RHEINISCH-WESTFÄLISCHE TECHNISCHE HOCHSCHULE AACHEN

LEHR- UND FORSCHUNGSGEBIET INFORMATIK II
RWTH Aachen · D-52056 Aachen · GERMANY
http://www-i2.informatik.rwth-aachen.de/lufgi2
Prof. Dr. Jürgen Giesl

Probeklausur Lösungsvorschlag
Informatik I - Programmierung
18. 2. 2002

Vorname: _____

Nachname: _____

Matrikelnummer: _____

Studiengang (bitte ankreuzen):

○ Informatik Diplom ○ Informatik Lehramt ○ Sonstige: _____

- Schreiben Sie bitte auf jedes Blatt **Vorname**, **Name** und **Matrikelnummer**.
- Geben Sie Ihre Antworten bitte in lesbarer und verständlicher Form an. Schreiben Sie bitte nicht mit roten Stiften.
- Bitte beantworten Sie die Aufgaben auf den **Aufgabenblättern**. Benutzen Sie auch die Rückseiten der **zur jeweiligen Aufgabe gehörenden** Aufgabenblätter.
- Antworten auf anderen Blättern können nur berücksichtigt werden, wenn **Name, Matrikelnummer und Aufgabennummer** deutlich darauf erkennbar ist.
- Was nicht bewertet werden soll, kennzeichnen Sie bitte durch **Durchstreichen**.
- Werden Täuschungsversuche beobachtet, so wird die Klausur mit **nicht bestanden** bewertet.
- Geben Sie bitte am Ende der Klausur **alle Blätter zusammen mit den Aufgabenblättern ab**.

	Anzahl Punkte	Erreichte Punkte
Aufgabe 1	18	
Aufgabe 2	11	
Aufgabe 3	14	
Aufgabe 4	18	
Aufgabe 5	23	
Aufgabe 6	16	
Summe	100	
Note	-	

Vorname	Name	Matr.-Nr.

Aufgabe 1 (Programmanalyse, 12 + 6 Punkte)

(a) Geben Sie die Ausgabe des Programms für den Aufruf `java M` an. Schreiben Sie hierzu jeweils die ausgegebenen Zeichen hinter den Kommentar "`AUSGABE:`".

```
public abstract class A {

  public static int anzahl = 0;
  public int nr = 0;

  public A () {
    anzahl = anzahl + 1;}
  public void drucke () {
    System.out.println ("A" + this);}
  public String toString() {
    return "(anzahl: " + anzahl + ", nr: " + nr + ")";}
}

public class B extends A {
  public B () {
    nr = anzahl;}
  public B (int nr) {
    this.nr = nr;}
  public void drucke () {
    System.out.println ("B" + this);}
}

public class M {
  public static void main(String[] argumente) {
    B x = new B(10);
    x.drucke();                                  // AUSGABE: B(anzahl: 1, nr: 10)
    A y = new B();
    y.drucke();                                  // AUSGABE: B(anzahl: 2, nr: 2)
    A z = y;
    z.nr = z.nr + B.anzahl;
    z.drucke();                                  // AUSGABE: B(anzahl: 2, nr: 4)
    y.drucke();}                                 // AUSGABE: B(anzahl: 2, nr: 4)
}
```

(b) Die Klasse `B` wird um die Methode `f` erweitert. Finden und erklären Sie die drei Fehler in dieser Methode, die das Compilieren verhindern.

```
public boolean f (A a) {
  A x = new A ();
  B y = a;
  int n = A.nr;
  return n < y.nr;
}
```

Vorname	Name	Matr.-Nr.

- `A x = new A ();` ist nicht erlaubt, da man keine Objekte einer abstrakten Klasse erzeugen kann.
- `B y = a;` ist nicht erlaubt, da man explizit von der Oberklasse A zur Unterklasse B konvertieren muss. Es müsste also z.B. `B y = (B) a` heißen.
- `A.nr` ist nicht erlaubt, denn `nr` ist ein nicht-statisches (Objekt-)Attribut.

Vorname	Name	Matr.-Nr.

Aufgabe 2 (Verifikation, 10 + 1 Punkte)

Der Algorithmus P berechnet den Quotienten zweier natürlicher Zahlen $\mathtt{x}$ und $\mathtt{y}$, wobei $\mathtt{x}$ durch $\mathtt{y}$ teilbar ist.

(a) Vervollständigen Sie die folgende Verifikation des Algorithmus P im Hoare-Kalkül, indem Sie die unterstrichenen Teile ergänzen. Hierbei dürfen zwei Zusicherungen nur dann direkt untereinander stehen, wenn die untere aus der oberen folgt. Hinter einer Programmanweisung darf nur eine Zusicherung stehen, wenn dies aus einer Regel des Hoare-Kalküls folgt.

Algorithmus: P
Eingabe: $\mathtt{x}, \mathtt{y} \in \mathbb{N} = \{0, 1, 2, ...\}$
Ausgabe: $\mathtt{res}$
Vorbedingung: $\mathtt{x} \geq 0 \wedge \mathtt{y} > 0 \wedge \mathtt{y}|\mathtt{x}$ (wobei $\mathtt{y}|\mathtt{x}$ für die Aussage "$\mathtt{y}$ teilt $\mathtt{x}$" steht)
Nachbedingung: $\mathtt{y} * \mathtt{res} = \mathtt{x}$

$\langle\ \mathtt{x} \geq 0 \wedge \mathtt{y} > 0 \wedge \mathtt{y}|\mathtt{x}\ \rangle$

$\langle\ \mathtt{x} \geq 0 \wedge \mathtt{y} > 0 \wedge \mathtt{y}|\mathtt{x} \wedge \mathtt{x} = \mathtt{x}\ \rangle$

```
z = x;
```

$\langle\ \mathtt{x} \geq 0 \wedge \mathtt{y} > 0 \wedge \mathtt{y}|\mathtt{x} \wedge \mathtt{z} = \mathtt{x}\ \rangle$

$\langle\ \mathtt{x} \geq 0 \wedge \mathtt{y} > 0 \wedge \mathtt{y}|\mathtt{x} \wedge \mathtt{z} = \mathtt{x} \wedge 0 = 0\ \rangle$

```
res = 0;
```

$\langle\ \mathtt{x} \geq 0 \wedge \mathtt{y} > 0 \wedge \mathtt{y}|\mathtt{x} \wedge \mathtt{z} = \mathtt{x} \wedge \mathtt{res} = 0\ \rangle$

$\langle\ \mathtt{x} \geq 0 \wedge \mathtt{y} > 0 \wedge \mathtt{y}|\mathtt{x} \wedge \mathtt{y} * \mathtt{res} + \mathtt{z} = \mathtt{x}\ \rangle$

```
while (z != 0) {
```

$\langle\ \mathtt{x} \geq 0 \wedge \mathtt{y} > 0 \wedge \mathtt{y}|\mathtt{x} \wedge \mathtt{z} \neq 0 \wedge \mathtt{y} * \mathtt{res} + \mathtt{z} = \mathtt{x}\ \rangle$

$\langle\ \mathtt{x} \geq 0 \wedge \mathtt{y} > 0 \wedge \mathtt{y}|\mathtt{x} \wedge \mathtt{y} * (\mathtt{res} + 1) + (\mathtt{z} - \mathtt{y}) = \mathtt{x}\ \rangle$

```
  res = res + 1;
```

$\langle \mathtt{x} \geq 0 \wedge \mathtt{y} > 0 \wedge \mathtt{y}|\mathtt{x} \wedge \mathtt{y} * \mathtt{res} + (\mathtt{z} - \mathtt{y}) = \mathtt{x}\ \rangle$

```
  z = z - y;
```

$\langle\ \mathtt{x} \geq 0 \wedge \mathtt{y} > 0 \wedge \mathtt{y}|\mathtt{x} \wedge \mathtt{y} * \mathtt{res} + \mathtt{z} = \mathtt{x}\ \rangle$

```
}
```

$\langle\ \mathtt{x} \geq 0 \wedge \mathtt{y} > 0 \wedge \mathtt{y}|\mathtt{x} \wedge \mathtt{z} = 0 \wedge \mathtt{y} * \mathtt{res} + \mathtt{z} = \mathtt{x}\ \rangle$

$\langle \mathtt{y} * \mathtt{res} = \mathtt{x} \rangle$

Vorname	Name	Matr.-Nr.

(b) Geben Sie (ohne Beweis) eine Variante für die `while`-Schleife an, die belegt, dass die Schleife für $\mathtt{x} \geq \mathtt{0} \wedge \mathtt{y} > \mathtt{0} \wedge \mathtt{y} | \mathtt{x}$ terminiert.

Die Variante ist z. Da nach Voraussetzung $y > 0$ gilt, verkleinert sich der Wert von z mit jedem Schleifendurchlauf.

Vorname	Name	Matr.-Nr.

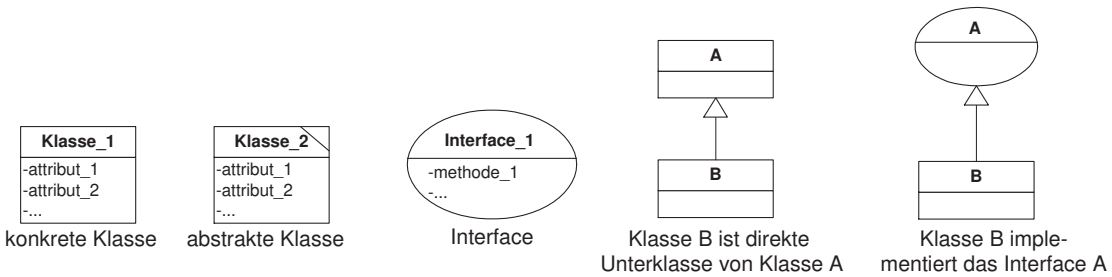
Aufgabe 3 (Datenstrukturen in Java, 5 + 3 + 6 Punkte)

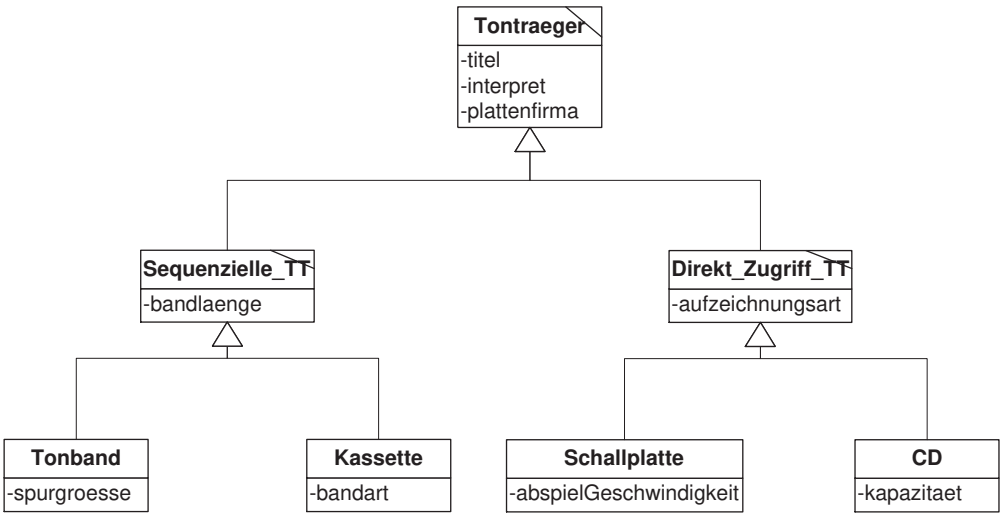
Sie haben die Aufgabe, ein Programm zu entwickeln, mit dem Tonträger verwaltet werden können. Dabei stellen Sie fest, dass es die folgenden vier verschiedenen Arten von Tonträgern gibt:

- Tonband: dieses ist charakterisiert durch Angabe von Titel, Interpret, Plattenfirma, Bandlänge und Spurgröße
- Schallplatte: diese ist charakterisiert durch Angabe von Titel, Interpret, Plattenfirma, Aufzeichnungsart und Abspielgeschwindigkeit
- CD: diese ist charakterisiert durch Angabe von Titel, Interpret, Plattenfirma, Aufzeichnungsart und Kapazität
- Kassette: diese ist charakterisiert durch Angabe von Titel, Interpret, Plattenfirma, Bandlänge und Bandart

(a) Entwerfen Sie eine geeignete Klassenhierarchie, um die oben aufgelisteten Arten von Tonträgern zu implementieren.

Achten Sie dabei darauf, dass gemeinsame Merkmale in abstrakten Klassen zusammengefasst werden. Notieren Sie Ihren Entwurf grafisch und verwenden Sie dazu die folgende Notation (geben Sie lediglich pro Klasse den Namen der Klasse und die Namen ihrer Attribute an und pro Interface den Namen des Interfaces und die Namen seiner Methoden). Verwenden Sie ausschließlich den Typ String für die Attribute.





(b) Implementieren Sie in der Klasse `Kassette` und in allen ihren Oberklassen, die sie definiert haben, je eine Methode `public String toString()`, welche eine String-Repräsentation aller Merkmale eines Tonträgers zurückliefert. Verwenden Sie Vererbungsmechanismen soweit wie möglich.

In `Tontraeger`:

```
public String toString() {
  return titel + ", " + interpret + ", " + plattenfirma;
}
```

In `Direkt_Zugriff_TT`:

```
public String toString() {
  return super.toString() + ", " + bandlaenge;
}
```

In `Kassette`:

```
public String toString() {
  return super.toString() + ", " + bandart;
}
```

(c) Implementieren Sie eine Methode `schallplattenfilter`, die ein Array von Tonträgern als Eingabe bekommt. Die Methode soll untersuchen, welche dieser Tonträger Schallplatten sind. Diese sollen als ein Array von Schallplatten (richtiger Länge) als Resultat zurückgeliefert werden.

```
public static Schallplatte[] schallplattenfilter (Tontraeger[] t) {


   int laenge = 0,
       j = 0;

   if (t == null)
      return null;

   for (int i = 0; i < t.length; i++)
       if (t[i] instanceof Schallplatte)
          laenge++;

   Schallplatte[] s = new Schallplatte[laenge];

   for (int i = 0; i < t.length; i++)
       if (t[i] instanceof Schallplatte) {
          s[j] = (Schallplatte) t[i];
          j++;
       }

   return s;

}
```

Vorname	Name	Matr.-Nr.

Aufgabe 4 (Programmierung in Java, 8 + 10 Punkte)

In dieser Aufgabe soll ein Städteverbindungsnetz realisiert werden. Durch eingehende Analyse wurde eine Schnittstellenspezifikation ermittelt, von der hier ein Teil angegeben ist.

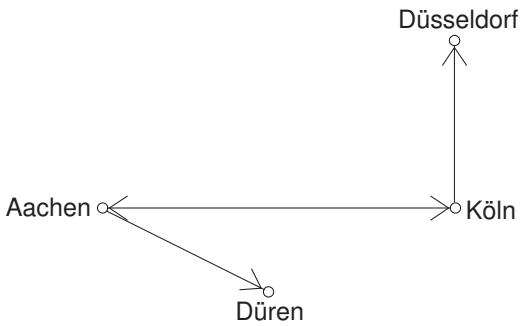
- **Klasse:** `public class Stadt`
  Methode: `public boolean gleich (Stadt s)`
  `/* Prüft, ob s gleich der aktuellen Stadt ist */`

- **Klasse:** `public class Stadtnetz`
  Attribute: `private Stadtelement kopf`
  Methoden: `public void stadtEinfuegen (Stadt s)`
  `/* Fügt eine neue Stadt in das Stadtnetz ein */`
  `public void verbindungEinfuegen (Stadt von, Stadt nach)`
  `/* Fügt eine Verbindung zwischen zwei Städten ein */`

- **Klasse:** `public class Verbindungselement`
  Attribute: `private Stadt wert`
  `private Verbindungselement nextVerbindung`
  Konstruktor: `public Verbindungselement (Stadt s, Verbindungselement next)`
  `/* Erzeugt neues Verbindungselement mit Attributen s und next */`
  Methoden: `public Stadt getWert ()`
  `public void setWert (Stadt wert)`
  `public Verbindungselement getNextVerbindung ()`
  `public void setNextVerbindung (Verbindungselement next)`
  `/* Selektoren für die Attribute */`

- **Klasse:** `public class Stadtelement extends Verbindungselement`
  Attribute: `private Stadtelement nextStadtelement`
  Konstruktor: `public Stadtelement (Stadt s, Verbindungselement vnext, Stadtelement snext)`
  `/* Erzeugt neues Stadtelement mit Attributen s, vnext und snext */`
  Methoden: `public Stadtelement getNextStadtelement ()`
  `public void setNextStadtelement (Stadtelement next)`
  `/* Selektoren für die Attribute */`

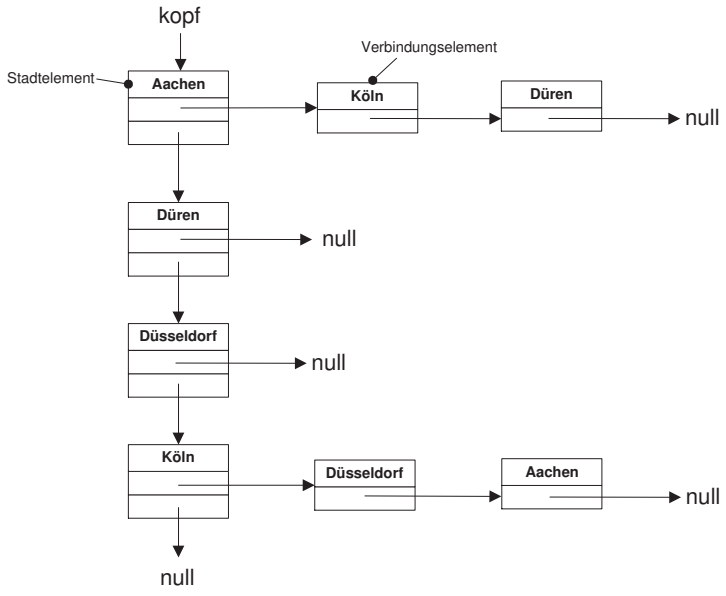
Als Beispiel betrachten wir das folgende Städteverbindungsnetz:



Dieses Städteverbindungsnetz könnte wie folgt realisiert werden.

Ein Stadtnetz ist also im Prinzip eine Liste von Städten, wobei aber zu jeder Stadt `A` eine der Liste der Städte $B_1, \ldots, B_n$ gespeichert wird, die man von ihr aus erreichen kann (d.h., bei denen es Verbindungen von `A` nach B_1, von `A` nach B_2, ..., von `A` nach B_n gibt).

Vorname	Name	Matr.-Nr.



(a) Implementieren Sie die Methode `public void stadtEinfuegen (Stadt s)` der Klasse `Stadtnetz` in Java. Die einzufügende Stadt soll dabei an das Ende der bereits existierenden Liste angehängt werden. Hierbei spielt es keine Rolle, ob die Stadt bereits in dem `Stadtnetz` auftritt. Die Implementierung muss *rekursiv* (ohne Verwendung von Schleifen) realisiert werden. Hierbei dürfen Sie Hilfsmethoden definieren, die aber nach außen nicht sichtbar sein sollen.

```
public void stadtEinfuegen (Stadt s) {

    if   (kopf == null)
         kopf = new Stadtelement (s, null, null);
    else stadtEinfuegen (s, kopf);
}

private static void stadtEinfuegen (Stadt s, Stadtelement element) {

    if   (element.getNextStadtelement() == null)
         element.setNextStadtelement (new Stadtelement (s, null, null));
    else stadtEinfuegen (s, element.getNextStadtelement());

}
```

Vorname	Name	Matr.-Nr.

(b) Implementieren Sie die Methode `public void verbindungEinfuegen(Stadt von, Stadt nach)` der Klasse `Stadtnetz` in Java. Auch hier soll eine neue Verbindung an das Ende der Liste angehängt werden und es spielt keine Rolle, ob die Verbindung bereits in dem `Stadtnetz` auftritt. Gehen Sie davon aus, dass beide übergebenen Städte im Stadtnetz vorhanden sind. Hierbei dürfen Sie wieder Hilfsmethoden definieren, die aber nach außen nicht sichtbar sein sollen.

```
public void verbindungEinfuegen (Stadt von, Stadt nach) {

    Verbindungselement verb;
    Stadtelement element = kopf;

    while (! element.getWert().gleich(von))
        element = element.getNextStadtelement();

    verb = element;

    while (verb.getNextVerbindung() != null)
        verb = verb.getNextVerbindung();

    verb.setNextVerbindung (new Verbindungselement(nach, null));
}
```

Vorname	Name	Matr.-Nr.

Aufgabe 5 (Funktionale Programmierung in Haskell, 4 + 2 + 3 + 4 + 2 + 8 Punkte)

(a) Die Funktionen `f`, `g`, und `h` sind wie folgt definiert:

```
f x = x + 1

g x = [2, 4] ++ x

h x =  \y -> [x] : [[y]]
```

Geben Sie den allgemeinsten Typ von `f`, `g` und `h` an. Gehen Sie hierbei davon aus, dass `2` und `4` den Typ `Int` haben und `+` den Typ `Int -> Int -> Int` hat.

Der allgemeinste Typ von `f` ist `Int -> Int`, der allgemeinste Typ von `g` ist `[Int] -> [Int]` und der allgemeinste Typ von `h` ist `a -> a -> [[a]]`.

(b) Geben Sie für die folgenden Ausdrücke jeweils das Ergebnis der Auswertung an.

(i) `(\x -> x 2) (\x -> x * 3)`

(ii) `map (\x -> x * 2) [2, 3, 4]`

(iii) `filter (\x -> x + 3 == 7) (map (\x -> x * 2) [2, 3, 4])`

(i) `6`

(ii) `[4,6,8]`

(iii) `[4]`

(c) Schreiben Sie eine Funktion `nth :: Int -> [a] -> a`, die zu einer Zahl `x` mit $1 \leq x \leq n$ und einer Liste $[e_1, \ldots, e_n]$ das `x`-te Element e_x liefert. Beispielsweise berechnet `nth 3 ['a','b','c']` das Ergebnis `'c'`. Hierbei dürfen Sie keine in Haskell vordefinierten Funktionen auf Listen außer den Datenkonstruktoren `[]` und `:` verwenden.

```
nth :: Int -> [a] -> a
nth 1     (y:ys) = y
nth (x+1) (y:ys) = nth x ys
```

(d) Eine Tabelle besteht aus Zeilen und Spalten, die Werte beliebigen Typs aufnehmen können. Jede Position in der Tabelle ist durch ihre Zeilen- und Spaltennummer gekennzeichnet. Eine Zeile kann als Liste von Einträgen realisiert werden und eine Tabelle kann dann als Liste von Zeilen dargestellt werden. Insgesamt lassen sich also Tabellen durch Listen von Listen wie folgt realisieren: Die Tabelle

	1	2	3
1	'a'	'b'	'c'
2	'd'	'e'	'f'

wird dann durch `[ ['a','b','c'], ['d','e','f'] ]` repräsentiert. Der Wert `'b'` steht hierbei in der ersten Zeile und der zweiten Spalte (d.h. an der Position `(1,2)`) und `'f'` steht in der zweiten Zeile und der dritten Spalte, d.h. an Position `(2,3)`.

Implementieren Sie in Haskell eine Funktion `summe :: Int -> [[Int]] -> Int`, die die Summe der Einträge einer vorgegebenen Spalte in der Tabelle liefert. Beispielsweise ergibt also `summe 2 [ [10,2,33,14], [4,5,6,8] ]` das Resultat `7`. In dieser Teilaufgabe betrachten wir nur Tabellen zur Speicherung von `Int`-Werten.

```
summe :: Int -> [[Int]] -> Int
summe x []            = 0
summe x (zeile:rest) = nth x zeile + summe x rest
```

(e) Definieren Sie in Haskell eine solche Datenstruktur zur Repräsentation von Tabellen mit Einträgen beliebigen Typs, welche der Beschreibung in Teil (d) entspricht. Durch Ihre Datenstruktur muss sichergestellt sein, dass an jeder Position der Tabelle nur ein Eintrag stehen kann. Hierbei dürfen Sie jedoch die in Haskell vordefinierten Listen *nicht* verwenden.

```
data Tab a = EmptyTab | AddRow (Row a) (Tab a) deriving Show

data Row a = EmptyRow | AddEntry a (Row a) deriving Show
```

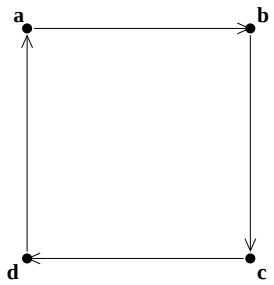
(f) Implementieren Sie in Haskell eine Funktion `suche`, die zu einem Eintrag und einer Tabelle eine Position angibt, an der dieser Eintrag in der Tabelle steht. Positionen sind hierbei wieder Paare von Zahlen `(n,m)`, wobei `n` die Zeilen- und `m` die Spaltennummer ist. Falls die Tabelle den Eintrag nicht enthält, soll `(0,0)` zurückgegeben werden. Verwenden Sie hierbei Ihre Datenstruktur aus Teil (e) und geben Sie auch die Typdeklaration von `suche` an. Sie dürfen davon ausgehen, dass der Vergleichsoperator `(==)` für alle Typen vordefiniert ist.
Hinweis: Implementieren Sie gegebenenfalls geeignete Hilfsfunktionen.

```
suche :: a -> Tab a -> (Int,Int)
suche x t = sucheVon x t 1

  where
       {- "sucheVon x t n" berechnet (n-1+zeile,spalte), falls x in der Tabelle
            t an der Position (zeile,spalte) auftritt und (0,0) sonst. -}
        sucheVon :: a -> Tab a -> Int -> (Int,Int)
        sucheVon x EmptyTab n = (0,0)
        sucheVon x (AddRow zeile rest) n | spalte == 0 = sucheVon x rest (n+1)
                                         | otherwise   = (n,spalte)
                                         where spalte = sucheInZeile x zeile 1

        {- "sucheInZeile x z m" berechnet m-1+spalte, falls x in der Zeile z
           an der Stelle "spalte" auftritt und 0 sonst -}
        sucheInZeile :: a -> Row a -> Int -> Int
        sucheInZeile x EmptyRow m = 0
        sucheInZeile x (AddEntry y rest) m | x == y = m
                                           | otherwise = sucheInZeile x rest (m+1)
```

Vorname	Name	Matr.-Nr.

Aufgabe 6 (Logische Programmierung in Prolog, 1 + 3 + 2 + 5 + 5 Punkte)

(a) Gegeben sei folgender Graph mit den Knoten **a**, **b**, **c**, **d**.



Erstellen Sie ein Prolog-Programm, in dem dieser Graph mit Hilfe eines zweistelligen Prädikats `kante` repräsentiert wird.

```
kante(a,b).
kante(b,c).
kante(c,d).
kante(d,a).
```

(b) Definieren Sie in Prolog ein einstelliges Prädikat `pfad`, wobei `pfad(L)` ausdrückt, dass die Liste `L` einen Pfad in dem Graph repräsentiert.
Ein Pfad in einem Graph `G` ist eine Liste $[\mathtt{V}_1, \ldots, \mathtt{V}_n]$ von Knoten aus `G` (mit $\mathtt{n} \geq 0$), so dass eine Kante von $\mathtt{V}_i$ nach $\mathtt{V}_{i+1}$ in `G` existiert für alle $\mathtt{i} \in \{1, .., \mathtt{n}-1\}$. Im Beispiel-Graphen gälte also z.B. `pfad([a,b,c])`.

```
pfad([]).
pfad([_]).
pfad([X,Y|R]) :- kante(X,Y), pfad([Y|R]).
```

Vorname	Name	Matr.-Nr.

(c) Formulieren Sie (basierend auf dem zuvor definierten Prädikat) eine Anfrage, um alle Pfade zu berechnen, die aus mindestens zwei Knoten bestehen und im Knoten `a` beginnen.

```
?- pfad(L), L = [a,_|_].
```

(d) Definieren Sie in Prolog ein zweistelliges Prädikat `min`, wobei `min(X,L)` ausdrückt, dass `X` das Minimum einer Liste `L` ist. Gehen Sie hierbei davon aus, dass `L` eine Liste von Zahlen ist. Es gilt also z.B. `min(2, [5,2,7])`.
Hinweis: Es gibt in Prolog vordefinierte Prädikate `=<` und `>=`.

```
min(X,[X]).
min(X,[Y,Z|L]) :- Y =< Z, min(X,[Y|L]).
min(X,[Y,Z|L]) :- Y >= Z, min(X,[Z|L]).
```

(e) Betrachten Sie folgendes Prolog-Programm:

```
nachfolger(X,succ(X)).
nachfolger(X,Y) :- nachfolger(X,Z), nachfolger(Z,Y).
```

Terminiert das Programm bei der Suche nach den ersten beiden Antworten auf die folgende Anfrage? Geben Sie ggf. die erste bzw. die ersten beiden Antworten an.

```
?- nachfolger(zero, U).
```

Das Programm liefert die beiden ersten Antworten `U = succ(zero)` und `U = succ(succ(zero))`.