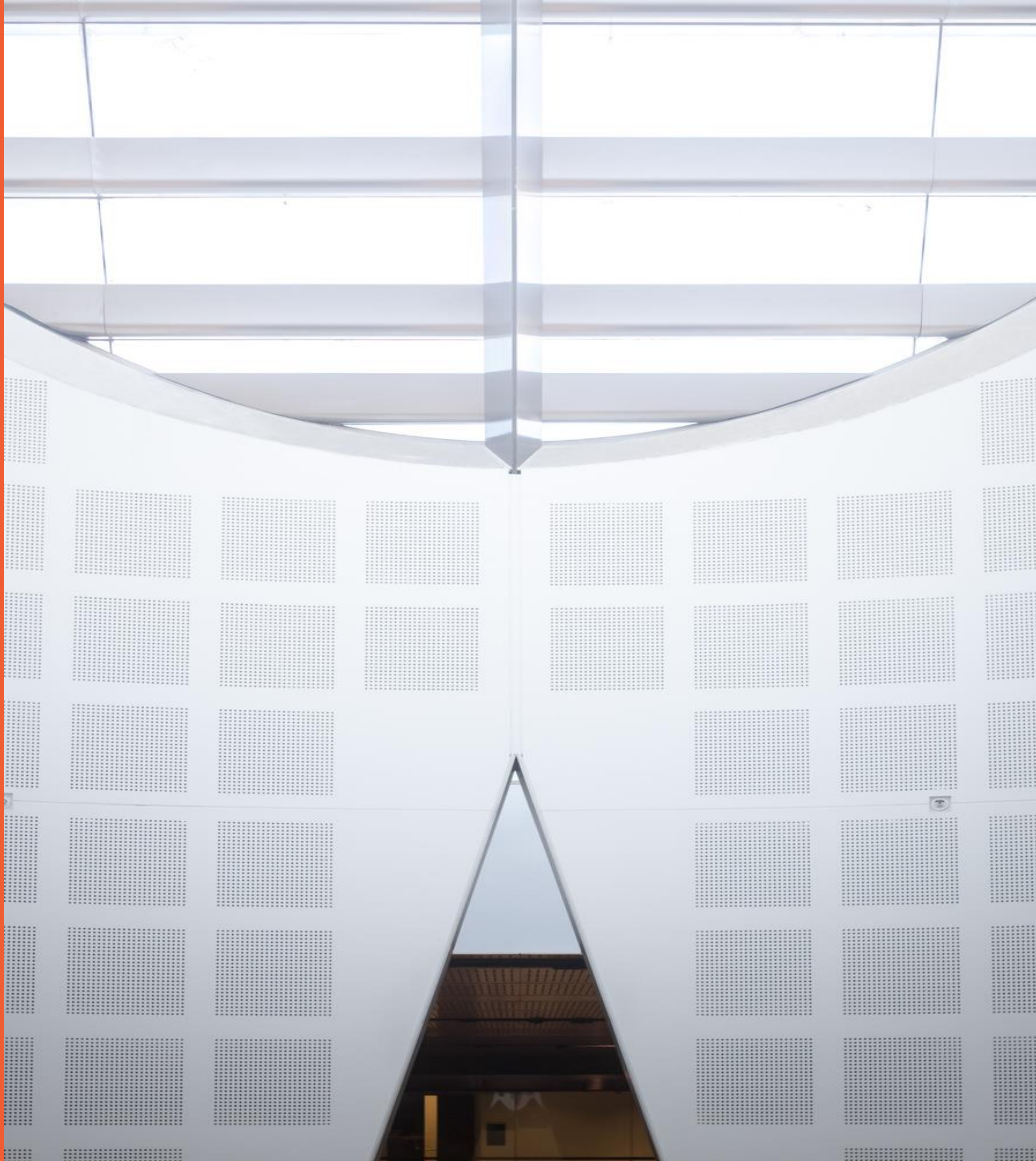
COMPx270: Randomised and Advanced Algorithms
Lecture 6: Hashing and Friends

Clément Canonne
School of Computer Science

THE UNIVERSITY OF SYDNEY



A question ❓

A SID is of the form 450687816 (9 digits, each between 0 and 9). How many distinct SIDs are there?

10^9

How many distinct students have there even been at U Syd?

much less than that

$\approx 10^6$-ish?

Dictionaries (Maps, Associative arrays...)

size 10^9

"universe"

size m

n points! $\approx 10^6$

$S \subseteq \mathcal{X}$

points

$|S| \ll |\mathcal{X}|$

$n \ll m$

	LIST	ARRAY	BST
INSERT	$\cancel{O(1)}$ $O(n)$	$O(1)$	$O(\log n)$
LOOKUP	$O(n)$	$O(1)$	$O(\log n)$
REMOVE	$O(n)$	$O(1)$	$O(\log n)$
SPACE	$O(n)$	$O(m)$	$O(n)$

Important detour: data representation

- Indexing something from $\mathcal{X}$ takes $\log_2 |\mathcal{X}| = \log_2 m$ bits.

We hide these logs, often.

- $\mathcal{F} = \{h : \mathcal{X} \to \mathcal{Y}\}$ $\rightarrow$ $|\mathcal{F}| = m'^{m}$

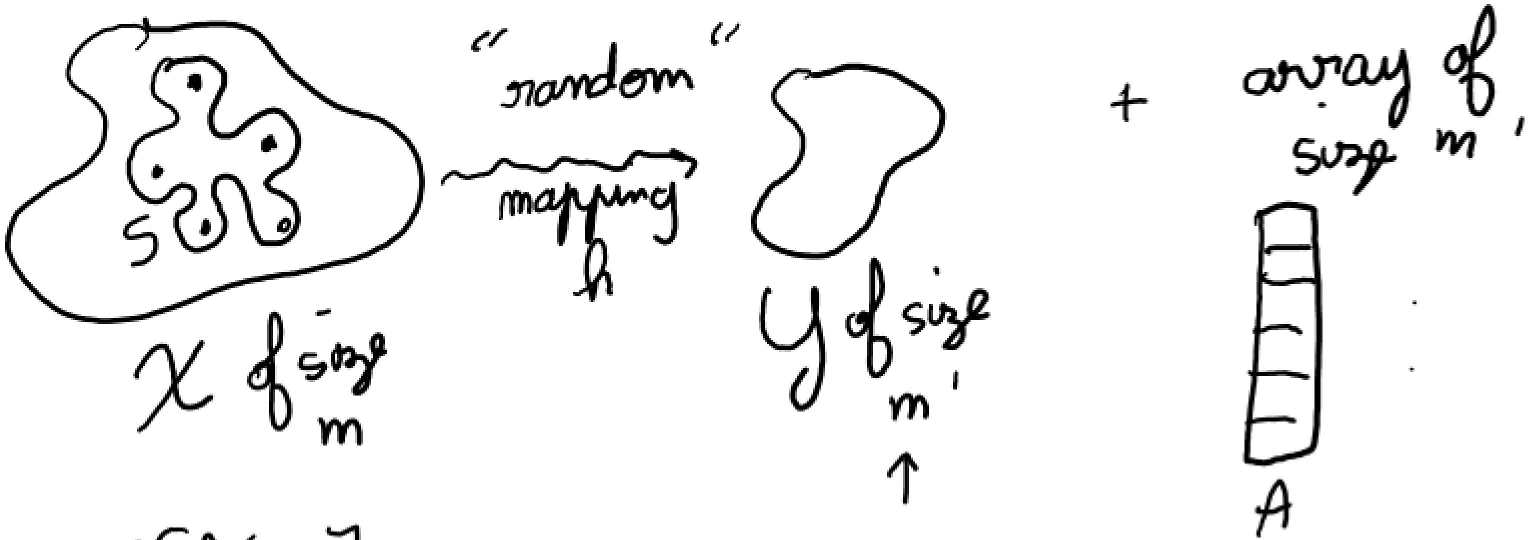
size m size m'

Need $m \log m'$ bits to represent an arbitrary function from $\mathcal{X}$ to $\mathcal{Y}$

Hash tables

A randomised solution to the Dictionary problem (ADT)

$|S| = n$



$n \le m' \ll m$

Store info about $x \in \mathcal{X}$ in $A[\underbrace{h(x)}_{\in \mathcal{Y}}]$

Ideally

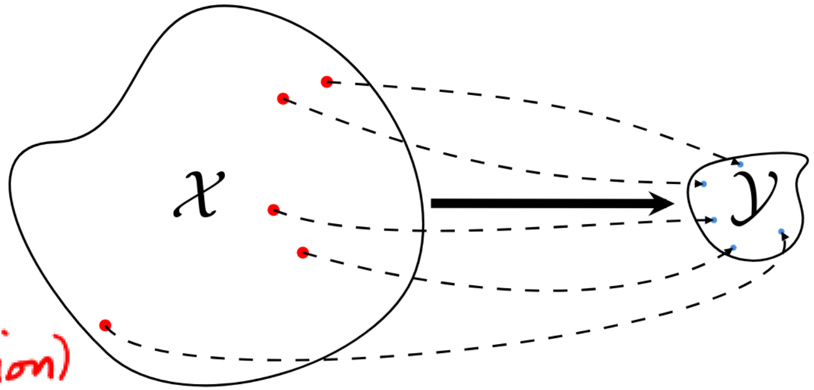
time $O(1)$ INSERT
$O(1)$ LOOKUP
$O(1)$ REMOVE

space $O(n)$

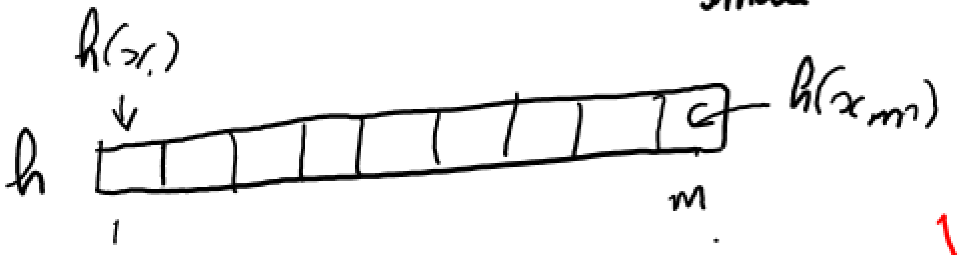
Total storage

$$\Omega\left(\text{"size of } h\text{"} + \underbrace{\text{size of } A}_{m'}\right)$$

Hash tables: what is random?



- The subset S (n points) is "typical" ? ✗ (big assumption)
  $\approx$ random
- The ~~function~~ hash function $h : \mathcal{X} \to \mathcal{Y}$ is random ✗ (too much memory)
- The hash function is "sort of random" ✓
  Pick h u.a.r. from $\mathcal{H} \subsetneq \{\mathcal{X} \to \mathcal{Y}\}$
  small



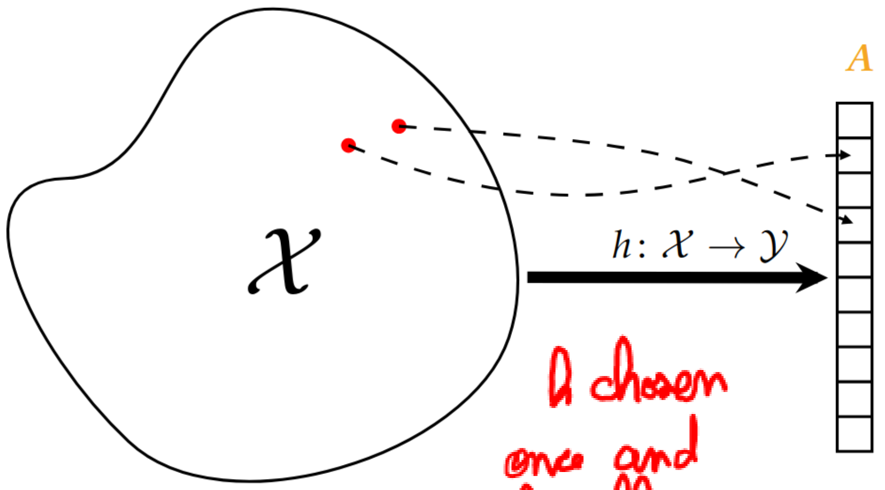
Want: $\forall x, x'$, $x \neq x'$: $\Pr_{h \sim \mathcal{H}}\left[h(x) = h(x')\right] \leq \frac{1}{|\mathcal{Y}|}$

(*) [universal hash family $\mathcal{H}$]

Hash tables: the data structure

$H = (\mathcal{H}, n, m, m', \text{"strategy"})$

Store: h, A



INSERT(x): $A[h(x)] \leftarrow 1$

LOOKUP(x): $A[h(x)] \overset{?}{=} 1$

REMOVE(x): $A[h(x)] \leftarrow 0$

Ideally

SPACE: $O(\log|\mathcal{H}| + m')$

h chosen once and for all when creating the data structure: $h \leftarrow \mathcal{H} \leftarrow$ hash family

Can set $m' = O(n)$?

Collision: $x, x' \in S$, $x \neq x'$ but $h(x) = h(x')$

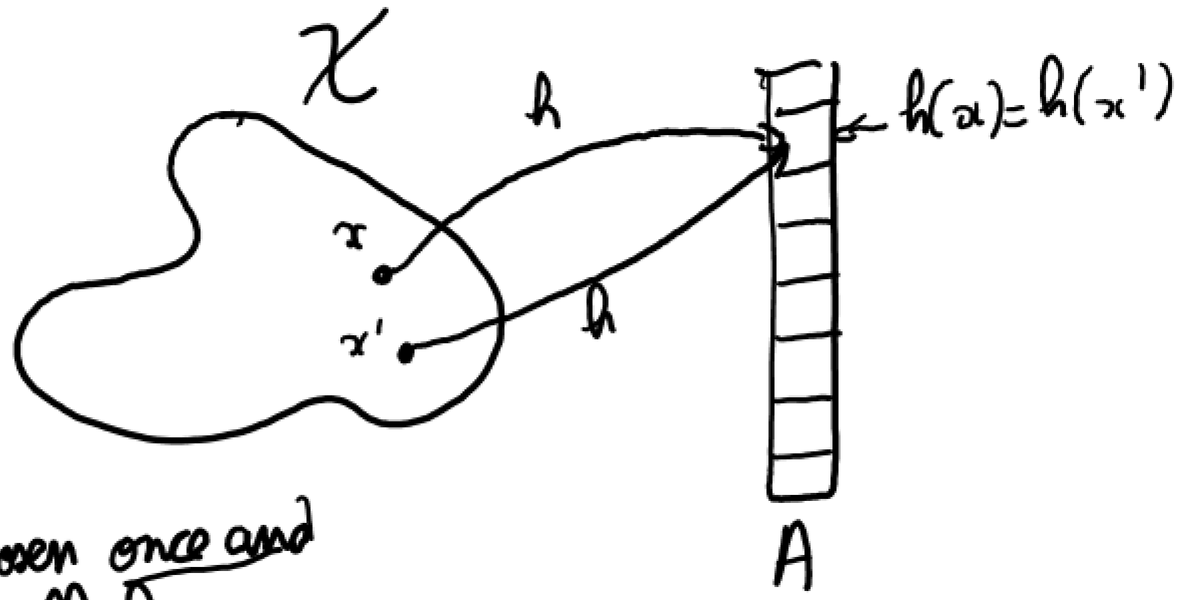
Hash tables: no collisions? 🎂

$\cancel{m' = \Omega(n)}$ "perfect hashing"

- Birthday paradox: $m' = \Omega(n^2)$
- Worse: if $m' = O(n)$, even if h was chosen truly uniformly at random there would be $\Theta\left(\frac{\log n}{\log \log n}\right)$ collisions in some bucket

Need a strategy to handle collisions.

Hash tables: ~~no~~ collisions 💥



h chosen once and for all from a "good" hash family $\mathcal{H}$

(h u.a.r. in $\mathcal{H}$)

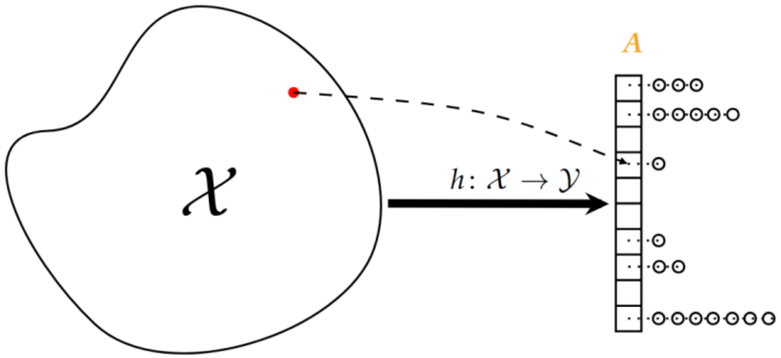
$|A| = m' = O(n)$

1. Store x in $A[h(x)]$, not just 0 or 1 (allows to detect collisions)

2. Strategy:
   - Chaining
   - Open addressing

Hash tables: collisions

Handling collisions: separate chaining



INSERT(x):
A[h(x)].INSERT(x)

LOOKUP(x):
A[h(x)].LOOKUP(x)

REMOVE(x):
A[h(x)].REMOVE(x)

Load $\alpha = \frac{n}{m}$,

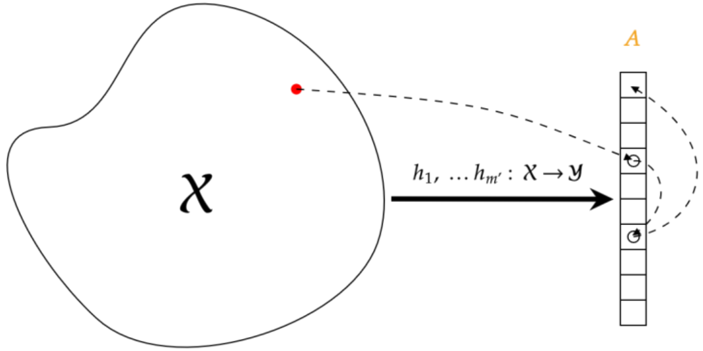
$\boxed{\text{Can have } \alpha > 1}$

Claim: all 3 have expected time complexity $O(1+\alpha)$

(worst case $O(n)$)

even the initial choice of h

Handling collisions: open addressing



Assume we have $h_1, \dots, h_{m'}$ hash functions "suitable"

INSERT(x):
For $t = 1$ to m':
if $A[h_t(x)] = x$ return
else if $A[h_t(x)] = \emptyset$ or $A[h_t(x)] = \perp$ then $A[h_t(x)] \leftarrow x$; return

$\boxed{\text{Need } \alpha \leq 1}$

LOOKUP(x):
For $t = 1$ to m':
if $A[h_t(x)] = x$ then return yes
else if $A[h_t(x)] = \emptyset$ then return no

REMOVE(x):
For $t = 1$ to m':
if $A[h_t(x)] = x$ then $A[h_t(x)] = \perp$
else if $A[h_t(x)] = \emptyset$ then return

Handling collisions: open addressing

Theorem. Assuming stuff, the expected time cplxty of LOOKUP is

$$O\left(\frac{1}{1-\alpha}\right).$$

stuff: $\forall x$, $(h_1(x), \ldots, h_{m'}(x))$ is a u.a.r. permutation of $[m']$.

Proof.

$$\mathbb{E}[T(n,m')] = O(1) + \alpha \cdot \mathbb{E}[T(n-1, m'-1)]$$

$$\leq O(1) + \alpha\, \mathbb{E}[T(n,m')]$$

"handwave"

$$\rightsquigarrow \mathbb{E}[T(n,m')] = O\left(\frac{1}{1-\alpha}\right)$$

Handling collisions: open addressing (linear probing)

$\cancel{h_1, \ldots, h_{m'}} \longrightarrow \boxed{h}$

$$h_1(x) = h(x)$$
$$h_2(x) = h(x) + 1 \quad [m']$$
$$h_3(x) = h(x) + 2 \quad [m']$$
$$\vdots$$
$$h_{m'}(x) = h(x) + m' - 1 \quad [m']$$

Theorem. (Knuth '62) Under some reasonable assumption, expected time cplxty of LOOKUP is

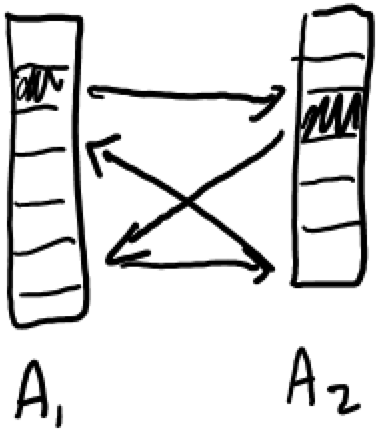
$$O\left(\frac{1}{(1-\alpha)^2}\right).$$

Handling collisions: open addressing (cuckoo hashing)

h_1, A_1
h_2, A_2

LOOKUP, REMOVE : $O(1)$ time (worst-case) !

INSERT : $O(1)$ expected



* Each inserted x is either in $A_1[h_1(x)]$ or $A_2[h_2(x)]$
  → only need to check 2 locations for LOOKUP and REMOVE

* INSERT can take longer:
  ① Try to insert x in $A_1[h_1(x)]$: full
  ② Try to insert x in $A_2[h_2(x)]$: full, so "evict" the y that was there and put x instead
  ③ Try to insert y in $A_1[h_1(y)]$: full, so "evict" the z that was there and put y instead
  [...]

Key fact (complicated to prove): there "eviction sequences" are short in expectation.

Hash tables: summary

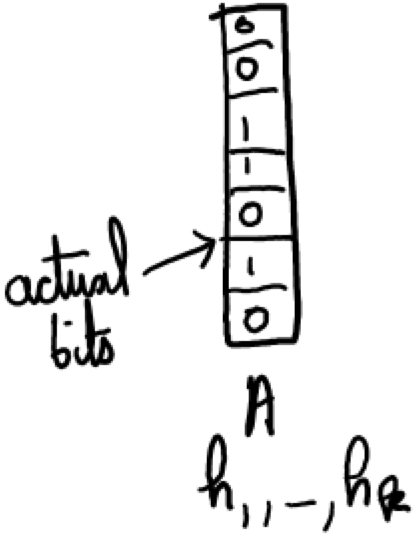
- h is not "a truly random function" but chosen once and for all, at random, from a small hash family $\mathcal{H}$ (chosen by the algorithm designer) usually at least a universal hash family
- h is then kept as part of the data structure: $O(\log|\mathcal{H}|)$ bits
- the data itself is not assumed to be random: only the choice of h from $\mathcal{H}$ is.
- no silver bullet: things are only good in expectation (or with high probability)

Can we do better? Bloom filters!

Faster: $O(1)$ time complexity in the worst case (for real)

Less space: (by constant factors)
at least

But LOOKUP is sometimes wrong: false positives



$$\text{LOOKUP}(x) = A[h_1(x)] \wedge A[h_2(x)] \wedge \ldots \wedge A[h_k(x)]$$

(k: controls space usage + probability of false positives)

to choose (parameter)

(More in tutorial)

Perspective: why does it work so well? 🤷

"Hashing is fundamental to many algorithms and data structures widely used in practice.For theoretical analysis of hashing, there have been two main approaches. First, one can assume that the hash function is truly random, mapping each data item independently and uniformly to the range. This idealized model is unrealistic because a truly random hash function requires an exponential number of bits to describe. Alternatively, one can provide rigorous bounds on performance when explicit families of hash functions are used, such as 2-universal or O(1)-wise independent families. For such families, performance guarantees are often noticeably weaker than for ideal hashing.

In practice, however, it is commonly observed that simple hash functions, including 2-universal hash functions, perform as predicted by the idealized analysis for truly random hash functions."

"Why Simple Hash Functions Work: Exploiting the Entropy in a Data Stream."
Mitzenmacher and Vadhan, 2008