Algorithmen und Datenstrukturen (ESE)
Entwurf, Analyse und Umsetzung von Algorithmen (IEMS)
WS 2012 / 2013

Vorlesung 7, Dienstag 4. Dezember 2012
(Dynamische Felder, Amortisierte Analyse)

Björn Buchhold
i.V. für Prof. Dr. Hannah Bast
Lehrstuhl für Algorithmen und Datenstrukturen
Institut für Informatik
Universität Freiburg

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Wer ist das da vorne?
  - 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

Erfahrungen mit dem Ü6 (Priority Queues)

UNI FREIBURG

- Zusammenfassung / Auszüge Stand 4. Dezember 16:09
  - Zeit ~4-8h, teilweise deutlich mehr
  - Mehr Grundlagen in den Programmiersprachen wären hilfreich
  - Checkstyle Fehler!
  - Die changeKey Methode hat ein Schwierigkeiten bereitet, wenn man Kopien in der PQ gespeichert hat
  - Es gab zusätzliche Schwierigkeiten mit C++.
  - Die Tests waren hilfreich um Fehler bei Grenzfällen (leere PQ) zu finden.
  - Die Vorlesung war verwirrend.
  - Ich weiß jetzt wie viele Millimeter ein Lichtjahr hat, danke!

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.
printf("%d\n", numbers[12]);  //  Prints random number.
string[] strings = new string[10];  // Pointer to 10 strings.
printf("%s\n", strings[7].c_str());  // 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 implementieren wir jetzt erstmal zusammen
    - Vorlesung: nur append (= neues Element anhängen)
    - Übungsblatt: append **und** remove (= letztes El. entfernen)

Version 1

UNI FREIBURG

- Einfachste Vergrößerungsstrategie
  - Wir vergrößern das Feld nach jedem append
  - Und machen es immer genauso groß, wie wir es brauchen
  - Die Laufzeitkurze zeigt quadratisches Verhalten, warum?
- 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 is $T_i \geq A \cdot i$ für irgendeine Konstante A
    - weil wir i Elemente reallozieren (umkopieren) müssen
  - Das macht zusammen:

$$T(n) = \sum_{i=1}^{n} T_i = \sum_{i=1}^{n} A \cdot i = A \cdot \sum_{i=1}^{n} i = A \cdot \frac{n^2+n}{2} > A \cdot \frac{n^2}{2} \in \Omega(n^2)$$

$$1+2+3+4+\ldots = \frac{n^2+n}{2}$$

Version 2

UNI FREIBURG

- Eine etwas vorausschauendere Vergrößer.strategie
  - Idee: beim Vergrößern zusätzlichen Platz lassen
  - Aber wieviel?
  - Lassen wir erstmal Platz für C zusätzliche Elemente, für ein beliebiges festes C, zum Beispiel C = 100 oder C = 1000
  - Die Laufzeitkurve ist immer noch quadratisch, warum?
- Analyse
  - Die meisten append Operationen kosten jetzt nur O(1)
  - Aber für i = C, 2C, 3C, ... ist nach wie vor $T_i \geq A \cdot i$
  - Das macht zusammen:

$$T(n) = \sum_{i=1}^{n/C} T_{i \cdot C} = A \cdot \sum_{i=1}^{n/C} i \cdot C = A \cdot C \cdot \frac{1}{2}\left(\frac{n^2}{C^2} + \frac{n}{C}\right) \geq \frac{A}{2C} n^2$$

$$\in \Omega(n^2)$$

Version 3

- Die "richtige" Vergrößerungsstrategie
  - Idee: immer doppelt so viel vergrößern wie nötig
  - Jetzt sieht die Laufzeitkurze linear aus (mit Sprüngen)
- Analyse
  - Jetzt kosten noch mehr appends nur $O(1) = B$
  - Für $i = 1, 2, 4, 8, 16 \ldots$ ist $T_i = B + T_i'$ und $T_i' \leq A \cdot i$ für irgendein A
  - Das macht zusammen:

$$T(n) = n \cdot B + \sum_{i=1}^{k} T_{2^i} = n \cdot B + A \cdot (2^{k+1} - 1) \leq n \cdot B + A \cdot 2^{k+1}$$

$$= n \cdot B + 2 \cdot 2^k A = n \cdot B + 2 \cdot n \cdot A$$

$$= (2A + B) \cdot n \leq O(n)$$

$k = \log_2 n$

$1 + 2 + 4 + 8 + \ldots = 2^{k+1} - 1$

$2^{\log n} = n$

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 1/4

UNI FREIBURG

- 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 $\leq B$
    - für irgendwelche Zahlen A und B unabhängig von n
- 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 2/4

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 $2 \cdot n$
- 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: $\text{cost}(O_1) + \ldots + \text{cost}(O_n) \leq (3A + B) \cdot n \simeq$ 

Amortisierte Analyse 3/4

UNI FREIBURG

- 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_j nach O_i mit $j - i \leq s_i/2$

1. Maximal $\lfloor s_i/2 \rfloor$ Elemente dazugekommen sein:

$s_j \leq s_i + \lfloor s_i/2 \rfloor = \lfloor \frac{3}{2} s_i \rfloor \Rightarrow s_j \leq \lfloor \frac{3}{2} s_i \rfloor \overset{c_i}{=} \Rightarrow$ muss nicht vergrößern

2. Maximal $\lfloor s_i/2 \rfloor$ Elemente weg

$s_j \geq s_i - \lfloor s_i/2 \rfloor = \lceil s_i/2 \rceil \Rightarrow 3 \cdot s_j = \lceil \frac{3}{2} s_i \rceil \geq \lfloor \frac{3}{2} s_i \rfloor = c_i$

$\Rightarrow$ muss nicht schrumpfen

Amortisierte Analyse 4/4

UNI FREIBURG

- Beweis des Korollars

  [$cost(O_1) + \ldots + cost(O_n) \leq (3A + B) \cdot n$]

  – Seien die Reallokationen bei $O_{i1}, \ldots, O_{il}$

  – Die Kosten dafür sind A mal $s_{i1} + \ldots + s_{il}$

  – Nach Lemma ist $i2 > i1 + s_{i1}/2$, $i3 > i2 + s_{i2}/2$ usw.

$s_{i1} < 2 \cdot (i_2 - i_1)$; $s_{i2} < 2 \cdot (i_3 - i_2)$; $\ldots s_{il-1} < 2 \cdot (i_l - i_{l-1})$

$s_{il} \leq n$

$2 \cdot (i_2 - i_1) + 2(i_3 - i_2) + \ldots + 2(i_l - i_{l-1}) + n$

$2 \cdot (i_2 - i_1 + i_3 - i_2 + \ldots i_l - i_{l-1}) + n$

$2 \cdot (i_l - i_1) + n < 2n + n = 3n$ ☐

Literatur / Links

UNI FREIBURG

- 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