

Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 6a, Dienstag, 30. Mai 2017
(Dynamische Felder, Teil 1)

Prof. Dr. Hannah Bast
Lehrstuhl für Algorithmen und Datenstrukturen
Institut für Informatik
Universität Freiburg

Blick über die Vorlesung heute

■ Organisatorisches

- Erfahrungen mit ÜB5

Universelles Hashing

■ Inhalt

- Felder fester Größe
- Dynamische Felder

in Java, C++, Python

Implementierung,
Laufzeitmessung,
Laufzeitanalyse

ÜB6: Implementierung verallgemeinern (auf dem Papier) +
einen "Kilometerzähler" implementieren und analysieren
(mit amortisierter Analyse → siehe Vorlesung 6b morgen)

■ Zusammenfassung / Auszüge

- Hat eine Weile gebraucht, alles zu verstehen
- Relativ zeitaufwändiges Blatt für viele
- Unit Tests waren nicht so einfach
- Histogramm interpretieren: gefühlt um drei Ecken denken
- Aufgabe 3: ohne Histogramme schwer dazu etwas zu sagen
- Durch Aufgabe 3 klarer, wie Universalität funktioniert
- Note to Self: Die Übungsblätter gehen bedeutend leichter, wenn man auf die Professorin hört
- Viel (sehr gutes) Feedback zum Feedback der Tutoren

■ Musterlösung

- Wie immer gibt es eine Musterlösung
- Für eine Klasse wollen wir das gerade mal live codeN

Um zu demonstrieren, dass es wirklich nicht viel Code ist und der Code an sich auch nicht kompliziert ist

■ Plagiate

- Es gibt immer noch einige Plagiate pro Übungsblatt
- Einige (wenige) treiben erheblichen Aufwand, um das Plagiat zu vertuschen
- Wir merken es aber trotzdem
- Verwenden Sie Ihre Energie doch lieber darauf, das Übungsblatt zu machen !

■ Eigenschaften

- Die Größe des Feldes wird bei der Deklaration festgelegt

Vorteil: effizient umzusetzen, weil nur einmal eine feste Menge Speicher alloziert werden muss

Nachteil: bei manchen Anwendungen weiß man vorher nicht, ein wie großes Feld man benötigt

■ Felder fester Größe in Java

<code>int[] numbers = new int[100];</code>	Feld von 100 ints
<code>System.out.println(numbers[12]);</code>	Druckt "0"
<code>String[] strings = new String[10];</code>	Feld von 10 strings
<code>System.out.println(strings[7]);</code>	Druckt "null"
<code>strings[8] = "doof";</code>	Setzt das 9-te Element

In Java werden die Elemente des Feldes bei der Deklaration grundsätzlich **immer** initialisiert

Bei nativen Typen wie `int` mit 0, bei Objekten mit `null`

■ Felder fester Größe in C++

```
int[] numbers = new int[100];  
cout << numbers[12] << endl;
```

Feld von 100 ints
Druckt irgendwas

```
string[] strings = new string[10];  
cout << strings[7] << endl;  
strings[8] = "doof";
```

Feld von 10 strings
Druckt leeren string
Setzt das 9-te Element

In C++ werden die Elemente bei nativen Datentypen wie `int` grundsätzlich **nicht** initialisiert (aus Effizienzgründen)

Bei Objekten wird dagegen grundsätzlich der Default-Konstruktor der Klasse aufgerufen

■ Felder fester Größe in Python

- In **Python** gibt es keine Felder fester Größe

Der Schwerpunkt bei Python liegt auf möglichst einfacher und schneller Entwicklung, nicht auf effizientem Code

- Die "Listen" von Python sind Felder variabler Größe, wobei die Elemente Referenzen auf (beliebige) Objekte sind

In **numpy** gibt es auch effizientere Felder (`numpy.array`), wo die Elemente Zahlen und keine Referenzen sind

Aber auch ein `numpy.array` hat schon variable Größe

■ Begriffsverwirrung

- Felder fester Größe werden auch manchmal **statische Felder** genannt, weil sich ihre Größe nicht ändert
- Das hat aber nichts mit statischer vs. dynamischer Allokation oder dem keyword **static** zu tun:

Bei dem keyword **static** geht es darum, ob Speicherplatz schon vom Compiler (statisch) oder erst zur Laufzeit (dynamisch) zugewiesen wird

- Statisch zugewiesener Platz hat eine feste Größe
- Aber ein Feld fester Größe kann auch dynamisch zugewiesen werden:

```
int[] array = new int[100]; // New array of 100 ints.
```

■ Eigenschaften

- Dynamische Felder können im Laufe ihres Lebens beliebig vergrößert und verkleinert werden

Alternative Bezeichnung daher: Felder variabler Größe

- Das (nicht-triviale) Speichermanagement ist dabei vor der Benutz-Person versteckt
- Zu Beginn kann das Feld entweder leer sein oder schon eine gewisse Größe haben

Wenn man weiß, dass man eine gewisse Größe sowieso braucht, spart man sich dann am Anfang das Management

■ Dynamische Felder in Java

- In Java nimmt man dafür `java.util.ArrayList`

```
ArrayList<String> strings = new ArrayList<String>();  
strings.add("doof");  
strings.add("doofer");  
strings.add("am doofsten");  
System.out.println(strings.get(0));  
strings.remove(strings.length() - 1);
```

*oder Bibliotheken
benutzen wie GUAVA
oder TROVE*

Druckt "doof"

Lösche letztes Element

- **Wichtig:** ArrayList funktioniert **nicht** mit nativen Typen, z.B. `int`, man muss dann Integer nehmen

Das ist für große Datenmengen sehr ineffizient ... wenn die Laufzeit zählt, sollte man dann lieber ein eigenes `ArrayInt` implementieren, das intern mit nativen Feldern realisiert ist

■ Dynamische Felder in C++

- In C++ nimmt man dafür `std::vector` aus der STL

```
std::vector<std::string> strings;  
strings.push_back("doof");  
strings.push_back("doofer");  
strings.push_back("am doofsten");  
cout << strings[0] << endl;  
strings.pop_back();
```

Druckt "doof"

Lösche letztes Element

- **Gut:** vector funktioniert mit allen Typen (nativ und Klassen) und ist genauso effizient wie ein Feld fester Größe

Grund: **templates** ... siehe Programmieren in C++, VL 8

■ Dynamische Felder in Python

- In Python hat man "Listen" und die sind immer dynamisch

```
strings = [];  
strings.append("doof");  
strings.append("doofer");  
strings.append("am doofsten");  
print(strings[0]);  
strings.pop();
```

Druckt "doof"

Lösche letztes Element

- Für native Typen gibt es in Python auch die Klasse `array`

Die ist effizienter als eine Liste, aber auch dynamisch in dem Sinne, dass sie Ihre Größe beliebig ändern kann ... wie gesagt, ein Feld fester Größe gibt es in Python nicht

- Realisierung dynamischer Felder (intuitiv)
 - Intern ein **fixed-size array (FSA)** von ausreichender Größe
 - Wenn Elemente dazu kommen oder entfernt werden:
 - Schauen:** passt die Größe des internen FSA noch?
 - Falls nein:** erzeuge ein neues FSA passender Größe und kopiere die Elemente vom alten in das neue FSA
 - Diesen Vorgang nennt man Reallokation**
 - Das implementieren wir jetzt zusammen
 - Erst mal **pushBack** neues Element anhängen
 - Dann auch **popBack** letztes Element entfernen

■ Vergrößerungsstrategie 1

- Wir vergrößern das Feld nach jedem `push_back` ... und machen es dabei aber immer nur genau **um eins** größer

Beobachtung: akkumulierte Laufzeit sieht quadratisch aus

■ Analyse

- Sei T_i die Laufzeit für das i -te `push_back`
- Dann ist $T_i \geq A \cdot i$ für irgendeine Konstante A

Bei einer Reallokation müssen alle Elemente kopiert werden

$$\sum_{i=1}^n T_i \geq \sum_{i=1}^n A \cdot i = A \cdot \sum_{i=1}^n i = A \cdot \frac{1}{2} n(n+1) = \Theta(n^2)$$

Dynamische Felder 7/9

*1/C besser, aber immer noch
QUADRATISCH*

■ Vergrößerungsstrategie 2

- Wie vorher, aber jetzt bei jedem Vergrößern **um C größer**, für ein festes C, zum Beispiel **C = 100** oder **C = 1000**

Beobachtung: akkumulierte Laufzeit immer noch quadratisch

■ Analyse

- Sei wieder T_i die Laufzeit für das i -te **push_back**

- Dann sind die meisten T_i jetzt **O(1)**

*Annahme:
n Vielfaches von C*

C = 1000 : 1000, 2000, 3000, ...

- Aber für $i = C, 2C, 3C, \dots$ ist nach wie vor $T_i \geq A \cdot i$

$$\sum_{i=1}^n T_i \geq \sum_{j=1}^{n/C} \underbrace{T_{Cj}}_{\geq A \cdot C \cdot j} \geq \sum_{j=1}^{n/C} A \cdot C \cdot j = A \cdot C \cdot \sum_{j=1}^{n/C} j$$

$$\rightarrow = A \cdot C \cdot \frac{1}{2} \underbrace{n/C \cdot (n/C + 1)}_{\geq n/C} \geq A \cdot C \cdot \frac{1}{2} \cdot \frac{n^2}{C^2} = \frac{1}{2} A \cdot \frac{1}{C} \cdot n^2$$

■ Vergrößerungsstrategie 3

- Wie vorher, aber jetzt machen wir bei jedem Vergrößern das Feld genau **doppelt so groß** wie vorher

Beobachtung: jetzt sieht die Laufzeitkurve linear aus

■ Analyse

Annahme: $n = 2^z$ Zw. potenz

- Jetzt Reallokationen nur noch bei $i = 1, 2, 4, 8, 16, \dots$

Bei den Reallokationen $T_i \leq A \cdot i$, sonst $T_i \leq A$

$$\sum_{i=1}^n T_i \leq \sum_{i=1}^n A + A(1 + 2 + 4 + \dots + 2^z)$$

$\underbrace{2^{z+1} - 1}_{2n}$

$$\leq A \cdot n + A \cdot (2n - 1)$$

$$\leq 3 \cdot A \cdot n = O(n) \quad \square$$

$$\begin{aligned} 1 + 2 + 4 + 8 &= 15 \\ &= 16 - 1 \\ &= 2^4 - 1 \end{aligned}$$

■ Entfernen von Elementen

- Analog zum Vergrößern, könnten wir das Feld auf die Hälfte verkleinern wenn es nur noch halbvoll ist

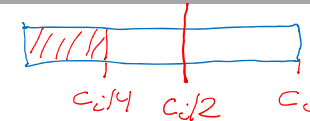
Aber: wenn man danach zwei pushBack macht, muss man das Feld gleich wieder vergrößern

Und: wenn man danach zwei popBack macht, muss man das Feld gleich wieder verkleinern

- Deswegen machen wir es (erst mal) so:

Wenn Feld **ganz voll** → Größe **verdoppeln**

Wenn Feld **viertel voll** → Größe **halbieren**



- Mehlhorn / Sanders
 - Kapitel 3: Representing Sequences by Arrays ...
- Plotly
 - <https://plot.ly>
- Doof
 - <http://de.wiktionary.org/wiki/doof>