smartlearn
virtual learning. real success.

MODUL 411
DATENSTRUKTUREN UND ALGORITHMEN ENTWERFEN UND ANWENDEN

RALPH MAURER

VERSION 1.1a

Inhaltsverzeichnis

1. Organisation

Modul 411 hat das Kompetenzfeld Apllication Engineering und setzt Erfahrungen im prozeduralen und objektbasierten Programmieren (Module 403 und 404) voraus.

Das Modul 411 beinhaltet folgende Handlungsziele festgelegt durch die ICT-Berufsbildung Schweiz:

1. Für ein gegebenes Problem eine geeignete Datenstruktur definieren und mit den Mitteln einer Programmier-sprache, wie Structs, Referenzen / Zeiger und Arrays umsetzen.
2. Ein Problem analysieren und einen geeigneten Algorithmus zur Lösung mit den Grundelementen Zuweisung, Verzweigung und Schleife entwerfen und mit Prozeduren und Funktionen umsetzen.
3. Algorithmen und Datenstrukturen hinsichtlich Speicher- und Zeitkomple- xität analysieren und dokumentieren.
4. Ein komplexeres Problem auf kleinere Teilprobleme zurückführen und je nach Problemstellung Iteration oder Rekursion einsetzen.
5. Abstrakte Datentypen, wie Liste, Set, Map etc. und die darauf definierten Operationen kennen und zielgerichtet einsetzen können.
6. Datenstrukturen und Algorithmen mit dem Debugger und weiteren Tools untersuchen und dabei speziell die Situation auf Stack und Heap Analysieren und in geeigneter Form darstellen.

2. Teil 1 – Einführung

2.1. Algorithmus

Wenn wir Computerprogramme schreiben, versuchen wir in der Regel ein Verfahren zu finden, welches es uns erlaubt, eine Vielzahl von Problemen zu lösen. Dieses Verfahren ist unabhängig von einer Programmiersprache und somit gleichermassen geeignet für viele Betriebssysteme und Programmiersprachen. Es handelt sich um eine schrittweise Methode, eine Abfolge von Anweisungen und Bedingungen, um ein oder mehrere Probleme zu lösen. Der Begriff Algorithmus wird in der Informatik verwendet, um ein Problemlösungsverfahren zu beschreiben, dass in wohldefinierten Einzelschritten innerhalb eines Computerprogramms implementiert ist.

Algorithmen treffen wir in verschiedenen Bereichen des Lebens an. Zum Beispiel lernten wir in der Grundschule wie man auf Papier grosse Zahlen addiert, subtrahiert, multipliziert und dividiert.

Beispiel Addition:

141175	Anweisung 1:	5 + 5 =	10 (Behalte 1)
+ 4415	Anweisung 2:	7 + 1 + 1 =	9
1	Anweisung 3:	1 + 4 =	5
145590	Anweisung 4:	1 + 4 =	5
	Anweisung 5:	4 + 0 =	4
	Anweisung 6:	1 + 0 =	1
			= 145590

Dieses einfache Beispiel aus der Grundschule zeigt, wie man schrittweise vorgeht, um grosse Zahlen zu addieren.

2. 2. Aufgabe 1a:

Es gibt aber auch ganz andere Beispiele von Algorithmen, wie Kochrezepte. Hier das Beispiel einer Anleitung zum Backen von Muffins:

Muffin-Rezept:

```
Verrühre 200g Butter, 200g Zucker, 4 Eier, 1 Päckchen Backpulver, 1 Päckchen Vanillezucker, 250 g Mehl, 3 Esslöffel Rum und falls Apfelzeit - drei mittelgroße geschälte und zerteilte Äpfel; sonst füge 100g Schokoladenstücken hinzu; fülle den Teig in Muffinförmchen; backe bei 175-200 Grad für etwa 30 min; bestäube die Muffins mit etwas Puderzucker.
```

Arbeitsanweisung

Schreiben Sie das Kochrezept für Muffins in eine Abfolge von Anweisungen in ein Antwort-Dokument. Denken Sie daran, dass die Apfelzeit eine bedingte Anweisung ist, das heisst, dass die Bedingung Apfelzeit nicht immer erfüllt ist:

2. 3. Aufgabe 1b:

Muffin-Rezept unter Berücksichtigung der Backzeit:

Arbeitsanweisung

Implementieren Sie als Iteration (Schleife) die Backzeit für das Muffin-Rezept. Gehen Sie davon aus, dass Sie die Backzeit regelmässig überprüfen, um festzustellen, ob die Muffins gar sind:

2. 4. Aufgabe 1c:

Muffin-Rezept für eine beliebige Anzahl Personen erweitern:

Arbeitsanweisung

Berücksichtigen Sie im Muffin-Kochrezept die Anzahl Personen. Parametrisieren Sie den Algorithmus durch eine Variable, die von aussen gesetzt werden kann (z. B. per Kommandozeile, aus einem File oder über eine Benutzeroberfläche kurz GUI). Die Anzahl Personen (Input) können vom Anwender frei gewählt werden

2. 5. Die drei Ebenen des Algorithmenentwurfs

› **Spezifikation**
Vor dem Schreiben eines Programmes muss das Problem spezifiziert werden. Es stellen sich präzise Fragen, die von Vorteil vor der eigentlichen Programmierarbeit beantwortet werden.
› **Algorithmus**
Vor dem Schreiben eines Programmes muss der genaue Ablauf von elementaren Aktionen, die das Problem schrittweise lösen, klar sein. Alle Einzelschritte und Vorbedingungen müssen bekannt sein und können mit zahlreichen Methoden dargestellt werden.
› **Programm**
Konkrete Formulierung des Algorithmus in einer Programmiersprache (sog. «glue code»). In Modul 411 wenden wir Java an.

Im kommenden Abschnitt gehen wir genauer auf die zwei ersten Ebenen des Algorithmen-entwurfs ein.

2. 5. 1. Spezifikation

Als Beispiel nehmen wir den euklidischen Algorithmus des grössten gemeinsamen Teiler ggT(a,b). Für beliebige Zahlen a und b wird der ggT(a,b) berechnet, also die grösste Zahl, die sowohl a und b teilt.

Wir unterscheiden in der Spezifikation zwischen Pre-conditions (Vorbedingungen) und Post-conditions (Nachbedingungen).

Beispiele von **Pre-conditions** für den ggT(a,b):

Welche Zahlen a, b sind zugelassen?

› Positive oder auch negative Zahlen?

Welche Grundoperationen sind erlaubt?

› ‹+›, ‹-› oder auch div und mod?

Beispiele für **Post-conditions** für den ggT(a,b):

› Was wird ausgegeben, falls m, n < 0?
› Was wird ausgegeben, falls die Eingabe nicht den pre-conditions genügt?

Pre-conditions umfassen alle relevanten Eigenschaften, die vor Ausführung des Algorithmus

gelten.

Post-conditions umfassen alle relevanten Eigenschaften, die nach Ausführung des Algorithmus gelten.

Das Praxisproblem der Spezifikation

In der Praxis werden häufig weniger formale Beschreibungen in natürlicher Sprache verwendet (Pflichtenhefte).

› Häufig umfangreich, mehrdeutig, inkonsistent
› Ist die Ausgabespezifikation das, was der Kunde wollte?
› Bekommt der Kunde von seinen «Zulieferern» das, was die Eingabespezifikation sagt?
› Tut der Algorithmus das, was in der Spezifikation steht?
› Tut der Algorithmus das, was der Kunde wollte?

Tipp: versuchen Sie, für Algorithmen immer eine «möglichst formale» Spezifikation zu schreiben!

2. 5. 2. Algorithmus

Definition von Algorithmus nach Adam Riese:

Geordnete Menge eindeutiger, diskreter, effektiv durchführbarer Schritte, die die gegebene Vorbedingung in endlich vielen Schritten in die gegebene Nachbedingung überführen.

Adam Riese schreibt von einer präzisen Vorschrift, wie in endlich vielen Schritten ein Problem gelöst werden kann!

Ein Beispiel aus dem Rechenbuch von Adam Riese 1574:

Duplieren Lehret wie du ein zahl zweyfaltigen solt. Thu ihm also: Schreib die zahl vor dich /mach ein Linien darunter/ heb an zu forderst / Duplir die erste Figur. Kompt ein zahl die du mit einer Figur schreiben magst / so setz die unden. Wo mit zweyen / schreib die erste / Die ander bahalt im sinn. Darnach duplir die ander / und gib darzu / das du behalten hast / und schreib abermals die erste Figur / wo zwo vorhanden / und duplir fort biß zur letzten / die schreibe gantz auß / als folgende Exempel aufweisen.

aus: A. Risen, Rechenbuch, 1574

Begriffe in der Definition für «Algorithmus» nach Adam Riese

Ausführung des Algorithmus erfolgt in **diskreten Schritten**:

› einzelne, einfache Elementaraktionen
› Welche dies sind hängt vom sog. Algorithmischen Modell ab
  › Maschineninstruktionen?
  › Hochsprachliche Konstruktionen (JAVA)?
  › Mathematische Funktionen wie z.B. log(x)?

Geordnete Menge:

Die Reihenfolge der Schritte genügt gewissen Regeln.

› Schritte müssen nicht unbedingt nacheinander ausgeführt werden.
› Es ist erlaubt, dass mehrere Schritte parallel von verschiedenen Recheneinheiten ausgeführt werden (parallele oder verteilte Algorithmen).
› Bei einer einzigen aktiven Recheneinheit allerdings gibt es einen ersten, zweiten, dritten Schritt usw. (sequentielle Algorithmen).

Eindeutigkeit:

Der Verfahrensablauf ist zu jedem Zeitpunkt determiniert, d.h.,

› zum Zeitpunkt der Ausführung eines Schrittes muss eindeutig und vollständig festgelegt sein, was zu tun ist; und
› gleiche Eingaben sollen zum gleichen Ablauf und Ergebnis führen.
› Ausnahme: randomisierte Algorithmen, deren Ablauf von (mathematisch bestimmten) Zufallsgrößen abhängt. Weitere Ausnahme ist die Parallelisierung durch Recheneinheiten.

Effektivität:

› Jeder Schritt des Verfahrens muss effektiv mechanisch oder rechenbar ausführbar sein.
› Bemerkung: «Effektivität» (= erzielt das gewünschte Resultat) ist nicht zu verwechseln mit «Effizienz» (= Wirtschaftlichkeit)

Terminierung:

Der Algorithmus hält nach endlich vielen Schritten mit einem Ergebnis an. Sofern die Eingaben der Eingabespezifikation genügen.

Nicht erlaubt sind z.B. Schrittfolgen, die nie ein Ergebnis liefern:

› Starte mit Zahl 0
› Addiere 2
› Falls Ergebnis ungerade, halte an, sonst weiter bei Schritt 2.

Korrektheit:

Korrektheit heisst, dass jedes berechnete Ergebnis der Ausgabespezifikation genügt, sofern die Eingaben der Eingabespezifikation eingehalten sind. D.h., Ein- und Ausgabeparameter genügen im Falle der Terminierung immer den Spezifikationen.

Es gibt zahlreiche Methoden zur Darstellung eines Algorithmus:

› Pseudocode
› Flussdiagramme (Ablaufdiagramm)
› Struktogramme
› UML-Diagramme
› Programmcode

Beispiel Pseudocode:

```
Finden des grössten Elements in Array
Algorithm maxOfArray( a, n ):
Input:
a = array of integers
n = Anzahl Elemente in a
Output:
max{ a[0], …, a[n-1] }
Index m mit a[m] = max
currentMax = a[0]
indexOfMax = 0
for i = 1 .. n-1:
     if a[i] > currentMax:
       currentMax = a[i]
       indexOfMax = i
output currentMax, indexOfMax
```

2. 5. 3. Collatz-Problem

Das Collatz-Problem (auch als ULAM Funktion oder als (3n+1)-Vermutung) bezeichnet, ist ein ungelöstes mathematisches Problem, das 1937 von Lothar Collatz gestellt wurde.

Bei dem Problem geht es um Zahlenfolgen, die nach einem einfachen Bildungsgesetz konstruiert werden:

1. Beginne mit irgendeiner natürlichen Zahl n > 0.
2. Ist n gerade, so nimm als Nächstes n/2,
3. Ist n ungerade, so nimm als Nächstes 3n + 1.
4. Wiederhole die Vorgehensweise mit der erhaltenen Zahl, solange n ≠ 1.

So erhält man zum Beispiel für die Startzahl n = 19 die Folge

19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1

Anscheinend mündet jede Folge mit n > 0 in den Zyklus 4, 2, 1 unabhängig davon, welche Startzahl n man probiert hat. Die Collatz-Vermutung lautet:

Jede so konstruierte Zahlenfolge mündet in den Zyklus 4, 2, 1 egal, mit welcher natürlichen Zahl n > 0 man beginnt.

Trotz zahlreicher Anstrengungen gehört diese Vermutung noch immer zu den ungelösten Problemen der Mathematik. Mehrfach wurden Preise für eine Lösung angeboten.

2. 6. Aufgaben 2

Arbeitsanweisung

Beschreiben Sie den Collatz-Algorithmus, im Antwortendokument, in den folgenden Darstellungsformen:

› Flussdiagramm
› Struktogramm
› Pseudocode

2. 7. Aufgabe 3:

Arbeitsanweisung

› Schreiben Sie einen schrittweisen Algorithmus in Prosa (erzählender und eindeutig formulierter, aufzählender Text) für eine automatische Autowaschanlage (Waschstrasse, kein manuelles Waschen). Gehen Sie detailliert vor.

3. Teil 2

3.1. Parameterübergabe

Die einfachste Methode, einem Programm ein paar Daten mit auf dem Weg zu geben, ist die Übergabe von Parametern auf der Kommandozeile. Die JAR-Datei finden Sie im Verzeichnis

```
/dist/.
```

Beispiel: «/home/vmadmin/Dokumente/Modul411/AB411_03_ParameterEingabe/dist/AB411_03_ParameterEingabe.jar»

```
//  @author:       Ralph.Maurer@iet-gibb.ch
//  Compilation:   javac AAB411_03_ParameterEingabe.java
//  Execution:     java -jar AB411_03_ParameterEingabe.jar
package ab411_03_parametereingabe;

public class AB411_03_ParameterEingabe {

    public static void main(String[] args){
        System.out.println(«Eingabe 1: >»+args[0]+»< und»);
        System.out.println(«Eingabe 2: >»+args[1]+»<»);
    }
}
```

mit der Eingabe

```
java -jar AB411_03_ParameterEingabe.jar Hallo 27
Eingabe 1: >Hallo< und
Eingabe 2: >27<
```

In diesem Fall war es wichtig zu wissen, wie viele Eingaben wir erhalten haben. Sollten wir auf einen Eintrag in der Stringliste zugreifen, die keinen Wert erhalten hat, dann passiert folgendes:

```
java -jar AB411_03_ParameterEingabe.jar
Exception in thread «main»
java.lang.ArrayIndexOutOfBoundsException: 1
at MeineEingaben.main(MeineEingaben.java:5)
```

Dieser Fehler lässt sich vermeiden, wenn wir zuerst die Anzahl der übergebenen Elemente überprüfen.

```
//  Compilation:   javac AB411_03_ParameterEingabe.java
//  Execution:     java -jar AB411_03_ParameterEingabe.jar
package ab411_03_parametereingabe;

public class AB411_03_ParameterEingabe {

    public static void main(String[] args){
         for (int i=0; i<args.length; i++)
            System.out.println(«Eingabe «+i+»: >»+args[i]+»<»);

    }
}
```

Beispiel

```
java -jar AB411_03_ParameterEingabe.jar Hallo 27 und noch viel mehr!
Eingabe 0: >Hallo<
Eingabe 1: >27<
Eingabe 2: >und<
Eingabe 3: >noch<
Eingabe 4: >viel<
Eingabe 5: >mehr!<
```

3. 2. Daten aus einer Datei auslesen

Das folgende Programm `ReadFile` liest ein Text aus und zeigt den Inhalt der Datei in der Ausgabe an.

```
//  Compilation:   javac AB411_03_ReadFile.java
//  Execution:     java -jar AB411_03_ReadFile.jar
package ab411_03_readfile;

import java.io.*;
import java.util.Scanner;

public class AB411_03_ReadFile {

    public static void main(String[] args) {
        // Dateiverzeichnispfad zur auszulesenden Datei.
        String
       datei=»/home/vmadmin/Dokumente/Modul411/AB411_03_ReadFile/Liste.txt»;
        // öffne Datei zum Auslesen
       try (Scanner scanner = new Scanner (new File(datei), «UTF-8»)) {
                // Zeilenzähler
                int counter = 0;
                // Solange es noch Zeilen gibt, lies diese aus.
                while (scanner.hasNextLine()){
                    String line = scanner.nextLine();
                    System.out.println(«Zeile «+counter+»: «+line);
                    counter++;
```

```
                }
                // schliesse die Datei
                scanner.close();
        } catch (FileNotFoundException e){
            e.printStackTrace();
        }
    }
}
```

Die Datei `Liste.txt` besteht aus 500 Vornamen und Nachnamen und ist im Modulverzeichnis `99_Div/AB411_03/Liste.txt` zu finden:

```
Ralph van der Mauer
Gabriele Deterti
Heidemarie Obst
Walters Hornung
Carola Braun
Simone Schmidt
Christine Kamenz
Valentin Ott
Marion Adamski
Jörg Gwinner
Hannelore Große
...
```

Scanner (java.util.Scanner)

Das Beispiel zeigt wie `Scanner` aus einer Datei liest. Hierzu muss dem `Scanner`-Objekt ein `File`-Objekt übergeben werden. Dies erhält seinerseits den Pfad zur Datei als `String` . Als zweiter, optionaler Parameter kann ein String des Zeichensatzes der Datei übergeben werden.

`Scanner` deklariert die Methode `hasNextLine()`. Sie liefert so lange `true` wie weitere Elemente ausgelesen werden können und lässt sich somit als Abbruchbedingung in einer `while` -Schleife einsetzen, die den Inhalt der Datei zeilenweiseweise ausgibt.

3. 3. Fehlerbehebung in Java

Das `Try-Catch` Kommando umschliesst einen Codeabschnitt und wird dafür verwendet mögliche Fehler (exceptions) innerhalb dieses Codeabschnittes abzufangen, sodass man darauf reagieren kann. Folgend ist die generelle Syntax dargestellt:

```
try {
    // code der gesichert laeuft
}
catch (ExceptionKlassenname variablenname) {
    // Fehlerbehandlung
}
```

Das `Try-Catch` Kommando besteht aus vier Teilen. Der Block der von try eingeschlossen wird, läuft gesichert ab kann möglicherweise einen Fehler (`exception`) abfangen. Die Fehlerbehandlung findet innerhalb des `Catch`-Blockes statt. Der `ExceptionKlassenname` beschreibt den Fehler auf den wir reagieren wollen. Der Variablenname benennt die `Exception` innerhalb des `Catch`-Blockes, sodass entsprechend der `Exception` innerhalb `Catch`-Blockes reagiert werden kann. Wenn die abgefangene Exception innerhalb des Try-blockes geworfen wird, so springt die Codeausführung direkt in den `Catch`-Block. Wenn die Exception nicht geworfen wird, wird der `Catch`-Block nie aufgerufen

```
try (Scanner scanner = new Scanner (new File(FilePath))) {
...
// Solange es noch Zeilen gibt, lies diese azs
...
       // schliesse die Datei
       ...
}
catch (FileNotFoundException e){
       e.printStackTrace();
}
```

Solange `Scanner scanner = new Scanner (new File(FilePath))` funktioniert, genauer sich die Datei `Liste.txt`» öffnen lässt, wird der `Catch`-Block nicht ausgeführt. Kann die Datei nicht geöffnet werden, weil beispielsweise keine Read-Berechtigungen oder der Dateiverzeichnispfad nicht stimmt, reagiert der `Catch`-Block mit folgendem Fehler:

```
java.io.FileNotFoundException:  /home/vmadmin/Dokumente/Modul411/
AB411_03_ReadFile/XXX/Liste.txt (Datei oder Verzeichnis nicht gefunden)
      at java.io.FileInputStream.open0(Native Method)
      at java.io.FileInputStream.open(FileInputStream.java:195)
      at java.io.FileInputStream.<init>(FileInputStream.java:138)
      at java.util.Scanner.<init>(Scanner.java:611)
      at  ab411_03_readfile.AB411_03_ReadFile.main(AB411_03_ReadFile.
java:15)
```

Sie können auch selber Fehlermeldungen formulieren und ausgeben:

```
try (Scanner scanner = new Scanner (new File(FilePath))) {
...
       // Solange es noch Zeilen gibt, lies diese azs
...
       // schliesse die Datei
       ...
}
catch (FileNotFoundException e){
  System.out.println(«»Datei + « nicht gefunden:\n»+e.getMessage());
}
```

3. 4. Daten in eine Datei schreiben

Wie man Daten in eine Datei speichert, zeigt folgendes Programm. Es speichert Benutzereingaben direkt und zeilenweise in eine Textdatei. Der Vorgang wird solange wiederholt, bis der Benutzer mit q abbricht.

```
//  Compilation:   javac AB411_03_WriteFile.java
//  Execution:     java -jar AB411_03_WriteFile.jar
package ab411_03_writefile;
import java.io.*;

public class AB411_03_WriteFile {

    public static void main(String[] args) throws IOException {
        String datei, in;
        // Dateiverzeichnispfad zur Datei.
           datei=»/home/vmadmin/Dokumente/Modul411/AB411_03_WriteFile/
Ausgabe.txt»;
       // Benutzereingaben lesen mit BufferedReader (InputStreamReader)
          BufferedReader input=new BufferedReader(new InputStreamRea-
der(System.in));
        try{
           // Benutzereingaben schreiben BufferedWriter (Filewriter)
           // 2. Paramter false erstellt Datei falls diese nicht vor-
handen ist
         BufferedWriter myWriter=new BufferedWriter(new FileWriter(da-
tei,false));
           do {
                System.out.println(«Eingabe»);
                // Eine Benutzereingabe einlesen
                in = input.readLine();
              // Abfrage, ob Programm zu beenden ist (break-Anweisung)
                if (!in.equals(«q»)){
                      // Eine Benutzereingabe auf eine Zeile in Datei
schreiben
                    myWriter.write(in+»\n»);
                }
                else
                {   // Datei schliessen
                    myWriter.close();
                    break;
                }
           }
           while(true);
        }
        catch (IOException eIO) {
            System.out.println(«Folgender Fehler trat auf: «+eIO);
        }
    }
}
```

Mit der Anweisung `write` schreiben wir eine Zeichenkette in die durch myWriter bekannte Datei `filenameOutput`. Mit den zwei Symbolen `\n` am Ende der Zeilen wird ein Zeilenumbruch getätigt.

BufferedWriter: Methode write

Wir verwenden eine `do{...}-while` Schleife, um Benutzereingabe bis zum Abbruch mit q zu ermöglichen. Die Klasse BufferedWriter hat die Aufgabe, Stream-Ausgaben zu puffern. Dazu enthält sie einen internen Puffer, in dem die Ausgaben von write zwischengespeichert werden. Erst wenn der Puffer voll ist oder die Methode flush aufgerufen wird, werden alle gepufferten Ausgaben in den echten Stream geschrieben. Das Puffern der Ausgabe ist immer dann nützlich, wenn die Ausgabe in eine Datei geschrieben werden soll. Durch die Verringerung der write -Aufrufe reduziert sich die Anzahl der Zugriffe auf die Datei, und die Performance wird erhöht.

Anwendung

Testen wir das Programm `AB411_03_WriteFile`:
Läuft hier etwas aus dem Ruder,
Ruft man nach dem grossen Bruder.
Bis man eines Tags erwacht,
Und alles ist dann überwacht.
Hör› ich dann von Geistes-Zwergen:
„Ich habe doch nichts zu verbergen!“
Empfehl› ich ihnen einen Blick,
In der Geschichte mal zurück.
Damals als von tausend Jahren,
Von heute 23 vergangen waren,
Wusste man, wohin es führt,
Wenn man alles kontrolliert.
DDR-Helpdesk
Schau in die Welt und merke endlich:
Freiheit ist nicht selbstverständlich!

```
/home/vmadmin/Dokumente/Modul411/AB411_03_WriteFile/Ausgabe.txt
run:
Eingabe
Läuft hier etwas aus dem Ruder,
Eingabe
Ruft man nach dem großen Bruder.
Eingabe
Bis man eines Tags erwacht,
Eingabe
Und alles ist dann überwacht.
Eingabe
Hör› ich dann von Geistes-Zwergen:
Eingabe
„Ich habe doch nichts zu verbergen!“
Eingabe
Empfehl› ich ihnen einen Blick,
Eingabe
In der Geschichte mal zurück.
Eingabe
Damals als von tausend Jahren,
Eingabe
Gerade 23 vergangen waren,
Eingabe
Wusste man, wohin es führt,
```

```
Eingabe
Wenn man alles kontrolliert.
Eingabe
DDR-Helpdesk
Eingabe
Schau in die Welt und merke endlich:
Eingabe
Freiheit ist nicht selbstverständlich!
q
```

3. 5. Thema Kryptographie

Als praktisches Beispiel machen wir einen kurzen Ausflug in die Kryptographie und werden ein Programm schreiben, dass Dateien mit einem Schlüssel codiert (verschlüsselt) und mit dem gleichen Schlüssel wieder decodiert (entschlüsselt).

Wir wenden hierzu eine einfache Methode an und ersetzen Zeichen systematisch durch andere Zeichen. Man sagt dieser Methode auch monoalphabetische Substitution.

Caesar-Codierung

Bei der Cäsarcodierung werden beispielsweise die Buchstaben innerhalb des Alphabets um eine bestimmte Anzahl verschoben. Diese Anzahl ist auch gerade der Schlüssel zum Text. Ist aus dem Modul 114 bekannt.

Diese Verschlüsselung ist sehr einfach und deshalb einfach zu knacken. Cäsar vereinfacht sich in den häufigsten Fällen, indem man Gross- und Kleinschreibung ignoriert:

Klar:	*Die Erde ist keine Scheibe*
Klar vereinfacht:	*die erde ist keine scheibe*
Geheim:	*GLH HUGH LVW NHLNH VFKHLEH*

Als versierter Hacker würde ich nach dem Zeichen suchen, dass am meisten in der Verschlüsselung vorkommt. Es kommt 7xH vor. Im deutschen Alphabet aber auch im Englischen ist der Buchstabe „e“ der häufigste. e ist der fünfte Buchstabe vom Alphabet und H der achte. 8-5=3 und schon ist der Schlüssel 3 geknackt!

XOR-Codierung

Betrachten wir eine schönere Substitution nämlich mit der XOR-Funktion und erinnern wir uns an den Java Operator.

Die Wahrheitstabelle von XOR sieht folgendermasen aus:

a	*b*	*a ^ b*
0	*0*	*0*
0	*1*	*1*
1	*0*	*1*
1	*1*	*0*

Angenommen wir wollen die binäre Zahl 011 (T) mit dem Schlüssel 110 (K) mit einer XOR -Operation verschlüsseln, dann beobachten wir folgende schöne Eigenschaft:

011 ^ 110 = 101

101 ^ 110 = 011

Wenden wir eine XOR-Operation auf T und K an und erhalten das Resultat R: T^K=R,
dann ist R^K=T.

XOR-Encrypter

Wir haben bereits genug Wissen, um diesen Verschlüsselungsalgorithmus korrekt zu programmieren. Die Datei `Gedicht.txt` ist im Modulverzeichnis `99_Div/AB411_03/Gedicht.txt` zu finden. Betrachten wir zunächst nur den Codierer (`Encrypter`) in Java

```
//  Compilation:   javac AB411_03_XOR.java
//  Execution:     java -jar AB411_03_XOR.jar
package ab411_03_xor;
import java.io.*;
import java.util.Scanner;

public class AB411_03_XOR {
    public static String encrypt(String text, int key) {
        // wir werden die Zeichen einzeln codieren
        char[] zeichen = text.toCharArray();
        // bitweise XOR-Verschlüsselung
        for (int i=0; i<zeichen.length; i++)
        // Mit (char)int wandle int in einen char um
        zeichen[i] = (char)(zeichen[i]^key);
        // wir erzeugen aus dem Array vom Typ char einen String
        return new String(zeichen);
       }
       public static void main(String[] args) throws IOException {
        // Dateiverzeichnispfad zur auszulesenden Datei.
        ...
        // Dateiverzeichnispfad zur Erstellung der neuen verschlüsselten Datei.
        ...
        // Versuche Datei zum Auslesen zu öffnen
        try (...) {
            // Writer für Verschlüsselte Zeile zu Speichern
            ...
            // Solange es noch Zeilen gibt, lies diese aus.
            ...
                // schliesse die zu lesende Datei
                ...
                // schliesse die zu verschlüsselte Datei
                ...
        } catch (FileNotFoundException eIO){
            System.out.println(«Folgender Fehler trat auf: «+eIO);
        }
    }
}
```

Gedicht.txt

Läuft hier etwas aus dem Ruder,	[óbqc7~re7rc`vd7vbd7srz7Ebsre;
Ruft man nach dem großen Bruder.	Ebqc7zvy7yvt7srz7pexÈry7Uebsre9
Bis man eines Tags erwacht,	U~d7zvy7r~yrd7Cvpd7re`vtc;
Und alles ist dann überwacht.	Bys7v{{rd7~dc7svyy7ëure`vtc9
Hör› ich dann von Geistes-Zwergen:	_áe07~t7svyy7axy7Pr~dcrd:M`repry-
„Ich habe doch nichts zu verbergen!“	^t7vur7sxt7y~tcd7mb7areurepry6
Empfehl› ich ihnen einen Blick,	Rzgqr{07~t7~yry7r~yry7U{~t|;
In der Geschichte mal zurück.	^y7sre7Prdt~tcr7zv{7mbeët|9
Damals als von tausend Jahren,	Svzv{d7v{d7axy7cvbdrys7]very;
Gerade 23 vergangen waren,	Prevsr7%$7arepvypry7`very;
Wusste man, wohin es führt,	@bddcr7zvy;7`x~y7rd7qëec;
Wenn man alles kontrolliert.	@ryy7zvy7v{{rd7|xycex{{~rec9
DDR-Helpdesk	SSE:_r{gsrd|
Schau in die Welt und merke endlich:	Dtvb7~y7s~r7@r{c7bys7zre|r7rys{~t-
Freiheit ist nicht selbstverständlich!	Qer~r~c7~dc7y~tc7dr{udcaredcóys{~t6

3. 6. Auftrag Ergänzen

Arbeitsanweisung

Ergänzen Sie das Programm `AB411_03_XOR`. Ziel ist, dass unser Gedicht korrekt verschlüsselt wird. Im obigen Beispiel wurder der Schlüssel (Key) 23 angewandt.

3. 7. Auftrag Erweitern

Arbeitsanweisung

Erweitern Sie das Programm: Der Benutzer

› soll selber den Verschlüsselungskey bestimmen können (1. Parameter),
› soll einen Dateiverzeichnispfad zur klaren oder verschlüsselnden Input-Datei angeben können (2. Parameter)
› und soll einen Dateiverzeichnispfad zur Speicherung der verschlüsselnden/klaren Output-Datei angeben können (3. Parameter).
› Das Programm soll parametriesiert von der Konsole aufgerufen werden können.

Beispiel des Aufrufs

```
java -jar AB411_03_XOR.jar 11 /decrypt.txt /encrypt.txt
java -jar AB411_03_XOR.jar 11 /decrypt.txt /encrypt.txt
```

Sollten Sie beim Aufruf mit der Konsole folgenden Fehler erhalten:

```
root@bmLP1:/home/vmadmin/Dokumente/Modul411/AB411_03_XOR_Solution/
dist# java -jar AB411_03_XOR_Solution.jar 23 /home/vmadmin/Dokumente/
Modul411/AB411_03_XOR_Solution/crypt.txt /home/vmadmin/Dokumente/Mo-
dul411/AB411_03_XOR_Solution/decrypt.txt
Exception in thread "main" java.lang.UnsupportedClassVersionError:
ab411_03_xor_solution/AB411_03_XOR_Solution : Unsupported major.minor
version 52.0
        at java.lang.ClassLoader.defineClass1(Native Method) ...
```

Das angesprochene Java der Konsole entspricht nicht Java 8!

4. Teil 3

4.1. Verwendung einfacher Datenstrukturen

4.1.1. Was ist eine Datenstruktur?

„In der Informatik und Softwaretechnik ist eine Datenstruktur ein Objekt zur Speicherung und Organisation von Daten. Es handelt sich um eine Struktur, weil die Daten in einer bestimmten Art und Weise angeordnet und verknüpft werden, um den Zugriff auf sie und ihre Verwaltung effizient zu ermöglichen.

Datenstrukturen sind nicht nur durch die enthaltenen Daten charakterisiert, sondern vor allem durch die Operationen auf diesen Daten, die Zugriff und Verwaltung ermöglichen und realisieren.“ Quelle: https://de.wikipedia.org/wiki/Datenstruktur

Das Wort Datenstruktur verrät schon, dass es sich um Daten handelt, die in irgendeiner Weise in Strukturen, die spezielle Eigenschaften besitzen, zusammengefasst werden. Diese Eigenschaften können sich z.B. darin auswirken, dass ein bestimmtes Datum (an dieser Stelle ist nicht das zeitliche Datum, sondern der Singular von Daten gemeint) schneller gefunden oder eine große Datenmenge platzsparender gespeichert werden kann.

Motivation für Arrays

Nehmen wir an, wir möchten nicht nur einen int, sondern viele davon verwalten. Dann könnten wir dies, mit dem uns bereits bekannten Wissen, in etwa so bewerkstelligen:

```
int a, b, c, d, e, f;
a=0;
b=1;
c=2;
d=3;
e=4;
f=5;
```

Das ist sehr aufwändig und nicht besonders elegant. Einfacher wäre es, wenn wir sagen könnten, dass wir n verschiedene int-Werte speichern möchten und diese dann über ihre Position innerhalb von n (also einen Index) ansprechen. Genau das nennen wir eine Liste oder ein Array von Elementen:

Achtung bei Indizierung der Arrayelemente

Zu beachten ist, dass das erste Element eines Arrays mit dem Index 0 und das letzte der n Elemente mit dem Index n-1 angesprochen werden.

Daran müssen wir uns gewöhnen und es ist eine beliebte Fehlerquelle. Es könnte sonst passieren, dass wir z.B. in einer Schleife alle Elemente durchlaufen möchten, auf das n-te Element zugreifen und einen Fehler verursachen:

```
int[] a = new int[10]; // Erzeugung eines Arrays der Größe 10
for (int i=0; i<=10; i++)
       System.out.println("a["+i+"]="+a[i]);
```

Wo liegt der Fehler der FOR-Schleife?

Was für eine Exception erzeugt Java in diesem Fall?

Finde den Fehler und korrigiere ihn im Antwort-Portfolio

Kleiner Rückblick zur main-Methode und ihrer Parameterübergabe

Es wurde bereits gezeigt, wie Parameter an ein Programm bei dessen Aufruf zu übergeben sind. Hier sehen wir nochmal den Codeabschnitt:

```
public static void main(String[] args){
  for (int i=0; i<args.length; i++)
  System.out.print
}
```

Auch hier wurde Gebrauch von einem Array args gemacht, in diesem Fall ein Array von `Strings`. Die Variablenbezeichnung args wird an dieser Stelle sehr häufig verwendet.

Deklaration und Zuweisung

Um ein n-elementiges Array zu erzeugen, können wir das Array zunächst deklarieren und anschliessend mit dem Befehl new den notwendigen Speicherplatz dafür bereitstellen:

```
<Datentyp>[] <name>;
<name> = new <Datentyp>[n];
```

Beide Zeilen lassen sich auch zu einer zusammengefassen:

```
<Datentyp>[] <name> = new <Datentyp>[n];
```

Literale Erzeugung

Es wurde bereits ein Beispiel gezeigt, wie über einen Index auf die einzelnen Elemente zugegriffen werden kann. Wir haben auch gesehen, dass die Einträge eines int-Arrays mit 0 initialisiert wurden.

Die Initialisierung findet aber nicht bei jedem Datentypen und nicht in jeder Programmiersprache statt, daher wird allgemein empfohlen, in einer Schleife dafür Sorge zutragen.

In dem folgenden Beispiel wollen wir ein Array erzeugen und neue Daten hineinschreiben:

```
int[] a = new int[2];
a[0] = 3;
a[1] = 4;
```

Sollten wir schon bei der Erzeugung des Arrays wissen, welchen Inhalt die Elemente haben sollen, dann können wir das mit der sogenannten literalen Erzeugung vornehmen:

```
int[] a = {1,2,3,4,5};
```

Arraygrösse bleibt unverändert

Wichtig an dieser Stelle ist noch zu wissen, dass wir die Grösse eines erzeugten Arrays nachträglich nicht mehr ändern können. Sollten wir mehr Platz benötigen, bleibt uns keine andere Lösung, als ein grösseres Array zu erzeugen und die Elemente zu kopieren. Wir haben beispielsweise ein Array mit 10 Elementen und wollen ein neues Array

mit 20 Elementen erzeugen und anschliessend die 10 Elemente kopieren:

```
char[] textListe1 = new char[10];
char[] textListe2 = new char[20];
for (int i=0; i<textListe1.length; i++)
    textListe2[i] = textListe1[i]
```

4. 2. Performance Testing

4. 2. 1. Elementeweises Kopieren vs. clone

Für das Kopieren eines Arrays stehen uns verschiedene Möglichkeiten zur Verfügung. Einen Weg haben wir im vorhergehenden Beispiel gesehen: das elementeweise Kopieren.

Interessanterweise gibt es keine schnellere Alternative. Wer das Kopieren in einer Zeile vornehmen möchte, kann für ein Array liste beispielsweise auf die clone-Funktion zurückgreifen:

```
<Datentyp>[] <name> = (<Datentyp> [])liste.clone();
```

Für einen kurzen Performancevergleich der beiden Methoden gibt es in Java mit

`System.currentTimeMillis()` beispielsweise die Möglichkeit, die aktuelle Systemzeit

in Millisekunden vor und nach einem Programmabschnitt auszulesen und anschliessend die Differenz anzugeben:

```
long startZeit = System.currentTimeMillis();

// Programmabschnitt, dessen Zeit gemessen werden soll

long stopZeit = System.currentTimeMillis();
long differenz = stopZeit-startZeit; // in ms
```

Liste für Performancevergleich vorbereiten

Wir verwenden eine Funktion Math.random, die Zufallszahlen vom Typ double gleichverteilt im Bereich von 0 bis 1 liefert.

Wenn wir also die Zufallszahl mit 100 multiplizieren und in einen int umwandeln, erhalten wir eine Zufallszahl aus dem Intervall [0,99].

Wir können uns für unseren Vergleich der beiden Kopiermethoden eine Liste mit 100’000 Elementen zufällig füllen:

```
int[] liste = new int[100000];
for (int i=0; i<liste.length; i++)
   liste[i] = (int)(100*Math.random())
```

Auftrag: Performancevergleich

Arbeitsanweisung

Kopieren Sie diese Liste 10'000 mal mit der clone-Funktion und messen Sie dabei die benötigte Gesamtzeit (Rechenzeit).

Anschliessend werden Sie genauso oft die gleiche Liste nochmal in einer Schleife elementeweise kopieren und analysieren, welches der beiden Verfahren schneller ist.

Erstellen Sie hierzu ein Projekt AB411_04_PerformanceTesting und geben Sie die Resultate wie folgt aus:

```
Dauer (clone): ... ms
Dauer (elementweise Kopieren) ... ms
```

Wiederholen Sie den Test mit einem Kopiervorgang von 100'000 mal. Was fällt Ihnen auf?

5. Teil 4

5.1. Einfache Datenstrukturen

Vereinfachte FOR-Schleife

Die vereinfachte for-Schleifennotation benötigt beim Durchlaufen von Arrays keine Schranken und keine Manipulation von Zählvariablen.

Die Syntax sieht wie folgt aus:

```
for (<Datentyp> <Variablenname> : <Liste>) {
  <Anweisungen>;
}
```

Wir sagen also nicht mehr explizit, welche Indizes innerhalb der Schleife durchlaufen werden sollen, sondern geben eine Variable an, die jeden Index beginnend beim kleinsten einmal annimmt.

```
Schauen wir uns ein Beispiel an:
int[] werte = {1,2,3,4,5,6}; // literale Erzeugung
// Berechnung der Summe int summe = 0;
for (int x : werte)
     summe += x;
```

Damit lässt sich die `ArrayIndexOutOfBoundsException` elegant vermeiden.

5.2. Mehrdimensionale Arrays

5.2.1. Matrix

Für verschiedene Anwendungen ist es notwendig, Arrays mit mehreren Dimensionen anzulegen und zu verwenden. Ein Spezialfall mit der Dimension 2 ist die Matrix. Wir haben ein Feld von Elementen der Grösse n x m.

Wir erzeugen eine n x m Matrix, indem wir zum Array eine Dimension dazunehmen:

```
int[][] a = new int[n][m];
```

Wir können auch mehr Dimensionen erzeugen. Beispiel(k x l x m x n):

```
int[][][][] a = new int[k][l][m][n];
```

Mehrdimensionale Arrays literal erzeugen

in unserem Programm als zweidimensionales Array definieren. Dann kann die literale Erzeugung sehr übersichtlich sein:

```
int[][] matrix = {{4, 5, 6},
                  {2,-9,-3}};
```

Dimensionierung auslesen und Matrix anzeigen

Jetzt wollen wir eine Funktion showMatrix schreiben, die ein zweidimensionales Array vom Datentyp int auf der Konsole ausgeben kann:

```
//  Compilation:   javac AB411_05_Matrix.java
//  Execution:     java -jar AB411_05_Matrix.jar
package ab411_05_matrix;

public class AB411_05_Matrix {

    public static void main(String[] args) {
        int[][] matrix = {{4, 5, 6},
                          {2,-9,-3}};
        for (int i=0; i<matrix.length; i++) {
            for (int j=0; j<matrix[i].length; j++){
                System.out.print(matrix[i][j]+"\t");
            }
            System.out.println();
        }
    }
}
```

In diesem Beispiel sehen wir gleich, wie wir auf die Längen der jeweiligen Dimensionen zurückgreifen können. In dem Beispiel haben wir die Länge 2 in der ersten und die Länge 3 in der zweiten Dimension. Die erste erfahren wir durch

```
matrix.length
```

und die zweite erfahren wir von der Liste an der jeweiligen Stelle in der Matrix mit

```
matrix[i].length
```

Als Ausgabe erhalten wir:

```
4  5  6
2 -9 -3
```

Die Ausgaben wurden in einer Zeile mit \t durch einen Tabulator räumlich getrennt.

Liste unterschiedlich langer Listen

Hier ein Beispiel, das die ersten vier Zeilen des Pascalschen Dreiecks zeigt (https://de.wikipedia.org/wiki/Pascalsches_Dreieck):

```
int[][] pascal = {{1},
{1,1},
{1,2,1},
{1,3,3,1},
{1,4,6,4,1}};
```

Da sich die Ausgabefunktion showMatrix jeweils auf die lokalen Listengrössen bezieht, ergeben die folgenden Zeilen die fehlerfreie Wiedergabe der Elemente:

```
1
1    1
1    2    1
1    3    3    1
1    4    6    4    1
```

Wir können also problemlos mit mehrdimensionalen Listen unterschiedlicher Längen arbeiten.

5. 3. John Conway's Game of Life

Video

John Horten Conway ist ein Mathematiker der 1970 ein System entdeckte mit verblüffenden Ergebnissen. Schauen Sie den Film:

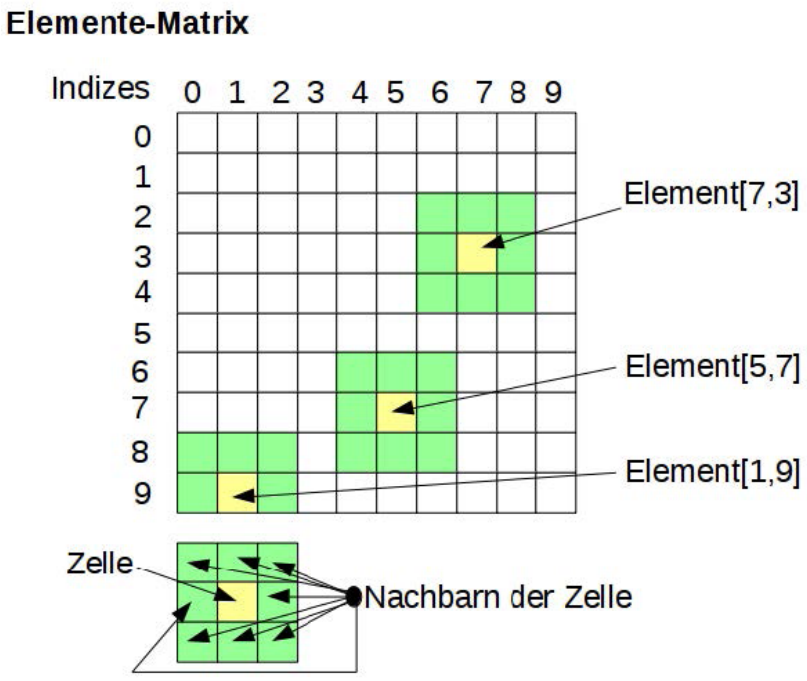
/99_Div/AB411_05/Stephen Hawkings The Meaning of Life (John Conways Game of Life segment).mp4

Lesen Sie:

https://de.wikipedia.org/wiki/Conways_Spiel_des_Lebens

Man stelle sich vor, eine Welt ist eine Matrix und bestünde nur aus 2 Dimensionen. Dies liesse sich wie ein grosses Gitter mit Zellen darstellen. Conway ging in seinem Game of Life von einer Matrix mit Billionen von Elementen aus. Wir arbeiten einfachheitshalber mit 10 Elementen:

Welt als Matri



Eintrag, wir nennen ihn jetzt mal Zelle, kann innerhalb dieser Welt zwei Zustände annehmen: sie ist entweder lebendig oder tot. Jede Zelle interagiert dabei von Generation zu Generation mit ihren maximal 8 Nachbarn.

Die vier Spielregeln

Die Überführun Jeder g der Welt zu einer neuen Generation, also die Überführung aller Zellen in ihren neuen Zustand, unterliegt den folgenden vier Spielregeln:

1. jede lebendige Zelle, die weniger als zwei lebendige Nachbarn hat, stirbt an Einsamkeit
2. jede lebendige Zelle mit mehr als drei lebendigen Nachbarn stirbt an Überbevölkerung
3. jede lebendige Zelle mit zwei oder drei Nachbarn fühlt sich wohl und lebt weiter
4. jede tote Zelle mit genau drei lebendigen Nachbarn wird wieder zum Leben erweckt

Die Idee besteht nun darin, eine konkrete oder zufällige Konstellation von lebendigen und toten Zellen in dieser Matrix vorzugeben. Das bezeichnen wir dann als die erste Generation. Die zweite Generation wird durch die Anwendung der vier Regeln auf jede der Zellen erzeugt. Es wird geprüft, ob Zellen lebendig bleiben, sterben oder neu entstehen. Wird ein solches System beispielsweise mit einer zufälligen Konstellation gestartet, so erinnert uns das Zusammenspiel von Leben und Tod z.B. an Bakterienkulturen in einer Petrischale, daher der Name "Spiel des Lebens".

5. 4. Programmierauftrag

Eine ausführliche Version von Conway's Game of Life in Java hat Edwin Martin programmiert.

http://www.bitstorm.org/gameoflife/

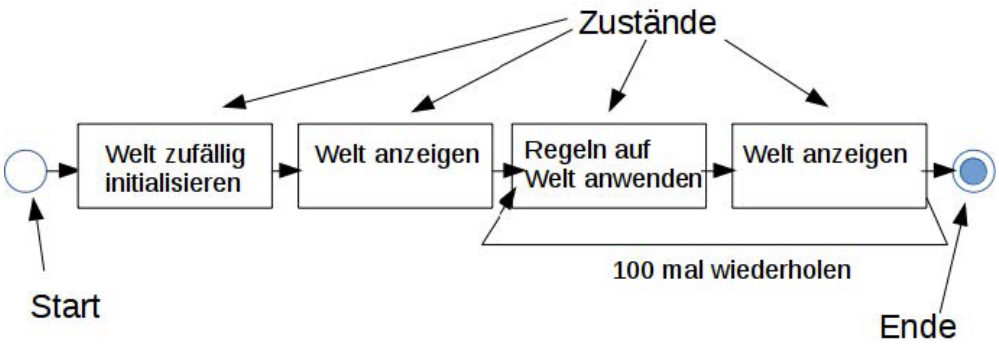
Testen Sie die Version als ausführbare Datei `/99_Div/AB411_05/GameOfLife.jar`.

Erster Projektentwurf

Unsere Aufgabe wird es nun sein, eine sehr einfache Version von Conways Game of Life zu implementieren. Schön wäre auch eine Generationsfolge, damit wir die sich ändernde Welt visuell verfolgen können.

Grobentwurf – endlicher Automat

Identifizierte Teilaufgaben sind in einem endlichen Automaten dargestellt:



Teilaufgaben ausformuliert

› Grobarchitektur des Programms festlegen: Interaktion der Funktionen, Austausch der Daten und Programmschleife.
› Datentyp für die zweidimensionale Weltmatrix definieren, dabei Lösungen für die Randproblematik finden.
› Variable Programmparameter festlegen: Weltmatrixgröße, Verteilung der lebendigen und toten Zellen in der Startkonfiguration.

› Funktion initWelt implementieren, die eine Welt erzeugt und entsprechend der vorgegebenen Verteilung füllt und zurückgibt.
› Funktion zeigeWelt implementieren, die die aktuelle Welt auf der Konsole ausgibt.
› Funktion wendeRegelnAn implementieren, die eine erhaltene Generationsstufe der Welt nach den Spielregeln in die nächste Generationsstufe überführt und das Resultat zurückgibt.

Welt und Randproblematik

In der main-Funktion werden wir eine zweidimensionale Matrix welt anlegen. Für die möglichen Zellzustände ist es naheliegend, eine Matrix vom Typ boolean zu verwenden. Eine tote Zelle wird durch false und eine lebendige Zelle entsprechend durch true repräsentiert.

Es gibt nun verschiedene Möglichkeiten mit der Randproblematik umzugehen. Wenn wir also die acht Nachbarn eines Weltelements erfragen, wollen wir gerne auf eine IndexOutOfBoundsException verzichten. Trotzdem soll die Abfrage der Nachbarn direkt und nicht abhängig von der jeweiligen Position sein. Ein legitimer Weg ist es statt eine 10x10 Matrix eine 12x12 anzuwenden.

Festlegung der Parameter

Die Welt-Matrix

Eine Welt der Dimension 10x10 sollte für den Einstieg genügen. Wir werden für den zusätzlichen Rand insgesamt also eine 12x12 Matrix benötigen.

Lebendige oder tote Zelle

Für die Startkonstellation wünschen wir uns 60% lebendige Zellen mit einer zufälligen Verteilung.

Grundgerüst des Programms

```
//  Compilation:   javac AB411_05_GameOfLife.java
//  Execution:     java -jar AB411_05_GameOfLife.jar
package ab411_05_gameoflife;

public class AB411_05_GameOfLife {

// global definierte Konstanten für die beiden Dimensionen
final static int DIM1 = 12;
final static int DIM2 = 12;

// liefert eine zufällig initialisierte Welt
public static boolean[][] initWelt() {
...}

// gibt die aktuelle Welt aus
public static void zeigeWelt(boolean[][] welt) {
...}

// wendet die 4 Regeln an und gibt die
// Folgegeneration wieder zurück
public static boolean[][] wendeRegelnAn(boolean[][] welt){
...}

public static void main(String[] args) {
    boolean[][] welt = initWelt();
    System.out.println("Startkonstellation");
    zeigeWelt(welt);
    for (int i=1; i<=100; i++){
```

```
        welt = wendeRegelnAn(welt);
        System.out.println("Generation "+i);
        zeigeWelt(welt);
     }
}
```

Tipp 1: Zufällige Initialisierung der Welt

Angenommen, wir wollen einen Zufallswert erzeugen, der in 60% der Fälle true liefert:

```
if (Math.random() > 0.4) // Zufallszahl im Intervall [0,1)
  welt[x][y] = true; // 60% true
else
  welt[x][y] = false; // 40% false
```

Effizienter wäre:

```
welt[x][y] = Math.random() > 0.4;
```

Tipp 2: Zufällige Initialisierung der Welt

Eine Hauptinformation für die Entscheidung, welche der Regeln greift, ist die Anzahl der vorhandenen Nachbarn. Es lohnt sich darüber nachzudenken, ob wir dafür eine eigene Funktion anbieten wollen.

Für die Nachbarschaftsberechnung einer Zelle benötigen wir die Weltmatrix sowie die Position mit x- und y-Koordinaten. Dann müssen wir in zwei Schleifen nur noch die lebenden Zellen aufsummieren und gegebenenfalls die Zelle selbst nochmal abziehen:

```
public static int anzNachbarn(boolean[][] welt, int x, int y) {
  int ret = 0;
  for (int i=x-1; i<=x+1; ++i)
    for (int j=y-1; j<=y+1; ++j)
      if (welt[i][j])
        ret += 1;

// einen Nachbarn zuviel mitgezählt?
if (welt[x][y])
  ret -= 1;

return ret;
}
```

Wenn uns die Nachbarschaftsfunktion zur Verfügung steht, können wir den Abschnitt zu den Spielregeln sehr übersichtlich gestalten.

Viel Erfolg!

5. 5. Ausblick

Wer jetzt denkt, dass Conway's Game of Life nur reine Spielerei ist, der täuscht sich. An dieser Stelle muss kurz erwähnt werden, dass dieses Modell aus Sicht der Berechenbarkeitstheorie ebenso mächtig ist, wie es beispielsweise die Turingmaschine und das Lambda-Kalkül sind. Informatikstudenten, die sich im Grundstudium mit dieser Thematik intensiv auseinandersetzen müssen, wissen welche Konsequenz das hat.

Für den interessierten Leser gibt es zahlreiche Webseiten und Artikel zu diesem Thema, eine Webseite möchte ich allerdings besonders empfehlen.

http://www.conwaylife.com/

Aber Vorsicht: Conway's Game of Life kann süchtig machen!

6. Teil 5

6.1. Zustandsdiagramme

Zustandsdiagramme beschreiben das Verhalten eines Systems. Es beschreibt die möglichen Zustände die ein bestimmtes Objekt annehmen kann, und wie sich ein Objekt nach eintreten eines Ereignisses (über mehrere Anwendungsfälle) verändert hat. Meist wird es für eine Klasse entworfen, um das Verhalten des Objektes während seiner Lebensdauer aufzuzeigen.

In der Informatik versteht man unter Automaten „mathematische Modelle und Systeme, die Informationen verarbeiten und somit Antworten auf Ereignisse oder Eingaben haben“. Auf solche Zustandsautomaten wird hier nicht näher eingegangen. Wir beschränken uns auf die gezeichneten Zustandsdiagramme, man kann aber auch als Zustandstabellen oder Zustandsmatrizen als Darstellungsform wählen. Ein Zustand wird in der Informatik als Kreis und in der Software-Technik als Rechteck dargestellt. Die Transitionen (Übergänge) sind als Pfeile gezeichnet.

Zustandsdiagramme starten mit einer Anfangstransition. Der Startzustand wird laut Notation mit einem unbeschrifteten Pfeil, oder durch die Pfeilbeschriftung „Start“ und einem gefüllten schwarzen Kreis gezeichnet. Doppelte Kreise hingegen markieren den Endzustand.

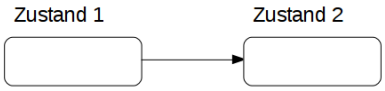
Zustandsdiagramme können:

› nebenläufig sein
› einen Oberzustand besitzen

Nützlich sind nebenläufige Zustandsdiagramme, wenn ein Objekt verschiedene unabhängige Verhalten aufweist. Nebenläufige Abschnitte in einem Zustandsdiagramm können Bereiche sein, in denen ein Objekt (z.B. Auftrag) sich in unterschiedlichen Zuständen gleichzeitig befindet.

Ein Oberzustand ist nützlich, um eine Transition an Unterzustände zu vererben, um nicht jede dieser Transitionen vor diese Unterzustände einzeln schreiben zu müssen:

Grafische Darstellung:



Durch Ereignisse werden Übergänge von einem Zustand 1 um nächsten Zustand 2 ausgelöst. Der Zustand kann Bedingungen an diese Ereignisse knüpfen, erst wenn diese erfüllt sind, kann der Zustand durch dieses Ereignis eingenommen werden. Die Bedingungen können dabei unabhängig vom Ereignis definiert sein. Ereignisse können Aktionen innerhalb eines Zustandes auslösen.

Drei spezielle Auslöser sind definiert:

› entry löst automatisch beim Eintritt einen Zustand aus
› exit löst automatisch beim Verlassen eines Zustandes aus,
› do wird immer wieder ausgelöst, solange der zustand aktiv ist, d.h. nicht verlassen wird.

Aktivitätsdiagramm

Aktivitätsdiagramme (spezielle Form des Zustandsdiagramms) verwendet man, um dynamische Aspekte eines Systems zu modellieren. Modelliert werden nacheinander ablaufende, oder möglicherweise nebenläufig ablaufende Schritte im Verarbeitungsprozess.

Wir können bei der Modellierung dieser dynamischen Aspekte auf Aktivitäten konzentrieren, die zwischen den Objekten stattfinden. Eine Aktivität ist ein Zustand mit einer internen Aktion und einer oder mehreren ausgehenden Transitionen. Sie ist ein einzelner Schritt im Ablauf. Wenn Transitionen durch die Bedingungen unterschieden werden, kann einer Aktivität auch mehrere Transitionen folgen. Aktivitäten können Bestandteil des Zustandsdiagramms sein, werden aber im Aktivitätsdiagramm besser dargestellt, dabei sind die Aktivitäten Objekten eindeutig zugeordnet. Sie können nacheinander, gleichzeitig oder abwechselnd laufen.

Aktivitäten oder Aktivitätsdiagramme sind entweder

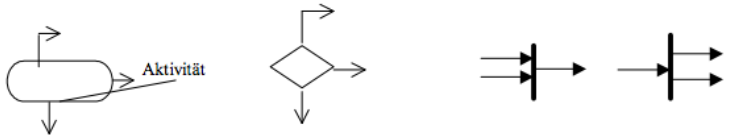
› einer Klasse
› einer Operation
› oder einem Anwendungsfall

zugeordnet.

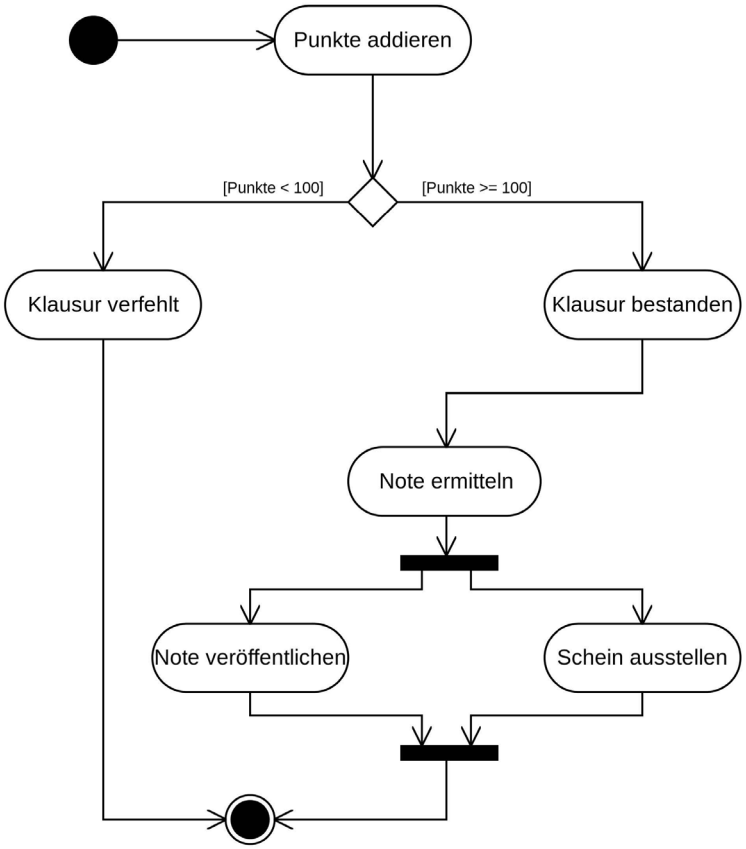
Eingehende Transitionen lösen die Aktivitäten aus. Existieren mehrere Transitionen, können Aktivitäten unabhängig voneinander ausgelöst werden. Die ausgehenden Transitionen werden mit Ereignispfeilen notiert, ohne explizit das Ereignis zu beschriften. Durch den Abschluss der Aktivität wird die Transition automatisch ausgelöst. Transitionen können synchronisiert und geteilt werden (kleine dicke Linien). Von diesen dicken Linien gehen Transitionen weg oder treffen ein.

Aktivitäten bewirken eine Änderung des Objektzustandes.

Grafische Darstellung:



entscheidung Synchronisation Splitting



In eckigen Klammern schreibt man Bedingungen (boolesche Ausdrücke),mit denen man ausgehende Transitionen beschriften kann.

Alternativ können aber auch Verzweigungspunkte verwendet werden. Eine nicht gefüllte Raute zeigt eine (Entscheidungs-) Aktivität, von ihr gehen verschiedene

Transitionen mit ihren Bedingungen aus.

Links ein einfaches Beispiel aus den Sorgen der Lernenden (Quelle: wikipedia)

6.1.1. Ein Baby und seine Zustände

Zustands- und Aktivitätsdiagrammen sehen also sehr ähnlich aus. Im folgenden Abschnitt wollen wir die Bedürfnisse eines neugeborenen Babys genauer untersuchen. Der Einfachheit verzichten wir aber auf Benutzerinteraktionen.

Zunächst definieren wir die vier unterschiedlichen Zustände GLÜCKLICH, HUNGER, ESSEN und SCHREIEN für unser Baby Charline-Anne.

Folgende Szenarien wollen wir dafür realisieren:

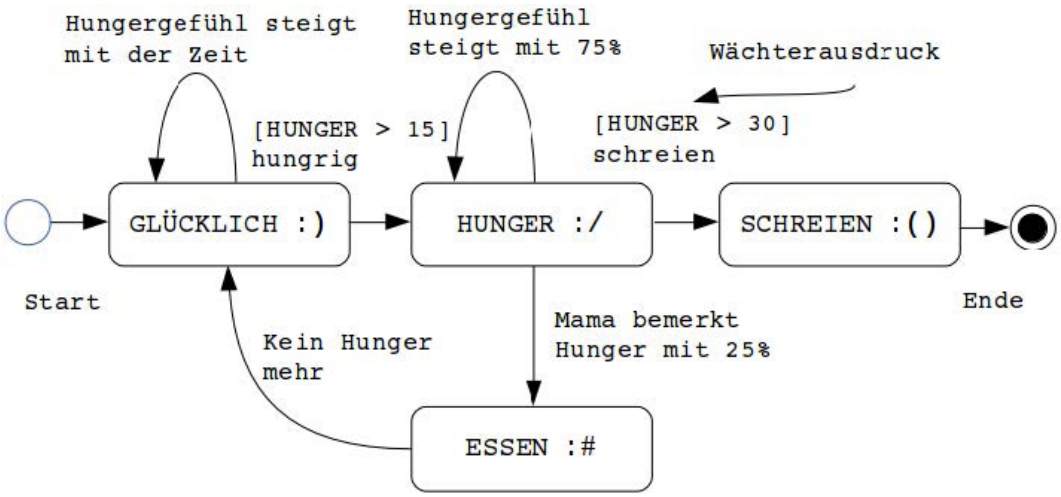
Wenn Charline-Anne keinen Hunger verspürt, dann ist sie GLÜCKLICH. Ist etwas Zeit vergangen, wird sich ein Hungergefühl langsam entwickeln.

Sollten mehr als 15 Einheiten Hunger vorhanden sein, dann wechseln wir in den Zustand HUNGER. Das Baby macht sich bemerkbar und verlangt ESSEN von der Mama und ist in 25% der Fälle erfolgreich. Bei Erfolg wechselt Charline-Anne in den Zustand ESSEN und empfindet kein Hungergefühl mehr. Anschliessend wechselt sie in den Zustand GLÜCKLICH.

Sollte sie beim Verlangen nach Essen über eine längere Zeit Pech haben und mehr als 30 Einheiten Hunger verspüren, schreit sie und wechselt damit in den Zustand SCHREIEN.

Zustandsdiagramme für Charline-Anne entwerfen

Hier sehen wir die Darstellung der Zustände für das Baby Charline-Anne im Zustandsdiagramm:



Zustände werden dabei durch abgerundete Kästchen und der entsprechenden Bezeichnung symbolisiert. Wir können zwischen Zuständen wechseln. Die Pfeile (Transitionen) zeigen die Überführungsrichtungen der Zustände und eine Beschreibung liefert die notwendigen Voraussetzungen dafür. Eine solche Verbindung kann einen Effekt beschreiben. Wenn wir eine Bedingung mit einem Zustandsübergang verknüpfen wollen, können wir wie in der Abbildung gezeigt, einen Wächterausdruck angeben.

Enumeratoren für Zustände einsetzen

Eine einfache Möglichkeit die unterschiedlichen Zustände zu definieren ist der Einsatz von

Enumeratoren. Ein Enumerator repräsentiert für die verschiedenen, konstanten Werte intern einfach Zahlen, ist aber wesentlich besser lesbar: enum <Bezeichner> {<Wert1>, <Wert2>, ...}

Für unsere vier unterschiedlichen Zustände können wir den Enumerator Zustand wie folgt angeben:

```
Das Baby Programm mit Enumeratoren
```

```
//  Compilation:   javac AB411_06_CharlineAnneZustaende.java
//  Execution:     java -jar AB411_06_CharlineAnneZustaende.jar
package ab411_06_charlineannezustaende;

public class AB411_06_CharlineAnneZustaende {
    enum BabyCharlineAnne {HUNGER, SCHREIEN, ESSEN, GLÜCKLICH};

    public static void main(String[] args) {
      BabyCharlineAnne babycharlineanne  = BabyCharlineAnne.GLÜCKLICH;
        int hunger = 0;
        boolean spielLaeuft = true;
        while (spielLaeuft) {
            switch (babycharlineanne) {
                case HUNGER:
                    System.out.println(":/... hunger");
                    if (Math.random()<0.25) // in 25% der Fälle merkt
                                            es Mama
                            babycharlineanne = BabyCharlineAnne.ESSEN;
                    break;
                case SCHREIEN:
                    System.out.println(":()... schreien");
                    spielLaeuft = false;
                    break;
                case ESSEN:
                    System.out.println(":#... essen");
                    hunger = 0;
                    babycharlineanne  = BabyCharlineAnne.GLÜCKLICH;
                    break;
                case GLÜCKLICH:
                    System.out.println(":)");
             }

            if (babycharlineanne != BabyCharlineAnne.ESSEN) {
                hunger += (int)(Math.random()*5);
                if (hunger>15)
                   babycharlineanne = BabyCharlineAnne.HUNGER; // :/
                if (hunger>30)
                   babycharlineanne = BabyCharlineAnne.SCHREIEN; // :()
            }
      }
    }
}
```

Schauen wir uns einen kleinen Testlauf an, bei dem die vielen glücklichen Smilies etwas abgekürzt dargestellt wurden:

```
:)
:/... hunger
:# ... essen
:)
:/... hunger
:# ... essen
:)
... Timer für Zustandsübergänge definieren
```

Das Programm war allerdings so schnell an uns vorbeigelaufen, dass wir die einzelnen Zustände und Übergänge nicht nachvollziehen konnten. Wir können am Ende der while-Schleife nach der Anpassung der Parameter eine kleine Pause von 400 ms einfügen:

```
try {
Thread.sleep(400);
    }
catch (InterruptedException e) {
          e.printStackTrace();
}
```

6. 2. Auftrag

Die Leiden junger Eltern – Vereinfachte Erweiterung

Ausgehend vom Zustand GLÜCKLICH kann Charline-Anne mit 50% Wahrscheinlichkeit in die Hosen machen und der Zustand HOSEVOLL wird erreicht. Die Mama kann mit einer Wahrscheinlichkeit von weiteren 90% die vollen Hosen bemerken und Charline-Anne wickeln. Dadurch ergibt sich ein Zustandsübergang von HOSEVOLL auf WICKELN. Ist Charline-Anne frisch gewickelt ist sie sofort wieder glücklich, wird Sie nicht gewickelt, beginnt das Baby sofort zu schreien.

Arbeitsanweisung

Erweitern Sie das Zustandsdiagramm und programmieren Sie die neuen Zustände und Zustandsübergänge unter Anwendung von 400ms pro Zustandsübergang. Finden Sie passende Smilies, um die Programmausgabe sinngemäss zu erweitern.

Optimierung

Die Aufgabenstellung ist stark vereinfacht und entspricht nicht vollständig der Natur eines Babys. Was müsste beachtet werden, wenn wir ein realitätsnäheres oder vollständigeres Aktivitätendiagramm erstellen möchte?

Gehen Sie nur von den Zuständen GLÜCKLICH – HUNGER – ESSEN – HOSEVOLL – WICKELN – SCHREIEN aus.

7. Teil 6

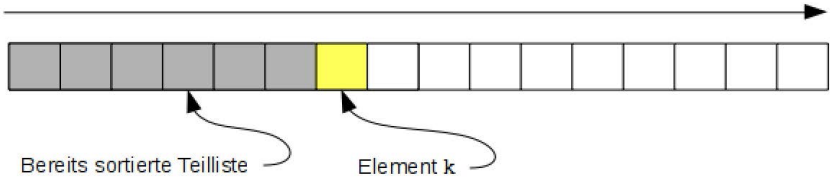
7.1. Sortieralgorithmen

7.1.1. InsertionSort

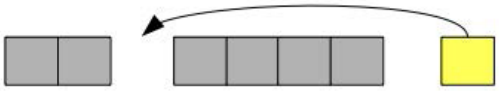
Das Problem unsortierte Daten in eine richtige Reihenfolge zu bringen eignet sich gut, um verschiedene Programmiertechniken und Laufzeitanalysen zu veranschaulichen.

Eine sehr einfache Methode Daten zu sortieren, ist das Einfügen in den sortierten Rest oder das Einfügen in die bereits sortierte Teilliste. Wir sortieren eine Liste, indem wir durch alle Positionen der Liste gehen und das aktuelle Element an dieser Position in die Teilliste davor einsortieren.

Wenn wir das Element k einsortieren möchten, können wir davon ausgehen, dass die Teilliste vor diesem Element, die Positionen 1 bis k-1, bereits sortiert ist.



Um das Element k in die Liste einzufügen, prüfen wir alle Elemente der Teilliste, beginnend beim letzten Element, ob das Element k kleiner ist, als das gerade zu prüfende Element j.Sollte das der Fall sein, so rückt das Element j auf die Position j+1.



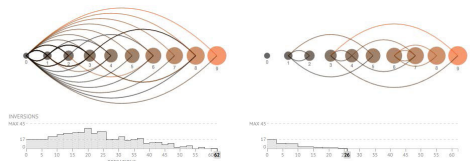
Das wird solange gemacht, bis die richtige Position für das Element k gefunden wurde.

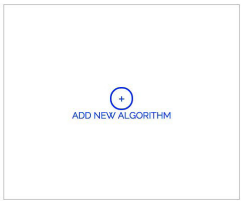
Weitere Erklärungen und Veranschaulichungen zu InsertionSort:

Video

Schauen Sie den Film: /99_Div/AB411_07/Insertion Sort Informatik.mp4

› Besuchen Sie http://sorting.at/ (Mit ADD NEW ALGORITHM können Sie den InsertionSort anzeigen)





7.1.2. Auftrag InsertionSort:

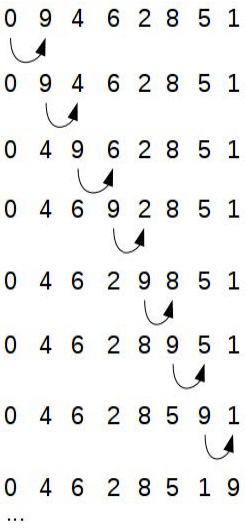
Programmieren Sie InsertionSort und benennen Sie das Projekt mit 411_07_Insertionsort. Verwenden Sie ein Array von Zahlen, das es zu sortieren gilt.

Bsp: Array, Methode, Ausführung.

```
int[] liste = {0,9,4,6,2,8,5,1,7,3};
insertionsort(liste);
run:
0 1 2 3 4 5 6 7 8 9 BUILD SUCCESSFUL (total time: 0 seconds)
```

7.1.3. BubbleSort

Der BubbleSort-Algorithmus vergleicht der Reihe nach zwei benachbarte Elemente einer n-elementigen Liste x und vertauscht sie, falls sie nicht in der richtigen Reihenfolge vorliegen. Ist er am Ende der Liste angekommen wird der Vorgang wiederholt.



Der Algorithmus endet, wenn alle Elemente in der richtigen Reihenfolge vorliegen, im letzten Durchgang also keine Vertauschoperationen mehr stattgefunden haben. Dies geschieht nach maximal (n-1)*(n/2) Schritten.

Aus praktischen Gesichtspunkten muss hier gesagt werden, dass zu sortierende Listen, die bereits schon eine gewisse Vorsortierung besitzen und nicht allzu gross sind, mit BubbleSort relativ schnell sortiert werden.

Folgender Algorithmus würde immer die maximale Anzahl an Schritten ausführen, selbst wenn die Liste bereits sortiert ist:

```
for (i=1 to n-1)
for (j=0 to n-i-1)
if (x[j] > x[j+1])
   vertausche x[j] und x[j+1]
```

Weitere Erklärungen und Veranschaulichungen zu BubbleSort:

Video

› Schauen Sie den Film: /99_Div/AB411_07/Bubble Sort Informatik.mp4
› Besuchen Sie http://sorting.at/ (Mit ADD NEW ALGORITHM können Sie den BubbleSort anzeigen)

7.1.4. Auftrag BubbleSort:

Programmieren Sie BubbleSort mit der IDE von Netbeans und benennen Sie das Projekt mit AB411_07_Bubblesort. Verwenden Sie ein Array von Zahlen, das es zu sortieren gilt.

Bsp: Array, Methode, Ausführung.

```
int[] liste = {0,9,4,6,2,8,5,1,7,3};
```

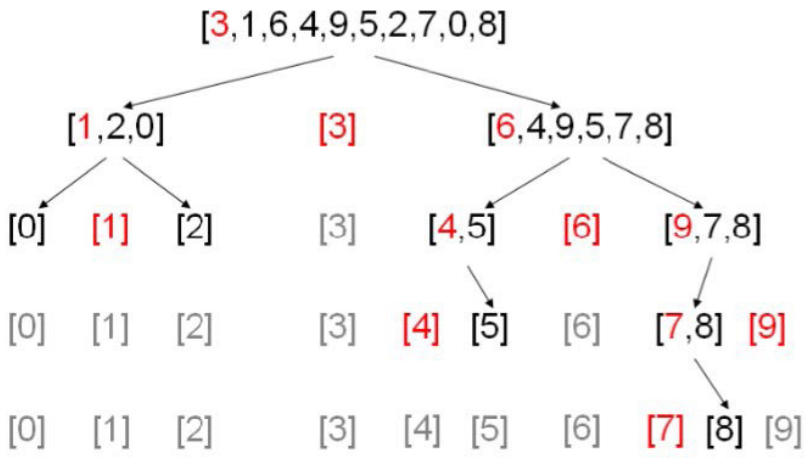
```
bubblesort(liste);
run:
0 1 2 3 4 5 6 7 8 9 BUILD SUCCESSFUL (total time: 0 seconds)
```

7.1.5. QuickSort

Der QuickSort-Algorithmus ist ein typisches Beispiel für die Entwurfstechnik Divide-and-Conquer. In der Informatik bezeichnet teile und herrsche ein Paradigma für den Entwurf von sehr effizienten Algorithmen.

Um eine Liste zu sortieren wird ein Element p ausgewählt und die Elemente der Liste in zwei neue Listen gespeichert, mit der Eigenschaft, dass in der ersten Liste die Elemente kleiner oder gleich p und in der zweiten Liste die Elemente grösser p liegen. Mit den beiden Listen wird wieder genauso verfahren.

Machen wir uns das an einem Beispiel klarer. Wir wollen in diesem Beispiel die Liste [3,1,6,4,9,5,2,7,0,8] sortieren. Dazu wählen wir ein Element (sog. Pivotelement), z.B. das erste Element in der Liste aus und teilen die Liste in zwei neue Listen auf. Diese beiden Listen werden wieder aufgespalten usw.



Das Pivot Element 3 wird zur Erzeugung zwei neuer Listen verwendet. Die erste Liste beinhaltet alle Elemente <3 und die zweite Liste alle Elemente >3.

Weitere Erklärungen und Veranschaulichungen zu QuickSort:

› Schauen Sie den Film: /99_Div/AB411_07/Quick Sort Informatik.mp4
› Besuchen Sie http://sorting.at/ (Mit ADD NEW ALGORITHM können Sie den QuickSort anzeigen)

7.1.6. Auftrag QuickSort:

Programmieren Sie QuickSort mit der IDE von Netbeans und benennen Sie das Projekt mit

AB411_07_Quicksort. Verwenden Sie ein Array von Zahlen, das es zu sortieren gilt.

Bsp: Array, Methode, Ausführung.

```
int[] liste = {0,9,4,6,2,8,5,1,7,3};
quicksort(liste);
run:
0 1 2 3 4 5 6 7 8 9 BUILD SUCCESSFUL (total time: 0 seconds)
```

8. Teil 7

8.1. HEAP und HEAPSORT

8.1.1. Heap

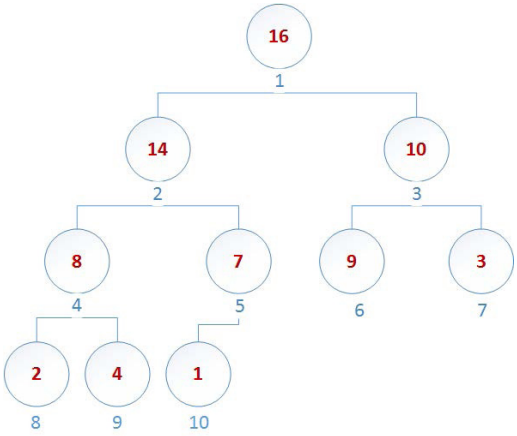
Ein Heap ist in der Informatik ein abstrakter Datentyp der auf einem binären Baum basiert. In einem Heap können Objekte oder Elemente abgelegt und aus diesem wieder entnommen werden. Sie dienen damit der Speicherung von Mengen. Den Elementen ist dabei ein Schlüssel zugeordnet, der die Priorität der Elemente festlegt. Häufig werden auch die Elemente selbst als Schlüssel verwendet. Über die Menge der Schlüssel muss daher eine totale Ordnung vorhanden sein, über welche die Reihenfolge der eingefügten Elemente festgelegt wird. Beispielsweise könnte die Menge der ganzen Zahlen zusammen mit der „Kleiner|Grössergleichrelation“ (<|>) als Schlüsselmenge fungieren. Es gibt auch andere Heap-Bedingungen.

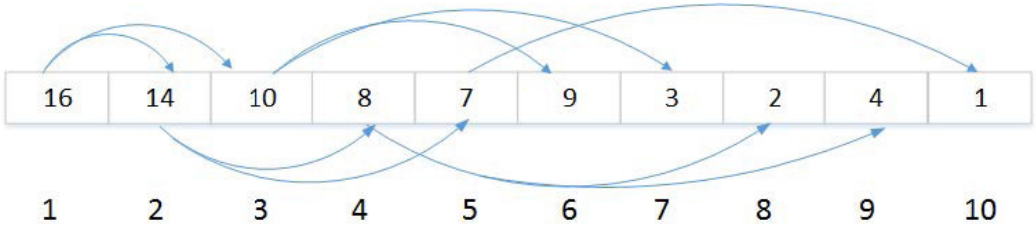
8.1.2. Heap-Datenstruktur

Man kann ein Array als fast vollständigen Binärbaum interpretieren und zur Darstellung eines Heap anwenden. Fast vollständig weil, alle Reihen des Binärbaumes gefüllt sind; ausser die untersten.

Betrachten wir ein Beispiel mit int-Zahlen, die wir später sortieren wollen. Wir geben uns eine Reihe mit 10 Elementen von ganzen Zahlen vor:

[16, 14, 10, 8, 7, 1, 9, 3, 2, 4] und denken uns dazu einen binären Baum mit 10 Blättern (Modell). Die Knoten werden "zeilenweise von links nach rechts" aufgefüllt. Rot sind die Werte aus der Zahlenreihe, die Knotenindizes sind blau gekennzeichnet und werden im folgenden als i bezeichnet.





Betrachten wir das Regelwerk der Knoten und seinen Werten:

Für einen Knoten i gelten

1. PARENT(i) à return [i/2]
2. LEFT(i) à return 2i
3. RIGHT(i) à return 2i + 1

Heap-Arten

Wir unterscheiden die Heap-Arten anhand ihrer Eigenschaften und verzichten auf konkrete

Beispiele.

Max-Heap

Die max-heap Eigenschaft besagt, dass für jeden Knoten gilt:

```
A[PARENT(i)] >= A[i]
```

Das heisst, der Wurzelknoten enthält das grösste Element.

Min-Heap

Die min-heap Eigenschaft besagt, dass für jeden Knoten gilt:

```
A[PARENT(i)] <= A[i]
```

Das heisst, der Wurzelknoten enthält das kleinste Element.

Bemerkung Heapsort-Algorithmus:
Für den Heapsort-Sortieralgorithmus verwenden wir die Max-Heap Eigenschaft.

Max-Heapify

Unter Max-Heapify verstehen wir die Bildung eines Heaps aus einem Array mit Index i unter Anwendung der Max-Heap Eigenschaft.

Algorithmus Max-Heapify:

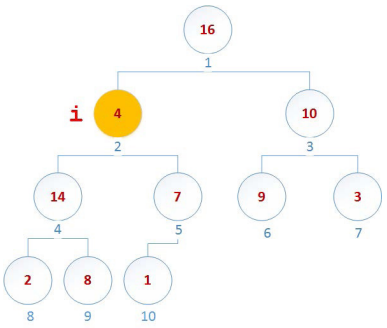
› Wir übergeben einer Prozedur ein Array A und einen Index i.
› LEFT(i) und RIGHT(i) sind max-Heaps
› A[i] ist möglicherweise kleiner als seine Kinder, was die max-heap Eigenschaft verletzt.
› Max-Heapify ist eine Pseudocode-Prozedur, welche die max-heap Eigenschaft wiederherstellt:

```
MAX_HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l <= heap_size[A] and A[l] > A[i]
       then largest = l
       else largest = i
       if r <= heap_size[A] and A[r] > A[largest]
       then largest = r
       if largest != i
       then exchange A[i] <-> A[largest]
              MAX-HEAPIFY(A, largest)
```

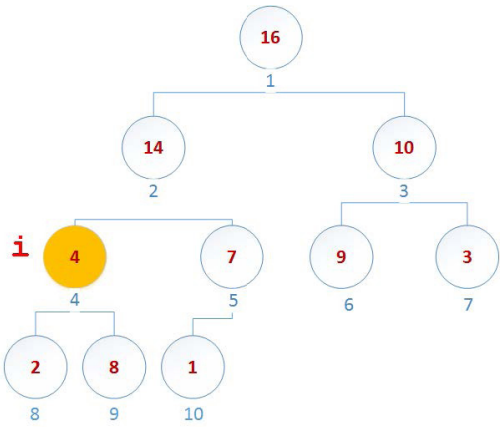
Betrachten wir nun den genauen Vorgang von MAX_HEAPIFY.

Angenommen: `MAX_HEAPIFY(A, 2)`

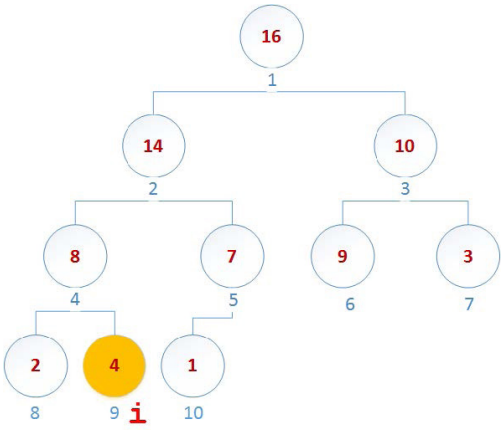
Schritt 1



Schritt 2



Schritt 3

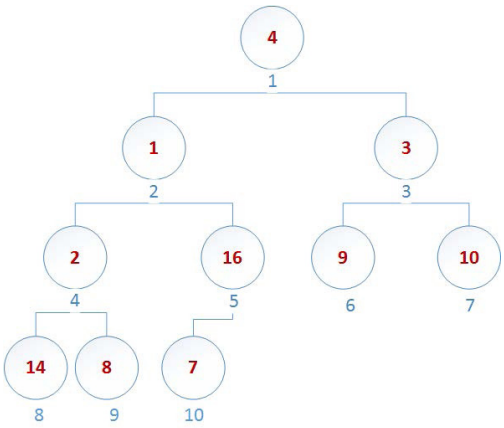


Algorithmus: BUILD_MAX_HEAP

Bauen wir nun einen Max-Heap mit einer Pseudocode-Prozedur BUILD_MAX_HEAP(A).

```
BUILD_MAX_HEAP(A)
heap_size[A] = length[A]
for (i = length[A] / 2, i = 1, i--)
MAX_HEAPIFY(A,i)
```

Jetzt üben wir schrittweise an einem Beispiel: `A = {4,1,3,2,16,9,10,14,8,7}`



Auftrag 1: Max-Heap

Array A:

4	1	3	2	16	9	10	14	8	7

Arbeitsanweisung

Führen Sie schrittweise das Max-Heapify aus. Die Schritte sind durch den neuen PARENT – Knoten des zu untersuchenden Teilbaumes getrennt. Total werden 5 Knoten für das vollständige Max-Heapifyi selektiert. Beachten Sie den Start: `MAX_HEAPIFY(A,(length[A]/2))`

Schreiben Sie die Lösung ins Antworten-Dokument

Algorithmus: HEAPSORT

Betrachten wir nun wie ausgehend von Max-Heap ein Sortieralgorithmus programmiert werden kann:

```
HEAPSORT(A)
BUILD_MAX_HEAP(A)
for (i = length[A], i = 2, i--)
  exchange A[1] <-> A[i]
  heap_size[A] = heap_size[A] - 1
  MAX_HEAPIFY(A,1)
```

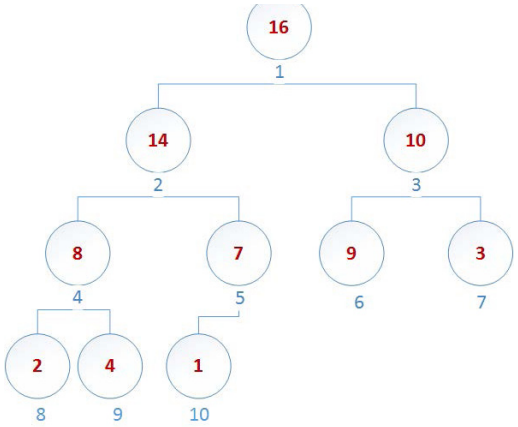
In Worten eines Softwareentwicklers oder -entwicklerin ausgedrückt, lässt sich der HEAPSORT-Algorithmus wie folgt zusammenfassen:

› Nach dem `BUILD_MAX_HEAP` Algorithmus befindet sich das grösste Element in `A[1]`.
› Jetzt muss `A[1]` und `A[n]` vertauscht werden.
› Danach sind die Kinder von `A[1]` weiterhin max-Heaps. `A[1]` verletzt allerdings die max-Heap Eigenschaft.
› Heap_size wir um eins reduziert, somit wird das letzte (und nun grösste Element) aus dem Heap genommen.
› `MAX_HEAPIFY(A,1)` stellt die max-Heap Eigenschaft wieder her.

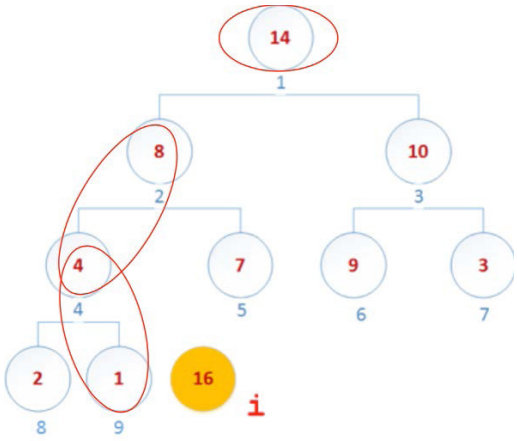
Auftrag 2: HEAPSORT

Arbeitsanweisung

› Wenden Sie HEAPSORT schrittweise auf folgenden Heap an:



Schritt 1 ist gegeben:



Auftrag 3: Coding HEAPSORT

Arbeitsanweisung

Programmieren Sie nun den HEAPSORT, so dass die Applikation mittels jar-Datei und Argumenten gestartet werden kann:

```
java -jar AB411_08_Heapsort.jar 16 14 10 8 7 9 3 2 4 1 1 2 3 4 7 8 9
>>10 14 16
```

9. Teil 8

9.1. Selection-Sort

Aufgabenstellung Selection Sort

Arbeitsanweisung

Ziel dieser Aufgabe ist es, Werte in einem unsortierten Array zu sortieren. Das heisst, ein Programm bekommt ein Array zum Beispiel von ganzen Zahlen (int). Das könnte so aussehen:

```
int[] list = {5, 8, 3, 6, 8, 9, 4, 2};
```

Algorithmus

Suchen Sie im Internet nach der Funktionsweise von Selection-Sort und am besten auch ein Beispiel, welches das Vorgehen des Algorithmus visualisiert. Sie können auch auf (sorting.at)

Tipps

Man wird eine Methode brauchen, die zwei Werte in dem Array tauscht, diese könnte so aussehen:

```
public static void swap(int[] array, int position1, int position2) {
[...]
}
```

Diese Methode soll das Array so ändern, dass der Wert, der an position1 stand, dann an position2 steht, und andersrum.

Eine weitere Methode, die gebraucht wird, ermittelt die Position des kleinesten Wertes in einem Teilarray.

```
public static int findMinimum(int[] array, int startIndex, int endIndex) {
[...]
}
```

`startIndex` und `stopIndex` sind die Positionen, an dem der zu untersuchende Teilbereich des Arrays anfängt, bzw. aufhört. Diese Angaben sind nötig, da ihr später den kleinsten Wert nur in eurem unsortierten Teil des Arrays finden wollt.

Um besser testen zu können, eignet sich eine Methode, die überprüft, ob ein Array sortiert ist oder nicht.

```
public static boolean isSorted(int[] array) {
[...]
}
```

Auch eine Methode, die das sortierte Array auf der Konsole ausgibt, sollte nicht fehlen