11. Elementare Datenstrukturen

Abstrakte Datentypen Stapel, Warteschlange, Implementationsvarianten der verketteten Liste, amortisierte Analyse [Ottman/Widmayer, Kap. 1.5.1-1.5.2, Cormen et al, Kap. 10.1.-10.2,17.1-17.3]

Abstrakte Datentypen

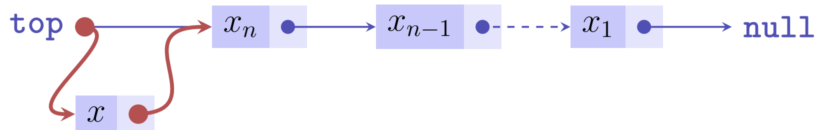
Wir erinnern uns[14] (Vorlesung Informatik I)

Ein *Stack* ist ein abstrakter Datentyp (ADT) mit Operationen

- `push`(x, S): Legt Element x auf den Stapel S.
- `pop`(S): Entfernt und liefert oberstes Element von S, oder `null`.
- `top`(S): Liefert oberstes Element von S, oder `null`.
- `isEmpty`(S): Liefert `true` wenn Stack leer, sonst `false`.
- `emptyStack`$()$: Liefert einen leeren Stack.

[14] hoffentlich

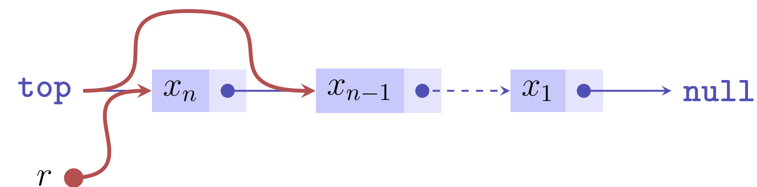
Implementation Push



`push`(x, S):

1. Erzeuge neues Listenelement mit x und Zeiger auf den Wert von `top`.
2. Setze `top` auf den Knotem mit x.

Implementation Pop



`pop`(S):

1. Ist `top`=`null`, dann gib `null` zurück
2. Andernfalls merke Zeiger p von `top` in r.
3. Setze `top` auf $p.next$ und gib r zurück

Analyse

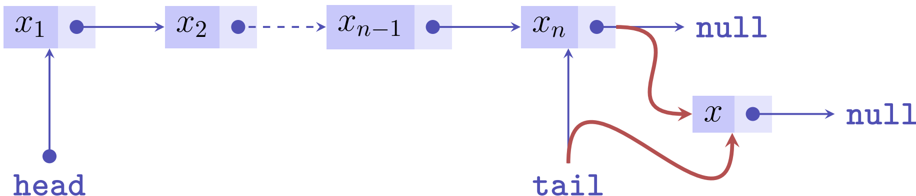
Jede der Operationen `push`, `pop`, `top` und `isEmpty` auf dem Stack ist in $\mathcal{O}(1)$ Schritten ausführbar.

Queue (Schlange / Warteschlange / Fifo)

Queue ist ein ADT mit folgenden Operationen:

- `enqueue`(x, Q): fügt x am Ende der Schlange an.
- `dequeue`(Q): entfernt x vom Beginn der Schlange und gibt x zurück (`null` sonst.)
- `head`(Q): liefert das Objekt am Beginn der Schlage zurück (`null` sonst.)
- `isEmpty`(Q): liefert `true` wenn Queue leer, sonst `false`.
- `emptyQueue`$()$: liefert leere Queue zurück.

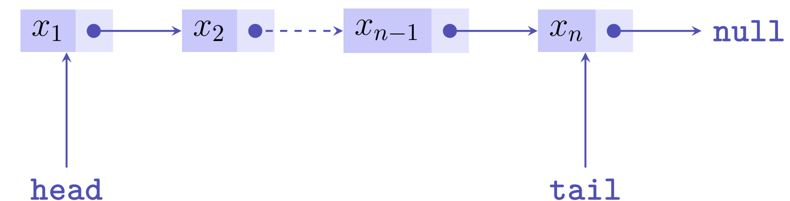
Implementation Queue



`enqueue`(x, S):

1. Erzeuge neues Listenelement mit x und Zeiger auf `null`.
2. Wenn `tail` $\neq$ `null` , setze `tail.next` auf den Knoten mit x.
3. Setze `tail` auf den Knoten mit x.
4. Ist `head` $=$ `null`, dann setze `head` auf `tail`.

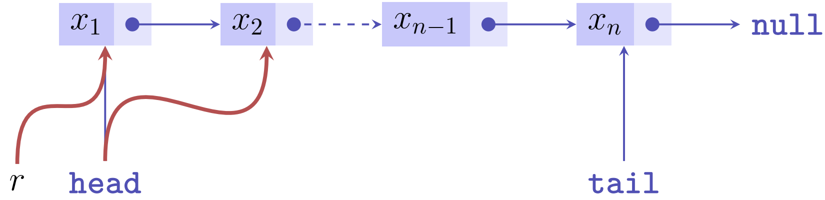
Invarianten!



Mit dieser Implementation gilt

- entweder `head` $=$ `tail` $=$ `null`,
- oder `head` $=$ `tail` $\neq$ `null` und `head.next` $=$ `null`
- oder `head` $\neq$ `null` und `tail` $\neq$ `null` und `head` $\neq$ `tail` und `head.next` $\neq$ `null`.

Implementation Queue



dequeue(S):

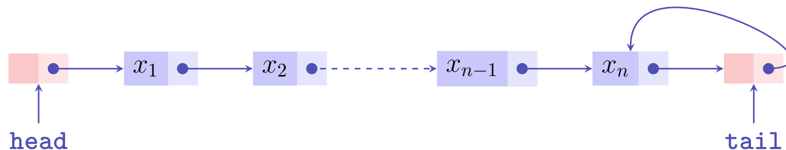
1. Merke Zeiger von head in r. Wenn $r =$ null, gib r zurück.
2. Setze den Zeiger von head auf head.next.
3. Ist nun head $=$ null, dann setze tail auf null.
4. Gib den Wert von r zurück.

Analyse

Jede der Operationen enqueue, dequeue, head und isEmpty auf der Queue ist in $\mathcal{O}(1)$ Schritten ausführbar.

Implementationsvarianten verketteter Listen

Liste mit Dummy-Elementen (Sentinels).

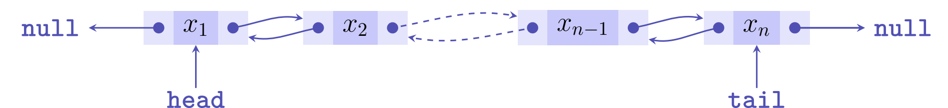


Vorteil: Weniger Spezialfälle!

Variante davon: genauso, dabei Zeiger auf ein Element immer einfach indirekt gespeichert. (Bsp: Zeiger auf x_3 zeigt auf x_2.)

Implementationsvarianten verketteter Listen

Doppelt verkettete Liste



Übersicht

	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) = Einfach verkettet
(B) = Einfach verkettet, mit Dummyelement am Anfang und Ende
(C) = Einfach verkettet, mit einfach indirekter Elementaddressierung
(D) = Doppelt verkettet

Prioritätswarteschlange (Priority Queue)

Priority Queue = Warteschlange mit Prioritäten.

Operationen

- `insert(x,p,Q)`: Füge Objekt x mit Priorität p ein.
- `extractMax(Q)`: Entferne Objekt x mit höchster Priorität und liefere es.

Implementation Prioritätswarteschlage

Mit einem Max-Heap!

Also

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

Multistack

Multistack unterstützt neben den oben genannten Stackoperationen noch

`multipop(s,S)`: Entferne die $\min(\text{size}(S), k)$ zuletzt eingefügten Objekte und liefere diese zurück.

Implementation wie beim Stack. Laufzeit von `multipop` ist $\mathcal{O}(k)$.

Akademische Frage

Führen wir auf einem Stack mit n Elementen n mal `multipop(k,S)` aus, kostet das dann $\mathcal{O}(n^2)$?

Sicher richtig, denn jeder `multipop` kann Zeit $\mathcal{O}(n)$ haben.

Wie bekommen wir eine schärfere Abschätzung?

Idee (Accounting)

Wir führen ein Kostenmodell ein:

- Aufruf von `push`: kostet 1 CHF und zusätzlich 1 CHF kommt aufs Bankkonto
- Aufruf von `pop`: kostet 1 CHF, wird durch Rückzahlung vom Bankkonto beglichen.

Kontostand wird niemals negativ. Also: maximale Kosten: Anzahl der `push` Operationen mal zwei.

Formalisierung

Bezeichne t_i die realen Kosten der Operation i. Potentialfunktion $\Phi_i \geq 0$ für den "Kontostand" nach i Operationen. $\Phi_i \geq \Phi_0 \; \forall i$.

Amortisierte Kosten der i-ten Operation:

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

Es gilt

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

Ziel: Suche Potentialfunktion, die teure Operationen ausgleicht.

Beispiel Stack

Potentialfunktion Φ_i = Anzahl Elemente auf dem Stack.

- $\texttt{push}(x, S)$: Reale Kosten $t_i = 1$. $\Phi_i - \Phi_{i-1} = 1$. Amortisierte Kosten $a_i = 2$.
- $\texttt{pop}(S)$: Reale Kosten $t_i = 1$. $\Phi_i - \Phi_{i-1} = -1$. Amortisierte Kosten $a_i = 0$.
- $\texttt{multipop}(k, S)$: Reale Kosten $t_i = k$. $\Phi_i - \Phi_{i-1} = -k$. Amortisierte Kosten $a_i = 0$.

Alle Operationen haben *konstante amortisierte Kosten*! Im Durchschnitt hat also Multipop konstanten Zeitbedarf. [15]

[15] Achtung: es geht nicht um den probabilistischen Mittelwert sondern den (worst-case) Durchschnitt der Kosten.

Beispiel binärer Zähler

Binärer Zähler mit k bits. Im schlimmsten Fall für jede Zähloperation maximal k Bitflips. Also $\mathcal{O}(n \cdot k)$ Bitflips für Zählen von 1 bis n. Geht das besser?

Reale Kosten t_i = Anzahl Bitwechsel von 0 nach 1 plus Anzahl Bitwechsel von 1 nach 0.

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

$$\Rightarrow t_i = l + 1$$

Beispiel binärer Zähler

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

Potentialfunktion Φ_i: Anzahl der 1-Bits von x_i.

$$\Rightarrow \Phi_i - \Phi_{i-1} = 1 - l,$$

$$\Rightarrow a_i = t_i + \Phi_i - \Phi_{i-1} = l + 1 + (1 - l) = 2.$$

Amortisiert konstante Kosten für eine Zähloperation. ☺

12. Wörterbücher

Wörterbuch, Selbstandornung, Implementation Wörterbuch mit Array / Liste / Skipliste. [Ottman/Widmayer, Kap. 3.3,1.7, Cormen et al, Kap. Problem 17-5]

Wörterbuch (Dictionary)

ADT zur Verwaltung von Schlüsseln aus $\mathcal{K}$ mit Operationen

- `insert`(k, D): Hinzufügen von $k \in \mathcal{K}$ in Wörterbuch D. Bereits vorhanden $\Rightarrow$ Fehlermeldung.
- `delete`(k, D): Löschen von k aus dem Wörterbuch D. Nicht vorhanden $\Rightarrow$ Fehlermeldung.
- `search`(k, D): Liefert `true` wenn $k \in D$, sonst `false`.

Idee

Implementiere Wörterbuch als sortiertes Array.

Anzahl Elementaroperationen im schlechtesten Fall

Suchen	$\mathcal{O}(\log n)$
Einfügen	$\mathcal{O}(n)$
Löschen	$\mathcal{O}(n)$

Andere Idee

Implementiere Wörterbuch als verkettete Liste

Anzahl Elementaroperationen im schlechtesten Fall

Suchen	$\mathcal{O}(n)$
Einfügen	$\mathcal{O}(1)$[16]
Löschen	$\mathcal{O}(n)$

[16] Unter der Voraussetzung, dass wir die Existenz nicht prüfen wollen.

Selbstanordnung

Problematisch bei der Verwendung verketteter Listen: lineare Suchzeit

Idee: Versuche, die Listenelemente so anzuordnen, dass Zugriffe über die Zeit hinweg schneller möglich sind

Zum Beispiel

- Transpose: Bei jedem Zugriff auf einen Schlüssel wird dieser um eine Position nach vorne bewegt.
- Move-to-Front (MTF): Bei jedem Zugriff auf einen Schlüssel wird dieser ganz nach vorne bewegt.

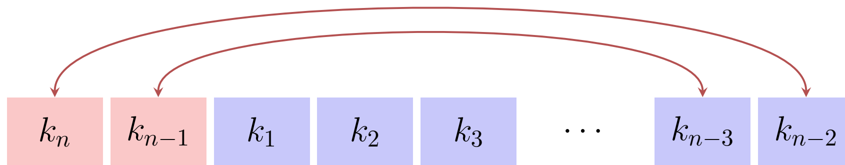
Transpose

Transpose:

Worst case: n Wechselnde Zugriffe auf k_{n-1} und k_n. Laufzeit: $\Theta(n^2)$

Move-to-Front

Move-to-Front:



n Wechselnde Zugriffe auf k_{n-1} und k_n. Laufzeit: $\Theta(n)$

Man kann auch hier Folge mit quadratischer Laufzeit angeben, z.B. immer das letzte Element. Aber dafür ist keine offensichtliche Strategie bekannt, die viel besser sein könnte als MTF.

Analyse

Vergleichen MTF mit dem bestmöglichen Konkurrenten (Algorithmus) A. Wie viel besser kann A sein?

Annahmen:

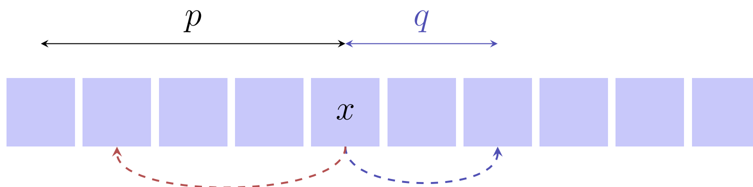
- MTF und A dürfen jeweils nur das zugegriffene Element x verschieben.
- MTF und A starten mit derselben Liste.

M_k und A_k bezeichnen die Liste nach dem k-ten Schritt. $M_0 = A_0$.

Analyse

Kosten:

- Zugriff auf x: Position p von x in der Liste.
- Keine weiteren Kosten, wenn x vor p verschoben wird.
- Weitere Kosten q für jedes Element, das x von p aus nach hinten verschoben wird.



Amortisierte Analyse

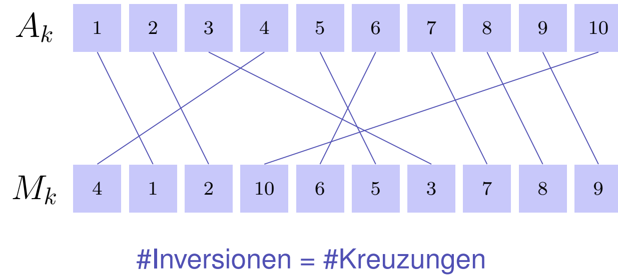
Sei eine beliebige Folge von Suchanfragen gegeben und seien $G_k^{(M)}$ und $G_k^{(A)}$ jeweils die Kosten im Schritt k für Move-to-Front und A. Suchen Abschätzung für $\sum_k G_k^{(M)}$ im Vergleich zu $\sum_k G_k^{(A)}$.

$\Rightarrow$ Amortisierte Analyse mit Potentialfunktion Φ.

Potentialfunktion

Potentialfunktion Φ = Anzahl der Inversionen von A gegen MTF.

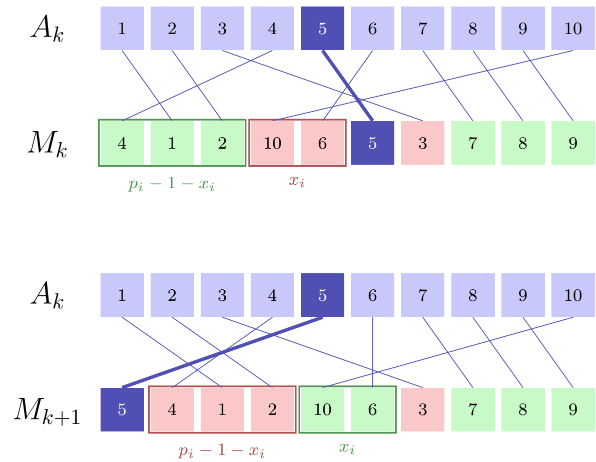
Inversion = Paar x, y so dass für die Positionen von x und y
$\left(p^{(A)}(x) < p^{(A)}(y)\right) \neq \left(p^{(M)}(x) < p^{(M)}(y)\right)$



#Inversionen = #Kreuzungen

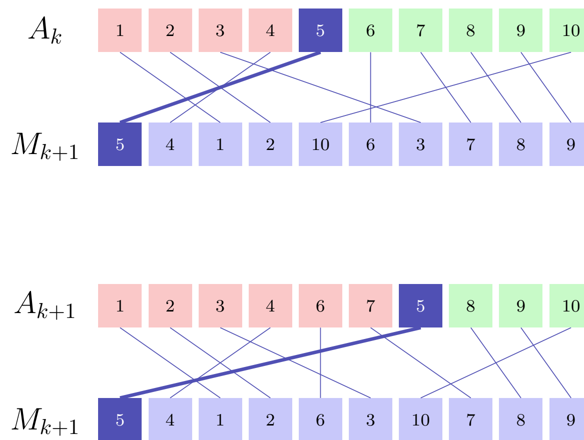
Abschätzung der Potentialfunktion: MTF

- Element i an Position $p_i := p^{(M)}(i)$.
- Zugriffskosten $C_k^{(M)} = p_i$.
- x_i: Anzahl Elemente, die in M vor p_i und in A nach i stehen.
- MTF löst x_i Inversionen auf.
- $p_i - x_i - 1$: Anzahl Elemente, die in M vor p_i und in A vor i stehen.
- MTF erzeugt $p_i - 1 - x_i$ Inversionen.



Abschätzung der Potentialfunktion: A

- Element i an Position $p^{(A)}(i)$.
- $X_k^{(A)}$: Anzahl Verschiebungen nach hinten (sonst 0).
- Zugriffskosten für i: $C_k^{(A)} = p^{(A)}(i) \geq p^{(M)}(i) - x_i$.
- A erhöht die Anzahl Inversionen höchstens um $X_k^{(A)}$.



Abschätzung

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

Amortisierte Kosten von MTF im k-ten Schritt:

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

Abschätzung

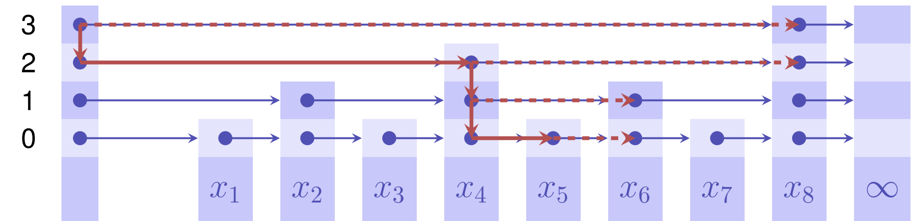
Kosten Summiert

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

MTF führt im schlechtesten Fall höchstens doppelt so viele Operationen aus wie eine optimale Strategie.

Coole Idee: Skiplisten

Perfekte Skipliste



$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

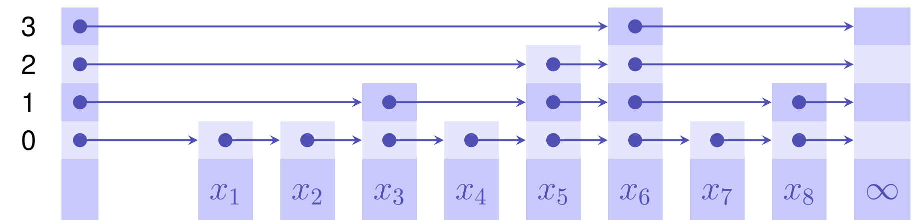
Beispiel: Suche nach einem Schlüssel x mit $x_5 < x < x_6$.

Analyse Perfekte Skipliste (schlechtester Fall)

Suchen in $\mathcal{O}(\log n)$. Einfügen in $\mathcal{O}(n)$.

Randomisierte Skipliste

Idee: Füge jeweils einen Knoten mit zufälliger Höhe H ein, wobei $\mathbb{P}(H = i) = \frac{1}{2^{i+1}}$.



Analyse Randomisierte Skipliste

Theorem

Die Anzahl an erwarteten Elementaroperationen für Suchen, Einfügen und Löschen eines Elements in einer randomisierten Skipliste ist $\mathcal{O}(\log n)$.

Der längliche Beweis, welcher im Rahmen dieser Vorlesung nicht geführt wird, betrachtet die Länge eines Weges von einem gesuchten Knoten zurück zum Startpunkt im höchsten Level.