Kapitel 1 - Grundlagen

Problemstellung
Algorithmen und Komplexität
Rekursionsgleichungen
Entwurf von Algorithmen
Datenstrukturen

Problemstellung

- Algorithmen: Suche nach guten Problemlösestrategien
- Datenstrukturen: Repräsentation der Daten ist wichtig für die Effizienz
- Trotz schnell wachsender Speicher- und Rechenkapazitäten ist die ökonomische Nutzung eine wichtige Fragestellung der Informatik:
  - Ressourcen können geteilt werden; dadurch viele nebenläufige Prozesse möglich (Beispiel: geringer „Memory-Footprint“ ist nützlich)
  - Ineffiziente Algorithmen erschöpfen schnell auch noch so große Kapazitäten
  - Theoretischen Informatik: noch offene Fragen nach effizienten Lösungen; Annäherung durch approximative Algorithmen

Ziele der Vorlesung

- Aufwandsquantifizierung von Algorithmen
  - Rechenzeit, Speicherplatz (später auch I/Os auf Netz, Platte, …)
  - Komplexitätsklassen
  - Analyse von Algorithmen
- Analyse eines gegebenen Problems
- Abbildung auf Datenstrukturen und Algorithmen
- Implementierungsfragen

Zentraler Begriff *Algorithmus*

- Definition

Ein Algorithmus ist ein *wohldefiniertes*, *schrittweises* *Verfahren* zur Lösung eines *allgemeinen* Problems.

- Etymologie (Wortherkunft)
  - Mathematiker *Al Chwarizmi,* ca. 783 - ca. 850 n.Chr.
  - Arabisches Lehrbuch „*Über das Rechnen mit indischen Ziffern* “
  - Machte Algebra im Westen bekannt
  - Lateinische Fassung beginnt mit: "Dixit Algorithmi …"

Beispiel für Algorithmus: *SummeBis(n)*

- Spezifikation der Aufgabe:

  Berechne die Summe der Zahlen von 1 bis *n*.

- Natürliche Sprache

  Initialisiere eine Variable summe mit 0. Durchlaufe die Zahlen von 1 bis n mit einer Variable *zähler* und addiere *zähler* jeweils zu *summe*. Gib nach dem Durchlauf den Text “Die Summe ist: “ und den Wert von *summe* aus.

- Pseudocode

```
setze summe := 0;
setze zähler := 1;
solange zähler ≤ n:
        setze summe := summe + zähler;
        erhöhe zähler um 1;
gib aus: “Die Summe ist “ und summe;
```

SummeBis(n) als Java-Programm

```
class SummeBis {
    public static void main (String[] arg) {
        int n = Integer.parseInt(arg[0]);
        int sum = 0;
        for (int i = 1; i <= n; ++i) {
          sum += i;
        }
        System.out.println („Die Summe ist “ + sum);
    }
}
```

Eigenschaften von Algorithmen

Statische	Allgemeinheit	Lösung für Problemklasse, nicht für konkrete Einzelaufgabe
	Operationalität	Einzelschritte sind wohldefiniert und ausführbar
	Endlichkeit	Die Notation des Algorithmus hat eine endliche Länge
	Änderbarkeit	Algorithmus ist anpassbar an modifizierte Anforderungen
Funktionale	Funktionalität	Algorithmus reagiert auf Eingaben und produziert Ausgaben
	Determiniertheit	Ergebnis ist festgelegt für jede Eingabe
	Robustheit	Algorithmus ist robust gegen Fehler unterschiedlicher Art
	Korrektheit	Algorithmus liefert nur richtige Ergebnisse
	Vollständigkeit	Algorithmus liefert alle gewünschten Ergebnisse
Dynamische	Terminierung	Algorithmus läuft für jede Eingabe nur endlich lange
	Determinismus	Ablauf ist für dieselbe Eingabe immer gleich
	Effizienz	Algorithmus ist sparsam im Ressourcenverbrauch

Effizienz von Algorithmen

- Kategorien
  - Rechenzeit (Anzahl der Einzelschritte)
  - Speicherplatzbedarf
  - Zugriffe auf Sekundärspeicher (z.B. Festplatte)
  - Kommunikationsaufwand (z.B. Netzwerk)
- konkrete Laufzeit eines Algorithmus hängt von vielen Faktoren ab
  - Takt der CPU, Länge der Eingabe, Implementierung der Basisoperationen
    → axiomatisches Rechnermodell als Vergleichsmaßstab (eingeschränkt) möglich
- Komplexität
  - Abhängig von Eingabedaten
  - z.B. abstrakte Rechenzeit $T(n)$ abhängig von n:
    - Bei der Suche in Mengen: Anzahl der benötigten Vergleiche bei n Elemente in der Menge
    - Bei Sortierverfahren: Anzahl der zu sortierenden Werte $a_1,\ldots a_n$
    - Bei der Suche nach Primzahlen: Obere Grenze des abzusuchenden Zahlenbereichs, d.h. suche Primzahlen, die kleiner oder gleich n sind
  - Üblicherweise asymptotische Betrachtung

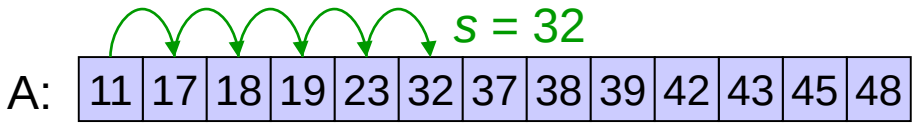
Beispiel: Suche in einer Liste

- Spezifikation der Aufgabe
  - Informell: Suche in einer Liste *A* den Wert *s*
  - Formaler: Gib für einen Suchwert *s* aus der Reihe *A*[1] … *A*[*n*] eine Position *i* mit *A*[*i*] = *s* aus bzw. melde „nicht gefunden“.
- Algorithmus *SequenzielleSuche* (Array *A*, Suchwert *s*)

  Durchlaufe alle Indizes *i* := 1, ..., *n*:

  Falls *A*[*i*] = *s*, gib *i* aus und beende den Durchlauf.

  Falls *s* nicht gefunden, gib Meldung „nicht gefunden“ aus
- Beispiel



  - SequenzielleSuche (A, 32) liefert 6.
  - SequenzielleSuche (A, 41) meldet „nicht gefunden“.
- Analyse der Laufzeit für die sequenzielle Suche
  - Erfolgreiche Suche: *n* Vergleiche maximal, *n*/2 im Durchschnitt
  - Erfolglose Suche: *n* Vergleiche

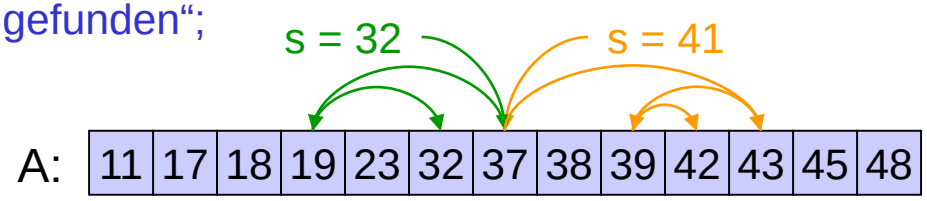
Algorithmus „Binäre Suche“

- Falls Array *A* sortiert ist, kann man *s* schneller suchen
  - Liegt s nicht in der Mitte von A, dann suche in der linken Hälfte von A, falls s zu klein ist bzw. in der rechten Hälfte, falls s zu groß ist.

- Algorithmus BinäreSuche (Array *A*, Suchwert *s*)

```
links := 1; rechts := Länge(A);  {jeweils aktuelle Ränder}
while links ≤ rechts do
    mitte := (links + rechts)/2;   {Mitte bestimmen, Ergebnis runden}
    if s = A[mitte] then return mitte;
    if s < A[mitte] then rechts := mitte – 1;
    if s > A[mitte] then links := mitte + 1;
return „nicht gefunden“;
```

- Beispiel



  - BinäreSuche (A, 32) liefert nach 3 Vergleichen die Antwort 6.
  - BinäreSuche (A, 41) meldet nach 4 Vergleichen „nicht gefunden“.

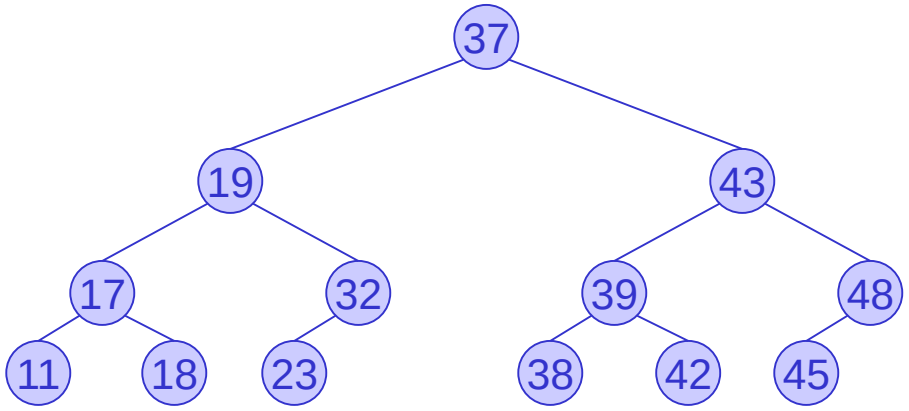
Effizienz der binären Suche

- Beispiel (n = 13 Einträge)

11	17	18	19	23	32	37	38	39	42	43	45	48

- Entscheidungsbaum

  Der Algorithmus BinäreSuche halbiert bei jedem Vergleich den verbleibenden Suchraum.



- Analyse der Laufzeit
  - Entscheidungsbaum hat Höhe $h = \lceil \log_2(n+1) \rceil - 1 = \lceil 3.8 \rceil - 1 = 3$
  - Suche benötigt also maximal $h + 1 = 4$ viele Vergleiche

- Vergleich der Suchverfahren
  - Beispiel n = 1.000
    - Sequenziell: 1.000 Vergleiche
    - Binäre Suche: 10 Vergleiche
  - Beispiel n = 1.000.000
    - Sequ.: 1.000.000 Vergleiche
    - Binäre Suche: 20 Vergleiche

Asymptotische Komplexitätsklassen

- Asymptotische Komplexitätsklassen
  - Zeigen, wie sich Laufzeiten *T(n)* für sehr große Eingaben verhalten $n \to \infty$
  - Maß für Komplexität unabhängig von konstanten Faktoren und Summanden
  - Klammern Rechnergeschwindigkeit, Aufwände für Initialisierung etc. aus

- Formal: O-Notation

$$O(f) = \{ g : \mathbb{N} \to \mathbb{R}^+ \mid \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : g(n) \leq c \cdot f(n) \}$$

- Sprechweise:

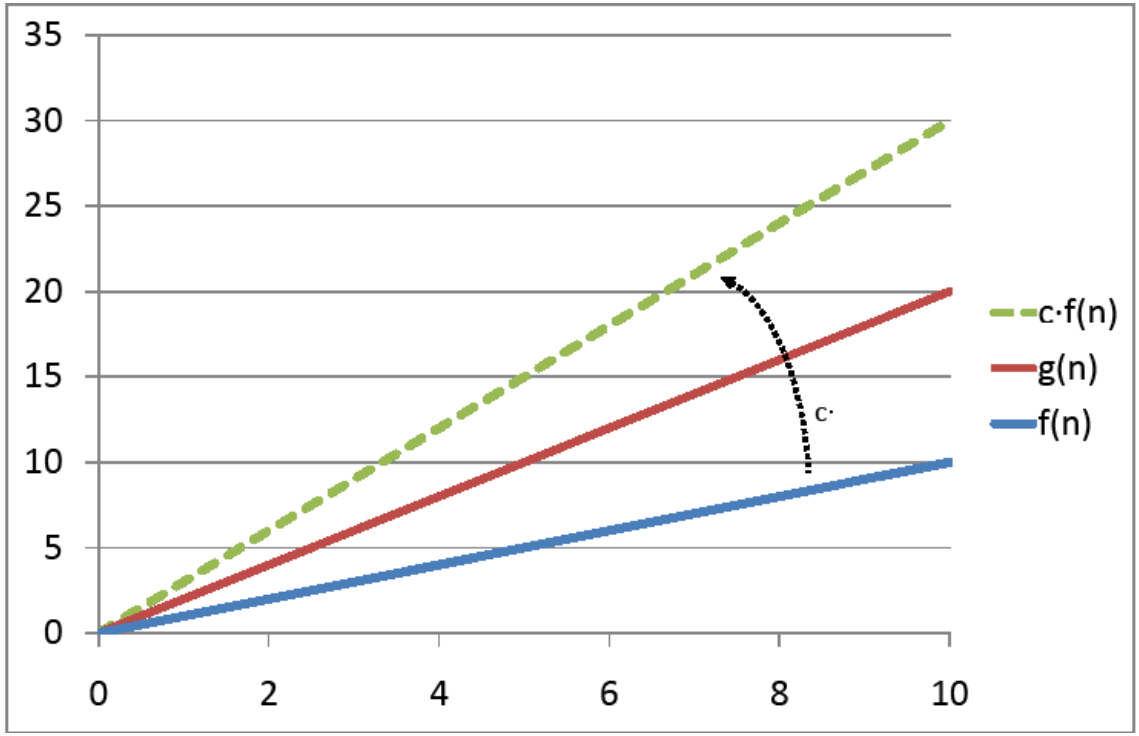
$g \in O(f)$: *f* ist obere Schranke von *g*, *g* wächst höchstens so schnell wie $O(f)$

Veranschaulichung: O-Notation (1)

$$O(f) = \{ g : \mathbb{N} \to \mathbb{R}^+ \mid \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : g(n) \leq c \cdot f(n) \}$$

- $g(n) = 2 \cdot n$, $f(n) = n \rightarrow g \in O(f)$
- $c = 3$, n_0 beliebig $\rightarrow g(n) = 2 \cdot n \leq 3 \cdot n = c \cdot f(n)$

konstante Faktoren werden vernachlässigt!



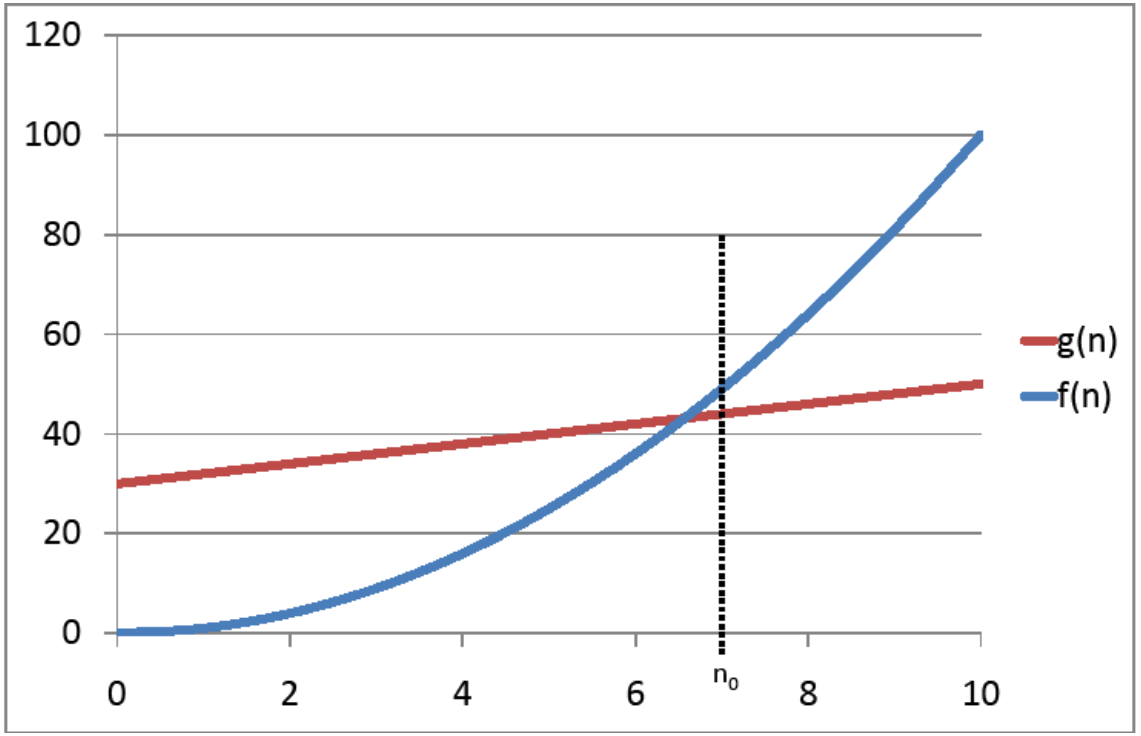
Veranschaulichung: O-Notation (2)

$$O(f) = \{ g : \mathbb{N} \to \mathbb{R}^+ \mid \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : g(n) \leq c \cdot f(n) \}$$

- $g(n) = 2 \cdot n + 30$, $f(n) = n^2 \rightarrow g \in O(f)$
- $c = 1$, $n_0 = 7 \rightarrow g(n) \leq f(n) \; \forall \; n \geq n_0 = 7$

für kleine *n*, kann „Laufzeit“ von *f(n)* „besser“ sein;

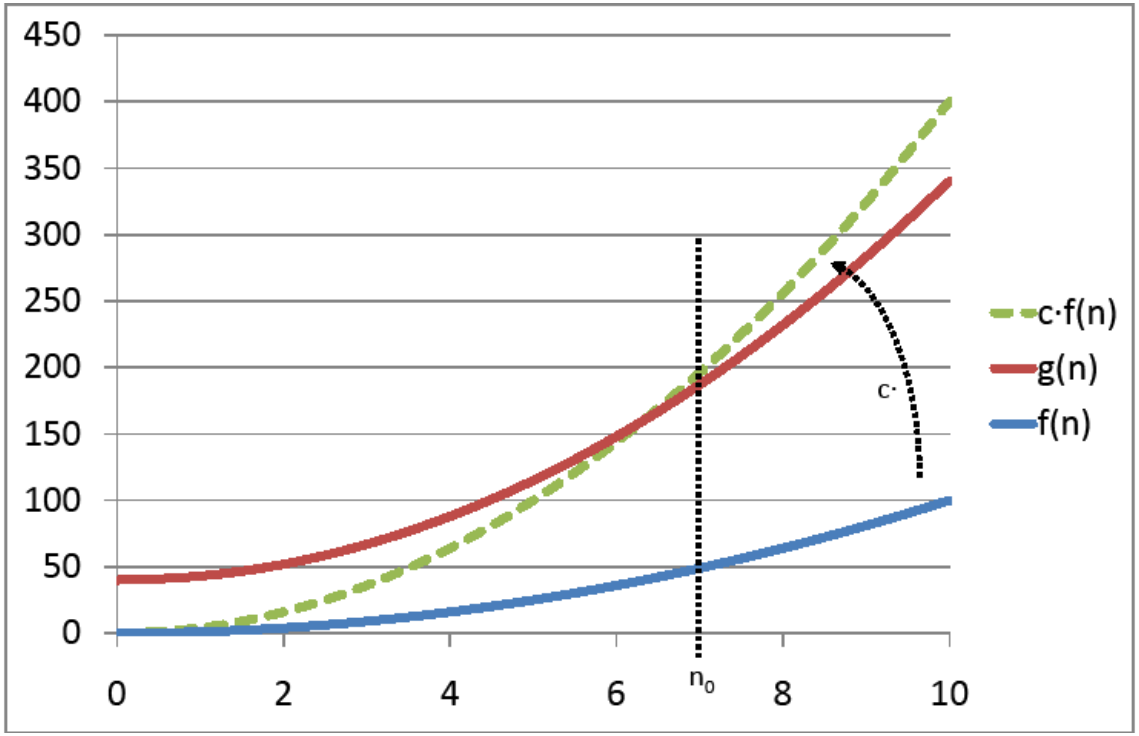
bei asymptotischer Betrachtung wächst *g(n)* jedoch langsamer!



Veranschaulichung: O-Notation (3)

$$O(f) = \{ g : \mathbb{N} \to \mathbb{R}^+ \mid \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : g(n) \leq c \cdot f(n) \}$$

- $g(n) = 3 \cdot n^2 + 40$, $f(n) = n^2 \rightarrow g \in O(f)$
- $c = 4$, $n_0 = 7 \rightarrow g(n) \leq 4 \cdot f(n) \; \forall \; n \geq n_0 = 7$



Veranschaulichung: O-Notation (4)

$$O(f) = \{ g : \mathbb{N} \to \mathbb{R}^+ \mid \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : g(n) \leq c \cdot f(n) \}$$

- Beispiel: $g(n) = 3^n$, $f(n) = 2^n$ → $g \notin O(f)$
- Zu zeigen: es gibt kein $c>0$, $n_0>0$, sodass $\forall n \geq n_0$: $3^n \leq c \cdot 2^n$
- Also: $\forall c > 0, \forall n_0 > 0, \exists n \geq n_0 : g(n) > c \cdot f(n)$
- Wir wählen ein beliebiges c > 0 und setzen an:

$$3^n > c \cdot 2^n$$

$$(\tfrac{3}{2})^n > c$$

$$\log_{3/2}\left((\tfrac{3}{2})^n\right) > \log_{3/2}(c)$$

$$n > \log_{3/2}(c)$$

- Insbesondere gilt die Ungleichung also für $n = max\{n_0, \lceil log_{3/2}(c) \rceil + 1\}$
  → $g \notin O(f)$

Komplexitätsklassen: Beispiele

- Typische Komplexitätsklassen

  Sei *n* die Länge der Eingabe (Bsp. Länge eines Array, Länge eines Strings)

Klasse	*Bezeichnung*	*Beispiel*	*n=10*	*n=100*
O(1)	konstant	Einzeloperation	1	1
O(log n)	logarithmisch	binäre Suche	4	7
O(n)	linear	sequenzielle Suche	10	100
O(n log n)		Sortieren eines Arrays	40	700
$O(n^2)$	quadratisch	Matrixmultiplikation	100	10.000
$O(n^3)$	kubisch	Invertierung einer Matrix	1.000	1.000.000
$O(n^k)$	polynomiell vom Grad *k*			
$O(2^n)$	exponentiell	Edit-Distanz naiv	1.000	1E30
O(n!)	Fakultät	Permutationen aufzählen	1E7	1E158
$O(n^n)$			1E10	1E200

ohne Betrachtung von Faktoren!

Regeln für die Laufzeitanalyse

- Elementare Anweisungen:
  - in *O(1).*
  - Wichtig: welche Anweisungen im gewählten Rechnermodell elementar? In höheren Programmiersprachen scheinbar elementare Anweisungen möglicherweise komplexe Anweisungsfolgen
  - Beispiel für Menge M, Objekt O: isElement(M,O);

- Folgen von Anweisungen:
  - Sei A, B eine Folge von Anweisungen mit $T_A \in O(f)$ und $T_B \in O(g)$
  - Dann gilt: $T = T_A + T_B \in O(max\ \{f, g\})$
  - Das bedeutet: Für Hintereinanderausführungen ist der höchste einzelne Aufwand maßgeblich

Regeln für die Laufzeitanalyse

- Schleifen:
  - Summe über die einzelnen Durchläufe
  - Falls Laufzeit Schleifenkörper unabhängig von Durchlauf:
    - $T_S \cdot d$, mit T_S Laufzeit Schleifenkörper, d Anzahl Durchläufe
  - Allgemein: Schranken für T_S, d: $T_S \in O(f)$, $d \in O(g)$, dann $T \in O(f \cdot g)$
- Bedingte Anweisungen:
  - $T = O(1) + O(f + g)$, $T_A \in O(f)$, $T_B \in O(g)$,
  - *If (*Bedingung*) then { A; } else { B; },*
    Bedingung in konstanter Zeit ausgewertet
- Methodenaufrufe:
  - Nicht-rekursiv: jede Methode kann einzeln analysiert werden, zusätzlich konstanter Overhead
  - Rekursiv: Rekursionsgleichungen (später)

Rechenregeln

Definiere Addition, Multiplikation und Maximumsbildung bildweise:

$\forall n: (f+g)(n) = f(n) + g(n)$, $(f \cdot g)(n) = f(n) \cdot g(n)$, etc.

1. Addition: $(f+g) \in O(\max\{f,g\}) = \begin{cases} O(g), & falls\ f \in O(g) \\ O(f), & falls\ g \in O(f) \end{cases}$

   Beispiel: Für $f \in O(n^2)$, $g \in O(n^3)$, $h \in O(n^2 \log n)$ gilt: $f+g+h \in O(n^3)$

2. Multiplikation: $a \in O(f) \ \wedge \ b \in O(g) \ \Rightarrow \ (a \cdot b) \in O(f \cdot g)$

3. Falls $\lim_{n\to\infty} \frac{g(n)}{f(n)}$ existiert, ist $g \in O(f)$ (Umkehrschluss gilt nicht!)

   Beispiel: Für $f(n) = n^2$ und $g(n) = 5n^2 + 100 \log n$ gilt $g \in O(f)$,

   da $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{5n^2+100 \log n}{n^2} = 5$

Weitere Landau-Symbole

- „Groß O“
  $$O(f) = \{g: \mathbb{N} \to \mathbb{R}^+ \mid \exists c > 0, \exists n_0 > 0, \forall n \geq n_0: \quad g(n) \leq c \cdot f(n)\}$$
- Omega
  $$\Omega(f) = \{g: \mathbb{N} \to \mathbb{R}^+ \mid \exists c > 0, \exists n_0 > 0, \forall n \geq n_0: \quad g(n) \geq c \cdot f(n)\}$$
- Theta
  $$\Theta(f) = \left\{g: \mathbb{N} \to \mathbb{R}^+ \;\middle|\; \exists c > 0, \exists n_0 > 0, \forall n \geq n_0: \quad \frac{1}{c} f(n) \leq g(n) \leq c \cdot f(n)\right\}$$
  Es gilt: $\Theta(f) = \Omega(f) \cap O(f)$
- „Klein O“
  $$o(f) = \{g: \mathbb{N} \to \mathbb{R}^+ \mid \exists c > 0, \exists n_0 > 0, \forall n \geq n_0: \quad g(n) < c \cdot f(n)\}$$
- „Klein Omega“
  $$\omega(f) = \{g: \mathbb{N} \to \mathbb{R}^+ \mid \exists c > 0, \exists n_0 > 0, \forall n \geq n_0: \quad 0 \leq c \cdot f(n) < g(n)\}$$

Schreib- und Sprechweisen

- Sprechweisen

  $g \in O(f)$: f ist obere Schranke von g, g wächst höchstens so schnell wie f

  $g \in \Omega(f)$: f ist untere Schranke von g, g wächst mindestens so schnell wie f

  $g \in \Theta(f)$: f ist die Wachstumsrate von g, g wächst wie f

  $g \in o(f)$: g wächst echt langsamer als f

- Schreibweisen
  - Statt $g \in O(f)$ oft $g(n) \in O(f(n))$ (Mathematisch nicht korrekt!)
  - Bsp: Binäre Suche hat Komplexität $\Theta(\log n)$

Analyse: Berechnung von Fibonacci–Zahlen

- Fibonacci, 1202 n.Chr. (ital. Mathematiker):
  - berühmte Kaninchen-Aufgabe:
    - Start: 1 Paar Kaninchen
    - Jedes Paar wirft nach 2 Monaten ein neues Kaninchenpaar
    - dann monatlich jeweils ein weiteres Paar
    - Wie viele Kaninchenpaare gibt es nach einem Jahr, wenn keines der Kaninchen vorher stirbt?
    - 1, 1, 2, 3, 5, 8, 13, 21, 34, …

- Anzahl im *n*-ten Monat lässt sich durch rekursive Funktion beschreiben:

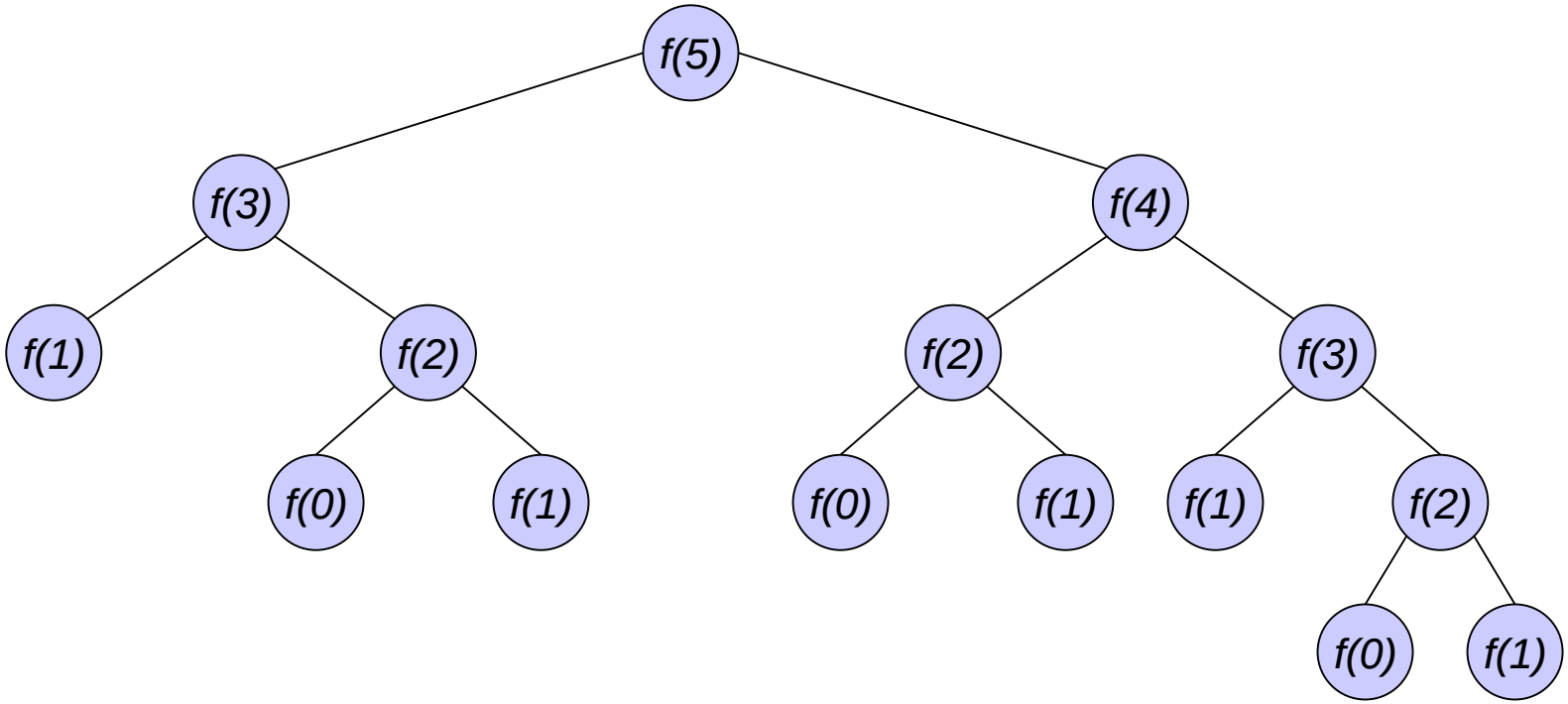
$$\mathrm{fib}(n) = \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & \text{falls } n > 1 \end{cases}$$

Naiver, rekursiver Algorithmus

- Definition eins zu eins in ein Programm
- int Fibonacci(*n*) {

```
 if (n <=1) then {                                  Terminierungsfall:
     return(n);                                        Rückgabe aktueller Wert
 } else {                                           höhere n-Werte:
     return Fibonacci(n – 1) + Fibonacci(n – 2);    Rekursive Aufrufe, Addition
 }
}                                                   Ende Fibonacci
```

Berechnungsbaum für Fibonacci



- wir zeigen später: Laufzeit $T_{rek}(n)$ liegt in $O(2^n)$ und $\Omega(2^{n/2})$
  → bessere Laufzeit möglich?
- viele Aufrufe treten mehrfach auf

Fibonacci-Zahlen endständig rekursiv

- Idee der endständigen Rekursion:
  - führe Ergebnis rekursiv mit
  - *n* zählt Schritte bis zum Ende
  - → Rekursionsaufruf ist letzte Aktion in Funktion; Rekursion ohne Nachklappern!
  - → alte Werte auf dem Stack werden nicht mehr benötigt.

```
int Fibonacci (n) {
    return fibonacci_intern (n, 0, 1);
}
int fibonacci_intern (n, result, previous) {
    if (n == 0) return result;
    else return fibonacci_intern (n – 1, result + previous, result);
}
```

- Endständige Rekursion ist Übergang zu einem iterativen Algorithmus

Iterativer Algorithmus

- Keine redundanten Berechnungen
- int Fibonacci(*n*) {

```
    result = 0, previous = 1;          Initialisiere Ergebnis-, Vorigevariable
    while (n > 0) {                    Schleife über Zahlen bis n rückwärts
        pprev = previous;              Merke Vorvoriges
        previous = result;             Merke Voriges
        result = result + pprev;       Aktuellen Wert berechnen
        -- n;                          Zähler runtersetzen
    }                                  Schleifenende; Ergebnis berechnet
    return result;                     Ergebnis zurückgeben
}                                      Ende Fibonacci
```

- Laufzeitanalyse
  - Linear: $T_{iter}(n) \in O(n)$
  - Enorme Verbesserung gegenüber (naiver) rekursiver Variante

Rekursion vs. Iteration

- Vergleich
  - Rekursive Formulierung oft eleganter
  - Iterative Lösung oft effizienter aber komplizierter
- Äquivalenz der Programmierprinzipien
  - Jede rekursive Lösung iterativ (d.h. mit Schleifen) lösbar und umgekehrt
  - Rekursion ist nicht immer schlecht: endständige Rekursion

Kapitel 1 - Grundlagen

Problemstellung
Algorithmen und Komplexität
Rekursionsgleichungen
Entwurf von Algorithmen
Datenstrukturen

Laufzeitanalyse Fibonacci - Zahlen

- Komplexität des iterativen Algorithmus: $T_{\text{iter}}(n) \in O(n)$ [offensichtlich]
  Komplexität des rekursiven Algorithmus: $T_{\text{rek}}(n) \in O(2^n)$, $T_{\text{rek}}(n) \in \Omega\left(2^{n/2}\right)$

- Sei $a > 0$ die Zeit für den Funktionsaufruf, die IF-Anweisung etc., dann ist die Laufzeit $T_{\text{rek}}(n)$ bei der rekursiven Berechnung:

$$T_{\text{rek}}(n) = \begin{cases} a & n \leq 1 \\ T_{\text{rek}}(n-1) + T_{\text{rek}}(n-2) + a & n > 1 \end{cases}$$

  → *Lösen einer Rekursionsgleichung nötig!*

- *Hinweis*
  - *Wir betrachten die Komplexität bzgl. der Laufzeit $T(n)$*
    - *kann durch effiziente Algorithmen evtl. verbessert werden*
    - *polynomieller Aufwand möglich? ja, durch iterativen Algorithmus!*
  - *Das Wachstum der Funktionswerte $\text{fib}(n)$ liegt in den selben Komplexitätsklassen*
    - *d.h. die Funktionswerte steigen exponentiell stark an*
    - *dieses Verhalten ist durch Spezifikation der Fibonacci-Zahlen vorgegeben*
    - *keine Änderung möglich*

Lösen von Rekursionsgleichungen: Sukzessives Einsetzen

– Bsp: $T(1) = 1$ und $T(n) = T(n-1) + n$ für $n > 1$

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= T(n-2) + (n-1) + n \\ &= \cdots \\ &= T(1) + 2 + \cdots + (n-2) + (n-1) + n \\ &= n \cdot (n+1)/2 \end{aligned}$$

– Bsp: $T(1) = 0$ und $T(n) = T(n/2) + n$ für $n > 1$

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + n \\ &= T\left(\frac{n}{4}\right) + \frac{n}{2} + n \\ &= T\left(\frac{n}{8}\right) + \frac{n}{4} + \frac{n}{2} + n \\ &= \cdots \\ &= T(1) + \cdots + \frac{n}{8} + \frac{n}{4} + \frac{n}{2} + n \\ &= 2 \cdot (n-1) \end{aligned}$$

Annahme: n ist 2er-Potenz

Beispiel: Fibonacci - Zahlen

- Es gilt: $T_{\text{rek}}(n)$ ist für $n > 1$ streng monoton wachsend: $T_{\text{rek}}(n) < T_{\text{rek}}(n+1)$
- Abschätzung nach oben: für alle $n > 2$ gilt:

$$\begin{aligned} T_{\text{rek}}(n) &= T_{\text{rek}}(n-1) + T_{\text{rek}}(n-2) + a \\ &< 2 \cdot T_{\text{rek}}(n-1) + a \quad < \quad 2 \cdot [2 \cdot T_{\text{rek}}(n-2) + a] + a \\ &= 4 \cdot T_{\text{rek}}(n-2) + 2 \cdot a + a \\ &< 8 \cdot T_{\text{rek}}(n-3) + 4 \cdot a + 2 \cdot a + a \\ &< \ldots \\ &< 2^n \cdot T_{\text{rek}}(n-n) + \sum_{i=1}^{n-1} 2^i \cdot a \\ &= 0 + \sum_{i=1}^{n-1} 2^i \cdot a \quad =_{(geom.Reihe)} \quad \frac{2^{n+1}-1}{2-1} \cdot a \quad \in O(2^n) \end{aligned}$$

$$2^x \cdot T_{\text{rek}}(n-x) + \sum_{i=1}^{x-1} 2^i \cdot a$$

- Abschätzung nach unten: für alle $n > 3$ gilt:

$$\begin{aligned} T_{\text{rek}}(n) &> T_{\text{rek}}(n-1) + T_{\text{rek}}(n-2) \\ &> 2 \cdot T_{\text{rek}}(n-2) \\ &> 4 \cdot T_{\text{rek}}(n-4) \\ &> \ldots \\ &> \begin{cases} 2^{n/2} \cdot T_{\text{rek}}(0), & n \text{ gerade} \\ 2^{(n-1)/2} \cdot T_{\text{rek}}(1), & n \text{ ungerade} \end{cases} \quad \in \Omega\left(2^{n/2}\right) \end{aligned}$$

$$2^x \cdot T_{\text{rek}}(n-2x)$$

Lösen von Rekursionsgleichungen: Master Theorem

- Für Rekursionsgleichungen der Form

$$T(n) = \begin{cases} c & ,n = 1 \\ a \cdot T\left(\frac{n}{b}\right) + c \cdot n & ,n > 1 \end{cases}$$

mit $a \geq 1, b > 1$ gilt:

$$T(n) = \begin{cases} O(n) & ,a < b \\ O(n \log n) & ,a = b \\ O\left(n^{\log_b a}\right) & ,a > b \end{cases}$$

- Statt $T\left(\frac{n}{b}\right)$ auch $T\left(\left\lceil\frac{n}{b}\right\rceil\right)$ oder $T\left(\left\lfloor\frac{n}{b}\right\rfloor\right)$, falls b kein Teiler von n
- Beispiel: $T(n) = T\left(\frac{n}{2}\right) + n \in O(n)$ [zuvor gezeigt: $T(n) = 2(n-1)$]

Master Theorem – Beweis

- Beweis: $T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n$

$$= a^2 \cdot T\left(\frac{n}{b^2}\right) + a \cdot c \cdot \frac{n}{b} + c \cdot n$$

$$= a^3 \cdot T\left(\frac{n}{b^3}\right) + a^2 \cdot c \cdot \frac{n}{b^2} + a \cdot c \cdot \frac{n}{b} + c \cdot n$$

$$= \ldots = c \cdot n \cdot \sum_{i=0}^{k} \left(\frac{a}{b}\right)^i \quad \text{für } n = b^k$$

- Fallunterscheidung:
  - $a < b$: $T(n) \leq c \cdot n \cdot \frac{1}{\left(1-\frac{a}{b}\right)} = c \cdot n \cdot \frac{b}{b-a} \in O(n)$
  - $a = b$: $T(n) = c \cdot n \cdot (k+1) \in O(n \cdot k) = O(n \log n)$
  - $a > b$: $T(n) = c \cdot n \cdot \left(\left(\frac{a}{b}\right)^{k+1} - 1\right) / \left(\frac{a}{b} - 1\right) = c \cdot n \cdot O\left(\left(\frac{a}{b}\right)^k\right)$

$$= c \cdot n \cdot O\left(\left(\frac{a}{b}\right)^{\log_b n}\right) = c \cdot n \cdot O\left(\frac{a^{\log_b n}}{n}\right) = O\left(a^{\log_b n}\right) = O\left(n^{\log_b a}\right)$$

Master Theorem erweitert

- Für Rekursionsgleichungen der Form

$$T(n) = \begin{cases} c & ,n = 1 \\ a \cdot T\left(\frac{n}{b}\right) + d(n) & ,n > 1 \end{cases} \quad \text{mit } d(n) \in O(n^\gamma), \gamma > 0$$

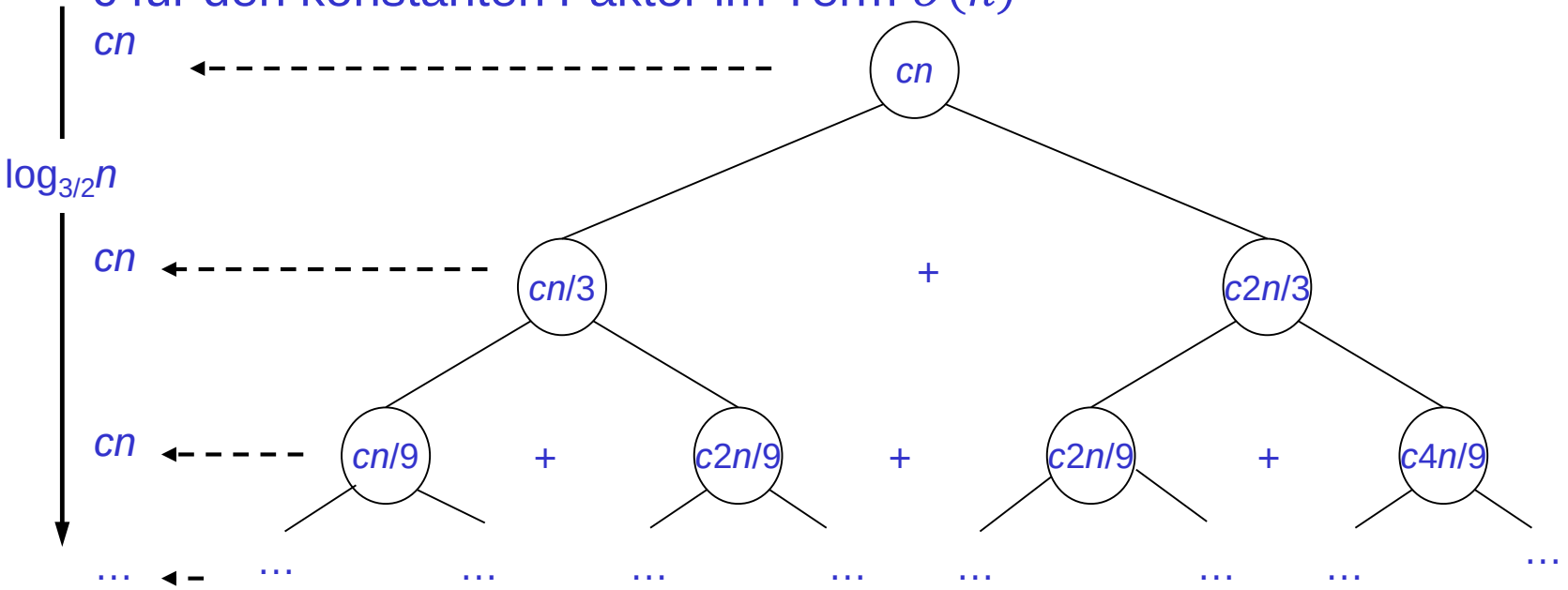
gilt:

$$T(n) = \begin{cases} O(n^\gamma) & ,a < b^\gamma \\ O(n^\gamma \log_b n) & ,a = b^\gamma \\ O\left(n^{\log_b a}\right) & ,a > b^\gamma \end{cases}$$

- Weitergehende Formen in der Literatur (vgl. Cormen et al. 73ff.)

Rekursionsgleichungen: Rekursionsbäume

- $T(n) = T(n/3) + T(2n/3) + O(n)$
- c für den konstanten Faktor im Term $O(n)$



∑: O(*n* log *n*)

- Kann mit sukzessivem Einsetzen verifiziert werden

Substitution

- Substitution: Schwierige Ausdrücke auf Bekanntes zurückführen

- Bsp: $T(n) = 2 \cdot T\left(\left\lfloor \sqrt{n} \right\rfloor\right) + \log n$
  - Substituiere $m = \log n$
  - Ergibt: $T(2^m) = 2 \cdot T\left(2^{m/2}\right) + m$
  - Setze $S(m) = T(2^m)$
  - Ergibt: $S(m) = 2 \cdot S(m/2) + m$
  - Mit „einfacher“ Lösung $S(m) = O(m \log m)$
  - Rücksubstitution:
    $T(n) = T(2^m) = S(m) = O(m \log m) = O(\log n \cdot \log\log n)$

Rekursionsbeweise

- Häufige Fehlerquelle:
  - Beispiel: Ist $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n \in O(n)$?
  - Behauptung: $T(n) \leq c \cdot n$ [und somit $\in$ O(n)]

$$
\begin{aligned}
T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + n \\
&\leq 2 \cdot c \cdot \frac{n}{2} + n \\
&= c \cdot n + n \\
&= (c+1) \cdot n \quad \in O(n)
\end{aligned}
$$

FALSCH!

  - Fehler: Behauptet wurde $T(n) \leq c \cdot n$, gezeigt aber nur $T(n) \leq (c+1) \cdot n$

→ exakter Wert der Konstanten wichtig

  - Richtige Lösung: siehe Übungsblatt

Kapitel 1 - Grundlagen

Problemstellung
Algorithmen und Komplexität
Rekursionsgleichungen
Entwurf von Algorithmen
Datenstrukturen

Entwurf von Algorithmen

- verschiedene Möglichkeiten Problem (Spezifikation) zu lösen
- starke Auswirkungen auf Effizienz
  - Suche in Mengen
    - lineare Suche $O\left(\frac{n}{2}\right)$ bzw. $O(n)$
    - binäre Suche $O(\log n)$
  - Fibonacci-Zahlen
    - naiv rekursiv $O(2^n)$
    - iterativ $O(n)$
  - Edit-Distanz (nächste Folie)
    - naiv ???
    - schlauer ???
- Bewährte Vorgehensweisen/Paradigmen, um Algorithmen zu entwerfen

  → werden im Laufe der Vorlesung vorgestellt

Edit-Distanz: Ähnlichkeit von Sequenzen

- Modell/Spezifikation:
  - $D(s, q)$ = Minimale Anzahl an Operationen, die man benötigt, um eine Sequenz s in eine Sequenz q zu umzuwandeln.
  - Zulässige Operationen: *Umbenennen* (:), *Einfügen* (↓), *Entfernen* (↑) jeweils eines einzelnen Zeichens an der aktuellen Position

- Beispiel: Umwandlung von „ACCCAGA“ in „CCCACAT“
  - Schritte: „entferne A“, „ersetze G durch C“, „füge T ein“

    ACCCAGA → CCCAGA → CCCACA → CCCACAT
  - Ergebnis: D(„ACCCAGA“, „CCCACAT“) = 3
  - Veranschaulichung durch *Alignment*

A C C C A G A	← Anfangssequenz
↑ | | | | : | ↓	← Folge von Edit-Operationen
C C C A C A T	← Zielsequenz

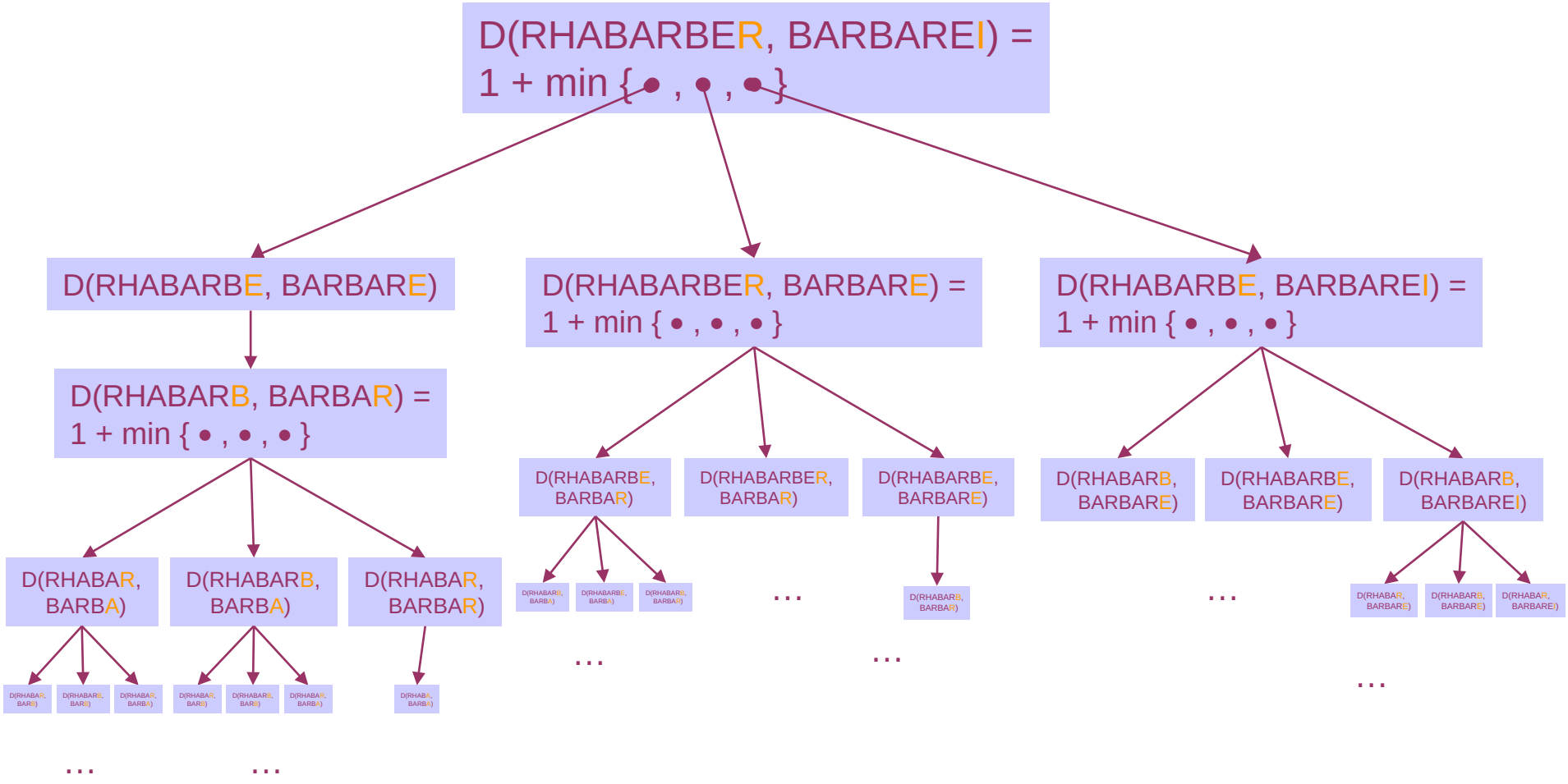
Formale Definition der Edit-Distanz

- Grundidee:
  - verkürze die Sequenzen sukzessive (3 Fälle)
  - bestimme rekursiv Distanz für die Präfixe
- Definitionen für Sequenz *t*:

Def.:	$t = t_1 \ldots t_n$	$len(t) = n$	$start(t) = t_1 \ldots t_{n-1}$	$last(t) = t_n$
Bsp.:	t = BARBAREI	$len(t) = 8$	$start(t)$ = BARBARE	$last(t)$ = I

$$D(s,q) = \begin{cases} len(q) & \text{falls } len(s) = 0 \\ len(s) & \text{falls } len(q) = 0 \\ D(start(s), start(q)) & \text{falls } last(s) = last(q) \\ 1 + \min \begin{cases} D(start(s), start(q)) \\ D(\quad s \quad , start(q)) \\ D(start(s), \quad q \quad) \end{cases} & \text{sonst} \end{cases}$$

Naive Berechnung der Edit-Distanz



$O(3^{n+m})$ Aufrufe für Sequenzlängen n, m

Beschleunigung der Berechnung

- Analyse
  - $O(3^{n+m})$ Aufrufe für Sequenzen der Längen n, m
  - Viele Aufrufe treten mehrfach auf
  - Es gibt nur $(m+1) \cdot (n+1) = O(m \cdot n)$ verschiedene Aufrufe

- Lösung
  - Speicherung der Ergebnisse aller Aufrufe: $O(m \cdot n)$ Speicherbedarf
  - Systematische Auswertung durch $O(m \cdot n)$ Operationen
  - Schema dieses Verfahrens heißt *dynamische Programmierung*

- Beschleunigung (Bsp. m,n ≈ 5, 50, 500)

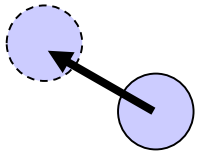
  dyn. Programmierung vs. naive Berechnung
  - $5 \cdot 5 = 25$ statt $3^{10} = 59.049$
  - $50 \cdot 50 = 2.500$ statt $3^{100} = 5.154 \cdot 10^{47}$
  - $500 \cdot 500 = 250.000$ statt $3^{1000} = 1.322 \cdot 10^{477}$

Berechnung durch dynamische Prog.

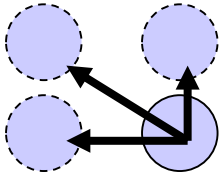
Berechnungsschema

		R	H	A	B	A	R	B	E	R
	0	1	2	3	4	5	6	7	8	9
B	1									
A	2									
R	3									
B	4									
A	5									
R	6									
E	7									
I	8									

für $len(q) = 0$:
$D(s,q) = len(s)$



falls $last(s) = last(q)$:
$D(s,q) = D(start(s), start(q))$



sonst :

$$D(s,q) = 1 + \min \begin{cases} D(start(s), start(q)) \\ D(\quad s \quad, start(q)) \\ D(start(s), \quad q \quad) \end{cases}$$

für $len(s) = 0$:
$D(s,q) = len(q)$

Ausführung → Übung

Paradigma: Dynamische Programmierung

- Optimierungsaufgabe
  - Optimiere eine Zielfunktion unter Beachtung von Nebenbedingungen
- Grundprinzip:
  - Finde optimale Lösungen für „kleine“ Elementarprobleme
  - konstruiere sukzessive „größere“ Lösungen bis Gesamtlösung
- Bottom-up-Strategie mit folgenden Schritten
  - Teilprobleme bearbeiten
  - Teilergebnisse in Tabellen eintragen
  - Zusammensetzen der Gesamtlösung
- Anwendungsbedingungen:
  - Optimale Lösung enthält optimale Teillösungen
  - Überlappende Teillösungen

Kapitel 1 - Grundlagen

Problemstellung
Algorithmen und Komplexität
Rekursionsgleichungen
Entwurf von Algorithmen
Datenstrukturen

Datenstrukturen

- Datenstrukturen
  - Organisationsformen für Daten
  - Funktionale Sicht: Containerobjekte mit Operationen, lassen sich als abstrakte Datentypen beschreiben.
  - Beinhalten Strukturbestandteile und Nutzerdaten
  - Können gleichförmig oder heterogen strukturiert sein
  - Anforderungen:
    - Statisch oder dynamisch bestimmte Größe
    - Transiente oder persistente Speicherung

- Betrachtete Beispiele
  - Sequenzen: Folgen, Listen, Warteschlangen
  - Graphen, insbesondere Bäume

Datentypen

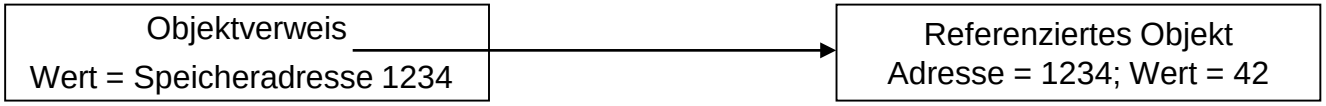
- Definition: Menge von Werten und Operationen auf diesen Werten

- Elementare (atomare) Datentypen: (Java)
  - int: ganze Zahlen 32-bit (sowie byte 8-bit, short 16-bit, long 64-bit)
  - boolean: true oder false
  - char: Zeichen
  - float: Fließkommazahlen 32-bit (sowie double 64-bit)

- Zusammengesetzte Typen:
  - Record: Datensatz (in Java nicht explizit; als Objekt o.ä.)
  - Set: Menge (in Java vordefiniert, inklusive Methoden zum Sortieren etc.)
  - Array: Reihung gleichartiger Daten

Objektverweise als Zeiger (Pointer)

- In Java nicht explizit
- Referenz auf ein anderes Objekt
- Besteht aus Speicheradresse des referenzierten Objekts

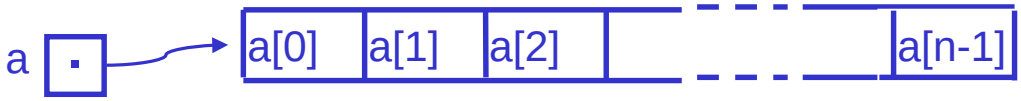


- Für dynamische Datenstrukturen: Speicher erst bei Bedarf
- In einigen Programmiersprachen explizite Speicherfreigabe
- Java hat garbage collection: falls keine Referenz mehr, wird Speicher freigegeben

Zusammengesetzte Typen: Arrays

- Array: Reihung (Feld) fester Länge von Daten gleichen Typs
  - z.B. a[i] bedeutet Zugriff auf das i-te Element eines Arrays a[]
  - Erlaubt effizienten Zugriff auf Elemente: konstanter Aufwand
  - Wichtig: Array-Grenzen beachten!!

  - Referenz-Typ: Verweis auf (Adresse der) Daten;



  - Vorsicht: Array a beginnt in Java bei 0 und geht bis a.length – 1!! (Häufige Fehlerquelle)

Beispiel: Sieb des Eratosthenes

- Sieb des Eratosthenes: (hellen. Gelehrter, ca. 276–195 v. Chr.)
  - Suche Primzahlen kleiner *n*
  - Arrayelemente effizient zugreifbar
  - Idee: a[i] = 1, falls i prim
    a[i] = 0, sonst

- Initialisiere Array-Werte bis *n* mit 1
- Setze Vielfache sukzessive auf 0
- Laufe bis zur Wurzel der Obergrenze
- Arrayeinträge 1: prim

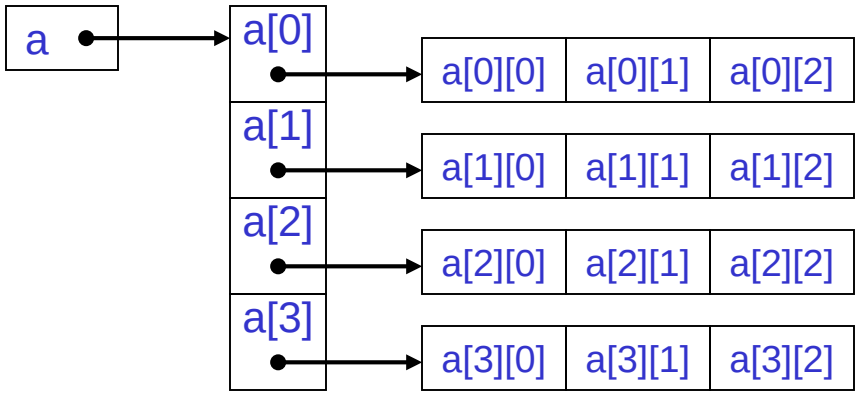
- Beispiel für 25:
  - Initialisiere a[i]=1 für i=2 bis 25
  - Durchlaufe: a[i]=0, Vielfache von 2, 3, 5;
  - 5 Wurzel von 25: fertig

i	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2			0		0		0		0		0		0		0		0		0		0		0	
3								0						0						0				
5																								0

Mehrdimensionale Arrays

- Zweidimensionale Arrays (= Matrizen) und deren Speicherung

a[0]	a[0][0]	a[0][1]	a[0][2]
a[1]	a[1][0]	a[1][1]	a[1][2]
a[2]	a[2][0]	a[2][1]	a[2][2]
a[3]	a[3][0]	a[3][1]	a[3][2]



- Deklaration

```
int [][] a = new int [4] [3];          // keine Initialisierung
int [][] m = { {1, 2, 3}, {4, 5, 6} }; // Initialisierung mit Konstanten
```

1	2	3
4	5	6

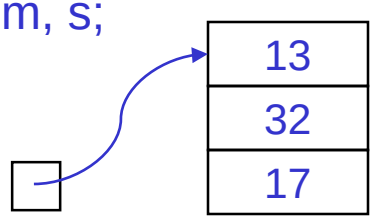
- Höhere Dimensionen

```
int [][][] q = new int [2][2][2];      // 3D: Quader
```

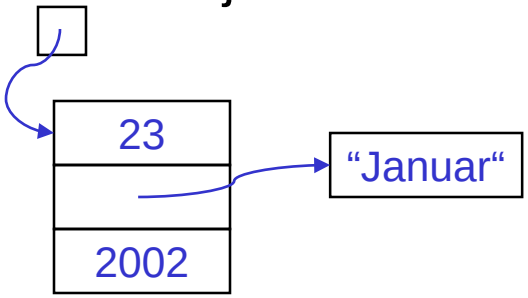
Benutzerdefinierte Datentypen: Klassen

- Zusammenfassung verschiedener Attribute zu einem Objekt

```
class Time {
   int h, m, s;
}

Time t;
```




```
class Date {
   int day;
   String month;
   int year;
}
```



- Beispiel: Rückgabe mehrerer Funktionsergebnisse auf einmal

Realisiert als Rückgabe eines einzigen komplexen Ergebnisobjekts:

```
static Time convert (int sec) {
    Time t = new Time();
    t.h = sec / 3600; t.m = (sec % 3600) / 60; t.s = sec % 60;
    return t;
}
```

Klassen vs. Arrays

Klassen

- Bestehen im allgemeinen aus *verschiedenartigen* Elementen:
  class c {String s; int i;}
- Jedes Element hat einen eigenen Namen: c.s, c.i
- Anzahl der Elemente wird *statisch* bei der Deklaration der Klasse festgelegt.

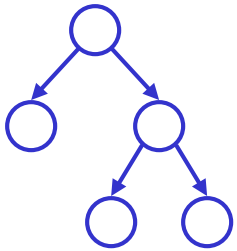
Arrays

- Bestehen immer aus mehreren *gleichartigen* Elementen: int[]
- Elemente haben keine eigenen Namen, sondern werden über Indizes angesprochen: a[i]
- Anzahl der Elemente wird *dynamisch* bei der Erzeugung des Arrays festgelegt:
  new int [*n*]

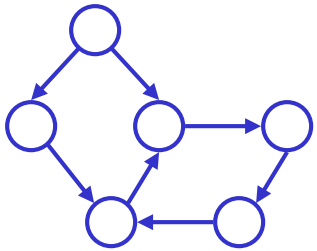
Dynamische Datenstrukturen

- Motivation
  - Länge eines Arrays nach der Erzeugung festgelegt
  - hilfreich wären unbeschränkt große Datenstrukturen
  - Lösungsidee: Verkettung einzelner Objekte zu größeren Strukturen
- Beispiele

Liste



Baum



allgem. Graph

- Charakterisierung
  - Knoten zur Laufzeit (dynamisch) erzeugt und verkettet
  - Strukturen können dynamisch wachsen und schrumpfen
  - Größe einer Struktur nur durch verfügbaren Speicherplatz beschränkt; muss nicht im vorhinein bestimmt werden.

Dynamische Datenstrukturen

- Häufig verwendet für Container-Typen (in der Regel für die Speicherung von Elementen eines bestimmten Typs)
- Beispiele: Listen, Stacks, Queues, Bäume

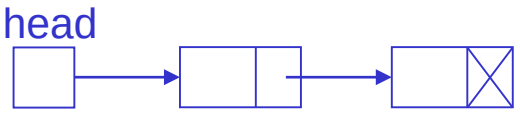
- Listen:
  - Definition: Sequenz/Folge von Elementen
  - Operationen: insert, delete, read

  - Verkettete Liste: (mit Objektverweisen)
    - Definition: Menge von Elementen, jedes Element Teil eines Knotens (node), der neben dem Inhalt (key) einen Verweis auf einen weiteren Knoten enthält

    - ein Vorteil: Umsortieren durch Neuverkettung einfach
    - ein Nachteil: Zugriff auf beliebiges Element erfordert ggf. vollständiges Durchlaufen

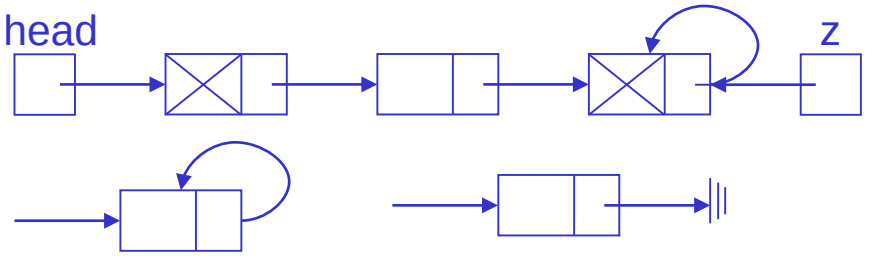
Implementierung: Verkettete Liste

```
class List () {
        Object first;
        List remainder;

}
```



– Deklaration und Initialisierung
  - List Head = null;

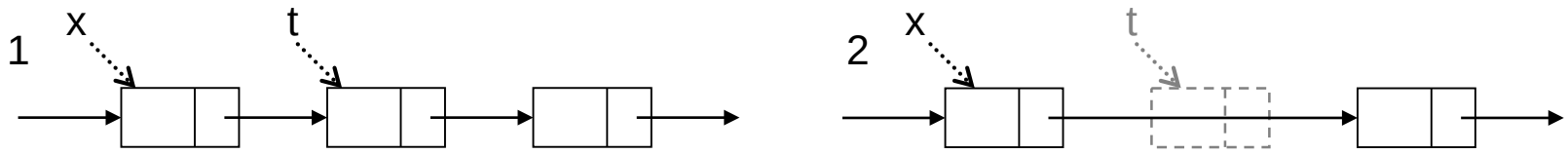
– Alternativen:
  - Anfang: zusätzlicher Leerknoten
  - 2 Knoten head, z
  - Für Listenende auch möglich:



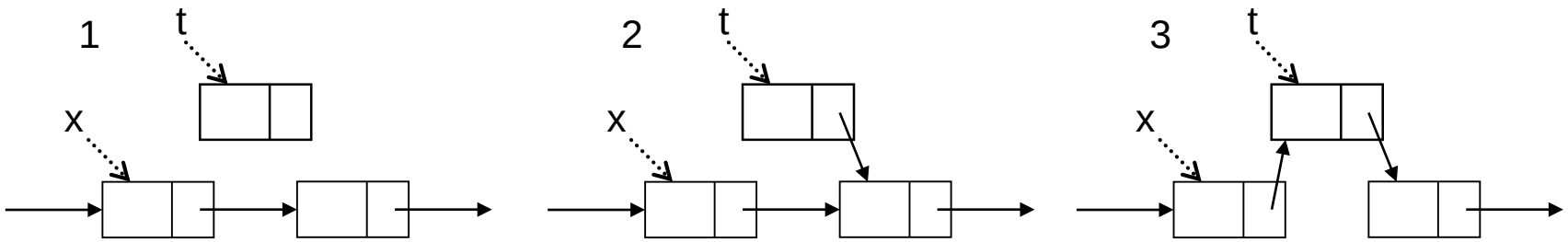
– Bei Implementierung der Operationen genaue Darstellung beachten!

Implementierung: Verkettete Liste

- Löschen: Knoten nach x (x.remainder)
  - x.remainder = x.remainder.remainder;
  - (Knoten ohne Referenz wird per garbage collection in Java entfernt, in anderen Sprachen muss selbst aufgeräumt werden!)



- Einfügen: Knoten t nach Knoten x
  - t.remainder = x.remainder; x.remainder = t;



- Achtung: Am Listenende und –anfang aufpassen!

Doppelt verkettete Liste

- Flexiblerer Zugriff (vorwärts, rückwärts)
  - Next, Prev
- Bei Änderungen: Zeiger auf das aktuelle Element; 2 Zeiger aktualisieren

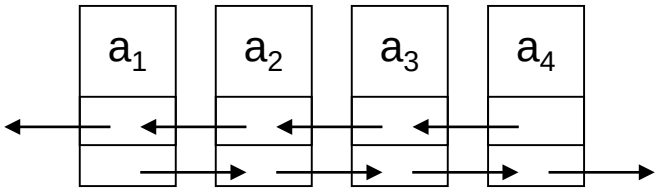
```
class List() {
    Object key;
    List prev, next;
}
```

Einfügen Knoten t nach Knoten x

```
t.prev = x;
t.next = x.next;
t.prev.next=t;
t.next.prev=t;
```

Löschen: Knoten x

```
x.prev.next = x.next;
x.next.prev = x.prev;
```



Abstrakte Datentypen

- Datenstruktur definiert durch auf ihr zugelassenen Methoden
- Spezielle Implementierung nicht betrachtet
- Definition über:
  - Menge von Objekten
  - Methoden auf diesen Objekten $\rightarrow$ Syntax des Datentyps
  - Axiome $\rightarrow$ Semantik des Datentyps
- Top-down Software-Entwurf
- Spezifikation
  - Zuerst „was“ festlegen, noch nicht „wie“
    - Spezifikation vs. Implementierung
  - Klarere Darstellung von Programmkonzepten
- Abstraktion in Java:
  - Abstract class
  - Interface

Beispiel: Algebraische Spezifikation Boolean

- Wertebereich:

  {true, false}

- Operationen:

  NOT (Zeichen $\neg$): boolean $\rightarrow$ boolean

  AND (Zeichen $\wedge$): boolean $\times$ boolean $\rightarrow$ boolean

  OR (Zeichen $\vee$): boolean $\times$ boolean $\rightarrow$ boolean

- Axiome:

  $\neg$ true = false; $\neg$false = true;

  x $\wedge$ true = x; x $\wedge$ false = false;

  x $\vee$ true = true; x $\vee$ false = x;

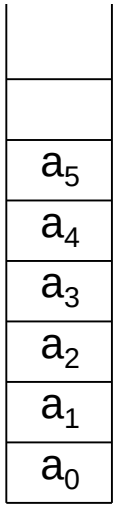
Stacks

- „Stapel“ von Elementen („Kellerspeicher“)
- Wie Liste: sequentielle Ordnung, aber nur Zugriff auf erstes Element:

```
interface Stack {
    void push (Object); // neues Element oben einfügen
    Object pop ();      // oberstes Element ausgeben und entfernen
}                       // d.h. LIFO: Last In, First Out
```

für Stack s und Object o gilt also:

s.push(o); s.pop() == o



Algebraische Spezifikation Stack

- Operationen:

StackInit:			$\rightarrow$	Stack
StackEmpty:		Stack	$\rightarrow$	Boolean
Push:	Element $\times$	Stack	$\rightarrow$	Stack
Pop:		Stack	$\rightarrow$	Element $\times$ Stack

- Axiome: Für alle Elementtyp x, Stack s gelten folgende Geleichungen:
  - Pop(Push(x,s)) = (x,s)
  - Push(Pop(s)) = s für StackEmpty(s) = FALSE
  - StackEmpty(StackInit) = TRUE
  - StackEmpty(Push(x,s)) = FALSE

- Undefinierte Operationen erfordern Fehlerbehandlung
  - Beispiel: Pop (StackInit)

Implementierung mit Pointern

```
class StackP implements Stack {
    Node head;

    StackP () {
        head = null;
    }

    void Push (Object t) {
        x = new Node();
        x.key = t;
        x.next = head;
        head = x;
    }

    Object Pop () {
        if(head == null) {
            // Fehlerbehandlung
        } else {
            x = head;
            head = x.next;
            return x.key;
        }
    }

    boolean isEmpty () {
        return (head == null);
    }

} // class StackP
```

Implementierung mit Arrays

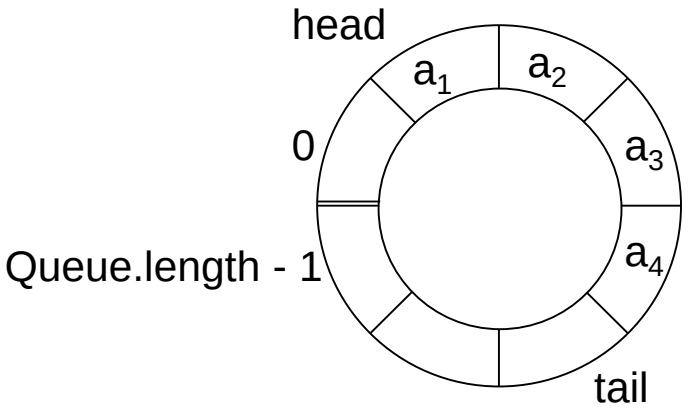
```
Class StackA implements Stack {
    int top;
    Object[] stack;

    StackA (int capacity) {
        top = 0;
        stack = new Object[capacity]
    }

    void Push (Object v) {
      if (top >= stack.length) {
        // Fehlerbehandlung Überlauf
      } else {
        stack [top] = v;
        top = top + 1;
      }
    }

    Object Pop () {
        if (top == 0) {
          // Fehlerbehandlung Unterlauf
        } else {
          top = top – 1;
          return stack [top];
        }
    }

    boolean IsEmpty () {
        return (top == 0);
    }

    boolean IsFull () {
        return (top >= stack.length);
    }

} // class StackA
```

Queues

- Spezifikation
  - Wie Liste: sequentielle Ordnung, aber:
  - Einfügen: neues Element am Ende anhängen (put)
  - Auslesen: vorderstes Element zurückgeben (get)
  - FIFO (First in, first out)
  - In Java:

    interface Queue { void put (Object); Object get(); }

- Implementierung als zyklisches Array:
  - \+ kein Speicher für Pointer nötig
  - – leere Elemente (Speicherplatzverlust)
  - – Beschränkte Länge



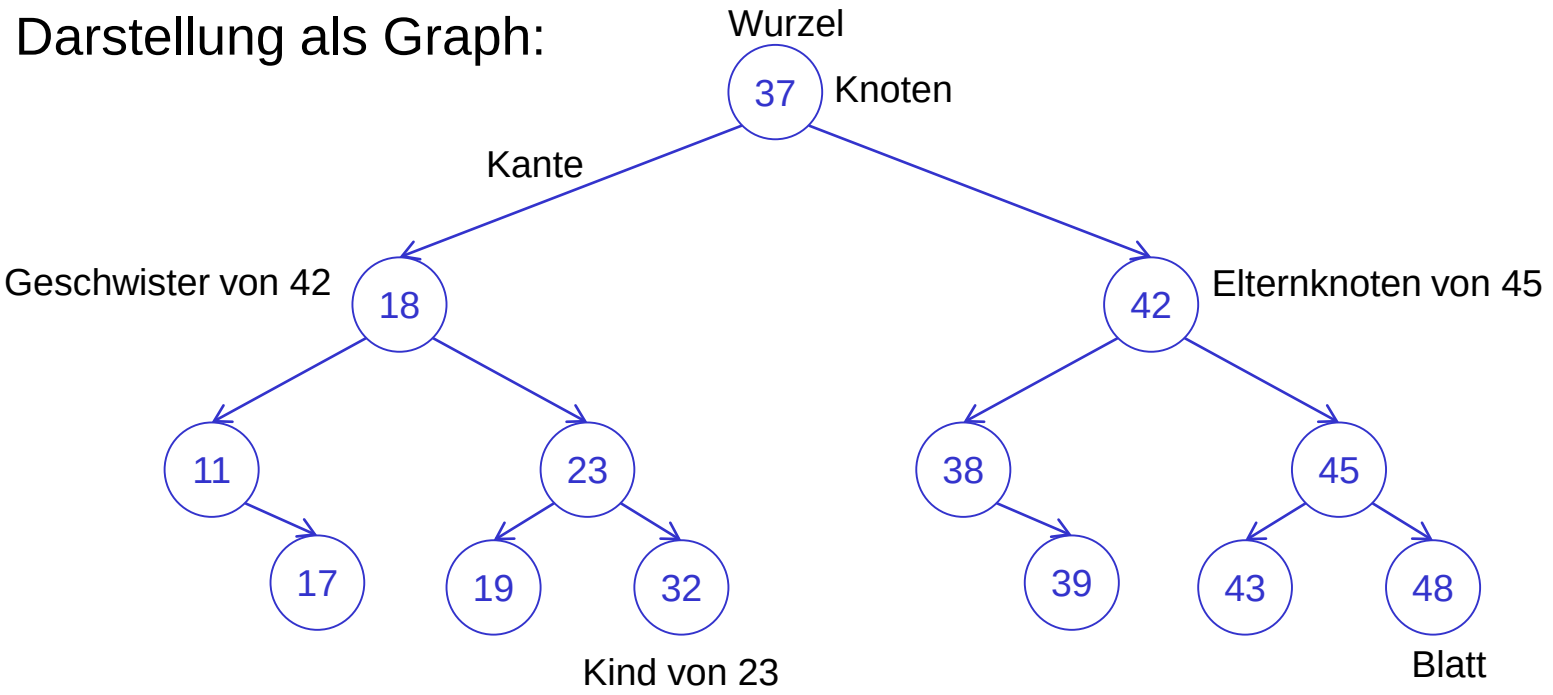
Implementierung mit Array

```
class QueueA implements Queue {
    int head, tail;
    Object [] queue;

    QueueA (int capacity) {
        head = 0; tail = 0;
        queue = new Object[capacity+1];
    }

    void Put (Object v) {
        if ((tail + 1) % queue.length == head) {
            // Fehlerbehandlung Überlauf
        } else {
            queue[tail] = v;
            tail = (tail + 1) % queue.length;
        }
    }

    Object Get () {
        if (head == tail) {
            // Fehlerbehandlung Unterlauf
        } else {
            Object t = queue[head];
            head = (head + 1) % queue.length;
            return t;
        }
    }

    boolean IsEmpty () {
        return (head == tail);
    }

    boolean IsFull () {
        return (head == (tail + 1) %
        queue.length);
    }

} // class QueueA
```

Bäume

- Menge von Knoten
- Relation, die Hierarchie definiert:
  - Jeder Knoten, außer Wurzel, hat Elternknoten (unmittelbar vorangehender Knoten)
  - Kante drückt Beziehung zwischen zwei direkt aufeinander folgenden Knoten aus

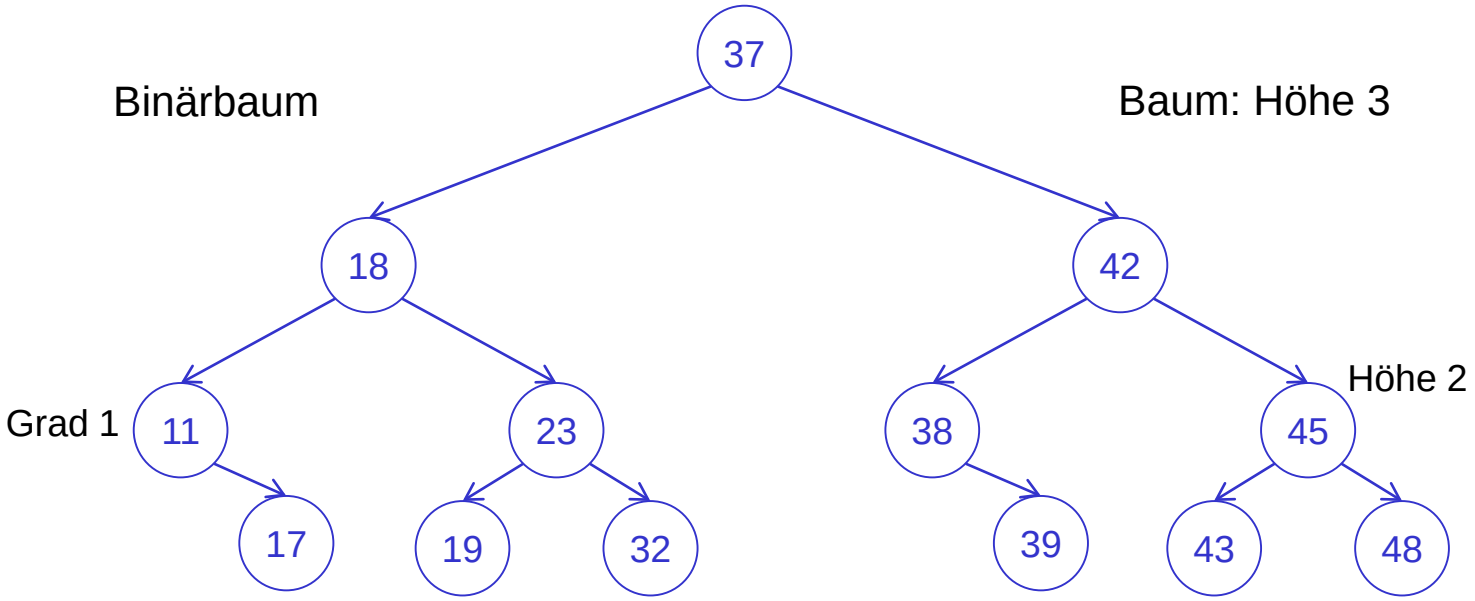
- Darstellung als Graph:



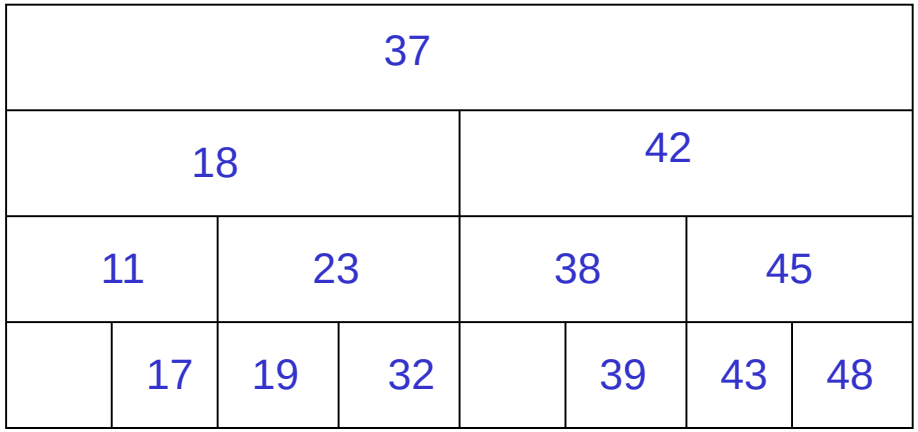
Eigenschaften von Bäumen & Knoten

- Pfad: Folge von Knoten, die jeweils direkte Nachfolger voneinander sind
- Pfadlänge: Anzahl der Kanten eines Pfades
- Höhe eines Knotens: Länge des Pfades von der Wurzel
- Höhe eines Baums: Maximale Pfadlänge im Baum

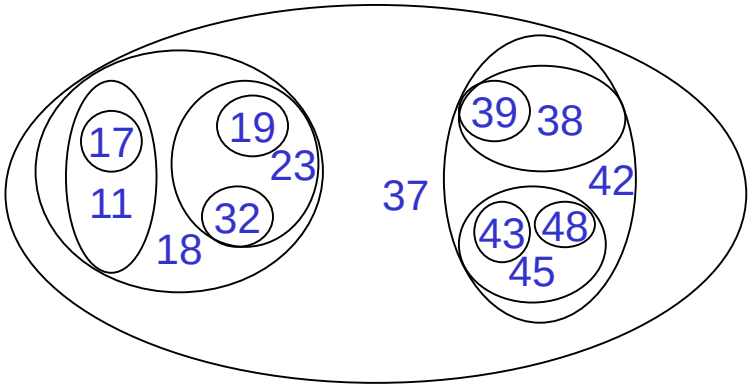
- Grad eines Knotens: Zahl der unmittelbaren Nachfolger
- Grad eines Baums: Maximale Gradzahl im Baum
- Binärbaum: Baum mit Grad 2



Alternative Baumdarstellungen



$$37\left(18\left(11(17),23(19,32)\right),42\left(38(39),45(43,48)\right)\right)$$



Baum

- Baumdefinition induktiv:
  - Ein einzelner Knoten ist ein Baum; gleichzeitig Wurzel
  - Sei n Knoten, $T_1, T_2, \ldots, T_k$ Bäume mit Wurzelknoten $n_1, n_2, \ldots, n_k$
    - n_i Kindknoten von n ergibt Baum; T_i dann i-te Teilbäume

- Maximaler Baum:
  - Baum der Höhe h vom Grad d hat maximale Knotenzahl $N(h, d)$
  - 1. Ebene: 1 Knoten (Wurzel)
  - 2. Ebene: d Knoten (d direkte Nachfolger möglich)
  - 3. Ebene: d^2 Knoten (d Mal d Nachfolger)
  - …
  - Insgesamt: geometrische Reihe

$$N(h, d) = 1 + d + d^2 + \cdots + d^h = \frac{d^{h+1} - 1}{d - 1}$$

Maximale und minimale Baumhöhe

- Baum vom Grad $d \geq 2$, n Knoten:
  - Maximale Höhe = $n-1$
  - Minimale Höhe = $\lceil \log_d(n(d - 1) + 1) \rceil - 1 \leq \lceil \log_d n \rceil$

- Beweis:
  - Maximale Höhe: n Knoten untereinander
  - Minimale Höhe: da Knotenzahl und Grad fest, folgt maximaler Baum
    - nutze N(h,d) von zuvor
    - finde kleinstes h, so dass n ≤ N(h,d)

      → N(h-1,d) < n ≤ N(h,d)

Fortsetzung Beweis

$$N(h-1,d) < n \leq N(h,d)$$

$$\frac{d^h - 1}{d-1} < n \leq \frac{d^{h+1}-1}{d-1}$$

$$d^h < n(d-1)+1 \leq d^{h+1}$$

$$h < \log_d(n(d-1)+1) \leq h+1$$

und wegen $n(d-1)+1 < nd \quad \forall n > 1$ folgt :

$$h = \lceil \log_d(n(d-1)+1) \rceil - 1$$

$$\leq \lceil \log_d(nd) \rceil - 1$$

$$= \lceil \log_d n \rceil$$

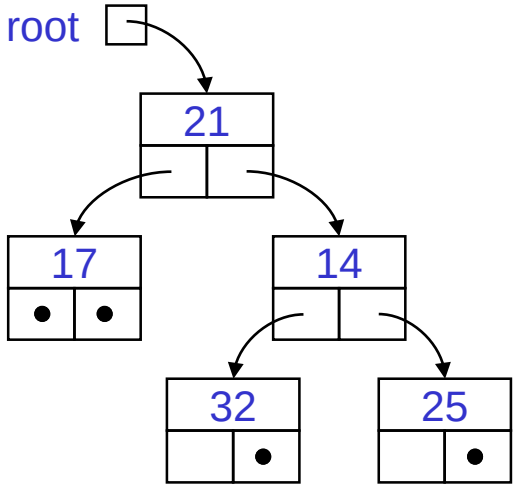
Für den Binärbaum $(d = 2)$:

$$h = \lceil \mathrm{ld}(n+1) \rceil - 1 \leq \lceil \mathrm{ld}\, n \rceil$$

Implementierung von Binärbäumen

```
class Tree {
    Object key;
    Tree left; Tree right;
}
```

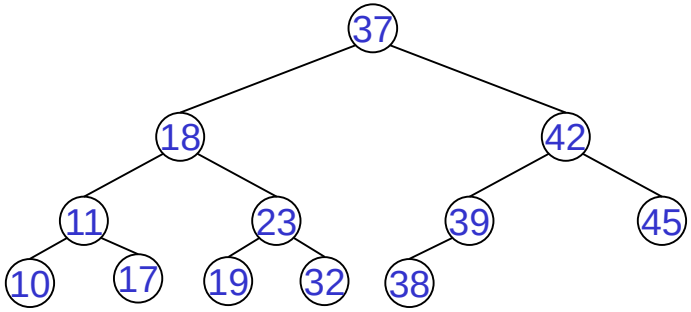
- Verankerung:
  - Baum ist Tree-Objekt:
    - Z.B. Variable Tree root;
  - Leere Teilbäume markiert durch null



Array-Einbettung

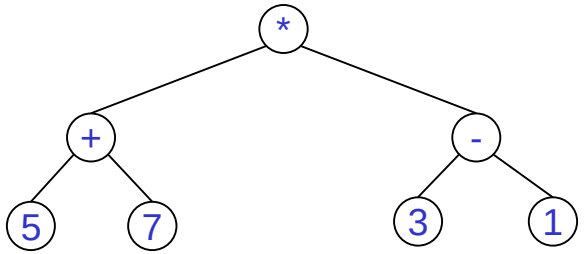
- Am besten für vollständige Bäume:
  - Alle Ebenen bis auf die letzte sind voll
  - letzte Ebene ist von links nach rechts gefüllt
- Als Array:
  - Elternknoten zu Knoten n an Position $\lfloor n / 2 \rfloor$
  - Kindknoten zu Knoten n an Positionen $2n$ und $2n + 1$
    - Existenz von Nachfolgern über Arraygrenzen abzufragen

- Beispiel:



37	18	42	11	23	39	45	10	17	19	32	38

Baumdurchläufe



- Tiefendurchlauf (depth first):
  - Durchlaufe zu jedem Knoten rekursiv die Teilbäume von links nach rechts
  - Preorder/Präfix: notiere erst einen Knoten, dann seine Teilbäume
    - Beispiel: * + 5 7 - 3 1 [polnische Notation]
  - Postorder/Postfix: notiere erst Teilbäume eines Knotens, dann ihn selbst
    - Beispiel: 5 7 + 3 1 - *
  - Inorder/Infix: notiere 1. Teilbaum, dann Knoten selbst, dann restliche Teilbäume
    - Beispiel: 5 + 7 * 3 - 1 [Mehrdeutigkeit möglich]
- Breitendurchlauf (breadth first):
  - Knoten ebenenweise durchlaufen, von links nach rechts
    - Beispiel: * + - 5 7 3 1
- Alle Durchläufe auf beliebigen Bäumen durchführbar
  - Inorder-Notation nur auf Binärbäumen gebräuchlich

Rekursiver Durchlauf für Binärbäume

```
Preorder (node) {
    if (node != null) {
        visit (node);
        Preorder (Node.left);
        Preorder (Node.right);
    }
}
Postorder (node) {
    if (node != null) {
        Postorder (Node.left);
        Postorder (Node.right);
        visit (node);
    }
}
```

```
Inorder (node) {
    if (node != null) {
        Inorder (Node.left);
        visit (node);
        Inorder (Node.right);
    }
}
```

Nicht-rekursiver Durchlauf für Binärbäume

```
PreorderStack (node) {
    Stack.Push (node);
    while (StackNotEmpty) {
        currentNode = Stack.Pop();
        if (currentNode != null) {
            visit (currentNode);
            Stack.Push (currentNode.right);
            Stack.Push (currentNode.left);
        }
    }
}
```

```
BreadthQueue (node) {
    Queue.Put (node);
    while (QueueNotEmpty) {
        currentNode = Queue.Get();
        if (currentNode != null) {
            visit (currentNode);
            Queue.Put (currentNode.left);
            Queue.Put (currentNode.right);
        }
    }
}
```