Technische Universität München
Fakultät für Informatik
Lehrstuhl für Effiziente Algorithmen
Dr. Hanjo Täubig
Jeremias Weihmann

Sommersemester 2014
Übungsblatt 3
5. Mai 2014

Grundlagen: Algorithmen und Datenstrukturen

Abgabetermin: Jeweilige Tutorübung in der Woche vom 12. bis 16. Mai

Tutoraufgabe 1

Diese Tutoraufgabe ist ein Kurztutorial für die Bankkonto-Methode, die in der Vorlesung für dynamische Arrays verwendet wurde. Die zweite Tutoraufgabe veranschaulicht die Theorie an verallgemeinerten dynamischen Arrays.

Sei S eine Menge von Operationen, und bezeichne $T(\sigma)$ (eine obere Schranke für) die Laufzeit einer Operation $\sigma \in S$ (diese Laufzeit kann vom aktuellen Zustand des Objekts, auf dem die Operation wirkt, sowie von Argumenten abhängen). Analog dazu bezeichne $T(\sigma_1, \sigma_2, \ldots, \sigma_m) := \sum_{i=1}^{m} T(\sigma_i)$ (die korrespondierende obere Schranke für) die Laufzeit der Operationsfolge $(\sigma_1, \sigma_2, \ldots, \sigma_m)$.

Ziel einer amortisierten Analyse ist, eine möglichst gute obere Schranke für $T(\sigma_1, \sigma_2, \ldots, \sigma_m)$ zu finden. Bei der Konto-Methode bestimmen wir dazu eine Funktion $\Delta : S \to \mathbb{R}$, die folgende Eigenschaften besitzt:

(i) Für alle legalen (d.h. ausführbaren) Operationsfolgen $(\sigma_1, \ldots, \sigma_m)$ gilt $\sum_{i=1}^{m} \Delta(\sigma_i) \geq 0$.

(ii) Δ ist möglichst gut gewählt.

Was Eigenschaft (ii) bedeutet, wird später noch spezifiziert. Für $\sigma \in S$ ist $\Delta(\sigma)$ die *Veränderung des Tokenkontos* durch die Operation σ. Wenn $\Delta(\sigma) > 0$, dann zahlt die Operation auf das Konto ein, falls $\Delta(\sigma) < 0$, dann hebt sie vom Konto ab. $A(\sigma) := T(\sigma) + \Delta(\sigma)$ nennen wir dann die *amortisierte Laufzeit* von σ. (In der Vorlesung wird $A(\sigma)$ auch Tokenlaufzeit genannt.) Entsprechend ist $A(\sigma_1, \ldots, \sigma_m) := \sum_{i=1}^{m} A(\sigma_i)$ die amortisierte Laufzeit der Operationsfolge $(\sigma_1, \ldots, \sigma_m)$.

Eigenschaft (i) sagt aus, dass das Tokenkonto nie negativ ist. Die Sinnhaftigkeit eines stets nichtnegativen Kontos ergibt sich aus folgender Beobachtung:

$$A(\sigma_1, \ldots, \sigma_m) = \sum_{i=1}^{m} A(\sigma_i) = \sum_{i=1}^{m} (T(\sigma_i) + \Delta(\sigma_i))$$
$$= T(\sigma_1, \ldots, \sigma_m) + \underbrace{\sum_{i=1}^{m} \Delta(\sigma_i)}_{\geq 0} \geq T(\sigma_1, \ldots, \sigma_m)$$

Die amortisierte Laufzeit der Operationsfolge ist also eine *obere Schranke* für ihre tatsächliche Laufzeit. Jede Tokeneinheit steht für eine gewisse konstante Menge an Laufzeit. Damit wird auch klar, was die (möglicherweise nicht ganzzahlige) Tokenzahl auf dem Konto aussagt: Es ist genau der Wert, um den die amortisierte Laufzeit die durch T gegebene (obere Schranke für die) tatsächliche Laufzeit übersteigt.

Nun zu Eigenschaft (ii). Das wichtigste Ziel ist, dass die amortisierten Laufzeiten $A(\sigma_1, \ldots, \sigma_m)$ von Operationsfolgen asymptotisch möglichst klein sind, damit man gute obere Schranken für die tatsächliche Laufzeit von Operationsfolgen erhält.

Zu diesem Zwecke ist es oft zielführend, Δ so zu wählen, dass $\max_{\sigma \in S}(A(\sigma))$ möglichst klein ist, d.h. die Operation mit der schlechtesten amortisierte Laufzeit soll möglichst geringe amortisierte Laufzeit besitzen. Wenn dieses Ziel erreicht ist, ist $\mathcal{O}(m \cdot \max_{\sigma \in S}(A(\sigma)))$ eine obere Schranke für die asymptotische Laufzeit von Worst-Case-Operationsfolgen (wobei die Länge m der Operationsfolgen die asymptotische Variable ist). Bei einer hinreichend guten Wahl von Δ und Analyse ist diese obere Schranke oft nicht schlecht.

Tutoraufgabe 2

Wir betrachten die Konto-Methode angewandt auf dynamische Arrays für beliebige Werte α und β mit $\alpha > \beta > 1$. Hierbei beschränken wir uns auf den analytisch etwas einfacheren Fall, dass $\alpha, \beta \in \mathbb{N}$. Die Analyse für nicht ganzzahlige α und β ist ähnlich. [1]

Man erinnere sich, dass dynamische Arrays über die Methoden `pushBack` (= Einfügen eines Elements am Ende der „Liste“) und `popBack` (Löschen des letzten Elements der „Liste“) verwaltet werden. Im Folgenden bezeichne n stets die aktuelle Anzahl der Elemente im Array, und w die Größe des Arrays. Unter bestimmten Voraussetzungen rufen `pushBack` und `popBack` die Methode `reallocate` als Untermethode auf:

Wenn bei `pushBack` *vor* der Einfügung des Elements das Array voll ist (d.h. $n = w$), dann wird die Methode `reallocate` aufgerufen, die ein neues Array der Größe βn anlegt und alle alten Elemente in das neue Array kopiert. Anschließend wird das neue Element eingefügt.

Wenn bei `popBack` nach der Löschung des letzten Elements der Füllstand des Arrays nur noch maximal $1/\alpha$ beträgt (d.h. $\alpha \cdot n \leq w$), dann wird die Funktion `reallocate` aufgerufen, die ein neues Array der Größe βn anlegt und alle Elemente in das neue Array kopiert. Wir fassen `reallocate` als eigenständige Methode auf.

Es ist nicht schwierig zu sehen, dass die tatsächliche Laufzeit von `pushBack` und `popBack` konstant ist (d.h. durch eine Konstante nach oben beschränkt), und die Laufzeit von `reallocate` durch $\mathcal{O}(1) + c \cdot n$ beschränkt ist, wobei c eine Konstante ist und n die Zahl der kopierten Elemente. Wie in der Vorlesung ist es legitim, $c = 1$ zu setzen, indem wir annehmen, dass wir die Laufzeit in einer Einheit messen, die gerade der Laufzeit einer

[1] In einem solchen Fall muss die Größe des neuen Arrays mithilfe der Aufrundungsfunktion als $\lceil \beta \cdot n \rceil$ definieren werden, was eine entsprechende Anpassung der Funktion Δ erfordert und die Analyse etwas komplizierter macht. In der Literatur werden in solchen Fällen eigentlich notwendige Auf- oder Abrundungen nicht selten einfach weggelassen, um die Analyse zu vereinfachen. Die Analyse ist dann zwar nicht mehr absolut präzise, aber zumindest größenordnungsmäßig „gut genug“ und man erkennt die dahinterstehende Idee. Der Leser ist dann gefordert, die Details selber zu ergänzen. Auf einen Blick zu erkennen, wann solche Vereinfachungen möglich sind, erfordert etwas Erfahrung.

einzelnen Kopieroperation entspricht. Wir erhalten damit:

$$\begin{aligned} T(\texttt{pushBack}) &\in \mathcal{O}(1) \\ T(\texttt{popBack}) &\in \mathcal{O}(1) \\ T(\texttt{reallocate}) &= \mathcal{O}(1) + n, \text{ wobei } n = \text{Anzahl der kopierten Elemente} \end{aligned}$$

Ziel dieser Aufgabe ist der Nachweis mit einer amortisierten Analyse, dass die Laufzeit von Operationen der Länge m auf einem zu Beginn leeren dynamischen Array in $\mathcal{O}(m)$ liegt (zu Beginn hat das Array Größe 1).

Dafür legen wir fest, dass `pushBack` $\beta/(\beta-1)$ Token auf das Konto einzahlt, `popBack` $\beta/(\alpha-\beta)$ Token einzahlt, und `reallocate` n Token vom Konto abhebt, wenn n Elemente kopiert werden, also:

$$\begin{aligned} \Delta(\texttt{pushBack}) &= \beta/(\beta-1) \\ \Delta(\texttt{popBack}) &= \beta/(\alpha-\beta) \\ \Delta(\texttt{reallocate}) &= -n, \text{ wobei } n = \text{Anzahl der kopierten Elemente} \end{aligned}$$

(a) Zeigen Sie, dass dieses Amortisationsschema zulässig ist, indem Sie zeigen, dass das Tokenkonto zu jedem Zeitpunkt nichtnegativ ist.

Hinweis: Bezeichne n_1 die Zahl der Elemente *unmittelbar* nach einem `reallocate` (im Falle von `pushBack` also noch vor der Einfügung des neuen Elements). Machen Sie sich klar, dass das Array *zu diesem Zeitpunkt* Größe $w_1 := \beta \cdot n_1$ hat, und $w_1 - n_1$ Positionen frei sind. Das nächste `reallocate` wird erst dann aufgerufen, wenn für die Zahl n der Elemente entweder $n = w_1$ oder $\alpha n \leq w_1$ gilt.

(b) Zeigen Sie, dass unter diesem Amortisationsschema die amortisierte Laufzeit jeder Operation in $\mathcal{O}(1)$ liegt, und folgern Sie, dass die Worst-Case-Laufzeit für Operationsfolgen der Länge m in $\mathcal{O}(m)$ liegt.

Tutoraufgabe 3

Sei $h : \mathbb{N}_0 \to \mathbb{R}$ eine Funktion mit $\exists n_0 \in \mathbb{N} : \forall n > n_0 : h(n) > 0$. In der Vorlesung haben wir einige Methoden kennengelernt, mit denen wir zeigen können, dass die Worst-Case Laufzeit eines Algorithmus in $\mathcal{O}(h(n))$ liegt. In diesem Kontext stellt sich die Frage, wie man zeigt, dass die Worst-Case Laufzeit eines Algorithmus *nicht* in $\mathcal{O}(h(n))$ liegt. Die Antwort lautet, dass man zeigen muss, dass eine unendliche Familie von Eingabeinstanzen wachsender Größe existiert, sodass die Laufzeit des Algorithmus *bezüglich dieser Familie von Instanzen* in $\omega(h(n))$ liegt.

Dazu betrachten wir exemplarisch das folgende Problem: Gegeben sei eine Folge von natürlichen Zahlen $(x_1, x_2, \ldots, x_n)$. Gefragt ist, ob n eine Quadratzahl ist und ob es eine Zahl gibt, die in dieser Folge (mindestens) zweimal vorkommt.

Wir betrachten den Algorithmus ExistsPairAndIsSquare für dieses Problem. Zeigen Sie, dass die Laufzeit dieses Algorithmus nicht in $\mathcal{O}(n^{3/2})$ liegt.

Algorithmus 1 : ExistsPairAndIsSquare

```
   Input : int[] (x_1, x_2, ..., x_n)
 1 int i := 1
 2 if √n ∉ ℕ then
 3 |  return Nein
 4 while i ≤ n do
 5 |  int j := 1
 6 |  while j ≤ n do
 7 |  |  if x_j = x_i and i ≠ j then
 8 |  |  |  return Ja
 9 |  |  j := j + 1
10 |  i := i + 1
11 return Nein
```

Hausaufgabe 1

Zeigen Sie, dass die folgenden Funktionen f und g bezüglich $\mathcal{O}$-Notation unvergleichbar sind, d.h., zeigen Sie dass $f(n) \notin \mathcal{O}(g(n))$ und $f(n) \notin \Omega(g(n))$ gilt.

$$f(n) = n^2 \quad \text{und} \quad g(n) = \begin{cases} n, & \text{falls n gerade} \\ n^3, & \text{falls n ungerade} \end{cases}$$

Hausaufgabe 2

Ordnen Sie die folgenden Funktionen so an, dass für zwei in der Anordnung aufeinanderfolgende Funktionen f und g gilt: $f(n) \in o(g(n))$ oder $f(n) \in \mathcal{O}(g(n))$. Beweisen Sie Ihre Anordnung. Verwenden Sie gegebenenfalls die Monotonie-, Konvergenz- und Unbeschränktheitseigenschaften elementarer Funktionen ohne Beweis. Dabei sind $c, \epsilon > 0$ von n unabhängige Konstanten.

$$\ln \operatorname{ld} n, \quad n^n, \quad e^{n \operatorname{ld} n}, \quad \sqrt{\ln n}, \quad n!, \quad \epsilon n^c, \quad n^{c+\epsilon}, \quad n^c$$

Hinweis: Verwenden Sie in gegebenenfalls die Rechenregeln aus der Vorlesung, die einen Zusammenhang zwischen dem Wachstumsverhalten von $f(n)$ und $g(n)$ und dem Wachstumsverhalten von $f'(n)$ und $g'(n)$ herstellen. Alternativ können Sie auch eine Regel von L'Hospital verwenden, die besagt:

Seien $f, g : \mathbb{R} \to \mathbb{R}$ zwei Funktionen mit entweder $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = 0$ oder $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$. Dann gilt $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$ sofern der Limes $\lim_{n\to\infty} \frac{f'(n)}{g'(n)}$ existiert.

An vielen Stellen ist es auch hilfreich, die Transitivitätsregeln von Übungsblatt 2 zu verwenden.

Hausaufgabe 3

Sie haben soeben dynamische Arrays mit $\alpha = 4$ und $\beta = 2$ (wie in der Vorlesung) für Ihren Chef implementiert. Nach einer kurzen Begutachtung Ihrer Lösung bittet Sie Ihr Chef, stattdessen $\alpha = 2$ zu verwenden. Er argumentiert, dass es Platzverschwendung sei, wenn das Array erst dann verkleinert wird, wenn bereits dreiviertel des Platzes nicht mehr verwendet wird. Er schlägt vor, das Array bereits dann zu verkleinern, wenn es nur noch halbvoll ist, d.h. wenn nach dem `popBack` $n \leq w/2$ gilt.

Überzeugen Sie ihn, dass dies eine schlechte Idee ist, indem Sie zeigen, dass die Worst-Case-Laufzeit $T(m)$ von Operationsfolgen der Länge m auf einem solchen Array nicht mehr in $\mathcal{O}(m)$ liegt.

Hinweis: Interpretieren Sie eine Operationsfolge als Eingabe eines Algorithmus, der die gegebene Operationsfolge ausführt. Zu zeigen ist unter dieser Interpretation, dass die Worst-Case-Laufzeit eines solchen Algorithmus nicht in $\mathcal{O}(m)$ liegt. Beschreiben Sie für unendlich viele verschiedene $m \in \mathbb{N}$ jeweils eine Operationsfolge bestehend aus insgesamt m `pushBack`- und `popBack`-Operationen, sodass die Laufzeit für diese so definierte Familie von Operationsfolgen in $\Omega(m^2)$ liegt. Definieren Sie die Operationsfolgen hierbei so, dass zunächst genügend Elemente in das Array eingefügt werden, und provozieren Sie dann in geeigneter Weise durch abwechselnde `popBack`- und `pushBack`-Operationen eine große Anzahl an teuren `reallocate`-Operationen. Orientieren Sie sich an Tutoraufgabe 3.