



# Informatik I

## Einfache Datenstrukturen

G. Zachmann  
Clausthal University, Germany  
[zach@in.tu-clausthal.de](mailto:zach@in.tu-clausthal.de)



## Motivation



- **Datenstrukturen** sind höheres Organisationskonzept
  - Vgl. die bislang behandelten Binärcodierungen elementarer Datentypen
- Algorithmen & Datenstrukturen sind 2 Seiten derselben Medaille!
- In diesem Abschnitt **Grundbausteine von Datenstrukturen**
  - Einige höhere Datenstrukturen werden später behandelt werden
- Ein „klassisches Buch“



Niklaus Wirth: *Algorithmen und Datenstrukturen*; Pascal Version, 5. Auflage, Teubner, 1999.



## Array



- Ein **eindimensionales Array** besteht aus einer bestimmten Anzahl von Datenelementen
  - Elemente haben gleichen Typ → **homogenes** Array (C, Java, allg. eher in statisch typisierten Sprachen)
  - Verschiedenen Typ → **inhomogenes** Array (Python, Smalltalk, ..., allg. eher in dynamisch typisierten Sprachen)

- Beispiel: Vektor, Zeile oder Spalte einer Tabelle

- Z.B. Abtastung eines Signals zu konstanten Zeitintervallen

Zeitpunkt	1	2	3	4	...	30	31
Signalstärke	10.5	10.5	12.2	9.8	...	13.1	13.3

- Elemente werden indiziert, d.h., Identifikation und Zugriff erfolgt über **Index** = ganze Zahl  $\geq 0$  (typ. der Form **a [i]**)
- Auf jedes Element des Array kann mit demselben, **konstanten Zeitaufwand** zugegriffen werden



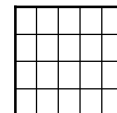
## Mehrdimensionale Arrays



- **Zweidimensionale Arrays** speichern die Werte mehrerer eindimensionaler Zeilen in Tabellen-(Matrix-)Form

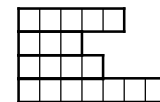
- Syntax: `a[i][j]`

- Analog  $n$ -dimensionale Arrays



- **Array von Arrays**

- ist auch 2-dim. Datenstruktur
- Nicht notw. quadratisch
- In den meisten Sprachen anders zu erzeugen / zuzugreifen / implementiert als (quadratisches) 2-dim. Array



- In Python gibt es eigtl. nur letzteres; in C++ gibt es beides



## Mathematische Interpretation



- Array = Funktion  $A : \mathbb{N} \mapsto T$ ,  $T = \text{Typ des Arrays (= der Elemente)}$
- Beispiel: eine Funktion  $t : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \mapsto \mathbb{R}$ , die einem Koordinatentripel einen Temperaturwert zuordnet (Wettersimulation)
  - Wert der Funktion an der Stelle  $(1,1,3)$ , also  $t(1,1,3)$ , findet sich dann in  $t[1][2][3]$
- Arrays eignen sich in der Praxis grundsätzlich nur dann zur Speicherung einer Funktion, wenn diese **dicht** ist, d.h., wenn die Abbildung für die allermeisten Indexwerte definiert ist
  - Sonst würde eine Arraydarstellung viel zuviel Platz beanspruchen
  - Außerdem geht dies nur für endliche Funktionen
- Wichtiger Spezialfall : strings = array of char
  - Viele Programmiersprachen haben dafür eigene Syntax / Implementierung



## Zeit-Aufwand für elementare Operationen



- Annahme: Array enthält N Elemente
- Element Nr i lesen: konstant [  $O(1)$  ]
- Element an Position i einfügen:  $\sim N$  [  $O(N)$  ]
- Element Nr i löschen:  $\sim N$  [  $O(N)$  ]
- Array löschen: konstant [  $O(1)$  ]



## Records, Structs, {Klassen} (Verbunde)



- Oft bestehen aber auch **Beziehungen zwischen Werten unterschiedlichen Typs**
  - Etwa zwischen Name und Monatsverdienst eines Beschäftigten
- Wir verbinden zusammengehörige Daten unterschiedlichen Typs zu einem **Verbund** = *record, struct*, Klasse
- Einzelteile eines Records / Structs / Klasse heißen **Attribute** oder **Members**
- Beispiel: Stammdaten

Name	"Mustermann"
Vorname	"Martin"
GebTag	10
GebMonat	05
GebJahr	1930
Familienstand	"verheiratet"
...	...



- Übliche Syntax zur Auswahl: Punkt-Notation
  - Beispiel: **s.name** oder **s.birthday**
  - Manchmal auch Pfeil-Notation: **s->name** oder **s->birthday**
- Komponenten eines Verbunds können von beliebigem Typ sein
- Also auch wieder Verbunde, Arrays, etc.
- Seien  $T_1, \dots, T_n$  die Typen der Members, dann hat der Record/Struct den (algebraischen) Typ  $T_1 \times \dots \times T_n$





## Verkettete Strukturen (linked structures)



"The name of the song is called 'Haddocks' Eyes.' "

"Oh, that's the name of the song, is it?" Alice said, trying to feel interested.

"No, you don't understand," the Knight said, looking a little vexed. "That's what the **name is called**. The **name really is** 'The Aged Aged Man.' "

"Then I ought to have said 'That's what the song is called' ?" Alice corrected herself.

"No, you oughtn't: that's quite another thing! The song is called 'Ways and Means,' but that is **only what it's called**, you know!"

"Well, what is the song, then?" said Alice, who was by this time completely bewildered.

"I was coming to that," the Knight said. "The **song really is** 'A-sitting On A Gate,' and the tune's my own invention."



Lewis Carroll  
*Through the Looking Glass*



## Verkette vs. Sequentielle Allozierung (*Allocation*)



- Ziel: Menge von Objekten abarbeiten
- **Sequential allocation**: ein Objekt nach dem anderen anordnen
  - Maschinenebene: aufeinanderfolgende Speicherstellen
  - Python / C++: Array von Objekten
- **Linked allocation**: jedes Objekt enthält Link / Zeiger / Referenz auf das nächste
  - Maschinenebene: Zeiger ist Speicheradresse des nächsten Objektes
  - Python: `object1.next = object2` ("alles ist ein Zeiger")
- Hauptunterschied:
  - Sequentiell: Indizierung wird unterstützt
  - Verkettet: Vergrößerung und Verkleinerung ist einfach
- Achtung: in Python gibt es scheinbar(!) beides für umsonst

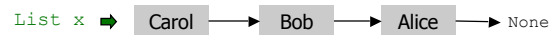


## Verkettete Liste (*Linked List*)



- **Liste** = Folge von Elementen  $a_0, a_1, \dots, a_{n-1}$ 
  - Elemente sind geordnet:  $a_i$  ist Nachfolger von  $a_{i-1}$  (wie bei Array)
  - es können an beliebiger Stelle Elemente eingefügt und wieder entfernt werden (i.A. anders als bei Array)
- Implementierung:
  - Üblicherweise mit Hilfe von verketteten Listenelementen
  - Listenelement enthält
    - "Nutzdaten" (*satellite data*) = eigentliche Elemente  $a_i$
    - **Zeiger** auf nachfolgendes Listenelement

```
class List:
    def __init__( self ):
        self.name = ""
        self.next = None
```



## Verkettete Liste Demo



```
a = List()
a.name = "Alice"
a.next = None
b = List()
b.name = "Bob"
b.next = a
c = List()
c.name = "Carol"
c.next = b
```



addr	value
C0	0
C1	0
C2	0
C3	0
C4	0
C5	0
C6	0
C7	0
C8	0
C9	0
CA	0
CB	0

main memory



## Traversierung einer Liste



- Musterbeispiel für das traversieren einer mit null endenden verketteten Liste

```
l = List()
... Liste füllen ...
li = l
while li != None:
    print li.name
    li = li.next
```



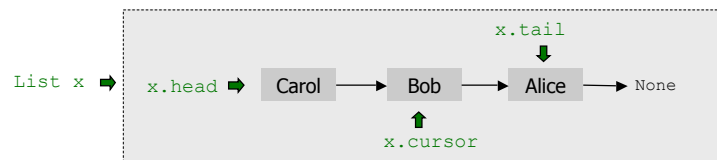
```
% ./list.py
```



## Liste mit mehr innerem "Wissen"



- Anforderungen:
  - Anhängen soll in 1 Schritt gehen → Liste muß letztes Element (*tail*) kennen
  - Am Anfang einfügen auch → Liste muß Anfang (*head*) kennen
  - Methode um "nächstes" Element zu erfragen (*Iterator*) → "*Cursor*" verwalten
- Liste soll Interna kapseln (verstecken):
  - Elemente der Liste verstecken
  - Head und Tail speichern
  - Cursor verwalten



```
class List:

    class ListElement:
        def __init__( self ):
            self.item = self.next = None

    def __init__( self ):
        self.head = None
        self.tail = None
        self.cursor = None

    def isEmpty(self):
        return self.head == None
```

G. Zachmann Informatik 1 - WS 05/06 Datenstrukturen 15

```
... (cont'd)

def append(self, item):
    if self.isEmpty():
        self.cursor = self.head = \ ← line continuation
        self.tail = ListElement()
    else:
        self.tail.next = ListElement()
        self.tail = self.tail.next
        self.tail.item = item
        self.tail.next = None
```

G. Zachmann Informatik 1 - WS 05/06 Datenstrukturen 16



```

# methods dealing with the iterator (cursor)

def rewind(self):
    self.cursor = head

def getCurrentItem(self):
    if self.cursor == None:           # Spezialfall abfangen!
        return None
    return self.cursor.item

def getNextItem(self):
    if self.cursor == None:
        return None
    self.cursor = self.cursor.next
    return getCurrentItem() # nicht etwa Code wiederholen!

```



G. Zachmann Informatik 1 - WS 05/06 Datenstrukturen 17

```

def insertAfterCurrent(self, item):
    if self.isEmpty():
        self.append(item)
        return
    if self.cursor == None:
        return # eigentlich nicht so gut
    z = ListElement()
    z.item = item
    z.next = self.cursor.next
    self.cursor.next = z



```

G. Zachmann Informatik 1 - WS 05/06 Datenstrukturen 18



```
def getNode( self, index ):  
    z = self.head  
    while index > 0 and z.next:  
        z = z.next  
        index -= 1  
    return z  
  
def insert( self, node, index ):  
    ...  
  
def findNode( self, item ):  
    ...
```

G. Zachmann Informatik 1 - WS 05/06 Datenstrukturen 19



## Weitere Operationen

- `getCursorPos()`: Position (Index) des aktuellen Elementes
- `setCursorAtPos( i )`: Setze aktuelles Element auf den Index `i`
- `delete()`: lösche ganze Liste
- `removeCurrent()`: lösche aktuelles Element aus Liste
- `insertBeforeCurrent( item )`: Setzt Element `e` vor die aktuelle Position; Achtung: Aufwand im worst-case  $\sim N$
- `find( item )`:
  - Suche **item** und setze Cursor auf entsprechende Zelle
  - Aufwand im worst-case  $\sim N$

G. Zachmann Informatik 1 - WS 05/06 Datenstrukturen 20

- `removeCurrent()`:
  - Entfernt Element an aktueller Position
  - Cursor zeigt anschließend auf nächstes Element (falls vorhanden, sonst auf Head)
  - Achtung: Aufwand kann proportional zu N sein (Man muß erst das Element vor aktueller Position finden)

The diagram shows a singly linked list with three nodes: 'a', 'e', and 'b'. Each node is represented as a rectangle divided into two parts: the left part contains the data value, and the right part contains a pointer to the next node. Node 'a' points to node 'e', and node 'e' points to node 'b'. A box labeled 'Cursor' has an arrow pointing to the data part of node 'e'. Another box labeled 'Nach Löschen' has an arrow pointing to the pointer part of node 'e', which is now pointing to node 'b', illustrating the update of the next pointer after the current element is removed.

G. Zachmann Informatik 1 - WS 05/06 Datenstrukturen 21

### Eigenschaften der einfach verketteten Liste

- man kann schnell auf Elemente hinter der aktuellen Position zugreifen
- will man auf Elemente davor zugreifen, muß man immer beim Anfang der Liste beginnen und die Position suchen,
  - Problem z.B. bei `removeCurrent()`, `insertBeforeCurrent()`
- Asymmetrie im Aufwand beim Durchlaufen der Kette

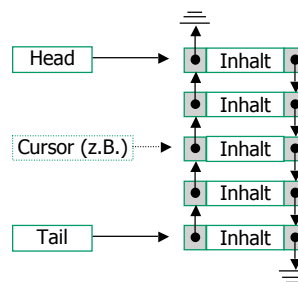
G. Zachmann Informatik 1 - WS 05/06 Datenstrukturen 22



## Doppelt verkettete Liste



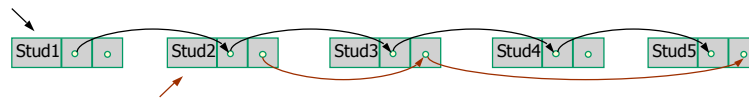
- Lösung: Doppelt verkettete Liste (*doubly linked list*)
- verkettet die Elemente in beide Richtungen
- Symmetrie im Aufwand beim Durchlaufen der Kette
- größerer Speicheraufwand
- Größerer Aufwand bei Entfernen / Einfügen



## Multi-Listen



- Auch *mehrdimensionale* Listen genannt
- Menge von Elementen gleichzeitig nach mehreren Kriterien organisiert
- Beispiel: Liste aller Studenten, mit Teilliste aller Informatik-Studenten
- Ziel: Elemente nur 1x vorhalten, aber verschiedene Listen / Teillisten
- Lösung: jede Organisation durch eine Verkettung dargestellt



- Jede Liste kann *für sich getrennt* verwaltet werden



## Beispiel: Dünnbesetzte Matrizen (*sparse matrix*)



- Matrix heißt **dünn besetzt**, wenn nur "wenige" Elemente  $\neq 0$  sind
  - "Wenig" ist Definitionssache, z.B. 10%
- Multi-Liste ist gängige Methode, um dünnbesetzte Matrix zu implementieren

