Hashing

Lecture · 11. June 2019
Tobias Maier and Peter Sanders
Hash Tables – Definitions

- set \( S \subseteq U = \text{Keys} \times \text{Values} \)
  - 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

\[ \langle \text{key}, \text{value} \rangle \]

**hash function**

\[ h(\cdot) \]

**mapping**

\[ h(\text{key}) \]
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 $\Omega$
- 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 $\text{Keys} \mapsto \{0..m - 1\}$
- 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 $\forall h \in \Omega : p_h = \frac{1}{m^{|\text{Keys}|}}$
- $\mathbb{P}[\varepsilon_{k_1,k_2}] = 1/m^*$
- #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 $\Omega$
- Events $\varepsilon \subset \Omega$
- Probability $p_x$ of $x \in \Omega$
- Probability of an event $P[\varepsilon] = \sum_{x \in \varepsilon} p_x$
- 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 $\forall h \in \Omega : p_h = \frac{1}{m|Keys|}$
- $P[\varepsilon_{k_1,k_2}] = 1/m^*$
- #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 \( X_e \)

\[ X_e = \begin{cases} 
1 & \text{if } h(e) = 0 \\
0 & \text{otherwise}
\end{cases} \]

\[ E[X_0] = E \left[ \sum_{e \in S} X_e \right] = \sum_{e \in S} E[X_e] \]
\[ = \sum_{e \in S} P[X_e = 1] = \frac{n}{m} \]
Hash Tables – Other Bounds

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

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

\[ \delta = \frac{m-n}{m} \]
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 $\rightarrow$ 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 $\rightarrow$ 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 $\rightarrow$ 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 \(\rightarrow\) 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 → constant number possible cells
- If all cells are full, move existing elements
  - Breadth-first-search
- \(d\) alternative buckets per element
  \(h_1(k), \ldots, h_d(k)\)

\(d\)-ary Bucket Cuckoo Hashing

Combination of different results, by:

[Pagh, Dietzfelbinger, Mehlhorn, Mitzenmacher, ...]
Space Efficient Hashing – Cuckoo Hashing

- Constant lookups independent of fill ratio
- Each element maps to a constant number of possible cells
- If all cells are full, move existing elements:
  - Breadth-first-search
- D alternative buckets per element:
  \( h_1(k), \ldots, h_d(k) \)

**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 $\rightarrow$ 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 $\rightarrow$ 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

The diagram illustrates the relationship between the enforced minimum load \( \delta_{\text{max}} \) and the time multiplied by \( \delta_{\text{max}} \) in nanoseconds (ns) for different Cuckoo parameters: 8/3, 8/2, 4/3, and 4/2. The x-axis represents the enforced minimum load, and the y-axis shows the time \( \times \delta_{\text{max}} \) in nanoseconds. The graph shows how the time increases as the enforced minimum load increases for each parameter.
Space Efficient Hashing – Final Size Unknown

- conservative estimate
  
- strict bound might not be reasonable

\[ n \leq n' \]

- less space efficient

\[ n \leq n' \]

\[ \epsilon n \]

\[ \epsilon n' \]
Space Efficient Hashing – Final Size Unknown

- conservative estimate
- optimistic estimate
  - might overfill
  - needs growing strategy

\[ n \approx n' \]

- 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

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
    \[ idx(k) = h(k) \cdot \frac{\text{size}}{\text{maxHash} + 1} \]
- 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
  \[
  \text{idx}(k) = h(k) \cdot \frac{\text{size}}{\text{maxHash} + 1}
  \]
- 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**

\[
idx(k) = h(k) \cdot \frac{size}{\text{maxHash} + 1}
\]

- **very fast migration** due to cache efficiency

- **inplace** variant going from right to left

  ![Diagram of inplace variant going from right to left]

  not portable
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
use subtables of unequal size (use powers of 2)

- $h_i(k)$ ⇒ $h_{it}(k)$ table and $h_{ip}(k)$ position in table
- doubling one subtable ⇔ small overall factor

use displacements to equalize load imbalance
Result – Insertion into Growing Table

![Graph showing time per operation vs. enforced min load](image)

- **DySECT** $B=8, H=3$
- **Cuckoo** $B=8, H=3$
- **Lin Prob**
- **Robin Hood**

**Expected time** per insertion:

$$\frac{1}{1-\delta}$$
Result – Word Count Benchmark

![Graph showing the comparison of time per operation against enforced min load δ for different hash table implementations.]

- DySECT $B=8$, $H=3$
- Cuckoo $B=8$, $H=3$
- Lin Prob
- Robin Hood

- CommonCrawl (avg. $12 \times$)
- 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](https://github.com/TooBiased/DySECT)