Funktionale und objektorientierte Programmierkonzepte
Übungsblatt 10

TECHNISCHE UNIVERSITÄT DARMSTADT

Entwurf

Achtung: Dieses Dokument ist ein Entwurf und ist noch nicht zur Bearbeitung/Abgabe freigegeben. Es kann zu Änderungen kommen, die für die Abgabe relevant sind. Es ist möglich, dass sich **alle** Aufgaben noch grundlegend ändern. Es gibt keine Garantie, dass die Aufgaben auch in der endgültigen Version überhaupt noch vorkommen und es wird keine Rücksicht auf bereits abgegebene Lösungen genommen, die nicht die Vorgaben der endgültigen Version erfüllen.

Hausübung 10

Gesamt: 32 Punkte

Mengen

Beachten Sie die Seite *Verbindliche Anforderungen für alle Abgaben* im Moodle-Kurs.

Verstöße gegen verbindliche Anforderungen führen zu Punktabzügen und können die korrekte Bewertung Ihrer Abgabe beeinflussen. Sofern vorhanden, müssen die in der Vorlage mit `TODO` markierten `crash`-Aufrufe entfernt werden. Andernfalls wird die jeweilige Aufgabe nicht bewertet.

Die für diese Hausübung relevanten Verzeichnisse sind `src/main/java/h10` und ggf. `src/test/java/h10`.

Einleitung

In diesem Übungsblatt werden Sie die bekannten Mengenoperationen (Schnitt und Differenz), sowie das Erstellen einer Teilmenge und des Kartesischen Produktes implementieren.

Eine Menge ist dabei realisiert als eine verkette Liste, welche aus Objekten der Klasse `ListItem<T>` besteht, wie bereits aus Kapitel 07 der FOP bekannt sein sollte.

Sie arbeiten in diesem Übungsblatt in den beiden Klasse `MySetAsCopy<T>` und `MySetInPlace<T>`, welche von der Klasse `MySet<T>` erben.

Die Klasse `MySet<T>` repräsentiert eine paarweise geordnete Menge, die einen `protected` Objektattribut `head` vom Typ `ListItem<T>`, sowie eine `protected`-Objektkonstante `cmp` vom Typ `Comparator<? super T>`.

Das Attribut `head` repräsentiert dabei den Kopf der Liste, also das erste Element der Liste bzw. Menge und der `Comparator` repräsentiert einen Vergleichsoperator, welcher zwei Objekte vom Typ `T` vergleichen kann.S

Alle Methoden, welche Sie in dieser Übung implementieren, werden Sie einmal *in-place* und einmal *as-copy* erstellen.

Verbindliche Anforderungen für die gesamte Hausübung:

(i) *In-place* bedeutet, dass Sie die **exakt** selben Objekte benutzen, auf welchen Sie arbeiten, also keine neuen Objekte mittels `new` erstellen. Das `key`-Attribut eines `ListItem<T>`-Objektes darf dabei **nicht** verändert werden. Jegliche Veränderung der Liste geschieht ausschließlich über das Überschreiben der `next`-Referenzen eines `ListItem<T>`-Objektes.

(ii) *As-copy* im Gegensatz zu *in-place* bedeutet, dass Sie neue Objekte mittels `new` erzeugen und keine bereits zuvor existierende Objekte verändern, welche das gleiche `key`-Attribut besitzen. Für jede Aufgabe, welche *as-copy* arbeitet gilt, dass kein `ListItem<T>`-Objekt verändert werden darf, also weder das `next`- noch das `key`-Attribut überschieben werden darf.

(iii) Für das Vergleichen von Objekten vom Typ `T` dürfen Sie ausschließlich den `cmp` verwenden.

(iv) Es dürfen **keine** Datenstrukturen aus der Java-Standardbibliothek verwendet werden. Die Aufgaben sollen ausschließlich mit den gegebenen Datenstrukturen (`ListItem<T>` und `MySet<T>`) gelöst werden.

H1: Teilmenge erstellen 4 Punkte

Um Sie mit dem Konzept von verzeigerten Strukturen vertraut und den Unterschied zwischen in-place und as-copy klar zu machen, werden Sie in dieser Aufgabe eine Teilmenge von der Menge bilden, auf der Sie arbeiten.

H1.1: `subset` *as-copy* 2 Punkte

Implementieren Sie in dieser Aufgabe die `public`-Objektmethode `subset` in der Klasse `MySetAsCopy<T>`. Die Methode besitzt als formalen Parameter `pred`, vom Typ `Predicate<? super T>`, und hat als Rückgabetyp `MySet<T>`. Die Rückgabe der Methode soll ein `MySetAsCopy<T>`-Objekt mit dem selben `Comparator` sein, welches so aufgebaut ist, dass nur `ListItem<T>`-Objekte, für dessen die `test` Methode des Prädikats `pred` den Wert `true` für das `key`-Attribut zurückliefert, enthalten sind.

Die Methode soll dabei, wie vorgegeben, *as-copy* sein, also es sollen nicht die selben Objekte enthalten sein in der Rückgabe, sondern Kopien der `ListItem<T>`-Objekte, auf welchen Sie arbeiten.

In der Abbildung 1 sehen Sie ein Beispiel für die Methode `subset` *as-copy*. Die Ergebnismenge ist eine neue Menge, welche nur die Elemente enthält aus der Eingabemenge, für die das Prädikat $(x \mod 2) = 0$ den Wert `true` zurückliefert.

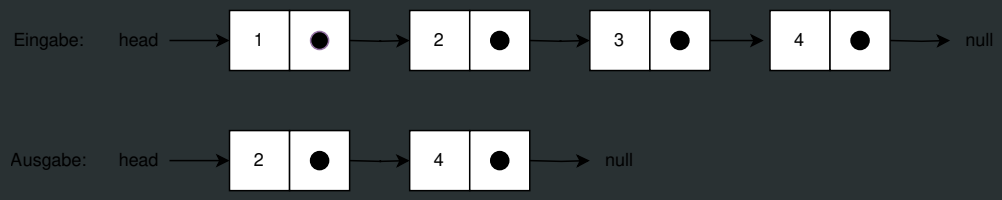

Abbildung 1: Beispiel für die Methode `subset` *as-copy* mit dem Prädikat $(x \mod 2) = 0$

Verbindliche Anforderungen:

(i) Die Eingabemenge darf nicht verändert werden.

(ii) Die `ListItem<T>`-Objekte in der Ergebnismenge müssen neu erstellt werden.

(iii) Die Eingabemenge darf nur einmal durchlaufen werden.

H1.2: `subset` *in-place* 2 Punkte

Implementieren Sie in dieser Aufgabe die `public`-Objektmethode `subset` in der Klasse `MySetInPlace<T>`. Die Methode besitzt als formalen Parameter `pred`, vom Typ `Predicate<? super T>`, und hat als Rückgabetyp `MySet<T>`. Die Rückgabe der Methode soll ein `MySetInPlace<T>`-Objekt sein, welches nur `ListItem<T>`-Objekte enthält, für den die `test`-Methode des Prädikats `pred` für das `key`-Attribut den Wert `true` zurückliefert.
Wie die Methode bereits vorgibt, soll die Methode *in-place* sein, also sollen die **exakt** selben `ListItem<T>`-Objekte in der Rückgabe enthalten sein, auf denen Sie arbeiten.

In der Abbildung 2 sehen Sie ein Beispiel für die Methode `subset` *in-place*. Die Ergebnismenge ist eine neue Menge, welche nur die Elemente enthält, für die das Prädikat $(x \mod 2) = 0$ den Wert `true` zurückliefert, wobei wir hier die Referenzen der Eingabemenge verändern. Die grauen Pfeile repräsentieren die alten Referenzen vor dem Aufruf der Methode, die gestrichelten Pfeile die neuen Referenzen nach dem Aufruf der Methode.

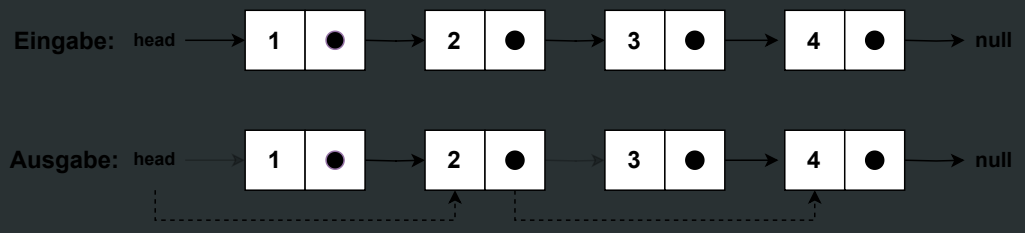

Abbildung 2: Beispiel für die Methode `subset` *as-copy* mit dem Prädikat $(x \mod 2) = 0$

Verbindliche Anforderungen:

(i) Die `ListItem<T>`-Objekte in der Ergebnismenge sollen exakt auf die selben `ListItem<T>`-Objekte referenzieren.

(ii) Die Eingabemenge wird ggf. verändert.

(iii) Die Eingabemenge darf nur einmal durchlaufen werden.

(iv) Die Ergebnismenge gibt die aktuell veränderte Menge zurück.

H2: Kartesisches Produkt von zwei Mengen 6 Punkte

In dieser Aufgabe werden Sie das Kartesische Produkt zweier Mengen implementieren.

> **Erinnerung:**
>
> Das Kartesische Produkt zweier Mengen M und N ist wie folgt definiert:
>
> $$M \times N := \{(x, y) \mid x \in M \land y \in N\}$$

H2.1: cartesianProduct *as-copy* 3 Punkte

Implementieren Sie die `public`-Objektmethode `cartesianProduct` in der Klasse `MySetAsCopy`. Die Methode besitzt als formalen Parameter `other` vom Typ `MySet<T>`. Der Rückgabetyp der Methode ist `MySet<ListItem<T>>`.

Die Rückgabe der Methode soll so aufgebaut sein, dass Sie ein neues `MySet`-Objekt zurückliefern, welches als Elemente `ListItem<ListItem<T>>` besitzt. Ein Objekt vom Typ `ListItem<ListItem<T>>` repräsentiert dabei ein Tupel, wobei der `key` des erste `ListItem<T>`-Objekts aus der Menge stammt, auf welcher Sie arbeiten, also $m \in M$. Der key des zweite `ListItem<T>`-Objekts kommt entsprechend aus dem aktualen Parameter.
Abschließend müssen Sie den richtigen `Comparator` der Menge, welche Sie zurückliefern, übergeben. Dieser soll lediglich das `key`-Attribut zweier `ListItem`-Objekte vergleichen.

H2.2: cartesianProduct *in-place* 3 Punkte

Implementieren Sie die `public`-Objektmethode `cartesianProduct` in der Klasse `MySetInPlace<T>`. Die Methode besitzt als formalen Parameter `other` vom Typ `MySet<T>`. Der Rückgabetyp der Methode ist `MySet<ListItem<T>>`.

Die Rückgabe der Methode soll dabei so aufgebaut sein, dass Sie ein neues `MySet<ListItem<T>>`-Objekt zurückliefern. Dieses enthält Elemente vom Typ `ListItem<ListItem<T>>`. Auch hier repräsentiert ein Objekt vom Typ `ListItem<ListItem<T>>` ein Tupel, welches genau so aufgebaut ist, wie in der vorherigen Aufgabe beschrieben.
Der Unterschied dieser Methode im Vergleich zur Methode davor liegt darin, dass in der Rückgabe dieser Implementation, **exakt** die selben `ListItem<T>`-Objekte verwendet werden sollen, auf denen Sie arbeiten. Das bedeutet, dass Sie keine neuen `ListItem<T>`-Objekte erzeugen, sondern mittels Überschreiben der `next`-Verweise, die Liste der Rückgabe aufbauen.

Denken Sie auch hier daran, einen neuen `Comparator` zur Erstellen für die Rückgabe, welcher genau das selbe machen soll, wie in der Aufgabe davor beschrieben.

H3: Differenz von zwei Sets 10 Punkte

In dieser Aufgabe werden Sie die Mengenoperation Differenz von zwei Mengen implementieren. Einmal *in-place* und einmal *as-copy*.

> **Erinnerung:**
>
> Der Schnitt von zwei Mengen M und N ist wie folgt definiert:
>
> $$M \setminus N := \{x \mid x \in M \land a \notin N\}$$

Beachten Sie, dass die Mengen, entgegen der formalen Definition von Mengen, sortiert sind. Verschieben Sie entsprechend die Referenzen auf den `ListItem<T>`-Objekte gegeben den Fällen, welche eintreten könnten beim Vergleich zweier Elemente.

H3.1: Difference *as-copy* — 5 Punkte

Implementieren Sie die `public`-Objektmethode `difference` in der Klasse `MySetAsCopy<T>`, welche einen formalen Parameter `other` vom Typ `MySet<T>` besitzt und als Rückgabetyp `MySet<T>` besitzt.

Die Rückkgabe der Methode soll dabei wie folgt aufgebaut sein. In der Rückgabe sollen Kopien aller `ListItem<T>`-Objekte enthalten sein, welche in dem `MySet<T>`-Objekt enthalten sind, auf dem die Methode aufgerufen wird, außer den `ListItem<T>`-Objekten der Menge im aktualen Parameter.

H3.2: Difference *in-place* — 5 Punkte

Implementieren Sie in dieser Aufgabe die `public`-Objektmethode `difference` in der Klasse `MySetInPlace<T>`. Die Methode hat einen formalen Parameter `other` vom Typ `MySet<T>` und besitzt den Rückgabetyp `MySet<T>`.

Die Rückgabe soll so aufgebaut sein, dass alle `ListItem<T>`-Objekte des `MySet<T>`-Objektes, auf welchem die Methode aufgerufen wird, enthalten sein soll, außer den Elementen, welche im aktualen Parameter `other` enthalten sind. Die Methode ist dabei *in-place*, was bedeutet, dass **exakt** die selben `ListItem<T>`-Objekte verwendet werden sollen, auf denen Sie arbeiten, und keine neuen `ListItem<T>`-Objekte erzeugt werden sollen.

H4: Schnitt von mehreren Sets — 12 Punkte

In dieser Aufgabe werden Sie die Mengenoperation Schnitt von mehreren Mengen implementieren Einmal *in-place* und einmal *as-copy*.

> **Erinnerung:**
>
> Der Schnitt von Mengen ist wie folgt definiert:
> Für zwei Mengen M und N:
>
> $$M \cap N := \{x \mid x \in M \land x \in N\}$$
>
> Für mehrere Mengen:
>
> $$M_1 \cap \cdots \cap M_n := M \cap M_n \text{ mit } M = M_1 \cap \cdots \cap M_{n-1}, \text{falls } n > 1$$

Achten Sie auch hier in beiden Methoden darauf, dass die Mengen, im Gegensatz zur formalen Definition, sortiert sind. Verschieben Sie entsprechend die Zeiger auf den `ListItem<T>`-Objekte gegeben den Fällen, welche eintreten könnten beim Vergleich zweier Elemente.

H4.1: Intersection *as-copy* — 6 Punkte

Implementieren Sie nun die `public`-Objektmethode `intersectionListItems` in der Klasse `MySetAsCopy<T>`. Die Methode hat als formalen Parameter `others` vom Typ `ListItem<MySet<T>>` und als Rückgabetyp `MySet<T>`.

Die Methode soll den Schnitt aller `MySet<T>`-Objekten aus der Liste `others` mit dem `MySet<T>`-Objekt bilden, auf dem die Methode aufgerufen wird.

Die Methode arbeitet dabei *as-copy*, also sollen wieder in der Rückgabe nur Kopien der entsprechenden `ListItem<T>`-Objekte enthalten sein, auf denen Sie arbeiten.

H4.2: Intersection *in-place* 6 Punkte

Implementieren Sie die `public`-Objektmethode `intersection` in der Klasse `MySetInPlace<T>`. Die Methode besitzt einen formalen Parameter `others` vom Typ `ListItem<MySet<T>>`, und hat als Rückgabetyp `MySet<T>`.

Die Methode soll den Schnitt aller `MySet<T>`-Objekten aus der Liste `others` mit dem `MySet<T>`-Objekt bilden, auf welcher die Methode aufgerufen wird.

Achten Sie darauf, dass die Rückgabe ein `MySet<T>`-Objekt sein soll, welches die **exakt** selben Elemente enthält, auf welchen Sie arbeiten.