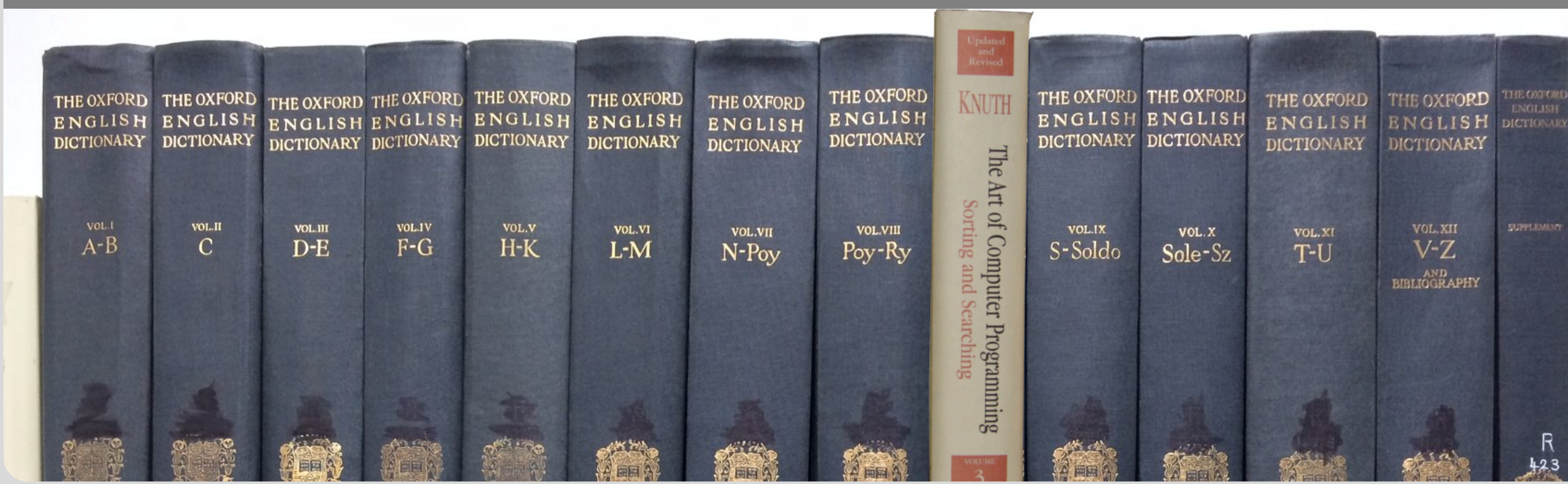
Karlsruhe Institute of Technology

Dynamic Space Efficient Hashing

Presentation · **05. September 2017**

Tobias Maier and Peter Sanders

Institute of Theoretical Informatics · Algorithmics Group



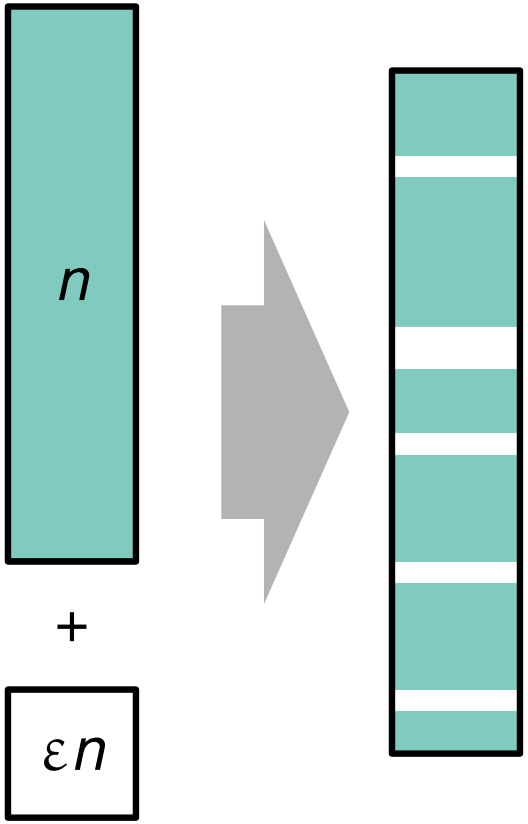
Basics – Hash Tables

- ubiquitous dictionary datastructure
  - insert
  - find (preferably $O(1)$)
  - erase (preferably $O(1)$)
- we do not consider chaining
  - not space efficient for small elements

Institute of Theoretical Informatics
Algorithmics Group

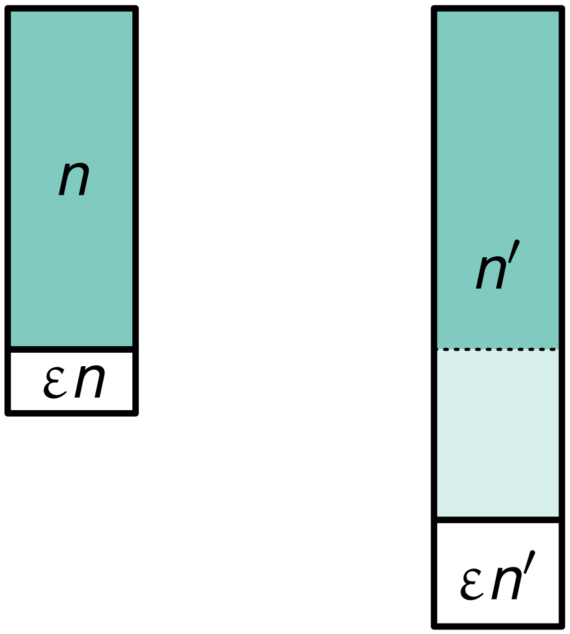
“Classic” Space Efficient Hashing

- densely filled table
- lots of collisions
  - needs good collision handling
- static size (post-initialization)
  - fixed number of elements



Final Size Not Known A Priori

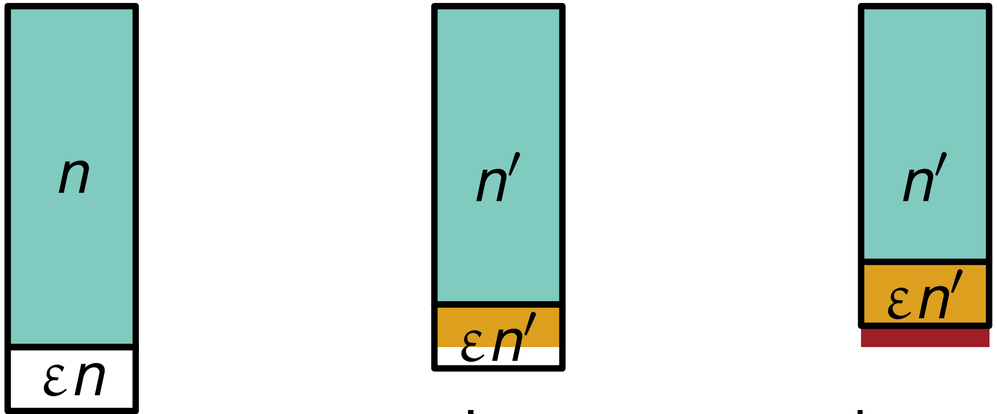
- conservative estimate $n \leq n'$
  - strict bound might not be reasonable
  - less space efficient



Institute of Theoretical Informatics
Algorithmics Group

Final Size Not Known A Priori

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



slow needs growing

Final Size Not Known A Priori

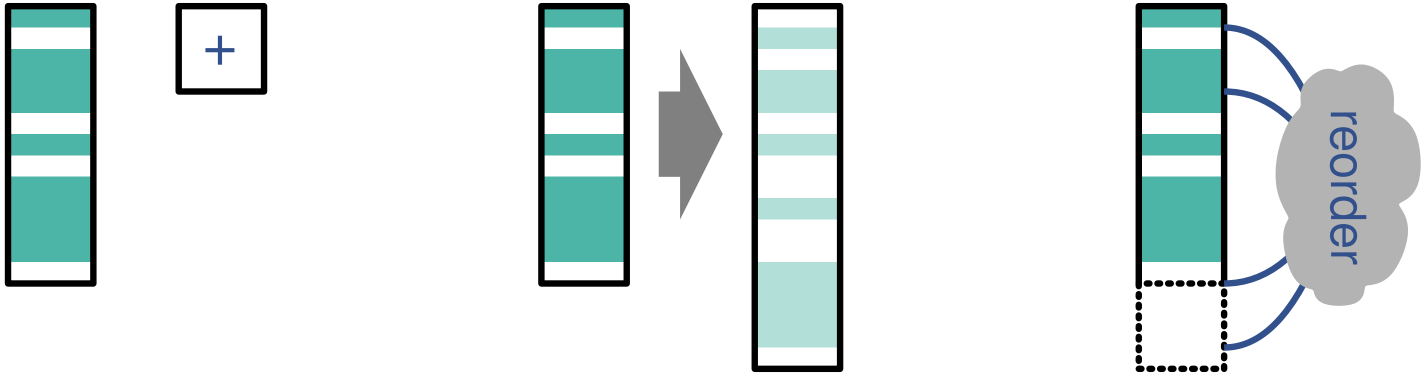
- conservative estimate
- optimistic estimate
- number of elements changes over time
  - ➨ cannot be initialized with max size

Institute of Theoretical Informatics
Algorithmics Group

Resizing

- growing has to be in small steps
- basic approaches

additional table | full migration | inplace+reorder



most common in libraries

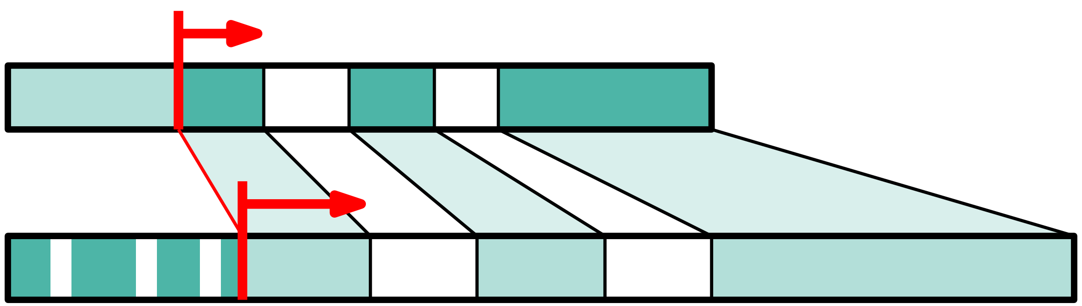
Institute of Theoretical Informatics
Algorithmics Group

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



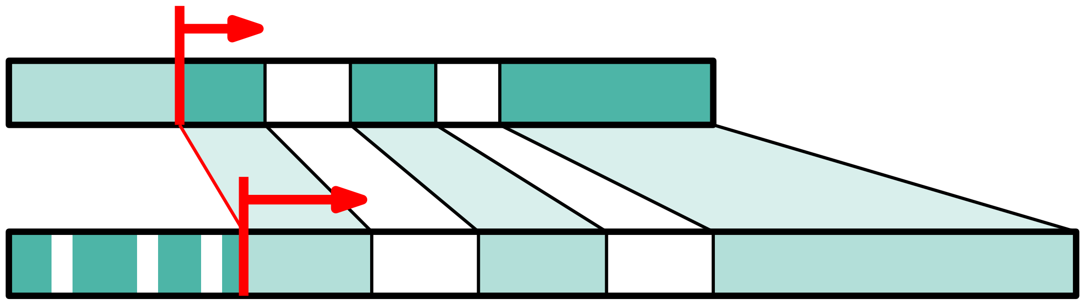
Institute of Theoretical Informatics
Algorithmics Group

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



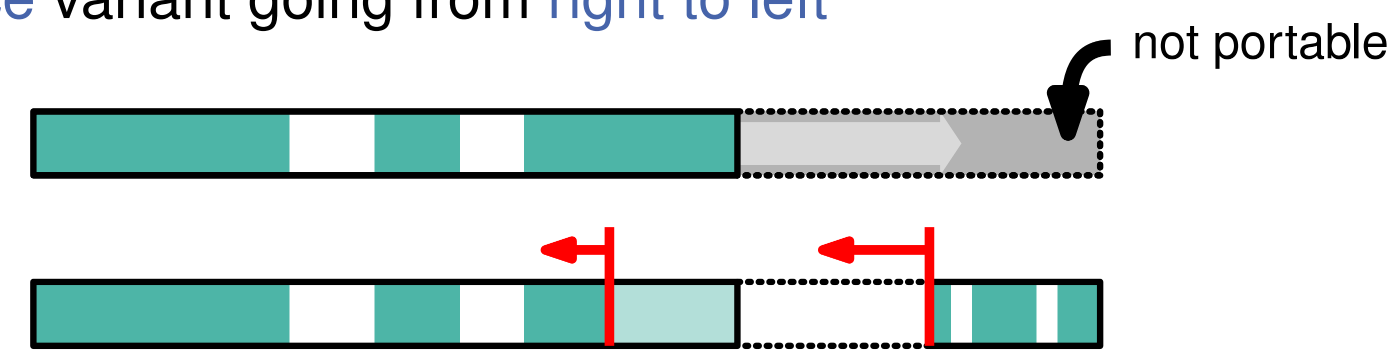
Institute of Theoretical Informatics
Algorithmics Group

Secondary Contribution – Efficient Growing

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

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

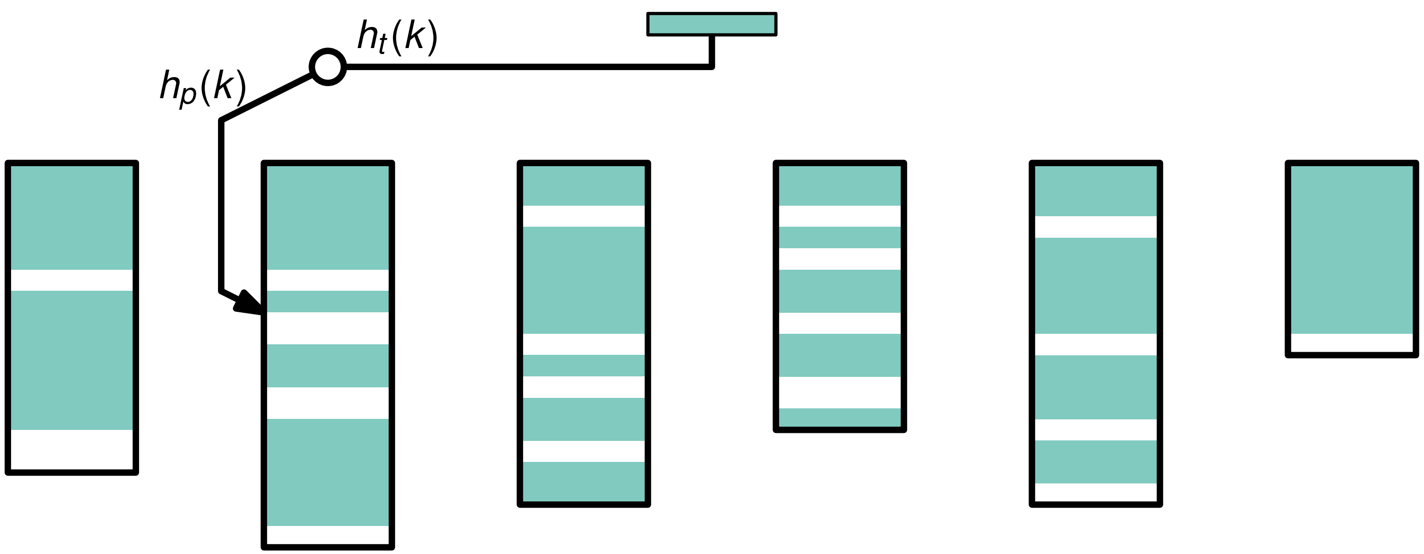
- very fast migration due to cache efficiency
- inplace variant going from right to left



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

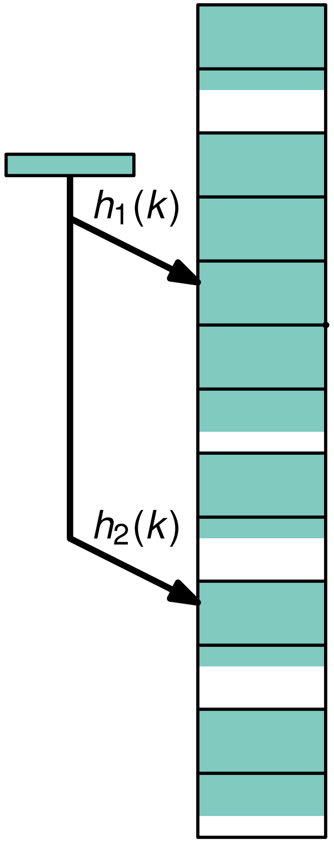


Institute of Theoretical Informatics
Algorithmics Group

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



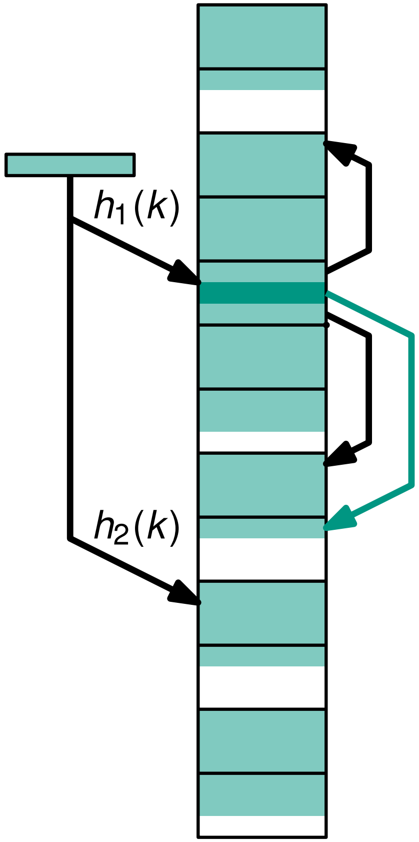
Institute of Theoretical Informatics
Algorithmics Group

Cuckoo Displacement

Karlsruhe Institute of Technology

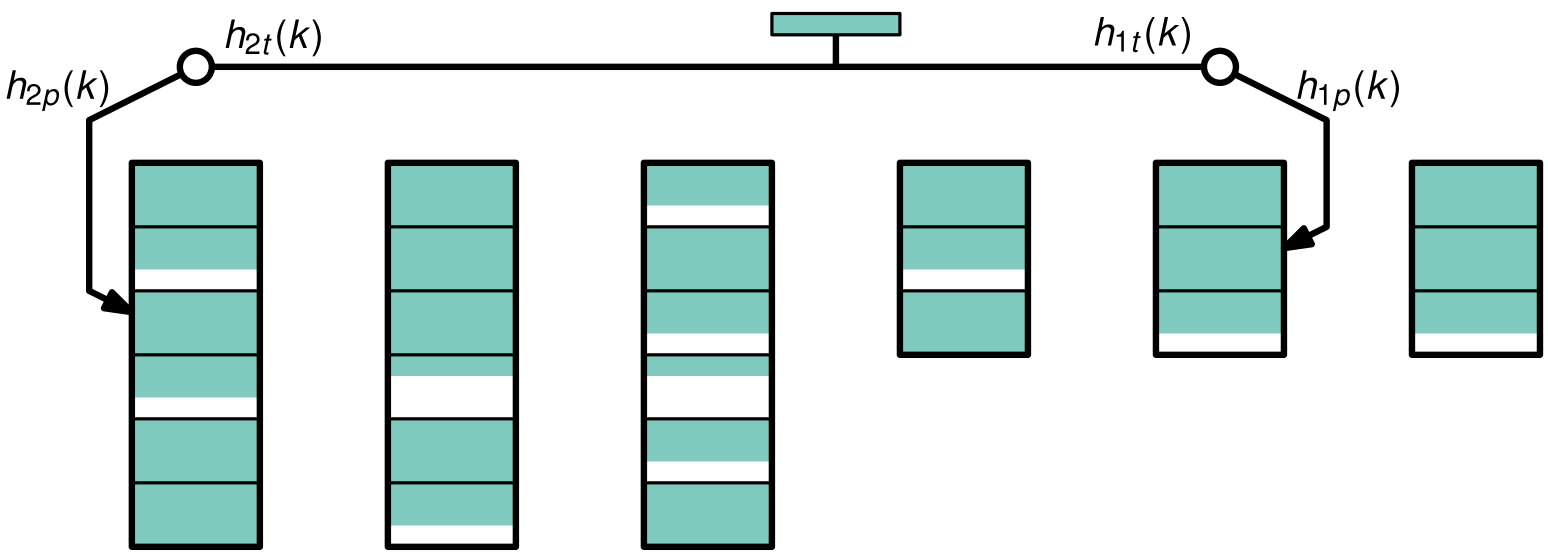
H-ary *B*-Bucket Cuckoo Hashing
[Pagh, Dietzfelbinger, Mehlhorn, Mitzenmacher, ...]

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



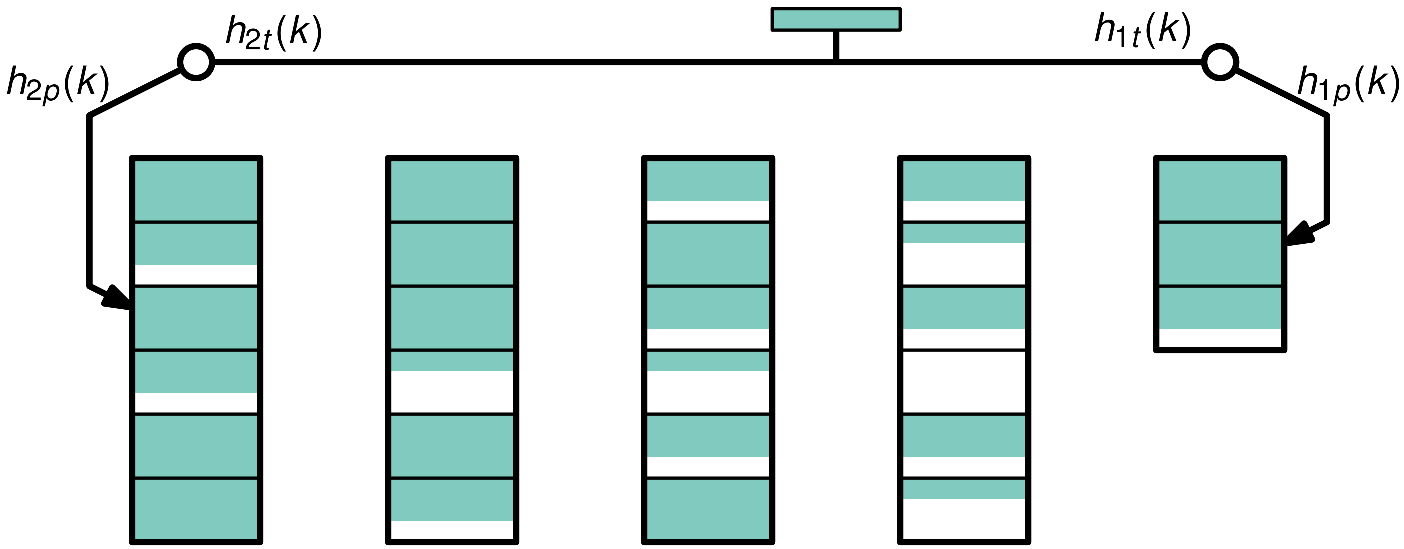
Contribution – Dynamic Space Efficient Cuckoo Table

- 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



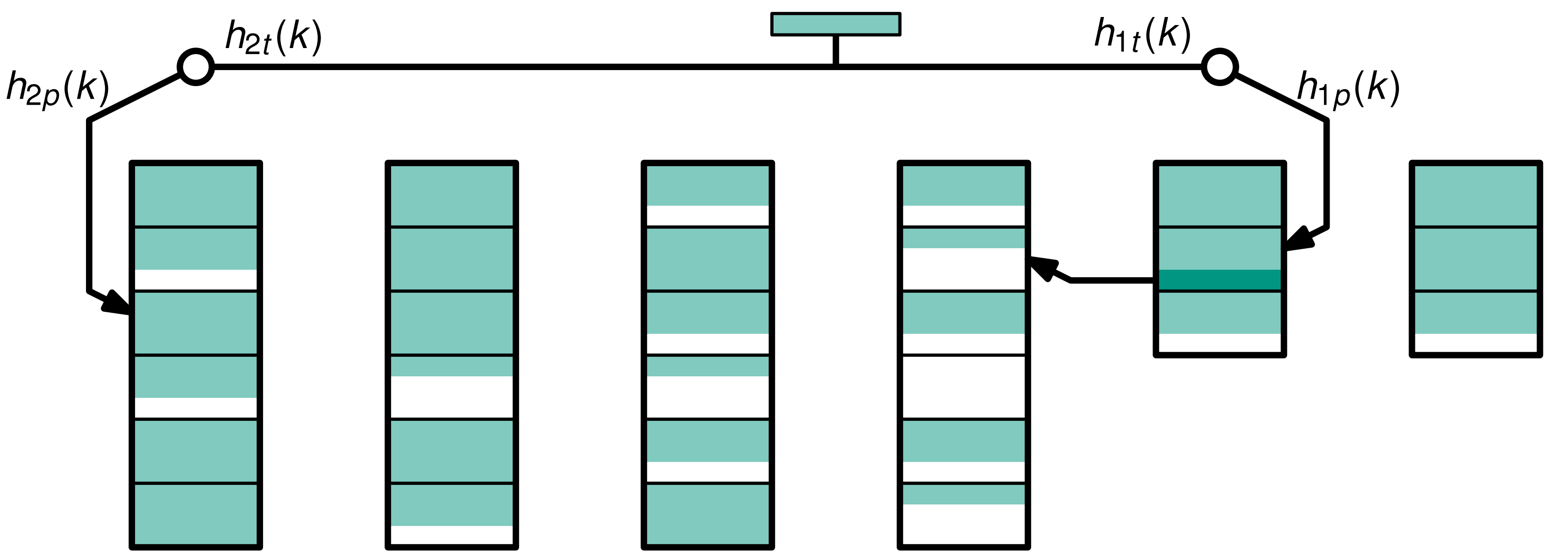
Contribution – Dynamic Space Efficient Cuckoo Table

- 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



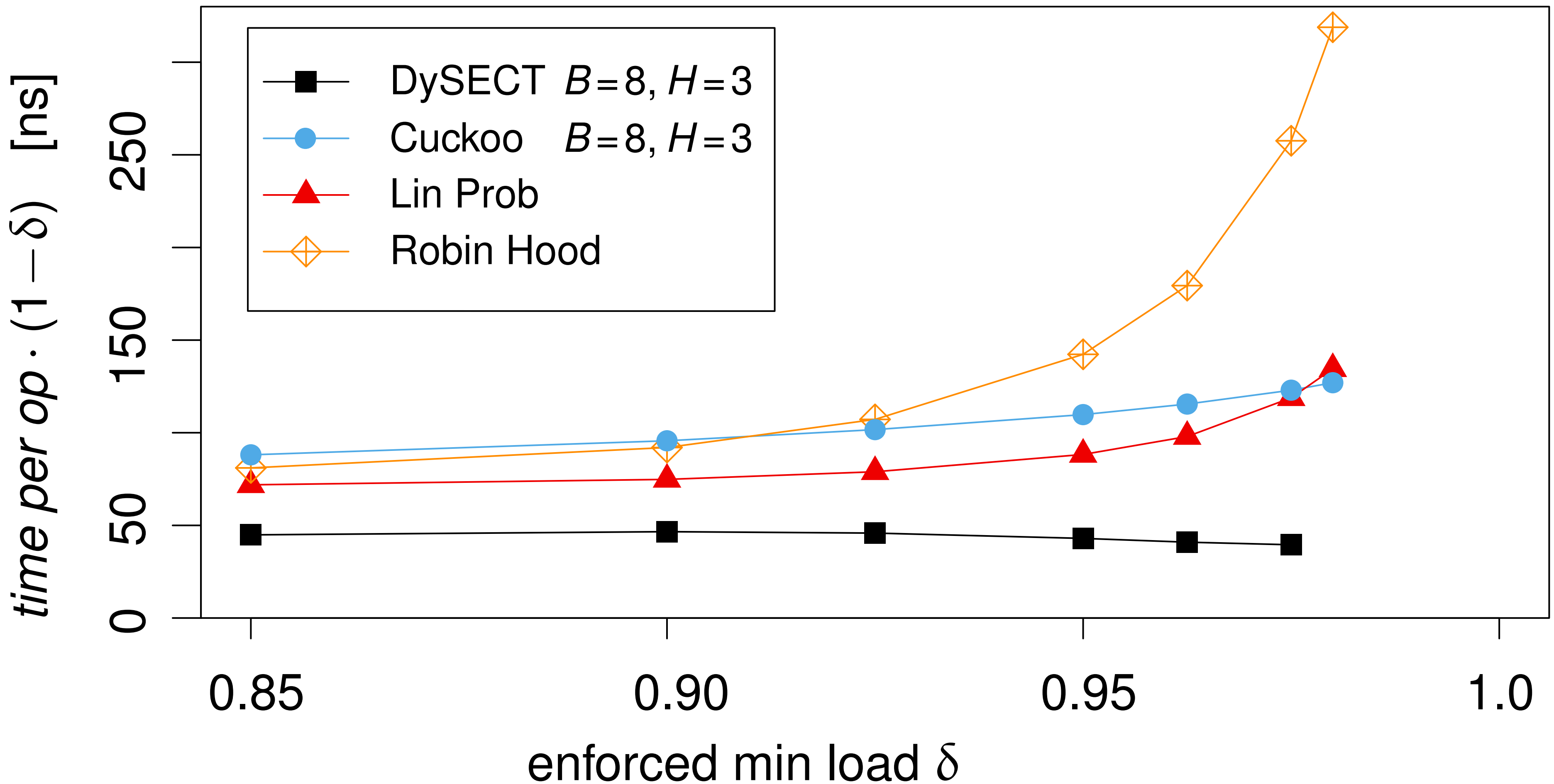
Contribution – Dynamic Space Efficient Cuckoo Table

- 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



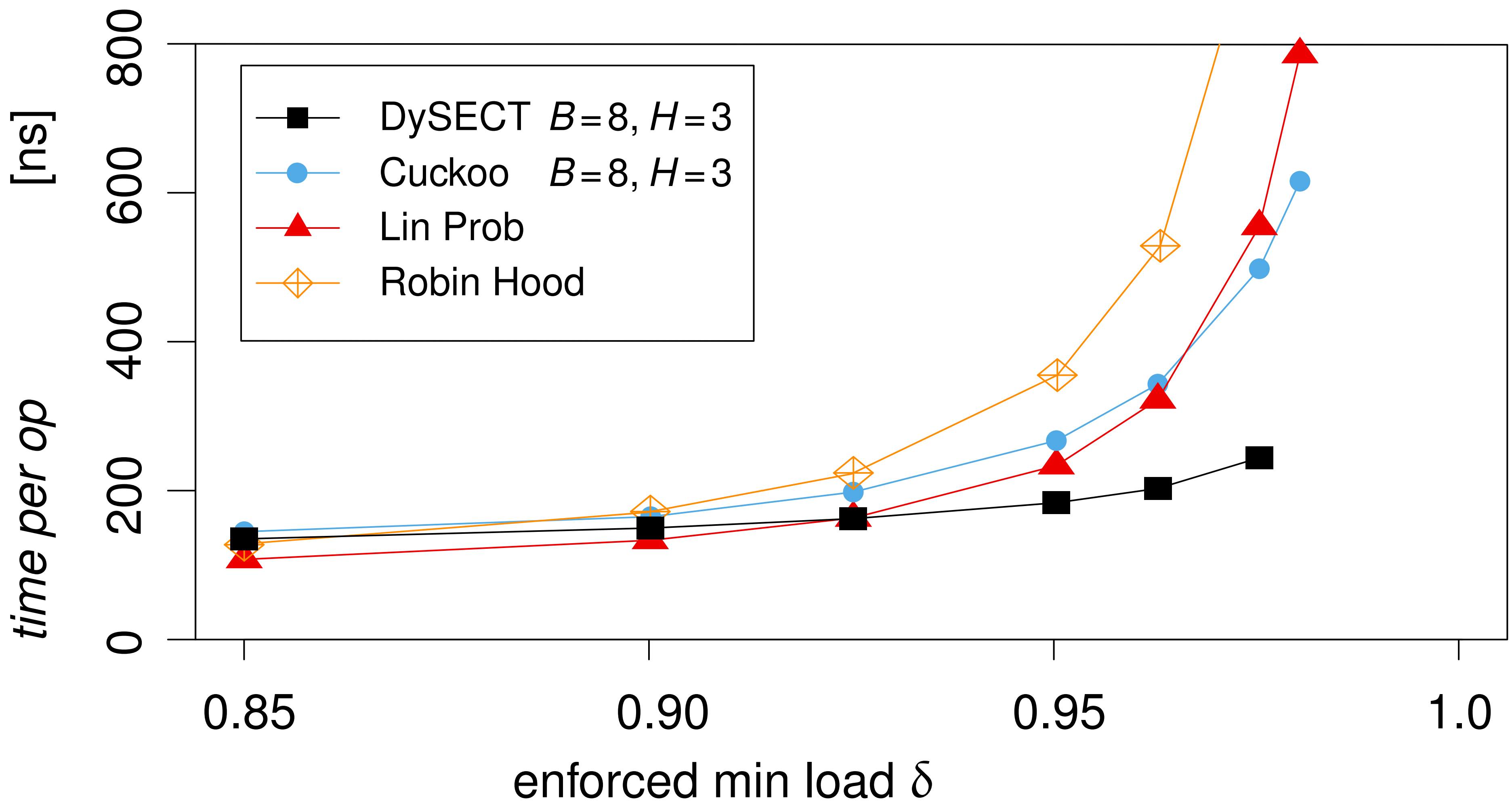
Institute of Theoretical Informatics
Algorithmics Group

Result – Insertion into Growing Table



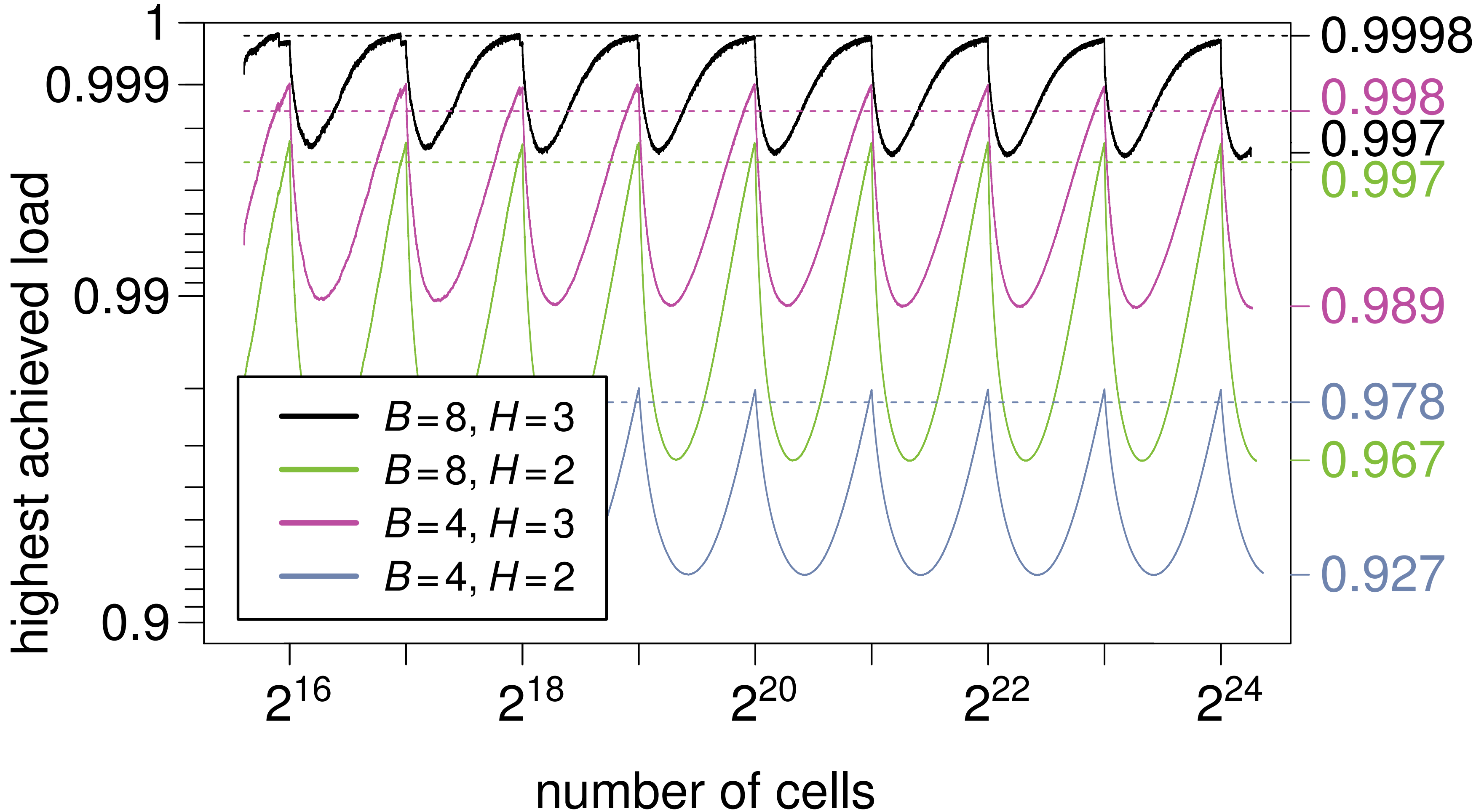
- $\frac{1}{1-\delta}$ “expected time” per insertion

Result – Word Count Benchmark



- CommonCrawl (avg. 12×)
- not normalized

Result – Load Bound



- we are in cooperation to prove bounds

Institute of Theoretical Informatics
Algorithmics Group

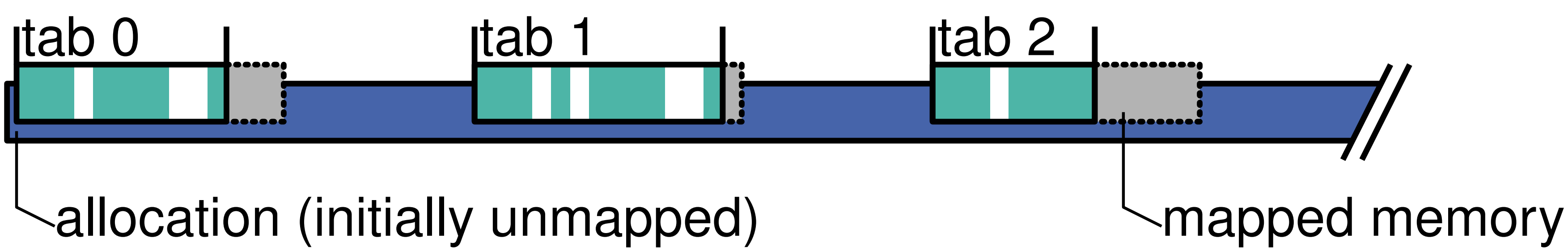
Conclusion

- 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

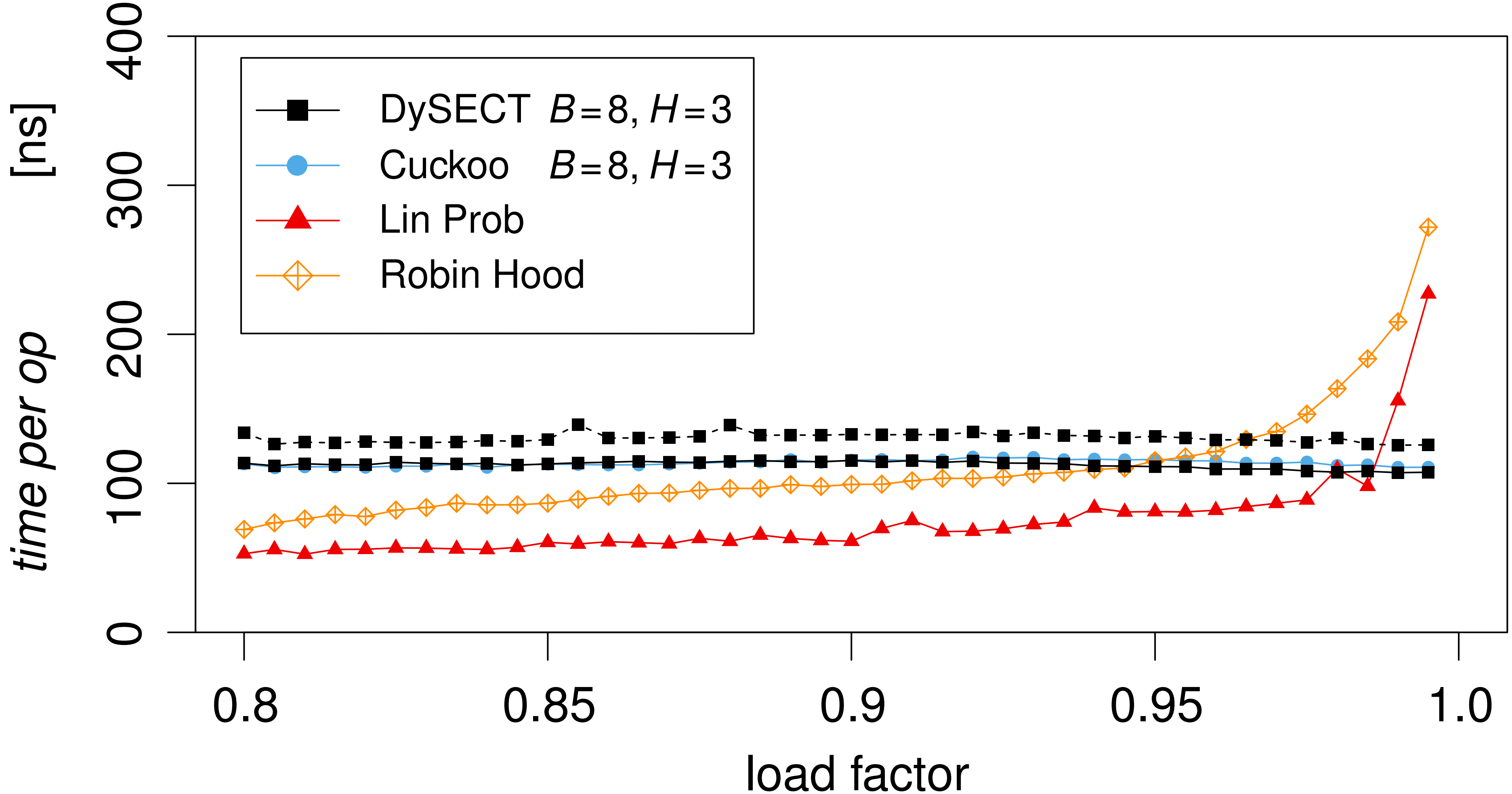
(Ab)using Overallocation

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



Institute of Theoretical Informatics
Algorithmics Group

Result - Successful Find



Institute of Theoretical Informatics
Algorithmics Group

Result - Unsuccessful Find