Dynamic Space Efficient Hash Tables

Tobias Maier, Peter Sanders
General: Algorithm engineering, basic algorithmic toolbox, graphs, parallel algorithms, big data, randomized algorithms

Hashing Related Previous Work
- \textit{d}-ary cuckoo hashing
- analysis of 2-way bucket cuckoo hashing
- fast construction for the above
- cache-, hash-, and space-efficient Bloom filters
- perfect hashing applied to model checking
- fast retrieval and perfect hashing using fingerprinting
- hashing vs sorting for aggregation in column-based DB
- concurrent hash tables
- space efficient dynamic hash tables

Fotakis, Pagh, S, Spirakis 03
S, Egner, Korst 00
Cain, S, Wormald 07
Putze, S, Singler 07
Edelkamp, S, Simecek 08
S, Zhou, [... ] 14
with SAP 15
Maier, S, Dementiev 16
Maier, S 17
Overview

- the problem and why standard solutions do not work
- simple solutions
- DySECT – Dynamic Space Efficient Cuckoo Table

Submitted to ESA, on arxiv tomorrow
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 $\epsilon$
- good constant factors

nice to have

- worst case constant time find
- whp constant time insert
Hashing with Chaining?

- 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
Modulo Operations
- mapping (hash value → table index)
  usual: \( \text{idx}(k) = \text{hash}(k) \mod \text{cap} \)
  for \( \text{cap} = 2^k \): \( \text{idx}(k) = \text{hash}(k) \& (\text{cap} - 1) \)
- circular vs. non-circular

Mapping by Scaling
- new: \( \text{idx}(k) = \text{hash}(k) \times \frac{\text{cap}}{\text{max_hash} + 1} \)
- 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 numbers in $[0, 1)$
- Scale to actual table size by multiplication
- Reallocation “essentially” becomes a sweep through memory
DySECT – Dynamic Space Efficient Cuckoo Table

\[ T \text{ subtables} \]

\[ 2 \cdot s \text{ cells} \]

\[ j\text{-th table} \]

\[ \text{bucket with } B \text{ cells} \]

\[ s \text{ cells} \]

\[ \text{new element its } H \text{ buckets} \]
DySECT

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

\( (H = 3, B = 8) \)

- DySECT
- Cuckoo
- Lin Prob
- Robin Hood

\[ \text{time} \times (1 - \delta_{\text{min}}) \] [ns]

\( \text{enforced min load} \quad \delta_{\text{min}} \)
Successful Find

\[(H = 3, B = 8)\]

![Graph showing load factor vs. time for different hash table approaches]

- DySECT
- Cuckoo
- Lin Prob
- Robin Hood

Tobias Maier, Peter Sanders:
Dynamic Space Efficient Hash Tables

Institute of Theoretical Informatics
Algorithmics
Unsuccessful Find

\((H = 3, B = 8)\)

![Graph showing the time for unsuccessful find (ns) as a function of load factor (δ) for different hash table algorithms: DySECT, Cuckoo, Lin Prob, and Robin Hood.](image)
Wordcount Mini-Benchmark \((H = 3, B = 8)\)

![Graph showing time per operation against enforced min load \(\delta_{\text{min}}\) for different hash table implementations: DySECT, Cuckoo, Lin Prob, and Robin Hood. The graph plots time per operation in nanoseconds (ns) on the y-axis against enforced min load on the x-axis. The x-axis ranges from 0.85 to 1.00, and the y-axis ranges from 0 to 800.](image)
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