9. Übungsblatt „Programmiersprachen“ WS 97/98

Leif Kornstaedt, Prof. Gert Smolka

Ausgabe: 15. Januar 1998 **Abgabe:** 26. Januar 1998

IX.1: Nebenläufigkeit (10 + 6 Punkte)

a. Die abstrakte Klasse `Repeat` dient dazu, beliebige Aktionen periodisch auszuführen. Sie hat folgende Schnittstelle:

```
abstract class Repeat {
  ...
  public synchronized void setDelay(long delay) { ... }
  public synchronized void start() { ... }
  public synchronized void stop() { ... }
  abstract public void run();
}
```

Instanzen von `Repeat` können sich in zwei Modi befinden, nämlich „off“ oder „on“. Nur im Modus „on“ wird die Methode `run`, die von abgeleiteten Klassen überschrieben werden muß, periodisch aufgerufen; über `setDelay` wird die Dauer der Pause in Millisekunden eingestellt, die zwischen zwei Aufrufen von `run` eingelegt wird. Beim Setzen einer Dauer von 0 oder weniger Millisekunden wird in den Modus „off“ übergegangen, ebenso beim Aufruf von `stop`. Durch Aufruf von `start` wird in den Modus „on“ übergegangen, vorausgesetzt, die eingestellte Dauer ist positiv. Initial beträgt die Dauer 0 und der Modus ist „off“.

Implementieren Sie alle Methoden von `Repeat`, die nicht abstrakt sind, sowie einen geeigneten Konstruktor.

b. Schreiben Sie eine Klasse `RepeatTest` mit einer Methode `public static void main (String[] args)`. Über die Kommandozeile übergibt man dieser paarweise Zeichenketten und ganze Zahlen; pro Paar werde periodisch mit der angegebenen Pausendauer die Zeichenkette auf die Standardausgabe geschrieben. Ein Beispiel für einen solchen Aufruf ist:

```
java RepeatTest a 100 b 200 c 400
```

Eine mögliche Ausgabe hiervon könnte folgendermaßen beginnen:

```
abcabaacbaabaacbaabaacbaabaacbabaacbaabaacbaabaacbaabaacbaa...
```

Das Programm soll nach zehn Sekunden abbrechen.

IX.2: Datenstruktur „Partition" (10 + 7 Punkte)

Eine *Partition* ist eine endliche Menge, deren Elemente in Äquivalenzklassen gruppiert sind, genannt *Blöcke*. In dieser Aufgabe soll eine Datenstruktur in Java implementiert werden, mit der sich Partitionen effizient modellieren lassen (siehe hierzu auch: Kurt Mehlhorn und Stefan Näher, *The LEDA Platform of Combinatorial and Geometric Computing*, Abschnitt 5.5, verfügbar unter `http://www.mpi-sb.mpg.de/~mehlhorn/LEDAbook.html`).

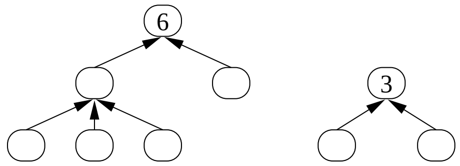
a. Elemente einer Partition modellieren wir durch die Klasse `PartitionItem` mit folgenden Methoden:

```
public class PartitionItem {
  ...
  public int getSize() { ... }
  public PartitionItem find() { ... }
  public void unifyWith(PartitionItem p) { ... }
  public boolean isInSameBlockAs(PartitionItem p) { ... }
}
```

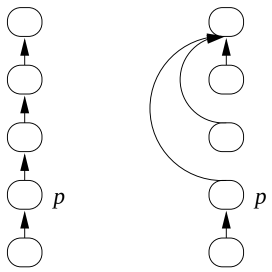
Die Instanziierung eines `PartitionItem` liefert ein Element, das sich alleine in einer neuen Äquivalenzklasse befindet. Die Methode `getSize` liefert die Anzahl der Elemente, die die zu dem Element gehörige Äquivalenzklasse enthält.

In jeder Äquivalenzklasse ist ein Element als eindeutiger Repräsentant ausgezeichnet. Die Methode `find` liefert diesen zurück. Durch Aufruf von `unifyWith` werden die Äquivalenzklassen des aufgerufenen und des übergebenen Elements vereinigt (dabei kann sich allerdings der Repräsentant einer der Äquivalenzklassen ändern). `isInSameBlockAs` liefert `true` genau dann, wenn zwei Elemente in derselben Äquivalenzklasse sind.

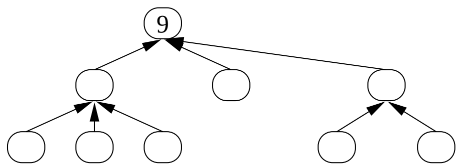
Zu implementieren sind Äquivalenzklassen als Bäume: Jedes Element enthält einen Zeiger auf seinen Vorgängerknoten; die Wurzel des so gebildeten Baumes ist der Repräsentant (dieser enthalte `null` als Elternzeiger). Weiterhin speichert jede Wurzel die Anzahl der Elemente, die sich in ihrer Äquivalenzklasse befinden. Folgende Graphik stellt zwei Äquivalenzklassen dar:



`find` ist nun einfach zu implementieren: Es muß nur die Verkettung der Elternzeiger durchlaufen werden, bis ein Nullzeiger angetroffen wird. Um nachfolgende Zugriffe effizienter zu machen, soll hierbei die sogenannte *path compression*-Technik angewendet werden: Jeder auf dem Weg angetroffene Elternzeiger soll nach dem Aufruf von `find` direkt auf die Wurzel zeigen, wie dies in dem nächsten Beispiel der Fall ist (nachdem ein `find` auf den mit p markierten Knoten angewendet wurde):



Um die Vereinigung zweier Äquivalenzklassen zu bilden, werden zunächst die Repräsentanten der beiden Elemente bestimmt. Sind diese verschieden voneinander, so wird der eine Repräsentant zu der Wurzel des anderen gemacht. Dabei wird der kleinere Baum (im Sinne von `getSize`) zu dem Kind des größeren gemacht, um die Bäume insgesamt balanciert zu halten. Die Vereinigung der beiden ganz oben abgebildeten Äquivalenzklassen sähe demzufolge so aus:



b. Schreiben Sie eine Methode `public static void main(String[] args)`, die als einziges Argument einen Integer erwartet. Entsprechend viele verschiedene Instanzen von `PartitionItem` sollen erzeugt werden. In einer Schleife werden nun je zwei Elemente zufällig ausgesucht (siehe hierzu die Klasse `java.util.Random`) und die zugehörigen Äquivalenzklassen vereinigt, und zwar solange, bis die größte Äquivalenzklasse mindestens die Hälfte aller Elemente enthält. Geben Sie dann die Größen der beiden größten Äquivalenzklassen aus sowie die Anzahl der benötigten Iterationen.

   Hinweis: Wenn Sie in einem Array für jede mögliche Äquivalenzklassengröße die Anzahl der Blöcke dieser Größe speichern, läßt sich diese Aufgabe recht leicht implementieren.