Informatik I
Einfache Datenstrukturen

G. Zachmann
Clausthal University, Germany
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 ≥ 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 = 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: ~N [O(N)]
- Element Nr i löschen: ~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 $T_1, \ldots, T_n$ die Typen der Members, dann hat der Record/Struct den (algebraischen) Typ $T_1 \times \ldots \times T_n$

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 Caroll
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 $a_0, a_1, ..., a_{n-1}$
  - 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:

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

  - Üblicherweise mit Hilfe von verketteten Listenelementen
  - Listenelement enthält
    - "Nutzdaten" (satellite data) = eigentliche Elemente a_i
    - Zeiger auf nachfolgendes Listenelement

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

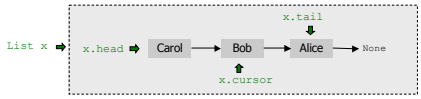
- Musterbeispiel für das traversieren einer mit null endenden verketteten Liste

```
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
```



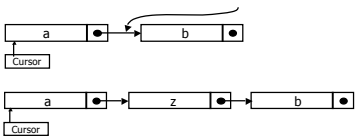
```
… (cont'd)

   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()    # nicht etwa Code wiederholen!
```



```
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
```

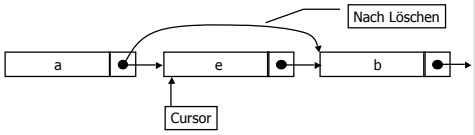



```
def getNode( self, index ):
    z = self.head
    while index > 0 and z.next:
        z = z.next
        index -= 1
    return z

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

def findNode( self, item ):
    ...
```


Weitere Operationen

- `getCursorPos()`: Position (Index) des aktuellen Elementes
- `setCursotAtPos( 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 ~ N
- `find( item )`:
  - Suche **`item`** und setze Cursor auf entsprechende Zelle
  - 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 aktueller Position finden)

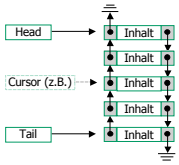


Eingenschaften der einfach verketteten Liste

- 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

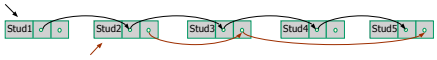
Doppelt verkettete Liste

- Lösung: Doppelt verkettet 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

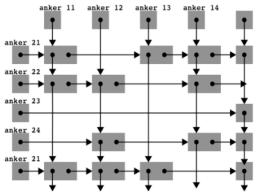
- Auch mehrdimensionale Listen genannt
- Menge von Elementen gleichzeitig nach mehreren Kriterien organisiert
- Beispiel: Liste aller Studenten, mit Teilliste aller Informatik-Studenten
- Ziel: Elemente nur 1x vorhalten, aber verschiedene Listen / Teillisten
- Lösung: jede Organisation durch eine Verkettung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



- Jede Liste kann für sich getrennt verwaltet werden

Beispiel: Dünnbesetzte Matrizen (*sparse matrix*)

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



Stack und Queue

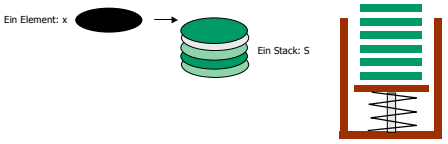
- Grundlegender Datentyp
- Menge von Operationen (add, remove, test if empty) auf generischen Daten
- Ähnlich wie Listen, aber mit zusätzlichen Einschränkungen / Vereinfachungen:
  - Einfügen immer nur am Kopf der Liste
  - Löschen auch nur an einem Ende (2 Möglichkeiten!)

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

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

Stack

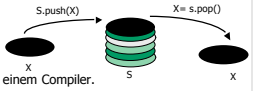
- deutsch: Stapel, Kellerspeicher
- Zunächst: abstrakte Datenstruktur, Container-Datentyp
- Elemente können einfügt und wieder entfernt werden
- direkter Zugriff nur auf das zuletzt eingefügte Element *(last in first out)*



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.
  - Implementierte Funktionsaufrufe in einem Compiler.
  - Parsen.
  - PostScript Sprache für drucker.
  - Reverse Polish calculators.



- Anzahl Operationen nicht minimal:
  - Eigentlich reichen push() und pop()

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

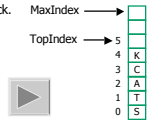
    ist äquivalent zu:

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

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

Stack Implementation (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]`



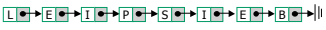
- Fehlerbehandlung:
  - `pop()` für leeren Stack und `push()` für vollen Stack erzeugen Fehler
- Ist in Python praktisch schon vorhanden durch die entspr. Listen-Methoden.

Wie vergrößert man ein Array geschickt?

- Problem: im voraus 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 $i\,c$ 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: $\sum_{i=1}^{d} 2^i = 2^{d+1} - 1 \approx 2N$
- Bem.: STL's `vector` implementiert diese Strategie (Python sicher auch `;-)`)

Implementierung mit Liste



- `push()` fügt ein Element am Kopf der Liste hinzu
- `pop()` entfernt erstes Element (am Kopf) der Liste
- `isFull()` nicht sinnvoll (bzw. liefert immer den Wert `false`)
- Vorteil
  - Speicherbedarf für Stack häufig nicht bekannt
  - bei Array muß max. Speicherplatz festgelegt werden, oder Resize
- Nachteil:
  - Mehr Verwaltungsaufwand
  - Mögl.weise nicht *"cache friendly"*

Exkurs: Wichtiges OOD-Prinzip

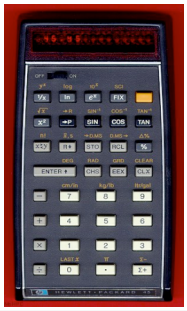
- *Information hiding*:
  - Klasse (hier Stack) gibt nur Schnittstelle (API = application programmer's interface) preis
    - Hier: `push()`, `pop()`, `peek()`, …
  - Versteckt interne Implementierungsdetails
    - Hier: Liste oder Array, doppelt oder einfach verkettet, mit Resize oder ohne, …
  - Versteckt außerdem interne Daten
    - Hier: Head- und Tail-Zeiger, gibt es Cursor oder nicht, Anzahl-Zähler oder nicht, …
- Vorteil: man kann interne Implementierungsdetails ändern, ohne daß Anwendungsprogramme von Stack etwas merken (außer mögl.weise Laufzeit)!
- Eines der wichtigsten Merkmale von OOP (genauer: OOD)

Python-Code

```
class Stack:
    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

An Ancient Calculator



HP 45.

Preis Im Jahr 1973: $395.
(Das entspricht $1600 im Jahr 2002.)

Was fehlt auf der Tastatur?

Beispiel-Anwendung für Stack: Postfix-Auswertung

J. Lukasiewicz (1878-1956)

- Postfix-Ausdrücke:
  - auch genannt: umgekehrte polnische Notation (UPN; RPN = *reverse polish notation*)
  - Aufbau von Ausdrücken: Erst die Operanden, dann der Operator
- Beispiel:
  Infix-Notation: (2+4)! / (11+4) ⇒
  Postfix-Notation: 2 4 + ! 11 4 + /
- Abarbeitung von Postfix-Ausdrücken: verwende Stack von Int's
  - der Ausdruck wird von links nach rechts gelesen
  - ist das gelesene Objekt ein Operand, wird es mit push() auf den Stack gelegt
  - ist das gelesene Objekt ein Operator, der n Parameter benötigt (ein n-stelliger Operator), wird er auf die n obersten Elemente des Stacks angewandt. Das Ergebnis ersetzt die n Elemente.

- Systematische Art, die Zwischenergebnisse zu speichern und Klammern zu vermeiden
- Beispiele:

```
% postfix.py
1 2 3 4 5 * + 6 * * +
6625        Infixausdruck: (1+(((2*((3+(4*5))*6)))

% postfix.py
7 16 16 16 * * * 5 16 16 * * 3 16 * 1 + + +
30001       Wandle 7531 von hexadezimal in dezimal um

% postfix.py
7 16 * 5 + 16 * 3 + 16 * 1 +
30001       Horner-Schema
```


Python Code

postfix.py

```
stack = Stack()
s = read_word()
while s != "":
   if s == "+":
      stack.push( stack.pop() + stack.pop() )
   elif s == "*"
      stack.push( stack.pop() * stack.pop() )
   else
      stack.push( int(s) )
   s = read_word()

print stack.pop()
```

Infix → Postfix

- Aufgabe: Konvertierung Infix- nach Postfix-Notation
- Algorithmus:
  - Linke Klammern: ignorieren
  - Rechte Klammern: pop und print
  - Operator: push
  - Integer: print

```
% ./infix.py
( 2 + ( ( 3 + 4 ) * ( 5 * 6 ) ) )
* 2 3 4 + 5 6 * * +

% infix.py | postfix.py
( 2 + ( ( 3 + 4 ) * ( 5 * 6 ) ) )
212
```

```
stack = Stack()
s = read word()
while s !="" :
    if s == "+":
        stack.push(s)
    elif s == "*":
        stack.push(s)
    elif s == ")":
        print stack.pop(), " ", # trailing comma!
    elif s == "(":
        pass                    # = NOP
    else:
        print s, " ",
```


- Postfix-Ausdrücke kommen immer noch in der Praxis vor
- Beispiele:
  - Taschenrechner (z.B. von HP, heute noch?)
  - Stackorientierte Prozessoren
  - Postscript-Dateien
- Weitere Anwendungen für Stack: Auswertung rekursiver Methoden / Funktionen
  - bei jedem rekursiven Aufruf müssen:
    - Parameter übergeben,
    - neuer Speicherplatz für lokale Variablen bereitgestellt,
    - Funktionswerte zurückgegeben werden
  - → *Stack-Frame*

Weitere Stack-Anwendung: Balancierte Klammern

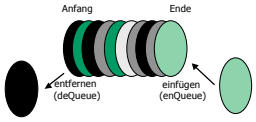
- Aufgabe: Bestimme ob die Klammern in einem String balanciert sind.
  - Bearbeite jedes Zeichen, eins nach dem anderen.
    - Linke Klammer: push
    - Rechte Klammer: pop und prüfe ob es paßt
    - Ignoriere andere Zeichen
  - Ausdruck ist balanciert, wenn der Stack nach Beendigung leer ist.

String	Balanced
( ) (( ))	true
((( ) ( )))	true
(( )) (( )	false
[([ ])]	true
[[( ])]	false
a[2*(i+j)] = a[b[i]];	true


```
Left_paren  = "([{"
Right_paren = ")]}"

def isBalanced(s):
  stack = Stack()
  for c in s:
     if c in Left_paren:
          stack.push(c)
     elif c in Right_paren:
          if stack.isEmpty():
               return false
          if Right_paren.find(c) != Left_paren.find(c):
               return false
  return stack.isEmpty()
```

Queue

- deutsch: Warteschlange, Puffer
- abstrakte Datenstruktur, Container-Datentyp
- Elemente können einfügt und wieder entfernt werden
- direkter Zugriff nur auf das zuerst eingefügte (*least recently added*) Element (daher: FIFO = *first in first out*)



Operationen

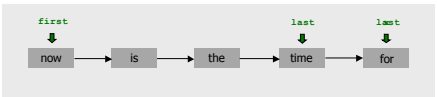
- **enqueue** Füge ein neues Objekt in die Warteschlange ein.
- **dequeue** Lösche und gebe aus das Objekt, das zuerst eingefügt wurde.
- **isEmpty** Ist die Warteschlange leer?

```
q = Queue()
q.enqueue("This")
q.enqueue("is")
q.enqueue("a")
print q.remove()
q.enqueue("test.")
while not q.isEmpty() :
    print q.dequeue()
```

A simple queue client

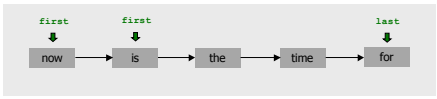
Implementierung als verkette Liste

- **enqueue**




```
x = List();
x.item = "for";
last.next = x;
last = x;
```

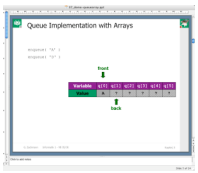

- **dequeue**

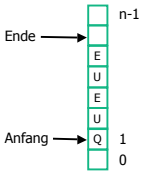



```
val = first.item;
first = first.next;
return val;
```

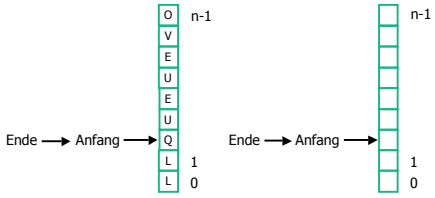
Implementierung mit Array

- begrenzter Speicherplatz: Array mit n Elementen
- zwei Zeiger: auf Anfang und Ende
- zyklischer Zugriff auf Elemente
  - erreicht ein Zeiger beim Inkrementieren den Wert n, wird er auf 0 zurückgesetzt





- **`Anfang == Ende`** → entweder ist Queue voll, oder leer



Implementierung in Python mit Liste

```
class Queue :
   def __init__( self ):
       self.first = self.last = None
   class List:                # nested class
      item = None             # satellite data
      next = None             # "pointer"
   def isEmpty(self):         # Methode in Queue
      return first == None
   def enqueue(self, item):
      x = List()
      x.item = item
      x.next = None
      if self.isEmpty():
         self.first = x
      else:
         self.last.next = x
      self.last = x
   def dequeue(self):
      val = self.first.item
      self.first = self.first.next  # unlink first item
      return val
```


- Bemerkung: Die Queue muß nicht homogen sein! (im Gegensatz zu den einfachen, analogen Implementierungen in Java/C++)
  - Da schon Liste (und Array) nicht homogen sein müssen
- Frage: stimmt **`dequeue()`** auch für den Fall, daß Liste genau 1 Element enthält ?
- Generelle Regel für Datenstrukturen-Entwurf: checke die "Ausnahmen"!! (Randfälle, *boundary cases*)
  - Stimmt die Funktion für den Fall, daß 0 oder 1 Element vorhanden ist?
  - Was passiert, wenn Cursor am Ende oder auf None steht?
  - ...

Anwendungen (nur Beispiele)

- In Programmen: alle Arten von Daten-Puffern
  - Dispensing requests on a shared resource (printer, processor)
  - Asynchronous data transfer (file IO, pipes, sockets)
    - man kann mehrere Elemente auf einen Schlag hinzufügen (z.B. Teil einer Datei von Festplatte)
    - danach kann man einzeln auf die Elemente zugreifen
  - Data buffers (MP3 player, portable CD player, Tastatur)
- Simulation
  - von Fertigungsprozessen: Objekte auf Förderbändern verhalten sich wie in einer Warteschlange
  - Wartezeiten bei McDonalds oder Call-Center, oder Verkehr vor Tunnel, oder ...

Exkurs: Poisson-Prozeß

- In der Natur viele Prozesse mit folgenden Eigenschaften:
  - Zeitpunkt des Ereignisse zufällig und unabhängig voneinander, z.B:
    - Radioaktives Material
    - Kunden kommen an Warteschlange an
    - HTML-Request an einem WWW-Server
    - Unfälle an einer Kreuzung
- Zugrunde liegende Annahme:
  - Seien T_i der Zeitpunkt des i-ten Ereignisses, seien $X_i = T_i - T_{i-1}$ die "Zwischenzeiten" (*inter-arrival times*)
  - Der Prozeß verhält sich nach dem k-ten Ereignis (für beliebiges k) genau wie am Anfang
  - Die X_i müssen alle identisch und unabhängig voneinander verteilt sein (i.i.d = *independent and identically distributed*)

Verteilung der Inter-Arrival Times

- Fundamentale Annahme: der Prozeß ist "gedächtnislos", d.h.,
  - Falls Ereignis noch nicht nach Zeitraum s eingetreten, dann ist Wahrscheinlichkeit dafür, daß Ereignis bis Zeit $s+t$ eintritt genauso hoch wie W.keit, daß Ereignis bis Zeit t eintritt

$$\forall s, t > 0 : P[X > t + s \mid X > s] = P[X > t]$$

- Man kann zeigen (o.Bew.):

$$P[X > t + s] = P[X > t] \cdot P[X > s]$$

- Die einzige Funktion, die diesen Prozeß beschreiben kann, ist

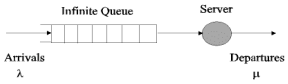
$$P[X > t] = e^{-rt}, \quad t \geq 0$$

  wobei r eine Skalierung ist.
- Die Zeit X zwischen zwei Ereignissen gehorcht dieser Dichtefunktion

$$f(t) = P[X = t] = re^{-rt}, \quad t \geq 0$$

M/M/1 Queuing Model

- Customers arrive at rate of λ per minute.
- Customers are serviced at rate of μ per minute.
- Use *Poisson process* to model arrivals and departures.
- Wichtigstes Warteschlangenmodell



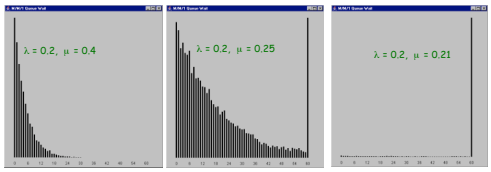
- How long does a customer wait in queue?

M/M/1 Queue: Implementation

```
while true :
    if nextArrival < nextDeparture :                          arrival
        customer = Customer()
        customer.arrive( nextArrival )
        q.enqueue( customer )          amount of time until next arrival
        nextArrival += exponential( lambda )
    else :
        if not q.isEmpty() :                                  departure
            hist.addDataPoint( math.min(60, q.length()) )
            customer = q.dequeue()
            customer.depart( nextDeparture )
            nextDeparture += exponential(mu)
                             amount of time until next departure
```


M/M/1 Queue Analysis

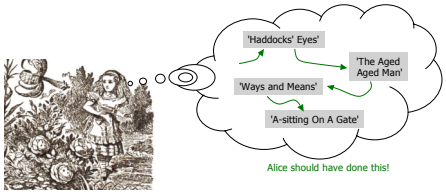
Wenn Abarbeitungsrate gegen Ankunftrate geht, warten immer mehr Kunden ≥60 Min., d.h., die Queue explodiert immer häufiger



("As service rate approaches arrival rate, service goes to h***.")

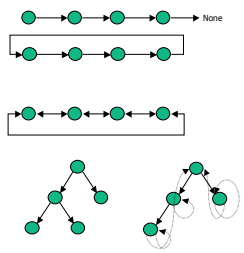
Folgerung

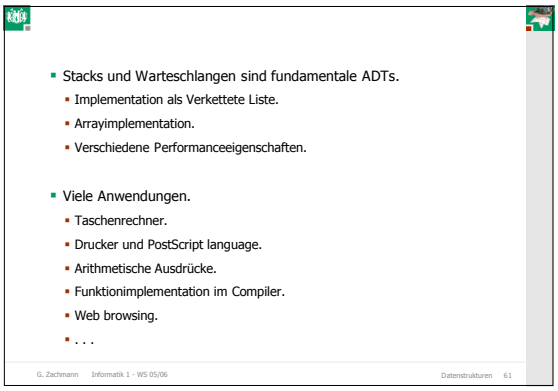
- *Sequential allocation*: unterstützt Indizierung, feste Größe.
- *Linked allocation*: variable Größe, unterstützt sequentiellen Zugriff.
- Verkettete Strukturen sind eine zentrale Datenstruktur und -abstraktion.



Vergleich von Varianten verketteter Strukturen

- Linked list.
- Circular linked list.
- Doubly linked list.
- Binary tree.
- Patricia tries.




Stacks und Warteschlangen sind fundamentale ADTs.
Implementation als Verkettete Liste.
Arrayimplementation.
Verschiedene Performanceeigenschaften.
Viele Anwendungen.
Taschenrechner.
Drucker und PostScript language.
Arithmetische Ausdrücke.
Funktionimplementation im Compiler.
Web browsing.
. . .
G. Zachmann Informatik 1 - WS 05/06 Datenstrukturen 61