Übung zu
Algorithmen und Datenstrukturen (für ET/IT)

Wintersemester 2012/13

Jakob Vogel

Computer-Aided Medical Procedures
Technische Universität München

Objektorientierung

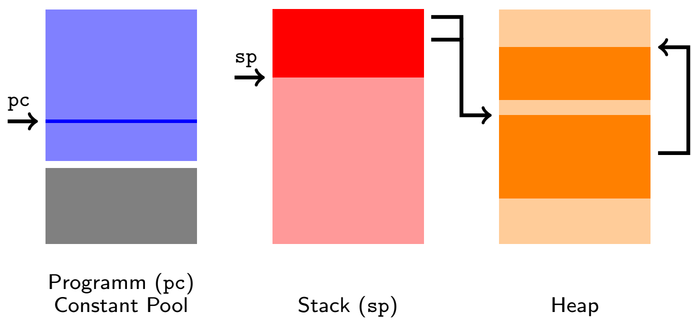
- Zusammengesetzt aus *Attributen* (*Member Variables*)
- Zusätzlich dazu *Methoden*

```
struct Kreis
{
    float x, y;
    float radius;

    // Verschiebt Kreis um (dx/dy) und gibt die Laenge der
    // Bewegung zurueck
    virtual float verschiebeKreis(float dx, float dy)
    {
        x += dx;
        y += dy;
        return std::sqrt(dx*dx + dy*dy);
    }
};
```

- Sichtbarkeiten, Vererbung und Polymorphie vernachlässigen wir (leider) bis auf weiteres...

Erinnerungen an den Heap...



- Notwendig bei variablen Platz-Anforderungen
- Manuelle Anforderung und (einmalige!) Freigabe
  - C : `malloc(...)` und `free(...)`
  - C++: `new`/`new[]` und `delete`/`delete[]`

Arrays auf dem Heap

- Arrays beliebigen Typs liegen als lokale Variablen **mit fixer Länge** auf dem Stack:

```
// Laenge in Bytes: 5*sizeof(Klasse)
Klasse array[5];
```

- Man kann solche Arrays auch auf dem Heap anlegen und dann die Größe zur Laufzeit wählen:

```
Klasse* array = new Klasse[n];
// ...
delete[] array;
```

- Jeder belegte Bereich muss nach Nutzung geräumt werden!
- Klassen wie `std::vector<...>` nehmen viel Arbeit ab!

Objekte auf dem Heap

- Objekte liegen bisher als lokale Variablen auf dem Stack:

```
Klasse obj( /* Hier stehen Konstruktor-Parameter! */ );
```

- Man kann sie auch auf dem Heap anlegen, und geht dann ähnlich wie bei Arrays vor:

```
Klasse* objPtr = new Klasse( /* Konstruktor-Parameter */ );
// ...
delete objPtr;
```

- Auch hier muss natürlich jeder belegte Speicherbereich nach Nutzung geräumt werden!
- Es gibt auch „Helfer“ hierfür, wie `boost::shared_ptr<...>`!

Zeiger

- Ein *Zeiger* (*Pointer*) ist ein Verweis auf einen Speicherbereich (Start-Adresse), geschrieben `<typ>*`.
- Der Operator `new` liefert einen solchen Verweis zurück.
- Man kann zu jeder Variable die Speicheradresse erfahren:

```
int wert = 12345;
int* adresseVonWert = &wert;
```

- Ebenso kann man zu jedem Zeiger das Objekt erhalten:

```
int* adresse = ...;
int wertBeiAdresse = *adresse;
```

- Arrays sind – streng genommen – auch nur Zeiger!

```
int wert = 12345;
int* adresseVonWert = &wert;
std::cout << adresseVonWert[0] << std::endl;
std::cout << *adresseVonWert   << std::endl;
```

Objekte auf dem Heap

- Der Typ eines Heap-Objekts ist nicht mehr „`Klasse`“, sondern „Zeiger auf `Klasse`“

```
Klasse obj;

// Member-Variablen
obj.variable = 12345;

// Member-Funktionen
obj.funktion();

// Ende Scope
```

```
Klasse* objPtr = new Klasse;

// Member-Variablen
objPtr->variable = 12345;

// Member-Funktionen
objPtr->funktion();

delete objPtr;
```

- Beachten Sie den Operator `->` beim Zeiger!
- Konstruktor und Destruktor werden jeweils korrekt aufgerufen.

☞ Code

Boolsche Logik

- Wahrheitswerte: 0/1, *wahr*/*falsch*, *true*/*false*
- „Grundrechenarten“:
  - $\neg$ (Negation, NOT)
  - $\wedge$ (Konjunktion, AND)
  - $\vee$ (Disjunktion, OR)
- Weitere Operationen sind etwa
  - $\oplus$ (XOR)
  - $\uparrow$ (NAND)
  - $\downarrow$ (NOR)
  - $\Rightarrow$ (Implikation)
  - $\Leftrightarrow$ (Äquivalenz)

Wahrheitstabellen

- Wegen der kleinen Wertemenge kann man alle Kombinationen oft direkt aufschreiben.
- Beispielsweise NAND, $\neg(a \land b) = a \uparrow b$:

a	b	$a \land b$	$\neg(a \land b)$	$a \uparrow b$
0	0	0	1	1
0	1	0	1	1
1	0	0	1	1
1	1	1	0	0

Eingabe-Kombinationen

Aufbau aus Teilausdrücken (linke Seite)

Aufbau aus Teilausdrücken (rechte Seite)

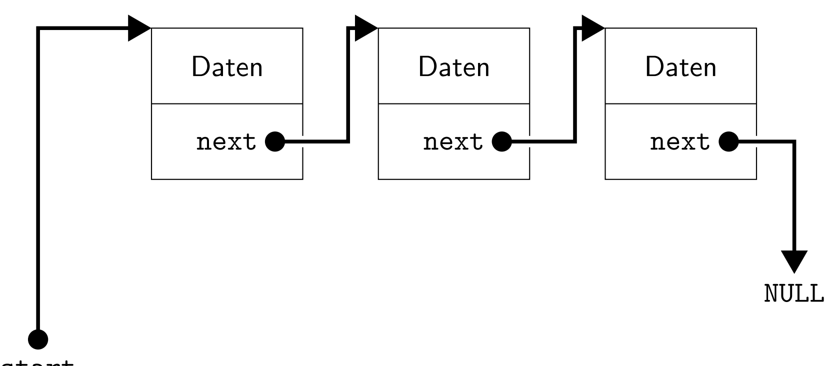
Lösungsblatt

Lazy Evaluation

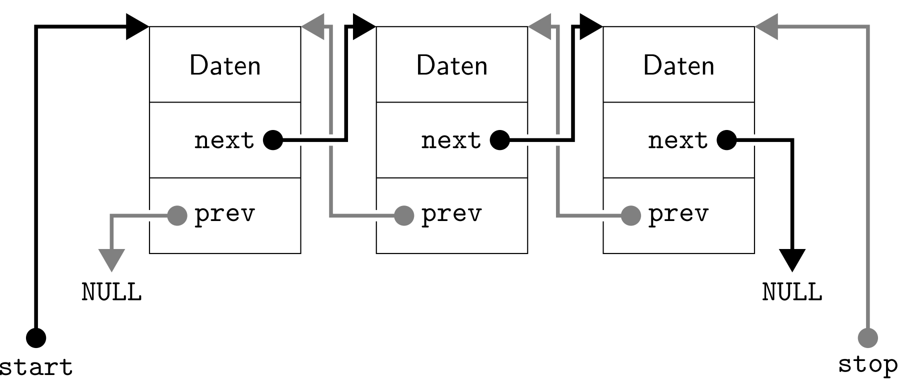
- Es gilt immer, unabhängig von Wahrheitswert x:
  - $\textit{wahr} \vee x = \textit{wahr}$
  - $\textit{falsch} \wedge x = \textit{falsch}$
- Trotzdem werten `|` und `&` immer beide Seiten aus, da *bitweise*.
- Die *logischen* Varianten `||` und `&&` kürzen hier ab (*lazy evaluation*).
- Normalerweise keine Seiteneffekte bei der Auswertung logischer Ausdrücke!

☞ Code

Einfach verkettete Liste



Doppelt verkettete Liste

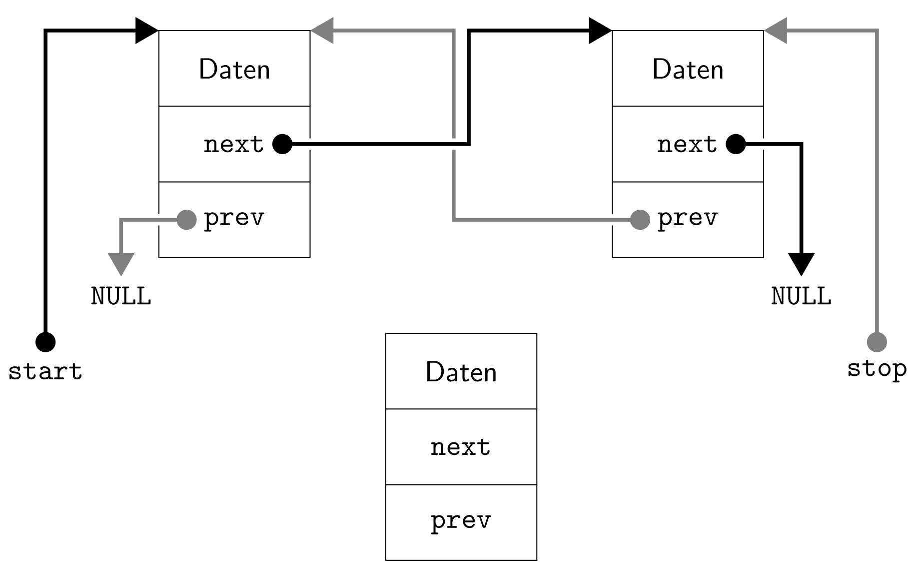


Doppelt verkettete Liste – Implementation

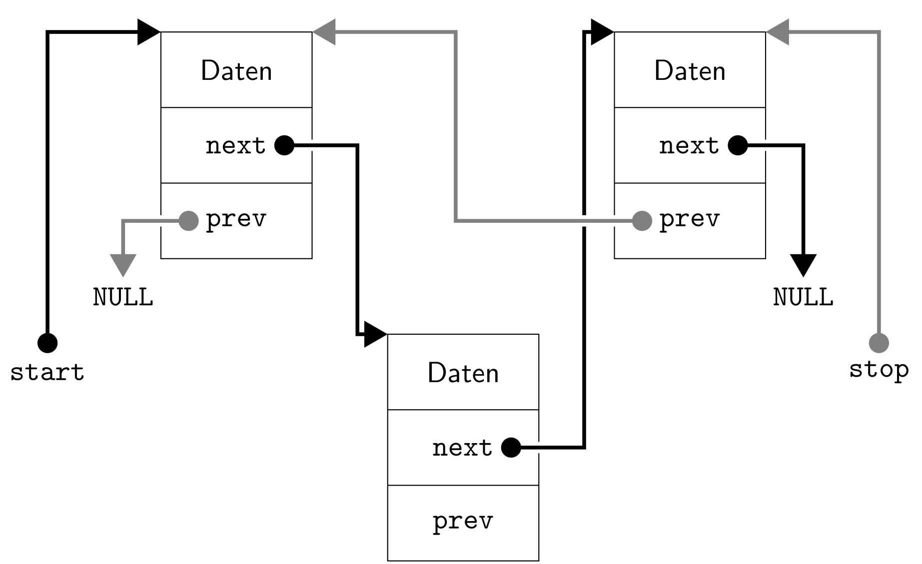
- Kapselung mit Klassen
- Einfach zu nutzendes Interface
- Aussagekräftige Ausgabe

☞ Code

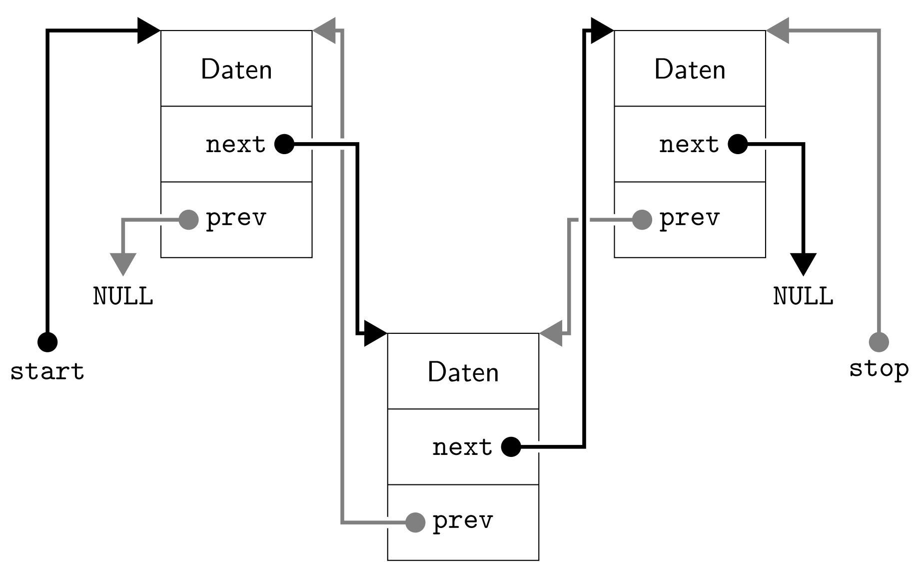
Doppelt verkettete Liste – Einfügen



Doppelt verkettete Liste – Einfügen



Doppelt verkettete Liste – Einfügen

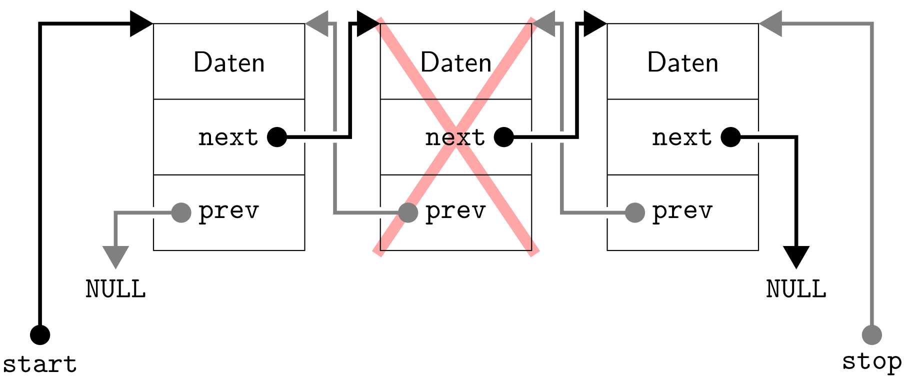


Doppelt verkettete Liste – Einfügen

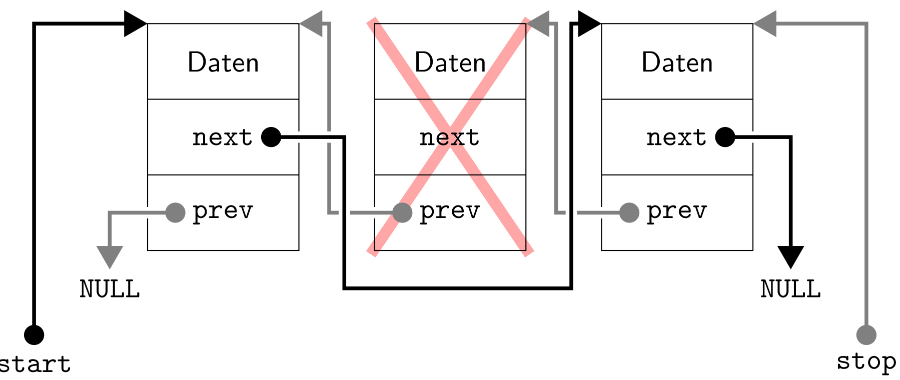
- Zeiger nicht verwechseln oder vergessen!
- Ketten konsistent halten!
- `start` und `stop` nicht vergessen!

☞ Code

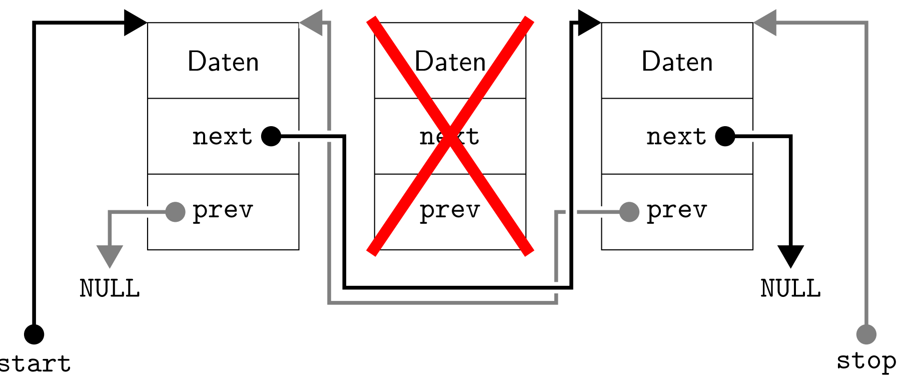
Doppelt verkettete Liste – Löschen



Doppelt verkettete Liste – Löschen



Doppelt verkettete Liste – Löschen



Doppelt verkettete Liste – Löschen

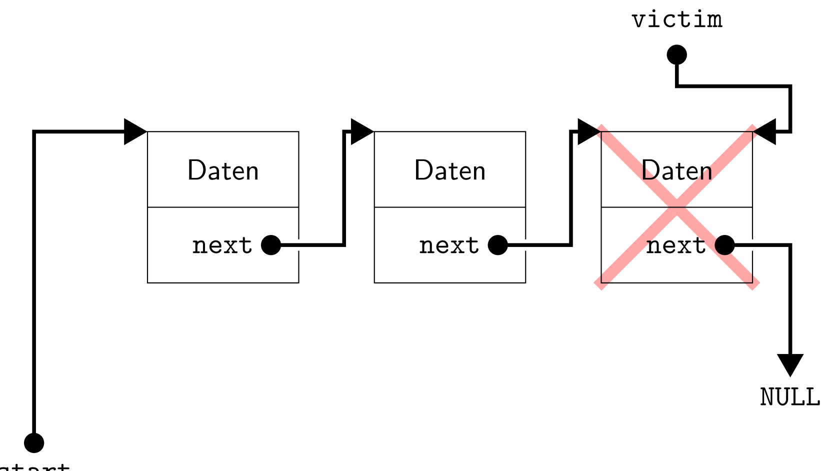
- Zeiger nicht verwechseln oder vergessen!
- Ketten konsistent halten!
- `start` und `stop` nicht vergessen!

☞ Code

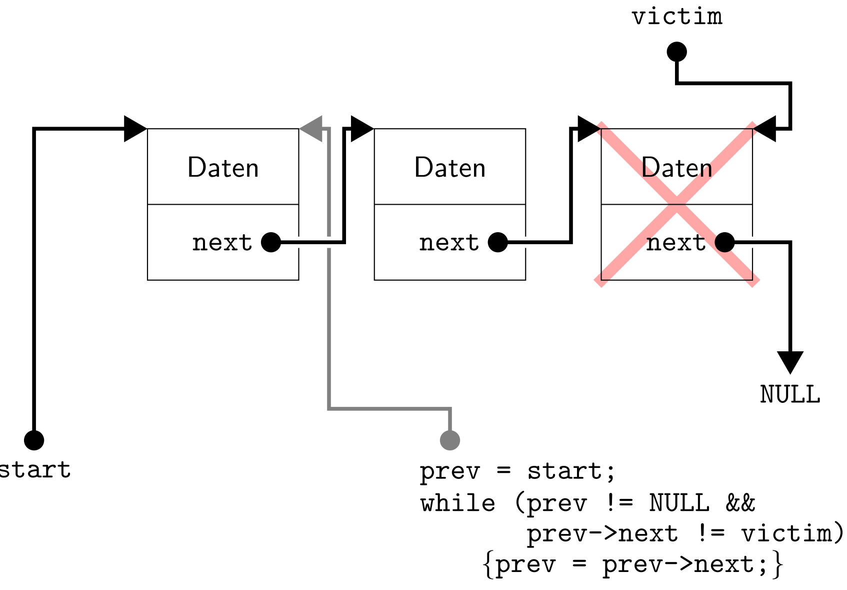
Doppelt verkettete Liste – Vor- und Nachteile

- \+ Durchlauf in beiden Richtungen möglich
- \+ Einfügen und Löschen ohne Durchläufe zur Identifikation der Nachbarn
- – Weiterer Speicherplatz für Zeiger `prev` nötig
- – Mehr Verwaltung bei Strukturänderungen

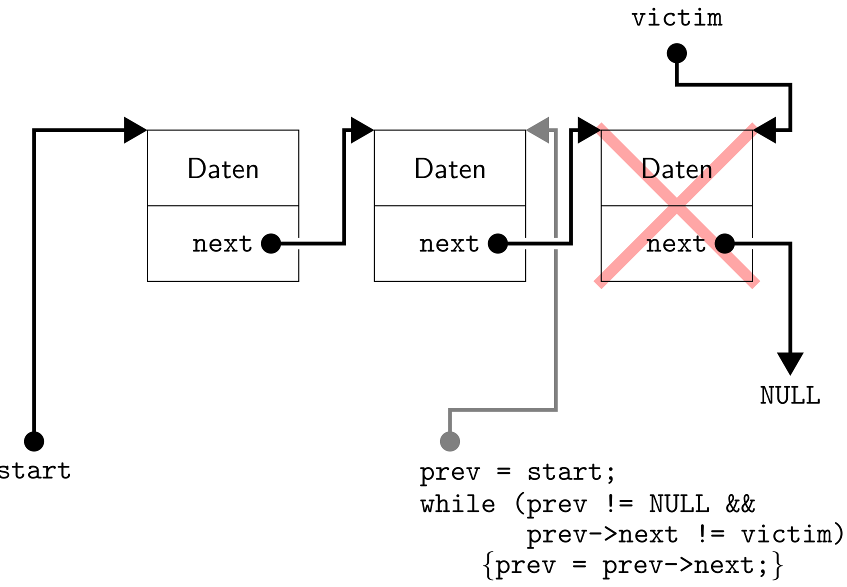
Einfach verkettete Liste – Löschen



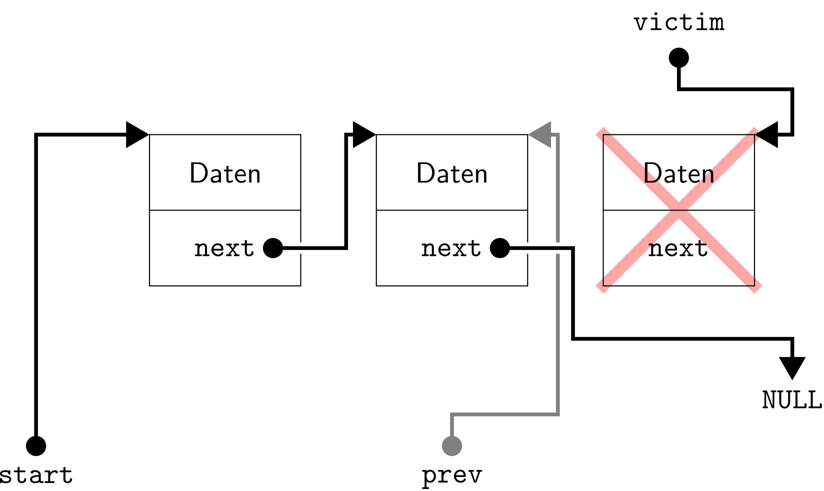
Einfach verkettete Liste – Löschen



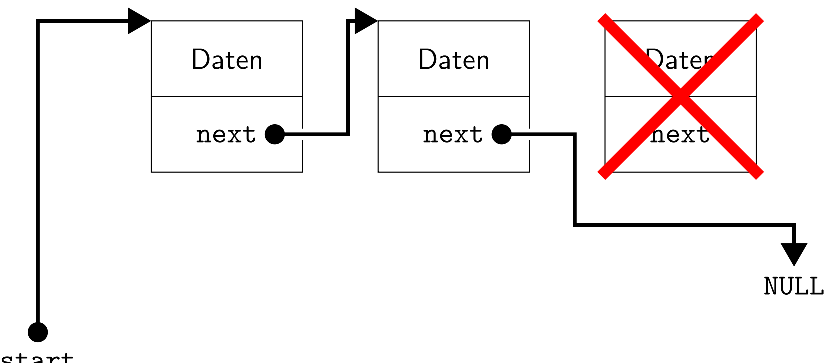
Einfach verkettete Liste – Löschen



Einfach verkettete Liste – Löschen



Einfach verkettete Liste – Löschen



Verkettete Listen

- Daten in „Containern“ mit Zeiger(n) auf Nachbar(n)
- Reihenfolge der Container im Speicher nicht definiert
- Durchlauf durch Verfolgen der Zeiger
- Dynamische Größe möglich