

Daten sind definiert durch

- den Wertebereich
- die Operationen

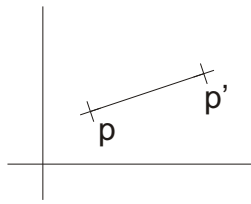
Datentypen und Datenstrukturen

- Beispiel: Stack und Queue mit Anwendungen
- Parsing arithmetischer Ausdrücke in polnischer Notation

Beispiel:

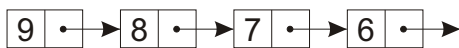
```
class A {
    float a,b;
    public void verschiebe(A);
    public void spiegle (A,A);
}
```

A <- Punkt



```
class B {
    float a,b;
    public float mult(B);
    public float add(B);
    public float betrag();
}
```

verkettete Liste:



```
B <- complex Number
float betrag() {
float erg;
erg = a*a + b*b
```

Abstrakter Datentyp (ADT)

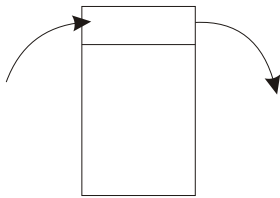
- 1) Signatur der Methoden
- 2) Verhalten der Methoden

Datenstruktur:

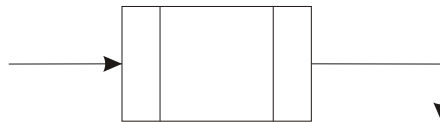
Implementierung auf algorithmischer Ebene

Implementierung in einer Programmiersprache

LIFO last in first out



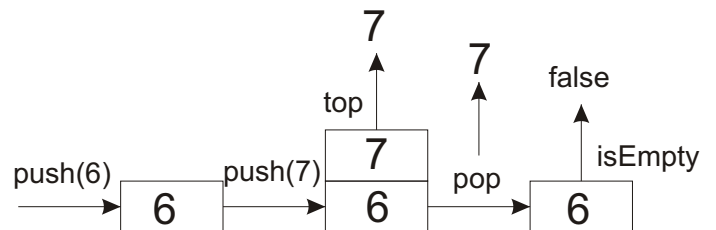
FIFO first in first out



Stack - Tellerstapel

```
void push(Object)
Object pop()
Object top
boolean isEmpty
```

Queue: Schlange an der Konzertkasse



Datenstruktur: Stack als Array

private Datenfelder

Feld elements[0,...,N-1]

Höhe des Stapels: num

Größe des Feldes N

isEmpty:

falls num == 0

return true;

andernfalls

false;

top:

falls (isEmpty() == false)

return elements[num-1];

andernfalls

Fehler!

pop:

return top();

num = num - 1;

(a + b) * (c + d)

Präfix-Notation MUL AX, BX

Infix-Notation a+b

Postfix-Notation ab+

auch polnische Notation, Vorteil: Ausdruck ist ohne Klammern eindeutig

(2+3) * 4 entspricht 23+4*

(2+(3*4)) entspricht 234*+

(a+b) * (c+d) ab+ cd+ *

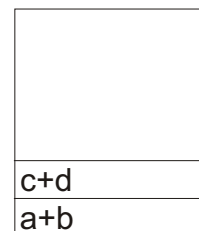
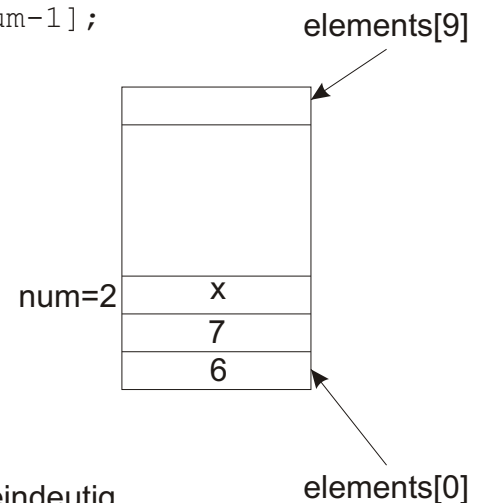
push (a) push (c)

push (b) push (d)

pop (a) pop (c)

pop (b) pop (d)

push (a+b) push (c+d)



Auswertung von Postfix-Ausdrücken mit Stack:

Durchlaufe den Ausdruck von links nach rechts:

- falls du einen Operanden o liest: push(o)

- falls du einen Operator # aus {+, -, *, /} liest:

o(1)=pop(), o(2)=pop()

o(3)=o(1)#o(2)

push o(3)

Am Ende enthält der Stack ein Element (mehr als zu Beginn). Das ist der Wert des Ausdrucks.

Auswertung von Postfix-Ausdrücken am Beispiel:

a b+c d- e f+*-

push(a); push(b); pop()→b; pop() → a; push(a+b);

push(c); push(d); pop()→d; pop() → c; push(c-d);

push(e); push(f); pop()→f; pop() → e; push(e+f);

pop()→e+f; pop()→c-d; push((e+f)*(c-d));

pop()→(e+f)*(c-d); pop()→a+b; push(Differenz);

Bei einer konkreten Aufgabe sind a, b, ... Zahlen. So sieht man die Korrektheit.

Warteschlangen: Queue

- Stack: LIFO-Prinzip (last in first out)
- Stack: FIFO-Prinzip (first in first out)

Object front(): Liefert das nächste Element; das was am längsten gewartet hat
enqueue(Object): Fügt ein Objekt der Menge hinzu
Object dequeue(): Liefert dasselbe Element wie front() und entfernt es zugleich
boolean isEmpty(): Test, ob noch ein Objekt wartet

Datenstruktur: Queue mit einem Feld

Die naheliegende Idee: Die wartenden Objekte werden in einem Feld gespeichert

Problem: Der genutzte Bereich im Feld "wandert", da dequeue() stets das erste Element dieses Bereiches löschen muss (das Element auf der gegenüberliegenden Seite von der Stelle, wo angehängt wird)

Lösung: Wir fassen das Feld elements[0,...,N-1] als zyklisch auf
elements[0] ist Nachfolger von elements[N-1]
(Veranschaulichung als Ring)

Der genutzte Bereich kann von der Form
elements[i], elements[i+1], ..., elements[N-1],
elements[0], elements[1], ..., elements[j]
sein.

Datenstruktur: Queue mit einem Feld

Was ist ein Nachteil dieser Datenstruktur?

Es kann zum StackOverflow kommen. Es handelt sich um eine statische Datenstruktur, die Kapazität steht von Beginn an fest.

Erste Idee: Ist die Kapazität erschöpft, wird von der Methode push() ein neuer, größerer Stack angelegt.

Nachteil: Daten müssen umgespeichert werden.

Alle anderen Stack-Operationen haben Laufzeit $O(1)$.

push() mit Umspeichern hat Laufzeit $O(N)$!

Ergo: Ungeeignetes Verfahren

Wir werden später eine dynamische Variante des Stacks kennenlernen (also eine andere Datenstruktur für den ADT Stack), bei der alle Operationen eine Laufzeit $O(1)$ haben. Stichwort: Einfach verkettete Listen.

Laufzeit

Was ist die Laufzeit einer Berechnung?

Der zeitliche Ressourcenverbrauch (ist noch genauer zu fassen)

Die Laufzeit eines Algorithmus ist die Laufzeit in Bezug auf mögliche Eingaben (später: worst-case, average-case Analyse).

Ziel: Verschiedene Algorithmen vergleichen

Dazu muss der Begriff Laufzeit rechnerunabhängig sein

Rechnerunabhängige Laufzeit? Wie soll das möglich sein?

Bei diesem Ansatz entsteht ein neues Problem: Die Anzahl der Schritte ist nicht nur vom Rechner abhängig, sondern sogar davon, wie ich den Algorithmus aufschreibe. Und was ist überhaupt ein Schritt?

Nur scheinbares Problem:

- Wir interessieren uns nicht für die absolute Zahl an Schritten, sondern für das Wachstum mit steigender Eingabegröße

Diese Thematik wird in anderen Materialien fortgeführt. Wir fassen zusammen:

Die Laufzeit T eines Algorithmus A ist eine Funktion in der Eingabegröße $T(n)$ gibt die Anzahl der Schritte an, die A bei einer Eingabe der Größe n ausführt.

Laufzeit eines Algorithmus

Beispiel: Algorithmus, der alle Elemente eines Arrays auf 0 setzt.

Größe des Problems: Anzahl n der Array-Elemente

Laufzeit $f(n)$ auf Maschine L: $(29n + 13) \mu\text{s}$

Laufzeit $f(n)$ auf Maschine M: $(17n + 52) \mu\text{s}$

Vernachlässigen wir den konstanten Summanden und den konstanten Faktor, können wir sagen, dass die Komplexität des Algorithmus von der Ordnung n ist (Lineare Ordnung), Schreibweise: $f(n) = O(n)$.