RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
AACHEN

LEHR- UND FORSCHUNGSGEBIET INFORMATIK 2
RWTH Aachen · D-52056 Aachen · GERMANY
http://programmierung.informatik.rwth-aachen.de/
Prof. Dr. Jürgen Giesl

LuFG Informatik II

Prof. Dr. Jürgen Giesl
Fabian Emmes, Carsten Fuhs, Carsten Otto

Prüfungsklausur Programmierung

25. 3. 2009

Vorname: _____

Nachname: _____

Matrikelnummer: _____

Studiengang (bitte ankreuzen):

○ Informatik Bachelor ○ Informatik Diplom ○ Informatik Lehramt
○ Mathematik Bachelor

○ 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 oder mit Bleistiften.
- Bitte beantworten Sie die Aufgaben auf den **Aufgabenblättern**. Benutzen Sie ggf. 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 sind.
- Was nicht bewertet werden soll, kennzeichnen Sie bitte durch **Durchstreichen**.
- Werden Täuschungsversuche beobachtet, so wird die Klausur mit **0 Punkten** bewertet.
- Geben Sie bitte am Ende der Klausur **alle Blätter** zusammen mit den Aufgabenblättern ab.

	Anzahl Punkte	Erreichte Punkte
Aufgabe 1	14	
Aufgabe 2	12	
Aufgabe 3	14	
Aufgabe 4	26	
Aufgabe 5	17	
Aufgabe 6	17	
Summe	100	
Prozentzahl		

Vorname	Name	Matr.-Nr.

Aufgabe 1 (Programmanalyse, 8 + 6 Punkte**)**

a) Geben Sie die Ausgabe des Programms für den Aufruf `java M` an. Schreiben Sie hierzu jeweils die ausgegebenen Zeichen in die Kästchen hinter den Kommentar „`OUT:`“.

```
public class A {
    public int x = 0;
    public A() {
        this.x = 1;
    }
    public void f(double x) {
        this.x += (int) x;
    }
}

public final class B extends A {
    public int x = 2;
    public B() {
        this.x = 3 + super.x;
    }
    public void f(double x) {
        this.x += 5;
    }
    public void f(int x) {
        this.x += 10;
    }
}

public class M {
    public static void main(String[] args) {
        A a = new A();
        System.out.println(a.x);              // OUT: [  1 ]
        a.f(2);
        System.out.println(a.x);              // OUT: [  3 ]
        B b = new B();
        System.out.println(b.x);              // OUT: [  4 ]
        b.f(2);
        System.out.println(b.x);              // OUT: [ 14 ]
        b.f(4.0);
        System.out.println(b.x);              // OUT: [ 19 ]
        A z = b;
        z.f(1);
        System.out.println(b.x + " " + z.x
                        + " " + ((A) b).x);   // OUT: [ 24 ] [  1 ] [  1 ]
    }
}
```

Vorname	Name	Matr.-Nr.

b) Wir schreiben zusätzlich zu `A` und `B` eine neue Klasse `C`. Welche drei Fehler treten beim Compilieren auf? Begründen Sie Ihre Antwort kurz.

```
public class C extends B {
    private C(String string) {
        string += string;
        super();
    }

    public void f(int x) {
        System.out.println(x/0);
    }

    public int g(int x) {
        return this.x;
    }

    public int g(int y) {
        f(y);
        return y;
    }
}
```

- *Die Klasse* `B` *ist als* `final` *deklariert, darf also nicht von Klasse* `C` *erweitert werden (*`C extends B`*)*
- *Der Konstruktoraufruf* `super()` *muss die erste Anweisung im Konstruktor* `C(String string)` *sein*
- *Die beiden Methoden* `g` *mit Argument vom Typ* `int` *können so nicht überladen werden, da sich ihre Signaturen nur im Namen des Parameters unterscheiden. Eine Überladung ist nur zulässig, falls sich die Argument-Typen unterscheiden.*

Vorname	Name	Matr.-Nr.

Aufgabe 2 (Verifikation, 10 + 2 Punkte**)**

Der Algorithmus P berechnet für jede natürliche Zahl $n \in \mathbb{N}$ den Wert 0.

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: $n \in \mathbb{N}$
Ausgabe: res
Vorbedingung: $n \geq 0$
Nachbedingung: $res = 0$

$\langle\ n \geq 0\ \rangle$

$\langle\ n \geq 0 \wedge 12 * n = 12 * n\ \rangle$

$res = 12 * n;$

$\langle\ n \geq 0 \wedge res = 12 * n\ \rangle$

$\langle\ n \geq 0 \wedge res = 12 * n \wedge 0 = 0\ \rangle$

$k = 0;$

$\langle\ n \geq 0 \wedge res = 12 * n \wedge k = 0\ \rangle$

$\langle\ k \leq 6 * n \wedge res = 12 * n - 2 * k\ \rangle$

`while` $(k < 6 * n)$ {

$\langle\ k \leq 6 * n \wedge res = 12 * n - 2 * k \wedge k < 6 * n\ \rangle$

$\langle\ k < 6 * n \wedge res - 2 = 12 * n - 2 * k - 2\ \rangle$

$res = res - 2;$

$\langle\ k < 6 * n \wedge res = 12 * n - 2 * k - 2\ \rangle$

$\langle\ k + 1 \leq 6 * n \wedge res = 12 * n - 2(k + 1)\ \rangle$

$k = k + 1;$

$\langle\ k \leq 6 * n \wedge res = 12 * n - 2 * k\ \rangle$

}

$\langle\ k \leq 6 * n \wedge res = 12 * n - 2 * k \wedge k \not< 6 * n\ \rangle$

$\langle\ res = 0\ \rangle$

Vorname	Name	Matr.-Nr.

b) Beweisen Sie die Terminierung des Algorithmus P.

Wir wählen die Variante $6 * n - k$. Dann gilt $k < 6 * n \Rightarrow 6 * n - k \geq 0$ und

$$\langle\ 6 * n - k = m \wedge k < 6 * n\ \rangle$$

$$\langle\ 6 * n - (k + 1) < m\ \rangle$$

$res = res - 1;$

$$\langle\ 6 * n - (k + 1) < m\ \rangle$$

$k = k + 1;$

$$\langle\ 6 * n - k < m\ \rangle$$

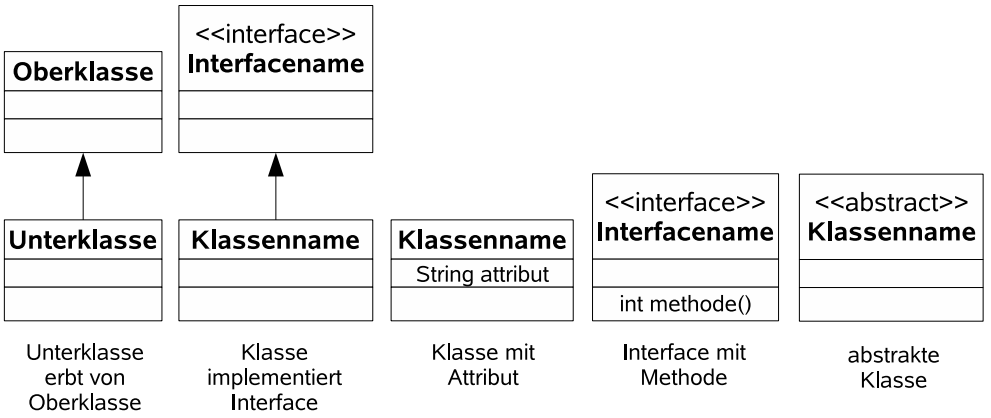
Vorname	Name	Matr.-Nr.

Aufgabe 3 (Datenstrukturen in Java, 6 + 8 Punkte**)**

Ihre Aufgabe ist es, eine objektorientierte Datenstruktur zur Verwaltung von Musikinstrumenten zu entwerfen. Bei der vorangehenden Analyse wurden folgende Eigenschaften der verschiedenen Musikinstrumente ermittelt.

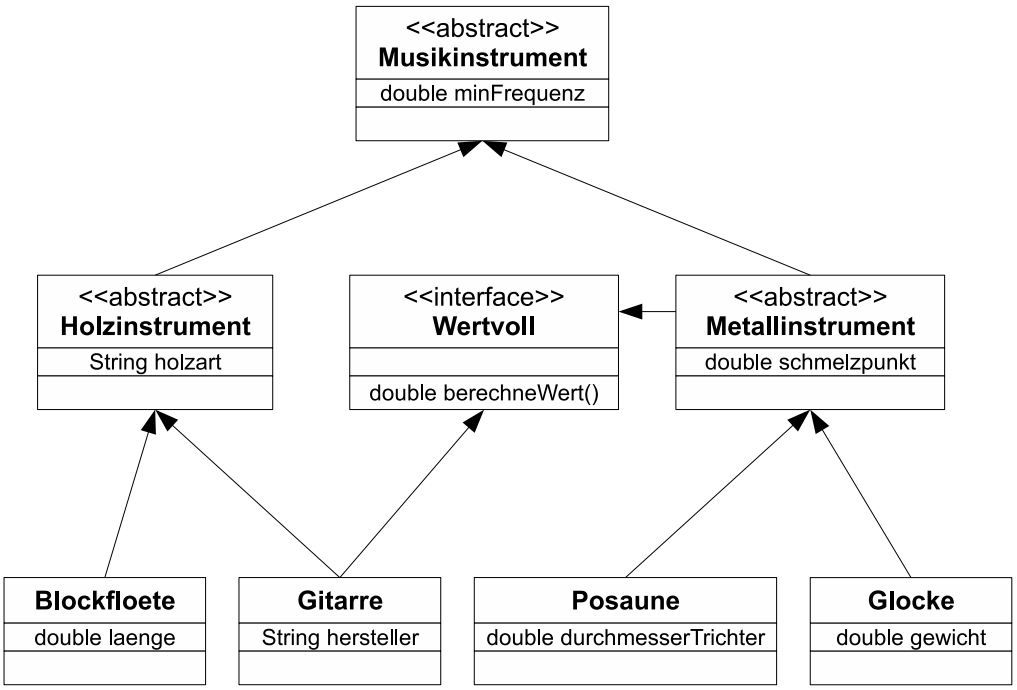
- Jedes Musikinstrument wird durch die minimale Frequenz gekennzeichnet, die sich mit dem Instrument erzeugen lässt.
- Ein Holzinstrument ist ein Musikinstrument, das aus einer bestimmten Holzart gefertigt ist.
- Eine Gitarre ist ein Holzinstrument, für das der Name des Herstellers ein wichtiges Attribut ist.
- Eine Blockflöte ist ebenfalls ein Holzinstrument. Hierbei gibt es Blockflöten mit unterschiedlichen Längen.
- Ein Metallinstrument ist ein Musikinstrument, für das der Schmelzpunkt des Instrumentes in Grad Celsius interessant ist.
- Eine Posaune ist ein Metallinstrument. Bei Posaunen ist der Durchmesser des Trichters eine wichtige Kenngröße.
- Glocken sind ebenfalls Metallinstrumente. Für eine Glocke ist ihr Gewicht ein kennzeichnender Wert.
- Alle Musikinstrumente aus Metall und Gitarren haben einen hohen Wiederverkaufswert. Daher gibt es für diese Musikinstrumente eine Methode, die ihren Wert in Euro berechnet.

a) Entwerfen Sie unter Berücksichtigung der Prinzipien der Datenkapselung eine geeignete Klassenhierarchie für die oben aufgelisteten Arten von Musikinstrumenten. Achten Sie darauf, dass gemeinsame Merkmale in (evtl. abstrakten) Oberklassen zusammengefasst werden. Notieren Sie Ihren Entwurf graphisch und verwenden Sie dazu die folgende Notation:



Geben Sie für jede Klasse ausschließlich den jeweiligen Namen und die Namen und Datentypen ihrer Attribute an. Methoden von Klassen müssen nicht angegeben werden. Geben Sie für jedes Interface ausschließlich den jeweiligen Namen sowie die Namen und Ein- und Ausgabetypen seiner Methoden an.

Vorname	Name	Matr.-Nr.



Vorname	Name	Matr.-Nr.

b) Implementieren Sie in Java eine Methode `wertvollstesInstrument`. Die Methode bekommt als Parameter ein Array von Musikinstrumenten übergeben. Sie soll als Ergebnis das wertvollste Instrument des Arrays zurückliefern (wobei nur Musikinstrumente aus Metall und Gitarren einen Wert haben). Falls das Array kein solches Instrument enthält, soll `null` zurückgegeben werden.

Gehen Sie dabei davon aus, dass das übergebene Array nicht der `null`-Wert ist und dass es keine `null`-Werte enthält. Kennzeichnen Sie die Methode mit dem Schlüsselwort „`static`“, falls angebracht.

```
public static Wertvoll wertvollstesInstrument(Musikinstrument[] array) {
    Wertvoll ergebnis = null;

    for (int i = 0; i < array.length; i++) {
        if (array[i] instanceof Wertvoll) {
            Wertvoll w = (Wertvoll) array[i];
            if (ergebnis == null ||
                    w.berechneWert() > ergebnis.berechneWert()) {
                ergebnis = w;
            }
        }
    }

    return ergebnis;
}

// Ob die Methode den Rueckgabetypen Wertvoll oder den Rueckgabetypen
// Musikinstrument hat, ist fuer die Punktevergabe unerheblich.
// Der Code sollte natuerlich korrekter Java-Code sein (ggf. muss man
// das Objekt vor der Rueckgabe also noch casten).
```

Vorname	Name	Matr.-Nr.

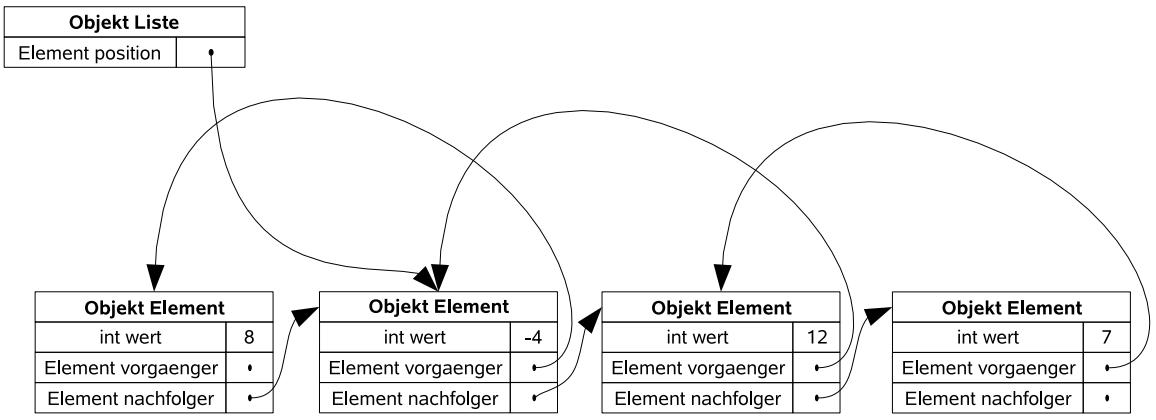
Aufgabe 4 (Programmierung in Java, 6 + 4 + 8 + 8 Punkte**)**

Die Klasse `Liste` repräsentiert eine Liste von Zahlen und verwendet dazu die Klasse `Element`. Dabei enthält jedes Objekt der Klasse `Element` eine Zahl, einen Verweis auf den Vorgänger und einen Verweis auf den Nachfolger. Die Zahl wird in dem Attribut `wert` gespeichert, das Attribut `vorgaenger` zeigt auf das vorhergehende Element in der Liste und das Attribut `nachfolger` zeigt auf das nächste Element in der Liste. Das letzte Element einer Liste hat keinen Nachfolger, so dass dessen Attribut `nachfolger` auf `null` zeigt. Entsprechend hat das erste Element einer Liste keinen Vorgänger, so dass dessen Attribut `vorgaenger` auf `null` zeigt. Objekte der Klasse `Liste` haben ein Attribut `position`, das auf ein Element der Liste zeigt. Eine leere Liste hat keine Elemente, so dass hier das Attribut `position` auf `null` zeigt.

```
public class Liste {
    private Element position;
}

public class Element {
    public int wert;
    public Element vorgaenger;
    public Element nachfolger;
}
```

Ein Beispiel für solch eine Liste hätte etwa die folgende Repräsentation im Speicher:



Verwenden Sie bei der Implementierung der folgenden Methoden die Schlüsselwörter `public` und `private` auf sinnvolle Weise und kennzeichnen Sie Methoden als `static`, falls angebracht.

Vorname	Name	Matr.-Nr.

a) Implementieren Sie die Methode

`public boolean strukturtest()`

der Klasse `Liste`.

Diese Methode soll testen, ob die Struktur der Verweise in der aktuellen Liste intakt ist.

Die Struktur der leeren Liste ist immer intakt. Die Struktur einer nicht-leeren Liste `l` ist genau dann intakt, wenn alle Elemente `e`, die von `l.position` aus durch (wiederholtes) Durchlaufen der Attribute `nachfolger` oder `vorgaenger` erreichbar sind, die folgenden beiden Bedingungen erfüllen:

- Der Nachfolger des Vorgängers von `e` ist `e` selbst, oder der Vorgänger von `e` ist `null`.
- Der Vorgänger des Nachfolgers von `e` ist `e` selbst, oder der Nachfolger von `e` ist `null`.

Gehen Sie davon aus, dass die Liste keine `nachfolger`-Zyklen enthält, d.h., wenn man den `nachfolger`-Verweisen folgt, erreicht man irgendwann `null`. Genauso gibt es auch keine `vorgaenger`-Zyklen. Sie dürfen beliebige Hilfsmethoden einführen. Verwenden Sie dabei aber keine Schleifen, sondern ausschließlich Rekursion.

```
public boolean strukturtest() {
    return strukturLinks(this.position)
        && strukturRechts(this.position);
}

private static boolean strukturLinks(Element e) {
    if (e == null || e.vorgaenger == null) {
        return true;
    }
    if (e.vorgaenger.nachfolger != e) {
        return false;
    }
    return strukturLinks(e.vorgaenger);
}

private static boolean strukturRechts(Element e) {
    if (e == null || e.nachfolger == null) {
        return true;
    }
    if (e.nachfolger.vorgaenger != e) {
        return false;
    }
    return strukturRechts(e.nachfolger);
}
```

Vorname	Name	Matr.-Nr.

b) Implementieren Sie die Methode

```
public Element erstes()
```

der Klasse `Liste`, die das *erste* Element der Liste zurückgibt. Das erste Element der Liste aus unserem Beispiel enthält z.B. die Zahl `8`. Falls die aktuelle Liste leer ist, soll der Wert `null` zurückgegeben werden.

Gehen Sie bei Ihrer Implementierung davon aus, dass die Struktur der aktuellen Liste intakt im Sinne von Aufgabenteil (a) ist.

Sie dürfen beliebige Hilfsmethoden einführen. Verwenden Sie dabei aber keine Schleifen, sondern ausschließlich Rekursion.

```
public Element erstes() {
    return erstes(this.position);
}

private static Element erstes(Element e) {
    if (e == null || e.vorgaenger == null) {
        return e;
    }
    return erstes(e.vorgaenger);
}
```

c) Implementieren Sie die Methode

```
public int aenderePosition(int n)
```

der Klasse `Liste`.

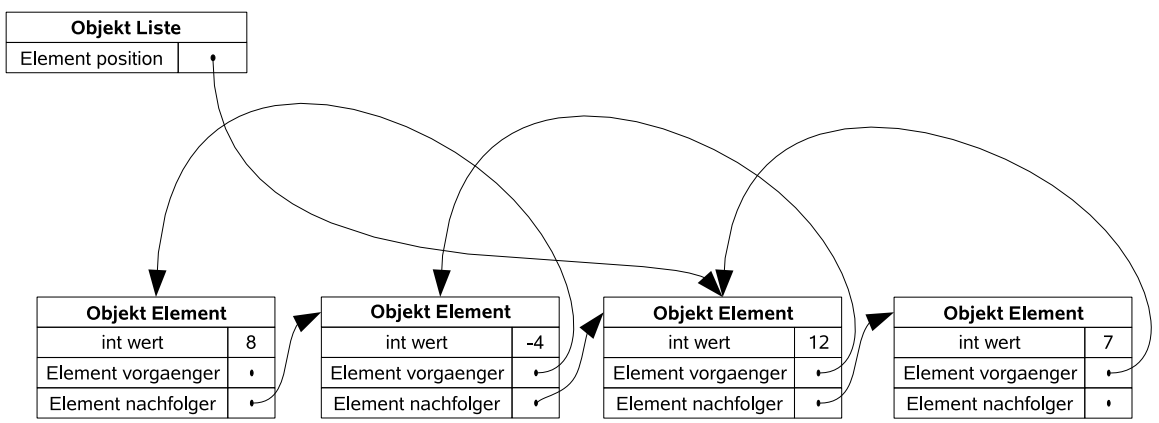
Für eine leere `Liste` hat die Methode unabhängig von `n` keine Seiteneffekte und gibt den Wert `0` zurück.

Für `n` ≥ 0 verschiebt diese Methode auf einer nicht-leeren Liste als Seiteneffekt das Attribut `position` der aktuellen `Liste` um `n` Elemente nach hinten, jedoch höchstens bis zum letzten Element der aktuellen Liste. Die Attribute der `Element`-Objekte werden hierbei nicht verändert.

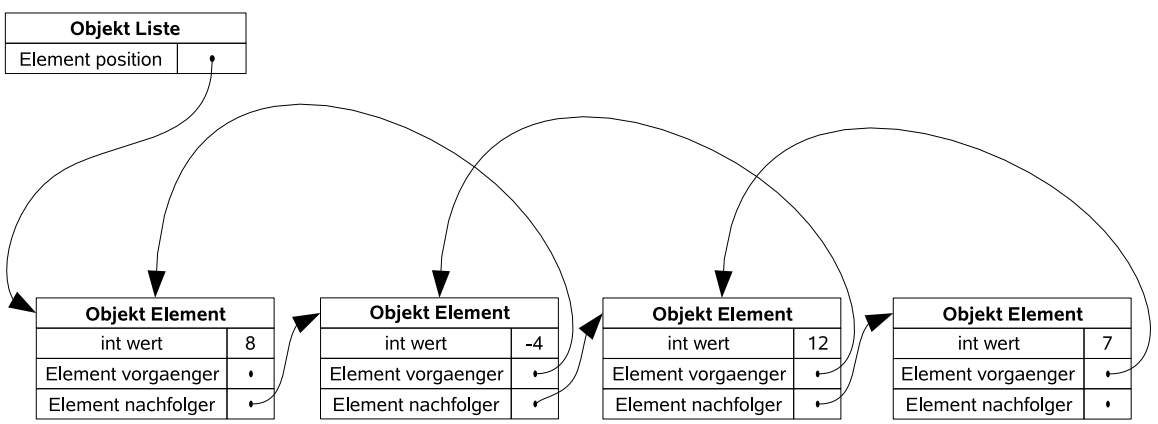
Entsprechend verschiebt die Methode für `n` < 0 auf einer nicht-leeren Liste als Seiteneffekt das Attribut `position` um `n` Elemente nach *vorne*, jedoch höchstens bis zum ersten Element der aktuellen Liste. Die Attribute der `Element`-Objekte werden hierbei nicht verändert.

In jedem Fall soll zurückgegeben werden, um wieviel und in welche Richtung das Attribut `position` der aktuellen `Liste` tatsächlich verschoben wurde. Dabei zeigen Werte > 0 eine Verschiebung nach hinten an und Werte < 0 eine Verschiebung nach vorne. Wie bereits oben erwähnt ist das Ergebnis für die leere Liste immer `0`.

Zum Beispiel würde ein Aufruf von `aenderePosition(1)` für unsere Beispielliste das Ergebnis `1` liefern und als Seiteneffekt die Liste wie folgt verändern:



Würde man stattdessen `aenderePosition(-4)` für die ursprüngliche Beispielliste aufrufen, so erhielte man das Ergebnis `-1`, und als Seiteneffekt würde sich die Liste wie folgt verändern:



Gehen Sie bei Ihrer Implementierung davon aus, dass die Struktur der aktuellen Liste intakt im Sinne von Aufgabenteil (a) ist.

Sie dürfen beliebige Hilfsmethoden einführen (diese Aufgabe lässt sich aber auch gut ohne die Einführung weiterer Hilfsmethoden lösen). Es ist Ihnen diesmal freigestellt, ob Sie Schleifen oder Rekursion verwenden.

```
public int aenderePosition(int i) {
    int res = 0;
    if (this.position == null) {
        return 0;
    }
    else {
        while (i > 0 && this.position.nachfolger != null) {
            res++;
            i--;
            this.position = this.position.nachfolger;
        }
        while (i < 0 && this.position.vorgaenger != null) {
            res--;
            i++;
            this.position = this.position.vorgaenger;
        }
        return res;
    }
}
```

d) Implementieren Sie die Methode

`public boolean hatMehrfaches()`

der Klasse `Liste`.

Diese Methode soll genau dann `true` zurückgeben, wenn es eine Zahl gibt, die mehr als einmal in der aktuellen Liste vorkommt. Bei der Liste aus unserem Beispiel würde die Methode `false` zurückgeben.

Gehen Sie bei Ihrer Implementierung davon aus, dass die Struktur der aktuellen Liste intakt im Sinne von Aufgabenteil (a) ist.

Sie dürfen beliebige Hilfsmethoden einführen. Es empfiehlt sich, die Methode `erstes` aus Aufgabenteil (b) zu verwenden und die Liste vom ersten Element ab zu durchlaufen. Es ist Ihnen wieder freigestellt, ob Sie Schleifen oder Rekursion verwenden.

```
public boolean hatMehrfaches() {
    Element e = this.erstes();
    return hatMehrfachesAbElement(e);
}

private static boolean hatMehrfachesAbElement(Element e) {
    if (e == null) {
        return false;
    }
    if (enthaelt(e.nachfolger, e.wert)) {
        return true;
    }
    return hatMehrfachesAbElement(e.nachfolger);
}

private static boolean enthaelt(Element e, int i) {
    if (e == null) {
        return false;
    }
    if (e.wert == i) {
        return true;
    }
    return enthaelt(e.nachfolger, i);
}
```

Vorname	Name	Matr.-Nr.

Aufgabe 5 (Funktionale Programmierung in Haskell, 4 + 2 + 5 + 6 Punkte**)**

a) Geben Sie den allgemeinsten Typ der Funktionen `f` und `g` an, die wie folgt definiert sind. Gehen Sie davon aus, dass `1` den Typ `Int` hat.

```
f x 1 = [f x x]
```

Lösung: nicht typkorrekt

```
g = \x y -> y 1
```

```
g :: a -> (Int -> b) -> b
```

b) Diese Aufgabe beschäftigt sich mit der Nahrungsaufnahme und -beschaffung von Nagetieren. Die einzige betrachtete Nagetierart ist die der Lemminge. Jeder Lemming hat einen nicht-negativen `Int`-Wert, der seinen Hunger symbolisiert. Frisst ein Lemming Nahrung, vermindert er seinen Hunger. Wir betrachten nur Nüsse als Nahrung. Diese haben einen nicht-negativen `Int`-Wert, der ihren Sättigungswert symbolisiert.

Die folgenden `data`-Deklarationen modellieren diesen Sachverhalt:

```
data Nagetier = Lemming Int

data Nahrung = Nuss Int
```

Gegeben sind die Funktionen `fressen` und `machWas`:

```
machWas :: (a -> b) -> [a] -> [b]
machWas f [x] = []
machWas f (x:xs) = (f x):machWas f xs

fressen :: Nagetier -> Nahrung -> Nagetier
fressen (Lemming i) (Nuss n)
    | n > i     = Lemming 0
    | otherwise = Lemming (i - n)
```

Welchen Wert liefert der folgende Ausdruck zurück?

```
machWas (\x -> fressen x (Nuss 3)) [Lemming 5, Lemming 2, Lemming 2]
```

Lösung: `[Lemming 2, Lemming 0]`

c) Ein Lemming versteckt seine im Sommer gesammelte Nahrung in einem Tunnelsystem. Ein Tunnelsystem besteht aus Kammern mit Nahrung, leeren Kammern und Verzweigungen in weitere Tunnelsysteme nach links und rechts.

```
data Tunnelsystem =
      Kammer Nahrung
    | LeereKammer
    | Verzweigung Tunnelsystem Tunnelsystem
```

Außerdem sei die folgende `data`-Deklaration `Richtung` gegeben:

```
data Richtung = Links | Rechts
```

Definieren Sie die Funktion `verstecken` vom Typ

```
Tunnelsystem -> Nahrung -> [Richtung] -> Tunnelsystem
```

Als Argumente bekommt die Funktion ein `Tunnelsystem`, eine `Nahrung n` und einen Weg vom Typ `[Richtung]` (also eine Liste von `Links`- und `Rechts`-Anweisungen, die ausdrücken, ob man an einer Verzweigung in den **linken** oder **rechten** Teilausdruck absteigt). Die Funktion `verstecken` liefert ein `Tunnelsystem` zurück, bei dem die Nahrung an der Stelle, auf die der Pfad zeigt, in eine Kammer gelegt wird. Gehen Sie hierbei davon aus, dass der Pfad in einem Term `LeereKammer` endet und ersetzen Sie diese durch eine `Kammer n`. An den anderen Stellen ändert sich das Tunnelsystem nicht.

```
verstecken :: Tunnelsystem -> Nahrung -> [Richtung] -> Tunnelsystem
verstecken LeereKammer n [] = Kammer n
verstecken (Verzweigung l r) s (Links:pfad)
    = Verzweigung (verstecken l s pfad) r
verstecken (Verzweigung l r) s (Rechts:pfad)
    = Verzweigung l (verstecken r s pfad)
```

Vorname	Name	Matr.-Nr.

d) Definieren Sie die Funktion `maxKammer` vom Typ `Tunnelsystem -> Tunnelsystem`. Die Funktion gibt die Kammer mit den meisten Nüssen in dem Tunnelsystem zurück. Falls das Tunnelsystem keine Kammer mit Nüssen enthält, wird `LeereKammer` zurückgegeben.

```
maxKammer :: Tunnelsystem -> Tunnelsystem
maxKammer LeereKammer = LeereKammer
maxKammer (Kammer n) = Kammer n
maxKammer (Verzweigung l r) = maxi (maxKammer l) (maxKammer r)
     where maxi (Kammer (Nuss n))  (Kammer (Nuss m))
               | n > m                        = Kammer (Nuss n)
               | otherwise                    = Kammer (Nuss m)
           maxi (Kammer n)  LeereKammer       = Kammer n
           maxi LeereKammer (Kammer m)        = Kammer m
```

Vorname	Name	Matr.-Nr.

Aufgabe 6 (Logische Programmierung in Prolog, (2 + 2) + 5 + (4 + 4) Punkte**)**

a) Geben Sie für die folgenden Paare von Termen den allgemeinsten Unifikator an oder begründen Sie kurz, warum dieser nicht existiert.

- `f(g(A,h(B)),C,D)` und `f(C,g(h(B),D),h(B))`

  ```
  A = h(B)
  C = g(h(B), h(B))
  D = h(B)
  ```

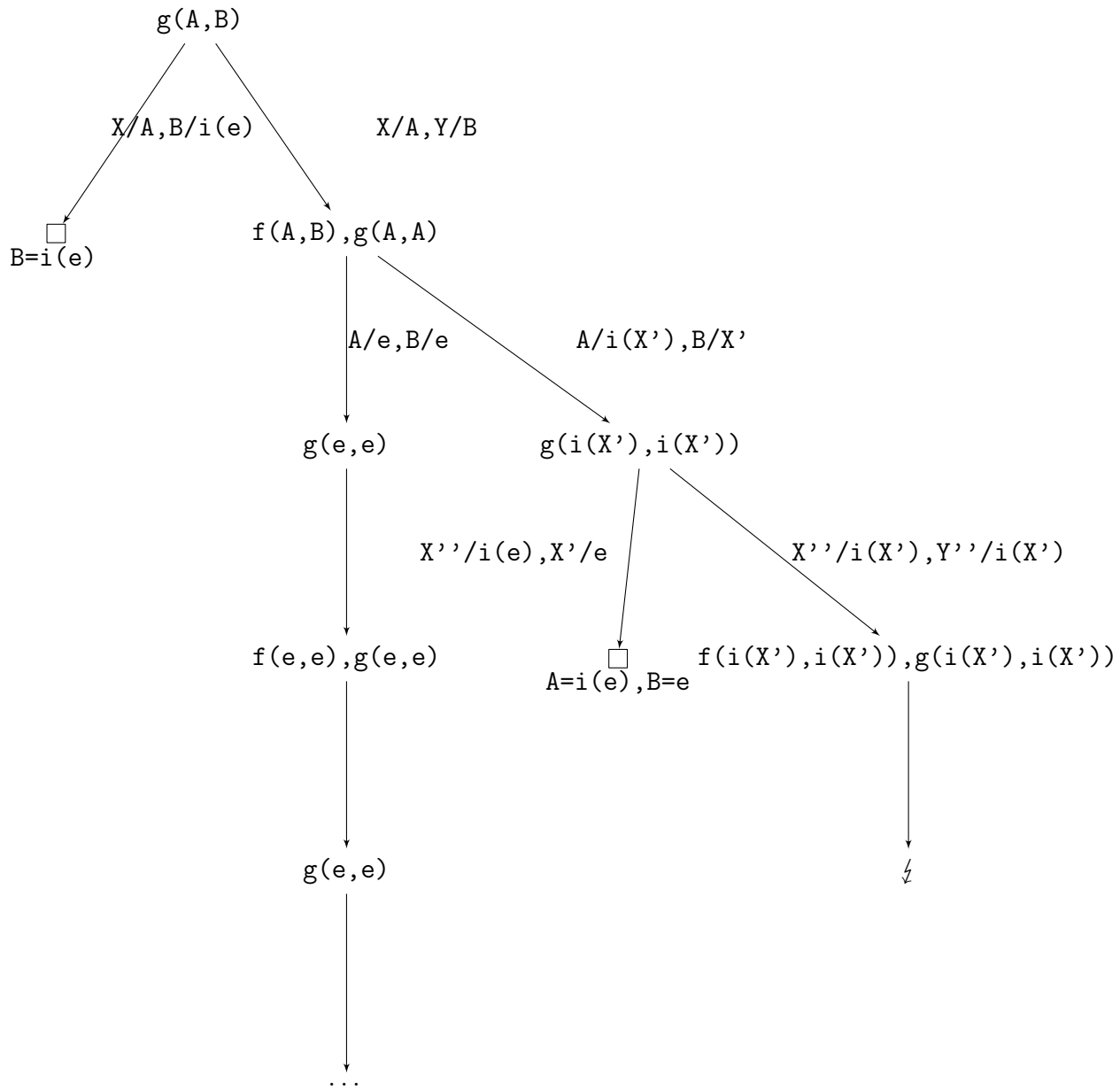
- `f(A,i(C),g(A,B))` und `f(h(C),B,g(D,D))`

  `A = h(C)`
  `B = i(C)`
  `g(h(C),i(C))` $\stackrel{!}{=}$ `g(D,D)` $\Rightarrow$ `h(C)` $\stackrel{!}{=}$ `i(C)` $\Rightarrow$ `clash failure`

Vorname	Name	Matr.-Nr.

b) Erstellen Sie für das folgende Logikprogramm den Beweisbaum zur Anfrage „`?- g(A,B).`“ und geben Sie alle Antwortsubstitutionen an. Sie dürfen Pfade abbrechen, falls diese offensichtlich unendlich sind, weil es auf dem Pfad zwei Knoten gibt, die dieselbe Anfrage enthalten. Kennzeichnen Sie unendliche Pfade durch „...“ und abgebrochene Pfade durch „↯“.

```
f(e,e).
f(i(X),X).
g(X,i(e)).
g(X,Y) :- f(X,Y), g(X,X).
```



Vorname	Name	Matr.-Nr.

20

c)

- Programmieren Sie das einstellige Prädikat `uniListe`, das für eine mindestens zweielementige Liste überprüft, ob alle Elemente der Liste miteinander unifizieren. Hierbei darf wie bei der in Prolog verwendeten Unifikation auf den Occur Check verzichtet werden. Beispielsweise liefert die Anfrage „`?- uniListe([f(X),f(3),Y]).`“ die Antwortsubstitution `X = 3, Y = f(3)` und die Anfrage „`?- uniListe([f(X),g(3),Y]).`“ das Ergebnis `false`.

```
uniListe([X,X]).
uniListe([X,X|XS]) :- uniListe([X|XS]).
```

- Programmieren Sie das zweistellige Prädikat `aListe`. Der erste Parameter ist hierbei eine Zahl `N` und der zweite Parameter ist eine Liste, die `N`-mal das Element `a` enthält. Beispielsweise ergibt die Anfrage „`?- aListe(3, XS).`“ die Antwortsubstitution `XS = [a,a,a]` und die Anfrage „`?- aListe(2, [X, Y]).`“ die Antwortsubstitution `X = a, Y = a`. Gehen Sie hierbei davon aus, dass bei Anfragen der erste Parameter immer eine nicht-negative konkrete Zahl ist.

```
aListe(0,[]).
aListe(X, [a|AS]) :- X > 0, Y is X - 1, aListe(Y, AS).
```