Prof. Dr. J. Giesl
C. Aschermann, F. Frohn, J. Hensel, T. Ströder

2. Klausur Programmierung WS 2014/2015

Vorname: _____

Nachname: _____

Matrikelnummer: _____

Studiengang (bitte **genau** einen markieren):

- ○ Informatik Bachelor
- ○ Informatik Lehramt (Bachelor)
- ○ Mathematik Bachelor
- ○ Sonstiges: _____

	Anzahl Punkte	Erreichte Punkte
Aufgabe 1	**14**	
Aufgabe 2	**14**	
Aufgabe 3	**16**	
Aufgabe 4	**36**	
Aufgabe 5	**20**	
Aufgabe 6	**20**	
Summe	**120**	

Allgemeine Hinweise:

- **Auf alle Blätter** (inklusive zusätzliche Blätter) müssen Sie **Ihren Vornamen, Ihren Nachnamen und Ihre Matrikelnummer** schreiben.
- Geben Sie Ihre Antworten in lesbarer und verständlicher Form an.
- Schreiben Sie mit **dokumentenechten** Stiften, nicht mit roten oder grünen Stiften und nicht mit Bleistiften.
- Bitte beantworten Sie die Aufgaben auf den Aufgabenblättern (benutzen Sie auch die Rückseiten).
- Geben Sie für jede Aufgabe **maximal eine** Lösung an. Streichen Sie alles andere durch. Andernfalls werden alle Lösungen der Aufgabe mit **0 Punkten** bewertet.
- Werden **Täuschungsversuche** beobachtet, so wird die Klausur mit **0 Punkten** bewertet.
- Geben Sie am Ende der Klausur **alle Blätter zusammen mit den Aufgabenblättern ab**.

Name: **Matrikelnummer:**

Aufgabe 1 (Programmanalyse): (14 Punkte)

Geben Sie die Ausgabe des 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 = 6;
  public static int y = 1;

  public A(){
    x += 1;
  }

  public void f(double x){
    x += 1;
    y = A.y + 1;
  }

  public void f(int x){
    A.y = x;
  }
}
```

```
public class B extends A {
  public int x = 1;

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

  public void f(double x){
    this.x = (int)x;
    A.x = super.x + 3;
  }
}
```

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


    a.f(3);
    System.out.println(A.y);                      // OUT: [     ]


    B b = new B(3);
    System.out.println(((A) b).x + " " + b.x);    // OUT: [     ]   [     ]

    System.out.println(B.y);                      // OUT: [     ]


    b.f(2.);
    System.out.println(b.x + " " + A.x);          // OUT: [     ]   [     ]


    A ab = b;
    ab.f(1.);
    System.out.println(ab.x + " " + ((B)ab).x); // OUT: [     ]   [     ]


    ab.f(5);
    System.out.println(A.x + " " + A.y);          // OUT: [     ]   [     ]
  }
}
```

Aufgabe 2 (Hoare-Kalkül): (10 + 4 = 14 Punkte)

Gegeben sei folgendes Java-Programm P, das zu zwei ganzen Zahlen `n` und $\texttt{k} \geq 0$ den Wert $\binom{\texttt{n}}{\texttt{k}} = \Pi_{\texttt{j}=1}^{\texttt{k}} \frac{\texttt{n}-\texttt{k}+\texttt{j}}{\texttt{j}}$ berechnet.

$\langle\ \texttt{k} \geq 0\ \rangle$ (Vorbedingung)

```
res = 1;
i = 0;
while (i < k) {
    i = i + 1;
    res = res * (n - k + i);
    res = res / i;
}
```

$\langle\ \texttt{res} = \Pi_{\texttt{j}=1}^{\texttt{k}} \frac{\texttt{n}-\texttt{k}+\texttt{j}}{\texttt{j}}\ \rangle$ (Nachbedingung)

a) 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.

Hinweise:

- Sie dürfen beliebig viele Zusicherungs-Zeilen ergänzen oder streichen. In der Musterlösung werden allerdings genau die angegebenen Zusicherungen benutzt.
- Das leere Produkt ist immer 1. Falls `k` den Wert 0 hat, gilt also $1 = \Pi_{\texttt{j}=1}^{\texttt{k}} \frac{\texttt{n}-\texttt{k}+\texttt{j}}{\texttt{j}}$.
- Bedenken Sie, dass die Regeln des Kalküls syntaktisch sind, weshalb Sie semantische Änderungen (beispielsweise von $\langle \texttt{x}+1 = \texttt{y}+1 \rangle$ zu $\langle \texttt{x} = \texttt{y} \rangle$) nur unter Zuhilfenahme der Konsequenzregeln vornehmen dürfen. Dies betrifft allerdings nicht das Setzen von Klammern, wenn dies durch Anwendung der Zuweisungsregel nötig ist (z. B. beim Hoare Tripel $\langle \texttt{x} = \texttt{y} * (\texttt{i} + 1) \rangle$ `i = i + 1;` $\langle \texttt{x} = \texttt{y} * \texttt{i} \rangle$).

$\langle k \geq 0 \rangle$

$\langle \_\_\_\_\_ \rangle$

```
res = 1;
```

$\langle \_\_\_\_\_ \rangle$

```
i = 0;
```

$\langle \_\_\_\_\_ \rangle$

$\langle \_\_\_\_\_ \rangle$

```
while (i < k) {
```

$\langle \_\_\_\_\_ \rangle$

$\langle \_\_\_\_\_ \rangle$

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

$\langle \_\_\_\_\_ \rangle$

```
    res = res * (n - k + i);
```

$\langle \_\_\_\_\_ \rangle$

```
    res = res / i;
```

$\langle \_\_\_\_\_ \rangle$

```
}
```

$\langle \_\_\_\_\_ \rangle$

$\langle \mathtt{res} = \Pi_{\mathtt{j}=1}^{\mathtt{k}} \frac{\mathtt{n}-\mathtt{k}+\mathtt{j}}{\mathtt{j}} \rangle$

b) Untersuchen Sie den Algorithmus P auf seine Terminierung. Für einen Beweis der Terminierung muss eine Variante angegeben werden und unter Verwendung des Hoare-Kalküls die Terminierung unter der Voraussetzung $\texttt{k} \geq 0$ bewiesen werden. Begründen Sie, warum es sich bei der von Ihnen angegebenen Variante tatsächlich um eine gültige Variante handelt.

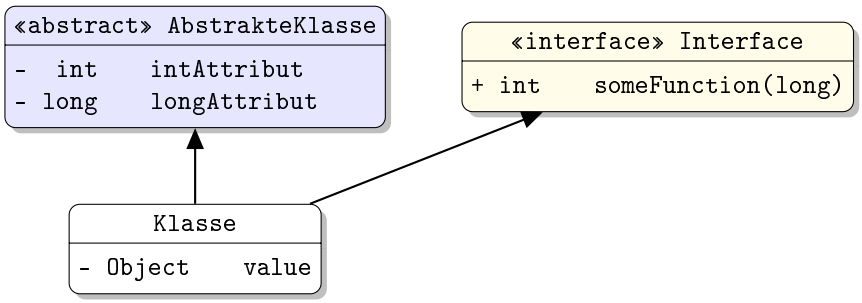
Aufgabe 3 (Klassen-Hierarchie): (7 + 9 = 16 Punkte)

In dieser Aufgabe betrachten wir verschiedene Arten von Studienleistungen und organisieren diese in einer Hierarchie.

- Zu einer Studienleistung wird der Name des zugehörigen Moduls gespeichert. Außerdem sind die maximal zu erreichende Punktzahl und die tatsächlich erreichte Punktzahl von Interesse.
- Studienleistungen sind entweder mündliche oder schriftliche Leistungen. Zu mündlichen Studienleistungen wird ein Prüfungsprotokoll angefertigt und als `String` gespeichert. Für schriftliche Prüfungsleistungen ist relevant, ob sie entzifferbar sind.
- Mündliche Leistungen sind entweder Referate oder mündliche Prüfungen. Ein Referat hat immer ein Thema.
- Schriftliche Leistungen können nur Hausaufgaben, Klausuren oder Seminararbeiten sein. Jede Seminararbeit hat einen Titel und verschiedene Referenzen (d. h. eine Sammlung der verwendeten Literatur-Quellen).
- Mündliche Studienleistungen, Klausuren und Seminararbeiten sind benotbar. Deshalb implementieren sie die Funktion `double berechneNote()`, welche die Note aus der gegebenen Punktzahl berechnet.
- Neben Referaten, mündlichen Prüfungen, Klausuren, Seminararbeiten und Hausaufgaben gibt es keine weiteren Studienleistungen.

a) Entwerfen Sie unter Berücksichtigung der Prinzipien der Datenkapselung eine geeignete Klassenhierarchie für die oben aufgelisteten Arten von Studienleistungen. 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 beschrieben, in dem der Name der Klasse sowie 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). Benutzen Sie `-`, um `private` abzukürzen, und `+` für alle anderen Sichtbarkeiten (wie z. B. `public`).

Hinweise:

- Sie brauchen keine "Uses" Beziehungen der Form $B \multimap A$ einzuzeichnen, die aussagen, dass `A` den Typ `B` verwendet.
- Implementiert eine Klasse die Methoden eines Interfaces, müssen Sie diese nicht erneut in der Klasse notieren.

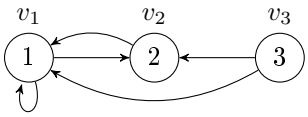
b) Schreiben Sie eine Java-Methode mit der folgenden Signatur:

```
public static double berechneNotenSchnitt(Studienleistung[] leistungen)
```

Diese Methode soll für alle benotbaren und entzifferbaren Studienleistungen des Eingabe-Arrays die Note berechnen und den Notenschnitt dieser Leistungen zurückgeben. Hierbei gilt eine Studienleistung auch dann als entzifferbar, falls keine Information über ihre Entzifferbarkeit vorliegt. Sie dürfen annehmen, dass das übergebene Array `leistungen` nicht `null` ist, dass es zu jedem Attribut geeignete Selektoren gibt und dass das Array `leistungen` mindestens eine benotbare und entzifferbare Leistung enthält.

Aufgabe 4 (Programmieren in Java): (3 + 4 + 10 + 6 + 4 + 9 = 36 Punkte)

Ein gerichteter Graph ist eine Datenstruktur, die aus Knoten und Kanten besteht. Jeder Knoten speichert einen Wert vom Typ `int`. Eine Kante $v_1 \to v_2$ verbindet einen Knoten v_1 mit einem Knoten v_2. Da Kanten gerichtet sind, sind die Kanten $v_1 \to v_2$ und $v_2 \to v_1$ unterschiedlich. Insbesondere kann es in *einem* Graph die Kante $v_1 \to v_2$ *und* die Kante $v_2 \to v_1$ geben. Es ist auch möglich, einen Knoten mit sich selbst zu verbinden, d. h., ein Graph kann die Kante $v_1 \to v_1$ enthalten.
Die Abbildung links zeigt beispielhaft den gerichteten Graph $\mathcal{G}$. Rechts ist die Datenstruktur zu sehen, mit der wir Knoten und Kanten darstellen.




```
public class Node {
  int value;
  NodeSet successors;
}
```

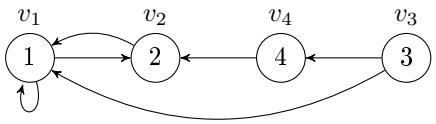
Wir speichern also pro Knoten einen Wert (im Attribut `value`) und einen Verweis auf eine Menge von Nachfolgern (im Attribut `successors`). Diese Menge ist mit der Klasse `NodeSet` realisiert. Eine Instanz der Klasse `NodeSet` repräsentiert eine Menge von Knoten, d. h. eine Menge von Instanzen der Klasse `Node`. Eine Kante $v_1 \to v_2$ gibt es genau dann, wenn die Menge v_1.`successors` den Knoten v_2 enthält. Für den Graph $\mathcal{G}$ gilt also z. B. v_1.`value` $= 1$ und v_1.`successors` $= \{v_1, v_2\}$.
Die Klasse `NodeSet` stellt die folgenden öffentlichen Methoden bereit:

- der Konstruktor ohne Argumente erzeugt eine leere Menge
- `boolean contains(Node n)` testet, ob `n` in der Menge enthalten ist
- `void add(Node n)` fügt `n` in die Menge ein, falls `n` noch nicht in der Menge enthalten ist
- `void remove(Node n)` entfernt `n` aus der Menge, falls `n` in der Menge enthalten ist
- `Node[] getElements()` gibt ein Array mit allen Elementen der Menge zurück

Sie dürfen in dieser Aufgabe davon ausgehen, dass die Argumente der zu implementierenden Methoden, die Rückgabewerte der Methoden der Klasse `NodeSet` und die Attribute der Klasse `Node` niemals `null` sind. Außerdem dürfen Sie davon ausgehen, dass das von `NodeSet.getElements()` zurückgelieferte Array nicht `null` ist und nicht `null` enthält.

a) Implementieren Sie eine öffentliche Methode `hasSelfLoop()` in der Klasse `Node`, die genau dann `true` zurückliefert, wenn es eine Kante `this` $\to$ `this` gibt, wobei `this` jener Knoten ist, auf dem die Methode aufgerufen wurde. Für den Graph $\mathcal{G}$ gilt also z. B. v_1.`hasSelfLoop`(), aber nicht v_2.`hasSelfLoop`().

b) Implementieren Sie eine öffentliche Methode `addBetween(Node n, Node m)` in der Klasse `Node`. Falls es eine Kante `this` $\rightarrow$ `m` gibt, soll diese Methode die Kante `this` $\rightarrow$ `m` entfernen und `n` zwischen `this` und `m` einfügen, sodass es Kanten `this` $\rightarrow$ `n` und `n` $\rightarrow$ `m` gibt. Andernfalls soll die Methode den Graph nicht verändern. Wenn v_4 ein Knoten mit Wert 4 ist, dann transformiert der Aufruf v_3.`addBetween`(v_4,v_2) den Graph $\mathcal{G}$ also zu folgendem Graph $\mathcal{G}'$:



Der Aufruf v_1.`addBetween`($\texttt{v}_4, \texttt{v}_3$) verändert den Graph $\mathcal{G}$ hingegen nicht, da es keine Kante $v_1 \rightarrow v_3$ gibt.

c) Implementieren Sie eine öffentliche Methode `getReachableNodes()` in der Klasse `Node`. Diese soll die Menge aller Knoten zurückliefern, die von dem Knoten `this` aus erreichbar sind. Diese Menge soll insbesondere auch den Knoten `this` selbst beinhalten. Für den Graph $\mathcal{G}$ gilt also z. B. v_1.`getReachableNodes`() = $\{v_1, v_2\}$ und v_3.`getReachableNodes`() = $\{v_1, v_2, v_3\}$. Bedenken Sie beim Implementieren der Methode, dass ein Graph (wie am Beispiel von $\mathcal{G}$ zu erkennen ist) Zyklen enthalten kann.

Hinweise:

- Die Lösung dieser Aufgabe vereinfacht sich, wenn Ihre Implementierung rekursiv arbeitet.
- Insbesondere kann es hilfreich sein, eine rekursive Hilfsmethode

```
void getReachableNodes(NodeSet res)
```

zu implementieren, welche die Menge `res` um alle Knoten erweitert, die vom aktuellen Knoten `this` aus erreichbar und noch nicht in `res` enthalten sind.

d) Implementieren Sie eine öffentliche Methode

```
List<Integer> smallerElements(int bound)
```

in der Klasse `Node`. Diese Methode soll eine Liste berechnen, welche diejenigen `value`-Werte der von `this` aus erreichbaren Knoten enthält, die kleiner als `bound` sind. Die Reihenfolge, in der die Werte in der Ergebnisliste stehen, ist unerheblich. Für den Graph $\mathcal{G}$ soll v_2.`smallerElements`(3) also eine Liste mit den Elementen `1` und `2` zurückliefern.

Hinweise:

- Sie können in dieser Teilaufgabe die Methode `getReachableNodes` auch dann verwenden, wenn Sie Teilaufgabe **c)** nicht gelöst haben.
- Die Klasse `LinkedList<T>` implementiert das Interface `List<T>` und verfügt über eine Methode `add(T o)`, die das Element `o` an das Ende der aktuellen Liste anhängt.
- Die Klasse `LinkedList<T>` hat einen Konstruktor ohne Argumente, der eine leere Liste erzeugt.

e) Wir betrachten das folgende Interface:

```
public interface Visitor {
  public void visit(Node n);
}
```

Implementieren Sie eine öffentliche Methode `accept` in der Klasse `Node`. Als Argument bekommt diese Methode ein Objekt `x` einer Klasse übergeben, die das Interface `Visitor` implementiert. Ihre Implementierung soll die `visit`-Methode des Objekts `x` für **alle** von dem Knoten `this` aus erreichbaren Knoten (und auch auf dem Knoten `this` selbst) aufrufen. Der "Ergebnistyp" der Methode `accept` ist `void`.

Verwenden Sie **nur Schleifen** und **keine Rekursion**.

Hinweise:

- Sie können in dieser Teilaufgabe die Methode `getReachableNodes` auch dann verwenden, wenn Sie Teilaufgabe **c)** nicht gelöst haben.

f) Ein *Independent Set* ist eine Teilmenge I der Knoten eines Graphen, sodass es keine Kante gibt, die zwei *unterschiedliche* Knoten aus I verbindet (d. h., Kanten der Form $v_1 \to v_1$ können in dieser Aufgabe ignoriert werden). Ein Independent Set I ist *maximal*, wenn jeder Knoten des Graphen, der nicht in I enthalten ist, mit mindestens einem Knoten aus I verbunden ist. Ein maximales Independent Set für den Graph $\mathcal{G}'$ aus Teilaufgabe **b)** wäre also z. B. $\{v_1, v_4\}$ oder $\{v_2, v_3\}$.

Wir erweitern die Klasse `Node` um die folgende Methode, die ein maximales Independent Set des Graphen berechnen soll, der aus `this` und allen von `this` aus erreichbaren Knoten besteht.

```
public NodeSet maximalIndependentSet() {
  IndependentSetVisitor x = new IndependentSetVisitor();
  accept(x);
  return x.result;
}
```

Vervollständigen Sie hierzu die folgende Klasse an der mit `...` gekennzeichneten Stelle:

```
public class IndependentSetVisitor implements Visitor {
  public NodeSet result = new NodeSet();
  public void visit(Node n) {
    ...
  }
}
```

Jedes Objekt `x` dieser Klasse hat ein Attribut `result`, um bereits berechnete Knoten des Independent Sets als Teilergebnis zu speichern. Zur Berechnung des Independent Sets soll beim Aufruf von `x.visit(n)` wie folgt vorgegangen werden: Der Knoten `n` wird genau dann zum Teilergebnis `x.result` hinzugefügt, wenn keiner seiner Nachfolger und keiner seiner Vorgänger bereits in `x.result` enthalten ist.

Aufgabe 5 (Haskell): (3 + 4 + 3 + 2 + 4 + 4 = 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 und die Funktion `==` den Typ `a -> a -> Bool` hat.

i)
```
f xs     []     zs     = f zs [] xs
f (x:xs) (y:ys) (z:zs) = if (x == z) then y else 1
```

ii)
```
g x []     = x
g x (y:ys) = y == ((g x ys) : y)
```

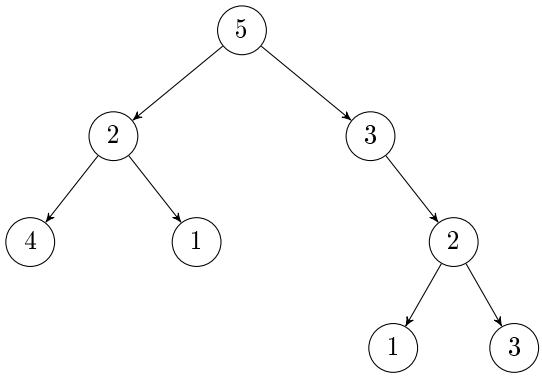
b) Bestimmen Sie, zu welchem Ergebnis die Ausdrücke `i` und `j` jeweils auswerten.

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

```
j :: [Int]
j = filter ((\h x -> h x > x) (\y -> y*y)) [-2,-1,0,1,2]
```

c) Implementieren Sie die Funktion `addElements :: [Int] -> [Int] -> [Int]` in Haskell, die zwei Listen `l1` und `l2` elementweise addiert und das Ergebnis als neue Liste zurückliefert. Hierbei soll das i-te Element der Ergebnisliste die Summe der i-ten Elemente von `l1` und `l2` sein, falls beide existieren, oder das i-te Element von `l1` bzw. `l2`, falls dieses nur in `l1` bzw. `l2` existiert. Zum Beispiel soll `addElements [2,5] [3,2,4]` die Liste `[5,7,4]` zurückgeben.

d) Wir betrachten nun binäre Bäume, in denen jeder Knoten einen `Int`-Wert speichert und höchstens zwei Nachfolger hat. Ein solcher Baum ist in der folgenden Grafik dargestellt:



Geben Sie die Definition einer Datenstruktur `Tree` für die Repräsentation entsprechender Bäume an.

e) Implementieren Sie die Funktion `sumWithinLevels :: Tree -> [Int]` in Haskell, die in einem gegebenen Baum die Integer-Werte aller Knoten auf gleicher Höhe addiert und in einer Liste zurückgibt. Dabei soll das i-te Element der Liste die Summe der Werte auf Höhe i beinhalten, wobei die Wurzel Höhe 1 hat. Zum Beispiel soll nach Aufruf der Funktion auf obigem Beispielbaum die Liste `[5,5,7,4]` zurückgegeben werden.

Hinweise:

- Sie dürfen die Funktion `addElements` aus Teilaufgabe **c)** verwenden.

f) Implementieren Sie die Funktion `height :: Tree -> Int` in Haskell, welche die Höhe eines Baums berechnet. Die *Höhe* eines Baums ist die maximale Anzahl von Knoten, die auf einem Pfad von der Wurzel zu einem Blatt des Baums liegen können. Die Höhe des in Aufgabe **d)** abgebildeten Baums ist also 4.

Hinweise:

- Sie dürfen die vordefinierte Funktion `max` zur Berechnung des Maximums zweier Zahlen benutzen. Beispielsweise liefert `max 3 4` das Ergebnis `4`.

Aufgabe 6 (Prolog): (2 + 6 + 3 + 5 + 4 = 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) $f(s(a), X, Y)), f(X, s(Y), b)$

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

b) Gegeben sei folgendes Prolog-Programm P.

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

Erstellen Sie für das Programm P den Beweisbaum zur Anfrage "`?- m(a,L).`" 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 "`?- m(a,L).`" an, die im Beweisbaum bis zur Höhe 4 enthalten sind.

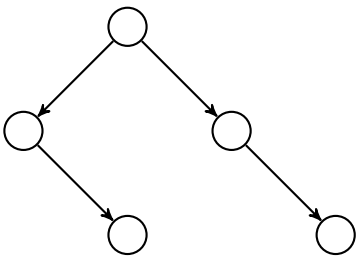
c) Implementieren Sie ein Prädikat `every_second` mit Stelligkeit 2 in Prolog, wobei `every_second(`t_1`,`t_2`)` genau dann gilt, wenn t_1 und t_2 Listen sind, sodass t_2 genau jedes zweite Element von t_1 enthält. Beispielsweise gilt `every_second`$([1,2,3,4],[2,4])$. Alle Anfragen der Form `every_second(`t_1`,`t_2`)`, bei denen t_1 keine Variablen enthält, sollen zu einem endlichen Beweisbaum führen.

Hinweise:

- Sie dürfen hierbei Prädikate späterer Aufgabenteile nur dann verwenden, wenn Sie diese auch selbst *korrekt* implementiert haben.

d) Natürliche Zahlen lassen sich mit Hilfe der Funktionssymbole `0` und `s` in sogenannter *Peano-Notation* darstellen (d. h., der Term `s(0)` stellt die Zahl 1 dar, `s(s(0))` stellt 2 dar, etc.). Implementieren Sie ein Prädikat `each` mit Stelligkeit 3 in Prolog, wobei `each(`t_1`,`t_2`,`t_3`)` genau dann gilt, wenn t_1 eine natürliche Zahl in Peano-Notation größer 0 ist und t_2 und t_3 Listen sind, sodass t_3 genau jedes t_1-te Element von t_2 enthält. Beispielsweise gilt `each(s(s(0)),`$[1,2,3,4],[2,4]$`)`. Alle Anfragen der Form `each(`t_1`,`t_2`,`t_3`)`, bei denen t_1 und t_2 keine Variablen enthalten, sollen zu einem endlichen Beweisbaum führen.

e) In dieser Teilaufgabe sollen Sie einen analogen Algorithmus zu Aufgabe 5 **f)** in Prolog programmieren. Binäre Bäume können in Prolog mit Hilfe der Funktionssymbole `empty` und `node` als Terme dargestellt werden. Dabei repräsentiert `empty` einen leeren Baum und `node(L,R)` repräsentiert einen Baum mit einem Wurzelknoten, der den Teilbaum `L` als linkes Kind hat und den Teilbaum `R` als rechtes Kind. Als Beispiel ist nachfolgend ein binärer Baum und seine Darstellung als Term angegeben.



```
node(node(empty,
          node(empty,
               empty)),
     node(empty,
          node(empty,
               empty)))
```

Implementieren Sie ein Prädikat `height` mit Stelligkeit 2 in Prolog, wobei `height(`t_1`,`t_2`)` genau dann gilt, wenn t_1 ein binärer Baum ist und t_2 die Höhe von t_1 als Integer-Zahl (d. h., hier müssen Sie die vordefinierten Zahlen in Prolog benutzen und nicht die Darstellung in Peano-Notation). Die *Höhe* eines Baums ist wieder die maximale Anzahl von Knoten, die auf einem Pfad von der Wurzel zu einem Blatt des Baums liegen können. Die Höhe des oben abgebildeten Baums ist also 3. Alle Anfragen der Form `height(`t_1`,`t_2`)`, bei denen t_1 keine Variablen enthält, sollen zu einem endlichen Beweisbaum führen.

Hinweise:

- Sie dürfen das Prädikat `maximum` mit Stelligkeit 3 benutzen, welches folgendermaßen implementiert ist:

```
maximum(X,Y,X) :- X >= Y.
maximum(X,Y,Y) :- Y > X.
```