Freie Universität Berlin,
Fachbereich Mathematik und Informatik
Institut für Informatik

Alp3:
Mitschrift zur Vorlesung

Algorithmen und Programmierung 3
Dozent: Wolfgang Munzer

Lizenz: Public Domain

von:
Cornelius Horstmann
Ludwig Schuster

Inhaltsverzeichnis

19.10.2010

0.1. **Informationen zur Veranstaltung.**

0.1.1. *Dozent.*

Name	Wolfgang Mulzer
Sprechzeiten	Di 16-17 R114
Email	mulzer@inf.fu-berlin.de
Webseite	http://www.inf.fu-berlin.de/lehre/WS10/ALP3/index.html

0.1.2. *Klausur.* **Donnerstag 17.02.2011, 14-16 Uhr, T9 HS + HS 1a (Silberlaube)**[1]

0.1.3. *Scheinkriterien.*

aktive Teilname:	≥ 60 der Übungspunkte (9 Tage Zeit pro Zettel)
Klausur:	≥ 50 der Klausurpunkte
Tutorium:	2 mal Vorrechnen; keine Anweseheitspflicht (aber Empfehlung ;))

0.1.4. *Literatur.*

Algorithm irgendwas (enlish wichtig siehe KVV)

Algorithmen und Datenstrukturen

0.1.5. *Überblick.*

ALP1: Funktionale Programmierung (Haskell)

ALP2: OOP (Java)

ALP3: Abstrakte Datentypen; Datenstrukturen; Algorithmen

ALP4: Nichtsequentielle Programmierung

ALP5: Netzprogrammierung

0.1.6. *It's all about Data!*

Datenstrukturen	Baum, Liste, Hastabelle, ...
Manipulation	Algorithmen (Quicksort, binäre Suche, Huffman-Kompression ...)
Zugriff	Stapel (Stack), Warteschlange (Queue), Wörterbuch (Dictionary) — Geheimnisprinzip [2]
Speicherung	Dateisystem, Freispeicherverwaltung

0.1.7. *Ziel:* Die verschiedenen Methoden mit Daten umzugehen, kennenzulernen und theoretisch analysieren

[1] incl. Freiversuch; die bessere Note (das Maximum der Punkte) wird gewertet

[2] je weniger man weiß desto glücklier ist man

1. Datenstrukturen

1.1. **Polynome.**

1.1.1. *Definition.* sei $n \in \mathbb{N}$ und $a_0, a_1, \cdot, a_n n + 1$ Zahlen ein Polynom vom Grad n ist der Formale Ausdruck $p(x) = a_0 + a_1 x + a_2 x^2 + \cdot + a_n x^n$

$$= \sum_{i=1}^{n} a_i x^i$$

$$x^5 + 2x^2 + x - 70$$

1.1.2. *Darstellung im Rechner.* Koeffizientendarstellung; Speichern: $a_0, a_1, \ldots, a_n$ in einem Array

1.1.3. *Operationen.*

- Auswerten bei x_0
- Addition $r(x) = p(x) + q(x)$
- Multiplikation $r(x) = p(x) + q(x)$

1.1.4. *Implementation der Operationen.*

Addition

$$a_0, a_1, \ldots, a_n$$
$$b_0, b_1, \ldots, a_n$$
$$c_0, c_1, \ldots, c_n$$
$$c_i := a_i + b_i \forall i = 0, \ldots, n$$

Auswertung Berechne: $p(x_0) = a_0 + a_1 \cdot x_0 + a_2 \cdot x_0^2 + \cdot + a_n \cdot x_0^n$
Hornerschema: $p(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \ldots, x_0 a_n))$

n mal addieren und n mal multiplizieren = O(n)

Multiplikation

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$
$$q(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_n x^n$$

$$q(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n$$

$$c_1 = \sum_{k=0}^{i} a_k b_{i-k}$$

$$a_j, b_j := 0 \forall j > n$$

um c_i zu berechnen muss man i mal addieren und i+1 mal multiplizieren (für $0 \leq i \leq n$)
$a_n - 1$ Additionen + $2_n - i + 1$ Multiplikationen
Insgesamt: $O(n^2)$ Operationen

1.1.5. *Bessere Implementation.* Fakt: Seien $Z_0, Z_1, \cdot Z_m$ m+1 Paarweise verschiedene Zahlen, Fest. Sei $p(x)$ Polynm vom Grad $n \leq m$. Dann ist p durch $p(Z_0), p(Z_1), \cdot, p(z_m)$ eindeutig bestimmt.
andere Darstellung; Wertdarstellung: Fixiere $Z_n, Z_1, \ldots, Z_m$ wie im Fakt. Speichere $p(Z_0), p(Z_1), \ldots, p(Z_m)$ in einem Array.

Addition

$$p(Z_0), p(z_1), \ldots, p(z_m)$$
$$q(Z_0), q(z_1), \ldots, q(z_m)$$

$$r(z_i) = p(z_i) + q(z_i)$$

$O(m) = O(n)$ Operationen, wenn $m = n$

Multiplikation

$$r(z_i) = p(z_i) \cdot q(z_i)$$

wenn $m = 2n$ ist r eindeutig festgelegt ist $O(n)$ Operationen

Auswertung

$O(n^2)$ Zeit

Die Laufzeiten in einer Tabelle

	Koeffizientendarstellung	Wertedarstellung
ADD	$O(n)$	$O(n)$
MULT	$O(n^2)$	$O(n)$
Einsetzen	$O(n)$	$O(n^2)$

1.1.6. *Umrechnung.* Wie kommt man von der Koeffizientendarstellung zur Wertedarstellung?

Theoretisch/Allgemein: n Werte einsetzen $\rightarrow O(n^2)$ Zeit
ABER: wir können uns $z_{a_i}, \ldots, z_n$ aussuchen!

Jetzt ziehen wir den Hasen aus dem Hut. TATATATAAAAA!

Sei ω eine primitive $n+1$ Einheitswurzel (Komplex)

d.h. $\omega^{n+1} = 1$ und $\omega^k \neq 1$ Für $K = 1, \ldots, n)$

setze: $z_i := \omega^i$ für $i = o, \ldots, n$

Wir wollen $p(\omega^0), p(\omega^1), \ldots, p(\omega^n)$ ausrechnen. nim an: $n+1$ ist zweierpotenz (sonst fülle mit 0 auf)

$$p(x) = \sum_{i=0}^{n} a_i x^i = \sum_{i=0}^{\frac{n+1}{2}-1} a_{2i} x^{2i} + x \sum_{i=0}^{\frac{n+1}{2}-1} a_{2a+1} x^{2i}$$
$$= g(x^2) + xu(x^2)$$

g, u sind Polynome mit halb so vielen Koeffizeinten wie p

Wir wollen einsetzen in g und $\omega^0, \omega^2, \omega^4, \ldots, \omega^{2n}$ das sind nur $\frac{n+1}{2}$ viele verschiedene Zahlen!

$$\omega^n + 1 = \omega^0 = 1$$

$$\omega^n + 1 = \omega^2$$

Bsp:

$$p(x) = 5x^7 + 3x^6 - x^5 + 2x^4 - 10x^3 + 2x^2 + x - 7$$

$$= g(x2) + x \cdot u(x^2)$$

wobei

$$g(x) = 3x^2 + 2x^2 + +2x - 7; u(x) = 5x^3 - x^2 - 10x$$

Algorithm1 berechne : ....

$$1 = \omega^0 = \omega^n + 1$$

$$\omega^2 = \omega^n + 3$$

$$\omega^4 = \omega^n + 5$$

$$\vdots$$

$$\omega^2 n = \omega^n - 1$$

das ist die schnelle Fourier-Transformation die unter anderem für KatzenBilder im Internet benutzt wird. Aber auch für Mobilfunk und Quantencomputer.

2. Algorithmen

2.1. **Definition.** Ein Rechenverfahren, das eine Eingabe in eine Ausgabe überführt und

- endlich beschreibbar ist,
- effektiv ist,
- allgemein ist,
- deterministisch ist.[3]

2.2. **Qualität.** Ein Algorithmus ist gut, wenn:

- er korrekt (terminiert immer und gibt richtige Antwort) ist,
- er schnell ist,
- er speichereffizient ist,
- gut lesbar/einfach zu verstehen ist.

[3]Das ist nicht zwingend notwendig

2.3. **Bewertung.** (wie findet man heraus dass ein Algorithums "gut" ist?)

Experimentell: problematisch, nicht bestandteil dieser Vorlesung

Theoretisch:
- abstrakte Beschreibung des Algorithmus im Pseudocode, in einem abstrakten Maschinenmodell.
- Darstellung der Laufzeit als Funktion der Eingabegröße
- Wir konzentrieren uns auf die schlimmstmögliche Eingabe (worst-case)

2.3.1. *Theoretische Bewertung.* von Algorithmen Um einen Algorithmus theoretisch zu analysieren, zählen wir "Schritte" und benutzte "Speicherzellen". Dazu benötigen wir ein $\Rightarrow$ Abstraktes Maschinenmodel.

2.4. **abstraktes Maschinenmodel.** Es gibt Viele Arten. z.B.:

- Turingmaschine
- λ-Kalkül
- Markov-Algorithmus

2.4.1. *die RAM.* (Registermaschine RandomAccessMachine) [4]

Speicher: unendlich viele Speicherzellen, nummeriert von $0, 1, 2, 3, \dots$ jede Zelle speichert eine ganze Zahl

CPU: unterstützt

- Arithmetik $(+, - + \cdot, geteilt, \mod, \dots)$
- Sprünge (JUMP,JNE,JG,...)
- indirekte Adressierung
- Move (Kopieroperation)

<u>26.10.2010</u>

2.5. **Analyse.** von Algorithmen

2.5.1. *Definition.* Ein Algorithmus A ist ein Programm für eine RAM.
Sei I eine Eingabe für A.
$T_A(I)$:= Anzahl der Schritte, die A bei Eingabe I ausführt (Laufzeit).
$S_A(I)$:= Anzahl der Speicherzellen, die A bei Eingabe I liest/schreibt (Speicherbedarf).
$T_A(I) := \max\limits_{I,|I|=n} T_A(I)$ (worst-case Laufzeit)
$S_A(I) := \max\limits_{I,|I|=n} S_A(I)$ (worst-case Speicherbedarf)

[4] bild eines Registers mit CPU

2.5.2. *Beispiel.* Finde das Maximum in einem Array

$\emptyset$	1	2	3	4
Größe	A[1]	A[2]	A[3]	...

MOVE $\emptyset \rightarrow [\emptyset] + 1$ schreibe Array hinter die Eingabe

MOVE $[\emptyset] \rightarrow [\emptyset] + 2$ initialisiere das aktuelle max

Loop: DEC $[\emptyset] + 1$ Restanzahl $--$

JZ $[\emptyset]+, Ende$ Nix mehr $\rightarrow$ Ende

JLE $[[\emptyset] + 1], [\emptyset] + 1, SKIP$ Aktuelles Element $\leq$ aktuelles Max; JLE=Jump Less Equal

MOVE $[[\emptyset] + 1] \rightarrow [\emptyset] - 2$ neues Max merken

Skip: JMP Loop

Ende: Return $[[\emptyset] + 1]$

Jetzt stelle man sich lustiges vor der Tafel rumgehüpfe vor. Einer zeigt, einer zählt und einer sagt was gemacht werden soll. Grandios!

Analyse:
$T_A(3; 2; 3; 1) = 14$
$S_A(3; 2; 3; 1) = 6$
im Allgemeinen:
$T_A(I) = 3 + 2n + 2(n - 1)+$(Anzahl der Änderungen von Max)
$T_A(n) \leq 3 + 2n + 2(n - 1) + n - 1 = 5n$ (worst-case; absteigendes Array)

dies ist der "Knuth-style" der Algorithmenanalyse

Fazit:
RAM liefert uns eine exakte, maschinenunabhängige Definition für die Laufzeit eines Algorithmus, ist aber anstrengend (mühsam) und schwer zu verstehen (der Code).
Deswegen: Im Folgenden werden wir Algorithmen im Pseudocode beschreiben, um sie besser verständlich zu halten, aber wir behalten die RAM im Hinterkopf.
Pseudocode="weißt schon"

Der Algorithmus im Pseudocode:

```
ArrayMax (length,A)
curMax <- A[length]
For counter := length-1 downto 1
        if A[counter] > curMax
                curMax <- A[counter]
return curMax
```

Eine Zeile Pseudocode soll konstant vielen RAM Anweisungen entsprechen.
Konsequenz: Beim Zählen der Schritte kommt es auf konstante Faktoren nicht an.
Mathematisch formal drücken wir das durch die O-Notation aus.

2.6. **O-Notation.**

2.6.1. *Definition.* Sei $f, g : \mathbb{N} \to R^+$
Dann $F = O(g) \iff \exists c > 0, n_0 \geq 1 : \forall n \geq n_0 : f(n) \geq c \cdot g(n)$
$f = O(g) \iff \exists c > 0, n_0 \geq 1 : \forall n \geq n_0 : f(n) \leq c \cdot g(n)$
$f = \Omega(g) \iff \exists c > 0, n_0 \geq 1 : \forall n \geq n_0 : f(n) \geq c \cdot g(n)$
$f = \Theta(g) :\Rightarrow f = O(g)$ und $f = \Omega(g)$

2.6.2. *Beispiel.*

$5n = \Theta(n)$

$n^2 - \sqrt{n} + 100n - \log n = O(n^2)$

Regeln:

$\log^\alpha n = O(n^\beta)\alpha, \beta \geq 0$
$\log^1 000n = O(n^0, 00001)$
$n^\alpha = O(n^\beta)0 < \alpha \geq \beta$
$\sum_{i=0}^{d} a_i n^i = \Theta(n^d)$, wenn $\alpha_d > 0$

2.6.3. *Vorteil.* der O-Notation: Sie ermöglicht es die Laufzeit knapp, prägnant und unabhängig von den Details des Modells anzugeben.

2.6.4. *Nachteil.* : Kann irreführen sein; niemand sagt, was c und n_0 sind. Außerdem können sie riesig sein. Die O-Notation macht eine asymptotische (gegen unendlich) Aussage über die Laufzeit eines Algorithmus für große Eingaben. Ignorieren Konstanten und Therme niedriger Ordnung.

3. Rekursion

28.10.2010

Zerlege ein Problem in kleine Teilprobleme. Löse die Teilprobleme und setze die Teillösungen zur Gesamtlösung zusammen.

Devide and Conquer/Teile und Herrsche

3.0.5. *Beispiele.*

- Fibonaccizahlen
- Mergesort
- Quicksort
- binärer Baum
- Fakultät
- Türme von Hanoi

3.1. **Analyse.** einer Rekursion am Beispiel von Fibonaccizahlen:

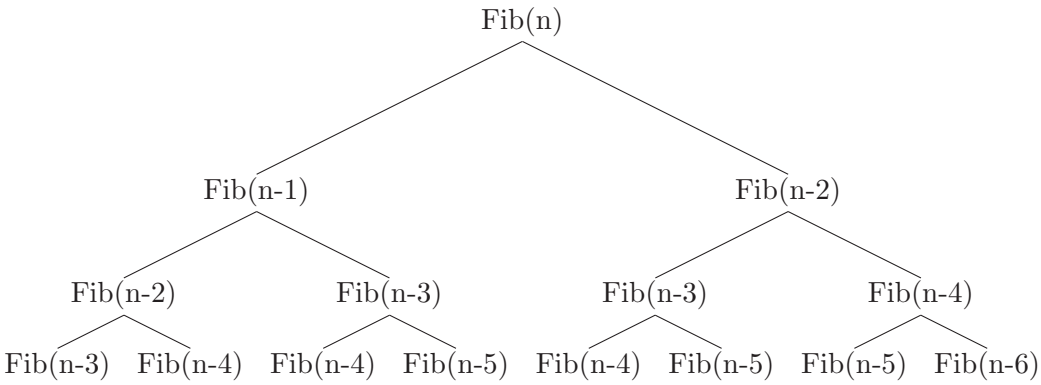
Fibonaccizahlen
$F(0) = F(1) = 1$
$F(0) = F(n-1) + F(n-1)$ für $n \geq 2$

```
Fib(n)
        if n=0 or n=1 then
                return 1
        else
                return fib(n-1)+fib(n-2)
```

Wie lange dauert es, Fib(n) zu berechnen?

3.1.1. *Analyse:* Visualisiere die Rekursion als Baum

- Ein Kntoen für jede Inkarnation der Funktion
- Kanten zwischen aufgerufener und aufrufender Inkarnation
- Schreibe an jeden Knoten den Aufwand für die Inkarnation



Daraus folgen zwei Fragen:

1) Wieviele Ebenen existieren?

2) Wie hoch ist der Aufwand in Ebene i (wenn $i = 0$ die erste Ebene ist)?

Ergebnisse:

1) Die Tiefe der Blätter ist zwischen n und $\frac{n}{2}$
   Es gibt $\frac{n}{2}$ volle Ebenen

2) Wenn $i \leq \frac{n}{2}$ ist der Aufwand $= 2^i$
   Wenn $n \geq i > \frac{n}{2}$ ist der Aufwand $\leq 2^i$

Zum Abschätzen des Aufwands addieren wir den Aufwand/Ebene über die Ebenen
$T(n) \geq \sum_{i=0}^{\frac{n}{2}-1} 2^i = 2^{\frac{n}{2}} - 1 = \Omega(2^{\frac{n}{2}})$
$T(n) \leq \sum_{i=0}^{n-1} 2^i = 2^n - 1 = O(2^n)$
Fazit: schlechter Algorithmus

3.1.2. *Optimierung:* Speichern der Werte (in einer Tabelle) um doppelter Berechnung vorzubeugen.

- Initialisiere ein Array F der Länge n mit -1 überall.
- `fib(n)`

```
if n=0 or n=1 then
        return 1
else if f[n] == -1 then
        f[n] = fib(n-1)+fib(n-2)
return f[n]
```

Laufzeit: $O(n)$
Speicher: $O(n)$

Dynamisches Programmieren: Verbindung von Rekursion und einer Tabelle zum speichern wiederkehrender Zwischenergebnisse

3.1.3. *Noch besser:*

```
Fib(n)
        if n=0 or n=1 then
                return 1
        else
                (a,b) <- (1,1)
                for i=2 to n
                        (a,b) <- (a+b,a)
                return a
```

O(n) Laufzeit Speicher O(1)

Im Allgemeinen: Ohne Verwendung von Rekursion ist ein Algorithmus im Allgemeinen schneller. Durch Verwendung von Rekursion ist der Code jedoch besser lesbar.

4. einfache Datenstrukturen

4.1. **LIFO.** (Stack,Stapel,Keller)

4.1.1. *Operationen.*

- **PUSH (x)** lege x auf den Stapel
- **POP** Entfene oberstes Element vom Stapel und gib es zurück
- **SIZE** gibt die Anzahl der Elemente auf dem Stapel zurück
- **ISEMPTY** Stapel leer?
- **TOP** Oberstes Element vom Stapel (ohne Entfernen)

4.1.2. *Implementiereung.*

1) als Array (mit fester Länge)
   Vorteile:
   - sehr einfach
   - sehr schnell (O(1)/Operation)

   Nachteile:
   - feste Feldlänge
2) Verkettete Liste (Elemente mit Zeiger auf das nächste Element.)
   Vorteile:
   - schnell (O(1)/Operation)
   - flexible Größe der Elemente

   Nachteile:
   - kompliziert(er)
   - Cacheperformance
3) Array (mit dynamischer Länge)
   Wie Array (fest), aber wenn das Array voll ist, legen wir ein neues Array doppelter Größe an und kopieren das alte Array dort hin.
   Vorteile:
   - einfach
   - flexibel

   Nachteile:
   - schnell? (Operation kann O(n) dauern)

4.2. **FIFO.** (Schlange)

- **ENQUEue (x)** Füge x ein
- **DEQUEUE** entferne ältestes Element und gib es zurück
- **SIZE** äquivalent LIFO

- **ISEMPTY** äquivalent LIFO
- **FRONT** äquivalent LIFO

5. Armortisierte Analyse

02.11.2010

Keiner von uns war in der Vorlesung

Wiederholung:

- Armotisierte Analyse
  - wenn wir zeigen können, dass jede Folge von m Operationen f(n) Zeit benötigt, dann definieren wir die armortisierten Kosten pro Operation sind $\frac{f(n)}{m}$
  - Beispiel: PUSH für Stapel in dynamischem Array hat armortisierte Kosten $O(1)$
- abstrakter Datentyp (ADT)
  - ein Typ, dessen Objekte nur über eine klar spezifizierte Schnittstelle manipuliert werden können
- ADT in Java
  - durch Vererbung und Kapselung
  - Interface für die Schnittstelle und konkrete Klassen für die Implementierung
  - Bsp.: interface Queue$< E >$
    public class ArrayQueue
    public class ListQueue
- deklariere Variablen vom Interface-Typ und verwende konkrete Klassen nur für die Objekterzeugung
  Bsp.: Queue$< Integer >$ q=new ListQueue$< Integer >$

04.11.2010

Ein Problem dabei: Wir wollen Queue um einfache Prioritäten erweitern.

```
enum Prio{LOW,NORMAL,URGENT}
public interface LNUQueue<E> extends Queue<E>{
        void enqueue(E element,Prio p) throws QueueFullException
        void reset();
}
```

- default Priorität: NORMAL
- wir wollen das älteste Element höchster Priorität
- reset setzt alles zurück bzw. löscht alle Queues und legt den Datentyp neu an

Idee: Schreibe eine neue Klasse mit drei Queues (low,normal,urgent) und frage die Queues entsprechend ab. Wie macht man das am geschicktesten?
Ziel: Zum Schluss wollen wir ArrayLNUQueue, ListLNUQueue und DanymicLNU-Queue haben.
Problem: Wir müssen Code replizieren. Das ist so nicht redundant. Lösung: Verwende/definiere eine abstrakte Oberklasse für das Queue-Management.

```
public abstract class AbstractLNUQueue<E>
        implements LNUQueue<E>{
                private Queue<E> low, normal, urgent;
                public void enqueue(E element, Prio p) throws ...{
                        switch (p){
                                case URGENT: urgent enqueue(element); break;
                                case NORMAL: normal ...
                        }
                }
                void reset() {
                        low = buildQueue();
                        normal = buildQueue();
                        urgent = buildQueue();
                }
                abstract protected Queue<E> buildQueue();
        }
public class ListLNUQueue<E> extends AbstractLNUQueue<E>{
        protected Queue<E> buildQueue(){
                return new ListQueue<E>();
        }
        public ListLNUQueue(){
                reset();
        }
}
```

An dieser Stelle kommt ein Bild, dass das ganze erklärt.

Das ist das Entwurfsmuster "factory method".

Def: Entwurfsmuster (engl. design pattern) ist eine Allgemeine und wiederverwendbare Lösung für ein wiederkehrendes Problem im Softwareentwurf.
In der OOP(objektorientierte Programmierung): bestimmte Form von Beziehungen und Interaktionen von Objekten.
Entwurfsmuster repräsentieren Erfahrungen und "best practices" des Softwareentwurfs. Sie bieten eine Terminologie um über Klassenentwurf nachzudenken und kommunizieren.

6. Prioritätswarteschlange

Erweiterung der Queue um "beliebige" Prioritäten

6.1. **Anwendungen.**

- scheduling

- Terminplanung
- in Algorithmen: kürzeste Wege, minimal aufgespannte Bäume, Sweepline

6.2. **Spezifikation.** Wie spezifizieren wir einen ADT? d.h. wie geben wir an, was die Operationen tun sollen?

6.2.1. *verbal.* Prioritätswarteschlange speichert Elemente aus einem total geordneten Universum
Operationen:

- **findMin()**: gib das kleinste Element in der Datenstruktur zurück, lasse die Datenstruktur unverändert. Wenn die Datenstruktur leer ist, wirf eine Exception.

  Effekt gib ein kleinstes Element aus der Datenstruktur zurück, lass die Datenstruktur unanagerührt

- **deletMin()**: wie findMin(), aber löscht auch das zurückgegeben Element aus der Datenstruktur
- **insert**: Var

  Effekt füge ein Element in die Datenstruktur ein

- **isEmpty**: Var

  Effekt gib true zurück, wenn Datenstruktur leer

- **size**: Var

  Effekt gib die Anzahl der Elemente in der Datenstruktur zurück

Gegenenfalls kann man noch eine Invariante [5] für die Datenstruktur angeben
Vorteile:

- leicht verständlich
- allgemein
- einfach

Nachteile:

- viel Text
- kann unterspezifiziert sein
- wenig formal, eignet sich nicht gut für Beweise und Computerprogramme

09.11.2010

6.2.2. *modellierend.* gib ein abstraktes Modell an, definiere die Operationen anhand des Modells Beispielsweise:

a) mathematisch: Sei U ein total geordnetes Universum
   Objekte: endliche Multimengen $S \subset U$
   Operationen:
   (a) findMin: $S \neq \emptyset$

[5] Bedingung, die immer gilt (vor und nach jeder Operation)

b) mit Haskell
Objekte: (Ord t)=¿[t]
Operationen: findMin :: (Ord t)=¿[t]-¿[t,[t]]
findMin xs Var: deleteMin(Ord t)=¿[t]-¿(t,[t])
deleteMin xs Vor xs/=[]
=(m,ys) where m=minimum xs ys=(takeWhile(¿m)xs)++tail(dropwhile(¿m)xs)
insert::(Ord t)=¿(t,[t]-¿[t])
insert(x,xs)=x:xs
isEmpty::(Ord t) = ¿ [t]- ¿ (Bool, [t])
isEmpty xs=(xs==[],xs)
size:(Ord t) = ¿ [t - ¿ (Int, [t])] size xs = (lenght xs,xs)

6.2.3. *algebraische Spezifikation.* Kein Modell, sondern wir geben axiomatische Beziehungen zwischen den Operationen an, die gelten müssen.

7. Prioritätswarteschlange

Hier fehlt ein Teil.

Wiederholung:

- ADP Prioritätswarteschlange, Implementierung
  (1) unsortierte verkettete Liste
  (2) sortierte verkettete Liste

11.11.2010

7.1. **Prioritätswarteschlange als verkettete Liste mit zwei Ebenen.**

- Fixiere einen Parameter m
- Speichere die Elemente als Folge von verketteten Listen, so dass alle Listen bis auf eine die Länge =m haben. Die letzte Liste hat die Länge $\leq m$.
- verlinke die Minima der einzelnen Listen
- wenn die Struktur n Elemente enthält, dann ist die Anzahl der Listen $l = \frac{n}{m}$

insert(x):
- füge x in die kürzeste Liste ein (lege ggf eine neue Liste an)
- aktualisiere das Minimum und die Min-Liste, falls nötig
- Laufzeit: $O(m)$ naiv, $O(1)$ mit besserer Implementierung

findMin:
- durchlaufe die Min-Liste und gib das kleinste Element zurück
- Laufzeit: $O(l)$ naiv, $O(1)$ mit besserer Implementierung

deleteMin:
- finde das Minimum wie bei findMin
- lösche das Minimum aus der Liste
- fülle diese Liste ggf mit einem Element aus der kurzen Liste auf
- aktualisiere die Min-Liste
- Laufzeit: $O(l + m)$

size,isEmpty: – zähle die Anzahl der Elemente in einem Attribut

– Laufzeit: $O(1)$

Wenn N=max Anzahl an Elementen in der Priority-Queue bekannt ist, können wir $m = \sqrt{N}$ wählen. ⇒ Alle Operationen benötigen Zeit $O(\sqrt{N})$

8. binärer Heap

- ein vollständiger binärer Baum mit heap-Eigenschaft
- wird oft als Array implementiert
- alle Ebenen bis auf die letzte sind voll besetzt
- die letzte Ebene ist von links aufgefüllt
- Elemente sind im Knoten gespeichert
- ein Knoten a ist immer kleiner/gleich seine Kinder b,c
- Das Minimum ist in der Wurzel

An dieser Stelle kommt die Spezifikation. Die schreibe ich nicht mit, da das im Tutorium behandelt wurde.

9. Wörterbuch

Erlaubt es Einträge zu speichern, nachschlagen und zu löschen.
Seien K,V (Key, Value) zwei Mengen. Die Objekte sind endliche Teilmengen $S \subseteq KxV$

9.1. **Operationen.**

put(k,v)

Vorr.: -

Effekt: entferne den bisherigen Schlüssel aus K und füge ihn neu ein

get(K)

Vorr.: es existiert dieser Schlüssel

Effekt: gib V zurück

remove(k)

Vorr.: es existiert dieser Schlüssel

Effekt: lösche den Schlüssel

16.11.2010

9.2. **Implementation.** Wie implementiert man ein Wörterbuch?

9.2.1. *Warm-Up.* Angenommen: $K = \{1, 2, \ldots, 100\}$. Sei elements ein Array der Länge 100

- put(K,V) $elements[K] \leftarrow V$
- get(K) *return* $elements[K]$
- remove(K) $elements[K] \leftarrow \perp$

Laufzeit: $O(1)$ für jede Operation

9.2.2. *Wieso funktioniert das nicht für allgemeines K?.*

- K kann sehr groß sein
- K muss nicht aus Zahlen bestehen

Lösungsansatz: finde eine "gute" Funktion $h : K \to \{0, \ldots, N-1\}$ welche jedem Schlüssel eine Position im Arrayelements zuweist (N ist die Länge des Arrays)
h heißt Hashfunktion[6]
Angenommen, wir haben h: Wie implementieren wir das Wörterbuch? Versuch: wir ersetzen in unserem warm-up jedes K durch ein h(K).
put(K,V) $elements[h(K)] \leftarrow V$
get(K) *return* $elements[h(K)]$
remove(K) $elements[h(K)] \leftarrow \perp$
Problem: Wenn $|K| > N$, dann existieren zwei Schlüssel K_1 und K_2 mit $\underbrace{h(K_1) = h(k_2)}_{Kollision}$
Lösung: es gibt 3 Möglichkeiten für eine Strategie zur Kollisionsbehandlung

(1) Verkettung: elements[i] speichert alle Einträge (K,v) mit $h(K) = i$ (in einer verketteten Liste)

(2) offene Adressierung: wenn bei put(K,V) elements[h(k)] schon besetzt ist, finde einen anderen Platz

(3) Kuckuck[7][8]: wenn bei put(K,V) elements[h(k)] schon besetzt ist, schaffe Platz

Eine Hashtabelle[9] besteht aus einem Array der Länge N, einer Hashfunktion $h : k \to \{0, \ldots, N-1\}$ und einer Strategie zur Kollisionsbehandlung.

9.2.3. *Wahl der Hashfunktion.*
Ziel weise jeden Schlüssel $k \in K$ einen Wert in $[0, \ldots, N-1]$ zu. Um Kollisionen gering zu halten soll h möglichst "zufällig"(bzw. gleichverteilt) sein, d.h. eventuel vorhandene Struktur in der zu speichernden Schlüsselmenge soll zerstört werden.
$h(K)$ wird in zwei Schritten berechnet:

- berechne einen Hashcode für k, d.h. weise k eine ganze Zahl zu.

  Bsp: Wenn K=int: tue nichts, gib k zurück
  Wenn K=float: multipliziere mit Zehnerpotenz und ggf. runden
  oder nim die IEEE-Darstellung

[6] to hash: zerhacken
[7] Kuckucks sind zu faul Kinder großzuziehen und lassen das andere machen
[8] Das Verfahren gibt es erst seit 6 Jahren
[9] Streuspeicher

K=char: Unicode/ASCII
K=String: Addiere die Zahlen der einzelnen Zeichen (s.O.) (nicht eindeutig)
oder interpretiere das Wort einfach als Zahl
In Java: Alle Java-Objekte bieten eine Funktion `hashCode()`, die einen Hashcode berechnet. Diese Funktion ist bei eigenen Klassen ggf. sinnvoll zu überschreiben.

9.2.4. *Kompressionsfunktion.* Bilde den Hashcode auf $[0, \ldots, N-1]$ ab. Die Kopmpressionsfunktion soll die Eingabe möglichst gut streuen, um die Wahrscheinlichkeit von Kollisionen gering zu halten.

Bsp: $z \to z \mod N$ (einfach, aber streut nicht gut[10])
besser: Sei p eine Primzahl, $p > n$ $z \to (z \; mod \; p) mod \; N$ (heuristisch besser)
noch besser: Sei $p > N$ Primzahl; Wähle zufällig $a, b \in \{0, \ldots, p-1\}; a \neq 0$:
$z \to ((az + b) \mod p) \mod N$

Also: $\underbrace{k \in K \underbrace{\to}_{hashCode} z \in \mathbb{Z} \underbrace{\to}_{Kompressionsfunktion} \{0, \ldots, N-1\}}_{h}$

Wir werden annehmen, dass sich h in $O(1)$ Zeit berechnen lässt.

9.2.5. *Kollisionsbehandlung: Verkettung.* [11] elements[i] speichert eine Liste, welche alle Einträge (K, v) mit $h(K) = I$

put(k,v): Durchsuche elements[h(k)] nach k. Falls vorhanden, setze Eintrag auf (k,v), sonst füge (k,v) an die Liste an.

get(k): Durchsuche elements[h(k)] nach k. Gib ggf den Wert zurück oder wirf eine exception

delete(k): Lösche den Eintrag für k aus element$[h(K)]$

Platzbedarf: $O(N + Anzahl(Eintraege))$

18.11.2010

Laufzeit:

Annahme: h verhält sich wie eine zufällige Funktion, d.h. für alle $k \in K, i \in \{0, \ldots N-1\}$: $Pr[h(k) = i] = \frac{1}{N}$ unabhängig von den anderen Schlüsseln

Vorstellung: Beim Anlegen der Hashtabelle wird h zufällig gewählt und diese Wahl ist geheim.

Sei S die aktuelle Menge von Einträgen in der Hashtabelle und sei k ein fester Schlüssel, auf dem die nächste Operation ausgeführt wird.

Daraus ergibt sich die Laufzeit: $O(\underbrace{1}_{\text{Berechnung von } h(k)} + \underbrace{L}_{\text{Länge der Liste bei } elements[h(k)]})$

[10] zerhackt eventuell vorhandene Struktur nicht gut
[11] engl. chaining

Für $K' \neq K$ sei $L_{K'} = \{1, \text{falls } h(K') = h(k); 0 \text{ sonst}\}$, dann gilt:

$$L \leq 1 + \sum_{(K',v) \in S, K' \neq K} L_{K'}$$

$$E_h[L] \leq E_h[1 + \sum_{(K',v) \in S, K' \neq K} L_{K'}]$$

$$= E_h[1] + \sum_{(K',v) \in S, K' \neq K} E[L_{K'}]$$

$$= 1 + \sum_{(K',v) \in S, K' \neq K} Pr[h(k') = h(k)]$$

$$= 1 + \sum_{(K',v) \in S, K' \neq K} \frac{1}{N}$$

$$\leq 1 + \frac{|S|}{N}$$

Die erwartete Laufzeit ist $O(1 + \frac{|S|}{N}) = O(1)$

Bemerkung: $\frac{|S|}{N}$ heißt Belegungsfaktor[12]. Wenn der Belegunsfaktor $O(1)$ ist, so liefert Hashing mit Verkettung erwartete Laufzeit $O(1)$. In der Praxis will man einen Belegungsfaktor von ca. 0,75 halten (durch ein dynamisches Array). Wir haben angenommen, dass h zufällig gewählt wurde. Das ist jedoch unpraktikabel. Aber: h kann zufällig aus einer viel kleineren Menge von Funktionen gewählt werden und der Beweis klappt weiterhin[13].

Vorteile	Nachteile
einfach	verschwendet Platz
schnell (im Erwartungswert)	

```
put(k,v)
  Durchsuche die verkettete Liste bei
  elements[h(k)] nach k. Ersetze den
  Wert durch v, bzw. lege einen neuen
  Eintrag an.

get(k)
  Durchsuche die verkettete Liste bei
  elements[h(k)] nach k. Gib den
  Wert fuer k zurueck, bzw. wirf eine
  NoSuchElementException.

remove(k)
  Durchsuche die verkettete Liste bei
  elements[h(k)] nach k. Loesche den
  Eintrag fuer k, bzw. wirf eine
  NoSuchElementException.
```

[12] en. load factor

[13] Das ist auch praktikabel und heißt universelles Hashing

9.2.6. *Kollisionsbehandlung: offene Adressierung.* Wenn $elements[h(k)]$ bereits belegt ist, suche eine neue Position[14] Diese Methode heißt lineares Sortieren[15]. Vorsicht beim Löschen: eine Lösung ist einen speziellen Platzhalter `AVAIL` zu verwenden, der freie Einträge anzeigt. $|S| \leq N$ muss gelten (brauchen dynamisches Array)

23.11.2010

Fakt: Wenn der Belegungsfaktor < 0.9 ist und h zufällig gewählt ist, so benötigen alle Operationen $O(1)$ erwartete Zeit.

Achtung: Durch Löschen kann die Tabelle vermüllt werden. → Baue die Tabelle in regelmäßigen Abständen neu aus ($O(1)$ amortisierte erwartete Laufzeit)

Vorteil: geringere Platzbedarf

Nachteil:

- löschen
- Klumpenbildung

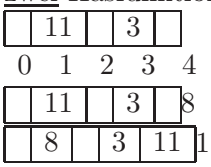
```
put(k,v)
  pos <- h(k)
  for i := 1 to N do
    if (elements[pos] == NULL || elements[pos] == DELETED || \\
elements[pos].k == k)
      elements[pos] <- (k,v)
      return
    pos <- (pos + 1) mod N
  throw TableFullException

get(k)
  pos <- h(k)
  for i := 1 to N do
    if (elements[pos] == NULL)
      throw NoSuchElementException
    if (elements[pos].k == k)
      return elements[pos].v
    pos <- (pos + 1) mod N
  throw NoSuchElementException

remove(k)
  pos <- h(k)
  for i := 1 to N do
    if (elements[pos] == NULL)
      throw NoSuchElementException
    if (elements[pos].k == k)
      elements[pos] <- DELETED
      return
    pos <- (pos + 1) mod N
```

[14] dabei wird der nächste freie Platz verwendet

[15] linear probing

```
  throw NoSuchElementException
```

9.2.7. *Kollisionsbehandlung: Kuckuck.* Wenn `elements[h(K)]` besetzt ist, schaffe Platz
Problem Wohin mit dem alten Element?
Lösung Jeder Eintrag hat zwei mögliche Positionen in der Tabelle → wir verwenden zwei Hasfunktionen.




```
get(k)
  if (elements[h1(k)].k == k)
    return elements[h1(k)].v
  if (elements[h2(k)].k == k)
    return elements[h2(k)].v
  throw NoSuchElementException

remove(k)
  if (elements[h1(k)].k == k)
    elements[h1(k)] <- NULL
    return
  if (elements[h2(k)].k == k)
    elements[h2(k)] <- NULL
    return
  throw NoSuchElementException

put(k,v)
  if (elements[h1(k)].k == k)
    elements[h1(k)] <- (k,v)
    return
  if (elements[h2(k)].k == k)
    elements[h2(k)] <- (k,v)
    return
  if (|S| == N)
    throw TableFullException
  pos <- h1(k)
  for i := 1 to N do
    if (elements[pos] == NULL)
      elements[pos] <- (k,v)
      return
    (k,v) <-> elements[pos]
    if (pos == h1(k))
      pos <- h2(k)
    else
      pos <- h1(k)
  Waehle neue Hashfunktionen und baue die Tabelle neu auf.
```

```
put(k,v);
```

`get, remove` brauchen $O(1)$ Zeit im worst-case

Fakt: Wenn der Belegungsfaktor klein genug ist und h_1, h_2 zufällig gewählt sind, so braucht `put(K,v)` $O(1)$ erwartete amortisierte Zeit.

9.3. **Iterator.** (Entwurfsmuster)

9.3.1. *Problem.* Wir wollen alle Einträge in einem Wörterbuch effizient durchlaufen und weiterhin das Geheimnisprinzip wahren (Zugriff nur durch eine Schnitstelle)

9.3.2. *Idee.* Füge Operationen zum Wörterbuch hinzu, welche das Durchlaufen unterstüzen (`firstElement(), nextElement()`)

9.3.3. *Probleme.*

- Können nicht mehrere Durchläufe gleichzeitig durchführen
- das Interface wird zugemüllt
- unflexibel

9.3.4. *Lösung.*

- Lagere das Durchlaufen in eine eigen Klasse aus[16]

25.11.2010

Wiederholung:

- Der ADT Wörterbuch: `put(K,v),get(K),remove(K)`
- Implementation durch Hashtabellen: alle Operationen in $O(1)$ erwarteter Zeit (ggf amortisiert)

10. geordnetes Wörterbuch

10.1. **Definition.**

K Schlüsselmenge, total geordnet

V Werte

Unser Ziel ist eine Teilmenge $S \subseteq KxV$ zu speichern
Operationen: `put,get,remove` wie bei ADT Wörterbuch
Zusätzlich:

- **min()**: bestimme den minimalen Schlüssel in S
- **max()**: bestimme den maximalen Schlüssel in S
- **succ(K)**: bestimme den kleinsten Schlüssel K' in S min $k' > K$
- **pred(K)**: bestimme den größten Schlüssel K' in S mit $K' < K$

[16] wird von Java unterstützt z.B. `for e:...`

10.2. **Implementierung.**

- Hashtabelle ist ungeeignet, weil `succ,pred` brauchen $O(n)$ Zeit
- alternativ: eine sortierte verkettete Liste → nicht gut, da $\Theta(n)$ worst-case Laufzeit
- Idee: Speichere eine Hierarchie von sortierten verketteten Listen

10.2.1. *Beispiel.* [17]

$L_3 = \infty \rightarrow$ 20
$L_2 = \infty \rightarrow$ $9 \rightarrow$ 20
$L_1 = \infty \rightarrow$ $4 \rightarrow$ $9 \rightarrow$ $12 \rightarrow$ 20
$L_0 = \infty \rightarrow 3 \rightarrow 4 \rightarrow 7 \rightarrow 9 \rightarrow 10 \rightarrow 12 \rightarrow 15 \rightarrow 20$

10.2.2. *Allgemein:*

L_0 speichert alle Einträge
L_1 speichert jeden 2. Eintrag
L_j speichert jeden 2^j. Eintrag
$\Longrightarrow$ wenn $|S| = n$, so brauchen wor $O(\log n)$ Listen

Suche nach einem Schlüssel K: (`pred,succ,get`)
Beginne an der obersten Liste; gehe waagerecht(nach rechts), bis der Nachfolger größer dem gesuchten und senkrecht (nach unten) falls nicht.
Laufzeit: $O(\log n)$ Ebenen $\cdot O(1)$ Vergleiche per Ebene $= O(\log n)$
Problem: Wie erhalten wir die Struktur beim Einfügen und Löschen?
Lösung: Gar nicht; Schwäche die Bedingung ab durch Nutzung des Zufalls. Ordne jedem Element eine Zufällige Höhe[18] zu. Die Hoffnung ist gut genug.

```
put(K,v):
        - suche nach K
                - falls vorhanden: aktualisiere den Wert.
                  Speichere den Suchpfad auf einem Stack
                - falls k nicht vorhanden:
                        - wirf eine faire Muenze bis zur ersten Zahl
                        - j:= Anzahl Wuerfe -1
                        - fuege K in die Liste L_0,L_1,...L_j ein
                          (lege ggf. neue Listen an)
                          (verwende den Stack,
                          um die Einfuegepositionen zu finden)
                        - verkette die neuen Knoten vertikal
get(K): wie vorher, ebenso pred und succ
remove(K):
        - suche K, loesche K aus allen Listen, die es enthalten
```

Analyse:

[17] Gleiche Schlüssel sind jeweils nochmal miteinander verkettet.
[18] =Anzahl der Listen, welche das Element enthalten

Platzbedarf : $O(n)$ erwartet

max Höhe : $O(\log n)$ mit hoher Wahrscheinlichkeit

Suchzeit :

Satz: Sei $S \subseteq KxV$ mit $|S| = n$ in einer Skipliste gespeichert und sei $k \in K$, dann benötigt die Suche nach K $O(\log n)$ Schritte im Erwartungswert.
Beweis: Sei $i \in \mathbb{N}_0$, sei T_i =Anzahl der Schritte in L_i auf dem Suchpfad hoch K.
$T_i = 0$, falls $L_i = \emptyset$
E[Suchzeit]=E$[\sum_{i=0}^{\infty} T_i]$=$\sum_{i=0}^{\infty} E[T_i]$

Ziel: Schätze $E[T_i]$ ab. Fallunterscheidung.

Fall 1: $i > \log n$, d.h. $i = \log n + j$ für $j \geq 1$
$E[T_i] \leq E[|L_i|]$[19]
für $s \in S$, definiere $X_s = 1$ falls $s \in L_i; 0,$ sonst
$E[|L_i|] = E[\sum_{s \in S} X_s] = \sum_{s \in S} E[X_s]$
$E[x_s] = Pr[s \in L_i] = \sum_{l=i}^{\infty} \frac{1}{2^{l+1}} = \frac{1}{2^i}$
daraus folgt
$\sum_{s \in S} E[X_s] = \frac{n}{2^i} = \frac{1}{2^j}$

30.11.2011

Fall 2: $O \leq i \leq \log n$
$L_i + 1 \quad \updownarrow$
$L_i \quad O \leftrightarrow O \leftrightarrow O \leftrightarrow O$
$\updownarrow$
Betrachte die Schritte L_i rückwärts. Wir laufen nach links, bis der aktuelle Knoten auch in $L_i + 1$ vorhanden ist. Das passiert mit Wahrscheinlichkeit $\frac{1}{2}$.
Also: Mit Wahrscheinlichkeit $\frac{1}{2}$ machen wir 1 Schritte.
Mit Wahrscheinlichkeit $\frac{1}{4}$ machen wir 2 Schritte.
Mit Wahrscheinlichkeit $\frac{1}{8}$ machen wir 3 Schritte.
Mit Wahrscheinlichkeit $\frac{1}{2^2}$ machen wir j Schritte.
$E[T_i] = \sum_{j=1}^{\infty} \frac{j}{2^j} = 2$
Also:

$$\begin{aligned} E[\text{Suchzeit}] = \sum_{i=0}^{\infty} E[T_i] &= \sum_{i=0}^{\log n} E[T_i] + \sum_{i=\log n+1}^{\infty} E[T_i] \\ &= \sum_{i=0}^{\log n} 2 + \sum_{j=1}^{\infty} \\ &= 2\log n + 2 + 1 = O(\log n) \end{aligned}$$

Vorteile	Nachteile
einfach	Laufzeit ist nicht `worst case`
gute Laufzeit (im EW)	brauchen Platz für die zusätzlichen Listen

[19] Anzahl der Knoten in L_i

10.3. **alternative Implementierung: Binäre Suchbäume.** eine andere Implementierung des ADT geordnetes Wörterbuch

Baum: Kreisfreier zusammenhängender Graph

gewurzelter Baum: Baum, in dem ein Knoten als Wurzel ausgezeichnet wurde

Höhe eines gewurzelten Baumes: Länge (# Kanten) eines längsten Weges von einem Knoten zur Wurzel

Blatt: Knoten ohne Kinder

binärer Baum: alle Knoten ≤ 2 Kinder

Implementierung des geordneten Wörterbuch aus binären Baum

- speichere die Einträge in den Konoten des Baumes
- jeder Knoten hat 2 Kinder

  ein linkes Kind und ein rechtes Kind

  diese können jewiels NULL (bzw. $\perp$) sein
- binäre Suchbaumeigenschaft bekannt... [20]

Was wissen wir über die Höhe des Baums?

h ist mindestens $(\log n)$

h ist höchstens $n - 1$

kann passieren: z.B. Füge $1, 2, 3, 4, \ldots, n$ in einen binären Suchbaum ein

02.12.2010

Wiederholung:

- Binärer Suchbaum
  - gewurzelt
  - geordnet
  - binär
  - mit der binären Suchbaumeigenschaft
  - alle Operationen in einem BSP brauchen Zeit O(Höhe des Baums)
  - Problem: BSP kann ausarten un Höhe $\Omega(n)$ haben $\rightarrow$ schlechte Laufzeit

10.3.1. *Aber:* Ein perfekter[21] hat Höhe $O(\log n)$[22].

10.3.2. *Idee:* modifiziere `put` und `remove` so, dass der Baum jederzeut perfekt ist, ähnlich dem binären Heap.

[20] im linken Teil darf nix sein was größer ist als was im rechten Teil ist.
[21] alle Ebenen bis auf die letzte sind voll besetzt
[22] wie beim binären Heap

10.3.3. *Nur:* Das ist schwer, denn um die BSB-Eigenschaft zu garantieren, muss der Baum ggf. stakr umstrukturiert werden.

Wir können nicht effizient sicherstellen, dass der Baum jederzeit perfekt ist[23]. Daher müssen wir die Anforderungen lockern, so dass der Baum immer Höhe $O(\log n)$ hat, aber gleichzeitig genug strukturelle Flexibilität bietet, um `put/remove` effizient zu unsterstützen. Dazu gibt es viele Möglichkeiten:

- AVL Bäume
- 2-3 Bäume
- a-b Bäume
- rot-schwarz Bäume
- $BB[\alpha]$ Bäume
- rank balanced Bäume
- Geschwisterbäume
- Treaps
- ...

11. AVL Bäume

11.0.4. *Geschichte.* Benannt nach Adelson-Velskii und Landis. Entdeckt: 1962

11.1. **Was ist ein AVL Baum?** Ein AVL-Baum T ist ein höhen-balancierter BSB. Das heißt, für jeden inneren Knoten [24] v in T gilt: die Höhen der beiden Unterbäume unterscheiden sich höchstens um eins[25].

Hier kommt ein kleines illustriertes Beispiel.

AVL-Bäume sind fast so gut, wie perfekte Bäume.

11.1.1. *Satz:* Die Höhe eines AVL-Baumes mit n Knoten ist $O(\log n)$
Beweis: Definiere n_h als die minimale Anzahl Knoten in einem AVL-Baum der Höhe h

$n_0 = 1$

$n_1 = 2$

$n_h = 1 + n_{h-1} + n_{h-2}$[26]

Behauptung: $n_h = 2^{\lceil \frac{h}{2} \rceil}$
Beweis: durch Induktion

IA: $n_0 = 1 = 2^{\lceil \frac{0}{2} \rceil} = 2^0$, $n_1 = 2 = 2^{\lceil \frac{1}{2} \rceil} = 2^1$

[23] das wäre zu teuer/würde zu lange dauern
[24] Knoten, der kein Blatt ist
[25] Die Höhe des leeren Baumes = -1
[26] Wurzel+linker Knoten+rechter Knoten

IS:

$$\begin{aligned} n_h = 1 + n_{h-1} + hh - 2 &\geq 2^{\lceil \frac{h-1}{2} \rceil} + 2^{\lceil \frac{h-2}{2} \rceil} \\ &\geq 2^{\lceil \frac{h}{2} \rceil - 1} + 2^{\lceil \frac{h}{2} \rceil - 1} \\ &\geq 2 \cdot 2^{\lceil \frac{h}{2} \rceil - 1} = 2^{\lceil \frac{h}{2} \rceil} \end{aligned}$$

Also: Wenn der AVL-Baum Höhe h hat, dann $n \geq 2^{\lceil \frac{h}{2} \rceil} \Leftrightarrow \log n \geq \lceil \frac{h}{2} \rceil \Rightarrow h \leq \log n$ $\square$
Tatsächlich gilt sogar $h \leq (\log_\phi 2) \log n$

11.2. **Operationen.**

11.2.1. *Konsistenz.* Wie stellen wir sicher, dass der BSB ein AVL-Baum bleibt, nach jedem `put/remove`?

- speichere die Höhe des zugehörigen Teilbaums in jedem Knoten[27]
- wenn ein Knoten unausgeglichen wird [28], müssen wir handeln!
- die l-Rotation ist eine lokale Operation um die Struktur des Baumes zu ändern. Sie lässt sich in $O(1)$ Zeit durchführen, durch Änderung konstant vieler Zeiger.[29]. Sier erhält die BSB-Eigenschaft
- symmetrisch dazu die r-Rotation
- nach einer l/r-Rotation erlangen wir in den meisten Fällen einen ausgeglichenen (Teil-)Baum (außer wenn (bei l-Rotation) der linkte Teilbaum größer als der rechte Teilbaum von v ist).
- im problematischen Fall (linker Teilbaum ¿ rechter Teilbaum) wird zuerst eine r-Rotation auf der Kante v-(linker Knoten) und danach die l-Rotation auf der neuen u-v Kante ausgeführt. Das nennt man Doppelrotation.

11.2.2. *Fazit.* Wenn u ein niedrigster unausgeglichener Knoten ist, so gibt es immer eine Rotation (l,r,lr,rl), die u ausgleicht.

07.12.2010

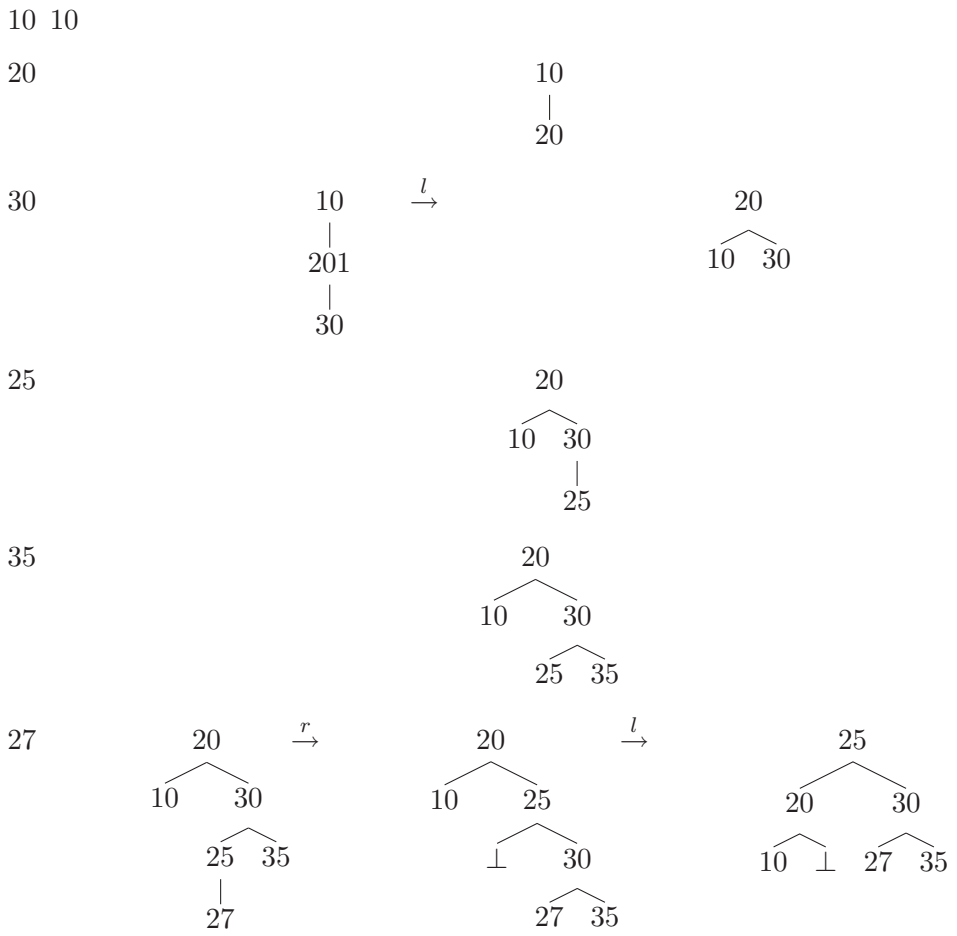
11.2.3. *Einfügen und Löschen.*

(1) Verfahre wie bei normalen BSB

(2) Fix-Up (Reparaturphase): verfolge rückwärts den Pfad zur Wurzel und führe die nötigen Rotationen durch, um die Knoten auszugleichen
$\Longrightarrow O(\log n)$ Laufzeit

[27] lässt sich bei `put/remove` leicht aktualisieren
[28] d.h. $|h_l - h_r| = 2$
[29] da nur 3 Zeiger geändert werden

11.3. **Beispiel.** 10,20,30,25,35,27
Einfügen der Zahlen:



11.4. **Vor- und Nachteile.**

- Vorteile:
  - liefern $O(\log n)$-worst-case Laufzeit
  - bieten eine kompakte Darstellung der Daten
- Nachteile:
  - Implementierung ist aufwändig
  - müssen recht oft rotieren

12. Rot-Schwarz-Bäume

Muss nicht gelernt werden, gilt als allgemeine Information.

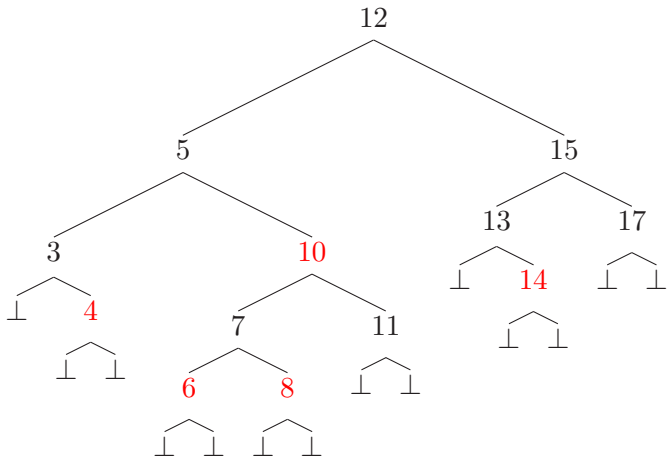
12.1. **Definition.** Rot-Schwarz-Bäume sind eine Veralgemeinerung von AVL-Bäumen. Sie lockern die Struktur (im vergleich zu AVL-Bäumen) noch weiter, um die Anzahl der Rotationen zu verringern. Jeder Knoten hat eine Farbe (rot/schwarz), so dass:

- die Wurzel ist schwarz

- die Hammerknoten[30] sind schwarz
- die Kinder von roten Knoten sind schwarz
- alle Pfade von der Wurzel zu einem Hammerknoten enthalten gleich viele schwarze Knoten

Rot-Schwarz-Bäume haben (wie AVL-Bäume) Höhe $O(\log n)$. Jeder AVL-Baum ist gleichzeitig auch ein AVL-Baum[31]. Beim Einfügen und Löschen hat man nun die Möglichkeit zu rotieren, und rot-färben. Bei RS-Bäumen muss man jeweils höchstens zweimal rotieren.

12.2. **Beispiel.**



13. (a,b)-Bäume

(a,b)-Bäume sind Mehrweg-Suchbäume, d.h. der Grad der Knoten kann variieren.

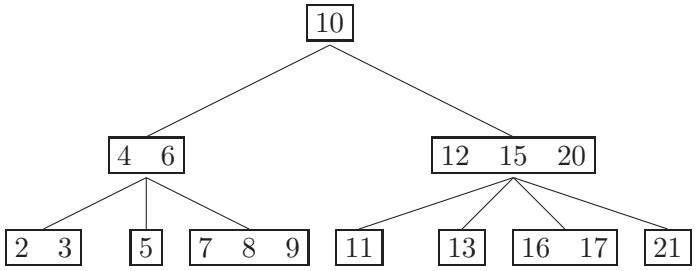
13.1. **Definition.**

- $a, b \in \mathbb{N}$, so dass $b \geq 2a - 1, a \geq 2$
- der Grad eines jeden Knoten ist $\leq b$
- der Grad der Wurzel ist ≥ 2
- der Grad jedes inneren knoten ist $\geq a$
- jeder Knoten speichert zwischen $a - 1$ und $b - 1$ viele Einträge (die Wurzel speichert mind. 1 Eintrag)
- die Tiefe[32] für jedes Blatt ist gleich

An diese Stelle kommt eine Illustration von Ludwig

[30] nichtexistente Blätter, an einem Knoten; nil/null
[31] für jeden AVL-Baum kann ich eine Färbung finden, so dass er ein RS-Baum ist
[32] Anzahl Kanten bis zur Wurzel

13.2. **Beispiel.**



13.3. **Operationen.**

13.3.1. *Suche.* wie in BSBs, aber müssen ggf mehrere Schlüssel per Knoten vergleichen. worst-case-Laufzeit: $O(b \log n)$.

13.3.2. *Einfügen.* am gegebenen Beispiel

An dieser Stelle sollte das ganze auch noch ein wenig besser illustriert werden.

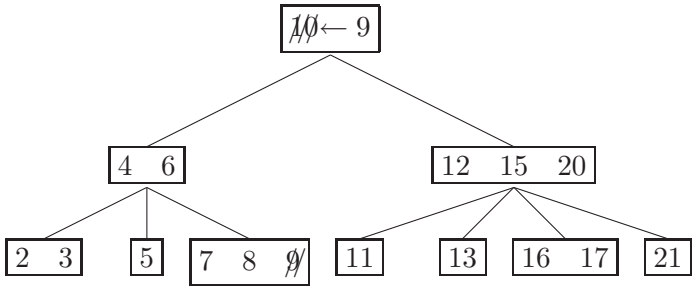
- eingefügt wird immer nur im Blatt
- suche das korrekte Blatt und füge ein
- wenn das Blatt schon "voll" ist, dann spalte den Knoten in drei Teile (wobei der zweite Teil ein Element und der erste und dritte möglichst gleich groß ist) und gib das mittlerste Element an den Elternknoten weiter (damit die anderen beiden Teile des Knoten dort als Kinder hinzugefügt werden können.
- verfahre mit dem Elternknoten ggf (wenn er voll ist) dementsprechend

09.12.2010

13.3.3. *Löschen.* K

- suche K
- wenn K in einem inneren Knoten ist:
  - suche den Vorgänger[33]
  - ersetze den Eintrag von K durch den Eintrag für den Vorgänger
  - Lösche den Vorgänger aus dem Blatt

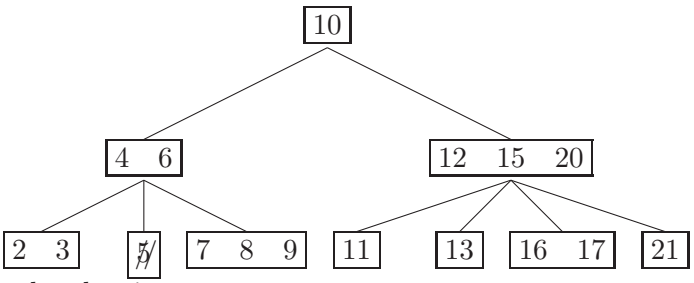
  $\Longrightarrow$ somit können wir uns auf den Fall beschränken, dass K in einem Blatt liegt
  Beispiel:[34] Lösche Element, dass nicht in einem Blatt ist
  Lösche 10



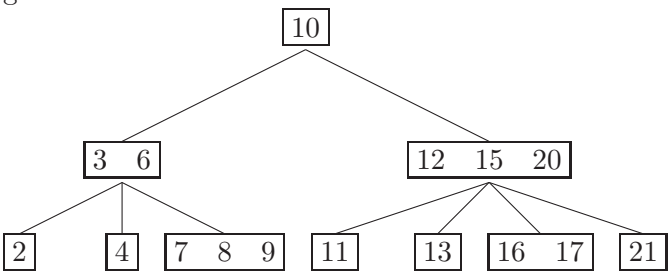
- wenn K in einem Blatt ist:

[33] oder Nachfolger

- ist einfach, wenn das Blatt $\geq a$ Schlüssel enthält
- wenn das Blatt $= a - 1$ Schlüssel enthält (Unterlauf)
  - versuche einen Schlüssel von den Geschwistern zu borgen
    Beispiel: borg dir den Schlüssel von deinen Geschwistern
    Lösche 5

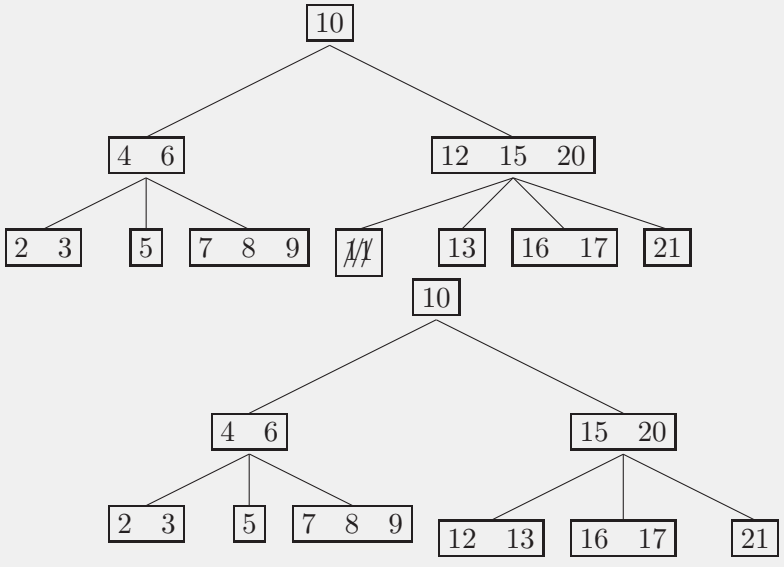


geht über in:



  - wenn das nicht geht
    - verschmelze mit einem Geschwisterknoten und dem entsprechenden Elternschlüssel

Beispiel: Familienglück
Lösche 11



    - ggf wiederhole mit dem Elternknoten

13.3.4. *Anwendungen.*

- $(2,3)-, (2,4)-$Bäume: als Alternative zu RS- bzw. AVL-Bäumen

13.3.5. *Externe Datenstrukturen.*

- ein riesiger Baum auf der Festplatte
- teure Operation: lesen/schreiben auf die Platte
  $\Longrightarrow$ der Baum soll möglichst geringe Höhe haben

- speichere so viele Einträge pro Knoten, wie in einen Festplattensektor passen
  $\Longrightarrow$ Die Leseoperationen werden optimal ausgenutzt und der Baum ist sehr Flach
- findet Verwendung in DBMS[35], Dateisystem (BTRFS)

14. Strings/Zeichenketten

14.1. **Definition.**

- $\sum$ Alphabet: endliche Menge von Zeichen
- String: endliche Folge von Symbolen aus $\sum$

14.2. **Fragestellungen:**

- Wie ähnlich sind zwei Zeichenketten?
- Suchen: finde alle Vorkommen einer Zeichenkette in einer anderen
- effiziente Speicherung: Kompression
- effiziente Wörterbücher für Zeichenketten (Tries)

14.2.1. *Beispiel für effiziente Speicherung.*

$\sum = (a, b, c, d)$
$s = abaaacdaab$
Wie codieren wie s effizient als Bitfolge? Einfach: Wie weisen jedem Zeichen$\lceil \log |\sum| \rceil$ viele Bits durch nummerierung zu.
$a \to 00, b \to 01, c \to 10, d \to 11$
$s = abaaacdaab \to 0001000000011011000001$
Somit brauchen wir $O(|s| \log |\sum|)$ Bits. Funktioniert, aber kann verschwenderisch sein (häufige Buchstaben sollen kürzere Codes bekommen)

14.3. **Code.**

14.3.1. *Definition.* Funktion c $\sum \to \{0,1\}*$. Eine Codierung eines Strings $s = G_1 G_2 \dots G_l$ ist somit $C(s) = C(G_1)C(G_2)\dots C(G_l)$
Ein Code muss die Eigenschaft haben, dass C eindeutig dekodierbar sein und einen möglichst kurzen Bitstring liefern.

[35] PostgreSQL, MariaDB

14.3.2. *Präfixfreiheit.* Wenn ein Code so beschaffen ist, dass kein Codewort Präfix eines anderen Codeworts ist, so heißt der Code präfixfrei. Präfixfreie Codes lassen sich eindeutig dekodieren und als Bäume darstellen.

14.12.2010

Beispiel: Präfixfrei
$\Sigma = \{a, b, c, d\}$

$$a \to 01$$
$$b \to 000$$
$$c \to 10$$
$$d \to 111$$

14.3.3. *Problem:* Gegeben ein Alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$ und Häufigkeiten $h_1, h_2, \ldots, h_k$ (=# Vorkommen von σ, im Text). Finde einen optimalen Präfixcode. d.h. finde einen Code $C : \Sigma \to \{0, 1\}*$, der die Gesamtlänge[36] minimiert.

14.3.4. *Idee:* Gehe gierig vor, d.h. optimiere lokal, hoffe auf ein globales Optimum.

Baue den Baum (=Präfixcode) von unten aus, wähle immer die "Symbole" mit kleinster Häufigkeit als Nächstes.

1 Lege ein Blatt für jedes Symbol an, Beschrifte jedes Blatt mit der entsprechenden Häufigkeit

2 Wähle zwei Teilbäume mit kleinstmöglichen Häufigkeiten, Vereine diese zu einem neuem Teilbaum, mit neuer Wurzel, addiere die Häufigkeiten.

3 Wiederhole, bis nur ein Baum übrig ist. Das ist der Code.

Satz: Huffmann-Codes sind optimale Präfixcodes, d.h. sie minimieren

$$\sum_{\sigma \in \Sigma} |C(\sigma)| h_\sigma$$

Lemma: 1 Sei $\sigma_i \in \Sigma$, so dass $h_{\sigma_i} \leq h_\sigma$ für alle $\sigma \in \Sigma$ Sei $\sigma_i \in \Sigma$, so dass $h_{\sigma_j} \leq h_\sigma$ für alle $\sigma \in \Sigma \backslash \{\sigma\}$

Dann xistiert optimaler Präfixcode, so dass in zugehörigen Baum σ_i und σ_j als Geschwisterhängen

Beweis: Sei $C*$ ein optimaler Präfixcode

Schritt 1 Es existiert ein optimaler Präfixcode C so dass das Blatt für σ_j mixmale Tiefe hat.

i Entweder das gilt schon in $C*$

ii Tausche das Blatt für σ_i mit einem Blatt von maximaler Tiefe. Dadurch wird die Codelänge für σ_i größer, die Länge für das alte Blatt kürzer, und die Gesamtlämge kann nicht größer werden, da σ_i minmale Häufigkeit hat.

Schritt 2 Existiert optimaler Präfixcode $C * **$, so dass σ_i und σ_j Geschwister sind.

[36] $\sum_{\sigma \in \Sigma} |C(\sigma)| h_\sigma$

- Gilt schon für $C**$
- sonst: σ_i hat einen Geschwisterknoten, da $C**$ optimal ist. Tausche diesen Knoten mit σ_j aus. Dadurch wird die Gesamtlänge nicht größer.

Beweis: des Satzes Induktion nach $K = |\Sigma|$

Anfang: $K = 2$

Schritt: Annahme: HK ist optimal für alle Alphabete und Häufigkeiten der Größe $K - 1$

z.Z.: ... für alle Alphabet und Häufigkeit der Größe K

Indirekt: Nimm an, existiert $\Sigma = \{\sigma_1, \ldots, \sigma_K\}$ und Häufigkeiten $h_{\sigma_1}, \ldots, h_{\sigma_K}$, so dass der Huffmanncode nicht optimal ist. Seien σ_i und σ_j die beiden Symbole, die der HuffmannAlgorithmus zuerst vereint. Nach Lemma 1 gibt es optimalen Code, wo σ_i und σ_j Geschwister sind.

Mitschrift von dem pad: http://pad.spline.de/alp32010

16.12.2011

Satz Der Huffman-Algorithmus liefert einen optimalen Präfixcode für gegebene Häufigkeiten.

Lemma Seien $\sigma_i, \sigma_j \in \Sigma$ mit minimalen Häufigkeiten. Dann existiert ein optimaler Päfixcode in dem die Blätter für σ_i und σ_j Geschwister sind. (bild 16.12.1)

Beweis des Satzes
Induktion nach $|\Sigma| = k$
wenn $k = 2$ ✓
Induktinsschritt:

- Induktionsannahme: HK ist optimal für alle Häufigkeiten und alle Σ mit $|\Sigma| = k - 1$
- zu zeigen: HK ist optimal für alle Häufigkeiten und alle Σ mit $|\Sigma| = k$

Beweis des Schritts: Indirekter Beweis
Nimm an, es existiert ein Alphabet Σ_i mit $|\Sigma| = k$, und Häufigkeiten $h_{\sigma_1}, ..., h_{\sigma_k}$, so dass HK nicht optimal ist. Seien σ_i und σ_j, sodass σ_i und σ_j von Huffman zuerst vereinigt wieden. Nach Lemma ex ein optimaler Präfix-Kode O, in dam σ_i und σ_j Geschwister sind.
(bild 16.12.2)

$$(*) \sum_{\sigma \in \Sigma} |T(\sigma)| h_\sigma > \sum_{\sigma \in \Sigma} |O(\sigma)| h_\sigma$$

Sei $\Sigma' := \Sigma \setminus \{\sigma_i, \sigma_j\} \cup \{\tau\} \quad |\Sigma'| = k - 1$
und $h_\tau := h_{\sigma_i} + h_{\sigma_j}$
Definiere zwei Bäume: (bild 16.12.3) T' ist der Huffman-Baum für $\Sigma' \Rightarrow T'$ ist optimal, nach IA

Aber:
(*) impliziert $\sum_{\sigma\in\Sigma'} |T'(\sigma)|h_\sigma > \sum_{\sigma\in\Sigma'} |O'(\sigma)|h_\sigma$
denn:

$$\sum_{\sigma\in\Sigma'} |T(\sigma)|h_\sigma = \sum_{\sigma\in\Sigma'} |T'(\sigma)|h_\sigma + h_{\sigma_i} + h_{\sigma_j}$$
$$\sum_{\sigma\in\Sigma'} |O(\sigma)|h_\sigma = \sum_{\sigma\in\Sigma'} |O'(\sigma)|h_\sigma + h_{\sigma_i} + h_{\sigma_j}$$

Würde heißen: O' ist besser als der Huffman-Kode.
Widerspricht der IA □

14.4. **Datenhompression.** – Huffman-Kodes sind optimal, aber nur unter bestimmten Annahmen.

1. Präfixcode
2. verlustfrei
3. auf Zeichenebene

- Kodes, die die Annahme 4 brechen:
  - Lauflängenkodierung

    $aaaaabbbaaaaabbbccccccc$
    $5a\ 3b\ 5a\ 3b\ 7c$

    Fasse sich wiederholende Zeichen in Gruppen zusammen. (z.B.: .PCX, .RLE)

    - finde wiederkehrende Muster und kodiere diese
      - Lempel-Ziv-Welch (LZW)
        compress, PKZIP, GIF (PNG)

- Verlustbehaftet(Annahme 2)
  - Bei Bildern/Musik sind nicht alle Informationen „wichtig“: versuche nur „relevante“ Daten zu speichern (JPEG, MP3, MP4, FLV)
    - Fourier-Transformation
    - Diskreter Kosinus-Transformation
    - Wavelet-Transformation

14.5. **Ähnlichkeiten von 2 Zeichenketten.** Neues Problem
Wie ähnlich sind zwei Zeichenketten?

z.B.: diff(linux), fc(win) (Programme die 2 Datein vergleichen)
Gegeben zwei Dateien, zeige die Unterschiede(minimum # Einfügen und Löschungen)

Formalisierung: 2 Zeichenketten
$s = \ s_1 \ s_2 \ ... \ s_k$
$t = \ t_1 \ t_2 \ ... \ t_k$

s = xyz - Weihnachten
t = WEIHNACHTEN

Formalisierung von Ähnlichkeite

- definiere elemenare Operationen (Einfügen, Löschen, Vertauschen). Was ist die min # Ops um s nach t zu überführen?(Editierabstand/Levenshtein-Metrik)
- Längste gemeinsame Teilfolge
  Gemeinsame Teilfolge: streiche Zeichen, bis bei beiden das gleiche steht
  (Bild 16.12.4)

04.01.2011

14.5.1. *Operationen.*

- Zeichen einfügen
- Zeichen löschen
- Zeichen vertauschen

Bestimme minimale Anzahl Operationen, um von s nach t zu kommen. (ii) Finde die längste gemeinsame Teilfolge [Bild]

z.B. NEUJAHRSTAG EPIHANIAS Längste gemeinsame Teilfolge: EHA oder EAA oder auch NAS

Wie berechnet man eine LGT? Zunächst bestimme die Längste LGT. * Rekursion von hinten * Schaue die letzten Buchstaben der beiden Wörter an: S_k und t_l

Wenn $S_k! = t_l$, bestimme die LGT von $S_1 \dots S_n - 1$ und $t_1 \dots t_l$ und $S_1 \dots S_k, t_1 \dots t_l - 1$ und nimm die längere von beiden (bzw, das Max der Längen).

Wenn $s_k = t_e$ bestimme rekursive die LGT von $s_1 \dots s_{k-1} und t_1 \dots t_{l-1}$ und gib diese $LGT + t_l$ zurück (bzw 1+Länge)

Wenn S oder t leer ist, dass gib die Folge, bzw 0 zurück.

[Bild]

Direkte implementierung der Rekursion ist ineffizient (exp. Laufzeit)

Lösung: Dynamisches Programmieren, speichere Werte in einer Tabelle. Sei LLGT ein (k+1)*(l+1)Array, das Zwischenergebnisse speichert. Zwei Möglichkeiten: - Implementiere die Rekursion, aber schaue in der Tabelle nach, falls nötig - Fülle die Tabelle von unten auf, mit for-Schleifen

```
1 for i:= 0 to k do
2         LLGT[i,0]<-0
3 for j:=0 to l do
4         LLGT[0,j]<-0
```

```
1 for i:=1 to k
2         for j:= 1 to l
3                 if s_i = t_j then
4                         LLGT[i,j]<-1+LLGT[i-1,j-11]
```

```
5                    else
6                        LLGT[i,j]<-max(LLGT[i-1,j],LLGT[i,j-1])
```

Laufzeit O(k*l)

Bsp.: s=Hund t=Funke LLGT

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	1	1	1
3	0	0	1	2	2
4	0	0	1	2	2
5	0	0	1	2	2

Die Antwort ist LLGT[k,l]. Wie findet man die LGT? -Folge den Pfeilen von LLGT[k,l]. Gib das Zeichen aus, jedesmal wenn ein /*roter Verweispfeil zwischen den Einträgen*/ überquert wird. Drehe das Ergebnis um. Zur Implementierung kann man ein zweites Array für die Pfeile anlegen, oder die Pfeile direkt aus LLGT rekonstruieren.

Stringsuche Gegeben $s = s_1 s_2 \dots s_k$ $t = t_1 t_2 \dots t_l$ $l \leq k$ Frage: Kommt t in s vor? Wenn ja, wo?

Listing 1. Naiver Algorithmus

```
1 for i:= 1 to k-l+1
2         |j<-1
3         |while (j=<l und s_{i+j-1}=t_j)
4         |     |j++;
5         |if (j=l+1)
6         |     |return i
7 return -1;
```

Laufzeit:O(kl)

6.01.2011

Wiederholung

- Das Problem der Suche in Zeichenketten
  $s = s_1 s_2 ... s_n$
  $t = t_1 ... t_l$ ¡- Muster

Frage: Ist t in s enthalten, und wenn ja, wo?

Listing 2. Naiver Algorithmus: O(kl)

```
1         for i:=1 to k-l+1
2                 |if s[i...i+l-1]=t then <- Kann man den Vergleich beschleunig
3                 |     |return i
4         return -1;
```

Idee: Können wir $s[...]$ und t Zahlen zuordnen, so dass, wenn $s[...] = t$ ist, die Zahlen gleich sind, und wenn $s[...] \neq t$ ist, die Zahlen wahrscheinlich verschieden sind?
Das geht z.B. durch Hashfunktionen.

```
for i:=1 to k-l+1
    |if h(s[i,...,i+l-1])=h(t) then
    |    |if s[i,...,i+l-1]]=f then
    |    |   |return i
return -1;
```

Wir vergleichen erst die Hashwerde von s[...] und t, und Vergleiche s[] mit t nur, wenn eine Kollision aufritt.
Der Vergleich der Hashwerte braucht nur O(1) Zeit, und Kollisionen sind selten, weswegen die Zeit für einen Vergleich zwischen s[...] und t im Fall $s[...] \neq t$ zu vernachlässigen ist.
=¿O(k+l)heuristische Laufzeit.
Problem: Wir müssen die Hashfunktion berechnen.
h(t) muss man nur einmal berechnen, fällt also nicht zu sehr ins Gewicht.
Aber: wir müssen
h(s[1...l]),h(s[2...l+1]),h(s[3...l+2])...
alle berechnen, in Zeit O(k)

Lösung: Wähle h, so dass sich h(s[i,...,i+l-1]) in O(1) Zeit berechnen lässt, wenn man h(s[i-1,...,i+l-2]) kennt.

$$h(s[i,...,i+l-1]) = (\sum_{j=0}^{l-1} |\Sigma|^{l-1-j} s_{i+j}) \mod p$$

$a_0 a_1 a_2 \dots a_k \in \Sigma^k$
$(\sum_{j=0}^{k} a_j * |\Sigma|^j) \mod p$

Bsp
$s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 \quad t_1 t_2 t_3 \ |\Sigma| = 4$
$h(s_1 s_2 s_3) = (s_1 * 4^2 + s_2 * 4^1 + s_3 * 4^0) \mod p$
$h(s_2 s_3 s_4) = (s_2 * 4^2 + s_3 * 4^1 + s_4 * 4^0) \mod p$
$h(s_3 s_4 s_5) = (s_3 * 4^2 + s_4 * 4^1 + s_5 * 4^0) \mod p$

$h(s_2 s_3 s_4) = (4 * h(s_1 s_2 s_3) - s_1 * 4^3 + s_4 * 4^0) \mod p$
$h(s_3 s_4 s_5) = (4 * h(s_2 s_3 s_4) - s_2 * 4^3 + s_5 * 4^0) \mod p$

Allgemein:
$h(s[i+1,....i+l]) = (|\Sigma| h(s[i,...,i+l-1]) - |\Sigma|^l s_i + s_{i+l}) \mod p$
D.h. man kann h(s[i+1,....i+l]) in O(1) aus h(s[i,....i+l-1]) berechnen.

Bem.

(1.) Man hofft auf Laufzeit O(k+l), wenn Kollisionen selten sind. Durch geeignete zufällige Wahl von p kann mann O(k+l) erwartete worst-case Laufzeit erreichen.

(2.) Interessante Idee:
Fingerprinting: Vergleiche Hashwerte statt der Strings selber
(z.B. Passwortverifikation, digitale Signatur, usw.)

15. Wörterbücher für Strings

Problem Wollen eine Menge von Zeichenketten speichern, sodass die Operationen des ADT geordnetes Wörterbuch unterstützt werden. Zur Vereinfachung ignorieren wir die Werte und betrachten nur die Schlüssel. 1.Lösung: Nimm eine bekannte Implementierung der ADT geordnetes Wörterbuch und speichere die Strings darin. Unschön: - Vergleiche von Strings benötigen nicht konstante Zeit z.B. Suchen von s in einem AVL-Baum kann O(—s—logn) Zeit benötigen. -Die spezielle Struktur der Daten wird ignoriert. 2.Lösung Verwende eine spezialisierte Datenstruktur für Zeichenketten. - bessere Laufzeit - mehr Struktur -¿ mehr Anwendungen Bsp S=HALLO,HAND,HUND,HAUS,TAG [Bild] Dies Datenstruktur heist Trie. Retrieval Ein ein Trie ist ein Mehrwegbaum. Jeder innere Knoten hat 1 bis —— Kinder. Die Kanten sind beschriftet mit Zeichen aus Σ, so dass jeder innere Knoten höchstens eine Kinidkate pro Zeichen in Σ hat. Die Blätter entsprechen den Wörtern, die in dem Trie gespeichert sind, und zwar erhält man das Wort zu einem Blatt v, indem man die Zeichen von der Wurzel zu v aneinanderreiht.

Problem Was ist, wenn ein Wort Präfix eines anderen Wortes ist? z.B. füge HALL zu S hinzu? 2 Reperaturmöglichkeiten (1) Erweitere die DS um Markierungen, die das Wortende anzeigen (2) €

11.01.2011

Wiederholung

- Stringsuche: Rabin-Karp Algorithmus Beschleunigt den naiven Algorithmus durch eine Hashfunktion
- Durch die spezielle Struktur der Hashfunktion können wir die Hashes für alle Teilstrings s in $O(|s|)$ Zeit ausrechnen
- Fingerprinting: repräsentiere eine Zeichenkette durch ein Hashwert

Tries: Implementierung des ADT geordnetes Wörterbuch für den Fall, dass die Schlüsselmenge aus Zeichenketten besteht.

- Gewurzelte Mehrwegbaum, in dem jeder ennere Knoten zwischen $1, ..., |\Sigma|$ Kinder hat.
- Jede Kindkante für einen inneren Hnoten ist mit einem Sümzol $\sigma \in \Sigma$ beschriftet, so dass jedes Zeichen höchstens einmal vorkommt. [kleines Bild]
- Die Blätter entsprechen der Zeichenkette im Trie.

[Bild]*

Problem: Was, wenn ein Wort Präfix eines anderen Wortes ist.
z.B. Füge HANDSCHUH in den obigen Trie ein.

- Eine einfache Lösung: Sei €$\notin \Sigma$ ein neues Zeichen. Füge €an alle Wörter hinten an.

  [Bild]

Speicherbedarf $K = K_1, K_2, ...K_l \qquad K_i \in \Sigma^+$

Der benötigte Platz ist $O(\sum_{i=1}^{l} |K_i|)$

Die Operationen put, get, pred, succ, remove lassen sich in jeweiles $O(|\Sigma||K|)$ interpretieren, wobei K der Schlüssel ist, auf dem die Opera ausgeführt wird. (bei pred, succ: $O(|\Sigma|(|K| + |K'|))$ bei pred,succ, wobei K' der Vorgänger bzw. Nachfolger ist.)

Komprimierte Trie: PATRICIA-Tries

- Ziehe Pfade, die aus Knoten mit jeweils mit einem Kind bestehen, zu einer Kante zusammen, die mit dem ensprechenden Teilstring beschriftet ist.
  K = {HALS, HALSAND, HALSBANDFARBE }

  [Bild]

PATRICIA-TRIE Ursprung:
Primitive
Algorhytmus
To
Receive
Information
Coded
In
American Language

- Kompremierte Tries haben nur O(l)

(l = # Schlüssel) viele Knoten statt $O(\sum_{i=1}^{l} |K_i|)$ und sind dadurch platzsparender. Sie bringen aber i.A. keinen asymptotischen Vorteil.

Es gibt aber eine wichtige Ausnahme
Suffix-Bäume
Ein Suffix-Baum ist ein Komprimierter Trie, der alle Suffixe eines gegegbenen Strings speichert.

z.B.: MISSISSIPPI€
hat Suffixe MISSISSIPPI€,ISSISSIPPI€,SSISSIPPI€,...,€

[Bild]

Ein String der Länge n hat O(n) Suffixe, also hat der Suffixbaum O(n) viele Knoten, aber die Gesamtgröße der Labels kann $\Theta(n^2)$ sein.

Anwendung: Verarbeite einen langen Text vor, so dass man viele verschiedene Suchen nach Mustern/Teilstring effizient durchführen kann.
Ein Suffixbaum erlaubt es, nach einem Muster in O(—t—) Zeit zu suchen.

Problem: Größe des Suffixbaumes kann quadratisch sein.
Lösung: Verwende Zeiger
MISSISSIPI€
Statt mit Teilstrings beschrifte die Kanten mit Zeigern in dem String s. Zeiger: start

und Endindex des Substrings.
=¿ O(n) Speicher

Konstruktion:

(a) Einfügen aller Suffixe in ein Komprimten Trie $O(n^2)$ Zeit.

(b) In O(n) Zeit mit spezial Algorithmus.

16. Graphen

16.1. **Definition.** Ein Graph G=(V,E) besteht aus V:Knoten und E:Kanten
Die Kanten sind Teilmengen der Größe 2 der Knotenmengen.

[Bild]

16.2. **Varianten.**

- Multigraphen: [Bild] Mehrfachknoten& Schleifen
- gewichtete Graphen [Bild]
- gerichtete Graphen

[Bild]

Bsp

- Bäume (Stamm-)
- Facebook
- Internet
- Eisenbahnnetz
- Stromnetz
- Spielgrraphen

13.01.2011

16.3. **Probleme auf Graphen.**

(a) Gegeben zwei Knoten $u, v \in V$, gibt es einen Weg von u nach v?

(b) Was ist die Länge eines kürzesten Weges von u nach v?
Mit gewichteten Kanten?

(c) Gegeben eines gerichteten Graph, in dem die Kanten Abhängigkeiten darstellen. ordne die Knoten, so dass kein Knoten von einem Nachfolger abhängig ist.

(d) Gegeben ein gewichteter Graph, was ist eine billigste Möglichkeit, alle Knoten miteinander zu verbinden?

(e) Gegeben zwei Knoten $u, v \in V$, was ist die minimale Anzahl an Kanten, dieman entfernen muss, um u und v zu trennen.

16.3.1. *Darstellung von Graphen im Rechner.* Adjazenzmatrix & Adjazenzliste

Adjazenzmatrix
$|v| \times |v|$ Array vom Typ boolean
$a[i][j] = \begin{cases} \text{true}, & \text{wenn eine Kante von hnoten i zu Knoten j gibt} \\ \text{false}, & \text{sonst} \end{cases}$

[Bild]

Adjazenzliste:
speichere: $|v|$ verkettete Listen, eine für jeden Knoten. Die Liste für Knoten i speichert die nachbarn von i.

[Bild]

Der ADT Graph
(Anmerkung Vertices und Node ist das gleiche)

(1) vertices(): gibt die Menge der Knoten zurück(Iterator)
(2) edges(): gibt die Menge der Kanten zurück(Iterator)
(3) incident Edges(v): gib die Inzidenten Kanten eines Knoten zurück(Iterator)
(4) insertEdge(u,v)
(5) removeEdge(e)
(6) newNode()
(7) deleteNode(v)
(8) endVertices(e): Finde Endknoten der Kante e
(9) opposite(e,v): finde den anderen Endknoten von e
(10) get/set- NodeInfo(u)
(11) get/set - EdgeInfo(e)

Implementierung:

	A-Matrix	A-Liste
(2)	O(—v—)	O(—v— + —E—)
(3)	O(—v—)	O(—v— + —E—)
...		

IA ist die Adjazenzliste effizienter als die A-Matrix.

16.3.2. *Durchsuchen von Graphen.* Gegeben ein Startknoten s, zähle systematisch alle Knoten auf, die sich von s erreichen lassen.

16.3.3. *Tiefensuche (DFS - depfh first search).* – gehe immer so weit wie möglich. Wenn es nicht mehr weiter geht, gehe zurück.

```
1 dfs(v):
2     process(v)
3     v.visited <- true
```

```
4       for all w adjazent to v do
5           if not w visited
6               dfs(w)
```

(Demonstration des Algo an einem Bild)

Laufzeit
O(—v—+—E—) mit Adjazenzliste
$O(|v|^2)$

Anwendungen von DFS:

- Kann man einen Knoten v von s aus erreichen?
- Finde einen Kreis im Graph
- finde einen aufspannden Baum

18.01.2011

Listing 3. DFS rekursiv

```
1  dfs(v)
2          v.found<-true
3          for e in v.incidentEdges() do
4                  w<-e opposite(v)
5                  if !w.found then
6                          dfs(w)
```

Listing 4. DFS iterativ

```
1  S<-new Stack
2  s.found <- true
3  S.push((s,s.incidentEdges()))
4  while (!S.isEmpty())
5          (v,e)<-S.top()
6          do
7                  if (e.hasNext())
8                          w<-p.next().opposite(v)
9                  else
10                         w <- nil
11         while (w!= nil && w.find)
12
13         if w= nil then
14                 s.pop()
15         else
16                 w.find <- true
17                 s.push((w,w.incidentEdges))
```

Listing 5. BFS

```
1  Q<-new Queue
2  s.found <- true
```

```
3
4  Q.enqueue(s)
5  while(!Q.isEmpty())
6          v<–Q.dequeue
7          for e in v.incidentEdges do
8                  w<–e.opposite(v)
9                  if(!w.found) then
10                         w.found <– true
11                         Q.enqueue(v)
```

Wiederholung:

- Der ADT: Graph
- Implementierungen:
  - Adjazenzmatrix
  - Adjazenzlisten
- DFS und BFS

16.3.4. *BFS - Breitensuche.* Gehe immer nur einen Schritt vorran. Wenn nicht gefunden, gehe einen Schritt weiter ... usw.
Laufzeit: $O(|V| + |E|)$

16.4. **kürzeste Wege in ungewichteten Graphen.**

16.4.1. *SPSP.* Gegeben: $s, t \in V$ Was ist ein Weg in G von s nach t, der die minimale Anzahl von Kanten benutzt?[37]

16.4.2. *SSSP.* Verallgemeinung: Gegeben sei ein Startknoten s; finde kürzeste Wege zu allen Knoten in $V/\{s\}$[38]
Wir lösen das SSSP-Problem. Im allgemenen lässt sich das auch nicht vermeiden, denn kürzeste Wege haben die **Teilpfadoptimaltitätseigenschaft**.

16.4.3. *Teilpfadoptimalität.* Sei $p : s \to u_1 \to \dots u_k \to \dots t$ ein kürzester Weg von s nach t, dann ist $s \to \dots u_k$ ein kürzester Weg von s nach u_k.
Beweis: Gäbe es einen kürzeren Weg von s nach u_k, so wäre p nicht optimal.

16.4.4. *Darstellung.* der kürzesten Wege von s zu allen Knoten in $V\{s\}$ als kürzeste-Wege-Baum:
Jeder Knoten speichert einen Zeiger zu seinem Vorgänger (v.pred) auf einem kürzesten Weg von s.
Zusätzlich: Jeder Knoten v speichert einen Wert v.d, die Länge eines kürzesten Weges.

Wir können die BFS verändern, so dass dieser einen kürzeste-Weg-Baum erstellt:

[37] single pair shortest path problem
[38] single source shortest path, sssp

Listing 6. BFS - kürzeste Wege

```
1  Q<–new Queue
2  s.found <– true
3  s.d<–0
4  s.pred<–nil
5
6  Q.enqueue(s)
7  while(!Q.isEmpty())
8          v<–Q.dequeue
9          for e in v.incidentEdges do
10                 w<–e.opposite(v)
11                 if(!w.found) then
12                         w.found <– true
13                         w.d<–v.d+1
14                         w.pred<–v
15                         Q.enqueue(v)
```

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

- Jeder Knoten wird höchstens einmal in Q eingefügt $\rightarrow \leq 1$ Durchlauf der while-Schleife pro Knoten
- Jede Kante wird ≤ 2 mal traversiert $\rightarrow \leq 2$ Durchläufe for-Schleife pro Kante

16.5. **kürzeste Wege in gewichteten Graphen.**

16.5.1. *Vorstellung:* Die BFS entspricht einer "Welle", die sich gleichmäßig ausbreichtet. Die Knoten hinter der Welle sind vollständig bearbeitet. Die knoten vir der Welle sind unentdeckt, oder in Q. Die Knoten in Q sind diejenigen, die eine Kante besitzen, die von der Welle geschnitten wird.

16.5.2. *Die "Welle" in gewichteten Graphen.* Sei G ein gewichteter Graph, in dem alle Kanten Gewichte ≥ 0 haben. Wir wollen wieder einen Algorithmus finden, der sich wellenförmig ausbreitet.
Problem: Wie findet man den nächsten Knoten? Idee: Benutze eine Prioritätswarteschlange.
Problem: Der erste Weg, der zu einem Knoten gefunden wird, muss nicht der beste sein. Idee: Müssen jedes mal, wenn die Welle einen Knoten trifft, überprüfen, ob sich dadurch die Wege verkürzen.

Listing 7. BFS - kürzeste Wege in gewichteten Graphen

```
1  Q<–new PriorityQueue
2  s.found <– true
3  s.d<–0
4  s.pred<–nil
5
6  Q.insert(s,0)
7  while(!Q.isEmpty())
8          v<–Q.deleteMin()
9          for e in v.incidentEdges do
```

```
10                  w<-e.opposite(v)
11                  if(!w.found) then
12                          w.found <- true
13                          w.d<-v.d+e.length
14                          w.pred<-v
15                          Q.insert(v,w.d)
16                  else
17                          if w is in Q
18                                  if v.d+e.length < w.d then
19                                          w.d <- w.d+e.length
20                                          w.pred <- v
21                                          Q.decreaseKey(w,w.d)
```

20.01.2011

Wiederholung:

- kürzeste Wege
  - Teilpfadoptimalität
  - SSSP, SPSP, APSP[39]
  - kürzeste Wege Baum
  - ungewichtete Graphen: BFS löst SSSP-Problem in Zeit: $O(|V| + |E|)$
  - gewichtete Graphen: Dijkstras[40] Algorithmus

16.5.3. *Dijkstras Algorithmus.*

Listing 8. Dijkstras Algorithmus

```
1  Q<-new PriorityQueue
2
3  for v in vertices() do
4          v.d<-\infinity; v.pred<-nil;Q.insert(v,\infinity)
5  s.d<-nil;Q.decreaseKey(s,0)
6  while(!Q.isEmpty())
7          v<-Q.extractMin()
8          for e in v.incidentEdges() do
9                  w<-e.opposite(v)
10                 if v.d+e.lenght<w.d then
11                         w.d<-v.d-e.length
12                         w.pred<-v
13                         Q.decreaseKey(w,w.d)
```

Bemerkung:
$infinity = \infty$

Laufzeit:

[39] all pair shortest path
[40] Niederländischer Informatiker, Tod:2002

- $|V|$ mal Q.insert()
- einmal Q.decreaseKey()
- $|V|$ mal Q.extractMin()
- $|E|$ mal Q.decreaseKey()

Die Laufzeit hängt von der Implementierung von Q ab.
Q als binärer Heap:

- insert: $O(\log n)$
- extractMin: $O(\log n)$
- decreaseKey: $O(\log n)$ [41]

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

16.5.4. *Korrektheit des Dijkstra Algorithmus.*
Satz: Für alle Knoten $v \in V, v \notin Q$ ist v.d Länge eines kürzesten Weges und die Vorgängerzeiger geben einen kürzesten Weg von s nach v.
Beweis: Induktion nach Anzahl der Durchläufe der while-Schleife.

IA:
- v=s
- der kürzeste Weg nach s ist korrekt berechnet

IS:
- Sei $v \leftarrow Q.extractMin()$
- zu zeigen: v.d ist die Länge eines kürzesten Weg von s nach v; und v.pred ist ein Vorgänger
- Sei p der Weg von s nach v, den die Vorgängerzeiger geben.
  Wir wissen: $|p| = v.d$
- Sei p' ein anderer Weg von s nach v
- zu zeigen: $|p'| \geq |p|$
- wir wissen, der Weg p' muss $V/Q \cup \{v\}$ irgendwann verlassen
- Sei $e = ab$ die Kante, wo dies zum ersten Mal passiert.
- schreibe $p' = p'_1 - a - b - p'_2$
- wir wissen, dass $|p'| \geq |p'_1| + ab.length$
- nach Induktionsannahme gilt: $|p'_1| \geq a.d$
- b ist in Q;
- $a.d + ab.length \underset{\text{wegen Verarbeitung von a}}{\geq} b.d \underset{\text{wegen } v<-Q.extractMin()}{\geq} v.d$
- $|p'| \geq |p'_1| + ab.length \geq a.d + ab.length \geq b.d \geq v.d = |p|$ $\square$

Bemerkungen

- Der Beweis funktioniert nur für Kantengewichte ≥ 0

[41] Zettel 5, Aufgabe 2

- wenn man nur s-t-Pfad finden will, kann man aufhören, wenn t aus Q extrahiert wird
- bidirektional: suche gleichzeitig von s und von t aus
- in der Praxis hat man oft Zusatzinformationen (bspw. Routenplanung: Knoten sind Punkte in der Ebene)
  - wie kann man diese Zusatzinformationen einsetzen
  - gesucht: $s \to t$ Pfad in einem gewichteten Graphen
  - zusätzlich haben wir einen Schätzer(Heuristik) $h : V \to \mathbb{R}^+$, der den Abstand zu t schätzt
  - wir sagen h ist konsistent, wenn für alle $a, b \in E$ gilt
    - $h(a) \leq ab.length + h(b)$
  - ändere die Kantengewichte: $ab.length' \leftarrow ab.length - h(a) + h(b)$
  - benutze Dijkstra mit diesen Längen (wenn h konstistent ist, sind alle Längen nicht negativ)
  - dieser Algorithmus heißt (A*-Algorithmus)

25.01.2011

Wiederholung:

- gerichtete Graphen $G = (V, E)$ mit Kantenlängen
- Dijkstra Algorithmus (berechnet alle kürzesten Wege vom Startknoten aus)
- Beschleunigung (für bspw. Routenplanung) mit dem A*-Algorithmus

16.6. **gewichtete Graphen mit negativem Kantengewicht.**

16.6.1. *Was passiert mit dem Dijkstra Algorithmus?* Dijkstra liefert keine korrekten Ergebnisse. Lösung: Bellman-Ford-Algorithmus (mit negativn Kantenlängen, aber ohne negative Kreise)
.[BILD: zeigt ein Beispiel mit negativem Kantengewicht]

16.6.2. *Warum sind negative Kreise nicht erlaubt?* Ohne negative Kreise gilt: Es gibt immer einen kürzesten Weg, der Doppelpunktfrei ist.
.[BILD: zeigt diese Behauptung]
Wenn es einen negativen Kreis gibt, dann ist die Frage nach dem kürzesten Weg sinnlos. (der negative Kreis würde beliebig oft durchlaufen werden.

16.6.3. *Weshalb sind negative Kantengewichte sinnvoll?* Wenn die Kantengewichte Kosten darstellen, dann können negative Kosten (Einkommen) sinnvoll sein.
Beispiel: Währungsrechner
.[BILD: Währungsrechner mit Euro, Dollar, Pfund]

16.7. **APSP.** [42] Kürzeste Wege für alle Knotenpaare
Gegeben:

- Gerichteter Graph $G = (V, E)$ mit
- $V = \{1, x, \ldots, n\}$ und
- Kantengewichten $a_{i,j}$ für alle $ij \in E$
- negative Kantengewichte sind erlaubt, aber
- nur positive Zyklen

G gegeben durch eine Matrix $A = (a_{ij})_{1 \leq i,j \leq n}$

$$a_{ij} = \text{Kantengewicht für } i, j \in E; 0 \text{ für } i = j; \infty \text{ sonst}$$

Gesucht: kürzeste Wege in Form von Matrix

$$D = (d_{ij})$$

$d_{ij} =$ länge des kürzesten Weges von i nach j; ∞, wenn solch ein Weg nicht existiert
Lösung: Algorithmus von Floyd-Warshall. Benutzt wird Dynamische Programmierung.

$$D^{(k)} = (d_{ij}^{(k)})$$

$d_{ij}^{(k)} =$ Länge eines kürzesten Wegs von i nach j mit Zwischenknoten aus $\{1, \ldots, k\}$

$$D^{(n)} = D$$

16.7.1. *Floyd-Warshall.*
Initialisierung:

$$D^{(0)} = A$$

Schritt:

$$D^{(k)} \to D^{(k+1)}$$

$$d_{ij}^{(k+1)} = min(d_{ij}^{(k)}, d_{i,k+1}^{(k)} + d_{k+1,j}^{(k)})$$

??Final:??

$$D = D^{(n)}$$

Laufzeit: $O(n^3)$ Berechnung der kürzesten Wege uber Vorgängermatrix

$$\prod = (p_{ij})$$

$p_{ij} =$ Vorgänger von j auf kürzesten Weg $i \to j$

$$p_{ij}^{(0)} = \text{i falls } ij \in E; \text{null sonst}$$

Speicher: $O(n^3)$
Im Schritt $k \to k+1$ kann $k-1$ überschrieben werden. Somit kann der Speicher auf $O(n^2)$ verringert werden.

[42] all pair shortest path

17. MST - Minimale Spannbäume

(minimal aufgespannte Bäume, minimum spanning tree → MST)
Gegeben: $G(V, E)$ ungerichteter, zusammenhängender Graph mit Kantengewichten

$$|\ |: E \to \mathbb{R}$$

Gesucht: Aufspannender Baum $T = (V, E')$ $E' \subseteq E$ mit minimalem Gesamtgewicht:

$$|T| = \sum_{e \in E'} |e|$$

17.1. **Schnitt.** Ein Schnitt von $G(V, E)$ ist eine Zerlegung von V in zwei nichtleere Teilmengen $(S, V \notin S)$
Satz: Sei $T = (V, E')$ ein MST von $G(V, E)$
Angenommen $e \in E$ ist Kante mit minimalem Gewicht aus $E_S\{uv|u \in S, v \in V \notin S\}$ für Schnitt $(S, V \notin S)$,
Dann gibt es in E' eine Kante $e' \in E_S$, so dass $|e'| \leq |e|$

E_s ist die Menge aller Kanten, die aus S nach $V \notin S$ führt.

27.01.2011

Für jede Kante minimalem Gewichts aus E_s gibt es einen MST, der diese Kante enthält.
Beweis a)
.[Bild: Illustration von 'Gegeben']
$e = e_3 \in T$ sei minimal in $E_s \cap E'$
Betrachte $T \cap \{e_1\}$, e_1 minimal in $E_s \to$ Kreis C mit e_1 auf C
$e_1 = uv$; Weg $u \to v$ in T enthält weitere Kante aus E_s.
Die Kante e_k hat Eigenschaft $|e_1| \leq |e_k|$

$$T' = (T \cup \{e_1\}\ /\{e_k\} \text{ ist aufspannender Baum}$$
$$|T'| = |T| + |e_1| - |e_k| \Rightarrow$$
$$|T'| \leq |T| \text{ schon minimal} \Rightarrow$$
$$|T'| = |T| \Rightarrow |e_1| = |e_k|$$

17.2. **Generischer MST-Algorithmus.** - Grundidee, die hinter vielen Algorithmen zur Lösung dieses Problems steht.
Gegeben: $G = (V, E)$
Gesucht: MST $T = (V, E')$

$E' = \emptyset$

solange (V, E') nicht zusammenhängend,

wähle Schnitt $(S, V\ /S)$, so dass $E' \cap E_s = \emptyset$

wähle E_s minimale Kante: $E' = E' \cup \{e\}$

17.3. **Algorithmen von Prim und Kruskal.** sind Umsetzungen des generischen Falls:

17.3.1. *Prim.* Wähle einen Startknoten u und setze initial $S = \{u\}$. Lasse Komponente S durch die ausgewählten Kanten wachsen.

17.3.2. *Kruskal.* Initial $E' = \emptyset$. Jeder Knoten eine isolierte Zusammenhangskomponente. Betrachte jeweils alle Kanten zwischen verschiedenen Zusammenhangskomponenten und wähle unter diesen die leichteste aus.

17.4. **Konkrete Umsetzung.**

17.4.1. *Prim.* ist dem Dijkstra Algorithmus sehr ähnlich.
Wir verwenden eine Prioritätswarteschlange für V. Schlüssel(v)=leichteste Kante von Startkomponente S nach v.
Laufzeit realisiert mit einer Halde/Heap: $O((n+m)\log n)$ wobei $n = |V|$ und $m = |E|$
.[Bild: Beispiel des Prim Algorithmus]

17.4.2. *Kruskal.* wird implementiert mit dem ADT: UNION-FIND

17.5. **Union-Find.** ist eine Datenstruktur zur Verwaltung von **disjunkten Mengen** (also von Mengenpartitionen). Jede Menge hat einen **Repräsentanten**.

17.5.1. *Operationen.*

FIND-SET(u) gibt den Repräsentanten der Menge von u

UNION(u,v) bewirkt die Vereinigung der Mengen von u und v (nach dieser Aktion haben u und v den gleichen Repräsentanten)

MAKE-SET(v) erzeugt eine einelementige Menge

17.5.2. *MST mit Kruskal.* $G = (V, E)$

```
1 E'=leere Menge (Initialisierung)
2 for all v in V
3         MAKE-SET(v)
4 sort E (aufsteigend nach Gewicht)
5 for all e=uv in E
6         if FIND-SET(u) != FIND-SET(v)
7                 E'=E' vereinigt e
8                 UNION(u,v)
```

Beispiel:
$E_{sortiert} = \underset{1}{ab}, \underset{1}{fg}, \underset{2}{bd}, \underset{2}{de}, \underset{3}{bc}, \underset{4}{ac}, \underset{5}{cd}, \underset{6}{cf}, \underset{7}{df}, \underset{8}{ef}$
.[Bild: Beispiel des Algorithmus von Kruskal]
Laufzeit: $U(\underset{MAKE-SET}{n} + m \underset{Sort}{\log m} + \underset{FIND-SET}{m \cdot 2} + \underset{AnzahlderUnion-Operationen}{(n-1)} \cdot \underset{naiveKostenschaetzung}{n})$
$=> O(m \log n + n^2)$

17.5.3. *gewichtete UNION-Heuristik.* Bei `UNION(u,v)` wird der Repräsentant der größeren Menge gewählt.
Für jede Menge wird benötigt:

V Die Menge aller Knoten

- einen Zeiger auf den Vorgänger (der Repräsentant)
- einen Zeiger auf den Nachfolger (für die sortierte Liste)
- Repräsentant hat zusätzlich noch eine Size

Bei gewichteter `UNION`-Heuristik verändert jeder Knoten seinen Repräsentanten höchstens $\lceil \log n \rceil$ mal. (Beweis über Induktion)
Laufzeit: $O(m \log n + n \log n)$. Da $m \geq n + 1 \rightarrow O(m \log n)$. Das ist nicht zu verbessern.

01.02.2011

Wiederholung:

- MST-Algorithmen
- Allgemeine Strategie / generischer Algorithmus: inkrementell; baue den MST Kante für Kante; beginne mit einem leeren Baum; füge immer eine sichere Kante hinzu
- Lemma: eine Kante e ist sicher für eine Kantenmenge A, wenn es einen Schnitt gibt, der A respektiert, so dass e eine leichteste Kante ist, die den Schnitt überquert
- Umsetzungen
  - Prim-Jarnik-Dijkstra: Lasse den baum von einem Startknoten aus wachsen, nimm immer die leichteste Kante, die die aktuelle Komponente verlässt
    Laufzeit: $O(|v| \log |v| + |E| \log |E|)$
  - Kruskal: sortiere die Kanten anch Gewicht; füge immer die nächste Kante ein, die zwei verschiedene Zusammenhangskomponenten verbindet
    Problem: wie verwaltet man die Zusammenhangskomponenten? Dafür haben wir den **ADT Union-Find**[43] verwendet.

17.5.4. *Naive Implementierung.* als Wald. Ein Baum für jede Menge in P, jede Menge wird durch die Wurzel des jeweiligen Baumes repräsentiert

- MAKESET(u): lege einen Knoten für u an
  Laufzeit: $O(1)$
- FINDSET(u): u liegt in einem Baum, welcher der Menge, die u enthält, entspricht; liefere die Wurzel des Baumes
  Laufzeit: O(Tiefe von u im Baum)
- UNION(u_1, u_2); wir kennen die Wurzeln der Bäume für u_1 und u_2; hänge u_1 unten an u_2
  Laufzeit: $O(1)$

Problem: `FINDSET` kann seht lange dauern; die Bäume können ausarten

[43] Disjoint-Set-Union

17.5.5. *Verbesserter Algorithmus.*
Lösung 1: `UNION BY SIZE`: bei `UNION`(u_1, u_2) hänge den kleinen Baum unter den größeren; Das sorgt dafür, dass jeder Baum Höhe $O(\log n)$ hat; d.h. `FIND` braucht $O(\log n)$ Zeit
Lösung 2: Pfadkompression: bei `FIND`, hänge alle Knoten auf dem Suchpfad hinterher unter die Wurzel
Satz: **UNION-FIND** mit Pfadkompression und **UNION-BY-SIZE** hat $O(\alpha(n))$[44] amortisierte Laufzeit pro Operation

17.6. **MST - Borůvka.** - der ältest MST-Algorithmus
Annahme: alle Kanten haben unterschiedliche Gewischte
Idee: lasse alle Zusammenhangskomponenten gleichzeitig wachsen

Listing 9. Borůvka - Algorithmus

```
1 A <- \emptyset
2 while |A| < n-1 do
3         B = \emptyset
4         for each connected component c of (V,A) do
5                 find the lightest edge e with exactly
6                 - one endpoint in ci
7                 B = B \cup {e}
8         A = A \cup B
```

Korrektheut kann durch wiederholte Anwendung des "sichere Kante" Lemmas gezeigt werden
Die Laufzeit beträgt $O(|E| \log |V|)$

03.02.2011

Wiederholung:

- Disjoint-Set-Union mit UnionBySize und Pfadkompression
- MST-Algorithmus von Borůvka
  Eingabe:
  - zusammenhängender gewichteter Graph mit paarweise verschiedenen Kantengewichten

Laufzeitanalyse des Algorithmus von Borůvka

- Behauptung: Borůvka braucht $O(|E| \log |V|)$ Zeit
  (1) Ein Durchlauf der while-Schleife benötigt $O(|E|)$ Zeit, weil wir müssen zwei Dinge tun:
    (a) bestimme die Zusammenhangskomponenten von (V,A); mit DFS oder BFS: $O(|V| + |A|)$ Zeit $= O(|E|)$
    (b) finde für jede Zusammenhangskomponente die leichteste inzidente Kante
      - gehe alle Kanten durch

[44] inverse Ackerman-Funktion

- betrachte die Zusammenhangskomponenten der Endpunkte (kennen wir aus (a))
- vergleiche das Kantengewichtmit dem aktuellen Minimum der Zusammenhangskomponenten und aktualisiere ggf.
- benötigt $O(|E|)$ Zeit

(2) Es gibt $O(\log |V|)$ Durchläufe der while-Schleife

- nach i Durchläufen der while-Schleife hat jede Zusammenhangskomponente $\geq 2^i$ Knoten
- Beweis durch Induktion $\square$

17.7. **Bessere MST Algorithmen.** Gibt es MST-Algorithmen mit Laufzeit $O(|E| \log |V|)$? Ja, es gibt viele. Der schnellste MST-Algorithmus braucht: $O(|E|\alpha(|e|))$ (Chazelle) bzw $O(|E|)$ Zeit durch Schummeln (Bit-Tricks oder Zufälligkeit)

18. Graphen und Spiele

Graphenalgorithmen eignen sich, um Puzzles zu lösen und Spiele zu gewinnen.

18.1. **Ein-Personen-Spiele.**

- MS-Solitaire
- Steck-Solitaire
- Sudoku

18.2. **Zwei-Personen-Spiele.**

- Schach
- Goo
- Schiffe-Versenken
- Halma
- TicTacToe
- Backgammon

18.3. **Was haben Spiele mit Graphen zu tun?** Wie kann man Spiele als Graphen darstellen?
Die **Knoten** sind die Stellungen und die **Kanten** sind die Züge. Wir Suchen einen (kürzesten) Weg von einer gegebenen (Start-) Stellung zu einer Siegstellung.
→ Graphensuche (DFS, BFS, A*-Suche)

Ein Problem dabei ist aber, dass der **Graph nicht explizit gegeben** ist und daraus die Frage ensteht, wo wir das Found-Attribut speichern.
Es gibt folgende Möglichkeiten

- verwende eine Hashtabelle, die alle bereits gesehenen Knoten enthält

- found wird gar nicht gespeichert; Knoten werden eventuell doppelt betrachtet; klappt mit DFS; spart Speicher; Backtracking

18.4. **Wie lösen wir zwei Spieler Spiele?** Darstellung als Baum. [Bild: Darstellung als Baum]