Informatik II
Einfache Datenstrukturen

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

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 $O_1, \ldots, O_n$
  - 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: Keys sind total geordnet
- 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

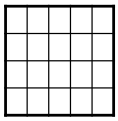
- Ein eindimensionales Array besteht aus einer bestimmten Anzahl von Datenelementen
  - Elemente haben gleichen Typ → homogenes Array (C, Java, wird allg. eher in statisch typisierten Sprachen verwendet)
  - Verschiedenen Typ → inhomogenes Array (Python, Smalltalk, ..., wird allg. eher in dynamisch typisierten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

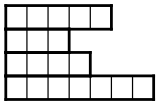
- Elemente werden indiziert, d.h., Identifikation und Zugriff erfolgt über Index = ganze Zahl ≥ 0 (typische Syntax: `a[i]`)
- Auf jedes Element des Array kann mit demselben, konstanten Zeitaufwand zugegriffen werden

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)]

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

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

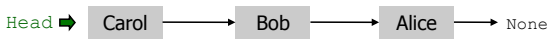
Verkettete vs. Sequentielle Allozierung (*Allocation*)

- 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

Verkettete Liste (*Linked List*)

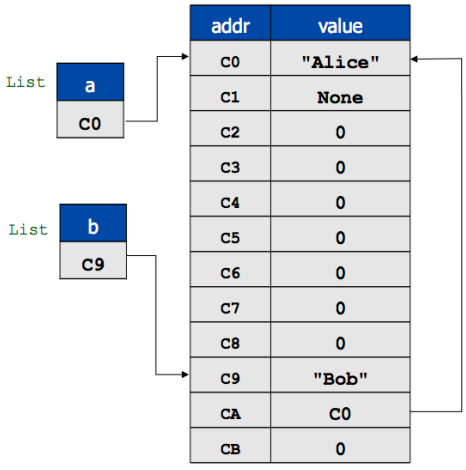
- Liste = Folge von Elementen $a_0, a_1, \ldots, a_{n-1}$
  - Elemente sind geordnet: a_i ist Nachfolger von a_{i-1} (wie bei Array)
  - es können an beliebiger Stelle Elemente eingefügt und wieder entfernt werden (i.A. anders als bei Array)
- Implementierung:
  - Üblicherweise mit Hilfe von verketteten 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

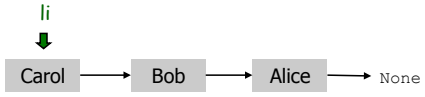
```
     a = ListElement()
a.name = "Alice"
a.next = None

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

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



Traversierung einer Liste

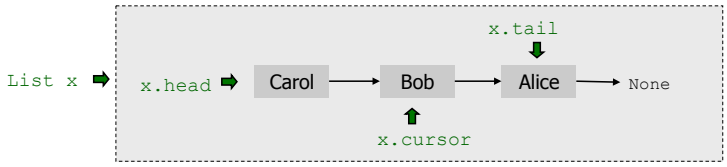
- Musterbeispiel für das Traversieren einer mit **`None`** endenden verketteten Liste:

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



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

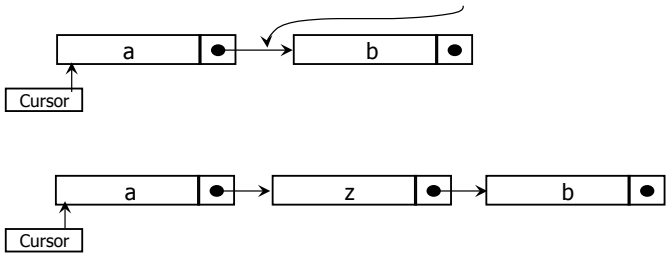


```
class List:

   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
```

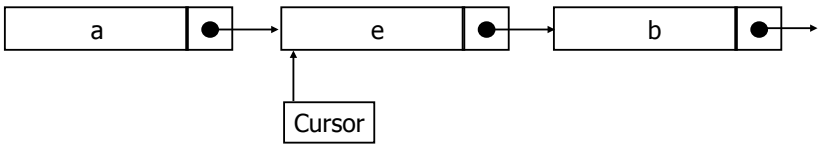



```
def getElementByIndex( self, index ):
    z = self.head
    while index > 0 and z.next:
        z = z.next
        index -= 1
    return z

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)

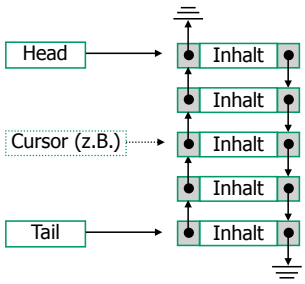


Eingenschaften der einfach verketteten Liste

- 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

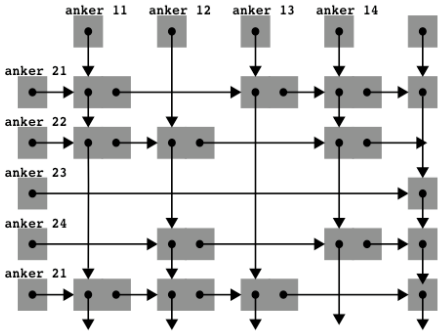
Multi-Listen

- 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



Stack und Queue

- Ähnlich wie Listen, aber mit zusätzlichen Einschränkungen / Vereinfachungen
- Gemeinsamkeit:
  - Einfügen immer nur am Kopf der Liste
  - Löschen auch nur an einem Ende der Liste

Der grundlegende Unterschied zwischen Stack und Queue

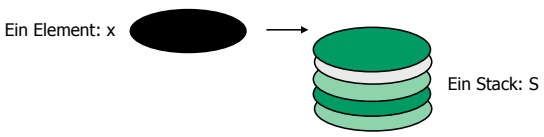
- Stack
  - Entferne das Objekt, das zuletzt hinzugefügt wurde
  - Heißt daher auch: LIFO = *"last in first out"*
  - Analogie: Cafeteriabehälter, surfen im Web.
  - „ Die letzten werden die ersten sein.“

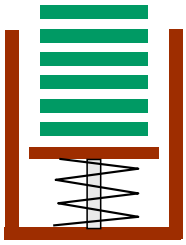
Pop-A-Filter

- Queue
  - Entferne das Objekt, das zuerst eingefügt wurde
  - Heißt daher auch: FIFO = *"first in first out"*
  - Analogie: Registrar's line.
  - „Wer zuerst kommt, malt zuerst“ („first come, first serve“)

Der Stack

- (Deutsch: Stapel, Kellerspeicher)
- Zunächst: abstrakte Datenstruktur, Container-Datentyp
- Elemente können einfügt und wieder entfernt werden
- Direkter Zugriff nur auf das zuletzt eingefügte Element *(last in first out)*

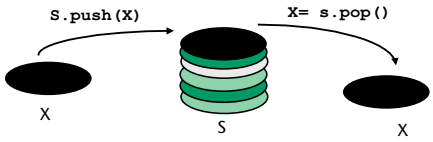




Grundlegende Operationen

- **`pop()`** liefert zuletzt auf den Stack gelegtes Element und löscht es
- **`push( X )`** legt ein Element X auf den Stack
- **`isEmpty()`** Ist der Stack leer?
- **`peek()`** liefert zuletzt auf den Stack gelegtes Element ohne Löschen

- Anwendungen.
  - Surfen im Web mit einem Browser.
  - Funktionsaufrufe
  - Parsen von Programmen
  - PostScript-Sprache für Drucker
  - Reverse Polish calculators (RPN)



- Bemerkung: diese Anzahl von Operationen ist nicht minimal:
  - Eigentlich reichen **`push`** und **`pop`**:

    ```
    X = S.peek()
    ```

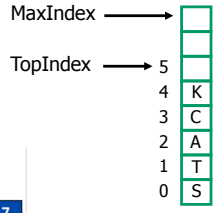
    ist äquivalent zu:

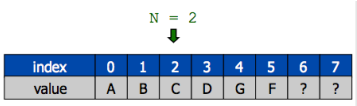
    ```
    X = S.pop()
    S.push(X)
    ```

  - **`peek()`** ist aber effizienter und wird häufig benötigt
- Oftmals gibt es weitere Operationen:
  - **`isFull:`** true, falls kein Element mehr auf den Stapel paßt
  - **`clear`**: entfernt alle Elemente vom Stack

Stack-Implementation mittels Array

- Implementierung eines Stacks mit Hilfe eines Arrays
  - **`s = array, N = #`** Objekte auf dem Stack
  - **`push`**: speichere Objekt in **`s[N]`**
  - **`pop`**: entferne ein Objekt aus **`s[N-1]`**





- Fehlerbehandlung bei:
  - **`pop`** für leeren Stack und **`push`** für vollen Stack erzeugen Fehler
- Ist in Python praktisch schon vorhanden durch die entsprechenden Listen-Methoden

Wie vergrößert man ein Array geschickt?

- Problem: im voraus ist nicht bekannt, wie groß das Array sein soll
- Also: zunächst ganz kleines Array erzeugen, dann *Resize*-Operation
- Erste Idee: Jedesmal, wenn Array voll,
  1. Neues Array erzeugen mit Größe $n + c$
  2. Elemente vom alten Array ins neue Array umkopieren
  3. Altes Array freigeben
- Nachteil: Daten werden bis zu $\frac{N^2}{2c}$ Mal umkopiert!
- Beweis: Sei N = Maximal-Größe des Arrays "am Ende"
  - Resize-Operation passiert N/c Mal
  - Bei Resize Nr i werden $i \cdot c$ viele Elemente kopiert
  - Zusammen: $\sum_{i=1}^{N/c} i \cdot c \approx \frac{N^2}{2c}$

- Bessere Idee:
  - Verwende die *repeated doubling* Strategie oder *doubling technique*
  - Wenn Array zu klein, führe Resize-Operation mit neuer Größe $2n$ aus
- Behauptung: Daten werden nur noch bis zu $2N$ Mal umkopiert
- Beweis:
  - Resize-Operation passiert nur noch $d = \lceil \log N \rceil$ Mal
  - Bei Resize Nr i werden 2^i viele Elemente kopiert
  - Zusammen:

$$\sum_{i=1}^{d} 2^i = 2^{d+1} - 1 \approx 2N$$

- Bemerkung: in C++ STL's **`vector`** implementiert diese Strategie (Python auch?)

Python-Code

```
class Stack(object):
   def __init__( self ):
       self.s = []
       self.N = 0           # wir verwalten Stack-Größe selbst,
                            # zu "Demo"-Zwecken (wäre nicht nötig
   def isEmpty(self):       # in Python)
       return N == 0

   def push(self, item):
       if N >= len(s):
          s.extend( len(s) * [None] )      # Länge verdoppeln
       s[N] = item
       N += 1

   def pop(self):
       if N == 0:
          return None       # Error-Code wäre besser
       N -= 1
       return s[N+1]
```

Erzeugt Liste der Länge len(s)
mit None initialisiert