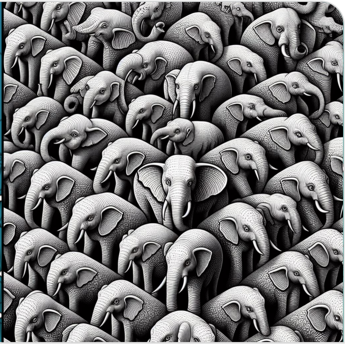
Karlsruhe Institute of Technology

Probability and Computing – Perfect Hashing

Hans-Peter Lehmann | WS 2024/2025





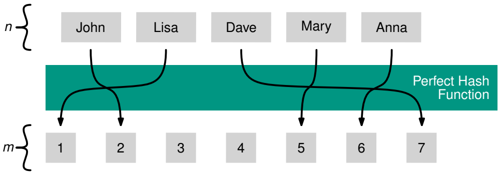
Content

The Perfect Hashing Problem

Perfect hashing data type (for universe D, $\varepsilon \geq 0$)

construct(S):
- input: $S \subseteq D$ of size $n = |S|$
- output: data structure P.

eval$_P(x)$:
- input: $x \in D$
- output: a number in $[m]$ where $m = (1 + \varepsilon)n$
- requirement: $x \mapsto \textbf{eval}_P(x)$ is injective on S



Remarks

- details about S are lost.
- note: P is "perfect hash function" but need not be random

Goals

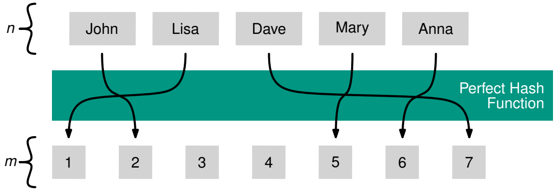
- ε is small // $\varepsilon = 0$: *Minimal* perfect hashing
- space requirement of P is $\mathcal{O}(n)$ bits
- ideally: running time of **eval**$_P$ is $\mathcal{O}(1)$
- ideally: running time of **construct** is $\mathcal{O}(n)$

Motivation for (Minimal-) Perfect Hashing

Short IDs

Replace keys with short unique identifies

$$\textbf{eval}_P(\text{“}\texttt{CreativeUserName}\text{”}) = 10241.$$



Updatable Retrieval: A hash table without keys

- assume we have MPHF P for S
- can store additional data $f(x) \in [k]$ on $x \in S$ in array of length m in position $\textbf{eval}_P(x)$.
  $\hookrightarrow$ array takes $m\lceil \log_2(k) \rceil$ bits

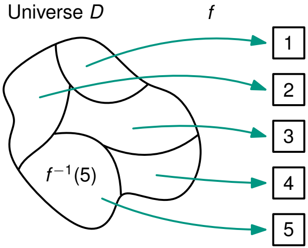
⚠ Weaker than a normal hash table:

- S is static (values updateable)
- trying to access $f(x)$ for $x \notin S$ gives undefined result
- trying to update $f(x)$ for $x \notin S$ destroys information

Minimal Perfect Hashing: Lower Space Bound

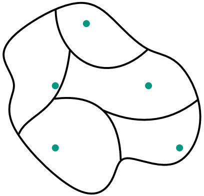
Ingredients

- $a = \binom{|D|}{n}$ possible input sets
- Each function can cover at most $b = \left(\frac{|D|}{n}\right)^n$ different inputs
- Need to differentiate between at least a/b different behaviors



$$\log_2\left(\frac{\binom{|D|}{n}}{\left(\frac{|D|}{n}\right)^n}\right) \overset{\text{Stirling}}{\approx} \log_2\left(\frac{\left(\frac{|D|e}{n}\right)^n}{\left(\frac{|D|}{n}\right)^n}\right)$$

$$= \log_2\left(e^n\right) = n\log_2 e \approx 1.44n$$



Possible input set for which f is an MPHF

- In contrast, storing S might need $\Omega(n\log(|D|))$ bits

Content

Construction Using Trial and Error

Exercise: What if we played the lottery until we win?

Let us try random hash functions until one is *minimal perfect* $(n = m)$ on S.

- What are the expected construction time and space consumption?
- Hint: Stirling's approximation: $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

Solution:

- There are n^n different random hash functions.
- $n!$ of those are minimal perfect on S.
- Success after trying $\frac{n^n}{n!} \approx e^n/\sqrt{2\pi n}$ random hash functions in expectation.
- Need to store seed of $\log_2\left(\frac{n^n}{n!}\right) \approx \log_2(e^n) = n\log_2 e \approx 1.44n$ bits.

- Problems?

Content

Construction Using Cuckoo Hashing and Retrieval

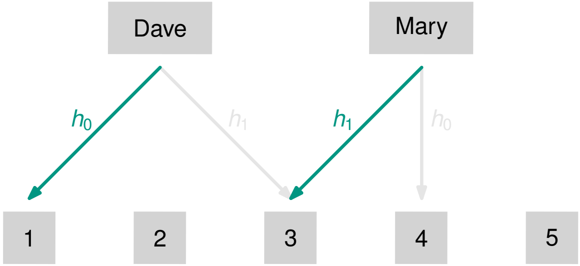
Cuckoo Hashing (abstract reminder)

Let $S \subseteq D$ of size $n = |S|$ and $h_1, \dots, h_k \sim \mathcal{U}([m]^D)$ where $\frac{n}{m} < c_k^*$ for some *threshold* c_k^*. *With high probability* there exists $\sigma(x) \in [k]$ for each $x \in S$ such that $x \mapsto h_{\sigma(x)}(x)$ is injective on S.

Perfect Hash Function from Retrieval

- Store $\sigma : S \to [k]$ in retrieval data structure R
- (non-minimal) PHF $P = (R, h_1, \dots, h_k)$ with

$$\mathbf{eval}_P(x) := h_{\mathbf{eval}_R(x)}(x).$$



1-bit retrieval:

Dave	0
Mary	1

PHF(Mary) = h_1(Mary) = 3

Construction Using Cuckoo Hashing and Retrieval

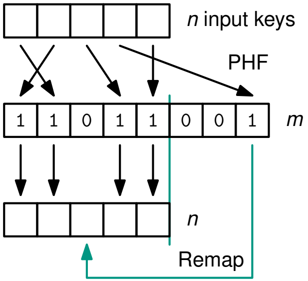
Space Consumption

Example with $k = 4$

- need $\frac{n}{m} < c_4^* \approx 0.9768 \rightsquigarrow \varepsilon \approx 0.0238$
- space needed for P is the space for R:
  $\approx n\lceil\log_2(k)\rceil = 2n$ bits using Bumped Ribbon Retrieval
  (see previous lecture)

"Repairing" a PHF to get MPHF

- *Remap* values $> n$ into holes left by previous keys
  (using Elias-Fano coding, not here)
- For $\varepsilon \approx 0.0238$, this needs $0.17n$ bits



ShockHash

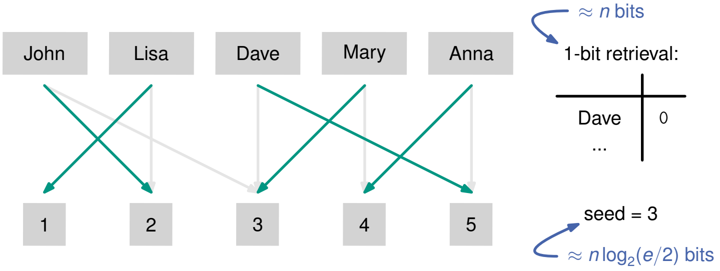
Exercise: What could go wrong?

Let's just avoid having to repair by using $m = n$ ($\varepsilon = 0$) and use 2 hash functions to save space.

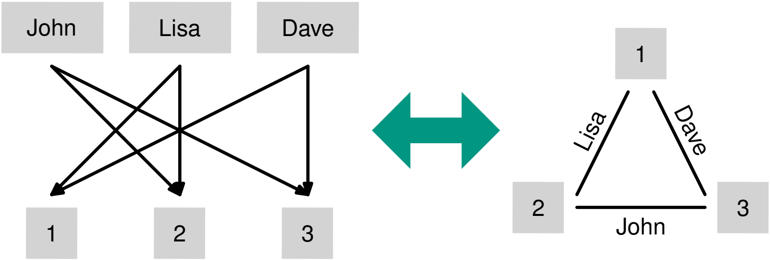
Solution: $\frac{n}{m} = 1 \gg c_2^* = \frac{1}{2}$, so there likely is no placement.

ShockHash Idea

- Retry different seeds until it is orientable
- Need to try $\approx (e/2)^n$ seeds, space close to optimal
- $\approx 2^n$ times faster than brute-force



ShockHash
Interpretations of the cuckoo graph



ShockHash

Proof: Probability that we can orient the ShockHash graph

- Labeled trees (Cayley's formula)
- Table rows can be in any order
- Hash values can be in any order
- Last edge can be anything

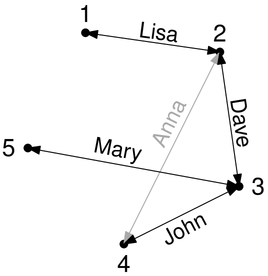
$$\mathbb{P} \geq \frac{n^{n-2} \ (n-1)! \ 2^{n-1} \ \ n^2}{n^{2n}}$$

$$= \frac{n!}{n^n} \cdot \frac{2^{n-1}}{n}$$

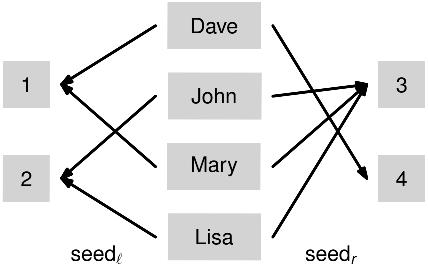
Brute force

Almost 2^n times higher probability

x	$h_0(x)$	$h_1(x)$
John	3	4
Lisa	2	1
Dave	2	3
Mary	5	3
Anna	?	?

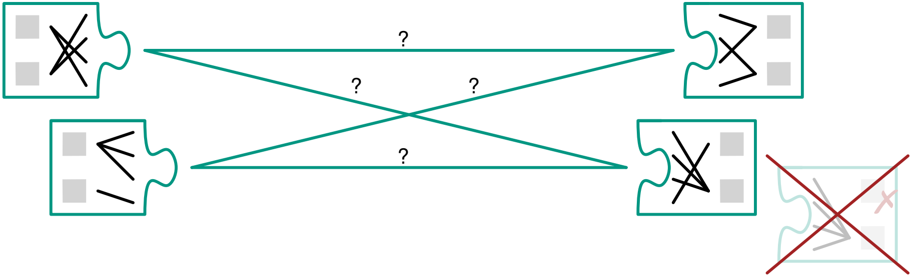


Bipartite ShockHash



- Partition output values
- Store two seeds and retrieval data structure

Bipartite ShockHash



- Build pool of $\sqrt{(e/2)^n} \approx 1.165^n$ seeds
- Filter seeds before combining, accuracy $0.836^{n/2}$ (not here)
- Only $\left(\sqrt{(e/2)^n} \cdot 0.836^{n/2}\right)^2 \approx 1.136^n$ combinations to test
  $\Rightarrow$ lower order term

Heads up

It is hard to show that reusing seeds from the pool does not hurt the success probability too much.

Content

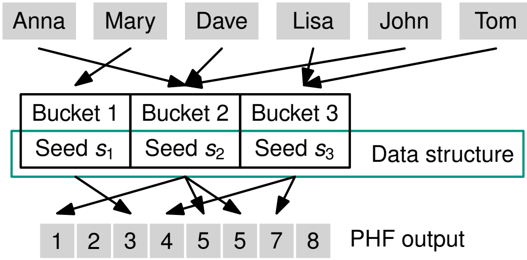
Construction Using Bucket Placement

Karlsruhe Institute of Technology

Perfect Hash Function
$P = (k, h, (g_i)_{i \in \mathbb{N}}, (s_1, \dots, s_k))$

- k is a number of *buckets*
- $h \sim \mathcal{U}([k]^D)$ assigns random bucket to each key
- $g_s \sim \mathcal{U}([m]^D)$ for each *seed* $s \in \mathbb{N}$
- s_i is the seed used by keys in bucket i

- $\mathbf{eval}_P(x) := g_{s_{h(x)}}(x)$
- $s_1, \dots, s_k$ are found using trial and error
- *huge design space*



Construction Using Bucket Placement

Design Space: Finding Seeds

Karlsruhe Institute of Technology

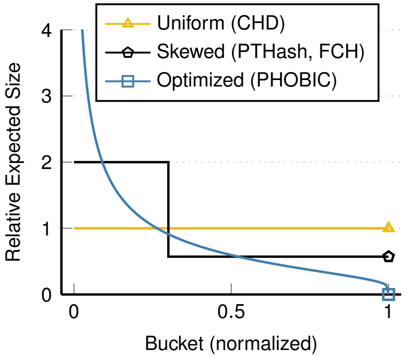
Problem: First buckets are easier to place into almost empty output domain. Last buckets take a long time.

Improvement 1

Sort buckets by their actual size and place largest buckets first.

Improvement 2

Bias bucket assignment function *h* such that it makes early buckets larger.



Content

1. (Minimal-) Perfect Hashing

2. Conclusion

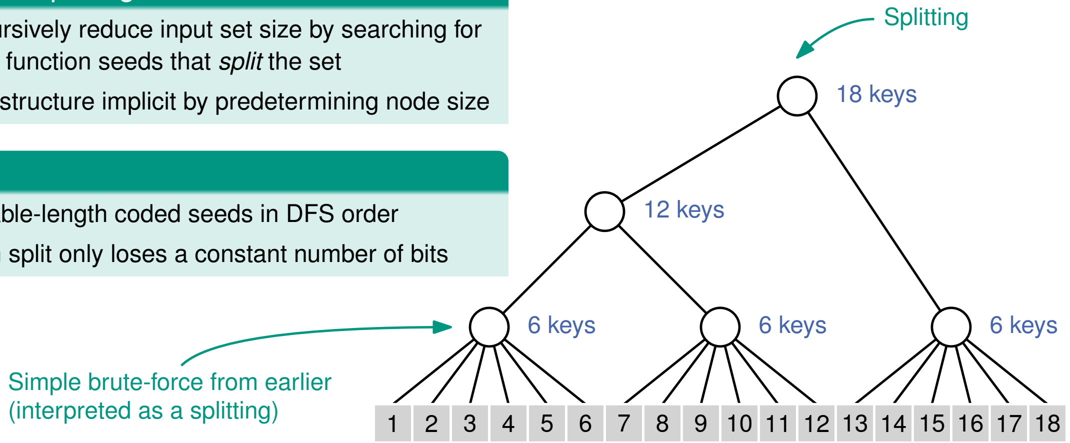
Construction Using Recursive Splitting

Recursive Splitting

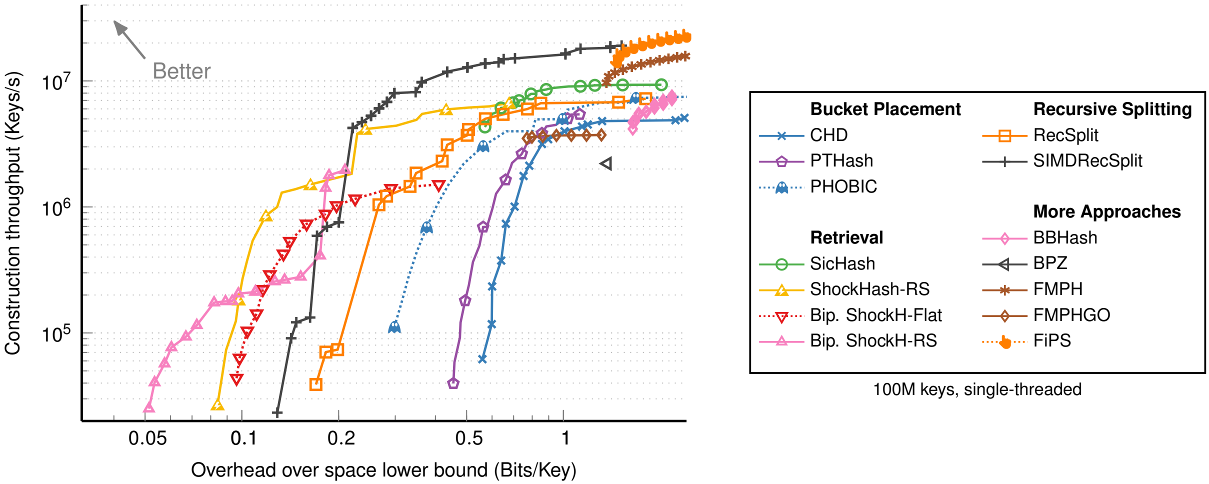
- Recursively reduce input set size by searching for hash function seeds that *split* the set
- Tree structure implicit by predetermining node size

Storage

- Variable-length coded seeds in DFS order
- Each split only loses a constant number of bits



Practical Comparison



100M keys, single-threaded

Perfect Hashing Variants

k-Perfect Hashing

- Up to k collisions on each output value are allowed
- Application: Find external memory page

Monotone Minimal Perfect Hashing

- Keep natural order of input keys (Rank data structure)
- Application: Databases

Conclusion

Karlsruhe Institute of Technology

Definition

(M)PHF for $S \subseteq U$ realises injective function on S, without storing S.

Perfect Hashing Through Trial and Error

Test seeds until one gives an MPHF.

Perfect Hashing Through Retrieval

Store one of multiple choices for each key.
Cuckoo Hashing + Retrieval $\rightarrow$ Perfect Hashing
$\rightarrow$ Updatable Retrieval (“hash table without keys”)

Perfect Hashing Through Bucket Placement

Hash keys to buckets. Greedily store seed for each bucket such that its keys do not collide with earlier keys.

Perfect Hashing Through Recursive Splitting

Recursively split set of keys until the set is small enough for trial and error.

Anhang: Mögliche Prüfungsfragen I

- Was zeichnet eine gute Perfekte Hashfunktion aus?
- Was sind upper und lower bounds an den Platzverbrauch?
- Wir haben Hashtabellen ohne Schlüssel kennengelernt. Was hat es damit auf sich?
- Wie kann man perfekte Hashfunktionen mit (Trial und Error | Retrieval | Bucket placement | Recursive splitting) konstruieren?