Technische Universität München
Fakultät für Informatik
Lehrstuhl für Effiziente Algorithmen
Dr. Hanjo Täubig
Tobias Lieber

Sommersemester 2011
Übungsblatt 2
30. Mai 2011

Grundlagen: Algorithmen und Datenstrukturen

Abgabetermin: In der jeweiligen Tutorübung

Hausaufgabe 1

In der Vorlesung wurde vorgestellt, wie man ein dynamisches Array implementieren kann, so dass jede Operation amortisiert höchstens $\mathcal{O}(1)$ Zeit verbraucht. Implementieren Sie die auf der Übungswebseite bereitgestellte abstrakte Klasse `UArray` in Java in der Klasse `UIArray`. Bitte verwenden Sie in den Hausaufgaben, wie in der Vorlesung, für `alpha` bzw. `beta` die Werte 4 bzw. 2.
Achten Sie bei der Abgabe Ihrer Aufgabe darauf, dass Ihre Klasse UIArray heißt und auf den Rechnern der Rayhalle (`rayhalle.informatik.tu-muenchen.de`) mit der bereitgestellten Datei `main` kompiliert werden kann. Anderenfalls kann eine Korrektur nicht garantiert werden.
Schicken Sie die Lösung per Email mit dem Betreff `[GAD] Gruppe <Gruppennummer>` an ihren Tutor.

Lösungsvorschlag

Siehe Übungswebseite.

Hausaufgabe 2

Erweitern Sie Ihre Implementierung aus Hausaufgabe 1 so zu einer Klasse `UIcArray`, dass die Operationen Ihrer Implementierung nicht nur amortisiert, sondern auch im Worst-Case, höchstens $O(1)$ Zeit benötigen.
Diese Aufgabe muss mit der Datei `mainc` kompilierbar sein.
Hinweis: Sie dürfen zusätzliche Arrays benutzen. Kopieren Sie geschickter als in der ersten Implementierung.

Lösungsvorschlag

Siehe Übungswebseite.

Hausaufgabe 3

In der Vorlesung wurde angesprochen, dass ein Dynamisches Array erst verkleinert werden darf, wenn das Array nur noch zur zu einem Viertel gefüllt ist. Beweisen Sie:
Wird das Array schon verkleinert, wenn der Füllstand des Arrays nur noch die Hälfte des reservierten Speicherplatzes beträgt, so stimmt die Behauptung nicht, dass jede Folge von m Operationen auf dem Array in Zeit $\mathcal{O}(m)$ abgearbeitet werden kann.

Lösungsvorschlag

Da nicht jede Folge von m Operationen in Zeit $\mathcal{O}(m)$ laufen soll, genügt es zu zeigen, dass es eine Folge von Operationen (in Abhängigkeit von m, also eigentlich unendlich viele Folgen) gibt die $\Omega(m^{1+\epsilon})$ Zeitschritte benötigt ($\epsilon > 0$). Wir geben eine Folge von Operationen an, die $\Theta(m^2)$ Zeitschritte benötigt.
Wir wählen $n := 2^{\lfloor(\log m)\rfloor-1}$. Somit ist n also die größte Zweierpotenz, die kleiner als $\frac{m}{2}$ ist. Außerdem ist diese Zahl mit Sicherheit größer als $\frac{m}{4}$.
Folgende Sequenz von Operationen benötigt nun $\Omega(m^2)$ Zeitschritte: n-Mal `PushBack` und danach $\lfloor\frac{m-n}{2}\rfloor$-mal abwechselnd: `PushBack` und `PopBack`.
Da n eine Zweierpotenz ist, wissen wir, dass das Array nach n `PushBack`-Operationen voll ist. Die nächste `PushBack`-Operation muss also mindestens $\frac{m}{4}$ Elemente in das neue Array kopieren. Die darauffolgende `PopBack`-Operation muss nun das Array wieder auf die Hälfte verkleinern. Was wieder mit mindestens $\frac{m}{4}$ Kopieroperationen verbunden ist.
Da $m - n \geq \frac{m}{2}$ wissen wir, dass mindestens $\frac{m}{2}$-mal eine Operation – `PushBack` bzw. `PopBack` – ausgeführt wird, die mit $\frac{m}{4}$ Kopieroperationenverbunden ist. Somit folgt also, dass die angegebene Sequenz von Operationen mindestens $(\frac{m}{2}\frac{m}{4}) \in \Omega(m^2)$ Zeitschritte benötigt.

Aufgabe 1

In der Vorlesung lernen wir, Algorithmen aus theoretischer Sicht zu analysieren. Neben dieser Art der Algorithmen-Analyse gibt es die experimentelle Analyse. Hierbei testet man für Eingaben das Laufzeitverhalten eines Algorithmus.
In der Abbildung 1 sind für drei Algorithmen die gemessenen oberen Schranken für die Laufzeit t in Abhängigkeit der Eingabegröße n einer solchen experimentellen Analyse angegeben. Ordnen Sie die Algorithmen nach ihrer Laufzeit.

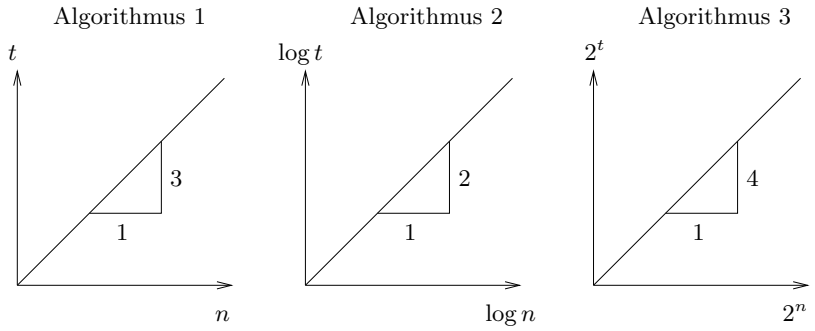

Abbildung 1: Graphen zu den Algorithmen in Aufgabe 1

Lösungsvorschlag

Es sei t_i die vom Algorithmus i, $i \in \{1, 2, 3\}$, benötigte Laufzeit bei Eingabelänge n. Dann gilt

$$\begin{aligned} t_1 &= 3 \cdot n \\ \log t_2 &= 2 \cdot \log n \\ 2^{t_3} &= 4 \cdot 2^n \ . \end{aligned}$$

Umstellen nach t_i ergibt

$$\begin{aligned} t_1 &= 3 \cdot n \\ t_2 &= 2^{2 \cdot \log n} = n^2 \\ t_3 &= \log(4 \cdot 2^n) = n + 2 \ . \end{aligned}$$

Somit gilt $t_3 < t_1 < t_2$ für $n \geq 4$.

Aufgabe 2

Wir betrachten nochmals den Minimum-Algorithmus aus der vorigen Woche (Siehe Algorithmus 1). Nachdem wir bereits wissen, dass die Worst-Case Laufzeit in $\mathcal{O}(n)$ liegt, wollen wir dieses Mal die *genaue* Average-Case Laufzeit berechnen (nicht die asymptotische Average-Case Laufzeit).
Sie dürfen davon ausgehen, dass jede Eingabe der Länge n aus n unterschiedlichen Zahlen besteht.

Algorithmus 1: Min

```
   Input : int A[], int n
1  int i = 1
2  int min = A[0]
3  while i < n do
4  |  if A[i] < min then
5  |  |  min = A[i]
6  |  i = i + 1
7  return min
```

Lösungsvorschlag

Da sich Worst-Case und Best-Case Laufzeiten von Algorithmus 1 nur um $n-1$ Operationen unterscheiden, bringt es hier nichts die asymptotische Analyse zu verwenden. Außerdem ist klar, dass alle Operationen außer der Zeile 5 immer ausgeführt werden. Somit ergeben sich also ca. $2 + 3(n-1) + 3 = 2 + 3n$ Operationen, die für jede Eingabe (eine Eingabe x ist ein n-Tupel $(x_1, x_2, \ldots, x_n)$ und jedes x_i ist eine Zahl) ausgeführt werden müßen.
Hinzu kommen die Anzahl der Updates (= Anzahl der neuen minimalen Elemente in einer Eingabesequenz), die anteilig für jede Sequenz berechnet werden:

$$\frac{1}{|I_n|} \sum_{x \in I_n} T(x)$$

wobei I_n die Menge aller Eingaben und $T(x) := |\{l \in \{2...n\} : (\forall k < l).[x_k > x_l]\}|$ die Anzahl der Updates für eine Eingabe x ist.
Somit ergibt sich für die gesamte Average-Case Laufzeit:

$$A(n) := 2 + 3n + \frac{1}{|I_n|} \sum_{x \in I_n} T(x)$$

Hierbei wollen wir jetzt den letzten Term auch noch in Abhängigkeit von n bestimmen. Führen wir uns nocheinmal vor Augen, was $T(x)$ aussagt: $T(x)$ gibt für eine Eingabe x

die Anzahl der Updates an, die eine Eingabe erzeugt. In einer Summe ist dies insofern ungünstig, da wir schlecht abschätzen können wieviele Eingaben genau j Updates ausführen. Um dennoch eine Abschätzung für diesen Term zu erhalten, werden wir die Summe umordnen.
Definieren wir $T_n(l) := |\{x \in I_n : (\forall k : 1 \leq k < l).[x_k > x_l]\}|$ als die Anzahl der Eingaben, die nach dem Vergleich des Elements x_l ein Update des Minimum-Elements ausführen müssen, so gilt:

$$\frac{1}{|I_n|}\sum_{x\in I_n} T(x) = \frac{1}{|I_n|}\sum_{l=2}^{n} T_n(l)$$

Hierbei ist zu beachten, dass eine Eingabe x genau $T(x)$-mal in unterschiedlichen $T_n(i)$ vorkommt. Somit wird also der Wert der Summe nicht verändert. Im Gegensatz zu $T(x)$ lässt sich der Wert $T_n(i)$ recht einfach berechnen. Er ist gerade:

$$T_n(i) = \binom{n}{i}(i-1)!(n-i)!$$

da wir i Elemente bis zum Minimum-Update an Stelle i benötigen wählen wir aus n Elementen der Eingabe gerade i aus ($\Rightarrow \binom{n}{i}$). Von diesen i Elementen ist das kleinste Element gerade das, welches in einer Eingabe an Position i steht (da es ja ein Minimum-Update verursacht). Die anderen $i-1$ Elemente können gerade auf $(i-1)!$ verschiedene Weisen angeordnet werden. Genauso gibt es für die restlichen $n-i$ Elemente, die nach der i-ten Stelle kommen gerade $(n-i)!$ Möglichkeiten diese anzuordnen. Somit ergibt sich also:

$$\frac{1}{|I_n|}\sum_{l=2}^{n} T_n(l) = \frac{1}{n!}\sum_{l=2}^{n}\binom{n}{l}(l-1)!(n-l)! = \frac{1}{n!}\sum_{l=2}^{n}\frac{n!}{(n-l)!(l!)}(l-1)!(n-l)! =$$

$$\sum_{l=2}^{n}\frac{1}{l} = H_n - 1$$

Mit ein wenig Analysis ($\int_1^n \frac{\mathrm{d}x}{x} = \ln n$) kann man zeigen, dass $H_n < 1 + \ln n$ gilt (das Verständnis dieser Ungleichung ist keineswegs Zentral für diese Vorlesung). Es kann sogar noch gezeigt werden, dass $\ln n < H_n$ gilt. Somit gilt insgesamt für die Average-Case Laufzeit:

$$1 + 3n + \ln n < A(n) < 2 + 3n + \ln n$$

Zusatzaufgabe 0

Betrachten Sie die Konto-Methode angewandt auf unbeschränkte Arrays für beliebige Werte α und β ($\alpha > \beta > 1$).
Zeigen Sie, dass in jedem Schritt genug Token zur Verfügung stehen, wenn bei jedem Aufruf von `pushBack` $\beta/(\beta-1)$ Token, bei jedem Aufruf von `popBack` $\beta/(\alpha-\beta)$ Token einbezahlt werden.
Zeigen Sie dafür, dass bei jedem Aufruf von `reallocate` genügend Token vorrätig sind. Überlegen Sie sich dazu, dass `reallocate` immer mit dem Wert $\beta n'$ (für ein $n' \in \mathbb{N}$)

aufgerufen wird und nach einem solchen Aufruf das Array $w = n'\beta$ Stellen hat, von denen n' belegt und $(\beta - 1)n' = ((\beta - 1)/\beta)w$ Stellen frei sind. `reallocate` wird erst wieder aufgerufen, falls $n = w$ oder $\alpha n \leq w$ ist.

Lösungsvorschlag

Wir betrachten, wie in der Aufgabe angesprochen, die Situation nach einem Aufruf von `reallocate`. Sei n' der aktuelle Wert von n. Dann ist $w = \beta n'$ und es sind n' Felder belegt. w ändert sich erst, sobald wieder `reallocate` aufgerufen wird. Dazu kann es in zwei Fällen kommen: $n = w$ oder $\alpha n \leq w$.
Im ersten Fall wurden bis zu dieser Situation $n - n'$ Elemente eingefügt (alle leeren Felder des Arrays gefüllt). Für jedes Einfügen (`pushBack`) wurden $\beta/(\beta-1)$ Token eingezahlt.

$$\begin{aligned}
(n - n')(\beta/(\beta - 1)) &= (n - n')(\beta/(\beta - 1)) \\
&= (\beta n' - n')(\beta/(\beta - 1)) \\
&= ((\beta - 1)n')(\beta/(\beta - 1)) \\
&= n'\beta \\
&= w \\
&= n
\end{aligned}$$

Insgesamt wurden also n Token bezahlt, die für die n Kopieroperationen benutzt werden können.
Im zweiten Fall wurden $n' - n$ Elemente gelöscht. Für jedes Löschen (`popBack`) wurden $\beta/(\alpha - \beta)$ Token einbezahlt.

$$\begin{aligned}
(n' - n)(\beta/(\alpha - \beta)) &= (\beta n' - \beta n)/(\alpha - \beta) \\
&= (w - \beta n)/(\alpha - \beta) \\
&\geq (\alpha n - \beta n)/(\alpha - \beta) \quad \text{(Vorraussetzung 2. Fall)} \\
&= (\alpha - \beta)n/(\alpha - \beta) \\
&= n
\end{aligned}$$

Insgesamt wurden also mindestens n Token einbezahlt, die für die n Kopieroperationen benutzt werden können.
In beiden Fällen können noch weitere Operationen zwischendurch hinzukommen, wie zum Beispiel ein ständiger Wechsel zwischen `pushBack` und `popBack`. Da in diesen Situationen allerdings nie ein Aufruf von `reallocate` stattfindet, werden nur zusätzliche Token einbezahlt.
Nun bleibt noch zu überprüfen, ob der erste Aufruf von `reallocate` „bezahlt“ werden kann: Am Anfang ist $n = 0$ und $w = 1$. `reallocate` wird demnach aufgerufen, sobald zwei Elemente nacheinander eingefügt werden. Dadurch werden $2 \cdot \beta/(\beta - 1)$ Token einbezahlt. Für `reallocate` wird hier 1 Token benötigt. Da $2 \cdot \beta/(\beta - 1) \geq 1$ gilt, ist auch dieser Fall gezeigt.