Algorithmen und Datenstrukturen

Abstrakte Datenstrukturen: Listen, Keller, Warteschlangen
Sortierung durch Gruppierung

Prof. Dr. Ralf Möller

Universität zu Lübeck

Institut für Informationssysteme

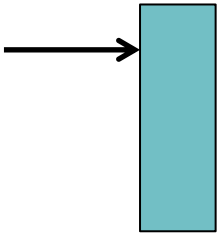
Felix Kuhr (Übungen)

sowie viele Tutoren

Strukturen zur Gruppierung von Daten

auch vertikale Darstellung möglich

- Arrays
  - Zugriff über Index (wir schreiben A[i] oder auch A[i] := …)
  - Funktion length ist definiert
  - Zeichenketten als spezielle Arrays (Notation "…")
  - Funktion A: I → D Notation: [3, 42, 55, 6]
- Tupel (Reihung von Komponenten)
  - Beispiel: ("Ralf", 55, 1.8) n-Tupel
  - (p, age, height) := ("Ralf", 55, 1.8)
  - Zugriff auch über benannte Funktionen
    - Namen von Funktionen, die auf Komponenten zugreifen, können vereinbart werden
    - Anzahl der Komponenten üblicherweise klein



auch horizontale Darstellung möglich

Wenn wir length effizient realisieren wollen…

- … müssten wir uns Arrays so vorstellen

- A[i] muss der Compiler entsprechend umsetzen
- Wir bleiben aber in der Darstellung bei

auch vertikale Darstellung möglich

Listen als abstrakte Datentypen (ADTs)

Notation: [4, 2, 9] oder [] für die leere Liste

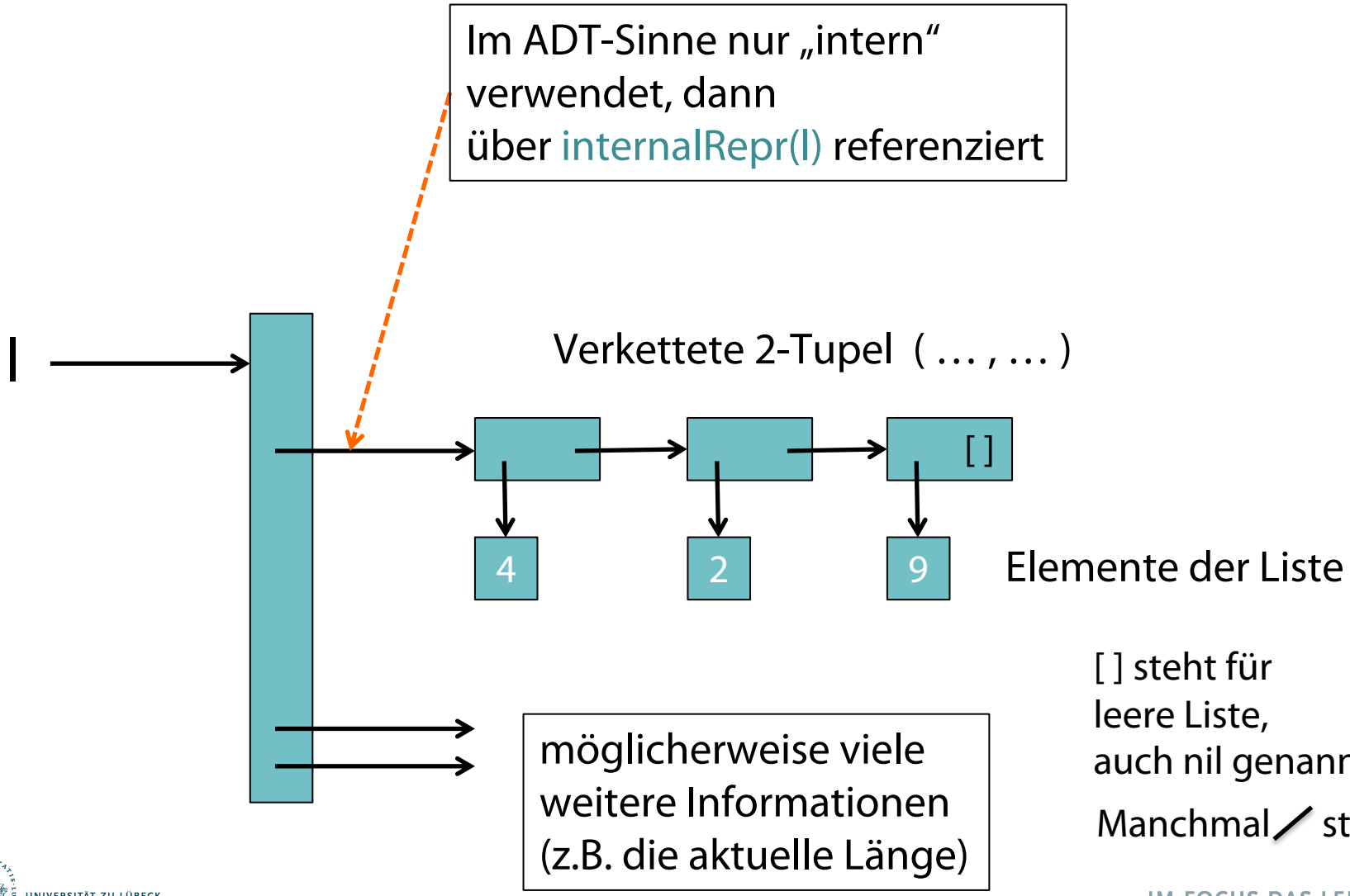
Operationen:

- function makeList() liefert neue Liste (am Anfang leer)
- procedure insert(e, l) fügt Element e am Anfang in Liste l ein, verändert l
- procedure delete(e, l) löscht Element e sofern enthalten, verändert l, wenn ein Element gelöscht wird
- function first(l) gibt Last-in-Element zurück (Fehler, wenn l leer)
- procedure deleteFirst(l) löscht Last-in-Element in l (Fehler, wenn l leer)
- function length(l) gibt Anzahl der Elemente in l zurück
- function mtList?(l) gibt true zurück, wenn l leer ist, sonst false

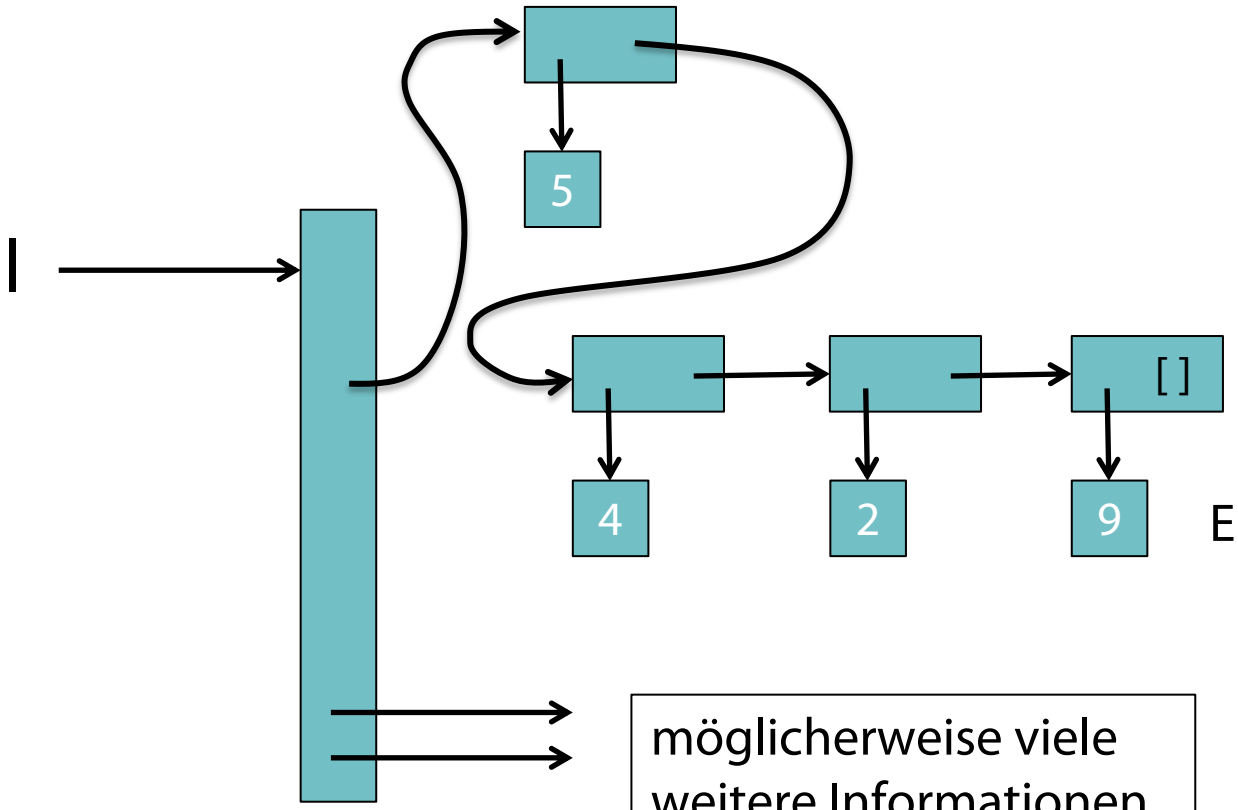
Iteration (last-in first-out):

- for e in l do ... *oder auch* for e ∈ l do ...

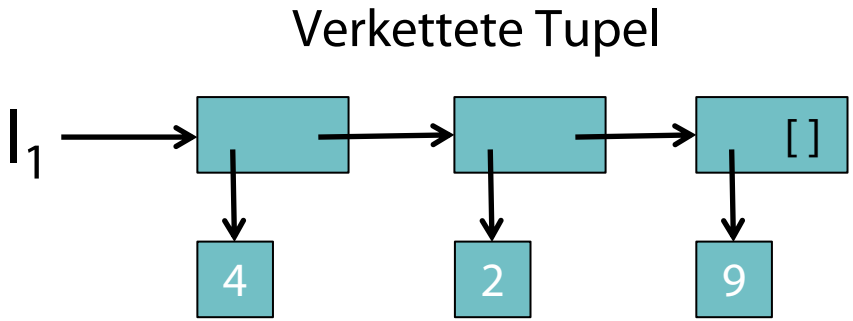
Listen intern (Beispiel)



insert(5, l)


l
5
4
2
9
[ ]
Elemente der Liste
möglicherweise viele
weitere Informationen
(z.B. die aktuelle Länge)

Listen als Glaskästen (“verkettete Liste”)



- Ausdruck (e, l) liefert Tupel mit Element e und Liste l
- Beispiele:
  - [4, 2, 9] = (4, (2, (9, [ ]))) [4] = (4, [ ])
- Sei l_1 = [4, 2, 9], dann Zugriff mit (e, l_2) := l_1
  dann gilt: e = 4 und l_2 = [2, 9] = (2, (9, [ ]))
- Zugriff auch über first(l) und rest(l)

Listen als Glaskästen

Notation: [4, 2, 9] oder [] bzw. () für die leere Liste

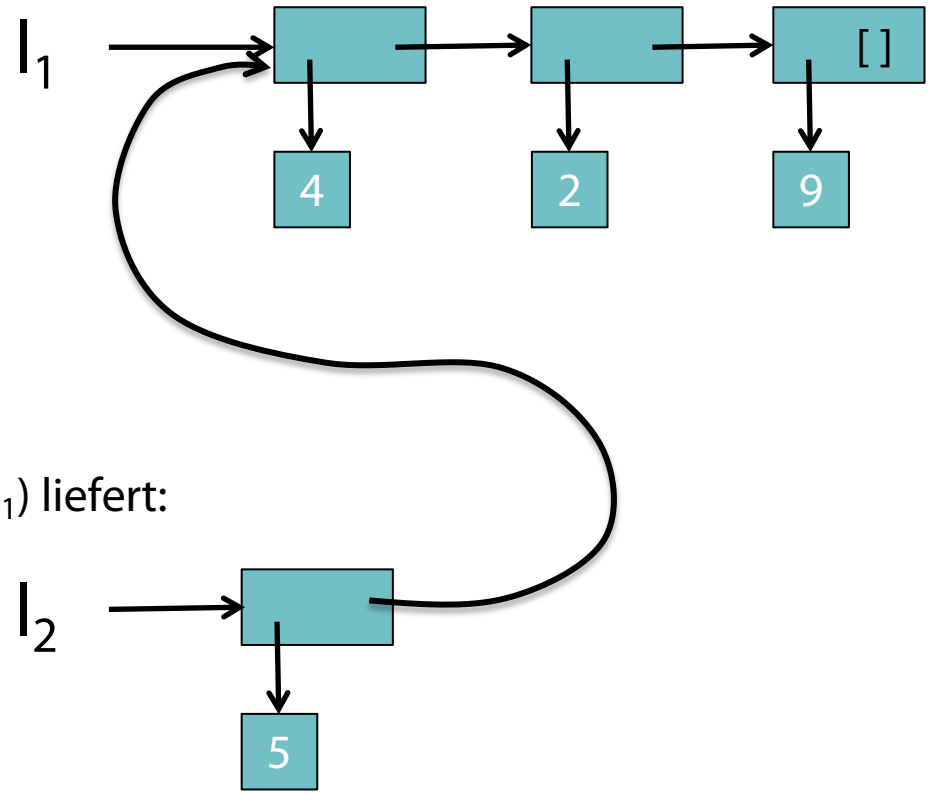
Operationen:

- function cons(e, l) fügt Element e am Anfang in Liste l ein,
  verändert l nicht, gibt eine neue, erweiterte Liste zurück
- function first(l) gibt die erste Komponente des Tupels zurück
  (Fehler, wenn l leer)
- function rest(l) gibt die zweite Komponente des Tupels zurück
  (Fehler, wenn l leer)
- function length(l) gibt Anzahl der Elemente in l zurück
- function mt?(l) gibt true zurück, wenn l = [ ] ist, sonst false

Iteration:

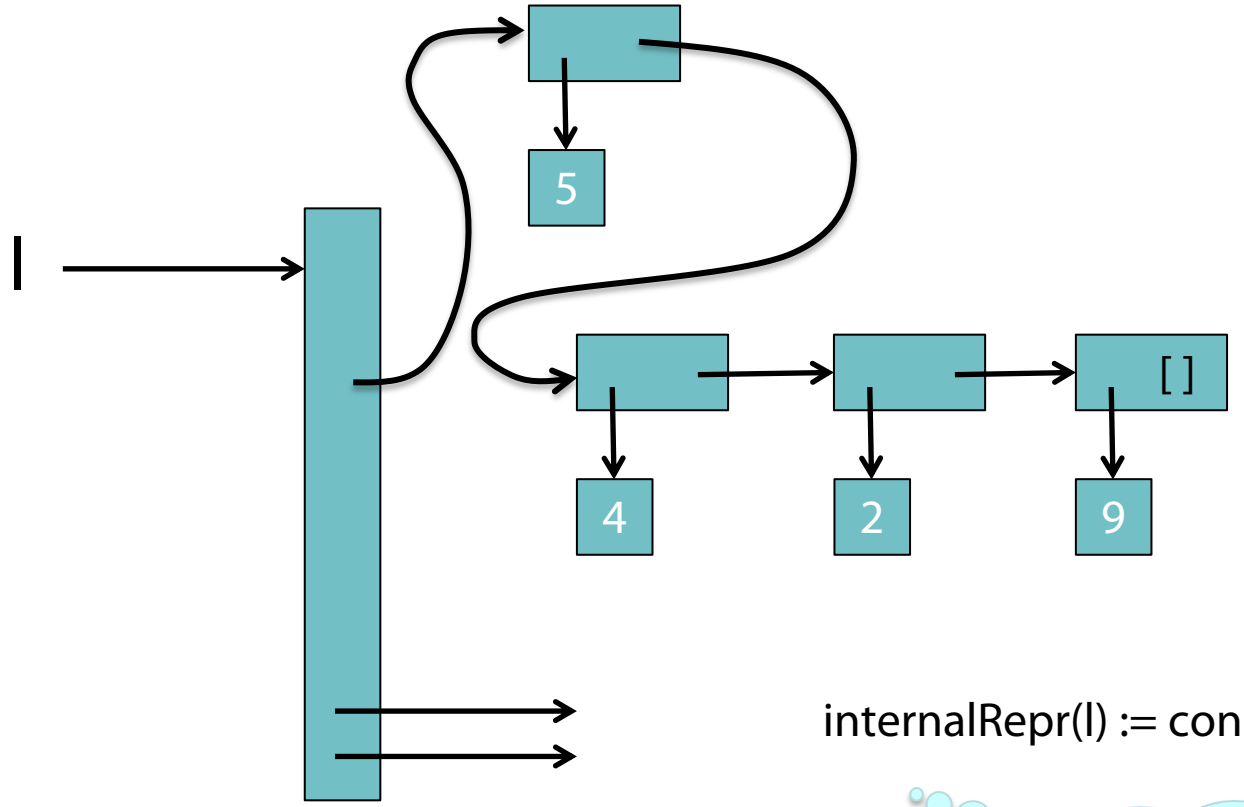
- for e in l do ... *oder auch* for e ∈ l do ...

Cons



$l_2 := cons(5, l_1)$

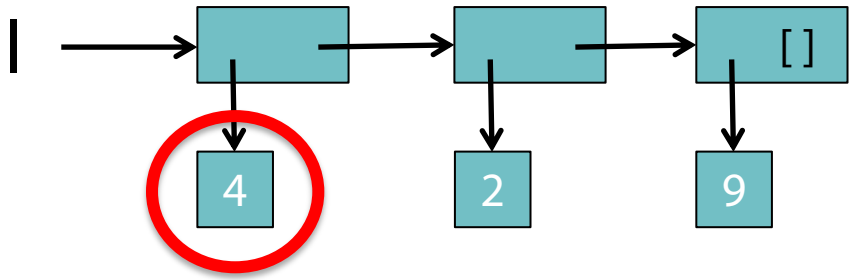
Wiederholung ADT-Listen: insert(5, l)



internalRepr(l) := cons(5, internalRepr(l))

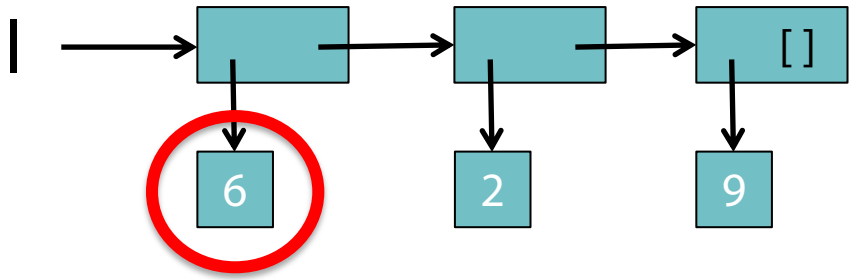
Aber von außen nicht sichtbar

Zugriff auf erste Komponente mit first



- first(l)

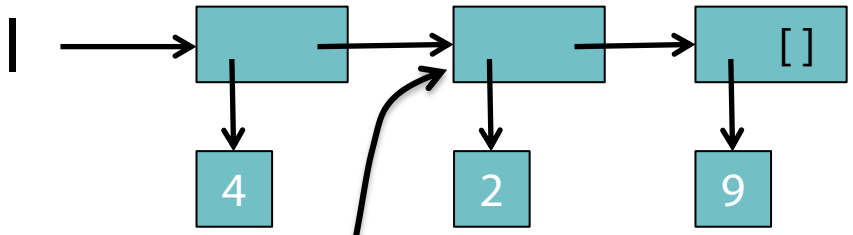
Manipulaton der ersten Komponente



- first(l) := 6

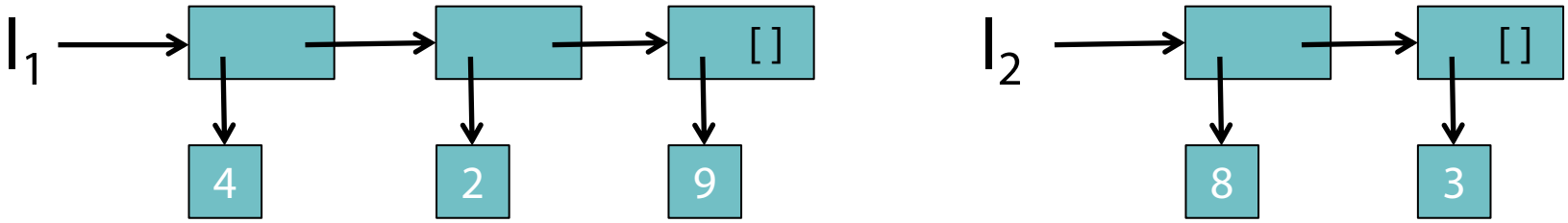
- Vergleiche das Setzen von Arrayelementen: A[i] := …

Zugriff auf zweite Komponente mit rest



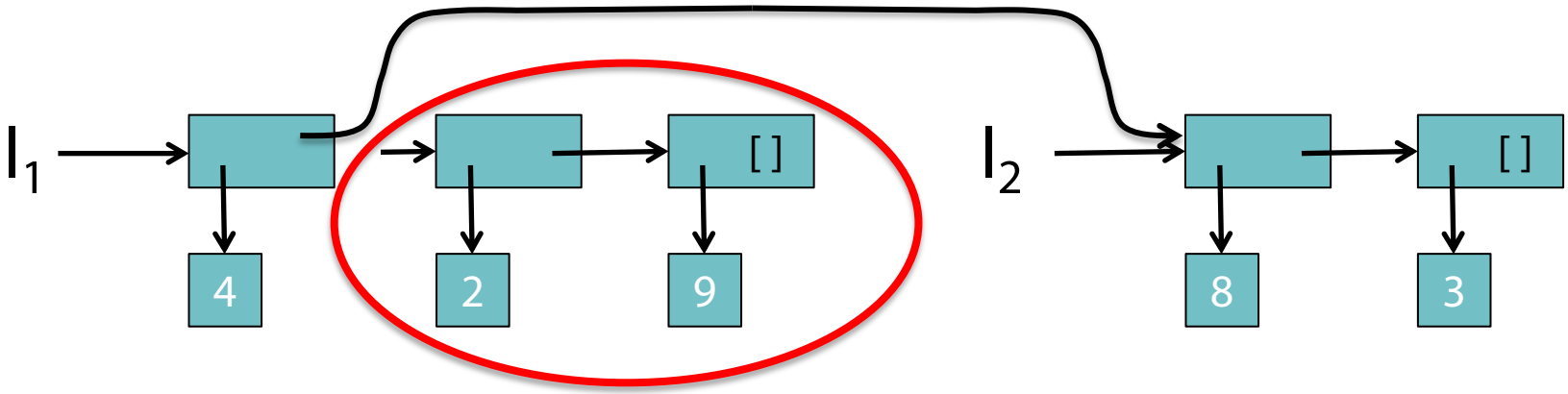
- rest(l)

Manipulaton der zweiten Komponente (1)



- Was bewirkt rest(l_1) := l_2 ?

Manipulaton der zweiten Komponente (2)



- rest(l_1) := l_2

Kellerspeicher / Stapelspeicher / Stack

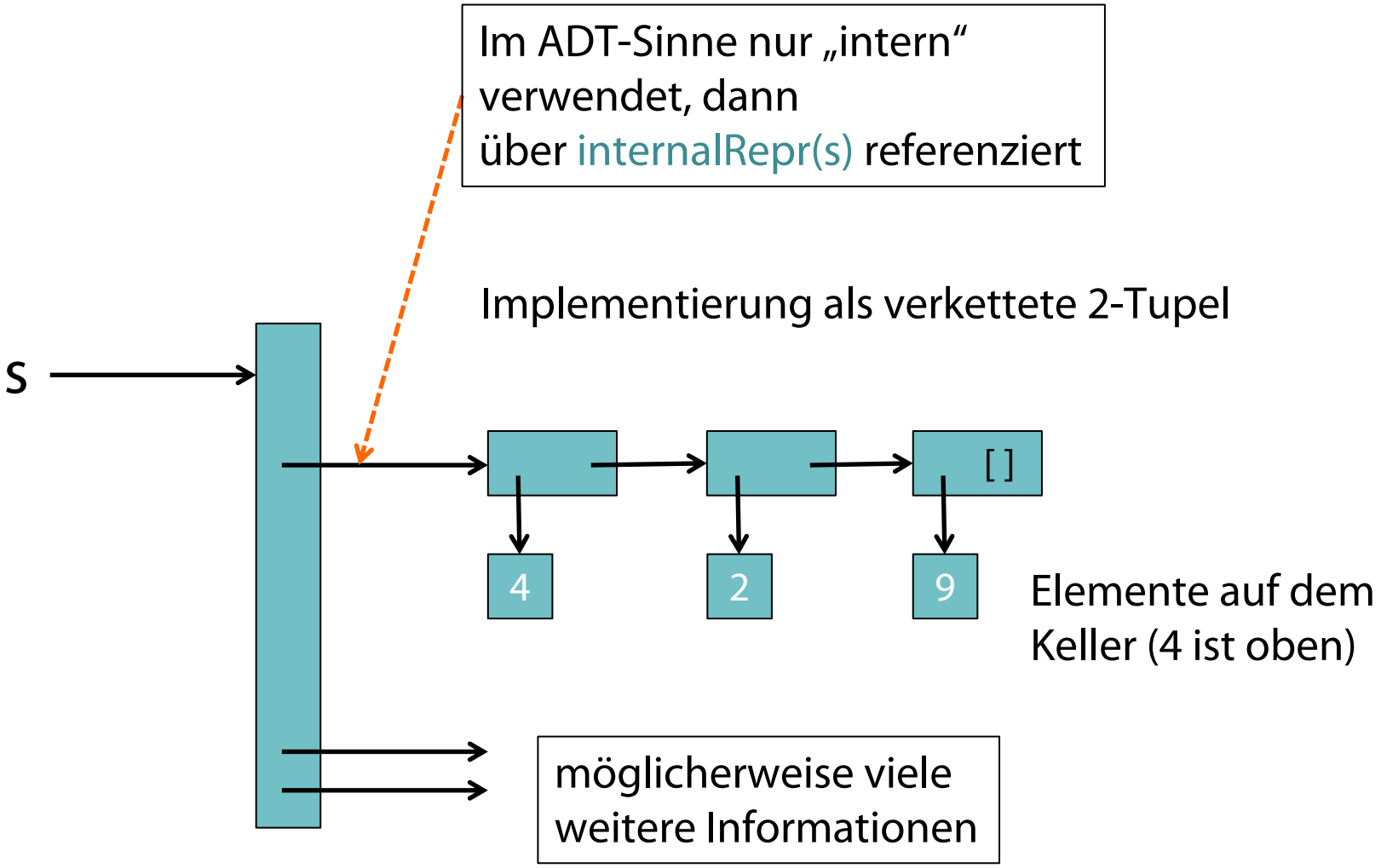
Notation: [4, 2, 9] (4 ist "oben")

Operationen:

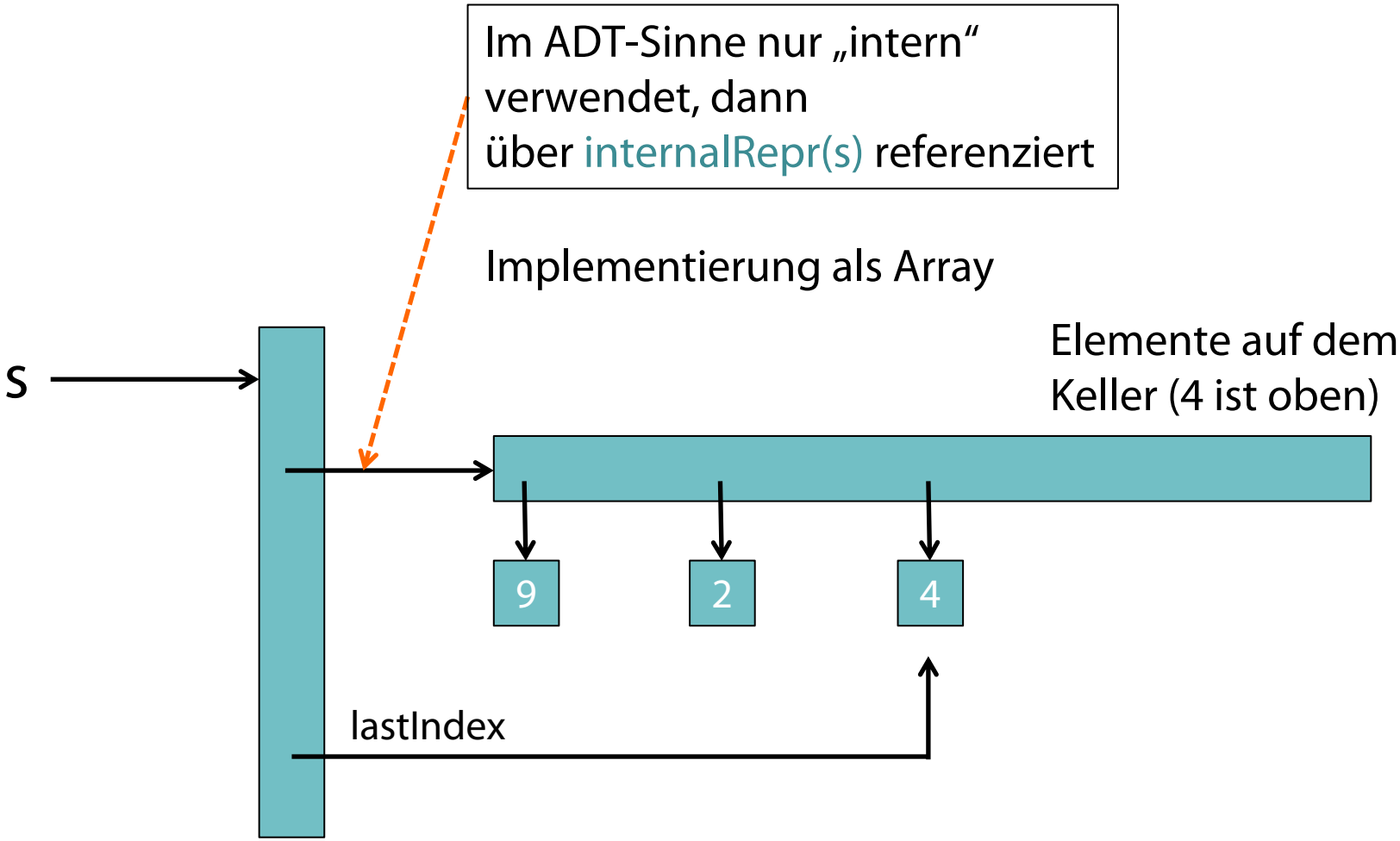
- function makeStack() liefert leeren Keller
- procedure push(e, s) fügt Element e oben in den Keller s ein, verändert s
- function top(s) gibt oberes Element zurück (Fehler, wenn s leer)
- procedure pop(s) löscht Top-Element in s (Fehler, wenn s leer)
- function mtStack?(s) gibt true zurück, wenn s leer ist, sonst false

Iteration: eigentlich nicht vorgesehen (aber im Prinzip realisierbar)

Keller intern (Beispiel: als Liste)



Keller intern (Beispiel 2: als Array)



Realisierung von Kellerspeichern

- Arrays
  - Größe muss vorher festgelegt werden
  - Keller kann “vollaufen”
  - Neues, größeres Array und Umkopieren
- Verkettete Liste
  - Weniger Speicherbedarf

Schlange / Queue (First-in-First-out-Speicher)

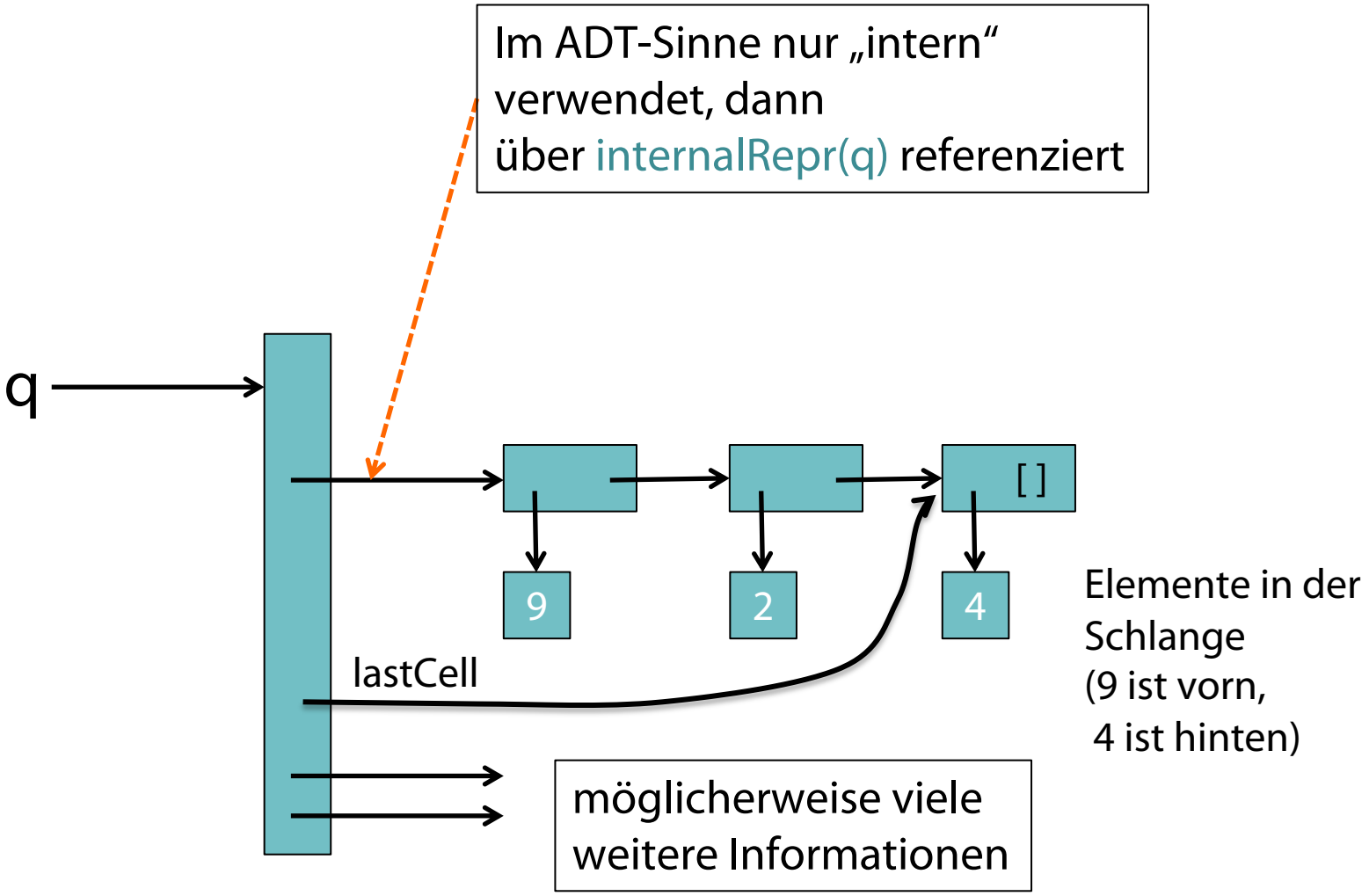
Notation: [4, 2, 9] (4 ist "hinten", 9 ist "vorn", kommt zuerst dran)

Operationen:

- function makeQueue() liefert leere Warteschlange
- procedure enqueue(e, q) fügt Element e hinten in die Schlange q ein, verändert q
- function next(q) gibt vorderes Element zurück, verändert q nicht
- function dequeue (q) gibt vorderes Element zurück, verändert q
- function mtQueue?(q) gibt true zurück, wenn q leer ist, sonst false

Iteration: nicht vorgesehen (evtl. wie Liste)

Queue intern (Beispiel)



Sortieren durch verallgemeinerte Gruppierung

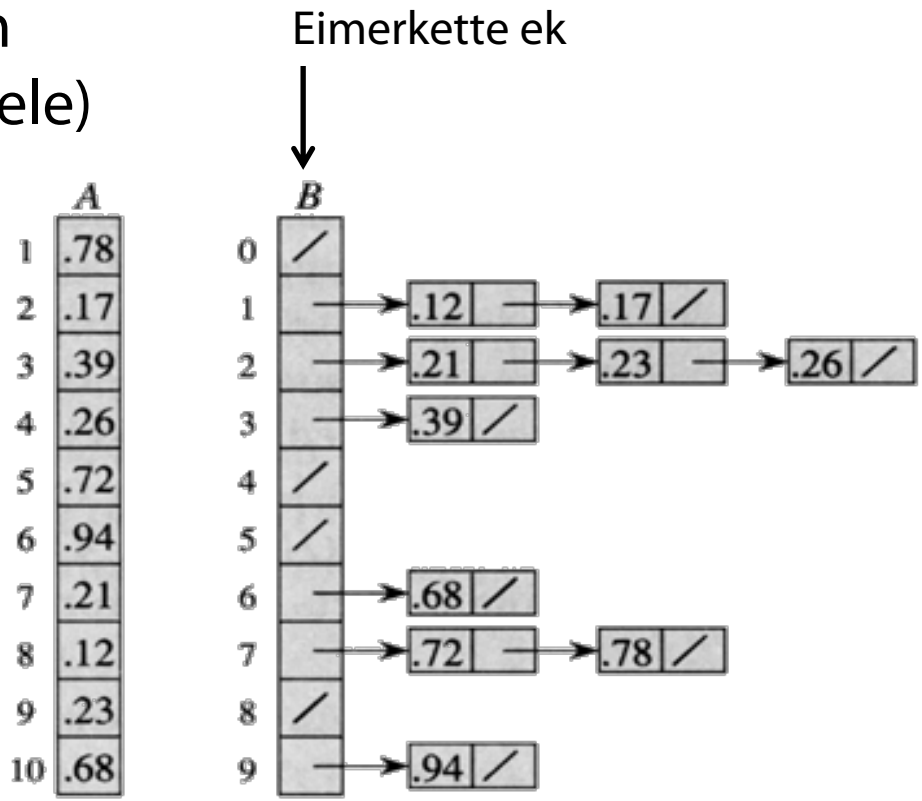
Bucket-Sort

1. procedure BUCKET-SORT (A)
2. $n \leftarrow$ length(A) , k ← Anzahl der Eimer
3. for $i = 1$ to n do
4. Füge $A[i]$ in den richtigen Eimer ein
5. for $i = 1$ to k do
6. Sortiere i-ten Eimer mit einer vergleichsbasierten Sortierfunktion
7. Hänge die Eimer in der richtigen Ordnung hintereinander

Bildquelle: Portugiesisches Wikipedia

Wie wollen wir die Eimerkette implementieren?

- Verkettete Liste oder Feld für Eimer**kette**?
- Verkettete Listen oder Felder für **Einzel**eimer?
  - Verkettete Listen sparen Platz (einige Eimer haben kaum Einträge, andere haben viele)
  - Aber mit verketteten Listen können wir "schnelle" Sortierverfahren wie Heap-Sort oder Quicksort nicht verwenden
  - **Sortierte Listen**!



Analyse von Bucketsort

- Sei $S(m)$ die Anzahl der Vergleiche für einen Eimer mit m Schlüsseln
- Setze n_i auf die Anzahl der Schlüssel im i-ten Eimer
- Gesamtzahl der Vergleiche = $\sum^{k}_{i=1} S(n_i)$ bei k Eimern

Analyse (2)

- Sei $S(m) \in O(m \log m)$
- Falls die Schlüssel gleichmäßig verteilt sind, beträgt die Eimergröße n/k
- Gesamtzahl der Vergleiche für all k Eimer

  $= k(n/k) \log(n/k)$

  $= n \log(n/k)$
- Falls $k=n/10$, dann reichen $n \log(10)$ Vergleiche (Laufzeit ist linear in n)

Analyse (3)

- Sei $S(m) \in O(m^2)$
- Falls die Schlüssel gleichmäßig verteilt sind, beträgt die Eimergröße n/k
- Gesamtzahl der Vergleiche für all k Eimer

  $= k(n/k)^2$

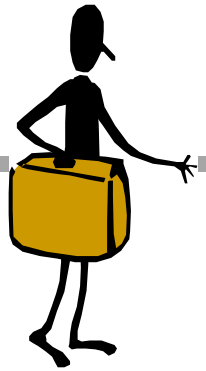
  $= n^2/k$
- Falls $k=n/\log(10)$, dann reichen $n \log(10)$ Vergleiche (Laufzeit ist linear in n, aber man muss mehr Speicher bereitstellen als bei $S(m) \in O(m \log m)$)

Lineare Sortierung: Einsicht

Je mehr man über das Problem weiß, desto eher kann man einen optimalen Algorithmus entwerfen

- Gesucht ist ein Verfahren **S**, so dass **{ P } S { Q }** gilt (Notation nach Hoare)
  - Vorbedingung: **P =?**
  - Invarianten („Axiome“): **I = ?**
  - Nachbedingung: **Q = ∀1≤i<j≤n: A[i] ≤ A[j]**
  - Nebenbedingungen: **?**

Zusammenfassung



- Sortieren durch Verteilen (lineares Sortieren)
  - In vorigen Einheiten:
    - Counting Sort
    - Radix Sort
  - Heute behandelt
    - Bucket Sort

- Wiederholung von elementaren Datenstrukturen
  - Listen, Keller, Warteschlangen