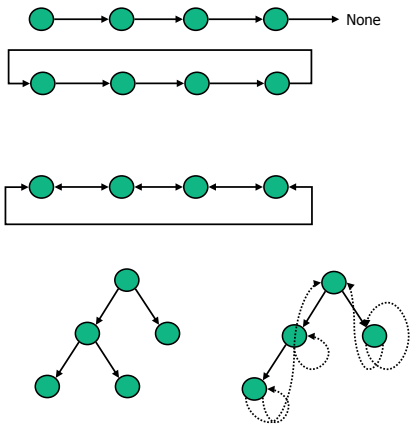
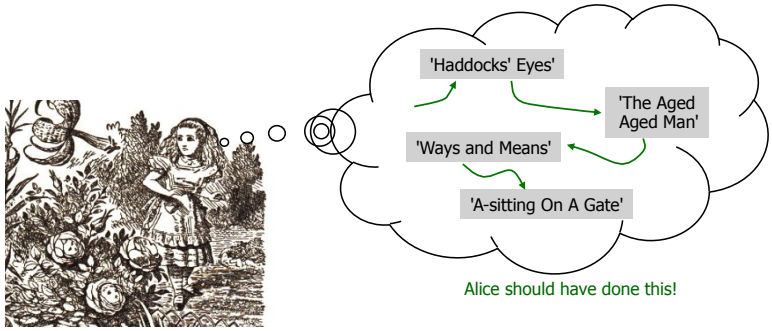
Vergleich von Varianten verketteter Strukturen

- Linked list
- Circular linked list
- Doubly linked list
- Später:
  - Binary tree
  - Patricia tries



Zusammenfassende Bemerkungen

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



- Stacks und Queues sind fundamentale astrakte Datentypen (ADTs):
  - Implementation als Verkettete Liste
  - Arrayimplementation
  - Verschiedene Performanceeigenschaften
- Viele Anwendungen:
  - Taschenrechner
  - Drucker und PostScript language
  - Arithmetische Ausdrücke
  - Funktionimplementation im Compiler
  - Web browsing
  - . . .

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

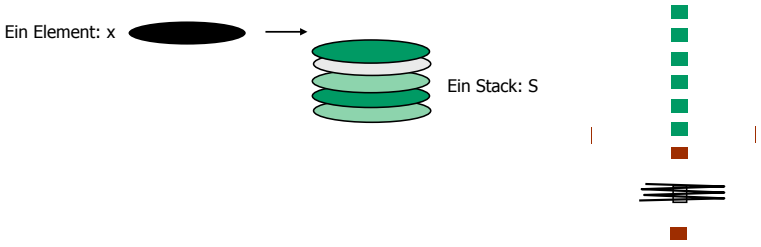
- Stack
  - Entferne das Objekt, das zuletzt hinzugefügt wurde
  - Heißt daher auch: LIFO = *"last in first out"*
  - Analogie: Cafeteriabehälter, surfen im Web.
  - „ Die letzten werden die ersten sein.“

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

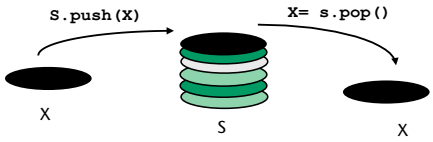
- (Deutsch: Stapel, Kellerspeicher)
- Zunächst: abstrakte Datenstruktur, Container-Datentyp
- Elemente können einfügt und wieder entfernt werden
- Direkter Zugriff nur auf das zuletzt eingefügte Element *(last in first out)*



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()
    ```

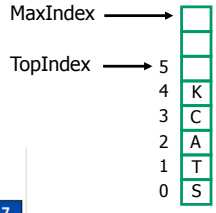
    ist äquivalent zu:

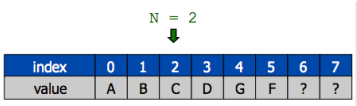
    ```
    X = S.pop()
    S.push(X)
    ```

  - **`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 vermutlich auch)

Python-Implementierung eines Stacks mittels Array

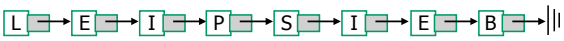
```
class Stack(object):
   def __init__( self ):
       self.s = [None]
       self.N = 0           # wir verwalten Stack-Größe selbst,
                            # zu "Demo"-Zwecken (wäre nicht nötig
   def isEmpty(self):       # in Python)
       return self.N == 0

   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: ein 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 die 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

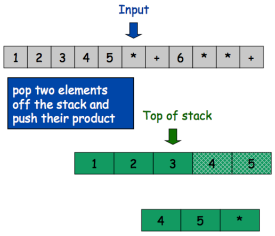
J. Lukasiewicz (1878-1956)

- Postfix-Ausdrücke:
  - auch umgekehrte polnische Notation genannt (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 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 (Teil von PDF)

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

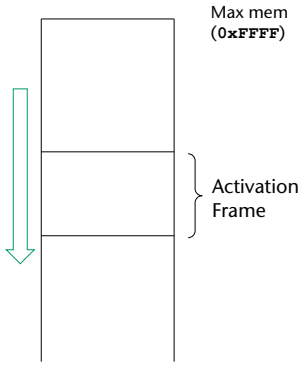
String	Balanced
( ) (( ))	true
((( ) ( )))	true
(( ))) (( )	false
[([ ])]	true
[[( ])]	false
a[2*(i+j)]= a[b[i]]	true


```
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()
```

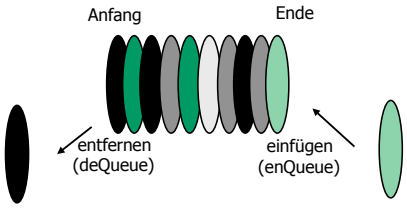
Anwendung: Implementierung von Function Calls

- Vereinfachung: Betrachte im Folgenden nur statisch typisierte Sprachen
- Wo werden lokale Variablen (und Argumente) gespeichert?
- Problem: eine Funktion kann auf sehr verschiedenen "Wegen" aufgerufen werden ...
- Lösung: Stack
- Definition: Activation Frame := alle Argumente + alle lokalen Variablen + einiges andere
- Andere Namen: activation record, procedure frame, ...
- Mehrere Funktionsaufrufe ergeben den sog. Call Stack



Queue

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



Operationen

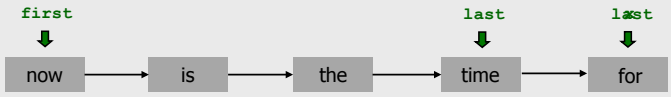
- **enqueue** Füge ein neues Objekt in die Warteschlange ein.
- **dequeue** Lösche und gebe aus das Objekt, das zuerst eingefügt wurde.
- **isEmpty** Ist die Warteschlange leer?

```
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

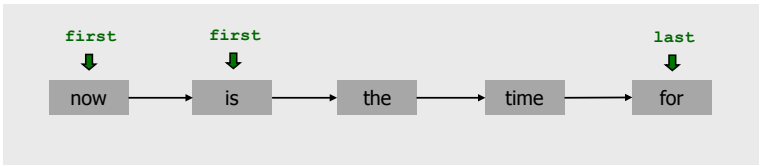
Implementierung als verkettete Liste

- **enqueue**




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

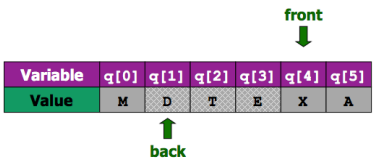
- **dequeue**

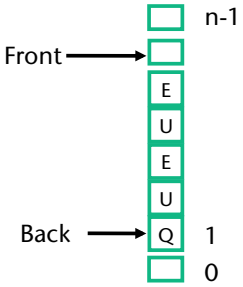



```
val = first.item;
first = first.next;
return val;
```

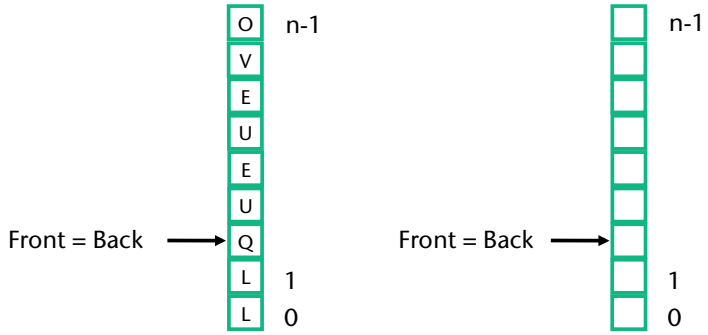

Implementierung mit Array

- 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





- **`Front == Back`** → 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
      self.item = None           # satellite data
      self.next = None           # "pointer"
   def isEmpty(self):       # Methode in Queue
      return self.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)
  - Data buffers (MP3 player, portable CD player, Tastatur)
- Simulation
  - In Fertigungsprozessen: Objekte auf Förderbändern verhalten sich wie in einer Warteschlange
  - Wartezeiten bei McDonalds oder Call-Center, oder Verkehr vor Tunnel, oder ...