Technische Universität Braunschweig
Institut für Anwendungssicherheit

Prof. Dr. Martin Johns

Programmieren I

05. Übungsblatt

Aufgabe 25: Schreiben Sie ein Programm, das quadratische Gleichungen der Form

$$ax^2 + bx + c = 0$$

mit $a \neq 0$ löst. Zuerst soll Ihr Programm die Koeffizienten a, b und c einlesen und anschließend die Lösungen gemäß der Formel

$$x_{1,2} = -\frac{p}{2} \pm \sqrt{\left(\frac{p}{2}\right)^2 - q}$$

mit $p = \frac{b}{a}$ und $q = \frac{c}{a}$ berechnen und ausgeben. Wenn keine reelle Lösung existiert, soll eine entsprechende Meldung erfolgen. Die Wurzel einer Zahl können Sie mit der Methode `Math.sqrt` berechnen.

Aufgabe 26: In der Informatik werden häufig Zufallszahlen benötigt, z. B. um Simulationen durchzuführen. Einer der ersten Algorithmen zur Erzeugung von *Pseudozufallszahlen* wurde von John von Neumann in der 1940er Jahren entwickelt.
Dieser Algorithmus wird *Mid-Square-Methode* genannt. Er beginnt mit einer Zahl, die als erste Zufallszahl genommen wird. Die Zahl wird quadriert. Anschließend werden die mittleren Ziffern der Quadratzahl als nächste Zahl verwendet. Falls die Quadratzahl nicht doppelt so lang ist, müssen vorne führende Nullen eingefügt werden. Als Beispiel betrachten wir die ersten Schritte des Algorithmus mit der Anfangszahl 4637:

```
4637 --> 21501769 = 4637*4637
5017 --> 25170289 = 5017*5017
1702 --> 02896804 = 1702*1702
8968 --> 80425024 = 8968*8968
4250 --> 18062500 = 4250*4250
 625 --> 00390625 =  625* 625
3906 --> ...
```

Meistens wiederholen sich nach relativ wenigen Schritten die Zufallszahlen. Aus diesem Grund wird der Algorithmus heute kaum noch verwendet.
Mit der Anfangszahl 4637 erzeugt die Mid-Square-Methode die folgenden Pseudozufallszahlen

$$4637, 5017, 1702, 8968, 4250, 625, 3906, \ldots, 6100, 2100, 4100, 8100, 6100, \ldots$$

Dann wiederholen sich die Zahlen 6100, 2100, 4100, 8100. Inklusive der Anfangszahl (4637) besteht die Folge aus 47 verschiedenen Zufallszahlen.
Schreiben Sie ein Java-Programm, das den Algorithmus auf alle vierstelligen Zahlen anwendet. Das heißt auf alle Anfangszahlen aus dem Bereich $1000, \ldots, 9999$. Ihr Programm soll ausgeben, wie viele verschiedene Zufallszahlen maximal von einer Anfangszahl erzeugt werden können. Außerdem soll Ihr Programm diese Folge ausgeben.
Ob Sie die Anfangszahl zu den Zufallszahlen mitzählen oder nicht, bleibt Ihnen überlassen.
Falls es mehrere Anfangszahlen geben sollte, die die maximale Anzahl der Zufallszahlen erzeugen, dürfen Sie eine auswählen.

Aufgabe 27: In Java gibt es sogenannte *checked Exceptions*, welche von `Exception` erben, sowie *unchecked Exceptions*, welch von `RuntimeException` erben.
Machen Sie sich mit den Unterschieden vertraut.
Ausführliche Dokumentation zu Exceptions in Java finden Sie beispielsweise unter: `https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html`.

Aufgabe 28: Das Programm in Abbildung 11 bricht während der Ausführung mit einem Fehler ab.
Erklären Sie die Fehlerursache und überlegen Sie sich, wie man diese beheben könnte.

Aufgabe 29: Diese Aufgabe basiert auf der Pflichtaufgabe dieses Übungsblatts. Sie sollen die `toString` Methode nun so anpassen, das diese den Baum mit jedem Knoten ausgibt. Diese sollen entsprechend ihrer Positionen im Baum eingerückt sein.
Ein Beispiel für eine solche Ausgabe ist in Abbildung 12 gegeben.

```
class DownTo {
  static void downToZero(int n) {
    if(n == 0) {
      return;
    }
    downToZero(n-1);
  }

  public static void main(String[] args) {
    int[] bounds = { 10, 100, 1000, 10000, 100000, 1000000 };
    for(int bound : bounds) {
      System.out.println("Testing for: " + bound);
      downToZero(bound);
    }
  }
}
```

Abbildung 11: Erklären Sie den zur Laufzeit auftretenden Fehler.

```
(
  (
    (
      2
    )
    4
    (
      6
    )
  )
  7
  (
    (
      (
        8
      )
      9
    )
    15
    (
      19
    )
  )
)
```

Abbildung 12: Eingerückte Ausgabe für den Baum aus Abbildung 13.

Pflichtaufgabe 30: In dieser Pflichtaufgabe sollen die Konzepte der Rekursion vertieft werden.
Ziel der Pflichtaufgabe ist es, einen binären Baum zu implementieren. Ein Beispiel für einen binären Baum ist in Abbildung 13 gegeben.

Definition: Ein binärer Baum ist ein azyklischer Graph in dem jeder Knoten maximal zwei Kinder hat.
Des Weiteren gilt, dass jeder Knoten k einen Wert v_k hat und alle Knoten tiefer und links im Baum von k kleinere Werte beinhalten und alle Knoten tiefer und rechts im Baum von k größere Werte beinhalten.
Der erste Knoten des Baum – der Knoten der keinen Elternknoten hat – wird als Wurzelknoten bezeichnet. Ein Knoten ohne Kind wird als *Blatt* bezeichnet.
Für eine formale Definition verweisen wir auf das Skript aus Algorithmen und Datenstrukturen[8]. Ansonsten hat auch Wikipedia eine kurze Einführung[9].

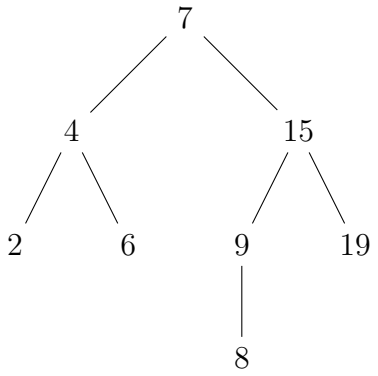

Abbildung 13: Ein binärer Baum. Der Knoten mit dem Wert 7 ist der Wurzelknoten des gesamten Baum. Die Knoten mit den Werten 2, 6, 8, 19 sind Blätter, da diese keine Kinderknoten haben.

[8] `https://www.ibr.cs.tu-bs.de/courses/ws1718/aud/skript/Skript-5.pdf`
[9] `https://de.wikipedia.org/wiki/Bin%C3%A4rbaum`

Geforderte Features:

- Erweitern Sie die in Abbildung 14 gegebene Klasse um eine Methode `insert`, die rekursiv neue Werte in den Baum einfügt. Diese sollen gemäß der oben genannten Definition des binären Baumes, an den korrekten Stellen als Knoten eingefügt werden. Überlegen Sie sich ein sinnvolles Verhalten, sollte bereits ein Knoten mit dem selben Wert existieren.

- Implementieren Sie eine zweite einfügen Methode, z.B. `insertIterative` genannt. Diese soll allerdings in iterativer Form programmiert werden.
  Welche Implementierung ist Ihnen leichter gefallen? Welche finden Sie eleganter? Bitte geben Sie Ihre Antworten im Javadoc Kommentar der zweiten einfügen Methode an.

- Implementieren Sie folgende Methoden:

  a) `heigth`: Zum Berechnen der Höhe des Baumes. Die Höhe ist definiert als die Anzahl der Kanten im längste Pfad vom Wurzelknoten zu einem der Blätter. Die Höhe des Baums aus Abbildung 13 ist beispielsweise 4 ($7 \rightarrow 15 \rightarrow 9 \rightarrow 8$)

  b) `exists`: Zum prüfen ob ein Wert im Baum vorhanden ist

  c) `min`: Auslesen des kleinsten Wertes im Baum

  d) `max`: Auslesen des größten Wertes im Baum

  Diese sollen alle rekursiv implementiert werden.

- Überschreiben Sie die `toString()` Methode, so dass diese den Baum formatiert ausgibt.

  Wie die formatierte Ausgabe aussieht, ist Ihnen freigestellt. Die Baumstruktur muss allerdings aus der Ausgabe erkennbar sein. Ein simples Beispiel für den Baum aus Abbildung 13 ist: `(((2) 4 (6)) 7 (((8) 9) 15 (19)))`.

- Einen binären Baum, in welchem jeder Knoten entweder gar keinen oder einen Kindknoten hat, nennt man einen *entarteten Baum*.

  Implementieren Sie eine Methode `isDegenerate` die prüft ob ein Baum *entartet* ist.

- In der Vorlesung *Rekursion und Funktionale Programmierung* haben sie das Java Feature *Funktionale Interfaces* kennengelernt.

  Schreiben Sie ein solches funktionales Interface, welches ein *Prädikat* (Eine Funktion die basierend auf Eingabewerten einen Wahrheitswert zurückgibt) darstellt.

  Die Baum Klasse soll nun um ein Methode erweitert werden, welche prüft ob für alle seine Werte ein übergebenes Prädikat wahr zurückgibt. Ein Beispiel für die Verwendung, hier ist die Methode `forAll` genannt, ist in Abbildung 15 gegeben.

  Diese Methode soll ebenfalls rekursiv implementiert werde.

- Implementieren Sie eine `main` Methode, welche die Funktionalität des Baumes und all seiner Methoden demonstriert.

Der zweite Teil dieser Pflichtaufgabe ist nun eine Klasse zur Repräsentation von Mengen über ganzen Zahlen zu implementieren. Diese soll zur Speicherung der Elemente die `Tree` Klasse verwenden.

- Implementieren Sie basierend auf dem Grundgerüst aus Abbildung 16 die `IntSet` Klasse mit Ihren Konstruktoren und den Methoden `insert` (ein neues Element hinzufügen), `contains` (prüfen ob ein Element bereits vorhanden ist), `union` (das Zusammenfügen von zwei Sets in einem neuen Set) und `intersection` (erstellen eines neuen Set bestehend aus den gemeinsamen Elementen von zwei Sets).

  Die `union` und `intersection` Methoden sollen die Parameter auf Gültigkeit prüfen. Wird etwa `null` übergeben, soll eine Exception geworfen werden.

  Hier können Sie wahlweise eine eigene Exception Klasse schreiben, oder aus der Java Standardbibliothek einen passenden Exception Typ verwenden. Begründen Sie in den Javadoc Kommentaren der Mtehoden ihre Entscheidung kurz.

- Implementieren Sie die `equals` und `toString` Methoden der `IntSet` Klasse.

  Hier müssen Sie unter Umständen die `Tree` Klasse entsprechend erweitern, um auf alle Werte des Baumes zugreifen zu können.

- Schreiben Sie eine main Methode, welche die Funktionalität der `IntSet` Klasse inklusive Fehlerbehandlung demonstriert.

Für die Umsetzung von jedem der obigen Teilaspekte erhalten Sie jeweils einen Punkt. Haben Sie mindestens einen der Teilaspekte korrekt bearbeitet, können Sie einen weiteren Punkt für Einhaltung der Programmierrichtlinien erhalten. Diese sind:

- Formatierung des Programms gemäß der Formatierungsrichtlinien.
- Javadoc Kommentare für jede Klasse, jedes Interface und jede public Methode.
- Keine überflüssigen Dateien (hier primär `.class` und `.jar` Dateien, aber auch Sonstiges, das nicht unter Quelltext oder Dokumentation fällt) im Git.

Für das Lösen dieser Pflichtaufgabe können Sie maximal 10 Punkte erhalten.
Berücksichtigt werden ausschließlich Ergebnisse, die bis zum 30.01.2020 um 23:59 in das Git Repository Ihrer Gruppe gepusht wurden.
Um Ihre Lösung bewertet zu bekommen, müssen Sie diese in der Woche vom 03.02.2020 - 07.02.2020 dem Tutor/der Tutorin in Ihrer Übungsgruppe vorstellen.
Nur compilierende Programme gelten als Lösung! Ein Programm das Teillösungen beinhaltet aber nicht compiliert oder aus anderen Gründen nicht ausführbar ist erhält 0 Punkte. Abgaben müssen in dem aufgabenspezifischen Order erfolgen: *Blatt*05. Code der nicht in diesem Order (oder Unterordner) liegt wird nicht bewertet.

```java
public class Tree {


    private int value;
    private Tree lhs; // left child
    private Tree rhs; // right child

    Tree(int value) {
        this.value = value;
    }

    ...
}
```

Abbildung 14: Attribute der Tree Klasse

```java
Tree t = new Tree(7);
int[] numbers = {4,6,5,2,1,3,15,19,11,9,13,8};

for(int number : numbers) {
  t.insert(number);
}
boolean allSmallerTwenty = t.forAll(v -> v < 20);
```

Abbildung 15: Prüfen ob alle Werte in einem binären Baum kleiner 20 sind.

```java
public class IntSet {

  private Tree items;

  public IntSet() { /* ... */ }
  public IntSet(int[] items) { /* ... */ }

  public void insert(int value) { /* ... */ }
  public boolean contains(int value) { /* ... */ }

  // Vereinigung
  public IntSet union(IntSet other) { /* ... */ }
  // Durchschnitt
  public IntSet intersection(IntSet other) { /* ... */ }

  public boolean equals(Object x) { /* ... */ }
  public String toString() { /* ... */ }
}
```

Abbildung 16: Grundgerüst für die *IntSet* Klasse.