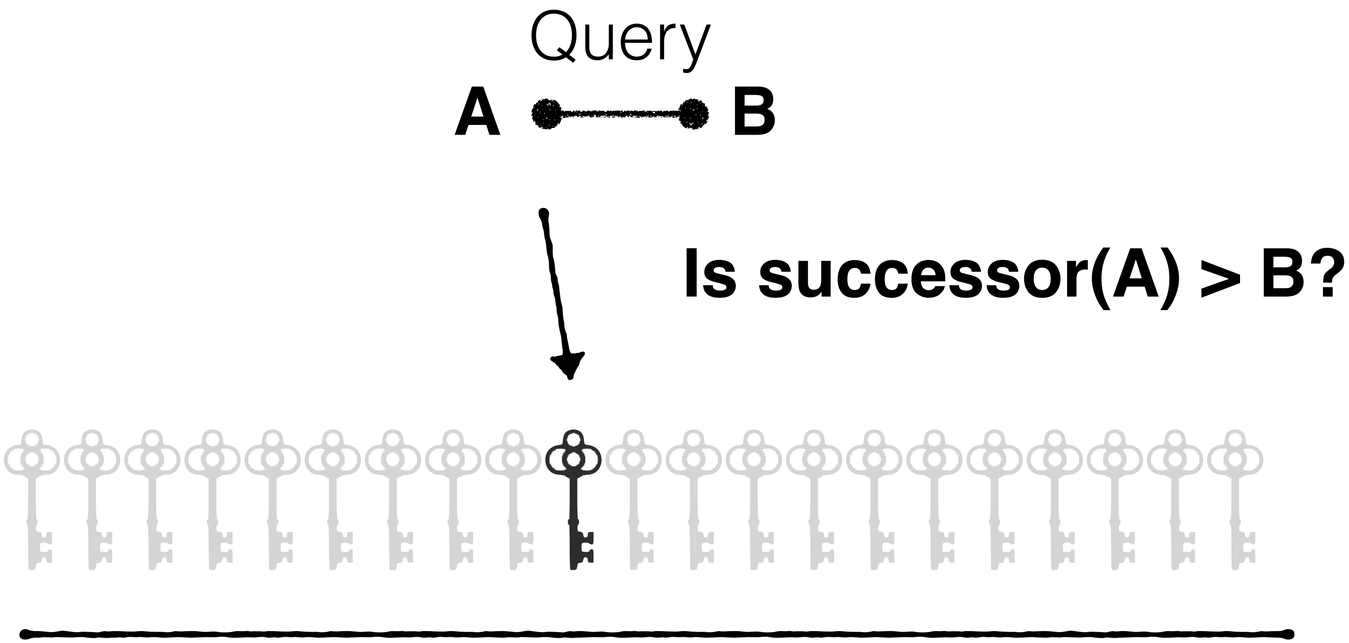
Store keys lexicographically

Query

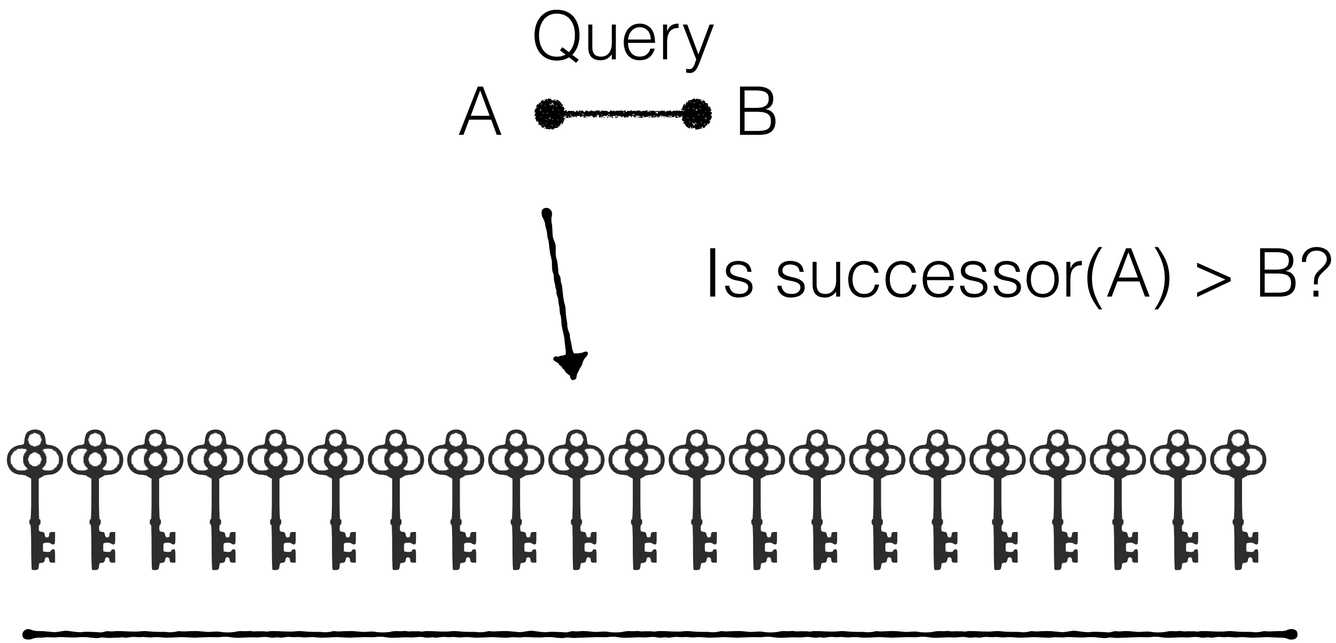
A **B**



Store keys lexicographically

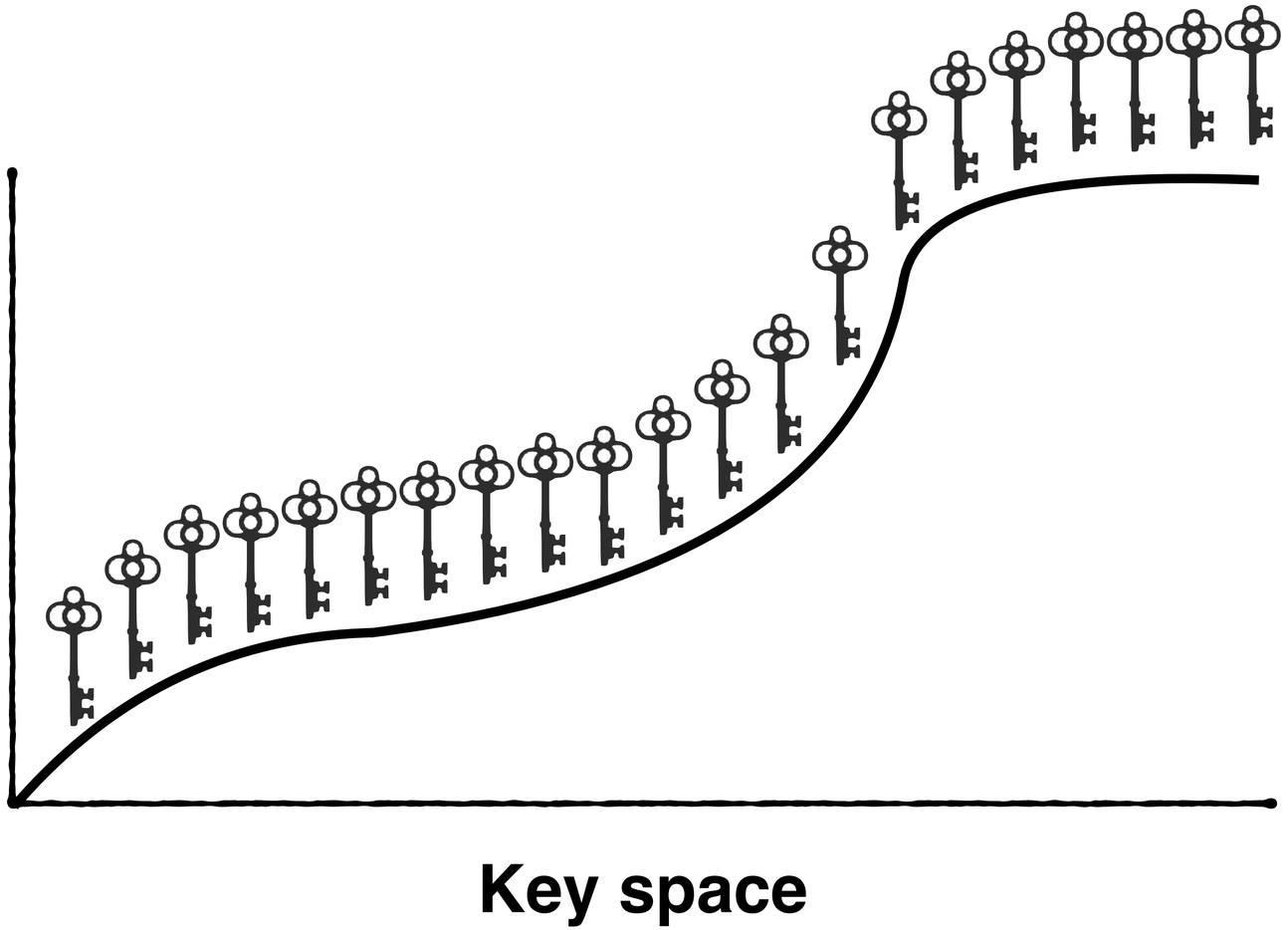


Store keys lexicographically

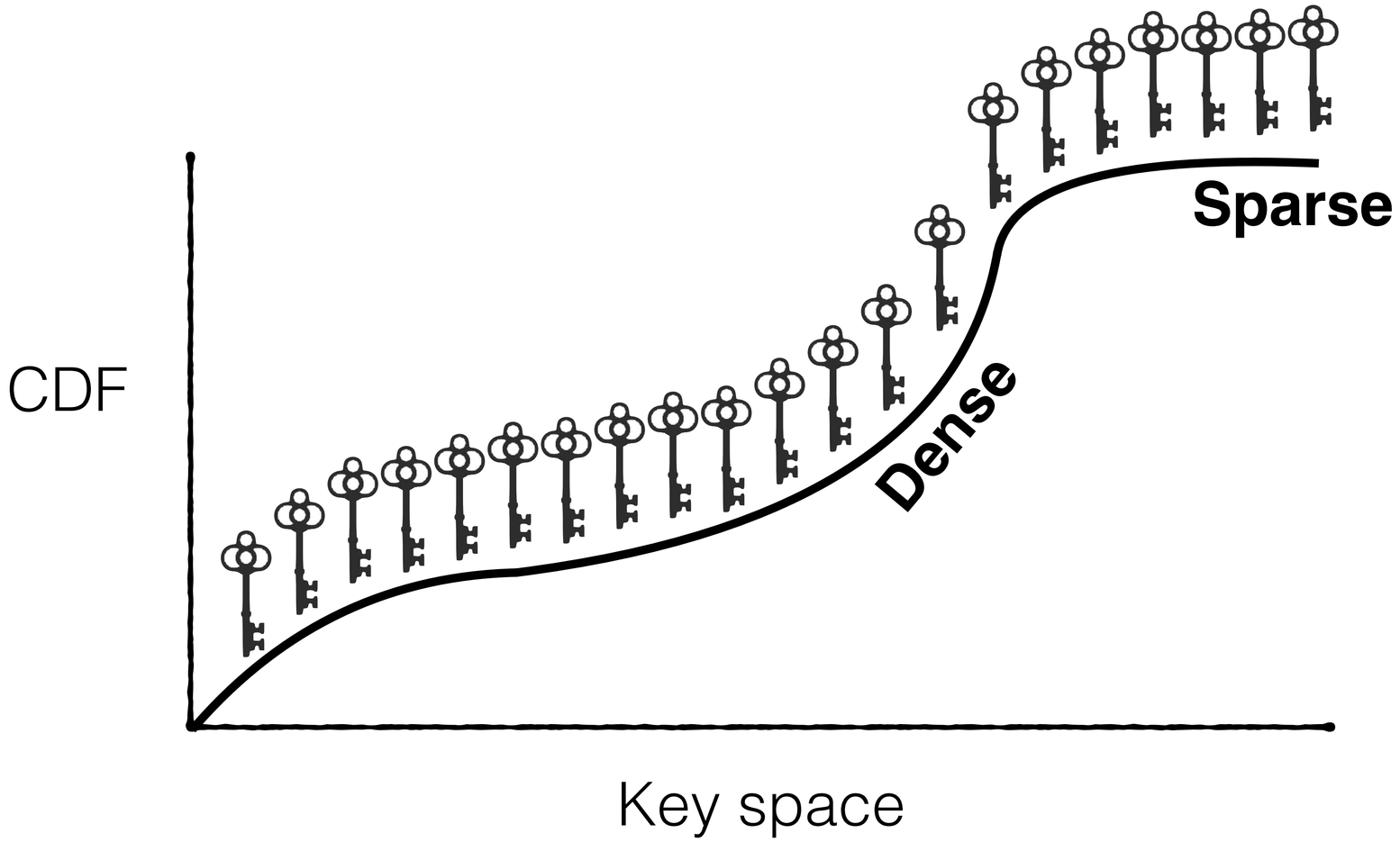


Requirement: support fast successor search

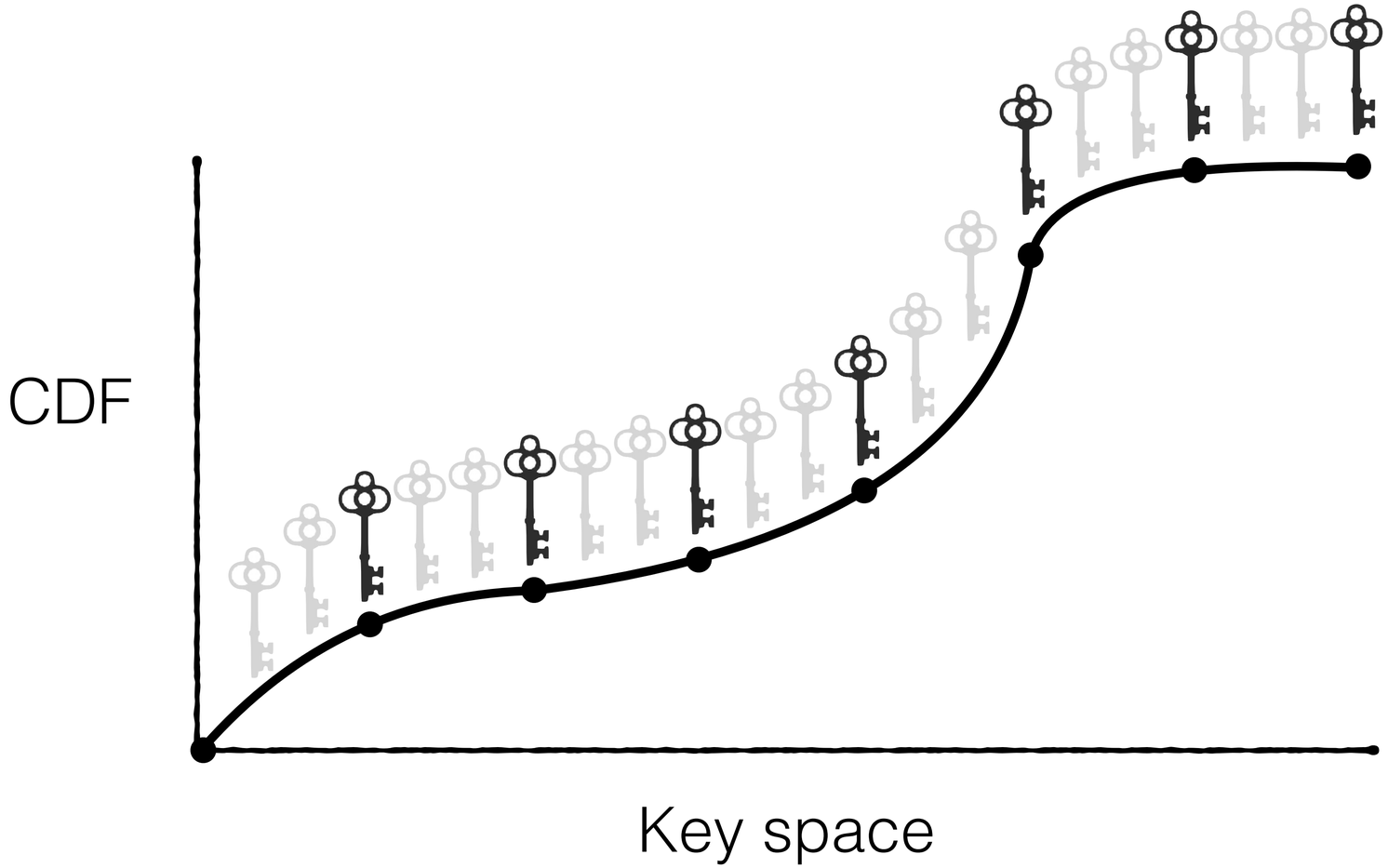
Cumulative Distribution Function (CDF)



Key space

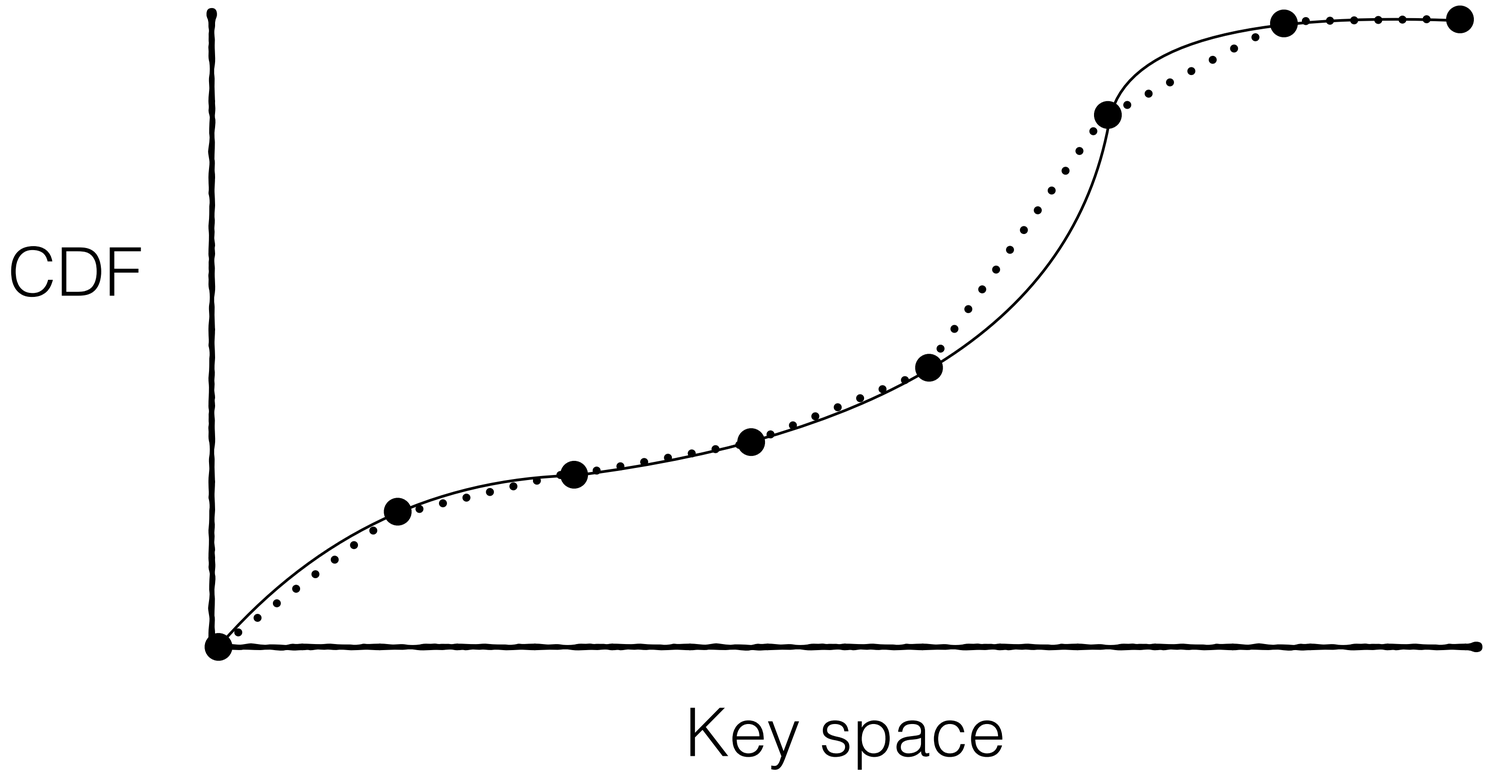

CDF
Sparse
Dense
Key space

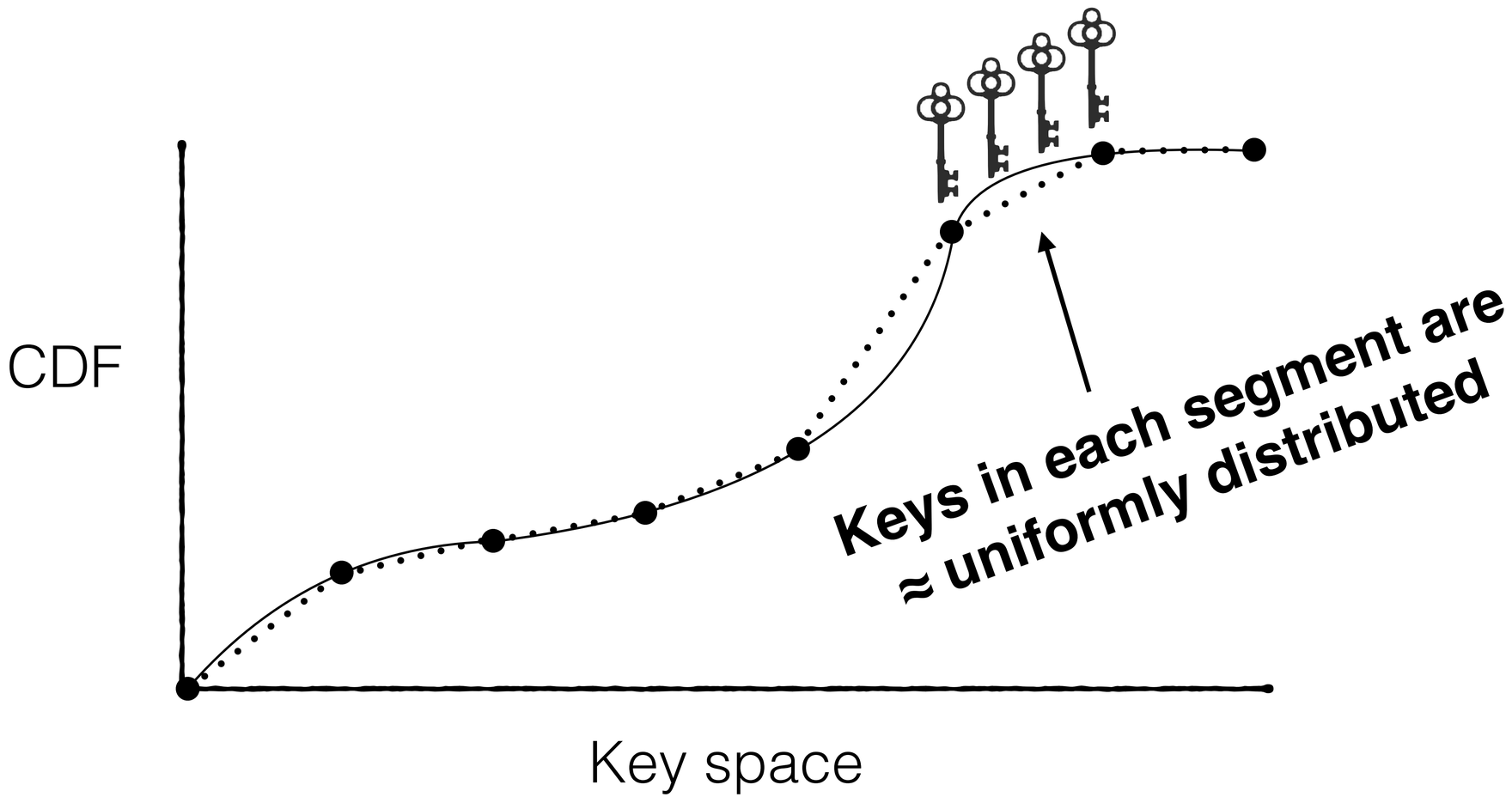
Sample every 1000-th key

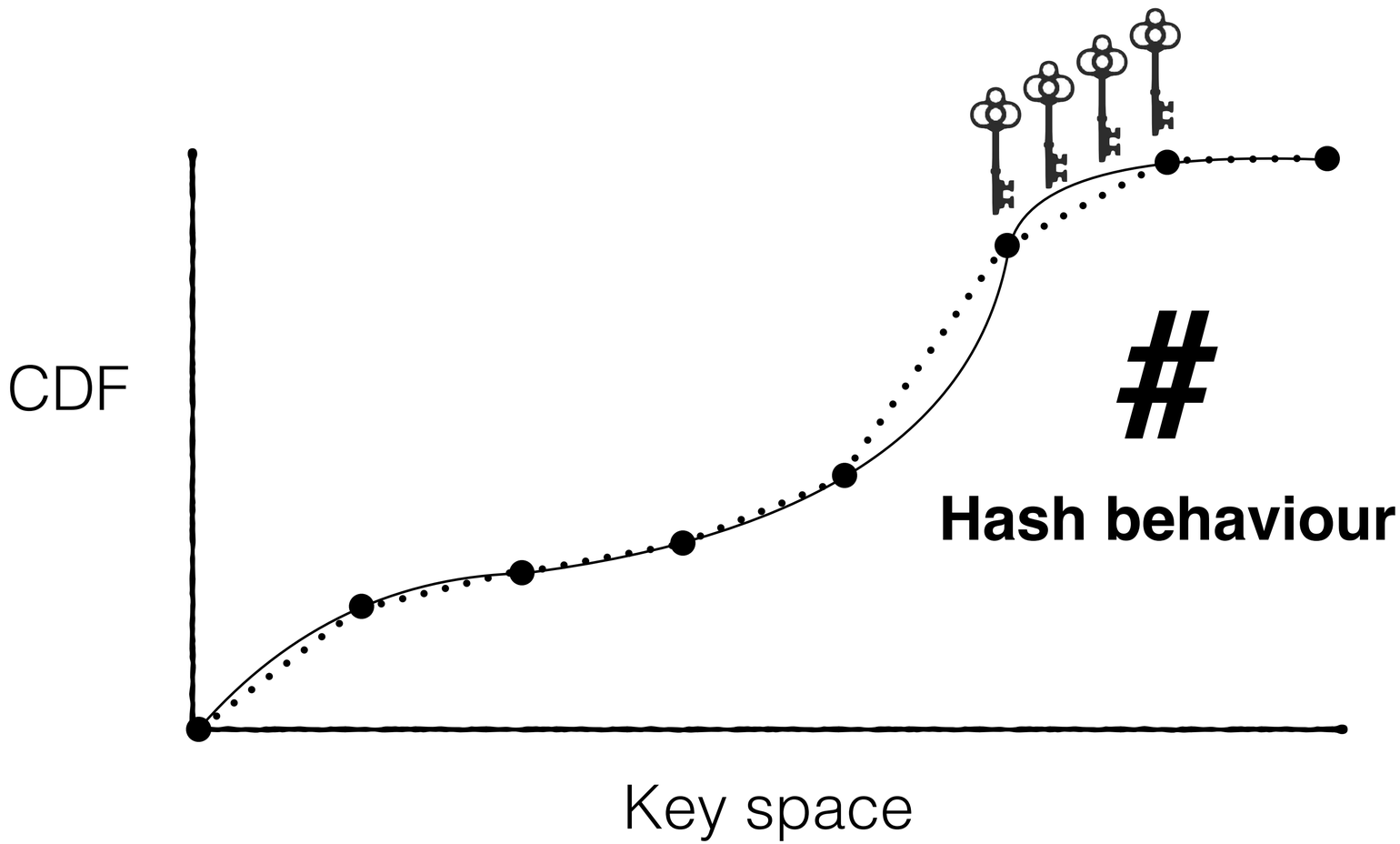


Sample every 1000-th key

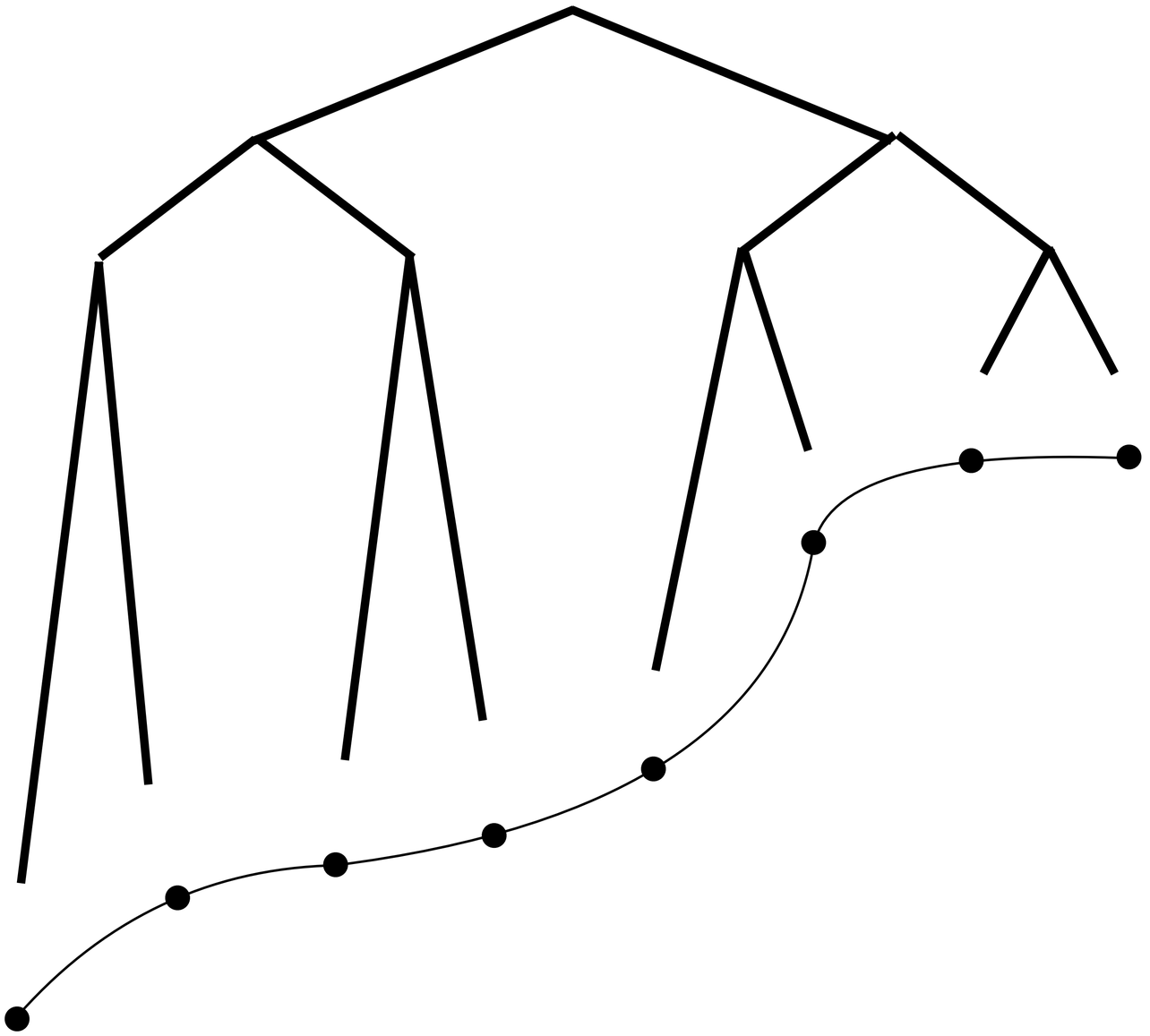
The samples approximate the distribution




CDF
Key space
Keys in each segment are
≈ uniformly distributed

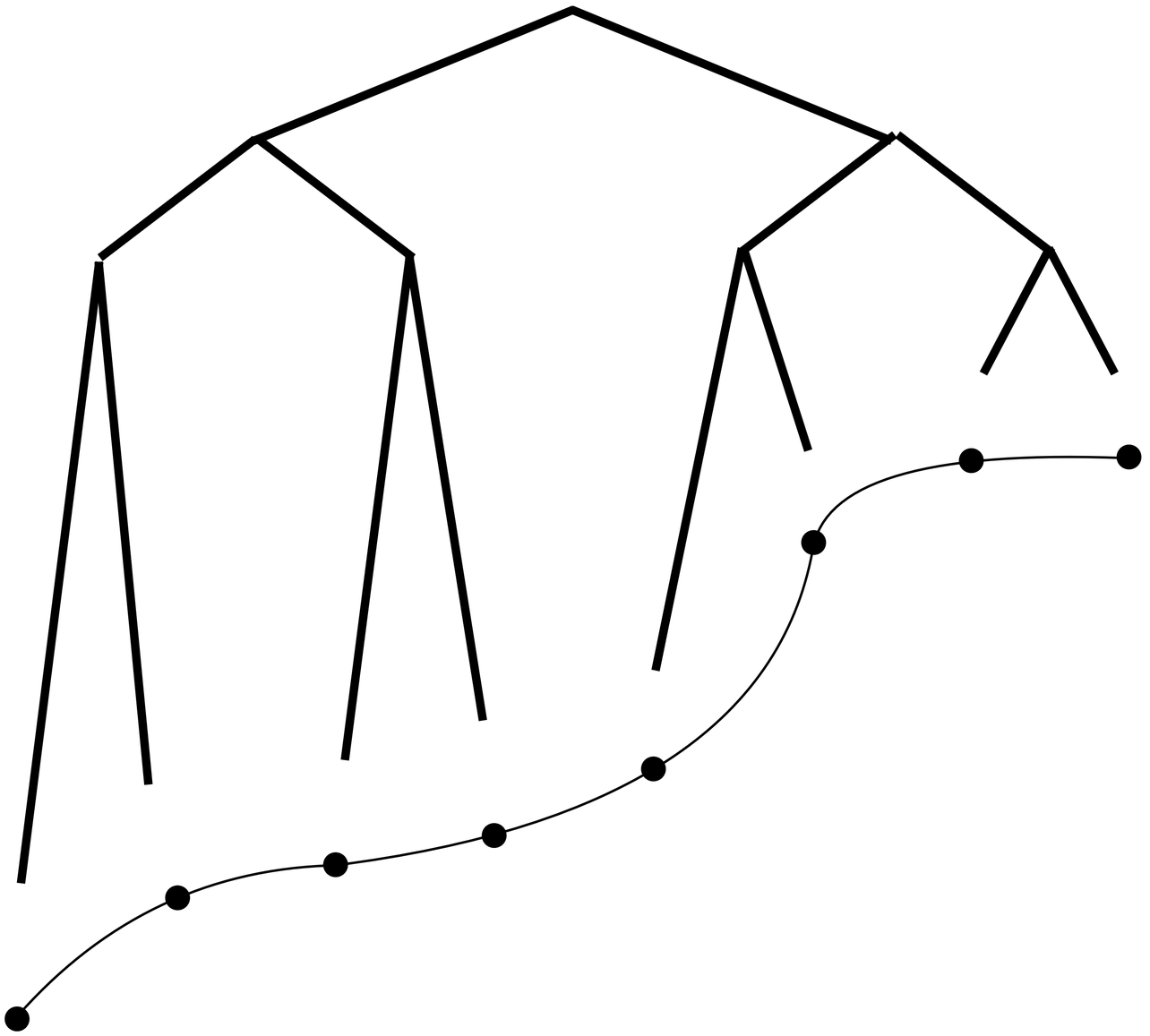

CDF
#
Hash behaviour
Key space

Build tree from samples



Build tree from samples

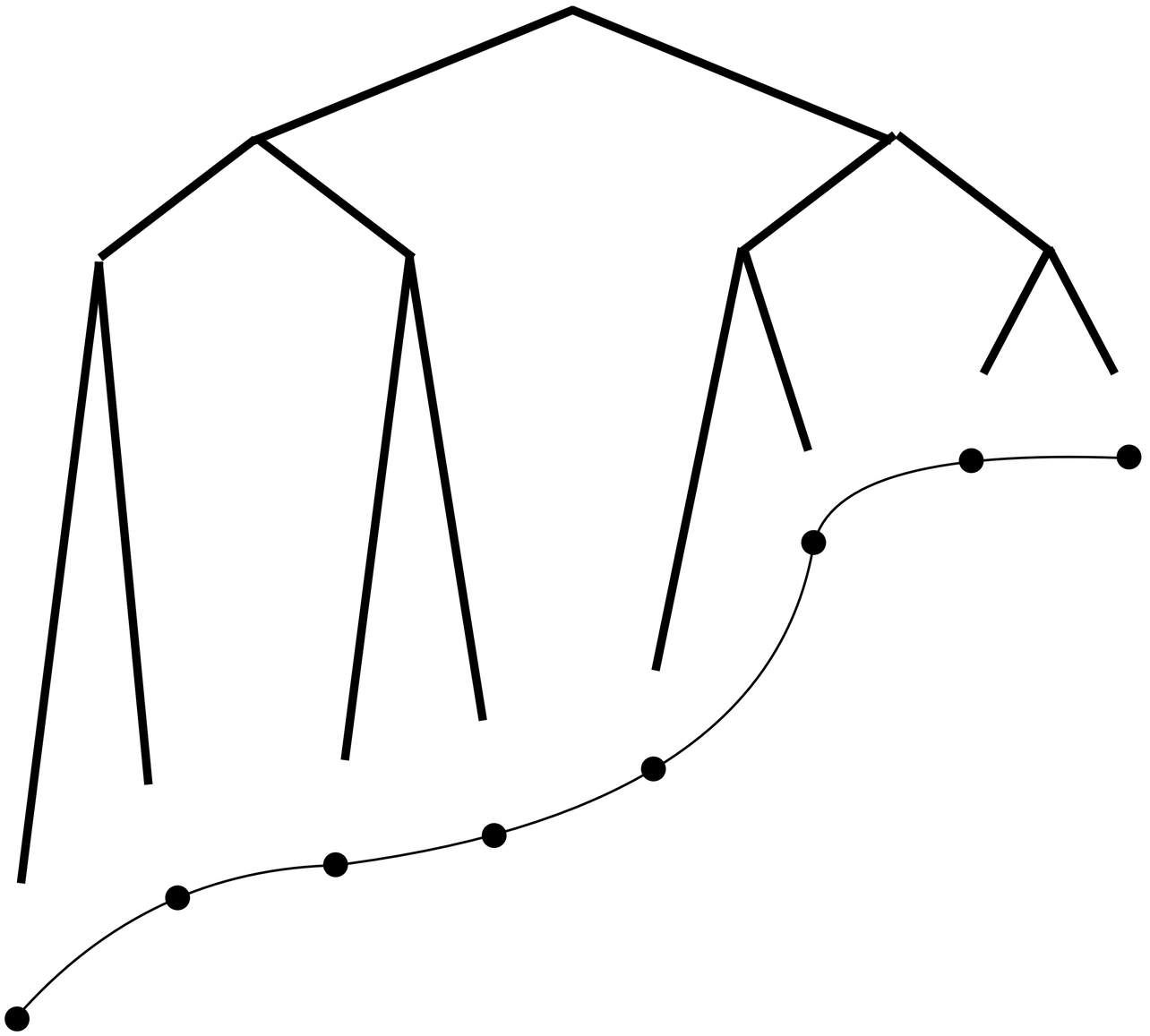
0.1% of data size

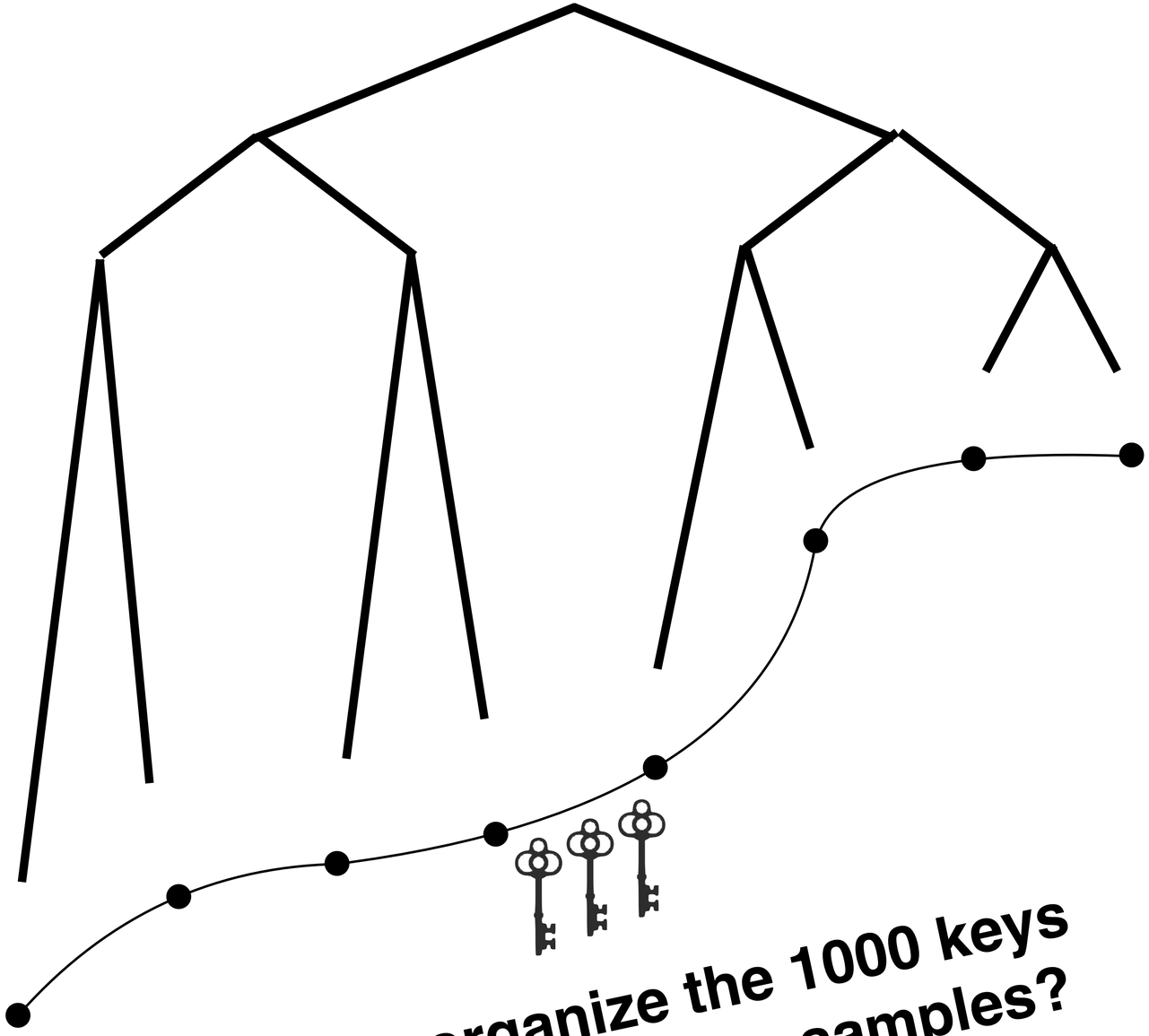


Build tree from samples

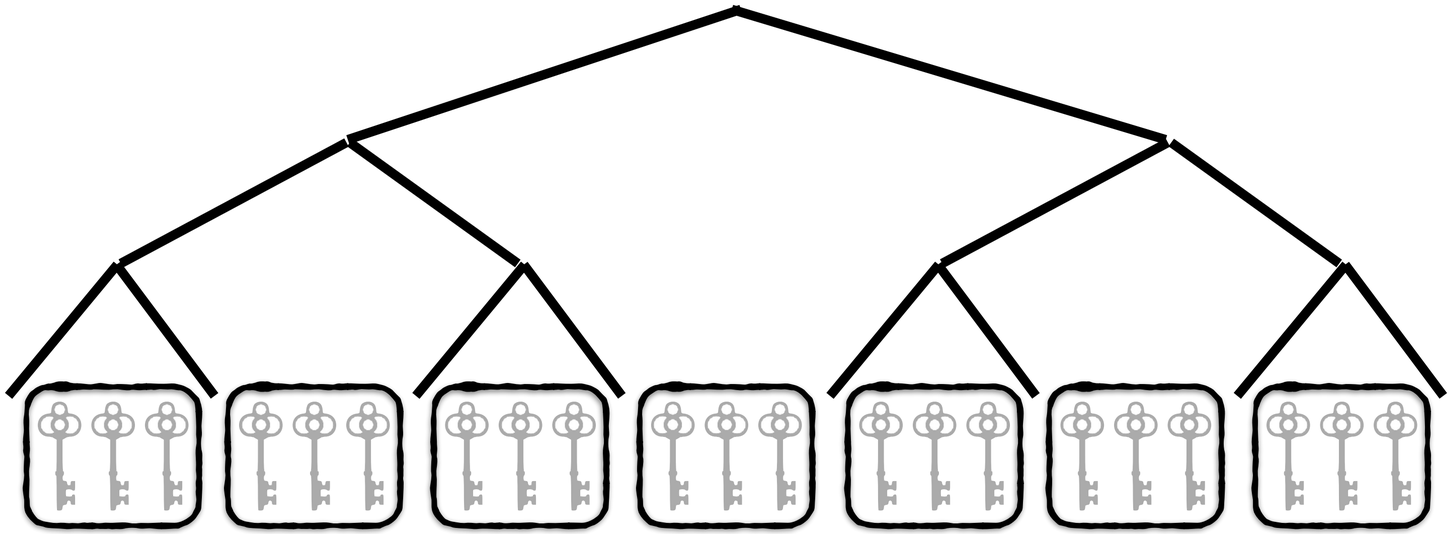
0.1% of data size

Fits in CPU caches



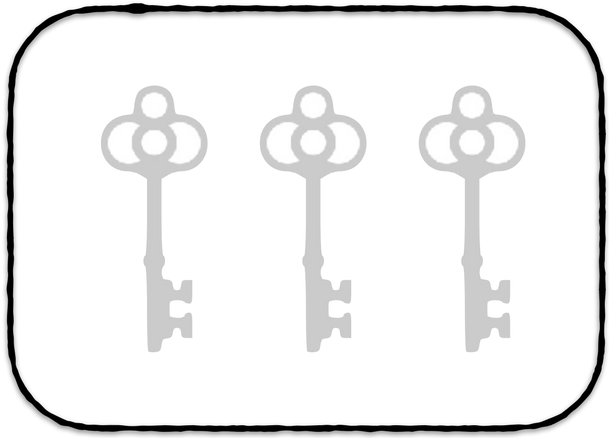


How to organize the 1000 keys in-between any two samples?

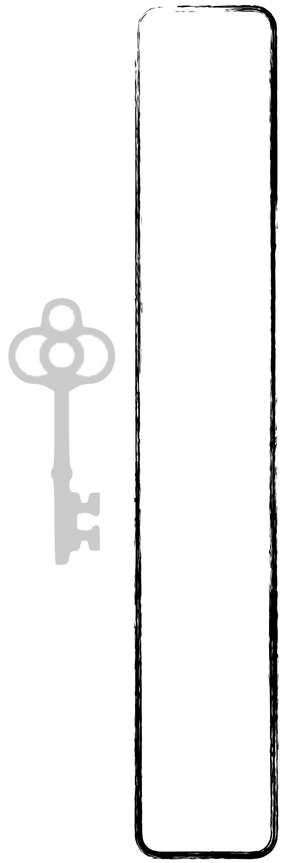


Infix Stores

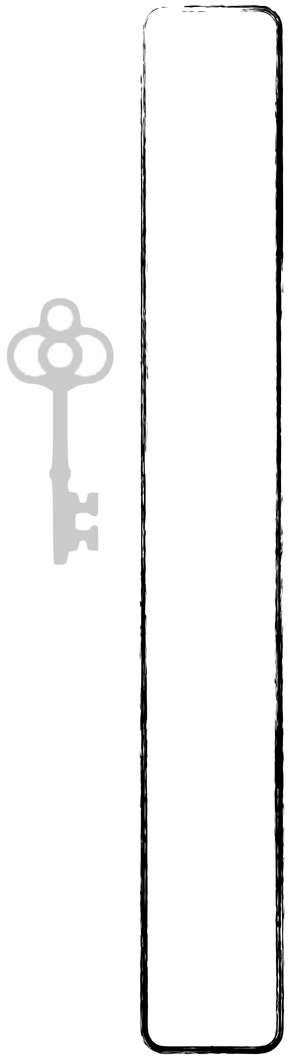
Order-preserving hash table of compressed keys



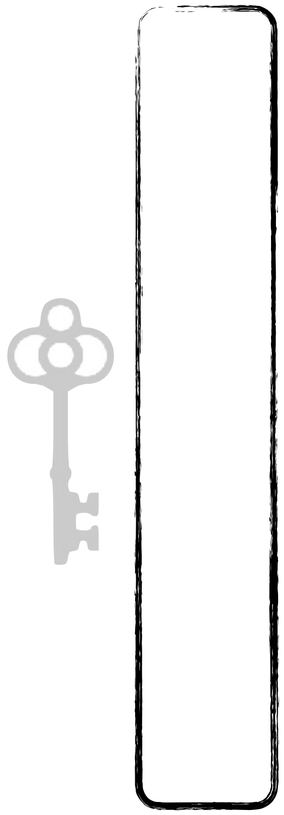
Infix Store



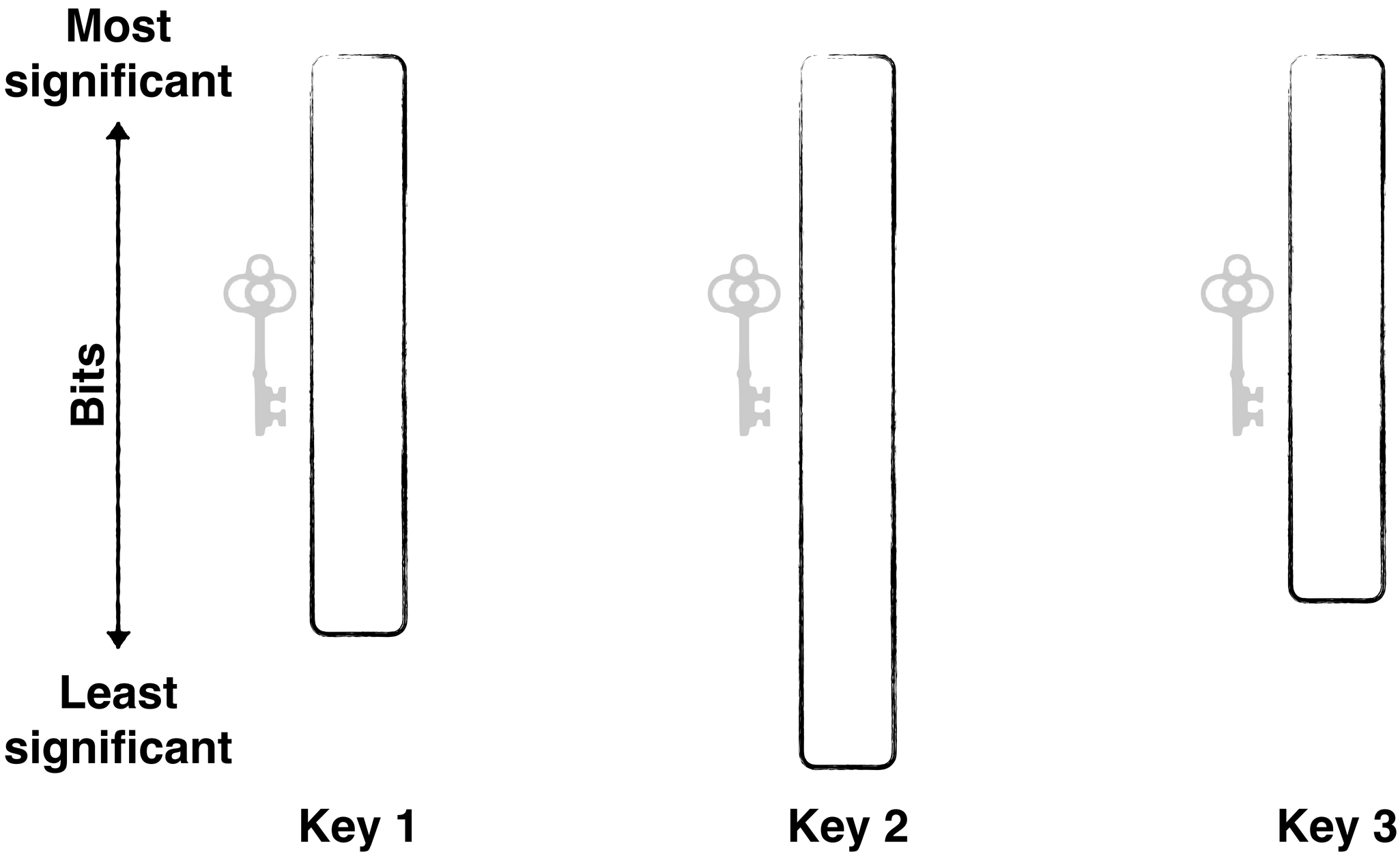
Key 1



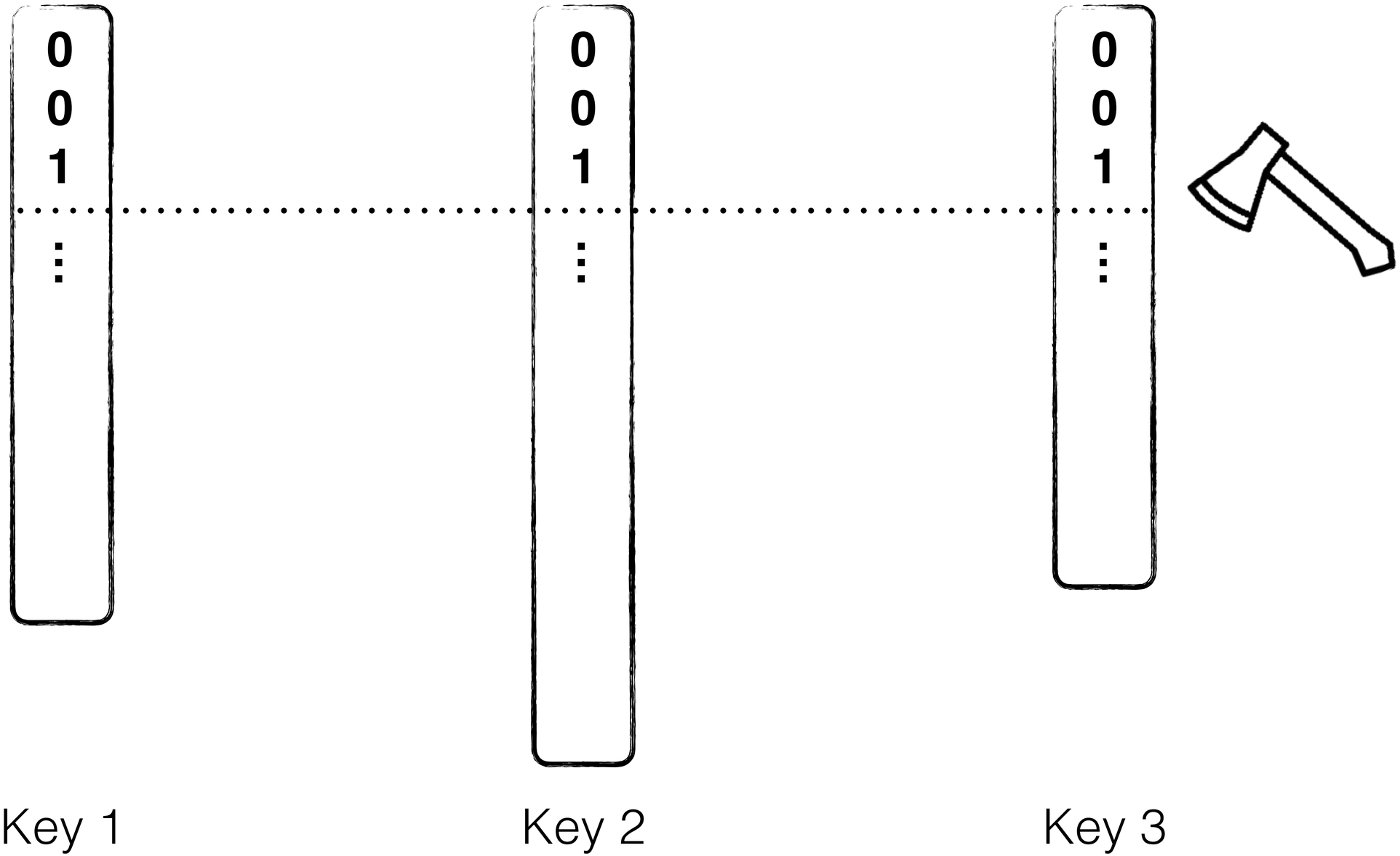
Key 2



Key 3


Most significant
Bits
Least significant
Key 1
Key 2
Key 3

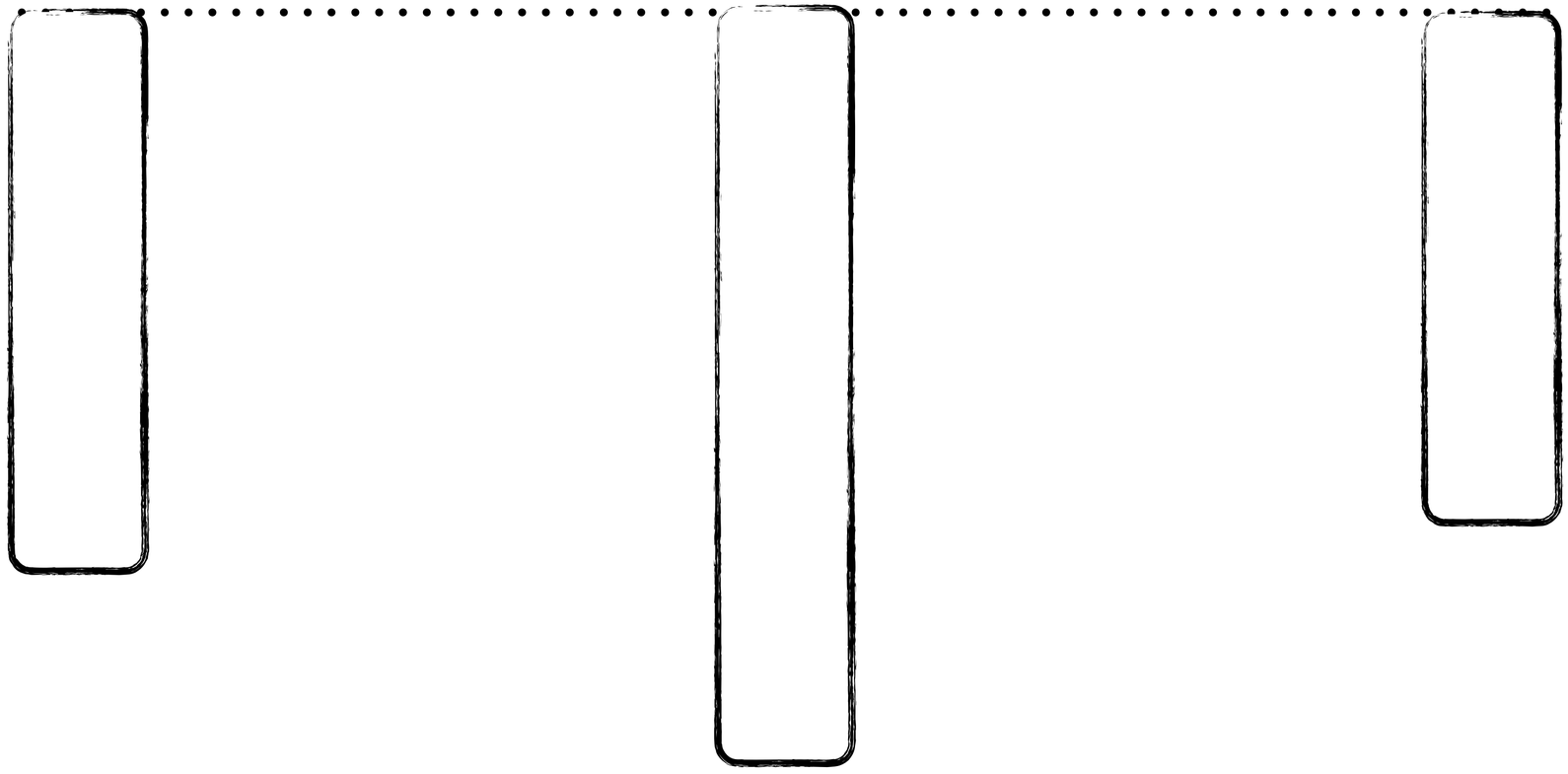
Remove longest common prefix (LCP)



We don’t lose information as LCP is in tree

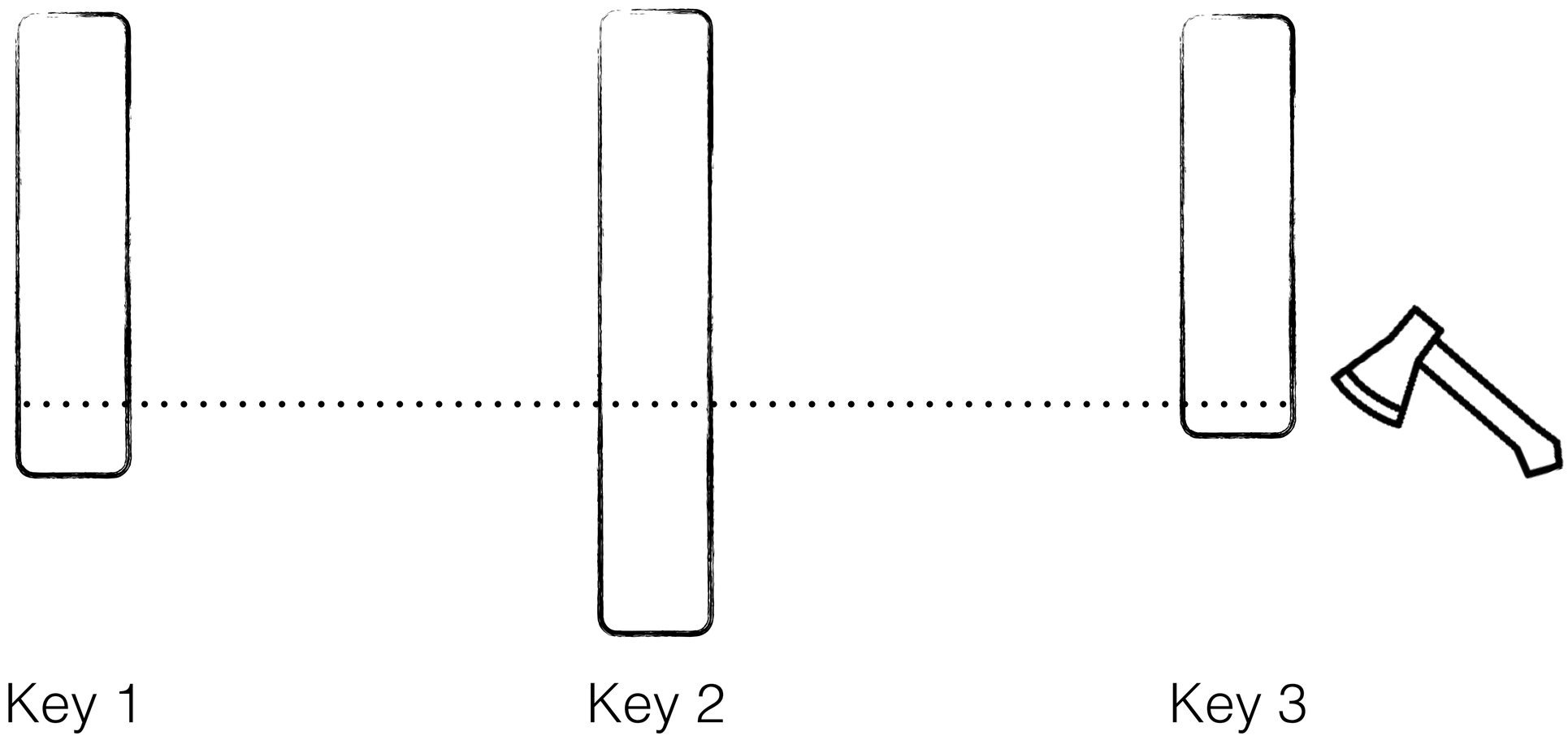
0
0
1

0
0
1

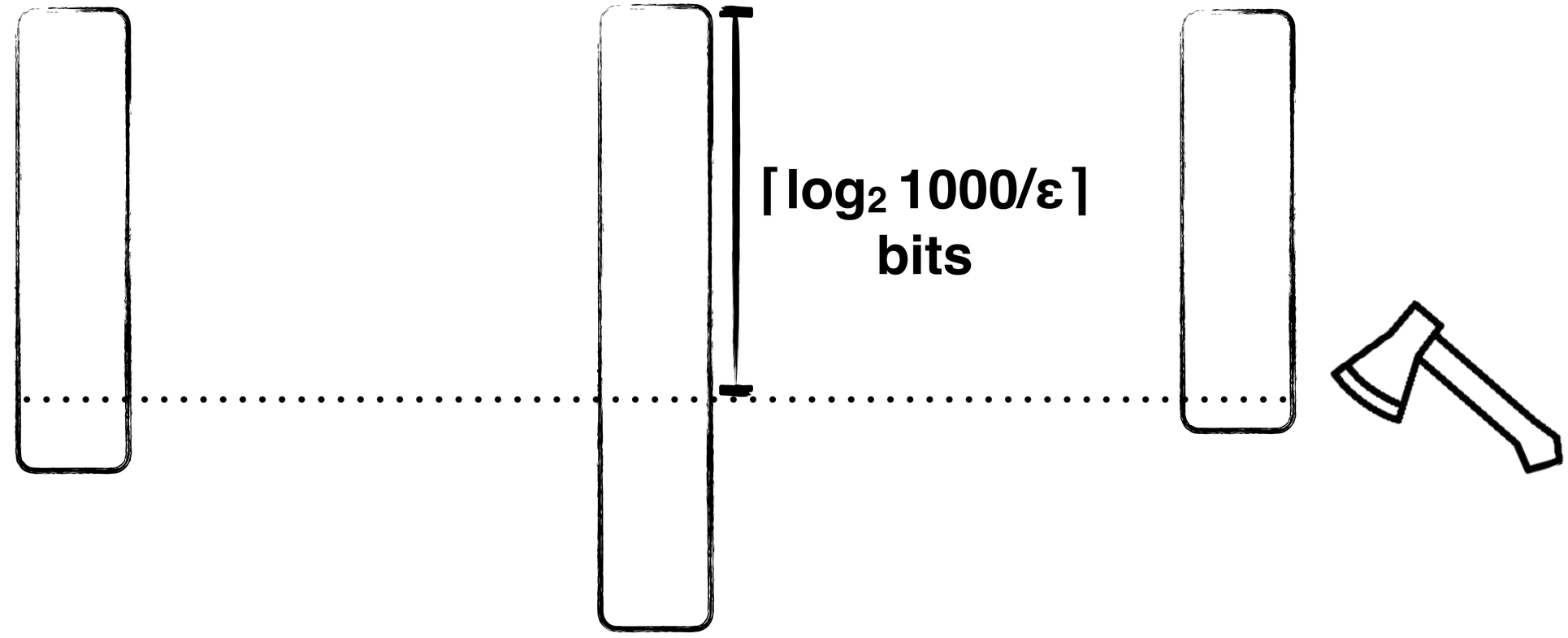


Key 1 Key 2 Key 3

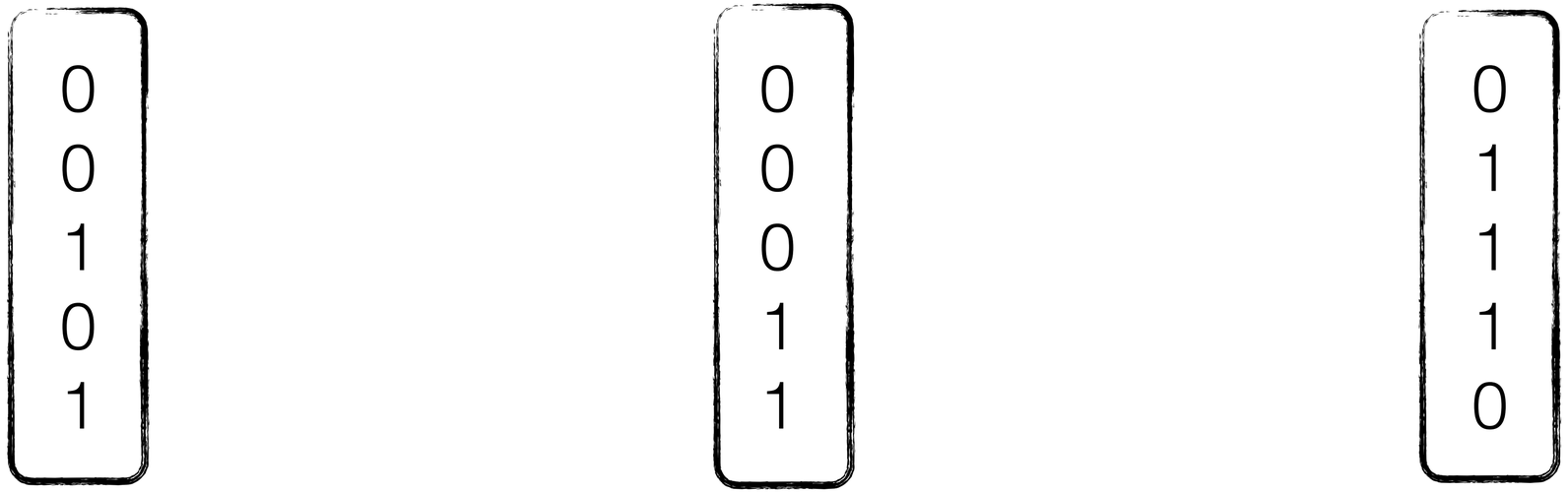
Remove Suffix to homogenize keys lengths



Remove Suffix to homogenize keys lengths



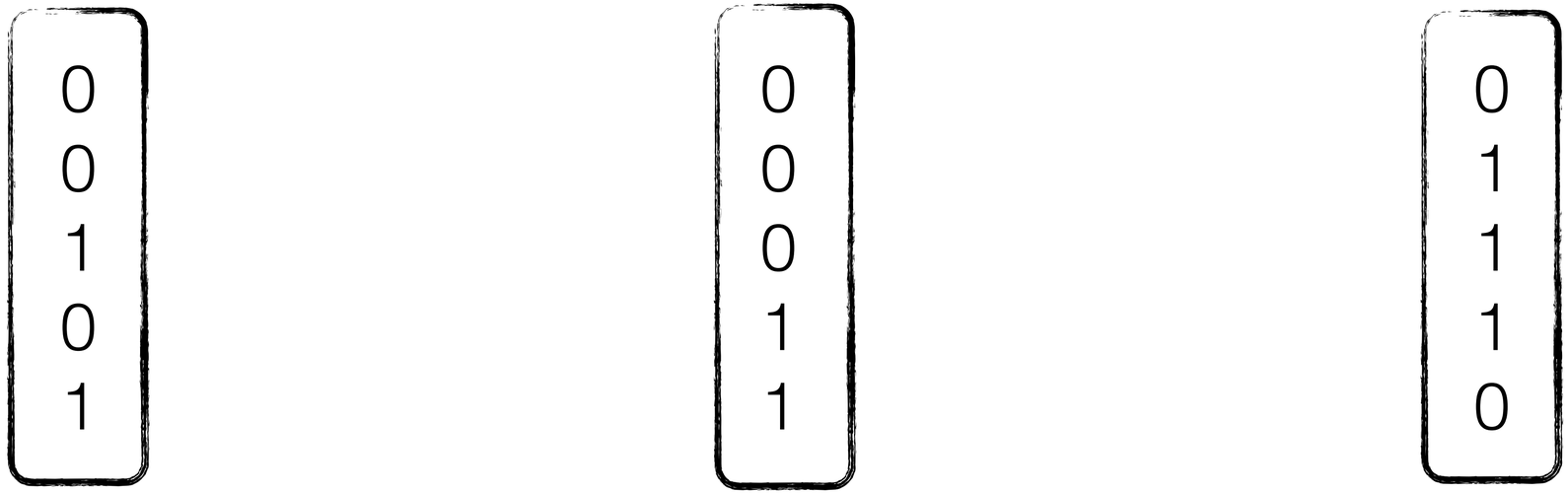
What's left are ≈ uniformly distributed "infixes"



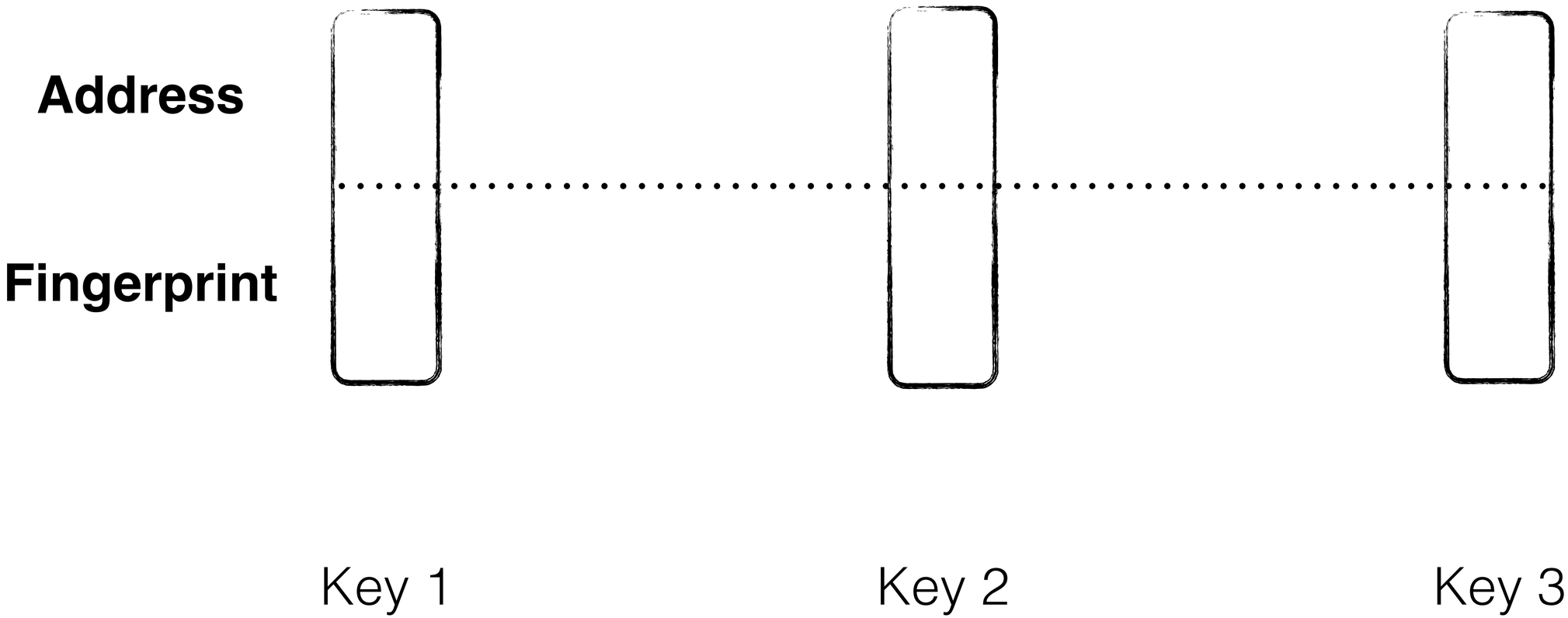
Key 1 Key 2 Key 3

What's left are ≈ uniformly distributed "infixes"

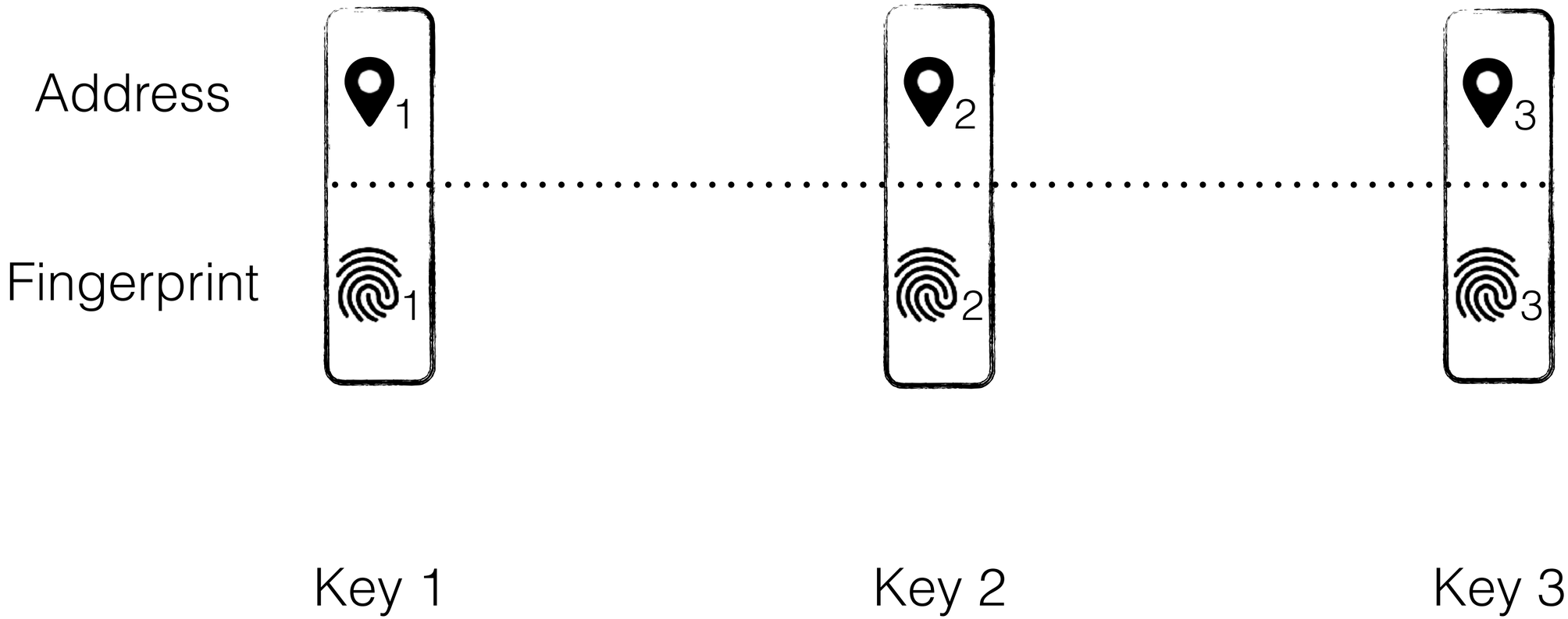
Store in a structure similar to a quotient filter



Store in a structure similar to a quotient filter



Divide into quotients and remainders



Divide into quotients and remainders

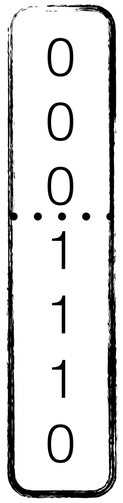
Address

10 bits

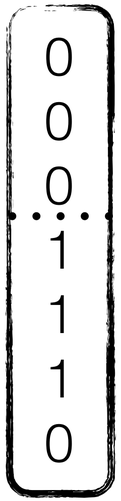
Fingerprint

$\log_2 (1 / \varepsilon)$

Key 1 Key 2 Key 3

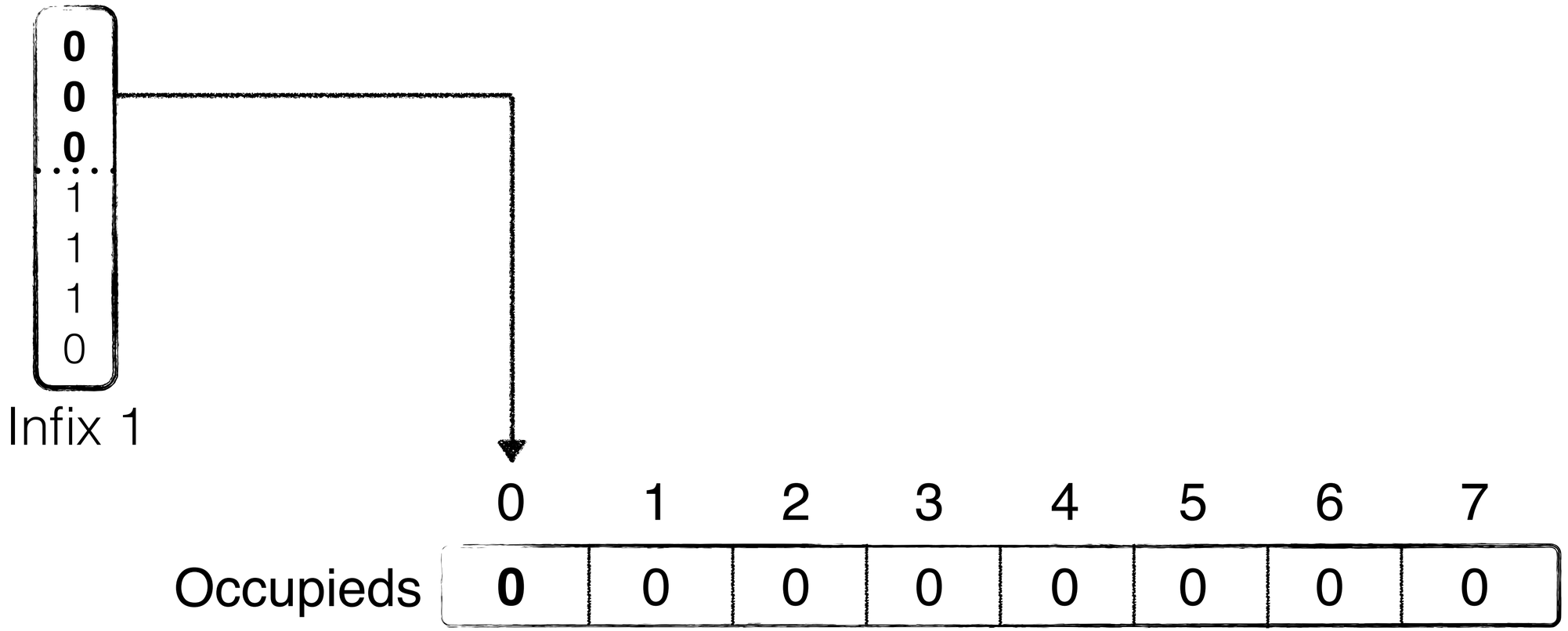


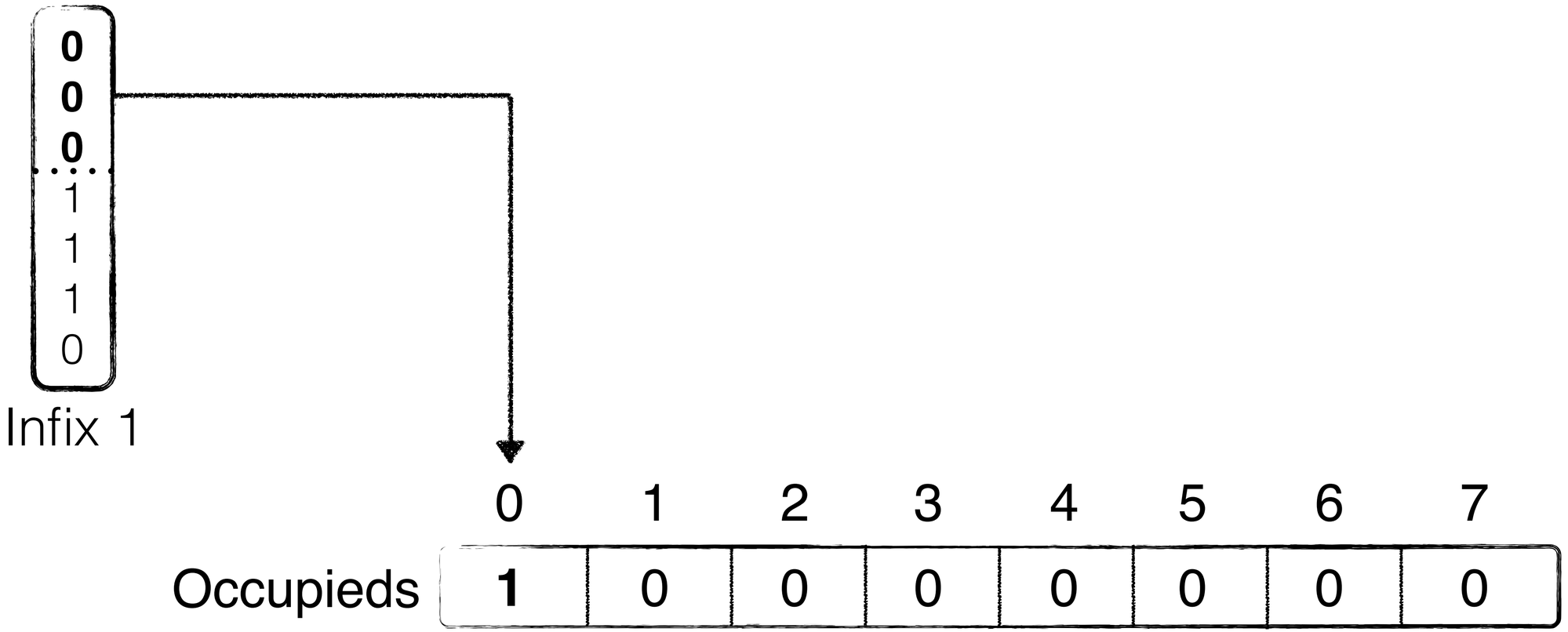
Infix 1

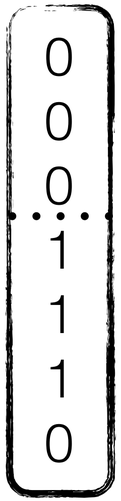


Infix 1

	0	1	2	3	4	5	6	7
Occupieds	0	0	0	0	0	0	0	0

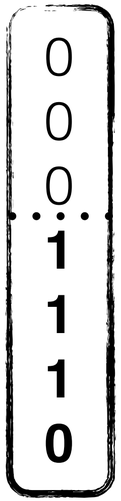

0
0
0
1
1
1
0
Infix 1
0 1 2 3 4 5 6 7
Occupieds
0 0 0 0 0 0 0 0


0
0
0
1
1
1
0
Infix 1
0 1 2 3 4 5 6 7
Occupieds
1 0 0 0 0 0 0 0

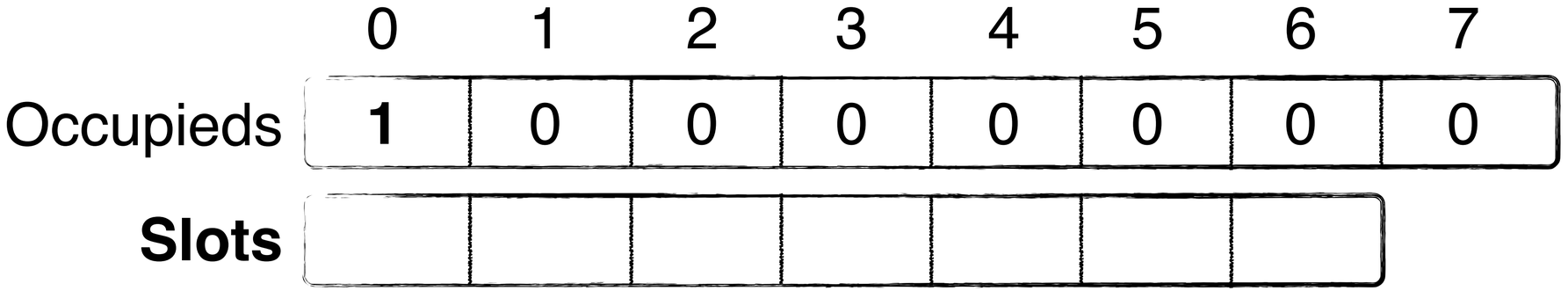


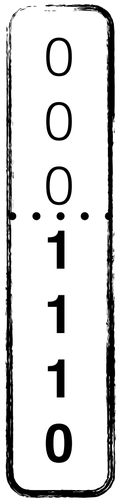
Infix 1

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

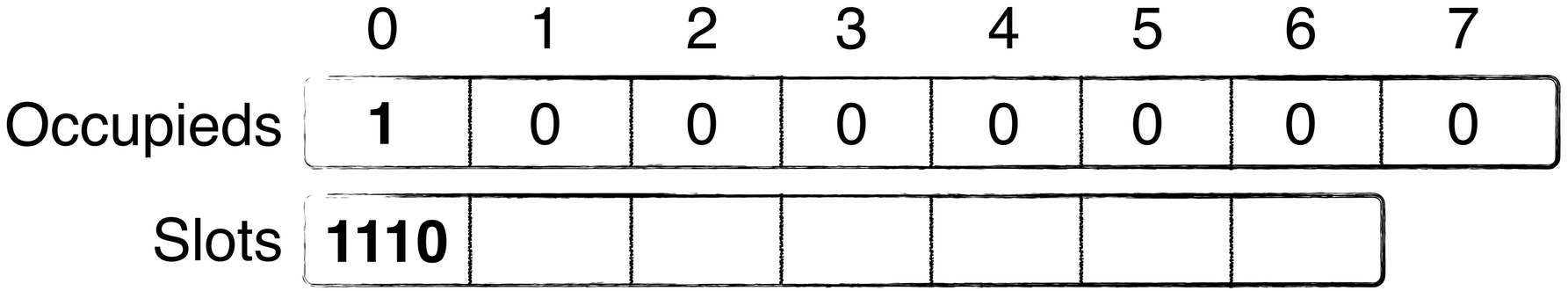


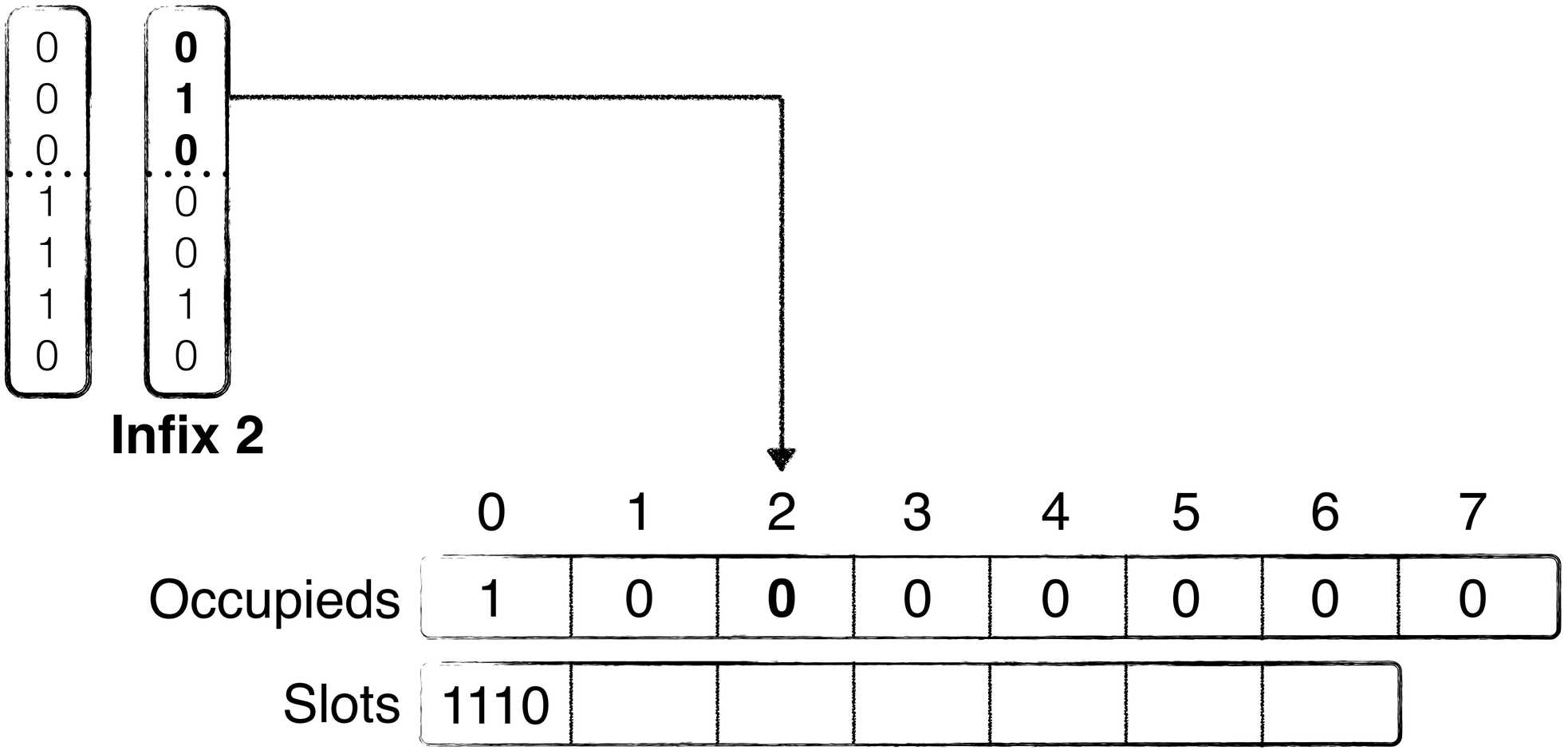
Infix 1

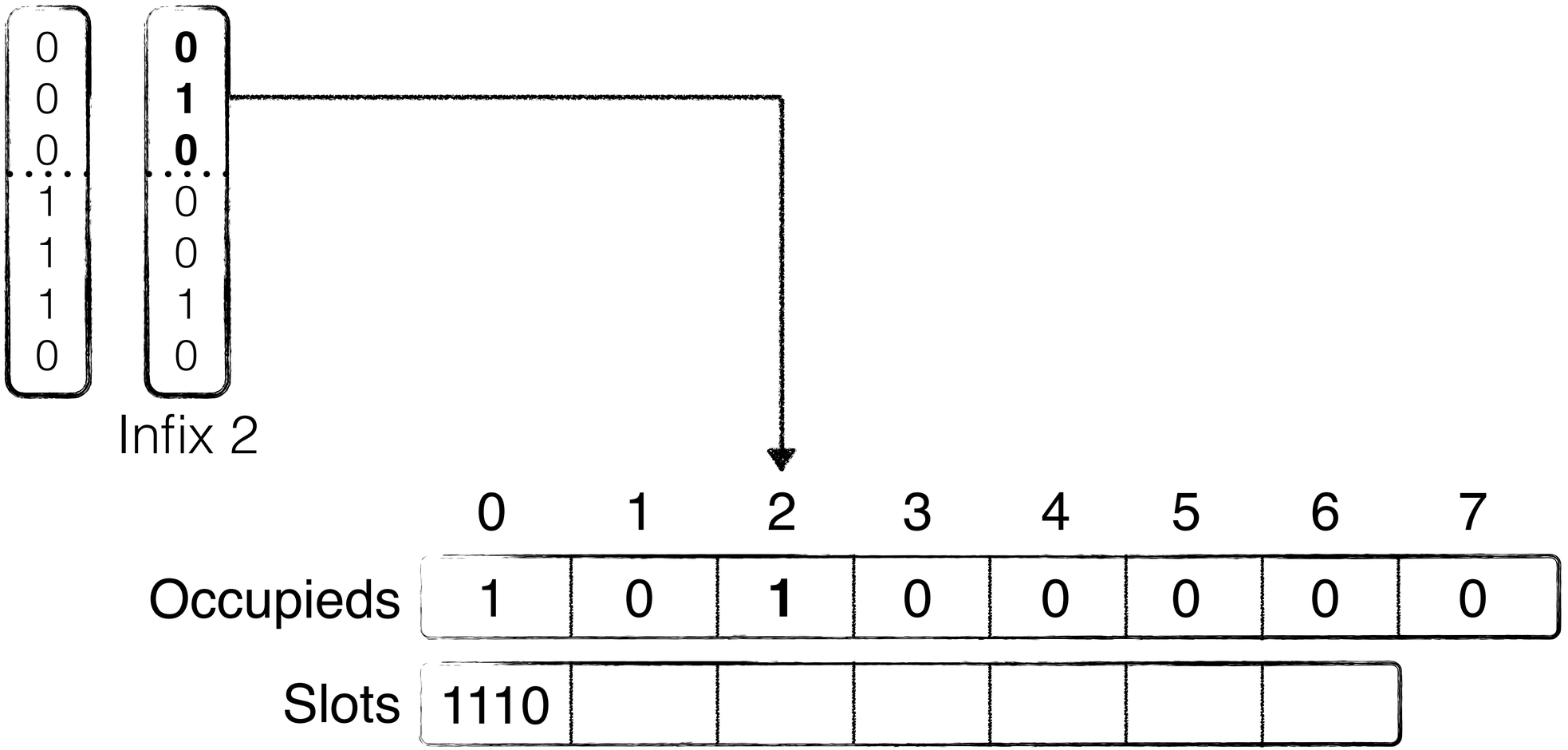


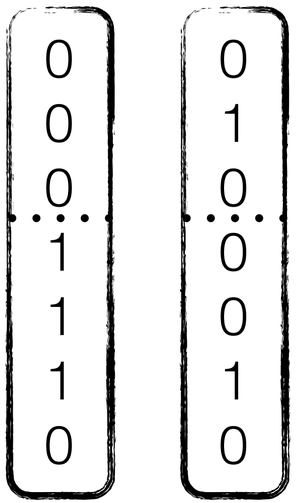


Infix 1



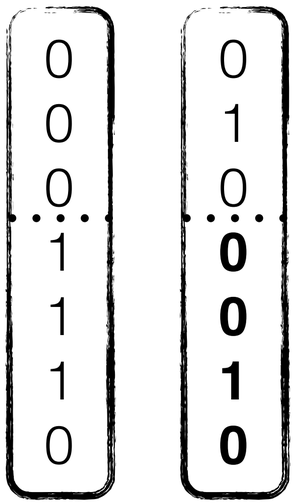

0
0
0
0
1
1
1
0
0
1
0
0
0
1
0
Infix 2
0
1
2
3
4
5
6
7
Occupieds
1
0
0
0
0
0
0
0
Slots
1110


0
0
0
1
1
1
0
0
1
0
0
0
1
0
Infix 2
0 1 2 3 4 5 6 7
Occupieds
1 0 1 0 0 0 0 0
Slots
1110



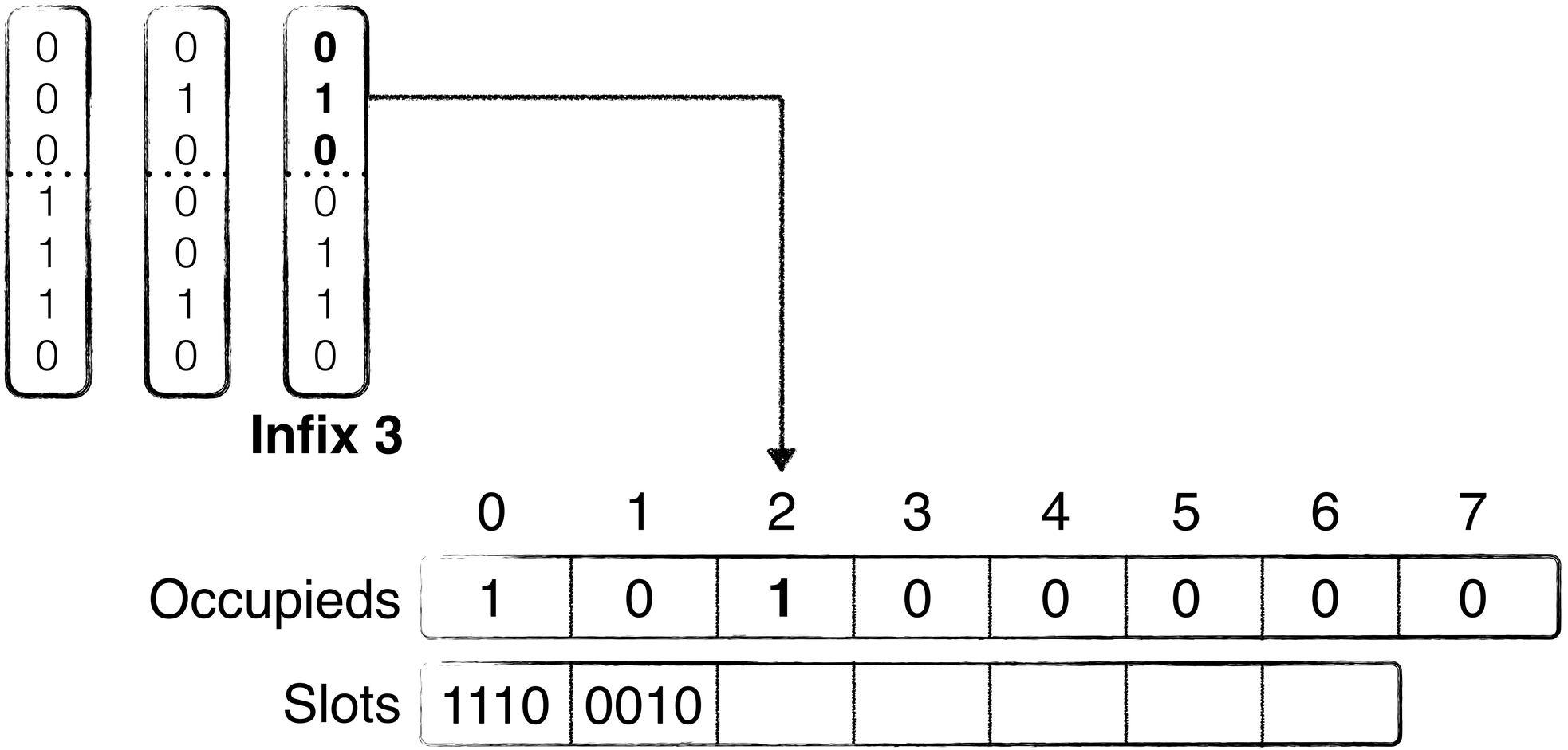
Infix 2

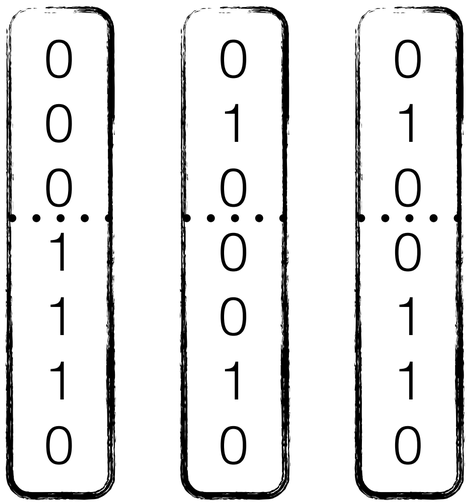
	0	1	2	3	4	5	6	7
Occupieds	1	0	**1**	0	0	0	0	0
Slots	1110							



Infix 2

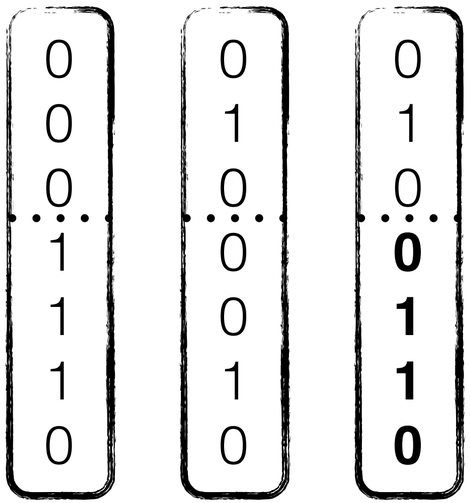
	0	1	2	3	4	5	6	7
Occupieds	1	0	**1**	0	0	0	0	0
Slots	1110	**0010**						


0
0
0
1
1
1
0
0
1
0
0
0
1
0
0
1
1
0
Infix 3
0
1
2
3
4
5
6
7
Occupieds
1
0
1
0
0
0
0
0
Slots
1110
0010

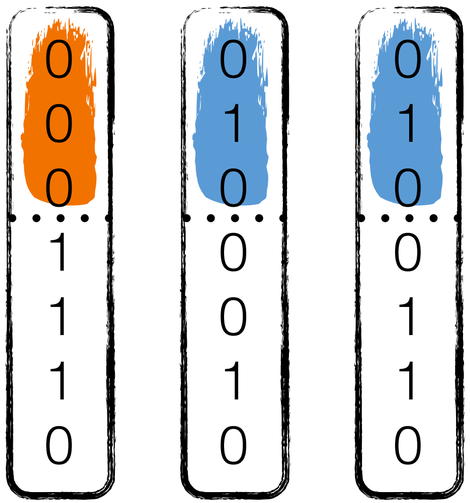


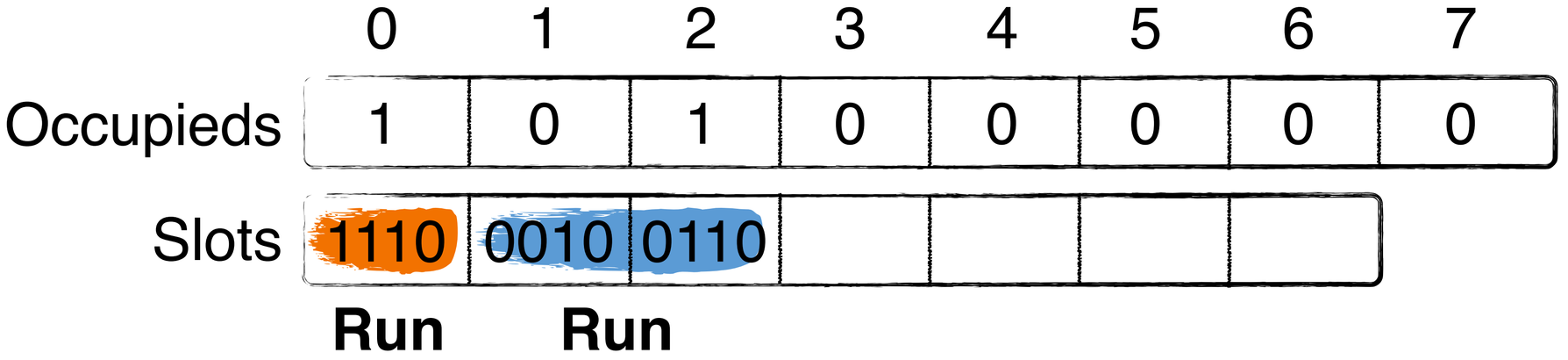
Infix 3

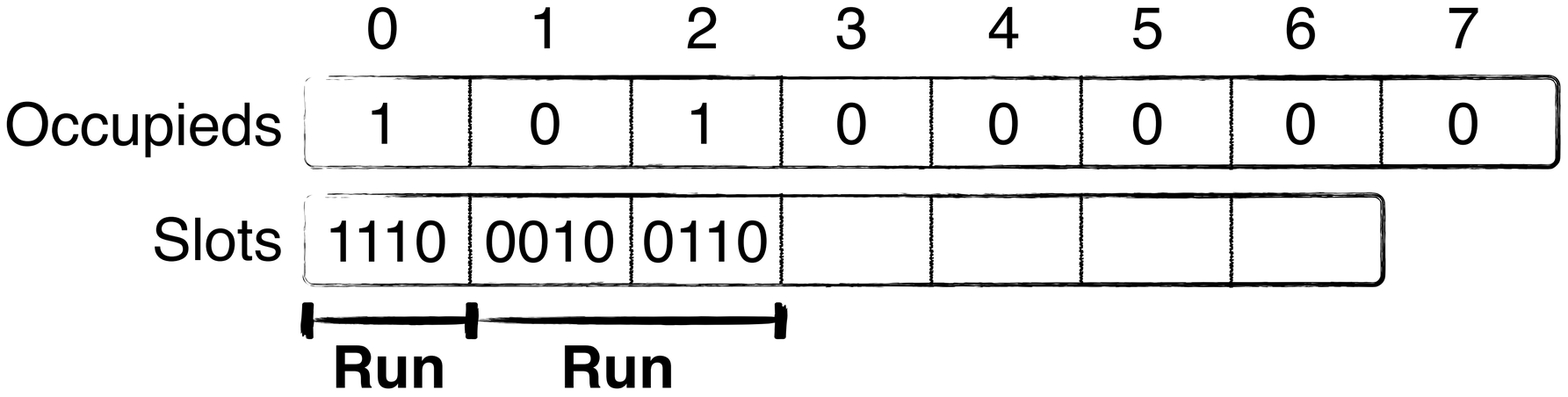
	0	1	2	3	4	5	6	7
Occupieds	1	0	**1**	0	0	0	0	0
Slots	1110	0010						

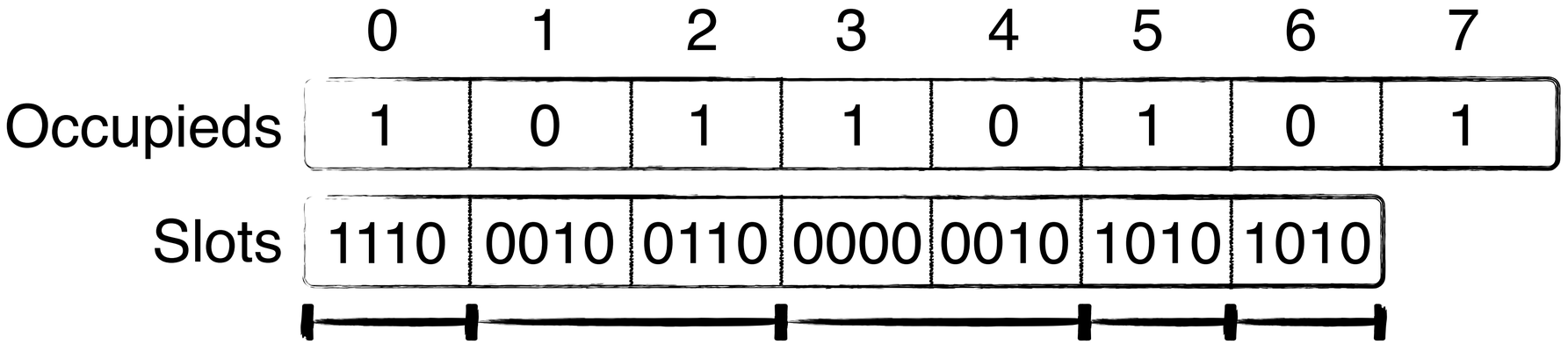


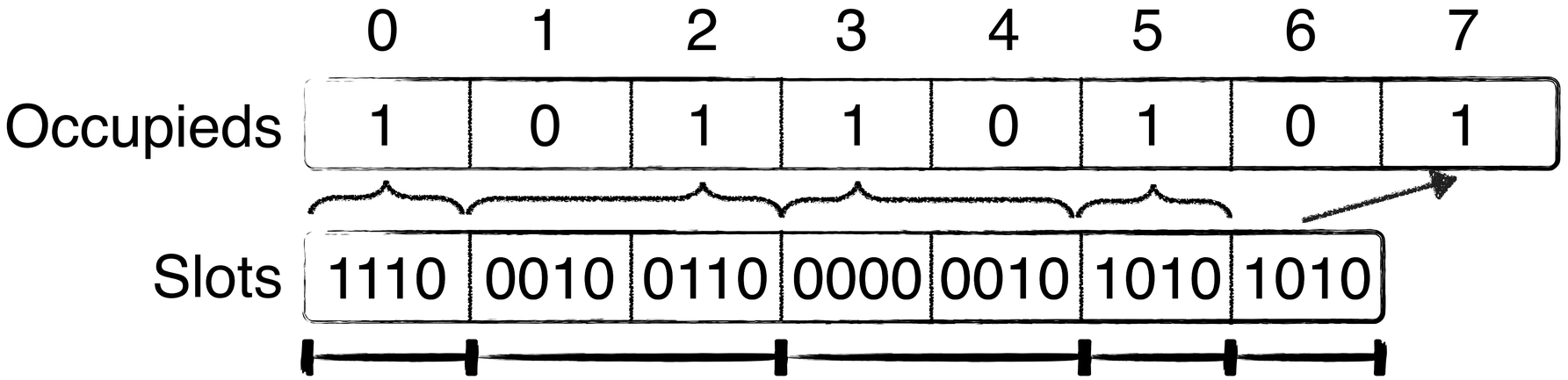
	0	1	2	3	4	5	6	7
Occupieds	1	0	**1**	0	0	0	0	0
Slots	1110	0010	**0110**					

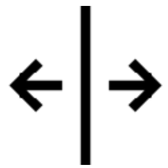

0
0
0
1
1
1
0
0
1
0
0
1
0
0
1
0
0
1
1
0


0
1
2
3
4
5
6
7
Occupieds
1
0
1
0
0
0
0
0
Slots
1110
0010
0110
Run
Run

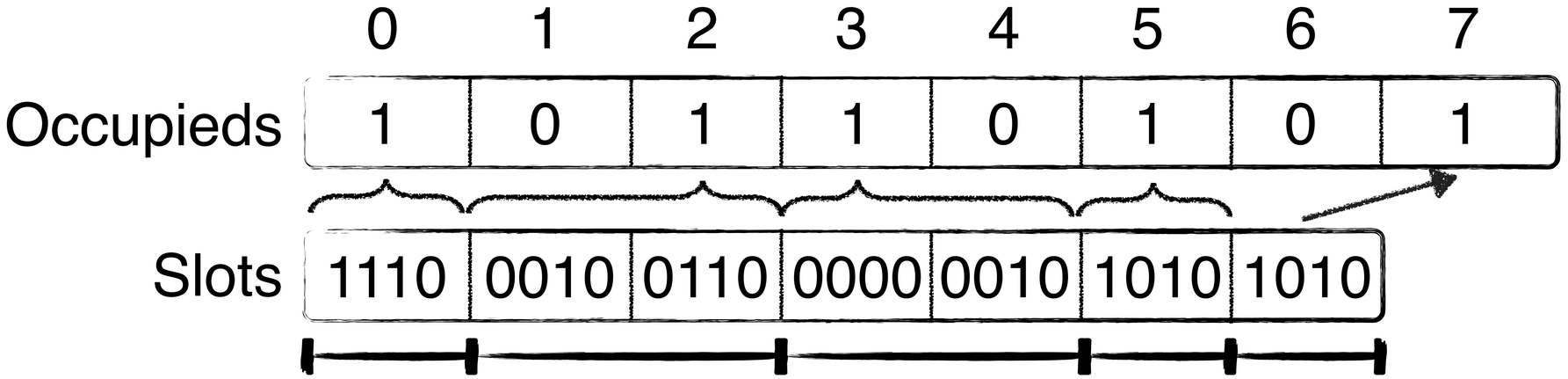

0
1
2
3
4
5
6
7
Occupieds
1
0
1
0
0
0
0
0
Slots
1110
0010
0110
Run
Run

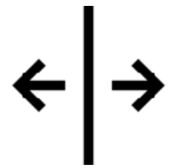

0
1
2
3
4
5
6
7
Occupieds
1
0
1
1
0
1
0
1
Slots
1110
0010
0110
0000
0010
1010
1010


0
1
2
3
4
5
6
7
Occupieds
1
0
1
1
0
1
0
1
Slots
1110
0010
0110
0000
0010
1010
1010

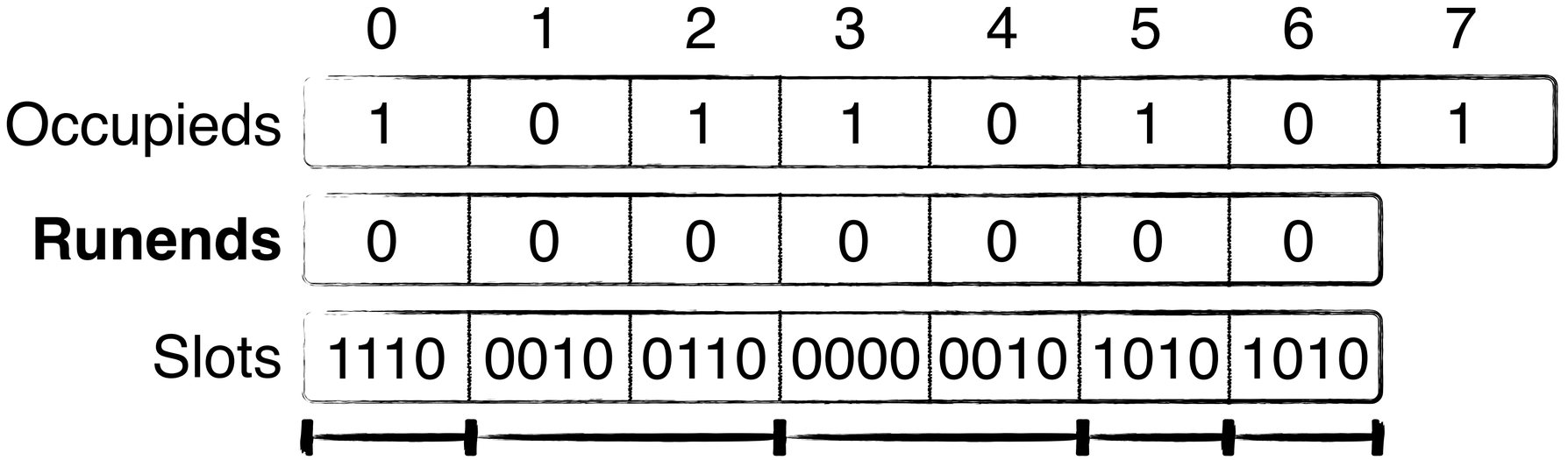


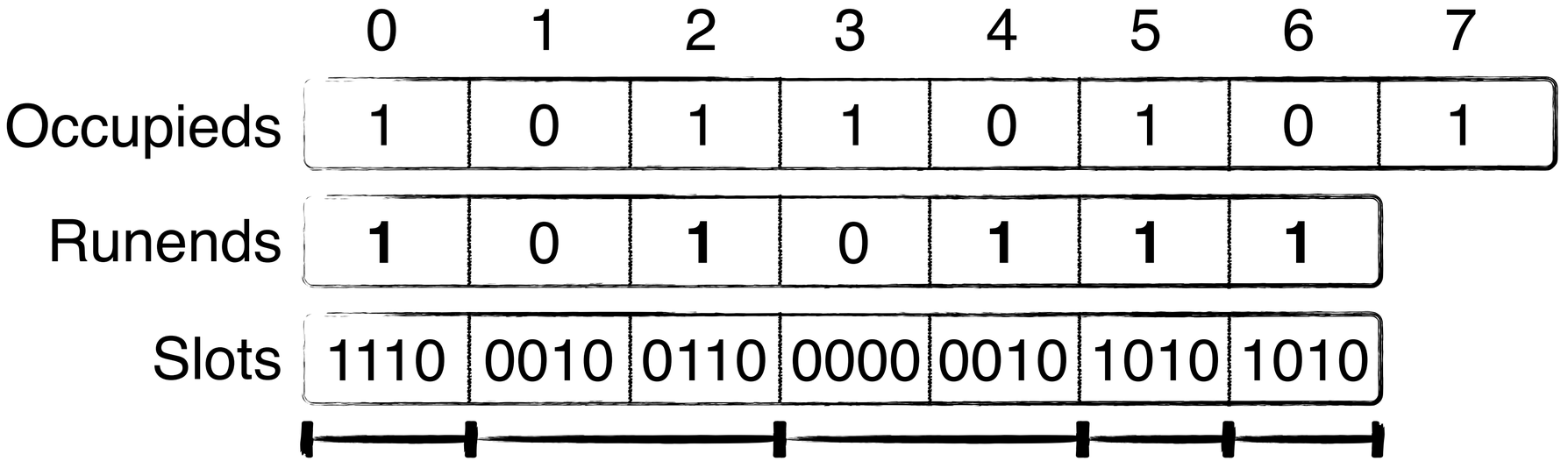
Delimit runs

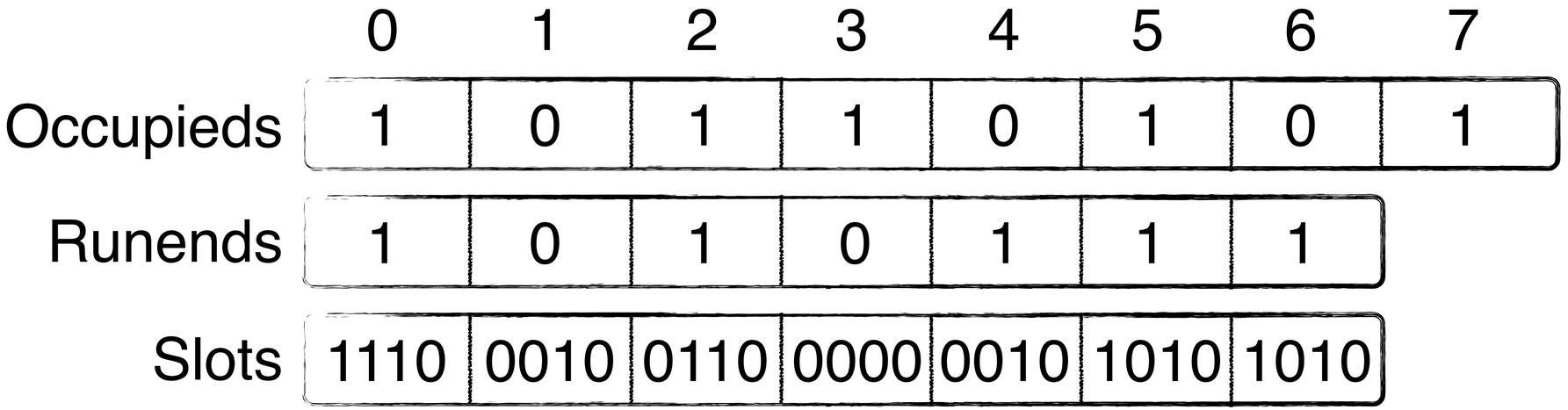




Delimit runs



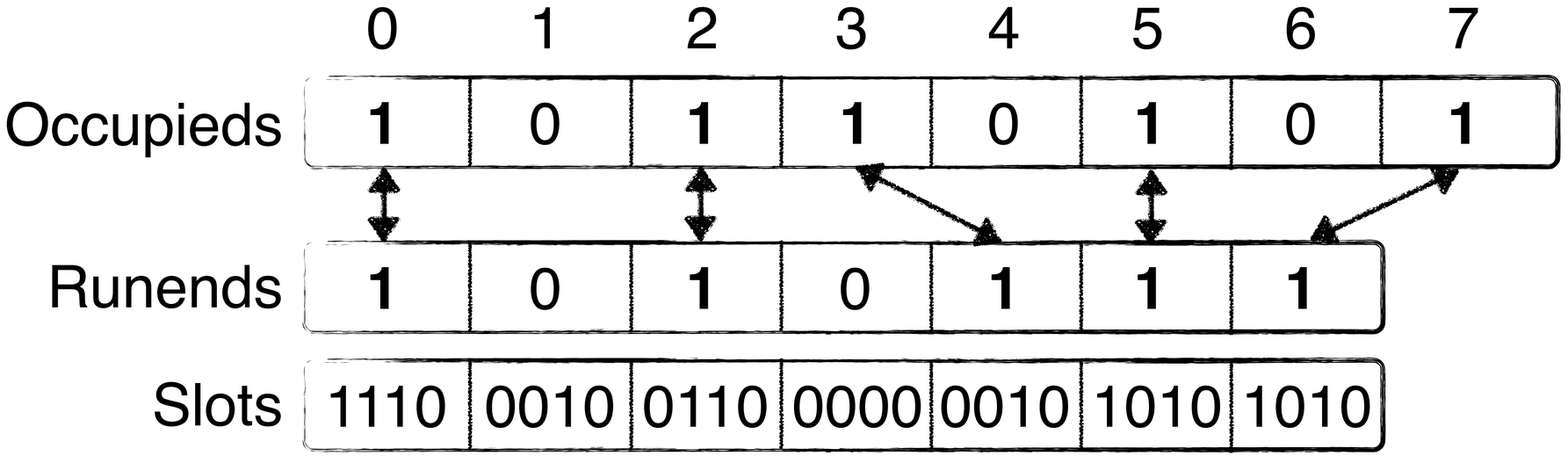

0
1
2
3
4
5
6
7
Occupieds
1
0
1
1
0
1
0
1
Runends
1
0
1
0
1
1
1
Slots
1110
0010
0110
0000
0010
1010
1010


0 1 2 3 4 5 6 7
Occupieds
1 0 1 1 0 1 0 1
Runends
1 0 1 0 1 1 1
Slots
1110 0010 0110 0000 0010 1010 1010

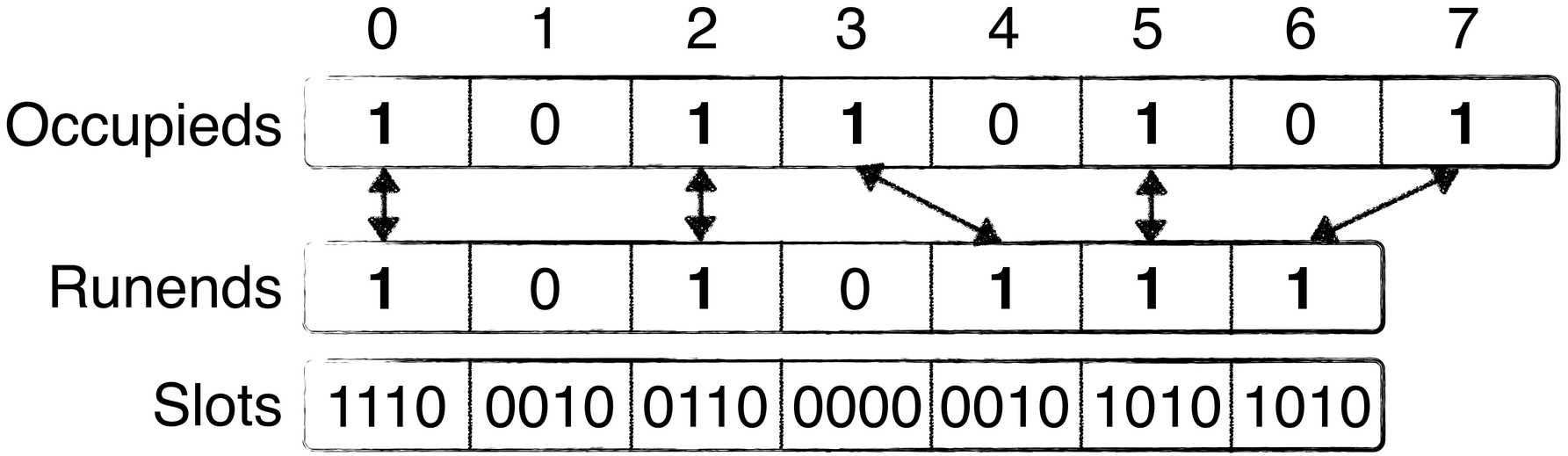
Matching 1 bits

	0	1	2	3	4	5	6	7
Occupieds	**1**	0	**1**	**1**	0	**1**	0	**1**
Runends	**1**	0	**1**	0	**1**	**1**	**1**	
Slots	1110	0010	0110	0000	0010	1010	1010	

Matching 1 bits

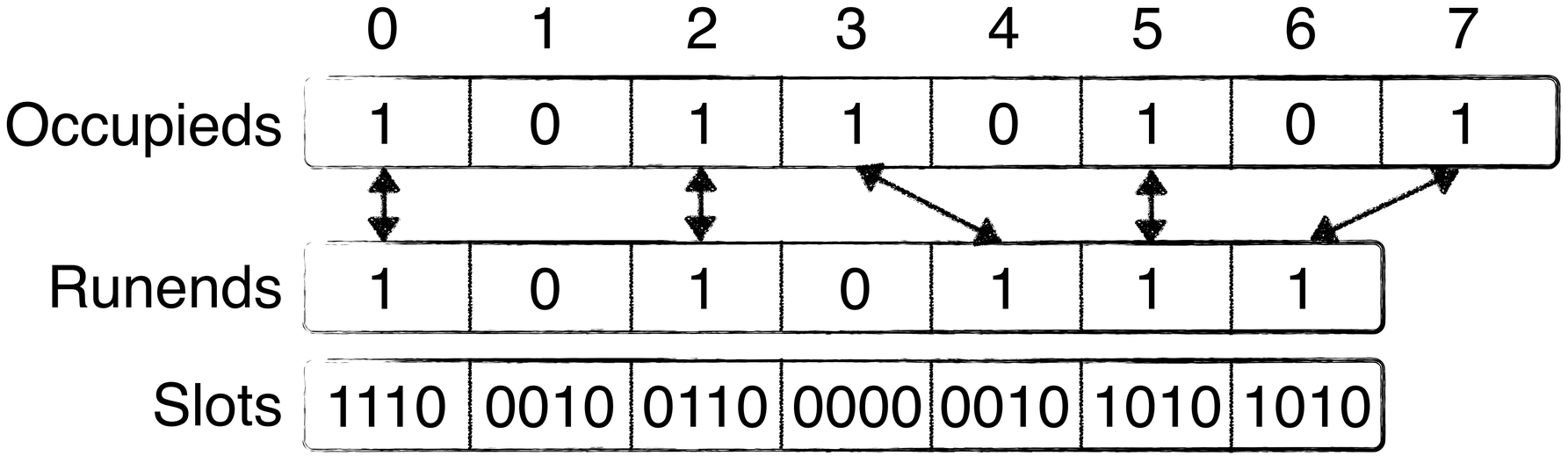


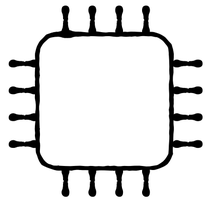
Matching 1s have equal rank



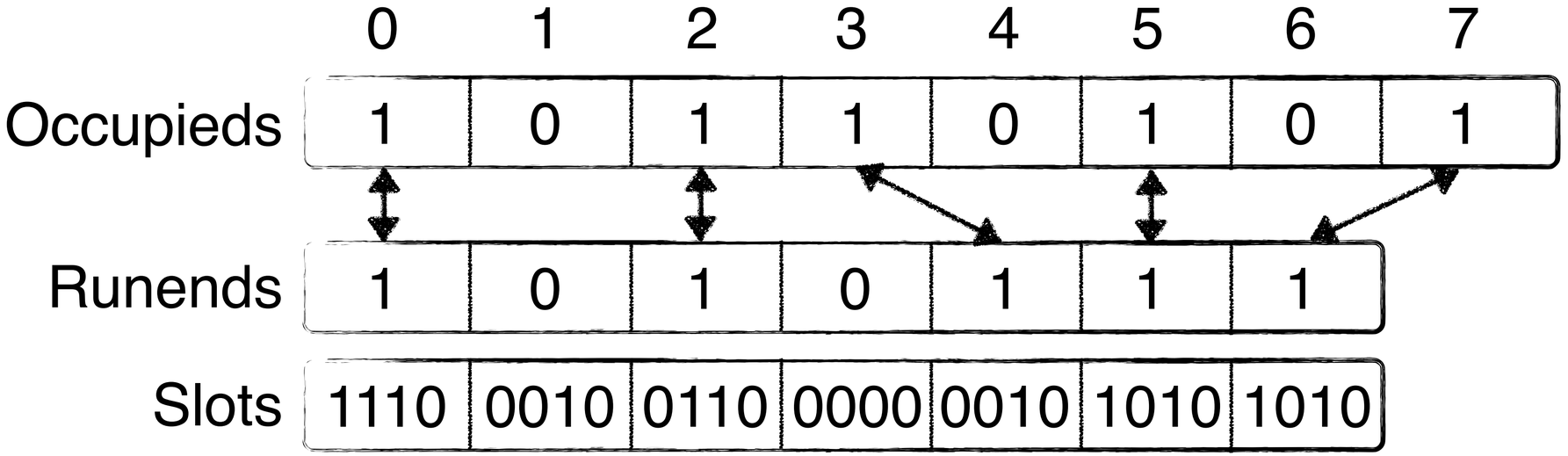
Matching 1s have equal rank

Locate using rank and select

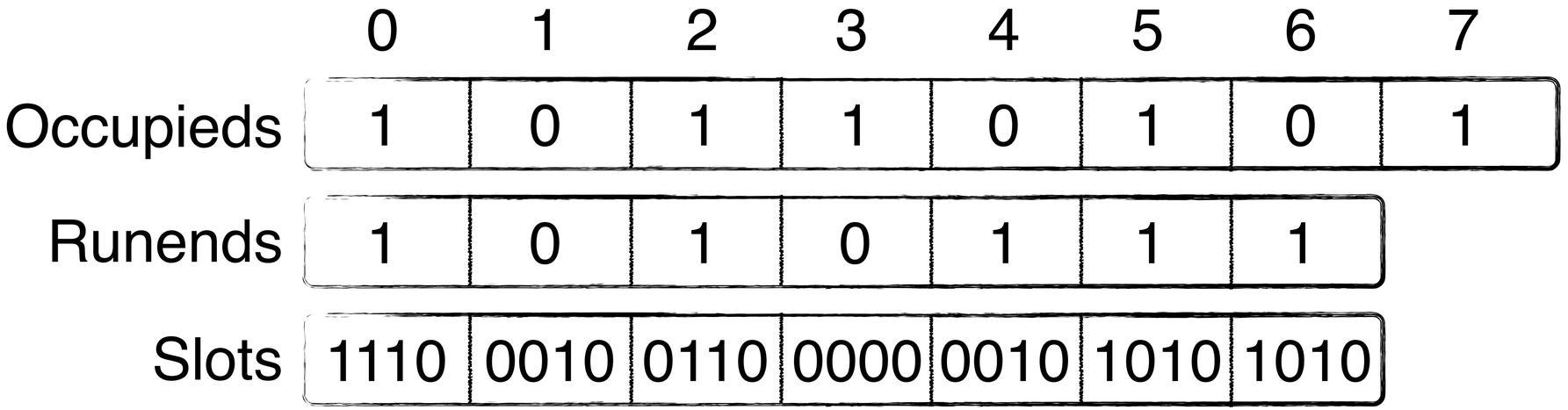




Efficient CPU implementations

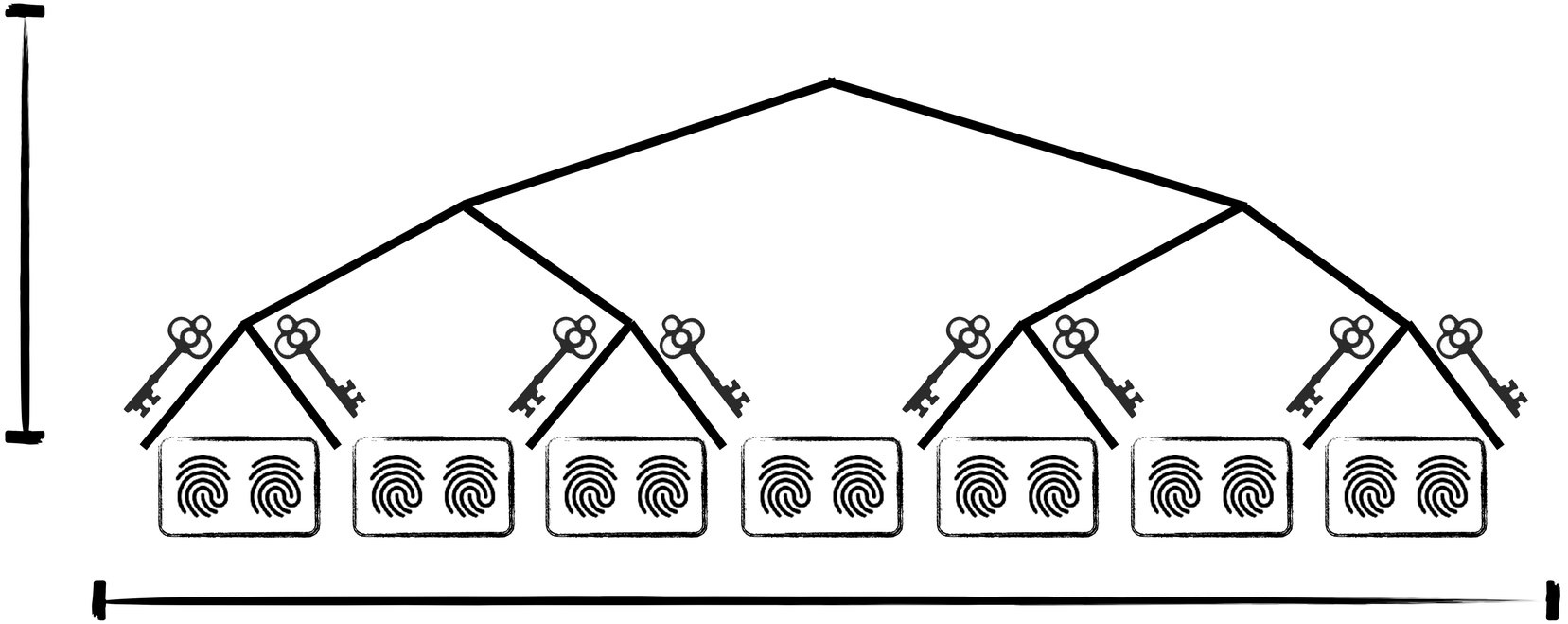


Order-perserving mini-filter with O(1) access time :)



Cached Tree

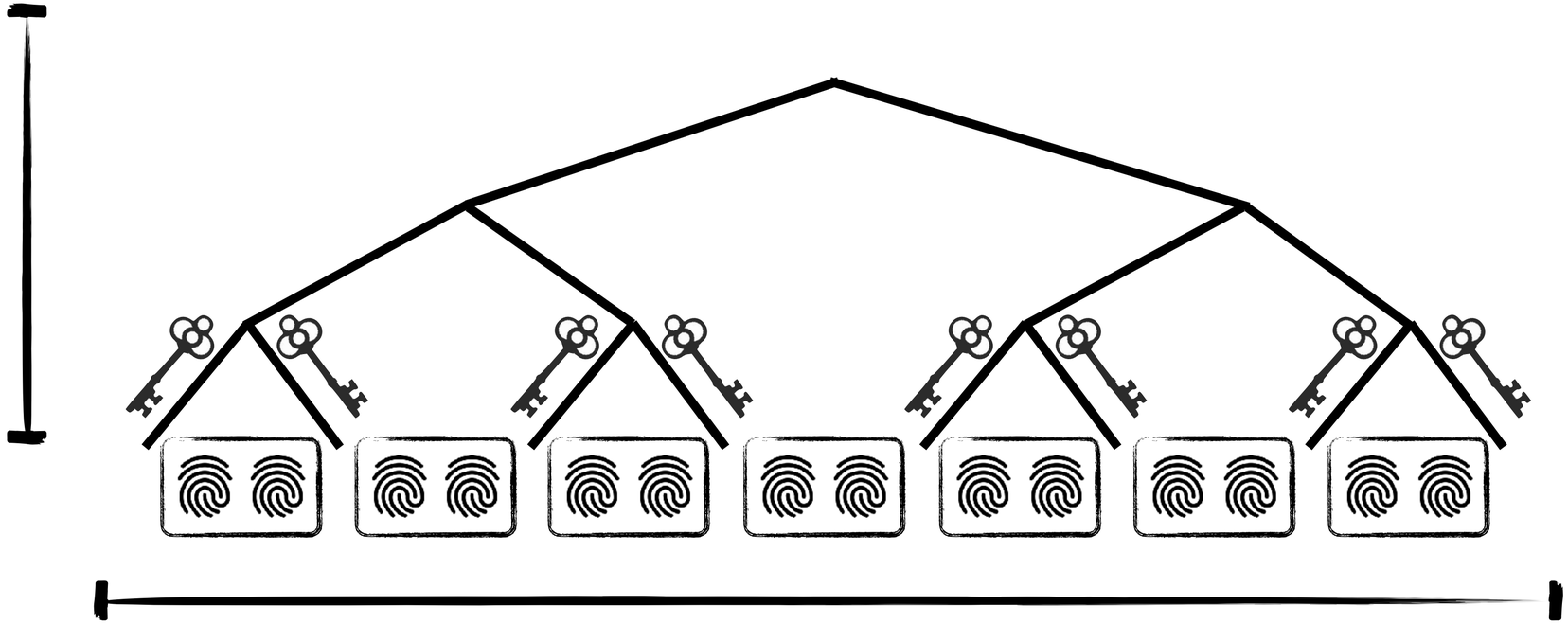
Every 1000th key



Infix stores

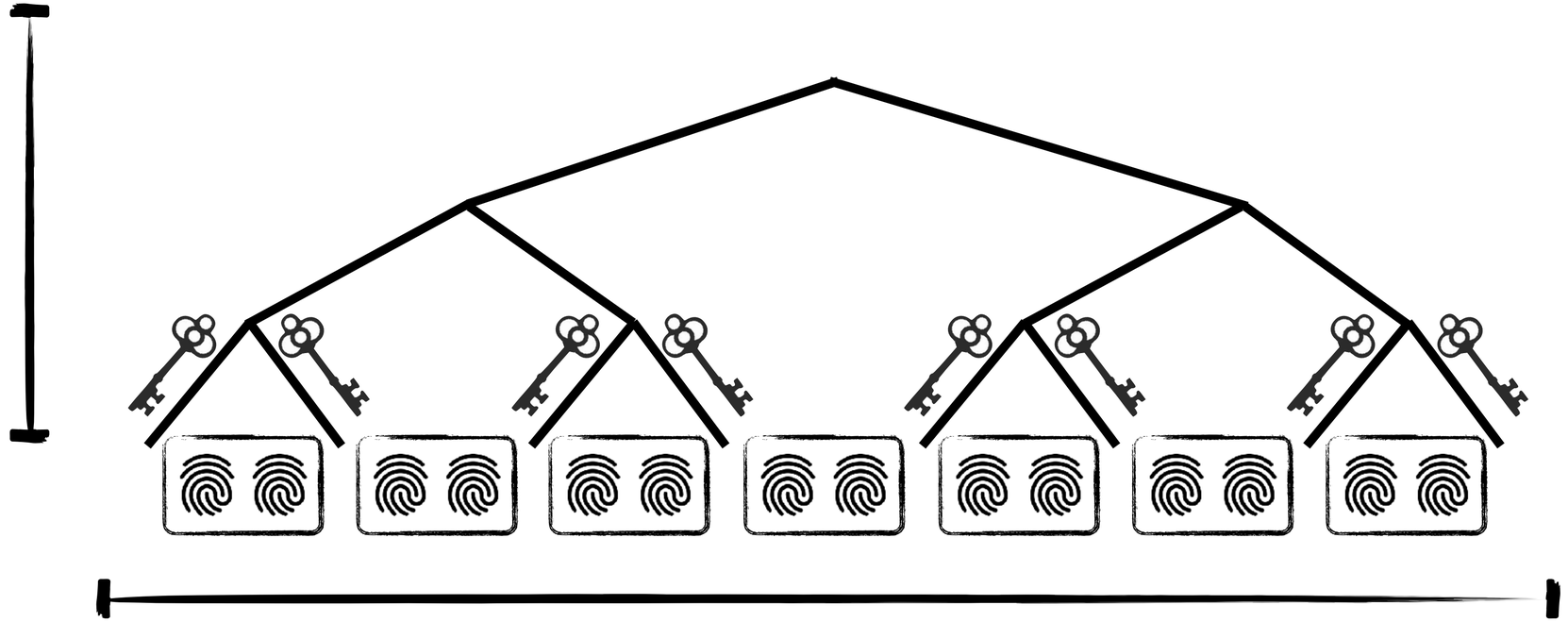
1000 keys each

Y-fast trie
O(log L)



Infix stores
1000 keys each

Y-fast trie
O(log L)

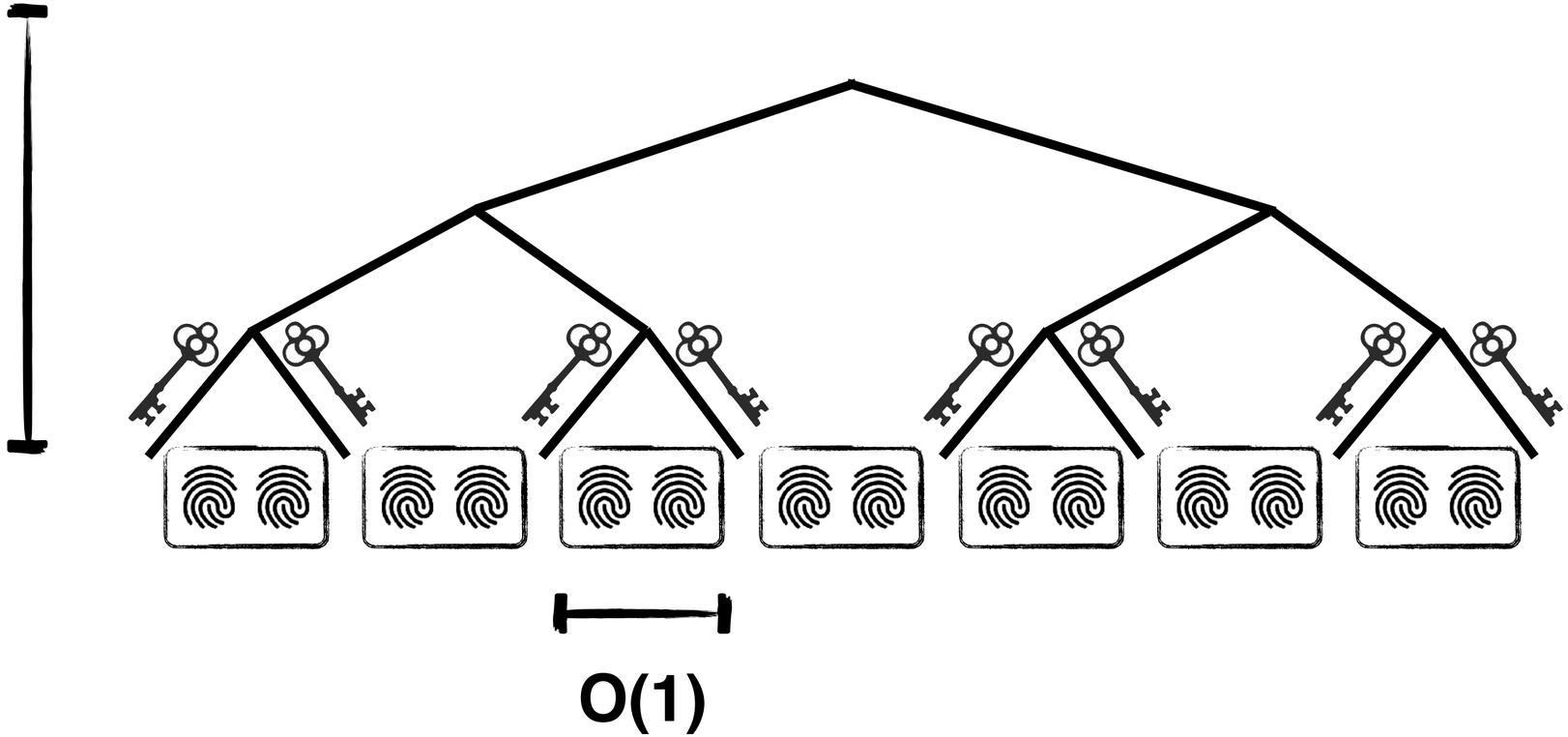


Infix stores
1000 keys each

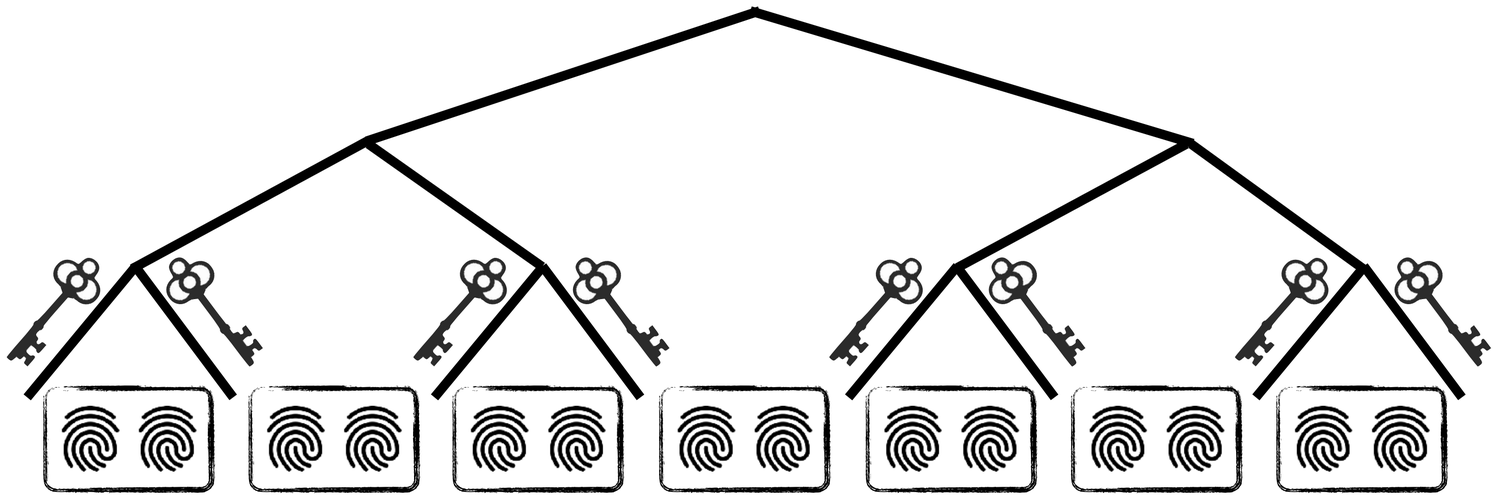
“Log-logarithmic worst-case range queries are possible in space Θ(N)”
Information Processing Letters, Vol. 17, 1983

Y-fast trie

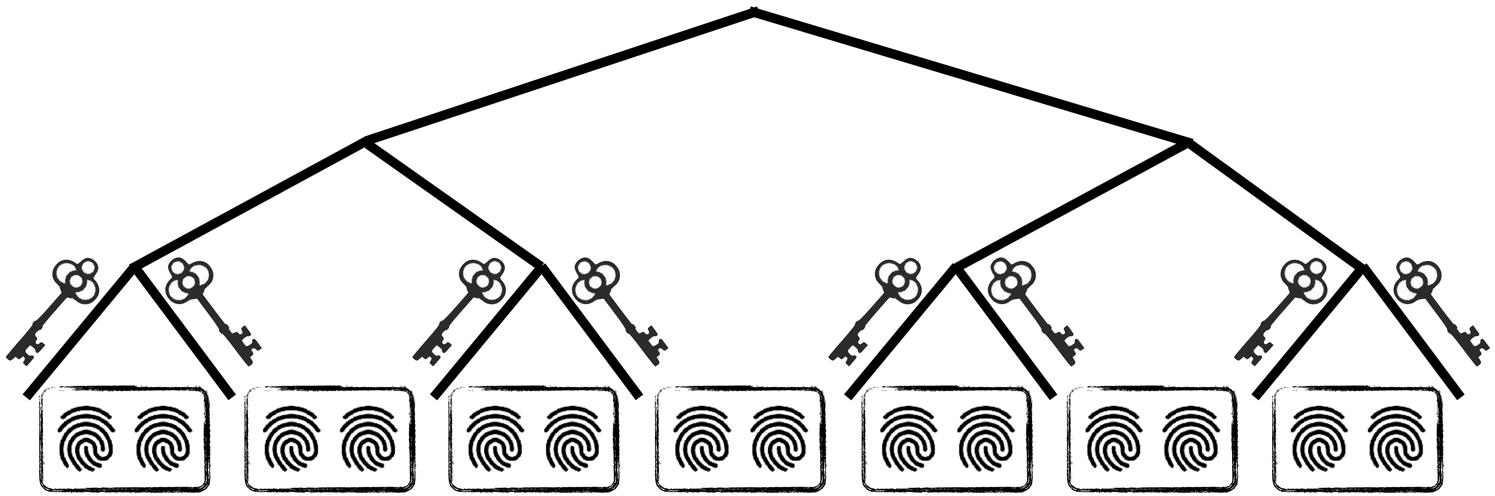
O(log L)



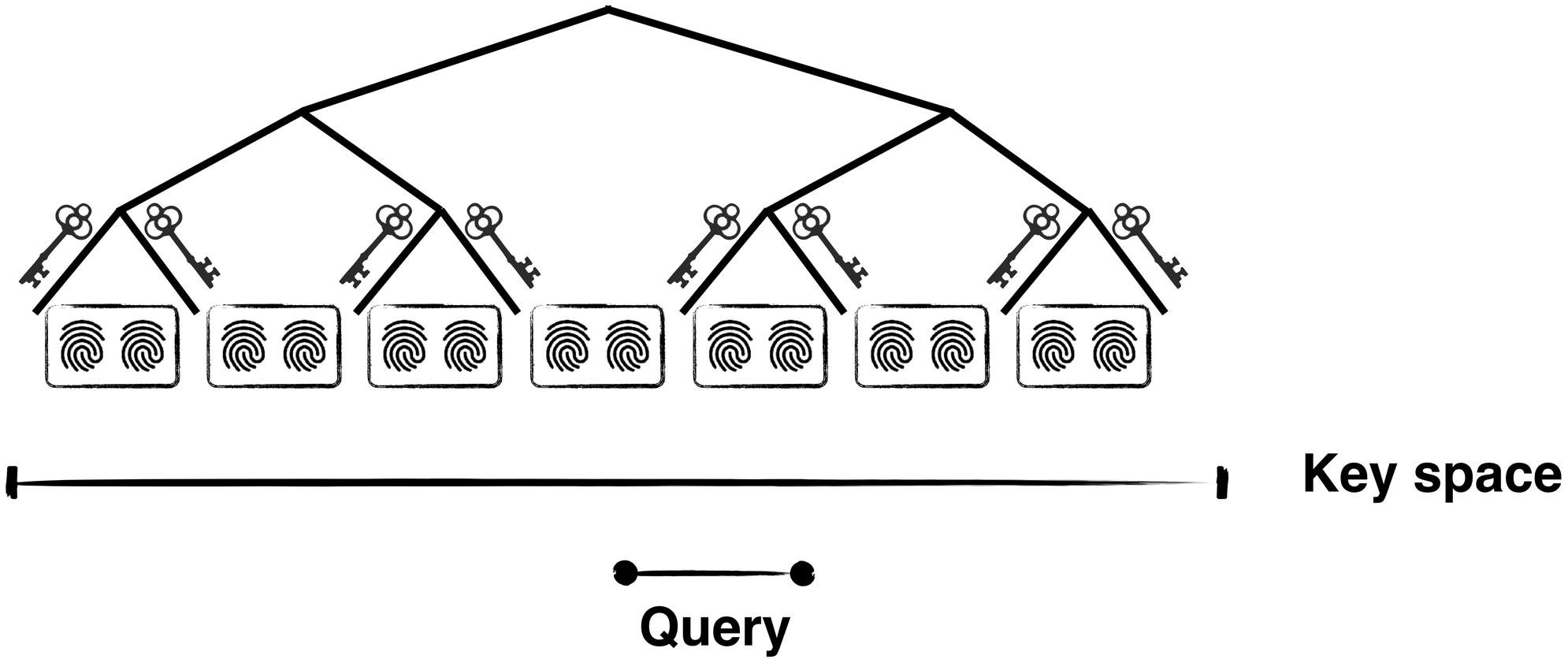
O(1)

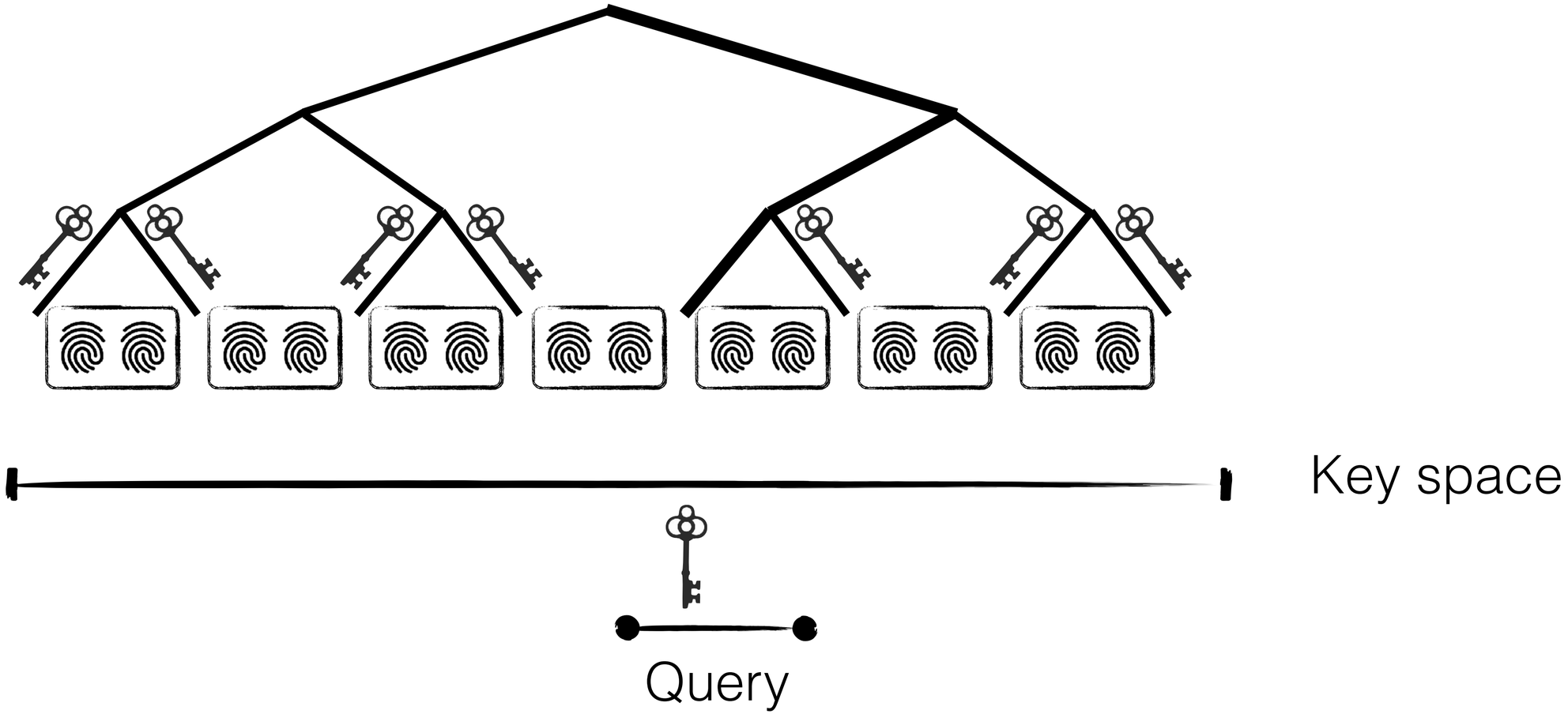


Runtime: O(log L) + O(1) = **O(log L)**

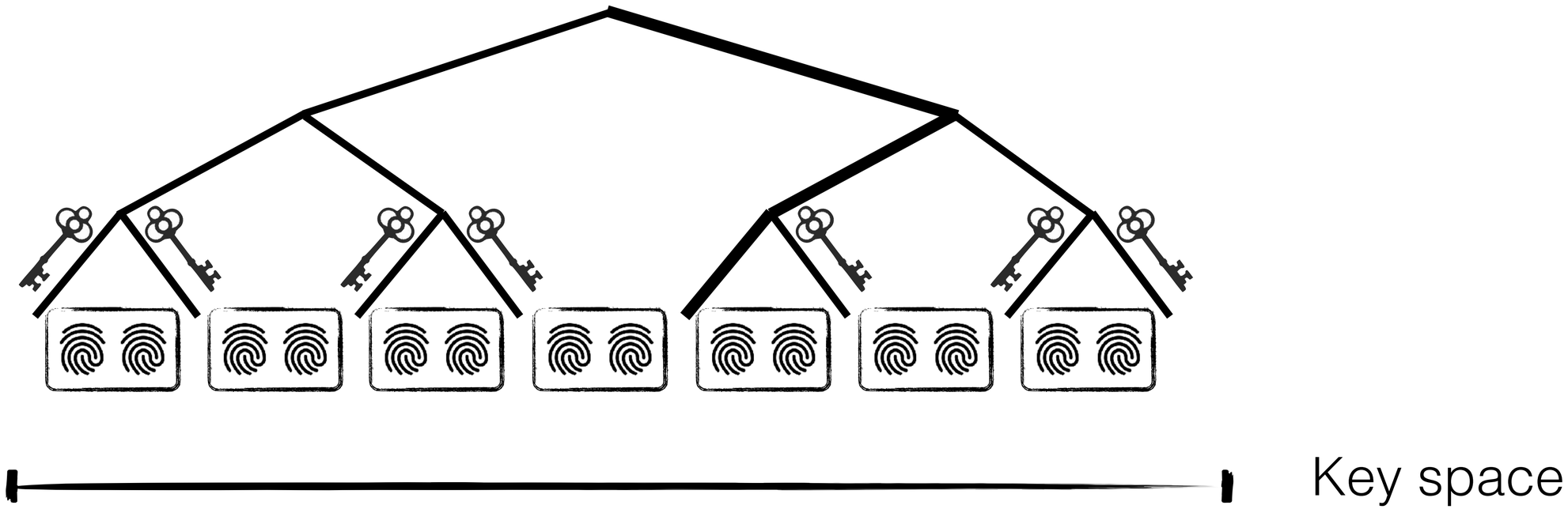


How to check if a range is empty?

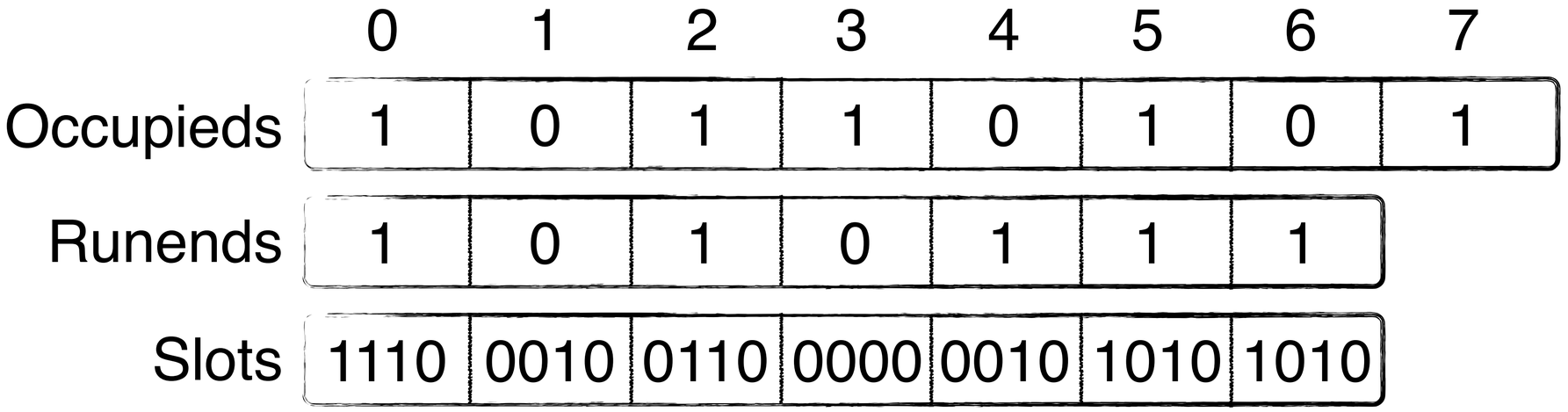

Key space
Query



Return true positive


Key space
Query

Query

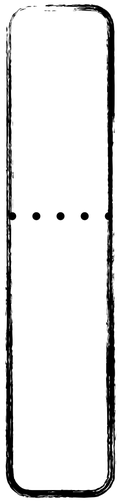

0 1 2 3 4 5 6 7
Occupieds
1 0 1 1 0 1 0 1
Runends
1 0 1 0 1 1 1
Slots
1110 0010 0110 0000 0010 1010 1010

Query

Compress endpoints

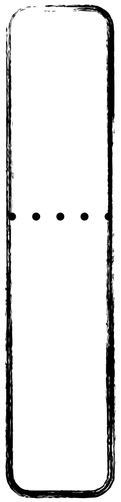
	0	1	2	3	4	5	6	7
Occupieds	1	0	1	1	0	1	0	1
Runends	1	0	1	0	1	1	1	
Slots	1110	0010	0110	0000	0010	1010	1010	

	0	1	2	3	4	5	6	7
Occupieds	1	0	1	1	0	1	0	1
Runends	1	0	1	0	1	1	1	
Slots	1110	0010	0110	0000	0010	1010	1010	

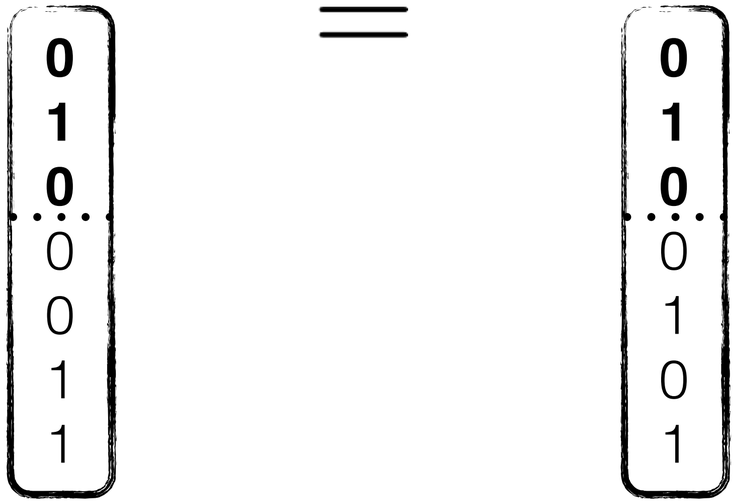


Quotient cases:

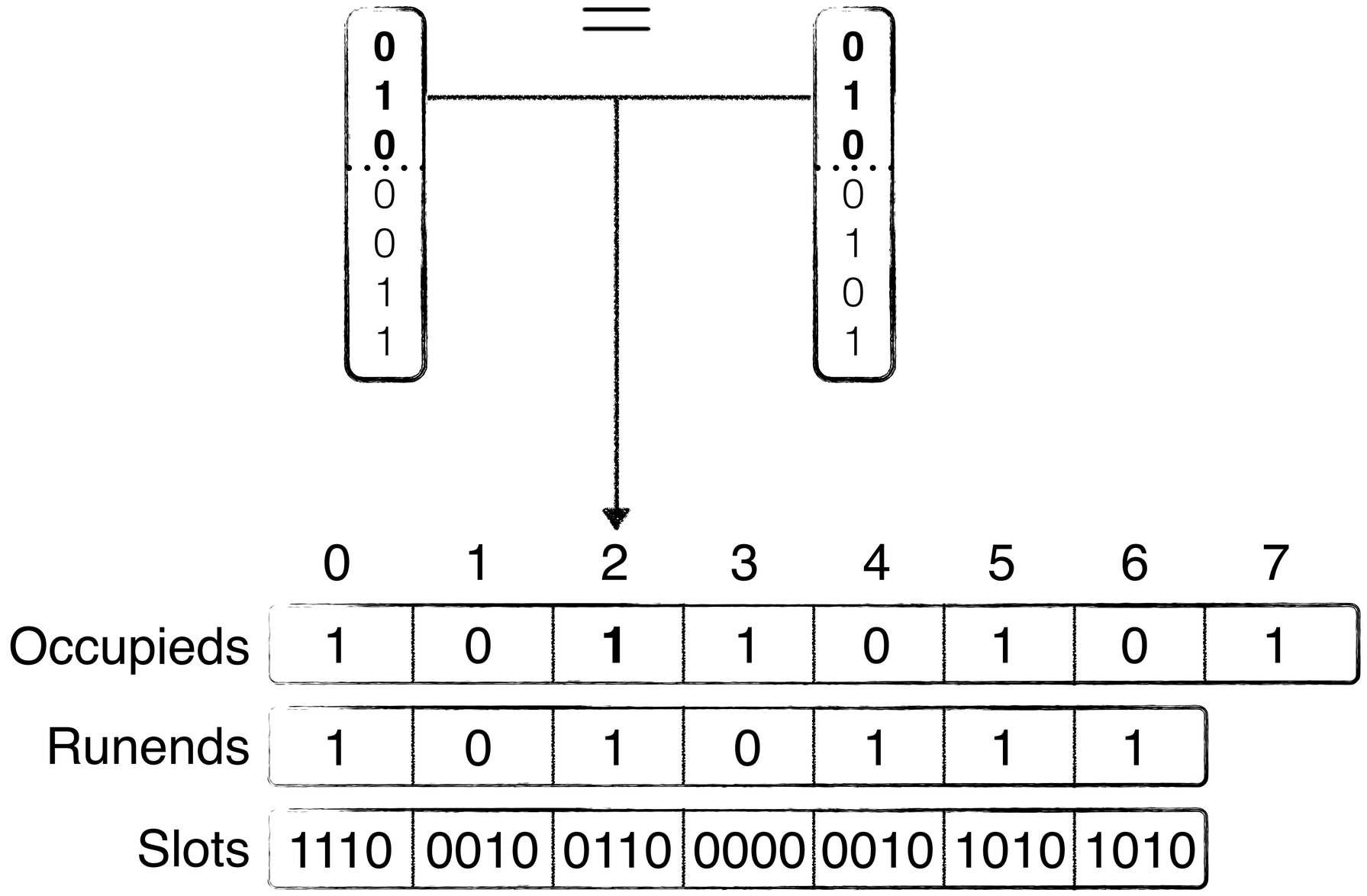
$=$ $\neq$

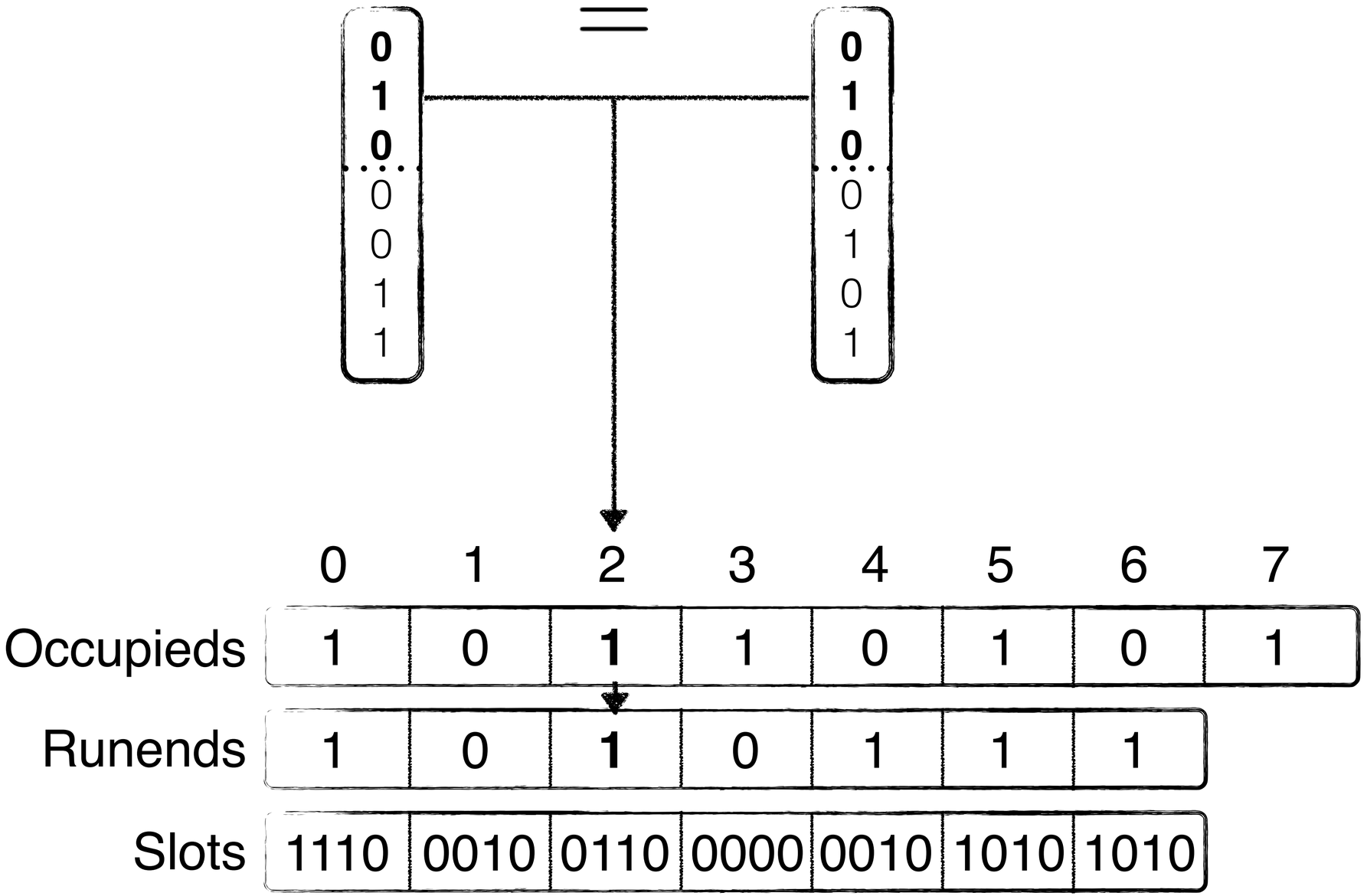


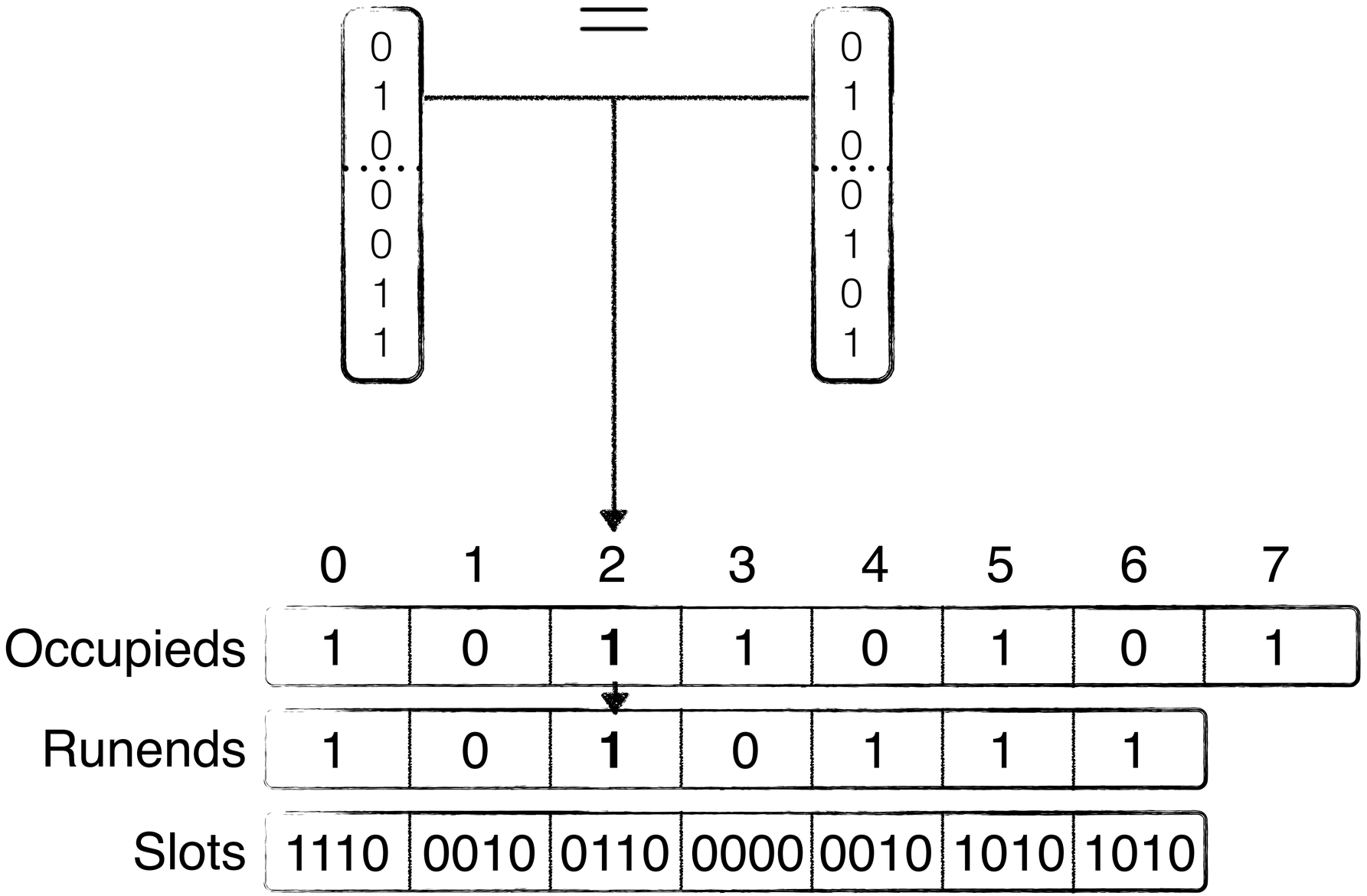
	0	1	2	3	4	5	6	7
Occupieds	1	0	1	1	0	1	0	1
Runends	1	0	1	0	1	1	1	
Slots	1110	0010	0110	0000	0010	1010	1010	

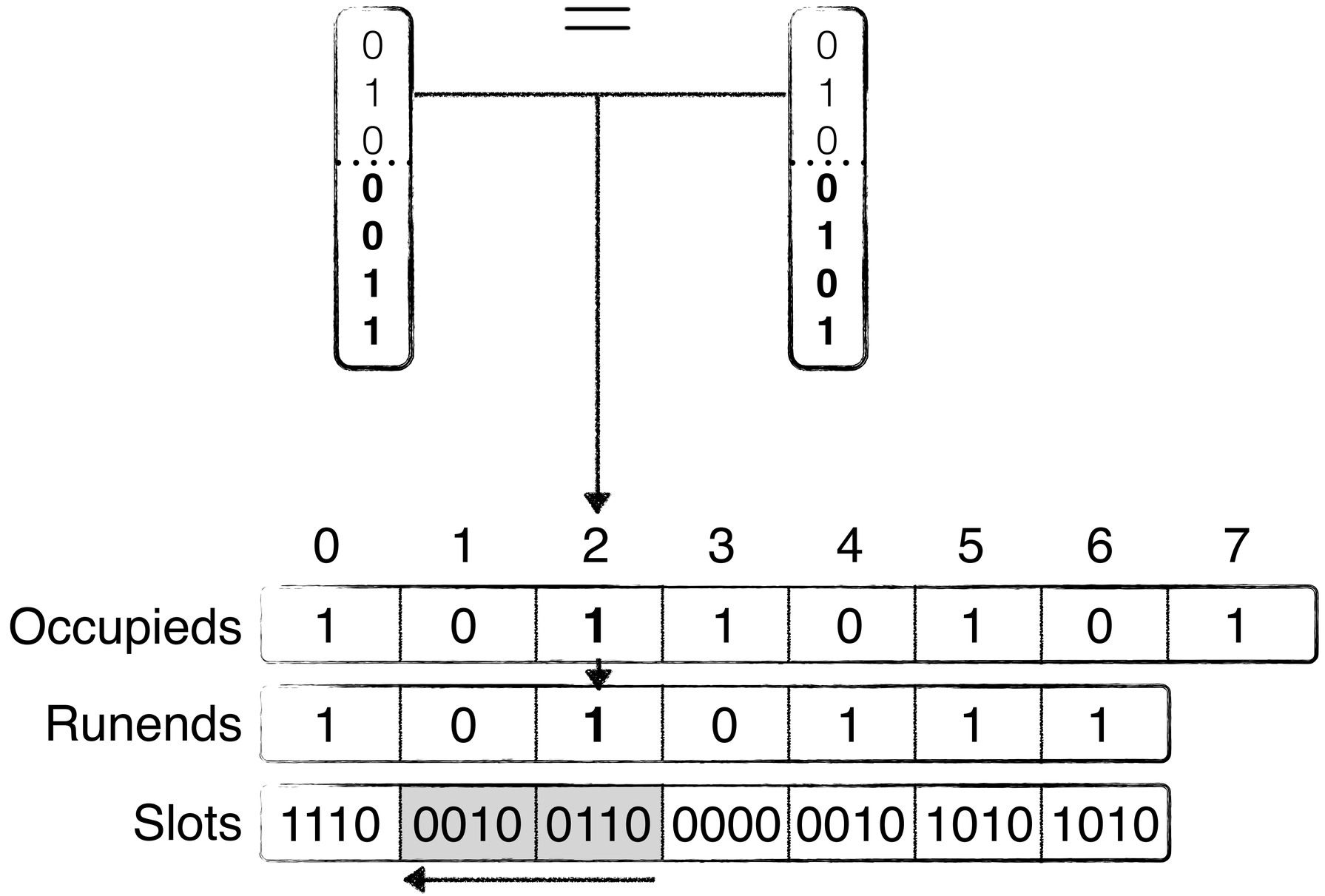


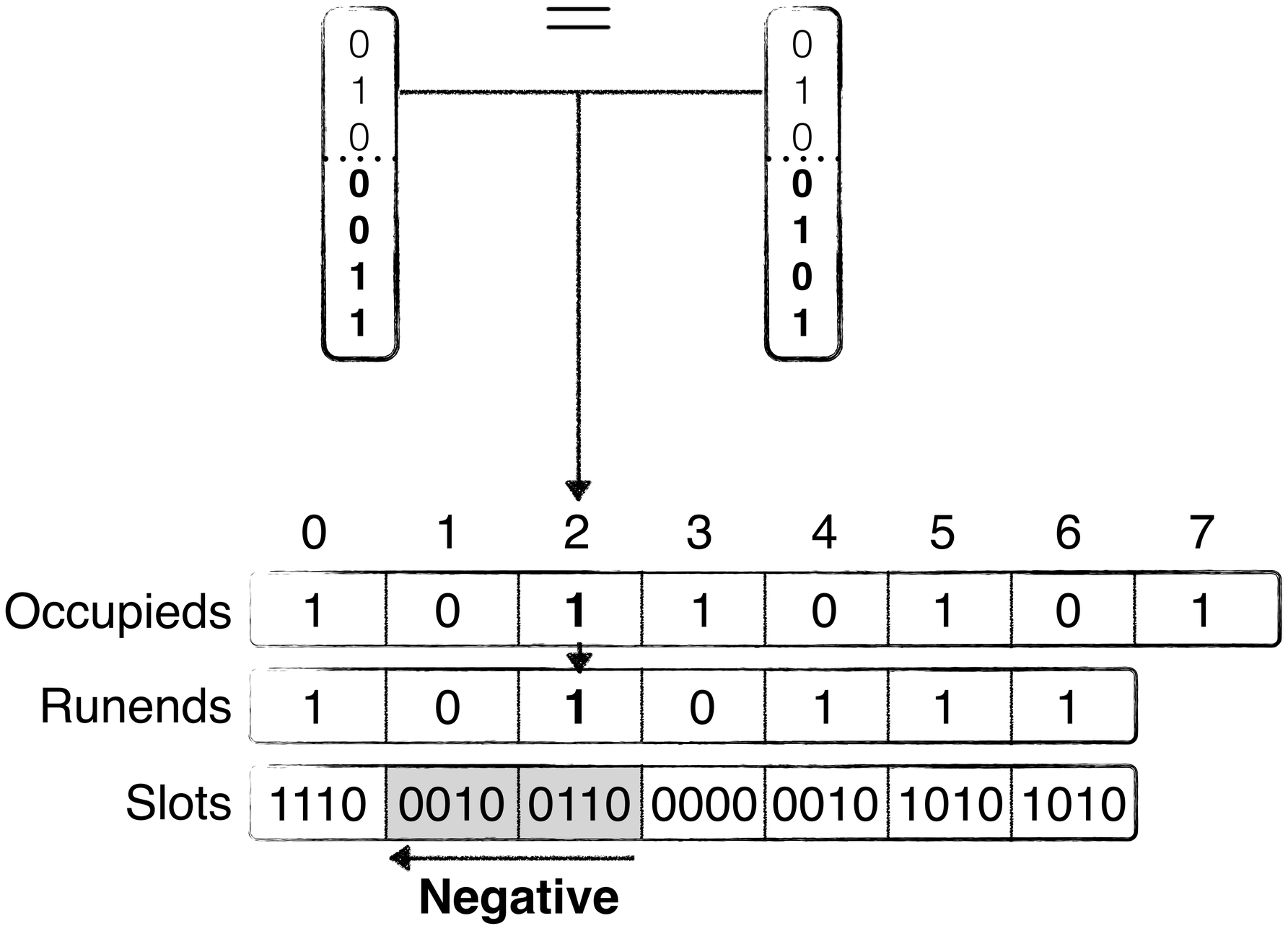
	0	1	2	3	4	5	6	7
Occupieds	1	0	1	1	0	1	0	1
Runends	1	0	1	0	1	1	1	
Slots	1110	0010	0110	0000	0010	1010	1010	


0
1
0
0
0
1
1
0
1
0
0
1
0
1
0 1 2 3 4 5 6 7
Occupieds 1 0 1 1 0 1 0 1
Runends 1 0 1 0 1 1 1
Slots 1110 0010 0110 0000 0010 1010 1010

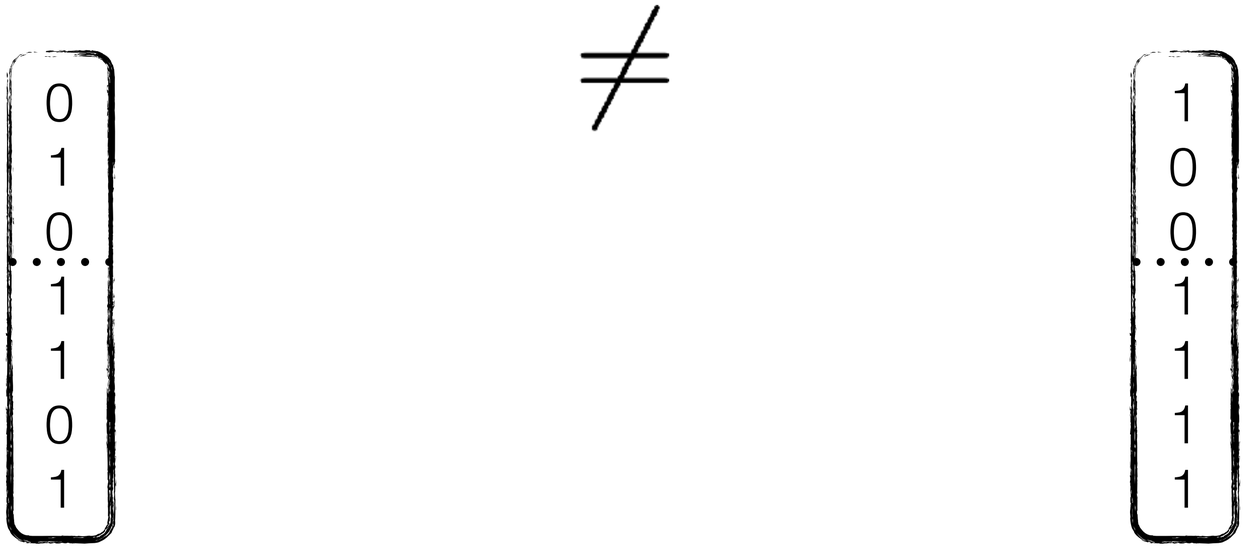

0
1
0
0
0
1
1
0
1
0
0
1
0
1
0 1 2 3 4 5 6 7
Occupieds
1 0 1 1 0 1 0 1
Runends
1 0 1 0 1 1 1
Slots
1110 0010 0110 0000 0010 1010 1010


0
1
0
0
0
1
1
0
1
0
0
1
0
1
0 1 2 3 4 5 6 7
Occupieds 1 0 1 1 0 1 0 1
Runends 1 0 1 0 1 1 1
Slots 1110 0010 0110 0000 0010 1010 1010

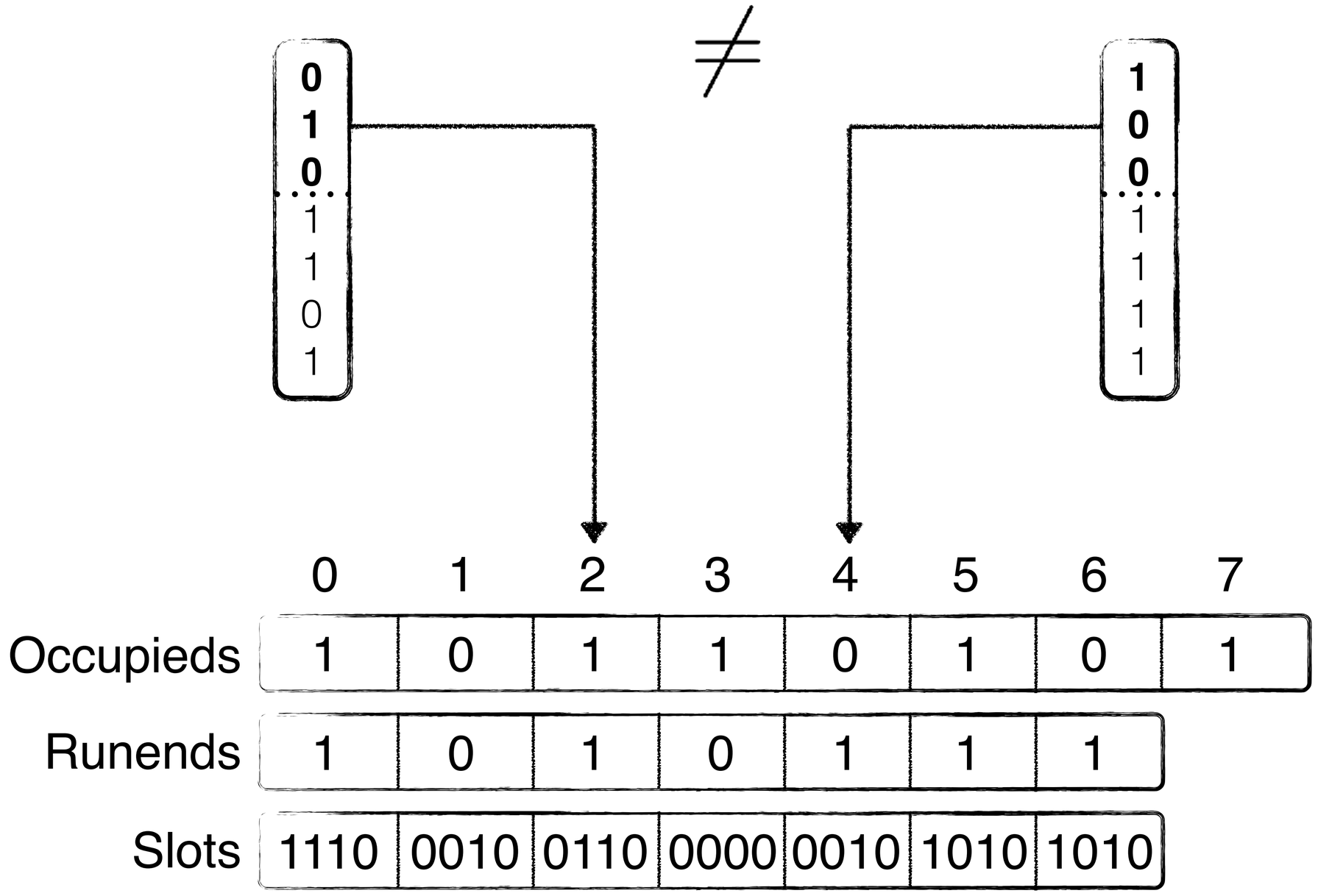

0
1
0
0
0
1
1
0
1
0
0
1
0
1
0 1 2 3 4 5 6 7
Occupieds
1 0 1 1 0 1 0 1
Runends
1 0 1 0 1 1 1
Slots
1110 0010 0110 0000 0010 1010 1010

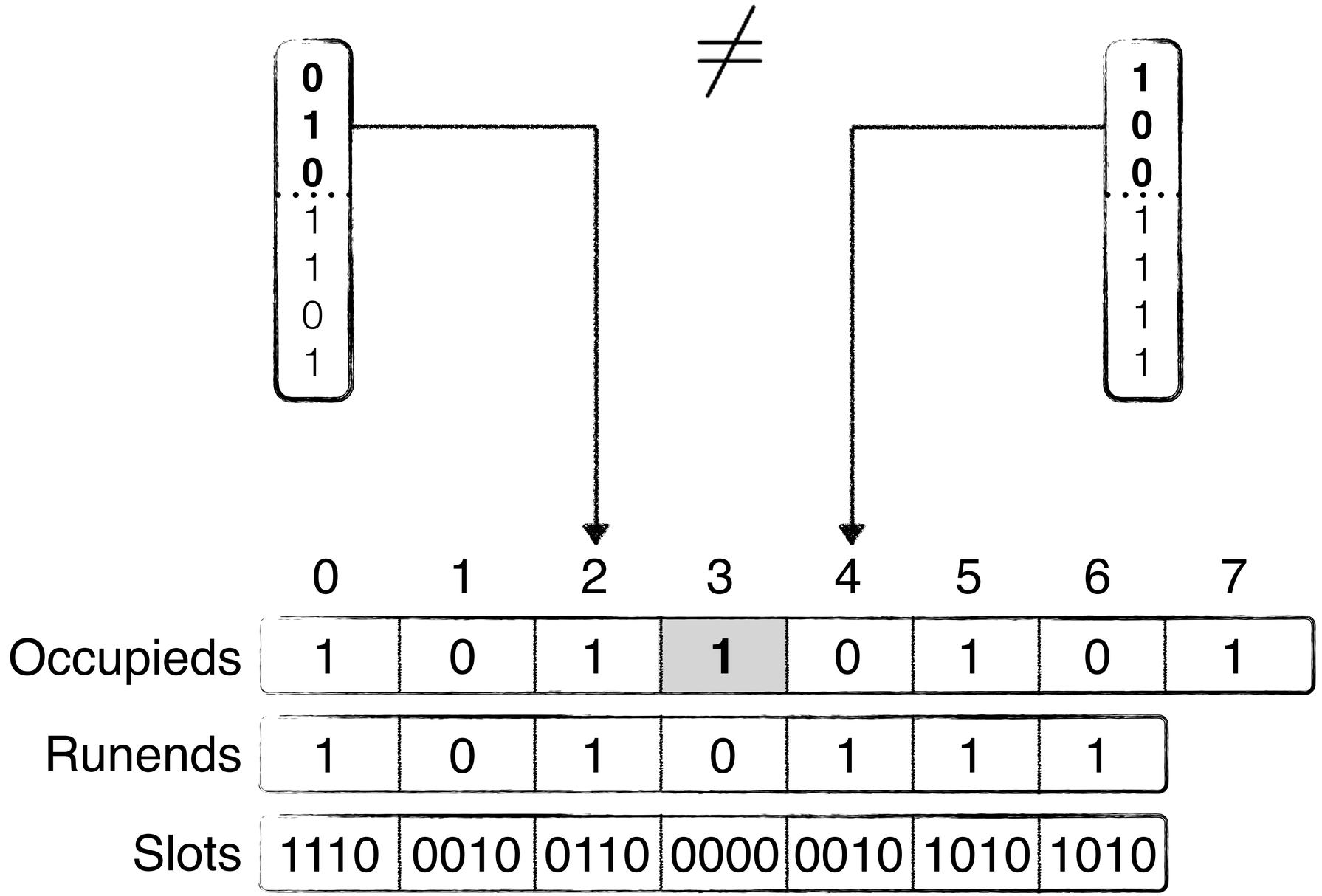

0
1
0
0
0
1
1
=
0
1
0
0
1
0
1
0 1 2 3 4 5 6 7
Occupieds
1 0 1 1 0 1 0 1
Runends
1 0 1 0 1 1 1
Slots
1110 0010 0110 0000 0010 1010 1010
Negative

	0	1	2	3	4	5	6	7
Occupieds	1	0	1	1	0	1	0	1
Runends	1	0	1	0	1	1	1	
Slots	1110	0010	0110	0000	0010	1010	1010	

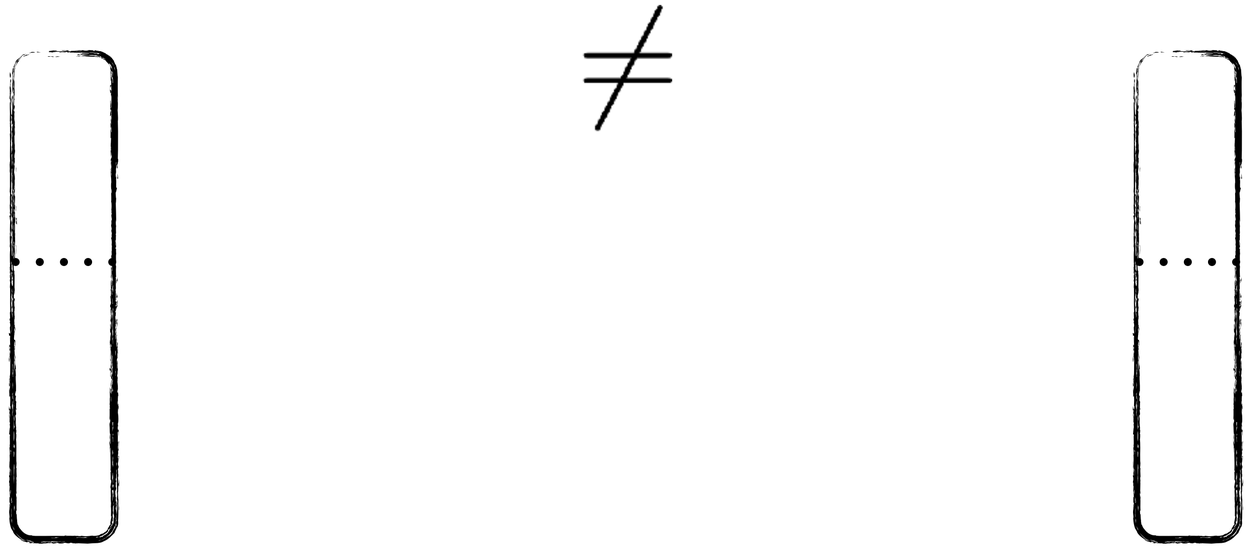


	0	1	2	3	4	5	6	7
Occupieds	1	0	1	1	0	1	0	1
Runends	1	0	1	0	1	1	1	
Slots	1110	0010	0110	0000	0010	1010	1010	

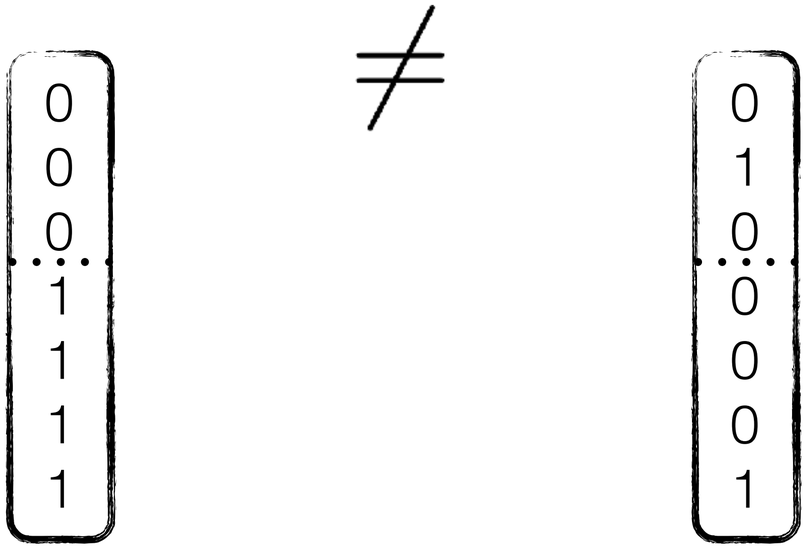

0
1
0
1
1
0
1
≠
1
0
0
1
1
1
1
0 1 2 3 4 5 6 7
Occupieds 1 0 1 1 0 1 0 1
Runends 1 0 1 0 1 1 1
Slots 1110 0010 0110 0000 0010 1010 1010



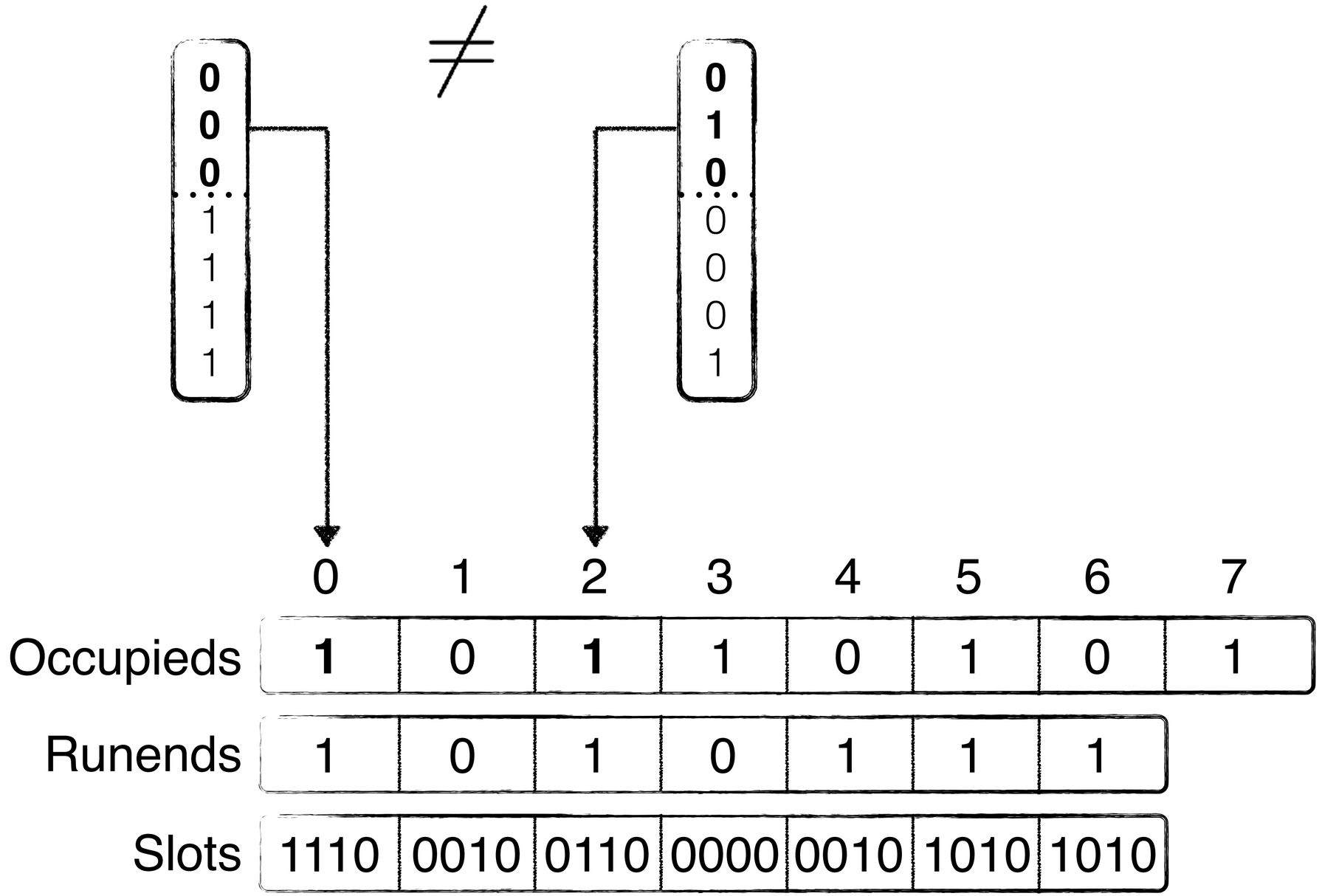
Run strictly in range: true positive

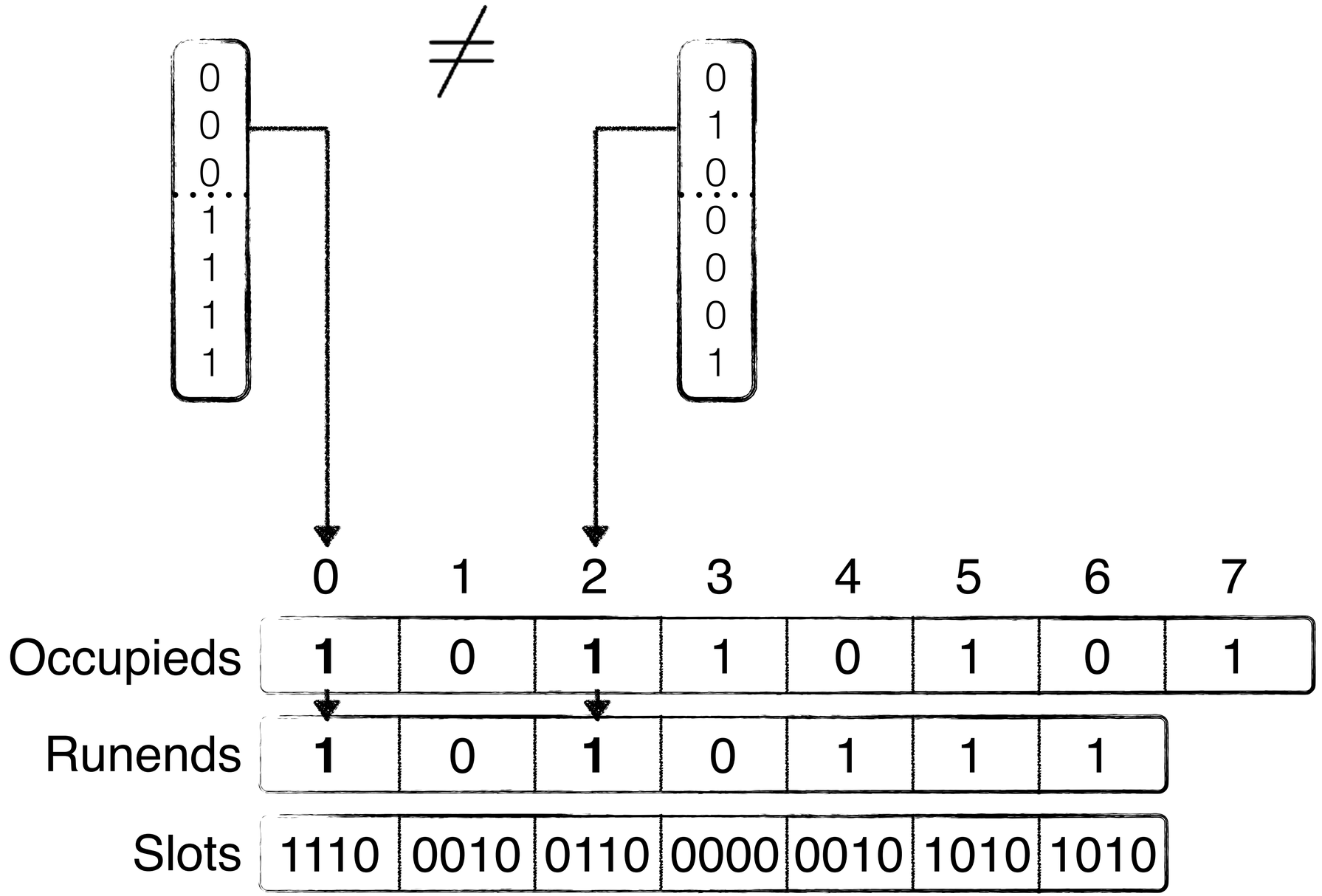


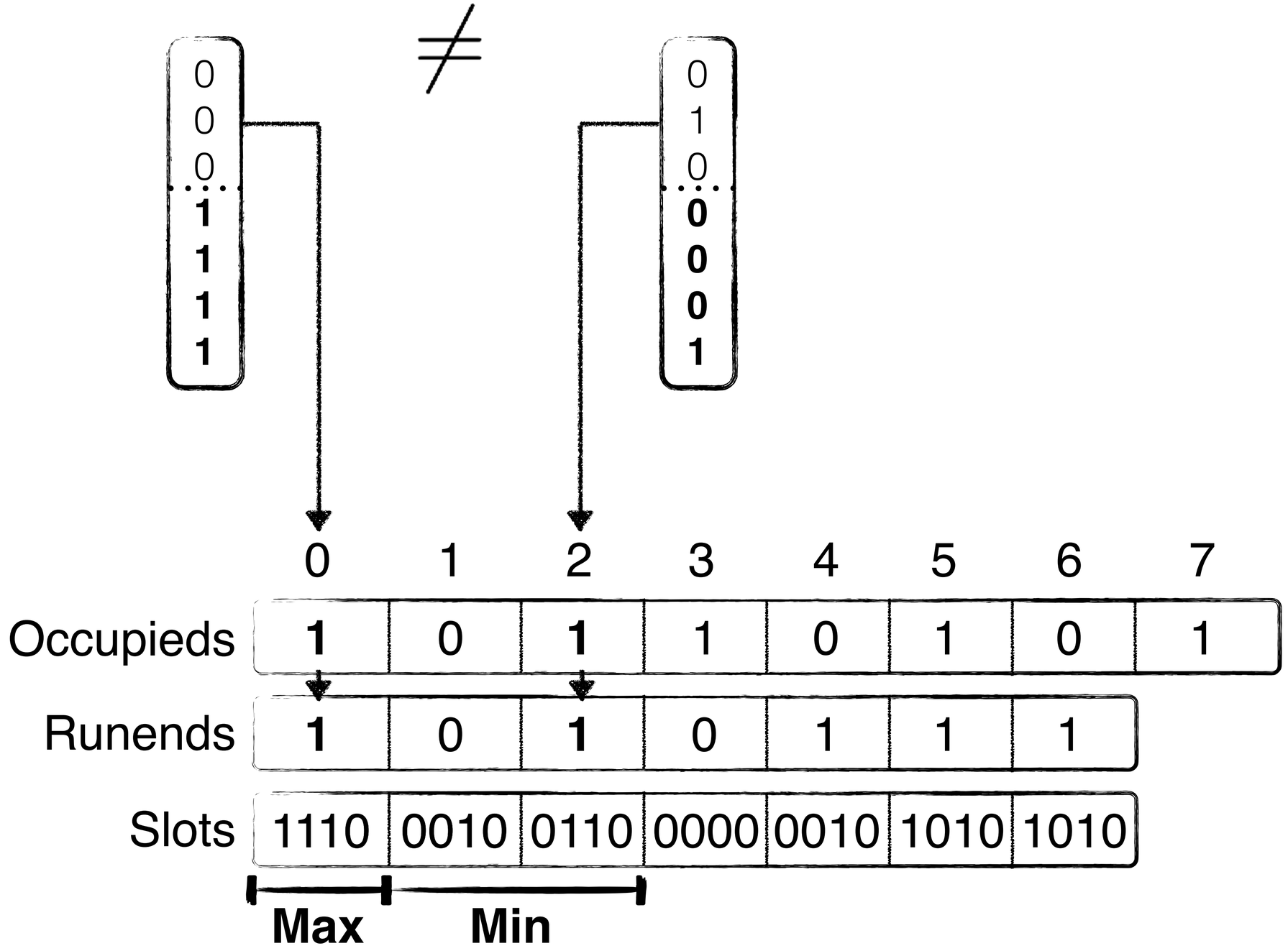
	0	1	2	3	4	5	6	7
Occupieds	1	0	1	1	0	1	0	1
Runends	1	0	1	0	1	1	1	
Slots	1110	0010	0110	0000	0010	1010	1010	

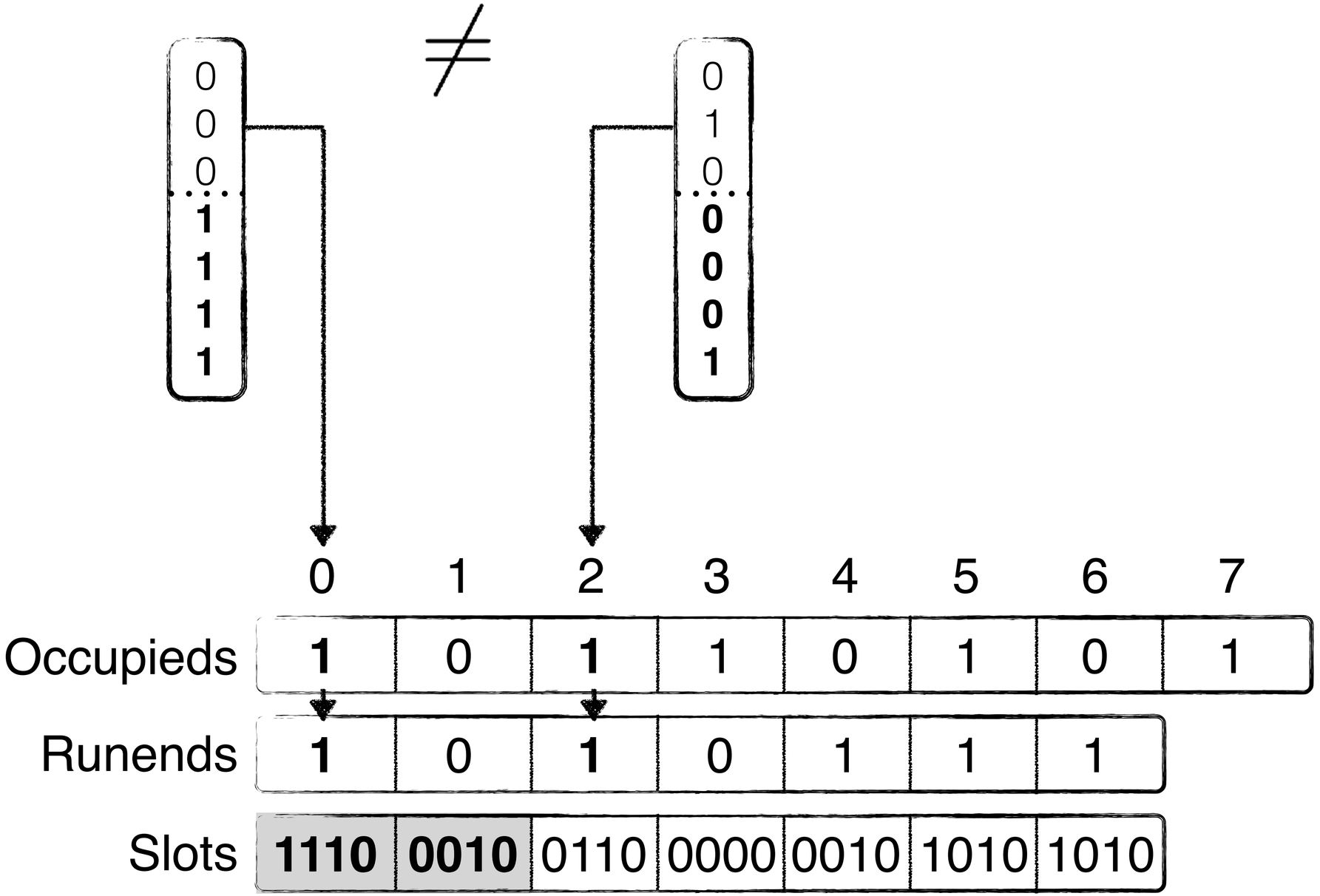


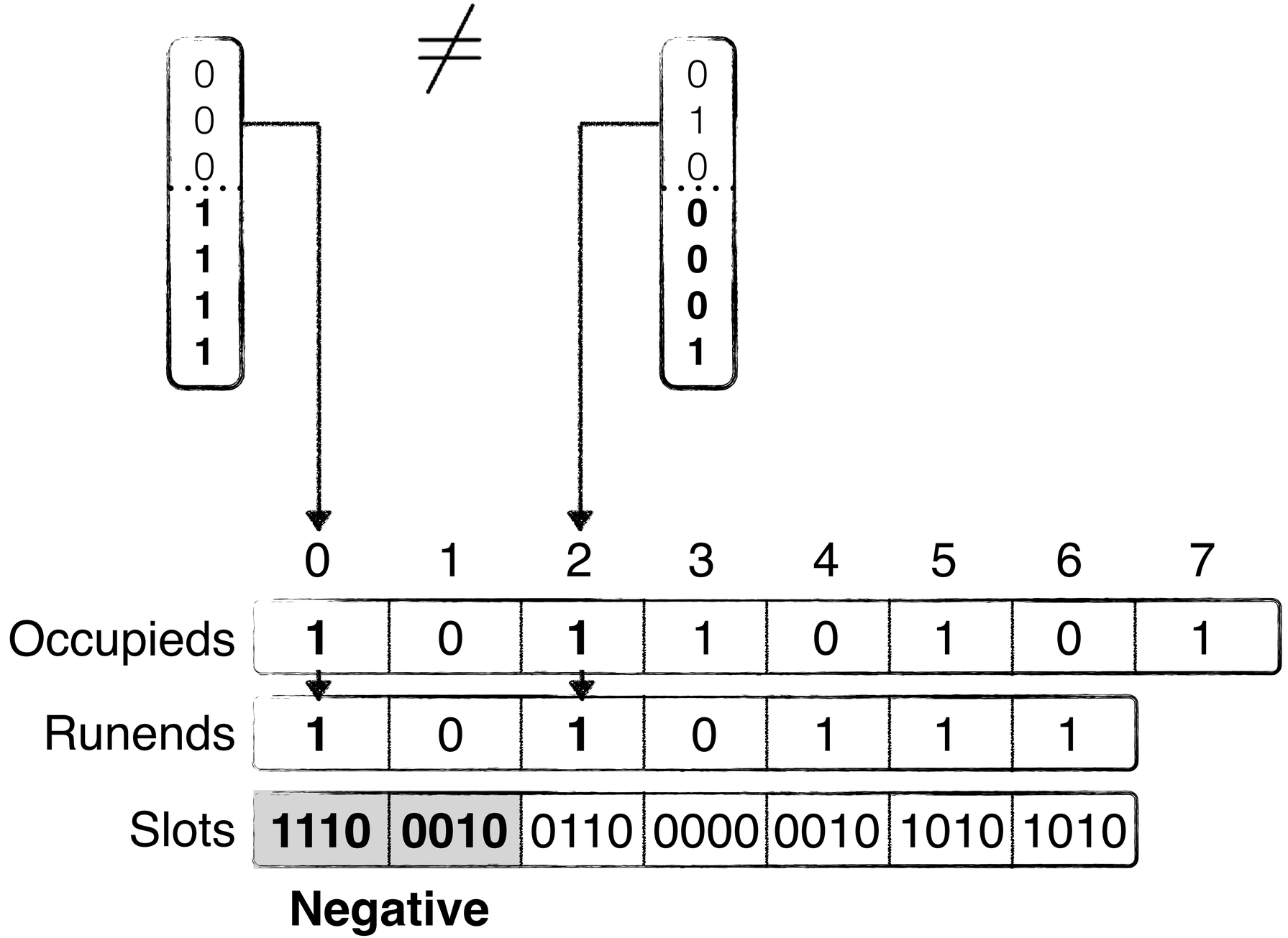
	0	1	2	3	4	5	6	7
Occupieds	1	0	1	1	0	1	0	1
Runends	1	0	1	0	1	1	1	
Slots	1110	0010	0110	0000	0010	1010	1010	


0
0
0
1
1
1
1
≠
0
1
0
0
0
0
1
0 1 2 3 4 5 6 7
Occupieds 1 0 1 1 0 1 0 1
Runends 1 0 1 0 1 1 1
Slots 1110 0010 0110 0000 0010 1010 1010

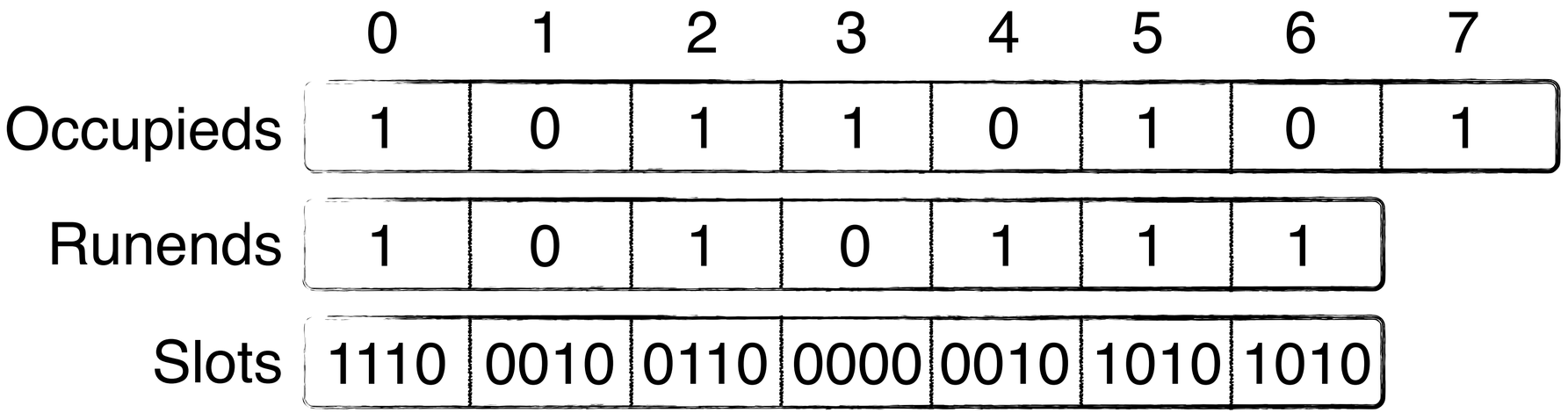

0
0
0
1
1
1
1
≠
0
1
0
0
0
0
1
0 1 2 3 4 5 6 7
Occupieds 1 0 1 1 0 1 0 1
Runends 1 0 1 0 1 1 1
Slots 1110 0010 0110 0000 0010 1010 1010


0
0
0
1
1
1
1
1
≠
0
1
0
0
0
0
0
1
0 1 2 3 4 5 6 7
Occupieds 1 0 1 1 0 1 0 1
Runends 1 0 1 0 1 1 1
Slots 1110 0010 0110 0000 0010 1010 1010
Max
Min


0
0
0
1
1
1
1
1
≠
0
1
0
0
0
0
0
1
0 1 2 3 4 5 6 7
Occupieds
1 0 1 1 0 1 0 1
Runends
1 0 1 0 1 1 1
Slots
1110 0010 0110 0000 0010 1010 1010

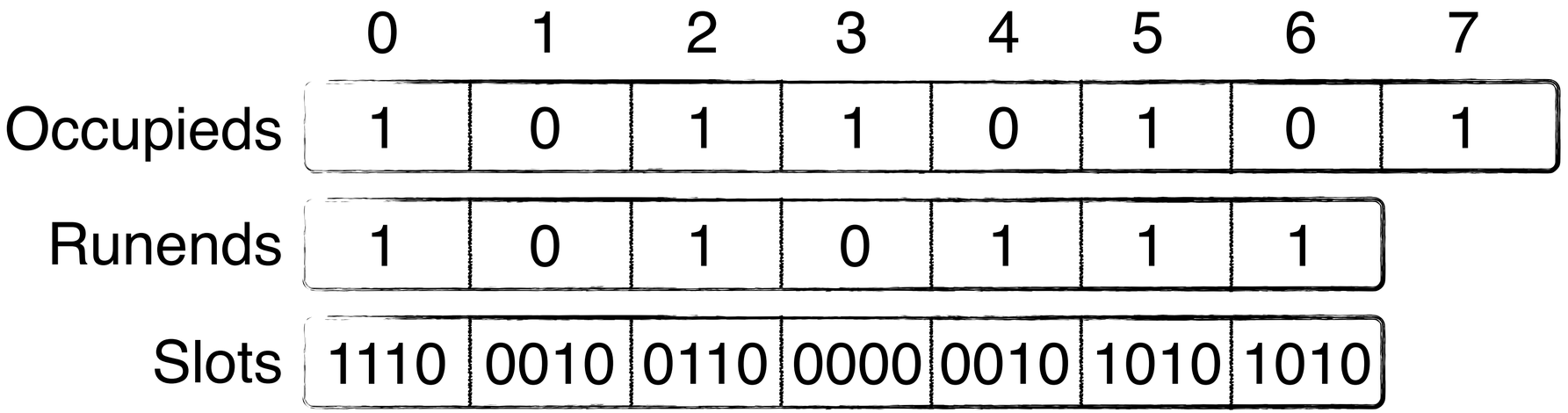

0
0
0
1
1
1
1
1
≠
0
1
0
0
0
0
0
1
0 1 2 3 4 5 6 7
Occupieds
1 0 1 1 0 1 0 1
Runends
1 0 1 0 1 1 1
Slots
1110 0010 0110 0000 0010 1010 1010
Negative

True positive if infix strictly between endpoints



True positive if infix strictly between endpoints

False positive only for collisions with endpoints



True positive if infix strictly between endpoints

False positive only for collisions with endpoints

Uniformity: FPR ≤ $O(2^{-F})$

	0	1	2	3	4	5	6	7
Occupieds	1	0	1	1	0	1	0	1
Runends	1	0	1	0	1	1	1	
Slots	1110	0010	0110	0000	0010	1010	1010	

False positive only for collisions with endpoints

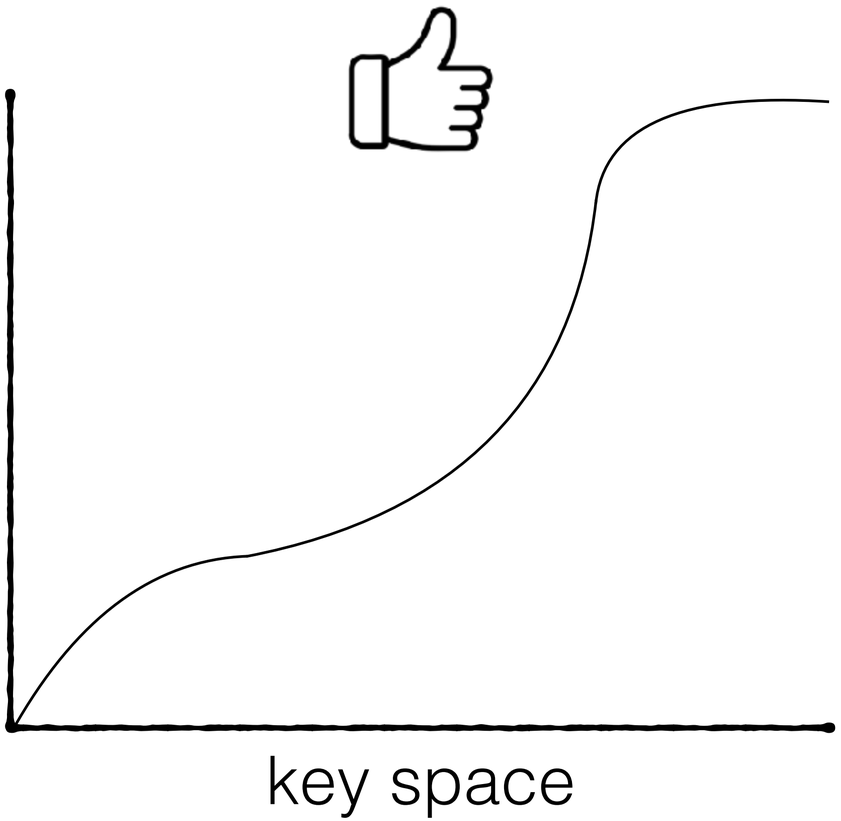
Uniformity: FPR ≤ $O(2^{-F})$

Range queries of any length

	0	1	2	3	4	5	6	7
Occupieds	1	0	1	1	0	1	0	1
Runends	1	0	1	0	1	1	1	
Slots	1110	0010	0110	0000	0010	1010	1010	

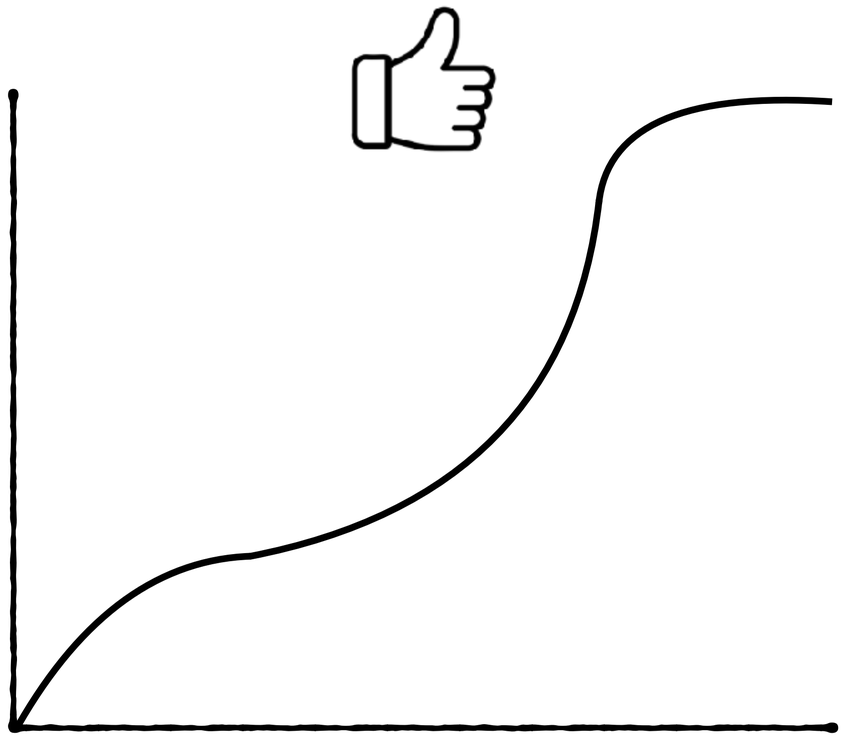
FPR ≤ $O(2^{-F})$ - a semi-robust guarantee

Smooth distributions
(Numeric data)

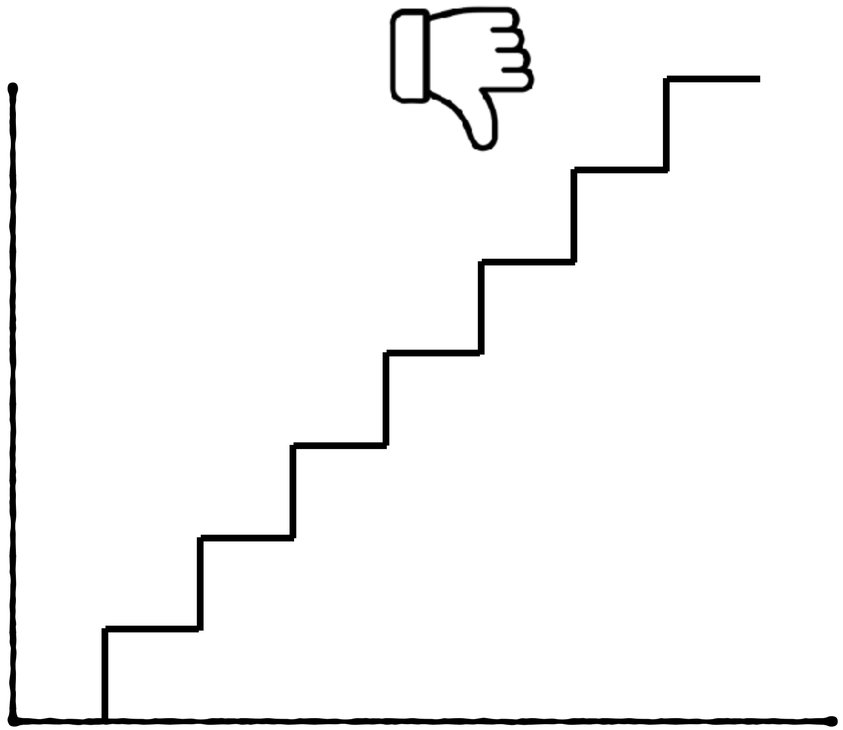


FPR $\leq O(2^{-F})$ - a semi-robust guarantee

Smooth distributions

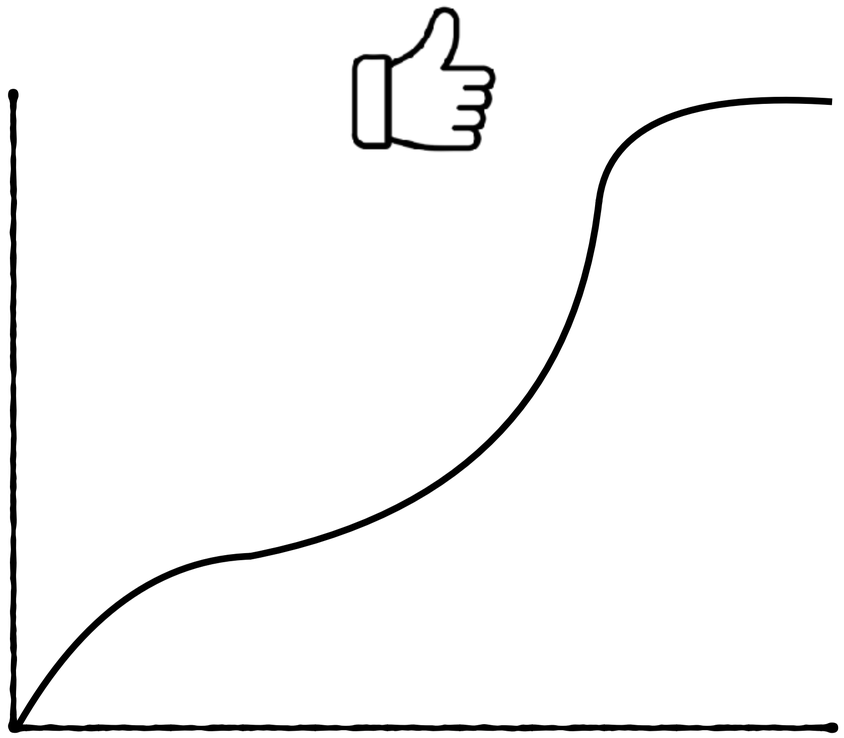


Jagged distributions

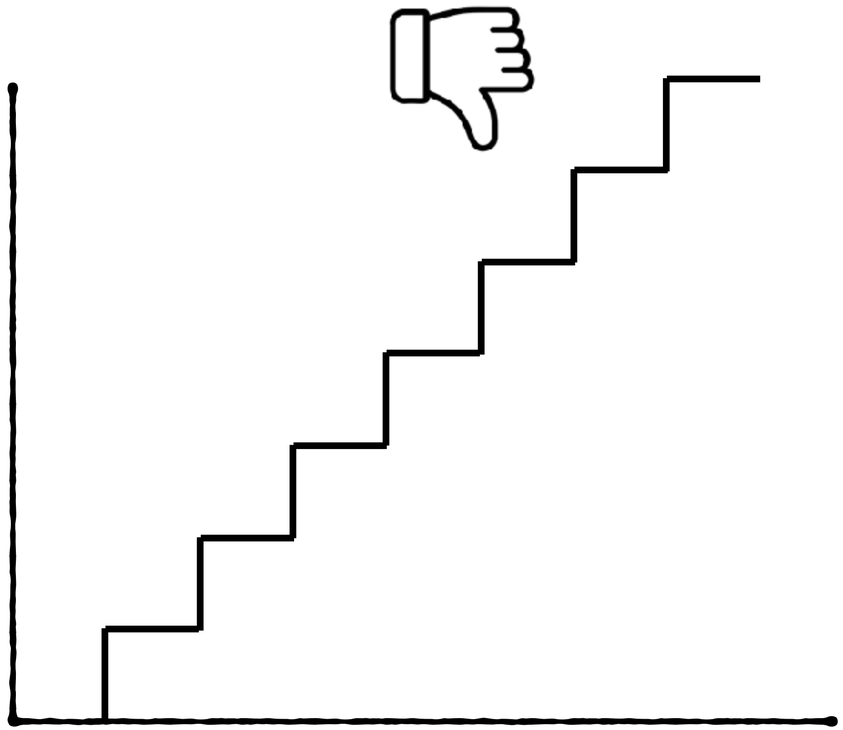


FPR $\leq O(2^{-F})$ - a semi-robust guarantee

Smooth distributions

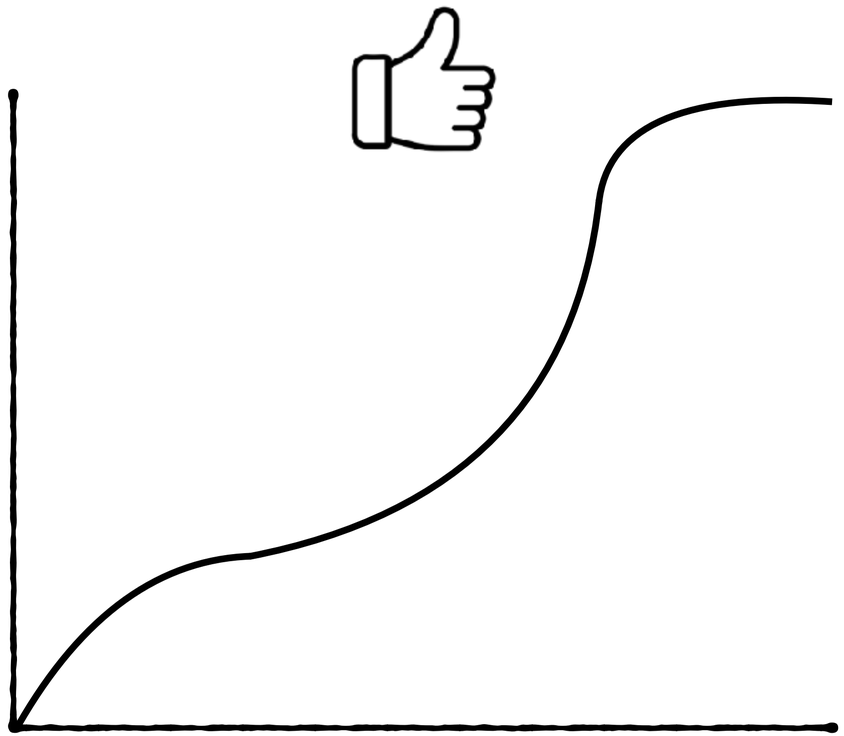


Jagged distributions
(Some natural language)

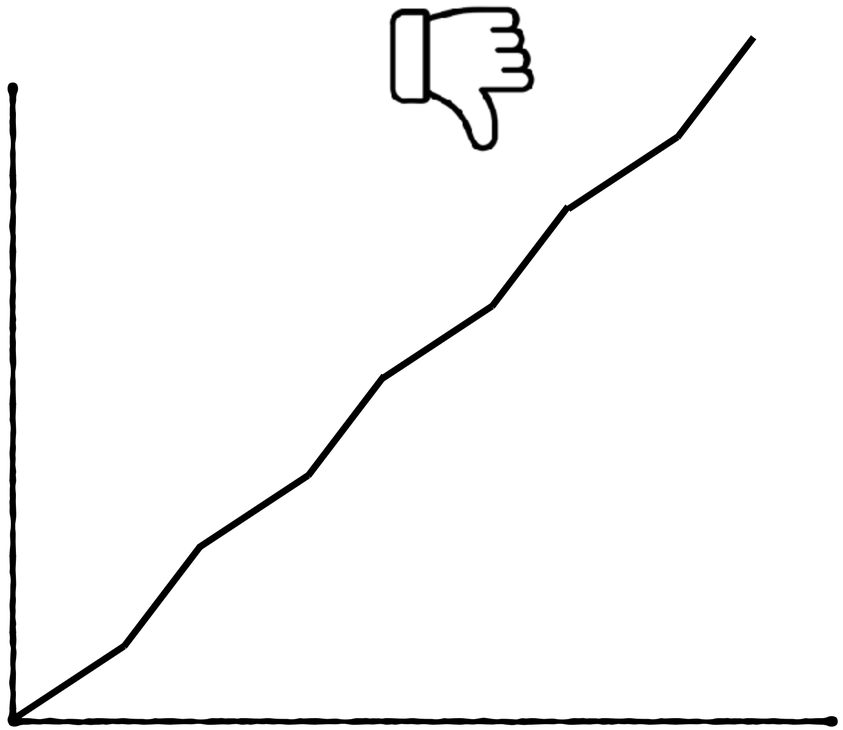


FPR $\leq O(2^{-F})$ - a semi-robust guarantee

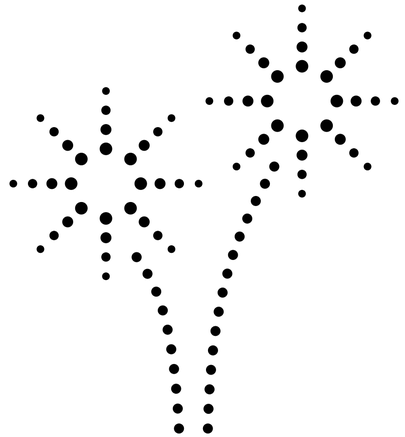
Smooth distributions



Jagged distributions



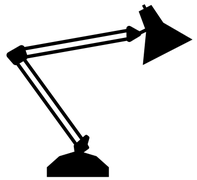
Use entropy encoding to smooth distribution



Congrats on finishing the course!

If you like:
Theory + Practical Impact

If you like:
Theory + Practical Impact



Tons of space to do research

If you like:
Theory + Practical Impact