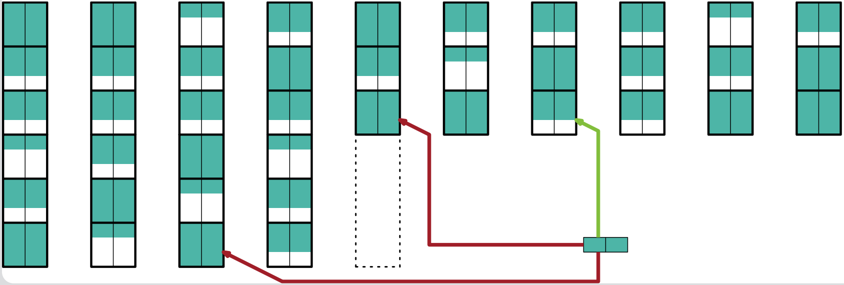
Karlsruhe Institute of Technology

Dynamic Space Efficient Hash Tables

Tobias Maier, Peter Sanders

ITI AG Sanders



Peter Sanders

General: Algorithm engineering, basic algorithmic toolbox, graphs, parallel algorithms, big data, randomized algorithms

Hashing Related Previous Work up to 2017

- *d*-ary cuckoo hashing — Fotakis, Pagh, S, Spirakis 03
- analysis of 2-way bucket cuckoo hashing — S, Egner, Korst 00
- fast construction for the above — Cain, S, Wormald 07
- cache-, hash-, and space-efficient Bloom filters — Putze, S, Singler 07
- perfect hashing applied to model checking — Edelkamp, S, Simecek 08
- fast retrieval and perfect hashing using fingerprinting — S, Zhou,[. . .] 14
- hashing vs sorting for aggregation in column-based DB — with SAP 15
- concurrent hash tables — Maier, S, Dementiev 16
- space efficient dynamic hash tables — Maier, S 17

Overview

- the problem and why standard solutions do not work
- simple solutions
- DySECT – **Dy**namic **S**pace **E**fficient **C**uckoo **T**able

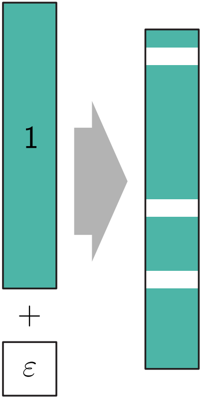
ESA 2017 and Algorithmica 2019 (with Stefan Walzer)

What we want

- constant amortized time insert, find, erase
- space close to lower bound (just the elements)
  load factor $\delta = \frac{1}{1+\epsilon}$ for small ϵ
- good constant factors

nice to have

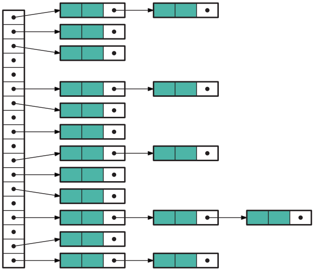
- worst case constant time find
- whp constant time insert



Hashing with Chaining?

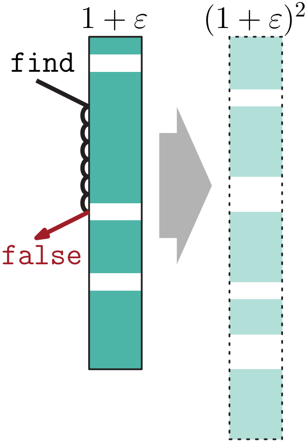
KIT
Karlsruhe Institute of Technology

+ grows dynamically and "smoothly"
– overhead for pointers
– eventually needs to grow basic table



Linear Probing?

- \+ can in principle be arbitrarily full
- \+ no overhead for pointers etc.
- \+ cache efficient
- – reallocate when full
  ⇒temporarily at least doubles space consumption
  (during the migration)
- – slow insert, erase and unsuccessful find when near full



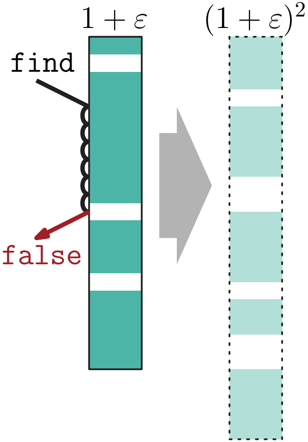
Engineering Linear Probing

Modulo Operations

- mapping (hash value → table index)
  usual: `idx(k) = hash(k) % cap`
  for `cap` = 2^k: `idx(k) = hash(k) & (cap-1)`
- circular vs. non-circular

Mapping by Scaling

- new: `idx(k) = hash(k) *` $\frac{\texttt{cap}}{\texttt{max\_hash + 1}}$
- different for circular tables



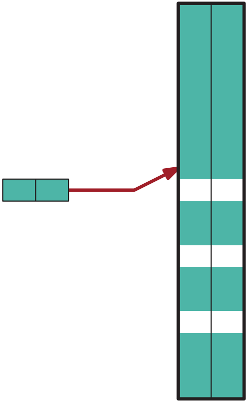
Using Rehashing for Collisions

- Recompute alternative cells using additional hash functions.
- Do this until you find a free cell

\+ shorter search distances

− disadvantages similar to linear probing

− less cache efficient



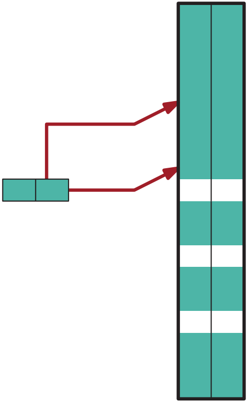
Using Rehashing for Collisions

- Recompute alternative cells using additional hash functions.
- Do this until you find a free cell

\+ shorter search distances

— disadvantages similar to linear probing

— less cache efficient



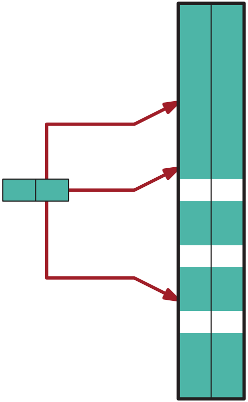
Using Rehashing for Collisions

- Recompute alternative cells using additional hash functions.
- Do this until you find a free cell

\+ shorter search distances

− disadvantages similar to linear probing

− less cache efficient



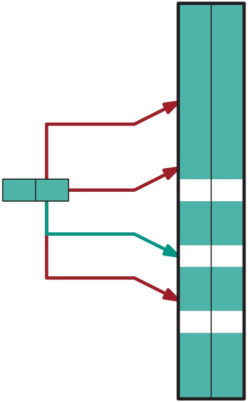
Using Rehashing for Collisions

- Recompute alternative cells using additional hash functions.
- Do this until you find a free cell

\+ shorter search distances

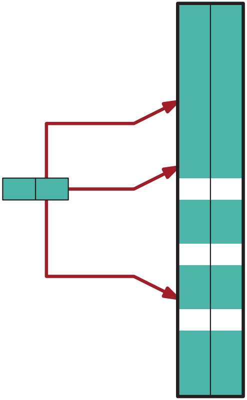
− disadvantages similar to linear probing

− less cache efficient



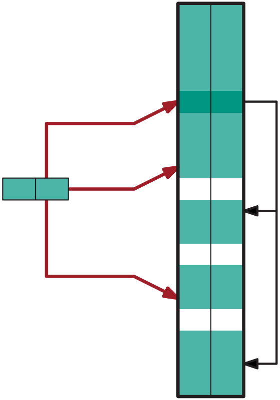
Cuckoo Hashing

- Similar to rehashing
- **Move items to reduce hash functions**



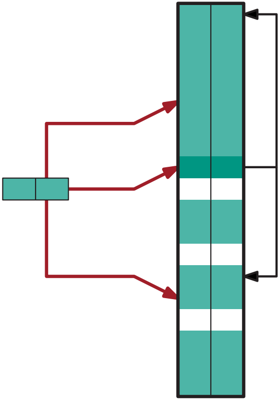
Cuckoo Hashing

- Similar to rehashing
- **Move items to reduce hash functions**



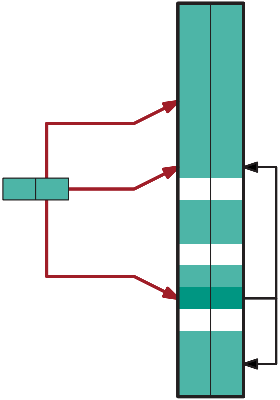
Cuckoo Hashing

- Similar to rehashing
- **Move items to reduce hash functions**



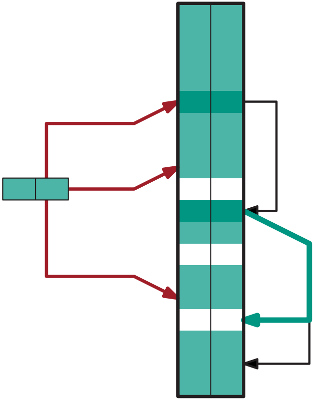
Cuckoo Hashing

- Similar to rehashing
- **Move items to reduce hash functions**



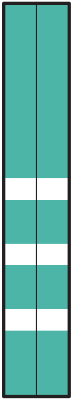
Cuckoo Hashing

- Similar to rehashing
- **Move items to reduce hash functions**



Cuckoo Hashing

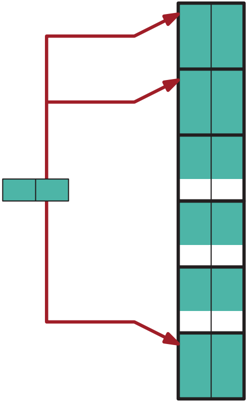
- Similar to rehashing
- **Move items to reduce hash functions**



H-ary Bucket Cuckoo Hashing

based on
Pagh Rodler 01, Fotakis Pagh S Spirakis 03, Dietzfelbinger Weidling 05

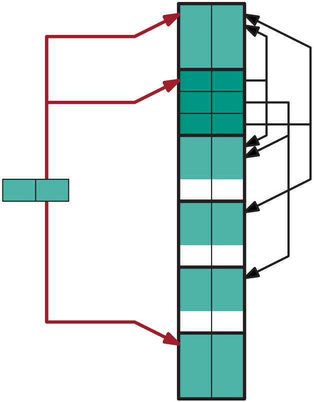
- *H* hash functions address *H* buckets
- buckets can store *B* elements each
- insert can move elements around (BFS or random walk)



H-ary Bucket Cuckoo Hashing

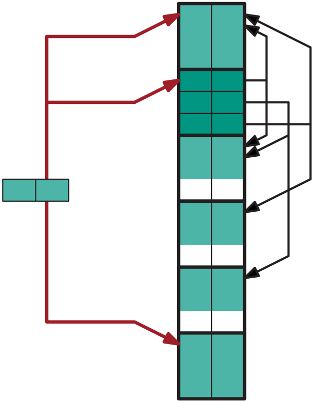
based on
Pagh Rodler 01, Fotakis Pagh S Spirakis 03, Dietzfelbinger Weidling 05

- *H* hash functions address *H* buckets
- buckets can store *B* elements each
- insert can move elements around (BFS or random walk)



H-ary Bucket Cuckoo Hashing

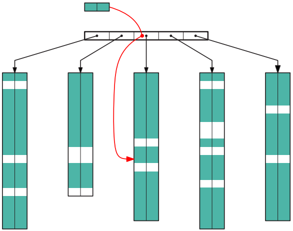
- \+ highly space efficient even for $H = 2$, $B = 4$
- \+ worst case constant find, erase
- \+ empirically $\approx 1/\epsilon$ average insertion time when not too close to capacity limit
- – reallocate when full



Folklore (?): The Subtable Trick

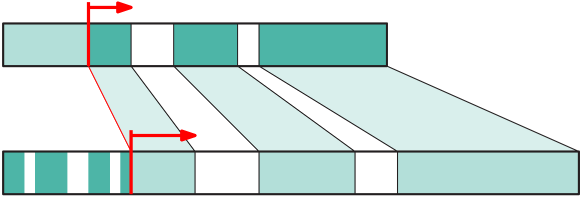
most significant bits of hash address one of T subtables

+ reallocation space overhead affects only a single subtable
+ low overhead for small T when upper level fits into cache
+ works for linear probing and cuckoo
− frequent reallocations lead to expensive insertions
− worst case insertion time determined by subtable reallocation
− danger of memory fragmentation with many different subtable sizes (past and present)

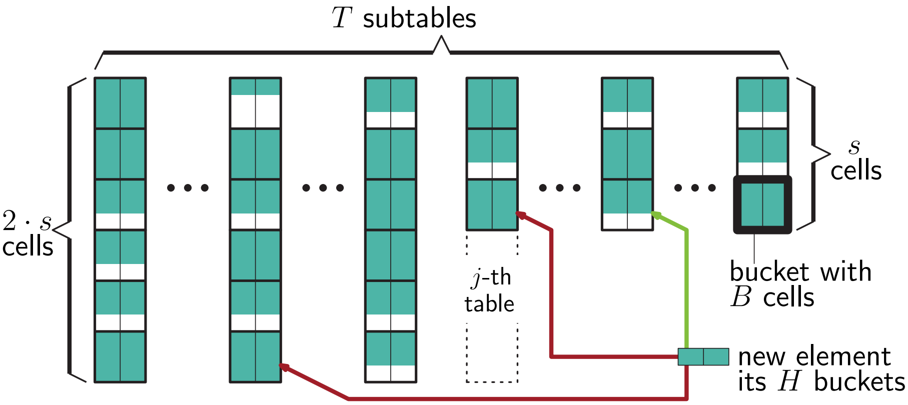


Mitigation: Cache Efficient Reallocation

- interpret bits of hash functions as number in $[0, 1)$
- scale to actual table size by multiplication
- reallocation “essentially” becomes a sweep through memory

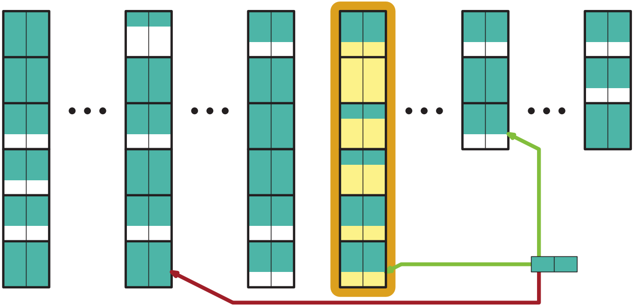


DySECT – Dynamic Space Efficient Cuckoo Table



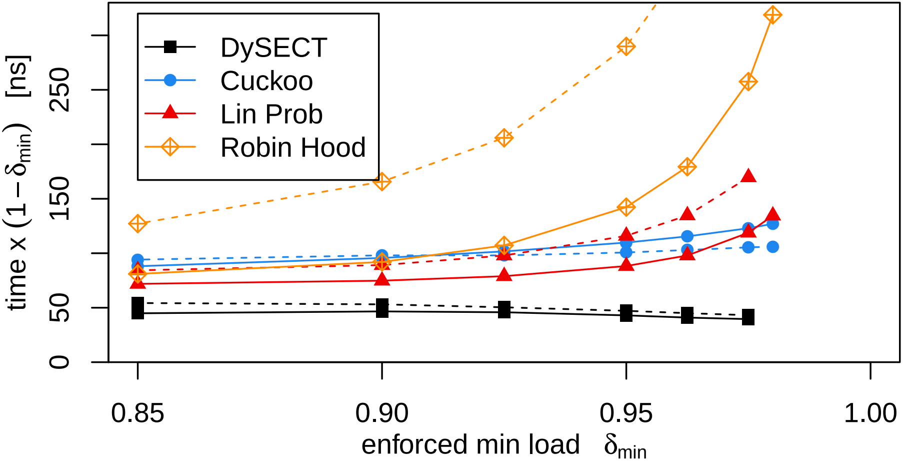
DySECT

- inherits most advantages from ordinary cuckoo –
  worst case constant find/erase, space efficiency (?), fast insert
- elements are migrated rarely $\rightsquigarrow$ fast insert
- subtable sizes are powers of two $\rightsquigarrow$ no fragmentation
- reallocation in small increments for large T
  $\rightsquigarrow$ constant insertion time whp when $T = \Omega(n)$



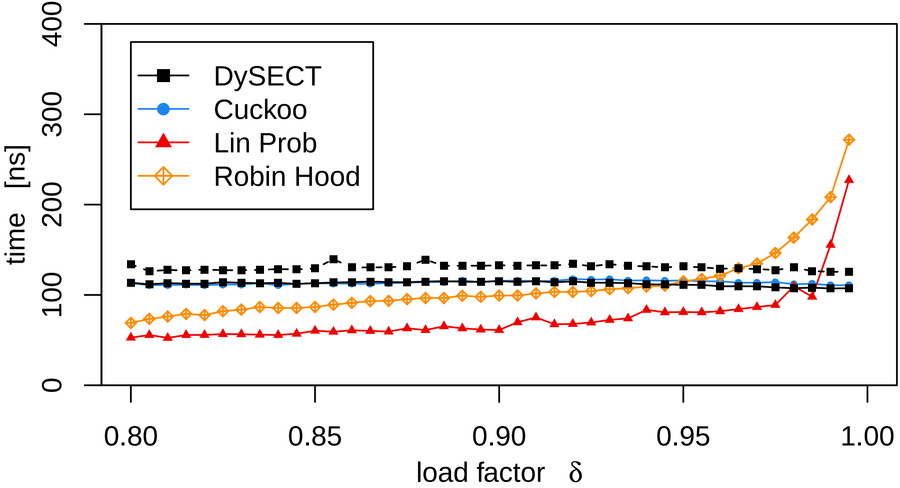
Dynamic Insertion Time

$(H = 3, B = 8)$



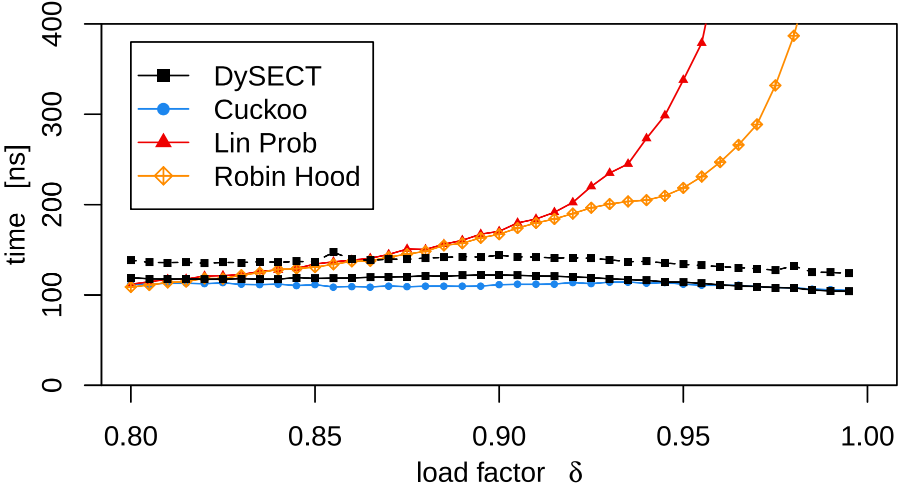
Successful Find

$(H = 3, B = 8)$

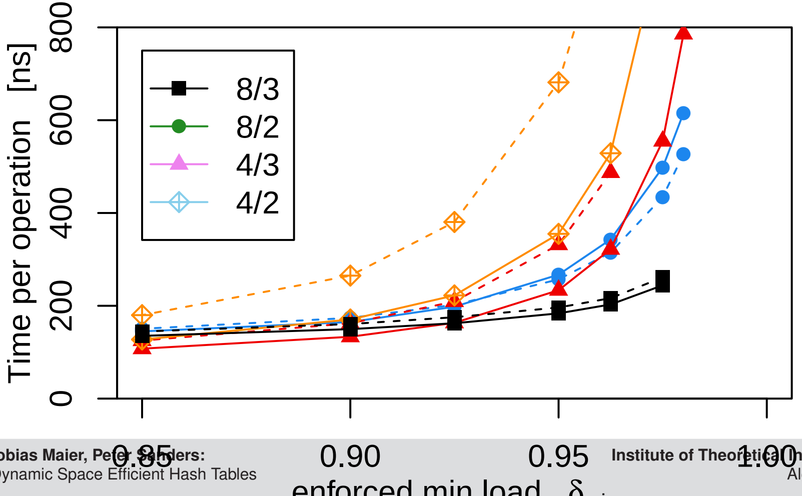


Unsuccessful Find

$(H = 3, B = 8)$



Wordcount Mini-Benchmark $(H = 3, B = 8)$



Summary

- first (?) “truly” space efficient dynamic hash tables
- subtables help (once more)
- scaling allows cache-efficient reallocation
- virtual memory overallocation helps (but not needed for DySECT)
- DySECT allows fast and non-amortized insertion