1. Introduction

Algorithmus

Eine aus endlich vielen Schritten bestehende, ausführbare Handlungsvorschrift zur eindeutigen Umwandlung von Eingabe- in Ausgabedaten

Allgemeine Charakteristika

1. Berechenbar

1. Finitheit: Algorithmus hat endliche Beschreibung
2. Terminierung: Algorithmus stoppt in endlicher Zeit
3. Effektivität: Schritte sind auf Maschine ausführbar

2. Bestimmt

1. Determiniertheit: Algorithmus liefert gleiche Ausgabe bei gleicher Eingabe
2. Determinismus: Algorithmus durchläuft gleiche Zustände bei gleicher Eingabe

3. Anwendbar

1. Allgemeinheit: Algorithmus für ganze Problemklasse anwendbar
2. Korrektheit: Falls Algorithmus terminiert, ist die Ausgabe richtig

Datenstrukturen

Eine Datenstruktur ist eine Methode um Daten für den Zugriff und die Modifikation zu organisieren
Datenstrukturen beinhalten:

1. Daten
2. Strukturbestandteile (Arrayindizes o.ä.)

Abstrakte Datentypen

z.B. Stack, hat nur abstrakte Operationen.

Datenstruktur

näher an der Maschine, z.B. Stack als Array

Datenstrukturen in Algorithmen

- Algorithmen verwenden Datenstrukturen
- Datenstrukturen wirken sich auf Effizienz aus

2. Sorting

Das Sortierproblem

Gegeben: Folge von Objekten
Gesucht: Sortierung gemäß bestimmten Schlüsselwertes

Schlüsselproblem

Schlüssel müssen nicht eindeutig, aber sortierbar sein
Im Folgenden Annahme, dass es totale Ordnung $\leq$ auf der Menge M aller Schlüsselwerte gibt
Betrachtung von Schlüsselwerten ohne Satellitendaten, meist Zahlen
Satellitendaten sind "unnötig", nur Schlüsselwert ist relevant

Insertion Sort

```
insertionSort(A)
        FOR i=1 TO A.length-1 DO
        // insert A[i] in pre-sorted sequence A[0..i-1]
        key=A[i];
        j=i-1; // search for insertion point backwards
        WHILE j>=0 AND A[j]>key DO
                A[j+1]=A[j]; // move elements to right
                j=j-1;
        A[j+1]=key;
```

- A ist ein Array/Liste/...
- $A[0..i-1]$ ist immer bereits sortiert
- Wert an der Stelle $A[i]$ wird dann im sortierten Bereich an der richtigen Stelle eingefügt, dabei wird alles immer verschoben

Laufzeitanalysen: O-Notation

Wie viele Schritte macht ein Algorithmus in Abhängigkeit von der Eingabekomplexität?

- Man nimmt meist Worst-Case für alle Eingaben gleicher Komplexität
- (Worst-Case-)Laufzeit: $T(n) = \max\{\text{Anzahl Schritte}\}$ für eine Aufgabe
- Komplexität wird meist von einem Faktor dominiert, wie z.B. der Anzahl zu sortierender Zahlen n

Laufzeitanalyse für einen Algorithmus

- Nehme ein festes n, z.B. Anzahl zu sortierender Elemente
- Wie oft wird jede Zeile maximal ausgeführt (in Abhängigkeit von n)?
- Jeder Zeile i wird Aufwand c_i zugeordnet, wird dann mit Anzahl der Ausführungen multipliziert
- Elementare Operationen (Zuweisung, Vergleich,...) haben konstanten Aufwand 1
- $T(n)$ ist dann sehr komplex, siehe Insertion Sort-Beispiel

Asymptotische Vereinfachung

- 1. Vereinfachung: Man nimmt nur dominanten Term $D(n)$ von $T(n)$
- 2. Vereinfachung: Nur abhängigen $A(n)$ Term betrachten, Vorfaktoren entfernen
  - Konstante Vorfaktoren sind von Berechnungsmodell, Leistung abhängig

Θ-Notation/Landau-Symbole

Seien $f, g : \mathbb{N} \to \mathbb{R}_{>0}$ Funktionen, $\mathbb{N}$ ist die Eingabekomplexität, $\mathbb{R}_{>0}$ die Laufzeit.

$$\Theta(g) := \{f : \exists c_1, c_2 \in \mathbb{R}, n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

Schreibweise: $f \in \Theta(g), f = \Theta(g)$

- $g(n)$ ist eine asymptotisch scharfe Schranke von $f(n)$
- Θ-Notation beschränkt eine Funktion asymptotisch von oben und unten
- Beispiel: Insertion Sort: $T(n) \in \Theta(n^2)$ für $c_1 = \frac{1}{2}, c_2 = 7, n_0 = 2$

O-Notation

g ist obere Schranke von f

$$O(g) := \{f : \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) \leq g(n)\}$$

Sprechweise: f wächst höchstens so schnell wie g
Schreibweise: $f = O(g), f \in O(g)$
$\Theta(g(n)) \subseteq O(g(n)) \rightsquigarrow f(n) \in \Theta(g) \Rightarrow f(n) \in O(g)$

Rechenregeln

- Konstanten: $f(n) = a, a \in \mathbb{R}_{>0} \Rightarrow f(n) \in O(1)$
- Skalarmultiplikation: $f \in O(g), a \in \mathbb{R}_{>0} \Rightarrow a \cdot f \in O(g)$
- Addition: $f_1 \in O(g_1), f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(\max\{g_1, g_2\}), \max$ ist punktweise
- Multiplikation: $f_1 \in O(g_1), f_2 \in O(g_2) \Rightarrow f_1 \cdot f_2 \in O(g_1 \cdot g_2)$

Ω-Notation

g ist untere Schranke von f

$$\Omega(g) := \{f : \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, 0 \leq cg(n) \leq f(n)\}$$

Sprechweise: f wächst mindestens so schnell wie g
Schreibweise: $f = \Omega(g), f \in \Omega(g)$
$\Theta(g(n)) \subseteq \Omega(g(n)) \rightsquigarrow f(n) \in \Theta(g) \Rightarrow f(n) \in \Omega(g)$

Zusammenhang O, Ω, Θ

$f(n) \in \Theta(g(n))$ gdw. $f(n) \in O(g(n))$ und $f(n) \in \Omega(g(n))$

Anwendung O-Notation

$f = O(g)$ ist üblich, $f \in O(g)$ ist wahre Bedeutung und besser, da $O(g)$ Menge ist
$O(n^4) = O(n^5)$ gilt, nicht jedoch $O(n^5) = O(n^4)$!

Ungleichungen

- $\leq$ nur mit O verwenden
- $\geq$ nur mit Ω verwenden

Insertion Sort Beispiel

Algorithmus macht maximal $T(n)$ viele Schritte, $T(n) \in \Theta(n^2)$
$\rightsquigarrow$ Laufzeit $\leq T(n) \in O(n^2)$
Für "gute" Eingaben (bereits vorsortiert) macht Algorithmus $\Theta(n)$ viele Schritte
Es wird aber mit Worst-Case gearbeitet, Insertion Sort hat quadratische Laufzeit

Komplexitätsklassen

Klasse	Bezeichnung	Beispiel
$\Theta(1)$	Konstant	Einzeloperation
$\Theta(\log n)$	Logarithmisch	Binäre Suche
$\Theta(n)$	Linear	Sequentielle Suche
$\Theta(n \log n)$	Quasilinear	Sortieren eines Arrays
$\Theta(n^2)$	Quadratisch	Matrixaddition
$\Theta(n^3)$	Kubisch	Matrixmultiplikation
$\Theta(n^k)$	Polynomiell	
$\Theta(2^n)$	Exponentiell	Travelling-Salesman
$\Theta(n!)$	Faktoriell	Permutationen

o-Notation, ω-Notation

Gelten für alle Konstanten, nicht nur eine

$$o(g) := \{f : \forall c \in \mathbb{R}_{>0}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) < cg(n)\}$$

$2n \in o(n^2), 2n^2 \notin o(n^2)$

$$\omega(g) := \{f : \forall c \in \mathbb{R}_{>0}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq cg(n) < f(n)\}$$

$\frac{n^2}{2} \in \omega(n), \frac{n^2}{2} \notin \omega(n^2)$

Bubble Sort

```
bubbleSort(A)
        FOR i=A.length-1 DOWNTO 0 DO
```

```
                FOR j=0 TO i-1 DO
                        IF A[j]>A[j+1] THEN SWAP(A[j],A[j+1]);
                        // SWAP: temp=A[j+1]; A[j+1]=A[j]; A[j]=temp;
```

- Quadratische Laufzeit
- Große Werte "steigen nach oben" und sammeln sich am Ende
- $A[i..A.length-1]$ ist nach jedem Durchlauf der äußeren Schleife korrekt

Merge Sort

Idee: Divide & Conquer (& Combine)

Teile Liste in Hälften, sortiere (rekursiv) Hälften, sortiere wieder zusammen
(Teil-)Sortierung erfolgt im Array selbst, Teillisten werden genutzt
Siehe auch 7. Advanced Designs

Algorithmus

```
mergeSort(A,l,r) // initial call: l=0,r=A.length-1
        IF l<r THEN // more than one element
                m=floor((l+r)/2); // m (rounded down) middle
                mergeSort(A,l,m); // sort left part
                mergeSort(A,m+1,r); // sort right part
                merge(A,l,m,r); // merge into one

merge(A,l,m,r) // requires l<=m<=r
                //array B with r-l+1 elements as temporary storage
        pl=l; pr=m+1; // position left, right
        FOR i=0 TO r-l DO // merge all elements
                IF pr>r OR (pl=<m AND A[pl]=<A[pr]) THEN
                        B[i]=A[pl];
                        pl=pl+1;
                ELSE //next element at pr
                        B[i]=A[pr];
                        pr=pr+1;
        FOR i=0 TO r-l DO A[i+l]=B[i]; //copy back to A
```

- Es wird zwischen Position l und r sortiert
- m ist der letzte Index des linken Teils
- Es wird aufgeteilt, bis die Teillisten Länge 1 haben
- Dann werden sie zusammengefügt und dabei sortiert
  - $merge$ nimmt immer das kleinste Element aus den beiden Listen und fügt es der Ergebnisliste in B hinzu
- Laufzeit $\Theta(n \cdot \log n)$
- $T(n) \geq \Omega(n \cdot \log n)$

Laufzeitanalyse: Rekursionsgleichungen

Rekursion manuell iterieren

Beispiel Merge Sort, $T(n)$ ist max. Anzahl an Schritten für Arrays der Größe n:
$T(n) \leq 2T(\frac{n}{2}) + c + dn \leq \cdots \leq 2^{\log_2 n} \cdot c + \log_2 n \cdot cn \in O(n \log n)$

Allgemeiner Ansatz: Mastermethode

Allgemeine Form der Rekursionsgleichung:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n), T(n) \in \Theta(1)$$

mit $a \geq 1, b > 1, f(n)$ asymptotisch positive Funktion.

Interpretation

- Problem wird in a Teilprobleme der Größe $\frac{n}{b}$ aufgeteilt
- Lösen jeder der a Teilprobleme benötigt Zeit $T(\frac{n}{b})$
- $f(n)$ umfasst Kosten für Aufteilen und Zusammenfügen

Mastertheorem

Seien $a \geq 1, b > 1$ konstant, $f(n)$ eine positive Funktion und $T(n)$ über den nicht-negativen ganzen Zahlen durch folgende Rekursiongleichung definiert:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n), T(1) \in \Theta(1)$$

$\frac{n}{b}$ wird hierbei entweder auf- oder abgerundet.
Dann besitzt $T(n)$ die folgenden asymptotischen Schranken:

1. Gilt $f(n) \in O(n^{\log_b(a)-\epsilon})$ für ein $\epsilon > 0$, dann gilt $T(n) \in \Theta(n^{\log_b a})$
2. Gilt $f(n) \in \Theta(n^{\log_b a})$, dann gilt $T(n) \in \Theta(n^{\log_b a} \cdot \log_2 n)$
3. Gilt $f(n) \in \Omega(n^{\log_b(a)+\epsilon})$ für ein $\epsilon > 0$ und $af\left(\frac{n}{b}\right) \leq cf(n)$ für ein $c < 1$ und hinreichend große n, dann ist $T(n) \in \Theta(f(n))$

Interpretation

Entscheidend ist das Verhältnis von $f(n)$ zu $n^{\log_b a}$:

1. Wenn $f(n)$ polynomiell kleiner als $n^{\log_b a}$, dann gilt $T(n) \in \Theta(n^{\log_b a})$
2. Wenn $f(n), n^{\log_b a}$ gleiche Größenordnung, dann $T(n) \in \Theta(n^{\log_b a} \cdot \log n)$
3. Wenn $f(n)$ polynomiell größer als $n^{\log_b a}$ und $af\left(\frac{n}{b}\right) \leq cf(n)$, dann $T(n) \in \Theta(f(n))$

- Regularität Fall 3: $af\left(\frac{n}{b}\right) \leq cf(n), c < 1$
- $f(n)$ dominiert asymptotisch den Ausdruck
- Fall 3 bedeutet, dass die Wurzel den größten Arbeitsaufwand verrichtet, diese wird hiermit sichergestellt
  Wenn das Mastertheorem nicht anwendbar ist, ist die Baumstruktur zu analysieren

Quicksort

Idee

- Divide & Conquer
- Mehr Arbeit in Aufteilen, Zusammenfügen kostenlos
- Wählt 1. Element als Pivot-Element
- Dann Partitionieren der Elemente, sodass $\leq$ Pivot links, $\geq$ Pivot rechts
- Rekursiv fortsetzen

Algorithmus

```
quicksort(A,l,r) // initial call: l=0,r=A.length-1
        IF l<r THEN //more than one element
                p=partition(A,l,r); // p partition index
                quicksort(A,l,p); // sort left part
                quicksort(A,p+1,r); // sort right part

partition(A,l,r) //requires l<r, returns int in l..r-1
        pivot=A[l];
        pl=l-1; pr=r+1; //move from left resp. right
        WHILE pl<pr DO
                REPEAT pl=pl+1 UNTIL A[pl]>=pivot; //move left up
                REPEAT pr=pr-1 UNTIL A[pr]=<pivot; //move right down
                IF pl<pr THEN Swap(A[pl],A[pr]);
                p=pr; //store current value
        return p // A[l..p] left, A[p+1..r] right
```

- Wenn das übergebene Array nur 1 Element hat, wird nichts getan
- Partition:
  - `pl, pr` werden vergrößert/verkleinert, bis das Element an der Position nicht passt (größer/kleiner als `pivot`)
  - falls dann `pl` noch kleiner als `pr` ist, sind die beiden Elemente zu vertauschen
  - Wiederholung
  - Am Ende, nachdem alles vertauscht wurde, wird der letzte Wert von `pr` zurückgegeben, dies ist dann der letzte Index des linken Teilarrays
- Wenn alle Teilarrays Größe 1 haben, ist man fertig

Laufzeit

Worst-Case

- Immer nur Arrays der Größe 1 abgespalten
- $\Theta(n^2)$

Best-Case

- Aufteilung in gleich große Arrays
- $\Theta(n \log n)$

Average Case

- Nehme erwartete Anzahl von Schritten über eine Verteilung der Komplexität n
- $T(n) = E_{D(n)}[t]$, t ist Anzahl der Schritte für x
- $O(n \log n)$

Randomisierte Variante

```
partition(A,l,r) //requires l<r, returns int in l..r-1
        j=RANDOM(l,r); Swap(A[l],A[j]); //j uniform in [l..r]
        pivot=A[l];
        ...
```

- Wähle zufälliges Element, vertausche es dann mit 1. Element, sonst alles gleich

Erwartete Laufzeit (Average-Case)

- Zufällige Wahl des Pivot-Elementes teilt Array im Durchschnitt mittig, unabhängig davon, wie Array aussieht
- Worst-Case: $T(n) = \max\{\#\text{steps for x}\}$
- Erwartete Laufzeit: $T(n) = \max\{E_A[\#\text{steps for x}]\}$
  - zufällige Wahl des Algorithmus A für schlechteste Eingabe, Komplexität n
  - $O(n \log n)$

Vergleich

Insertion Sort

- $\Theta(n^2)$
- Einfach
- Für kleine $n \leq 50$ beste Wahl

Merge Sort

- Beste asymptotische Laufzeit $\Theta(n \log n)$

Quicksort

- Worst-Case $\Theta(n^2)$, randomisiert erwartet $\Theta(n \log n)$
- Praxis: Schneller als Merge Sort, da weniger Kopieroperationen
- Implementierungen nutzen Insertion Sort für kleine n

Untere Schranke für vergleichsbasiertes Sortieren

Hier werden nur deterministische Algorithmen betrachtet, im Durchschnitt gilt dies aber auch für randomisierte Algorithmen

Genereller Algorithmus

```
sortByComp(n) // n is size of input-array A
// returns array I with sorted indexes:
// A[ I[i] ] =< A[ I[i+1] ] for i=0,…,n-1
        done=false;
        WHILE !done DO
                determine (i,j); // arbitrarily
                comp(i,j); // returns A[i] =< A[j]?
                set done; // true or false
        compute I from comp-information only;
        return I
```

- Erhält Informationen über `A` nur durch Vergleichsresultate für gewählte Indizes `i,j`
- Alle Sortieralgorithmen bisher sind vergleichsbasiert

Theorem der unteren Schranke

Jeder (korrekte) vergleichsbasierte Sortieralgorithmus muss mindestens $\Omega(n \log n)$ viele Vergleiche machen.

Radix-Sort

Ansatz

- Schlüssel sind d-stellige Werte in D-närem Zahlensystem
- "Buckets" erlauben Einfügen, Entnehmen in eingefügter Reihenfolge
  - konstanter Zeitaufwand
  - Umsetzung durch Queues

Algorithmus

```
radixSort(A) // keys: d digits in range [0,D-1]
// B[0][],…, B[D-1][] buckets (init: B[k].size=0)
        FOR i=0 TO d-1 DO //0 least, d-1 most sign. digit
                FOR j=0 TO n-1 DO putBucket(A,B,i,j);
                a=0;
                FOR k=0 TO D-1 DO //rewrite to array
                        FOR b=0 TO B[k].size-1 DO
                                A[a]=B[k][b]; //read out bucket in order
                                a=a+1;
                        B[k].size=0; //clear bucket again
        return A

putBucket(A,B,i,j) // call-by-reference
        z=A[j].digit[i]; // i-th digit of A[j]
        b=B[z].size; // next free spot
        B[z][b]=A[j];
        B[z].size=B[z].size+1;
```

- i-te Iteration ($i \in [0..d-1]$):
  1. Sortiere Zahlen anhand i. Ziffer in entsprechenden Bucket
  2. Gehe aufsteigend durch Buckets und führe in nächster Stelle im Array ein
- Mit höchstwertiger Ziffer beginnen funktioniert nicht

Laufzeit

$O(d \cdot (n + D))$
D oft als konstant angesehen $\rightsquigarrow O(dn)$
Linear, wenn d auch als konstant angesehen
Eindeutige Schlüssel für n Elemente benötigen $d = \Theta(\log_D n)$ Ziffern $\rightsquigarrow O(n \log n)$

3. Basic Data Structures

Stacks

Abstrakter Datentyp Stack

- `new(S)`: neuer, leerer Stack `S`
- `isEmpty(S)`: boolean, ob `S` leer
- `pop(S)/pop()`: löscht oberstes Element von `S`, gibt es zurück; Fehler wenn `S` leer
- `push(S,k)/S.push(k)`: `k` als oberstes Element auf `S`; Fehler, wenn `S` voll
  LIFO: last in, first out

Beispiel Bitcoin

Bitcoin nutzt Stacks, um verschiedene Werte während dem Verifikationsprozess zu speichern.

Stacks als Array

- Annahme: maximale Größe `MAX` des Stacks vorher bekannt
- Zeiger `S.top` zeigt auf oberstes Element
- Zeiger wird bei Operationen passend bewegt

Alogrithmen

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

isEmpty(S)
        IF S.top<0 THEN
```

```
                return true
        ELSE
                return false;

pop(S)
        IF isEmpty(S) THEN
                error 'underflow'
        ELSE
                S.top=S.top-1;
                return S.A[S.top+1];

push(S,k)
        IF S.top==MAX-1 THEN
                eror 'overflow'
        ELSE
                S.top=S.top+1;
                S.A[S.top]=k;
```

Stacks variabler Größe

Wenn voll:

- Kopiere in größeres, zusammenhängendes Array oder
- Verteile auf viele Arrays, Siehe Verkettete Listen

Einfache Lösung

Wenn voll, Array mit 1 Feld mehr erstellen, alles kopieren

Laufzeit

Wenn n Elemente in Array, n `push`-Befehle führen zu $\Omega(n^2)$ Kopier-Schritten
Durchschnittlich $\Omega(n)$ Kopier-Schritte pro `push`

Was tun

- Trivial: Unendlich viel speicher reservieren
- Gesucht: Lösung die maximal jeweils $O(\#\text{Elemente})$ braucht
  - Wenn Grenze erreicht, verdopple Speicher und kopiere um
  - Schrumpfe und kopiere, wenn weniger als $\frac{1}{4}$ benötigt

Algorithmen, Laufzeitanalyse

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

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

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

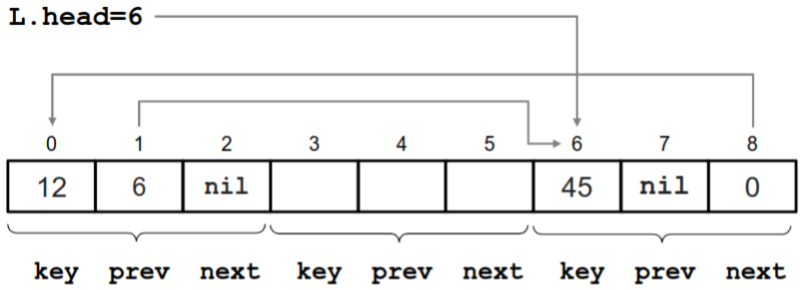
- `RESIZE(A,m)` reserviert neuen Speicher der Größe `m`, kopiert `A` um, fixt Referenz
- Im Schnitt für jeden der mindestens n Befehle $\Theta(1)$ Umkopierschritte

Verkettete Listen

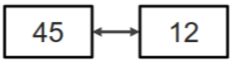
Datenstruktur doppelt verkettete Liste

- Element `x` besteht aus:
  - `key`: Wert
  - `prev`: Zeiger auf Vorgänger/`nil`
  - `nexxt`: Zeiger auf Nachfolger/`nil`
- `head` zeigt auf erstes Element (`nil` für leere Liste)

Verkettete Listen durch Arrays



entspricht doppelt verketteter Liste



Elementare Operationen auf verketteten Listen

Im Pseudocode wird wie in Java von short circuit evaluation und call by reference/value verwendet

Suche

```
search(L,k) // returns pointer to k in L (or nil)
        current=L.head;
        WHILE current != nil AND current.key != k DO // short circuit evaluation
                current=current.next;
        return current;
```

Laufzeit = $\Theta(n)$

Einfügen

```
insert(L,x) // inserts element x in L
        x.next=L.head;
        x.prev=nil;
        IF L.head != nil THEN
                L.head.prev=x;
        L.head=x;
```

Laufzeit = $\Theta(1)$

- Prüft nicht, ob Wert bereits in Liste ist
- Wenn zuerst suche nach Wert stattfinden soll, $\Omega(n)$

Löschen

```
delete(L,x) // deletes element x from L
IF x.prev != nil THEN
        x.prev.next=x.next
ELSE
        L.head=x.next;
IF x.next != nil THEN
        x.next.prev=x.prev;
```

Laufzeit = $\Theta(1)$

- `x` ist Verweis auf zu löschendes Element
- Wenn Wert gelöscht werden soll, muss dieser erst gesucht werden $\rightsquigarrow \Omega(n)$

Vereinfachung per Wächter/Sentinels

- Ziel: eliminiere die Spezialfälle für Listenanfang/-ende
- Sentinel `L.sent` hinzugefügt, `head = L.sent.next`, `head.prev = L.sent`, `L.sent.key = nil`
- Für letztes Element `x` gilt: `x.next = L.sent` und `L.sent.prev = x`
- Sentinel ist von außen nicht sichtbar
- Leere Liste besteht nur aus Sentinel

Löschen mit Sentinels

```
deleteSent(L,x) // deletes x from L with sentinel
        x.prev.next=x.next;
        x.next.prev=x.prev;
```

Andere Operationen müssen auch angepasst werden

Queues

Abstrakter Datentyp Queue

- `new(Q)`: Erzeugt neue, leere Queue namens `Q`
- `isEmpty(Q)`: Gibt an, ob `Q` leer
- `dequeue(Q)`: Gibt vorderstes Element aus `Q` zurückt, löscht es aus `Q`, Fehler wenn `Q` leer

- `enqueue(Q,k)`: Schreibt `k` als neues hinterstes Element auf `Q`, Fehler wenn `Q` voll
- FIFO: first in, first out

Queues als virtuelles, zyklisches Array

- Problem mit Array-Implementierung:
  Queue "wandert", wenn Werte eingefügt/entfernt werden
- Führe `Q.rear, Q.front` für Zeiger auf Anfang und Ende ein
- Es gibt Ein Maximum für die Anzahl gleichzeitig in einer Queue: `MAX`
- Wenn `Q.rear, Q.front` auf selben Wert verweisen:
  - Speichere boolean `empty`, um anzugeben, ob Array vol oder leer
  - Alternativ: reserviere ein Element des Arrays als Abstandshalter

Algorithmen

- `Q` leer, wenn `front==rear` und `empty==true`
- `Q` voll, wenn `front==rear` und `empty==false`

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

isEmpty(Q)
        return Q.empty;

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

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

Queues durch einfach verkettete Listen

`front` und `rear` sind nun Zeiger auf Listenelemente

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

isEmpty(Q)
        IF Q.front==nil THEN
                return true
        ELSE
                return false;

dequeue(Q)
        IF isEmpty(Q) THEN
                error 'underflow'
        ELSE
                x=Q.front;
                Q.front=Q.front.next;
                return x;

enqueue(Q,x)
        IF isEmpty(Q) THEN
                Q.front=x;
        ELSE
                Q.rear.next=x;
        x.next=nil;
        Q.rear=x;
```

Anzahl Operationen Queues, Stacks, verkettete Listen

- Stack:
  - Push: $\Theta(1)$
  - Pop: $\Theta(1)$
- Queue:
  - Enqueue: $\Theta(1)$
  - Dequeue: $\Theta(1)$
- Verkettete Liste:
  - Einfügen: $\Theta(1)$
  - Löschen: $\Theta(1)$
  - Suchen: $\Theta(n)$
  - Löschen eines Wertes: $\Omega(n)$

Binäre Bäume

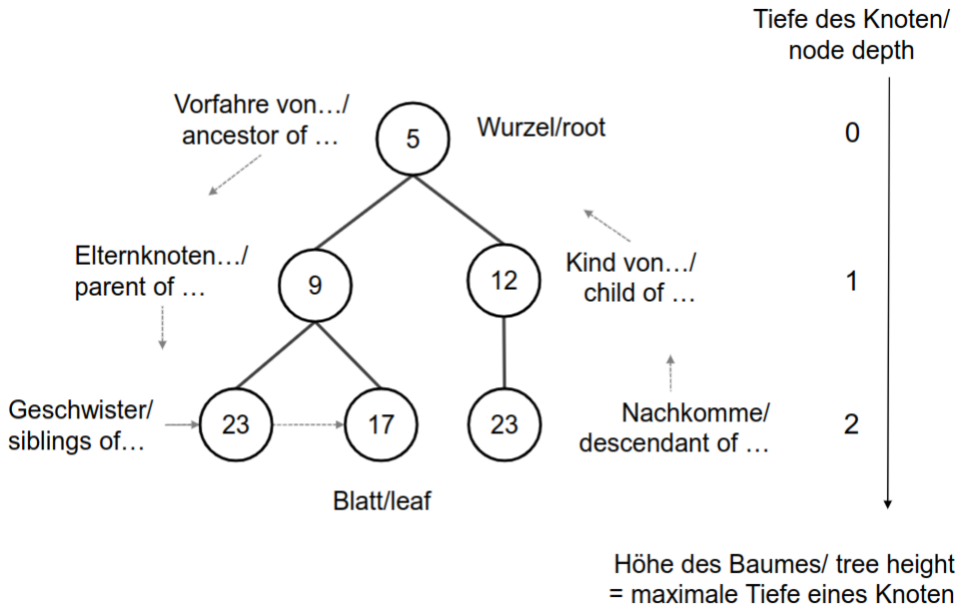
Bäume durch verkettete Listen

- `T.root` verweist auf Wurzelknoten des Baumes `T`
- Jeder Knoten enthält:
  - `key`: Wert
  - `child[]`: Array von Zeigern auf Kinder
  - manchmal auch `parent`: Zeiger auf Elternknoten
    Baum-Bedingung:
    Baum ist leer oder es gibt einen Knoten `r` (Wurzel), sodass jeder Knoten `v` von der Wurzel aus per eindeutiger Sequenz von `child`-Zeigern erreichbar ist:
    `v = r.child[i_1].child[i_2]. .....child[i_m]`

Eigenschaften von Bäumen

- Bäume sind azyklisch
- Für nicht-leeren Baum gibt es genau $\#Knoten - 1$ viele Einträge $\neq nil$ über alle Listen `child[]`
- Man kann Bäume als (ungerichtete) Graphen darstellen, jedoch ist hier dann die Reihenfolge der Kinder relevant, da sie `child[]` abbilden muss

Begrifflichkeiten



Binärbaum

Jeder Knoten hat maximal 2 Kinder: `left = child[0], right = child[1]`
Ausgangsgrad/outdegree jedes Knotens ist ≤ 2
Markiere Knoten auch graphisch als linkes/rechtes Kind
Halbblatt: Knoten mit genau einem Kind
linker/rechter Teilbaum eine Knotens: Baum der links/rechts am Knoten hängt, d.h. der Baum, der den linken/rechten Kindknoten als Wurzel hat
Höhe des leeren Baumes ist -1
Höhe nicht-leeren Baumes = $\max\{\text{Höhe aller Teilbäume der Wurzel}\} + 1$

Inorder-Traversieren von Binärbäumen

Beispielanwendung: Serialisierung

```
inorder(x)
        IF x != nil THEN
                inorder(x.left);
                print x.key;
                inorder(x.right);
```

Bei Bedarf mit Wrapper: `inoderTree(T) = inorder(T.root)`
$T(n)$ = Laufzeit bei n Knoten, $T(n) \in O(n)$
Verschiedene Bäume können gleiche Inorder haben

Pre- und Postorder-Traversieren von Binärbäumen

```
preorder(x)
        IF x != nil THEN
                print x.key;
                preorder(x.left);
                preorder(x.right);

postorder(x)
        IF x != nil THEN
                postorder(x.left);
                postorder(x.right);
                print x.key;
```

Preorder kann für Syntaxbäume bei funktionalen Programmiersprachen genutzt werden
Siehe auch FOP #TODO Verweis auf Racket einfügen

Preorder-Traversieren für Kopieren

1. Betrachte Knoten und lege Kopie an
2. Wiederhole die rekursive für Teilbäume

Postorder-Traversieren für Löschen

1. Postorder löscht Teilbäume
2. Postorder betrachtet Knoten, an dem die Teilbäume hängen, erst danach, löscht zuletzt
   Verschiedene Bäume können gleiche Pre-/Postorder haben

Binärbaum aus Preorder, Inorder und eindeutigen Werten

1. Preorder identifiziert Wurzel
2. Inorder identifiziert Werte im rechten/linken Teilbaum
3. Bilde Teilbäume rekursiv
   Statt Pre- auch Postorder möglich

Abstrakter Datentyp Baum

- `new(T)`: Erzeugt neuen Baum `T`
- `search(T,k)`: Gibt Element `x` aus `T` mit `x.key==k` oder `nil` zurück
- `insert(T,x)`: Fügt `x` in `T` ein
- `delete(T,x)`: Löscht `x` aus `T`
  Oft gibt es weitere Baum-Operationen wie Wurzel, Höhe, Traversieren, ...

Suchen

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

Starte mit `search(T.root,k)`
Laufzeit = $\Theta(n)$
Jeder Knoten wird maximal einmal besucht, im schlechtesten Fall aber auch jeder Knoten

Einfügen

```
insert(T,x) // x.parent==x.left==x.right==nil;
        IF T.root != nil THEN
                T.root.parent=x;
                x.left=T.root;
        T.root=x;
```

Laufzeit = $\Theta(1)$
Erzeugt linkslastigen Baum

Löschen

Idee: Ersetze `x` durch Halbblatt ganz rechts, es gibt auch andere Möglichkeiten
Sonderfälle beachten: Halbblatt hat selbst Wert `x` oder ist Wurzel

```
delete(T,x) // assumes x in T
        y=T.root;
        WHILE y.right!=nil DO
                y=y.right;
        connect(T,y,y.left);
        IF x != y THEN
                y.left=x.left;
                IF x.left != nil THEN
                        x.left.parent=y;
                y.right=x.right;
                IF x.right != nil THEN
                        x.right.parent=y;
                connect(T,x,y);

connect(T,y,w) // connects w to y.parent
        v=y.parent;
        IF y != T.root THEN
                IF y == v.right THEN
                        v.right=w;
                ELSE
                        v.left=w;
        ELSE
                T.root=w;
        IF w != nil THEN
                w.parent=v;
```

Bei `connect` muss `w` nicht an `y` hängen
Laufzeit `connect` = $\Theta(1)$
Laufzeit `delete` = $\Theta(h)$, h ist Höhe des Baumes, $h = n$ ist möglich

Binäre Suchbäume (Binary Search Tree, BST)

Wir nehmen totale Ordnung auf den Werten an
Binärer Suchbaum: Binärbaum, sodass für alle Knoten `z` gilt:

- Wenn `x` Knoten im linken Teilbaum von `z`, dann `x.key <= z.key`
- Wenn `y` Knoten im rechten Teilbaum von `z`, dann `y.key >= z.key`

Order und eindeutige Werte

Aus Pre-/ Postorder und eindeutigen Werten kann man eindeutige BST konstruieren

1. Identifiziere Wurzel
2. Identifiziere Werte anhand der Regeln
3. Bilde Teilbäume rekursiv
   Mit Inorder und eindeutigen Werten lässt sich kein eindeutiger BST konstruieren

Suche

```
search(x,k) // first call x=root
        IF x==nil OR x.key==k THEN
                return x;
        IF x.key > k THEN
                return search(x.left,k)
        ELSE
                return search(x.right,k);
```

Laufzeit $O(h)$, h Höhe des Baumes

Iterative Suche

```
iterative-search(x,k) // first call x=root
        WHILE x != nil AND x.key != k DO
                IF x.key > k THEN
                        x=x.left
                ELSE
                        x=x.right;
        return x;
```

Einfügen

```
insert(T,z) // may insert z again, z.left==z.right==nil
        x=T.root; px=nil;
        WHILE x != nil DO
                px=x;
                IF x.key > z.key THEN
                        x=x.left
                ELSE
                        x=x.right;
        z.parent=px;
        IF px==nil THEN
                T.root=z
        ELSE
                IF px.key > z.key THEN
                        px.left=z
                ELSE
                        px.right=z;
```

Laufzeit $O(h)$

Löschen

Zu löschender Knoten ist `z`, Fallunterscheidung:

- `z` hat maximal ein Kind:
  Kind anstelle von `z` setzen, fertig
  Wenn `z` Blatt ist, löschen trivial
  Bedingungen an Struktur/Werte bleiben erhalten
- Rechtes Kind von `z` hat kein linkes Kind:
  Analog: Linkes Kind von `z` hat kein rechtes Kind
  Rechtes Kind an die Stelle von `z` setzen, linkes Kind von `z` wird linkes Kind von rechtem Kind
  BST-Bedingung bleibt erhalten
- Kleinster Nachfahre vom rechten Kind von `z`:
  1. Finde kleinsten Nachfahren
  2. Ersetze `z` durch kleinsten Nachfahren
  3. Da kleinster Nachfahre kein linkes Kind haben kann, entsteht hier kein Problem
  4. Rechtes Kind des kleinsten Nachfahren an die Stelle des kleinsten Nachfahren

Transplantation

hängt Teilbaum `v` an Elternknoten von `u`

```
transplant(T,u,v)
        IF u.parent==nil THEN
                T.root=v
        ELSE
                IF u==u.parent.left THEN
                        u.parent.left=v
                ELSE
                        u.parent.right=v;
        IF v != nil THEN
                v.parent=u.parent;
```

Laufzeit = $\Theta(1)$

Algorithmus

```
delete(T,z)
        IF z.left==nil THEN
                transplant(T,z,z.right)
        ELSE
                IF z.right==nil THEN
```

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

Laufzeit = $O(h)$

Höhe des BST

Laufzeit

Verkettete Liste:

- Einfügen: $\Theta(1)$
- Löschen: $\Theta(1)$
- Suchen: $\Theta(n)$
  BST:
- Einfügen: $O(h)$
- Löschen: $O(h)$
- Suchen: $O(h)$
  BST ist besser, wenn viele Such-Operationen durchgeführt werden und h im Vergleich zu n relativ klein ist

Best-/Worst-Case

Best-Case:

- Vollständig: Alle Blätter haben gleiche Tiefe
- $h = O(\log_2 n)$
- Laufzeit = $O(\log_2 n)$
  Worst-Case:
- Degeneriert: Lineare Liste
- $h = n - 1$
- Laufzeit = $\Omega(n)$

Durchschnittliche Höhe

Analyse ohne Einfügen und Löschen

```
randomlyBuiltTree(D) // D data set
        T=newTree();
        WHILE D != ø DO
                Pick d uniformly from D;
                insert(T,newNode(d));
                remove d from D;
        return T;
```

Die erwartete Höhe $E[h]$ des Baumes `T`, erzeugt durch `randomlyBuiltTree(D)`, für eine Datenmenge `D` mit n Werten ist $E[h] = \Theta(\log_2 n)$.

Suchbäume als Suchindex

- Knoten speichert nur Primärschlüssel und Zeiger auf Daten
- Bereichssuche ist möglich
- Sekundärindizes/zusätzliche Indizes kosten Speicherplatz und sind daher nur sinnvoll, wenn oft nach ihnen gesucht wird
- Z.B. sekundärer Baum mit alphabetischer Sortierung für eine Suche auf Namen

4. Advanced Data Structures

Rot-Schwarz-Bäume (RS-Bäume)

Anwendung: Linux Completely Fair Scheduling

Verwendet RS-Bäume. um Worst-Case-Laufzeit $(\log n)$ zu erreichen
`key`: virtual run time eines Prozesses in Sekunden

1. Nächsten Prozess holen
2. Addiere zugewiesene Zeit
3. Füge mit aktualisierter Zeit wieder ein

Baumkunde

Ein RS-Baum ist ein binärer Suchbaum, sodass gilt:

1. Jeder Knoten ist rot oder schwarz (`x.color=red/black`)
2. Die Wurzel ist schwarz, wenn der Baum nicht leer ist
3. Wenn ein Knoten rot ist, sind seine Kinder schwarz (Nicht-Rot-Rot-Regel)
4. Für jeden Knoten hat jeder Pfad im Teilbaum zu einem Blatt/Halbblatt die gleiche Anzahl an schwarzen Knoten

Halbblätter

Halbblätter sind Knoten mit nur einem Kind
Halbblätter im RS-Baum sind schwarz, sonst wird direkt mindestens eine Regel verletzt

Schwarzhöhe eines Knoten

Die Schwarzhöhe eines Knoten `x` ist die eindeutige Anzahl an schwarzen Knoten auf dem Weg zu einem Blatt/Halbblatt im Teilbaum des Knoten
Für leeren Baum setzt man $SH(nil) = 0$

Höhe eines RS-Baums

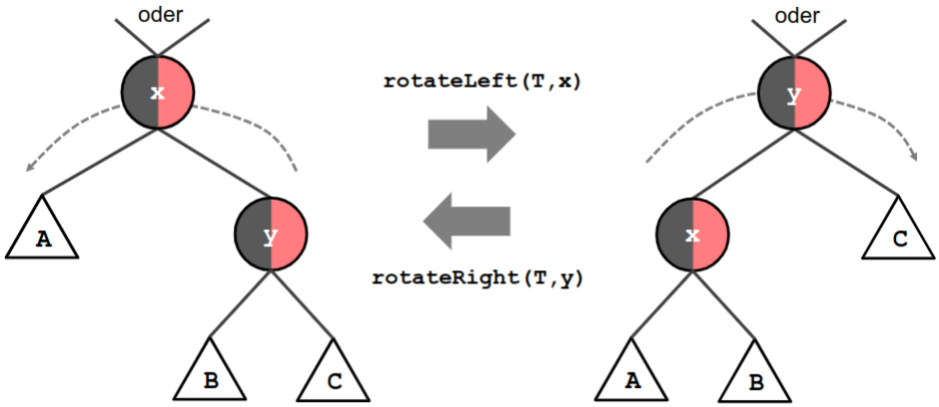
Ein RS-Baum mit n Knoten hat maximale Höhe $h \leq 2 \cdot \log_2(n+1)$
Intuition:

1. In jedem Unterteilbaum gleiche Anzahl schwarzer Knoten auf jedem Pfad
2. Maximal zusätzlich gleiche Anzahl roter Knoten auf diesem Pfad
3. Daher einigermaßen ausbalanciert und Höhe $O(\log n)$

Implementierungen mittels Sentinel

- `T.root.parent = T.sent`
- `T.sent.key=nil; T.sent.color=black;`
- `T.sent.parent = T.sent, T.sent.left=T.sent; T.sent.right=T.sent;`
- Alles immer wohldefiniert dank Einführuing von `T.sent` (Sentinel des Baumes T)

Rotation




```
rotateLeft(T,x) // x.right!=nil
        y=x.right;
        x.right=y.left;
        IF y.left != nil THEN
                y.left.parent=x;
        y.parent=x.parent;
        IF x.parent==T.sent THEN
                T.root=y
        ELSE
                IF x==x.parent.left THEN
                        x.parent.left=y
                ELSE
                        x.parent.right=y;
        y.left=x;
        x.parent=y;
```

- Rot-Schwarz-Baum-Bedingungen sind nach Rotation eventuell verletzt
- Laufzeit = $\Theta(1)$

Einfügen

1. Finde Elternknoten `y` wie im BST
2. Färbe neuen Knoten `z` rot
3. Stelle RS-Baum-Bedingung wieder her

```
insert(T,z) // z.left==z.right==nil;
        x=T.root; px=T.sent;
        WHILE x != nil DO
                px=x;
                IF x.key > z.key THEN
                        x=x.left
                ELSE
                        x=x.right;
        z.parent=px;
        IF px==T.sent THEN
                T.root=z
        ELSE
                IF px.key > z.key THEN
                        px.left=z
                ELSE
                        px.right=z;
        z.color=red;
        fixColorsAfterInsertion(T,z);
```

- Funktioniert wie beim BST mit Sentinel

Aufräumen

```
fixColorsAfterInsertion(T,z)
        WHILE z.parent.color==red DO
                IF z.parent==z.parent.parent.left THEN
                        y=z.parent.parent.right;
                        IF y!=nil AND y.color==red THEN
                                z.parent.color=black;
```

```
                        y.color=black;
                        z.parent.parent.color=red;
                        z=z.parent.parent;
                ELSE
                        IF z==z.parent.right THEN
                                z=z.parent;
                                rotateLeft(T,z);
                        z.parent.color=black;
                        z.parent.parent.color=red;
                        rotateRight(T,z.parent.parent);
        ELSE
        // do the same, but exchange left and right
T.root.color=black;
```

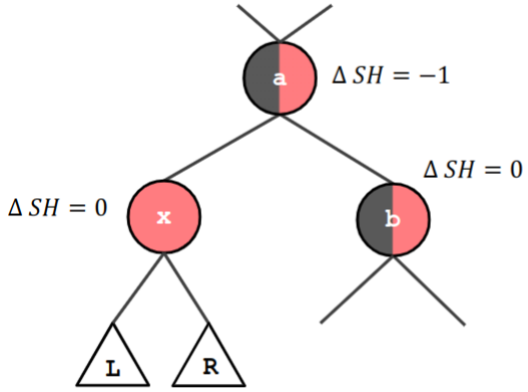
Laufzeit = $O(h) = O(\log n)$

Löschen

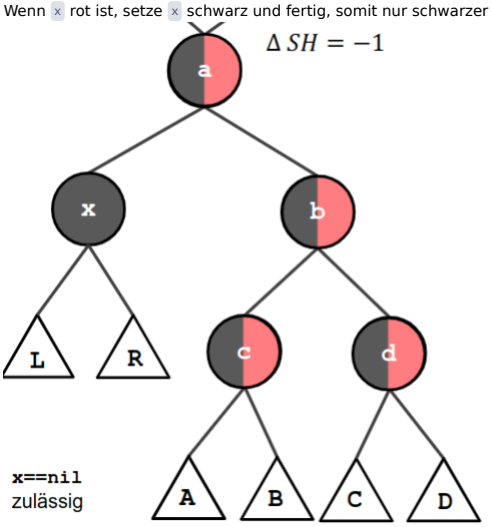
- Größtenteils analog zum BST
- Sei `z` der entfernte Knoten und `y` der Knoten, der `z` ersetzt.
  Dann erbt `y` die Farbe von `z`, wenn `y` schwarz war, müssen Farben angepasst werden

Fixup bei `y.color==black`

$\Delta SH := SH(\text{linker Teilbaum}) - SH(\text{rechter Teilbaum})$ f.a. Knoten
Beim Löschen kann die Schwarzhöhe nur sinken

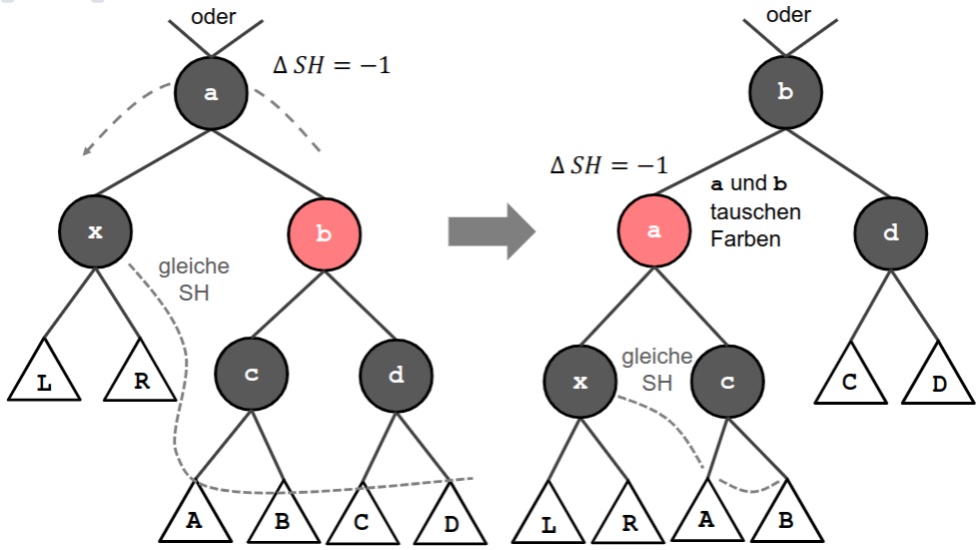


Fall $\Delta SH = 1$ in `a` ist analog
Wenn `x` rot ist, setze `x` schwarz und fertig, somit nur schwarzer Fall zu betrachten



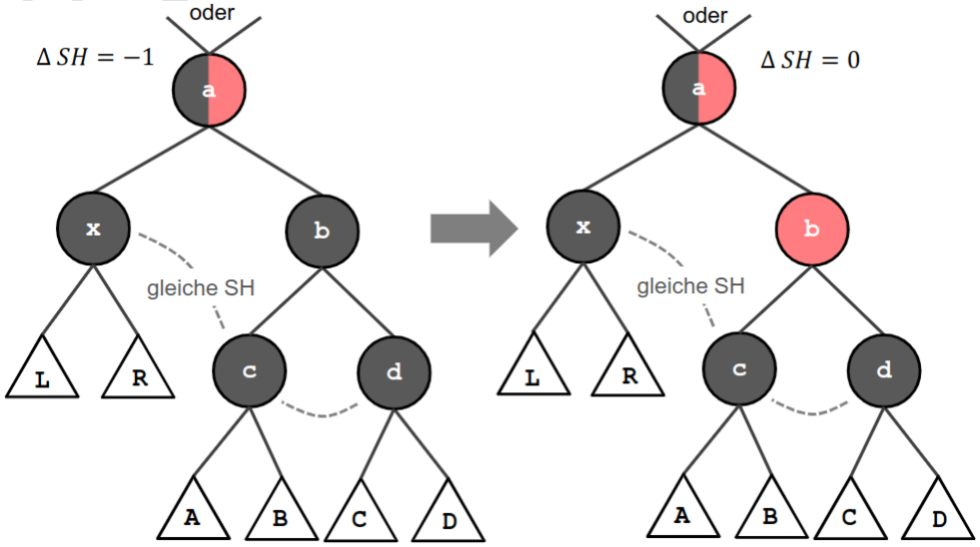
Fallunterscheidung:

1. `a` schwarz, `b` rot



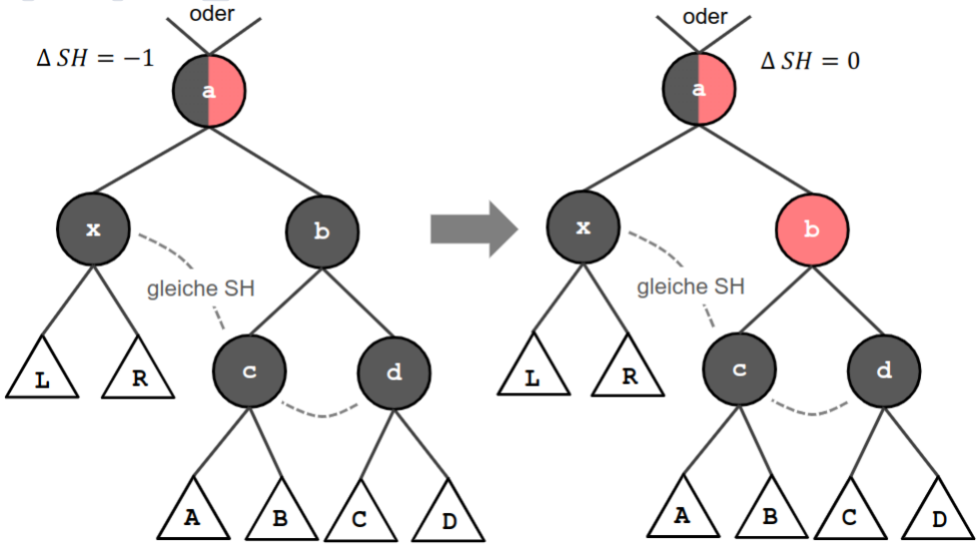
Wird zu allen Fällen außer 2b)

2. a) `a` rot, `b` schwarz, `c,d` nicht rot



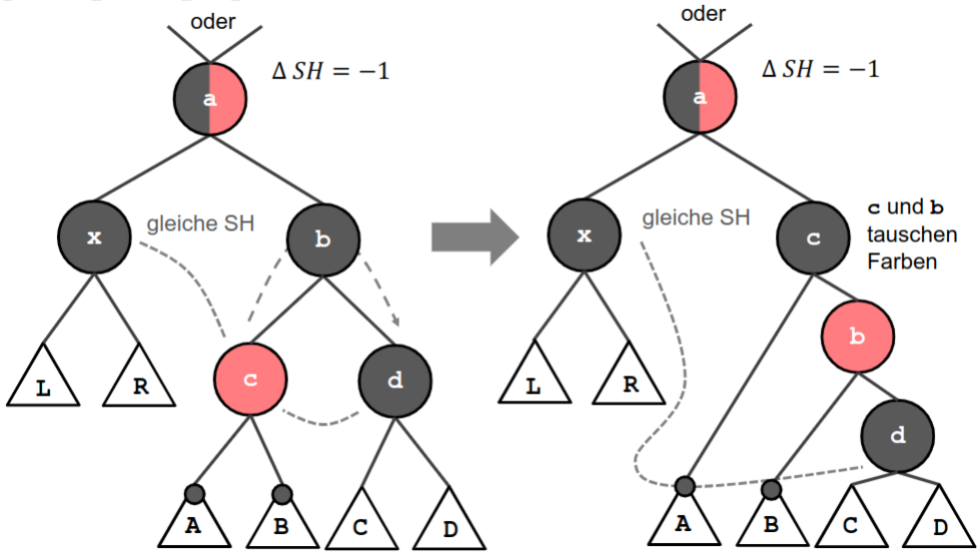
`b` auf schwarz setzen, um ursprüngliche SH zu erreichen

3. b) `a` schwarz, `b` schwarz, `c,d` nicht rot



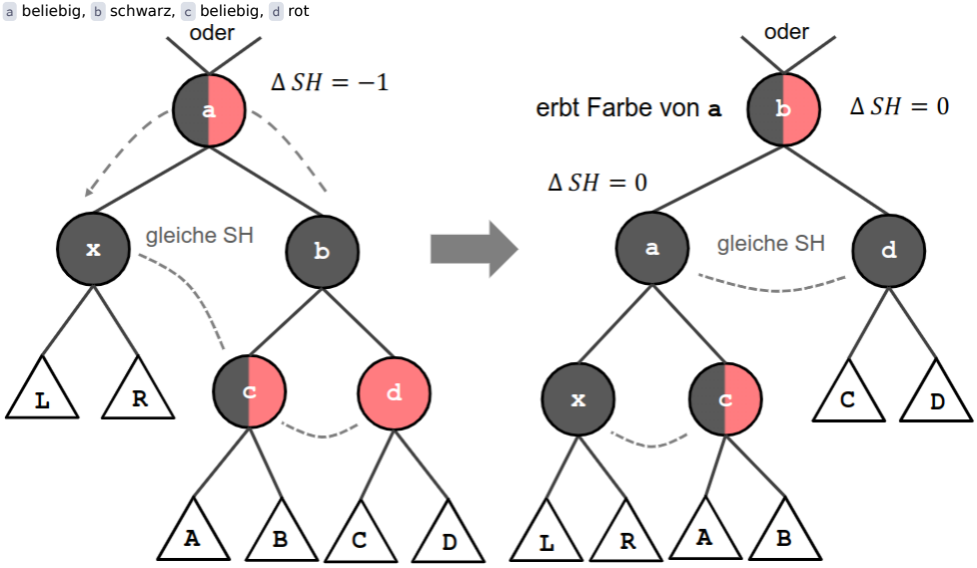
Wenn `a` schwarz ist, dann gilt für Elternknoten $\Delta SH = \pm 1$.
Verfahre also rekursiv mit `a` als neuem `x`

4. `a` beliebig, `b` schwarz, `c` rot, `d` nicht rot



Wird zu Fall 4

5. `a` beliebig, `b` schwarz, `c` beliebig, `d` rot



Ursprüngliche SH von vor der Entfernung wieder hergestellt

Algorithmus

```
transplant(T,u,v) // with sentinel, also works for v==nil
        IF u.parent==T.sent THEN
                T.root=v
        ELSE
                IF u==u.parent.left THEN
                        u.parent.left=v
                ELSE
                        u.parent.right=v;
        IF v != nil THEN
                v.parent=u.parent;

delete(T,z)
        a=z.parent; dsh=nil;
        IF z.left==z.right==nil THEN // z leaf
                IF z.color==black AND z!=T.root THEN
                        IF z.parent.left==z THEN dsh=right ELSE dsh=left;
                transplant(T,z,nil);
        ELSE IF z.left==nil THEN // z half leaf
                y=z.right;
                transplant(T,z,z.right);
                y.color=z.color;
        ELSE IF z.right==nil THEN // z half leaf
                y=z.left;
                transplant(T,z,z.left);
                y.color=z.color;
        ELSE // z has two children
                y=z.right; a=y; wentleft=false;
                WHILE y.left != nil DO
                        a=y; y=y.left; wentleft=true;
                IF y.parent != z THEN
                        transplant(T,y,y.right);
                        y.right=z.right;
                        y.right.parent=y;
                transplant(T,z,y);
                y.left=z.left;
                y.left.parent=y;
```

```
                IF y.color==black THEN
                        IF wentleft THEN dsh=right ELSE dsh=left;
                y.color=z.color;
        IF dsh!=nil THEN fixColorsAfterDeletion(T,a,dsh);
```

- `a` ist ein Zeiger auf den Knoten, in dem die tiefste Imbalance entstehen könnte
- `dsh` = ΔSH für Knoten `a`: `nil` = 0, `left` = 1, `right` = -1
- In den Fällen, in denen `z` ein Halbblatt ist, muss `y.color==red` sein, da sonst die SH-Regel verletzt wäre.
  Dann kann man einfach `y` umhängen und die Farbe von `z` kopieren
- In den Fällen, in denen `z` kein (Halb-)Blatt ist, muss eine Fallunterscheidung stattfinden, je nachdem, ob `y` rechtes oder linkes Kind ist, davon ist dann auch die eventuelle Imbalance abhängig

```
fixColorsAfterDeletion(T,a,dsh)
        IF dsh==right THEN // extra black node on the right
                x=a.left; b=a.right; c=b.left; d=b.right;
                IF x!=nil AND x.color==red THEN // x is red, easy to solve
                        x.color=black;
                ELSE IF a.color==black AND b.color==red THEN // case 1
                        rotateLeft(T,a);
                        a.color=red; b.color=black;
                        fixColorsAfterDeletion(T,a,dsh);
                ELSE IF a.color==red AND b.color==black // case 2a
                                AND (c==nil OR c.color=black)
                                AND (d==nil OR d.color=black) THEN
                        a.color=black; b.color=red;
                ELSE IF a.color==black AND b.color==black // case 2b
                                AND (c==nil OR c.color==black)
                                AND (d==nil OR d.color==black) THEN
                        b.color=red;
                        IF a==a.parent.left THEN dsh=left
                        ELSE IF a==a.parent.right THEN dsh=right ELSE dsh=nil;
                        fixColorsAfterDeletion(T,a.parent,dsh);
                ELSE IF b.color==black AND c!=nil AND c.color==red // case 3
                                AND (d==nil OR d.color==black) THEN
                        rotateRight(T,b);
                        c.color=black; b.color=black;
                        fixColorsAfterDeletion(T,a,dsh);
                ELSE IF b.color==black AND d!=nil AND d.color==red THEN // case 4
                        rotateLeft(T,a);
                        b.color=a.color; a.color=black; d.color=black;
        ELSE // dsh==left, extra black node on the left
                // do the same, but exchange left and right
```

- `dsh=right` impliziert `b!=nil`
- Der letzte `ELSE`-branch ist für den linkslastigen Fall
- Außer in Fall 2b) führen rekursive Aufrufe im nächsten Schritt zum Rekursionsende

Laufzeiten

`y` suchen hat wie beim BST Laufzeit $O(h) = O(\log n)$
Falls Rekursion in Fixup eintritt, ist die Laufzeit konstant
$\rightsquigarrow$ Gesamtlaufzeit Löschen = $O(h) = O(\log n)$

Worst-Case-Laufzeiten RSB

- Einfügen: $\Theta(\log n)$
- Löschen: $\Theta(\log n)$
- Suchen: $\Theta(\log n)$

AVL-Bäume

Optimierte Konstanten:

- RS-Bäume: $h \leq 2 \cdot \log n$
- AVL-Bäume: $h \leq 1.441 \cdot \log n$
  Balance in Knoten x mit angehängtem rechtem und linkem Teilbaum: $B(x) = height(\text{rechter Teilbaum}) - height(\text{linker Teilbaum})$
  Konvention: $height(\text{leerer Baum}) = -1$
  Ein AVL-Baum ist ein binärer Suchbaum, sodass für die Balance $B(x)$ in jedem Knoten x gilt: $B(x) \in \{-1, 0, +1\}$

Höhe

Ein AVL-Baum mit n Knoten hat die maximale Höhe $h \leq 1.441 \cdot \log_2 n$

AVL-Baum vs. RS-Baum

- AVL-Baum:
  Differenz von rechtem und linkem Teilbaum desselben Knotens ≤ 1
  Einfügen und Löschen verletzen in der Regel öfter die Baum-Bedingung, mehr Aufwand zum Rebalancieren
- RS-Baum:
  Höhenfaktor von rechtem und linkem Teilbaum desselben Knotens ≤ 2
  Suchen dauert eventuell länger
  AVL-Bäume geeigneter, wenn mehr Such-Operationen und weniger Einfüge- und Lösch-Operationen

AVL $\subset$ RS, AVL $\neq$ RS

Jeder nicht-leere AVL-Baum der Höhe h lässt sich als RS-Baum mit Schwarzhöhe $\lceil \frac{h+1}{2} \rceil$ darstellen.
Für gerade h gibt es sogar einen Baum mit roter Wurzel, Schwarzhöhe $\frac{h}{2}$, der alle anderen RS-Baumbedingungen erfüllt.
Für jede Höhe $h \geq 3$ gibt es einen RS-Baum, der kein AVL-Baum ist.

Einfügen

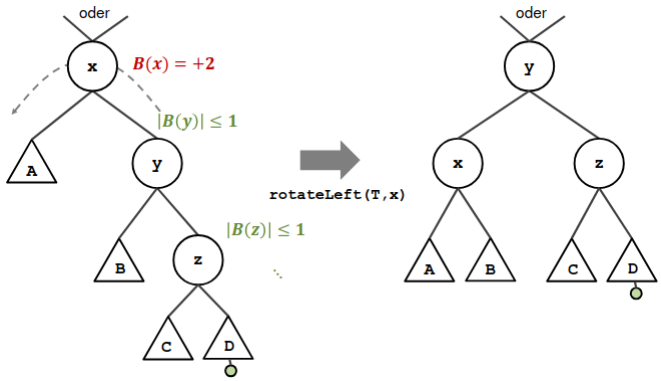
Funktioniert wie beim BST mit Sentinel, zuzüglich eventuellem Rebalancieren

```
insert(T,z) // z.left==z.right==nil;
        x=T.root; px=T.sent;
        WHILE x != nil DO
                px=x;
                IF x.key > z.key THEN
                        x=x.left
                ELSE
                        x=x.right;
        z.parent=px;
        IF px==T.sent THEN
                T.root=z
        ELSE
                IF px.key > z.key THEN
                        px.left=z
                ELSE
                        px.right=z;
        fixBalanceAfterInsertion(T,z);
```

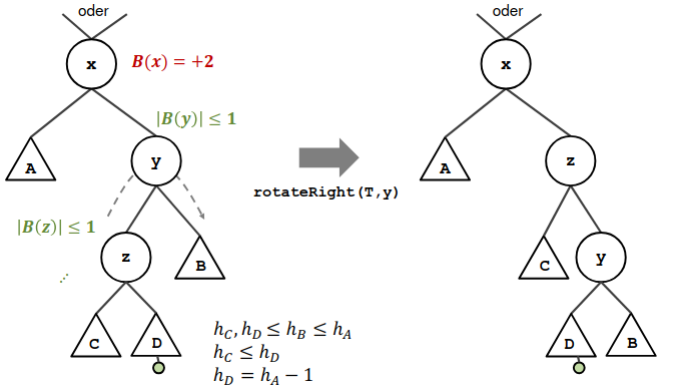
Rebalacieren

- Da immer in Blättern eingefügt wird, werden nur die Knoten direkt darüber eventuell nicht mehr balanciert
- Das Einfügen muss nicht immer dazu führen, dass die Balance nicht mehr vorhanden ist
- Unterscheide 4 Fälle:

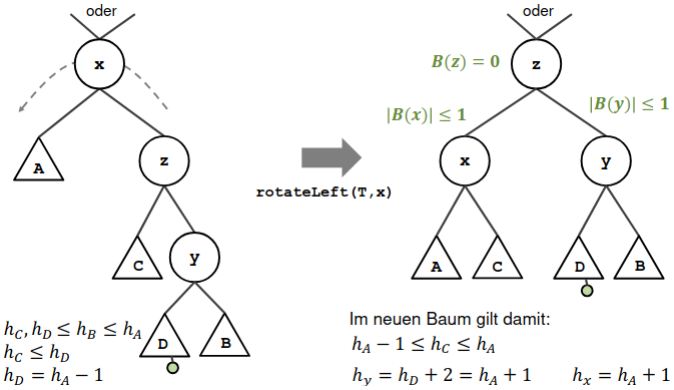
1. Einfache Rotation:



   Nach dem Einfügen und Rotieren ist die Höhe gleich der Höhe vor dem Einfügen
2. Hier finden zwei Rotationen statt:



$h_C, h_D \leq h_B \leq h_A$
$h_C \leq h_D$
$h_D = h_A - 1$

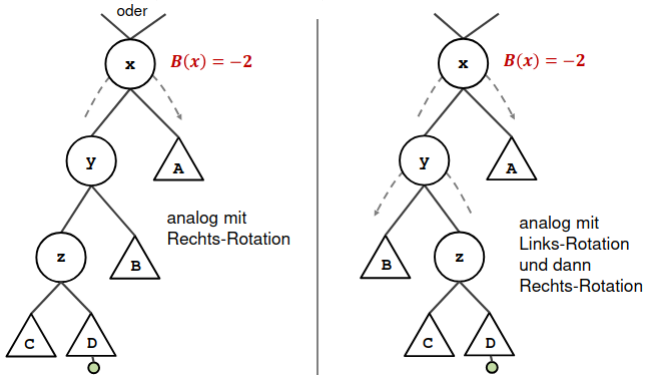


$h_C, h_D \leq h_B \leq h_A$
$h_C \leq h_D$
$h_D = h_A - 1$

Im neuen Baum gilt damit:
$h_A - 1 \leq h_C \leq h_A$
$h_y = h_D + 2 = h_A + 1 \qquad h_x = h_A + 1$

Höhe ist auch hier unverändert

3. Fall 3 und 4 sind spiegelverkehrt und analog zu den Fällen 1 und 2



Laufzeit

Gesamtlaufzeit $O(h) = O(\log n)$
Suche hat Laufzeit $O(h)$, Rebalancieren ist nur einmal nötig, also ist das konstant

Löschen

Analog zum BST, aber Rebalacierung eventuell bis in die Wurzel nötig
Gesamtlaufzeit $O(h) = O(\log n)$

Worst-Case-Laufzeiten

- Einfügen: $\Theta(\log n)$
- Löschen: $\Theta(\log n)$
- Suchen: $\Theta(\log n)$
  AVL-Bäume haben bessere theoretische Konstanten als Rot-Schwarz-Bäume, sind je nach Daten und Operationen aber in der Praxis nur unwesentlich schneller.

Splay-Bäume

Selbst-organisierende Datenstrukturen

Selbst-Organisierende Listen

Ansatz: einmal angefragte Werte werden voraussichtlich noch öfter angefragt
Variante für Bäume: Splay trees

Anwendung: SQUID

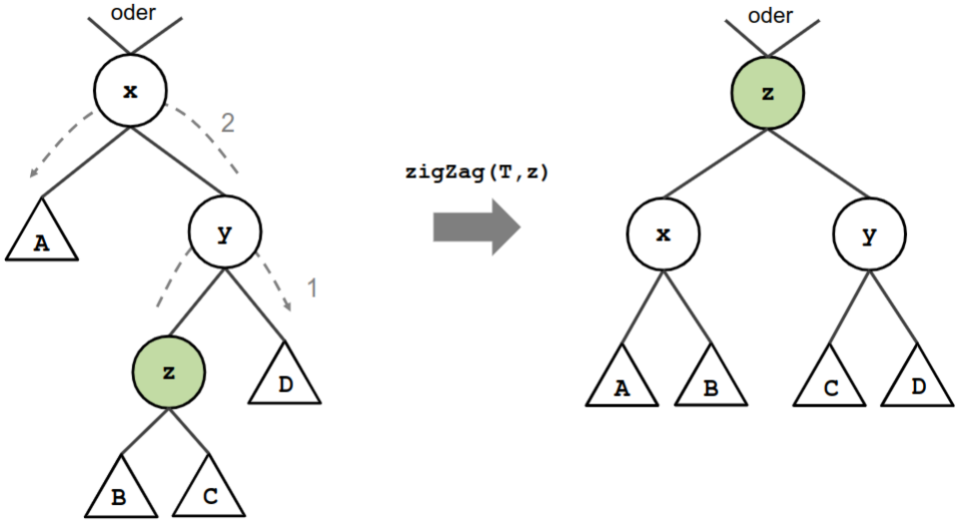
- Web-Cache-Proxy
- Speichert Access Control Listen (ACL) für http-Zugriffe als Splay-Tree

Splay-Operationen

- Splay-Bäume bilden Untermenge der BST
- Spüle gesuchten oder neu eingefügten Knoten an die Wurzel
- `splay(T,z)` = Folge von Zig-, Zig-Zig und Zig-Zag-Operationen

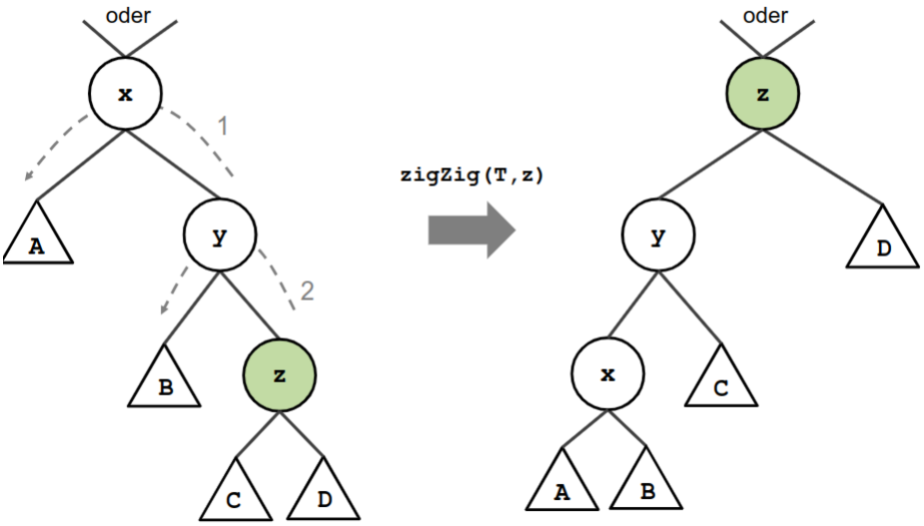
Zig-Zag-Operation

Rechts-Links- oder Links-Rechts-Rotation



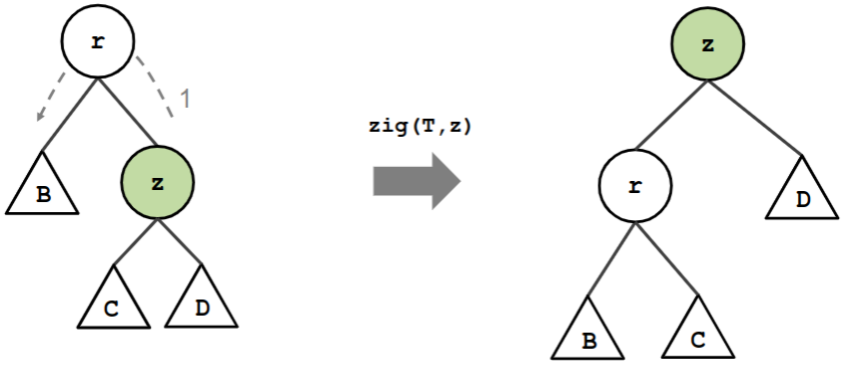
Zig-Zig-Operation

Links-Links- oder Rechts-Rechts-Rotation



Zig-Operation

Einfache Links- oder Rechts-Rotation
Wird verwendet, falls `z` direkt unter Wurzel hängt



Splay-Operation

```
splay(T,z)
        WHILE z != T.root DO
                IF z.parent.parent==nil THEN
                        zig(T,z);
                ELSE
                        IF z==z.parent.parent.left.left OR z==z.parent.parent.right.right THEN
                                zigZig(T,z);
                        ELSE
                                zigZag(T,z);

zigZig(T,z)
        IF z==z.parent.left THEN
                rotateRight(T,z.parent.parent);
                rotateRight(T,z.parent);
        ELSE
                rotateLeft(T,z.parent.parent);
                rotateLeft(T,z.parent);

zigZag(T,z) // disclaimer: Not official, I wrote this!
        IF z==z.parent.left THEN
                rotateRight(T,z.parent);
                rotateLeft(T,z.parent);
        ELSE
                rotateLeft(T,z.parent);
                rotateRight(T,z.parent);

zig(T,z) // disclaimer: Not official, I wrote this!
        IF z==z.parent.left THEN
                rotateRight(T,z.parent);
        ELSE
                rotateLeft(T,z.parent);
```

Gesamtlaufzeit $O(h)$

Suchen

```
search(T,k)
        x=T.root;
        WHILE x != nil AND x.key != k DO
```

```
                IF x.key < k THEN
                        x=x.right
                ELSE
                        x=x.left;
// now search is done
        IF x==nil THEN
                return nil
        ELSE
                splay(T,x);
                return T.root;
```

Suche und Splayen haben Laufzeit $O(h)$ $\rightsquigarrow$ Gesamtlaufzeit $O(h)$
Alternative: Bei erfolgloser Suche letzten besuchten Knoten nach oben splayen

Einfügen

1. Suche analog zum Einfügen bei BST Einfügepunkt
2. Spüle eingefügten Knoten `x` per Splay-Operation nach oben

Laufzeit

1. Position im BST suchen: $O(h)$
2. `splay(T,x)`: $O(h)$
   $\rightsquigarrow$ Gesamtlaufzeit $O(h)$

Löschen

1. Spüle gesuchten Knoten `x` per Splay-Operation nach oben
2. Lösche `x`
   Wenn einer der beiden Teilbäume leer ist, fertig
3. Spüle den "größten" Knoten `y` im linken Teilbaum per Splay-Operation nach oben
   `y` kann keinen rechtes Kind haben, da größter Wert im linken Teilbaum
4. Hänge rechten Teilbaum an `y` an

Laufzeit

1. `splay(T,x)`: $O(h)$
2. `x` löschen: $O(1)$
3. `y` im linken Teilbaum `L` finden, `splay(L,y)`: $O(h) + O(h) = O(h)$
4. Anhängen: $O(1)$
   $\rightsquigarrow$ Gesamtlaufzeit $O(h)$

Laufzeit Splay-Bäume

- Amortisierte Laufzeit:
  Laufzeit pro Operation über mehrere Operationen hinweg
- Für $m \geq n$ Operationen auf einem Splay-Baum mit maximal n Knoten ist die Worst-Case-Laufzeit $O(m \cdot \log_2 n)$, also $O(\log_2 n)$ pro Operation.
- Zusätzlich: Oft gesuchte Elemente werden sehr schnell gefunden

(Binäre Max-)Heaps

Ein binärer Max-Heap ist ein binärer Baum, der

1. bis auf das unterste Level vollständig und im untersten Level von links gefüllt ist
2. Für alle Knoten $x \neq T.root$ gilt: $x.parent.key \geq x.key$

- Heaps sind keine BSTs, linke Kinder können größere Werte als rechte Kinder haben!
- Bei Min-Heaps sind die Werte in Elternknoten jeweils kleiner

Eigenschaften

- Da Baum (fast) vollständig ist, gilt $h \leq \log n$
- Maximum des Heaps steht in der Wurzel

Heaps durch Arrays

- speichere Anzahl Knoten in `H.length` (leerer Heap `H.length==0`)
- Duale Sichtweise als Pointer oder als Array (j ist Index im Array):
  - $j.parent = \left\lceil \frac{j}{2} \right\rceil - 1$
  - $j.left = 2(j+1) - 1$
  - $j.right = 2(j+1)$

Einfügen

- Position durch Baumstruktur vorgegeben
- Vertausche nach oben, bis Max-Eigenschaft wieder erfüllt

```
insert(H,k) // as (unlimited) array
        H.length=H.length+1;
        H.A[H.length-1]=k;
        i=H.length-1;
        WHILE i>0 AND H.A[i] > H.A[i.parent]
                SWAP(H.A,i,i.parent);
                i=i.parent;
```

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

Lösche Maximum

1. Ersetze Maximum durch "letztes" Blatt
2. Stelle Max-Eigenschaften wieder her, indem Knoten nach unten gegen das Maximum der beiden Kinder getauscht wird (`heapify`)

```
extract-max(H) // as (unlimited) array
        IF isEmpty(H) THEN
                return error ‘underflow‘
        ELSE
                max=H.A[0];
                H.A[0]=H.A[H.length-1];
                H.length=H.length-1;
                heapify(H,0);
                return max;

heapify(H,i) // as (unlimited) array
        maxind=i;
        IF i.left<H.length AND H.A[i]<H.A[i.left] THEN
                maxind=i.left;
        IF i.right<H.length AND H.A[maxind]<H.A[i.right] THEN
                maxind=i.right;
        IF maxind != i THEN
                SWAP(H.A,i,maxind);
                heapify(H,maxind);
```

Laufzeit beider Algorithmen $O(h) = O(\log n)$

Heap-Konstruktion aus Array

Blätterindizes: $\lceil \frac{n-1}{2} \rceil, \ldots, n-1$
Blätter sind für sich triviale Max-Heaps
Baue rekursiv per `heapify` Max-Heaps für Teilbäume

```
buildHeap(H) // array A has already been copied to H.A
        H.length=A.length;
        FOR i = ceil((H.length-1)/2)-1 DOWNTO 0 DO
                heapify(H,i);
```

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

Heap-Sort

Gibt Einträge in Array `A` in absteigender Größe aus

```
heapSort(H) // array A has already been copied to H.A
        buildHeap(H);
        WHILE !isEmpty(H) DO PRINT extract-max(H);
```

Laufzeit $O(n \cdot h) = O(n \log n)$
Alternativ: speichere in jeder `WHILE`-Iteration `max=extract-max(H)` in `H.A[H.length]=max`, um sortierte Liste am Ende aufsteigend im Array `A` zu haben.

Abstrakter Datentyp Priority Queue

- `new(Q)`: erzeugt neue, leere Priority Queue namens `Q`
- `isEmpty(Q)`: gibt an, ob Queue `Q` leer
- `max(Q)`: gibt "größtes" Element aus Queue `Q` zurück, Fehler wenn leer
- `extract-max(Q)`: gibt "größtes" Element aus `Q` zurück, löscht es aus `Q`, Fehler wenn leer
- `insert(Q,k)`: fügt Wert `k` zu Queue `Q` hinzu
  Implementation kann Priority Heap verwenden (Java)

B-Bäume

Ein B-Baum von Grad t ist ein Baum, bei dem

1. jeder Knoten außer der Wurzel zwischen $t-1$ und $2t-1$ Werte `key[0], key[1], ...` hat, die Wurzel hat zwischen 1 und $2t-1$ Werte
2. die Werte innerhalb eines Knoten aufsteigend geordnet sind
3. die Blätter alle die gleiche Höhe haben
4. jeder innerer Knoten mit n Werten $n+1$ Kinder hat, sodass für alle Werte k_j aus dem j-ten Kind gilt: $k_0 \leq key[0] \leq k_1 \leq key[1] \leq \cdots \leq k_{n-1} \leq key[n-1] \leq k_n$

Darstellung

- `x.n`: Anzahl Werte des Knotens `x`
- `x.key[0], ..., x.key[x.n-1]`: Geordnete Werte in Knoten `x`
- `x.child[0], ..., x.child[x.n]`: Zeiger auf Kinder in Knoten `x`

Höhe

- Mindestens 1 Wert in Wurzel
- Mindestens 2 Knoten in Tiefe 1 mit jeweils mindestens t Kindern
- Mindestens $2t$ Knoten in nächster Tiefe mit jeweils mindestens t Kindern
- Mindestens $2t^2$ Knoten in nächster Tiefe mit jeweils mindestens t Kindern, usw.
- In jedem Knoten außer Wurzel mindestens $t-1$ Werte
- Anzahl Werte n im B-Baum im Vergleich zur Höhe h: $n \geq 2t^h - 1$, also $log_t \frac{n+1}{2} \geq h$
- $\rightsquigarrow$ Ein B-Baum vom Grad t mit n Werten hat maximale Höhe $h \leq \log_t \frac{n+1}{2}$
- Für größere t also flacher als vollständiger Binärbaum

Anwendung

- MySQL speichert Werte in B-Bäumen
- Lesen/Schreiben in Blöcken: mehrere Werte (z.B. Index-Einträge) auf einmal

Suche

```
search(x,k)
        WHILE x != nil DO
                i=0;
```

```
                WHILE i < x.n AND x.key[i] < k DO i=i+1;
                IF i < x.n AND x.key[i]==k THEN
                        return (x,i);
                ELSE
                        x=x.child[i];
        return nil;
```

\2. While-Schleife: Maximal $2t \in O(1)$ Iterationen
Laufzeit $O(t \cdot h) = O(\log_t n)$

Baumkunde

- B-Baum vom Grad t: max. $2t$, min. t Kinder pro Knoten $\neq$ Wurzel
  Alternative Definition: max. t, min. $\frac{t}{2}$ Kinder pro Knoten $\neq$ Wurzel
- 2-3-4-Baum/(2,4)-Baum: B-Baum mit $t = 2$
- B+-Baum: alle Werte in Blättern, innerer Knoten enthalten Werte erneut
  Vorteil: innere Knoten speichern nur kurzen Schlüssel, nicht auch noch Daten(-zeiger)
  Nachteil: Findet Werte erst im Blatt
  Alternativer Name: B*-Baum
- Alternative Bedeutung B*-Baum: B-Baum mit Füllgrad min. $\frac{2}{3}$ pro Knoten $\neq$ Wurzel

Einfügen

Idee

- Einfügen erfolgt immer in einem Blatt
- Wenn Blatt weniger als $2t - 1$ Werte hat, dann einfügen und fertig
- Wenn nicht:

Splitten

- Wenn Blatt bereits $2t - 1$ Werte, dann teile es in zwei Blätter mit je $t - 1$ Werten, füge mittleren Wert im Elternknoten ein
- Wenn dadurch Elternknoten mehr als $2t - 1$ Werte hat, rekursiv nach oben
- Splitten an der Wurzel: Neue Wurzel wird erzeugt, Höhe des Baumes wächst um 1
  B-Baum-Einfügen splittet beim Suchen und läuft nur einmal hinab, sonst werden teure Disk-Operationen zweimal ausgeführt, einmal beim ab-, einmal beim aufsteigen.

Informeller Algorithmus

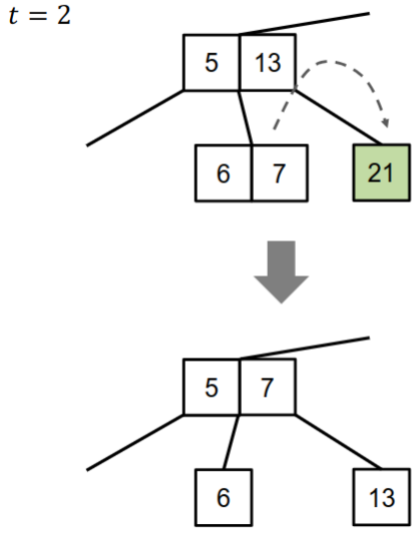
```
insert(T,z)
        Wenn Wurzel schon 2t-1 Werte, dann splitte Wurzel
        Suche rekursiv Einfügeposition:
                Wenn zu besuchendes Kind 2t-1 Werte, splitte es erst
        Füge z in Blatt ein
```

Laufzeit $O(t \cdot h) = O(\log_t n)$
Schleifeninvariante:
Bei der Suche hat der aktuelle Knoten immer weniger als $2t - 1$ Werte, da sonst vorher gesplitted.
Eventuelles Splitten ist also problemlos möglich.
Auch das Blatt hat am Ende weniger als $2t - 1$ Werte.

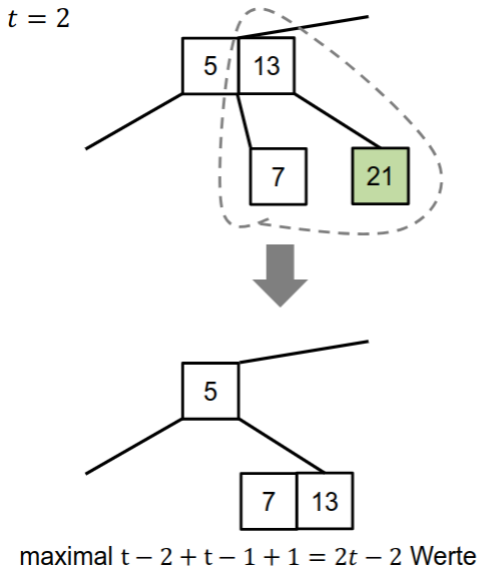
Löschen

Löschen im Blatt

- Wenn Blatt noch mehr als $t - 1$ Werte hat, dann einfach entfernen
- Wenn $t - 1$ Werte im Blatt mit zu löschendem Wert sind, linker oder rechter Geschwisterknoten hat mind. t Werte, dann rotiere Werte von Geschwisterknoten und Elternknoten:

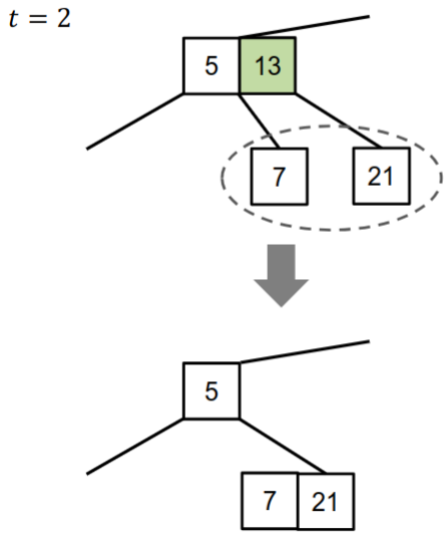


- Wenn $t - 1$ Werte im Blatt mit zu löschendem Wert sind, linker oder rechter Geschwisterknoten haben auch $t - 1$ Werte, dann verschmelze einen Geschwisterknoten mit Wert aus Elternknoten, dieser hat nun eventuell zu wenig Werte:



Löschen im inneren Knoten

- Verschieben:
  Wenn sich mehr als $t-1$ Werte in einem der beiden Kindknoten befinden, dann größten Wert (vom linken Kind) bzw. kleinsten Wert (vom rechten Kind) nach oben kopieren
- Verschmelzen:
  Wenn sich jeweils $t-1$ Werte in beiden Kindknoten befinden, dann Kindknoten verschmelzen, eventuell hat Elternknoten nun zu wenig Werte:



B-Baum-Löschen läuft auch nur einmal hinab, stelle dazu sicher, dass zu besuchendes Kind mindestens t Werte hat.

Allgemeines Verschmelzen ohne Löschen

Zu besuchendes Kind, rechter, linker Geschwisterknoten (sofern existent) haben nur $t-1$ Werte.
Dann ist Verschmelzen ohne weitere Änderungen möglich, wenn der Elternknoten vorher mindestens t Werte hat.

Allgemeines Rotieren/Verschieben ohne Löschen

Zu besuchendes Kind hat nur $t-1$ Werte, aber ein Geschwisterknoten hat mehr als $t-1$ Werte, dann kann man dies ohne Änderungen oberhalb tun

Informeller Algorithmus

```
delete(T,k)
        Wenn Wurzel nur 1 Wert und beide Kinder t-1 Werte haben, verschmelze Wurzel und Kinder (reduziert Höhe um 1)
        Suche rekursiv Löschposition:
                Wenn zu besuchendes Kind nur t-1 Werte hat, verschmelze es oder rotiere/verschiebe
        Entferne Wert k in inneren Knoten/Blatt
```

Laufzeit $O(t \cdot h) = O(\log_t n)$
Schleifeninvariante:
Aktueller Knoten hat zu diesem Zeitpunkt mindestens t Werte, sonst wäre er vorher verschmolzen worden oder es wäre rotiert worden.
Beim Verschmelzen/Verschieben des Kindes kann die Anzahl der Werte im aktuellen Knoten nicht unter $t-1$ fallen.
Entfernen aus Blatt problemlos möglich, da mindestens t Werte vorhanden.
Entfernen im inneren Knoten durch Verschieben oder Verschmelzen.

Worst-Case-Laufzeiten

- Einfügen: $\Theta(\log_t n)$
- Löschen: $\Theta(\log_t n)$
- Suchen: $\Theta(\log_t n)$
- O-Notation versteckt konstanten Faktor t für Suche innerhalb eine Knoten:
  $t \cdot \log_t n = t \cdot \frac{\log_2 n}{\log_2 t}$ ist in der Regel größer als $\log_2 n$, also nur vorteilhaft, wenn Daten blockweise eingelesen werden.

5. Randomized Data Structures

- Deterministische Datenstrukturen:
  Bisher war alles deterministisch, Verhalten für identische Eingaben immer gleich
- Randomisierte Datenstrukturen:
  Nun hängt Verhalten auch von zufälligen Entscheidungen der Datenstruktur ab

Skip Lists

Idee

- Zweidimensionale Datenstruktur
- Füge rekursiv Express-Listen ein
- Diese haben weniger Elemente als die ursprüngliche Liste

Suche mittels Express-Listen

Beginne in Express-Liste:

- Wenn Element gefunden, ausgeben
- Wenn nächstes Element kleiner-gleich gesuchtem Element, weiter nach rechts
- Wenn nächstes Element in Express-Liste größer als gesuchtes Element, nach unten

Implementierung

- `L.head`: erstes/oberstes Element der Liste
- `L.height`: Höhe der Skiplist
- `x.key`: Wert
- `x.next`: Nachfolger
- `x.prev`: Vorgänger
- `x.down`: Nachfolger Liste unten
- `x.up`: Nachfolger Liste oben
- `nil`: kein Nachfolger / leeres Element

Suchalgorithmus

```
search(L,k)
        current=L.head;
        WHILE current != nil DO
                IF current.key == k THEN return current;
                IF current.next != nil AND current.next.key <= k
                THEN current=current.next
                ELSE current=current.down;
        return nil;
```

Laufzeit hängt von Expresslisten ab

Auswahl der Elemente für Express-Listen

Wähle jedes Element aus Liste mit Wahrscheinlichkeit p (z.B. $p = \frac{1}{2}$) für übergeordnete Liste
Durchschnittliche Höhe $h \in O(\log_{\frac{1}{p}} n)$

Durchschnittliche Laufzeit für Suchen

Im schlimmsten Fall wird Suche erst in unterster Liste beendet
Wenn Skip-Liste Höhe h, braucht man im Durchschnitt $\frac{h}{p} \in O(h) = O(\log n)$ viele Schritte
$\rightsquigarrow$ Durchschnittliche Laufzeit = $O(h)$

Einfügen

- Füge auf unterster Ebene ein und dann eventuell auf Ebenen darüber
- Zufällige Wahl mit Wahrscheinlichkeit p auf jeder Ebene
- Durchschnittliche Laufzeit = $O(h)$

Löschen

- Entferne Vorkommen des Elements auf allen Ebenen
- Durchschnittliche Laufzeit = $O(h)$

Laufzeiten und Speicherbedarf

- Einfügen $\Theta(\log_{\frac{1}{p}} n)$
- Löschen $\Theta(\log_{\frac{1}{p}} n)$
- Suchen $\Theta(\log_{\frac{1}{p}} n)$
- O-Notation versteckt Faktor $\frac{1}{p}$
- Speicherbedarf im Durchschnitt: $\frac{n}{1-p}$

Anwendung

Einfügen/Löschen unterstützen parallele Verarbeitung (z.B. Multi-Core-Systeme), da nur sehr lokale Änderungen
Bäume mit Re-Balacierung können dies nicht
Dafür logarithmische Laufzeit nur im Durchschnitt, also nicht garantiert

Hash Tables

Idee

$h : \text{Datenmenge} \rightarrow [0, T.length - 1]$ ist uniform und unabhängig verteilt
Datum x wird auf Arrayeintrag $h(x)$ in Hashtabelle/Array `T[]` abgebildet und dann dort gespeichert
Suche: ist `x` in `T[h(x)]` vorhanden?
Löschen: Lösche `x` aus `T[h(x)]`
Einfügen, Suchen, Löschen mit konstant vielen Array-Operationen

Kollisionsauflösung

- Wenn Array-Eintrag schon belegt, bilde verkettete Liste und füge neues Element vorne ein
- Es gibt weitere Arten der Kollisionsauflösung

Hash Tables mit verketteten Listen

- Einfügen immer noch konstante Anzahl Array-/Listen-Operationen
- Suchen/Löschen benötigen so viele Schritte, wie jeweilige Liste lang ist
- Wenn Hashfunktion uniform verteilt, dann hat jede Liste im Erwartungswert $\frac{n}{T.length}$ viele Einträge

Laufzeit

Bei uniform und unabhängig verteilten Hashwerten benötigen Suchen und Löschen im Durchschnitt $\Theta(\frac{n}{T.length})$ viele Schritte.
Einfügen benötigt im Worst-Case $\Theta(1)$ viele Schritte.
Wählt man $T.length \approx n$, ergibt sich im Durchschnitt konstante Laufzeit.

Gute Hash-Funktionen

"Universelle" Hash-Funktion

- Interpretiere (Binär-)Daten als Zahlen zwischen 0 und $p-1$, p ist prim, $p >> T.length$
- Wähle zufällige $a, b \in [0, p-1], a \neq 0$, setze $h_{a,b}(x) := ((a \cdot x + b) \mod p) \mod T.length$
- Verteilung und Unabhängigkeit/Kollisionsresistenz gewährleistet
  Kryptographische Hash-Funktionen wie MD5, SHA1: $\{0,1\}^* \to \{0,1\}^{160}$
  MD5, SHA1 nicht sicher, besser SHA2/SHA3 verwenden
  Setze $h(x) = MD5(x) \mod T.length$

Anwendungen

z.B. MySQL

Hash Tables vs. Bäume

- Hash Table:
  - Nur Suche nach bestimmten Wert möglich
  - In der Regel Hashtable größer als zu erwartende Anzahl Einträge
- Baum:
  - Schnelles Traversieren möglich (z.B. nächstkleinerer Wert), auch Bereichssuche

Laufzeiten, Speicherbedarf

- Einfügen: $\Theta(1)$ im Worst-Case
- Löschen: $\Theta(1)$ im Durchschnitt
- Suchen: $\Theta(1)$ im Durchschnitt
- Speicherbedarf in der Regel größer als n, üblicherweise ca. $1,33 \cdot n$

Bloom-Filter

Speicherschonende Wörterbücher mit kleinem Fehler

Beispiel: Schlechte Passwörter vermeiden

1. Speichere schlechte Passwörter in Bloom-Filter
2. Prüfe, ob eingegebenes Passwort im Bloom-Filter ist
   Starke Passwörter, die fälschlicherweise dem Wörterbuch zugeordnet werden, sind ärgerlich, aber nicht sehr schlimm

Anwendungen

- NoSQL-Datenbanken: Abfragen für nicht-vorhandene Elemente verhindern
- Bitcoin: Prüfen von Transaktionen ohne gesamte Daten zu laden
- Früher auch Chrome-Browser: Erkennen schädlicher Webseiten

Erstellen

Gegeben:

- n Elemente $x_0, x_1, \ldots, x_{n-1}$ beliebiger Komplexität
- m Bits speicher, üblicherweise in einem Bit-Array
- k "gute" Hash-Funktionen $H_0, \ldots, H_k - 1$ mit Bildbereich $0, 1, \ldots, m-1$
- Empfohlene Wahl: $k = \frac{m}{n} \cdot \ln 2$ ergibt Fehlerrate von ca. 2^{-k}, üblicherweise $k \in [5, 20]$

```
initBloom(X,BF,H) // H array of functions H[j], BF bit-array, X array of objects
        FOR i=0 TO BF.length-1 DO BF[i]=0; // initialise array with 0-entries
        FOR i=0 TO X.length-1 DO
                FOR j=0 TO H.length-1 DO
                        BF[H[j](X[i])]=1;
```

- Schreibe für jedes Element in jede Bit-Position $H_0(x_i), \ldots, H_{k-1}(x_i)$ eine 1
- Eventuell werden dabei Einträge mehrmals auf 1 gesetzt

Suchen

```
searchBloom(BF,H,y) // H array of functions H[j]
        result=1;
        FOR j=0 TO H.length-1 DO
                result=result AND BF[H[j](y)];
        return result;
```

- Gibt an, dass `y` im Wörterbuch ist, gdw. alle k Einträge für `y` in `BF==1` sind
- Wenn `y` nicht im Wörterbuch, kann Algorithmus eventuell trotzdem 1 zurückgeben
- Daher "gute" Hash-Funktionen und Größe des Filters nicht zu klein wählen
- Keine false negatives, nur false positives
- Wenn BF nur bis zur Hälfte mit 1en gefüllt und Hash-Funktionen uniforme und unabhängige Werte liefern, dann Fehler $\leq 2^{-k}$

Beispielrechnung

$n = 100.000$ Passwörter, je 10 ASCII-Zeichen

- Baumstruktur:
  - Speicherbedarf: 8.000.000 Bits + Baumstruktur
  - Suchen: ca. $\log_2 100.000 \approx 17$ Elemente betrachten
- Bloom-Filter mit $k = 7, m = k \cdot n \cdot \ln 2$
  - Speicherbedarf: ca. 1.000.000 Bits
  - Suchen: $k = 7$ Mal hashen und $k = 7$ Array-Zugriffe

6. Graph Algorithms

Graphen

(Endliche) gerichtete Graphen

Besteht aus:

1. (Endliche) Kontenmenge V
2. (Endliche) Kantenmenge $E \subseteq V \times V$; $(u, v) \in E$: Kante von Knoten u zu v

- Schleifen, Zyklen, isolierte Knoten möglich
- Keine Mehrfachkanten zwischen Knoten, Anordnung der Knoten irrelevant

Ungerichtete Graphen

Endlicher Graph, aber $(u, v) \in E \Leftrightarrow (v, u) \in E$
Alternative Darstellung: $\{u, v\}$ anstelle von $(u, v), (v, u)$

Pfadfinder

Konten v ist von Knoten u im Graphen $G = (V, E)$ erreichbar, wenn es Pfad $(w_1, \ldots, w_k) \in V^k$ gibt, sodass $(w_i, w_{i+1}) \in E$ für $i = 1, 2, \ldots, k-1, w_1 = u, w_k = v$
u ist immer von u mit leerem Pfad $k = 1$ erreichbar (Schleife)
Länge des Pfades = $k - 1$ = Anzahl Kanten
$(w_1, \ldots, w_k)$ ist kürzester Pfad von u nach v, wenn es keinen kürzeren Pfad gibt.
$shortest(u, v) :=$ Länge eines kürzesten Pfades von u nach v
Kürzester Pfad muss nicht eindeutig sein

Zusammenhänge

Ungerichteter Graph ist zusammenhängend, wenn jeder Knoten von jedem anderen Knoten aus erreichbar ist.
Gerichteter Graph ist stark zusammenhängend, wenn jeder Knoten von jedem anderen Knoten aus (gemäß Kantenrichtung) erreichbar ist.

Graphen und Bäume

Graphenbäume haben keine Ordnung auf Kindern
G ist ein Baum, wenn V leer ist oder es ein $r \in V$ (Wurzel) gibt, sodass jeder Knoten v von r per eindeutigem Pfad erreichbar ist.

Subgraphen

G' muss wieder ein Graph gleichen Typs (gerichtet/ungerichtet) sein
Graph $G' = (V', E')$ ist Subgraph/Teilgraph/Untergraph des Graphen $G = (V, E)$, wenn $V' \subseteq V$ und $E' \subseteq E$.

Darstellung von Graphen

Adjazenzmatrix

i Zeile, j Spalte

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

- Bei ungerichteten Graphen ist Matrix spiegelsymmetrisch zur Hauptdiagonalen
- Speicherbedarf = $\Theta(|V|^2)$
- Eintrag $a_{i,j}^{(m)}$ der m-ten Potenz A^m der Adjazenzmatrix A eines Graphen gibt die Anzahl der Wege an, die von Knoten i zu j entlang genau m Kanten führen

Adjazenzliste

Array mit verketteten Listen (sortiert/unsortiert)
Speicherbedarf = $\Theta(|V| + |E|)$

Gewichtete Graphen

Besitzt zusätzlich Funktion $w : E \to \mathbb{R}$
Ungerichtete, gewichtete Graphen: $w((u, v)) = w((v, u))$ f.a. $(u, v) \in E$
Zu Kanten (u, v) ist noch Wert $w((u, v))$ abzuspeichern

Breitensuche/Breadth-First Search (BFS)

Besuche zuerst unmittelbare Nachbarn, dann deren Nachbarn usw.
Anwendungen: Web Crawling, Broadcasting, Garbage Collection, ...

Algorithmus

```
BFS(G,s) //G=(V,E), s=source node in V
        FOREACH u in V-{s} DO
                u.color=WHITE; u.dist=+∞; u.pred=NIL;
        s.color=GRAY; s.dist=0; s.pred=NIL;
        newQueue(Q);
        enqueue(Q,s);
        WHILE !isEmpty(Q) DO
                u=dequeue(Q);
                FOREACH v in adj(G,u) DO
                        IF v.color==WHITE THEN
                                v.color=GRAY; v.dist=u.dist+1; v.pred=u;
```

```
                        enqueue(Q,v);
                u.color=BLACK;
```

- `dist`: Distanz von `s`
- `pred`: Vorgängerknoten
- `WHITE`: Knoten noch nicht besucht
- `GRAY`: in Queue für nächsten Schritt
- `BLACk`: fertig
- `adj(G,u)`: Liste aller Knoten $v \in V$ mit $(u,v) \in E$, Reihenfolge egal

Kürzeste Pfade ausgeben

```
PRINT-PATH(G,s,v)
//assumes that BFS(G,s) has already been executed
        IF v==s THEN
                PRINT s
        ELSE
                IF v.pred==NIL THEN
                        PRINT 'no path from s to v'
                ELSE
                        PRINT-PATH(G,s,v.pred);
                        PRINT v;
```

Laufzeit ohne BFS = $O(|V|)$

Abgeleiteter BFS-Baum

Definiere Subgraph $G^s_{pred} := V^s_{pred}, E^s_{pred}$ von G durch:

- $V^s_{pred} := \{v \in V : v.pred \neq NIL\} \cup \{s\}$
- $E^s_{pred} := \{(v.pred, v) : v \in (V^s_{pred} \backslash \{s\})\}$ und $(v, v.pred)$ für ungerichtete Graphen
  G^s_{pred} ist BFS-Baum zu G:
- Enthält alle von s aus erreichbaren Knoten in G.
- F.a. $v \in V^s_{pred}$ existiert genau ein Pfad von s in G^s_{pred}, der ein kürzester Pfad von s zu v in G ist.

Tiefensuche/Depth-First Search (DFS)

Zuerst alle noch nicht besuchten Nachfolgeknoten besuchen
(So weit wie möglich vom aktuellen Knoten weglaufen)

```
DFS(G) //G=(V,E)
        FOREACH u in V DO
                u.color=WHITE;
                u.pred=NIL;
        time=0;
        FOREACH u in V DO
                IF u.color==WHITE THEN
                        DFS-VISIT(G,u)

DFS-VISIT(G,u)
        time=time+1;
        u.disc=time;
        u.color=GRAY;
        FOREACH v in adj(G,u) DO
                IF v.color==WHITE THEN
                        v.pred=u;
                        DFS-VISIT(G,v);
        u.color=BLACK;
        time=time+1;
        u.finish=time;
```

- `time`: globale Variable
- `disc`: discovery time
- `finish`: finish time
  Laufzeit = $O(|V| + |E|)$
  Standardordnung der Knoten ist gemäß der Nummer

DFS-Wald = Menge von DFS-Bäumen

Subgraph $G_{pred} = (V, E_{pred})$ von G:
$E_{pred} := \{(v.pred, v) : v \in V, v.pred \neq NIL\}$ (ungerichtete Graphen: auch $(v, v.pred)$)
DFS-Baum gibt nicht unbedingt den kürzesten Weg wieder

Charakterisierung der Kanten in G

Zeichne in DFS-Baum G_{pred} auch restliche Kanten ein, dann ergeben sich folgende Typen:

1. Baumkante:
   alle Kanten in G_{pred}
2. Vorwärtskante:
   alle Kanten in G zu Nachkommen in G_{pred}, die keine Baumkante sind
3. Rückwärtskante:
   alle Kanten in G zu Vorfahren in G_{pred}, die keine Baumkante sind, alle Schleifen
4. Kreuzkante:
   Alle anderen Kanten in G

Kantenart erkennen

Man kann zu jedem Zeitpunkt während der DFS die soeben betrachte Kante (u, v) untersuchen. Sie ist:

1. Baumkante, wenn `v.color==WHITE`
2. Rückwärtskante, wenn `v.color==GRAY`

3. Vorwärtskante, wenn `v.color==BLACK` und `u.disc<v.disc`
4. Kreuzkante, wenn `v.color==BLACK` und `v.disc<u.disc`

Kantenarten in ungerichteten Graphen

In einem ungerichteten Graphen G entstehen durch DFS nur Baum- und Rückwärtskanten.

DFS Anwendungen

Job Scheduling

Graph:

- Knoten sind Jobs
- Kante: Job X muss vor Job Y beendet sein
  Problem: In welcher Reihenfolge sollen die Jobs bearbeitet werden?

Anwendungen

- Spreadsheets: Formeln aktualisieren
- makefiles
- Tensorflow: Computation Graphs

Abstrakte Modellierung: Topologische Sortierung

- Topologische Sortierung funktioniert nur für dags (gerichtete Graphen ohne Zyklen).
- Topologische Sortierung eines dag $G = (V, E)$:
  Sortiere Knoten in linearer Ordnung, so dass für alle Knoten $u, v \in V$ gilt, dass u vor v in der Ordnung kommt, wenn $(u, v) \in E$.
- Kanten gehen nur nach rechts
- Sortierung muss nicht eindeutig sein

Topologisches Sortieren mittels DFS

```
TOPOLOGICAL-SORT(G) // G=(V,E) dag
        newLinkedList(L);
        run DFS(G) but, each time a node is finished, insert in front of L
        return L.head
```

Laufzeit = $O(|V| + |E|)$, da Einfügen in Liste vorne in Zeit $O(1)$
Wenn die Anzahl eingehender Kanten bekannt ist:

```
Kahn1962(G) //G=(V,E) dag, inbound[u]=#edges to u
        WHILE !isEmpty(V) DO
                pick vertex u with inbound[u]==0
                add u at the end of a list L
                inbound[v]=inbound[v]-1 for each v with (u,v) in E
                remove u from V and all (u,*) from E
```

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

Starke Zusammenhangskomponenten

Eine starke Zusammenhangskomponente eines gerichteten Graphen $G = (V, E)$ ist eine Knotenmenge $C \subseteq V$, sodass

1. es zwischen je zwei Knoten $u, v \in C$ einen Pfad von u nach v gibt
2. es keine Menge $D \subseteq V$ mit $C \subset D$ gibt, für die 1. auch gilt (C ist maximal)
   Ein Graph kann mehrere starke Zusammenhangskomponenten (strongly connected components, SCCs) haben.

Eigenschaften

Verschiedene SCCs sind disjunkt
Zwei SCCs sind nur in eine Richtung verbunden:
Seien C, D SCCs, $u, v \in C, w, x \in D$ und gebe es einen Pfad $u \to w$.
Dann kann es keinen Pfad $x \to v$ geben (C, D sonst identisch).

SCC Algorithmus

Lasse zweimal DFS laufen:
Auf G und auf transponiertem Graphen $G^T := (V, E^T)$ mit $E^T := \{(v, u) : (u, v) \in E\}$ (Kanten umgedreht)

SCCs im transponierten Graphen

- SCCs bleiben identisch in G, G^T
- nur Übergänge drehen sich um

Algorithmus

```
SCC(G) // G=(V,E) directed graph
        run DFS(G)
        compute G_T
        run DFS(G_T) but visit vertices in main loop in descending finish time from 1
        output each DFS tree in 3 as one SCC
```

- `G_T` ist G^T
- Laufzeit = $O(|V| + |E|)$
- Graph muss nicht zusammenhängend sein

Idee

- Nach 1. Durchführung der DFS liegt Knoten mit höchster finish time in einem SCC
- In transponiertem Graphen kann man diesen SCC nicht verlassen
- Nachdem nun erster SCC abgearbeitet ist, wird der verbleibende Knoten mit der höchsten finish time verwendet, wiederhole bis alle Knoten abgearbeitet sind
- Gib alle Bäume aus

Algorithmendesign

- Kosarajus Algorithmus:
  Hier betrachtet, zwei DFS Ausführungen
- Tarjans Algorithmus, Pfad-basierter Algorithmus:
  jeweils nur eine DFS-Ausführung, speichern sich mehr Informationen unterwegs
- Asymptotisch alle gleich schnell
- Tarjans und pfad-basierter Algorithmus schneller in Praxis

Minimale Spannbäume

Für einen zusammenhängenden, ungerichteten, gewichteten Graphen $G = (V, E)$ mit Gewichten w ist der Subgraph $T = (V, E_T)$ von G ein Spannbaum, wenn T azyklisch ist und alle Knoten verbindet.
Der Spannbaum ist minimal, wenn $w(T) := \sum_{\{c,v\} \in E_T} w(\{u, v\})$ minimal f.a. Spannbäume von G ist.
MST muss nicht unbedingt minimale Kantenanzahl haben, da das Gesamtgewicht minimiert wird.

Anwendung: Broadcast in Netzwerken

Broadcast: Verteile Nachricht an alle Switches
Zu verhindern: "Broadcast Storm": Nachricht stets zyklisch weiterverteilt
Spanning Tree Protocal:
Wähle "Root Bridge" als Wurzel des Spannbaums
Gewicht abhängig von Geschwindigkeit und Entfernung von Root Bridge

Allgemeiner MST-Algorithmus

```
genericMST(G,w) // G=(V,E) undirected, connected graph, w weight function
        A=∅
        WHILE A does not form a spanning tree for G DO
                find safe edge {u,v} for A
                A = A ∪{{u,v}}
        return A
```

- `A` Teilmenge der Kanten eines MST
- Kante `{u,v}` ist sicher, wen $A \cup \{\{u, v\}\}$ noch Teilmenge eines MST ist

Terminologie

- Schnitt $(S, V \backslash S)$ partitioniert Knoten des Graphen in 2 Mengen
- $\{u, v\}$ überbrückt Schnitt $(S, V \backslash S)$, wenn $u \in S, v \in (V \backslash S)$
- Schnitt $(S, V \backslash S)$ respektiert $A \subseteq E$, wenn keine Kante $\{u, v\} \in A$ Schnitt überbrückt
- $\{u, v\}$ leichte Kante für $(S, V \backslash S)$, wenn $w(\{u, v\})$ minimal f.a. den Schnitt überbrückenden Kanten

Leicht = sicher

$\{u, v\}$ sicher für A, wenn $A \cup \{\{u, v\}\}$ Teilmenge eines MST
Sei A Teilmenge eines MST, $(S, V \backslash S)$ Schnitt, der A respektiert, $\{u, v\}$ eine leichte Kante, die den Schnitt überbrückt.
Dann ist $\{u, v\}$ sicher für A.

Algorithmendesign

Leicht = sicher $\rightsquigarrow$ Greedy-Strategie für konkrete Implementierung

- Kruskal lässt parallel mehrere Unterbäume eines MST wachsen
- Prim konstruiert MST Knoten für Knoten
  Beide Algorithmen funktionieren auch für negative Kantengewichte

Algorithmus von Kruskal

```
MST-Kruskal(G,w) // G=(V,E) undirected, connected graph, w weight function
        A=∅
        FOREACH v in V DO set(v)={v};
        Sort edges according to weight in nondecreasing order
        FOREACH {u,v} in E according to order DO
                IF set(u)!=set(v) THEN // u and v connected otherwise,
                        A = A ∪{{u,v}}     // adding {u,v} would create cycle
                        UNION(G,u,v);
        return A
```

- Jeder Knoten hat Attribut `set`
- `UNION(G,u,v)` setzt $set(w) = set(u) \cup set(v)$ f.a. Knoten $w \in set(u) \cup set(v)$
- $set(u), set(v)$ sind disjunkt oder identisch

Laufzeit

Mit vielen Optimierungen (komplexere Datenstruktur, ...): Laufzeit = $O(|E| \cdot \log |E|)$
Laufzeit = $O(|E| \cdot \log |V|)$

Algorithmus von Prim

```
MST-Prim(G,w,r) // r root in V, MST given through v.pred values
        FOREACH v in V DO {v.key=∞; v.pred=NIL;}
        r.key=-∞; Q=V;
        WHILE !isEmpty(Q) DO
                u=EXTRACT-MIN(Q); // smallest key value
                FOREACH v in adj(u) DO
                        IF v∈Q and w({u,v})<v.key THEN
                                v.key=w({u,v});
                                v.pred=u;
```

- Algorithmus fügt, beginnend mit Wurzelknoten, immer leichte Kante zu zusammenhängender Menge hinzu
- Auswahl der nächsten Kante gemäß `key`-Wert, der stets aktualisiert wird

- A implizit definiert durch $A = \{\{v, v.pred\} : v \in (V \backslash (\{r\} \cup Q))\}$

Laufzeit

Laufzeit = $O(|E| + |V| \cdot \log |V|)$
mit vielen Optimierungen, speziell Fibonacci-Heaps

Kürzeste Wege in (gerichteten) Graphen

Single-Source Shortest Path (SSSP)

Finde von Quelle s aus jeweils den kürzesten Pfad zu allen anderen Knoten
Kürzester Pfad: Gesamtgewicht reduzieren, Länge egal
Länge eines Pfades $p := (v_1, \ldots, v_k) \in V^k$ von $u = v_1$ zu $v = v_k$:

- $w(p) := \sum_{i=1}^{k-1} w((v_i, v_{i+1}))$
- $shortest(u, v) := \begin{cases} \min\{w(p) : p \text{ Pfad von } u \text{ nach } v\} & \text{wenn } v \text{ erreichbar von } u \\ \infty & \text{sonst} \end{cases}$

SSSP vs. BFS, DFS, MST

- BFS, DFS beachten Kantengewichte nicht
- BFS findet kürzeste "Kantenwege", nicht kürzeste "Gewichtswege"
- MST (für ungerichtete Graphen) minimiert Gesamtgewicht des Baumes
  - Dies bedeutet nicht unbedingt eine Minimierung der Gewichtswege

Zyklen und negative Kantengewichte

- Negative Kantengewichte sind erlaubt
- Zyklen mit negativem Gesamtgewicht sind nicht erlaubt
- Kürzeste Pfade können keine Zyklen mit positivem Gesamtgewicht haben
- Kürzeste Pfade enthalten höchstens Zyklen mit Gesamtgewicht 0
- Es gibt stets einen kürzesten Pfad mit Kantenlänge $\leq |V| - 1$

Kürzeste Teilpfade

- Teilpfad $s \to x$ eines kürzesten Pfades $s \to x \to z$ ist auch kürzester Pfad von s nach x
- Sonst gäbe es ja kürzeren Pfad von s nach x

Algorithmen für SSSP

- Gemeinsame Idee: Lockerung/Relaxation
- Bellman-Ford funktioniert allgemein, auch ungerichtet
  - Laufzeit = $O(|V| \cdot |E|)$
- Algorithmus für dags (gerichtete, azyklische Graphen) funktioniert nur für dags
  - Laufzeit = $O(|V| + |E|)$
- Dijkstra funktioniert nur für nicht-negative Kantengewichte, auch ungerichtet
  - Laufzeit = $O(|V| \cdot \log |V| + |E|)$

relax, initSSSP

```
relax(G,u,v,w)
        IF v.dist > u.dist + w((u,v)) THEN
                v.dist=u.dist + w((u,v));
                v.pred=u;
```

Verringere aktuelle Distanz von Knoten v, wenn durch Kante (u, v) kürzere Distanz erreichbar

```
initSSSP(G,s,w)
        FOREACH v in V DO
                v.dist=∞;
                v.pred=NIL;
        s.dist=0;
```

Zu Beginn: Distanz = ∞ f.a. Knoten $\neq s$

Bellman-Ford-Algorithmus

```
Bellman-Ford-SSSP(G,s,w)
        initSSSP(G,s,w);
        FOR i=1 TO |V|-1 DO
                FOREACH (u,v) in E DO
                        relax(G,u,v,w);
        FOREACH (u,v) in E DO // check for negative cycle
                IF v.dist > u.dist+w((u,v)) THEN
                        return false;
        return true;
```

Laufzeit = $\Theta(|E| \cdot |V|)$ wegen geschachtelter `FOR`-Schleifen
Testet, ob negativer Zyklus erreichbar, gibt `false` zurück, falls dies zutrifft

SSSP mittels topologischer Sortierung

```
TopoSort-SSSP(G,s,w) // G dag
        initSSSP(G,s,w);
        execute topological sorting
        FOREACH u in V in topological order DO
                FOREACH v in adj(u) DO
                        relax(G,u,v,w);
```

Kanten auf dem kürzesten Pfad werden nacheinander gelockert
Laufzeit = $\Theta(|E| + |V|)$

Dijkstra-Algorithmus

```
Dijkstra-SSSP(G,s,w)
        initSSSP(G,s,w);
        Q=V; // let S=V\Q, Q is a set
        WHILE !isEmpty(Q) DO
                u=EXTRACT-MIN(Q); // wrt. dist
                FOREACH v in adj(u) DO
                        relax(G,u,v,w);
```

Voraussetzung: $w((u,v)) \leq 0$ f.a. u, v, also alle Kanten
Laufzeit = $\Theta(|V| \cdot \log |V| + |E|)$ mittels Fibonacci-Heaps

Negative Kanten

Wenn man den absoluten Wert der kleinsten Kante zu allen Werten addiert, addiert man den Wert so oft, wie Anzahl Kanten auf dem kürzesten Weg

A*-Algorithmus

- Suche kürzesten Weg von s zu einem Ziel t
- Dijkstra sucht lokal vom gegenwärtigen Punkt aus günstigsten nächsten Schritt, ignoriert aber Zielrichtung
- Füge Heuristik hinzu, die vom Ziel her denkt

```
A*(G,s,t,w)
        init(G,s,t,w);
        Q=V; //let S=V-Q
        WHILE !isEmpty(Q) DO
                u=EXTRACT-MIN(Q);
                IF u==t THEN break;
                FOREACH v in adj(u) DO
                        relax(G,u,v,w);
```

- Jeder Knoten u bekommt Attribut `u.heur` zugewiesen, z.B. Abstand Luftlinie zum Ziel
- `EXTRACT-MIN` sucht Minimum über `u.dist + u.heur`
- Auch nicht-negative Kantengewichte
- Dijkstra ist A* mit Heuristik 0
- A* mit monotoner Heuristik ist Dijkstra mit Kantengewichten `w(u,v)+v.heur-u.heur` und `s.dist=s.heur`
  - A *und Dijkstra wählen dann gleiche Knoten, da* `u.dist+u.heur==u.dist`, `t.dist==t.dist` *(linke Seite A*, rechte Seite Dijkstra)

Bedingungen für optimale Lösung

1. Heuristik überschätzt nie Kosten: `u.heur` $\leq shortest(u,v)$, `t.heur == 0`
2. Heuristik ist monoton: F.a. $u, v \in E$ gilt `u.heur` $\leq$ `w(u,v) + v.heur`

Maximaler Fluss in Graphen

Netzwerkflüsse

Idee

Kanten haben (aktuellen) Flusswert und (maximale) Kapazität
Ziel: Finde maximalen Fluss von s nach t
Jeder Knoten außer s und t hat gleichen ein- und ausgehenden Fluss

Ein Flussnetzwerk ist ein gewichteter, gerichteter Graph $G = (V, E)$ mit Kapazität(sgewicht) c, sodass $c(u,v) \geq 0$ für $(u,v) \in E$ und$c(u,v) = 0$ für $(u,v) \notin E$; mit zwei Knoten $s, t \in V$ (Quelle, Senke), sodass jeder Knoten von s aus erreichbar ist und t von jedem Knoten aus erreichbar ist.
Ein Fluss $f : V \times V \to \mathbb{R}$ für ein Flussnetzwerk $G = (V, E)$ mit Kapazität c, Quelle s, Senke t erfüllt:

- $0 \leq f(u,v) \leq c(u,v)$ f.a. $u, v \in V$ (Fluss zwischen 0 und max. Kapazität)
- $\sum_{v \in V} f(u,v) = \sum_{v \in V} f(v,u)$ f.a. $u \in (V \backslash \{s,t\})$ (gleicher ein- und ausgehenden Fluss)

Maximale Flüsse

Der Wert $|f|$ eines Flusses $f : V \times V \to \mathbb{R}$ für ein Flussnetzwerk $G = (V, E)$ mit Kapazität c, Quelle s, Senke t ist $|f| := \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s)$

Transformationen

Antiparallele Kanten sind durch neue Knoten zu elimieren:

Mehrere Quellen und Senken sind zu vereinigen, indem eine neue Quelle/Senke mit Verbindungen zu allen andere und unendlicher Kapazität kreiert wird.

Ford-Fulkerson-Methode

Suche Pfad von s nach t, der noch erweiterbar (bzgl. des Flusses) ist, Pfad wird im Restkapazitätsgraphen gesucht, der die möglichen Zu- und Abflüsse beschreibt

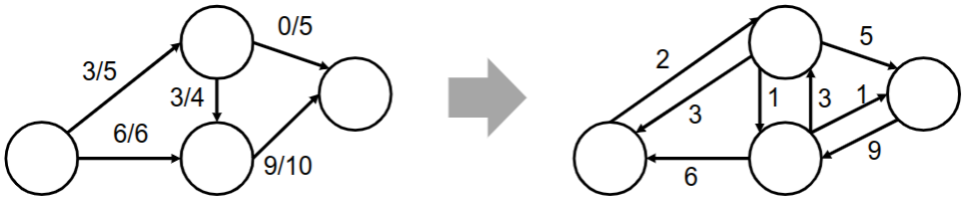
Reste

Restkapazität $c_f(u,v) := \begin{cases} c(u,v) - f(u,v) & \text{falls } (u,v) \in E \\ f(v,u) & \text{falls } (v,u) \in E \\ 0 & \text{sonst} \end{cases}$
Fall 1: Wie viel eingehenden Fluss über (u,v) könnte man zu v hinzufügen?

Fall 2: Wie viel abgehenden Fluss über (u,v) kann man wegnehmen und so zu v hinzufügen?
Wohldefiniert, da nach Transformationen nicht mehr sowohl (u,v) als auch (v,u) im Netzwerk sind

Restkapazitäts-Graph

$G_f := (V, E_f)$ mit $E_f := \{(u,v) \in v \times V : c_f(u,v) > 0\}$
Im Restkapazitäts-Graphen sind antiparallele Kanten erlaubt.



Restkapazitäten ausnutzen

Finde Pfad von s zu t in G_f und erhöhe (für Kanten in G) bzw. erniedrige (für Gegenrichtung) um Minimum $c_f(u,v)$ aller Werte auf dem Pfad in G

Ford-Fulkerson-Algorithmus

```
Ford-Fulkerson(G,s,t,c)
        FOREACH e in E DO e.flow=0;
        WHILE there is path p from s to t in G_flow DO
                c_flow(p)= min{c_flow(u,v): (u,v) in p}
                FOREACH e in p DO
                        IF e in E THEN
                                e.flow=e.flow+c_flow(p)
                        ELSE
                                e.flow=e.flow-c_flow(p)
```

- G_f ist `G_flow`, c_f ist `c_flow`
- Pfadsuche z.B. per DFS/BFS
- Laufzeit = $O(|E| \cdot u \cdot |f^*|)$
  - f^* ist maximaler Fluss, Fluss wächst um bis zu $\frac{1}{u}$ pro Iteration
- Laufzeit = $O(|V| \cdot |E|^2)$ mit Verbesserung
- Es wird ein zufälliger Pfad in G_f verwendet

Max-Flow Min-Cut Theorem

Sei $f : V \times V \to \mathbb{R}$ Fluss für ein Flussnetzwerk $G = (V, E)$ mit Kapazität c, Quelle s, Senke t.
Dann sind äquivalent:

1. f ist ein maximaler Fluss für G.
2. Der Restkapazitätsgraph G_f enthält keinen erweiterbaren Pfad.
3. $|f| = \min_{S \subseteq V : s \in S, t \notin S} c(S, V \backslash S)$ mit $s \in S$, $t \in (V \backslash S)$, wobei $c(S, V \backslash S)$ für $s \in S$, $t \in (V \backslash S)$die Kapazität eines Schnitts $(S, V \backslash S)$ ist.

Beispielanwendung: Bipartites Matching

- Flussproblem ist hier overkill
- Bipartiter Graph:
  Knotenmenge zerfällt in zwei disjunkte Mengen, sodass Kanten nur zwischen den Mengen verlaufen
  - z.B. Menge an Käufern und Verkäufern
- Maximales bipartites Matching:
  Finde maximale Anzahl von Kanten, sodass jeder Käufer genau einem Verkäufer zugeordnet wird
  - Wende Transformationen an, um eine Quelle und Senke zu erhalten
  - Setze Kapazität = 1 f.a. Kanten
  - Fluss = 1 bedeutet, dass Kante aktiv ist
  - Wert des Flusses gibt an, wie viele aktive Kanten aus s ausgehen bzw. wie viele in t ankommen

7. Advanced Designs

Auswahl algorithmischer Entwurfmethoden

Divide & Conquer

Löse rekursiv (disjunkte) Teilprobleme
Siehe Quicksort, Merge Sort

Backtracking

Durchsuche iterativ Lösungsraum
Siehe Backtracking

Dynamisches Programmieren

Löse rekursiv (überlappende) Teilprobleme durch Wiederverwenden/Speichern
Siehe Dynamische Programmierung

Greedy

Baue Lösung aus Folge lokal bester Auswahlen zusammen
Siehe Greedy-Algorithmen, Kruskal, Prim, Dijkstra

Metaheuristiken

Übergeordnete Methoden für Optimierungsprobleme
Siehe Metaheuristiken

Backtracking

Prinzip

Finde Lösungen $x := (x_1, x_2, \ldots, x_n)$ per "Trial-and-Error", indem Teillösung $(x_1, x_2, \ldots, x_{i-1})$ durch Kandidaten x_i ergänzt wird, bis Gesamtlösung erreicht ist, oder bis festgestellt, dass keine Gesamtlösung erreichbar, ist, dann wird Kandidat x_{i-1} revidiert

Beispiel: Sudoku

```
SUDOKU-BACKTRACKING(B) // B[0…3][0…3] board
        IF isFull(B) THEN
                print "solution: "+B;
        ELSE
                (i,j)=nextFreePos(B);
                FOR v=1 TO 4 DO
                        IF isAdmissible(B,i,j,v) THEN // no rules broken?
                                B[i,j]=v;
                                SUDOKU-BACKTRACKING(B);
                B[i,j]=empty;
```

- Letzte Zeile wird nur ausgeführt, wenn die Teillösung nicht zum Ziel geführt hat, dann wird im vorherigen Feld die nächste Zahl versucht
  Backtracking kann man als Tiefensuche auf Rekursionsbaum betrachten, wobei aussichtslose Lösungen evtl. frühzeitig abgeschnitten werden.
  Es ist auch "intelligentere" erschöpfende Suche, die aussichtslose Lösungen vorher aussortiert

Lösungssuche

1. Finde eine Lösung
2. Finde alle Lösungen
3. Finde beste Lösung

Beispiel: Regulärer Ausdruck

- Mustersuche in Strings
- Aufwand kann exponentiell werden

Dynamische Programmierung

Prinzip

- Teile Problem in (überlappende) Teilprobleme
- Löse rekursiv Teilprobleme, verwende dabei Zwischenergebnisse wieder (Memoization)
- Rekonstruiere Gesamtlösung
- Schwierigkeit: Finden geeigneter Rekursionen

Beispiel: Fibonacci

```
Fib-Rek(n) // n>=1
        IF n=<2 THEN
                return 1;
        ELSE
                return Fib-Rek(n-1)+Fib-Rek(n-2);
```

- Vereinfachte Laufzeitabschätzung: $T(n) \in \Theta(2^n)$
- Werte werden mehrfach berechnet
- Lösung: Werte zwischenspeichern (Memoization)

Fibonacci mit Memoization

```
FibDyn(n) // n>=1
        F[]=ALLOC(n); //F_i at F[i-1]
        FOR i=0 TO n-1 DO F[i]=0;
        return FibDynRek(n-1,F);

FibDynRek(i,F) // i>=0
        IF F[i]!=0 THEN return F[i]; // already calculated
        IF i=<1 THEN
                f=1
        ELSE
                f=FibDynRek(i-1,F)+FibDynRek(i-2,F);
        F[i]=f;
        return f;
```

- Wenn Basisfall erreicht ist, nur noch Addieren und Auslesen zu tun
- Laufzeit $\Theta(n)$

Minimum Edit Distance/Levenshtein-Distanz

Ziel:

- Messen der Ähnlichkeit von Texten
- Definiere 3 Buchstaben-Operationen:
  1. `ins(S,i,b)`: fügt an `i`-ter Position Buchstabe `b` in String `S` ein
  2. `del(S,i)`: löscht an `i`-ter Position Buchstaben in `S`
  3. `sub(S,i,b)`: ersetzt an `i`-ter Position in S den Buchstaben durch `b`
- Messe Ähnlichkeit anhand der Anzahl der benötigten Operationen zur Überführung zweier Texte ineinander
- Kosten/Operation ist 1, manchmal 2 für Substitution
- Nutze noch `copy(S,i)` für das Kopieren des `i`-ten Buchstabens, Kosten: 0
  Algorithmische Sichtweise:
- String `X[1..m]` ist von links nach rechts in String `Y[1..n]` zu überführen
- Zu jedem Zeitpunkt ist `X[1..i]` bereits in `Y[1..j]` transformiert
  `D[i][j]` sei Distanz, um `X[1..i]` in `Y[1..j]` zu überführen (`i,j >= 1`)
  Betrachte nun nächsten Schritt um `X` in `Y` zu überführen:

- `copy`: `D[i][j] = D[i-1][j-1]`
  Bereits `X[1..i-1]` in `Y[1..j-1]` überführt, jetzt kostenfrei kopieren
- `sub`: `D[i][j] = D[i-1][j-1] + 1`
  Bereits `X[1..i-1]` in `Y[1..j-1]` überführt, jetzt ersetzen
- `del`: `D[i][j] = D[i-1][j] + 1`
  Bereits `X[1..i-1]` in `Y[1..j]` überführt, jetzt `X[i]` löschen
- `ins`: `D[i][j] = D[i][j-1] + 1`
  Bereits `X[1..i]` in `Y[1..j-1]` überführt, jetzt `Y[j]` einfügen
  Fasse `copy` und `sub` zusammen:
- `copy/sub`: `D[i][j] = D[i-1][j-1] + (X[i] != Y[j])` (Ausdruck ist 1 wenn wahr, sonst 0)
  Suche nach der besten Strategie ist nun:
  `D[i][j] = min {D[i-1][j-1] + (X[i] != Y[j]), D[i-1][j] + 1, D[i][j-1] + 1}`
  Es gilt noch folgendes für die Ränder:
- `D[0][j] = j`: Füge `j` Buchstaben `Y[1..j]` zu leerem String `X[1..0]` hinzu
- `D[i][0] = i`: Lösche `i` Buchstaben `X[1..i]`, um leeren String `Y[1..0]` zu erhalten

Algorithmus

verwendet dynamische Programmierung und Memoization

```
MinEditDist(X,Y,m,n) // X=X[1..m], Y=Y[1..n]
        D[][]=ALLOC(m,n);
        FOR i=0 TO m DO D[i][0]=i;
        FOR j=0 TO n DO D[0][j]=j;
        FOR i=1 TO m DO
                FOR j=1 TO n DO
                        IF X[i]=Y[j] THEN s=0 ELSE s=1;
                        D[i][j]=min{D[i-1][j-1]+s,D[i-1][j]+1,D[i][j-1]+1};
        return D[m][n];
```

Laufzeit und Speicherbedarf $\Theta(mn)$
Dieser Algorithmus füllt das zweidimensionale Array `D[][]` mit den Distanzen. Es gilt:

Rückwärtsschrittfolge `D[m][n]` zu `D[0][0]` entlang der ausgewählten Minima (bzw. bis Rand erreicht) gibt Operationen an:

(↘) : copy (±0)
↘ : sub (+1)
→ : del
↓ : ins

	X[1…i]								
D	0	1	2	3	4	5	6	7	8
0	0	1	2	3	4	5	6	7	8
1	1	1	2	3	4	5	6	7	8
2	2	2	2	3	4	5	6	7	8
3	3	2	3	3	4	5	5	6	7
4	4	3	2	3	4	5	6	6	7
5	5	4	3	3	3	4	5	6	7
6	6	5	4	4	4	4	5	6	7
7	7	6	5	5	5	5	4	5	6
8	8	7	6	6	6	6	5	4	5
9	9	8	7	7	7	7	6	5	4
10	10	9	8	8	8	8	7	6	5

Y[1…j]

Bei den Diagonalen Schritten gilt:

- Es wird kopiert, wenn die Distanz sich nicht verändert
- Es wird substituiert, wenn sich die Distanz erhöht
  Im Allgemeinen gibt es mehrere mögliche Sequenzen

Greedy-Algorithmen

Prinzip

Finde Lösung $x = (x_1, x_2, \ldots, x_n)$, indem Teillösung (x_1, x_2, x_{i-1}) durch den Kandidaten x_i ergänzt wird, der lokal am günstigsten erscheint

Beispiele

- Dijkstra: Wähle immer Knoten, der die kürzeste Distanz hat
- Kruskal: Wähle jeweils leichteste Kante
  Greedy-Algorithmen funktionieren oft, aber nicht immer (z.B. Dijkstra und negative Kantengewichte)

Traveling Salesperson Problem (TSP)

Gegeben vollständiger (un-)gerichteter Graph $G = (V, E)$ mit Kantengewichten $w : E \to \mathbb{R}$, finde Tour p mit minimalem Kantengewicht $w(p)$. Eine Tour ist ein Weg $p = (v_0, v_1, \ldots, v_n)$ entlang der Kanten $(v_1, v_{i+1}) \in E, i = 0, 1, \ldots, n-1$, der bis auf Start- und Endknoten $v_0 = v_n$ jeden Knoten genau einmal besucht ($V = \{v_0, v_!, \ldots, v_{n-1}\}$)
Graph $G = (V, E)$ ist vollständig, wenn es f.a. $u, v \in V, u \neq v$ eine Kante $(u, v) \in E$ gibt.
Unvollständiger Graph mit Tour lässt sich erweitern, indem man fehlende Kanten (u, v) verboten teuer macht: $w((u, v)) := |V| \cdot \max_{e \in E}\{|w(e)|\} + 1$ f.a. $(u, v) \notin E$

TSP vs. Dijkstra

- Allgemeiner TSP-Algorithmus:
  - Finde optimale Route, die durch jeden Knoten geht und zum Ausgangspunkt zurückkehrt
- Dijkstra löst anderes Problem:
  - Finde optimalen Pfad vom Ausgangspunkt aus
  - Besucht eventuell nicht alle Knoten und betrachtet auch nicht Rückkehr

Ansatz Greedy-Algorithmus für TSP

Starte mit beliebigem oder gegebenem Knoten.
Nehme vom gegenwärtigen Knoten aus die Kante zu noch nicht besuchtem Knoten, die kleinstes Gewicht hat.
Wenn kein Knoten mehr übrig, gehe zu Startpunkt zurück.

```
Greedy-TSP(G,s,w) // |V|=n, s starting node
        FOREACH v in V DO v.color=white;
        tour[]=ALLOC(n); tour[0]=s; tour[0].color=gray;
```

```
        FOR i=1 TO n-1 DO
                tour[i]=EXTRACT-MIN(adj(tour[i-1]));
                        // get (white) neighbor with minimum edge weight
                tour[i].color=gray;
        return tour;
```

Ist zu gierig!

Effizienter Algorithmus für TSP

Vermutlich schwierig zu finden
Siehe auch NP-Vollständigkeit

Metaheuristiken

Heuristik

- Dedizierter Suchalgorithmus für Optimierungsproblem, der gute (eventuell nicht optimale) Lösung für spezielles Problem findet
- Problem-abhängig: Arbeitet mit konkretem Problem

Metaheuristik

- Allgemeine Vorgehensweise, um Suche für beliebige Optimierungsprobleme zu leiten
- Problem-unabhängig: Arbeitet mit abstrakten Problemen

Lokale Suche/Hill-Climbing-Strategie

1. Finde erste Lösung
2. Suche in Nähe bessere Lösungen, bis keine Verbesserung mehr/Zeit um

Hill-Climbing-Algorithmus

```
HillClimbing(P) // maxTime constant
        sol=initialSol(P); // initial solution
        time=0;
        WHILE time<maxTime DO
                new=perturb(P,sol); // slightly alter solution
                IF quality(P,new)>quality(P,sol) THEN
                        sol=new; // replace solution with new one iff it's better
                time=time+1;
        return sol;
```

Beispiel TSP

- `initSol`: Wähle beliebige Tour, z.B. per Greedy-Algorithmus
- `perturb`: Tausche 2 zufällige Knoten
- `quality`: Gewicht der aktuellen Tour

Lokale/Globale Maxima

Eventuell bleibt Hill-Climbing-Algorithmus in lokalem Maximum hängen, da stets nur leichte Lösungsänderungen in aufsteigender Richtung!
Siehe auch: 6.3. Extremwerte

Iterative, lokale Suche

1. Führe lokale Suche durch
2. Beginne Suche nochmal von vorne, z.B. mit neuer zufälliger Lösung, eventuell auch mehrmals
3. Akzeptiere beste gefundene Lösung
   Problem: Zufällige Lösungen könnten auch schlecht sein

Simulated Annealing

- "Annealing" in Metallverarbeitung:
  Härten von Metallen durch Erhitzen auf hohe Temperatur und langsames Abkühlen
- Entscheide je nach Temperatur, in welche Richtung gesucht wird

1. Temperatur zu Beginn hoch, kühlt langsam ab
2. Je höher Temperatur, desto wahrscheinlicher Sprung in schlechte Richtung

- Mit Wahrscheinlichkeit in schlechte Richtung

Ansatz

Akzeptiere auch Lösung `new` mit `quality(new)<quality(sol)` mit Wahrscheinlichkeit:
$rand(0,1) < e^{\frac{quality(new)-(quality(sol)}{temperature}}$
$temperature$ nimmt mit Zeit ab:

- Zu Beginn heiße Temperatur: Akzeptiere oft viel schlechtere Lösungen
- Am Ende kühlere Temperatur: Akzeptiere selbst wenig schlechtere Lösungen fast nie
  Gegen Ende fast Hill-Climbing-Strategie

```
SimulatedAnnealing(P) // maxTime constant, TempSched[] temparature annealing
        sol=initialSol(P);
        time=0;
        WHILE time<maxTime DO
                new=perturb(P,sol);
                temperature=TempSched[time];
                d=quality(P,new)-quality(P,sol); r=random(0,1); // equally distributed
                IF d>0 OR r=<exp(d/temperature) THEN
                        sol=new;
                time=time+1;
        return sol;
```

- Bestimmung eines guten "Annealing schedule" (Starttemperatur und Abnahme) ist nicht Teil der Veranstaltung

Weitere Metaheuristiken

Es gibt noch viele mehr, z.B. Schwarmoptimierung, Ameisenkolonialisierung, ...

Tabu Search

- Suche bessere Lösung in der Nähe ausgehend von aktueller Lösung
- Speichere eine Zeit lang schon besuchte Lösungen, vermeide diese Lösungen
- Wenn keine bessere Lösung in der Nähe, akzeptiere auch schlechtere Lösung

Evolutionäre Algorithmen

- Beginne mit Lösungspopulation
- Wähle beste Lösungen zur Reproduktion aus
- Bilde durch Überkreuzungen und Mutationen der besten Lösungen neue Lösungen
- Ersetze schlechteste Lösungen durch diese neue Lösungen

8. NP

Ansatz

Problem ist leicht, wenn es in Polynomialzeit lösbar ist.
Worst-Case-Laufzeit des Algorithmus ist also $\Theta\left(\sum_{i=0}^{k} a_i n^i\right) = poly(n)$ mit konstanten a_i, k
Leicht zu lösende Problem:

- Sortieren eines Arrays
- Breitensuche im Graphen
- Minimale Spannbäume berechnen
- ...
  Probleme mit leicht zu überprüfender Lösung:
- TSP
- Faktorisieren
- ...
  Unentscheidbare Probleme:
- Halteproblem
- Code-Erreichbarkeit
- ...

Berechnungsprobleme vs. Entscheidungsprobleme

Berechnungsproblem:

- Gegeben: Problem P
- Gesucht: Lösung S
- Beispiel: Berechne kürzeste Pfade im Graphen
  Entscheidungsproblem:
- Gegeben: Problem P
- Gesucht: Hat P Eigenschaft E? Antwort ist wahr/falsch
- Beispiel: Ist gerichteter Graph stark zusammenhängend?
  Im Folgenden werden nur Entscheidungsprobleme betrachtet
  Man kann jedes Berechnungs- in ein Entscheidungsproblem überführen, so dass Polynomialzeit-Lösung für Entscheidungsproblem auch Polynomialzeit-Lösung für Berechnungsproblem ergibt.

Beispiel: Faktorisieren

Faktorisierungsproblem (Berechnungsproblem):

- Gegeben: n-Bit Zahl $N \geq 2$
- Gesucht: Primfaktoren von N
  $\Rightarrow$ Entscheidungsproblem:
- Gegeben: n-Bit Zahl $N \geq 2$, zahl B
- Gesucht: Ist kleinster Primfaktor von N maximal B?

```
Factorize(N) // N>1
        WHILE N>1 DO
                p=computeFactor(N);
                print p;
                N=N/p;

computeFactor(N) // use decideFactor(N,B) as sub, N>1, computes prime factor of N
        L=1; U=N;
        WHILE L!=U DO
                M=L+floor((U-L)/2);
                IF decideFactor(N,M)==1 THEN U=M ELSE L=M+1;
        return L;

decideFactor(N,B)
        ...
        return d; // d==0 or d==1
```

- `sub` steht für Suboutine
  In jeder Iteration wird Suchintervall um Hälfte reduziert, runden kann man ignorieren
  Zu Beginn Intervalllänge N, also nach $\Theta(\log_2 N) = \Theta(n)$ Iterationen fertig
  Laufzeit $\Theta(\log_2 N) = \Theta(n)$ Iterationen von `decideFactor`, in jeder Iteration konstanter Aufwand
  Laufzeit `Factorize`:
  In jeder Iteration wird Primfaktor $p \geq 2$ abgespalten, also maximal $\Theta(\log_2 N) = \Theta(n)$ Iterationen
  Annahme der Laufzeit von `decideFactor`: $poly(n)$
  Gesamtlaufzeit $\Theta(n^2 \cdot poly(n))$

Berechnung durch Entscheidung

Berechnungsproblem:

- Gegeben: Problem P
- Gesucht: Lösung S
  Kreiere daraus Entscheidungsproblem:
- Gegeben: Problem P, String s
- Gesucht: Ist s Präfix der Binärdarstellung einer Lösung S?

```
compute(P) // use decide(P,s) as sub
        s=""; // empty string
        IF decide(P,s)==0 THEN return "no solution";
        done=false;
        WHILE !done DO
                szero=decide(P,s+"0");
                sone =decide(P,s+"1");
                IF szero==0 AND sone==0 THEN // solution found
                        done=true
                ELSE IF szero==1 THEN s=s+"0" ELSE s=s+"1";
        return "solution " + s;

decide(P,s)
        ...
        return d; // d==0/d==1
```

Sofern Bitlänge der Lösungen polynomiell beschränkt ist und `decide` in Polynomialzeit läuft, läuft `compute` auch in Polynomialzeit
Es wird bit-weise in richtige Richtung gesucht
Laufzeit: $\Theta\left(2 \cdot \max_S |S| + 1\right)$ Iterationen von `decide`

Komplexitätsklassen P und NP

Komplexitätsklasse P

Betrachte Entscheidungsproblem für Eigenschaft als Menge: $L_E := \{P : P \text{ hat Eigenschat } E\}$
L kommt von "language"
Beispiel: $L_{Sc} := \{G : G \text{ ist gerichtetet, stark zusammenhängender Graph}\}$
Komplexitätsklasse P:
Ein Entscheidungsproblem L_E ist genau dann in der Komplexitätsklasse P, wenn es einen Polynomialzeit-Algorithmus A_{L_E} mit Ausgabe 0/1 gibt, der stets korrekt entscheidet, ob eine Eingabe P die Eigenschaft E hat oder nicht, also $P \in L_E \Leftrightarrow A_{L_E}(P) = 1$ für alle P gilt.
Eigentliche Definition: Algorithmus = Turing-Maschine und Problem-Universum = $\{0,1\}^*$

Komplexitätsklasse NP

Das Prüfen einer vermeintlichen Lösung ist einfach für L_E:

- Gegeben: Problem P und vermeintliche Lösung S
- Entscheide: Zeigt S, dass P Eigenschaft E hat oder nicht?
- S dient als zusätzliche Entscheidungshilfe, heißt auch "witness", Zeuge, Zertifikat,... für P
  Technische Einschränkung:
  Lösungen S sind von polynomieller Komplexität in Eingabeproblem P, meist: Lösungen S haben polynomielle Bitlänge (in Bitlänge von P)

Beispiel

$L_{Fakt} := \{(N,B) : V > \text{ hat Primfaktor } \leq B\}$
Gegenwärtig ist es unklar, wie in Polynomialzeit ohne Hilfe (und ohne Quantencomputer) entschieden werden kann, ob Eingabe (N,B) in L_{Fakt} ist oder nicht
Mit Hilfe ist das Entscheiden einfach:
Zeuge S zu $P = (N,B)$ ist Faktor p von N mit $1 < p \leq B$

```
verify(N,B,p) // check alleged solution
1 IF N>1 AND 1<p=<B AND p|N THEN return 1 else return 0;
```

Es wird nicht geprüft, ob p prim ist, wenn der zusammengesetzte Faktor in der Schranke B liegt, dann ist p erst recht ein Primfaktor
Es gibt keine falshce Hilfe für nicht-zugehörige Eingaben:

- Wenn $(N,B) \in L_{Fakt}$, dann gibt es ein S, das `verify` akzeptieren lässt
- Wenn $(N,B) \notin L_{Fakt}$, dann gibt es kein S, das `verify` akzeptieren lässt
  Entscheidung mit Hilfe muss in beiden Fällen richtig sein

NP (Nicht-deterministische Polynomialzeit)

Ein Entscheidungsproblem L_E ist in der Komplexitätsklasse NP gdw. es einen Polynomialzeit-Algorithmus A_{L_E} mit Ausgabe 0/1 gibt, der bei Eingabe eines Zeugen S_P für Eingabe $P \in L_E$ bzw. für jede Eingabe S_P für Eingabe $P \notin L_E$ stets korrekt entscheidet, ob eine Eingabe P die Eigenschaft E hat oder nicht, also $P \in L_E \Leftrightarrow \exists S_P : A_{L_E}(P, S_P) = 1$ f.a. P gilt.
Äquivalent: F.a. P gilt $P \notin L_E \Leftrightarrow \forall S_P : A_{L_E}(P, S_P) = 0$
Komplexität der Hilfseingabe S_P polynomiell in der von P

P vs. NP

Jedes Problem in P ist auch in NP: Algorithmus A_{L_E} entscheidet ohne Hilfe $\rightsquigarrow$ P $\subseteq$ NP
Bis heute ist offen, ob auch NP $\subseteq$ P gilt

Mögliche Welten

Faktorisieren ist in NP, jedoch ist nicht klar, ob Faktorisieren auch in P liegt.

1. Wahrscheinlichste Welt, Bild wird durch Quantum-Computer verfeinert:
   P $\neq$ NP, Faktorisieren $\notin$ P $\rightsquigarrow$ Faktorisieren schwierig
2. P $\neq$ NP, Faktorisieren $\in$ P $\rightsquigarrow$ Faktorisieren leicht
3. P = NP $\rightsquigarrow$ Alle Probleme sind leicht

NP-Vollständigkeit

Ziel: Identifiziere schwierigsten Probleme in NP

NPC (NP-Complete): Klasse der NP-vollständigen Probleme
Eigenschaften:

1. NPC $\subseteq$ NP
2. Wenn P $\neq$ NP, dann definitiv NPC $\not\subseteq$ P

Reduktionen (Problemtransformationen)

Siehe Berechnung durch Entscheidung für Problemdefinition
Wenn das Entscheidungsproblem leicht ist, ist das Berechnungsproblem es auch
Das Entscheidungsproblem ist mindestens so schwierig wie das Berechnungsproblem

Transfer auf NP-Entscheidungsprobleme

NP-Problem L_A:

- Gegeben: Problem P
- Gesucht: Entscheidung
  Reduktion auf NP-Problem L_B:
- Gegeben: Problem Q
- Gesucht: Entscheidung
  Die Reduktion von L_A auf L_B ist Polynomialzeit-Algorithmus R, sodass gilt:
  $P \in L_A \Leftrightarrow R(P) \in L_B$ f.a. P, Schreibweise: $L_A \leq L_B$
  Die Reduktion transformiert Problem P in Problem $Q = R(P)$, sodass eine korrekte Entscheidung für Q automatisch eine korrekte Entscheidung für P liefert

```
decideA(P)
        Q=R(P);
        return decideB(Q);

decideB(Q)
        ...
        return d; // boolean
```

NP-vollständige Probleme

Komplexitätsklasse NPC (NP-vollständige Probleme):
Alle Probleme $L_C \in$ NP, sodass $L_A \leq L_C$ f.a. $L_A \in$ NP
Zwei Bedingungen an L_C:

1. $L_C \in$ NP
2. jedes NP-Problem ist auf L_C reduzierbar (L_C ist NP-hart)

Beispiel für Reduktion: Hamiltonscher Zyklus $\leq$ TSP

- HamCycle für G:
  Gibt es Tour (jeden Knoten einmal besuchen und zu Startknoten zurück) im Graphen G?
- TSP für (G, B):
  Gibt es Tour im Graphen G mit Gewicht maximal B?
  Beide Probleme sind in NP
  Reduktion:
  Existierende Kanten bekommen Gewicht 0, vervollständige anschließend Graphen mit Kanten mit Gewicht 1, setze $B = 0$
  Nun zu zeigen: $G \in$ HamCycle $\Leftrightarrow R(G) = (G^*, B) \in$ TSP

SAT: Die Mutter aller NP-vollständigen Probleme

Gegeben:
Boolesche Formel ϕ aus $\vee, \wedge, \neg$ in n variablen $x_1, x_2, \ldots, x_n$
ϕ hat polynomielle Komplexität in n
Gesucht:
Entscheide, ob ϕ erfüllende Belegung hat oder nicht
SAT $\in$ NP: Gegeben ist Belegung als Zeuge, werte Formel aus

SAT ist NP-hart

Zu zeigen:
Jedes Problem $L_A \in NP$ lässt sich auf SAT reduzieren (Definition der Härte)
Sei nun also $L_A \in NP$ ein beliebiges Problem, dann hat es einen $poly$-Algorithmus `verifyA(P,S)`.

- Man kann alles als Bits betrachten, die eine (sehr lange) boolesche Formel darstellen können
- Kodiere nun folgende Teile von `verifyA` als boolsche Formel:
  - Legitime Anfangszustände
  - Legitime Endzustände
  - Legitime Rechenschritte
- Polynomiell, da `verifyA` polynomiell ist
  Reduktion $R(P)$ von L_A auf SAT berechnet dann:
  ϕ_P(alle Eingabebits) = gültiger Anfangszustand für $P \wedge$ gültige Übergänge $\wedge$ Endzustand mit Ausgabe $d = 1$
  Wenn P in L_A ist, gibt es eine Lösung S, die `verifyA` mit $d = 1$ akzeptiert, dann gibt es aber auch eine erfüllende Belegung für "Rechenschritte" ϕ_P
  Wenn P nicht in L_A ist, gibt es keine Lösung S, die `verifyA` akzeptiert, dann gibt es aber auch keine erfüllende Belegung für "Rechenschritte" ϕ_P

SAT $\leq$ 3SAT

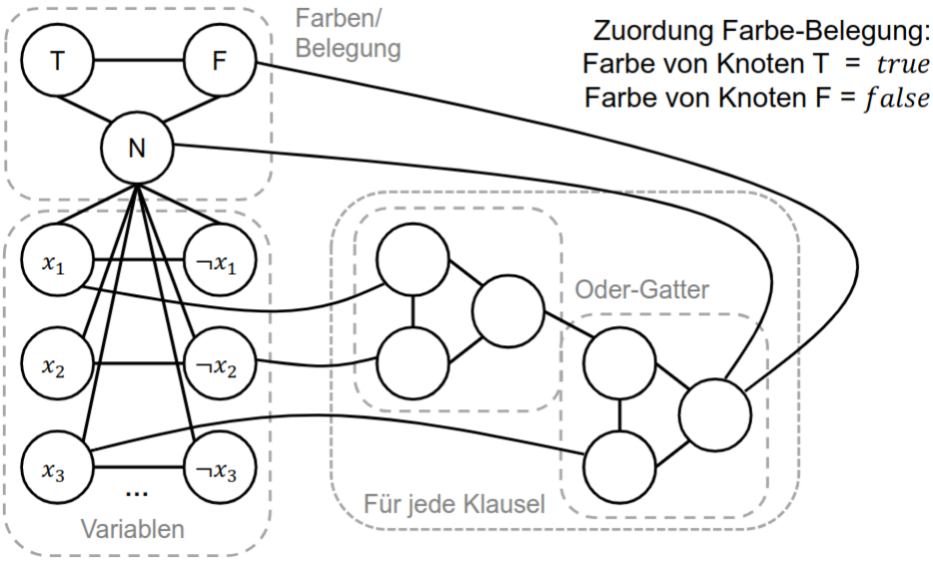
Boolesche Formeln in konjunktiver Normalform (KNF) mit jeweils 3 Literalen:
$\phi(x_1, x_2, x_3, x_4) = (\neg x_2 \wedge x_3 \wedge x_4) \vee (x_1 \wedge \neg x_2 \wedge x_3) \vee (x_4 \wedge x_3 \wedge x_4)$
KNF = Und-Verknüüfung von Klauseln, Klausel = Oder-Verknüpfung
Klausel besteht aus 3 Literalen $x_j \in \{x_j, \neg x_j\}$
Falls weniger Literale in Klausel, transformiere: $(x_j) = (x_j \wedge x_j \wedge x_j), (x_j \wedge x_k) = (x_j \wedge x_k \wedge x_k)$
3SAT:
Gegeben:
Boolesche 3KNF-Formel ϕ in n Variablen $x_1, x_2, \ldots, x_n$, ϕ hat polynomielle Komplexität in n
Gesucht:
Entscheide, ob ϕ erfüllende Belegung hat oder nicht
SAT: Boolesche Formel σ aus $\vee, \wedge, \neg$ in n Variablen $y_1, y_2, \ldots, y_n$ (σ polynomielle Komplexität in n)
lässt sich in Polynomialzeit überführen zu

3SAT: 3KNF-Formel ϕ in $poly(n)$ Variablen $x_1, x_2, \ldots, x_{poly(n)}$ (ϕ polynomielle Komplexität in n)
sodass σ erfüllbar ist gdw. ϕ erfüllbar ist $\rightsquigarrow$ SAT $\leq$ 3SAT

3-Färbbarkeit von Graphen

3COLORING für G:
Gibt es eine Knotenfärbung im Graphen G mit 3 Farben, sodass benachbarte Knoten nie die gleiche Farbe haben?
3COLORING $\in$ NP:
Gegeben Färbung, durchlaufe Knoten und prüfe jeweils Farbe der Nachbarknoten

3SAT $\leq$ 3COLORING



- Bilde Graphen dieser Struktur für alle Klauseln in gegebener 3SAT-Formel ϕ
- Der Farben/Belegung-Teil legt die Farben fest, da jeder dieser 3 Knoten eine andere Farbe haben muss
- Für jede Variable in ϕ bilde zwei Knoten mit beiden Literalen, verbinde diese
  - Verbinde alle Literalknoten mit der als N (neutral) festgelegten Farbe
  - Somit müssen alle Literale $true$ oder $false$ sein, und die Negierung ist auch erfüllt
- Lege nun mit 2 Kanten zu den Farben für jede Klausel einen Knoten als $true$ fest
- Schließe nun alle Literale entsprechend den Klauseln an jeden Klauselgraph an
- Wenn der resultierende Graph 3-färbbar ist, so ist ϕ erfüllbar

Einer für alle, alle für einen

Wenn Problem L_B NP-vollständig ist und $L_B \leq l_C$ für $L_C \in$ NP gilt, dann ist auch L_C NP-vollständig.
$L_A \leq L_B$ per Reduktion R_{AB}, $L_B \leq L_C$ per Reduktion $R_{BC} \rightsquigarrow L_A \leq L_C$ per Reduktion $R_{AC} := R_{BC} \circ R_{AB}$ (Hintereinanderausführung)
Also folgt aus 3SAT $\leq$ 3COLORING und 3COLORING $\in$ NP auch, dass 3COLORING NP-vollständig ist.

NPC – eine Auswahl

- SAT: Ist Formel ϕ erfüllbar?
- 3SAT: Ist Formel ϕ in 3KNF erfüllbar?
- 3COLORING: Ist Graph mit 3 Farben kantenkonsistent färbbar?
- HamCycle: Gibt es eine Tour im Graphen?
- TSP: Gibt es eine Tour im Graphen, mit Gesamtgewicht $\leq B$?
- VertexCover: Gibt es im Graphen eine Knotenmenge der Größe $\leq B$, sodass jede Kante an einem der Knoten hängt?
- IndependentSet: Gibt es im Graphen Knotenmenge der Größe $\geq B$, sodass kein Knotenpaar durch eine Kante verbunden ist?
- Knapsack: Für Gegenstände mit Wert und Volumen, gibt es eine Auswahl mit Gesamtwert $\geq W$, aber Gesamtvolumen $\leq V$?
- ...

P vs. NP vs. NPC

Für jedes NP-vollständige Problem L_C gilt: $L_C \in$ P $\Leftrightarrow$ P=NP.
Wenn es also einen Polynomialzeit-Algorithmus für ein $L_C \in$ NPC gibt, dann gibt es einen Polynomialzeit-Algorithmus für jedes Problem in NP.

Approximation

NPC-Probleme sind vermutlich nicht effizient lösbar, aber eventuell leicht approximierbar

```
3SAT-Approx(φ, n)
        A[]=ALLOC(n); // assigment for variables
        FOR i=1 TO n DO
                A[i]=true resp. A[i]= false with probability 1/2
        return A;
```

- Algorithmus erfüllt im Erwartungswert mindestens die Hälfte aller Klauseln

2-Färbbarkeit und 2SAT in P

2-Färbbarkeit von Graphen ist relativ einfach

Idee:

- Farbe eines Knoten bestimmt eindeutig Farben der Nachbarknoten
- Prüfe jeweils, ob Färbung Widerspruch erzeugt
  Ansatz:
- Beginne mit einem Knoten und beliebiger Farbe

- Durchlaufe den Graphen per BFS, färbe Knoten und identifiziere eventuelle Widersprüche

```
2ColoringSub(G,s,col) // G=(V,E), s node
        s.color=col; newQueue(Q); enqueue(Q,s);
        WHILE !isEmpty(Q) DO
        u=dequeue(Q);
        IF u.color==BLACK THEN nextcol=RED // change colour
        ELSE nextcol=BLACK;
        FOREACH v in adj(G,u) DO
                IF v.color==u.color THEN return 0; // check for contradiction
                If v.color==WHITE THEN // only accept nodes without colour
                        v.color=nextcol;
                        enqueue(Q,v);
        return 1; // no contradiction
```

Zunächst nur für zusammenhängenden Graphen mit vorgegebenem Startknoten und vorgegebener Startfarbe
Man muss eventuell mit anderem Startknoten nochmal starten, wie ist die Farbe zu wählen?

Von gerichtet zu ungerichtet

Betrachte ungerichteten Graphen, Lösungsmenge ändert sich nicht
Bei Neustart keine Kante zwischen Zusammenhangskomponenten:
Jede individuelle 2-Färbung der Zusammenhangskomponenten kann zu 2-Färbung des Graphen kombiniert werden

```
2Coloring(G) // G=(V,E) undirected graph
        FOREACH u in V Do u.color=WHITE;
        FOREACH u in V DO
                IF u.color==WHITE THEN
                        IF 2ColoringSub(G,u,BLACK)==0 THEN return 0;
        return 1;
```

Algorithmus findet ohne zusätzlichen Aufwand auch Färbung
Laufzeit $\Theta(|V|+|E|)$

2-SAT

Keine Symmetrie zwischen Belegungen und Farben bei 2-Färbbarkeit

Implikationsgraph aus 2-SAT

Konstruiere aus Formel ϕ (gerichteten) Implikationsgraphen $G=(V,E)$:

1. Knotenmenge V besteht aus Literalen $x_1, \neg x_1, x_2, \neg x_2, \ldots, x_n, \neg x_n$
2. Für jede Klausel $(x_j \wedge x_k)$ nehme Kanten $(\neg x_j, x_k)$ und $(\neg x_k, x_j)$ auf

Starke Zusammenhangskomponenten im Implikationsgraphen

Formel ist erfüllbar gdw. in keiner Zusammenhangskomponenten $x_j, \neg x_j$ für ein j liegen

Erfüllende Belegung berechnen

Annahme: kein x_j und $\neg x_j$ in gleicher SCC
SCC-dag:
Graph mit Superknoten aus allen Knoten einer SCC
Es gibt eine Kante zwischen SCCs, wenn es eine Kante für zwei Knoten aus den SCCs

1. Sortiere SCC-dag topologisch
2. Erfüllende Belegung (wohldefiniert, da kein x_j und $\neg x_j$ in gleicher SCC, wenn erfüllbar):
   $x_j = true$, wenn x_j in SCC nach SCC mit $\neg x_j$
   $x_j = false$, wenn $\neg x_j$ in SCC nach SCC mit x_j

MAX-2SAT

Gegeben: 2SAT-Formel ϕ, Zahl k
Gesucht: Gibt es eine Belegung, die mindestens k Klauseln erfüllt?
MAX-2SAT $\in$ NPC
MAX-2SAT $\in$ NP:
Gegeben ist eine Belegung als Zeuge; prüfe, ob mindestens k Klauseln erfüllt werden

3SAT $\leq$ MAX-2SAT

Man kann eine 3SAT-Formel $\sigma(x_1, x_2, \ldots, x_n)$ mit m Klauseln auf ein MAX-2SAT-Problem mit Formel $\phi(x_1, \ldots, x_n, w_1, \ldots, w_m)$ reduzieren.
Hierfür führt man m neue Variablen $w_1, \ldots w_m$ ein, für jede Klausel eine.
Anschließend bildet man für jede Klausel in σ folgendes Konstrukt in ϕ, hier für die h-te Klausel (X_i, X_j, X_k) dargestellt:

$$\begin{aligned}\phi(x_1, \ldots, x_n, w_1, \ldots, w_m) = \ldots &\wedge (X_i) \wedge (X_j) \wedge (X_k) \wedge (w_h) \\ &\wedge (\neg X_i \vee \neg X_j) \wedge (\neg X_i \vee \neg X_k) \wedge (\neg X_j \vee \neg X_k) \\ &\wedge (\neg X_i \vee \neg w_h) \wedge (\neg X_j \vee \neg w_h) \wedge (\neg X_k \vee \neg w_h) \wedge \ldots\end{aligned}$$

Nun gilt für $w_h = false$: Wenn die Klausel in σ nicht erfüllbar ist, dann sind maximal 6 der 10 Klauseln in ϕ erfüllbar
Wenn die Klausel in σ erfüllbar ist, dann sind bei geeigneter Wahl für w_h nie mehr als 7 Klauseln erfüllbar.
Somit lautet die Reduktion:

- Wenn σ erfüllbar ist, dann sind mindestens $k = 7m$ Klauseln in ϕ erfüllbar.
- Wenn σ nicht erfüllbar ist, dann sind weniger als $k = 7m$ Klauseln in ϕ erfüllbar.

9. Appendix

Alle Inhalte in diesem Abschnitt stammen von mir, daher bitte mit Vorsicht genießen!

String-Matching

Problemstellung und Begriffe

Problemstellung:
Finden von Textmustern P der Länge $lenPat$ in einem Text T der Länge $lenTxt$, beides sind Zeichenketten, können also auch als Buchstaben-Arrays aufgefasst werden.

Logischerweise gilt $lenPat \leq lenTxt$, die Zeichen von P und T sind alle aus demselben endlichen Alphabet Σ.
Gesucht sind nun alle gültigen Verschiebungen in, mit denen P in T auftaucht, diese sollen in einer Liste/einem Array zurückgegeben werden.
Gesucht sind also alle $sft \in \mathbb{N}$, für die $T[sft, \dots, sft + lenPat - 1] = P$ gilt, woraus wiederum folgt, dass $T[sft + j] = P[j]$ f.a. $j \in \{0, 1, \dots, lenPat - 1\}$ gelten muss.

Naives String-Matching

```
NaiveStringMatching(T,P) // P pattern to look for in text T
        lenTxt = length(T);
        lenPat = length(P);
        L = []; // list of matches
        FOR sft=0 TO lenTxt-lenPat DO
                isValid = true;
                FOR j=0 TO lenPat-1 DO
                        IF P[j] != T[sft+j] THEN isValid=false;
                IF isValid THEN
                        L = append(L,sft);
        return L;
```

- Laufzeit: $O((lenTxt - lenPat + 1) \cdot lenPat)$
- Jeder Index in T, für den potentiell ein Match gefunden werden könnte, wird untersucht, indem jeweils die nächsten $lenPat$ Zeichen untersucht werden
- Wenn es nach dem Durchlauf der Schleife nicht zu einer Unstimmigkeit kam, wird der entsprechende Index zu L hinzugefügt, sonst nicht
- Problem: Informationen aus Bearbeitung des Index sft werden bei Index $sft + 1$ nicht weitergegeben

String-Matching mit endlichen Automaten

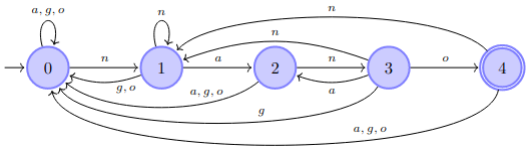
Nun werden deterministische, endliche Automaten (DFA) genutzt, um das Problem des naiven String-Matchings zu überwinden und deutlich bessere Laufzeiten zu erzielen.
Siehe auch AFE #TODO Verweis auf DFA einfügen
Hier wird eine vereinfachte, auf das Problem angepasste Version der DFAs verwendet.
Sei im Folgenden P ein gegebenes Muster der Länge $lenPat$.

- Zuerst muss eine Vorverarbeitungs-/Preprocessingphase stattfinden, in der der DFA konstruiert wird.
- Der DFA hat genau $lenPat + 1$ interne Zustände, st ist die Variable, die immer den aktuellen Zustand speichert.
- Der DFA hat als einzigen akzeptierenden Zustand $lenPat$, Startzustand ist 0.
- Die Übergangsfunktion δ gibt für jeden Zustand st und jedes eingelesene Zeichen $w \in \Sigma$ den nächsten Zustand $\delta(st, w)$ aus.
- Für den Algorithmus wird am Ende nur die Übergangsfunktion δ und die $lenPat$ als Eingabe benötigt, da der Rest nicht relevant ist, wenn der Automat wie oben konstruiert wurde.

```
FSMMatching(T,δ,lenPat)
        lenTxt = length(T);
        L = []; // list of matches
        st = 0; // first state
        FOR sft=0 TO lenTxt-1 DO
                st=δ(st,T[sft]); // get next state
                IF st=lenPat THEN // accepting state
                        L = append(L,sft-lenPat+1);
        return L;
```

- Laufzeit ohne Preprocessing: $O(lenTxt)$

Beispiel-DFA



- Hier ein DFA, der für $\Sigma = \{a, g, n, o\}$ das Muster $P = [n, a, n, o]$ erkennt
- Sämtliche DFA, die hier konstruiert werden, werden ungefähr diese Struktur haben, man kann sich bei der Konstruktion also grob hieran orientieren
- Insbesondere seien angemerkt:
  - Der Startzustand mit entsprechender Schleife für alle Buchstaben aus dem Alphabet, die nicht dem ersten Buchstaben des Musters entsprechen
  - Der akzeptierende Zustand, der für $P[0]$ als nächsten Zustand 1 hat
  - Der eindeutige Pfad des gesuchten Musters, der bei einem falschen Zeichen entsprechend auf Zustand 0, bzw. 1 für $P[0]$ zurückführt

Rabin-Karp

Idee

- Das Alphabet Σ mit $|\Sigma| = d$ wird durch die Zahlen $\{0, 1, \dots, d - 1\}$ identifiziert, hier wird zur Einfachheit $d = 10$ verwendet, für das lateinische Alphabet wäre $d = 26$ nötig
- Nehme nun p, Dezimaldarstellung des Musters P, und vergleiche diese mit entsprechend langen Abschnitten aus T mit $t_{sft} := T[sft, \dots, sft + lenPat - 1]$, dann gilt für ein Match an der Stelle sft, wenn $t_{sft} = p$ gilt

Einfacher Algorithmus

```
RabinKarpMatchBasic(T,P)
        n = T.length; m = P.length;
        h = 10^(m-1); // biggest power of 10
        p = 0; t_0 = 0; L = []; // initialise variables
        FOR i=0 TO m-1 DO
                p = 10p + P[i]; // calculate decimal representation of P
                t_0 = 10t_0 + T[i]; // calculate initial value for t_sft with sft=0
        FOR sft=0 TO n-m DO
                IF p==t_sft THEN // match found
                        L = append(L,sft);
                IF sft<n-m THEN // iff there is another iteration of the loop
                        t_(sft+1) = 10(t_sft - T[sft]h) + T[sft+m]; // next t_sft value
        return L;
```

- Die Berechnung von t_{sft+1} erfolgt folgendermaßen:
  1. Die höchste Stelle wird abgezogen, dafür ist die Berechnung von `h` notwendig, das die höchste Zehnerpotenz in dem Muster ist

2. Die nun verbleibende Zahl wird mit 10 multipliziert, um sie zu "verschieben"
3. Der nächste Eintrag in T wird addiert, er füllt die in 2. frei gewordene Stelle

- Problem:
  Mit wachsender Länge des Musters werden die arithmetischen Berechnungen zu groß, um sie als konstant anzusehen

Weniger einfacher Algorithmus

```
RabinKarpMatch(T,P,q) // q is a prime number
        n = T.length; m = P.length;
        h = (10^(m-1)) (mod q);
        p = 0; t_0 = 0; L = [];
        FOR i=0 TO m-1 DO
                p = (10p + P[i]) (mod q);
                t_0 = (10t_0 + T[i]) (mod q);
        FOR sft=0 TO n-m DO
                IF p==t_sft THEN // potential match
                        b = true;
                        FOR j=0 TO m-1 DO // check for match
                                IF P[j] != T[sft + j] THEN
                                        b = false;
                                        break;
                        IF b THEN
                                L = append(L,sft);
                IF sft<n-m THEN // iff there is another iteration of the loop
                        t_(sft+1) = (10(t_sft - T[sft]h) + T[sft+m]) (mod q);
        return L;
```

- Lösung für Problem: modulo-Rechnung mit einer Primzahl bei t_{sft} und p
- Nun kann es jedoch false positives geben, daher muss im Falle eines potentiellen Matches nochmal überprüft werden, ob es sich wirklich um ein Match handelt

Rekursionsbäume

Grobe Struktur eines Rekursionsbaums:

- Wurzel ist Initialaufruf
- Für jeden rekursiven Aufruf in einer Ausführung erhält der Knoten des Aufrufs einen Kindknoten
- Gibt es keine rekursive Aufrufe, z.B. beim Anker, so ist der Knoten ein Blatt