SE1, Aufgabenblatt 11

Softwareentwicklung I – Wintersemester 2012/13

Listen implementieren

MIN-CommSy-URL: ..... https://www.mincommsy.uni-hamburg.de/
CommSy-Projektraum ..... SE1 CommSy WiSe 12/13
Ausgabewoche ..... 10. Januar 2013

Kernbegriffe

Das Konzept einer Liste (also einer unbeschränkten Sammlung mit einer manipulierbaren Reihenfolge, die auch Duplikate zulässt) lässt sich auf vielfältige Weise implementieren. Im Java Collections Framework gibt es für das Interface `List` zwei klassische imperative Implementationen: `ArrayList` und `LinkedList`.

`ArrayList` basiert auf dem Konzept der „wachsenden" Arrays: Für eine neue Liste wird ein Array der Größe *k* angelegt, so dass *k* Elemente direkt gespeichert werden können. Sobald die Liste mehr als *k* Elemente aufnehmen muss, wird ein neues Array der Größe *2*k* angelegt, und alle Elemente aus dem alten Array werden in das neue Array kopiert. Es wird deshalb zwischen der *Kapazität* und der *Kardinalität* einer `ArrayList` unterschieden: Die Kapazität ist die aktuelle Größe des Arrays, die Kardinalität hingegen ist die momentane Anzahl an Elementen in der Liste. Es gilt immer: *Kapazität* >= *Kardinalität*. Die Kardinalität ist dabei für Klienten über die Methode `size()` zugreifbar. Die Kapazität ist dagegen (genau wie das zugrundeliegende Array) ein Implementationsdetail.

`LinkedList` basiert auf der Verkettung von Kettengliedern/Knoten über Referenzen. Jeder Knoten hält eine Referenz auf das eigentlich zu speichernde Element sowie eine Referenz auf den nächsten Knoten und eine Referenz auf den vorherigen Knoten (*doppelt verkettete Liste*). Gäbe es keine Referenz auf den vorigen Knoten, wäre die verkettete Liste nur *einfach verkettet*. Bei verketteten Listen werden häufig so genannte *Wächterknoten* verwendet, um Sonderfälle (z. B. beim Einfügen oder Entfernen am Listenanfang und -ende) zu vermeiden.

Lernziele

Den Umgang mit Arrays und Referenzen vertiefen, die Implementationsformen Array-basierte Liste und verkettete Liste umsetzen können, Vor- und Nachteile der jeweiligen Implementation abschätzen können.

Aufgabe 11.1 Titel-Listen mit „wachsenden" Arrays implementieren

In Programmen wie iTunes ist es möglich, die Musiktitel in einer Musikdatenbank in Form von Wiedergabelisten zu organisieren. Solche Listen (wir nennen sie kurz Titel-Listen) enthalten eine beliebige Anzahl von Titeln, in einer vom Benutzer festgelegten Reihenfolge. Es sind auch echte Listen in dem Sinne, dass derselbe Titel mehrfach vorkommen kann. Wir wollen in SE 1 solche Titel-Listen auch erstellen können und haben deshalb das Interface `TitelListe` definiert. Auch wenn eine Titel-Liste normalerweise mit der Position 1 beginnt, soll unsere Liste (der Einfachheit halber) mit dem Index 0 beginnen.

11.1.1 Kopiert das Projekt `SE1Tunes` in euer Arbeitsverzeichnis und öffnet es. Das Projekt enthält ein Interface `TitelListe` und zwei unfertige Implementierungen dazu (`ArrayTitelListe` und `LinkedTitelListe`) sowie passende JUnit-Testklassen (`ArrayTitelListenTest` und `LinkedTitelListenTest`). Führt nacheinander die JUnit-Tests in `ArrayTitelListenTest` und `LinkedTitelListenTest` aus. Dazu könnt ihr jeweils in BlueJ mit rechts auf eine Testklasse klicken und den Menüpunkt *Alles testen* auswählen. **Haltet schriftlich fest**, was die Ausgaben für euch bedeuten.

11.1.2 **Erstellt eine Zeichnung** einer Array-Titelliste mit der Kapazität 10 und der Kardinalität 4.

11.1.3 Vervollständigt die Implementation der `ArrayTitelListe` in zwei Schritten:

Im ersten Schritt implementiert ihr alles so, dass das das Array nicht wächst. Nun laufen alle Tests durch bis auf `testeFuegeEinVergroessern`, `testeLoeschenAusGrosserListe` und `testeVergroessern`. **Tipp**: damit der Balken im ersten Schritt grün wird, könnt ihr die drei Tests auskommentieren und erst im zweiten Schritt wieder hinzunehmen.

Im zweiten Schritt wird die `fuegeEin`-Methode so ergänzt werden, dass das Array wächst. Entfernt die Kommentarzeichen bei den drei auskommentierten Tests. Am Ende sollten alle Tests durchlaufen.

Es ist hilfreich, während der schrittweisen Implementierung die Tests regelmäßig laufen zu lassen und die Methode `schreibeAufKonsole` an geeigneten Stellen von Hand aufzurufen, um den Zustand des Arrays und damit die Korrektheit eines Zwischenschritts zu überprüfen. Auch der Debugger ist hilfreich, um zu verstehen, was eine augenblickliche Implementation genau macht.

11.1.4 Implementiert einen weiteren Konstruktor, der die Angabe einer Anfangskapazität für die Listenimplementation erlaubt.

Aufgabe 11.2 Titel-Listen mit verketteten Listen implementieren

11.2.1 Die Klasse `LinkedTitelListe` soll das Interface `TitelListe` mit einer doppelt verketteten Liste implementieren. Intern enthält sie zwei Wächterknoten, einen für den Listenanfang, einen für das Ende. Die Wächterknoten markieren technisch die Grenzen der Liste und enthalten keine Titel. Die Wächterknoten garantieren, dass jeder echte Knoten einen Vorgänger und einen Nachfolger hat, was die Implementation enorm vereinfacht. Schaut euch die Klasse `DoppelLinkKnoten` und die unfertige Implementierung der Klasse `LinkedTitelListe` an. **Erstellt zwei Zeichnungen**, die zeigen, wie ein neuer Eintrag in diese Liste eingefügt wird. In den Zeichnungen soll deutlich werden, wie die Referenzen der Knoten verändert werden. Die Zeichnungen sollen sowohl die Knoten als auch die Titelobjekte zeigen.

11.2.2 Vervollständigt die Implementation der `LinkedTitelListe`. Auch in diesem Fall soll der Fortschritt und die Korrektheit durch regelmäßiges Testen überwacht werden.

11.2.3 *Zusatzaufgabe*: Die in der `LinkedTitelListe` verwendete Klasse `DoppellinkKnoten` ist global sichtbar, obwohl sie nur innerhalb der Listen-Implementation verwendet werden sollte. Java bietet mit dem Konzept der *verschachtelten Klassen* (engl. *nested classes*) die Möglichkeit, Klassendefinitionen in andere Klassendefinitionen einzubetten und so z.B. die Sichtbarkeit von Klassen auf den Kontext einer umgebenden Klasse zu beschränken. Bettet die Klasse `DoppellinkKnoten` in die `LinkedTitelListe` ein (als *static nested class*), so dass sie außerhalb dieser Klasse nicht mehr sichtbar ist.

11.2.4 *Zusatzaufgabe*: Erstellt eine alternative Implementation, die keine Wächterknoten verwendet. Vergleicht die Implementationen und erklärt eurem Betreuer die Unterschiede.

Aufgabe 11.3 Effizienz von Implementationen vergleichen

11.3.1 Schreibt eine Klasse, die das Laufzeitverhalten der beiden Listen-Implementationen misst. Es soll gemessen werden, wie sich die Listen beim *Einfügen* an der *ersten* Position der Liste und beim *Zugreifen* auf das *mittlere* Element der Liste verhalten. **Diskutiert schriftlich**, welche Liste in welcher Situation besser einzusetzen ist.

Zum Messen der Zeit könnt ihr die Methode `System.nanoTime()` verwenden. Beachtet dabei, dass sehr große Listen angelegt werden müssen, um einen messbaren Effekt zu erzielen. Versucht beispielsweise Listen mit 50.000 Einträgen.