Advanced Data Structures

Lecture 11: Minimal Perfect Hashing

Florian Kurpicz

PINGO

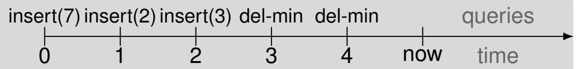
https://pingo.scc.kit.edu/489786

Recap: Retroactive Data Structures

Operations

- $\texttt{INSERT}(t, \textit{operation})$: insert operation at time t
- $\texttt{DELETE}(t)$: delete operation at time t
- $\texttt{QUERY}(t, \textit{query})$: ask *query* at time t

- for a priority queue updates are
  - insert
  - delete-min
- time is integer ℹ for simplicity otherwise use order-maintenance data structure

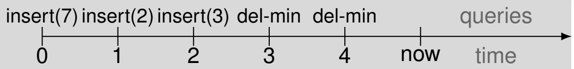


Recap: Retroactive Data Structures

Operations

- $\texttt{INSERT}(t, \textit{operation})$: insert operation at time t
- $\texttt{DELETE}(t)$: delete operation at time t
- $\texttt{QUERY}(t, \textit{query})$: ask *query* at time t

- for a priority queue updates are
  - insert
  - delete-min
- time is integer ⓘ for simplicity otherwise use order-maintenance data structure



Definition: Partial Retroactivity

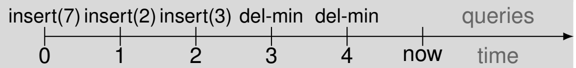
QUERY is only allowed for $t = \infty$ ⓘ now

Recap: Retroactive Data Structures

Operations

- $\texttt{INSERT}(t, \mathit{operation})$: insert operation at time t
- $\texttt{DELETE}(t)$: delete operation at time t
- $\texttt{QUERY}(t, \mathit{query})$: ask *query* at time t

- for a priority queue updates are
  - insert
  - delete-min
- time is integer ℹ for simplicity otherwise use order-maintenance data structure



Definition: Partial Retroactivity

$\texttt{QUERY}$ is only allowed for $t = \infty$ ℹ now

Definition: Full Retroactivity

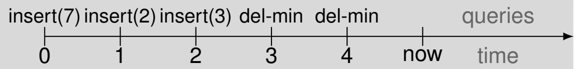
$\texttt{QUERY}$ is allowed at any time t

Recap: Retroactive Data Structures

Operations

- `INSERT`$(t, \mathit{operation})$: insert operation at time t
- `DELETE`(t): delete operation at time t
- `QUERY`(t, query): ask *query* at time t

- for a priority queue updates are
  - insert
  - delete-min
- time is integer ℹ for simplicity otherwise use order-maintenance data structure



Definition: Partial Retroactivity

`QUERY` is only allowed for $t = \infty$ ℹ now

Definition: Full Retroactivity

`QUERY` is allowed at any time t

Definition: Nonoblivious Retroactivity

`INSERT`, `DELETE`, and `QUERY` at any time t but also identify changed `QUERY` results

Hashing (1/2)

Karlsruhe Institute of Technology

- $h\colon \{0,\dots,u-1\} \to \{0,\dots,m-1\}$

- n objects
- from universe $U = \{0,\dots,u-1\}$
- hash table of size m ℹ m close to n
- $m \ll u$

Hashing (1/2)

- $h\colon \{0,\ldots,u-1\} \to \{0,\ldots,m-1\}$

- n objects
- from universe $U = \{0,\ldots,u-1\}$
- hash table of size m ℹ m close to n
- $m \ll u$

Definition: Totally Random

- $\mathbb{P}[h(x) = t] = 1/m$
- independent of $h(y)$ for all $x \neq y \in U$
- requires $\Theta(u \log m)$ bits of space to store ℹ too big

Hashing (1/2)

- $h\colon \{0, \ldots, u-1\} \to \{0, \ldots, m-1\}$

- n objects
- from universe $U = \{0, \ldots, u-1\}$
- hash table of size m ℹ m close to n
- $m \ll u$

Definition: Totally Random

- $\mathbb{P}[h(x) = t] = 1/m$
- independent of $h(y)$ for all $x \neq y \in U$
- requires $\Theta(u \log m)$ bits of space to store ℹ too big

Definition: Universal

- choose h from family H with $\mathbb{P}_{h \in H}[h(x) = h(y)] = O(1/m)$ for all $x \neq y \in U$
- family is small to enable efficient encoding

Hashing (1/2)

- $h \colon \{0, \ldots, u-1\} \to \{0, \ldots, m-1\}$

- n objects
- from universe $U = \{0, \ldots, u-1\}$
- hash table of size m ℹ m close to n
- $m \ll u$

Definition: Totally Random

- $\mathbb{P}[h(x) = t] = 1/m$
- independent of $h(y)$ for all $x \neq y \in U$
- requires $\Theta(u \log m)$ bits of space to store ℹ too big

Definition: Universal

- choose h from family H with $\mathbb{P}_{h \in H}[h(x) = h(y)] = O(1/m)$ for all $x \neq y \in U$
- family is small to enable efficient encoding

- $h(x) = (ax \mod u) \mod m$ for $0 < a < p$ and p being prime $> u$
- $h(x) = ax$ » $(\log u - \log m)$ for m, u being powers of two

- Why is this family easier to store? **PINGO**

Hashing (2/2)

Definition: k-wise Independent

- choose h from family H with $\mathbb{P}[h(x_1) = t_1 \& \ldots \& h(x_k) = t_k] = O(1/m^k)$ for distinct $x_1, \ldots, x_k \in U$

Hashing (2/2)

Definition: k-wise Independent

- choose h from family H with $\mathbb{P}[h(x_1) = t_1 \& \ldots \& h(x_k) = t_k] = O(1/m^k)$ for distinct $x_1, \ldots, x_k \in U$

- implies universal

Hashing (2/2)

Karlsruhe Institute of Technology

Definition: k-wise Independent

- choose h from family H with $\mathbb{P}[h(x_1) = t_1 \& \dots \& h(x_k) = t_k] = O(1/m^k)$ for distinct $x_1, \dots, x_k \in U$

- implies universal

- $h(x) = ((\sum_{i=0}^{k-1} a_i x^i) \mod p) \mod m$ for $0 \leq a_i < p$ and $0 < a_{k-1} < p$

- pairwise ($k = 2$) independence is stronger than universal
- $h(x) = ((ax + b) \mod u) \mod m$

Hashing (2/2)

Definition: k-wise Independent

- choose h from family H with $\mathbb{P}[h(x_1) = t_1 \& \ldots \& h(x_k) = t_k] = O(1/m^k)$ for distinct $x_1, \ldots, x_k \in U$

- implies universal

- $h(x) = ((\sum_{i=0}^{k-1} a_i x^i) \mod p) \mod m$ for $0 \leq a_i < p$ and $0 < a_{k-1} < p$

- pairwise ($k = 2$) independence is stronger than universal
- $h(x) = ((ax + b) \mod u) \mod m$

Definition: Simple Tabulation Hashing

- view x as vector $x_1, \ldots, x_c$ of characters
- totally random hash table T_i for each character
- $h(x) = T_1(x_1) \texttt{ xor } \ldots \texttt{ xor } T_c(x_c)$

Hashing (2/2)

Karlsruhe Institute of Technology

Definition: k-wise Independent

- choose h from family H with $\mathbb{P}[h(x_1) = t_1 \& \ldots \& h(x_k) = t_k] = O(1/m^k)$ for distinct $x_1, \ldots, x_k \in U$

- implies universal

- $h(x) = ((\sum_{i=0}^{k-1} a_i x^i) \mod p) \mod m$ for $0 \leq a_i < p$ and $0 < a_{k-1} < p$

- pairwise ($k = 2$) independence is stronger than universal
- $h(x) = ((ax + b) \mod u) \mod m$

Definition: Simple Tabulation Hashing

- view x as vector $x_1, \ldots, x_c$ of characters
- totally random hash table T_i for each character
- $h(x) = T_1(x_1) \texttt{ xor } \ldots \texttt{ xor } T_c(x_c)$

- Why can we use totally random hash tables?
  PINGO

Hashing (2/2)

Karlsruhe Institute of Technology

Definition: k-wise Independent

- choose h from family H with $\mathbb{P}[h(x_1) = t_1 \& \dots \& h(x_k) = t_k] = O(1/m^k)$ for distinct $x_1, \dots, x_k \in U$

- implies universal

- $h(x) = ((\sum_{i=0}^{k-1} a_i x^i) \mod p) \mod m$ for $0 \leq a_i < p$ and $0 < a_{k-1} < p$

- pairwise ($k = 2$) independence is stronger than universal
- $h(x) = ((ax + b) \mod u) \mod m$

Definition: Simple Tabulation Hashing

- view x as vector $x_1, \dots, x_c$ of characters
- totally random hash table T_i for each character
- $h(x) = T_1(x_1) \texttt{ xor } \dots \texttt{ xor } T_c(x_c)$

- Why can we use totally random hash tables?
  PINGO

- $O(cu^{1/c})$ space
- $O(c)$ time to compute
- 3-wise independent

Minimal Perfect Hashing

Definition: Perfect Hash Function

- injective hash function
- maps n objects to m slots

- lower space bound for $m = (1 + \epsilon)n$ is

$$\log e - \epsilon \log \frac{1 + \epsilon}{\epsilon}$$

- for m close to n there are likely collisions

Minimal Perfect Hashing

Karlsruhe Institute of Technology

Definition: Perfect Hash Function

- injective hash function
- maps n objects to m slots

- lower space bound for $m = (1+\epsilon)n$ is

$$\log e - \epsilon \log \frac{1+\epsilon}{\epsilon}$$

- for m close to n there are likely collisions

Definition: Minimal Perfect Hash Function

- bijective hash function
- maps n objects to $m = n$ slots
- $h\colon N \to [0, n)$

- lower space bound as for PHF with $\epsilon = 0$:

$$\log e \approx 1.44$$

- no collisions

Minimal Perfect Hashing

Karlsruhe Institute of Technology

Definition: Perfect Hash Function

- injective hash function
- maps n objects to m slots

- lower space bound for $m = (1+\epsilon)n$ is

$$\log e - \epsilon \log \frac{1+\epsilon}{\epsilon}$$

- for m close to n there are likely collisions

Definition: Minimal Perfect Hash Function

- bijective hash function
- maps n objects to $m = n$ slots
- $h\colon N \to [0, n)$

- lower space bound as for PHF with $\epsilon = 0$:

$$\log e \approx 1.44$$

- no collisions

- can we make PHF to MPHF? **PINGO**

BDZ (RAM) Algorithm [BPZ13]

- for each object calculate three *potential* slots (h_0, h_1, and h_2)
- for each slot that contains only one object, remove the object from all its other slots
- one slot per object
- if that does not work use other hash functions
- use rank data structure to map slots to $[0, n)$

- example on the board

- 1.95 bits per object when $m = 1.23n$

BDZ (RAM) Algorithm [BPZ13]

- for each object calculate three *potential* slots (h_0, h_1, and h_2)
- for each slot that contains only one object, remove the object from all its other slots
- one slot per object
- if that does not work use other hash functions
- use rank data structure to map slots to $[0, n)$

- example on the board

- 1.95 bits per object when $m = 1.23n$

- how to check if hash function works
- interpret each slot as node in a hypergraph
- objects are edges
- if graph is peelable, we have a feasible mapping

Definition: Peelable

A hypergraph is peelable, if it is possible to obtain a graph without edges by iteratively taking away edges that contain a node with degree 1

- example on the board

Compress, Hash, and Displace [BBD09a]

- partition keys into buckets
- set $m = (1 + \epsilon)n$ 1.01n
- sort partitions by size
- starting with largest bucket, find universal hash function mapping all keys to empty slots
- if key mapped to non-empty slot, try next hash function
- for each bucket store universal hash function
- use rank data structure to map slots to $[0, n)$

- example on the board

Compress, Hash, and Displace [BBD09a]

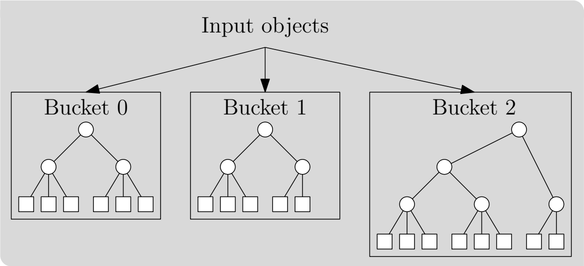
- partition keys into buckets
- set $m = (1 + \epsilon)n$ ⓘ $1.01n$
- sort partitions by size
- starting with largest bucket, find universal hash function mapping all keys to empty slots
- if key mapped to non-empty slot, try next hash function
- for each bucket store universal hash function
- use rank data structure to map slots to $[0, n)$

- example on the board

- can be used as PHF
- there are a lot of tricks w.r.t. bucket sizes and size distributions
- requires around 2.05 bits per object

RecSplit Overview [EGV20a]

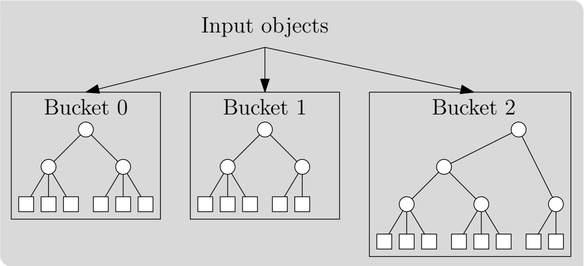
- partition keys into buckets of size b
- for each bucket compute splitting trees
- split keys into smaller sets
- stop when sets have size ℓ



RecSplit Overview [EGV20a]

- partition keys into buckets of size b
- for each bucket compute splitting trees
- split keys into smaller sets
- stop when sets have size ℓ

- upper aggregation levels have fanout 2
- lower two aggregation levels have fanout
  - $\max\{2, \lceil 0.35\ell + 0.55 \rceil\}$
  - $\max\{2, \lceil 0.21\ell + 0.9 \rceil\}$

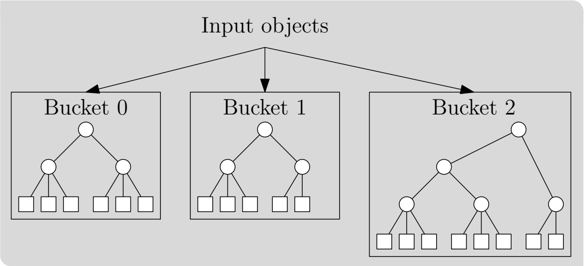


RecSplit Overview [EGV20a]

- partition keys into buckets of size b
- for each bucket compute splitting trees
- split keys into smaller sets
- stop when sets have size ℓ

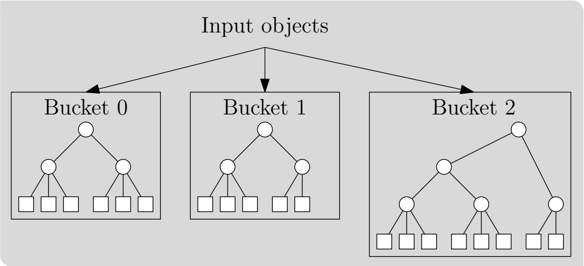
- upper aggregation levels have fanout 2
- lower two aggregation levels have fanout
  - $\max\{2, \lceil 0.35\ell + 0.55 \rceil\}$
  - $\max\{2, \lceil 0.21\ell + 0.9 \rceil\}$

- last level is leaf level
- find bijections



RecSplit Splitting Tree

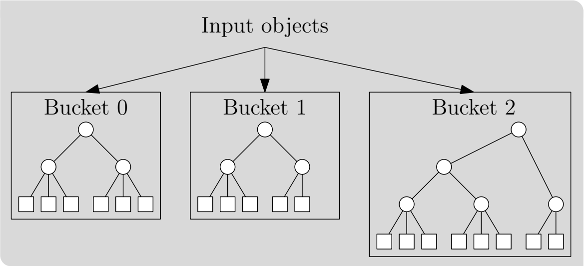
- tree structure is well defined
- store information for each node in preorder
- store hash function for each splitter
- encode function using Golomb-Rice



RecSplit Splitting Tree

- tree structure is well defined
- store information for each node in preorder
- store hash function for each splitter
- encode function using Golomb-Rice

- encodings of splitting trees stored in one bit vector
- use Elias-Fano to store
  - size of buckets
  - starting position of bucket in bit vector



Golomb Encoding [Gol66]

Definition: Golomb Code

Given an integer $x > 0$ and a constant $b > 0$, the Golomb code consists of

- $q = \lfloor \frac{x}{b} \rfloor$
- $r = x - qb = x \,\%\, b$
- $c = \lceil \lg b \rceil$

with

$$(x)_{\mathrm{Gol}(b)} = (q)_1 (r)_2$$

where $(r)_2$ depends on its size

- $r < 2^{\lfloor \lg b \rfloor - 1}$: r requires $\lfloor \lg b \rfloor$ bits and starts with a 0
- $r \geq 2^{\lfloor \lg b \rfloor - 1}$: r requires $\lceil \lg b \rceil$ bits and starts with a 1 and it encodes $r - 2^{\lfloor \lg b \rfloor - 1}$

Golomb Encoding [Gol66]

Definition: Golomb Code

Given an integer $x > 0$ and a constant $b > 0$, the Golomb code consists of

- $q = \lfloor \frac{x}{b} \rfloor$
- $r = x - qb = x \,\%\, b$
- $c = \lceil \lg b \rceil$

with

$$(x)_{\mathrm{Gol}(b)} = (q)_1 (r)_2$$

where $(r)_2$ depends on its size

- $r < 2^{\lfloor \lg b \rfloor - 1}$: r requires $\lfloor \lg b \rfloor$ bits and starts with a 0
- $r \geq 2^{\lfloor \lg b \rfloor - 1}$: r requires $\lceil \lg b \rceil$ bits and starts with a 1 and it encodes $r - 2^{\lfloor \lg b \rfloor - 1}$

- b has to be fixed for all codes
- still variable length

Golomb Encoding [Gol66]

Definition: Golomb Code

Given an integer $x > 0$ and a constant $b > 0$, the Golomb code consists of

- $q = \lfloor \frac{x}{b} \rfloor$
- $r = x - qb = x \,\%\, b$
- $c = \lceil \lg b \rceil$

with

$$(x)_{\mathrm{Gol}(b)} = (q)_1 (r)_2$$

where $(r)_2$ depends on its size

- $r < 2^{\lfloor \lg b \rfloor - 1}$: r requires $\lfloor \lg b \rfloor$ bits and starts with a 0
- $r \geq 2^{\lfloor \lg b \rfloor - 1}$: r requires $\lceil \lg b \rceil$ bits and starts with a 1 and it encodes $r - 2^{\lfloor \lg b \rfloor - 1}$

- b has to be fixed for all codes
- still variable length

- Golomb-Rice is special case where r is power of two

Golomb Encoding [Gol66]

Definition: Golomb Code

Given an integer $x > 0$ and a constant $b > 0$, the Golomb code consists of

- $q = \lfloor \frac{x}{b} \rfloor$
- $r = x - qb = x \,\%\, b$
- $c = \lceil \lg b \rceil$

with

$$(x)_{\mathrm{Gol}(b)} = (q)_1 (r)_2$$

where $(r)_2$ depends on its size

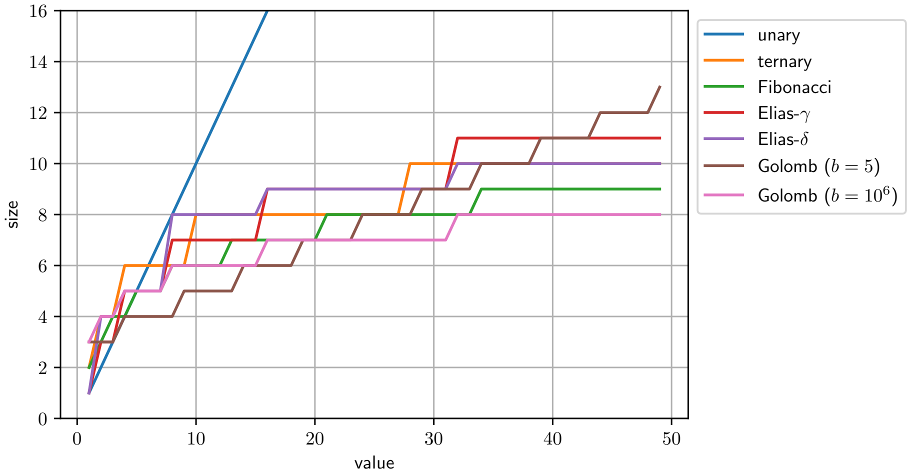
- $r < 2^{\lfloor \lg b \rfloor - 1}$: r requires $\lfloor \lg b \rfloor$ bits and starts with a 0
- $r \geq 2^{\lfloor \lg b \rfloor - 1}$: r requires $\lceil \lg b \rceil$ bits and starts with a 1 and it encodes $r - 2^{\lfloor \lg b \rfloor - 1}$

- b has to be fixed for all codes
- still variable length

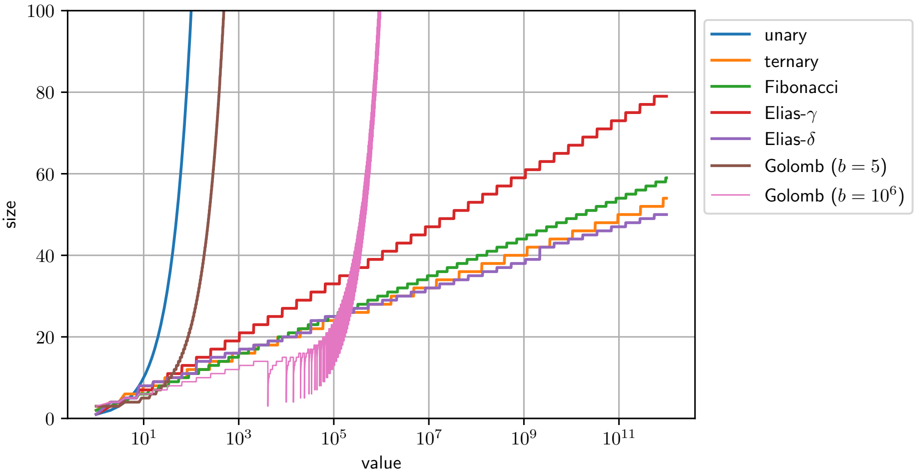
- Golomb-Rice is special case where r is power of two

- for $b = 5$, there are 4 remainders: $00, 01, 100, 101$, and 110
- $2^{\lfloor \lg 5 \rfloor - 1} = 2$
- $0, 1 < 2$: 00 and 01 require 2 bits
- $2, 3, 4 \geq 2$: require 3 bits and encode $0, 1, 2$ starting with 1

Comparison of Codes (1/2)

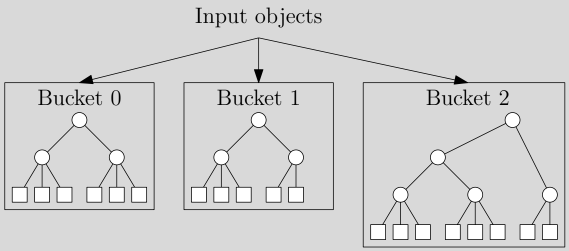


Comparison of Codes (2/2)



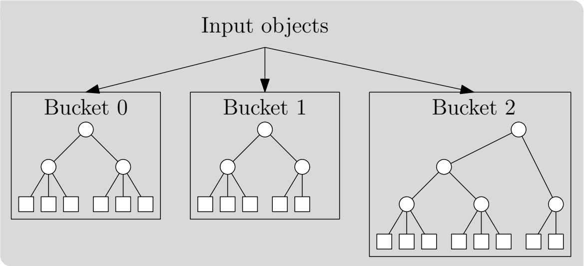
RecSplit Leaves

- find perfect hash function for keys in leaves
- test hash functions brute force
- use hash value modulo ℓ
- set bit in “bit vector” of length ℓ
- all bits set indicates bijection



RecSplit Queries

- find bucket
- follow splitting tree
- accumulate number of objects to the left
- use bijection in leaf
- result is sum of
  - objects in previous buckets
  - objects to the left in splitting tree
  - value of bijection



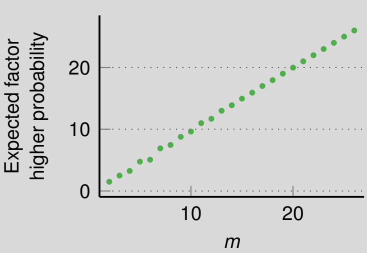
Parallel RecSplit

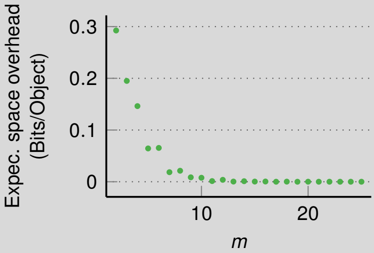
Karlsruhe Institute of Technology

- Dominik Bez, Florian Kurpicz, Hans-Peter Lehmann, and Peter Sanders. "High Performance Construction of RecSplit Based Minimal Perfect Hash Functions". In: *CoRR* abs/2212.09562 (2022)
- results to be presented at ESA'23
- based on a Domink Bez' Master's thesis

- randomly distribute objects in leaf in two sets A and B
- hash objects in both set
- two “bit vectors”: cyclic shift one until all bits are set when `OR`ed
- store hash function *and* rotation

- randomly distribute objects in leaf in two sets A and B
- hash objects in both set
- two “bit vectors”: cyclic shift one until all bits are set when ORed
- store hash function *and* rotation

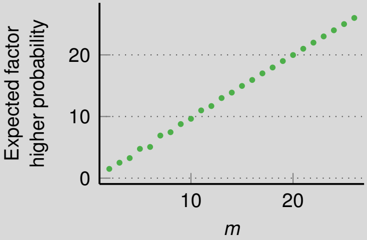


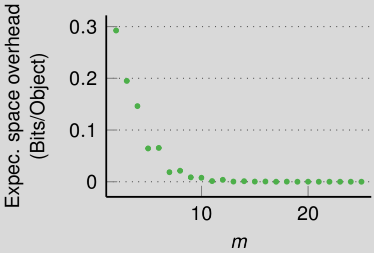


- randomly distribute objects in leaf in two sets A and B
- hash objects in both set
- two "bit vectors": cyclic shift one until all bits are set when `OR`ed
- store hash function *and* rotation

Lemma: Rotation Fitting

Let $|A| = \mathbb{A}$, $|B| = \mathbb{B}$, and $\mathbb{P}(R)$ be the probability of finding a bijection using rotation fitting. Let $\mathbb{P}(B)$ denote the probability of finding a bijection using RecSplit's brute force strategy. Then,
$\mathbb{P}(R) \to m\mathbb{P}(B)$ for $m \to \infty$.





Rotation Fitting (2/3)

Proof (Sketch)

- consider number of different injective functions under cyclic shifts
- bit vector of length m with $\mathbb{B}$ set bits
- total number of equivalence classes under rotation is $\frac{1}{m}\sum_{d \text{ divides } \gcd(\mathbb{A},\mathbb{B})} \phi(d)\binom{m/d}{\mathbb{B}/d}$
- probability of the event $\mathcal{I}$ that there is a rotation has the m least significant bits set is

$$\mathbb{P}(\mathcal{I}) \geq m \frac{1}{\sum_{d \text{ divides } \gcd(\mathbb{A},\mathbb{B})} \phi(d)\binom{m/d}{\mathbb{B}/d}},$$

- $\phi(i) = |\{j \leq i \colon \gcd(i,j) = 1\}|$ is Euler's totient function

Proof (Sketch, cnt.)

- determine the probability $\mathbb{P}(R)$ using the events
  - $\mathcal{A}$: `popcount(a)=`$\mathbb{A}$
  - $\mathcal{B}$: `popcount(b)=`$\mathbb{B}$
  - B: found bijection using brute-force

Rotation Fitting (3/3)

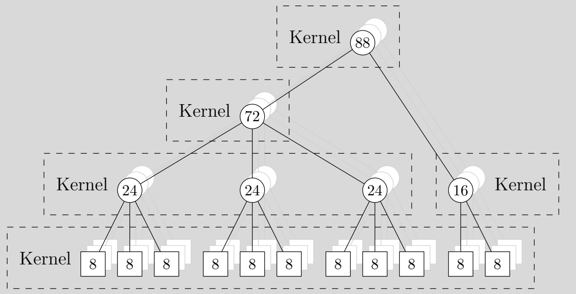
Proof (Sketch, ctn.)

$$\begin{aligned}
\mathbb{P}(R) &= \mathbb{P}(\mathcal{A})\mathbb{P}(\mathcal{B})\mathbb{P}(\mathcal{I}) \\
&\geq \frac{m!}{(m-\mathbb{A})!m^{\mathbb{A}}} \cdot \frac{m!}{(m-\mathbb{B})!m^{\mathbb{B}}} \cdot \mathbb{P}(\mathcal{I}) = \frac{m!}{m^m} \cdot \frac{m!}{\mathbb{A}!\mathbb{B}!} \cdot \mathbb{P}(\mathcal{I}) = \mathbb{P}(B) \cdot \frac{m!}{\mathbb{A}!\mathbb{B}!} \cdot \mathbb{P}(\mathcal{I}) \\
&\geq \mathbb{P}(B) \cdot \frac{m!}{\mathbb{A}!\mathbb{B}!} \cdot m \frac{1}{\sum_{d|\mathrm{gcd}(\mathbb{A},\mathbb{B})} \phi(d)\binom{m/d}{b/d}} = \mathbb{P}(B) \cdot m \cdot \frac{m!}{m! + (\mathbb{A}!\mathbb{B}!)\sum_{d|gcd(\mathbb{A},\mathbb{B}),d\neq 1} \phi(d)\binom{m/d}{b/d}} \\
&= \mathbb{P}(B) \cdot m \cdot \frac{1}{1 + \sum_{d|\mathrm{gcd}(\mathbb{A},\mathbb{B}),d\neq 1} \phi(d) \frac{(m/d)!\mathbb{A}!\mathbb{B}!}{m!(\mathbb{A}/d)!(\mathbb{B}/d)!}} \\
&\sim \mathbb{P}(B) \cdot m \cdot \frac{1}{1 + \sum_{d|\mathrm{gcd}(\mathbb{A},\mathbb{B}),d\neq 1} \phi(d)\sqrt{d}\frac{\mathbb{A}^{\mathbb{A}-\mathbb{A}/d}\mathbb{B}^{\mathbb{B}-\mathbb{B}/d}}{m^{m-m/d}}} \\
&\to \mathbb{P}(B) \cdot m \text{ for } m \to \infty
\end{aligned}$$

Parallel RecSplit on the GPU

Computing on the GPU

- several streaming multiprocessors (SMs)
- each SM contains many arithmetic logic units (ALUs)
- several threads operat in lock-step (warp)
- to hide latencies, each SM is oversubscribed with more threads than ALUs
- in CUDA, kernels are functions that can be executed on the GPU
- a kernel is executed on a grid of thread blocks



- use GPU to determine splitting and bijections

Experimental Evaluation

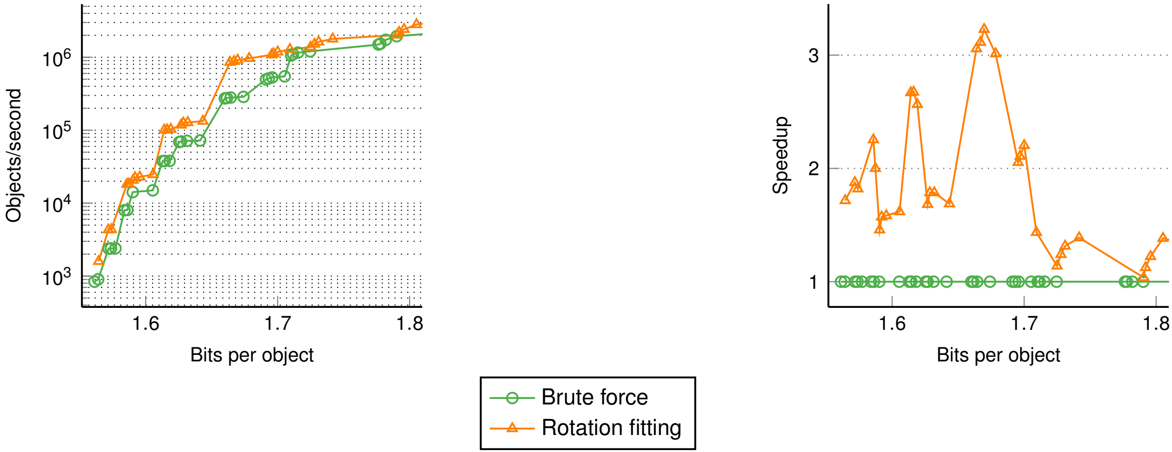
Karlsruhe Institute of Technology

- Intel i7 11700 processor with 8 cores (16 hardware threads (HT)), base clock: 2.5 GHz
- AVX-512.
- Ubuntu 22.04 with Linux 5.15.0
- NVIDIA RTX 3090 GPU

- AMD EPYC 7702P processor with 64 cores (128 hardware threads), base clock: 2.0 GHz
- AVX2
- Ubuntu 20.04 with Linux 5.4.0

- GNU C++ compiler v.11.2.0 (`-O3 -march=native`)

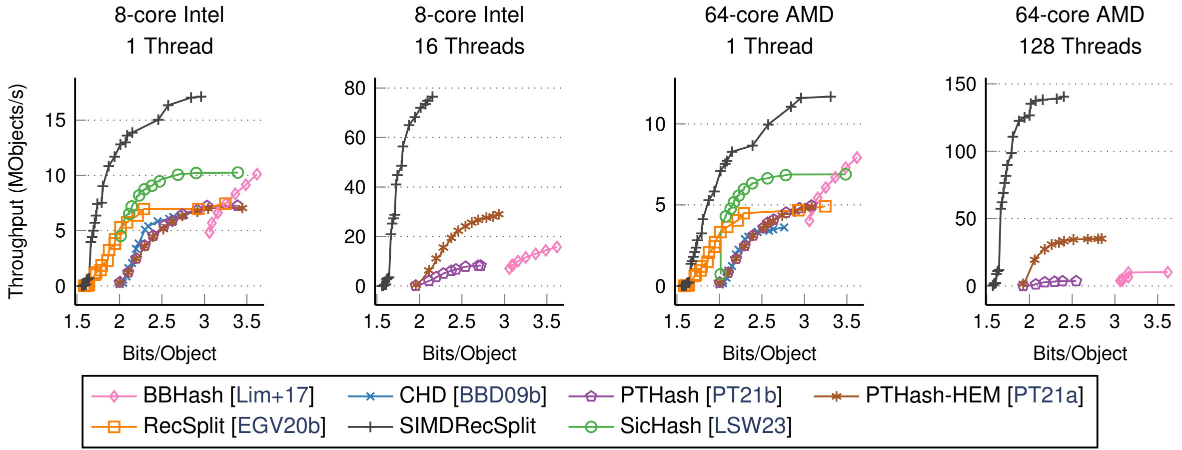
Rotation Fitting



Overview Results

Configuration	Method	Bijections	Threads	B/Obj	Constr.	Speedup
$\ell = 16, b = 2000$	RecSplit [EGV20b]	Brute force	1	1.560	1175.4	1
	RecSplit	Brute force	16	1.560	206.5	5
	SIMDRecSplit	Rotation fitting	1	1.560	138.0	8
	SIMDRecSplit	Rotation fitting	16	1.560	27.9	42
	GPURecSplit	Brute force	GPU	1.560	1.8	655
	GPURecSplit	Rotation fitting	GPU	1.560	1.0	1173
$\ell = 18, b = 50$	RecSplit [EGV20b]	Brute force	1	1.707	2942.9	1
	RecSplit	Brute force	16	1.713	504.0	5
	SIMDRecSplit	Rotation fitting	1	1.709	58.3	50
	SIMDRecSplit	Rotation fitting	16	1.708	12.3	239
	GPURecSplit	Brute force	GPU	1.708	5.2	564
	GPURecSplit	Rotation fitting	GPU	1.709	0.5	5438
$\ell = 24, b = 2000$	GPURecSplit	Brute force	GPU	1.496	2300.9	—
	GPURecSplit	Rotation fitting	GPU	1.496	467.9	—

Comparison with Competitors

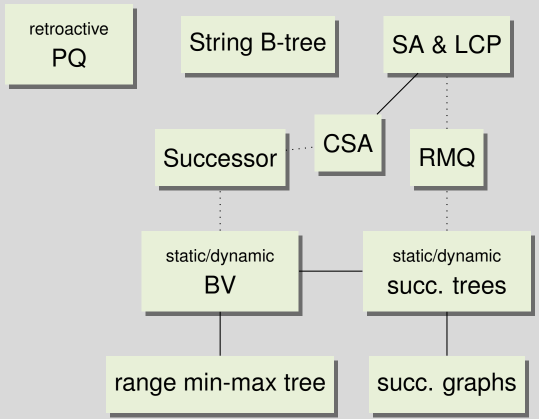


Conclusion and Outlook

This Lecture

- minimal perfect hash functions

Advanced Data Structures



Conclusion and Outlook

Karlsruhe Institute of Technology

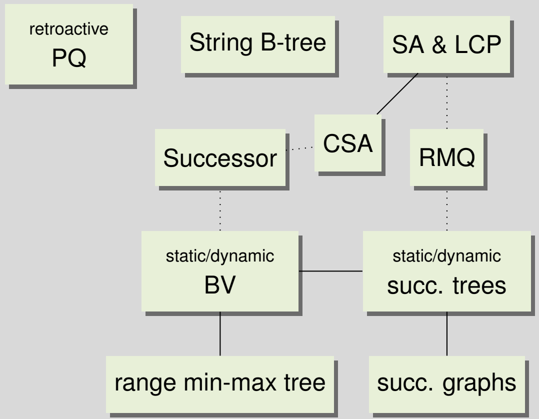
This Lecture

- minimal perfect hash functions

Next Lecture

- dynamic data structures

Advanced Data Structures



Bibliography I

[BBD09a] Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. “Hash, Displace, and Compress”. In: *ESA*. Volume 5757. Lecture Notes in Computer Science. Springer, 2009, pages 682–693. DOI: 10.1007/978-3-642-04128-0_61.

[BBD09b] Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. “Hash, Displace, and Compress”. In: *ESA*. Volume 5757. Lecture Notes in Computer Science. Springer, 2009, pages 682–693. DOI: 10.1007/978-3-642-04128-0_61.

[Bez+22] Dominik Bez, Florian Kurpicz, Hans-Peter Lehmann, and Peter Sanders. “High Performance Construction of RecSplit Based Minimal Perfect Hash Functions”. In: *CoRR* abs/2212.09562 (2022).

[BPZ13] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. “Practical perfect hashing in nearly optimal space”. In: *Inf. Syst.* 38.1 (2013), pages 108–131. DOI: 10.1016/j.is.2012.06.002.

Bibliography II

[EGV20a] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. "RecSplit: Minimal Perfect Hashing via Recursive Splitting". In: *ALENEX*. SIAM, 2020, pages 175–185. DOI: 10.1137/1.9781611976007.14.

[EGV20b] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. "RecSplit: Minimal Perfect Hashing via Recursive Splitting". In: *ALENEX*. SIAM, 2020, pages 175–185. DOI: 10.1137/1.9781611976007.14.

[Gol66] Solomon W. Golomb. "Run-length Encodings (Corresp.)". In: *IEEE Trans. Inf. Theory* 12.3 (1966), pages 399–401. DOI: 10.1109/TIT.1966.1053907.

[Lim+17] Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. "Fast and Scalable Minimal Perfect Hashing for Massive Key Sets". In: *SEA*. Volume 75. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 25:1–25:16. DOI: 10.4230/LIPICS.SEA.2017.25.

Bibliography III

[LSW23] Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. “SicHash – Small Irregular Cuckoo Tables for Perfect Hashing”. In: *ALENEX*. SIAM, 2023, pages 176–189. DOI: 10.1137/1.9781611977561.CH15.

[PT21a] Giulio Ermanno Pibiri and Roberto Trani. “Parallel and External-Memory Construction of Minimal Perfect Hash Functions with PTHash”. In: *CoRR* abs/2106.02350 (2021).

[PT21b] Giulio Ermanno Pibiri and Roberto Trani. “PTHash: Revisiting FCH Minimal Perfect Hashing”. In: *SIGIR*. ACM, 2021, pages 1339–1348. DOI: 10.1145/3404835.3462849.