Funktionale und objektorientierte Programmierkonzepte
Übungsblatt 10

Prof. Karsten Weihe

Übungsblattbetreuer:	Lars Waßmann und Nhan Huynh
Wintersemester 23/24	v1.0.1
Themen:	Verzeigerte Strukturen
Relevante Foliensätze:	06 und 07
Abgabe der Hausübung:	19.01.2024 bis 23:50 Uhr

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 verkettete Liste, bestehend aus Objekten der Klasse `ListItem<T>`, wie bereits aus Kapitel 07 der FOP bekannt sein sollte.

Sie werden in diesem Übungsblatt in den beiden Klassen `MySetAsCopy<T>` und `MySetInPlace<T>` arbeiten, welche von der Klasse `MySet<T>` erben. Die Klasse `MySet<T>` repräsentiert eine paarweise geordnete Menge, die ein `protected` Objektattribut `head` vom Typ `ListItem<T>` und eine `protected`-Objektkonstante `cmp` vom Typ `Comparator<? super T>` besitzt.

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

Folgende Eigenschaften gelten:

(1) Eine Menge mit n besitzt n `ListItem<T>`-Objekte, d.h. eine leere Menge besitzt 0 `ListItem<T>`-Objekte (`head = null`).

(2) Falls $x = y$ für $x, y \in M$ gilt, gibt `cmp` den Wert 0 zurück.

(3) Falls $x < y$ für $x, y \in M$ gilt, gibt `cmp` eine negative Zahl zurück.

(4) Falls $x > y$ für $x, y \in M$ gilt, gibt `cmp` eine positive Zahl zurück.

Alle Methoden, die 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 denen 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, die *as-copy* arbeitet, gilt, dass kein `ListItem<T>`-Objekt verändert werden darf, also weder das `next`- noch das `key`-Attribut überschrieben 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.

Hinweis:

Sie können in dieser Übung davon ausgehen, dass alle Elemente in der Menge (Schlüsselwerte eines `ListItem<T>`) und alle Parameter nicht `null` sind.

H1: Teilmenge erstellen ?? Punkte

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

Verbindliche Anforderung:

Die Eingabemenge darf nur einmal durchlaufen werden.

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

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

Die Methode soll, gemäß der Vorgabe, *as-copy* sein. Das bedeutet, dass die Rückgabemenge Kopien der `ListItem<T>`-Objekte enthält, auf denen Sie arbeiten.

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

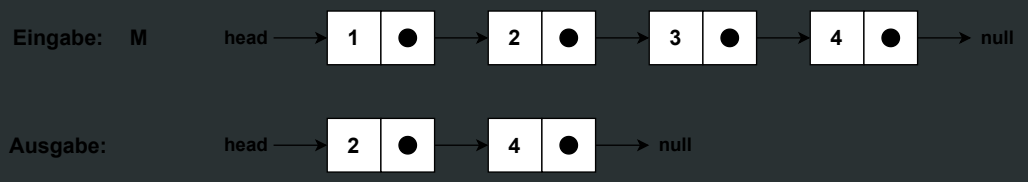

Abbildung 1: Beispiel für die Methode `subset` *as-copy* mit dem Prädikat $(x \mod 2) = 0$

Verbindliche Anforderungen:

(i) Die Eingabemenge darf nicht verändert werden.

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

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

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

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

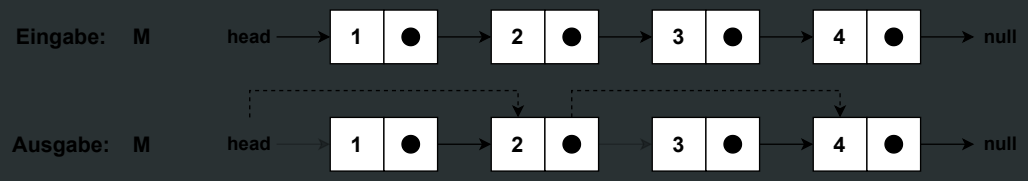

Abbildung 2: Beispiel für die Methode `subset` *in-place* mit dem Prädikat $(x \mod 2) = 0$

Verbindliche Anforderungen:

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

(ii) Die Eingabemenge wird ggf. verändert.

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

H2: Kartesisches Produkt von zwei Mengen ?? 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:

$$M \times N := \{(x, y) \mid x \in M \wedge y \in N\}$$

Verbindliche Anforderungen:

Der Vergleichsoperator für Tupel ist wie folgt definiert: $(x_1, y_1) \leq (x_2, y_2)$ gilt genau dann, wenn $x_1 \leq x_2$. Falls $x_1 = x_2$, wird zusätzlich überprüft, ob $y_1 \leq y_2$ gilt.

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

Implementieren Sie die `public`-Objektmethode `cartesianProduct` in der Klasse `MySetAsCopy<T>`. Die Methode hat 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 ein neues `MySetAsCopy<ListItem<T>>`-Objekt zurückgegeben wird, das `ListItem<ListItem<T>>`-Objekte als Elemente besitzt. Ein Objekt vom Typ `ListItem<ListItem<T>>` repräsentiert dabei ein Tupel. Der `key` jedes `ListItem<ListItem<T>>`-Objekts ist eine verkettete Liste, die immer genau aus zwei `ListItem<T>`-Objekten besteht. Die beiden `key` der `ListItem<T>`-Objekte stellen jeweils ein Tupel $(m, n) \in M \times N$ aus der Ergebnismenge dar. Der `key` des ersten `ListItem<T>`-Objekts stammt dabei aus der Menge M, auf der Sie momentan arbeiten, und der `key` des zweiten `ListItem<T>`-Objekts aus dem aktuellen Parameter.

Abschließend müssen Sie den richtigen `Comparator` der Menge, die Sie zurückliefert, übergeben. Dieser soll lediglich das `key`-Attribut zweier `ListItem`-Objekte vergleichen.

In Abbildung 3 sehen Sie ein Beispiel für die Methode `cartesianProduct` *as-copy*. Die Ergebnismenge ist eine neue Menge, die aus einem Tupel von Elementen aus den Mengen $M \times N$ besteht.

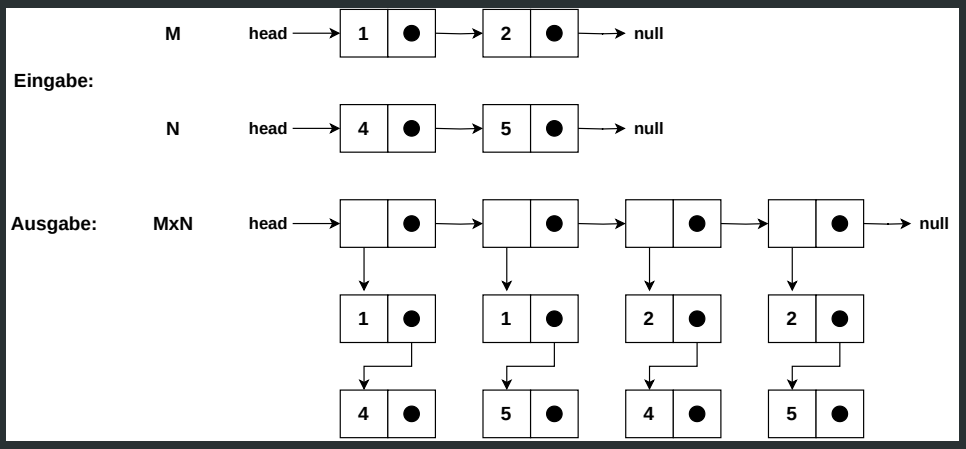

Abbildung 3: Beispiel für die Methode `cartesianProduct` *as-copy*

Verbindliche Anforderungen:

Die Elemente aller Mengen müssen kopiert werden.

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

Implementieren Sie die `public`-Objektmethode `cartesianProduct` in der Klasse `MySetInPlace<T>`. Die Methode hat 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 `MySetInPlace<ListItem<T>>`- Objekt zurückgibt. Dieses enthält Elemente vom Typ `ListItem<ListItem<T>>`. Auch hier repräsentiert ein Objekt vom Typ `ListItem<ListItem<T>>` ein Tupel, das genau so aufgebaut ist, wie in der vorherigen Aufgabe beschrieben.

Der Unterschied dieser Methode im Vergleich zur vorherigen liegt darin, dass in der Rückgabe dieser Implementation die ursprünglichen Elemente der Mengen komplett von den anderen Elementen entkoppelt werden, d.h. die ursprünglichen `ListItem<T>`-Objekte haben keinen Nachfolgerverweis mehr.

Denken Sie auch hier daran, einen neuen `Comparator` zur Erstellung für die Rückgabe zu verwenden, der genau das Gleiche machen soll wie in der vorherigen Aufgabe beschrieben.

In Abbildung 4 sehen Sie ein Beispiel für die Methode `cartesianProduct` *in-place*. Die Ergebnismenge ist eine neue Menge, die aus einem Tupel von den Elementen aus den Mengen $M \times N$ besteht. Die grauen Pfeile repräsentieren die alten Referenzen vor dem Aufruf der Methode, die gestrichelten Pfeile die neuen Referenzen nach dem Aufruf der Methode.

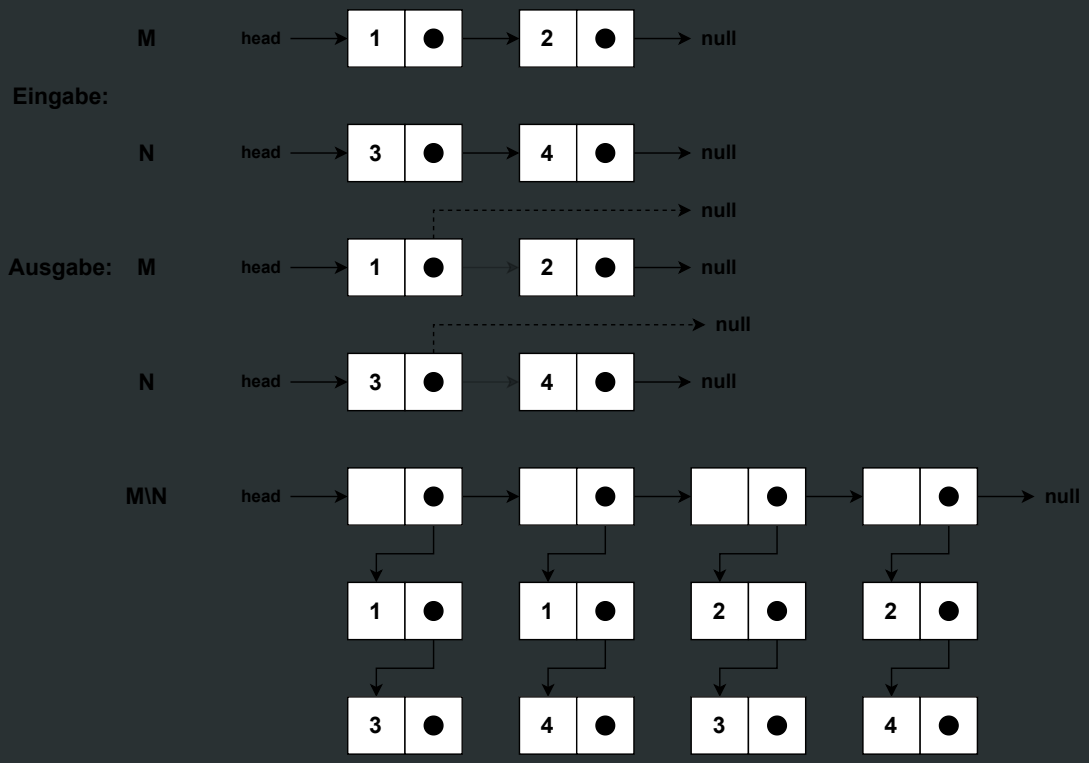

Abbildung 4: Beispiel für die Methode `cartesianProduct` *in-place*

Verbindliche Anforderung:

Wir können das kartesische Produkt leider nicht wirklich *in-place* umsetzen, weshalb Sie in dieser Aufgabe neue `ListItem<T>`- und `ListItem<ListItem<T>>`-Objekte erstellen dürfen.

H3: Differenz von zwei Sets ?? Punkte

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

Erinnerung:

Die Differenz von zwei Mengen M und N ist wie folgt definiert:

$$M \setminus N := \{x \mid x \in M \wedge x \notin N\}$$

Verbindliche Anforderung:

Beachten Sie, dass die Mengen, entgegen der formalen Definition von Mengen, sortiert sind. Das bedeutet, wir können diese Eigenschaft ausnutzen, indem wir die Referenzen auf die `ListItem<T>`-Objekte gegeben den Fällen, die beim Vergleich zweier Elemente auftreten könnten, verschieben.

Konkret heißt das, dass Sie alle Mengen nur einmal durchlaufen werden dürfen.

Hinweise:

(1) Zum Vergleich von Elementen verwenden Sie den `cmp` der aktuellen Menge.

(2) Wie verhalten sich die Laufvariablen, wenn $x = y, x \in M_i, y \in M_j$ für $i \neq j$ gilt?

(3) Wie verhalten sich die Laufvariablen, wenn $x < y, x \in M_i, y \in M_j$ für $i \neq j$gilt?

(4) Wie verhalten sich die Laufvariablen, wenn $x > y, x \in M_i, y \in M_j$ für $i \neq j$ gilt?

H3.1: Difference *as-copy* ?? Punkte

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

Die Rückgabe der Methode soll so aufgebaut sein, dass sie ein neues `MySetAsCopy<T>`-Objekt zurückliefert, das `ListItem<T>`-Objekte als Elemente besitzt. In der Rückgabe sollen Kopien aller `ListItem<T>`-Objekte enthalten sein, die im `MySet<T>`-Objekt enthalten sind, auf dem die Methode aufgerufen wird, außer den `ListItem<T>`-Objekten der Menge im aktuellen Parameter.

In Abbildung 5 sehen Sie ein Beispiel für die Methode `difference` *as-copy*. Die Ergebnismenge ist eine neue Menge, die nur die Elemente enthält, die in der Eingabemenge enthalten sind, aber nicht in der Menge im aktuellen Parameter.

Verbindliche Anforderungen:

(i) Die Eingabemengen dürfen nicht verändert werden.

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

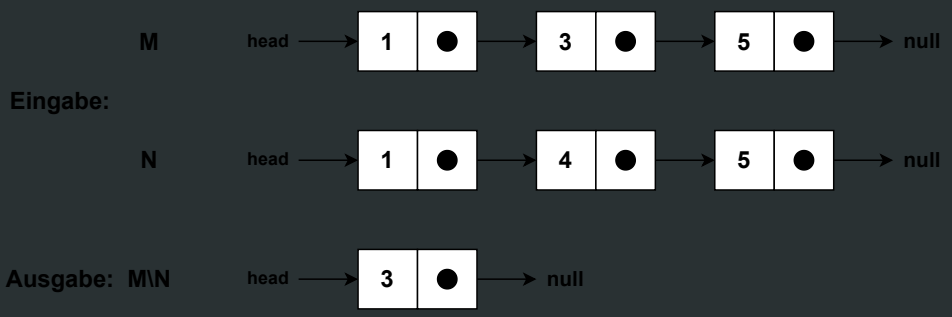

Abbildung 5: Beispiel für die Methode `difference` *as-copy*

H3.2: Difference *in-place* ?? Punkte

Implementieren Sie in dieser Aufgabe die `public`-Objektmethode `difference` in der Klasse `MySetInPlace<T>`. Die Methode hat einen formalen Parameter `other` vomTyp `MySet<T>` und den Rückgabetyp `MySet<T>`. Die Rückgabe der Methode soll so aufgebaut sein, dass sie ein `MySetInPlace<T>`-Objekt zurückliefert, das `ListItem<T>`-Objekte als Elemente besitzt. Die Rückgabe soll so aufgebaut sein, dass alle `ListItem<T>`-Objekte des `MySet<T>`- Objekts, auf dem die Methode aufgerufen wird, enthalten sind, außer den Elementen, die im aktuellen 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.

In Abbildung 6 sehen Sie ein Beispiel für die Methode `difference` *in-place*. Die Ergebnismenge ist eine neue Menge, die nur die Elemente enthält, die in der Eingabemenge enthalten sind, aber nicht in der Menge im aktuellen Parameter. Die grauen Pfeile repräsentieren die alten Referenzen vor dem Aufruf der Methode, die gestrichelten Pfeile die neuen Referenzen nach dem Aufruf der Methode.

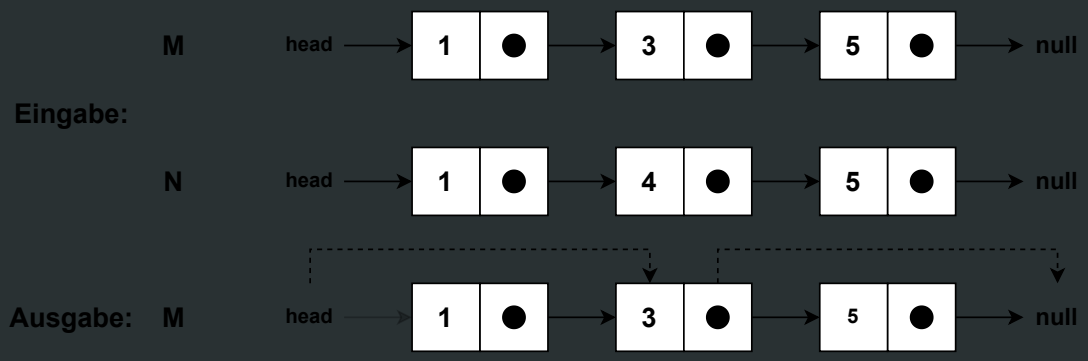

Abbildung 6: Beispiel für die Methode `difference` *in-place*

Verbindliche Anforderungen:

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

(ii) Die Eingabemengen werden ggf. verändert.

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

H4: Schnitt von mehreren Sets ?? 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:

$$M_1 \cap \cdots \cap M_n := M \cap M_n \text{ mit } M = M_1 \cap \cdots \cap M_{n-1}, \text{falls } n > 1$$

Verbindliche Anforderung:

Beachten Sie, dass die Mengen, entgegen der formalen Definition von Mengen, sortiert sind. Das bedeutet, dass wir diese Eigenschaft ausnutzen können, indem wir die Referenzen der Laufvariablen auf die `ListItem<T>`-Objekte in den Fällen, die beim Vergleich zweier Elemente auftreten könnten, „verschieben“.

Konkret heißt das, dass Sie alle Mengen nur einmal durchlaufen werden dürfen.

Hinweise:

(1) Zum Vergleich von Elementen verwenden Sie den `cmp` der aktuellen Menge.

(2) Wie verhalten sich die Laufvariablen, wenn $x = y, x \in M_i, y \in M_j$ für $i \neq j$ gilt?

(3) Wie verhalten sich die Laufvariablen, wenn $x < y, x \in M_i, y \in M_j$ für $i \neq j$gilt?

(4) Wie verhalten sich die Laufvariablen, wenn $x > y, x \in M_i, y \in M_j$ für $i \neq j$ gilt?

H4.1: Intersection *as-copy* ?? Punkte

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

Die Rückgabe der Methode soll so aufgebaut sein, dass sie ein neues `MySetAsCopy<T>`-Objekt zurückliefert, das `ListItem<T>`-Objekte als Elemente besitzt.

Die Methode bildet den Schnitt aller `ListItem<ListItem<T>>`-Objekte aus der Liste `heads`, welche die Mengen `MySet<T>` darstellen.

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

In Abbildung 7 sehen Sie ein Beispiel für die Methode `intersectionListItems` *as-copy*. Die Ergebnismenge ist eine neue Menge, die nur die Elemente enthält, die in allen Mengen enthalten sind.

Verbindliche Anforderungen:

(i) Die Eingabemengen dürfen nicht verändert werden.

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

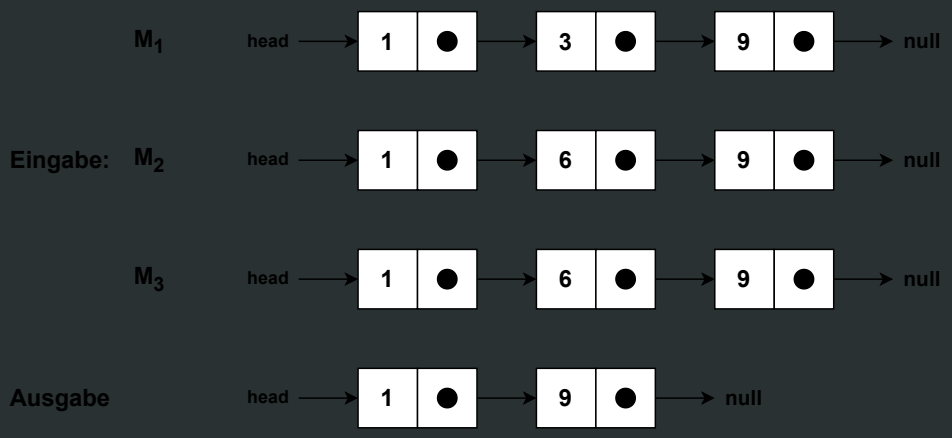

Abbildung 7: Beispiel für die Methode `intersectionListItems` *as-copy*

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

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

Die Rückgabe der Methode soll so aufgebaut sein, dass sie ein `MySetInPlace<T>`-Objekt zurückliefert, das `ListItem<T>`-Objekte als Elemente besitzt.

Die Methode bildet den Schnitt aller `ListItem<ListItem<T>>`-Objekte aus der Liste `heads`, welche die Mengen `MySet<T>` darstellen.

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

In Abbildung **??** sehen Sie ein Beispiel für die Methode `intersectionListItems` *in-place*. Die Ergebnismenge ist eine neue Menge, welche nur die Elemente enthält, welche in allen Mengen enthalten sind. Die grauen Pfeile repräsentieren die alten Referenzen vor dem Aufruf der Methode, die gestrichelten Pfeile die neuen Referenzen nach dem Aufruf der Methode.

Verbindliche Anforderungen:

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

(ii) Die Eingabemengen werden ggf. verändert.

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