Algorithmen und Datenstrukturen

Sequenzen

ADT – Datenstruktur

Schnittstelle des Datentyps

Interpretation der Sorten- und Funktionssymbole

ADT

Verhalten des Datentyps

Implementierung

verwendbarer Datentyp

Datenstruktur

Sequenzen - informal

- **Beschreibung:** Folge von Elementen eines gemeinsamen **Grundtyps** (*z.B. Zahlen, Strings, ...*)
- **Wertebereich:** Menge aller endlichen Folgen von Elementen des Grundtyps

Interpretation

Interface

Operation	gibt zurück	Beschreibung
empty()	Sequenz	erzeugt eine leere Sequenz
length(*S*)	int	Länge der Sequenz *S*
insert(*S*,*x*,*p*)	Sequenz	fügt *x* in *S* an Position *p* ein
delete(*S*,*p*)	Sequenz	löscht Element an Position *p* aus *S*
get(*S*,*p*)	Grundtyp	Wert des Elements an Position *p* in *S*
concat(*S*,*R*)	Sequenz	fügt Sequenzen *S* und *R* aneinander

ADT Sequence

empty()	Sequenz
length(*S*)	int
insert(*S*,*x*,*p*)	Sequenz
delete(*S*,*p*)	Sequenz
get(*S*,*p*)	Grundtyp
concat(*S*,*R*)	Sequenz

type Sequence =

sorts T, int, seq

functions

empty: $\rightarrow$ seq

length : seq $\rightarrow$ int

insert: seq $\times$ T $\times$ int $\rightarrow$ seq

delete: seq $\times$ int $\rightarrow$ seq

get: seq $\times$ int $\rightarrow$ T

concat: seq $\times$ seq $\rightarrow$ seq

end.

Interpretation durch Bedingungen (1)

- **T:** Datentyp ***I*(T)** des Grundtyps
- **int:** Datentyp der ganzen Zahlen
- **seq:** Menge aller endlichen Folgen von Elementen aus **T** mit:
- **empty() [empty: $\rightarrow$ seq]**
  - **post:** eine neue leere Sequenz ist erzeugt
- **length(*S*) [length : seq $\rightarrow$ int]**
  - **post:** Anzahl der Elemente des Grundtyps in *S*
- **insert(S, *x*, *p*) [insert: seq $\times$ T $\times$ int $\rightarrow$ seq]**
  - **pre:** $0 \leq p \leq$ length(*S*) (*für 0-indizierte Sequenzen*)
  - **post:** *x* ist an Position *p* in *S* eingefügt. Falls $p <$ length(*S*), so sind alle Elemente von *S* ab Position *p* um eine Position nach rechts verschoben.

Interpretation durch Bedingungen (2)

- **delete(*S*, *p*) [delete: seq × int → seq]**
  - **pre:** $0 \leq p < \text{length}(S)$ (*für 0-indizierte Sequenzen*)
  - **post:** Element von *S* an Position *p* ist aus *S* entfernt. Elemente von *S* ab Position *p*+1 sind um eine Position nach links verschoben.
- **get(*S*, *p*) [get : seq × int → T]**
  - **pre:** $0 \leq p < \text{length}(S)$ (*für 0-indizierte Sequenzen*)
  - **post:** Element in *S* an Position *p*
- **concat(S, *R*) [concat: seq × seq → seq]**
  - **pre:** *S* und *R* haben den gleichen Grundtyp
  - **post:** Sequenz mit length(*S*)+length(*R*) Elementen, in der auf die Elemente von *S* die Elemente von *R* folgen

Implementierungen des Typs Sequence

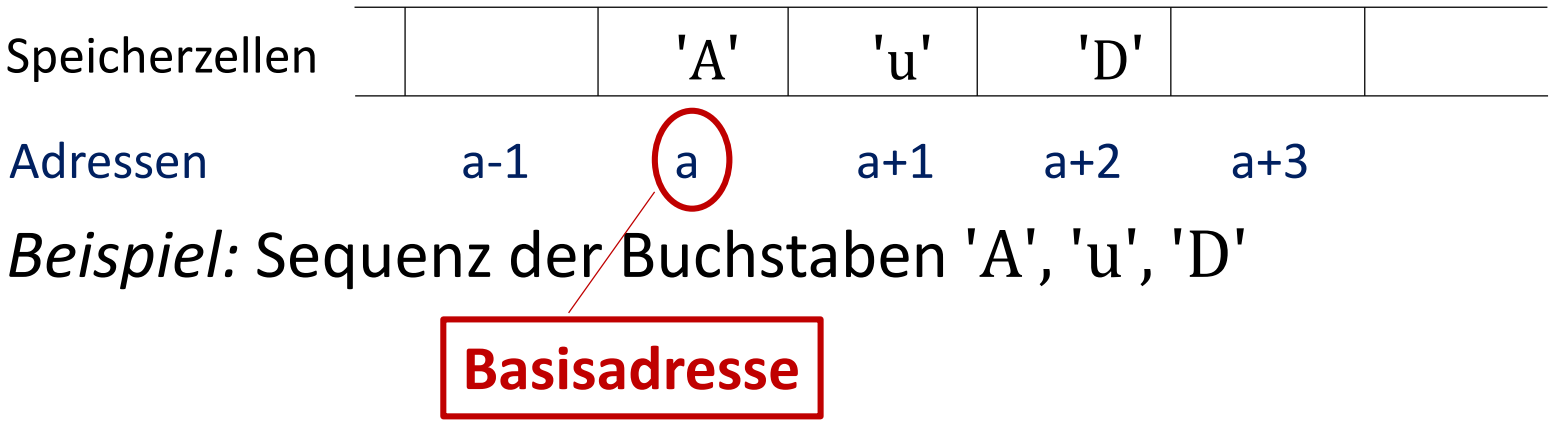
Es existieren verschiedene Implementierungen des ADT Sequenz, *u.a.*

- Rechtssequenzen
- Linkssequenzen
- Array
- einfach verkettete Liste
- doppelt verkettete Liste

Arrays (Felder)

In den meisten Implementierungen (Sprachen):

- Elemente der Sequenz werden im Speicher an aufeinanderfolgenden Adressen abgelegt.
- Aus der ersten Adresse und dem Index (Position) werden die Adressen errechnet, an denen die anderen Elemente gespeichert sind **(Adressfunktion).**



- *Beispiel:* Sequenz der Buchstaben 'A', 'u', 'D'

Variablen – Adressen - Werte

- Jede **Variable** wird an einer ihr eindeutig zugeordneten **Adresse** (im Hauptspeicher) gespeichert.
  → *s. Speicherbelegungstabelle (GdP)*

Variable	x	y
Adresse	4	8

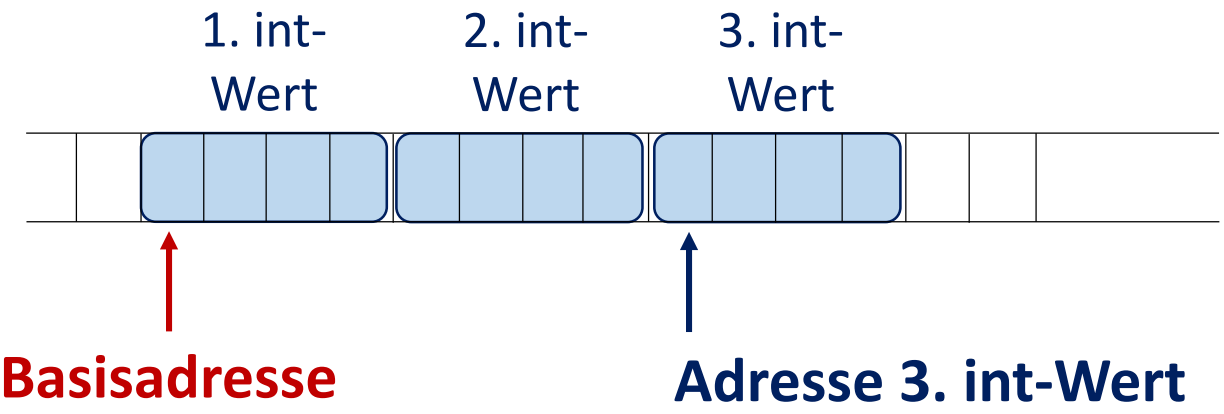
- Weiterhin hat die Variable
  (nach ihrer Initialisierung) zu jedem Zeitpunkt einen **Wert**.

- Bei Variablenzugriffen muss zuerst die richtige Stelle im Speicher (**Adresse**) bestimmt werden, dann kann der dort abgelegte **Wert** bestimmt oder verändert werden.

- *Beispiel* Lesen des Wertes der Variablen *x*
  Variablenname *x* ⟶ **Adresse** (*hier* 4) ⟶ **Wert** (*z.B.* 17)
  Inhaltsbestimmung
  Dereferenzieren

Arrays (2)

- Grundtyp **T** bestimmt, wie viele benachbarte Speicherzellen für ein Element gebraucht werden.
  → **Breite** des Typs **T: size(T)**
- *Beispiel:* ganze Zahlen vom Typ **int** werden oft durch vier Byte kodiert
  → *in vielen Rechnerarchitekturen vier Speicherzellen*

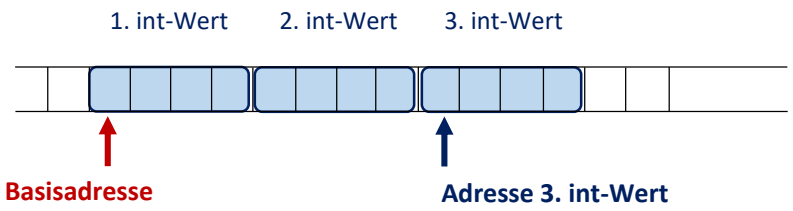


Arrays (3)

- Für Zugriff auf Element mit Index p (get(A,p), `A[p]` in Python):
  1. Berechnung der Adresse des Elements mit Index p
  2. Ermittlung des an dieser Adresse gespeicherten Wertes

zu 1) Adressfunktion für ein Array A (0-*indiziert*)

- mit $\boldsymbol{n}$ Elementen
- eines Typs **T** der Breite **size(T)**
- an Basisadresse $\boldsymbol{\alpha}$



ordnet jeder Position (Index) p, $0 \leq p < n$,
die Adresse des (p+1)-ten Elements zu: $\boldsymbol{\alpha_A(p) = \alpha + p \cdot \text{size(T)}}$

zu 2) an Adresse gespeicherter Wert: **content($\alpha_A(p)$)**

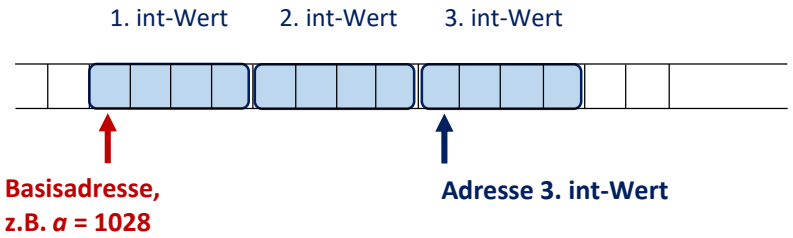
Arrays - Beispiel

- Für Zugriff auf Element mit Index *p*:
  1. Berechnung der Adresse des Elements mit Index *p*
  2. Ermittlung des an dieser Adresse gespeicherten Wertes

- *Beispiel:* Zugriff auf das dritte Element `A[2]= 17`:

$$\alpha_A(p) = a + p \cdot \text{size(T)}$$
$$= 1028 + 2 \cdot 4$$
$$= 1036$$

content(1036) = 17



Arrays mit definierbarem Indexbereich

- Manche Programmiersprachen erlauben den Indexbereich von Arrays festzulegen (*low* ... *high*)
- **Adressfunktion** für ein Array *A:*
  - mit Indizes von *low* bis *high*
  - eines Typs **T** der Breite **size(T)**
  - an Basisadresse ***a***

ordnet jeder Position *p* (Index)
die Adresse des (*p*-low+1)-ten Elements zu:

$$\alpha_A(p) = a + (p - low) \cdot \text{size(T)}$$

Mehrdimensionale Arrays

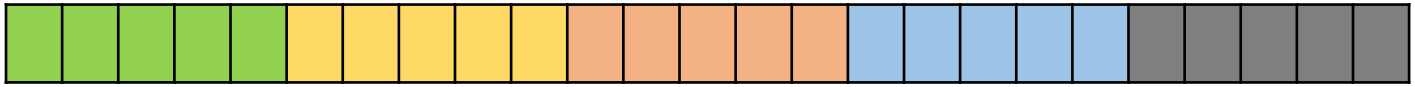
- hier nur zweidimensional (*leicht zu verallgemeinern*)
- Matrix

`S[0][0]`	`S[0][1]`	`S[0][2]`	`S[0][3]`	`S[0][4]`
`S[1][0]`	`S[1][1]`	`S[1][2]`	`S[1][3]`	`S[1][4]`
`S[2][0]`	`S[2][1]`	`S[2][2]`	`S[2][3]`	`S[2][4]`
`S[3][0]`	`S[3][1]`	`S[3][2]`	`S[3][3]`	`S[3][4]`
`S[4][0]`	`S[4][1]`	`S[4][2]`	`S[4][3]`	`S[4][4]`

- in vielen Sprachen `S[i,j]` statt `S[i][j]`

Zeilendominierte Architektur

S[0][0]	S[0][1]	S[0][2]	S[0][3]	S[0][4]	S[1][0]	...



-- 1. Zeile -- i=0 | -- 2. Zeile -- i=1 | -- 3. Zeile -- i=2 | -- 4. Zeile -- i=3 | -- 5. Zeile – i=4

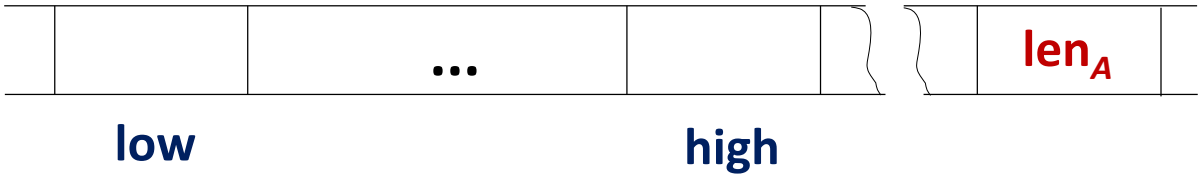
- **Adressfunktion:** Adresse von `S[i][j]` in einem Array der Größe $m \times n$ (mit $0 \leq i < m$, $0 \leq j < n$):

$$\alpha_A(i,j) = a + ((i \cdot n) + j) \cdot \text{size(T)}$$

- analog spaltendominierte Architektur

Statische *versus* dynamische Arrays

- **statisch:** Größe ($\mathbf{len_A}$) des Arrays *A* ist unveränderlich
  - Anzahl der Elemente wird beim Erzeugen festgelegt und gespeichert, *z.B.*



  - Zugriff auf Arrayelemente außerhalb der Grenzen erkennbar
- **dynamisch:** Größe kann zur Laufzeit beeinflusst werden
  - obere Grenze kann Variable sein *oder*
  - Arrays können zu jeder Zeit verlängert werden
  - oft wird aktuelle Länge gespeichert (*Fehlererkennung*)

Array-Implementierung der Operationen des ADT Sequenz

Zeiger

- Implementierungen verwenden den Datentyp Zeiger/Pointer

- Wertebereich: **Adr** ∪ {**null**}
  - **Adr:** Menge aller Adressen von Speicherzellen, die das Programm verwenden kann
  - **null:** spezieller Wert, der für keine verwendbare Speicheradresse steht

Zeiger: Funktionen

- **Dereferenzieren: content(zeiger)**
  - Zugriff auf Wert an der in **zeiger** gemerkten Adresse
  - ***Achtung:* null *kann nicht dereferenziert werden (Fehler):***
    content(null) ist nicht definiert!

- **Wertänderung: setContent(zeiger, wert)**
  - ändert den Inhalt/Wert, der an der in **zeiger** gemerkten Adresse gespeichert ist, auf **wert**
  - ***Achtung:* Wert von null kann nicht gesetzt werden *(Fehler):***
    setContent(null, wert) ist für keinen Wert wert zulässig!

Referenzieren

- **address(*x*)**
  - liefert Adresse zu
    - einer Variablen ***x***
    - oder einem Speicherobjekt ***x***
  - liefert niemals null

Variable	x	y
Adresse	4	8

Implementierung des Interface des ADT Sequence mit Arrays (1)

- Beschränkung auf 0-indizierte Arrays mit dynamischen Grenzen für Elemente eines gegebenen Grundtyps **T**

- **empty() [empty: → seq]**
  - **post:** eine neue leere Sequenz ist erzeugt

  len ⟵ 0
  a ⟵ eine geeignete Basisadresse

- **length(*S*) [length : seq → int]**
  - **post:** Anzahl der Elemente des Grundtyps in *S*

  gib len_S aus

Implementierung des Interface des ADT Sequence mit Arrays (2)

- **insert(S, x, p) [insert: seq × T × int → seq]**
  - **pre:** $0 \leq p \leq \text{length}(S)$
  - **post:** x ist an Position p in S eingefügt. Falls $p < \text{length}(S)$, so sind alle Elemente von S ab Position p um eine Position nach rechts verschoben.

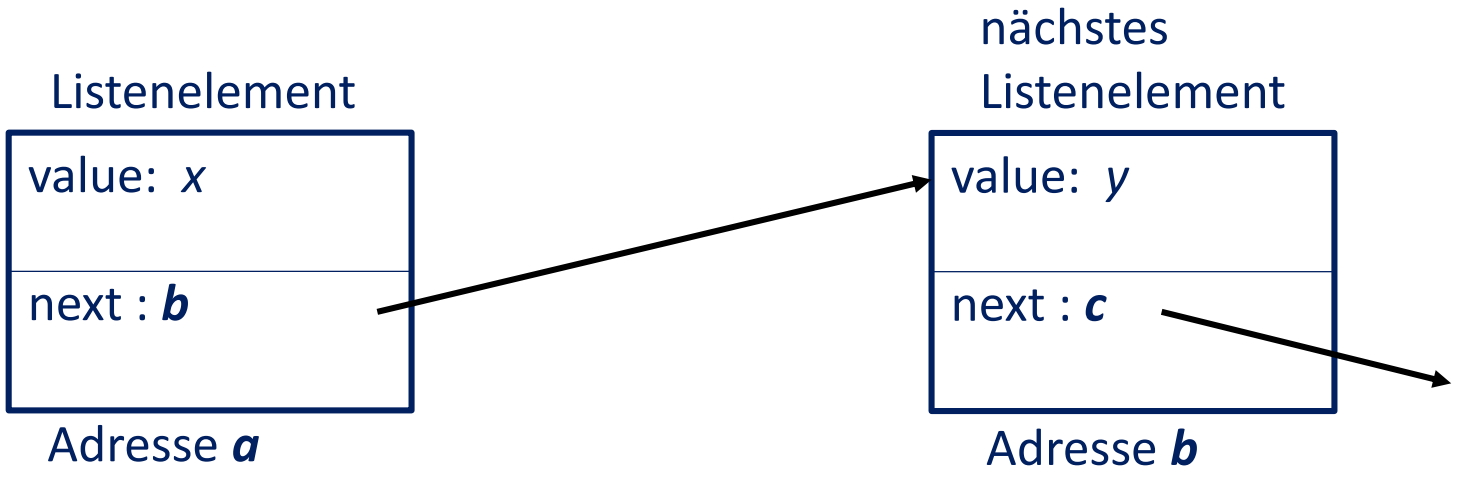
$i \longleftarrow \text{length}(S) - 1$
solange $i \geq p$
    setContent($\alpha_S(i)$+size(T), content($\alpha_S(i)$))
    $i \longleftarrow i - 1$
setContent($\alpha_S(p)$, x)
$\text{len}_S \longleftarrow \text{length}(S) + 1$
gib S aus

Weitere Operationen in den Übungen

Verkettete Listen

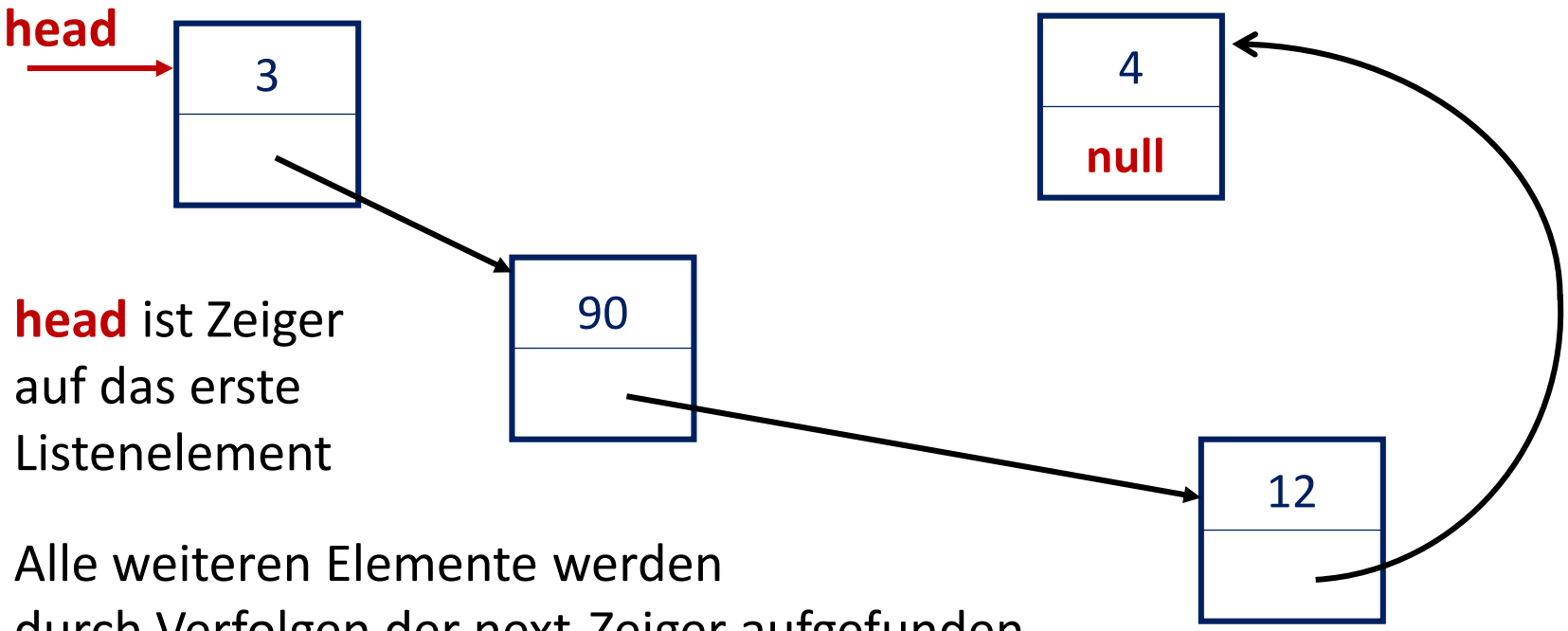
Einfach verkettete Liste

- Elemente werden im Speicher an nicht zusammenhängenden Stellen abgelegt.
- Jedes Element merkt sich
  - den Wert des Grundtyps und
  - die Adresse des nächsten Elements (**Zeiger**).



Beispiel einfach verkettete Liste

- Liste [3, 90, 12, 4]



head ist Zeiger auf das erste Listenelement

Alle weiteren Elemente werden durch Verfolgen der next-Zeiger aufgefunden.
→ **head** „ist“ die Liste

Listenelement als ADT (Interface)

type ListElement =

sorts T, p, le

functions

$$\begin{aligned} &\text{new: } T \to le \\ &\text{getValue : } le \to T \\ &\text{setValue: } le \times T \to le \\ &\text{getNext: } le \to p \\ &\text{setNext: } le \times p \to le \end{aligned}$$

end.

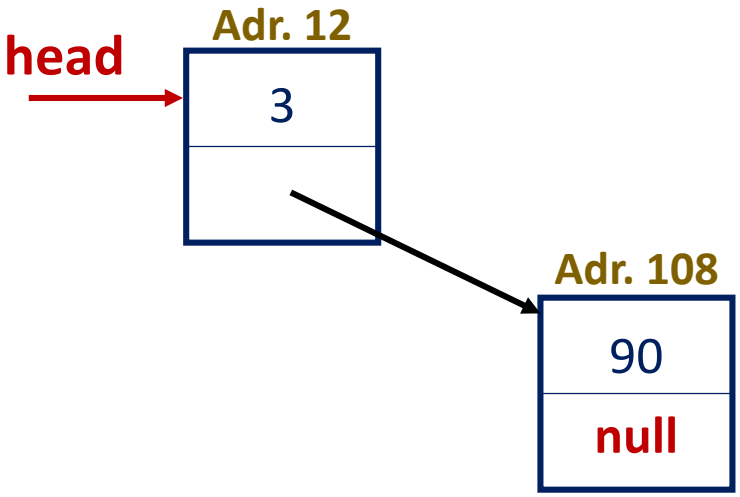
Listenelement: Interpretation

- **T:** Datentyp ***I*(T)** des Grundtyps
- **p:** Datentyp Zeiger (mit Werten Adr ∪ {null})
- **le:** Werte aus *I*(T) × (Adr ∪ {null}) mit:
- **new(*x*)** erzeugt **(*x*, null)** **[new: T → le]**
- **getValue((*x*,*nle*)) = *x*** **[getValue : le → T]**
- **setValue((*x*,*nle*), *y*) = (*y*,*nle*)** **[setValue: le × T → le]**
- **getNext((*x*,*nle*)) = *nle*** **[getNext: le → p]**
- **setNext((*x*,*nle*), *n*) = (*x*, *n*)** **[setNext: le × p → le]**

Eine Liste ist ein Zeiger auf ein (das erste) Listenelement (oder null für die leere Liste): head

Beispiel einfach verkettete Liste, Forts.

- Liste [3, 90]



head = 12
content(head) = (3, 108)
getValue(content(head)) = 3
getNext(content(head)) = 108

content(108) = (90, null)
address((90,null)) = 108

Implementierung des Interface des ADT Sequence mit Listen (1)

- **empty() [empty: → seq]**
  - **post:** eine neue leere Sequenz ist erzeugt

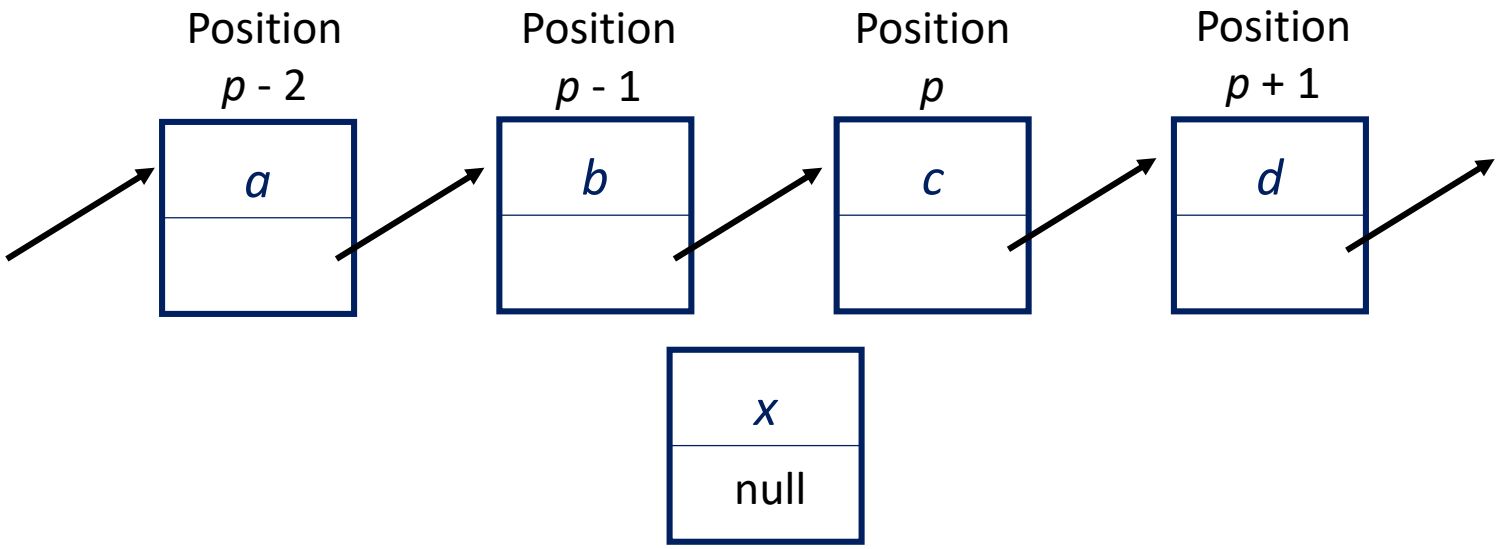
```
head ⟵ null
gib head aus
```

- **length(*S*) [length : seq → int]**
  - **post:** Anzahl der Elemente des Grundtyps in *S*

```
len ⟵ 0
akt ⟵ head      # (akt auf head von S setzen; ist ein Zeiger)
solange akt nicht null ist
        len ⟵ len + 1
        akt ⟵ getNext(content(akt))      # (Zeiger akt auf das nächste
                                           Element setzen)
gib len aus
```

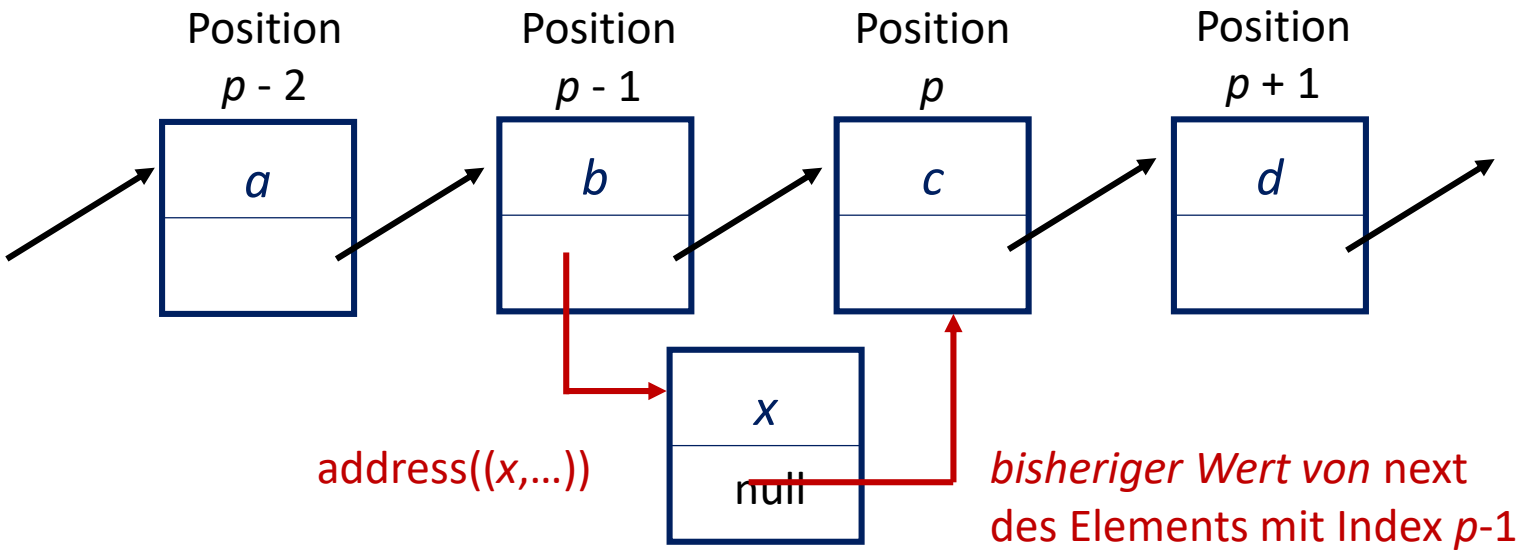
Implementierung des Interface des ADT Sequence mit Listen (2)

- **insert(*S*, *x*, *p*) [insert: seq × T × int → seq]**
  - **pre:** $0 \leq p \leq \text{length}(S)$
  - **post:** *x* ist an Position *p* in *S* eingefügt. Falls $p < \text{length}(S)$, so sind alle Elemente von *S* ab Position *p* um eine Position nach rechts verschoben.



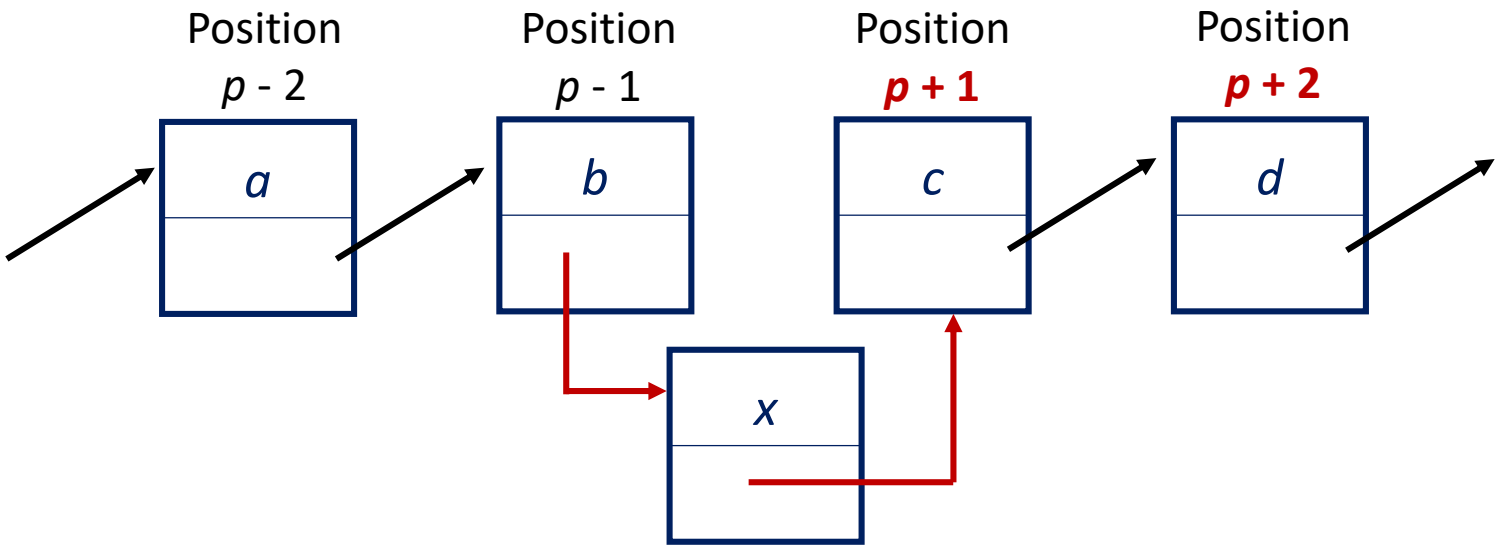
Implementierung des Interface des ADT Sequence mit Listen (2)

- **insert(*S*, *x*, *p*) [insert: seq × T × int → seq]**
  - **pre:** $0 \leq p \leq \text{length}(S)$
  - **post:** *x* ist an Position *p* in *S* eingefügt. Falls $p < \text{length}(S)$, so sind alle Elemente von *S* ab Position *p* um eine Position nach rechts verschoben.



Implementierung des Interface des ADT Sequence mit Listen (2)

- **insert(*S*, *x*, *p*) [insert: seq × T × int → seq]**
  - **pre:** $0 \leq p \leq \text{length}(S)$
  - **post:** *x* ist an Position *p* in *S* eingefügt. Falls $p < \text{length}(S)$, so sind alle Elemente von *S* ab Position *p* um eine Position nach rechts verschoben.

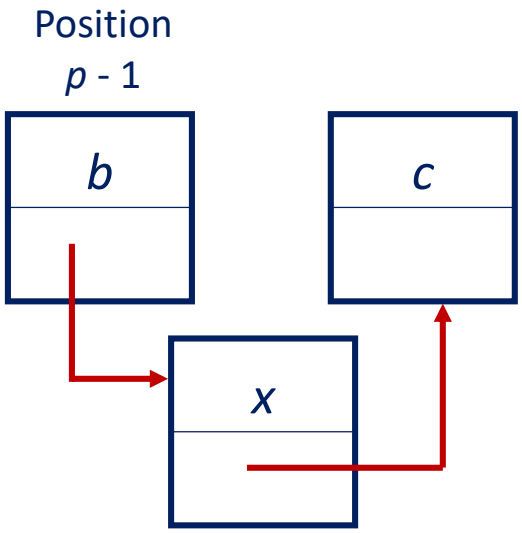


insert(*S*, *x*, *p*) – Entwurf

1. Erzeuge ein neues Listenelement *n* mit Wert *x*
2. Gehe von **head** über die **next**-Zeiger zum Element mit Index *p*-1
3. Setze den **next**-Zeiger des neuen Listenelements *n* auf den Wert des **next**-Zeigers des Elements mit Index *p*-1
4. Setze den Wert des **next**-Zeigers des Elements mit Index *p*-1 auf die Adresse des neuen Listenelements *n*

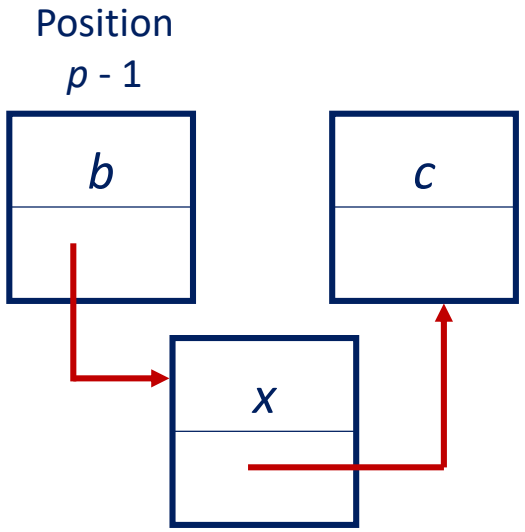
Ausnahme: p = 0:

2. Setze den **next**-Zeiger des neuen Listenelements *n* auf den Wert von **head**
3. Setze **head** auf die Adresse des neuen Listenelements *n*



insert(*S*, *x*, *p*) – Pseudocode

```
n ⟵ new(x)     # neues Listenelement mit Wert x
falls p = 0
    setNext(n, head)
    head ⟵ address(n)
sonst
    count ⟵ 0
    akt ⟵ head
    solange count < p - 1
            count ⟵ count + 1
            akt ⟵ getNext(content(akt))
    setNext(n, getNext(content(akt)))
    setNext(content(akt), address(n))
gib S aus
```



Weitere Operationen in den Übungen

Doppelt verkettete Liste

- Liste kann vorwärts und rückwärts durchlaufen werden
- Listenelemente:
  - Zeiger auf Vorgängerelement (**prev**)
  - Wert des Grundtyps **T**
  - Zeiger auf Nachfolgerelement (**next**)
- Liste bestimmt durch zwei Zeiger:
  - Zeiger auf das erste Element (**head**)
  - Zeiger auf das letzte Element (**tail**)
- prev(content(head)) = next(content(tail)) = null

Doppelt verkettete Liste anschaulich

- D-Listenelement:

- D-Liste: