Martin Götze, Bernhard Streit

8. Übung zu Informatik B

Tutorium: Marco Block, Fr 12–14 Uhr

1. **Länder & anderes**

Betrachten Sie die folgenden Java-Klassen. Was ist der Output eines Aufrufes der main-Methode in der Klasse Maryland, begründen Sie ihre Antwort!

```
1  public class Place extends Object {
2      Place() { /* null constructor */ }
3      public void printMe() { System.out.println("Buy␣it."); }
4  }
5  public class Region extends Place {
6      Region() { /* null constructor */ }
7      public void printMe() { System.out.println("Box␣it."); }
8  }
9  public class State extends Region {
10     State() { /* null constructor */ }
11     public void printMe() { System.out.println("Ship␣it."); }
12 }
13 public class Maryland extends State {
14     Maryland() { /* null constructor */ }
15     public void printMe() { System.out.println("Read␣it."); }
16     public static void main(String[] args) {
17         Region mid = new State();
18         State md = new Maryland();
19         Object obj = new Place();
20         Place usa = new Region();
21         md.printMe();
22         mid.printMe();
23         ((Place) obj).printMe();
24         obj = md;
25         ((Maryland) obj).printMe();
26         obj = usa;
27         ((Place) obj).printMe();
28     }
29 }
```

Es ergibt sich folgenden Ausgabe nach einem Aufruf der main-Mehtode der Klasse Maryland:

```
Read it.
Ship it.
Buy it.
Read it.
Box it.
```

Bergündung:

Ausgabe in Zeile 21: Hier wird md.printMe() aufgerufen, da md eine Referenz auf eine Maryland-Instanz ist (bei überschriebenen Methoden entscheidet der Typ der Instanz), wird „Read it.“ ausgegeben.

Ausgabe in Zeile 22: Hier wird mid.printMe() aufgerufen, da mid eine Referenz auf eine State-Instanz ist (bei überschriebenen Methoden entscheidet der Typ der Instanz), wird „Ship it.“ ausgegeben.

Ausgabe in Zeile 23: Hier wird ((Place) obj).printMe() aufgerufen, da obj eine Referenz auf eine Place-Instanz ist (bei überschriebenen Methoden entscheidet der Typ der Instanz), wird „Buy it.“ ausgegeben.

Ausgabe in Zeile 25: Hier wird ((Maryland) obj).printMe() aufgerufen, da `obj = md` eine Referenz auf eine Maryland-Instanz ist (bei überschriebenen Methoden entscheidet der Typ der Instanz), wird „Read it.“ ausgegeben.

Ausgabe in Zeile 27: Hier wird ((Place) obj).printMe() aufgerufen, da ja jetzt `obj = usa` eine Referenz auf eine Region-Instanz ist (bei überschriebenen Methoden entscheidet der Typ der Instanz), wird „Box it.“ ausgegeben.

2. **Stapel als Schlangen**

Beschreiben Sie, wie der ADT Warteschlange mit zwei Stacks implementiert werden kann. Was ist die Laufzeit der Methoden enqueue(o) und dequeue() in diesem Fall?

Zugrunde liegt folgende Idee:

Man hat zwei Stacks S_{enq} und S_{deq} zu Verfügung, beim Einfügen von Elementen in die Warteschlange werden diese einfach mit S_{enq}.push() auf den equeue-Stack (dafür steht S_{enq}) abgelegt. Soll nun das erste Element ausgegeben werden und ist S_{deq} (der dequeue-Stack) leer, so werden alle Elemente auf S_{enq} auf S_{deq} umgeschichtet, einfach der Reihe nach, nun liegt aber das zuerst eingefügte Element oben (denn es lag unten in S_{enq}, wurde also zuletzt umgelegt). Nun (oder im Falle eines nicht leeren S_{deq}) kann also mit S_{deq}.pop() das zuerst in die Queue eingefügte Element ausgegeben werden, man hat also im Pseudocode für die typischen Queue-Methoden, zuerst für enqueue und dequeue

```
Benutze die privaten Stacks enqeueStack, dequeueStack, initialisiere sie
    im Konstruktor

Funktion enqueue(Object obj)
    versuche enqueueStack.push(obj)
        falls "StackVoll"-Fehler → löse einen "QueueVoll"-Fehler aus

Funktion Object dequeue()
    Wenn dequeueStack.isEmpty()
        umschichten();// Aufruf der "Umschichtungsmethode", siehe unten.
    Wenn dequeueStack.isEmpty() // immer noch leer, also ist die
        Schlange leer
        löse einen "QueueLeer"-Fehler aus.
    Gib dequeueStack.pop() zurück

// Nun die wichtige Verschiebe-Methode

private Funktion umschichten()
    // solange verschieben, wie was da ist.
    Solange nicht enqueueStack.isEmpty()
        versuche dequeueStack.push(enqueueStack.pop())
            falls "StackVoll"-Fehler, löse einen "QueueVoll"-Fehler aus
```

Es bleiben noch die anderen Methoden des ADT Queue zu konkretiesieren, sie können aber leicht aus die Stack-Methoden zurückgeführt werden und front() geht wie dequeue():

```
Funktion int size()
    Gib enqueueStack.size() + dequeueStack.size() zurück

Funktion boolean isEmpty()
    Gib true zurück, wenn size() > 0, sonst false

Funktion Object front()
    Wenn dequeueStack.isEmpty()
        umschichten(); // Aufruf der "Umschichtungsmethode", siehe oben.
    Wenn dequeueStack.isEmpty() // immer noch leer, also ist die
        Schlange leer
```

```
        löse einen "QueueLeer"−Fehler aus.
    Gib dequeueStack.top() zurück
```

Damit ist der ADT queue mit zwei Stacks realisiert.

Es bleibt der Aufwand der en-/dequeue-Methode zu analysieren, enqueue ist offenbar nicht aufwändiger (ich erschrecke mich jedesmal ;-)) als ein push() für den Stack, anders sieht es da schon für dequeue/front aus, das kann für ein einzelnes Element sehr aufwändig sein, stehen jedoch Einfüge- und Entferneoperationen in einem normalen Verhältnis (soll heißen: werden alle in die Queue eingefügten Elemente auf wieder entfernt), so fällt aus, das für ein einzelnes Element für beide Operationen (enqueue/dequeue) zusammen genau zwei push()- und zwei pop()- Operationen benötigt werden, d.h. im Mittel sind enqueue und dequeue zusammen doppelt so teuer wie push() und pop() zusammen, können also insbesondere so im Mittel in konstanter Zeit erledigt werden.

3. **Schlangen als Stapel**

   Beschreiben Sie, wie man den ADT Stack mit zwei Queues implementieren kann. Was ist die Laufzeit der Methoden pop() und push() in diesem Fall?

   Die Grundidee ist folgende: Man verwendet die zwei Queues Q_1 und Q_2 so, dass die Schlange Q_2 nur für die pop() und top()-Operationen benötigt wird, in ihr wird das oberste Element des Stacks aufbewahrt und sie wird zum „Umdrehen“ verwendet, falls sie leer ist und man das zuletzt eingefügte Element herausholen möchte, d.h. falls zwei oder mehr pop()-s hintereinander ausgeführt werden, und so die ganze Queue Q_1 leeren muss.

   Eingefügt, ge-push()-t, dagegen werden die Elemente, nach einfügen des (falls vorhanden) Elementes von Q_2 in Q_1, einfach in die Schlange Q_2, sie stehen dort zum pop() bereit.

   Nun also zum Pseudocode der beiden Hauptmethoden push() und pop() des ADT Stack:

```
Verwende zwei private Queues queue1, queue2, die im Konstruktor leer
    erzeugt werden

Funktion push(Object obj)
    Wenn nicht queue2.isEmpty()
        versuche queue1.enqueue(queue2.dequeue())
            falls ein "QueueVoll"-Fehler auftritt, löse einen "StackVoll
                "-Fehler aus
    queue2.enqueue(obj);
    /* queue2 hat jetzt ein Element, wenn hier ein "QueueVoll"-Fehler
        auftritt, ist irgendwas
       mit der Queue schiefgelaufen. */

Funktion Obejct pop()
    Wenn queue2.isEmpty()
        holeErstes();
        // die "Umschichtung"-Methode, siehe unten
    Wenn queue2.isEmpty()
        // immer noch leer -> keine Elemente
        löse einen "StackLeer"-Fehler aus
    Gib queue2.dequeue() zurück

private Funktion holeErstes()
    // Solange schieben, bis keine Elemente mehr da sind
    Wenn nicht queue1.isEmpty()
        Verwende die Variable
        Object letztes ← queue1.dequeue()
        Solange nicht queue1.isEmpty() // letztes ist nicht das letzte
            // wieder einfügen
            queue2.enqueue(letztes);
             letztes ← queue1.dequeue();
        // das letzte wieder in Queue1:
        queue1.enqueue(letztes);
        Vertausche queue1 und queue2 mit eine Hilfsvariablen
            (einfach die Referenzen verschieben)
```

   Es bleiben die fehlenden Methoden für den ADT Stack anzugeben, diese können aber leicht auf die entsprechenden Queue-Methoden zurückgeführt werden:

```
Funktion Object top()
    Wenn queue2.isEmpty()
        holeErstes();
        // die "Umschichtung"-Methode, siehe unten
    Wenn queue2.isEmpty()
        // immer noch leer -> keine Elemente
        löse einen "StackLeer"-Fehler aus
    Gib queue2.front() zurück

Funktion int size()
    Gib queue1.size() + queue2.size() zurück

Funktion boolean isEmpty() // leer <=> keine Elemente
    Gib (size() = 0) zurück.
```

Nun zur Aufwandsanalyse: Für push() wird genauso viel Aufwand benötigt, wie für enqueue(), also ist konstante Zeit möglich, anders sieht es da schon für pop() aus: Werden sehr viele push()-Operationen hintereinander und dann sehr viele pop() Operationen hintereinander ausgeführt, ist es nicht möglich, dies in konstanter Zeit (auch nicht im Mittel) für jede einzelne Operation zu erreichen (mit 2 Queues nicht, sonst schon), denn man kann ja mit Queues nur auf früher eingefügte Elemente zurückgreifen, bei vielen aufeinanderfolgenden pop()s muss man also jedesmal die ganze Queue durchlaufen und hat so für die einzelne pop()-Operation den Aufwand O(Länge des Stacks).

4. **Felder als Doppelschlange**

   Implementieren Sie in Java den ADT deque mittels eines Arrays, das man zirkulär benutzt.

   Im Prinzip wird die Deque genauso implementiert, wie die Queue mit einem zirkulären Array realisiert wurde (→ Vorlesung, Goodrich): Mit zwei int-Variablen, die eine enthält den Index des ersten, die andere den Index des letzten Elements.

   Es bleibt eigentlich nur die Frage zu klären, wie man ein volles von einem leeren Array unterscheidet, hierfür haben wir folgenden Weg gewählt: Die int-Variablen, die den Kopf und den Schwanz der Schlange anzeigen, sind ständig im Bereich $[0, 2\,\text{Länge des Arrays} - 1]$ und zwar aus folgendem Grund: Man kann so leicht erkennen, wann die Schlange leer ist, nämlich dann, wenn der Schwanz vor dem Kopf steht (die int-Variablen, nicht notwendig die Indizes) und wenn sie voll ist: nämlich dann, wenn Kopf = Schwanz − (Länge-1) ist. Dies kann durch den doppelten Laufbereich sehr gut erreicht werden, der positive Laufbereich wurde dabei gewählt (anstelle $[-\text{Länge}, \text{Länge}-1]$, da nur so die modulo-Operationen richtig funktionieren (d.h. immer positive Array-Inzides liefern).

   Doch nun zur Implementierung, zunächst das Interface:

```java
/** Martin Götze, Bernhard Streit
  *   8. Übung, Aufgabe 4
  *
  *  Ein Interface für eine Deque und Definitionen der zugehörigen
  *  Exceptions.
  */

public interface Deque {
    // zunächst die Zugriffsmethoden:
    public int size(); // Anzahl der Wartenden
    public Object first() throws DequeEmptyException; // erstes Element
    public Object last() throws DequeEmptyException; // letztes Element
    public boolean isEmpty(); // Deque leer?
    // und die Methoden, die die Deque ändern:
    public void insertFirst(Object obj) throws DequeFullException; // obj vorn einfügen
    public void insertLast(Object obj) throws DequeFullException; // obj hinten einfügen
    public Object removeFirst() throws DequeEmptyException; // vorn löschen
    public Object removeLast() throws DequeEmptyException; // hinten löschen
}

/** Nun die Fehlerklassen, zunächst eine Basisklasse */
class DequeException extends RuntimeException{
    public DequeException() { super(); }
    public DequeException(String msg) { super(msg); }
}

/** Für volle Deques */
class DequeFullException extends DequeException {
```

```
    public DequeFullException() { super(); }
    public DequeFullException(String msg) { super(msg); }
}

/** Für leere Deques */
class DequeEmptyException extends DequeException {
    public DequeEmptyException() { super(); }
    public DequeEmptyException(String msg) { super(msg); }
}
```

und dann die Implementierung mit Arrays:

```
/** Martin Götze, Bernhard Streit
 *  8. Übung, Aufgabe 4
 *
 *  Eine Realisierung des Deque-Interfaces mittels eines festen, zirkulär genutzten
 *  Arrays
 */
public class ArrayDeque implements Deque {
    // Die Kapizität des Array muss verwaltet werden:
    public final int kapazität;
    // Standardwert
    public final static int DEFAULTKAP = 1024;

    // Das Array und zwei int's, um sich Kopf und Schwanz zu merken:
    private Object[] feld;
    private int head; // erster Eintrag
    private int tail; // letzter Eintrag
    // tail < head heißt: Liste leer

    /** Konstruktoren, einer ohne Argumente, der die Defaultkapazität setzt, einer mit,
     *  dem die Kapazität übergeben wird. */
    public ArrayDeque() {
        this(DEFAULTKAP); // Standard
    }

    public ArrayDeque(int kap) throws IllegalArgumentException {
        // kap == gewünschte Kapazität
        super();
        if (kap <= 0) throw new IllegalArgumentException("Nur positive Zahlen.");
        kapazität = kap;
        feld = new Object[kapazität];
        head = kapazität+1;
        tail = kapazität; // noch keine Elemente
        // es ist immer 0 <= head, tail < 2*kapazität
        /* head und tail können nicht < kapazität gehalten werden,
         * da dann nicht zwischen voller und leerer Deque unterschieden
         * werden kann */
    }

    // Nun kann man die Deque-Methoden mit Leben füllen:
```

```
    // zunächst die Zugriffsmethoden:
    public int size() {
        // also: es sind tail−head+1 Elemente in der Deque
        // [head .... ....   ... tail]
        return tail−head+1;
    }

    public Object first() throws DequeEmptyException {
        if (isEmpty()) throw new DequeEmptyException("Die Schlange ist leer.");
        // ansonsten (zirkulär −> modulo rechnen) Kopf liefern:
        return feld[head % kapazität];
    }

    public Object last() throws DequeEmptyException {
        // wie first(), nur hinten
        if (isEmpty()) throw new DequeEmptyException("Die Schlange ist leer.");
        // ansonsten (zirkulär −> modulo rechnen):
        return feld[tail % kapazität];
    }

    public boolean isEmpty() {
        // also: Leer ist, wenn keine Elemente da sind:
        return (size() == 0);
    }

    // zusätzliche Methode: isFull(), überprüft, ob die Deque voll ist
    public boolean isFull() {
        // voll ist, wenn kapazität Elemente da sind
        return (size() == kapazität);
    }

    // und die Methoden, die die Deque ändern:
    public void insertFirst(Object obj) throws DequeFullException {
        if (isFull()) throw new DequeFullException("Die Schlange ist voll.");
        // ansonsten: head erniedrigen und Element einfügen (zirkulär):
        head−−; // achtung !! könnte kleiner als 0 werden
        if (head < 0) {
            head += kapazität;
            tail += kapazität;
        }
        feld[head % kapazität] = obj;
    }

    public void insertLast(Object obj) throws DequeFullException {
        // wie ... First, nur hinten
        if (isFull()) throw new DequeFullException("Die Schlange ist voll.");
        tail++; // Achtung : könnte >= 2*kapazität werden:
        if (tail >= 2*kapazität) {
            tail −= kapazität;
            head −= kapazität;
```

```
        }
        feld[tail % kapazität] = obj;
    }

    public Object removeFirst() throws DequeEmptyException {
        if (isEmpty()) throw new DequeEmptyException("Die Schlange ist leer.");
        // ansonsten: Element lesen
        Object obj = feld[head % kapazität];
        // Eintrag löschen & head anpassen
        feld[(head++) % kapazität] = null;
        // und zurückgeben:
        return obj;
    }

    public Object removeLast() throws DequeEmptyException {
        // vgl. ... First()
        if (isEmpty()) throw new DequeEmptyException("Die Schlange ist leer.");
        // ansonsten: Element lesen
        Object obj = feld[tail % kapazität];
        // Eintrag löschen & tail anpassen
        feld[(tail--) % kapazität] = null;
        // und zurückgeben:
        return obj;
    }
}
```

Nun noch ein kleines Testprogramm:

```
/** Bernhard Streit, Martin Götze
 *    8. Übung, 4. Aufgabe
 *
 * Eine Testklasse für eine ArrayDeque, es werden ein Paar Zahlen eingefügt und
 * ausgelesen und es werden die Exceptions überprüft, die Zahlen können in der
 * Kommandozeile Übergeben werden.
 */

// ein bißchen Zufall ...
import java.util.Random;

public class ArrayDequeTest {
    /** Diese Klasse dient nur zum Testen, daher gibt es nur die main-Methode
     * und eine Methode, die den Status der Deque ausgibt.
     */
    private static Deque deque = new ArrayDeque(5);

    /** Gibt Größe, Kopf und Schwanz der Deque aus. */
    private static void status() {
        // Kopf und Schwanz ermitteln:
        String vorn, hinten;
```

```
        try {
            vorn = deque.first().toString();
            hinten = deque.last().toString();
        }
        catch (DequeEmptyException e) {
            vorn = "leer";
            hinten = "leer";
        }
        System.out.println(" Status : " + vorn + " --- " + deque.size()
            + " Elemente --- " + hinten);
    }

    public static void main(String[] argumente) {
        Random zufall = new Random();
        int zahlen [];
        // Eingabe verarbeiten:
        if (argumente.length > 0) {
            zahlen = new int[argumente.length];
            for (int i = 0; i < argumente.length; i++) {
                try {
                    zahlen[i] = Integer.parseInt(argumente[i]);
                }
                catch (NumberFormatException e) {
                    zahlen[i] = 0;
                }
            }
        }
        else // Standrardzahlen:
            zahlen = new int []{1, 2, 3, 4, 5, 6, 7, 8};
        // So: die Zahlen in die Deque einfügen:
        for (int i = 0; i < zahlen.length; i++) {
            boolean vorn = (zufall.nextInt(2) == 0);
            // vorne einfügen ??
            System.out.print("Füge die Zahl " + zahlen[i] + " " +
                (vorn ? "vorn  " : "hinten") + " in die Deque ein ...");
            try {
                if (vorn) deque.insertFirst(new Integer(zahlen[i]));
                else deque.insertLast(new Integer(zahlen[i]));
            }
            catch (DequeException e) {
                System.out.print(e.getMessage());
            }
            System.out.println();
            status();
        }
        // und wieder herausholen:
        for (int i = 0; i < zahlen.length; i++) {
            boolean vorn = (zufall.nextInt(2) == 0);
            // vorne holen?
            System.out.print("Lese " + (vorn ? "vorn  " : "hinten") + " 
                aus Deque ...");
            try {
                if (vorn) System.out.print(deque.removeFirst());
                else System.out.print(deque.removeLast());
```

```
            }
            catch (DequeException e) {
                System.out.print(e.getMessage());
            }
            System.out.println();
            status();
        }
        // So, genug getestet.
    }
}
```

mit der Ausgabe

```
> java ArrayDequeTest
Füge die Zahl 1 hinten in die Deque ein ...
 Status : 1 --- 1 Elemente --- 1
Füge die Zahl 2 hinten in die Deque ein ...
 Status : 1 --- 2 Elemente --- 2
Füge die Zahl 3 vorn   in die Deque ein ...
 Status : 3 --- 3 Elemente --- 2
Füge die Zahl 4 hinten in die Deque ein ...
 Status : 3 --- 4 Elemente --- 4
Füge die Zahl 5 hinten in die Deque ein ...
 Status : 3 --- 5 Elemente --- 5
Füge die Zahl 6 hinten in die Deque ein ... Die Schlange ist voll.
 Status : 3 --- 5 Elemente --- 5
Füge die Zahl 7 hinten in die Deque ein ... Die Schlange ist voll.
 Status : 3 --- 5 Elemente --- 5
Füge die Zahl 8 vorn   in die Deque ein ... Die Schlange ist voll.
 Status : 3 --- 5 Elemente --- 5
Lese hinten aus Deque ...5
 Status : 3 --- 4 Elemente --- 4
Lese vorn   aus Deque ...3
 Status : 1 --- 3 Elemente --- 4
Lese hinten aus Deque ...4
 Status : 1 --- 2 Elemente --- 2
Lese vorn   aus Deque ...1
 Status : 2 --- 1 Elemente --- 2
Lese vorn   aus Deque ...2
 Status :  leer --- 0 Elemente --- leer
Lese vorn   aus Deque ... Die Schlange ist leer.
 Status :  leer --- 0 Elemente --- leer
Lese hinten aus Deque ... Die Schlange ist leer.
 Status :  leer --- 0 Elemente --- leer
Lese vorn   aus Deque ... Die Schlange ist leer.
 Status :  leer --- 0 Elemente --- leer
> java ArrayDequeTest 1 4 2 3
Füge die Zahl 1 vorn   in die Deque ein ...
 Status : 1 --- 1 Elemente --- 1
Füge die Zahl 4 hinten in die Deque ein ...
 Status : 1 --- 2 Elemente --- 4
Füge die Zahl 2 vorn   in die Deque ein ...
 Status : 2 --- 3 Elemente --- 4
Füge die Zahl 3 hinten in die Deque ein ...
 Status : 2 --- 4 Elemente --- 3
```

```
Lese vorn   aus Deque ...2
 Status : 1 --- 3 Elemente --- 3
Lese vorn   aus Deque ...1
 Status : 4 --- 2 Elemente --- 3
Lese vorn   aus Deque ...4
 Status : 3 --- 1 Elemente --- 3
Lese vorn   aus Deque ...3
 Status :  leer --- 0 Elemente --- leer
> java ArrayDequeTest 1 4 2 a d 4 r
Füge die Zahl 1 vorn   in die Deque ein ...
 Status : 1 --- 1 Elemente --- 1
Füge die Zahl 4 vorn   in die Deque ein ...
 Status : 4 --- 2 Elemente --- 1
Füge die Zahl 2 hinten in die Deque ein ...
 Status : 4 --- 3 Elemente --- 2
Füge die Zahl 0 hinten in die Deque ein ...
 Status : 4 --- 4 Elemente --- 0
Füge die Zahl 0 hinten in die Deque ein ...
 Status : 4 --- 5 Elemente --- 0
Füge die Zahl 4 hinten in die Deque ein ... Die Schlange ist  voll .
 Status : 4 --- 5 Elemente --- 0
Füge die Zahl 0 hinten in die Deque ein ... Die Schlange ist  voll .
 Status : 4 --- 5 Elemente --- 0
Lese hinten aus Deque ...0
 Status : 4 --- 4 Elemente --- 0
Lese vorn   aus Deque ...4
 Status : 1 --- 3 Elemente --- 0
Lese hinten aus Deque ...0
 Status : 1 --- 2 Elemente --- 2
Lese hinten aus Deque ...2
 Status : 1 --- 1 Elemente --- 1
Lese vorn   aus Deque ...1
 Status :  leer --- 0 Elemente --- leer
Lese hinten aus Deque ... Die Schlange ist  leer .
 Status :  leer --- 0 Elemente --- leer
Lese hinten aus Deque ... Die Schlange ist  leer .
 Status :  leer --- 0 Elemente --- leer
```

Scheint zu klappen.

5. **Einer ist es der fehlt ...**

   In einem Array der Größe n werden ganze Zahlen aus dem Bereich von 0 bis n gespeichert. Die Einträge seien paarweise verschieden ($\rightarrow$ es fehlt genau eine Zahl). Nehmen Sie an, die Einträge seien aufsteigend sortiert. Geben Sie in Pseudocode einen $O(\log_2 n)$-Algorithmus an, der bestimmt, welche ganze Zahl zwischen 0 und n nicht im Array vertreten ist. Begründen Sie die Laufzeit.

   Die fehlende Zahl ist offenbar die Nummer des ersten Array-Platzes, dessen Nummer nicht mit seinem Inhalt übereinstimmt (denn wenn vor $0 \leq k \leq n$ alle Zahlen da sind, stehen auf den k Plätzen vor dem Platz mit der Nummer k die Zahlen $0, \ldots, k-1$, also, falls k da ist wg. der Sortierung auf dem k-ten Platz sicher k).

   Man geht nun nach dem Divide & Impera (von mir aus auch Divide & Coquer, aber für irgendwas muss ich mein Latein doch verwenden) Prinzip vor: Man betrachtet zwei int-Variablen *vorn* und *hinten*, die den Bereich der fehlenden Zahl eingrenzen, sie werden mit 0 resp. $n-1$ initialisiert. Ist vorn = hinten, so gib hinten zurück. Ist vorn nicht gleich hinten, so betrachte das Feld in der Mitte zwischen vorn und hinten. Stimmt dort Platznummer mit Eintrag überein, so fahre mit dem hinteren Teil fort, sonst mit dem vorderen.

   Nun aber zum Pseudocode:

```
Funktion werFehlt(int[] zahlen)
    n ← Länge des Feldes Zahlen
    vorn ← 0
    hinten ← n − 1
    Solange (vorn < hinten)
        mitte ← (vorn + hinten) / 2
        Wenn zahlen[mitte] = mitte
            vorn ← mitte + 1; // hinteren Teil untersuchen
        Sonst
            hinten ← mitte; // vorderen Teil untersuchen
    // Jetzt ist vorn = hinten, also
    Gib hinten zurück
```

   hinten hat immer der Wert eines Feldes, auf dem nicht die richtige Zahl steht, denn hinten steht am Anfang auf einem solchen Feld und wird auch immer nur auf solche Felder gesetzt. Wird aber vorn einmal auf ein solches Feld gesetzt, so ist vorn dann sicherlich das erste solche Feld, denn das Feld direkt davor ist ja mitte, und wenn vorn vorschoben wird, ist mitte richtig besetzt, danach wird also immer nur noch hinten verschoben und der Algorithmus liefert somit stets die Nr des ersten Feldes mit falschem Eintrag, d.h. die fehlende Zahl.

   Nur zur Laufzeit: Sei ein Feld der Länge n gegeben, außerhalb und innerhalb der while-Schleife werden nur Operationen mit konstantem Aufwand durchgefürt, es kommt also darauf an, zu zeigen, dass die Anzahl der while-Schleifen Durchläufe gerade $O(\log_2 n)$ ist, man zeigt dies durch vollständige Induktion, genauer zeigt man durch Induktion nach ℓ, dass: Ist $\ell \in \mathbb{N}_0$ mit

   $$|\text{vorn} - \text{hinten}| \leq 2^\ell - 1$$

   so wird die while-Schleife noch höchstens ℓ mal durchlaufen:

   - **Induktionsanfang** $\ell = 0$: Das ist klar, denn ist

     $$|\text{vorn} - \text{hinten}| \leq 2^0 - 1 = 1 - 1 = 0,$$

     so wird die while-Schleife wegen vorn = hinten kein weiteres Mal durchlaufen.

- **Induktionsvoraussetzung** Für ein $\ell \in \mathbb{N}_0$ gelte, dass die while-Schleife noch höchstens ℓ mal durchlaufen wird, wann immer

  $$|\text{vorn} - \text{hinten}| \leq 2^\ell - 1$$

  gilt.
- **Induktionsschluss** Seien also vorn und hinten mit

  $$|\text{vorn} - \text{hinten}| \leq 2^{\ell+1} - 1$$

  gegeben. Man betrachtet nun einen Durchlauf der while-Schleife und unterscheidet:

  - Es ist zahlen[mitte] = mitte, dann ist nach einem Durchlauf der while-Schleife nun $\text{vorn}_{\text{neu}} = \text{mitte} + 1$, also ist nun

    $$\begin{aligned} |\text{vorn}_{\text{neu}} - \text{hinten}| &= |\text{mitte} + 1 - \text{hinten}| \\ &= \text{hinten} - \left\lceil \frac{\text{vorn} + \text{hinten}}{2} \right\rceil \\ &\leq \text{hinten} - \frac{\text{vorn} + \text{hinten}}{2} \\ &\leq \frac{1}{2} \cdot 2^{\ell+1} - 1 \\ &< 2^\ell - 1 \end{aligned}$$

    Die Schleife wird also nach Induktionsvoraussetzung höchstens ℓ weitere Male, zusammen also höchstens $(\ell + 1)$ mal durchlaufen.
  - Es ist zahlen[mitte] != mitte, dann ist nach einem Durchlauf nun $\text{hinten}_{\text{neu}} = \text{mitte}$, also gilt

    $$\begin{aligned} |\text{vorn} - \text{hinten}_{\text{neu}}| &= |\text{vorn} - \text{mitte}| \\ &= \left\lfloor \frac{\text{vorn} + \text{hinten}}{2} \right\rfloor - \text{vorn} \\ &\leq \text{hinten} - \frac{\text{vorn} + \text{hinten}}{2} \\ &\leq \frac{1}{2} \cdot 2^{\ell+1} - 1 \\ &< 2^\ell - 1 \end{aligned}$$

    Die Schleife wird also nach Induktionsvoraussetzung höchstens ℓ weitere Male, zusammen also höchstens $(\ell + 1)$ mal durchlaufen.

  In beiden Fällen ergibt sich also die Behauptung.

Man hat also für ein Feld der Länge n höchstens $\lceil \log_2 n \rceil + 1$ Schleifendurchläufe, denn es ist

$$2^{\lceil \log_2 n \rceil + 1} - 1 = 2 \cdot 2^{\lceil \log_2 n \rceil} - 1 \geq 2 \cdot n - 1 \geq n$$

das ist aber $O(\log_2 n)$, da z.B. für $n \geq 4$ sicher

$$\lceil \log_2 n \rceil + 1 \leq \log_2 n + 2 \leq 2 \log_2 n$$

Das war aber zu zeigen, denn daraus ergibt sich, wie oben erläutert, eine Laufzeit von $O(\log_2 n)$.