Algorithmen und Datenstrukturen
Übungsblatt 01

Prof. Karsten Weihe

TECHNISCHE UNIVERSITÄT DARMSTADT

SoSe 2022	v1.2
Themen:	Verzeigerte Strukturen
Relevante Foliensätze:	Kapitel 05+06+07 aus der FOP – finden Sie auch im AuD-Kurs in moodle
Abgabe der Hausübung:	bis 22.04.2022, 23:50 Uhr

Hausübung 01 — Gesamt: 32 Punkte

Verzeigerte Strukturen

Verbindliche Anforderungen für alle Hausübungen:

Das Dokumentieren und Kommentieren Ihres Quelltextes ist nicht verbindlich, wird zum besseren Verständnis Ihrer Lösung jedoch empfohlen. Alle zur Bewertung dieser Hausübung relevanten Deklarationen von Klassen, Methoden (hierzu zählen auch Konstruktoren) und Attributen sind bereits in der Quelltext-Vorlage enthalten und dürfen nicht modifiziert oder entfernt werden. Ihnen steht aber frei, Hilfskonstrukte in Form von weiteren Klassen, Methoden und Attributen zu erstellen, sofern dies nicht explizit auf dem Übungsblatt verboten wurde und Ihre Hilfskonstrukte nicht gegen verbindliche Anforderungen verstoßen. Datenstrukturen und Hilfsmethoden aus der Java-Standardbibliothek sowie Arrays sind nicht erlaubt, sofern dies nicht explizit auf dem Übungsblatt gefordert oder erlaubt wurde. Ihre Methoden müssen auch dann funktionieren, wenn Aufrufe von in der Vorlage deklarierten Methoden (auch von solchen, welche von Ihnen implementiert werden) durch andere, korrekte Implementationen ersetzt werden.

Der Verstoß gegen verbindliche Anforderungen führt zu Punktabzügen und kann die korrekte Bewertung Ihrer Abgabe unter Umständen beeinflussen. Die Implementation einer in der Quelltext-Vorlage deklarierten Methode wird nur bewertet, wenn der mit `TODO` markierte Exception-Wurf entfernt wird.

Hinweise für alle Hausübungen:

Die zu verwendenen Zugriffsmodifizierer sind in der Vorlage bereits gegeben und werden auf dem Übungsblatt nicht immer angegeben. Beachten Sie die Informationen im Moodle-Abschnitt *Technisches und Probe-Übungsblatt*.

Bei Fragen stehen wir Ihnen vorzugsweise im Moodle-Kurs und in den Sprechstunden zur Verfügung.

Die für diese Hausübung in der Vorlage relevanten Verzeichnisse sind `src/main/h01` und `src/test/h01`.

Einleitung

Algorithmen arbeiten auf Datenstrukturen, und Datenstrukturen in der AuD basieren im Wesentlichen auf zwei Konzepten aus der FOP: einerseits wahlfreier Zugriff durch Index oder Schlüsselwert (Arrays, Maps), andererseits verzeigerte Strukturen. Verzeigerte Strukturen haben Sie in Kapitel 07, Folien 116-192 der FOP in Form von linearen Listen kennengelernt. Auf diesem ersten Übungsblatt in der AuD geht es nun um „Fingerübungen“ mit einer etwas komplexeren verzeigerten Struktur: Listen von Listen.

Für Fingerübungen dieser Art ist es zweckmäßig, ***ausnahmsweise*** von der goldenen Regel abzuweichen, dass Implementationsdetails in den `private`-Bereich einer einkapselnden Klasse gehören. Daher werden die folgenden Methoden direkt mit den Listenelementen arbeiten, nicht mit einer Klasse wie `MyLinkedList` aus der FOP. Insbesondere sind bei diesem Übungsblatt alle Klassen, Methoden und Attribute und Methoden `public`. Dies muss aber nicht auf die von Ihnen hinzugefügten Hilfskonstrukte zutreffen.

H1: Listen

Auf diesem Übungsblatt unterscheiden wir zwischen zwei Arten von Listen: *Hauptlisten* und *Einzellisten*: Jedes Listenelement einer Hauptliste verweist in seinem `key`-Attribut auf jeweils eine Einzelliste, und jedes Listenelement einer Einzelliste verweist in unserem Fall in seinem `key`-Attribut auf jeweils ein `Double`-Objekt. Grundlage hierfür ist Klasse `ListItem` aus Kapitel 07, Folien 116-192 der FOP: Elemente, die eine Einzelliste bilden, sind vom Typ `ListItem<Double>`, und die, die eine Hauptliste bilden, sind vom Typ `ListItem<ListItem<Double>>`.

Unter einem Verweis `list` auf eine *korrekt gebildete Liste* vom Schlüsselwerttyp `T` verstehen wir auf diesem Übungsblatt, dass `list` auf den Kopf einer Liste verweist, die, wie in der FOP, aus `ListItem<T>`-Objekten durch Verkettung mittels `next` gebildet ist. Wenn `list` `null` referenziert, sprechen wir bei `list` von einer leeren Liste.

Bei einer Liste von Listen (also `ListItem<ListItem<T>>`) kommt als Bedingung dafür, sie korrekt gebildet zu nennen, hinzu, dass nicht nur die Hauptliste, sondern auch alle Einzellisten korrekt gebildet sind, und dass keine zwei Einzellisten ein `ListItem`-Objekt gemeinsam haben. Die Schlüsselwerte eines Elements einer Hauptliste kann `null` sein. Übersetzt bedeutet das, dass das Element eine leere Liste als Schlüssel hat. Für die Elemente von Einzellisten erlauben wir nicht, dass deren Schlüssel `null` sein können.

H2: Zerlegen von Einzellisten 24 Punkte

In den folgenden Teilaufgaben implementieren Sie in vier Methoden den gleichen Algorithmus mit unterschiedlichen verbindlichen Anforderungen.

Jede der Methoden hat einen Parameter `listOfLists` vom formalen Typ `ListItem<ListItem<Double>>` sowie einen weiteren Parameter `limit` vom formalen Typ `double`. Der Rückgabetyp jeder dieser Methoden ist ebenfalls `ListItem<ListItem<Double>>`. Die gegebene Hauptliste ist stets korrekt gebildet.

Die Rückgabe ist eine korrekt gebildete Liste von Listen und enthält die *selben* `Double`-Referenzen wie `listOfLists` (sollte eine `Double`-Referenz mehrfach in `listOfLists` vorkommen, soll sie auch gleich häufig in der Rückgabe vorkommen). Die Reihenfolge der Referenzen in den Einzellisten soll in der Rückgabe identisch sein. Wir sagen, dass eine Referenz e_1 vor einer Referenz e_2 ist, wenn (1) entweder e_1 und e_2 zur selben Einzelliste gehören und e_1 in dieser Einzelliste vor e_2 kommt oder (2) e_1 und e_2 in unterschiedlichen Einzellisten sind und die Einzelliste von e_1 vor der von e_2 in der Hauptliste kommt.

Der Unterschied zwischen `listOfLists` und der Rückgabe ist folgender: Die Summe jeder Einzelliste in der Rückgabe darf `limit` nicht überschreiten. Das heißt, dass eine Einzelliste $E = (e_1, \ldots, e_n)$ mit n Elementen, deren Summe `limit` überschreitet, in der Rückgabe in $m \leq n$ Einzellisten $E_1, \ldots, E_m$ aufgeteilt ist, wobei Einzellisten $E_1, \ldots, E_m$ nur aus Elementen aus E bestehen. Weiter gilt für zwei (in Einzelliste E aufeinanderfolgenden) Referenzen e_i, e_{i+1}, dass diese in der Rückgabe in zwei aufeinanderfolgende Listen E_k und E_{k+1} aufgeteilt sind, wenn e_{i+1} plus die Summe von E_k `limit` überschreitet.

Wenn eine Einzelliste eine `Double`-Referenz enthält, welche das gegebene `limit` alleine überschreitet, soll eine `RuntimeException` mit folgender Nachricht geworfen werden: `"element at (<i>, <j>) exceeds limit by <n>"`. Hierbei wird `"<i>"` durch die Position der Einzelliste in der Hauptliste, `"<j>"` durch die Position der `Double`-Referenz in der Einzelliste und `"<n>"` durch den Betrag, um den der gegebene Wert das Limit überschreitet, ersetzt. Für i und j wird der erste Position verwendet, an der das Limit überschritten wird.

Anmerkung:

Sie können davon ausgehen, für `double`/`Double` nur Zahlen verwendet werden, also nicht z.B. `Double.NaN`. Weiter testen wir Ihre Implementation nur mit nicht-negativen Einzellisten-Werten und Limits.

Beispiel:

Sei f eine Funktion, die für eine gegebene Liste *list* und ein gegebenes Limit *limit* in Form eines `double`-Werts wie beschrieben auf eine Zerlegung abbildet. Mit $list = ((1.0, 3.0, 5.0), (), (1.0, 2.0, 3.0, 4.0, 5.0))$ und den angebenen Limits erreichen wir mit f das folgende Verhalten:

- $f(list, 15.0) = \cdots = ((1.0, 3.0, 5.0), (), (1.0, 2.0, 3.0, 4.0, 5.0))$
- $f(list, 10.0) = \cdots = f(list, 14) = ((1.0, 3.0, 5.0), (), (1.0, 2.0, 3.0, 4.0), (5.0))$
- $f(list, 9.0) = ((1.0, 3.0, 5.0), (), (1.0, 2.0, 3.0), (4.0, 5.0))$
- $f(list, 6.0) = \cdots = f(list, 8.0) = ((1.0, 3.0), (5.0), (), (1.0, 2.0, 3.0), (4.0), (5.0))$
- $f(list, 5.0) = ((1.0, 3.0), (5.0), (), (1.0, 2.0), (3.0), (4.0), (5.0))$
- $f(list, 4.0) =$ Exception mit $i = 0$, $j = 2$ und $n = 1$

Unbewertete Verständnisfrage:

Wenn Sie zweimal `Double.valueOf` für eine *kleine* Zahl `n` aufrufen, erhalten Sie zweimal *dasselbe* Objekt. Bei *größeren* Zahlen ist das aber nicht unbedingt der Fall. Wieso ist das der Fall? Was müssen Sie beim Schreiben Ihrer Lösung beachten?

H2.1: Einzellisten *as-copy* iterativ zerlegen — 6 Punkte

Zuerst implementieren Sie in Klasse `DoubleListOfListsProcessor` die Klassenmethode `partitionListsAsCopyIteratively`, welche die folgenden verbindlichen Anforderungen erfüllen muss:

Verbindliche Anforderungen:

- Die gegebene Liste und die darin enthaltenen Listen dürfen zu keiner Zeit modifiziert werden (daher *as-copy*).
- Rekursion ist nicht erlaubt.

H2.2: Einzellisten *as-copy* rekursiv zerlegen — 6 Punkte

Nun implementieren Sie in Klasse `DoubleListOfListsProcessor` die Klassenmethode `partitionListsAsCopyRecursively`, welche identisch zu der Methode aus H2.1 ist, außer dass sich die verbindlichen Anforderungen unterscheiden:

Verbindliche Anforderungen:

- Die gegebene Liste und die darin enthaltenen Listen dürfen zu keiner Zeit modifiziert werden (daher *as-copy*).
- Schleifen sind nicht erlaubt.

Hinweis (für `divideListsAsCopyRecursively` und `divideListsInPlaceRecursively`):

Sobald Sie von der Korrektheit Ihrer iterativen Ansätze aus H2.1 und H2.2 überzeugt sind, können Sie diese nutzen, um die in dieser Aufgabe und H2.4 zu implementierenden Methoden schrittweise zu entwickeln: Beispielsweise ist erste Version von `partitionListsAsCopyRecursively` einfach eine Kopie von `partitionListsAsCopyIteratively`. Die beiden Schleifen können Sie nacheinander auf jeweils eine rekursive Hilfsmethode abbilden.

H2.3: Einzellisten *in-place* iterativ zerlegen — 6 Punkte

Implementieren Sie in Klasse `DoubleListOfListsProcessor` die Klassenmethode `partitionListsInPlaceIteratively`, welche identisch zu den Methoden aus Aufgaben H2.1 und H2.2 ist, außer dass sich die verbindlichen Anforderungen unterscheiden:

Verbindliche Anforderungen:

1. Das Erstellen neuer `ListItem`-Objekte für Einzellisten ist nicht erlaubt. Das Erstellen einzelner `ListItem`-Objekte für Hauptlisten ist nur dann erlaubt, wenn die gegebenen `ListItem`-Objekte nicht ausreichen. Daher *in-place*.
2. Rekursion ist nicht erlaubt.

Anmerkung (für `divideListsInPlaceIteratively` und `divideListsInPlaceRecursively`):

Die in dieser Aufgabe zu implementierende Methode und die in H2.4 zu implementierenden Methode dürfen die gegebene Hauptliste sowie die darin enthaltenen Einzellisten auch dann modifizieren, wenn eine Exception geworfen werden muss.

H2.4: Einzellisten *in-place* rekursiv zerlegen — 6 Punkte

Zuletzt implementieren Sie in Klasse `DoubleListOfListsProcessor` die Klassenmethode `partitionListsIn-`

`PlaceRecursively`, welche identisch zu den Methoden aus H2.1, H2.2 und H2.3 ist, außer dass sich die verbindlichen Anforderungen unterscheiden:

Verbindliche Anforderungen:

1. Das Erstellen neuer `ListItem`-Objekte für Einzellisten ist nicht erlaubt.
2. Das Erstellen neuer `ListItem`-Objekte für Hauptlisten ist nur erlaubt, wenn das Limit einer Einzelliste überschritten wird. In diesem Fall muss das `ListItem`-Objekt in der Hauptliste, welches die sodann zerlegte Einzelliste referenziert, aber weiter verwendet werden und der Schlüssel dieses `ListItem`-Objekts darf nicht modifiziert werden.
3. Schleifen sind nicht erlaubt.

H3: IO-Operationen — 8 Punkte

Verbindliche Anforderungen:

Für diese Aufgabe dürfen Sie Arrays und Objektmethoden von `String` verwenden.

H3.1: Liste von Listen auf Textdatensenke hinausschreiben — 4 Punkte

Implementieren Sie in Klasse `DoubleListOfListsProcessor` die rückgabelose Klassenmethode `write`, welche einen ersten Parameter `writer` vom formalen Typ `java.io.Writer` (Kapitel 08 der FOP, Folien 154-156) und einen zweiten Parameter `listOfLists` vom formalen Typ `ListItem<ListItem<Double>>` hat.

Die Methode `write` schreibt die Einzellisten und die in diesen enthaltenen Dezimalzahlen (mit Darstellung von `Double.toString`) in der Reihenfolge in `writer`, in der sie in der Hauptliste beziehungsweise Einzelliste auftreten. Zwei Einzellisten werden mittels eines Zeilenumbruchs (Darstellung mittels `"\n"`) getrennt. Ein einzelnes `"#"` dient zur Trennung zweier Dezimalzahlen und zur Darstellung von leeren Einzellisten. Im Fall, dass die Hauptliste leer ist, soll in `write` keine Schreiboperation ausgeführt werden.

Anmerkung (für `read` und `write`):

Die Methoden dürfen davon ausgehen, dass `writer` bzw. `reader` korrekt mit einer Textdatensenke bzw. Textdatenquelle verbunden sind und dass es sich bei `listOfLists` um eine korrekt gebildete Liste von Listen handelt. Exceptions, die in `write` und `read` entstehen können, dürfen ignoriert werden.

Beispiel:

Eine für die im Beispiel von H2 gegebene Hauptliste erstellte Textdatei würde wie folgt aussehen:

```
1.0#3.0#5.0
#
1.0#2.0#3.0#4.0#5.0
```

H3.2: Liste von Listen aus Datei einlesen — 4 Punkte

Zuletzt implementieren Sie in Klasse `DoubleListOfListsProcessor` die Klassenmethode `read`, welche einen Parameter `reader` vom formalen Typ `BufferedReader` (Kapitel 08 der FOP, Folien 144-153) und Rückgabetyp `ListItem<ListItem<Double>>` hat.

Das Format der von `read` gelesenen Datei ist gleich dem in H3.1 beschriebenen Format. Sie können davon ausgehen, dass die einzulesende Datei nur von einer korrekt funktionierenden Methode `write` geschrieben wurde. Beachten Sie, dass eine leere Datei valide ist und eine leere Hauptliste darstellt.

> **Unbewertete Verständnisfrage:**
>
> Warum sollen Sie `Writer` und `BufferedReader` und nicht beispielsweise einfach den Name der Datei als String verwenden?

H4: Tests

> **Hinweis:**
>
> Auch wenn Sie für das Erstellen von Tests keine Punkte erhalten: Das Teilen jeglicher Tests zur Hausübung (auch solcher, welche nicht auf Basis dieser Aufgabe erstellt wurden) ist nicht erlaubt. Unser Ziel ist, die Verbreitung fehlerhafter Tests in Ihrem Sinne zu verhindern.
>
> Die von uns bereitgestellten Public Tests überprüfen nur einen kleinen Teil Ihrer Implementation. Erfüllt Ihre Lösung nicht alle Public Tests, erhalten Sie auf keinen Fall die volle Punktzahl. Im Umkehrschluss bedeutet dies aber nicht, dass Sie die volle Punktzahl erhalten, wenn Ihre Lösung alle Public Tests besteht.
>
> Der folgende Leitfaden dient als Unterstützung zum Aufbau Ihrer eigenen Tests. Sie können vom Leitfaden abweichen und dabei mindestens genauso aussagekräftige Testergebnisse erzeugen.

Schreiben Sie für die von Ihnen in `DoubleListOfListsProcessor` implementierten Methoden JUnit-Tests.

Dafür erstellen Sie mindestens zehn Listen von Listen von `Double`, wobei die Hauptliste in jeder davon mindestens 100 Elemente enthält. Mindestens 50 der Einzellisten enthält jeweils mindestens 100 Elemente. Mindestens eine Einzelliste hat genau ein Element. Sie wählen `limit` zu jeder Liste von Listen so, dass mindestens 50 Einzellisten Gesamtsummen bis zu `limit` und mindestens 50 Einzellisten Gesamtsummen über `limit` haben. Die genauen Längen und die einzelnen `double`-Werte generieren Sie zufällig.

Mit jeder so generierten Liste von Listen überprüfen Sie, ob die Rückgabe die Spezifikation in allen Details erfüllt (das umfasst auch den Vergleich mit der Eingabeliste bezüglich der Reihenfolge der `Double`-Referenzen).

Erweitern Sie diese Tests, indem Sie jede generierte Liste von Listen mit der Methode aus H3 in eine Datei schreiben, mit der Methode aus H3 eine Kopie erstellen und jede Kopie mit dem Original wie oben vergleichen: dieselbe Länge der Hauptliste und dieselbe Länge jeder Einzelliste.

Darüber hinaus testen Sie für jede Methode aus H2, ob die zu erwartenden Exceptionwürfe tatsächlich geschehen und auch jeweils korrekt sind.

Schließlich testen Sie alle Methoden aus H2 analog zu den obigen Tests noch mit einer leeren Hauptliste.