Übung 3 – Verkettete Listen

Arbeiten Sie im Skript das *Kapitel Verkettete Listen* durch. Suchen Sie sich weitere Informationen, wo Sie die Ausführungen nicht verstehen. Sprechen Sie mit Ihren Kommilitoninnen und Kommilitonen. Fragen Sie im Forum. **Lesen Sie die Aufgaben vollständig und markieren Sie sich zentrale Aspekte. Verwenden Sie keine Klassen der Java API und beachten Sie konsistent die Zugriffrechte.** Verwenden Sie für die Bennenung Ihrer Testmethoden die im Skript definierten Konventionen (*Kapitel Unittesting*).

Aufgabe 1

Implementieren Sie die generische Klasse `EVL<T>` aus den Lernmaterialien und erweitern Sie sie um folgende Instanzmethoden:

- `getFirst()` und `getLast()`:
  Die Methoden geben das erste bzw. letzte Element der Liste zurück.
- `addLast(T e)`:
  Die Methode nimmt eine Referenzvariable `e` an und fügt ein neues Item mit `e` als Inhalt hinten an die einfach verkettete Liste an.
- `removeLast()`:
  Die Methode entfernt das letzte Listenelement und gibt dessen Inhalt zurück.
- `contains(T e)`:
  Die Methode prüft ob die Liste den übergebenen Referenzwert beinhaltet.
- `size()`:
  Die Methode gibt die Anzahl der Listenelemente innerhalb der Liste zurück.
- `toString()`:
  Die Methode gibt die Inhalte EVL getrennt von einem Bindestrich als String zurück (Beispiel: `1-42-13-27-0-11`). Implementieren Sie die Methode rekursiv.

Hinweise: Überlegen Sie, was Sie noch brauchen, um `getLast()` und `addLast()` geschickt zu implementieren. Was haben Sie bei `removeLast` bei einer einfach verketteten Liste für ein Problem? Wie können Sie `size()` geschickt implementieren, wie ungeschickt?

Implementieren Sie Exceptionhandling, wo es sinnvoll erscheint – Überlegen Sie welche Exceptions Sie aus vorigen Übungen kennen. Schreiben Sie JUnit-Tests für Ihre Implementierung und testen Sie Ihre neuen Methoden für mindestens zwei verschiedene Datentypen.

Aufgabe 2

Wir wollen in der folgenden Aufgabe eine `Schlange<T>` mit einer `EVL<T>` umsetzen. Da wir die Funtionalität der EVL zuvor in einer eigenen Klasse `EVL<T>` gekapselt haben, können wir diese hier nutzen. Gehen Sie dabei wie folgt vor:

- Implementieren Sie eine generische Klasse `SchlangeMitEVL<T>`, die das Interface `Schlange<T>` implementiert.
- Legen Sie intern eine Instanzvariable vom Typ `EVL<T>` an (anstatt eines Arrays wie bei `SchlangeMitArray<T>`).
- Stellen Sie einen parameterlosen Konstruktor zur Verfügung (oder nutzen Sie eine entsprechende Alternative), in dem Sie Ihre EVL-Instanzvariable instanziieren.
- Verwenden Sie für die Implementierung der Schnittstellen-Methoden die entsprechende Funktionalität (Methoden) Ihrer `EVL`-Instanz.
- Da eine EVL nicht wirklich eine Kapazität hat, geben Sie hier `Integer.MAX_VALUE` zurück (siehe Java API)
- Testen Sie Ihre Implementierung mit JUnittests.

Anmerkung: Hätten wir nicht die Klasse `EVL<T>` geschrieben, müssten wir alle Funktionalität einer EVL direkt in `SchlangeMitEVL` implementieren. Die Kapselung hat den Vorteil, dass wir die Klasse `EVL<T>` auch woanders verwenden können – bspw. wenn wir einen Stapel mit einer EVL umsetzen möchten.

Aufgabe 3

Wir wollen die Klassen `SchlangeMitArray` des vorherigen Aufgabenblattes und die Klasse `SchlangeMitEVL` hinsichtlich ihrer Effizienz betrachten. Überlegen Sie sich vorher, wo es Unterschiede geben könnte. *Was sind eventuell jeweils Vor- und Nachteile?*

Implementieren Sie eine Klasse `TimeTestSchlange` mit einer `main`-Methode. Instanziieren Sie in der `main` je ein Objekt vom Typ `SchlangeMitArray<Integer>` und vom Typ `SchlangeMitEVL<Integer>`. Deklarieren Sie außerdem drei `long` Variablen `start`, `finish`, `elapsed`. Fügen Sie in beide Objekte 100000 Integer-Werte hinzu und löschen Sie diese im Anschluss wieder (**Hinweis:** `for-Schleife`). Verwenden Sie folgendes Codebeispiel, um die Laufzeit jeweils für die beiden Objekte zu messen und geben Sie die Ergebnisse auf dem Bildschirm aus.

```
start = System.currentTimeMillis();
//Ihr Code fuer Einfuegen und Loeschen
finish = System.currentTimeMillis();
elapsed = finish - start;
```

Vergleichen Sie Ihre Ergebnisse zu den beiden unterschiedlichen Implementierungen (Array vs. EVL). *Was fällt Ihnen auf?*

Aufgabe 4

Ergänzen Sie die Klasse `EVL<T>` um eine Instanzmethode `zip`, die ein weiteres `EVL<T>`-Objekt `other` annimmt und in die gegebene `EVL`-Instanz `this` „einflechtet“ (wie ein Reißverschluss). Nachher soll `this` abwechselnd je ein bisheriges Element und eines von `other` enthalten, in der vorherigen Reihenfolge. Das erste Element soll aus `this` stammen, sofern `this` nicht leer gewesen ist.

Falls eine der beiden `EVL` kürzer ist als die andere, werden die übrigen Elemente der längeren `EVL` einfach angehängt. `other` soll nachher leer sein. Es sollen in der Methode keine neuen (Hilfs-)Objekte erzeugt werden, sondern lediglich die vorhandenen neu zusammengehängt werden. Es wird kein Wert zurückgegeben. Machen Sie sich eine Skizze wie viele Referenzen Sie zwischenspeichern müssen.

Beispiel: Enthält die `EVL<T>`-Instanz `evl` die Werte `(4,1,7)` und `other` die Werte `(3,5,2,8,6)`, so enthält `evl` nach dem Aufruf von `evl.zip(other)` die Werte `(4,3,1,5,7,2,8,6)`.

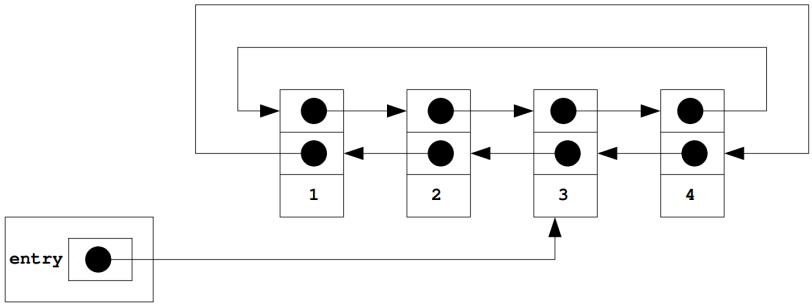
Hinweis: Machen Sie sich bevor Sie die Methode programmieren Gedanken über den Algorithmus, bspw. mit einer Skizze. Testen Sie die Implementierung Ihrer `zip`-Methode mithilfe von JUnittests.

Aufgabe 5

Schreiben Sie eine Klasse `RDVL<T>`, die eine *geschlossene* doppelt verkettete Liste darstellt. In einer solchen Liste verweisen die Referenzen `prev` und `next` jedes Listenelements auf Nachbarlistenelemente, sind also nicht `null`. Auch das „erste“ und „letzte“ Listenelement verweisen aufeinander. Die Listenelemente bilden somit einen Ring. Das RDVL-Objekt verweist mittels einer Referenz `entry` (statt `first` und `last`) auf ein Listenelement des Rings. `entry` ist `null`, wenn die Liste keine Listenelemente enthält. Die Struktur ist in der folgenden Abbildung dargestellt.

Ihre Klasse soll die folgenden Methoden liefern:

- `isEmpty()`:
  Gibt einen Wahrheitswert zurück, ob die Liste leer ist.
- `size()`:
  Die Methode gibt einen ganzzahligen Wert zurück, der angibt, wieviele Listenelemente momentan enthalten sind.



- `add( T e)`:
  Es wird ein neues Listenelement, dass `e` beinhaltet **vor** `entry` eingefügt. `entry` wird im Anschluss **nicht** verschoben.
- `remove()`:
  Das aktuelle Listenelement wird gelöscht und der Inhalt daraus zurückgegeben. Danach steht `entry` auf dem nächsten Element.
- `element()`:
  Der Inhalt des aktuellen Listenelements (`entry`) wird zurückgegeben.
- `next(int s)`:
  Verschiebt das aktuelle Element (`entry`) um `s` Schritte nach vorne.
- `prev(int s)`:
  Verschiebt das aktuelle Element (`entry`) um `s` Schritte nach hinten.

Führen Sie an Stellen, wo Sie es sinnvoll finden Exceptionhandling ein. Testen Sie Ihre Implementierung mit JUnittests.

Aufgabe 6

Das *Josephus-Problem* ist wie folgt definiert:
Gegeben sind zwei ganze Zahlen $n, k > 0$. Es werden die Zahlen $1, ..., n$ im Kreis angeordnet. Dann wird, von der 1 an zählend, jede k-te Zahl entfernt, wonach der Kreis jeweils wieder geschlossen und von der folgenden Position an weitergezählt wird. Zu bestimmen ist die letzte Zahl, die übrig bleibt.

Schreiben Sie eine Klasse `Josephus` zur Lösung des Josephus-Problems für beliebige $n, k > 0$. Diese Klasse enthält die Methode `loese`. Diese Methode nimmt n und k als Argumente an und gibt die letzte verbleibende Zahl zurück.
Verwenden Sie die Klasse `RDVL<T>` aus der vorherigen Aufgabe zur Lösung dieses Problems.

Beispiel: Es sei $n = 5$ und $k = 2$. Die fünf Zahlen bilden einen Kreis:

```
   1
5     2
 4   3
```

Zuerst wird 2 gelöscht:

```
   1
5     3
   4
```

Dann 4:

```
   1
5     3
```

Geht man wieder $k = 2$ Positionen weiter, kommt man zur 1, die nun gelöscht wird:

```
5     3
```

Zwei Positionen weiter liegt nun die 5, die gelöscht wird:

```
   3
```

Damit bleibt die 3 übrig und ist die Lösung des Josephus-Problems für $n = 5$ und $k = 2$.

Zusatzaufgabe (Programmierübung Projekt Euler)

Die Summe der Quadrate der ersten 10 natürlichen Zahlen ist

$$1^2 + 2^2 + ... + 10^2 = 385$$

Das Quadrat der Summe der ersten 10 natürlichen Zahlen ist

$$(1 + 2 + ... + 10)^2 = 552 = 3025$$

Somit ist die Differenz aus der Summe der Quadrate der ersten zehn natürlichen Zahlen und dem Quadrat der Summe $3025 - 385 = 2640$.

Finden Sie die Differenz aus der Summe der Quadrate der ersten einhundert natürlichen Zahlen und dem Quadrat der Summe.

`https://projekteuler.de/problems/6`

a. Implementieren Sie die Aufgabe als statische Methode, die einen `int`-Wert `range` an nimmt (für das obige Beispiel 10) und die Differenz der beiden Summen als Ganzzahl zurück gibt.

b. Testen Sie Ihre Methode mit JUnit Tests.