Technische Universität Braunschweig

Institut für
Betriebssysteme und Rechnerverbund

Prof. Dr. H. Langendörfer · Prof. Dr. M. Zitterbart

Vordiplomklausur Praktische Informatik

16. März 1999

Name, Vorname: ____________________________________

Matrikelnummer: ____________________________________

Fachrichtung, Fachsemester: ____________________________

Zeitdauer: 4 Stunden

Zugelassene Hilfsmittel: Vorlesungs-Unterlagen, Übungen und eigene Aufzeichnungen.
Keine alten Klausuren.

Maximale Punktzahl: 78. Die Klausur ist mit 32 Punkten bestanden.

Aufgabe	max. Punkte	err. Punkte
1	6	
2	7	
3	10	
4	15	
5	10	
6	12	
7	8	
8	10	
$\sum$	78	

Note: __________

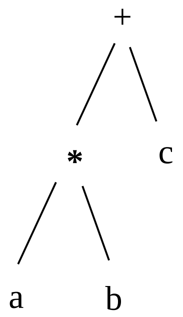
Aufgabe 1: Bäume

(2 + 3 + 1)

Ein arithmetischer Ausdruck A mit Additionen und Multiplikationen über einer Menge V von Variablen werde wie folgt als markierter Binärbaum $T(A)$ dargestellt:

- Die Blätter sind mit Variablen $v \in V$ markiert.
- Sind $T(A_1)$ und $T(A_2)$ zwei solche Bäume, so stellt der Baum, der entsteht, wenn man $T(A_1)$ und $T(A_2)$ als Söhne eines neuen Knotens einsetzt, der mit $+$ bzw. $*$ markiert ist, den arithmetischen Ausdruck $A_1 + A_2$ bzw. $A_1 * A_2$ dar.

Der Ausdruck $(a * b) + c$ wird beispielsweise durch den folgenden Baum dargestellt:



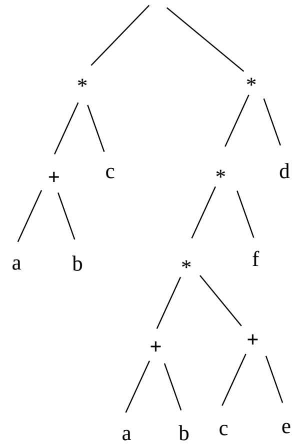
(a) Geben Sie den Binärbaum für den Ausdruck $(a + b) * c + (d * ((a + b) * (c + e) * f))$ an.

(b) Formulieren Sie einen Algorithmus in JAVA-Pseudocode, der einen als Baum vorliegenden arithmetischen Ausdruck auswertet.

(c) Stellen Sie den Ausdruck aus (a) als gerichteten, azyklischen Graphen dar, in dem gemeinsame arithmetische Teilausdrücke nur einmal vorkommen.

Lösung 1. Bäume

(a) Binärbaum:



(b) `Inhalt(t)` liefert den Inhalt des Wurzelknotens des Baums `t` zurück

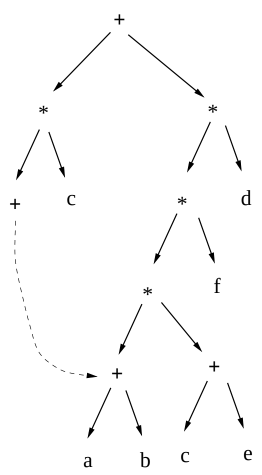
```
int Auswertung(Tree t)
{
  if (t == null)
    return 0;

  if (Inhalt(t) ist Element von V)
    return(Inhalt(t));

  if (Inhalt(t) == "+")
    return (Auswertung(linkerTeilbaum(t)) + Auswertung(rechterTeilbaum(t)));

  if (Inhalt(t) == "*")
    return (Auswertung(linkerTeilbaum(t)) * Auswertung(rechterTeilbaum(t)));
}
```

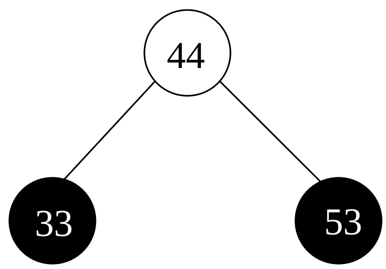
(c) Graph:



Aufgabe 2: Rot-Schwarz-Bäume

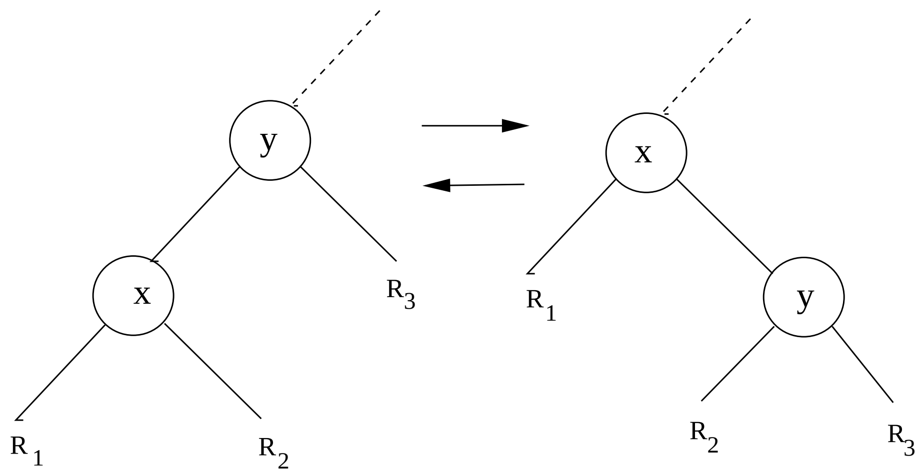
(2 + 2 + 3)

(a) Betrachten Sie folgenden Rot-Schwarz-Baum T. Geben Sie den Rot-Schwarz-Baum T' an, der aus T durch Einfügen der Werte $\{25, 38, 50, 45\}$ hervorgeht. Geben Sie dabei alle entstehenden (Teil-)Bäume an.



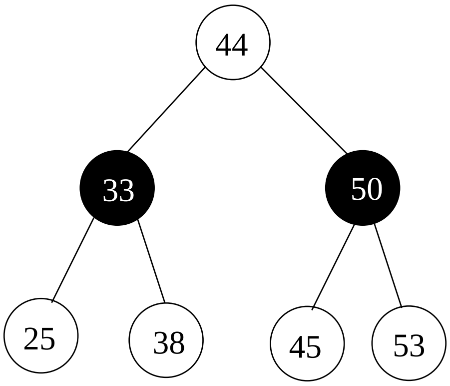
(b) Die Datenstruktur der Rot-Schwarz-Bäume soll für die folgenden Aufgaben dahingehend erweitert werden, daß zusätzlich in jedem Knoten k die *Höhe* von k, d.h. die Höhe des Teilbaums, der k als Wurzel besitzt, gespeichert wird.
Für welche Knoten muß beim Einfügen bzw. Löschen eines Knotens die Höhe neu berechnet werden, wenn *keine* Rotationen auftreten?

(c) Nehmen wir an, daß im folgenden Ausschnitt Rotationen vorkommen können. Gegeben sei ein Rot-Schwarz-Baum R mit n Knoten. Betrachten Sie folgen Ausschnitt aus R mit den Teilbäumen R_1, R_2, R_3 sowie den Knoten x, y. Für welche Knoten im Ausschnitt und im Rest von R muß die Höhe bei einer Rechtsrotation um y bzw. Linksrotation um x neu berechnet werden? Welchen Zeitaufwand benötigen die Rotationen? Begründen Sie Ihre Antworten.



Lösung 2. Rot-Schwarz-Bäume

(a) Baum:



(b) Für alle Knoten auf dem Weg zur Wurzel, bis keine Höhenänderung mehr stattfindet.

(c) Die Höhe muß für `x`, `y` sowie alle Knoten aus b) neu berechnet werden. Die Rotationen benötigen $O(log\, n)$, da Höhenänderungen u.U. bis zur Wurzel propagiert werden müssen.

Aufgabe 3: Sortieren

(4 + 6)

(a) Gegeben sei eine zu sortierende Zahlenfolge, nicht in einem Array, sondern in Form einer linearen, einfach verketten Liste. Die Zugriffszeit auf das erste und das jeweils nächste Element ist $O(1)$ und auf das n-te Element $O(n)$.

   (1) Welcher Sortieralgorithmus aus der Vorlesung besitzt unter diesen Umständen die geringste Komplexität? Begründung.

   (2) Was ist die Laufzeit im schlechtesten Fall für Ihren gewählten Algorithmus?

(b) Nehmen Sie an, die einzigen Werte, die als Schlüsselwerte auftreten, sind 0 und 1. Wie wirkt sich dies auf das worst-case-Verhalten von Quicksort aus, wie auf InsertionSort?

   (1) Geben Sie für beide Algorithmen eine Eingabe an, die den worst case beschreibt und geben Sie die Laufzeit in diesen Fällen an.

   (2) Geben Sie einen Algorithmus an, der die Sortierung in $O(n)$ herstellt.

Lösung 3. Sortieren

(a) (1) Ideal ist Mergesort: Divide Schritt hat jetzt Komplexität O(n) statt O(1), aber egal, da Conquer nach wie vor die Komplexität O(n) hat.
Quicksort ist auch möglich, aber Partitionierungs-Algorithmus muß komplett geändert werden, damit er noch in Aufwand O(n) funktioniert, ansonsten steigt der Aufwand.
falsch: InsertionSort, $O(n^2)$ bleibt, wenn im sortierten Array von vorne eingefügt wird. $O(n^3)$ wenn standardmäßig von hinten eingefügt wird. Heapsort Komplexität steigt ebenfalls

   (2) Mergesort: $O(nlogn)$,

$$T(n) = \begin{cases} O(1) & \text{falls} \quad n = 1 \\ O(n) + 2 * T(n/2) + O(n) & \text{falls} \quad n > 1 \end{cases}$$

Quicksort: $O(nlogn)$, im schlechtesten Fall $O(n^2)$,

$$T(n) = \begin{cases} O(1) & \text{falls} \quad n = 1 \\ O(n) + T(i) + T(n-i) + O(1) & \text{falls} \quad n > 1 \end{cases}$$

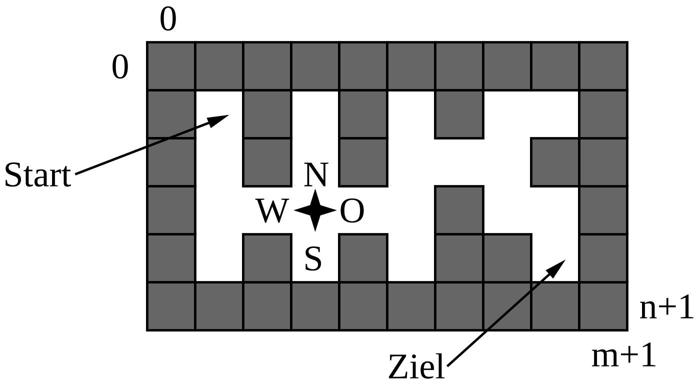
(b) (1) Quicksort: ungünstige Partitionierung möglich, Laufzeit $O(n^2)$. Eingabe: Array zur Hälfte aus 1'en und zur anderen Hälfte aus 0'en.
InsertionSort: andauerndes Vertauschen von Elementen nötig, Laufzeit $O(n^2)$. Eingabe: Array zuerst $n/2$ 1'en, dann $n/2$ 0'en, also genau umgekehrte Sortierung. Bei Implementierung als verkette Liste: Array zuerst $n/2$ 0'en, dann $n/2$ 1'en.

   (2) **Mögliche Lösung**:
1. Anzahl der 0'en und 1'en im Array zählen, Aufwand $O(n)$.
2. Array mit der unter 1. gezählten Anzahlen 0'en und danach 1'en füllen, Aufwand $O(n)$.
Gesamtaufwand: $O(n) + O(n) = O(n)$

Aufgabe 4: Algorithmenkonstruktion

(4 + 3 + 3 + 5)

Gegeben sei ein Labyrinth aus freien Wegen und Wänden, sowie 4 Bewegungsrichtungen (N, O, S, W). Gesucht ist ein Weg, d.h. eine durchgehende Folge freier und paarweise verschiedener Felder vom Start (1,1) zum Ziel (m,n).



Die Implementierung sei durch eine $(m \times n)$-Matrix gegeben, die von einer Reihe von Wandfeldern umgeben ist, so daß sich insgesamt ein $(m+2) \times (n+2)$ großes Labyrinth ergibt. Hierfür sind folgende JAVA-Deklarationen vorgegeben:

```
class Punkt {
  public int x,y;
}

class Labyrinth {
  public final static int     Wand = 0, Weg = 1;
  public final static int     Nord = 0, Ost = 1, Sued = 2, West = 3;

  int m, n;             // m = Breite, n = Hoehe des Labyrinths
  Punkt Delta[West+1]; // relative Bewegungsrichtungen fuer Nord/Sued/Ost/West,
                        // Nord: Delta[Nord]={-1,0}, Ost: Delta[Ost]={0,1}, ...
  int lab[][];          // 2D-Array, enthaelt Labyrinth: Groesse=[m+2][n+2],
                        // moegliche Werte: Wand, Weg
}
```

(a) Programmanalyse:

Gegeben sei folgende JAVA-Funktion `existiert_Weg`, die überprüft, ob es von der als Parameter `koord` gegebenen Stelle aus einen Weg durch das Labyrinth bis zum Ziel (m,n) gibt. Die Suche wird initial mit dem Startpunkt (1,1) aufgerufen. Gehen Sie davon aus, daß die Variablen `m, n, Delta, lab` geeignet initialisiert sind.

```
boolean existiert_Weg ( Punkt koord ) {
  Punkt   pneu = new Punkt();
  int     aktRichtung;
  boolean gefunden = false;

  if ( koord.x == m and koord.y == n )
    gefunden = true;
  else
    for ( int aktRichtung = Nord; aktRichtung <= West; aktRichtung++ ) {
      pneu.x = koord.x + Delta[aktRichtung].x;
      pneu.y = koord.y + Delta[aktRichtung].y;
      if ( lab[pneu.x][pneu.y] == Weg )
        gefunden = gefunden || existiert_Weg(pneu);
    }
  return gefunden;
}
```

Welche(n) Fehler enthält die Funktion `existiert_Weg`? Begründen Sie Ihre Antwort!

(b) Programmkorrektur:

Modifizieren Sie die Funktion `existiert_Weg` so, daß sie keine(n) Fehler mehr enthält. Geben Sie alle Änderungen an!

(c) Datenstrukturen:

Unter welchen Voraussetzungen lassen sich die Wege des Labyrinths als Baum darstellen?

Definieren Sie unter dieser Prämisse eine Datenstruktur `Raum` in JAVA, mit deren Hilfe die Wege des Labyrinths als Baum mit dem Start als Wurzel darstellbar sind. Verwenden Sie die bisherigen Koordinaten als Schlüssel.

(d) Programmentwurf:

Schreiben Sie eine Funktion `erzeuge_Baum`, die aus einem Labyrinth in Array-Darstellung ausgehend vom Start für die Wege einen Baum gemäß Ihrer Definition in (c) erzeugt.

Geben Sie an, wie diese Funktion aufgerufen werden muß, wenn der Baum mit dem Start(1,1) als Wurzel aufgebaut werden soll.

Lösung 4. Algorithmenkonstruktion

(a) `existiert_Weg` terminiert nicht!

Es wird in **allen** Richtungen gesucht. Damit läuft `existiert_Weg` immer wieder einen bereits gegangenen Weg zurück. **Folge:** endlose Rekursion!

Zwei Syntaktische Fehler:

```
if ( koord.x == m && koord.y == n )
```

```
for ( aktRichtung = Nord; aktRichtung <= West; aktRichtung++ )
```

(b) Zur Vermeidung von Zyklen muß `existiert_Weg` sich merken, welche Felder schon besucht wurden. Deshalb werden die Werte, die ein Feld in `lab[][]` enthalten kann um den Wert `Besucht = 2` erweitert. Bei jedem Eintritt in die Funktion `existiert_Weg` wird dann die Belegung des aktuellen Feldes von `Weg` auf `Besucht` geändert und vor dem Verlassen der Funktion wieder in `Weg` zurückgeändert.

```
class Labyrinth {
  public final static int     Wand = 0, Weg = 1, Besucht = 2;
  public final static int     Nord = 0, Ost = 1, Sued = 2, West = 3;

...

  boolean existiert_Weg ( Punkt koord ) {
    Punkt   pneu = new Punkt();
    int     aktRichtung;
    boolean gefunden = false;

    lab[koord.x][koord.y] = Besucht;  // NEU

    if ( koord.x == m && koord.y == n )
      gefunden = true;
    else
      for ( aktRichtung = Nord; aktRichtung <= West; aktRichtung++ ) {
        pneu.x = koord.x + Delta[aktRichtung].x;
        pneu.y = koord.y + Delta[aktRichtung].y;
        if ( lab[pneu.x][pneu.y] == Weg )
          gefunden = gefunden || existiert_Weg(pneu);
      }

    lab[koord.x][koord.y] = Weg;     // NEU

    return gefunden;
  }
}
```

(c) Das Labyrinth ist Zyklen - frei, d.h. der Weg zwischen 2 Punkten des Labyrinths ist eindeutig bestimmt.

```
class LabTree {
  Punkt   koord;
  LabTree Nachfolger[West+1];
}
```

(d)
```
LabTree erzeuge_Baum ( Punkt koord ) {
  Punkt   pneu = new Punkt ();
  int     aktRichtung;
  LabTree hier = new LabTree();

  lab[koord.x][koord.y] = Besucht;

  hier.koord.x = koord.x;
  hier.koord.y = koord.y;

  for ( aktRichtung = Nord; aktRichtung <= West; aktRichtung++ ) {
    pneu.x = koord.x + Delta[aktRichtung].x;
    pneu.y = koord.y + Delta[aktRichtung].y;
    if ( lab[pneu.x][pneu.y] == Weg )
      hier.Nachfolger[aktRichtung] = erzeugeBaum ( pneu );
    else
      hier.Nachfolger[aktRichtung] = null; // keine gueltige Richtung
  }

  lab[koord.x][koord.y] = Weg;

  return hier;
}
```

Die Funktion muß wie folgt aufgerufen werden, wenn der Baum mit dem Start(1,1) als Wurzel aufgebaut werden soll:

```
Wurzel = erzeuge_Baum(1,1);
```

Aufgabe 5: Abstrakte Datentypen

(6 + 4)

Gegeben ist ein Datentyp **STRING** basierend auf dem Elementtyp **CHAR** und den folgenden Signaturen:

`create:`		$\rightarrow$ **STRING**
`a:`	**STRING** $\times$ **CHAR**	$\rightarrow$ **STRING** `(einfügen)`
`k:`	**STRING**	$\rightarrow$ **CHAR** `(Kopf-Zeichen)`
`r:`	**STRING**	$\rightarrow$ **STRING** `(Rest-Zeichenkette ohne Kopf-Zeichen)`
`leer:`	**STRING**	$\rightarrow$ `BOOLEAN`

und den Axiomen: $s \in$ **STRING** , $c \in$ **CHAR**

```
(A1) leer(create) := TRUE
(A2) leer(a(s,c)) := FALSE
(A3) k(a(s,c)) := IF leer(s) THEN c ELSE k(s)
(A4) r(create) := create
(A5) r(a(s,c)) := IF leer(s) THEN create ELSE a(r(s),c)
```

(a) Vereinfachen Sie unter Anwendung der Axiome die folgenden Ausdrücke so weit wie möglich: $x1, x2, x3 \in$ **CHAR**

```
r(a(create,k(a(create,x1)))) = ?
r(a(create,k(a(a(a(create,x1),x2),x3)))) = ?
a(r(a(r(create),x1)),x2) = ?
```

(b) Der Datentyp **STRING** soll durch eine weitere Operation
concat: **STRING** $\times$ **STRING** $\rightarrow$ **STRING**
erweitert werden, wobei concat zwei Zeichenketten aneinanderfügen soll.

Stellen Sie die zusätzlich erforderlichen Axiome auf (es geht auch mit einem Axiom) und erläutern Sie, warum Ihre Lösung das Gewünschte leistet.

Lösung 5. Abstrakte Datentypen

(a)
```
r(a(create,k(a(create,x1)))) =
r(a(create,x1)) =
create
```
```
r(a(create,k(a(a(a(create,x1),x2),x3)))) =
r(a(create,k(a(a(create,x1),x2)))) =
r(a(create,k(a(create,x1)))) =
r(a(create,x1)) =
create
```
```
a(r(a(r(create),x1)),x2) =
a(r(a(create,x1)),x2) =
a(create,x2)
```

(b)
```
concat(s,t) := IF leer(t) THEN s ELSE concat(a(s,k(t)), r(t))
```

oder

```
concat(s,create) := s
concat(s,t) := concat(a(s,k(t)), r(t))
```

Erläuterung: Ist der zweite String leer, ist das Ergebnis der erste String, unabhängig davon, ob dieser leer oder nicht leer ist.

Enthält der zweite String Zeichen, so wird rekursiv das jeweils erste Zeichen aus dem zweiten String entfernt und an den ersten String angehängt, bi der obere Fall eintritt.

Aufgabe 6: Programmverifikation

(2 + 9 + 1)

(a) Skizzieren Sie die wichtigsten Unterschiede zwischen WP– und Hoare–Kalkül.

(b) Gegeben sei das folgende JAVA-Programmstück. Beweisen Sie die totale Korrektheit mit Hilfe des **wp-Kalküls**. Geben Sie zuerst die Schleifeninvariante Q an.

```
int u, v, x, y, z;
{ P ≡ (x > 0 ∧ y > 0) }
z = 0;
u = x;
v = y;
while (u != 0)
{
   z = z + (v * (u % 2));
   u = u / 2;
   v = v * 2;
}
{ z = x * y }
```

(c) Geben Sie die Komplexität des obigen Programmausschnitts an (Begründung!).

Lösung 6. Programmverifikation

(a) Hoare:

- Vorwärtsanalyse, ausgehend vom Anfangszustand werden Auswirkungen des Programms berechnet
- Überprüfung, ob Resultat mit gewünschtem Endzustand übereinstimmt
- nicht zielorientiert

WP:

- Rückwärtsanalyse, geht von der Nachbedingung aus
- Anforderungen an den Anfangszustand (Vorbedingung) werden hergeleitet
- zielorientiert

(b) $P \equiv (x > 0 \wedge y > 0)$
$N \equiv (z = x * y)$
$Q \equiv (z + (u * v) = x * y)$

(1) z.z.: $P \Rightarrow wP(„z = 0, u = x, v = y“, Q)$
$wP(„z = 0, u = x, v = y“, z + (u * v) = x * y)$
$\equiv wP(„z = 0, u = x“, z + (u * y) = x * y)$
$\equiv wP(„z = 0“, z + (x * y) = x * y)$
$\equiv 0 + (x * y) = x * y \equiv TRUE$
folgt somit aus P $\square$

(2) z.z.: $Q \wedge (u \neq 0) \Rightarrow wP(„z = z + v * (u\%2), u = u/2, v = v * 2“, Q)$
$wP(„z = z + v * (u\%2), u = u/2, v = v * 2“, z + (u * v) = x * y)$
$\equiv wP(„z = z + v * (u\%2), u = u/2“, z + (2 * u * v) = x * y)$
$\equiv wP(„z = z + v * (u\%2)“, z + (2 * \frac{u}{2} * v) = x * y)$

Fall 1: u gerade
$\equiv wP(„z = z + v * (u\%2)“, z + (u * v) = x * y)$
$\equiv (z + (u * v) = x * y)$ folgt aus Q ☑

Fall 2: u ungerade
$\equiv wP(„z = z + v * (u\%2)“, z + (u - 1) * v = x * y)$
$\equiv (z + v + (u - 1) * v = x * y)$
$\equiv (z + (u * v) = x * y)$ folgt aus Q ☑

(3) z.z.: $Q \wedge (u = 0) \Rightarrow N$
$(z + (u * v) = x * y \wedge u = 0) \Rightarrow z = x * y$
klar $\square$

(4) Terminiertheit: u wird mit x initialisiert (lt. Vorbedingung >0) und pro Schleifendurchlauf jeweils halbiert. Mithin ist u irgendwann gleich 0, entsprechend terminiert die Schleife.

(c) $O(ln\, n)$, da u pro Schleifendurchlauf halbiert wird.

Aufgabe 7: Java-Programmierung

(3 +5)

Betrachten Sie die folgenden zwei Java Klassen.

```
class X {                          |   void swap2 () {
  double x = 3.33;                 |     super.t = super.x;
  int    y = 4;                    |     super.x = this.x;
  short  z = -42;                  |     x = (short)super.t;
  double t;                        |     super.swap();
                                   |     swap();
  void swap () {                   |   }
    t = x;                         |
    x = y;                         |   void scramble(double z, int x, short y) {
    y = z;                         |     z = z + x * this.x;
    z = (short)t;                  |     x = x - super.y;
  }                                |     super.y = (short)-y;
}                                  | (*)
                                   |     swap();
class Y extends X {                | (*)
  double y = 24.88;                |     super.swap();
  int    z = -6;                   | (*)
  short  x = 2;                    |     this.swap2();
  short  t;                        | (*)
                                   |   }
  void swap () {                   | }
    t = x;                         |
    x = z;                         |
    z = y;                         |
    y = t;                         |
  }                                |
```

(a) In der Methode `Y.swap` fehlen die notwendigen Typkonvertierungen. Korrigieren Sie diese Methode und begründen Sie jede Änderung kurz.

(b) Geben Sie die Zustände aller Variablen (beider Klassen) bei den mit `(*)` gekennzeichneten Stellen für den Aufruf `scramble(3.1,5,(short)9)` an.

Lösung 7. Java-Programmierung

(a) Korrekte Methode `Y.swap`:

```
void swap () {
(1)    t = x;
(2)    x = (short)z;
(3)    z = (int)y;
(4)    y = t;
}
```

Erläuterung: Zeile `(1)`: Beide Var. vom gleichen Typ,
Zeile `(2)`: `short` hat eine kleinere Wertemenge als `int`,
Zeile `(3)`: `int` hat eine kleinere Wertemenge als `double`,
Zeile `(4)`: Konvertierung erfolgt automatisch.

(b) Zustandsübergangtabelle (temp Variablen nicht beachtet):

x	y	z	this.x	this.y	this.z	super.x	super.y	super.z
1	9	13.1	2	24.88	-6	3.33	-9	-42
1	9	13.1	-6	2.0	24	3.33	-9	-42
1	9	13.1	-6	2.0	24	-9.0	-42	3
1	9	13.1	24	-9.0	2	-42.0	3	-6

Aufgabe 8: Graphen

(5 + 5)

(a) Ein Graph heißt *k-färbbar*, falls man seinen Knoten k Farben so zuordnen kann, daß keine zwei Knoten derselben Farbe durch eine Kante verbunden sind. Die Frage, ob ein beliebiger Graph mit einer bestimmten Anzahl von Farben färbbar ist, wird als *Färbungsproblem* bezeichnet.
Die Lösung für ein beliebiges Färbungsproblem sei gegeben. Erarbeiten Sie eine allgemeine Lösung für das folgende Tagungsproblem, indem Sie es auf das Färbungsproblem zurückführen.

> Es findet eine Informatiker-Tagung mit einer beliebigen Anzahl von Teilnehmern statt. Auf dieser Tagung werden n Vorträge gehalten, wobei aus Zeitersparnisgründen eine möglichst große Anzahl an Vorträgen parallel stattfinden soll. Jedem Teilnehmer wird aber das Recht eingeräumt, die Vorträge seiner Wahl zu besuchen. Möchte ein Teilnehmer zwei Vorträge hören, können diese Vorträge nicht parallel stattfinden. Jeder Vortrag dauert exakt eine Stunde, Pausen sind - da Informatiker wenig Schlaf brauchen - nicht vorgesehen. Gesucht ist die minimale Anzahl an Stunden, die die Tagung dauern muß, um die Wünsche aller Teilnehmer zu erfüllen.

(b) Ein *hamiltonscher Pfad* in einem Graphen $G = (V, E)$ ist ein Pfad, der jeden Knoten $v \in V$ genau einmal enthält. Geben Sie ein hinreichendes Kriterium dafür an, daß ein gerichteter, azyklischer Graph einen hamiltonschen Pfad enthält. Begründen Sie Ihre Angabe.

Hinweis: Topologisches Sortieren.

Lösung 8. Graphen

(a) Konstruiere einen Graph G mit den Vorträgen als Knoten. Zwei Knoten werden durch eine Kante verbunden, wenn es mindestens einen Teilnehmr gibt, der beide Vorträge hören will. Die chromatische Zahl von G (minimale Anzahl der Farben, die benötigt wird, um G zu färben), ist gleich der Anzahl der Stunden, die die Tagung dauert.

(b) Ein gerichteter, azyklischer Graph enthält genau dann einen hamiltonschen Pfad, wenn er genau eine topologische Sortierung besitzt.

Begründung: Aus VL bekannt, daß jeder azyklische Graph G mindestens eine topologische Sortierung besitzt. Besitzt G mehrere topologische Sortierungen, gibt es in jeder topologischen Sortierung $S = (e_1, \ldots, e_n)$ mindestens zwei Knoten e_i, e_{i+1}, die nicht durch eine Kante verbunden sind. Damit kann G aber keinen hamiltonschen Pfad enthalten, da kein Weg von e_i zu e_{i+1} führt. Hieraus folgt, daß jeder azyklische Graph, der einen hamiltonschen Pfad enthält, nur eine topologische Sortierung besitzen kann.