Algorithms and Data Structures

Amortized Analysis

Ulf Leser

- Two Examples
- Two Analysis Methods
- Dynamic Tables
- SOL - Analysis

- Remark
  - [OW93], 5th edition, covers SOL-Analysis but not the other parts
  - [Cor09] covers everything, [Cor03] only other parts

Setting

- SOL: Sequences of operations influencing each other
  - We have a sequence Q of operations on a data structure
    - Searching SOL and rearranging a SOL
  - Operations are not independent – by changing the data structure, costs of subsequent operations are influenced
- Conventional WC-analysis produces misleading results
  - Assumes all operations to be independent
  - Search order in workload does not influence WC result
- Amortized analysis analyzes the complexity of a sequence of interfering operations
  - In other terms: We seek the worst average cost of each operation in any sequence

„Amortizing“

- Economics: How long does it take until a (high) initial investment pays off because it leads to continuous business improvements (less costs, more revenue)?
- Example
  - Investment of 6000€ leads to daily rev. increase from 500 to 560€
  - Investment amortized after 100 days

- WC: Look at all days independently
  - Look at difference cost / revenue
  - Compare 560-6000 to 500-0
  - Do not invest! Never!

Algorithmic Example 1: Multi-Pop (mpop)

- Assume a stack S with a special operation: mpop(k)
  - mpop(k) pops min(k, |S|) elements from S
  - Implementation: mpop calls pop k times
- Assume any sequence Q of operations push, pop, mpop
  - E.g. Q={push,push,mpop(k),push,pop,push,mpop(k),…}
- Assume costs c(push)=1, c(pop)=1, c(mpop(k))=k
- What cost do we expect for a given Q with |Q|=n?
  - Cost of ops in Q: 1 (push) or 1 (pop) or k (mpop)
  - In the worst case, k can be n
    - n-1 times push, then one mpop(n)
    - Worst case of a single operation is O(n)
  - For n operations: Total worst-case cost: $O(n^2)$

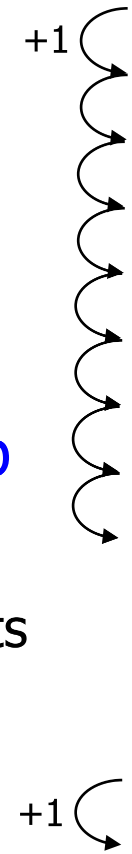
Note: True costs only ~2*n

Problem

- Clearly, the cost of Q is in $O(n^2)$, but this is not tight
- A simple thought shows: The cost of Q is in O(n)
  - Every element can be popped only once
    - No matter if this happens through a pop or a mpop
  - Pushing an element costs 1, popping it costs 1
  - A given Q can at most push n elements and pop n elements
    - Every pushed element can be popped only once
  - Thus, the total cost is in O(n)
    - It is maximally ~2n
- We want to derive such a result in a systematic manner
  - Analyzing SOLs is not that easy

Example 2: Bit-Counter

- We want to generate bitstrings by iteratively adding 1
  - Starting from 0
  - Assume bitstrings of length k
  - Roll-over counter if we exceed 2^k-1
- Q is a sequence of „+1“
- We count as cost of an operation the number of bits we have to flip
- Classical WC analysis
  - A single operation can flip up to k bits
    - “1111111” +1
  - Worst case cost for Q: O(k*n)



Bitstring, k=8	Bits flipped	Aggregated costs
00000000		
00000001	1	1
00000010	2	3
00000011	1	4
00000100	3	7
00000101	1	8
00000110	2	10
00000111	1	11
00001000	4	15
00001001	1	16
00001010	2	18
...	...	...
11111111	1	...
00000000	8	...

Problem

- Again, this complexity is overly pessimistic / not tight
- Cost actually is in O(n)
  - The right-most bit is flipped in every operation: cost=n
  - The second-rightmost bit is flipped every second time: n/2
  - The third …: n/4
  - …
  - Together

$$\sum_{i=0}^{k-1} \frac{n}{2^i} < n * \sum_{i=0}^{\infty} \frac{1}{2^i} = 2 * n$$

b_3	b_2	b_1	b_0
0	0	0	0
0	0	0	**1**
0	0	**1**	**0**
0	0	1	**1**
0	**1**	**0**	**0**
0	1	0	**1**
0	1	**1**	**0**
0	1	1	**1**
1	**0**	**0**	**0**
…	…	…	…
$= \frac{n}{8}$	$= \frac{n}{4}$	$= \frac{n}{2}$	$= \frac{n}{1}$

- Two Examples
- Two Analysis Methods
  - Accounting Method
  - Potential Method
- Dynamic Tables
- SOL - Analysis

Accounting Analysis

- Idea: We create an account for Q
- Operations put / withdraw some amounts of "money"
- We choose these amounts such that the current state of the account is always (throughout Q) an upper bound of the actual cost incurred by Q
  - Let c_i be the true cost of operation i, d_i its effect on the account
  - We require

$$\forall 1 \le k \le n : \sum_{i=1}^{k} c_i \le \sum_{i=1}^{k} d_i$$

  - Additional constraint: The account must never become negative
  - "≤" gives us more freedom in analysis than "="
- It follows: An upper bound for the account (d) after Q is also an upper bound for the true cost (c) of Q

Application to mpop

- Assume $d_{push}=2$, $d_{pop}=0$, $d_{mpop}=0$
- Upper bounds?
  - Clearly, d_{push} is an upper bound on c_{push} (which is 1)
  - But neither d_{pop} nor d_{mpop} are upper bounds for c_{pop} / c_{mpop}
- Let's try: $d_{push}=2$, $d_{pop}=1$, $d_{mpop}=n$
  - Now all individual d's are upper bounds for their c's
  - But this doesn't help (worst-case analysis)

We again assumed independence of ops

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} d_i \leq n * n \in O(n^2)$$

- But: We only have to show that the sum of d's for any prefix of Q is higher than the sum of c's

Application to mpop

- Assume again: $d_{push}=2$, $d_{pop}=0$, $d_{mpop}=0$
- Summing these up along a sequence of ops yields an upper bound on the real cost
  - Idea: Whenever we push an element, we pay 1 for the push and 1 for the operation that will (sometime later) pop exactly this element
    - It doesn't matter whether this will be through a pop or a mpop
    - Recall: For every pop, there must have been a push before
  - Thus, when it comes to a pop or mpop, there is always "enough money" on the account
    - Deposited by previous push's
    - "enough": Enough such that the sum remains an upper bound
- This proves $$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} d_i \leq 2*n \in O(n)$$

Q	|Q|	c_i	d_i	Account
	0			**0**
d_{push}	1	1	**2**	0+2-1=**1**
d_{push}	2	1	**2**	1+2-1=**2**
d_{pop}	1	1	0	2+0-1=**1**
d_{push}	2	1	**2**	1+2-1=**2**
d_{mpop}	0	2	0	2+0-2=**0**
d_{push}	1	1	**2**	0+2-1=**1**
d_{push}	2	1	**2**	1+2-1=**2**
d_{push}	3	1	**2**	2+2-1=**3**
d_{pop}	2	1	0	3+0-1=**2**
...		...	$\leq$ **2n**	

Choose d‘s carefully

- Assume d_{push}=1, d_{pop}=1, d_{mpop}=1
  - Assume Q={push,push,push,mpop(3)}
  - $\sum c=6 > \sum d = 4$
- Assume d_{push}=1, d_{pop}=0, d_{mpop}=0
  - Assume Q={push,push,mpop(2)}
  - $\sum c=4 > \sum d = 2$
- Assume d_{push}=3, d_{pop}=0, d_{mpop}=0
  - Fine as well, but not as tight (but also leads to O(n))
- Take-Away: We must chose d such that the upper bound inequality always holds

Application to Bit-Counter

- Look at the sequence Q‛ of flips generated by a sequence Q
  - Every +1 creates a sequence of [0…k] flip-to-0 and [0…1] flip-to-1
    - There is no „flip to 1" if we roll-over
  - Since only flips cost, Q' can be used to study the cost of Q
- Let's set $d_{flip\text{-}to\text{-}1}=2$ and $d_{flip\text{-}to\text{-}0}=0$
  - Note: We start with only 0 and can flip-to-0 any 1 only once
    - Before we flip-to-1 again, again enabling one flip-to-0 etc.
  - Idea: When we flip-to-1, we pay 1 for flipping and 1 for the back-flip-to-0 that might happen at some later time in Q'
    - There can be only one flip-to-0 per single flip-to-1
  - Thus, the account is always an upper bound on the actual cost
- Same idea: No flip-to-0 (pop) without prev. flip-to-1 (push)

Application to Bit-Counter -2-

- We know that the account is always an upper bound on the actual cost for any prefix of Q
- Every step of Q creates a sequence of flip-to-1 (at most one) and flip-to-0 in Q'
- This sequence in Q' costs at most 2
  - There can be only on flip-to-1, and all flip-to-0 are free
- Every step in Q creates a sequence in Q' costing at most 2
- Thus, Q is bound by O(n)
- qed.

- Two Examples
- Two Analysis Methods
  - Accounting Method
  - Potential Method
- Dynamic Tables
- SOL - Analysis

Potential Method: Idea

- In the accounting method, we assign a cost to every operation and compare aggregated accounting costs of ops with aggregated real costs of ops
- In the potential method, we assign a potential $\Phi(D)$ to the data structure D manipulated by Q
  - Think of the potential as potential future cost
- As ops from Q change D, they also change D's potential
- The trick is to design Φ such that we can use it to derive an upper bound on the real cost of Q

- "Accounting" and "potential" methods are quite similar – use whatever is easier to apply for a given problem

Potential Function

- Let D_0, D_1, … D_n be the states of D when applying Q
- We define the amortized cost d_i of the i'th operation as $d_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
- We derive the amortized cost of Q as

$$\sum_{i=1}^{n} d_i = \sum_{i=1}^{n} (c_i + \phi(D_i) - \phi(D_{i-1})) = \sum_{i=1}^{n} c_i + \phi(D_n) - \phi(D_0)$$

- Idea: If we find a Φ such that (a) we can obtain formulas for the amortized costs for all individual d_i and (b) $\Phi(D_n) \geq \Phi(D_0)$, we have an upper bound for the real costs
  - Because then:

$$\sum_{i=1}^{n} d_i = \sum_{i=1}^{n} c_i + \phi(D_n) - \phi(D_0) \geq \sum_{i=1}^{n} c_i$$

Details: Always Pay in Advance

- Operations raise or lower the potential of D
- We need to find a function Φ such that
  - Req. 1: $\Phi(D_i)$ depends on a property of D_i (future cost)
  - Req. 2: $\Phi(D_n) \geq \Phi(D_0)$ [here we will always have $\Phi(D_0)=0$]
  - Req. 3: We can compute $d_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
- As within a sequence we do not know its future, we also have to require that $\Phi(D_i)$ never is negative
  - Otherwise, the amortized cost of the prefix Q[1…i] would not be an upper bound of the real costs at step i
- Idea: Always pay in advance

Example: mpop

- We use the number of objects on the stack as its potential
- Then
  - Req. 1: $\Phi(D_i)$ depends on a property of D_i
    - Future cost: To empty a stack with n elements, we need cost n
  - Req. 2: $\Phi(D_n) \geq \Phi(D_0)$ and $\Phi(D_0)=0$
  - Req. 3: Compute $d_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$ for all ops:
    - Assume $x=\Phi(D_i)$
    - If op is push: $d_i = c_i + (x - (x-1)) = 1 + 1 = 2$
    - If op is pop: $d_i = c_i + (x - (x+1)) = 1 - 1 = 0$
    - If op is mpop(k): $d_i = c_i + (x - (x+k)) = k - k = 0$
- Thus, $2*n \geq \Sigma d_i \geq \Sigma c_i$ and Q is in O(n)

Example: Bit-Counter

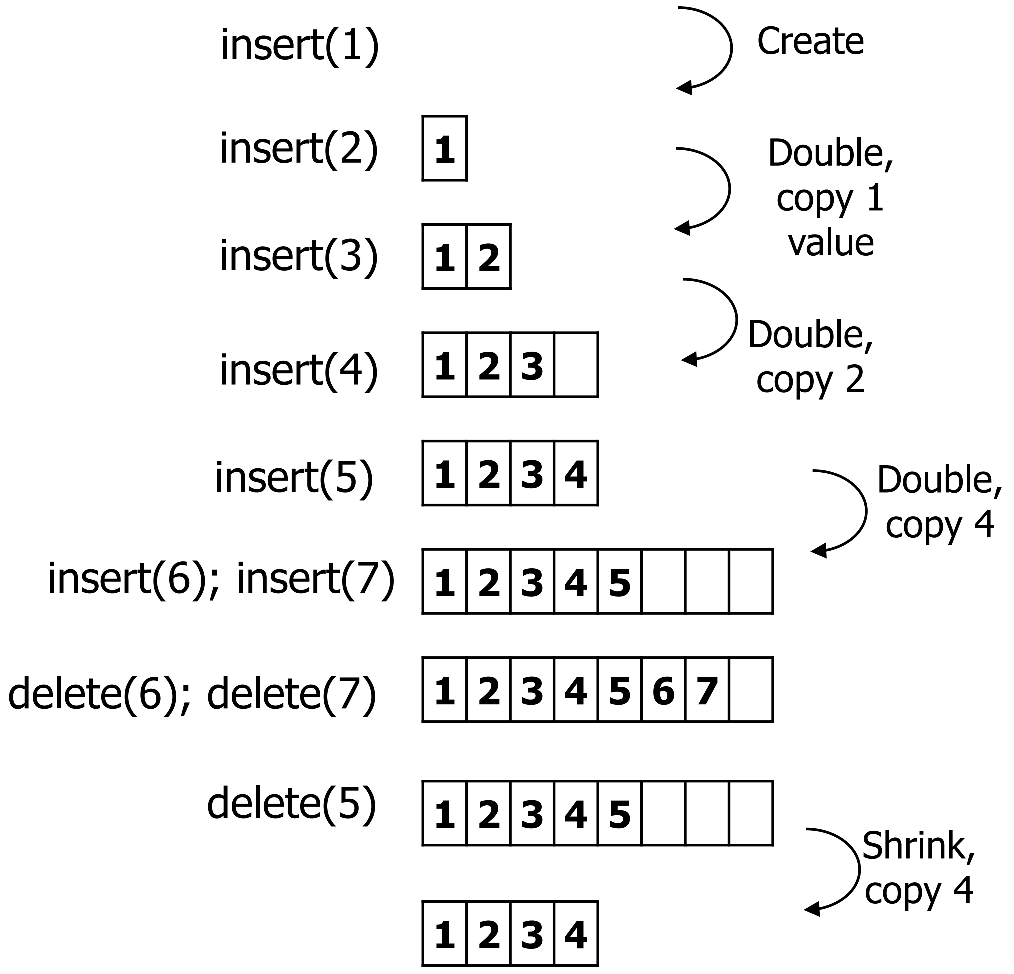
- We use the number of „1“ in the bitstring as its potential
- Then
  - Req. 1: $\Phi(D_i)$ depends on a property of D_i
  - Req. 2: $\Phi(D_n) \geq \Phi(D_0)$ and $\Phi(D_0)=0$
  - Req. 3: We compute $d_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$ for all ops
    - Let the i'th operation incur t_i flip-to-0 and 0 or 1 flip-to-1
    - Thus, $c_i \leq t_i + 1$
    - If $\Phi(D_i)=0$, then operation i has flipped all positions to 0; this implies that previously they were all 1, which means that $\Phi(D_{i-1})=k$
    - If $\Phi(D_i)>0$, then $\Phi(D_i)=\Phi(D_{i-1})-t_i+1$
    - In both cases, we have $\Phi(D_i) \leq \Phi(D_{i-1})-t_i+1$
    - Thus, $d_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (t_i+1) + (\Phi(D_{i-1})-t_i+1) - \Phi(D_{i-1}) \leq 2$
- Thus, $2*n \geq \Sigma d_i \geq \Sigma c_i$ and Q is in O(n)

- Two Examples
- Two Analysis Methods
- Dynamic Tables
  - SOL will be complicated … we still try to get familiar with the analysis method using simpler problems …
- SOL - Analysis

Dynamic Tables

- We use amortized analysis for something useful: Complexity of operations on a dynamic table
- Assume an array T and a sequence Q of inserts/deletes
- Dynamic Tables: Keep the array small, yet avoid overflows
  - Start with a table T of size 1
  - When inserting and T is full, we double |T|; upon deleting and T is only half-full, we reduce |T| by 50%
  - "Doubling", "reducing" means: Copying data to a new array
  - Assumption: Copying an element of an array costs 1
- Thus, any operation (ins or del) costs either 1 or |T|

Example



- Conventional WC analysis
  - Complexity of any operation is O(n)
  - Complexity of any Q is $O(n^2)$
- But (again)
  - Ops not independent
  - When we double (costly) at some time, we don't have to do so again for quite a while

With Potential Method

1: $\Phi(D_i)$ depends on a property of D_i
2: $\Phi(D_n) \geq \Phi(D_0)$
3: $d_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

1	**2**	**3**	**4**	**5**	**6**		

|T|=8; num(T)=6

- Let num(T) be the current number of elements in T
- We use potential $\Phi(T) = 2*num(T) - |T|$
  - Intuitively a "potential"
    - Immediately before an expansion, num(T)=|T| and $\Phi(T)=|T|$, so there is much potential in T (we saved for the expansion to come)
    - Immediately after an expansion, num(T)=|T|/2+1 and $\Phi(T)=2$; almost all potential has been used, we need to save again for next expansion
  - Formally
    - Requirement 1: Of course
    - Requirement 2: As T is always at least half-full, $\Phi(T)$ is always ≥ 0; we start with |T|=0, and thus $\Phi(T_n)-\Phi(T_0) \geq 0$

Continuation

1: $\Phi(D_i)$ depends on a property of D_i
2: $\Phi(D_n) \geq \Phi(D_0)$
3: $d_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

- Req. 3: Let's look at $d_i = c_i + \Phi(T_i) - \Phi(T_{i-1})$ for insertions
- Without expansion

$$\begin{aligned} d_i &= 1 + (2*num(T_i)-|T_i|) - (2*num(T_{i-1})-|T_{i-1}|) \\ &= 1 + 2*num(T_i)-2*num(T_{i-1}) - |T_i| + |T_{i-1}| \\ &= 1 + \quad 2 \quad + \quad 0 \\ &= 3 \end{aligned}$$

- With expansion

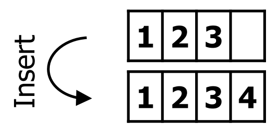
$$\begin{aligned} d_i &= num(T_i) + (2*num(T_i) - |T_i|) - (2*num(T_{i-1})-|T_{i-1}|) \\ &= num(T_i) + 2*num(T_i) - |T_i| - 2*num(T_{i-1}) + |T_{i-1}| \\ &= num(T_i) + 2*num(T_i) - 2*(num(T_i)-1) - 2*(num(T_i)-1) + num(T_i)-1 \\ &= 3*num(T_i) - 2*num(T_i) + 2 - 2*num(T_i) + 2 + num(T_i) - 1 \\ &= 3 \end{aligned}$$

- Thus, $3*n \geq \Sigma d_i \geq \Sigma c_i$ and Q is in O(n) (for only insertions)

Continuation

1: $\Phi(D_i)$ depends on a property of D_i
2: $\Phi(D_n) \geq \Phi(D_0)$
3: $d_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

- Req. 3: Let's look at $d_i = c_i + \Phi(T_i) - \Phi(T_{i-1})$ for insertions



$|T_{i-1}|=4$; num(T)=3
$|T_i|=4$; num(T)=4

- Case 1: Without expansion ($|T_i| = |T_{i-1}|$)

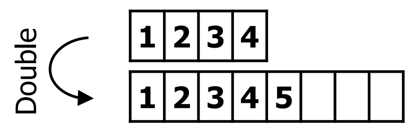
$$\begin{aligned} d_i &= 1 + (2 \cdot num(T_i) - |T_i|) - (2 \cdot num(T_{i-1}) - |T_{i-1}|) \\ &= 1 + 2 \cdot num(T_i) - 2 \cdot num(T_{i-1}) - |T_i| + |T_{i-1}| \\ &= 1 + 2 + 0 \\ &= 3 \end{aligned}$$

One additional element (the 2); No change in capacity (the 0)

Continuation

1: $\Phi(D_i)$ depends on a property of D_i
2: $\Phi(D_n) \geq \Phi(D_0)$
3: $d_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

- Req. 3: Let's look at $d_i = c_i + \Phi(T_i) - \Phi(T_{i-1})$ for insertions



$|T_{i-1}|=4$; num(T)=4
$|T_i|=8$; num(T)=5

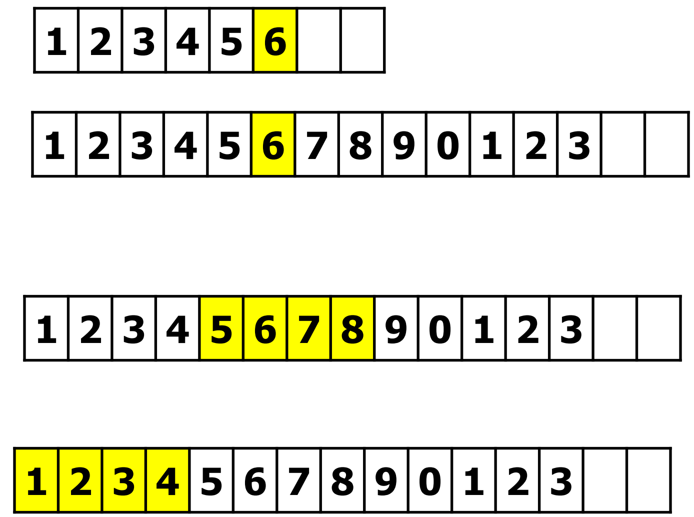
- Case 2: With expansion:

$$
\begin{aligned}
d_i &= num(T_i) + (2 \cdot num(T_i) - |T_i|) - (2 \cdot num(T_{i-1}) - |T_{i-1}|) \\
&= num(T_i) + 2 \cdot num(T_i) - |T_i| - 2 \cdot num(T_{i-1}) + |T_{i-1}| \\
&= num(T_i) + 2 \cdot num(T_i) - 2 \cdot (num(T_i) - 1) - 2 \cdot (num(T_i) - 1) + num(T_i) - 1 \\
&= 3 \cdot num(T_i) - 2 \cdot num(T_i) + 2 - 2 \cdot num(T_i) + 2 + num(T_i) - 1 \\
&= 3
\end{aligned}
$$

- In both cases $3n \geq \sum d_i \geq \sum c_i$ and Q is in O(n)

Intuition

- For inserts, we deposit 3 because
  - 1 for the insertion (the real cost)
  - 1 for the time when we need to copy this new element at the next expansion
    - These 1's fill the account with $|T_i|/2$ before the next expansion
  - 1 for one of the $|T_i|/2$ elements already in A after the last expansion
    - These fill the account with another $|T_i|/2$ before the next expansion
- Thus, we have enough credit at the next expansion

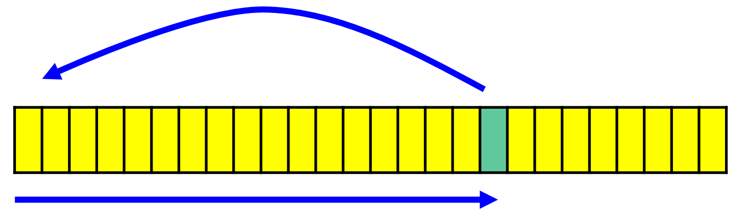
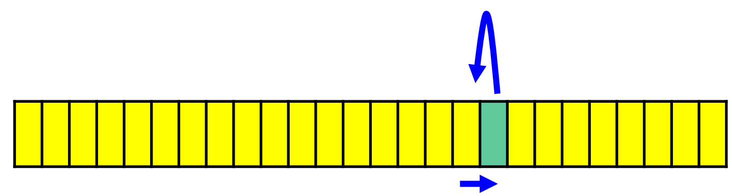
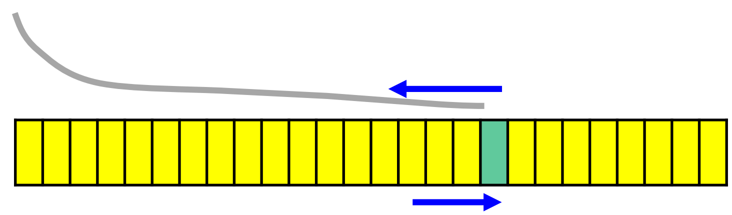


Problem: Deletions

- Our strategy for deletions so far is not very clever
  - Assume a table with num(T)=|T|
  - Assume a sequence Q = {I,D,I,D,I,D,I …}
  - This sequence will perform |T|+|T|/2+|T|+|T|/2+ … real ops
  - As |T| is O(n), this Q really is in $O(n^2)$ and not in O(n)
- Simple trick: Do only contract when num(T)=|T|/4
  - Leads to amortized cost of O(n) for any sequence of operations
  - We omit the proof (see [Cor03])

- Two Examples
- Two Analysis Methods
- Dynamic Tables
- SOL – Analysis
  - Goal and idea
  - Preliminaries
  - A short proof

Re-Organization Strategies

- Recall self-organizing lists (SOL)
  - Accessing the i'th element costs i
  - After searching an element, we change the list L
- Three strategies
  - MF, move-to-front: 
  - T, transpose: 
  - FC, frequency count: 

Notation

- Assume we have a strategy A and a workload S on list L
- After accessing 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 of step l when using strategy A
    - This means: F_{MF}/X_{MF} for strategy MF, F_T/X_T … F_{FC}/X_{FC}
    - Note: Our three strategies never back-swap: $\forall l$: $X_{MF}(l)=X_T(l)=X_{FC}(l)=0$
      - But a new strategy A could
- Let $C_A(S)$ be the total access cost of A incurred by S
  - Again: C_{MF} for strategy MF, C_T for T, C_{FC} for FC
- With conventional worst-case analysis, we can only derive that $C_A(S)$ is in $O(|S|*|L|)$ – for any A
  - Searched element always at last positions, swaps ignored

Theorem

- Theorem (Amortized costs)
  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

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

- What does this mean?
  - We don't learn more about the absolute complexity of SOLs
  - But we learn that MF is quite good
  - Any strategy with the same constraints (only series of swaps) will at best be roughly twice as good as MF
    - Assuming $C_A(S) >> |S|$ and for $|S| \rightarrow \infty$: $X(S) < F(S)$ for any strategy
  - Despite its simplicity, MF is a fairly safe bet for all workloads

Idea of the Proof

- We will compare access costs in L between MF and any A
- Think of both strategies (MF, A) 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 list states by looking at the number of inversions ("Fehlstellungen")
  - Actually, we only analyze how the number of inversions changes
- We will show that the number of inversions defines a potential of a pair of lists that helps to derive an upper bound on the differences in real costs

Content of this Lecture

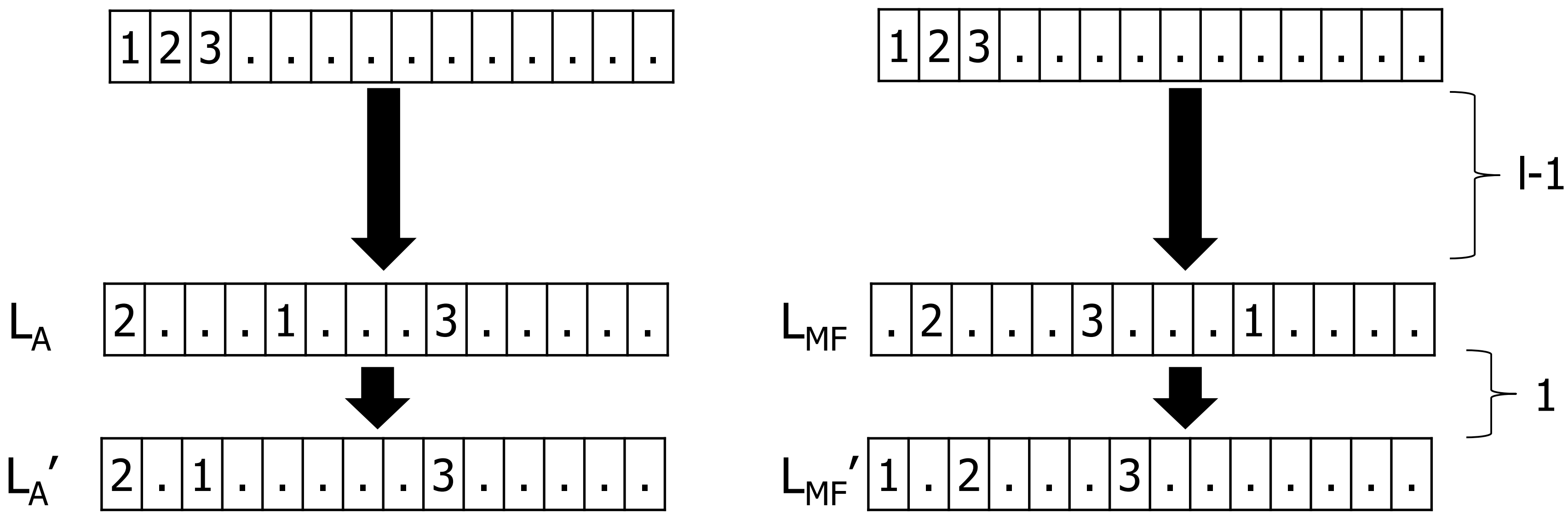
- Two Examples
- Two Analysis Methods
- Dynamic Tables
- SOL - Analysis
  - Goal and idea
  - Preliminaries
  - A short proof

Inversions

- Let L and L�much be permutation of the set {1, 2, …, 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 $1 \leq i,j \leq n$ and $i \neq j$)*
  - *The number of inversions between L and L' is denoted by inv(L, L')*
- Remarks
  - Different order: Once i before j, once i after j
  - Obviously, inv(L, L') = inv(L', L)
  - Example: inv({4,3,1,5,7,2,6}, {3,6,2,5,1,4,7}) = 12
- Without loss of generality, we assume that L={1,…,n}
  - Because we only look at changes in number of inversions and not at the actual set of inversions

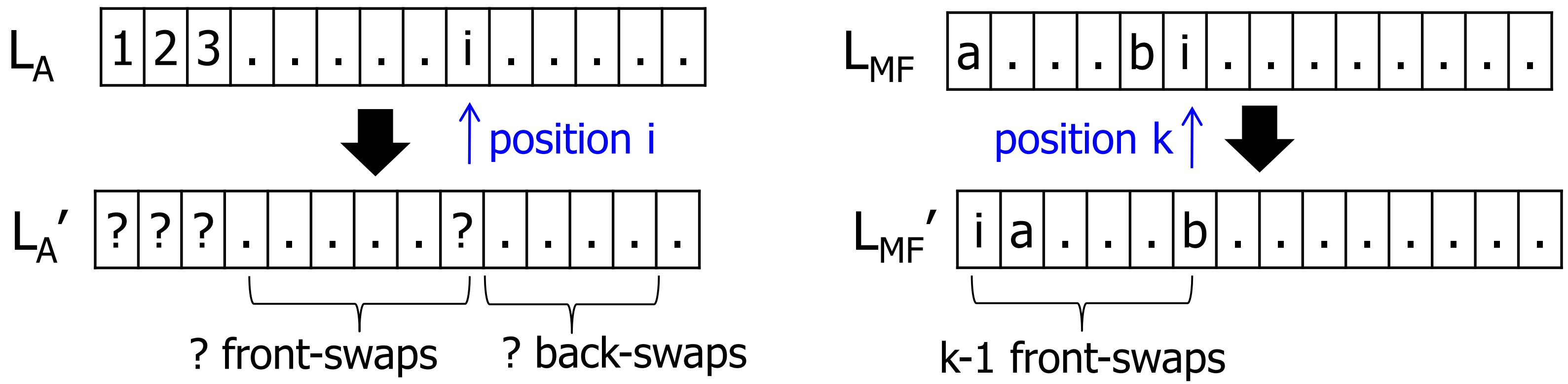
Sequences of Changes

- Assume we applied l-1 steps of S on L, creating L_{MF} using MF, and L_A using A
- Let us consider the next step l, creating L_{MF}' and L_A'

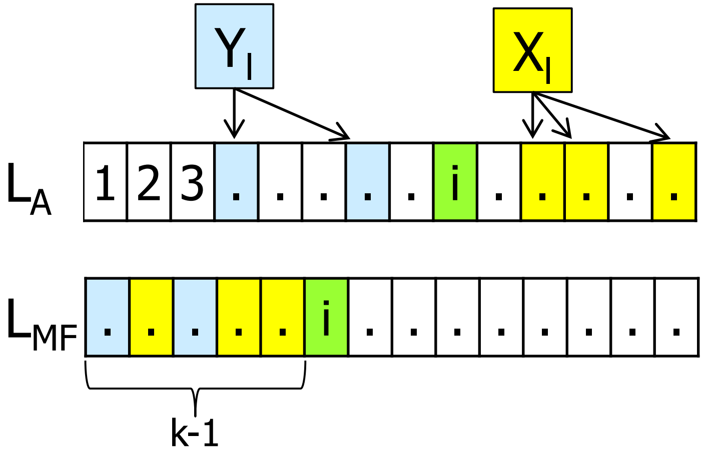


Inversion Changes

- How does step l change the number of inv's between L_{MF}/L_A?
- We compute inv(L_{MF}', L_A') from inv(L_{MF}, L_A)
  - Assume step l accesses element i from L_A
  - We may assume it is at position i
  - Let this element i be at some position k in L_{MF}
  - Access in L_A costs i, access in L_{MF} costs k
  - After step l, A performs an unknown number of swaps; MF performs exactly k-1 front-swaps



Counting Inversion Changes 1



- Let X_l be the set of values that are before position k in L_{MF} and after position i in L_A
- Let Y_l be the values before pos. k in L_{MF} and before i in L_A
  - Clearly, $|X_l| + |Y_l| = k-1$
- All pairs (i,c) with $c \in X_l$ are inversions between L_A and L_{MF}
  - There may be more; but only inv's with i are affected in this step
- After step l, MF moves element i to the front
  - Assume first that A does simply nothing
  - All inversions (i,c) with $c \in X_l$ disappear (there are $|X_l|$ many)
  - But $|Y_l|=k-1-|X_l|$ new inversions appear
  - Thus: $inv(L_{MF}',L_A') = inv(L_{MF},L_A) - |X_l| + k-1-|X_l|$
  - But A does something

Counting Inversion Changes 2

L_A	1	2	3	.	.	.	j	.	i	.	.	.	.	.
L_{MF}`	i	.	.	.	.	.	.	.	.	.	.	.	.	.

- Assume: In step l, let A perform $F_A(l)$ front-swaps and $X_A(l)$ back-swaps
- Every front-swap (swapping i before any j) in L_A decreases $inv(L_{MF}',L_A')$ by 1
  - Before step l, 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 in L_{MF}')
  - After step l, i is before j in both L_A' and L_{MF}' – inversion removed
- Equally, every back-swap increases $inv(L_{MF}',L_A')$ by 1
- Together: After step l, we have

$$inv(L_{MF}',L_A') = \underbrace{inv(L_{MF},L_A)}_{\text{Before step l}} \underbrace{- |X_l| + k-1-|X_l|}_{\text{through MF}} \underbrace{- F_A(l) + X_A(l)}_{\text{through A}}$$

Amortized Costs

Was c_l … was d_l … we switch to Cor09 notation

- Let $t_{MF}(l)$ be the real cost of strategy MF for step l
- We use the number of inversions as potential function $\Phi(L_A,L_{MF})=inv(L_A, L_{MF})$ on the pair L_A, L_{MF}
- Definition
  - *The amortized costs of step l, called a_l, are*

    $$a_l = t_{MF}(l) + inv(L_A(l), L_{MF}(l)) - inv(L_A(l-1), L_{MF}(l-1))$$
  - *Accordingly, the amortized costs of sequence S, |S|=m, are*

    $$\Sigma a_l = \Sigma t_{MF}(l) + inv(L_A(m), L_{MF}(m)) - inv(L_A(0), L_{MF}(0))$$
- This is a proper potential function
  - 1: Φ depends on a property of the pair L_A, L_{MF}
  - 2: inv() can never be negative, so $\forall l$: $\Phi(L_A(l), L_{MF}(l)) \geq \Phi(L,L)=0$
- Let's look at how operations change the potential

Content of this Lecture

- Two Examples
- Two Analysis Methods
- Dynamic Tables
- SOL - Analysis
  - Goal and idea
  - Preliminaries
  - A short proof (after much preparatory work)

Putting it Together

- We know for every step l from S accessing some i: $inv(L_{MF}', L_A') = inv(L_{MF}, L_A) - |X_l| + k-1-|X_l| - F_A(l) + X_A(l)$ and thus $inv(L_{MF}', L_A') - inv(L_{MF}, L_A) = -|X_l|+k-1-|X_l| - F_A(l) + X_A(l)$
- Since $t_{MF}(l)=k$, we get amortized costs of

$$\begin{aligned} a_l &= t_{MF}(l) + inv(L_A', L_{MF}') - inv(L_A, L_{MF}) \\ &= k - |X_l|+k-1-|X_l| - F_A(l) + X_A(l) \\ &= 2(k-|X_l|) - 1 - F_A(l) + X_A(l) \end{aligned}$$

- Recall that Y_l ($|Y_l|=k-1-|X_l|$) are those elements before i in both lists. This implies that $k-1-|X_l| \leq i-1$ or $k-|X_l| \leq i$
  - There can be at most i-1 elements before position i in L_A
- Therefore: $a_l \leq 2i - 1 - F_A(l) + X_A(l)$

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 number of inversions that are affected

- Recall that Y_l ($|Y_l|=k-1-|X_l|$) are those elements before i in both lists. This implies that $k-1-|X_l| \leq i-1$ or $k-|X_l| \leq i$
  - There can be at most i-1 elements before position i in L_A
- Therefore: $a_l \leq 2i - 1 - F_A(l) + X_A(l)$

Aggregating

- We also know the real cost of accessing i using A: $t_A(l)=i$
- Together: $a_l \leq 2t_A(l) - 1 - F_A(l) + X_A(l)$
- Aggregating this inequality over all a_l in S, we get

$$\Sigma a_l \leq 2*C_A(S) - |S| - F_A(S) + X_A(S)$$

- By definition, we also have (m=|S|)

$$\Sigma a_l = \Sigma t_{MF}(l) + inv(L_A{}^m, L_{MF}{}^m) - inv(L_A{}^0, L_{MF}{}^0)$$

- Since $\Sigma t_{MF}(l) = C_{MF}(S)$ and $inv(L_A{}^0, L_{MF}{}^0)=0$, we get

$$C_{MF}(S) + inv(L_A{}^m, L_{MF}{}^m) \leq 2*C_A(S) - |S| - F_A(S) + X_A(S)$$

- It finally follows (inv()≥0)

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

Summary

- Looking at sequences of operations with self-organization creates a new class of problem
  - Things change during a workload
  - These changes (positively) influence future costs of operations
  - Not at random – we follow a strategy
- 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)
  - But beware the assumptions (e.g., only single swaps)
  - Original work: Sleator, D. D. and Tarjan, R. E. (1985). "Amortized efficiency of list update and paging rules." *Communications of the ACM* ***28(2): 202-208.***