Übungsblatt 6

Musterlösung

Aufgabe 1 Ein Ring, sie zu testen...

Aufgabe 1.1 Testfälle definieren (60 %)

Die Klasse *RingBuffer* enthält einen Konstruktor und vier Methoden. Der Konstruktor bekommt als Parameter die Kapazität des Ringpuffers übergeben. Also müssen die Tests mit verschiedenen Kapazitäten durchgeführt werden. Wir haben uns hier für die Kapazitäten 0, 1, 3 und 1000 entschieden. 0 ist ein Grenzfall, da es die kleinste mögliche Kapazität darstellt und zudem die einzige Kapazität ist, bei der sich nichts in den Ringpuffer einfügen lässt. Die Kapazität 1 stellt auch einen Grenzfall dar, denn sie ist die kleinste, bei der sich tatsächlich etwas im Puffer speichern lässt. 3 und 1000 stellen Normalfälle dar, wobei 1000 eigentlich nur der Versuch ist zu testen, ob der Ringpuffer auch mit einer großen Anzahl von Elementen funktioniert: Der Test kann allerdings auch keine absolute Sicherheit bringen, denn wenn der Ringpuffer intern einfach mit einer festen, aber deutlich größeren Array-Größe als 1000 arbeiten würde, würden wir dies mit einem Test mit einer Kapazität von 1000 auch nicht entdecken. Da wir den Ringpuffer aber selbst implementieren werden, wissen wir, dass die Größe des Puffers tatsächlich von der Kapazität abhängig sein wird.

Negative Kapazitäten machen keinen Sinn. Wir wollen sie nicht in die regelmäßigen Tests aufnehmen. Nur der Vollständigkeit halber: die Implementierung bricht bei einer negativen Kapazitätsangabe mit einer *NegativeArraySizeException* ab, denn Java lehnt die Erzeugung von Arrays mit negativer Größe ab.

Die vier Methoden der Klasse *RingBuffer* sind *push*, *pop*, *peek* und *size*. Mit einer Ausnahme kann man sie nur in Kombination testen. Ein Trivialfall ist der Aufruf von *size* unmittelbar nach der Konstruktion des Puffers. Hier sollte immer 0 zurückkommen, da der Puffer zu Beginn ja leer ist. Ansonsten kann man eigentlich nur Folgen von *push* und *pop* testen und jeweils überprüfen, ob *size*, *peek* und *pop* selbst die erwarteten Ergebnisse liefern. Dabei sollte die Anzahl der Aufrufe von *push* auf jeden Fall die Kapazität des Puffers übersteigen und der Puffer zwischenzeitlich auch überfüllt sein, so dass Werte verloren gehen. Der Fall, dass *pop* oder *peek* bei einer *size* von 0 aufgerufen werden, kann nicht automatisiert getestet werden, da nicht bekannt ist, wie das Ergebnis in diesem Fall eigentlich aussehen soll.

Für die Kapazität 0 kann man nur *size* und *push* testen, da *pop* und *peek* kein gültiges Ergebnis liefern können[1]. Dabei sollte *size* immer 0 liefern, auch wenn *push* aufgerufen wurde, da der Puffer ja keine Kapazität besitzt, um einen Wert zu speichern. In der Klasse *RingBufferTest* sieht das so aus:

```
15      public void testCapacity0()
16      {
17          RingBuffer ringBuff1 = new RingBuffer(0);
18          assertEquals(0, ringBuff1.size());
19          ringBuff1.push(10);
20          assertEquals(0, ringBuff1.size());
21      }
```

[1] In der Implementierung erzeugen beide eine „ArithmeticException: / by zero“.

Die Methode *size* wurde zusätzlich für die Kapazitäten 1 und 3 getestet. Für die Kapazität 1 wird folgende Folge ausgeführt und nach jedem Schritt überprüft, ob die Größe die erwartete ist: *push(10), pop()* (Puffer wieder leer), *push(20), push(30)* (20 geht verloren, d.h. die Größe bleibt 1), *pop()* (Puffer wieder leer), *push(40), push(50), push(60)* (40 und 50 gehen verloren, d.h. die Größe bleibt 1).

```
24      public void testSize1()
25      {
26          RingBuffer ringBuff1 = new RingBuffer(1);
27          assertEquals(0, ringBuff1.size());
28          ringBuff1.push(10);
29          assertEquals(1, ringBuff1.size());
30          ringBuff1.pop();
31          assertEquals(0, ringBuff1.size());
32          ringBuff1.push(20);
33          ringBuff1.push(30);
34          assertEquals(1, ringBuff1.size());
35          ringBuff1.pop();
36          assertEquals(0, ringBuff1.size());
37          ringBuff1.push(40);
38          ringBuff1.push(50);
39          ringBuff1.push(60);
40          assertEquals(1, ringBuff1.size());
41      }
```

Für die Kapazität 3 wird folgende Folge ausgeführt: *push(10), push(20), push(30)* (Puffer ist voll), *pop(), push(40), push(50)* (20 geht verloren, d.h. die Größe bleibt 3), *pop(), pop(), pop()* (Puffer ist leer).

```
44      public void testSize3()
45      {
46          RingBuffer ringBuff1 = new RingBuffer(3);
47          assertEquals(0, ringBuff1.size());
48          ringBuff1.push(10);
49          assertEquals(1, ringBuff1.size());
50          ringBuff1.push(20);
51          assertEquals(2, ringBuff1.size());
52          ringBuff1.push(30);
53          assertEquals(3, ringBuff1.size());
54          ringBuff1.pop();
55          assertEquals(2, ringBuff1.size());
56          ringBuff1.push(40);
57          ringBuff1.push(50);
58          assertEquals(3, ringBuff1.size());
59          ringBuff1.pop();
60          assertEquals(2, ringBuff1.size());
61          ringBuff1.pop();
62          assertEquals(1, ringBuff1.size());
63          ringBuff1.pop();
64          assertEquals(0, ringBuff1.size());
65      }
```

Die Methode *pop* wurde für die Kapazitäten 1 und 3 getestet, wobei die gleichen Folgen wie bei den Tests von *size* verwendet werden, nur dass nun die Rückgaben von *pop* überprüft werden. Den Fall, dass mehr aus dem Puffer entnommen wird, als darin enthalten ist, haben wir nicht getestet, da nicht klar ist, welche Rückgabe wir eigentlich erwarten würden:

```
68      public void testPop1()
69      {
70          RingBuffer ringBuff1 = new RingBuffer(1);
71          ringBuff1.push(10);
72          assertEquals(10, ringBuff1.pop());
73          ringBuff1.push(20);
74          ringBuff1.push(30);
75          assertEquals(30, ringBuff1.pop());
76          ringBuff1.push(40);
77          ringBuff1.push(50);
78          ringBuff1.push(60);
79          assertEquals(60, ringBuff1.pop());
80      }
```

```
83      public void testPop3()
84      {
85          RingBuffer ringBuff1 = new RingBuffer(3);
86          ringBuff1.push(10);
87          assertEquals(10, ringBuff1.pop());
88          ringBuff1.push(20);
89          ringBuff1.push(30);
90          assertEquals(20, ringBuff1.pop());
91          assertEquals(30, ringBuff1.pop());
92          ringBuff1.push(40);
93          ringBuff1.push(50);
94          ringBuff1.push(60);
95          assertEquals(40, ringBuff1.pop());
96          assertEquals(50, ringBuff1.pop());
97          ringBuff1.push(70);
98          ringBuff1.push(80);
99          ringBuff1.push(90);
100         assertEquals(70, ringBuff1.pop());
101         assertEquals(80, ringBuff1.pop());
102         assertEquals(90, ringBuff1.pop());
103     }
```

Dieselben Tests werden für *peek* durchgeführt, da *peek* ja immer genau dasselbe wie ein nachfolgendes *pop* zurückliefern sollte:

```
106     public void testPeek1()
107     {
108         RingBuffer ringBuff1 = new RingBuffer(1);
109         ringBuff1.push(10);
110         assertEquals(10, ringBuff1.peek());
111         ringBuff1.pop();
112         ringBuff1.push(20);
113         ringBuff1.push(30);
114         assertEquals(30, ringBuff1.peek());
115         ringBuff1.pop();
116         ringBuff1.push(40);
117         ringBuff1.push(50);
118         ringBuff1.push(60);
119         assertEquals(60, ringBuff1.peek());
120     }

123     public void testPeek3()
124     {
125         RingBuffer ringBuff1 = new RingBuffer(3);
126         ringBuff1.push(10);
127         assertEquals(10, ringBuff1.peek());
128         ringBuff1.pop();
129         ringBuff1.push(20);
130         ringBuff1.push(30);
131         assertEquals(20, ringBuff1.peek());
132         ringBuff1.pop();
133         assertEquals(30, ringBuff1.peek());
134         ringBuff1.pop();
135         ringBuff1.push(40);
136         ringBuff1.push(50);
137         ringBuff1.push(60);
138         assertEquals(40, ringBuff1.peek());
139         ringBuff1.pop();
140         assertEquals(50, ringBuff1.peek());
141         ringBuff1.pop();
142         ringBuff1.push(70);
143         ringBuff1.push(80);
144         ringBuff1.push(90);
145         assertEquals(70, ringBuff1.peek());
146         ringBuff1.pop();
147         assertEquals(80, ringBuff1.peek());
148         ringBuff1.pop();
149         assertEquals(90, ringBuff1.peek());
150     }
```

Für größere Kapazitäten wurde noch ein programmierter Test hinzugefügt. Es wird immer die Folge *push(), pop(), push()* wiederholt und zwar dreimal so oft, wie der Puffer groß ist (hier 1000

Elemente). Für die ersten 1000 Durchläufe wird erwartet, dass die Größe kontinuierlich zunimmt und alle hinzugefügten Elemente sich auch wieder entnehmen lassen. *pushValue* wird in jedem Durchlauf um 2 erhöht, *popValue* nur um 1. Danach verharrt die Größe bei der Kapazität und jeder zweite Wert geht verloren, d.h. auch *popValue* wird in Zweierschritten erhöht:

```
153     public void testAll1000()
154     {
155         final int n = 1000;
156         final RingBuffer ringBuff1 = new RingBuffer(n);
157         int pushValue = 0;
158         int popValue = 0;
159 
160         // Puffer füllen n * (push, pop, push), noch geht nichts verloren
161         while (pushValue < n * 2) {
162             assertEquals(pushValue / 2, ringBuff1.size());
163             ringBuff1.push(pushValue++);
164             assertEquals(popValue, ringBuff1.peek());
165             assertEquals(popValue++, ringBuff1.pop());
166             ringBuff1.push(pushValue++);
167         }
168 
169         // Puffer voll, jeder zweite Wert geht verloren
170         while (pushValue < n * 6) {
171             assertEquals(n, ringBuff1.size());
172             ringBuff1.push(pushValue++);
173             assertEquals(++popValue, ringBuff1.peek());
174             assertEquals(popValue++, ringBuff1.pop());
175             ringBuff1.push(pushValue++);
176         }
177     }
```

Aufgabe 1.2 Teststärke prüfen (20 %)

Im Konstruktor kann die Zuweisung an *buffer* wegen des *final* zwar nicht weggelassen werden, aber eine Zuweisung von *null* hat praktisch denselben Effekt. Außerdem wurde die Größe des Arrays geändert. Mangels vorhandener Operatoren wurde hier testweise ein + 1 ergänzt.

In *push* wurde die erste Bedingung sowohl auf *false* als auch auf *true* gesetzt. Da eine Variation des Vergleichsoperators an dieser Stelle eigentlich gleichbedeutend damit ist, denn *buffer.length* kann ja nicht kleiner 0 sein, wurde auf weitere Änderungen an dieser Stelle verzichtet. Für die zweite Bedingung wurde zusätzlich auch der Vergleichsoperator variiert. Im Kontext des Programms ist <= dabei identisch mit *true*. Interessant ist >=, denn hier laufen alle Tests weiterhin erfolgreich durch. Dies ist aber kein Fehler, da >= der Bedingung lediglich Möglichkeiten hinzufügt, die in der Implementierung nicht vorkommen, d.h. *entries* > *buffer.length* wird niemals wahr. Das Entfernen des Aufrufs von *pop* hat denselben Effekt wie das Setzen der vorherigen Bedingung auf *false*, weshalb darauf verzichtet wurde. Beim Eintragen des Werts in den Puffer wurde einmal das Modulo entfernt und einmal das Weiterzählen der Elementanzahl. Im Prinzip hätte man hier statt *value* auch noch eine Konstante zuweisen können. Dies ist aber praktisch gleichbedeutend mit der Rückgabe des immer selben Array-Elements durch *peek*, was stattdessen probiert wurde.

Auf *pop* wirken sich auch Änderungen an *peek* aus, weshalb die erste und die letzte Zeile nicht geändert wurden. Beim Weiterzählen der Position wurden einmal das + 1 und einmal das Modulo entfernt. Ebenso wurde auch das Herunterzählen der Eintragsanzahl entfernt.

Bei *size* wurde einfach eine Konstante zurückgegeben, da sich die *return*-Anweisung ja nicht entfernen lässt.

Die Ergebnisse dieser Änderungen sind in der folgenden Tabelle zu sehen (die Namen der Tests sind gekürzt). Bis auf den oben diskutierten Fall werden alle Änderungen von Tests erkannt:

Zeile	Änderung	Cap0	Size1	Size3	Pop1	Pop3	Peek1	Peek3	All1000
30	buffer = null	×	×	×	×	×	×	×	×
30	new int[capacity + 1]	×	×	×	×	×	×	×	×
39	if (false)	✓	×	×	×	×	×	×	×
39	if (true)	×	✓	✓	✓	✓	✓	✓	✓
40	if (false)	✓	×	×	✓	×	✓	×	×
40	if (true)	✓	×	×	✓	×	✓	×	×
40	<= buffer.length	✓	×	×	✓	×	✓	×	×
40	>= buffer.length	✓	✓	✓	✓	✓	✓	✓	✓
43	% buffer.length wegl.	✓	✓	×	✓	×	✓	×	×
43	++ weglassen	✓	×	×	✓	×	✓	×	×
53	return buffer[0];	✓	✓	✓	✓	×	✓	×	×
63	+ 1 weglassen	✓	✓	✓	✓	×	✓	×	×
63	% buffer.length wegl.	✓	×	×	×	×	×	×	×
64	--entries wegl.	✓	×	×	✓	×	✓	×	×
75	return 0	✓	×	×	✓	✓	✓	✓	✓

Aufgabe 2 Spürhund (20 %)

Der Ringpuffer wurde in die Klasse *Walker* aus der Musterlösung für Übungsblatt 3 integriert. Es wurde ein neues Attribut hinzugefügt, das zwei Aufgaben hat. Zum einen ist es *null*, solange sich ein Objekt noch nicht im Verfolgermodus befindet. Zum anderen speichert es die Schritte der Spielfigur, sobald diese verfolgt wird.

```
18      private RingBuffer stepsToFollow = null;
```

Am Anfang der Methode *act* wird geprüft, ob der Verfolgermodus bereits aktiv ist. Wenn ja, wird die aktuelle Rotation der Spielfigur aufgezeichnet, d.h. in den Ringpuffer eingefügt.

```
30          if (stepsToFollow != null) {
31              stepsToFollow.push(player.getRotation());
32          }
```

Dann bewegt sich der NPC, wie bisher, vorwärts, d.h. in Richtung seiner aktuellen Rotation, und spielt den Schritt-Sound ab. Danach wird wieder geprüft, ob der Verfolgermodus aktiv ist. Wenn ja, wird eine Rotation aus dem Puffer entfernt und als aktuelle Drehung gesetzt.

```
52          if (stepsToFollow != null) {
53              avatar.setRotation(stepsToFollow.pop());
54          }
```

Ansonsten wird zuerst das bisherige Verhalten ausgeführt, also das Erhöhen des Schrittzählers und die Wende, wenn dieser das Maximum erreicht hat. Danach wird dann geprüft, ob der Verfolgermodus nun aktiviert werden muss. Dazu wird für die aktuelle Rotation des NPC geprüft, ob sich die Spielfigur in der Richtung auf einer Höhe mit diesem befindet und ob sich die Spielfigur auch davor (und nicht dahinter) befindet. Da es vier Richtungen gibt, gibt es diesen Test viermal, wovon maximal nur einer wahr sein kann. Ist der Test erfolgreich, wird der Abstand der Spielfigur vom NPC bestimmt. Ist dieser Abstand klein genug (höchstens vier Schritte), wechselt der NPC in den Verfolgermodus. Dazu wird ein Ringpuffer mit genau so vielen Einträgen wie der Abstand erzeugt und mit der aktuellen Rotation des NPC bis auf einen Eintrag gefüllt. Da sich die Spielfigur vor dem NPC befindet, kann deren Position genau durch diese Schritte erreicht werden, wobei der erste Schritt nicht im Puffer gespeichert werden muss, weil der NPC immer mit einem Schritt vorwärts beginnt. Ein Platz im Puffer muss auch frei bleiben, weil *act* immer erst eine neue Rotation einträgt, bevor wieder eine entnommen wird.

```
55          else {
```

```
56              // Weiterzählen
57              stepsSoFar = stepsSoFar + 1;
58
59              // Wenn maximale Anzahl erreicht, umdrehen und Zählung neu beginnen
60              if (stepsSoFar == maxSteps) {
61                  avatar.setRotation(avatar.getRotation() + 2);
62                  stepsSoFar = 0;
63              }
64
65              // Ist Spielfigur vor dieser Figur?
66              if (avatar.getRotation() == 0
67                          && player.getX() > avatar.getX()
68                          && player.getY() == avatar.getY()
69                      || avatar.getRotation() == 1
70                          && player.getX() == avatar.getX()
71                          && player.getY() > avatar.getY()
72                      || avatar.getRotation() == 2
73                          && player.getX() < avatar.getX()
74                          && player.getY() == avatar.getY()
75                      || avatar.getRotation() == 3
76                          && player.getX() == avatar.getX()
77                          && player.getY() < avatar.getY()) {
78                  // Entfernung zur Spielfigur bestimmen
79                  final int distance = max(abs(player.getX() - avatar.getX()),
80                          abs(player.getY() - avatar.getY()));
81
82                  // Wenn auch in Sichtweite, dann Puffer anlegen und Weg eintragen
83                  if (distance <= 4) {
84                      stepsToFollow = new RingBuffer(distance);
85                      for (int i = 1; i < distance; ++i) {
86                          stepsToFollow.push(avatar.getRotation());
87                      }
88                  }
89              }
90          }
```