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

Sommersemester 2011
Übungsblatt 2
19. 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.

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.

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.

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.

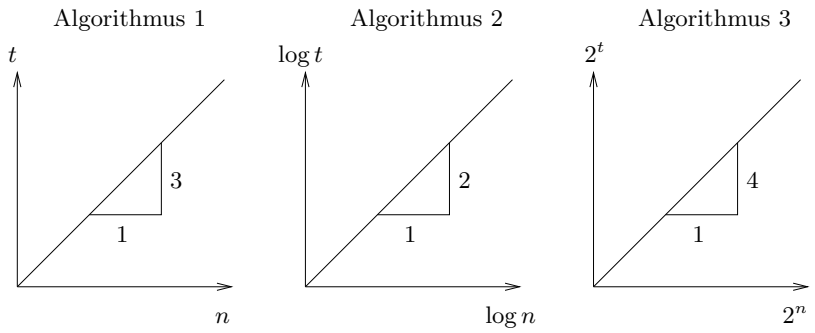

Abbildung 1: Graphen zu den Algorithmen in Aufgabe 1

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
```

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.