Dr. Annette Bieniusa
Mathias Weber, M. Sc.
Peter Zeller, M. Sc.

TU Kaiserslautern
Fachbereich Informatik
AG Softwaretechnik

Übungsblatt 7: Software-Entwicklung 1 (WS 2017/18)

Ausgabe: 04.12.17
Abgabe: 11.12.17

Probeklausur

Die Probeklausur ist freiwillig und soll eine Hilfestellung für Sie sein. Einerseits können Sie den Prozess schon einmal durchlaufen bevor es wirklich zählt. Andererseits können Sie Ihren aktuellen Stand unter Klausurbedingungen testen und erhalten mit der Korrektur Feedback.

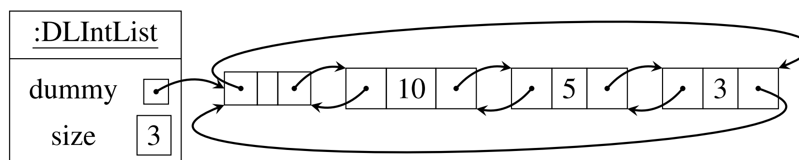
Die Probeklausur wird am 09.01.2018 um 17:30 Uhr in der Mensa stattfinden. Die Teilnahme ist freiwillig, jedoch müssen Sie über das STAT System angemeldet sein! Die **Anmeldung** ist ab sofort geöffnet und wird voraussichtlich am **21. Dezember** nach der Vorlesung geschlossen.

Es werden Ihnen 1/3 der in der Probeklausur erreichten Punkte als Übungspunkte gutgeschrieben.

Aufgabe 1 Doppelt verkettete Listen (17 Punkte)

Verwenden Sie für diese Aufgabe keine Elemente aus der Java Standard-Bibliothek.

In Kapitel 11 der Vorlesung (Abschnitt 4.1) werden doppelt verkettete Listen mit Dummy-Element kurz vorgestellt. Zum Beispiel lässt sich die Liste $[10, 5, 3]$ wie folgt repräsentieren:



Dabei repräsentieren die Boxen mit 3 Feldern jeweils einen `DNode` (siehe unten) in der Liste. Das linke Feld ist die Referenz zum Vorgänger-Knoten (`prev`). Das mittlere Feld ist der gespeicherte `int`-Wert (`value`). Das rechte Feld ist die Referenz zum nächsten Knoten (`next`).

```
class DNode {
  private DNode prev;
  private int value;
  private DNode next;

  // ...
}
```

Laden Sie sich für diese Aufgabe die Dateien `DLNode.java` und `DLIntList.java` herunter. Diese enthalten bereits eine Implementierung einer doppelt verketteten Liste mit Dummy-Element, welche Sie in dieser Aufgabe um weitere Methoden erweitern sollen. Sie sollen dabei **keine zusätzlichen Attribute** hinzufügen und die bestehenden Attribute nicht verändern.

Schreiben Sie JUnit Testfälle, um die von Ihnen geschriebenen Methoden zu testen

Hinweis: Sie können Ihren Code mit dem Java-Visualizer ausführen, um ihn besser zu verstehen und Fehler zu finden. Den Visualizer finden Sie online zum Beispiel[1] unter:
http://cscircles.cemc.uwaterloo.ca/java_visualize/

a) Schreiben Sie eine Methode `void add(int index, int element)`, welche das gegebene Element an Position `index` in die Liste einfügt. Dabei bezeichnet der Index `0` die erste Position in der Liste.

b) Schreiben Sie eine Methode `int[] toIntArray()`, welche ein Array mit den Werten der Liste zurückgibt.

c) Schreiben Sie eine Methode `int indexOf(int element)`, welche die Position des ersten Vorkommens von `element` in der Liste zurückgibt. Wenn das Element nicht in der Liste vorkommt, soll `-1` zurückgegeben werden.

d) Schreiben Sie eine Methode `int lastIndexOf(int element)`, welche die Position des letzten Vorkommens von `element` in der Liste zurückgibt. Wenn das Element nicht in der Liste vorkommt, soll `-1` zurückgegeben werden. Durchlaufen Sie für diese Aufgabe die Liste von hinten nach vorne.

e) Schreiben Sie eine Methode `void remove(int element)`, welche das erste Vorkommen des gegebenen Elements aus der Liste entfernt.

f) Schreiben Sie eine Iterator-Klasse `DLIntListRevIterator` und eine Methode `DLIntListRevIterator reverseIterator()` in der Klasse `DLIntList` um einen Iterator für eine Liste zu bekommen. Der Iterator soll die Methoden `hasNext` und `next` enthalten, welche analog zum `IntListIterator` aus der Vorlesung funktionieren sollen, aber die Liste beginnend beim letzten Element rückwärts durchläuft.

g) Schreiben Sie eine Prozedur `static int sum(DLIntList list)` in einer Klasse `Sum`, welche den Iterator verwendet, um die Summe aller Einträge in der Liste zu berechnen.

Aufgabe 2 Carambolage-Spiel, Teil 1 (7 Punkte)

Ziel dieser Aufgabe ist es ein kleines Spiel zu implementieren und dabei Objekte und Klassen verwenden, um den Zustand des Spiels zu repräsentieren. Carambolage ist eine Billard-Variante, die mit drei Kugeln gespielt wird. Der sogenannte Spielball muss dabei so gestoßen werden, dass er die beiden anderen Bälle, die sogenannten Objektbälle, berührt. Wir wollen hier eine Variante für einen Spieler mit beliebig vielen Spielzügen implementieren. Dabei sollen die Anzahl der Versuche und die Anzahl der erfolgreichen Versuche mitgezählt und angezeigt werden.
Mehr Details zum Original-Spiel finden Sie unter https://de.wikipedia.org/wiki/Carambolage.

Wir wollen die Position und Geschwindigkeit der Bälle jeweils durch Vektoren darstellen. In dieser Aufgabe sollen Sie eine Klasse `Vec2` schreiben, welche einen Vektor mit zwei Komponenten repräsentiert und nützliche mathematische Methoden bereitstellt, die Sie für den zweiten Teil der Aufgabe verwenden können.

Testen Sie Ihren Code mit den JUnit Testfällen aus `Vec2Tests.java`.

a) Erstellen Sie die Klasse `Vec2` mit einer x- und y-Koordinate vom Typ `double`. Markieren Sie diese Attribute mit `final` als unveränderlich. Erstellen Sie einen passenden Konstruktor `Vec2(double x, double y)`, welcher die beiden Werte initialisiert.

b) Schreiben Sie eine Methode `double length()`, welche die Länge des Vektors berechnet.

[1] Eine Alternative finden Sie unter http://www.pythontutor.com/visualize.html. Dort müssen Sie noch die Sprache von Python auf Java umstellen.

Die Länge eines Vektors $\begin{pmatrix} x \\ y \end{pmatrix}$ ist definiert als $\left\| \begin{pmatrix} x \\ y \end{pmatrix} \right\| = \sqrt{x^2 + y^2}$.

c) Schreiben Sie eine Methode `Vec2 mult(double factor)`, welche einen neuen Vektor zurückgibt, der um den gegeben Faktor skaliert ist.

Es gilt: $\begin{pmatrix} x \\ y \end{pmatrix} \cdot k = \begin{pmatrix} k \cdot x \\ k \cdot y \end{pmatrix}$.

d) Schreiben Sie eine Methode `Vec2 plus(Vec2 v)` und eine Methode `Vec2 minus(Vec2 v)`, mit denen zwei Vektoren addiert bzw. voneinander subtrahiert werden können.

Es gilt: $\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_1 + x_2 \\ y_1 + y_2 \end{pmatrix}$ und $\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} - \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_1 - x_2 \\ y_1 - y_2 \end{pmatrix}$.

e) Schreiben Sie eine Methode `double skalarProdukt(Vec2 v)`, welche das Skalarprodukt mit einem anderen Vektor ausrechnet.

Das Skalarprodukt zweier Vektoren $\begin{pmatrix} x_1 \\ y_1 \end{pmatrix}$ und $\begin{pmatrix} x_2 \\ y_2 \end{pmatrix}$ ist definiert als $\left\langle \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}, \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} \right\rangle = x_1 \cdot x_2 + y_1 \cdot y_2$.

f) Schreiben Sie eine Methode `double distanceTo(Vec2 v)`, welche den Abstand zu einem anderen Vektor ausrechnet. Der Abstand zwischen zwei Vektoren v_1 und v_2 ist gleich $\|v_1 - v_2\|$.

g) Schreiben Sie eine Methode `Vec2 normalized(double newLength)`, welche einen neuen Vektor mit der gleichen Richtung aber Länge `newLength` zurückgibt. Wenn die Länge des aktuellen Vektors 0 ist, soll der unveränderte Vektor zurückgegeben werden.

Aufgabe 3 Carambolage-Spiel, Teil 2 (13 Punkte)

Im zweiten Teil der Aufgabe geht es darum, das eigentliche Carambolage-Spiel zu implementieren. Die Aufgaben sind hier mit Absicht weniger präzise gestellt. Bitte treffen Sie hier Entwurfsentscheidungen gegebenenfalls selbst, insbesondere welche Klasse welche Funktionalität implementieren soll! Beachten Sie dabei, was für das Spiel hier sinnvoll ist. **Wichtig:** Am Ende sollte **nicht** Ihr gesamter Code in einer einzelnen Klasse stehen.

Laden Sie sich für diese Aufgabe die Vorlage `carambolage` herunter. Die Vorlage enthält die Klasse `GUI`, welche das graphische Benutzeroberfläche verwaltet. Diese Klasse müssen Sie für diese Aufgabe weder anpassen noch verstehen. Die Klasse `SEGraphics` bietet Methoden zum Zeichnen von Kreisen, Linien und Text, sowie eine `reset`-Methode um das aktuelle Bild mit einer gegeben Farbe zu überschreiben. Diese Klasse müssen Sie nicht anpassen, aber Sie müssen die Methoden dieser Klasse verwenden.

a) Implementieren Sie eine Klasse `Ball`. Ein Ball hat eine aktuelle Position und eine aktuelle Geschwindigkeit (können jeweils durch einen Vektor dargestellt werden). Außerdem hat ein Ball eine Farbe und einen Radius.

b) Erweitern Sie den Konstruktor der Klasse `Spielfeld` so, dass die drei Bälle erstellt werden, die für das Spiel benötigt werden (siehe Screenshot auf der vorherigen Seite).

c) Erweitern Sie die Methode `zeichnen` in der Klasse `Spielfeld` so, dass sie das aktuelle Spielfeld mit den Bällen zeichnet. Diese Methode erhält als Parameter ein `SEGraphics`-Objekt. Verwenden Sie zum Zeichnen die Methoden dieses Objekts (siehe Datei `SEGraphics.java`). Diese Methode wird mehrmals pro Sekunde von der Klasse `Gui` aus aufgerufen, so dass Sie sich nicht um den Aufruf dieser Methode kümmern müssen.

d) Erweitern Sie die Methode `simulationsSchritt` in der Klasse `Spielfeld` so, dass die Bälle auf dem Spielfeld sich gemäß ihrer aktuellen Geschwindigkeit bewegen. Die Methode `simulationsSchritt` wird ebenfalls mehrmals pro Sekunde von der Klasse `Gui` aus aufgerufen. Der Parameter `deltaT` gibt dabei die vergangene Zeit an und sollte zur Berechnung der Bewegung verwendet werden.

   Für einen Ball mit Geschwindigkeit-Vektor v und Positions-Vektor p, ist die neue Position $p' = p + v \cdot deltaT$.

   Durch Reibung sollte sich in jedem Simulationsschritt auch die Geschwindigkeit verringern. Sie können zum Beispiel die folgende Formel verwenden, um die neue Geschwindigkeit v' zu berechnen:

   $v' = v - \left(v \cdot \frac{20 \cdot deltaT}{||v||}\right)$

e) Passen Sie die Methode `mouseClicked` so an, dass sich damit der Spielball stoßen lässt. Die Methode wird von der Klasse `GUI` immer dann aufgerufen, wenn der Spieler mit der Maus auf das Spielfeld klickt. Die Parameter `x` und `y` geben die Position des Klicks an.

   Um den Ball zu stoßen können Sie zum Beispiel die aktuelle Geschwindigkeit des Spielballs auf die Differenz zwischen der Position des Klicks und der aktueller Position des Spielballs setzen. Damit rollt der Ball dann in Richtung des Klicks und ist schneller, wenn man weiter weg vom Ball klickt. Sie können sich aber auch eine andere Steuerung überlegen, um den Spielball zu stoßen.

   Es sollte nur dann möglich sein, den Ball zu stoßen, wenn sich kein Ball mehr bewegt.

f) Erweitern Sie die Simulation so, dass Bälle nicht mehr aus dem Spielfeld rollen können, sondern stattdessen vom Rand des Spielfeldes abprallen. Ein Ball kollidiert zum Beispiel mit dem rechten Rand, wenn seine x-Position plus sein Radius größer als die Breite des Spielfeldes ist und er sich nach rechts bewegt. Um eine Kollision mit dem rechten Rand zu behandeln, muss man die Geschwindigkeit in x-Richtung umkehren. Die Geschwindigkeit in y-Richtung bleibt in dem Fall unverändert. Die Fälle für die anderen 3 Ränder funktionieren analog.

g) Erweitern Sie das Spiel um eine Anzeige, welche die folgenden Informationen enthält:

   1. Die Anzahl der Versuche (wie oft der Ball gestoßen wurde)
   2. Die Anzahl der erfolgreichen Versuche, also der Stöße nach denen der Spielball beide Objektbälle berührt hat.

   Verwenden Sie die `drawString`-Methode von `SEGraphics` für die Anzeige.

h) Erweitern Sie die Simulation, so dass Bälle auch voneinander abprallen können.

   Dazu kann der Wikipedia-Artikel zur elastischen Kollision hilfreich sein[2]. Dort findet man unter “Two-Dimensional Collision With Two Moving Objects” die folgenden Formeln:

$$
\begin{aligned}
\mathbf{v}'_1 &= \mathbf{v}_1 - \frac{2 \cdot m_2}{m_1 + m_2} \cdot \frac{\langle \mathbf{v}_1 - \mathbf{v}_2, \mathbf{x}_1 - \mathbf{x}_2 \rangle}{||\mathbf{x}_1 - \mathbf{x}_2||^2} \cdot (\mathbf{x}_1 - \mathbf{x}_2), \\
\mathbf{v}'_2 &= \mathbf{v}_2 - \frac{2 \cdot m_1}{m_1 + m_2} \cdot \frac{\langle \mathbf{v}_2 - \mathbf{v}_1, \mathbf{x}_2 - \mathbf{x}_1 \rangle}{||\mathbf{x}_2 - \mathbf{x}_1||^2} \cdot (\mathbf{x}_2 - \mathbf{x}_1)
\end{aligned}
$$

   Dabei ist x_1 die Position von Ball 1, v_1 die Geschwindigkeit von Ball 1 vor der Kollision und v'_1 die neue Geschwindigkeit danach. Die Variablen x_2, v_2 und v'_2 sind entsprechend für Ball 2. Die Variablen m_1 und m_2 stehen für die Masse der Bälle. Da die Bälle in unserem Spiel alle die gleiche Masse haben, kann dieser Teil ignoriert werden. Die Notation $\langle v, w \rangle$ steht wie oben für das Skalarprodukt und die Notation $||v||$ für die Länge eines Vektors.

   Hinweis: Durch Ungenauigkeiten in der Simulation kann es dazu kommen, dass Bälle miteinander kollidieren, während sie sich schon voneinander weg bewegen. In diesem Fall sollte die Geschwindigkeit der Bälle nicht verändert werden, da es sonst passieren kann, dass zwei Bälle sich ineinander verhaken.

[2] http://en.wikipedia.org/wiki/Elastic_collision

Freiwillige Zusatzaufgabe (0 Punkte)

Die folgende Aufgabe können Sie selbstständig bearbeiten und abgeben. Dazu finden Sie im Exclaim eine zusätzliche Übung "SE1WS17Z", wo Sie Ihre Lösung als Programm `BossBattle.java` hochladen können. Die Abgaben werden automatisch getestet, aber nicht bewertet.

Bei der Lösung der Aufgaben kommt es auch auf die Effizienz des Algorithmus an. Mit einer langsamen Lösung werden Sie wahrscheinlich für große Eingaben einen Timeout erhalten.

Diese Aufgabe stammt vom "Northwestern Europe Regional Contest (NWERC) 2017" und steht unter der Lizenz "Creative Commons Attribution-ShareAlike".

Boss Battle

You are stuck at a boss level of your favourite video game. The boss battle happens in a circular room with n indestructible pillars arranged evenly around the room. The boss hides behind an unknown pillar. Then the two of you proceed in turns.

- First, in your turn, you can throw a bomb past one of the pillars. The bomb will defeat the boss if it is behind that pillar, or either of the adjacent pillars.
- Next, if the boss was not defeated, it may either stay where it is, or use its turn to move to a pillar that is adjacent to its current position. With the smoke of the explosion you cannot see this movement.

The last time you tried to beat the boss you failed because you ran out of bombs. This time you want to gather enough bombs to make sure that whatever the boss does you will be able to beat it. What is the minimum number of bombs you need in order to defeat the boss in the worst case? See Figure 1 for an example.

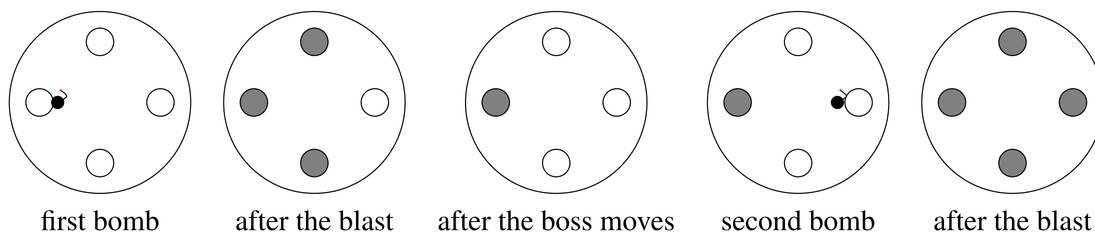


Abbildung 1: Example for $n = 4$. In this case 2 bombs are enough. Grey pillars represent pillars where the boss cannot be hiding. The bomb is represented in black.

Input

The input consists of:

- One line with a single integer n ($1 \leq n \leq 100$), the number of pillars in the room.

Output

Output the minimum number of bombs needed to defeat the boss in the worst case.

Sample Input 1	Sample Output 1
4	2

Sample Input 2	Sample Output 2
7	5