Algorithmen & Datenstrukturen

Einführung

Was ist ein Algorithmus?

- Verfahren zur Lösung eines Problems, unabhängig von Implementierung in konkreter Programmiersprache.
- Vorteil: Konzentrieren auf das Problem, nicht auf die Eigenarten der Sprache.

Gemeinsame Konzepte der Programmiersprachen:

- Wertzuweisungen/Ausdrücke (in C: `y = 2*x + c`)
- Bedingte Anweisung (in C: `if/else, switch/case`)
- Iterative Anweisung (in C: `for, while, do/while`)
- Ein- und Ausgabeanweisung (in C: `scanf, printf`)
- Prozedur-/Funktionsaufrufe

Einführung (2)

Literatur:

- J. Loeckx, K. Mehlhorn, R. Wilhelm: Grundlagen der Programmiersprachen. B.G. Teubner Verlag.
- K.C. Louden: Programming Languages. PWS Publishing Company.
- C. Ghezzie, M. Jazayeri: Konzepte der Programmiersprachen. Oldenbourg Verlag.
- H.-P. Gumm, M. Sommer: Einführung in die Informatik. Oldenbourg Verlag.

Einführung (3)

Welche Eigenschaften interessieren uns?

- **Korrektheit** (E. Dijkstra: Testen kann die Anwesenheit, aber nicht die Abwesenheit von Fehlern zeigen.)
- **Laufzeit** (wie schnell wird das Problem gelöst?)
- **Speicherplatz** (wird zusätzlicher Speicher benötigt?)

Weitere interessante Eigenschaften:

- **Kommunikationszeit** (parallele/verteilte Algorithmen)
- **Güte** (exakte oder approximative Lösung?)

Einführung (4)

Wichtiger als Performanz?

- Wartbarkeit/Erweiterbarkeit
- Entwicklungszeit/Einfachheit
- Zuverlässigkeit/Ausfallsicherheit
- Bedienbarkeit

$\Rightarrow$ *Software-Engineering* (Prof. Dr. Beims)

Einführung (5)

Wenn ein Problem mit verschiedenen Algorithmen gelöst werden kann:

- Wie bewerten/vergleichen wir Algorithmen?
- Wann ist ein Algorithmus besser als ein anderer?
- Was sind gute/schlechte Algorithmen?

$\Rightarrow$ Laufzeit messen und vergleichen **???**

Probleme beim Messen und Vergleichen der Laufzeit:

- unterschiedlich schnelle Hardware/gute Compiler
- unterschiedliche Betriebssysteme/Laufzeitumgebungen
- unterschiedliche Eingabedarstellungen/Datenstrukturen

Einführung (6)

Systemumgebung 1:

Hardware:	Pentium III mobile, 1GHz, 256 MB
Linux:	Kernel 2.4.10, gcc 2.95.3
Windows:	XP Home (SP1), Borland C++ 5.02

Systemumgebung 2:

Hardware:	Pentium M (Centrino), 1,5GHz, 512 MB
Linux:	Kernel 2.6.4, gcc 3.3.3
Windows:	XP Home (SP2), Borland C++ 5.5.1

Einführung (7)

Messergebnisse: Zahlen sortieren

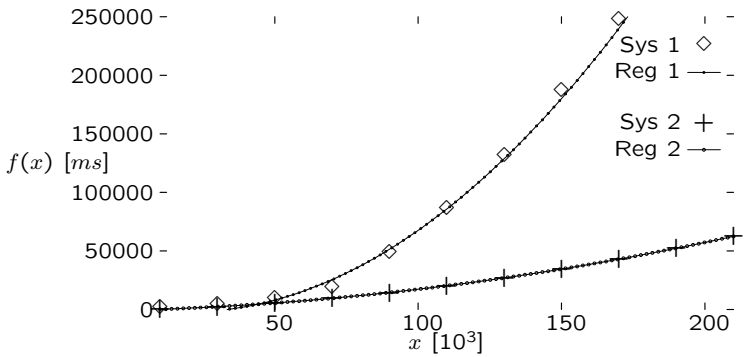
input size	System 1		System 2	
	Linux	XP [s]	Linux	XP [s]
8192	1	0	1	0
16384	3	2	2	1
32768	9	4	6	3
65536	34	17	21	9
131072	221	137	72	29

Problem: Bewertung der Messergebnisse

- Unterschiede aufgrund Compiler oder Betriebssystem?
- Skalierung: doppelte Eingabegröße, vierfache Laufzeit?

Einführung (8)

Laufzeit beschreibbar durch Polynom $ax^2 + bx + c$ **???**



Reg.1:	$10.6 \cdot x^2$	-	$399.4 \cdot x$	+	1507.3
Reg.2:	$1.1 \cdot x^2$	+	$69.9 \cdot x$	-	648.1

Einführung (9)

Minimieren der Summe der Fehlerquadrate:

$$F(x) := \sum_i (y_i - (ax_i^2 + bx_i + c))^2 \longrightarrow \min$$

Koeffizienten bestimmen durch quadratische Regression:

$$\frac{\partial F}{\partial a} = 0 \quad \frac{\partial F}{\partial b} = 0 \quad \frac{\partial F}{\partial c} = 0$$

$\Rightarrow$ es ist ein Gleichungssystem mit drei Gleichungen und drei Unbekannten zu lösen

Problem: Koeffizienten sind abhängig von Hard-/Software

Lösung: Korrekturfaktoren

Einführung (10)

Messen der Laufzeit:

- Implementieren in einer konkreten Sprache/Compiler.
- Festgelegt bei Ausführung: Rechner/Eingabemenge.

Probleme:

- Festlegen auf Norm nur schwer möglich.
- Ergebnisse lassen sich nur schwer übertragen.
- Aussage über Skalierung basiert auf Vermutung.
- Speicherbegrenzung: Paging/Swaping, Cache-Effekte

$\Rightarrow$ **Messen und Vergleichen der Laufzeiten ist nicht praktikabel oder sinnvoll!**

Bewertung von Algorithmen

Auswege:

- **idealisiertes Model** (RAM: Random Access Machine)
  - * Festgelegter Befehlssatz (Assembler-ähnlich)
  - * abzählbar unendlich viele Speicherzellen
  - $\Rightarrow$ Laufzeit: Anzahl ausgeführter RAM-Befehle
  - $\Rightarrow$ Speicherbedarf: Anzahl benötigter Speicherzellen
- **charakteristische Parameter ermitteln**
  - * Sortieren: Schlüsselvergleiche, Vertauschungen
  - * Arithmetik: Additionen, Multiplikationen

Mehr zu RAM, Turing-Maschinen und Komplexitätstheorie in der Vorlesung *Theoretische Informatik* (Prof. Dr. Dalitz)

Bewertung von Algorithmen (2)

Laufzeit und Speicherbedarf sind in der Regel abhängig von der Größe der Eingabe.

Warum ist die Komplexität der Algorithmen interessant?

Beispiel: Traveling Salesperson Problem

Gegeben: Eine Menge von Orten, die untereinander durch Wege verbunden sind. Die Wege sind unterschiedlich lang.
Gesucht: Eine Rundreise, die durch alle Orte genau einmal führt und unter allen Rundreisen minimale Länge hat. (Tourenplanung)

Bester bekannter Algorithmus: Laufzeit $\approx 2^n$ bei n Orten.

Bewertung von Algorithmen (3)

Annahme: Rechengeschwindigkeit beträgt 1GOp/s:

- 1.000.000.000 Schritte pro Sekunde
- 3.600.000.000.000 Schritte pro Stunde
- 86.400.000.000.000 Schritte pro Tag

$\Rightarrow$ Lösbare Problemgröße

- am Tag: 46 Städte
- im Jahr: 55 Städte
- in 100 Jahren: 61 Städte

in Deutschland: $\approx$ 5000 Städte

Bewertung von Algorithmen (4)

Frage: Löst ein schnellerer Rechner das Problem???
Antwort: **Nein!**

Projektion: Geschwindigkeit verdoppelte sich alle 1,5 Jahre.

- in 10 Jahren: Lösbare Problemgröße am Tag: 52 Orte
- in 100 Jahren: Lösbare Problemgröße am Tag: 112 Orte

Computer	Dauer	Anzahl Schritte
A	1 Tag	2^n
B	1 Tag	$2 \cdot 2^n = 2^{n+1}$

Nur theoretische Lösung, denn: Die Touren müssen jetzt geplant werden, nicht in 100 Jahren.

Bewertung von Algorithmen (5)

Lösung 1: bessere Algorithmen. **Speed is fun!**

Laufzeit	Dauer für 5000 Städte	#Städte pro Tag
$\approx n^2$	1 Sekunde	9.295.160
$\approx n^3$	2 Minuten	44.208
$\approx n^4$	7 Tage	3.048
$\approx n^5$	99 Jahre	612

Frage: Gibt es bessere Algorithmen für das Problem???

Bewertung von Algorithmen (6)

Lösung 2: parallele/verteilte Systeme
Problem auf mehreren Prozessoren/Computern gleichzeitig bearbeiten
$\Rightarrow$ *Parallele/Verteilte Systeme* (Prof. Dr. Ueberholz)

Lösbare Problemgröße am Tag bei linearem Speedup

#Rechner	TSP $\approx 2^n$	Matrixmultiplikation $\approx n^3$
1	46	44.208
100	53	205.197
1.000	56	442.083
10.000	59	952.440

Asymptotische Komplexität

Genaue Angabe der Komplexitätsfunktion ist oft schwierig oder unmöglich.

Unscharfe Aussagen: Abstrahieren von

- additiven Konstanten,
- konstanten Faktoren und
- Termen niedrigerer Ordnung.

Beispiel:

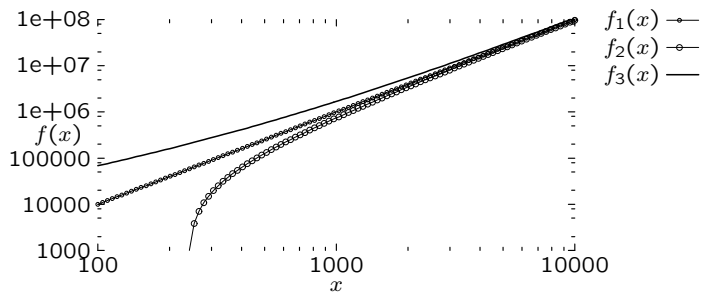
$$\begin{aligned} & 7 \cdot \log_2(n) + 5 \cdot n^3 + 3 \cdot n^4 \\ \leq\ & 7 \cdot n^4 + 5 \cdot n^4 + 3 \cdot n^4 \\ \leq\ & 15 \cdot n^4 \\ \approx\ & n^4 \end{aligned}$$

Asymptotische Komplexität (2)

Dürfen Terme niedriger Ordnung vernachlässigt werden???

Betrachten wir dazu die folgenden Funktionen:

$$\begin{aligned} f_1(x) &= x^2 \\ f_2(x) &= x^2 - 17 \cdot x \cdot \log_2(x) - 139 \cdot x - 1378 \\ f_3(x) &= x^2 + 64 \cdot x \cdot \log_2(x) + 241 \cdot x + 4711 \end{aligned}$$



Asymptotische Komplexität (3)

Anmerkungen:

- Bei kleinen Eingabegrößen sind die konstanten Faktoren und Terme niedriger Ordnung entscheidend.
- Große Konstanten: theoretisch gute Lösungen sind oft praktisch nicht akzeptabel.
- **Hard-/Software-abhängige Konstanten fallen nicht mehr ins Gewicht.**

Aufwandsklassen

Definition: Sei $g : \mathbb{N}_0 \to \mathbb{N}_0$ eine Funktion. Dann bezeichnet $O(g)$ die Menge der Funktionen, die asymptotisch höchstens so stark wachsen wie g. $\to g$ ist obere Schranke!

$$O(g) = \{f : \mathbb{N}_0 \to \mathbb{N}_0 \mid \exists c \in \mathbb{N} : \lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c\}$$

Wir schreiben:	$O(n)$	für	$O(g)$	falls $g(n) = n$
	$O(n^k)$	für	$O(g)$	falls $g(n) = n^k$
	$O(\log(n))$	für	$O(g)$	falls $g(n) = \log(n)$
	$O(\sqrt{n})$	für	$O(g)$	falls $g(n) = \sqrt{n}$
	$O(2^n)$	für	$O(g)$	falls $g(n) = 2^n$

Aufwandsklassen (2)

Beispiele:

- $2 \cdot n^2 + 37 \cdot n^3 \in O(n^3)$
- $2 \cdot n^2 + 37 \cdot n^3 \in O(n^4)$
- $2 \cdot n^2 + 37 \cdot n^3 \notin O(n^2)$
- $42 \cdot n \cdot \log(n) + 3 \cdot n^2 \in O(n^2)$
- $17 \cdot \sqrt{n} + 139 \cdot n \in O(n)$
- $17 \cdot \sqrt{n} + 139 \cdot n \notin O(\log(n))$
- $928 \cdot n^4 + 0.7 \cdot 2^n \in O(2^n)$
- $c_1 \cdot n + c_2 \cdot n^2 + \ldots + c_k \cdot n^k \in O(n^k)$ für $c_1, \ldots, c_k$ konstant

Aufwandsklassen (3)

Definition: Sei $g : \mathbb{N}_0 \to \mathbb{N}_0$ eine Funktion. Dann bezeichnet $\Omega(g)$ die Menge der Funktionen, die asymptotisch mindestens so stark wachsen wie g. $\to g$ ist untere Schranke!

$$\Omega(g) = \{f : \mathbb{N}_0 \to \mathbb{N}_0 \mid \exists c \in \mathbb{N} : \lim_{n\to\infty} \frac{g(n)}{f(n)} \leq c\}$$

Beispiele:

- $2 \cdot n^2 + 37 \cdot n^3 \in \Omega(n^3)$
- $2 \cdot n^2 + 37 \cdot n^3 \in \Omega(n^2)$
- $2 \cdot n^2 + 37 \cdot n^3 \notin \Omega(n^4)$
- $42 \cdot n \cdot \log(n) + 3 \cdot n^2 \in \Omega(n)$
- $17 \cdot \sqrt{n} + 139 \cdot n \in \Omega(n)$
- $c_1 \cdot n + c_2 \cdot n^2 + \ldots + c_k \cdot n^k \in \Omega(n^k)$ für $c_1, \ldots, c_k$ konstant

Aufwandsklassen (4)

Definition: Sei $g : \mathbb{N}_0 \to \mathbb{N}_0$ eine Funktion. Dann bezeichnet $\Theta(g)$ die Menge der Funktionen, die asymptotisch genauso stark wie g wachsen.

$$\Theta(g) = \{f : \mathbb{N}_0 \to \mathbb{N}_0 \mid f \in O(g) \wedge f \in \Omega(g)\}$$

Beispiele:

- $2 \cdot n^2 + 37 \cdot n^3 \in \Theta(n^3)$
- $42 \cdot n \cdot \log(n) + 3 \cdot n^2 \in \Theta(n^2)$
- $17 \cdot \sqrt{n} + 139 \cdot n \in \Theta(n)$
- $928 \cdot n^4 + 0.7 \cdot 2^n \in \Theta(2^n)$
- $c_1 \cdot n + c_2 \cdot n^2 + \ldots + c_k \cdot n^k \in \Theta(n^k)$ für $c_1, \ldots, c_k$ konstant

Aufwandsklassen (5)

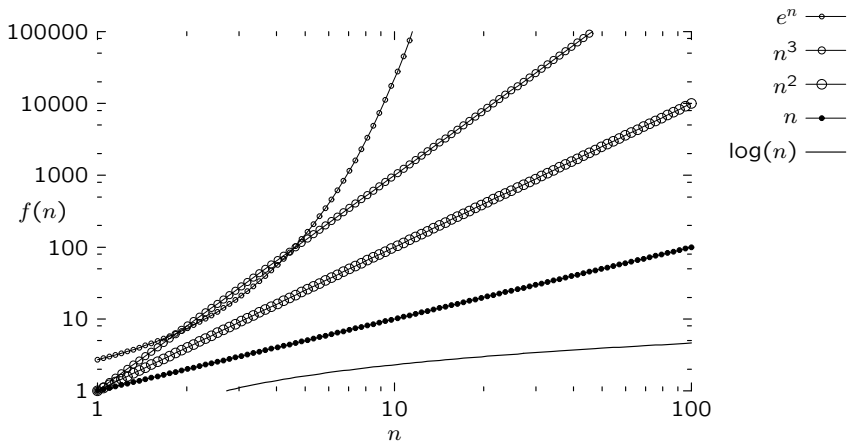
Wichtige Aufwandsklassen:

$O(1)$	konstant	$O(n^2)$	quadratisch
$O(\log(n))$	logarithmisch	$O(n^3)$	kubisch
$O(\log^k(n))$	poly-logarithmisch	$O(n^k)$	polynomiell
$O(n)$	linear	$O(2^n)$	exponentiell
$O(n \cdot \log(n))$			

Inklusionen der wichtigsten Aufwandsklassen:

$$O(1) \subset O(\log(n)) \subset O(\log^2(n)) \subset O(\sqrt{n}) \subset O(n) \subset O(n \cdot \log(n)) \subset O(n^2) \subset O(n^3) \subset O(2^n)$$

Vergleich der Aufwandsklassen



Vergleich der Aufwandsklassen (2)

GOp/sec	n^2	n^3	n^4	e^n
1	31.622	1.000	177	20
10	100.000	2.154	316	23
100	316.227	4.641	562	25
1000	1.000.000	10.000	1.000	27
$\Delta =$ $\cdot 10$	$\cdot 3.16$	$\cdot 2.15$	$\cdot 1.78$	$+2.3$

$O(n^2)$: $n_1 = \sqrt{1e9} = 31.622$

$n_2 = \sqrt{1e10} = \sqrt{1e9} \cdot \sqrt{10} = n_1 \cdot \sqrt{10}$

$O(e^n)$: $n_1 = \ln(1e9) = 20.723\ldots$

$n_2 = \ln(1e10) = \ln(1e9) + \ln(10) = n_1 + \ln(10)$

Vergleich der Aufwandsklassen (3)

Annahme: 1.000.000.000 Schritte pro Sekunde.

Lösbare Problemgröße bei verschiedenen Zeitvorgaben:

Aufwand	1 Sek	1 Min	1 Std	1 Tag
n^2	31.622	244.948	1.897.366	9.295.160
n^3	1.000	3.914	15.326	44.208
n^4	177	494	1.377	3.048
n^5	63	143	324	612
e^n	20	24	28	32

Komplexitätsmaße

Man unterscheidet die Laufzeit

- im besten Fall (best case)
- im Mittel (average case)
- im schlechtesten Fall (worst case)

Vergleich: Lineare Suche vs. binäre Suche

Algorithmus	best case	average case	worst case
lineare Suche	1	$N/2$	N
binäre Suche	1	$\log_2(N)$	$\log_2(N)$

Komplexitätsmaße (2)

Probleme bei average case:

- Worüber bildet man den Durchschnitt?
- Sind alle Eingaben der Länge N gleich wahrscheinlich (Gleichverteilung)?
- Technisch oft sehr viel schwieriger durchzuführen als worst-case Analyse.

Murphys Gesetz: Alles was schiefgehen kann, wird auch schiefgehen. $\Rightarrow$ Immer wenn ich das Programm ausführe, warte ich ewig.

Ungeeignet für kritische Anwendungen, bei denen maximale Reaktionszeiten garantiert werden müssen $\rightarrow$ *Echtzeitsysteme* (Prof. Dr. Quade)

Worst-Case Komplexität

Definition: (Worst-Case Komplexität)
W_n: Menge der zulässigen Eingaben der Länge n.
$A(w)$: Anzahl Schritte von Algorithmus A für Eingabe w.

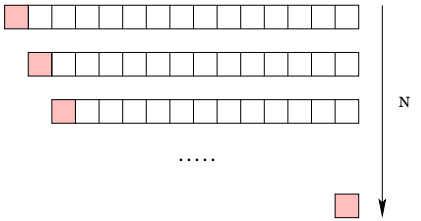
Worst-Case Komplexität (im schlechtesten Fall):

$$T_A(n) = \sup\{A(w) \mid w \in W_n\}$$

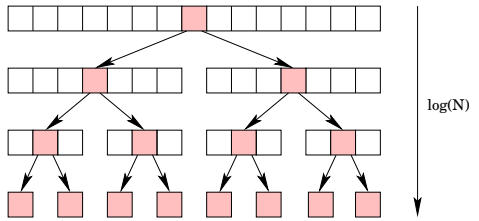
ist eine obere Schranke für die maximale Anzahl der Schritte, die Algorithmus A benötigt, um Eingaben der Größe n zu bearbeiten.

Worst-Case Komplexität: Beispiel

lineare Suche:



binäre Suche:



Zum Vergleich:

N	$\log(N)$
1.000.000	20
1.000.000.000	30
1.000.000.000.000	40

Average-Case Komplexität

Definition: (Average-Case Komplexität)
W_n: Menge der zulässigen Eingaben der Länge n.
$A(w)$: Anzahl Schritte von Algorithmus A für Eingabe w.

Average-Case Komplexität (erwarteter Aufwand):

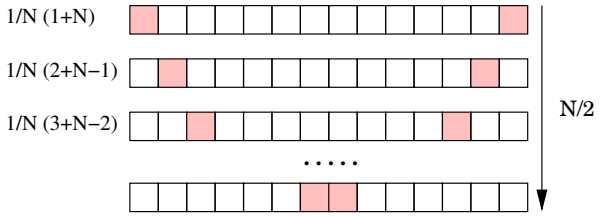
$$\overline{T}_A(n) = \frac{1}{|W_n|} \cdot \sum_{w \in W_n} A(w)$$

ist die mittlere Anzahl von Schritten, die Algorithmus A benötigt, um eine Eingabe der Größe n zu bearbeiten. Wir setzen hier eine Gleichverteilung voraus $\rightarrow$ arithmetischer Mittelwert

Average-Case Komplexität: Beispiel

lineare Suche:

Kosten: $1, \ldots, N$ Vergleiche
erwartete Kosten: $\frac{1}{N} \cdot (1 + 2 + 3 + \ldots + N)$

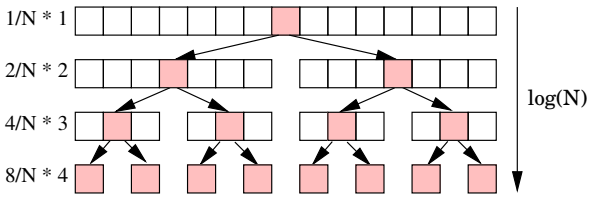


$$\frac{1}{N} \cdot (1 + 2 + 3 + \ldots + N) = \frac{1}{N} \cdot \frac{N(N+1)}{2} = \frac{N+1}{2}$$

Average-Case Komplexität: Beispiel (2)

binäre Suche:

Kosten: $1, \ldots, \log(N)$
zur Vereinfachung: $N = 2^x - 1$
erwartete Kosten: $\frac{1}{N} \cdot (1 + 2 \cdot 2 + 4 \cdot 3 + \ldots + 2^{x-1} \cdot x)$



Average-Case Komplexität: Beispiel (3)

Behauptung:

$$\sum_{i=1}^{x} i \cdot 2^{i-1} = (x-1) \cdot 2^x + 1$$

Beweis mittels vollständiger Induktion:

I.A. $x = 1$:

$$1 \cdot 2^0 = 1 \cdot 1 = 1 \stackrel{!}{=} 0 \cdot 2^1 + 1 = 1$$

I.S. $x \rightarrow x + 1$:

$$\begin{aligned} \sum_{i=1}^{x+1} i \cdot 2^{i-1} &= \sum_{i=1}^{x} i \cdot 2^{i-1} + (x+1) \cdot 2^x \\ &\stackrel{I.V.}{=} (x-1) \cdot 2^x + 1 + (x+1) \cdot 2^x \\ &= 2x \cdot 2^x + 1 = x \cdot 2^{x+1} + 1 \end{aligned}$$

Average-Case Komplexität: Beispiel (4)

Aus der Annahme $N = 2^x - 1$ folgt: $\log_2(N+1) = x$

Somit ergibt sich:

$$\begin{aligned}
\frac{1}{N} \cdot \sum_{i=1}^{x} i \cdot 2^{i-1} &= \frac{1}{N} \cdot [(x-1) \cdot 2^x + 1] \\
&= \frac{1}{N} \cdot [(\log_2(N+1) - 1) \cdot (N+1) + 1] \\
&= \frac{1}{N} \cdot [(N+1) \cdot \log_2(N+1) - N] \\
&\approx \log_2(N+1) - 1 \text{ für große } N
\end{aligned}$$

Im Mittel verursacht binäres Suchen also nur etwa eine Kosteneinheit weniger als im schlechtesten Fall.

Divide & Conquer

Entwurfsprinzip:

- **Divide** the problem into subproblems.
- **Conquer** the subproblems by solving them recursively.
- **Combine** subproblem solutions.

Beispiele:

- Binäre Suche
- Potenzieren einer Zahl
- Matrix-Multiplikation
- Quicksort

Divide & Conquer: Potenzieren einer Zahl

Problem: Berechne x^n für ein $n \in \mathbb{N}$.

- Einfacher Algorithmus:

```
erg := 1
for i := 1 to n do
    erg := erg * x
```

  $\rightarrow$ Laufzeit: $\Theta(n)$ Multiplikationen

- Divide & Conquer: **???**

Divide & Conquer: Matrix-Multiplikation

Eingabe: zwei $n \times n$-Matrizen A und B
Ausgabe: $C = A \cdot B$

Es gilt:

$$\begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$

mit

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

Divide & Conquer: Matrix-Multiplikation (2)

Einfacher Algorithmus:

```
for i := 1 to n do
    for j := 1 to n do
        c[i][j] := 0
        for k := 1 to n do
            c[i][j] := c[i][j] + a[i][k] * b[k][j]
```

→ Laufzeit: $\Theta(n^3)$ Additionen/Multiplikationen

Divide & Conquer: Matrix-Multiplikation (3)

Aufteilen der $n \times n$-Matrizen in jeweils vier $\frac{n}{2} \times \frac{n}{2}$-Matrizen:

$$\left(\begin{array}{c|c} r & s \\ \hline t & u \end{array}\right) = \left(\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right) \cdot \left(\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right)$$

$$C = A \cdot B$$

mit

$$\begin{aligned} r &= ae + bg \\ s &= af + bh \\ t &= ce + dg \\ u &= cf + dh \end{aligned}$$

$\Rightarrow$ 8 Multiplikationen von $\frac{n}{2} \times \frac{n}{2}$-Matrizen
4 Additionen von $\frac{n}{2} \times \frac{n}{2}$-Matrizen

→ Laufzeit: $T(n) = 8 \cdot T(n/2) + 4 \cdot (n/2)^2 = \ldots = \Theta(n^3)$

Matrix-Multiplikation: Strassens Idee

$$\left(\begin{array}{c|c} r & s \\ \hline t & u \end{array}\right) = \left(\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right) \cdot \left(\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right)$$

mit

$$\begin{aligned} P_1 &= a \cdot (f - h) \\ P_2 &= (a + b) \cdot h \\ P_3 &= (c + d) \cdot e \\ P_4 &= d \cdot (g - e) \\ P_5 &= (a + d) \cdot (e + h) \\ P_6 &= (b - d) \cdot (g + h) \\ P_7 &= (a - c) \cdot (e + f) \end{aligned}$$

und

$$\begin{aligned} r &= P_5 + P_4 - P_2 + P_6 \\ s &= P_1 + P_2 \\ t &= P_3 + P_4 \\ u &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

→ Laufzeit: $T(n) = 7T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})$

Matrix-Multiplikation: Strassens Idee (2)

Vergleich der Laufzeiten: (Annahme 1 GOp/s)

n	n^3	time	$n^{2.807}$	time	$n^{2.397}$	time
1.000	1e9	1 s	2.64e8	1 s	1.55e7	1 s
10.000	1e12	16 m	1.70e11	3 m	3.87e9	4 s
100.000	1e15	11 t	1.09e14	2 t	9.66e11	16 m
1.000.000	1e18	31 j	9.98e16	3 j	2.41e14	3 t

s: Sekunden
m: Minuten
t: Tage
j: Jahre

Rekursionsgleichungen

Master-Theorem:

$$T(n) = a \cdot T(n/b) + \Theta(n^k)$$

Unterscheide drei Fälle:

- für $a < b^k$ gilt: $T(n) = \Theta(n^k)$
- für $a = b^k$ gilt: $T(n) = \Theta(n^k \cdot \log(n))$
- für $a > b^k$ gilt: $T(n) = \Theta(n^{\log_b a})$

Rekursionsgleichungen (2)

Beispiele:

- Binäre Suche: $a = 1$, $b = 2$, $k = 0$

  $$a = b^k \;\rightarrow\; T(n) = \Theta(n^k \cdot \log(n)) = \Theta(\log(n))$$

- Matrix-Multiplikation: $a = 8$, $b = 2$, $k = 2$

  $$a > b^k \;\rightarrow\; T(n) = \Theta(n^{\log_b a}) = \Theta(n^3)$$

Algorithmen & Datenstrukturen

Sortieren

Quicksort

Einige Fakten:

- 1962 von C.A.R. Hoare veröffentlicht.
- divide & conquer Algorithmus
- eines der schnellsten allgemeinen Sortierverfahren
- in-situ: kein zusätzlicher Speicherplatz zur Speicherung von Datensätzen erforderlich (ausser einer konstanten Anzahl von Hilfsspeicherplätzen für Tauschoperationen)
- praxistauglich

Quicksort (2)

1. **Divide:** Wähle aus allen Werten einen beliebigen Wert p aus und teile die Folge in zwei Teilfolgen K und G auf:
   - K enthält Werte die kleiner oder gleich p sind,
   - G enthält Werte die größer oder gleich p sind.

2. **Conquer:** Sortiere K und G rekursiv
3. **Combine:** trivial

Noch offene Punkte:

- aufteilen der Folge in zwei Teilfolgen
- wählen des Pivot-Elements

Quicksort (3)

Pivot-Element: erstes Element der Teilfolge

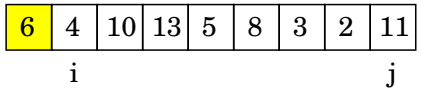
Aufteilen der Folge in zwei Teilfolgen:

```
partition(int l, int r)
    p := A[l],  i := l+1,  j := r
    repeat
        while (A[i] ≤ p) and (i < r) do i := i + 1
        while (A[j] ≥ p) and (j > l) do j := j - 1
        if i < j then swap(i, j)
    until j ≤ i
    swap(l, j)
    return j
```

→ Laufzeit: $\Theta(n)$ für n Elemente

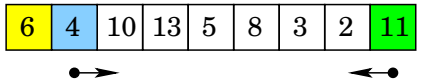
Quicksort (4)

Beispiel:



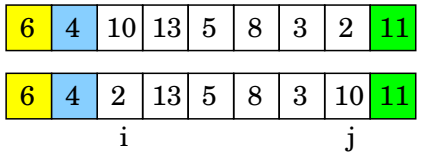
Quicksort (4)

Beispiel:



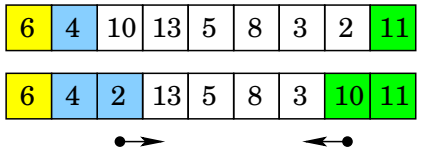
Quicksort (4)

Beispiel:



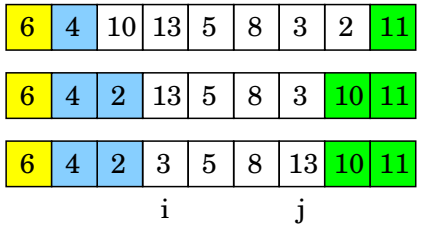
Quicksort (4)

Beispiel:



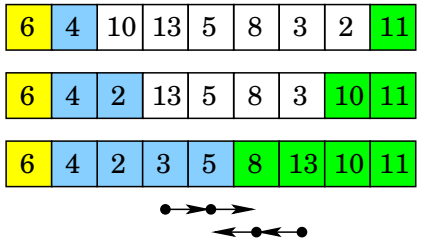
Quicksort (4)

Beispiel:



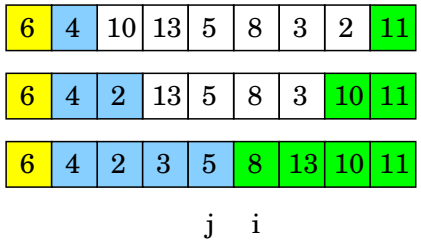
Quicksort (4)

Beispiel:



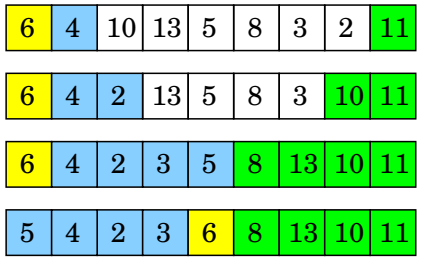
Quicksort (4)

Beispiel:



Quicksort (4)

Beispiel:



Quicksort (5)

Pseudo-Code:

```
quicksort(int l, int r)
    if l < r
        m := partition(l, r)
        quicksort(l, m-1)
        quicksort(m+1, r)
```

initial: quicksort(0, n-1)

Quicksort (6)

worst-case Analyse:

betrachte sortierte Folge: bei jedem rekursiven Aufruf ist die Teilfolge K leer und Teilfolge G ist um ein Element (dem Pivot-Element) kürzer geworden.

$$
\begin{aligned}
T(n) &= T(n-1) + \Theta(n) \\
&= T(n-2) + \Theta(n-1) + \Theta(n) \\
&= T(n-3) + \Theta(n-2) + \Theta(n-1) + \Theta(n) \\
&\vdots \\
&= T(1) + \Theta(2) + \ldots + \Theta(n-2) + \Theta(n-1) + \Theta(n) \\
&= \Theta\left(\sum_{k=1}^{n} k\right) = \Theta\left(\frac{n(n+1)}{2}\right) = \Theta(n^2)
\end{aligned}
$$

Quicksort (7)

best-case Analyse:

- die Folge wird bei jeder Aufteilung halbiert
- $T(n) = 2 \cdot T(n/2) + \Theta(n)$

Das Master-Theorem liefert für $a = 2$, $b = 2$ und $k = 1$:

$$T(n) = \Theta(n^k \cdot \log(n)) = \Theta(n \cdot \log(n))$$

Frage:

Welche Laufzeit hat Quicksort, wenn die Aufteilung immer in einem festen Verhältnis, z.B. $\frac{1}{10} : \frac{9}{10}$ erfolgt?

Quicksort (8)

Fazit:

- worst case: $T(n) \in \Theta(n^2)$
- average case: $T(n) \in \Theta(n \cdot \log(n))$
- best case: $T(n) \in \Theta(n \cdot \log(n))$

Problem: Laufzeit $O(n^2)$ bei stark vorsortierten Folgen

Lösung: Zufallsstrategie: wähle als Pivot-Element ein zufälliges Element aus $A[l...r]$ und vertausche es mit $A[l]$.

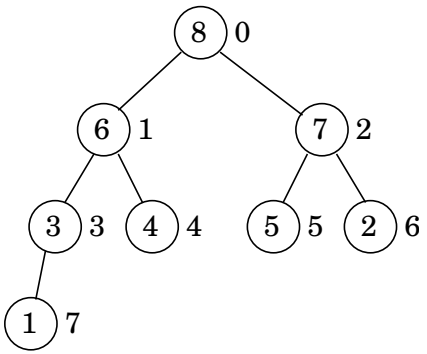
$\Rightarrow$ Laufzeit ist unabhängig von der zu sortierenden Folge

$\Rightarrow$ mittlere/erwartete Laufzeit: $\Theta(n \cdot \log(n))$

Heapsort

Heap: eine Folge $F = k_0, \ldots, k_n$ von Schlüsseln, so dass $k_i \geq k_{2i+1}$ und $k_i \geq k_{2(i+1)}$ gilt, sofern $2i+1 \leq n$ bzw. $2(i+1) \leq n$.

Beispiel: $F = 8, 6, 7, 3, 4, 5, 2, 1$



Heap-Eigenschaft ist erfüllt:

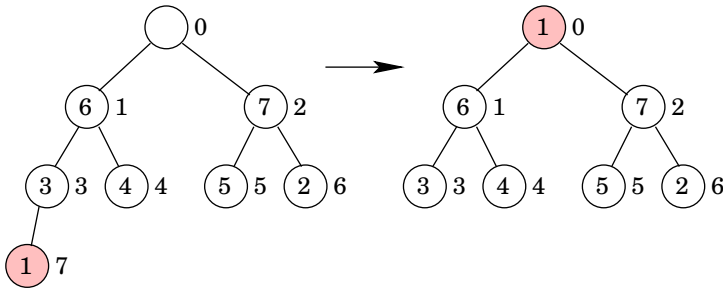
- $k_0 = 8 \geq k_1 = 6$ und $k_0 \geq k_2 = 7$
- $k_1 = 6 \geq k_3 = 3$ und $k_1 \geq k_4 = 4$
- $k_2 = 7 \geq k_5 = 5$ und $k_2 \geq k_6 = 2$
- $k_3 = 3 \geq k_7 = 1$

Heapsort (2)

Bestimmung des Maximums ist leicht: k_0 ist das Maximum.

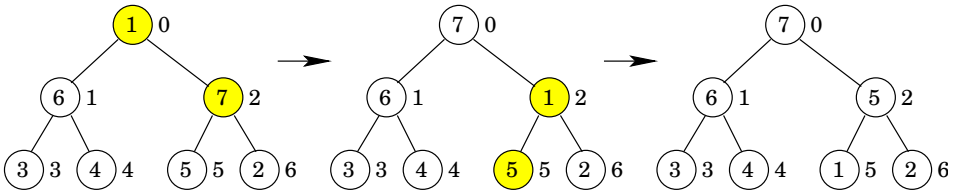
Das nächst kleinere Element wird bestimmt, indem das Maximum aus F entfernt wird und die Restfolge wieder zu einem Heap transformiert wird:

1. Setze den Schlüssel mit dem größten Index an die erste Position. $\Rightarrow$ Heap-Eigenschaft verletzt!



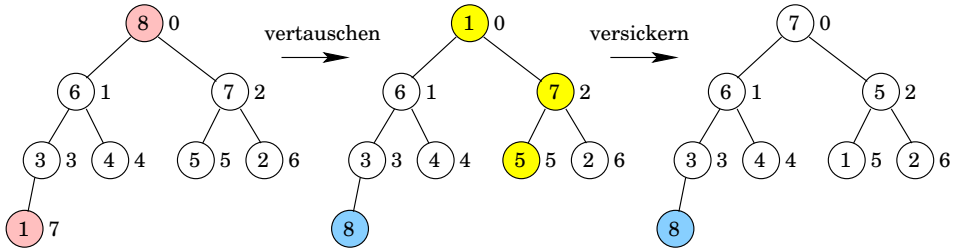
Heapsort (3)

2. Schlüssel versickern lassen, indem er immer mit dem größten seiner Nachfolger getauscht wird, bis entweder beide Nachfolger kleiner sind oder der Schlüssel unten angekommen ist.

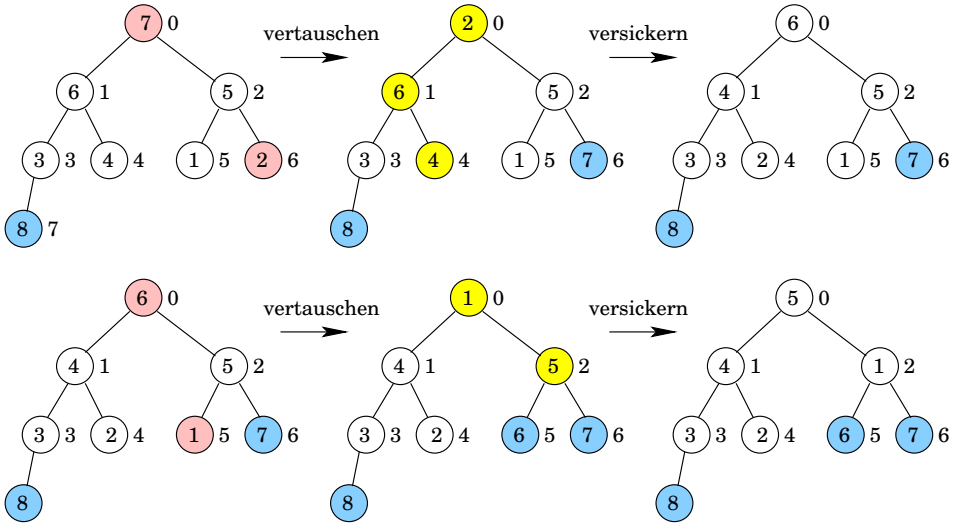


Heapsort (4)

Die Datensätze können sortiert werden, indem das jeweils aus dem Heap entfernte Maximum an die Stelle desjenigen Schlüssels geschrieben wird, der nach dem Entfernen des Maximums nach k_0 übertragen wird.



Heapsort (5)



usw.

Heapsort (6)

Analyse:

- Es erfolgen $n-1$ Vertauschungen außerhalb der Funktion `versickern`.
- Innerhalb von `versickern` wird ein Schlüssel wiederholt mit einem seiner Nachfolger vertauscht, wobei der Datensatz jeweils eine Stufe tiefer wandert.
- Heap mit n Datensätzen hat $\lceil \log_2(n+1) \rceil$ viele Ebenen.

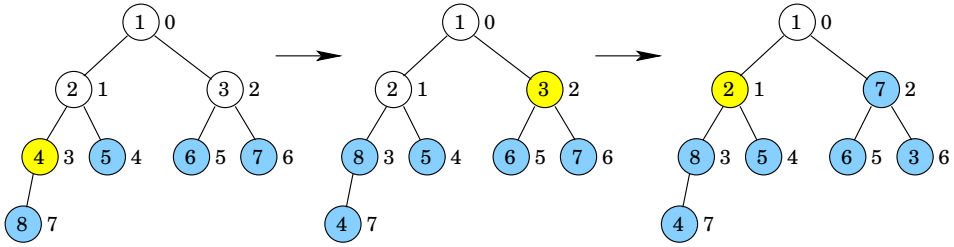
$\Rightarrow$ obere Schranke von $O(n \cdot \log(n))$ Vertauschungen

Heapsort (7)

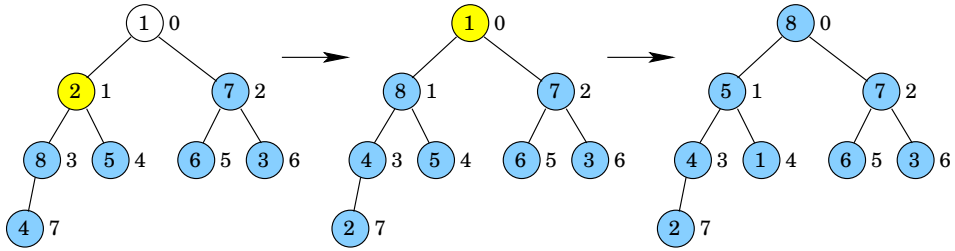
Wie wird die Anfangsfolge in einen Heap transformiert?

Idee: Lasse die Schlüssel $k_{n/2-1}, \ldots, k_0$ versickern. Dadurch werden schrittweise immer größere Heaps aufgebaut, bis letztlich ein Heap übrig bleibt.

Beispiel: $F = 1, 2, 3, 4, 5, 6, 7, 8$



Heapsort (8)



Analyse: aufbauen des initialen Heaps ist in linearer Zeit möglich, weil ...

Heapsort (9)

Damit ergibt sich insgesamt: Laufzeit $O(n \cdot \log(n))$

Anmerkungen:

- eine Vorsortierung der Eingabefolge schadet und nützt der Sortierung nichts
- Heapsort benötigt nur konstant viel zusätzlichen Speicherplatz (in-situ Sortierverfahren)

Untere Schranke

allgemeine Sortierverfahren: Algorithmen, die nur Vergleichsoperationen zwischen Schlüsseln verwenden.

Satz: Jedes allgemeine Sortierverfahren benötigt zum Sortieren von n verschiedenen Schlüsseln im schlechtesten Fall und im Mittel wenigstens $\Omega(n \cdot \log(n))$ Schlüsselvergleiche.

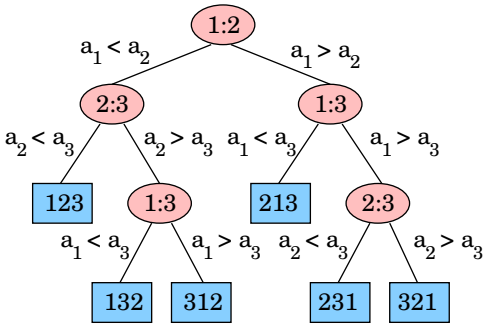
In [1] wird Sortierverfahren vorgestellt, das arithmetische Operationen und die Floor-Funktion benutzt, und das im Mittel eine Laufzeit von $O(n)$ hat.

---

[1] W. Dobosiewicz: Sorting by distributive partitioning. Information Processing Letters, 7(1), 1978

Untere Schranke (2)

Entscheidungsbaum:



- innere Knoten sind mit $i : j$ beschriftet für $i, j \in \{1, 2, \ldots, n\}$
- linker Teilbaum enthält nachfolgende Vergleiche, falls $a_i < a_j$
- rechter Teilbaum: nachfolgende Vergleiche, falls $a_i > a_j$
- jedes Blatt: Permutation $(\pi(1), \pi(2), \ldots, \pi(n))$ bezeichnet die Sortierung $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$

Untere Schranke (3)

Ein Entscheidungsbaum kann jedes allgemeine Sortierverfahren modellieren:

- jeweils ein Baum für jede Eingabegröße n
- Laufzeit des Algorithmus = Länge des gewählten Pfades
- worst-case Laufzeit = Höhe des Baums

Untere Schranke (4)

worst-case Analyse:

- es gibt $n!$ verschiedene Permutationen über n Zahlen
- ein Entscheidungsbaum hat mindestens $n!$ Blätter
- ein Binärbaum der Höhe h hat maximal $2^h - 1$ Blätter

$\Rightarrow 2^h \geq n!$

$$
\begin{aligned}
h &\geq \log(n!) && \text{log ist monoton steigend} \\
&\geq \log((n/e)^n) && \text{Stirling-Formel} \\
&= \log(n^n) - \log(e^n) \\
&= n \cdot \log(n) - n \cdot \log(e) \\
&\in \Omega(n \cdot \log(n))
\end{aligned}
$$

Untere Schranke (5)

average-case Analyse: Beweis durch Widerspruch

- **Behauptung:** Die mittlere Tiefe eines Entscheidungsbaums mit k Blättern ist wenigstens $\log_2(k)$.
- **Annahme:** Es existiert ein Entscheidungsbaum mit k Blättern dessen mittlere Tiefe kleiner als $\log_2(k)$ ist.

Sei T ein solcher Baum, der unter all diesen Bäumen der kleinste ist. T habe k Blätter. Dann gilt:

- T hat einen linken Teilbaum T_1 mit k_1 Blättern
- T hat einen rechten Teilbaum T_2 mit k_2 Blättern
- es gilt $k_1 < k$, $k_2 < k$ und $k_1 + k_2 = k$

Untere Schranke (6)

Da T der kleinste Baum ist, für den die Annahme gilt, erhalten wir für T_1 und T_2:

mittlere Tiefe$(T_1) \geq \log_2(k_1)$

mittlere Tiefe$(T_2) \geq \log_2(k_2)$

Jedes Blatt in T_1 bzw. T_2 auf Tiefe t hat in T die Tiefe $t+1$. Also gilt insgesamt:

$$\begin{aligned} \mathrm{mT}(T) &= \frac{k_1}{k}(\mathrm{mT}(T_1)+1) + \frac{k_2}{k}(\mathrm{mT}(T_2)+1) \\ &\geq \frac{k_1}{k}(\log_2(k_1)+1) + \frac{k_2}{k}(\log_2(k_2)+1) \\ &= \frac{1}{k}(k_1 \cdot \log_2(2k_1) + k_2 \cdot \log_2(2k_2)) \end{aligned}$$

Untere Schranke (7)

Unter der Nebenbedingung $k_1 + k_2 = k$ hat die Funktion

$$\mathrm{mT}(T) = \frac{1}{k}(k_1 \cdot \log_2(2k_1) + k_2 \cdot \log_2(2k_2))$$

ein Minimum bei $k_1 = k_2 = k/2$. Damit gilt

$$\mathrm{mT}(T) \geq \frac{1}{k}\left(\frac{k}{2}\log_2(k) + \frac{k}{2}\log_2(k)\right) = \log_2(k)$$

im Widerspruch zur Annahme.

Jeder Entscheidungsbaum zur Sortierung von n Zahlen hat mindestens $n!$ Blätter, daher gilt:

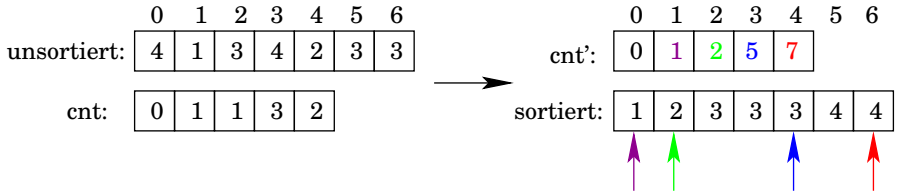
$$\mathrm{mT}(T) \geq \log_2(n!) \geq \log_2((n/e)^n) \in \Omega(n \log(n))$$

Counting Sort

Sortierverfahren ohne Schlüsselvergleiche!

Eingabe: $a[0...N]$ mit $a[i] \in \{0,\ldots,k\}$
Ausgabe: $b[0...N]$ sortiert
Hilfsspeicher: $cnt[0...k]$ zum Zählen

Idee: Zähle die Häufigkeiten der Zahlen $0,\ldots,k$ in der zu sortierenden Liste und berechne daraus die Position der jeweiligen Zahl in der sortierten Liste.



Counting Sort (2)

Pseudo-Code:

```
for i := 0 to k-1 do                    /* init */
    cnt[i] := 0
for j := 0 to n-1 do                    /* count */
    cnt[a[j]] := cnt[a[j]] + 1
for i := 1 to k-1 do                    /* collect */
    cnt[i] := cnt[i] + cnt[i-1]
for j := n-1 downto 0 do                /* rearrange */
    b[cnt[a[j]] - 1] := a[j]
    cnt[a[j]] := cnt[a[j]] - 1
```

Counting Sort (3a)

	0	1	2	3	4	5	6	7	8	9	10	11	12
a:	3	6	7	5	3	5	6	2	9	1	2	7	3
cnt:	0	1	2	3	0	2	2	2	0	1			

Counting Sort (3b)

	0	1	2	3	4	5	6	7	8	9	10	11	12
a:	3	6	7	5	3	5	6	2	9	1	2	7	3
cnt:	0	1	2	3	0	2	2	2	0	1			

cnt:	0	1	3	6	6	8	10	12	12	13			

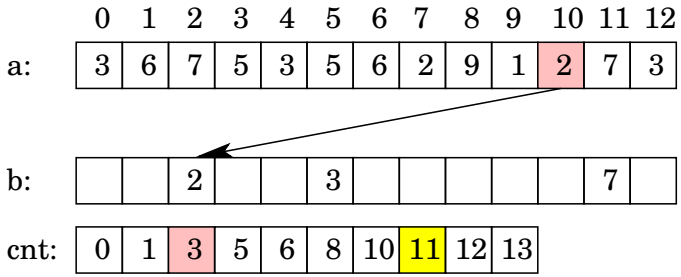
Counting Sort (3c)

	0	1	2	3	4	5	6	7	8	9	10	11	12
a:	3	6	7	5	3	5	6	2	9	1	2	7	3
b:						3							
cnt:	0	1	3	6	6	8	10	12	12	13			

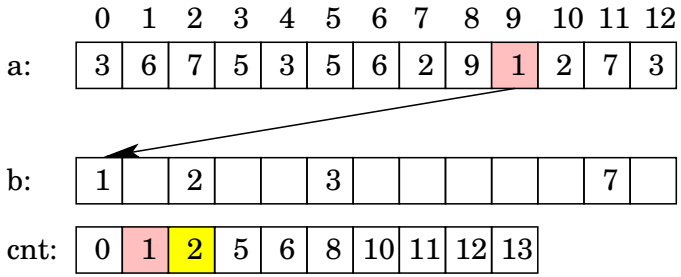
Counting Sort (3d)

	0	1	2	3	4	5	6	7	8	9	10	11	12
a:	3	6	7	5	3	5	6	2	9	1	2	7	3
b:						3						7	
cnt:	0	1	3	5	6	8	10	12	12	13			

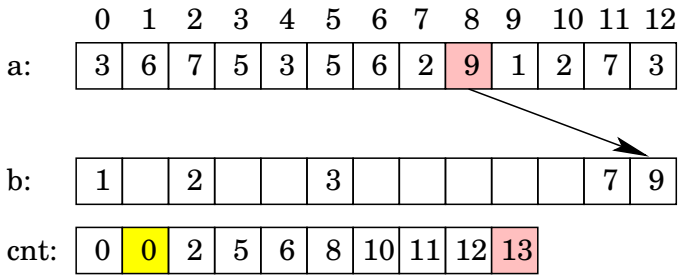
Counting Sort (3e)



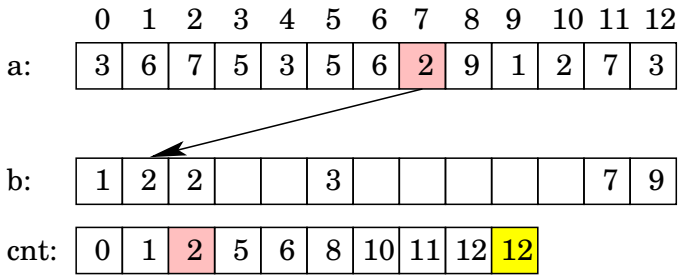
Counting Sort (3f)



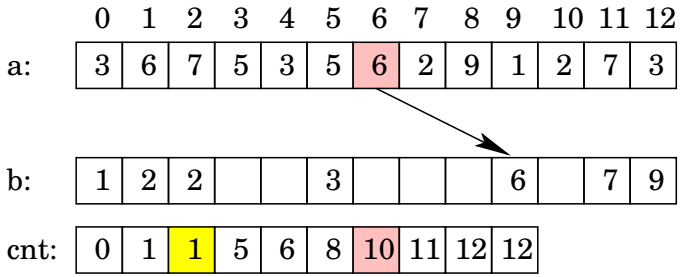
Counting Sort (3g)



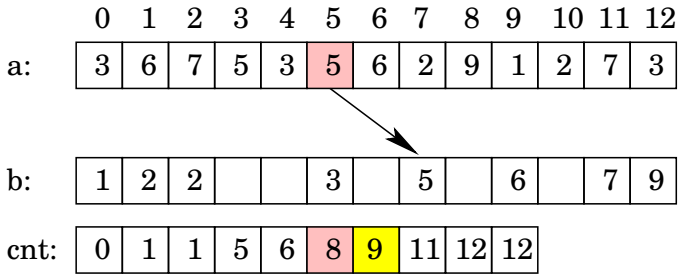
Counting Sort (3h)



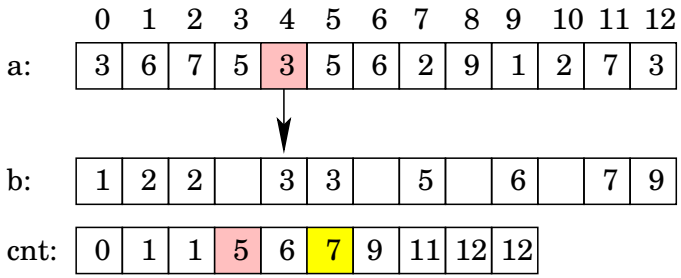
Counting Sort (3i)



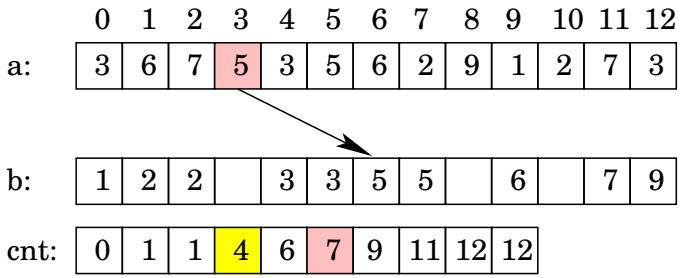
Counting Sort (3j)



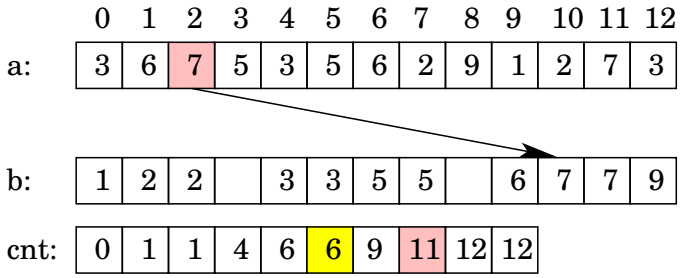
Counting Sort (3k)



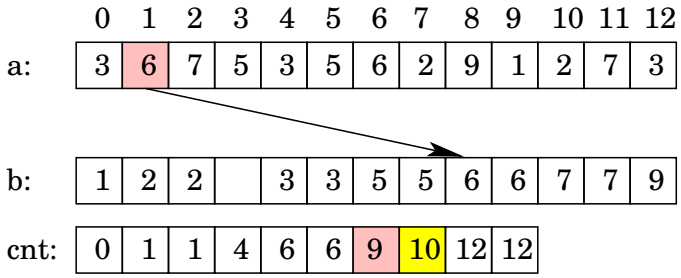
Counting Sort (3l)



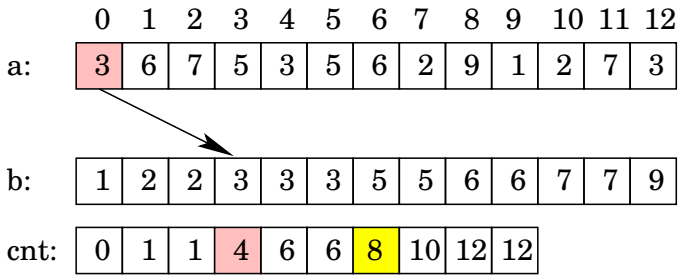
Counting Sort (3m)



Counting Sort (3n)



Counting Sort (3o)

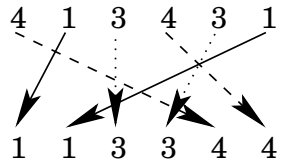


Counting Sort (4)

Laufzeit:

init	$\Theta(k)$
count	$\Theta(n)$
collect	$\Theta(k)$
rearrange	$\Theta(n)$
	$\Theta(n+k)$

Counting Sort ist ein **stabiles Sortierverfahren**: gleiche Elemente stehen nach dem Sortieren in der gleichen Reihenfolge wie vor dem Sortieren.



Radix Sort

Problem: Counting Sort ist ineffizient bzgl. Laufzeit und Speicherplatz, wenn der Wertebereich groß ist im Vergleich zur Anzahl der Zahlen: $\Theta(n+k) = \Theta(n^2)$ für $k = n^2$

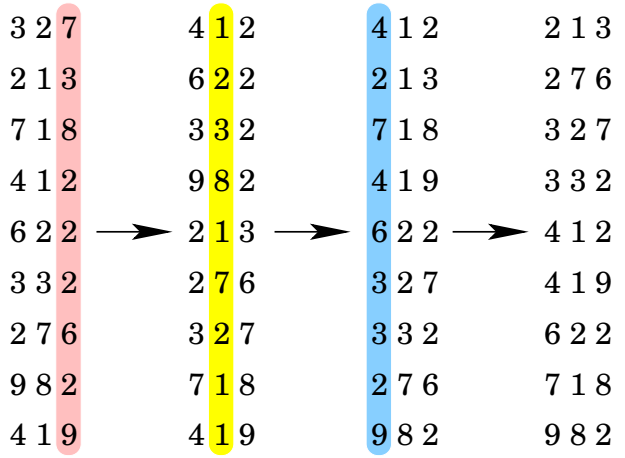
Beispiel: es sollen 100.000 Datensätze mit einem 32Bit-Schlüssel sortiert werden, also $k = 2^{32} \approx 4.300.000.000$

- $n + k \approx 4.300.000.000$
- $n \cdot \log(n) \approx 1.600.000$

Lösung: Sortiere die Zahlen anhand der einzelnen Ziffern, beginnend mit der niederwertigsten Stelle. Verwende ein stabiles Sortierverfahren wie Counting Sort. Falls nötig, müssen führende Nullen eingefügt werden.

Radix Sort (2)

Beispiel:



Radix Sort (3)

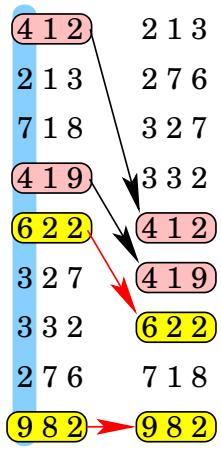
Korrektheit: Induktion über die betrachtete Position

I.A. nach der ersten Sortierphase sind die Zahlen bezüglich der Position 0 sortiert

I.V. die Zahlen sind bezüglich der $t-1$ niederwertigsten Ziffern sortiert

I.S. Sortiere nach Ziffer t:

- zwei Zahlen, die sich an Position t unterscheiden, sind richtig sortiert
- zwei Zahlen, die an Position t dieselbe Ziffer haben, sind nach I.V. richtig sortiert und bleiben aufgrund des stabilen Sortierverfahrens sortiert



Radix Sort (4)

Laufzeit:

- fasse jeweils r bits zu einer Ziffer zusammen
- bei einer Wortlänge von b bit müssen b/r Phasen durchlaufen werden

Frage: Wie groß muss r im Verhältnis zu b sein, um eine gute Laufzeit zu erzielen?

- Counting Sort hat Laufzeit $\Theta(n+k)$. Hier: $\Theta(n+2^r)$
- bei b/r Phasen erhalten wir: $\Theta(b/r \cdot (n+2^r))$
- Extremwert bestimmen: bei welchem Wert von r wird die Laufzeit minimal?

Praxis: 32bit-Wörter in Gruppen von 8bit $\Rightarrow$ 4 Phasen

Algorithmen & Datenstrukturen

Suchen

Exponentielle Suche

Gesucht wird ein Element mit Schlüssel k. Die exponentielle Suche eignet sich zum Suchen in sortierten, linearen Listen, deren Länge nicht bekannt ist.

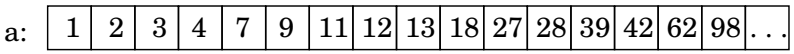
Bestimme in exponentiell wachsenden Schritten einen Bereich, in dem der Schlüssel liegen muss:

```
i = 1;
while (k > a[i].key)
    i *= 2;
```

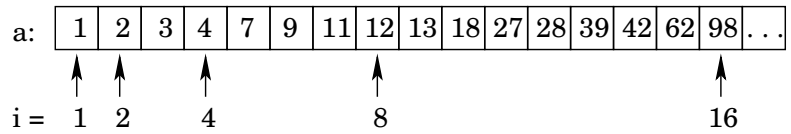
Danach gilt: `a[i/2].key` $<$ `k` $\leq$ `a[i].key`
Dieser Bereich wird mittels binärer Suche durchsucht.

Exponentielle Suche: Beispiel

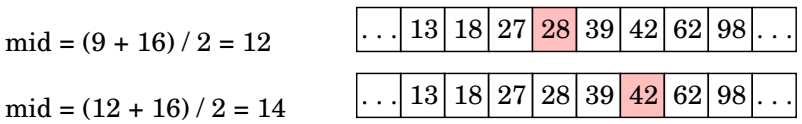
Wir suchen den Schlüssel $k = 42$ in folgender Liste:



(1) bestimme den Bereich, in dem k liegen muss:



(2) binäre Suche im Bereich `a[i/2+1] ... a[i]`:



Exponentielle Suche: Laufzeit

Voraussetzung: Werte sind paarweise verschieden (dies ist z.B. dann gegeben, wenn Schlüsselwerte gesucht werden, Telefonnummer, Matrikelnummer, Personalnummer, usw.)

$\Rightarrow$ Schlüssel wachsen mindestens so stark wie die Indices

$\Rightarrow$ die Bereichsbestimmung erfolgt in $\log_2(k)$ Schritten

$\Rightarrow$ der zu durchsuchende Bereich hat höchstens k Zahlen und kann in Zeit $\log_2(k)$ durchsucht werden

insgesamt ergibt sich also eine Laufzeit von $\Theta(\log(k))$

Falls die Folge auch gleiche Werte enthalten darf, ist keine Laufzeitabschätzung möglich.

Interpolations-Suche

Idee: Schätze die Position des Elements mit Schlüssel k.

Beispiel: Im Telefonbuch steht der Name `Zimmermann` weit hinten, wohingegen `Brockmann` eher am Anfang steht.

Sei l linke Grenze, r rechte Grenze des Suchbereichs:

- bei der binären Suche wurde der Index des nächsten zu inspizierenden Elements bestimmt als

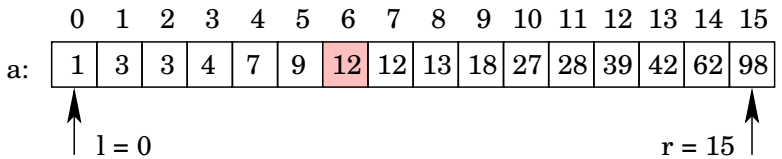
$$m \;=\; l + \frac{1}{2} \cdot (r - l).$$

- bei der Interpolationssuche inspiziere als nächstes das Element auf Position

$$m \;=\; l + \frac{k - a[l].key}{a[r].key - a[l].key} \cdot (r - l).$$

Interpolations-Suche: Beispiel (a)

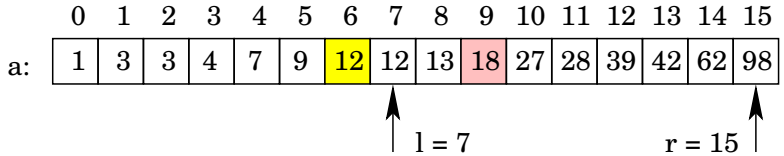
gesucht: $k = 42$



m = 0 + (42 – 1) / (98 – 1) * (15 – 0) = 6

Interpolations-Suche: Beispiel (b)

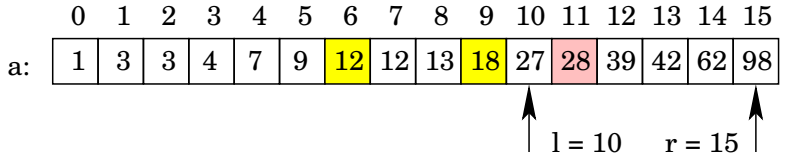
gesucht: $k = 42$



m = 7 + (42 – 12) / (98 – 12) * (15 – 7) = 9

Interpolations-Suche: Beispiel (c)

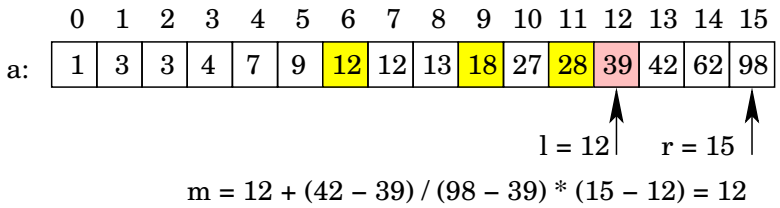
gesucht: $k = 42$



m = 10 + (42 – 27) / (98 – 27) * (15 – 10) = 11

Interpolations-Suche: Beispiel (d)

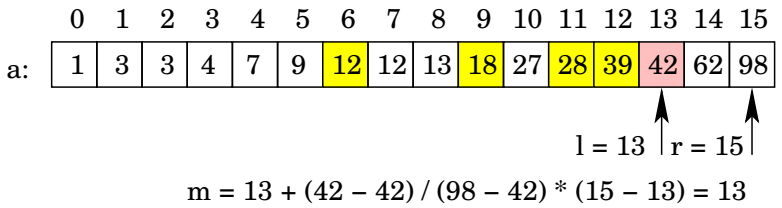
gesucht: $k = 42$



m = 12 + (42 – 39) / (98 – 39) * (15 – 12) = 12

Interpolations-Suche: Beispiel (e)

gesucht: $k = 42$



m = 13 + (42 – 42) / (98 – 42) * (15 – 13) = 13

Interpolations-Suche: Laufzeit

Im Mittel werden $\log(\log(N)) + 1$ Schlüsselvergleiche ausgeführt [1], aber dieser Vorteil geht durch die auszuführenden arithmetischen Operationen oft verloren.

Im worst-case hat das Verfahren eine lineare Laufzeit, also $\Theta(n)$ bei n Schlüsselwerten.

Beispiel: $k = 10$ und $F = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1000$

---

[1] A.C. Yao and F.F. Yao: The complexity of searching an ordered random table. Proceedings of the Symposium on Foundations of Computer Science, 1976

Hash-Verfahren

Schlüsselsuche durch Berechnung der Array-Indices!

Idee:

- bei einer Menge von Werten $K \subseteq \{0, 1, \ldots, m-1\}$
- verwende ein Array $A[0 \ldots m-1]$
- und speichere die Schlüssel wie folgt:

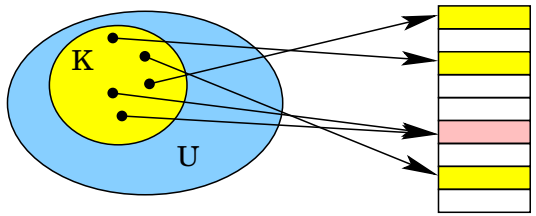
$$A[k] = \begin{cases} x & \text{falls } x.key \in K \text{ und } x.key = k \\ nil & \text{sonst} \end{cases}$$

$\Rightarrow$ suchen, einfügen und löschen in $\Theta(1)$ Schritten

Problem: Wertebereich kann sehr groß sein
8-stellige Namen $\rightarrow$ mehr als 208.000.000.000 Werte

Hash-Verfahren (2)

Lösung: verwende eine Hash-Funktion h, um den Wertebereich U, speziell die Menge der Schlüsselwerte K, auf die Zahlen $0, \ldots, m-1$ abzubilden, also $h : U \to \{0, \ldots, m-1\}$.



Hash-Funktion i.Allg. nicht injektiv: verschiedene Schlüssel werden auf dieselbe Hash-Adresse abgebildet → Kollision

Hash-Funktion wird zum Plazieren und Suchen verwendet, muss einfach/effizient zu berechnen sein

Hash-Verfahren (3)

Schlüssel, die nicht als Zahlen interpretiert werden können, müssen vorher geeignet umgerechnet werden.

Beispiel: Zeichenketten der Länge 3

- Die ASCII-Darstellung wird als Binärzahl interpretiert.
- Interpretiere die Binärzahlen als Ziffern einer Zahl zur Basis 256.

Input	k_1	k_2	k_3	k	$h(k) = k \bmod 11$
`i`	0	0	105	105	6
`i2`	0	105	50	26930	2
`ii`	0	105	105	26985	2
`iii`	105	105	105	6908265	1
`i_2`	105	95	50	6905650	4

Hash-Verfahren (4)

Hash-Verfahren muss zwei Forderungen genügen:

1. es sollen möglichst wenige Kollisionen auftreten
2. Adress-Kollisionen müssen effizient aufgelöst werden

Wahl der Hash-Funktion:

- soll die zu speichernden Datensätze möglichst gleichmäßig auf den Speicherbereich verteilen, um Adress-Kollisionen zu vermeiden
- Häufungen in der Schlüsselverteilung sollen sich nicht auf die Verteilung der Adressen auswirken
- es gilt: wenn eine Hash-Funktion $\sqrt{\pi n/2}$ Schlüssel auf eine Tabelle der Größe n abbildet, dann gibt es fast sicher eine Kollision (für $n = 365$ ist $\sqrt{\pi n/2} \approx 23$)

Hash-Funktion: Division-Rest-Methode

Der Schlüssel wird ganzzahlig durch die Länge der Hash-Tabelle dividiert. Der Rest wird als Index verwendet:

$$h(k) = k \bmod m$$

zu beachten:

- m soll keinen kleinen Teiler haben!
- m soll keine Potenz der Basis des Zahlensystems sein! Beispiel: für $m = 2^r$ hängt der Hash-Wert nur von den letzen r Bit ab

$\Rightarrow$ wähle m als Primzahl, die nicht nah an einer Potenz der Basis des Zahlensystems liegt. Beispiel: $m = 761$, **aber nicht:** $m = 509$ (nah an 2^9) oder $m = 997$ (nah an 10^3)

Hash-Funktion: Multiplikative Methode

Sei $m = 2^r$ eine Zweierpotenz. Bei einer Wortgröße w wähle eine Zahl a so, dass $2^{w-1} < a < 2^w$ ist.

$$h(k) = (a \cdot k \bmod 2^w) >> (w - r)$$

Anmerkungen:

- Wähle a nicht zu nah an 2^w
- Modulo-Operation und Rechts-Shift ist schnell
- gute Ergebnisse für $a \approx \frac{\sqrt{5}-1}{2} \cdot 2^w$ (goldener Schnitt)

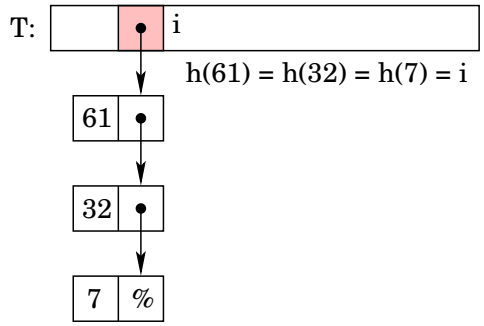
Beispiel: Für $w = 8$, $r = 3$, $a = 191$ und $k = 23$ erhalten wir

	10111111	191
*	00010111	23
10001001	01001	4393

Hash-Verfahren: Verkettung der Überläufer

Probleme treten auf

- beim Einfügen, wenn die berechnete Hash-Adresse nicht leer ist
- bei der Suche, wenn der berechnete Platz ein anderes Element enthält



Kollisionsauflösung: Die Überläufer können in einer linearen Liste verkettet werden, die an den Hash-Tabelleneintrag angehängt wird.

Verkettung der Überläufer: Bewertung

Die durchschnittliche Anzahl der Einträge in $h(k)$ ist N/M, wenn N Einträge durch die Hash-Funktion gleichmäßig auf M Listen verteilt werden. **Belegungsfaktor:** $\alpha = \frac{N}{M}$

Mittlere Laufzeit bei erfolgreicher Suche:

- beim Einfügen des j-ten Schlüssels ist die durchschnittliche Listenlänge $\frac{j-1}{M}$
- bei einer späteren Suche nach dem j-ten Schlüssel betrachten wir also im Durchschnitt $1+\frac{j-1}{M}$ Einträge, wenn stets am Listenende eingefügt wird und keine Elemente gelöscht wurden

$$\overline{C} = \frac{1}{N} \sum_{j=1}^{N} \left(1 + \frac{j-1}{M}\right) = 1 + \frac{N-1}{2M} \approx 1 + \frac{N}{2M}$$

Verkettung der Überläufer: Bewertung (2)

in der Regel gilt $N \in O(M)$ (die Größe der Hash-Tabelle ist ungefähr so groß wie die Anzahl der Datensätze)

$$\Rightarrow \quad \alpha = \frac{N}{M} = \frac{O(M)}{M} \in O(1)$$

zum Vergleich: binäre Suche hat Laufzeit $\Theta(\log(N))$ plus Aufwand $\Theta(N \log(N))$ zum Sortieren der Datensätze

Problem: verkettete Listen sind umständlich zu Programmieren und dynamische Speicheranforderung ist teuer.

Lösung: speichere alle Datensätze innerhalb des Arrays

offene Hash-Verfahren

Idee: Speichere die Überläufer in der Hash-Tabelle, nicht in zusätzlichen Listen

- ist die Hash-Adresse $h(k)$ belegt, so wird systematisch eine Ausweichposition gesucht
- die Folge der zu betrachtenden Speicherplätze für einen Schlüssel nennt man **Sondierungsfolge**
- die Hash-Funktion hängt vom Schlüssel und von der Anzahl durchgeführter Plazierungsversuche ab:

$$h : U \times \{0, \ldots, m-1\} \to \{0, \ldots, m-1\}$$

- die Sondierungsfolge muss eine Permutation der Zahlen $0, \ldots, m-1$ sein, damit alle Einträge der Hash-Tabelle genutzt werden können

offene Hash-Verfahren (2)

Anmerkungen:

- beim Einfügen und Suchen wird dieselbe Sondierungsfolge durchlaufen
- beim Löschen wird der Datensatz nicht gelöscht, sondern nur als gelöscht markiert (der Wert wird ggf. bei späterem Einfügen überschrieben)
- je voller die Tabelle wird, umso schwieriger wird das Einfügen neuer Schlüssel

lineares Sondieren

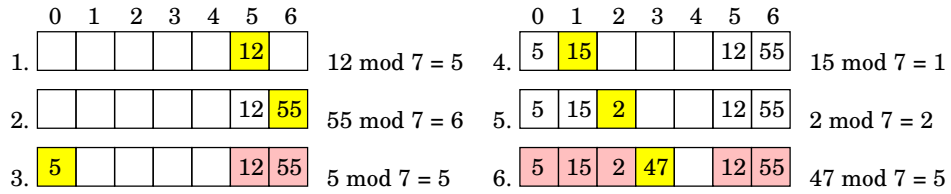
zu einer gegebenen Hash-Funktion $h'(k)$ sei

$$h(k, i) = (h'(k) + i) \bmod m.$$

Beispiel: betrachte das Einfügen der Schlüssel

$$12, 55, 5, 15, 2, 47$$

für $M = 7$ und $h'(k) = k \bmod 7$



quadratisches Sondieren

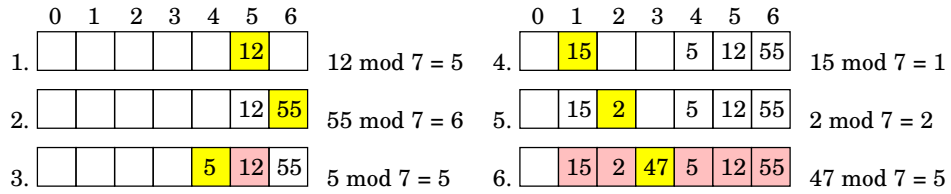
zu einer gegebenen Hash-Funktion $h'(k)$ sei

$$h(k, i) = (h'(k) + (-1)^i \cdot \lceil i/2 \rceil^2) \bmod m.$$

Beispiel: betrachte das Einfügen der Schlüssel

$$12, 55, 5, 15, 2, 47$$

für $M = 7$ und $h'(k) = k \bmod 7$



double Hashing

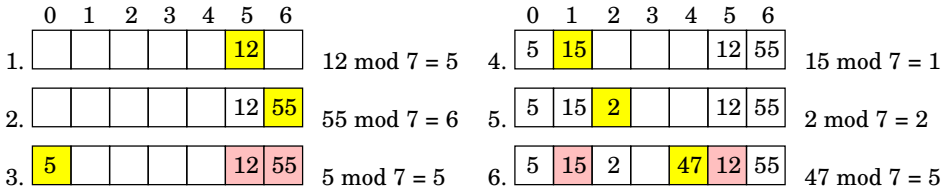
zu zwei gegebenen Hash-Funktionen $h_1(k)$ und $h_2(k)$ sei

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m.$$

Beispiel: betrachte das Einfügen der Schlüssel

$$12, 55, 5, 15, 2, 47$$

für $M = 7$, $h_1(k) = k \bmod 7$ und $h_2 = 1 + k \bmod 5$



Algorithmen & Datenstrukturen

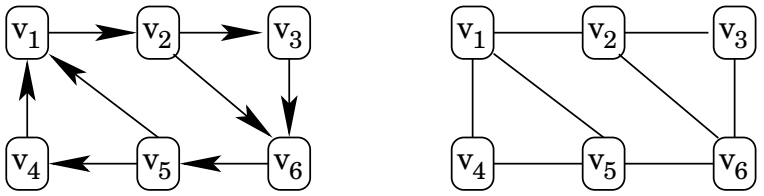
Graphalgorithmen

Grundlagen

Ein **gerichteter Graph** $G = (V, E)$ besteht aus

- einer endlichen Menge von **Knoten** $V = \{v_1, \ldots, v_n\}$ und
- einer Menge von gerichteten **Kanten** $E \subseteq V \times V$

Bei einem **ungerichteten Graphen** $G = (V, E)$ sind die Kanten ungeordnete Paare: $E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$



gerichteter Graph

ungerichteter Graph

Motivation

Graphen verwenden, wo Sachverhalt darstellbar ist durch

- eine Menge von Objekten (Entitäten)
- Beziehungen zwischen den Objekten

Beispiele:

- **Routenplanung:** Städte sind durch Straßen verbunden
- **Kursplanung:** Kurse setzen andere Kurse voraus
- **Produktionsplanung:** Produkte werden aus Teilen zusammengesetzt
- **Schaltkreisanalyse:** Bauteile sind durch elektrische Leitungen verbunden
- **Spiele:** Objekte: Status, Beziehungen: Spielzüge

Begriffe: gerichtete Graphen

Sei $G = (V, E)$ ein gerichteter Graph und $u, v \in V$.

- Sei $e = (u, v)$ eine Kante. Knoten u ist der **Startknoten**, v der **Endknoten** von e. Die Knoten u und v sind **adjazent**, Knoten u bzw. v und Kante e sind **inzident**.
- $indeg(u)$: **Eingangsgrad** von u, Anzahl der in u einlaufenden Kanten. $outdeg(u)$: **Ausgangsgrad** von u, Anzahl der aus u auslaufenden Kanten.
- Ein Graph $G' = (V', E')$ ist ein **Teilgraph** von G, geschrieben $G' \subseteq G$, falls $V' \subseteq V$ und $E' \subseteq E$.
- $p = (v_0, v_1, \ldots, v_k)$ ist ein **gerichteter Weg** in G der Länge k von Knoten u nach Knoten w, falls gilt: $v_0 = u$, $v_k = w$ und $(v_{i-1}, v_i) \in E$ für $1 \leq i \leq k$.

  p ist **einfach**, wenn kein Knoten mehrfach vorkommt.

Begriffe: ungerichtete Graphen

Sei $G = (V, E)$ ein ungerichteter Graph und $u, v \in V$.

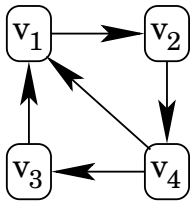
- Sei $e = \{u, v\}$ eine Kante. Die Knoten u und v sind **adjazent**, Knoten u bzw. v und Kante e sind **inzident**. Knoten u und v sind die **Endknoten** der Kante e.
- Der **Knotengrad** von u, geschrieben $deg(u)$, ist die Anzahl der zu u inzidenten Kanten.
- Ein Graph $G' = (V', E')$ ist ein **Teilgraph** von G, geschrieben $G' \subseteq G$, falls $V' \subseteq V$ und $E' \subseteq E$.
- $p = (v_0, v_1, \ldots, v_k)$ ist ein **ungerichteter Weg** in G der Länge k von Knoten u nach Knoten w, falls gilt: $v_0 = u$, $v_k = w$ und $\{v_{i-1}, v_i\} \in E$ für $1 \leq i \leq k$.

Speicherung von Graphen: Adjazenz-Matrix

Sei n die Anzahl der Knoten in Graph $G = (V, E)$. Die Adjazenz-Matrix für G ist eine $n \times n$-Matrix $A_G = (a_{ij})$ mit

$$a_{ij} = \begin{cases} 0, & \text{falls } (v_i, v_j) \notin E, \\ 1, & \text{falls } (v_i, v_j) \in E. \end{cases}$$

Beispiel:



gerichtet:

A	1	2	3	4
1	0	1	0	0
2	0	0	0	1
3	1	0	0	0
4	1	0	1	0

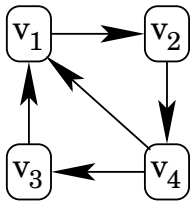
ungerichtet:

A	1	2	3	4
1	0	1	1	1
2	1	0	0	1
3	1	0	0	1
4	1	1	1	0

Speicherung von Graphen: Adjazenz-Liste

Für jeden Knoten v eines Graphen $G = (V, E)$ werden in einer doppelt verketteten Liste $Adj[v]$ alle von v ausgehenden Kanten gespeichert.

Beispiel:



gerichtet:

$Adj[v_1] = \{v_2\}$
$Adj[v_2] = \{v_4\}$
$Adj[v_3] = \{v_1\}$
$Adj[v_4] = \{v_1, v_3\}$

ungerichtet:

$Adj[v_1] = \{v_2, v_3, v_4\}$
$Adj[v_2] = \{v_1, v_4\}$
$Adj[v_3] = \{v_1, v_4\}$
$Adj[v_4] = \{v_1, v_2, v_3\}$

Speicherung von Graphen: Vergleich

Sei $G = (V, E)$ ein Graph mit n Knoten und m Kanten. Zur Speicherung von G wird folgender Platz benötigt:

- Adjazenz-Matrix: $O(n^2)$, geeignet für **dichte** Graphen
- Adjazenz-Liste: $O(n + m)$

Adjazenz-Matrizen unterstützen sehr gut Aufgaben wie: Falls Knoten u und v adjazent sind, tue ...

Adjazenz-Listen unterstützen sehr gut das Verfolgen von Kanten (für alle zu Knoten u inzidenten Kanten tue ...)

Hinzufügen und Löschen von Knoten werden nicht unterstützt. Es gibt voll dynamische Datenstrukturen ...

Durchsuchen von gerichteten Graphen

Aufgabe: Die Suche soll alle von einem gegebenen Startknoten s aus erreichbaren Knoten finden.

Sei D eine Datenstruktur zum Speichern von Kanten.

Algorithmus:

markiere alle Knoten als „unbesucht“
markiere den Startknoten s als „besucht“
füge alle aus s auslaufenden Kanten zu D hinzu
solange D nicht leer ist:
    entnehme eine Kante (u, v) aus D
    falls der Knoten v als „unbesucht“ markiert ist:
        markiere Knoten v als „besucht“
        füge alle aus v auslaufenden Kanten zu D hinzu

Durchsuchen von gerichteten Graphen (2)

Anmerkung: In der Datenstruktur D speichern wir diejenigen Kanten, von denen vielleicht noch unbesuchte Knoten erreicht werden können.

Laufzeit:

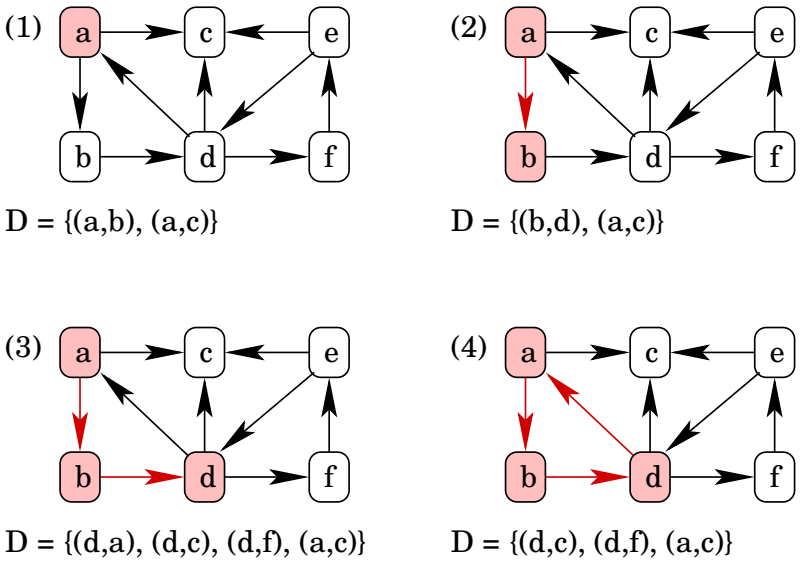
- jede Kante wird höchstens einmal in D eingefügt
- jeder Knoten wird höchstens einmal inspiziert

$\Rightarrow$ Laufzeit proportional zur Anzahl der vom Startknoten aus erreichbaren Knoten und Kanten, also $O(|V| + |E|)$

Typ der Datenstruktur bestimmt die **Durchlaufordnung:**

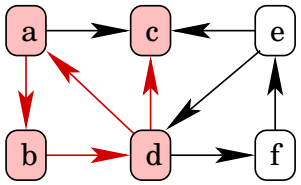
- Stack (last in, first out): **Tiefensuche**
- Liste (first in, first out): **Breitensuche**

Tiefensuche



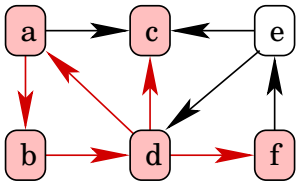
Tiefensuche (2)

(5)



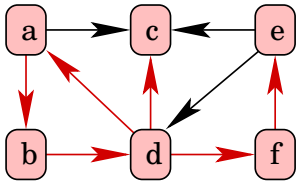
D = {(d,f), (a,c)}

(6)



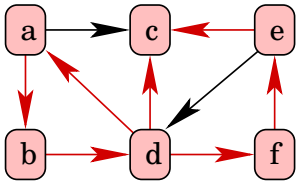
D = {(f,e), (a,c)}

(7)



D = {(e,c), (e,d), (a,c)}

(8)



D = {(e,d), (a,c)} . . .

Tiefensuche (3)

- **Baumkante:** Kante, der die Tiefensuche folgt
- **Vorwärtskante:** Kante $(u, v) \in E$ mit dfb[v] $>$ dfb[u], die keine Baumkante ist
- **Querkante:** Kante $(u, v) \in E$ mit dfb[v] $<$ dfb[u] und dfe[v] $<$ dfe[u]
- **Rückwärtskante:** Kante $(u, v) \in E$ mit dfb[v] $<$ dfb[u] und dfe[v] $>$ dfe[u]

Rekursive Tiefensuche:

```
markiere alle Knoten als „unbesucht“
dfbZähler := 0
dfeZähler := 0
tiefensuche(s)
```

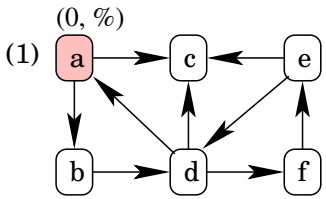

Tiefensuche (4)

tiefensuche(u):

```
markiere u als „besucht“
dfb[u] := dfbZähler
dfbZähler := dfbZähler + 1
betrachte alle Kanten (u, v) ∈ E:
      falls Knoten v als „unbesucht“ markiert ist:
            tiefensuche(v)
dfe[u] := dfeZähler
dfeZähler := dfeZähler + 1
```

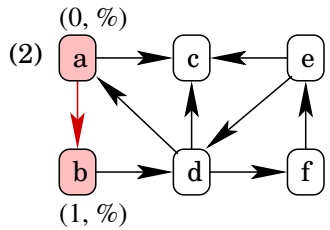

Tiefensuche (5)

(1)



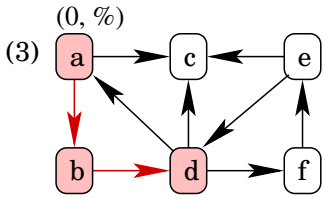
D = {(a,b), (a,c)}

(2)



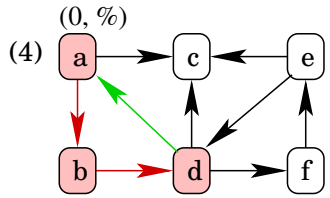
D = {(b,d), (a,c)}

(3)



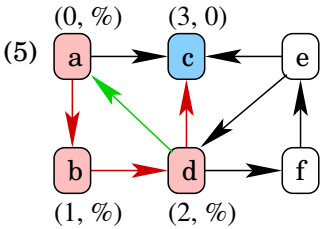
D = {(d,a), (d,c), (d,f), (a,c)}

(4)

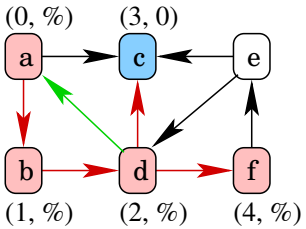


D = {(d,c), (d,f), (a,c)}

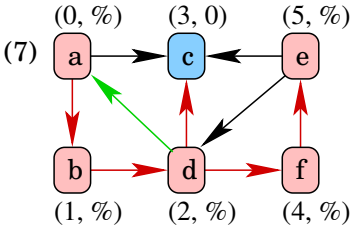
Tiefensuche (6)

(5)


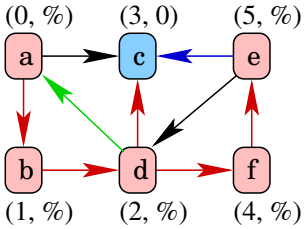
D = {(d,f), (a,c)}

(6)


D = {(f,e), (a,c)}

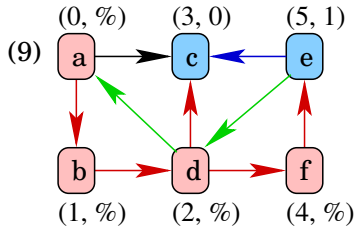
(7)


D = {(e,c), (e,d), (a,c)}

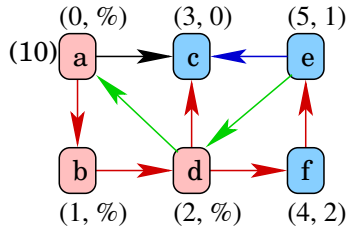
(8)


D = {(e,d), (a,c)}

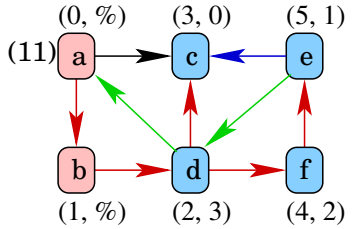
Tiefensuche (7)

(9)


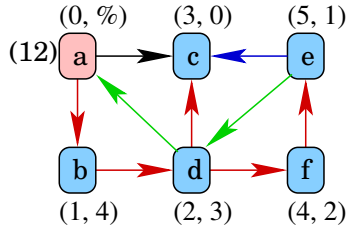
D = {(a,c)}

(10)


D = {(a,c)}

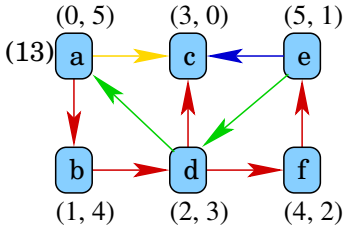
(11)


D = {(a,c)}

(12)


D = {(a,c)}

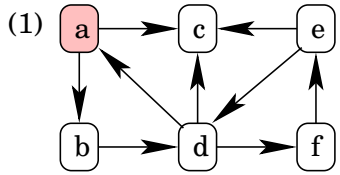
Tiefensuche (8)

(13)


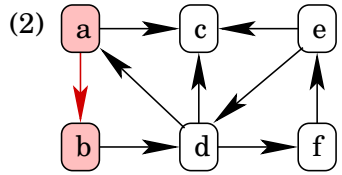
D = {}

- rot markierte Kanten: Baumkanten
- grün markierte Kanten: Rückwärtskanten
- blau markierte Kanten: Querkanten
- gelb markierte Kanten: Vorwärtskanten

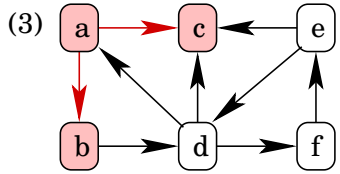
Breitensuche

(1)


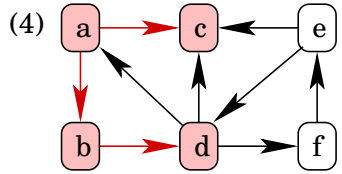
D = {(a,b), (a,c)}

(2)


D = {(a,c), (b,d)}

(3)


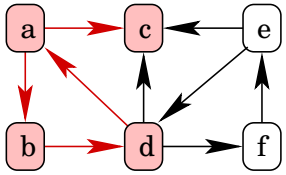
D = {(b,d)}

(4)


D = {(d,a), (d,c), (d,f)}

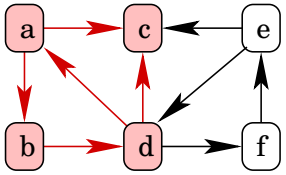
Breitensuche (2)

(5)



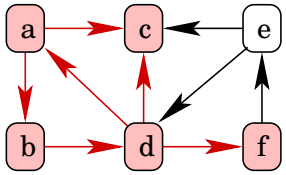
D = {(d,c), (d,f)}

(6)



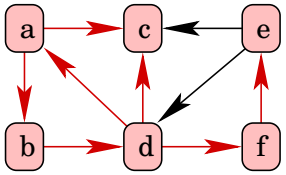
D = {(d,f)}

(7)



D = {(f,e)}

(8)



D = {(e,c), (e,d)} . . .

Durchsuchen von ungerichteten Graphen

Obige Suche funktioniert mit leichter Modifikation auch für ungerichtete Graphen.

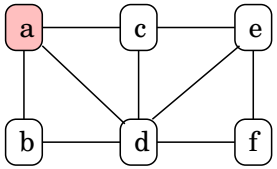
Sei D eine Datenstruktur zum Speichern von Kanten.

Algorithmus:

markiere alle Knoten als „unbesucht“
markiere den Startknoten s als „besucht“
füge alle mit s inzidenten Kanten zu D hinzu
solange D nicht leer ist:
    entnehme eine Kante $\{u, v\}$ aus D
    falls der Knoten u/v als „unbesucht“ markiert ist:
        markiere Knoten u/v als „besucht“
        füge alle zu u/v inzidenten Kanten zu D hinzu

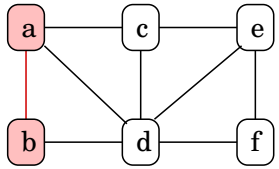
Breitensuche: ungerichtet

(1)



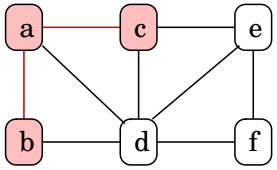
D = {{a,b}, {a,c}, {a,d}}

(2)



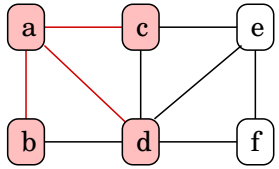
D = {{a,c}, {a,d}, {b,d}}

(3)



D = {{a,d}, {b,d}, {c,d}, {c,e}}

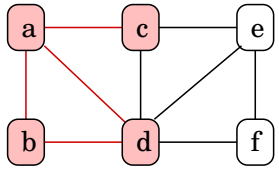
(4)



D = {{b,d}, {c,d}, {c,e}, {d,e}, {d,f}}

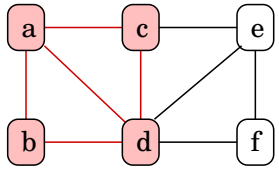
Breitensuche: ungerichtet (2)

(5)



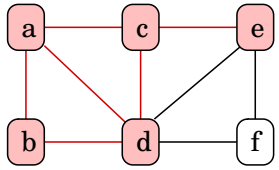
D = {{c,d}, {c,e}, {d,e}, {d,f}}

(6)



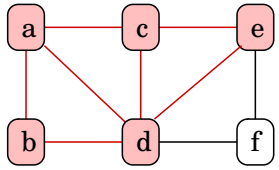
D = {{c,e}, {d,e}, {d,f}}

(7)



D = {{d,e}, {d,f}, {e,f}}

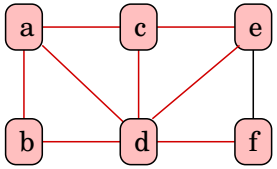
(8)



D = {{d,f}, {e,f}}

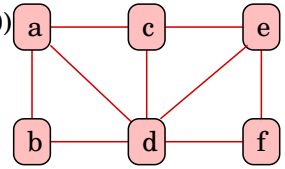
Breitensuche: ungerichtet (3)

(9)



D = {{e,f}}

(10)



D = {}

Minimale Spannbäume

Definition:

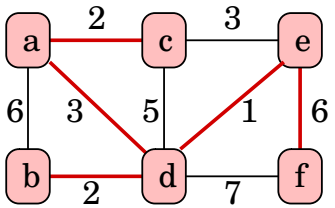
- ein ungerichteter Graph $G = (V, E)$ heißt **zusammenhängend**, wenn gilt: für alle $u, v \in V$ existiert ein Weg von u nach v.
- ein **Spannbaum** T von $G = (V, E)$ ist ein zusammenhängender Teilgraph $T = (V, E')$ von G mit $|V| - 1$ Kanten

gegeben: ein ungerichteter, zusammenhängender Graph $G = (V, E)$ mit Kostenfunktion $c : E \to \mathbb{R}^+$

gesucht: ein Spannbaum $T = (V, E')$ von G mit minimalen Kosten:

$$c(T) = \sum_{e \in E} c(e)$$

Minimale Spannbäume: Beispiel



Algorithmen:

- Kruskal: basiert auf Union-Find Datenstruktur, Laufzeit $O(|E| \cdot \log |V|)$
- Karger, Klein, Tarjan [1]: zur Zeit bester Algorithmus, randomisiert, erwartete Laufzeit $O(|V| + |E|)$

---

[1] A randomized linear-time algorithm to find minimum spanning trees. Journal of the ACM, 1995.

Minimale Spannbäume: Motivation

Verkabelung und Routing

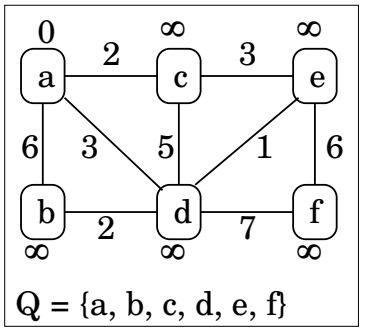
- alle Häuser ans Telefonnetz anschließen: aus Kostengründen möglichst wenig Kabel verlegen
- Stromversorgung von elektrischen Bauteilen auf einer Platine
- Routing:
  * CISCO IP Multicast
  * Spanning Tree Protocol
- es werden nur die Straßen repariert, so dass nach wie vor alle Häuser erreichbar sind

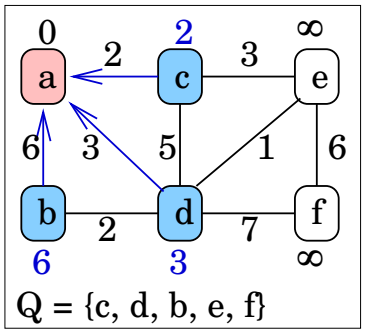
Prims Algorithmus

Sei Q eine Datenstruktur zum Speichern von Knoten. Die Knoten sind mit den Kantenwerten gewichtet.

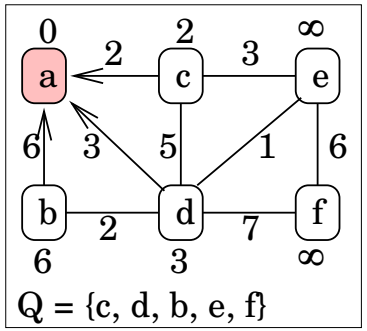
$Q := V$
$key[v] := \infty$ for all $v \in V$
$key[s] := 0$ for some arbitrary $s \in V$
while $Q \neq \emptyset$ do
    $u :=$ ExtractMin(Q)
    for all $v \in Adj(u)$ do
        if $v \in Q$ and $c((u,v)) < key[v]$
        then $key[v] := c((u,v))$
            $\pi[v] := u$

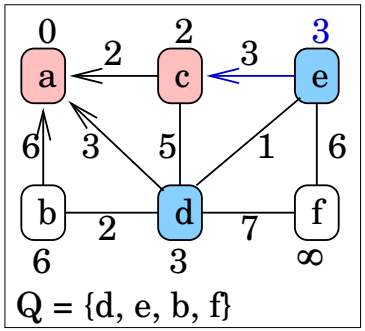
Prims Algorithmus: Beispiel (a)



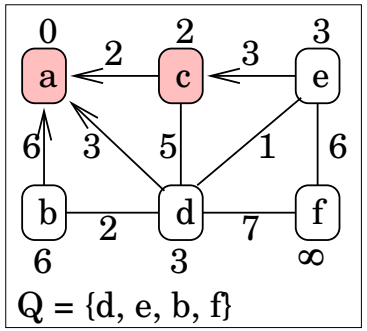


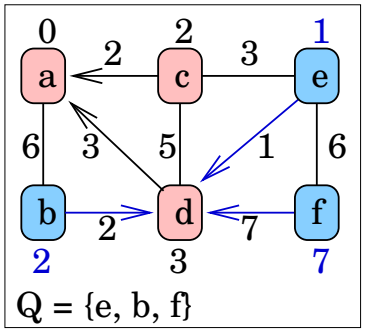
Prims Algorithmus: Beispiel (b)



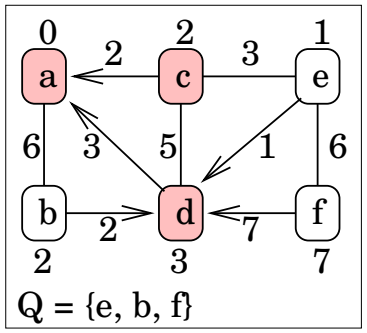


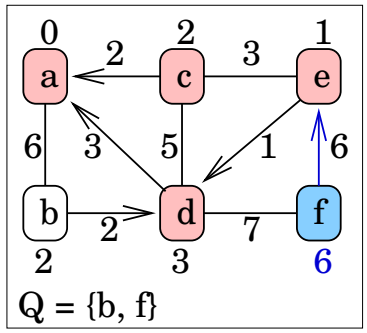
Prims Algorithmus: Beispiel (c)



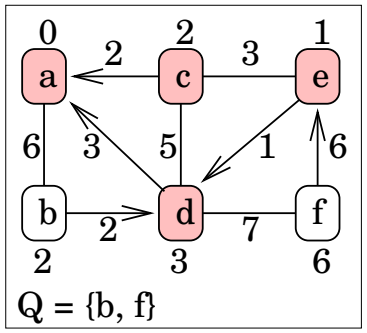


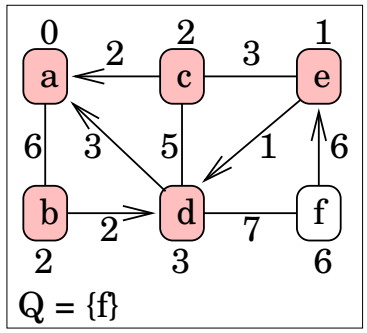
Prims Algorithmus: Beispiel (d)



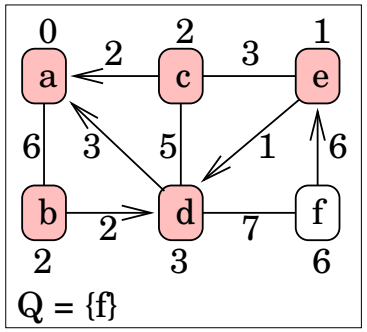


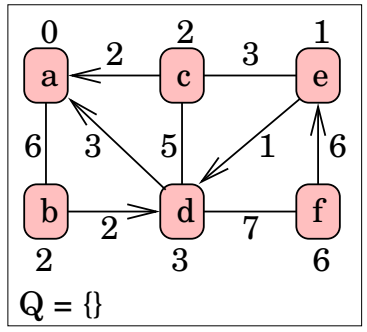
Prims Algorithmus: Beispiel (e)





Prims Algorithmus: Beispiel (f)





Prims Algorithmus: Bewertung

Laufzeit:

$$T = \Theta(V) \cdot T_{ExtractMin} + \Theta(E) \cdot T_{DecreaseKey}$$

Wird die Datenstruktur Q mittels Array implementiert:

- $T_{ExtractMin} \in O(V)$
- $T_{DecreaseKey} \in O(1)$

$\Rightarrow$ Laufzeit $O(V^2)$

Anmerkungen: implementiere Q als

- Binär-Heap: Laufzeit $O(E \log V)$
- Fibbonacci-Heap: Laufzeit $O(E + V \log V)$

Kürzeste Wege

gegeben: ein ungerichteter, zusammenhängender Graph $G = (V, E)$ mit Kostenfunktion $c : E \to \mathbb{R}^+$

single source shortest paths:
Suche von einem gegebenen Knoten $s \in V$ die kürzesten Wege zu allen anderen Knoten.

all pairs shortest paths:
Suche für jedes Paar $(u, v) \in V^2$ den kürzesten Weg von u nach v.

Wir betrachten hier nur die erste Variante.

Kürzeste Wege: Motivation

Reiseplanung: Finde den kürzesten Weg zum Urlaubsort.

Kostenminimierung: Finde Zugverbindung

- mit möglichst kurzer Reisezeit,
- bei der man möglichst wenig umsteigen muss, oder
- die möglichst preiswert ist.

$\Rightarrow$ z.B. Fahrplanberechnung der Deutschen Bahn AG

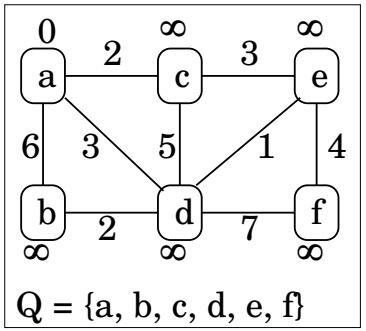
Routing im Internet: OSPF (Open Shortest Path First)

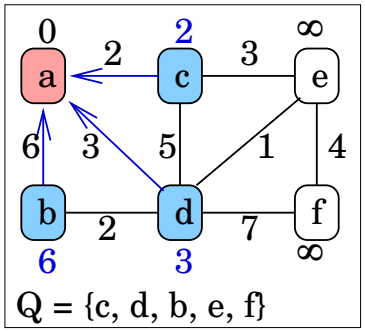
Dijkstras Algorithmus

Sei Q eine Datenstruktur zum Speichern von Knoten. Die Knoten sind mit den Kantenwerten gewichtet.

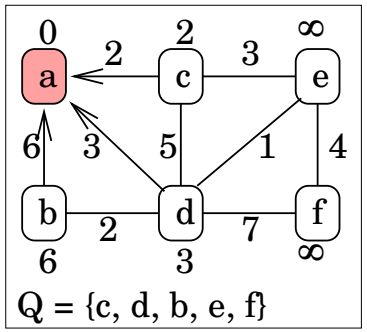
```
Q := V
d[v] := ∞ for all v ∈ V
d[s] := 0 for some arbitrary s ∈ V
while Q ≠ ∅ do
        u := ExtractMin(Q)
        for all v ∈ Adj(u) do
                if v ∈ Q and d[v] > d[u] + c((u,v))
                then d[v] := d[u] + c((u,v))
                        π[v] := u
```

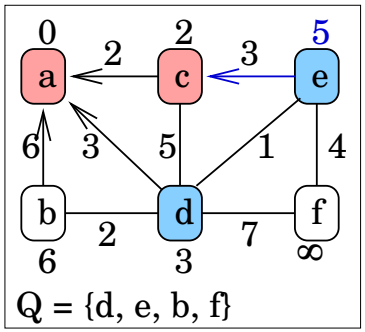

Dijkstras Algorithmus: Beispiel (a)



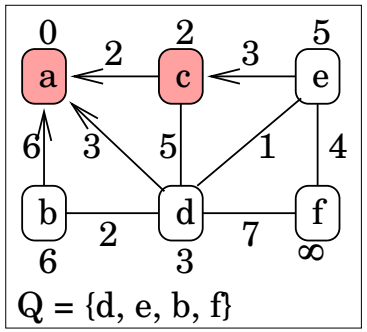


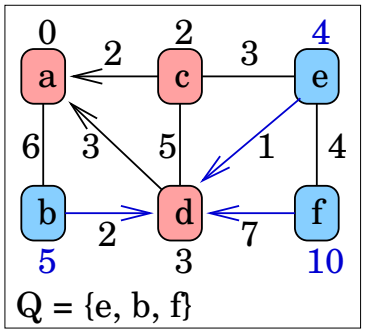
Dijkstras Algorithmus: Beispiel (b)



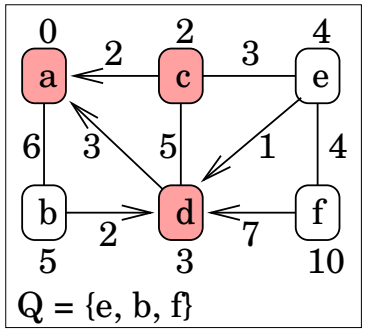


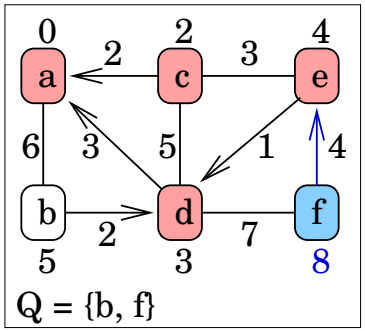
Dijkstras Algorithmus: Beispiel (c)



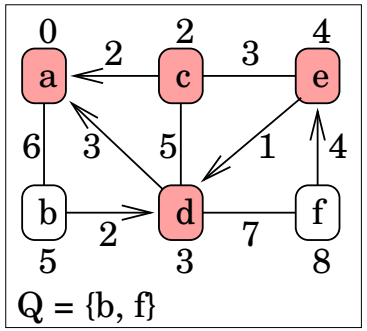


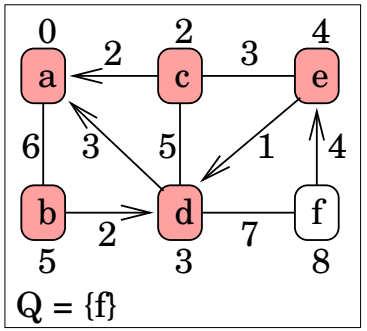
Dijkstras Algorithmus: Beispiel (d)



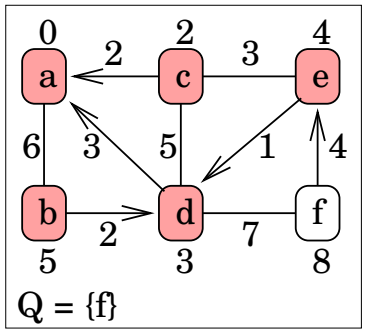


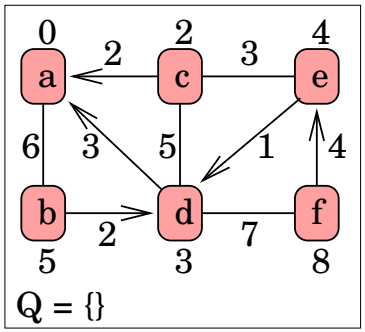
Dijkstras Algorithmus: Beispiel (e)





Dijkstras Algorithmus: Beispiel (f)





Dijkstras Algorithmus: Bewertung

Laufzeit:

$$T = \Theta(V) \cdot T_{ExtractMin} + \Theta(E) \cdot T_{DecreaseKey}$$

Gleiche Laufzeit wie bei Prims Algorithmus!
Implementiere Q als

- Array: Laufzeit $O(V^2)$
- Binär-Heap: Laufzeit $O(E \log V)$
- Fibbonacci-Heap: Laufzeit $O(E + V \log V)$

Ungewichtete Graphen: modifizierte Breitensuche

Algorithmen & Datenstrukturen

Datenstrukturen: Einführung

Die richtige Organisationsform einer Menge von Daten hat erheblichen Einfluss darauf, wie effizient sich bestimmte Operationen ausführen lassen! → Prims Algorithmus

Aufgabe: Finde Speicherungsform, so dass die Operationen möglichst effizient ausführbar sind.

Wesentliche Einflussgrößen:

- Effizienz bzgl. Laufzeit, Speicherbedarf oder einfache Programmierbarkeit???
- Ist die Folge der Operationen bekannt? oder:
  Sind relative Häufigkeiten der Operationen bekannt?

Datenstrukturen: Einführung (2)

Gesucht: Datenstruktur für einfügen, suchen, entfernen

Implementierung 1: unsortiertes Array

- einfügen: am Ende anhängen $\rightarrow O(1)$
- suchen: lineare Suche $\rightarrow O(n)$
- entfernen: finden und tauschen $\rightarrow O(n)$

Implementierung 2: sortiertes Array

- einfügen: Position finden und verschieben $\rightarrow O(n)$
- suchen: binäre Suche $\rightarrow O(\log(n))$
- entfernen: finden und verschieben $\rightarrow O(n)$

Datenstrukturen: Einführung (3)

Einfluss der relativen Häufigkeiten der Operationen auf die Laufzeit bei unterschiedlichen Implementierungen:
(Annahme: Belegung sei m, m viel größer als n)

- einfügen/suchen = 1/n

  unsortiert: $1 \cdot O(1) + n \cdot O(m) \rightarrow O(n \cdot m)$
  sortiert: $1 \cdot O(m) + n \cdot O(\log(m)) \rightarrow O(n \log(m))$

- einfügen/suchen = 1/1

  unsortiert: $1 \cdot O(1) + 1 \cdot O(m) \rightarrow O(m)$
  sortiert: $1 \cdot O(m) + 1 \cdot O(\log(m)) \rightarrow O(m)$

- einfügen/suchen = n/1

  unsortiert: $n \cdot O(1) + 1 \cdot O(m) \rightarrow O(m)$
  sortiert: $n \cdot O(m) + 1 \cdot O(\log(m)) \rightarrow O(n \cdot m)$

Datenstrukturen: Einführung (4)

Ein **abstrakter Datentyp** besteht aus einer oder mehreren Mengen von Objekten und darauf definierten Operationen.

Beispiel: Polynom

- Objektmenge: Polynome mit ganzzahligen Koeffizienten
- Operationen: Addition, Multiplikation, erste Ableitung

Beispiel: Landkarte

- Objektmenge: Städte, Wege zwischen den Städten
- Operationen: nächster Nachbar, Distanz zweier Städte

Daten und Operationen auf den Daten gehören zusammen! → modulare Programmierung

Datenstrukturen: Einführung (5)

Warum **abstrakter** Datentyp? Die Semantik (Bedeutung) der Operationen kann unabhängig von der Realisierung axiomatisch definiert werden.

Beispiel: Definition eines Stack (last in first out)

- empty(init()) = true
- empty(push(s,e)) = false
- pop(push(s,e)) = s
- pop(init()) = undefiniert
- top(push(s,e)) = e
- top(init()) = undefiniert

→ Mathematischer Hintergrund: Algebra

Datenstrukturen: Einführung (6)

Die Axiome definieren das Zusammenspiel der Daten, nicht die Implementierung der Operationen.

Beispiel: Kalenderdatum

- differenz(d, addiere(d,n)) = n
- wochentag(d) = wochentag(addiere(d,7))

Nur Kenntnis der Axiome notwendig, um den Datentyp anwenden zu können. Implementierung kann also jederzeit geändert werden, muss aber die Axiome garantieren.

- C/C++: Header-Datei
- Java: Interface

Schnittstellendefinition wichtig für große Projekte!

Datenstrukturen: Einführung (7)

Unterscheidung:

- **Datentyp:** die vorhandenen Grundtypen (`int`, `float`, ...) und die daraus mittels Strukturierungsmethoden (`union`, `struct`, ...) gebildeten zusammengesetzten Typen.
- **abstrakter Datentyp:** besteht aus einer oder mehreren, mit mathematischen Methoden festgelegten Mengen von Objekten und darauf definierten Operationen.
- **Datenstruktur:** konkrete Implementierung der Objektmengen eines abstrakten Datentyps.

Datenstruktur: Baum

- in der Informatik weit verbreitet: Entscheidungsbäume, Syntaxbäume, Ableitungsbäume, ...
- gehören zu den wichtigsten Datenstrukturen:
  * spannende Bäume
  * Suchbäume
  * Verzeichnisbäume (hierarchische Dateisysteme)
  * hierarchische Datenbanksysteme (LDAP)
- verallgemeinerte Liste: ein Element (**Knoten**) hat nicht nur einen Nachfolger (lineare Liste), sondern eine endliche, begrenzte Anzahl Nachfolger (**Kinder**)

Datenstruktur: Baum (2)

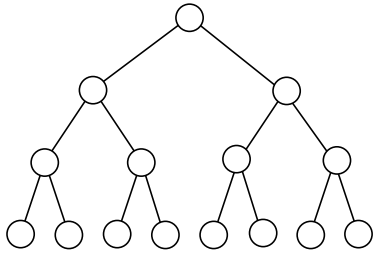
Begriffe:

- der Knoten ohne Vorgänger (**Elter**) heißt **Wurzel**
- Knoten ohne Nachfolger heißen **Blätter**
- **innere Knoten:** Knoten $\neq$ Blätter
- **Binärbaum:** Baum, bei dem jeder innere Knoten genau zwei Kinder hat (allgemein: k-näre Bäume)
- **Tiefe eines Knotens:** Abstand des Knotens von der Wurzel (Anzahl der Kanten)
- **Höhe des Baumes:** maximale Abstand eines Blattes von der Wurzel (Anzahl der Kanten)

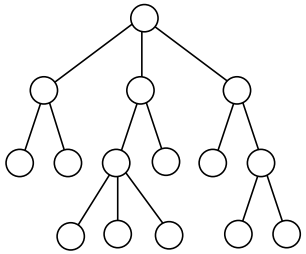
Datenstruktur: Baum (3)

Begriffe: (Fortsetzung)

- auf den Nachfolgern kann eine **Ordnung** bestehen: linker/rechter Nachfolger, i-ter Nachfolger, ...
- **vollständiger Baum**: in jeder Ebene maximale Anzahl Knoten und alle Blätter haben gleiche Tiefe



vollständiger Binärbaum

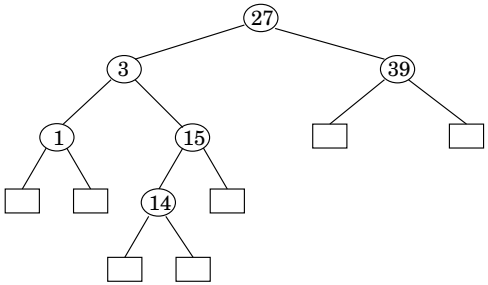


ternärer Baum

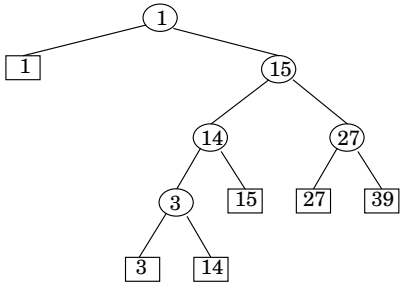
Suchbäume

Suchbäume sind geordnete binäre Bäume. Unterscheidung:

- **Suchbäume:** innere Knoten speichern Schlüsselwerte
- **Blattsuchbäume:** Blätter speichern Schlüsselwerte, innere Knoten nur Wegweiser (max. Wert des linken Teilbaums)



Suchbaum



Blattsuchbaum

Suchbäume (2)

Suche eines Schlüssels k in einem Suchbaum:

1. beginne bei der Wurzel p
2. vergleiche k mit dem bei p gespeicherten Schlüssel k_p
   - $k < k_p$: setze Suche mit linkem Nachfolger von p fort
   - $k > k_p$: setze Suche mit dem rechten Kind von p fort

$\Rightarrow$ wird ein Blatt erreicht, ist k nicht im Baum gespeichert

Einfügen eines Schlüssels k in einen Suchbaum:

1. suche den Schlüssel k im Suchbaum
2. falls k nicht im Baum enthalten ist, endet die Suche bei einem Blatt: füge k ein und erzeuge zwei neue Blätter

Suchbäume (3)

Problem: Die Form des Suchbaums (also die Laufzeit der Suchoperation) hängt stark von der Einfügereihenfolge ab!

Lösung: AVL-Bäume, Rot/Schwarz-Bäume, B/B*-Bäume

Implementierung der Suchbäume: Übungsaufgabe

Suchbäume (4)

Entfernen eines Schlüssels k aus einem Suchbaum:

1. suche den Schlüssel k im Suchbaum (Knoten sei u)
2. unterscheide anhand der Nachfolger von u:
   - beide Nachfolger sind Blätter: mache u zu einem Blatt
   - nur ein Nachfolger ist Blatt: hänge inneren Knoten an den Elter von u
   - beide Nachfolger sind innere Knoten: suche im rechten Teilbaum von u den kleinsten Schlüssel $k' > k$. Der Knoten v, der k' speichert, heißt symmetrischer Nachfolger von u. Ersetze k durch k' und entferne Knoten v