Funktionale und objektorientierte Programmierkonzepte Übungsblatt 10

Prof. Karsten Weihe

Übungsblattbetreuer:	Nhan Huynh
Wintersemester 22/23	v1.0
Themen:	Verzeigerte Listenstrukturen
Relevante Foliensätze:	07
Abgabe der Hausübung:	20.01.2023 bis 23:50 Uhr

Hausübung 10 — Gesamt: 25 Punkte

Verzeigerte Strukturen

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 beschäftigen wir uns mit Referenzen und werden eine verzeigerte Listenstruktur implementieren. In der Vorlesung haben Sie bereits die verzeigerte Listenstruktur `LinkedList` kennengelernt, wobei die Verkettung der Elemente mittels `ListItem` dargestellt wird. Ein `ListItem` (Knoten) „umhüllt“ ein einzelnes Element aus der Liste und hat einen Verweis auf seinen direkten Nachfolgeknoten, d.h. wir können über die Nachfolgeknoten immer zum Nachfolgeelement gelangen. Dies stellt die Verkettung einer Liste dar.

Abbildung 1: Eigene Linked List-Klasse auf Basis der Vorlesung

Damit wir besser verstehen, wie eine verzeigerte Struktur funktioniert, werden wir eine eigene verzeigerte Struktur implementieren und `ListItem` als Grundlage verwenden. Wir implementieren in dieser Hausübung eine besondere Art von Liste, die sogenannte *Skip-Liste*[1]. Sie ist ebenfalls generisch mit einem Typparameter `T` und unterscheidet sich von einer normalen Liste dahingehend, dass sie eine sortierte Liste und eine randomisierte Datenstruktur ist.

Wie ist nun eine Skip-Liste aufgebaut und was bedeutet randomisiert in diesem Kontext? Eine Skip-Liste ist eine mehrdimensionale Liste, d.h. eine Liste von Listen. Wir bezeichnen die einzelnen Listen, die die anderen Listen enthalten, als *Ebenen* und die einzelnen Elemente einer Ebene als *Knoten*. Eine Ebene stellt eine sogenannte *Express-Liste* dar, die uns unter Umständen einen schnelleren Zugriff auf die tatsächlichen Elemente ermöglicht. Radomosieirt bedeutet in diesem Kontext, dass das Einfügen eines Elements in einer Ebene auf einer Wahrscheinlichkeit basiert, die wir später noch genauer betrachten werden.

[1] `https://de.wikipedia.org/wiki/Liste_(Datenstruktur)#Skip-Liste`

Eine Skip-Liste L mit Höhe h und w Elementen $e_0, \ldots, e_{n-1}$ von Typ T mit $h, n, w \in \mathbb{N}$ besitzt folgende Eigenschaften:

1. L besteht aus h Ebenen $E_0, \ldots, E_{h-1}$. Dabei wird E_0 als *oberste Ebene* und E_{h-1} als *unterste Ebene* bezeichnet.
2. Jede Ebene E_i besteht aus jeweils $2 \leq \text{count}(E_i) \leq w + 1$ *Express-Knoten* $n_{i,0}, \ldots, n_{i,\text{count}(E_i)}, \ldots$.
   2.1. Der erste Express-Knoten $n_{i,0}$ einer Ebene E_i wird als *Sentinel-Knoten*[2] dieser Ebene bezeichnet.
   2.2. Mit Ausnahme der Sentinel-Knoten referenziert jeder Express-Knoten über das `value`-Attribut jeweils ein Element von Typ T der tatsächlichen Liste. (D.h. das `value`-Attribut eines Sentinel-Knotens ist `null`)
   2.3. Jedes in einer Ebene von einem Express-Knoten referenzierte Element wird auch in der Nachfolgeebene von einem Express-Knoten referenziert. In der letzten Ebene werden alle Elemente referenziert.
3. Alle Sentinels sind sortiert nach Ebene in einer verzeigerten Liste enthalten, die von `head` referenziert wird.
4. Ein Element in einem Listen-Knoten ist ein Express-Knoten, der das tatsächliche Element sowie Verweise auf den Vorgänger-, unteren und oberen Knoten enthält
5. Ein Element wird immer in der untersten Ebene eingefügt und basierend auf einer Wahrscheinlichkeit p in eine höhere Ebene übertragen, d.h. im Durchschnitt hat eine Ebene E_i mit $i \in \{0, \ldots, h-1\}$ ca. $p^k \cdot w$ Elemente, wobei w die Anzahl der Elemente ohne Sentinel-Knoten in der untersten Ebene ist und $k = h - i$.

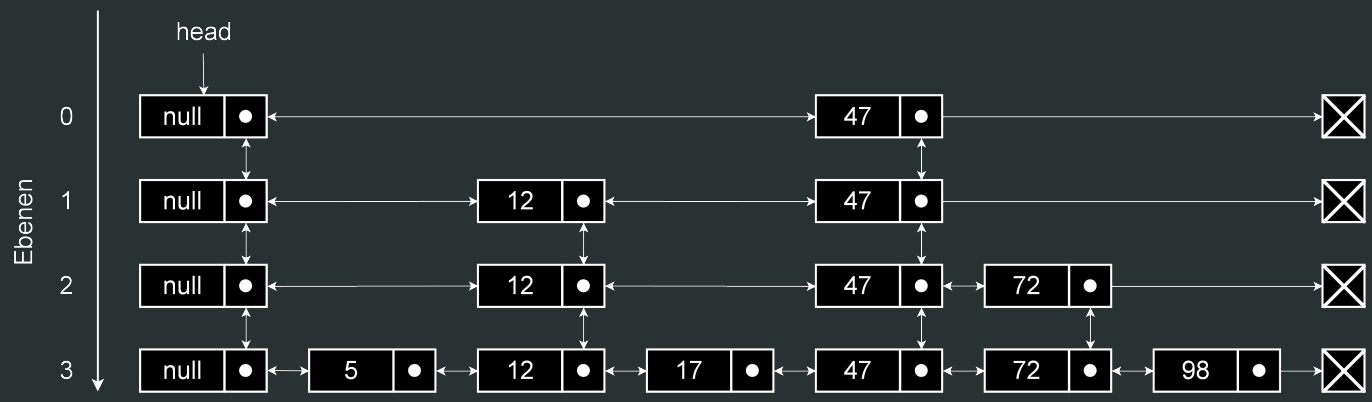

Abbildung 2: Beispiel eigene Skip-List-Klasse auf Basis von ListItem mit Höhe 4, 6 Elementen und $p = 0.5$

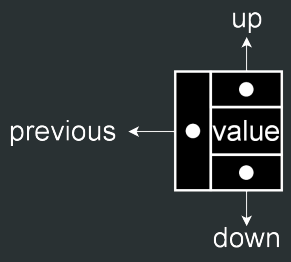

Abbildung 3: Visualisierung von `ExpressNode`

Eine Skip-Liste wird also als `ListItem<ExpressNode<T>>` dargestellt. Wie bereits erwähnt, enthält ein `ExpressNode` das umhüllende Element `value` vom generischen Typ `T` und Verweise zu dem Vorgänger, dem unteren und oberen Knoten namens `prev`, `up` und `down`. Sie sind alle vom Typ `ListItem<ExpressNode<T>>`. Um den Nachfolger zu erhalten, wird der Verweis `next` aus `ListItem` verwendet. Die `SkipList<T>` besitzt also

1. einen Verweis auf dem Kopf `head` der Liste,
2. eine maximale Höhe `maxHeight`,
3. eine Wahrscheinlichkeit `probability`, die angibt, wie wahrscheinlich es ist, dass ein Element in eine höhere Ebene übertragen wird,
4. einen Vergleichsoperator `Comparator<T>`, die die Ordnung der Elemente in der Liste vorgibt.
5. eine aktuelle Höhe `height` und eine aktuelle Anzahl an Elementen `size`.

Für mehr Informationen lesen Sie sich am besten die Java Dokumentation der Klassen `ListItem<T>`, `ExpressNode<T>`, `Probability` und `SkipList<T>` durch.

[2] Hierbei handelt es sich um Dummy-Knoten. Siehe `https://de.wikipedia.org/wiki/Sentinel_(Programmierung)`.

H1: Überprüfen, ob Element in der Liste vorhanden ist ?? Punkte

Implementieren Sie die Methode `contains` in der Klasse `SkipList`. Die Methode `contains` soll überprüfen, ob ein Element in der Liste vorhanden ist und gibt genau dann `true` zurück, wenn das Element in der Liste vorhanden ist. Ansonsten soll `false` zurückgegeben werden. Wir verwenden folgende Strategie, um zu überprüfen, ob ein Element in der Liste vorhanden ist:

1) Wir beginnen auf der obersten Ebene (`head`).

2) Wir prüfen, ob das nächste Element auf der aktuellen Ebene das gesuchte Element ist.

   2.1) Falls ja, geben wir `true` zurück.

   2.2) Falls das nächste Element **kleiner** als das gesuchte Element ist oder nicht existiert, gehen wir vom **aktuellen** Element eine Ebene **tiefer** und wiederholen Schritt 2. Es existiert kein Nachfolger Element, falls wir am Ende der Ebene angekommen sind. (Die Suche auf der nächsten Ebene wird nicht von vorne beginnen, sondern vom aktuellen Element aus!)

   2.3) Falls das nächste Element **größer** als das gesuchte Element ist, gehen wir zum **Nachfolger** des aktuellen Elementes und wiederholen Schritt 2.

3) Die Suche läuft so lange, bis wir auf der **untersten** Ebene angekommen sind. Falls wir auf der untersten Ebene angekommen sind und das gesuchte Element trotzdem **nicht gefunden** wird, geben wir `false` zurück.

In der Abbildung 4 wird ein beispielhafter Ablauf der Methode `contains` dargestellt.

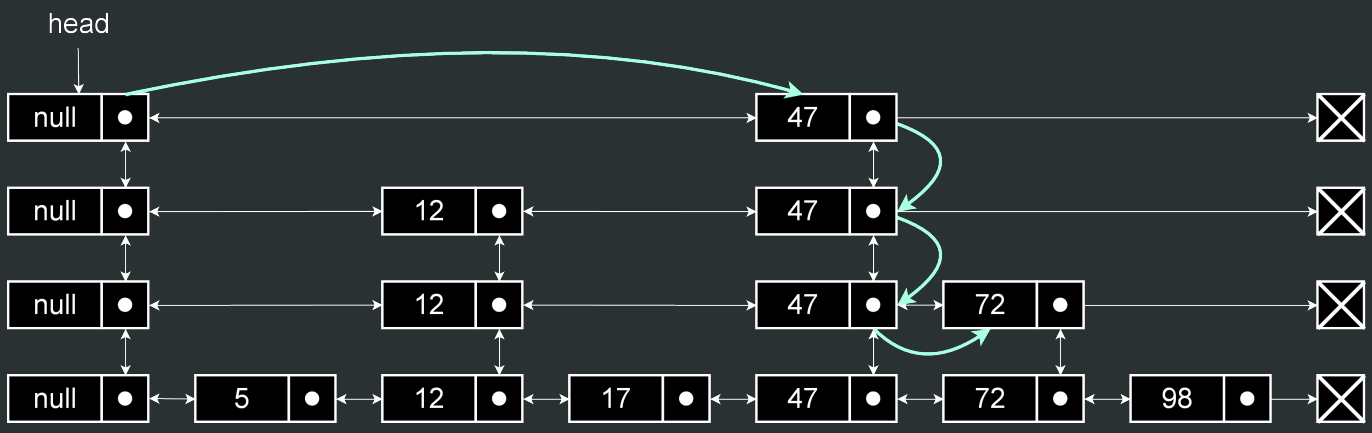

Abbildung 4: Beispiel für die Suche nach dem Element 72

Verbindliche Anforderungen:

i. Die Liste darf nur einmal durchlaufen werden.

ii. Es dürfen keine neuen `ListItem`-Objekte erzeugt werden.

iii. Die Anzahl an Vergleichen für die Suche nach dem Element soll minimal sein. Beispielsweise sind für die Suche nach dem Element 72 sind in Abbildung 4 nur zwei Vergleiche notwendig (72 mit 47 auf Ebene 0 und 72 mit 72 auf Ebene 2).

H2: Einfügen von Elementen ?? Punkte

Implementieren Sie die Methode `add` in der Klasse `SkipList`. Die Methode `add` soll ein Element in die Liste einfügen. Wir verwenden folgende Strategie, um ein Element in die Liste einzufügen:

1) Wir beginnen auf der obersten Ebene (`head`).

2) Wir verwenden die Suchstrategie aus H1, um die passende Einfügeposition auf der untersten Ebene zu finden.

3) Haben wir die passende Einfügeposition gefunden, fügen wir das neue Element in der untersten Ebene hinzu.

4) Nun müssen wir schauen, ob das Element auf einer höheren Ebene eingefügt werden soll.

   4.1) Dazu verwenden wir das Interface `Probability`, das uns vorgibt, ob ein Element auf einer höheren Ebene eingefügt werden soll. Die Methode `nextBoolean` gibt genau dann `true` zurück, falls das Element auf einer höheren Ebene übertragen werden soll.

   4.2) Wiederhole Schritt 4.2 bis die Methode `nextBoolean` `false` zurückgibt.

In der Abbildung 5 wird ein beispielhafter Ablauf der Methode `add` dargestellt. Die durchgezogenen Pfeile visualisieren die Suche nach der passenden Einfügestelle und die gestrichelten Pfeile das Einfügen des Elements.

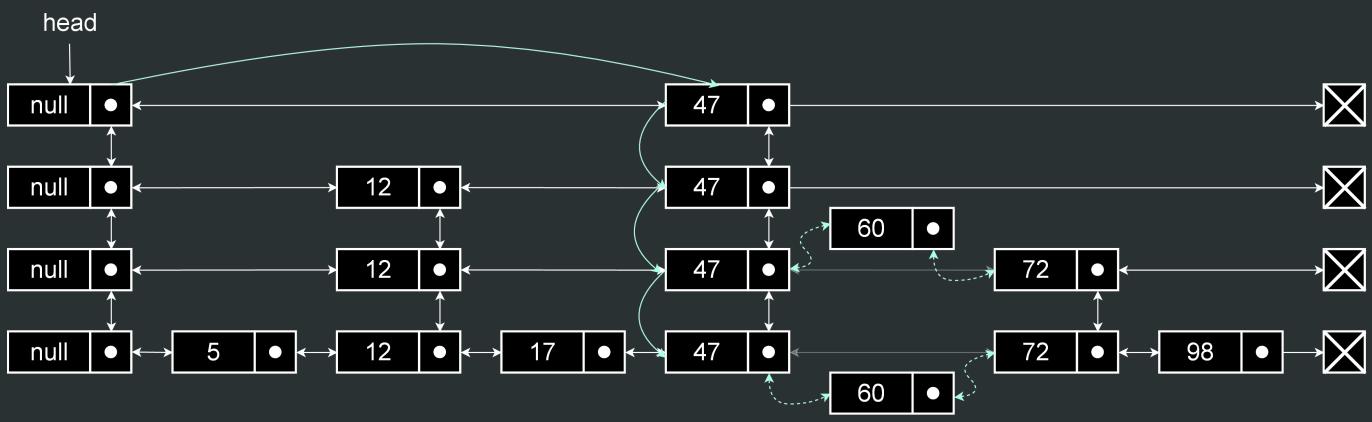

Abbildung 5: Beispiel für das Einfügen des Elements 60

Verbindliche Anforderungen:

i. Die Liste darf nur einmal durchlaufen werden.

ii. Die Anzahl an Vergleichen für die Suche nach der passenden Einfügeposition soll minimal sein. Für das Einfügen des Elements 60 sind bspw. drei Vergleiche notwendig (60 mit 47 auf Ebene 0, 60 mit 72 auf Ebene 2 und 60 mit 72 auf Ebene 3).

iii. Wenn der übergebene `key` bereits in der Liste vorhanden ist, also eine `ExpressNode e` existiert, sodass `cmp.compare(key, e.value)==0` gilt, dann soll der übergebene `key` nach `e` eingefügt werden.

Erinnerung:

- Eine Skip-Liste besitzt eine maximale Höhe, d.h. es dürfen keine neuen Ebenen erzeugt werden, falls die maximale Höhe erreicht wurde.
- Beachten Sie, dass jede Ebene mit einem Sentinel-Knoten beginnt.
- Vergessen Sie ebenfalls nicht die Größe und die aktuelle Höhe der Liste anzupassen!

H3: Entfernen von Elementen ?? Punkte

Implementieren Sie die Methode `remove` in der Klasse `SkipList`. Die Methode `remove` soll das erste Vorkommnis eines Elementes aus allen Ebenen entfernen und verwendet folgende Strategie:

1) Wir beginnen auf der obersten Ebene (`head`).
2) Wir verwenden die Suchstrategie aus H1, um das zu entfernende Element zu finden.
3) Entferne jedes Vorkommen des Elements aus der aktuellen und allen unteren Ebenen.
4) Falls eine Ebene keine Elemente (außer dem Sentinel-Knoten) mehr enthält, muss diese Ebene entfernt werden.

In der Abbildung 6 wird ein beispielhafter Ablauf der Methode `remove` dargestellt. Die durchgezogenen Pfeile visualisieren die Suche nach der passenden Löschposition und die gestrichelten Pfeile das Entfernen des Elements.

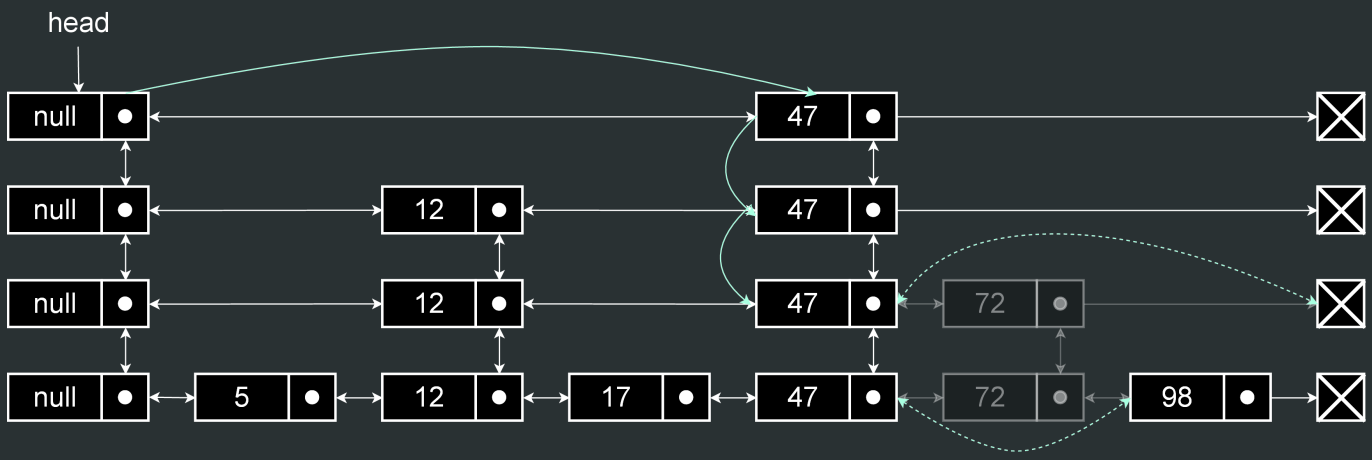

Abbildung 6: Beispiel für das Entfernen des Elements 72

Verbindliche Anforderung:

i. Die Liste darf nur einmal durchlaufen werden.
ii. Es dürfen keine neuen `ListItem`-Objekte erzeugt werden.
iii. Die Anzahl an Vergleichen für die Suche nach der passenden Löschposition soll minimal sein. Für das Entfernen des Elements 72 sind bspw. zwei Vergleiche notwendig (72 mit 47 auf Ebene 0 und 72 mit 72 auf Ebene 2).
iv. Wenn der übergebene `key` mehrfach in der Liste vorhanden ist, also mehrere `ExpressNodes` $e_0, \ldots, e_n$ existieren, sodass `cmp.compare(key,` e_i`.value)==0` für ein $i \in [0, n]$ und $n \in \mathbb{N}$ gilt, dann soll das erste Vorkommen des `key` aus der Liste entfernt werden (also e_0).

Erinnerung:

Vergessen Sie nicht die Größe und die aktuelle Höhe der Liste anzupassen!