Datenstrukturen und Algorithmen

Peter Rossmanith

Theoretische Informatik, RWTH Aachen

26. Juli 2011

RWTH AACHEN

Organisatorisches

Vorlesung:
Peter Rossmanith (Raum 6113)

Sprechstunde: Mittwoch 11:00–12:00

Übung:
Alexander Langer, Felix Reidl
`langer@cs.rwth-aachen.de`
`reidl@cs.rwth-aachen.de`

RWTH AACHEN

Organisatorisches

Vorlesung: Dienstag und Freitag, 14:00 Uhr, Audimax

Globalübung: Montag, 14:00 Uhr, AH IV

(Manchmal muß Vorlesung und Übung getauscht werden)

Tutorübungen: Montag, Dienstag, Mittwoch (ab 11.04.2011)

Anmeldung ab 7.4., 16 Uhr, über
`https://aprove.informatik.rwth-aachen.de/`

Ausgabe des Übungsblatts in den Tutorübungen

Alle Materialen auf Webseite
`tcs.rwth-aachen.de/lehre/DA/SS2011/`

RWTH AACHEN

Organisatorisches

Klausur:

02.08.2011, zwischen 8 und 12 Uhr
12.09.2011, zwischen 12 und 15 Uhr

Anmeldung: zur "Prüfung Datenstrukturen und Algorithmen"
Abmeldung: bis letzter Freitag im Mai (27.5.)

Zulassungskriterien:

50% der Punkte aus den Hausaufgaben (ab 11.04.2011)

50% der Punkte aus selbstständigen Präsenzübungen während der Tutorübungen (ab 18.04.2011)

Abgabe der Hausaufgaben in Gruppen von bis zu vier Personen

RWTH AACHEN

Einordnung

Datenstrukturen und Algorithmen baut auf diesen Vorlesungen auf:

- Programmierung
- Diskrete Strukturen
- Analysis
- Stochastik

Einordnung

Vorlesungen, die Datenstrukturen und Algorithmen ergänzen:

- Berechenbarkeit und Komplexität
- Formale Systeme, Automaten, Prozesse
- Numerisches Rechnen
- Effiziente Algorithmen

Drei sehr umfassende Bücher

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein.
  Introduction to Algorithms.
  The MIT Press, 2d edition, 2001.
- T. Ottman and P. Widmayer.
  Algorithmen und Datenstrukturen.
  Spektrum Verlag, 2002.
- K. Mehlhorn and P. Sanders.
  Algorithms and Data Structures: The Basic Toolbox.
  Springer, 2008.

Weitere interessante Bücher

- A. V. Aho, J. E. Hopcroft, and J. D. Ullman.
  The Design and Analysis of Computer Algorithms.
  Addison-Wesley, 1974.
- S. Skiena.
  The Algorithm Design Manual.
  Telos, 1997.

Bücher für die Analyse von Algorithmen

- R. L. Graham, D. E. Knuth, and O. Patashnik.
  Concrete Mathematics.
  Addison-Wesley, 1989.
- R. Sedgewick and P. Flajolet.
  An Introduction to the Analysis of Algorithms.
  Addison-Weseley Publishing Company, 1996.

Bücher für die Analyse von Algorithmen

- A. Steger.
  Diskrete Strukturen I. Kombinatorik, Graphentheorie, Algebra.
  Springer-Verlag, 2001.
- T. Schickinger und A. Steger.
  Diskrete Strukturen II. Wahrscheinlichkeitsrechnung und Statistik.
  Springer-Verlag, 2001.

Speziellere Bücher

- Warren S. H.
  Hacker's Delight.
  Addison-Wesley Longman, 2002.
- J. L. Hennessy and D. A. Patterson.
  Computer Architecture: A Quantitative Approach.
  The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufman Publishers, 3rd edition, 1990.

Die Klassiker

- D. E. Knuth.
  Seminumerical Algorithms, volume 2 of *The Art of Computer Programming*.
  Addison-Wesley, 2nd edition, 1969.
- D. E. Knuth.
  Fundamental Algorithms, volume 1 of *The Art of Computer Programming*.
  Addison-Wesley, 2d edition, 1973.
- D. E. Knuth.
  Sorting and Searching, volume 3 of *The Art of Computer Programming*.
  Addison-Wesley, 1973.

Komplexität von Algorithmen

Eine zentrale Frage:

Zwei verschiedene Algorithmen lösen dasselbe Problem. Welcher ist schneller?

Unterscheide:

1. Die Laufzeit eines konkreten Algorithmus
2. Die Geschwindigkeit mit der sich ein Problem lösen läßt
   → Komplexitätstheorie

Komplexität von Algorithmen

Gegeben: Algorithmus A und Algorithmus B

Wir wählen eine bestimmte Eingabe.

Ergebnis:

- Algorithmus A benötigt 10 Sekunden
- Algorithmus B benötigt 15 Sekunden

Ist Algorithmus A schneller? Er ist auf dieser Eingabe schneller.

Es gibt aber viele andere mögliche Eingaben.

Worst-Case–Komplexität von Algorithmen

Lösung:

Die Laufzeit eines Algorithmus ist nicht eine Zahl, sondern eine Funktion $\mathbf{N} \to \mathbf{N}$.

Sie gibt die Laufzeit des Algorithmus für jede Eingabelänge an.

Die Laufzeit für Eingabelänge n ist die schlechteste Laufzeit aus allen Eingaben mit Länge n.

Wir nennen dies die Worst-Case–Laufzeit.

Sequentielle Berechenbarkeitshypothese

Church'sche These:

Die Menge der algorithmisch lösbaren Probleme ist für alle vernünftigen Berechnungsmodelle identisch.

→ Vorlesung Berechenbarkeit und Komplexität

„Theorem“
Sequentielle Berechenbarkeitshypothese:

Die benötigte Laufzeit für ein Problem auf zwei unterschiedlichen sequentiellen Berechenbarkeitsmodellen unterscheidet sich nur um ein Polynom.

Ein beliebiges Polynom ist für uns zu grob!

Die RAM (Random Access Machine)

Für die Laufzeitanalyse legen wir uns auf ein Modell fest:
Die Random Access Machine (RAM).

- Einem normalen von Neumann–Computer ähnlich
- Ein sehr einfaches Modell
- Algorithmus auf RAM und Computer: Anzahl der Anweisungen unterscheidet sich nur um konstanten Faktor

Die RAM (Random Access Machine)

Eine RAM besteht aus:

1. Einem Speicher aus Speicherzellen 1, 2, 3,...
2. Jede Speicherzelle kann beliebige Zahlen speichern
3. Der Inhalt einer Speicherzelle kann als Anweisung interpretiert werden
4. Einem Prozessor der einfache Anweisungen ausführt
5. Anweisungen: Logische und Arithmetische Verknüpfungen, (bedingte) Sprünge
6. Jede Anweisung benötigt konstante Zeit

Grenzen des RAM-Modells

- Jede Speicherzelle enthält beliebige Zahlen:
  Unrealistisch, wenn diese sehr groß werden
- Jede Anweisung benötigt gleiche Zeit:
  Unrealistisch, wegen Caches, Sprüngen

Daumenregel: Bei n Speicherzellen, $O(\log n)$ bits verwenden.


```
void quicksort(int N)
{
    int i,j,l,r,k,t;
    l=1; r=N;
    if (N>M)
      while(1) {
        i=l-1; j=r; k=a[j];
        do {
            do { i++; } while(a[i]<k);
            do { j--; } while(k<a[j]);
            t=a[i]; a[i]=a[j]; a[j]=t;
        } while(i<j);
        a[j]=a[i]; a[i]=a[r]; a[r]=t;
        if (r-i ≥ i-l) {
            if (i-l>M) { push(i+1,r); r=i-1; }
            else if (r-i>M) l=i+1;
            else if (stack_is_empty) break;
            else pop(l,r);
        }
        else
            if (r-i>M) { push(l,i-1); l=i+1; }
            else if (i-l>M) r=i-1;
            else if (stack_is_empty) break;
            else pop(l,r);
      }
    for(i=2;i ≤ N;i++)
        if (a[i-1]>a[i]) {
            k=a[i]; j=i;
            do { a[j]=a[j-1]; j--; } while(a[j-1]>k);
            a[j]=k;
        }
}
```

Quicksort – Ein Beispiel

Statt einer Worst-Case–Analyse führen wir eine Average-Case–Analyse durch.

Die Eingabe besteht aus den Zahlen $1, \ldots, n$ in zufälliger Reihenfolge.

Wie lange benötigt Quicksort im Durchschnitt zum Sortieren?

Es sind $O(n \log n)$ Schritte auf einer RAM.

Für eine genauere Analyse, müssen wir das Modell genau festlegen.


```
sw -4(r29),r30
add r30,r0,r29
sw -8(r29),r31
subui r29,r29,#464
sw 0(r29),r2
sw 4(r29),r3
sw 8(r29),r4
sw 12(r29),r5
...
sw 40(r29),r12
lw r11,(r30)
lw r9,4(r30)
slli r1,r11,#0x2
lhi r8,((ﻪ+4)>>16)&0xffff
addui r8,r8,(ﻪ+4)&0xffff
lhi r12,((ﻪ)>>16)&0xffff
addui r12,r12,(ﻪ)&0xffff
add r7,r1,r12
sgt r1,r11,r9
beqz r1,L8
addi r4,r30,#-408
add r10,r0,r4
L11:
addi r3,r8,#-4
L51:
add r2,r0,r7
lw r5,(r7)
L15:
addi r3,r3,#4
L49:
lw r1,(r3)
slt r1,r1,r5
bnez r1,L49
addi r3,r3,#4
addi r3,r3,#-4
addi r2,r2,#-4
L50:
lw r31,(r2)
slt r1,r5,r31
bnez r1,L50
addi r2,r2,#-4
addi r2,r2,#4
lw r6,(r3)
sw (r3),r31
sltu r1,r3,r2
bnez r1,L15
sw (r2),r6
lw r12,(r3)
sw (r2),r12
lw r12,(r7)
sw (r3),r12
sub r1,r7,r3
srai r5,r1,#0x2
sub r1,r3,r8
srai r2,r1,#0x2
sge r1,r5,r2
beqz r1,L24
sw (r7),r6
sgt r1,r2,r9
beqz r1,L25
addi r1,r3,#4
sw (r4),r1
addi r4,r4,#4
sw (r4),r7
addi r4,r4,#4
j L11
addi r7,r3,#-4
L25:
sgt r1,r5,r9
beqz r1,L34
addi r8,r3,#4
j L51
addi r3,r8,#-4
L24:
sgt r1,r5,r9
beqz r1,L32
addi r1,r3,#-4
sw (r4),r8
addi r4,r4,#4
sw (r4),r1
addi r4,r4,#4
j L11
addi r8,r3,#4
L32:
sgt r1,r2,r9
bnez r1,L11
addi r7,r3,#-4
L34:
seq r1,r4,r10
bnez r1,L8
addi r4,r4,#-4
lw r7,(r4)
addi r4,r4,#-4
j L11
lw r8,(r4)
L8:
slli r1,r11,#0x2
lhi r2,((ﻪ)>>16)&0xffff
addui r2,r2,(ﻪ)&0xffff
add r7,r1,r2
addi r3,r2,#8
sleu r1,r3,r7
beqz r1,L40
addi r4,r2,#4
L42:
lw r1,(r4)
lw r2,(r3)
sgt r1,r1,r2
beqz r1,L41
add r5,r0,r2
add r2,r0,r3
addi r31,r3,#-4
L44:
lw r12,(r31)
sw (r2),r12
addi r31,r31,#-4
lw r1,(r31)
sgt r1,r1,r5
bnez r1,L44
addi r2,r2,#-4
sw (r2),r5
L41:
addi r3,r3,#4
sleu r1,r3,r7
bnez r1,L42
addi r4,r4,#4
L40:
lw r2,0(r29)
lw r3,4(r29)
lw r4,8(r29)
...
lw r12,40(r29)
lw r31,-8(r30)
add r29,r0,r30
jr r31
lw r30,-4(r30)
```


Quicksort auf dem DLX-Prozessor

Die Laufzeit im Durchschnitt beträgt

$$10A_N + 4B_N + 2C_N + 3D_N + 3E_N + 2S_N + 3N + 6,$$

Schritte, wobei

$$\begin{aligned}
A_N &= \frac{2N - M}{M + 2} \\
B_N &= \frac{1}{6}(N+1)\left(2H_{N+1} - 2H_{M+2} + 1 - \frac{6}{M+2}\right) + \frac{1}{2} \\
C_N &= N + 1 + 2(N+1)(H_{N+1} - H_{M+2}) \\
D_N &= (N+1)(1 - 2H_{M+1}/(M+2)) \\
E_N &= \frac{1}{6}(N+1)M(M-1)/(M+2) \\
S_N &= (N+1)/(2M+3) - 1.
\end{aligned}$$

und $H_n = 1 + 1/2 + 1/3 + \cdots + 1/n$.

O-Notation

Definition

$$\begin{aligned}
O(f) &= \left\{ g \colon \mathbf{N} \to \mathbf{R} \;\middle|\; \exists c \in \mathbf{R}^+ \, \exists N \in \mathbf{N} \, \forall n \geq N \colon |g(n)| \leq c|f(n)| \right\} \\
\Omega(f) &= \left\{ g \colon \mathbf{N} \to \mathbf{R} \;\middle|\; \exists c \in \mathbf{R}^+ \, \exists N \in \mathbf{N} \, \forall n \geq N \colon |g(n)| \geq c|f(n)| \right\} \\
\Theta(f) &= \left\{ g \colon \mathbf{N} \to \mathbf{R} \;\middle|\; \exists c_1, c_2 \in \mathbf{R}^+ \, \exists N \in \mathbf{N} \, \forall n \geq N \colon \right. \\
&\qquad \left. c_1|f(n)| \leq |g(n)| \leq c_2|f(n)| \right\}.
\end{aligned}$$

Informell:

- $g = O(f) \;\approx\; g$ wächst langsamer als f
- $g = \Omega(f) \;\approx\; g$ wächst schneller als f
- $g = \Theta(f) \;\approx\; g$ wächst so schnell wie f

O-Notation – Alternative Definition

Eine weitere Möglichkeit, $O(f)$, $\Omega(f)$ und $\Theta(f)$ zu definieren, falls $f(n) = 0$ nur für endlich viele n zutrifft.

$$O(f) = \left\{ g : \mathbf{N} \to \mathbf{R} \;\middle|\; \limsup_{n\to\infty} \frac{|g(n)|}{|f(n)|} \text{ existiert} \right\}$$

$$\Omega(f) = \left\{ g : \mathbf{N} \to \mathbf{R} \;\middle|\; \liminf_{n\to\infty} \frac{|g(n)|}{|f(n)|} \neq 0 \right\}$$

$$\Theta(f) = \left\{ g : \mathbf{N} \to \mathbf{R} \;\middle|\; \liminf_{n\to\infty} \frac{|g(n)|}{|f(n)|} \neq 0 \text{ und } \limsup_{n\to\infty} \frac{|g(n)|}{|f(n)|} \text{ existiert} \right\}$$

O-Notation

Das folgende Theorem besagt, daß beide Definitionen für die O-Notation übereinstimmen.

Theorem

Es seien $f, g : \mathbf{N} \to \mathbf{R}$ zwei Funktionen. Es sei nur endlich oft $f(n) = 0$.
Dann existiert der Grenzwert $\limsup_{n\to\infty} |g(n)|/|f(n)|$ genau dann, wenn es Zahlen $c, N > 0$ gibt, so daß $|g(n)| \leq c|f(n)|$ für alle $n \geq N$ gilt.

Beweis.
„$\Longrightarrow$“
$\limsup_{n\to\infty} |g(n)|/|f(n)| = c$
$\Rightarrow c + \epsilon \geq |g(n)|/|f(n)|$ und $f(n) \neq 0$ bis auf endlich viele Ausnahmen.
$\Rightarrow$ ab einem $N \in \mathbf{N}$, gilt $c + \epsilon \geq |g(n)|/|f(n)|$.
Insbesondere gilt dann auch $|g(n)| \leq (c + \epsilon)|f(n)|$.

„$\Longleftarrow$“ Es gebe nun ein $N > 0$ und ein $c > 0$, so daß
$\forall n \geq N$: $|g(n)| \leq c|f(n)|$.
Dann gilt $0 \leq |g(n)|/|f(n)| \leq c$ oder $g(n) = 0$ für alle $n \geq N$.
Es sei $a_n = |g(n)|/|f(n)|$. Diese Folge liegt in $[0, c]$.
Bolzano-Weierstraß $\Rightarrow$ größter Häufungswert.
Dann existiert auch $\limsup_{n\to\infty} |g(n)|/|f(n)|$. □

Theorem

Es seien $f, g : \mathbf{N} \to \mathbf{R}$ zwei Funktionen und $c \in \mathbf{R}$ eine Konstante.

Dann gilt das folgende:

1. $O(cf(n)) = O(f(n))$
2. $O(f(n)) + O(g(n)) = O(|f(n)| + |g(n)|)$
3. $O(f(n) + g(n)) = O(f(n)) + O(g(n))$
4. $O(f(n))O(g(n)) = O(f(n)g(n))$
5. $O(f(n)g(n)) = f(n)O(g(n))$
6. $\sum_{k=1}^{n} O(f(k)) = O(nf(n))$ *falls* $|f(n)|$ *monoton steigt*
7. $f(O(1)) = O(1)$, *falls* $|f(n)|$ *monoton steigt*

Wie beweisen wir eine Gleichung der Form

$$O(f(n)) = O(g(n))$$

oder allgemein eine Gleichung, mit O-Notation auf der linken *und* rechten Seite?

In Wirklichkeit ist es $O(f(n)) \subseteq O(g(n))$.

Wir beweisen:
Für jedes $\hat{f}(n) = O(f(n))$ gilt $\hat{f}(n) = O(g(n))$.

Wir beweisen $O(f(n)) + O(g(n)) = O(|f(n)| + |g(n)|)$:

Beweis.
Sei $\hat{f}(n) = O(f(n))$ und $\hat{g}(n) = O(g(n))$ beliebig.
Dann gilt

$$|\hat{f}(n)| \leq c|f(n)| \text{ und } |\hat{g}(n)| \leq c|g(n)|$$

für ein c und große n.

$$\Rightarrow |\hat{f}(n)| + |\hat{g}(n)| \leq c(|f(n)| + |g(n)|)$$

Daraus folgt $O(|f(n)|) + O(|g(n)|) = O(|f(n) + g(n)|)$.
Es gilt aber $O(f(n)) = O(|f(n)|)$ und $O(g(n)) = O(|g(n)|)$. □

Schätze $\log(n^2 + 3n + 5)$ ab.

Theorem
$f : \mathbf{N} \to \mathbf{R}$ *und* $g : \mathbf{R} \to \mathbf{R}$.

$\lim_{n\to\infty} f(n) = 0$.

$g : \mathbf{R} \to \mathbf{R}$ *in einer Umgebung des Ursprungs* k *mal stetig differenzierbar.*

Dann gilt

$$g(f(n)) = \sum_{i=0}^{k-1} g^{(i)}(0)\frac{f(n)^i}{i!} + O(f(n)^k).$$

$\log(n^2+3n+5) = 2\log(n)+\log(1+3/n+5/n^2) = 2\log(n)+O(1/n)$

Beweis.
Der Satz von Taylor besagt:

$$g(z) = \sum_{i=0}^{k-1} \frac{g^{(i)}(0)}{i!} z^i + R_{k-1}$$

mit

$$R_{k-1} = \frac{g^{(k)}(\xi)}{k!} z^k \text{ für ein } \xi \text{ mit } |\xi| \leq |z|,$$

falls z nahe bei 0.
Für uns heißt das:

$$g(f(n)) = \sum_{i=0}^{k-1} \frac{g^{(i)}(0)}{i!} (f(n))^i + R_{k-1}$$

bis auf endlich viele n, da $n \to \infty$.
Es bleibt zu zeigen, daß $R_{k-1} = O(f(n)^k)$. □

Beweis.

$$R_{k-1} = \frac{g^{(k)}(\xi)}{k!} f(n)^k \text{ für ein } \xi \in [-|f(n)|, |f(n)|],$$

falls z nahe bei 0.

$$\Rightarrow R_{k-1} \leq \max_{\xi \in [-\epsilon, \epsilon]} \frac{g^{(k)}(\xi)}{k!} f(n)^k$$

für ein $\epsilon > 0$ bis auf endlich viele n.

Da $g^{(k)}$ stetig und $[-\epsilon, \epsilon]$ kompakt ist, existiert das Maximum nach dem Satz von Weierstraß.

Also gilt $R_{k-1} = O(f(n)^k)$. □

Beispiele

$$\frac{1}{1+1/n} = 1 + O\left(\frac{1}{n}\right)$$
$$\frac{1}{1+1/n} = 1 - \frac{1}{n} + O\left(\frac{1}{n^2}\right)$$
$$\frac{1}{1+1/n} = 1 - \frac{1}{n} + \frac{1}{n^2} + O\left(\frac{1}{n^3}\right)$$

$$\sqrt{n+1} = \sqrt{n} \cdot \sqrt{1+1/n} = \sqrt{n}(1 + O(1/n)) = \sqrt{n} + O(1/\sqrt{n})$$

Java

Wir verwenden oft Java für Datenstrukturen und Algorithmen.

Die Vorlesung ist aber von der Programmiersprache unabhängig.

Lernziel sind die einzelnen Algorithmen und Datenstrukturen, nicht ihre Umsetzung in Java.

Für das Verständnis der Vorlesung sind Kenntnisse in einer objektorientierten Sprache notwendig.

Generische Typen

Nur Konzepte aus der Vorlesung Programmierung.
Mit einer Ausnahme.

Java
```
public class DoublePair {
    Double a;
    Double b;
    public Pair(Double a, Double b) {this.a = a; this.b = b;}
    public void setfirst(Double a) {this.a = a;}
    public void setsecond(Double b) {this.b = b;}
    public Double first() {return a;}
    public Double second() {return b;}
    public String toString() {return"[" + a + "," + b + "]";}
}
```

Generische Typen

Nur Konzepte aus der Vorlesung Programmierung.
Ausnahme: Generische Typen!

Java
```
public class Pair⟨A, B⟩ {
    protected A a;
    protected B b;
    public Pair(A a, B b) {this.a = a; this.b = b;}
    public void setfirst(A a) {this.a = a;}
    public void setsecond(B b) {this.b = b;}
    public A first() {return a;}
    public B second() {return b;}
    public String toString() {return"[" + a + "," + b + "]";}
}
```


Java
```
public class Array⟨D⟩ {
    protected D[ ] a;
    protected int size;
```

Java
```
    public void set(int i, D d) {
        if(i ≥ size) {
            size = i + 1;
            realloc();
        }
        a[i] = d;
    }
```


Java
```
    private void realloc() {
      if(size ≥ a.length) {
        D[ ] b = (D[ ]) new Object[2 ∗ size];
        for(int i = 0; i < a.length; i++) if(a[i] ≠ null) b[i] = a[i];
        a = b;
      }
    }
```


Java
```
    public D get(int i) {
        if(i ≥ size) return null;
        return a[i];
    }
```

Warum eine eigene Array-Klasse?

1. Erweiterbarkeit
2. Typsicherheit
3. Anfertigen von Statistiken

Java

```
public interface Map⟨K, D⟩ {
    public void insert(K k, D d);
    public void delete(K k);
    public D find(K k);
    public boolean iselement(K k);
    public int size();
    public boolean isempty();
    public Iterator⟨K, D⟩ iterator();
    public SimpleIterator⟨K⟩ simpleiterator();
    public Array⟨K⟩ array();
    public void print();
}
```

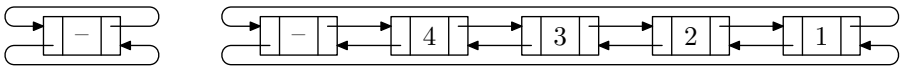

Assoziative Arrays

Java

```
public abstract class Dictionary⟨K, D⟩ implements Map⟨K, D⟩ {
  public abstract void insert(K k, D d);
  public abstract void delete(K k);
  public abstract Iterator⟨K, D⟩ iterator();
  public D find(K k) {
      Iterator⟨K, D⟩ it;
      for(it = iterator(); it.more(); it.step()) {
          if (it.key().equals(k)) {
              return it.data();
          }
      }
      return null;
  }
```


Doppelt verkettete Listen

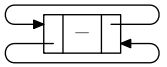
- Zusätzlicher Listenkopf
- Daten in Knoten gespeichert
- Zeiger zum Vorgänger und Nachfolger
- Zyklisch geschlossen



Geeignet für kleine assoziative Arrays.

Für Adjazenzlistendarstellung sehr geeignet.

Der Konstruktor muß den Listenkopf head erzeugen.
Der Vorgänger und Nachfolger von head ist head selbst.



Java

```
public class List⟨K, D⟩ extends Dictionary⟨K, D⟩ {
    Listnode⟨K, D⟩ head;
    public List() {
        head = new Listnode⟨K, D⟩(null, null);
        head.pred = head.succ = head;
    }
```

Java

```
class Listnode⟨K, D⟩ {
    K key;
    D data;
    Listnode⟨K, D⟩ pred, succ;
    Listnode(K k, D d) {
        key = k; data = d; pred = null; succ = null; }
    void delete() {
        pred.succ = succ; succ.pred = pred; }
    void copy(Listnode⟨K, D⟩ n) {
        key = n.key; data = n.data; }
    void append(Listnode⟨K, D⟩ newnode) {
        newnode.succ = succ; newnode.pred = this;
        succ.pred = newnode; succ = newnode; }
}
```


Java

```
public void append(K k, D d) {
    head.pred.append(new Listnode⟨K, D⟩(k, d));
}
```

Java

```
public void prepend(K k, D d) {
    head.append(new Listnode⟨K, D⟩(k, d));
}
```

Java

```
public void delete(K k) {
    Listnode⟨K, D⟩ n = findnode(k);
    if(n ≠ null) n.delete();
}
```


Java

```
public D find(K k) {
    Listnode⟨K, D⟩ n = findnode(k);
    if(n ≡ null) return null;
    return n.data;
}
```

Java

```
protected Listnode⟨K, D⟩ findnode(K k) {
    Listnode⟨K, D⟩ n;
    head.key = k;
    for(n = head.succ; !n.key.equals(k); n = n.succ);
    head.key = null;
    if(n ≡ head) return null;
    return n;
}
```


Einfach verkettete Listen

- Kein Zeiger auf Vorgänger
- Einfachere Datenstruktur
- Operationen können komplizierter sein

Frage: Wie kann ein Element *vor* einen gegebenen Knoten eingefügt werden?

Ersetzen und alten Knoten anfügen!

Vorsicht: Eine gefährliche Technik.

Listen – Laufzeit

Manche Operation sind schnell, manche langsam...

Operation	Laufzeit
append()	$\Theta(1)$
delete ()*	$\Theta(1)$
delete ()	$\Theta(n)$
find ()	$\Theta(n)$
insert ()	$\Theta(n)$

* direktes Löschen eines Knotens, nicht über Schlüssel

Iteratoren

Mittels eines Iterators können wir die Elemente einer Map aufzählen.

Java

```
public interface Iterator⟨K, D⟩ {
    public void step();
    public boolean more();
    public K key();
    public D data();
}
```


Iteratoren

Mittels eines Iterators können wir die Elemente einer Map aufzählen.

Java

```
Iterator⟨K, D⟩ it;
for(it = map.iterator(); it.more(); it.step()) {
  System.out.println(it.key());
}
```


Java

```
public class Stack⟨D⟩ {
  private List⟨Object, D⟩ stack;
  private int size;
  public Stack() {stack = new List⟨Object, D⟩(); size = 0; }
  public boolean isempty() {return size ≡ 0; }
  public D pop() {
    D x = stack.firstnode().data;
    stack.firstnode().delete();
    size−−;
    return x;
  }
  public void push(D x) {stack.prepend(null, x); size++; }
  public int size() {return size; }
}
```

Java

```
public class Queue⟨D⟩ {
  private List⟨Object, D⟩ queue;
  private int size;
  public Queue() {queue = new List⟨Object, D⟩(); size = 0;}
  public boolean isempty() {return size ≡ 0;}
  public D dequeue() {
    D x = queue.lastnode().data;
    queue.lastnode().delete();
    size−−;
    return x;
  }
  public void enqueue(D x) {queue.prepend(null, x); size++;}
  public int size() {return size;}
}
```


Lineare Suche

Wir suchen x im Array $a[0, \ldots, n-1]$

Algorithmus

function *find1*(**int** x) **boolean** :
for $i = 0$ **to** $n-1$ **do**
  if $x = a[i]$ **then return** *true* **fi**
od;
return *false*

Die innere Schleife ist langsam.

Lineare Suche – verbessert

Wir verwenden ein *Sentinel*-Element am Ende.

Das Element $a[n]$ muß existieren und unbenutzt sein.

Algorithmus

function *find2*(**int** x) **boolean** :
$i := 0$;
$a[n] := x$;
while $a[i] \neq x$ **do** $i := i + 1$ **od**;
return $i < n$

Die innere Schleife ist sehr schnell.

Lineare Suche – ohne zusätzlichen Platz

Algorithmus

function *find3*(**int** x) **boolean** :
if $x = a[n-1]$ **then return** *true* **fi**;
$temp := a[n-1]$;
$a[n-1] := x$;
$i := 0$;
while $a[i] \neq x$ **do** $i := i + 1$ **od**;
$a[n-1] := temp$;
return $i < n-1$

Erheblicher Zusatzaufwand.

Innere Schleife immer noch sehr schnell.

Lineare Suche – Analyse

Wieviele Vergleiche benötigt eine erfolgreiche Suche nach x im Mittel?

Wir nehmen an, daß jedes der n Elemente mit gleicher Wahrscheinlichkeit gesucht wird.

Es gilt $\Pr[x = a[i]] = 1/n$ für $i \in \{0, \ldots, n-1\}$.

Sei C die Anzahl der Vergleiche:
$C = i + 1$ gdw. $x = a[i]$

Lineare Suche – Analyse

Wir suchen den Erwartungswert von C.

$$\Pr[C = i] = 1/n \text{ für } i = 1, \ldots, n$$

$$E(C) = \sum_{k=1}^{n} k \Pr[C = k] = \sum_{k=1}^{n} \frac{k}{n} = \frac{n+1}{2}$$

Es sind im Mittel $(n+1)/2$ Vergleiche.

Lineare Suche – Analyse

Unser Ergebnis:

Es sind im Mittel $(n+1)/2$ Vergleiche.

Ist das Ergebnis richtig?

Wir überprüfen das Ergebnis für kleine n.

- $n = 1$?
- $n = 2$?

Binäre Suche

Wir suchen wieder x in $a[0, \ldots, n-1]$.

Algorithmus

```
function binsearch(int x) boolean :
l := 0; r := n − 1;
while l ≤ r do
  m := ⌊(l + r)/2⌋;
  if a[m] < x then l := m + 1 fi;
  if a[m] > x then r := m − 1 fi;
  if a[m] = x then return true fi
od;
return false
```

Wir halbieren den Suchraum in jedem Durchlauf.

Binäre Suche

Java

```
public boolean binsearch(D d) {
  int l = 0, r = size − 1, m, c;
  while(l ≤ r) {
    m = (l + r)/2;
    c = d.compareTo(get(m));
    if(c ≡ 0) return true;
    if(c < 0) r = m − 1;
    else l = m + 1;
  }
  return false;
}
```


Binäre Suche – Analyse

Java

```
function binsearch(int x) boolean :
l := 0; r := n − 1;
while l ≤ r do
  m := ⌊(l + r)/2⌋;
  if a[m] > x then l := m + 1 fi;
  if a[m] < x then r := m − 1 fi;
  if a[m] = x then return true fi
od;
return false
```

Es sei $n = r - l + 1$ die Größe des aktuellen Unterarrays.

Im nächsten Durchgang ist die Größe $m - l$ oder $r - m$.

Binäre Suche – Analyse

Lemma

Es sei $a \in \mathbf{R}$ *und* $n \in \mathbf{N}$. *Dann gilt*

1. $\lfloor a + n \rfloor = \lfloor a \rfloor + n$
2. $\lceil a + n \rceil = \lceil a \rceil + n$
3. $\lfloor -a \rfloor = -\lceil a \rceil$

Binäre Suche – Analyse

Im nächsten Durchlauf ist die Größe des Arrays $m - l$ oder $r - m$.

Hierbei ist $m = \lfloor (l + r)/2 \rfloor$.

Die neue Größe ist also

- $m - l = \lfloor (l + r)/2 \rfloor - l = \lfloor (r - l)/2 \rfloor = \lfloor (n - 1)/2 \rfloor$ oder
- $r - m = r - \lfloor (l + r)/2 \rfloor = \lceil (r - l)/2 \rceil = \lceil (n - 1)/2 \rceil$.

Im schlimmsten Fall ist die neue Größe des Arrays

$$\lceil (n - 1)/2 \rceil.$$

Rekursionsgleichung für binäre Suche

Sei S_n die Anzahl der Schleifendurchläufe im schlimmsten Fall bei einer erfolglosen Suche.

Wir erhalten die Rekursionsgleichung

$$S_n = \begin{cases} 0 & \text{falls } n < 1, \\ 1 + S_{\lceil (n-1)/2 \rceil} & \text{falls } n \geq 1. \end{cases}$$

Die ersten Werte sind:

n	0	1	2	3	4	5	6	7	8
S_n	0	1	2	2	3	3	3	3	4

Wir suchen eine geschlossene Formel für S_n.

Lösen der Rekursionsgleichung

Wir betrachten den Spezialfall $n = 2^k - 1$:

$$\left\lceil \frac{(2^k - 1) - 1}{2} \right\rceil = \left\lceil \frac{2^k - 2}{2} \right\rceil = \lceil 2^{k-1} - 1 \rceil = 2^{k-1} - 1.$$

Daher: $S_{2^k-1} = 1 + S_{2^{k-1}-1}$ für $k \geq 1$

$$\Rightarrow S_{2^k-1} = k + S_{2^0-1} = k$$

Binäre Suche – Analyse

n	0	1	2	3	4	5	6	7	8
S_n	0	1	2	2	3	3	3	3	4

Vermutung: $S_{2^k} = S_{2^k-1} + 1$

S_n steigt monoton $\Rightarrow S_n = k$, falls $2^{k-1} \leq n < 2^k$.

Oder falls $k - 1 \leq \log n < k$.

Dann wäre $S_n = \lfloor \log n \rfloor + 1$.

Binäre Suche – Analyse

Wir vermuten $S_n = \lfloor \log n \rfloor + 1$ für $n \geq 1$.

Induktion über n:

$S_1 = 1 = \lfloor \log 1 \rfloor + 1$

$n > 1$:

$$S_n = 1 + S_{\lceil (n-1)/2 \rceil} \overset{\text{I.V.}}{=} 1 + \lfloor \log \lceil (n-1)/2 \rceil \rfloor + 1.$$

Noch zu zeigen:

$$\lfloor \log n \rfloor = \lfloor \log \lceil (n-1)/2 \rceil \rfloor + 1$$

$\rightarrow$ Übungsaufgabe.

Binäre Suche – Analyse

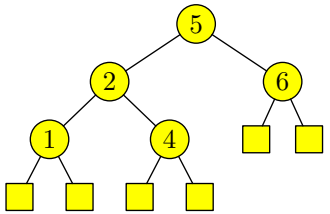
Theorem
Binäre Suche benötigt im schlimmsten Fall genau

$$\lfloor \log n \rfloor + 1 = \log(n) + O(1)$$

viele Vergleiche.

Die Laufzeit ist $O(\log n)$.

Binäre Suchbäume



- Als assoziatives Array geeignet
- Schlüssel aus geordneter Menge
- Linke Kinder kleiner, rechte Kinder größer als Elternknoten
- Externe und interne Knoten
- Externe Knoten zu einem Sentinel zusammenfassen?

Binäre Suchbäume

Java
```
public class Searchtree < K extends Comparable<K>, D >
                        extends Dictionary<K, D> {
  protected Searchtreenode<K, D> root;
```

Java
```
class Searchtreenode < K extends Comparable<K>, D > {
  K key;
  D data;
  Searchtreenode<K, D> left, right, parent;
```


Binäre Suchbäume – Suchen

Java
```
public D find(K k) {
  if(root ≡ null) return null;
  Searchtreenode<K, D> n = root.findsubtree(k);
  return n ≡ null ? null : n.data;
}
```

Im Gegensatz zu Liste:
Zusätzlicher Test auf **null**.

Binäre Suchbäume – Suchen

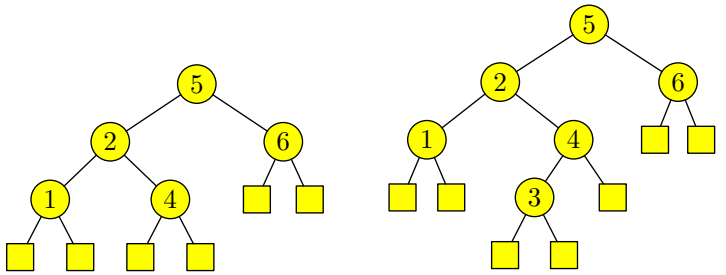
Java

```
Searchtreenode⟨K, D⟩ findsubtree(K k) {
  int c = k.compareTo(key);
  if(c > 0) return right ≡ null ? null : right.findsubtree(k);
  else if(c < 0) return left ≡ null ? null : left.findsubtree(k);
  else return this;
}
```

Wieder zusätzlicher Test auf **null**.
Durch Sentinel verhindern!

Besser: Iterativ statt rekursiv.

Binäre Suchbäume – Einfügen



Wo fügen wir 3 ein?

Wie fügen wir es ein?

In den richtigen externen Knoten!

Binäre Suchbäume – Einfügen

Java

```
public void insert(K k, D d) {
  if(root ≡ null) root = new Searchtreenode⟨K, D⟩(k, d);
  else root.insert(new Searchtreenode⟨K, D⟩(k, d));
}
```


Binäre Suchbäume – Einfügen

Java

```
public void insert(Searchtreenode⟨K, D⟩ n) {
  int c = n.key.compareTo(key);
  if(c < 0) {
      if(left ≠ null) left.insert(n);
      else {left = n; left.parent = this; }
  }
  else if(c > 0) {
      if(right ≠ null) right.insert(n);
      else {right = n; right.parent = this; }
  }
  else copy(n);
}
```

Binäre Suchbäume – Löschen

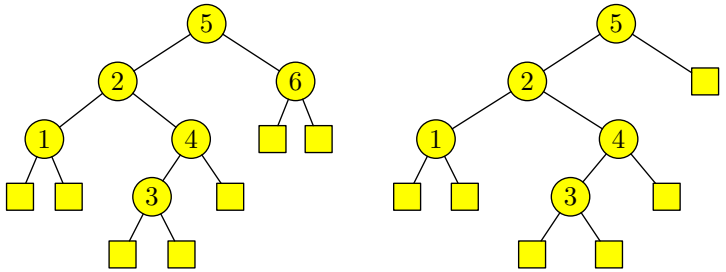
Beim Löschen unterscheiden wir drei Fälle:

- Blatt (einfach)
- Der Knoten hat kein linkes Kind (einfach)
- Der Knoten hat ein linkes Kind (schwierig)

Damit sind alle Fälle abgedeckt!
Warum kein Fall: Kein rechtes Kind?

Binäre Suchbäume – Löschen

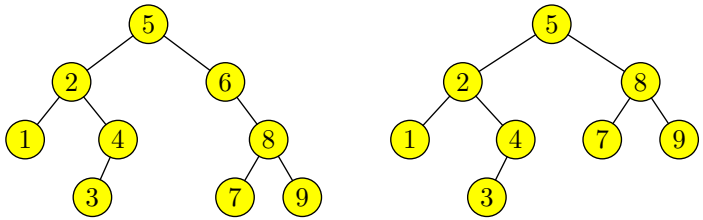
Löschen eines Blatts:



Ein Blatt kann durch Zeigerverbiegen gelöscht werden.

Binäre Suchbäume – Löschen

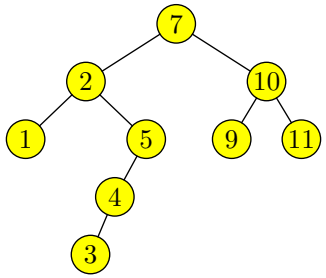
Löschen eines Knotens ohne linkes Kind:



Wir können kopieren oder Zeiger verbiegen.

Binäre Suchbäume – Löschen

Löschen eines Knotens mit linkem Kind:



1. Finde den größten Knoten im linken Unterbaum
2. Kopiere seinen Inhalt
3. Lösche ihn

Binäre Suchbäume – Löschen

In der Klasse Searchtree<K,D>:

Java

```
public void delete(K k) {
  if(root ≡ null) return;
  if(root.left ≡ null && root.right ≡ null && root.key ≡ k)
    root = null;
  else {
    Searchtreenode⟨K, D⟩ n = root.findsubtree(k);
    if(n ≠ null) n.delete();
  }
}
```


Binäre Suchbäume – Löschen

Java

```
void delete() {
  if(left ≡ null && right ≡ null) {
    if(parent.left ≡ this) parent.left = null;
    else parent.right = null; }
  else if(left ≡ null) {
    if(parent.left ≡ this) parent.left = right;
    else parent.right = right;
    right.parent = parent; }
  else {
    Searchtreenode⟨K, D⟩ max = left;
    while(max.right ≠ null) max = max.right;
    copy(max); max.delete();
  }
}
```


Binäre Suchbäume – Löschen

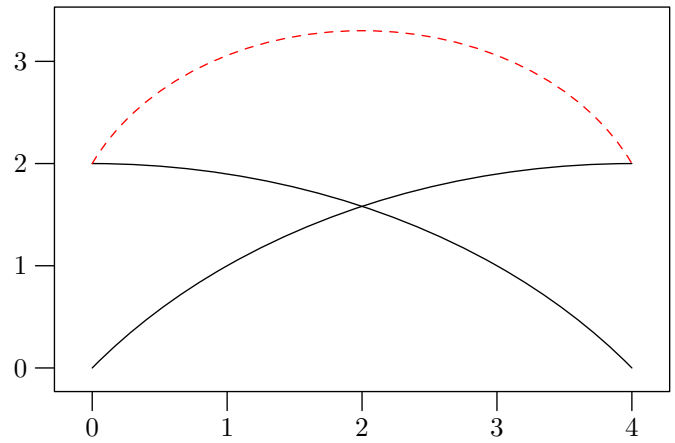
In Searchtree<K,D> jetzt korrekt:

Java

```
public void delete(K k) {
  if(root ≡ null) return;
  if(root.key.equals(k))
    if(root.left ≡ null && root.right ≡ null) {
      root = null; return;
    }
    else if(root.left ≡ null) {
      root = root.right; root.parent = null; return;
    }
  Searchtreenode⟨K, D⟩ n = root.findsubtree(k);
  if(n ≠ null) n.delete();
}
```

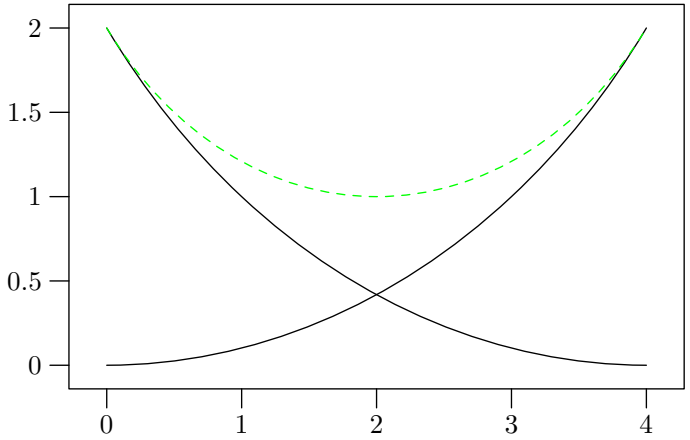

Binäre Suchbäume – Analyse

Eine kleine Vorbemerkung:



Summe schlechte Näherung des Maximums.

Binäre Suchbäume – Analyse



Summe gute Näherung des Maximums.

Die Kurven sind steiler.

Binäre Suchbäume – Analyse

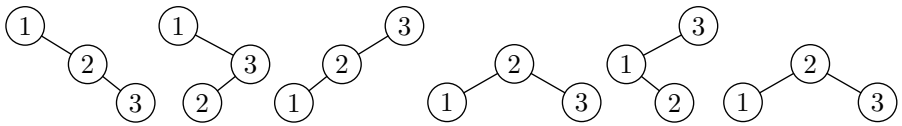
Wir fügen die Knoten $1, \ldots, n$ in zufälliger Reihenfolge in einen leeren Suchbaum ein.

Sei T_n die Höhe dieses Suchbaums.

Wir interessieren uns für $E(T_n)$.

Wir betrachten erst einmal T_0, T_1, T_2 und T_3:

- $E(T_0) = 0$
- $E(T_1) = 1$
- $E(T_2) = 2$
- $E(T_3) = 8/3$



Allgemeiner Fall:

Die Wurzel des Baums enthält $W \in \{1, \ldots, n\}$.

$$\Pr[W = k] = 1/n \text{ für } k \in \{1, \ldots, n\}$$

Wie sieht der Rest des Baums aus, falls $W = k$?

In den linken Teilbaum wurden $\{1, \ldots, k-1\}$ in zufälliger Reihenfolge eingefügt.

Seine Höhe ist T'_{k-1}.

Die Höhe des rechten Teilbaums ist T''_{n-k}.
Die Gesamthöhe ist $T_n = \max\{T'_{k-1}, T''_{n-k}\} + 1$.

Die Gesamthöhe ist $T_n = \max\{T'_{k-1}, T''_{n-k}\} + 1$.

$$E(T_n) = \frac{1}{n}\sum_{k=1}^{n} E(\max\{T'_{k-1}, T''_{n-k}\} + 1)$$

Wir können das Maximum durch die Summe abschätzen:

$$E(T_n) \leq \frac{1}{n}\sum_{k=1}^{n}\Big(E(T_{k-1}) + E(T_{n-k}) + 1\Big)$$

Leider zu grob! Führt zu einer schlechten Abschätzung.

Problem:

$E(\max\{X, Y\}) \leq E(X) + E(Y)$ korrekt, aber zu ungenau.

$E(\max\{X, Y\}) \leq \max\{E(X), E(Y)\}$ genau genug, aber zu unkorrekt.

Führe neue Zufallsvariablen ein:

$$\hat{T}_n = 2^{T_n},\ \ \hat{T}'_n = 2^{T'_n},\ \ \hat{T}''_n = 2^{T''_n}$$

$$ET_n = \frac{1}{n}\sum_{k=0}^{n-1} E\Big(\max\{T'_k, T''_{n-k-1}\} + 1\Big)$$

$$E\hat{T}_n = \frac{1}{n}\sum_{k=0}^{n-1} E\Big(2^{\max\{T'_k, T''_{n-k-1}\}+1}\Big)$$

Die Kurven sind jetzt steiler!

Vereinfachen wir diese Rekursionsgleichung zunächst:

$$E\hat{T}_n = \frac{1}{n}\sum_{k=0}^{n-1} E\Big(2^{\max\{T'_k, T''_{n-k-1}\}+1}\Big) =$$

$$= \frac{1}{n}\sum_{k=0}^{n-1} 2E\big(\max\{2^{T'_k}, 2^{T''_{n-k-1}}\}\big) = \frac{2}{n}\sum_{k=0}^{n-1} E\big(\max\{\hat{T}'_k, \hat{T}''_{n-k-1}\}\big)$$

$$\leq \frac{2}{n}\sum_{k=0}^{n-1} E\big(\hat{T}'_k + \hat{T}''_{n-k-1}\}\big) =$$

$$= \frac{2}{n}\sum_{k=0}^{n-1}\big(E\hat{T}_k + E\hat{T}_k\}\big) = \frac{4}{n}\sum_{k=0}^{n-1} E\hat{T}_k$$

Diese Rekursionsgleichung läßt sich mit Standardmethoden lösen.
→ Vorlesung Analyse von Algorithmen

$$E\hat{T}_n = \frac{4}{n}\sum_{k=0}^{n-1} E\hat{T}_k$$

Wir „lösen“ diese Gleichung nicht.

Wir zeigen nur, daß $E\hat{T}_n \leq (n+3)^{\underline{3}} = (n+3)(n+2)(n+1)$:

$n = 1$: $E\hat{T}_1 = 2 \leq (1+3)^{\underline{3}}$

$n > 1$:

$$E\hat{T}_n \leq \frac{4}{n}\sum_{k=0}^{n-1} E\hat{T}_k \overset{\text{I.V.}}{\leq} \frac{4}{n}\sum_{k=0}^{n-1}(k+3)^{\underline{3}} = (n+3)^{\underline{3}}.$$

Polynome verhalten sich gut beim Integrieren.

Lemma

$$\int_0^x t^k \, dt = \frac{x^{k+1}}{k+1}$$

$$\sum_{i=0}^{n-1} i^{\underline{k}} = \frac{n^{\underline{k+1}}}{k+1}$$

Fallende Potenzen verhalten sich gut beim Summieren.

Lemma

Es seien $p_1, \ldots, p_n \in [0,1]$ *mit* $p_1 + \ldots + p_n = 1$.

Des weiteren seien $w_1, \ldots, w_n \geq 0$.

Dann gilt

$$2^{\sum_{k=1}^n w_k p_k} \leq \sum_{k=1}^n 2^{w_k} p_k.$$

Beweis.

Wir betrachten $f(t) = 2^a t + 2^b(1-t)$ und $g(t) = 2^{at+b(1-t)}$ im Einheitsintervall.

Es sei $a \neq b$.

- f ist linear
- g ist überall links- oder rechtsgekrümmt: Denn $g''(t) = g(t)(\ln 2)^2(a-b)$.
- O.B.d.A. $a > b$, dann ist g rechtsgekrümmt.
- $f(0) = g(0) \leq f(1) = g(1) \Rightarrow f(t) \geq g(t)$

Daraus folgt

$$2^{\sum_{k=1}^n w_k p_k} \leq \sum_{k=1}^n 2^{w_k} p_k$$

für $n = 2$. □

Beweis.

Für $n > 2$ gilt:

$$2^{\sum_{k=1}^n k p_k} = 2^{np_n + \sum_{k=1}^{n-1} k p_k} = 2^{np_n + \left(\sum_{k=1}^{n-1} \frac{k p_k}{1-p_n}\right)(1-p_n)} \leq$$

$$\overset{\text{I.V.}}{\leq} 2^n p_n + 2^{\sum_{k=1}^{n-1} \frac{k p_k}{1-p_n}}(1-p_n) \leq$$

$$\overset{\text{I.V.}}{\leq} 2^n p_n + \left(\sum_{k=1}^{n-1} 2^k \frac{p_k}{1-p_n}\right)(1-p_n) =$$

$$= \sum_{k=1}^n 2^k p_k$$

□

Kommen wir zurück zu

$$E\hat{T}_n \leq (n+3)^{\underline{3}}.$$

Es gilt:

$$2^{ET_n} = 2^{\sum_{k=1}^{n} k \Pr[T_n=k]} \leq \sum_{k=1}^{n} 2^k \Pr[T_n = k] = E(2^{T_n})$$

Und damit:

$$ET_n \leq \log(E\hat{T}_n) \leq \log((n+3)^{\underline{3}}) = \\ = \log(n^3(1 + O(1/n)) = 3\log(n) + O(1/n)$$

Fertig!

Theorem (Mittlere Höhe eines Suchbaums)

Werden in einen leeren binären Suchbaum n verschiedene Schlüssel in zufälliger Reihenfolge eingefügt, dann ist die erwartete Höhe des entstehenden Suchbaums $O(\log n)$.

Wir verzichten auf eine Analyse für gemischtes Einfügen und Löschen.

Hier ist nicht so viel bekannt.

Experimente zeigen gutes Verhalten.

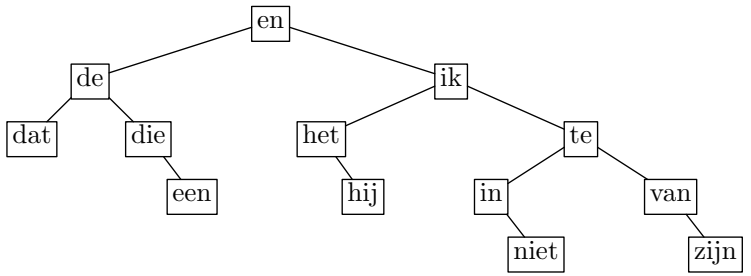
Die Höhe eines Suchbaums ist für seine Geschwindigkeit maßgebend.

Theorem

Die Operationen Einfügen, Löschen und Suchen benötigen $O(h)$ Zeit bei einem binären Suchbaum der Höhe h.

Optimale Suchbäume

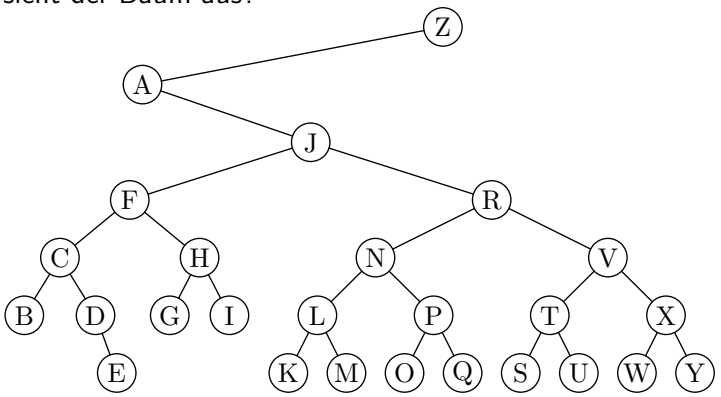
Ein optimaler Suchbaum für die 13 häufigsten Wörter des Buches Max Havelaar.



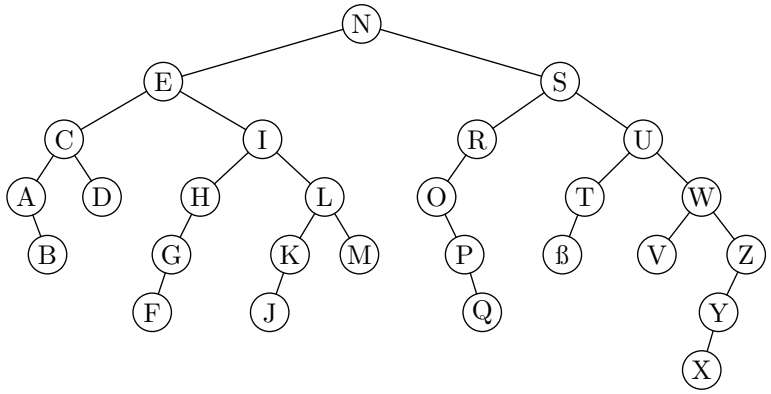
Wort	de	en	het	van	ik	te	dat	die
Anzahl	4770	2709	2469	2259	1999	1935	1875	1807

(Häufigkeitstabelle nach Wikipedia)

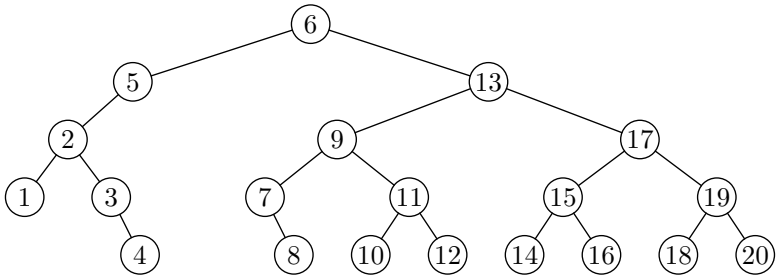
Ein optimaler Suchbaum enthält A bis Z. Die W'keit für A und Z sei 0.49, der Rest auf B–Y verteilt.
Wie sieht der Baum aus?



Optimale Suchbäume



Ein optimaler Suchbaum enthält die Zahlen 1–20. Auf jede wird mit W'keit 1/50 zugegriffen. Mit W'keit 3/5 wird 5.5 gesucht.
Wie sieht der Baum aus?



Optimale Suchbäume

- Es seien $k_1 < \cdots < k_n$ Schlüssel.
- Wir suchen k_i mit Wahrscheinlichkeit p_i.

Es gelte $\sum_{i=1}^{n} p_i = 1$.

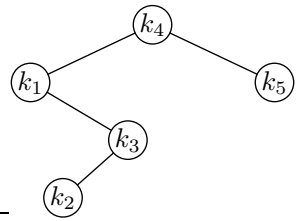
Definition

Ein optimaler Suchbaum für $k_1, \ldots, k_n$ gemäß den Zugriffswahrscheinlichkeiten $p_1, \ldots, p_n$ ist

- ein Suchbaum für $k_1, \ldots, k_n$,
- und hat eine minimale Anzahl von Vergleichen im Erwartungswert, falls k_i mit Wahrscheinlichkeit p_i gesucht wird.

Es sei $w_{i,j} = \sum_{k=i}^{j} p_k$.

Falls es in einem optimalen Suchbaum einen Unterbaum gibt, der die Schlüssel $k_i, \ldots, k_j$ enthält, dann sei $e_{i,j}$ der Erwartungswert der Anzahl der Vergleiche, die bei einer Suche in diesem Unterbaum durchgeführt werden.



i	1	2	3	4	5
p_i	1/4	1/8	1/6	1/4	5/24

$w_{1,5} = 1$, $w_{2,3} = 7/24$, $e_{2,2} = 1/8$, $e_{2,3} = 10/24$, $e_{1,3} = 23/24$

Wie können wir $w_{i,j} = \sum_{k=i}^{j} p_k$ berechnen?

Algorithmus

```
for i = 1,...,n do
  for j = 1,...,n do
    w[i,j] := 0;
    for k = i,...,j do
      w[i,j] := w[i,j] + p[k]
    od
  od
od
```

Wie schnell ist dieser Algorithmus?

Geht es schneller?

Wie können wir $w_{i,j} = \sum_{k=i}^{j} p_k$ schneller berechnen?

Durch dynamisches Programmieren:

Java

```
for i = 1,...,n do
  w[i,i] := p[i];
  for j = i+1,...,n do
    w[i,j] := w[i,j-1] + p[j]
  od
od
```

Wie schnell ist dieser Algorithmus?

Warum ist er korrekt?

Wie berechnen wir $e_{i,j}$?

Lemma

- $e_{i,j} = 0$ *für* $i > j$
- $e_{i,j} = \min_{i \le r \le j} (e_{i,r-1} + e_{r+1,j}) + w_{i,j}$ *für* $i \le j$

Beweis.
Ein Baum mit den Knoten $k_i, \ldots, k_j$ hat eine Wurzel k_r.
Mit Wahrscheinlichkeit $w_{i,j}$ wird der gesuchte Schlüssel mit k_r verglichen.
Im Mittel werden $e_{i,r-1}$ Vergleiche im linken und $e_{r+1,j}$ im rechten Unterbaum durchgeführt. □

i	1	2	3	4	5
p_i	1/4	1/8	1/6	1/4	5/24

w	1	2	3	4	5
1	0.25	0.375	0.542	0.792	1.0
2	0.0	0.125	0.292	0.542	0.75
3	0.0	0.0	0.167	0.417	0.625
4	0.0	0.0	0.0	0.25	0.458
5	0.0	0.0	0.0	0.0	0.208

e	1	2	3	4	5
1	0.25	0.5	0.958	1.542	2.167
2	0.0	0.125	0.417	0.917	1.375
3	0.0	0.0	0.167	0.583	1.0
4	0.0	0.0	0.0	0.25	0.667
5	0.0	0.0	0.0	0.0	0.208

$$e_{i,j} = \min_{i \le r \le j} (e_{i,r-1} + e_{r+1,j}) + w_{i,j}$$

RWTH AACHEN

Resultierender Algorithmus:

Algorithmus

for $i = 1, ..., n$ **do** $e[i, i-1] := 0$ **od**;
for $l = 0, ..., n$ **do**
  for $i = 1, ..., n - l$ **do**
    $j := i + l$;
    $e[i, j] := \infty$;
    for $r = i, ..., j$ **do**
      $e[i, j] := min\{e[i, j], e[i, r-1] + e[r+1, j] + w[i, j]\}$
    od
  od
od

Laufzeit: $O(n^3)$

RWTH AACHEN

Optimale Suchbäume – Allgemeiner Fall

- Es seien $k_1 < \cdots < k_n$ Schlüssel.
- Wir suchen k_i mit Wahrscheinlichkeit p_i.
- Wir suchen zwischen k_i und k_{i+1} mit Wahrscheinlichkeit q_i.
- (Mit W'keit q_0 wird links von k_1 gesucht.)
- (Mit W'keit q_n wird rechts von k_n gesucht.)

Natürlich gelte

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1.$$

RWTH AACHEN

Optimale Suchbäume – Allgemeiner Fall

Definition
Ein optimaler Suchbaum für $k_1, \ldots, k_n$ gemäß den Zugriffswahrscheinlichkeiten $p_1, \ldots, p_n$ und $q_0, \ldots, q_n$ ist

- ein Suchbaum für $k_1, \ldots, k_n$,
- und hat eine minimale Anzahl von Vergleichen im Erwartungswert, falls k_i mit Wahrscheinlichkeit p_i und zwischen k_i und k_{i+1} mit Wahrscheinlichkeit q_i gesucht wird.

RWTH AACHEN

Java

```
public void opt_searchtree(int n, Array⟨K⟩ keys,
                           Array⟨Double⟩ p, Array⟨Double⟩ q) {
  double[ ][ ] e = new double[n + 2][n + 1];
  double[ ][ ] w = new double[n + 2][n + 1];
  int[ ][ ] root = new int[n + 2][n + 1];
  for(int i = 1; i ≤ n+1; i++) w[i][i−1] = e[i][i−1] = q.get(i−1);
  for(int l = 0; l ≤ n; l++)
    for(int i = 1; i + l ≤ n; i++) {
      e[i][i + l] = Double.MAX_VALUE;
      w[i][i + l] = w[i][i + l − 1] + p.get(i + l) + q.get(i + l);
      for(int r = i; r ≤ i + l; r++) {
        Double t = e[i][r − 1] + e[r + 1][i + l] + w[i][i + l];
        if(t < e[i][i + l]) {e[i][i + l] = t; root[i][i + l] = r; }
      }
    }
  construct_opt_tree(1, n, keys, root);
}
```


Konstruktion optimaler Suchbäume

Java

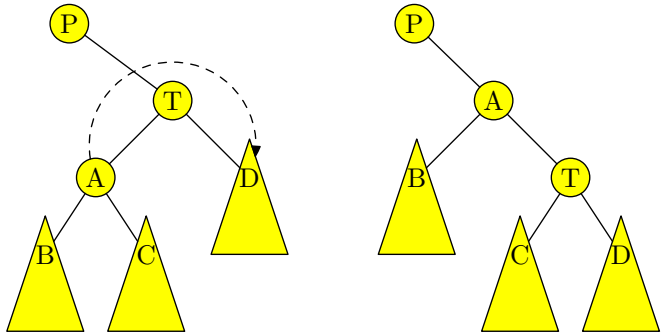
```
void construct_opt_tree(int i, int j,
       Array⟨K⟩ keys, int[ ][ ] root) {
  if(j < i) return;
  int r = root[i][j];
  insert(keys.get(r), null);
  construct_opt_tree(i, r − 1, keys, root);
  construct_opt_tree(r + 1, j, keys, root);
}
```


Konstruktion optimaler Suchbäume

Theorem

Wir können einen optimalen Suchbaum für n Schlüssel in $O(n^3)$ Schritten konstruieren.

Umstrukturierung durch Rotationen



Eine Rechtsrotation um T.

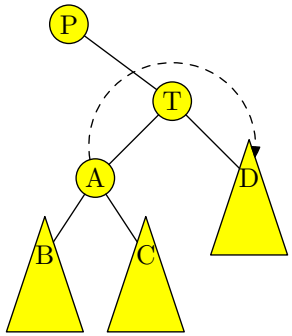
Die Suchbaumeigenschaft bleibt erhalten.

B, C, D können nur aus externen Knoten bestehen.

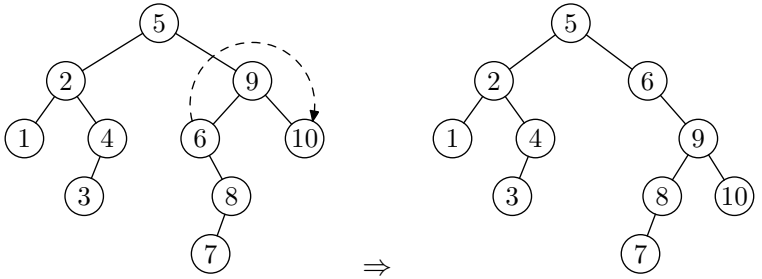
Laufzeit: Konstant!

Java

```
void rotateright() {
  Searchtreenode⟨K, D⟩ p, a, b, c, d;
  p = this.parent;
  a = this.left; d = this.right;
  b = a.left; c = a.right;
  if(p ≠ null) {
    if(p.left ≡ this) p.left = a;
    else p.right = a;
  }
  a.right = this; a.parent = p;
  this.left = c; this.parent = a;
  if(c ≠ null) c.parent = this;
}
```

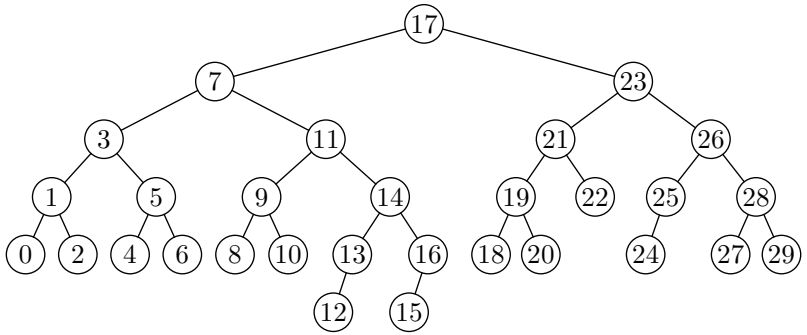


Beispiel



Frage: Kann man einen Knoten durch Rotationen zu einem Blatt machen?

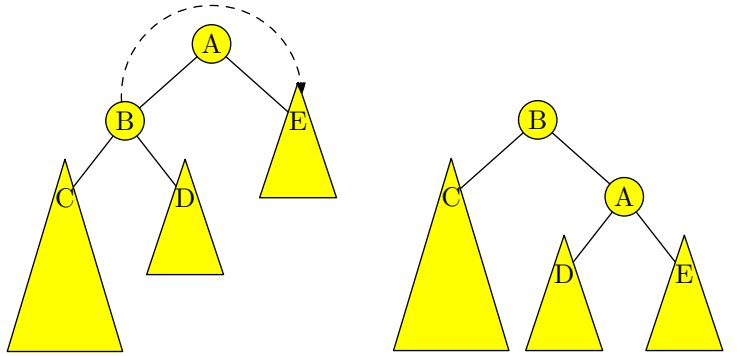
AVL-Bäume



AVL-Bäume werden annähernd balanziert gehalten.

AVL-Eigenschaft:
Die Höhen des rechten und linken Unterbaums jedes Knotens unterscheiden sich höchstens um 1.

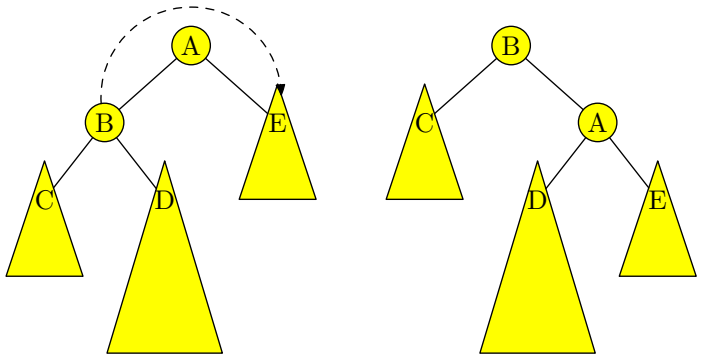
AVL-Bäume – Einfügen



Durch Einfügen in C ist die AVL-Bedingung verletzt.

Reparieren durch Rechtsrotation um A.

AVL-Bäume – Einfügen

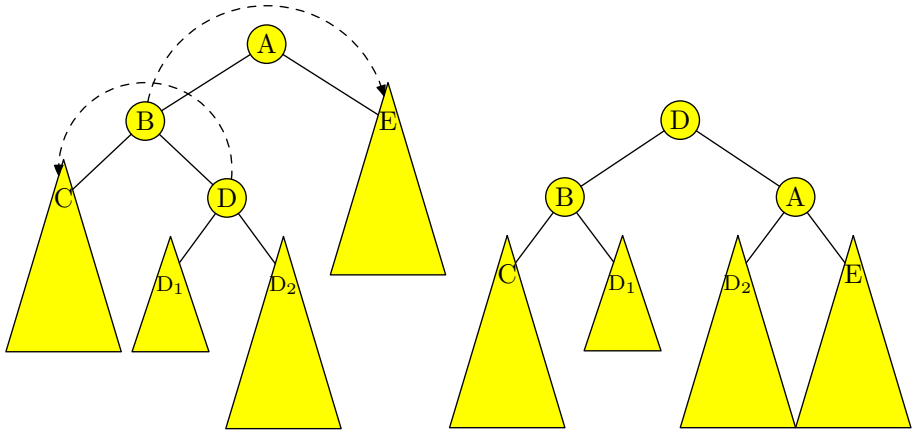


Durch Einfügen in D ist die AVL-Bedingung verletzt.

Reparieren durch Rechtsrotation um A?

Lösung: Erst um B links, dann um A rechts rotieren!

AVL-Bäume – Einfügen



Durch Einfügen in D ist die AVL-Bedingung verletzt.

Erst um B links, dann um A rechts rotieren!

Wir nennen dies auch eine Doppelrotation.

AVL-Bäume – Einfügen

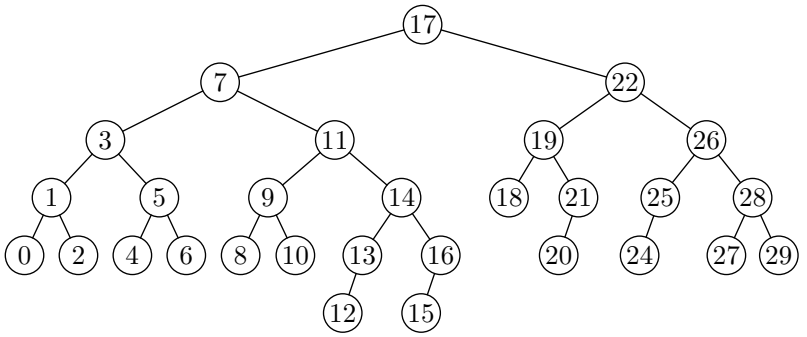
Durch Einfügen können nur Knoten auf dem Pfad von der Wurzel zum eingefügten Blatt unbalanziert werden.

Algorithmus

procedure *avl − insert*(*key k*) :
  insert(*k*);
  n := *findnode*(*k*);
  while *n* ≠ *root* **do**
    if *n unbalanced* **then** *rebalance n* **fi**;
    n := *parent*(*n*);
  od

Es werden höchstens zwei Rotationen durchgeführt.

Beispiel



Java

```
void rebalance() {
 computeheight();
 if(height(left) > height(right) + 1) {
  if(height(left.left) < height(left.right)) left.rotateleft();
  rotateright();
 }
 else if(height(right) > height(left) + 1) {
  if(height(right.right) < height(right.left))right.rotateright();
  rotateleft();
 }
 if(parent ≠ null) ((AVLtreenode⟨K, D⟩)parent).rebalance();
}
```


Java

```
public void insert(K k, D d) {
  if(root ≡ null) root = new AVLtreenode⟨K, D⟩(k, d);
  else root.insert(new AVLtreenode⟨K, D⟩(k, d));
  ((AVLtreenode⟨K, D⟩)root.findsubtree(k)).rebalance();
  repair_root();
}
```

Java

```
void repair_root() {
  if(root ≡ null) return;
  while(root.parent ≠ null) root = root.parent;
}
```


AVL-Bäume – Einfügen

- Nur Knoten auf Pfad zur Wurzel können unbalanziert werden.
- Durch Rotation oder Doppelrotation reparieren.
- Dadurch nimmt Höhe ab!
- $\Rightarrow$ Danach wieder balanziert.
- $\Rightarrow$ Es muß nur einmal repariert werden.
- Einfügen benötigt maximal zwei Rotationen.

AVL-Bäume – Löschen

Wiederholung

Drei Möglichkeiten beim Löschen:

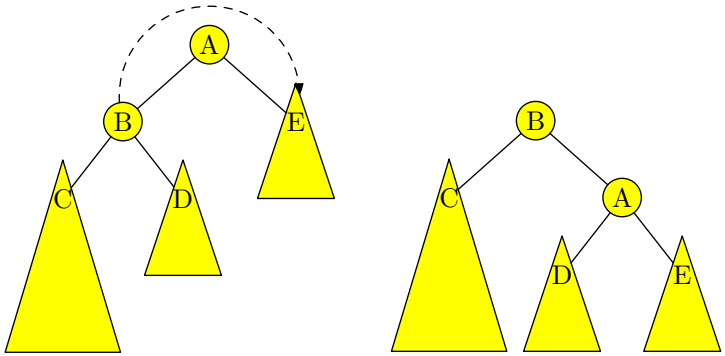
1. Ein Blatt
2. Kein linkes Kind
3. Es gibt linkes Kind

Nur bei den ersten beiden Fällen ändert sich die Höhe direkt!

Nur Knoten auf dem Pfad zur Wurzel können unbalanziert werden.

$\Rightarrow$ Wieder durch Rotationen reparieren.

AVL-Bäume – Löschen



Durch Löschen aus E ist AVL-Bedingung in A verletzt.

Reparieren durch Rechtsrotation um A.

Die Höhe ist dadurch gesunken!
Elternknoten von A kann wieder unbalanziert werden!

Java

```
void delete() {
  if(left ≡ null && right ≡ null) {
    if(parent.left ≡ this) parent.left = null;
    else parent.right = null;
    ((AVLtreenode⟨K, D⟩)parent).rebalance(); }
  else if(left ≡ null) {
    copy(right);
    if(right.left ≠ null) right.left.parent = this;
    if(right.right ≠ null) right.right.parent = this;
    left = right.left; right = right.right;
    rebalance(); }
  else {
    Searchtreenode⟨K, D⟩ max = left;
    while(max.right ≠ null) max = max.right;
    copy(max); max.delete(); }
}
```


AVL-Bäume – Löschen

Java

```
public void delete(K k) {
  if(root ≡ null) return;
  AVLtreenode⟨K, D⟩ n;
  n = (AVLtreenode⟨K, D⟩)root.findsubtree(k);
  if(n ≡ null) return;
  if(n ≡ root && n.left ≡ null && n.right ≡ null) root = null;
  else n.delete();
  repair_root();
}
```


AVL-Bäume – Analyse

Theorem
Ein AVL-Baum der Höhe h besitzt zwischen F_h und $2^h - 1$ viele Knoten.

Definition
Wir definieren die $n$te Fibonaccizahl:

$$F_n = \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ F_{n-1} + F_{n-2} & \text{falls } n > 2 \end{cases}$$

Beweis.
Trivial: Ein vollständiger Binärbaum der Höhe h hat genau $2^h - 1$ viele interne Knoten (und ist ein AVL-Baum).

Ein Baum der Höhe 0 hat keinen internen Knoten und $F_0 = 0$.

Ein Baum der Höhe 1 hat genau einen internen Knoten und $F_1 = 1$.

Falls $h > 1$, dann hat der linke oder rechte Unterbaum Höhe $h - 1$ und damit mindestens F_{h-1} viele interne Knoten.

Der andere Unterbaum hat mindestens Höhe $h - 2$ und damit mindestens F_{h-2} viele interne Knoten.

Insgesamt macht das mindestens $F_{h-1} + F_{h-2} = F_h$ viele Knoten. □

AVL-Bäume – Analyse

Theorem
Die Höhe eines AVL-Baums mit n Knoten ist $\Theta(\log n)$.

Beweis.
Es gilt $F_h = \Theta(\phi^h)$ mit $\phi = (1 + \sqrt{5})/2$.

$$F_h \leq n \Rightarrow h \log(\phi) + O(1) \leq \log n$$
$$n \leq 2^h - 1 \Rightarrow \log n \leq h + O(1)$$

□

AVL-Bäume – Analyse

Theorem
Einfügen, Suchen und Löschen in einen AVL-Baum mit n Elementen benötigt $O(\log n)$ Schritte.

Beweis.
Die Höhe des AVL-Baumes in $\Theta(\log n)$.
Einfügen, Suchen und Löschen benötigt $O(h)$ Schritte, wenn h die Höhe des Suchbaums ist.
Das Rebalanzieren benötigt ebenfalls $O(h)$ Schritte. □

Treaps

Suchbaumeigenschaft:

- Alle Knoten im linken Unterbaum kleiner als die Wurzel
- Alle Knoten im rechten Unterbaum größer als die Wurzel

Heapeigenschaft (Min-Heap):

- Alle Knoten im linken Unterbaum größer als die Wurzel
- Alle Knoten im rechten Unterbaum größer als die Wurzel

Ein Heap ist einfacher als ein Suchbaum.

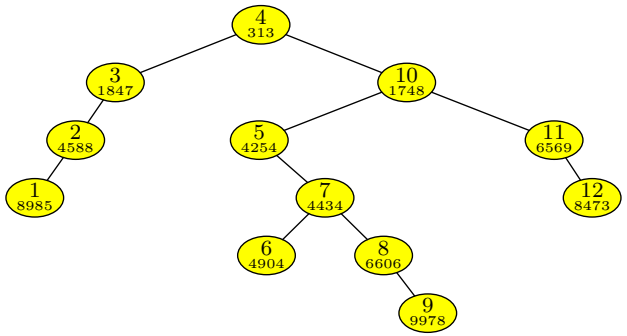
Treaps

Definition

Ein Treap ist ein binärer Baum mit:

- Jeder Knoten hat einen Schlüssel
- Jeder Knoten hat eine Priorität
- Der Baum ist ein Suchbaum bezüglich der Schlüssel
- Der Baum ist ein Heap bezüglich der Prioritäten

Treaps – Beispiel



Die Schlüssel sind groß und die Prioritäten klein geschrieben.

Treaps

Theorem

Gegeben seien n Schlüssel mit paarweise verschiedenen Prioritäten. Dann gibt es genau einen Treap, der diese Schlüssel und Prioritäten enthält.

Beweis.

Die Wurzel ist eindeutig:
Der Schlüssel mit minimaler Priorität.

- Der linke Unterbaum besteht aus allen kleineren Schlüsseln
- Der rechte Unterbaum besteht aus allen größeren Schlüsseln
- Induktion: Der linke und rechte Unterbaum ist eindeutig

□

Treaps

Lemma

Gegeben sei ein Treap mit paarweise verschiedenen Prioritäten. Dann ist die Form dieselbe wie bei einem binären Suchbaum, in den die Schlüssel in Reihenfolge der Prioritäten eingefügt wurden.

Beweis.

Die Wurzel im Treap hat die kleinste Priorität.
Wird sie in einen leeren Suchbaum als erste eingefügt, ist und bleibt sie die Wurzel des Suchbaums.

Durch vollständige Induktion haben auch die rechten und linken Teilbäume dieselbe Form wie im Treap. □

Theorem

Gegeben sei ein Treap der Größe n, dessen Prioritäten zufällig uniform aus der Menge $\{1, \ldots, m\}$ stammen, wobei $m \geq n^3$. Dann ist die Höhe des Treaps im Erwartungswert $O(\log n)$.

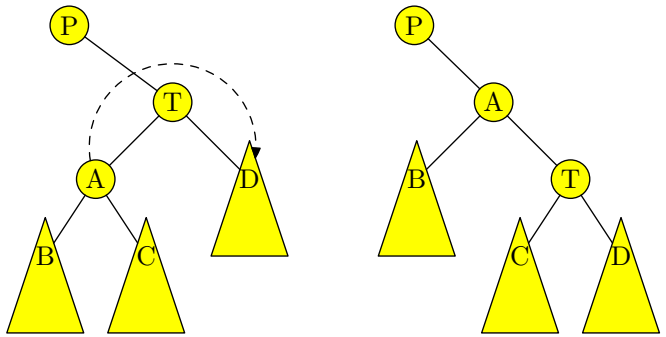
Beweis.

Seien $P_1, \ldots, P_n$ die Prioritäten.
Dann gilt $\Pr[P_i = P_j] \leq 1/n^3$ für $i \neq j$.
Es gibt weniger als n^2 Paare von Prioritäten. Die Wahrscheinlichkeit, daß irgendein Paar identisch ist, ist höchstens

$$\sum_{i \neq j} \Pr[P_i = P_j] \leq n^2 \cdot \frac{1}{n^3} = \frac{1}{n}.$$

Der Erwartungswert der Höhe ist höchstens
$O(\log n) + \frac{1}{n} \cdot n = O(\log n)$. □

Rotate-Down



Die Heapeigenschaft sei erfüllt, außer für T:
T habe eine zu große Priorität.

Nach der Rotation gilt wieder, daß die Heapeigenschaft erfüllt ist, außer für T.

Wir können T nach unten rotieren, bis es ein Blatt ist.

Rotate-Down

Java

```
void rotate_down(Treapnode⟨K, D⟩ n) {
  while(true) {
    if(n.left ≠ null) {
      if(n.right ≡ null ||
        ((Treapnode⟨K, D⟩)n.left).weight ≤
        ((Treapnode⟨K, D⟩)n.right).weight) n.rotateright();
      else n.rotateleft();
    }
    else if(n.right ≡ null) break;
    else n.rotateleft();
  }
  repair_root();
}
```

Ergebnis: Die Heapeigenschaft ist erfüllt, außer vielleicht für n.

n ist zu einem Blatt geworden.

Treaps – Löschen

Java

```
public void delete(K k) {
  if(root ≡ null) return;
  Treapnode⟨K, D⟩ n = (Treapnode⟨K, D⟩)root.findsubtree(k);
  if(n ≡ null) return;
  rotate_down(n);
  super.delete(k);
}
```

Wir rotieren den Knoten nach unten und entfernen ihn dann.

Treaps – Einfügen

Einfügen ist das Gegenteil von Löschen.

Löschen:

1. Knoten nach unten rotieren
2. Als Blatt entfernen

Einfügen:

1. Knoten als Blatt einfügen
2. Nach oben zur richtigen Position rotieren

Treaps – Einfügen

Java

```
void rotate_up(Treapnode⟨K, D⟩ n) {
  while(true) {
    if(n.parent ≡ null) break;
    if(((Treapnode⟨K, D⟩)n.parent).weight ≤ n.weight) break;
    if(n.parent.right ≡ n) n.parent.rotateleft();
    else n.parent.rotateright();
  }
  repair_root();
}
```

Wir rotieren nach oben bis wir die Wurzel erreichen oder der Elternknoten kleinere Priorität hat.

Treaps – Einfügen

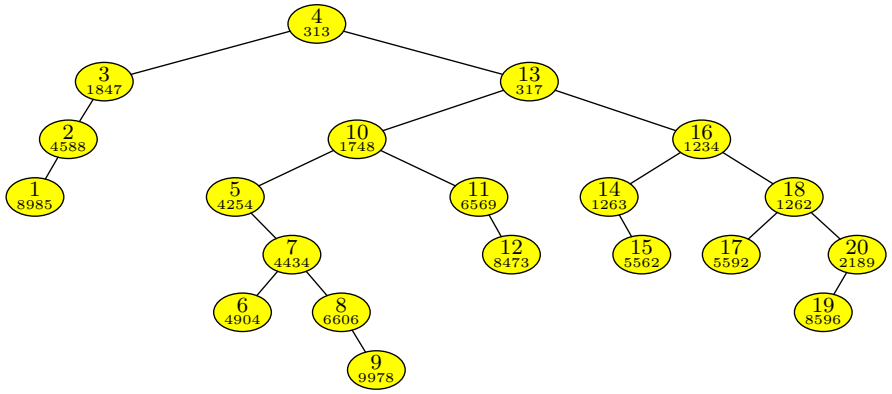
Java

```
public void insert(K k, D d) {
  if(root ≡ null) root = new Treapnode⟨K, D⟩(k, d, generator);
  else {
    root.insert(new Treapnode⟨K, D⟩(k, d, generator));
    rotate_up((Treapnode⟨K, D⟩)root.findsubtree(k));
  }
}
```

1. Normal als Blatt einfügen
2. Hochrotieren

Einfügen von 20 Elementen in einen Treap

Die Prioritäten sind zufällig gewählt.



Treaps

Theorem
Einfügen, Suchen und Löschen in einen Treap mit n Elementen benötigt $O(\log n)$ Schritte im Mittel, falls die Prioritäten aus einem ausreichend großen Bereich zufällig gewählt werden.

- Treaps sind in der Praxis sehr schnell.
- Standardimplementation für assoziative Arrays in LEDA.
- Einfach zu analysieren.
- Einfach zu implementieren.
- Sehr hübsch.

Splay-Bäume

- Splay-Bäume sind eine selbstorganisierende Datenstruktur.
- Basiert auf binären Suchbäumen.
- Restrukturiert durch Rotationen.
- Keine Zusatzinformation in Knoten.
- Nur amortisiert effizient.
- Einfach zu implementieren.
- Viele angenehme Eigenschaften (z.B. „selbstlernend“)
- Nur eine komplizierte Operation: splay

Die Splay-Operation

Gegeben ist ein binärer Suchbaum und ein Knoten x.

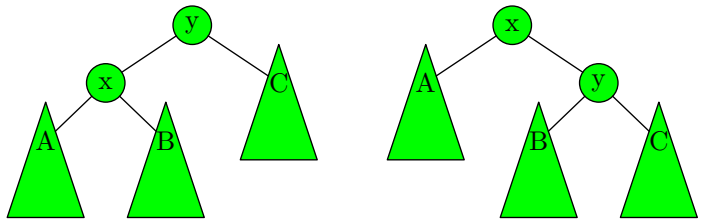
Algorithmus

procedure *splay*(*node* x) :
  while $x \neq$ *root* **do**
    splaystep(x)
  od

Wir führen Splay-Schritte auf x aus, bis es zur Wurzel wird.

Ein Splay-Schritt ist ein zig, zag, zig-zig, zig-zag, zag-zig oder zag-zag.

Zig und Zag

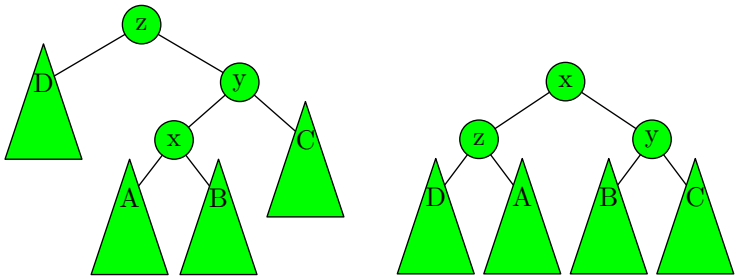


Ein zig auf x ist eine Rechtsrotation des Vaters von x.

Sie wird nur ausgeführt, wenn x das linke Kind der Wurzel ist.

Ein zag ist eine Linksrotation des Vaters, wenn x das rechte Kind der Wurzel ist.

Zig-zag und Zag-zig

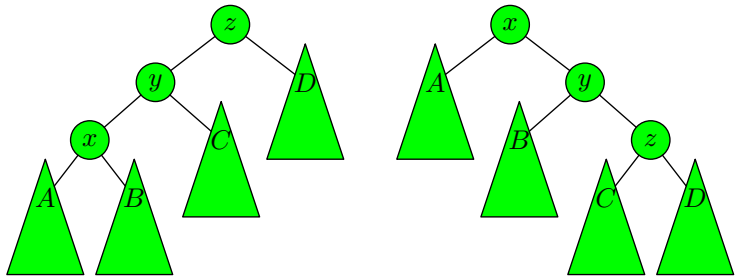


Ein Zig-zag auf x ist eine Rechtsrotation auf y gefolgt von einer Linksrotation auf z.

Dabei muß y das rechte Kind von z und x das linke Kind von y sein.

Zag-zig ist symmetrisch hierzu.

Zig-zig und Zag-zag



Ein Zig-zig auf x ist eine Rechtsrotation auf z gefolgt von einer Rechtsrotation auf y.

Dabei muß y das linke Kind von z und x das linke Kind von y sein.

Diese Operation sieht unerwartet aus!

Zag-zag ist wieder symmetrisch hierzu.

Amortisierte Analyse

Jeder Knoten x habe ein Gewicht $g(x) \in \mathbf{N}$.

Definition

- Der Kontostand eines Splay-Baums T ist $\sum_{v \in T} r(v)$.
- $r(v) = \left\lfloor \log(\bar{g}(v)) \right\rfloor$.
- $\bar{g}(v) = \sum_{u \in T(v)} g(u)$.
- $T(v)$ ist der Unterbaum mit Wurzel v

Amortisierte Analyse

Definition
Gegeben sei ein Splay-Schritt, der einen Splay-Baum T in einen Splay-Baum T' verwandelt.
Die amortisierten Kosten dieses Schrittes betragen

$$\sum_{v \in T'} r(v) - \sum_{v \in T} r(v) + 1.$$

Ein Schritt sind die tatsächlichen Kosten.

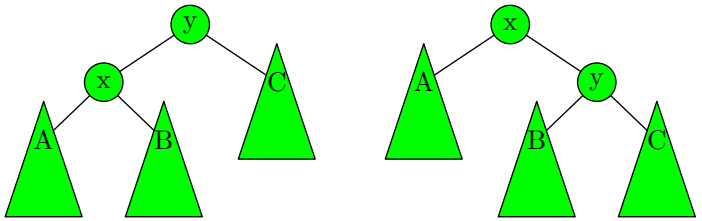
Der Rest ist eine Einzahlung oder Abhebung vom Konto.

Lemma

Die amortisierten Kosten eines

- *zig sind* $\leq 1 + 3(r(y) - r(x))$,
- *zig-zag sind* $\leq 3(r(z) - r(x))$,
- *zig-zig sind* $\leq 3(r(z) - r(x))$.

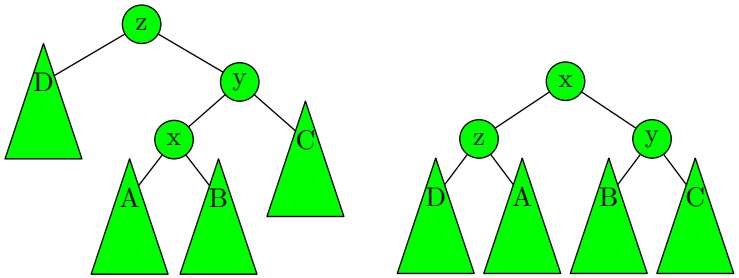
x, y, z sind die Knoten in den entsprechenden Zeichnungen.
Auf x wird die Operation ausgeführt.
z ist Großvater, y ist Vater von x.



- $r'(x) = r(y)$
- $r'(y) \leq r(y)$
- $r(y) \geq r(x)$

Die amortisierten Kosten eines zig sind

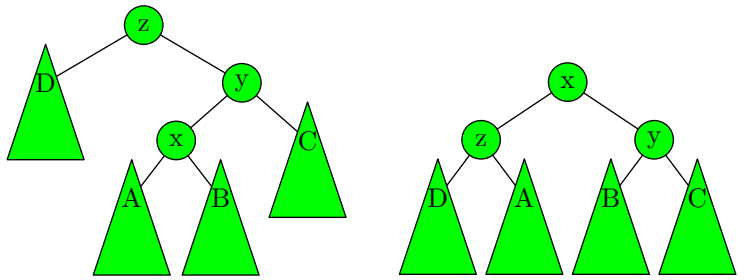
$$1 + r'(x) - r(x) + r'(y) - r(y) \leq 1 + r'(y) - r(x)$$
$$\leq 1 + r(y) - r(x) \leq 1 + 3(r(y) - r(x)).$$



Wir nehmen erst einmal $r(z) > r(x)$ an.

- $r'(x) = r(z)$
- $r'(y) \leq r'(x) = r(z)$
- $r'(z) \leq r'(x) = r(z)$
- $r(y) \geq r(x)$
- $1 \leq r(z) - r(x)$

$$1+r'(x)+r'(y)+r'(z)-r(x)-r(y)-r(z) \leq 1+r'(y)+r'(z)-r(x)-r(y)$$
$$\leq r(z) - r(x) + r(z) + r(z) - r(x) - r(x) = 3(r(z) - r(x)).$$



Jetzt nehmen wir $r(z) = r(x)$ an.

- $r'(x) = r(z)$
- $r'(y) < r(x)$ oder $r'(z) < r(x)$ (sonst $r'(x) > r(z)$)

$$1 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)$$
$$\leq 1 + r'(y) + r'(z) - r(x) - r(y) \leq 0 = 3(r(z) - r(x))$$

(Zig-zig ähnlich)

Splay-Bäume – Analyse

Theorem

Die amortisierten Kosten einer Splay-Operation auf einem Knoten x sind $O\big(\log(\bar{g}(w)/\bar{g}(x))\big)$, wenn w die Wurzel des Baums ist.

Beweis.

Die amortisierten Kosten sind höchstens

$$1 + 3(r(v_t) - r(v_{t-1})) + 3(r(v_{t-1}) - r(v_{t-2})) + \\ + \ldots + 3(r(v_2) - r(v_1)) = 1 + 3r(v_t) - 3r(v_1),$$

wobei $v_t = w$ und $v_1 = x$.

$1+3r(w)-3r(x) = O(\log(\bar{g}(w))-\log(\bar{g}(x))) = O(\log(\bar{g}(w)/\bar{g}(x)))$

□

RWTH AACHEN

Splay-Bäume – Suchen

Wir suchen folgendermaßen nach Schlüssel k:

1. Normale Suche im Suchbaum
2. Endet in Knoten x mit Schlüssel k oder in x^+ oder x^-
3. Wende Splay auf den gefundenen Knoten an
4. Siehe nach, ob k in Wurzel

Amortisierte Laufzeit:

$$O\left(\log\left(\frac{(\bar{g}(w))}{\bar{g}(x)}\right)\right) \text{ erfolgreiche Suche}$$

$$O\left(\log\left(\frac{\log(\bar{g}(w))}{\min\{\bar{g}(x^+), \bar{g}(x^-)\}}\right)\right) \text{ erfolglose Suche}$$

RWTH AACHEN

Splay-Bäume – Einfügen

Wir fügen einen Schlüssel k mit Gewicht a ein, der noch nicht vorhanden ist:

- Normale Suche im Suchbaum
- Einfügen eines neuen Knotens als Blatt
- Splay-Operation auf diesen Knoten

Amortisierte Laufzeit:

Das Konto wird zunächst erhöht.
x sei der neu eingefügte Knoten.
Die Splay-Operation benötigt $O(\log(\bar{g}(\bar{w}))/\bar{g}(x))$.

RWTH AACHEN

Splay-Bäume – Löschen

Wir löschen einen Schlüssel k:

1. Suche nach dem Schlüssel k
2. Siehe nach k in der Wurzel ist
3. Splay-Operation auf dem größten Knoten im linken Unterbaum
4. Klassisches Löschen von k

Amortisierte Laufzeit:
Zuerst wie Suche:

$$O\left(\log\left(\frac{(\bar{g}(w))}{\bar{g}(x)}\right)\right) \text{ erfolgreiche Suche}$$

$$O\left(\log\left(\frac{\log(\bar{g}(w))}{\min\{\bar{g}(x^+), \bar{g}(x^-)\}}\right)\right) \text{ erfolglose Suche}$$

Dann sinkt der Kontostand, was wir aber nicht ausnutzen.

RWTH AACHEN

Splay-Bäume als assoziatives Array

Theorem
In einem anfänglich leeren Splay-Baum können n Operationen (Suchen, Einfügen, Löschen) in $O(n \log n)$ Schritten ausgeführt werden.

Beweis.
Wir setzen $g(x) = 1$.

Dann ist $\bar{g}(T)$ die Anzahl der Knoten im Baum.
Also ist $\bar{g}(T) \leq n$.
Die amortisierten Kosten einer Operation sind
$O(\log(\bar{g}(T)) = O(\log n)$.
(Beim Einfügen kommt zur Splay-Operation noch die Erhöhung des Kontostands um $O(\log n)$ hinzu.) □

Amortisierte Analyse – Wiederholung

Eine Datenstruktur werde durch n Operationen verändert:

$$D_0 \xrightarrow{t_1} D_1 \xrightarrow{t_2} D_2 \xrightarrow{t_3} \cdots \xrightarrow{t_n} D_n$$

Die Zeit dafür ist $\sum_{k=1}^{n} t_i$.

Klassische Analyse:

- Jede Operation benötigt höchstens $f(n)$ Schritte
- Die Gesamtzeit ist $O(f(n)n)$.

Problematisch, wenn t_i sehr schwankend.

Amortisierte Analyse – Wiederholung

$$D_0 \xrightarrow{t_1} D_1 \xrightarrow{t_2} D_2 \xrightarrow{t_3} \cdots \xrightarrow{t_n} D_n$$

Wir definieren uns eine Potentialfunktion Φ mit:

1. $\Phi(D_0) = 0$
2. $\Phi(D_i) \geq 0$
3. $a_i := t_i + \Phi(D_i) - \Phi(D_{i-1})$ für $i > 0$
4. a_i nicht sehr schwankend

Amortisierte Analyse:

- Zeige, daß $a_i \leq g(n)$
- Gesamtzeit höchstens $O(ng(n))$

$$ng(n) \geq \sum_{k=1}^{n} a_i = \sum_{k=1}^{n} t_i + \Phi(D_n) + \Phi(D_0) \geq \sum_{k=1}^{n} t_i$$

Splaytrees und Optimale Suchbäume

Theorem
Gegeben sei ein Suchbaum T für n Schlüssel. Eine bestimmte Folge von m Suchoperationen in T benötige Zeit t.
Wenn wir die n Schlüssel in einen Splay-Baum einfügen und dann dieselbe Folge von Suchoperatioen ausführen, dauert dies $O(n^2 + t)$.

Bedeutung:

Asymptotisch verhalten sich Splay-Bäume ebensogut wie optimale Suchbäume.

Der Splay-Baum benötigt aber Zeit, um die Zugriffshäufigkeiten zu „lernen“.

Beweis.
Sei $d(k)$ die Tiefe des Knotens mit Schlüssel k in T und d die Gesamttiefe von T.
Wir definieren $g(k) = 3^{d-d(k)}$ als Gewichtsfunktion.

Die amortisierten Kosten einer Suche nach k sind:

$$O\left(\log\left(\frac{(\bar{g}(w))}{\bar{g}(k)}\right)\right)$$

Es gilt $\bar{g}(w) \leq \sum_{i=1}^{n} 3^{d-d(k_i)} \leq \sum_{j=0}^{d} 2^j 3^{d-j} \leq 3^{d+1}$ und
$\bar{g}(k) \geq g(k) = 3^{d-d(k)}$.
Die Kosten sind daher höchstens $O(\log(3^{d+1}/3^{d-d(k)}) = O(d(k))$.

Die Suche in T benötigt aber $\Omega(d(k))$ Zeit.

Das Aufbauen des Splaytrees geht in $O(n^2)$. □


```
private void splay(Searchtreenode⟨K, D⟩ t) {
 while(t.parent ≠ null) {
   if(t.parent.parent ≡ null) {
     if(t ≡ t.parent.left) t.parent.rotateright(); // Zig
     else t.parent.rotateleft(); } // Zag
   else if(t ≡ t.parent.left && t.parent ≡ t.parent.parent.left) {
     t.parent.parent.rotateright(); // Zig-zig
     t.parent.rotateright(); }
   else if(t ≡ t.parent.left && t.parent ≡ t.parent.parent.right) {
       t.parent.rotateright(); // Zig-zag
       t.parent.rotateleft(); }
   else if(t ≡ t.parent.right && t.parent ≡ t.parent.parent.right) {
     t.parent.parent.rotateleft(); // Zag-zag
     t.parent.rotateleft(); }
   else if(t ≡ t.parent.right && t.parent ≡ t.parent.parent.left) {
       t.parent.rotateleft(); // Zag-zig
       t.parent.rotateright(); }
 }
 root = t;
}
```


Java

```
public boolean iselement(K k) {
  if(root ≡ null) return false;
  Searchtreenode⟨K, D⟩ n = root, last = root;
  int c;
  while(n ≠ null) {
    last = n;
    c = k.compareTo(n.key);
    if(c < 0) n = n.left;
    else if(c > 0) n = n.right;
    else {splay(n); return true; }
  }
  splay(last); return false;
}
```


Java

```
public void insert(K k, D d) {
  super.insert(k, d);
  iselement(k);
}
```

Java

```
public D find(K k) {
  iselement(k);
  if(root ≠ null && root.key.equals(k)) return root.data;
  return null;
}
```

Java

```
public void delete(K k) {
  if(!iselement(k)) return;
  if(root.left ≠ null) {
    Searchtreenode⟨K, D⟩ max = root.left;
    while(max.right ≠ null) max = max.right;
    splay(max);
  }
  super.delete(k);
}
```

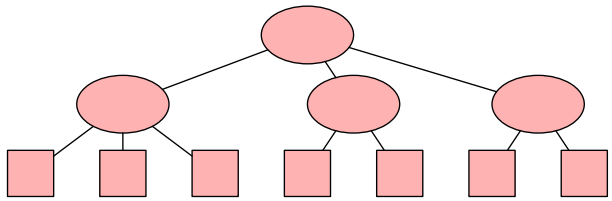

(a, b)-Bäume

Es sei $a \geq 2$ und $b \geq 2a - 1$.

Definition

Ein (a, b)-Baum ist ein Baum mit folgenden Eigenschaften:

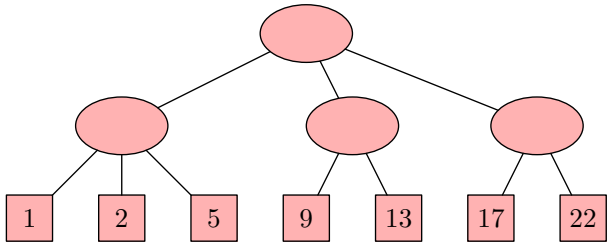
1. Jeder Knoten hat höchstens b Kinder.
2. Jeder innere Knoten außer der Wurzel hat mindestens a Kinder.
3. Alle Blätter haben die gleiche Tiefe.



Ein $(2, 3)$-Baum.

(a, b)-Bäume als assoziatives Array

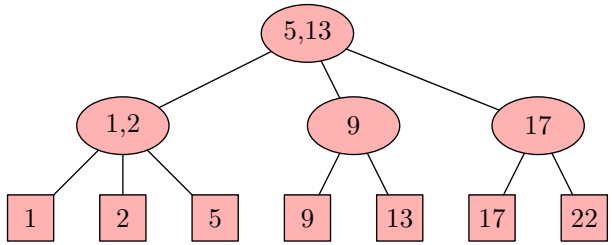
Wir speichern Schlüssel und Datenelemente nur in den Blättern:



Die Schlüssel sind von links nach rechts geordnet.

(a, b)-Bäume als assoziatives Array

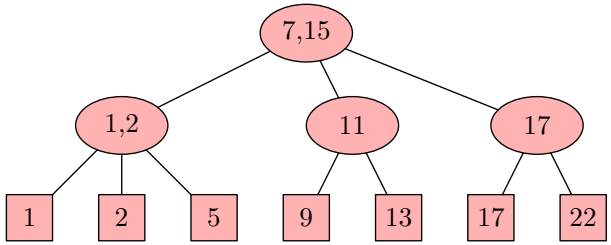
Als Suchhilfe enthält ein innerer Knoten mit m Kindern genau $m - 1$ Schlüssel.



Jetzt kann effizient nach einem Element gesucht werden.

(a, b)-Bäume als assoziatives Array

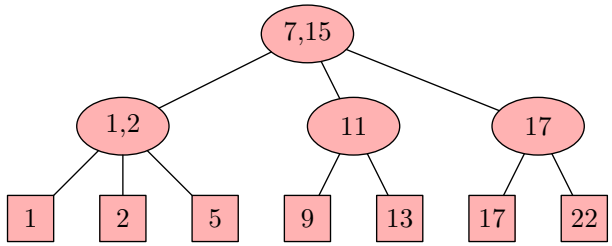
Wir bestehen nicht darauf, daß die Hilfsschlüssel in inneren Knoten in den Blättern vorkommen:



$\rightarrow$ etwas flexibler

(a, b)-Bäume – Einfügen

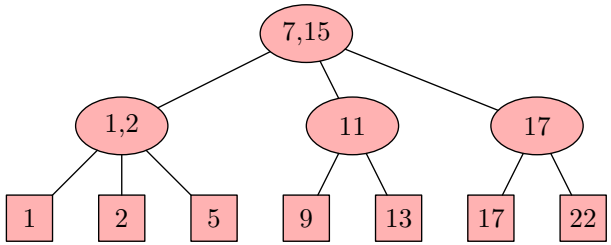
Die Blätter enthalten die Schlüssel in aufsteigender Reihenfolge.



- Neues Blatt an richtiger Stelle einfügen
- Problem, falls Elternknoten mehr als b Kinder hat
- $\rightarrow$ in zwei Knoten mit $\lfloor (b+1)/2 \rfloor$ und $\lceil (b+1)/2 \rceil$ Kindern teilen
- Jetzt kann Vater überfüllt sein etc.

(a, b)-Bäume – Löschen

Die Blätter enthalten die Schlüssel in aufsteigender Reihenfolge.



- Blatt mit Schlüssel entfernen
- Problem, falls Elternknoten weniger als a Kinder hat
- $\rightarrow$ mit einem Geschwisterknoten vereinigen
- Dieser muß vielleicht wieder geteilt werden
- Der nächste Elternknoten kann nun wieder unterbelegt sein

(a, b)-Bäume als assoziatives Array

- $(2, 3)$-Bäume sind als assoziatives Array geeignet
- Löschen, Suchen, Einfügen in $O(\log n)$
- Welche anderen (a, b) sind interessant?

Bei großen Datenmengen, die nicht mehr im Hauptspeicher Platz haben, sind die bisher betrachteten Datenstrukturen nicht gut geeignet.

Verwende $(m, 2m)$-Bäume mit großem m

B-Bäume und Datenbanken

Ein B-Baum ist ein $(m, 2m)$-Baum.

Wir wählen m so groß, daß ein Knoten soviel Platz wie eine Seite im Hintergrundspeicher benötigt (z.B. 4096 Byte).

Blätter eines Elternknotens gemeinsam speichern.

Zugriffszeit:

Nur $O(\log_m(n))$ Zugriffe auf den Hintergrundspeicher.

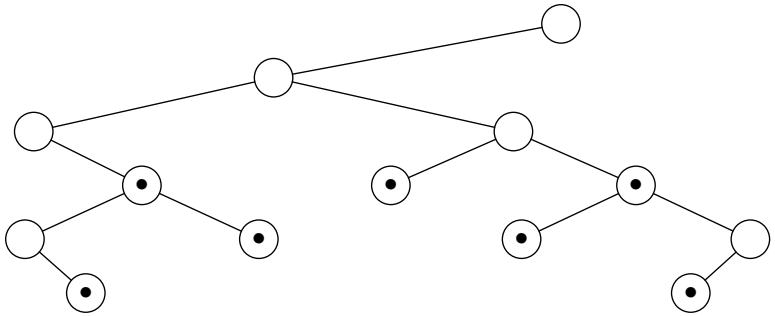
Wurzel kann immer im RAM gehalten werden.

Falls $m \approx 500$, dann enthält ein B-Baum der Höhe 3 bereits mindestens $500 \cdot 500 \cdot 500 = 125\,000\,000$ Schlüssel und Datenelemente.

Jede Suche greift auf nur zwei Seiten zu!

Tries

In vergleichsbasierten Suchbäumen wird nicht in Schlüssel hineingeschaut.



In Tries entspricht die $i$te Verzweigung dem $i$ten Zeichen des Schlüssels.

Obiger Trie enthält die Schlüssel AAB, AABAB, AABB, ABA, ABB, ABBA und ABBBA.

Hashing

Wie können wir eine partielle Funktion $\{1, \ldots, n\} \to \mathbf{N}$ effizient speichern?
Wie können wir ein assoziatives Array für die Schlüssel $\{1, \ldots, n\}$ effizient implementieren?

Durch ein Array mit n Elementen.

Aber: Gewöhnlich speichern wir n Schlüssel aus einem riesigen Universum U.

$S \subseteq U$, $|S|$ klein, $|U|$ sehr groß.

Ein Array der Größe $|U|$ ist zu groß.

Ein Array der Größe $O(|S|)$ wäre optimal.

Finde eine Funktion $h\colon U \to \{1, \ldots, |S|\}$, die auf S injektiv ist.

Hashing

Es sei $S \subseteq U$.
Wir haben eine Funktion $h\colon U \to \{1, \ldots, n\}$, so daß

- $n \geq |S|$,
- $h(x) \neq h(y)$ für alle $x \neq y$, $x, y \in S$.

Wir können $x \in S$ in der $n$ten Position eines Arrays speichern.

Probleme:

- Wie finden wir h?
- Können wir $h(x)$ effizient berechnen?
- Ist S überhaupt bekannt? Ändert es sich?

Universelles Hashing

Wir nehmen an, S ist bekannt und ändert sich nicht.

Beispiel: Tabelle der Schlüsselwörter in einem Compiler.

Es sei $n \geq |S|$ und $\mathcal{H}$ die Menge aller Funktion $U \to \{1, \ldots, n\}$.
Offensichtlich enthält $\mathcal{H}$ Funktionen, die injektiv auf S sind.

Idee:
Ersetze $\mathcal{H}$ durch eine kleinere Klasse von Funktionen, die für jedes $S' \subseteq U$, $|S| = |S'|$ eine auf S' injektive Funktion h enthält.

Universelles Hashing

Definition

Es sei $\mathcal{H}$ eine nicht-leere Menge von Funktionen $U \to \{1, \ldots, m\}$.
Wir sagen, daß $\mathcal{H}$ eine universelle Familie von Hashfunktionen ist, wenn für jedes $x, y \in U$, $x \neq y$ folgendes gilt:

$$\frac{|\{\, h \in \mathcal{H} \mid h(x) = h(y) \,\}|}{|\mathcal{H}|} \leq \frac{1}{m}$$

Theorem

Es sei $\mathcal{H}$ eine universelle Familie von Hashfunktionen $U \to \{1, \ldots, m\}$ für das Universum U und $S \subseteq U$ eine beliebige Untermenge.
Wenn $x \in U$, $x \notin S$ und $h \in \mathcal{H}$ eine zufällig gewählte Hashfunktion ist, dann gilt

$$E\Big(|\{\, y \in S \mid h(x) = h(y) \,\}|\Big) \leq \frac{|S|}{m}.$$

Beweis.

$$E\Big(|\{\, y \in S \mid h(x) = h(y) \,\}|\Big) =$$

$$\sum_{y \in S} \Pr[h(x) = h(y)] = \sum_{y \in S} \frac{|\{\, h \in \mathcal{H} \mid h(x) = h(y) \,\}|}{|\mathcal{H}|} \leq \frac{|S|}{m}$$

Weitere Konsequenz:

Lemma

Sei $x \in U$ beliebig, $\mathcal{H}$ eine universelle Familie von Hashfunktionen $U \to \{1, \ldots, m\}$ und k eine beliebige Zahl aus $\{1, \ldots, m\}$.
Dann gilt $\Pr[h(x) = k] = 1/m$, falls h zufällig aus $\mathcal{H}$.

Beweis.

Nehmen wir an, es gibt ein y mit $h(y) = k$. Dann setze $S = \{y\}$ und wende das letzte Theorem an:

$$E\Big(|\{\, y \in S \mid h(x) = h(y) \,\}|\Big) \leq \frac{|S|}{m}.$$

Hier folgt daraus $\Pr[h(x) = k] \leq 1/m$.
Wenn es kein y mit $h(y) = k$ gäbe, dann wäre $\Pr[h(x) = k] = 0$.
Das geht aber nicht, denn $\sum_{k=1}^{m} \Pr[h(x) = k] = 1$. □

Universelles Hashing

Im folgenden nehmen wir an, daß wir eine zufällige Hashfunktion aus einer geeigneten universellen Familie verwenden.

In der Praxis verwendet man oft einfachere Hashfunktionen, aber Experimente deuten an, daß sie sich praktisch so gut wie universelle Hashfunktionen verhalten.

Sie funktionieren im Worst-Case nicht, aber haben sich doch praktisch bewährt. Wir müssen darauf achten, daß die Schlüssel möglichst gleichmäßig und unabhängig voneinander verteilt werden.

Hashing mit Verkettung

Auf Hashing basiertes assoziatives Array:

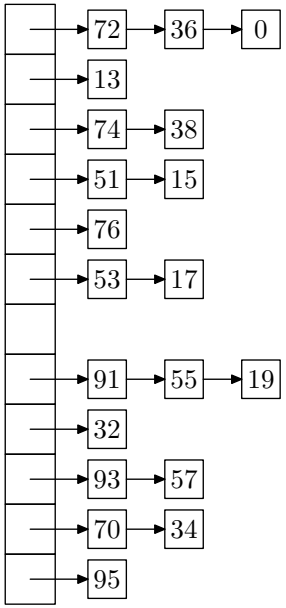
Wie behandeln wir Kollisionen?

Eine einfache Möglichkeit:

- Die Hashtabelle hat m Positionen
- Jede Position besteht aus einer verketteten Liste
- Die Liste auf Position k enthält alle x mit $h(x) = k$

Vorteile: Einfügen, Suchen und Löschen sehr einfach umzusetzen.

Nachteile: Speicherverschwendung?



Beispiel einer Hashtabelle mit Verkettung

- 20 Elemente
- Tabellengröße 12
- Lastfaktor $\alpha = 5/3$

Hashing mit Verkettung – Analyse

Hashtabelle der Größe m mit n Elementen gefüllt.

Erfolglose Suche:

Sei x ein Element, das nicht in der Tabelle ist.

$$\Pr[h(x) = k] = 1/m$$

Sei L_i die Größe der Liste auf Position i.
Erwartete Anzahl besuchter Listenelemente:

$$\frac{1}{m}\sum_{k=1}^{m} L_i = \frac{n}{m} = \alpha$$

Laufzeit $O(\alpha)$.

Hashing mit Verkettung – Analyse

Erfolgreiche Suche:
Wir wählen ein Element x aus der Tabelle zufällig aus.
Es bestehe S aus den anderen Elementen in der Tabelle.

Erwartete Anzahl von anderen Elementen in der Liste, die x enthält:

$$E(\{\, y \in S \mid h(x) = h(y) \,\}) \leq \frac{n-1}{m}$$

Da x zufällig gewählt wurde, ist x ein zufälliges Element in dieser Liste und daher an zufälliger Position.

$\rightarrow$ Im Durchschnitt sind die Hälfte der anderen Elemente vor x.

Anzahl besuchter Listenelemente im Durchschnitt:
$\leq 1 + \alpha/2$ (x und Hälfte der Fremden)

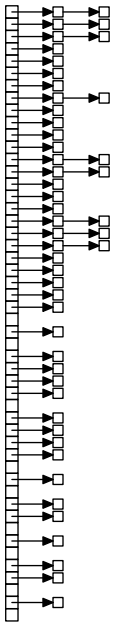
Hashing mit Verkettung – Analyse

Insgesamt erhalten wir folgende Abschätzung:

Theorem
Sei eine Hashtabelle mit Verkettung der Größe m gegeben, die mit n Schlüsseln gefüllt ist.
Bei universellem Hashing benötigt

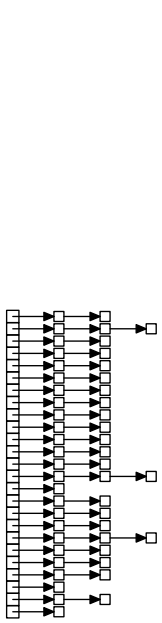
1. *eine erfolglose Suche nach einem beliebigen Element $x \in U$ im Durchschnitt höchstens $\alpha = n/m$ viele Vergleiche,*
2. *eine erfolgreiche Suche nach einem zufällig gewählten Element x aus der Tabelle im Durchschnitt höchstens $1 + \alpha/2$ viele Vergleiche.*

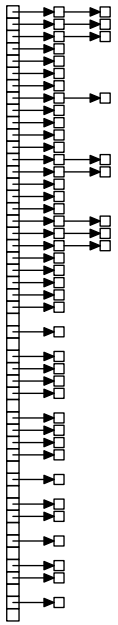
Hashing ist sehr effizient, wenn wir die Hashfunktion schnell auswerten können und α nicht zu groß ist.



Ein größeres Beispiel einer Hashtabelle mit Verkettung

- 50 Elemente
- Tabellengröße 50
- Lastfaktor $\alpha = 1$





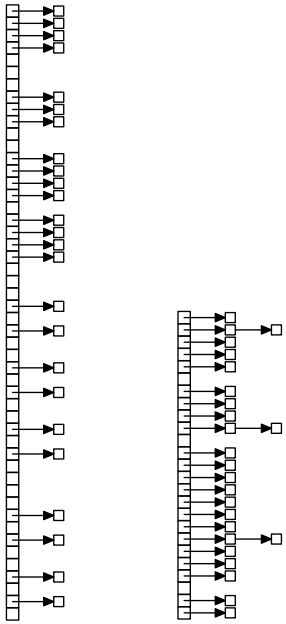
Rehashing

Nach Einfügen ist Tabelle zu voll

- Lastfaktor $\alpha = 2$
- Rehashing auf doppelte Größe
- $\rightarrow$ Lastfaktor 1

Wie teuer?

Amortisierte Kosten: $O(1)$



Rehashing

Nach Löschen ist Tabelle zu leer (Speicherplatz!)

- Lastfaktor $\alpha = 1/2$
- Rehashing auf halbe Größe
- $\rightarrow$ Lastfaktor 1

Wie teuer?

Amortisierte Kosten: $O(1)$

Java

```
int slot(K k) {
  int n = k.hashCode() % tablesize;
  return n > 0 ? n : −n;
}
```

Java

```
public D find(K k) {
  int l = slot(k);
  if(table.get(l) ≡ null) return null;
  return table.get(l).find(k);
}
```


Java

```
public void insert(K k, D d) {
  int l = slot(k);
  if(table.get(l) ≡ null) table.set(l, new List⟨K, D⟩());
  if(!table.get(l).iselement(k)) size++;
  table.get(l).insert(k, d);
  rehash();
}
```


Java

```
public void delete(K k) {
  int l = slot(k);
  if(table.get(l) ≡ null ||!table.get(l).iselement(k)) return;
  table.get(l).delete(k);
  if(table.get(l).isempty()) table.set(l, null);
  size−−;
  rehash();
}
```

Java

```
void rehash() {
  if(size ≤ tablesize && 4 ∗ size + 10 ≥ tablesize) return;
  int newtablesize = 2 ∗ size + 10;
  Array⟨List⟨K, D⟩⟩ newtable;
  newtable = new Array⟨List⟨K, D⟩⟩(newtablesize);
  Iterator⟨K, D⟩ it;
  for(it = iterator(); it.more(); it.step()) {
    int l = it.key().hashCode() % newtablesize;
    if(l < 0) l = −l;
    if(newtable.get(l) ≡ null) newtable.set(l, new List⟨K, D⟩());
    newtable.get(l).insert(it.key(), it.data());
  }
  table = newtable; tablesize = newtablesize;
}
```


Andere Methoden zur Kollisionsauflösung

Neben Verkettung gibt es viele andere Methoden, Kollision zu behandeln:

- Linear Probing
- Quadratic Probing
- Double Hashing
- Sekundäre Hashtabelle

und viele andere...

Universelle Familien von Hashfunktionen

Sei $U = \{0, \ldots, p-1\}$, wobei p eine Primzahl ist.
Es sei $h_{a,b}(x) = ((ax + b) \bmod p) \bmod m$.
Wir definieren

$$\mathcal{H} = \{\, h_{a,b} \mid 1 \leq a < p,\ 0 \leq b < p \}$$

Theorem
$\mathcal{H}$ ist eine universelle Familie von Hashfunktionen.

Es seien $x, y \in \{0, \ldots, p-1\}$, $x \neq y$.
Wir wollen zunächst zeigen, daß die Funktion

$$f : (a, b) \mapsto (ax + b \bmod p, ay + b \bmod p)$$

für $a, b \in \{0, \ldots, p-1\}$ injektiv und somit auch bijektiv ist.

$$\begin{aligned}
&(ax + b \bmod p, ay + b \bmod p) = (a'x + b' \bmod p, a'y + b' \bmod p) \\
\Leftrightarrow &(ax + b - b' \bmod p, ay + b - b' \bmod p) = (a'x \bmod p, a'y \bmod p) \\
\Leftrightarrow &(b - b' \bmod p, b - b' \bmod p) = ((a' - a)x \bmod p, (a' - a)y \bmod p) \\
\Leftrightarrow &a' = a \wedge b' = b
\end{aligned}$$

Nach wie vor gelte $x, y \in \{0, \ldots, p-1\}$, $x \neq y$.
Für wieviele Paare (a, b) haben $c_x := ax + b \bmod p$ und $c_y := ay + b \bmod p$ den gleichen Rest modulo m?

Wir haben auf der letzten Folie bewiesen, daß sich für jedes Paar (a, b) ein eindeutiges Paar (c_x, c_y) ergibt. Für ein festes c_x gibt es nur

$$\lceil p/m \rceil - 1 = \left\lfloor \frac{p+m-1}{m} \right\rfloor - 1 \leq \frac{p-1}{m}$$

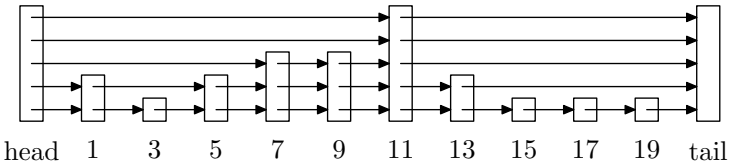
viele mögliche Werte von c_y mit $c_x \equiv c_y \mod m$ und $c_x \neq c_y$.

Weil p verschiedene Werte für c_x existieren, gibt es insgesamt höchstens $p(p-1)/m$ Paare der gesuchten Art.

$$\frac{|\{\, h \in \mathcal{H} \mid h(x) = h(y) \,\}|}{|\mathcal{H}|} \leq \frac{p(p-1)/m}{p(p-1)} \leq \frac{1}{m}$$

RWTH AACHEN

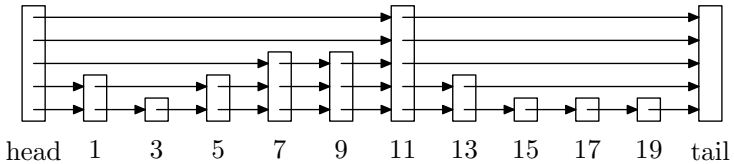
Skip-Lists

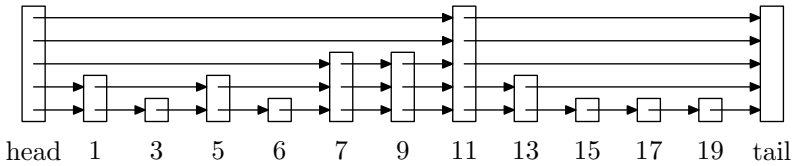


1. Schlüssel nach Größe einsortiert.
2. Es gibt beliebig viele Listen.
3. Anzahl ist geometrisch verteilt.
4. Anzahl ändert sich nicht.
5. Suchen: Von oben nach unten.

RWTH AACHEN

Skip-Lists – Einfügen

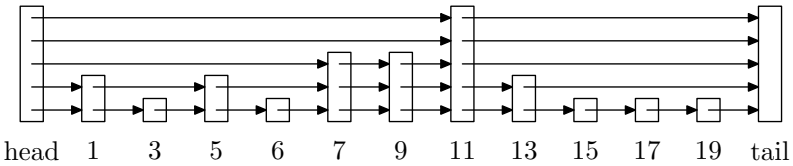


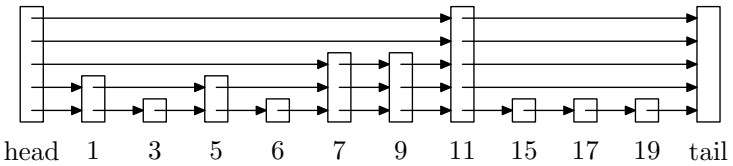


Einfügen des Elements 6.

RWTH AACHEN

Skip-Lists – Löschen





Löschen des Elements 13.

RWTH AACHEN

Java

```
public class Skiplist < K extends Comparable⟨K⟩, D >
                extends Dictionary⟨K, D⟩ {
  int size;
  double prob = 0.5;
  Random rand;
  class Node {
    Array⟨Node⟩ succ;
    K key;
    D data;
  }
  Node head, tail;
```


Java

```
  public Skiplist() {
    head = new Node();
    tail = new Node();
    head.succ = new Array⟨Node⟩();
    tail.succ = new Array⟨Node⟩();
    head.succ.set(0, tail);
    size = 0;
    rand = new Random();
  }
```

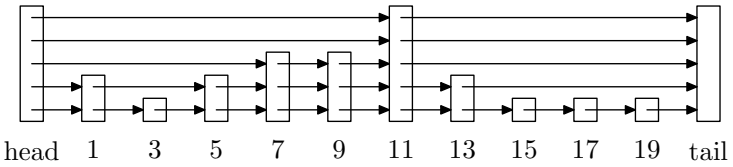
Es gibt noch weitere Konstruktoren, die es erlauben, den Zufallsgenerator und p vorzugeben.

Java

```
  public D find(K k) {
    Node n = findnode(k);
    if(n ≡ null) return null;
    return n.data;
  }
```

Java

```
  public boolean iselement(K k) {
    return findnode(k) ≠ null;
  }
```

Java

```
  Node findnode(K k) {
    Node n = head;
    for(int i = head.succ.size() − 1; i ≥ 0; i−−)
      while(n.succ.get(i) ≠ tail &&
            n.succ.get(i).key.compareTo(k) ≤ 0)
              n = n.succ.get(i);
    if(n ≡ head ||!n.key.equals(k)) return null;
    return n;
  }
```

Java

```
public void insert(K k, D d) {
  delete(k);
  int s = 1;
  while(rand.nextDouble() ≥ prob) s++;
  Node n = new Node();
  n.key = k; n.data = d;
  n.succ = new Array⟨Node⟩(s);
  Node m = head;
  for(int i = 0; i < s; i++)
    if(i ≥ head.succ.size()) head.succ.set(i, tail);
  for(int i = s − 1; i ≥ 0; i−−) m = insert_on_level(i, m, n);
  size++;
 }
```


Java

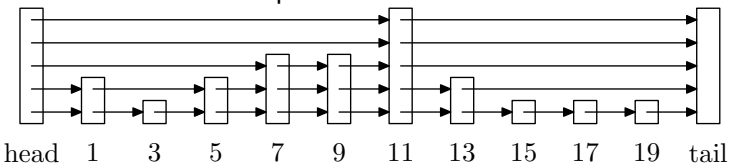
```
Node insert_on_level(int i, Node m, Node n) {
  while(m.succ.get(i) ≠ tail
        && m.succ.get(i).key.compareTo(n.key) < 0) {
      m = m.succ.get(i);
  }
  n.succ.set(i, m.succ.get(i));
  m.succ.set(i, n);
  return m;
}
```


Java

```
public void delete(K k) {
 Node n = head;
 if(iselement(k)) size−−; else return;
 for(int i = head.succ.size() − 1; i ≥ 0; i−−) {
   while(n.succ.get(i) ≠ tail &&
         n.succ.get(i).key.compareTo(k) < 0) n = n.succ.get(i);
   if(n.succ.get(i) ≠ tail && n.succ.get(i).key.equals(k))
     n.succ.set(i, n.succ.get(i).succ.get(i));
 }
}
```


Skip-Lists – Analyse

Es sei h die Höhe, n die Anzahl der Schlüssel und p die Wahrscheinlichkeit der Skip-List.



Die erfolgreiche Suche führt entlang eines Wegs, der oben in head beginnt und schlimmstens unten im gesuchten Knoten endet.

Sehen wir uns den Weg rückwärts an!
Wieweit gehen wir im Durchschnitt, bis der Weg nach oben führt?
$\rightarrow 1/(1-p)$ viele Schritte!
$\rightarrow$ Insgesamt $h/(1-p) = O(h)$ Schritte im Erwartungswert.

Skip-Lists – Analyse

Erfolgreiche Suche: $O(h)$

Wie hoch ist eine Skip-Liste mit n Elementen?
Es sei h_i die Höhe des $i$ten Knotens.

$$\Pr[h_i \geq t] = (1-p)^{t-1}$$

$$\Pr[h_i \geq t \text{ für ein } i] \leq n(1-p)^{t-1}$$

Setze $t = -2\log_{1-p}(n) + 1 = 2\log_{1/(1-p)}(n) + 1 = O(\log n)$.

$$\rightarrow \Pr[h = O(\log n)] \geq 1 - n(1-p)^{\log_{1-p}(1/n^2)} = 1 - \frac{1}{n}$$

Also gilt $E(h) = O(\log n)(1-1/n) + O(n) \cdot 1/n = O(\log n)$.

Skip-Lists – Analyse

Theorem
Skip-Listen unterstützen die Operationen Einfügen, Löschen und Suchen in erwarteter Zeit $O(\log n)$.
Der Speicherverbrauch ist im Erwartungswert $O(n)$.

Beweis.
Löschen benötigt asymptotisch so viel Zeit wie Suchen, also $O(\log n)$.
Einfügen ebenfalls, außer die Höhe nimmt zu. Die Zeit dafür ist aber im Mittel nur $O(1)$, obwohl sie (mit kleiner Wahrscheinlichkeit) unbeschränkt groß werden kann.
Jeder Knoten benötigt im Durchschnitt $O(1)$ Platz, insgesamt ergibt das $O(n)$. □

Skip-Lists – Fragen

Wie lange benötigt eine erfolglose Suche nach einem Element, das größer ist als alle Schlüssel in der Skip-List?

Welchen Fehler darf man bei der Implementierung nicht machen, um dies zu vermeiden:

Die Liste ist fast leer, doch das Einfügen geht sehr, sehr langsam.

Mengen

Der abstrakte Datentyp Menge sollte folgende Operationen unterstützen:

- $x \in M$?
- $M \rightarrow M \cup \{x\}$
- $M \rightarrow M \setminus \{x\}$
- $M = \emptyset$?
- wähle irgendein $x \in M$

Möglicherweise auch

- $M_1 \rightarrow M_2 \cup M_3$
- $M_1 \rightarrow M_2 \cap M_3$
- $M_1 \rightarrow M_2 \setminus M_3$
- ...

Mengen können durch assoziative Arrays implementiert werden:

Java

```
public class Set<K> {
  private final Map<K,?> h;
  public Set() {h = new Hashtable<K,Integer>();}
  public Set(Map<K,?> m) {h = m;}
  public void insert(K k) {h.insert(k,null);}
  public void delete(K k) {h.delete(k);}
  public void union(Set<K> U) {
    SimpleIterator<K> it;
    for(it = U.iterator(); it.more(); it.step())
      insert(it.key());}
  public boolean iselement(K k) {return h.iselement(k);}
  public SimpleIterator<K> iterator() {
   return h.simpleiterator();}
  public Array<K> array() {return h.array();}
}
```


Bitarrays

Wenn das Universum U klein ist, können wir Mengen durch Bitarrays implementieren.

Laufzeiten:

- Suchen, Einfügen, Löschen: $O(1)$
- Vereinigung, Schnitt: $O(|U|)$
- Auswahl: $O(|U|)$ (oder $O(1)$ mit Zusatzzeigern)

Insertion Sort

Wir sortieren ein unsortiertes Array, indem wir wiederholt Elemente in ein bereits sortiertes Teilarray einfügen.

Eingabe:

12	73	36	71	79	67	3	17	32	31	14	14	33	4	74	23

Ausgabe:

3	4	12	14	14	17	23	31	32	33	36	67	71	73	74	79

Insertion Sort

12	73	36	71	79	67	3	17	32	31	14	14	33	4	74	23

Algorithmus

```
procedure insertionsort(n) :
  for i = 2,...,n do
    j := i;
    while j ≥ 2 and a[j − 1] > a[j] do
      vertausche a[j − 1] und a[j];
      j := j − 1
    od
  od
```

Inversionen

Die Laufzeit von Insertion Sort ist $O(n^2)$.

Definition
Sei $\pi \in S_n$ eine Permutation. Die Menge der Inversionen von π ist

$$I(\pi) = \{ (i,j) \in \{1,\ldots,n\}^2 \mid i < j \text{ und } \pi(i) > \pi(j) \}.$$

Beim Sortieren durch Einfügen werden Schlüssel nur durch Vertauschungen benachbarter Elemente bewegt.
So eine Vertauschung verringert die Anzahl der Inversionen höchstens um eins.
Wenn die Eingabe genau falschherum sortiert ist, hat Insertion Sort die Laufzeit $\Theta(n^2)$.
Was ist die durchschnittliche Laufzeit?

Inversionen

Theorem
Eine zufällig gewählte Permutation $\pi \in S_n$ hat im Erwartungswert $n(n-1)/4$ Inversionen.

Beweis.
Es gibt $n(n-1)/2$ viele Paare (i,j) mit $1 \leq i < j \leq n$.
Wegen

$$\Pr[\pi(i) > \pi(j)] = \frac{1}{2} \text{ falls } i \neq j$$

gilt

$$E(|I(\pi)|) = \sum_{i<j} \Pr[\pi(i) > \pi(j)] = \frac{n(n-1)}{4}.$$

□

Inversionen

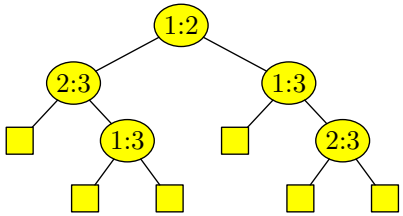
Theorem
Jedes Sortierverfahren, das Schlüssel nur durch Vertauschungen benachbarter Elemente bewegt, benötigt im Durchschnitt $\Omega(n^2)$ Zeit.

Folgerung:

Wenn wir schneller sein wollen, müssen Schlüssel über weite Strecken bewegt werden.

Vergleichsbäume

Insertion Sort mit $n = 3$:



Jeder vergleichsbasierte Sortieralgorithmus hat einen Vergleichsbaum.

Die Wurzel ist der erste Vergleich.

Links folgen die Vergleiche beim Ergebnis kleiner.
Rechts folgen die Vergleiche beim Ergebnis größer.

Vergleichsbäume

Lemma
Der Vergleichsbaum eines vergleichsbasierten Sortieralgorithmus hat mindestens $n!$ Blätter.

Beweis.
Wenn zwei Permutationen zum gleichen Blatt führen, wird eine von ihnen falsch sortiert.

Es gibt aber $n!$ viele Permutationen. □

Theorem
Jeder vergleichsbasierte Algorithmus benötigt für das Sortieren einer zufällig permutierten Eingabe im Erwartungswert mindestens

$$\log(n!) = n\log n - n\log e - \frac{1}{2}\log n + O(1)$$

viele Vergleiche.

Beweis.
Sei T ein entsprechender Vergleichsbaum. Die mittlere Pfadlänge zu einem Blatt ist am kleinsten, wenn der Baum balanziert ist. In diesem Fall ist die Höhe $\log(n!)$. Stirling-Formel:

$$n! = \frac{1}{\sqrt{2\pi n}}\left(\frac{n}{e}\right)^n\left(1 + O(n^{-1})\right).$$

□

Mergesort

Wir sortieren ein unsortiertes Array durch einen Divide-and-Conquer–Algorithmus:

Eingabe:

12	73	36	71	79	67	3	17	32	31	14	14	33	4	74	23

Ausgabe:

3	4	12	14	14	17	23	31	32	33	36	67	71	73	74	79

Frage: Wie zerteilen wir die Eingabe in zwei unabhängige Teilprobleme?

Mergesort

Eine einfache Strategie: Teilen in der Mitte.

12	73	36	71	79	67	3	17

32	31	14	14	33	4	74	23

1. Teile das Array in der Mitte.
2. Sortiere beide Hälften.
3. Mische beide in eine sortierte Folge.

3	12	17	36	67	71	73	79

4	14	14	23	31	32	33	74

3	4	12	14	14	17	23	31	32	33	36	67	71	73	74	79

Mergesort

Mischen ist der schwierige Teil.

3	12	17	36	67	71	73	79

4	14	14	23	31	32	33	74

3	4	12	14	14	17	23	31	32	33	36	67	71	73	74	79

Algorithmus, der $a[l], \ldots, a[m-1]$ mit $a[m], \ldots, a[r]$ mischt:

Algorithmus

$i := l; j := m; k := l;$
while $k \leq r$ **do**
  if $a[i] \leq a[j]$ **and** $i < m$ **or** $j > r$
  then $b[k] := a[i]; k := k+1; i := i+1$
  else $b[k] := a[j]; k := k+1; j := j+1$ **fi**
od;
for $i = l, ..., r$ **do** $a[k] := b[k]$ **od**

Mergesort

Algorithmus

procedure *mergesort*(l, r) :
  if $l \geq r$ **then return fi**;
  $m := \lceil (r+l)/2 \rceil$;
  mergesort$(l, m-1)$;
  mergesort(m, r);
  $i := l; j := m; k := l;$
  while $k \leq r$ **do**
    if $a[i] \leq a[j]$ **and** $i < m$ **or** $j > r$
    then $b[k] := a[i]; k := k+1; i := i+1$
    else $b[k] := a[j]; k := k+1; j := j+1$ **fi**
  od;
  for $i = l, ..., r$ **do** $a[k] := b[k]$ **od**

Analyse von Mergesort

Das Mischen dauert $\Theta(n)$.

Sei $T(n)$ die Laufzeit. Wir erhalten die Gleichung

$$T(n) = \Theta(n) + T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil).$$

Falls n eine Zweierpotenz ist, gilt

$$T(n) \leq cn + 2T(n/2).$$

Wiederholtes Einsetzen liefert

$$T(n) \leq c\left(n + 2\frac{n}{2} + 4\frac{n}{4} + \cdots\right) = O(n \log n).$$

Mergesort

Mergesort hat interessante Eigenschaften:

1. Der Teile-Teil ist sehr einfach.
2. Der Conquer-Teil ist kompliziert.
3. Er verbraucht viel Speicherplatz (nicht „in-place“)
4. Er ist stabil (gleiche Schlüssel behalten ihre Reihenfolge)
5. ...

Quicksort

Wir sortieren ein unsortiertes Array durch einen Divide-and-Conquer–Algorithmus:

Eingabe:

67	32	17	36	3	4	79	14	31	23	71	74	73	33	14	12

Ausgabe:

3	4	12	14	14	17	23	31	32	33	36	67	71	73	74	79

Frage: Wie zerteilen wir die Eingabe in zwei unabhängige Teilprobleme auf eine andere Art?

Quicksort

Anstatt in der Mitte zu teilen, wählen wir ein Pivot-Element p und teilen in drei Teile:

1. Alle Schlüssel kleiner als p
2. p selbst
3. Alle Schlüssel größer als p

67	32	17	36	3	4	79	14	31	23	71	74	73	33	14	12

33	32	17	36	3	4	12	14	31	23	14	67	73	74	71	79

Sortiere dann rekursiv den ersten und dritten Teil.

Quicksort

67	32	17	36	3	4	79	14	31	23	71	74	73	33	14	12

33	32	17	36	3	4	12	14	31	23	14	67	73	74	71	79

Algorithmus

procedure *quicksort*(L, R) :
  if $R \leq L$ **then return fi**;
  $p := a[L]; l := L; r := R + 1;$
  do
    do $l := l + 1$ **while** $a[l] < p$;
    do $r := r - 1$ **while** $p < a[r]$;
    vertausche $a[l]$ *und* $a[r]$;
  while $l < r$;
  $a[L] := a[l]; a[l] := a[r]; a[r] := p;$
  quicksort$(L, r - 1)$; *quicksort*$(r + 1, R)$

Analyse von Quicksort

Wir nehmen an, die Eingabe besteht aus n paarweise verschiedenen Zahlen und daß jede Permutation gleich wahrscheinlich ist.

Was ist der Erwartungswert der Laufzeit?

Wir werden nur die Anzahl der Vergleiche analysieren.

Sei C_n die erwartete Anzahl von Vergleichen für eine Eingabe der Länge n.

Analyse von Quicksort

1. Offensichtlich ist $C_0 = C_1 = 0$.
2. Die Anzahl der direkten Vergleiche ist $n + 1$.
3. Falls k die endgültige Position des Pivot-Elements ist, dann gibt es noch C_{k-1} und C_{n-k} Vergleiche in den beiden rekursiven Aufrufen.
4. Falls $n \geq 2$, dann

$$C_n = n + 1 + \frac{1}{n}\sum_{k=1}^{n}(C_{k-1} + C_{n-k}).$$

Um einen geschlossenen Ausdruck für C_n zu erhalten, lösen wir diese Rekursionsgleichung.

Analyse von Quicksort

Sei $n \geq 2$. Sei $D_n = nC_n$. Dann

$$\begin{aligned} D_{n+1} - D_n &= \Big((n+1)(n+2) + \sum_{k=1}^{n+1}(C_{k-1} + C_{n+1-k})\Big) - \\ &\quad \Big(n(n+1) + \sum_{k=1}^{n}(C_{k-1} + C_{n-k})\Big) \\ &= 2(n+1) + C_n + C_n = 2(n+1) + 2D_n/n \end{aligned}$$

und wir erhalten die Rekursionsgleichung

$$D_{n+1} = 2(n+1) + \frac{n+2}{n}D_n.$$

Dividieren durch $(n+1)(n+2)$ ergibt

$$\frac{D_{n+1}}{(n+1)(n+2)} = \frac{2}{n+2} + \frac{D_n}{n(n+1)} \text{ für } n \geq 2.$$

Analyse von Quicksort

$$\frac{D_{n+1}}{(n+1)(n+2)} = \frac{2}{n+2} + \frac{D_n}{n(n+1)} \text{ für } n \geq 2.$$

Wiederholtes Einsetzen ergibt für $n \geq 3$

$$\frac{D_n}{n(n+1)} = \frac{2}{n+1} + \frac{2}{n} + \cdots + \frac{2}{5} + \frac{D_2}{6}$$

oder

$$\begin{aligned} C_n &= (n+1)\Big(\frac{2}{n+1} + \frac{2}{n} + \cdots + \frac{2}{5}\Big) + nC_2 \\ &= 2nH_n + O(n) = 2n\ln(n) + O(n). \end{aligned}$$

Die durchschnittliche Laufzeit von Quicksort ist $O(n \log n)$.

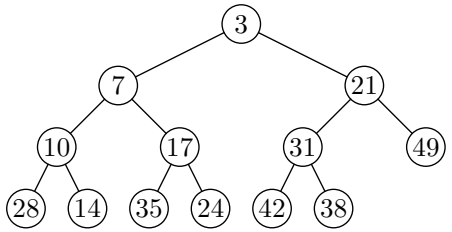

```
public void quicksort() {
  Stack<Pair<Integer, Integer>> stack =
                          new Stack<Pair<Integer, Integer>>();
  stack.push(new Pair<Integer, Integer>(1, size - 1));
  int min = 0;
  for(int i = 1; i < size; i++) if(less(i, min)) min = i;
  D t = get(0); set(0, get(min)); set(min, t);
  while(!stack.isempty()) {
    Pair<Integer, Integer> p = stack.pop();
    int l = p.first(), r = p.second();
    int i = l - 1, j = r, pivot = j;
    do {
      do {i++;} while(less(i, pivot));
      do {j--;} while(less(pivot, j));
      t = get(i); set(i, get(j)); set(j, t);
    } while(i < j);
    set(j, get(i)); set(i, get(r)); set(r, t);
    if(r - i > 1) stack.push(new Pair<Integer, Integer>(i + 1, r));
    if(i - l > 1) stack.push(new Pair<Integer, Integer>(l, i - 1));
  }
}
```

Quicksort

Quicksort hat evenfalls interessante Eigenschaften:

1. Der Teile-Teil ist schwierig.
2. Der Herrsche-Teil ist sehr einfach.
3. Die durchschnittliche Laufzeit ist sehr gut.
4. Die worst-case Laufzeit ist sehr schlecht.
5. Die innere Schleife ist sehr schnell.

Heaps



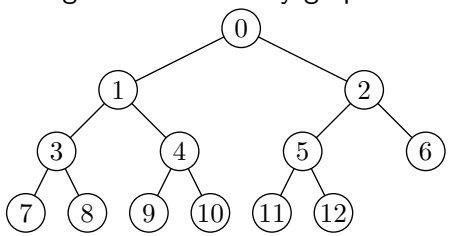
Definition

Ein Heap ist ein Binärbaum, der

- die Heapeigenschaft hat (Kinder sind größer als der Vater),
- bis auf die letzte Ebene vollständig besetzt ist,
- höchstens eine Lücke in der letzten Ebene hat, die ganz rechts liegen muß.

Heaps

Ein Heap kann sehr gut in einem Array gespeichert werden:



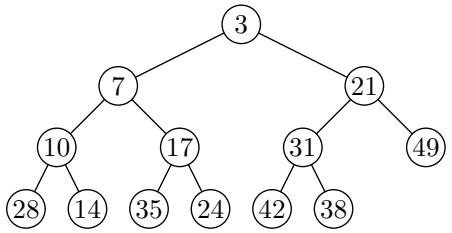
0	1	2	3	4	5	6	7	8	9	10	11	12	13

- Linkes Kind von i ist $2i + 1$
- Rechtes Kind von i ist $2i + 2$
- Vater von i ist $\lfloor (i - 1)/2 \rfloor$

Heaps

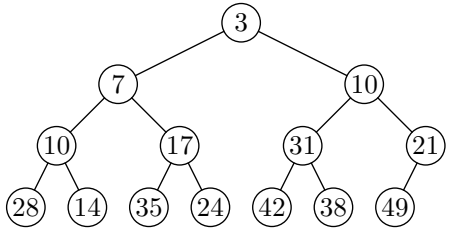
Gegeben sind n Schlüssel $a_1, \ldots, a_n$.

Frage: Wie können wir einen Heap konstruieren, der diese Schlüssel enthält?



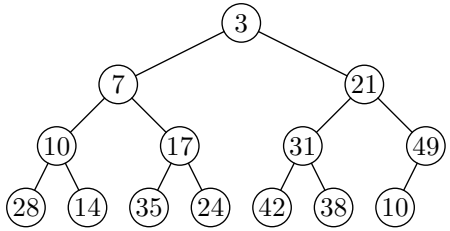
Antwort: Nacheinander in einen leeren Heap einfügen.

Einfügen in einen Heap



Es soll der Schlüssel 10 eingefügt werden.

1. Hänge den Schlüssel an das Ende
2. Die Heap-Eigenschaft ist jetzt verletzt
3. Lasse den neuen Schlüssel zur richtigen Stelle aufsteigen.



Java

```
int bubble_up(int i) {
  while(i > 0 &&!less((i − 1)/2, i)) {
    swap((i − 1)/2, i);
    i = (i − 1)/2;
  }
  return i;
}
```


Ursprüngliches Problem:

Gegeben sind n Schlüssel $a_1, \ldots, a_n$.

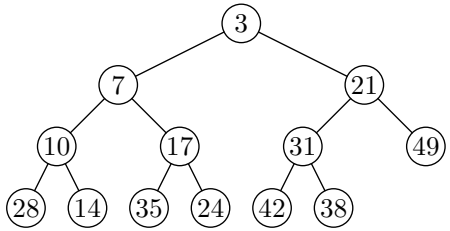
Frage: Wie können wir einen Heap konstruieren, der diese Schlüssel enthält?

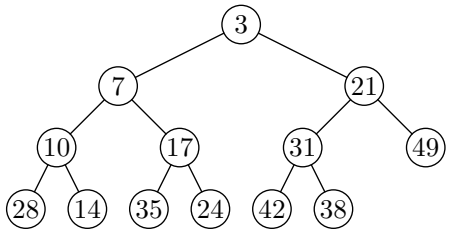
Java

```
public Heap(Array⟨D⟩ a) {
  super();
  for(int i = 0; i < a.size(); i++) set(i, a.get(i));
  for(int i = 1; i < size(); i++) bubble_up(i);
}
```


So wird der Heap schrittweise aufgebaut.

Wir bilden einen Heap aus den Schlüsseln 7, 14, 21, 28, 35, 42, 49, 3, 10, 17, 24, 31, und 38:





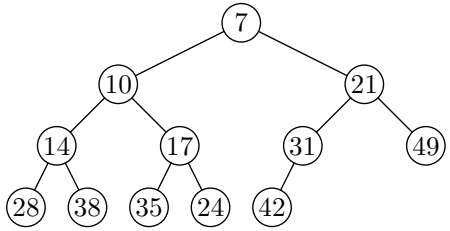
1. Welches ist der größte Schlüssel?
2. Welches ist der kleinste Schlüssel?

Der kleinste Schlüssel ist in der Wurzel.

→ Wiederholtes Entfernen des kleinsten Schlüssels liefert die Schlüssel in aufsteigender Reihenfolge.

Können wir die Wurzel aus einem Heap effizient entfernen?

Entfernen der Wurzel – extract-min

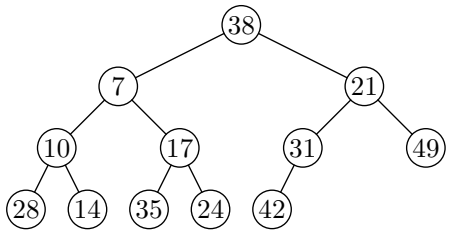


1. Ersetze den Schlüssel der Wurzel durch den Schlüssel des letzten Knoten
2. Lösche den letzten Knoten
3. Jetzt ist die Heap-Eigenschaft verletzt
4. Lasse den Schlüssel in der Wurzel hinuntersinken.

Java

```
int bubble_down(int i) {
  int j;
  while(true) {
    if(2 * i + 2 ≥ size() || less(2 * i + 1, 2 * i + 2)) j = 2 * i + 1;
    else j = 2 * i + 2;
    if(j ≥ size() || less(i, j)) break;
    swap(i, j);
    i = j;
  }
  return i;
}
```


extract-min



Hinuntersinken: Zwei Vergleiche pro Schritt.

Alternative:

1. Hinuntersinken bis zum Blatt.
2. Dann von dort aufsteigen.

Heapsort

Der Algorithmus Heapsort ist jetzt einfach:

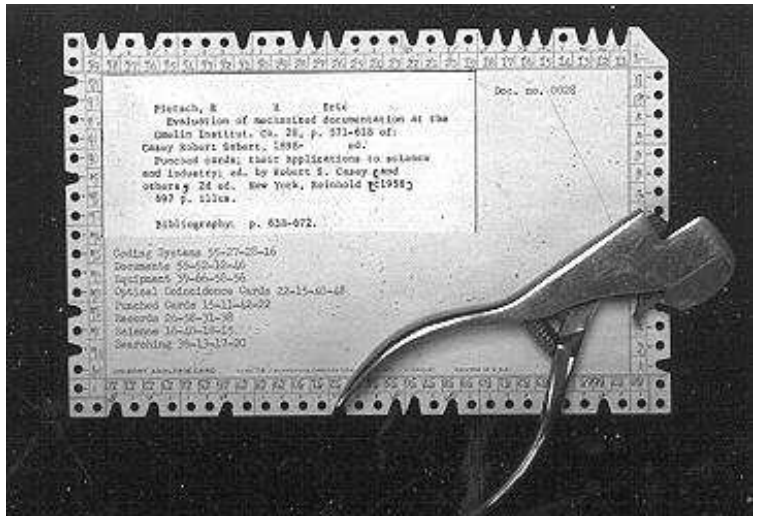
1. Konstruiere einen Heap
2. Entferne wiederholt das kleinste Element
3. Speichere es an der frei werdenden Position

Laufzeit: $O(n \log n)$

Einfügen und extract_min in $O(\log n)$ Zeit

Heapsort ist ein in-place–Verfahren.

Radixsort



Gegeben seien Binärzahlen einer gewissen Länge:

11101010000010001000	01000000110011010001
10110001100111111001	00000000010000000011
01111100111110100100	01111100111110100100
00001011110100010001	00001011110100010001
01011011000110000000	01011011000110000000
11010111000100000110	01101011111000001001
00010111011101000011	00010111011101000011
01101011111000001001	11010111000100000110
10001101100000001100	10001101100000001100
11100000010000001010	11100000010000001010
10101010000011110000	10101010000011110000
10100100001011001010	10100100001011001010
00000000010000000011	10110001100111111001
01000000110011010001	11101010000010001000

Sortiere nach dem ersten Bit.

Bitstrings in Matrix $A[1 \ldots n, 1 \ldots w]$.
Vor dem Bitarray stehe $00 \ldots 0$ und danach $11 \ldots 1$.

Algorithmus

procedure *radix_exchange_sort*(i, a, b) :
  if $i > w$ **then return fi**;
  $s := a$; $t := b$;
  while $s < t$ **do**
    while $A[s, i] = 0$ **do** $s := s + 1$;
    while $A[t, i] = 1$ **do** $t := t - 1$;
    vertausche $A[s]$ *und* $A[t]$
  od;
  vertausche $A[s]$ *und* $A[t]$;
  radix_exchange_sort$(i + 1, a, t)$;
  radix_exchange_sort$(i + 1, s, b)$

11101010000010001000	11101010000010001000
10110001100111111001	01111100111110100100
01111100111110100100	01011011000110000000
00001011110100010001	11010111000100000110
01011011000110000000	10001101100000001100
11010111000100000110	11100000010000001010
00010111011101000011	10101010000011110000
01101011111000001001	10100100001011001010
10001101100000001100	10110001100111111001
11100000010000001010	00001011110100010001
10101010000011110000	00010111011101000011
10100100001011001010	01101011111000001001
00000000010000000011	00000000010000000011
01000000110011010001	01000000110011010001

Sortiere nach dem letzten Bit.
Dann nach dem vorletzten usw. ($\rightarrow$ stabil!)

Straight-Radix-Sort

Algorithmus

procedure *straight_radix_sort*(i) :
  $pos0 := 1; pos1 := n + 1;$
  for $k = 1, ..., n$ **do** $pos1 := pos1 - A[k, i]$ **od**;
  for $k = 1, ..., n$ **do**
    if $A[k, i] = 0$ **then** $B[pos0] := A[k]; pos0 := pos0 + 1$
      else $B[pos1] := A[k]; pos1 := pos1 + 1$ **fi**
  od

Sortiere stabil nach dem i-ten Bit.

Ergebnis ist in B.

Straight-Radix-Sort

Vollständiger Algorithmus:

Algorithmus

procedure *straight_radix_sort* :
  for $i = w, ..., 1$ **do**
    $pos0 := 1; pos1 := n + 1;$
    for $k = 1, ..., n$ **do** $pos1 := pos1 - A[k, i]$ **od**;
    for $k = 1, ..., n$ **do**
      if $A[k, i] = 0$ **then** $B[pos0] := A[k]; pos0 := pos0 + 1$
        else $B[pos1] := A[k]; pos1 := pos1 + 1$ **fi**
    od;
    $A := B$;
  od;

Order-Statistics

Eingabe:

Eine Menge von n Schlüsseln aus einer geordneten Menge
Eine Zahl k, $1 \leq k \leq n$

Ausgabe:

Der k-te Schlüssel (nach Größe)

Spezialfall:

Median, der Schlüssel in der Mitte.

Einfachste Lösung:

1. Sortieren
2. Den Schlüssel an Postition k zurückgeben

Quickselect

Wie Quicksort, aber nur die richtige Seite rekursiv behandeln:

Algorithmus

procedure *quickselect*(k, L, R) :
  if $R \leq L$ **then return** $a[k]$ **fi**;
  $p := a[L]; l := L; r := R + 1;$
  do
    do $l := l + 1$ **while** $a[l] < p$;
    do $r := r - 1$ **while** $p < a[r]$;
    vertausche $a[l]$ *und* $a[r]$
  while $l < r$;
  $a[L] := a[l]; a[l] := a[r]; a[r] := p;$
  if $k = r$ **then return** p
  else if $k < r$ **then return** *quickselect*(k, L, r)
  else return *quickselect*(k, r, R) **fi**

Quickselect – Analyse

Bei Quicksort hatten wir diese Rekursionsgleichung für die Anzahl der Vergleiche:

$$C_n = n + 1 + \frac{1}{n}\sum_{i=1}^{n}(C_{i-1} + C_{n-i}).$$

Für Quickselect gilt:

- Mit W'keit $1/n$ ist das Pivotelement der gesuchte Schlüssel
- Mit W'keit $(k-1)/n$ ist der gesuchte Schlüssel links
- Mit W'keit $(n-k)/n$ ist der gesuchte Schlüssel rechts

Quickselect – Analyse

Für $n > 1$ haben wir:

$$C_n = n + 1 + \frac{1}{n}\sum_{i=1}^{k-1} C_{n-i} + \frac{1}{n}\sum_{i=k+1}^{n} C_{i-1}$$

$$\leq n + 1 + \frac{2}{n}\sum_{i=\lceil n/2 \rceil}^{n-1} C_i$$

Außerdem ist $C_0 = C_1 = 0$.

Zeige mit Induktion:

$$C_n \leq 4n$$

(Einfach → Übungsaufgabe)

Quickselect

Theorem

Quickselect findet den Schlüssel mit Rang k in einer n-elementigen Menge in $O(n)$ Schritten – im Erwartungswert.

Ein deterministischer Algorithmus

1. Falls $n < 30$: Sortiere und finde so das Ergebnis.
2. Bilde $m = \lfloor n/5 \rfloor$ Gruppen mit je fünf Schlüsseln. Bis zu vier Schlüssel bleiben übrig.
3. Berechne den Median jeder Gruppe.
4. Berechne rekursiv den Median p dieser $\lfloor n/5 \rfloor$ Mediane.
5. Verwende p als Pivotelement und führe damit Quickselect aus.

Welchen Rang r hat p?

p ist der Median von $m = \lfloor n/5 \rfloor$ vielen kleinen Medianen
$\rightarrow$ mindestens $m/2 - 1$ kleine Mediane sind kleiner als p
Jeder kleine Median hat zwei Schlüssel die kleiner sind
$\rightarrow$ mindestens $3m/2 - 1$ kleinere Schlüssel als p

Ebenso: Mindestens $3m/2 - 1$ größere Schlüssel als p
Das sind jeweils $3\lfloor n/5 \rfloor/2 - 1 \geq 3n/10 - 3/2 \geq n/4$

Deterministisches Selektieren – Analyse

Die Anzahl der Vergleiche ist jetzt

$$C_n \leq n + 1 + C_{\lfloor n/5 \rfloor} + C_{\lfloor 3n/4 \rfloor}$$

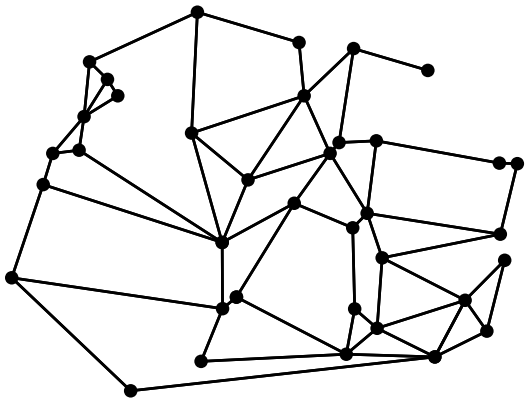
falls $n > 30$ und $O(1)$ falls $n \leq 30$:

- $C_{\lfloor n/5 \rfloor}$ für das rekursive Finden der Mediane
- $C_{\lfloor 3n/4 \rfloor}$ für die nächste Suche

Es folgt $C_n = O(n)$, da $1/5 + 3/4 < 1$.

Wir können also den Schlüssel mit Rang k in linearer Zeit finden.

Graphen



Graphen

Definition
Ein ungerichteter Graph ist ein Paar (V, E), wobei V die Menge der Knoten und $E \subseteq \binom{V}{2}$ die Menge der Kanten ist.

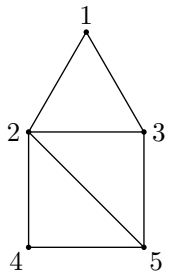
Definition
Ein gerichteter Graph ist ein Paar (V, E), wobei V die Menge der Knoten und $E \subseteq V \times V$ die Menge der Kanten ist.

Oft betrachten wir Graphen mit Knoten- oder Kantengewichten.

Dann gibt es zusätzlich Funktionen $V \rightarrow \mathbf{R}$ oder $E \rightarrow \mathbf{R}$.

Darstellung von Graphen

Adjazenzmatrix



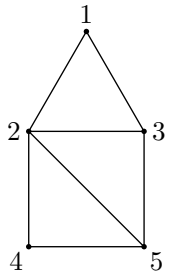
$$\begin{pmatrix} \cdot & 1 & 1 & 0 & 0 \\ \cdot & \cdot & 1 & 1 & 1 \\ \cdot & \cdot & \cdot & 0 & 1 \\ \cdot & \cdot & \cdot & \cdot & 1 \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

Speicherbedarf: $\Theta(|V|^2)$

Für gerichtete Graphen wird die ganze Matrix verwendet.

Darstellung von Graphen

Adjazenzliste



1	2, 3
2	1, 3, 4, 5
3	1, 2, 5
4	2, 5
5	2, 3, 4

Speicherbedarf: $\Theta(|V| + |E|)$.

In $O(n^2)$ Schritten kann zwischen beiden Darstellungen konvertiert werden.

Darstellung von Graphen

Java

```
public class Graph {
 int n; // number of nodes
 int m; // number of edges
 Set⟨Node⟩ nodes;
 Map⟨Node, List⟨Node, Edge⟩⟩ neighbors;
```

Wir wählen die Darstellung durch eine Adjazenzliste.

Java

```
class Node implements Comparable⟨Node⟩ {
  String name;
  double weight;
  static int uniq = 0;
  int f;
  public Node() {f = uniq; name = ":" + uniq++; weight = 0.0; }
  public Node(double w) {this(); weight = w; }
  public String toString() {return name; }
  public int hashCode() {return f; }
  public boolean equals(Object o) {
    if (this ≡ o) return true;
    if (!(o instanceof Node)) return false;
    return f ≡ ((Node)o).f;
  }
  public int compareTo(Node n) {
    if(weight ≡ n.weight) return f − n.f;
    else if(weight < n.weight) return − 1;
    else return 1;
  }
}
```

```
class Edge implements Comparable<Edge> {
  Node s, t;
  double weight;
  static int uniq = 0;
  int f;
  public Edge(Node i, Node j) {f = uniq++; s = i; t = j; weight = 0; }
  public Edge(Node i, Node j, double w) {this(i, j); weight = w; }
  public double weight() {return weight; }
  public void setweight(double d) {weight = d; }
  public int hashCode() {return f; }
  public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Edge)) return false;
    return f == ((Edge)o).f;
  }
  public int compareTo(Edge e) {
    if(weight == e.weight) return f - e.f;
    else if(weight < e.weight) return -1;
    else return 1;
  }
}
```


Java

```
public Node addnode() {
  Node newnode = new Node();
  nodes.insert(newnode);
  neighbors.insert(newnode, new List<Node, Edge>());
   // n++;
  return newnode;
}
```


Java

```
public void addedge(Node u, Node v, Double w) {
  neighbors.find(u).insert(v, new Edge(u, v, w));
  m++;
}
```

Java

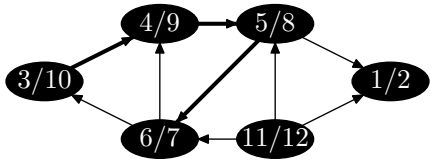
```
public boolean isadjacent(Node u, Node v) {
  return neighbors.find(u).iselement(v);
}
```


Tiefensuche

Tiefensuche ist ein sehr mächtiges Verfahren, das iterativ alle Knoten eines gerichteten oder ungerichteten Graphen besucht.

- Sie startet bei einem gegebenen Knoten und färbt die Knoten mit den Farben weiß, grau und schwarz.
- Sie berechnet einen gerichteten Tiefensuchwald, der bei einem ungerichteten Graph ein Baum ist.
- Sie ordnet jedem Knoten eine Anfangs- und eine Endzeit zu.
- Alle Zeiten sind verschieden.
- Die Kanten des Graphen werden als Baum-, Vorwärts-, Rückwärts- oder Querkanten klassifiziert.

Tiefensuche – Beispiel



- Die Kanten des Tiefensuchwaldes sind dick dargestellt.
- Ein Knoten ist anfangs weiß.
- Ein Knoten ist grau, während er aktiv ist.
- Danach wird er schwarz.

RWTH AACHEN

Java

```
public void DFS(Map⟨Node, Integer⟩ d, Map⟨Node, Integer⟩ f,
    Map⟨Node, Node⟩ p) {
  SimpleIterator⟨Node⟩ it;
  Map⟨Node, Integer⟩ color = new Hashtable⟨Node, Integer⟩();
  for(it = nodeiterator(); it.more(); it.step())
    color.insert(it.key(), WHITE);
  int time = 0;
  for(it = nodeiterator(); it.more(); it.step())
    if(color.find(it.key()) ≡ WHITE)
      time = DFS(it.key(), time, color, d, f, p);
}
```

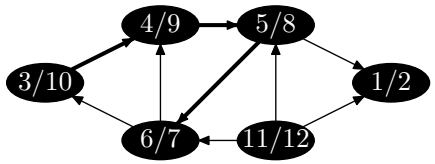
RWTH AACHEN

Java

```
public int DFS(Node u, int t,
    Map⟨Node, Integer⟩ c, Map⟨Node, Integer⟩ d,
    Map⟨Node, Integer⟩ f, Map⟨Node, Node⟩ p) {
  d.insert(u, ++t);
  c.insert(u, GRAY);
  Iterator⟨Node, Edge⟩ adj;
  for(adj = neighbors.find(u).iterator(); adj.more(); adj.step())
    if(c.find(adj.key()) ≡ WHITE) {
      p.insert(adj.key(), u);
      t = DFS(adj.key(), t, c, d, f, p);
    }
  f.insert(u, ++t);
  c.insert(u, BLACK);
  return t;
}
```

RWTH AACHEN

Taxonomie der Kanten

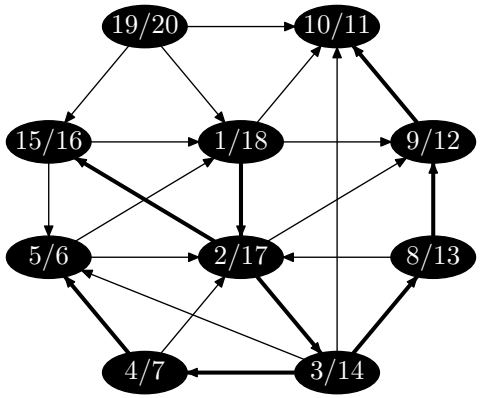


1. Eine Baumkante ist im DFS-Wald (geht von einem Knoten zu einem seiner Kinder im DFS-Wald).
2. Eine Vorwärtskante geht von einem Knoten zu einem seiner Nachfahren im DFS-Wald (aber nicht Kind).
3. Eine Rückwärtskante geht von einem Knoten zu einem seiner Vorfahren im DFS-Wald.
4. Eine Querkante verbindet zwei im DFS-Wald unvergleichbare Knoten.

Frage: Welchen Typ hat jede Kante in diesem Beispiel?

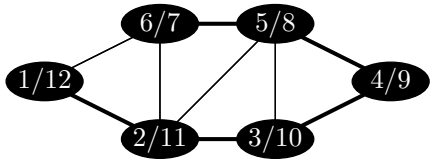
RWTH AACHEN

Taxonomie der Kanten



Frage: Welchen Typ hat jede Kante in diesem Beispiel?

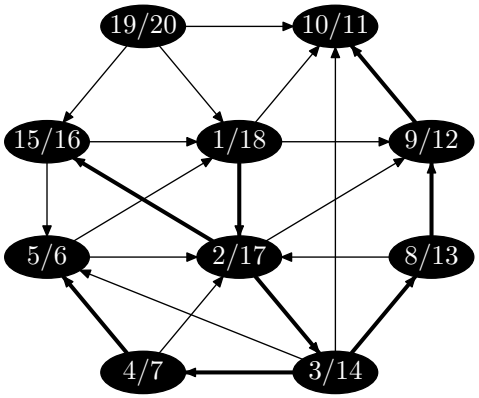
DFS – Ungerichtete Graphen



Wir erhalten immer einen Baum, wenn der Graph zusammenhängend ist.

Implementierung: Eine ungerichtete Kante wird durch Kanten in beide Richtungen dargestellt.

Taxonomie der Kanten



Betrachte Kante (u, v):

1. $d(u) < d(v)$ und $f(v) < f(u) \iff$ Baum- oder Vorwärtskante
2. $d(v) < d(u)$ und $f(u) < f(v) \iff$ Rückwärtskante
3. sonst Querkante

Tiefensuche

Theorem
Gegeben sei ein gerichteter Graph $G = (V, E)$ (in Adjazenzlistendarstellung).
Durch Tiefensuche kann ein DFS-Wald inklusive der Funktionen $d : V \to \mathbf{N}$, $f : V \to \mathbf{N}$ in $O(|V| + |E|)$ Schritten (also in linearer Zeit) berechnet werden.

Beweis.
(Skizze) Solange ein Knoten grau ist, wird jede inzidente Kante einmal besucht. Jeder Knoten wechselt seine Farbe nur zweimal, jedesmal mit konstantem Aufwand.
Jede Kante wird daher ebenfalls nur einmal besucht. □

Zusammenhangskomponenten

Definition

Es sei $G = (V, E)$ ein ungerichteter Graph. Ein Pfad der Länge k von u_1 nach u_{k+1} ist eine Folge $(u_1, u_2), (u_2, u_3), \ldots, (u_k, u_{k+1})$ von Kanten aus E wobei $u_1, \ldots, u_{k+1}$ paarweise verschieden sind.

Wir sagen u und v sind zusammenhängend, wenn es einen Pfad von u nach v gibt.

Eine Menge $C \subseteq V$ ist eine Zusammenhangskomponente, wenn alle Knoten in C zusammenhängend sind und es keine echte Obermenge von C mit dieser Eigenschaft gibt.
(Alternativ: Die Zusammenhangskomponenten sind die Äquivalenzklassen der Relation „zusammenhängend“.)

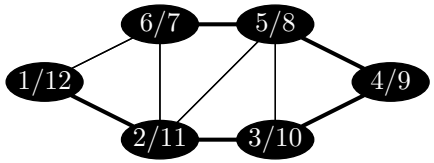
Zusammenhangskomponenten

Theorem

Die Zusammenhangskomponenten eines ungerichteten Graphen können in linearer Zeit gefunden werden.

Beweis.

Die Knoten, die jeweils in einem Aufruf der Tiefensuche schwarz werden, gehören zu einer Komponente. □



Finden von Kreisen

Theorem

Gegeben sei ein gerichteter Graph G. Dann können wir in linearer Zeit feststellen, ob G azyklisch ist (keine Kreise enthält).

Beweis.

Führe eine Tiefensuche auf G aus.
Behauptung: G ist genau dann azyklisch, wenn es keine Rückwärtskanten gibt.
$\Rightarrow$ Wenn es eine Rückwärtskante von u nach v gibt, dann gibt es auch einen Pfad von v nach u im DFS-Wald. Dies ist ein Kreis.
$\Leftarrow$ Angenommen es gibt einen Kreis. Sei u der Knoten auf dem Kreis mit minimalem $d(u)$ und v der Knoten auf dem Kreis vor u. Dann gilt $d(u) < d(v)$ und $f(v) < d(u)$.
Also ist (v, u) eine Rückwärtskante. □

Starke Zusammenhangskomponenten

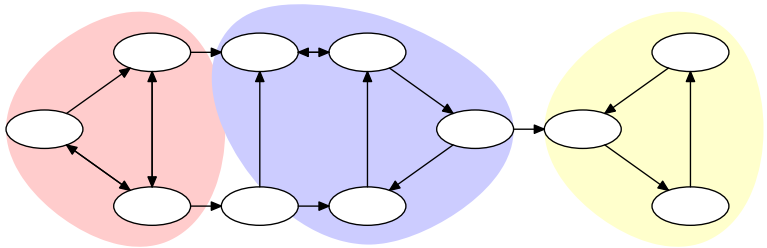
Definition

Es sei $G = (V, E)$ ein gerichteter Graph. Ein Pfad der Länge k von u_1 nach u_{k+1} ist eine Folge $(u_1, u_2), (u_2, u_3), \ldots, (u_k, u_{k+1})$ von Kanten aus E wobei $u_1, \ldots, u_{k+1}$ paarweise verschieden sind.

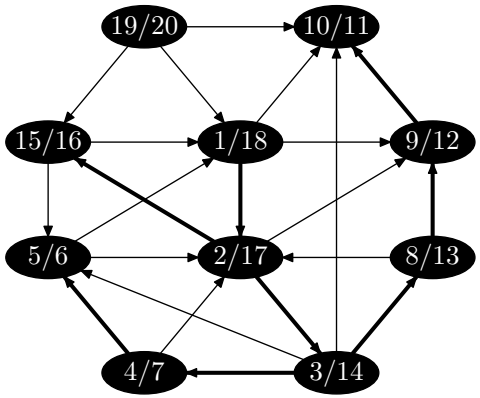
Eine Menge $C \subseteq V$ ist eine starke Zusammenhangskomponente, wenn es Pfade zwischen allen Paaren von Knoten in C gibt und es keine echte Obermenge von C mit dieser Eigenschaft gibt.

Die starken Komponenten sind eine Verfeinerung der Zusammenhangskomponenten des entsprechenden ungerichteten Graphen.

Starke Komponenten – Beispiel

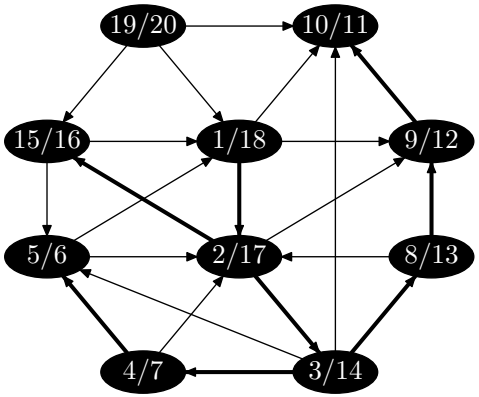


Es gibt genau vier starke Komponenten.



Lemma
Der Knoten mit der größten finish time nach einer DFS ist in einer Quellenkomponente.

Beweis.
Er ist Wurzel eines DFS-Baums T. Wenn es einen anderen DFS-Baum gäbe, von dem eine Kante nach T führte, dann müßte dieser danach besucht werden. □



Quellenkomponente: Starke Komponente, in die keine Kante hineinführt

Senkenkomponente: Starke Komponente, aus der keine Kante herausführt

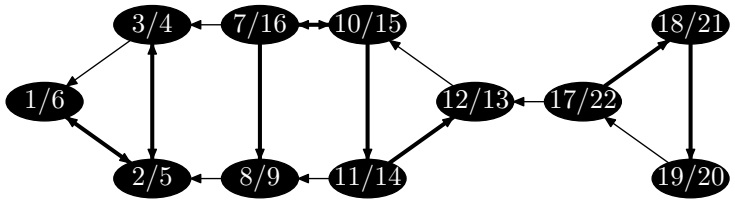
G_r: Wie G, aber Richtung der Kanten umgekehrt

Korollar
Der Knoten mit der größten finish time nach einer DFS in G_r ist in einer Senkenkomponente von G.

Die starken Komponenten von G und G_r sind identisch.

Starke Komponenten

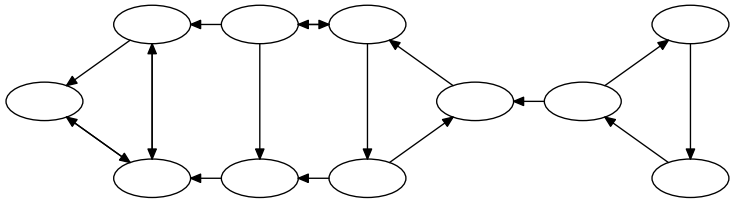
Lemma
Gegeben sei ein gerichteter Graph G mit einem DFS-Wald. Der DFS-Wald bleibt samt discover und finish times ein möglicher DFS-Wald, wenn die Quellenkomponente entfernt wird, die den Knoten mit maximaler finish time enthält.



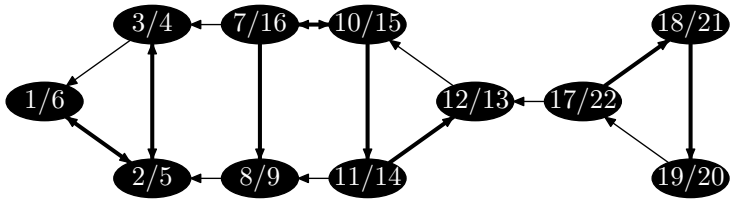
Beweis.
Der Baum spannt die ganze starke Komponente auf. Er wurde als letzter besucht. □

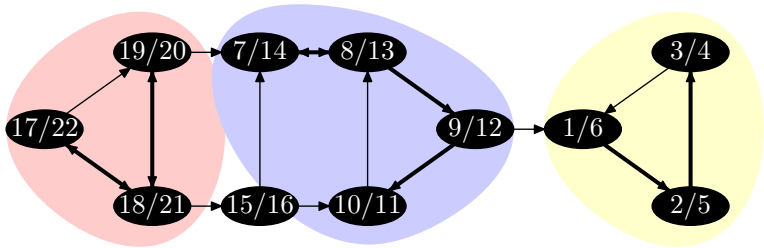
Starke Komponenten – Algorithmus von Kosaraju

1. Bilde G_r (Kanten umdrehen)
2. Führe Tiefensuche auf G_r aus
3. Ordne die Knoten nach absteigender finish time $f(v)$
4. Führe in dieser Reihenfolge Tiefensuche auf G aus
5. Jeder Baum im DFS-Wald spannt eine starke Komponente auf



Starke Komponenten – Algorithmus von Kosaraju



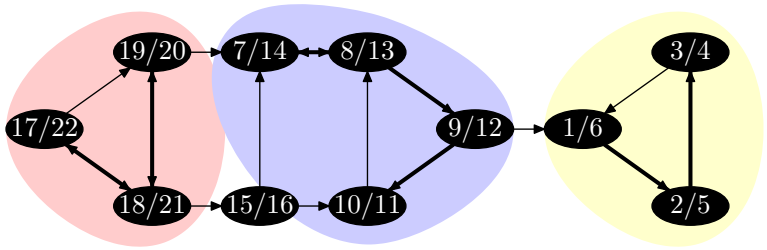


Theorem

Der Algorithmus von Kosaraju berechnet die starken Komponenten eines gerichteten Graphen.

Beweis.

Die DFS auf G beginnt in einem Knoten einer Senkenkomponente von G. Dabei entsteht ein DFS-Baum, der diese starke Komponente aufspannt.
Die DFS fährt dann wieder in einer Senkenkomponente des Restgraphen fort und so weiter. □



Topologisches Sortieren

Ein reflexiv-transitive Hüllen eines DAG entspricht einer Halbordnung.

Definition

Es sei M eine Menge. Wir sagen $\leq \subseteq M \times M$ ist eine Halbordnung auf M, wenn

1. Für alle $x \in M$ gilt $x \leq x$
2. Für alle $x, y \in M$ gilt $x \leq y \wedge y \leq x \Rightarrow x = y$
3. Für alle $x, y, z \in M$ gilt $x \leq y \wedge y \leq z \Rightarrow x \leq z$

Topologisches Sortieren:

Bette eine Halbordnung in eine Ordnung ein.

Ordne die Knoten eines DAG so, daß keine Kante von einem größeren zu einem kleineren Knoten führt.

Topologisches Sortieren – Anwendungen

Topologisches Sortieren kann viele Probleme lösen.

- In welcher Reihenfolge sollen Vorlesungen gehört werden?
- Wie baut man ein kompliziertes Gerät?
- Entwurf von Klassenhierarchien.
- Task scheduling.
- Tabellenkalkulation.
- Optimierung beim Compilieren.
- ...

Frage: Kann jede Halbordnung topologisch sortiert werden?

Topologisches Sortieren

Theorem
Jede endliche Halbordnung kann in eine (totale) Ordnung eingebettet werden.

Beweis.
Sei $\leq \subseteq M \times M$ eine endliche Halbordnung auf M.
Da M endlich ist, dann gibt es ein $x \in M$, so daß es kein $y \in M$ mit $y \neq x$ und $y \leq x$ gibt.
Wir können die Ordnung mit x als kleinstem Element beginnen und dann eine Ordnung von $M \setminus \{x\}$ anfügen.
Durch Induktion sieht man, daß dies eine (totale) Ordnung auf M ist. □

Frage: Was ist mit unendlichen Mengen?

Topologisches Sortieren

Theorem
G sei ein DAG. Wenn wir die Knoten von G nach den finish times einer DFS umgekehrt anordnen, sind sie topologisch sortiert.

Beweis.
Es seien u und v Knoten von G mit $f(u) < f(v)$
Daher kommt u nach v in der konstruierten Ordnung.

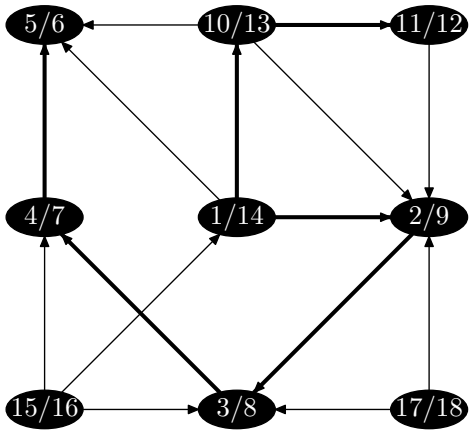
Wir zeigen, daß es keine Kante von u nach v gibt:

Falls $d(v) < d(u)$ müßte es eine Rückwärtskante sein, die es nicht gibt (DAG).

Also $d(v) > f(u)$ und v blieb weiß bis zur finish time von u.

Nur möglich, wenn keine Kante von u nach v. □

Topologisches Sortieren – Beispiel

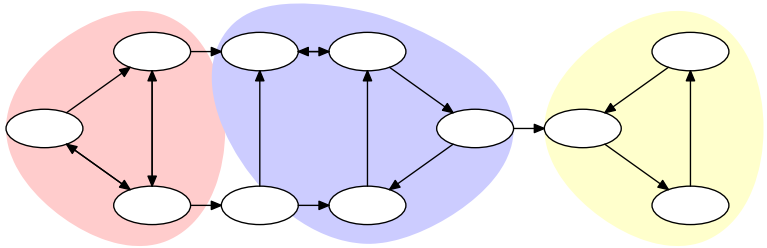


Topologisches Sortieren in $O(|V| + |E|)$ Schritten.

s-t Connectivity

Gegeben: Knoten s und t in gerichtetem Graph

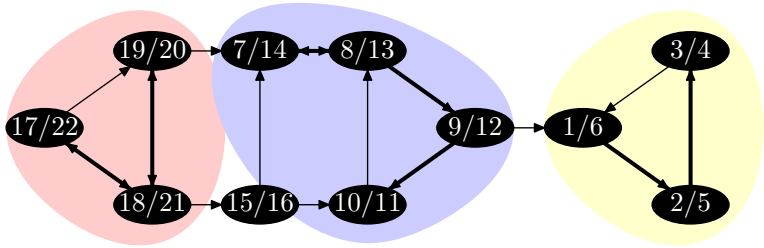
Frage: Ist t von s erreichbar (durch einen gerichteten Pfad)?



s-t Connectivity

Führe eine DFS aus, starte bei s.

Pfad von s nach $t \iff f(t) < f(s)$.

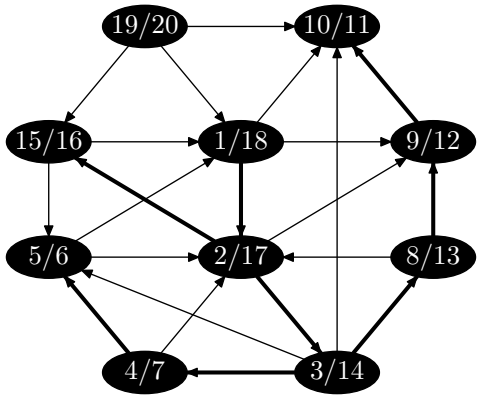


Beweis: Wenn s schwarz wird, sind alle von s erreichbaren Knoten schwarz.

s-t Connectivity

Theorem

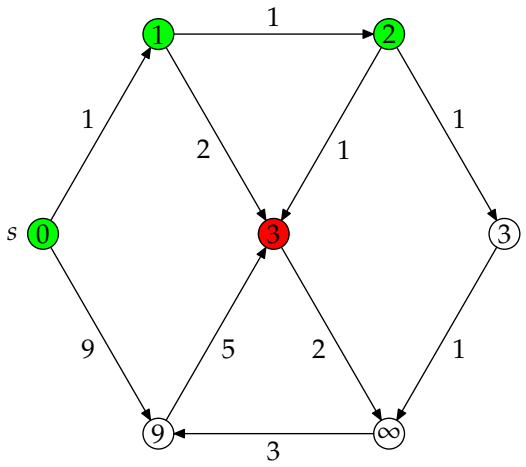
Gegeben sei ein gerichteter Graph $G = (V, E)$ und $s \in V$.
Wir können in linearer Zeit alle von s erreichbaren Knoten finden.



Single Source Shortest Paths

Gegeben ein gerichteter Graph $G = (V, E)$ mit nicht-negativen Kantengewichten *length* : $E \to \mathbf{Q}$ und ein Knoten $s \in V$, finde die kürzesten Wege von s zu allen Knoten.

- Wir lösen das Problem mit dynamischer Programmierung.
- Menge F von Knoten, deren Abstand bekannt ist.
- Anfangs ist $F = \{s\}$.
- F wird in jeder Iteration größer.
- Invariante: Kein Knoten $v \notin F$ hat kleineren Abstand zu s als jeder Knoten in F.



- Grüne Knoten: F
- Roter Knoten aktiv, relaxiert seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.

Korrektheit

Lemma

- *Jeder grüne Knoten enthält den Abstand von s.*
- *Jeder weiße Knoten enthält Abstand von s über grüne Knoten.*

Beweis.

Sei v einer weißer Knoten, dessen Beschriftung minimal ist.
Betrachte einen kürzesten Pfad von s nach v und den ersten weißen Knoten u auf diesem Pfad.
Es gilt $u = v$, da der Abstand von s zu u mindestens so groß ist wie der Abstand von s zu v.
Wenn ein Knoten grün wird, garantiert die Relaxation, daß die zweite Bedingung weiter gilt. □

Der Algorithmus von Dijkstra

Algorithmus

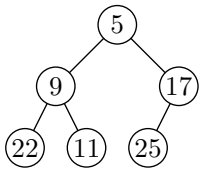
procedure *Dijkstra*(s) :
$Q := V - \{s\}$;
for v *in* Q **do** $d[v] := \infty$ **od**;
$d[s] := 0$;
while $Q \neq$ *emptyset* **do**
  choose v *in* Q *with minimal* $d[v]$;
  $Q := Q - \{v\}$;
  forall u *adjacent* **to** v **do**
    $d[u] := min\{d[u], d[v] + length(v, u)\}$
  od
od

Wie implementieren wir Q?

Priority Queues (Prioritätswarteschlangen)

Operationen einer Prioritätswarteschlange Q:

1. Einfügen von x mit Gewicht w (insert)
2. Finden und Entfernen eines Elements mit minimalem Gewicht (extract-min)
3. Das Gewicht eines Elements x auf w verringern (decrease-weight)



Heap: alle Operationen in $O(\log n)$ Schritten
(n ist die aktuelle Anzahl von Elementen im Heap)

Algorithmus von Dijkstra – Laufzeit

Theorem
Der Algorithmus von Dijkstra berechnet die Abstände von s zu allen anderen Knoten in $O((|V| + |E|) \log |V|)$ Schritten.

Beweis.
Es werden $|V|$ Einfügeoperationen, $|V|$ extract-mins und $|E|$ decrease-keys ausgeführt.
Verwenden wir einen Heap für die Prioritätswarteschlange, ergibt sich die verlangte Laufzeit. □

Ein Fibonacci-Heap benötigt für ein Einfügen und extract-min $O(\log n)$ und für decrease-key nur $O(1)$ amortisierte Zeit.

Dijkstra: $O(|V| \log |V| + |E|)$

Java

```
public void dijkstra(Node s, Map⟨Node, Node⟩ pred) {
 SimpleIterator⟨Node⟩ it;
 for(it = nodeiterator(); it.more(); it.step()) it.key().weight = 1e10;
 s.weight = 0.0;
 Heap⟨Node⟩ H = new Heap⟨Node⟩();
 for(it = nodeiterator(); it.more(); it.step()) H.insert(it.key());
 while(!H.isempty()) {
   Node v = H.extract_min();
   Iterator⟨Node, Edge⟩ inc;
   for(inc = neighbors.find(v).iterator(); inc.more(); inc.step())
     if(inc.key().weight > v.weight + inc.data().weight) {
       pred.insert(inc.key(), v);
       inc.key().weight = v.weight + inc.data().weight;
       H.decrease_key(inc.key());
     }
 }
}
```


Spezialfall DAG

Können wir kürzeste Pfade in DAGs schneller finden?

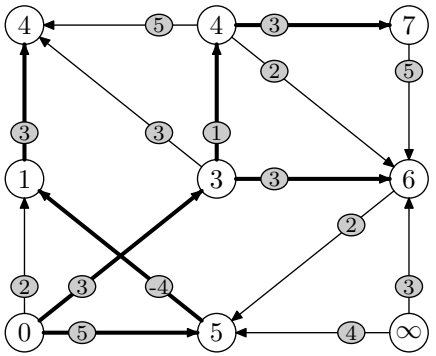
Theorem
Kürzeste Pfade von einem Knoten s in einem DAG können in linearer Zeit gefunden werden.

Beweis.
Relaxiere Knoten in topologischer Reihenfolge.

Laufzeit: $O(|V| + |E|)$.
Korrektheit: Jeder Knoten, der relaxiert, kennt zu diesem Zeitpunkt seinen echten Abstand. □

Frage: Sind negative Gewichte hier erlaubt?

Kürzeste Wege mit negativen Kantengewichten



Dijkstra funktioniert nicht mit negativen Kantengewichten.

Der Algorithmus von Bellman und Ford

Idee:

- Relaxiere alle Kanten
- Wiederhole dies, bis keine Änderung

Warum korrekt?

Induktion über die Länge eines kürzesten Pfads.
(Also genügen n Wiederholungen)

Was passiert bei Kreisen mit negativem Gewicht?

$\rightarrow$ Keine Terminierung.

Der Algorithmus von Bellman und Ford

Algorithmus

function *Bellman* − *Ford*(s) **boolean** :
for v *in* V **do** $d[v] := \infty$ **od**;
$d[s] := 0$;
for $i = 1$ **to** $|V| - 1$ **do**
  forall (u, v) *in* E **do**
    $d[u] := min\{d[u], d[v] + length(v, u)\}$
  od
od;
forall (u, v) *in* E **do**
  if $d[u] > d[v] + length(v, u)$ **then return** *false* **fi**
od;
return *true*

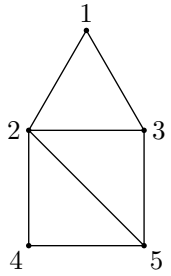
Der Algorithmus von Bellman und Ford

Theorem
Gegeben sei ein gerichteter Graph $G = (V, E)$ mit Kantengewichten $E \rightarrow \mathbf{R}$ und ein Knoten $s \in V$. Wir können in $O(|V| \cdot |E|)$ feststellen, ob ein Kreis mit negativem Gewicht existiert, und falls nicht, die kürzesten Wege von s zu allen Knoten berechnen.

Beweis.
Wir haben die Korrektheit bereits nachgewiesen.
Zur Laufzeit: Jeder Knoten wird $|V|$ mal relaxiert, also wird auch jede Kante $|V|$ mal relaxiert (in konstanter Zeit). □

Rekapitulation: Darstellung von Graphen

Adjazenzliste



1	2, 3
2	1, 3, 4, 5
3	1, 2, 5
4	2, 5
5	2, 3, 4

Alle bisherigen Algorithmen:

Adjazenzliste gute Darstellung

Kritische Operation: Alle ausgehenden Kanten besuchen.

All Pairs Shortest Paths

Eingabe: Gerichteter Graph mit Kantengewichten

Ausgabe: Abstände und kürzeste Wege zwischen allen Knotenpaaren

Laufzeit $O((|V|^2 + |V| \cdot |E|) \log |V|)$ mit Dijkstra.

$\rightarrow$ Wende Dijkstra auf jeden Knoten an.

Algorithmus von Floyd

Algorithmus

procedure *Floyd*1 :
for $i = 1, ..., n$ **do**
  for $j = 1, ..., n$ **do** $d[i, j, 0] := length[i, j]$ **od**
od;
for $k = 1, ..., n$ **do**
  for $i = 1, ..., n$ **do**
    for $j = 1, ..., n$ **do**
      $d[i, j, k] := min\{d[i, j, k-1], d[i, k, k-1] + d[k, j, k-1]\}$
    od
  od
od

Der Abstand von i nach j ist in $d[i, j, n]$ zu finden.

Theorem

Die kürzesten Wege zwischen allen Knotenpaaren eines gerichteten Graphen $G = (V, E)$ können in $O(|V|^3)$ Schritten gefunden werden.

Beweis.

Per Induktion: $d[i, j, k]$ enthält die Länge des kürzesten Pfades von i nach j, wenn nur $1, \ldots, k$ als Zwischenstationen erlaubt sind. Dann enthält $d[i, j, n]$ den wirklichen Abstand. □

Algorithmus von Floyd

Einfachere Version:

Algorithmus

procedure *Floyd* :
for $i = 1, ..., n$ **do**
  for $j = 1, ..., n$ **do** $d[i, j] := length[i, j]$ **od**
od;
for $k = 1, ..., n$ **do**
  for $i = 1, ..., n$ **do**
    for $j = 1, ..., n$ **do**
      $d[i, j] := min\{d[i, j], d[i, k] + d[k, j]\}$
    od
  od
od

Spezialfall: Transitive Hülle – Algorithmus von Warshall

Frage: Zwischen welchen Knotenpaaren gibt es einen Weg?

Algorithmus

procedure *Warshall* :
for $i = 1, ..., n$ **do**
  for $j = 1, ..., n$ **do** $D[i,j] := A[i,j]$ **od**
od;
for $k = 1, ..., n$ **do**
  for $i = 1, ..., n$ **do**
    for $j = 1, ..., n$ **do**
      $D[i,j] := D[i,j] \;||\; D[i,k] \;\&\&\; D[k,j]$
    od
  od
od

Transitive Hülle

Theorem

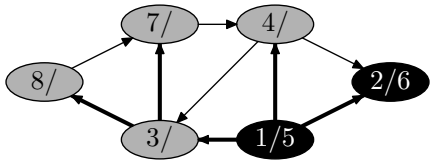
Wir können die transitive Hülle eines gerichteten Graphen $G = (V, E)$, der k starke Zusammenhangskomponenten hat, in $O(|V| + |E'| + k^3)$ Schritten berechnen, wobei E' die Kanten der transitiven Hülle sind.

Beweis.

1. Berechne die starken Zusammenhangskomponenten
2. Schrumpfe jede SCC X zu einem Knoten
3. Berechne die transitive Hülle H dieses Graphen
4. Für jede Kante (X, Y) in H gib alle Kanten (x, y) mit $x \in X, y \in Y$ aus.

□

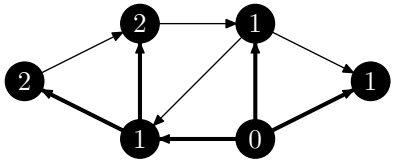
Breitensuche



Breitensuche ist das „Gegenteil“ von Tiefensuche.

- Tiefensuche: Aktive Knoten auf Stack
- Breitensuche: Aktive Knoten in FIFO-Queue
- Intelligente Suche: Weder Stack noch Queue
- Breitensuchbaum enthält kürzeste Pfade (ungewichtet)
- Discovery- und Finishzeiten wenig Anwendungen

Breitensuche



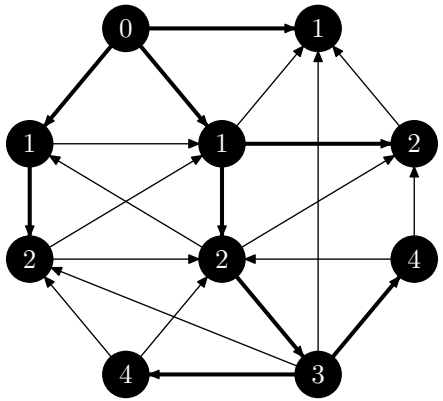
Knoten wird aktiv:

Abstand berechnen und Kante in BFS-Baum

Anwendung:

Alle Abstände und kürzesten Wege von s berechnen

Breitensuche – Größeres Beispiel




```
Java
public void BFS(Node s, Map⟨Node, Node⟩ p) {
  SimpleIterator⟨Node⟩ it;
  Map⟨Node, Integer⟩ color = new Hashtable⟨Node, Integer⟩();
  Queue⟨Node⟩ q = new Queue⟨Node⟩();
  for(it = nodeiterator(); it.more(); it.step()) {
    it.key().weight = 1.0e10; color.insert(it.key(), 0); } // white
  s.weight = 0.0; q.enqueue(s); color.insert(s, 1); // gray
  while(!q.isempty()) {
    Node v = q.dequeue();
    Iterator⟨Node, Edge⟩ adj;
    for(adj = neighbors.find(v).iterator(); adj.more(); adj.step()) {
      Node u = adj.key();
      if(color.find(u) ≡ 0) {
        color.insert(u, 1); // gray
        u.weight = v.weight + 1; p.insert(u, v); q.enqueue(u);
      }
    }
    color.insert(v, 2); // black
  }
}
```

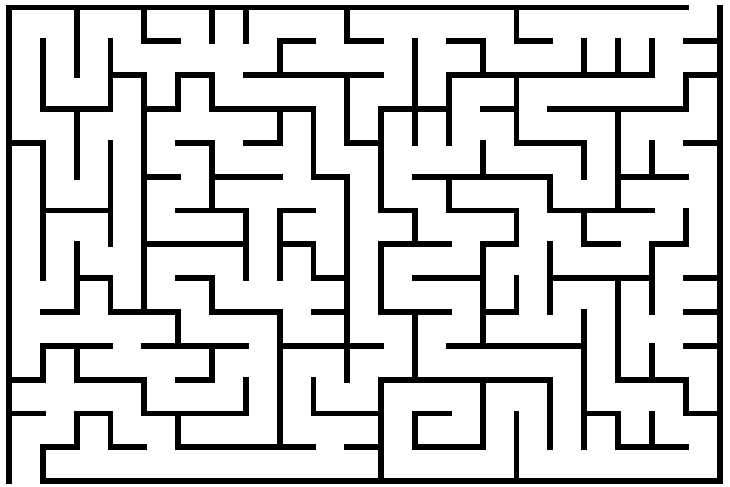

Breitensuche

Theorem
Gegeben sei ein gerichteter oder ungerichteter Graph $G = (V, E)$ und ein Knoten $s \in V$.
Die kürzesten Wege von s zu allen anderen Knoten und die zugehörigen Abstände können in $O(|V| + |E|)$ berechnet werden.

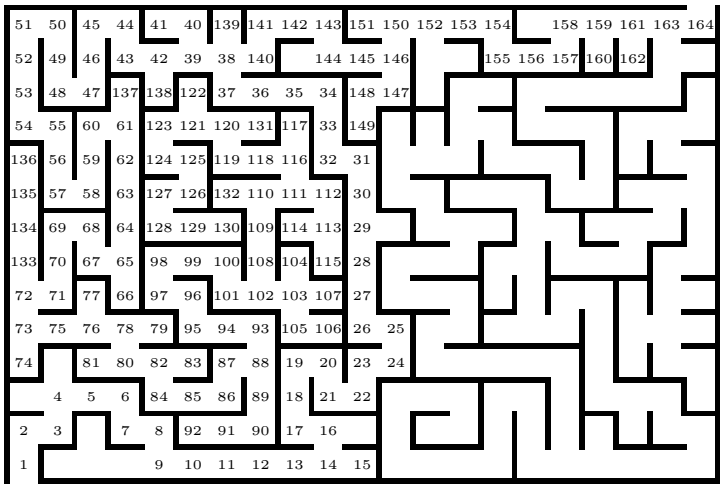
Beweis.
Wir verwenden Breitensuche.
Die Laufzeit ist offensichtlich linear.

Der Korrektheitsbeweis sei hier weggelassen (etwas lang und technisch). □

Beispiel

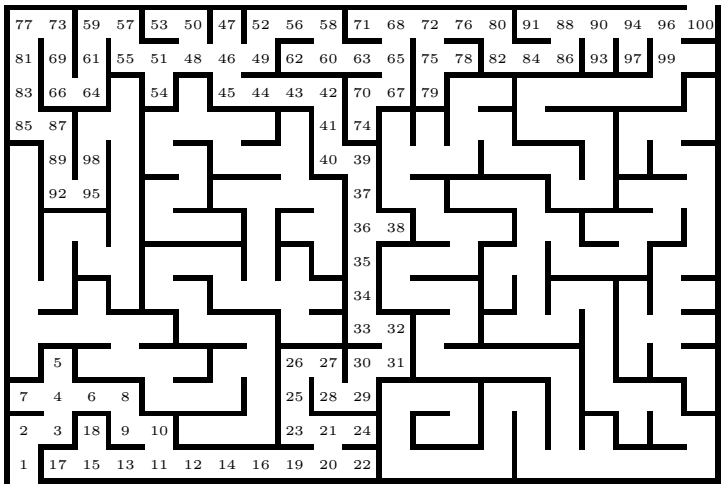


Beispiel: DFS



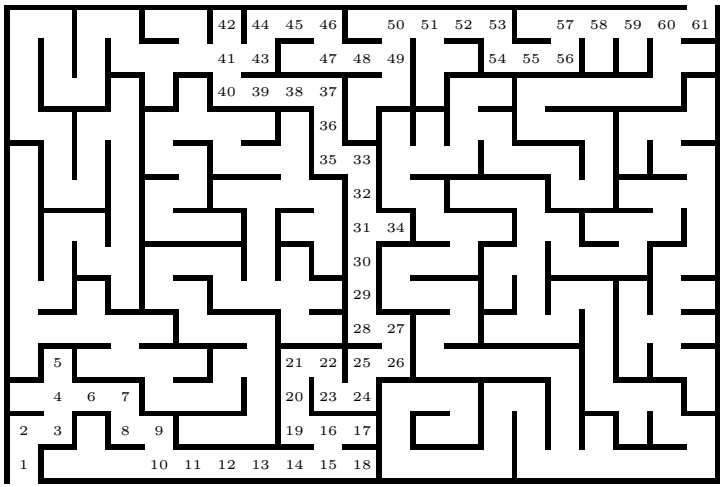
RWTH AACHEN

Beispiel: BFS



RWTH AACHEN

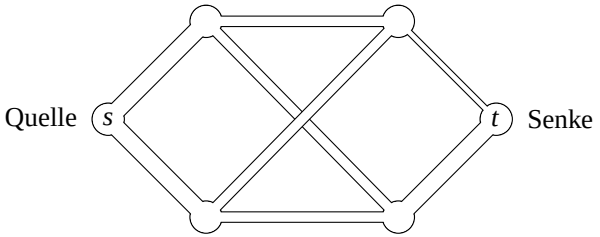
Beispiel: LC



RWTH AACHEN

Netzwerkfluß

Gegeben ist ein System von Wasserrohren:



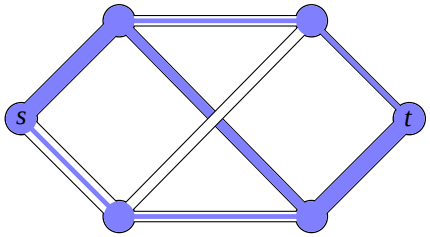
Die Kapazität jedes Rohres ist 3, 5 oder 8 l/s.

Frage: Wieviel Wasser kann von der Quelle zur Senke fließen?

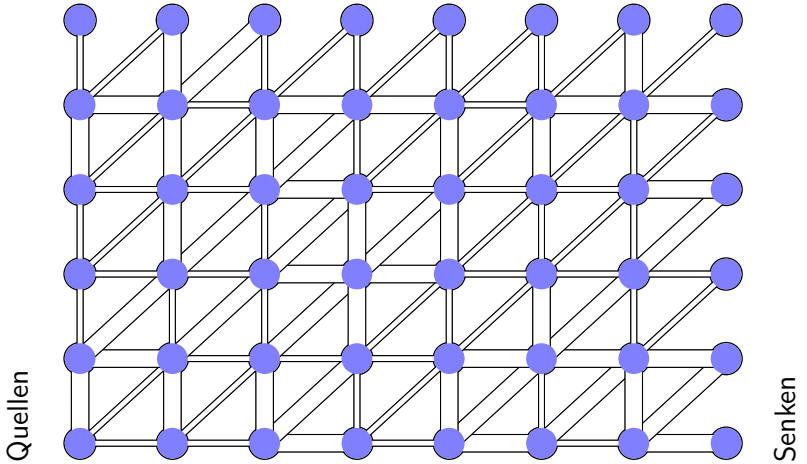
RWTH AACHEN

Netzwerkfluß

Antwort: Maximal 11 l/s sind möglich.

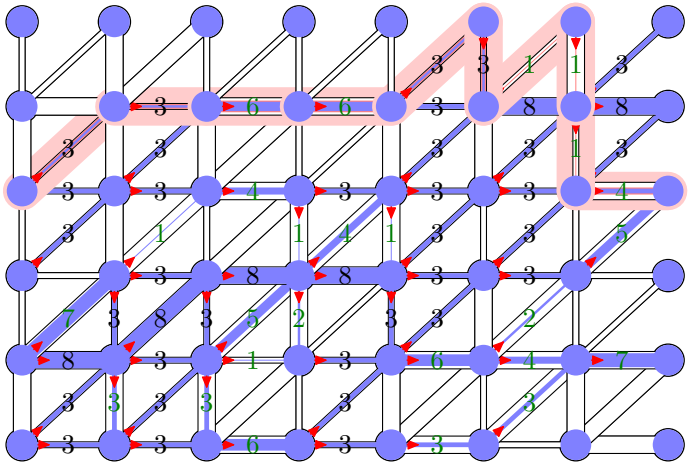


Netzwerkfluß – Aufgabe



Die Kapazitäten sind 3 und 8.

Netzwerkfluß – Lösung



Der maximale Fluß beträgt 30.

s-t-Netzwerke

Definition

Ein s-t-Netzwerk (flow network) ist ein gerichteter Graph $G = (V, E)$, wobei

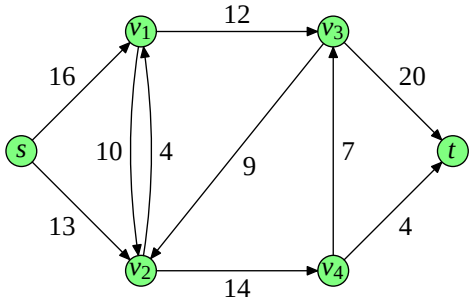
1. jede Kante $(u, v) \in E$ eine Kapazität $c(u, v) \geq 0$ hat,
2. es eine Quelle $s \in V$ und eine Senke $t \in V$ gibt.

Es ist bequem, anzunehmen daß jeder Knoten auf einem Pfad von s nach t liegt.

Falls $(u, v) \notin E$ setzen wir $c(u, v) = 0$.

Es kann Kanten (u, v) und (v, u) mit verschiedener Kapazität geben.

Beispiel eines s-t-Netzwerks



Die Kanten sind mit den Kapazitäten $c(u, v)$ beschriftet.

Flüsse

Definition

Ein Fluß ist eine Funktion $f\colon V \times V \to \mathbf{R}$, die Paare von Knoten auf reelle Zahlen abbildet und diese Bedingungen erfüllt:

- Zulässigkeit: Für $u, v \in V$ gilt $f(u, v) \leq c(u, v)$.
- Symmetrie: Für $u, v \in V$ gilt $f(u, v) = -f(v, u)$.
- Flußerhaltung: Für $u \in V - \{s, t\}$ gilt $\sum_{v \in V} f(u, v) = 0$.

Der Wert $|f|$ eines Flusses ist definiert als $|f| = \sum_{u \in V} f(s, u)$.

Dies ist gerade der Gesamtfluß aus der Quelle heraus.

Maximale Flüsse

Das Problem des maximalen Flusses:

Gegeben: Ein s-t-Netzwerk.

Gesucht: Ein Fluß mit maximalem Wert.

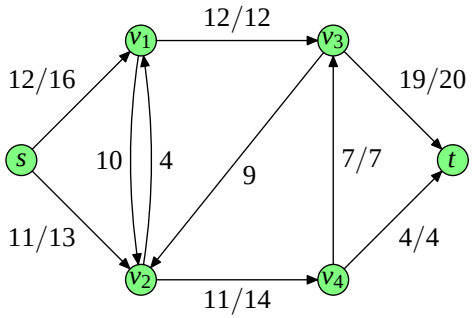
Viele Anwendungen

Beispiel: Wieviele Leitungen müssen zerstört sein, damit zwei Computer nicht mehr miteinander kommunizieren können.

Weiteres Beispiel: ISS-Problem

- Weltraumtouristen machen Angebote
- Sie benötigen spezielle Ausrüstung
- Ausrüstung kann mehrfach benutzt werden
- Wer soll mitgenommen werden?

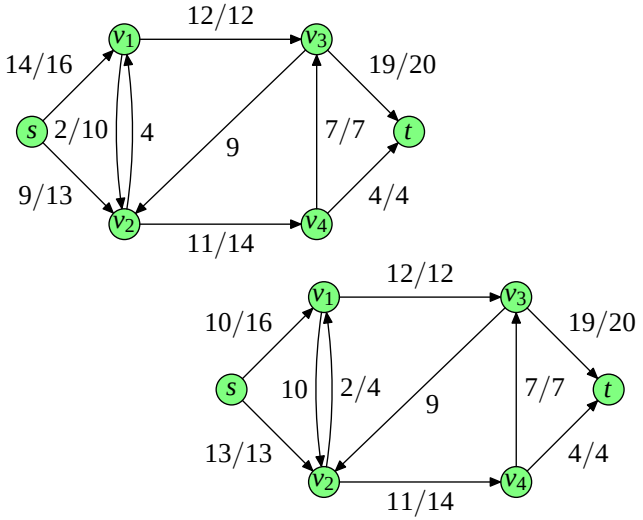
Was ist der maximale Fluß?



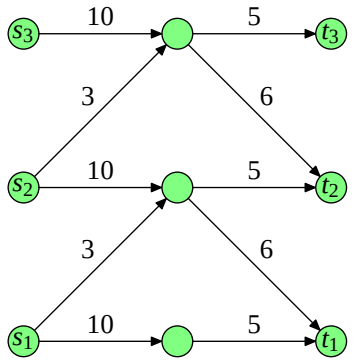
Der maximale Fluß ist 23.

Die Kanten sind mit $c(u, v)$ beschriftet oder mit $f(u, v)/c(u, v)$, falls $f(u, v) > 0$.

Andere optimale Lösungen

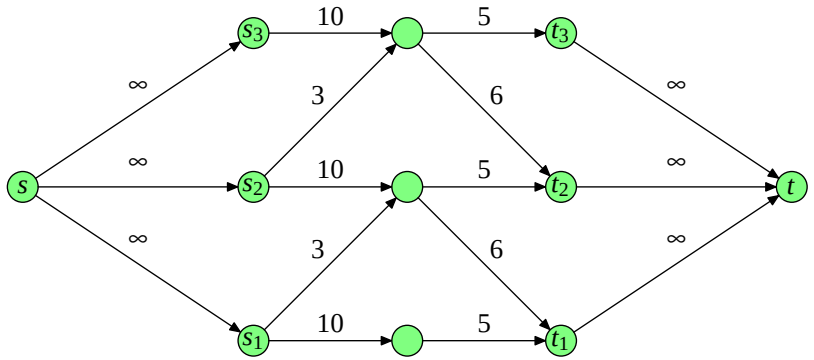


Mehrfache Quellen oder Senken



Mehrfache Quellen oder Senken sind eine Verallgemeinerung des Problem des maximalen Flusses.

Mehrfache Quellen oder Senken



→ Neue „Superquelle“ und „Supersenke“ hinzufügen.

Existenz des maximalen Flusses

Existiert stets der maximale Fluß

$$\max\{\, |f| \mid f \text{ ist ein } s\text{-}t\text{-Fluß in } G \,\}?$$

Ja, denn die Menge aller Flüsse ist abgeschlossen im $\mathbf{R}^m$ und sie ist nicht leer.

Die stetige Funktion, die einen Fluß auf ihren Wert abbildet, hat daher ein Maximum:

$$|f| = \sum_{u \in V} f(s,u) \text{ ist stetig!}$$

(Satz von Weierstrass)

Einige Notationen

Es ist bequem einige Abkürzungen zu verwenden:

- $f(X,Y) = \sum_{x \in X} \sum_{y \in Y} f(x,y)$ für $X, Y \subseteq V$
- $f(x,Y) = \sum_{y \in Y} f(x,y)$ für $Y \subseteq V$
- $f(X,y) = \sum_{x \in X} f(x,y)$ für $X \subseteq V$
- $X - y$ statt $X - \{y\}$

Lemma A

Falls f ein s-t-Fluß für $G = (V,E)$ ist, dann gilt:

1. $f(X,X) = 0$ für $X \subseteq V$
2. $f(X,Y) = -f(Y,X)$ für $X, Y \subseteq V$
3. $f(X \cup Y, Z) = f(X,Z) + f(Y,Z)$ für $X, Y, Z \subseteq V$ mit $X \cap Y = \emptyset$
4. $f(Z, X \cup Y) = f(Z,X) + f(Z,Y)$ für $X, Y, Z \subseteq V$ mit $X \cap Y = \emptyset$

Dieses Lemma ist sehr nützlich, um wichtige Eigenschaften über Flüsse abzuleiten.

Lemma A

Um Lemma A zu beweisen, dürfen wir nur die Eigenschaften eines s-t-Flusses verwenden, also Zulässigkeit, Symmetrie und Flußerhaltung.
Beweis für $f(X,X) = 0$:

$$\begin{aligned} f(X,X) &= \frac{1}{2}\Big(\sum_{x_1 \in X} \sum_{x_2 \in X} f(x_1,x_2) + \sum_{x_1 \in X} \sum_{x_2 \in X} f(x_1,x_2) \Big) \\ &= \frac{1}{2}\Big(\sum_{x_1 \in X} \sum_{x_2 \in X} f(x_1,x_2) + \sum_{x_1 \in X} \sum_{x_2 \in X} f(x_2,x_1) \Big) \\ &= \frac{1}{2} \sum_{x_1 \in X} \sum_{x_2 \in X} \Big(f(x_1,x_2) + f(x_2,x_1) \Big) = 0 \end{aligned}$$

Hier genügt die Symmetrie allein! (Rest als Übungsaufgabe.)

Anwendung des Lemmas

Der Fluß in die Senke sollte intuitiv dem Fluß aus der Quelle entsprechen:

$$f(s, V) = f(V, t)$$

Beweis mit Lemma A:

$$\begin{aligned} f(s,V) &= f(V,V) - f(V-s,V) \\ &= -f(V-s,V) \\ &= f(V,V-s) \\ &= f(V,t) + f(V,V-s-t) \\ &= f(V,t) \text{ wegen Flußerhaltung} \end{aligned}$$

Residualnetzwerke

„Netzwerk minus Fluß = Residualnetzwerk“

Definition
Gegeben ist ein Netzwerk $G = (V, E)$ und ein Fluß f. Das Residualnetzwerk $G_f = (V, E_f)$ zu G und f ist definiert vermöge

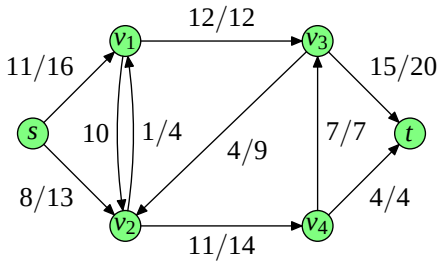
$$E_f = \{\, (u,v) \in V \times V \mid c_f(u,v) > 0 \,\},$$

wobei

$$c_f(u,v) = c(u,v) - f(u,v).$$

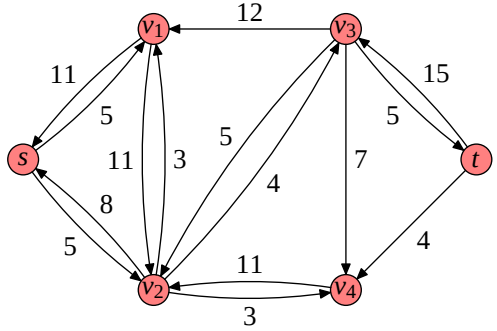
c_f ist die Restkapazität.
Das s-t-Netzwerk G_f hat die Kapazitäten c_f.

Beispiel



s-t-Netzwerk mit Fluß f

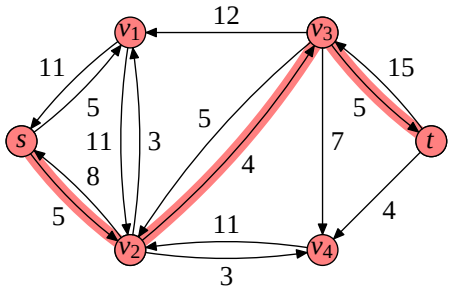
Residualnetzwerk G_f



Augmentierende Pfade

Ein s-t-Pfad p in G_f heißt augmentierender Pfad.

$c_f(p) = \min\{\, c_f(u,v) \mid (u,v) \text{ ist auf } p \,\}$ heißt Restkapazität von p.
Beispiel:



Die Restkapazität dieses Pfades ist 4.

Die Ford–Fulkerson–Methode

Algorithmus

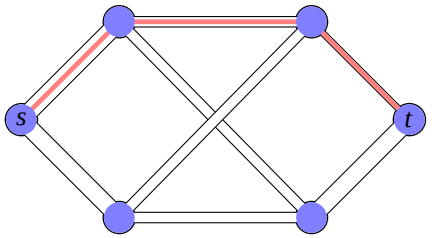
Initialisiere Fluß f zu 0
while es gibt einen augmentierenden Pfad p
  do augmentiere f entlang p
return f

$$f_p(u,v) = \begin{cases} c_f(p) & \text{falls } (u,v) \text{ auf } p \\ -c_f(p) & \text{falls } (v,u) \text{ auf } p \\ 0 & \text{sonst} \end{cases}$$

Augmentiere f entlang p: $f := f + f_p$

f_p ist ein Fluß in G_f

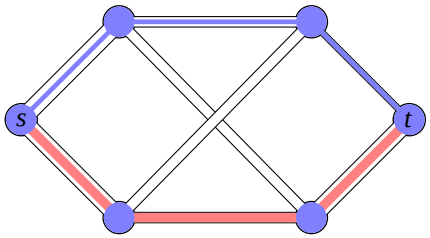
Die Ford–Fulkerson–Methode



Anfangs ist der Fluß 0.

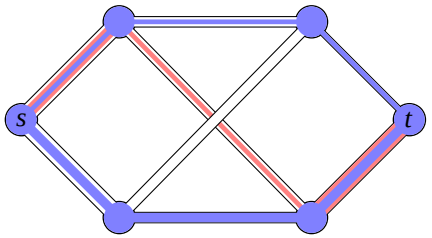
Der augmentierende Pfad ist rot eingezeichnet.

Die Ford–Fulkerson–Methode



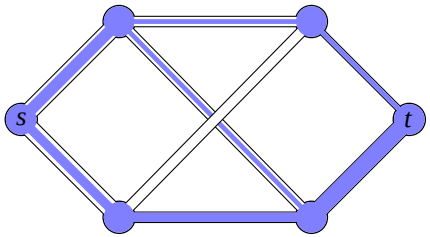
Der augmentierende Pfad hat Kapazität 5.

Die Ford–Fulkerson–Methode



Der augmentierende Pfad hat Kapazität 3.

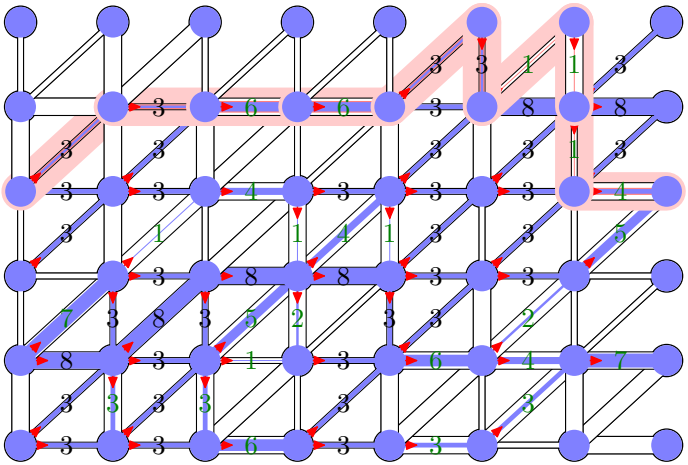
Die Ford–Fulkerson–Methode



Jetzt gibt es keinen augmentierenden Pfad mehr.

Der Fluß ist maximal.

Beispiel



Korrektheit

Lemma B
Sei $G = (V, E)$ ein s-t-Netzwerk und f ein Fluß in G.

Sei f' ein Fluß in G_f.

Dann ist $f + f'$ ein Fluß in G.

Konsequenz:
Die Ford–Fulkerson–Methode berechnet einen Fluß.

Beweis.
Wir müssen zeigen, daß $f + f'$ zulässig, symmetrisch und flußerhaltend ist. □

Beweis (Symmetrie)

$$\begin{aligned}(f + f')(u, v) &= f(u, v) + f'(u, v) \\ &= -f(v, u) - f'(v, u) \\ &= -(f(v, u) + f'(v, u)) \\ &= -(f + f')(v, u)\end{aligned}$$

Beweis (Flußerhaltung)

Sei $u \in V - \{s, t\}$.

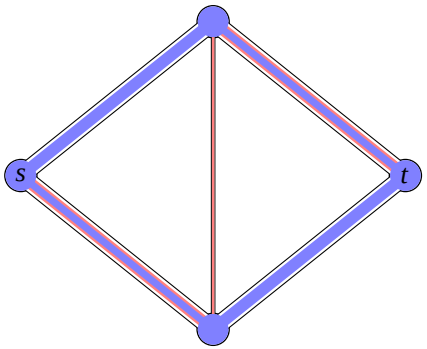
$$\begin{aligned}(f + f')(u, V) &= f(u, V) + f'(u, V) \\ &= 0 + 0 \\ &= 0\end{aligned}$$

Beweis (Zulässigkeit)

$$\begin{aligned}(f + f')(u, v) &= f(u, v) + f'(u, v) \\ &\leq f(u, v) + c_f(u, v) \\ &= f(u, v) + (c(u, v) - f(u, v)) \\ &= c(u, v)\end{aligned}$$

Der Beweis verwendet, daß f' ein Fluß in G_f ist, aber nicht, daß f ein Fluß in G ist.

Laufzeit der Ford–Fulkerson–Methode



Die Laufzeit kann beliebig schlecht sein.

Laufzeit der Ford–Fulkerson–Methode

Ein Flußproblem ist integral, wenn alle Kapazitäten ganzzahlig sind.

Theorem
Die Ford–Fulkerson–Methode benötigt nur $O(f^*)$ Iterationen, um ein integrales Flußproblem zu lösen, falls der Wert eines maximalen Flusses f^* ist.

Beweis.
In jeder Iteration wird der Wert des Flusses um $c_f(p) \geq 1$ erhöht.
Er ist anfangs 0 und am Ende f^*. □

Korollar
Bei rationalen Kapazitäten terminiert die Ford–Fulkerson–Methode.

Schnitte in Netzwerken

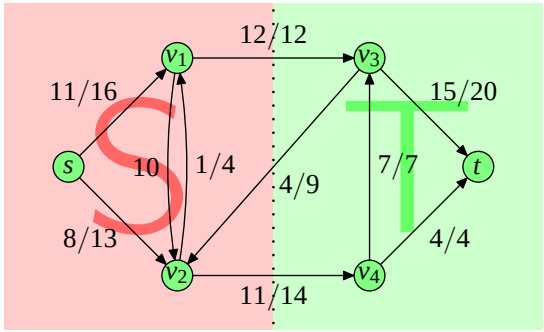
Definition
Ein Schnitt (S, T) in einem s-t-Netzwerk $G = (V, E)$ ist eine Partition $S \cup T = V$, $S \cap T = \emptyset$ mit $s \in S$ und $t \in T$.

Wenn f ein Fluß in G ist, dann ist $f(S, T)$ der Fluß über (S, T).

Die Kapazität von (S, T) ist $c(S, T)$.

Ein minimaler Schnitt ist ein Schnitt mit minimaler Kapazität.

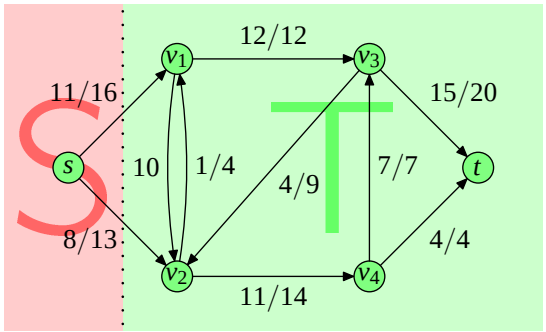
Schnitte in Netzwerken



Der Fluß über (S, T) ist 19.

Die Kapazität von (S, T) ist 26.

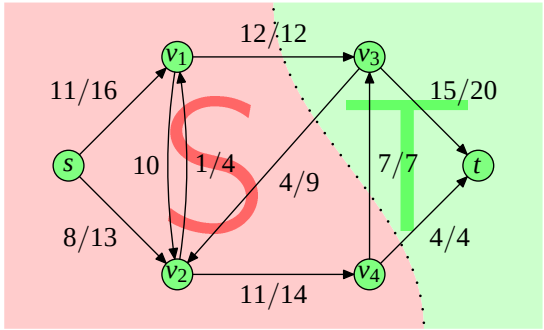
Schnitte in Netzwerken



Der Fluß über (S, T) ist 19.

Die Kapazität von (S, T) ist 29.

Schnitte in Netzwerken



Der Fluß über (S, T) ist 19.

Die Kapazität von (S, T) ist 23.

Fluß über einen Schnitt

Lemma C

Der Fluß über einen Schnitt und der Wert des Flusses sind identisch, d.h. $f(S,T) = |f|$.

Beweis.

$$\begin{aligned} f(S,T) &= f(S,V) - f(S,V-T) = f(S,V) - f(S,S) \\ &= f(S,V) = f(s,V) + f(S-s,V) \\ &= f(s,V) = |f| \end{aligned}$$

□

Spezialfälle:

$|f| = f(s, V-s) = f(V-t, t)$

Max-flow Min-cut Theorem

Theorem

Sei f ein Fluß im s-t-Netzwerk $G = (V,E)$.

Dann sind äquivalent:

1. *f ist ein maximaler Fluß*
2. *In G_f gibt es keinen augmentierenden Pfad*
3. *$|f| = c(S,T)$ für einen Schnitt (S,T)*

Folgerungen

1. Falls die Ford–Fulkerson–Methode terminiert, berechnet sie einen maximalen Fluß.
2. Die Kapazität eines kleinsten Schnittes gleicht dem Wert eines größten Flusses.

Beweis.

1. $\rightarrow$ 2.

Sei f ein maximaler Fluß.

Nehmen wir an, es gebe einen augmentierenden Pfad p.

Dann ist $f + f_p$ ein Fluß in G mit $|f + f_p| > |f|$.

Das ist ein Widerspruch zur Maximalität von f. □

Beweis.

2. $\rightarrow$ 3.

G_f hat keinen s-t-Pfad.

$S := \{ v \in V \mid$ es gibt einen s-v-Pfad in $G_f \}$

$T := V - S$

Dann ist (S,T) ein Schnitt und es gilt $f(u,v) = c(u,v)$ für alle $u \in S$, $v \in T$.

Nach Lemma C gilt dann $f(S,T) = c(S,T) = |f|$. □

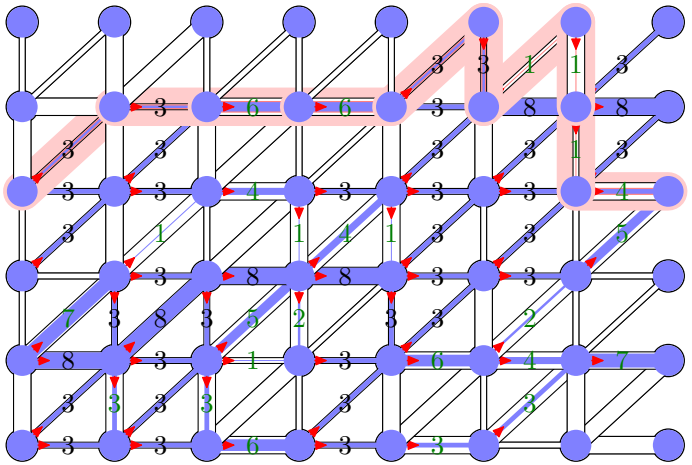
Beweis.
3. $\rightarrow$ 1.

Sei f ein beliebiger Fluß.

$$\begin{aligned} |f| &= f(S,T) \\ &= \sum_{u\in S}\sum_{v\in T} f(u,v) \\ &\leq \sum_{u\in S}\sum_{v\in T} c(u,v) \\ &= c(S,T) \end{aligned}$$

Der Wert jedes Flusses ist also höchstens $c(S,T)$. Erreicht er sogar $c(S,T)$ ist er folglich maximal. □

Wo ist ein minimaler Schnitt?



Wie findet man einen minimalen Schnitt?

1. Einen maximalen Fluß berechnen.
2. Eine Kante (u,v) ist kritisch, wenn $c(u,v) = f(u,v)$.
3. S besteht aus Knoten, die von s aus über unkritische Kanten erreicht werden können.
4. T besteht aus allen anderen Knoten.

Es gibt aber bessere, direkte Methoden!

Ganzzahlige Flüsse

Theorem
Wenn alle Kapazitäten ganzzahlig sind, dann findet die Ford–Fulkerson–Methode einen maximalen Fluß f, so daß alle $f(u,v)$ ganzzahlig sind.

Beweis.
Induktion zeigt, daß die Kapazität eines augmentierenden Pfads ganzzahlig ist und $f(u,v)$ stets ganzzahlig bleiben. □

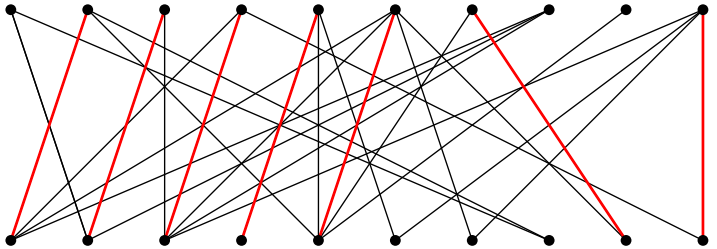
Bipartites Matching

Gegeben: Ein bipartiter, ungerichteter Graph (V_1, V_2, E).

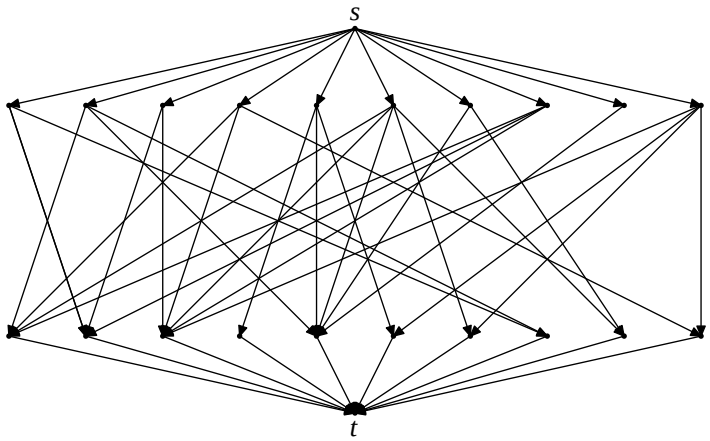
Gesucht:
Ein Matching (Paarung) maximaler Kardinalität.

Ein Matching ist eine Menge paarweise nicht inzidenter Kanten, also $M \subseteq E$ mit $m_1, m_2 \in M,\ m_1 \neq m_2 \Rightarrow m_1 \cap m_2 = \emptyset$.

Beispiel

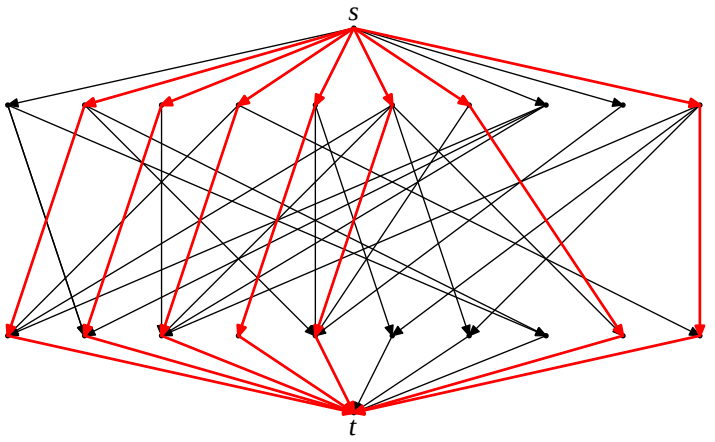


Lösung als Flußproblem

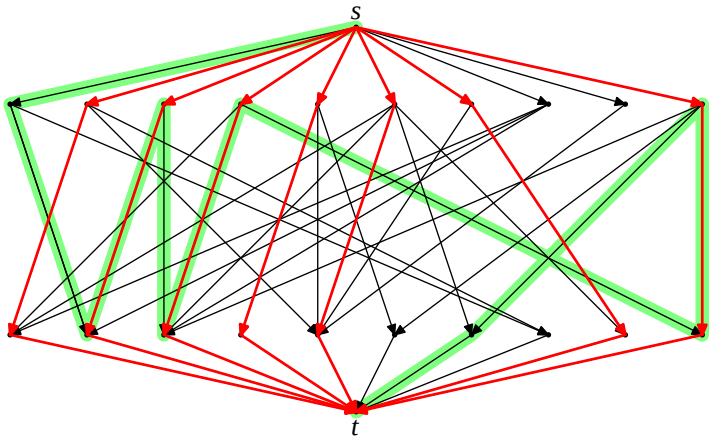


Alle Kapazitäten sind 1.
Maximaler ganzzahliger Fluß entspricht einem Matching maximaler Kardinalität.

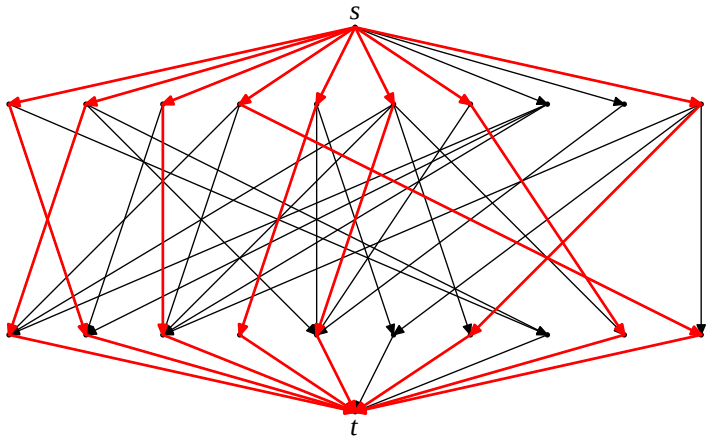
Gibt es einen augmentierenden Pfad?



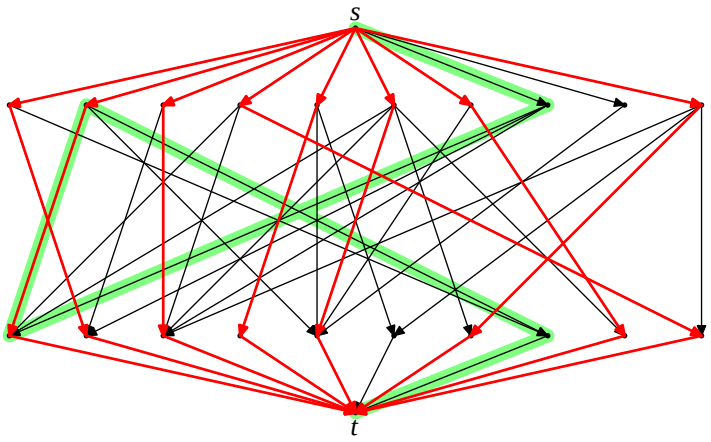
Gibt es einen augmentierenden Pfad? – Ja



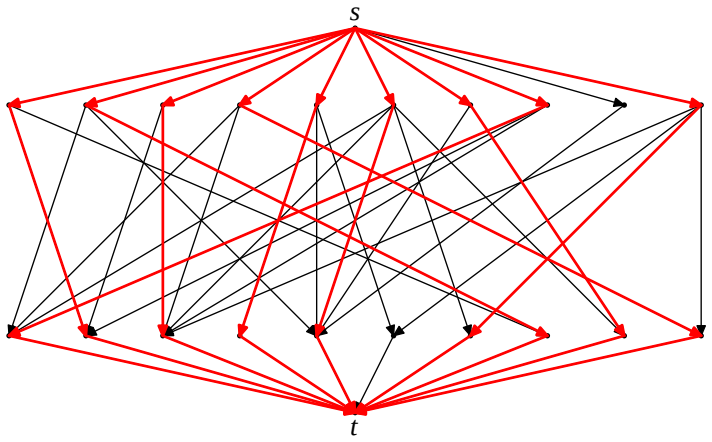
⇒ Neues Matching



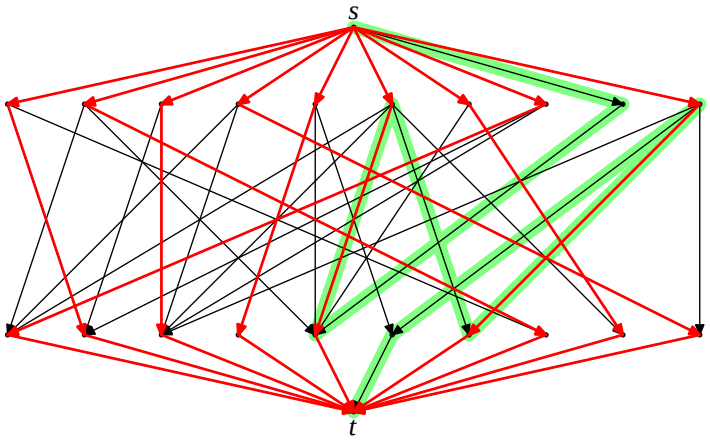
Es gibt wieder einen augmentierenden Pfad



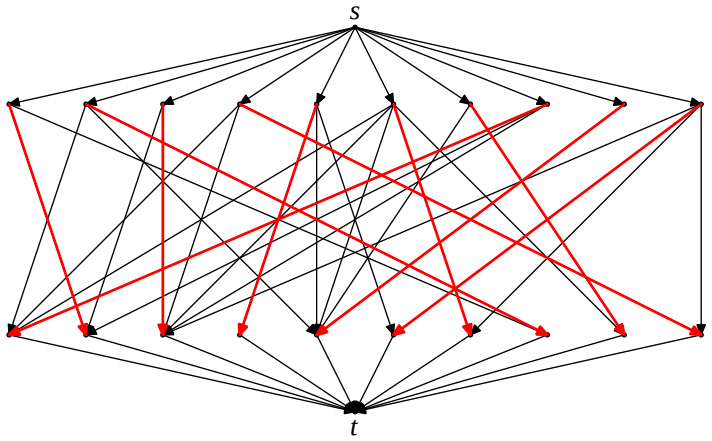
⇒ Neues Matching



Es gibt wieder einen augmentierenden Pfad



Ergebnis: Perfektes Matching



Laufzeit

Finden eines Matchings maximaler Kardinalität dauert nur $O(|E| \cdot \min\{|V_1|, |V_2|\})$ mit der Ford–Fulkerson–Methode.

- Der Fluß ist höchstens $f^* = \min\{|V_1|, |V_2|\}$.
- Finden eines Pfads dauert $O(|E|)$.

Variante der Ford–Fulkerson–Methode

Dieser Algorithmus funktioniert bei ganzzahligen Kapazitäten:

Algorithmus

$K \leftarrow 2^{\lfloor \log_2(\max\{ c(u,v) \mid (u,v) \in E \}) \rfloor}$
$f \leftarrow 0$
while $K \geq 1$ **do**
  while es gibt einen augmentierenden Pfad p mit $c_f(p) \geq K$ **do**
    augmentiere f entlang p
  $K \leftarrow K/2$
return f

Die Laufzeit ist $O(|E|^2 \log K)$.
$\Rightarrow$ vergleiche mit $O(|E| f^*)$.

Variante der Ford–Fulkerson–Methode

Theorem

Die Laufzeit dieser Variante beträgt $O(|E|^2 \log C)$, wobei $C = \max\{ c(u,v) \mid (u,v) \in E \}$.

Beweis.

- Die Restkapazität eines minimalen Schnitts ist stets höchstens $2K|E|$.
- Für jedes K gibt es nur $|E|$ Augmentierungen
- Es gibt $O(\log C)$ verschiedene K

□

Der Edmonds–Karp–Algorithmus

Die Ford–Fulkerson–Methode kann sehr langsam sein, auch wenn das Netzwerk klein ist.

Der Edmonds–Karp–Algorithmus ist polynomiell in der Größe des Netzwerks.

Algorithmus

Initialisiere Fluß f zu 0
while es gibt einen augmentierenden Pfad **do**
  finde einen kürzesten augmentierenden Pfad p
  augmentiere f entlang p
return f

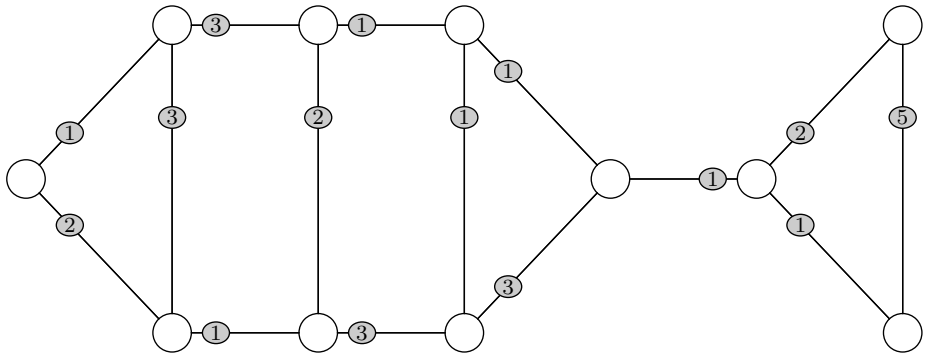
Unterschied: Es wird ein kürzester Pfad gewählt

Der Edmonds–Karp–Algorithmus

Algorithmus

for each edge $(u,v) \in E$ **do**
  $f(u,v) \leftarrow 0$
  $f(v,u) \leftarrow 0$
while there exists a path from s to t in G_f **do**
  $p \leftarrow$ a shortest path from s to t in G_f
  $c_f(p) \leftarrow \min\{ c_f(u,v) \mid (u,v) \text{ is in } p \}$
  for each edge (u,v) in p **do**
    $f(u,v) \leftarrow f(u,v) + c_f(p)$
    $f(v,u) \leftarrow -f(u,v)$
return f

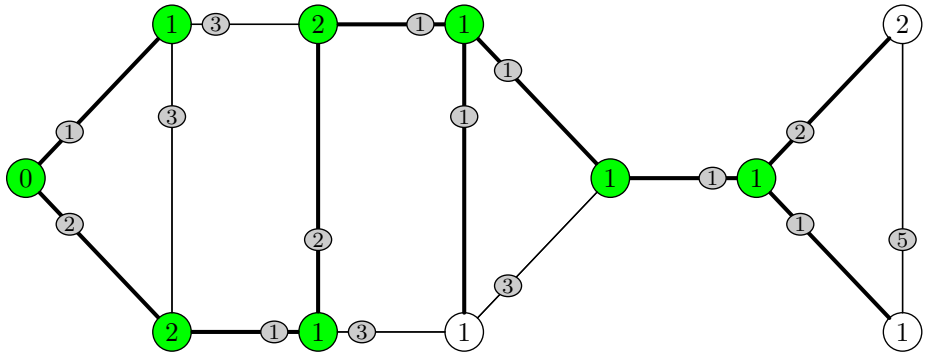
Minimale Spannbäume



Eingabe: Ungerichteter Graph mit Kantengewichten

Ausgabe: Ein Baum, der alle Knoten enthält und minimales Kantengewicht hat

Der Algorithmus von Prim – Beispiel



- Beginne mit leerem Baum (nur Wurzel)
- Hänge wiederholt billigste Kante an ohne einen Kreis zu schließen

RWTH AACHEN

Java

```
public void prim(Map⟨Node, Node⟩ p) {
  SimpleIterator⟨Node⟩ it;
  for(it = nodeiterator(); it.more(); it.step()) it.key().weight = 1e10;
  it = nodeiterator(); it.key().weight = 0.0;
  Heap⟨Node⟩ H = new Heap⟨Node⟩();
  for(it = nodeiterator(); it.more(); it.step()) H.insert(it.key());
  while(!H.isempty()) {
    Node u = H.extract_min();
    Iterator⟨Node, Edge⟩ et;
    for(et = neighbors.find(u).iterator(); et.more(); et.step()) {
      Node v = et.key();
      Edge e = et.data();
      if(H.iselement(v) && e.weight < v.weight) {
        p.insert(v, u);
        v.weight = e.weight; H.decrease_key(v);
      }
    }
  }
}
```

RWTH AACHEN

Der Algorithmus von Prim – Laufzeit

Laufzeit für einen Graphen $G = (V, E)$.

- Anzahl von extract_min: $|V|$
- Anzahl von decrease_key: $|E|$

Laufzeit ist $O((|V| + |E|) \log |V|)$, falls wir einen Heap als Prioritätswarteschlange verwenden.

Laufzeit ist $O(|V| \log |V| + |E|)$, falls wir stattdessen einen Fibonacci-Heap nehmen.

Korrektheit des Algorithmus folgt später.

RWTH AACHEN

Greedy Algorithmen – Münzen wechseln

Es gibt diese acht Euromünzen:

Was ist die minimale Zahl von Münzen um 3.34 Euro zu zahlen?

Antwort:
Wir brauchen sechs Münzen:

Es ist unmöglich, weniger Münzen zu verwenden.

RWTH AACHEN

Münzwechsel – Ein Greedy-Algorithmus

Wir wollen den Betrag n wechseln:

Algorithmus

$r := n$;
while $r > 0$ **do**
  Choose biggest coin c with value$(c) \leq r$;
  $S := S$ *cup* $\{c\}$;
  $r := r - value(c)$
od;
return S

Korrektheit

Lemma A

Sei C eine Münze und v ein Betrag, der mindestens so groß ist wie der Wert von C.
Dann ist es suboptimal, v mit Münzen kleiner als C auszudrücken.

Beweise das Lemma für jede Münze von der kleinsten bis zur größten.
Nimm als Beispiel.
Sei v mindestens 1 EUR. Nehmen wir an, v kann mit genau k und kleineren Münzen für die verbleibenden $v - 50k$ Cents optimal bezahlt werden.
Da diese $v - 50k$ Cents optimal ausgezahlt werden, muß $v - 50k < 50$ und somit auch $100 \leq v < 50(k+1)$ gelten. Es folgt $k \geq 2$, ein Widerspruch zur Optimalität.

Korrektheit

Theorem

Der Greedy-Algorithmus für den Münzwechsel ist optimal.

Beweis.

Nimm an, $C_1, C_2, \ldots, C_n$ ist eine Greedy-Lösung (mit $C_i \geq C_{i+1}$).
Zeige mit Induktion über k, daß eine optimale Lösung mit $C_1, C_2, \ldots, C_k$ beginnt.

Falls dem nicht so wäre, gäbe es eine optimale Lösung $C_1, C_2, \ldots, C_{k-1}, C'_k, \ldots, C'_m$ wobei $C_k > C'_i$ für $i = k, \ldots, m$.
Da $C'_k + \cdots + C'_m \geq C_k$ ist dies ein Widerspruch zu Lemma A. □

Briefmarkensammeln

Funktioniert der Greedy-Algorithmus auch für Briefmarken aus Manchukuo?

Welche Briefmarken für einen 20 fen–Brief?

Der Greedy-Algorithmus führt nicht zu einer optimalen Lösung und findet manchmal gar keine!

Matroide

Der Korrektheitsbeweis für Greedy-Algorithmen kann sehr trickreich sein. Münzwechsel ist hierfür ein Beispiel.
Viele Beweise können allerdings mit Hilfe von der Theorie der Matroide geführt werden.

Definition (Matroid)

Ein Matroid $M = (S, \mathcal{I})$ besteht aus einer Basis S und einer Familie $\mathcal{I} \subseteq 2^S$ von unabhängigen Mengen mit:

1. Falls $A \subseteq B$ und $B \in \mathcal{I}$, dann $A \in \mathcal{I}$ (M ist hereditary).
2. Falls $A, B \in \mathcal{I}$ und $|A| < |B|$ dann gibt es ein $x \in B$ so daß $A \cup \{x\} \in \mathcal{I}$ (M hat die Austauscheigenschaft).

Matroide

Beispiel – Der graphische Matroid

Sei $G = (V, E)$ ein ungerichteter Graph.

Sei $\mathcal{F} = \{ F \subseteq E \mid (V, F) \text{ ist azyklisch} \}$.

Dann ist (E, F) ein Matroid.

Zum Beweis machen wir zunächst eine Beobachtung:
Es sei $G = (V, E)$ ein Wald. Dann verbindet die Kante $e \in E$ zwei Bäume in G gdw. $G = (V, E \cup \{e\})$ kreisfrei ist.

Beweis.

- Vererbungseigenschaft: Wegnehmen von Kanten kann keine Kreise schließen.
- Austauscheigenschaft: Seien $G_A = (V, A)$ und $G_B = (V, B)$ Teilwälder von G und $|A| < |B|$
  1. Beobachtung: Ein Wald mit k Kanten besteht aus $|V| - k$ Bäumen.
  2. G_A hat mehr Bäume als G_B
  3. Es gibt einen Baum T in G_B, der zwei Bäume in G_A verbindet
  4. Es gibt eine Kante in T, die keinen Kreis in G_A schließt

□

Matroide

Wir nennen eine unabhängige Menge A maximal, wenn keine echte Obermenge von A unabhängig ist.

Lemma (Lemma G1)

Alle maximalen unabhängigen Mengen eines Matroids haben die gleiche Größe.

Beweis.

Angenommen, daß A und B maximale unabhängige Mengen sind, aber $|A| < |B|$. Nach der Austauscheigenschaft gibt es ein $x \in B$, so daß $A \cup \{x\}$ unabhängig ist. Das widerspricht der Maximalität von A. □

Gewichtete Matroide

- Ein gewichtetes Matroid ist ein Matroid $M = (S, \mathcal{I})$ mit einer Gewichtsfunktion $w\colon S \to \mathbf{Q}$.
- Wir nennen eine Menge maximalen Gewichts unter allen maximalen unabhängigen Mengen optimal.
- Viele Optimierungsprobleme können durch das Finden einer optimalen Menge in einem Matroid gelöst werden.

Beispiel

Ein minimaler Spannbaum ist eine optimale Menge im graphischen Matroid.

Der Greedy-Algorithmus auf gewichteten Matroiden

Sei $M = (S, \mathcal{I})$ ein Matroid mit Gewichten w.

Dieser Algorithmus findet eine optimale Menge.

Algorithmus

function *Greedy*(S) :
$R :=$ *emptyset*;
sort S *into* $(s[1], ..., s[n])$ *with* $w(s[i]) \geq w(s[i+1])$;
for $i = 1, ..., n$ **do**
  if R *cup* $\{s[i]\}$ *subseteq* I **then** $R := R$ *cup* $\{s[i]\}$ **fi**
od;
return R

Die Laufzeit ist $O(n \log n + nf(n))$, wenn ein Vergleich konstante Zeit braucht, und der Unabhängigkeitstest $O(f(n))$.

Korrektheit

Lemma (G2)

Sei $M = (S, \mathcal{I})$ ein Matroid mit Gewichten w. Wenn $x \in S$ maximales Gewicht unter allen x hat, für die $\{x\}$ unabhängig ist, dann gibt eine optimale Menge, die x enthält.

Beweis.

Sei B eine optimale Menge mit $x \notin B$. Wir haben $w(y) \leq w(x)$ für jedes $y \in B$, weil $\{y\}$ nach der Vererbungseigenschaft unabhängig ist. Beginne mit $A = \{x\}$ und füge Elemente von B zu A hinzu (A bleibt unabhängig, wegen der Austauscheigenschaft) bis $|A| = |B|$. Dann ist $A = B - \{y\} \cup \{x\}$ für ein $y \in B$ und daher gilt $w(A) \geq W(B)$.

□

Kontraktion eines Matroids

Definition

Sei $M = (S, \mathcal{I})$ ein Matroid und $x \in S$.
Dann ist $M' = (S', \mathcal{I}')$ die Kontraktion von M um x, wobei

- $S' = \{\, y \in S \mid \{x, y\} \in \mathcal{I}, x \neq y \,\}$,
- $\mathcal{I}' = \{\, A \subseteq S - \{x\} \mid A \cup \{x\} \in \mathcal{I} \,\}$.

Wir beobachten, daß der Greedy-Algorithmus auf M' arbeitet, nachdem er x gewählt hat.

Lemma (G3)

Sei $M = (S, \mathcal{I})$ ein Matroid mit Gewichten w. Sei x das erste Element, daß durch den Greedy-Algorithmus gewählt wird. Dann ist eine optimale Menge für die Kontraktion M' von M um x zusammen mit x eine optimale Menge für M.

Beweis.

Sei $x \in A$ und $B = A - \{x\}$. Dann ist A maximal unabhängig in M gdw. B maximal unabhängig in M' ist.
Da $w(A) = w(B) + w(x)$ gilt, ist A optimal in M gdw. B optimal in M' ist. □

Korrektheit des Greedy-Algorithmus

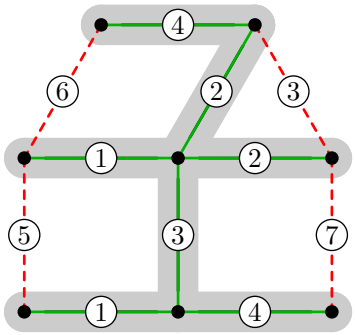
Theorem

Der Greedy-Algorithmus berechnet eine optimale Menge in einem gewichteten Matroid.

Beweis.

Aus $G2$ und $G3$ folgt die Korrektheit mittels Induktion über die Größe der maximalen unabhängigen Menge, die nach $G1$ wohldefiniert ist. □

Kruskal: Minimaler Spannbaum



Kruskals Algorithmus – Implementierung

Algorithmus

function *Kruskal*(G, w) :
$A :=$ *emptyset*;
for *each vertex* v *in* $V[G]$ **do**
  Make_Set(v)
od;
sort the edges of E *into nondecreasing order by weight*;
for *each edge* $\{u, v\}$ *in* E, *nondecreasingly* **do**
  if *Find_Set*$(u) \neq$ *Find_Set*(v) **then**
    $A := A$ *cup* $\{\{u, v\}\}$;
    Union(u, v)
  fi
od;
return A

Korrektheit und Laufzeit der Implementierung

Die Korrektheit folgt als Korollar aus der Korrektheit des allgemeinen Greedy-Algorithmus und der Beobachtung zum graphischen Matroid.
Laufzeit mit $m = |E|$ und $n = |V|$ und $m \geq n - 1$:
n Make-Set-Operationen,
$O(m \log m)$ Zeit für das Sortieren der Kanten, und
$O(m)$ Find-Set- und Union-Operationen.

Mit einer geeigneten Union-Find-Implementierung zusammen $O(m \log m)$.

Union-Find: naiv

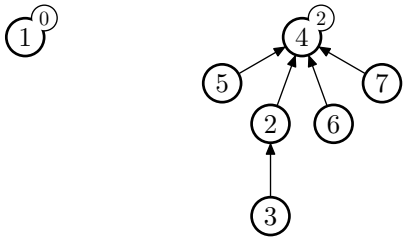
union(a, b): Hänge *find*(a) bei *find*(b) ein

Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)\ldots$

Union-Find: union by rank

union(a, b): Verwende die ranghöhere Wurzel

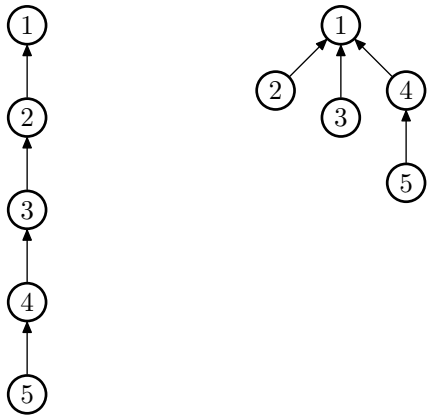
Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)$, *union*$(3, 4)$



Union-Find: Pfadkompression

find(a): Komprimiere durchlaufene Pfade

Beispiel: *find*(4)



Union-Find

Algorithmus

procedure *Make_Set*(x) :
$p[x] := x$;
$rank[x] := 0$

- Wir betrachten jeweils eine Menge $\{0, \ldots, n-1\}$
- Ein Array für die Eltern eines für den Rang
- Ein Baum pro Menge repräsentiert durch die Wurzel
- Wurzel hier kurzgeschlossen

RWTH AACHEN

Union-Find

Algorithmus

function *Find_Set*(x) :
if $x \neq p[x]$ **then** $p[x] := Find - Set(p[x])$ **fi**;
return $p[x]$;

Algorithmus

procedure *Union*(x, y) :
$x := Find\_Set(x)$;
$y := Find\_Set(y)$;
if $rank[x] > rank[y]$ **then** $p[y] := x$
  else $p[x] := y$;
    if $rank[x] = rank[y]$ **then** $rank[y]$++**fi**;
fi;

RWTH AACHEN

Java

```
public class Partition {
  int[ ] s;
  public Partition(int n) {
    s = new int[n];
    for(int i = 0; i < n; i++) s[i] = i;
  }
  public int find(int i) {
    int p = i, t;
    while(s[p] ≠ p) p = s[p];
    while(i ≠ p) {t = s[i]; s[i] = p; i = t; }
    return p;
  }
  public void union(int i, int j) {s[find(i)] = find(j); }
}
```

RWTH AACHEN

Union-Find – Analyse

Zunächst keine Pfadkompression.

Lemma

Falls eine Union-Find-Datenstruktur einen Baum mit m Elementen enthält, dann ist seine Höhe höchstens $\log m + 1$.

Beweis.

Wird ein Element neu hinzugefügt, dann ist die Höhe des Baums $1 = \log(1) + 1$.
Werden zwei Bäume zu einem kombiniert, dann ist die Höhe anschließend unverändert, außer die Höhen beider Bäume waren exakt gleich h.
Falls die Bäume vorher k und m Elemente enthielten, galt $h \leq \log(k) + 1 \leq \log(m) + 1$.
Daher $h + 1 \leq \log(2m) + 1 \leq \log(k + m) + 1$. □

RWTH AACHEN

Union-Find – Analyse

Theorem
In einer anfangs leeren Union-Find–Datenstruktur mit Rangheuristik werden m Operationen in $O(m \log m)$ Zeit ausgeführt.

Beweis.

- Es gibt stets höchstens m Elemente
- Die Höhe aller Bäume ist durch $\log(m) + 1$ beschränkt
- Union und Find benötigt also nur $O(\log m)$ Zeit

□

Rang und Pfadkompression

Mittels amortisierter Analyse (Tarjan 1975): m Operationen in $O(m\alpha(m))$ mit
$\alpha(m)$ funktionale Inverse der Ackermannfunktion

Tarjan 1979, Fredman, Saks 1989: Das ist optimal!

Beweis recht kompliziert...

Algorithmische Geometrie

Probleme der Algorithmischen Geometrie haben üblicherweise diese Eigenschaften:

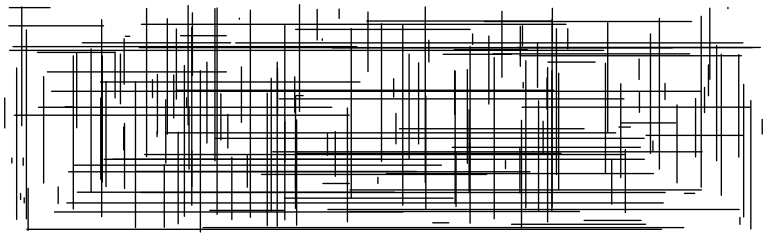
- Die Eingabe besteht aus Punkten, Segmenten, Kreisbögen usw. in der euklidischen Ebene.
- Die Fragestellung ist relativ einfach.
- Sehr große Eingaben müssen bewältigt werden.

Anwendungen beispielsweise im VLSI-Design.

Schnitte von Segmenten

Eingabe: Horizontale und vertikale Segmente

Ausgabe: Paare von Segmenten, die sich schneiden



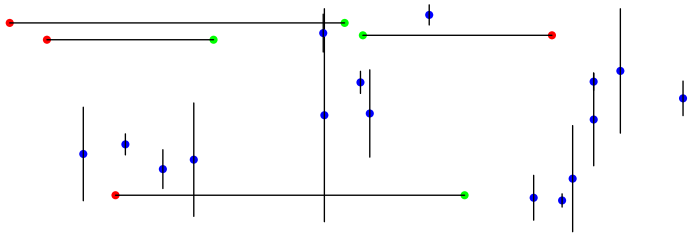
Naiver Algorithmus:

Teste alle Paare, ob sie sich schneiden.

Laufzeit: $\Theta(n^2)$

Variante: Finde heraus, ob es einen Schnitt gibt.

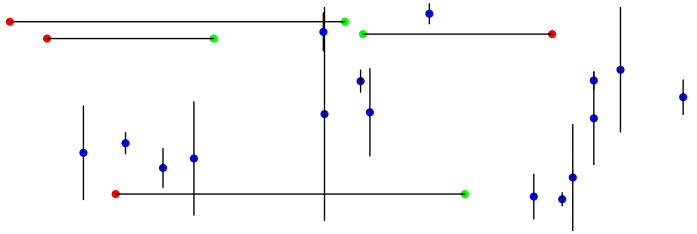
Sweepline-Algorithmen



- Interessante Punkte: Nach x-Koordinate sortieren
- Es gibt ein aktive Menge von Segmenten
- Eine imaginäre vertikale Linie bewegt sich von links nach rechts
- Roter Punkt: Segment in aktive Menge aufnehmen
- Grüner Punkt: Segment aus aktiver Menge entfernen
- Blauer Punkt: Segment mit aktiver Menge vergleichen

Suche nur Schnitte zwischen horizontalem und vertikalem Segment.

Sweepline-Algorithmen



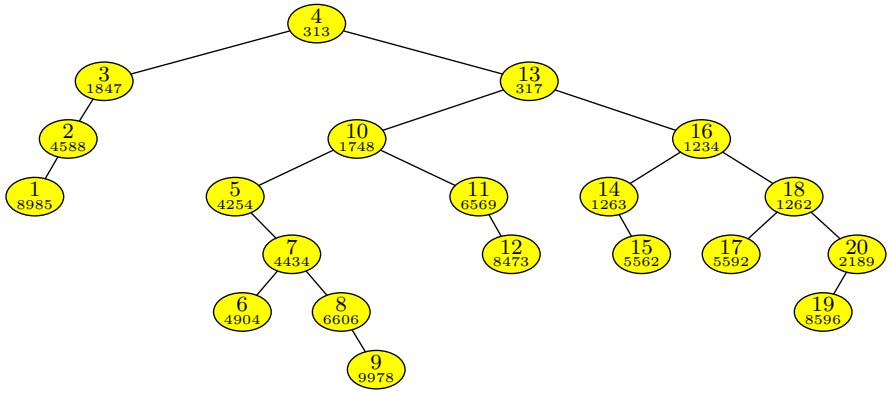
- Wie finden wir bei einem vertikalen Segment die geschnittenen horizontalen Segmente?
- Welche Datenstruktur für die aktive Menge?

Lösung: Speicher y-Koordinaten Y der aktiven Menge in balanziertem Suchbaum.

Gibt es einen Schnitt? $\rightarrow O(\log |Y|)$ Schritte

Alle Schnitte S ausgeben: $O(\log |Y| + |S|)$ Schritte

In aktive Menge einfügen oder löschen: $O(\log |Y|)$ Schritte



Aufgabe:
Finde alle y-Koordinaten zwischen a und b.

Laufzeit: $O(\log |Y| + |S|)$

1. Finde größtes Element, daß kleiner oder gleich a ist.
2. Gehe von dort aus Element aufsteigend durch
3. Beende, wenn b überschritten.

Range-Search

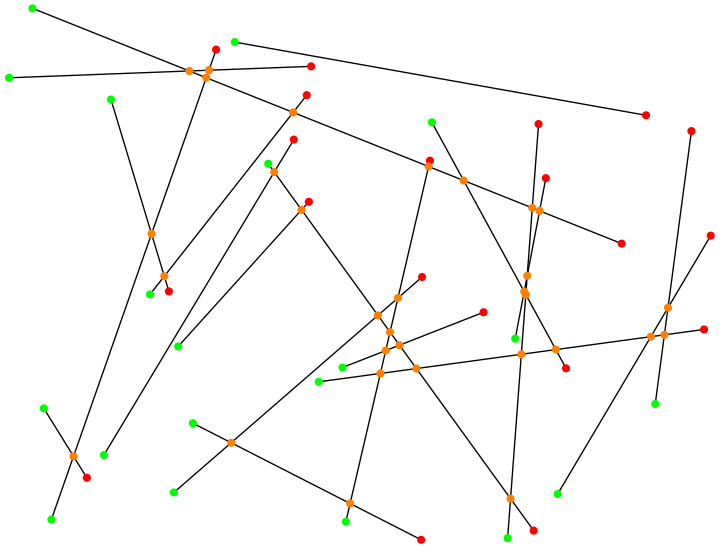
Operationen:

1. Einfügen einer Zahl x
2. Löschen einer Zahl x
3. Ausgabe aller gespeicherter Zahlen in $[a, b]$

Welche Datenstrukturen sind geeignet?

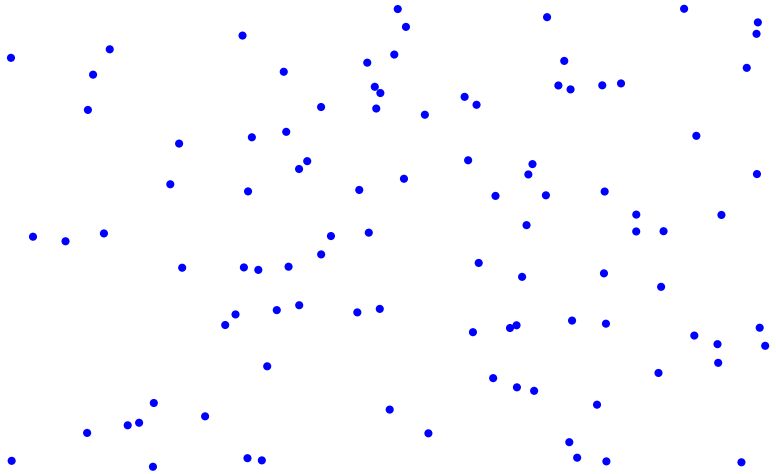
- Arrays?
- Listen?
- AVL-Bäume?
- Splay-Bäume?
- $(2, 3)$-Bäume?
- Treaps?
- Skiplists?
- Hashtabellen?

Das allgemeine Segmentschnittproblem

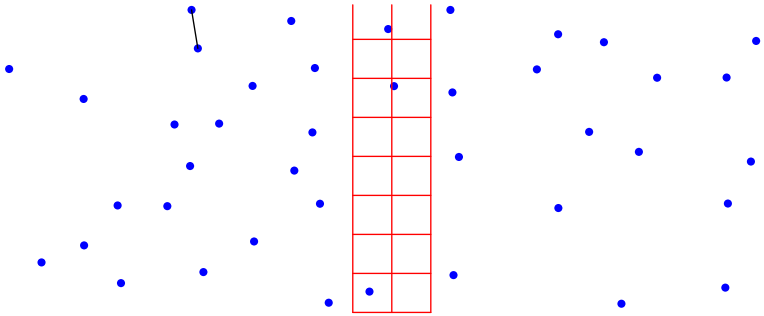


Laufzeit: $O(n \log n + |S|)$

Nächste Nachbarn



Nächste Nachbarn – Divide-and-Conquer



1. Sortiere nach x-Koordinate
2. Teile in linke und rechte Hälfte
3. Finde rekursiv nächste Nachbarn links und rechts (Abstand d)
4. Finde nächste Nachbarn in einem Streifen der Breite $2d$ um die Trennlinie

Nächste Nachbarn – Analyse

Es sei $T(n)$ die Zeit, um n Punkte zu bearbeiten, nachdem sie nach x-Koordinate sortiert wurden.

Nehmen wir an, n sei eine Zweierpotenz.

Wir müssen zwei Probleme der Größe $n/2$ lösen und anschließend die Punkte im Mittelstreifen behandeln.

Es ergibt sich diese Rekursionsgleichung:

$$T(n) = 2T(n/2) + O(n)$$

Dies ist die selbe Rekursionsgleichung wie für Mergesort.

Laufzeit: $O(n \log n)$

Textalgorithmen – Stringmatching

Eingabe: Zwei Strings v und w

Frage: Kommt v als Unterstring in w vor?

Genauer:

$u, v \in \Sigma^*$, wobei Σ ein Alphabet ist.

Gibt es $x, y \in \Sigma^*$, so daß $xvy = w$ gilt?

Freie Monoide

Definition

Ein Alphabet Σ ist eine nichtleere, endliche Menge von Symbolen.

Ein Monoid $(M, e, \circ)$ ist eine Halbgruppe $(M, \circ)$ mit einem neutralem Element e.

Ein Monoid $(M, e, \circ)$ ist von $\Sigma \subseteq M$ frei erzeugt, wenn sich jedes $w \in M$ eindeutig als $w = a_1 \circ \cdots \circ a_n$ darstellen läßt, wobei $a_i \in \Sigma$.

Wir bezeichnen das von einem Alphabet Σ frei erzeugte Monoid mit Σ^*.

Homomorphismen

Definition

Es seien $(M_1, e_1, \circ)$ und $(M_2, e_2, \cdot)$ zwei Monoide.

Eine Abbildung $h\colon M_1 \to M_2$ ist ein (Monoid-)Homomorphismus, falls

1. $h(x \circ y) = h(x) \cdot h(y)$ für alle $x, y \in M_1$
2. $h(e_1) = e_2$

Ein bijektiver Homomorphismus ist ein Isomorphismus.

Freie Monoide

Theorem

Ist ein Monoid $(M, e, \circ)$ von einer Basis $B \subseteq M$ frei erzeugt, dann ist ein Homomorphismus auf M durch seine Bilder auf B bereits eindeutig bestimmt.

Beweis.

Es sei $h : M \to X$ ein Homomorphismus.
Dann ist $h(w) = h(a_1 \circ \ldots \circ a_n) = h(a_1) \cdot \ldots \cdot h(a_n)$. □

Theorem
Ein von einem Alphabet Σ frei erzeugtes Monoid ist bis auf Isomorphismus eindeutig bestimmt.

Beweis.
Es seien $(M_1, e_1, \circ)$ und $(M_2, e_2, \cdot)$ von Σ frei erzeugte Monoide.

Es sei $h\colon M_1 \to M_2$ ein Homomorphismus, den wir vermöge $h(a) = a$ für $a \in \Sigma$ festlegen.

Behauptung: h ist ein Isomorphismus.

Wir müssen beweisen, daß h injektiv und surjektiv ist.

- $u \neq v \Rightarrow u_1 \dots u_n \neq v_1 \dots v_m \Rightarrow h(u_1) \dots h(u_n) \neq h(v_1) \dots h(v_m) \Rightarrow h(u) \neq h(v)$
- $u \neq v \Leftarrow u_1 \dots u_n \neq v_1 \dots v_m \Leftarrow h(u_1) \dots h(u_n) \neq h(v_1) \dots h(v_m) \Leftarrow h(u) \neq h(v)$

□

Stringmatching

Sei Σ ein Alphabet.

1. $u = a_1 \dots a_n \in \Sigma^*$ mit $a_i \in \Sigma$
2. $v = b_1 \dots b_m \in \Sigma^*$ mit $b_i \in \Sigma$

Frage: Kommt v in u vor?

Genauer: Gibt es ein j, so daß $b_i = a_{j+i}$ für alle $1 \leq i \leq m$?

Direkter Algorithmus: Probiere alle j.

a	b	r	a	c	a	d	a	b	r	a

Fenster der Länge m gleitet über u.

Laufzeit $\Theta(n \cdot m)$

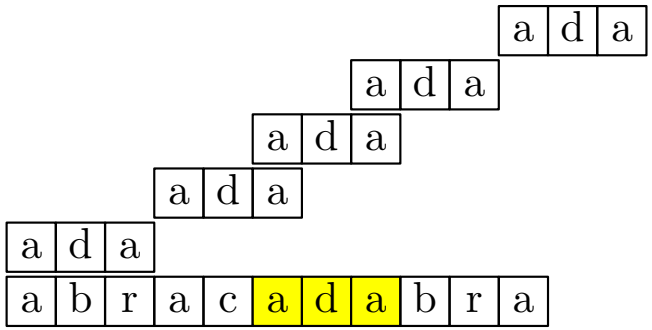
Stringmatching

Wie kann der naive Algorithmus verbessert werden?

a	b	r	a	c	a	d	a	b	r	a

1. Von hinten vergleichen
2. Fenster möglichst weit verschieben
3. v vor der Suche analysieren und Hilfstabellen erstellen

Stringmatching



Verschiebe Fenster um $\delta(x)$, wenn x der erste Mismatch von hinten ist.

Wie berechnen wir $\delta(x)$?

Algorithmus von Boyer und Moore (vereinfacht)

Algorithmus

function *Boyer* − *Moore*1(u, v) *integer* :
$i := |v|; j := |v|;$
repeat
  if $u[i] = v[j]$
  then $i := i - 1; j := j - 1$
  else
    if $|v| - j + 1 > delta(u[i])$ **then** $i := i + |v| - j + 1$
                else $i := i + delta(u[i])$ **fi**;
    $j := |v|$
  fi
until $j < 1$ **or** $i > |u|$;
return $i + 1$

Algorithmus von Boyer und Moore

Diese einfache Variante hat Worst-Case-Laufzeit $O(|u| \cdot |v|)$.

Praktisch ist sie aber sehr bewährt und zeigt oft die Laufzeit $O(|u|/|v|)$.

Durch eine Modifikation läßt sich die Worst-Case-Laufzeit $O(|u| + |v|)$ erzielen.

In der Praxis lohnt sich das allerdings nicht.

Algorithmus von Rabin und Karp

a	b	r	a	c	a	d	a	b	r	a

Gegeben u und v mit $|v| = m$.
Wähle eine geeignete Hashfunktion $\Sigma^* \to \mathbf{N}$.

1. Berechne $h(v)$ und $h(u[i \ldots i + m - 1])$ für $i = 1, \ldots, |u| - m + 1$.
2. Vergleiche v mit $u[i \ldots i + m - 1]$ falls Hashwerte übereinstimmen.

Bei einer guten Hashfunktion unterscheiden sich die Hashwerte fast immer, wenn die Strings verschieden sind.

Wie lange dauert es alle $h(u[i \ldots i + m - 1])$ zu berechnen?

Algorithmus von Rabin und Karp

Wähle eine Hashfunktion, so daß alle

$$h(u[i \ldots i + m - 1])$$

in $O(m + |u|)$ Schritten berechnet werden können.

Zum Beispiel:

$$h(a_1 \ldots a_n) = \left(\sum_{i=1}^{n} q^i a_i \right) \bmod p,$$

wobei p und q große Primzahlen sind.

Es gilt

$$h(a_2 \ldots a_{n+1}) = \left(h(a_1 \ldots a_n)/q - a_1 + q^n a_{n+1} \right) \bmod p.$$

Editdistanz

Gegeben seien zwei Zeichenketten u und v.

Wir dürfen u auf drei Arten ändern:

1. Ersetzen: Ersetze ein Symbol in u durch ein anderes
2. Löschen: Entferne ein Symbol aus u (Länge wird kleiner)
3. Einfügen: Füge ein Symbol an beliebiger Stelle in u ein (Länge wird größer)

Frage: Wieviele solche Operationen sind mindestens nötig, um u zu v zu verwandeln.

Anwendung: Wie ähnlich sind die beiden Zeichenketten (z.B. DNA-Strings)?

Editdistanz

Lösung durch dynamisches Programmieren.

```
GACGTCAGCTTACGTACGATCATTTGACTACG

GACGTCAGCTACAGTAGATCATTTGACTACG
```


Divide-and-Conquer

- Teilen der Eingabe in möglichst gleich grosse Teile
- Rekursives Lösen (Beherrschen) der Teilprobleme
- Zusammenfügen der Teillösungen zur Gesamtlösung

Dynamisches Programmieren

- Optimale Lösungen werden aus optimalen Teillösungen zusammengesetzt.
- Eine Tabelle von Teillösungen wird bottom-up aufgebaut.
- Sonderfall Memoization

0/1-Knapsack

Gegeben sei ein Rucksack mit einer unbeschränkten Kapazität und verschiedene Gegenstände. Jeder Gegenstand hat eine gegebene Größe und einen gegebenen Wert. Es gibt einen Zielwert. Frage: Können wir Gegenstände auswählen, die in den Rucksack hineinpassen und deren zusammengezählte Werte den Zielwert überschreiten?

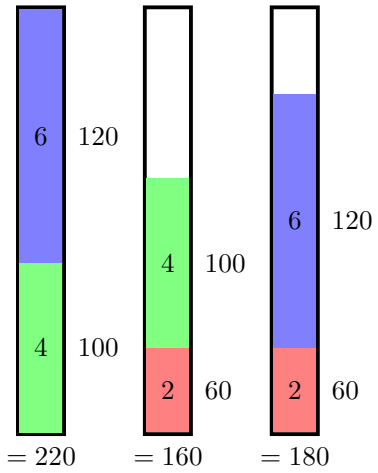
Formaler:

Definition

Eingabe: Positive Integer $(c_1, \ldots, c_n)$, $(v_1, \ldots, v_n)$, und C
Ausgabe: Der Maximalwert v, so daß es eine Menge $I \subseteq \{1, \ldots, n\}$ gibt mit

- $\sum_{i \in I} c_i \leq C$
- $\sum_{i \in I} v_i = v$

Knapsack



0/1-Knapsack

0/1-Knapsack kann durch *brute force* gelöst werden, indem alle Untermengen $I \subseteq \{1, \ldots, n\}$ aufgezählt werden und beide Bedingungen überprüft werden.

Es gibt allerdings 2^n solche Untermengen. Zu langsam!

Können wir dieses Problem durch dynamisches Programmieren lösen?

Ja, indem alle Unterprobleme mit unterer Kapazität zuerst gelöst werden.

0/1-Knapsack

Betrachte den folgenden Algorithmus:

Algorithmus

procedure *Knapsack* :
for $i = 1, ..., n$ **do**
  for $k = 0, ..., C$ **do**
    $best[0, k] := 0$;
    $best[i, k] := best[i-1, k]$;
    if $k \geq c\_i$ **and** $best[i, k] < v\_i + best[i-1, k-c\_i]$
    then $best[i, k] := v\_i + best[i-1, k-c\_i]$ **fi**
  od
od

$best[i, k]$ ist der höchste Wert, den man mit Gegenständen aus der Menge $\{1, \ldots, i\}$ erhalten kann, die zusammen Kapazität k haben.

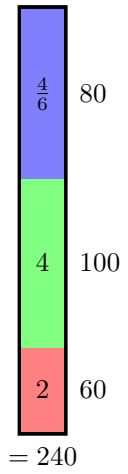
Laufzeit

Die Laufzeit ist $O(Cn)$.

Wir nennen einen solchen Algorythmus pseudo-polynomiell.

Die Laufzeit ist nicht polynomiell in der Eingabelänge, aber polynomiell in der Eingabelänge und der Größe der Zahlen in der Eingabe.

Falls C klein ist, dann ist dieser Ansatz viel schneller als alle Untermengen durchzuprobieren.



Greedy-Algorithmen

- Wie bei Dynamic Programming beinhalten optimale Lösungen optimale Teillösungen
- Anders als bei Dynamic Programming gilt die greedy choice property: eine lokal optimale Lösung ist stets Teil einer global optimalen Lösung
- Korrektheitsbeweise über Theorie der Matroide, Greedoide, Matroideinbettungen, ...

Flußalgorithmen

Modelliere Optimierungsproblem als Flußproblem.

Varianten:

- Maximaler Fluß
- Fluß mit maximalen Gewinn
- Matchings maximaler Kardinalität
- Matchings maximalem Gewichtes

Lineares Programmieren

- Viele Probleme können als Maximierung einer linearen Funktion mit linearen Ungleichungen als Nebenbedingungen ausgedrückt werden.
- Mehr dazu in der Vorlesung Effiziente Algorithmen

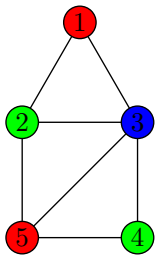
Randomisierte Algorithmen

- Beispiele in der Vorlesung nutzen Zufall, um den *worst-case* in Form eines *adversary* mit hoher Wahrscheinlichkeit zu vermeiden, sowie um den Algorithmus und/oder die Analyse zu vereinfachen
- Eine Vielzahl von weiteren Techniken zum Entwurf in der Vorlesung Randomisierte Algorithmen

Backtracking

- Tiefensuche auf dem Raum der möglichen Lösungen
- ein natürlicher Weg, schwere Entscheidungsprobleme zu lösen
- Ausdruck “backtrack” von D.H. Lehmer in den 50er Jahren
- Verfeinerung der *brute force*-Suche
- In der Praxis gut bei Problemen wie 3-Färbbarkeit oder 0/1-Knapsack

Backtracking



Algorithmus

function *backtrack*$(v1, ..., vi)$:
if $(v1, ..., vi)$ *is a solution* **then return** $(v1, ..., vi)$ **fi**;
for *each* v **do**
  if $(v1, ..., vi, v)$ *is acceptable vector* **then**
    $sol := backtrack(v1, ..., vi, v)$;
    if $sol \neq ()$ **then return** sol **fi**
  fi
od;
return ()

Branch-and-Bound

- 1960 vorgeschlagen von A.H. Land und A.G. Doig für Linear Programming, allgemein nützlich für Optimierungsprobleme
- branching auf einem Suchbaum wie bei Backtracking
- bounding und pruning führt zum Auslassen von uninteressanten Zweigen
- Geeignet für Spiele wie Schach oder schwere Optimierungsprobleme

Beispiel: 0/1-Knapsack

- Verzweige, je nachdem, ob ein Gegenstand mitgenommen wird oder nicht, in einer festen Reihenfolge
- Eine Teillösung $I \subseteq \{1, \ldots, k\}$ ist zulässig, solange $\sum_{i \in I} c_i \leq C$
- Wenn I nicht zulässig ist, kann auch keine Lösung, die I enthält, zulässig sein
- Außerdem ist $\sum_{i \in I} v_i + \sum_{i \in \{k+1, \ldots, n\}} v_i$ eine obere Schranke für den Profit jeder Lösung, die I erweitert, wenn wir in I alle Gegenstände bis zum k. berücksichtigt haben
- Also können wir alle Zweige abschneiden, die entweder nicht zulässig sind oder deren obere Schranke unter dem bis jetzt optimalen Profit liegt.

Das Problem des Handlungsreisenden

Wir untersuchen das folgende Problem:

Definition (TSP)

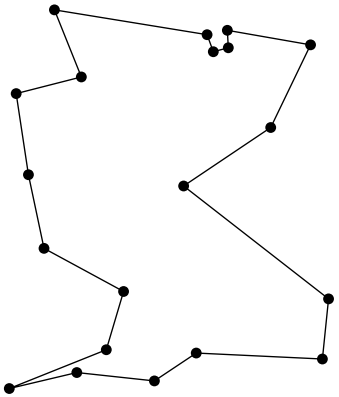
Eingabe: Ein Graph $G = (V, E)$ mit Kantengewichten $length : E \to \mathbf{Q}$
Ausgabe: Eine geschlossene Rundreise über alle Knoten mit minimaler Gesamtlänge.

Wir denken zum Beispiel an n Städte, die alle besucht werden müssen.

Was ist die minimale Länge einer Tour, die alle Städte besucht und im gleichen Ort beginnt und endet?

Das Problem des Handlungsreisenden

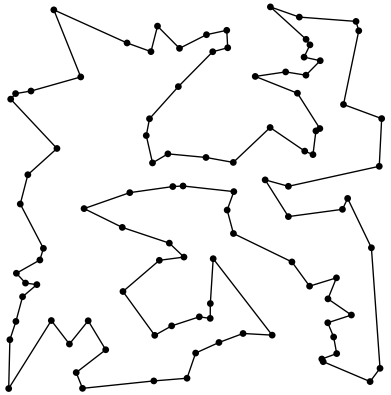
Ein kleines Beispiel:



RWTH AACHEN

Das Problem des Handlungsreisenden

Ein mittelgroßes Beispiel:



RWTH AACHEN

Der naive Ansatz

Der einfachste Ansatz dieses Problem zu lösen besteht darin, alle Möglichkeiten durchzuprobieren.

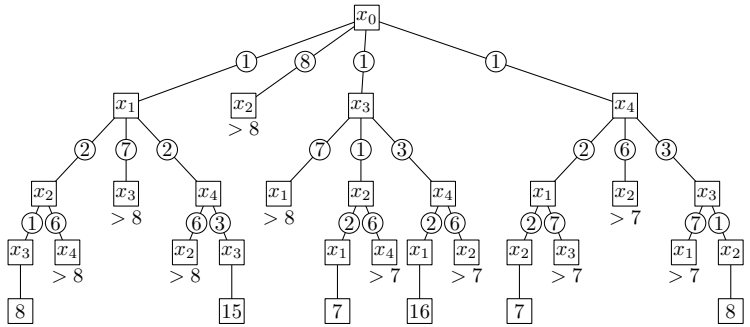
Wieviele Möglichkeiten gibt es?

Es gibt $(n-1)!$ mögliche Rundreisen, da es $n!$ Permutationen gibt. Wir könnten alle Rundreisen durchprobieren, ihre Kosten berechenen und die billigste auswählen.

Es eine schnellere Lösung mit Hilfe von dynamischer Programmierung. Wir versuchen es mit Branch-and-Bound.

RWTH AACHEN

Branch and Bound - Beispiel TSP

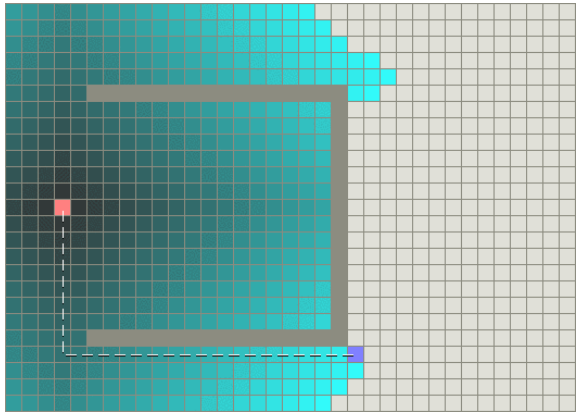


RWTH AACHEN

A*-Suche

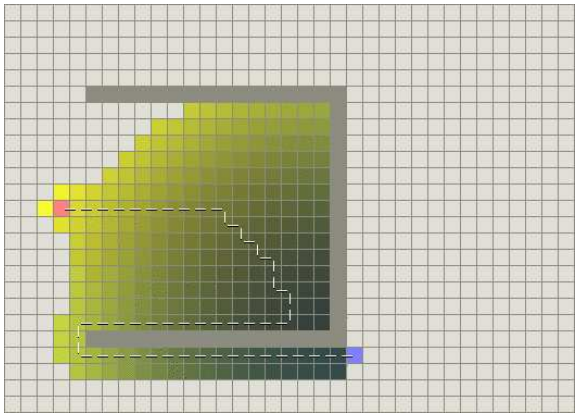
- 1968 von Peter Hart, Nils Nilsson, and Bertram Raphael als A-search (A*-search mit optimaler Heuristik)
- Graphsuchverfahren mit Anwendungen in KI (Planung) und Suche in großen Suchräumen
- Generalisierung von Dijkstras Algorithmus und best-first search
- Knotenexpansion in Reihenfolge von $f(x) = g(x) + h(x)$ mit tatsächlichen Kosten g und Heuristik h

Dijkstra



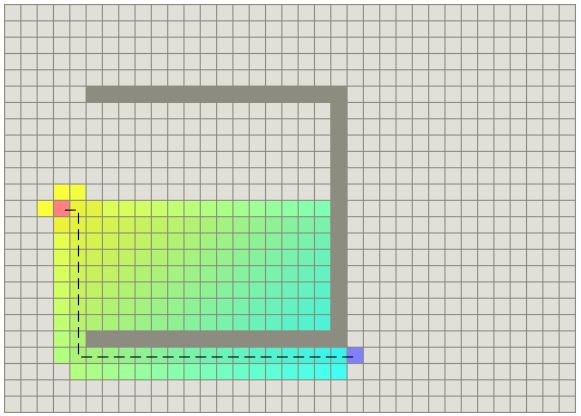
Bilder von Amit Patel/Stanford

Best-First-Search



Bilder von Amit Patel/Stanford

A*



Bilder von Amit Patel/Stanford

Algorithmus

```
procedure a − star(s) :
open := new priority queue
open.enqueue s;
closed := emptyset
while (true) do current := open.extract_min;
  if current = goal then break fi; closed.insert (current);
  for all neighbors of current do
    cost = g(current) + weight(current, neighbor);
    if cost < g(neighbor) then
      open.remove(neighbor); closed.remove(neighbor) fi;
    if!open.iselement(neighbor) and!closed.iselement(neighbor)
      then g(neighbor) := cost; neighbor.parent := current;
      open.enqueue(neighbor) with rank g(neighbor) +
h(neighbor);
    fi
  od
od
reconstruct reverse path from goal to start
  // by following parent pointers
```


Eigenschaften der A*-Suche

Theorem

Der A-Algorithmus findet einen optimalen Pfad, wenn h optimistisch ist.*

Beweis.

Wenn der Algorithmus terminiert, ist die Lösung billiger als die optimistischen Schätzungen für alle offenen Knoten. □

Eigenschaften der A*-Suche

Theorem

Für jede Heuristik h betrachtet die A-Suche die optimale Anzahl von Knoten unter allen zulässigen Algorithmen.*

Beweis.

Nimm an, Algorithmus B mit Heuristik h betrachtet einen Knoten x nicht, der in A* einmal offen war. Dann beträgt $h(x)$ höchstens Kosten des Lösungspfades von B und kann B nicht ausschließen, daß es einen Pfad über x gibt, der billiger ist. Also ist B nicht zulässig. □

A* - Kosten

- Laufzeit: Im allgemeinen Anzahl der betrachteten Knoten exponentiell in der Länge des optimalen Pfades
- stark abhängig von der Güte der Heuristik: polynoniell wenn $|h(x) - h^*(x)| \in O(\log h^*(x))$
- Problem: Speicherplatz ebenfalls exponentiell
- Varianten/Verbesserungen: IDA*,MA*,SMA*,AO*, Lifelong Learning A*, ...

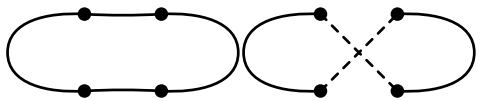
Heuristiken

- Hier: Verfahren, die in der Praxis oft funktionieren, aber für die man keine worst-case-Schranken zeigen kann
- Beispiele: Lokale Suche, Simulated Annealing, Genetische Algorithmen

Lokale Suche

- Heuristik für Optimierungsprobleme
- Definiere Nachbarschaft im Suchraum
- Gehe jeweils zum lokal optimalen Nachbarn
- Problem: lokale Minima

Lokale Suche für TSP



ENDE