LMU LUDWIG-MAXIMILIANS-UNIVERSITY MUNICH
DEPARTMENT INSTITUTE FOR INFORMATICS
DATABASE SYSTEMS GROUP

Algorithmen und Datenstrukturen
Kapitel 2: Datenstrukturen

Skript zur Vorlesung

Algorithmen und Datenstrukturen

Sommersemester 2015

Ludwig-Maximilians-Universität München

(c) PD Dr. Matthias Renz 2015,

basierend auf dem Skript von Prof. Dr. Martin Ester, Prof. Dr. Daniel A. Keim, Dr. Michael Schiwietz und Prof. Dr. Thomas Seidl

Datenstrukturen allgemein

Man unterscheidet zwischen:

- **Statische Datenstrukturen:**
  - Besitzen ein festes Speicherschema, das sich während der Programmlaufzeit nicht ändert.
- **Dynamische Datenstrukturen:**
  - Struktur ist während der Ausführung eines Programmes veränderbar
  - Organisation beruht auf Zeigern
  - Für viele Probleme notwendig

2.1. Statische Datenstrukturen

Statische Datenstrukturen

- **Atomare (unstrukturierte) Datentypen** können durch Anwendung von Konstruktoren **zusammengesetzte (strukturierte) Datentypen** definiert werden.

Atomare Datentypen	Strukturierte Datentypen

`boolean`
`char`
`byte, short, int, long`
`float, double`
	Array
	Record

- Jede Komponente ist von atomarem oder strukturiertem Typ.
- Komponenten der „untersten“ Strukturebene sind stets vom atomaren Typ.
- Zugriff und Adressierung strukturierter Datentypen erfolgen über die einzelnen Komponenten.

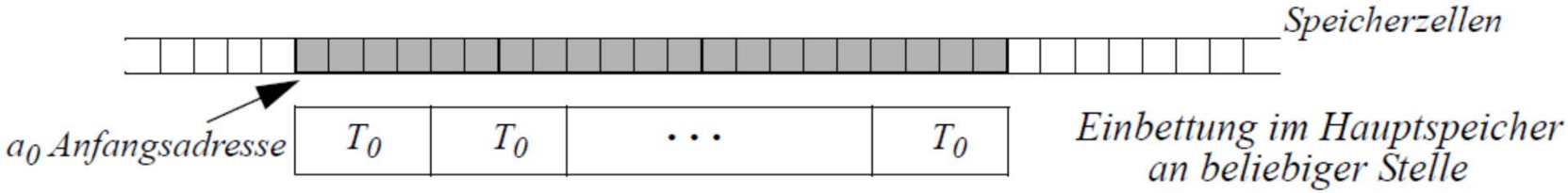
Arrays I

Felder (ARRAYs, Reihungen)

- Dienen zur **indizierten** Aufnahme einer **Reihe von Informationen gleichen Typs**.
- Array-Notation in Java:
  T_0`[] name = new` T_0`[limit]`
  (T_0 heißt Grundtyp, bzw. Komponententyp)
  Beispiel: float[] floatArray = **new float** [50]; *// Array von **float**-Werten*
- Arrays entsprechen **homogenen Produkt** $T_0 \times T_0 \times \cdots \times T_0$.
- **Grundtyp** ist beliebig
  (auch ein Array ist möglich → *mehrdimensionale Arrays*)
- **Indextyp** ist auf einen Teilbereich von **int** beschränkt:
  $[0..limit - 1]$
- Kardinalität eines Arrays: $card(T) = card(T_0)^{limit}$.

Arrays II

- Speicherung durch Aneinanderreihung der Komponenten.



- **Adressfunktion: Adresse der i-ten Komponente** ist

$$a_0 + (i-1) \cdot SIZE(T_0)$$

$SIZE(T_0)$ = Anzahl der von T_0 benötigten Speicherzellen

- Zugriff auf jede der Komponenten in konstanter Zeit möglich, falls T_0 atomar.

Operationen zwischen Arrays

- Operation zwischen ganzen Arrays:

  Nur **Wertzuweisung** ‚=' erlaubt.
- ‚=´-Operation nur bei **exakter Typgleichheit** möglich

- Wertzuweisung kopiert die **Referenz** auf das Array (d.h. seine Anfangsadresse a_0), **nicht seinen Inhalt**.
- Nach der Wertzuweisung existieren als **zwei Referenzen** auf denselben Speicherplatz.
- Vergleiche wie ‚==' oder ‚>' sind nicht möglich.

Beispiele für Arrays


```
// Array von float-Werten
float floatArray = new float[50];

// Array der Klasse Date
Date[] geburtstage = new Date[23];

// Explizite Initialisierung
String[] waehrungen = {„USDollar“, „Euro“, „Yen“};

// Wertzuweisung
float[] floatArray2 = floatArray;
```

Array-Parameter

Problemstellung: Eine Prozedur soll in der Lage sein, Felder unterschiedlicher Größe zu bearbeiten (z.B. bei Sortierung oder Zeichensuche).

→ Array-Parameter ohne statische Festlegung der Indexgrenzen erforderlich

In Java werden die Parameter den aufgerufenen Methoden wie folgt übergeben:

- **call-by-value** (Übergabe einer Kopie des Werts), wenn der Parametertyp ein einfacher Datentyp ist (`int`, `float`, ...)
- **call-by-reference** (Übergabe einer Referenz auf das Objekt), wenn der Parametertyp ein strukturierter Datentyp ist (eine Klasse oder ein Array)

Beispiel für Arrays in Java


```
boolean isInCharArray(char[] a, char x)
{
  for (int i = 0, i < a.length; i++)
    if (a[i] == x) return true;
  return false;
}
```

Anmerkungen:

- Der Index des Arrays beginnt seine Zählung stets bei 0.
- Die aktuelle Größe liefert das Attribut *Arrayinstanz*`.length`.
- Der Zugriff auf die letzte Komponente erfolgt durch *Arrayinstanz*`[length-1]`.

Records I

Verbunde (Records, Klassen)

- Records, bzw. Klassen dienen zur Zusammenfassung **inhomogener**, aber **zusammengehöriger Informationen**.

- Wichtigste Anwendung:
  Repräsentation komplexer, strukturierter Informationen beliebiger Objekte.
- Werden häufig als Baustein größerer Datenstrukturen verwendet (z.B. Array von Objekten)

Records II

- Aufbau einer Klasse T basiert auf innerhalb der Struktur eindeutigen **Selektoren** S_i.

```
class T {
    T_1 S_1;
    T_2 S_2;
    …
    T_n S_n;
}
```

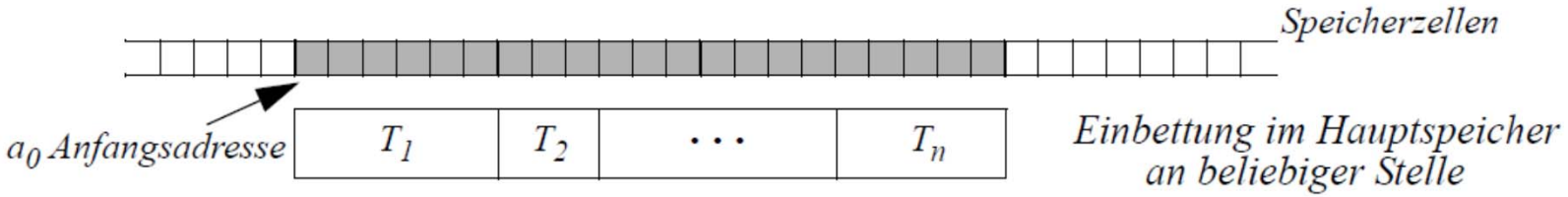
- Records entsprechen **homogenen Produkt** $T_1 \times T_2 \times \cdots \times T_n$.
- Grundtypen T_1, T_2, …, T_n sind beliebig
- Mehrere Selektoren gleichen Typs können auch getrennt werden (durch ‚,‘)

- Kardinalität eines Records:

$$card(T) = \prod_{i=1}^{n} card(T_i)\,.$$

Records III

- Speicherung durch Aneinanderreihen der Repräsentationen der Grundtypen.



- Der Compiler kennt den Speicherbedarf der Grundtypen und kann bereits während des Compilervorganges die **offset**s der einzelnen Selektoren berechnen mit:

$$offset(S_i) = \sum_{j=1}^{i-1} SIZE(T_j)$$

- **Adresse der Komponente S_i:** $a_0 + offset(S_i)$
- Zugriff auf beliegige Komponenten in konstanter Zeit möglich, da die Größe von Records fest ist.

Beispiel für Records in Java


```
// Record Datum
class Datum {
  public int Jahr;
  public int Monat;
  public int Tag;
}
```

```
// Record Person
class Person {
  public String Nachname, Vorname;
  public Datum Geburt, KundeSeit;
}
```

```
Person Kundenliste[] = new Person[300];
int eintrittsalter = Kundenliste[272].KundeSeit.Jahr
                   - Kundenliste[272].Geburt.Jahr;
```

Anmerkung:

- Der Zugriff auf die einzelnen Komponenten erfolgt über den ‚.'-Operator, z.B. T.S_2.

2.2. Dynamische Datenstrukturen

Dynamische Datenstrukturen

- Datenstrukturen heißen **dynamisch**, wenn sie während des Programmlaufes beliebig wachsen oder schrumpfen und ihre Struktur in gewissen Grenzen verändern können.

- Konstruktion **dynamischer Datenstrukturen** durch **Zeiger** („**pointer**“) unterstützt.

Zeiger I

- **Zeiger** sind **Verweise auf eine Objektrepräsentation** im Hauptspeicher
- In Java ist jede Deklaration einer Klasse oder eines Arrays eine **implizite** Verknüpfung des **Instanznamens** (also Objektreferenz) mit einem **Zeiger** T auf die entsprechende Klasse, bzw. Array.

  T_0 `Objektreferenz;`

  Anmerkungen:
  - T_0 ist der Klassenname
  - Objektreferenz wird implizit mit T (= ein Zeiger auf T_0) verknüpft

- T_0 ist der **Grundtyp** von T.

Zeiger II

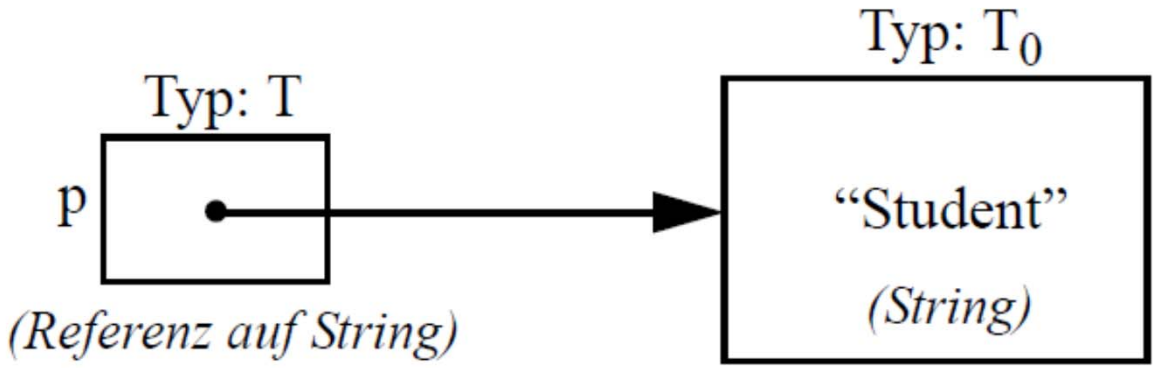
- **Wertebereich** von T:
  Menge aller Zeiger auf Objekte vom Grundtyp T_0, die im bisherigen Verlauf des Programms erzeugt wurden.

  Anm.:
  Im Unterschied zu Assembler-Adressen sind Zeiger also typgebunden.

  Spezieller Wert: $\boldsymbol{null} \in W(T)$,
  = ein Zeiger, der „auf nichts zeigt“ (definierte Adresse „Null“)

Zeiger III

- **Dynamische Speicherzuordnung** durch **new**
- Beispiel in Java:
  `String p = new String(„Student“);`
- Erzeugt im Hauptspeicher „irgendwo“ Speicherplatz für einen Wert (Objekte) vom Typ T_0 (String) und lässt die Objektreferenz (= Zeiger) p auf diesen Speicherplatz zeigen.



Zeiger IV

- **Zugriff** auf referenzierte Objekte erfolgt über **Instanznamen**
- Es gibt **keinen Dereferenzierungsoperator**.
  (wie z.B. „^“ bei der Programmiersprache MODULA-2)

Speicherfreigabe in Java:

- Speicherplatz, der mit **new** reserviert wurde, wird in Java nicht explizit freigegeben
- Automatische Speicherbereinigung durch **garbage collection**.
- Dadurch wird Speicherplatz von Objekten, die nicht mehr durch Referenzen zugänglich sind, freigegeben.

Verkettete Strukturen I

- **Zeiger** dienen dazu, zunächst unabhängige Datenobjekte miteinander zu verketten.

- Hierzu wird dem Objekt eine (oder mehrere) Objektreferenzen hinzugefügt.

```
class VObjekt {
  public Data object;
  public Object next;
}
```

```
class Data {
  public int key;
  public String info;
}
```

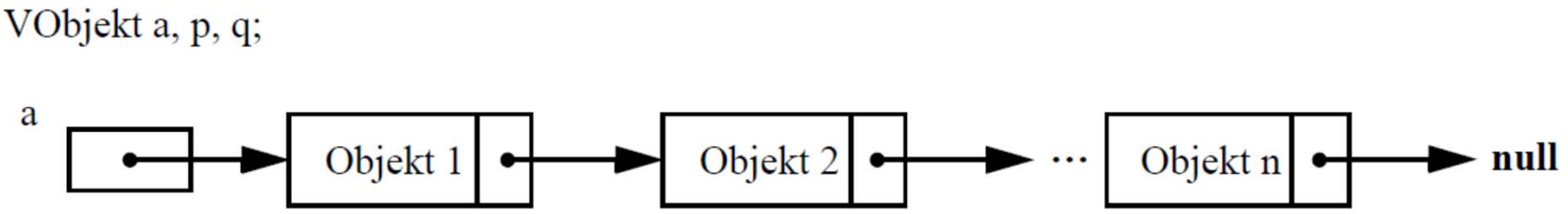
Verkettete Strukturen II

- Abhängig von der Art der Verkettung und den auf der Struktur vorgesehenen Operationen unterscheiden wir z.B. folgende dynamische Datenstrukturen:
  - Lineare Listen
  - Kreisförmig verkettete Listen
  - Doppelt verkettete Listen
  - Schlangen (Queues) / Stapel (Stacks)
  - Bäume
  - Graphen

Lineare Listen I

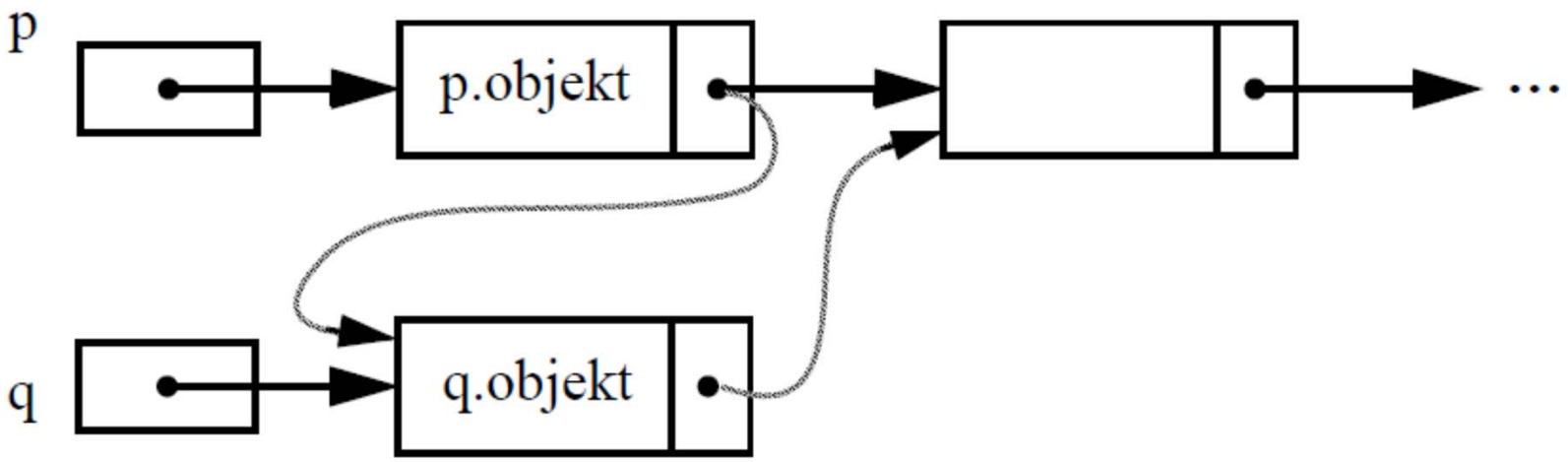
- **Definition:**
  Eine **Lineare Liste** ist eine verkettete Folge von Objekten eines Typs T, zusammen mit einem Zeiger a auf das erste Objekt. Dieser Zeiger heißt **Anker** der Liste.

- Lineare Listen sind **rekursive Datenstrukturen**, da jedes Listenelement wiederum einen Zeiger auf eine Liste enthält.
- Eine rekursive Implementierung von Listendurchläufen (s. später) ist jedoch aus Effizienzgründen ungeeignet.



Allgemeine Operationen auf lineare Listen I

Einfügen des Elements q hinter p:

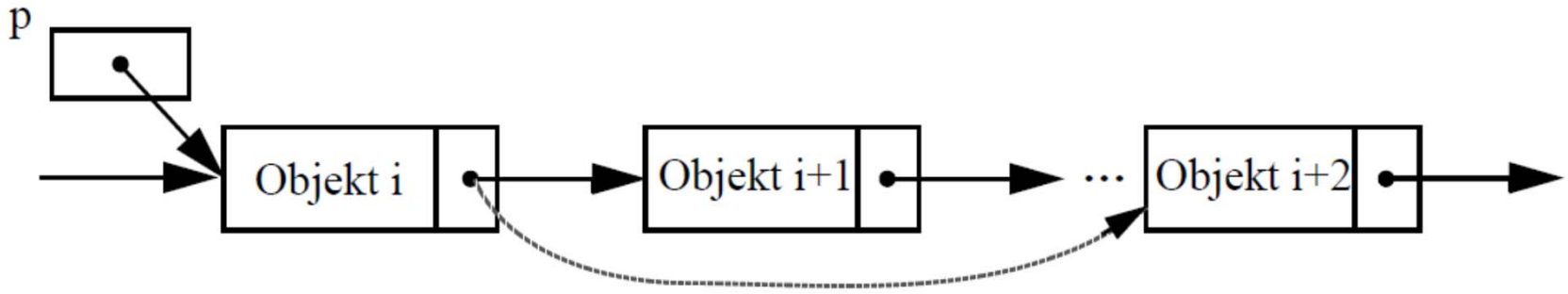


Programmcode:

```
q.next = p.next;
p.next = q;
```

Allgemeine Operationen auf lineare Listen II

Entfernen des Nachfolgers des Elements p:



Programmcode:

```
q.next = p.next.next;
```

Anmerkung:

Das Element mit Inhalt „Object i+1“ ist damit **nicht gelöscht**, sondern nur aus der Liste entfernt!

Allgemeine Operationen auf lineare Listen III

Durchlaufen linearer Listen:

- Lineare Listen werden stets „von Element zu Element“ durchlaufen, indem man den $next$-Zeigern folgt.

- **Allgemein:** Sei q ein Zeiger auf das erste Listenelement:

```
while (q != null)
{
   // führe Operation auf dem Element q aus
   q = q.next;
}
```

Allgemeine Operationen auf lineare Listen IV

Beispiel 1: Suche eines Elementes mit (Objekt == x)

```
while ((q != null) && (q.Objekt != x ))
    q = q.next;
```

q zeigt dann auf das gesuchte Element, falls es existiert,
oder `q == null`, falls das Element nicht existiert.

Beispiel 2: Zugriff auf das i-te Listenelement

Voraussetzung: Elementanzahl der Liste ist nicht kleiner als i

```
for (int j = 1; j < i; j++)
    q = q.next;
```

Allgemeine Operationen auf lineare Listen IV

Zeitaufwand

im schlechtesten Fall für eine Liste mit n Elementen

Operation	**Sequentiell gespeichert** (Array)	**Verkettet gespeichert** (Liste)
Suche Objekt in sortierter Liste	$O(\log n)$	$O(n)$
Suche i-tes Objekt	$O(1)$	$O(i)$
Einfügen an bekannter Stelle	$O(n)$	$O(1)$
Entfernen an bekannter Stelle	$O(n)$	$O(1)$

Spezielle Lineare Listen I

Kreisförmigverkettete Liste:

- Lineare Liste
- Zeiger des letzen Listenelements zeigt auf erstes Listenelement (statt auf „null“ wie bei gewöhnlichen linearen Listen)

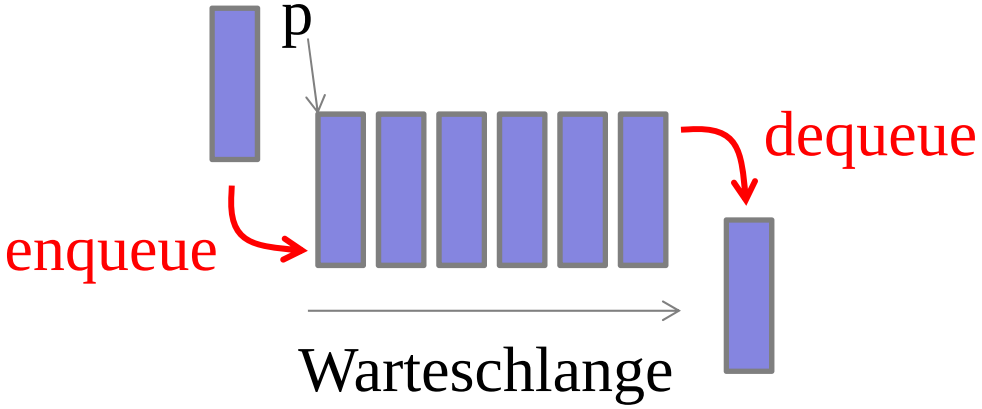
Doppeltverkette Liste:

- Lineare Liste
- Listenelemente haben **zwei Zeiger**:
  - Ein Zeiger auf **nachfolgenden Element** (falls vorhanden, ansonst. „null“)
  - Ein Zeiger auf **vorhergehendes Element** (falls vorhanden, ansonst. „null“)

Spezielle Lineare Listen II

Schlangen (Queues):

- Linearen Liste mit zwei Operationen: enqueue, dequeue
- **Enqueue:** Neues Element am Listenkopf einfügen.
- **Dequeue:** Entnahme des letzten Listenelement.

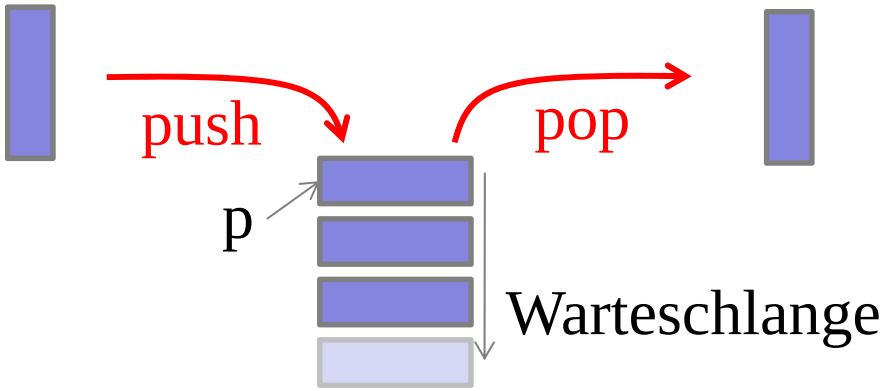


- Liste mit Fist-In-First-Out (FIFO)-Eigenschaft (FIFO-queue)
- Verwendung: Funktion eines Puffers (buffer)

Spezielle Lineare Listen II

Stapel (Stack):

- Linearen Liste mit zwei Operationen: push, pop
- **Push:** Neues Element am Listenkopf einfügen.
- **Pop:** Entnahme des **ersten** Listenelementes.

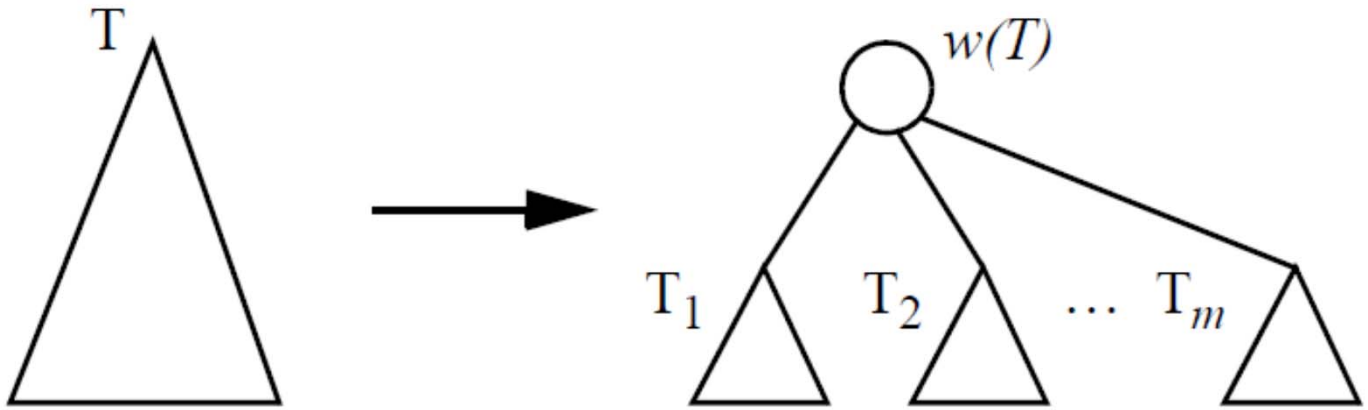


- Liste mit Last-In-First-Out (LIFO)-Eigenschaft (LIFO-queue)

Baumstrukturen

- **Definition:**
  Ein **Baum** ist eine endliche Menge T von Elementen (**Knoten**), mit:
  - Es gibt einen ausgezeichneten Knoten $w(T)$, die **Wurzel** von T.
  - Die restlichen Knoten sind in $m \geq 0$ disjunkte Mengen $T_1, \ldots, T_m$ zerlegt, die ihrerseits Bäume sind. $T_1, \ldots, T_n$ heißen **Teilbäume** der Wurzel $w(T)$.

  (*rekursive Definition*)



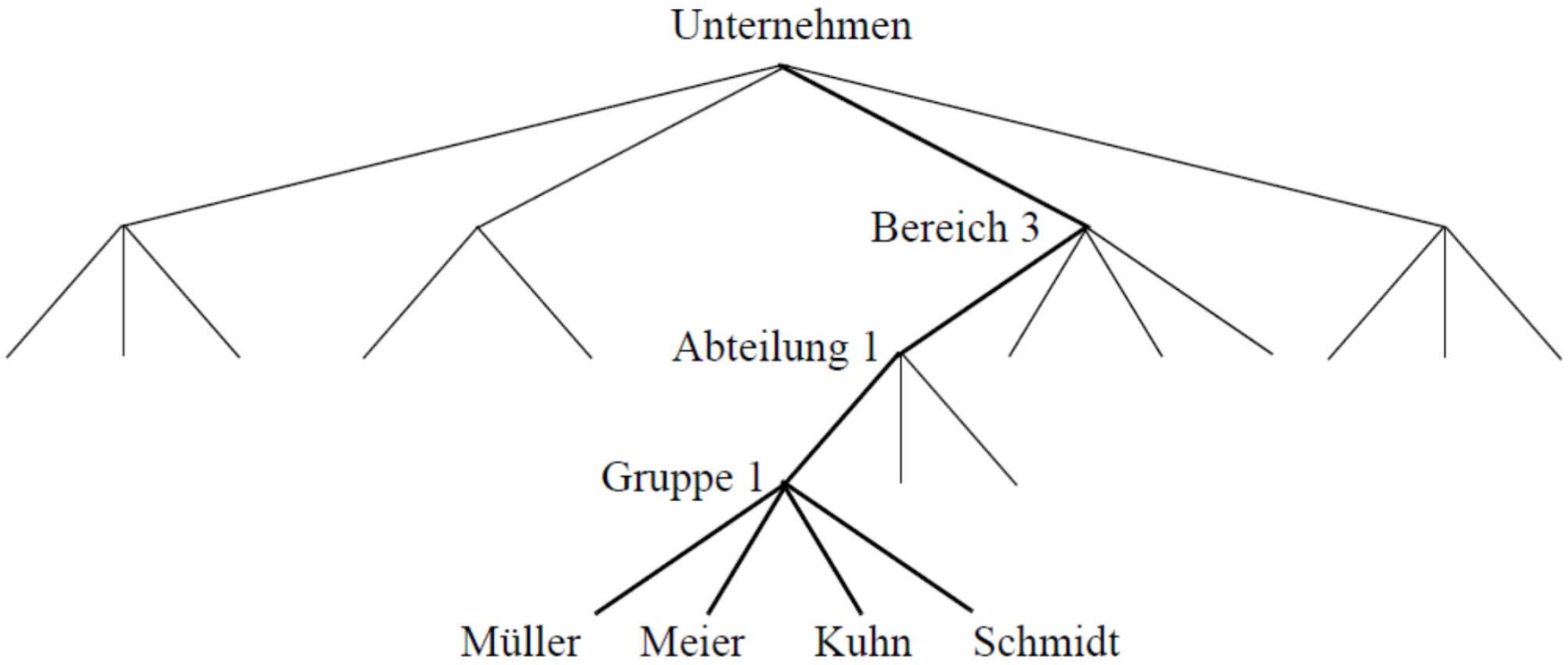
Baumstrukturen

- Eine **lineare Liste** ist ein (degenerierter) Baum, in dem jeder Knoten höchstens einen Teilbaum besitzt.

- Bäume erlauben es, **hierarchische Beziehungen zwischen Objekten** darzustellen.
- Beispiele:
  - Die Gliederung eines Buches in Kapitel, Abschnitte, Unterabschnitte
  - Die Aufteilung Deutschlands in Bundesländer, Bezirke, Kreise, Gemeinden

Baumstrukturen

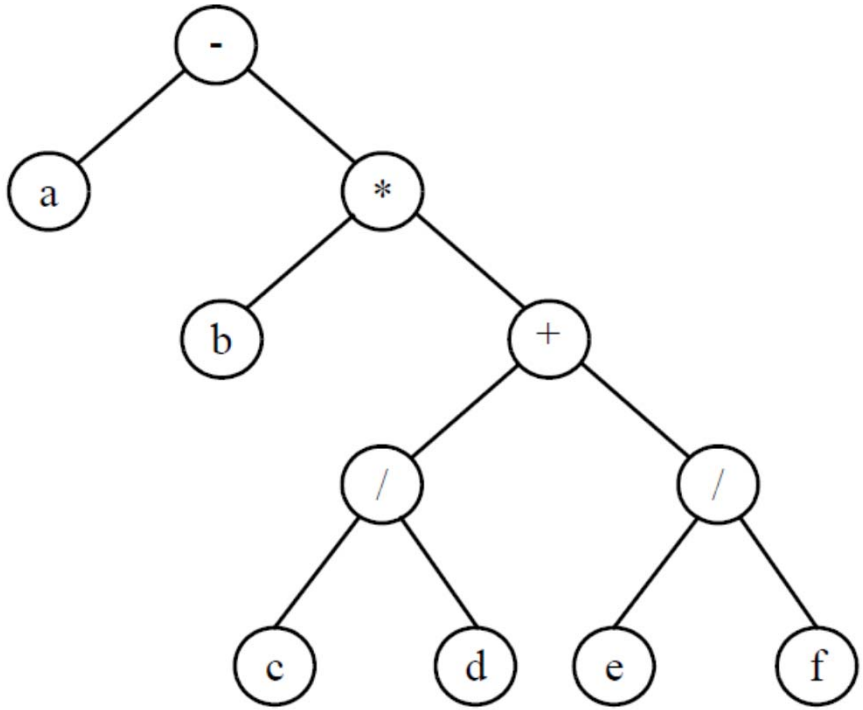
- weitere Beispiele:
  - Die Strukturierung eines Unternehmens in Bereiche, Abteilungen, Gruppen und Angestellte



Baumstrukturen

- weitere Beispiele:
  - Darstellung (geklammerter) arithmetischer Ausdrücke als **binäre Bäume** (d.h. $m \leq 2$ für alle Knoten des Baumes). Diese Bäume heißen **Operatorbäume**.

Operatorbaum für:
$a - b * (c \,/\, d + e \,/\, f)$



Begriffe zu Baumstrukturen I

- Der **Grad** eines Knoten x, $\mathbf{deg}(\boldsymbol{x})$, ist gleich der Anzahl der Teilbäume von x. Gilt $\deg(x) = 0$, so nennt man x ein **Blatt**.
- Der **Grad des Baumes** ist der maximale Grad aller seiner Knoten.

- Jeder Knoten $x \neq w(T)$ hat einen eindeutigen Vorgänger $v(x)$, auch **Vater** von x genannt.
- $s(x)$ bezeichnet die Menge aller **Söhne** (auch Kinder oder Nachfolger von x genannt)
- $b(x)$ bezeichnet die Menge aller **Brüder** von x (auch Geschwister von x genannt).
- Alle Brüder in $b(x)$ haben denselben Vater wie x.

Begriffe zu Baumstrukturen II

- Ein **Pfad** in einem Baum ist eine Folge von Knoten $p_1, \ldots, p_n$ mit: $p_i = v(p_{i+1}), i = 1, \ldots, n-1$.
- Die **Länge des Pfades** ist n.

- Der **Level eines Knotens** x:

$$lev(x) = \begin{cases} 1 & f\ddot{u}r\ x = w(T) \\ lev\big(v(x)\big) + 1 & f\ddot{u}r\ x \neq w(T) \end{cases}$$

- Damit ist $lev(x)$ gleich der Anzahl der Knoten auf dem Pfad von der Wurzel zum Knoten x.

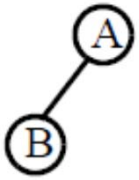
Begriffe zu Baumstrukturen III

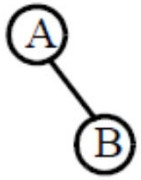
- Die **Höhe** eines Baumes ist gleich dem maximalen (d.h. maximal möglichen) Level seiner Knoten.
- Dies entspricht der Länge des längsten von der Wurzel ausgehenden Pfades innerhalb des Baumes.

- Ein **Wald** ist eine geordnete Menge von $n \geq 0$ disjunkten Bäumen.
- Entfernt man die Wurzel eines Baumes, so erhält man einen Wald.

Binäre Bäume

- **Definition:**
  Ein **binärer Baum** ist eine endliche Menge B von Knoten, die:
  - entweder leer ist
  - Oder aus einer Wurzel und zwei diskunkten Bäumen besteht, dem linken und dem rechten Teilbaum der Wurzel.

- Damit sind: 

und 

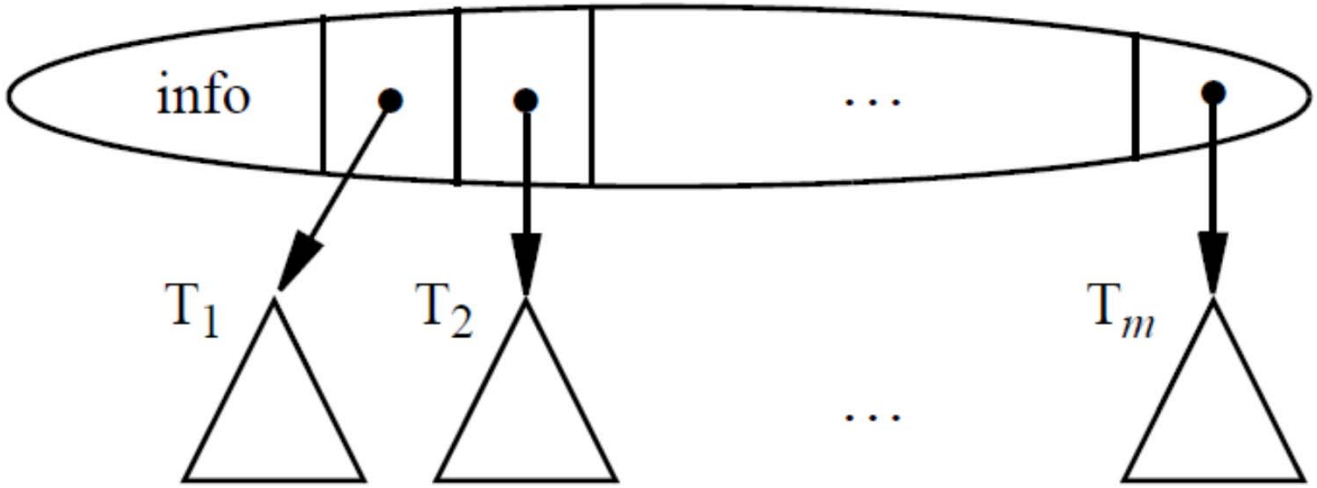
zwei verschiedene binäre Bäume.

- Für alle Knoten x eines binären Baumes gilt: $\deg(x) \leq 2$.
- Beispiel: Arithmetischer Ausdruck
- Dabei gilt:
  - Innere Knoten → Operatoren
  - Blätter → Operanden

Speicherdarstellung von Bäumen I

- **Erster allgemeiner Ansatz:**

```
class T {
   public Data info;   // Data ist eine beliebige Klasse
   public T1A1;
   …
   public TmAm;        // 0 ≤ m ≤ Grad des Baumes
}
```



Speicherdarstellung von Bäumen II

Nachteile dieser Darstellungsart:

- Verschiedene Knotentypen sind nötig, abhängig von dem jeweiligen Grad des Knotens
- Bei der Definition des Knotentyps müssen die Knotentypen der Söhne bekannt sein.

Besserer Ansatz:

- Einen Knotentyp mit fester Anzahl m (= Grad des Baumes) von Zeigerfeldern verwenden.
- Aber:
  Schlechtere Speicherplatzausnutzung durch unnötige Zeigerfelder, denn…

Speicherdarstellung von Bäumen III

Proposition: Sei T ein Baum vom Grad k mit n Knoten, dann gilt:

$n \cdot (k-1) + 1$ der $n \cdot k$ Zeigerfelder sind mit **null** besetzt.

Beweis: In den n Knoten gibt es genau $n-1$ Zeiger, die nicht **null** sind. Damit ist die Anzahl der **null**-Felder gleich

$$n \cdot k - (n-1) = n \cdot (k-1) + 1$$

Folgerung: Die Speicherplatzverschwendung sp der Zeigerfelder ist:

$$sp = \frac{count(null\text{–}Felder)}{count(Zeigerfelder)} = \frac{n \cdot (k-1) + 1}{n \cdot k} \approx \frac{k-1}{k}$$

Schon bei binären Bäumen ($k = 2$) liegt sp bei 50% und hat damit den minimalen Wert für alle möglichen k.