Algorithms and Data Structures

Self-Organizing Lists

Ulf Leser
Assumptions for Searching

- Until now, we implicitly assumed that every element of our list is \textit{searched with the same probability}, i.e., with the same frequency.
- Accordingly, we treated all elements equal and tried to reduce the worst-case runtime for all elements.
- We may sort the list by \textit{properties of its elements}, but we never considered \textit{properties of their usage}.
- This setting sometimes is inadequate.
# Searches on the Web  [Germany, 2010, Google Zeitgeist]

<table>
<thead>
<tr>
<th>Schnellst wachsende Suchbegriffe</th>
<th>Die häufigsten Suchbegriffe</th>
<th>Meist gesuchte Personen</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. wm 2010</td>
<td>1. facebook</td>
<td>1. lena meyer-landrut</td>
</tr>
<tr>
<td>2. chatroulette</td>
<td>2. youtube</td>
<td>2. jörg kachelmann</td>
</tr>
<tr>
<td>3. ipad</td>
<td>3. berlin</td>
<td>3. daniela katzenberger</td>
</tr>
<tr>
<td>4. dsds 2010</td>
<td>4. ebay</td>
<td>4. justin bieber</td>
</tr>
<tr>
<td>5. immobilien scouts4</td>
<td>5. google</td>
<td>5. shakira</td>
</tr>
<tr>
<td>6. iphone 4</td>
<td>6. wetter</td>
<td>6. katy perry</td>
</tr>
<tr>
<td>7. facebook</td>
<td>7. tv</td>
<td>7. david guetta</td>
</tr>
<tr>
<td>8. zalando</td>
<td>8. gmx</td>
<td>8. miley cyrus</td>
</tr>
<tr>
<td>9. google street view</td>
<td>9. you</td>
<td>9. rihanna</td>
</tr>
<tr>
<td>10. studi vz</td>
<td>10. test</td>
<td>10. megan fox</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Beliebte Produkte</th>
<th>Meist gesuchte Nachrichten</th>
<th>Beliebte Bildersuchen</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ipod</td>
<td>1. bayern</td>
<td>1. ipad</td>
</tr>
<tr>
<td>2. handy</td>
<td>2. menowin fröhlich</td>
<td>2. lena meyer-landrut</td>
</tr>
<tr>
<td>3. schuhe</td>
<td>3. jörg kachelmann</td>
<td>3. larissa riquelme</td>
</tr>
<tr>
<td>4. fernseher</td>
<td>4. stuttgart 21</td>
<td>4. mehrzad marashi</td>
</tr>
<tr>
<td>5. iphone</td>
<td>5. iphone</td>
<td>5. menowin fröhlich</td>
</tr>
<tr>
<td>6. notebook</td>
<td>6. fc bayern</td>
<td>6. vampire diaries</td>
</tr>
<tr>
<td>7. wii</td>
<td>7. aschewolke</td>
<td>7. frisuren 2010</td>
</tr>
<tr>
<td>8. ipad</td>
<td>8. daniela katzenberger</td>
<td>8. kesha</td>
</tr>
</tbody>
</table>
2016 [Google Zeitgeist]

<table>
<thead>
<tr>
<th>Trends</th>
<th>Trends</th>
<th>Trends</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Suchbegriffe</strong></td>
<td><strong>Schlagzeilen</strong></td>
<td><strong>Promis national</strong></td>
</tr>
<tr>
<td>2. Pokémon Go</td>
<td>2. Donald Trump</td>
<td>2. Sarah Lombardi</td>
</tr>
<tr>
<td>3. iPhone 7</td>
<td>3. US-Wahl</td>
<td>3. Helena Fürst</td>
</tr>
<tr>
<td>*** Mehr</td>
<td>*** Mehr</td>
<td>*** Mehr</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Trends</th>
<th>Trends</th>
<th>Trends</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Promis international</strong></td>
<td><strong>Abschiede</strong></td>
<td><strong>Fragen: Warum ...?</strong></td>
</tr>
<tr>
<td>1. Donald Trump</td>
<td>1. Tamme Hanken</td>
<td>1. Warum ist Prince gestorben?</td>
</tr>
<tr>
<td>5. Antoine Griezmann</td>
<td>5. Bud Spencer</td>
<td>5. Warum Brexit?</td>
</tr>
<tr>
<td>*** Mehr</td>
<td>*** Mehr</td>
<td>*** Mehr</td>
</tr>
</tbody>
</table>
## 2018 [Google Trends]

### Allgemeine Suchbegriffe
1. WM
2. Daniel Küblböck
3. Jens Büchner
4. Avicii
5. Medaillenspiegel

### Schlagzeilen
1. Mondfinsternis
2. Euro Lira
3. Hochzeit Harry Meghan
4. Chemnitz
5. Hambacher Forst

### Abschiede
1. Jens Büchner
2. Avicii
3. Mac Miller
4. Stephen Hawking
5. Stan Lee

### Persönlichkeiten
1. Daniel Küblböck
2. Meghan Markle
3. Jan Ullrich
4. Hans-Georg Maaßen
5. Demi Lovato

### Serien
1. Babylon Berlin
2. Bad Banks
3. Tannbach
4. Haus des Geldes
5. Altered Carbon

### Sportevents
1. WM
2. Medaillenspiegel
3. Olympia
4. Deutschland Schweden
5. Handball EM

### Was-Fragen
1. Eichenprozessionsspinner was tun?
2. Was hilft gegen Wespen?
3. Was sind Permanenzen?
4. Was ist mit Daniel Küblböck?
5. Was bedeutet Rs?

### Wo-Fragen
1. Wo ist der Mond?
2. Wo ist die ISS?
3. Wo liegt Uruguay?
4. Wo läuft heute Fußball?
5. Wo spielt Neymar?

### Wie-Fragen
1. Wie oft war Frankreich Weltmeister?
2. Wie muss Deutschland spielen um weiter zu kommen?
3. Wie heißt der Sohn von Kate und William?
Changing Frequencies [Google Zeitgeist]
Changing Word Usage [Google n'gram viewer]
Zipf-Distribution

- Many events are not equally but Zipf-distributed
  - Let $f$ be the frequency of an event and $r$ its rank in the list of all events sorted by frequency
  - Zipf’s law: $f \sim k/r$ for some constant $k$

- Examples
  - Search terms on the web
  - Purchased goods
  - Words in a text
  - Sizes of cities
  - Opened files in a OS
  - …

Source: http://searchengineland.com/the-long-tail-of-search-12198
Changing the Scenario

• Assume we have a list L of values
• L is searched very often
• But: Elements in L are searched with different frequencies
• How can we organize L such that a series of searches following this frequency distribution is as fast as possible?
• Can we organize L such that searches are fast even when the frequencies of searches change arbitrarily?
• Let L organize itself depending on its usage
Content of this Lecture

• **Self-Organizing Lists**
  – Fixed frequencies
  – Dynamic frequencies

• **Organization Strategies**

• **Analysis**
Simple Case: Fixed Frequencies

• For simplicity, we assume L has \( n=|L| \) different elements
• Let \( p_i \) be the relative (and fixed) frequency at which the \( i \)’th element is searched (\( 1 \leq i \leq n \))
• Example: Assume \( p_i \) is distributed with \( p_i = \frac{1}{(1+i)^2} \cdot c \)
  – Assume \( n=25 \)
  – \( c \): normalization factor to ensure \( \sum p_i = 1 \)
  – Yields something like 41%, 18%, 10%, 6%, 4%, 3%, 2%, 1%, ...
  – Equal distribution would be 4%, 4%, 4%, 4%, ....
Analysis

- What are the expected costs for a series of searches following the frequency distribution?
- Option 1: Assume L is sorted by a key and we search L with log(n) comparisons upon each search
  - Independent of p_i’s; that’s how we did it so far
  - Expected cost for 100 searches: 100*\log(n) \sim 500
- Option 2: Assume L is sorted by p_i and we search L linearly upon each search
  - In 41% of cases: 1 access; in 18% 2 accesses; in 10% 3; ...
  - For 100 searches: 1*41+2*18+3*10+4*6+5*4+6*3+ ... \sim 380
Other Distributions

- If $p_i = 1/(1+i)^3*c$, we need only $\sim 200$ accesses for the frequency-sorted list, but still $\sim 500$ for the value-sorted list
  - Access frequencies: 62, 18, 7, 4, ...
- If $p_i = 1/n$, we have 1336 versus $\sim 500$ accesses
  - Equal distribution, access frequencies: 4, 4, 4, 4, ...
- Summary
  - Sorting the list by “popularity” may make sense
  - *Gain (or loss) in efficiency* can be computed in advance if frequency of accesses are known (and do not change)
Content of this Lecture

• Self-Organizing Lists
  – Fixed frequencies
  – Dynamic frequencies
• Organization Strategies
• Analysis
Self-Organizing Lists

• More interesting scenario
  – Access frequencies are not known in advance
  – Access frequencies change over time
    • Implication: It is not optimal to log searches for some time, then compute popularity, then re-sort list

• Our model of self-organization
  – After each access, we may change the order in the list
  – Searching the (currently) i’th element of the list costs i operations
    • I.e., L is implemented as linked list
    • Using arrays doesn’t help – we don’t know where the searched value is

• This scenario is called a self-organizing linear list (SOL)
Application: Caching

- Often, applications need to read more data from disk than there is main memory
  - Especially if there are more than one app running
- Reading from disk is \( \sim 10.000 \) times slower than memory
- **Caching:** OS keeps those data blocks in memory for which it expects that they will be reused (in the near future)
- There is not enough space to keep all ever used blocks
- Thus, when loading new blocks, the OS has to evict blocks from the cache – which ones?
  - Those that probably will not be reused in the near feature
Caching and SOLs

- OS keeps a SOL S with all block IDs sorted by popularity
- The top-k blocks of the list are cached
- When loading a new block b, the OS ...
  - evicts the k’th block in S from memory
  - loads b into the free space
  - re-organizes S to reflect the change in popularity of b
- Prominent strategies in caching
  - Most recently used: Popularity is the time stamp of the last usage
  - Most frequently used: Popularity is the number of access until now
- See course on Operating Systems (or/and Databases)
Content of this Lecture

- Self-Organizing Linear Lists
- Organization Strategies
- Analysis
Organization Strategies

• Many proposals in the literature
• Many are very application specific
• Three general strategies are popular
  – MF, move-to-front:
    After searching an element e, move e to the front of L
    • This is “most recently used” in OS terms
  – T, transpose:
    After searching an element e, swap e with its predecessor in L
  – FC, frequency count:
    Keep an access frequency counter for every element in L and keep
    L sorted by this counter. After searching e, increase counter of e
    and move e “up” to keep sorted’ness
    • This is “most frequently used” in OS terms
Visual
Properties

• Move-to-Front, MF
  – If a rare element is accessed, it “jams” the list head for some time
  – Bursts of frequent same-element accesses are well supported
  – No problem with changes in popularity over time (trends)

• Transpose, T
  – Problems with fast changing trends – slow adaptation
  – Frequently accessing same-elements well supported
    • After some swing-in time

• Frequency Count, FC
  – Requires $O(n)$ additional space
  – Re-sorting requires WC $O(\log(n))$ time (binsearch in $L[1...e]$)
    • Rather $O(1)$ in practice – local moves
  – Slow adaptation to changing trends – old counts dominate list head
Examples

- For each strategy, we can find sequences of accesses that are very well supported and others that are not.
- Example: \( L = \{1, 2, \ldots, 7\} \), \( n = 7 \); assume two workloads:
  - \( S_1: \{1, 2, \ldots, 7, 1, 2, \ldots, 7, 1, 2, \ldots \} \) (ten times)
  - \( S_2: \{1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, \ldots, 6, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7\} \)
  - Each workload performs 70 searches, each element is accessed 10 times with the same relative frequency 1/7.

- Assume an arbitrary static order of \( L \):
  - There are seven different costs 1, \ldots, 7.
  - Each cost is incurred 10 times.
  - Average cost per search for \( S_1 \) and for \( S_2 \):
    \[
    \frac{1}{10 \times n} \times \left( \sum_{i=1}^{n} 10 \times i \right) = 4
    \]
MF: Average Cost

- **MF / S₁**
  - In the first subsequence, we require i ops for the i’th access
  - L then looks like 7, 6, 5, 4, 3, 2, 1
  - We need 7 ops per element for all following subsequence
  - Together

- **MF / S₂**
  - First subsequence requires 10 = 1 + 9 ops
  - Second requires 2 + 9
  - Third requires 3 + 9
  - Together

\[ \frac{1}{10 \times n} \left( \sum_{i=1}^{n} i + 7 \times 9 \times n \right) = 6.7 \]

**Almost worst case**

\[ \frac{1}{10 \times n} \left( \sum_{i=1}^{n} i + 9 \times n \times 1 \right) = 1.3 \]

**Almost best case**
FC: Average Cost

- FC / $S_1$ (all counters are initialized with 0)
  - First subsequence costs $\sum_i$ and doesn’t change order
    - Assuming stable sorting; now all counters are 1
  - Same for all other subsequences
  - Together
    - [Ignoring the constant re-sorting costs] $\frac{1}{10 \times n} \times 10 \times \left( \sum_{i=1}^{n} i \right) = 4$

- FC / $S_2$
  - First subsequence costs 10 and no change in order
  - Second subsequence costs 20 and no change in order
  - Same for all other subsequences
  - Together
    - [Ignoring the constant re-sorting costs] $\frac{1}{10 \times n} \times \left( \sum_{i=1}^{n} 10 \times i \right) = 4$

$S_1: \{1, 2, \ldots , 7, \ 1 \ldots 7, \ 1, \ldots 7\}$

$S_2: \{1, \ldots, \ 2, \ldots \ 6, \ 7, \ldots \}$
T: Average Cost

- **T/ S₁**
  - First subsequence costs \( \Sigma i = 28 \)
  - Order now is 2,3,4,5,6,7,1 – next subseq costs 7+1+2+…5+7 = 29
  - Order now is 3,4,5,6,2,7,1 – next subseq costs 7+… = 30
  - ...

<table>
<thead>
<tr>
<th>Access</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>2</th>
<th>7</th>
<th>1</th>
<th>Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>
Worst Case Complexity

• Lemma

*The worst case complexity of MF and T for searching a workload W in a SOL L is $O(|W|*|L|)$*

• Proof
  – A workload W consists of $|W|$ requests
  – A request consists of a search and a move
  – Since a search may access any element, it is in $O(|L|)$ in worst case
  – Moves in Mf and in T are in $O(1)$
  – qed.

• Note: FC is worse (re-sorting)
Optimal Strategies

- “Optimality” of a strategy depends on the sequence of accesses
- Conventional analysis assumes worst-case for every single access, which is $O(n)$ for every search in every strategy
- Overly pessimistic: Accesses (by self-organization) influence (decrease!) the cost of subsequent accesses
- Using a clever trick, we can derive estimates about the relative costs for different strategies over any sequence
- This trick is called amortized analysis
- This will take some time (next lecture)
Exemplary Questions

• Consider a list \(L\{1,2,3,4,5\}\) and the following workload \(S\{1,3,33,5,5,5,5,5,5\}\). Analyze the cost of answering \(S\) using the MF, the T, and the FC strategy.

• Consider a list \(L, |L| = n\), of \(n\) different elements and a workload \(S\) which accesses element \(i\) with relative frequency \(p_i = 1/(1+i)^2*c\). Which of our three strategies is optimal for \(S\)?

• OS often use the least-recently used strategy for managing a cache. Is LRU equivalent to our MF, T, or FC strategy?