Algorithmen und Datenstrukturen 1

Prof. Dr. Carsten Lecon

Überblick (voraussichtlich)

1. Organisatorisches
2. Algorithmenbegriff
   a) Grundbegriffe
   b) Notation von Algorithmen
   c) Analyse / Bewertung von Algorithmen
3. Entwurf von Algorithmen
   1. Greedy
   2. Divide & Conquer
   3. Rekursion
      - Backtracking
4. Datenstrukturen
   1. Abstrakte Datentypen
   2. Grundlegende Datenstrukturen
   3. Bäume
5. Sortieren

Überblick (aktuell)

1. Organisatorisches
2. Algorithmenbegriff
   - Grundbegriffe
   - Notation von Algorithmen
   - Analyse / Bewertung von Algorithmen
3. Entwurf von Algorithmen I
   - Greedy
   - Divide & Conquer
4. Datenstrukturen I
   - Felder (eindimensional, mehrdimensional, dynamisch)
5. Entwurf von Algorithmen II
   - Rekursion / Backtracking
6. Datenstrukturen II
   - Stapel (Stack)
   - Schlange (Queue)

Überblick (aktuell)

7. Datenstrukturen III (Bäume)
   - Binäre Suchbäume
   - 1-3-4-Bäume
   - Rot-Schwarz-Bäume
8. Entwurf von Algorithmen III (Sortieralgorithmen)
   - Selectionsort
   - Insertionsort (Wdh.)
   - Shellsort
   - Mergesort (Wdh.)
   - Bubblesort
   - Heapsort
   - Quicksort

Elementare Datenstrukturen

- Felder (*Arrays*)
- Stapel (*Stack*)
- Schlange (*Queue*)
- Verkettete Listen (*Lists*)
- Bäume (*Trees*)

Elementare Datenstrukturen

- Felder (*Arrays*)
- Stapel (*Stack*)
- Schlange (*Queue*)
- Verkettete Listen (*Lists*)
- Bäume (*Trees*)

Felder - Lernziele

- Kennen der Eigenschaften und Varianten des Datentyps Feld
- Kennen des Umgangs mit Feldern
- Kennen von Algorithmen auf Feldern

Inhalt – Felder (Arrays)

1. Einführung
2. Eindimensionale Felder
3. Mehrdimensionale Felder
4. Dynamische Felder

Einführung

- Merkmale von Feldern
  - **Grundlegend**e Datenstruktur, praktisch in allen Programmiersprachen verfügbar
  - Feste, **geordnete** Ansammlung von einzelnen Elementen
  - Die einzelnen Elemente haben alle den **gleichen Datentyp**.
  - Der Zugriff auf ein Element erfolgt stets in **konstanter Zeit**.
  - Der Zugriff auf ein Element erfolgt durch einen **Index**.
  - Indizes sind **ganze Zahlen** (oder verlustlos in ganze Zahlen konvertierbar).
  - Felder stehen in direktem Zusammenhang zum **Speichersystem** des Rechners.
    - **In Folge** im Hauptspeicher angeordnet
  - Die Größe von Feldern muss häufig **im Voraus bekannt** sein.
  - Es sind **ein- und mehrdimensionale** Felder möglich.

Inhalt – Felder (Arrays)

1. Einführung
2. Eindimensionale Felder
3. Mehrdimensionale Felder
4. Dynamische Felder

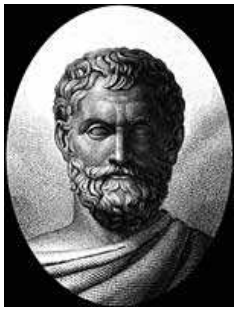
Eindimensionale Felder

- Definitionen:
  - **`Feldname [minIndex…maxIndex] of Elementtyp`**
  - **`Feldname[Elementzahl] of Elementtyp`**
    (Index im Bereich 1..Elementzahl)
  - **`Feldname[] of Elementtyp`**
- Initialisierung:
  - **`0`**, **`0.0`**, **`false`**, **`null`**
- Zugriff:
  - auf einzelnes Element: **`Feldname[idx]`**
    (undefiniert, falls **`idx`** außerhalb Indexbereich)
  - auf Anzahl der Elemente: **`length(Feldname)`**

Eindimensionale Felder

- Deklaration einer Array-Variablen:
  - **`Elementtyp [] Feldname`**
  - **`Elementtyp [] Feldname = {el`$_0$**`, el`$_1$**`, …, el`$_{n-1}$**`}`**
- Erzeugung eines Arrays (Java):
  - **`Feldname = new Elementtyp[Elementzahl]`**
    (Index im Bereich **`0..Elementzahl-1`**; Initialisierung: s.o.)
- Zugriff auf Array-Elemente (Java):
  - auf einzelnes Element: **`Feldname[idx]`**
    (falls **`idx`** außerhalb Indexbereich: **`IndexOutOfBoundsException`**)
  - auf Anzahl der Elemente: **`Feldname.length`**

Eindimensionale Felder



Bildquelle: http://www.s9.com

Beispiel: Sieb des Eratosthenes (Primzahlermittlung)

Prinzip (Primzahlen bis n):

1. Annahme: Jede Zahl ist Primzahl (bis auf 1 und 2): Schreibe alle Zahlen bis n auf.
2. Streiche alle Vielfachen der bislang kleinsten Primzahl p (zuerst 2) durch („Aussieben“).
3. Wiederhole dies mit p, bis $p^2>n$ ist.
4. Die nicht durchgestrichenen Zahlen sind Primzahlen.

Eindimensionale Felder

```
1  procedure berechnePrimzahlen(n)
2    notPrim[n] of boolean
3    p=2
4    while (p·p <n) do
5      if not notPrim[p] then
6        for j= p·p to n step p do
7          notPrim[j] = true
8        end for
9      end if
10     p := p+1
11   end while
```

Eindimensionale Felder

```
11    for p=1 to n do
12      if not notPrim[p] then print(p)
13         end if
14       end for
15    end procedure
```

Eindimensionale Felder

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
F	F	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
F	F	F	T	F	T	F	T	T	T	F	T	F	T	T	T	F	T	F	T	F	T	F	T	F	T	T

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
F	F	F	T	F	T	F	T	T	T	F	T	F	T	T	T	F	T	F	T	T	T	F	T	T	T	T

Eindimensionale Felder

- Beispiel: Ermittlung Minimum / Maximum

- Wie viele Vergleiche benötigt man bei n Zahlen?
  - n-1
- Geht es besser?
  - nein

Eindimensionale Felder

```
1    function Minium(A)
2      min = A[1]
3      for i=2 to length(A) do
4        if A[i] < min then
5          min=A[i]
6        end if
7      end for
9      return min
10   end function
```

Eindimensionale Felder

```
1     function Maximum(A)
2       max = A[1]
3       for i=2 to length(A) do
4         if A[i] > max then
5           max=A[i]
6         end if
7       end for
9       return max
10    end function
```

Eindimensionale Felder

- Simultanes Finden von Minimum und Maximum
  - Einfach zu ergänzen (Zusammenfügen beider Funktionen) → O(n)
  - Es geht aber schneller → O(n/2)

Eindimensionale Felder

- Simultanes Finden von Minimum und Maximum
  - Immer zwei Elemente gleichzeitig betrachten:
    - Vergleich beider Zahlen
    - Die kleinere mit Minimum, die größere mit Maximum vergleichen
    - → Maximal 3 · n/2 Vergleiche

Eindimensionale Felder

```
1       function SimultanMinMax(A)
2         if length(A) modulo 2 = 1 then // ungerade
3           start = 2
4           min = A[1]
5           max = A[1]
6         else
7           start = 3
8           if A[1] < A[2] then
9             min = A[1]
10            max = A[2]
11          else
12            min = A[2]
13            max = A[1]
14          end if
15        end if
```

Eindimensionale Felder

```
16        for i=start to length(A) step 2 do
17          if A[i] < A[i+1] then
18            minIndx = i
19            maxIndx = i+1
20          else
21            minIndx = i+1
22            maxIndx = i
23          end if
24          if A[minIndx] < min then min=A[minIndx]
25          end if
26          if A[maxIndx] > max then max=A[maxIndx]
27          end if
28        end for
29        return min,max
30      end function
```

Eindimensionale Felder

- Laufzeit Simultanes Finden von Minimum und Maximum

	Erste Abfrage	Zweite Abfrage	Abfrage in Schleife	Summe
n ungerade	1	0	3*(n-1)/2	3n/2 -1/2
n gerade	1	1	3*(n-2)/2	3n/2-1

- → Höchstens 3n/2 Vergleiche → **O(n)**

Inhalt – Felder (Arrays)

Mehrdimensionale Felder

- Bislang schon bei Backtracking-Algorithmen Irrgarten und Springerproblem:
  - `private char feld[][] = new char[8][8];`

- In Folge im Hauptspeicher angeordnet

Mehrdimensionale Felder

- Definition:
- **Feldname[minIdx$_1$ … maxIdx$_1$, …, minIdx$_n$ … maxIdx$_n$] of Elementtyp**

- Zugriff auf mehrdimensionale Felder:
- Auf einzelnes Element: **Feldname[idx$_1$, …, id$_n$]**
- Auf Anzahl der Dimension i: **length(Feldname, i)**

Mehrdimensionale Felder

- Deklaration in Java:
  - `Elementtyp Variablenname []…[];`

- Beispiel Deklaration und Initialisierung:
  - `char feld[][];`
  - `feld = new char[8][8];`

- Zugriff:
  - `char c=feld[2][1];`
  - `int laengeZeile = feld.length;`
  - `int laengeSpalte = feld[0].length;`

Mehrdimensionale Felder

- Weiteres Beispiel Deklaration:
  - `int feld[][];`
  - `feld = new int[8][8];`

- Beispiel Initialisierung:
  - `int feld[][] = new { {1,1}, {3,4} };`
  - `int laengeSpalte = feld[0].length;`

Mehrdimensionale Felder

- Mehrdimensionale Felder mit unterschiedlichen Dimensionen:
  - `1`
  - `2 3`
  - `4 5 6`

- In Java:
  - `int arrayInt[][] = { {1}, {2,3}, {4,5,6} };`

- Wie wird die Ausgabe programmiert?

Mehrdimensionale Felder

Ausgabe (Array-Variable `arrayInt`):

```
for (int j=0; j<arrayInt.length; j++) {
  for (int k=0; k<arrayInt[j].length; k++) {
    System.out.print(arrayInt[j][k]);
  }
  System.out.println();
}
```

Mehrdimensionale Felder

- Beispiel Sudoku (9x9-Felder)
- Regeln:
  - Jedes Feld muss eine Ziffer 1..9 enthalten.
  - Eine Ziffer darf nur einmal je
    - Spalte
    - Zeile
    - 3x3-Subblock

    vorkommen.

Mehrdimensionale Felder

	3							
			1	9	5			
	9	8					6	
8				6				
4					3			1
				2				
	6					2	8	
			4	1	9			5
							7	

Sudoku: Aufgabe

Bildquelle: Wikipedia

Mehrdimensionale Felder

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Sudoku: Lösung

Bildquelle: Wikipedia

Inhalt – Felder (Arrays)

1. Einführung
2. Eindimensionale Felder
3. Mehrdimensionale Felder
4. Dynamische Felder

Dynamische Felder

- Manchmal kann man die Größe eines Feldes nicht im Voraus bestimmen:
  - Sortierprogramm, für das die Anzahl der zu sortierenden Elemente nicht begrenzt sein soll
  - Lesen aus einer Datei mit unbestimmter Größe
  - Speichern des Weges bei der Backtracking-Lösung für den Irrgarten
- …

Dynamische Felder

- Intuitives (?) Vorgehen:
  - Mit angemessener Größe beginnen
    - Problem: Leere, unbenutzte Feldelemente
      - Länge muss extra gespeichert werden.
    - Bei Bedarf aufstocken
- Konkret:
  - Erstellung neues Feld
  - Kopieren der alten Daten in das neue Feld

Dynamische Felder

- Hinweise:
  - Vergrößern eines Feldes kostet Zeit → häufiges Vergrößern vermeiden
  - Je nach Problem oder Anwendung sinnvolle Startgröße wählen
  - Sinnvolle Vergrößerungsstrategie wählen
  - Prüfen, ob für Vergrößerung Speicher zur Verfügung steht
  - Gefahren bei kleinen Vergrößerungsschritten:
    - Häufiges Vergrößern (Zeit)
  - Gefahr bei großen Vergrößerungsschritten:
    - Speicherplatz reicht nicht

Dynamische Felder

- Man kann sich selber dynamische Felder implementieren, aber:
  - Es gibt in vielen Programmiersprachen entsprechende vorgefertigte Datenstrukturen.
- Beispiel Java:
  - java.util.ArrayList<Elementtyp>
  - java.util.Vector<Elementtyp>
  - uvm.


AbstractCollection, AbstractList, AbstractQueue, AbstractSequentialList, AbstractSet, ArrayBlockingQueue, ArrayDeque, ArrayList, AttributeList, BeanContextServicesSupport, BeanContextSupport, ConcurrentLinkedQueue, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArraySet, DelayQueue, EnumSet, HashSet, JobStateReasons, LinkedBlockingDeque, LinkedBlockingQueue, LinkedHashSet, LinkedList, PriorityBlockingQueue, PriorityQueue, RoleList, RoleUnresolvedList, Stack, SynchronousQueue, TreeSet, Vector

http://docs.oracle.com/javase/7/docs/api/

Dynamische Felder

- Beispiel `java.util.Vector`
- Feld ist Objekt der Klasse `Vector`
- Elemente sind Objekte mit Referenzdatentyp
- → Wrapper für Basisdatentypen (`int`, `char`, `float`, …)
  - `Integer`, `Character`, `Float`, …
- Methoden:
  - Einfügen
  - Löschen
  - Größe abfragen
  - Suchen in Feld
- Beispiel: Irrgarten/ Springerproblem

Dynamische Felder

- Methoden zum Einfügen (Auswahl):
  - `add(E element)`
  - `add(int index, E element)`

- Methoden zum Löschen:
  - `remove(int index) / removeElementAt(int index)`
  - `remove(Object o)`
  - `removeAll() / clear()`

Dynamische Felder

- Methoden zum Auslesen (Auswahl):
  - `elementAt(int index)`
  - `firstElement()`
  - `lastElement()`
  - `indexOf(Object o)`
  - `indexOf(Object o, int index)`
  - `contains(Object o)`
  - `size()`
  - `isEmpty()`

Felder - Zusammenfassung

- Einfache, grundlegende Datenstruktur
- Elemente haben den gleichen Typ
- Schneller Zugriff auf Elemente, unabhängig von Elementanzahl
- (Fast) kein Verwaltungs-Overhead
- Mehrere Dimensionen möglich
- Normalerweise initial festgelegte Feldgröße
- Alternativ dynamische Felder