

Algorithmen

17. Juli 2007

Die Vorlesung wurde gehalten im

SOMMERSEMESTER 2007

von

NINA ZWEIG

und

PROF. DR. MICHAEL KAUFMANN

Diese Vorlesungsmitschrift wurde angefertigt von:

TILL HELGE HELWIG

Lektorat: Nina Zweig

Diese Mitschrift erhebt keinen Anspruch auf Vollständigkeit. Ich bemühe mich zwar darum, aber im Endeffekt sind es meine persönlichen Notizen aus der Vorlesung und kein offizielles Skript. Verbesserungsvorschläge nehme ich gerne entgegen und freue mich natürlich auch, wenn jemand die Mitschrift gebrauchen kann und als Gegenleistung vielleicht korrekturliest.

1 Worum geht es?

1. Algorithmendesign

- von der alltagssprachlichen Fragestellung zum mathematisch formalen Problem
- Formulierung des Lösungswegs
- Korrektheit
- Analyse
 - des Laufzeitverhaltens und
 - des Speicherbedarfs

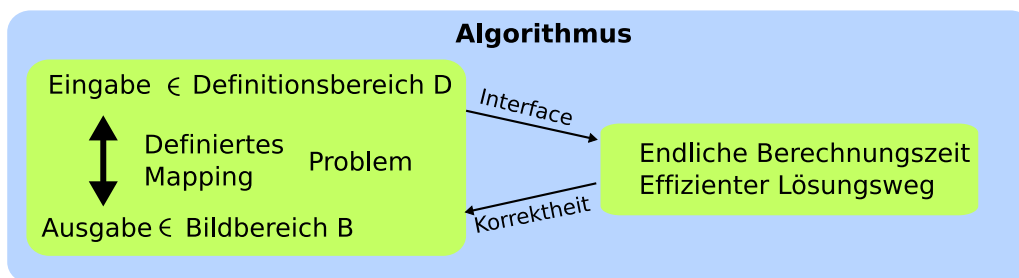
2. Implementierung

- Datenstrukturen
- Java, OOP

3. Experimentelle Analyse als Teil des "Algorithm Engineering"

2 Was ist ein Algorithmus?

Nach Donald E. Knuth:



Instanz: konkrete Eingabe $I \in D$ für einen Algorithmus

3 Analyse von Algorithmen

- Laufzeit
- Speicherverbrauch

Sei A ein Algorithmus mit D und B und $I \in D$ eine Instanz, dann kann die Laufzeit $T_A(I)$ berechnet bzw. gemessen werden.

3.1 Allgemeine Komplexität

1. worst-case-Komplexität:

$$T_{A,wc}(n) = \max_{I \in D} \{T_A(I) \mid |I| = n\}$$

Was ist "n"?

Charakteristische Größe einer Eingabe, z.B. Anzahl der Elemente einer Menge, Größe eines Arrays, etc.

2. average-case-Komplexität:

$$T_{A,avg}(n) = \frac{1}{|\{I \mid |I|=n\}|} \sum_{\substack{I \in D \\ |I|=n}} T_A(I)$$

3.2 Asymptotische Notation

- Implementierung
- Compiler/Interpreter
- Maschinenbefehlssatz

Asymptotische Notation als Betrachtung des Funktionsverhaltens für große n . Dazu: Unterteilung des Algorithmus in elementare Operationen: solche mit “konstanter” Laufzeit, z.B. Zuweisungen, kleinere Additionen, Inkrementieren, ... Diese haben die Laufzeit $\mathcal{O}(1)$.

3.3 \mathcal{O} -Notation

“Groß-O” von: $\mathcal{O}(f(n))$:= Menge von Funktionen $g(n)$, die sich asymptotisch “gleich” verhalten, im Sinne von:

$$\exists c > 0, \exists n_0, \forall n \geq n_0: g(n) \leq c \cdot f(n)$$

Beispiel: $g(n) = 5n^2 + 13n + 5$

$g(n) \in \mathcal{O}(n^2)$?

Ja: $f(n) = n^2, \exists c : c = 13 \Rightarrow n \geq 3 \Rightarrow g(n) \leq 13n^2$

Anschaulich: \mathcal{O} -Notation beschreibt “obere Schranke”.

3.4 \mathcal{o} -Notation

“klein-o” von: $\mathcal{o}(n)$ ist eine Menge von Funktionen $g(n)$ mit:

$$\forall c > 0, \exists n_0 > 0, \forall n \geq n_0 : g(n) \leq c \cdot f(n)$$

Beispiel: Ist $g(n) \in \mathcal{O}(n^2)$?

Nein: $c = 1 \nexists n_0$, so dass $\forall n \geq n_0 : g(n) \leq c \cdot f(n)$

aber: $g(n) \in \mathcal{O}(n^3)$

Anschaulich: \mathcal{O} -Notation beschreibt die “echt größere Schranke”.

3.5 Ω -Notation

“Groß-Omega”: $\exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : g(n) \geq c \cdot f(n)$ “untere Schranke”

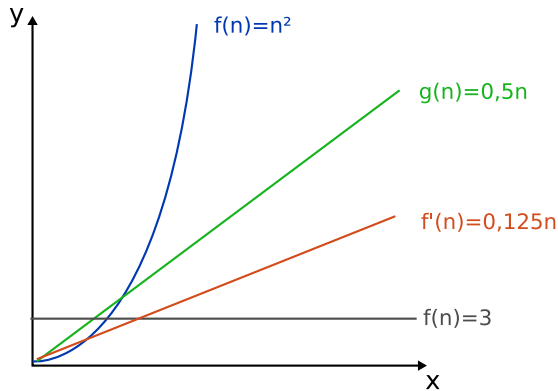
3.6 ω -Notation

“klein-omega”: $\forall c > 0, \exists n_0 > 0, \forall n \geq n_0 : g(n) \geq c \cdot f(n)$ “echt kleinere untere Schranke”

3.7 Θ -Notation

“Theta”: Schnittmenge $\Theta(f(n)) = \mathcal{O}(f(n)) \cap \Omega(f(n))$

Beispiel:



$$g(n) \in O(f(n))$$

$$g(n) \in \Theta(f'(n)) \Leftrightarrow g(n) \in O(f'(n)) \wedge g(n) \in \Omega(f'(n))$$

$$g(n) \in \omega(f''(n))$$

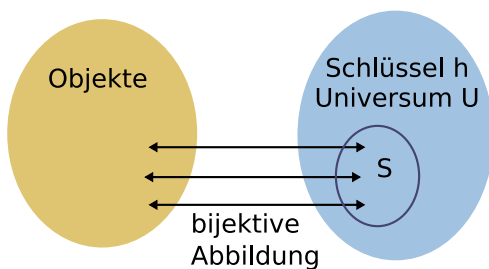
Anmerkung: Schreibweise oft: $g(n) = O(n^2)$

Zurückkommend auf die Notation: $O(1) \hat{=}$ konstante Laufzeit bedeutet, dass wir die genaue Anzahl von Rechenschritten nicht kennen (müssen), solange wir annehmen können, dass sie auf jedem System in "kleiner", von der Größe des Operanden unabhängige Anzahl von Schritten berechnet werden.

Beispiel: Gleichheit von zwei Objekten
 \Rightarrow Zeigervergleich \Rightarrow elementare Operation

Schwierig: Addition von Integern
 \Rightarrow bis zu einer natürlichen Grenze elementar \Rightarrow danach nicht mehr

4 Suchen



Object
-key
-info

Eingabe: Datenstruktur d mit Menge von Objekten O , deren Menge von Schlüsseln S und ein Schlüssel $h \in U$.

Ausgabe: Objekt $o \in d$ mit $o.key == k$, wenn $o \notin d$ Rückgabe 'null'.

Konvention: $d.get(i)$ gibt i -tes Element aus d zurück (auch für Arrays - normal $A[i]$).

Annahme: Elemente der Menge S im Rechner darstellbar $\Rightarrow \exists f : S \Rightarrow \mathbb{N}, \mathbb{R}$.
z.B.

Universum U im Folgenden \mathbb{N} . Wir benutzen Datenstruktur d mit folgenden Operationen:

- Initialisieren
- Einfügen
- Suchen

Einfache Datenstrukturen: Liste, Array

4.1 Lineare Suche

Sei $U = \{1, \dots, n\}$, gesucht ist $k \in U$.

```
Object o = null;
for Object object in d do
    if object.key == k then
        o = object;
        break;
return o;
```

Zeile 2 in Java 1.4 (Objekte mit Interface List):

```
for (Iterator it = d.iterator(); it.hasNext(); ) {
    Object object = it.next();
    ...
}
```

Finite Berechnung: \checkmark

Korrektheit: Schleifeninvariante im k -ten Durchlauf: o zeigt auf null, wenn Objekt bisher nicht gefunden, sonst auf Objekt.

Laufzeitanalyse: worst case

Zeilen 1,3-6: konstant

Zeile 2: $\mathcal{O}(n)$

\Rightarrow Gesamtlaufzeit: $\mathcal{O}(n)$

Laufzeitanalyse: average case

Annahme, dass $k \in S$ gleichverteilt \Rightarrow jeder Schlüssel hat gleiche Wahrscheinlichkeit, gesucht zu werden.

$$T_{\text{lin. Such, avg}}(n) = \frac{1}{n} \sum_{i=1}^n i = \frac{n(n+1)}{2}$$

Typische Falle beim Programmieren

```
for (int i = 0; i < d.size(); ++i)
    Object object = d.get(i);
```

Wenn d ist Liste, $d.get(i) = \mathcal{O}(n) \Rightarrow$ worst-case-Laufzeit: $\mathcal{O}(n^2)$

4.2 Binäre Suche

Eingabe: Datenstruktur d enthält Menge von Objekten O mit Schlüsselmenge S , $k \in S$.

Zusicherung: Elemente in d nach ihrem Schlüssel (aufsteigend) sortiert.

Ausgabe: siehe lineare Suche

Idee: Größe der Datenstruktur (des Suchraumes) in jedem Schritt halbieren.

```

1 Object binarySearch(DataStructure d, int k)


---


2 if (d.size() == 1)
3   return (d.get(0).key==k ? d.get(0) : null);
           Bedingung      true      false
4 if (d.size() == 0)
5   return null;
6 int probeIndex = d.size() / 2;
7 Object probe = d.get(probeIndex);
8 if (probe.key == k)
9   return probe;
10 else
11   return (probe.key > k ? // k liegt "vor" probe
12     binarySearch(d.firstHalf(), k) :
13     binarySearch(d.secondHalf(), k));

```

Finitheit: Datenstrukturgröße wird in jedem Schritt halbiert. Für $d.size() \in \{0, 1\}$ wird Rückgabe erzwungen. \checkmark

Korrektheit: Durch Sortierung kann k nur in der beibehaltenen Hälfte liegen.

Laufzeitanalyse:

Zeilen 1-6: $\mathcal{O}(1)$

Zeile 7: $\mathcal{O}(1)$ für Array
 $\mathcal{O}(n)$ für Liste

Zeilen 11-13: "Durchschneiden"
 $\mathcal{O}(n)$ für Liste (zum Element laufen, durchschneiden)
 $\mathcal{O}(n)$ für Array (Kopieren)

$\Rightarrow \log n$ rekursive Aufrufe
 \Rightarrow jeder Aufruf hat $\mathcal{O}(n)$
 $\Rightarrow \mathcal{O}(n \log n)$

Verbesserung: Suchraum durch zwei zusätzliche Variablen low , $high$ einschränken.

```
Object binarySearch(DataStructure d, int k, int l, int h)
```

```
// Annahme: Beim ersten Aufruf l=0, h=d.size()-1
if ((h-1) == 1)
    return (d.get(0).key == k ? d.get(0) : null);
if ((h-1) == 0)
    return null;
int probeIndex = 1 + (h-1+1) / 2;
Object probe = d.get(probeIndex);
if (probe.key == k)
    return probe;
else
    return (probe.key > k ?
            binarySearch(d, k, l, probeIndex - 1) :
            binarySearch(d, k, probeIndex + 1, h));
```

Laufzeitanalyse:Zeilen 1-3,5-7: $\mathcal{O}(1)$ Zeile 4: $\mathcal{O}(1)$ für Array
 $\mathcal{O}(n)$ für ListeZeile 8: $\mathcal{O}(1)$ ⇒ Gesamtlaufzeit: $\mathcal{O}(n \log n)$ für Liste, $\mathcal{O}(\log n)$ für Array**Alternative:** Laufzeit als Rekursion

$$T(n) = \underbrace{c_1}_{\approx \text{Zeilen 1-7}} + \underbrace{T\left(\frac{n}{2}\right)}_{\text{Zeile 8}}, T(1) = T(0) = c_2$$

Geratene geschlossene Form:

$$T(n) = c_1 \cdot \log n + c_2$$

Induktionsbeweis:zu zeigen: $T(n) = c_1 + T\left(\frac{n}{2}\right)$ mit $T(1) = c_2 \Leftrightarrow T(n) = c_1 \cdot \log n + c_2$ Induktionsanfang: $T(1) = c_2 = c_1 \cdot 0 + c_2 \checkmark$ Induktionsannahme: Sei $T(k) = c_1 \cdot \log k + c_2 \forall k \leq n, k, n \in \mathbb{N}$ Induktionsschritt: Sei $n' = 2k$ für irgendein $k \leq n$.

$$\begin{aligned} T(n') &= c_1 + T\left(\frac{n'}{2}\right) && \text{Rekursionsgleichung} \\ &= c_1 + T(k) \\ &= c_1 + c_1 \cdot \log k + c_2 && \text{Induktionsannahme} \\ &= c_1(\log(2k)) + c_2 \\ &= c_1 \cdot \log n + c_2 \end{aligned}$$

■

Laufzeit: average case

Im Allgemeinen: In Schritt i können genau 2^{i-1} Elemente gefunden werden.

Annahme: $n = 2^p - 1, p \in \mathbb{N}$

$$T_{\text{bin.Such,avg}}(n) = \frac{1}{n} \sum_{i=1}^p i \cdot 2^{i-1} = \frac{1}{2n} \sum_{i=1}^p i \cdot 2^i = \frac{1}{2n} \sum_{i=0}^p i \cdot 2^i$$

$$S_p := \frac{1}{2n} \sum_{i=0}^p i \cdot 2^i$$

$$S_p + (p+1) \cdot 2^{p+1} = \sum_{i=0}^p (i+1) \cdot 2^{i+1} = \underbrace{\sum_{i=0}^p i \cdot 2^{i+1}}_{2 \cdot S_p} + \underbrace{\sum_{i=0}^p 2^{i+1}}_{2^{p+2}-2}$$

$$\Rightarrow S_p = 2^{p+1}(p-1) + 2$$

Zurück zu T :

$$\begin{aligned} T_{\text{bin.Such,avg}}(n) &= \frac{1}{2n} ((p-1) \cdot 2^{p+1} + 2) && n = 2^p - 1, p \approx \log n \\ &\approx \frac{1}{n} (n \cdot \log n - n) \\ &= \log n - 1 \end{aligned}$$

Im Durchschnitt nur einen Aufruf weniger als im schlechtesten Fall.

4.3 Interpolationssuche

Motivation: Sie suchen "Laubfrosch" und schlagen den Duden bei "Paradoxon" auf Seite 800 auf.

$L = 12$... Buchstabe

$P = 16$... Buchstabe

Sie suchen weiter auf Seite $\frac{12}{16} \cdot 800 = 600$

Allgemein: Lineare Interpolationssuche

$$probeIndex = \left\lfloor \frac{k-d.get(l).key}{\Delta} \cdot (k-l+1) \right\rfloor + l$$

k ... gesuchter Schlüssel

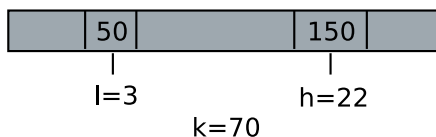
l ... "lower bound"

h ... "higher bound"

Δ ... noch "offener" Suchraum der Schlüssel

$\Delta : g.get(h).key - d.get(l).key$

Beispiel: $\left\lfloor \frac{70-50}{150-50} \cdot (22-3+1) \right\rfloor + 3 = 8$



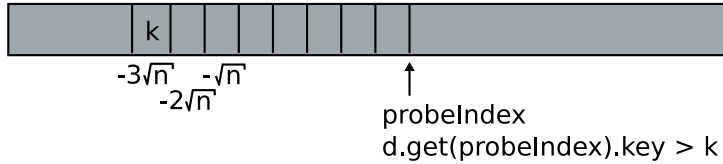
Laufzeitanalyse unter der Annahme der Gleichverteilung: Wir zeigen, dass $T_{\text{Interpol,avg}} = \mathcal{O}(\log \log n)$.

Interpol: Leichte Variante der Interpolationssuche.

Idee: Suche Objekt an der Stelle $probeIndex$.

Suche sukzessive an

$$\begin{aligned} & probeIndex + i \cdot \sqrt{n} \quad , \quad \text{wenn } k \text{ größer und} \\ & probeIndex - i \cdot \sqrt{n} \quad , \quad \text{wenn } k \text{ kleiner} \end{aligned}$$



Frage: Wie oft müssen wir i inkrementieren (dekrementieren) bis der Suchraum auf \sqrt{n} eingegrenzt ist.
 \Rightarrow Mindestens zwei Vergleiche an Stelle $probeIndex, probeIndex + \sqrt{n}$.

Wie hoch ist der Erwartungswert für die Anzahl C von nötigen Vergleichen?

$$\begin{aligned} C &= \sum_{i \geq 1} i \cdot Pr[\text{genau } i \text{ Vergleiche nötig}] \\ &= \sum_{i \geq 1} Pr[\text{mindestens } i \text{ Vergleiche nötig}] \end{aligned}$$

Würfel:

$$\frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 2 + \frac{1}{6} \cdot 3 + \frac{1}{6} \cdot 4 + \frac{1}{6} \cdot 5 + \frac{1}{6} \cdot 6 = 3,5$$

$1 + \frac{5}{6} + \frac{4}{6} + \frac{3}{6} + \frac{2}{6} + \frac{1}{6} \dots$ Wahrscheinlichkeit, dass mindestens eines 1 gewürfelt wird.

Wann sind mehr als 2 Vergleiche nötig?

Wenn die tatsächliche Position x von k von der erwarteten Position $\mu := probeIndex$ um mehr als \sqrt{n} abweicht.

Wo liegt μ ? Gleichverteilung

\Rightarrow Jeder Schlüssel innerhalb von Δ hat dieselbe Wahrscheinlichkeit $p = \frac{k-d.get(l).key}{\Delta}$, dass er kleiner ist als k .

Damit ist die Wahrscheinlichkeit, dass von $h-l+1 = n$ Elementen, die wir aus Δ ziehen, genau j kleiner sind, $\binom{n}{j} p^j (1-p)^{n-j}$.

Somit

$$\begin{aligned} \mu &= \sum_{j=1}^n j \binom{n}{j} p^j (1-p)^{n-j} = pn \\ \sigma^2 &= p(1-p)n \end{aligned}$$

Die Wahrscheinlichkeit, dass der Wert einer Zufallsvariablen um mehr als t von μ abweicht:

$$\begin{aligned} Pr[|x - \mu| \geq t] &\leq \frac{\sigma^2}{t^2} \quad \text{Chebyshev} \\ &\leq \frac{p(1-p)n}{(j-2)^2 n} \quad t := (j-2) \cdot \sqrt{n} \end{aligned}$$

in unserem Fall: $Pr[|x - pn| \geq (j-2) \cdot \sqrt{n}]$

Wegen $p(1-p) \leq \frac{1}{4} \Rightarrow C \leq 2 + \sum_{i \geq 3} \frac{1}{4} \left(\frac{1}{(i-2)^2} \right) = 2,4$

$T(n) = C + T(\sqrt{n})$

$T(1) = \mathcal{O}(1)$

$T(n) \leq 2,4 \log \log n$

Rekursionen

Beispiel:

$$\begin{aligned}
 T(n) &= T(\sqrt{n}) + c \\
 &= T(\sqrt{\sqrt{n}}) + c + c \\
 &= T(n^{\frac{1}{2}}) + 2c \\
 &\vdots \\
 \textit{i-ter Schritt} &= T(n^{\frac{1}{2^i}}) + i \cdot c \\
 &= T(n^{\frac{1}{2^{i+1}}}) + (i+1) \cdot c \\
 &\vdots \\
 \textit{nach x Schritten} &= \underbrace{T(2)}_{=\mathcal{O}(1)} + x \cdot c
 \end{aligned}$$

$$\begin{aligned}
 n^{\frac{1}{2^x}} &= 2 \\
 \Leftrightarrow \log(n^{\frac{1}{2^x}}) &= \frac{1}{2^x} \cdot \log n = 1 \\
 \log n &= 2^x \Rightarrow \log \log n = x
 \end{aligned}$$

$$\begin{aligned}
 T(n) &= (x+1) \cdot c = 2,4 \log \log n \\
 T(n) &= 5 \cdot T\left(\frac{9}{10}n\right) + n \\
 &= \mathcal{O}(n^k) \quad k \textit{ konst.}
 \end{aligned}$$

Master-Theorem

Löse $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ mit $a, b \geq 1$ und $f(n) > 0$.

Satz: Sei $a \geq 1, b > 1$ konstant und $f(n), T(n)$ nicht negativ mit $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ (wobei $\frac{n}{b}$ für $\lceil \frac{n}{b} \rceil$ bzw. $\lfloor \frac{n}{b} \rfloor$ steht).

1. Für $f(n) = \mathcal{O}(n^{(\log_b a) - \epsilon})$, $\epsilon > 0$ ist $T(n) = \Theta(n^{\log_b a})$.
2. Für $f(n) = \Theta(n^{\log_b a})$ ist $T(n) = \Theta(n^{\log_b a} \cdot \log_b n)$.
3. Für $f(n) = \Omega(n^{\log_b a + \epsilon})$, $\epsilon > 0$ und $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$ für $c < 1 \Rightarrow T(n) = \Theta(f(n))$.

Beispiele:

- $T(n) = 9 \cdot T\left(\frac{n}{3}\right) + n$
 $a = 9, b = 3$
 $f(n) = n^{\log_3 9 - 1} = n^{2-1} = n$
 \Rightarrow **Fall 1:** $T(n) = \mathcal{O}(n^{\log_3 9}) = \mathcal{O}(n^2)$
- $T(n) = T\left(\frac{n}{2}\right) + 1$ (binäre Suche)
 $a = 1, b = 2$
 $f(n) = 1 = n^{\log_2 1} = n^0 = 1$
 \Rightarrow **Fall 2:** $T(n) = \mathcal{O}(1 \cdot \log_2 n) = \mathcal{O}(\log n)$

- $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n \log n$ ("divide and conquer")
 $a = 2 = b$
 $f(n) = n^{\log_2 2 - \epsilon}$? **Geht nicht! Satz nicht anwendbar.**
 Fall 3 fordert $\Omega(n^{1+\epsilon})$, was durch $n \log n$ nicht erfüllt ist.

Beweis: Annahme: $n = b^i, i \in \mathbb{N}$
 Teilprobleme haben Größe $1, b, b^2, \dots, b^i$

Lemma: Sei $a > 1, b > 1, f(n)$ nicht negativ, $n = b^i$

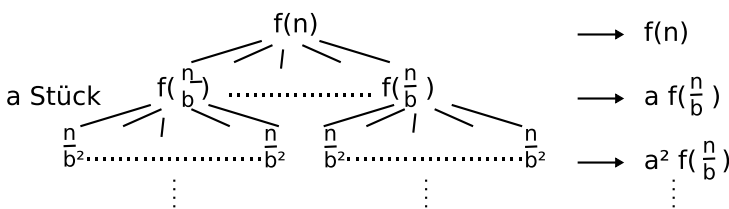
$$T(n) = \begin{cases} \mathcal{O}(1) & \text{für } n = 1 \\ a \cdot T\left(\frac{n}{b}\right) + f(n) & \text{für } n = b \end{cases}$$

dann gilt:

$$T(n) = \Theta\left(n^{\log_b a} + \sum_{j=0}^{\log_b n - 1} a^j \cdot f\left(\frac{n}{b^j}\right)\right)$$

Beweis:

$$\begin{aligned} T(n) &= f(n) + a \cdot T\left(\frac{n}{b}\right) \\ &= f(n) + a \left(f\left(\frac{n}{b}\right) + a \cdot T\left(\frac{n}{b^2}\right) \right) \\ &= f(n) + a \cdot f\left(\frac{n}{b}\right) + a^2 \cdot T\left(\frac{n}{b^2}\right) \\ &= f(n) + a \cdot f\left(\frac{n}{b}\right) + a^2 \cdot f\left(\frac{n}{b^2}\right) + a^3 \cdot T\left(\frac{n}{b^3}\right) \\ &\vdots \\ &= \sum_{j=0}^{\log_b n - 1} a_j \cdot f\left(\frac{n}{b^j}\right) + \Theta\left(n^{\log_b a}\right) \end{aligned}$$



$$\begin{aligned} &= a^{\log_b a} \cdot \mathcal{O}(1) + \sum_{j=0}^{\log_b n - 1} a_j \cdot f\left(\frac{n}{b^j}\right) \\ &= n^{\log_b a} \cdot \mathcal{O}(1) + \sum_{j=0}^{\log_b n - 1} a_j \cdot f\left(\frac{n}{b^j}\right) \end{aligned}$$

Lemma: Sei $a \geq 1, b > 1, f(n)$ definiert auf $b^i, i \geq 0$. Sei $g(n)$ definiert als $g(n) = \sum_{j=0}^{\log_b n - 1} a^j \cdot f\left(\frac{n}{b^j}\right)$.

1. Ist $f(n) = \mathcal{O}(n^{(\log_b a) - \epsilon})$, so ist $g(n) = \mathcal{O}(n^{\log_b a})$

Beweis:

Mit $f(n) = \left(\sum_{j=0}^{\log_b n - 1} \frac{1}{b^j}\right)^{\log_b n - 1}$ ist $f\left(\frac{n}{b^j}\right) = \left(\sum_{l=0}^{\log_b \frac{n}{b^j} - 1} \frac{1}{b^l}\right)^{\log_b \frac{n}{b^j} - 1}$.

$$\begin{aligned}
 g(n) &= \mathcal{O}\left(\sum_{j=0}^{\log_b n - 1} a^j \cdot \left(\frac{n}{b^j}\right)^{\log_b n - 1}\right)^{\log_b a - \epsilon} \\
 &= \mathcal{O}\left(n^{\log_b a - \epsilon} \cdot \sum_{j=0}^{\log_b n - 1} \left(\frac{a \cdot b^\epsilon}{b^{\log_b a}}\right)^j\right) \\
 &= \mathcal{O}\left(n^{\log_b a - \epsilon} \sum_{j=0}^{\log_b n - 1} (b^\epsilon)^j\right) \\
 &= \mathcal{O}\left(n^{\log_b a - \epsilon} \frac{b^{\epsilon \cdot \log_b n} - 1}{b^\epsilon - 1}\right) \\
 &= \mathcal{O}\left(n^{\log_b a - \epsilon} \cdot \frac{n^\epsilon - 1}{b^\epsilon - 1}\right) \\
 &\leq \mathcal{O}(n^{\log_b a - \epsilon} \cdot n^\epsilon) \\
 &= \mathcal{O}(n^{\log_b a})
 \end{aligned}$$

■

2. Mit $f(n) = \Theta(n^{\log_b a})$ gilt $g(n) = \Theta(n^{\log_b a} \cdot \log_b n)$.

Beweis:

Mit $f(n) = \Theta(n^{\log_b a})$ ist $f\left(\frac{n}{b^j}\right) = \Theta\left(\left(\frac{n}{b^j}\right)^{\log_b a}\right)$.

$$\begin{aligned}
 g(n) &= \Theta\left(\sum_{j=0}^{\log_b n - 1} a^j \cdot \left(\frac{n}{b^j}\right)^{\log_b a}\right) \\
 &= \Theta\left(n^{\log_b a} \cdot \sum_{j=0}^{\log_b n - 1} a^j \cdot \left(\frac{1}{b^{\log_b a}}\right)^j\right) \\
 &= \Theta\left(n^{\log_b a} \cdot \sum_{j=0}^{\log_b n - 1} a^j \cdot \left(\frac{1}{a}\right)^j\right) \\
 &= \Theta\left(n^{\log_b a} \cdot \sum_{j=0}^{\log_b n - 1} 1\right) \\
 &= \Theta(n^{\log_b a} \cdot \log_b n)
 \end{aligned}$$

■

3. Ist $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$, $c < 1$, $n > b$, so ist $g(n) = \Theta(f(n))$.

Beweis:

Da $g(n) = \sum_{j=0}^{\log_b n - 1} a^j \cdot f\left(\frac{n}{b^j}\right)$ ist $g(n) \geq f(n)$.

Noch zu zeigen: $g(n) = \mathcal{O}(f(n))$

Wegen $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$ für $c < 1$ gilt $a^j \cdot f\left(\frac{n}{b^j}\right) \leq c^j \cdot f(n)$

$$\Rightarrow g(n) \leq \sum_{j=0}^{\log_b n - 1} c^j \cdot f(n) = f(n) \cdot \sum_{j=0}^{\log_b n - 1} c^j = f(n) \cdot \frac{c^{\log_b n} - 1}{c - 1} \leq \mathcal{O}(f(n))$$

■

Es gilt $a^{\log_b n} = n^{\log_b a}$, denn $n^{\log_b a} = b^{\log_b n \cdot \log_b a} = a^{\log_b n}$.

5 Sortieren

Eingabe: Menge O von Objekten, auf denen eine Ordnung definiert ist, so dass für je zwei Objekte $o_1, o_2 \in O$ jeweils entweder

- a) $o_1 < o_2$
- b) $o_1 == o_2$
- c) $o_1 > o_2$ gilt.

Ausgabe: Die selbe Menge sortiert, so dass für alle Objekte o_i ($:= o_i$ steht an Position i) und o_j mit $0 \leq i < j < |O|$ gilt $o_i \leq o_j$.

Im Folgenden gehen wir von zu sortierendem Array S mit natürlichen Zahlen aus.

5.1 Sortieren durch Einfügen

Idee: Zweites S' mit der gleichen Länge $n := |S|$. Suche in Runde $i \geq 0$ die kleinste Zahl aus S , streiche sie in S , füge sie an Position i in S' ein.

Streichen implementieren

- durch eine negative Zahl (falls Eingabe nur positiv)
- durch *null*-Objekt
- durch zusätzliches *boolean* Array

Fintheit \checkmark , Korrektheit \checkmark

Laufzeitanalyse: n Runden mit jeweils $\mathcal{O}(n)$ Laufzeit, $\sum_{i=0}^{n-1} n - i = \frac{n \cdot (n-1)}{2}$

Schlechte Laufzeit, weil wir jedes Element aus S mit jedem anderen vergleichen.

Idee: Wähle sogenanntes Pivotelement p und bilde zwei Mengen

$$K = \{x \in S \mid x \leq p\}$$

$$G = \{x \in S \mid x > p\}$$

Wenn K und G sortiert sind, können wir S wie folgt zusammensetzen:

$$S = K[0] \dots K[|K| - 1] p G[0] \dots G[|G| - 1]$$

Sei $k := |K|$ und $g := |G|$

\Rightarrow Position von p im sortierten Array ist $S[k]$.

\Rightarrow Verfahren rekursiv auf die Teilmengen K und G anwenden. Wir führen Variablen *leftIndex*, *rightIndex* ein, um zu sortierende Teilbereiche zu kennzeichnen.

Es ist klar, dass alle Elemente aus K kleiner sind als alle Elemente aus G , d.h. wir sparen $k \cdot g$ Vergleiche.

5.2 QuickSort

Grobe Idee:

```

QuickSort (int[] s, int leftIndex, int rightIndex)


---


if (leftIndex < rightIndex)
    int indexOfPivotElement = partition(s, leftIndex, rightIndex);
    QuickSort (s, leftIndex, indexOfPivotElement - 1);
    QuickSort (s, indexOfPivotElement + 1, rightIndex);

```

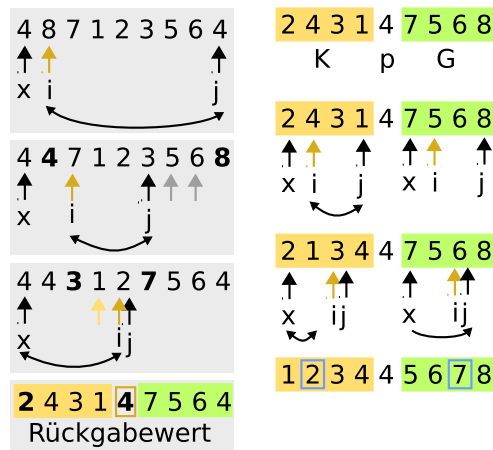
Was leistet Funktion *partition*?

- Wählt ein Pivotelement p aus. Hier $p := S[leftIndex]$
- Setzt alle Elemente kleiner oder gleich p in linke "Hälfte", größere in rechte "Hälfte" und p auf die Position dazwischen.
- Diese Position wird zurückgegeben.

Idee: Wir benutzen zwei Zeiger

$i := leftIndex + 1$

$j := rightIndex$



Das Verfahren teilt das Problem in Teilprobleme, "bezwingt" diese durch rekursive Anwendung und kombiniert diese Teillösungen dann zur Gesamtlösung.

⇒ Divide and Conquer Verfahren

Analyse: Allgemeine Rekursionsformel: $T(n) = \underbrace{\Theta(n)}_{Divide} + \underbrace{T(|K|) + T(|G|)}_{Conquer} + \underbrace{\mathcal{O}(1)}_{Combine}$

Worst-case Analyse: Pivotelement ist Minimum bzw. Maximum

⇒ entweder $|K| = 0$ oder $|G| = 0$

⇒ $T(n) = \Theta(n) + T(n - 1) + \mathcal{O}(1)$

⇒ $\mathcal{O}(n^2)$

⇒ genauso schlecht wie Sortieren durch Einfügen

Average-case Analyse: Annahme: Elemente sind paarweise verschieden. In jedem Teilproblem wählen wir zufällig ein Element aus allen als Pivotelement.

⇒ Die Wahrscheinlichkeit, das Element zu ziehen, das in der sortierten Folge an Position j steht, ist $\frac{1}{n}$ für alle Elemente des Teilproblems.

Sei $\overline{QS}(n)$ die erwartete Anzahl von Vergleichen im Feld der Größe n .

Klar: $\overline{QS}(0) = \overline{QS}(1) = 0$

Für $n \geq 2$: $\overline{QS}(n) = \underbrace{n}_{Divide} + E(\overline{QS}(j - 1) + \overline{QS}(n - j))$

Wenn das j -te Element der sortierten Folge als Pivotelement gewählt wird.

Wegen Linearität des Erwartungswertes:

$$\overline{QS}(n) = n + \frac{1}{n} \cdot \sum_{j=1}^n (\overline{QS}(j - 1) + \overline{QS}(n - j)) = n + \frac{2}{n} \sum_{j=1}^{n-1} \overline{QS}(j)$$

Trick:

$$n \cdot \overline{QS}(n) = n^2 + 2 \cdot \sum_{j=0}^{n-1} \overline{QS}(j) \tag{1}$$

$$(n+1) \cdot \overline{QS}(n+1) = (n+1)^2 + 2 \cdot \sum_{j=0}^n \overline{QS}(j) \tag{2}$$

2-1:

$$\begin{aligned} (n+1) \cdot \overline{QS}(n+1) - n \cdot \overline{QS}(n) &= (n+1)^2 - n^2 + 2 \cdot \overline{QS}(n) \\ \Leftrightarrow (n+1) \cdot \overline{QS}(n+1) &= n^2 + 2n + 1 - n^2 + \overline{QS}(n) \cdot (2+n) \\ \Rightarrow \overline{QS}(n+1) &\leq 2 + \frac{n+2}{n+1} \cdot \overline{QS}(n) \\ &\Rightarrow 2 + \frac{n+2}{n+1} \left(2 + \frac{n+1}{n} \cdot \overline{QS}(n-1) \right) \\ &= 2 + (n+2) \cdot \left(\frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \dots \right) \\ &= 2 + 2 \cdot (n+2) \cdot \underbrace{\left(\frac{1}{n+1} + \frac{1}{n} + \frac{1}{n-1} + \dots \right)}_{\text{harmonische Reihe}} \\ &= 2 + 2 \cdot (n+2) \cdot \sum_{i=1}^{n+1} \frac{1}{i} \\ &\leq 2 + 2 \cdot (n+2) \cdot (\ln n + 1) \\ &= \mathcal{O}(n \log n) \end{aligned}$$

Anmerkung: Annahme kann erzwungen werden durch

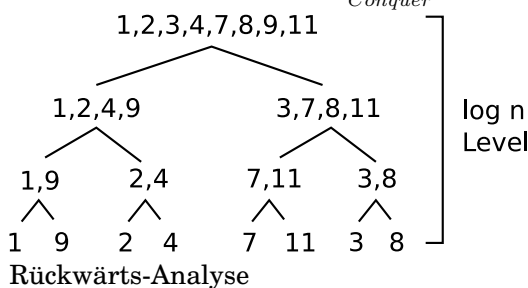
- Shuffling
- Wählen des Pivotelements "uniformly at random"

5.3 MergeSort

Idee: Teile zu sortierende Menge in zwei gleichgroße Hälften und sortiere diese (Divide & Conquer).

Kombiniere: $\begin{matrix} 1 & 7 & 9 & 11 \\ 2 & 3 & 4 & 8 \end{matrix} > 1\ 2\ 3\ 4\ 7\ 8\ 9\ 11$

Allgemeine Rekursion: $T(n) = \underbrace{\mathcal{O}(n)}_{\text{Divide}} + 2 \cdot \underbrace{T\left(\frac{n}{2}\right)}_{\text{Conquer}} + \underbrace{\Theta(n)}_{\text{Combine}}$



MergeSort hat die Laufzeit $\mathcal{O}(n \log n)$

Variante: *m*-Wege-Mischen

Füge immer *m* sortierte Teilmengen zusammen.

$\Rightarrow \Theta(n \log_m n)$

5.4 HeapSort

Heap: Ist ein binärer Baum mit Wurzel *r*. Formal ist ein Baum ein Graph, als solcher ist er ein Paar $T = (V, E)$
V ... Menge der Knoten
E $\subseteq V \times V$... Menge der Kanten (Relation)
 Kanten $e = (v, w)$ sind gerichtet. *v* ist Vater von *w*. *w* ist Kind von *v*.

Was ist ein Baum?

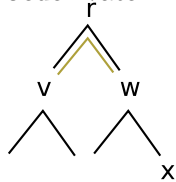
Ein zyklfreier Graph. (Hierfür betrachten wir Kanten als ungerichtet.)

Ein *Zykel* ist ein Pfad, der mindestens einen Knoten mehrfach durchläuft.

Ein *Pfad* ist eine Folge von Kanten $P(v, w) = \{(v, v_1), \dots, (v_{k-1}, w)\}$, so dass $(v, v_i) \in E, (v_{k-1}, w) \in E$ und $\forall 1 \leq i < k - 1 : (v_i, v_{i+1}) \in E$.

Binärer Baum:

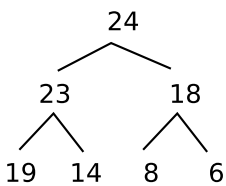
- Jeder Knoten außer der Wurzel hat einen Vater
- Jeder Vater hat höchstens zwei Kinder



Wurzel von T
P(v,w) Pfad
v,w ... Kinder von T
v ... linkes Kind von T
r ... Vater von *w*
 Ein Knoten ohne Kinder heißt Blatt

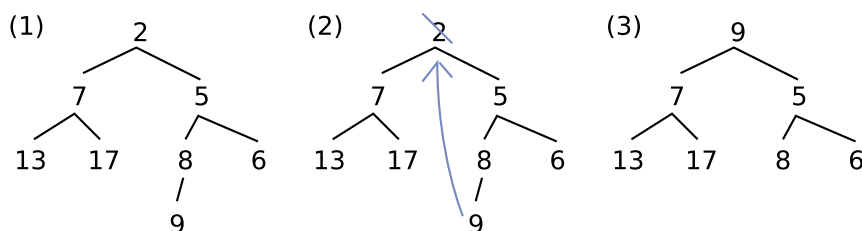
Idee beim HeapSort: Sei *S* eine Menge zu sortierender Zahlen. Die Knoten speichern die Zahlen. Baue Heap so auf, dass für alle Knoten gilt, dass die Kinder kleinere Werte haben.

\Rightarrow Heapeigenschaft

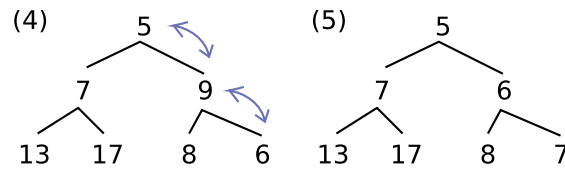


Wenn die Teilbäume gleich groß sind, muss man nur $\log n$ Vergleiche machen, bis man den richtigen Vater gefunden hat.

Beispiel: HeapSort



- Sortieren durch Minimum-Suche
- Minimum steht in Wurzel \Rightarrow Streichen und Heap neu aufbauen (2)
- Füge letztes Element im Baum an Stelle der Wurzel ein (3)
- Stelle Heapeigenschaft wieder her:



- Lasse 9 im Baum runtersinken (4)
- Falls $9 \geq \min(\text{key}(lchild), \text{key}(rchild))$, tausche 9 mit $\min(lsohn, rsohn)$ per Iteration (5) \Rightarrow Heapeigenschaft erhalten
- Iteration mit Minimumsuche! \checkmark

Korrektheit:

1. Im Heap steht in jeder Iteration das minimale Element oben. Dieses wird jeweils herausgeschrieben.
2. Durch Runtersinkenlassen wird Heapeigenschaft wiederhergestellt.

Laufzeit: $\mathcal{O}(n \cdot T(\text{Wiederherstellen der Heapeigenschaft}))$

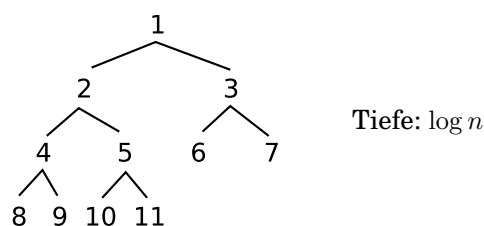
Ziel: $\mathcal{O}(\log n)$ für Wiederherstellen der Heapeigenschaft

1. Vergleiche des Schlüssels mit Schlüssel der Kinder und anschließender Tausch möglichst in $\mathcal{O}(1)$.
2. Schätze die Zahl der Iterationen beim Runtersinken mit dem Ziel $\mathcal{O}(\log n)$.

Benutze ausgewogene Bäume: Es gibt $k \geq 0$, so dass alle Blätter die Tiefe k oder $k + 1$ haben und Blätter auf Stufe $k + 1$ sitzen "ganz links".

Ausgewogene Bäume haben eine kurze und einfache **Darstellung als Array** mit folgenden Eigenschaften:

Für Knoten mit Index i hat $\text{parent}(v)$ den Index $\lfloor \frac{i}{2} \rfloor$, $\text{lchild}(v)$ den Index $2i$ und $\text{rchild}(v)$ den Index $2i + 1$.



Baum existiert nur implizit und wird explizit als Array dargestellt: 1. geht in $\mathcal{O}(1)$ und 2.?

Es fehlt: Initialisierung des Arrays: for all $x \in S$ do Insert(x, h)

```
Insert( $x, h$ )
```

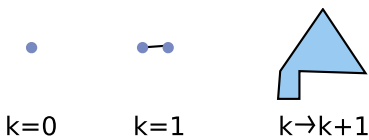
```
Sei  $n$  die Größe von  $h$ 
 $n \leftarrow n+1$ 
 $A(n) \leftarrow x$ 
 $finish \leftarrow (i=1)$ 
while not finish do
   $j \leftarrow \lfloor \frac{i}{2} \rfloor$ 
  if  $A(j) < A(i)$  then
    tausche( $A(i), A(j)$ )
     $i \leftarrow j$ 
  else
     $finish \leftarrow true$ 
od
```

Laufzeit:

- Initialisierung: $\mathcal{O}(n \cdot \text{Höhe}(h))$
- $n \cdot T(\text{Minimale Suche und Wiederherstellung}) = \mathcal{O}(n \cdot \text{Höhe}(h))$

Lemma: Hat ein ausgewogener Baum die Höhe k , so hat er mindestens 2^k Knoten.

Beweis: per Induktion über k



\Rightarrow Ein ausgewogener Baum mit n Knoten hat eine Höhe kleiner als $\log n$.

Satz: HeapSort geht in $\mathcal{O}(n \log n)$

5.5 BucketSort

Gegeben seien Wörter über einem Alphabet Σ und $|\Sigma| = m$.
Sortiere lexikographisch:

Ordnung: $a < b, a < aa < ab$

Fall 1: Wörter haben alle Länge 1.

Wir stellen m Fächer zur Verfügung (für a, b, c, \dots, z) und werfen die Wörter in die entsprechenden Fächer. Anschließend werden die Fächer konkateniert.

Sei $n = \#\text{Wörter}$, k die Länge ($k = 1$), dann ist die Laufzeit $\mathcal{O}(n + m)$.

Implementierung: Einzelne Fächer als lineare Listen und diese zusammen als Array.

Fall 2: Alle Wörter haben Länge k .

Sei ein Wort $a^i = a_1^i a_2^i \dots a_k^i$.

Idee: Sortiere zuerst nach dem letzten Zeichen, dann nach dem vorletzten usw. Damit sind Elemente, die zum Schluss ins gleiche Fach fallen, automatisch geordnet.
 $\Rightarrow \mathcal{O}((n + m)k)$

Beispiel: 124, 223, 324, 321, 123

$k = 3$

Fächer:	1	2	3	4
1	321 ₁		223 ₂ 123 ₃	124 ₄ 324 ₅
2		321 ₁ 223 ₂ 123 ₃ 124 ₄ 324 ₅		
3	123 ₁ 124 ₂	223 ₃	321 ₄ 324 ₅	

[Indizes zeigen die Reihenfolge des Aufsammelns.]

Problem: Aufsammeln der Listen.

\Rightarrow Überspringe leere Listen mit dem Ziel, eine Laufzeit von $\mathcal{O}(nk + m)$ statt $\mathcal{O}(nk + mk)$ zu erreichen.

Trick: Erzeuge Paare (j, x_j^i) mit $1 \leq i \leq n$ und $1 \leq j \leq k$. Sortiere nach der zweiten Komponente und dann nach der ersten. Dann liegen im j -ten Fach die Zeichen sortiert vor, die an der j -ten Stelle vorkommen.

Beispiel: Bilde (j, x_j) mit $1 \leq j \leq 3$

$j = 3$: (3, 1), (3, 3), (3, 3), (3, 4), (3, 4)

$j = 2$: (2, 2), (2, 2), (2, 2), (2, 2), (2, 2)

1	2	3	← 1. Komponente
1	2	1	
1	2	3	← springe zu Fach 3
2	2	3	
3	2	4	← springe zu Fach 4
3	2	4	

↑
2. Komponente

Dadurch wird Durchlaufen leerer Fächer vermieden.

Nach dem Preprocessing sortiere wie oben mit dem Wissen, welche Fächer in der j -ten Iteration ($1 \leq j \leq k$) leer bzw. nicht leer sind.

$\Rightarrow \mathcal{O}(nk + m)$ (Jedes Element wird k -mal angeschaut)

Fall 3: Wir nehmen an a^i hat Länge l_i und $L = \sum_{i=1}^n l_i$ mit dem Ziel $\mathcal{O}(L + m)$

Idee: Sortiere Wörter der Länge nach. Beziehe zuerst die langen Wörter in den Algorithmus ein.

1. Erzeuge Paare (l_i, a_i) .

2. Sortiere Paare durch BucketSort nach ihrer ersten Komponente.
Sei $L(k)$ die Liste der Wörter der Länge k .
3. Erzeuge L Paare (j, x_j^i) mit $1 \leq i \leq n$ und $1 \leq j \leq l_i$.
Sortiere zuerst nach der zweiten Komponente und dann nach der ersten. Damit erhalten wir lineare Listen, die nicht lineare Fächer angeben für die Iterationen $1 \leq j \leq l_{max}$.
4. Sortiere a_i durch BucketSort wie im Fall 2. Betrachte nur $L(k)$ -Listen.

Satz: BucketSort sortiert n Wörter der Gesamtlänge L über dem Alphabet Σ mit $|\Sigma| = m$ mit der Laufzeit $\mathcal{O}(L + m)$.

5.6 Auswahlproblem

Finde Median.

Gegeben ist eine Menge S von Zahlen und eine Zahl $1 \leq i \leq n$. Finde ein $x \in S$ mit $|\{y | y > x\}| = i$.

1. Idee: Sortiere und laufe dann die Menge ab bis zur i -ten Stelle. $\rightarrow \mathcal{O}(n \log n)$
2. Idee: Ähnlich wie QuickSort. Bei perfekter Teilung erhält man $n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \dots \leq 2n$
Also läuft es im Mittel in linearer Zeit, wenn p zufällig ist.

Deterministisch: $|S| = n$

1. Wir teilen n Elemente in 5-er Gruppen auf, erhalten also $\frac{n}{5}$ Gruppen. Bestimme den Median in jeder Gruppe.
2. Suche rekursiv den Median x der Mediane.
3. Teile die Menge S bezüglich x in Mengen S_1, S_2 auf. Sei $|S_1 \cup x| = k, |S_2| = n - k$.
4. Algorithmus:

```

if i=k then return x
else if i<k then Auswahl( $S_1$ , i)
else Auswahl( $S_2$ , i-k)

```

Wieso betrachten wir Mediane?

$\frac{1}{10}n$ der Mediane sind größer als x und $\frac{1}{10}n$ der Mediane sind kleiner als x . Die betreffenden Gruppen liefern mindestens 3 Elemente, die größer bzw. kleiner sind als x . Also hat S_1 mindestens $\frac{3}{10}n$ Element, genauso S_2 . Außerdem ist $|S_1| \leq \frac{7}{10}n$, genauso $|S_2|$.

$\Rightarrow T(n) = T(\frac{n}{5}) + T(\frac{7}{10}n) + \mathcal{O}(n)$ für $n \geq d$ konstant
 $T(n) = \mathcal{O}(1)$ für $n \leq d$.

Behauptung: $T(n) \leq c \cdot n$ für c konstant

$$\begin{aligned}
 T(n) &\leq c \cdot \frac{n}{5} + c \cdot \frac{7}{10}n + a \cdot n \\
 &\leq c \cdot \frac{9}{10}n + a \cdot n \\
 &\leq c \cdot n \quad \text{für } a \leq \frac{c}{10}
 \end{aligned}$$

■

```

public Object search (int key, Tree tree)
{
    Node currentEl = tree.getRoot();
    boolean found = false;
    int currentKey;
    while (currentEl != null && !found) {
        currentKey = currentEl.getObject().getKey();
        if (currentKey == key)
            found = true;
        else
            currentEl = currentKey > key ? currentEl.lchild() : currentEl.rchild();
    }
    if (currentEl == null)
        return null;
    return currentEl.getObject();
}

```

Satz: In Zeit $\mathcal{O}(n)$ können wir deterministisch das i -t größte Element aus einer Menge der Größe n finden.

Bemerkung: Wieso 5-er Gruppen und keine 3-er Gruppen, 7-er Gruppen, ...?

Wahl von a ? Wahl von c ?

$c = 10 \cdot a$, $a \cdot n = \frac{n}{5} \cdot \text{Median}(5 \text{ Elemente})$

Für 5 Elemente braucht man zur Berechnung des Medians $4 + 3 + 2 = 9$ Vergleiche.

6 Balancierte Bäume

Mit einem Heap kann man das minimale Element schnell finden und alle anderen Elemente in $\mathcal{O}(n)$. Wie kann man binäre Bäume so aufbauen, dass man alle Element in $\mathcal{O}(n \log n)$ finden kann?

6.1 Bäume

Bäume bestehen aus Knoten v , die auf andere Knoten zeigen können. In binären Bäumen zeigt jeder Knoten auf höchstens zwei andere Knoten, seine sogenannten Kinder. Wir untersuchen *lchild* (linkes Kind) und *rchild* (rechtes Kind). Der linke Teilbaum von v sind alle Knoten, die über $v.lchild$ erreicht werden können. Dies gilt für den rechten Teilbaum analog. Knoten speichern Objekte, die eindeutig mit einer Zahl $n \in \mathbb{N}$ als Schlüssel assoziiert sind. Man unterscheidet zwei Arten der Speicherung.

Knotenorientiert: Ein Objekt pro Knoten. Für alle v gilt, dass der Schlüssel im linken Teilbaum kleiner ist als der Schlüssel des Objekts v , der wiederum kleiner ist als der Schlüssel von allen Objekten im rechten Teilbaum.

Blattorientiert: Ein Objekt pro Blatt. Ein Schlüssel pro Nicht-Blatt ("innerer Knoten"). Für alle v gilt, dass der Schlüssel von Objekten im linken Teilbaum von v kleiner ist als der Schlüssel von v und wieder wiederum kleiner ist als die Schlüssel aller Objekte im rechten Teilbaum von v .

Hier: Knotenorientierte Speicherung. Wie können *Suche* und *Einfügen* implementiert werden?

Einfügen: Zuerst rufen wir *Suche*(v) auf und lassen uns den letzten besuchten Knoten zurückgeben. Der zuletzt besuchte Knoten hat nur ein Kind. \Rightarrow Füge Objekt als neues Kind ein.

Streiche(x) : Zuerst führen wir *Suche(x)* aus. Ist $x \in T$, endet die Suche in Knoten u mit dem zugehörigen Schlüssel x .

Fall 1: u ist ein Blatt \Rightarrow streiche u

Fall 2: u hat nur ein Kind $w \Rightarrow$ streiche u und setze w an die Stelle von u als Kind im Elternknoten v von u

Fall 3: u hat zwei Kinder \Rightarrow suche den Knoten v mit dem größten Schlüssel im linken Teilbaum (einmal links, dann rechts, bis es nicht mehr geht $\Rightarrow v$) $\Rightarrow v$ hat höchstens ein Kind. Tausche die Objekte von u und v . Lösche dann Knoten mit Fall 1 oder 2.

Komplexität der Operationen: $\mathcal{O}(h)$, wobei h die Höhe des Baumes ist (entspricht also dem längsten Pfad von der Wurzel zu einem Blatt).

Problem: Höhe kann $\mathcal{O}(n)$ sein.

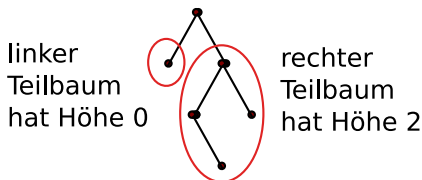
Idee:

1. Baue Baum von Zeit zu Zeit wieder ausgewogen auf
2. Balancierte Bäume

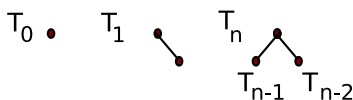
6.2 AVL-Bäume

Definition: Sei u ein Knoten in einem binären Baum. $Balance(u)$ von u bezeichnet die Differenz der Höhen von rechtem und linkem Teilbaum.

Balance = 2 - 0 = 2



Definition: Ein binärer Baum heißt *AVL-Baum*, falls für alle Knoten $v \in V$ $|Balance(v)| \leq 1$ gilt. Wir wollen nun zeigen, dass die Höhe in einem solchen Baum $\mathcal{O}(n \log n)$ ist. Dazu definieren wir **FIBONACCI-Bäume** rekursiv wie folgt:



FIBONACCI-Zahlen sind ebenfalls rekursiv definiert: $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2} \forall n \geq 2$

Folgerung: T_n enthält F_n Blätter.

Behauptung: AVL-Baum der Höhe h enthält mindestens F_n Blätter.

$n = 0 \checkmark$

$n = 1:$

$n \geq 2:$ Man erhält blattärmsten AVL-Baum, wenn m an jeweils den blattärmsten AVL-Baum der Höhe $h - 1$ mit dem blattärmsten AVL-Baum der Höhe $h - 2$ kombiniert.

Nach Induktionsannahme hat dieser Baum mindestens $F_{n-1} + F_{n-2} = F_n$ Blätter. ■

Bekannt: Für $h \geq 0$ gilt

$$F_h = \frac{\alpha^h - \beta^h}{\sqrt{5}}$$

$$\alpha = \frac{1 + \sqrt{5}}{2} \approx 1,6$$

$$\beta = \frac{1 - \sqrt{5}}{2} \approx -0,6$$

Lemma: Ein AVL-Baum mit n Knoten hat die Höhe $\mathcal{O}(\log n)$.

Beweis: Der Baum hat höchstens n Blätter.

$$n \geq \text{Blätter} \geq F_n$$

Wenn h groß genug gewählt wird, erhält man:

$$F_h \geq \frac{\alpha^h}{2\sqrt{5}}$$

$$n \geq \frac{\alpha^h}{2\sqrt{5}}$$

$$\Rightarrow h = \frac{\log(2\sqrt{5} \cdot n)}{\alpha}$$

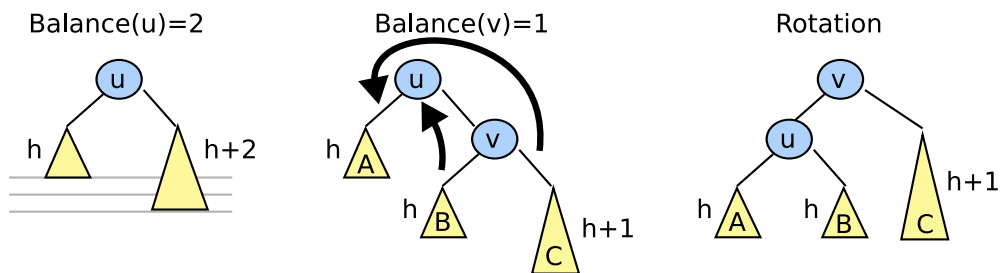
$$\Rightarrow h \in \mathcal{O}(\log n)$$

Wie geht jetzt das Einfügen und Streichen?

Zusätzlich assoziiert zu Knoten ist seine Balance.

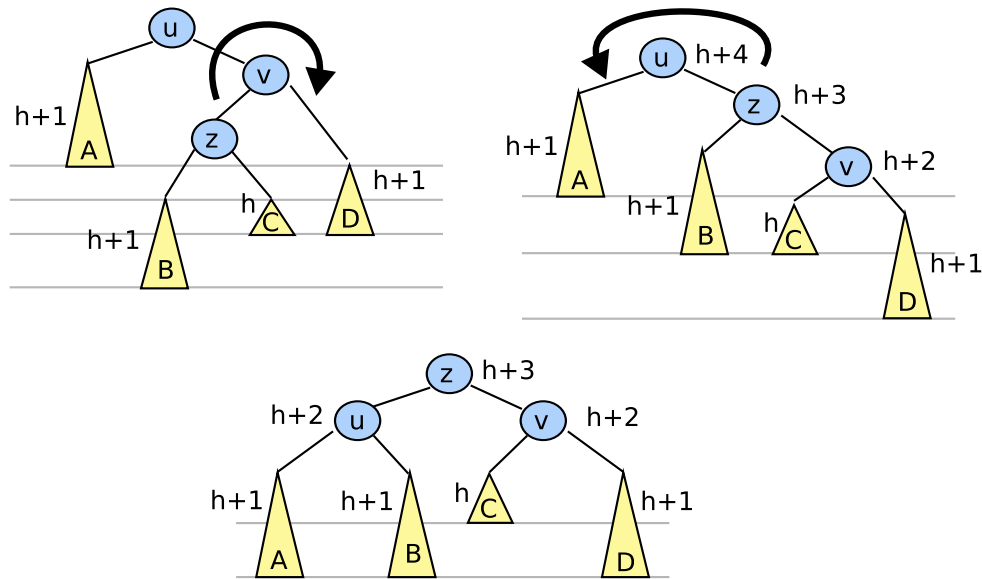
Einfügen: Durch das Einfügen eines Knotens w können die Balancen einiger Vorfahren die Werte ± 2 annehmen.

Sei u der *tiefste* dieser Vorfahren und sei ohne Beschränkung der Allgemeinheit $Balance(u) = 2$, so wurde w im rechten Teilbaum eingefügt. Sei nun v das rechte Kind von u , dann muss sich die Höhe zusätzlich erhöht haben, also ist die Balance von $v = \pm 1$.



Beachte:

1. Rotation bewahrt die Suchbaumeigenschaft.
2. u und v haben nach der Rotation eine Balance von 0
3. Alle Knoten in A , B und C behalten ihre (legale) Balance
4. Die Höhe von v ist nach der Rotation gleich der Höhe von u vor dem Einfügen von w . Also sind nach der Rotation alle Vorfahren von w balanciert.



Zusammenfassung: $Balance(u) = H(rTB(v)) - H(lTB(v))$, wobei $H(t)$ die Höhe des Baums t ist und $rTB(t)$ bzw. $lTB(t)$ rechter bzw. linker Teilbaum von t sind.
 AVL- Bäume haben für jeden Knoten v eine $Balance(v) \leq 1$.

Problem: Balance kann beim Einfügen und Streichen verloren gehen.

Es gibt 4 Rotationsoperationen:

Sei u der tiefste Knoten mit Imbalance.

- $Balance(u) = 2, Balance(v) = 1 \Rightarrow Rotation.links(u)$
- $Balance(u) = -2, Balance(v) = -1 \Rightarrow Rotation.rechts(u)$
- $Balance(u) = 2, Balance(v) = -1 \Rightarrow Doppelrotation.rechtsLinks(u)$
- $Balance(u) = -2, Balance(v) = 1 \Rightarrow Doppelrotation.linksRechts(u)$

Beachte: Diese Operationen können durch Umhängen von Zeigern in $\mathcal{O}(1)$ implementiert werden.

Streichen von Elementen: Streichen erstmal wie im einfachen Suchbaum. Imbalancen wie beim Einfügen von unten nach oben (tiefster Knoten zuerst auflösen).

Vorsicht: Für $|Balance(u)| = 2$ und $|Balance(v)| = 1$ hat der Teilbaum an v nach der (Doppel-)Rotation eine geringere Höhe als vorher der von u . Daher muss die Suche nach Imbalancen nach oben fortgesetzt werden.

Satz:
 Balancierte Bäume wie AVL-Bäume erlauben Suchen, Einfügen und Streichen in $\mathcal{O}(\log n)$ Zeit, wobei n die Knotenanzahl ist.

6.3 B-Bäume

Für große Datenmengen, die nicht in den Hauptspeicher passen, sind AVL-Bäume ungeeignet. Hier kommen B-Bäume zum Zuge, die für Zugriff auf externen Speicher entworfen wurden.

Definition: Ein B-Baum der Ordnung $k \geq 2$ ist ein Suchbaum

- dessen Blätter alle dieselbe Tiefe haben
- dessen Wurzel mindestens zwei und höchstens $2k - 1$ Kinder hat
- in dem jeder andere (innere) Knoten mindestens k Kinder und höchstens $2k - 1$ Kinder hat

Lemma: Sei T ein B-Baum der Ordnung k mit Höhe h und n Blättern, dann gilt: $2k^{h-1} \leq n \leq (2k - 1)^h$

Merksatz für "Höhe/Tiefe":

Wie *hoch* stehen Sie in Ihrer Firmenhierarchie? "Unter mir sind 4 Leute."
 Wie *tief* stehen Sie? "Über mir sind x Leute."

Logarithmieren der Formel aus dem Lemma ergibt

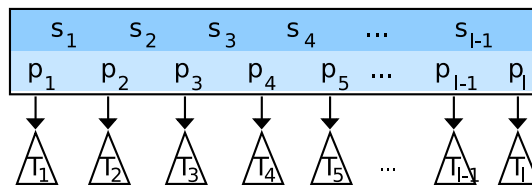
$$\log_{2k-1} n \leq h \leq 1 + \log_k \left(\frac{n}{2} \right)$$

⇒ Höhe vom B-Baum liegt in $\mathcal{O}(\log n)$

Operationen: Zugriff (Suche), Einfügen, Streichen

Annahme: Wir speichern knotenorientiert.

Betrachte Knoten u mit $k \leq l \leq 2k - 1$ Kindern.



Die Schlüssel k_1 bis k_l sind sortiert. Sei T_i der i -te Teilbaum von u , dann gilt für alle Knoten $v \in T_i$ und alle Schlüssel s in v :

- $s \leq s_i$, falls $i = 1$
- $s_{i-1} \leq s \leq s_i$, falls $1 < i < l$
- $s_{i-1} < s$, falls $i = l$

Suche (a, S)

Starte in Wurzel w und suche den kleinsten Schlüssel s_i in w mit $a \leq s_i$

Wenn $a = s_i \Rightarrow$ Element gefunden

Falls $s_i = \text{null}$: Suche rekursiv weiter im rechtesten Kind

Sonst: Suche rekursiv weiter im Teilbaum T_i

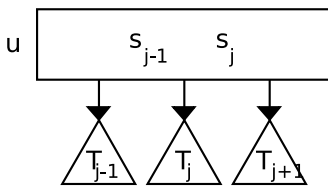
Laufzeit: $\mathcal{O}(\underbrace{\log_k n}_{\text{Höhe}} \cdot \underbrace{k}_{\text{lineare Suche}})$ kann verbessert werden zu $\mathcal{O}(\log_k n \cdot \underbrace{\log k}_{\text{binäre Suche}})$

Einfügen(a, S)

(Erfolgreiche) Suche endet in Blatt b mit Elternknoten v , der l Kinder hat.
 Sei wieder s_i der kleinste Schlüssel mit $a < s_i$ (falls existent). Füge neuen Schlüssel (samt Objekt) vor s_i ein (oder ganz rechts) und füge neues Blatt b' ein, auf das der Pointer nach a zeigt.
 Falls l vor dem Einfügen $< 2k - 1$ ist: Alles okay.
 Falls $l = 2k - 1$: Spalte v in zwei Knoten v' und v'' mit jeweils k Kindern. Vor der Spaltung enthält v $2k - 1$ Schlüssel. Nimm den mittleren Schlüssel und füge ihn im Vaterknoten an die richtige Stelle.
 Dadurch kann es rekursiv nach oben zu weiteren Aufsplittungen kommen.
 Falls die Wurzel gesplittet werden muss, wird oberhalb von der alten Wurzel ein neuer Knoten kreiert, der als einzigen Schlüssel den mittleren Schlüssel der alten Wurzel bekommt und auf v' und v'' zeigt. (Daher gilt für die Wurzel, dass sie mindestens zwei Kinder haben muss.)

Streichen eines Schlüssels aus einem B-Baum: **Annahme:** $a = s_j$ kommt in Knoten u vor
 Mittels $\text{Suche}(a, S)$ suche zunächst Knoten u , in dem a als Schlüssel s_j vorkommt.

Allgemeine Situation:

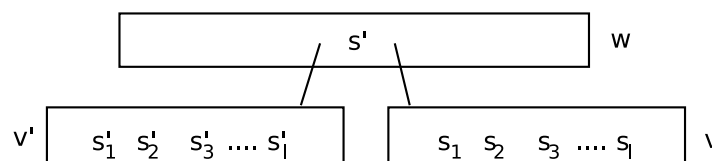


Fallunterscheidung:

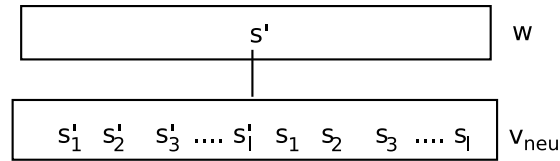
1. T_j ist ein Blatt, dann lösche s_j und T_j .
2. T_j ist kein Blatt. Sei s der größte Schlüssel in T_j , dann ersetze s_j durch s und lösche s samt seinem Blatt in T_j .

Sei v der Knoten in T_j mit dem größten Schlüssel und r die ursprüngliche Anzahl der Kinder von v mit $k \leq r \leq 2k - 1$.

1. Falls $r > k$ (bzw. $r > 2$, falls v die Wurzel ist), ist alles okay, denn die B-Baum-Eigenschaft bleibt erhalten.
2. Sei $r = k$ und v nicht die Wurzel:
 Sei w der Elternknoten von v und v' das Kind links von v und s' der Schlüssel in w zwischen diesen beiden Kindern.



Verschmelze nun v' und v :



Wegen $k \leq l \leq 2k - 1$ hat v_{neu} jetzt mindestens $k+k-1 = 2k-1$ Kinder und höchstens $2k-1+k-1 = 3k-2$ Kinder.

Falls v_{neu} mehr als $2k-1$ Kinder hat, wird er wieder (wie beim Einfügen auch) gesplittet. Jedes der Kinder hat dann mindestens k Kinder und höchstens $\lceil \frac{3k}{2} \rceil - 1$ Kinder.

Anmerkung: Hat v_{neu} genau $2k-1$ Kinder, darf nicht gespalten werden. Jedoch hat nun w ein Kind verloren und so muss rekursiv nach oben fortgefahren werden.

Zusammenfassung: B-Bäume

Zugriff (Suchen), Einfügen und Streichen kann in B-Bäumen der Ordnung k , die n Schlüssel verwalten, in Zeit $\mathcal{O}(k \cdot \log n)$ durchgeführt werden.

6.4 Kantenmarkierte Suchbäume (“Tries”)

Bisher haben wir Suchbäume so organisiert, dass der direkte Vergleich von Schlüsseln den weiteren Suchpfad im Baum bestimmt. Für sehr lange Schlüssel über einem Alphabet Σ sind Vergleiche der Art “1. Buchstabe von $x < 1$. Buchstabe von y ?” sinnvoller.

Idee: Jeder Knoten hat wieder mehrere Kinder, aber für jeden Buchstaben des Alphabets höchstens einen.

Definition: Sei Σ ein festes, endliches Alphabet.

1. Ein Baum heißt Σ -markiert, falls gilt:
 - (a) Jede Kante $(u, v) \in E$ ist mit einem Buchstaben $a \in \Sigma$ markiert. v heißt dann a -Sohn von u .
 - (b) Jeder Knoten hat höchstens einen a -Sohn.

2. Sei $T = (V, E)$ ein Σ -markierter Baum, dann ist $\phi : V \rightarrow E^*$ definiert durch:

$$\begin{aligned} \phi(v) &= \epsilon \text{ (leeres Wort), wenn } v \text{ Wurzel von } T \\ \phi(v) &= \phi(u)a, \text{ wenn } v \text{ der } a\text{-Sohn von } u \text{ ist} \end{aligned}$$

Wir sagen: v definiert das Wort $\phi(v) \in \Sigma^*$. $\phi(v)$ ist das Wort, das entsteht, wenn man die Buchstaben entlang des Pfades $P(w, v)$ von der Wurzel w zum Knoten v konkateniert.

3. Sei $S \subseteq \Sigma^*$, dann heißt T kantenmarkierter Suchbaum für S (engl. “Trie”) falls gilt:

- (a) Für jedes Blatt $v \in T$ ist $\phi(v)$ das Anfangswort (Präfix) eines Wortes $x \in S$.
- (b) Zu jedem $x \in S$ gibt es genau ein Blatt v , so dass $\phi(v)$ das Anfangswort von x ist.

Das Blatt v trägt als Information dasjenige Element $x \in S$, das mit $\phi(v)$ beginnt.
 \Rightarrow blattorientierte Speicherung

Beachte: Einen Trie gibt es nur für solche Mengen S , die präfixfrei sind, d.h. kein $x \in S$ ist ein echtes Anfangswort eines $y \in S$.

Präfixfreiheit lässt sich stets erreichen, indem ein neuer Buchstabe zu Σ hinzugefügt wird, z.B. $\#$. Dieser Buchstabe markiert ab jetzt das Ende eines Wortes, so dass z.B. $BAU\#$ nicht mehr Präfix von $BAUM\#$ ist.

Auf Σ sei eine Ordnung definiert, z.B. $a < b < \dots < z$ (lexikalische Sortierung). Seien nun die Söhne jedes Knotens nach dieser Ordnung sortiert, so sind in den Blättern die Elemente von S aufsteigend von links nach rechts sortiert.

Nachteile von Tries:

- Tries sind oft sehr dünn besetzt, d.h. fast alle Knoten haben weniger als $|\Sigma|$ Kinder. Diese Kinder könnte man verwalten
 1. als Array:
 - Vorteil: direkter Zugriff auf den nächsten Buchstaben per Index
 - Nachteil: Platzbedarf $|\Sigma| \cdot \#Knoten$ statt nur $\sum_{v \in V} Grad(v)$
 2. als Liste:
 - Vorteil: Platzbedarf $\sum_{v \in V} Grad(v)$
 - Nachteil: Nur lineare Suche beim Zugriff
- Es kann lange Wege im Baum geben ohne Verzweigungspunkte.

6.5 PATRICIA-Bäume

Idee:

1. Setze $\Sigma = \{0, 1\}$ und betrachte Binärcodierung von Buchstaben und Wörtern.
2. Ziehe die Wege ohne Verzweigung zu einem Knoten zusammen. Kennzeichne diesen Knoten mit der Anzahl der übersprungenen Buchstaben.
 \Rightarrow PATRICIA-Bäume (Practical Algorithms To Retrieve Information Coded In Alphanumerics)
 - Patricia-Bäume sind geeignet für sehr lange Schlüssel unterschiedlicher Länge.
 - Sie sind binäre Bäume, deren linker Sohn immer der 0-Sohn ist (rechts der 1-Sohn).
 - Jeder innere Knoten hat ein Feld 'skip', das angibt, wie lang der Weg ist, den dieser Knoten repräsentiert, d.h. wie viele Buchstaben überlesen werden müssen.
 - In den Blättern stehen die Schlüssel.

Beispiel: Suche(THAT, P), P ... Patricia-Baum (vom Beiblatt).

THAT: 1011101000000110111

1 + 0 = 1. Buchstabe= 1 \Rightarrow rechts

1 + 11 = 12. Buchstabe= 0 \Rightarrow links

12 + 1 = 13. Buchstabe= 0 \Rightarrow links

\Rightarrow Steht im gefundenen Blatt und ist also im Baum enthalten.

Beispiel: Suche(USA, P), P ... Patricia-Baum (vom Beiblatt).

USA: 110001011000001

1 + 0 = 1. Buchstabe= 1 \Rightarrow rechts

1 + 11 = 12. Buchstabe= 0 \Rightarrow links

12 + 1 = 13. Buchstabe= 0 \Rightarrow links

\Rightarrow Steht nicht im Blatt und ist daher nicht im Baum vorhanden.

6.6 Randomisierte Suchbäume: Skiplists (W. Pugh)

Idee von Skiplists ist, dass eine sortierte Liste mit zusätzlichen, in einer bestimmten Weise zufällig gewählten Verlinkungen die wesentlichen Operationen “Suchen”, “Einfügen” und “Löschen” in $\mathcal{O}(\log n)$ unterstützt.

Gegeben:

- Eine sortierte Teilmenge S aus einem Universum U mit einer natürlichen Ordnung, d.h. $S = \{x_1, x_2, \dots, x_n\}$ mit $x_i < x_{i+1}$ für alle $1 \leq i < n$.
- Eine Levelzuweisung $\mathcal{L} : S \rightarrow \mathbb{N}^+$

Die Levelzuweisung definiert Teilmengen, die Levelmengen L_i , so dass $x \in L_i \Leftrightarrow L(x) \geq i$. Es gilt also $S = L_1 \supseteq L_2 \supseteq \dots \supseteq L_v \supseteq \emptyset$, wobei L_v das höchste, nicht-leere Level repräsentiert.

Eine Skiplist wird nun wie folgend gebildet:

- Definiere einen Anfangsknoten, den “header”, der jeweils auf das kleinste Element jedes Level L_i zeigt.
- Für alle L_i : Jedes Element in L_i zeigt auf das nächstgrößere Element in L_i .
- Diese Zeiger sind mit i markiert und pro Knoten absteigend sortiert.
- Jedes größte Element im Level L_i zeigt auf “null”.

Wie kann die Levelzuweisung erfolgen?

Zufällige Levelzuweisung:

Wähle für die Skiplist $0 \leq p \leq 1$. In dem Moment, in dem ein Element x eingefügt wird, zieh so lange eine positive Zufallszahl ≤ 1 bis das Ergebnis $\geq p$. Die Anzahl der Züge ist der Level l , der x zugewiesen wird.

```
public int randomLevel (double p)
```

```
int level = 1;
Random random = new Random(11265);
while random.nextDouble() < p
    ++level;
return level;
```

Die zurückgegebene Levelzuweisung folgt einer geometrischen Verteilung, d.h. dass die Wahrscheinlichkeit, dass $level = k$ ist, ist:

$$P[level = k] = p^{k-1}(1 - p)$$

Und der Erwartungswert einer solchen Verteilung ist

$$E(level) = \sum_{k=1}^{\infty} k \cdot p^{k-1} \cdot (1 - p) = \frac{1}{1 - p}$$

\Rightarrow Erwarteter Platzverbrauch einer solcher Art zufälligen Skiplist ist $\mathcal{O}(n)$.

Wie hoch ist die höchste erwartete Levelzuweisung?

Lemma: Die Zahl der Level v bei zufälliger Levelzuweisung hat den Erwartungswert $\mathcal{O}(\log n)$. Es gilt sogar, dass $v \in \mathcal{O}(\log n)$ mit hoher Wahrscheinlichkeit.

[“Mit hoher Wahrscheinlichkeit” bedeutet, dass das Gegenereignis höchstens Wahrscheinlichkeit $\frac{1}{n}$ hat.]

Beweis: $v = \max \{L(x) | x \in S\}$

$L(x)$ ist geometrisch verteilt mit Erwartungswert $\frac{1}{1-p}$. Die Wahrscheinlichkeit, dass $L(x)$ größer als ein gewähltes $t \in \mathbb{N}$ ist, ist $< p^t$. Da die Levelzuweisung für alle Elemente unabhängig ist, ist die Wahrscheinlichkeit, dass es ein Element x (von n Elementen) gibt mit $L(x) > t$ kleiner oder gleich $n \cdot p^t$ ist (für $p = \frac{1}{2}$ also $\frac{n}{2^t}$).

Wir setzen $t = \alpha \cdot \log n$ [$\alpha > 2$].

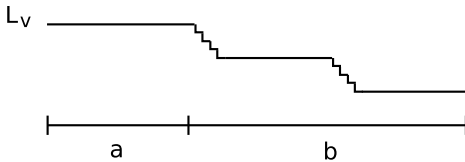
$\Rightarrow P[v > \alpha \cdot \log n] \leq \frac{n}{2^{\alpha \cdot \log n}} = \frac{1}{n^{\alpha-1}} < \frac{1}{n}$ für $p = \frac{1}{2}$.

\Rightarrow Höhe der Skiplist mit hoher Wahrscheinlichkeit in $\mathcal{O}(\log n)$ ist.

Suche(y, S)

Die Zeiger eines Knotens seien absteigend nummeriert mit der Nummer des Levels des Elements, auf das der Zeiger zeigt. Starte im *header* mit *levelpointer* auf das maximale, nicht leere Level v . In dem aktuellen Knoten überprüfen wir für die Zeiger kleiner oder gleich dem *levelpointer* in absteigender Reihenfolge, ob der Wert des Elements, auf das sie zeigen, kleiner oder gleich y ist. Gibt es keinen solchen Zeiger, dann ist y nicht in S enthalten. Ansonsten setze den *levelpointer* auf das Level des gefundenen Zeigers und folge dem Zeiger auf das nächste Element. Setze die Suche dort rekursiv fort.

Laufzeit: Wir unterteilen die Suchstrecke in zwei Teile:



a ... Wegstrecke zum maximalen Level L_v

b ... Wegstrecke zu niedrigeren Levels

a wird gebunden durch die Anzahl der Elemente in L_v . Wir betrachten dazu die Anzahl von Knoten in Level v' : $v' = \log_{\frac{1}{p}} n$

Ganz allgemein:

Die Wahrscheinlichkeit, dass ein Knoten x in Level $k + 1$ ist, ist $< p^k$. Die Anzahl von Elementen in Level k ist erwartet $\leq n \cdot p^k = 1$. Wir suchen k , so dass erwartet nur ein Element in dem Level ist.

$$n = \left(\frac{1}{p}\right)^k \Leftrightarrow \log_{\frac{1}{p}} n = k$$

b: Wir machen eine Rückwärtsanalyse, d.h. wir gehen von dem gefundenen Element zum höchsten Level. Sei also $C(k)$ die erwarteten Kosten, um in einem Knoten mit Level $\geq k$ zu kommen, mit $C(0) = 0$, $C(1) = 1$. Um in einem Knoten mit Level k zu sein, waren wir vorher (Rückwärtsanalyse!) entweder in einem anderen Knoten in dem gleichen Level oder in dem gleichen Knoten auf Level $k - 1$.

Da die Wahrscheinlichkeit, dass der zuletzt besuchte Knoten in Level $k - 1$ gleichzeitig auch in Level k ist, ist p , dass er es nicht ist $1 - p$.

$$C(1) = 1$$

$$C(k) = \underbrace{(1-p)}_C \underbrace{(1+C(k))}_A + \underbrace{p}_D \underbrace{(1+C(k+1))}_E$$

A ... Kosten für Folgen von Zeiger

B ... Wir waren schon in Level k

C ... Wahrscheinlichkeit, dass wir in Level k bleiben

D ... Wahrscheinlichkeit, dass wir ein Level aufsteigen

E ... Kosten, um Level $k - 1$ zu erreichen

$$C(k) = \frac{k}{p} \text{ (Umformungen)}$$

Da das maximale k in $\mathcal{O}(\log n)$ liegt, ist auch b in $\mathcal{O}(\log n)$ und damit der gesamte Suchpfad in $\mathcal{O}(\log n)$.

Einfügen(y, S)

Bestimme das Level $L(y)$ wie beschrieben. In jedem Level suche Einfügestelle, lege $2 \cdot L(y)$ Zeiger neu an.
 \Rightarrow Laufzeit $\mathcal{O}(\log n)$

Streiche(y, S)

Führe zuerst $Suche(y, S)$ aus. Lege in jedem der besuchten Level die Zeiger auf y auf den Nachfolger von y . \Rightarrow Laufzeit $\mathcal{O}(\log n)$

Satz: In einer zufälligen Skiplist für die Menge S der Größe n können Suchen, Einfügen und Streichen in erwarteter Zeit $\mathcal{O}(\log n)$ implementiert werden.

Da keinerlei Rebalancierungen ausgeführt werden müssen, ist diese Datenstruktur sehr einfach.

7 Graphenalgorithmen

Bisher haben wir uns hauptsächlich mit ungerichteten Graphen beschäftigt. Wenn Kanten nicht eine Menge von Knoten, sondern ein geordnetes Paar darstellen, dann nennen wir den Graph *gerichtet*.

Das Element $e = (v, w) \in E$ heißt Kante von v nach w . v ist der Startknoten und w der Zielknoten. w ist Nachbar(-knoten) von v bzw. w ist *adjazent* zu v .

Für einen Pfad $P(v, w)$ in einem gerichteten Graphen muss die Kantenfolge aus Kanten mit der richtigen Richtung bestehen, d.h. $P(v, w) = (e_1, e_2, \dots, e_k)$, dann muss

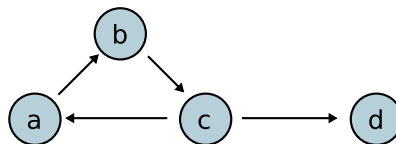
$$e_1 = (v, v_1)$$

$$e_k = (v_{k-1}, w)$$

Für alle $1 < i < k$ ist $e_i = (v_{i-1}, v_i)$

Allgemein: Startknoten der $(1 < i \leq k)$ -ten Kante ist Zielknoten der $(i - 1)$ -ten Kante.

Beispiel: Der Pfad (a, d) existiert. Nämlich $P(a, d) = ((a, b), (b, c), (c, d))$. Der Pfad $P(d, a)$ existiert nicht.



Ein gerichteter Graph heißt *zyklisch*, wenn er mindestens einen gerichteten Zykel enthält und sonst *azyklisch* (engl. DAG = directed acyclic graphs). Falls es einen gerichteten Pfad von v nach w in G gibt, so schreibt man $v \xrightarrow{*} w$.

Konventionsgemäß bzw. mathematisch korrekt:

- (v, w) : gerichtete Kanten
- $\{v, w\}$: ungerichtete Kanten

7.1 Darstellung von Graphen

Sei $G = (V, E)$ mit $V = \{0, 1, \dots, n\}$. Es gibt zwei Darstellungsarten für Graphen:

1. Darstellung als *Adjazenzmatrix* $A = (a_{ij})$:

$$a_{ij} = \begin{cases} 1 & , \text{ falls } e = (i, j) \in E \\ 0 & , \text{ sonst} \end{cases}$$

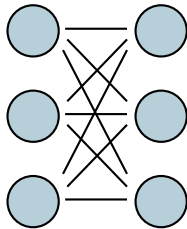
Bei ungerichteten Graphen ist die Adjazenzmatrix symmetrisch. Der Platzbedarf für diese Darstellung beträgt $\mathcal{O}(n^2)$. Das ist günstig, wenn die Anzahl der Kanten $m \in \mathcal{O}(n^2)$.

Aber: Insbesondere in realen Netzwerken ist E "dünn", d.h. die meisten Knoten haben nur sehr wenige Kanten zu anderen Knoten. $\Rightarrow m \approx \mathcal{O}(n)$

Insbesondere zwei Graphenklassen haben nur $\mathcal{O}(n)$ Kanten:

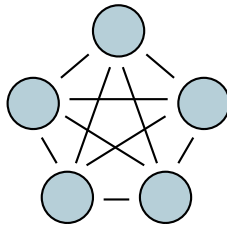
- Bäume: $n - 1$ Kanten
- Planare Graphen sind solche, die sich auf der Ebene zeichnen lassen, ohne dass sich Kanten kreuzen. Nicht-planare, ungerichtete Graphen sind beispielsweise:

$K_{3,3}$



Vollständig
bipartiter
Graph

K_5



Vollständiger
Graph
5-Clique

Ein Graph heißt "bipartit", wenn sich seine Knotenmenge V in zwei Teilmengen teilen lässt, so dass es zwischen Knoten der selben Teilmenge keine Kanten gibt.

Mitteilung: Ein planarer Graph hat $m \leq 3n - 6$ Kanten, also $m \in \mathcal{O}(n)$.

2. Darstellung als *Adjazenzliste*:

Speichere für jeden Knoten v seine Nachbarn in einer verketteten Liste. Wenn G gerichtet ist, hat jeder Knoten zwei Listen:

- $InAdj(v) = \{w \in V \mid (w, v) \in E\}$
- $OutAdj(v) = \{w \in V \mid (v, w) \in E\}$

Wenn G ungerichtet ist:

- $Adj(v) = \{w \in V \mid \{w, v\} \in E\}$

Platzbedarf: $\mathcal{O}(n + m)$

Nachteil: Zugriffszeit auf eine Kante ist abhängig von der Listenlänge, also dem Knotengrad.

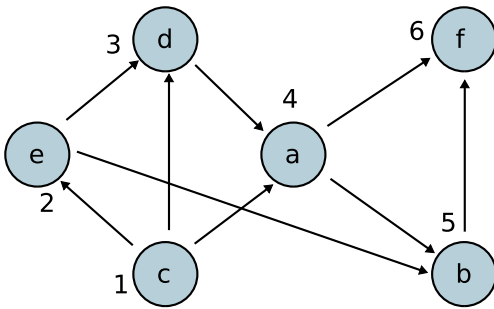
Bei gerichteten Graphen unterscheiden wir zwei Knotengrade:

- Eingangsgrad $indegree(v) = |InAdj|$
- Ausgangsgrad $outdegree(v) = |OutAdj|$

Ein gerichteter Graph ist ein *gerichteter Baum*, wenn gilt:

- (a) $\exists v_0 \in V$ mit $indegree(v_0) = 0$ und
- (b) $\forall v \in V \setminus \{v_0\} : indegree(v) = 1$ und
- (c) Graph muss azyklisch sein

7.2 Topologisches Sortieren



Sei $G = (V, E)$ ein gerichteter Graph, dann heißt die Abbildung $num : V \rightarrow \{1, 2, \dots, n\}$ mit $n = |V|$ topologische Sortierung, falls für alle $e \in E$ gilt: $(e = (u, v)) : num(u) \leq num(v)$

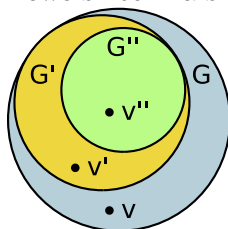
Lemma: G besitzt genau dann eine topologische Sortierung, wenn er azyklisch ist.

Beweis:

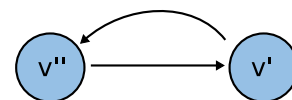
“ \Rightarrow “: Angenommen G ist zyklisch, dann ist $(v_0, v_1, \dots, v_k = v_0)$ ein Kreis. Es muss gelten: $num(v_0) < num(v_1) < \dots < num(v_k) = num(v_0)$ Widerspruch!

“ \Leftarrow “: Sei G azyklisch.
 Behauptung: G enthält einen Knoten v mit $indeg(v) = 0$ (# eingehender Kanten).
 Beweis per Induktion:

- Anfang: $|V| = 1$: klar
- Schritt: $|V| > 1$: Entferne beliebigen Knoten $v \in V$ und erhalte einen Graph $G' = (V', E')$ mit $V' = V \setminus \{v\}$, $E' = E \cap (V' \times V')$.
 Nach Annahme ist G' azyklisch und enthält ein v' mit $indeg(v') = 0$.
 Entferne v' aus G und erhalte G'' , der wieder ein v'' enthält mit $indeg(v'') = 0$. Lies den Beweis nochmals mit $v = v''$.



Es gilt: Entweder ist $indeg(v') = 0$ in G oder $indeg(v'') = 0$ in G , weil andernfalls:
 $(v'', v') \in E$ und
 $(v', v'') \in E$
 Das kann aber nicht sein, da G azyklisch ist.



topSort (G)

$|V|=1$: \checkmark

$|V|>1$:

wähle v mit $indeg(v)=0$

entferne v und weiter rekursiv auf $G'=(V', E')$

sei $num' : V' \rightarrow \{1, \dots, |V'|\}$ eine topologische Sortierung für G'

$$\text{Dann } num(w) = \begin{cases} num'(w) + 1 & , \text{ falls } w \neq v \\ 1 & , \text{ falls } w = v \end{cases}$$

```

count ← 0
while ∃v ∈ V mit indeg(v)=0 do
    count++
    num(v) ← count
    streiche v und alle ausgehenden Kanten
endwhile
if count < |V| then "G zyklisch"
    
```

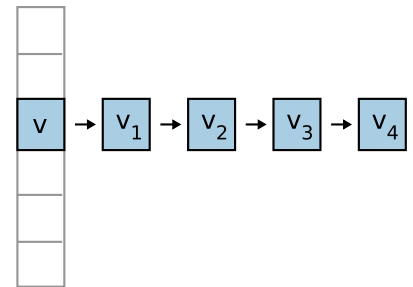
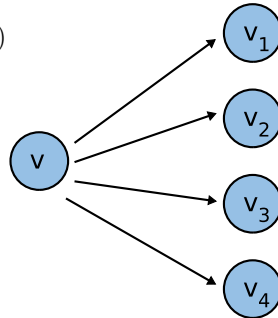
Ziel: $\mathcal{O}(\underbrace{n}_{|V|} + \underbrace{m}_{|E|}) \Rightarrow$ Adjazenzlisten!

Ausgehende Kanten löschen: $\mathcal{O}(outdeg(v))$

Alle löschen: $\sum_v outdeg(v) = \mathcal{O}(n + m)$

Benutze in-zähler-Array: beim Löschen der Kante (v, w) dekrementiere $inzaehler[w]$,

Merke alle Knoten mit $inzaehler = 0$ in Menge *ZERO* (als Stack) $\Rightarrow \mathcal{O}(n + m)$



7.3 Durchmusterungsalgorithmen

Bestimme alle Knoten, die von einem gegebenen $s \in V$ erreichbar sind.

```

S ← {s}
Markiere alle Kanten als unbenutzt.
while ∃(v,w) ∈ E mit v ∈ S und (v,w) unbenutzt do
    sei e=(v,w) eine solche Kante
    markiere e als benutzt
    S ← S ∪ {w}
endwhile
    
```

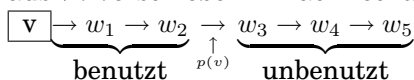
Invariante: In S sind diejenigen Knoten, die über benutzte Kanten erreichbar sind.

Probleme:

1. Realisierung benutzt \leftrightarrow unbenutzt
2. Finde geeignete Kante
3. Realisierung von S

Lösung:

1. Halte für jede Adjazenzliste einen Trennzeiger $p(v)$ zwischen benutzten und unbenutzten Kanten aus V . Verschiebe ihn nach rechts.



2. Wir halten $\tilde{S} \subseteq S$.
In \tilde{S} stehen Knoten, für die es noch unbenutzte ausgehende Kanten gibt.

3. "Initialisierung", "Einfügen" und "Ist $w \in S$?" sind Operationen.

Implementiere S als Boolean-Array, \Rightarrow jeweils in $\mathcal{O}(1)$

Operationen auf \tilde{S} : "Initialisierung", " $\tilde{S} = \emptyset$ ", "Einfügen", "Wähle $v \in \tilde{S}$ beliebig und streiche es eventuell"

Implementiere \tilde{S} als Stack oder als Queue oder als Heap.

Allgemeiner Durchmusterungsalgorithmus

siehe Extrablatt

Durch Änderung von Zeile 9 (Addieren von w an letzter/erster Stelle) ändert sich das Durchmusterungsverhalten stark.

letzte Stelle: Breitensuche (Breadth First Search, BFS)

erste Stelle: Tiefensuche (Depth First Search, DFS)

7.3.1 Tiefensuche

Implementierung ist speicherplatzineffizient (wegen der Zeiger). Rekursive Form der Tiefensuche:

```

1  void DFS (Node v, boolean[] visited)
2  visited[v] = true;
3  for all (v, w) ∈ E do
4      if not visited[w]
5          DFS(w, visited);
6      endif
7  endfor

```

Überzeugen Sie sich von der Äquivalenz der Algorithmen.

Die Tiefensuche unterteilt die Kanten des Graphen in vier Klassen T, B, F, C je nach Art des Besuches. Sei (v, w) die in Zeile 3 betrachtete Kante, dann

- (A) gehört (v, w) zu den **Baumkanten** T (tree), wenn w vorher nicht besucht wurde.
- (B) gehört (v, w) zu den **Vorwärtskanten** F (forward), falls w schon besucht wurde und $\exists w \xrightarrow{*}_T w$, d.h. wenn es einen Pfad von v nach w in T gibt.
- (C) gehört (v, w) zu den **Rückwärtskanten** B (backward), falls w schon besucht wurde und $\exists w \xrightarrow{*}_T v$.
- (D) gehört (v, w) zu den **Querkanten** C (cross), falls weder $v \xrightarrow{*}_T w$, noch $w \xrightarrow{*}_T v$ existieren.

Wir erweitern den rekursiven Algorithmus um $dfsnum[v]$ und $compnum[v]$, wobei

$dfsnum[v]$ angibt, als wievielter Knoten v zum ersten Mal besucht wurde.

$compnum[v]$ angibt, als wievielter Knoten v 's DFS-Aufruf beendet war (completed).

```

for all  $v \in V$  do
  visited[v] = false
  Initialisiere dfsnum[v] und compnum[v] auf 0
endfor
Setze globale Zähler z1, z2 auf 0
Initialisiere leere Mengen T, B, C, F
for all  $v \in V$  do
  if not visited[v] then
    DFS(v)
  endif
endfor

```

Laufzeit: $\mathcal{O}(n + m)$, da

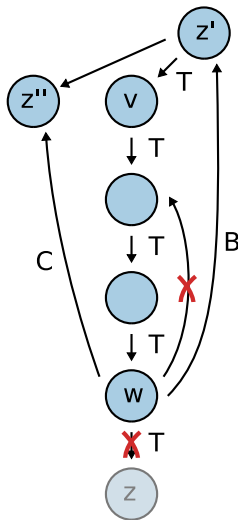
- die Einzellaufzeit eines Aufrufs ohne Rekursion $\mathcal{O}(1 + \text{outdegree}(v))$ ist.
- für jeden Knoten nur einmal ein DSF-Aufruf gemacht wird: $\mathcal{O}(\sum_{v \in V} (1 + \text{outdegree}(v))) = \mathcal{O}(n + m)$

Lemma: Eigenschaften, Teil 1

- Kantenklassen T, B, C, F bilden eine Partition der Kantenmenge E . (Beweis ist trivial.)
- T entspricht dem Wald (Menge von Bäumen) der rekursiven Aufrufe. (Beweis ergibt sich einfach aus dem Algorithmus.)
- $v \xrightarrow{T} w \Leftrightarrow \text{dfsnum}[v] \leq \text{dfsnum}[w] \wedge \text{compnum}[w] < \text{compnum}[v]$
- Seien $v, w, z \in V$ mit $v \xrightarrow{T} w$, $(w, z) \in E$ und $\neg(v \xrightarrow{T} z)$, dann gilt:
 - $\text{dfsnum}[z] < \text{dfsnum}[v]$
 - $(w, z) \in B$
 - $\text{compnum}[z] > \text{compnum}[v] \Leftrightarrow (w, z) \in B$
 - $\text{compnum}[z] < \text{compnum}[v] \Leftrightarrow (w, z) \in C$

Beweis:

- $v \xrightarrow{T} w$
 \Leftrightarrow Aufruf $DFS(w)$ ist geschachtelt in $DFS(v)$
 $\Leftrightarrow \text{dfsnum}[v] \leq \text{dfsnum}[w] \wedge \text{compnum}[w] \leq \text{compnum}[v]$
- Schrittweise:



- aus $v \xrightarrow{*}_T w$ folgt, dass $dfsnum[v] \leq dfsnum[w]$
- aus $(w, z) \in E$ folgt, dass $DFS(v)$ aufgerufen wird, bevor $DFS(w)$ und $DFS(v)$ enden.
- $\neg(v \xrightarrow{*}_T w) \Leftrightarrow DFS(z)$ nicht in $DFS(v)$ geschachtelt ist (folgt aus c).

$\Rightarrow DFS(z)$ startet vor dem Aufruf $DFS(v)$

$\Rightarrow dfsnum[z] < dfsnum[v]$ (i bewiesen)

- Wegen $dfsnum[z] < dfsnum[w]$ folgt, dass $(w, z) \notin T$ und wegen $(w, z) \notin T \wedge dfsnum[z] < dfsnum[v] < dfsnum[w]$ folgt, dass $(w, z) \notin F \Rightarrow (w, z) \in B \cup C$ (wegen a). (Beweis für ii)
- $(w, z) \in B \stackrel{\text{Def}}{\Leftrightarrow} z \xrightarrow{*}_T w \Leftrightarrow z \xrightarrow{*}_T v$ [da $v \xrightarrow{*}_T w$ existiert und wir wissen, dass $\neg(v \xrightarrow{*}_T z)$ ist, muss das die einzige Möglichkeit sein] $\Rightarrow compnum[z] > compnum[v]$ (Beweis zu iii)
- $(w, z) \in C \stackrel{\text{Def}}{\Leftrightarrow}$ weder $w \xrightarrow{*}_T z$ noch $z \xrightarrow{*}_T w$

Wir wissen, dass $dfsnum[z] < dfsnum[w]$. Wegen c folgt nun, dass $compnum[z] < compnum[w] < compnum[v]$ (denn sonst gäbe es $z \xrightarrow{*}_T w$) (Beweis zu iv).

Lemma: Eigenschaften, Teil 2 Sei $(v, w) \in E$

- e) $(v, w) \in T \cup F \Leftrightarrow dfsnum[v] < dfsnum[w]$
- f) $(v, w) \in B \Leftrightarrow dfsnum[w] < dfsnum[v] \wedge compnum[w] > compnum[v]$
- g) $(v, w) \in C \Leftrightarrow dfsnum[w] < dfsnum[v] \wedge compnum[w] < compnum[v]$

Beweis:

- e) $(v, w) \in T \cup F \Rightarrow v \xrightarrow{*}_T w \Rightarrow dfsnum[v] \leq dfsnum[w]$

Sei nun $dfsnum[v] \leq dfsnum[w]$. Da $(v, w) \in E$, muss $DFS(w)$ vor Abschluss von $DFS(v)$ aufgerufen werden.

\Rightarrow echte Schachtelung der Aufrufe

$\Rightarrow v \xrightarrow{*}_T w$

$\Rightarrow (v, w) \in T \cup F$

Bemerkungen:

1. Ob eine Kante $e = (v, w)$ in T, B, C, F liegt, kann also (Lemma) durch Vergleich der $dfsnum$ und $compnum$ festgestellt werden.
2. In azyklischen Graphen gibt es keine Rückwärtskanten, d.h. $\forall (v, w) \in E \text{ compnum}[v] > \text{compnum}[w]$.
 \Rightarrow Nummerierung $num(v) = n + 1 - \text{compnum}[v]$ ergibt eine topologische Sortierung.

7.3.2 Starke Zusammenhangskomponenten (SZKs)

Definition: Ein gerichteter Graph $G = (V, E)$ heißt stark zusammenhängend genau dann, wenn $\forall v, w \in V$ gilt: $v \xrightarrow[E]{*} w$ und $w \xrightarrow[E]{*} v$.

Die maximal stark zusammenhängenden Teilgraphen von G heißen starke Zusammenhangskomponenten (Maximal: Es kann keine Kante hinzugefügt werden ohne dass diese Bedingung verletzt wird.)

Idee für Algorithmus: Erweitere DFS

Sei $G' = (V', E')$ der bisher erforschte Teil von G . Verwalte die starken Zusammenhangskomponenten von G :

Starte $V' = \{S\}, E' = \emptyset, SZK = \{\{S\}\}$

Invariante: SZK enthält in jedem Schritt die SZK von G' .

Sei nun V' eine sich im Verlauf ergebende Teilmenge aus V mit $v \in V'$ und (v, w) der nächsten Kante in der DFS.

Ist $w \notin V'$, d.h. $(v, w) \in T$, dann erweitere SZK um $\{w\}$.

Ist $w \in V'$, dann füge (u.U.) mehrere SZKs zu einer zusammen.

Welche SZKs müssen zusammengefügt werden?

Notwendige Bezeichnungen:

- SZK K_i heißt *abgeschlossen*, wenn alle DFS Aufrufe für Knoten in K_i abgeschlossen sind.
- Die *Wurzel* von K_i ist der Knoten mit der kleinsten $dfsnum$ in K_i .
- Eine Teilfolge von Knoten, sortiert nach ihrer $dfsnum$ heißt *unfertig*, falls ihre DFS-Aufrufe aufgerufen, aber ihre SZK noch nicht abgeschlossen ist.
- Die Menge W enthalte die nach $dfsnum$ sortierte Folge von Wurzeln der noch nicht abgeschlossenen SZKs.

Beobachtungen am Beispiel:

1. $\forall v$ unfertig: $v \xrightarrow[E]{*} g$
2. $\nexists (v, w) \in E: v \in$ abgeschlossene Komponenten, $w \in$ nicht abgeschlossene Komponenten
 Von g ausgehende Kanten:
 - $(g, d) \in C$: Tut nichts, d liegt in abgeschlossener Komponente.
 - $(g, b) \in B$: Vereintigt 4 Komponenten $\{b, c\} \{e\} \{f\} \{g\}$
 \Rightarrow unfertig
 - $(g, h) \in T$: Erzeugt eine neue Komponente.

Proposition:

1. Eine SZK ist *maximal* in G , wenn sie abgeschlossen ist.
2. Eine SZK ist *abgeschlossen*, wenn der DFS-Aufruf ihrer Wurzel abgeschlossen ist.

Beweis:

1. Da von der SZK keine unbekanntes Kanten mehr ausgehen können, kann keine weitere Kante von ihr erreicht werden.
2. Trivial.

Verallgemeinerung: Implementiere Menge *unfertig* und Menge W (Wurzeln) als Stacks (LIFO, last in first out), so dass:

$(v, w) \in T$: *push*(w , *unfertig*)
 push(w , W)

$(v, w) \in C \cup F$: **Tue nichts.**

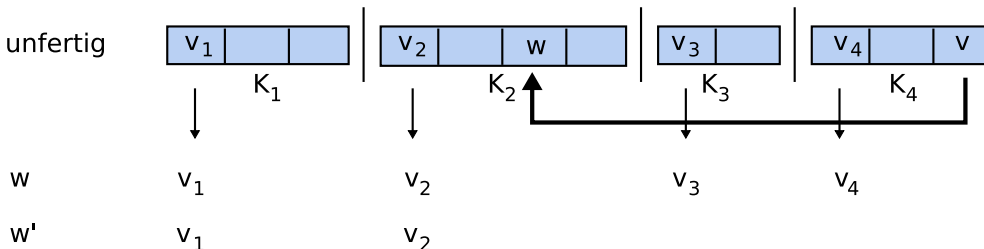
$(v, w) \in B$: Vereinige alle SZK, ausgehend von $SZK(v)$, deren Wurzeln z $dfsnum[v] > dfsnum[w]$

```

while dfsnum[top(w)] > dfsnum[w] do
  pop(w)
endwhile

```

D.h. wir entfernen die anderen Wurzeln aus dem Stack.

Beispiel:

$\Rightarrow K_2, K_3$ und K_4 werden vereinigt durch *pop*(w) bis *top*(w) = Wurzel der $SZK(w)$

Wenn $DFS(v)$ abgeschlossen ist und $v = top(w)$ ist, wird die SZK ausgegeben (wegen Proposition).

```

do
  z = pop(unfertig)
  print(z)
until (z = v)
pop(w)

```

Korrektheit: Invarianten

- (I_1) $\nexists (v, w) \in E$ mit v in den abgeschlossenen Komponenten und w in den nicht abgeschlossenen Komponenten.
- (I_2) Knoten aller nicht abgeschlossenen SZKs (sortiert nach $dfsnum$) liegen in einem Pfad in E' , ihre Wurzeln in einem Baumpfad.
- (I_3) Knoten jeder nicht abgeschlossenen Komponente K_i bilden ein "Intervall" im Stack *unfertig*, wobei das zu unterst liegende Element die Wurzel von K_i ist.

Zu zeigen: Invarianten bleiben beim Einfügen von Kanten erhalten.

1. Fall: $(v, w) \in T$

(I_1): Es wird keine SZK abgeschlossen

(I_2): Sei r die Wurzel der Komponente von r . Da die Wurzel einer SZK keinstes $dfsnum$ und größtes $compnum$ hat, folgt aus Lemma c): $v \xrightarrow{T} w$

Wegen $(v, w) \in T$ gilt: $r \xrightarrow{T} w$.

Wegen der Induktionsvoraussetzung bleibt I_2 erhalten.

(I_3): Trivial.

2. Fall: $(v, w) \in B$

(I_1): Es wird keine SZK abgeschlossen.

(I_2): Pfad wird verkürzt.

(I_3): Löschen der obersten Wurzeln vereinigt Intervalle zu einem neuen. Da vorher I_2 galt, bildet dieses Intervall eine eigene Komponente.

Satz: SZKs eines gerichteten Graphen können in $\mathcal{O}(n + m)$ berechnet werden.

Bemerkungen:

1. Der Test, ob ein Knoten $w \in unfertig$ ist (für die Frage, ob SZKs vereinigt werden müssen) kann durch ein bool'sches Array implementiert werden.
2. Jeder Knoten wird höchstens einmal in *unfertig* und W aufgenommen.

IN DEN VORLESUNGEN AM 28.06.2007, 03.07.2007 UND 05.07.2007 KAM EINE PRÄSENTATION ZUM EINSATZ, DIE SICH AUF DER INTERNETSEITE ZUR VORLESUNG FINDEN LÄSST.

8 Hashing

In einem Universum $U = [0, \dots, N - 1]$ sei $S \subseteq U$ die zu verwaltende Menge.

Zur Verfügung stehen sollen die Operationen "Zugriff", "Einfügen" und "Streichen".

Hashtafel: $T[0, \dots, m - 1]$

Hashfunktion: $h : U \rightarrow [0, \dots, m - 1], a \rightarrow T[h(a)]$

Beispiel: $N = 50, m = 3, S = \{2, 21\}$

$$h(x) = x \bmod 3$$

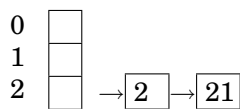
0	21
1	
2	2

Problem: Kollisionen, also $h(x) = h(y)$ für $x \neq y$, z.B. $x = 2, y = 20$

Ziel: kleine Hashtafel, wenig Kollisionen

8.1 Hashing mit Verkettung (Chaining)

Die i -te Liste in Tafel T ($0 \leq i \leq m - 1$) enthält alle $x \in S$ mit $h(x) = i$



Laufzeit: worst-case: $\mathcal{O}(n)$ (alle Elemente in einer Liste), aber im Mittel viel besser.

Wahrscheinlichkeitsannahmen:

- $h(x)$ kann in $\mathcal{O}(1)$ ausgewertet werden
- $|h^{-1}(i)| = \frac{|U|}{m}$ für alle $i = 0, \dots, m - 1$ (gleichmäßige Verteilung der Zahl der Einträge)
 h soll Einträge von U gleichmäßig verteilen
- Für eine Folge von n Operationen gilt: Wahrscheinlichkeit, dass j -tes Element der Folge ein festes $x \in U$ ist, ist $\frac{1}{N}$.
 - ⇒ Operationen sind unabhängig und gleichverteilt.
 - ⇒ Sei x_k Argument der k -ten Operation
 $\text{prob}(h(x_k) = i) = \frac{1}{m}$
d.h. auch Funktionswerte sind gleichverteilt.

Definition:

$$\delta_h(x, y) = \begin{cases} 1 & \text{, falls } x \neq y, h(x) = h(y) \\ 0 & \text{, sonst} \end{cases}$$

$$\delta_h(x, S) = \sum_{y \in S} \delta_h(x, y) \text{ (Zahl der Element in } T(h(x)))$$

⇒ Kosten von Operation $\text{Zugriff}(x) = 1 + \delta_h(x, S)$

Satz: Mittlere Kosten von $\text{Zugriff}(x)$ sind $1 + \frac{n}{m} = 1 + \beta$ ($\beta \dots$ Belegungsfaktor)

Beweis: Sei $h(x) = i$ und p_{ik} die Wahrscheinlichkeit, dass Liste i genau k Elemente enthält, dann ist $p_{ik} = \binom{n}{k} \cdot \left(\frac{1}{m}\right)^k \cdot \left(1 - \frac{1}{m}\right)^{n-k}$. Erwartungswert der Kosten des Zugriff $f(x)$ ist:

$$\sum_{k \geq 0} p_{ik}(1+k) = \underbrace{\sum_{k \geq 0} p_{ik}}_{=1} + \sum k \cdot \binom{n}{k} \cdot \left(\frac{1}{m}\right)^k \cdot \left(1 - \frac{1}{m}\right)^{n-k}$$

Da $k \cdot \binom{n}{k} = n \cdot \binom{n-1}{k-1}$, gilt

$$\begin{aligned} &= 1 + \frac{n}{m} \cdot \sum \binom{n-1}{k-1} \cdot \left(\frac{1}{m}\right)^{k-1} \cdot \left(1 - \frac{1}{m}\right)^{n-k} \\ &= 1 + \frac{n}{m} \cdot \left(\frac{1}{m} + \left(1 - \frac{1}{m}\right)\right)^{n-1} = 1 + \frac{n}{m} \end{aligned}$$

Wie groß sollte β sein?

$\beta \leq 1$: Zugriff erwartet in $\mathcal{O}(1)$

$\beta \leq \frac{1}{4}$: Platz für Hashtafel und Listen ist $\mathcal{O}(n+m) = \mathcal{O}(n + \frac{n}{\beta})$
 $\approx \mathcal{O}(n)$ für geeignet große β , z.B. $\beta = \frac{1}{2}$

Beachte: Durch Einfügen und Streichen kann β schnell zu groß bzw. zu klein werden. \Rightarrow Rehashing.

\Rightarrow Folge von Hashtafeln T_0, T_1, \dots der Größe $m, 2m, 4m, \dots$ (T_i hat die Größe $2^i \cdot m$)

\Rightarrow Falls $\beta = 1$: Umspeichern in $T_{i+1} \Rightarrow \beta = \frac{1}{2}$

\Rightarrow Falls $\beta = \frac{1}{4}$: Umspeichern nach $T_{i-1} \Rightarrow \beta = \frac{1}{2}$

Technik: Amortisierte Analyse zeigt, dass n Operationen erwartet in $\mathcal{O}(n)$ gehen.

Alternative: Offene Adressierung. Nur 1 Element pro Tafel­eintrag. Folge von Hash­funktionen. Ist 1. Eintrag belegt, probiere zweiten.

h_1, h_2, \dots z.B. $h_i(x) = (h(x) + 1) \bmod m$ (linear probing) oder $h_i(x) = (h(x) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$ (quadratic probing)

8.2 Perfektes Hashing

Die Hashfunktion soll injektiv sein. Sei S bekannt.

Stufe 1: Hashing mit Verkettung \Rightarrow Listen

Stufe 2: Für jede Liste eine eigene injektive Hashfunktion.

Wahl einer injektiven Hashfunktion: Sei $U = \{0, \dots, N-1\}$, $h_k : \{0, \dots, N-1\} \rightarrow \{0, \dots, m-1\}$, $k \in \{1, \dots, N-1\}$ mit $h_k(x) = ((k \cdot x) \bmod N) \bmod m$.

Sei S bekannt. Wähle k mit h_k injektiv. Wir messen Injektivität wie folgt:

$$b_{ik} = |\{x \in S \mid h_k(x) = i\}| \text{ für } 1 \leq k \leq N-1, 0 < i \leq m-1$$

also bestimmen wir die Zahl der $x \in S$ mit gleichem Hashwert i für h_k . Dann ist $b_{ik}(b_{ik}-1) = |\{(x, y) \in S^2 \mid x \neq y, h_k(x) = h_k(y) = i\}|$ also die Zahl der Paare, die Injektivität verletzen.

Sei $B_k = \sum_{i=0}^{m-1} b_{ik}(b_{ik}-1)$. Falls $B_k < 2$, dann ist $h_k|_S$ injektiv.

Ist $b_{ik} = \frac{n}{m}$ für alle $i = 0, \dots, m-1$, dann gilt: $B_k = \sum_{i=0}^{m-1} \left(\frac{n}{m}\right) \cdot \left(\frac{n}{m} - 1\right) = n \cdot \left(\frac{n}{m} - 1\right)$

Lemma: Mit obigen Voraussetzungen und einer Primzahl N gilt:

$$\sum_{k=1}^{N-1} \sum_{i=0}^{m-1} b_{ik} \cdot (b_{ik} - 1) \leq 2 \cdot \frac{n(n-1)}{m} \cdot (N-1)$$

Beweis: siehe Literatur.

Bemerkung: Die mittlere Zahl der Kollisionen ist $2 \cdot \frac{n(n-1)}{m}$.
 \Rightarrow Bei $m > n(n-1)$ ist die mittlere Anzahl von Kollisionen < 2 .
 \Rightarrow Es existieren $h_k, h_{k|S}$ injektiv.

Korollar: Mit obiger Voraussetzung gilt:

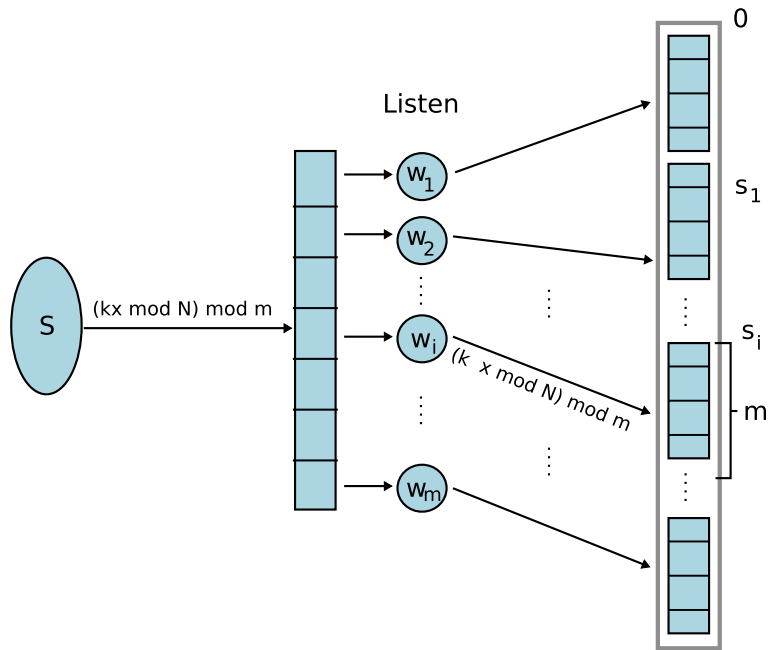
1. Es existiert ein $k \in [1, \dots, N-1]$, so dass $B_k \leq 2 \cdot \frac{n(n-1)}{m}$.
2. Sei $A = \left\{ k \mid B_k > 4 \cdot \frac{n(n-1)}{m} \right\}$ und $|A| > \frac{N-1}{2}$.
 $\Rightarrow \sum_{k \geq 1} B_k \geq \sum_{k \in A} B_k > \left(\frac{N-1}{2}\right) \cdot \frac{4(n-1)n}{m} = 2 \cdot \frac{(n-1)n}{m} (N-1)$
 \Rightarrow Mindestens $\frac{N-1}{2}$ h_k haben $B_k \leq \frac{4n(n-1)}{m}$ Widerspruch!

Korollar: Mit obigen Voraussetzungen gilt:

1. Ein $k \in [1, \dots, N-1]$ mit $B_k \leq 2 \cdot \frac{n(n-1)}{m}$ kann in Zeit $\mathcal{O}(m + Nn)$ bestimmt werden.
2. Sei $m = n(n-1) + 1$, dann gibt es ein k mit $h_{k|S}$ injektiv und k kann in Zeit $\mathcal{O}(n^2 + nN)$ gefunden werden.
3. Sei $m = 2n(n-1) + 1$, dann ist die Hälfte der $h_{k|S}$ injektiv. Bestimmungzeit ist randomisiert $\mathcal{O}(n^2)$.
4. Ein $k \in [1, \dots, N-1]$ mit $B_k \leq 4 \cdot \frac{n(n-1)}{m}$ kann randomisiert in Zeit $\mathcal{O}(m + n)$ gefunden werden.
5. Sei $m = n$. Ein k mit $B_k \leq 2(n-1)$ kann in Zeit $\mathcal{O}(nN)$ gefunden werden.
6. Sei $m = n$. Randomisiert kann ein k mit $B_k \leq 4 \cdot (n-1)$ in $\mathcal{O}(n)$ gefunden werden.

Realisierung:

1. Bei $m = n$ gibt es ein h_k , so dass die Länge der Listen $\mathcal{O}(\sqrt{n})$, da B_k maximal das Quadrat der Längen und $B_k < 4n$ ist.
2. Wende auf jede Teilmenge mehrmals Hashing an. Sei $|S| = n = m$.
 - (a) Sei k gewählt, so dass $B_k \leq 4(n-1) < 4n$.
 - (b) Sei $w_i = \{x \in S \mid h_k(x) = i\}$, $b_i = |w_i|$ und $m_i = 2 \cdot b_i \cdot (b_i - 1) + 1$. Wähle k_i mit $h_{k_i}(x) = (k_i x \bmod N) \bmod m$.



h_{k_i} ist injektiv für w_i

3. Sei $s_i = \sum_{j < i} m_j$. Speichere x in $T(s_i + j)$ wobei $i = (kx \bmod N) \bmod m$ und $j = (k_i x \bmod N) \bmod m_i$.

Platz: $m = \sum_{i=0}^{n-1} m_i = \sum_{i=0}^{n-1} 2 \cdot b_i(b_i - 1) + 1 = n + 2 \cdot B_k \leq 9n = \mathcal{O}(n)$

Laufzeit:

1. $\mathcal{O}(n)$

2. & 3. $\mathcal{O}(\sum |w_i|) = \mathcal{O}(n)$

Satz: Mit obigen Voraussetzungen kann für S eine perfekte Hashfunktion mit $\mathcal{O}(1)$ Zugriffszeit und Tafel der Größe $\mathcal{O}(n)$ in Zeit $\mathcal{O}(nN)$ deterministisch und in $\mathcal{O}(n)$ randomisiert gefunden werden.

Es gibt auch die dynamische Version, bei der S nach und nach aufgebaut wird. Dies geht in Zeit $\mathcal{O}(n)$ bei einer Tafel der Größe $\mathcal{O}(n)$.

Beim Hashing mit Verkettung wird eine Listenlänge von $1 + \frac{n}{m} = 1 + \beta$ erwartet, aber die erwartete Länge der längsten Liste ist $\mathcal{O}\left(\frac{\log n}{\log \log n}\right)$.

Beweis: Sei S zufällig aus U gewählt, $\text{prob}(h(x) = i) = \frac{1}{m}$ für $x \in S$, $i \in [0, \dots, m - 1]$, L die Länge der längsten Liste und $l(i)$ die Länge der Liste i .

$$\text{prob}(l(i) \geq j) \leq \binom{n}{m} \left(\frac{1}{m}\right)^j$$

Es gilt:

$$\begin{aligned} \text{prob}(\max_i l(i) \geq j) &\leq \sum_{i=0}^{m-1} \text{prob}(l(i) \geq j) \\ &\leq m \cdot \binom{n}{j} \left(\frac{1}{m}\right)^j \\ &= m \cdot \frac{n!}{(n-j)! \cdot j!} \left(\frac{1}{m}\right)^j \\ &\leq n \cdot \left(\frac{n}{m}\right)^{j-1} \cdot \frac{1}{j!} \end{aligned}$$

$$\begin{aligned}
 E(L) &= \sum_{j \geq 1} \text{prob}(\max l(i) \geq j) \\
 &\leq \sum_{j \geq 1} \min \left\{ 1, n \cdot \left(\frac{n}{m}\right)^{j-1} \cdot \frac{1}{j!} \right\}
 \end{aligned}$$

Sei $j_0 = \min \left\{ j \mid n \left(\frac{n}{m}\right)^{j-1} \cdot \frac{1}{j!} \leq 1 \right\} \leq \min \{j \mid n \leq j!\}$, da $\frac{n}{m} \leq 1$.

Es gilt $j! \geq \left(\frac{j}{2}\right)^{\frac{j}{2}}$, also gilt: $j_0 = \mathcal{O}\left(\frac{\log n}{\log \log n}\right)$

$$\begin{aligned}
 \Rightarrow E(L) &\leq \sum_{j=1}^{j_0} 1 + \sum_{j > j_0} \frac{1}{j_0^{j-j_0}} \\
 &= \mathcal{O}\left(\frac{\log n}{\log \log n}\right)
 \end{aligned}$$

Beachte: Im Mittel sind Längen $\mathcal{O}(1 + \beta)$. Es gibt Listen mit Länge $\mathcal{O}\left(\frac{\log n}{\log \log n}\right)$.

ZUR VORLESUNG VOM 17.07.2007 GIBT ES EINE PRÄSENTATION, DIE AUF DER VORLESUNGSHOMEPA-GE ZU FINDEN IST. IM FOLGENDEN DAHER NUR DER ZUGEHÖRIGE TAFELANSCHRIEB.

9 Matching im Dating-Business

Formalisierung

Eingabe: Rangordnung von einer Person zu allen anderen (weiblich/männlich)

Formal: Zwei Mengen $A = \{1, \dots, n\}$ und $B = \{1, \dots, n\}$

Rangordnung:

- $\forall a \in A \exists R_a : B \rightarrow \{1, 2, \dots, n\}$, R_a ist bijektiv
- $\forall b \in B \exists R_b : A \rightarrow \{1, 2, \dots, n\}$, R_b ist bijektiv

Ausgabe: Gesucht sind n Dupel.

Bijektive Zuordnung $M : A \rightarrow B$ suchen (Matching), so dass M stabil ist.

Was ist Stabilität? $\forall a \in A$ und $(a, \underbrace{M(a)}_{=b})$

- $\forall a \neq a' \in A : R_{a'}(b) > R_{a'}(\underbrace{M(a')}_{=b'})$ und $R_b(a) > R_b(a')$
- $\forall b' \neq b \in B : R_{b'}(a) > R_{b'}(a')$ und $R_a(b) > R_a(b')$

Beweis zu Satz 9.1:

Sei M das vom Algorithmus berechnete Matching.

Widerspruch: Angenommen $\exists (m, w) \in M, (m', w') \in M$, so dass

- $R_w(m') > R_w(m)$ und $R_{m'}(w) > R_{m'}(w')$ oder
- $R_{w'}(m) > R_{w'}(m')$ und $R_m(w') > R_m(w)$

Beweis zu stabilen Matchings:

$S \neq S^*$, so dass m nicht mit der besten für ihn erreichbaren Frau w an einem Tisch sitzt.

$\Rightarrow w$ präferiert m'

Wenn m der erste Mann im Algorithmus ist, dem das passiert, ist m' bisher nicht abgelehnt worden.

$\Rightarrow m'$ findet w attraktiver als alle anderen $w' \neq w$

Da w für m erreichbar ist, muss es S' geben mit $(m, w) \in S'$.

Sei w' die Frau, mit der m' zusammensitzt. \Rightarrow Widerspruch!

Inhaltsverzeichnis

1	Worum geht es?	2
2	Was ist ein Algorithmus?	2
3	Analyse von Algorithmen	2
3.1	Allgemeine Komplexität	2
3.2	Asymptotische Notation	3
3.3	\mathcal{O} -Notation	3
3.4	\mathcal{O} -Notation	3
3.5	Ω -Notation	3
3.6	ω -Notation	3
3.7	Θ -Notation	3
4	Suchen	4
4.1	Lineare Suche	5
4.2	Binäre Suche	6
4.3	Interpolationssuche	8
5	Sortieren	13
5.1	Sortieren durch Einfügen	13
5.2	QuickSort	13
5.3	MergeSort	15
5.4	HeapSort	16
5.5	BucketSort	18
5.6	Auswahlproblem	20
6	Balancierte Bäume	21
6.1	Bäume	21
6.2	AVL-Bäume	22
6.3	B-Bäume	24
6.4	Kantenmarkierte Suchbäume ("Tries")	27
6.5	PATRICIA-Bäume	28
6.6	Randomisierte Suchbäume: Skiplists (W. Pugh)	29
7	Graphenalgorithmen	31
7.1	Darstellung von Graphen	32
7.2	Topologisches Sortieren	33
7.3	Durchmusterungsalgorithmen	34
7.3.1	Tiefensuche	35
7.3.2	Starke Zusammenhangskomponenten (SZKs)	38

8 Hashing	40
8.1 Hashing mit Verkettung (Chaining)	41
8.2 Perfektes Hashing	42
9 Matching im Dating-Business	45

Index

- Adjazenzliste, 32
- Adjazenzmatrix, 32
- Analyse
 - Laufzeit-, 8
 - von Algorithmen, 2
- Auswahlproblem, 20
- Bäume, 21
 - AVL-, 22
 - B-, 24
 - balancierte, 21
 - PATRICIA-, 28
- Baum, 16
 - ausgewogener, 17, 18
 - binärer, 16
- BucketSort, 18
- Chaining, 41
- DFS, 35
- Durchmusterung, 34
- Gleichverteilung, 8
- Graphenalgorithmen, 31
- Hashing, 40
- Heap, 16
- HeapSort, 16
- Interpolationssuche, 8
- Komplexität
 - allgemeine, 2
- Master, 10
- Median, 20
- MergeSort, 15
- Notation
 - asymptotische, 3
 - O-, 3
 - o-, 3
 - Omega-, 3
 - omega-, 3
 - Theta-, 3
- Pfad, 16
- Pivotelement, 13
- Präfix, 27
- Quicksort, 13
- Rekursionen, 10
- Skiplists, 29
- Sortieren, 13
 - BucketSort, 18
 - HeapSort, 16
 - MergeSort, 15
 - QuickSort, 13
 - topologisches, 33
- Suchbäume
 - kantenmarkierte, 27
 - randomisierte, 29
- Suche
 - binäre, 6
 - Interpolations-, 8
 - lineare, 5
 - Tiefen-, 35
- Suchen, 4
- Tiefensuche, 35
- Tries, 27
- Zusammenhangskomponenten
 - starke, 38
- Zykel, 16