Lehrstuhl Algorithmen
und Komplexität
Prof. Dr. Fabian Kuhn
Sebastian Daum, Oleksii Saukh

Algorithmen und Datenstrukturen
(Informatik II) SS 2014

http://ac.informatik.uni-freiburg.de/teaching

UNI FREIBURG

Übungsblatt 3

Abgabe: Dienstag, 27. Mai, 16:00 Uhr

Laden Sie Ihre Lösungen für theoretische *Übungen (ab diesem Übungsblatt) im Format "loesung-xx-y.pdf" hoch, wobei "xx" die Nummer des aktuellen Übungsblattes ist (bei Bedarf mit führender Null) und "y" die Aufgabennummer, d.h., erzeugen Sie* pro (theoretische) Aufgabe *eine* .pdf*-Datei. Scans, Bilder von Scans etc. bitte zuerst ins pdf-Format umwandeln. Bei Programmieraufgaben müssen Sie – wann immer es relevant/durchführbar ist – Unit Tests sowie einen Stylecheck durchführen. Laden Sie außerdem eine Datei "erfahrungen.txt" hoch, in welcher Sie Ihre Erfahrungen und Meinungen zu dem Übungsblatt teilen.*

Aufgabe 1 (5 Punkte)

Man nennt einen Sortieralgorithmus *stabil*, wenn, zwei (oder mehr) Elemente mit gleichem Wert nach dem Sortieren die gleiche Reihenfolge im ursprünglichen Array haben wie vor dem Sortieren. Mathematischer ausgedrückt: Wenn $a_1, \dots, a_n$ die Elemente im unsortierten Array sind und p_i die *Position* von Element a_i im *sortierten* Array, dann:

$$p_i < p_j \quad \Rightarrow \quad (a_i < a_j \vee (a_i = a_j \wedge i < j))$$

Kann ein Algorithmus das nicht garantieren, nennt man ihn *instabil*.

Welche von den Sortieralgorithmen *SelectionSort*, *MergeSort* und *QuickSort*, so wie sie in der Vorlesung vorgestellt wurden, ist stabil und welche instabil? Bringen Sie für diejenigen Algorithmen, die Sie als instabil identifizieren, ein Gegenbeispiel.

Hinweis: Für ein Gegenbeispiel könnten Sie ein Array aus *Tupeln* $(value, pos)$ verwenden, z.B.

$$(1,1), (5,2), (2,3), (1,4), (5,5).$$

Wenn dann im sortierten Array z.B. das Element $(1,4)$ vor dem Element $(1,1)$ auftaucht, dann ist der Algorithmus offenbar instabil.

Aufgabe 2 (5 Punkte)

Angenommen Sie haben ein Array A aus Elementen, welche aus zwei Schlüsseln key_1 und key_2 bestehen, wobei die Werte *key* aus $\{1, \dots, n\}$ stammen. Sie sollen das Array nun lexikografisch sortieren, d.h., im sortierten Array soll für zwei Elemente a_i und a_{i+1} gelten, dass entweder $a_i.key_1 < a_{i+1}.key_1$, oder, falls $a_i.key_1 = a_{i+1}, key_1$, dass dann $a_i.key_2 \leq a_{i+1}.key_2$. Wenn Sie dafür *CountingSort* verwenden und einmal nach dem einen Schlüssel sortieren und dann nach dem anderen, nach welchem der Schlüssel müssen Sie zuerst sortieren und geben Sie für jeden Sortiervorgang an, ob dieser stabil sein *muss* oder *nicht*. Begründen Sie kurz Ihre Lösung.

Aufgabe 3 (15 Punkte)

In dieser Aufgabe wollen wir ein Array A von n Integers aus dem Wertebereich $\mathbb{M} := \{0, \dots, M\}$ besonders geschickt sortieren, wobei $M \gg n$ (M ist "viel größer" als n). Dazu machen wir explizit Gebrauch von der Idee aus der vorhergehenden Aufgabe, indem wir jeden Wert $x \in \mathbb{M}$ als k-stellige Zahl in Basis b interpretieren — d.h., $x = (x_{k-1}x_{k-2}x_{k-3}\dots x_2x_1)_b$, wobei jedes $x_i \in \{0, 1, \dots, b-1\}$ ($b = 2$ entspricht der Binärdarstellung, $b = 10$ der gewohnten Dezimaldarstellung) — und dann nach den einzelnen Ziffern sortieren.

Im public*-Ordner finden Sie vorgefertigte Strukturdateien, welche Sie herunterladen und anpassen, bzw. mit Programmcode füllen können.*

1. Schreiben Sie dazu zuerst eine Prozedur *sortByDigit*(A, b, d), welche ein Array A, eine Basis b und einen Divisor d als Eingabe nimmt, und A passend nach nur einer Ziffer sortiert. Sie können dafür auch den in der Vorlesung vorgestellten Algorithmus *CountingSort* verwenden und diesen anpassen.

2. Implementieren Sie dann den Algorithmus *radieschenSort*(A, b), welcher mit Hilfe von *sortByDigit* das Array A sortiert. Zum Testen können Sie Basis $b = 2$ oder $b = 10$ verwenden.

3. Führen Sie nun mehrere Tests durch, indem Sie ein Array A mit zufälligen Werten aus $\mathbb{M}$ befüllen, wobei Sie M auf MAXINT setzen ("signed" 32-bit). Für die Länge n des Arrays iterieren Sie durch die Werte $2^{10}, 2^{11}, \ldots, 2^{20}$ und für die Basis b durch die Werte $2^1, 2^2, \ldots, 2^{20}$. Führen Sie jeden Tests 9-mal durch (das sind ingesamt 1980 Testläufe), messen Sie die Zeit, die Ihr Algorithmus zum Sortieren braucht, und geben Sie den *Median* (nicht den Durchschnitt!) zurück, d.h., jeweils den fünfthöchsten bzw. fünftniedrigsten Wert. Vergleichen Sie die Ergebnisse in einem Tabellenkalkulationsprogramm (Excel, OpenCalc, ...).

4. Analysieren Sie den Algorithmus nach seiner Laufzeit $T_b(n, M)$ in Abhängigkeit von n, M und b. Eruieren und *beweisen* Sie, wie Sie b wählen müssen, so daß die Laufzeit asymptotisch für *beliebige* M (allerdings $M > n$) optimal ist. D.h., geben Sie ein b' an, so dass $T_{b'}(n, M) \in O(T_b(n, M))$ für alle $b \in \mathbb{N}$ und alle $M \in \mathbb{N}$.

Aufgabe 4 (5 Punkte)

Erläutern Sie (z.B. mittels Pseudo-Code), wie man unter Benutzung von 2 *Stacks* eine *Queue* implementieren kann. Begründen Sie kurz, warum Ihre Lösung funktioniert.

Aufgabe 5 (10 Punkte)

Implementieren Sie eine Datenstruktur *DoublyLinkedList*, in welcher jedes Element einen Schlüssel enthält, sowie zwei Zeiger auf das Element davor und das danach. Dafür werden Sie zwei Klassen schreiben müssen - *ListElement* und *DoublyLinkedList*.

Die erste Klasse enthält ein Feld *key* und zusätzlich zwei Zeiger $previous$ und $next$ auf die Nachbarelemente in der Liste.

Die zweite Klasse implementiert *Doubly Linked List* - eine doppelt verkettete Liste aus Elementen *ListElement*.

Im public*-Ordner finden Sie vorgefertigte Strukturdateien, welche Sie herunterladen und anpassen, bzw. mit Programmcode füllen können.*