Aufgabe 1 (Programmanalyse): (15 Punkte)

Geben Sie die Ausgabe des folgenden Java-Programms für den Aufruf `java M` an. Tragen Sie hierzu jeweils die ausgegebenen Zeichen in die markierten Stellen hinter „`OUT:`“ ein.

```
public class A {

    public static int x = 5;

    public double y = 3;

    public A() {
        y = x * 2;
    }

    public A(int x) {
        this.y = x;
        this.x += x;
    }

    public void f(double y) {
        this.y = y;
    }
}
```

```
public class B extends A {

    public int x = 1;

    public B(int x) {
        this.y += x;
        this.x += x;
    }

    public void f(Float y) {
        this.y = 2 * y;
    }

    public void f(double y) {
        this.y = 3 * y;
    }
}
```

```
public class M {
    public static void main(String[] args) {
        A a = new A();
        System.out.println(A.x + " " + a.y);    // OUT: [     ]   [     ]

        A a2 = new A(2);
        System.out.println(A.x + " " + a2.y);   // OUT: [     ]   [     ]

        B b = new B(7);
        System.out.println(b.x + " " + b.y);    // OUT: [     ]   [     ]

        System.out.println(A.x);                // OUT: [     ]

        a.f(2.3);
        System.out.println(a.y);                // OUT: [     ]

        b.f(2.4f);
        System.out.println(b.y);                // OUT: [     ]

        A ab = b;
        ab.f(2.5);
        System.out.println(ab.y);               // OUT: [     ]
    }
}
```

Lösung:

```
public class M {
    public static void main(String[] args) {
        A a = new A();
        System.out.println(A.x + " " + a.y);  // OUT: [  5  ]    [ 10.0]

        A a2 = new A(2);
        System.out.println(A.x + " " + a2.y); // OUT: [  7  ]    [ 2.0 ]

        B b = new B(7);
        System.out.println(b.x + " " + b.y);  // OUT: [  8  ]    [ 21.0]

        System.out.println(A.x);              // OUT: [  7  ]

        a.f(2.3);
        System.out.println(a.y);              // OUT: [ 2.3 ]

        b.f(2.4f);
        System.out.println(b.y);              // OUT: [ 7.2 ]

        A ab = b;
        ab.f(2.5);
        System.out.println(ab.y);             // OUT: [ 7.5]
    }
}
```

Aufgabe 2 (Hoare-Kalkül): (11 Punkte)

Gegeben sei folgendes Java-Programm P.

```
⟨ true ⟩                          (Vorbedingung)
i = 0;
x = a;
y = a;
while (x <= 0) {
    i = i + 1;
    y = y + 1;
    x = x + y;
}
⟨ y > 0 ⟩                         (Nachbedingung)
```

Hinweise:

- Die Schleifeninvariante hat die Form $\langle \mathtt{x} = \sum_{j=0}^{\mathtt{i}} \ldots \wedge \mathtt{y} = \mathtt{a} + \mathtt{i} \rangle$
- Sie können ausnutzen, dass für jede Funktion f folgendes gilt: Aus $\sum_{j=0}^{i} f(j) > 0$ und $f(j+1) > f(j)$ für alle Zahlen j folgt $f(i) > 0$.
- Sie dürfen beliebig viele Zusicherungs-Zeilen ergänzen oder streichen. In der Musterlösung werden allerdings genau die angegebenen Zusicherungen benutzt.
- Bedenken Sie, dass die Regeln des Kalküls syntaktisch sind, weshalb Sie semantische Änderungen (beispielsweise von $x+1 = y+1$ zu $x = y$) nur unter Zuhilfenahme der Konsequenzregeln vornehmen dürfen. Benötigte Klammern dürfen Sie auch ohne Konsequenzregel ergänzen.

Vervollständigen Sie die Verifikation des Algorithmus P auf der folgenden Seite 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.

Lösung:

$\langle \mathsf{true} \rangle$
$\langle 0 = 0 \wedge \mathtt{a} = \mathtt{a} \wedge \mathtt{a} = \mathtt{a} \rangle$

```
i = 0;
```

$\langle \mathtt{i} = 0 \wedge \mathtt{a} = \mathtt{a} \wedge \mathtt{a} = \mathtt{a} \rangle$

```
x = a;
```

$\langle \mathtt{i} = 0 \wedge \mathtt{x} = \mathtt{a} \wedge \mathtt{a} = \mathtt{a} \rangle$

```
y = a;
```

$\langle \mathtt{i} = 0 \wedge \mathtt{x} = \mathtt{a} \wedge \mathtt{y} = \mathtt{a} \rangle$
$\langle \mathtt{x} = \sum_{j=0}^{\mathtt{i}} (\mathtt{a} + j) \wedge \mathtt{y} = \mathtt{a} + \mathtt{i} \rangle$

```
while (x <= 0) {
```

$\langle \mathtt{x} = \sum_{j=0}^{\mathtt{i}} (\mathtt{a} + j) \wedge \mathtt{y} = \mathtt{a} + \mathtt{i} \wedge \mathtt{x} \leq 0 \rangle$
$\langle \mathtt{x} + \mathtt{y} + 1 = \sum_{j=0}^{\mathtt{i}+1} (\mathtt{a} + j) \wedge \mathtt{y} + 1 = \mathtt{a} + \mathtt{i} + 1 \rangle$

```
    i = i + 1;
```

$\langle \mathtt{x} + \mathtt{y} + 1 = \sum_{j=0}^{\mathtt{i}} (\mathtt{a} + j) \wedge \mathtt{y} + 1 = \mathtt{a} + \mathtt{i} \rangle$

```
    y = y + 1;
```

$\langle \mathtt{x} + \mathtt{y} = \sum_{j=0}^{\mathtt{i}} (\mathtt{a} + j) \wedge \mathtt{y} = \mathtt{a} + \mathtt{i} \rangle$

```
    x = x + y;
```

$\langle \mathtt{x} = \sum_{j=0}^{\mathtt{i}} (\mathtt{a} + j) \wedge \mathtt{y} = \mathtt{a} + \mathtt{i} \rangle$

```
}
```

$\langle \mathtt{x} = \sum_{j=0}^{\mathtt{i}} (\mathtt{a} + j) \land \mathtt{y} = \mathtt{a} + \mathtt{i} \land \mathtt{x} > 0 \rangle$
$\langle \mathtt{y} > 0 \rangle$

- erste Implikation:

$$
\begin{array}{lll}
 & \mathtt{i} = 0 \land \mathtt{x} = \mathtt{a} \land \mathtt{y} = \mathtt{a} & \\
\iff & \mathtt{i} = 0 \land \mathtt{x} = \sum_{j=0}^{i} (\mathtt{a} + j) \land \mathtt{y} = \mathtt{a} & \text{denn } \sum_{j=0}^{0} (\mathtt{a} + j) = \mathtt{a} \\
\iff & \mathtt{i} = 0 \land \mathtt{x} = \sum_{j=0}^{i} (\mathtt{a} + j) \land \mathtt{y} = \mathtt{a} + \mathtt{i} & \text{denn } \mathtt{a} + 0 = \mathtt{a} \\
\implies & \mathtt{x} = \sum_{j=0}^{i} (\mathtt{a} + j) \land \mathtt{y} = \mathtt{a} + \mathtt{i} &
\end{array}
$$

- Kurz: $\sum_{j=0}^{0} (\mathtt{a} + j) = \mathtt{a}$
- zweite Implikation:

$$
\begin{array}{lll}
 & \mathtt{x} + \mathtt{y} + 1 = \sum_{j=0}^{\mathtt{i}+1} (\mathtt{a} + j) \land \mathtt{y} + 1 = \mathtt{a} + \mathtt{i} + 1 & \\
\iff & \mathtt{x} + \mathtt{y} + 1 = \sum_{j=0}^{\mathtt{i}} (\mathtt{a} + j) + \mathtt{a} + \mathtt{i} + 1 \land \mathtt{y} + 1 = \mathtt{a} + \mathtt{i} + 1 & \\
\iff & \mathtt{x} + \mathtt{a} + \mathtt{i} + 1 = \sum_{j=0}^{\mathtt{i}} (\mathtt{a} + j) + \mathtt{a} + \mathtt{i} + 1 \land \mathtt{y} + 1 = \mathtt{a} + \mathtt{i} + 1 & \text{denn } \mathtt{y} + 1 = \mathtt{a} + \mathtt{i} + 1 \\
\iff & \mathtt{x} = \sum_{j=0}^{\mathtt{i}} (\mathtt{a} + j) \land \mathtt{y} + 1 = \mathtt{a} + \mathtt{i} + 1 & \\
\iff & \mathtt{x} = \sum_{j=0}^{\mathtt{i}} (\mathtt{a} + j) \land \mathtt{y} = \mathtt{a} + \mathtt{i} & \\
\impliedby & \mathtt{x} = \sum_{j=0}^{\mathtt{i}} (\mathtt{a} + j) \land \mathtt{y} = \mathtt{a} + \mathtt{i} \land \mathtt{x} \leq 0 &
\end{array}
$$

- Kurz: Letzter Summand ist $a + \mathtt{i} + 1 = \mathtt{y} + 1$
- dritte Implikation:

$$
\begin{array}{lll}
 & \mathtt{x} = \sum_{j=0}^{\mathtt{i}} (\mathtt{a} + j) \land \mathtt{y} = \mathtt{a} + \mathtt{i} \land x > 0 & \\
\implies & \mathtt{a} + \mathtt{i} > 0 \land \mathtt{y} = \mathtt{a} + \mathtt{i} \land x > 0 & \text{siehe Hinweise} \\
\iff & \mathtt{y} > 0 \land \mathtt{y} = \mathtt{a} + \mathtt{i} \land x > 0 & \text{denn } \mathtt{y} = \mathtt{a} + \mathtt{i} \\
\implies & \mathtt{y} > 0 &
\end{array}
$$

- Kurz: Letzter Summand $\mathtt{a} + \mathtt{i} = \mathtt{y}$ ist positiv (siehe Hinweise)

Alternative Lösung:

$\langle \mathsf{true} \rangle$
$\langle a = a \rangle$

```
i = 0;
```

$\langle a = a \rangle$

```
x = a;
```

$\langle a = \mathtt{x} \rangle$

```
y = a;
```

$\langle \mathtt{y} = \mathtt{x} \rangle$
$\langle \mathtt{y} \geq \mathtt{x} \rangle$

```
while (x <= 0) {
```

$\langle \mathtt{y} \geq \mathtt{x} \land \mathtt{x} \leq 0 \rangle$
$\langle \mathtt{y} + 1 \geq \mathtt{x} + \mathtt{y} + 1 \rangle$

```
    i = i + 1;
```

$\langle \mathtt{y} + 1 \geq \mathtt{x} + \mathtt{y} + 1 \rangle$

```
    y = y + 1;
```

$\langle \mathtt{y} \geq \mathtt{x} + \mathtt{y} \rangle$

```
    x = x + y;
```

$\langle \mathtt{y} \geq \mathtt{x} \rangle$

```
}
```

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

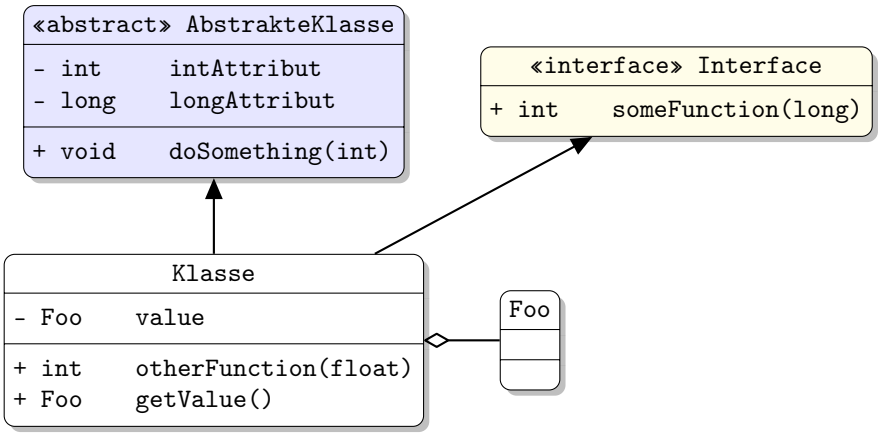
Aufgabe 3 (Klassen-Hierarchie): (9 + 9 = 18 Punkte)

In dieser Aufgabe soll ein Teil des Fahrradmarktes modelliert werden.

- Ein wesentliches Kennzeichen eines Fahrrads ist sein Gewicht und sein Besitzer, der ein Mensch ist.
- Ein Fahrrad ist entweder ein Mountainbike, ein Citybike, ein Rennrad oder ein Crossrad. Andere Arten von Fahrrädern gibt es nicht.
- Mountainbikes haben ungewöhnlich dicke Reifen. Deshalb ist die Reifendicke in Millimetern ein wichtiges Merkmal eines Mountainbikes.
- Wichtig für Citybikes ist die Größe des Einkaufskörbchens.
- Zeitfahrräder sind spezielle Rennräder, die eine besonders aerodynamische Sitzposition ermöglichen. Deswegen ist ihr Luftwiderstand von Interesse.
- Ein Fahrradliebhaber ist ein Mensch, der eine größere Sammlung von Fahrrädern besitzt.
- Crossräder und Mountainbikes haben gemeinsam, dass sie geländetauglich sind. Sie ermöglichen ihrem Besitzer daher, durch unwegsames Gelände zu fahren. Beachten Sie, dass es auch geländetaugliche Fahrzeuge geben könnte, die keine Fahrräder sind.
- E-Bikes sind bestimmte Citybikes, die die Möglichkeit bieten, einen Motor anzuschalten.

a) Entwerfen Sie unter Berücksichtigung der Prinzipien der Datenkapselung eine geeignete Klassenhierarchie für die oben aufgelisteten Sachverhalte. Notieren Sie keine Konstruktoren oder Selektoren. Sie müssen nicht markieren, ob Attribute `final` sein sollen. Achten Sie darauf, dass gemeinsame Merkmale in Oberklassen bzw. Interfaces zusammengefasst werden und markieren Sie alle Klassen als abstrakt, bei denen dies sinnvoll ist.

Verwenden Sie hierbei die folgende Notation:



Eine Klasse wird hier durch einen Kasten dargestellt, in dem der Name der Klasse sowie alle in der Klasse definierten bzw. überschriebenen Attribute und Methoden in einzelnen Abschnitten beschrieben werden. Weiterhin bedeutet der Pfeil $B \blacktriangleright A$, dass A die Oberklasse von B ist (also `class B extends A` bzw. `class B implements A`, falls A ein Interface ist). Der Pfeil $B \diamond\!\!- A$ bedeutet, dass A ein Objekt vom Typ B benutzt. Benutzen Sie `-`, um `private` abzukürzen, und `+` für alle anderen Sichtbarkeiten (wie z. B. `public`). Fügen Sie Ihrem Diagramm keine Kästen für vordefinierte Klassen wie `String` hinzu.

b) Schreiben Sie eine Java-Methode mit der folgenden Signatur:

```
public static int fahrradTour(Mensch[] teilnehmer)
```

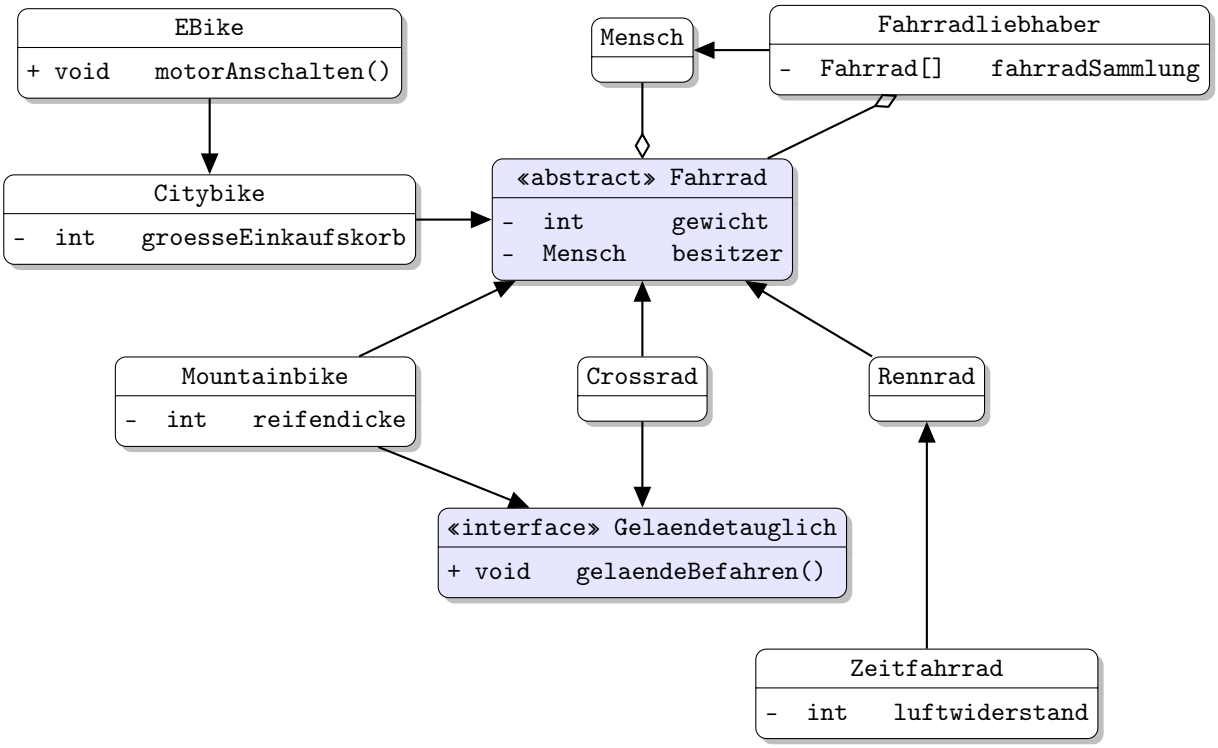
Diese Methode soll wie folgt vorgehen: Jeder Fahrradliebhaber im Array `teilnehmer` stellt all seine E-Bikes für Teilnehmer zur Verfügung, die keine Fahrradliebhaber sind. Zu diesem Zweck schaltet er die Motoren all seiner E-Bikes an. Anschließend wählt er ein beliebiges geländetaugliches Fahrrad in seiner Sammlung und fährt damit ins Gelände. Sie können dabei davon ausgehen, dass jeder Fahrradliebhaber mindestens ein geländetaugliches Fahrrad besitzt. Der Rückgabewert der Methode gibt an, wie viele (potentielle) Teilnehmer nicht an der Fahrradtour teilnehmen können, weil nicht genügend E-Bikes zur Verfügung stehen. Wenn `teilnehmer` also n Menschen enthält, die keine Fahrradliebhaber sind und alle Fahrradliebhaber in `teilnehmer` zusammen m E-Bikes besitzen, dann gibt die Methode 0 zurück, wenn $m \geq n$ gilt. Ansonsten gibt die Methode $n - m$ zurück.

Hinweise:

- Gehen Sie hierbei davon aus, dass es für alle Attribute geeignete Selektoren (get- und set-Methoden) gibt.

Lösung:

a) Die Zusammenhänge können wie folgt modelliert werden:

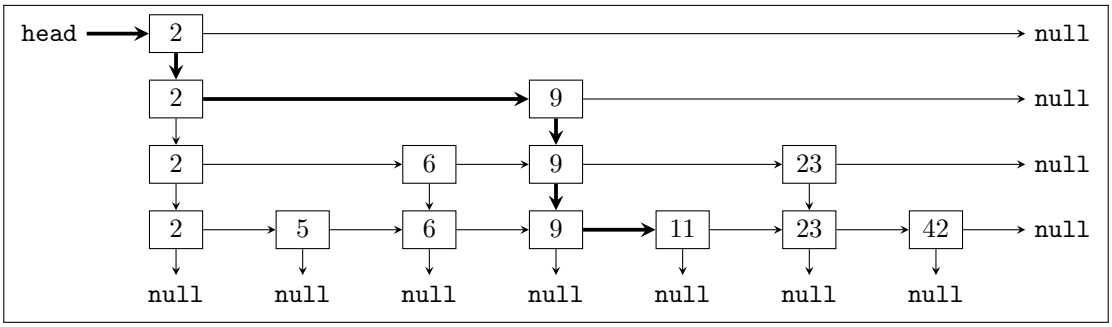


b)
```
public static int fahrradTour(Mensch[] teilnehmer) {
  int n = 0;
  int m = 0;
  for (Mensch t: teilnehmer) {
    if (t instanceof Fahrradliebhaber) {
      Gelaendetauglich gt = null;
      for (Fahrrad f: ((Fahrradliebhaber) t).getFahrradSammlung()) {
        if (f instanceof EBike) {
          ((EBike) f).motorAnschalten();
          m++;
        } else if (f instanceof Gelaendetauglich) {
```

```
            gt = (Gelaendetauglich) f;
          }
        }
        gt.gelaendeBefahren();
      } else {
        n++;
      }
    }
    if (m >= n) {
      return 0;
    } else {
      return n-m;
    }
  }
```

Aufgabe 4 (Programmieren in Java): (9 + 14 + 9 + 4 = 36 Punkte)

In dieser Aufgabe beschäftigen wir uns mit Skip-Listen. Eine Skip-Liste besteht aus mehreren sortierten Listen, die in mehreren Ebenen angeordnet sind. Die oberen Ebenen enthalten dabei immer eine Teilmenge der Elemente der Ebene darunter. Neben dem Verweis auf das nächste Element der Liste hat jedes Element noch einen Verweis auf das gleiche Element in der Ebene darunter.



In dieser Datenstruktur kann man, ähnlich wie in einem binären Suchbaum, effizient suchen. Wenn man nach einem Wert n sucht, geht man zunächst solange auf der gleichen Ebene weiter, bis der nächste Wert größer als n wäre. Dann setzt man die Suche auf der Ebene darunter vom letzten Element kleiner n fort. Man überspringt also Elemente auf den unteren Ebenen.

Beispiel:
Das obige Beispiel ist eine Skip-Liste mit 4 Ebenen. Sucht man im obigen Beispiel nach der Zahl 11, beginnt man oben links bei dem Element, das von `head` referenziert wird. Von diesem ersten Element prüft man, ob das nächste Element existiert. Da die oberste Liste nur die 2 enthält, geht man sofort auf die Ebene darunter, zum ersten Element auf der zweiten Ebene. Von dort prüft man wieder den Nachfolger. Da $9 < 11$ gilt, geht man auf der zweiten Ebene einen Schritt weiter. Hier ist die zweite Liste ebenfalls zu Ende, also geht man beim aktuellen Element 9 auf die dritte Ebene. Hier kann man keinen Schritt weiter laufen, da $23 > 11$ gilt. Also geht man beim aktuellen Element 9 eine Ebene weiter nach unten. Dort vergleicht man wieder mit dem Nachfolger und findet schließlich das gesuchte Element 11. Der Pfad ist in der Abbildung oben mit dickeren Pfeilen eingezeichnet.
Betrachten wir als zweites Beispiel die Suche nach der Zahl 8. Auch hier beginnen wir oben links. Das nächste Element ist `null`, also gehen wir im aktuellen Element 2 eine Ebene nach unten. Auf der zweiten Ebene ist das nächste Element die 9. Da $9 > 8$ ist, gehen wir nicht zum Nachfolger, sondern im aktuellen Element 2 wieder eine Ebene nach unten. Hier gehen wir dann einen Schritt weiter, da $6 < 8$ gilt. Dann müssen wir aber wieder eine Ebene nach unten, da $9 > 8$ ist. Auf der vierten und letzten Ebene können wir ebenfalls nicht weiter gehen, wieder wegen $9 > 8$. Da keine weitere Ebene darunter existiert, können wir schließen, dass das gesuchte Element 8 nicht in der Liste enthalten ist.

In dieser Aufgabe gehen wir davon aus, dass die Datenstruktur mit den folgenden Klassen realisiert ist:

```
public class SkipList {
  SkipNode head;
  int height; //Anzahl der Ebenen der Skip-Liste
}

public class SkipNode {
  int value;
  SkipNode nextElement; //naechstes Element auf der gleichen Ebene
  SkipNode nextLayer; //gleiches Element auf der Ebene darunter
}
```

Hinweise:

Sie dürfen in allen Teilaufgaben zusätzliche Hilfsmethoden schreiben. Geben Sie bei diesen Hilfsmethoden jeweils an, in welcher Klasse diese implementiert sind.

a) Implementieren Sie eine öffentliche Methode `boolean contains(int n)` in der Klasse `SkipList`. Diese soll `true` zurückgeben, falls die Zahl `n` in der Skip-Liste enthalten ist und ansonsten `false`. Die Methode

soll dabei wie in den obigen Beispielen beschrieben vorgehen, d.h. sie soll so viele Elemente auf den unteren Ebenen der Liste wie möglich überspringen.

Verwenden Sie in dieser Aufgabe **nur Rekursion** und **keine Schleifen**.

b) Schreiben Sie eine öffentliche Methode `int length()` in der Klasse `SkipList`. Diese soll die Anzahl der verschiedenen, in der Liste gespeicherten Zahlen zurückgeben (für die Beispielliste ergibt sich also 7). Außerdem soll sie prüfen, ob die Skip-Liste die folgenden Kriterien tatsächlich erfüllt:

- Jede der Ebenen ist eine sortierte Liste ohne Duplikate.
- Der `nextLayer`-Pointer zeigt genau dann auf `null`, wenn es ein `SkipNode` auf der untersten Ebene ist.

Falls eine dieser Bedingungen nicht erfüllt ist, soll eine `InvalidSkipListException` geworfen werden und ein sinnvoller String als `reason` übergeben werden. Diese Exception ist wie folgt definiert:

```
public class InvalidSkipListException extends Exception {
  private String reason;
  public InvalidSkipListException(String r) {
    this.reason = r;
  }
  public String toString() {
    return "Invalid Skip List, Reason: " + reason;
  }
}
```

Hinweise:

- Gehen Sie davon aus, dass das Attribut `height` angibt, wie viele Ebenen die Liste hat.
- Die Anzahl verschiedener Zahlen in einer korrekten Skip-Liste entspricht der Länge der Liste in der untersten Ebene.

c) Schreiben Sie eine öffentliche Methode `ArrayList<Integer> toArrayList()` in der Klasse `SkipList`. Diese soll die Elemente der Skip-Liste in eine `ArrayList` aus dem Java Collections Framework einfügen. Nutzen Sie die Methode `length` aus dem vorherigen Aufgabenteil, um die initiale Kapazität der `ArrayList` passend zu setzen. Fangen Sie die eventuell auftretende `InvalidSkipListException` und geben Sie in dem Fall eine Fehlermeldung aus. Bei einem Fehler soll die Methode eine leere `ArrayList` zurück geben.

Verwenden Sie in dieser Aufgabe **nur Schleifen** und **keine Rekursion**.

Hinweise:

- Die Klasse `ArrayList` hat neben dem Konstruktor `ArrayList()`, der eine leere `ArrayList` erzeugt, noch einen Konstruktor `ArrayList(int initialCapacity)`, der die initiale Kapazität der neuen `ArrayList` auf den übergebenen Wert setzt.
- Die Klasse `ArrayList` hat eine nicht-statische Methode `add(e)`, die ein Element `e` am Ende der aktuellen `ArrayList` einfügt.
- Sämtliche in einer Skip-Liste enthaltenen Werte finden sich in der Liste auf der untersten Ebene.

d) Passen Sie die Deklarationen der Klassen `SkipList` und `SkipNode` so an, dass anstelle von `int`-Werten beliebige Objekte in der Liste gespeichert werden können, die das Interface `Vergleichbar` implementieren.

Sie brauchen **keine Methoden** anpassen, **nur** die **Klassen und Attribute**. Auch das Interface `Vergleichbar` brauchen Sie nicht schreiben.

Lösung:

SkipList

a)
```
public boolean contains(int n) {
  if(head == null) {
    return false;
  } else {
    return head.contains(n);
  }
}
```

SkipNode

```
boolean contains(int n) {
  if(this.value == n) {
    return true;
  } else if(this.nextElement != null && this.nextElement.value <= n) {
    return this.nextElement.contains(n);
  } else if(this.nextLayer != null) {
    return this.nextLayer.contains(n);
  } else {
    return false;
  }
}
```

SkipList

b)
```
public int length() throws InvalidSkipListException {
  SkipNode currentHead = this.head;
  int length = 0;
  for(int currentLayer = this.height; currentLayer > 0; --currentLayer) {
    SkipNode current = currentHead;
    while(current != null) { //Schleife ueber eine Ebene
      if (current.nextElement != null
          && current.nextElement.value <= current.value) {
        throw new InvalidSkipListException("Nicht Sortiert");
      }
      if(current.nextLayer == null) {
        if (currentLayer > 1) {
          throw new InvalidSkipListException("Fehlender Layer-Pointer");
        } else {
          length += 1; //nur Elemente auf der untersten Ebene zaehlen
        }
      } else if(currentLayer == 1) {
        throw new InvalidSkipListException("Layer unter der untersten Ebene");
      }
      current = current.nextElement;
    }
    currentHead = currentHead.nextLayer;
  }
  return length;
}
```

SkipList

c)
```
public ArrayList<Integer> toArrayList() {
  int c;
  try {
    c = this.length();
```

```
        } catch (InvalidSkipListException e) {
          System.out.println(e.toString());
          return new ArrayList<>();
        }
        ArrayList<Integer> v = new ArrayList<>(c);

        if(this.head != null) {
          SkipNode current = this.head;
          while(current.nextLayer != null) {
            current = current.nextLayer;
          }
          while(current != null) {
            v.add(current.value);
            current = current.nextElement;
          }
        }
        return v;
      }
```

d)

```
public class SkipList<T extends Vergleichbar> {
  SkipNode<T> head;
  int height;
}

public class SkipNode<T extends Vergleichbar> {
  T value;
  SkipNode<T> nextElement;
  SkipNode<T> nextLayer;
}
```

Aufgabe 5 (Haskell): (4 + 4 + 2 + 3 + 7 = 20 Punkte)

a) Geben Sie zu den folgenden Haskell-Funktionen `f` und `g` jeweils den allgemeinsten Typ an. Gehen Sie hierbei davon aus, dass alle Zahlen den Typ `Int` haben.

```
f 0 y z = []
f x y z = x : f y (x-1) z

g x y z = if x then g False y z else y == (x : z)
```

b) Bestimmen Sie, zu welchem Ergebnis die Ausdrücke `i` und `j` jeweils auswerten.

```
i :: [Int]
i = filter (\x -> x > 1) ((\y -> y ++ y) [1,2,3])

j :: [Int]
j = ((\f x -> map x) (\y -> 3+y) (\z -> 2*z)) [1,2,3]
```

c) Geben Sie die Definition einer Datenstruktur `FList a` an. Diese soll Listen von Funktionen repräsentieren, die Argumente des Typs `a` auf Resultate des Typs `a` abbilden.

d) Implementieren Sie die Funktion `apply :: FList a -> a -> a` in Haskell, sodass `apply fs x` die Funktionen aus `fs` nacheinander auf `x` anwendet. Hierbei soll zuerst die erste Funktion der Liste auf `x` und anschließend immer die nachfolgende Funktion auf das neu berechnete Ergebnis angewendet werden. Wenn `fs` beispielsweise eine Liste vom Typ `FList Int` ist, die die Liste `[(\x -> x+1),(\x -> x*x)]` repräsentiert, soll der Ausdruck `apply fs 2` zu 9 ausgewertet werden.

e) Implementieren Sie die Funktion `sublists :: [a] -> [[a]]` in Haskell, sodass `sublists xs` alle Teillisten $[f_1, \ldots, f_k]$ von `xs` berechnet. Als Teilliste bezeichnen wir eine Liste, die aus `xs` entsteht, wenn beliebig viele Elemente weggelassen werden. Die Reihenfolge der übrigen Elemente bleibt dabei erhalten. Beispielsweise soll der Ausdruck `sublists xs [4,0,2]` zu einer Liste mit den Elementen `[4,0,2]`, `[4,0]`, `[4,2]`, `[4]`, `[0,2]`, `[0]`, `[2]` und `[]` ausgewertet werden.

Verwenden Sie in dieser Aufgabe **keine vordefinierten Funktionen** außer `map`, `++` sowie Datenkonstruktoren (`[]`, `:`).

Lösung:

a) i) `f :: Int -> Int -> a -> [Int]`

ii) `g :: Bool -> [Bool] -> [Bool] -> Bool`

b)
```
i = [2,3,2,3]
j = [2,4,6]
```

c)
```
data FList a = Nil | Cons (a -> a) (FList a)
```

d)
```
apply :: FList a -> a -> a
apply Nil         x = x
apply (Cons f fs) x = apply fs (f x)
```

e)
```
sublists :: [a] -> [[a]]
sublists []     = [[]]
sublists (x:xs) = (map (x:) xs') ++ xs'
                  where xs' = sublists xs
```

Aufgabe 6 (Prolog): (2 + 6 + 3 + 4 + 5 = 20 Punkte)

a) Geben Sie zu den folgenden Termpaaren jeweils einen allgemeinsten Unifikator an oder begründen Sie, warum sie nicht unifizierbar sind. Hierbei werden Variablen durch Großbuchstaben dargestellt und Funktionssymbole durch Kleinbuchstaben.

i) $f(s(X), Y, s(a)), f(Y, X, Z)$

ii) $g(s(X), Y, X), g(Z, Y, s(Y))$

b) Gegeben sei folgendes Prolog-Programm P.

```
p(s(X),s(Y),Z) :- p(Y,s(s(Z)),X), p(0,Y,Z).
p(X,s(X),X).
```

Erstellen Sie für das Programm P den Beweisbaum zur Anfrage "`?- p(R,s(s(0)),0).`" bis zur Höhe 4 (die Wurzel hat dabei die Höhe 1). Markieren Sie Pfade, die zu einer unendlichen Auswertung führen, mit ∞ und geben Sie alle Antwortsubstitutionen zur Anfrage "`?- p(R,s(s(0)),0).`" an, die im Beweisbaum bis zur Höhe 4 enthalten sind. Geben Sie außerdem zu jeder dieser Antwortsubstitutionen an, ob sie von Prolog gefunden wird. Geben Sie schließlich noch ein logisch äquivalentes Programm an, das durch eine einzige Vertauschung entsteht und bei dem Prolog jede Antwortsubstitution zur obigen Anfrage findet.

c) Implementieren Sie ein Prädikat `deleteLast` mit Stelligkeit 2 in Prolog, wobei `deleteLast(`t_1`,`t_2`)` genau dann gilt, wenn t_1 eine nicht-leere Liste ist und t_2 die Liste, die sich daraus ergibt, wenn man das letzte Element von t_1 löscht. Alle Anfragen der Form `deleteLast(`t_1`,`t_2`)`, bei denen t_1 keine Variablen enthält, sollen terminieren.

d) Implementieren Sie ein Prädikat `middle` mit Stelligkeit 2 in Prolog, wobei `middle(`t`,`m`)` genau dann für eine nicht leere Liste $t = [e_1, \ldots, e_n]$ gilt, wenn m das mittlere Element von t ist, d. h. $m = e_{n/2}$, falls n gerade ist, und $m = t_{(n+1)/2}$, falls n ungerade ist. Beispielsweise sollen `middle([1,2,3],M)` und `middle([1,2,3,4],M)` jeweils die einzige Antwort `M = 2` liefern. Alle Anfragen der Form `middle(`t`,`m`)`, bei denen t keine Variablen enthält, sollen terminieren.

Hinweise:

- Sie dürfen das Prädikat `deleteLast` aus Teilaufgabe **c)** verwenden, auch wenn Sie dieses Prädikat nicht implementiert haben.

e) Binäre Bäume lassen sich in Prolog mit Hilfe der Funktionssymbole `leaf` und `node` als Terme darstellen. Dabei repräsentiert `leaf` ein einzelnes Blatt und `node(L,R)` repräsentiert einen Baum mit einem Wurzelknoten, der den Teilbaum `L` als linkes Kind und den Teilbaum `R` als rechtes Kind hat. Implementieren Sie ein Prädikat `tree` mit Stelligkeit 2 in Prolog, sodass die Anfrage `tree(T,`h`)` für eine gegebene natürliche Zahl h alle mit Hilfe der Funktionssymbole `leaf` und `node` dargestellten binären Bäume aufzählt, die höchstens Höhe h haben. Hierbei hat ein einzelnes Blatt Höhe 1. Beispielsweise soll die Anfrage `tree(T,3)` folgende Antworten liefern.

```
T = leaf ;
T = node(leaf, leaf) ;
T = node(leaf, node(leaf, leaf)) ;
T = node(node(leaf, leaf), leaf) ;
T = node(node(leaf, leaf), node(leaf, leaf)) ;
false.
```

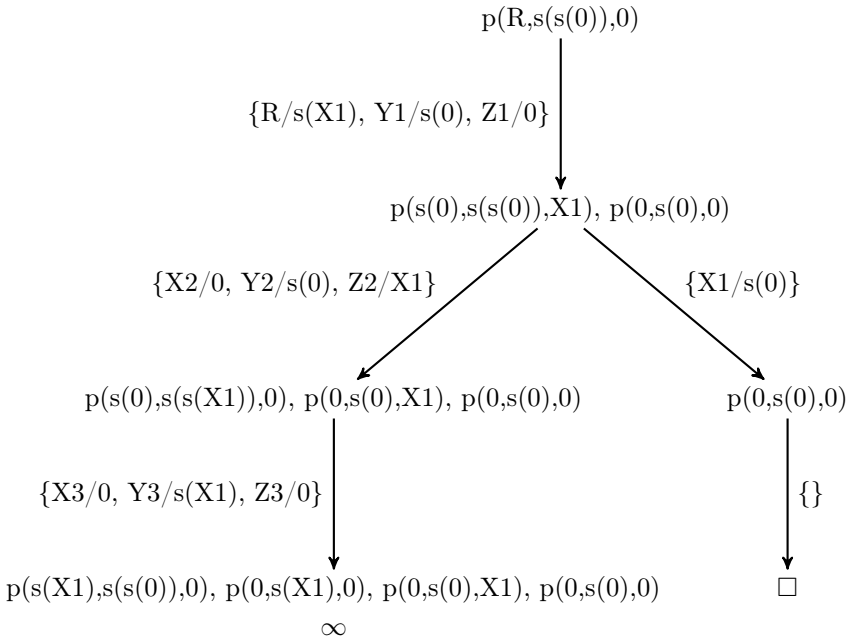
Alle Anfragen der Form `tree(`t`,`h`)`, bei denen h keine Variablen enthält, sollen terminieren.

Lösung:

a) i) $f(s(X), Y, s(a)), f(Y, X, Z)$: occur failure X in $s(X)$

ii) $g(s(X), Y, X), g(Z, Y, s(Y))$: $Z/s(s(Y)), X/s(Y)$

b)



Die (einzige) Antwortsubstitution innerhalb des Beweisbaums ist {R/s(s(0))}. Diese wird von Prolog nicht gefunden.

Mit der folgenden Reihenfolge der Klauseln findet Prolog alle Antwortsubstitutionen:

```
p(X,s(X),X).
p(s(X),s(Y),Z) :- p(Y,s(s(Z)),X), p(0,Y,Z).
```

c)
```
deleteLast([_],[]).
deleteLast([X|XS],[X|YS]) :- deleteLast(XS,YS).
```

d)
```
middle([M],M).
middle([M,_],M).
middle([_|XS],M) :- deleteLast(XS,YS), middle(YS,M).
```

Alternative mit append:

```
middle(XS,M) :- append(YS,[M|ZS],XS), length(YS,L), length(ZS,L).
middle(XS,M) :- append(YS,[M,_|ZS],XS), length(YS,L), length(ZS,L).
```

e)
```
tree(leaf,H) :- H > 0.
tree(node(L,R),H) :- H > 0, HDec is H-1, tree(L,HDec), tree(R,HDec).
```