Klausur Programmierung WS 2013/2014

Vorname: _____

Nachname: _____

Matrikelnummer: _____

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

- ◦ Informatik Bachelor
- ◦ Informatik Lehramt (Bachelor)
- ◦ Sonstiges: _____
- ◦ Mathematik Bachelor
- ◦ Informatik Lehramt (Staatsexamen)

	Anzahl Punkte	**Erreichte Punkte**
Aufgabe 1	**15**	
Aufgabe 2	**15**	
Aufgabe 3	**30**	
Aufgabe 4	**20**	
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 (Antwortbereiche sind durch Kästen markiert).
- 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 und der Täuschungsversuch gemeldet, was eine Exmatrikulation und strafrechtliche Verfolgung nach sich ziehen kann.
- Geben Sie am Ende der Klausur **alle Blätter zusammen mit den Aufgabenblättern ab**.
- Halten Sie bitte Ihren Studierendenausweis und einen Lichtbildausweis zur Kontrolle bereit (die BlueCard gilt innerhalb der RWTH als beides).
- Die Bearbeitungszeit für diese Klausur beträgt zwei Stunden.

Unterschrift: _____

Aufgabe 1 **(10 + 5 = 15 Punkte)**

```
static void func(int a[ ]) {
    for (int i = 1; i < a.length; i++) {
        int j = i;
        int t = a[j];
        while (j > 0 && a[j − 1] < t) {
            a[j] = a[j − 1];
            j−−;
        }
        a[j] = t;
        // Ausgabe
    }
}
```

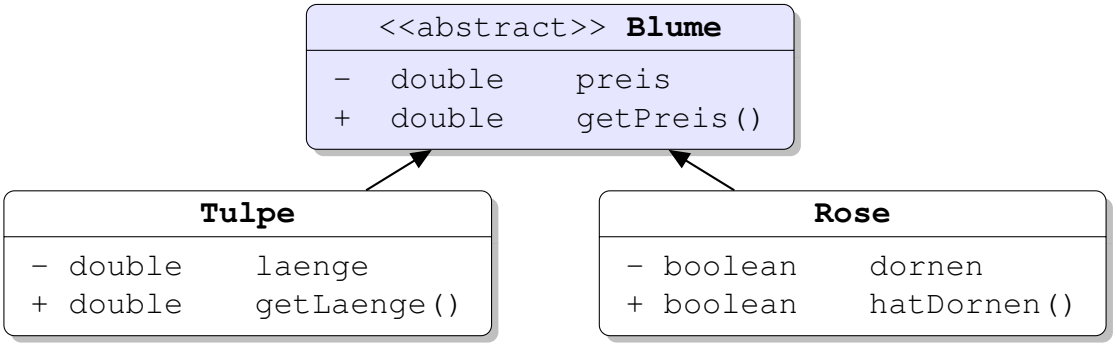
a) Die Methode *func* wird mit dem Argument $[7, 5, 2, 4, 1, 3]$ aufgerufen. Bestimmen Sie für jeden Durchlauf der äußeren Schleife die Werte der Variablen i und a an der mit "Ausgabe" markierten Stelle im Quellcode und füllen Sie die unten stehende Tabelle aus.

i	a

b) Was müssten Sie an den obigen Programm ändern, damit das Array a nach dem Aufruf aufsteigend sortiert ist? Ihre Änderung soll keine weiteren Programmzeilen hinzufügen!

Aufgabe 2 **(15 Punkte)**

Betrachten Sie folgendes Klassendiagramm. Die Sichtbarkeiten + und - bedeuten `public` und `private`. Die Methoden in den Klassen sind Getter für die entsprechenden Attribute.



Implementieren Sie die untenstehende Methode in der Klasse `Blume`, welche den Gesamtpreis aller im übergebenen Array enthaltenen Rosen ohne Dornen berechnet. Sollten keine Rosen ohne Dornen in diesem Array vorhanden sein, soll die Methode `0.0` zurückgeben.

Hinweis:
Mit **if**(o **instanceof** T) {...} können Sie testen, ob das Objekt o vom Typ T ist.

```
public static double preisDornloseRosen(Blume[ ] blumen) {




}
```

Aufgabe 3 **(8 + 4 + 18 = 30 Punkte)**

Im Folgenden sind einige Java-Klassen gegeben. Darunter finden Sie auch die Klasse *TreeSet*, die eine Menge von **int**s als geordneten Binärbaum implementiert.

```
class Node {
  int val; // der Wert dieses Knotens
  Node left; // das linke Kind dieses Knotens
  Node right; // das rechte Kind dieses Knotens
  Node parent; // der Vater dieses Knotens

  boolean contains(int v) {/* TODO */}

  /* Findet das Minimum dieses Teilbaumes. */
  Node min() {
    if (this.left == null) {
      return this;
    } else {
      return this.left.min();
    }
  }
  ...
}

/* Implementierung einer Menge als sortierter Binärbaum. */
class TreeSet implements Iterable<Integer> {
  /* Wurzel des Baumes. */
  private Node root = null;

  /* Testet, ob die gegebene Zahl in der Menge enthalten ist. */
  boolean contains(int v) {
    if (this.root != null) {
      return this.root.contains(v);
    } else {
      return false;
    }
  }

  public Iterator<Integer> iterator() {
    return new TreeSetIterator(this.root);
  }
  ...
}
```

```
class TreeSetIterator implements Iterator⟨Integer⟩ {
  /* Jenes Element, dessen val der nächste Aufruf von next zurückliefert. */
  private Node n = null;

  TreeSetIterator(Node root) {
    if (root != null) {
      this.n = root.min();
    }
  }

  public boolean hasNext() {
    return this.n != null;
  }

  public Integer next() throws NoSuchElementException {
    if (this.n == null) {/* TODO */}
    int res = this.n.val;
    forward();
    return res;
  }

  void forward() {/* TODO */}
  ...
}
```

a) Implementieren Sie die Methode *Node.contains*. Diese soll überprüfen, ob die als Argument übergebene Zahl in dem Teilbaum, auf dem die Funktion aufgerufen wird, enthalten ist.

b) Vervollständigen Sie die Methode *TreeSetIterator.next* an der mit TODO gekennzeichneten Stelle.

Die Klasse *TreeSetIterator* bietet die Möglichkeit, in aufsteigend sortierter Reihenfolge über alle Elemente eines *TreeSet* zu iterieren. Die Methode *TreeSetIterator.next*

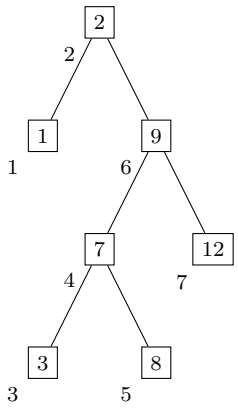


Abbildung 1: Reihenfolge, in der durch einen Baum iteriert werden soll

hat die Aufgabe, *n.val* zurückzugeben. Darüber hinaus ruft *TreeSetIterator.next* die Hilfsmethode *TreeSetIterator.foward* auf. Die Aufgabe dieser Hilfsmethode ist es, das Feld *n* so zu aktualisieren, dass es jenen *Node* enthält, dessen *val* beim nächsten Aufruf von *TreeSetIterator.next* zurückgegeben werden muß. Wenn das letzte Element erreicht wurde, muß *n* auf den Wert *null* gesetzt werden.

c) Implementieren Sie die Methode *TreeSetIterator.forward*. Die Aufgabe dieser Methode wird in Teilaufgabe b) beschrieben.

In Abbildung 1 ist die Reihenfolge, in der über die Knoten des dargestellten Baumes iteriert werden soll, in Form der Zahlen unten links neben den Knoten beispielhaft dargestellt.

Hinweise:

- Um das nächste Element, das der Iterator zurückgeben soll, zu finden, können Sie wie folgt vorgehen: Falls das aktuelle Element ein rechtes Kind hat, suchen Sie nach dem Minimum des durch dieses Kind definierten Teilbaums. Andernfalls suchen Sie unter den Vorgängern (d.h., unter jenen Knoten, die erreicht werden können, indem man mindestens einmal dem Feld *parent* folgt) des aktuellen Elements nach einem geeigneten Element.
- Bedenken Sie, daß die Klasse *TreeSet* eine *Menge* implementiert. Das heißt, daß ein *TreeSet* jedes Element *höchstens einmal* enthalten kann.
- Die Methode *Iterator.next* wirft eine *NoSuchElementException*, wenn bereits über alle Elemente iteriert wurde, sodaß keine weiteren Elemente mehr zur Verfügung stehen.

```
boolean contains(int v) {




}
```

```
public Integer next() throws NoSuchElementException {
    if (this.n == null) {
      // TODO




    }
    int res = this.n.val;
    forward();
    return res;
  }
```

```
void forward() {




}
```

Aufgabe 4 **(15 + 5 = 20 Punkte)**

Gegeben sei das folgende Java-Programm P, das zu einer Eingabe $n \geq 0$ den Wert $\frac{1}{6}n(n+1)(2n+1)$ berechnet.

$\langle\ n \geq 0\ \rangle$ (Vorbedingung)
  $i = 0;$
  $res = 0;$
  while $(i < n)$ {
    $i = i + 1;$
    $res = res + i * i;$
  }
$\langle\ res = \frac{1}{6}n(n+1)(2n+1)\ \rangle$ (Nachbedingung)

a) Vervollständigen Sie die Verifikation des Algorithmus P auf der nachfolgenden 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:

- Es gilt $\frac{1}{6}i(i+1)(2i+1) + (i+1)^2 = \frac{1}{6}(i+1)(i+2)(2(i+1)+1)$ für alle $i \in \mathbb{Z}$.
- Sie dürfen beliebig viele Zusicherungs-Zeilen ergänzen oder streichen. In unserem Lösungsvorschlag wird allerdings genau die angegebene Anzahl an 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. Beachten Sie, daß Sie bei der Anwendung der Zuweisungsregel eventuell Klammern setzen müssen. Zum Beispiel:
  $\langle (i+1)^2 = k \rangle$
  $i = i + 1;$
  $\langle i^2 = k \rangle$

Matrikelnummer: **Name:**

$\langle n \geq 0 \rangle$

$\langle$ _____ $\rangle$

```
i = 0;
```

$\langle$ _____ $\rangle$

```
res = 0;
```

$\langle$ _____ $\rangle$

$\langle$ _____ $\rangle$

```
while (i < n){
```

$\langle$ _____ $\rangle$

$\langle$ _____ $\rangle$

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

$\langle$ _____ $\rangle$

```
   res = res + i * i;
```

$\langle$ _____ $\rangle$

```
}
```

$\langle$ _____ $\rangle$

$\langle res = \frac{1}{6}n(n+1)(2n+1) \rangle$

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

Matrikelnummer: **Name:**

Aufgabe 5 **(4 + 6 + 10 = 20 Punkte)**

a) Schreiben Sie eine Haskell-Funktion *onlyPositives* :: $[Int] \rightarrow [Int]$, welche als Argument eine Liste l bekommt. Das Ergebnis von *onlyPositives* ist jene Liste, die man erhält, wenn man alle nicht-positiven Elemente aus der Liste l entfernt.

Beispiel: *onlyPositives* $[-3, 2, 5, 0, -1, 7] = [2, 5, 7]$

b) Gegeben ist folgendes Haskell-Programm:

mystery = *foldl* $(\backslash x\ y \rightarrow y : x)$ [ ]

Was ist das Ergebnis von *mystery* $[1, 2, 3, 4, 5, 6]$?

c) Schreiben Sie eine Haskell-Funktion *prefixsum*, welche zu einer Liste von *Int*s eine Liste ihrer Präfixsummen berechnet. Das heißt, das n-te Element des Ergebnisses soll die Summe der ersten n Elemente der Liste sein.

Beispiel: *prefixsum* $[3, 2, 5, 0, 3] \; = \; [3, 5, 10, 10, 13]$.

Aufgabe 6 **(4 + 6 + 10 = 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) $f(X, g(Y, Y), a), f(g(Z, Z), Z, Y)$

ii) $f(g(X, Y), Z, X), f(Z, g(a, b), Y)$

iii) $f(X, Y, Z), f(Y, Z, g(X, Y))$

iv) $f(g(X, Y), X, Z), f(g(Z, Z), Y, a)$

b) Gegeben sei das folgende Prolog-Programm.

```
riddle([],[]).
riddle(XS,[X|YS]) :- something(XS,X,ZS), riddle(ZS,YS).

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

Geben Sie alle Antwortsubstitutionen zur Anfrage

```
?- riddle([1,2],A).
```

in der Reihenfolge an, in der Prolog sie findet. Sie brauchen **nicht** die einzelnen Auswertungsschritte anzugeben (auch wenn Sie das als Nebenrechnung natürlich dürfen - ein schnellerer Weg zur Lösung besteht allerdings darin, ein intuitives Verständnis für das Programm zu entwickeln, sodaß man das Ergebnis direkt angeben kann).

c) Gegeben sei ein Prädikat `add/3` in Prolog, sodaß für den Fall, daß `XS` eine aufsteigend sortierte Liste ist, `add(X,XS,YS)` genau dann wahr ist, wenn `YS` aus `XS` entsteht, indem `X` sortiert eingefügt wird. Es gilt zum Beispiel `add(3,[1,2,5],[1,2,3,5])` und `add(1,[1,2],[1,1,2])`. Nutzen Sie dieses Prädikat, um ein weiteres Prädikat `sort/2` in Prolog zu implementieren, sodaß `sort(XS,YS)` genau dann wahr ist, wenn `XS` und `YS` Listen sind, welche genau die gleichen Elemente enthalten und `YS` aufsteigend sortiert ist. Es gilt also z. B. `sort([1,3,2,1],[1,1,2,3])`, aber nicht `sort([1,1,2],[1,2])`. Benutzen Sie **keine** vordefinierten Prädikate außer `add/3` (insbesondere gehören Prädikate wie `=/2` zu den vordefinierten Prädikaten).

Hinweise zur Haskell-API:

- `concat :: [[a]] -> [a]`:
  Konkateniert eine Liste von Listen, etwa wird $[[1,2,3],[4,5],[6,7,8]]$ zu $[1,2,3,4,5,6,7,8]$.

- `foldl :: (a -> b -> a) -> a -> [b] -> a`:
  Der Aufruf `foldl f a [b1,b2,...,bn]` entspricht dem Aufruf `f (... (f (f (f a b1) b2) b3 )... ) bn`, d. h. `f` wird sukzessive auf die Element der Liste aufgerufen, wobei das Ergebnis des vorherigen Aufrufs als erster Parameter übergeben wird.

- `foldr :: (a -> b -> b) -> b -> [a] -> b`
  Die Funktion `foldr` arbeitet wie `foldl`, arbeitet die Liste aber von rechts nach links statt von links nach rechts ab.

- `map :: (a -> b) -> [a] -> [b]`:
  Der Aufruf `map f [a1,a2,...]` ergibt die Liste `[f a1,f a2,...]`.

- `filter :: (a -> Bool) -> [a] -> [a]`:
  Der Aufruf `filter f [a1,a2,...]` liefert eine Liste derjenigen Elemente `ai`, für welche `f ai` wahr ist.

- `zip :: [a] -> [b] -> [(a,b)]`
  Die Funktion zip bildet aus zwei Listen eine Liste von Paaren. Wenn eine Liste länger ist als die andere, werden überschüssige Elemente ignoriert.
  Beispiel: `zip [1,2,3] [2,3,4,5] = [(1,2),(2,3),(3,4)]`.

- `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`
  zipWith verallgemeinert zip in der Hinsicht, dass keine Paare erzeugt werden sondern die zwei Elemente aus den jeweiligen Listen durch eine gegebene Funktion abgebildet werden. So erzeugt z.B. `zipWith (+)` eine Liste von Summen und `zipWith (\x y -> (x,y))` entspricht schlicht zip.

- `init :: [a] -> [a]`
  Die Funktion `init` liefert eine Liste ohne ihr letztes Element zurück.

- `last :: [a] -> a`
  Die Funktion `last` liefert das letzte Element einer Liste zurück.