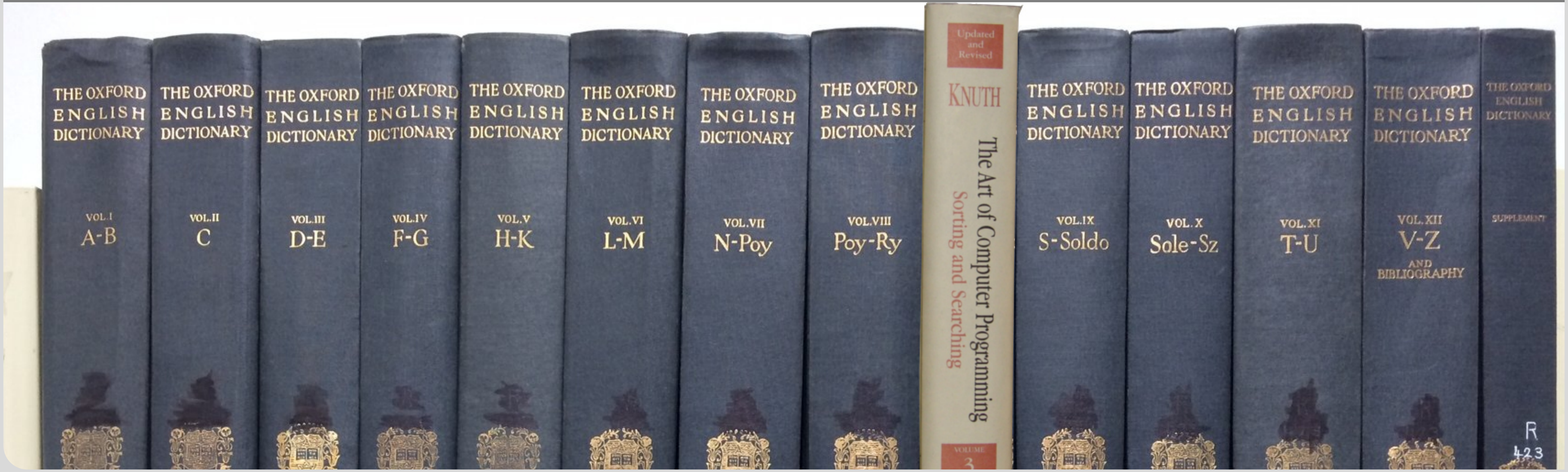
Karlsruhe Institute of Technology

Hashing

Lecture · **11. June 2019**
Tobias Maier and Peter Sanders

INSTITUTE OF THEORETICAL INFORMATICS · ALGORITHMICS GROUP



KIT – University of the State of Baden-Wuerttemberg and
National Laboratory of the Helmholtz Association

www.kit.edu

Hash Tables – Definitions

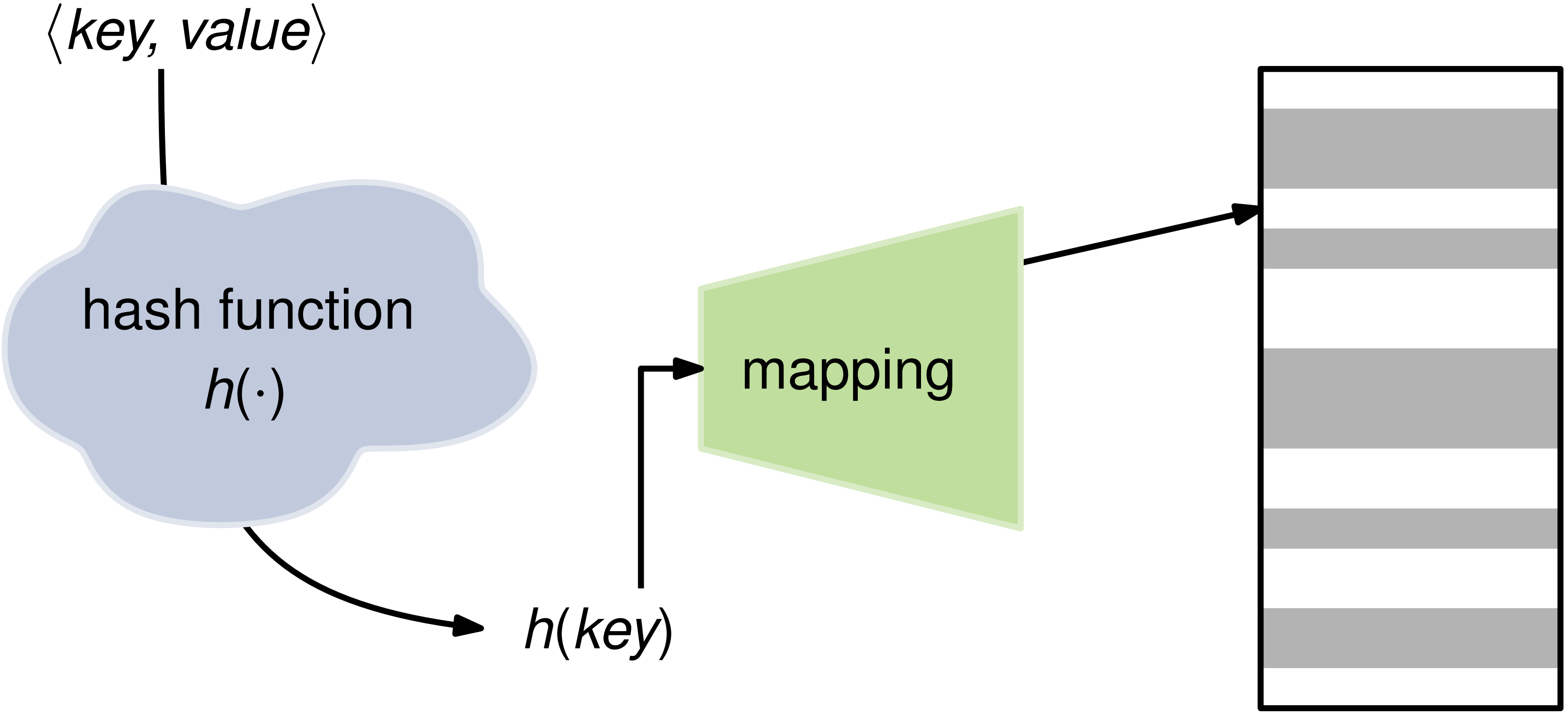
- set $S \subseteq U = \mathit{Keys} \times \mathit{Values}$
  - each Key is unique in S
  - $n = |S|$ elements in m cells

- Operations
  - insert
  - find
  - erase

All preferably in O(1)

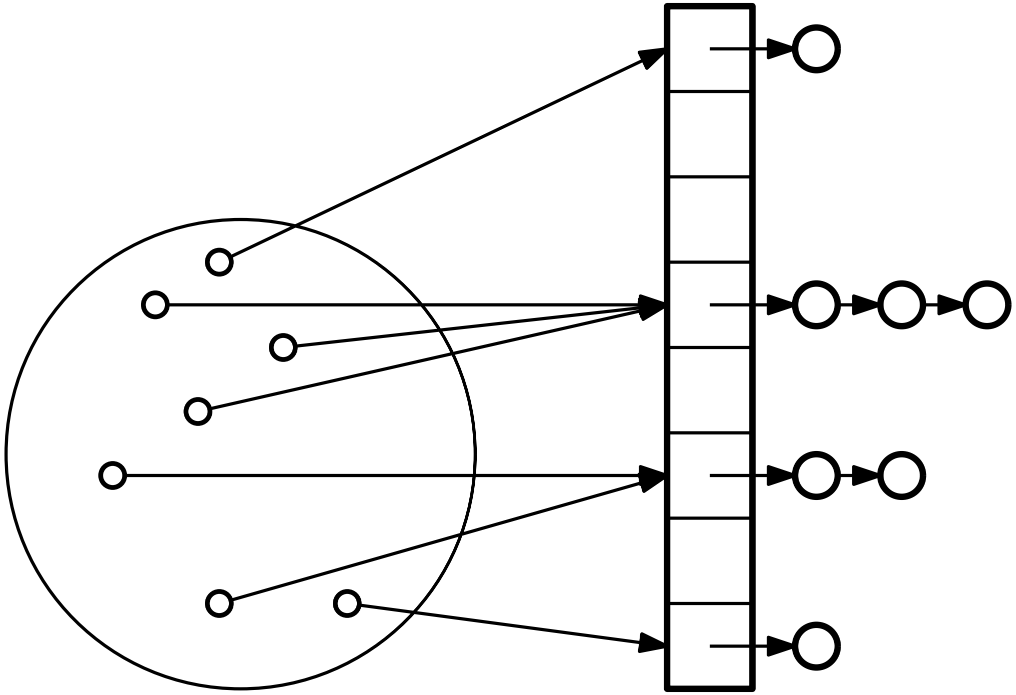
Hash Tables – Mapping

- Position depends on the **key**
- **Independent of** the **time** of insertion



Hash Tables – Chaining = Balls into Bins

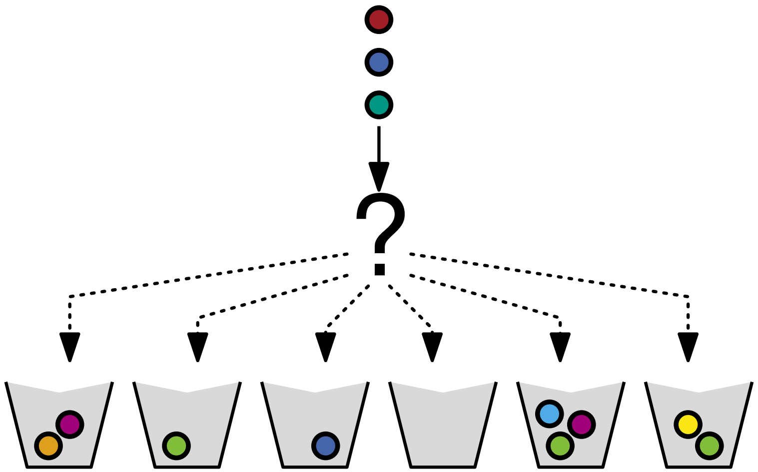
- Worst case find is in O(n)
- Probabilistic Bounds

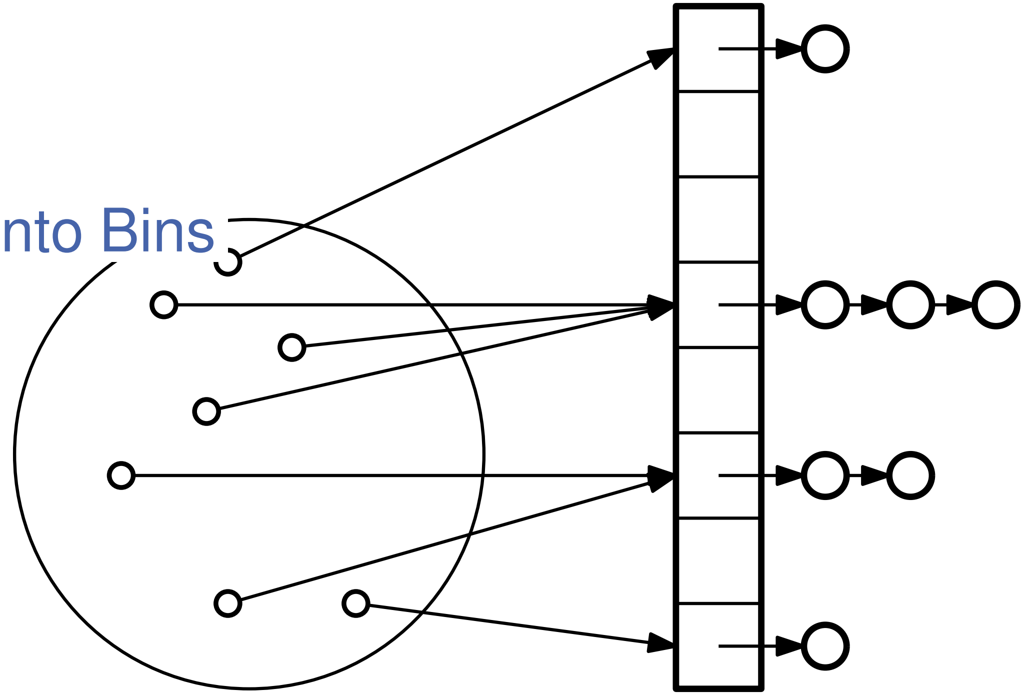


Hash Tables – Chaining = Balls into Bins

- Worst case find is in O(n)
- Probabilistic Bounds

Hashing with Chaining = Balls into Bins





Excursion – Probability Theory

- sample space Ω
- events $\varepsilon \subset \Omega$
- probability p_x of $x \in \Omega$
- probability of an event
  $\mathbb{P}[\varepsilon] = \sum_{x \in \varepsilon} p_x$
- random variable
  $X : \Omega \rightarrow \mathbb{R}$
- expectation
  $E[X] = \sum_{y \in \Omega} p_y X(y)$

For Example:

- random hash functions / mapping
  $Keys \mapsto \{0..m-1\}$
- keys k_1 and k_2 have a collision
  $\varepsilon_{k_1,k_2} = \{h \in \Omega : h(k_1) = h(k_2)\}$
- uniform distribution
  $\forall h \in \Omega \ : \ p_h = \frac{1}{m^{|Keys|}}$
- $\mathbb{P}[\varepsilon_{k_1,k_2}] = 1/m$ *
- #elements hashed to 0
  $X_0 = |\{x \in S \ : \ h(x) = 0\}|$
- expected #elements in one cell
  $E[X_0] = \frac{n}{m}$ *

* assuming a uniform hash function

Excursion – Probability Theory

- sample space Ω
- events $\varepsilon \subset \Omega$
- probability p_x of $x \in \Omega$
- probability of an event
  $\mathbb{P}[\varepsilon] = \sum_{x \in \varepsilon} p_x$
- random variable
  $X : \Omega \to \mathbb{R}$
- expectation
  $E[X] = \sum_{y \in \Omega} p_y X(y)$

For Example:

- random hash functions / mapping
  $Keys \mapsto \{0..m-1\}$
- keys k_1 and k_2 have a collision
  $\varepsilon_{k_1,k_2} = \{h \in \Omega : h(k_1) = h(k_2)\}$
- uniform distribution
  $\forall h \in \Omega \ : \ p_h = \frac{1}{m^{|Keys|}}$
- $\mathbb{P}[\varepsilon_{k_1,k_2}] = 1/m$ *
- #elements hashed to 0
  $X_0 = |\{x \in S \ : \ h(x) = 0\}|$
- expected #elements in one cell
  $E[X_0] = \frac{n}{m}$ *

* assuming a uniform hash function

Hash Tables – Chaining (probabilistic) Bound

Linearity of the Expectation

$E[X+Y] = E[X] + E[Y]$

this is always true independent of correlations between X and Y

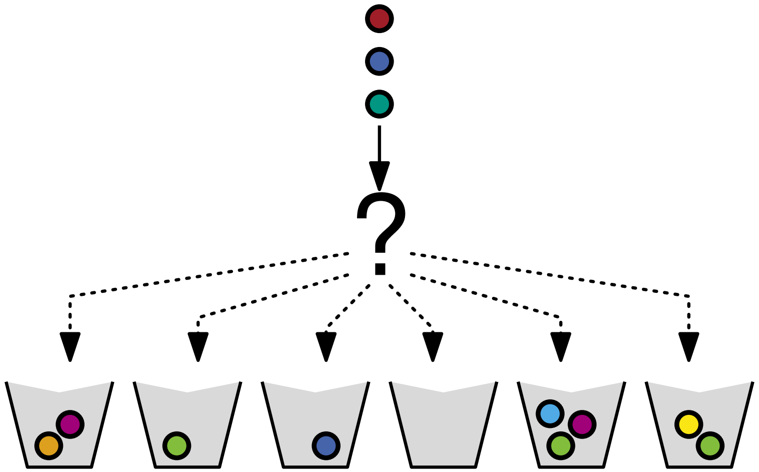
Consider one $\{0,1\}$ random variable for each element X_e

$$X_e = \begin{cases} 1 & h(e) = 0 \\ 0 & otherwise \end{cases}$$

$$E[X_0] = E\left[\sum_{e \in S} X_e\right] = \sum_{e \in S} E[X_e]$$

$$= \sum_{e \in S} \mathbb{P}[X_e = 1] = \frac{n}{m}$$

$E[X_0] = \frac{n}{m}$



Hash Tables – Other Bounds

$$\delta = \frac{m-n}{m}$$

- **Multi Hashing**
  for each collision use a new hash function
  - $t[h_1], t[h_2], \ldots$ have $p = \delta$ chance to be empty
  - $E[\#probes_{insert}] = E[\#probes_{find\ x \notin S}] = \frac{1}{\delta}$
    $E[\#probes_{find\ x \in S}]$ abhängig vom Einfügezeitpunkt

- **Linear Probing**
  in case of a collision use the next empty cell
  - probability of finding a cell depends on its predecessor
  - $E[\#probes_{insert}] = E[\#probes_{find\ x \notin S}] = O(\frac{1}{\delta^2})^*$
    $E[\#probes_{find\ x \in S}] = O(\frac{1}{\delta})^*$

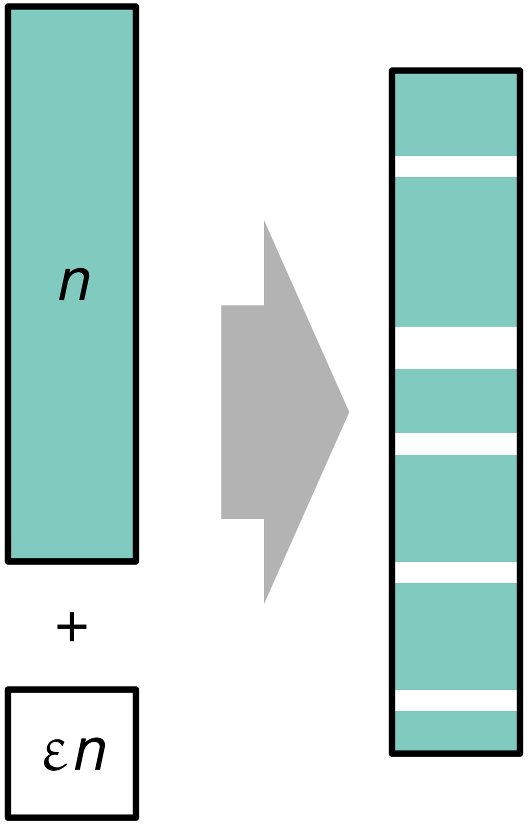
* needs stronger assumption than uniform hash function, e.g. fully random hash function

Hash Tables – More Hashing Issues

- High probability and worst case guarantees
  - ➨ more requirements on the hash functions
- Hashing as a means of load balancing in parallel systems, e.g., storage servers
  - Different disk sizes and speeds
  - Adding disks / replacing failed disks without much copying

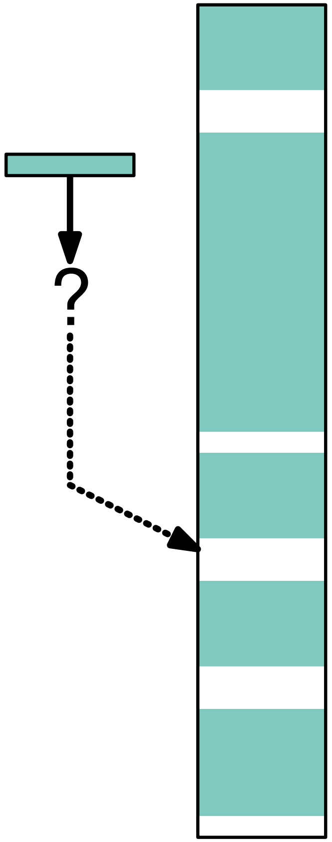
Space Efficient Hashing

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



Space Efficient Hashing – Cuckoo Hashing

- constant lookups independent of fill ratio
- element → const. number possible cells
- if all cells are full, move existing elements

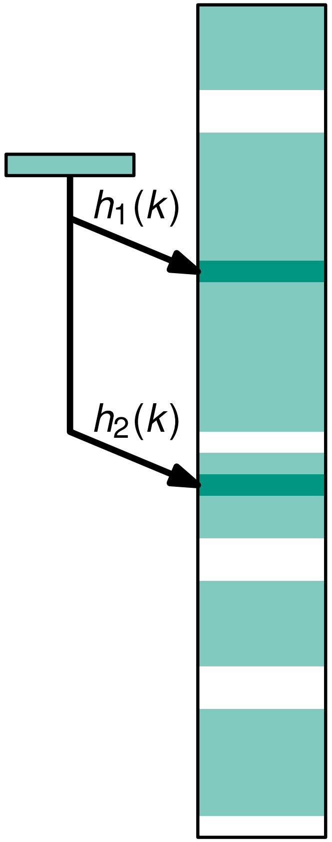


d-ary Bucket Cuckoo Hashing

combination of different results, by:
[Pagh, Dietzfelbinger, Mehlhorn, Mitzenmacher, ...]

Space Efficient Hashing – Cuckoo Hashing

- constant lookups independent of fill ratio
- element → const. number possible cells
- if all cells are full, move existing elements
  - breadth-first-search
  - 2 alternative buckets per element $h_1(k), h_2(k)$

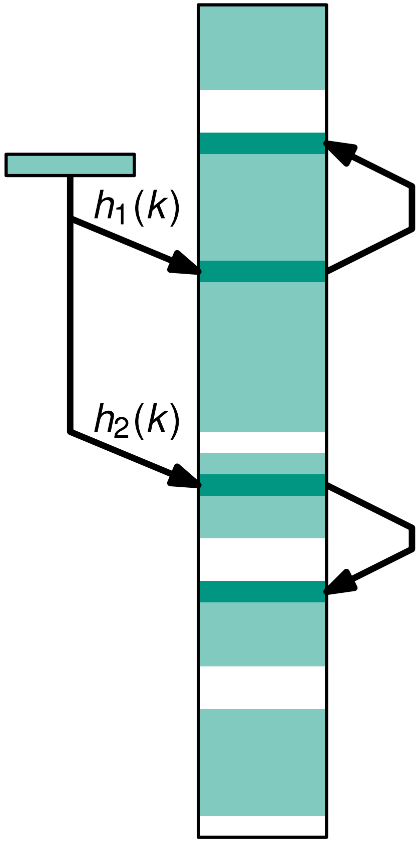


d-ary Bucket Cuckoo Hashing

combination of different results, by:
[Pagh, Dietzfelbinger, Mehlhorn, Mitzenmacher, ...]

Space Efficient Hashing – Cuckoo Hashing

- constant lookups independent of fill ratio
- element → const. number possible cells
- if all cells are full, move existing elements
  - breadth-first-search
  - 2 alternative buckets per element $h_1(k), h_2(k)$

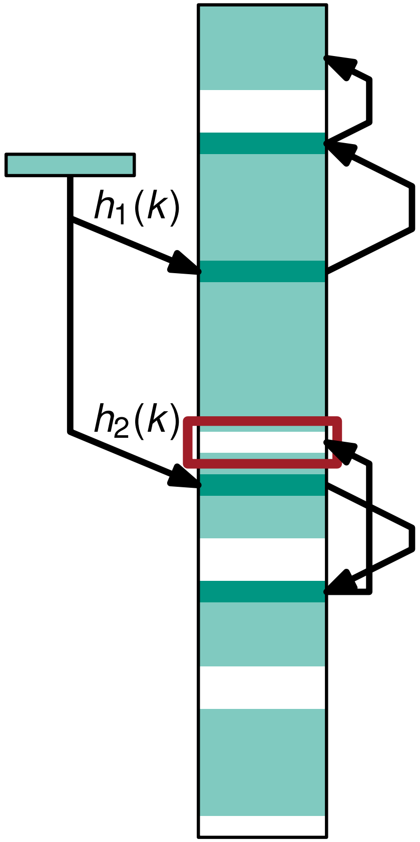


d-ary Bucket Cuckoo Hashing

combination of different results, by:
[Pagh, Dietzfelbinger, Mehlhorn, Mitzenmacher, ...]

Space Efficient Hashing – Cuckoo Hashing

- constant lookups independent of fill ratio
- element → const. number possible cells
- if all cells are full, move existing elements
  - breadth-first-search
  - 2 alternative buckets per element $h_1(k), h_2(k)$

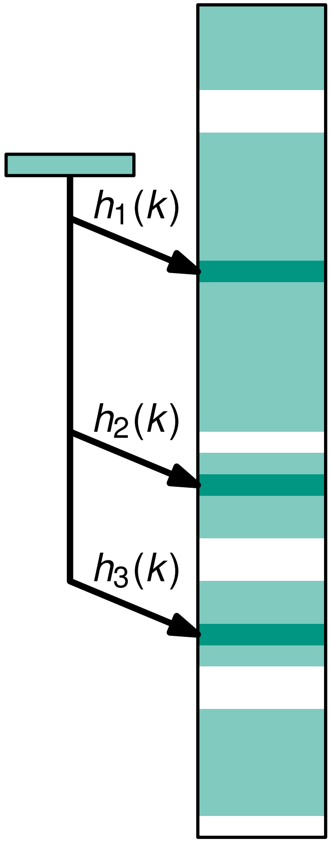


d-ary Bucket Cuckoo Hashing

combination of different results, by:
[Pagh, Dietzfelbinger, Mehlhorn, Mitzenmacher, ...]

Space Efficient Hashing – Cuckoo Hashing

- constant lookups independent of fill ratio
- element → const. number possible cells
- if all cells are full, move existing elements
  - breadth-first-search
  - d alternative buckets per element $h_1(k), ..., h_d(k)$

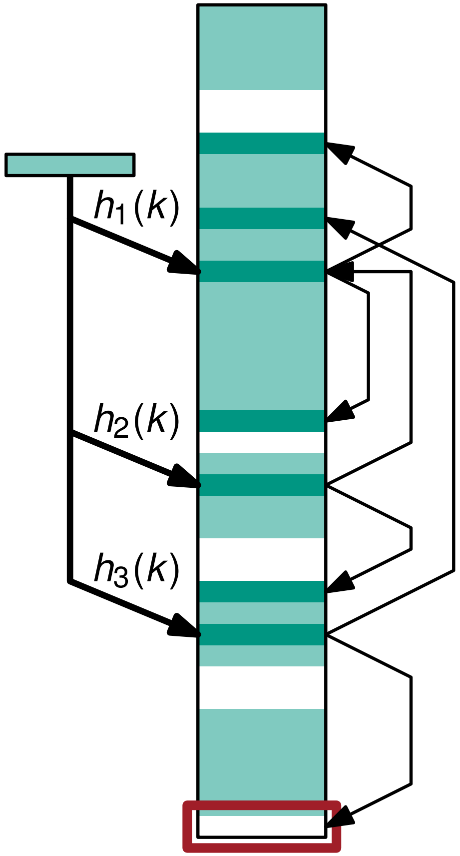


d-ary Bucket Cuckoo Hashing

combination of different results, by:
[Pagh, Dietzfelbinger, Mehlhorn, Mitzenmacher, ...]

Space Efficient Hashing – Cuckoo Hashing

- constant lookups independent of fill ratio
- element → const. number possible cells
- if all cells are full, move existing elements
  - breadth-first-search
  - d alternative buckets per element $h_1(k), \dots, h_d(k)$

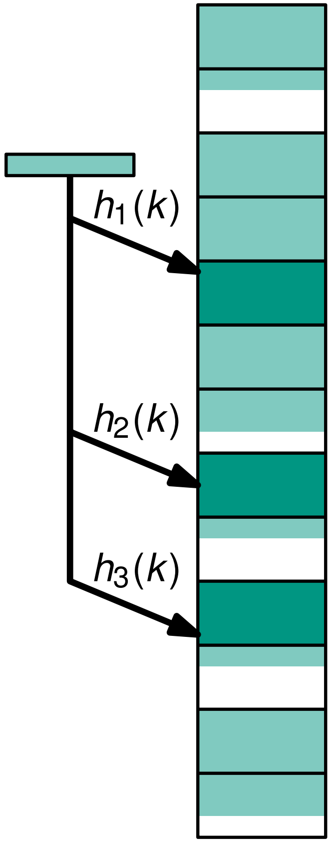


d-ary Bucket Cuckoo Hashing

combination of different results, by:
[Pagh, Dietzfelbinger, Mehlhorn, Mitzenmacher, ...]

Space Efficient Hashing – Cuckoo Hashing

- constant lookups independent of fill ratio
- element → const. number possible cells
- if all cells are full, move existing elements
  - ➨ breadth-first-search
  - d alternative buckets per element $h_1(k), \ldots, h_d(k)$
  - buckets of B cells



d-ary Bucket Cuckoo Hashing

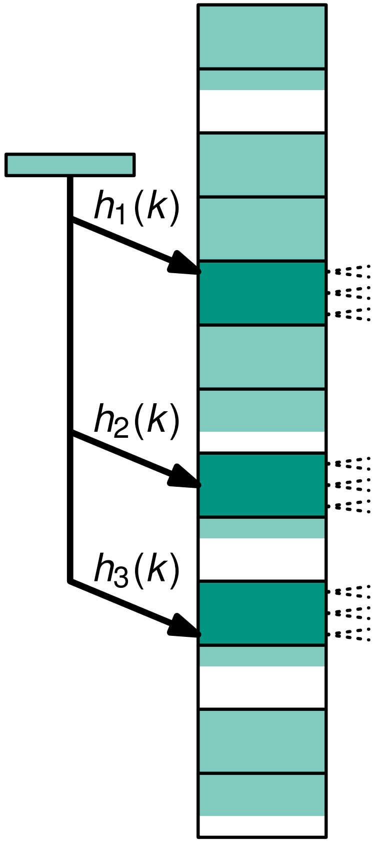
combination of different results, by:
[Pagh, Dietzfelbinger, Mehlhorn, Mitzenmacher, ...]

Space Efficient Hashing – Cuckoo Hashing

- constant lookups independent of fill ratio
- element → const. number possible cells
- if all cells are full, move existing elements
  - breadth-first-search
  - *d* alternative buckets per element $h_1(k), \ldots, h_d(k)$
  - buckets of *B* cells

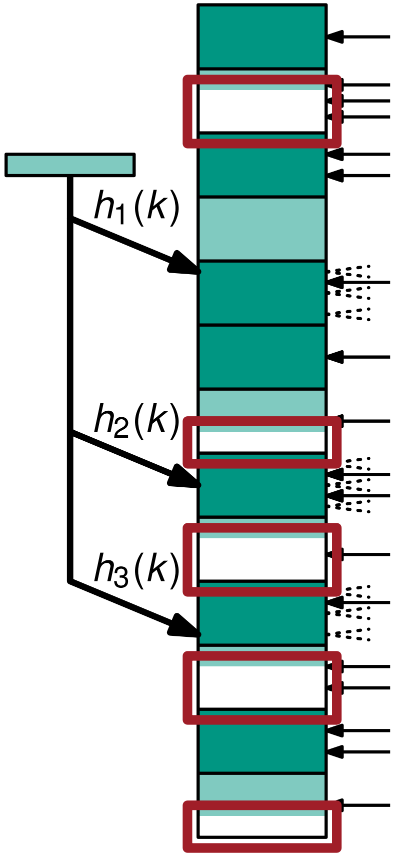
d-ary Bucket Cuckoo Hashing

combination of different results, by:
[Pagh, Dietzfelbinger, Mehlhorn, Mitzenmacher, ...]



Space Efficient Hashing – Cuckoo Hashing

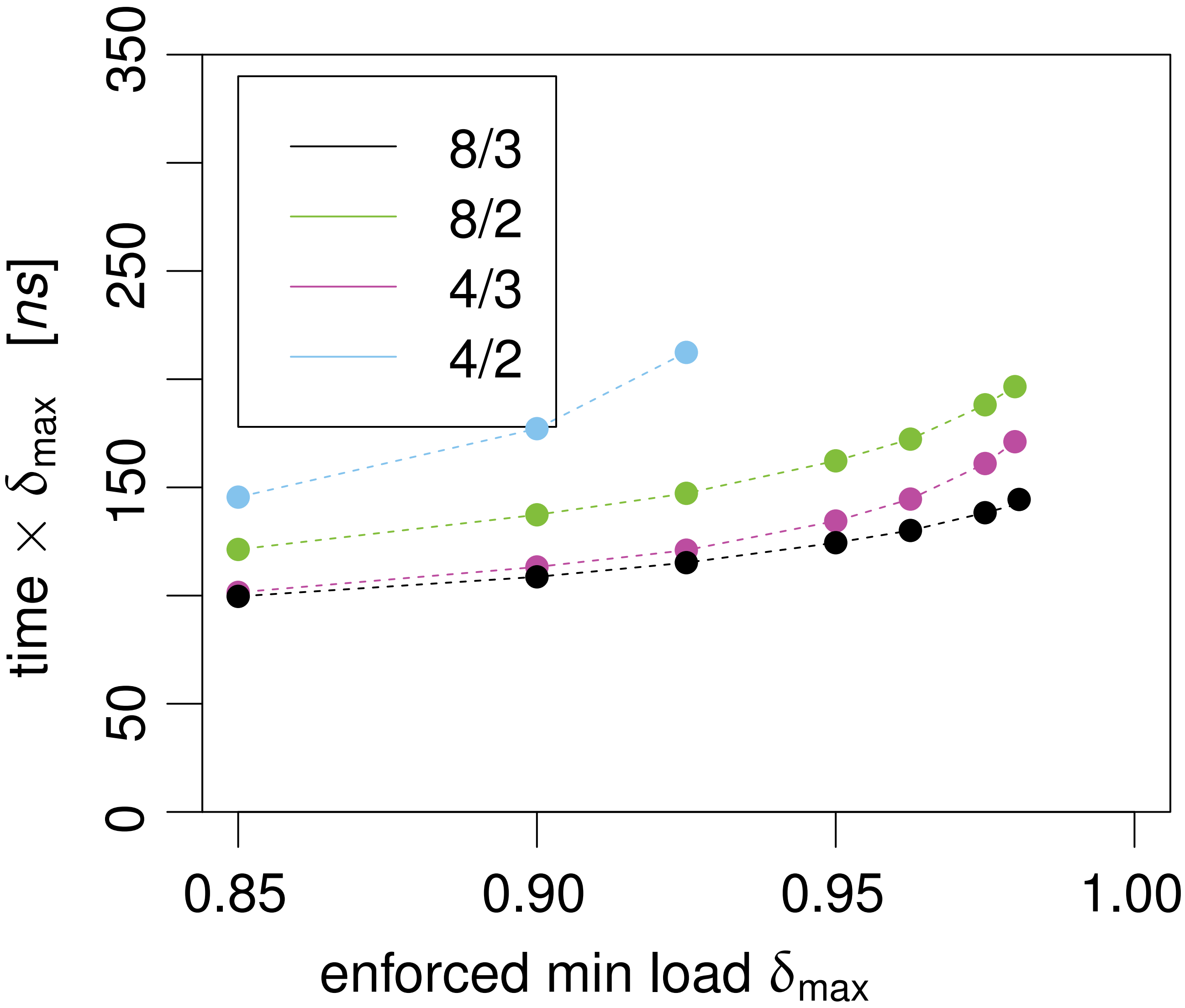
- constant lookups independent of fill ratio
- element → const. number possible cells
- if all cells are full, move existing elements
  - breadth-first-search
  - d alternative buckets per element $h_1(k), \ldots, h_d(k)$
  - buckets of B cells



d-ary Bucket Cuckoo Hashing

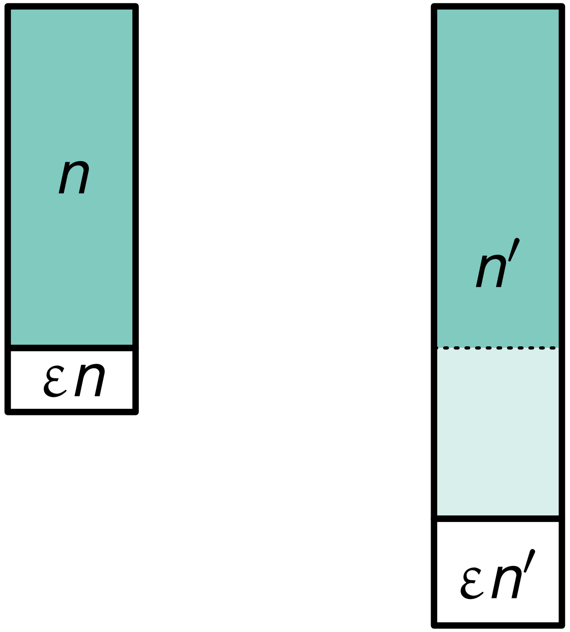
combination of different results, by:
[Pagh, Dietzfelbinger, Mehlhorn, Mitzenmacher, ...]

Space Efficient Hashing – Cuckoo Parameters



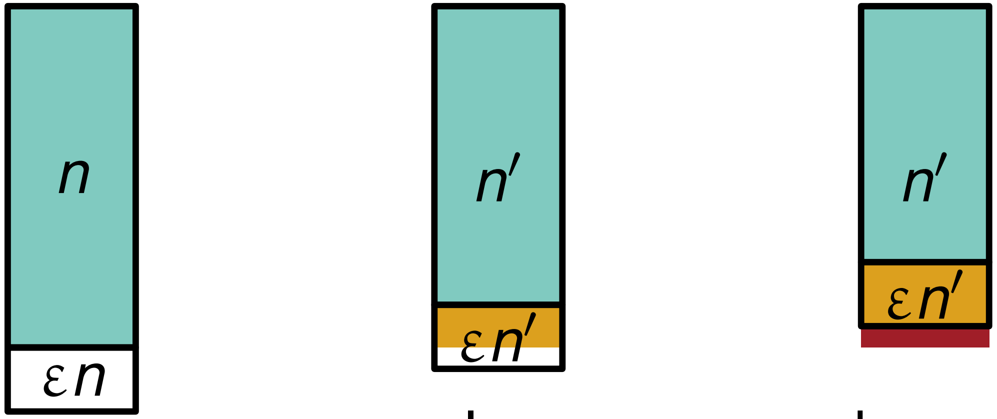
Space Efficient Hashing – Final Size Unknown

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



Space Efficient Hashing – Final Size Unknown

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



slow

needs growing

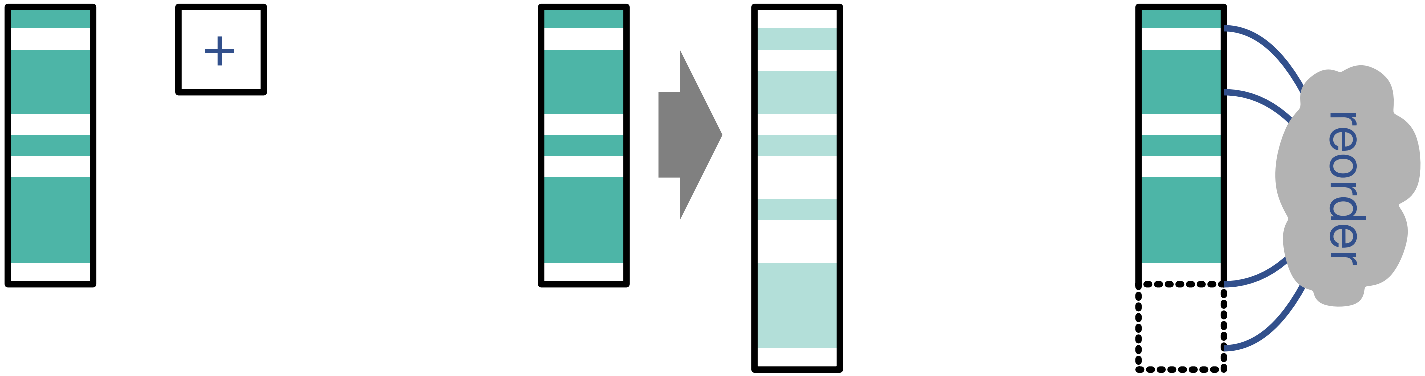
Space Efficient Hashing – Final Size Unknown

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

Space Efficient Hashing – Resizing

- growing has to be in small steps

- basic approaches



additional table

full migration

inplace+reorder

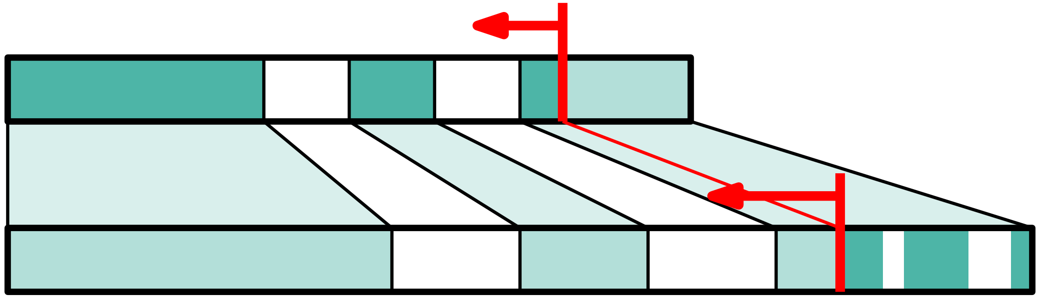
most common
in libraries

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

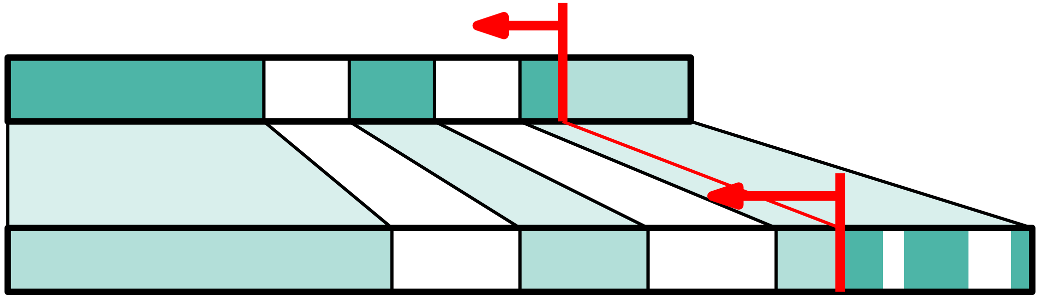


Secondary Contribution – Efficient Growing

- 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

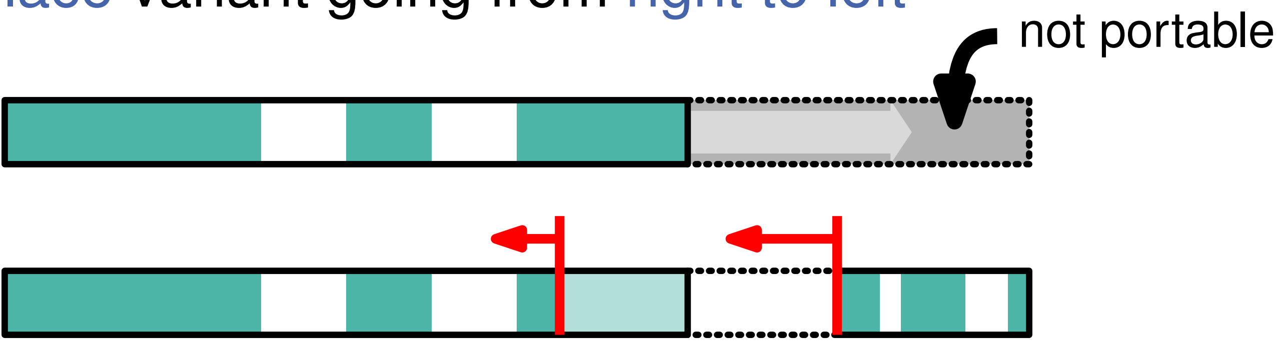


Secondary Contribution – Efficient Growing

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

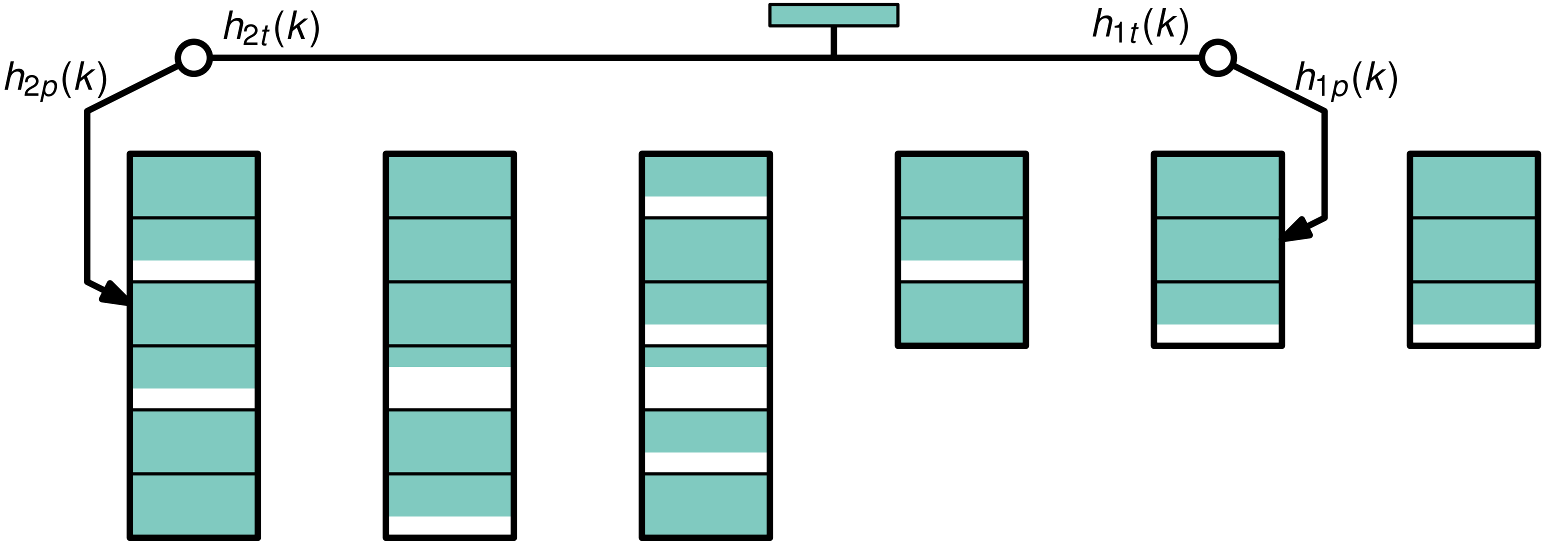
$$idx(k) = h(k) \cdot \frac{size}{maxHash + 1}$$

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



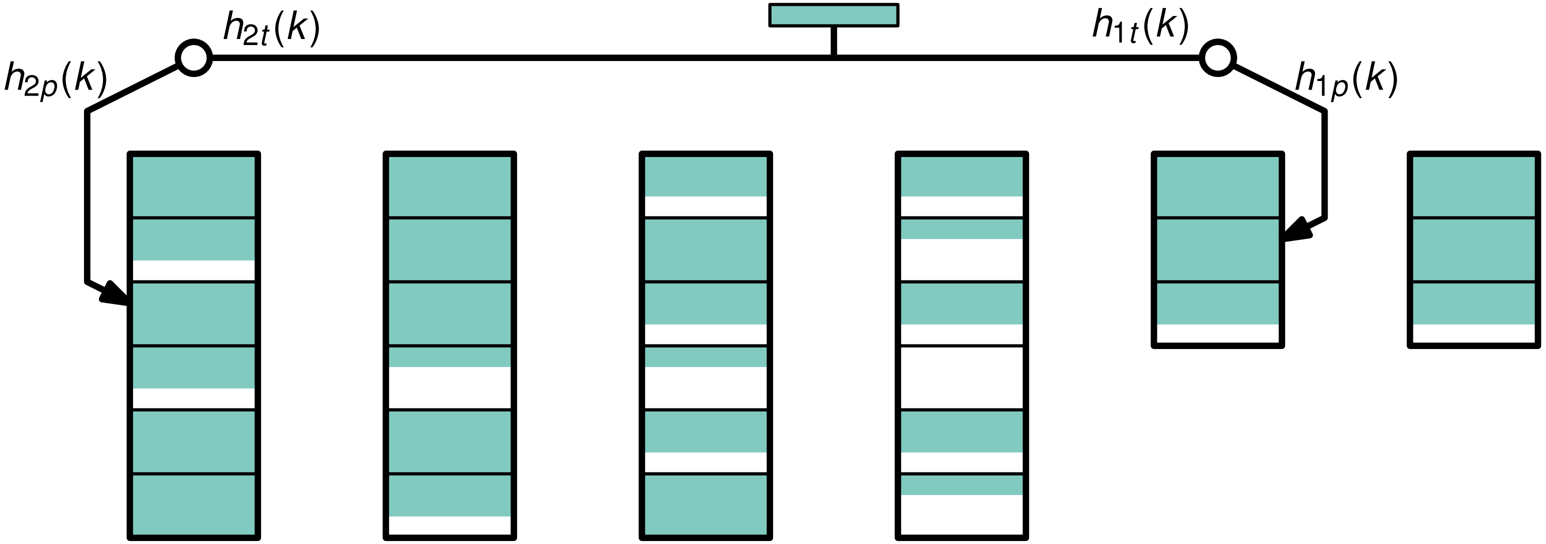
Contribution – Dynamic Space Efficient Cuckoo Table

- 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



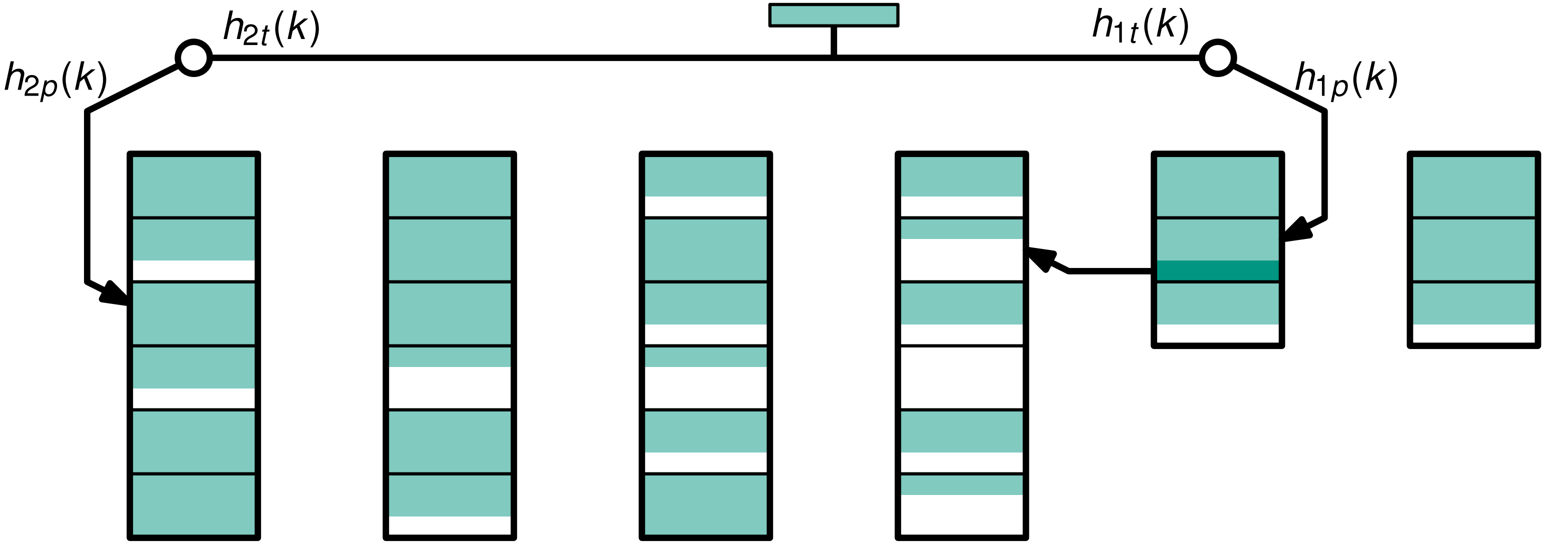
Contribution – Dynamic Space Efficient Cuckoo Table

- 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

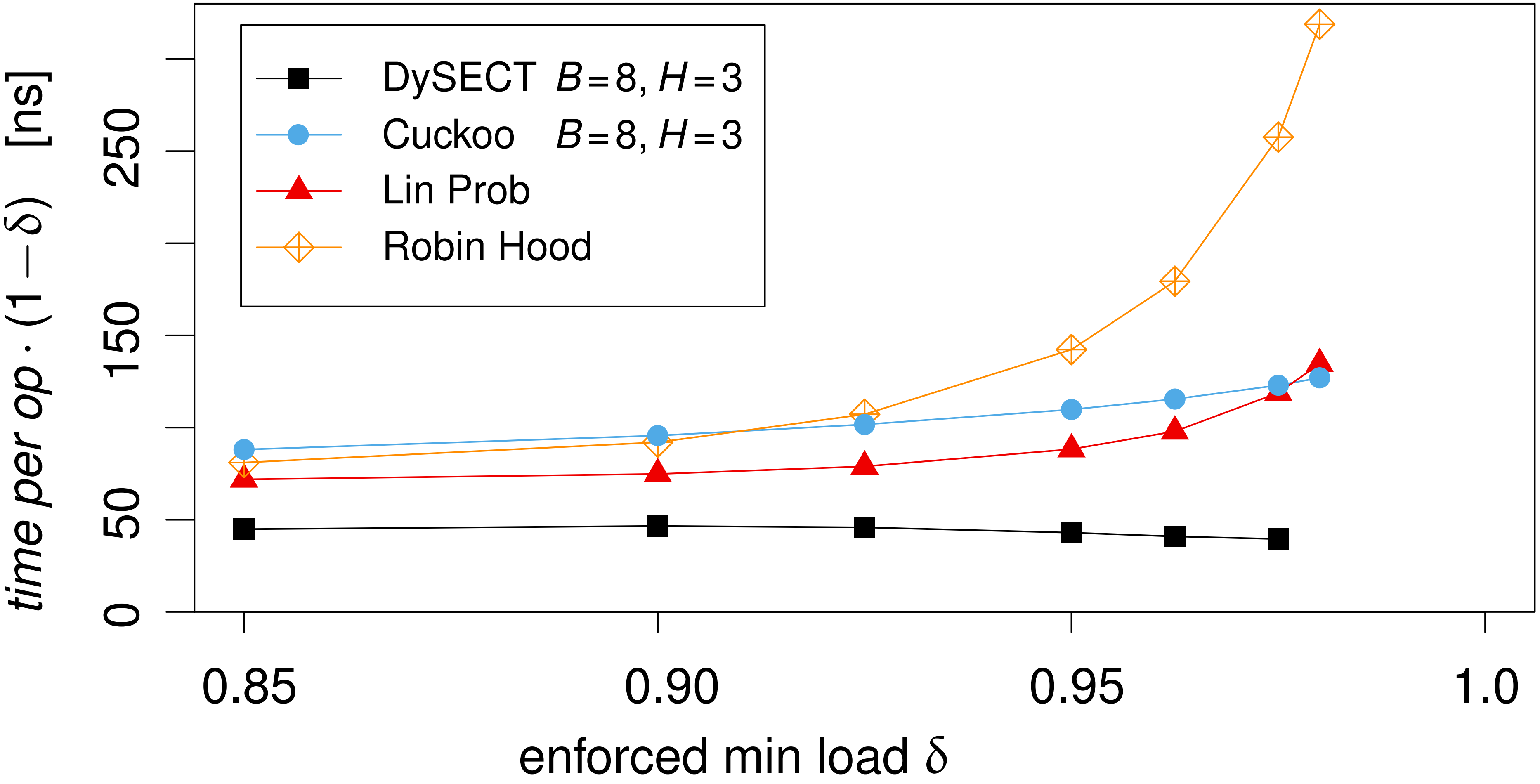


Contribution – Dynamic Space Efficient Cuckoo Table

- 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

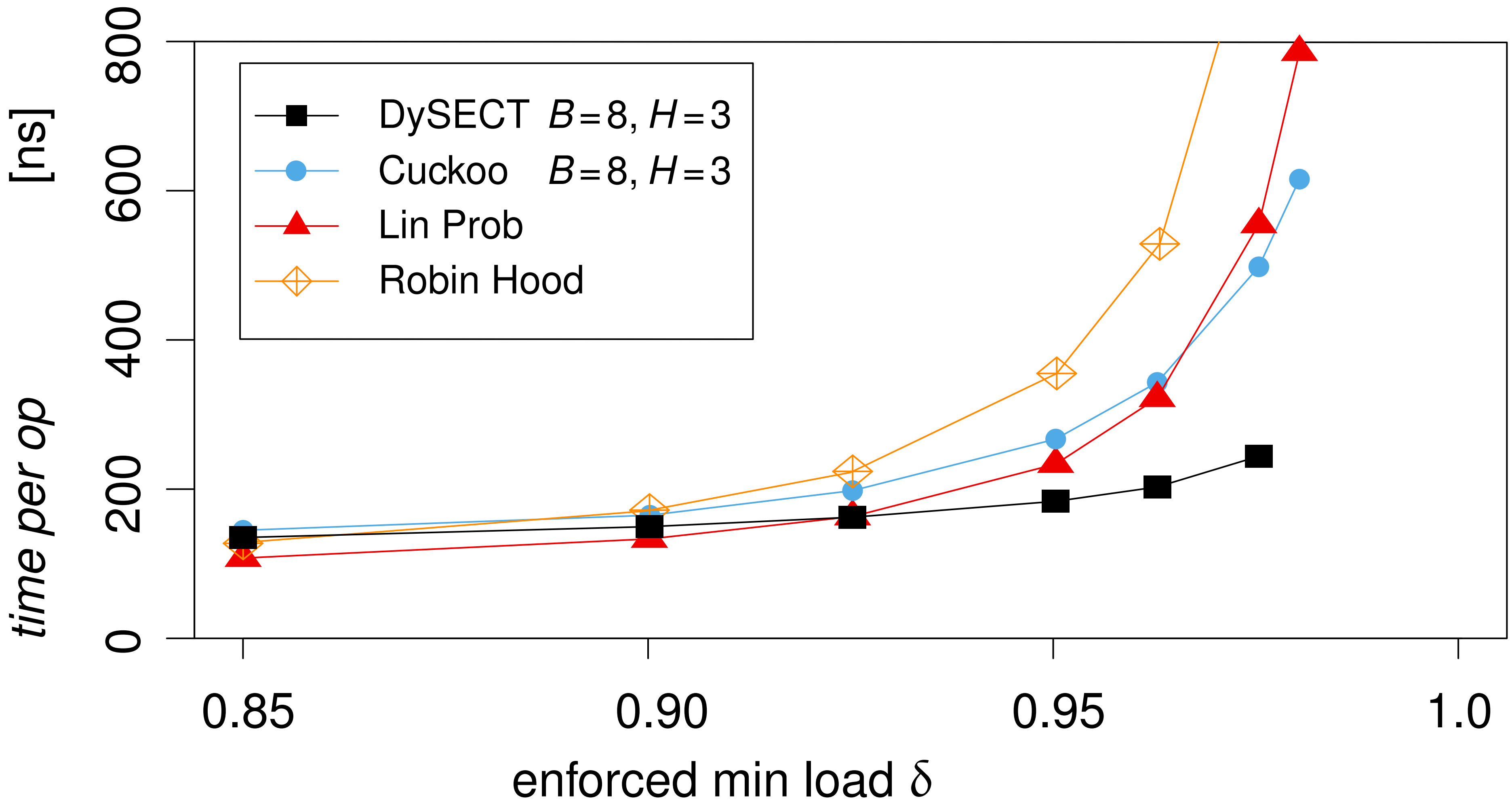


Result – Insertion into Growing Table



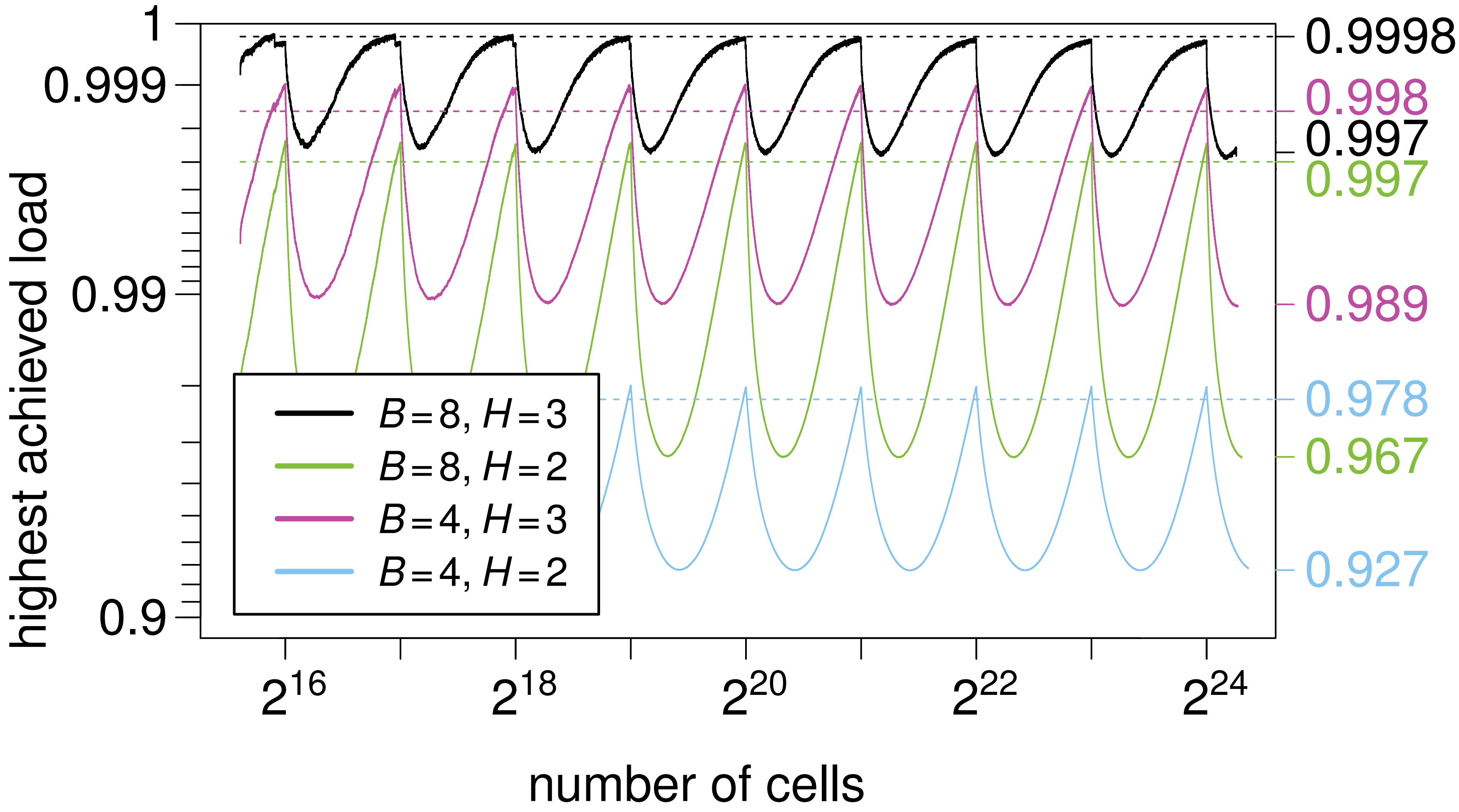
- $\frac{1}{1-\delta}$ “expected time” per insertion

Result – Word Count Benchmark



- CommonCrawl (avg. 12×)
- not normalized

Result – Load Bound



- we are in cooperation to prove bounds

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`