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: 22 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 Übungblatt beschäftigen wir uns mit verzeigerten Listenstrukturen. Konkret realisieren wir eine geordnete Liste mittels Knoten (`ListItem`) da, wobei ein Knoten ein einzelnes Element aus der Liste enthält und einen Verweis auf den Nachfolgerknoten und somit werden die Listenelemente miteinander verkettet.

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

Wir verwenden die Klasse `ListItem` als Grundlage für die Implementierung der Skip-Liste. Im Gegensatz zu `ListItem` ist eine Skip-Liste sortiert. Die Skip-Liste[1] ist folgendermaßen aufgebaut:

1. Jede Ebene enthält als erstes Element einen Sentinel[2]-Knoten, der als Wert `null` hat.
2. Alle Elemente bis auf dem Sentinel-Wert sind ungleich `null`
3. Die nachfolgenden Elemente werden nach dem Sentinel-Knoten angehängt.
4. Ein Element in einem `ListItem` ist ein `ExpressNode` und enthält das Element, Verweise auf den Vorgänger-, unteren und oberen Knoten.
5. Ein Verweis nach oben/unten ist immer ein Verweis auf einen `ExpressNode` mit demselben Element.
6. Die Höhe der Liste entspricht der Anzahl der Ebenen.
7. Die oberste Ebene wird mit dem Verweis `head` angesprochen.
8. Die unterste Ebene enthält alle Elemente der Liste und alle höheren Ebenen enthalten eine Teilmenge der Elemente der untersten Ebene, wobei folgendes gilt:

   Sei h_i die Ebene i, dann enthält h_i maximal so viele Elemente wie h_{i+1}.

Die unterste Ebene der Skip-Liste enthält alle Elemente der Liste und die höhere Ebene enthalten nur eine Teilmenge der Elemente der untersten Ebene. Die höheren Ebene bieten einen schnelleren Zugriff auf die Elemente der untersten Ebene.

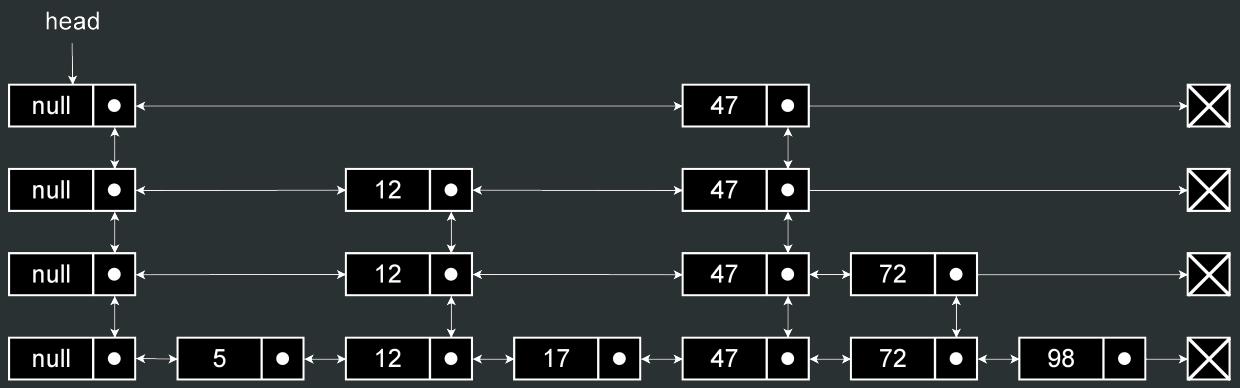

Abbildung 2: Beispiel eigene Skip-List-Klasse auf Basis von ListItem mit Höhe 4 und 6 Elementen

[1] https://de.wikipedia.org/wiki/Liste_(Datenstruktur)#Skip-Liste
[2] 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 3 wird ein beispielhafter Ablauf der Methode `contains` dargestellt.

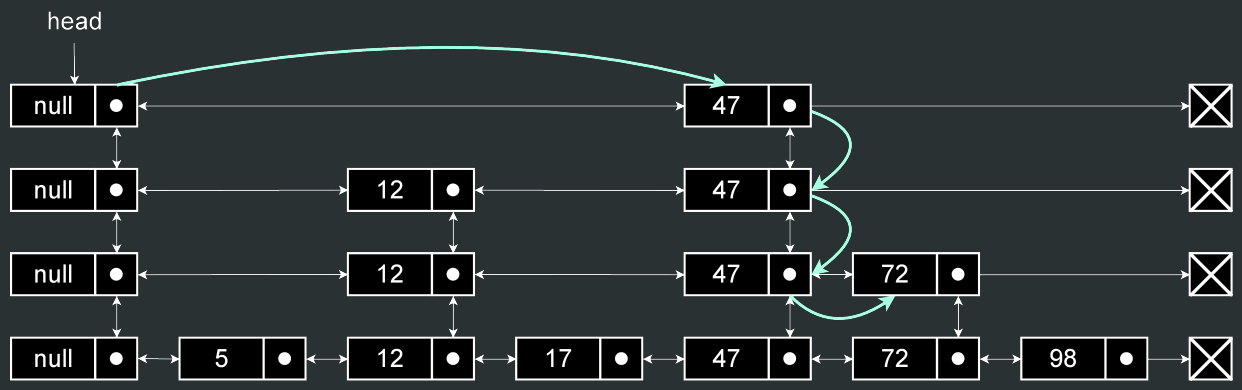

Abbildung 3: Beispiel für die Suche nach dem Element 72

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. Für die Suche nach dem Element 72 sind 2 Vergleiche notwendig.

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 fügen das Element immer auf der untersten Ebene ein, d.h. suche nach der passenden Einfügeposition.

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

4) Dazu verwenden wir das Interface `Probability`, die uns vorgibt, ob ein Element auf einer höheren Ebene eingefügt werden soll. Die Methode `nextBoolean` gibt `true` zurück, wenn das Element auf einer höheren Ebene. Wir wiederholen das solange bis die Methode `nextBoolean` `false` zurückgibt.

In der Abbildung 4 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.

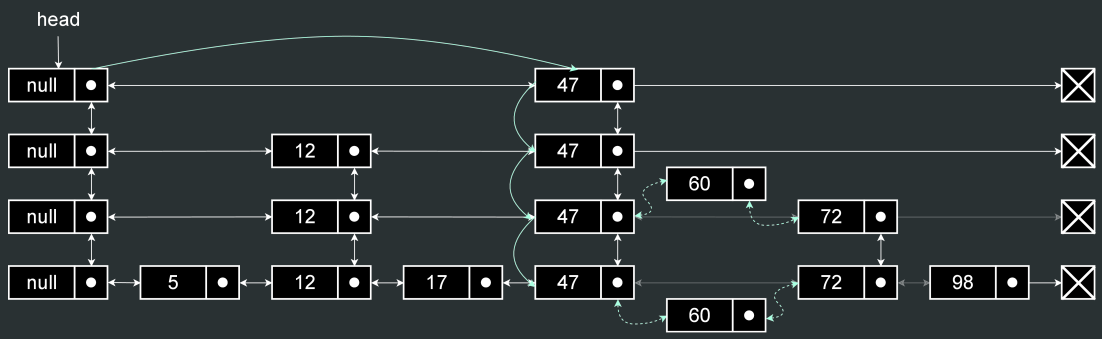

Abbildung 4: Beispiel für das Einfügen des Elements 60

Verbindliche Anforderungen:

i. Die Liste darf nur einmal durchlaufen werden.

ii. Eine Skip-Liste eine maximale Höhe besitzt, d.h. es dürfen keine neuen Ebenen erzeugt werden, falls die maximale Höhe erreicht ist.

iii. Die Anzahl an Vergleichen für die Suche nach der passenden Einfügeposition soll minimal sein. Für das Einfügen des Elements 60 sind 3 Vergleiche notwendig.

Erinnerung:

Beachte, 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 verwenden 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 5 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.

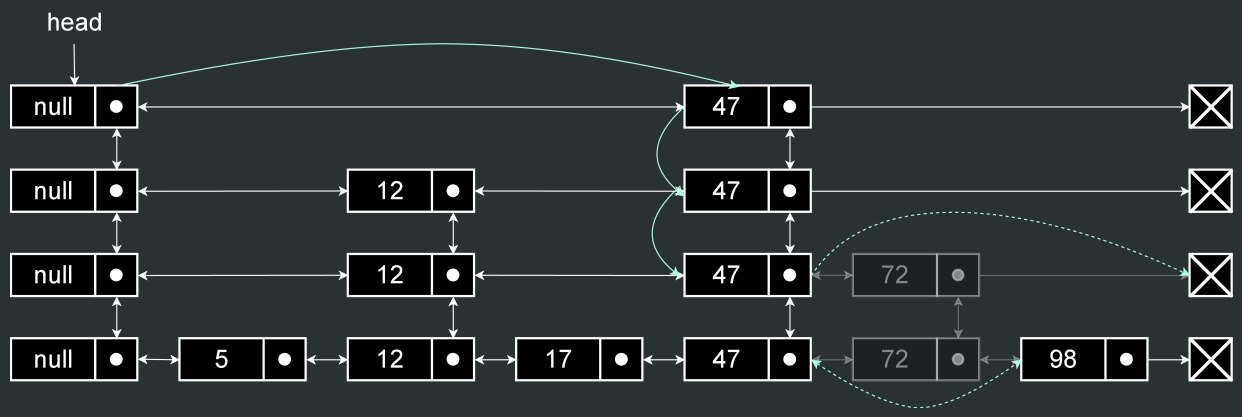

Abbildung 5: Beispiel für das Entfernen des Elements 72

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 2 Vergleiche notwendig.

Erinnerung:

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