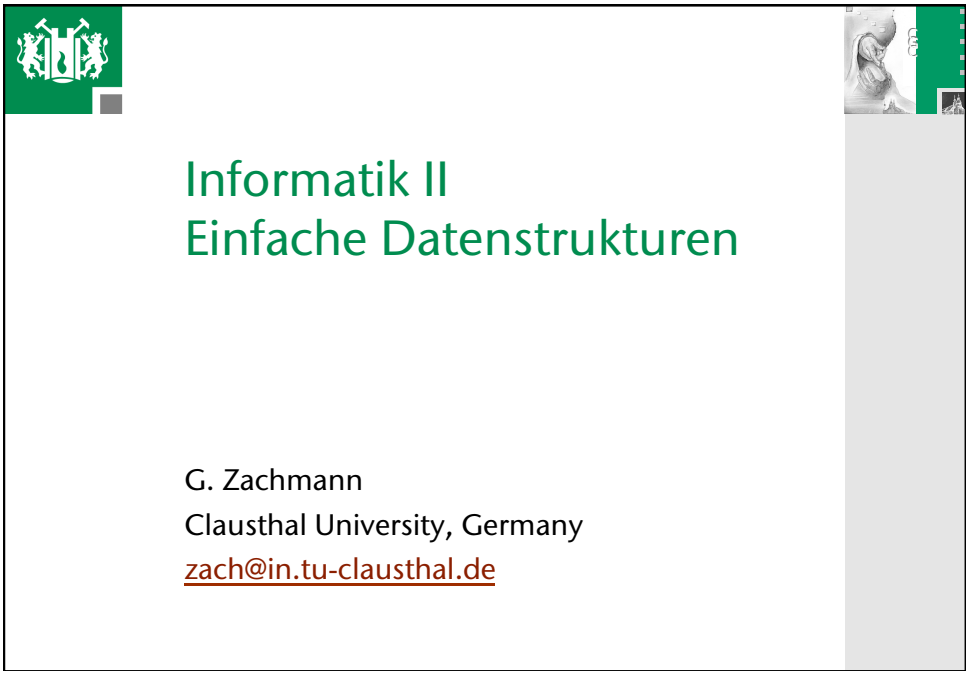
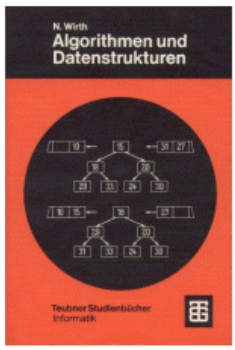


Informatik II
Einfache Datenstrukturen

G. Zachmann
Clausthal University, Germany
zach@in.tu-clausthal.de

Wieso eigtl. "Datenstrukturen"?

- Algorithmen & Datenstrukturen sind 2 Seiten derselben Medaille!



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

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* oder Instanzvariablen (bei Klassen)
- 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 $T_1, \ldots, T_n$ die Typen der Elemente, dann hat der Record/Struct den (algebraischen) Typ $T_1 \times \ldots \times T_n$

Das Datenbank-Problem

- Gegeben: Menge mit *n* Objekten $O_1, ..., O_n$
  - Jedes Objekt O_i hat zugehörigen Schlüssel (*keys*) $k(O_i)$
  - Beispiel: Kundendaten (Name, Adresse, Kreditkartennummer, Kundennummer)
    - Key = Name (mit lexikographischer Ordnung) oder Kundennr. (+ Ordnung auf Zahlen)
- Annahme: Auf der Menge der Keys existiert eine totale Ordnung
- Grundlegende Operationen:
  - Suchen (k): findet Objekt O mit Schlüssel k(O) = k in der Datenbank (falls es dort existiert) und gibt es aus, sonst NULL
  - Einfügen (O): fügt Objekt O in die Datenbank ein
  - Entfernen (O): entfernt Objekt O aus der Datenbank (falls es dort existiert)
- Gesucht: Datenstruktur, die diese Operationen effizient unterstützt

Das Array

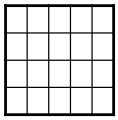
- Ein eindimensionales Array besteht aus einer bestimmten Anzahl von Datenelementen
  - Elemente haben gleichen Typ → homogenes Array (C, Java; wird allg. eher in statisch typisierten Sprachen verwendet)
  - Verschiedenen Typ → inhomogenes Array (Python, Smalltalk, ...; wird allg. eher in dynamisch typisierten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

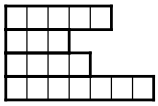
- Elemente werden indiziert, d.h., Identifikation und Zugriff erfolgt über Index = ganze Zahl ≥ 0 (typische Syntax: `a[i]`)
- Auf jedes Element des Array kann mit demselben, konstanten Zeitaufwand zugegriffen werden

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 = 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 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: ~N [O(N)]
- Element Nr i löschen: ~N [O(N)]
- Array löschen: konstant [O(1)]

Arrays in Python

- Array = Folge beliebiger und inhomogener Werte
- Beispiele (Array-Literale):

```
studenten = [ "Meier", "Mueller", "Schmidt" ]
l = [ 1, 2, 3 ]
l = [ "null", "acht", 15 ]
```

- Elementenummerierung: von 0 bis Anzahl-1 !
- Zugriff mit **[]**

```
student_des_monats = studenten[2]
studenten[0] = "Becker"
```

- Mit append() werden neue Elemente am Ende der Liste hinzugefügt

```
studenten.append("Bach")
```

Operationen auf Arrays (in Python)

Operationen auf Listen	
`s[i]`	Ergibt Element i der Sequenz s
`s[i:j]`	Ergibt einen Teilbereich (*slice*)
`len(s)`	Ergibt die Anzahl der Elemente in s
`min(s)`	Ergibt Minimum
`max(s)`	Ergibt Maximum
`s.append(x)`	Fügt neues Element x an das Ende des Arrays
`s.extend(l)`	Fügt ein neues Array l an das Ende von s
`s.count(x)`	Zählt das Vorkommen von x in s
`s.index(x)`	Liefert kleinsten Index i mit s[i] == x
`s.insert(i,x)`	Fügt x am Index i ein
`s.remove(x)`	Liefert Element i und entfernt es aus dem Array
`s.reverse()`	Invertiert die Reihenfolge der Elemente
`s.sort([cmpfunc])`	Sortiert die Elemente

- Erzeugen eines Arrays von Zahlen mit **`range`** :
  - Syntax: **`range( [start], stop [,step] )`**
  - Beispiel:

```
x = range(0,100)    # 0, ..., 99
x = range(10)       # 0, ..., 9
x = range(1,17,2)   # 1, 3, 5, …, 17
for i in range(0,N):
    ...
for i in x:
    ...
```

Beispiel: Mischen eines Arrays

- Aufgabe: zufällige Permutation von (0,...,N-1) erzeugen

```
import sys
import random
N = int( sys.argv[1] )
a = range( 0, N )
for i in range( 0, N ):
    r = random.randint(0, i)
    a[r], a[i] = a[i], a[r]
print a
```

- Beispiel:

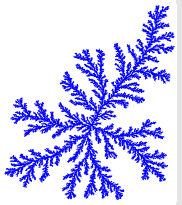
Mehrdimensionale Arrays

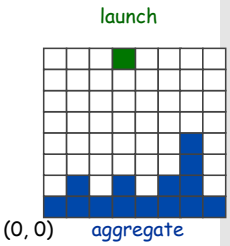
- Arrays können als Elemente auch selbst wieder Arrays enthalten

```
a = [1, "Dave", 3.14, ["Mark", 7, 9, [100, 101]], 10]
a[1]        # Ergibt "Dave"
a[3][2]     # Ergibt 9
a[3][3][1] # Ergibt 101
a[3][2][0] # Ergibt Fehlermeldung:
           #   TypeError: 'int' object is unsubscriptable
```

Beispiel: Diffusion Limited Aggregation

- Modell für fraktales Wachstum
- Grundlage ist die Brown'sche Molekularbewegung
- Beispiele:
  - Anlagerung von Rußteilchen an Wänden und Kaminen
  - Bildung von Fellzeichnungen bei Zebra, Tiger, Leopard,...
  - Korallenwachstum
- Ansatz: Monte-Carlo-Simulation
  1. Erzeuge einen Partikel an der *launch site*
  2. Der Partikel wandert zufällig durch das 2-D Gitter bis
     - er einen anderen Partikel berührt ⇒ dann wird er dort angelagert
     - er das Gitter wieder verlässt
  3. Gehe wieder zu 1.

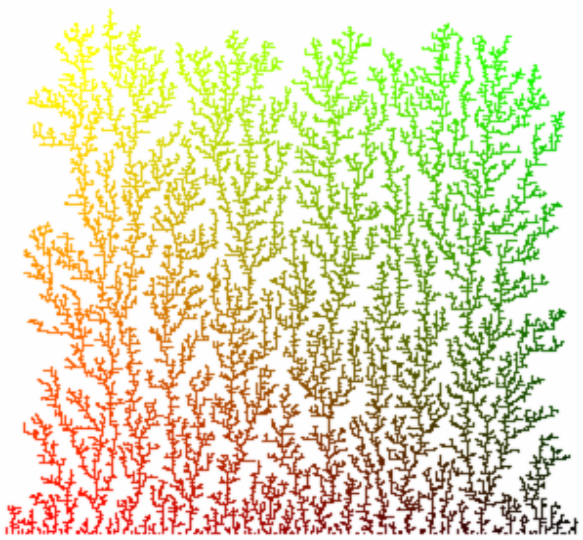





```
import Image
import random
N = 200                    # size of image/grid
launch = N - 10;           # y-pos of launch site
grid = []                  # grid for particles
# init grid for particle motion
for i in range( 0, N ):
    b = []
    for j in range( 0, N ):
        b.append( False )  # False = "not occupied"
    grid.append(b)
for i in range(0, N):
    grid[i][0] = True    # fill bottom row with particles
# create image to render grid
im = Image.new("RGB", (N, N), (256, 256, 256) )
```

(Fortsetzung nächste Folie ...)

```
done = False
while not done:
   x = random.randint( 0, N )          # init new particle
   y = launch                          # at the top line of the grid

   # perform random walk
   while (x < N - 2) and (x > 1) and (y < N - 2) and (y > 1):
      r = random.random()
      if r < 0.25:                      # make 1 random step
          x -= 1                        # left, right, up, or down
      elif r < 0.50:
          x += 1
      elif r < 0.65:
          y += 1
      else:
          y -= 1

      # check if particle touches the "plant"
      if grid[x-1][y] or grid[x+1][y] or grid[x][y-1] or grid[x][y+1]:
          grid[x][y] = True               # occupy grid cell
          im.putpixel( (x,y), (1,1,1) )   # draw new particle
          if y > launch:                  # height of plant > grid
              done = True                 # -> stop growth
          break                           # start with new particle
```

Aufwand für die "Datenbank"-Operationen

- Objekt hinzufügen:

```
Array A vergrößern → Array der Größe N+1
A[N] ← Objekt
```

  - Aufwand ~ N (für Vergrößern des Arrays)

- Objekt suchen:

```
Input:  Key k
Output: Objekt O mit key(O) = k, falls gefunden
        None, falls Key nicht gefunden
for i = 0 ... N-1:
   if key( A[i] ) == k:
      return A[i]
return None
```

  - Erwarteter Aufwand ~ N

- Vorteile: sehr einfache Datenstruktur

- Nachteile:
  - Speicherbedarf nicht vorhersagbar → ständiges Vergrößern nötig
    - Müssen wir auch verkleinern?
  - Hoher Aufwand (= Laufzeit) für Einfügen / Löschen / Suchen

Tupel in Python

- Tupel = geordnete Menge beliebiger Objekte
- Tupel sind, wie Strings, nicht veränderbar (*immutable*)
- Beispiel:

```
t1 = (12, "17", 42)
t2 = (t1, )             # Tupel mit 1 Elem
```

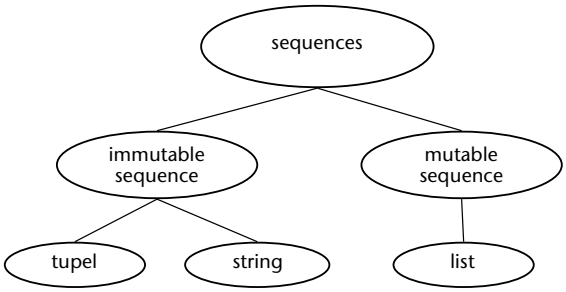
- Operationen: wie für Arrays, ohne die verändernden Operationen
- Häufige Verwendung:
  - Durchführen eines "Swap":

```
(x, y) = (y, x)
```

  - Rückgabe mehrerer Funktionswerte:

```
return (x,y,z)        # z.b. ein Punkt
```


Hierarchie von Sequenz-Typen (in Python)



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 Caroll
Through the Looking Glass

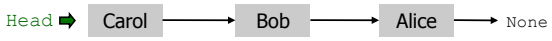
Verkettete vs. Sequentielle Allozierung (*Allocation*)

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

Die verkettete Liste (*Linked List*)

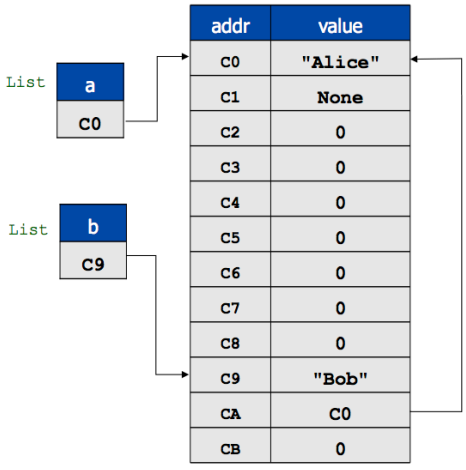
- Liste = Folge von Elementen $a_0, a_1, \ldots, a_{n-1}$
  - Elemente sind geordnet: a_i ist Nachfolger von a_{i-1} (wie bei Array)
  - es können an beliebiger Stelle Elemente eingefügt und wieder entfernt werden (i.A. anders als bei Array)
- Implementierung:
  - Üblicherweise mit Hilfe von verketteten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

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



Demo zur verketteten Liste

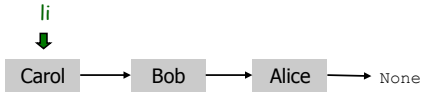
```
     a = ListElement()
a.name = "Alice"
a.next = None

     b = ListElement()
b.name = "Bob"
b.next = a

     c = ListElement()
c.name = "Carol"
c.next = b
```



Traversierung einer Liste

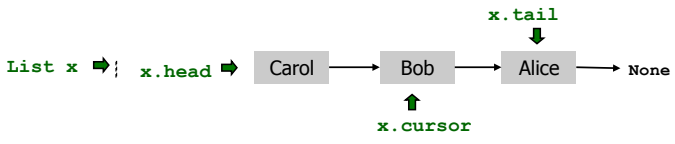
- Musterbeispiel für das Traversieren einer mit **None** endenden verketteten Liste:

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



Listen 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
- Die Liste (die Klasse) soll Interna kapseln (verstecken):
  - Elemente der Liste müssen von außen nicht sichtbar sein
  - Head und Tail speichern
  - Cursor verwalten

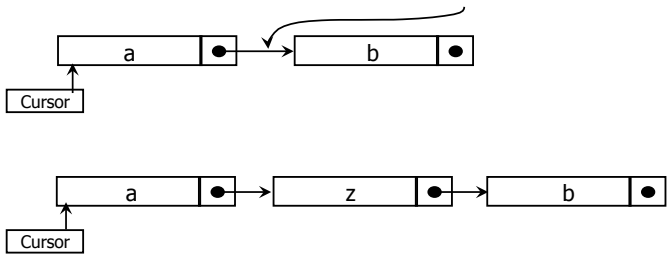


```
class List:

   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

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

line continuation


```
# 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()          # nie Code wiederholen!
```

```
def insertAfterCurrent(self, item):
    if self.isEmpty():
        self.append(item)
        return
    if self.cursor == None    # "can't happen"
        return                # eigentlich nicht so gut
    z = ListElement()
    z.item = item
    z.next = self.cursor.next
    self.cursor.next = z
```

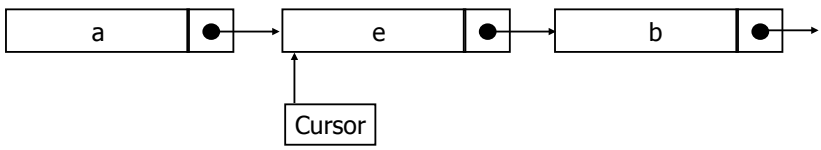



```
def getElementByIndex( self, index ):
    z = self.head
    while index > 0 and z.next:
        z = z.next
        index -= 1
    return z

def insertAtIndex( self, node, index ):
    ...

def findKey( self, key ):
    ...
```

Weitere Operationen

- **`cursorPos()`**: Position (Index) des aktuellen Elementes
- **`setCursorPos( i )`**: Setze aktuelles Element auf den Index i
- **`delete()`**: leere die ganze Liste
- **`removeCurrent()`**: lösche aktuelles Element aus Liste
- **`insertBeforeCurrent( item )`**: Setzt Element **`item`** vor die aktuelle Position; Achtung: Aufwand im worst-case ~ N
- **`find( key )`**:
  - Suche Element mit Schlüssel **`key`** und setze Cursor auf entsprechendes Element
  - Aufwand im worst-case ~ N

- **`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 der Cursor-Position finden)

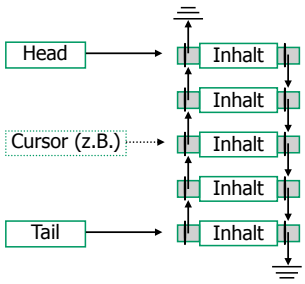


Eingenschaften der einfach verketteten Liste

- 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 (vorwärts / rückwärts)

Doppelt verkettete Liste

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

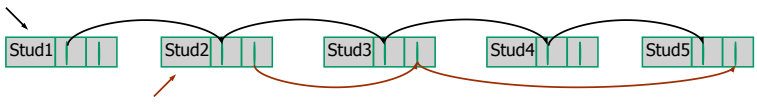


Zusammenfassung der Aufwände bei Listen

- N = Anzahl Elemente in der Liste
- Platzbedarf ~ N
- Laufzeit für Einfügen / Entfernen an der Cursor-Position = konstant
- Laufzeit für die Suche nach einem Key ~ N

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 darstellen



- Jede Liste kann für sich getrennt verwaltet werden

Beispiel: Dünnbesetzte Matrizen (*sparse matrix*)

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