Fachhochschule Nordwestschweiz
Hochschule für Technik

Algorithmen und Datenstrukturen 2
Prof. Dr. C. Stamm christoph.stamm@fhnw.ch Tel.: 056 202 78 32

Übung 3

Ausgabe: KW 40
Besprechung: KW 42

LinkedList (Bewertete Gruppenarbeit)

Die `LinkedList` des Java Collections Framework wird in der Praxis wenig verwendet, weil sie nur für Stacks, Queues und Deques gut geeignet ist, aber nicht wirklich für veränderbare Listen, wo Listenelemente an beliebigen Stellen eingefügt oder gelöscht werden können. Das ist untypisch, denn die verkettete Liste bietet sich dort an, wo die Listenlänge im Verlauf des Programms stark variiert.

a) Welches sind die Vor- und Nachteile der `LinkedList` gegenüber der `ArrayList` bezüglich dem Einfügen und Löschen an beliebiger Stelle?

In dieser Übung implementieren Sie in **kleinen Teams** schrittweise eine neue doppelt-verkettete Liste `DLinkedList<E>`, welche sowohl das Interface `List` als auch eine Erweiterung davon (`IList`) implementiert, eine verbesserte Performanz bei den wichtigsten Listenoperationen aufweist und hohe Sicherheit bietet. Bewertet werden Performanz, Sicherheit, Design der Datenstruktur, Code-Qualität und Kompatibilität mit den vorgegebenen Tests.

Wenn Sie das Interface `IList` betrachten, so spielt dort der Datentyp `ListItem` eine zentrale Rolle. Ein ListItem ist eine direkte oder indirekte Referenz auf ein Listenelement der Liste. Darüber können Sie neue Listenelemente einfügen (`addAfter(…)`, `addBefore(…)`), bestehende löschen (`remove(…)`, `delete(…)`) oder auch ganz einfach nur auf die zugehörigen Daten zugreifen (`get(…)`). Die entsprechenden Details sind im Interface beschrieben.

b) Beginnen Sie mit dem Design der Datenstruktur. Überlegen Sie sich, wie Sie dem Benutzer Zugriff auf die Listenelemente gewähren können ohne zu viel an Sicherheit zu verlieren. Wie kann verhindert werden, dass die Benutzerin keine Listenelemente falsch miteinander verkettet? Wie kann verhindert werden, dass ein Listenelement (nicht Datenobjekt) in mehr als einer Liste drin enthalten ist? Wie kann verhindert werden, dass bei einer Listenoperation ein Listenelement als Parameter angegeben wird, welches gar nicht zu dieser Liste gehört?

   Diskutieren Sie diese Fragen im Team und erarbeiten Sie Lösungsansätze.

c) Implementieren Sie Ihre Klasse `DLinkedList<E>`. Erben Sie nur die effizient realisierbaren Methoden der abstrakten Basisklasse `AbstractList<E>`. Teilen Sie sich dann die Methoden aus dem Interface `IList` in der Gruppe auf und implementieren Sie die Ihnen zugewiesenen Methoden. Beachten Sie, dass ein paar Basisoperationen existieren, welche immer wieder benötigt werden und vielleicht zuvor realisiert werden sollen:

   - `linkInFront(ListItem item)`: fügt das existierende `item` am Anfang der Liste ein;
   - `linkInBack(ListItem item)`: fügt das existierende `item` am Ende der Liste ein;
   - `linkInAfter(ListItem prev, ListItem item)`: fügt das existierende `item` nach `prev` in der Liste ein;
   - `unlink(ListItem item)`: entfernt das `item` aus der Liste, löscht das zugehörige Datenobjekt aber nicht.

d) Damit Ihre Liste wie eine normale Liste im Java Collections Framework verwendet werden kann, müssen Sie auch das Java spezifische Iteratorkonzept realisieren, wo der Iterator jeweils zwischen den Listenelementen steht. Implementieren Sie dazu das Interface `IListIterator`, welches `ListIterator` um eine einzige Methode erweitert. Diese zusätzliche Methode ist quasi das Bindeglied zwischen dem Java Iteratorkonzept und dem Konzept der (in)direkten Zugriffe auf die Listenelemente. Als Hilfestellung gebe ich Ihnen zwei von mir bereits realisierte Methoden:

```
public void remove() {
    if (m_curModCount != modCount) throw new ConcurrentModificationException();
    if (m_returned == null) {
       throw new IllegalStateException();
    } else {
       if (m_returned == m_next) {
          m_next = m_returned.m_next;
       } else {
          m_index--;
       }
       DLinkedList.this.remove(m_returned);
       m_returned = null;
       m_curModCount++;
    }
}

public void set(E data) {
    if (m_returned == null) {
       throw new IllegalStateException();
    } else {
        m_returned.m_data = data;
    }
}
```

Bemerkungen zum Code:

- `m_next` ist eine Instanzvariable des Iterators und zeigt auf das Element rechts vom Iterator.
- `m_returned` ist eine Instanzvariable und zeigt auf das Listenelement, welches zuletzt mit `next()` oder `previous()` durchlaufen wurde.
- Die Exception `IllegalStateException` wird dann geworfen, wenn der Iterator nicht in einem gültigen Zustand ist, um zum Beispiel die Methode `remove` auszuführen (studieren Sie dazu die Details in der Java-Dokumentation zu `ListIterator`[1]).
- Es wird davon ausgegangen, dass in der Klasse `DLinkedList` eine Methode mit dem gleichen Namen `remove` vorhanden ist. In diesem Fall kann der Java-Compiler nicht wissen, ob mit `remove` die Methode der inneren Iteratorklasse oder die Methode der äusseren Klasse `DLinkedList` gemeint ist. Auch wenn sich die beiden Methoden anhand der Parameterlisten unterscheiden lassen würden, ist eine explizite Angabe der Klasse in Form von `Class.this.methode` notwendig.
- `modCount` ist ein Zähler in der abstrakten Basisklasse `AbstractList` und `m_curModCount` ist ein entsprechender Zähler des Iterators. Beide zählen, wie viele strukturelle Veränderungen die Liste schon mitgemacht hat. Damit wird das Fail-fast-Prinzip des Java Iterators realisiert. Beim Erzeugen des Iterators wird der Wert von `m_curModCount` mit dem Wert von `modCount` initialisiert. Wird vom Iterator aus eine strukturelle Listenveränderung vorgenommen, so werden beide Zähler um eins erhöht (Im Fall von `remove()` geschieht die Erhöhung von `modCount` in der Methode `DLinkedList.remove()` Methode). Wird hingegen ausserhalb des Iterators eine strukturelle Listenveränderung vorgenommen, so wird nur der `modCount` erhöht. Sobald also die beiden Zählerwerte nicht mehr identisch sind, wird der Iterator für unbrauchbar erklärt, indem er bei der nächsten Iteration eine `ConcurrentModificationException` wirft.

e) Bevor Sie beginnen Ihre Implementierung zu testen, überprüfen Sie im Quellcode nochmals, ob Sie die Preconditions und allenfalls auch ein paar wichtige Postconditions mit Assertions abgesichert haben. Dadurch vermeiden Sie die Verschleppung von Fehlern über Methodengrenzen hinweg und Erleichtern die allfällige Fehlersuche stark!

   Stellen Sie sicher, dass in den mitgelieferten Unit-Tests Ihre Implementierungen getestet werden. In `DLinkedListTest1` wird Ihre Implementierung mit der `LinkedList` verglichen. In `DLinkedListTest2` werden zwei Instanzen Ihrer verketteten Liste für eine Vielzahl von Tests verwendet. Führen Sie die beiden mitgelieferten Unit-Tests aus und nehmen Sie allfällige Korrekturen an Ihrem Code vor.

[1] http://docs.oracle.com/javase/8/docs/api/java/util/ListIterator.html