Datenstrukturen und Programmierverfahren

WS 1995/96

Prof. Dr. W. Effelsberg

Lehrstuhl für Praktische Informatik IV

Universität Mannheim

© Wolfgang Effelsberg

Übersicht

1. Grundlagen

1.1 Einleitung, Motivation

1.2 Elementare Datenstrukturen (Array, Liste, Stack, Queue)

1.3 Bäume

1.4 Analyse von Algorithmen/ Komplexität/ O-Kalkül

2. Prinzipielle Problemlösungstechniken

2.1 Teile-und-Herrsche (Divide and Conquer)

2.2 Greedy - Algorithmen

2.3 Dynamische Programmierung

2.4 Vollständige Enumeration

2.5 Rückverfolgung (Backtracking)

2.6 Branch-and-Bound

3. Sortieren

3.1 Elementare Methoden (Auswahl, Einfügen, Bubble Sort)

3.2 Quicksort

3.3 Heap-Sort

3.4 Merge-Sort

3.5 Radix-Sort

3.6 Externes Sortieren

4. Suchen

- 4.1 Elementare Methoden auf flachen Datenstrukturen
- 4.2 Binärbäume als Suchbäume
- 4.3 Mehrwegbäume als Suchbäume
- 4.4 Digitale Suchbäume (Radix-Suche)
- 4.5 Hash-Verfahren

5. Graphenalgorithmen

- 5.1 Elementare Methoden
- 5.2 Konnektivität
- 5.3 Gewichtete Graphen
- 5.4 Gerichtete Graphen
- 5.5 Flüsse in Netzwerken

6. Algorithmen auf Zeichenfolgen

- 6.1 Suche nach Strings
- 6.2 Pattern - Matching
- 6.3 Datenkomprimierung

7. Geometrische Algorithmen

- 7.1 Elementare Methoden
- 7.2 Konvexe Hülle
- 7.3 Bereichssuche

Literatur zur Vorlesung

Robert Sedgewick

Algorithmen in C.

Addison Wesley, 1992

Thomas Ottmann / **Peter Widmayer**

Algorithmen und Datenstrukturen.

2. Auflage, BI - Verlag

Mannheim, 1993

Ellis Horowitz / **Sartaj Sahni** / **Susan Anderson - Freed**

Grundlagen von Datenstrukturen in C.

International Thomson Publishing, 1994

1.1. Motivation

Ziel:Entwurf von **qualitativ hochwertigen Algorithmen**

Frage: Was ist das?

Analyse eines Programms:

a) allgemeine Kriterien

1. Erfüllt das Programm die ursprünglichen Spezifikationen der Aufgabe?
2. Arbeitet es fehlerfrei?
3. Enthält das Programm eine Dokumentation, die Aufschluß über Benutzung und Arbeitsweise gibt?
4. Ist der Programmcode mühelos lesbar? Ist Softwarewartung möglich?

b) Leistungsverhalten

1. Macht das Programm effizienten Gebrauch von primärem und sekundärem **Speicherplatz**?
2. Steht die **Laufzeit** in einem akzeptablen Verhältnis zur Aufgabe des Programms?

Wir werden uns in diesem Semester mit dem **Leistungsverhalten** beschäftigen.

1.2. Elementare Datenstrukturen

1.2.1 Arrays (Felder)

- Feste Anzahl von Datenelementen gleichen Typs; die Elemente werden indiziert.

1 - dimensional

a:

a_1	a_2	. . .

Index: Von der unteren Grenze bis zur oberen Grenze durchgehend, ganzzahlig

2 - dimensional

b:

b_{11}	b_{12}	b_{13}			
b_{21}	b_{22}	b_{23}			
b_{31}	b_{32}	b_{33}			

beliebig viele Dimensionen möglich.

Vorteile:

- auf jedes Element kann unmittelbar zugegriffen werden
- ein Array kann auch sequentiell durchlaufen werden
- die konsekutive Anordung von Arrayelementen korrespondiert zur konsekutiven Anordnung von Speicherworten
  => die meisten Arrayzugriffe sind sehr schnell
- entspricht den Vektoren und Matrizen der Mathematik, daher einfache, natürliche Abbildung von Anwendungsdaten

Nachteile:

- Größe fest (Pascal : bei Übersetzung / C : beim Eintritt in das Modul)
- Einfügen oder Löschen von Elementen erfordert im allgemeinen Kopieren von Daten.
- Speicherplatzverschwendung bei dünn besetzten Feldern oder Matrizen

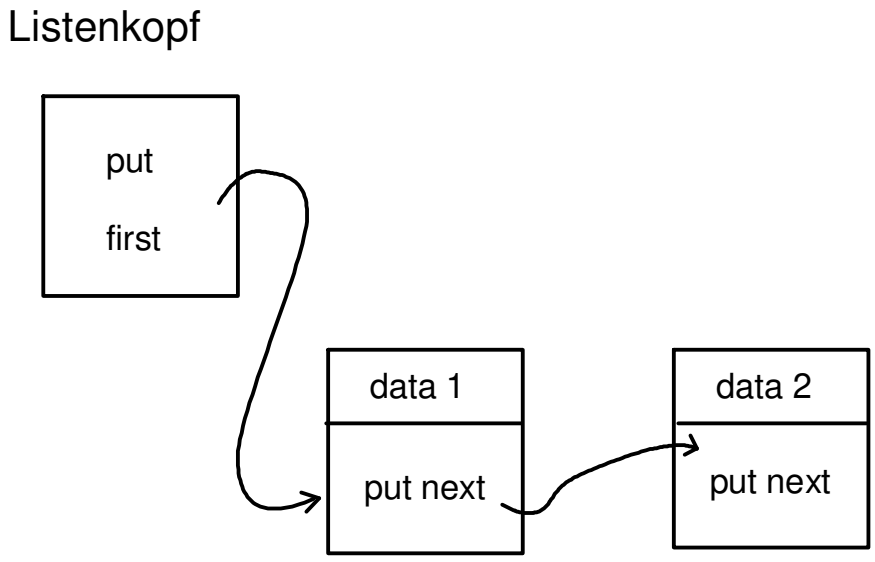
1.2.2. Verkettete Listen ("linked lists")

- Veränderliche Anzahl von Elementen gleichen Typs
- Anordnung der Elemente im Speicher muß explizit verwaltet werden

Implementierungsmöglichkeiten:

(a) Listen ohne explizite Plazierung

- jedes Element kann an beliebiger Stelle im Speicher stehen
- jedes Element benötigt einen eigenen Pointer (= Speicheradresse)
- Pointer werden im Listenelement selbst untergebracht
- Explizit angelegt; Pointer auf den Anfang (Ende) der Liste

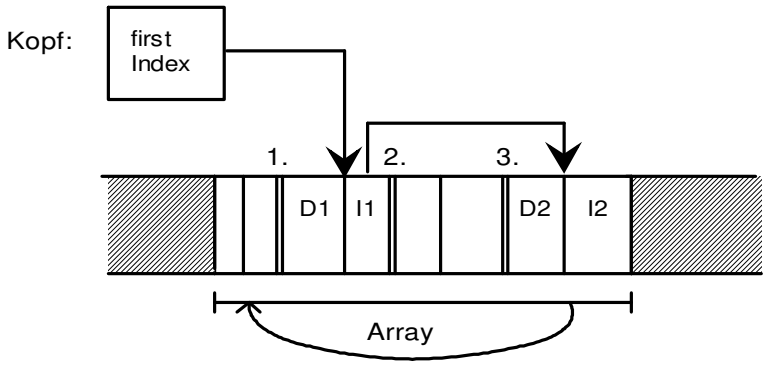


(b) Listen mit expliziter Plazierung

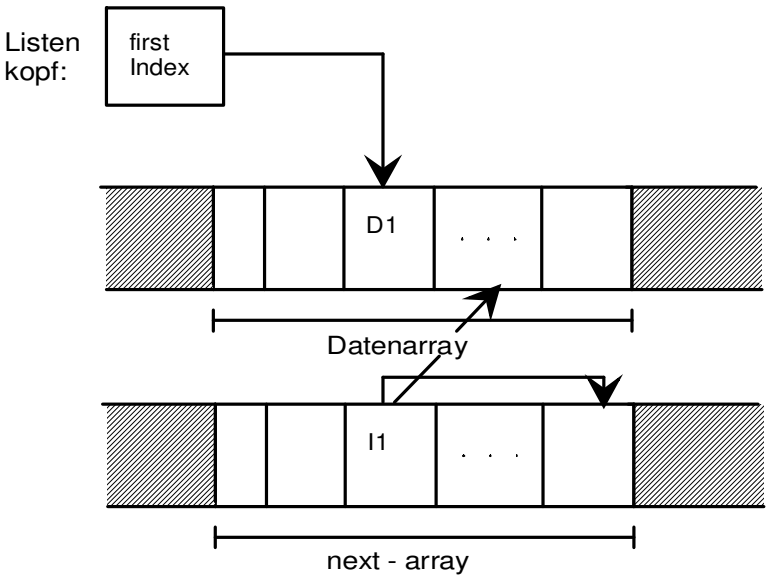
- jedes Element kann an einer beliebigen Stelle eines Arrays stehen
- Pointer werden durch die Array - Indizes ersetzt

Direkte und indirekte Verkettung

(a) Direkte Verkettung



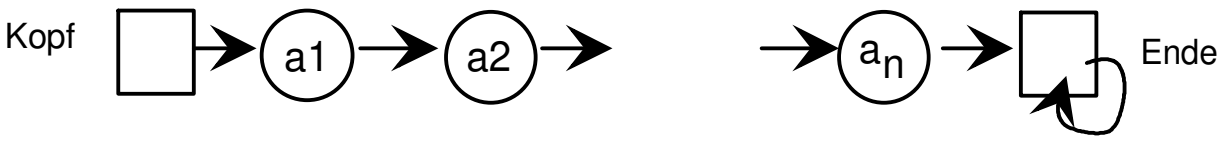
(b) Indirekte Verkettung



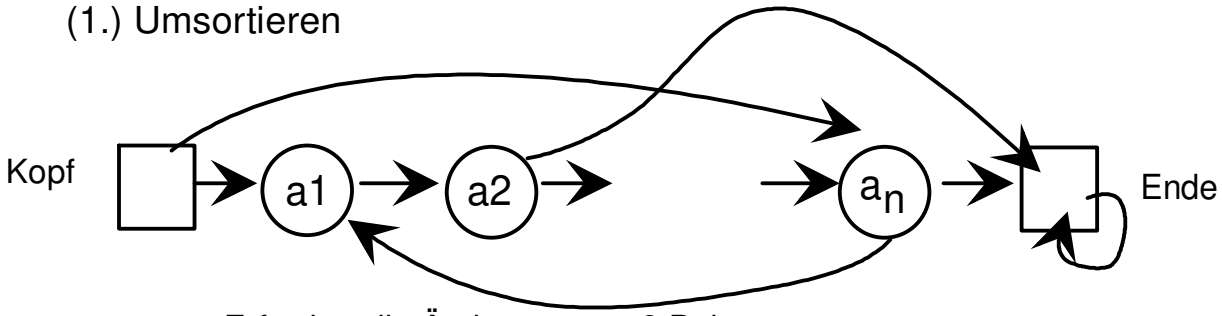
Vorteile:

- Daten und Verwaltungsinformation getrennt
- gleiche Daten können zu mehreren Ketten gehören

Einfach verkettete Listen

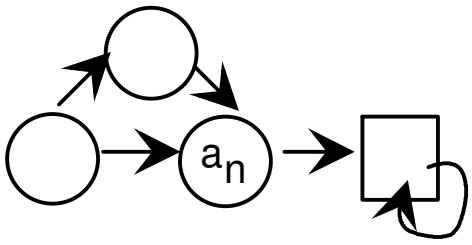


(1.) Umsortieren

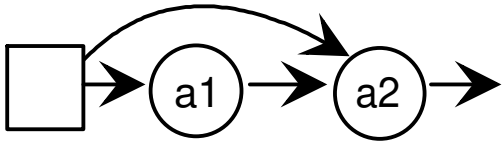


- Erfordert die Änderung von 3 Pointern
  (unabhängig von der Länge der Liste)

(2.) Einfügen eines neuen Elementes



(3.) Löschen eines Elementes



Probleme der einfach verketteten Liste

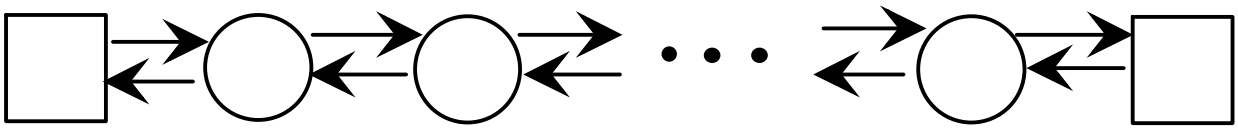
- Zugriff erfolgt immer sequentiell
- Vohergehende Elemente sind nicht leicht verfügbar

=> - Einfügen eines Elementes nur **hinter** einer bestimmten Position möglich
   - Löschen von Elementen nicht einfach möglich (Vorgänger "merken")

Implementierung in C

```
struct node
   { int key; struct node *next; };
struct node *head, *z, *t;
listinitialize ()
   {
     head = (struct node *) malloc (sizeof *head);
     z = (struct node *) malloc (sizeof *z);
     head->next = z; z->next = z;
   }
deletenext (struct node *t)
   {
   struct node *temp;
   temp = t->next;
   t->next = t->next->next;
   free(temp);
   }
struct node *insertafter (int v, struct node *t)
   {
     struct node *x;
     x = (struct node *) malloc (sizeof *x);
     x->key = v; x->next = t->next;
     t->next = x;
     return x;
   }
```

1.2.2. Doppelt verkettete Liste



Vorteil: Einfügen und Löschen einfacher

Nachteil: Doppelt soviele Pointeroperationen pro Änderungsvorgang; höherer Speicherbedarf

Implementierung in C

Übungsaufgabe !!

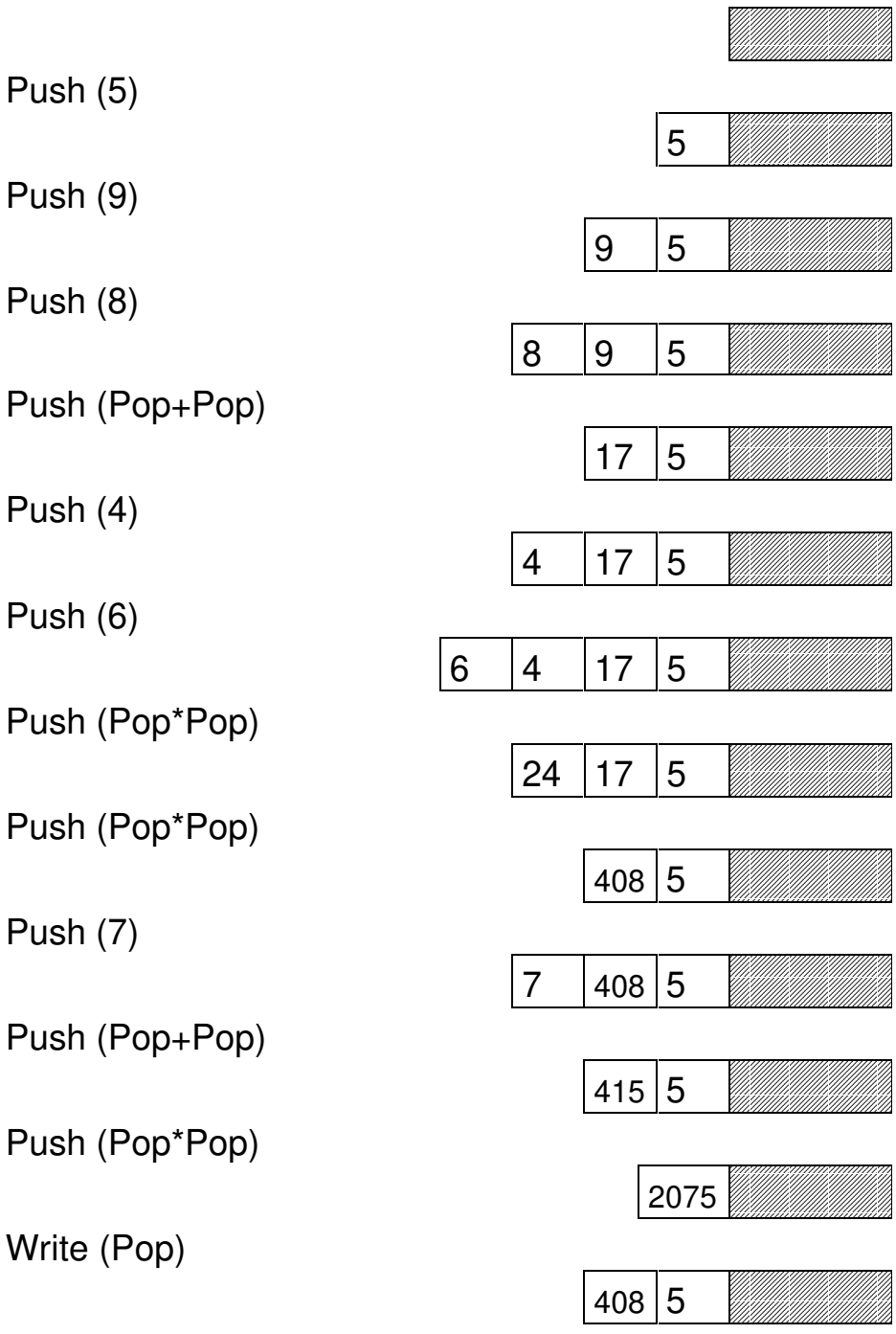
1.2.3. Stapel oder Kellerspeicher (Stack)

- Ein Stapel ist eine Liste, an deren Anfang Elemente eingefügt oder gelöscht werden können. Nur das erste Element ist zugreifbar.
  Es gibt die Verarbeitungsfolge LIFO (Last - In, First - Out).
- Operationen

Push (arg)	:	Einfügen von arg am Anfang des Stapels
Pop	:	Auslesen und Löschen des ersten Elementes

Verwendungsbeispiel für den Stapel

Berechnung von arithmetischen Ausdrücken



1.2.4 Warteschlange (Queue)

Eine **Warteschlange (Queue)** ist eine Liste, bei der ein Einfügen eines neuen Elements nur am Anfang und ein Lesen/ Entfernen eines Elements nur am Ende erfolgen kann.
Die Bearbeitungsfolge ist FIFO (First In, First Out).

Warteschlangen treten in der Informatik sehr häufig auf, z.B. Druckaufträge am Drucker, Datenpakete in der Netzanschlußkarte, usw.
Die Darstellung im Rechner erfolgt durch verkettete Listen mit entsprechend eingeschränkten Einfüge- und Löschoperationen.

1.3. Bäume

Definition 1

Ein Baum besteht aus einer nichtleeren Menge von *Knoten* und einer Menge von *Kanten*.
Jede Kante verbindet genau zwei Knoten.

Ein *Pfad* ist eine Liste von unterschiedlichen Knoten, in der aufeinanderfolgende Knoten durch Kanten verbunden sind.

Es gibt einen ausgezeichneten Knoten, der als *Wurzel* bezeichnet wird.

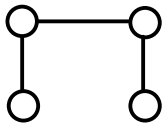
Zwischen der Wurzel des Baumes und jedem anderen Knoten gibt es nur genau einen Pfad.

Definition 2 (als spezieller Graph)

Ein Graph G ist ein Baum, wenn gilt (alternativ):

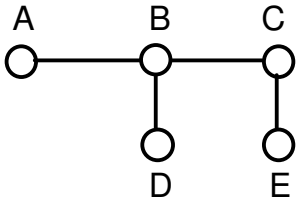
- G enthält keinen Zyklus, aber durch Hinzufügen einer beliebigen Kante würde ein Zyklus entstehen.

  Beispiel:



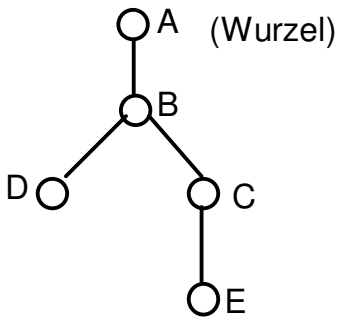
- Je zwei Knoten von G sind durch genau einen Pfad verbunden.
- G ist zusammenhängend, aber Entfernen einer beliebigen Kante macht G nicht - zusammenhängend.

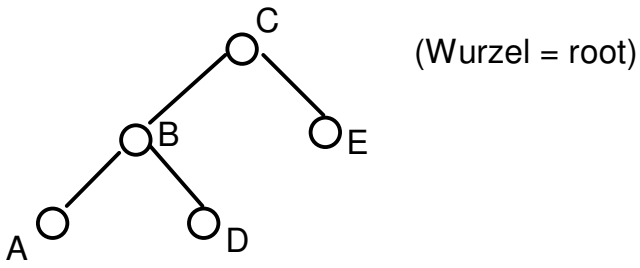
  Beispiel:



Im folgenden: „ Orientierte Bäume“

„ Wurzelbäume“

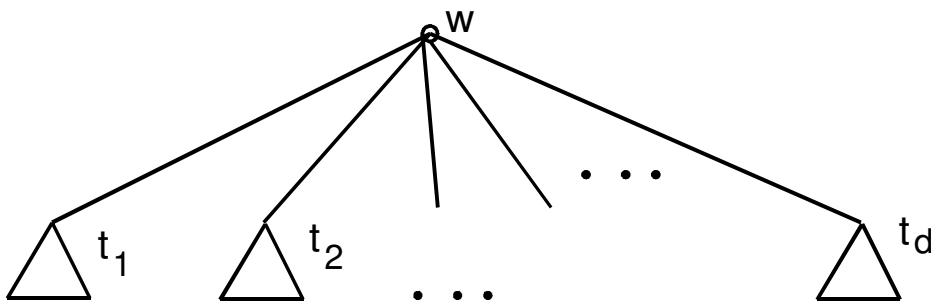




Definition 3 (rekursiv)

(1) Der aus einem einzigen Knoten bestehende Baum ist ein Baum der Ordnung *d*.

(2) Sind t_1,...., t_d beliebige Bäume der Ordnung *d*, so erhält man einen (weiteren) Baum der Ordnung *d*, indem man die Wurzeln von t_1,...., t_d zu Söhnen einer neugeschaffenen Wurzel *w* macht. t_i ($1 \leq i \leq d$) heißt *i-ter Teilbaum* der Wurzel *w*.



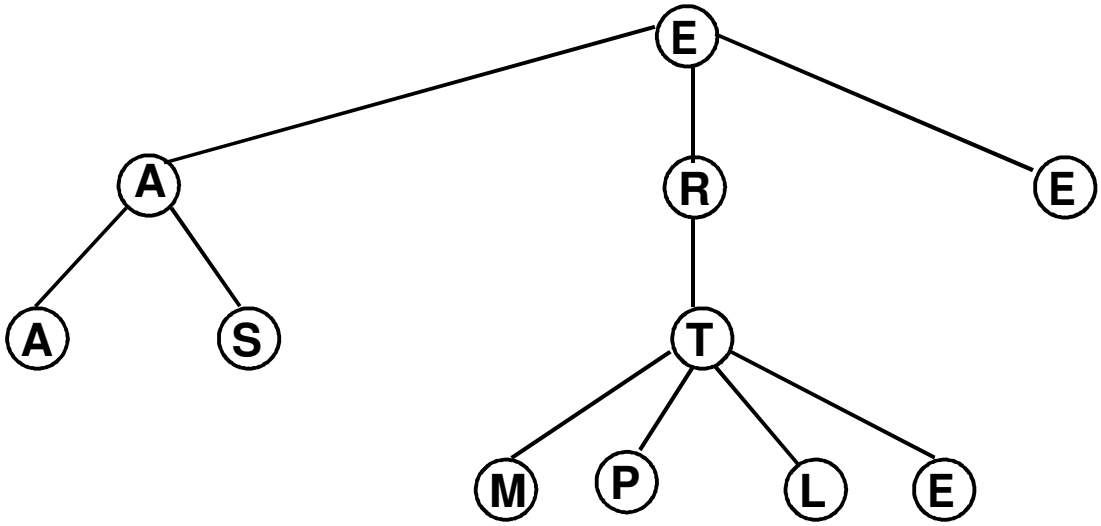
(Ein spezieller Baum, bei dem jeder Vater genau d Söhne hat.)

Terminologie

- Jeder Knoten (außer der Wurzel) hat genau einen **Elternknoten**.
- Die mit ihm verbundenen Knoten sind seine **Kinder** oder **Söhne**.
- Ein Knoten, der keine Kinder hat, heißt **Blattknoten**.
- Jeder Knoten ist die Wurzel eines **Unterbaumes**.
- Ein Baum heißt vom **Rang** d, wenn jeder Knoten außer den Blattknoten maximal d Söhne hat.
- Jedem Knoten kann ein **Ebene** zugeordnet werden

  Ebene = Anzahl der Knoten auf dem Pfad zur Wurzel
- **Höhe** eines Baumes = maximale Ebenen - Nummer
- Die **Pfadlänge** eines Baumes ist die Summe aller Pfadlängen von einem Knoten zur Wurzel.
- Ein **vollständiger** Baum ist ein Baum, bei dem alle Ebenen bis auf die unterste vollständig besetzt sind.

Beispiel für einen allgemeinen Baum



Darstellung von allgemeinen Bäumen

Problem: Jeder Knoten kann eine beliebige Anzahl von Kindern haben! Deshalb können die Zeiger auf die Kinder nicht im Elternknoten gespeichert werden.

Lösungen:
- „Parent - link“ Darstellung
- Listendarstellung der Kinder

Parent - link

- Zu jedem Knoten wird nur der Zeiger auf seinen Elternknoten gespeichert.
- Möglich falls der Baum nur von unten nach oben durchlaufen werden soll.

Listendarstellung der Knoten

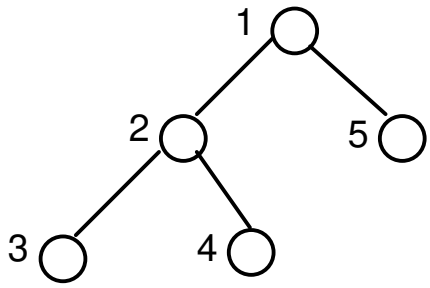
- Ein Elternknoten hat einen Zeiger zu seinem ganz linken Kind.
- Die Kinder sind als einfach verkettete Liste angelegt.
- Jeder Knoten hat zwei Zeiger:
  - einen zu seinem rechten Bruder (oder zum Elternknoten zurück)
  - einen zu seinem ganz linken Kind (falls es ein solches gibt)

Ordnungen von Bäumen

- Eine Ordnung ist eine Abbildung des Baumes auf eine lineare Struktur ("Plattklopfen des Baumes").
- Bildung einer eindeutigen Knotenfolge derart, daß jeder Knoten genau einmal darin vorkommt.
- Drei Ordnungs - Schemata
    - Preorder (Präfix - Notation)
    - Inorder (Infix - Notation)
    - Postorder (Postfix - Notation)

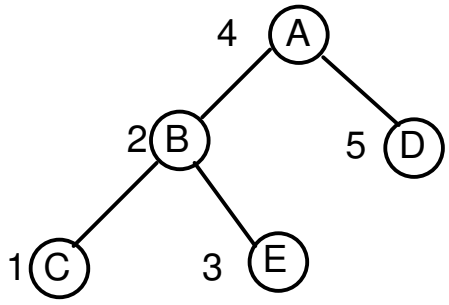
Preorder

Regel:
1. Wurzel
2. Linker Unterbaum
3. Rechter Unterbaum



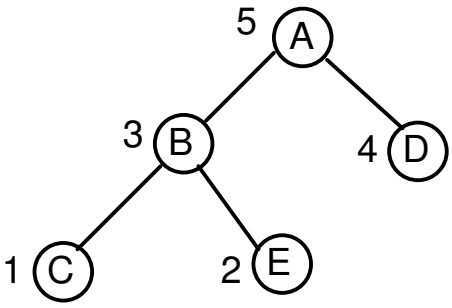
Inorder

Regel: 1. Linker Unterbaum
2. Wurzel
3. Rechter Unterbaum



Postorder

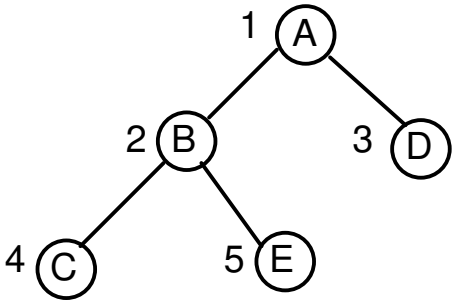
Regel: 1. Linker Unterbaum
2. Rechter Unterbaum
3. Wurzel



Level - Order

Regel: 1. Alle Knoten einer Ebene werden von links nach rechts besucht.

2. Alle Ebenen werden von oben nach unten behandelt.



Binärbäume

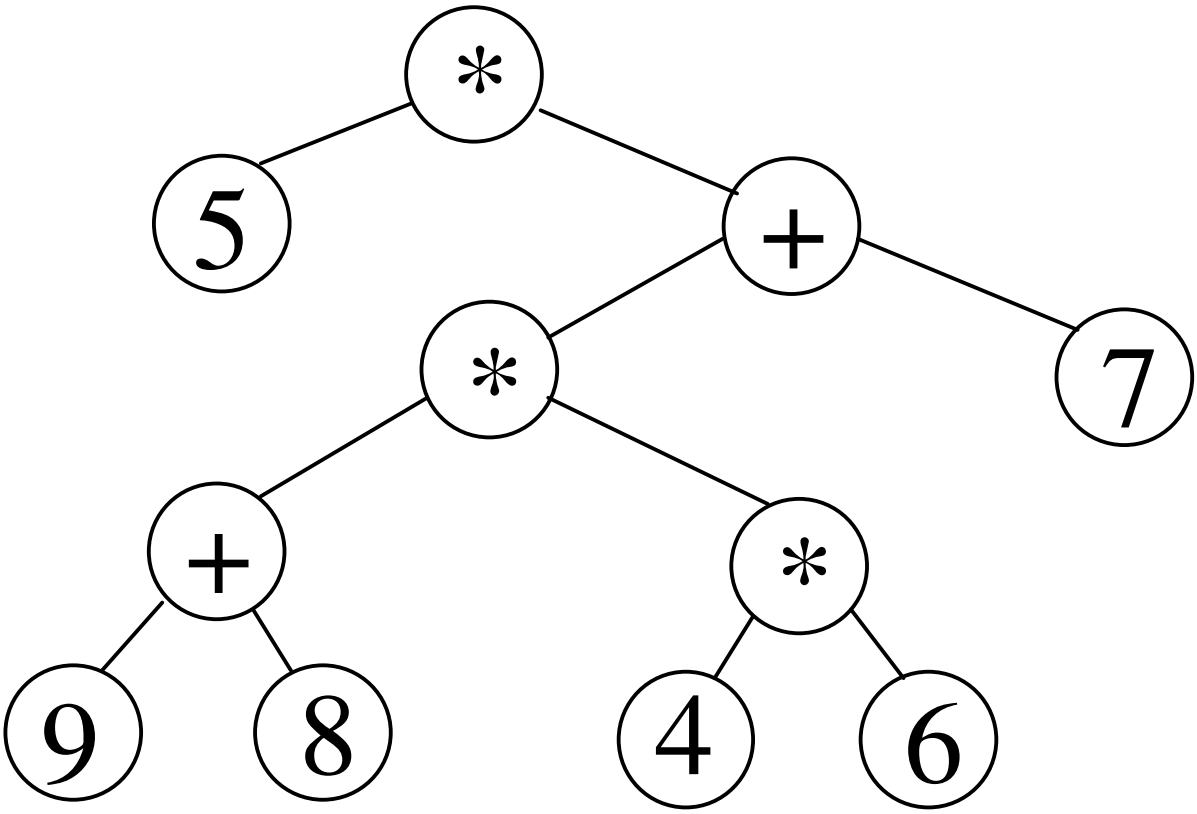
Definition:
Ein Baum heißt *Binärbaum* genau dann, wenn er vom Rang 2 ist, d.h. wenn jeder Elternknoten maximal zwei Söhne hat.

Anmerkung 1:
In der Literatur findet sich auch eine abweichende Definition von *Rang:*
Ein Baum ist vom Rang d, wenn jeder Knoten außer den Blattknoten **genau** d Söhne hat.

Anmerkung 2 :
Binärbäume sind maschinenintern leicht darzustellen.
Es werden pro Knoten zwei Zeiger fest eingerichtet.

Binärbaum, Beispiel 1: Operatorbaum

$$5 * (((9+8) * (4*6))+7)$$



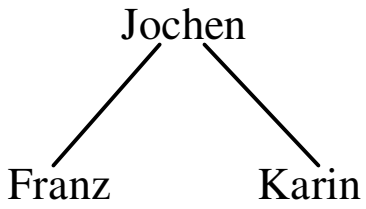
Dient zur maschineninternen Darstellung von arithmetischen Ausdrücken.

Eine Bearbeitung in Postorder berechnet den Wert des Ausdrucks.

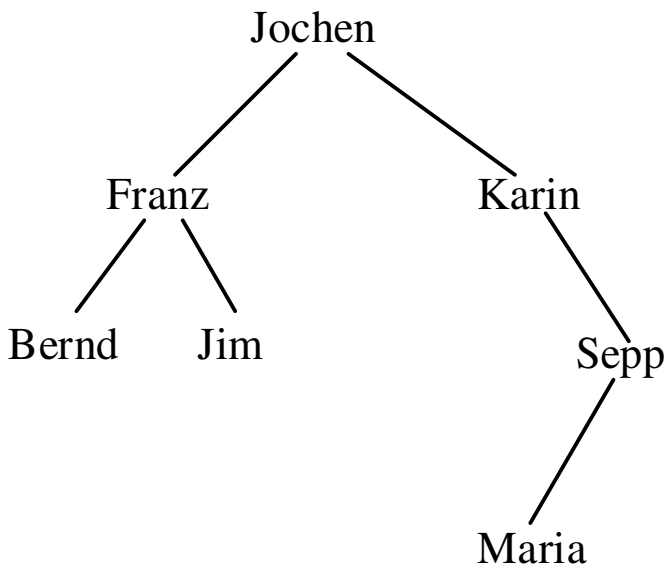
Binärbaum, Beispiel 2: alphabetische Suchbäume

Zwei alphabetisch sortierte Binärbäume

a)



b)



1.4. Analyse von Algorithmen, Komplexität, O-Kalkül

1.4.1. Beurteilung des Leistungsverhaltens von Algorithmen

- Unabhängig vom Rechner
  => Standardoperationen, die in einer Einheitszeit ausgeführt werden
- Kriterien

  Benötigte Ressourcen:

  - Rechenzeit
  - Speicherbedarf

  in Abhängigkeit von der Problemgröße für den

  - ungünstigsten Fall (worst case)
  - günstigsten Fall (best case)
  - durchschnittlichen Fall
  - häufigsten Fall.

Typische Vorgehensweise:

- Charakterisierung der Eingangsdaten (Anzahl, Werte, Verteilung)
- Bestimmung der zeitkritischen Operationen des Algorithmus
- Bestimmung von worst- case und mittleren Werten für Rechenzeit und gegebenenfalls Speicherbedarf in Abhängigkeit von der Problemgröße

O-Kalkül

Berechnung der Komplexität unter Abstraktion von Details:

- von Konstanten (die maschinenabhängig sind)
- von weniger dominanten Termen (die für große Werte des Komplexitätsparameters N nicht mehr signifikant sind)

Definition 1:

Seien f und g zwei positive, reellwertige Funktionen, die auf einem gemeinsamen Bereich D definiert sind.

Dann bedeutet

$$f = O(g) ,$$

daß es eine Konstante $c > 0$ und einen Grenzwert x_0 gibt derart, daß

$$f(x) \leq cg(x) \text{ für alle } x > x_0$$

gilt.

Man sagt: "f wächst asymptotisch höchstens wie g" oder "f ist O(g)".

Definition 2:

Es bedeutet

$$f = \Omega (g)$$

wenn es eine Konstante $c > 0$ und einen Grenzwert x_0 gibt derart, daß

$$f(x) \geq cg(x) \quad \text{für alle } x > x_0$$

gilt.

Man sagt, f wächst asymptotisch mindestens wie g.

Beispiel 1: Lineare Suche

Gegeben sei eine unsortierte Liste mit N Elementen, in der ein bestimmter Schlüsselwert zu suchen ist.

Im Mittel wird der gesuchte Wert nach N/2 Schritten (Vergleichen) gefunden.

Im "worst case" wird der gesuchte Wert nach N Schritten gefunden.

Die lineare Suche ist O (N).

Beispiel 2: Sortieren durch Auswahl

Gegeben seien N Zahlen, die zu sortieren sind.

Algorithmus Auswahlsortieren

Wiederhole N-mal

- Markiere die erste Zahl der Eingabemenge.
- Vergleiche sie mit der zweiten, dritten usw.
- Wenn eine kleinere als die erste gefunden wird, markiere diese.
- Übertrage die markierte kleinste Zahl in den Ausgabe - Datenstrom und sperre sie für die weitere Bearbeitung.

Komplexität des Sortierens durch Auswahl

Eingabeparameter: N = Anzahl der Elemente

Vergleich zweier Zahlen in konstanter Zeit c_1 möglich.

Innere Schleife: $t_1 = N \cdot c_1$

Äußere Schleife: $t_2 = N \cdot t_1 = N^2 c_1$

$$t = t_2 = N^2 c_1 = O(N^2)$$

Sortieren durch Auswahl ist $O(N^2)$.

Einfache Rechenregeln des O-Kalküls

(1)	f	=	$O(f)$
(2)	$O(O(f))$	=	$O(f)$
(3)	$kO(f)$	=	$O(f)$ mit einer Konstanten k
(4)	$O(f+k)$	=	$O(f)$
(5)	$O(f) + O(f)$	=	$O(f)$
(6)	$O(f) * O(g)$	=	$O(fg)$
(7)	$O(f) + O(g)$	=	$O(f+g)$
(8)	$O(fg)$	=	$O(f) * O(g)$.

Konvention: Diese Gleichungen werden von links nach rechts gelesen!

Beweis von (7) als Beispiel

(7) O(f) + O(g) = O(f+g)

$h_1(n) = O(f)$, also: Es gibt c_1, n_1 mit $h_1(n) \leq c_1 f(n)$,
für alle $n > n_1$

$h_2(n) = O(g)$, also: Es gibt c_2, n_2 mit $h_2(n) \leq c_2 g(n)$,
für alle $n > n_2$

Addition liefert

$h_1(n) + h_2(n) \leq c_1 f + c_2 g$ für alle $n > \max(n_1,n_2)$,

Setzen wir $h_3(n) = h_1(n) + h_2(n)$, $c_3 = \max(c_1,c_2)$ und $n_3 = \max(n_1,n_2)$, so gilt

$h_3(n) \leq c_3 (f + g)$, für alle $n > n_3$,

was nach Definition $h_3(n) = O(f + g)$ ist.

Zwei Arten von Schranken

Für ein gegebenes Problem läßt sich oft eine untere Schranke für die Komplexität einer Lösung des Problems **aus der Theorie** angeben, unabhängig vom Algorithmus.

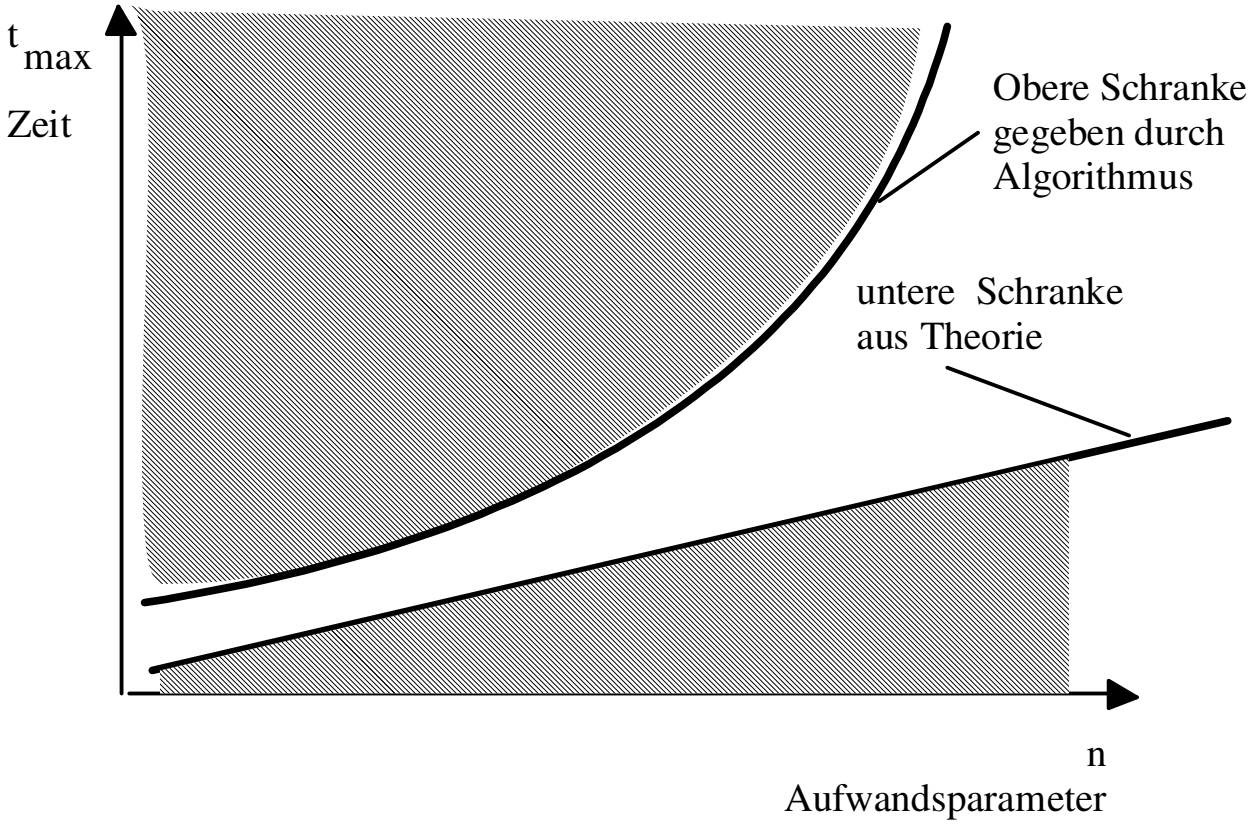
Beispiel P_1: Suche das Minimum aus N unsortierten Zahlen.
Jede Zahl muß mindestens einmal angesehen (verglichen) werden. Daher gilt: P_1 hat Komplexität $\Omega(N)$.

Darüber hinaus läßt sich für einen gegebenen Algorithmus die Abhängigkeit der Anzahl der Operationen von N ermitteln und dadurch **eine für diesen Algorithmus spezifische** Schranke ermitteln.

Beispiel: Ein **dummer** (!) Algorithmus A_1 **sortiert** die N Zahlen und erhält dann das Minimum als erste Zahl des Ergebnisses.

Sortieren durch Auswahl:	$O(N^2)$
Bestimmung der ersten Zahl:	$O(1)$
A_1 hat also Komplexität	$O(N^2)$

Obere und untere Schranken



Die Erforschung solcher Schranken wird in der Komplexitätstheorie behandelt, einem wichtigen Zweig der Theoretischen Informatik.

Wichtige Beispiele für asymptotisches Verhalten

O(1)	Laufzeit ist konstant. Alle Operationen werden eine feste Anzahl mal ausgeführt, unabhängig von N.
O(log N)	Laufzeit steigt schwach, wenn Problem in Teilprobleme zerlegt wird, deren Zusammenfügen leichter ist.
O(N)	Laufzeit wächst linear. Die gleichen Operationen werden typischerweise auf jedem Datenelement einmal durchgeführt.
O(N log N)	Laufzeit wächst etwas stärker als linear an, Problem wird in Teilprobleme zerlegt, von denen alle einzeln gelöst werden.
$O(N^2)$	Laufzeit wächst stark mit der Problemgröße. Häufig, wenn Paare von Eingabedaten bearbeitet werden.
$O(2^N)$	Laufzeit wächst exponentiell. Häufig, wenn für jedes neue Datenelement die bisherige Arbeit noch einmal wiederholt werden muß.

Beispiele für die Anwendung des O-Kalküls

$3n + 2 = O(n)$, da $3n + 2 \leq 4n$ für alle $n \geq 2$ gilt.

$3n + 3 = O(n)$, da $3n + 3 \leq 4n$ für alle $n \geq 3$ gilt.

$100n + 6 = O(n)$, da $100n + 6 \leq 101n$ für alle $n \geq 101$ gilt.

$10n^2 + 4n + 2 = O(n^2)$, da $10n^2 + 4n + 2 \leq 11n^2$ für alle $n \geq 5$ gilt.

$1000n^2 + 100n - 6 = O(n^2)$, da $1000n^2 + 100n - 6 \leq 1001n^2$ für alle $n \geq 100$ gilt.

$6*2^n + n^2 = O(2^n)$, da $6*2^n + n^2 \leq 7*2^n$ für alle $n \geq 4$ gilt.

$3n + 3 = O(n^2)$, da $3n + 3 \leq 3n^2$ für alle $n \geq 2$ gilt.

$10n^2 + 4n + 2 = O(n^4)$, da $10n^2 + 4n + 2 \leq 10n^4$ für alle $n \geq 2$ gilt.

$3n + 2 \neq O(1)$, da es für jede Konstante c und jedes n_0 ein $n \geq n_0$ gibt, so daß

$3n + 2 > c$ gilt.

$10n^2 + 4n + 2 \neq O(n)$.

Zur Übung: Komplexität der Multiplikation von zwei n x n-Matrizen ?

Theorem zum polynomialen Aufwand

Theorem:

Wenn $f(n) = a_m n^m + ... + a_1 n + a_0$ gilt, dann ist $f(n) = O(n^m)$.

Beweis:

$$\sum_{i=0}^{m} a_i n^i \leq \sum_{i=0}^{m} a_i n^i$$

$$\leq a_m n^m + \sum_{i=0}^{m-1} a_i n^i$$

Es gilt: $n^m \geq n^{m-1}$ für alle $n \geq 1$, $m \geq 0$

Damit gilt:

$$a_m n^m + \sum_{i=0}^{m-1} a_i n^i \leq a_m n^m + a_{m-1} n^m + \sum_{i=0}^{m-2} a_i$$

$$\leq n^m \sum_{i=0}^{m} a_i$$

Mit $c = \sum_{i=0}^{m} a_i$ ist

$$... = c * n^m$$ q.e.d.

Anmerkungen

1) Die O-Notation sagt nur etwas über das Verhalten des Algorithmus für großes N aus!
Was ist groß?
N muß so groß sein, daß die Vernachlässigung kleinerer Beiträge berechtigt ist.

2) Die O-Notation bezieht sich immer auf worst-case-Verhalten und stellt eine obere Grenze dar! Das Verhalten mit durchschnittlichen Eingabemengen kann ganz anders sein.

3) Die Konstante c kann bei kleinerem N einen erheblichen Einfluß haben:

=> In der Praxis könnte ein O(N)-Algorithmus langsamer sein als ein O(N*logN)-Algorithmus (z.B. für mittelgroßes N).

4) Wir interessieren uns in der Regel dafür, die **kleinstmögliche** Funktion g(x) zu finden, für die gilt: $f(x) = 0(g(x))$