

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

Die Hinweise und Vorschläge in diesem Dokument sollen der Lösungsfindung dienen und erheben demnach weder Anspruch auf Vollständigkeit noch Korrektheit. Sollten Sie Fehler finden, würden wir uns freuen, wenn Sie uns diese mitteilen. (Kontaktinformationen finden Sie auf unserer Webpräsenz.)

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)

- 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.

```
// requires 0 <= index && index <= size
public void add(int index, int element) {
    addBefore(findNode(index), element);
}

private DNode findNode(int index) {
    DNode node = dummy;
    if (index <= size / 2) {
        // Index von Vorne suchen:
        for (int i = 0; i <= index; i++) {
            node = node.getNext();
        }
    } else {
        // Index von Hinten suchen:
        for (int i = size; i > index; i--) {
            node = node.getPrev();
        }
    }
    return node;
}

private void addBefore(DNode node, int element) {
    DNode n = new DNode(node.getPrev(), element, node);
```

```

        node.getPrev().setNext(n);
        node.setPrev(n);
        size = size + 1;
    }

```

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

```

public int[] toArray() {
    int[] result = new int[size()];
    DNode node = dummy;
    for (int i = 0; i < size; i++) {
        node = node.getNext();
        result[i] = node.getValue();
    }
    return result;
}

```

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.

```

public int indexOf(int element) {
    DNode node = dummy;
    for (int i = 0; i < size; i++) {
        node = node.getNext();
        if (element == node.getValue()) {
            return i;
        }
    }
    return -1;
}

```

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.

```

public int lastIndexOf(int element) {
    DNode node = dummy;
    for (int i = size - 1; i >= 0; i--) {
        node = node.getPrev();
        if (element == node.getValue()) {
            return i;
        }
    }
    return -1;
}

```

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

```

public void remove(int element) {
    DNode node = dummy;
    for (int i = 0; i < size; i++) {
        node = node.getNext();
        if (element == node.getValue()) {
            removeNode(node);
            break;
        }
    }
}

```

```

private void removeNode(DNode node) {
    node.getPrev().setNext(node.getNext());
    node.getNext().setPrev(node.getPrev());
    size = size - 1;
}

```

- 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.

```

// Iterator-Methode in Klasse DLIntList:
public DLIntListRevIterator reverseIterator() {
    return new DLIntListRevIterator(dummy);
}

class DLIntListRevIterator {
    private DNode dummy;
    private DNode node;

    public DLIntListRevIterator(DNode dummy) {
        this.dummy = dummy;
        node = dummy;
    }

    public boolean hasNext() {
        return node.getPrev() != dummy;
    }

    public int next() {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        node = node.getPrev();
        return node.getValue();
    }
}

```

- 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.

```

public class Sum {
    public static int sum(DLIntList list) {
        int sum = 0;
        DLIntListRevIterator iter = list.reverseIterator();
        while (iter.hasNext()) {
            sum = sum + iter.next();
        }
        return sum;
    }
}

```

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

```

class Vec2 {
    final double x;
    final double y;

    public Vec2(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double length() {
        return Math.sqrt(lengthSquared());
    }

    public double lengthSquared() {
        return x * x + y * y;
    }

    public Vec2 mult(double factor) {
        return new Vec2(x * factor, y * factor);
    }

    public Vec2 plus(Vec2 other) {
        return new Vec2(x + other.x, y + other.y);
    }

    public Vec2 minus(Vec2 other) {
        return new Vec2(x - other.x, y - other.y);
    }

    public double skalarProdukt(Vec2 other) {
        return x * other.x + y * other.y;
    }

    public double distanceTo(Vec2 position) {
        return this.minus(position).length();
    }

    public Vec2 normalized(double newLength) {
        double len = length();
        if (len > 0) {
            return new Vec2(x * newLength / len, y * newLength / len);
        } else {
            return this;
        }
    }
}

```

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

```

import java.awt.Color;

public class Ball {
    private Vec2 position;
    private Vec2 velocity;
    private final double radius;
    private final Color color;
    public boolean touched = false;

    public Ball(Vec2 position, Vec2 velocity, double radius, Color color) {
        this.position = position;
        this.velocity = velocity;
    }
}

```

```

    this.radius = radius;
    this.color = color;
}

public void bewegen(double deltaT) {
    position = position.plus(velocity.mult(deltaT));
    if (velocity.length() < 1) {
        velocity = new Vec2(0,0);
    } else {
        velocity = velocity.minus(velocity.normalized(30*deltaT));
    }
}

public void kollisionMitRand(double breite, double hoehe) {
    if (position.x < radius && velocity.x < 0) { // linker Rand
        // Umkehren der Geschwindigkeit in x-Richtung
        velocity = new Vec2(-velocity.x, velocity.y);
        // Ball wieder ins Spielfeld schieben
        position = new Vec2(radius, position.y);
    } else if (position.x > breite - radius && velocity.x > 0) { // rechter Rand
        velocity = new Vec2(-velocity.x, velocity.y);
        position = new Vec2(breite - radius, position.y);
    }

    if (position.y < radius && velocity.y < 0) { // oberer Rand
        velocity = new Vec2(velocity.x, -velocity.y);
        position = new Vec2(position.x, radius);
    } else if (position.y > hoehe - radius && velocity.y > 0) { // unterer Rand
        velocity = new Vec2(velocity.x, -velocity.y);
        position = new Vec2(position.x, hoehe - radius);
    }
}

public void zeichnen(SEGraphics seg) {
    seg.drawCircle(position.x, position.y, radius, color);
}

public boolean kollidiertMit(Ball ball2) { // Testet auf Kollision
    return position.distanceTo(ball2.position) <= radius + ball2.radius;
}

public static void kollision(Ball b1, Ball b2) { // simuliert Kollision
    // Geschwindigkeits-Unterschied zwischen Baellen (delta velocity)
    Vec2 dv = b2.velocity.minus(b1.velocity);
    // Positions-Unterschied zwischen Baellen (delta position)
    Vec2 dp = b2.position.minus(b1.position);

    // Berechnen des Skalarprodukts:
    double skalar = dv.skalarProdukt(dp);
    // Berechnen der quadrierten Distanz zwischen Baellen:
    double distSquared = dp.lengthSquared();

    double dist = Math.sqrt(distSquared);
    if (skalar < 0 && dist > 0) {
        // skalar1 < 0 um zu pruefen, ob Baelle aufeinander zu fliegen
        // dist > 0 um Division durch 0 zu verhindern

        b1.velocity = b1.velocity.plus(dp.mult(skalar / distSquared));

        b2.velocity = b2.velocity.minus(dp.mult(skalar / distSquared));
    }
}

boolean isStopped() {

```

```

        return velocity.lengthSquared() < 0.01;
    }

    public void pushTowards(Vec2 pos) {
        velocity = pos.minus(position);
    }
}

import java.awt.Color;

public class Spielfeld {
    private int breite;
    private int hoehe;
    private Ball[] baelle;
    private Ball spielball;
    private int wins = 0;
    private int tries = 0;
    private boolean roundWon = false;

    public Spielfeld(int breite, int hoehe) {
        this.breite = breite;
        this.hoehe = hoehe;
        initBaelle();
    }

    public int getBreite() {
        return breite;
    }

    public int getHoehe() {
        return hoehe;
    }

    private void initBaelle() {
        baelle = new Ball[3];
        spielball = new Ball(new Vec2(breite * 2 / 3, hoehe / 2), new Vec2(0, 0),
20, Color.WHITE);
        baelle[0] = spielball;
        baelle[1] = new Ball(new Vec2(breite * 1 / 3, hoehe * 1 / 3), new Vec2(0, 0)
, 20, new Color(200, 50, 50));
        baelle[2] = new Ball(new Vec2(breite * 1 / 3, hoehe * 2 / 3), new Vec2(0, 0)
, 20, new Color(200, 50, 50));
    }

    public void simulationsSchritt(double deltaT) {
        for (Ball ball : baelle) {
            ball.bewegen(deltaT);
            ball.kollisionMitRand(breite, hoehe);
        }
        for (int i = 0; i < baelle.length - 1; i++) {
            Ball ball1 = baelle[i];
            for (int j = i + 1; j < baelle.length; j++) {
                Ball ball2 = baelle[j];
                if (ball1.kollidiertMit(ball2)) {
                    kollision(ball1, ball2);
                }
            }
        }
    }

    private void kollision(Ball ball1, Ball ball2) {
        Ball.kollision(ball1, ball2);
        if (ball1 == spielball) {
            ball2.touched = true;
        }
    }
}

```

```

    }
    if (!roundWon && allBallsTouched()) {
        roundWon = true;
        wins++;
    }
}

private boolean allBallsTouched() {
    for (Ball ball : baelle) {
        if (ball != spielball && !ball.touched) {
            return false;
        }
    }
    return true;
}

public void zeichnen(SEGraphics seg) {
    // Hintergrund:
    seg.reset(new Color(20, 100, 80));

    // Spielstand:
    seg.drawString("Versuche: " + tries, 20, 20, Color.WHITE);
    seg.drawString("Erfolge: " + wins, 20, 50, Color.WHITE);

    // Baelle:
    for (Ball ball : baelle) {
        ball.zeichnen(seg);
    }
}

private boolean allStopped() {
    for (Ball ball : baelle) {
        if (!ball.isStopped()) {
            return false;
        }
    }
    return true;
}

public void mouseClicked(int x, int y) {
    if (allStopped()) {
        nextTry();
        spielball.pushTowards(new Vec2(x, y));
    }
}

private void nextTry() {
    tries++;
    roundWon = false;
    for (Ball ball : baelle) {
        ball.touched = false;
    }
}
}
}

```