Algorithms and Data Structures

Amortized Analysis

Marius Kloft

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

- This lecture is not covered in [OW93] but in [Cor03]

Setting

- SOL setting: Dependent operations
  - 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
  - Changing search order in a workload does not influence WC
- Amortized analysis analyzes the complexity of any sequence of operations of length n
  - Or the worst average cost of each operation in any sequence

Example 1: Multi-Pop

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

Note: 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
  - Within Q, we can at most push O(n) elements and, hence, also only pop O(n) elements
  - Thus, the total cost is in O(n)
- We want to derive such a result in a more systematic manner (analyzing SOLs is not that easy)

Example 2: Bit-Counter

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

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
…		

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

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

- 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 a constant amount 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 of Q
  - Let c_i be the true cost of operation i, d_i its effect on the account
  - We require
$$\forall 1 \leq k \leq n : \sum_{i=1}^{k} c_i \leq \sum_{i=1}^{k} d_i$$
  - Especially, the account must never become negative (or the inequality at this point is broken)
- It follows: An upper bound for the account (d) is also an upper bound for the true cost (c)

Application to mpop

- Assume d_{push}=2, d_{pop}=0, d_{mpop}=0
- Clearly, the account can never become zero
- Summing these up yields an upper bound on the real cost
  - Clearly, d_{push} is an upper bound on c_{push} (which is 1)
  - Idea: Whenever we push an element, we pay 1 for the push and 1 for the operation that will (at same later time) pop exactly this element
    - It doesn't matter whether this will be through a pop or a mpop
  - Thus, when it comes to a pop or mpop, there is always enough money on the account (deposited by previous push's)
- This proves: $$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} d_i \leq 2*n \in O(n)$$

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))

Application to Bit-Counter

- Look at the sequence Q‘ of flips generated by a sequence Q
  - For every +1, we flip exactly once from 0 to 1 and perform a sequence of flips from 1 to 0
    - There is no „flip to 1“ if we roll-over

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
…		

Application to Bit-Counter (Continued)

- Assume $d_{\text{flip-to-1}}$=2 and $d_{\text{flip-to-0}}$=0
  - Clearly, $d_{\text{flip-to-1}}$ is an upper bound to $c_{\text{flip-to-1}}$
  - 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'
  - As we start with only 0 and can backflip any 1 only once, there is always enough money on the account for the flip-to-0's
  - Thus, the account is an upper bound on the actual cost
- As every operation in Q can pay at most 2 (there is at most 1 flip-to-1), Q is in O(n)

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
…		

- 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
- As ops from Q change D, they also change D's potential
- The trick is to design Φ such that we can (again) use it to derive an upper bound on the real cost of Q

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 then can derive the amortized cost of Q as

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

- Rough idea: If we find a Φ such that (a) we 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

Details: Always Pay in Advance

- Operations raise or lower the potential (~future cost) of D
- We need to find a function Φ such that
  - 1: $\Phi(D_i)$ depends on a property of D_i
  - 2: $\Phi(D_n) \geq \Phi(D_0)$ [and we will always have $\Phi(D_0)=0$]
  - 3: We can compute $d_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$ for any possible op
- 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 sequence Q[1-i] is no upper bound in the real costs
- Idea: Always pay in advance

Example: mpop

- We use the number of objects on the stack as its potential
- Then
  - 1: $\Phi(D_i)$ depends on a property of D_i
  - 2: $\Phi(D_n) \geq \Phi(D_0)$ and $\Phi(D_0)=0$
  - 3: Compute $d_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
    - If op is push: $d_i = c_i + 1 = 2$
    - If op is pop: $d_i = c_i - 1 = 0$
    - If op is mpop(k): $d_i = c_i - \#elements\_taken\_from\_stack = 0$

e.g., both equaling k if at least k elements are on stack

- Thus, $2*n \geq \Sigma d_i \geq \Sigma c_i$ and Q is in O(n)

Example: Bit-Counter

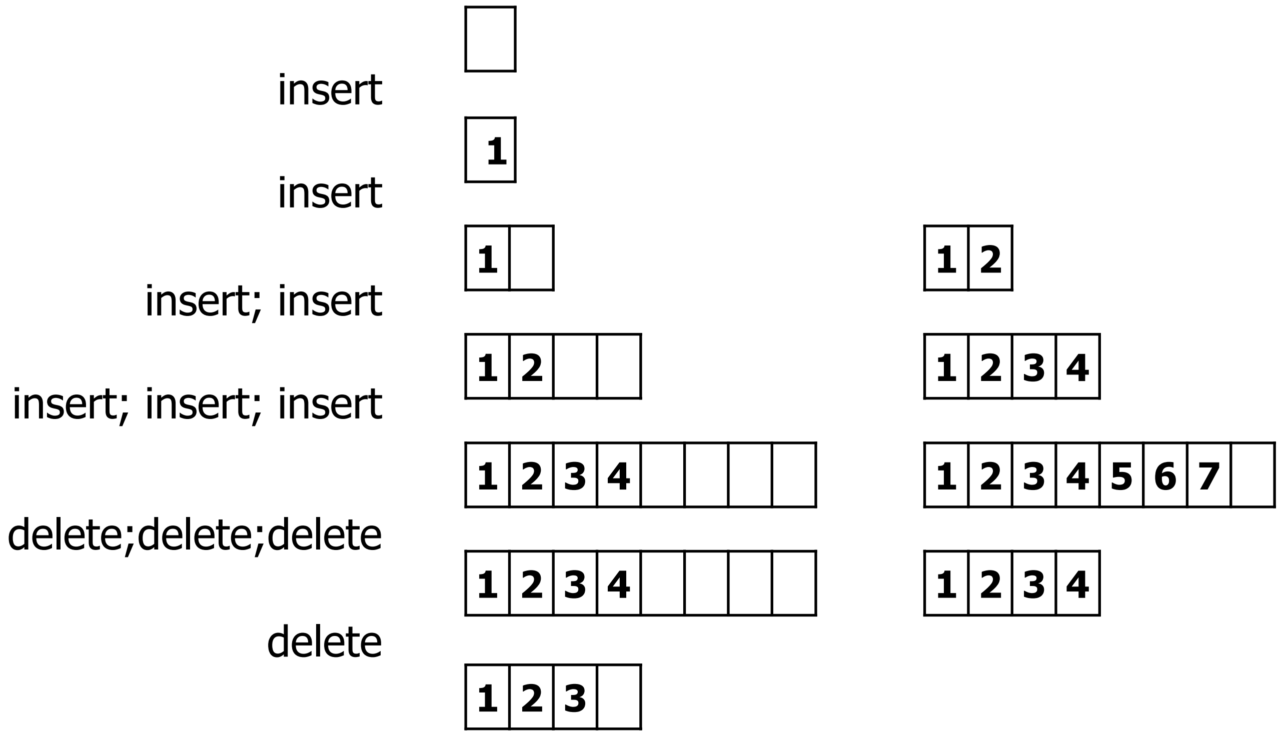
- We use the number of '1's in the bitstring as its potential
- Then
  - 1: $\Phi(D_i)$ depends on a property of D_i
  - 2: $\Phi(D_n) \geq \Phi(D_0)$ and $\Phi(D_0)=0$
  - 3: Compute $d_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
    - Let the i'th operation incur one flip to 1 (or no flip to 1 if roll-over) and t_i flips to 0
    - Thus, $c_i \leq t_i + 1$
    - If $\Phi(D_i)=0$, the this op has flipped all positions to 0, and previously they were all 1 and we had $\Phi(D_{i-1})=k$
    - 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, $d_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (t_i+1) + (\Phi(D_{i-1})-t_i+1) - \Phi(D_{i-1}) = 2$
- 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 - Analysis

Dynamic Tables

- We now use amortized analysis for something more useful: Complexity of operations on a dynamic table
- Assume an array T and a sequence Q of insert/delete ops
- Dynamic Tables: Keep the array small, yet avoid overflows
  - Start with a table T of size 1
  - When inserting but T is full, we double |T|; upon deleting and A is only half-full, we reduce |T| by 50%
  - "Doubling", "reducing" means: Copying data to a new location
  - If the i'th operation is a insertion (or deletion), it costs either 1 or i (depending on whether or not the array is full)
- Conventional WC analysis
  - As i can be up to n for |Q|=n, the complexity of insertion is O(n)
  - Complexity of any Q is $O(n^2)$

Example



insert
1
insert
1
1 2
insert; insert
1 2
1 2 3 4
insert; insert; insert
1 2 3 4
1 2 3 4 5 6 7
delete;delete;delete
1 2 3 4
1 2 3 4
delete
1 2 3

With Potential Method

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

- Let num(T) be the current number of elements in T
- We use potential $\Phi(T) = 2*\text{num}(T) - |T|$
  - Intuitively a "potential"
    - Immediately before an expansion, num(T)=|T| and Φ(T)=|T|, so there is much potential in T (we saved for the expansion to come)
    - Immediately after an expansion, num(T)=|T|/2 and Φ(T)=0; all potential has been used, we need to save again for the next expansion
  - Formally
    - 1: Of course
    - 2: As T is always at least half-full, Φ(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: $d_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

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

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

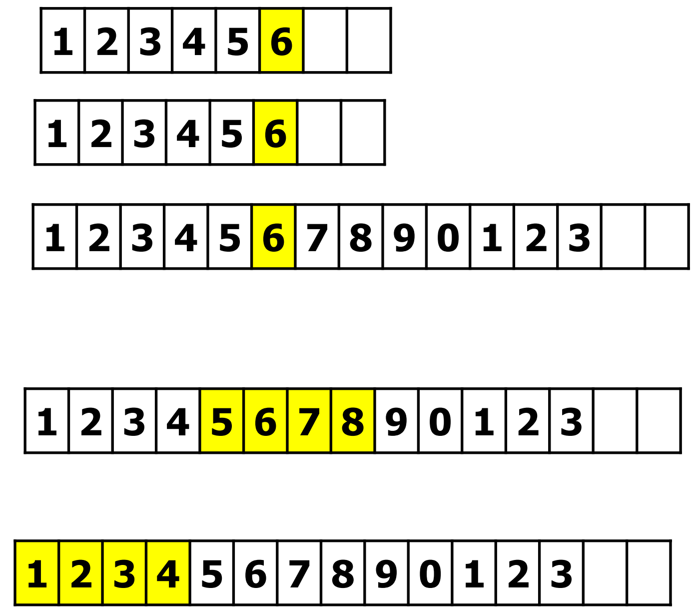
- With expansion

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

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

Intuition

- Consider accounting method
- For insert', we deposit 3 because
  - 1 for the insertion (the real cost)
  - 1 for the time that 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 $|T_i|/2$ 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), Q is in $O(n^2)$ and not in O(n)
- Simple trick: Wait until num(T)=|T|/4, then reduce T by 50%
  - 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

- Think of self-organizing lists again
- When searching an element, we change the list L
  - As usual: Accessing the i'th element costs i
- 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

Notation

- Assume we have an arbitrary strategy A and a sequence S of accesses on list L
- After accessing element i, A may move i as follows
  - Consecutive swaps of i with (adjacent) predecessor (toward front) or successor (toward back)
  - Only swap i (multiple times), do not swap (j,k) with j≠i and k≠i
  - When using strategy A, let $F_A(l)$ be the number of front-swaps of i and $X_A(l)$ the number of back-swaps of i in step l
    - This means: 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 cost of A incurred by S
  - Again: C_{MF} for strategy MF, C_T for T, C_{FC} for FC
- Conventional WC analysis gives $\forall A$: $C_A(S)$ is in $O(|S|*|L|)$

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 A / MF
  - But we learn that MF is quite good
  - Any strategy following the same constraints (only series of swaps) will at best be roughly twice as good as MF
    - Usally $X_A(S)=0$
  - Despite its simplicity, MF is a fairly safe bet in whatever circumstances (= sequences)

Idea of the Proof

- We will compare access costs in L using MF and any 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 list states by looking at the number of inversions (“Fehlstellungen”)
  - Actually, we shall only analyze how the number of invs 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

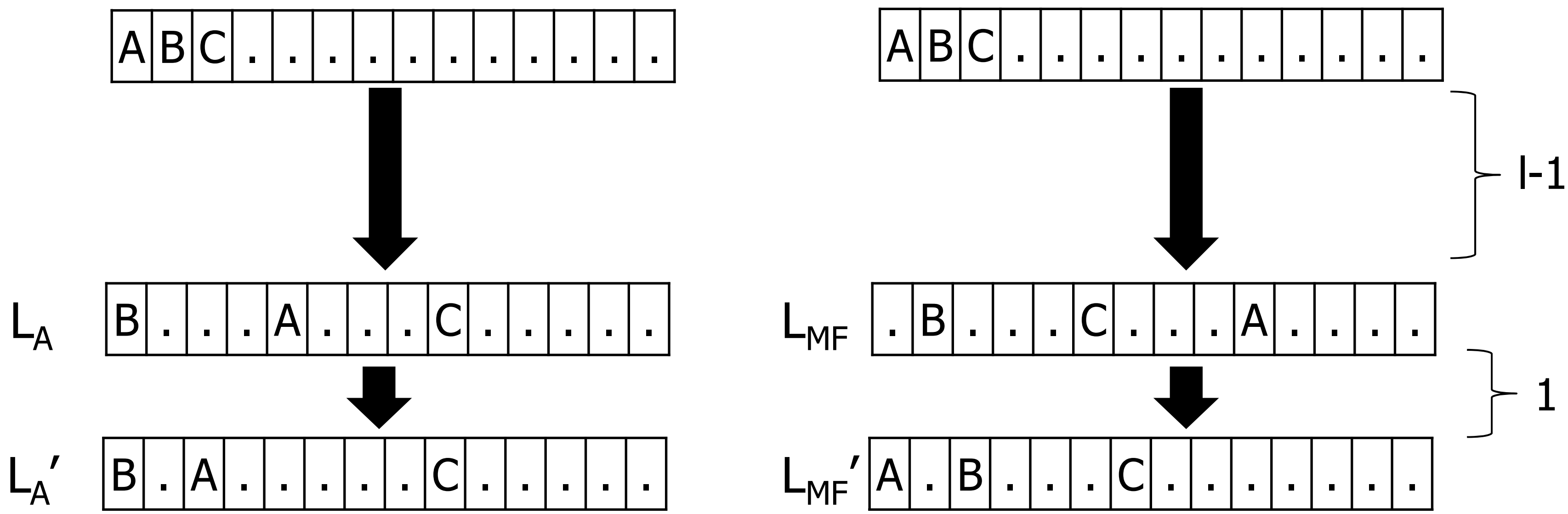
- Two Examples
- Two Analysis Methods
- Dynamic Tables
- SOL - Analysis
  - Goal and idea
  - Preliminaries
  - A short proof

Inversions

- Let L and L‵ be permutations of the set {1, 2, …, n}
- Definition
  - *An unordered 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 $1 \leq i < j \leq n$)*
  - *The number of inversions between L and L' is written inv(L, L')*
- Remarks
  - Different order: Once i before j, once i after j
  - Obviously, inv(L, L') = inv(L', L)
- Examples: inv((1,2,3), (2,3,1)) = |{ {1,2}, {1,3} }| = 2
  - inv((1,...,n), (n,...,1)) = n(n-1)/2
- 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

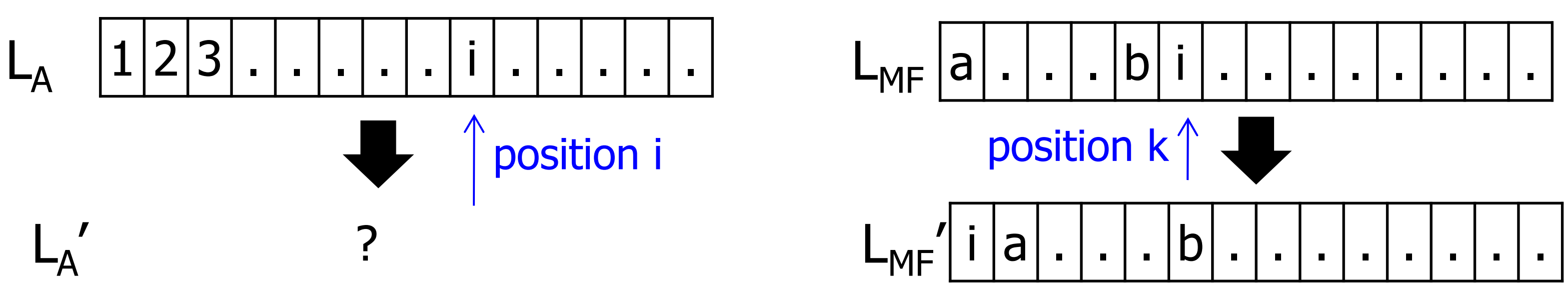
Sequences of Changes

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

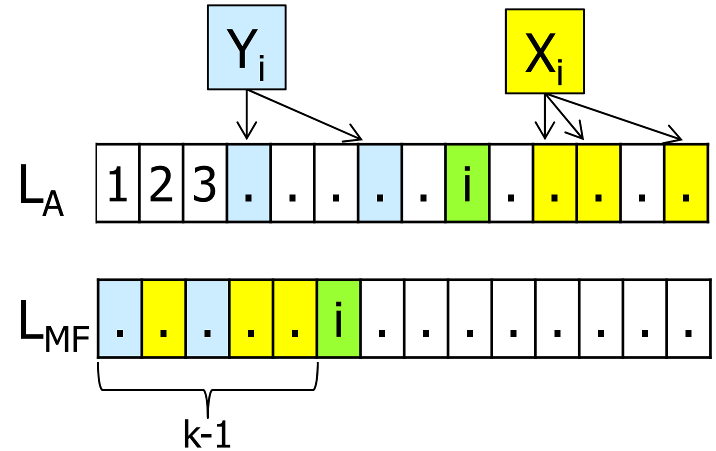


Inversion Changes

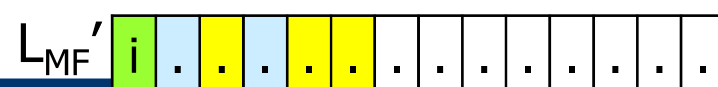
- How does l change the number of inv's between L_{MF} / L_A?
- Can we compute inv(L_{MF}', L_A') from inv(L_{MF}, L_A)?
  - Assume step l accesses element i from L_A
  - We may assume it is at position i
  - Let this element i be at position k in L_{MF}
  - Access in L_A costs i, access in L_{MF} costs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



Counting Inversion Changes 1



- Let X_i be the set of values that are before position k in L_{MF} and after position i in L_A
- Le Y_i be the values before position k in L_{MF} and before i in L_A
  - Clearly, $|X_l| + |Y_l| = k-1$
- All pairs {i,c} with $c \in X_i$ are inversions between L_A and L_{MF}
  - There may be more; but only those 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 $c \in X_i$ 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	.	.	.	.	.	.	.	.	.	.	.	.	.

- 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}'
- 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 OW notation

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

$$d_l = c_l + inv(L_A^l, L_{MF}^l) - inv(L_A^{l-1}, L_{MF}^{l-1})$$

  - *Accordingly, the amortized costs of sequence S, |S|=m, are*

$$\sum d_l = \sum c_l + inv(L_A^m, L_{MF}^m) - inv(L_A^0, L_{MF}^0)$$

- 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 $\Phi(L_A^n,L_{MF}^n) \geq \Phi(L,L)=0$
- 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 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
  $inv(L_{MF}', L_A') - inv(L_{MF}, L_A) = -|X_l| + k-1-|X_l| - F_A(l) + X_A(l)$
- Using the fact that $c_l = k$ for MF, we get amortized costs of

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

- Recall that $|Y_l| = k-1-|X_l|$ 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: $d_l \leq 2i - 1 - F_A(l) + X_A(l)$

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 $|Y_l|=k-1-|X_l|$ 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: $d_l \leq 2i - 1 - F_A(l) + X_A(l)$

Aggregating

- We also know the cost of accessing i using A: that's i
- Together: $d_l \leq 2C_A(l) - 1 - F_A(l) + X_A(l)$
- Aggregating this inequality over all a_l (hence S), we get

$$\sum d_l \leq 2*C_A(S) - |S| - F_A(S) + X_A(S)$$

- By definition, we also have

$$\sum d_l = \sum c_l + inv(L_A^m, L_{MF}^m) - inv(L_A^0, L_{MF}^0)$$

- Since $\sum c_l = C_{MF}(S)$ 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 $(inv() \geq 0)$

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

Summary

- Self-organization creates a type of problem we were not confronted with before
  - Things change during program execution
  - But 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.***