

# Hashing-based data structures

The most important techniques behind Yahoo! are hashing,  
hashing and hashing! – Udi Manber

# What is this course about

- ▶ Hashing is absolutely fundamental in both theory and practice of algorithm design
- ▶ Focus on *a family* of hashing-based data structures with different functionalities but based on common ideas
  - ▶ Hash tables (containers)
  - ▶ Filters
  - ▶ Other functionalities
- ▶ Applying common mathematical model of random hypergraphs

# Hash tables

- ▶ *Goal*: maintain a (possibly evolving) set of objects belonging to a large “universe”
- ▶ Objects are specified by *keys* (e.g. configurations, ID numbers, words, etc.)  $\Rightarrow$  *associative array*
- ▶ *Applications*: data containers, indexing, deduplication, data bases, path finding, compilers, etc.

# Hash tables

## ► Notation

- $\mathcal{U}$  : universe of all possible keys (*Examples:* strings, IP addresses, game configurations, ...)
- $S$  : subset of keys (actually stored in the dictionary),  $|S| \ll |\mathcal{U}|$
- $|S| = n$

## ► Supported operations:

- $\text{INSERT}(x)$  : add  $x \in \mathcal{U}$
- $\text{DELETE}(x)$  : delete  $x$
- $\text{LOOKUP}(x)$  : find/access  $x$

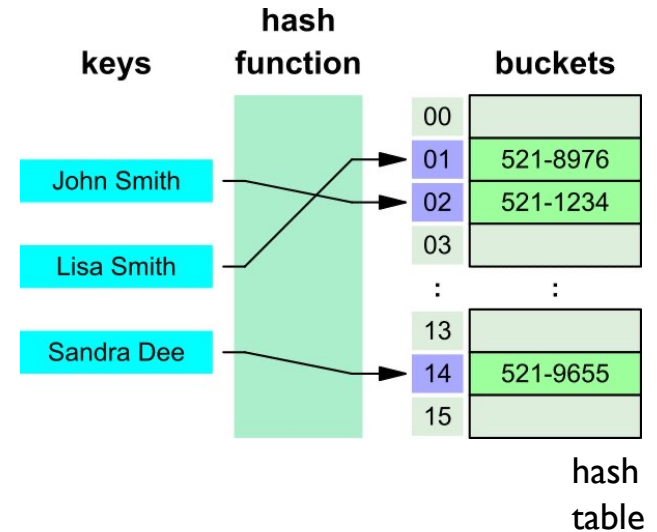
*Dictionary* data  
structure

# Hash functions

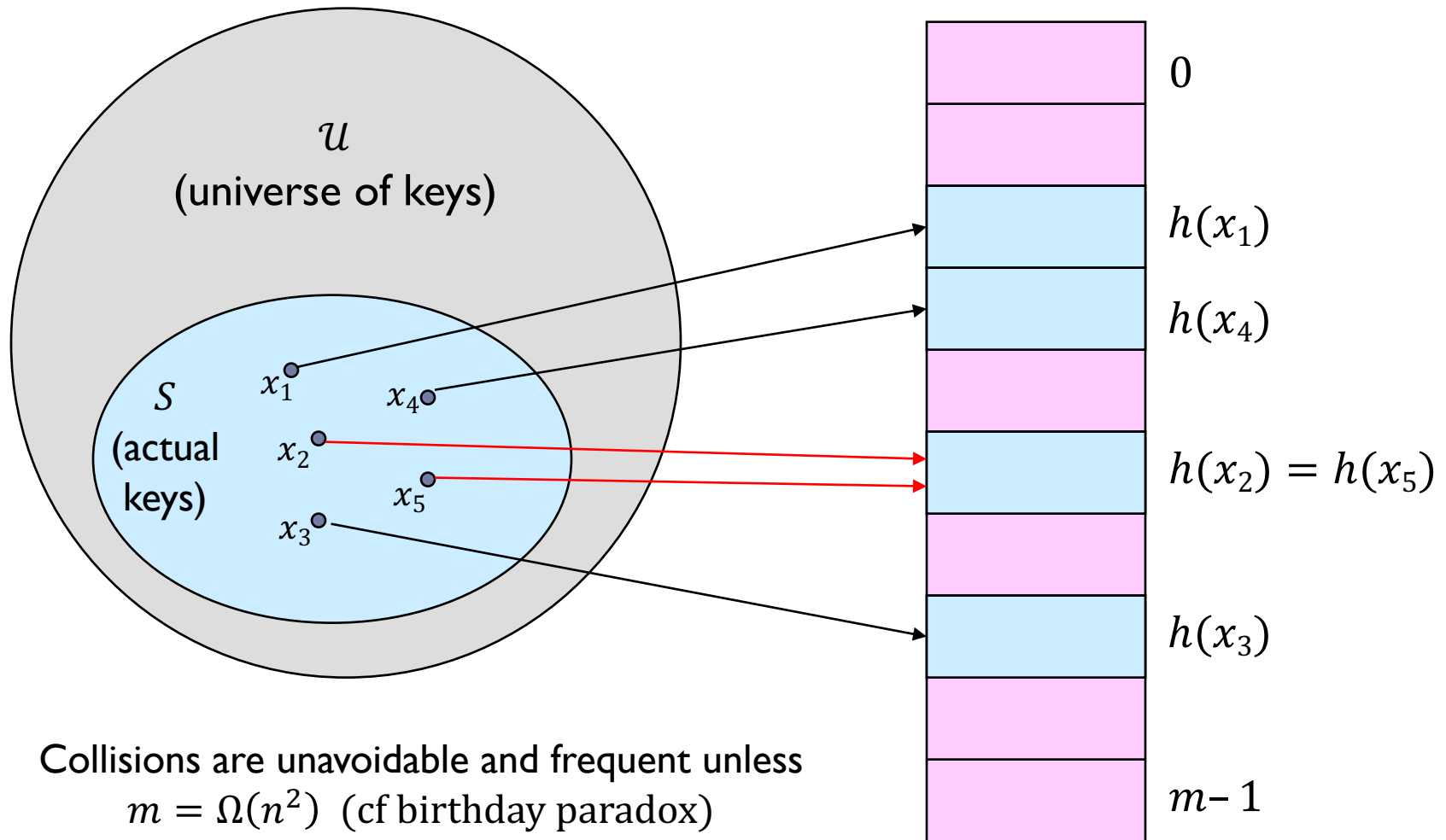
- ▶ **Hash function  $h$ :** Mapping from  $\mathcal{U}$  to entries of hash table  $T[0..m-1]$ .

$$h : \mathcal{U} \rightarrow \{0, 1, \dots, m-1\}$$

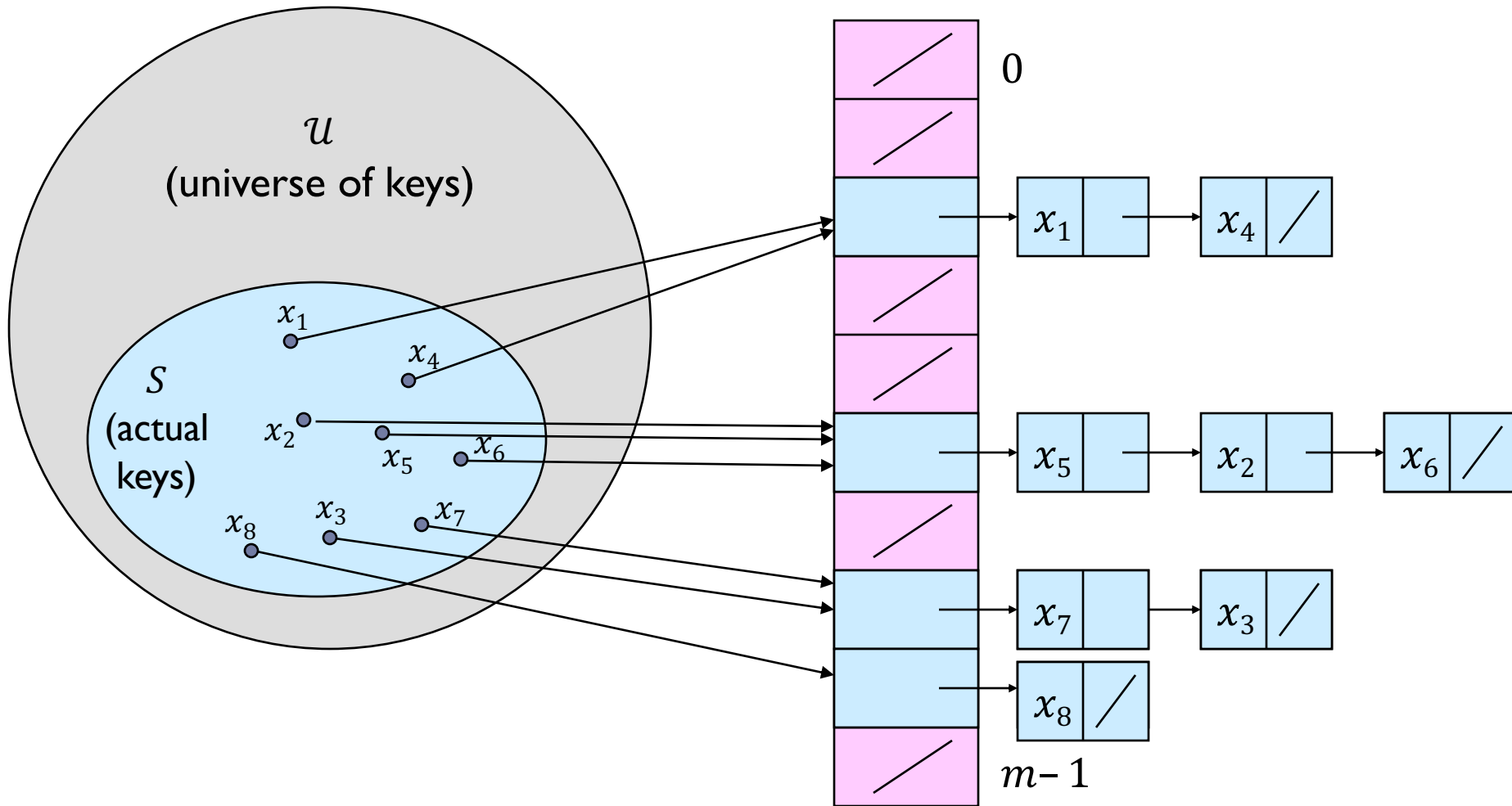
- ▶  $h(x)$  is the **hash value** (or simply **hash**) of key  $x$



# Hashing and collisions



# I. Collision Resolution by Chaining



# Hashing with chaining

- ▶  $\alpha = n/m$  load factor (in practice, a constant close to 1)
- ▶  $\text{INSERT}(x) : O(1)$
- ▶  $\text{LOOKUP}(x), \text{DELETE}(x) : O(1 + \alpha)$  *in expectaton*, assuming *uniform hashing*
  - ▶  $\Rightarrow O(1)$  if  $n = O(m)$
  - ▶ in contrast to other containers (lists, search trees, ...) hash tables provide *expected constant-time* operations



## II. Collision Resolution by Open Addressing

- ▶ All elements are stored in the hash table itself
- ▶  $\Rightarrow \alpha = \frac{n}{m} < 1$ , no pointers
- ▶ hash function  $h(x, i)$  where  $i = 0, 1, 2, \dots, m - 1$ , and  $\langle h(x, 0), h(x, 1), \dots, h(x, m - 1) \rangle$  is a permutation
- ▶ when inserting/looking up  $x$ , probe  $h(x, 0), h(x, 1), \dots$  (*probe sequence*) until
  - ▶ we find  $x$ , or
  - ▶ the bucket contains *nil*, or
  - ▶  $m$  buckets have been unsuccessfully probed
- ▶ deletion is complicated, needs a special key "deleted", time may not be dependent on the load factor

# Performance of Open Addressing

- Assuming that  $\langle h(x, 0), h(x, 1), \dots, h(x, m - 1) \rangle$  is a random permutation (uniformly drawn), the expected number of probes in an insertion (or unsuccessful search) with open addressing is

$$1/(1 - \alpha),$$

where  $\alpha = n/m$  the load factor

- Proof:*

let  $p_i = P[i \text{ first buckets are full}] = \alpha^i \quad (p_0 = 1)$

$$\begin{aligned} E[\text{number of probes}] &= 1 + \sum_{i=1}^{m-1} i \cdot P[i - 1 \text{ full} \\ &\quad \text{buckets followed by an empty one}] = \\ &= \sum_{i=1}^{m-1} i \cdot (p_{i-1} - p_i) = 1 + \sum_{i=1}^{m-1} p_i \approx \end{aligned}$$

$$1 + \alpha + \alpha^2 + \alpha^3 + \dots = 1/(1 - \alpha)$$

# Performance of Open Addressing

- ▶ Assuming that  $\langle h(x, 0), h(x, 1), \dots, h(x, m - 1) \rangle$  is a random permutation (uniformly drawn), the expected number of probes in an insertion (or unsuccessful search) with open addressing is

$$1/(1 - \alpha),$$

where  $\alpha = n/m$  the load factor

- ▶ The expected number of probes for a successful search is

$$\frac{1}{n} \sum_{i=0}^{n-1} \frac{1}{1 - \frac{i}{m}} \leq (1/\alpha) \ln(1/(1 - \alpha))$$

# What makes a good hash function

- ▶ easy to store and to compute (in  $O(1)$  time)
- ▶ distributes keys into buckets *as uniformly as possible* (behaves like a "random function")
  - ▶  $h$  is fixed, keys are random
    - ▶ *chaining*: division method, multiplication method
    - ▶ *open addressing*: linear probing, quadratic probing, double hashing
  - ▶ keys are fixed,  $h$  is randomly drawn from some *family of functions*
    - ▶ *practical advantage*:  $h$  can be selected at execution time

# Denial of Service via Algorithmic Complexity Attacks

Scott A. Crosby  
*scrosby@cs.rice.edu*

Dan S. Wallach  
*dwallach@cs.rice.edu*

*Department of Computer Science, Rice University*

## Abstract

We present a new class of low-bandwidth denial of service attacks that exploit algorithmic deficiencies in many common applications' data structures. Frequently used data structures have "average-case" expected running time that's far more efficient than the worst case. For example, both binary trees and hash tables can degenerate to linked lists with carefully chosen input. We show how an attacker can effectively compute such input, and we demonstrate attacks against the hash table implementations in two versions of Perl, the Squid web proxy, and the Bro intrusion detection system. Using bandwidth less than a typical dialup modem, we can bring a dedicated Bro server to its knees; after six minutes of carefully chosen packets, our Bro server was dropping as much as 71% of its traffic and consuming all of its CPU. We show how modern universal hashing techniques can yield performance comparable to commonplace hash functions while being provably secure against these attacks.

sume  $O(n)$  time to insert  $n$  elements. However, if each element hashes to the same bucket, the hash table will also degenerate to a linked list, and it will take  $O(n^2)$  time to insert  $n$  elements.

While balanced tree algorithms, such as red-black trees [11], AVL trees [1], and treaps [17] can avoid predictable input which causes worst-case behavior, and universal hash functions [5] can be used to make hash functions that are not predictable by an attacker, many common applications use simpler algorithms. If an attacker can control and predict the inputs being used by these algorithms, then the attacker may be able to induce the worst-case execution time, effectively causing a denial-of-service (DoS) attack.

Such algorithmic DoS attacks have much in common with other low-bandwidth DoS attacks, such as stack smashing [2] or the ping-of-death<sup>1</sup>, wherein a relatively short message causes an Internet server to crash or misbehave. While a variety of techniques *can* be used to address these DoS attacks, com-



# Breaking Murmur: Hash-flooding DoS Reloaded

Dec 14th, 2012

DISCLAIMER: Do not use any of the material presented here to cause harm. I will find out where you live, I will surprise you in your sleep and I will tickle you so hard that you will promise to behave until the end of your days.

## The story so far

Last year, at 28c3, [Alexander Klink](#) and [Julian Wälde](#) presented [a way to run a denial-of-service attack](#) against web applications.

One of the most impressive demonstrations of the attack was sending crafted data to a web application. The web application would dutifully parse that data into a hash table, not knowing that the data was carefully chosen in a way so that each key being sent would cause a collision in the hash table. The result is that the malicious data sent to the web application elicits worst case performance behavior of the hash table implementation. Instead of amortized constant time for an insertion, every insertion now causes a collision and degrades our hash table to nothing more than a fancy linked list, requiring linear time in the table size for each consecutive insertion. Details can be found in the [Wikipedia article](#).

# Universal hashing

- ▶ **Definition:** A family  $H$  of hash functions is called **universal** iff for any pair of keys  $x \neq y$ ,

$$P_{h \in H}[h(x) = h(y)] \leq 1/m$$

(Equiv., the nb of hash functions  $h$  with  $h(x) = h(y)$  is  $\leq |H|/m$ )

- ▶ **Theorem:** under hashing with chaining, the expected time (over  $h \in H$ ) of INSERT, DELETE, LOOKUP is  $O(1 + \alpha)$

**NB:** no assumption on the distribution of keys

- ▶ Universal class of hash functions first introduced by Carter&Wegman (1979), construction based on elementary number theory

# Universal hash functions: matrix method

- ▶  $\mathcal{U} = [0..2^u - 1], m = 2^w$
- ▶  $M$ : random  $w \times u$  boolean matrix
- ▶  $h_M(x) = Mx$ , where multiplication is done over  $(\oplus, \wedge)$
- ▶ *Example:* 
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$
- ▶ Let  $x, y \in \mathcal{U}, x \neq y$ . Show that  $P_h[h(x) = h(y)] \leq \frac{1}{2^w} = \frac{1}{m}$
- ▶ *Proof:* Assume  $x$  and  $y$  differ in  $i$ -th bit which is 0 in  $x$  and 1 in  $y$ .  $h(x)$  does not depend on the  $i$ -th column of  $h$ .  $h(y)$  are all different over  $2^w$  choices of the  $i$ -th column. Therefore  $h(x) = h(y)$  with probability  $\leq 1/2^w$ .



## Universal hash function: prime table size

- ▶ Assume we are hashing IP addresses  $x_1.x_2.x_3.x_4$  with  $0 \leq x_i \leq 255$
- ▶ Choose  $m > 255$  a prime number
- ▶ Consider quadruples  $a = (a_1, a_2, a_3, a_4)$  with  $0 \leq a_i \leq m - 1$
- ▶ Define

$$h_a(x_1.x_2.x_3.x_4) = (a_1 \cdot x_1 + a_2 \cdot x_2 + a_3 \cdot x_3 + a_4 \cdot x_4) \bmod m$$

- ▶  $H = \{h_a\}$  is a universal family
- ▶ **Proof:** Let  $x = x_1.x_2.x_3.x_4$  and  $y = y_1.y_2.y_3.y_4$ . Assume  $x_3 \neq y_3$ . If  $h(x) = h(y)$ , then

$$\begin{aligned} & a_3(x_3 - y_3) \\ &= a_1(x_1 - y_1) + a_2(x_2 - y_2) + a_4(x_4 - y_4) \bmod m \end{aligned}$$

Since  $x_3 - y_3 \neq 0$  and  $m$  is prime,  $a_3$  is uniquely defined. Since  $a_3$  is chosen at random,  $P_h[h(x) = h(y)] \leq \frac{1}{m}$

# Multiply-shift universal hashing

- ▶ *c-universal* family  $H$ : for any pair of keys  $x \neq y$ ,


$$P_{h \in H}[h(x) = h(y)] \leq c/m$$

- ▶ Multiply-shift hashing is 2-universal [[Dietzfelbinger et al. A reliable randomized algorithm for the closest-pair problem, J.Algorithms, 25:19–51, 1997](#)]

$h_a: [0..2^u - 1] \rightarrow [0..2^w - 1]$  defined by

$h_a(x) = \lfloor (ax \bmod 2^u) / 2^{u-w} \rfloor$  for a random  $u$ -bit odd integer  $a$

- ▶ for more details [[M.Thorup, High Speed Hashing for Integers and Strings, arxiv:1504.06804, 2019](#)]



# Perfect hashing



# Motivation

- ▶ Can we guarantee a *worst-case*  $O(1)$  time for hash table operations?
- ▶ *Yes* if the set of keys is *static*
- ▶ Naive solution: sorting
  - ▶ construction  $O(n \log n)$
  - ▶ space  $O(n)$
  - ▶ lookup  $O(\log n)$

*next slides*

$O(n)$  *expected*

$O(n)$

$O(1)$

*hash function takes  
 $O(n)$  bits to store*

# Collisions: analysis

- ▶ What is the expected number of collisions?
  - ▶ i.e. number of pairs  $(x, y), x \neq y$  and  $h(x) = h(y)$
- ▶  $X_{xy} = 1$  iff  $h(x) = h(y)$
- ▶  $E\left[\sum_{x \neq y} X_{xy}\right] = \sum_{x \neq y} E[X_{xy}] = \binom{n}{2} \frac{1}{m} \approx \frac{n^2}{2m}$

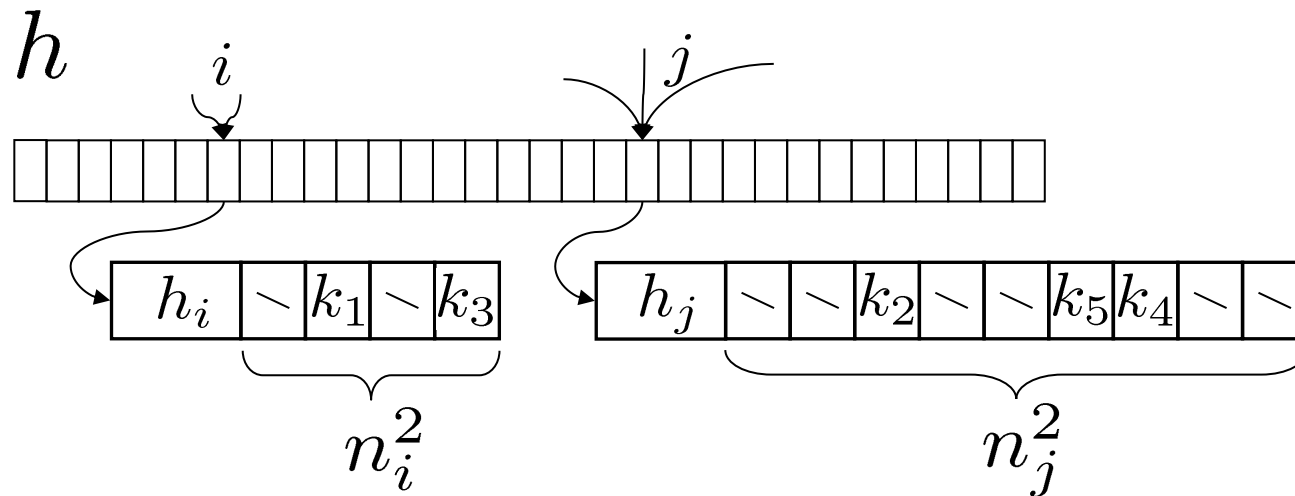
## ▶ Remarks:

- ▶ if  $m \approx n^2$ , then we have  $\frac{1}{2}$  expected collisions  $\Rightarrow$   
 $P[\exists \text{ collision}] \leq \frac{1}{2}$  (cf birthday paradox)
- ▶ by iterating, we can build a hash table with NO collision after  $O(1)$  trials *in expectation*

Markov inequality $P[X \geq a] \leq \frac{E[X]}{a}$
--

# Perfect hashing: 2-level scheme

- ▶ Fredman, Komlós, Szemerédi (1984)
- ▶ Guarantees  $O(1)$  *worst-case* time of LOOKUP for a *static* set of keys. Solution uses universal hashing.
- ▶ 2-level hash scheme:



- ▶ LOOKUP: *worst-case*  $O(1)$

Why  $\sum n_i^2$  can be made  $\leq 2n$

▶  $\sum n_i^2 = n + 2 \cdot \sum \binom{n_i}{2} \leftarrow \# \text{ of colliding pairs}$

▶  $E[\# \text{ of colliding pairs}] = \binom{n}{2} \frac{1}{n} \approx \frac{n}{2}$

▶  $\Rightarrow E[\sum n_i^2] = 2n \Rightarrow P[\sum n_i^2 > 4n] < 1/2$   
by Markov inequality

▶ Algorithm (sketch)

▶ hash to primary table of size  $O(n)$  using a universal h.f.  $h$

▶ hash each non-empty bucket to a table of size  $n_i^2$ ; if  $\sum n_i^2 > 4n$ , rehash

▶ using a universal h.f.  $h_i$ ; if collision, rehash until there is none (expected  $O(1)$  time by birthday paradox)

# Perfect hashing is practical

- ▶ practical implementations exist, e.g. `gperf` in C++



# *Minimal* Perfect Hash Functions (MPHF)

- ▶ MPHF: bijective perfect hash function, i.e.  $h: S \rightarrow [1..|S|]$
- ▶ enumeration of keys of  $S$
- ▶ efficient construction algorithms of MPFH exist, see e.g. [Fox et al., Comm.ACM 32, 1992]
- ▶ space efficiency: a few bits per key
- ▶ lower bound  $\approx 1.44$  bits/key, cf [Mehlhorn 82], [Belazzougui, Botelho, Dietzfelbinger 09]



# Cuckoo hashing

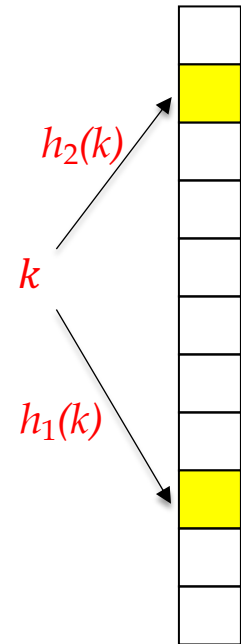
(perfect hashing with open addressing)

# Power of choice : Two-choice hashing

- ▶  $n$  keys,  $n$  buckets
- ▶ if each key is placed to a random bucket, the fullest bucket will have  $O(\log n / \log \log n)$  keys, w.h.p.
- ▶ [\[Azar et al. 2000\]](#) if each key is mapped to two random buckets and the less full bucket is chosen, the fullest bucket will have  $O(\log \log n)$  keys, w.h.p.

# Cuckoo hashing

- ▶ Introduced by Pagh&Rodler in 2001
- ▶ uses two independent hash functions  $h_1$  and  $h_2$
- ▶ LOOKUP( $x$ ): check buckets  $T[h_1(x)]$  and  $T[h_2(x)]$



# Cuckoo hashing

- ▶ Introduced by Pagh&Rodler in 2001
- ▶ uses two independent hash functions  $h_1$  and  $h_2$
- ▶ LOOKUP( $x$ ): check buckets  $T[h_1(x)]$  and  $T[h_2(x)]$
- ▶ INSERT( $x$ ):

$pos \leftarrow h_1(x)$

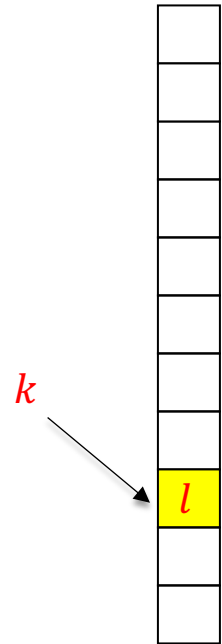
**loop**

**if**  $T[pos]$  is empty **then**

$T[pos] \leftarrow x$ ; **return**

**swap** values of  $x$  and  $T[pos]$

$pos \leftarrow$  alternative position for  $x$



# Cuckoo hashing

- ▶ Introduced by Pagh&Rodler in 2001
- ▶ uses two independent hash functions  $h_1$  and  $h_2$
- ▶ LOOKUP( $x$ ): check buckets  $T[h_1(x)]$  and  $T[h_2(x)]$
- ▶ INSERT( $x$ ):

$pos \leftarrow h_1(x)$

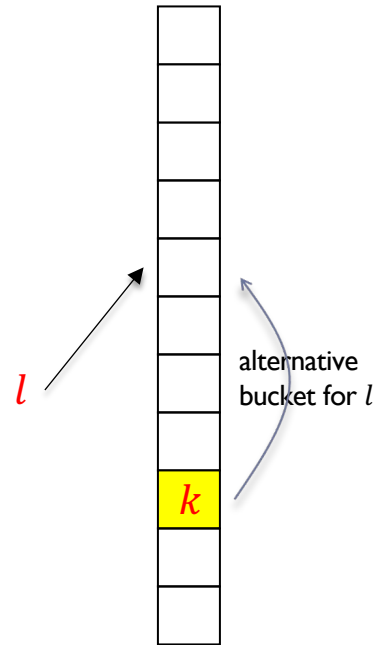
**loop**

**if**  $T[pos]$  is empty **then**

$T[pos] \leftarrow x$ ; **return**

**swap** values of  $x$  and  $T[pos]$

$pos \leftarrow$  alternative position for  $x$



# Cuckoo hashing

- ▶ Introduced by Pagh&Rodler in 2001
- ▶ uses two independent hash functions  $h_1$  and  $h_2$
- ▶ LOOKUP( $x$ ): check buckets  $T[h_1(x)]$  and  $T[h_2(x)]$
- ▶ INSERT( $x$ ):

$pos \leftarrow h_1(x)$

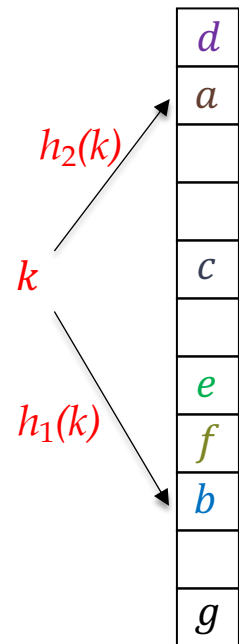
**loop**

**if**  $T[pos]$  is empty **then**

$T[pos] \leftarrow x$ ; **return**

**swap** values of  $x$  and  $T[pos]$

$pos \leftarrow$  alternative position for  $x$



# Cuckoo hashing

- ▶ Introduced by Pagh&Rodler in 2001
- ▶ uses two independent hash functions  $h_1$  and  $h_2$
- ▶ LOOKUP( $x$ ): check buckets  $T[h_1(x)]$  and  $T[h_2(x)]$
- ▶ INSERT( $x$ ):

$pos \leftarrow h_1(x)$

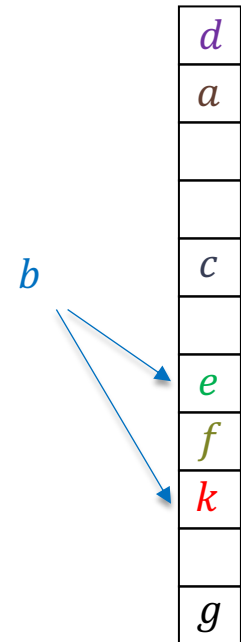
**loop**

**if**  $T[pos]$  is empty **then**

$T[pos] \leftarrow x$ ; **return**

**swap** values of  $x$  and  $T[pos]$

$pos \leftarrow$  alternative position for  $x$





# Cuckoo hashing

- ▶ Introduced by Pagh&Rodler in 2001
- ▶ uses two independent hash functions  $h_1$  and  $h_2$
- ▶ LOOKUP( $x$ ): check buckets  $T[h_1(x)]$  and  $T[h_2(x)]$
- ▶ INSERT( $x$ ):

$pos \leftarrow h_1(x)$

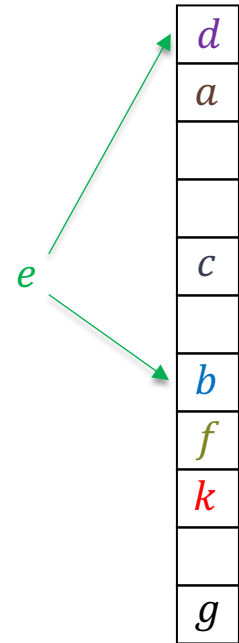
**loop**

**if**  $T[pos]$  is empty **then**

$T[pos] \leftarrow x$ ; **return**

**swap** values of  $x$  and  $T[pos]$

$pos \leftarrow$  alternative position for  $x$



# Cuckoo hashing

- ▶ Introduced by Pagh&Rodler in 2001
- ▶ uses two independent hash functions  $h_1$  and  $h_2$
- ▶ LOOKUP( $x$ ): check buckets  $T[h_1(x)]$  and  $T[h_2(x)]$
- ▶ INSERT( $x$ ):

$pos \leftarrow h_1(x)$

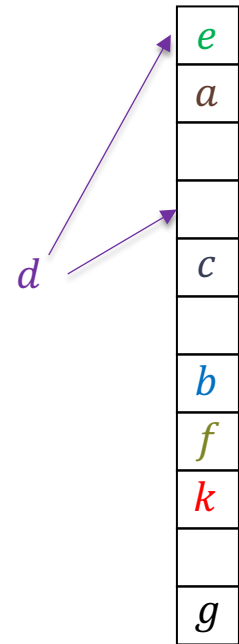
**loop**

**if**  $T[pos]$  is empty **then**

$T[pos] \leftarrow x$ ; **return**

**swap** values of  $x$  and  $T[pos]$

$pos \leftarrow$  alternative position for  $x$



# Cuckoo hashing

- ▶ Introduced by Pagh&Rodler in 2001
- ▶ uses two independent hash functions  $h_1$  and  $h_2$
- ▶ LOOKUP( $x$ ): check buckets  $T[h_1(x)]$  and  $T[h_2(x)]$
- ▶ INSERT( $x$ ):

$pos \leftarrow h_1(x)$

**loop**

**if**  $T[pos]$  is empty **then**

$T[pos] \leftarrow x$ ; **return**

**swap** values of  $x$  and  $T[pos]$

$pos \leftarrow$  alternative position for  $x$

<i>e</i>
<i>a</i>
<i>d</i>
<i>c</i>
<i>b</i>
<i>f</i>
<i>k</i>
<i>g</i>

# Cuckoo hashing

- ▶ Introduced by Pagh&Rodler in 2001
- ▶ uses two independent hash functions  $h_1$  and  $h_2$
- ▶ LOOKUP( $x$ ): check buckets  $T[h_1(x)]$  and  $T[h_2(x)]$
- ▶ INSERT( $x$ ):

$pos \leftarrow h_1(x)$

**loop**  $n$  times

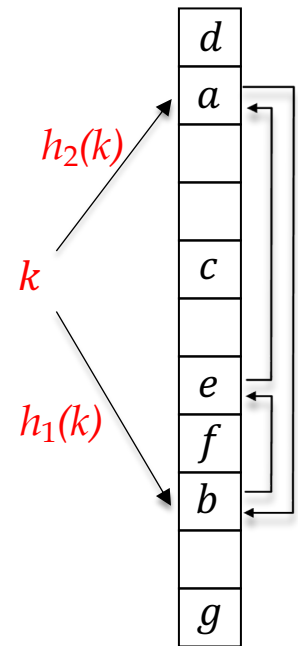
**if**  $T[pos]$  is empty **then**

$T[pos] \leftarrow x$ ; **return**

**swap** values of  $x$  and  $T[pos]$

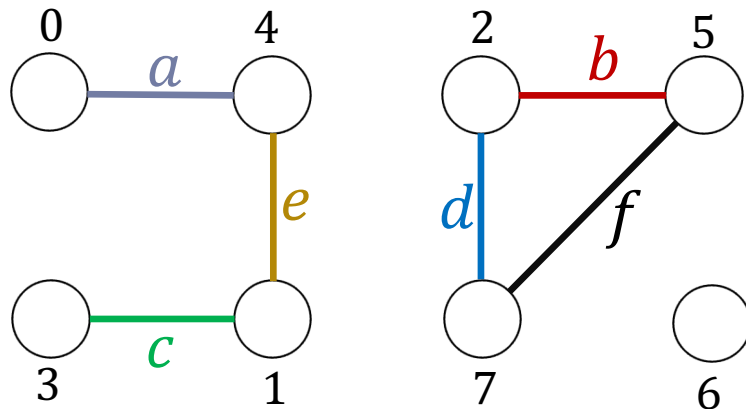
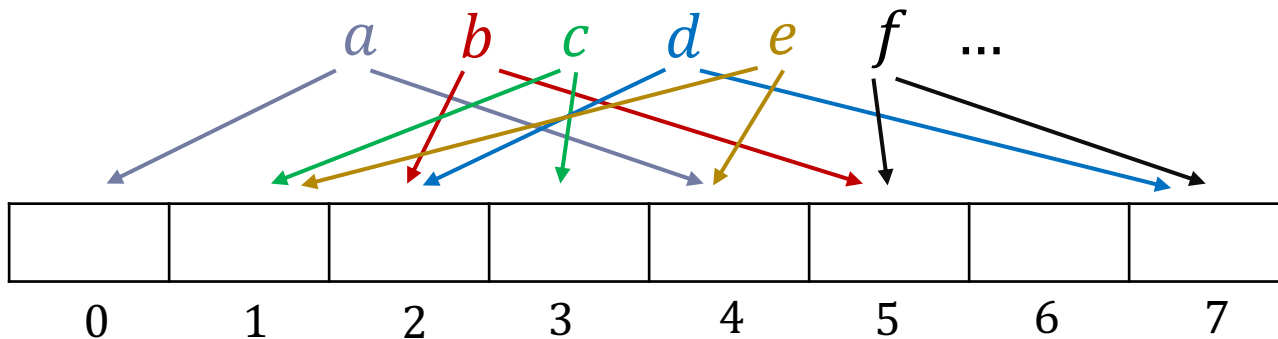
$pos \leftarrow$  alternative position for  $x$

rehash from scratch



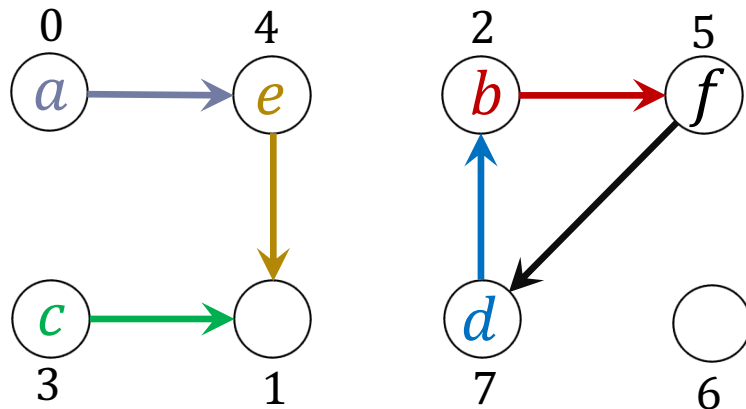
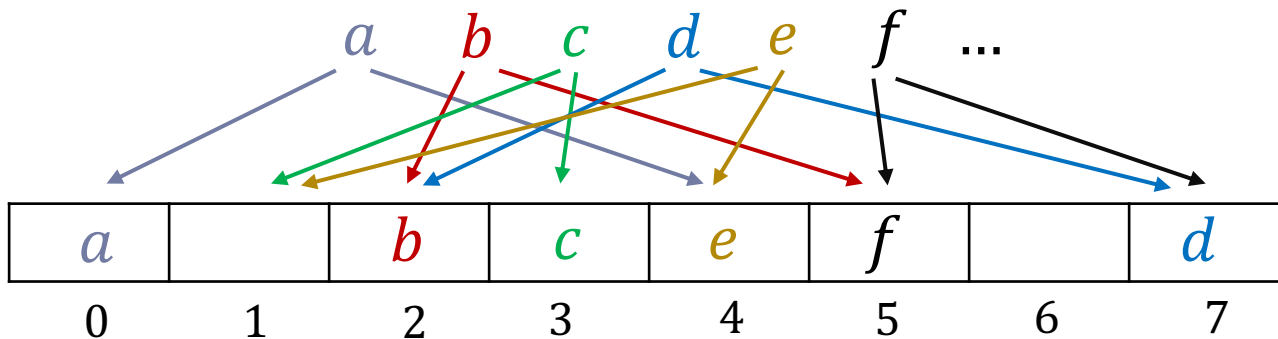
# Cuckoo graph

- Once  $h_1$  and  $h_2$  are selected, each set of keys induces a graph



# Cuckoo graph

- Once  $h_1$  and  $h_2$  are selected, each set of keys induces a graph, and a placement induces an orientation



# Analysis of cuckoo hashing

- ▶ *Def*: an undirected graph is *orientable* if all edges can be directed so that the outdegree of each node is  $\leq 1$
- ▶ **Q1**: how likely that a random Erdos-Rényi graph with  $m$  nodes and  $n$  edges is orientable?
- ▶ **Q2**: how likely is a rehash? what is the expected time of an insertion? what is the expected time of successively building a hash table? ..

# Analysis of cuckoo hashing: orientability

- ▶ Orientability of a random graph  $G_{n,m}$  undergoes a *phase transition* depending on load factor  $\alpha = n/m$ :
  - ▶ for  $\alpha < 1/2$ ,  $P[G_{n,m} \text{ is orientable}] \rightarrow 1$  as  $n \rightarrow \infty$
  - ▶ for  $\alpha > 1/2$ ,  $P[G_{n,m} \text{ is orientable}] \rightarrow 0$  as  $n \rightarrow \infty$
- ▶ Therefore, we should allocate  $m > 2n$



# Analysis of cuckoo hashing: time

- ▶ Precise analysis is somewhat complicated
- ▶ Simplification:
  - ▶ consider undirected graphs (instead of directed)
  - ▶ analyze the probability of a path of length  $d$  between two nodes (this covers cycles)

## Cost of insertions: How many iterations?

*Theorem:* if  $n < \frac{m}{2(1+\delta)}$ , then  $P[\text{shortest path from } i \text{ to } j \text{ is of length } d] \leq \frac{1}{m} (1 + \delta)^{-d}$

$$\Rightarrow P[j \text{ is accessible from } i] = O\left(\frac{1}{m}\right)$$

$$\Rightarrow \text{number of accessible nodes is } n \cdot O\left(\frac{1}{m}\right) = O(1)$$

*Proof:*

- ▶  $d = 1 \Rightarrow P[\text{an edge connects } i \text{ and } j] \leq \frac{2}{m^2} \Rightarrow P[\text{exists an edge between } i \text{ and } j] \leq \frac{2n}{m^2} < \frac{1}{m} (1 + \delta)^{-1}$
- ▶ *induction:*  $d \Rightarrow d + 1$ 
  - ▶ for a fixed  $k$ ,  $\frac{1}{m} (1 + \delta)^{-d} \cdot \frac{1}{m} (1 + \delta)^{-1} = \frac{1}{m^2} (1 + \delta)^{-(d+1)}$
- ▶ summing over  $m$  possibilities for  $k$ , we obtain  $\frac{1}{m} (1 + \delta)^{-(d+1)}$

# Cost of rehashing: how likely is a cycle?

- ▶ If  $n < \frac{m}{2(1+\delta)}$ , careful analysis shows that the probability of a rehash is  $O\left(\frac{1}{n^2}\right)$
- ▶ A rehash involves  $n$  insertions, each taking expected  $O(1)$  time  $\Rightarrow$  amortized cost of rehashing is  $O\left(\frac{1}{n}\right)$  time per insertion
- ▶ for more details see

<https://www.itu.dk/people/pagh/papers/cuckoo-undergrad.pdf>

<http://www.cs.utoronto.ca/~noahfleming/CuckooHashing.pdf>

# Cuckoo hashing: summary

- ▶ LOOKUP, DELETE: worst-case  $O(1)$  (two probes)
- ▶ INSERT: *expected*  $O(1)$
- ▶ deletions supported
- ▶ no dynamic memory allocation (as in chaining)
- ▶ reasonable memory use, but load factor  $< 1/2$
- ▶ generalization:  $(H, b)$ -cuckoo hashing
  - ▶  $H$  hash functions (instead of 2), each bucket carries  $b$  items
  - ▶ admits a much higher load factor, e.g.  $(3,4)$ -tables admits load factor of over 99.9% [[Walzer, ICALP 2018](#)]

# Critical load for $H > 2$ or $b > 1$

$b = 1$

$H$	2	3	4	5	6	7
critical load (orientability threshold)	0.5	0.918	0.976	0.992	0.997	0.999

$H = 2$

$b$	1	2	3	4	5	8	10
critical load	0.5	0.897	0.959	0.980	0.989	0.997	0.999

# Don't Throw Out Your Algorithms Book Just Yet: Classical Data Structures That Can Outperform Learned Indexes

by Peter Bailis, Kai Sheng Tai, Pratiksha Thaker, and Matei Zaharia

11 Jan 2018

There's recently been a lot of excitement about a new proposal from authors at Google: to replace conventional indexing data structures like B-trees and hash maps by instead fitting a neural network to the dataset. The paper compares such learned indexes against several standard data structures and reports promising results. For range searches, the authors report up to 3.2x speedups over B-trees while using 9x less memory, and for point lookups, the authors report up to 80% reduction of hash table memory overhead while maintaining a similar query time.

While learned indexes are an exciting idea for many reasons (e.g., they could enable self-tuning databases), there is a long literature of other optimized data structures to consider, so naturally researchers have been trying to see whether these can do better. For example, Thomas Neumann posted about using spline interpolation in a B-tree for range search and showed that this easy-to-implement strategy can be competitive with learned indexes. In this post, we examine a second use case in the paper: memory-efficient hash tables. We show that for this problem, a simple and beautiful data structure, the cuckoo hash, can achieve 5-20x less space overhead than learned indexes, and that it can be surprisingly fast on modern hardware, running nearly 2x faster. These results are interesting because the cuckoo hash is *asymptotically* better than simpler hash tables at load balancing, and thus makes optimizing the hash function using machine learning less important: it's always great to see cases where beautiful theory produces great results in practice.

## Going Cuckoo for Fast Hash Tables

Let's start by understanding the hashing use case in the learned indexes paper. A typical hash function distributes keys randomly across the slots in a hash table, causing some slots to be empty, while others have collisions, which require some form of chaining of items. If lots of memory is available, this is not a problem: simply create many more slots than there are keys in the table (say, 2x) and collisions will be rare. However, if memory is scarce, heavily loaded tables will result in slower lookups due to more chaining. The authors show that, by learning a hash function that *spreads the input keys more evenly* throughout