

# Algorithmen 1

## Dynamische Arrays & amortisierte Analyse



# Datenstrukturen

## Was ist das überhaupt?

- Algorithmus: Abfolge von Schritten. Jeder Schritt ...
  - liest (wenige) Daten im Speicher
  - verarbeitet die gelesenen Daten
  - schreibt Daten in den Speicher
- Zugriff auf den Speicher mittels Speicheradressen
- Datenstruktur: Abstraktionsebene über dem Speicher
  - versteckt, wie Daten genau im Speicher abgelegt sind
  - bietet Schnittstelle mit angenehmen Operationen

# Zwei Arten der Datenablage

## Felder (Arrays)

- Menge aufeinanderfolgender Speicherzellen
- Zugriff mit Adresse bzw. Index an beliebiger Stelle in  $\Theta(1)$
- sehr nah an der Hardware
- heute: Komfort-Funktionen

Index:	0	1	2	3	4	5	6	7	8	9	10	11
Adresse:	30	31	32	33	34	35	36	37	38	39	40	41
Daten:	4	48	89	1	0	9	13	7	32	76	17	5

# Zwei Arten der Datenablage

## Felder (Arrays)

- Menge aufeinanderfolgender Speicherzellen
- Zugriff mit Adresse bzw. Index an beliebiger Stelle in  $\Theta(1)$
- sehr nah an der Hardware
- heute: Komfort-Funktionen

## Verzeigerte Strukturen

- viele kleine Stückchen Speicher (Knoten)

Index:	0	1	2	3	4	5	6	7	8	9	10	11
Adresse:	30	31	32	33	34	35	36	37	38	39	40	41
Daten:	4	48	89	1	0	9	13	7	32	76	17	5

3	4	5
4	48	17

17	18	19
42	⊥	⊥

48	49	50
7	23	17

23	24	25
15	⊥	⊥

# Zwei Arten der Datenablage

## Felder (Arrays)

- Menge aufeinanderfolgender Speicherzellen
- Zugriff mit Adresse bzw. Index an beliebiger Stelle in  $\Theta(1)$
- sehr nah an der Hardware
- heute: Komfort-Funktionen

## Verzeigerte Strukturen

- viele kleine Stückchen Speicher (Knoten)
- ein Knoten speichert:
  - Daten, die uns tatsächlich interessieren
  - Speicheradressen anderer Knoten (Zeiger)

Index:	0	1	2	3	4	5	6	7	8	9	10	11
Adresse:	30	31	32	33	34	35	36	37	38	39	40	41
Daten:	4	48	89	1	0	9	13	7	32	76	17	5

3	4	5
4	48	17

17	18	19
42	⊥	⊥

48	49	50
7	23	17

23	24	25
15	⊥	⊥

# Zwei Arten der Datenablage

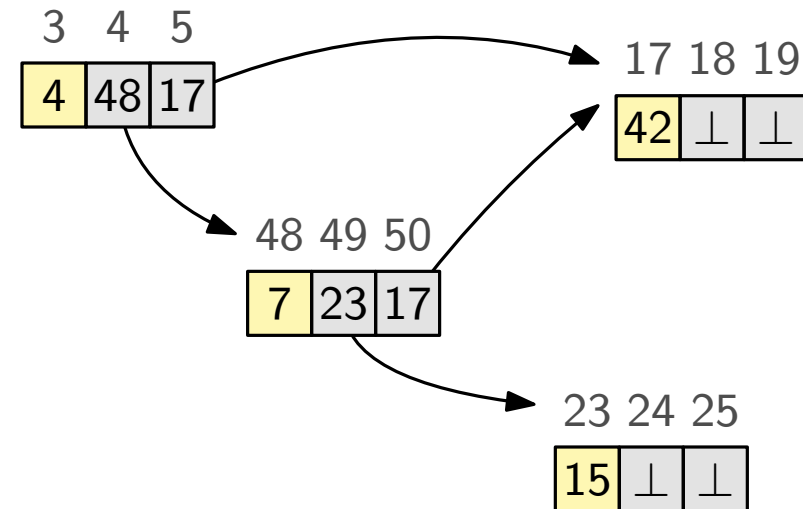
## Felder (Arrays)

- Menge aufeinanderfolgender Speicherzellen
- Zugriff mit Adresse bzw. Index an beliebiger Stelle in  $\Theta(1)$
- sehr nah an der Hardware
- heute: Komfort-Funktionen

## Verzeigerte Strukturen

- viele kleine Stückchen Speicher (Knoten)
- ein Knoten speichert:
  - Daten, die uns tatsächlich interessieren
  - Speicheradressen anderer Knoten (Zeiger)

Index:	0	1	2	3	4	5	6	7	8	9	10	11
Adresse:	30	31	32	33	34	35	36	37	38	39	40	41
Daten:	4	48	89	1	0	9	13	7	32	76	17	5



# Zwei Arten der Datenablage

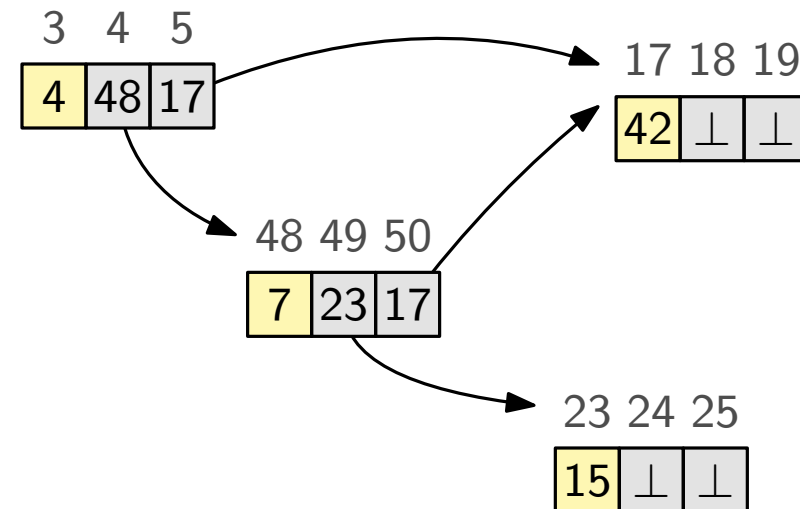
## Felder (Arrays)

- Menge aufeinanderfolgender Speicherzellen
- Zugriff mit Adresse bzw. Index an beliebiger Stelle in  $\Theta(1)$
- sehr nah an der Hardware
- heute: Komfort-Funktionen

## Verzeigerte Strukturen

- viele kleine Stückchen Speicher (Knoten)
- ein Knoten speichert:
  - Daten, die uns tatsächlich interessieren
  - Speicheradressen anderer Knoten (Zeiger)
- Zugriff durch Navigation entlang Zeiger
- abstrahiert stärker von der Hardware
- nächstes Mal: Listen als einfaches Beispiel

Index:	0	1	2	3	4	5	6	7	8	9	10	11
Adresse:	30	31	32	33	34	35	36	37	38	39	40	41
Daten:	4	48	89	1	0	9	13	7	32	76	17	5



# Beschränkte und Unbeschränkte Arrays

## Beschränkte Arrays

- beim Erstellen des Arrays muss ich festlegen, wie viel Speicher ich brauche
- nachträgliches Vergrößern schwierig: nachfolgende Speicherzellen ggf. schon belegt
- Problem: benötigter Speicher ist ggf. durch äußere Faktoren bedingt (Nutzereingabe)

Index:	0	1	2	3	4	5	6	7	8	9	10	11
Adresse:	30	31	32	33	34	35	36	37	38	39	40	41
Daten:	4	48	89	1	0	9	13	7	32	76	17	5



# Beschränkte und Unbeschränkte Arrays

## Beschränkte Arrays

- beim Erstellen des Arrays muss ich festlegen, wie viel Speicher ich brauche
- nachträgliches Vergrößern schwierig: nachfolgende Speicherzellen ggf. schon belegt
- Problem: benötigter Speicher ist ggf. durch äußere Faktoren bedingt (Nutzereingabe)

Index:	0	1	2	3	4	5	6	7	8	9	10	11
Adresse:	30	31	32	33	34	35	36	37	38	39	40	41
Daten:	4	48	89	1	0	9	13	7	32	76	17	5

## Ich wünsche mir: unbeschränkte Arrays

- Verhalten wie bei Arrays (direkter Zugriff mittels Index)
- zusätzliche Funktionen:

■ 

4	48	89	1	0
---	----	----	---	---

**pushBack(9)** → 

4	48	89	1	0	9
---	----	----	---	---	---

■ 

4	48	89	1	0
---	----	----	---	---

**popBack()** → 

4	48	89	1
---	----	----	---

■ 

4	48	89	1	0
---	----	----	---	---

**size()** → 5

# Beschränkte und Unbeschränkte Arrays

## Beschränkte Arrays

- beim Erstellen des Arrays muss ich festlegen, wie viel Speicher ich brauche
- nachträgliches Vergrößern schwierig: nachfolgende Speicherzellen ggf. schon belegt
- Problem: benötigter Speicher ist ggf. durch äußere Faktoren bedingt (Nutzereingabe)

Index:	0	1	2	3	4	5	6	7	8	9	10	11
Adresse:	30	31	32	33	34	35	36	37	38	39	40	41
Daten:	4	48	89	1	0	9	13	7	32	76	17	5

## Ich wünsche mir: unbeschränkte Arrays

- Verhalten wie bei Arrays (direkter Zugriff mittels Index)
- zusätzliche Funktionen:

■ 

4	48	89	1	0
---	----	----	---	---

**pushBack(9)** → 

4	48	89	1	0	9
---	----	----	---	---	---

■ 

4	48	89	1	0
---	----	----	---	---

**popBack()** → 

4	48	89	1
---	----	----	---

■ 

4	48	89	1	0
---	----	----	---	---

**size()** → 5

- Wünschen kann man sich ja viel, aber wie setzen wir das um?

# Unbeschränkte Arrays: naives Vorgehen

## Umsetzung von `pushBack(x)`

Index:	0	1	2	3	4
Adresse:	30	31	32	33	34
Daten:	4	48	89	1	0

# Unbeschränkte Arrays: naives Vorgehen

## Umsetzung von `pushBack(x)`

- erstelle neues Array, das um 1 größer ist

Index:	0	1	2	3	4	0	1	2	3	4	5
Adresse:	30	31	32	33	34	82	83	84	85	86	87
Daten:	4	48	89	1	0						

# Unbeschränkte Arrays: naives Vorgehen

## Umsetzung von `pushBack(x)`

- erstelle neues Array, das um 1 größer ist
- kopiere Daten von altem in neues Array

Index:	0	1	2	3	4	0	1	2	3	4	5
Adresse:	30	31	32	33	34	82	83	84	85	86	87
Daten:	4	48	89	1	0	4	48	89	1	0	

# Unbeschränkte Arrays: naives Vorgehen

## Umsetzung von `pushBack(x)`

- erstelle neues Array, das um 1 größer ist
- kopiere Daten von altem in neues Array
- schreibe  $x$  in die freie Zelle am Ende

Index:	0	1	2	3	4	0	1	2	3	4	5
Adresse:	30	31	32	33	34	82	83	84	85	86	87
Daten:	4	48	89	1	0	4	48	89	1	0	x

# Unbeschränkte Arrays: naives Vorgehen

## Umsetzung von `pushBack(x)`

- erstelle neues Array, das um 1 größer ist
- kopiere Daten von altem in neues Array
- schreibe  $x$  in die freie Zelle am Ende
- lösche altes Array

Index:	0	1	2	3	4
Adresse:	30	31	32	33	34
Daten:	4	48	89	1	0

0	1	2	3	4	5
82	83	84	85	86	87
4	48	89	1	0	x



# Unbeschränkte Arrays: naives Vorgehen

## Umsetzung von `pushBack(x)`

- erstelle neues Array, das um 1 größer ist
- kopiere Daten von altem in neues Array
- schreibe  $x$  in die freie Zelle am Ende
- lösche altes Array

Index:	0	1	2	3	4	0	1	2	3	4	5
Adresse:	30	31	32	33	34	82	83	84	85	86	87
Daten:	4	48	89	1	0	4	48	89	1	0	x

Wie lange dauert das Einfügen von  $n$  Elementen?



# Unbeschränkte Arrays: naives Vorgehen

## Umsetzung von `pushBack(x)`

- erstelle neues Array, das um 1 größer ist
- kopiere Daten von altem in neues Array
- schreibe  $x$  in die freie Zelle am Ende
- lösche altes Array

Index:	0	1	2	3	4
Adresse:	30	31	32	33	34
Daten:	4	48	89	1	0

0	1	2	3	4	5
82	83	84	85	86	87
4	48	89	1	0	x

## Kosten für das Einfügen von Elementen

- Einfügen von  $i$ tem Element: Kosten  $\Theta(i)$
- Einfügen von  $n$  Elementen:  $\sum_{i=1}^n \Theta(i) = \Theta\left(\sum_{i=1}^n i\right) = \Theta(n^2)$
- im Schnitt: Kosten  $\Theta(n)$  pro Operation  $\rightarrow$  viel zu teuer

# Unbeschränkte Arrays: naives Vorgehen

## Umsetzung von `pushBack(x)`

- erstelle neues Array, das um 1 größer ist
- kopiere Daten von altem in neues Array
- schreibe  $x$  in die freie Zelle am Ende
- lösche altes Array

Index:	0	1	2	3	4
Adresse:	30	31	32	33	34
Daten:	4	48	89	1	0

0	1	2	3	4	5
82	83	84	85	86	87
4	48	89	1	0	x

## Kosten für das Einfügen von Elementen

- Einfügen von  $i$ tem Element: Kosten  $\Theta(i)$
- Einfügen von  $n$  Elementen:  $\sum_{i=1}^n \Theta(i) = \Theta\left(\sum_{i=1}^n i\right) = \Theta(n^2)$
- im Schnitt: Kosten  $\Theta(n)$  pro Operation  $\rightarrow$  viel zu teuer

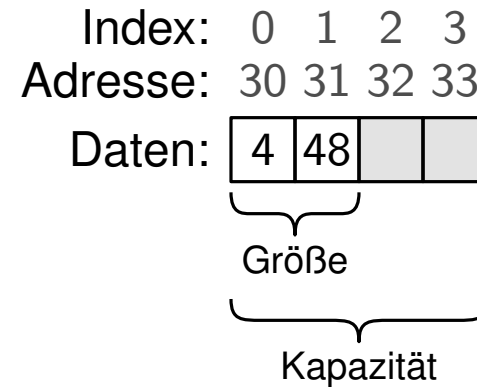
## Müssen wir wirklich jedes Mal umkopieren?

- alloziere großzügig etwas mehr Speicher
- nur dann kopieren, wenn der allozierte Speicher voll ist

# Unbeschränkte Arrays: besseres Vorgehen

## Größe vs. Kapazität

- Kapazität: für das Array allozierter Speicher
- Größe: tatsächlich benutzte Speicherzellen
- nach außen sichtbar: Größe (via `size()`)



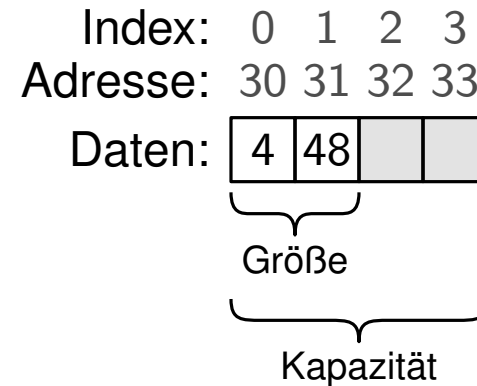
# Unbeschränkte Arrays: besseres Vorgehen

## Größe vs. Kapazität

- Kapazität: für das Array allozierter Speicher
- Größe: tatsächlich benutzte Speicherzellen
- nach außen sichtbar: Größe (via `size()`)

## Umsetzung von `pushBack(x)`

- falls Kapazität  $>$  Größe: einfach einfügen
- sonst: erstelle neues Array mit doppelter Kapazität und verfare wie bisher



# Unbeschränkte Arrays: besseres Vorgehen

## Größe vs. Kapazität

- Kapazität: für das Array allozierter Speicher
- Größe: tatsächlich benutzte Speicherzellen
- nach außen sichtbar: Größe (via `size()`)

## Umsetzung von `pushBack(x)`

- falls Kapazität  $>$  Größe: einfach einfügen
- sonst: erstelle neues Array mit doppelter Kapazität und verfahren wie bisher

Index:	0	1	2	3
Adresse:	30	31	32	33
Daten:	4	48		

**Beispiel:**

`pushBack(89)` `pushBack(1)` `pushBack(0)`

# Unbeschränkte Arrays: besseres Vorgehen

## Größe vs. Kapazität

- Kapazität: für das Array allozierter Speicher
- Größe: tatsächlich benutzte Speicherzellen
- nach außen sichtbar: Größe (via `size()`)

## Umsetzung von `pushBack(x)`

- falls Kapazität  $>$  Größe: einfach einfügen
- sonst: erstelle neues Array mit doppelter Kapazität und verfahren wie bisher

Index:	0	1	2	3
Adresse:	30	31	32	33
Daten:	4	48	89	

**Beispiel:**

`pushBack(89)` `pushBack(1)` `pushBack(0)`

# Unbeschränkte Arrays: besseres Vorgehen

## Größe vs. Kapazität

- Kapazität: für das Array allozierter Speicher
- Größe: tatsächlich benutzte Speicherzellen
- nach außen sichtbar: Größe (via `size()`)

## Umsetzung von `pushBack(x)`

- falls Kapazität  $>$  Größe: einfach einfügen
- sonst: erstelle neues Array mit doppelter Kapazität und verfahren wie bisher

Index:	0	1	2	3
Adresse:	30	31	32	33
Daten:	4	48	89	1

**Beispiel:**

`pushBack(89)` `pushBack(1)` `pushBack(0)`

# Unbeschränkte Arrays: besseres Vorgehen

## Größe vs. Kapazität

- Kapazität: für das Array allozierter Speicher
- Größe: tatsächlich benutzte Speicherzellen
- nach außen sichtbar: Größe (via `size()`)

## Umsetzung von `pushBack(x)`

- falls Kapazität  $>$  Größe: einfach einfügen
- sonst: erstelle neues Array mit doppelter Kapazität und verfahren wie bisher

Index:	0	1	2	3		0	1	2	3	4	5	6	7
Adresse:	30	31	32	33		82	83	84	85	86	87	88	89
Daten:	4	48	89	1									

**Beispiel:**

`pushBack(89)` `pushBack(1)` `pushBack(0)`



# Unbeschränkte Arrays: besseres Vorgehen

## Größe vs. Kapazität

- Kapazität: für das Array allozierter Speicher
- Größe: tatsächlich benutzte Speicherzellen
- nach außen sichtbar: Größe (via `size()`)

## Umsetzung von `pushBack(x)`

- falls Kapazität  $>$  Größe: einfach einfügen
- sonst: erstelle neues Array mit doppelter Kapazität und verfahren wie bisher

Index:	0	1	2	3	0	1	2	3	4	5	6	7
Adresse:	30	31	32	33	82	83	84	85	86	87	88	89
Daten:	4	48	89	1	4	48	89	1				

**Beispiel:**

`pushBack(89)` `pushBack(1)` `pushBack(0)`

# Unbeschränkte Arrays: besseres Vorgehen

## Größe vs. Kapazität

- Kapazität: für das Array allozierter Speicher
- Größe: tatsächlich benutzte Speicherzellen
- nach außen sichtbar: Größe (via `size()`)

## Umsetzung von `pushBack(x)`

- falls Kapazität  $>$  Größe: einfach einfügen
- sonst: erstelle neues Array mit doppelter Kapazität und verfahren wie bisher

Index:	0	1	2	3	0	1	2	3	4	5	6	7
Adresse:	30	31	32	33	82	83	84	85	86	87	88	89
Daten:	4	48	89	1	4	48	89	1	0			

**Beispiel:**

`pushBack(89)` `pushBack(1)` `pushBack(0)`

# Unbeschränkte Arrays: besseres Vorgehen

## Größe vs. Kapazität

- Kapazität: für das Array allozierter Speicher
- Größe: tatsächlich benutzte Speicherzellen
- nach außen sichtbar: Größe (via `size()`)

## Umsetzung von `pushBack(x)`

- falls Kapazität  $>$  Größe: einfach einfügen
- sonst: erstelle neues Array mit doppelter Kapazität und verfahren wie bisher

Index:	0	1	2	3	0	1	2	3	4	5	6	7
Adresse:	30	31	32	33	82	83	84	85	86	87	88	89
Daten:	4	48	89	1	4	48	89	1	0			

**Beispiel:**

`pushBack(89)` `pushBack(1)` `pushBack(0)`



# Unbeschränkte Arrays: besseres Vorgehen

## Größe vs. Kapazität

- Kapazität: für das Array allozierter Speicher
- Größe: tatsächlich benutzte Speicherzellen
- nach außen sichtbar: Größe (via `size()`)

## Umsetzung von `pushBack(x)`

- falls Kapazität  $>$  Größe: einfach einfügen
- sonst: erstelle neues Array mit doppelter Kapazität und verfahren wie bisher

Index:	0	1	2	3	0	1	2	3	4	5	6	7
Adresse:	30	31	32	33	82	83	84	85	86	87	88	89
Daten:	4	48	89	1	4	48	89	1	0			

**Beispiel:**

`pushBack(89)` `pushBack(1)` `pushBack(0)`

**Wie lange dauert das Einfügen von  $n$  Elementen?**

# Unbeschränkte Arrays: besseres Vorgehen

## Größe vs. Kapazität

- Kapazität: für das Array allozierter Speicher
- Größe: tatsächlich benutzte Speicherzellen
- nach außen sichtbar: Größe (via `size()`)

## Umsetzung von `pushBack(x)`

- falls Kapazität  $>$  Größe: einfach einfügen
- sonst: erstelle neues Array mit doppelter Kapazität und verfahren wie bisher

## Kosten für das Einfügen von Elementen

- Kosten für  $i$ tes Element, wenn  $i = 2^j + 1$ :  $\Theta(i) = \Theta(2^j)$
- sonst:  $\Theta(1)$

Index:	0	1	2	3	0	1	2	3	4	5	6	7
Adresse:	30	31	32	33	82	83	84	85	86	87	88	89
Daten:	4	48	89	1	4	48	89	1	0			

**Beispiel:**

`pushBack(89)` `pushBack(1)` `pushBack(0)`

# Unbeschränkte Arrays: besseres Vorgehen

## Größe vs. Kapazität

- Kapazität: für das Array allozierter Speicher
- Größe: tatsächlich benutzte Speicherzellen
- nach außen sichtbar: Größe (via `size()`)

Index:	0	1	2	3	0	1	2	3	4	5	6	7
Adresse:	30	31	32	33	82	83	84	85	86	87	88	89
Daten:	4	48	89	1	4	48	89	1	0			

## Umsetzung von `pushBack(x)`

- falls Kapazität > Größe: einfach einfügen
- sonst: erstelle neues Array mit doppelter Kapazität und verfare wie bisher

**Beispiel:**

`pushBack(89)` `pushBack(1)` `pushBack(0)`

## Kosten für das Einfügen von Elementen

- Kosten für  $i$ tes Element, wenn  $i = 2^j + 1$ :  $\Theta(i) = \Theta(2^j)$
- sonst:  $\Theta(1)$

- Einfügen von  $n$  Elementen:  $\sum_{i=1}^n \Theta(1) + \sum_{j=0}^{\log_2 n} \Theta(2^j)$

# Unbeschränkte Arrays: besseres Vorgehen

## Größe vs. Kapazität

- Kapazität: für das Array allozierter Speicher
- Größe: tatsächlich benutzte Speicherzellen
- nach außen sichtbar: Größe (via `size()`)

Index:	0	1	2	3	0	1	2	3	4	5	6	7
Adresse:	30	31	32	33	82	83	84	85	86	87	88	89
Daten:	4	48	89	1	4	48	89	1	0			

## Umsetzung von `pushBack(x)`

- falls Kapazität > Größe: einfach einfügen
- sonst: erstelle neues Array mit doppelter Kapazität und verfahren wie bisher

**Beispiel:**

`pushBack(89)` `pushBack(1)` `pushBack(0)`

## Kosten für das Einfügen von Elementen

- Kosten für  $i$ tes Element, wenn  $i = 2^j + 1$ :  $\Theta(i) = \Theta(2^j)$
- sonst:  $\Theta(1)$

- Einfügen von  $n$  Elementen:  $\sum_{i=1}^n \Theta(1) + \sum_{j=0}^{\log_2 n} \Theta(2^j) = \Theta(n) + \Theta(2^{\log_2 n}) = \Theta(n)$

# Unbeschränkte Arrays: besseres Vorgehen

## Größe vs. Kapazität

- Kapazität: für das Array allozierter Speicher
- Größe: tatsächlich benutzte Speicherzellen
- nach außen sichtbar: Größe (via `size()`)

Index:	0	1	2	3	0	1	2	3	4	5	6	7
Adresse:	30	31	32	33	82	83	84	85	86	87	88	89
Daten:	4	48	89	1	4	48	89	1	0			

## Umsetzung von `pushBack(x)`

- falls Kapazität > Größe: einfach einfügen
- sonst: erstelle neues Array mit doppelter Kapazität und verfare wie bisher

**Beispiel:**

`pushBack(89)` `pushBack(1)` `pushBack(0)`

## Kosten für das Einfügen von Elementen

- Kosten für  $i$ tes Element, wenn  $i = 2^j + 1$ :  $\Theta(i) = \Theta(2^j)$
- sonst:  $\Theta(1)$

■ Einfügen von  $n$  Elementen:  $\sum_{i=1}^n \Theta(1) + \sum_{j=0}^{\log_2 n} \Theta(2^j) = \Theta(n) + \Theta(2^{\log_2 n}) = \Theta(n)$

- im Schnitt: Kosten  $\Theta(1)$  pro Operation  $\rightarrow$  besser geht's nicht



# Unbeschränkte Arrays: Anmerkungen

## Speicherverschwendung

- wir allozieren mehr Speicher als wir tatsächlich brauchen
- aber: höchstens einen Faktor 2 zu viel

# Unbeschränkte Arrays: Anmerkungen

## Speicherverschwendung

- wir allozieren mehr Speicher als wir tatsächlich brauchen
- aber: höchstens einen Faktor 2 zu viel

## Was machen wir bei `popBack()`?

- hier kann man verschiedene Strategien verfolgen
- z.B.: einfach die Größe des Arrays verringern aber Kapazität beibehalten  $\rightarrow \Theta(1)$

# Unbeschränkte Arrays: Anmerkungen

## Speicherverschwendung

- wir allozieren mehr Speicher als wir tatsächlich brauchen
- aber: höchstens einen Faktor 2 zu viel

## Was machen wir bei `popBack()`?

- hier kann man verschiedene Strategien verfolgen
- z.B.: einfach die Größe des Arrays verringern aber Kapazität beibehalten  $\rightarrow \Theta(1)$

## Worst Case vs. Durchschnitt

- im Worst Case benötigt eine einzelne Operation  $\Theta(n)$  Zeit ( $n = \text{Arraygröße}$ )
- aber: im Schnitt benötigt jede Operation nur  $\Theta(1)$  Zeit

# Unbeschränkte Arrays: Anmerkungen

## Speicherverschwendung

- wir allozieren mehr Speicher als wir tatsächlich brauchen
- aber: höchstens einen Faktor 2 zu viel

## Was machen wir bei `popBack()`?

- hier kann man verschiedene Strategien verfolgen
- z.B.: einfach die Größe des Arrays verringern aber Kapazität beibehalten  $\rightarrow \Theta(1)$

## Worst Case vs. Durchschnitt

- im Worst Case benötigt eine einzelne Operation  $\Theta(n)$  Zeit ( $n = \text{Arraygröße}$ )
- aber: im Schnitt benötigt jede Operation nur  $\Theta(1)$  Zeit
- wir sagen: `pushBack()` hat **amortisiert** konstante Laufzeit

# Unbeschränkte Arrays: Anmerkungen

## Speicherverschwendung

- wir allozieren mehr Speicher als wir tatsächlich brauchen
- aber: höchstens einen Faktor 2 zu viel

## Was machen wir bei `popBack()`?

- hier kann man verschiedene Strategien verfolgen
- z.B.: einfach die Größe des Arrays verringern aber Kapazität beibehalten  $\rightarrow \Theta(1)$

## Worst Case vs. Durchschnitt

- im Worst Case benötigt eine einzelne Operation  $\Theta(n)$  Zeit ( $n = \text{Arraygröße}$ )
- aber: im Schnitt benötigt jede Operation nur  $\Theta(1)$  Zeit
- wir sagen: `pushBack()` hat **amortisiert** konstante Laufzeit

## Abgrenzung zum Average Case

(werden wir später noch kennen lernen)

- Average Case: Durchschnitt über alle möglichen Eingaben/Ausführungen
- amortisiert: Durchschnitt über alle Operationen für beliebige (Worst Case) Eingabe

# Unbeschränkte Arrays in der Wildnis

C++

<https://en.cppreference.com/w/cpp/container/vector>

## std::vector

---

The complexity (efficiency) of common operations on vectors is as follows:

- Random access - constant  $\mathcal{O}(1)$
- Insertion or removal of elements at the end - amortized constant  $\mathcal{O}(1)$

---

<b>operator[]</b>	access specified element (public member function)
<b>size</b>	returns the number of elements (public member function)
<b>capacity</b>	returns the number of elements that can be held in currently allocated storage (public member function)
<b>push_back</b>	adds an element to the end (public member function)
<b>pop_back</b>	removes the last element (public member function)

---

# Unbeschränkte Arrays in der Wildnis

**Java** <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/ArrayList.html>

## Class ArrayList<E>

---

Each `ArrayList` instance has a *capacity*. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an `ArrayList`, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.

### Method Summary

Modifier and Type	Method	Description
boolean	<code>add(E e)</code>	Appends the specified element to the end of this list.
<b>E</b>	<code>get(int index)</code>	Returns the element at the specified position in this list.
int	<code>size()</code>	Returns the number of elements in this list.
void	<code>trimToSize()</code>	Trims the capacity of this <code>ArrayList</code> instance to be the list's current size.

# Unbeschränkte Arrays: Zusammenfassung

## Effiziente Operationen ((amortisiert) konstant)

- Zugriff an beliebiger Stelle
- hinten einfügen
- hinten löschen



# Unbeschränkte Arrays: Zusammenfassung

## Effiziente Operationen ((amortisiert) konstant)

- Zugriff an beliebiger Stelle
- hinten einfügen
- hinten löschen

## Langsame Operationen (linear)

- einfügen an beliebiger Stelle
- löschen an beliebiger Stelle
- zwei Arrays konkatenieren
- Teilarray löschen

# Ausflug: Amortisierte Analyse

## Situation

- Sequenz von vielen Operationen
- manche sind teuer, die meisten sind günstig → günstig im Durchschnitt

## Amortisierte Analyse

- Worst Case über alle möglichen Sequenzen von Operationen
- Garantie: Gesamtlaufzeit verhält sich, als wäre jede Operation günstig

# Ausflug: Amortisierte Analyse

## Situation

- Sequenz von vielen Operationen
- manche sind teuer, die meisten sind günstig → günstig im Durchschnitt

## Amortisierte Analyse

- Worst Case über alle möglichen Sequenzen von Operationen
- Garantie: Gesamtlaufzeit verhält sich, als wäre jede Operation günstig

## Jetzt: verschiedene Techniken für die Analyse

- Aggregation
- Charging
- Konto
- Potential
- jeweils am Beispiel von  $n$  **pushBack** Operationen

# Amortisierte Analyse: Aggregation

## Allgemeines Vorgehen

- summiere die Kosten für alle Operationen
- teile Gesamtkosten durch Anzahl Operationen

# Amortisierte Analyse: Aggregation

## Allgemeines Vorgehen

- summiere die Kosten für alle Operationen
- teile Gesamtkosten durch Anzahl Operationen

## Am Beispiel **pushBack**

- Kosten für das Einfügen von  $n$  Elementen:  $\sum_{i=1}^n \Theta(1) + \sum_{j=0}^{\log_2 n} \Theta(2^j) = \Theta(n)$
- durchschnittliche Kosten pro Operation:  $\Theta(1)$

# Amortisierte Analyse: Aggregation

## Allgemeines Vorgehen

- summiere die Kosten für alle Operationen
- teile Gesamtkosten durch Anzahl Operationen

## Am Beispiel **pushBack**

- Kosten für das Einfügen von  $n$  Elementen:  $\sum_{i=1}^n \Theta(1) + \sum_{j=0}^{\log_2 n} \Theta(2^j) = \Theta(n)$
- durchschnittliche Kosten pro Operation:  $\Theta(1)$

## Anmerkungen

- schön einfach und direkt
- manchmal unpraktikabel, insbesondere bei mehrere Operationen unterschiedlichen Typs

# Amortisierte Analyse: Charging

## Allgemeines Vorgehen

- verteile Kosten-Tokens von teuren zu günstigen Operationen (Charging)
- zeige: jede Operation hat am Ende nur wenige Tokens

# Amortisierte Analyse: Charging

## Allgemeines Vorgehen

- verteile Kosten-Tokens von teuren zu günstigen Operationen (Charging)
- zeige: jede Operation hat am Ende nur wenige Tokens

## Am Beispiel **pushBack**

- $i$ te Operation hat  $i$  Tokens, falls  $i = 2^j + 1$ , sonst 1 Token





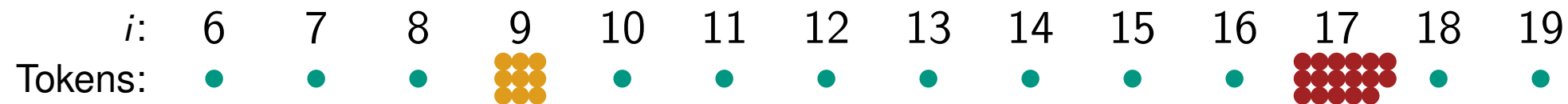
# Amortisierte Analyse: Charging

## Allgemeines Vorgehen

- verteile Kosten-Tokens von teuren zu günstigen Operationen (Charging)
- zeige: jede Operation hat am Ende nur wenige Tokens

## Am Beispiel **pushBack**

- $i$ te Operation hat  $i$  Tokens, falls  $i = 2^j + 1$ , sonst 1 Token



- verschiebe von  $2^j + 1$  je zwei Tokens auf jede der  $2^{j-1} - 1$  Operation in  $(2^{j-1} + 1, 2^j]$



# Amortisierte Analyse: Charging

## Allgemeines Vorgehen

- verteile Kosten-Tokens von teuren zu günstigen Operationen (Charging)
- zeige: jede Operation hat am Ende nur wenige Tokens

## Am Beispiel **pushBack**

- $i$ te Operation hat  $i$  Tokens, falls  $i = 2^j + 1$ , sonst 1 Token

$i:$	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Tokens:	●	●	●	●●●●	●	●	●	●	●	●	●	●●●●●●	●	●

- verschiebe von  $2^j + 1$  je zwei Tokens auf jede der  $2^{j-1} - 1$  Operation in  $(2^{j-1} + 1, 2^j]$

**Wie viele Kostentoken haben die Operationen nach der Verschiebung?**

# Amortisierte Analyse: Charging

## Allgemeines Vorgehen

- verteile Kosten-Tokens von teuren zu günstigen Operationen (Charging)
- zeige: jede Operation hat am Ende nur wenige Tokens

## Am Beispiel **pushBack**

- $i$ te Operation hat  $i$  Tokens, falls  $i = 2^j + 1$ , sonst 1 Token

$i:$	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Tokens:	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●	●

- verschiebe von  $2^j + 1$  je zwei Tokens auf jede der  $2^{j-1} - 1$  Operation in  $(2^{j-1} + 1, 2^j]$

# Amortisierte Analyse: Charging

## Allgemeines Vorgehen

- verteile Kosten-Tokens von teuren zu günstigen Operationen (Charging)
- zeige: jede Operation hat am Ende nur wenige Tokens

## Am Beispiel **pushBack**

- $i$ te Operation hat  $i$  Tokens, falls  $i = 2^j + 1$ , sonst 1 Token

$i:$	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Tokens:	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●	●

- verschiebe von  $2^j + 1$  je zwei Tokens auf jede der  $2^{j-1} - 1$  Operation in  $(2^{j-1} + 1, 2^j]$
- falls  $i = 2^j + 1$ : Operation wird  $2^j - 2 = i - 3$  Tokens los  $\rightarrow$  3 Tokens verbleiben
- sonst: Operation erhält höchstens zwei zusätzliche Tokens  $\rightarrow$  maximal 3 Tokens

# Amortisierte Analyse: Charging

## Allgemeines Vorgehen

- verteile Kosten-Tokens von teuren zu günstigen Operationen (Charging)
- zeige: jede Operation hat am Ende nur wenige Tokens

## Am Beispiel **pushBack**

- $i$ te Operation hat  $i$  Tokens, falls  $i = 2^j + 1$ , sonst 1 Token

$i:$	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Tokens:	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●	●

- verschiebe von  $2^j + 1$  je zwei Tokens auf jede der  $2^{j-1} - 1$  Operation in  $(2^{j-1} + 1, 2^j]$
- falls  $i = 2^j + 1$ : Operation wird  $2^j - 2 = i - 3$  Tokens los  $\rightarrow$  3 Tokens verbleiben
- sonst: Operation erhält höchstens zwei zusätzliche Tokens  $\rightarrow$  maximal 3 Tokens

## Anmerkung

- lokal: hohe Kosten fallen an  $\rightarrow$  charge sie rückwirkend auf vorherige Operationen
- global: aufpassen, dass keine Operation zu viele Token bekommt

# Amortisierte Analyse: Konto

## Allgemeines Vorgehen

- günstige Operation: bezahlt mehr als sie tatsächlich kostet → ins Konto einzahlen
- teure Operation: bezahlt tatsächlich Kosten zum Teil mit Guthaben aus dem Konto

# Amortisierte Analyse: Konto

## Allgemeines Vorgehen

- günstige Operation: bezahlt mehr als sie tatsächlich kostet → ins Konto einzahlen
- teure Operation: bezahlt tatsächlich Kosten zum Teil mit Guthaben aus dem Konto

## Am Beispiel **pushBack**

- günstige Operation ( $i \in (2^{j-1} + 1, 2^j]$ ): Kosten 1, bezahle 3 → zahle 2 ins Konto ein
- Guthaben nach Operation  $2^j$  beträgt mindestens  $2 \cdot (2^j - (2^{j-1} + 1)) = 2^j - 2$

# Amortisierte Analyse: Konto

## Allgemeines Vorgehen

- günstige Operation: bezahlt mehr als sie tatsächlich kostet → ins Konto einzahlen
- teure Operation: bezahlt tatsächlich Kosten zum Teil mit Guthaben aus dem Konto

## Am Beispiel **pushBack**

- günstige Operation ( $i \in (2^{j-1} + 1, 2^j]$ ): Kosten 1, bezahle 3 → zahle 2 ins Konto ein
- Guthaben nach Operation  $2^j$  beträgt mindestens  $2 \cdot (2^j - (2^{j-1} + 1)) = 2^j - 2$
- teure Operation ( $i = 2^j + 1$ ): Kosten  $i$ , bezahle 3 → hebe  $i - 3 = 2^j - 2$  vom Konto ab



# Amortisierte Analyse: Konto

## Allgemeines Vorgehen

- günstige Operation: bezahlt mehr als sie tatsächlich kostet → ins Konto einzahlen
- teure Operation: bezahlt tatsächlich Kosten zum Teil mit Guthaben aus dem Konto

## Am Beispiel **pushBack**

- günstige Operation ( $i \in (2^{j-1} + 1, 2^j]$ ): Kosten 1, bezahle 3 → zahle 2 ins Konto ein
- Guthaben nach Operation  $2^j$  beträgt mindestens  $2 \cdot (2^j - (2^{j-1} + 1)) = 2^j - 2$
- teure Operation ( $i = 2^j + 1$ ): Kosten  $i$ , bezahle 3 → hebe  $i - 3 = 2^j - 2$  vom Konto ab
- jede Operation bezahlt nur 3 und Konto ist nie negativ  $\Rightarrow$  amortisiert  $\Theta(1)$

# Amortisierte Analyse: Konto

## Allgemeines Vorgehen

- günstige Operation: bezahlt mehr als sie tatsächlich kostet → ins Konto einzahlen
- teure Operation: bezahlt tatsächlich Kosten zum Teil mit Guthaben aus dem Konto

## Am Beispiel **pushBack**

- günstige Operation ( $i \in (2^{j-1} + 1, 2^j]$ ): Kosten 1, bezahle 3 → zahle 2 ins Konto ein
- Guthaben nach Operation  $2^j$  beträgt mindestens  $2 \cdot (2^j - (2^{j-1} + 1)) = 2^j - 2$
- teure Operation ( $i = 2^j + 1$ ): Kosten  $i$ , bezahle 3 → hebe  $i - 3 = 2^j - 2$  vom Konto ab
- jede Operation bezahlt nur 3 und Konto ist nie negativ  $\Rightarrow$  amortisiert  $\Theta(1)$

## Anmerkung

- lokal: günstige Operationen zahlen vorausschauend Kosten für spätere Operationen
- global: aufpassen, dass nie mehr abgehoben wird als schon eingezahlt wurde
- sehr ähnlich zum Charging, aber leicht andere Perspektive

# Amortisierte Analyse: Potential

## Allgemeines Vorgehen

- bei der Konto-Methode:
  - definiere wie viel jede Operation (amortisiert) bezahlt
  - daraus ergibt sich: Änderung des Kontostands

# Amortisierte Analyse: Potential

## Allgemeines Vorgehen

- bei der Konto-Methode:
  - definiere wie viel jede Operation (amortisiert) bezahlt
  - daraus ergibt sich: Änderung des Kontostands
- bei der Potential-Methode:
  - definiere Kontostand abhängig vom Zustand der Datenstruktur → Potentialfunktion
  - daraus ergibt sich: die amortisierten Kosten jeder Operation

# Amortisierte Analyse: Potential

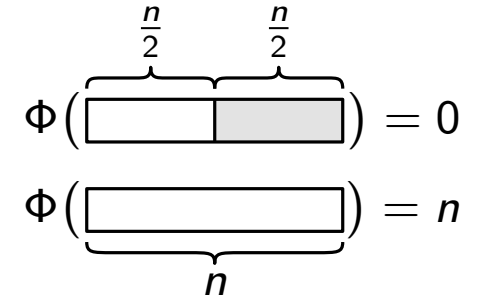
## Allgemeines Vorgehen

- bei der Konto-Methode:
  - definiere wie viel jede Operation (amortisiert) bezahlt
  - daraus ergibt sich: Änderung des Kontostands
- bei der Potential-Methode:
  - definiere Kontostand abhängig vom Zustand der Datenstruktur → Potentialfunktion
  - daraus ergibt sich: die amortisierten Kosten jeder Operation
- genauer:
  - Potentialfunktion  $\Phi(A)$ : bildet den Zustand des Arrays  $A$  auf eine Zahl  $\geq 0$  ab
  - $A_{\text{vor}}$  und  $A_{\text{nach}}$ : die Zustände vor bzw. nach einer Operation
  - amortisierte Kosten = tatsächliche Kosten +  $\Phi(A_{\text{nach}}) - \Phi(A_{\text{vor}})$

# Amortisierte Analyse: Potential

## Am Beispiel **pushBack**

- Potentialfunktion:  $\Phi(A) = 2 \cdot A.\text{size}() - A.\text{capacity}()$
- beachte:  $\Phi(A) \geq 0$

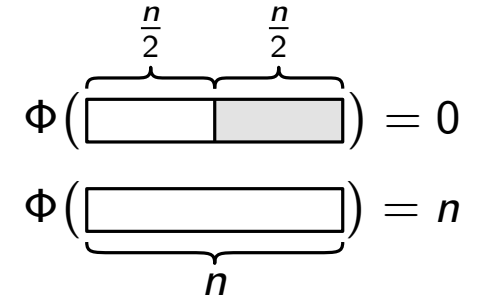


**Erinnerung:** amortisierte Kosten = tatsächliche Kosten +  $\Phi(A_{\text{nach}}) - \Phi(A_{\text{vor}})$

# Amortisierte Analyse: Potential

## Am Beispiel **pushBack**

- Potentialfunktion:  $\Phi(A) = 2 \cdot A.\text{size}() - A.\text{capacity}()$
- beachte:  $\Phi(A) \geq 0$
- günstige Operation:
  - Größe wächst um 1; Kapazität bleibt gleich
  - $\Phi(A_{\text{nach}}) - \Phi(A_{\text{vor}}) = 2$
  - amortisierte Kosten = tatsächliche Kosten + 2  $\in \Theta(1)$

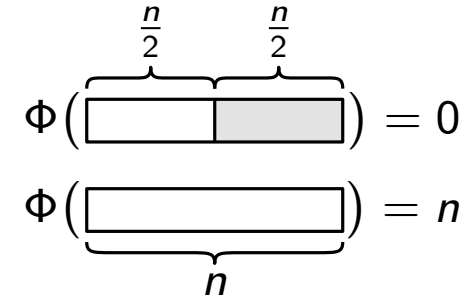


**Erinnerung:** amortisierte Kosten = tatsächliche Kosten +  $\Phi(A_{\text{nach}}) - \Phi(A_{\text{vor}})$

# Amortisierte Analyse: Potential

## Am Beispiel **pushBack**

- Potentialfunktion:  $\Phi(A) = 2 \cdot A.\text{size}() - A.\text{capacity}()$
- beachte:  $\Phi(A) \geq 0$
- günstige Operation:
  - Größe wächst um 1; Kapazität bleibt gleich
  - $\Phi(A_{\text{nach}}) - \Phi(A_{\text{vor}}) = 2$
  - amortisierte Kosten = tatsächliche Kosten + 2  $\in \Theta(1)$
- teure Operation ( $i = 2^j + 1$ ):
  - Größe wächst um 1; Kapazität wächst um  $2^j$
  - $\Phi(A_{\text{nach}}) - \Phi(A_{\text{vor}}) = 2 - 2^j$
  - amortisierte Kosten = tatsächliche Kosten + 2 -  $2^j \in \Theta(1)$



**Erinnerung:** amortisierte Kosten = tatsächliche Kosten +  $\Phi(A_{\text{nach}}) - \Phi(A_{\text{vor}})$

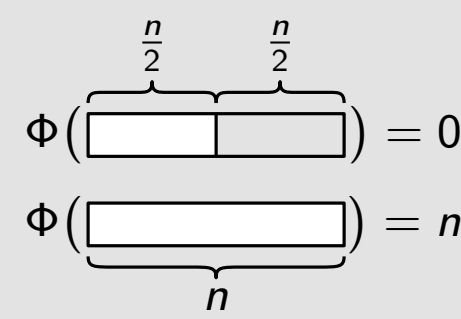


# Amortisierte Analyse: Potential – Anmerkungen

## Wie kommt man auf die Potentialfunktion?

- das ist der Knackpunkt; wenn man die erstmal hat ist der Rest einfach

**Erinnerung:**  $\Phi$  für `pushBack()`

$$\Phi(A) = 2 \cdot A.size() - A.capacity()$$


The diagram illustrates two states of an array:

- A full array of size  $n$  with  $\Phi = n$ .
- A half-full array of size  $n/2$  with  $\Phi = 0$ .

# Amortisierte Analyse: Potential – Anmerkungen

## Wie kommt man auf die Potentialfunktion?

- das ist der Knackpunkt; wenn man die erstmal hat ist der Rest einfach
- Interpretation 1: Maß dafür, wie nah man an einer teuren Operation ist
  - niedrig: kommende Operationen sind günstig
  - hoch: bald kommt vermutlich eine teure Operation

**Erinnerung:**  $\Phi$  für `pushBack()`

$$\Phi(A) = 2 \cdot A.\text{size}() - A.\text{capacity}()$$

$$\Phi\left(\overbrace{\boxed{\phantom{a}}}^{\frac{n}{2}} \overbrace{\boxed{\phantom{a}}}^{\frac{n}{2}}\right) = 0$$

$$\Phi\left(\underbrace{\boxed{\phantom{a}}}_{n}\right) = n$$





# Amortisierte Analyse: Zusammenfassung

## Wann ist eine amortisierte Betrachtungsweise angebracht?

- bei einer Folge unterschiedlich teurer Operationen
- vor jeder teuren Operation kommen immer ausreichend viele günstige Operationen

# Amortisierte Analyse: Zusammenfassung

## Wann ist eine amortisierte Betrachtungsweise angebracht?

- bei einer Folge unterschiedlich teurer Operationen
- vor jeder teuren Operation kommen immer ausreichend viele günstige Operationen

## Was garantiert eine amortisierte Laufzeitschranke?

- gute Gesamtlaufzeit (Worst Case) über alle Operationen hinweg
- keine Garantie für einzelne Operation (relevant z.B. in Echtzeitsystemen oder bei Parallelisierung)

# Amortisierte Analyse: Zusammenfassung

## Wann ist eine amortisierte Betrachtungsweise angebracht?

- bei einer Folge unterschiedlich teurer Operationen
- vor jeder teuren Operation kommen immer ausreichend viele günstige Operationen

## Was garantiert eine amortisierte Laufzeitschranke?

- gute Gesamtlaufzeit (Worst Case) über alle Operationen hinweg
- keine Garantie für einzelne Operation (relevant z.B. in Echtzeitsystemen oder bei Parallelisierung)

## Techniken für die amortisierte Analyse

- Aggregation: summiere Kosten über alle Operationen
- Charging: charge Kosten von teuren Operationen auf vorherige günstige Operationen
- Konto: lass günstige Operationen später folgende teure Operationen vorab bezahlen
- Potential: definiere Unordnungskonto, das misst wie unschön der aktuelle Zustand ist

# Zusammenfassung

## Beschränkte Arrays

- ein Stück Speicher (konsequente Speicherzellen) auf das man mittels Adresse zugreift
- andere Arten Daten zu speichern basieren immer auf diesem Grundbaustein



# Zusammenfassung

## Beschränkte Arrays

- ein Stück Speicher (konsequente Speicherzellen) auf das man mittels Adresse zugreift
- andere Arten Daten zu speichern basieren immer auf diesem Grundbaustein

## Unbeschränkte Arrays

- Interface für dynamisch wachsendes Array → abstrahiert von einem Stück Speicher
- effiziente Implementierung mittels sinnvoller Größenänderungs-Strategie

# Zusammenfassung

## Beschränkte Arrays

- ein Stück Speicher (konsequente Speicherzellen) auf das man mittels Adresse zugreift
- andere Arten Daten zu speichern basieren immer auf diesem Grundbaustein

## Unbeschränkte Arrays

- Interface für dynamisch wachsendes Array → abstrahiert von einem Stück Speicher
- effiziente Implementierung mittels sinnvoller Größenänderungs-Strategie

## Drei Sichtweisen auf eine Datenstruktur

- **Mathe** abstraktes Objekt (hier: Folge von Zahlen  $\langle 4, 48, 89, 1, 0, 9, 13, 7, 32, 76, 17, 5 \rangle$ )
- **Softwaretechnik** Funktionalität (hier: Zugriff mit Index, **pushBack**, **popBack**, **size**)
- **Algorithmik** Repräsentation und effiziente Umsetzung

# Zusammenfassung

## Beschränkte Arrays

- ein Stück Speicher (konsequente Speicherzellen) auf das man mittels Adresse zugreift
- andere Arten Daten zu speichern basieren immer auf diesem Grundbaustein

## Unbeschränkte Arrays

- Interface für dynamisch wachsendes Array → abstrahiert von einem Stück Speicher
- effiziente Implementierung mittels sinnvoller Größenänderungs-Strategie

## Drei Sichtweisen auf eine Datenstruktur

- **Mathe** abstraktes Objekt (hier: Folge von Zahlen  $\langle 4, 48, 89, 1, 0, 9, 13, 7, 32, 76, 17, 5 \rangle$ )
- **Softwaretechnik** Funktionalität (hier: Zugriff mit Index, **pushBack**, **popBack**, **size**)
- **Algorithmik** Repräsentation und effiziente Umsetzung

## Amortisierte Analyse

- wichtige Technik für die Analyse von Algorithmen

# Hochschulgruppen

## Was ist das?

- studentisch organisierte Gruppen
- große thematische Bandbreite: von sozialem Engagement bis hin zum Segelfliegen

## Mitgliederwerbung

- viele Gruppen suchen neue Mitglieder
- einige Infoveranstaltungen in den nächsten Wochen
- siehe auch: Link auf der Homepage