11. Fundamental Data Structures

Abstract data types stack, queue, implementation variants for linked lists [Ottman/Widmayer, Kap. 1.5.1-1.5.2, Cormen et al, Kap. 10.1.-10.2]

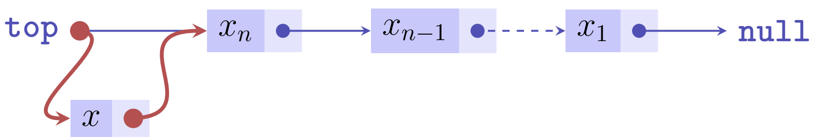
Abstract Data Types

We recall

A *stack* is an abstract data type (ADR) with operations

- `push`(x, S): Puts element x on the stack S.
- `pop`(S): Removes and returns top most element of S or `null`
- `top`(S): Returns top most element of S or `null`.
- `isEmpty`(S): Returns `true` if stack is empty, `false` otherwise.
- `emptyStack`$()$: Returns an empty stack.

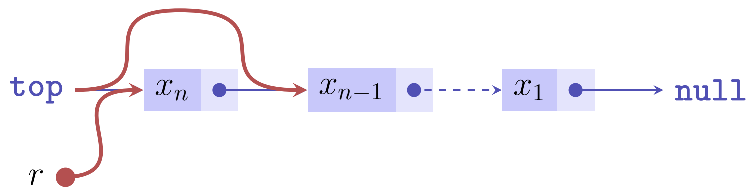
Implementation Push



`push`(x, S):

1. Create new list element with x and pointer to the value of `top`.
2. Assign the node with x to `top`.

Implementation Pop



`pop`(S):

1. If `top`=`null`, then return `null`
2. otherwise memorize pointer p of `top` in r.
3. Set `top` to $p.next$ and return r

Analysis

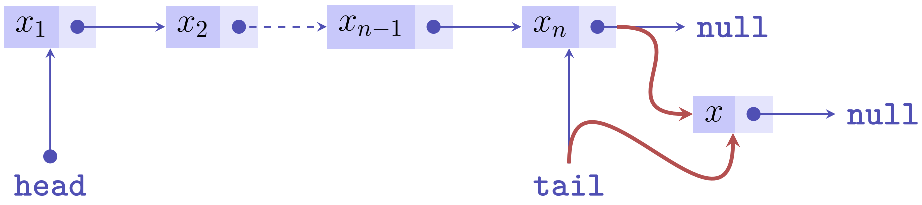
Each of the operations `push`, `pop`, `top` and `isEmpty` on a stack can be executed in $\mathcal{O}(1)$ steps.

Queue (fifo)

A queue is an ADT with the following operations

- `enqueue`(x, Q): adds x to the tail (=end) of the queue.
- `dequeue`(Q): removes x from the head of the queue and returns x (`null` otherwise)
- `head`(Q): returns the object from the head of the queue (`null` otherwise)
- `isEmpty`(Q): return `true` if the queue is empty, otherwise `false`
- `emptyQueue`$()$: returns empty queue.

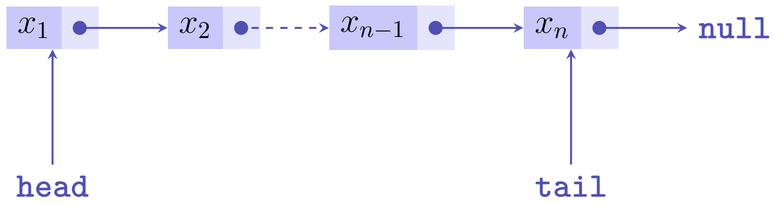
Implementation Queue



`enqueue`(x, S):

1. Create a new list element with x and pointer to `null`.
2. If `tail` $\neq$ `null`, then set `tail.next` to the node with x.
3. Set `tail` to the node with x.
4. If `head` $=$ `null`, then set `head` to `tail`.

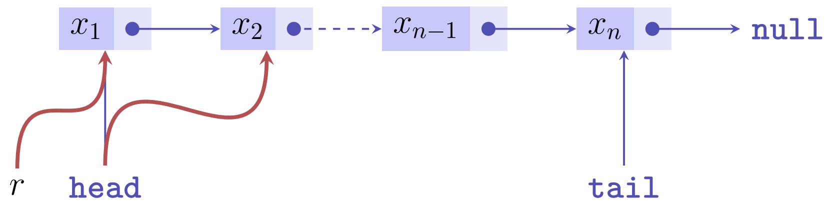
Invariants



With this implementation it holds that

- either `head` $=$ `tail` $=$ `null`,
- or `head` $=$ `tail` $\neq$ `null` and `head.next` $=$ `null`
- or `head` $\neq$ `null` and `tail` $\neq$ `null` and `head` $\neq$ `tail` and `head.next` $\neq$ `null`.

Implementation Queue



dequeue(S):

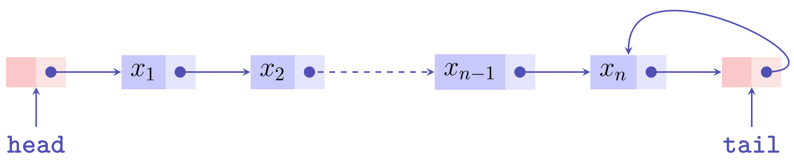
1. Store pointer to head in r. If $r = \text{null}$, then return r .
2. Set the pointer of head to head.next.
3. Is now head $=$ null then set tail to null.
4. Return the value of r.

Analysis

Each of the operations enqueue, dequeue, head and isEmpty on the queue can be executed in $\mathcal{O}(1)$ steps.

Implementation Variants of Linked Lists

List with dummy elements (sentinels).

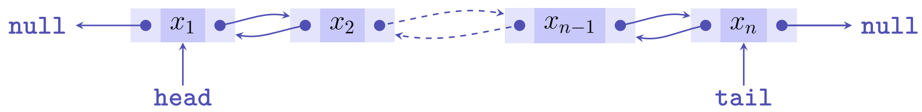


Advantage: less special cases

Variant: like this with pointer of an element stored singly indirect.
(Example: pointer to x_3 points to x_2.)

Implementation Variants of Linked Lists

Doubly linked list



Overview

	enqueue	delete	search	concat
(A)	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
(B)	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
(C)	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
(D)	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$

(A) = singly linked
(B) = Singly linked with dummy element at the beginning and the end
(C) = Singly linked with indirect element addressing
(D) = doubly linked

priority queue

Priority Queue

Operations

- `insert(x,p,Q)`: Enter object x with priority p.
- `extractMax(Q)`: Remove and return object x with highest priority.

Implementation Priority Queue

With a Max Heap

Thus

- `insert` in $\mathcal{O}(\log n)$ and
- `extractMax` in $\mathcal{O}(\log n)$.

12. Amortized Analyis

Amortized Analysis: Aggregate Analysis, Account-Method, Potential-Method [Ottman/Widmayer, Kap. 3.3, Cormen et al, Kap. 17]

Multistack

Multistack adds to the stack operations `push` und `pop`

`multipop(s,S)`: remove the $\min(\text{size}(S), k)$ most recently inserted objects and return them.

Implementation as with the stack. Runtime of `multipop` is $\mathcal{O}(k)$.

Academic Question

If we execute on a stack with n elements a number of n times `multipop(k,S)` then this costs $\mathcal{O}(n^2)$?

Certainly correct because each `multipop` may take $\mathcal{O}(n)$ steps.

How to make a better estimation?

Amortized Analysis

- Upper bound: *average* performance of each considered operation in the *worst case*.

$$\frac{1}{n}\sum_{i=1}^{n}\text{cost}(\text{op}_i)$$

- Makes use of the fact that a few expensive operations are opposed to many cheap operations.
- In amortized analysis we search for a credit or a potential function that captures how the cheap operations can “compensate” for the expensive ones.

Aggregate Analysis

Direct argument: compute a bound for the total number of elementary operations and divide by the total number of operations.

Aggregate Analysis: (Stack)

-
-

$$\sum_{i=1}^{n} \text{cost}(\text{op}_i) \leq 2n$$

$$\textit{amortized cost}(\text{op}_i) \leq 2 \in \mathcal{O}(1)$$

Accounting Method

Model

- The computer is driven with coins: each elementary operation of the machine costs a coin.
- For each operation op_k of a data structure, a number of coins a_k has to be put on an account A: $A_k = A_{k-1} + a_k$
- Use the coins from the account A to pay the true costs t_k of each operation.
- The account A needs to provide enough coins in order to pay each of the ongoing operations op_k: $A_k - t_k \geq 0 \, \forall k$.

$\Rightarrow$ a_k are the amortized costs of op_k.

Accounting Method (Stack)

- Each call of `push` costs 1 CHF and additionally 1 CHF will be deposited on the account. ($a_k = 2$)
- Each call to `pop` costs 1 CHF and will be paid from the account. ($a_k = 0$)

Account will never have a negative balance.

$a_k \leq 2 \, \forall \, k$, thus: constant amortized costs.

Potential Method

Slightly different model

- Define a *potential* Φ_i that is *associated to the state of a data structure* at time i.
- The potential shall be used to level out expensive operations und therefore needs to be chosen such that it is increased during the (frequent) cheap operations while it decreases for the (rare) expensive operations.

Potential Method (Formal)

Let t_i denote the real costs of the operation op_i.

Potential function $\Phi_i \geq 0$ to the data structure after i operations.
Requirement: $\Phi_i \geq \Phi_0 \ \forall i$.

of the ith operation:

$$a_i := t_i + \Phi_i - \Phi_{i-1}.$$

It holds

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (t_i + \Phi_i - \Phi_{i-1}) = \left(\sum_{i=1}^{n} t_i\right) + \Phi_n - \Phi_0 \geq \sum_{i=1}^{n} t_i.$$

Example stack

Potential function Φ_i = number element on the stack.

- push(x, S): real costs $t_i = 1$. $\Phi_i - \Phi_{i-1} = 1$. Amortized costs $a_i = 2$.
- pop(S): real costs $t_i = 1$. $\Phi_i - \Phi_{i-1} = -1$. Amortized costs $a_i = 0$.
- multipop(k, S): real costs $t_i = k$. $\Phi_i - \Phi_{i-1} = -k$. amortized costs $a_i = 0$.

All operations have *constant amortized cost*! Therefore, on average Multipop requires a constant amount of time. [16]

[16] Note that we are not talking about the probabilistic mean but the (worst-case) average of the costs.

Example Binary Counter

Binary counter with k bits. In the worst case for each count operation maximally k bitflips. Thus $\mathcal{O}(n \cdot k)$ bitflips for counting from 1 to n. Better estimation?

Real costs t_i = number bit flips from 0 to 1 plus number of bit-flips from 1 to 0.

$$...0\underbrace{1111111}_{l \text{ Einsen}} + 1 = ...1\underbrace{0000000}_{l \text{ Zeroes}}.$$

$$\Rightarrow t_i = l + 1$$

Binary Counter: Aggregate Analysis

Count the number of bit flips when counting from 0 to $n-1$.

Observation

- Bit 0 flips for each $k-1 \rightarrow k$
- Bit 1 flips for each $2k-1 \rightarrow 2k$
- Bit 2 flips for each $4k-1 \rightarrow 4k$

Total number bit flips $\sum_{i=0}^{n-1} \frac{n}{2^i} \leq n \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$

Amortized cost for each increase: $\mathcal{O}(1)$ bit flips.

Binary Counter: Account Method

Observation: for each increment exactly one bit is incremented to 1, while many bits may be reset to 0. Only a bit that had previously been set to 1 can be reset to 0.

$a_i = 2$: 1 CHF real cost for setting $0 \to 1$ plus 1 CHF to deposit on the account. Every reset $1 \to 0$ can be paid from the account.

Binary Counter: Potential Method

$$...0\underbrace{1111111}_{l \text{ ones}}+1 = ...1\underbrace{0000000}_{l \text{ zeros}}$$

potential function Φ_i: number of 1-bits of x_i.

$$\Rightarrow \Phi_0 = 0 \leq \Phi_i \, \forall i$$
$$\Rightarrow \Phi_i - \Phi_{i-1} = 1 - l,$$
$$\Rightarrow a_i = t_i + \Phi_i - \Phi_{i-1} = l + 1 + (1 - l) = 2.$$

Amortized constant cost for each count operation. ☺

13. Dictionaries

Dictionary, Self-ordering List, Implementation of Dictionaries with Array / List /Skip lists. [Ottman/Widmayer, Kap. 3.3,1.7, Cormen et al, Kap. Problem 17-5]

Dictionary

ADT to manage keys from a set $\mathcal{K}$ with operations

- `insert`(k, D): Insert $k \in \mathcal{K}$ to the dictionary D. Already exists $\Rightarrow$ error messsage.
- `delete`(k, D): Delete k from the dictionary D. Not existing $\Rightarrow$ error message.
- `search`(k, D): Returns `true` if $k \in D$, otherwise `false`

Idea

Implement dictionary as sorted array

Worst case number of fundamental operations

Search	$\mathcal{O}(\log n)$ ☺
Insert	$\mathcal{O}(n)$ ☹
Delete	$\mathcal{O}(n)$ ☹

Other idea

Implement dictionary as a linked list

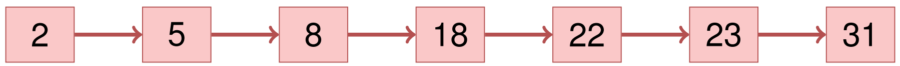
Worst case number of fundamental operations

Search	$\mathcal{O}(n)$ ☹
Insert	$\mathcal{O}(1)$[17] ☺
Delete	$\mathcal{O}(n)$ ☹

[17] Provided that we do not have to check existence.

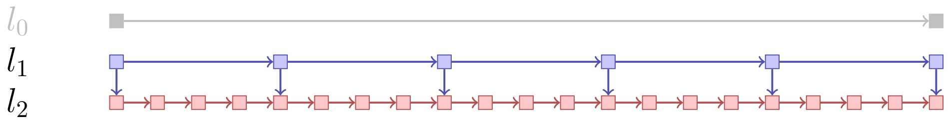
13.1 Skip Lists

Sorted Linked List



Search for element / insertion position: *worst-case* n Steps.

Sorted Linked List with two Levels



- Number elements: $n_0 := n$
- Stepsize on level 1: n_1
- Stepsize on level 2: $n_2 = 1$

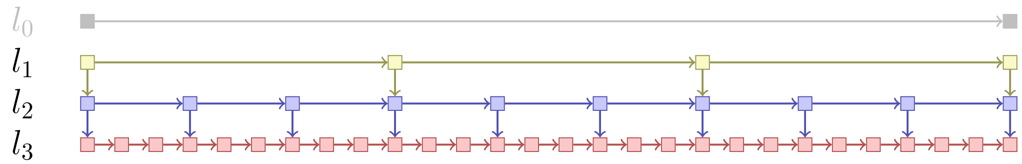
$\Rightarrow$ Search for element / insertion position: worst-case $\frac{n_0}{n_1} + \frac{n_1}{n_2}$.

$\Rightarrow$ Best Choice for[18] n_1: $n_1 = \frac{n_0}{n_1} = \sqrt{n_0}$.

Search for element / insertion position: *worst-case* $2\sqrt{n}$ steps.

[18] Differentiate and set to zero, cf. appendix

Sorted Linked List with two Levels



- Number elements: $n_0 := n$
- Stepsizes on levels $0 < i < 3$: n_i
- Stepsize on level 3: $n_3 = 1$

$\Rightarrow$ Best Choice for (n_1, n_2): $n_2 = \frac{n_0}{n_1} = \frac{n_1}{n_2} = \sqrt[3]{n_0}$.

Search for element / insertion position: *worst-case* $3 \cdot \sqrt[3]{n}$ steps.

Sorted Linked List with k Levels (Skiplist)

- Number elements: $n_0 := n$
- Stepsizes on levels $0 < i < k$: n_i
- Stepsize on level k: $n_k = 1$

$\Rightarrow$ Best Choice for $(n_1, \ldots, n_k)$: $n_{k-1} = \frac{n_0}{n_1} = \frac{n_1}{n_2} = \cdots = \sqrt[k]{n_0}$.

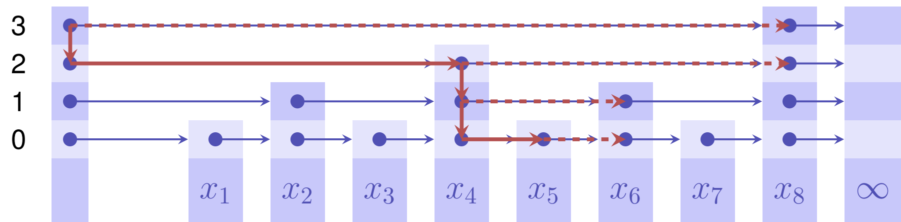
Search for element / insertion position: *worst-case* $k \cdot \sqrt[k]{n}$ steps[19] (Derivation: Appendix).

Assumption $n = 2^k$
$\Rightarrow$ worst case $\log_2 n \cdot 2$ steps and $\frac{n_i}{n_{i+1}} = 2 \, \forall \, 0 \le i < \log_2 n$.

[19] (Herleitung: Anhang)

Search in a Skiplist

Perfect skip list



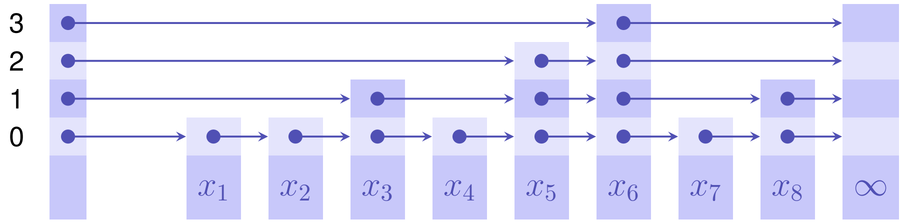
$x_1 \le x_2 \le x_3 \le \cdots \le x_9$.
Example: search for a key x with $x_5 < x < x_6$.

Analysis perfect skip list (worst cases)

Search in $\mathcal{O}(\log n)$. Insert in $\mathcal{O}(n)$.

Randomized Skip List

Idea: insert a key with random height H with $\mathbb{P}(H = i) = \frac{1}{2^{i+1}}$.



Analysis Randomized Skip List

Theorem

The expected number of fundamental operations for Search, Insert and Delete of an element in a randomized skip list is $\mathcal{O}(\log n)$.

The lengthy proof that will not be presented in this courseobserves the length of a path from a searched node back to the starting point in the highest level.

13.2 [Self Ordering]

not covered in class

Self Ordered Lists

Problematic with the adoption of a linked list: linear search time

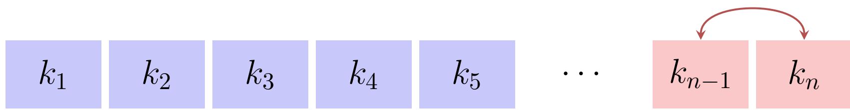
Idea: Try to order the list elements such that accesses over time are possible in a faster way

For example

- Transpose: For each access to a key, the key is moved one position closer to the front.
- Move-to-Front (MTF): For each access to a key, the key is moved to the front of the list.

Transpose

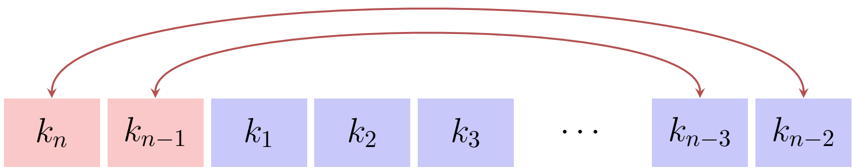
Transpose:



Worst case: Alternating sequence of n accesses to k_{n-1} and k_n.
Runtime: $\Theta(n^2)$

Move-to-Front

Move-to-Front:



Alternating sequence of n accesses to k_{n-1} and k_n. Runtime: $\Theta(n)$

Also here we can provide a sequence of accesses with quadratic runtime, e.g. access to the last element. But there is no obvious strategy to counteract much better than MTF..

Analysis

Compare MTF with the best-possible competitor (algorithm) A. How much better can A be?

Assumptions:

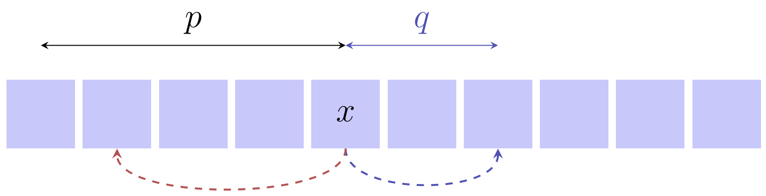
- MTF and A may only move the accessed element.
- MTF and A start with the same list.

Let M_k and A_k designate the lists after the kth step. $M_0 = A_0$.

Analysis

Costs:

- Access to x: position p of x in the list.
- No further costs, if x is moved before p
- Further costs q for each element that x is moved back starting from p.



Amortized Analysis

Let an arbitrary sequence of search requests be given and let $G_k^{(M)}$ and $G_k^{(A)}$ the costs in step k for Move-to-Front and A, respectively. Want estimation of $\sum_k G_k^{(M)}$ compared with $\sum_k G_k^{(A)}$.

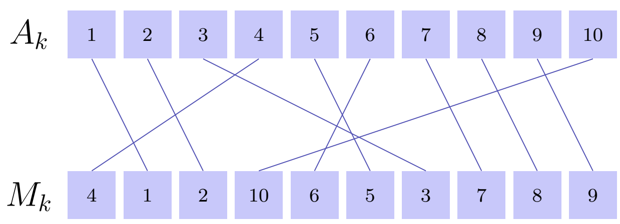
$\Rightarrow$ Amortized analysis with potential function Φ.

Potential Function

Potential function Φ = Number of inversions of A vs. MTF.

Inversion = Pair x, y such that for the positions of a and y

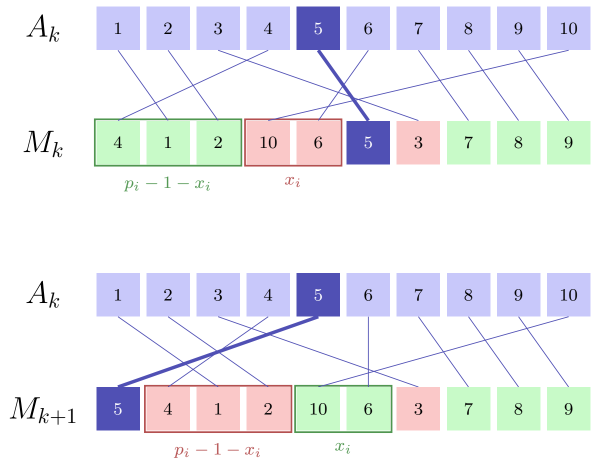
$\left(p^{(A)}(x) < p^{(A)}(y)\right) \neq \left(p^{(M)}(x) < p^{(M)}(y)\right)$



#inversion = #crossings

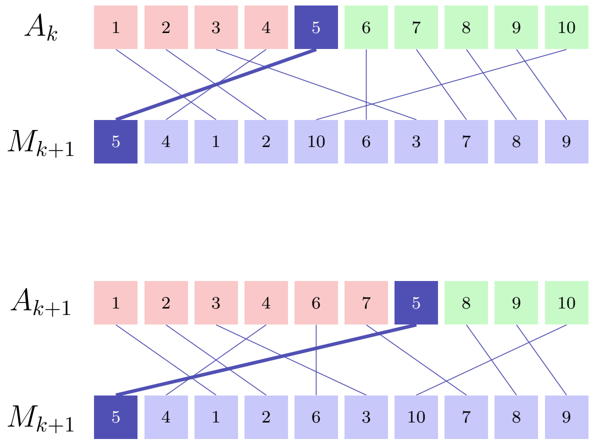
Estimating the Potential Function: MTF

- Element i at position $p_i := p^{(M)}(i)$.
- access costs $C_k^{(M)} = p_i$.
- x_i: Number elements that are in M before p_i and in A after i .
- MTF removes x_i inversions.
- $p_i - x_i - 1$: Number elements that in M are before p_i and in A are before i.
- MTF generates $p_i - 1 - x_i$ inversions.



Estimating the Potential Function: A

- Wlog element i at position $p^{(A)}(i)$.
- $X_k^{(A)}$: number movements to the back (otherwise 0).
- access costs for i: $C_k^{(A)} = p^{(A)}(i) \geq p^{(M)}(i) - x_i$.
- A increases the number of inversions maximally by $X_k^{(A)}$.



Estimation

$$\Phi_{k+1} - \Phi_k \leq -x_i + (p_i - 1 - x_i) + X_k^{(A)}$$

Amortized costs of MTF in step k:

$$\begin{aligned} a_k^{(M)} &= C_k^{(M)} + \Phi_{k+1} - \Phi_k \\ &\leq p_i - x_i + (p_i - 1 - x_i) + X_k^{(A)} \\ &= (p_i - x_i) + (p_i - x_i) - 1 + X_k^{(A)} \\ &\leq C_k^{(A)} + C_k^{(A)} - 1 + X_k^{(A)} \leq 2 \cdot C_k^{(A)} + X_k^{(A)}. \end{aligned}$$

Estimation

Summing up costs

$$\begin{aligned} \sum_k G_k^{(M)} = \sum_k C_k^{(M)} \leq \sum_k a_k^{(M)} &\leq \sum_k 2 \cdot C_k^{(A)} + X_k^{(A)} \\ &\leq 2 \cdot \sum_k C_k^{(A)} + X_k^{(A)} \\ &= 2 \cdot \sum_k G_k^{(A)} \end{aligned}$$

In the worst case MTF requires at most twice as many operations as the optimal strategy.

13.3 Appendix

Mathematik zur Skipliste

[k-Level Skiplist Math]

Let the number of data points n_0 and number levels $k > 0$ be given and let n_l be the numbers of elements skipped per level l, $n_k = 1$. Maximum number of total steps in the skip list:

$$f(\vec{n}) = \frac{n_0}{n_1} + \frac{n_1}{n_2} + \dots \frac{n_{k-1}}{n_k}$$

Minimize f for $(n_1, \dots, n_{k-1})$: $\frac{\partial f(\vec{n})}{\partial n_t} = 0$ for all $0 < t < k$,

$\frac{\partial f(\vec{n})}{\partial n_t} = -\frac{n_{t-1}}{{n_t}^2} + \frac{1}{n_{t+1}} = 0 \Rightarrow n_{t+1} = \frac{n_t^2}{n_{t-1}}$ and $\frac{n_{t+1}}{n_t} = \frac{n_t}{n_{t-1}}$.

[k-Level Skiplist Math]

Previous slide $\Rightarrow \frac{n_t}{n_0} = \frac{n_t}{n_{t-1}} \frac{n_{t-1}}{n_{t-2}} \dots \frac{n_1}{n_0} = \left(\frac{n_1}{n_0}\right)^t$

Particularly $1 = n_k = \frac{n_1^k}{n_0^{k-1}} \Rightarrow n_1 = \sqrt[k]{n_0^{k-1}}$

Thus $n_{k-1} = \frac{n_0}{n_1} = \sqrt[k]{\frac{n_0^k}{n_0^{k-1}}} = \sqrt[k]{n_0}$.

Maximum number of total steps in the skip list: $f(\vec{n}) = k \cdot (\sqrt[k]{n_0})$

Assume $n_0 = 2^k$, then $\frac{n_l}{n_{l+1}} = 2$ for all $0 \le l < k$ (skiplist halves data in each step) and $f(n) = k \cdot 2 = 2 \log_2 n \in \Theta(\log n)$.