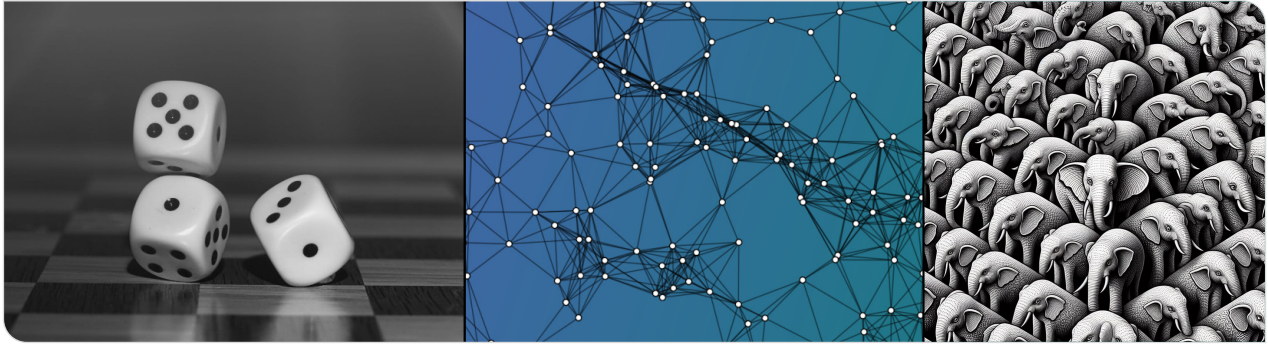
Karlsruhe Institute of Technology

Algorithm Engineering

Probability and Computing – Perfect Hashing

Stefan Walzer | WS 2025/2026



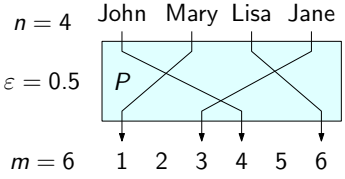
Content

Perfect Hash Function (PHF) and Minimal Perfect Hash Function (MPHF)

The PHF 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: $x \mapsto$ **eval**$_P(x)$ is injective on S



Perfect Hash Function (PHF) and Minimal Perfect Hash Function (MPHF)

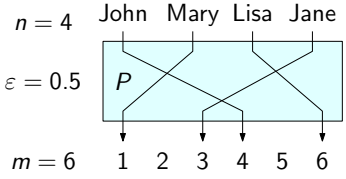
The PHF data type, for universe D, $\varepsilon \geq 0$

construct(S):

input:	$S \subseteq D$ of size $n = \lvert S \rvert$
output:	data structure P.

eval$_P(x)$:

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



Goals

- ε small // $\varepsilon = 0$: *Minimal* perfect hash function (MPHF)
- space requirement of P is $\mathcal{O}(n)$ bits
  - $\approx 1.44n$ bits is necessary and sufficient for $\varepsilon = 0$
  - note: storing S might need $\Omega(n \log(\lvert D \rvert))$ bits.
- ideally: running time of **eval** is $\mathcal{O}(1)$
- ideally: running time of **construct** is $\mathcal{O}(n)$

Perfect Hash Function (PHF) and Minimal Perfect Hash Function (MPHF)

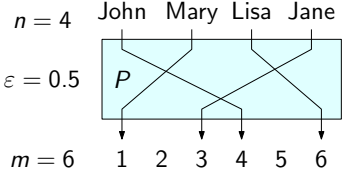
The PHF data type, for universe D, $\varepsilon \geq 0$

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

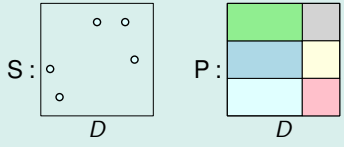
eval$_P(x)$:
- input: $x \in D$
- output: a number in $[m]$ where $m = (1+\varepsilon)n$
- requirement: $x \mapsto$ **eval**$_P(x)$ is injective on S

Goals

- ε small // $\varepsilon = 0$: *Minimal* perfect hash function (MPHF)
- space requirement of P is $\mathcal{O}(n)$ bits
  - $\approx 1.44n$ bits is necessary and sufficient for $\varepsilon = 0$
  - note: storing S might need $\Omega(n \log(|D|))$ bits.
- ideally: running time of **eval** is $\mathcal{O}(1)$
- ideally: running time of **construct** is $\mathcal{O}(n)$



Intuition



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

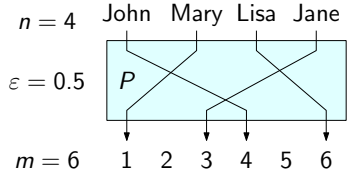
Motivation for (Minimal-) Perfect Hashing

Short IDs

Replace keys with short unique identifies

$$\textbf{eval}_P(\text{“}\texttt{creativeUserName@example.org}\text{”}) = 10241.$$

$$\textbf{eval}_P(\text{“}\texttt{ACGGGTCAGTA}\text{”}) = 563. \quad \text{// } k\text{-mers in bioinformatics}$$



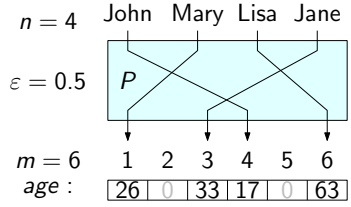
Motivation for (Minimal-) Perfect Hashing

Short IDs

Replace keys with short unique identifies

$$\mathbf{eval}_P(\texttt{“creativeUserName@example.org”}) = 10241.$$

$$\mathbf{eval}_P(\texttt{“ACGGGTCAGTA”}) = 563.$$ // k-mers in bioinformatics



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 $\mathbf{eval}_P(x)$.
  $\hookrightarrow$ 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

Content

Space Lower Bound & Brute Force Construction

Exercise: $\approx n \log_2 e$ bits are sufficient

For any $S \subseteq D$ of size n we have $\Pr_{h \sim \mathcal{U}([n]^D)}[h \text{ is injective on } S] = \frac{n!}{n^n}$.

$\hookrightarrow$ success after trying $\approx \frac{n^n}{n!}$ random hash functions

$\hookrightarrow$ requires storing seed of $\log_2\left(\frac{n^n}{n!}\right) \approx \log_2(e^n) \approx 1.44n$ bits.

Exercise: $\approx n \log_2 e$ bits are necessary

- Show that $\approx \frac{n^n}{n!}$ distinct MPHF data structures are needed to handle all possible inputs.
- Conclude that $\approx \log_2 \frac{n^n}{n!} \approx n \log_2 e$ bits are needed for worst-case inputs.

Plan for Today

- lots of ideas and intuition
- few details

Content

Content

Elias-Fano from the “Advanced Data Structure” Lecture

Theorem: Elias-Fano Encoding

An Elias-Fano data structure encodes a sequence $0 \leq a_1 \leq \ldots \leq a_n \leq u$ with

- access time $\mathcal{O}(1)$ // input $i \in [n]$, output $a_i \in [u]$
- space $n(2 + \lceil \log_2 \frac{u}{n} \rceil)$ // Intuition: mean *distance* between a_i and a_{i+1} is $\frac{u}{n}$; hence $\approx \log \frac{u}{n}$ bits per element

Idea (ignoring divisibility issues)

- Partition $[u]$ into n buckets of size $\frac{u}{n}$.
- Unary encode bucket sizes $\rightsquigarrow 2n$ bits
- keep lower $\log_2 \frac{u}{n}$ bits of entries in array
- fast access using select data structure

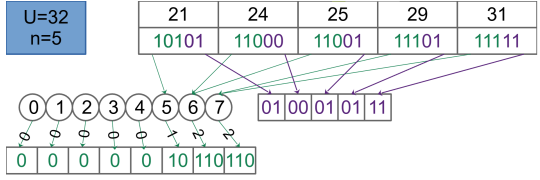


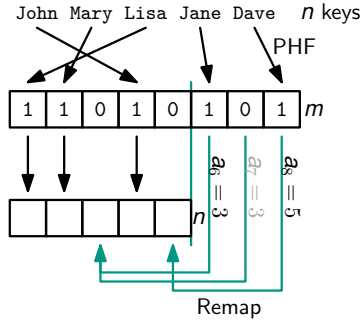
illustration by Wikipedia user Trobolt
https://commons.wikimedia.org/wiki/File:Elias-fano_1.svg

Content

“Repairing” a Non-Minimal PHF to make it Minimal

Idea

Remap keys in the $m - n$ extra slots into the gaps left within the first n slots.



“Repairing” a Non-Minimal PHF to make it Minimal

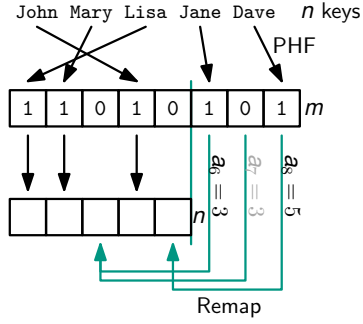
Idea

Remap keys in the $m - n$ extra slots into the gaps left within the first n slots.

Additional data structure

- Let $1 \leq a_{n+1} \leq \ldots \leq a_m \leq n$ be such that a used slot $i \in \{n+1, \ldots, m\}$ is assigned an empty slot $a_i \in [n]$, without collisions.
- Store $a_{n+1} \leq \ldots \leq a_m$ using Elias-Fano

$$\text{space} = (m-n)(2 + \lceil \log_2 \tfrac{n}{m-n} \rceil) = \varepsilon n \cdot (2 + \lceil \log_2 \tfrac{1}{\varepsilon} \rceil)$$



Content

From Exponential to Linear Time using Partioning

Karlsruhe Institute of Technology

Lemma

Assume an MPHF has

- construction time $\mathcal{O}(e^{c_1 n})$
- space $c_2 n$ bits // optimal if $c_2 = \log_2 e$

Then for $\lambda > 0$ there exists MPHF with

- expected construction time $\mathcal{O}(n \frac{e^{\lambda(e^{c_1}-1)}}{\lambda})$ // "$\mathcal{O}(n)$"
- space $c_2 n + \mathcal{O}(n \frac{\log \lambda}{\lambda})$ // "$(c_2 + \varepsilon)n$" for any small ε

From Exponential to Linear Time using Partioning

Lemma

Assume an MPHF has

- construction time $\mathcal{O}(e^{c_1 n})$
- space $c_2 n$ bits // optimal if $c_2 = \log_2 e$

Then for $\lambda > 0$ there exists MPHF with

- expected construction time $\mathcal{O}(n \frac{e^{\lambda(e^{c_1}-1)}}{\lambda})$ // "$\mathcal{O}(n)$"
- space $c_2 n + \mathcal{O}(n \frac{\log \lambda}{\lambda})$ // "$(c_2 + \varepsilon)n$" for any small ε

Idea: Partitioned MPHF

- Partition input set $S \subseteq D$ into $S_1, \ldots, S_k$ using hash function $h \sim \mathcal{U}([k]^D)$
- overall MPHF given by
  - MPHFs P_i on S_i for $i \in [k]$
  - prefix sums $\sigma_i = |S_1| + \cdots + |S_{i-1}|$ for $i \in [k]$
    $\hookrightarrow$ $\mathbf{eval}_P(x) = \mathbf{eval}_{P_{h(x)}}(x) + \sigma_{h(x)}$

From Exponential to Linear Time using Partioning

Lemma

Assume an MPHF has

- construction time $\mathcal{O}(e^{c_1 n})$
- space $c_2 n$ bits // optimal if $c_2 = \log_2 e$

Then for $\lambda > 0$ there exists MPHF with

- expected construction time $\mathcal{O}(n \frac{e^{\lambda(e^{c_1}-1)}}{\lambda})$ // "$\mathcal{O}(n)$"
- space $c_2 n + \mathcal{O}(n \frac{\log \lambda}{\lambda})$ // "$(c_2 + \varepsilon)n$" for any small ε

Idea: Partitioned MPHF

- Partition input set $S \subseteq D$ into $S_1, \ldots, S_k$ using hash function $h \sim \mathcal{U}([k]^D)$
- overall MPHF given by
  - MPHFs P_i on S_i for $i \in [k]$
  - prefix sums $\sigma_i = |S_1| + \cdots + |S_{i-1}|$ for $i \in [k]$
    $\hookrightarrow$ $\mathbf{eval}_P(x) = \mathbf{eval}_{P_{h(x)}}(x) + \sigma_{h(x)}$

Analysis sketch for $k = \frac{n}{\lambda}$

- space for storing $\sigma_1, \ldots, \sigma_k$:
  - $\frac{n}{\lambda}(2 + \lceil \log \lambda \rceil)$ with Elias-Fano
- expected construction time for part 1:
  - $|S_1| \sim \mathrm{Bin}(n, \frac{\lambda}{n}) \approx \mathrm{Pois}(\lambda)$
  - $\mathbb{E}[e^{c_1 |S_1|}] \approx \sum_{i \geq 0} e^{-\lambda} \frac{\lambda^i}{i!} \cdot e^{c_1 i} =$
    $e^{-\lambda} \sum_{i \geq 0} \frac{(\lambda e^{c_1})^i}{i!} = e^{-\lambda} e^{\lambda e^{c_1}} = e^{\lambda(e^{c_1}-1)}.$

Remark: Perfect Partitioning

Variant: k-perfect hash function (k-PHF)

- similar to perfect hash function
- up to k keys mapped to the same value
- *minimal*: $m = \lceil \frac{n}{k} \rceil$

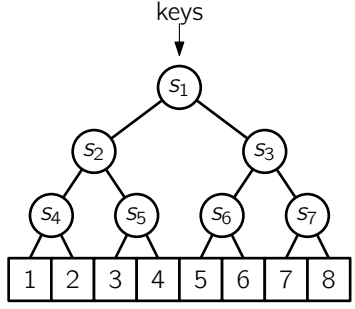
Useful in practice but not yet well understood. Active research @ITI Sanders.

Content

MPHF via Recursive Splitting

Idea (for $n = 2^d$)

Nodes in binary tree store seeds $s_1, \ldots, s_{n-1}$ of a hash function that split incoming keys perfectly in half.



MPHF via Recursive Splitting

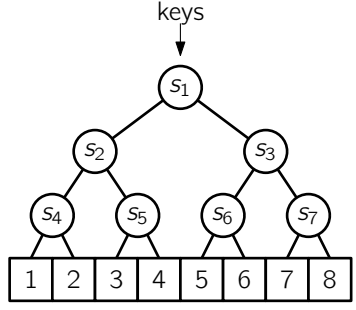
Idea (for $n = 2^d$)

Nodes in binary tree store seeds $s_1, \ldots, s_{n-1}$ of a hash function that split incoming keys perfectly in half.

Expected construction time $\mathcal{O}(n^{3/2})$

- splitting n keys in half:
  - work $\mathcal{O}(n)$ for trying a seed
  - success probability $\binom{n}{n/2} 2^{-n} = \Theta(1/\sqrt{n})$

  $\hookrightarrow$ expected work $\mathcal{O}(n^{3/2})$
- work for entire tree:
  $\mathcal{O}(n^{3/2} + 2 \cdot (\frac{n}{2})^{3/2} + 4 \cdot (\frac{n}{4})^{3/2} + \ldots) = \mathcal{O}(n^{3/2})$.



MPHF via Recursive Splitting

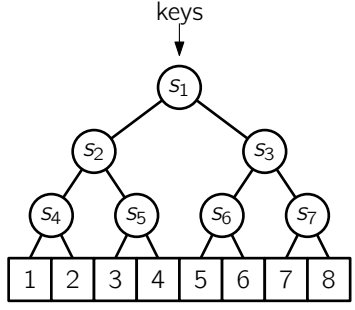
Idea (for $n = 2^d$)

Nodes in binary tree store seeds $s_1, \ldots, s_{n-1}$ of a hash function that split incoming keys perfectly in half.

Expected construction time $\mathcal{O}(n^{3/2})$

- splitting n keys in half:
  - work $\mathcal{O}(n)$ for trying a seed
  - success probability $\binom{n}{n/2} 2^{-n} = \Theta(1/\sqrt{n})$

  $\hookrightarrow$ expected work $\mathcal{O}(n^{3/2})$
- work for entire tree:
  $\mathcal{O}(n^{3/2} + 2 \cdot (\frac{n}{2})^{3/2} + 4 \cdot (\frac{n}{4})^{3/2} + \ldots) = \mathcal{O}(n^{3/2})$.



Issues (not addressed here)

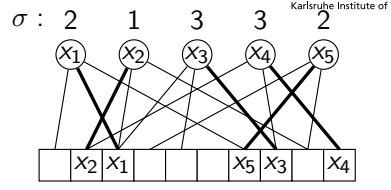
- query time $\Omega(\log n)$
- how to encode seeds compactly?
- what if n is not a power of 2?

Content

PHF via Cuckoo Hashing + 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.

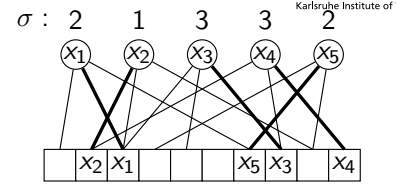


(picture assumes $h_1(x) < h_2(x) < h_3(x)$)

PHF via Cuckoo Hashing + Retrieval

Cuckoo Hashing (abstract reminder)

Let $S \subseteq D$ of size $n = |S|$ and $h_1, \ldots, h_k \sim \mathcal{U}([m]^D)$ 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.



(picture assumes $h_1(x) < h_2(x) < h_3(x)$)

Perfect Hash Function from Retrieval

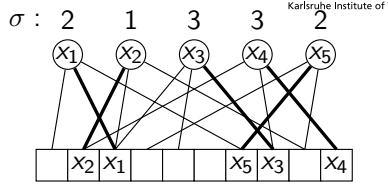
- Store $\sigma : S \to [k]$ as retrieval data structure R
- (non-minimal) PHF $P = (R, h_1, \ldots, h_k)$ with

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

PHF via Cuckoo Hashing + Retrieval

Cuckoo Hashing (abstract reminder)

Let $S \subseteq D$ of size $n = |S|$ and $h_1, \ldots, h_k \sim \mathcal{U}([m]^D)$ 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.



(picture assumes $h_1(x) < h_2(x) < h_3(x)$)

Perfect Hash Function from Retrieval

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

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

Example with $k = 3$

- use $\frac{n}{m} < c_3^{\Delta} \approx 0.81 \rightsquigarrow \varepsilon \approx 0.23$
  $\hookrightarrow$ placement found using peeling
- space needed for P is the space for R:
  $\approx 1.23n\lceil \log_2(k) \rceil = 2.26n$ bits
  with peeling-based retrieval from last time
- can use the same hashes and peeling twice!

Small Heavily Overloaded Cuckoo Table (ShockHash)

Cuckoo Hashing + Retrieval with $k = 2$ and $n = m$

- only $\approx n$ bits in space optimal retrieval data structure
- success probability $\approx (2/e)^n$ // not obvious
  $\hookrightarrow$ need seed of expected size $\approx \log((e/2)^n)$
- total space $\approx n \log_2(e/2) + n = n \log_2(e) = \text{OPT}$.

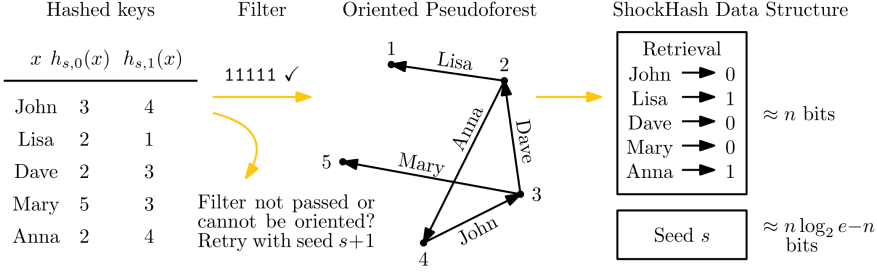


illustration by Hans-Peter Lehmann. "Filter" checks if all n positions are hit at least once to more quickly reject seeds

Small Heavily Overloaded Cuckoo Table (ShockHash)

Cuckoo Hashing + Retrieval with $k = 2$ and $n = m$

- only $\approx n$ bits in space optimal retrieval data structure
- success probability $\approx (2/e)^n$ // not obvious
  $\hookrightarrow$ need seed of expected size $\approx \log((e/2)^n)$
- total space $\approx n \log_2(e/2) + n = n \log_2(e) = \text{OPT}$.

Running Time

- $n(e/2)^n = \mathcal{O}(1.36^n)$
- can be improved to $\mathcal{O}(1.17^n)$

recall: naive brute force is $\mathcal{O}(e^n) = \mathcal{O}(2.72^n)$

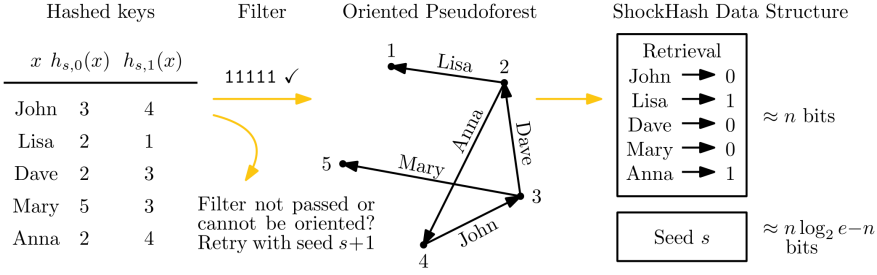
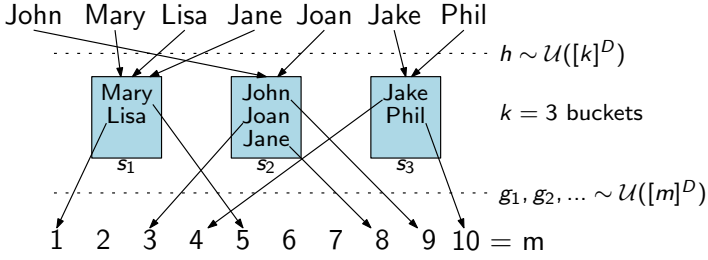


illustration by Hans-Peter Lehmann. "Filter" checks if all n positions are hit at least once to more quickly reject seeds

Content

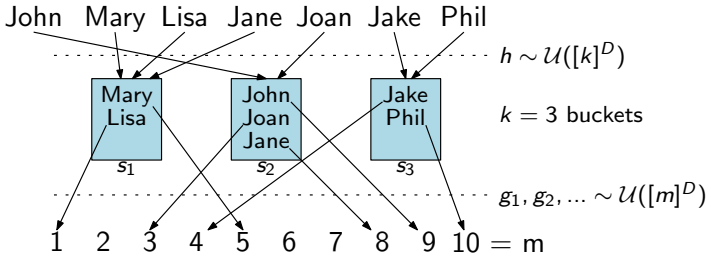
(M)PHF via Bucket Placement



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

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

(M)PHF via Bucket Placement



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

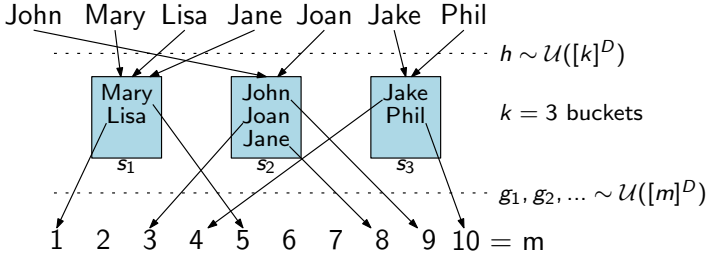
- $\textbf{eval}_P(x) := g_{s_{h(x)}}(x)$
- $s_1, \ldots, s_k$ are found one by one using trial and error
- *huge design space*

Observation 1

Placing buckets in order of decreasing size reduces construction time.

// in the example: choose s_2 first.

(M)PHF via Bucket Placement



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

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

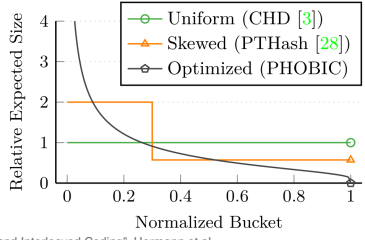
Observation 1

Placing buckets in order of decreasing size reduces construction time.

// in the example: choose s_2 first.

Observation 2

Deliberately non-uniform partitioning reduces construction time.

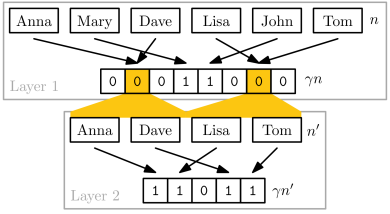


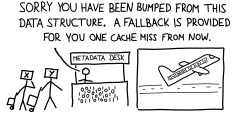
Expected bucket size functions from "PHOBIC: Perfect Hashing with Optimized Bucket Sizes and Interleaved Coding", Hermann et al.

Content

PHF via Fingerprinting and Bumping

Illustration taken from Hans-Peter Lehmann's dissertation.



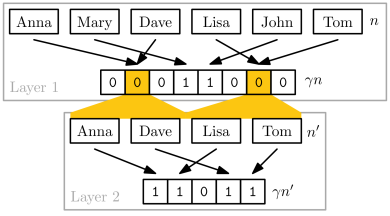


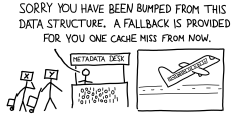
$P(x)$	1	2	3	4	5	6
x	Mary	John	Anna	Lisa	Dave	Tom

- use fingerprint function $f : D \to [\gamma n]$
- bitvector $B[1..\gamma n]$ indicates unique fingerprints
- keys with colliding fingerprints are *bumped* to next layer
- rank data structure maps fingerprint to its rank in layer

PHF via Fingerprinting and Bumping

Illustration taken from Hans-Peter Lehmann's dissertation.





$P(x)$	1	2	3	4	5	6
x	Mary	John	Anna	Lisa	Dave	Tom

- use fingerprint function $f : D \to [\gamma n]$
- bitvector $B[1..\gamma n]$ indicates unique fingerprints
- keys with colliding fingerprints are *bumped* to next layer
- rank data structure maps fingerprint to its rank in layer

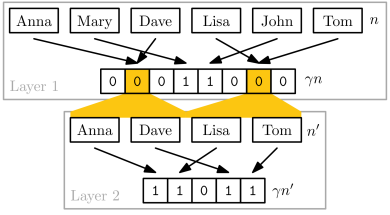
Lemma

i The expected space requirement of this approach is $\gamma e^{1/\gamma} + o(1)$ bits per key.

ii Space is minimised for $\gamma = 1$ with $\approx e$ bits per key.

PHF via Fingerprinting and Bumping

Illustration taken from Hans-Peter Lehmann's dissertation.



$P(x)$	1	2	3	4	5	6
x	Mary	John	Anna	Lisa	Dave	Tom

- use fingerprint function $f : D \to [\gamma n]$
- bitvector $B[1..\gamma n]$ indicates unique fingerprints
- keys with colliding fingerprints are *bumped* to next layer
- rank data structure maps fingerprint to its rank in layer

Lemma

i The expected space requirement of this approach is $\gamma e^{1/\gamma} + o(1)$ bits per key.

ii Space is minimised for $\gamma = 1$ with $\approx e$ bits per key.

Proof sketch of (i).

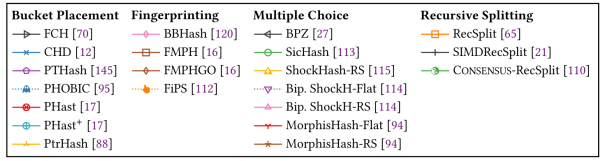
A key has a unique fingerprint in the first layer with probability

$$\Pr_{X \sim \mathrm{Bin}(n-1, \frac{1}{\gamma n})}[X = 0] \approx \Pr_{X \sim \mathrm{Pois}(1/\gamma)}[X = 0] = e^{-1/\gamma}.$$

In expectation $n \cdot e^{-1/\gamma}$ keys handled with γn bits $\rightsquigarrow \gamma e^{1/\gamma}$ bits per key.

Content

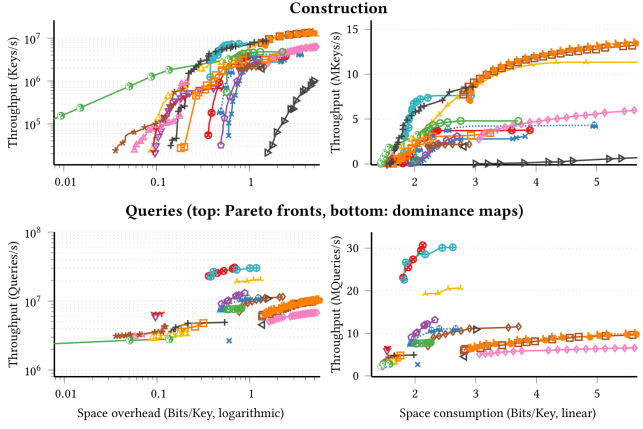
What's Best in Practice?



- See "Modern Minimal Perfect Hashing: A Survey" by Hans-Peter Lehmann et al.
- Active research @ITI Sanders.

Oversimplified takeaways

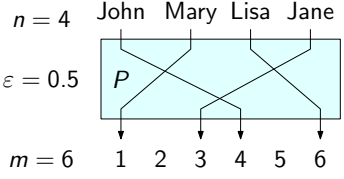
- least space: recursive splitting
- fastest queries: bucket placement
- fastest construction: fingerprinting



Summary

Perfect Hash Function (PHF) for $S \subseteq D$

- maps D to range $[m]$
- "perfect" on S, i.e. no collisions on S
- *minimal* perfect (MPHF) if $m = n := |S|$
  - $n \log_2 e$ bits necessary and sufficient for MPHF



Motivation

- map long keys to short unique IDs
- updatable retrieval // "hash table without keys"

Techniques

- Partitioning
- Recursive Splitting
- Fingerprinting & Bumping
- Bucket Placement
- Cuckoo + Retrieval

Appendix: Possible Exam Questions I

- What is a (minimum) perfect hash function?
  - What makes for a *good* (M)PHF?
- In what sense does a PHF realise a “hash table without keys”?
- How much space does an MPHF require at least?
  - How can $\approx n \log_2 e$ bits achieved?
  - What is the idea for the proof that $\approx n \log_2 e$ bits are necessary?
- Techniques for constructing (M)PHFs:
  - Why can MPHF constructions with exponential running time still be useful?
  - What is the idea of recursive splitting? What running time was achieved?
  - How did we use cuckoo hashing and retrieval to construct a PHF?
    - Offer a concrete construction. What ε and what space requirement do you get?
  - How does bucket placement work?
    - What refinement regarding partitioning into buckets did we hint at?
  - How does fingerprinting with bumping work?
    - What is the bumping probability?
    - What is the space requirement?