PT2: Serie 5

Abgabetermin	**26.06.2018, 21:00 Uhr**
Übungstermin	21.06.2018

Abstrakter Datentyp: Double-Ended Queue

Schnittstelle

Gegeben sei folgende Schnittstelle, die den abstrakten Datentyp einer *Deque* (Double-Ended QUEue) repräsentiert:

```
package de.uni_potsdam.hpi;

public interface Deque {
    int capacity();
    int size();
    void clear();
    void addFirst(Object e) throws DequeFull;
    void addLast(Object e) throws DequeFull;
    Object removeFirst() throws DequeEmpty;
    Object removeLast() throws DequeEmpty;
}
```

Eine Klasse, die die Schnittstelle `Deque` implementiert, soll dabei folgende Eigenschaften besitzen:

- Der Konstruktor der Klasse erwartet die maximale Kapazität des Deque-Exemplars. Die Methode `capacity()` gibt diese Größe zurück.
- Mit der Methode `addLast(Object)` wird ein Element in die Warteschlange am Ende eingefügt. Falls die Kapazität bereits erreicht ist, löst `addLast` die Ausnahme `de.uni_potsdam.hpi.DequeFull` aus. Gleichermaßen fügt `addFirst` am Anfang ein.
- Die Methode `removeFirst()` liefert das erste Elemente der Warteschlange und entfernt es aus dieser. Sollte die Warteschlange leer sein, so löst `removeFirst` die Ausnahme `de.uni_potsdam.hpi.DequeEmpty` aus. Gleichermaßen löscht `removeLast` ein Element am Ende.
- Die Methode `size()` liefert die aktuelle Zahl von Elementen in der Warteschlange.
- Die Methode `clear()` löscht alle Elemente aus der Warteschlange.

Modultests

Schreiben Sie auf Basis von JUnit eine Sammlung von Testfällen `DequeTest.java`, die mindestens folgende Eigenschaften testet:

- Für eine leere Deque `q` gilt nach `q.addLast(o)`: `o == q.removeFirst()`.
- Fügt man in eine Deque mit 7 Elementen eines mit `addLast` ein, erhält man es mit `removeLast` sofort zurück.
- Fügt man in eine Deque mit 7 Elementen eines mit `addFirst` ein, erhält man es mit `removeFirst` sofort zurück.
- Fügt man in eine Deque 6 mal den Wert `null` ein und danach ein von `null` verschiedenes Objekt `o`, und führt danach wiederholt `removeFirst` aus, so erhält man 6 mal den Wert `null`, und danach `o`.
- Nach Aufruf von `clear` liefert `size` den Wert 0.
- `removeLast` liefert für eine leere Deque eine Ausnahme.
- `addFirst` liefert für eine volle Deque eine Ausnahme.

Implementierungen

Implementieren Sie den abstrakten Datentypen `Deque` mithilfe folgender zwei Strategien. Alle Methoden sollen die Komplexität O(1) besitzen (vorausgesetzt alle Elementaroperationen, einschließlich der Speicherallozierung, erfolgen in konstanter Zeit).

- `ArrayDeque`: Als Ringpuffer auf Basis eines Arrays. Für einen solchen Ringpuffer sind neben dem Array selbst u.A. zwei Integer-Variablen (etwa `first` und `last`) erforderlich, die z.B. den ersten und den letzten belegten Index anzeigen.
- `LinkedDeque`: Als doppelt verkettete Liste. Zur Erreichung konstanter Zeit für alle Operationen empfiehlt es sich, eine Referenz nicht nur auf den Anfang der Liste, sondern auch auf das Ende der Liste zu führen. Alternativ ist es auch erlaubt, erstes und letztes Element der List miteinander zu verbinden, ggf. unter Einsatz eines Wächterelements.

Beginnen Sie mit Spezialisierungen der oberen Testsammlung als `ArrayDequeTest` und `LinkedDequeTest` im selben Paket.

Performanztests

Vergleichen Sie die Laufzeit beider Implementierungen für folgendes Anwendungsszenario:

In eine Warteschlange der Kapazität 1000 werden 500 Elemente eingefügt, danach wird oft abwechselnd ein Element entfernt und der Leerstring (`""`) eingefügt. Vergleichen Sie insbesondere die Zeit für dieses Einfügen und Entfernen.

Speichern Sie Ihr Testprogramm außerhalb des Pakets als `src/DequePerformance.java`, sowie Ihre Messergebnisse und die kurze Diskussion als `performance_tests.txt` im Wurzelverzeichnis.

Abgabe

Reichen Sie Ihre Lösung in Form eines einzelen gzip-komprimierten Tarfiles ein, welches sämtliche Klassen im Quelltext sowie dieses ant-File enthält.

Das ant-File übersetzt bereits Ihre Klassen und liefert einen TestRunner. Sorgen Sie noch dafür, dass auch `ant all` alle Quelltextdateien übersetzt (s. Dokumentation).

Ihre Lösung wird in einer Folgeaufgabe zum **Code-Review** einigen Kommilitonen zur Verfügung gestellt. Achten Sie darauf, dass aus Ihrem Code **Ihre Autorenschaft möglichst nicht hervorgeht**.

Es sollen keine Binärdateien, Validationsskripte, Editorbackups, lokale Repositories, o.Ä. enthalten sein.

```
/
|--build.xml
|--performance_tests.txt
|--src
    |--DequePerformance.java
    |--de
       |--uni_potsdam
          |--hpi
             |--ArrayDeque.java
             |--ArrayDequeTest.java
             |--Deque.java
             |--DequeEmpty.java
             |--DequeFull.java
             |--DequeTest.java
             |--LinkedDeque.java
             |--LinkedDequeTest.java
             |--[...]
```

Hinweise zur Ausführung

Java sucht die zur Übersetzung und Ausführung benötigten Klassen u.A. im sogenannten `CLASSPATH`. Für diese Aufgabe wird empfohlen die gleichname Umgebungsvariable so zu setzen, dass sie die absoluten Pfade (getrennt mit `:`) zu den JUnit-JARs und deren Abhängigkeiten enthält.

Beispiel für JUnit4 und Bash-kompatible Shells auf Unix:

```
export CLASSPATH=/path/to/jars/junit-4.12.jar:/path/to/jars/hamcrest-core-1.3.jar
ant all
ant test
```

Zusatzaufgabe

Ermitteln Sie mithilfe eines Analysewerkzeugs (z.B. JaCoCo, Covertura, Clover, Quilt oder NoUnit), wie vollständig Ihre Testsuite ist, und vervollständigen Sie sie zu 100% Code-Abdeckung (bezüglich des von Ihrem Werkzeug realisierten Messverfahrens).

Beschreiben und belegen Sie Ihr Vorgehen in `coverage.pdf` im Wurzelverzeichnis.