Algorithms and Data Structures

Open Hashing

Ulf Leser
Open Hashing

- **Open Hashing**: Store all values inside hash table A
- **General framework**
  - No collision: Business as usual
  - Collision: Chose another index and probe again (is it “open”?)
  - As second index might be full as well, probing must be iterated
- **Many suggestions on how to chose the next probe index**
- **In general, we want a strategy (probe sequence) that**
  - ... ultimately visits any index in A (and few twice before)
  - ... is deterministic – when searching, we must follow the same order of indexes (probe sequence) as for inserts
Reaching all Indexes of A

• Definition

Let $A$ be a hash table, $|A|=m$, over universe $U$ and $h$ a hash function for $U$ into $A$. Let $I=\{0, ..., m-1\}$. A probe sequence is a deterministic, surjective function $s: U \times I \rightarrow I$

• Remarks

– We use $j$ to denote elements of the sequence: Where to jump after $j-1$ probes
– $s$ need not be injective – a probe sequences may cross itself
  • But it is better if it doesn’t
– We typically use $s(k, j) = (h(k) - s'(k, j)) \mod m$ for a properly chosen function $s'$

• Example: $s'(k, j) = j$, hence $s(k, j) = (h(k) - j) \mod m$
Searching

- Let $s'(k, 0) := 0$
- We assume that $s$ cycles through all indexes of $A$
  - In whatever order
- Probe sequences longer than $m-1$ usually make no sense, as they necessarily look into indexes twice
  - But beware of non-injective functions

```c
1. func int search(k int) {
2.   j := 0;
3.   first := h(k);
4.   repeat
5.     pos := (first-s'(k, j) mod m;
6.     j := j+1;
7.   until (A[pos]=k) or
8.     (A[pos]=null) or
9.     (j=m)
10.   if (A[pos]=k then
11.     return pos;
12.   else
13.     return -1;
14. end if;
15.}
```
Deletions

- Deletions are a problem
  - Assume \( h(k) = k \mod 11 \) and \( s(k, j) = (h(k) + 3\times j) \mod m \)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>ins(1); ins(6)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ins(23)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>23</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ins(12)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>23</td>
<td></td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>del(23)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>search(12)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>?</td>
<td>6</td>
<td>12</td>
</tr>
</tbody>
</table>
Remedies

• **Leave a mark** (tombstone)
  – During search, jump over tombstones
  – During insert, tombstones may be replaced

• **Re-organize list**
  – Keep pointer p to index where a key should be deleted
  – Walk to end of probe sequence (first empty entry)
  – Move *last non-empty entry* to index p
  – Requires to completely run through the probe sequence for every deletion (otherwise only n/2 on average)
  – **Not compatible** with strategies that keep probe sequences sorted
    • See later
Open versus External collision handling

• Pro
  – We do not need more space than reserved – more predictable
  – A typically is filled more homogeneously – less wasted space

• Contra
  – More complicated
  – Depending on method, we get worse average-case / worst-case complexities for insertion/deletion/sort
    • Especially deletions have overhead
  – A gets full; we cannot go beyond $\alpha = 1$
  – If A gets very large, we can elegantly store overflow chains on external memory
Overview

- We will look into **three strategies**
  - **Linear probing**: \( s(k, j) := (h(k) - j) \mod m \)
  - **Double hashing**: \( s(k, j) := (h(k) - j \cdot h'(k)) \mod m \)
  - **Ordered hashing**: Any \( s \); values in probe sequence are kept sorted

- **Others**
  - **Quadratic hashing**: \( s(k, j) := (h(k) - \text{floor}(j/2)^2 \cdot (-1)^j) \mod m \)
    - Less vulnerable to local clustering than linear hashing
  - **Uniform hashing**: \( s \) is a random permutation of \( I \) dependent on \( k \)
    - High administration overhead, guarantees shortest probe sequences
  - **Coalesced hashing**: \( s \) arbitrary; entries are linked by add. pointers
    - Like overflow hashing, but overflow chains are in \( A \); needs additional space for links
Content of this Lecture

- Open Hashing
  - Linear Probing
  - Double Hashing
  - Ordered Hashing
Linear Probing

- Probe sequence function: \( s(k, j) := (h(k) - j) \mod m \)
  - Assume \( h(k) = k \mod 11 \)

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\end{array}
\]

\[\text{ins(1); ins(7); ins(13)}\]

\[\begin{array}{cccccccccc}
& & 1 & 13 & & & & & & 7 & \\
\end{array}\]

\[\text{ins(23)}\]

\[\begin{array}{cccccccccc}
23 & 1 & 13 & & & & & & 7 & \\
\end{array}\]

\[\text{ins(12)}\]

\[\begin{array}{cccccccccc}
23 & 1 & 13 & & & & & & 7 & 12 \\
\end{array}\]

\[\text{ins(10)}\]

\[\begin{array}{cccccccccc}
23 & 1 & 13 & & & & & & 7 & 10 & 12 \\
\end{array}\]

\[\text{ins(24)}\]

\[\begin{array}{cccccccccc}
23 & 1 & 13 & & & & & & 7 & 24 & 10 & 12 \\
\end{array}\]
Analysis

- The longer a chain ...
  - the more different values of h(k) it covers
  - the higher are the chances to produce more collisions
  - the faster it will grow, the faster it will merge with other chains

- Assume an empty position p left of a chain of length n and an empty position q with an empty cell to the right
  - Also assume h is uniform
  - Chances to fill q with next insert: 1/m
  - Chances to fill p with the next insert: n/m

- Linear probing tends to quickly produce long, completely filled stretches of A with high collision probabilities
In Numbers (Derivation of Formulas Skipped)

- Scenario: Some inserts, then many searches
  - Expected number of probes per search are most important

\[ C_n \approx \frac{1}{2} \left( 1 + \frac{1}{(1 - \alpha)} \right) \]

\[ C'_n \approx \frac{1}{2} \left( 1 + \frac{1}{(1 - \alpha)^2} \right) \]

<table>
<thead>
<tr>
<th>( \alpha )</th>
<th>( C_n ) (erfolgreich)</th>
<th>( C'_n ) (erfolglos)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.50</td>
<td>1.5</td>
<td>2.5</td>
</tr>
<tr>
<td>0.90</td>
<td>5.5</td>
<td>50.5</td>
</tr>
<tr>
<td>0.95</td>
<td>10.5</td>
<td>200.5</td>
</tr>
<tr>
<td>1.00</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Source: S. Albers / [OW93]
Quadratic Hashing

erfolgreiche Suche:

\[
C_n \approx 1 - \frac{\alpha}{2} + \ln\left(\frac{1}{1 - \alpha}\right)
\]

erfolglose Suche:

\[
C'_n \approx \frac{1}{1 - \alpha} - \alpha + \ln\left(\frac{1}{1 - \alpha}\right)
\]

<table>
<thead>
<tr>
<th>(\alpha)</th>
<th>(C_n) (erfolgreich)</th>
<th>(C'_n) (erfolglos)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.50</td>
<td>1.44</td>
<td>2.19</td>
</tr>
<tr>
<td>0.90</td>
<td>2.85</td>
<td>11.40</td>
</tr>
<tr>
<td>0.95</td>
<td>3.52</td>
<td>22.05</td>
</tr>
<tr>
<td>1.00</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Source: S. Albers / [OW93]
Discussion

- Disadvantage of linear (and quadratic) hashing: Problems with the original hash function $h$ are preserved
  - Probe sequence only depends on $h(k)$, not on $k$
    - $s'(k, j)$ ignores $k$
  - All synonyms $k$, $k'$ will create the same probe sequence
    - Two keys that form a collision are called synonyms
  - Thus, if $h$ tends to generate clusters (or inserted keys are non-uniformly distributed in $U$), also $s$ tends to generate “clusters” (i.e., sequences filled from multiple keys)
Content of this Lecture

- Open Hashing
  - Linear Probing
  - Double Hashing
  - Ordered Hashing
Double Hashing

- **Double Hashing**: Use a second hash function $h'$
  - $s(k, j) := (h(k) - j \cdot h'(k)) \mod m$ (with $h'(k) \neq 0$)
  - Further, we want that $-h'(k) \not| m$ (done if $m$ is prime)
- $h'$ should spread $h$-synonyms
  - If $h(k) = h(k')$, then hopefully $h'(k) \neq h'(k')$
    - Otherwise, we preserve problems with $h$
  - Optimal case: $h'$ statistically independent of $h$, i.e.,
    - $p(h(k) = h(k') \land h'(k) = h'(k')) = p(h(k) = h(k')) \cdot p(h'(k) = h'(k'))$
    - If both are uniform: $p(h(k) = h(k')) = p(h'(k) = h'(k')) = \frac{1}{m}$
- **Example**: If $h(k) = k \mod m$, then $h'(k) = 1 + k \mod (m-2)$
**Example** (Linear Probing produced 9 collisions)

- \( h(k) = k \mod 11; \quad h'(k) = 1 + k \mod 9 \)

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

- \( h(k) = 1; \quad h'(k) = 6 \)
  \[ s(k, 1) = 5 \]

- \( h(k) = 1; \quad h'(k) = 4 \)
  \[ s(k, 1) = 3 \]

- \( h(k) = 2; \quad h'(k) = 7 \)
  \[ s(k, 1) = 5 \]
  \[ s(k, 2) = 1 \]
  \[ s(k, 3) = 8 \]
Analysis

• Would need a lengthy proof

\[ C' \leq \frac{1}{1 - \alpha} \]

\[ C_n \approx \frac{1}{\alpha} \times \ln\left(\frac{1}{(1 - \alpha)}\right) \]

<table>
<thead>
<tr>
<th>(\alpha)</th>
<th>(C_n) (erfolgreich)</th>
<th>(C'_n) (erfolglos)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.50</td>
<td>1.39</td>
<td>2</td>
</tr>
<tr>
<td>0.90</td>
<td>2.56</td>
<td>10</td>
</tr>
<tr>
<td>0.95</td>
<td>3.15</td>
<td>20</td>
</tr>
<tr>
<td>1.00</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Another Example

ins(23); ins(13)

ins(34)
h(k)=1; h'(k)=8
s(k, 1)=7

ins( 12)
h(k)=1; h'(k)=4
s(k, 1)=3

ins( 10)

ins( 24)
h(k)=2; h'(k)=7
s(k, 1)=5
Observation

- We change the order of insertions (and nothing else)

\[
\begin{align*}
\text{ins}(34); \text{ins}(13) & : 34 & 13 & \quad 34 & 13 \\
\text{ins}(23) & : 34 & 13 & 23 \\
& h(k)=1; h'(k)=6 \\
& s(k, 1)=5 \\
\text{ins}(12) & : 34 & 13 & 12 & 23 \\
& h(k)=1; h'(k)=4 \\
& s(k, 1)=3 \\
\text{ins}(10) & : 34 & 13 & 12 & 23 & 10 \\
\text{ins}(24) & : 34 & 13 & 12 & 23 & 24 & 10 \\
& h(k)=2; h'(k)=7 \\
& s(k, 1)=5 \\
& s(k, 2)=1 \\
& s(k, 3)=8
\end{align*}
\]
Observation

- The number of collisions depends on the order of inserts
  - Because \( h' \) spreads \( h \)-synonyms differently for different values of \( k \)
- We cannot change the order of inserts, but ...
- Observe that when we insert \( k' \) and there already was a \( k \) with \( h(k)=h(k') \), we actually have two choices
  - Until now we always looked for a new place for \( k' \)
  - Why not: set \( A[h(k')]=k' \) and find a new place for \( k \)?
  - If \( s(k',1) \) is filled but \( s(k,1) \) is free, then the second choice is better
  - Insert is faster, searches will be faster on average
Brent’s Algorithm

- **Brent’s algorithm:**
  Upon collision, propagate key for which the next index in probe sequence is free; if both are occupied, propagate \( k' \)
- Improves only successful searches
  - Otherwise we have to follow the chain to its end anyway
- One can show that the average-case probe length for successful searches now is **constant** (\( \sim 2.5 \) accesses)
  - Even for relatively full tables
Content of this Lecture

- Open Hashing
  - Linear Probing
  - Double Hashing
  - Ordered Hashing
Idea

• Can we do something to improve unsuccessful searches?
  – Recall overflow hashing: If we keep the overflow chain sorted, we can stop searching after \( n/2 \) comparisons on average

• Transferring this idea: We must keep the keys in any probe sequence ordered
  – We have seen with Brent’s algorithm that we have the choice which key to propagate whenever we have a collision
  – Thus, we can also choose to always propagate the smaller of both keys – which generates a sorted probe sequence

• Result: Unsuccessful are as fast as successful searches
  – Note: This trick cannot be combined with Brent’s algorithm – conflicting rules
Details

- In Brent’s algorithm, we only replace a key if we can insert the replaced key directly into A.
- Now, we must replace keys even if the next slot in the probe sequence is occupied:
  - We run through probe sequence until we meet a key that is smaller.
  - We insert the new key here.
  - All subsequent keys must be replaced (moved in probe sequence).
- Note that this doesn’t make inserts slower than before:
  - Without replacement, we would have to search the first free slot.
  - Now we replace until the first free slot.
• Imagine ins(6) would first probe position 1, then 4
• Since 6<9, 9 is replaced; imagine the next slot would be 8
• Since 9<14, 14 is replaced

• Problem
  – 14 is not a synonym of 9 – two probe sequences cross each other
  – Thus, we don’t know where to move 14 – the next position in general requires to know the “j”, i.e., the number of hops that were necessary to get from h(14) to slot 8
Solution

- Ordered hashing only works if we can compute the next offset without knowing j
  - E.g. linear hashing (offset -1) or double hashing (offset \(-h'(k)\))
- But – is the method still correct?
  - Yes (for formal proof, see [OW93])
  - The critical points are where \textit{where probe sequences cross}
  - Imagine that we had a sequence X-Y-Z (with X<Y<Z). An insert triggers a replacement of Y with some Y'.
    - This implies that Y'<Y<Z (or no replacement had happened)
    - But we don’t know if X<Y’ – can this be a problem?
    - No – X and Y’ cannot be synonyms (or no crossing had happened)
    - Thus, we cannot enter the probe sequence of X with search key Y’
    - Since Y’<Y, Y’ cannot make a search break too early
Wrap-Up

- **Open hashing** can be a good alternative to overflow hashing even if the fill grade approaches 1
  - Very little average-case cost for look-ups with double hashing and Brent’s algorithm or using ordered hashing
    - Depending which types of searchers are more frequent
- Open hashing suffers from having only static place, but guarantees to not request more space once A is allocated
  - Less memory fragmentation
Dynamic Hashing

- **Dynamic Hashing** adapts the size of the hash table
  - Once fill degree exceeds (falls under) a threshold, increase (decrease) table size

- Used a lot in databases
  - Hash table in main memory, all synonyms in one disc block
  - We increase hash table when synonym block overflows

- Main problem: **Avoid rehashing**
  - Even if |A| increases, our original hash function (using m) will never address the new slots
  - Undesirable: Create new hash function and rehash all values

- Linear hashing, extensible hashing, virtual hashing, ...