Ludwig-Maximilians-Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Skript zur Vorlesung

Algorithmen und Datenstrukturen

im Sommersemester 2010

Prof. Dr. Hans-Peter Kriegel

Skript unter Mitarbeit von Prof. Dr. Martin Ester, Prof. Dr. Daniel A. Keim, Dr. Michael Schiwietz und Prof. Dr. Thomas Seidl

Kapitel 1 Einführung

1.1 Algorithmen und ihre Analyse

1.1.1 Der Begriff des Algorithmus

Unter einem Algorithmus versteht man eine **präzise**, **endliche** Verarbeitungsvorschrift, die so formuliert ist, daß die in der Vorschrift notierten **Elementaroperationen** von einer mechanisch oder elektronisch arbeitenden Maschine durchgeführt werden können. Aus der Präzision der sprachlichen Darstellung des Algorithmus muß die **Abfolge** der einzelnen Verarbeitungsschritte **eindeutig** hervorgehen. Hierbei sind **Wahlmöglichkeiten** zugelassen, deren Auswahl jedoch genau festliegen muß. Die Ausführung eines Algorithmus (Prozeß) geschieht durch ein **Ausführungsorgan** (Prozessor), indem die spezifizierten Elementaroperationen in der festgelegten Reihenfolge abgearbeitet werden.

Beispiele für "Alltagsalgorithmen"

Prozeß	Algorithmus	Typische Schritte im Algorithmus
Pullover stricken	Strickmuster	Stricke Rechtsmasche, stricke Linksmasche
Modellflugzeug bauen	Montageanleitung	Leime Teil A an den Flügel B
Kuchen backen	Rezept	Nimm 3 Eier; schaumig schlagen
Kleider nähen	Schnittmuster	Nähe seitlichen Saum
Beethovensonate spielen	Notenblatt	

Solche alltäglichen Vorschriften ähneln Algorithmen; sie sind aber nur selten exakt ausformuliert und enthalten oft Teile, die vom Ausführenden mehrdeutig interpretiert werden können. In der Vorlesung behandeln wir natürlich Computer-Algorithmen, und zur Notation verwenden wir die Programmiersprache Java.

Durch einen Algorithmus werden mittels einer Reihe von Anweisungen Objekte manipuliert, um von einer spezifischen Eingabe eine spezifische Ausgabe zu erhalten. Ein Algorithmus beschreibt somit eine Abbildung $f : E \rightarrow A$ von der Menge der zulässigen Eingabedaten E in die Menge der Ausgabedaten A.

aber: Nicht jede Abbildung $f : E \rightarrow A$ läßt sich durch einen Algorithmus realisieren (Berechenbarkeit!).

1.1.2 Typische Eigenschaften von Algorithmen

Algorithmen besitzen charakteristische Eigenschaften:

- **Abstrahierung**
  Ein Algorithmus löst im allgemeinen eine Klasse von Problemen. Die Wahl eines konkreten, aktuell zu lösenden Problems aus dieser Klasse erfolgt über geeignete Wahl von Parametern.
- **Finitheit**
  Die Beschreibung eines Algorithmus selbst besitzt eine endliche Länge (statische Finitheit). Ferner darf ein Algorithmus zu jedem Zeitpunkt nur endlich viel Platz belegen (dynamische Finitheit), d.h. die bei der Abarbeitung des Algorithmus entstehenden Datenstrukturen und Zwischenergebnisse sind somit endlich.
- **Terminierung**
  Algorithmen, die für jede Eingabe nach endlich vielen Schritten ein Resultat liefern und anhalten, heißen terminierend, sonst nicht-terminierend. In der Praxis interessieren häufig nur terminierende Algorithmen, aber in speziellen Bereichen, wie z.B. bei Betriebssystemen (Basis-Algorithmen eines Rechners) oder bei der Überwachung von Anlagen, Produktionsstätten bzw. Verkehrsampeln oder beim Befehlsholezyklus der CPU (Lade nächsten Befehl; Führe Befehl aus), sind auch nicht-terminierende Algorithmen von Bedeutung.
- **Determinismus**
  Ein Algorithmus heißt deterministisch, wenn zu jedem Zeitpunkt seiner Ausführung höchstens eine Möglichkeit der Fortsetzung besteht. Hat ein Algorithmus an mindestens einer Stelle zwei oder mehr Möglichkeiten der Fortsetzung, von denen eine ausgewählt werden kann, so heißt er nichtdeterministisch. Kann man den Fortsetzungsmöglichkeiten Wahrscheinlichkeiten zuordnen, so spricht man von stochastischen Algorithmen.
- **Determiniertheit**
  Algorithmen sind im allgemeinen determiniert, d.h. wird ein Algorithmus mit den gleichen Eingabewerten und Startbedingungen wiederholt, so liefert er stets das gleiche Ergebnis. Eine Erweiterung bilden die nicht-determinierten Algorithmen, die bei gleichen Startbedingungen unterschiedliche Ergebnisse berechnen können. Diese Eigenschaft nimmt man manchmal in Kauf, z.B. wenn ein exakter Lösungsalgorithmus eine hohe Komplexität hat und man darum heuristische Methoden anwendet, also auf die erschöpfende Bearbeitung aller Fälle (und damit oft auf die absolut beste Lösung) verzichtet. Ein terminierender, deterministischer Algorithmus ist immer determiniert. Ein terminierender, nichtdeterministischer Algorithmus kann determiniert oder nicht-determiniert sein.

Eine faszinierende und zentrale Aufgabe der Informatik ist es, Probleme durch Algorithmen zu lösen. Ist ein solcher Lösungsalgorithmus entdeckt und formuliert, so ist das gestellte Problem aus Sicht der Theorie “erledigt”, da jetzt scheinbar kein Problem mehr vorliegt. Wenngleich durch Algorithmen große Problembereiche als “gelöst” erscheinen, so suchen Informatiker oft weiter nach einem **möglichst guten** Algorithmus, d.h. die **Effizienz** von Algorithmen spielt eine große Rolle:

Man kann beweisen, daß es zu jedem Algorithmus unendlich viele verschiedene, äquivalente Algorithmen gibt, die die gleiche Aufgabe lösen. Die Suche nach schnelleren oder kompakteren Algorithmen oder der Beweis, daß es solche nicht geben kann, sind interessante Fragestellungen, mit denen sich die Informatik beschäftigt.

1.1.3 Grundlegende Bestandteile von Algorithmen

Zur Ausführung eines Algorithmus benötigt man verschiedene Arten von Kontrollstrukturen:

Sequenz (Folge von Anweisungen)

- Zu einem Zeitpunkt wird nur ein Schritt ausgeführt.
- Jeder Schritt wird genau einmal ausgeführt: keiner wird wiederholt, keiner wird ausgelassen.
- Die Schritte werden in der Reihenfolge ausgeführt, in der sie niedergeschrieben sind (d.h. nacheinander).
- Mit der Beendigung des letzten Schrittes endet der gesamte Algorithmus.

Selektion (Auswahl, bedingte Anweisung)

Diese Kontrollstruktur erlaubt die bedingte Ausführung von Anweisungen

- Einfache Form:

  Falls Bedingung
  dann Sequenz

- Bedingte Anweisung mit Alternative (allgemeine Form):

  Falls Bedingung
  dann Sequenz 1
  sonst Sequenz 2

Anmerkung: Die einfache Form ist ein Spezialfall der allgemeinen Form, bei der Anweisung 2 die leere Anweisung ist (“tue nichts”).

- Mehrfachauswahl:

  Falls
  Bedingung 1 **dann** Sequenz 1
  Bedingung 2 **dann** Sequenz 2
  …
  Bedingung n **dann** Sequenz n
  andernfalls
  Sequenz n + 1

Die Bedingungen 1 bis n müssen sich gegenseitig ausschließen; d.h. es dürfen nicht zwei Bedingungen gleichzeitig erfüllt sein.

Iteration (Wiederholung, Schleife)

- Wiederholte Ausführung einer Anweisung (oder einer Folge von Anweisungen), bis eine Abbruchbedingung erfüllt ist.

  Wiederhole
  Anweisung(en)
  bis Bedingung {Abbruchbedingung}

  Spezialfall: Endlosschleife

  Wiederhole
  Anweisung(en)
  immer

- Zweite Form der Iteration: Ausführung von Anweisungen, solange eine Bedingung erfüllt ist.

  Solange Bedingung **führe aus**
  Anweisung(en) {Rumpf der Schleife}

Unterschied zur ersten Form:
Die Bedingung wird vor der Ausführung des Schleifenrumpfes geprüft. Deshalb ist diese Form vorzuziehen, wenn damit gerechnet werden muß, daß in manchen Fällen bereits beim erstmaligen Eintritt in die Schleife die Abbruchbedingung erfüllt ist, der Schleifenrumpf also nicht ausgeführt werden soll.

Die beiden Formen der Iteration sind äquivalent, sie lassen sich (unter Verwendung der bedingten Anweisung) ineinander überführen.

Anmerkungen

1. Unbeabsichtigte Endlosschleifen entstehen häufig, weil die Abbruchbedingung nicht korrekt formuliert wurde.
2. Sequenz, Selektion und Iteration genügen, um **jeden Algorithmus** auszudrücken!

1.1.4 Laufzeitanalyse von Algorithmen

Die **Effizienz** ist ein wichtiges Kriterium zum Vergleich verschiedener Algorithmen zur Lösung ein und desselben Problems. Sie wird bestimmt durch den benötigten Aufwand des Algorithmus (seine **Komplexität**) in Abhängigkeit von einer speziellen Eingabesituation.

Wesentliche Effizienzkriterien sind
- die Laufzeit des Algorithmus
- der benötigte Speicherplatz

Häufig: ‘trade-off’ bei der Optimierung eines dieser beiden Kriterien dahingehend, daß das andere Kriterium verschlechtert wird.

In der Regel ist das wichtigere Kriterium die Laufzeit. Wir werden uns daher im folgenden auf die **Laufzeitanalyse** beschränken.

Laufzeitanalyse

1. Ansatz: Direktes **Messen der Laufzeit**, z.B. in Millisekunden.

➞ abhängig von vielen Parametern, wie Rechnerkonfiguration, Rechnerlast, Compiler, Betriebssystem, Programmiertricks, u.a., und damit kaum übertragbar und ungenau.

2. Ansatz: Zählen der benötigten **Elementaroperationen** des Algorithmus in Abhängigkeit von der Größe der Eingabe.

➞ das algorithmische Verhalten wird als Funktion der benötigten Elementaroperationen dargestellt. Die Charakterisierung dieser elementaren Operationen ist abhängig von der jeweiligen Problemstellung und dem zugrundeliegenden Algorithmus. Beispiele für Elementaroperationen sind Zuweisungen, Vergleiche, arithmetische Operationen, Zeigerdereferenzierungen oder Arrayzugriffe.

➞ das Maß für die Größe der Eingabe ist abhängig von der Problemstellung, z.B.

Problem	Größe der Eingabe
Suche eines Elementes in einer Liste	Anzahl der Elemente
Multiplikation zweier Matrizen	Dimension der Matrizen
Sortierung einer Liste von Zahlen	Anzahl der Zahlen

Beispiel: Sequentielle Suche eines Elementes in einem Array.

13	7	5	23	8	18	17	31	3	11	9	30	24	27	21	19

Funktion:

```
int Seqsearch(int[] A, int x)
{
 int i = 0, high = A.length - 1;
      while ((i <= high) && (A[i] != x))
          i++;
      if  (i <= high) return i;
      else return -1;
}
```

→ wesentliche Operationen, auch **Grundoperationen** genannt: Ausführungen der **while**-Schleife

Diese Anzahl ist **abhängig** von:

- der **Größe** des Arrays (n; fest vorgegeben)
- der **Position** des gesuchten Elementes innerhalb des Arrays (variabel)

Wir unterscheiden daher:

- den **durchschnittlichen Zeitbedarf** $T_{\varnothing}(n)$ eines Algorithmus, charakterisiert durch die durchschnittliche Anzahl $A_{\varnothing}(n)$ benötigter Grundoperationen für alle Eingaben der Größe n, und
- den **Zeitbedarf im schlechtesten Fall** $T_{worst}(n)$, charakterisiert durch $A_{worst}(n)$, die Anzahl benötigter Grundoperationen im schlechtesten Fall aller Eingaben der Größe n.

Sei E_n die Menge aller möglichen Eingaben der Größe n und $a(e)$, $e \in E_n$, die Anzahl von Grundoperationen, die ein gegebener Algorithmus bei Eingabe von e ausführt. Sei weiterhin $p(e)$ die Wahrscheinlichkeit, mit der die Eingabe e auftritt ($\sum_{e \in E_n} p(e) = 1$).

Dann gilt:

- $A_{\varnothing}(n) = \sum_{e \in E_n} p(e) \cdot a(e)$
- $A_{worst}(n) = \max_{e \in E_n}(a(e))$

Im obigen Beispiel:

Sei q die Wahrscheinlichkeit, daß x im Array vorhanden ist und sei jede Position für x gleichwahrscheinlich. Bezeichne weiterhin e_i, $0 \le i \le n-1$, die Menge aller Eingaben mit $x = A[i]$ und entsprechend e_n die Menge aller Eingaben, die x nicht enthalten.

Dann gilt: $p(e_i) = q/n$, $0 \le i \le n-1$ und $p(e_n) = 1 - q$.

Hieraus ergibt sich:

$$\text{Im Falle } q = 1: \quad A_{\varnothing}(n) = (n+1)/2.$$

$$\text{Im Falle } q = \frac{1}{2}: \quad A_{\varnothing}(n) = \frac{(n+1)}{4} + \frac{n}{2} \approx \frac{3}{4} \cdot n.$$

$$A_{worst}(n) = \max_{0 \le i \le n}(a(e_i)) = n.$$

Durch Weglassen multiplikativer und additiver Konstanten wird allein das Wachstum der Laufzeitfunktion $T(n)$ betrachtet. Man erhält eine von der Programmumgebung und anderen äußeren Einflußgrößen (z.B.: dem *Zeitbedarf einer Vergleichsoperation, der Effizienz der Programmierung,* ...) unabhängige Charakterisierung der (asymptotischen) Komplexität des Algorithmus.

→ O-Notation verwenden

Definition: *O-Notation*

$$A_{\varnothing}(n) = \sum_{e \in E_n} p(e) \cdot a(e) = \sum_{i=0}^{n} p(e_i) \cdot a(e_i) = \sum_{i=0}^{n-1} \frac{q}{n} \cdot (i+1) + (1-q) \cdot n =$$

$$= \frac{q}{n} \cdot \sum_{i=1}^{n} i + (1-q) \cdot n = \frac{q}{n} \cdot \frac{n \cdot (n+1)}{2} + (1-q) \cdot n =$$

$$= q \cdot \frac{n+1}{2} + (1-q) \cdot n$$

Seien $f:(N \to \Re^+)$ und $g:(N \to \Re^+)$.

$f \in O(g) \Leftrightarrow \exists\ n_0 \in N,\ c \in \Re^+$ mit $\forall\ n \geq n_0$ ist $f(n) \leq c \cdot g(n)$.

Man sagt auch: f wächst höchstens so schnell wie g.

Gebräuchlichere Schreibweise: $f = O(g)$ für $f \in O(g)$

Anmerkung: Da $O(g)$ genau genommen eine (unendliche) Menge von Funktionen beschreibt, ist es zwar genauer, zu sagen: $f \in O(g)$. In der Literatur hat sich jedoch das '=' eingebürgert. Wichtig ist es, darauf zu achten, daß insbesondere die Symmetrie der '='-Relation hier **nicht** gilt.

Mit dieser Notation gilt: $T_{\varnothing}(n) = O(A_{\varnothing}(n))$ sowie $T_{worst}(n) = O(A_{worst}(n))$.

Im Beispiel ist also:

$$T_{\varnothing}(n) = O(q \cdot \frac{n+1}{2} + (1-q) \cdot n) = O(n) \quad \text{und} \quad T_{worst}(n) = O(n).$$

Funktionen, die bei der Laufzeitanalyse von Algorithmen realistischerweise auftauchen, sind meist monoton wachsend und von 0 verschieden. Zur Überprüfung obiger Eigenschaft betrachtet man dabei den Quotienten $f(n)/g(n)$.

Nach Definition gilt für $f = O(g)$: $f(n)/g(n) \leq c$ für $n \geq n_0$.

Man betrachte den Grenzwert: $\lim_{n \to \infty} \frac{f(n)}{g(n)}$. Existiert dieser (d.h. er ist $< \infty$), so ist $f = O(g)$.

Rechnen mit der O-Notation:

- Elimination von Konstanten: $2 \cdot n = O(n)$, $n/2 + 1 = O(n)$
- Bilden oberer Schranken: $2 \cdot n = O(n^2)$, $3 = O(\log n)$

Wichtige Klassen von Funktionen:

	Sprechweise	Typische Algorithmen
$O(1)$	konstant	
$O(\log n)$	logarithmisch	Suchen auf einer Menge
$O(n)$	linear	Bearbeiten jedes Elementes einer Menge
$O(n \cdot \log n)$		Gute Sortierverfahren, z.B. Heapsort
$O(n \cdot \log^2 n)$		
...		
$O(n^2)$	quadratisch	primitive Sortierverfahren
$O(n^k), k \geq 2$	polynomiell	
...		
$O(2^n)$	exponentiell	Backtracking-Algorithmen

Optimalität von Algorithmen

Man sagt: ein Algorithmus *A* ist eine **(worst-case)-optimale Lösung** eines gegebenen Problems, falls für jeden Lösungsalgorithmus *B* aus der Algorithmenklasse von *A* gilt:

$$T^A_{worst} = O(T^B_{worst}).$$

Zurück zu unserem Beispiel der Suche in einem Array:

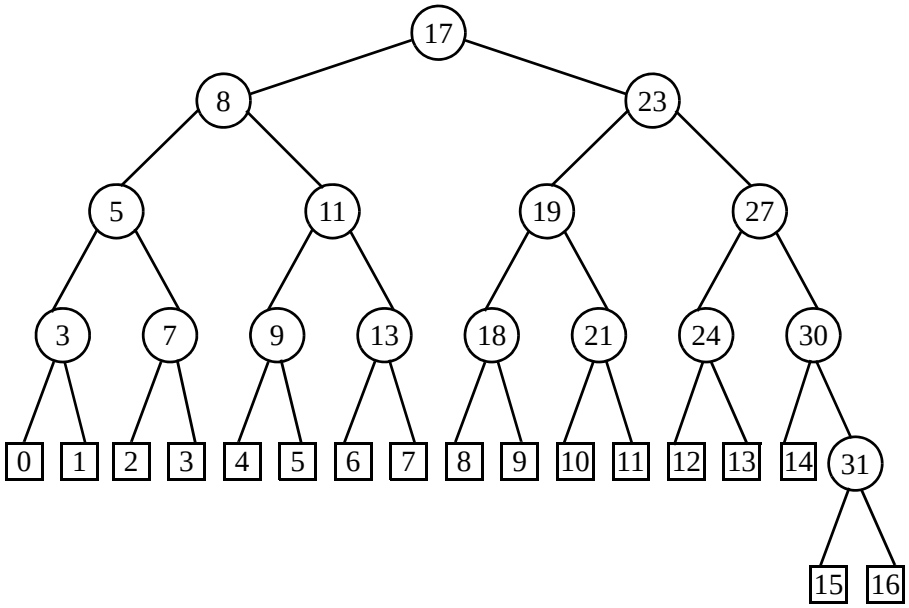
Die Array-Komponenten seien nun in aufsteigender Reihenfolge sortiert.

3	5	7	8	9	11	13	17	18	19	21	23	24	27	30	31

➙ Anwendung der **binären Suche**

Durch Vergleich des Suchschlüssels mit dem mittleren Element des noch relevanten Teilarrays kann jeweils die Hälfte der Elemente von einer weiteren Betrachtung ausgeschlossen werden.

Entscheidungsbaum zum obigen Array:



Funktion:

```
int Binsearch (int[] A, int x)
{
 int m= 0, u = 0, o = A.length - 1;
 boolean gefunden = false;
 while ((u <= o) && (!gefunden))
     {
      m = (u + o) / 2;
      if  (x == A[m]) gefunden = true; // x gefunden an der Stelle m
      else if (x < A[m]) o = m - 1;  else  u = m + 1;
     } // end while
 if (!gefunden) return -1;
 else return m;
}
```

Analyse des binären Suchalgorithmus:

Der Algorithmus arbeitet durch sukzessive Halbierung relevanter Teilarrays. (**while** ...)
In jedem Halbierungsschritt wird:

- das mittlere Element mit dem Suchschlüssel verglichen (x = A[m]) und
- die Suche erfolgreich abgebrochen bzw. das neue Teilarray festgelegt (**else if** ...).

Betrachten wir die Vergleiche mit dem jeweils mittleren Element (x = A[m]) als Grundoperation, so ergibt sich für $A_{worst}(k)$ folgende Rekursionsgleichung:

$A_{worst}(k) = 1 + A_{worst}(\lfloor k/2 \rfloor)$ (für $k > 1$) mit dem Anfangswert $A_{worst}(1) = 1$.

Damit erhalten wir:

$$A_{worst}(n) = 1 + A_{worst}(\lfloor n/2 \rfloor) = 2 + A_{worst}(\lfloor n/4 \rfloor) = \ldots = \lfloor \log_2(n) \rfloor + 1 .$$

Damit: $T_{worst}(n) = O(\log_2(n))$.

Man kann weiterhin zeigen, daß bei Gleichverteilung gilt:

$$A_{\varnothing}(n) \approx \lfloor \log_2(n) \rfloor + 1/2 .$$

Damit: $T_{\varnothing}(n) = O(\log_2(n))$.

Anmerkung: Der binäre Suchalgorithmus ist **optimal**, d.h. $O(\log n)$ ist eine untere Schranke für den Aufwand zur Suche eines bestimmten Elementes in einer Liste von n Elementen.

Begründung: Der binäre Entscheidungsbaum über einem sortierten n-elementigen Array hat eine Höhe von $O(\log n)$. D.h. für jeden korrekten Suchalgorithmus sind im schlechtesten Fall immer $O(\log n)$ Vergleiche notwendig.

Realistisches Beispiel:

Suche in Telefonbüchern unter der Annahme, das Array liegt komplett im Hauptspeicher und eine (Vergleichs-)Operation kostet 0,1 msec.

Ort	Anzahl Einträge	sequentielle Suche		binäre Suche	
		$A_{worst}(n)$	$T_{worst}(n)$	$A_{worst}(n)$	$T_{worst}(n)$
Gündlkofen	500	500	50 msec.	9	0,9 msec.
Würzburg	150.000	150.000	15 sec.	18	1,8 msec.
Tokio	10.000.000	10.000.000	> 16 min.	24	2,4 msec.

Für weiterführende Analysen der Komplexität von Algorithmen werden allgemeinere Formen der O-Notation verwendet:

Definition: *allgemeine O-Notation*

(i) $f = \Omega(g)$, falls $g = O(f)$; f wächst mindestens so schnell wie g.

(ii) $f = \Theta(g)$, falls $f = O(g)$ und $g = O(f)$; f und g wachsen mit gleicher Ordnung.

(iii) $f = o(g)$, falls $(f(n)/g(n))_{n \in N}$ eine Nullfolge; f wächst langsamer als g.

(iv) $f = \omega(g)$, falls $g = o(f)$; f wächst schneller als g.

Einige grundlegende Beziehungen sind:

- Seien p und p' **Polynome** vom Grad d bzw. d':

$$p(n) = c_d \cdot n^d + c_{d-1} \cdot n^{d-1} + \ldots + c_1 \cdot n + c_0$$

$$p'(n) = c'_{d'} \cdot n^{d'} + c'_{d'-1} \cdot n^{d'-1} + \ldots + c'_1 \cdot n + c'_0$$

  mit $c_d > 0$ und $c'_{d'} > 0$. Dann gilt:
  - $p = \Theta(p') \Leftrightarrow d = d'$
  - $p = o(p') \Leftrightarrow d < d'$
  - $p = \omega(p') \Leftrightarrow d > d'$
- Es gilt: $\log^k(n) = o(n^\varepsilon) \quad \forall\ k > 0$ und $\forall\ \varepsilon > 0$.
- Es gilt: $n^k = o(2^n) \quad \forall\ k > 0$.
- Es gilt: $2^{n/2} = o(2^n)$.

Anmerkung: Die Ω-Notation wird häufig als untere Schranke für die Laufzeit **aller** Algorithmen zur Lösung eines Problems angegeben. Sie charakterisiert die **Komplexität des Problems**.

Beispiel: Das Problem, aus einer ungeordneten Zahlenfolge das Minimum zu bestimmen, hat Komplexität $\Omega(n)$.

Ein Algorithmus heißt **asymptotisch optimal**, wenn seine worst-case Laufzeit mit der unteren Schranke der Komplexität des Problems zusammenfällt.

Anmerkung: Der Vergleich von Algorithmen auf Basis der O-Notation vernachlässigt multiplikative und additive Konstanten. Ein direkter Vergleich der Laufzeit für bestimmte Eingaben ist damit nicht möglich.

Beispiel: Für einen bestimmten Compiler und einen bestimmten Rechner gilt:

Algorithmus 1 : $1000 \cdot n^2$ Laufzeit
Algorithmus 2 : $5 \cdot n^3$ Laufzeit

→ Algorithmus 1 ist erst schneller ab $n = 200$!!

Anmerkung: Für die Bearbeitung großer Eingabemengen eignen sich praktisch nur Algorithmen mit einer Komplexität von $O(n)$ oder $O(n \cdot \log n)$. Exponentielle Algorithmen ($O(2^n)$) können dagegen nur auf sehr kleine Eingaben angewendet werden.

→ damit entscheidet die Komplexität des Algorithmus darüber, welche Eingabegrößen noch in praktikabler Laufzeit verarbeitet werden können.

Vergleich verschiedener Algorithmen

Komplexität des Algor. in Anzahl Operationen	Max. n für 10^3 Zeiteinheiten	Max. n für 10^5 Zeiteinheiten	Steigerungs-faktor	Zeit für $n = 250$ (1 Operat. = 0,01 ms)
$1000 \cdot n$	1	100	100	2,5 sec.
$50 \cdot n \cdot \log n$	9	342	38	0,69 sec.
$5 \cdot n^2$	14	141	10	3,1 sec.
$0{,}5 \cdot n^3$	12	27	2,3	78 sec.
2^n	9	16	1,7	$6 \cdot 10^{62}$ Jahre

(In Spalte 2 und Spalte 3 wird dabei angenommen, daß 1 Operation 1 Zeiteinheit benötigt.)

1.2 Datenstrukturen

Man unterscheidet statische und dynamische Datenstrukturen. **Statische** Datenstrukturen besitzen ein festes Speicherschema, das sich während der Programmlaufzeit nicht ändert. Für viele Probleme sind jedoch Datenstrukturen notwendig, deren Struktur während der Ausführung des Programms veränderbar ist. Derartige Strukturen werden als **dynamisch** bezeichnet. Ihre Organisation beruht auf der Verwendung von Zeigern.

1.2.1 Statische Datenstrukturen

Ausgehend von **atomaren** (**unstrukturierten**) 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

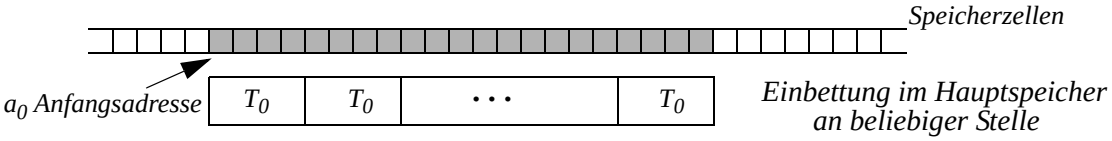
- Jede Komponente ist von atomarem oder strukturiertem Typ.
- Komponenten der “untersten” Strukturebene sind stets von atomarem Typ.
- Zugriff und Adressierung strukturierter Datentypen erfolgen über die einzelnen Komponenten.

Felder (ARRAYs; Reihungen)

Arrays dienen zur **indizierten** Aufnahme **einer Reihe von Informationen gleichen Typs**. Die Array-Notation in Java ist:

T_0 name **[]** = **new** T_0 [limit]; (T_0 heißt Grundtyp bzw. Komponententyp)

- Das Array entspricht damit dem **homogenen Produkt** $T_0 \times T_0 \times \ldots \times T_0$.
- Der Grundtyp ist beliebig (auch ein Array ist möglich; siehe 'mehrdimensionale Arrays'). Der Indextyp ist auf einen Teilbereich von **int** beschränkt: [0..limit-1]
- **Kardinalität** eines Arrays: $card(T) = card(T_0)^{limit}$.
- Speicherung durch Aneinanderreihung der Komponenten des Grundtyps



  Die **Adresse der i-ten Komponente** ist: $a_0 + (i-1) \cdot SIZE(T_0)$, wobei $SIZE(T_0)$ die Anzahl der von T_0 benötigten Speicherzellen angibt. Diese Funktion nennt man auch eine **Adreßfunktion**.

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

- Als Operation zwischen ganzen Arrays ist allein die **Wertzuweisung** '=' (bei exakter Typgleichheit) erlaubt. Die Wertzuweisung kopiert die Referenz auf das Array (d.h. seine Anfangsadresse a_0), nicht seinen Inhalt. Nach der Wertzuweisung existieren also zwei Referenzen auf denselben Speicherplatz. Vergleiche wie '==' oder '>' sind nicht möglich.

Beispiele:

float FloatArray [] = **new float** [50]; // *Array von **float**-Werten*

Date Geburtstage [] = **new** Date [23]; // *Array der Klasse Date*

String Waehrungen [] = {"USDollar", "Euro", "Schilling"}; // *Explizite Initialisierung*

float FloatArray2 [] = FloatArray; // *Wertzuweisung*

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 sind erforderlich

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

- **call-by-value** (Übergabe einer Kopie des Werts), wenn der Parametertyp eine 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:

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

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

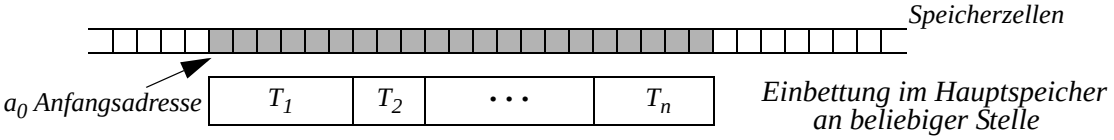
Verbunde (RECORDs, Klassen)

Records bzw. Klassen dienen zur Zusammenfassung **inhomogener**, aber **zusammengehöriger Informationen**. Wichtigste Anwendung ist die Repräsentation komplexer, strukturierter Informationen beliebiger Objekte. Häufig werden Records als Baustein größerer Datenstrukturen (z.B. von Arrays) verwendet.

Der Aufbau einer Klasse T basiert auf innerhalb der Struktur eindeutigen **Selektoren** S_i:

```
class T {
        T1 S1;
        T2 S2;
        ...
        Tn Sn;
}
```

- Ein Record entspricht dem **heterogenen Produkt**: $T_1 \times T_2 \times \ldots \times T_n$.
- Die Grundtypen $T_1, T_2, \ldots, T_n$ sind beliebig; mehrere Selektoren gleichen Typs können auch, durch ‘,’ getrennt, zusammengefaßt werden.
- **Kardinalität**: $card(T) = \prod_{i=1}^{n} card(T_i)$
- Speicherung durch Aneinanderreihen der Repräsentationen der Grundtypen



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

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

Die Speicheradresse der Komponente S_i ist damit $a_0 + offset(S_i)$.

Da die Größe von Records fest ist, ist ein Zugriff auf beliebige Komponenten in konstanter Zeit möglich.

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

Beispiel:

```
class   Datum { // Record Datum
                    public int Jahr;
                    public int Monat;
                    public int Tag;
        }
class   Person { // Record Person
                    public String Nachname, Vorname;
                    public Datum Geburt, Kunde_seit;
        }
```

```
Person Kundenliste[] = new Person[300]; // Ein Array von Personen
int EintrittsAlter = Kundenliste[272].Kunde_seit.Jahr - Kundenliste[272].Geburt.Jahr;
```

1.2.2 Dynamische Datenstrukturen

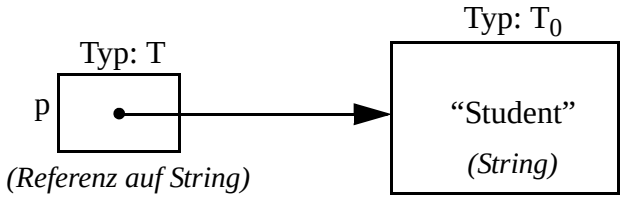
Zeiger ("**pointer**") unterstützen die Konstruktion **dynamischer Datenstrukturen**; diese heißen dynamisch, da sie während des Programmlaufes beliebig wachsen oder schrumpfen und ihre Struktur in gewissen Grenzen verändern können.

Zeigertypen

Zeiger: Verweis 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 auf die entsprechende Klasse bzw. Array.

```
T0 Objektreferenz;      // wobei T0 der Klassenname ist und die Objektreferenz
                        // implizit mit T (= ein Zeiger auf T0) verknüpft wird.
```

- T_0 ist der **Grundtyp** von T
- **Wertebereich** von T: Menge aller Zeiger auf Objekte vom Grundtyp T_0, die im bisherigen Verlauf des Programms erzeugt wurden. Im Unterschied zu Assember-Adressen sind Zeiger also typgebunden.
- spezieller Wert: **null** $\in W(T)$ = Zeiger, der "auf nichts zeigt" (definierte Adresse "Null").
- **Dynamische Speicherzuordnung** durch **new**, z.B. 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.



- Der Zugriff auf die referenzierten Objekte erfolgt direkt über den Instanznamen. Es gibt also im Unterschied zu Programmiersprachen wie MODULA-2 keinen **Dereferenzierungsoperator** wie z.B. "^".
- Der Speicherplatz, den man mit **new** reserviert hat, gibt man in Java nicht explizit frei. Stattdessen führt das Laufzeitsystem (Java Virtual Machine) zu (fast) beliebigen Zeitpunkten eine automatische Speicherbereinigung durch (garbage collection). Dabei wird der Speicherplatz von Objekten, die nicht mehr durch Referenzen zugänglich sind, zur Wiederverwendung freigegeben.

Verkettete Strukturen

Zeiger dienen dazu, zunächst unabhängige Datenobjekte miteinander zu verketten. Hierzu wird dem Objekt eine (oder mehrere) Objektreferenzen hinzugefügt:

```
            class VObjekt {
                                public Data Objekt;
                                public VObjekt next;
                        }
z.B.:       class Data      {
                                public int key;
                                public String info;
                        }
```

Abhängig von der Art der Verkettung und den auf der Struktur vorgesehenen Operationen unterscheiden wir z.B. folgende dynamische Datenstrukturen:

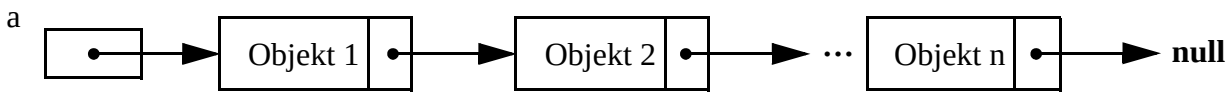
- lineare Listen
- kreisförmig verkettete Listen

- doppelt verkettete Listen
- Schlangen (Queues) / Stapel (Stacks)
- Bäume
- Graphen.

Lineare Listen

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 der im weiteren Verlauf beschriebenen Listenoperationen ist jedoch aus Effizienzgründen ungeeignet.

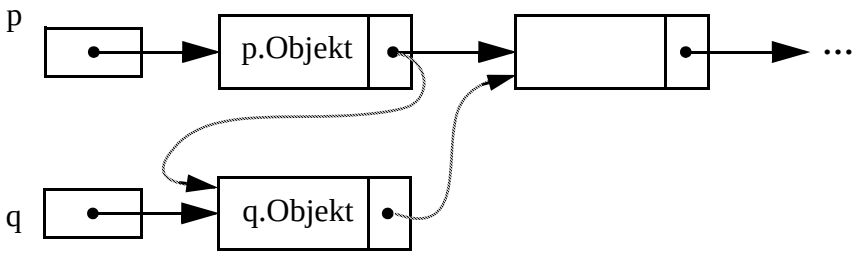
VObjekt a, p, q;



In den folgenden Prozeduren und Programmstücken seien die obigen Typdeklarationen von *VObjekt* und von *Data* gültig.

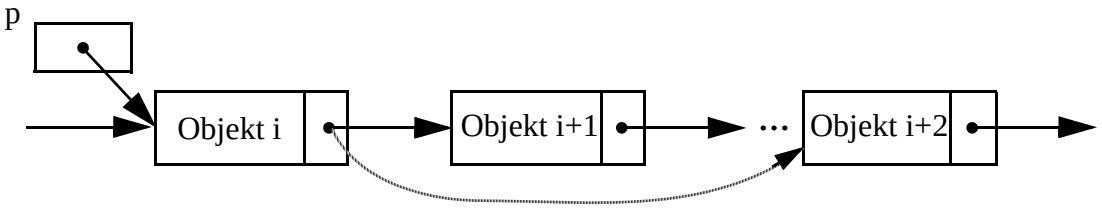
Allgemeine Operationen auf linearen Listen

1. Einfügen des Elements q hinter p



Programmcode: q.next := p.next; p.next := q;

2. Entfernen des Nachfolgers des Elements p



Programmcode: p.next = p.next.next;

Das Element mit Inhalt Objekt i+1 ist damit **nicht gelöscht**, sondern nur aus der Liste entfernt.

3. 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;
```

Beispiel 1: Suchen 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;
```

Zeitaufwand im schlechtesten Fall für eine Liste mit *n* Elementen

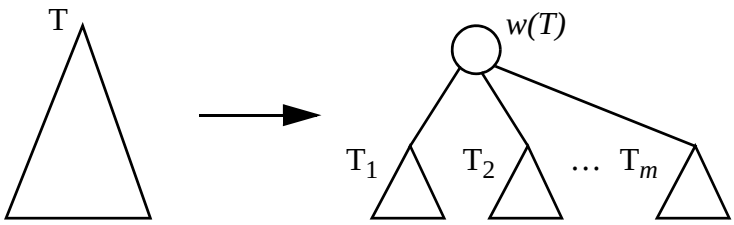
Operation	**sequentiell gespeichert (ARRAY)**	**verkettet gespeichert (Liste)**
Suche Objekt in sortierter Liste	$c \cdot \log n$	$c \cdot n$
Suche i-tes Objekt	konstant	$c \cdot i$
Einfügen an bekannter Stelle	$c \cdot n$	konstant
Entfernen an bekannter Stelle	$c \cdot n$	konstant

Baumstrukturen

Ein **Baum** ist eine endliche Menge T von Elementen, **Knoten** genannt, mit:

(1) Es gibt einen ausgezeichneten Knoten $w(T)$, die **Wurzel** von T

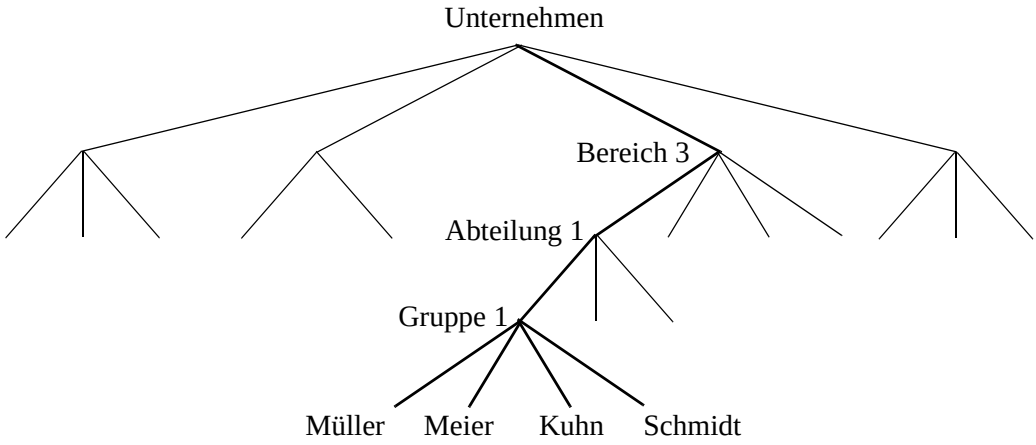
(2) Die restlichen Knoten sind in $m \geq 0$ disjunkte Mengen $T_1, \ldots, T_m$ zerlegt, die ihrerseits Bäume sind. $T_1, \ldots, T_m$ heißen **Teilbäume** der Wurzel $w(T)$. (*rekursive Definition*)



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

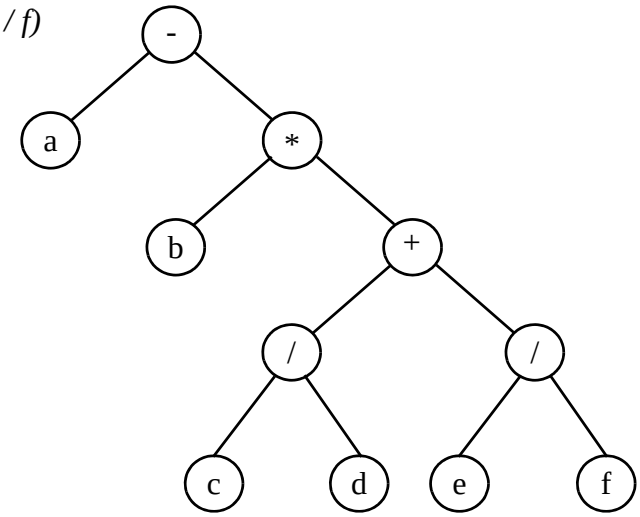
Bäume erlauben es, **hierarchische Beziehungen zwischen Objekten** darzustellen. Derartige Hierarchien treten vielfach in der realen Welt auf, z.B.:

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



- Die Gliederung eines Buches in Kapitel, Abschnitte, Unterabschnitte
- Die Aufteilung Deutschlands in Bundesländer, Bezirke, Kreise, Gemeinden
- 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)*



Der **Grad** eines Knotens *x*, ***deg(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) und *b(x)* die Menge aller **Brüder** von *x* (auch Geschwister von *x* genannt). Alle Brüder in *b(x)* haben denselben Vater wie *x*.

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

Der **Level** eines Knotens *x*, ***lev(x)*** ist: $lev(x) = \begin{cases} 1 & \text{für } x = w(T) \\ lev(v(x)) + 1 & \text{für } x \neq w(T) \end{cases}$.

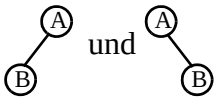
Damit ist $lev(x)$ gleich der Anzahl der Knoten auf dem Pfad von der Wurzel zum Knoten *x*.

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.

Ein **binärer Baum** ist eine endliche Menge B von Knoten, die

- entweder leer ist
- oder aus einer Wurzel und zwei disjunkten binären Bäumen besteht, dem linken und dem rechten Teilbaum der Wurzel.

Damit sind: 

und zwei verschiedene binäre Bäume, aber als Bäume gleich.

Für alle Knoten x eines binären Baumes gilt: $deg(x) \leq 2$.

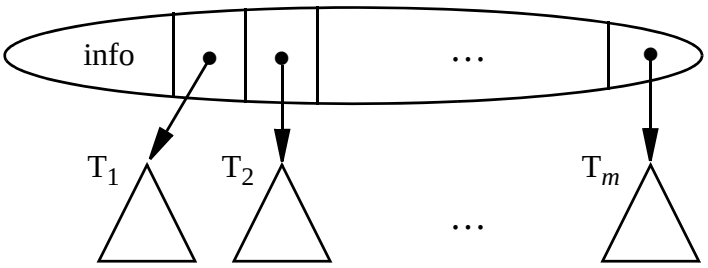
Beispiel: Arithmetischer Ausdruck (s.o.)

Dabei gilt:
- innere Knoten → Operatoren
- Blätter → Operanden

Speicherdarstellung von Bäumen

Erster allgemeiner Ansatz:

```
class T    {
               public Data info; // Data ist eine beliebige Klasse
               public T1 A1;
               . . .
               public Tm Am;    // 0 ≤ m ≤ Grad des Baumes
           }
```



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.

→ **schlechte Speicherplatzausnutzung** durch unnötige Zeigerfelder, denn:

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

$n \cdot (k\text{-}1)+1$ 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:

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

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