TU München, Fakultät für Informatik
Lehrstuhl III: Datenbanksysteme
Prof. Alfons Kemper, Ph.D.

Übung zur Vorlesung *Einführung in die Informatik 2 für Ingenieure (MSE)*

Christoph Anneser (anneser@in.tum.de)
http://db.in.tum.de/teaching/ss23/ei2/

Lösungen zu Blatt Nr. 7
Dieses Blatt wird am Montag, den 19.06.2023 besprochen.

Aufgabe 1: Rekursion mit Boxen

Gegeben sei eine Menge an Boxen. Jede Box hat eine Länge, eine Höhe und eine Breite. Sie sollen nun den höchst-möglichen Stapel an Boxen bauen bzw. herausfinden, wie hoch maximal gestapelt werden kann. Dabei muss jedoch beachtet werden, dass eine Box b_1 nur dann auf eine Box b_2 gestellt werden kann, wenn Länge, Höhe und Breite von b_1 kleiner sind als die von b_2.

Aufgabe: Vervollständigen Sie den untenstehenden Code an den mit `todo` gekennzeichneten Stellen.

Optional: Wir berechnen damit jedoch für manche Boxen die maximale Höhe mehrmals, wie können wir dies umgehen? Tipp: Verwenden Sie *Dynamische Programmierung*.

```
class Box implements Comparable<Box> {
    public Box(int id, int width, int heigth, int length) {
        this.id = id;
        this.width = width;
        this.height = heigth;
        this.length = length;
    }

    protected int id;
    protected int width;
    protected int height;
    protected int length;

    @Override
    public int compareTo(Box o) {
        if (this.width < o.width && this.length < o.length && this.height <
        ↪  o.height) return -1;
        return 1;
    }
}


public class Main {
    static int getMaxHeigth(Set<Box> remainingBoxes, Box lastBox) {
        if (remainingBoxes.isEmpty())
            return lastBox != null ? lastBox.height : 0;
                // create deep copy of hash set
```

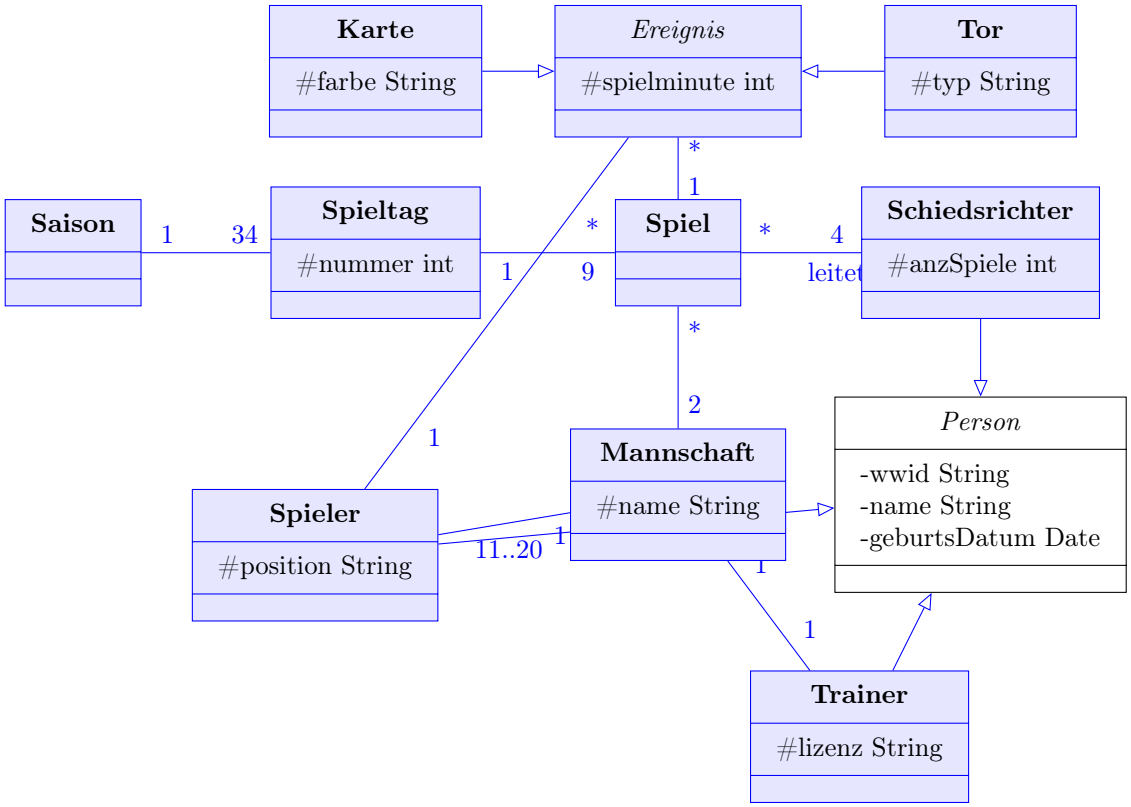
```
        Set<Box> remainingBoxesCopy = new HashSet<>();
        remainingBoxesCopy.addAll(remainingBoxes);

        int maximalSubStackHeight = 0;
        for (Box box : remainingBoxes) {
            if (lastBox == null || box.compareTo(lastBox) < 0) {
                remainingBoxesCopy.remove(box);
                maximalSubStackHeight = max(getMaxHeigth(remainingBoxesCopy,
                ↪  box), maximalSubStackHeight);
                remainingBoxesCopy.add(box);
            }
        }

        return (lastBox != null ? lastBox.height : 0) + maximalSubStackHeight;
    }

    public static void main(String[] args) {
        Set<Box> boxes = new HashSet<>();
        boxes.add(new Box(1, 1, 2, 3));
        boxes.add(new Box(2, 4, 3, 3));
        boxes.add(new Box(3, 2, 2, 5));
        boxes.add(new Box(4, 1, 2, 1));
        boxes.add(new Box(5, 5, 5, 4));

        int maxHeigth = getMaxHeigth(boxes, null);
        System.out.println(maxHeigth);
    }
}
```

Aufgabe 2: UML Fußball Saison

- In einer Fußball-Saison gibt es 34 Spieltage. An jedem Spieltag finden genau 9 Spiele statt. Spieltage werden in aufsteigender Reihenfolge nummeriert.
- Es treffen in jedem Spiel genau zwei Mannschaften aufeinander. Jedes Spiel wird von vier Schiedsrichtern geleitet.
- Jede Mannschaft hat einen Namen, besteht aus mindestens 11, aber maximal 20 Spielern, und hat genau einen Trainer.
- Für die Ereignisse Tor und Karte muss sowohl die Spielminute als auch der beteiligte Spieler gespeichert werden. Außerdem muss der Typ des Tors (z.B. Elfmeter oder Eigentor) und die Farbe der Karte gespeichert werden.
- Für alle Schiedsrichter, Trainer und Spieler wird der Name, das Geburtsdatum sowie eine eindeutige Identifikationsnummer (WWID) im System gespeichert. *Die abstrakte Klasse Person ist bereits gegeben.* Für jeden Schiedsrichter wird zudem die Anzahl bereits geleiteter Spiele gespeichert. Jeder Trainer hat eine bestimmte Lizenz, z.B. A, B, und C. Ein Spieler hat eine Stammposition, z.B. Torwart oder Stürmer.

Ergänzen Sie das folgende UML-Klassendiagramm, in dem bereits die abstrakte Klasse Person gegeben ist, um die noch fehlenden Klassen und Assoziationen, um diesen Sachverhalt strukturell sinvoll darzustellen und Redundanz so weit wie möglich zu vermeiden. Führen Sie passende Attribute ein. Verwenden Sie entsprechende Assoziationen und Multiplizitäten, falls nötig. Sichtbarkeiten (Modifier) der Attribute müssen *nicht* explizit angegeben werden.

Deklarieren Sie bitte in der Klausur abstrakte Klassen eindeutig, z.B. mit «abstract»!



Teil 1: Objektorientierte Modellierung (in UML) und Programmierung (in Java)

Der erste Teil der Vorlesung zu Java und UML ist mit diesem Übungsblatt abgeschlossen. Jetzt ist also *die* Gelegenheit, den Stoff zu wiederholen. Dies erspart Ihnen nicht nur Lernaufwand am Ende des Semesters, sondern Sie haben außerdem noch die Möglichkeit in dieser letzten Übungsstunde zu Java offene Fragen zu klären. Zur Inspiration diese (unvollständige) Liste:

- UML
  - Klassendiagramme, Objektdiagramme
  - Assoziationen (Multiplizitäten, Aggregation vs. Komposition, Navigation)
  - Operationen (Konstruktoren, Beobachterfunktionen, Mutatoren, Access Modifier)
  - Vererbung (Generalisierung, Spezialisierung)
- Java
  - Umsetzung von UML in Java
  - Gemeinsame Unterobjekte
  - Typisierung
  - Garbage Collection
  - Werte vs. Objekte
  - Klassenattribute vs. Objektattribute
  - Methoden-Überladung
  - Vererbung und dynamisches Binden
  - Rekursion
  - Java Collections Framework (Set, List, Queue, Map)
  - Generische Typen
  - Iteration (for each loop, Iteratoren)
- Datenstrukturen
  - Komplexitätsangaben für Operationen ($\mathcal{O}(1)$, $\mathcal{O}(\log n)$, $\mathcal{O}(n)$)
  - Menge (Set), Verkettete Liste (LinkedList), Keller (Stack)
  - Binäre Suchbäume (Degenerierung, AVL-Baum)
  - Hashing (Double Hashing, Hashing with Chaining, Hashing with Linear Probing)

Aufgabe 3: Motivation von DBMS

Nennen Sie drei typische Probleme, die bei dem Verzicht auf ein Datenbankverwaltungssystem eintreten können. Überlegen Sie sich jeweils ein Beispiel bei dem das Problem auftritt.

Im folgenden eine Liste von möglichen Problemen:

Redundanz. Ohne DBMS kommt es leicht zu Redundanz in den Daten und diese kann bei Veränderungen zu Inkonsistenzen führen. Ein DMBS garantiert die **Konsistenz** der Daten.

Beschränkte Zugriffsmöglichkeit. Verknüpfung der Daten ist ohne DBMS sehr schwer, da diese dann oft nicht konsistent modelliert und abgespeichert sind.

Probleme durch Mehrbenutzerbetrieb. Gleichzeitige Änderung der gleichen Daten durch verschiedene Benutzer führt schnell zu Anomalien. Ein DMBS garantiert die logische **Isolation** der Anfragen verschiedener Benutzer.

Datenverlust. Bei Speicherung der Daten in Dateien sind meist höchstens regelmäßige Backups möglich. Ein DBMS garantiert hingegen die **Dauerhaftigkeit** jeder durchgeführten Operation. Weiterhin werden Änderungen ganz oder gar nicht durchgeführt (**Atomarität**).

Integritätsverletzung. In der realen Welt gibt es komplexe Integritätsbedingungen, die sich bei Speicherung in Dateien leicht verletzen lassen.

Sicherheitsprobleme. Ein Beispiel ist der Datenschutz, da nicht alle Benutzer Zugriff auf alle Daten haben sollten. Ein DBMS bietet hierfür Rollen und Zugriffsrechte.

Hohe Entwicklungskosten. Bei der Eigenentwicklung der Datenspeicherung in einem Anwendungsprogramm erfindet man das Rad neu und muss alle oben genannten Probleme lösen. Dies führt zu erheblichen Entwicklungskosten. DBMS hingegen sind getestete Standardkomponenten, die man sehr einfach einsetzen kann.

Die vier wichtigsten Eigenschaften der Datenverarbeitung in einem DBMS haben eine leicht zu merkende Abkürzung: **ACID**. Dies steht für **A**tomicity, **C**onsistency, **I**solation und **D**urability.

Aufgabe 4: Terminologie

Professoren			
PersNr	Name	Rang	Raum
2125	Sokrates	C4	226
2126	Russel	C4	232
2127	Kopernikus	C3	310
2133	Popper	C3	52
2134	Augustinus	C3	309
2136	Curie	C4	36
2137	Kant	C4	7

Abbildung 1: Professoren in der relationalen Modellierung

Beschreiben Sie die folgenden Begriffe der relationalen Modellierung. Verwenden Sie die Relation Professoren aus Abbildung 1 um Beispiele für die einzelnen Konzepte anzugeben:

Attribut • Schlüssel • Relation • Domäne • Tupel • Schema • Ausprägung

Hier eine kurze Beschreibung der Konzepte mit Beispielen:

Attribut. Ein Attribut ist eine Eigenschaft der Entitäten der Relation (entspricht einer Spalte, bei der Relation Professoren z.B. Name).

Schlüssel. Ein Schlüssel identifiziert ein Tupel eindeutig (z.B. die PersNr des Professors).

Relation. Eine Relation besteht aus Schema und Ausprägung (die gesamte Tabelle).

Domäne. Der Wertebereich eines Attributs (bei der PersNr z.B. alle `int` Zahlen).

Tupel. Eine Entität einer Relation (entspricht einer Zeile, z.B. Professor Russel).

Schema. Die Attribute einer Relation mit deren Typen (in diesem Fall `{[PersNr: integer, Name: varchar(30), Rang: varchar(2), Raum: integer]}`).

Ausprägung. Die Gesamtheit aller Tupel einer Relation (alle Zeilen bis auf die Kopfzeilen).