Python-Implementierung eines Stacks mittels Array

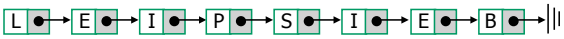
```
class Stack(object):
   def __init__( self ):
       self.s = [None]
       self.N = 0          # wir verwalten Stack-Größe selbst,
                           # zu "Demo"-Zwecken (wäre nicht nötig
   def isEmpty(self):      # in Python)
       return self.N == 0

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

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

Erzeugt Liste der Länge len(s)
mit None initialisiert

Implementierung mit Liste



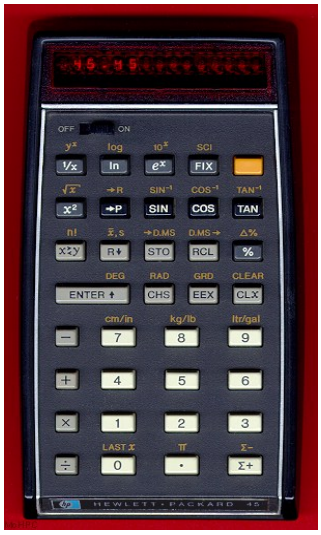
- **`push()`** fügt ein Element am Kopf der Liste hinzu
- **`pop()`** entfernt erstes Element (am Kopf) der Liste
- **`isFull()`:** nicht sinnvoll (bzw. liefert immer den Wert **`false`**)

- Vorteil
  - Speicherbedarf für einen Stack ist häufig nicht bekannt
  - Bei Array muß max. Speicherplatz festgelegt werden, oder Resize
- Nachteil:
  - Mehr Verwaltungsaufwand
  - Möglicherweise 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)

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

- Postfix-Ausdrücke:
  - auch umgekehrte polnische Notation genannt (UPN; RPN = *reverse polish notation*)
  - Aufbau von Ausdrücken: Erst die Operanden, dann der Operator

J. Lukasiewicz (1878-1956)

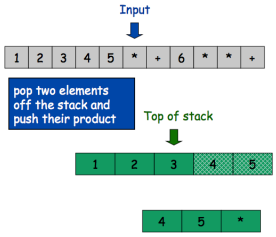
- Beispiel:

  Infix-Notation: (2+4)! / (11+4) ⇒

  Postfix-Notation: 2 4 + ! 11 4 + /

- Abarbeitung von Postfix-Ausdrücken: verwende Stack von Zahlen
  - Der Ausdruck wird von links nach rechts gelesen
  - Ist das gelesene Objekt ein Operand, wird es auf den Stack ge-push-t
  - 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 auf dem Stack.

- Dies ist eine systematische und einfache Methode, 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 nach dezimal um
% postfix.py
7 16 * 5 + 16 * 3 + 16 * 1 +
30001         Horner-Schema
```

Der Algorithmus in Python

```
stack = Stack()

s = read_word()  # Ann. liest ein Wort bis zum naechsten Space
while s != "":
   if s == "+":
      stack.push( stack.pop() + stack.pop() )
   elif s == "*"
      stack.push( stack.pop() * stack.pop() )
   else
      stack.push( int(s) )       # Ann.: nur Integer-Operanden

   s = read_word()

print stack.pop()
```


Infix → Postfix

- Aufgabe: Konvertierung von Infix- nach Postfix-Notation
- Beobachtung: Operanden erscheinen in derselben Reihenfolge, Operatoren nicht
- Algorithmus:
  - Linke Klammern → ignorieren
  - Rechte Klammern → pop und print
  - Operator → push
  - Operand → ausgeben

```
% ./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 == "+" or s == "*":
       stack.push(s)
   elif s == ")":
       print stack.pop(), " ",
   elif s == "(":
       pass                  # = NOP
   else:           # must be operator
       print s, " ",
```

- Postfix-Ausdrücke kommen immer noch in der Praxis vor
- Beispiele:
  - Taschenrechner (z.B. von HP)
  - Stackorientierte Prozessoren
  - Postscript-Dateien
- Weitere Anwendungen für Stack: Aufruf von Funktionen
  - Bei jedem 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.
- Algorithmus: bearbeite jedes Zeichen, eins nach dem anderen
  - Linke Klammer: push
  - Rechte Klammer: pop und prüfe ob es die selbe "Klammerklasse" ist
  - Ignoriere alle anderen Zeichen
  - Ausdruck ist balanciert ⇔ der Stack ist nach Beendigung leer

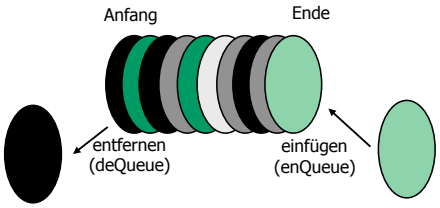
String	Balanced
`( ) ( ( ) )`	`true`
`( ( ( ) ( ) ) )`	`true`
`( ( ) ) ) ( ( )`	`false`
`[ ( [ ] ) ]`	`true`
`[ [ ( ] ) ]`	`false`
`a[2*(i+j)]= a[b[i]]`	`true`

```
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
     # else: Zeichen c ignorieren
  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

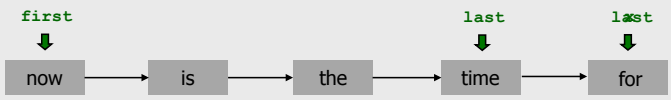
- **`enqueue`** Füge ein neues Objekt in die Warteschlange ein.
- **`dequeue`** Lösche und gebe aus das Objekt, das zuerst eingefügt wurde.
- **`isEmpty`** Ist die Warteschlange leer?

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

A simple queue client

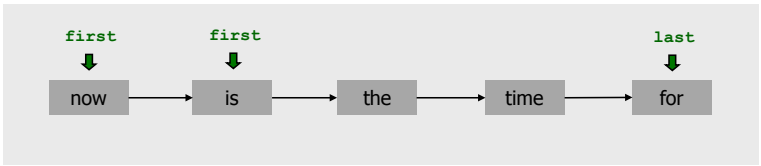
Implementierung als verkettete Liste

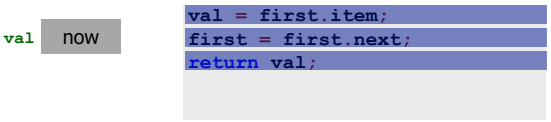
- **`enqueue`**




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

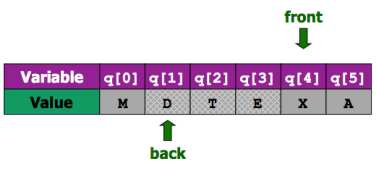
- **dequeue**

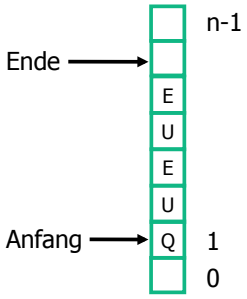




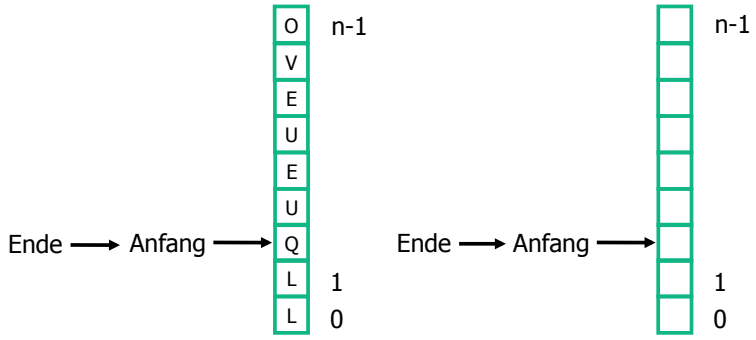
Implementierung mit Array

- 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 ListElem:          # nested class
      item = None           # satellite data
      next = None           # "pointer"
   def isEmpty(self):       # Methode in Queue
      return first == None
   def enqueue(self, item):
      x = ListElem()
      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 ?

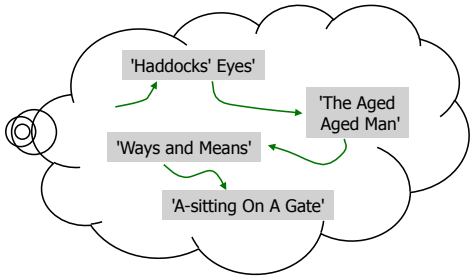
- Generelle Regel für Datenstrukturen-Entwurf: checke die "Ausnahmen"!! (Randfälle, *boundary cases*)
  - Stimmt die Funktion für den Fall, daß 0 oder 1 Element vorhanden ist?
  - Was passiert, wenn Cursor am Ende oder auf None steht?
  - ...

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 ...

Folgerung

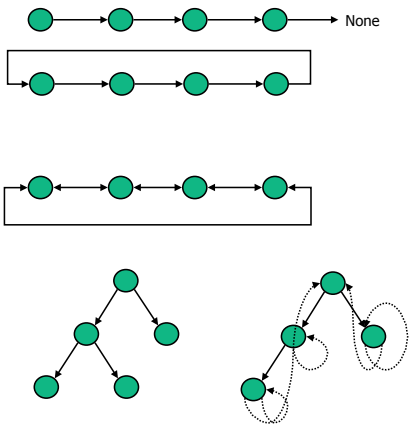
- *Sequential allocation*: unterstützt Indizierung, feste Größe.
- *Linked allocation*: variable Größe, unterstützt sequentiellen Zugriff.
- Verkettete Strukturen sind eine zentrale Datenstruktur und -abstraktion.



Alice should have done this!

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.
  - . . .