

AuP 1

Algorithmen -

ausführbare Handlungsaufforderung aus
endlich vielen Schritten zur ein-
deutigen Umwandlung von Ein zu Ausgabedaten

Charakteristika

Finitheit - endliche Beschreibung

Terminierung

Effektivität - Schritte auf Maschine ausführbar

berechenbar

gleicher Input

Determinismus

$x_1 = x_2 \Rightarrow y_1 = y_2$

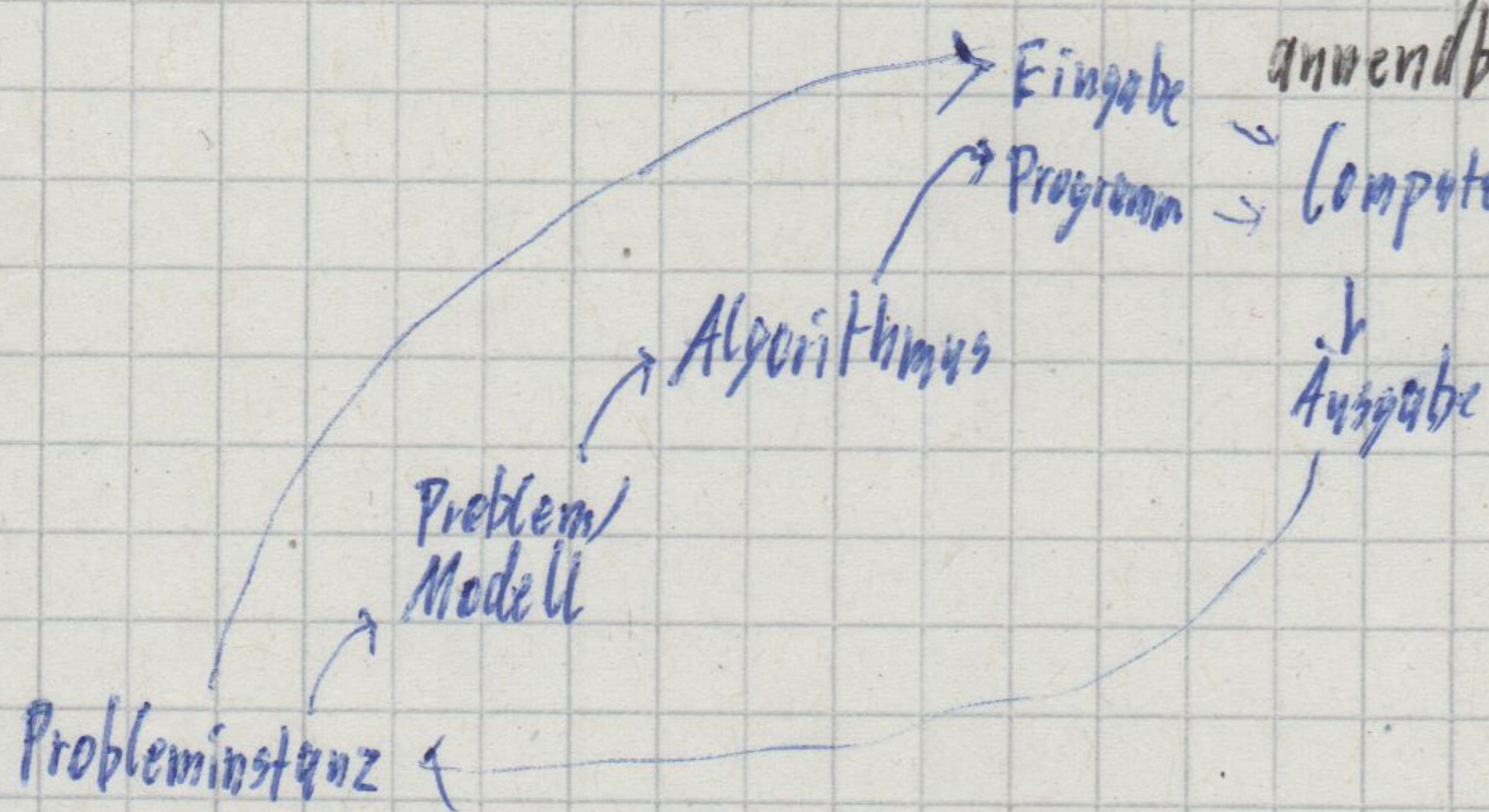
Determiniertheit

gleiche Schritte

gleicher Output

Allgemeinheit
Korrektheit

bestimmt



Ziele

Analysetechniken
korrektheit, Terminierung,
wichtige Algorithmen
geeignete Modellierungen
Aufwand
Entwurfsparadigmen

Übungen: Umsetzen

Datenstrukturen

Methode Daten für den Zugriff
und die Modifikation zu organisieren

Daten

Strukturbestandteile

Ziele

Analyse Aufwand
einfache
Pro/Kontrolle
komplexe

ADTs Abstrakte Datenstrukturen

beides
auch
Datenstruktur

Wiss z.B. Stack mit Operationen
isEmpty, pop, push

Datenstrukturen

Wie? z.B. Stack Operationen
als Array oder verkettete Liste

Problem

Algorithmus $\xrightarrow{\text{verwendet}}$ Datenstruktur

kann auch als Problem aufgefasst werden

Sortierproblem

Objekt

Schlüsselwert

Satellitendaten

gegeben: Sequenz Objekte,
totale Ordnung auf M , Menge möglicher Schlüsselwerte
 $\vdash \forall x, y \in M : x \leq y \vee y \leq x$ „alles vergleichbar“
 ! trotzdem Objekte mit gleichem Schlüssel erlaubt

Insertion Sort

durchforsern und Objekt nach links richtig einsetzen

For $i = 1$ To $A.length - 1$ DO

key = $A[i]$

$j = i - 1$

WHILE $j \geq 0$ AND $A[j] > key$ DO

$A[j + 1] = A[j]$

$j = j - 1$

$A[j + 1] = key$

$\Theta(n^2)$

$\Theta(1)$	constant	Einzeloperation
$\Theta(\log n)$	logarithmisch	binäre Suche
$\Theta(n)$	linear	sequentielle Suche
$\Theta(n \log n)$	quasilinear	Sortieren Array
$\Theta(n^2)$	quadratisch	Matrix Multiplikation
$\Theta(n^3)$	kubisch	Matrix Multiplikation
$\Theta(n^k)$	polynomisch	Nichtlineare Gleichungen
$\Theta(2^n)$	Exponentiell	Rekurrenz Gleichungen
$\Theta(n!)$	Faktoriell	Permutationen

$$\log_a(b) = \frac{\ln(b)}{\ln(a)}$$

$$\ln(ab) = \ln(a) + \ln(b)$$

$$\ln(a^b) = b \ln(a)$$

$$\log_{\frac{a}{b}}(b) = -\log_a(b)$$

Terminierung: Jede Ausführung WHILE-Schleife
erniedrigt j , bricht bei $j < 0$ ab \Rightarrow terminiert

Korrektheit: Invariante Bei jedem Eintritt für Zählerwert i und
beiletzter Ausführung

$A[0] \dots A[i-1]$ sortierte ursprüngl. Werte

$A[i] \dots A[n]$ unverändert

Induktionsbasis: Bei $i=0$ $A[0]$ sortiert, Rest gleich

Schritt: Vor $(i-1)$ -ten Ausführung gilt $A[0 \dots i-2]$ sortiert

$A[i-1]$ wird eingesetzt und größeres nach

rechts verschoben \Rightarrow Invariante gilt auch für i

\Rightarrow Bei Austritt mit $i=n$ ist $A[0 \dots n-1]$ sortiert

Laufzeitanalyse: Schritte in Abhängigkeit Eingabekomplexität

meist Worst-Case-Laufzeit: $T(n) = \max \{ \text{Anzahl Schritte für } x \text{ der Komplexität } n \}$

Analyse: Wie oft wird eine Zeile max. ausgeführt

Kosten für einen Schritt \Rightarrow stark von Berechnungsmodell \Rightarrow Annahme elementare Operationen 1 Schritt

Asymptotische Vereinfachung (Landau-Symbole)

Funktionen $f, g: \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$
Eingabekomplexität Laufzeit

Für untere Schranke nach schlechtesten Eingabe suchen

beschränkt
asymptotisch

oben und unten

oben

unten

Schranke oben
Schranke unten

$$\begin{aligned}\Theta(g) &= \{ f : \exists c_1, c_2 \in \mathbb{R}_{>0}, \forall n \in \mathbb{N}, \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \} \quad \text{wobei } f = \Theta(g) \text{ statt } f \in \Theta(g) \\ O(g) &= \{ f : \exists c \in \mathbb{R}_{>0}, \forall n \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) \leq c g(n) \} \\ \Omega(g) &= \{ f : \exists c \in \mathbb{R}_{>0}, \forall n \in \mathbb{N}, \forall n \geq n_0, c g(n) \leq f(n) \} \\ o(g) &= \{ f : \forall c \in \mathbb{R}_{>0}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) < c g(n) \} \\ \omega(g) &= \{ f : \forall c \in \mathbb{R}_{>0}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq c g(n) < f(n) \}\end{aligned}$$

Regeln: $g \in \mathbb{R}_{>0}$ $a \in \mathbb{F}[1]$

$f_1, f_2 \in \mathbb{F}$

$f \in \mathbb{F}(g) \Rightarrow a f \in \mathbb{F}(a g)$

$$\begin{aligned}f_1 f_2 \in \mathbb{F}(g_1), f_2 f \in \mathbb{F}(g_2) &\Rightarrow f_1 + f_2 \in \mathbb{F}(\max\{g_1, g_2\}) \\ &\Rightarrow f_1 \cdot f_2 \in \mathbb{F}(f_1 \cdot g_2)\end{aligned}$$

$$O(n) \subset \mathbb{F}(n^2)$$

strikkt

Merge Sort ~ Idee Divide & Conquer (& Combine)

Bubble Sort

```

mergeSort(A, l, r)
IF r >= l return
m = floor((l+r)/2)
mergeSort(A, l, m)
mergeSort(A, m+1, r)
merge(A, l, m, r) // zwei Pointer durchläuft beide Teilarrays 1mal
                    Kopieren in Kopie[ache]
    
```

```

merge(A, l, m, r)
pl = l; pr = m+1;
FOR i = 0 TO r-1 DO
    IF pr > r OR (pl <= m AND A[pl] <= A[pr]) THEN
        B[i] = A[pl];
        pl++;
    ELSE
        B[i] = A[pr];
        pr++;
FOR i = 0 TO r-1 DO A[i+1] = B[i];
    
```

Laufzeitanalyse: $T(n) \leq 2 \cdot T(n/2) + c + dn$

$$T(n) \in O(n \log n)$$

IF $\lceil \log_2 n \rceil$ Floor Merge

Mastermethode

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \quad a \geq 1, b > 1, f(n) \text{ asymptotisch positiv}$$

$$\epsilon > 0 \quad f(n) \in O(n^{\log_b(a)-\epsilon}) \Rightarrow T(n) \in \Theta(n^{\log_b a})$$

$$f(n) \in \Theta(n^{\log_b a}) \Rightarrow T(n) \in \Theta(n^{\log_b a} \cdot \log_2 n)$$

$$f(n) \in \Omega(n^{\log_b(a)+\epsilon}) \Rightarrow \exists n_0 \forall n \geq n_0 \quad a f\left(\frac{n}{b}\right) \leq c f(n) \quad c < 1, n \geq n_0 \Rightarrow T(n) \in \Theta(f(n))$$

Quicksort - Aufteilen nach Wert, zusammenfügen kostetlos

```

quicksort(A, l, r)
IF l < r THEN
    p = partition(A, l, r);
    quicksort(l, p);
    quicksort(p+1, r);
    
```

```

partition(A, l, r)
pivot = A[l];
pl = l-1; pr = r+1;
WHILE pl < pr DO
    REPEAT pl = pl+1 UNTIL A[pl] >= pivot;
    REPEAT pr = pr-1 UNTIL A[pr] <= pivot;
    IF pl < pr THEN swap(A[pl], A[pr]);
    p = pr;
return p;
    
```

Terminierung: Induktion immer ein $A[p] \geq \text{pivot}$ rechts von pl , analog pr
in jedem Durchlauf min. $pl = pl + 1$

Konkurrenz: Schleifeninvariante $A[l..pl] \leq \text{pivot}$ nur Ele. $\leq \text{pivot}$
 $A[pr..r] \geq \text{pivot}$ nur Ele. $\geq \text{pivot}$
Abbruch mit $pl = pr$ oder $pl-1 = pr$
Es gilt $p < r$ und $p \geq l$

Laufzeit: Worst Case: schon sortiert $\Theta(n^2)$

Best Case: $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n) \Rightarrow \Theta(n \log n)$

auch noch $\Theta(n \log n)$ bei $T(n) = T(0,7n) + T(0,9n) + \Theta(n)$

Worst-Case-Laufzeit

$$T(n) = \max \{ \# \text{Schritte f\"ur } x \}$$

intuitiver Ansatz Average-Laufzeit

$$T(n) = E_{D(n)} [\# \text{Schritte f\"ur } x]$$

erwartete Anzahl von Schritten über Verteilung
 $D(n)$ auf Eingabedaten der Komplexit\"at n

Aber was ist realistische Verteilung auf Eingaben?

Quicksort f\"ur zuf\"allige Eingaben im Erwartung $\Theta(n \log n)$

\Rightarrow randomized quicksort

partition (A, l, r)

$j = \text{RANDOM}(l, r); \text{ Swap}(A[l], A[j]);$

... // wie normales quicksort

Ermittelter Laufzeit $\Theta(n \log n)$ (t\"atig im Durchschnitt mittig)

Insertion Sort

Laufzeit $\Theta(n^2)$

einfache
Implementierung

f\"ur kleine $n \leq 50$
oft beste Wahl

Merge Sort

Laufzeit $\Theta(n \log n)$

(asymptotische Lauf.

Quicksort

Worst-Case-Laufzeit $\Theta(n^2)$

randomisiert: erwartete Laufzeit $\Theta(n \log n)$

in Praxis meist schneller als Merge,
da weniger Kopieroperationen

Implementierungen schalten f\"ur
kleine n meist auf Insertion Sort

// in Java Dual-Pivot Quicksort

Untere Schranke f\"ur vergleichsbasiertes Sortieren $\Omega(n \cdot \log n)$

sortByComp(α)

done = false;

WHILE !done DO

determine(i, j);

comp(i, j);

set done;

compute(); //only from comp info

return i ;

$n!$: verschiedene Array Permutationen $\pi(\alpha)$

Sei m Anzahl vergleiche 2^m bits Informationen

\Rightarrow f\"ur korrekten Algorithmus $2^m \geq n!$

$$m \geq \log(n!)$$

Stirling Approximation

$$m + \Omega(n \cdot \log(n)) \quad n! \approx \left(\frac{n}{e}\right)^n$$

Radix-Sort

```
radixSort(A) // keys d digits in Range [0, D-1]
    // B[0:D-1] buckets
    FOR i=0 TO d-1 DO
        FOR j=0 TO n-1 DO putBucket(A, B[i], j);
        a=0;
        FOR k=0 TO D-1 DO
            FOR b=0 TO B[k].size-1 DO
                A[a] = B[k][b];
                a=a+1;
            B[B[k].size]=0;
        return A;
```

Korrektheit: Nach i -ter Iteration ist Array
gemäß letzter i Ziffern sortiert

Induktionsschritt: Fall $(i+1)$ -te Ziffer verschieden
gleich

Laufzeit: $O(d \cdot (n + D))$

$O(dn)$

oft d und D als konstant angesehen, dann $O(n)$

aber eindeutige Schlüssel für n Elemente
benötigen $d = \Theta(\log_D n)$ Ziffern $\Rightarrow O(n \log n)$

Stacks LIFO

abstrakter Datentyp: new(S) - erzeugt leeren Stack namens S
isEmpty(S)

pop(S) - gibt oberstes zurück und löscht es vom Stack
push(S, k) - schreibt k als oberstes auf Stack

algebraische Spezifikation: Stack mit new, isEmpty, push, pop

- Regeln: 1) nach new(S) unmittelbar isEmpty(S) ergibt true
- 2) nach push(S, k), unmittelbar pop(S) ergibt k
- 3) ...

Stack als Array: Annahme maximale Größe MAX vorher bekannt
array A, Zeiger top

variable Größe: - Aufteilung auf mehrere Arrays, verkettete Liste
- Umkopieren

jeweils um eins größeres Array Laufzeit $S_2(n)$ durchschnittlich propush
Verdoppeln, Schrumpfen sobald nur Viertel Laufzeit da min n push bis kopieren durchschnitt $O(1)$

new(S)

```
S.A[] = ALLOCATE(1);
S.top = -1;
S.memsize = 1;
```

isEmpty(S)

```
IF S.top < 0 THEN
    return true;
ELSE return false;
```

Java stack

default capacity increment = 2

pop(S)

```
IF isEmpty(S) THEN
    error 'underflow'
ELSE
    S.top = S.top + 1;
    IF 4 * (S.top + 1) == S.memsize THEN
        S.memsize = S.memsize / 2;
        RESIZE(S.A, S.memsize);
    RETURN S.A[S.top + 1];
```

push(S, k)

```
S.top = S.top - 1;
S.A[S.top] = k;
IF S.top + 1 == S.memsize THEN
    S.memsize = 2 * S.memsize;
RESIZE(S.A, S.memsize);
```

Queue FIFO

abstrakter Datentyp: new(Q)
isEmpty(Q)
dequeue(Q)
enqueue(Q, k)

als zyklisches Array:

```
new(Q)
Q.A[] = ALLOCATE(MAX);
Q.front = 0;
Q.rear = 0;
Q.empty = true;
```

isEmpty(Q)

return Q.empty;

```
dequeue(Q)
IF isEmpty(Q) THEN
    error 'underflow'
ELSE
    Q.front = Q.front + 1 mod MAX;
    IF Q.front == Q.rear THEN
        Q.empty = true;
    RETURN Q.A[Q.front - 1 mod MAX];
```

als Liste

```
new(Q)
Q.front = nil;
Q.rear = nil;
```

```
isEmpty(Q)
RETURN Q.front == nil
```

dequeue(Q)

```
IF isEmpty(Q) THEN
    error 'underflow'
ELSE
```

```
x = Q.front;
Q.front = Q.front.next;
RETURN x;
```

enqueue(Q, x)

```
IF isEmpty(Q) THEN
    Q.front = x;
ELSE
```

```
Q.rear.next = x;
x.next = nil;
Q.rear = x;
```

```
enqueue(Q, k)
IF Q.rear == Q.front AND !Q.empty THEN error 'overflow'
ELSE
    Q.A[Q.rear] = k;
    Q.rear = Q.rear + 1 mod MAX;
    Q.empty = false;
```

verkettete Liste

elementare Operationen

search(L, k)
insert(L, x)
delete(L, x)

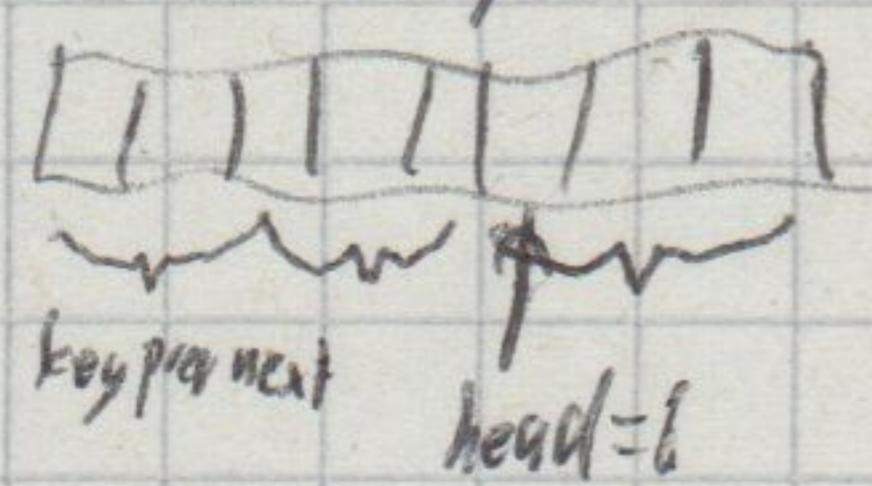
search(L, k)

```
current = L.head
WHILE current != nil AND current.key != k DO
    current = current.next;
return current;
```

Datenstruktur

L.head
Jedes Element +
key
prev
next

z.B. durch Arrays



insert(L, x)

```
x.next = L.head;
t.prev = nil;
IF L.head == nil THEN
    L.head.prev = x
    L.head = x;
```

delete(L, x)

```
IF x.prev == nil THEN
    x.prev.next = x.next;
ELSE
    L.head = x.next;
IF x.next == nil THEN
    x.next.prev = x.prev;
```

Vereinfachung durch

Sentinel mit key = nil
head auf erstes echtes Element
head.prev = sent and sent
(last.next = sent has some ref.)

als Suchindex

Bereichssuche X[i]

Sekundärindizes

	Stack	Queue	Linked List
Einfügen	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Löschen	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Suchen	/	/	$\Theta(n)$

	binär Baum	Best case h = $O(\log_2 n)$	Worst h = n	$E[h] = \Theta(\log_2 n)$
Best Case	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(h)$
Worst Case	$\Theta(n)$	$\Theta(h)$	$\Theta(n)$	$\Theta(h)$

	Binärbaum
$\Theta(1)$	$\Theta(h)$
$\Theta(n)$	$\Theta(h)$
$\Theta(h)$	$\Theta(h)$

Bäume

acykatisch, Wurzel zusammenhängend

Darstellung als ungerichteter Graph (reihenfolge durch Anordnung)

Wurzel/root Tiefe/depth 0

alle Vorfahre/ancestor \Rightarrow Nachkommen/descendant²
Elternk./parent of \Rightarrow Kind/child
nur direkt

Binärbaum jeder Knoten max. 2 Kinder
Halbbalkt \Rightarrow genau ein Kind

Geschwister/siblings

Blatt/leaf

Höhe leerer Baum -1

Traversierung

Inorder
Preorder
Postorder) allow reconstruction
with inorder or suchbaum
and identical Works

connect(T, y, w)

```
y = y.parent;
IF y != T.root THEN
    IF y != v.right THEN
        v.right = w;
    ELSE
        v.left = w;
    ELSE
        T.root = w;
    IF w != nil THEN
        w.parent = y;
```

delete(T, x)

```
y = T.root;
WHILE y.right != nil DO
    y = y.right;
    connect(T, y, y.left);
```

abstrakter Datentyp

```
new(T)
search(T, k)
insert(T, x)
delete(T, x)
```

search(t, root, k)

```
IF x == nil THEN return nil;
IF x.key == k THEN return x;
y = search(x.left, k);
IF y != nil THEN return y;
return search(x.right, k);
```

insert(t, x)

```
IF T.root != nil THEN
    T.root.parent = x;
    x.left = T.root;
    T.root = x;
```

```
IF x != y THEN
    y.left = x.left;
    IF x.left == nil THEN
        x.left.parent = y;
        y.right = x.right;
    IF x.right != nil THEN
        x.right.parent = y;
    connect(T, x, y);
```

Binäre Suchbäume

$\forall z \quad \text{I}_z \leftarrow \text{linker Teilbaum } x, \text{key} \leq z, \text{key}$
 $\forall y \quad \text{I}_y \leftarrow \text{rechter Teilbaum } y, \text{key} > z, \text{key}$

G(h)

interactive-search(x, k)

```
WHILE x != nil AND x.key != k DO
    IF x.key > k THEN
        x = x.left
    ELSE
        x = x.right
    return x;
```

transplant(t, u, v) $\Theta(1)$

```
IF u.parent == nil THEN
    t.root = v;
ELSE
    IF u == u.parent.left THEN
        u.parent.left = v;
    ELSE
        u.parent.right = v;
IF v != nil THEN
    v.parent = u.parent;
```

$\Theta(h)$ insert(t, z)

$x = T.root; px = \text{nil};$

WHILE $x \neq \text{nil}$ DO

$px = x;$

IF $x.key > z.key$ THEN

$x = x.left;$

ELSE

$x = x.right;$

$z.parent = px;$

IF $px == \text{nil}$ THEN

$T.root = z$

ELSE

IF $px.key > z.key$ THEN

$px.left = z;$

ELSE

$px.right = z;$

delete(t, z) // $\Theta(h)$

```
IF z.left == nil THEN
    transplant(t, z, z.right)
ELSE
    IF z.right == nil THEN
        transplant(t, z, z.left)
    ELSE
        y = z.right;
        WHILE y.left != nil DO y = y.left
```

```
IF y.parent != z THEN
    transplant(t, y, y.right);
    y.right.parent = z.right;
    y.right.parent = y;
    transplant(t, z, y);
    y.left = z.left;
    y.left.parent = y;
```

Rot-Schwarz-Bäume (n Knoten garantiert $h \leq 2 \cdot \log_2(n+1)$)

Anwendungen: Linux Completely Fair Scheduling
jede TreeMap

1) \forall Knoten: rot \oplus schwarz

Beweis Schr. 0.

Leerer $2^{SH(x)} - 1 = 2^0 - 1 = 0$ Knoten \times rot \times schwarz

2) Wurzel.color = schwarz

Teilbäume Schwarzhöhe = $SH(x)$ oder $SH(x)-1$

3) Knoten rot \rightarrow Kinder schwarz

Nicht Rot-Rot-Regel

4) \exists Knoten ∇ Pfade zu (Halb-)Blatt
gleiche Anzahl schwarzer Knoten \rightarrow Schwarzhöhe
inkl. Knoten selbst

Voraussetzung $2^{SH(x)} - 1$ Knoten in Teilbäumen

mind. $(2^{SH(x)-1}) + (2^{SH(x)-1}) + 1 = 2^{SH(x)-1}$ im Teilbaum von x

Sei h Höhe Baum mit Wurzel r und n Knoten

$$SH(r) \geq h/2 \Rightarrow n \geq 2^{\frac{h}{2}} - 1$$

$$\Rightarrow \log_2(n+1) \geq h/2$$

Implementierung mit schwarzen Sentinel, der nur auf sich selbst zeigt.

Algorithmus Hilfsfunktionen

5(1) rotateLeft(T, x) // $x.right != null$

$y = x.right$
 $x.right = y.left$

IF $y.left != null$ THEN

$y.left.parent = x$

$y.parent = x.parent$

IF $x.parent == T$ setzt THEN

$T.root = y$

ELSE

IF $x == x.parent.left$ THEN

$x.parent.left = y$

ELSE

$x.parent.right = y$

$y.left = x$

$x.parent = y$

insert(T, z) // $O(\log n)$

// wie binärbaum

~~z.color = red;~~

fixColors after insertion(T, z);

fixColors after insertion(T, z); // $O(h) = O(\log n)$)

solange $z.parent.color == \text{red}$ // verstoß, Rot-Rot-Regel

kein Unter Vorfahr schwarz Onkel

|

unterer roter an Rand rotieren
oberer Rote Farbe mit Parent tauschen
Jetzt schwarzen hochschierien

fertig

delete(T, z) // $O(\log n)$

// wie bin Baum

IF hochgelöschter.color was Black

Fixup (ungleichgewicht in hochgelöschter.parent)
 ΔSH auf Seite des hochgelöschten { links: -1
rechts: +1
links-mitte: }

neues Kind ist
neues Kind + schwarz
fertig

neues Kind schwarz
Schwester rot und deren Kinder
Schwester + rot schwarz

neues Kind rot
notiere rottes nach
oben
nach unten notiertes = rot

Außeres rot
rotiere Knoten mit $\Delta SH-1$
zu Seite mit kleinerem
Fahrer wird gewechselt

parent rot
parent + schwarz
fertig

parent schwarz
Ungleichgewicht
in Großvater

AVL-Bäume Georgie Maximowitsch Adel'son-Velski und Jewgeni Michailowitsch Landis

Garantie $h \leq 1,447 \cdot \log n$ (Optimierte Konstanten gegenüber Rot-Schwarz)

Balance in Knoten x : $B(x) = \text{Höhe}(\text{rechter Teilbaum}) - \text{Höhe}(\text{linker Teilbaum})$

$\forall x \quad B(x) \in \{-1, 0, +1\}$ Beweisidee $n_h = \min \text{Anzahl Knoten AVL-Baum Höhe } h$

aber höherer rebalancieren Aufwand

besser Suchen Modifizieren
AVL \leq Rot-Schwarz

AVL Baum Höhe $h \Rightarrow$ Rot-Schwarz $SH = \lceil \frac{h+1}{2} \rceil$

$$n_h = 1 + n_{h-1} + n_{h-2} \quad \text{Fibonacci} \quad F_0=0 \quad F_1=1$$

$$= F_{h+2} - 1$$

$$F_n = F_{n-1} + F_{n-2}$$

$$F_{h+1} \approx \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+3}$$

$$h \approx 1,447 \log_2 n$$

rekursiv
 h gerade / h ungerade
für beide Teilebaum
 $h_{\text{Teil}} = h-2$ $h' = h-1$
 $h_{\text{Teil}} = h-2$ Rote Wurzel

Rot Schwarz Baum schwarz Wurzel
rote Wurzel $SH = \lceil \frac{h+1-2}{2} \rceil = \frac{h-1}{2}$
 $SH = \frac{h-1}{2}$

Anfang
 0 0
 $SH = 1 = \lceil \frac{h+1}{2} \rceil$ $SH = 0 = \frac{h}{2}$

Für jede Höhe $h \geq 3 \exists$ Rot-Schwarz
der kein AVL-Baum

Einfügen $O(\log n)$

wie Binärbaum

für rekursiv von eingefügtem

wenn Knoten $B(x) = \pm 2$

falls notwendig rotiere in Ebene drunter
Höhenzunahm nach außen

rotiere x

rebalancieren nur einmal nötig!

Löschen $O(\log n)$

wie Binärbaum

rebalancierung etl. bis Wurzel nötig

Splay Bäume ~~Setzt Annahme~~: einmal angefragte voraussichtlich öfter
gleiches Prinzip Selbst-Organisierende Linke
angefragtes wird hervor

Anwendung z.B. SQUID • Web-Cache-Proxy (Access Control Listen für HTTP-Zugriffe)

Bei Suchen und Einfügen nach oben spülen $O(h)$

- > Zig-Zag-Operation erst unten dann aber rotieren
- \ Zig-Zig-Operation oben unten
- \ Zig-Operation

Amortisierte Laufzeit für $m \geq n$ $\frac{\text{Operations}}{\text{Knoten}} \leq \max$
Worst Case $O(m \cdot \log_2 n)$
pro Operation $O(\log_2 n)$

Löschen $\times O(h)$

\times nach oben spülen \rightarrow Löschen \in Teilbaum L und R
größtes in L hochspülen \rightarrow R drehen

(Binäre Max-)Heaps (keine BST)

- 1) bis auf unterstes Level vollständig $\Rightarrow h \leq \log n$
unteres Level von links aufgefüllt
- 2) $\forall x \neq \text{root}: x.\text{parent.key} \geq x.\text{key}$ \Rightarrow Maximum in root

Einfügen $O(\log n)$

nach oben Tauschen
bis erfüllt

Lösche Maximum
ersetze durch letztes Blatt
nach unten durch max Kinder tauschen
bis erfüllt

durch Anraus mit H.length

$$j.\text{parent} = \lceil \frac{j}{2} \rceil - 1$$

$$j.\text{left} = 2(j+1) - 1$$

$$j.\text{right} = 2(j+1)$$

$$O(n \cdot h) = O(n \log n)$$

Heap-Konstruktion aus unsortiertem Array
heapify von Blättern aus, da diese triviale Max-Heaps
von $\lceil \frac{H.length - 1}{2} \rceil - 1$ bis 0.

\Rightarrow Heapsort $O(n \log n)$

Anwendung abstrakter Datentyp Priority Queue zum
(new(Q), isEmpty(Q), max(Q), extract-max(Q), insert(Q, k))

untergrenze Anzahl Werte

$$1) \forall \text{ Knoten } x \text{ root: } t-1 \leq x.\text{n} \leq 2t-1 \quad (\geq \text{key}[0], \dots)$$

2) Werte in Knoten aufsteigend geordnet

3) \forall Blätter gleiche Höhe

4) Inneren Knoten $x.n = t-1$ Kinder

\forall Werte aus $k_0 \leq x.\text{key}[0] \leq \dots \leq x.\text{key}[t-1] \leq \dots$

Höhe

Wurzel: min. 1 Wert

1. Ebene: min. 2 Knoten mit min. 2 Kindern

i-te Ebene: min. $2^{t(i-1)}$ Knoten mit je min. 2 Kindern

$$\Rightarrow n \geq 1 + (t-1) \cdot \sum_{i=1}^h 2^{t-1} = 1 + 2(t-1) \cdot \frac{2^h - 1}{2 - 1}$$

$$\Rightarrow \log_2 \frac{n+1}{2} \geq h$$

$$\frac{2^h}{2} \leq n \leq 2^h$$

Alternative Bedeutung

Anwendung

Lesen/Schreiben
in Blöcken von $\approx 12\text{-}16\text{K Bytes}$ \Rightarrow mehrere Werte
auf einmal

($\mathcal{O}(\log_2 n)$)

Einfügen

in BT-Baum, falls $2t-1$ Werte
falls schon $2t-1$ Werte vorher splitten
Mittlerer Wert in höheren Knoten zw. 2
Teilknoten als neue Kinder
rekursiv $t+1$
split an Wurzel erzeugt neue Wurzel

Baumknoten

alternative Definition

$$f = t-2$$

$$\text{daher } \frac{t}{2} \leq x.n \leq t$$

$\frac{2^h}{2} \leq$ Baum / $(2, 4)$ -Baum
für $t=2$

(statt Suche, dann Splitten)

Suchen und Splitten

splitten schon

bei Suche

\Rightarrow Reduktion teure direkt

Operativer

$\mathcal{O}(\log_2 n)$

Löschen - auch hier Löschen und Splittern

- im Blatt

f noch $\geq t-1 \Rightarrow$ fertig

retieren Wert von Geschwisterknoten

somit verschmelzen Geschwisterknoten \rightarrow evtl. parent zu klein

im inneren Knoten

$\left. \begin{array}{l} \text{Wert von einem der beiden direkten Kinder hochholen} \\ \text{sonst (falls Kinder zu klein) diese verschmelzen evtl. parent jetzt zu klein} \end{array} \right\}$

jetzt

zu klein

randomisierte Datensstrukturen (nicht deterministisch)

Skip Lists Worst Case $\Omega(n) \sim \Theta(\log_p n)$

rekursiv Express Listen mit ausgewählten Elementen

Wahrscheinlichkeit p Aufnahme in Expressliste

durchschnittliche Höhe $h = O(\log_p n)$

Suche

Worst-Case Beendigung auf unterster Ebene

im Durchschnitt nach Erwartungswert $\frac{h}{p}$ Schritte pro Ebene

$$\Rightarrow \frac{h}{p} = O(h) = O(\log n) \text{ Durchschnitt}$$

Einfügen Durchschnitt $O(h)$

Suchen Position

mit p in höhere Lüte \Rightarrow Erwartungswert $\frac{n}{p} E(1)$

Lösche Durchschnitt $O(h)$

nach Fund auch in allen höheren Ebenen

L. head
height

x. key

next, prev, down, up

nil - kein Nachfolger
sentinels

Linearität Erwartungswert

$$E\left[\sum_{i=1}^n a_i \cdot X_i\right] = \sum_{i=1}^n a_i \cdot E[X_i]$$

$$\Rightarrow E\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n E[X_i] = p n \text{ im Durchschnitt}$$

7. Ebene $n p^n$ Elemente

Erwartungswert geometrisch verteilt
 $\frac{1}{p}$ zufallsvariante

$$\Theta(\log_p n)$$

$$\text{Durchschnittlicher Speicherbedarf } n \cdot \sum_{i=0}^h p^i = \frac{n}{1-p}$$

+ unterstützt parallele Verarbeitung
nur sehr lokale Änderungen

Hash Tables $\sim \Theta(1)$

$h(x)$ - uniform und unabhängig abhängt von $T[h(x)]$
in $[0, T.length - 1]$ verteilt

Kollision \Rightarrow hier verkettete Liste zur Auflösung

Einfügen konstant $\Theta(1)$

Suchen/Löschen wie Liste lang

$h(x)$ uniform \Rightarrow Erwartungswert $\frac{n}{T.length}$

$\Theta\left(\frac{n}{T.length}\right)$ bei $T.length \geq n$
Durchschnitt konstant

Anwendung z.B. MySQL hash indexes
Btrees best support

+ default load factor 0.75 good tradeoff

- expected workload important

- kein schnelles Treidern zu Nachbarn

„Universelle“ Hash-funktion p prim
zufällig $a, b \in [0, p-1]$

$$h_a, b(x) = ((ax+b) \bmod p) \bmod T.length$$

Verteilung $t_x, t \in [0, p-1]$ gilt $P[(ax+b) \bmod p = t] = \frac{1}{p}$

Zu gegebenen t, x, a genau ein b mit $h_a, b(x) = t$

Unabhängigkeit/Kollisionsresistenz $t_x + y, t_x, t_y \in [0, p-1]$

$$P\left[\begin{array}{l} (ax+b) \bmod p = t_x \\ (ay+b) \bmod p = t_y \end{array}\right] = \frac{1}{p^2}$$

Kryptographische MD5, SHA-2, SHA-3

Bloom-Filter Speicherbesparende Wörterbücher mit kleinem Fehler z.B. schlechte Passwörter

n Elemente beliebiger Komplexität

m Bits Speicher üblicherweise Bit-Array

k gute Hash Funktionen H_1, H_2, \dots, H_k
mit Bildbereich in m

$$\text{empfohlen } k = \frac{m}{n} \ln 2$$

\Rightarrow Fehlerrate ca. 2^{-k}

üblicherweise $k = 5, 6, \dots, 20$

Erstellen:

init 0

$$BF[H_i(x_j)] = 1 \text{ für}$$

Wenn Filter groß genug (nur zur Hälftezen)

$$\Rightarrow \text{Fehler} \leq 2^{-k}$$

Suchen

Test ob Bits aller Hashtests gesetzt

Löschen

neuen ausführen

v Anzahl zu speichern auf 0 setzen, wenn alle gefascht

Vereinigen zweier durch Bitwizes oder

(endliche) Graphen

$$G = (V, E)$$

vertices
Edges
 $E \subseteq V \times V$

gerichtet
stark zusammenhängend
 $V = \{1, 2\}$
 $E = \{(1, 2)\}^3$
 v_1 von v_2 erreichbar
ungerichtet
 $V = \{1, 2\}^3$
 $E = \{\{1, 2\}\}^3$
 $(\exists u, v \in V) (u, v) \in E \Leftrightarrow (v, u) \in E$
zusammenhängend v_1 von v_2 erreichbar

Subgraph G' , wenn gpb. auch ungerichtet (gleicher Typ)

$$\text{und } V' \subseteq V \quad E' \subseteq E \quad E' \subseteq V \times V'$$

$V = \{1, 2\}$
 $E = \{(1, 2)\}^3$
 b ist Baum $\Leftrightarrow V = \emptyset \vee \exists r \in V \text{ Frei}$
eindeutiger Pfad von r zu v

v von u erreichbar $\Leftrightarrow \exists (w_1, \dots, w_k) \in V^k : w_i \in (w_j, w_{j+1}) \in E \quad \forall i \quad w_1 = u \quad w_k = v$

Länge des Pfades = $k - 1 = \# \text{Kanten}$

kürzester Pfad \Leftrightarrow kürzerer

quadratisch Adjazenzmatrix
 $A[i,j] = \begin{cases} 1, & \text{wenn Kante von } i \text{ zu } j \\ 0, & \text{sonst} \end{cases}$

$$\text{Speicherbedarf} = \Theta(|V|^2)$$

ungerichtet $\Rightarrow A$ (spiegel-)symmetrisch

noch (Adjazenzmatrix)

$$A^m[i,j] = \text{Anzahl Wege von } i \text{ nach } j \text{ mit genau } m \text{ Kanten}$$

Adjazenzliste

Array von Verkettetenlisten $v \mapsto \text{List}(v \text{ verbundene vertices})$

$$\text{Speicherbedarf} = \Theta(|V| + |E|)$$

Gewichtete Graphen

zusätzlich Funktion $w: E \rightarrow \mathbb{R}$

für ungerichtete $w((u,v)) = w((v,u)) \quad \forall (u,v) \in E$

Algorithmen

$$O(|V| + |E|)$$

jeder Knoten nur einmal
und dazugehörige Kanten

BFS - Breadth First Search - erst unmittelbare Nachbarn

zusätzliche vertice attribute: distance, predecessor, color (WHITE = nicht besucht)

init $\forall v, \text{null WHITE}$

start Knoten s , null GRAY

Queue mit Startknoten

dazu $y = \text{degrene}$

$v \notin \text{adjacent}$

$v = p$ ~~aus~~ y .dist + 1 $\leftarrow u$ GRAY

enqueue(v)

$u \leftarrow \text{black}$

GRAY = in Queue für nächsten Schritt

BLACK = fertig

z.B. Web Crawling, Kontakte in P2P, Broadcasting, Garbage Collection

Iteration	u	v	Q
0	-	-	[]
1	s	(1, 2, 3)	[s]
2	1	(1, 2, 3)	[1, 2]

abgeleiteter BFS-Baum

$$V^S_{\text{pred}} = \{v \in V \mid v.\text{pred} \neq \text{NIL}\} \cup \{s\}$$

$$E^S_{\text{pred}} = \{(v.\text{pred}, v) \mid v \in V^S_{\text{pred}}\} \Rightarrow \{(s, 1), (s, 2), (1, 3), (2, 3)\}$$

\Rightarrow kürzeste Pfade (durch pred.) (Ausgabewert BFS $O(|V|)$)

DFS - Depth-First Search - erst noch nicht besuchte / möglichst weit weg

zusätzliche Attribute: ~~pred~~ pred, color, discover, finish time

globale Variable time = 0

init ($u = s$)

time++

~~v.color = GRAY~~ disc = time

rekursive in alle weißen Adjacenten

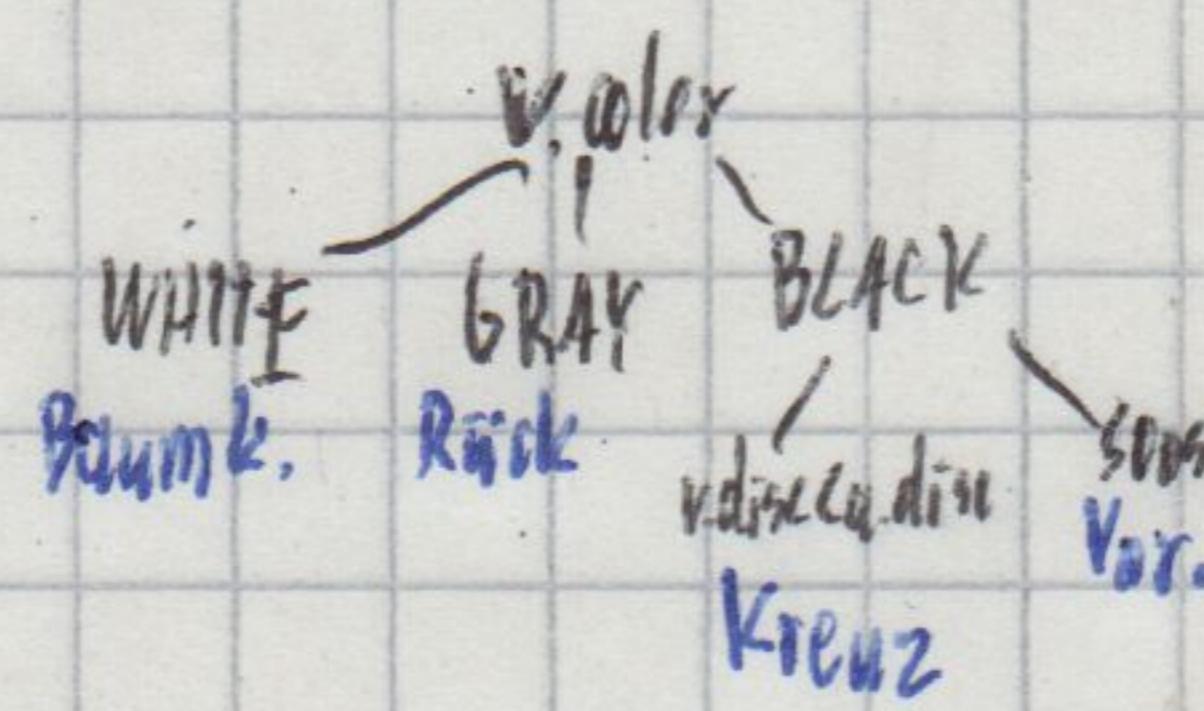
~~v.color = BLACK~~

time++

$u.\text{finish} = \text{time}$ restart

\Rightarrow keine Zeit doppelt für unbedeckte

	1	2	3	4	5
Knoten	1	2	3	4	5
Entdeckungszeit	1	2	3	4	5
Abschlusszeit	6	7	8	9	10
Vergangen	nil	1	2	3	nil



DFS-Wald = Menge DFS-Bäume

$$b_{\text{pred}} = (V, E_{\text{pred}})$$

$$E_{\text{pred}} = \{(v.\text{pred}, v) \mid v \in V, v.\text{pred} \neq \text{nil}\}$$

Charakterisierung Kanten

im Algorithmus (u, v) betrachtet

$\Rightarrow v.\text{color} = \text{WHITE}$

Baumkante ($\Rightarrow \# E_{\text{pred}}$)

$\Rightarrow v.\text{color} = \text{BLACK}$

Vorwärtskante (\Rightarrow zu Nachkommen in b_{pred} nicht Baumkante)

$\wedge u.\text{disc} < v.\text{disc}$

Rückwärtskante (\Rightarrow Vorfahren (linker Verzweigungsschleifen))

$\Rightarrow v.\text{color} = \text{GRAY}$

Kreuzkante (\Rightarrow sonst)

$\wedge v.\text{disc} < u.\text{disc}$

BRUNNEN

ungerichteter Graph: nur Baum und Rückwärtskanten

$O(|V| + |E|)$

Topologisches Sortieren

z.B. Job Scheduling (Spreadsheet formulieren, makefile, machine learning)

nur bei dag = directed acyclic graph

lineare Ordnung Knoten mit $\overrightarrow{u \rightarrow v}$

$$(u, v) \in E \Rightarrow u \text{ vor } v$$

run DFS \Rightarrow lineare Ordnung nach finishTime absteigend

Beweis: Folie der Kanten

Alternatives Verfahren, wenn Anzahl eingeckender
Kanten bekannt:
suche Node mit 0 eingehenden Kanten aus!
entferne aus Graph

Starke Zusammenhangskomponente (SCC) (SCC - strongly connected components)

a) $\forall u, v \in C \exists$ Pfad von u nach v

b) $\exists C \subseteq D \subseteq V$: für die a) auch gilt
(maximal)

Eigenschaften

SCCs disjunkt (keine gemeinsamen Knoten)

Falls Pfad von einer zur anderen \Rightarrow keinen zurück

Algorithmus to find SCCs in Graph

Kosaraju Algorithmus

run DFS(G)

compute G^T // edges umdrehen (gleiche SCCs)

run DFS(G^T) but visit vertices in main loop
in descending finish time from DFS(G)

\Rightarrow DFS trees are SCCs

Alternativen

Tarjans Algorithmus

Pfad-basierter Algorithmus

\Rightarrow je nur 1 DFS-ausführen,
mehr Infos auf dem Weg

MST - Minimaler Spannbaum nur für ungerichtete, gewichtete

z.B. Multicast

Spannbaum =zyklischer Subgraph T ,
der V Knoten verbindet

Terminologie

Schnitt ($S, V \setminus S$) = Partitionierung Knoten in 2

respektiert ASE $\Leftrightarrow \exists (u, v) \in A$ der Schnitt
überbrückt

$\in U \setminus S \quad v \in V \setminus S$

minimal $w(T) = \sum_{(u, v) \in E_T} w(u, v)$

$w(u, v) = \min_{r \in R} w(u, r) + w(r, v)$

$O(|E| \cdot \log |E|)$ mit vielen Optimierungen $O(|E| \cdot \log |V|)$

Kruskal - lässt mehrere Unterbäume parallel wachsen

zusätzliches Attribut set mit nur mit sich selbst für v.
gleich Mengen

Kanten in aufsteigender Reihenfolge

if $u.set \neq v.set$ $\{u, v\} \in E$ $w(u, v)$ gebrückt? $Set(u) \cup Set(v)$

vereinige Sets

$set(u), v \in set(u)$

Kante zu T hinzufügen

speziell Fibonacci Heaps

mit vielen Optimierungen $O(|E| + |V| \cdot \log |V|)$

Prim - Knoten für Knoten

root can
be random?

zusätzliche Attribute key, pred init ∞ , null

global ΔQ neue nach zu bearbeitender Knoten

root Knoten, key = ∞

extract-min(Q) kleinster key

falls kleiner reduziert key Nachbarn

auf weight Verbindung, setzen pred $\leftarrow u$

$\Rightarrow T = \{(u, v, pred)\} \cap \{v \in V \mid \exists u \in Q\}$

$\begin{array}{c} \text{a, b, c, d, e, f, g, h} \\ \text{a, b, c, d, e, f, g, h} \\ \text{a, b, c, d, e, f, g, h} \\ \text{a, b, c, d, e, f, g, h} \\ \text{a, b, c, d, e, f, g, h} \\ \text{a, b, c, d, e, f, g, h} \\ \text{a, b, c, d, e, f, g, h} \\ \text{a, b, c, d, e, f, g, h} \end{array}$

leichte Kante (u, v) für Schnitt ($S, V \setminus S$)

$\Leftrightarrow w(u, v) = \min(\text{überbrückende Kanten})$

leicht \Rightarrow sicher Beweis: Annahme MST ohne diese

Konstruktion kürzeren ~~und teurer~~ mit dieser

zyklich

\Rightarrow entferne Schleife

\Rightarrow Schnitt A

~~abreissen~~ der von dieser

überbrückt

Greedy-Strategie

Korrektheit

Schnitt ($R, V \setminus R$) respektiert T

$(u, v, pred)$ leichte Kante

$(A, V \setminus A)$

(zu füllen)

SSSP - Single Source Shortest Path kürzeste „Gewichtswege“ \leftrightarrow BFS „Kantenwege“

$$\text{Pfadlänge } w(p) = \sum_{i=1}^{k-1} w((v_i, v_{i+1}))$$

$p: (v_1 \dots v_k) \in V^*$

Theorie:

Graph ohne Negativen Schwingungen - Zyklen mit Gewicht < 0 , sonst Gewicht beliebig klein

kürzeste Pfade keine Zyklen mit positivem gesamtgewicht

\Rightarrow kürzeste Pfade höchstens eliminierbare Zyklen mit Gewicht $= 0$

kürzeste Teilstücke auch kürzeste Pfade

Algorithmen zusätzliche Attribute dist, pred
Grundidee

relax(b, u, v, w)

wenn über u kürzer nach v
 \Rightarrow Weg aktualisieren

3 Algorithmen mit ^{anderen} zunehmenden Verbedingungen

Bellman-Ford-Algorithmus - funktioniert allgemein

init

$(N-1) \times$ relax aller Kanten

keine negativen Zyklen, wenn $v.dist \leq u.dist + w(u, v)$ $\forall (u, v)$

$O(|V| \cdot |E|)$

Korrektheit Pfad Knotenzahl $\leq |V|$ (s schon init drin)
jeder Schritt wächst entdeckt, kann nicht zerstört werden, da somit kürzester Pfad

für dags $O(|V| + |E|)$

init
topologisch sortieren
in dieser Reihenfolge
relax vorausgehender Kanten

Korrektheit: Reihefolge des kürzesten Pfad

Dijkstra Algorithmus - nur, $\forall w(u, v) \geq 0$

init global $Q = V$
while !isEmpty Q
 $u = \text{Extract-Min}(Q)$ // dist
relax all adjacent

Korrektheit
dist von u immer final
sonst ginge weg über einen aus Q , was nicht kürzer sein kann

$O(|V| \cdot \log |V| + |E|)$

bei negativen Kanten aber schon

A* Algorithmus - benötigt Heuristik

wie Dijkstra mit

$w(u, v) = w(u, v) + u.\text{heur} - v.\text{heur}$

neuer
Überschätzung $v.\text{heur} \leq \text{shortest}(u, v)$
 $u.\text{heur} \leq w(u, v) + v.\text{heur}$
mindestens

z.B. Bipartiter Matching (z.B. Käufer, Verkäufer)

Netzwerkflüsse

von Quellen s ^{Kosten} zu Senken t von t von s erreichbar

maximaler Fluss - Ford-Fulkerson

Kapazität $c(u, v) \geq 0$

$(u, v) \notin E \Rightarrow c(u, v) = 0$

Restkapazität

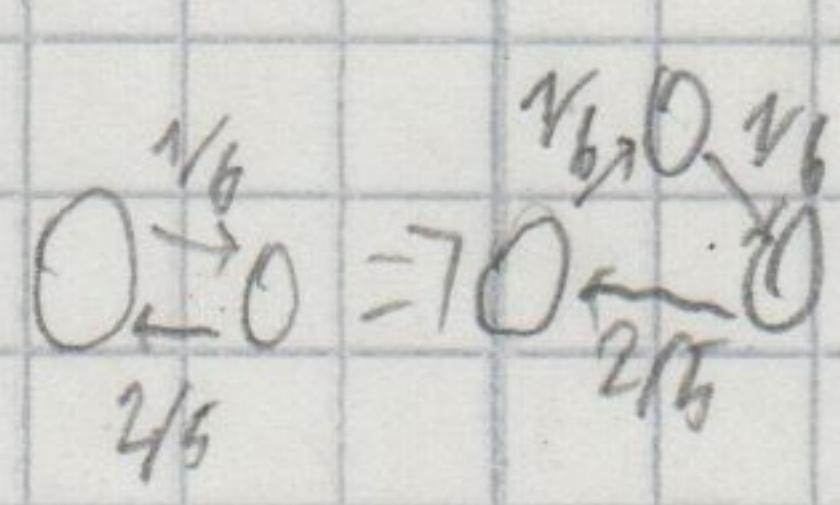
$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{falls } (u, v) \in E \\ f(u, v) & \text{falls } (u, v) \notin E \\ 0 & \text{sonst} \end{cases}$

Antiparalleler Kante erlaubt

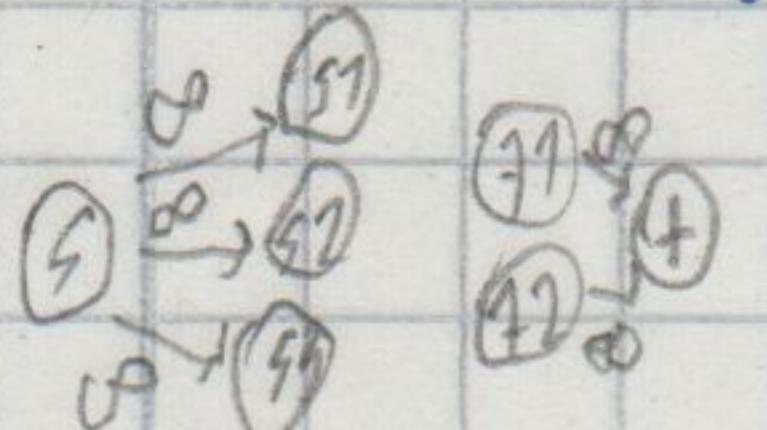
Fluss $v_{u,v} \quad 0 \leq f(u, v) \leq c(u, v)$

$\sum_{v \in V \setminus \{s\}} f(s, v) = \sum_{v \in V \setminus \{v\}} f(v, v)$

Wert $|f| = \sum_{v \in V \setminus \{s\}} f(s, v) = \sum_{v \in V \setminus \{v\}} f(v, v)$



vorherige Transformationen
Eliminieren antiparalleler Kanten
 Vereinigung Quellen & Senken



Algorithmus

init alle.flow = 0

solange Pfad p from stat in Restkapazität graph b_{flow}

$c_{\text{flow}}(p) := \min(c_{\text{flow}}(u, v) | (u, v) \in p)$

durch $e \in p$

Fall $e \in E$ $e.\text{flow} += c_{\text{flow}}(p)$

$e \in E$ $e.\text{flow} -= c_{\text{flow}}(p)$

BRUNNEN

Korrektheit Max-Flow Min-Cut Theorem

f maximaler Fluss $\Rightarrow b_f$ keinen erweiterbaren Pfad

$\Leftrightarrow |f| = \min_S c(S, V \setminus S)$ mit $s \in S$ & $t \in V \setminus S$

Kapazität Schnitt $\sum_{u \in S, v \in V \setminus S} c(u, v)$

Advanced Designs

Divide & Conquer
Löse rekursiv (disjunkte) Teilprobleme

z.B. Quicksort, Mergesort

Backtracking

durchsuche iterativ Lösungsraum
Lösung $x = (x_1, x_2, \dots, x_n)$ per Trial and Error

Teillösung durch Kandidaten x_i ergänzen

Feststellung keine Beantwortung erreichbar

\Rightarrow rerevidieren Kandidat x_{i+1}

~ Tiefensuche auf Rekursionsbaum
abschneiden aussichtsloser Lösungen

„intelligenter“ als Brute-Force

Lösungssuche

- feine
- alle
- beste

z.B. Sudoku

$(i, j) = \text{next free Pos}$
try all Einträge nach den Regeln

regelmäßiger Ausdruck - Mustersuche in Strings

exponentieller Aufwand

Dynamische Programmierung

überlappende Teillprobleme

Memoization der Zwischenergebnisse

z.B. Fibonacci-Zahlen

Speichere Zwischenergebnisse in Array

Minimum Edit Distance

$X[1..m]$ in $Y[1..n]$ überführen

- insert
- delete
- substitution
- (copy) ohne Kosten

Teilproblem

$D[i][j] := \text{Distanz von } X[1..i] \text{ zu } Y[1..j]$

$D[0][j] = j$

$D[i][0] = i$

$D[i][j] = \min \{ D[i-1][j-1] + (\text{if } X[i] = Y[j]) \text{ else } 1, \dots \}$

$D[i-1][j] + 1 \quad \text{del}$

$D[i][j-1] + 1 \quad \text{ins}$

Berechnung durch $\text{for } i=1 \text{ to } m \text{ for } j=1 \text{ to } n \dots$

 $\Theta(mn)$

Greedy-Algorithmus

Lösung $x = (x_1, x_2, \dots, x_n)$

Ergänzung lokel günstigstem x_i

z.B. Dijkstra, MST-Kruskal, Prim

Traveling Salesman Problem - jeder Kasten genau einmal

Greedy: hinzufügen billigste Kante zu unbenannten Kästen
aber nur Näherung

Metaheristiken \Rightarrow Heuristiken eines speziellen Problems

allgemeine Vorgehensweise

zum Leiten Suche bei

Optimierungsproblemen

Tabu Search

suche in Nähe, vermeide schon besuchte
wenn keine bessere in Nähe akzeptiere auch schlechter

Evolutionäre Algorithmen

Lösungspopulation
beste für Reproduktion

z.B. für lokale Suche

Hill-Climbing-Strategie

$sol = \text{initialSol}(P)$

time = 0

WHILE time < maxTime DO

new = perturb(P, sol);

IF quality(P, new) > quality(P, sol) THEN

$sol = new$

time++

return sol

\Downarrow

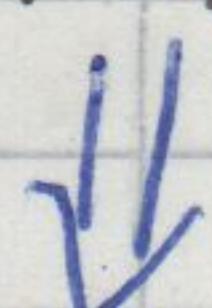
Heretikre Lokale Suche
mehrmauls starten, \Rightarrow beste Lösung

Simulated Annealing ^{von Metfall.}
akzeptiere auch schlechtere Lösung
mit $e^{\frac{\text{quality}(\text{new}) - \text{quality}(\text{sol})}{\text{temperatur}}}$

Schwarmoptimierung, Ameisenkolonialisierung

Komplexitätsklassen

Berechnungsproblem
 $P \rightarrow PS$



Entscheidungsproblem
 $P \rightarrow \{0, 1\}$

Bitlänge Lösung und decide
 polynomial beschränkt
 \Rightarrow compute polynomial

Reduktion
 $compute(P, S) \xrightarrow{?} \text{decide}(P, S)$
 $s = "0"$
 ob Teil einer Lösung
 IF $\text{decide}(P, S) == 0$ THEN return "no solution";
 $\text{done} = \text{false}$
 WHILE !done DO
 $s_{\text{zero}} = \text{decide}(P, S + "0");$
 $s_{\text{one}} = \text{decide}(P, S + "1");$
 IF $s_{\text{zero}} == 0 \text{ AND } s_{\text{one}} == 0$ THEN
 $\text{done} = \text{true}$
 ELSE IF $s_{\text{zero}} == 1$ THEN $S = S + "0"$
 ELSE $S = S + "1"$
 return S

unentscheidbar

Haltproblem
 Problem $H(P) = \begin{cases} 1 & \text{falls } P \text{ PNP} \\ 0 & \text{sonst} \end{cases}$

$H^*(P) = \begin{cases} \text{hält ab falls } H(P) = 0 \\ \text{hält nicht ab sonst} \end{cases}$

Code-Erreichbarkeit darauf zurückführbar

Def H Def H^*
 $H(H^*) = 1 \Leftrightarrow H^*(H^*) \text{ anhält} \Rightarrow H(H^*) = 0$
 \Rightarrow Widerspruch \Rightarrow unentscheidbar

$P \in L_E \Leftrightarrow A_{L_E}(P) = 1 \quad \forall P \exists \text{ Polynomialzeit-Algorithmus } A_{L_E}$

$\Leftrightarrow L_E \text{ ist in } P$

Menge $L_E = \{P \mid P \text{ hat Eigenschaft E}\}$

NP vermeintliche Lösung / Witness / Zertifikat \Rightarrow hat polynomiale Komplexität
 in polynomialer Zeit überprüfbar

z.B. Primfaktorzerlegung

$P \in L_E \xrightarrow{\text{zeuge}} \exists S_P : A_{L_E}(P, S_P) = 1 \quad \forall P$

$P \subseteq NP$ aber $P \neq NP$?

$P = NP \Rightarrow (L \in NP \Rightarrow L \in P)$

$\Rightarrow NP$ Complete SNP falls $P \neq NP \Rightarrow NP \not\subseteq P$

To visit every node exactly once
 z.B. Hamiltonian Cycle $\subseteq TSP$

assing existing edges of all other nodes

$SAT \leq MAX-2SAT$

Belegung dient min. k Konzessionsanträgen

$a(x_1, \dots, x_n) = \neg_1(x_1 \vee \neg x_2 \vee x_3 \vee \dots \vee x_n)$

\vdash in Klauseln

$\emptyset \models (x_1, \dots, x_n, w_1, \dots, w_m)$

$= \neg_1(x_1) \neg_1(x_2) \neg_1(x_n) \neg_1(w_i)$

$\neg_1(x_i, w_j) \neg_1(\neg x_i, w_j) \neg_1(x_i, \neg w_j)$

SAT - Mutter aller NPC-Probleme

$\neg_1(x_i, w_j) \neg_1(x_j, w_i) \neg_1(x_k, w_i) \neg_1(x_i, w_k)$ Erfüllende Belegung für boolesche Formel mit n Variablen?

Erfüllbar \Rightarrow min 7m in \emptyset

nicht \Rightarrow weniger

$\forall P \in NP$ hat polynomialen Verify-Algorithmus \Rightarrow kodiere Anfangszustand als boolesche Formell

bleibt polynomial

Approximation? 3SAT $E[K_i] = \text{Prob}(K_i = 1) = \begin{cases} 1/2 & \text{wenn } x_i \text{ legitime Schritte} \\ 3/4 & \text{x}_i \vee x_j \text{ legitimen Endzustand} \\ 7/8 & x_i \vee x_j \vee x_k \end{cases}$

bleibt polynomial

Linearität Erwartungswert $E[\text{erfüllt Klauseln}] = E\left[\sum_{i=1}^n E[K_i]\right] = \sum_{i=1}^n E[K_i] \geq m/2$

Min. Algorithmen

Behauptung erfüllt im Erwartungswert $\frac{1}{2}$ aller Klauseln

BRUNNEN

2-Färbbarkeit einfach
 die Färbung mit BFS bis Widerspruch
 ! eventuell Neustart nötig
 dafür auf ungerichtetem Ergebnis gleich

2SAT grau Färbbarkeit $\Leftrightarrow \neg x_j \vee \neg x_k \text{ und } \neg x_k \vee \neg x_j$ nicht
 konstruiere Implikationsgraph
 $V = \{x_i\} \cup \{\neg x_i\}$ $E = \{(x_j, \neg x_k), (\neg x_k, \neg x_j)\} \cup \{(\neg x_j, x_k)\}$

SCC in topologischer Sortierung
 $x_j \leftarrow \neg x_j$ zu letzter verkommt
 $\neg x_j \leftarrow \neg \neg x_j$