PTHash

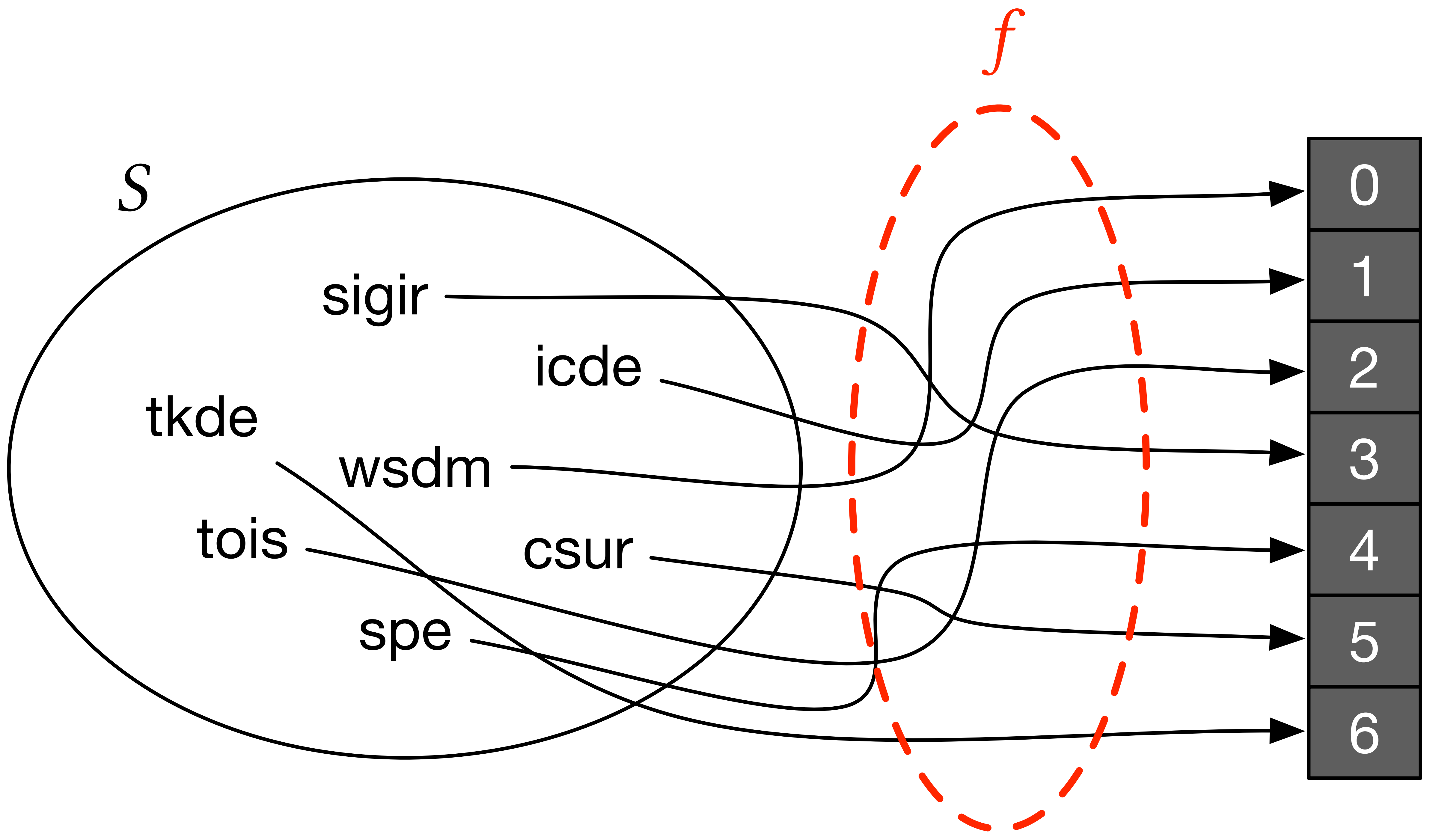
Revisiting FCH Minimal Perfect Hashing

Giulio Ermanno Pibiri, Roberto Trani
ISTI-CNR, Italy

Minimal Perfect Hashing (MPH)

Given a set S of n distinct keys, a function f that *bijectively* maps the keys of S into the range $\{0,\ldots,n-1\}$ is called a *minimal perfect hash function* for S.

- **Lower bound of 1.44 bits/key.**
- **Built once and evaluated many times.**
- Many algorithms available, like:
    - FCH (1992)
    - CHD (2009)
    - EMPHF (2014)
    - GOV (2016)
    - BBHash (2017)
    - RecSplit (2019)
    - PTHash (2021)



Context and Motivations

Space-efficient and *fast retrieval* of $\langle$*key*, *value*$\rangle$ pairs from a *static* set.

Some examples:

- Reserved words in programming languages.
- Garbage collectors.
- Command names in interactive systems.
- Lexicon of inverted indexes.
- Indexing of *q*-grams for language models.
- Indexing of *k*-mers of DNA.
- Web page URLs: DNS, page ranking, ecc.

Indeed MPH: Fast and Compact Immutable Key-Value Stores

Alex Shinn Follow
Feb 4, 2018 · 9 min read

https://medium.com/indeed-engineering/indeed-mph-fast-and-compact-immutable-key-value-stores-302a1cf2eef0

PTHash — Overview

- **Flexibility**: minimal and non-minimal perfect hash functions
- **Space/Time Efficiency**: fast lookup within compressed space
- **External-Memory Scaling**: use disk if not enough RAM is available
- **Parallel Construction**: use more threads to speed up construction
- **Configurable**: can offer different trade-offs (between construction time, lookup time, and space effectiveness)
- **C++ code** available at: https://github.com/jermp/pthash

code quality: c/c++ A+

FCH Construction

Fox, Chen, and Heath, SIGIR'92

- Distribute keys into m *buckets* using hashing and compute a displacement d_i for bucket i such that $f(x) = (h(x) + d_i) \bmod n$, and no collisions occur.
- Use $m = \lceil cn/\log_2 n \rceil$ buckets for n keys and a given parameter c.
- **One memory access per lookup.**

d_0	0	tkde		
d_1	5	sigir	spe	tois
d_2	2	icde		
d_3	5	csur	wsdm	

FCH Construction — Search

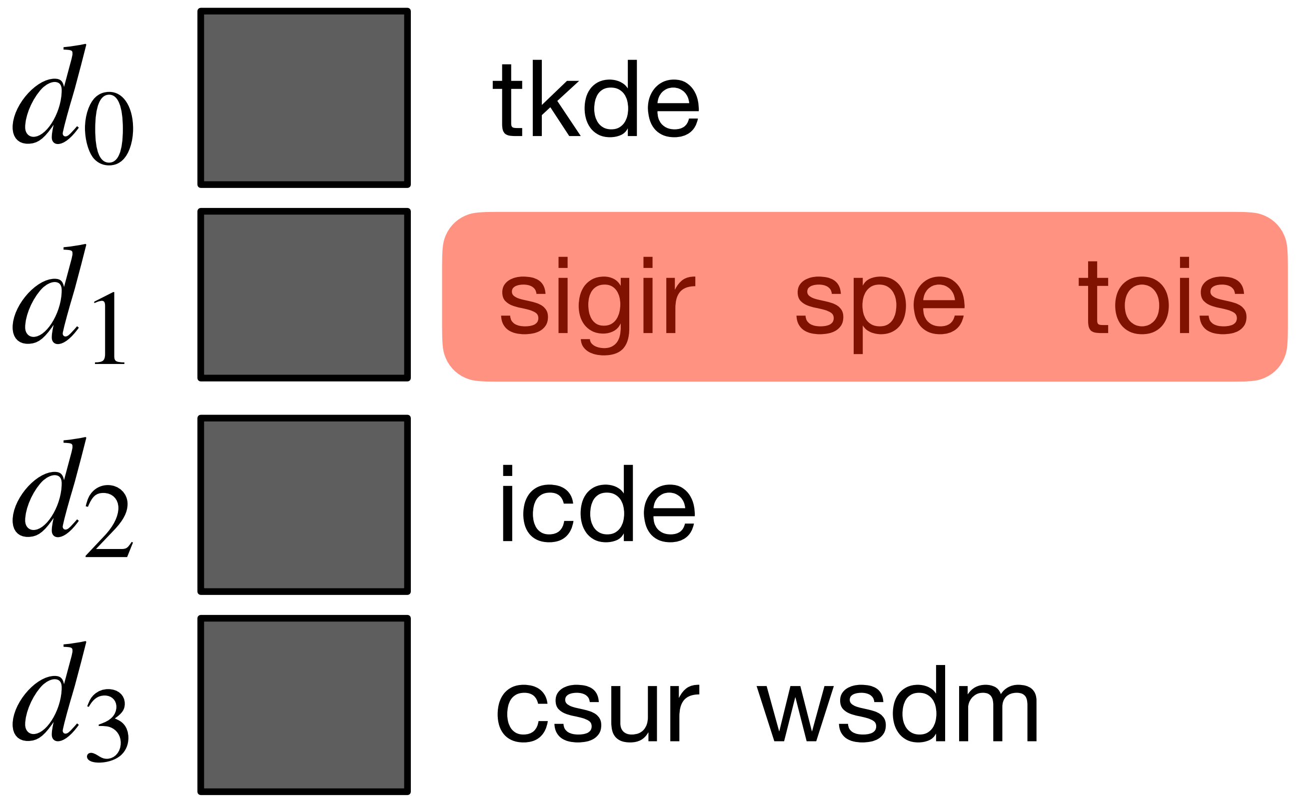
How to compute displacements?

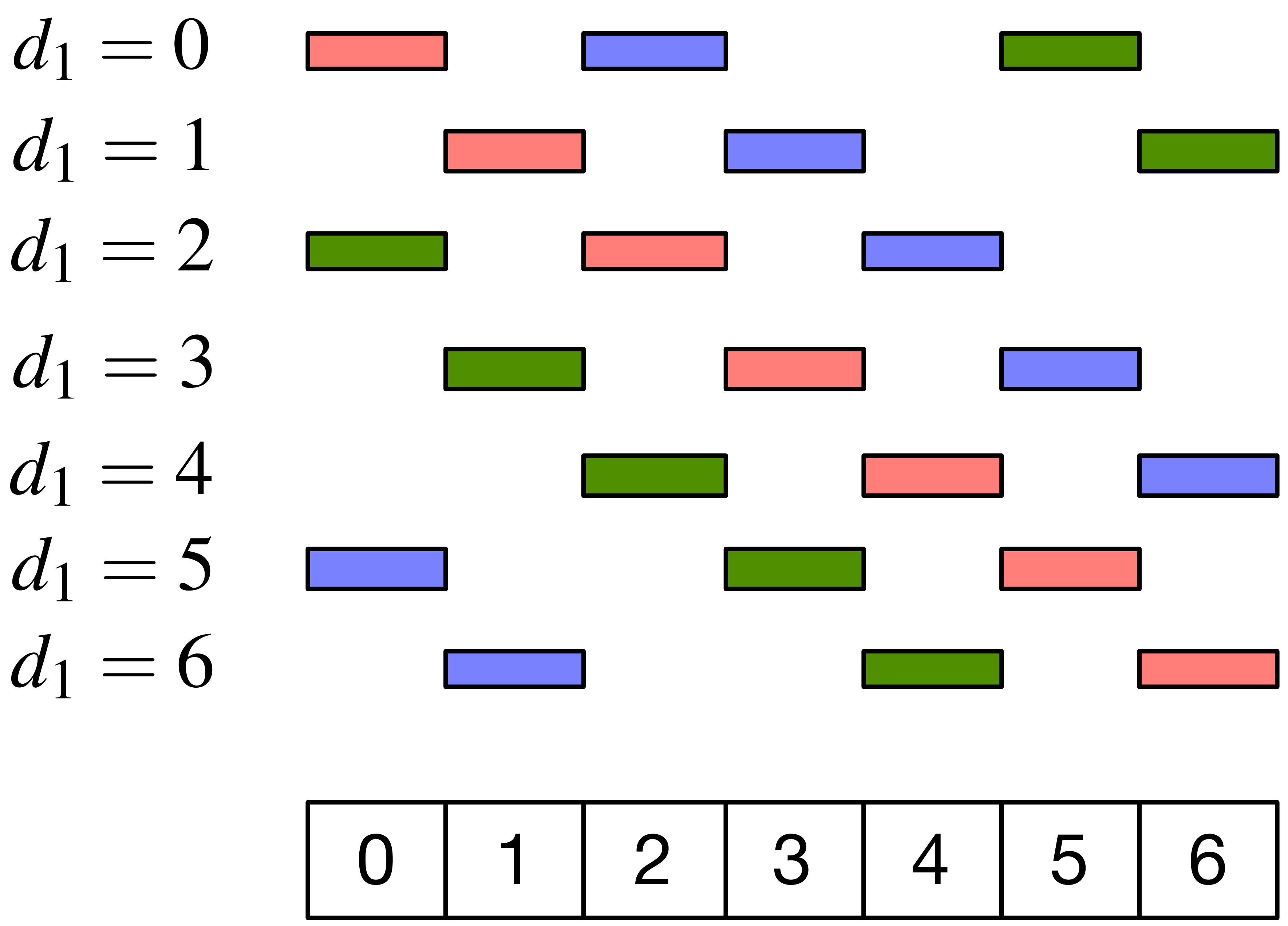
d_0	tkde
d_1	sigir spe tois
d_2	icde
d_3	csur wsdm

FCH Construction — Search

How to compute displacements?

$$f(x) = (h(x) + d_i) \bmod n$$

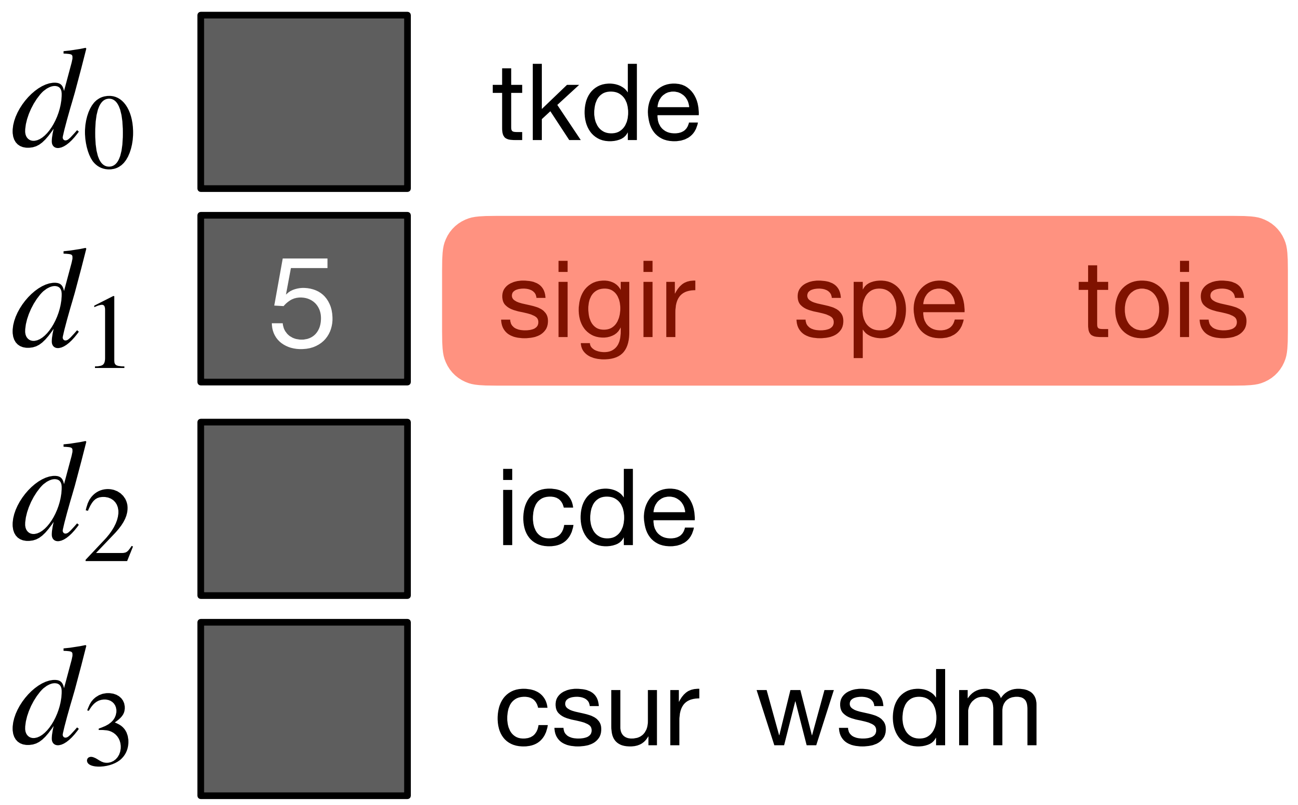


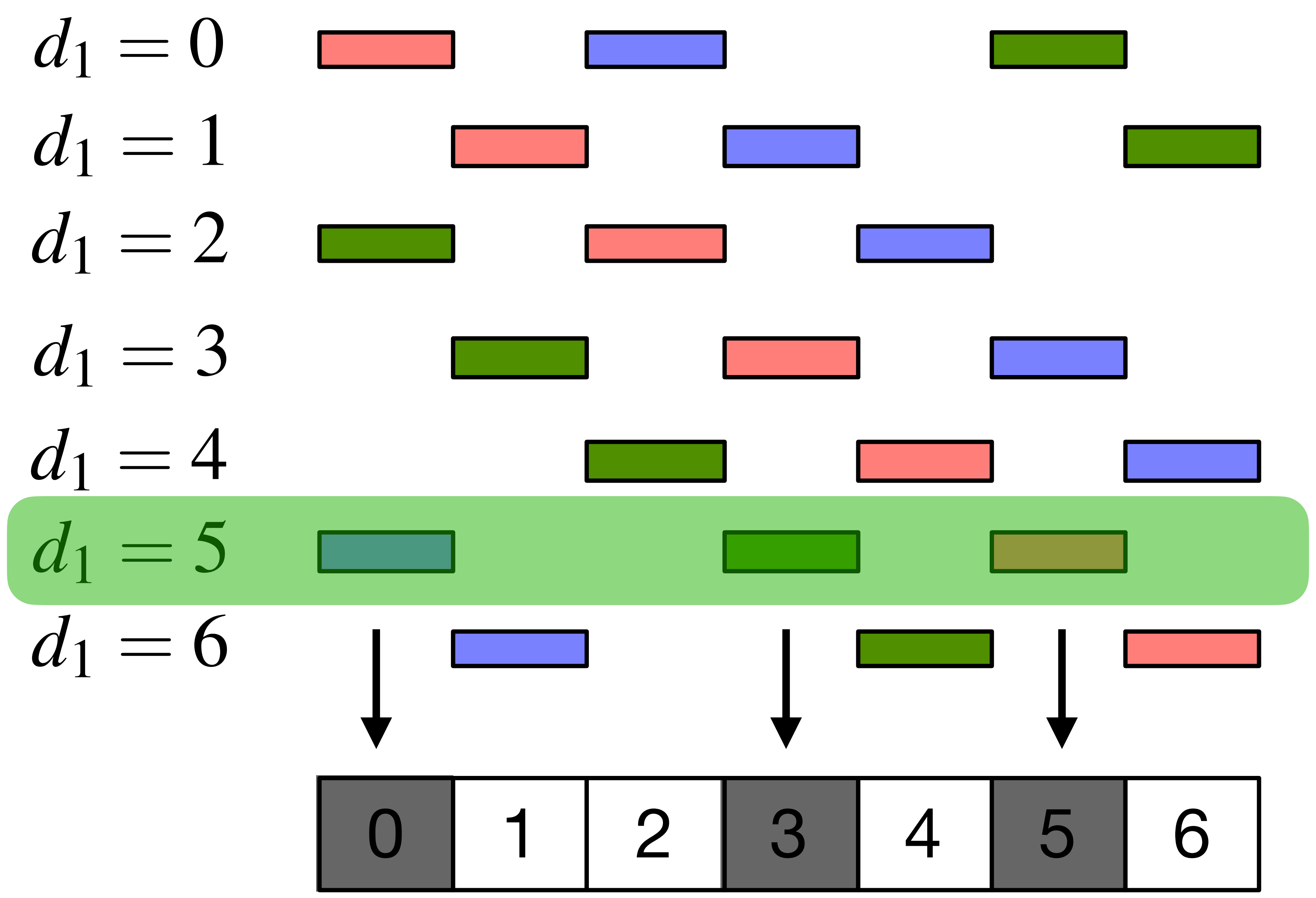


FCH Construction — Search

How to compute displacements?

$$f(x) = (h(x) + d_i) \bmod n$$

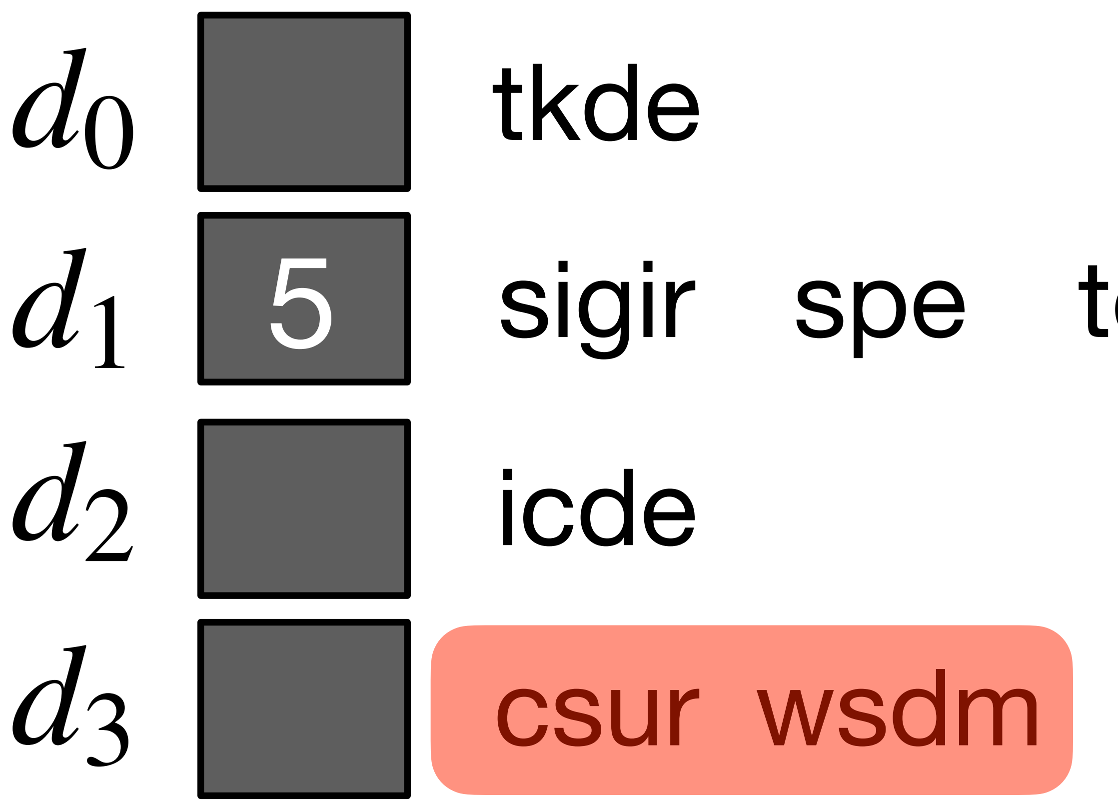


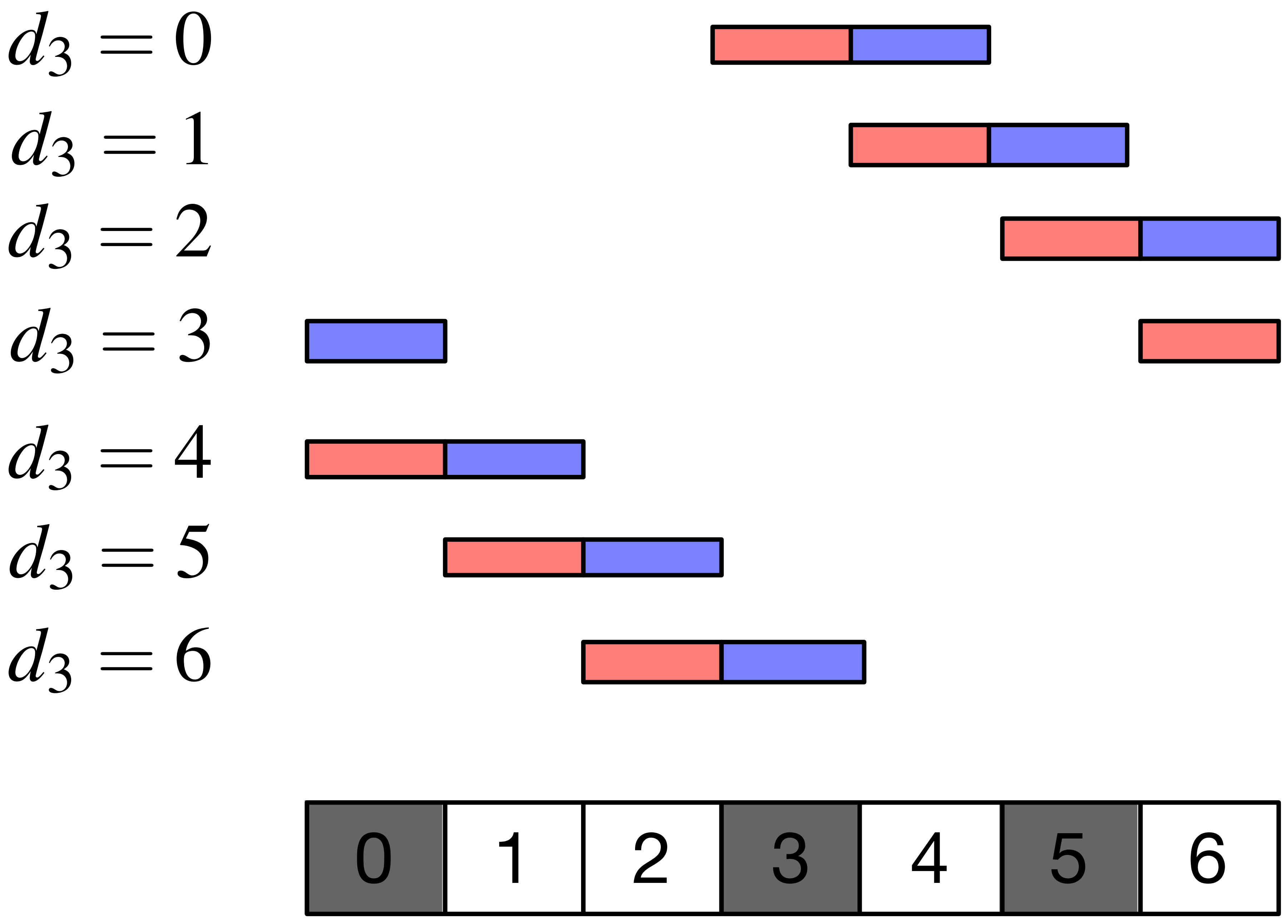


FCH Construction — Search

How to compute displacements?

$$f(x) = (h(x) + d_i) \bmod n$$

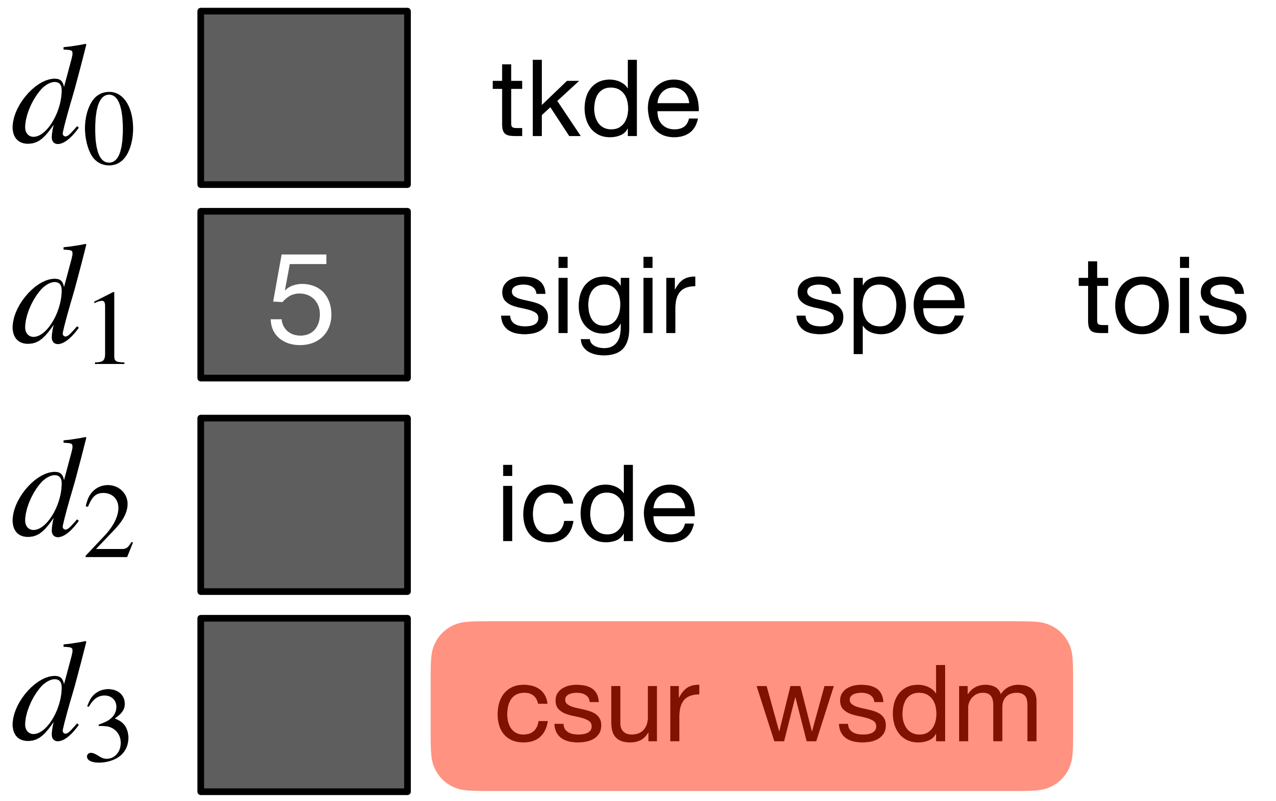


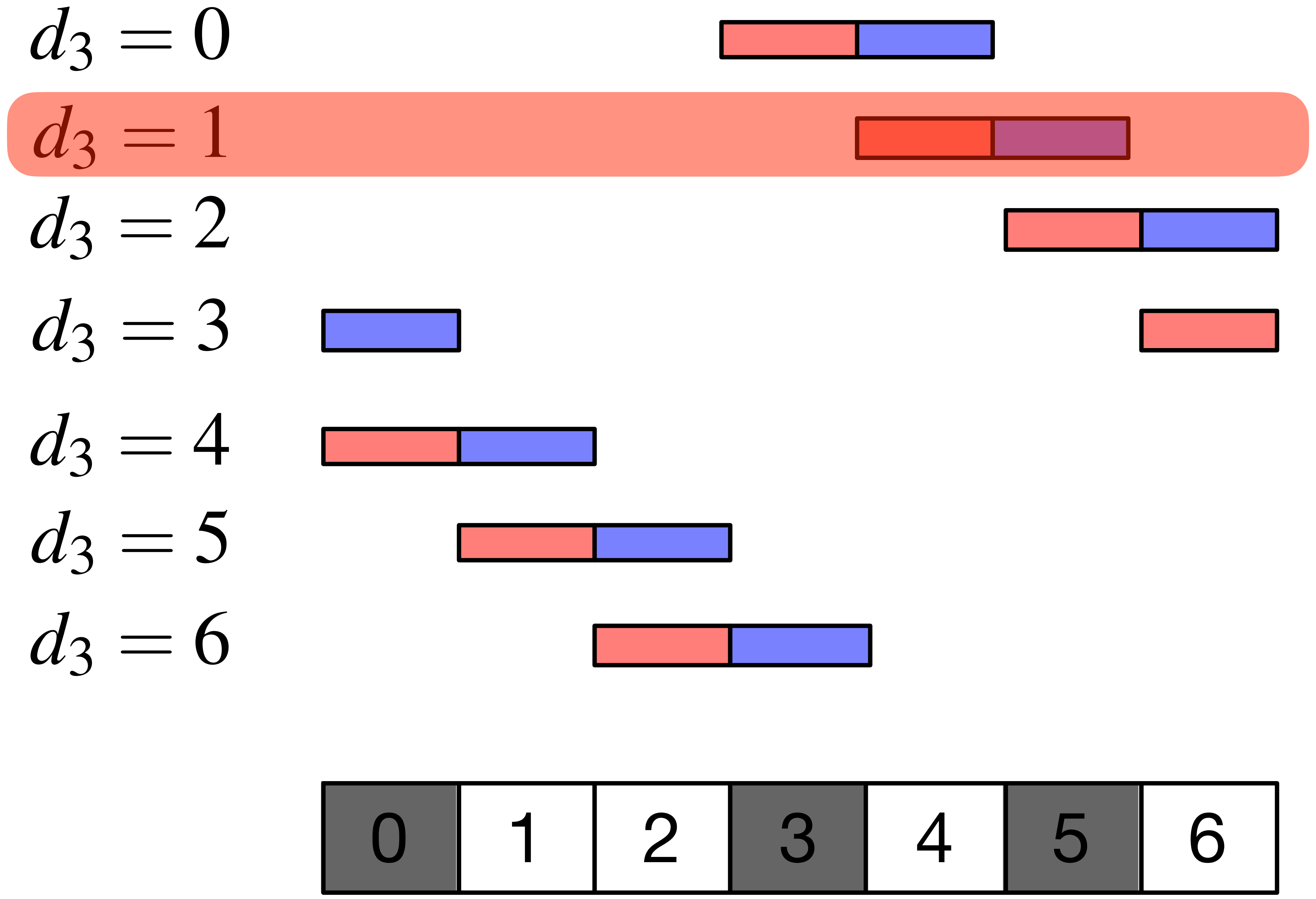


FCH Construction — Search

How to compute displacements?

$$f(x) = (h(x) + d_i) \bmod n$$

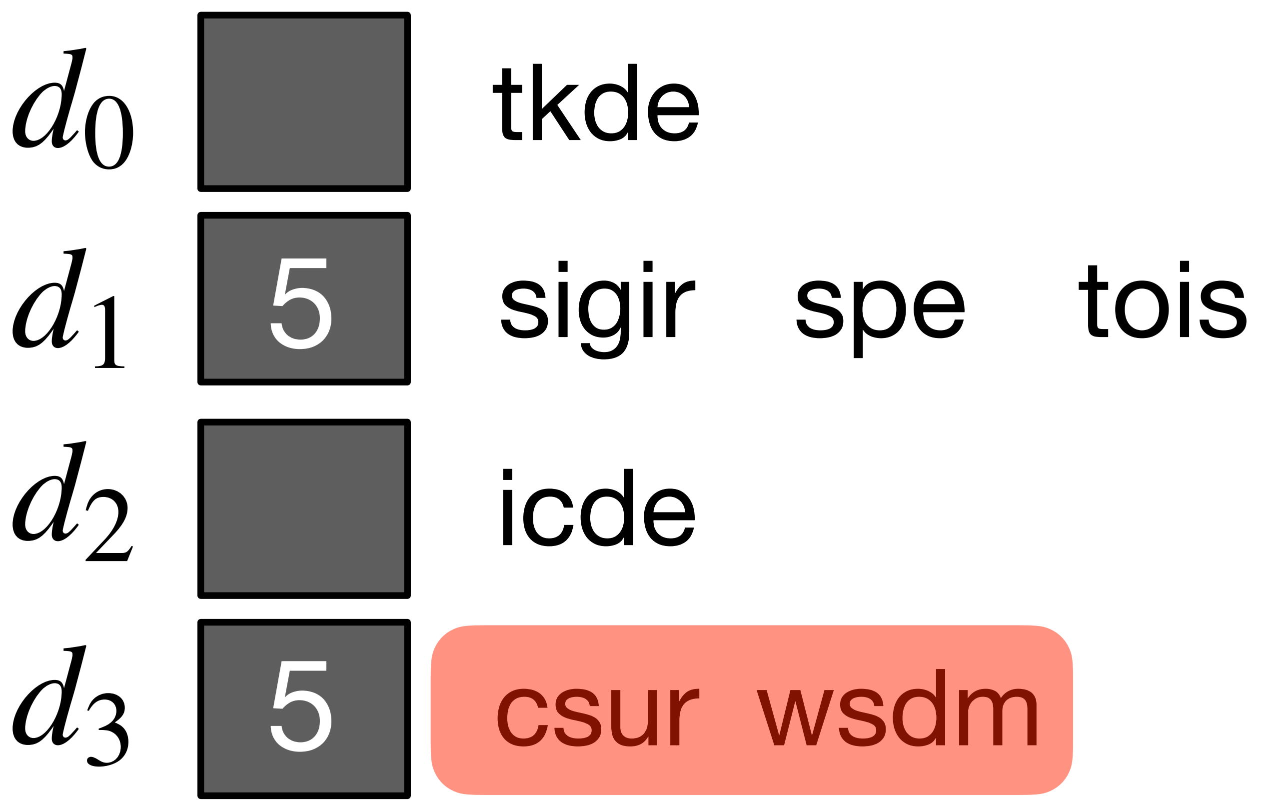


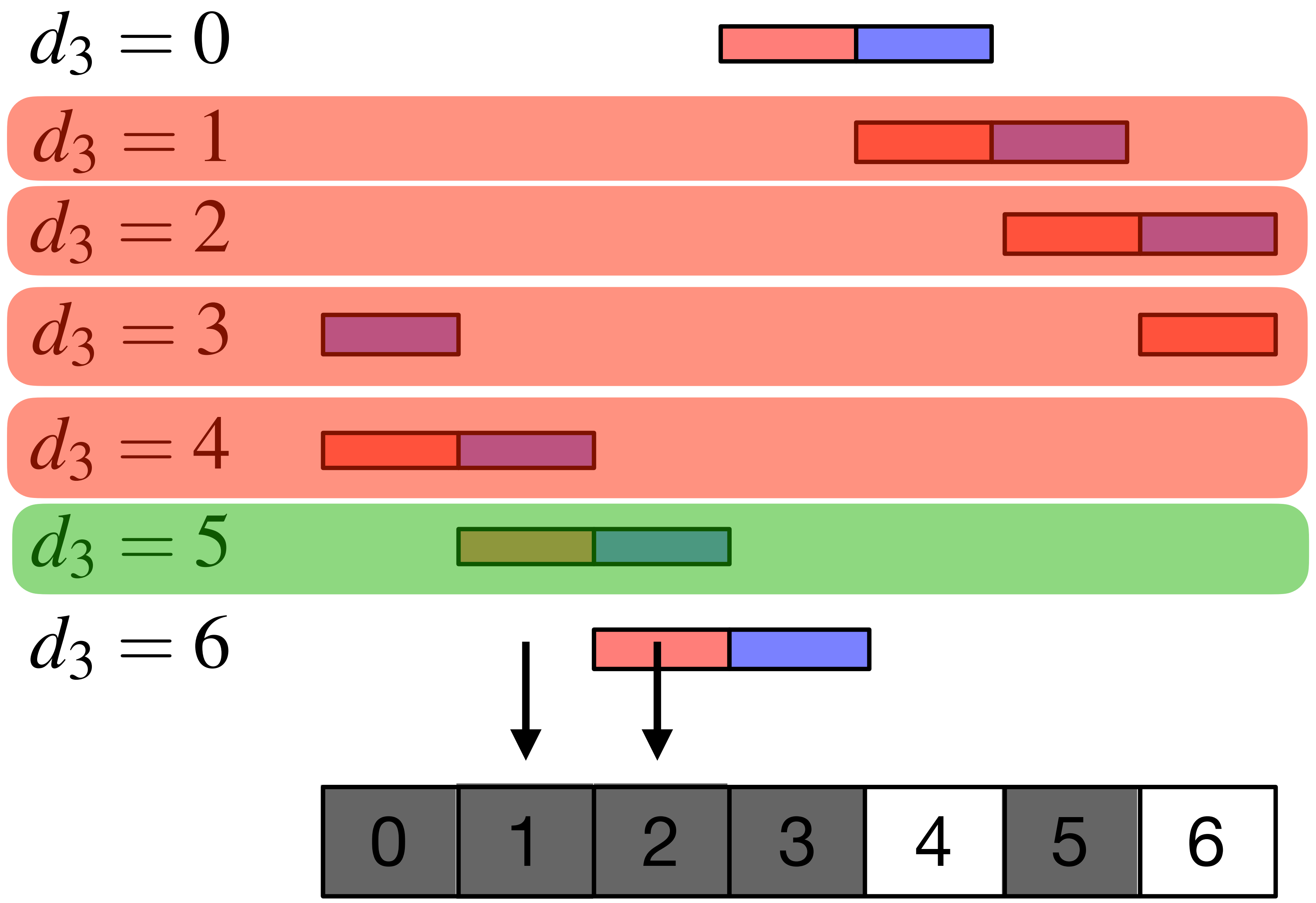


FCH Construction — Search

How to compute displacements?

$$f(x) = (h(x) + d_i) \bmod n$$





FCH Construction — Remarks

- To guarantee that all positions in the table are tested with *uniform probability*, displacements have to be tried at random → the best we can hope for is $\lceil \log_2 n \rceil$ bits per bucket.
  For $\lceil cn/\log_2 n \rceil$ buckets, it costs cn total bits. Large space for large c.

- Up to n trials to "fit" a pattern.
  If a successful displacement is not found for a bucket → rehash.
  Slow for small c.
  Example. For 10^8 64-bit random keys and $c = 3.0$, FCH takes 1h 10m: other techniques can do the same in 1m or less.

- **Extremely fast lookup.**

Our Research Question

Is it possible to combine the lookup efficiency of FCH with fast construction on large datasets and good compression effectiveness?

PTHash — Intuition

- If the table of displacements were **compressible**, we could afford to use a parameter $c' > c$ and run the search faster, such that the size of the compressed table is $\approx cn$ bits.
- Now, how to achieve compression? Re-design the **search step**.

PTHash — From Displacements to Pilots

$$f(x) = (h(x) + d_i) \bmod n \quad \Rightarrow \quad f(x) = (h(x) \oplus h(k_i)) \bmod n$$

PTHash — From Displacements to Pilots

$$f(x) = (h(x) + d_i) \bmod n \quad \Rightarrow \quad f(x) = (h(x) \oplus h(k_i)) \bmod n$$

- The bitwise **XOR** between two random fingerprints is another random fingerprint → displacement of keys at random.
- New random patterns generated with every tried pilot, even when pilots are tried **in order**, that is:

  $$k_i = 0,1,2,3,\ldots$$

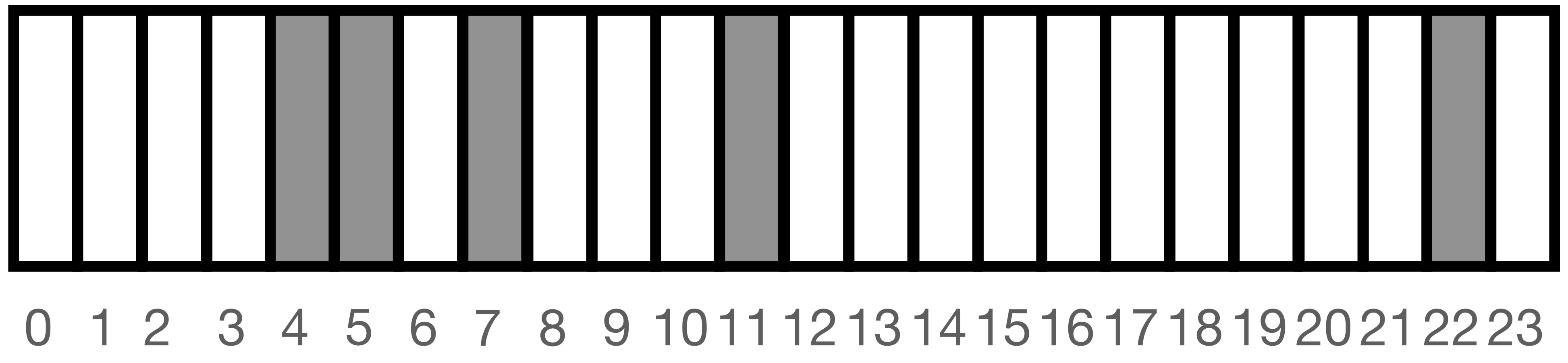
→ **Pilots will be small on average and repetitive, *hence* compressible.**

PTHash — From Displacements to Pilots

$x =$ “A View From the Top of the World”

$k_i =$ 0

$n = 24$



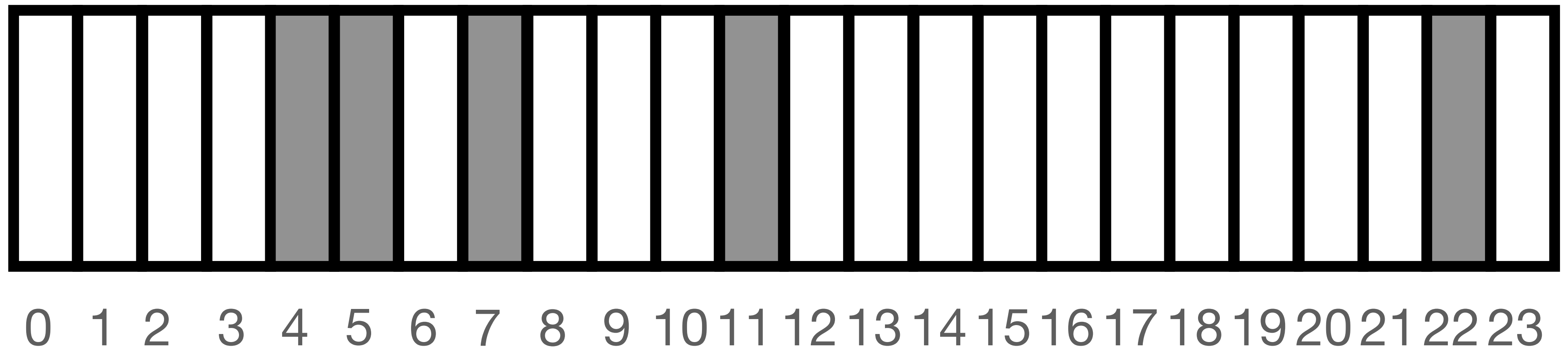
PTHash — From Displacements to Pilots

$x =$ “A View From the Top of the World”

$k_i =$ 0

$h(x) =$ 0001100101011101111001010000111111010101110001101111011100011011 57.8%

$n = 24$



PTHash — From Displacements to Pilots

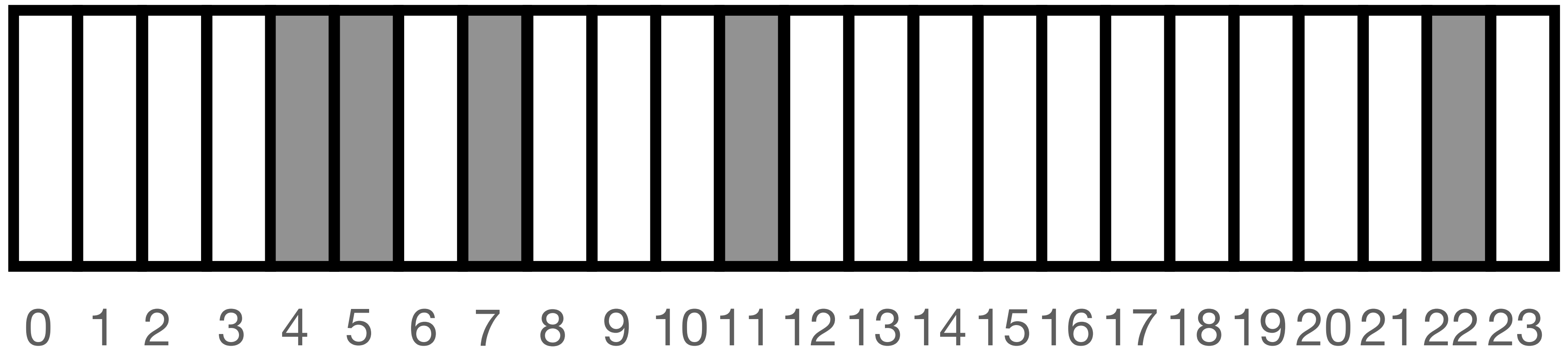
$x =$ "A View From the Top of the World"

$k_i =$ 0

$h(x) =$ 0001100101011101111001010000111111010101110001101111011100011011 57.8%

$h(k_i) =$ **1001010000110011110100010111001011011000001110010011111001000011** **48.4%**

$n = 24$



PTHash — From Displacements to Pilots

$x =$ “A View From the Top of the World”

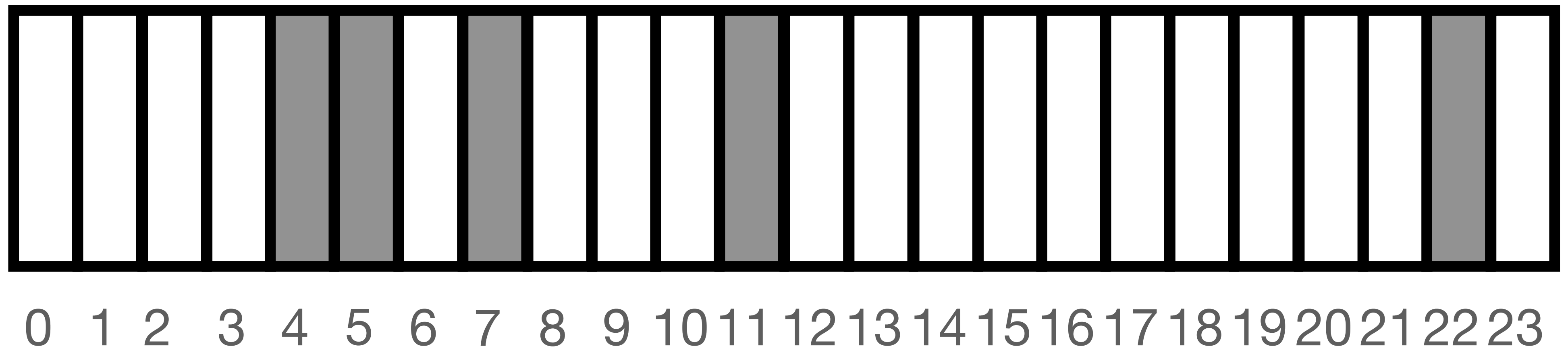
$k_i =$ 0

$h(x) =$ 0001100101011101111001010000111111010101110001101111011100011011 57.8%

$h(k_i) =$ 1001010000110011110100010111001011011000001110010011111001000011 48.4%

$h(x) \oplus h(k_i) =$ 1000110101101110001101000111110100001101111111111001001010110000 56.2%

$n = 24$



PTHash — From Displacements to Pilots

$x =$ “A View From the Top of the World”

$k_i =$ **0**

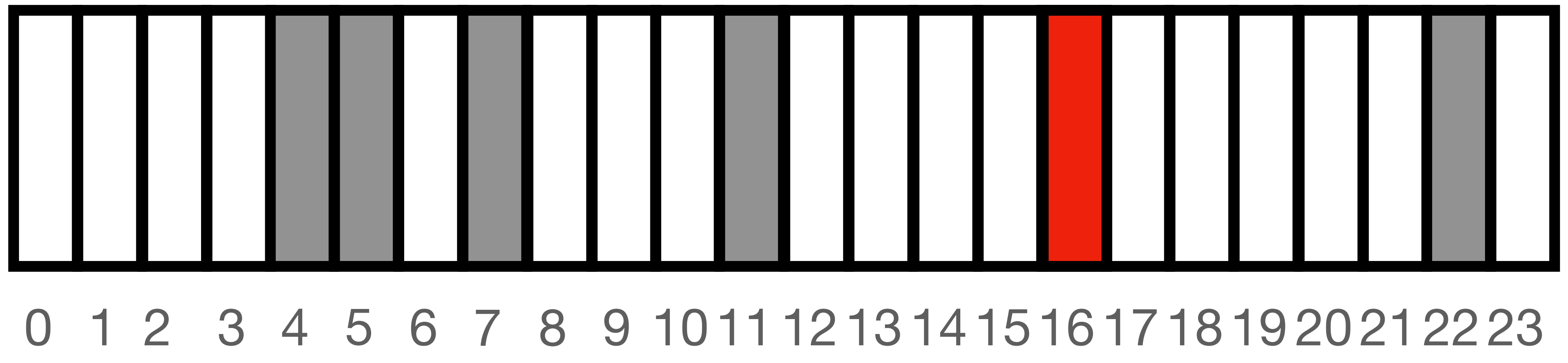
$h(x) =$ 0001100101011101111001010000111111010101110001101111011100011011   57.8%

$h(k_i) =$ **1001010000110011110100010111001011011000001110010011111001000011**   **48.4%**

$h(x) \oplus h(k_i) =$ **1000110101101110001101000111110100001101111111111001001010110000**   **56.2%**

$\downarrow \bmod 24$

$n = 24$



PTHash — From Displacements to Pilots

$x =$ "A View From the Top of the World"

$k_i =$ 1

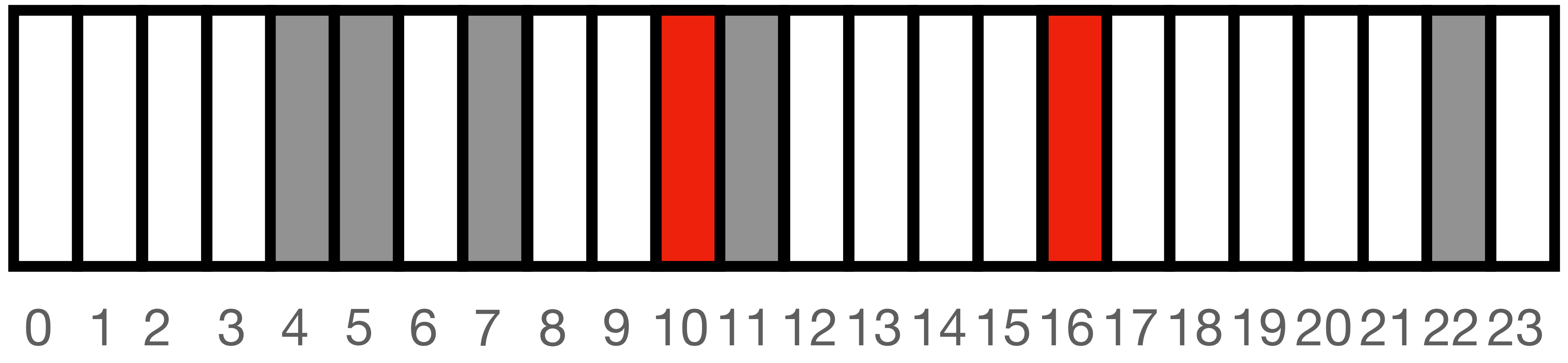
$h(x) =$ 0001100101011101111001010000111111010101110001101111011100011011 57.8%

$h(k_i) =$ **0101110001011100001010011001111010001001001011110100010000101001** **45.3%**

$h(x) \oplus h(k_i) =$ **0100010100000001110011001001000101011100111010011011001100110010** **43.7%**

$\downarrow \bmod 24$

$n = 24$



PTHash — From Displacements to Pilots

$x =$ “A View From the Top of the World”

$k_i =$ 2

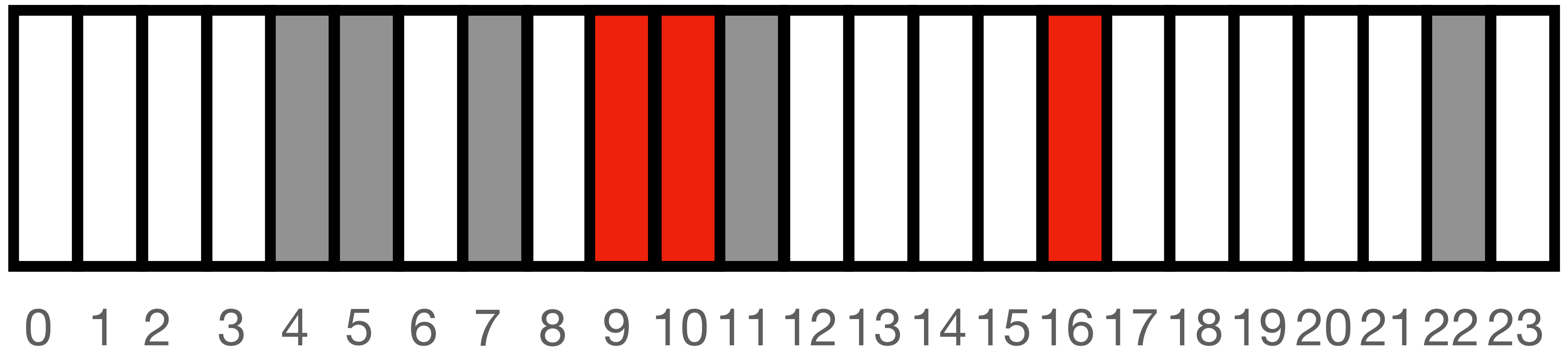
$h(x) =$ 0001100101011101111001010000111111010101110001101111011100011011 57.8%

$h(k_i) =$ **001000001011001111111000111110101001000010110110011101101101010** **53.1%**

$h(x) \oplus h(k_i) =$ **0011100111101110000111011111001010011101100111011100110001110001** **57.8%**

$\big\downarrow \bmod 24$

$n = 24$



PTHash — From Displacements to Pilots

$x =$ "A View From the Top of the World"

$k_i =$ 3

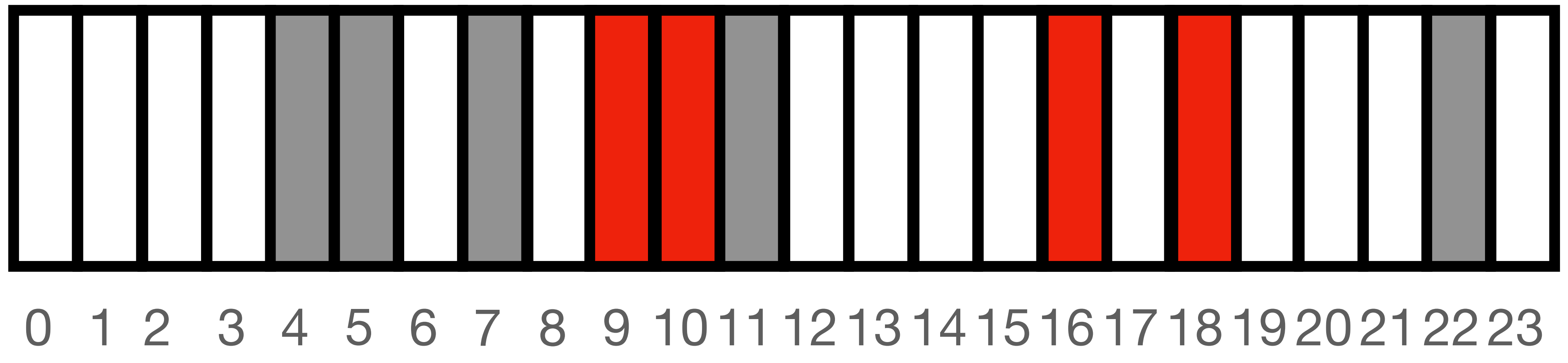
$h(x) =$ 0001100101011101111001010000111111010101110001101111011100011011 57.8%

$h(k_i) =$ 0111110010000101100011000110101000010110011101111101010001011001 50.0%

$h(x) \oplus h(k_i) =$ 0110010111011000011010010110010111000011101100010010001101000010 43.3%

$\downarrow \bmod 24$

$n = 24$



PTHash — From Displacements to Pilots

$x =$ “A View From the Top of the World”

$k_i =$ **4**

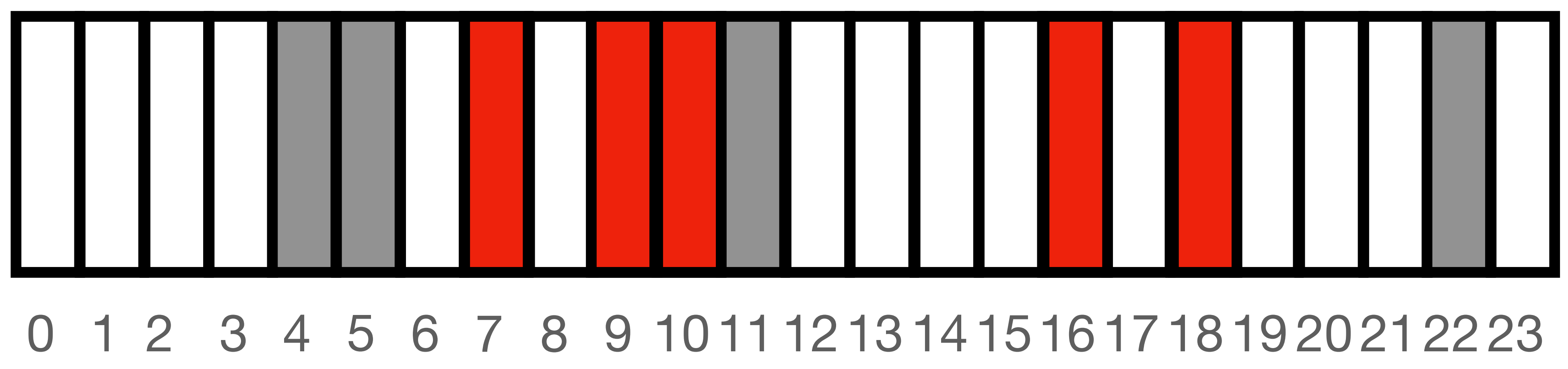
$h(x) =$ 0001100101011101111001010000111111010101110001101111011100011011 57.8%

$h(k_i) =$ **0000100111101110010111000100000111111011100100100010010000100100** **43.8%**

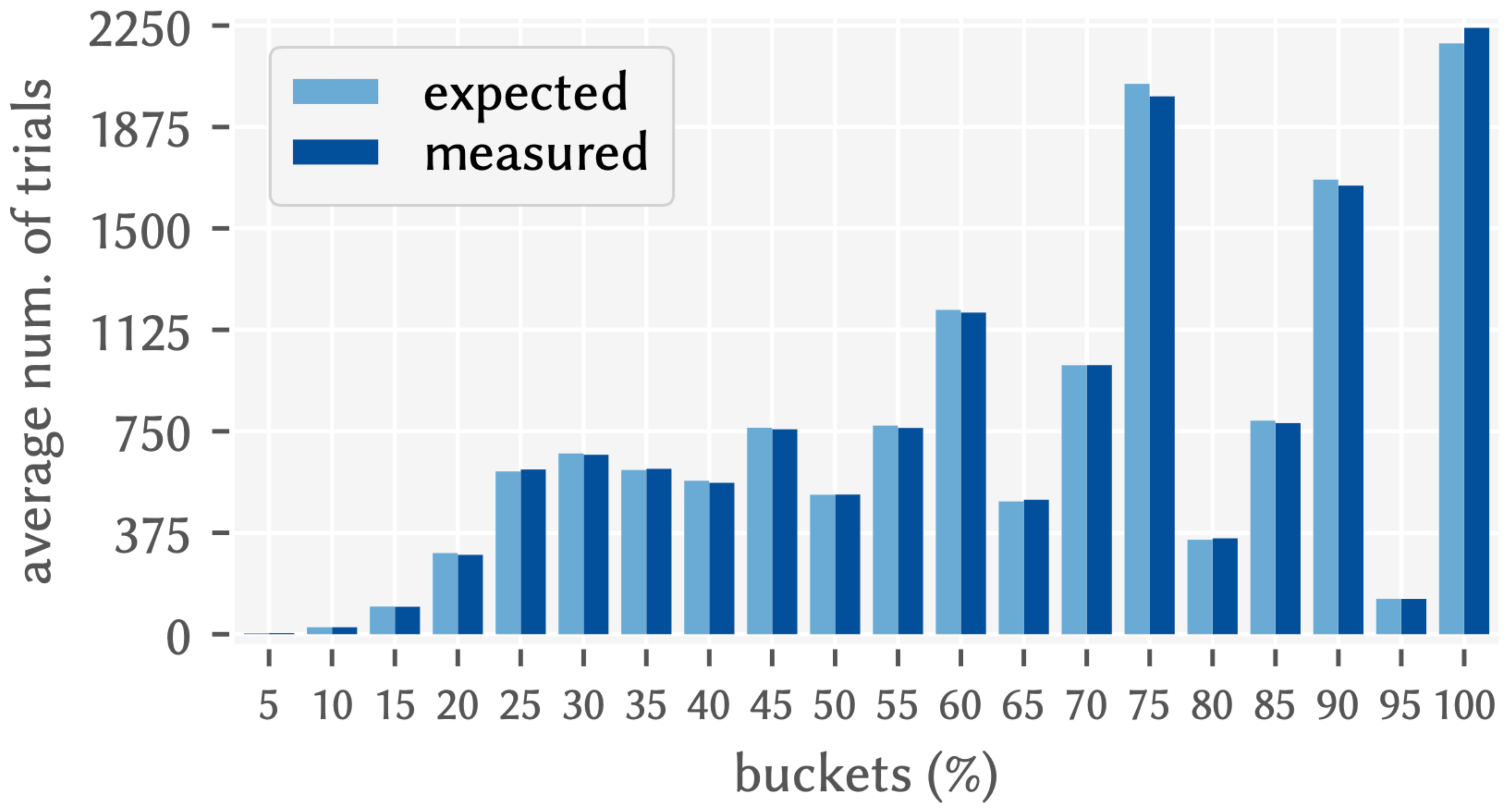
$h(x) \oplus h(k_i) =$ **0001000010110011101110010100111000101110010101001101001100111111** **51.6%**

$$\big\downarrow \bmod 24$$

$n = 24$



PTHash — From Displacements to Pilots



$n = 10^6$ keys, $c = 3.5$ (1.76×10^5 buckets)

$c \rightarrow$	2.5	3.0	3.5	4.0	4.5	5.0	5.5	6.0	6.5	7.0
FCH	16.69	16.85	16.93	16.93	16.87	16.77	16.65	16.49	16.32	16.14
PTHash	13.42	11.68	10.32	9.29	8.48	7.82	7.27	6.82	6.45	6.11

Empirical entropy of the tables, for $n = 10^6$ keys

PTHash — Limiting the Load Factor

- Allocate a search space of n/α slots, $0 < \alpha \leq 1$.
- More slots $\rightarrow$ **faster search and smaller pilots.**

(*Perfect* hash function: need to re-rank some positions to guarantee *minimal* output.
See the paper for details.)

PTHash — Example

- For 10^8 64-bit random keys and $c = 3.0$ (3 bits/key), FCH takes **1h 10m**.
- PTHash with $\alpha = 0.99$, $c = 6.8$, and **Front-Back Dictionary-based compression** achieves the **same space** (3 bits/key) but builds in **37s** (114×).
- **Both functions evaluate in 35-37 nanosec/key.**

Benchmark with 1B 64-bit random keys

- Processor: Intel i9-9900K @ 3.6 GHz, 32 KiB of L1, 256 KiB of L2 cache
- OS: Ubuntu 20
- Compiler: gcc 9.2.1, with flags `-march=native -O3`
- construction in internal memory
- construction is single-threaded

Method	$n = 10^9$ constr. (secs)	space (bits/key)	lookup (ns/key)
FCH, $c = 3$	—	—	—
FCH, $c = 4$	15904	4.00	35
FCH, $c = 5$	2937	5.00	35
FCH, $c = 6$	2133	6.00	35
FCH, $c = 7$	1221	7.00	35
CHD, $\lambda = 4$	1972	2.17	419
CHD, $\lambda = 5$	5964	2.07	417
CHD, $\lambda = 6$	23746	2.01	416
EMPHF	374	2.61	199
GOV	875	2.23	175
BBHash, $\gamma = 1$	253	3.06	170
BBHash, $\gamma = 2$	152	3.71	143
BBHash, $\gamma = 5$	100	6.87	113
RecSplit, ℓ=5, b=5	233	2.95	220
RecSplit, ℓ=8, b=100	936	1.80	204
RecSplit, ℓ=12, b=9	5700	2.23	197
PTHash			
(i) C-C, α=0.99, c=7	1042	3.23	37
(ii) D-D, α=0.88, c=11	308	3.94	64
(iii) EF, α=0.99, c=6	1799	2.17	101
(iv) D-D, α=0.94, c=7	689	2.99	55

(A part of) Table 5 from [1].

Benchmark with 1B 64-bit random keys

- Processor: Intel i9-9900K @ 3.6 GHz, 32 KiB of L1, 256 KiB of L2 cache
- OS: Ubuntu 20
- Compiler: gcc 9.2.1, with flags `-march=native -O3`
- construction in internal memory
- construction is single-threaded

Method	$n = 10^9$		
	constr. (secs)	space (bits/key)	lookup (ns/key)
FCH, $c = 3$	—	—	—
FCH, $c = 4$	15904	4.00	35
FCH, $c = 5$	2937	5.00	35
FCH, $c = 6$	2133	6.00	35
FCH, $c = 7$	1221	7.00	35
CHD, $\lambda = 4$	1972	2.17	419
CHD, $\lambda = 5$	5964	2.07	417
CHD, $\lambda = 6$	23746	2.01	416
EMPHF	374	2.61	199
GOV	875	2.23	175
BBHash, $\gamma = 1$	253	3.06	170
BBHash, $\gamma = 2$	152	3.71	143
BBHash, $\gamma = 5$	100	6.87	113
RecSplit, ℓ=5, b=5	233	2.95	220
RecSplit, ℓ=8, b=100	936	1.80	204
RecSplit, ℓ=12, b=9	5700	2.23	197
PTHash			
(i) C-C, α=0.99, c=7	1042	3.23	37
(ii) D-D, α=0.88, c=11	308	3.94	64
(iii) EF, α=0.99, c=6	1799	2.17	101
(iv) D-D, α=0.94, c=7	689	2.99	55

(A part of) Table 5 from [1].

Benchmark with large string collections

Dataset	Number of strings
ClueWeb09-Full URLs	4 780 950 911
GoogleBooks 3-gr	7 384 478 110

- **construction in external memory**
- **construction is multi-threaded**

Numbers in parentheses refer to the parallel construction using 8 threads.
All PTHash configurations use $\alpha = 0.94$ and $c = 7.0$.

Method	ClueWeb09-Full URLs			GoogleBooks 3-gr		
	construction (seconds)	space (bits/key)	lookup (ns/key)	construction (seconds)	space (bits/key)	lookup (ns/key)
PTHash (D-D)	7234 (4869)	2.96	120	9770 (5865)	2.91	91
PTHash (PC)	7161 (4859)	2.58	175	9756 (5736)	2.56	143
PTHash (EF)	7225 (4788)	2.32	214	9649 (5849)	2.31	208
PTHash-HEM (D-D)	4651 (3632)	2.75	152	5215 (3510)	2.71	135
PTHash-HEM (PC)	4522 (3541)	2.58	192	5015 (3366)	2.57	190
PTHash-HEM (EF)	4627 (3631)	2.32	235	5179 (3512)	2.31	230
EMPHF	24862	2.61	231	37731	2.61	220
EMPHF-HEM	3980	3.31	304	5606	3.06	304
GOV	8228 (5400)	2.23	232	10782 (6461)	2.23	242
BBHash ($\gamma = 1.0$)	19360 (18391)	3.07	320	20178 (9554)	3.07	305
BBHash ($\gamma = 2.0$)	11074 (10348)	3.71	236	10254 (5404)	3.71	235

(A part of) Table 5 from [2].

Conclusions

- PTHash is a novel MPHF algorithm combining good space effectiveness and fast construction with the excellent lookup performance of FCH.

- PTHash can be tuned to consume space similar to another method and, yet, it provides remarkably better lookup performance, with feasible or better construction speed.

- Our library at **https://github.com/jermp/pthash** is well-engineered and also supports multi-threading and external-memory scaling.

References

1. Giulio Ermanno Pibiri and Roberto Trani. "*PTHash: Revisiting FCH Minimal Perfect Hashing*". In Proceedings of the 44th International Conference on Research and Development in Information Retrieval (SIGIR). 2021.

2. Giulio Ermanno Pibiri and Roberto Trani. "*Parallel and External-Memory Construction of Minimal Perfect Hash Functions with PTHash*". ArXiv. 2021.