Technische Universität München
Institut für Informatik
Lehrstuhl für Effiziente Algorithmen
Dr. Michal Mnuk

SS 1998
Übungsblatt 6
24. Juni 1998

Grundlegende Algorithmen

Abgabe bis 8. Juli 1998 in der Übung

Aufgabe 16

Die Fibonaccizahlen sind rekursiv definiert durch

- $F_0 := 0$, $F_1 := 1$, und
- $F_n := F_{n-1} + F_{n-2}$.

Zeigen Sie, daß die Fibonaccizahlen exponentiell wachsen, genauer, daß es eine Konstante c gibt, so daß $F_n \in \Omega(c^n)$. Zusammen mit der oberen Schranke aus der Vorlesung folgt dann also $\log(F_n) \in \Theta(n)$.

Aufgabe 17

Wir wollen hier an einem ganz einfachen Beispiel die Technik der *amortisierten Komplexitätsanalyse* vorstellen, die beim Studium von effizienten Datenstrukturen eine wichtige Rolle spielt. Besonders dann, wenn man gewisse „Aufräumarbeiten" (man denke etwa an das Ausbalancieren von Bäumen) erledigt werden, während die eigentlich interessanten Zugriffe passieren.
Auf eine zu Anfang leere Datenstruktur wird eine Folge $(O_1, \ldots, O_n)$ von n Operationen angewandt. Wie lange dauert es im Worst–Case, eine solche Folge abzuarbeiten? Man kann natürlich für jede Operation den Worst–Case einzeln abschätzen, was aber häufig zu pessimistisch ist. Besser ist folgendes Bild: Abhängig von der Eingabefolge durchläuft die Datenstruktur eine Folge von Zuständen Z_i. Jeder Zustand wird abhängig vom Vorgängerzustand, dessen Bewertung und der zu ihm führenden Operation bewertet. Formal induziert die Eingabefolge also eine Folge $((Z_i, w_i))$. Den Wert w_i kann man als „bisher eingesparte Arbeit" interpretieren.
Zu Anfang ist w_0 immer 0. Wenn eine „billige" Operation durchgeführt wird, so zahlt man parallel dazu einen Betrag auf das „Arbeitskonto" ein, bei einer „teuren" Operation, kann man das „Arbeitskonto" ganz oder teilweise aufbrauchen. Das „Arbeitskonto" darf nie überzogen werden. Für die Kosten einer Folge gilt nun

$$
\begin{aligned}
&T((O_1, \ldots, O_n)) \\
&= \sum_{i=1}^{n} (\text{ Tatsächliche Kosten für Operation } O_i \text{ im Zustand } Z_i) \\
&\overset{w_0=0}{=} -w_n + \sum_{i=1}^{n} [(\text{ Tatsächliche Kosten für Operation } O_i \text{ im Zustand } Z_i) + w_i - w_{i-1}] \\
&\overset{w\geq 0}{\leq} \sum_{i=1}^{n} [(\text{ Tatsächliche Kosten für Operation } O_i \text{ im Zustand } Z_i) + w_i + w_{i-1}] \\
&=: \sum_{i=1}^{n} (\textit{Amortisierte} \text{ Kosten für Operation } O_i \text{ (z.z.: Unabhängig vom Zustand } Z_i.))
\end{aligned}
$$

Wenn es nun gelingt, die Summanden in eckigen Klammern gleichmäßig für beliebige Eingabefolgen nur abhängig von der Art der i–ten Operation abzuschätzen, so bezeichnet man diese Summanden auch als die *amortisierten Kosten* der entsprechenden Operation O_i. Die letzte Summe hängt dann nur davon ab, wie häufig die verschiedenen Typen von Operationen vorkommen. Diese Abschätzung kann besser sein, als wenn man gleich die einzelnen Summanden der ersten Summe abgeschätzt hätte. Die Kunst besteht darin, die Regeln zum Aktualisieren des (gedachten) „Arbeitskontos" geschickt zu wählen.
Betrachten Sie nun als Beispiel einen STACK, implementiert durch eine anfangs leere zyklische Liste. Die möglichen Operationen bestehen aus $\{\texttt{push}, \texttt{pop\_all}\}$. Bei der Operation `push` wird ein beliebiges Objekt auf den Stapel gelegt, beim `pop_all` werden alle Objekten herausgeholt. Die tatsächlichen Kosten seien c für ein `push` und $c \cdot k$ beim `pop_all`, also proportional zur aktuellen Höhe $k \leq n$ des Stacks (k beschreibt hier den Zustand ausreichend). Welche Kosten entstehen bei Anwendung einer beliebigen Folge solcher Operationen der Länge n? Es ist klar, daß die Abschätzung durch $c \cdot n^2$ zwar nicht falsch, aber viel zu grob ist.

Zeigen Sie, daß für alle $(O_1, \ldots, O_n)$ gilt:

$$T((O_1, \ldots, O_n)) \leq 2cn.$$

Tip: Finden Sie eine geeignete Vorschrift, das „Arbeitskonto" w_i zu aktualisieren, so daß die amortisierten Kosten für jede der Operationen `push` und `pop_all` in jedem Zustand k höchstens $2c$ sind. Machen Sie sich zunutze, daß ein `pop_all` nur dann teuer, ist, wenn vorher entsprechend viele `push`–Operationen durchgeführt worden sind.

Aufgabe 18

Beweisen Sie, daß die in der Vorlesung vorgestellten Funktionen `is_member`, `delete` und `insert` korrekt arbeiten, d.h. daß `is_member` genau dann die Antwort *Ja* liefert, wenn der gesuchte Schlüssel im Wörterbuch enthalten ist, und daß nach dem Aufruf von `insert` und `delete` die Suchbaumeingenschaft in allen Knoten wieder hergestellt ist.

Aufgabe 19

Sei T ein AVL-Baum. Entwerfen Sie anhand von `AVL_insert` die Funktion `AVL_delete`, die einen Schlüssel in T löscht. Diese Funktion ruft zunächst `delete` für Suchbäume und sorgt anschließlich für Rebalancierung des Baumes.