Algorithms and Data Structures

Self-Organizing Lists

Ulf Leser

Assumptions for Searching

- Until now, we implicitly assumed that every element of our list is 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 any element
- We may sort the list by properties of its elements, but we never considered properties of its usage
- This setting sometimes is inadequate

Searches on the Web [Germany, 2010, Google Zeitgeist]

Schnellst wachsende Suchbegriffe

1. wm 2010
2. chatroulette
3. ipad
4. dsds 2010
5. immobilienscout24
6. iphone 4
7. facebook
8. zalando
9. google street view
10. studi vz

Die häufigsten Suchbegriffe

1. facebook
2. youtube
3. berlin
4. ebay
5. google
6. wetter
7. tv
8. gmx
9. you
10. test

Meist gesuchte Personen

1. lena meyer-landrut
2. jörg kachelmann
3. daniela katzenberger
4. justin bieber
5. shakira
6. katy perry
7. david guetta
8. miley cyrus
9. rihanna
10. megan fox

Beliebte Produkte

1. ipod
2. handy
3. schuhe
4. fernseher
5. iphone
6. notebook
7. wii
8. ipad

Meist gesuchte Nachrichten

1. bayern
2. menowin fröhlich
3. jörg kachelmann
4. stuttgart 21
5. iphone
6. fc bayern
7. aschewolke
8. daniela katzenberger

Beliebte Bildersuchen

1. ipad
2. lena meyer-landrut
3. larissa riquelme
4. mehrzad marashi
5. menowin fröhlich
6. vampire diaries
7. frisuren 2010
8. kesha

Germany 2014 [Google trends]

Trending
Suchbegriffe
1. WM 2014
2. Michael Schumacher
3. iPhone 6
4. ImmobilienScout24
5. BSI Sicherheitstest

••• More

Trending
Nachrichten
1. WM 2014
2. Michael Schumacher
3. Robin Williams
4. Ebola
5. Uli Hoeneß

••• More

Trending
Sport
1. WM 2014
2. Medaillenspiegel
3. Olympia 2014
4. Neymar
5. Manuel Neuer

••• More

Trending
Musik
1. Conchita Wurst
2. ESC 2014
3. Böhse Onkelz
4. Mieze Katz
5. Aneta Sablik

••• More

Trending
Technik
1. iPhone 6
2. Threema
3. Netflix
4. Samsung Galaxy S5
5. iOS 8

••• More

Most Searched
Personen
1. Michael Schumacher
2. Helene Fischer
3. Jennifer Lawrence
4. Melanie Müller
5. Conchita Wurst

••• More

2016 [Google Zeitgeist]

Trends
Suchbegriffe

1. EM 2016
2. Pokemon Go
3. iPhone 7
4. Brexit
5. Olympia

••• Mehr

Trends
Schlagzeilen

1. Brexit
2. Donald Trump
3. US-Wahl
4. AfD
5. Brüssel

••• Mehr

Trends
Promis national

1. Nico Rosberg
2. Sarah Lombardi
3. Helena Fürst
4. Vanessa Mai
5. Jan Böhmermann

••• Mehr

Trends
Promis international

1. Donald Trump
2. Melania Trump
3. Terence Hill
4. Brigitte Nielsen
5. Antoine Griezmann

••• Mehr

Trends
Abschiede

1. Tamme Hanken
2. David Bowie
3. Roger Cicero
4. Prince
5. Bud Spencer

••• Mehr

Trends
Fragen: Warum ...?

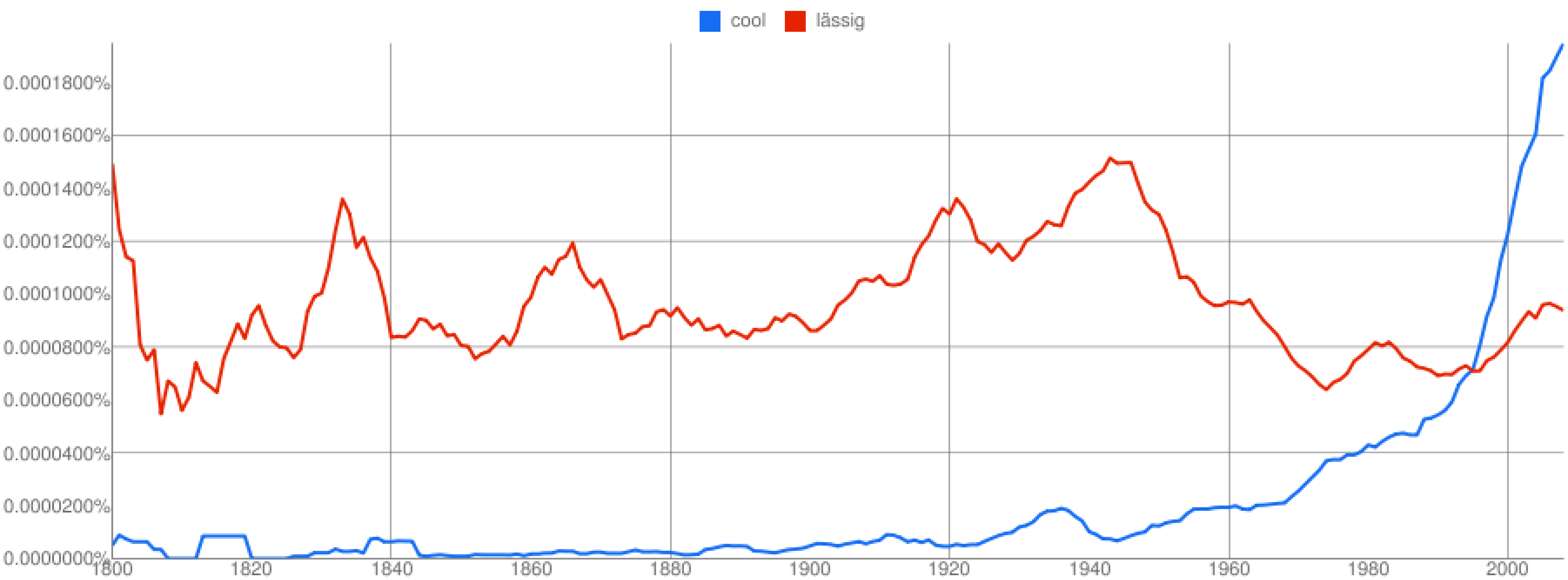
1. Warum ist Prince gestorben?
2. Warum haben Katzen Angst vor G...
3. Warum ist Italien Gruppensieger?
4. Warum Hamsterkäufe?
5. Warum Brexit?

••• Mehr

Changing Frequencies [Google Zeitgeist]

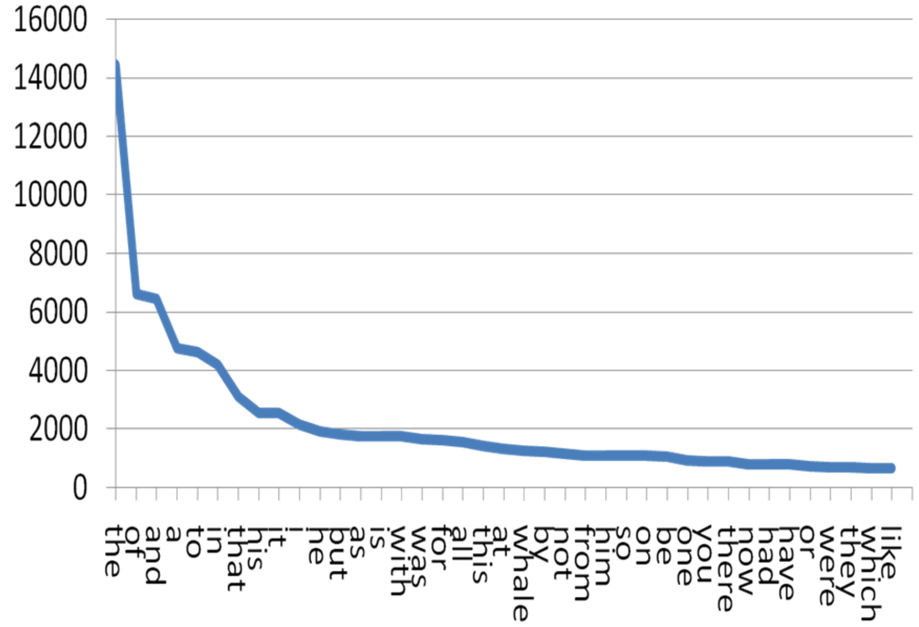
damit!

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 ~ 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 ($1 \le i \le n$)
- Example: Assume p_i is distributed with $p_i = 1/(1+i)^2 * c$
  - Assume n=25
  - c: normalization factor to ensure $\sum p_i = 1$
  - Yields something like 41%, 18%, 10%, 6%, 4%, 3%, 2%, 1%, ...

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) ~ 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+ ... ~ 380

Other Distributions

- If $p_i=1/(1+i)^3*c$, we need only ~200 accesses for the frequency-sorted list, but still ~500 for the value-sorted list
  - Access frequencies: 62, 18, 7, 4, ...
- If $p_i=1/n$, we have 1336 versus ~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 generally 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 ~10000 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

- The OS must keep a SOL S with all block IDs sorted by their popularity (= past/expected times they were read)
- 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)

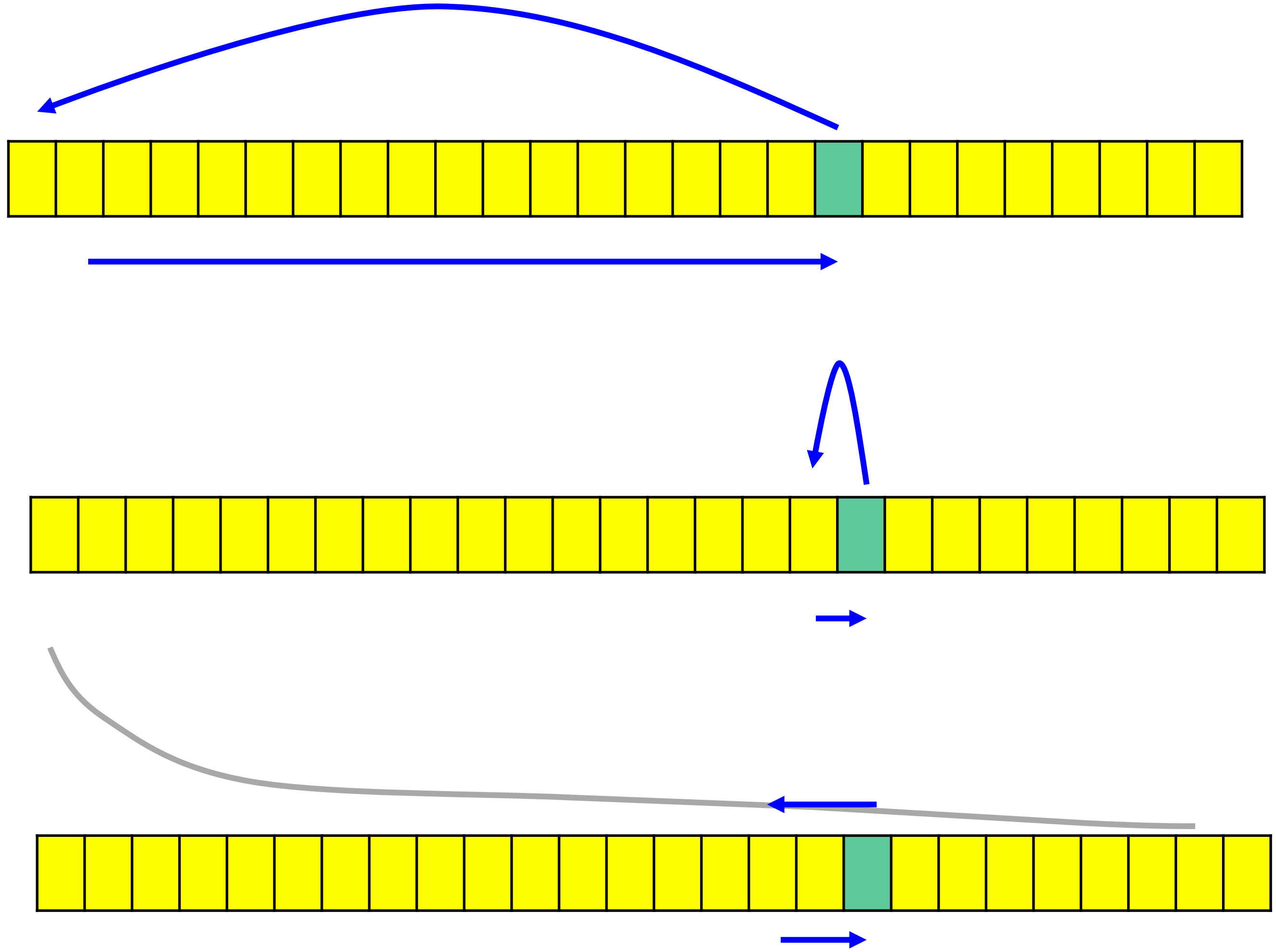
Content of this Lecture

- 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 "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,...7}, n=7; assume two workloads
  - S_1: {1,2,...7, 1,2,...7, 1,2,... ... ...7} (ten times)
  - S_2: {1,1,1,1,1,1,1,1,1,1, 2,2,2,... ... 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, ... 7
  - Each cost is incurrent 10 times
  - Average cost per search for S_1 and for S_2: $\frac{1}{10*n}*\left(\sum_{i=1}^{n} 10*i\right) = 4$

MF: Average Cost

S_1: {1,2,...7, 1...7, 1,... ...7}
S_2: {1,..., 2,... ... 6, 7, ...}

Almost worst case

- MF / S_1
  - 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

$$\frac{1}{10*n}\left(\sum_{i=1}^{n} i + 7*9*n\right) = 6.7$$

- MF / S_2
  - First subsequence requires 10=1+9 ops
  - Second requires 2+9
  - Third requires 3+9
  - Together

Almost best case

$$\frac{1}{10*n}\left(\sum_{i=1}^{n} i + 9*n*1\right) = 1.3$$

FC: Average Cost

- FC / S_1 (all counters are initialized with 0)
  - First subsequence costs ∑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]

$$\frac{1}{10*n}*10*\left(\sum_{i=1}^{n} i\right) = 4$$

- 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]

$$\frac{1}{10*n}*\left(\sum_{i=1}^{n} 10*i\right) = 4$$

T: Average Cost

- T/ S_1
  - First subsequence costs $\sum i = 28$
  - 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
  - ...

Access	3	4	5	6	2	7	1	Costs
1	3	4	5	6	2	1	7	7
2	3	4	5	2	6	1	7	5
3	3	4	5	2	6	1	7	1
4	4	3	5	2	6	1	7	2
5	4	5	3	2	6	1	7	3
6	4	5	3	6	2	1	7	5
7	4	5	3	6	2	7	1	7

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 even slightly 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}. 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?