Reihungen und Prozeduren

Martin Wirsing

in Zusammenarbeit mit
Moritz Hammer und Axel Rauschmayer

http://www.pst.ifi-lmu.de/lehre/SS06/info2/

Ziele

- Die Datenstruktur der Reihungen verstehen: mathematisch und im Speicher
- Eindimensionale und mehrdimensionale Reihungen verstehen
- Grundlegende Algorithmen auf Reihungen kennen lernen: Suche im ungeordneten Feld
- Prozeduren (nichtstatische Methoden) als (imperative) Abstraktionen für Algorithmen verstehen

Reihungen und deren mathematische Darstellung

Beispiel

Ein Reihung a der Länge 6 kann folgendermaßen dargestellt werden:

a:	´V´	´E´	´R´	´L´	´A´	´G´
Index:	0	1	2	3	4	5

a kann beschrieben werden als die Abbildung

$$a : \{0, \ldots, 5\} \longrightarrow \textbf{char}$$

$$a[i] = \begin{cases} \text{´V´} & \text{falls } i = 0 \\ \text{´E´} & \text{falls } i = 1 \\ \vdots & \\ \text{´G´} & \text{falls } i = 5 \end{cases}$$

Reihungen

- Eine Reihung (auch Feld, Array genannt) ist ein **Tupel von Komponentengliedern gleichen Typs**, auf die über einen Index direkt zugegriffen werden kann.

- Mathematisch kann eine Reihung mit *n* Komponenten vom Typ `type` als endliche Abbildung

$$I_n \longrightarrow \texttt{type}$$

mit Indexbereich I_n = {0, 1,..., n – 1}beschrieben werden.
n bezeichnet die **Länge** der Reihung (auch genannt Dimension *dim* in Info 1).

- Da `type` ein beliebiger Typ ist, kann man auch Reihungen als Komponenten haben $\Rightarrow$ **mehrdimensionale Reihungen**.

Reihungen und deren Speicherdarstellung

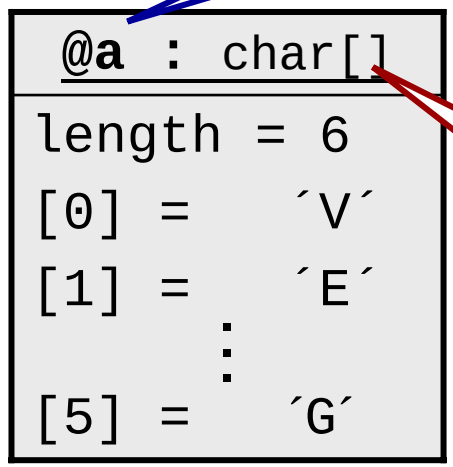
In **Java** wird eine Reihung mit *n* Elementen vom Typ `type` aufgefasst als ein **Zeiger auf einen Verbund (Record) mit den n+1 Komponenten** (Attributen)

```
int   length
type  0
  ⋮
type  (n - 1)
```

Gleicher Typ `type`

Eindeutiger Identifikator (dem Programmierer NICHT bekannt)

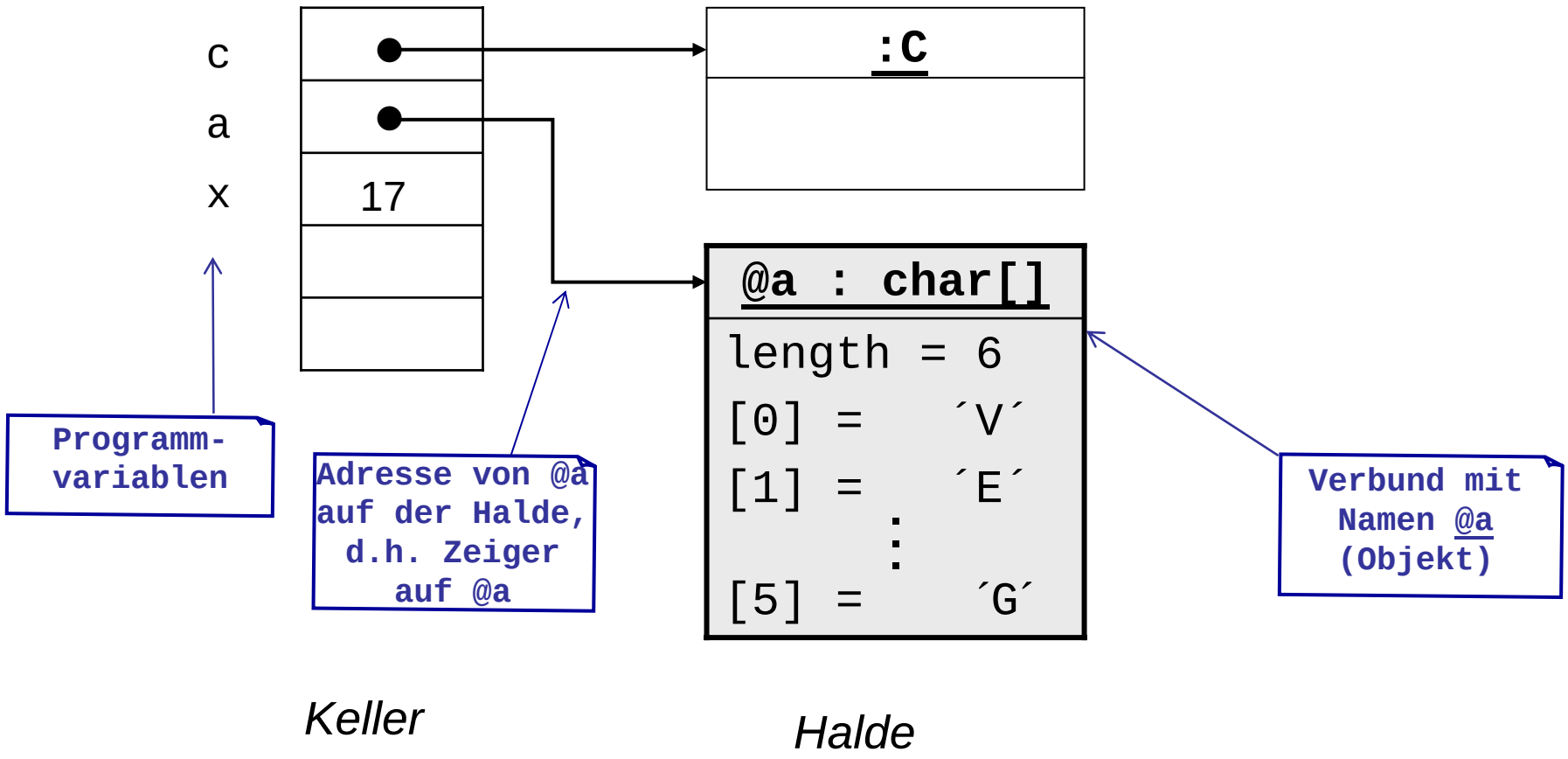
Darstellung des Vektors (als Element eines **Objektdiagramms** in **UML)**



Typ/Klasse der Reihung
Bemerkung: In Java sind Reihungen spezielle Objekte - siehe später -

Reihungen und deren Speicherdarstellung

Die **Speicherorganisation** der Reihung a hat folgende Gestalt



Struktur des Datenspeichers

Der Datenspeicher eines Java-Programms besteht aus zwei Teilen:

- der Keller für die lokalen Variablen und ihre Werte
- die Halde („heap“) für die Reihungen (und die Objekte – siehe später)

Zugriff auf Reihungen

- `a[i]` bezeichnet den Zugriff auf die i-te Komponente der Reihung a (entspricht `get(a, i)` in Info 1) .

  Beispiel:

  Mit `a[0], a[1],..., a[5]` kann man auf die Komponenten der Beispielreihung a zugreifen.

- `a.length` gibt die Länge der Reihung an (entspricht `dim(a)` in Info 1).

  Im **Beispiel** hat `a.length` den Wert `6` .

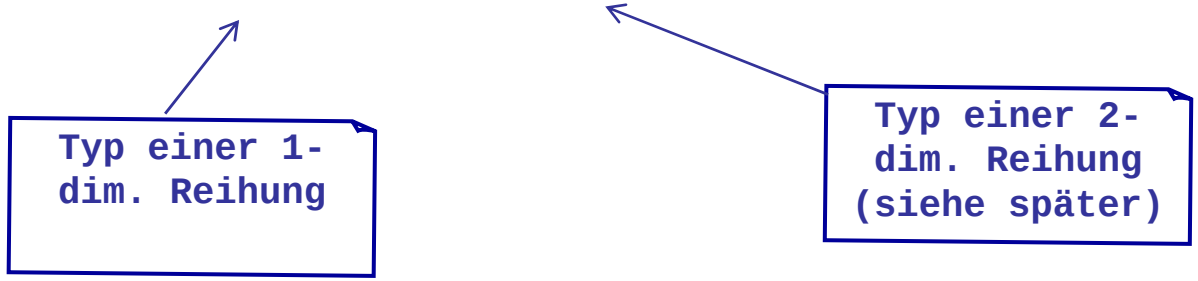
Deklaration von Reihungstypen und -variablen

In Java haben **Reihungstypen** die Form

```
type[]... []
```

Beispiel:

```
int[], int[][], boolean[]
```



Eindimensionale Reihungen

Deklaration einer Reihung mit Elementen vom Typ `type` :

```
type[] var = new type[n];
```

Reihungstyp

Erzeugt neue Reihung der Länge n, bei der jede Komponente mit dem Standardwert von `type` initialisiert wird.

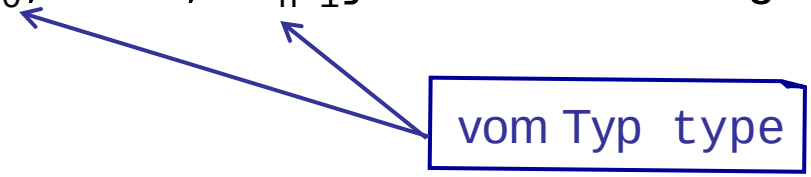
Die Deklaration deklariert lokale Variable `var` vom Type `type[]` und reserviert Speicherplatz für eindimensionale Reihung der Länge n

- Durch die Deklaration werden außerdem implizit `n` zusammengesetzte Variablen `var[0],..., var[n-1]` erzeugt, mit denen man auf die Werte der Komponenten von `var` zugreifen und diese Werte verändern kann.
- Standardwert (Defaultwert) von `int` : `0`
  von `double` : `0.0`
  von `boolean` : `false`

Eindimensionale Reihungen: Initialisierung

Sofort Anfangswerte zuweisen („Initialisierung“)

- `type[] var = {`v_0`, ..., `v_{n-1}`} // sofortige Zuweisung`



! Diese Art der Initialisierung ist aber nur in einer Deklaration zulässig.

- oder **Initialisierung der einzelnen Komponenten**:

```
type[] var = new type[n];
var[0] = v0;
  :
var[n-1] = vn-1;
```

Eindimensionale Reihungen: Initialisierung

Beispiel:

Eindimensionale Reihung a vom Typ **`char`**`[],` d.h. Reihung mit Elementen aus **`char`**.

- **Zuweisung aller Anfangswerte:**

```
char[] a = {´V´, ´E´, ´R´, ´L´, ´A´, ´G´}
```

- **Initialisierung durch Einzelzuweisungen an die Komponenten:**

```
char[] a = new char[6];
a[0] = ´V´;  a[1] = ´E´;   a[2] = ´R´;
a[3] = ´L´;  a[4] = ´A´;   a[5] = ´G´;
```

Eindimensionale Reihungen:

- Man kann **beliebige einzelne Buchstaben ändern**

```
a[3] = ´R´;  //(entspricht update(a, 3,´R´) in Info 1)
a[5] = ´T´;
```

Das ergibt ´V´ ´E´ ´R´ ´R´ ´A´ ´T´ als neuen Wert der Reihung.
Außerdem hat a[3] nun den Wert ´R´.

- Oder **eine gesamte Reihung zuweisen**:

```
char[] c = {´L´,´M´,´U´};
a = c;
```

Bemerkung: Da in Java die Länge der Reihung aber nicht Bestandteil des Typs ist, kann einer Feldvariablen eine Reihung mit einer anderen als der initial angegebenen Länge zugewiesen werden.

Typischer Durchlauf durch Reihungen

- **Die Länge einer Reihung steht in dem Attribut `length`**

```
int x = 10; int[] myArray = new int [x*x+1];
int länge = myArray.length;
```

- **for-Schleifen eignen sich gut um Reihungen zu durchlaufen**

```
for (int x : myArray) x = 2*x;  bzw.
for (int k=0; k<länge; k++) myArray[k] = 2*myArray[k];
```

- **Typische Suche nach einem Element `e` in einer Reihung - mit vorzeitigem Verlassen:**

```
boolean gefunden = false;
for (int x : myArray)
  if (x == e) {gefunden = true; break;}
```

Mehrdimensionale Reihungen

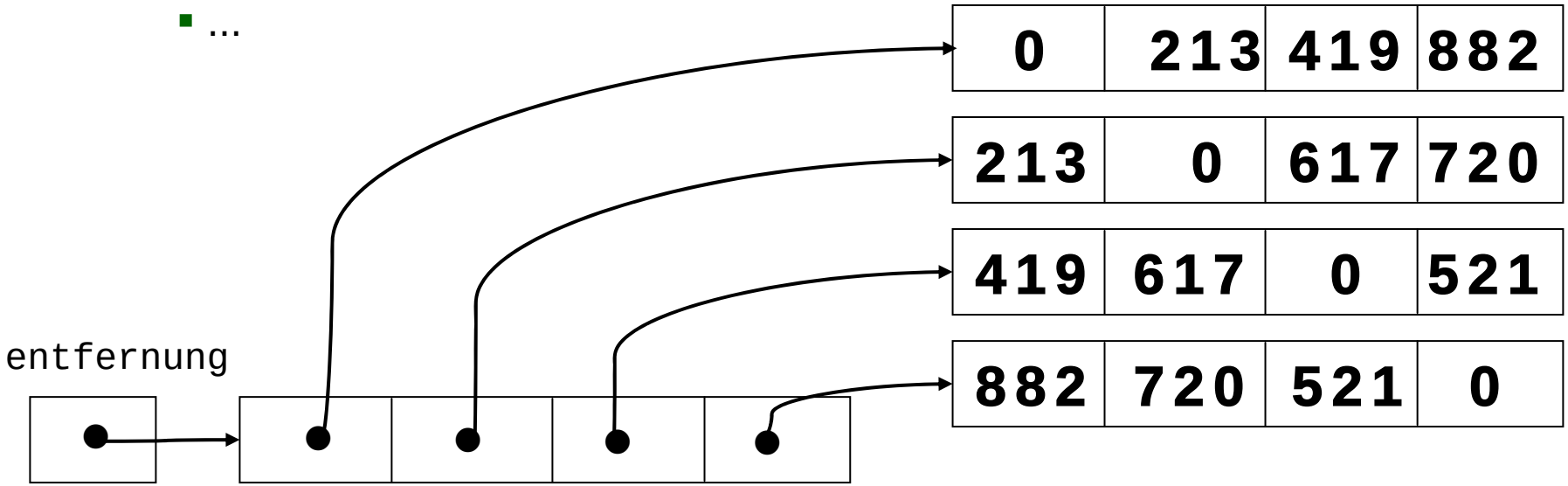
- Matrizen sind mehrdimensionale Reihungen
- Man benutzt Matrizen zur Speicherung und Bearbeitung von
  - Bildern
  - Operationstabellen
  - Wetterdaten
  - Graphen
  - Distanztabellen
  - ...

Mehrdimensionale Reihungen

- **Deklaration**
  - `int [][] entfernung;`
  - `boolean[][] xorTabelle`
- **Deklaration mit gleichzeitiger Erzeugung der Reihung**
  - `int[][] entfernung = new int[4][4];`
- **Deklaration mit Initialisierung**
  - `boolean[][] xorTabelle = {{false, true}, {true, false}};`
  - ```
    int[][]entfernung = {{  0, 213, 419, 882},
                         {213,   0, 617, 720},
                         {419, 617,   0, 521},
                         {882, 720, 521,   0}};
    ```
    ```

Mehrdimensionale Reihungen - gibt‘s gar nicht!

- Mehrdimensionale Reihungen braucht man **eigentlich** nicht
  - Eine zweidimensionale Reihung ist eine Reihung von Zeilen
  - Eine dreidimensionale Reihung ist eine Reihung von zweidimensionalen Reihungen
  - ...



Mehrdimensionale Reihungen

Allgemein

```
type[]...[] var = new type[n1]...[ni][]...[];  (i > 0)
```

deklariert eine Variable `var` vom Typ `type[]...[]`, reserviert Speicherplatz für eine mehrstufige Reihung.

- Mindestens die Länge n_1 des ersten Indexbereiches muss angegeben werden.

Initialisierung:

```
type[][] var = {f0,..., fn1-1};
```

Reihungen von Werten vom Typ `type[]`

Eine mehrdim. Reihung ist eine Reihung von Reihungen

- Dabei können die Längen von $f_0, \ldots, f_{n_1-1}$ unterschiedlich sein.
- Analog für höherdimensionale Reihungen.

Mehrdimensionale Reihungen

Ausdrücke

- Zusammengesetzte Variablen

  `var[`i_1`]...[`i_n`]`

- Initialisierungsausdrücke der Form

  new `type[`n_1`]...[`n_i`][]...[]` bzw. $\{v_0, \ldots v_{n_1-1}\}$

Reihungen: Krumm und schief

- **Die Dimension einer Reihung ist nicht Teil ihres Typs**
  - Folglich können verschieden große Reihungen den gleichen Typ haben

```
int [] alt = new int[3];
int [] neu = new int[17];
alt = neu;                    // das ist in Java möglich!
```

- **Eine Matrix kann verschieden lange Zeilen haben**
  - `int[][] pascalDreieck = {`

```
                    {1},
                  {1, 1},
                {1, 2 , 1},
              {1, 3 , 3 , 1},
            {1, 4 , 6  , 4 , 1}        };
```

- **Wie durchläuft man krumme Reihungen?**
  - Die innere for-Schleife muss die Länge der zu durchlaufenden Zeilen selber bestimmen
  - Das geht mittels des `length`-Attributs

Reihungen: Krumm und schief

- **Durchlauf durch `schiefArray`**

```
for (int zeile=0; zeile < schiefArray.length; zeile++)
  for (int spalte=0; spalte < schiefArray[zeile].length; spalte++)
    tuWasSinnvollesMit schiefArray[zeile][spalte];
```

- **Beispiel Pascal-Dreieck**

```
for (int zeile=0; zeile < pascalDreieck.length; zeile++)
  {  pascalDreieck[zeile][0] = 1; pascalDreieck[zeile][zeile]= 1;
     for (int spalte=1; spalte < pascalDreieck[zeile].length-1; spalte++)
        pascalDreieck[zeile][spalte] =
        pascalDreieck[zeile-1][spalte-1] + pascalDreieck[zeile-1][spalte];
  }
```

Prozeduren (statische Methoden)

Prozeduren dienen zur **Abstraktion von Algorithmen**.

- Durch **Parametrisierung** wird von der Identität der Daten abstrahiert: Eingabedaten können an die speziellen aktuellen (Parameter-) Werte angepasst werden.
- Durch **Spezifikation** des (Ein-/Ausgabe-) Verhaltens wird von den Implementierungsdetails abstrahiert: Vorteile sind
  - **Lokalisierung (Locality):** Die Implementierung einer Abstraktion kann verstanden oder geschrieben werden ohne die Implementierungen anderer Abstraktionen kennen zu müssen.
  - **Änderbarkeit (Modifiability):** Jede Abstraktion kann reimplementiert werden ohne dass andere Abstraktionen geändert werden müssen.

In Java werden **Prozeduren** durch **statische Methoden** realisiert.

Beispiel: Suche nach minimalem Element

```
/**
* searches the minimal element of the specified array
*  @param a: the array.
*  pre: a.length >= 1,
*       i.e. the specified array contains at least one element.
*  @return an element minElem of the specified array with minimum
*  value.
*  post: minElem is smaller or equal than all elements of a.
*/
public static int minElem(int[] a)
```

Spezifikation von Prozeduren

- Die **Spezifikation einer Prozedur** besteht aus:
  - **Prozedurkopf**
    bestehend aus Name, Parameterliste, Rückgabetyp und Sichtbarkeit
  - **(Kommentar mit) Verhaltensbeschreibung**
    bestehend aus
    - **allgemeiner Beschreibung des Verhaltens**
    - **Vor- und Nachbedingungen**
    - **Beschreibung der Parameter**
    - **Beschreibung des Ergebnisses**
    - **Beschreibung des Verhaltens bei Ausnahmen (später)**

Spezifikation von Prozeduren

In Javadoc strukturieren wir die Prozedurspezifikation mit Hilfe folgender Markierungen:

- **pre:** <text> beschreibt die Vorbedingung für den Aufruf der Prozedur.
- **post:** <text> beschreibt die Nachbedingung und damit das Verhalten der Prozedur.

  Bemerkung: Vor- und Nachbedingung können z.B. in OCL spezifiziert und in Java mittels `assert` getestet werden (siehe später).
- **@param** <text> beschreibt die formalen Parameter der Prozedur. **@param** ist eine Standardmarkierung (Tag) von javadoc.
- **@return** <text> beschreibt den Ergebniswert der Prozedur (falls der Ergebnistyp verschieden von `void` ist). **@return** ist eine Standardmarkierung (Tag) von javadoc.
- (**@throws** <text> beschreibt das Verhalten der Prozedur bei Ausnahmen. **@throws** ist eine Standardmarkierung (Tag) von javadoc.)

Suche nach minimalem Element

Implementierung durch Bestimmung des Index des minimalen Elements

Gegeben sei folgendes Feld:

3	-1	15	1	-1

Algorithmus:

- Bezeichne *minIndex* den Index des kleinsten Elements
- Initialisierung *minIndex = 0*
- Durchlaufe die ganze Reihung. In jedem Schritt *i* vergleiche den Wert von *minIndex* (d.h. `a[minIndex]`) mit dem Wert des aktuellen Elements (d.h. `a[i]`). Falls `a[i] < a[minIndex]` setze *minIndex = i*

Suche nach minimalem Element

Java Implementierung durch Bestimmung des Index des minimalen Elements

```
public static int minElem(int[] a) {


    int minIndex = 0;
    for (int i = 1; i < a.length; i++) // Optimierung, da
                                       // a[0] < a[0] falsch ist
    {       if (a[i] < a[minIndex])
            minIndex = i;
    }
    int minElem = a[minIndex];  // minElem ist der Wert
                               // des kleinsten Elements;
                               // minIndex ist der am weitesten links
                               // stehende Index eines kleinsten Elements
    return minElem;

}
```

Suche nach minimalem Element

Java Implementierung durch Bestimmung des Index des minimalen Elements mit Zusicherung der Vor- und Nachbedingung

```
public static int minElem(int[] a) {
      assert a.length >= 1;

      int minIndex = 0;
      for (int i = 1; i < a.length; i++) // Optimierung, da
                                         // a[0] < a[0] falsch ist
      {       if (a[i] < a[minIndex])
              minIndex = i;
      }
      int minElem = a[minIndex];  // minElem ist der Wert
                                  // des kleinsten Elements;
                                  // minIndex ist der am weitesten links
                                  // stehende Index eines kleinsten Elements
      assert isSmallerOrEqual(minElem, a)
              : "Nachbedingung: minElem = "+ minElem;
      return minElem;
}
```

Suche nach minimalem Element

… wobei `smallerOrEqual` folgende Prozedur ist:

```
/**
 * checks whether the specified value is smaller or equal than
 * all values of the elements of the specificed array
 * @param a: the array, e:  the value
 * @return true iff
 * the specified value e is smaller or equal than
 * all values of the  elements of the specified array a.
 *
 */
static boolean isSmallerOrEqual(int e, int[] a) {
       boolean b = true;
       for (int x : a)
           if (!(e <= x)) {b = false; break;}
       return b;
}
```

Zusammenfassung

- Reihungen sind mathematisch gesehen endliche Abbildungen von einem Indexbereich auf einen Elementbereich.
- Im Speicher werden Reihungen repräsentiert als Zeiger auf Vektoren (vgl. später Objekte)
- Prozeduren dienen zur Abstraktion von Algorithmen.
  - Durch Parametrisierung wird von der Identität der Daten abstrahiert.
  - Durch Spezifikation des (Ein-/Ausgabe-) Verhaltens wird von den Implementierungsdetails abstrahiert.
- Ein klassischer Suchalgorithmus ist
  - die Suche nach dem kleinsten Element in einer (ungeordneten) Reihung.