Discord: timostucki

Meine Aufgaben Ratings:

- Best Case: Ihr macht alle Aufgaben (, wenn die Zeit reicht)
- Falls nicht: Ich mache Ratings zur Wichtigkeit der einzelnen Aufgaben

- (Keine offizielle Empfehlung, meine subjektive Meinung auf Basis meiner eigenen Erfahrung als Student in diesem Kurs)
- ! Sagen nichts über die Schwierigkeit der Aufgaben aus !

Wichtig für die Prüfung:
(Prüfungs- oder
prüfungsähnliche Aufgaben)

Wichtig für euer Verständnis:
(Aber nicht in Prüfungsform)

Wichtig für tiefes Verständnis/
Zusatzaufgaben:
(Bspw. viele vom gleichen Typ)

Aufgabe 1: Loop-Invarianten

1. Um die Loop-Invariante einfacher schreiben zu können, dürfen Sie `min(arr, i)` benutzen. Hier steht `min(arr, i)` für das minimale Element in dem Array `arr` von Index `0` bis und mit Index `i`. Alternativ könnte man auch formale Notation benutzen, in dem man mit Quantoren arbeitet. Zum Beispiel, falls `m` = `min(arr, i)`, dann könnten Sie äquivalent folgendes schreiben

$$\forall\, 0 \leq j \leq i\, (\texttt{arr}[j] \leq m)$$

```
int min(int[] arr) {
  // Precondition: arr != null 0 < arr.length
  int m = arr[0];
  int i = 1;

  // Loop-Invariante:
  while (i < arr.length) {
    if (arr[i] < m) {
      m = arr[i];
    }

    i++;
  }

  // Postcondition: m = min(arr, arr.length)
  return m;
}
```

Aufgabe 1: Loop-Invarianten

2.
```
String append(String str1, String str2) {
  // Precondition: str1 != null && str2 != null
  String s1 = str1;
  String s2 = str2;

  // Loop-Invariante:
  while (!s2.equals("")) {
    s1 = s1 + s2.charAt(0);
    s2 = s2.substring(1);
  }

  // Postcondition: s.equals(str1 + str2)
  return s1;
}
```

Achtung: Die Bedingung `str1 != null && str2 != null` ist wichtig, damit Aufrufe wie `s2.equals()`, `s2.charAt(0)` und `s2.substring(1)` überhaupt möglich sind. Der Aufruf `s2.substring(1)` produziert das gleiche Resultat wie `s2.substring(1, s2.length())`.

Aufgabe 2: Database

In dieser Aufgabe implementieren Sie für eine Datenbank von Personengesundheitsdaten das Deklassifizieren von Einträgen (Task a) und das Verlinken von Einträgen (Task b). Alle Unteraufgaben können separat gelöst werden.
Die Datenbank selber ist bereits mit der Klasse `Database` implementiert. Die Datenbank hält eine Liste von Einträgen, welche durch die Klasse `Item` repräsentiert werden. Die folgenden 4 Paragraphen erklären alle in der Vorlage gegebenen Klassen im Detail.

`Item` Die Klasse `Item` repräsentiert einen Datenbankeintrag mit 4 Attributen: eine ID (int), ein Alter (int), einen Gesundheitswert (int), und ein Sicherheitslevel, welches durch die Klasse `Level` repräsentiert wird. Alter und Gesundheitswert sind immer ≥ 0. Die Methoden `Item.getID()`, `Item.getAge()`, `Item.getHealth()`, `Item.getLevel()` geben jeweils die ID, das Alter, den Gesundheitswert, und das Sicherheitslevel eines Eintrags zurück. Die Methode `Item.setHealth(int newHealth)` setzt den Gesundheitswert auf `newHealth`. Die anderen Attribute können nicht geändert werden.

`Level` Die Klasse `Level` repräsentiert ein Sicherheitslevel. Ein Sicherheitslevel wird über eine Liste von Integern definiert, welches in einem Attribut der Klasse `Level` gespeichert wird und von der Methode `Level.getPoints()` zurückgegeben wird. Ein Level `A` ist *verwandt* mit einem Level `B`, falls die Summe der Werte in `A.getPoints()` gleich der Summe der Werte in `B.getPoints()` ist. Zum Beispiel ist das Level $[1,2,3,4]$ verwandt mit den Levels $[10]$ und $[4,6]$ (die Summe ist überall 10), aber nicht mit dem Level $[4,5]$.

Aufgabe 2: Database

`ItemFactory` Die Klasse `ItemFactory` wird verwendet, um Datenbankeinträge zu erstellen. Die Methode `ItemFactory.createItem(Level level, int id, int age, int health)` gibt ein Exemplar der Klasse `Item` zurück, deren Attribute mit den Argumenten initialisiert wurden.

`Database` Die Klasse `Database` repräsentiert eine Datenbank und hat folgende vorgegebene Methoden:

- `Database.getItemFactory()` gibt ein Exemplar von `ItemFactory` zurück. Die `ItemFactory` `I` ist assoziiert mit der Datenbank `D`, falls `I` von `D.getItemFactory()` zurückgegeben wird.
- `Database.add(Item item)` fügt der Datenbank den Eintrag `item` hinzu.
- `Database.getItems()` gibt die Liste aller Einträge zurück, welcher der Datenbank hinzugefügt wurden. Sie dürfen annehmen, dass für eine Datenbank `D` alle Einträge in `D.getItems()` eine einzigartige ID haben, über `D.add` hinzugefügt wurden, über `D.getItemFactory()` erstellt wurden, und keiner anderen Datenbank hinzugefügt werden. Ein hinzugefügter Eintrag wird nie wieder entfernt.

1. Implementieren Sie die Methode `ItemFactory.createDeclass(Level level, int id, int targetId)`, die einen *Deklassifikationseintrag* zurückgibt. Ein Deklassifikationseintrag ist selber ein Eintrag, also ein Exemplar der Klasse `Item`. Ein Deklassifikationseintrag hat damit auch eine ID, ein Sicherheitslevel, ein Alter, und einen Gesundheitswert, welche von den

Aufgabe 2: Database

entsprechenden getter-Methoden zurückgegeben werden. ID und Sicherheitslevel eines Deklassifikationseintrags sind jeweils das `id` und `level` Argument des `createDeclass` Aufrufs, mit welchem der Eintrag erstellt wurde. Das Alter und der Gesundheitswert eines Deklassifikationseintrags sind jeweils das Alter und der Gesundheitswert des *Zieleintrags* vom Deklassifikationseintrag. Der Zieleintrag von einem Deklassifikationseintrag `D` ist der Eintrag `E`, so dass

- `E.getID()` gleich dem Parameter `targetId` ist, mit welchem `D` erstellt wurde; *und*
- `E` aus der Datenbank ist, mit welcher die `ItemFactory` assoziiert ist, mit welcher `D` erstellt wurde.

Falls es keinen Zieleintrag gibt, wird eine `IllegalArgumentException` von der Methode `createDeclass` geworfen. Beachten Sie, dass Zieleinträge selber Deklassifikationseinträge sein können. Ein Aufruf der Methode `Item.setHealth(h)` auf einem Deklassifikationseintrag hat keinen Effekt; dies wird nicht in den Tests überprüft.

Ein Deklassifikationseintrag `R` *erreicht* einen Eintrag `A`, falls entweder `A` der Zieleintrag von `R` ist oder falls der Zieleintrag von `R` ein Deklassifikationseintrag ist, welcher `A` erreicht. Die Methode `createDeclass` wirft eine `IllegalArgumentException`, falls der zurückzugebene Deklassifikationseintrag `R` einen Eintrag erreicht, dessen Level verwandt ist mit dem Level von `R`. Zur Erinnerung: Der Paragraph über die Klasse `Level` erklärt, wann zwei Level verwandt sind.

Aufgabe 2: Database

2. Implementieren Sie die Methode `Database.createLink(List<Integer> ids)`. Der Methodenaufruf `D.createLink(ids)` *verlinkt* alle Einträge der Datenbank `D` miteinander, welche eine ID haben, die im Argument `ids` enthalten ist. Wenn `E.setHealth(h)` auf einem Eintrag `E` aufgerufen wird, dann wird der Gesundheitswert aller Einträge, welche mit `E` verlinkt sind, auf das Argument `h` gesetzt. Einträge können beliebig oft verlinkt werden und verlinken ist transitiv, das heisst, wenn ein Eintrag `A` mit einem Eintrag `B` verlinkt ist und `B` mit einem Eintrag `C` verlinkt ist, dann ist `A` auch mit `C` verlinkt. Verlinken ist auch immer symmetrisch, das heisst, wenn `A` mit `B` verlinkt ist, dann ist auch `B` mit `A` verlinkt. Zusätzlich ist verlinken reflexiv, das heisst, ein Eintrag ist immer mit sich selber verlinkt.

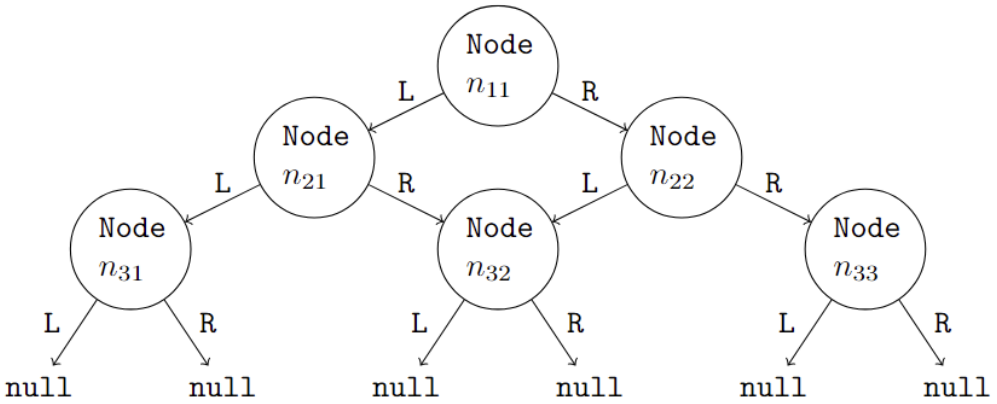
   Der Aufruf `D.createLink(ids)` soll eine `IllegalArgumentException` werfen, falls es eine ID im Argument `ids` gibt, für welche es keinen Eintrag mit der gleichen ID in der Datenbank `D` gibt.

Wir geben zwei Testdateien zur Verfügung. "DatabaseTest.java" enthält Tests, welche wir an einer Prüfung geben würden. "GradingDatabaseTest.java" enthält Tests, welche wir zum Korrigieren einer Prüfung verwenden würden. Testen Sie Ihre Lösung zuerst ausgiebig mit "DatabaseTest.java" (am besten fügen Sie selber neue Tests hinzu) und dann können Sie "GradingDatabaseTest.java" verwenden, um zu sehen wie Ihre Lösung an einer Prüfung abgeschnitten hätte.

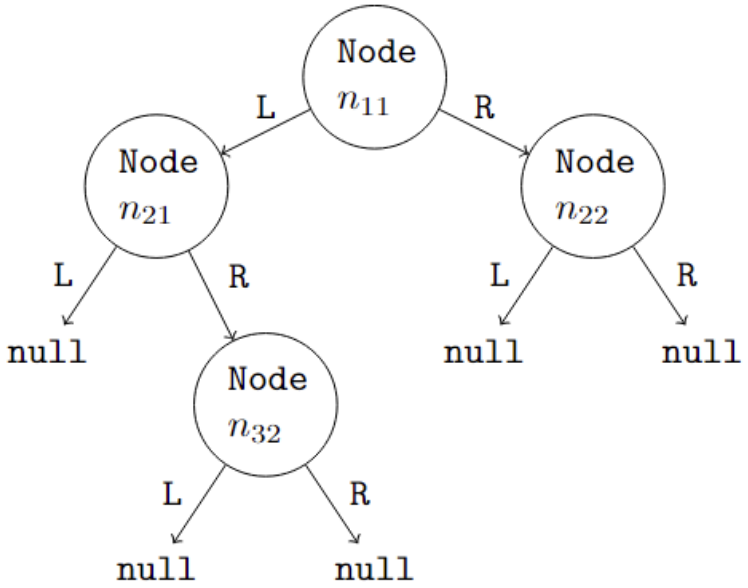
Aufgabe 3: Pyramide

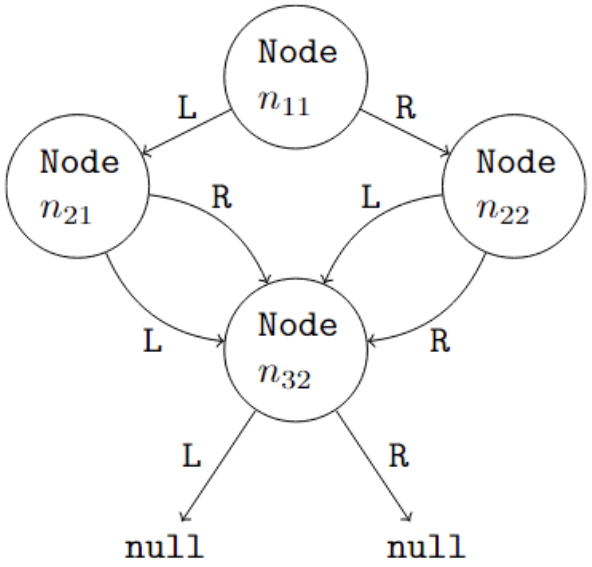
Die Klasse `Node` repräsentiert einen Knoten in einem gerichteten Graphen, wobei es für jeden Knoten n_1 höchstens zwei gerichtete Kanten von n_1 zu anderen Knoten n_2, n_3 geben kann (n_2 und n_3 können gleich sein). Wir unterscheiden dabei zwischen dem linken und dem rechten Knoten. Die Methode `Node.getLeft()` gibt den linken Knoten und `Node.getRight()` den rechten Knoten zurück (als `Node`-Objekt). Wenn der linke Knoten von n_1 nicht existiert, dann gibt `Node.getLeft()` `null` zurück (analog für den rechten Knoten).

Das Ziel dieser Aufgabe ist, für ein `Node`-Objekt zu entscheiden, ob der durch das `Node`-Objekt definierte Graph einer Pyramide entspricht. Zum Beispiel entspricht der folgende Graph einer Pyramide.



Aufgabe 3: Pyramide





Aufgabe 3: Pyramide

Beachten Sie, dass der rechte Knoten von n_{21} gleich ist wie der linke Knoten von n_{22} (das heisst die Node-Objekte sind gleich!). Ein Graph (wie oben repräsentiert) definiert eine Pyramide genau dann, wenn folgende Bedingungen gelten:

- Der Graph kann in $k \geq 1$ Stufen (Stufe 1, Stufe 2,..., Stufe k) aufgeteilt werden, wobei Stufe i aus i unterschiedlichen Knoten n_{i1}, n_{i2}, ..., n_{ii} besteht. Falls der Graph k Stufen hat, dann hat dieser genau $\frac{k(k+1)}{2}$ unterschiedliche Knoten (Knoten aus verschiedenen Stufen sind unterschiedlich).
- Für Stufe i $(1 \leq i < k)$ gilt: der linke Knoten von n_{ij} $(1 \leq j \leq i)$ ist durch $n_{(i+1)j}$ gegeben und der rechte Knoten von n_{ij} ist durch $n_{(i+1)(j+1)}$ gegeben.
- Für Stufe k gilt: es gibt keinen linken und keinen rechten Knoten für n_{kj} $(1 \leq j \leq k)$.

Die folgenden Graphen entsprechen zum Beispiel keinen Pyramiden:
Implementieren Sie die `boolean isPyramid(Node node)`-Methode, welche, für den Graph G durch `node` definiert, entscheidet, ob G eine Pyramide definiert. Sie dürfen annehmen, dass G keine Zyklen hat. Die Methode soll eine `IllegalArgumentException` werfen, wenn das Argument `null` ist.

Tipp: Prüfen Sie die Bedingungen Stufe für Stufe, beginnend bei Stufe 1.

Aufgabe 4: Rechnungen (erweitert)

In dieser Aufgabe erweitern Sie eine vorherige Aufgabe, in welcher ein System für Stromverbräuche Rechnungen erstellt. Konkret gibt es drei Erweiterungen: (1) Es sollen auch nicht korrekt formatierte Eingabedateien gehandhabt werden. (2) Ein Kunde kann eine beliebige Anzahl von Verbrauchswerten haben. (3) Es gibt eine neue Unteraufgabe b. In der folgenden Aufgabenbeschreibung für Unteraufgabe a sind die Änderung in **bold** markiert.

a) Vervollständigen Sie die `process`-Methode in der Klasse `Bills`. Die Methode hat zwei Argumente: einen `Scanner`, von dem Sie den Inhalt der Eingabedatei lesen sollen, und einen `PrintStream`, in welchen Sie die unten beschriebenen Informationen schreiben.

   Ihr Programm muss **auch mit manchen nicht korrekt formatierten Eingabedatein umgehen**. **Die Aufgabestellung gibt an, wie mit nicht korrekt formatierten Eingaben umzugehen ist**. Ein Beispiel einer korrekt formatierten Datei finden Sie im Projekt unter dem Namen "Data.txt". Exceptions im Zusammenhang mit Ein- und Ausgabe können Sie ignorieren.

   Eine valide Eingabedatei enthält Zeilen, die entweder den Tarif, der angewendet werden soll, oder die Daten für den Stromverbrauch eines Kunden beschreiben. Der Verbrauch eines Kunden ist niemals grösser als 100000 Kilowattstunden.

   Eine Tarifbeschreibung hat folgendes Format:

$$\texttt{Tarif}\,␣\,n\,␣\,l_1\,␣\,p_1\,\ldots\,l_n\,␣\,p_n$$