Humboldt-Universität zu Berlin
Institut für Informatik
Prof. Dr. Ulf Leser
E. Angriman, F. Nelles, M. Sänger, Dr. P. Schäfer

Berlin, den 08.05.2019

Übungen zur Vorlesung
Algorithmen und Datenstrukturen

Übungsblatt 3

Abgabe: Mittwoch, den 22.05.2019, bis 11:10 Uhr vor der Vorlesung im Hörsaal. Die Übungsblätter sind in Gruppen von 2/3 Personen zu bearbeiten. Die Lösungen sind auf nach Aufgaben getrennten Blättern abzugeben. Heften Sie bitte die zu einer Aufgabe gehörenden Blätter vor der Abgabe zusammen. Vermerken Sie auf allen Abgaben Ihre Namen, Ihre **CMS-Benutzernamen**, Ihre **Abgabegruppe** (z.B. AG123) aus Moodle, und den **Übungstermin** (z.B. Gruppe 2 oder Mo 13 Uhr bei M. Sänger), an dem Sie Ihre korrigierten Blätter zurückerhalten möchten.
Beachten Sie die Informationen auf der Übungswebseite (https://hu.berlin/algodat19).

Konventionen:

- Mit log wird der Logarithmus $\log_2$ zur Basis 2 bezeichnet.
- Für ein Array A ist $|A|$ die Länge von A, also die Anzahl der Elemente in A. Die Indizierung aller Arrays auf diesem Blatt beginnt bei 1 (und endet also bei $|A|$).

Aufgabe 1 (Schreibtischtests) — 4+4+4=12 Punkte

In den folgenden Unteraufgaben sollen Sie jeweils einen *Schreibtischtest* durchführen. Das heißt, Sie führen auf Papier einen gegebenen Algorithmus für gegebene Eingaben aus. In der Unteraufgabe ist jeweils angegeben, welche Zwischenergebnisse Sie als Lösung einreichen sollen. Notieren Sie ein Array entweder als Liste in eckigen Klammern (also in der Form $[a_1, \ldots, a_n]$) oder wie in den VL-Folien als Tabelle der Form | a_1 | ... | a_n | .

a) Führen Sie einen Schreibtischtest für den Algorithmus **QuickSort** aus der VL für das Eingabe-Array $A = [4, 6, 11, 10, 5, 22, 3, 18, 8]$ durch, wobei Sie als Ordnung die natürliche Ordnung auf natürlichen Zahlen annehmen. Als Pivot-Element wählen Sie das am weitesten rechts stehende Element des aktuellen Teil-Arrays. Geben Sie den aktuellen Wert von A nach jeder Swap-Operation an. Unterstreichen Sie jeweils das in diesem Aufruf betrachtete Teil-Array.

b) Führen Sie einen Schreibtischtest für den Algorithmus **MergeSort** aus der VL für das Eingabe-Array $A = [\mathrm{d}, \mathrm{f}, \mathrm{k}, \mathrm{j}, \mathrm{e}, \mathrm{v}, \mathrm{c}, \mathrm{r}, \mathrm{h}]$ durch, wobei Sie als Ordnung die alphabetische Ordnung auf den Buchstaben annehmen. Geben Sie die Zwischenergebnisse nach jedem Aufruf von MergeSort bzw. Merge in Form eines Graphen wie auf den VL-Folien an (siehe z.B. Folie 8).

c) Führen Sie einen Schreibtischtest für den Algorithmus **BucketSort** aus der VL für das Alphabet $\Sigma = \{0, 1, 2, 3\}$, $m = 3$, und das Eingabe-Array

$$A = [010, 012, 023, 022, 011, 112, 003, 102, 020]$$

durch. Wir nehmen weiterhin an, dass das Alphabet Σ linear geordnet ist, und dass die Ordnung auf den Zeichenketten die *lexikographische* Ordnung ist. Geben Sie die Zwischenergebnisse wie auf den Vorlesungsfolien an.

Aufgabe 2 (Sortierung spezieller Arrays) **3 + 3 + 3 + 3 + 3 = 15 Punkte**

Ein *allgemeines Sortierverfahren* ist ein Sortierverfahren, welches die zu sortierenden Elemente nur vergleichen kann und sonst keinerlei Eigenschaften der Elemente ausnutzt. Beweisen oder widerlegen Sie für jede der Teilaufgaben getrennt: Es gibt ein allgemeines Sortierverfahren, welches ein Array A von n beliebigen (in konstanter Zeit vergleichbaren) Elementen im Worst Case in Laufzeit $\mathcal{O}(n)$ sortiert, falls...

a) ... A so vorsortiert ist, dass alle Elemente der ersten Hälfte von A kleiner sind als alle Elemente der zweiten Hälfte.

b) ... die Elemente von A vom Anfang beginnend bis zu einem maximalen Element aufsteigend sortiert sind und ab diesem bis zum Ende absteigend sortiert sind.

c) ... die Länge von A durch 10 teilbar ist, und jeweils 10 aufeinanderfolgende Elemente (vom Anfang beginnend) aufsteigend sortiert sind.

d) ... für jedes $1 \leq i \leq n - 10$ gilt: $A[i] \leq A[i + 10]$.

e) ... in A nur höchstens $m \in \mathbb{N}_{>0}$ viele unterschiedliche Elemente vorkommen. Hierbei sei m eine Konstante.

Hinweise: Für den Fall, dass es ein solches allgemeines Sortierverfahren gibt, können Sie als Beweis die Idee eines konkreten Verfahrens beschreiben. Sie dürfen beliebig viel zusätzlichen Speicherplatz verwenden. Falls kein solches allgemeines Sortierverfahren existiert, können Sie die in der Vorlesung gezeigte untere Schranke für allgemeine Sortierverfahren nutzen.

Aufgabe 3 (Untere Schranke für `merge`) **6 Punkte**

Gegeben seien zwei aufsteigend sortierte Arrays X und Y mit jeweils n Schlüsseln. Zeigen Sie: Lässt man als einzige Operation Vergleiche von je zwei Schlüsseln zu, so benötigt jeder deterministische Algorithmus im Worst Case mindestens $2n - 1$ Vergleiche, um beide Arrays zu einem aufsteigend sortierten Gesamtarray Z der Länge $2n$ zu verschmelzen.

Hinweis: Beweisen Sie per Widerspruch: Überlegen Sie sich zunächst eine Instanz, in der beim `Merge` möglichst viele Vergleiche anfallen. Zeigen Sie für diese Instanz dann, dass jeder der Vergleiche nötig ist, da sich der Algorithmus sonst für diese Instanz und eine leicht modifizierte Instanz exakt gleich verhält und der Algorithmus somit nicht korrekt sein kann.

Aufgabe 4 (Linked List) **(3+2+2+2)+(4+4)=17 Punkte**

In dieser Aufgabe sollen Sie eine einfach verkettete Liste und die folgenden dazugehörigen Methoden implementieren. Eine einfach verkettete Liste ist eine Folge von Knoten und jeder Knoten besteht aus einem Wert (Integer) und einem Zeiger zu seinem nachfolgenden Knoten. Der Zeiger ist `null`, wenn der Knoten der letzte in der Liste ist. Der erste Knoten in der Liste wird als Kopf (head) bezeichnet und hat die Position 0 und n ist die Anzahl der Elemente der Liste.

Erster Teil Implementieren Sie die folgenden Standardmethoden einer verketteten Liste, welche in der Datei `LinkedListExercise.java` schon deklariert, aber noch nicht implementiert sind:

1. `nodeAt(int pos)`: Gibt den Knoten an Position `pos`, mit $0 \leq pos < n$, der Liste zurück. Falls der Knoten an der Position nicht existiert, kann *null* zurückgegeben werden.

2. `insert(int val, int pos)`: Fügt einen neuen Knoten mit Wert `val` an die Position `pos` der Liste ein. Falls $pos < 0$ oder $pos > n$ ist, wird eine InvalidPositionException geworfen. Nutzen Sie dafür

   throw new InvalidPositionException();

3. `elementAt(int pos)`: Gibt den Wert des Knoten an Position `pos` der Liste zurück. Falls die übergebene Position nicht existiert, wird eine InvalidPositionException geworfen.

4. `delete(int pos)`: Löscht den Knoten an Position `pos` in der Liste. Falls die übergebene Position nicht existiert, wird eine InvalidPositionException geworfen.

Zweiter Teil Nachdem Sie die vier Standardmethoden implementiert haben, benutzen wir diese, um komplexere Methoden zu implementieren. Implementieren Sie dazu die folgenden Methoden in der Datei `LinkedListExercise.java`:

1. `reverse()`: Diese Methode dreht den Inhalt der Liste um; Zum Beispiel soll eine Liste $1 \to 2 \to 3$ nach Aufruf von `reverse()` die Form $3 \to 2 \to 1$ haben.

2. `split(int val)`: Ordnet die Knoten der Liste so um, dass jeder Knoten mit einem Wert $\geq$ `val` vor jedem Knoten mit einem Wert $<$ `val` kommt. Eine Liste $1 \to 4 \to 3 \to 2 \to 5$ soll also nach einem Aufruf von `split(int val)` mit `val` $= 3$ zum Beispiel die Form $3 \to 4 \to 5 \to 1 \to 2$ haben. Nach der `split`-Operation darf die Reihenfolge der Knoten mit Wert $\geq$ `val` und die Reihenfolge der Knoten mit Wert $<$ `val` beliebig sein.

Hinweis: Verwenden Sie zur Realsierung von `reverse()` und `split()` ausschließlich den in `LinkedListExercise.java` definierten ADT LinkedList und die im ersten Teil realisierten Methoden. Sie können dabei beliebige Hilfsmethoden zur Klasse hinzufügen, dürfen jedoch weder weitere Klassen der Java-Standardbibliothek noch externen Code verwenden. Für `reverse()` und `split()` darf Ihr Algorithmus $\mathcal{O}(n)$ zusätzlichen Speicherplatz benutzen (womöglich um eine temporäre Liste zu bauen). Die volle Punktzahl gibt es jeweils für eine Laufzeit von $\mathcal{O}(n)$. Die Vorlage `LinkedListExercise.java` finden Sie auf der Übungswebseite und auf Moodle. Stellen Sie sicher, dass alle Testfälle in der `main`-Methode der Datei `LinkedListExercise.java` erfolgreich durchlaufen. Achten Sie außerdem auf Randbedingungen und Spezialfälle, die von den Testfällen vielleicht nicht vollstädig abgedeckt werden.
Hinweis zur Abgabe: Ihr Java-Programm muss auf dem Rechner *gruenau2* laufen. Kommentieren Sie Ihren Code, sodass die einzelnen Schritte nachvollziehbar sind. Die Abgabe des von Ihnen modifizierten Source-Codes `LinkedListExercise.java` erfolgt über Moodle.