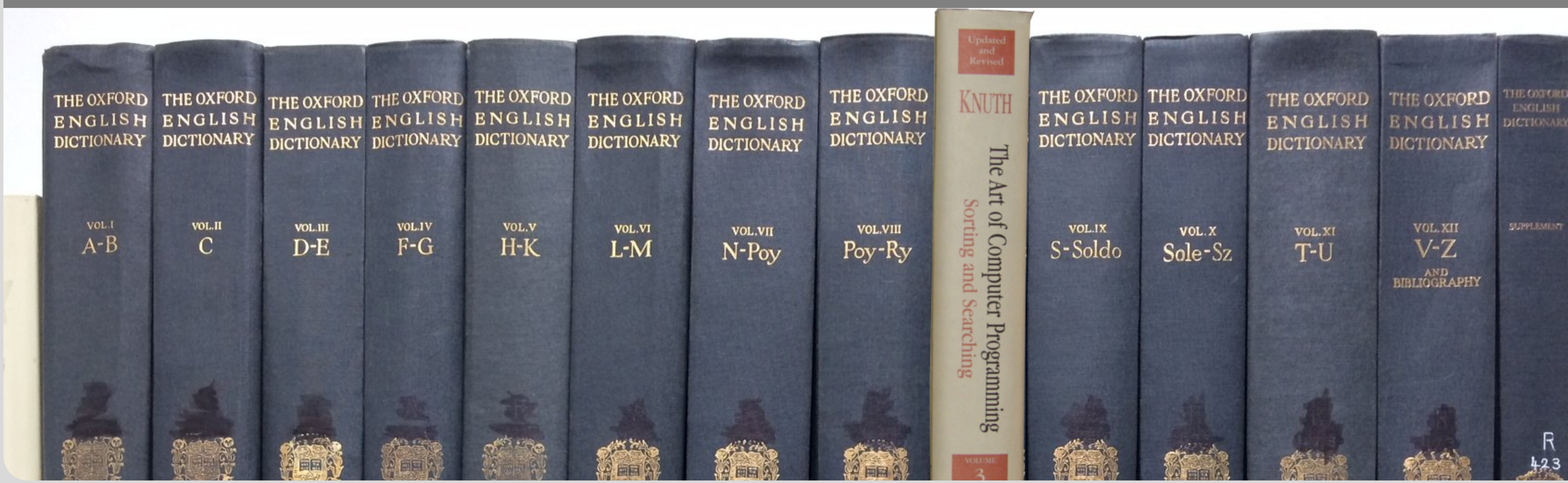
Karlsruhe Institute of Technology

Flexible Hash Table Implementations for Near Drop in Replacement

Presentation · 07. September 2017
Tobias Maier

Institute of Theoretical Informatics · Algorithmics Group



Possible Topics/Focuses

- interface construction
- collection of small tidbits
  - increasing memory inplace (2 methods)
  - removing contention from shared variables
- more on DySECT (the paper presented at ESA)

Institute of Theoretical Informatics
Algorithmics Group

Interface – Main Ideas

- user expectation
  - similar library functions
  - interface used in literature
- possible problems (analyzing an interface)
  - more powerful interface
  - ease of implementation
  - misusability

Interface – Functionality and Performance

- update operation

```
auto it = table.find(k);
if (it != end) it->second += 5;
```

- using dedicated update

```
bool b = table.update(k,
                      [](value_type& cur) {
                          cur+=5;
                      });
```


Institute of Theoretical Informatics
Algorithmics Group

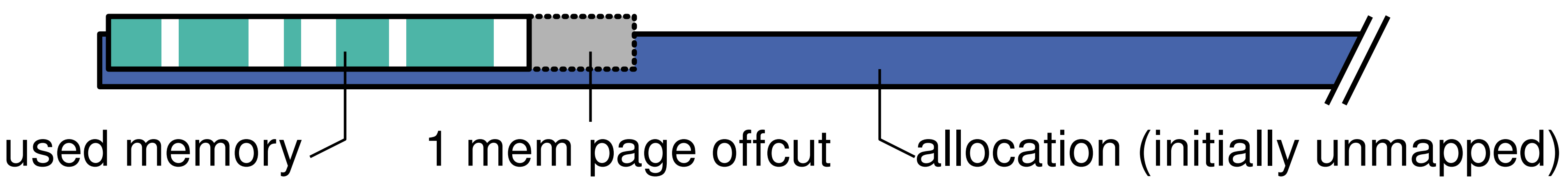
Interface – Misusability

- iterator as return type
  - iterators give pointers
  - moving elements can lead to errors
- operations invalidate iterators
  - refresh iterators
- change iterators to not return pointers

Institute of Theoretical Informatics
Algorithmics Group

Overallocation – For Inplace Resize

- very large virtual memory allocation
- only changed cells are mapped to physical memory
- writing to virtual memory $\approx$ increasing local allocation
  - ⊞ inplace growing
  - ⊟ limited portability



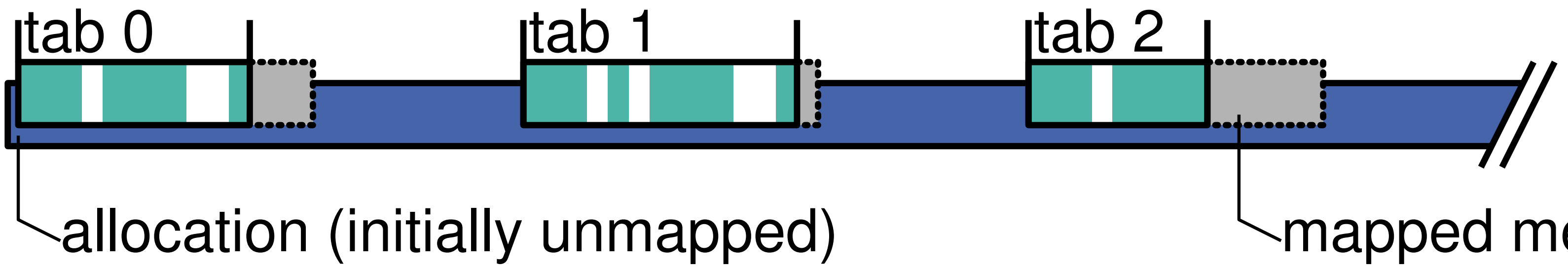
Overallocation – Multi Table Inplace Resize

- subtables are islands of physical memory in a virtual allocation
- equally space tables in virtual memory

  $tab_i \ @ \ M/T \cdot i$

  M = *overalloc size*
  T = *number of subtables*

  - no explicit indirection



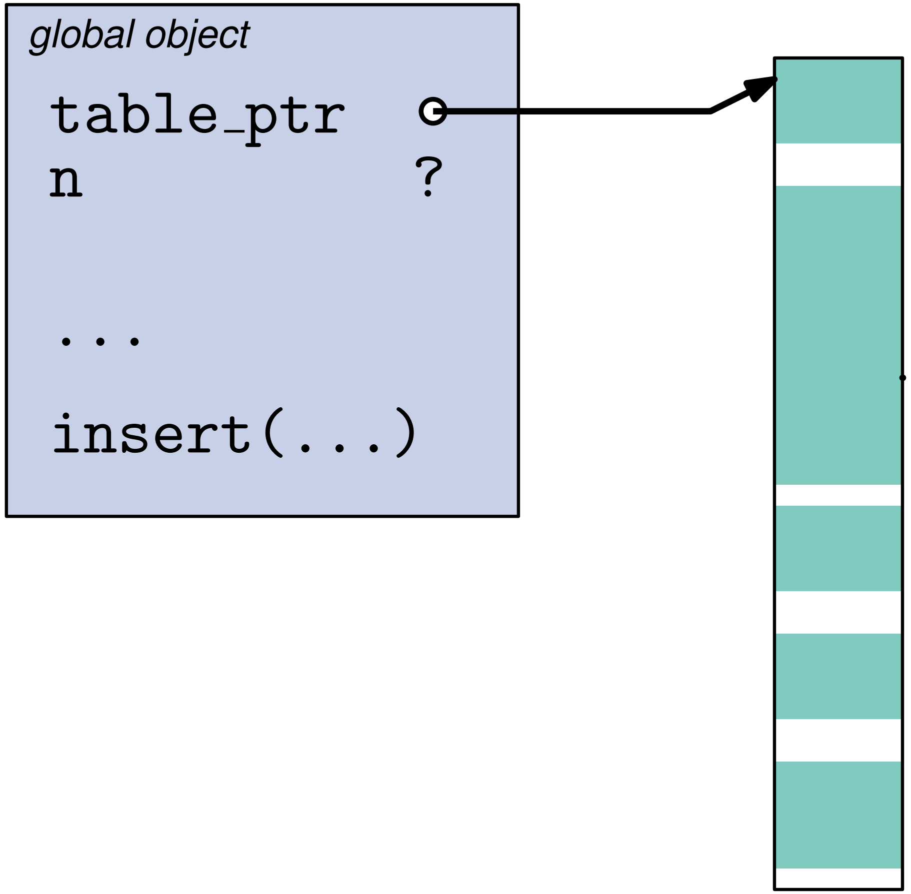
Parallelization – Some Initial Thoughts

- ⊞ Parallele Datenstrukturen für Informationsaustausch
- ⊞ Große Tabelle $\Rightarrow$ viele unabhängige Zugriffe
- ⊟ Wachsende Tabellen benötigen Koordination
  - ⊟ Häufiger Zugriff auf die selben Variablen
  - ⊞ häufig vermeidbar durch lokale Duplizierung
- ⊟ Beschränkt durch Speicherbandbreite

Institute of Theoretical Informatics
Algorithmics Group

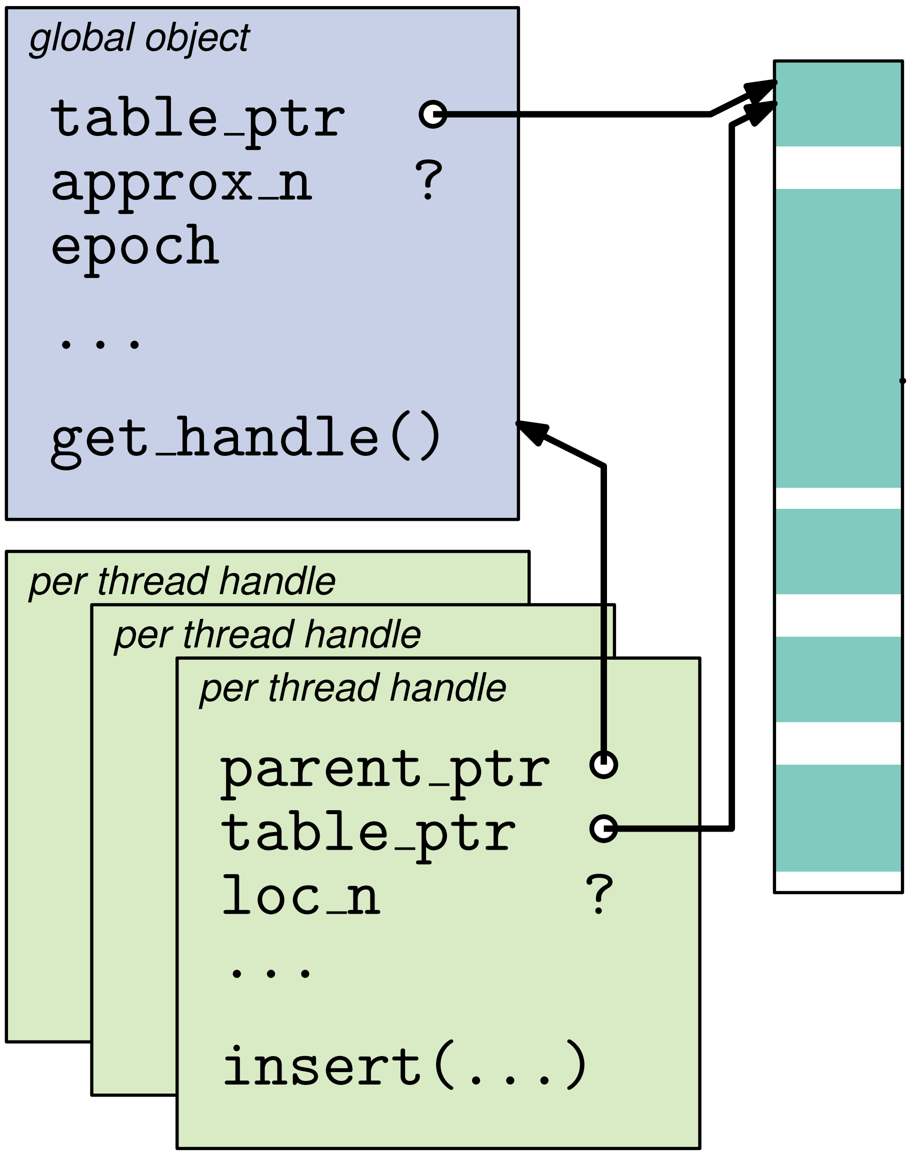
Parallelization – Optimizing Often Used Members

- `table_ptr` is smart
  - ➨ copying is expensive
- `n` has contentious updates
  - ➨ exact value is changing



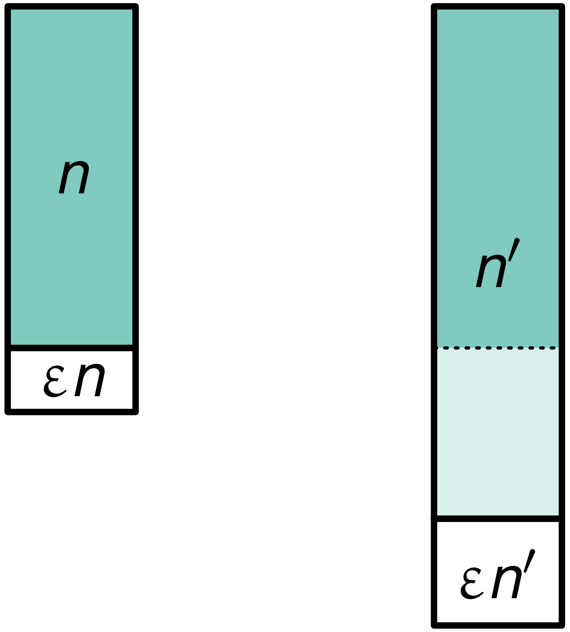
Parallelization – Optimizing Often Used Members

- `table_ptr` is smart
  - copying is expensive
  - cache the pointer copy
- `n` has contentious updates
  - exact value is changing
  - update `approx_n` with local counts
- move interface to handle to avoid misusability



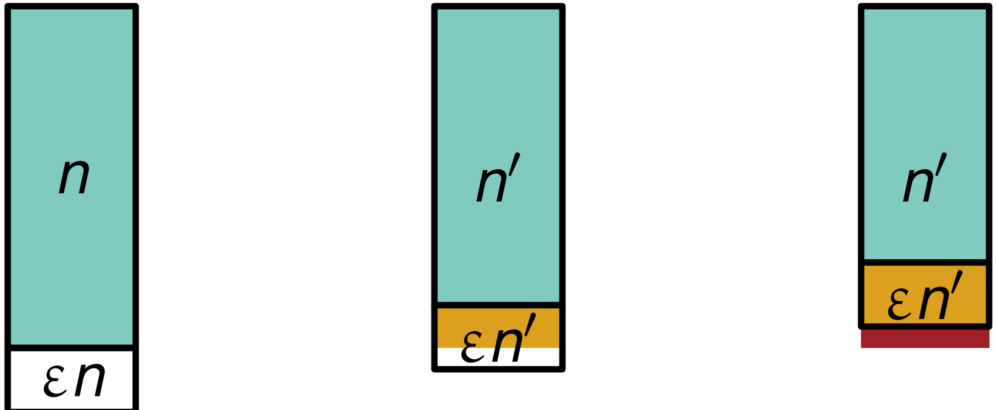
Final Size Not Known A Priori

- conservative estimate $n \leq n'$
  - strict bound might not be reasonable
  - less space efficient



Final Size Not Known A Priori

- conservative estimate
- optimistic estimate $n \approx n'$
  - might overfill
  - needs growing strategy



slow needs growing

Final Size Not Known A Priori

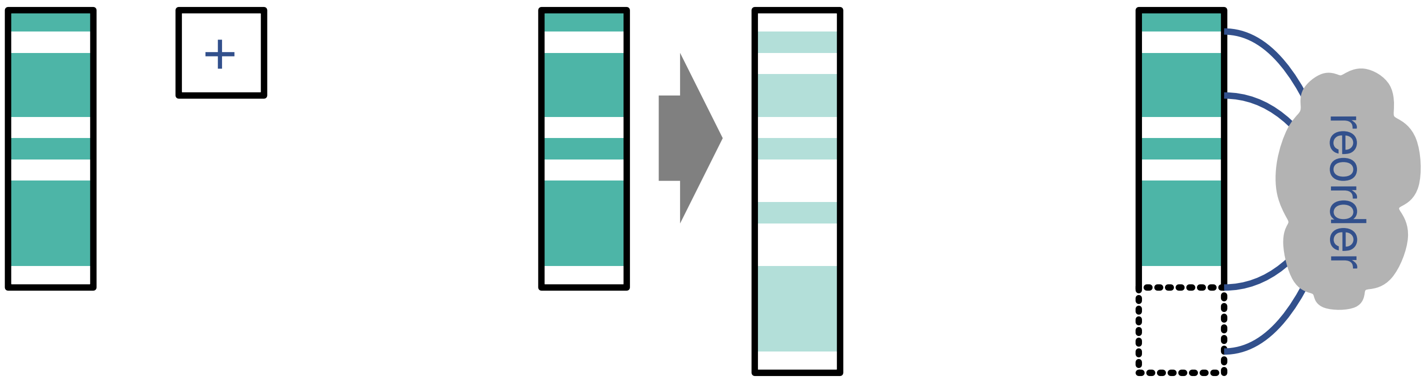
Karlsruhe Institute of Technology

- conservative estimate
- optimistic estimate
- number of elements changes over time
  - ➡ cannot be initialized with max size

Resizing

- growing has to be in small steps
- basic approaches

additional table | full migration | inplace+reorder



most common
in libraries

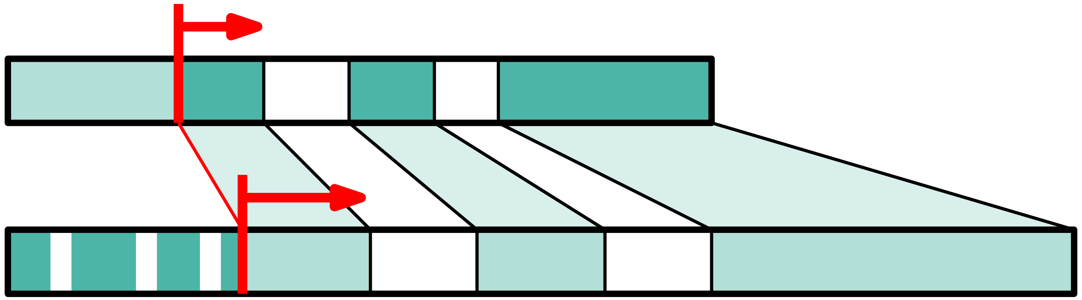
Institute of Theoretical Informatics
Algorithmics Group

Secondary Contribution – Efficient Growing

- addressing the table (no powers of two)
  - conventional wisdom: modulo table size
  - faster: use hash value as scaling factor

$$idx(k) = h(k) \cdot \frac{size}{maxHash + 1}$$

- very fast migration due to cache efficiency

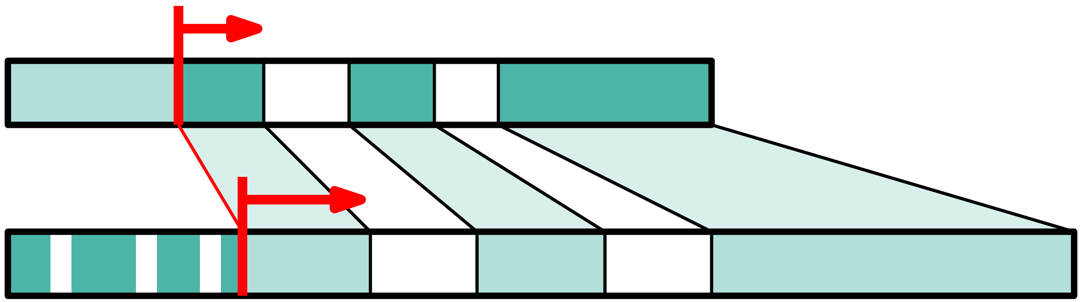


Secondary Contribution – Efficient Growing

- addressing the table (no powers of two)
  - ~~conventional wisdom: modulo table size~~
  - faster: use hash value as scaling factor

$$idx(k) = h(k) \cdot \frac{size}{maxHash + 1}$$

- very fast migration due to cache efficiency



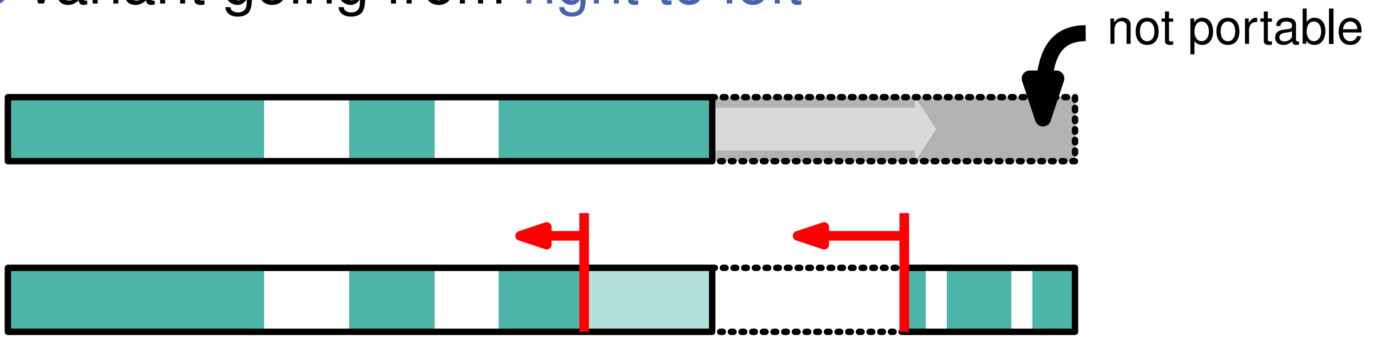
Institute of Theoretical Informatics
Algorithmics Group

Secondary Contribution – Efficient Growing

- addressing the table (no powers of two)
  - ~~conventional wisdom: modulo table size~~
  - faster: use hash value as scaling factor

$$idx(k) = h(k) \cdot \frac{size}{maxHash + 1}$$

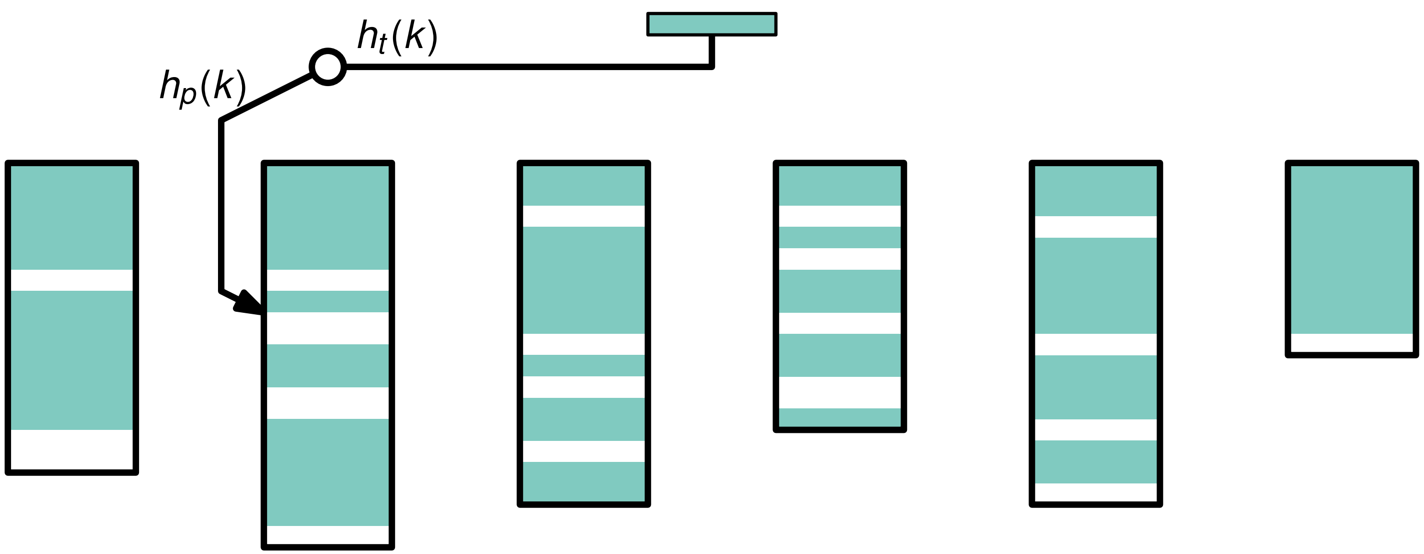
- very fast migration due to cache efficiency
- inplace variant going from right to left



Institute of Theoretical Informatics
Algorithms Group

Multi Table Approach

- $T = 2^c$ subtables with expected equal count
  - ➨ reduces memory during subtable migration
- $h(k) \Rightarrow h_t(k)$ for the subtable $h_p(k)$ within the table

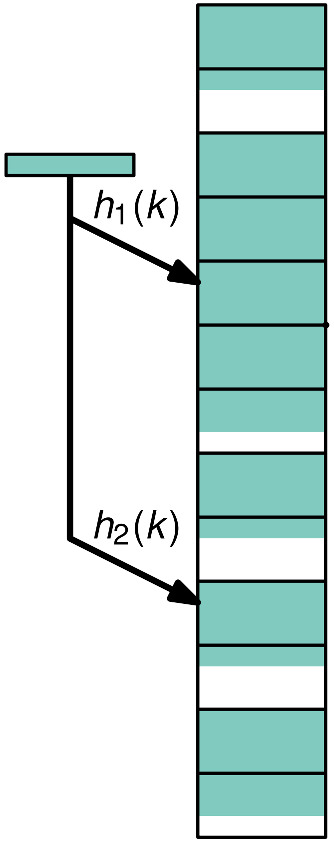


Institute of Theoretical Informatics
Algorithmics Group

Cuckoo Displacement

H-ary *B*-Bucket Cuckoo Hashing
[Pagh, Dietzfelbinger, Mehlhorn, Mitzenmacher, ...]

- buckets of *B* cells
- *H* alternative buckets per element
  $h_1(k), \ldots, h_H(k)$
- if buckets are full, move existing elements
  - ➨ breadth-first-search

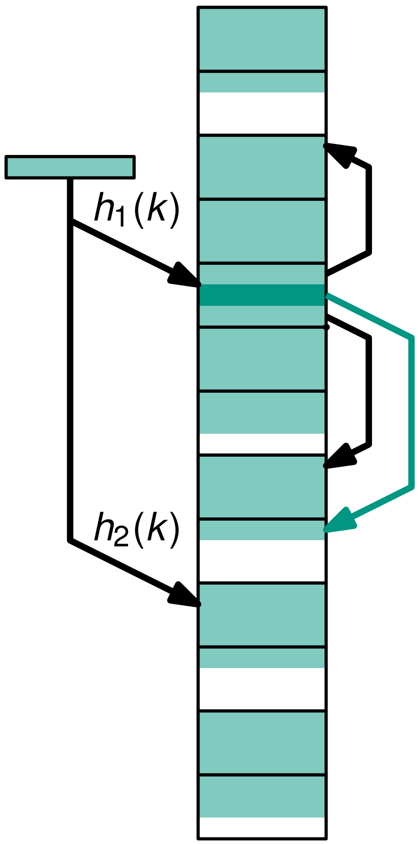


Institute of Theoretical Informatics
Algorithmics Group

Cuckoo Displacement

H-ary B-Bucket Cuckoo Hashing
[Pagh, Dietzfelbinger, Mehlhorn, Mitzenmacher, ...]

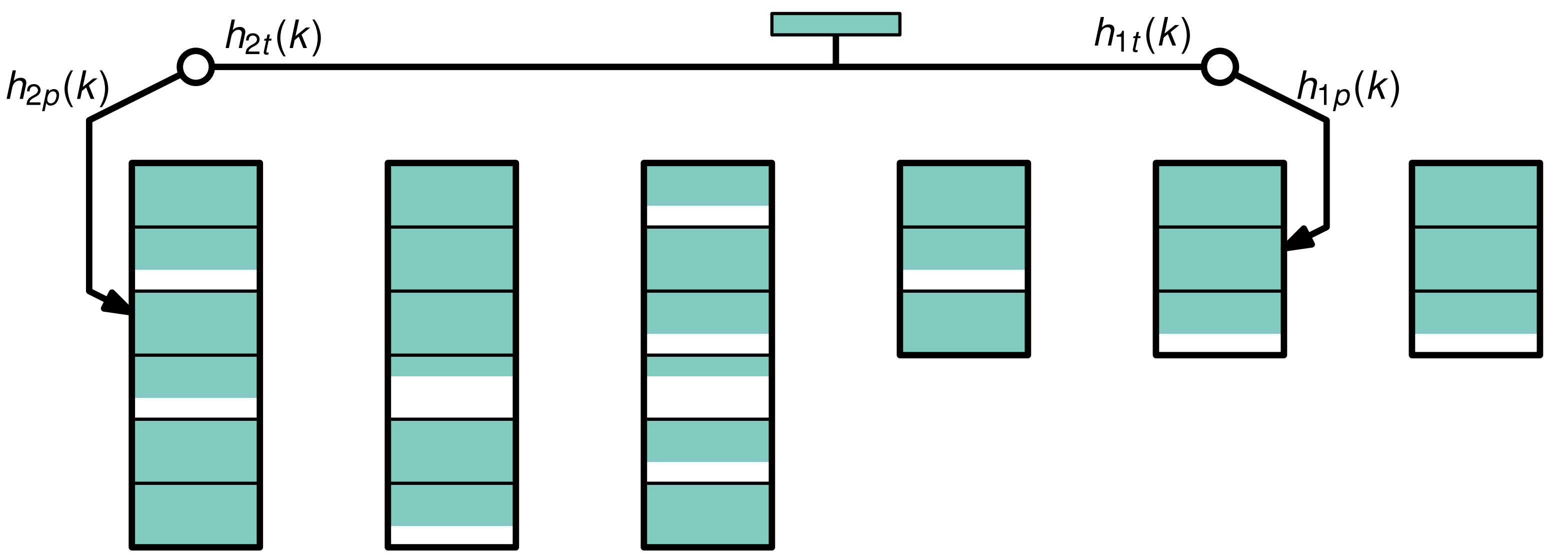
- buckets of B cells
- H alternative buckets per element
  $h_1(k), \ldots, h_H(k)$
- if buckets are full, move existing elements
  - breadth-first-search



Institute of Theoretical Informatics
Algorithmics Group

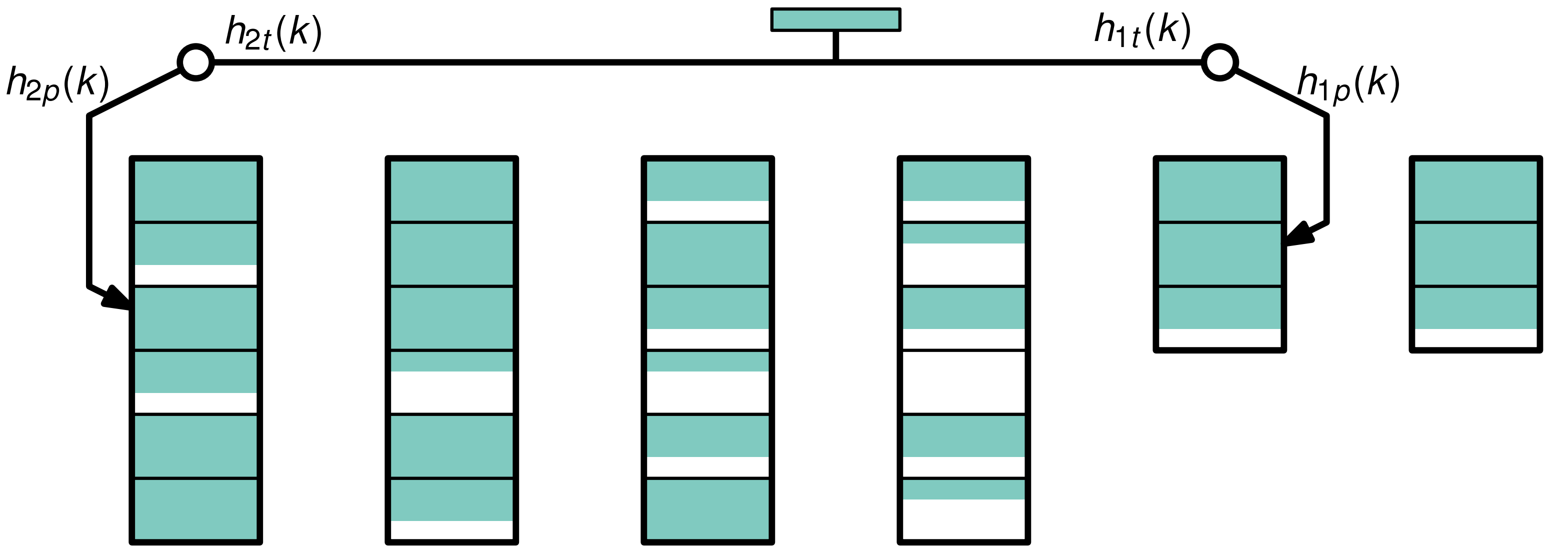
Contribution – Dynamic Space Efficient Cuckoo Table

- use subtables of unequal size (use powers of 2)
  - $h_i(k) \Rightarrow h_{it}(k)$ table and $h_{ip}(k)$ position in table
  - doubling one subtable $\Leftrightarrow$ small overall factor
- use displacements to equalize load imbalance



Contribution – Dynamic Space Efficient Cuckoo Table

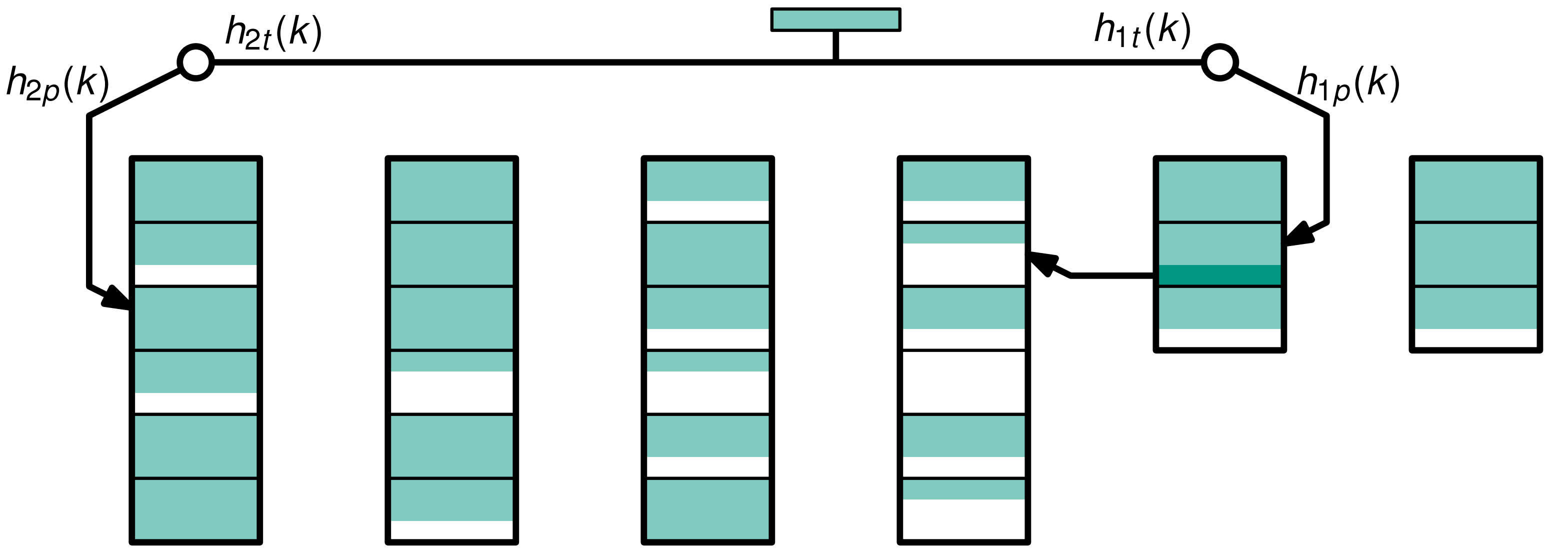
- use subtables of unequal size (use powers of 2)
  - $h_i(k) \Rightarrow h_{it}(k)$ table and $h_{ip}(k)$ position in table
  - doubling one subtable $\Leftrightarrow$ small overall factor
- use displacements to equalize load imbalance



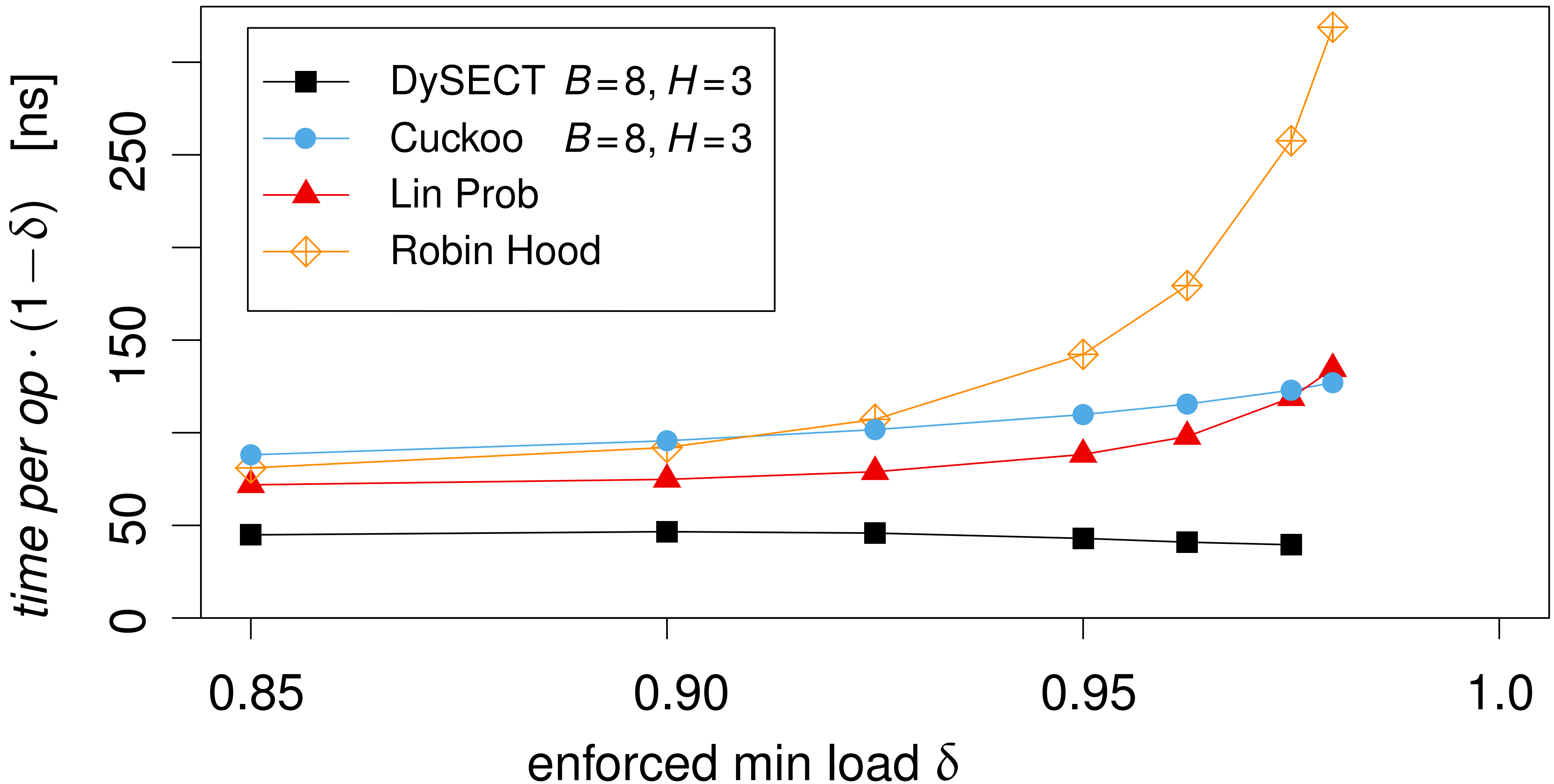
Institute of Theoretical Informatics
Algorithmics Group

Contribution – Dynamic Space Efficient Cuckoo Table

- use subtables of unequal size (use powers of 2)
  - $h_i(k) \Rightarrow h_{it}(k)$ table and $h_{ip}(k)$ position in table
  - doubling one subtable $\Leftrightarrow$ small overall factor
- use displacements to equalize load imbalance



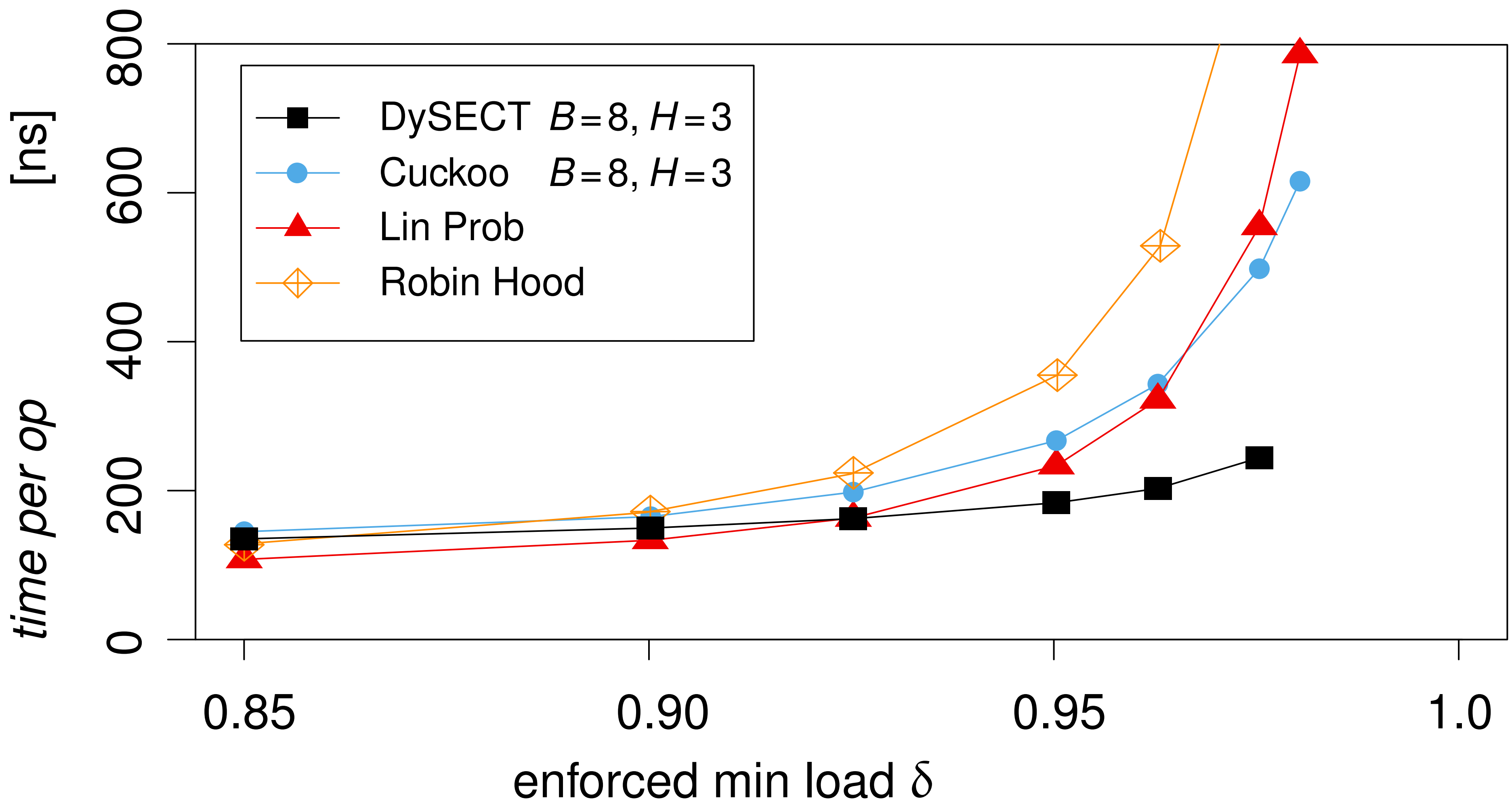
Result – Insertion into Growing Table



- $\frac{1}{1-\delta}$ "expected time" per insertion

Institute of Theoretical Informatics
Algorithmics Group

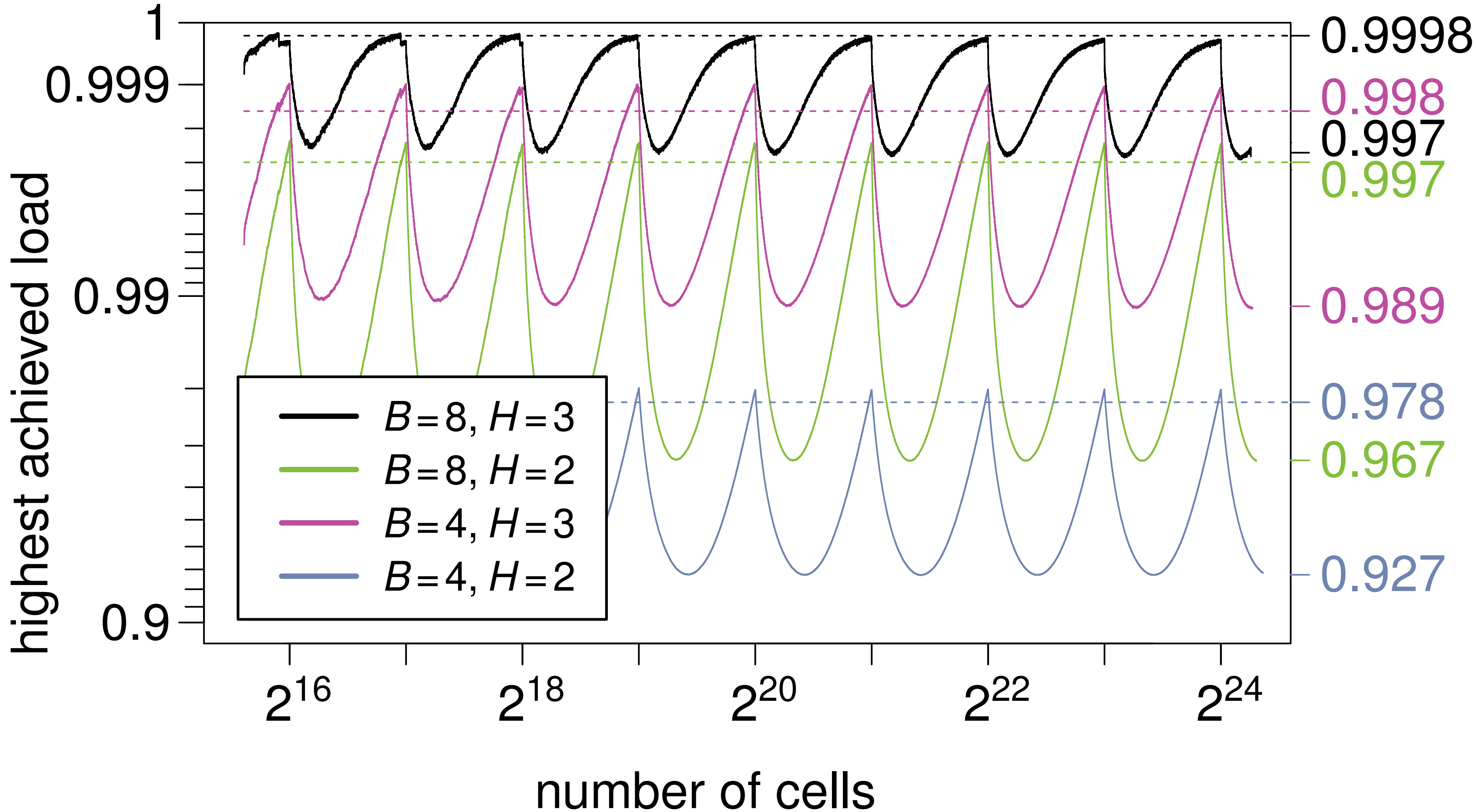
Result – Word Count Benchmark



- CommonCrawl (avg. 12×)
- not normalized

Institute of Theoretical Informatics
Algorithmics Group

Result – Load Bound



- we are in cooperation to prove bounds

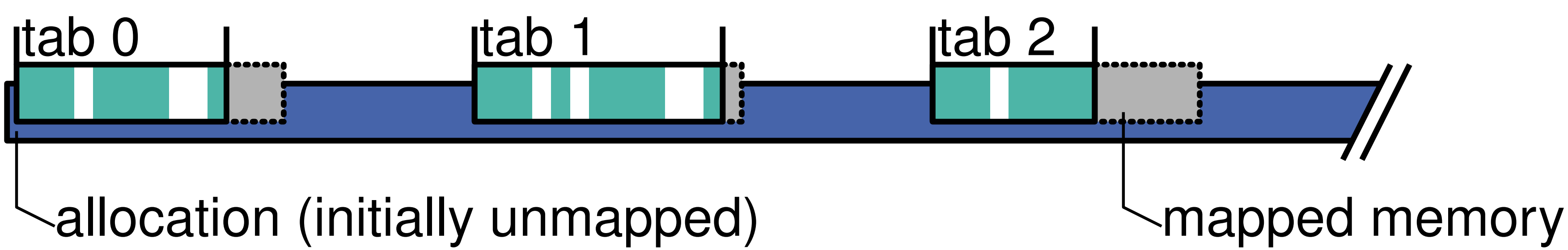
Conclusion

- only dynamic tables offer true space efficiency
- lack of published work on dynamic hash tables
  - ➨ even simple techniques are largely unpublished
- DySECT
  - ➨ no overallocation
  - ➨ constant lookup
  - ➨ addressing uses bit operations
- cuckoo displacement offers more untapped potential
- code available: `https://github.com/TooBiased/DySECT`

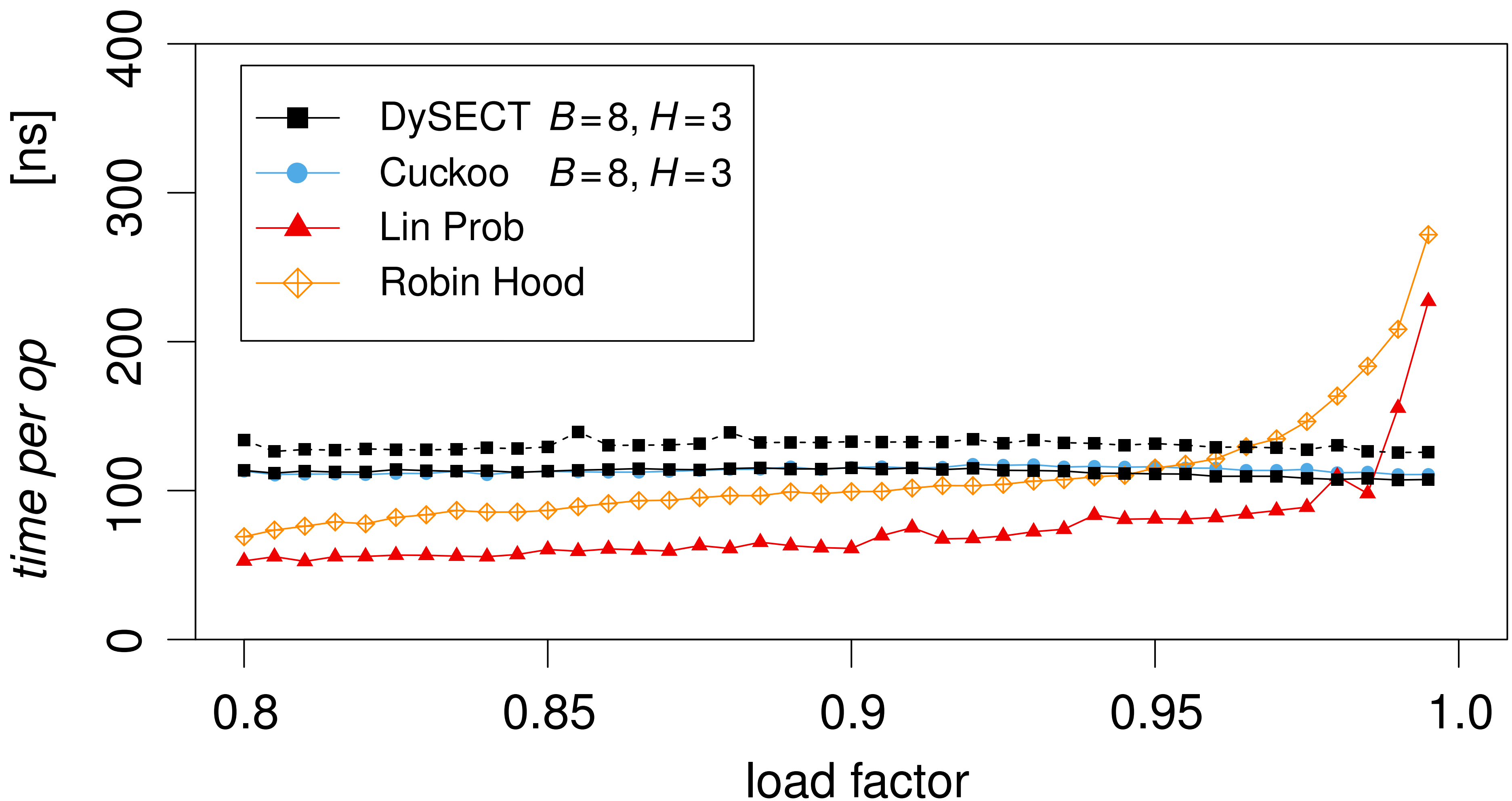
Institute of Theoretical Informatics
Algorithmics Group

(Ab)using Overallocation

- subtables are islands of physical memory in a virtual allocation
- writing to virtual memory $\approx$ increasing local allocation
  - ⊞ inplace growing
  - ⊞ no explicit indirection
  - ⊟ limited portability



Result - Successful Find



Result - Unsuccessful Find