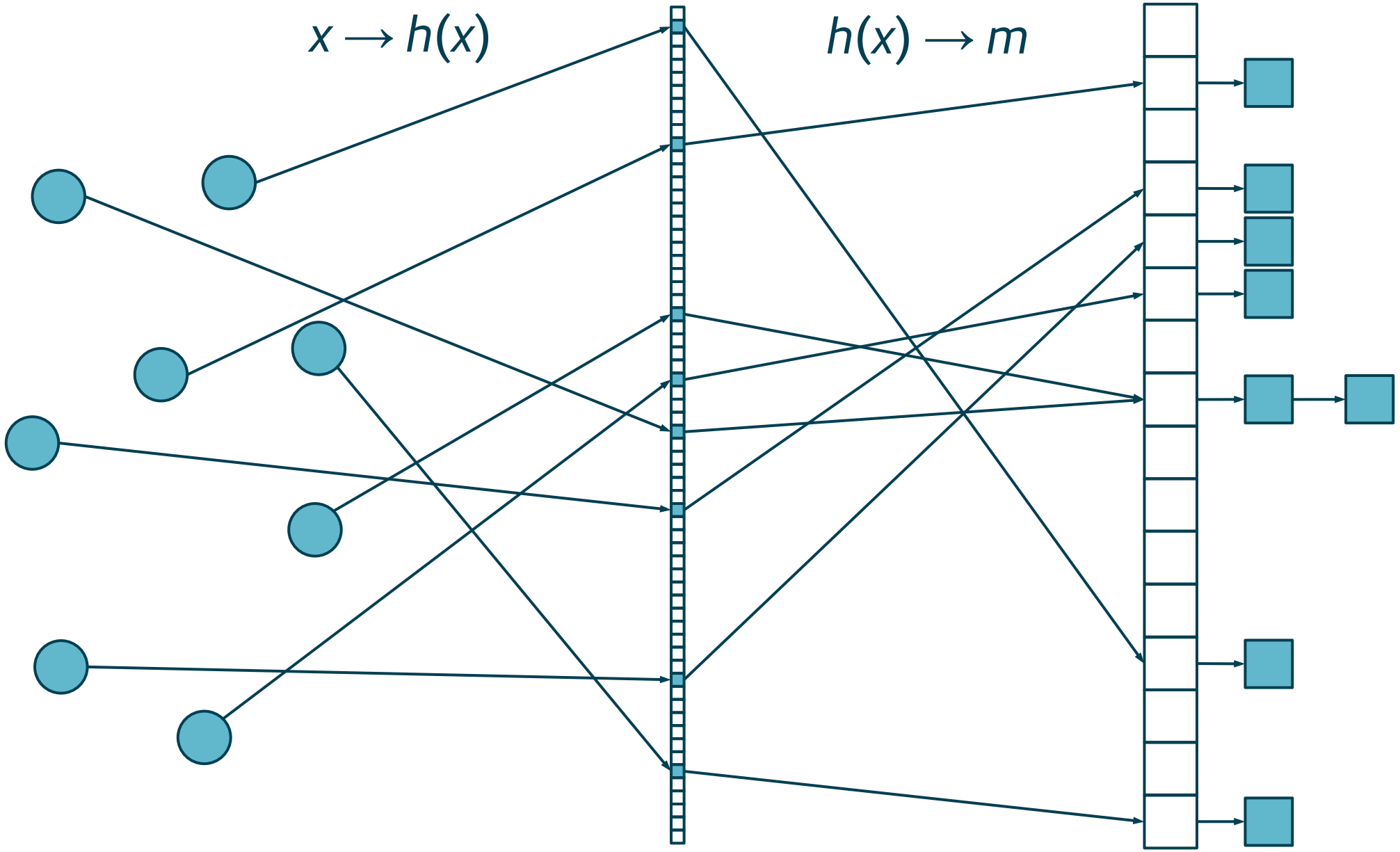
# Perfect Hashing in an Imperfect World

`using std::cpp 2024`

Joaquín M López Muñoz <joaquin.lopezmunoz@gmail.com>

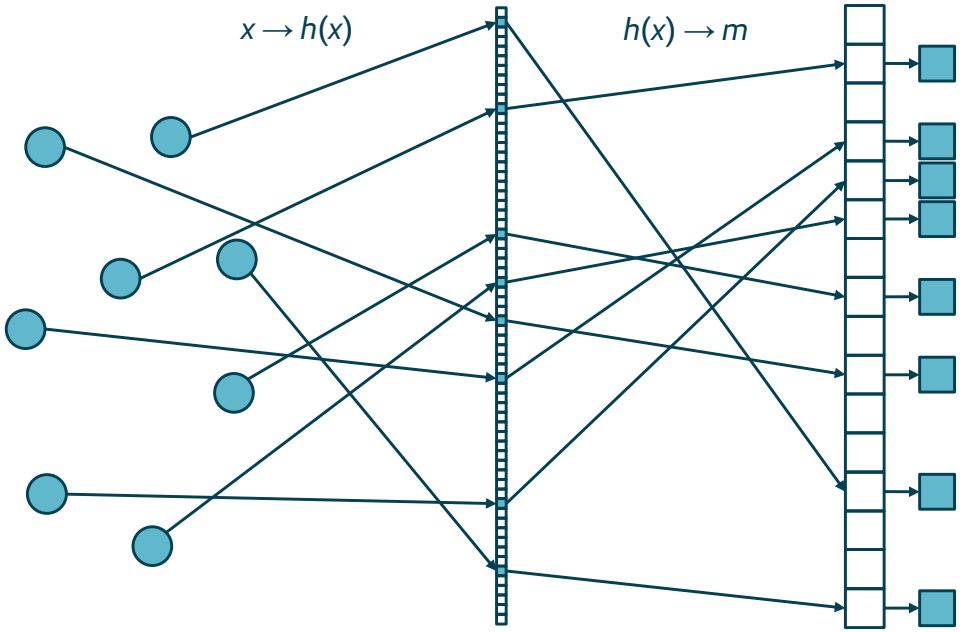
Madrid, April 2024


$x \to h(x)$
$h(x) \to m$

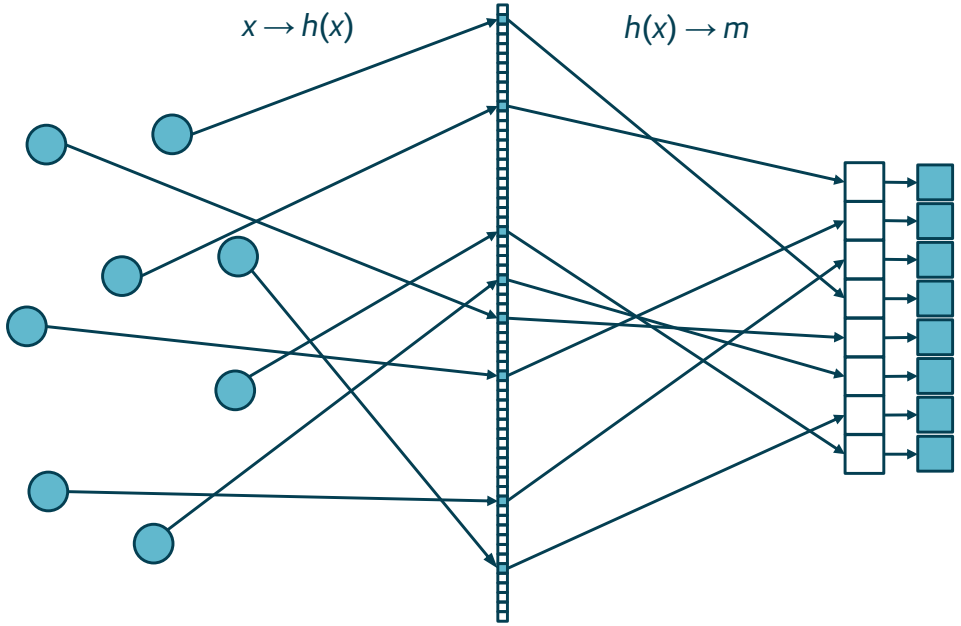
# Sometimes you get lucky



Perfect hashing

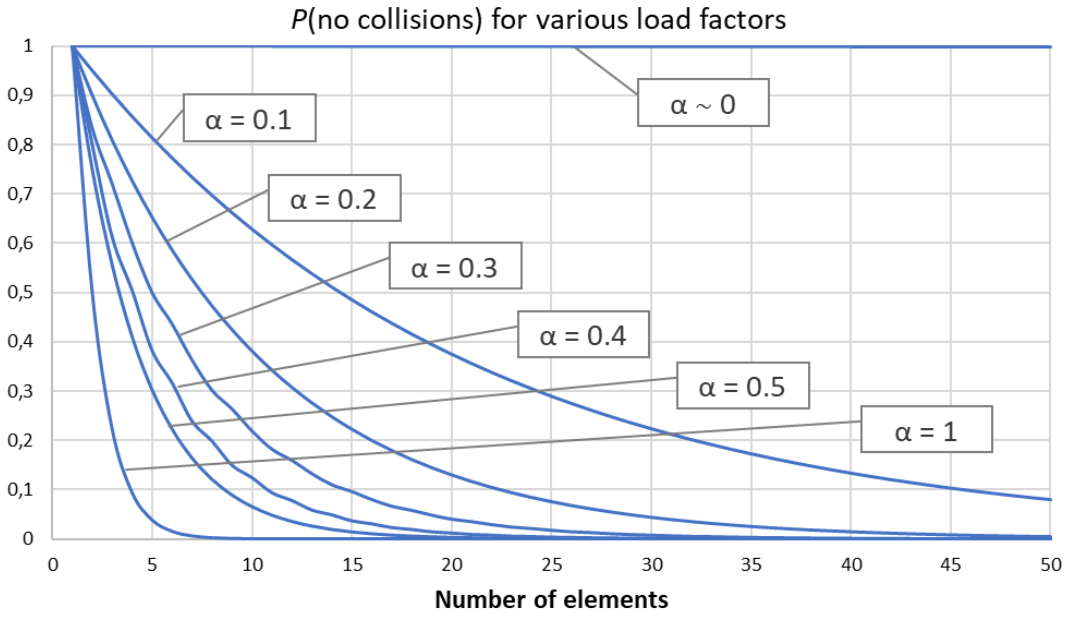



Minimal perfect hashing



# How lucky?

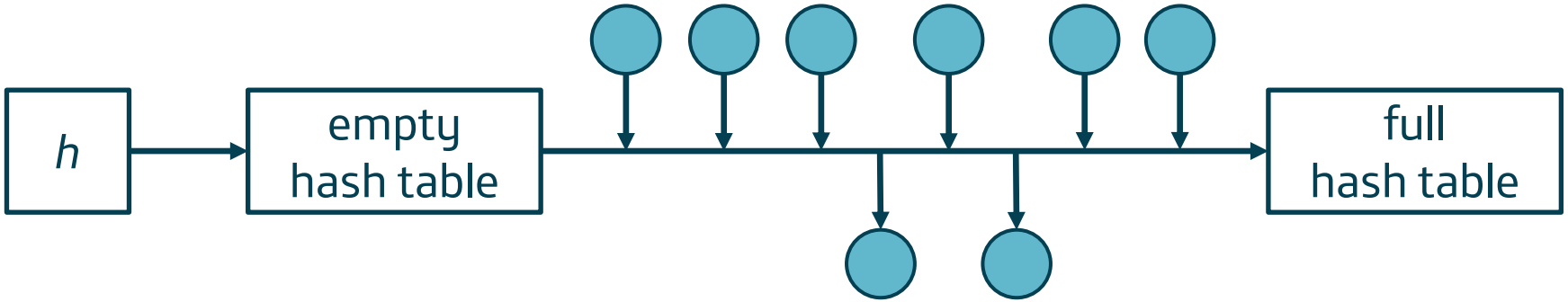
- $n$ elements, $m$ buckets — $P(\text{no collisions}) = \frac{\binom{m}{n} n!}{m^n} = \frac{m!}{m^n (m-n)!}$
- $m \to \infty$ — $P(\text{no collisions}) \to 1$
- $m = n^2$ — $P(\text{no collisions}) \geq 0.5$
- $\alpha = {}^{n}/_{m} = 1$ — $P(\text{no collisions}) = \frac{n!}{n^n} \sim \frac{\sqrt{2\pi n}}{e^n} = O\left(e^{c \ln n - n}\right)$



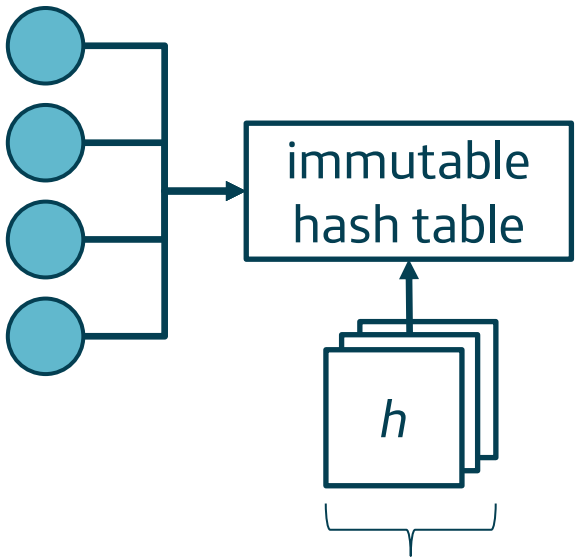

- So… very lucky

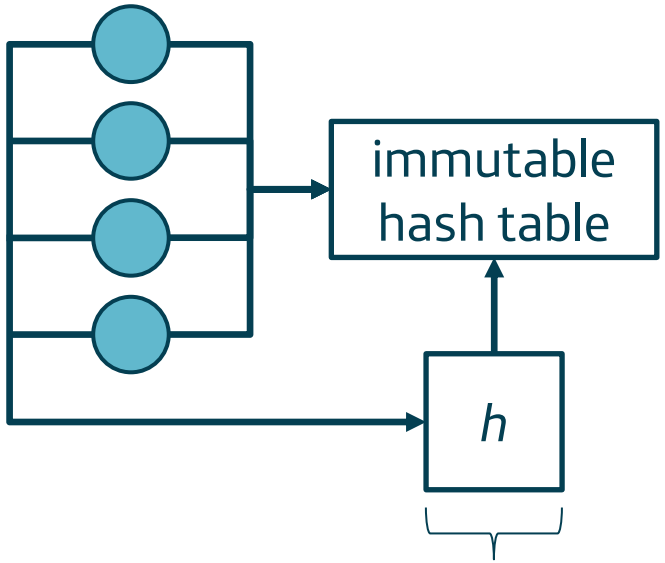
# Enlightenment

Regular hashing




Perfect hashing



# Pros and cons



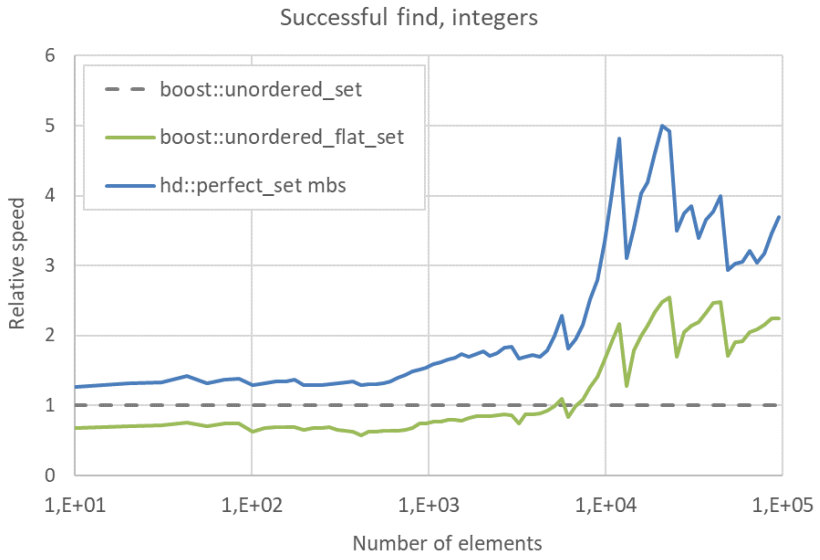
- Guaranteed $O(1)$ lookup
- And extremely fast at that
  - Very few branches
  - Particularly with minimal perfect hashing
  - Additional performance opportunities with `constexpr`

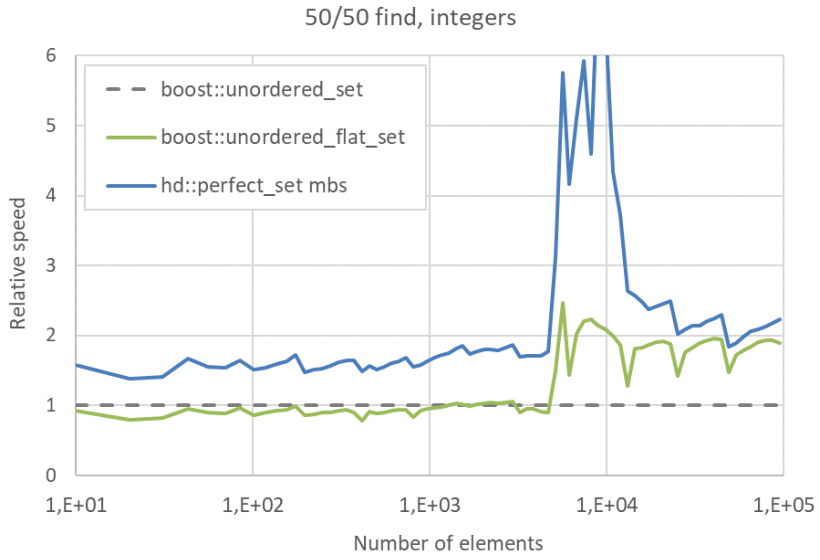
- Immutability
- Long construction times
- Can fail on initialization
  - Equivalent to pathological $O(n)$ behavior in regular hashing

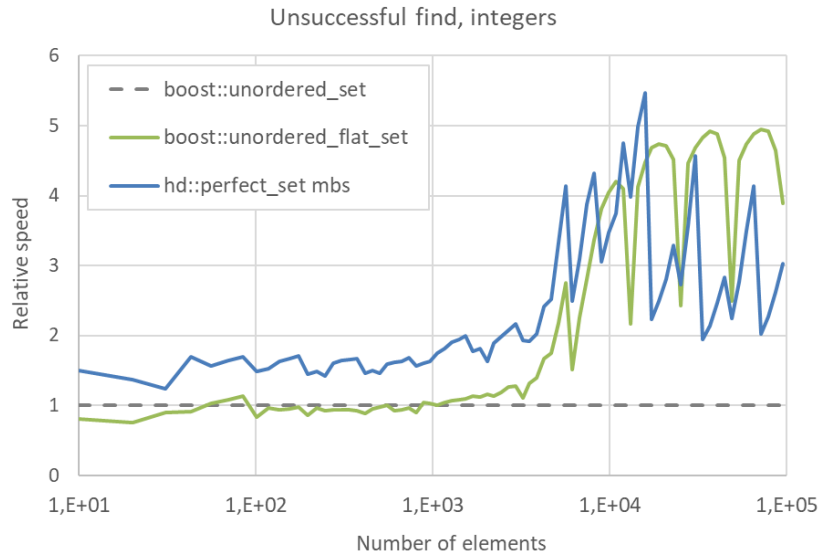
- Perfect hash tables are no replacement for regular hashing

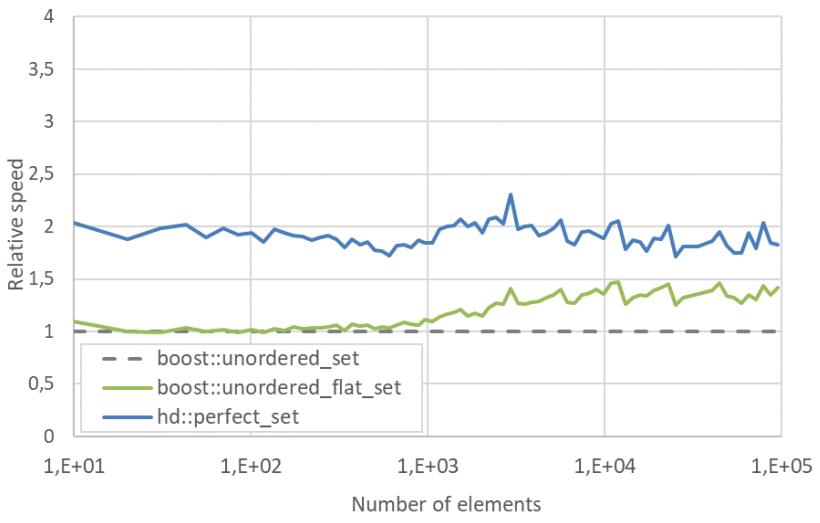
# Relative performance (without tuning)

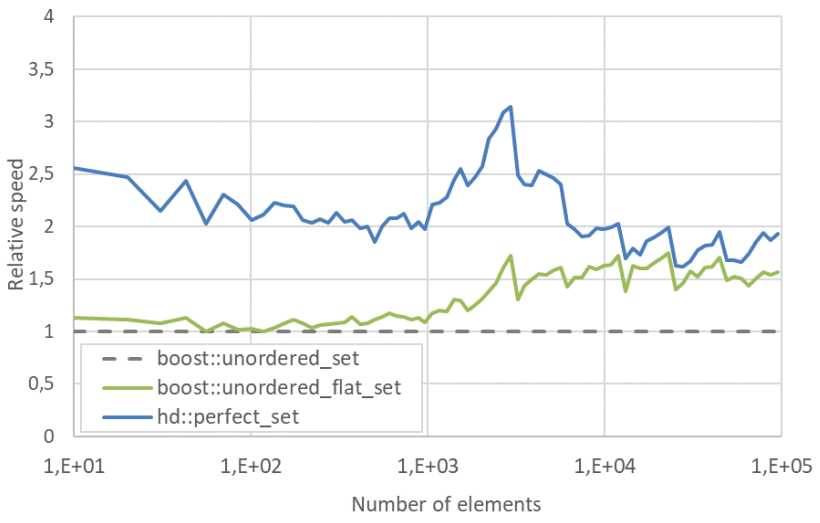


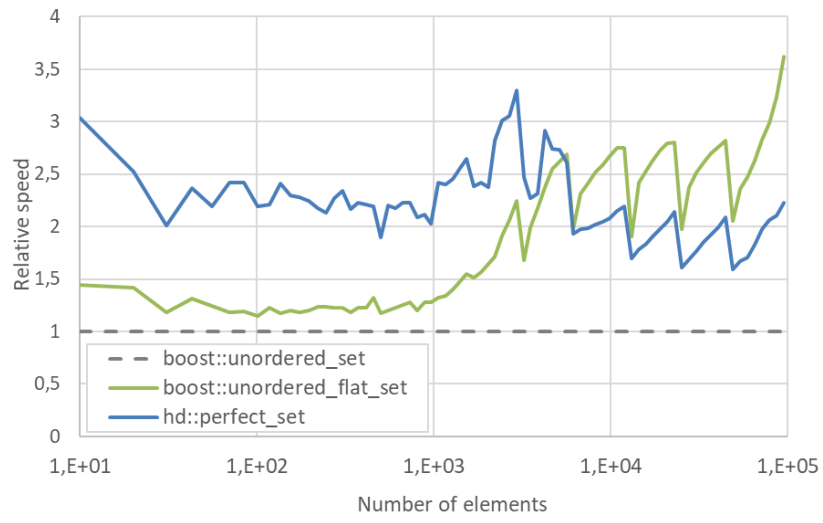


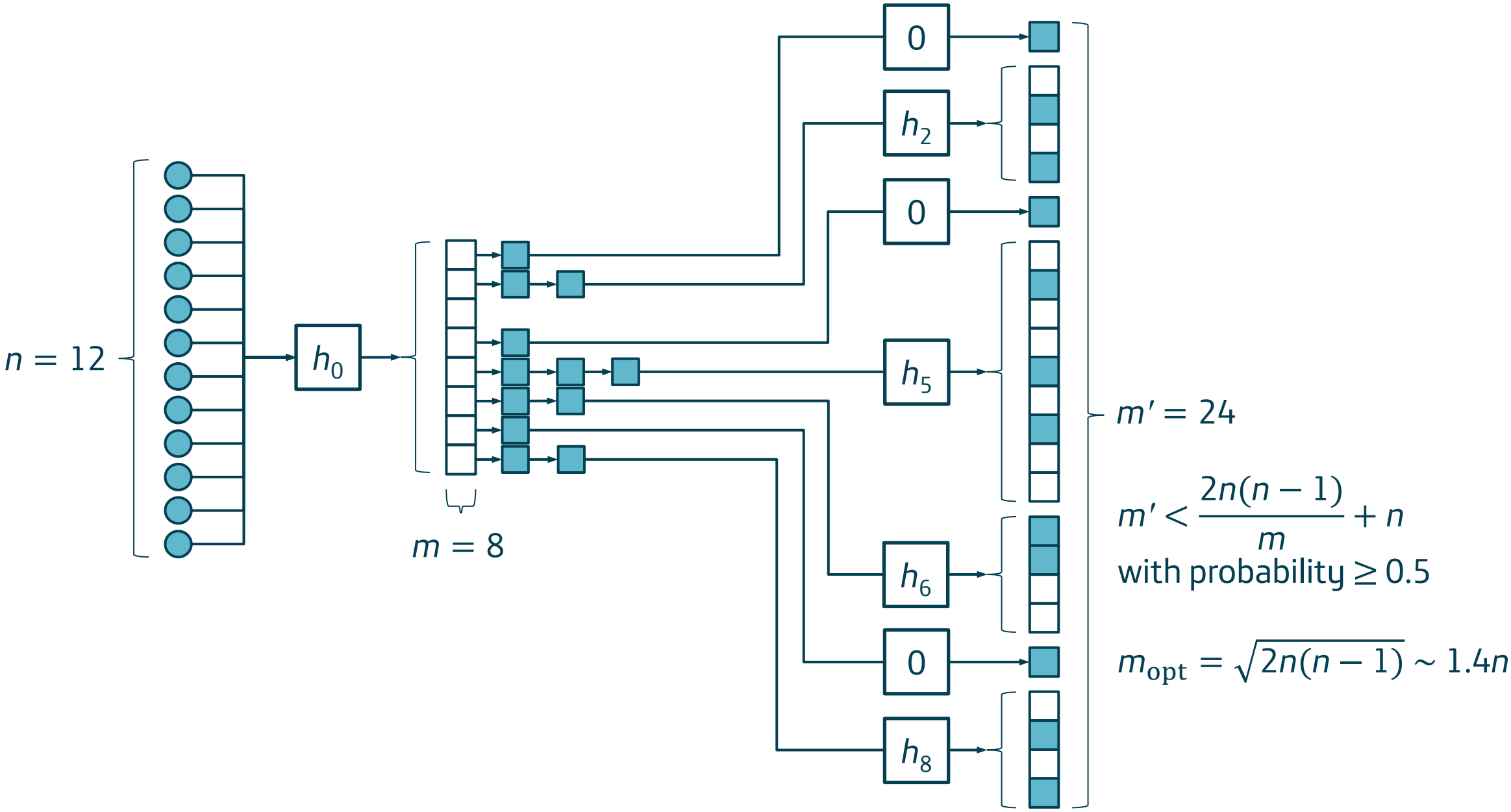



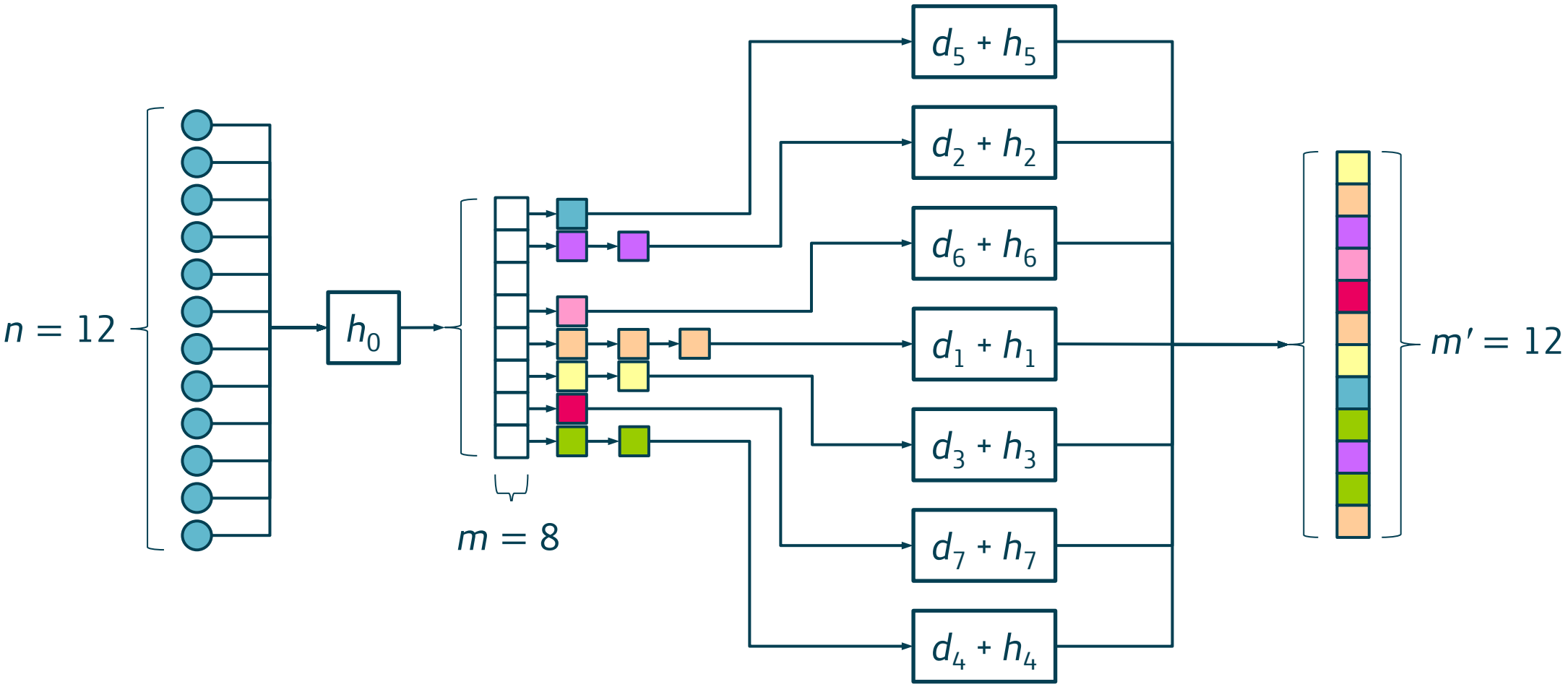



# Techniques

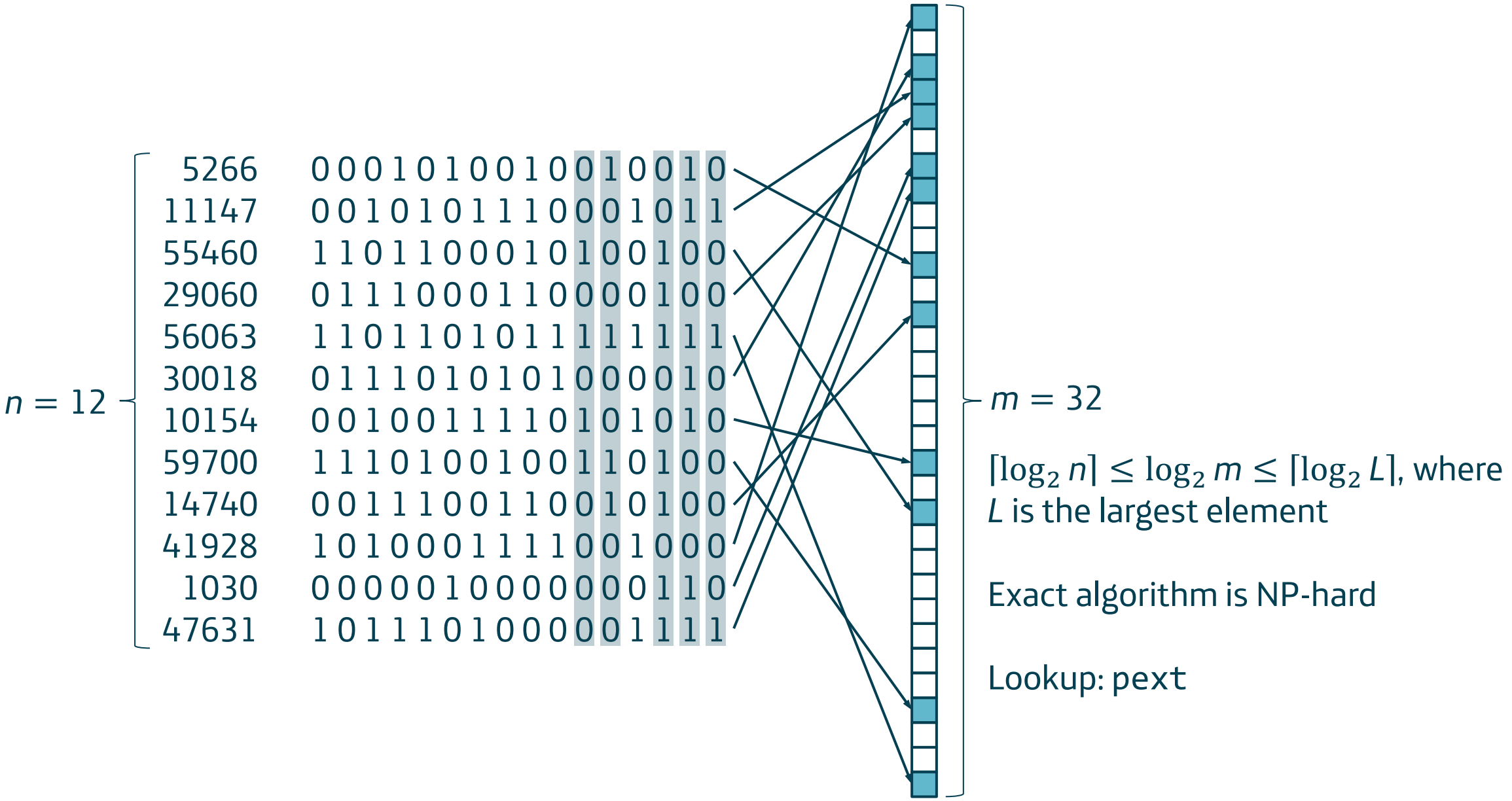

$n = 12$
$h_0$
$m = 8$
0
$h_2$
0
$h_5$
$h_6$
0
$h_8$
$m' = 24$
$m' < \frac{2n(n-1)}{m} + n$
with probability $\geq 0.5$
$m_{\text{opt}} = \sqrt{2n(n-1)} \sim 1.4n$

# HD(C): Hash, Displace (and Compress)





# Minimal cover

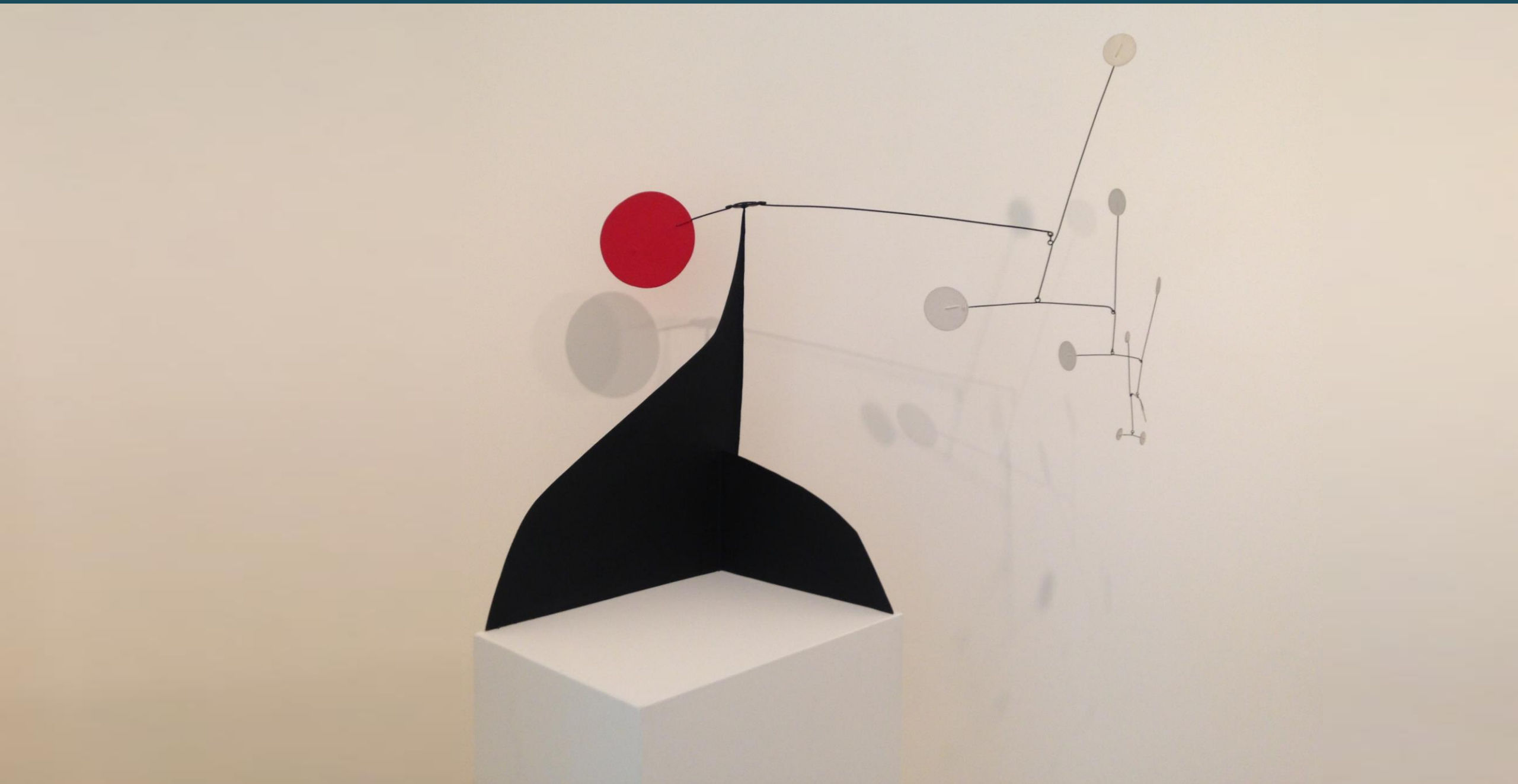



$\lceil \log_2 n \rceil \leq \log_2 m \leq \lceil \log_2 L \rceil$, where $L$ is the largest element

Exact algorithm is NP-hard

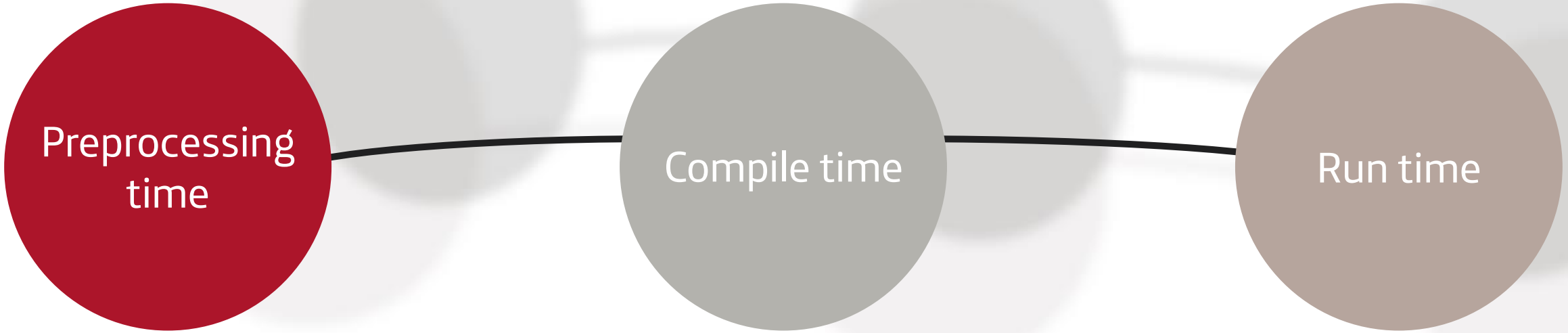
Lookup: `pext`

# Construction time



# When to build a perfect hash table






- External tool → source code
- Works for any C++ standard
- Fast build times
- Complicates project build

- Using `constexpr` magic
- Requires later C++ standards
- Compilation times suffer (can be confined to its own TU)

- Sometimes the only option
- Doesn't require later C++ standards
- Compilation times OK
- Potentially slower?
- What if construction fails?

# Libraries

- Douglas Schmidt 1989-2003
- Preprocessing time, `xx.gperf` → `xx.h` (C or C++ code)
- Only strings, oriented towards keyword recognition
  - Used by GCC in various places
- Non-minimal, plus allows for “near-perfect” hash functions
- $h(s) = \sum_{i=1}^{K}(v[s[p_i] + d_i]) + v\left[s[\text{len}(s) - 1]\right] + \text{len}(s)$
- Tries to minimize *K* (number of string positions checked) and max *h*(*s*)

```
#include <string.h>

#define TOTAL_KEYWORDS 100
#define MIN_WORD_LENGTH 2
#define MAX_WORD_LENGTH 20
#define MIN_HASH_VALUE 2
#define MAX_HASH_VALUE 166
/* maximum key range = 165, duplicates = 0 */

class Perfect_Hash
{
private:
  static inline unsigned int
  hash (const char *str, size_t len);
public:
  static const char *
  in_word_set (const char *str, size_t len);
};
```

```
inline unsigned int
Perfect_Hash::hash (const char *str, size_t len)
{
  static unsigned char asso_values[] =
    {
      167, 167, 167, 167, 167, 167, 167, 167, 167, 167,
      167, 167, 167, 167, 167, 167, 167, 167, 167, 167,
      167, 167, 167, 167, 167, 167, 167, 167, 167, 167,
      167, 167, 167, 167, 167, 167, 167, 167, 167, 167,
      167, 167, 167, 167, 167, 167, 167, 167, 167, 167,
      167,  50,  45, 167, 167, 167, 167, 167, 167, 167,
      167, 167, 167, 167, 167,  80, 167,  10,  45, 167,
      167,  35, 167, 167, 167, 167,  60, 167,  35,  60,
       55,  20,   0,   5,  15,  35,   0, 167, 167,   0,
        5, 167, 167, 167, 167, 167, 167,  20,  10,  10,
      ... /* 256 entries */
    };
  unsigned int hval = len;

  switch (hval)
    {
      default:
        hval +=
          asso_values[static_cast<unsigned char>(str[4])];
      /*FALLTHROUGH*/
      case 4:
        hval +=
          asso_values[static_cast<unsigned char>(str[3])];
      /*FALLTHROUGH*/
      case 3:
      case 2:
        hval +=
          asso_values[static_cast<unsigned char>(str[1])];
      /*FALLTHROUGH*/
      case 1:
        hval +=
          asso_values[static_cast<unsigned char>(str[0])];
        break;
    }
  return hval;
}
```

```
const char *
Perfect_Hash::in_word_set (const char *str, size_t len)
{
  static const char * wordlist[] =
    {
      "", "",
      "pr",
      "", "", "", "", "",
      "sim",
      "pscr",
      "",
      "prurel",
      "ee",
      "Ycy",
      "npar",
      "ltrie",
      "nrtrie",
      "ne",
      "nlE",
      "ltcc",
      "rceil",
      "nltrie",
      "npolint",
      "lnapprox",
      ... /* 167 entries */
    };

  if (len <= MAX_WORD_LENGTH && len >= MIN_WORD_LENGTH)
    {
      unsigned int key = hash (str, len);

      if (key <= MAX_HASH_VALUE)
        {
          const char *s = wordlist[key];

          if (*str == *s && !strcmp (str + 1, s + 1))
            return s;
        }
    }
  return 0;
}
```

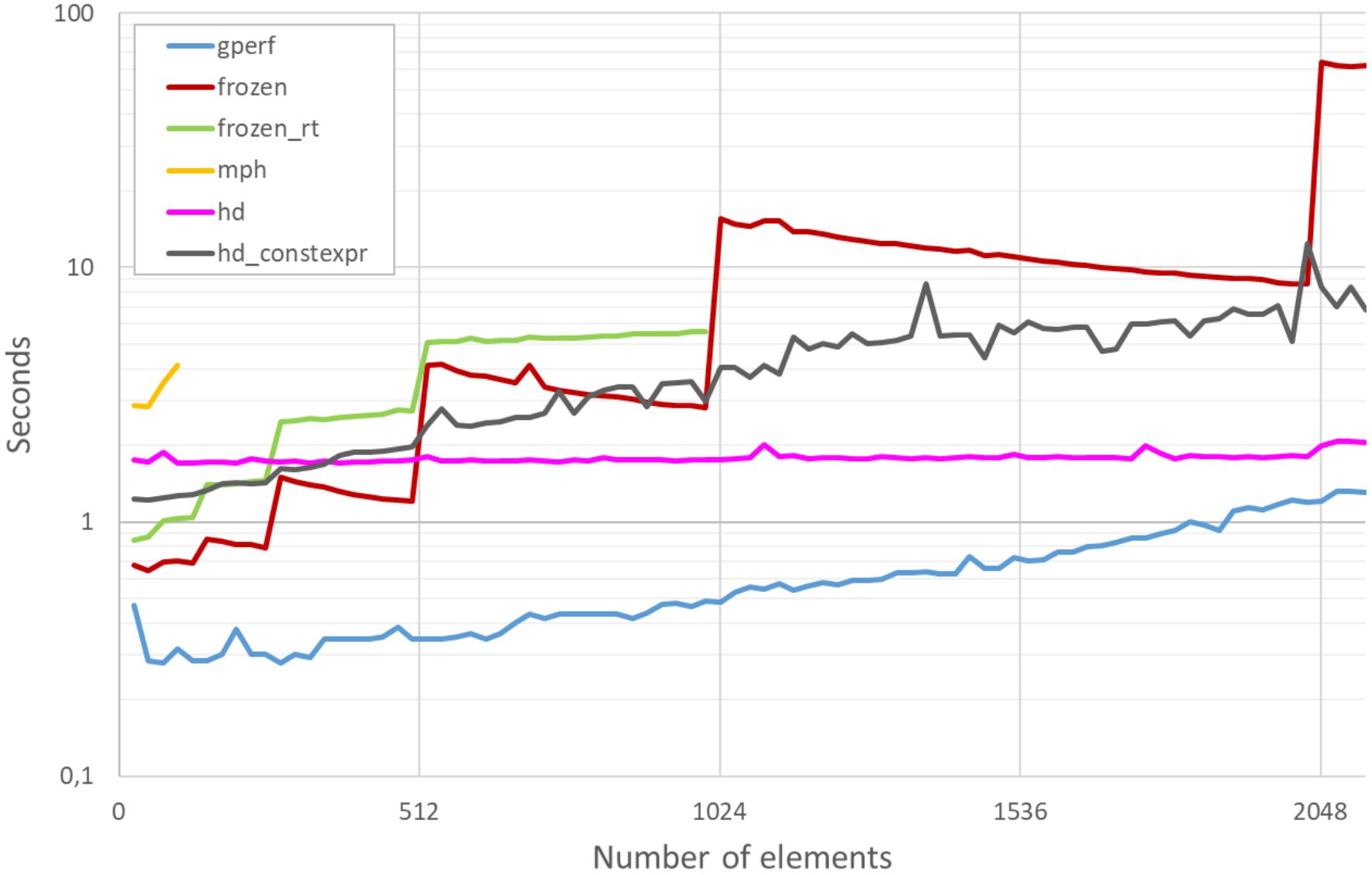
- Serge Guelton et al. 2017-today
- Compile time, `C++14`, sets/maps, map values mutable if non-`constexpr`
  - `-fconstexpr-ops-limit` required save for very small tables
- Works with any key type
  - Off-the-shelf support for integral types (Wang's 64 bit mixing) and `frozen::string` (FNV-1a)
- Non-minimal (*m* is a power of 2)
- Based on HD(C), UHF: $h(x, seed)$
  - Primary array → secondary array → element array

- Kris Jusiak 2024
- Compile time, C++20, function from key to value
  - `-fconstexpr-ops-limit` required save for very small tables
- Supported keys: integral types and strings (max size 8)
- Supported values: integral types
- Algorithms
  - $n < 4$, integrals: generated `switch`
  - $n < 4$, same-sized strings (4 or 8): generated `switch` on SWAR values
  - else: minimal cover + `pext`

- WIP
- `hd::perfect_set`
  - Run time, `C++11`
  - Any key value (`boost::hash` by default)
  - Minimal HD(C), $h_0 = \text{low}(h)$, UHF: $\text{high}(h \cdot m)$
    - Primary array $\rightarrow$ secondary array
- `hd::constexpr_perfect_set`
  - Compile-time version (watch out, `boost::hash` not `constexpr`)
  - `-fconstexpr-ops-limit` required for medium sized tables ($n > 1{,}000$)

# Creation times

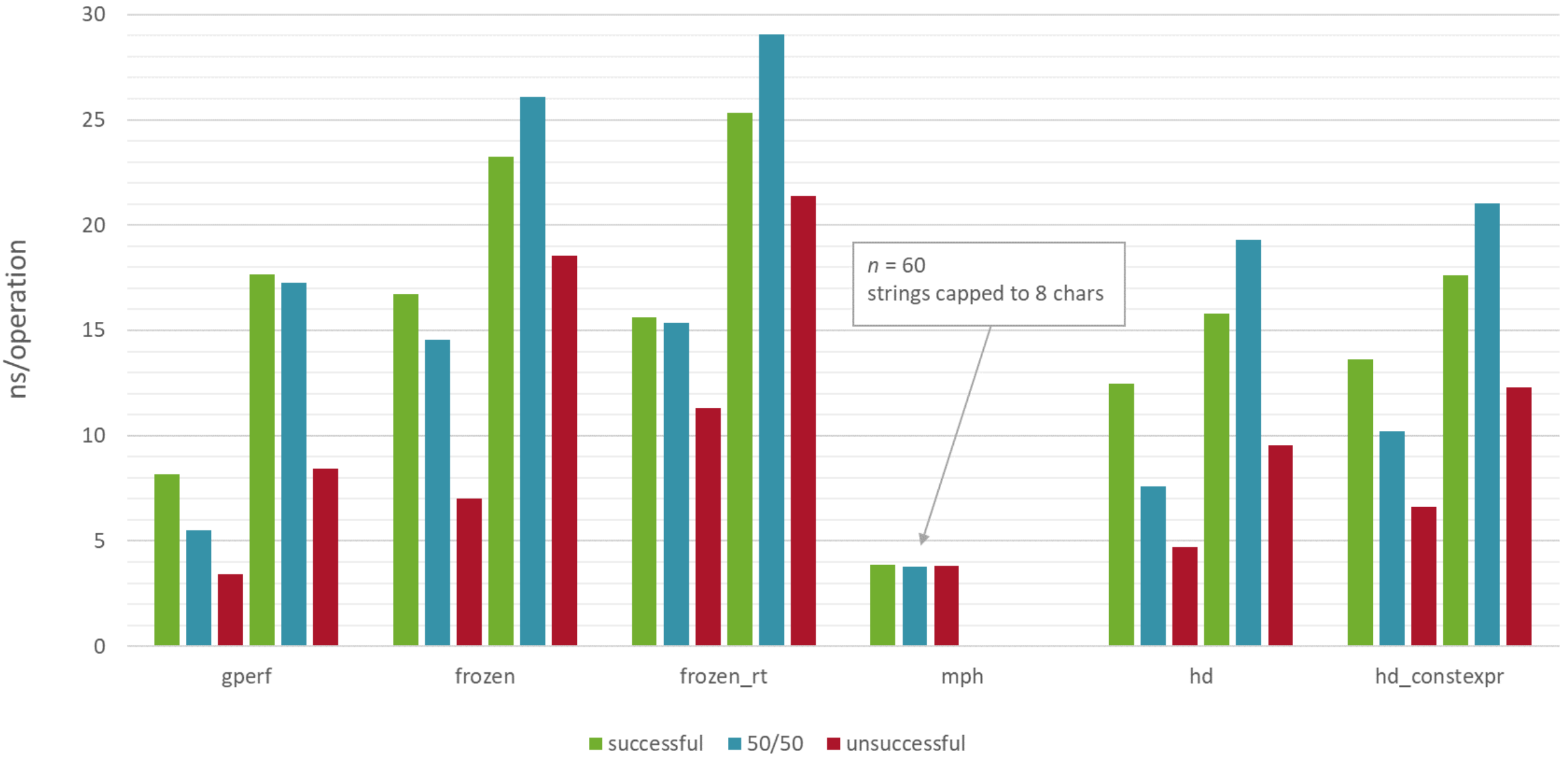





GCC 13.2.0, MSYS2, Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz, 8 GB RAM

# Lookup times



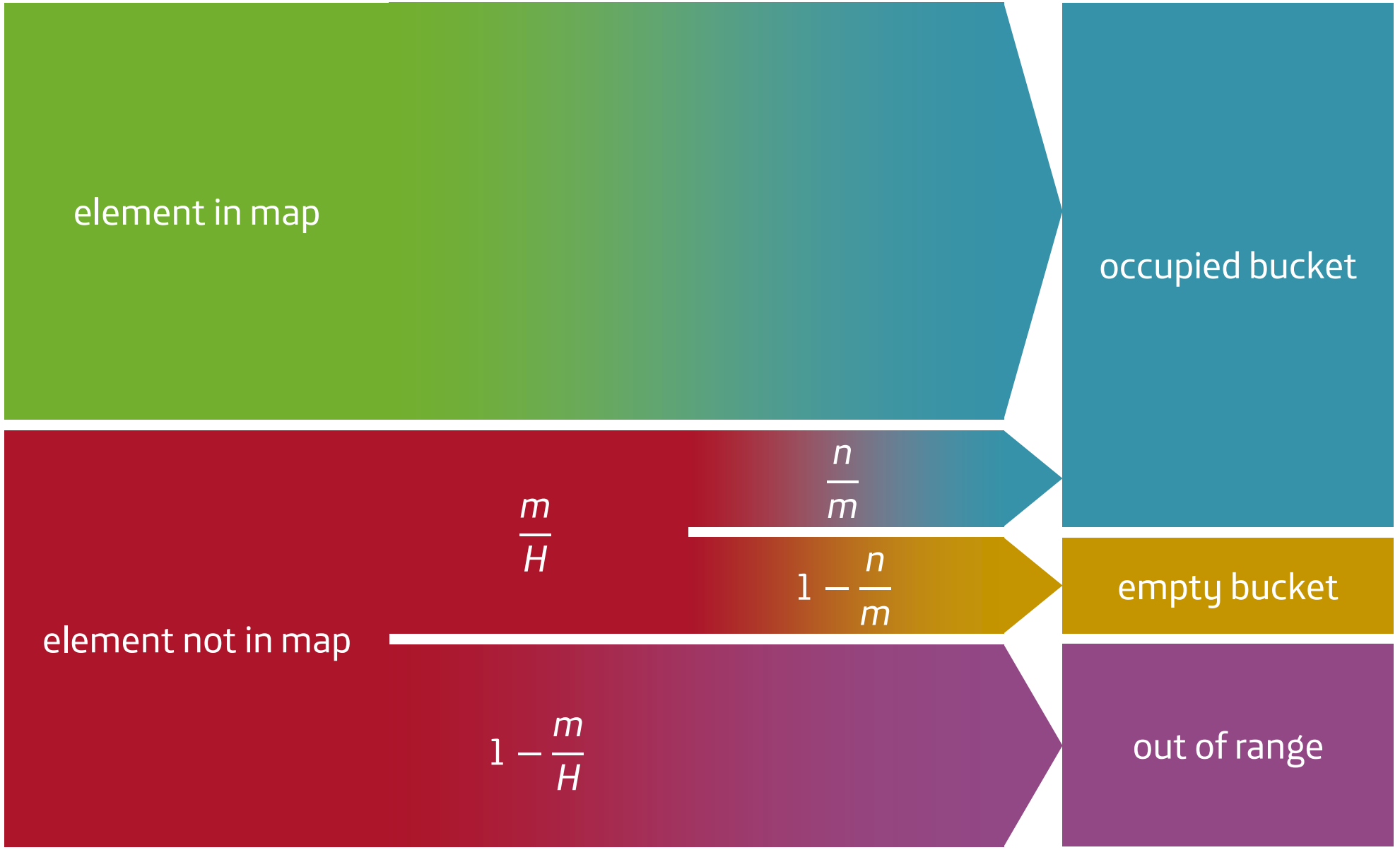



GCC 13.2.0, MSYS2, Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz, 8 GB RAM

❶ `constexpr table<T, N>::table(Input)`

❷ `constexpr table<Input>::table()`

- ❶ is enough for UHF approaches
- ❷ may potentially be faster (particularly with engineered hash functions)
  - But requires structural strings and generates unwieldy symbols
- `constexpr` only allows transient dynamic memory (as of C++23)
  - footprint(`table`) ≠ `sizeof(table)` (❷ can go `static`)
  - footprint(`table`) = $f$(`std::size(Input)`) → OK
  - footprint(`table`) = $f$(`Input`) → double computation


element in map
occupied bucket
$\frac{n}{m}$
$\frac{m}{H}$
$1 - \frac{n}{m}$
empty bucket
element not in map
$1 - \frac{m}{H}$
out of range

# Use cases

# What's perfect hashing good for?



Translation tables
Keyword detection

Update seldom, read many
(e.g. JSON objects)

Faster `switches`

*Database indexing*

https://godbolt.org/z/qfKEhE4bM

```
unsigned y=0;

void foo(unsigned x)
{
  switch(x){
    case 0:  y*=2; break;
    case 1:  y+=7; break;
    case 2:  y*=2; break;
    case 3:  y-=5; break;
    case 4:  y/=4; break;
    default:       break;
  }
}
```

```
foo(unsigned int):
        cmp     edi, 4
        ja      .L1
        mov     edi, edi
        jmp     [QWORD PTR .L4[0+rdi*8]]
.L4:
        .quad   .L6
        .quad   .L7
        .quad   .L6
        .quad   .L5
        .quad   .L3
.L3:
        shr     DWORD PTR y[rip], 2
.L1:
        ret
.L6:
        sal     DWORD PTR y[rip]
        ret
.L7:
        add     DWORD PTR y[rip], 7
        ret
.L5:
        sub     DWORD PTR y[rip], 5
        ret
y:
        .zero   4
```

# Sparse `switch`

https://godbolt.org/z/Y79csGve3

```
unsigned y=0;

void foo(unsigned x)
{
  switch(x){
    case 0:  y*=2; break;
    case 2:  y+=7; break;
    case 3:  y*=2; break;
    case 10: y-=5; break;
    case 11: y/=4; break;
    default:       break;
  }
}
```

```
foo(unsigned int):
        cmp     edi, 11
        ja      .L1
        mov     edi, edi
        jmp     [QWORD PTR .L4[0+rdi*8]]
.L4:
        .quad   .L6
        .quad   .L1
        .quad   .L7
        .quad   .L6
        .quad   .L1
        .quad   .L1
        .quad   .L1
        .quad   .L1
        .quad   .L1
        .quad   .L1
        .quad   .L1
        .quad   .L5
        .quad   .L3
.L3:
        shr     DWORD PTR y[rip], 2
.L1:
        ret
.L6:
        sal     DWORD PTR y[rip]
        ret
.L7:
        add     DWORD PTR y[rip], 7
        ret
```

```
.L5:
        sub     DWORD PTR y[rip], 5
        ret
y:
        .zero   4
```

# Sparser `switch`

https://godbolt.org/z/4K6WqWdv9

```
unsigned y=0;

void foo(unsigned x)
{
  switch(x){
    case 10: y*=2; break;
    case 21: y+=7; break;
    case 32: y*=2; break;
    case 49: y-=5; break;
    case 52: y/=4; break;
    default:       break;
  }
}
```

```
foo(unsigned int):
        cmp     edi, 32
        je      .L2
        jbe     .L15
        cmp     edi, 49
        je      .L6
        cmp     edi, 52
        jne     .L16
        shr     DWORD PTR y[rip], 2
        ret
.L15:
        cmp     edi, 10
        je      .L2
        cmp     edi, 21
        jne     .L17
        add     DWORD PTR y[rip], 7
        ret
.L2:
        sal     DWORD PTR y[rip]
        ret
.L17:
        ret
.L16:
        ret
.L6:
        sub     DWORD PTR y[rip], 5
        ret
```

```
y:
        .zero   4
```

# Outsmarting the compiler



https://godbolt.org/z/9sKdbhhrc

```
constexpr std::pair<int,int> bs[] = {
  {32,2}, {49,3}, {10,0}, {-1,5},
  {52,4}, {21,1}, {-1,5}, {-1,5}
};

inline int ph(unsigned x)
{
  auto b = bs[x&7];
  return x==b.first? b.second: 5;
}

unsigned y=0;

void foo(unsigned x)
{
  switch(ph(x)){
    case 0:  y*=2; break;
    case 1:  y+=7; break;
    case 2:  y*=2; break;
    case 3:  y-=5; break;
    case 4:  y/=4; break;
    case 5:        break;
    default:       std::unreachable();
  }
}
```

```
foo(unsigned int):
        mov     eax, edi
        and     eax, 7
        mov     edx, DWORD PTR bs[4+rax*8]
        cmp     edi, DWORD PTR bs[0+rax*8]
        je      .L10
.L1:
        ret
.L10:
        jmp     [QWORD PTR .L4[0+rdx*8]]
.L4:
        .quad   .L7
        .quad   .L8
        .quad   .L7
        .quad   .L6
        .quad   .L5
        .quad   .L1
.L7:
        sal     DWORD PTR y[rip]
        ret
.L5:
        shr     DWORD PTR y[rip], 2
        ret
.L6:
        sub     DWORD PTR y[rip], 5
        ret
```

```
.L8:
        add     DWORD PTR y[rip], 7
        ret
y:
        .zero   4
bs:
        .long   32
        .long   2
        .long   49
        .long   3
        .long   10
        .long   0
        .long   -1
        .long   5
        .long   52
        .long   4
        .long   21
        .long   1
        .long   -1
        .long   5
        .long   -1
        .long   5
```

# No string `switches`



https://godbolt.org/z/zcrGhWh6j

```
unsigned y=0;

void foo(std::string_view x)
{
        if(x=="0") y*=2;
  else if(x=="1") y+=7;
  else if(x=="2") y*=2;
  else if(x=="3") y-=5;
  else if(x=="4") y/=4;
}
```

```
foo(std::string_view):
        cmp     rdi, 1
        je      .L11
.L1:
        ret
.L11:
        movzx   eax, BYTE PTR [rsi]
        cmp     al, 48
        jne     .L3
.L9:
        sal     DWORD PTR y[rip]
        ret
.L3:
        cmp     al, 49
        jne     .L5
        add     DWORD PTR y[rip], 7
        ret
.L5:
        cmp     al, 50
        je      .L9
        cmp     al, 51
        jne     .L7
        sub     DWORD PTR y[rip], 5
        ret
```

```
.L7:
        cmp     al, 52
        jne     .L1
        shr     DWORD PTR y[rip], 2
        ret
y:
        .zero   4
```

# String sequential check → packed `switch`



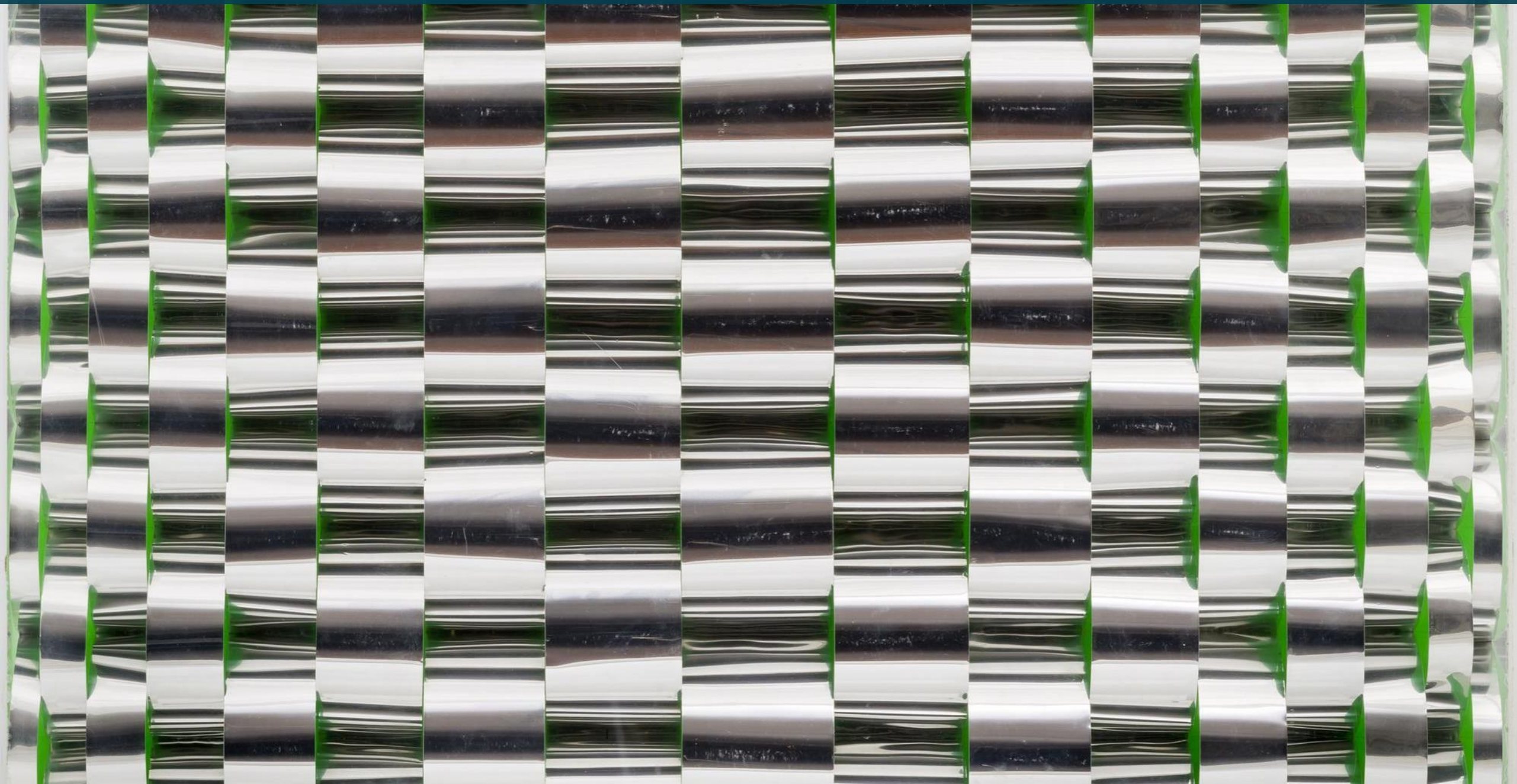
https://godbolt.org/z/4njjj4cbh

```
inline int ph(std::string_view x)
{
  return x.size()==1?
    x[0]-'0': 5;
}

unsigned y=0;

void foo(std::string_view x)
{
  switch(ph(x)){
    case 0:  y*=2; break;
    case 1:  y+=7; break;
    case 2:  y*=2; break;
    case 3:  y-=5; break;
    case 4:  y/=4; break;
    default:       break;
  }
}
```

```
foo(std::string_view):
        cmp     rdi, 1
        je      .L9
.L1:
        ret
.L9:
        movsx   eax, BYTE PTR [rsi]
        sub     eax, 48
        cmp     eax, 4
        ja      .L1
        jmp     [QWORD PTR .L4[0+rax*8]]
.L4:
        .quad   .L6
        .quad   .L7
        .quad   .L6
        .quad   .L5
        .quad   .L3
.L6:
        sal     DWORD PTR y[rip]
        ret
.L3:
        shr     DWORD PTR y[rip], 2
        ret
.L5:
        sub     DWORD PTR y[rip], 5
        ret
```

```
.L7:
        add     DWORD PTR y[rip], 7
        ret
y:
        .zero   4
```

# Conclusions



# Conclusions

- Key insight: perfection achieved through immutability
  - Perfect hashing covers a narrower problem space than regular hash tables
- Two algorithmic approaches: UHF, engineered hash function
- Three construction strategies: preprocessing time, compile time, run time
  - Run time has received very little attention so far
- `gperf` is a really good tool
- Compile-time constructs are still subpar
  - Transient `constexpr` dynamic memory allocation (P2670R1)
  - Complex code generation still difficult
- Stay tuned to #boost-unordered in cpplang.slack.com

# Perfect Hashing in an Imperfect World

## Thank you

github.com/joaquintides/usingstdcpp2024



`using std::cpp 2024`

Joaquín M López Muñoz <joaquin.lopezmunoz@gmail.com>

Madrid, April 2024