Algorithms and Data Structures

Self-Organizing Lists

Ulf Leser
Assumptions for Searching

- Until now, we always assumed that every element of our list is searched with the same probability, i.e., with the same frequency.
- Accordingly, we treated all elements of the list equal.
- We may sort the list by properties of its values, but we did never consider properties of their usage.
- This setting often is the right one and often the wrong one.
## 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 mayer-landrut</td>
</tr>
<tr>
<td>2. chatroulette</td>
<td>2. youtube</td>
<td>2. jörg lachemann</td>
</tr>
<tr>
<td>3. ipad</td>
<td>3. berlin</td>
<td>3. daniela katzenberger</td>
</tr>
<tr>
<td>4. dies 2010</td>
<td>4. easy</td>
<td>4. justin bieber</td>
</tr>
<tr>
<td>5. immobilenscout24</td>
<td>5. google</td>
<td>5. shakira</td>
</tr>
<tr>
<td>6. iphone</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. omv</td>
<td>8. miley cyrus</td>
</tr>
<tr>
<td>9. google street view</td>
<td>9. you</td>
<td>9. rhema</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. ipad</td>
<td>1. bayern</td>
<td>1. ipad</td>
</tr>
<tr>
<td>2. handy</td>
<td>2. menewin fröhlich</td>
<td>2. lena mayer-landrut</td>
</tr>
<tr>
<td>3. achune</td>
<td>3. jörg lachemann</td>
<td>3. larissa riquelme</td>
</tr>
<tr>
<td>4. fernseher</td>
<td>4. stuttgart 21</td>
<td>4. mehrzaad marashi</td>
</tr>
<tr>
<td>5. iphone</td>
<td>5. iphone</td>
<td>5. menewin 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. trisuren 2010</td>
</tr>
<tr>
<td>8. ipad</td>
<td>8. daniela katzenberger</td>
<td>8. kasha</td>
</tr>
</tbody>
</table>
Changing Frequencies [Google Zeitgeist]
Changing Word Usage [Google n'gram viewer]

- cool
- lässig
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 $k$
    - Similar. Power law: $f \sim 1/r^k$ with $k>1$
- Examples
  - Search terms on the web
  - Purchases of goods
  - Words in a text
  - Number of links from a web-side
  - Sizes of cities
  - …

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: Not all values in \( L \) are searched \textit{with the same frequencies}
• How can we organize \( L \) such that searches are as fast as possible?
• Let \( L \) \textit{organize itself depending on its usage}
Content of this Lecture

- Self-Organizing Lists
- Organization Strategies
- Analysis
Simple Case: Fixed Search Frequencies

- For simplicity, we assume $L$ has $n = |L|$ different values.
- Assume that we **know the relative frequency** $p_i$ with which each of the $n$ values in $L$ will be searched ($1 \leq i \leq n$).
- Assume $p_i$ is distributed with $p_i = 1/(1+i)^2 \cdot c$.
  - Assume $n = 25$.
  - $c$: normalization factor to ensure $\sum p_i = 1$.
  - Yields something like 41, 18, 10, 6, 4, 3, 2, 1, 1, 1, 1, 1, ...
Analysis

• What are our expected costs?

• Option 1: Assume \textit{L is sorted by values} and we search L with \(\log(n)\) comparisons upon each search
  – Expected cost for 100 searches: \(100*\log(n) \sim 500\)

• Option 1: Assume \textit{L is sorted by \(p_i\)} and we search L linearly upon each search
  – In 41\% of cases 1 access; in 18\% 2; in 10\% 3; ... 
  – For 100 searches: \(41+2*18+3*10+6*4+4*5+3*2+1*7+ ... = 386\)
Other Distributions

- Using $p_i = 1/(1+i)^3*c$, we have **200 accesses for the frequency-sorted list**, but still $\sim 500$ for the value-sorted list
  - Access frequencies: 62, 18, 7, 4, ...
- But: For $p_i = 1/n$, 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 before-hand by counting # of operations and comparing these to binsearch
Self-Organizing Lists (SOL)

• More interesting scenarios
  – Access order follows unknown pattern
    • Probabilities are heavily skewed over time
  – Popularities change over time

• Implication: It is not optimal to log searches for some time, then compute popularity, then re-sort list

• Further assumptions
  – 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, the user wants to read more data from disk than there is main memory
  - Especially if there are more than one users
- Reading from disk is ~1000 times slower than from memory
- **Caching**: OS keep 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 could keep a SOL $S$ with all block IDs sorted by their popularity
- The top-k of these blocks are cached
- When loading a new block $b$, the OS ...
  - Evicts the last block in $S$ from memory
  - Loads $b$ into the free space
  - Re-organize $S$ 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
Re-Organization Strategies

• Many proposals in the literature
  – For certain access distribution, certain data types, certain hardware, certain constraints, certain applications, ...

• Three popular strategies
  – MF, move-to-front:
    After searching an element e, move e to the front of L
  – 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
Properties

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

• T
  – Problems with fast changing trends – slow adaptation
  – Frequently accessing frequent elements well supported – after some tuning time

• FC
  – Requires $O(n)$ additional space – prohibitive for large $L$
  – Re-sorting requires WC $O(\log(n))$ time (binsearch in $L[1...e]$)
    • Rather $O(1)$ on average
  – 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: \( L = \{1,2,\ldots,7\}, \ n=7 \)
  
  - \( S_1: \{1,2,\ldots,7, 1,2,\ldots,7, 1,2,\ldots,\ldots,7\} \) (ten times)
  
  - \( S_2: \{1,1,1,1,1,1,1,1,2,2,2,\ldots,6, 7,7,7,7,7,7,7,7,7,7,7\} \)
  
  - Each sequence performs 70 searches, each element is accessed with the same relative frequency \( 1/7 \)

• Assume any static order
  
  - There are seven different costs \( 1, \ldots, 7 \)
  
  - Each cost is incurred 10 times
  
  - Thus, the **average cost** will be
    \[
    \frac{1}{10 \times n} \left( \sum_{i=1}^{n} 10 \times i \right) = 4
    \]
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 require 7 ops per element for every further subsequence.
  - Together:
    - Much worse than static order
    \[
    \frac{1}{10 \times n} \left( \sum_{i=1}^{n} i + 7 \times 9 \times 7 \right) = 6.7
    \]

- **MF / S2**
  - First subsequence requires \( 10 = 1 + 9 \) ops.
  - Second requires \( 2 + 9 \).
  - Third requires \( 3 + 9 \).
  - Together:
    - Much better than static order
    \[
    \frac{1}{10 \times n} \left( \sum_{i=1}^{n} i + 9 \times 7 \times 1 \right) = 1.3
    \]
FC: Average Cost

- **FC / S1** (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 re-sorting costs
      $$\frac{1}{10^n} \times 10 \times \left( \sum_{i=1}^{n} i \right) = 4$$

- **FC / S2**
  - 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 re-sorting costs
      $$\frac{1}{10^n} \times \left( \sum_{i=1}^{n} 10 \times i \right) = 4$$
T: Average Cost

- **T/ S1**
  - 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
  - ...

<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>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>7</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>
Optimal Strategies

- “Optimality” of a strategy depends on the sequence of accesses.
- Conventional worst-case estimation uses worst-case for every single access, which is $O(n)$ for every strategy.
- This is overly pessimistic:
  - Accesses influence the cost of subsequent accesses.
  - Constructing real worst cases can be quite hard, if not impossible.
- Using a clever trick, we can derive estimates about the relative costs for different strategies over any sequence.
- This trick is called amortized analysis.
Content of this Lecture

- Self-Organizing Linear Lists
- Organization Strategies
- Analysis
  - Goal and idea
  - Preliminaries
  - A short proof
Notation

- Assume we have a self-organizing strategy $A$ and a sequence $S=\{s_i\}$ of accesses to a list $L$
  - As usual: Accessing the $i$'th element costs $i$
- After an access to element $i$, $A$ may move $i$ by swapping
  - Swap with predecessor (to-front) or successor (to-back)
  - Let $F_A(l)$ be the number of front-swaps and $X_A(l)$ the number of back-swaps after access number $l$
    - $F_A/X_A$ for strategy $A$, $F_{MF}/X_{MF}$ for strategy $MF$, $F_T/X_T$ ... $F_{FC}/X_{FC}$
    - Of course, $\forall l: X_{MF}(l)=X_T(l)=X_{FC}(l)=0$
- Let $C_A(S)$ be the total access costs of $A$ incurred by $S$
  - Again: $C_{MF}$ for strategy $MF$, $C_T$ for $T$, $C_{FC}$ for $FC$
  - Using conventional worst-case analysis, we can only derive that $C_A(S)$ is in $O(|S|\cdot|L|)$ – for any strategy
Theorem

Let $A$ be any self-organizing strategy for a SOL $L$, $MF$ be the move-to-front strategy, and $S$ be a sequence of accesses to $L$. Then

$$C_{MF}(S) \leq 2C_A(S) + X_A(S) - F_A(S) - |S|$$

What does this mean?
- We don't really learn more about the complexity of $A$ / $MF$
- But we learn that $MF$ is really good
- Any strategy following the same constraints (only swaps) will at best be roughly twice as good as $MF$
  - Assuming $C(S) \gg |S|$ and for $|S| \to \infty$: $X(S) \sim F(S)$ for any strategy
- $MF$, despite its simplicity, is a fairly safe bet in whatever circumstances ( = sequences)
Idea of the Proof

- We will not derive counts for $C_A(S)$ or $C_{MF}(S)$, but compare “costs” for each access in $L$ using MF and using $A$.

- Think of both strategies running $S$ on two copies of the same initial list $L$.
  - After each step, $A$ and MF perform different swaps, so all list states except the first very likely are different.

- We will compare these two lists and count a certain property – the number of inversions (“Fehlstellungen”).
  - Actually, we only look how the number of inversion changes.

- Finally, we can use the real costs as an upper bound to the number of inversions between both lists.

- This will prove the theorem.
Content of this Lecture

- Self-Organizing Linear Lists
- Organization Strategies
- Analysis
  - Goal and idea
  - Preliminaries
  - A short proof (after much preparatory work)
Inversions

- Let L and L' be permutation of the set \{1, 2, ..., n\}
  - I.e., |L|=|L'|=n

- Definition
  - A pair \((i,j)\) is called an inversion of \(L\) and \(L'\) iff \(i\) and \(j\) are in different order in \(L\) than in \(L'\) (for 0 \(\leq i,j \leq n\) and \(i \neq j\))
  - The number of inversions between \(L\) and \(L'\) is written \(\text{inv}(L, L')\)

- Remarks
  - Different order: Once \(i\) before \(j\), once \(i\) after \(j\)
  - Obviously, \(\text{inv}(L, L') = \text{inv}(L', L)\)

- Example: \(\text{inv}(\{4,3,1,5,7,2,6\}, \{3,6,2,5,1,4,7\}) = 12\)

- It is easy to deduce \(L''\) such that \(\text{inv}(L,L') = \text{inv}(\{1,...,n\},L'')\)
  - Thus, we may always assume that the first list is \{1,...,n\}
Inversion Changes

• Assume we applied 1-1 steps creating $L_{MF}$ using MF and $L_{A}$ using A
• Let us consider the next step 1, creating $L_{MF'}$ and $L_{A'}$
• How does 1 change the number of inversions, i.e., how can we compute $inv(L_{MF'}, L_{A'})$ from $inv(L_{MF}, L_{A})$?
  – Assume 1 accesses element i from $L_{A}$
  – We may assume it is at position i
  – Let i be at position k in $L_{MF}$
  – Access in $L_{A}$ costs i, in $L_{MF}$ it costs k
  – After 1, A performs an unknown number of swaps; MF performs exactly k front-swaps
Counting Inversion Changes 1

- Let $X_i$ be the set of values that are before position $k$ in $L_{MF}$ and after position $i$ in $L_A$
  - Clearly, $|X_i| + |Y_i| = k-1$
- Le $Y_i$ be the set of values that are before position $k$ in $L_{MF}$ and before position $i$ in $L_A$
- All pairs $(i, c)$ with $c \in X_i$ are inversions between $L_A$ and $L_{MF}$
- After $l$, MF moves element $i$ to the front
  - All inversions from $X_i$ disappear (these are $|X_i|$ many)
  - But $|Y_i| = k-1-|X_i|$ new inversions appear
- Thus $\text{inv}(L_{MF}', L_A') = \text{inv}(L_{MF}, L_A) - |X_i| + k-1-|X_i|$
  - If $A$ did nothing – but $A$ is doing something – wait a minute
Counting Inversion Changes 2

- In step $l$, let $A$ perform $F_A(i)$ front-swaps and $X_A(i)$ back-swaps
  - Swaps depend on $i$, not $l$
- Every front-swap (e.g. $j$) in $L_A$ decreases $\text{inv}(L_{MF}', L_A')$ by 1
  - Before the swap, $j$ must be before $i$ in $L_A$ (it is a front-swap), but after $i$ in $L_{MF}'$ (because $i$ now is the first element)
  - After the swap, $i$ is before $j$ in both $L_A'$ and $L_{MF}'$
- Equally, every back-swap increases $\text{inv}(L_{MF}', L_A')$ by 1
- Together: After step $l$, we have
  $$\text{inv}(L_{MF}', L_A') = \text{inv}(L_{MF}, L_A) - |X_i| + k-1-|X_i| - F_A(i) + X_A(i)$$
Amortized Costs

- Let $t_l$ be the real costs of strategy MF for step $l$
- Definition (a central measure for the proof)
  - The amortized costs of step $l$, $a_l$ are
    \[ a_l = t_l + \text{inv}(L_{A}^{l}, L_{MF}^{l}) - \text{inv}(L_{A}^{l-1}, L_{MF}^{l-1}) \]
  - Accordingly, the amortized costs of sequence $S$ are
    \[ \sum a_l = \sum t_l + \text{inv}(L_{A}^{m}, L_{MF}^{m}) - \text{inv}(L_{A}^{0}, L_{MF}^{0}) \]

- Explanation
  - There is no simple “why this measure” – the trick will follow in a minute (we will make a connection between costs and inv)
  - Costs are called amortized because we consider costs that follow – if a step is costly, we amortize its costs over all subsequent steps
Content of this Lecture

- Self-Organizing Linear Lists
- Organization Strategies
- Analysis
  - Goal and idea
  - Preliminaries
  - A short proof
Putting it Together

- We know for every step \( l \) from \( S \) accessing \( i \):
  \[
  \text{inv}(L_{MF}', L_A') = \text{inv}(L_{MF}, L_A) - |X_i| + k - 1 - |X_i| - F_A(i) + X_A(i)
  \]
  and thus
  \[
  \text{inv}(L_{MF}', L_A') - \text{inv}(L_{MF}, L_A) = -|X_i| + k - 1 - |X_i| - F_A(i) + X_A(i)
  \]

- Using the fact that \( t_l = k \) for \( MF \), we get amortized costs of

  \[
  a_l = t_l + \text{inv}(L_A', L_{MF}') - \text{inv}(L_A, L_{MF}) \\
  = k - |X_i| + k - 1 - |X_i| - F_A(i) + X_A(i) \\
  = 2(k - |X_i|) - 1 - F_A(i) + X_A(i)
  \]

- Recall that \( |Y_i| = k - 1 - |X_i| \) are those elements before \( i \) in both lists. This implies that \( k - 1 - |X_i| \leq i - 1 \) or \( k - |X_i| \leq i \)
  - There can be at most \( i - 1 \) elements before position \( i \) in \( L_A \)

- Therefore: \( a_l \leq 2i - 1 - F_A(i) + X_A(i) \)
Putting it Together

• This is the central trick!
• Because we only looked at inversions (and hence the sequence of values), we can draw a connection between the value that is accessed and the affected inversions
• Further, we know the cost of accessing $i$ using $A$: that’s $i$
• Together: $a_i \leq 2C_A(i) - 1 - F_A(i) + X_A(i)$

• Recall that $|Y_i| = k-1-|X_i|$ are those elements before $i$ in both lists. This implies that $k-1-|X_i| \leq i-1$ or $k-|X_i| \leq i$
  – There can be at most $i-1$ elements before position $i$ in $L_A$
• Therefore: $a_i \leq 2i - 1 - F_A(i) + X_A(i)$
Aggregating

• Aggregating this inequality over all $a_i$ (hence $S$), we get
  $\sum a_i \leq 2C_A(S) - |S| - F_A(S) + X_A(S)$
• We now use the previous measure
  – That’s why we defined it as such
    $\sum a_i = \sum t_i + \text{inv}(L_A^m, L_{MF}^m) - \text{inv}(L_A^0, L_{MF}^0)$
• Since $\sum t_i = C_{MF}(S)$ and $\text{inv}(L_A^0, L_{MF}^0)=0$, we get
  $C_{MF}(S) + \text{inv}(L_A^m, L_{MF}^m) \leq 2C_A(S) - |S| - F_A(S) + X_A(S)$

• It finally follows ($\text{inv()}\geq 0$)
  $C_{MF}(S) \leq 2C_A(S) - |S| - F_A(S) + X_A(S)$
Why the Heck?

- Self-organization creates a type of problem we were not confronted with before
  - Things change a lot during program execution
  - But not at random – we follow a strategy
  - Amortized analysis should be kept in mind as a possibility for analysis for such cases
- Analysis is none-trivial, but
  - Helped to find a elegant and surprising conjecture
  - Very interesting in itself: We showed relationships between measures we never counted (and could not count easily)