Prof. Dr. Marc Stamminger
Prof. Dr. Michael Philippsen
Department Informatik, FAU

30. März 2016

# Klausur: Algorithmen und Datenstrukturen

**Angaben zur Person (Bitte Etikett aufkleben bzw. in Druckbuchstaben ausfüllen!):**

Name, Vorname: ....................................

Laufende Nr.: ....................................

Bitte kleben Sie hier das Etikett auf.

Matrikelnummer:

..............................

**Folgende Hinweise bitte lesen und Kenntnisnahme durch Unterschrift bestätigen!**

- **Hilfsmittel außer Schreibmaterialien sind *nicht* zugelassen**.
- Lösungen müssen in den dafür vorgesehenen freien Raum geschrieben werden. Sollte der Platz nicht ausreichen, verwenden Sie zunächst (mit kurzem Hinweis) die Zusatzseite am Ende. Weitere Zusatzseiten müssen von der Aufsicht ausgegeben und *eingeheftet* werden.
- Sie können Schmierpapier von der Aufsicht anfordern. ***Das Schmierpapier darf nicht mit abgegeben werden.***
- Können Sie die Prüfung aus gesundheitlichen Gründen nicht fortsetzen, dann müssen Sie Ihre Prüfungsunfähigkeit durch Vorlage eines *erweiterten ärztlichen Attestes* beim Prüfungsamt nachweisen. Melden Sie sich in jedem Fall bei der Aufsicht und lassen Sie sich das *entsprechende Formular* aushändigen.
- ***Überprüfen Sie diese Klausur auf Vollständigkeit (14 Seiten inkl. Deckblatt) und einwandfreies Druckbild!***

Durch meine Unterschrift bestätige ich den Empfang der vollständigen Klausurunterlagen und die Kenntnisnahme der obigen Informationen.

Erlangen, den 30.03.2016 ..............................................................

(Unterschrift)

---

Nicht von der Kandidatin bzw. vom Kandidaten auszufüllen!

**Bewertung** (Punkteverteilung unter Vorbehalt):

| Aufgabe | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | $\sum$ |
|---|---|---|---|---|---|---|---|---|---|
| Maximal | 14 | 9 | 17 | 10 | 14 | 20 | 20 | 16 | 120 |
| Erreicht | | | | | | | | | |
| | | | | | | | | | |

## Aufgabe 1 (Wissensfragen) (14 Punkte)

Bei den folgenden Teilaufgaben werden richtige Kreuze positiv (+) und falsche **oder fehlende** Kreuze entsprechend negativ (–) gewertet. Jede Teilaufgabe wird mit mindestens 0 Punkten bewertet. Pro Teilaufgabe ist mind. eine Aussage wahr. Kreuzen Sie alle richtigen Aussagen an:

**a)** Angenommen, `s1` und `s2` sind korrekt initialisierte Variablen vom Typ `String`. Welche der folgenden Anweisungen verursachen eine Fehlermeldung (entweder zur Übersetzungszeit mit `javac` oder zur Ausführungszeit mit `java`)?

- ☒ `boolean istNull = null.equals(s1);`
- ☐ `if (s1 == s2) System.out.println("sind␣gleich");`
- ☒ `boolean b = s1.charAt(s1.length()) == s2.charAt(0);`
- ☐ `int i = s1.toUpperCase().indexOf("null");`

**b)** Welche Aussagen sind wahr?

```
void sterneSehen(int x, int y) {
    for (int a = 0; a < x; a++) {
        for (int b = 0; b <= y; b++) {
            System.out.print("*");
        }
        System.out.println();
    }
}
```

- ☐ Der Aufruf `sterneSehen(10, 5)` gibt 50 Sterne aus.
- ☒ Der Aufruf `sterneSehen(10, 5)` gibt 60 Sterne aus.
- ☒ Für positive $x$ und $y$ hat `sterneSehen` eine Laufzeit von $\mathcal{O}(x \cdot y)$.
- ☐ `sterneSehen` terminiert nicht, wenn mindestens ein Argument negativ ist.

It Lösung nicht,
ich bin aber dafür!

**c)** Unter welchen Umständen ist die *sequentielle* (lineare) Suche nach einem Wert $v$ in einem aufsteigend sortierten Feld *schneller* als die *binäre* Suche?

- ☐ $v$ ist kleiner als das erste Element im Feld.
- ☒ $v$ ist genau das erste Element im Feld.
- ☐ Die Länge des Feldes ist keine 2er-Potenz.
- ☐ Die sequentielle Suche kann *niemals* schneller als die binäre Suche sein.

**d)** Bei einer einfach verketteten Liste $\mathcal{L}$ mit $n$ Elementen ...

- ☐ kann die Reihenfolge der Elemente in $\mathcal{O}(log(n))$ umgedreht werden.
- ☐ braucht das sortierte Einfügen ($\mathcal{L}$ bereits sortiert) im *worst-case* $\mathcal{O}(\sqrt{n})$ Vergleiche.
- ☒ hat eine Methode, die prüft, ob $\mathcal{L}$ leer ist, einen Zeitaufwand von $\mathcal{O}(1)$.
- ☐ kann das Einfügen am Listenende mit konstantem Aufwand durchgeführt werden.

**e)** Welche Aussagen zu Graphen stimmen?

☒ Hat mehr als ein Knoten Grad 0, dann ist der Graph *nicht* zusammenhängend.

☐ Ein Knoten mit Eingangsgrad 0 nennt man *Senke*.

☐ Ein DAG hat stets *genau eine* Quelle (Wurzel) und *genau eine* Senke.

☒ Ein DAG mit mindestens zwei Knoten ist *niemals* stark zusammenhängend.

**f)** *„Graham's Einpackalgorithmus"* aus der Vorlesung ...

☐ löst das Rucksackproblem für $n$ Elemente mit $\mathcal{O}(n^2)$ zusätzlichem Speicher.

☒ ermittelt die konvexe Hülle einer Punktewolke.

☐ ist ein gieriger Algorithmus zur Münzminimierung für kanonische Münzsysteme.

☒ untersucht und verwirft evtl. provisorische Lösungen.

**g)** Seien $v$ und $w$ konstante Laufzeiten und $T(n)$ die *worst-case*-Laufzeit einer Methode für ein Feld der Länge $n$. Was gilt für die asymptotisch obere Schranke im $\mathcal{O}$-Kalkül?

☐ $\mathcal{O}(g(n)) = \{h|\ \exists c > 0\ \exists n_0 > 0\ \forall n \geq n_0 :\ 0 \leq c \cdot g(n) \leq h(n)\}$

☒ $\mathcal{O}(g(n)) = \{h|\ \exists c_1 > 0\ \exists c_2 > 0\ \exists n_0 > 0 : \forall n \geq n_0 : 0 \leq c_1 \cdot g(n) \leq h(n) \leq c_2 \cdot g(n)\}$

☐ Für $T(0) = v$ und $T(n) = T(0.666 \cdot n) + w$ gilt: $T(n) \in \mathcal{O}(log(n))$

☐ Für $t(n) \in \mathcal{O}(m(n))$ und $f(n) \in \mathcal{O}(g(n))$ gilt: $w \cdot t(n) \cdot f(n) \in \mathcal{O}(m(n) \cdot g(n))$

**Aufgabe 2 (Streuspeicherung)** (9 Punkte)

**a)** Gegeben seien die folgenden Schlüssel $k$ zusammen mit ihren Streuwerten $h(k)$:

| $k$ | A | B | C | D | E |
|---|---|---|---|---|---|
| $h(k)$ | 1 | 3 | 3 | 3 | 0 |

Fügen Sie die Schlüssel $A - E$ in alphabetisch aufsteigender Reihenfolge in die folgende Streutabelle ein und lösen Sie Kollisionen durch verkettete Listen auf.

| *Fach* | *k (Liste, zuletzt eingetragener Schlüssel rechts)* |
|---|---|
| 42 | X $\rightsquigarrow$ Y $\rightsquigarrow$ Z $\rightsquigarrow\perp$ *(Beispiel)* |
| 0 | E |
| 1 | A |
| 2 | |
| 3 | B -> C -> D |
| 4 | |
| 5 | |
| 6 | |

**b)** Zum quadratischen Sondieren stehen nun zusätzlich die Streuwerte $h_i(k)$ zur Verfügung:

| $k$ | A | B | C | D | E |
|---|---|---|---|---|---|
| $h(k)$ | 1 | 3 | 3 | 3 | 0 |
| $h_1(k)$ | 2 | 4 | 4 | 4 | 1 |
| $h_2(k)$ | 5 | 0 | 0 | 0 | 4 |
| $h_3(k)$ | 3 | 5 | 5 | 5 | 2 |

Fügen Sie die Schlüssel $A - E$ in dieser Reihenfolge in eine neue Streutabelle mit den Fächern 0 bis 6 ein. Lösen Sie Kollisionen diesmal durch *quadratisches Sondieren* mit den obigen $h_i(k) = (h(k)+i^2)\ mod\ 7$ auf. Geben Sie in der folgenden Tabelle zu jedem Schlüssel an, welche Fächer Sie in welcher Reihenfolge sondiert haben und ob die Sondierung aufgrund einer Primär- ( $\overset{P}{\mapsto}$ ) oder Sekundärkollision ( $\overset{S}{\mapsto}$ ) notwendig wurde. Das jeweils zuerst betrachtete Fach ist bereits vorgedruckt.

| $k$ | *sondierte Fächer und Art der Kollision* |
|---|---|
| Z | 42 $\overset{P}{\mapsto}$ 47 $\overset{S}{\mapsto}$ 11 *(Beispiel)* |
| A | 1 |
| B | 3 |
| C | 3 P -> 4 |
| D | 3 P -> 4 S -> 0 |
| E | 0 P -> 1 S -> 4 S -> 2 |

## Aufgabe 3 (Binäre Suche) (17 Punkte)

Im Folgenden sollen Sie einen Schlüssel `t` in einem Feld `ts` mittels *binärer Suche* lokalisieren. Für die Schlüssel vom Typ `T` gibt es zwei Vergleicher `c1` bzw. `c2` und das Feld `ts` ist so sortiert, dass:

$$\forall i < j : ts[i] \prec_{c1} ts[j] \vee (ts[i] =_{c1} ts[j] \wedge ts[i] \preceq_{c2} ts[j])$$

Beispiel: Tupel `T := (String, Integer)`, `c1` vergleicht `Strings`, `c2` vergleicht `Integers`:

| | | | | | |
|---|---|---|---|---|---|
| $(AuD, 42)$ | $(AuD, 666)$ | $(AuD, 666)$ | $(PFP, 41)$ | $(PFP, 666)$ | $(PFP, 4711)$ |

Hinweis zur API der Methode `int compare(T o1, T o2)` im Interface `Comparator<T>`: *Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.*

Ergänzen Sie die <u>iterative</u> Methode `suche` so, dass sie den Index von `t` in `ts` mit einer Laufzeit von $\mathcal{O}(log(ts.length))$ zurückgibt, falls `t` in `ts` vorkommt, andernfalls sei ihr Ergebnis $-1$:

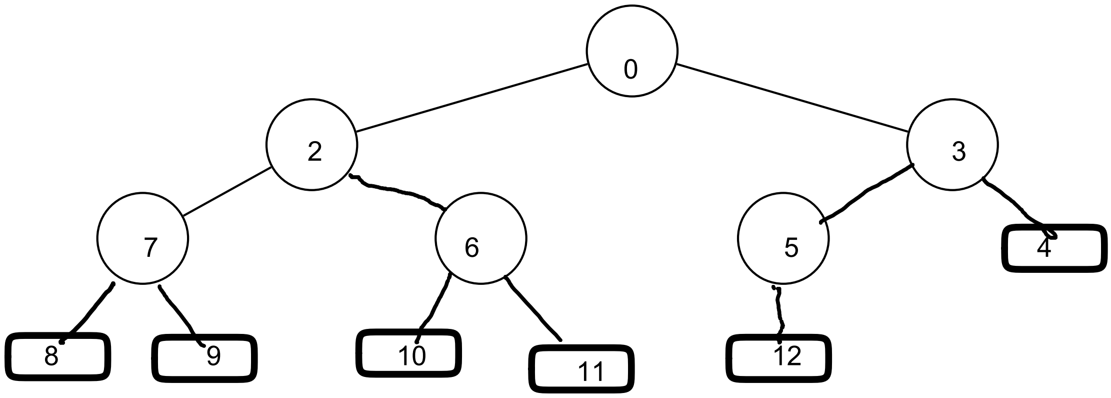
```
<T> int suche(T[] ts, T t, Comparator<T> c1, Comparator<T> c2) {
    int a = 0, m, z = ts.length - 1; // Anfang, Mitte, Ende

    // schon alles durchsucht?
    while ( m != a && m != z )
        // neue Mitte zwischen a und z bestimmen (abgerundet)
        m = (a + z)/2;
        if ( c1.compare(t, ts[m]) < 0 )
            // Fall A: t VOR ts[m] laut c1
            z = m-1;
        } else if ( c1.compare(t, ts[m]) == 0 )
            // Fall B: t NAHE BEI ts[m] laut c1
            if ( c2.compare(t,ts[m] ) < 0
                // Fall B1: t VOR ts[m] laut c2
                z = m-1;
            } else if ( c2.compare(t,ts[m]) == 0 )
                // Fall B2: gefunden :)!
                return m;
            } else {
                // Fall B3: t NACH ts[m] laut c2
                a = m+1;
            }
        } else {
            // Fall C: t NACH ts[m] laut c1
            a = m+1;
        }
    }
    // nicht gefunden :(
    return -1;
}
```

## Aufgabe 4 (Halden) (10 Punkte)

Gegeben sei folgende Feld-Einbettung einer Min-Halde:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 3 | 7 | 6 | 5 | 4 | 8 | 9 | 10 | 11 | 12 |

**a)** Stellen Sie die Halde graphisch als Baum dar; ergänzen Sie ggf. Knoten und/oder Kanten:




**b)** Entfernen Sie das kleinste Element (die Wurzel 0) aus der *unten gegebenen* initialen Halde, stellen Sie die Haldeneigenschaft wieder her und geben Sie das Endergebnis an:

| *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* | *11* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 3 | 7 | 6 | 5 | 4 | 8 | 9 | 10 | 11 | 12 |

↙ 0 *entfernen* ↙

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 3 | 7 | 10 | 5 | 4 | 8 | 9 | 12 | 11 |

**c)** Fügen Sie nun den Wert 1 in die *unten gegebene* initiale Halde ein, stellen Sie die Haldeneigenschaft wieder her und geben Sie das Endergebnis an:

| *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* | *11* | *12* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 3 | 7 | 6 | 5 | 4 | 8 | 9 | 10 | 11 | 12 | |

↘ 1 *einfügen* ↘

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 1 | 7 | 6 | 3 | 4 | 8 | 9 | 10 | 11 | 12 | 5 |

## Aufgabe 5 (Sortieren) (14 Punkte)

**a)** Führen Sie „*Sortieren durch Einfügen*“ lexikographisch *aufsteigend*, *in-situ* und <u>*stabil*</u> in einem Schreibtischlauf auf folgendem Feld aus. Jede Zeile stellt den Zustand des Feldes dar, nachdem das *jeweils nächste* Element in die Endposition verschoben wurde. Der bereits sortierte Teilbereich steht vor |||. Gleiche Elemente tragen zwecks Unterscheidung ihre „Objektidentität“ als Index (z.B. `"`$A_1$`".equals("`$A_2$`")` aber `"`$A_1$`" != "`$A_2$`"`).

<table>
<tr><th>L |||</th><th>$A_1$</th><th>$B_1$</th><th>F</th><th>$A_2$</th><th>$B_2$</th></tr>
<tr><td>$A_1$</td><td>L |||</td><td>$B_1$</td><td>F</td><td>$A_2$</td><td>$B_2$</td></tr>
<tr><td>$A_1$</td><td>$B_1$</td><td>L |||</td><td>F</td><td>$A_2$</td><td>$B_2$</td></tr>
<tr><td>$A_1$</td><td>$B_1$</td><td>F</td><td>L |||</td><td>$A_2$</td><td>$B_2$</td></tr>
<tr><td>$A_1$</td><td>$A_2$</td><td>$B_1$</td><td>F</td><td>L |||</td><td>$B_2$</td></tr>
<tr><td>$A_1$</td><td>$A_2$</td><td>$B_1$</td><td>$B_2$</td><td>F</td><td>L |||</td></tr>
</table>

**b)** Ergänzen Sie die folgende Methode so, dass sie die Zeichenketten im Feld `a` lexikographisch aufsteigend *durch Einfügen sortiert*. Sie muss *iterativ*, *in-situ* und <u>*stabil*</u> sortieren. Außerdem dürfen Sie keine weiteren Variablen deklarieren, als die bereits vorgegebenen. Sie dürfen davon ausgehen, dass kein Eintrag im Feld `null` ist.

```
void sortierenDurchEinfuegen(String[] a) {
    // Hilfsvariable:
    String tmp;
    // bearbeite die Elemente der Reihe nach:
    for (int n = 1; n < a.length; n++) {
        // entnimm das naechste Element aus dem Feld:
        tmp = a[n];
        // verschiebe dieses, bis die Zielposition frei ist:
        int i = n - 1;
        while (i >= 0 && tmp.compareTo(a[i]) < 0) {
            // schiebe dabei alle Elemente Richtung Ende:
            a[i+1] = a[i];
            i-- ;
        }
        // setze das entnommene Element ein:
        a[i+1] = temp;
    }
}
```

**Aufgabe 6 (Dynamische Programmierung)** (20 Punkte)

Die sogenannten *Großen Schröder-Zahlen* sind für ganzzahlige positive $n$ wie folgt definiert:

$$a(n) = \begin{cases} 1 & falls \quad n \leq 1 \\ a(n-1) + \sum_{i=1}^{n-1} a(i) \cdot a(n-i) & sonst \end{cases}$$

**a)** Um welche Art der Rekursion handelt es sich bei `a(n)`?

`a(n):` Kaskadierende Rekursion

**b)** Ergänzen Sie die naive Implementierung der obigen Funktion ohne weitere Optimierungen:

```
long a(int n) {
    if (n <= 1) {
        // Basisfall:
        return 1;
    } else {
        // Rekursion:
        long an = a(n-1);
        for (i=1; i<n ; i++) {
          an += a(i) · a(n-i);
        }
        return an;
    }
}
```

**c)** Mittels *Dynamischer Programmierung* soll `aDP(n)` jede *Schröder-Zahl* höchstens einmal berechnen, auch wenn sie mehrmals benötigt wird. Dazu soll `aDP(n)` bereits berechnete $a(i)$ im Feld `mem[i]` verwalten. Das Feld `mem` wird nur bei Bedarf und höchstens bis zum erforderlichen Umfang vergrößert – dabei müssen die bisherigen Werte gerettet werden!

```
long[] mem;

long aDP(int n) {
    // Speicher ggf. passend vergroessern oder neu anlegen:
    if (mem == null || n ≥ mem.length) {
        long[] oldMem = mem;
        mem = new long[n+1];
        if (oldMem != null) {
            for (i=0; i < oldMem.length; i++) {
                mem[i] = oldMem[i];
            }
        }
    }
```

```
    if (n <= 1) {
        // Basisfall:
        mem[n] = 1;
    } else if ( mem[n]==0 ) {
      return mem[n];
    else {
       mem[n] = aDp(n-1);
       for(int i=1; i<n; i++)
           mem[n] += aDp(i)* aDp(n-1);
    }

    return mem[n];

}
```

## Aufgabe 7 (Graphen) (20 Punkte)

In dieser Aufgabe wird ein gerichteter Graph $G = (V, E)$ als Adjazenzmatrix `am` repräsentiert. Sie dürfen davon ausgehen, dass `am` wohlgeformt ist (also quadratisch und ohne `null`-Zeilen).

**a)** Ergänzen Sie die Methode `sammle`, die alle mit einem Knoten `k` in $G$ direkt oder indirekt verbundenen Knoten in der Menge `verb` sammelt. Adjazente Knoten gelten ***un**geachtet* der Kantenrichtung als *verbunden*. Bereits besuchte Knoten werden in `bes` verwaltet:

```
void sammle(boolean[][] am, int k, Set<Integer> verb, Set<Integer> bes) {
    if (!bes.contains(k)) {
        verb.add(k);
        bes.add(k);

        for (int i = 0; i < am.length ;i++) {
            if (am[i][k])
                sammle (am, i, verb, bes);
        }
        for (int i=0; i < am[k].length; i++) {
            if (am[k][i])
                sammle(am, i, verb, bes);
        }
    }
}
```

**b)** Die Methode `mszt` soll die Knotenmengen aller maximalen *schwach zusammenhängenden* Teilgraphen (sog. schwache Zusammenhangskomponenten) zurückgeben:

```
List<Set<Integer>> mszt(boolean[][] am) {
    // Ergebnisliste:
    List<Set<Integer>> ergebnis = new LinkedList<>();
    // Menge besuchter Knoten:
    Set<Integer> bk = new HashSet<>();
    // Ermittle alle Teilgraphen mittels Hilfsmethode sammle:
    for (int k = 0; k < am.length; k++) {
        // falls k noch nicht besucht => sammle mit k verbundene Knoten
        if (!bk.contains(k)) {
            set<Integer> verb = new HashSet();
            sammle(am, k, verb, bk);
            ergebnis.add(verb);
        }
    }
    return ergebnis;
}
```

**c)** *Definition aus der Vorlesung: „Ein Graph $G' = (V', E')$ ist ein* ***induzierter Teilgraph*** *eines Graphen $G = (V, E)$ genau dann, wenn $V' \subseteq V$ und $E' = \{(v, w) \in E \mid v, w \in V'\}$."*

Ergänzen Sie die Methode `itg` so, dass sie aus `am` jede Kante $(x, y)$ entfernt, wenn nicht sowohl $x$ als auch $y$ in `vs` ($\triangleq V'$) enthalten sind. Entfernen Sie (zur Vereinfachung) *keine* Knoten aus `am`.

```
void itg(boolean[][] am, Set<Integer> vs) {
```

```
for (int i=0; i< am.length, i++){        //alle Felder aus am prüfen
   for ( int j=0; j< am[i].length; j++){
      if(am[i][j]){
         if( vs.contains(i) && vs.contains(j)){
            //ok
         }
         else{
            am[i][j] = false;        //Felder "entfernen"
         }
      }
   }
}
```

```
}
```

## Aufgabe 8 (ADT) (16 Punkte)

Gegeben seien folgende abstrakte Datentypen:

---

**adt** *S // Menge (Set)*
**sorts** *S, int, boolean*
**ops**

| | | | |
|---|---|---|---|
| *Empty:* | | $\mapsto S$ | *// erzeugt leere Menge:* $M = \emptyset$ |
| *Add:* | $S \times int$ | $\mapsto S$ | *// ergänzt Wert n:* $M \leftarrow M \cup \{n\}$ |
| *isIn:* | $S \times int$ | $\mapsto boolean$ | *// prüft, ob Wert n in M enthalten ist:* $n \overset{?}{\in} M$ |

**axs**
... *// aus Platzgründen weggelassen*
**end** *S*

---

**adt** *G // Graph*
**sorts** *G, int, S, boolean*
**ops**

| | | | |
|---|---|---|---|
| *New:* | | $\mapsto G$ | *// erzeugt neuen Graphen:* $G = (V, E)$ *mit* $V = \emptyset$ |
| *Node:* | $G \times int$ | $\mapsto G$ | *// ergänzt Knoten n:* $V \leftarrow V \cup \{n\}$ |
| *Edge:* | $G \times int \times int$ | $\mapsto G$ | *// ergänzt Kante* $(a, b)$*:* $V \leftarrow V \cup \{a, b\}$, $E \leftarrow E \cup \{(a, b)\}$ |
| *collect:* | $G$ | $\mapsto S$ | *// ermittelt Menge V aller Knoten in G* |
| *path:* | $G \times int \times int$ | $\mapsto boolean$ | *// existiert* ***gerichteter*** *Weg zwischen zwei Knoten?* |
| *isRoot:* | $G \times int$ | $\mapsto boolean$ | *// prüft, ob der Knoten eine Quelle in G ist* |

**axs**
... *// im Folgenden zu ergänzen*
**end** *G*

---

Zusätzlich stehen Ihnen die Datentypen *int* und *boolean* mit den aus Java bekannten Ausprägungen und Operatoren zur Verfügung.

**a)** Ergänzen Sie den ADT $G$ um Axiome für die Operation `collect`, die die Menge $V$ aller Knoten im Graphen ermittelt:

$$collect(New) = Empty$$

S vorher um n erweitert

$$collect(Node(g, n)) = Add(\overbrace{collect(g)}, n)$$

$$collect(Edge(g, a, b)) = \underbrace{Add(\underbrace{Add(collect(g), a)}_{\text{S um a erweitern}}, b)}_{\text{S um b erweitern}}$$

**b)** Ergänzen Sie den ADT $G$ um Axiome für die Operation `path`, die genau dann `true` ergibt, wenn es im **gerichteten** Graphen $G$ (d.h. $edge(a, b)$ erzeugt eine **gerichtete** Kante) einen Pfad zwischen den zwei übergebenen Knoten $x$ und $y$ gibt:

$$path(New,\ x,\ y) = false$$
$$path(g,\ x,\ x) = true$$
$$path(Edge(g,\ a,\ b),\ a,\ b) = true$$

$$path(Node(g,\ n),\ x,\ y) = path(g, x, y)$$

$$path(Edge(g,\ a,\ b),\ x,\ y) = \begin{cases} path(g,x,a) \wedge path(g,b,y) \vee path(g,y,a) \wedge path(g,b,x) & : true \\ path(g,x,y) & \end{cases}$$

**c)** Ergänzen Sie den ADT $G$ um Axiome für die Operation `isRoot`, die genau dann `true` ergibt, wenn der übergebene Knoten eine *Quelle* im **gerichteten** Graphen $G$ ist:

$$isRoot(New,\ x) = false$$
$$isRoot(Node(g,\ n),\ x) =$$
$$= (x = n) \wedge \neg\ isIn(collect(g),\ x) \vee isRoot(g,\ x)$$

$$isRoot(Edge(g,\ a,\ b),\ x) =$$

**Zusatzseite**