# Algorithms and Data Structures

## Amortized Analysis

Ulf Leser

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

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

# Setting

- Note: We study a special setting - not rare, yet novel here
- We have a sequence Q of operations on a data structure
  - Searching and rearranging a SOL
- Operations are not independent – by changing the data structure, cost of subsequent operations is influenced
- A conventional WC-analysis produces misleading results
- Amortized analysis analyzes the complexity of any sequence of operations of length n
  - Or the worst-case average cost of an operation in any sequence
  - This is not the classical average case – we do not study an "average" sequence, but the average 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, than 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
- 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
- Every operation deposits or withdraws from the account
- We choose the amounts deposited/withdrawn 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

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

  - In particular, the account must never become negative
- It follows: An upper bound for the account is also an upper bound for the true cost

# Application to mpop

- Assume $d_{push}$=2, $d_{pop}$=0, $d_{mpop}$=0
- Clearly, the account of any Q can never be zero
- We show that the sum of these costs yields an upper bound on the actual cost
  - Clearly, $d_{push}$ is an upper bound on $c_{push}$ (which is 1)
  - Whenever we push an element, we pay 1 for the actual cost and 1 for the operation that will (at same later time) pop exactly this element
    - It doesn't matter whether this will be in 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: $$\sum_{i=1}^{n} c_i \le \sum_{i=1}^{n} d_i \le 2 * n \in O(n)$$

# Application to Bit-Counter

- Look at the sequence Q‘ of flips generated by a sequence Q
  - Every +1 corresponds to a sequence of 0-k flip-to-0 and 0-1 flip-to-1
    - There is no „flip to 1“ if we roll-over
- Assume $d_{flip\text{-}to\text{-}1}$=2 and $d_{flip\text{-}to\text{-}0}$=0
  - Clearly, $d_{flip\text{-}to\text{-}1}$ is an upper bound to $c_{flip\text{-}to\text{-}1}$
  - 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)

- 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 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 $\Phi$ 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 derive the amortized cost of Q as

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

- If we find a $\Phi$ such that (a) we get formulas for the amortized costs of all $d_i$ and (b) $\Phi(D_n) \geq \Phi(D_0)$, we have an upper bound for the real costs

# Always Pay in Advance

- Operations raise or lower the potential of D
- We need to find a function Φ such that
  - 1: $\Phi(D_i)$ depends on a property of D
  - 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
- We need to 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
  - 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): $d_i = c_i - k = 0$
- 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
  - 1: $\Phi(D_i)$ depends on a property of D
  - 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 $t_i$ flip-to-0 and 0 or 1 flip-to-1
    - 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 have $\Phi(D_{i-1})=k$
    - 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 - 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
- We want to keep the array small, yet avoid overflows
- Method: When inserting and T is full, we double |T|; upon deleting and A is only half-full, we reduce |T| by 50%
  - Actually, deletion will be handled differently later
- Thus, if the i'th operation is a insertion (or deletion), it costs either 1 or i; as i can be up to n, the complexity of insertion is O(n); thus, the complexity of Q is $O(n^2)$
  - Assuming that we copy A to a new location for growing / shrinking

# Amortized Analysis with Potential Method

- Let num(T) be the current number of elements in T
- We use potential $\Phi(T) = 2*num(T) - |T|$
  - 1: Of course
  - 2: As T is always at least half-full, $\Phi(T)$ is always $\geq 0$
  - 2: We start with $|T|=0$, and thus $\Phi(T_n)-\Phi(T_0)\geq 0$
  - 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, $num(T)=|T|/2$ and $\Phi(T)=0$; all potential has been used, we need to save again for the next expansion

# Operations

- 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

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

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

# Why 3

- There is an intuitive explanation for the cost of exactly 3
- Think in terms of accounting method
- When we insert an element, we deposit 3 on the account
  - 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
    - 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: 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

- 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 a self-organizing strategy A and a sequence $S=\{s_l\}$ of accesses to L
- 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
- With conventional worst-case analysis, we can only derive that $C_A(S)$ is in $O(|S|*|L|)$ – for any strategy

# 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
    - Assuming $C(S) >> |S|$ and for $|S| \rightarrow \infty$: $X(S) \sim F(S)$ for any strategy
  - 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 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 and determine a certain property – the number of inversions ("Fehlstellungen")
  – Actually, we only analyze how the number of inversion changes
- We will show that the number of inversions define a potential of a pair of lists and can be used to derive an upper bound on the differences in real costs

# Content of this Lecture

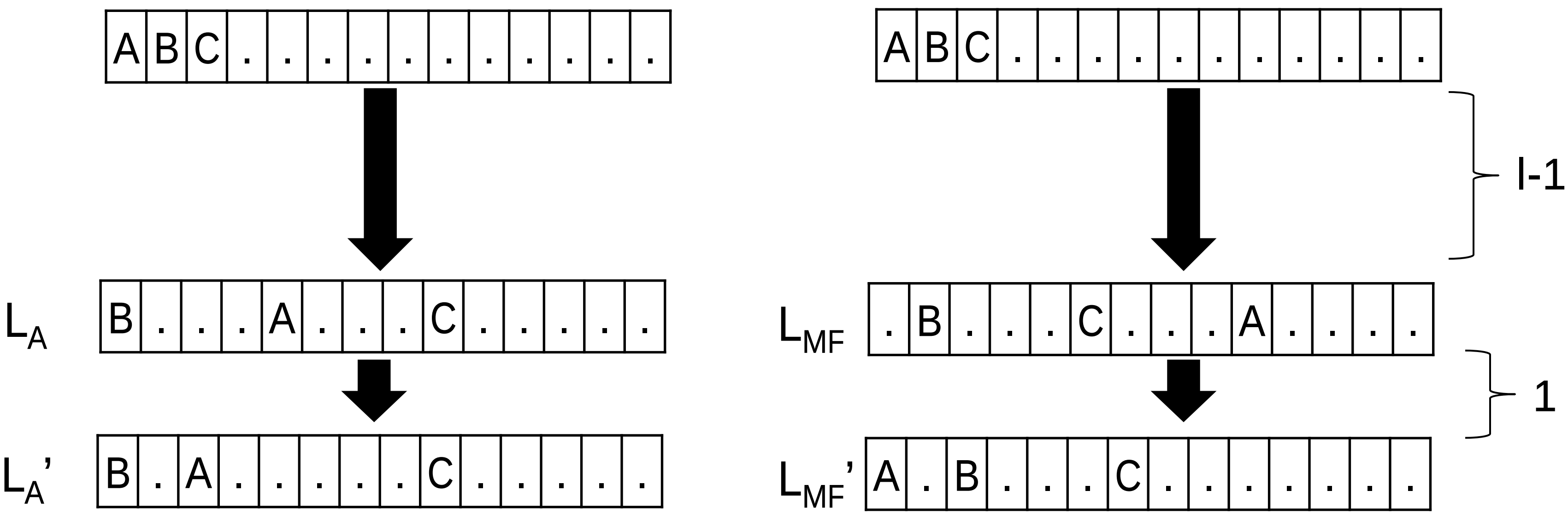
- Two Examples
- Two Analysis Methods
- Dynamic Tables
- SOL - Analysis
  - Goal and idea
  - Preliminaries
  - A short proof

# Inversions

- Let L and L‘ be 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 $0 \le i,j \le n$ and $i \neq j$)*
  - *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)
- 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

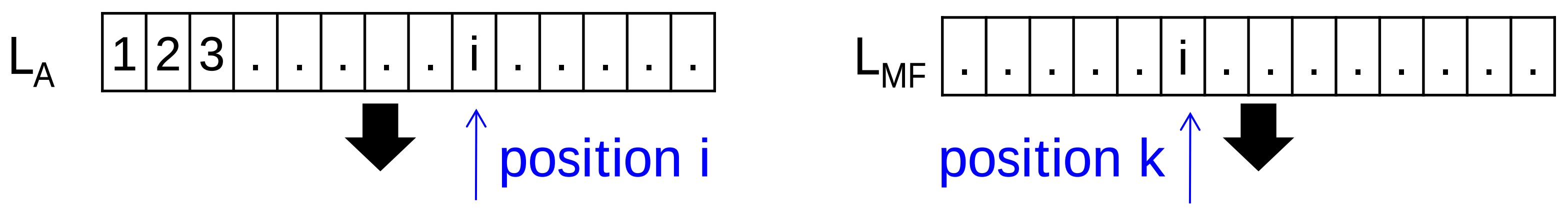
# Sequences of Changes

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

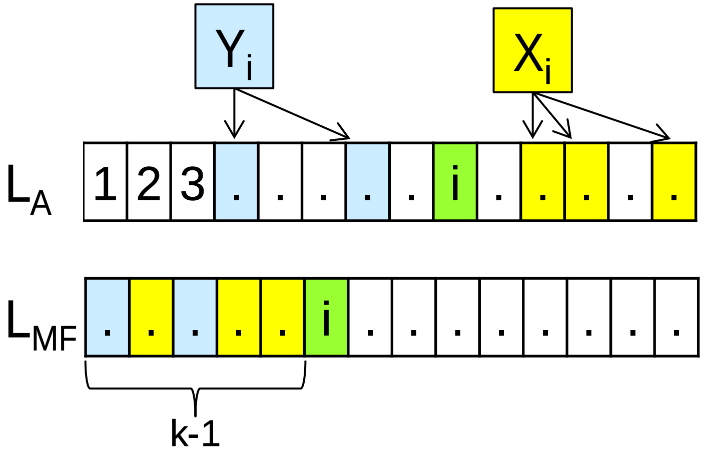


# Inversion Changes

- How does I change the number of inversions between $L_{MF}$ and $L_A$; can we compute inv($L_{MF}$', $L_A$') from inv($L_{MF}$, $L_A$)?
  – Assume I 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 I, 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, $|X_i| + |Y_i| = k\text{-}1$
- All pairs (i,c) with $c \in X_i$ are inversions between $L_A$ and $L_{MF}$
  - There may be more; but only those including i are affected
- After step I, MF moves element i to the front
  - All inversions (i,c) with $c \in X_i$ disappear (there are $|X_i|$ many)
  - But $|Y_i|=k\text{-}1\text{-}|X_i|$ new inversions appear
- If A did nothing: $inv(L_{MF}', L_A') = inv(L_{MF}, L_A) - |X_i| + k\text{-}1\text{-}|X_i|$
  - But A is doing something

# Counting Inversion Changes 2

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $L_A$ | 1 | 2 | 3 | . | . | . | j | . | i | . | . | . | . | . |
| $L_{MF}$‘ | i | . | . | . | . | . | . | . | . | . | . | . | . | . |

- In step l, let A perform $F_A(i)$ front-swaps and $X_A(i)$ back-swaps
- Every front-swap (swapping i before j) in $L_A$ decreases $inv(L_{MF}',L_A')$ 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 in $L_{MF}$')
  - After the swap, 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

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

# Amortized Costs

Was $c_l$ ... was $d_l$ ... we switch to OW notation

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

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

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

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

- 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: $inv(L_{MF}',L_A') = inv(L_{MF},L_A) - |X_i| + k-1-|X_i| - F_A(i) + X_A(i)$ and thus $inv(L_{MF}',L_A') - inv(L_{MF},L_A) = -|X_i|+k-1-|X_i| - F_A(i) + X_A(i)$
- Using the fact that $t_l=k$ for MF, we get amortized costs of

$$\begin{aligned} a_l &= t_l + inv(L_A', L_{MF}') – inv(L_A, L_{MF}) \\ &= k - |X_i|+k-1-|X_i| - F_A(i) + X_A(i) \\ &= 2(k-|X_i|) - 1 - F_A(i) + X_A(i) \end{aligned}$$

- 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 number of inversions that are affected

- 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: $a_l \leq 2i - 1 - F_A(i) + X_A(i)$

# Aggregating

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

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

- By definition, we also have

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

- Since $\sum t_l = C_{MF}(S)$ 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)
  - 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.***