**RWTH Aachen** **WS 2013/14**
**Lehrgebiet Theoretische Informatik** Übungsblatt 4
Frohn–Reidl–Rossmanith–Sánchez–Ströder 18.11.2013

# Übung zur Vorlesung Programmierung

## Aufgabe T8

Ein *Stack* ist eine Datenstruktur, die stets nur Zugriff auf das zuletzt eingefügte Element ermöglicht. Folglich stellt ein Stack in der Regel Funktionen zum Ablegen eines Elementes (*push*) und zum Zugriff auf das zuletzt abgelegte Element (*pop*, *peek*) zur Verfügung.
Implementieren Sie eine Klasse `Stack`, die eine veränderbare[1] Implementierung der Datenstruktur Stack für nicht negative ganze Zahlen mit folgenden Methoden zur Verfügung stellt:

- **void** *push*(**int** $x$): Legt das Element $x$ auf dem Stack ab. Wenn $x$ negativ ist, dann bleibt der Stack unverändert.
- **int** *pop*(): Entfernt das zuletzt abgelegte Element vom Stack und liefert dieses zurück. Falls der Stack leer ist, wird $-1$ zurückgegeben.
- **int** *peek*(): Liefert das zuletzt abgelegte Element zurück, ohne es zu entfernen. Falls der Stack leer ist, wird $-1$ zurückgegeben.

Eine *Queue* ist eine Datenstruktur, die stets nur Zugriff auf jenes Element ermöglicht, das am längsten Teil der Queue ist. Folglich stellt eine Queue in der Regel Funktionen zum Einfügen eines neuen Elements (*enqueue*) und zum Zugriff auf das älteste Element (*dequeue*) zur Verfügung.
Implementieren Sie eine Klasse `Queue`, die eine veränderbare Implementierung der Datenstruktur Queue für nicht negative ganze Zahlen mit folgenden Methoden zur Verfügung stellt:

- **void** *enqueue*(**int** $x$): Fügt das Element $x$ zu der Queue hinzu. Wenn $x$ negativ ist, dann bleibt die Queue unverändert.
- **int** *dequeue*(): Entfernt das älteste Element aus der Queue und liefert dieses zurück. Falls die Queue leer ist, wird -1 zurückgegeben.

## Aufgabe T9

Gegeben sei folgendes Java-Programm $P$:

$\langle\varphi\rangle$ (Vorbedingung)

```
r = 1;
while (r * r < x){
  r = r + 1;
}
```

$\langle\psi\rangle$ (Nachbedingung)

[1] Das heißt Methoden, die den Stack verändern, manipulieren die `Stack`-Instanz, auf der die Methode aufgerufen wurde, statt neue Instanzen zu erzeugen.

a) Als Vorbedingung $\varphi$ für das oben aufgeführte Programm $P$ gelte $\texttt{x} > 0$ und als Nachbedingung $\psi$ gelte[2] $\texttt{r} = \lceil\sqrt{\texttt{x}}\,\rceil$. Vervollständigen Sie die folgende Verifikation des Algorithmus, indem Sie die leeren Zeilen durch korrekte Zusicherungen entsprechend des Hoare-Kalküls 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.

$\langle \texttt{x} > 0 \rangle$

$\langle$_____$\rangle$

```
r = 1;
```

$\langle$_____$\rangle$

$\langle$_____$\rangle$

```
while (r * r < x){
```

$\langle$_____$\rangle$

$\langle$_____$\rangle$

```
   r = r + 1;
```

$\langle$_____$\rangle$

```
}
```

$\langle$_____$\rangle$

$\langle \texttt{r} = \lceil\sqrt{\texttt{x}}\,\rceil \rangle$

*Hinweise:*

- 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.

b) Untersuchen Sie den Algorithmus $P$ auf seine Terminierung. Für einen Beweis der Terminierung muss eine Variante angegeben werden und mit Hilfe des Hoare-Kalküls die Terminierung bewiesen werden.

*Hinweis: Sie dürfen für den Terminierungsbeweis zusätzlich zur Schleifenbedingung auch* $\texttt{r} > 0$ *zu Beginn jedes Schleifendurchlaufs annehmen.*

[2] hierbei ist $\lceil x \rceil$ die ganzzahlige Aufrundung von $x$; siehe auch `http://de.wikipedia.org/wiki/Abrundungsfunktion_und_Aufrundungsfunktion`

**Aufgabe H8 (8+2 Punkte)**

Gegeben sei folgendes Java-Programm $P$:

$\langle\varphi\rangle$ (Vorbedingung)

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

$\langle\psi\rangle$ (Nachbedingung)

a) Als Vorbedingung $\varphi$ für das oben aufgeführte Programm $P$ gelte $\texttt{n} \geq 0$ und als Nachbedingung $\psi$ gelte $\texttt{res} = (\frac{\texttt{n}\cdot(\texttt{n}+1)}{2})^2$. Vervollständigen Sie die folgende Verifikation des Algorithmus, indem Sie die leeren Zeilen durch korrekte Zusicherungen entsprechend des Hoare-Kalküls 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.

$\langle \texttt{n} \geq 0 \rangle$

$\langle$_____$\rangle$

```
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 * i;
```

$\langle$_____$\rangle$

```
}
```

$\langle$_____$\rangle$

$\langle \texttt{res} = (\frac{\texttt{n}\cdot(\texttt{n}+1)}{2})^2 \rangle$

*Hinweise:*

- 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.
- Es gilt $(\frac{x\cdot(x+1)}{2})^2 + (x+1)^3 = (\frac{(x+1)\cdot(x+2)}{2})^2$ für alle $x \in \mathbb{R}$.

b) Untersuchen Sie den Algorithmus $P$ auf seine Terminierung. Für einen Beweis der Terminierung muss eine Variante angegeben werden und mit Hilfe des Hoare-Kalküls die Terminierung bewiesen werden.

**Aufgabe H9 (5+5 Punkte)**[3]

Erweitern sie die Klasses *Set* aus Aufgabe T7 indem sie zu der Klasse die Methoden *intersection*(*Set other*) und *union*(*Set other*) hinzufügen. Die Methode *intersection*(*Set other*) soll ein *Set* zurückgeben was nur die Elemente enthält, die sowohl in der Menge sind worauf die Methode aufgerufen wurde, als auch in der Menge *other*. Die Methode *union*(*Set other*) soll ein *Set* zurückgeben was alle Elemente aus der Menge worauf die Methode aufgerufen wuerde enthält, als auch die Element aus der Menge *other*. Dabei sollen die Attribute jeder Instanz der Klasse *Set*, wie davor, unveränderbar sein, nachdem sie im Konstruktor gesetzt wurden. Eine leere Menge kann dabei auch durch den Wert *null* representiert werden.

## Abgabe zum 26.11.2013

[3]Bitte Quelltext für die Abgabe ausdrucken und zusätzlich per E-mail an den jeweiligen Tutor senden.