Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Einführung in die Informatik
Sommersemester 2016

Prof. Dr. Wolfram Burgard
Alexander Schiotka
Andreas Kuhner

Übungsblatt 13

Abgabe / Besprechung in Absprache mit dem Tutor

Hinweis:

Aufgaben immer per E-Mail (eine E-Mail pro Blatt und Gruppe) an den zuständigen Tutor schicken (Bei Programmieraufgaben Java Quellcode und evtl. benötigte Datendateien).

Aufgabe 13.1

Wie lautet die Ausgabe des folgenden Java-Programms?

```
class A131 {
  public static void main(String arg[]) {
    int i1 = 2, i2 = 1;
    System.out.println( i1++ );
    int i3 = i2 * (++i1);
    System.out.println( i3 );
    System.out.println( i2++ );
    i3 /= i1++;
    System.out.println( i1 );
    i1 *= i2;
    System.out.println( i1 );
    System.out.println( i1 * i2 + i2 * i3);
  }
}
```

Aufgabe 13.2

1. Bestimmen Sie die Laufzeiten der folgenden Programmstücke in Abhängigkeit von `n`. Begründen Sie jeweils Ihre Antwort.

   (a)
   ```
   int c = 0;
   for (int i = 0; i < n; ++i) {
     for(int j = 0; j < i / 2; ++j) {
       c++;
     }
   }
   ```

   (b)
   ```
   int c = 0;
   for (int i = 0; i < n; ++i) {
     for(int j = 0; j < n; ++j) {
       for(int k = 0; k < j; ++k) {
         c++;
       }
   ```

```
    }
  }
```

(c)
```
int c = 0;
for (int i = 1; i < n; i *= 2) {
  c++;
}
```

(d)
```
int fibonacci(int n) {
  if(n == 1 || n == 2) {
    return 1;
  } else {
    return fibonacci(n - 1) + fibonacci(n - 2);
  }
}
```

2. Die Funktionen `f` und `g` sind für positive Integer-Eingabewerte definiert und haben folgende Laufzeiten in Abhängigkeit des Eingabewertes:

	Best-Case	Worst-Case
f(n)	$O(1)$	$O(n^3)$
g(n)	$O(n^2)$	$O(n^3)$

Führen Sie eine Best- und Worst-Case Abschätzung für die folgende Methode in Abhängigkeit von der Länge `len` des Arrays durch. Begründen Sie Ihre Antwort.

```
void method(int[] array) {
  int len = array.length;
  for (int i = 0; i < len; i++) {
    int k = f(len);
    if(array[i] == k) {
      g(len);
    } else {
      g(2);
    }
  }
}
```

Aufgabe 13.3

Die Zahl π kann durch die so genannte Gregory-Leibniz-Reihe

$$\pi = \sum_{n=0}^{\infty} (-1)^n \frac{4}{2n+1}$$

angenähert werden. Eine obere Schranke für den Fehler der Approximation nach n Summanden ist gegeben durch:

$$|R(n)| \leq \frac{4}{2n+1}$$

1. Implementieren Sie eine Methode, die die Zahl π approximiert und die Approximation zurückliefert. Die gewünschte Genauigkeit (obere Schranke) soll als Parameter an die Funktion übergeben werden.

2. Schreiben Sie für Ihre Implementierung eine Testmethode mittels `JUnit`, die testet, ob die Zahl π bis auf die gewünschte Genauigkeit approximiert wird. Überprüfen Sie dafür die Genauigkeit der Approximation für $|R(n)| \leq 0.0001$ und $|R(n)| \leq 0.0000001$. Hinweis: Verwenden Sie `Math.PI` als Referenz für π.

Aufgabe 13.4

Gegeben sei folgendes Java-Programm:

```
public static int f(int a, int b) {
  if(b == 0) {
    return 1;                   //Zeile 1
  }
  if ((b % 2) == 0) {
    return f (a * a, b / 2);    //Zeile 4
  }
  return a*f (a, b - 1);        //Zeile 6
}
```

1. Was wird bei dem Aufruf $f(2,4)$ zurückgegeben?

2. Zeichnen Sie die Activation Records für den Aufruf $f(2,4)$ zu dem Zeitpunkt, an dem die maximale Rekursionstiefe erreicht ist.

Aufgabe 13.5

Betrachten Sie die folgende Klassendefinition:

```
class Margherita {
    Margherita(){
        price = 5.0;
        topping = "Tomate, Kaese";
    }
    public Margherita makeLarger(){
        price += 3.0;
        return this;
    }
    public String toString() {
        return "Preis: " + price + "; Belag: " + topping;
    }
    protected double price;
    protected String topping;
}
class Salami extends Margherita {
    public Salami() {
        super();
        topping = topping + ", Salami";
        price -= 1.0;
```

```
    }
    public String toString() {                          // Zeile 22
        return "Sonder-" + super.toString();
    }
}
class Mozzarella extends Margherita {
    public Mozzarella() {
        super();
        topping = "Tomate, Mozzarella";
        price = price + 2.5;
    }
}
public class Pizza {
    public static void main(String[] args) {
        Margherita p1 = new Margherita();
        Margherita p2 = new Mozzarella();
        Margherita p3 = p2.makeLarger();
        System.out.println(new Salami());
        System.out.println(p1.makeLarger());
        System.out.println(p2);
        System.out.println(p3);
    }
}
```

1. Welche Ausgabe liefert die `main`-Methode der Klasse `Pizza`?
2. Was bewirkt das Schlüsselwort `protected` bei der Deklaration der Instanzvariablen `price` und `topping`?
3. Was würde hingegen das Schlüsselwort `private` anstelle von `protected` bewirken?
4. Was würde hingegen das Schlüsselwort `public` anstelle von `protected` bewirken?
5. Handelt es sich bei der `toString`-Methode in Zeile 22 um Overloading oder Overriding?

Aufgabe 13.6

Eine Warteschlange (engl. „Queue“) ist eine Datenstruktur, die Objekte nach dem FIFO-Prinzip (First In – First Out) verwaltet, d.h. dass Objekte genau in der Reihenfolge ausgegeben werden, in der sie in die Warteschlange eingefügt werden. In dieser Aufgabe soll eine Warteschlange mit einer einfach verketteten Liste implementiert werden.

```
public class Queue {
  public Queue() {...}
  public void push(Object o) {...}
  public Object pop() {...}
  ...
  protected QueueNode head;
  protected QueueNode tail;
}

public class QueueNode {
  ...
  private Object content;
  ...
}
```

1. Geben Sie die vollständige Klassendefinition eines Speicherelements `QueueNode` mit geeigneten Zugriffsmethoden sowie typischem Konstruktor an und implementieren Sie diese.

2. Implementieren Sie den Konstruktor sowie die Methoden `push()` und `pop()` der Klasse `Queue`. Die Methode `push()` fügt dem Speicher ein Element hinzu. Die Methode `pop()` liefert das nächste abzuarbeitende Element zurück und löscht dieses aus der Warteschlange.

3. Geben Sie den Aufwand für ihre Methoden in Abhängigkeit der Anzahl an gespeicherten Objekten an.

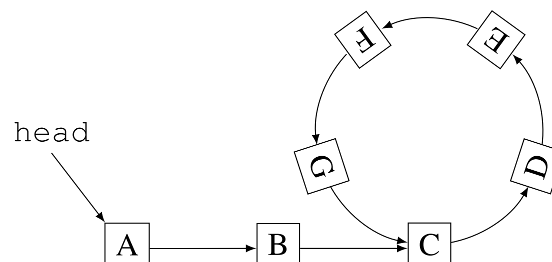
Aufgabe 13.7

Betrachten Sie den folgenden Ausschnitt einer Klassendefinition für eine einfach verkettete Liste.

```
class SingleLinkedList {
    public SingleLinkedList() { head = null; }
    public Node searchNode(Object o) {
        Node n = head;
        while (n != null && !n.content().equals(o))
            n = n.nextNode();
        return n;
    }
    public boolean hasCycle() { ... }
    public Node head;
}

class Node {
    public Node (Object o, Node n) {
        content = o;
        nextNode = n;
    }
    public Object content() { ... }
    public void setContent(Object o) { ... }
    public Node nextNode() { ... }
    public void setNextNode(Node n) { ... }
    Object content;
    Node nextNode;
}
```

Betrachten Sie folgendes Beispiel für eine Liste, die einen Zyklus enthält:



1. Wie verhält sich die Methode `searchNode(Object o)` der Klasse `SingleLinkedList`, falls sich ein Zyklus in der Liste befindet?

2. Implementieren Sie die Methode `hasCycle()` der Klasse `SingleLinkedList`. Diese Methode soll genau dann `true` liefern, falls ein Zyklus in der Liste ist. Gehen Sie dabei folgendermaßen vor: Erzeugen Sie zwei Referenzen auf den Kopf der Liste. Erstellen Sie eine Schleife, die pro Durchlauf die eine Referenzvariable um ein Element bzw. die andere Referenzvariable um zwei Elemente weiterbewegt. Führen Sie anschließend folgende Abfragen durch. Falls eine der beiden Referenzen das Ende der Liste erreicht, so ist kein Zyklus enthalten. Falls beide Referenzen auf das gleiche Element der Liste zeigen, ist ein Zyklus vorhanden.

Aufgabe 13.8

Entscheiden Sie, ob die folgenden Aussagen richtig oder falsch sind.

	Richtig	Falsch
Die Menge der Algorithmen ist abzählbar.	☐	☐
Jede Funktion lässt sich durch einen Algorithmus realisieren.	☐	☐
Jeder Algorithmus beschreibt eine Funktion.	☐	☐
Die Frage, ob zwei Programme auf die gleiche Eingabe stets die gleiche Ausgabe liefern, ist entscheidbar.	☐	☐
Ein partiell korrektes Programm kann nie ein falsches Ergebnis zurückgeben.	☐	☐
Es gibt Programme, für die gezeigt werden kann, dass sie total korrekt sind.	☐	☐