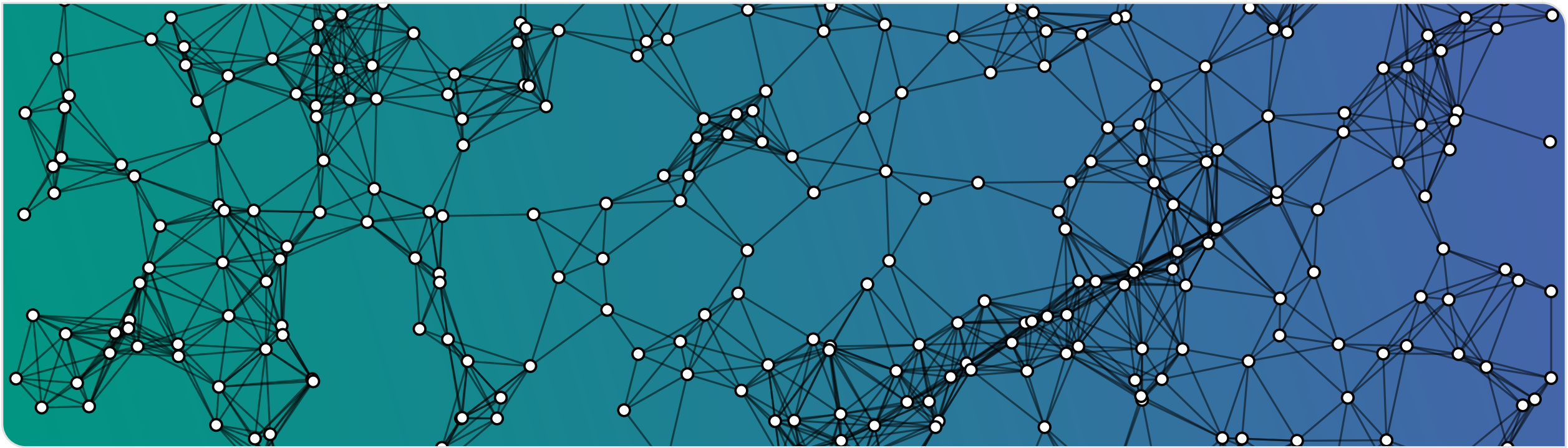
Karlsruher Institut für Technologie

Algorithmen 1

Übung 2
Amortisierte Analyse, Sortieren



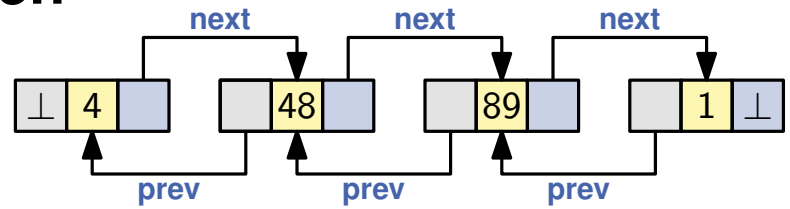
Datenstrukturen bisher in Algo 1

Array (dynamisch / unbeschränkt)

4	48	89	1	0	9	13	7	32	76	17	5

- *wahlfreier* Zugriff (*random access*)
  - Schreiben / Lesen von Wert an Stelle i in $O(1)$ Zeit, z.B. $A[i] := 5$
- **pushBack**() in $O(1)$ Zeit (amortisiert)

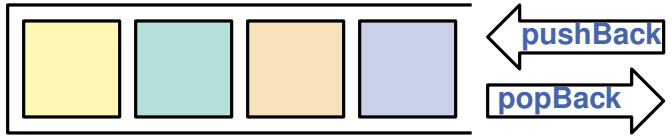
Listen



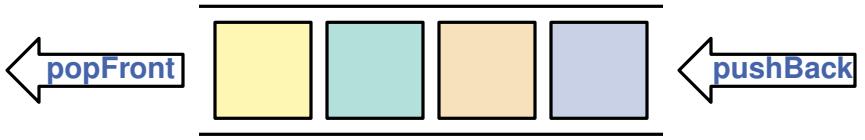
- Einfügen, Löschen, etc. in $O(1)$ Zeit
- **kein** Wahlfreier Zugriff

Weitere Datenstrukturen

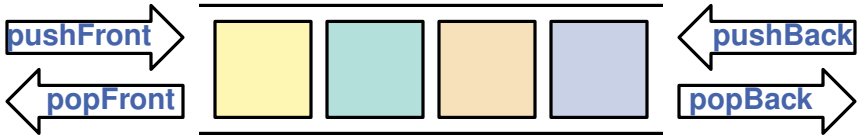
- Stack



- Queue



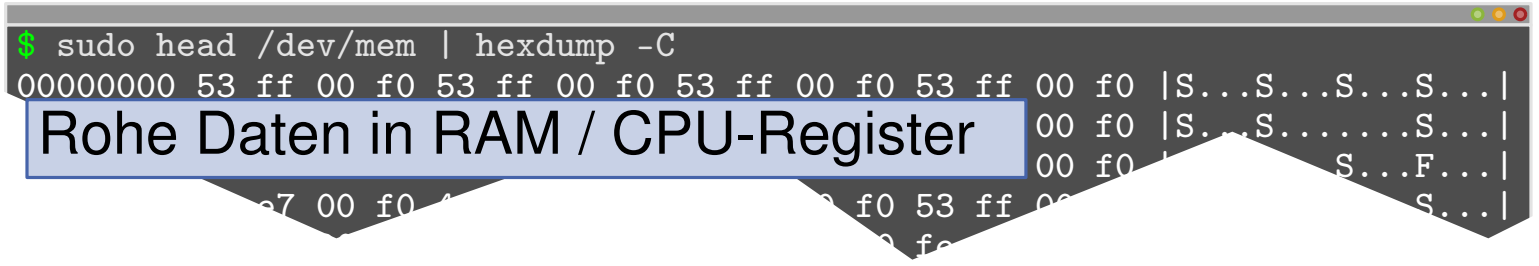
- Double-ended queue

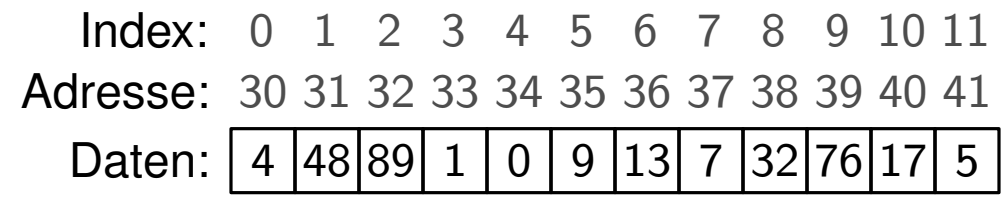


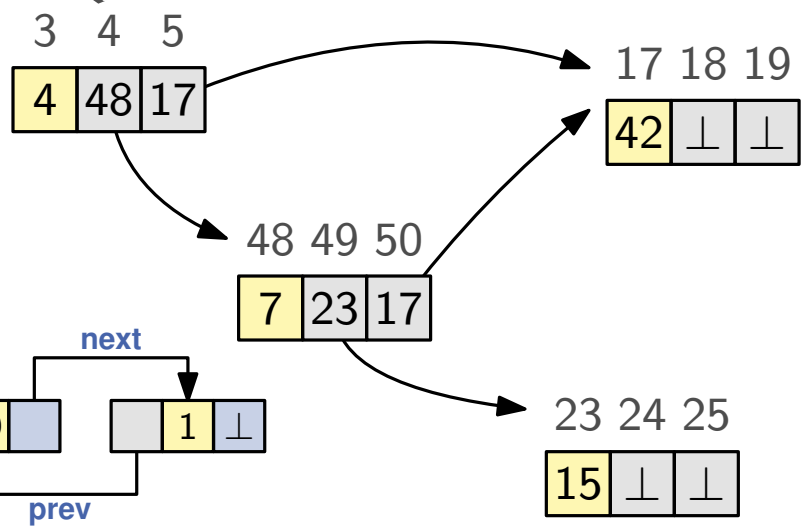
Motivation: Datenstrukturen

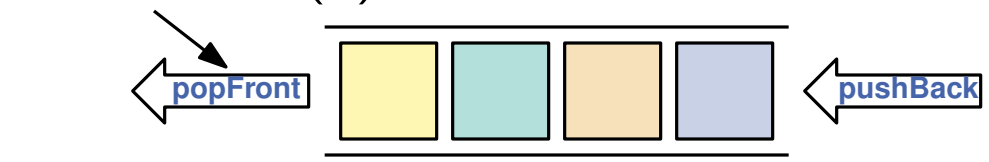
Was sind Datenstrukturen und wofür sind sie gut?

- Abstraktion des Speichers
  - Rad nicht neu erfinden
  - keine unnötigen Details bei Arbeit auf hohem Abstraktionslevel









Motivation: Amortisierte Analyse

- Laufzeit von Operationen auf DS nicht immer gleich
  - z.B.: Operation meist günstig, seltener teurer

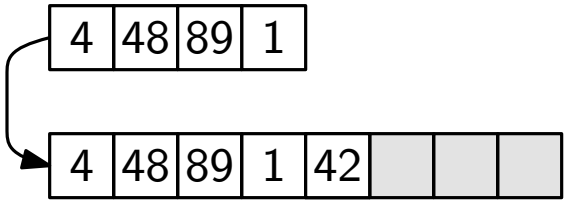
Idee

- **amortisierte Kosten**: durchschnittliche Kosten pro Operation in einer Folge von Operationen

Verschiedene Techniken zur Analyse

- Aggregation
- Charging
- Konto
- **Potential**

Beispiel: unbeschränktes Array



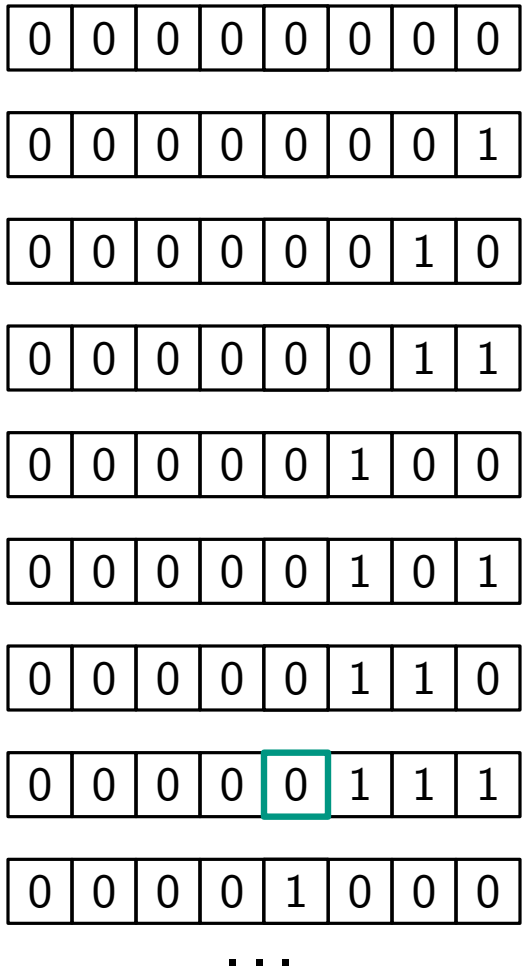
Beispiel: Binärzähler

Idee

- Array A verwaltet Bits
- Funktion **increment()** erhöht Zähler

Algorithmus

- suche Stelle i mit rechtester 0
- setze Stelle i auf 1
- setze Stellen rechts von i auf 0



Beispiel: Binärzähler

Idee

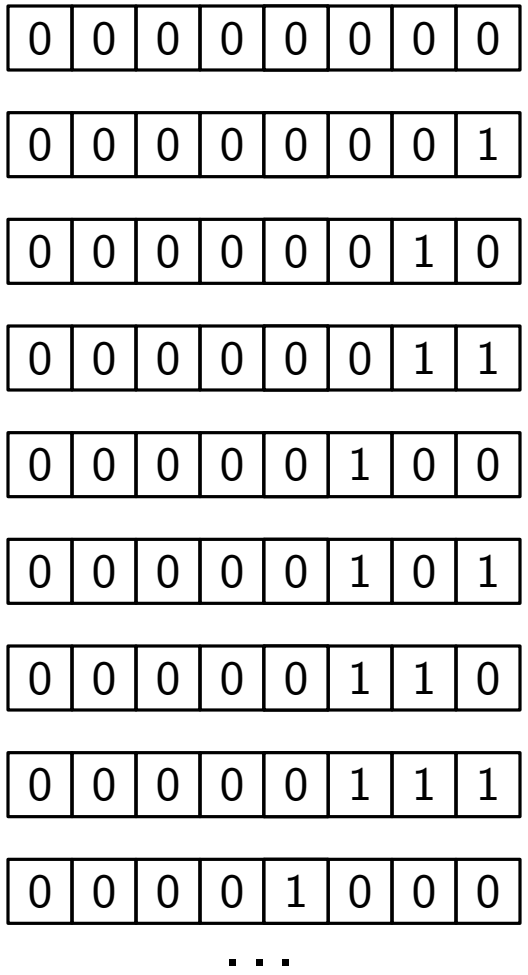
- Array A verwaltet Bits
- Funktion **increment()** erhöht Zähler

Algorithmus in Pseudocode

increment()

```
i := 0
while A[i] = 1 do
    A[i] := 0
    i := i + 1
A[i] := 1
```

Laufzeit?



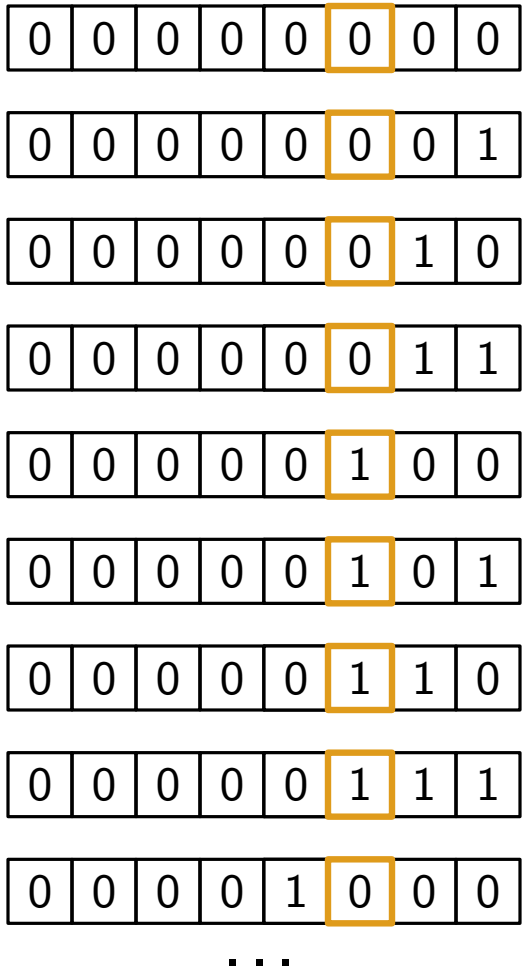
Analyse: Aggregatmethode

Idee

- berechne Gesamtkosten
- teile durch Anzahl von Operationen

Beobachtung

- nicht jedes Bit flipt bei jeder Operation
- $A[0]$ flipt jedes mal, $A[1]$ jedes zweite, etc.
- $A[i]$ flipt bei jedem 2^i-ten Inkrement
- Bei n Aufrufen: $A[i]$ flipt $\lfloor \frac{n}{2^i} \rfloor$ mal



Analyse: Aggregatmethode

Beobachtung

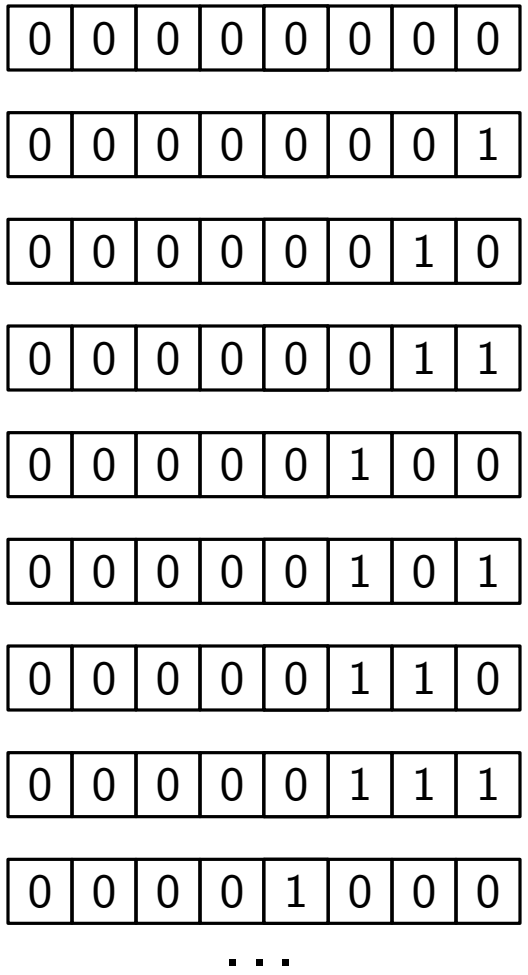
- nicht jedes Bit flipt bei jeder Operation
- $A[0]$ flipt jedes mal, $A[1]$ jedes zweite, etc.
- $A[i]$ flipt bei jedem 2^i-ten Inkrement
- Bei n Aufrufen: $A[i]$ flipt $\lfloor \frac{n}{2^i} \rfloor$ mal

In Summe ergibt sich für k bits:

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

D.h.: pro Operation nur 2 Bit-Flips

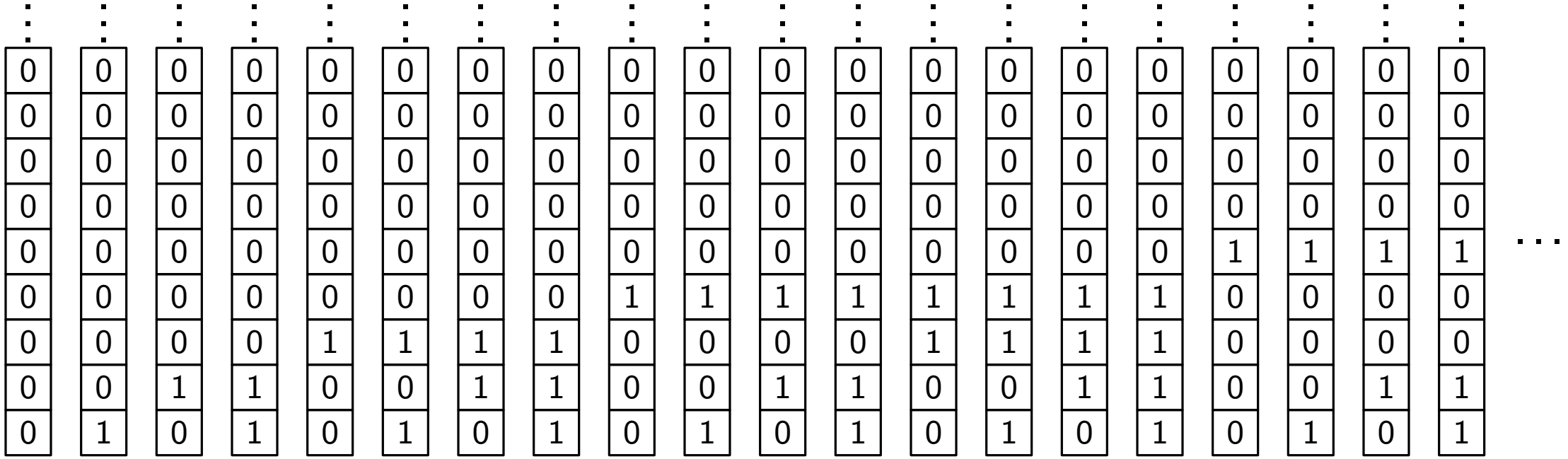
Somit: **amortisierte Kosten** in $O(1)$



Analyse mittels Charging

Idee

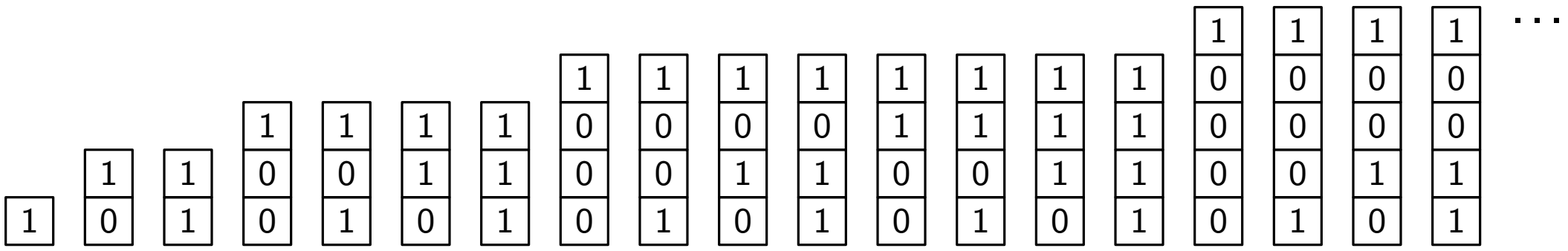
- teure Operationen legen Kosten um auf günstige Operationen



Analyse mittels Charging

Idee

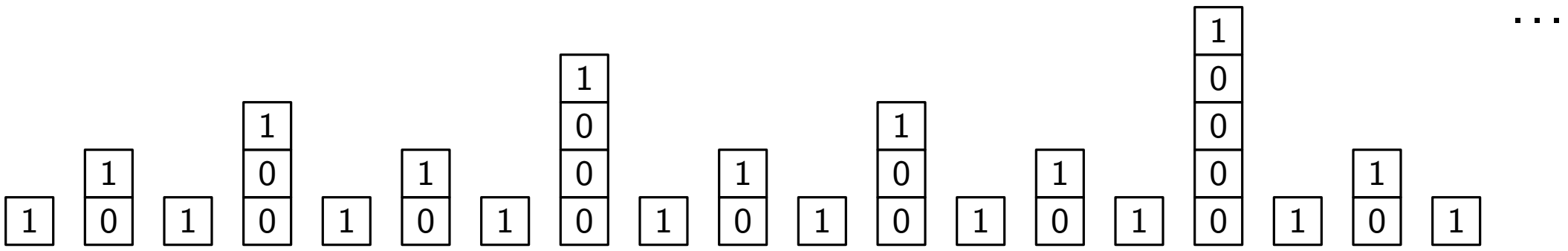
- teure Operationen legen Kosten um auf günstige Operationen



Analyse mittels Charging

Idee

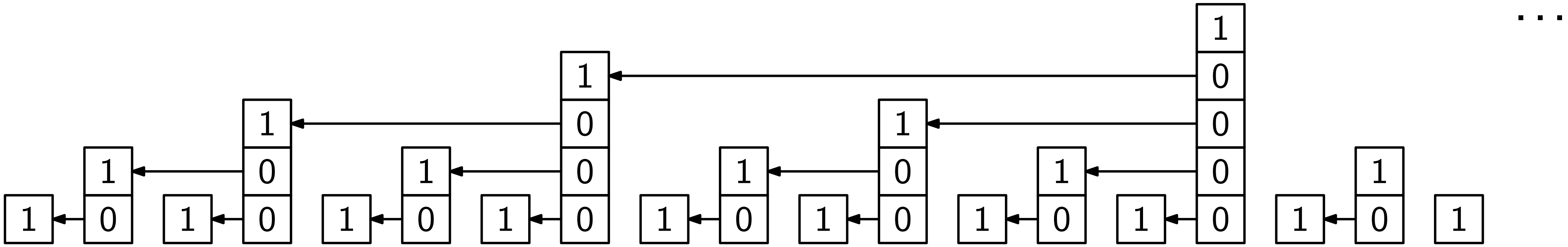
- teure Operationen legen Kosten um auf günstige Operationen



Analyse mittels Charging

Idee

- teure Operationen legen Kosten um auf günstige Operationen



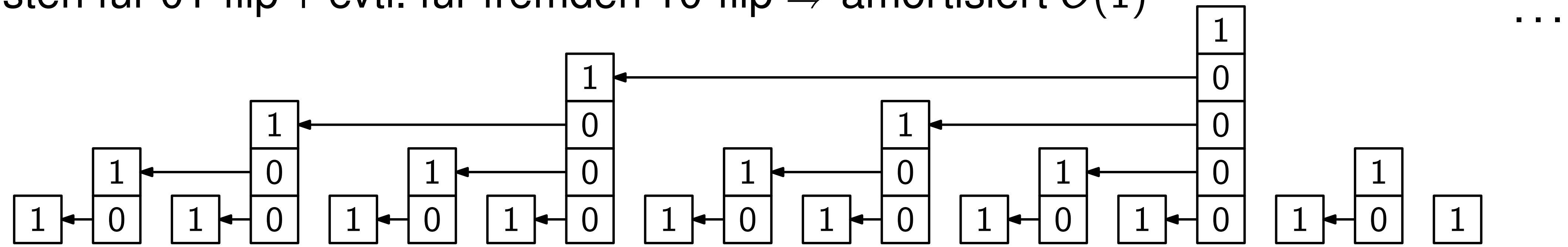
Analyse mittels Charging

Idee

- teure Operationen legen Kosten um auf günstige Operationen

Vorgehen hier:

- lege Kosten von 10-flip auf letzte Operation welche zuvor 01-flip ausgeführt hat
  - das geht für jeden 10-flip
- pro Operation übrig:
  - Kosten für 01-flip + evtl. für fremden 10-flip $\Rightarrow$ amortisiert $O(1)$



Analyse: Potentialmethode

Idee

- definiere Potentialfunktion $\Phi(D_i)$
  - Maß für Unaufgeräumtheit von D zum Zeitpunkt i
- definiere amortisierte Kosten $\hat{c}_i = c_i + \underbrace{\Phi(D_i) - \Phi(D_{i-1})}_{\text{Anstieg des Potenzials}}$

tatsächliche Kosten

Analyse: Potentialmethode

Idee

- definiere Potentialfunktion $\Phi(D_i)$
  - Maß für Unaufgeräumtheit von D zum Zeitpunkt i
- definiere amortisierte Kosten $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

Definition sinnvoll?

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n} (c_i) + \sum_{i=1}^{n} (\Phi(D_i) - \Phi(D_{i-1}))$$

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

Falls $\Phi(D_n) \geq \Phi(D_0)$: amortisierte Kosten obere Schranke für tatsächliche Kosten

$\Rightarrow$ fordere $\Phi(D_i) \geq \Phi(D_0)$ f.a. $i > 0$ oder: $\Phi(D_0) = 0$ und $\Phi(D_i) \geq 0$ f.a. $i > 0$

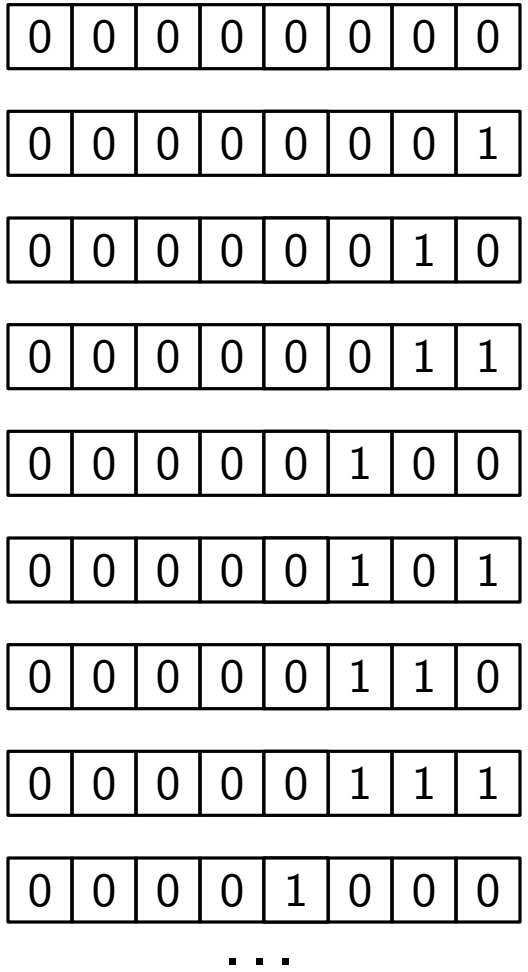
Analyse: Potentialmethode

Potentialfunktion für Binärzähler:

- def. $\Phi(D_i) = \sum_{j=0}^{k-1} A[j]$ (Anzahl 1-bits zum Zeitpunkt i)

Dann ergibt sich:

- $\Phi(D_0) = 0$, $\Phi(D_i) \geq 0$ f.a. $i > 0$ *gültige Pot.fun.* :)
- tatsächliche Kosten: $c_i = \#01\text{-flips} + \#10\text{-flips}$
- Potentialänderung: $\#01\text{-flips} - \#10\text{-flips}$
- amort. Kosten: $\hat{c}_i = 2 \cdot \#01\text{-flips} \leq 2$



Definition: amortisierte Kosten $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

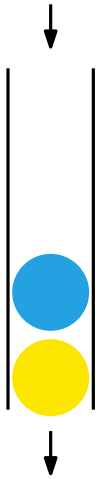
Beispiel 2: Stacks und Queues

Wiederholung

Queue

- **push**(e) $O(1)$
- **pop**() $O(1)$
- **size**() $O(1)$

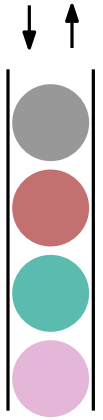
z.B. mittels verketteter Liste



Stack

- **push**(e) $O(1)$
- **pop**() $O(1)$
- **size**() $O(1)$

z.B. mittels verketteter Liste



Frage

Angenommen wir können Stacks (als Blackbox) verwenden.

(Wie) können wir daraus eine Queue bauen?

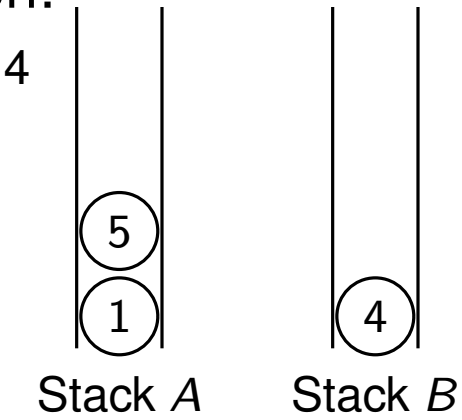
Eine Queue aus Stacks

Algorithmische Umsetzung

- **push**: auf A pushen — $O(1)$
- **pop**: — $O(|A|)$
  - falls B nicht leer: von B poppen — $O(1)$
  - falls B leer: alles von A nach B verschieben — $O(|A|)$

Operationen:

- **push**: 3, 1, 4
- **pop**()
- **pop**()
- **push**: 1,5



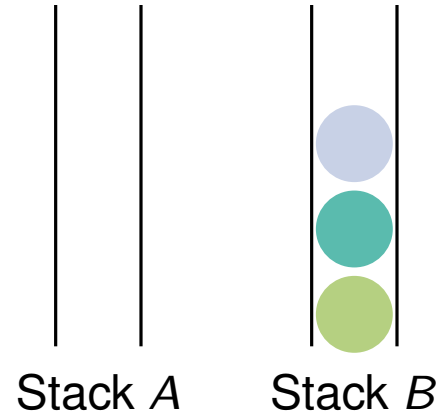
Analyse: Kontomethode

Idee

- günstige Operationen bauen Guthaben auf
- teure Operationen nutzen Guthaben

Umsetzung hier

- Gesamtkosten linear in **push** und **pop** Aufrufen
- Bei **push**: zahle 3
- Bei **pop**: zahle 1
  - Falls B leer: nutze 2 Guthaben für A.**pop()** und B.**push()** (für alle Elemente in A)
  - Falls B voll: B.**pop**() mit Kosten 1
- Konto wird nie negativ
- $3, 1 \in \Theta(1) \Rightarrow$ konstante amortisierte Kosten



Analyse: Potentialmethode

Potentialfunktion

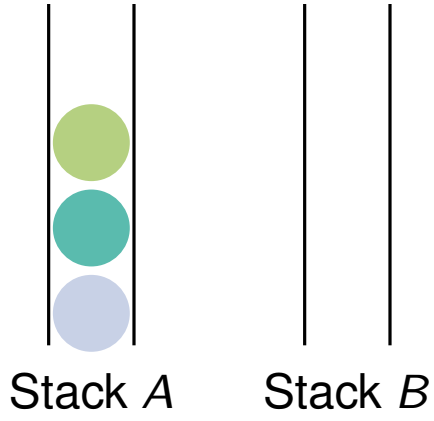
- Definiere $\Phi(D_i) = 2 \cdot |A_i|$ (Anzahl Elemente auf A zum Zeitpunkt i)

Analyse

- $\Phi(D_0) = 0$, $\Phi(D_i) \geq 0$ f.a. $i > 0$ (d.h. gültige Potentialfunktion)
- amortisierte Kosten **push**:
  - $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 3$
- amortisierte Kosten **pop**:
  - $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1$

} amortisierte Kosten in $O(1)$

  - $c_i = 2 \cdot |A_{i-1}| + 1$
  - $\Phi(D_i) - \Phi(D_{i-1}) = -2|A_{i-1}|$



Definition: amortisierte Kosten $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

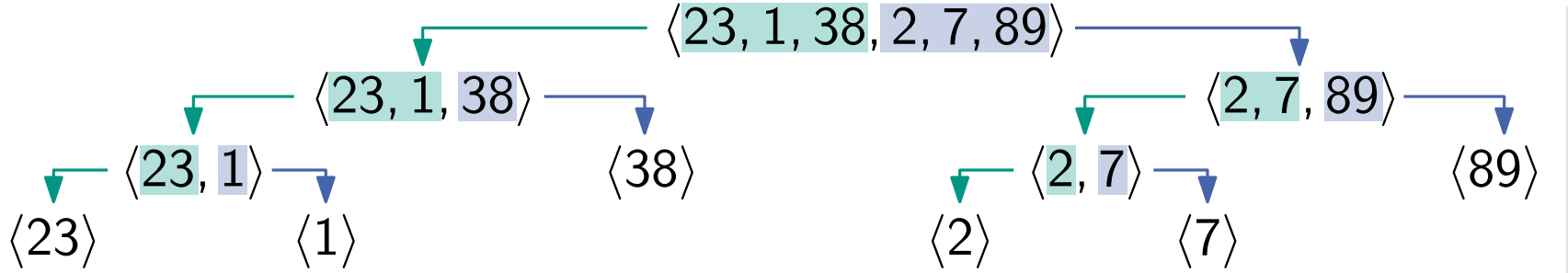
Sortieren

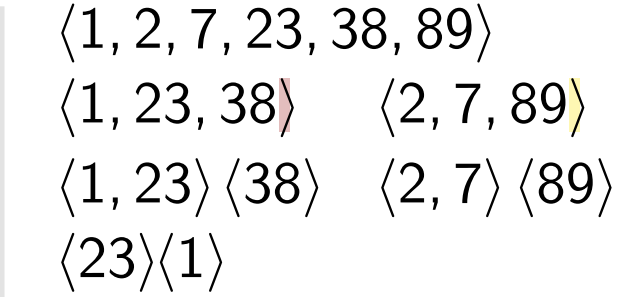
- Gegeben: n Elemente aus einer geordneten Menge (z.B.: Zahlen) $\langle 23, 1, 38, 2, 7, 89 \rangle$
- Gesucht: sortierte Folge dieser Elemente $\langle 1, 2, 7, 23, 38, 89 \rangle$

Algorithmen

- InsertionSort ($\Theta(n^2)$)
  - Position jeden Elements durch binäre Suche im bisher sortierten Teil ermitteln

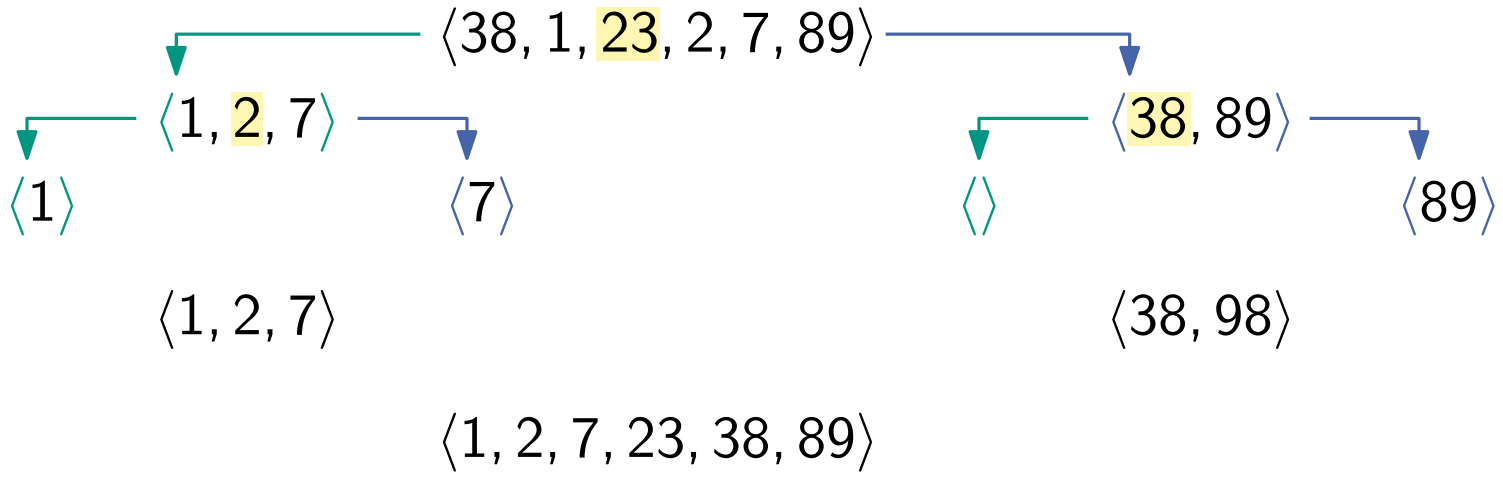
    $\langle 1, 2, 7, 23, 38, 89 \rangle$
- MergeSort ($\Theta(n \log(n))$)
  - Teile ein Array rekursiv in zwei Hälften, welche wieder zusammengeführt werden





Sortieren

- QuickSort: Laufzeit erwartet $O(n \log n)$
  - Teile ein Array rekursiv anhand eines Pivots in zwei Teile, welche jeweils alle Elemente kleiner bzw. größer dem Pivot enthalten



- Die Wahl des Pivots beeinflusst die Laufzeit
- Bereits gesehen
  - Worst Case $\rightarrow$ Pivot ist immer kleinstes oder größtes Element
  - Mittels zufällig gewähltem Pivot kommt man nah an den Best Case

Warum?

QuickSort – Pivots

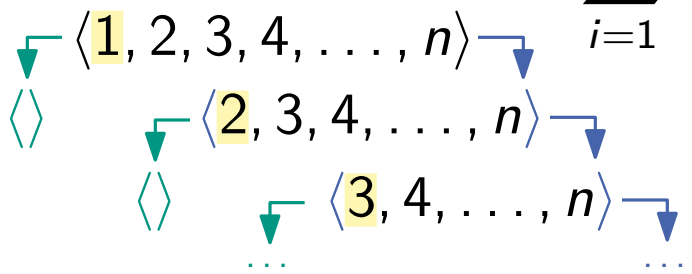
- Worst Case vs. Average Case – Adversary Sichtweise
  - Teile ein Array rekursiv anhand eines Pivots in zwei Teile, welche jeweils alle Elemente kleiner bzw. größer dem Pivot enthalten

$$\langle 38, 1, 23, 2, 7, 89 \rangle$$

- Wenn wir nichts über die Eingabe wissen, kann jeder Eintrag im Array ein gutes oder schlechtes Pivot-Element sein
  - wähle irgendein Element als Pivot $\rightarrow$ Beispiel: Pivot ist immer der erste Eintrag im Array
  - Was kann im schlimmsten Falls passieren?

$$\sum_{i=1}^{n} i = \Theta(n^2)$$

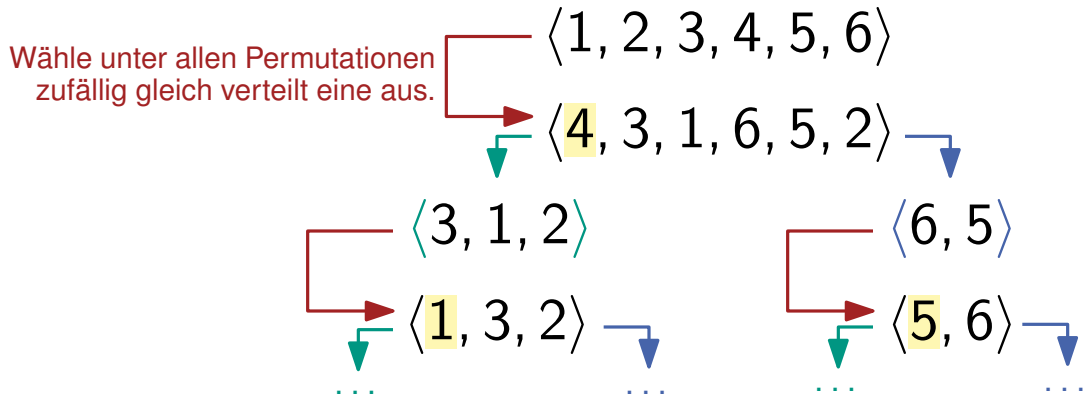
- Auftritt: Adversary
  - kennt den Algorithmus
  - baut eine Instanz die für schlechte Laufzeit sorgt



Adapted from https://scryfall.com/card/mid/129/bloodthirsty-adversary

QuickSort – Pivots

- Wie können wir Quicksort Adversary die Tour vermasseln?
  - ... selbst wenn sie den Algorithmus kennt?
- Unser Algo braucht eine Komponente die sie nicht vorhersehen kann! **Zufall!**
- Idee: Bevor wir das Pivot-Element wählen, wird die Eingabe zufällig gemischt.
  - Die Wahl von Quicksort Adversary wird irrelevant!



- Problem:
  - Extra Aufwand (um sortieren zu können, wird erstmal gemischt)
  - Wird die Laufzeit dadurch wirklich besser?

Adapted from https://scryfall.com/card/afr/79/tricksters-talisman

QuickSort – Pivots

- Nach dem zufälligen Mischen ? des Arrays wählen wir das erste Element als Pivot.
- Wie kommt die zufällige Permutation zustande?

$\langle a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9 \rangle \longrightarrow \langle a_5, a_1, a_7, a_6, a_3, a_9, a_4, a_2, a_0, a_8 \rangle$

  - Für Index 0: Wähle aus n Elementen gleich verteilt eins aus
  - Für Index 1: Wähle aus $n - 1$ Elementen gleich verteilt eins aus

  ...

  - Für Index i: Wähle aus $n - i$ Elementen gleich verteilt eins aus
- Welches Element landet beim Mischen an erster Stelle?
  - Das Pivot-Element wird nur von der ersten zufälligen Wahl beeinflusst
  - Die Sortierung der restlichen Elemente ist irrelevant
- Demzufolge kann als Pivot auch einfach direkt ein *zufälliges* Element gewählt werden

Bei einem beliebigen aber **festen Pivot** kann Quicksort Adversary eine schlechte Laufzeit erzwingen.
Bei einem **zufällig gewählten Pivot** kann Quicksort Adversary die Laufzeit nicht beeinflussen.

QuickSort – Pivots

- Mit zufälliger Wahl des Pivots ist es unwahrscheinlicher, dass wir den Worst Case treffen.
- Welche Laufzeit können wir erwarten? $O(n \log(n))$ (siehe VL)
- Jetzt: Alternativer Beweis
  - Laufzeit ist bestimmt durch die Anzahl der Vergleiche X
  - Betrachte die Elemente in sortierter Reihenfolge $\langle a_0, a_1, a_2, \ldots, a_i, \ldots, a_j, \ldots, a_{n-1}, a_n \rangle$
  - Betrachte jedes Paar a_i, a_j von Elementen und frage: Werden a_i und a_j verglichen?
  - Das hängt davon ab, welche Elemente als Pivots gewählt werden! **Zufall!**
  - Indikatorzufallsvariable $X_{ij} = \begin{cases} 1, & \text{wenn } a_i \text{ und } a_j \text{ verglichen werden} \\ 0, & \text{sonst} \end{cases}$

$$\mathbb{E}[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \Pr[X_{ij} = 1]$$

Dank der Linearität von $\mathbb{E}$ können wir die X_{ij} einzeln betrachten, anstatt alle auf einmal!

$$X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij} \rightarrow \mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n} \mathbb{E}\left[\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \Pr[X_{ij} = 1]$$

Linearität von $\mathbb{E}$: $\mathbb{E}[c \cdot Y + d \cdot Z] = c \cdot \mathbb{E}[Y] + d \cdot \mathbb{E}[Z]$

Indikatorvariable Y: $\mathbb{E}[Y] = \cancel{0 \cdot \Pr[Y = 0]} + 1 \cdot \Pr[Y = 1]$

QuickSort – Pivots

- Mit zufälliger Wahl des Pivots ist es unwahrscheinlicher, dass wir den Worst Case treffen.
- Welche Laufzeit können wir erwarten?
- Jetzt: Alternativer Beweis
  - Laufzeit ist bestimmt durch die Anzahl der Vergleiche X

$$\mathbb{E}[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \Pr[X_{ij} = 1]$$

  - Betrachte die Elemente in sortierter Reihenfolge $\langle a_0, a_1, a_2, \ldots, a_i, \ldots, a_j, \ldots, a_{n-1}, a_n \rangle$
  - Betrachte jedes Paar a_i, a_j von Elementen und frage: Werden a_i und a_j verglichen?
  - Das hängt davon ab, welche Elemente als Pivots gewählt werden!
  - Fall 1: Das Pivot ist links von a_i oder rechts von a_j
    - a_i und a_j kommen in das gleiche Teilarray
  - Fall 2: Das Pivot ist zwischen a_i und a_j
    - a_i und a_j kommen in verschiedene Teilarrays
  - Fall 3: Das Pivot ist a_i oder a_j

Dieses Pivot hat keinen Einfluss darauf ob a_i und a_j verglichen werden!

Mit diesem Pivot werden a_i und a_j niemals verglichen! $X_{ij} = 0$

Mit diesem Pivot werden a_i und a_j auf jeden Fall verglichen! $X_{ij} = 1$

QuickSort – Pivots

- Betrachte die Pivots die zufällig gewählt werden
- Pivots außerhalb von $\{a_i, \ldots, a_j\}$ interessieren uns nicht
- Das Schicksal von X_{ij} wird besiegelt wenn das erste Pivot in $\{a_i, \ldots, a_j\}$ fällt
- Dieser Bereich enthält $j - i + 1$ Elemente
- Wie wahrscheinlich ist es, dass $X_{ij} = 1$ gilt, wenn wir schon wissen, dass das Pivot in diesem Bereich ist?

$$\mathbb{E}[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \Pr[X_{ij} = 1]$$

$$\langle a_0, a_1, a_2, \ldots, a_i, \ldots, a_j, \ldots, a_{n-1}, a_n \rangle$$

$$\Pr[X_{ij} = 1] = \frac{2}{j - i + 1}$$

  - Fall 2: Das Pivot ist zwischen a_i und a_j
    - a_i und a_j kommen in verschiedene Teilarrays
  - Fall 3: Das Pivot ist a_i oder a_j

Mit diesem Pivot werden a_i und a_j niemals verglichen! $X_{ij} = 0$

Mit diesem Pivot werden a_i und a_j auf jeden Fall verglichen! $X_{ij} = 1$

QuickSort – Pivots

$$\mathbb{E}[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \Pr[X_{ij} = 1]$$

$$= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} \frac{2}{k}$$

$j = n$	$\rightarrow j-i+1 =$	$n-i+1$
$\cdots$	$\cdots$	$\cdots$
$j = i+2$	$\rightarrow j-i+1 = i+2-i+1 =$	3
$j = i+1$	$\rightarrow j-i+1 = i+1-i+1 =$	2

$$\mathbb{E}[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \Pr[X_{ij} = 1]$$

$$\Pr[X_{ij} = 1] = \frac{2}{j-i+1}$$

QuickSort – Pivots

$$\mathbb{E}[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \Pr[X_{ij} = 1]$$

$$= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

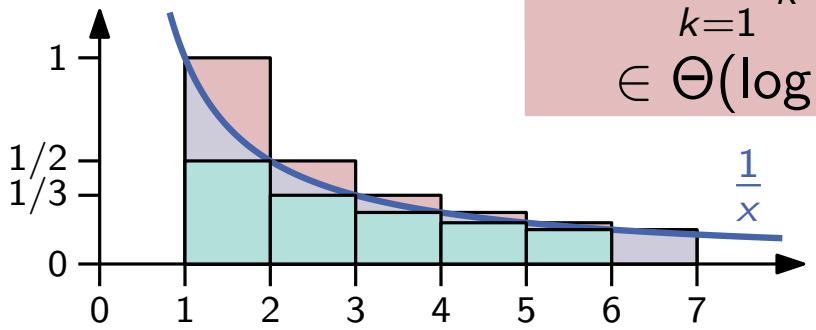
$$= \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} \frac{2}{k}$$

$$\leq \sum_{i=1}^{n} \sum_{k=2}^{n} \frac{2}{k}$$

$$= n \cdot 2 \cdot \sum_{k=2}^{n} \frac{1}{k}$$

Harmonische Summe

- Harmonische Zahl $H_n = \sum_{k=1}^{n} \frac{1}{k} \in \Theta(\log(n))$



$$\int_1^n \frac{1}{x}\,dx \leq \sum_{k=1}^{n} \frac{1}{k} = H_n$$

$$\sum_{k=2}^{n} \frac{1}{k} \leq \int_1^n \frac{1}{x}\,dx = \left[\ln(x)\right]_1^n = \ln(n)$$

$= H_n - 1$

$$\mathbb{E}[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \Pr[X_{ij} = 1]$$

$$\Pr[X_{ij} = 1] = \frac{2}{j-i+1}$$

Theorem
Auf jeder Eingabe der Länge n benötigt QuickSort mit zufälligen Pivots erwartet $\leq 2n\ln(n)$ Vergleiche.