

ICALP presentation, July 12, 1999

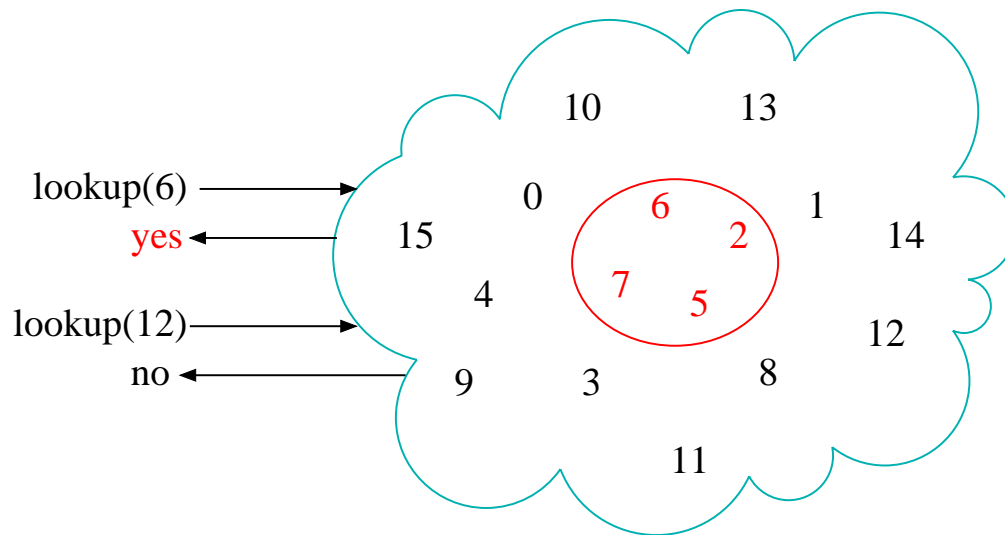
Low redundancy in static dictionaries with $O(1)$ lookup time

Rasmus Pagh

 **BRICS**, Aarhus University

The problem

A static dictionary stores a subset S of a finite universe U .

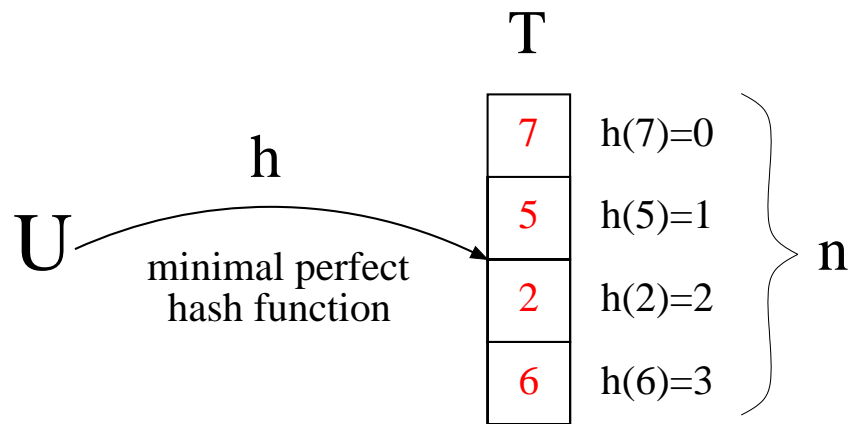


Model: Unit cost RAM with multiplication, word size w . $U = \{0, \dots, 2^w - 1\}$.

Bit vectors of length 2^w allow constant time lookup.

Can we achieve this using less memory (as a function of $n = |S|$ and w)?

Dictionary using minimal perfect hashing



```
proc lookup(x)
  return (T[h(x)]=x);
end
```

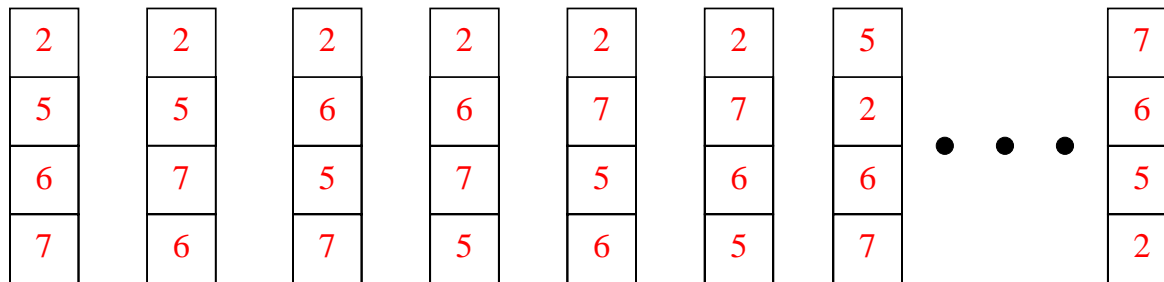
The [minimal perfect hash functions](#) of Fredman, Komlós & Szemerédi (1982):

Evaluation time: $O(1)$.

Space usage: $o(n)$ machine words.

Redundancy in the hash table

There are $n!$ ways of arranging the elements of S in a table:



So a table uses at least $\log_2(n!) = n \log_2 n - \Theta(n)$ bits more than optimum.

Example: The set of $n \approx 5,400,000$ Danish personal identity numbers.

- Table size: $25n$ bits.
- Table redundancy: $\log_2(n!) > 20n$ bits.

The information theoretical minimum

To represent any subset of U of size n we need at least

$$B = \lceil \log_2 \binom{2^w}{n} \rceil = n (w - \log_2 n + \Theta(1)) \text{ bits.}$$

How close can we get to B bits of space without sacrificing lookup efficiency?

Previous work

The dictionary of Brodnik & Munro (1994):

Lookup time: Worst case $O(1)$.

Space usage: $O(B)$ bits — later improved to $B + O(B / \log^{(3)} B)$ bits.

Approach:

Two-level splitting into subranges which are stored using either

- A pointer to a “table of small ranges” (dense subsets), or
- Non-oblivious hashing (sparse subsets).

Main result

A static dictionary with:

Lookup time: Worst case $O(1)$.

Space usage: $B + o(n) + O(\log w)$ bits.

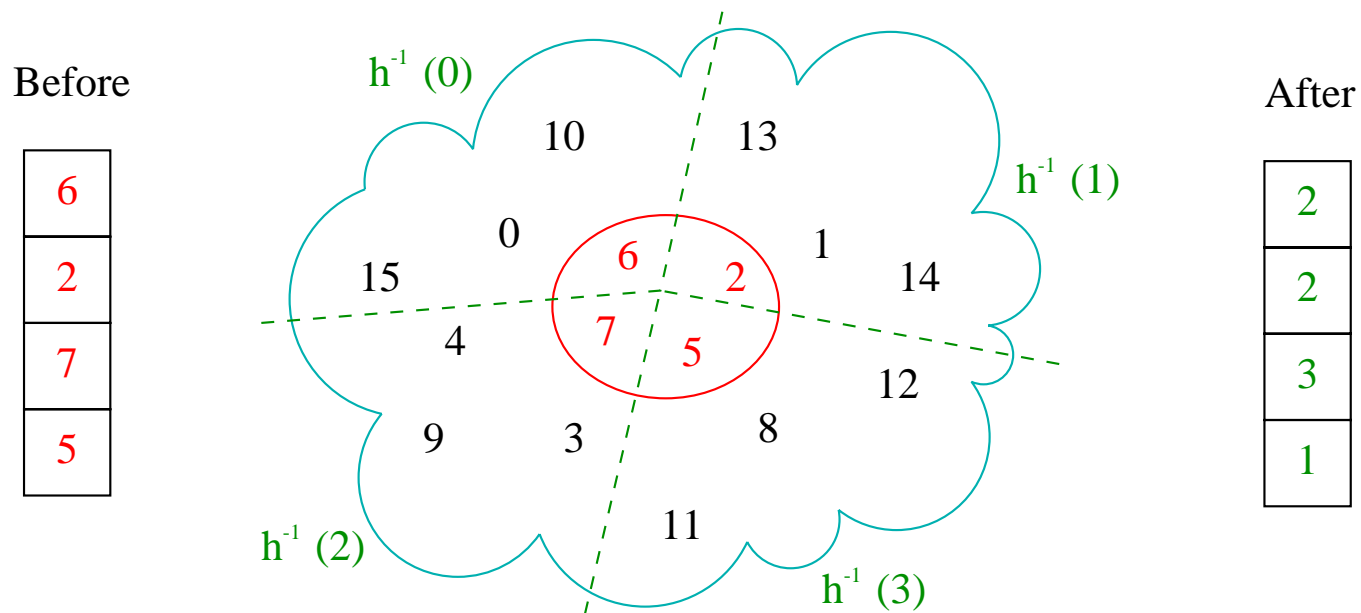
Redundancy reduction factor:

Non-sparse sets: $\log w (\log n)^{1-o(1)}$.

Sparse sets: $\Theta(n)$.

A simple observation on saving space in hash tables

For a fixed hash function, only certain values are possible in each hash table cell.



Each hash table element can be stored *relative* to a set of size $\approx 2^w / n$.

About $\log_2 n$ bits are saved for each element!

Quotient functions

A **quotient function** q (of a hash function h) maps from explicit to “compressed” form.

$x \mapsto (h(x), q(x))$ must be 1-1.

Using $T' = [q(T[0]), q(T[1]), \dots, q(T[n-1])]$:

```
proc lookup'(x)
  return (T'[h(x)] = q(x));
end
```

Example: $h(x) = (kx \bmod p) \bmod n$, $q(x) = (x \operatorname{div} p, (kx \bmod p) \operatorname{div} n)$.

A fast, space-efficient minimal perfect hashing scheme

The minimal perfect hash functions of Schmidt & Siegel (1990):

Evaluation time: $O(1)$.

Space usage: $O(n + \log w)$ bits (optimal).

Corresponding **quotient function**:

Evaluation time: $O(1)$.

Space usage: No extra space needed.

Range: $\{0, \dots, r\}$, where $r = O(2^w/n)$.

Putting things together

We have seen the ingredients of a static dictionary with $O(1)$ worst case lookup time, and space usage:

Hash function: $O(n + \log w)$ bits.

Hash table: $n (w - \log_2 n + O(1))$ bits.

Total: $B + O(n + \log w)$ bits.

Bottlenecks

Minimal perfect hash function: The representation is $\Omega(n)$ bit redundant.

Hash table: The quotient function values generally don't span all bit combinations.

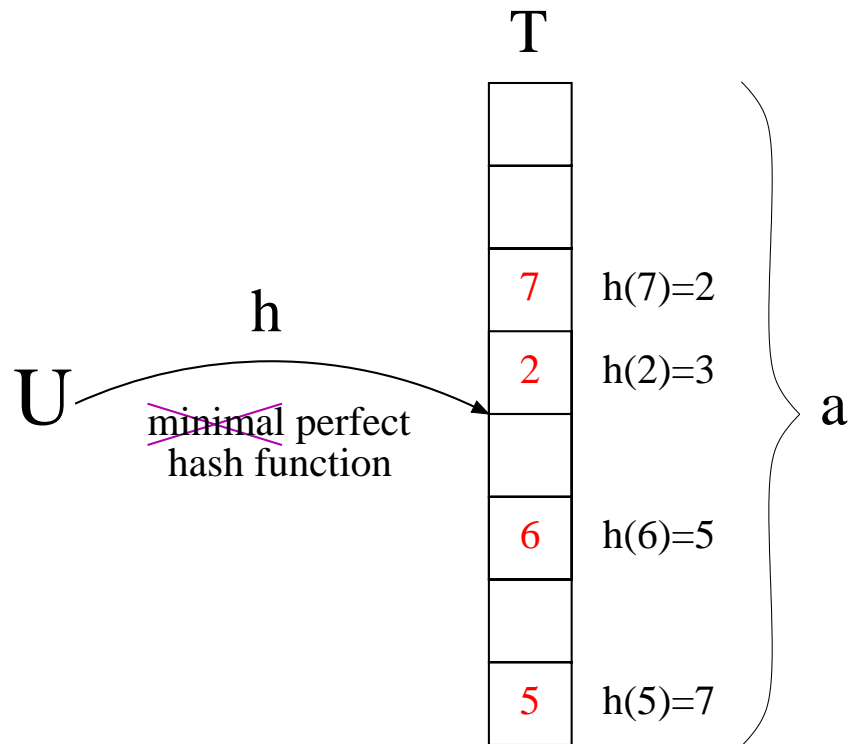


Can we

- Avoid the use of minimal perfect hashing?
- Make sure that $|q[U]|$ is close to (but not higher than) a power of two?

Abandoning minimality

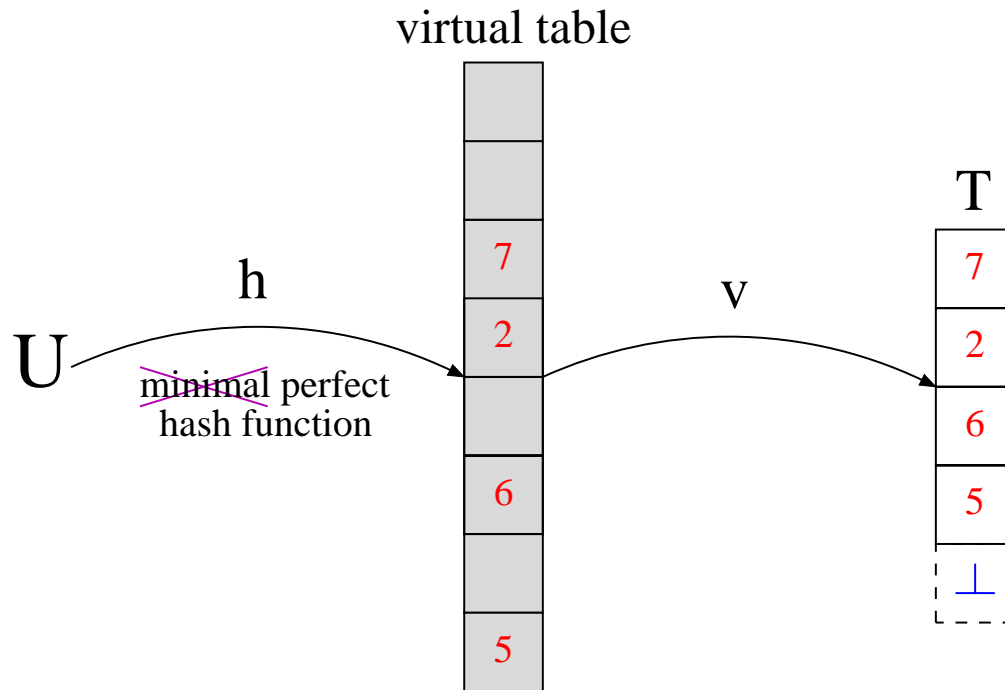
Perfect hash functions with larger range consume less memory . . .



. . . and $|q[U]|$ can be adjusted by changing the range slightly.

The virtual hash table

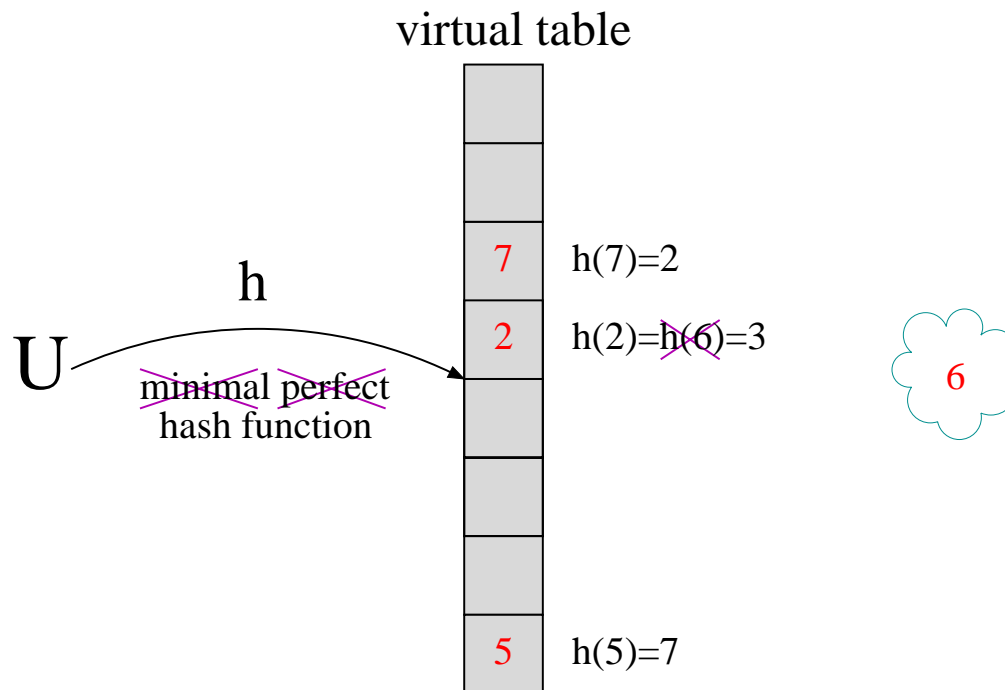
To cope with holes in the table, we introduce a partial function v , defined and 1-1 on the non-empty table entries.



For virtual table size $a = n (\log n)^{O(1)}$ there is an efficient data structure for v .

Abandoning perfection

Universal hash functions are *nearly perfect* when the range is large.



The few colliding elements can be stored using a less efficient method.

Putting things together — tighter

Let n_1 denote the number of elements in the “primary” dictionary.

Hash function: $O(\log n + \log w)$ bits.

Virtual table: $n_1 \log_2\left(\frac{ea}{n}\right) + O\left(\frac{n \log^2 \log n}{\log n}\right)$ bits.

Hash table: $n_1 \log_2\left(\frac{2^w}{a}\right) + o(-)$ bits.

Dictionary 2: $(n - n_1) \log_2\left(\frac{e2^w}{n}\right) + o(-)$ bits.

Total: $n \log_2\left(\frac{e2^w}{n}\right) + O\left(\frac{n \log^2 \log n}{\log n} + \log w\right)$
 $= B + o(n) + O(\log w)$ bits.

Conclusion and open problems

We have seen that:

- Using a **quotient function** one can save $\approx n \log_2 n$ bits in a hash table.
- *Vanishing* redundancy per element in a dictionary can be achieved.

It would be interesting to:

- See *any* lower bound.
- Improve the virtual table mapping.