

Data Structures and Algorithms

Bloom Filters

CS 225
Brad Solomon

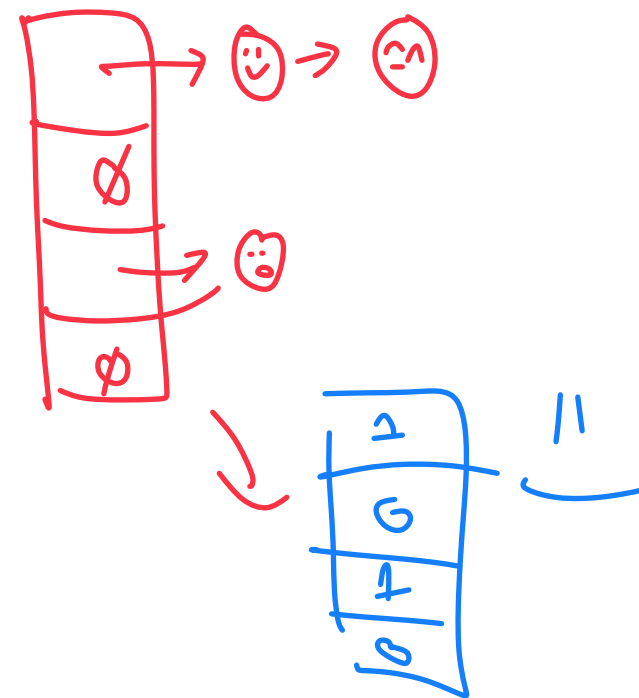
November 19, 2025

Not on
exam 3
(but on Final)



UNIVERSITY OF
ILLINOIS
URBANA - CHAMPAIGN

Department of Computer Science



Learning Objectives

Review when you would prefer different data structures

Build a conceptual understanding of a bloom filter

Review probabilistic data structures and one-sided error

Formalize the math behind the bloom filter

Running Times (Tradeoff Highlights)



	Hash Table	AVL	Linked List
Find	Expectation*: $O(1)^{***}$ Worst Case: $O(n)$	$O(\log n)$	$O(n)$
Insert	Expectation*: $O(1)^{***}$ Worst Case: $O(n)$	$O(\log n)$	$O(1)$
Storage Space	$O(n)$	$O(n)$	$O(n)$

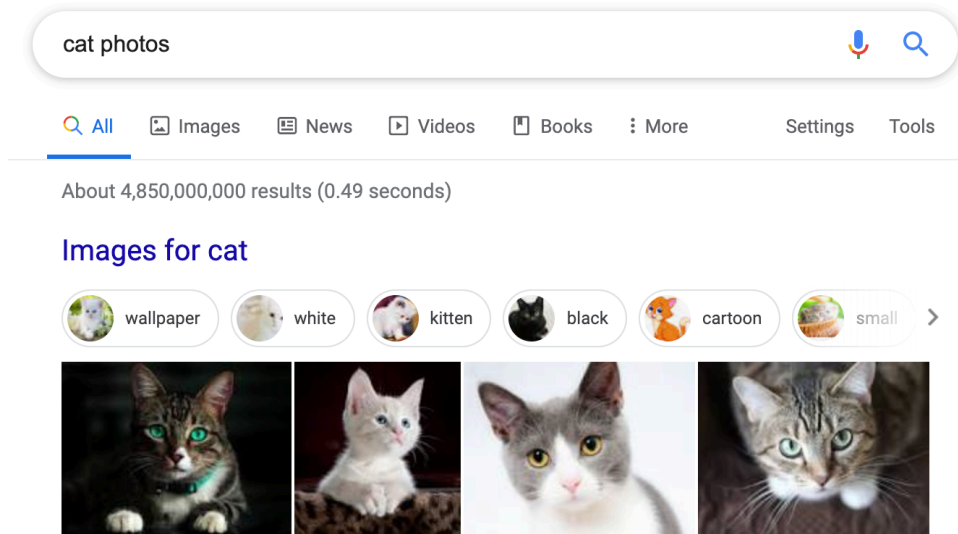
Handwritten notes:

- Red circles around the $O(1)^{***}$ values in the Find and Insert rows for Hash Table.
- Red arrows pointing from the $O(1)$ in the Insert row for Linked List to the $O(1)^{***}$ in the Insert row for Hash Table.
- Red arrows pointing from the $O(\log n)$ in the Find row for AVL to the $O(1)$ in the Insert row for Linked List.
- Red arrow pointing from the $O(n)$ in the Find row for Linked List to the $O(1)$ in the Insert row for Linked List.
- Red arrow pointing from the $O(n)$ in the Storage Space row for AVL to the $O(n)$ in the Storage Space row for Linked List.
- Red arrow pointing from the $O(n)$ in the Storage Space row for Hash Table to the $O(n)$ in the Storage Space row for Linked List.
- Red handwritten text: $O(1)$ with an arrow pointing to the $O(1)$ in the Insert row for Linked List.
- Red handwritten text: *sep chaining* with an arrow pointing to the $O(1)$ in the Insert row for Linked List.

Memory-Constrained Data Structures

What method would you use to build a search index on a collection of objects *in a memory-constrained environment*?

Constrained by Big Data (Large N)



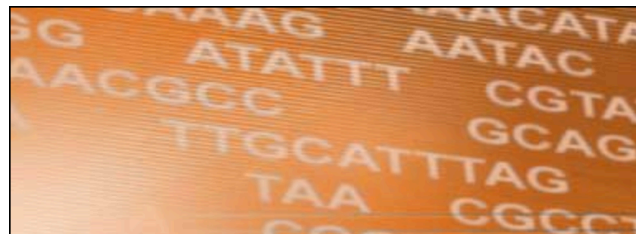
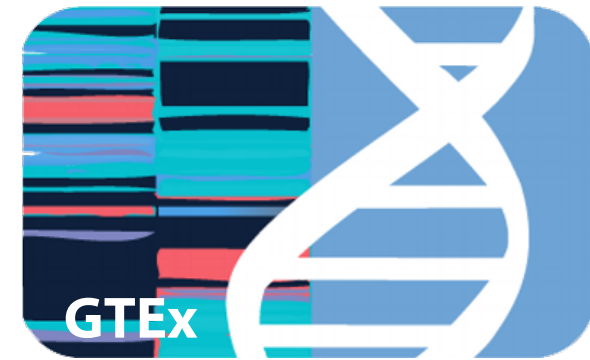
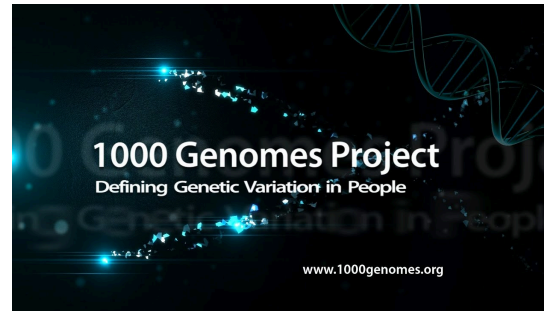
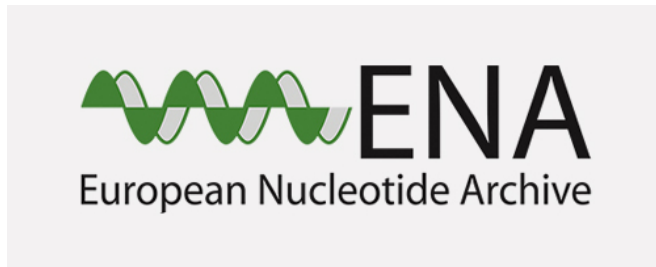
Google Index Estimate: >60 billion webpages

Google Universe Estimate (2013): >130 trillion webpages

Memory-Constrained Data Structures

What method would you use to build a search index on a collection of objects *in a memory-constrained environment*?

Constrained by Big Data (Large N)



SRA

Sequence Read Archive (SRA) makes biological sequence data available to the research community to enhance reproducibility and allow for new discoveries by comparing data sets. The SRA stores raw sequencing data and alignment information from high-throughput sequencing platforms, including Roche 454 GS System®, Illumina Genome Analyzer®, Applied Biosystems SOLiD System®, Helicos Heliscope®, Complete Genomics®, and Pacific Biosciences SMRT®.

Sequence Read Archive Size: >60 petabases (10^{15})

Memory-Constrained Data Structures

What method would you use to build a search index on a collection of objects *in a memory-constrained environment*?

Constrained by Big Data (Large N)

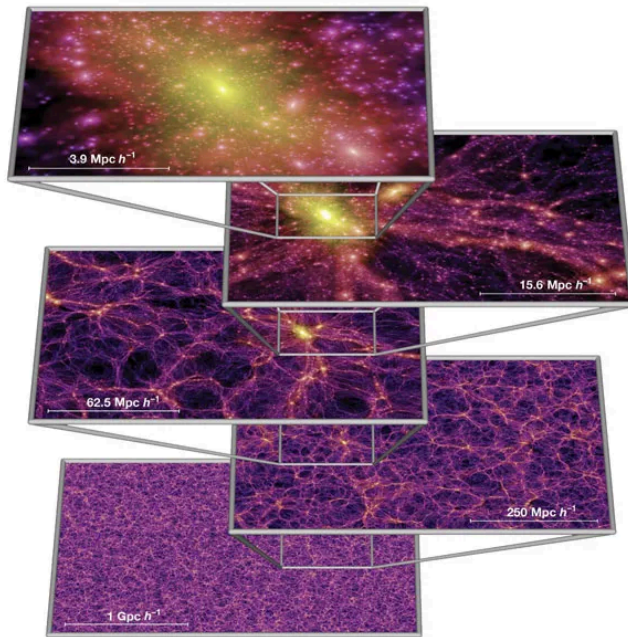


Image: <https://doi.org/10.1038/nature03597>

Sky Survey Projects	Data Volume
DPOSS (The Palomar Digital Sky Survey)	3 TB
2MASS (The Two Micron All-Sky Survey)	10 TB
GBT (Green Bank Telescope)	20 PB
GALEX (The Galaxy Evolution Explorer)	30 TB
SDSS (The Sloan Digital Sky Survey)	40 TB
SkyMapper Southern Sky Survey	500 TB
PanSTARRS (The Panoramic Survey Telescope and Rapid Response System)	~ 40 PB expected
LSST (The Large Synoptic Survey Telescope)	~ 200 PB expected
SKA (The Square Kilometer Array)	~ 4.6 EB expected

Table: <http://doi.org/10.5334/dsj-2015-011>

Estimated total volume of one array: 4.6 EB

What method would you use to build a search index on a collection of objects *in a memory-constrained environment*?

reduce data size

cache < 1 second

RAM Hours - Days

disk Months

network Years

seek

(Estimates are Time x 1 billion courtesy of <https://gist.github.com/hellerbarde/2843375>)

Memory-Constrained Data Structures



What method would you use to build a search index on a collection of objects *in a memory-constrained environment*?

1) Break dataset into pieces, process linearly / in parallel

2) Store somewhere / AWS



3) Pre process to a smaller search space

4) Complex data

Reducing storage costs

1) Throw out information that isn't needed

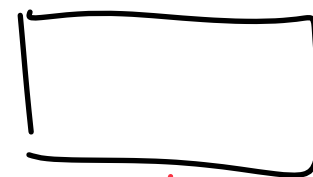
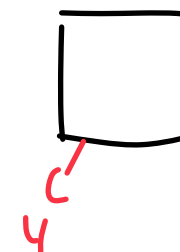
↳ partition & preprocess

Lossy data compression

2) Compress the dataset

↳ Reversible / exact / Lossless

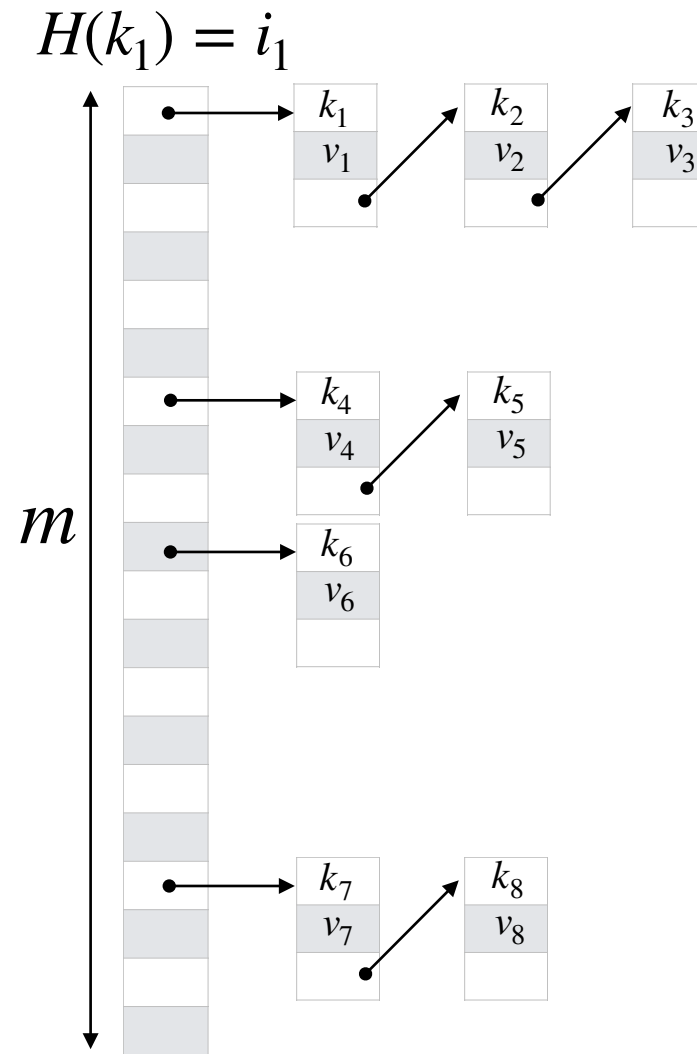
"AAAAABBB" = "A5B3"



Run length encoding

Reducing a hash table

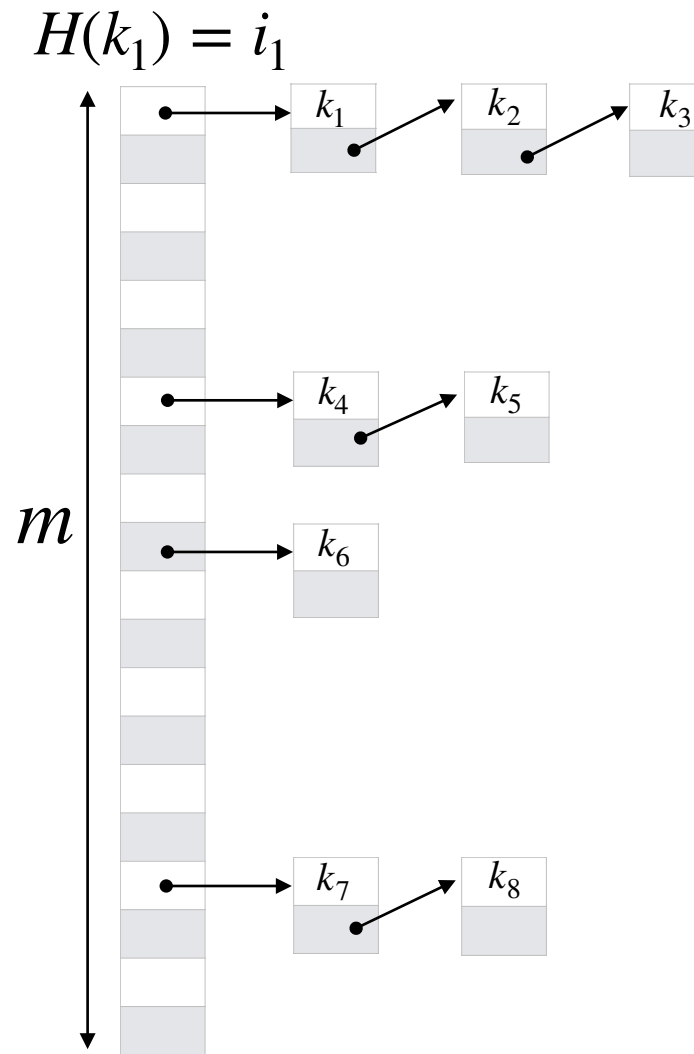
What can we remove from a hash table?



Reducing a hash table

What can we remove from a hash table?

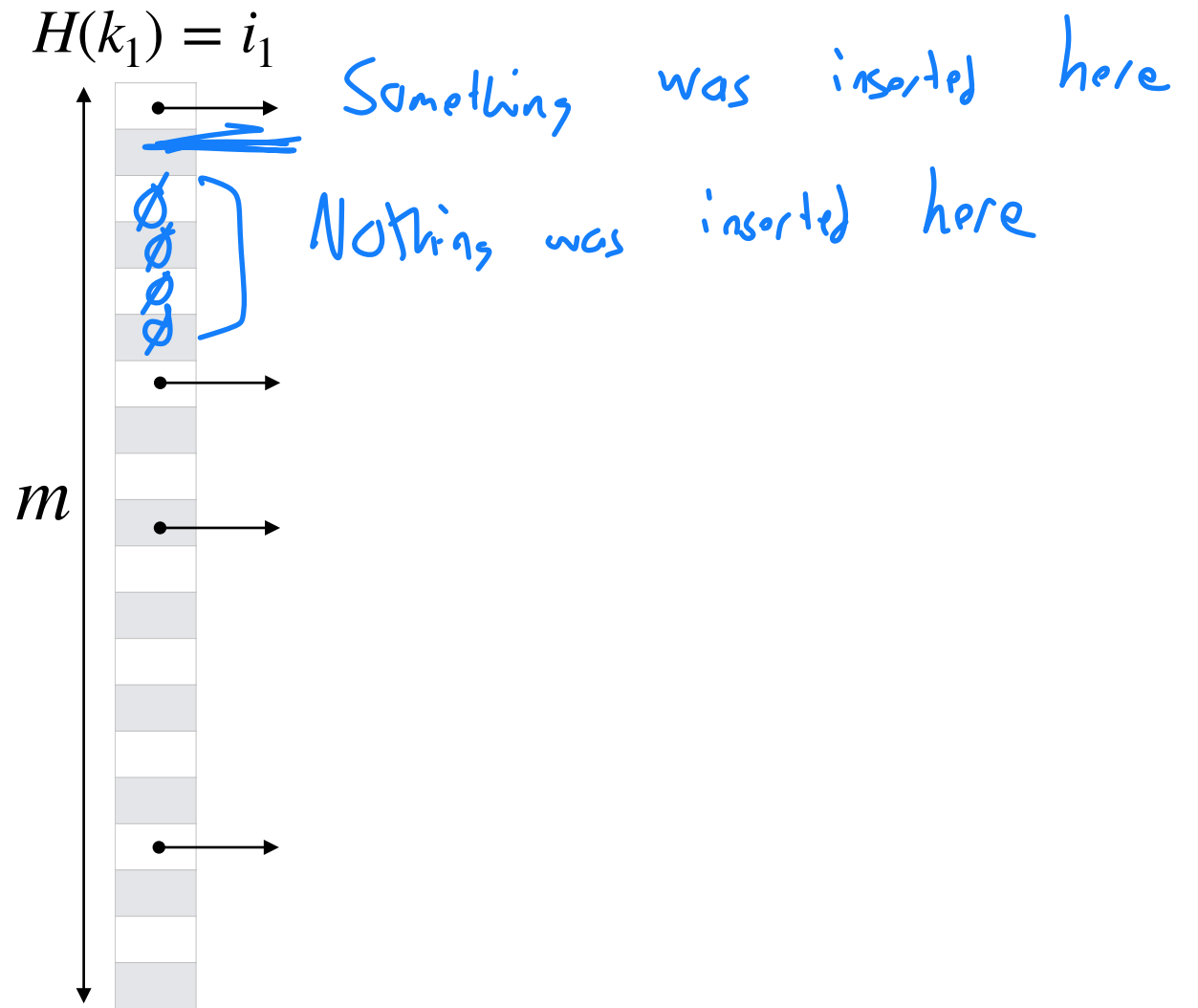
Take away values



Reducing a hash table

What can we remove from a hash table?

Take away values and keys



Reducing a hash table

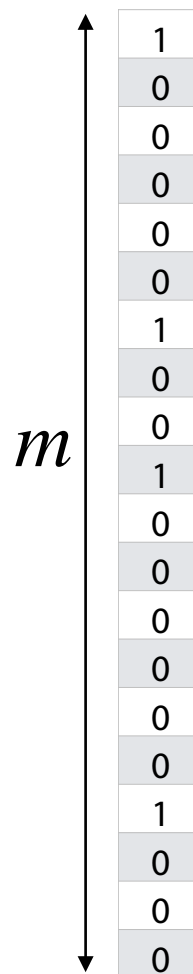


What can we remove from a hash table?

Take away values and keys

This is a ***bloom filter***

$$H(k_1) = i_1$$



1 if something hashed here
0 if nothing hashed here



Bloom Filter ADT

Constructor

Insert

Find

Bloom Filter: Insertion

S = { 16, 8, 4, 13, 29, 11, 22 } $16 \% 7 = 2$

$$h(k) = k \% 7$$

0	0
1	0 1
2	0 1
3	0
4	0 1
5	0
6	0 1

$$29 \% 7 = 1$$

Bloom filters ignore collisions !!

1) Hash item to address

2) Insert by setting value to 1

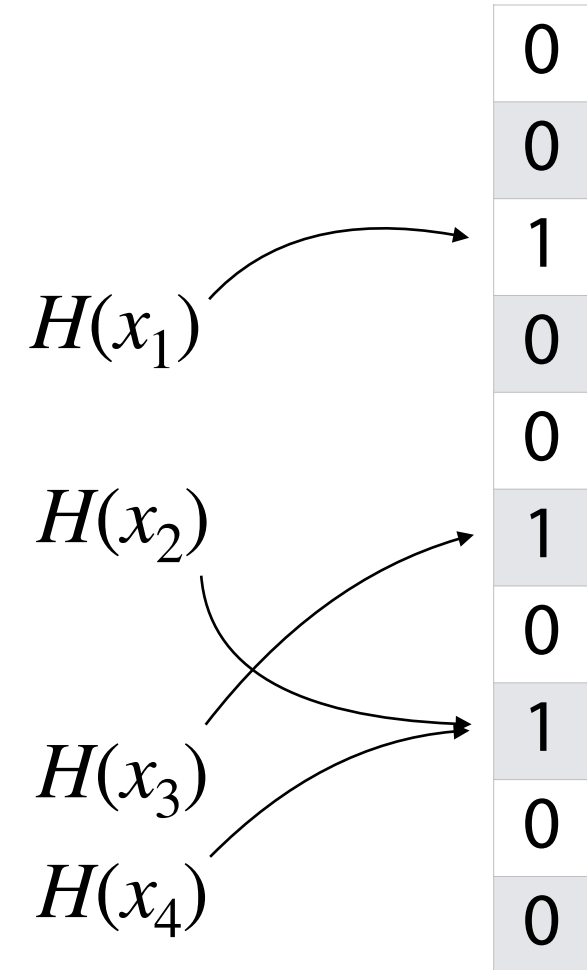
↳ If value already 1, stay 1

$O(1)$

Bloom Filter: Insertion

An item is inserted into a bloom filter by hashing and then setting the hash-valued bit to 1

If the bit was already one, it stays 1



Bloom Filter: Deletion

$S = \{ 16, 8, 4, 13, 29, 11, 22 \}$

$h(k) = k \% 7$

0	0
1	1 0
2	1
3	0
4	1
5	0
6	1 0

$_delete(13)$

1) Hash

2) Set 1 to 0

$_delete(29)$

$_find(8)$ Oh no!

Bloom Filter: Deletion

Tradeoff

Due to hash collisions and lack of information,
items cannot be deleted!

avoid collision detection

x_4 deletion
removes x_2
by mistake!

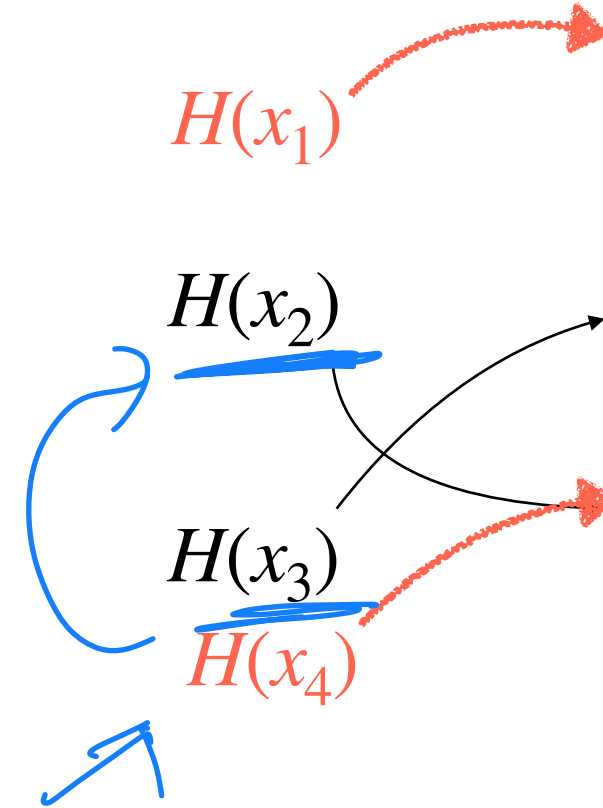
$H(x_1)$

$H(x_2)$

$H(x_3)$

$H(x_4)$

0
0
0
0
0
1
0
0
0
0



Bloom Filter: Search

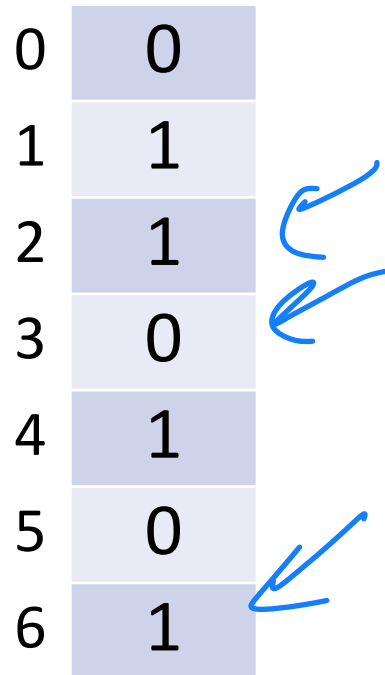
$O(1)$

Query if item exists in dataset

$S = \{16, 8, 4, 13, 29, 11, 22\}$

$h(k) = k \% 7$

0	0
1	1
2	1
3	0
4	1
5	0
6	1



\checkmark True
find(16)
↳ Hash item

↳ Return value at address

find(20) true

↳ this is actually False!

find(3) False

Has a chance of being wrong!

Bloom Filter: Search

The bloom filter is a *probabilistic* data structure!

If the value in the BF is 0:

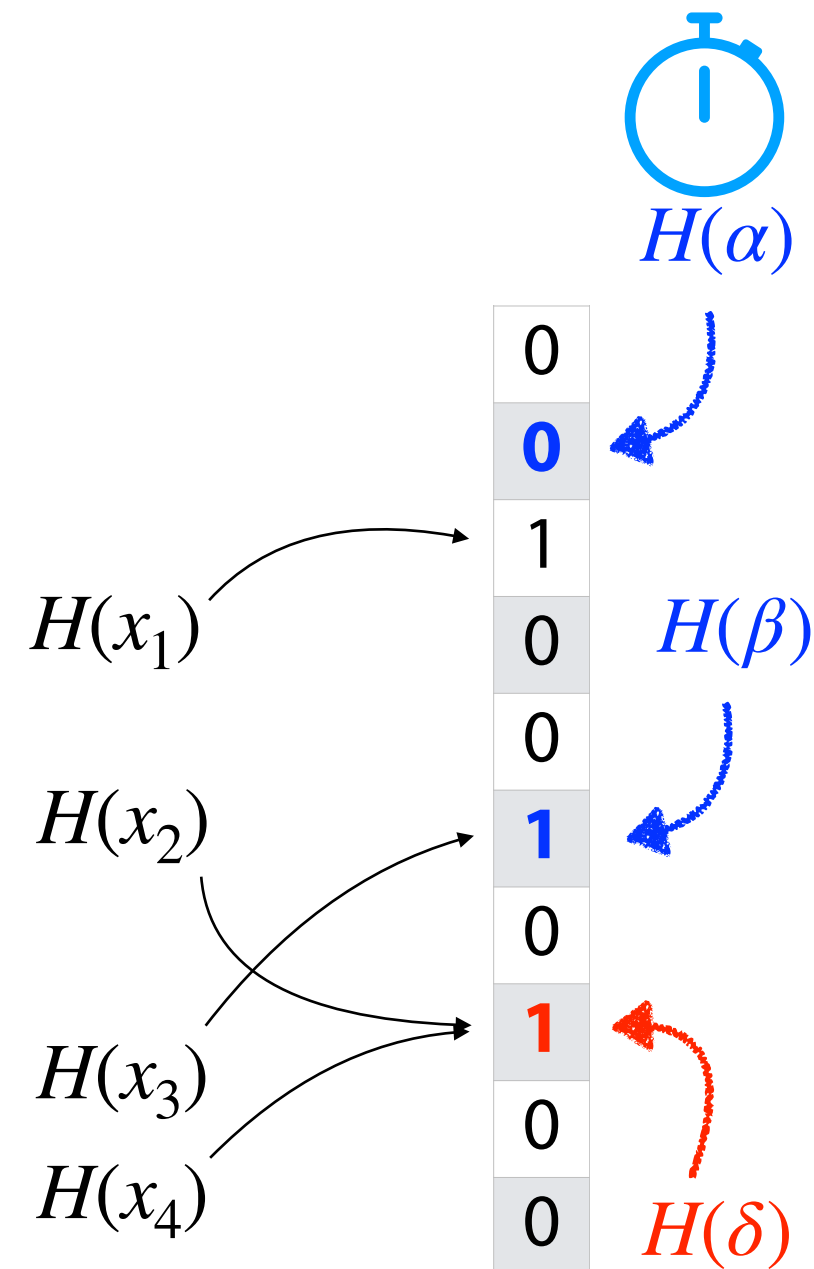
100% of the time, the item is not present

If the value in the BF is 1:

the item might be present

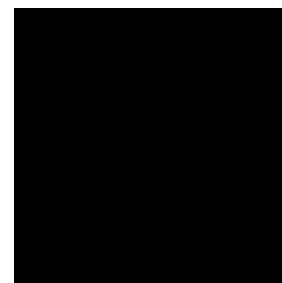
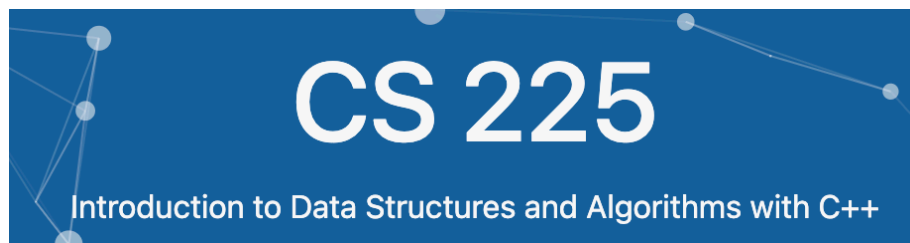
or

its a False Positive

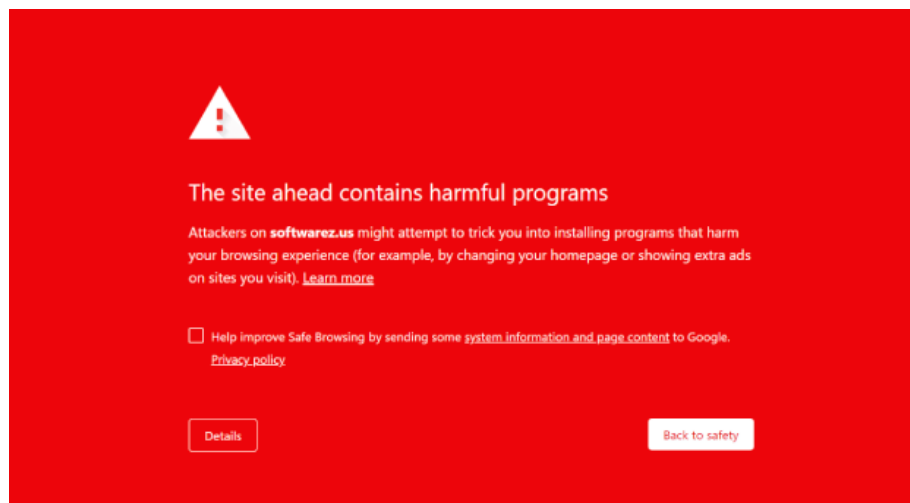


Probabilistic Accuracy: Malicious Websites

Imagine we have a detection oracle that identifies if a site is malicious



"Not malicious"



"Malicious"

It's better to be safe
↗

Probabilistic Accuracy: Malicious Websites

Imagine we have a detection oracle that identifies if a site is malicious

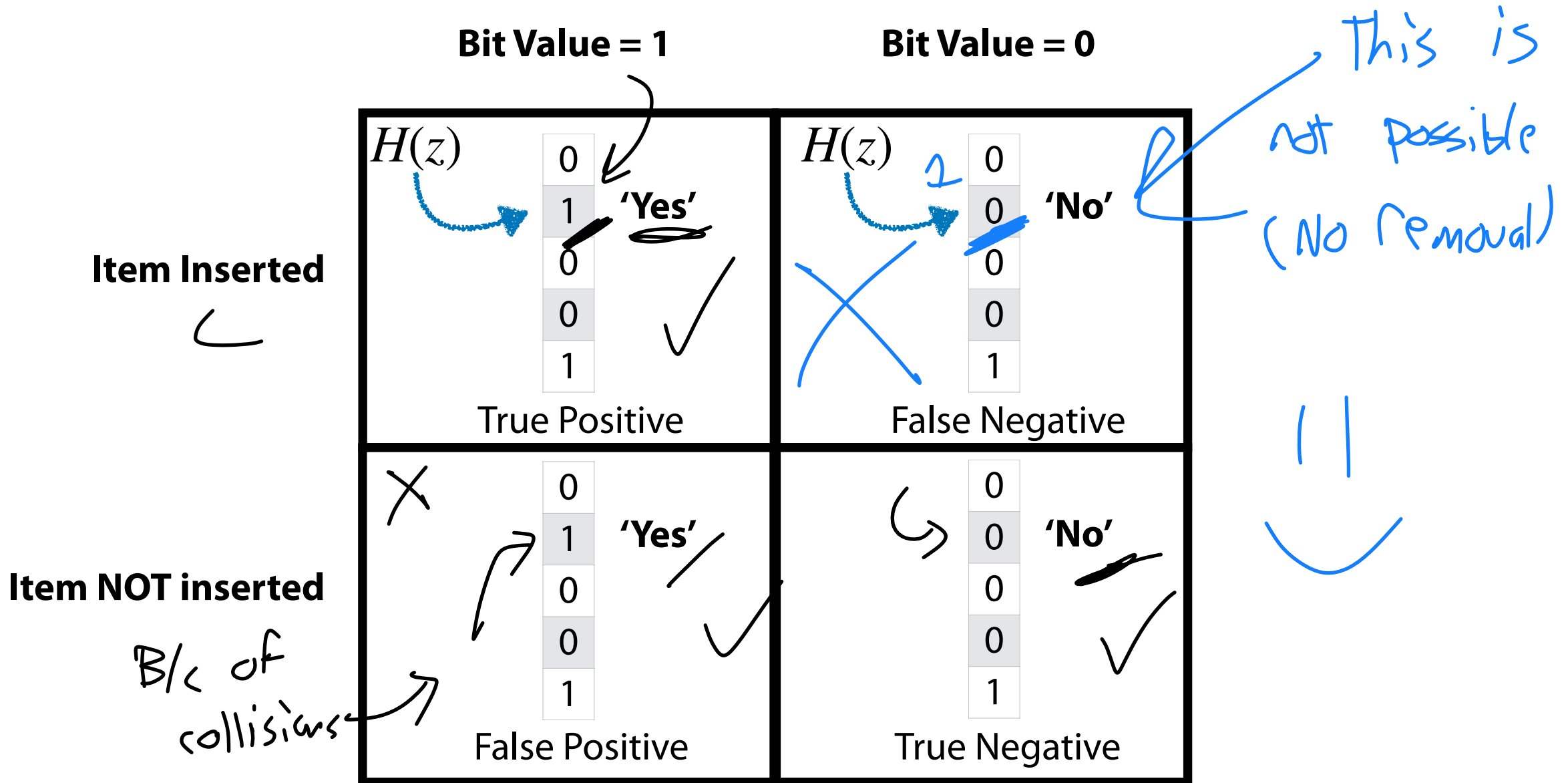
True Positive: Oracle says M , website is M

False Positive: Oracle says M , website is not M (safe)
↳ This is annoying

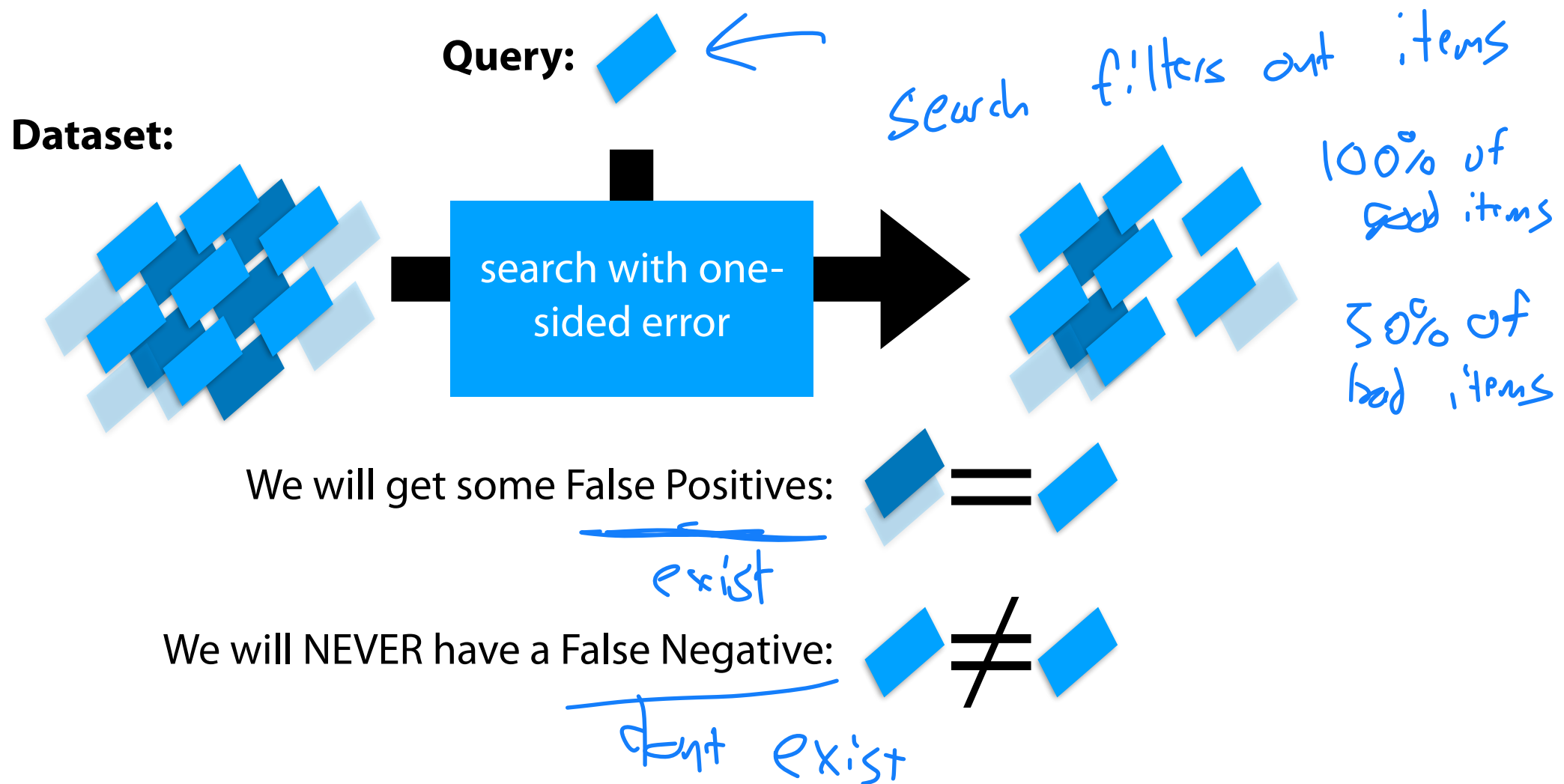
False Negative: Oracle says Safe (not M) , website is M
↳ This is BAD

True Negative: Safe, Safe

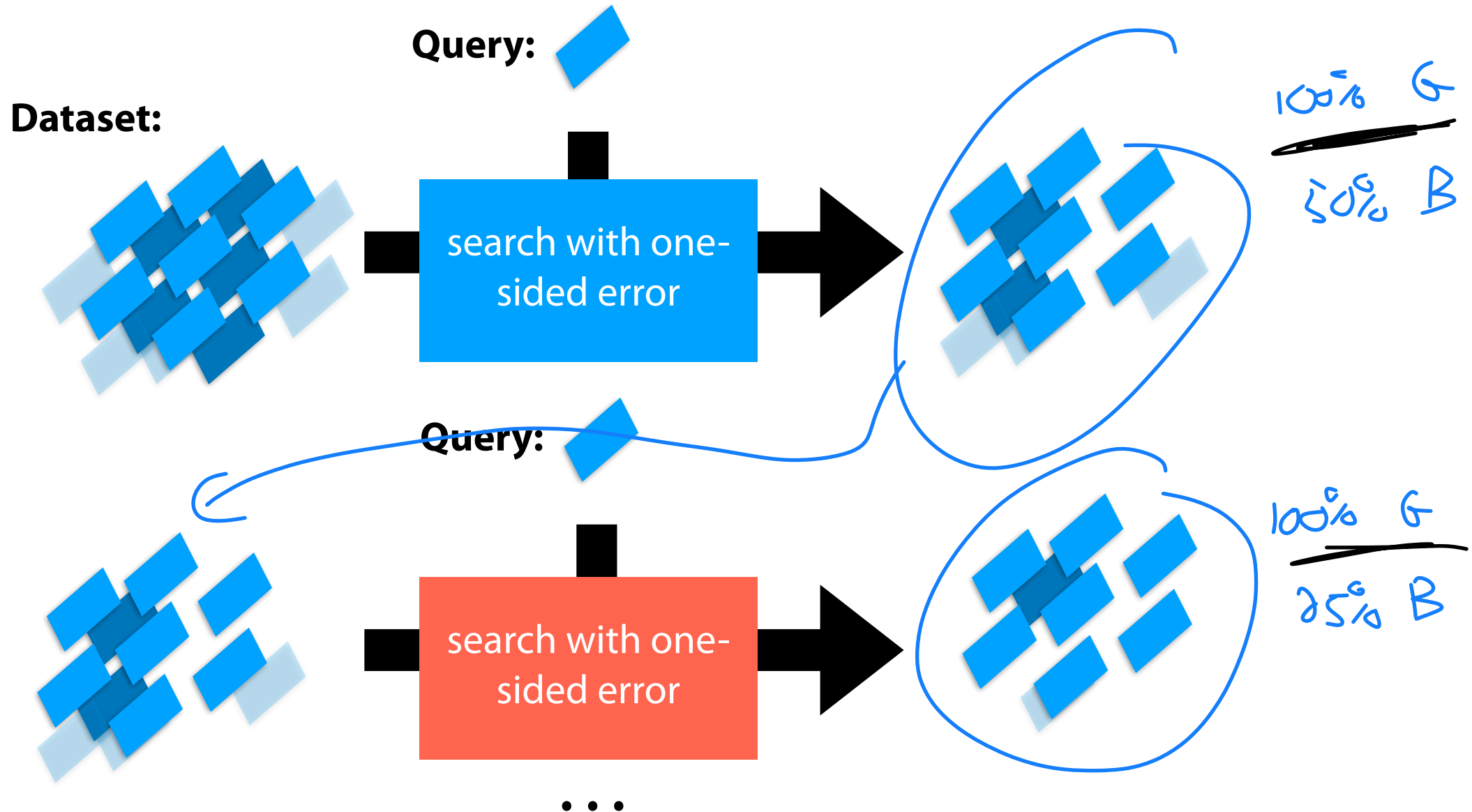
Imagine we have a **bloom filter** that **stores malicious sites...**



Probabilistic Accuracy: One-sided error

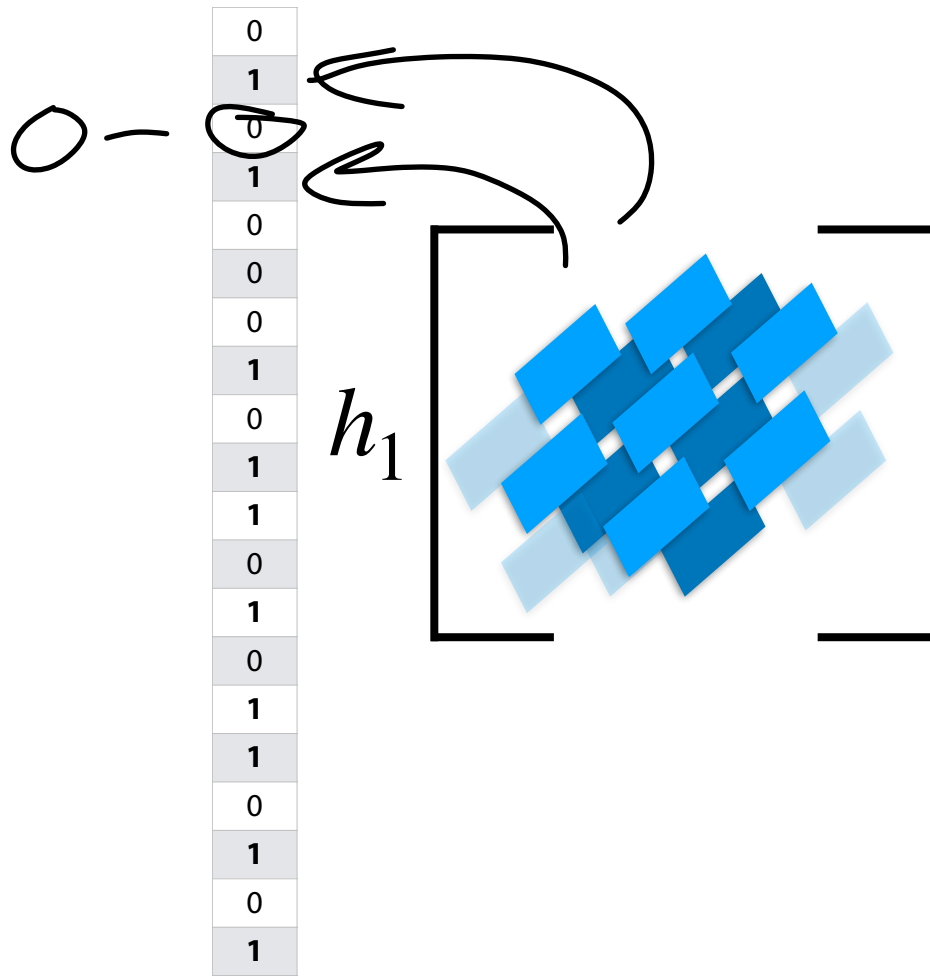


Probabilistic Accuracy: One-sided error



Bloom Filter: Repeated Trials

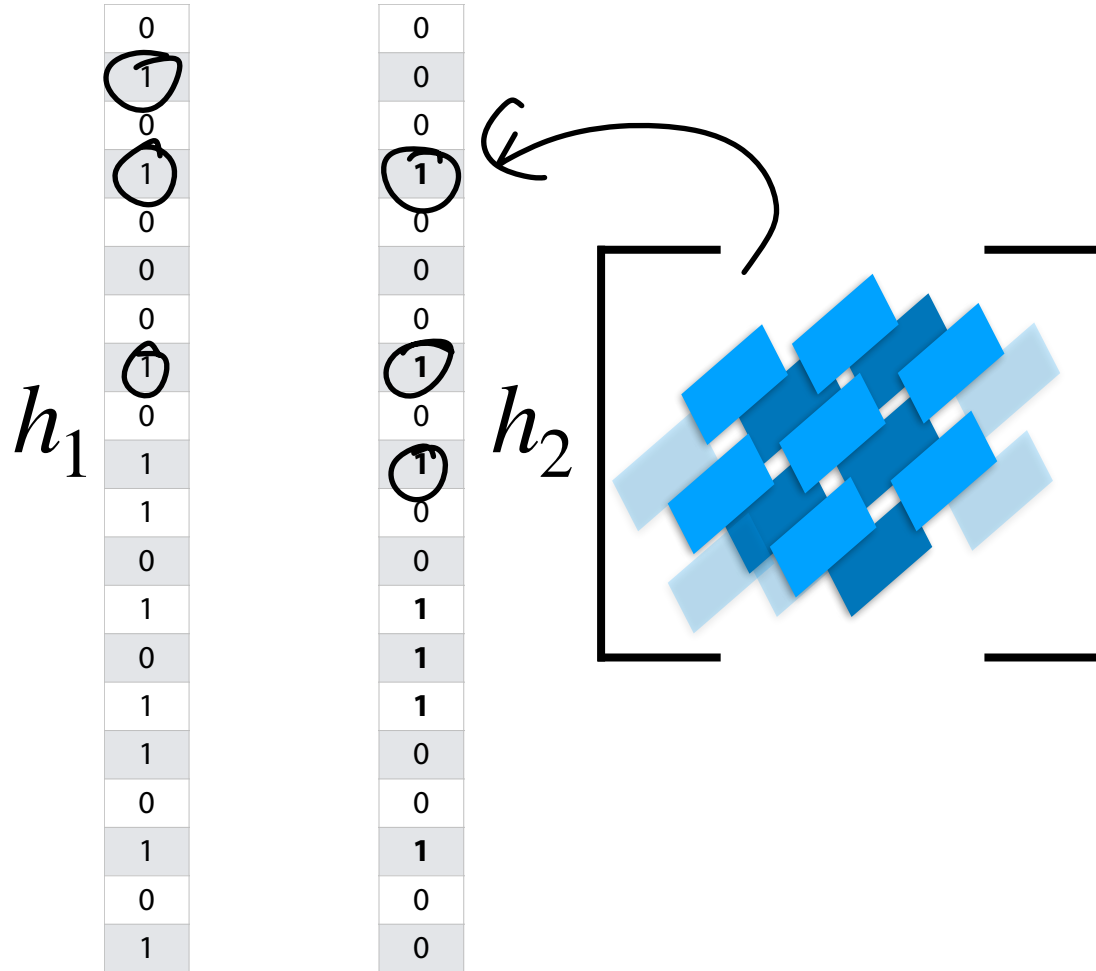
Use many hashes/filters; add each item to each filter



Bloom Filter: Repeated Trials

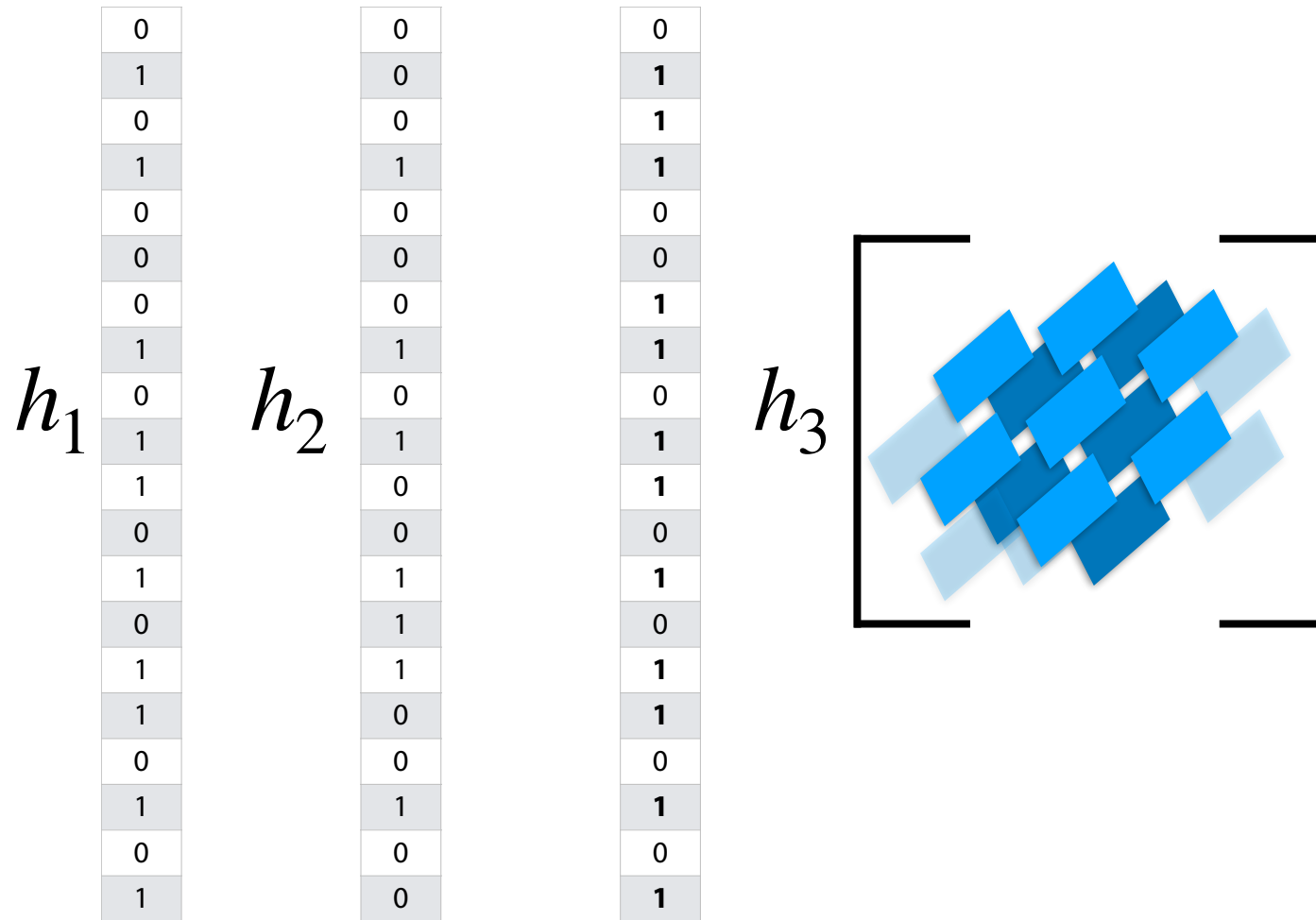
← Repeated random trials

Use many hashes/filters; add each item to each filter



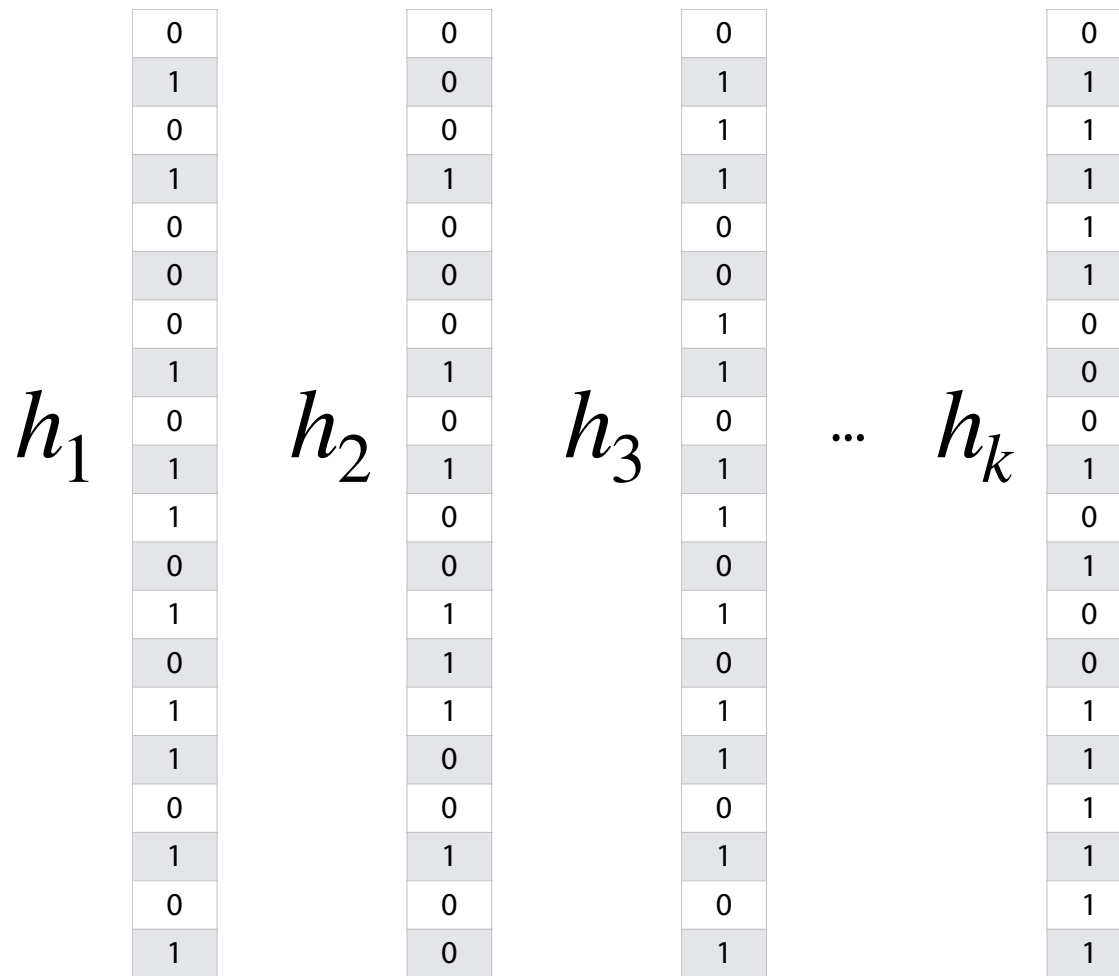
Bloom Filter: Repeated Trials

Use many hashes/filters; add each item to each filter



Bloom Filter: Repeated Trials

Use many hashes/filters; add each item to each filter



Bloom Filter: Repeated Trials

Query for y

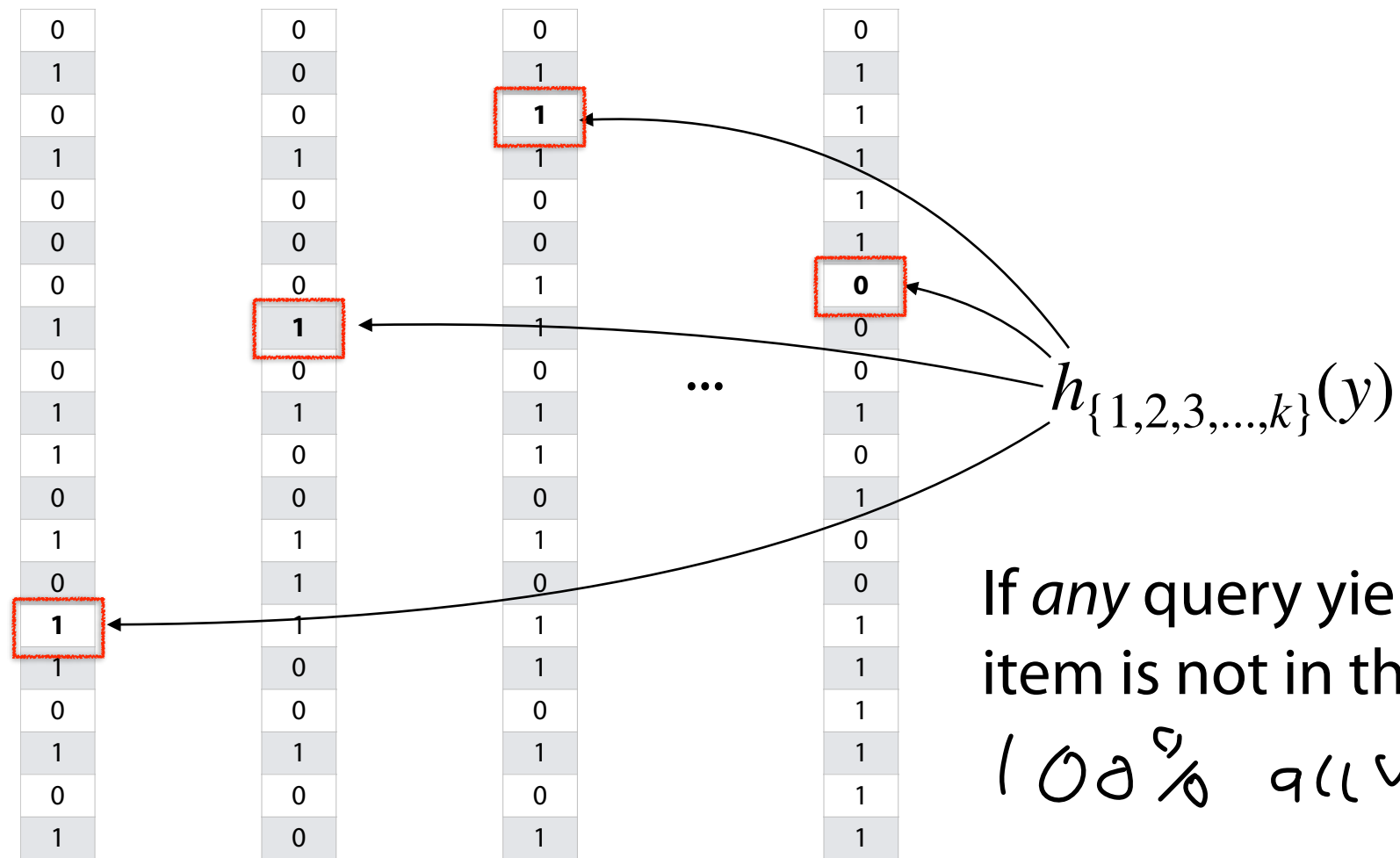
0	0	0	0
1	0	1	1
0	0	1	1
1	1	1	1
0	0	0	1
0	0	0	1
0	0	1	0
1	1	1	0
0	0	0	0
1	1	1	1
1	0	1	0
0	0	0	1
1	1	1	0
0	1	0	0
1	1	1	1
1	0	1	1
0	0	0	1
1	1	1	1
0	0	0	1
1	0	1	1

...

$$h_{\{1,2,3,\dots,k\}}(y)$$

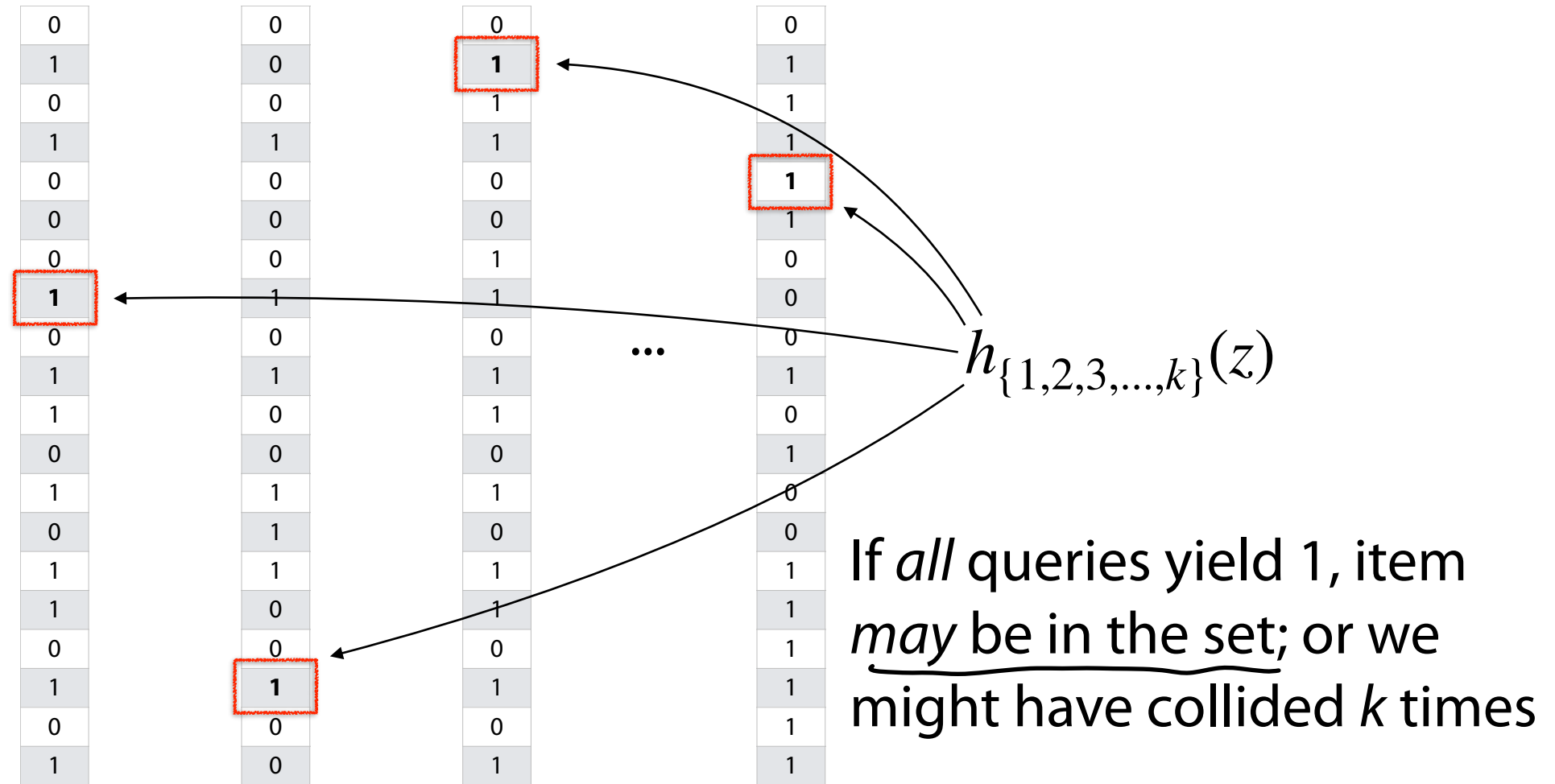
Bloom Filter: Repeated Trials

Is must be by collision



If *any* query yields 0,
item is not in the set
100% accuracy

Bloom Filter: Repeated Trials



Bloom Filter: Repeated Trials



Using repeated trials, even a very bad filter can still have a very low FPR!

If we have k bloom filter, each with a FPR p , what is the likelihood that **all** filters return the value '1' for an item we didn't insert?

$$\text{FPR} = 50\%$$

$$\left(\frac{1}{2}\right)^k$$

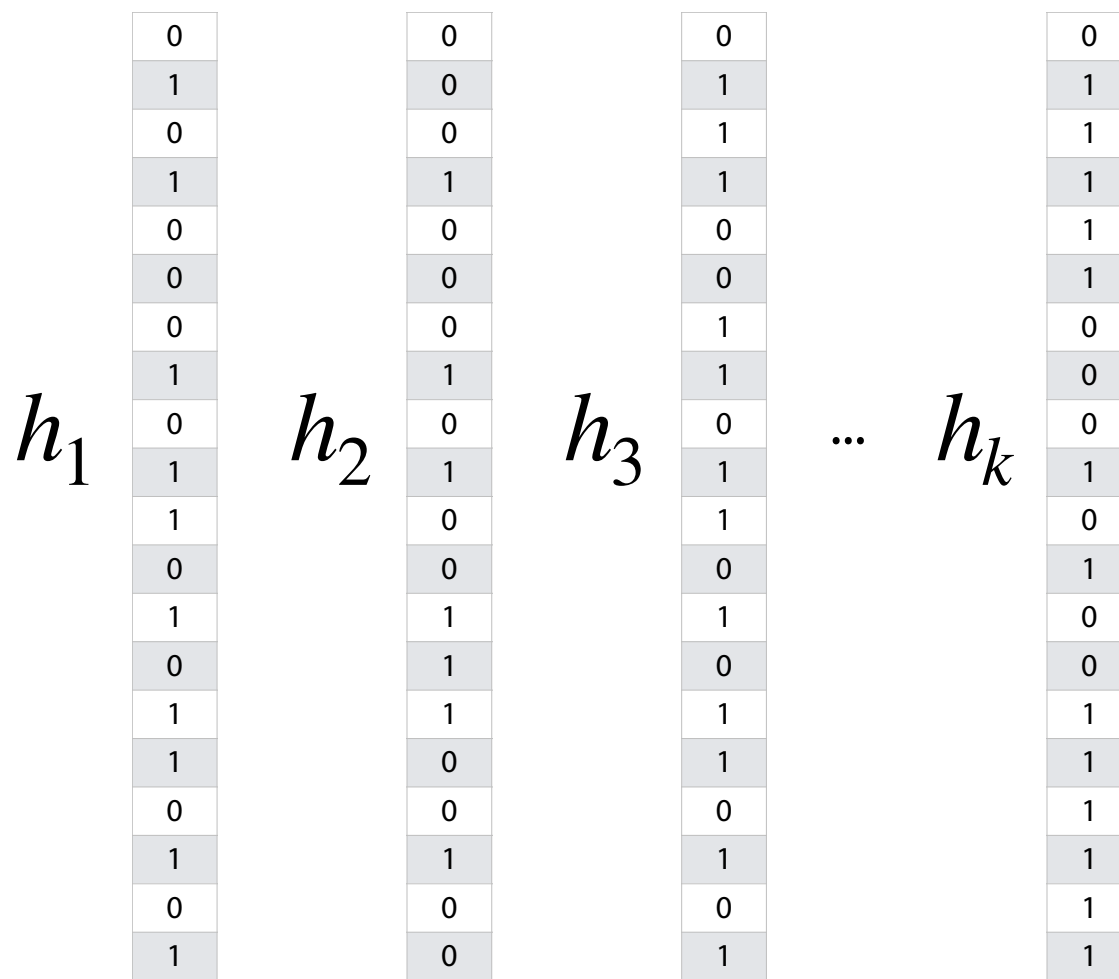
$$k = 10$$

$$\rightarrow 0.00097$$

of hash functions
is any runtime

Bloom Filter: Repeated Trials

But doesn't this hurt our storage costs by storing k separate filters?

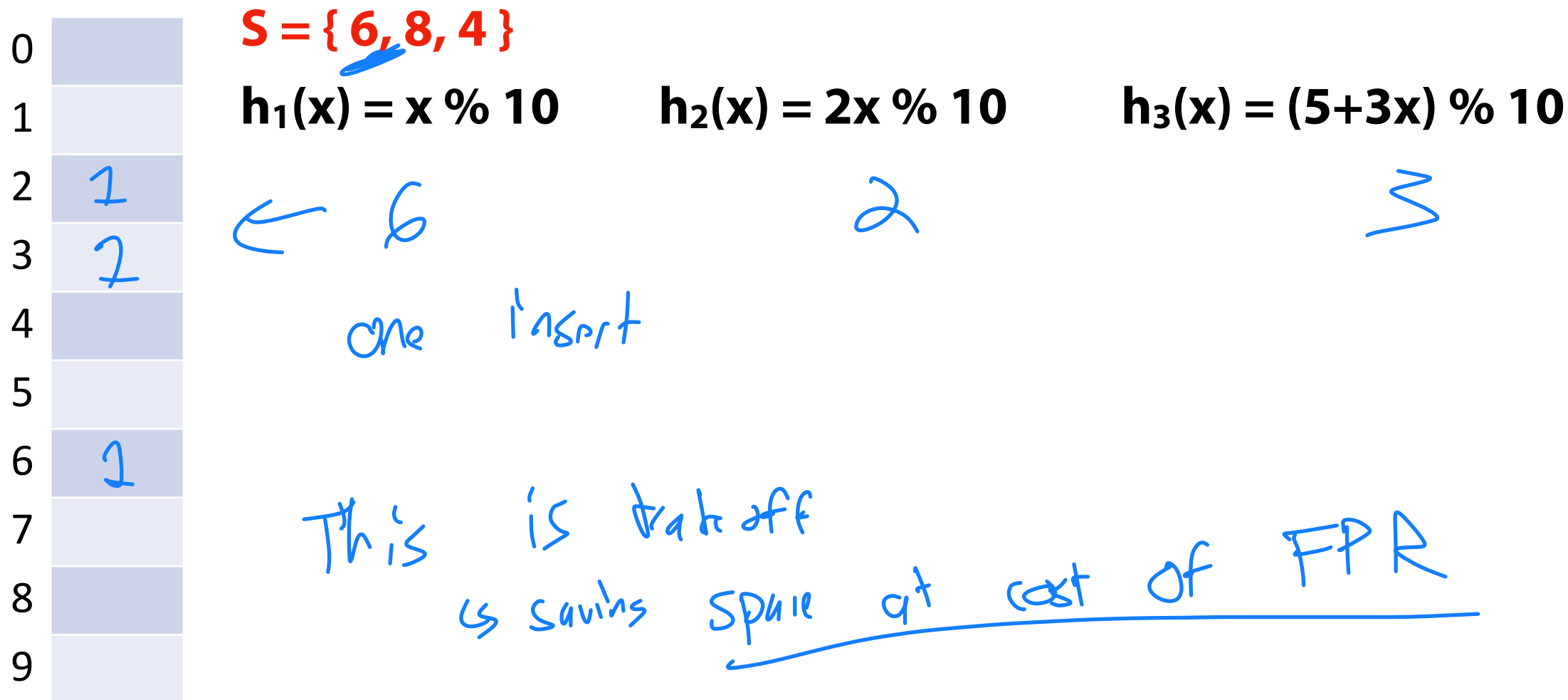


Is $n \cdot k$
truly better
than n ?

m

Bloom Filter: Repeated Trials

Rather than use a new filter for each hash, one filter can use k hashes



Bloom Filter: Repeated Trials

Rather than use a new filter for each hash, one filter can use k hashes

0	0	$h_1(x) = x \% 10$	$h_2(x) = 2x \% 10$	$h_3(x) = (5+3x) \% 10$
1	0			
2	1	<code>_find(1)</code>		
3	1			
4	1			
5	0			
6	1	<code>_find(16)</code>		
7	1			
8	1			
9	1			

Bloom Filter



A probabilistic data structure storing a set of values

$$H = \{h_1, h_2, \dots, h_k\}$$

Built from a bit vector of length m and k hash functions

Insert / Find runs in: _____

Delete is not possible (yet)!

0
0
1
0
0
1
0
1
0
0