Algorithmen und Datenstrukturen (ESE)
Entwurf, Analyse und Umsetzung von Algorithmen (IEMS)
WS 2014 / 2015

Vorlesung 7, Donnerstag 4. Dezember 2014
(Dynamische Felder, Amortisierte Analyse)

Junior-Prof. Dr. Olaf Ronneberger
Image Analysis Lab
Institut für Informatik
Universität Freiburg

[basiert auf der gleichnamigen Veranstaltung von Prof. Dr. Hannah Bast, Lehrstuhl für Algorithmen und Datenstrukturen, mit Folien aus „Informatik 2" von Prof. Dr. Matthias Teschner]

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Ihre Erfahrungen mit dem 6. Übungsblatt
- Dynamische Felder
  - Was ist das (im Gegensatz zu statischen Feldern)
  - Standardbibliotheken dazu in Java und in C++
  - Ein paar eigene Implementierungsvarianten
  - Laufzeitanalyse dieser Varianten
  - Ein schönes Beispiel wo man Mathematik braucht, um zu verstehen, warum man es so machen muss und nicht anders
  - Übungsblatt 7: Erweiterung des Codes aus der Vorlesung
  - Präsenz-Übung am nächsten Montag: Tests und Checkstyle in Java, und natürlich sonstige Fragen.

Erfahrungen mit dem Ü6 (PriorityQueue)

UNI FREIBURG

- Zusammenfassung / Auszüge Stand 4.12. 08:30
  - Zeitaufwand: ca. 4-5 Stunden
  - wie private Methoden testen? → eigentlich gar nicht, in Ausnahmefällen mit FRIEND_TEST
  - welcher Vorteil, die Implementationen ins .cpp File zu schreiben? → kleineres Executable, schnellere Kompilierung, wichtig für „globale Variablen", für Libraries, und für closed source
  - was ist *-inl.h? → steht im Google C++ Style Guide
  - verändertes cpplint? → Jenkins nutzt die cpplint, die sie eingecheckt haben
  - Tippfehler in Folien → korrigiert. Vielen Dank
  - komische Checkstyle Fehler → Übung am Montag

Felder fester Größe

UNI FREIBURG

- ... gibt es sowohl in Java:

```
int[] numbers = new int[100];  // Array of 100 ints, initializ. to 0.
System.out.println(numbers[12]);  // Prints 0.
String[] strings = new String[10];  // Array of 10 strings.
System.out.println(strings[7]);  // Prints empty string.
strings[8] = "hello";
```

- ... als auch in C++

```
int[] numbers = new int[100];  // Pointer to 100 ints, no initializ.
std::cout << numbers[12];  //  Prints random number.
string[] strings = new string[10];  // Pointer to 10 strings.
std::cout << strings[7];  // Prints empty string.
strings[8] = "hello";
```

- Größe muss bei der Erzeugung festgelegt werden!
  - Die benötigte Größe ergibt sich aber oft erst im Laufe des Progrs

Dynamische und Statische Felder

UNI FREIBURG

- Der Name "statisch" ist etwas irreführend
  - Es hat nichts mit dem keyword static in Java oder in C++ zu tun
  - Die Felder sind auch nicht statisch in dem Sinne, dass der Speicherplatz schon vor der Ausführung des Programmes alloziert wird, im Gegenteil
  - Was statisch ist, ist die **Größe** des Feldes
    - die muss bei der Erzeugung des Feldes explizit angegeben werden
    - und kann danach nicht mehr geändert werden
  - Von daher ist Feld fester Größe bzw. fixed-size array ein besserer Name

Dynamische Felder

UNI FREIBURG

- ... können beliebig vergrößert / verkleinert werden
  - In Java haben wir dafür bisher immer ArrayList benutzt

```
ArrayList<String> words = new ArrayList<String>();
words.add("hello");
words.add("world");
words.add("ohai");
System.out.println(words.get(0));  // Will print hello.
words.clear();  // Remove all elements.
```

  - In C++ nimmt man dafür std::vector

```
vector<string> words;
words.push_back("hello");
...
words.resize(2);  // Keep only first 2 elements.
words.clear();  // Remove all elements.
```

Dynamische Felder — Implementierung

UNI FREIBURG

- Das Prinzip ist ganz einfach
  - Man hat intern ein fixed-size array
    - von der Größe, die man gerade braucht
  - Wenn Elemente dazu kommen:
    - erzeugt man ein neues fixed-size array der benötigten Größe
    - und kopiert die Elemente vom alten in das neue Feld
  - Wenn Elemente entfernt werden
    - erzeugt man ein neues fixed-size array der benötigten Größe
    - und kopiert die Elemente von alten in das neue Feld
  - Das sehen wir uns jetzt an
    - Vorlesung: nur append (= neues Element anhängen)
    - Übungsblatt: append **und** remove (= letztes El. entfernen)

Version 1: Jedes Mal vergrößern (1/7)

UNI FREIBURG

- Einfachste Vergrößerungsstrategie
  - Wir vergrößern das Feld vor jedem append
  - Und machen es immer genauso groß, wie wir es brauchen

Version 1: Jedes Mal vergrößern (2/7)

UNI FREIBURG

Die Klasse Drumherum

```
public class DynamicArray {

  public DynamicArray() {
    this.data = new int[0];
    this.nofElements = 0;
  }

  public int getCapacity() {
    return this.data.length;
  }

  public void append(int item) {
    // … nächste Folie
  }

  private int[] data;
  private int nofElements;
}
```

Version 1: Jedes Mal vergrößern (3/7)

UNI FREIBURG

Die eigentliche append Methode

```
public void append(int item) {
    // grow the capacity
    int newCapacity = this.getCapacity() + 1;
    int[] newData = new int[newCapacity];
    for (int i = 0; i < nofElements; i++) { newData[i] = data[i]; }
    this.data = newData;

    // append item
    this.data[nofElements] = item;
    nofElements++;
  }
```

Version 1: Jedes Mal vergrößern (4/7)

- Hauptpgramm: 100.000 Elemente einfügen

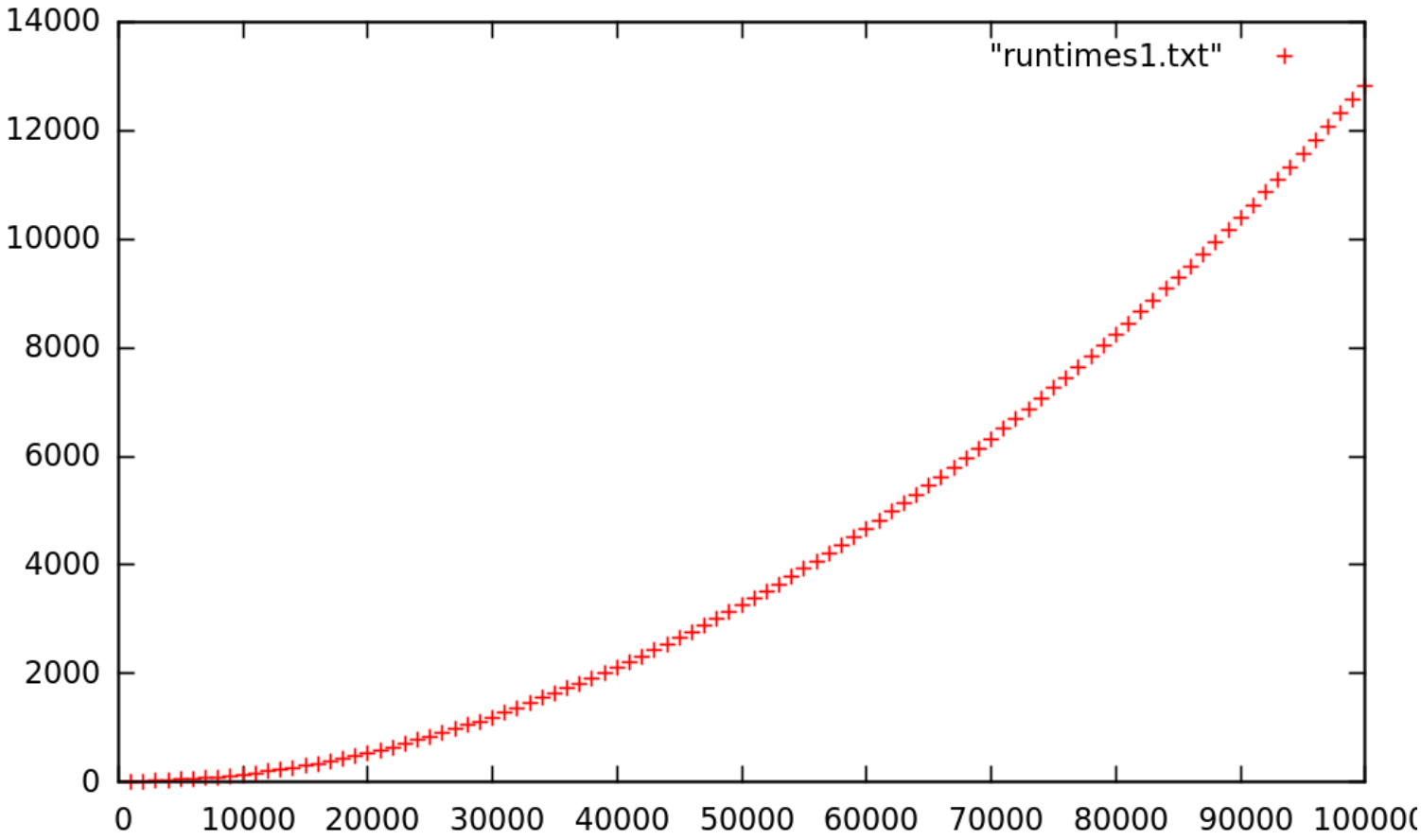
```
public class DynamicArrayMain {
  public static void main(String[] args) {
    DynamicArray list = new DynamicArray();
    long start = System.currentTimeMillis();

    // Append 100.000 ints to array.
    for (int i = 1; i <= 100 * 1000; ++i) {
      list.append(i);
      // Take the time after a certain amount of appends.
      if (i % 1000 == 0) {
        long now = System.currentTimeMillis();
        System.out.println(i + "\t" + (now - start));
      }
    }
  }
}
```

a = i++;

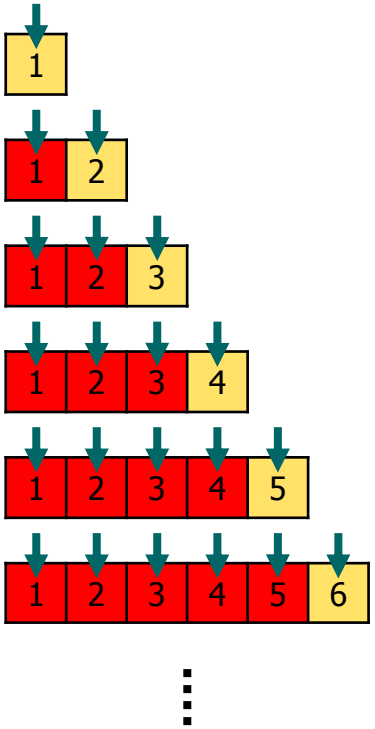
a = ++i;

Version 1: Jedes Mal vergrößern (5/7)



- Die Laufzeitkurze zeigt quadratisches Verhalten, warum?

Version 1: Jedes Mal vergrößern (6/7)



O(1)	1 Element reinschreiben
O(1**+1**)	1 Element reinschreiben und 1 kopieren
O(1**+2**)	1 Element reinschreiben und 2 kopieren
O(1**+3**)	1 Element reinschreiben und 3 kopieren
O(1**+4**)	1 Element reinschreiben und 4 kopieren
O(1**+5**)	1 Element reinschreiben und 5 kopieren

Version 1: Jedes Mal vergrößern (6/7)

UNI FREIBURG

- Analyse
  - Sei $T(n)$ die Laufzeit für eine Folge von n append Operationen
  - Sei T_i die Laufzeit für die i-te append Operation
  - Dann ist $T_i = A \cdot i$ für irgendeine Konstante A
    - weil wir $i-1$ Elemente reallozieren (umkopieren) müssen
  - Das macht zusammen:

$$T(n) = \sum_{i=1}^{n} T_i = \sum_{i=1}^{n} A \cdot i = A \cdot \underbrace{\sum_{i=1}^{n} i}_{=1+2+3+\ldots} = A \cdot \frac{n^2+n}{2} = \mathcal{O}(n^2)$$

Version 2: Jedes C. Mal vergrößern (1/5)

UNI FREIBURG

- Eine etwas vorausschauendere Vergrößerungsstrategie
  - Idee: beim Vergrößern zusätzlichen Platz lassen
  - Aber wieviel?
  - Lassen wir erstmal Platz für C zusätzliche Elemente, für ein beliebiges festes C, zum Beispiel C = 5 oder C = 100

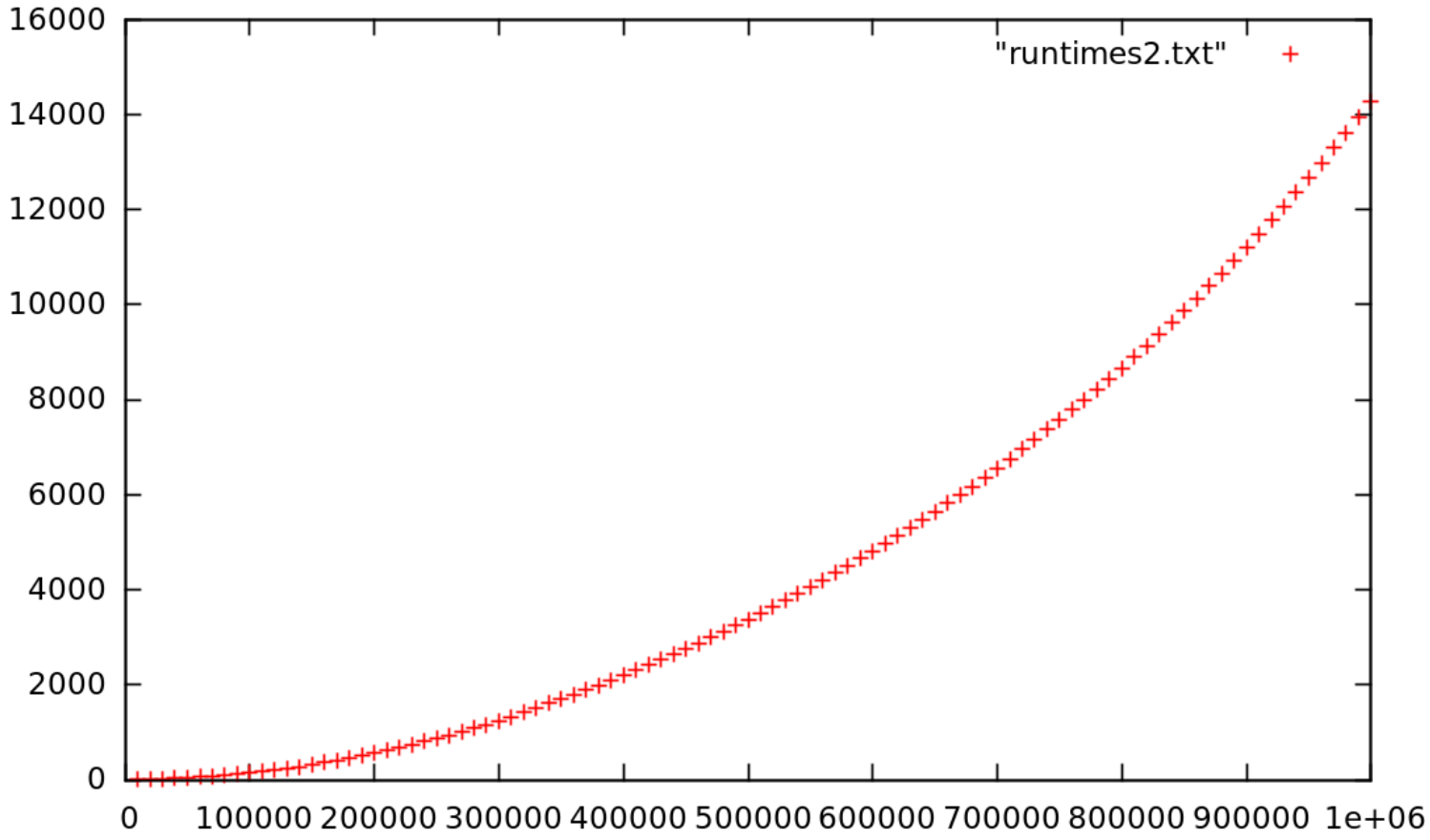
Version 2: Jedes C. Mal vergrößern (2/5)

UNI FREIBURG

Die Append Funktion

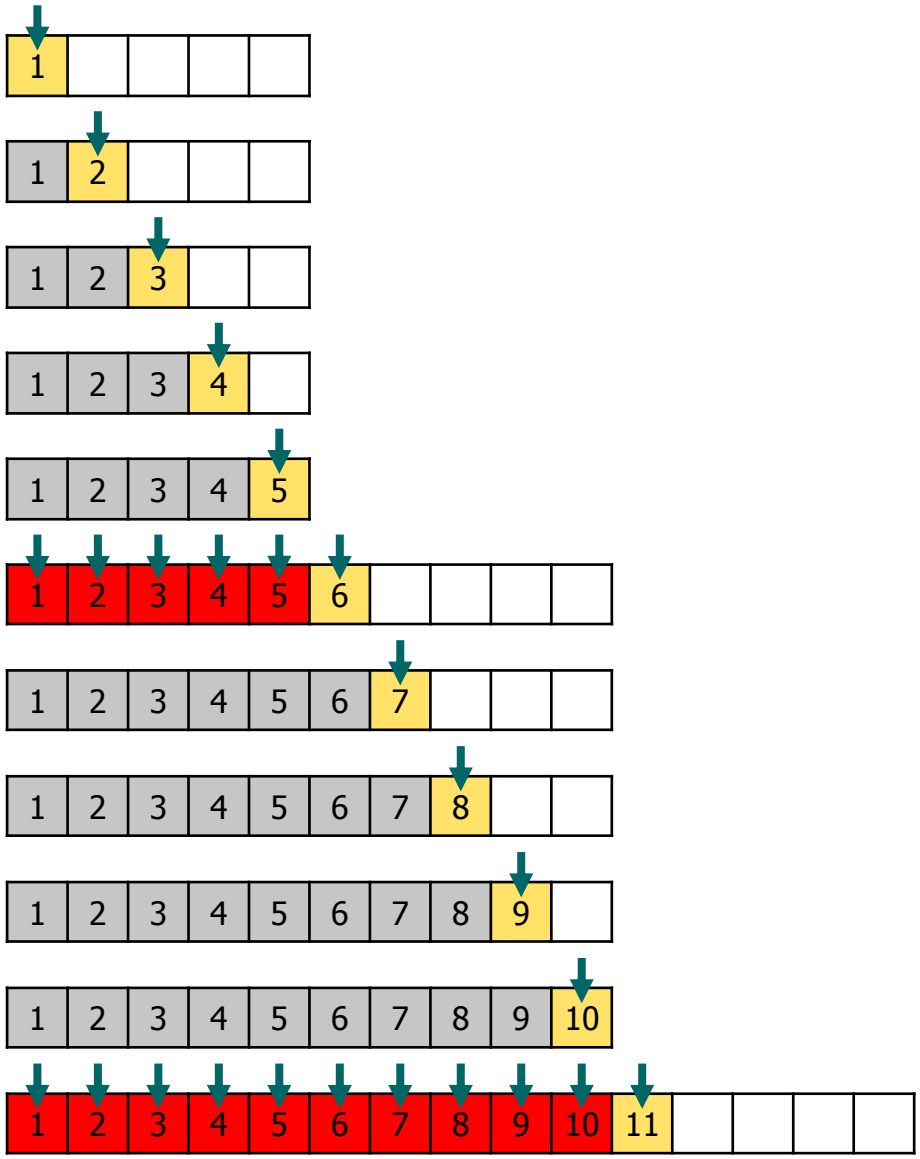
```
public void append(int item) {
  // grow the capacity if needed
  if (nofElements + 1 > this.getCapacity()) {
    int newCapacity = this.getCapacity() + 100;
    int[] newData = new int[newCapacity];
    for (int i = 0; i < nofElements; i++) { newData[i] = data[i]; }
    this.data = newData;
  }
  // append item
  this.data[nofElements] = item;
  nofElements++;
}
```

Version 2: Jedes C. Mal vergrößern (3/5)



- Die Laufzeitkurve ist immer noch quadratisch, warum?

Version 2: Jedes C. Mal vergrößern (4/5)



O(1)

O(1)

O(1)

O(1)

O(1)

O(1**+5**)

O(1)

O(1)

O(1)

O(1)

O(1**+10**)

Version 2: Jedes C. Mal vergrößern (5/5)

- Analyse
  - Die meisten append Operationen kosten jetzt nur O(1)
  - Aber alle C Schritte kommen die Kosten fürs Umkopieren hinzu: C, 2C, 3C, ... Das macht zusammen:

$$
\begin{aligned}
T(n) &= \sum_{i=1}^{n} A \cdot 1 + \sum_{i=1}^{n/C} A \cdot i \cdot C \\
&= A \cdot n + A \cdot C \cdot \sum_{i=1}^{n/C} i \\
&= A \cdot n + A \cdot C \cdot \frac{\frac{n^2}{C^2} + \frac{n}{C}}{2} \\
&= An + \frac{A}{2C} n^2 + \frac{A}{2} n \qquad = \mathcal{O}(n^2)
\end{aligned}
$$

  - Der Faktor vor dem n^2 ist kleiner geworden, aber immer noch quadratische Laufzeit. Haben Sie eine bessere Idee?

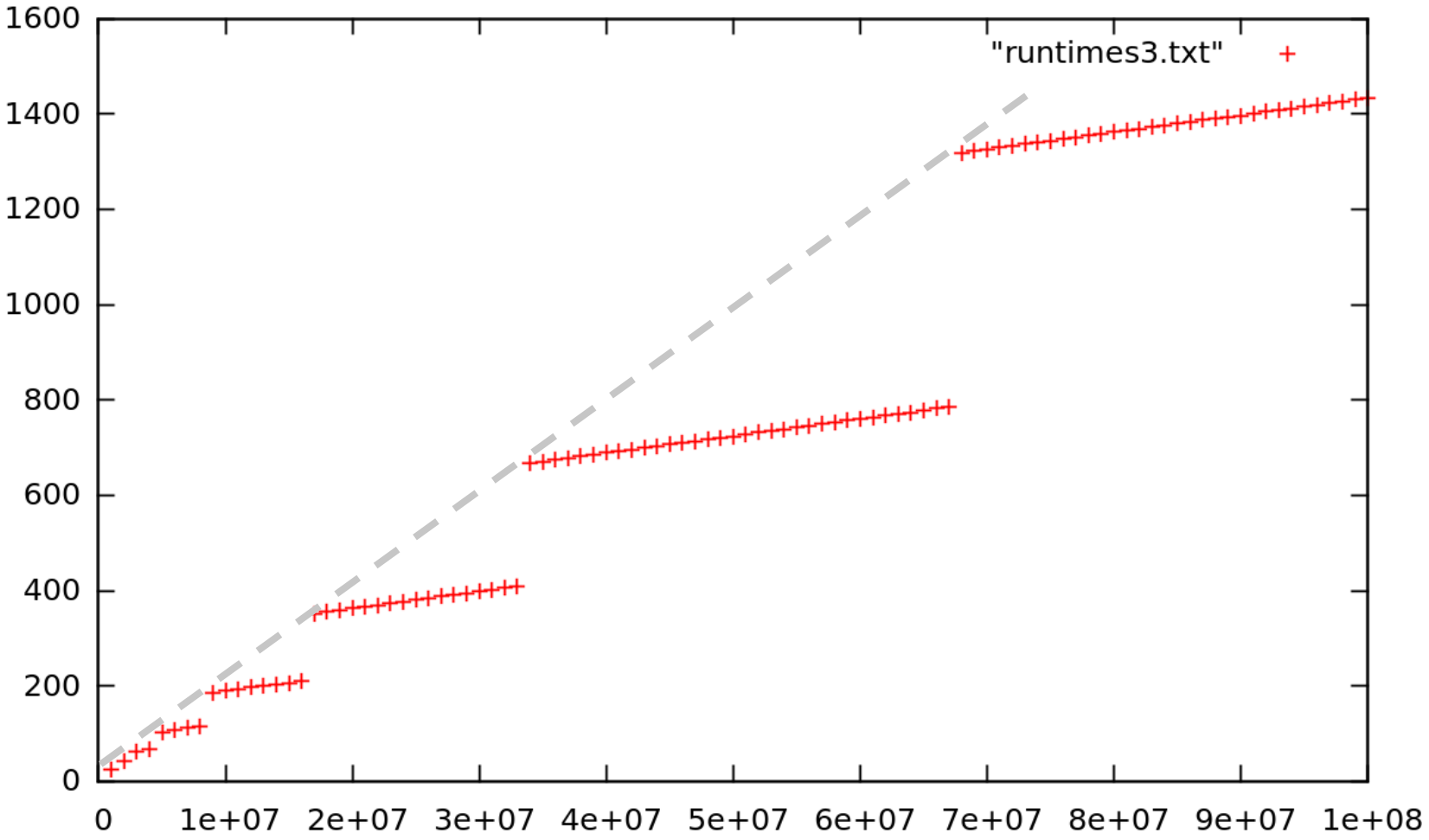
Version 3: Größe immer verdoppeln (1/4)

UNI FREIBURG

- Die "richtige" Vergrößerungsstrategie
  - Idee: Größe immer verdoppeln

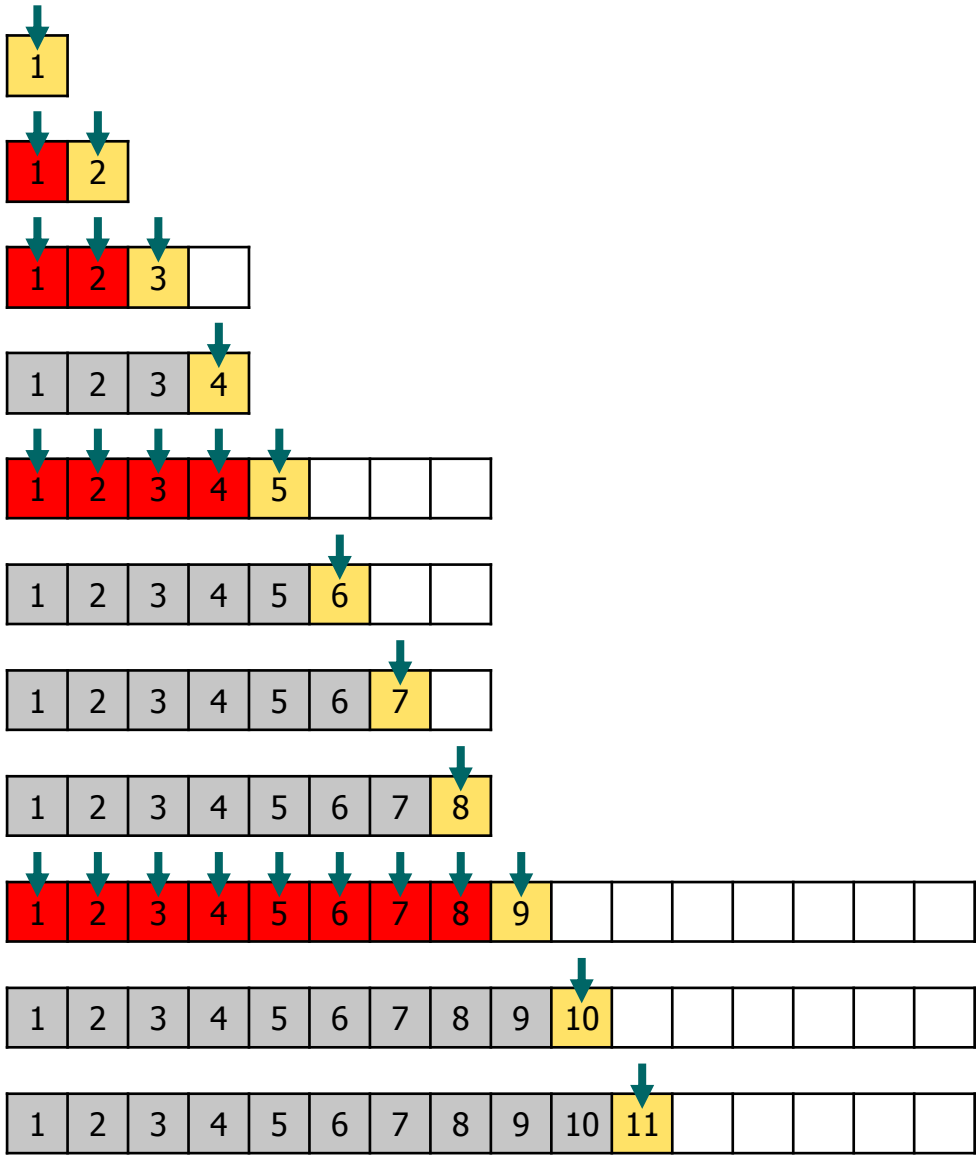
```
public void append(int item) {
  // grow the capacity if needed
  if (nofElements + 1 > this.getCapacity()) {
    int newCapacity = this.getCapacity() * 2;
    if (newCapacity == 0) { newCapacity = 1; }
    int[] newData = new int[newCapacity];
    for (int i = 0; i < nofElements; i++) { newData[i] = data[i]; }
    this.data = newData;
  }
  // append item
  this.data[nofElements] = item;
  nofElements++;
}
```

Version 3: Größe immer verdoppeln (2/4)



- Jetzt sieht die Laufzeitkurze linear aus (mit Sprüngen)

Version 3: Größe immer verdoppeln (3/4)



O(1)

O(1**+1**)

O(1**+2**)

O(1)

O(1**+4**)

O(1)

O(1)

O(1)

O(1**+8**)

O(1)

O(1)

Version 3: Größe immer verdoppeln (4/4)

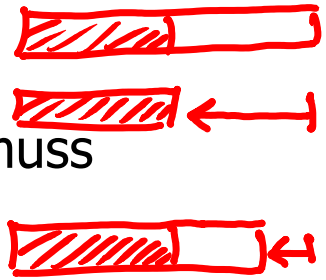
- Analyse
  - Jetzt kosten noch mehr appends nur O(1)
  - Alle 2^i Schritte kommen die Kosten $A \cdot 2^i$ hinzu (für $i = 0,1,2,\ldots,k$ mit $k = \text{floor}(\log_2 (n-1))$)
  - Das macht zusammen:

$$
\begin{aligned}
T(n) = n \cdot A + A \cdot \underbrace{\sum_{i=0}^{k} 2^i}_{=1+2+4+8+\cdots} &= n \cdot A + A \cdot (2^{k+1} - 1) \\
&\leq n \cdot A + A \cdot 2^{k+1} \\
&= n \cdot A + 2A \cdot 2^k \\
&\leq n \cdot A + 2A \cdot n \\
&= 3A \cdot n \\
&= \mathcal{O}(n)
\end{aligned}
$$

Dynamische Felder — Verkleinern

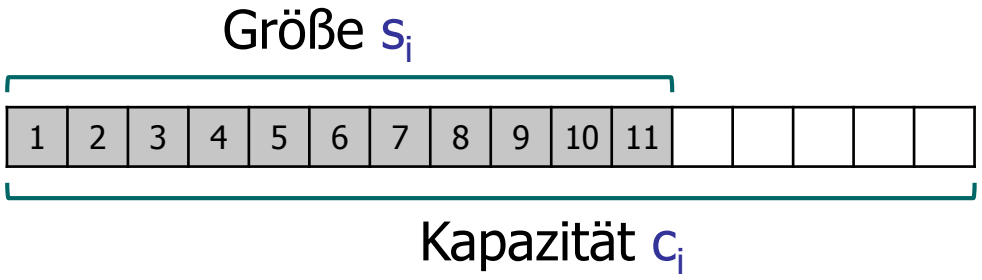
UNI FREIBURG

- Was machen wir wenn Element entfernt werden
  - Analog zum Vergrößern, könnten wir das Feld auf die Hälfte verkleinern wenn es nur noch halbvoll ist
  - Aber Achtung, wenn man danach ein append macht muss man es gleich wieder vergrößern
  - Deswegen lassen wir etwas Luft beim Verkleinern
    - z.B. auf 75% verkleinern wenn nur noch halbvoll



- Analyse
  - Jetzt wird's schwierig, warum?
  - Wenn wir eine beliebige Folge von append und remove Operationen haben, können wir nicht mehr so leicht vorhersagen, wann realloziert werden muss

Amortisierte Analyse 1/7



- Notation
  - Gegeben n Operationen $O_1, \ldots, O_n$
  - Sei s_i die Größe des Feldes nach Operation O_i $(s_0 := 0)$
  - Sei c_i die Kapazität des Feldes nach Operation O_i $(c_0 := 0)$
  - Sei $cost(O_i)$ die Zeit für Operation O_i (unser T_i von vorher)
    - wir nehmen an die ist $\leq A \cdot s_i$ falls Reallokation nötig
    - und ansonsten $\leq A \cdot 1$
    - für irgendeine Konstante A unabhängig von n

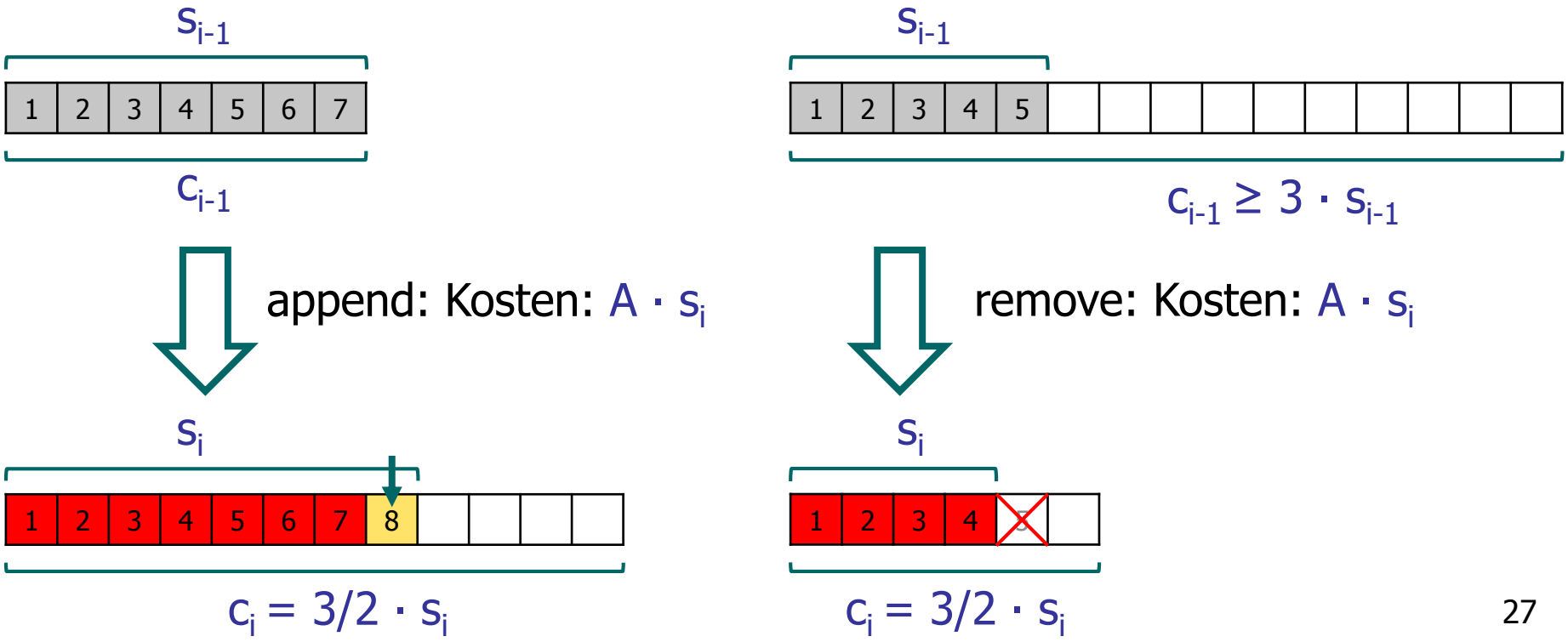
Beispiel

(kein echtes Beispiel; nur zur Illustration der Variablen)

Operationen			Feldgröße	Kapazität	Kosten
O_1	append	Realloc.	s_1	c_1	$A \cdot s_1$
O_2	append		s_2	c_2	$A \cdot 1$
O_3	append		s_3	c_3	$A \cdot 1$
O_4	remove		s_4	c_4	$A \cdot 1$
O_5	remove	Realloc.	s_5	c_5	$A \cdot s_5$
O_6	append		s_6	c_6	$A \cdot 1$
O_7	remove		s_7	c_7	$A \cdot 1$
O_8	append		s_8	c_8	$A \cdot 1$
O_9	append	Realloc.	s_9	c_9	$A \cdot s_9$
O_{10}	append		s_{10}	c_{10}	$A \cdot 1$
...	...				
O_n	append		s_n	c_n	$A \cdot 1$

Amortisierte Analyse 2/7

- Wir analysieren folgende Implementierungsversion
  - Falls O_i append: vergrößern genau dann wenn $s_{i-1} = c_{i-1}$
  - Falls O_i remove: verkleinern genau dann wenn $3 \cdot s_{i-1} \leq c_{i-1}$
  - In beiden Fällen sorgen wir dafür, dass danach $c_i = 3/2 \cdot s_i$



Amortisierte Analyse 3/7

UNI FREIBURG

- Beweisidee
  - Teuer sind nur die Operationen, wo realloziert werden muss
  - Wenn gerade realloziert wurde, dauert es eine Weile, bis wieder realloziert werden muss
  - Anders gesagt: nach einer teuren Operation kommt eine ganze Reihe billiger Operationen
  - **Idee für den Beweis:** wenn nach einer Operation die X gekostet hat, X Operationen kommen, die alle nur 1 kosten, sind die Gesamtkosten bei n Operationen höchstens $2 \cdot n$

Amortisierte Analyse 4/7

Operationen			Feldgröße	Kapazität	Kosten
O_1	append	Realloc.	s_1	c_1	**$A \cdot s_1$**
O_2	append		s_2	c_2	$A \cdot 1$
O_3	append		s_3	c_3	$A \cdot 1$
O_4	remove		s_4	c_4	$A \cdot 1$
O_5	remove	Realloc.	s_5	c_5	**$A \cdot s_5$**
O_6	append		s_6	c_6	$A \cdot 1$
O_7	remove		s_7	c_7	$A \cdot 1$
O_8	append		s_8	c_8	$A \cdot 1$
O_9	append	Realloc.	s_9	c_9	**$A \cdot s_9$**
O_{10}	append		s_{10}	c_{10}	$A \cdot 1$
...	...				
O_n	append		s_n	c_n	$A \cdot 1$

Lemma („Hilfssatz"):
Mindestabstand: $s_1 / 2$

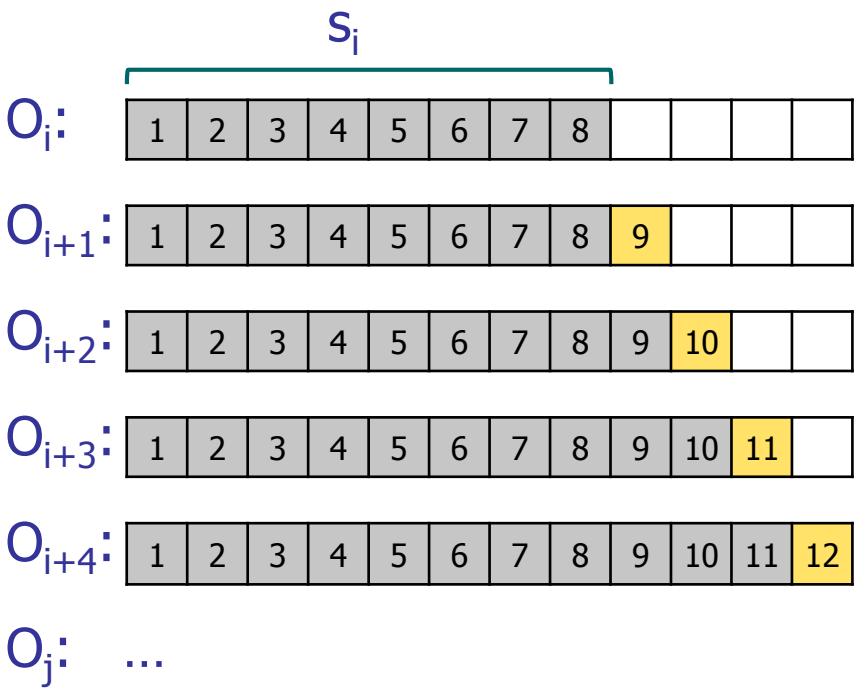
Mindestabstand: $s_5 / 2$

Korollar („(triviale) Schlussfolgerung"):

Gesamtkosten $\leq 4A \cdot n$ ← **$\leq 4A \cdot n$**

- Formal beweisen wir Folgendes
  - Lemma: Wenn bei O_i eine Reallokation stattfindet und dann erst wieder bei O_j, dann ist $j - i > s_i / 2$
  - Korollar: $cost(O_1) + ... + cost(O_n) \leq 4A \cdot n$

Fall 1: s_i / 2 appends



lineare Kosten: $A \cdot s_i$ (Reallokation)

konstante Kosten: $A \cdot 1$

konstante Kosten: $A \cdot 1$

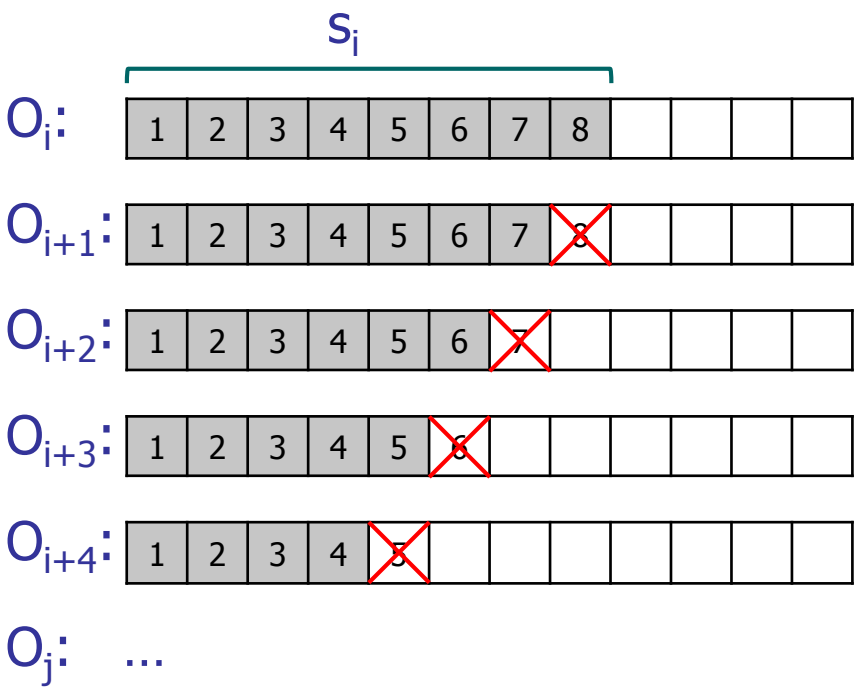
konstante Kosten: $A \cdot 1$

konstante Kosten: $A \cdot 1$

} $s_i/2$ mal

(früheste nächste Reallokation)

Fall 2: s_i / 2 removes



lineare Kosten: $A \cdot s_i$ (Reallokation)

konstante Kosten: $A \cdot 1$

konstante Kosten: $A \cdot 1$

konstante Kosten: $A \cdot 1$

konstante Kosten: $A \cdot 1$

$s_i/2$ mal

(früheste nächste Reallokation)

Amortisierte Analyse 5/7

- Beweis des Lemmas

  [Wenn bei O_i eine Reallokation stattfindet und dann erst wieder bei O_j, dann ist $j - i > s_i/2$]

  – Nach O_i ist die Kapazität genau $\text{floor}(3/2 \cdot s_i) = c_i$

  – Betrachte eine Operation O_k nach O_i mit $k - i \leq s_i/2$:

  - Fall 1: Seit Reallokation können maximal $\text{floor}(s_i/2)$ Elemente dazugekommen sein:

$$s_k \leq s_i + \lfloor s_i/2 \rfloor = \left\lfloor \frac{3}{2} s_i \right\rfloor = c_i$$

Kapazität ist noch OK, keine Vergrößerung nötig

  - Fall 2: Seit Reallokation können maximal $\text{floor}(s_i/2)$ Elemente weg sein:

$$s_k \geq s_i - \lfloor s_i/2 \rfloor = \lceil s_i/2 \rceil \quad \Rightarrow \quad 3 \cdot s_k \geq \left\lceil \frac{3}{2} s_i \right\rceil \geq \left\lfloor \frac{3}{2} s_i \right\rfloor = c_i$$

Kapazität ist noch OK, keine Verkleinerung nötig

Amortisierte Analyse 6/7

Operationen			Feldgröße	Kapazität	Kosten
O_1	append	Realloc.	s_1	c_1	**$A \cdot s_1$**
O_2	append		s_2	c_2	$A \cdot 1$
O_3	append		s_3	c_3	$A \cdot 1$
O_4	remove		s_4	c_4	$A \cdot 1$
O_5	remove	Realloc.	s_5	c_5	**$A \cdot s_5$**
O_6	append		s_6	c_6	$A \cdot 1$
O_7	remove		s_7	c_7	$A \cdot 1$
O_8	append		s_8	c_8	$A \cdot 1$
O_9	append	Realloc.	s_9	c_9	**$A \cdot s_9$**
O_{10}	append		s_{10}	c_{10}	$A \cdot 1$
...	...				
O_n	append		s_n	c_n	$A \cdot 1$

i_1, i_2, i_3

Lemma:

Mindestabstand: $s_{i1} / 2$

Mindestabstand: $s_{i2} / 2$

- Beweis des Korollars

  [$cost(O_1) + ... + cost(O_n) \leq 4A \cdot n$]

  - Seien die Reallokationen bei $O_{i1}, ..., O_{il}$
  - Die Kosten für alle Reallokationen sind A mal $s_{i1} + ... + s_{il}$
  - Nach Lemma ist $i_2 - i_1 > s_{i1}/2$, und $i_3 - i_2 > s_{i2}/2$ usw.

Amortisierte Analyse 7/7

– Daraus: Aussagen über s_i:

$$\begin{aligned} s_{i_1} &< 2(i_2 - i_1) \\ s_{i_2} &< 2(i_3 - i_2) \\ &\vdots \\ s_{i_{\ell-1}} &< 2(i_\ell - i_{\ell-1}) \\ s_{i_\ell} &\leq n \end{aligned}$$

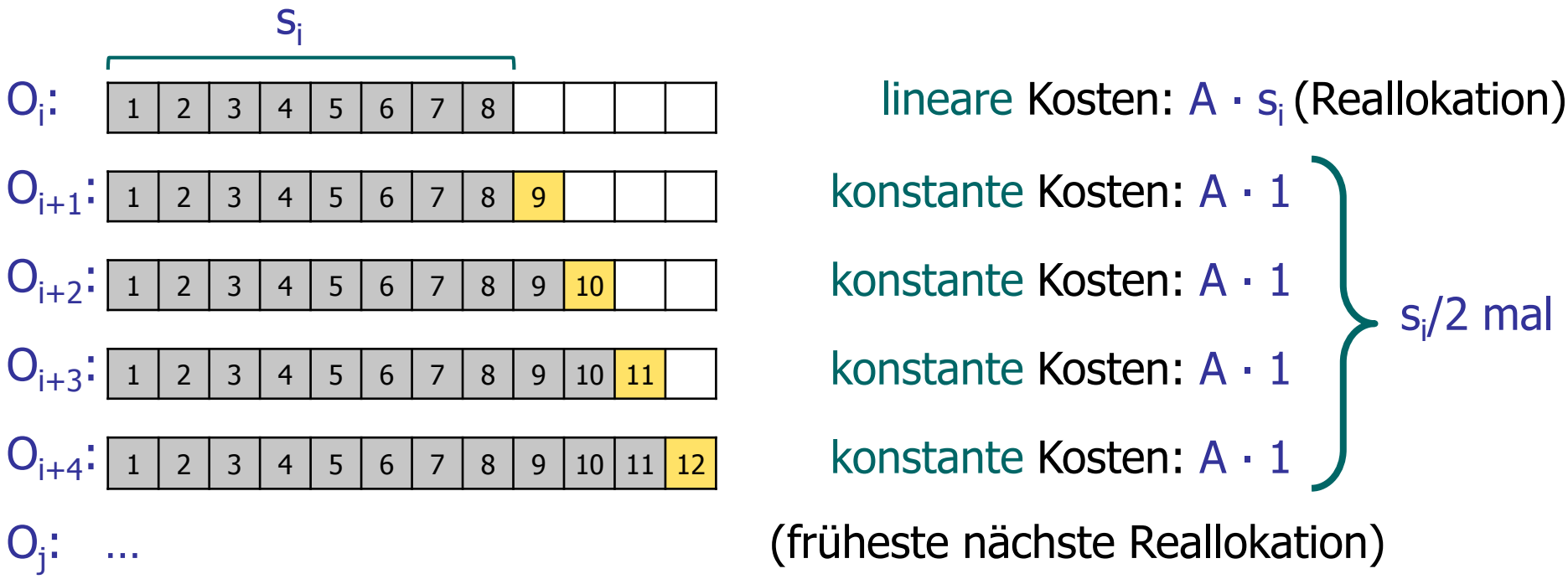
– Die Kosten für alle Reallokationen sind:

$$\begin{aligned} \text{Reallokationskosten} &= A \cdot (s_{i_1} + s_{i_2} + \cdots + s_{i_\ell}) \\ &< A \cdot (2(i_2 - i_1) + 2(i_3 - i_2) + \cdots + 2(i_\ell - i_{\ell-1}) + n) \\ &= A \cdot (2(i_2 - i_1 + i_3 - i_2 + \cdots + i_\ell - i_{\ell-1}) + n) \\ &= A \cdot (2(i_\ell - i_1) + n) \\ &\leq A \cdot (2n + n) \\ &= 3A \cdot n \end{aligned}$$

– Hinzu kommen die jeweils konstanten Kosten für ein „normales“ append oder remove: $\leq A \cdot n$ Also insgesamt: $\leq 4A \cdot n$

Alternativer Beweis:

Laufzeitanalyse für eine lokale worst-case Sequenz



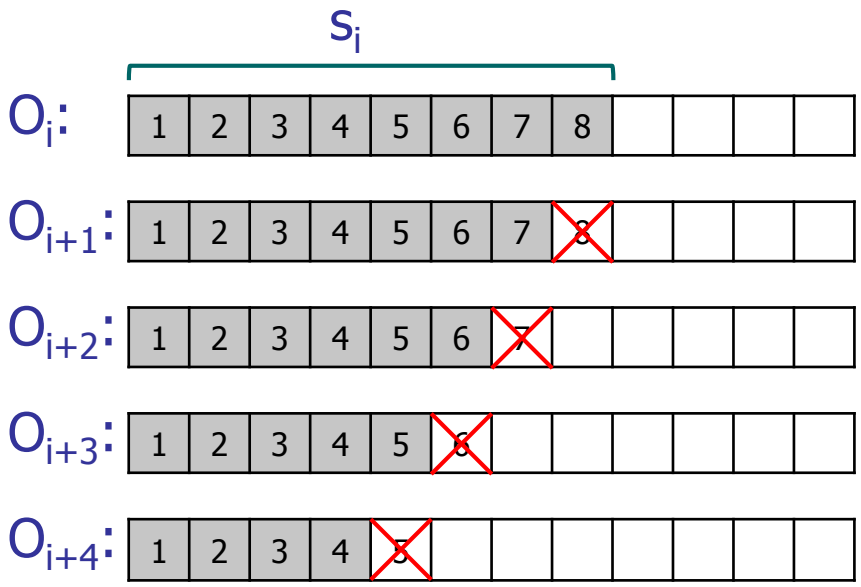
Gesamt: Kosten von $A \cdot 3/2 s_i$ für $s_i/2+1$ Operationen

Kosten pro Operation:

$$\frac{A \cdot \frac{3}{2} s_i}{\frac{1}{2} s_i + 1} \leq \frac{A \cdot \frac{3}{2} s_i}{\frac{1}{2} s_i} = 3A = \text{const.}$$

Alternativer Beweis:

Laufzeitanalyse für eine lokale worst-case Sequenz



lineare Kosten: $A \cdot s_i$ (Reallokation)

konstante Kosten: $A \cdot 1$

konstante Kosten: $A \cdot 1$

konstante Kosten: $A \cdot 1$

konstante Kosten: $A \cdot 1$

$s_i/2$ mal

(früheste nächste Reallokation)

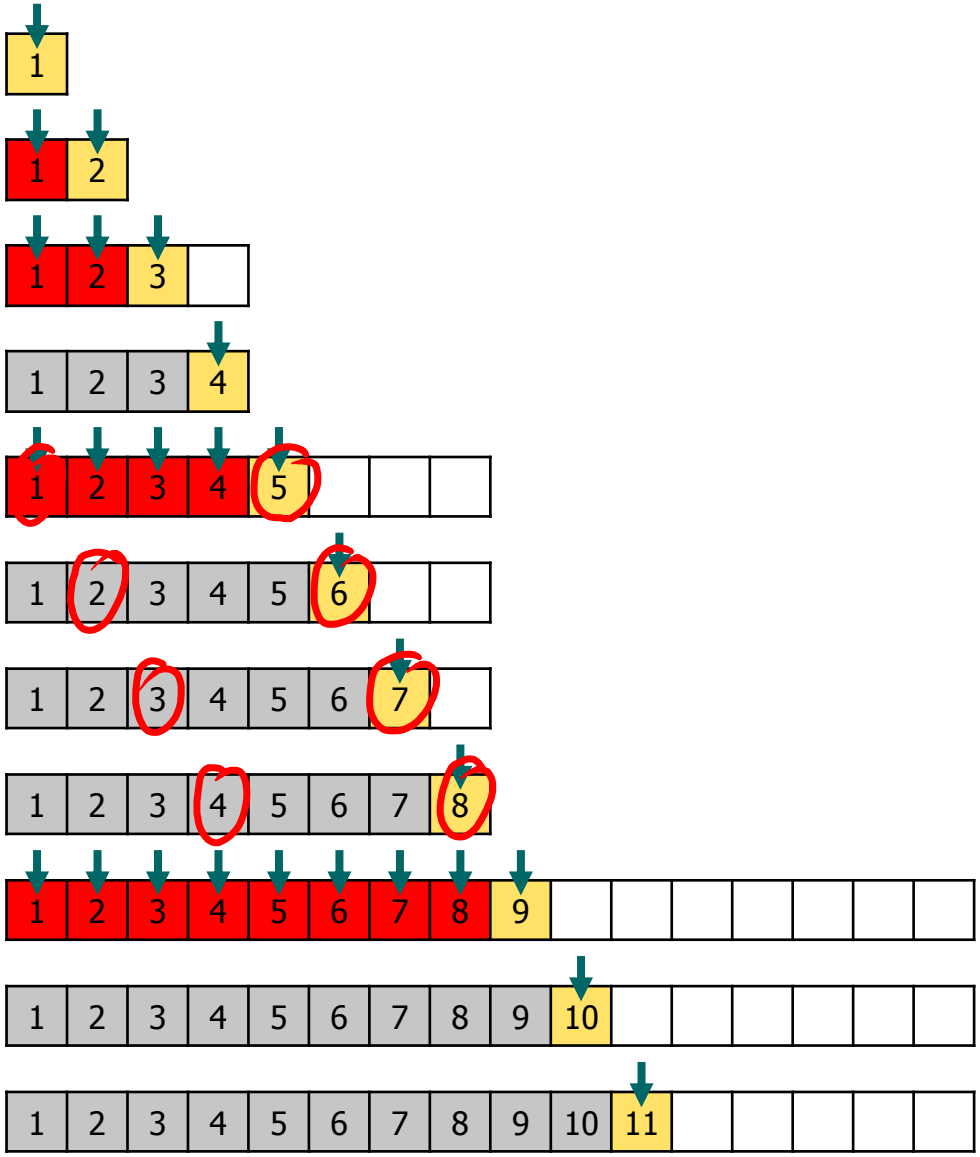
(selbe Gesamtkosten wir vorherige Folie)

Noch ein alternativer Beweis

UNI FREIBURG

- Bankkonto-Paradigma
  - Idee: „erst Sparen, dann ausgeben“
  - Bei jeder Operation mit konst. Kosten, legen wir ein paar Münzen auf ein Konto → immer noch konstante Kosten
  - Wenn dann eine Operation mit linearen Kosten kommt, bezahlen wir sie mit den Münzen vom Konto
  - Für die Strategie mit dem Verdoppeln müssen wir 2 Münzen pro Operation einzahlen

Bankkonto-Paradigma



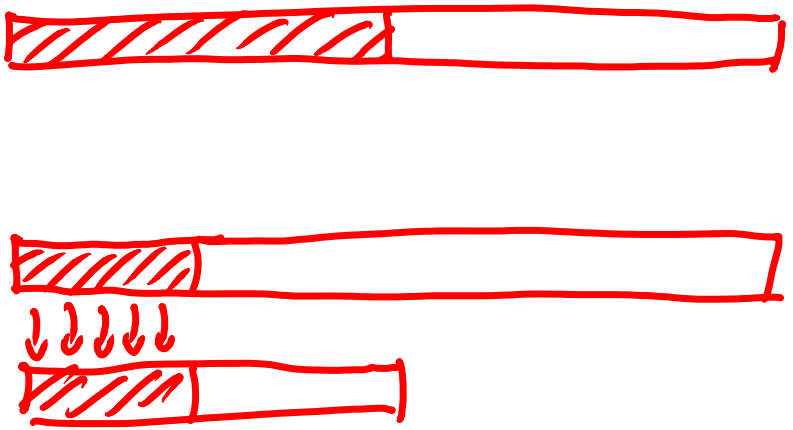
Kosten	Ein/Auszahlung	Kontostand
O(1)	+2	2
O(1**+1**)	+2 -1	3
O(1**+2**)	+2 -2	3
O(1)	+2	5
O(1**+4**)	+2 -4	3
O(1)	+2	5
O(1)	+2	7
O(1)	+2	9
O(1**+8**)	+2 -8	3
O(1)	+2	5
O(1)	+2	7

Bankkonto-Paradigma

UNI FREIBURG

- Warum gerade 2 Münzen pro append()-Operation einzahlen?
  - Jedes eingefügte Element muss später umkopiert werden (erste Münze)
  - und wegen des Vergrößerungsfaktor 2 ist für jedes neue Element schon ein altes im Array, das auch umkopiert werden muss (zweite Münze)

Bankkonto-Paradigma



- Verkleinerungsstrategie: Wenn ¼ voll, dann auf die Hälfte verkleinern. Wieviele Münzen pro remove()-Operation einzahlen?
  - Worst-Case: Das Array wurde gerade vergößert → halbvoll
  - Wir müssen capacity/4 Elemente entfernen, bis zum Umkopieren der verbliebenen capacity/4 Elemente
  - Eine Münze pro Operation reicht aus.

Übungsblatt 7

UNI FREIBURG

- 1: Implementieren Sie remove()
- 2: Plotten Sie ein paar Kurven dazu
- 3: Implementieren sie ein WorstCaseConstantAccessTimeArray (MagicSystem für Java)
- 4: Finden Sie heraus, welche Strategie ArrayList / std::vector nutzen
- 5: Zusatzaufgabe Theorie: Formel für Vergrößerungfaktor bei gewünschten Kosten pro Operation.

Literatur / Links

- Dynamische Felder
  - In Mehlhorn/Sanders:

    3.2 Unbounded Arrays
  - In Cormen/Leiserson/Rivest

    18.4 Dynamic Tables
  - In Wikipedia

    http://en.wikipedia.org/wiki/Dynamic_array
  - In C++ und in Java

    http://www.sgi.com/tech/stl/Vector.html

    http://docs.oracle.com/javase/1.4.2/docs/api/java/util/ArrayList.html

Montag Präsenz-Übung mit Fokus auf Tests und Checkstyle in Java.