Fakultät für Informatik
Lehrstuhl 11
Prof. Dr. Petra Mutzel
Dipl.-Inf. Andre Droschinsky

Übungen zur Vorlesung
Effiziente Algorithmen
Sommersemester 2018

Übungsblatt 7
Besprechungszeit:
4.6.-7.6.2018

Bitte beachten Sie die Hinweise zu den Übungen und der Übungsgruppenverteilung auf der Homepage der Übung.

Aufgabe 7.1 – Wiederholung (6 Punkte)

- Was ist amortisierte Analyse und worin bestehen die Unterschiede zur üblichen Worst-Case-Analyse?
- Handelt es sich bei der amortisierten Analyse um eine Worst-Case- oder Average-Case-Analyse?
- Welche amortisierten Analysemethoden haben Sie in der Vorlesung kennen gelernt? Wie funktionieren diese Methoden?
- Was sind die wichtigsten Lemmata und Ideen für den Beweis der amortisierten Laufzeit der Union-Find-Datenstruktur?
- Wie ist die Aufteilung der Kosten auf die Kostenträger in dem Beweis?

Aufgabe 7.2 – Amortisierte Analyse (6 Punkte)

Sei z eine Zahl in Trinärdarstellung, jede Ziffer hat also den Wert 0,1 oder 2. Die Darstellung ist analog zu den Binärzahlen aus der Vorlesung.

a) Zeigen Sie, eine Folge von n `PlusEins`-Operationen beginnend bei $z = 0$ hat amortisierte Kosten $\mathcal{O}(1)$ pro Operation. Benutzen Sie dazu die 3 Ihnen bekannten Analysemethoden. Tipp: Orientieren Sie sich an den Beweisen der Vorlesung zum Binärzähler.

b) Nehmen Sie an, Sie starten bei $z = 17$, und anstelle einer `PlusEins`-Operation haben sie Operationen `Plus(i)`, wobei i eine beliebige natürliche Zahl zwischen 1 und einem festen $k \in \mathbb{N}$ ist.

Wie hoch sind die amortisierten Kosten pro `Plus(i)`-Operation in einer Folge von n solchen Operationen? Beweisen Sie Ihre Aussage mit amortisierter Analyse.

Aufgabe 7.3 – Union-Find (8 Punkte)

In der Vorlesung wird die gewichtete Vereinigungsregel nach der Größe der Bäume verwendet (hänge Baum mit kleinerer Anzahl an Elementen an den Baum mit größerer Anzahl an Elementen). Alternativ kann auch die gewichtete Vereinigungsregel nach der Höhe der Bäume verwendet werden (hänge den Baum mit der kleineren Höhe an den Baum mit der größeren Höhe).

Beweisen Sie das folgende Lemma: *UNION mit gewichteter Vereinigungsregel nach der Höhe der Bäume liefert immer Bäume, deren Höhe $h(T)$ durch $\log s(T)$ nach oben beschränkt ist.*

Hinweis: Das Lemma befindet sich auf Seite 21 der Union-Find-Folien (`5-KruskalMST`) und $s(T)$ ist als die Anzahl der Knoten im Baum T definiert.

Aufgabe 7.4 – Amortisierte Analyse (10 Punkte)

Gegeben sei eine Feld-Datenstruktur mit den folgenden Operationen.

- `A=newArray(n)`: Erzeugt ein Feld fester Größe, in dem genau n Elemente gespeichert werden können.
- `x=A.get(i)`: Gibt das i-te Element x des Feldes A zurück, wobei gilt $1 \leq i \leq n$.
- `A.set(i,x)`: Speichert an i-ter Stelle das Element x ab.
- `n=A.size()`: Gibt die Größe n des Feldes zurück.
- `A.delete()`: Löscht das Feld A.

Mit Hilfe dieser Feld-Datenstruktur soll ein Stapel realisiert werden. Dazu sollen zunächst die Elemente mit Hilfe einer Methode `push` nacheinander in dem Feld gespeichert werden, wobei das tiefste Element des Stapels auf Position 1 gespeichert wird. Falls das Feld voll ist, soll ein neues Feld angelegt werden, das doppelt so groß ist wie das vorherige Feld. Die vorhandenen Elemente sowie das neu einzufügende Element sollen dabei in das neue Feld kopiert werden. Das alte Feld wird danach gelöscht.

Für einen Stapel S existieren folgenden Operationen.

- `n=S.elements()`: Gibt die Anzahl der Stapelelemente zurück. Es gilt stets $0 \leq$`S.elements()`$\leq$`A.size()`.
- `S.push(x)`: Legt des Element x auf den Stapel. Das Feld A soll bei Bedarf durch ein doppelt so großes Feld ersetzt werden.

Wenn beispielsweise auf einen Stapel mit 4 Elementen, dessen Feld A Platz für 4 Elemente hat, ein Element gelegt werden soll, soll ein neues Feld mit Kapazität 8 erzeugt werden, in das die ersten vier Elemente von A, sowie das neue Element x kopiert werden.

Hinweis: Diese Verdopplungstaktik kommt in der Praxis in vielen Datenstrukturen zum Einsatz, beispielsweise beim Hashing (später) oder bei der C++ Klasse `std::vector`.

a) Gegeben sei ein Stapel S mit Kapazität `A.size()=1`. Überlegen Sie, wie Sie die Methode `push` implementieren können, so dass eine Folge von n `push`-Operationen auf S amortisierte Kosten $\mathcal{O}(n)$ hat. Zur Vereinfachung nehmen Sie an, dass nur die Operationen `get` und `set` auf einem Feld tatsächliche Kosten 1 haben. Alle anderen Operationen auf A, die Operation `S.elements()`, sowie etwaige Vergleiche oder Schleifendurchläufe werden mit tatsächlichen Kosten 0 angenommen.

Der Stapel soll jetzt um die Methode `pop` ergänzt werden. Diese soll das oberste Element des Stapels zurückgeben und löschen.

- `x=S.pop()`: Gibt das oberste Element des Stapels zurück und löscht es. Diese Operation ist nur auf einem nicht leeren Stapel erlaubt.

Um nicht zu viel ungenutzten Speicher im zugrunde liegenden Feld A zu haben, soll die Methode `pop` so implementiert, dass der Füllstand des Feldes A immer mindestens 1/4 beträgt (außer das Feld ist leer).

b) Gegeben sei ein Stapel S mit Kapazität 1. Überlegen Sie, wie Sie die Methode `pop` implementieren können, so dass jede gültige Folge von insgesamt n `push` und `pop`-Operationen auf S amortisierte Kosten $\mathcal{O}(n)$ hat. Gültig bedeutet, die Operation `pop` wird nur auf einem nicht leerem Stapel ausgeführt. Die tatsächlichen Kosten der Operationen sind wie in a). Tipp: Es ist sinnvoll, die Arraygrößen so zu wählen, dass diese stets Zweierpotenzen sind.