


## 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!)

G. Zachmann Informatik 1 - WS 05/06 Datenstrukturen 26

## 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.“



Pop-A-Filter

## Queue

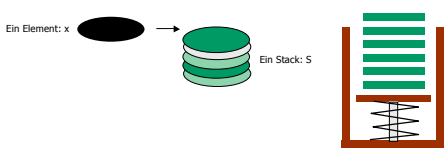
- Entferne das Objekt, das zuerst eingefügt wurde
- Daher auch: FIFO = "first in first out"
- Analog: Registrar's line.
- „Wer zuerst kommt, malt zuerst“ („first come, first serve“)



G. Zachmann Informatik 1 - WS 05/06 Datenstrukturen 27

## Stack

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

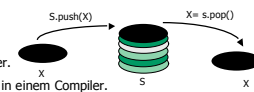


Ein Element: x → Ein Stack: S

G. Zachmann Informatik 1 - WS 05/06 Datenstrukturen 28

## 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.

G. Zachmann Informatik 1 - WS 05/06 Datenstrukturen 29

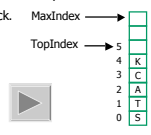
- Anzahl Operationen nicht minimal:
  - Eigentlich reichen `push()` und `pop()`

```
x = s.peek()
ist äquivalent zu:
x = s.pop()
s.push(x)
```
  - `peek()` ist aber effizienter und wird häufig benötigt
- weitere Operationen
  - `isEmpty()`: true, falls kein Element mehr auf den Stapel paßt
  - `clear()`: entfernt alle Elemente vom Stack

G. Zachmann Informatik 1 - WS 05/06 Datenstrukturen 30

## Stack Implementation (Array)

- Implementierung eines Stacks mit Hilfe eines Arrays.
  - `s = array`, `N = #` Objekte auf dem Stack. MaxIndex →
  - `push`: speichere Objekt in `s[N]` TopIndex → 5
  - `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.

G. Zachmann Informatik 1 - WS 05/06 Datenstrukturen 31

### 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,
  - Neues Array erzeugen mit Größe  $N+c$
  - Elemente vom alten Array ins neue Array umkopieren
  - 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:  $\sum_{i=1}^{N/c} i \cdot c \approx \frac{N^2}{2c}$

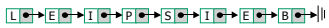
G. Zachmann Informatik 1 - WS 05/06 Datenstrukturen 32

### 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 ;-)

G. Zachmann Informatik 1 - WS 05/06 Datenstrukturen 33

### 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"

G. Zachmann Informatik 1 - WS 05/06 Datenstrukturen 34

### 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)

G. Zachmann Informatik 1 - WS 05/06 Datenstrukturen 35

### Python-Code

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


    def isEmpty(self):
        return N == 0

    def push(self, item):
        if N >= len(s):
            s.extend( len(s) * [None] ) # Länge verdoppeln
            s[N] = item
            N += 1 # Erzeugt Liste der Länge len(s)
                  # mit None initialisiert
        else:
            s.append(item)

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

G. Zachmann Informatik 1 - WS 05/06 Datenstrukturen 36

### An Ancient Calculator




HP 45.  
 Preis Im Jahr 1973: \$395.  
 (Das entspricht \$1600 im Jahr 2002.)  
 Was fehlt auf der Tastatur?

G. Zachmann Informatik 1 - WS 05/06 Datenstrukturen 37

### Beispiel-Anwendung für Stack: Postfix-Auswertung

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

Infix-Notation:  $(2+4)! / (11+4) \Rightarrow$   
 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.



J. Lukosewicz (1878-1956)

G. Zachmann Informatik 1 - WS 05/06 Datenstrukturen 38

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

```

% postfix.py
1 2 3 4 5 * + 6 * * +
6625 Infixausdruck: (1+((12*(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
  
```

G. Zachmann Informatik 1 - WS 05/06 Datenstrukturen 39

### Python Code

```

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()
  
```

postfix.py

G. Zachmann Informatik 1 - WS 05/06 Datenstrukturen 40

### 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, " ",
  
```

G. Zachmann Informatik 1 - WS 05/06 Datenstrukturen 41

- 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

G. Zachmann Informatik 1 - WS 05/06 Datenstrukturen 42

### 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

G. Zachmann Informatik 1 - WS 05/06 Datenstrukturen 43

```
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()
```