

Grundlagen: Algorithmen & Datenstrukturen: TL;DR

(Version: 10. August 2020)

Zusammenfassung von Robyn Kölle zur Vorlesung
Grundlagen: Algorithmen und Datenstrukturen
von PD Dr. Tobias Lasser, basierend auf den
Vorlesungsfolien. Ich übernehme keinerlei Haftung,
Fehler bitte melden.

Robyn Kölle

Basics

Effizienzmaße:

- I_n : Menge der Instanzen der Größe n eines Problems

Instanz: Algorithmus mit einer bestimmten Eingabe

- Worst-Case - Laufzeit:

$$t(n) = \max \{T(i) \mid i \in I_n\} \quad T: \text{Laufzeit des Algorithmus}$$

pessimistische Abschätzung, liefert Garantie für Laufzeit

- Average - Case - Laufzeit:

$$t(n) = \frac{1}{|I_n|} \cdot \sum_{i \in I_n} T(i)$$

durchschnittliche Laufzeit, nicht unbedingt typischer Fall in der Praxis

- Best-Case - Laufzeit:

$$t(n) = \min \{T(i) \mid i \in I_n\}$$

optimistisch Vergleich mit worst case liefert Aussage über Abweichung innerhalb Instanzen gleicher Größe

- Tatsächliche Eingabeverteilung:

$$t(n) = \sum_{i \in I_n} p_i \cdot T(i) \quad p: \text{probability}$$

Landau - Notation:

Funktionen, die asymptotisch ...

$O(f)$: nicht schneller

$\Omega(f)$: nicht langsamer

$\Theta(f)$: gleich ... als f wachsen

$o(f)$: langsamer

$\omega(f)$: schneller

Übersicht: „Standard“-Funktionen:

$$O(1) \leq O(\log n) \leq O(\sqrt{n}) \leq O(n) \leq O(n \log n) \\ \leq O(n^k) \leq O(2^n) \leq O(n!) \leq O(n^n)$$

Übersicht: Funktionen einordnen

Seien $f(n), g(n)$ Funktionen:

- $f \in O(g) \iff \limsup_{n \rightarrow \infty} \left| \frac{f(n)}{g(n)} \right| < \infty$
- $f \in \Omega(g) \iff \liminf_{n \rightarrow \infty} \left| \frac{f(n)}{g(n)} \right| > 0$
- $f \in \Theta(g) \iff 0 < \liminf_{n \rightarrow \infty} \left| \frac{f(n)}{g(n)} \right| \leq \limsup_{n \rightarrow \infty} \left| \frac{f(n)}{g(n)} \right| < \infty$
- $f \in o(g) \iff \lim_{n \rightarrow \infty} \left| \frac{f(n)}{g(n)} \right| = 0$
- $f \in \omega(g) \iff \lim_{n \rightarrow \infty} \left| \frac{f(n)}{g(n)} \right| = \infty$

Quelle: <https://de.wikipedia.org/wiki/Landau-Symbole>

Zufallsvariable:

- $X: \Omega \longrightarrow \mathbb{R}$ diskret, falls Ω endlich
 $\nwarrow$ Ergebnismenge
- Wertebereich $W_X := X(\Omega) := \{x \in \mathbb{R} \mid \exists \omega \in \Omega \text{ s.d. } X(\omega) = x\}$
- Wahrscheinlichkeit $Pr[X=x] = \sum_{\substack{\omega \in \Omega, \\ X(\omega)=x}} Pr[\omega]$
- Erwartungswert $E[X] := \sum_{\omega \in \Omega} X(\omega) \cdot Pr[\omega]$
- Falls Ereignisse gleich wahrscheinlich: $E[X] = \frac{1}{|\Omega|} \sum_{\omega \in \Omega} X(\omega)$

- Linearität der Erwartungswerts:

Für $X = \sum_{i=1}^n a_i X_i$ ($a_1, \dots, a_n \in \mathbb{R}$) gilt:

$$\mathbb{E}[X] = \sum_{i=1}^n a_i \mathbb{E}[X_i]$$

Master - Theorem:

- Rekursionsgleichungen auflösen
- Satz (vereinfachtes Master - Theorem):

Seien $a, b, c, d \in \mathbb{R}_{>0}$ Konstanten, $n := b^k$ mit $k \in \mathbb{N}_0$.

Für die Rekursionsgleichung

$$r(n) = \begin{cases} a & \text{falls } n=1 \\ cn + d \cdot r\left(\frac{n}{b}\right) & \text{falls } n>1 \end{cases}$$

gilt:

$$r(n) = \begin{cases} \Theta(n) & \text{falls } d < b \\ \Theta(n \log n) & \text{falls } d = b \\ \Theta(n^{\log_b d}) & \text{falls } d > b \end{cases}$$

Sequenzen

Sequenzen:

- lineare Struktur $S = \langle e_0, \dots, e_{n-1} \rangle$

Klassische Repräsentationen:

- Array (Zugriff über Index)

Vorteil: direkter Zugriff über Index $s[i]$

Nachteil: dynamische Größenänderung ineffizient

- Liste (Zugriff über Vorgänger / Nachfolger)

Vorteil: effizientes Einfügen / Löschen von Teilsequenzen

Nachteil: kein direkter Zugriff auf Elemente, im Speicher verteilt

Wichtige Operationen:

- `get(index)`
- `set(index, element)`
- `pushBack(element)` hinten einfügen
- `popBack()` hinten löschen
- `size()`

Dynamische Arrays:

- wenn Array-Größe nicht mehr ausreicht: neues Array mit doppelter Größe (Daten kopieren)
- wenn Array zu groß ($|\text{Elemente}| \leq \frac{1}{4} \text{Arraygröße}$): neues Array mit halber Größe

Doppelt verkettete Listen:

- jedes Element hat Vorgänger & Nachfolger
- Dummy-Element für Vorgänger des ersten / Nachfolger des letzten Elements

Wichtige Operationen:

- `splice(Handle a, Handle b, Handle t)`:
entfernt $\langle a, \dots, b \rangle$ aus Sequenz und fügt hinter t ein
- `isEmpty()` true falls $h == h.\overset{\text{Head}}{\text{next}}()$
- `first()`
- `last()`
- `remove(Handle)`
- `moveAfter(Handle, Handle)`
- ...

Einfach verkettete Listen:

- Elemente haben nur Nachfolger, keine Vorgänger

Besonderheiten:

- `splice(Handle a, Handle b, Handle t)` benötigt `a.previous` (da dessen `next` auf `b.next` gesetzt werden muss)
- sinnvoll wenn `findNext()` Vorgänger des Treffers liefert (zum Löschen)
- Pointer auf letztes Element sinnvoll s.d. `pushBack` in $O(1)$

Stacks & Queues:

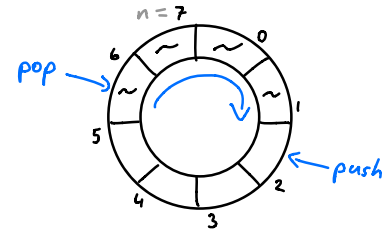
- `FIFO`: first in first out $\rightarrow$ Queue
- `LIFO`: last in first out $\rightarrow$ Stack
- Dequeues (double-ended-queues) $\leftrightarrow$ 0 0 0 0 0 0 0 $\leftrightarrow$

Operationen:

- Stack
 - pushBack
 - popBack
 - last bzw. top
- Queue
 - pushBack
 - popFront
 - first

Circular Queues:

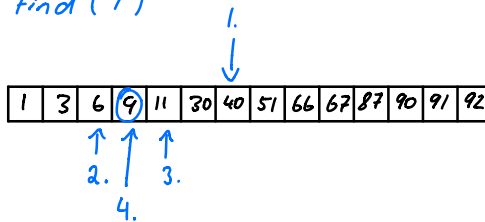
- max. Anzahl Elemente n
- Array der Größe $n+1$
- mind. ein Array-Element bleibt immer frei (um zu entscheiden ob voll / leer)
- Zugriff per Index möglich



Binäre Suche: $O(\log n)$

- in sortierter Sequenz:
 - Element in Mitte ansehen
 - falls größer als gesuchtes Element: in Teilsequenz links weiter suchen
 - falls kleiner: in Teilsequenz rechts weiter suchen

find(9)



PriorityQueues:

- M Menge von Elementen
- $\text{key}(e)$ Priorität von Element $e \in M$

Wichtige Operationen:

- $\text{build}(\{e_1, \dots, e_n\})$: $M = \{e_1, \dots, e_n\}$
- $\text{insert}(e)$: $M = M \cup \{e\}$
- $\text{min}()$: gibt Element mit minimaler Priorität zurück
- $\text{deleteMin}()$: entfernt Element mit minimaler Priorität und gibt es zurück

Addressierbare PriorityQueues:

- $\text{insert}()$ gibt Handle auf eingefügtes Element zurück
- $\text{remove}(h)$: löscht Element mit Handle h
- $\text{decreaseKey}(h, k)$: reduziere Priorität d. Elements mit Handle h auf / um k
- $\text{merge}(Q)$: $M = M \cup Q$; $Q = \emptyset$;

Hashing

Assoziative Arrays:

- speichert Elemente
- Element e identifiziert über $\text{key}(e)$

Wichtige Operationen:

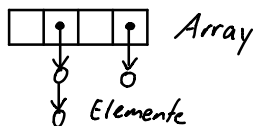
- $\text{insert}(\text{Element } e)$
- $\text{remove}(\text{Key } k)$
- $\text{find}(\text{Key } k)$

Hashing-Idee:

- Array mit Platz für m Elemente
- Universum von möglichen Schlüsseln U
- Hashfunktion $h: U \rightarrow \{0, \dots, m-1\}$ Zuordnung von Schlüssel zu Array-Index
- Kollision: wenn h mehreren Keys denselben Index zuordnet
- n gespeicherte Elemente

Hashing mit Verkettung:

- Kollisionsauflösung durch Verkettung: Array mit Listen



- Platzverbrauch $O(n+m)$
 - ↖ Array-Größe
 - ↖ Anzahl gespeicherter Elemente
- insert: $O(1)$
- remove / find: worst case $O(n)$ (alle Elemente auf selben Index gemappt)

Universelles Hashing:

c - universelle Familien von Hashfunktionen:

- sei $c \in \mathbb{N}$ eine Konstante, H eine Familie von Hashfunktionen $h: U \rightarrow \{0, \dots, m-1\}$

- H heißt c-universell, falls:

$$|\{h \in H \mid h(x) = h(y)\}| \leq \frac{c}{m} \cdot |H| \quad \forall x, y \in U, x \neq y$$

Anzahl der Hashfunktionen,
die Kollisionen für x, y verursachen

- falls H c-universell, gilt: $\Pr[h(x) = h(y)] \leq \frac{c}{m} \quad \forall x \neq y$
- Bei Hashing mit Verkettung mit zufälliger Hashfunktion $h \in H$ (H c-universell) ist die erwartete Laufzeit von remove / find in $O(1 + c \cdot \frac{2}{m})$.

Prinzip:

- Schlüssel sind Bitstrings bestimmter Länge (Annahme)
- Tabellengröße m prim $\Rightarrow \mathbb{Z}_m$ ist Körper, hat genau ein Inverses bzgl. Multiplikation zu jedem Element
- $w := \lfloor \log_2 m \rfloor$
- Bitstring in k Teile zu je w Bits zerstückeln
- jeden dieser Teile als Zahl $x_i \in [0, \dots, 2^w - 1]$ interpretieren
- Schlüssel ist k -Tupel dieser Teile: $x = (x_1, \dots, x_k) \in \{0, \dots, 2^w - 1\}^k$
- $a := (a_1, \dots, a_k) \in \{0, \dots, m-1\}^k$ beliebig
- $\langle a, x \rangle := \sum_{i=1}^k a_i x_i$ Skalarprodukt
- Hashfunktion $h_a(x) := \langle a, x \rangle \bmod m$ definieren
- Beispiel:

32-Bit-Schlüssel, Tabellengröße $269 = m$

$\Rightarrow$ Schlüssel unterteilt zu $w = \lfloor \log_2 269 \rfloor = 8$ Bits, also $k = 4$ Teile

$$32 : 8 = 4$$

$\Rightarrow$ Schlüssel sind 4-Tupel mit Elementen aus $[0, \dots, 2^8-1]$

z.B. $x = (11, 7, 4, 3)$

$\Rightarrow a$ ist 4-Tupel aus $[0, \dots, 268]$

z.B. $a = (2, 4, 261, 16)$

$\Rightarrow$ Hashfunktion $h_a(x) = (2x_1 + 4x_2 + 261x_3 + 16x_4) \bmod 269$

konkret für Beispiel- x :

$$h_a(x) = (2 \cdot 11 + 4 \cdot 7 + 261 \cdot 4 + 16 \cdot 3) \bmod 269 = 66$$

- h_a ist 1-universell, wenn m prim

Perfektes Hashing:

- Annahme: statische Menge S von n Elementen
- Schlüssel $k_1, \dots, k_n$
- H_m : c -universelle Familie von Hashfunktionen auf $\{0, \dots, m-1\}$
- $C(h)$: Anzahl an Kollisionen in S für $h \in H_m$
- Ziel: keine Kollisionen
- Lemma: $\mathbb{E}[C(h)] \leq cn(n-1) \cdot \frac{1}{m}$ $h \in H_m$
- Lemma: $C(h) \leq 2cn(n-1) \cdot \frac{1}{m}$ für mind. die Hälfte der $h \in H_m$
- Lemma: $m \geq cn(n-1) + 1 \Rightarrow$ mind. die Hälfte der $h \in H_m$ bildet die Schlüssel injektiv auf $\{0, \dots, m-1\}$ ab

Strategie:

- Zufälliges $h \in H_m$ wählen, mit $m \geq cn(n-1) + 1$
- h auf Kollisionen prüfen, bei Kollision neu wählen
- nach durchschnittlich 2 Versuchen erfolgreich

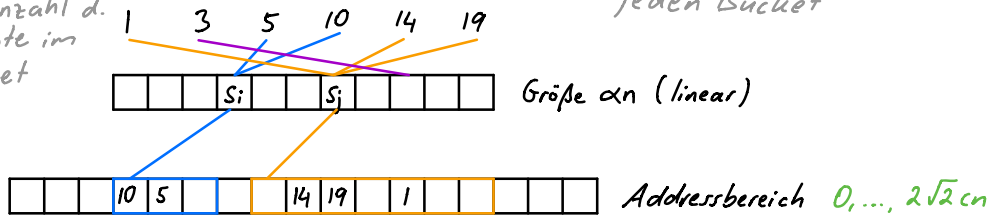
m groß!
quadratisch
bzgl. n

Zwei-stufiges Hashing:

- Ziel: lineare Tabellengröße
- Hashing in 2 Stufen:
 - Hashfunktion h_m mit wenigen Kollisionen wählen (ca. 2 Versuche):
 $m = \lceil \alpha n \rceil$ mit $\alpha = \sqrt{2} c$
 Schlüssel $\rightarrow$ Buckets konstanter durchschnittlicher Größe
 - Hashfunktion h_l ohne Kollisionen wählen (ca. 2 Versuche pro h_l):
 Kollisionsauflösung der Buckets aus Stufe 1

$l = cb(b-1) + 1$ mit
 $b :=$ Anzahl d. Elemente im Bucket

$\hookrightarrow$ quadratisch viel Platz für jeden Bucket

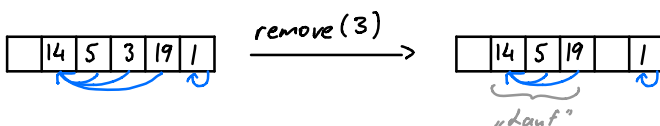


- Satz: Für eine beliebige (aber feste) Menge von n Schlüsseln kann eine perfekte Hashfunktion mit Zielmenge $\{0, \dots, 2\sqrt{2}cn\}$ in linearer erwarteter Laufzeit konstruiert werden.

Hashing mit linearem Sondieren:

Array für Tabelle

- Element e mit $i := h(\text{key}(e))$ im ersten freien Ort $T[i], T[i+1], \dots$
- Vorteile:
 - kein zusätzlicher Speicheraufwand
 - Cache-Effizienz (zusammenhängende Speicherzellen)
- Löschen: Invariante sicher stellen, dass für jedes Element e mit idealer Position $i := h(\text{key}(e))$ und tatsächlicher Position j gilt: $T[i], T[i+1], \dots, T[j]$ besetzt



- $\text{find}(e) : i := h(\text{key}(e))$ berechnen, dann laufen bis e erreicht
- Satz: Wenn n Elemente in einer Hashtabelle T der Größe $m > 2 \cdot n$ mit linearem Sondieren mittels zufälliger Hashfunktion h gespeichert werden, ist für jedes $T[i]$ die erwartete Länge eines **Laufes** in T , der $T[i]$ enthält, konstant.

$\Rightarrow$ erwartete Laufzeit von find / insert / remove in $O(1)$.

- $h(k, i) = (h(k) + i) \bmod m$ ← Tabellengröße
↑ "Versuche" mit Kollisionen

Quadratisches Sondieren:

- $h(k, i) = (h(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m \quad (c_2 \neq 0)$
- möglichst surjektive Abbildung auf $\{0, \dots, m-1\}$ z.B. mit m prim

Double Hashing:

- $h(k, i) = (h(k) + i \cdot h'(k)) \bmod m$
↑ Kollisionsauflösung durch zweite Hashfunktion h'

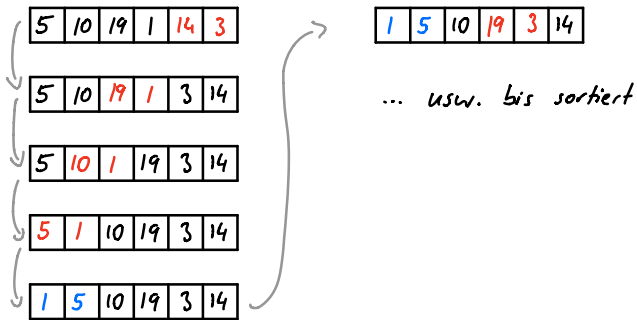
mit $h'(k)$ teilerfremd zu $m \quad \forall k$,

z.B. $h'(k) = 1 + k \bmod m-1 \quad (m \text{ prim})$

Sortieren

BubbleSort: $O(n^2)$

- In jeder Runde bei verbleibender Eingabesequenz je zwei benachbarte Elemente mit falscher Reihenfolge vertauschen:



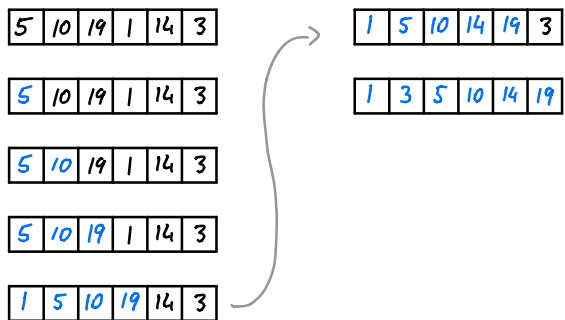
Selection Sort: $\Theta(n^2)$

- Kleinstes Element d. Eingabesequenz $\rightarrow$ Ende d. Ausgabesequenz:



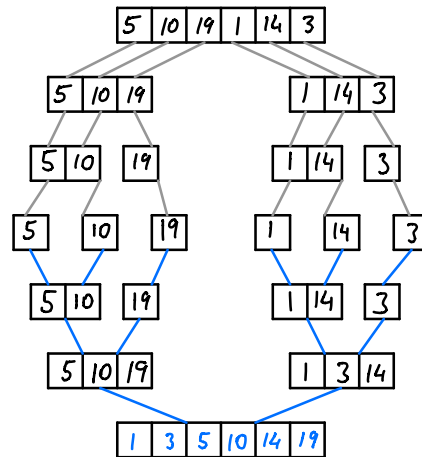
InsertionSort: $O(n^2)$

- Element aus Eingabesequenz richtig in Ausgabesequenz einfügen:



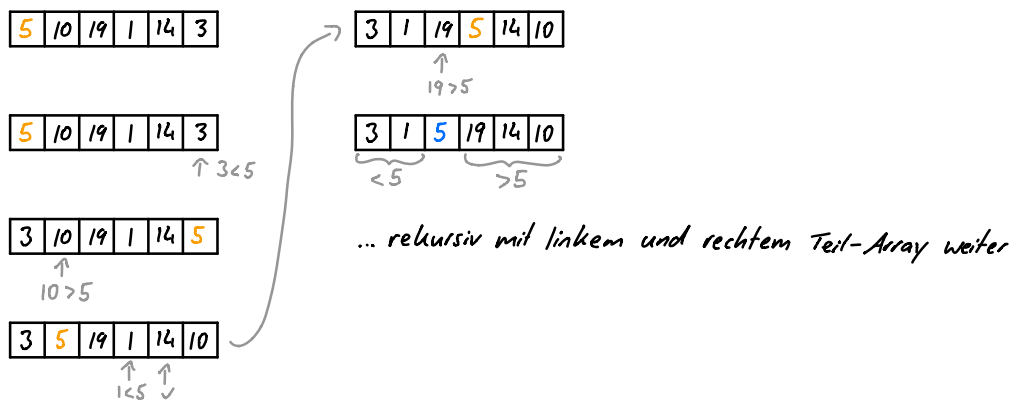
MergeSort: $O(n \log n)$ Zeit, $O(n)$ Speicher für Verschmelzen

- Eingabesequenz rekursiv in Teile zerlegen, diese separat sortieren, und zur Ausgabesequenz verschmelzen:



QuickSort: $O(n^2)$, aber $O(n \log n)$ average case

- **Pivot-Element** wählen
- Aufspaltung in Teilmengen: links kleiner als Pivot, rechts größer
 - z.B. Pivot links: von rechts zu Pivot, falls kleiner: Tauschen, dann von links zu Pivot, und falls größer: Tauschen usw.
 - bzw. umgekehrt
- Teilmengen rekursiv



QuickSelect: $O(n)$ erwartete Laufzeit

Sortieren, und Element
an Index k ausgeben

- k -kleinstes Element schneller finden als in $O(n \log n)$
- Vorgehen wie bei QuickSort, aber immer nur relevantes Teil-Array bzgl.

Pivot-Element betrachten (in jeder Rekursionsstufe):

Falls Index des Pivot-Elements (nach Aufteilung) kleiner als k : rechts weiter,

falls größer: links weiter,

falls gleich: k -kleinstes Element gefunden

RadixSort: $\Theta(d(n+k))$

- Schlüssel Zahlen aus $\{0, \dots, k^d - 1\}$ (d Stellen von Ziffern aus $\{0, \dots, k-1\}$)
- entsprechend niederwertigster Ziffer der jew. Elemente stabil sortieren (gleichwertige Elemente behalten ihre relative Reihenfolge bei), dann selbes für nächst-höherwertige Ziffer, usw.:

012	213	112	001
-----	-----	-----	-----

001	012	112	213
-----	-----	-----	-----

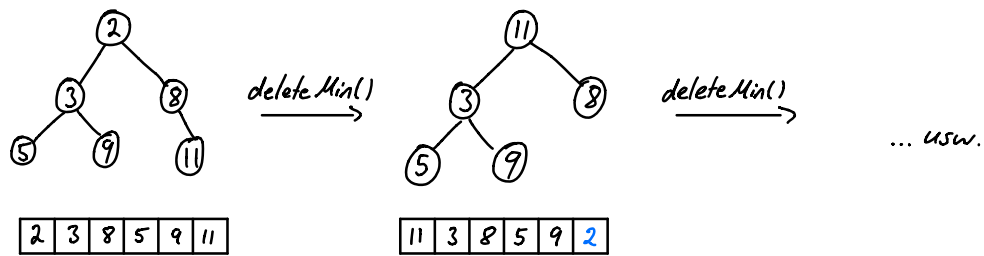
001	012	112	213
-----	-----	-----	-----

001	012	112	213
-----	-----	-----	-----

 fertig

HeapSort: $O(n \log n)$

- binären Min-Heap (nächstes Kapitel) als Array verwenden
- Min-Heap mit `build()` initialisieren $O(n)$
- `deleteMin()` bis Heap leer $O(n \log n)$
- diese gelöschten Elemente landen hinten im Array (was nicht mehr Teil des Min-Heaps ist)



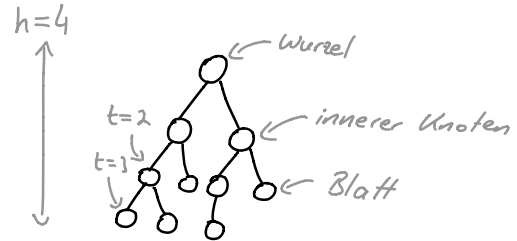
Übersicht:

	Laufzeit	Speicher	Stabil	
BubbleSort	$O(n^2)$	-	✓	
SelectionSort	$O(n^2)$	-	✗	
InsertionSort	$O(n^2)$	-	✓	best case $O(n)$
Merge Sort	$O(n \log n)$	$O(n)$	✓	
QuickSort	$O(n^2)$	-	✗	average case $O(n \log n)$
HeapSort	$O(n \log n)$	-	✗	

Bäume

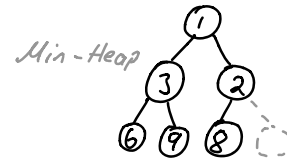
Binary Tree:

- Jeder Knoten maximal 2 Nachfolger
 - Tiefe t eines Knotens: Anzahl Kanten bis zur Wurzel
 - Höhe h : $\max\{t_i\} + 1$
 - Vollständig, falls: alle Blätter selbes t $\wedge$ alle Knoten 0 oder 2 Kinder
-
- Diagram illustrating a binary tree structure with height $h=4$. The root is labeled "Wurzel". The tree shows nodes at levels $t=0$ (root), $t=1$ (internal nodes), and $t=2$ (leaves). The tree is not full because not all nodes have two children.



Binary Heap:

- Binary Tree mit
 - Form-Invariante: alle Ebenen vollständig, unterste Ebene darf unvollständig sein (aber „von links aufgefüllt“)
 - Heap-Invariante: jeder Knoten $\leq$ oder $\geq$ seine Kinder (Min-/Max-Heap)
- 



Wichtige Operationen:

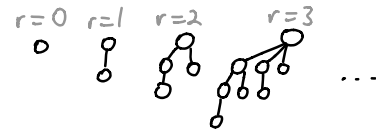
- min: trivial $O(1)$
- deleteMin: $O(\log n)$
 - Wurzel löschen
 - letztes Element $v \rightarrow$ Wurzel
 - siftDown mit v
- siftDown(v): lässt v absinken, bis Heap-Invariante wiederhergestellt $O(\log n)$
- insert: am Ende einfügen, siftUp $O(\log n)$
- siftUp(v): analog siftDown(v) $O(\log n)$

Vergleich anhand Priority Queue:

Priority Queue als ...	unsortierte Liste	sortierte Liste	binary heap
build	$O(n)$	$O(n \log n)$	$O(n)$
insert	$O(1)$	$O(n)$	$O(\log n)$
min	$O(n)$	$O(1)$	$O(1)$
delete Min	$O(n)$	$O(1)$	$O(\log n)$

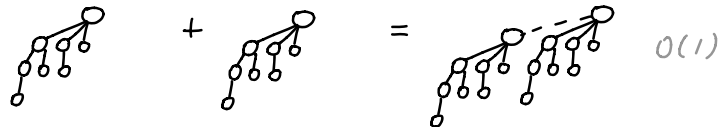
Binomial-Tree:

- Rang $r = 0$: einzelner Knoten
- Binomial-Tree von Rang r hat Nachfolger von Rang $r-1, r-2, \dots, 2, 1, 0$ in dieser Reihenfolge.
- Tiefe $t = r$
- In tiefe l sind $\binom{r}{l}$ Knoten
- Insgesamt 2^r Knoten
- Maximaler Grad r (in Wurzel)



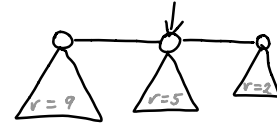
Wichtige Operationen:

- merge: zwei Binomial Trees vom Rang $r-1 \rightarrow$ ein Binomial-Tree vom Rang r



Binomial-Heap:

- Menge von Binomial-Trees, jeder Binomial-Tree erfüllt Min-Heap Eigenschaft, zu jedem Rang r maximal ein Baum
- Pointer auf Wurzel mit minimalem Key
- n Knoten $\Rightarrow \max. \lfloor \log_2 n \rfloor + 1$ Binomial-Bäume



Wichtige Operationen:

- merge: Bäume mit gleichem Rang mergen, andere aufnehmen. $O(\log n)$ mit $n = \max\{n_1, n_2\}$

Entspricht Binär-Addition:

$$\left. \begin{array}{l} n_1 = 5_{10} = 101_2 \text{ Knoten} \\ n_2 = 5_{10} = 101_2 \text{ Knoten} \end{array} \right\} n' = \begin{array}{r} 101_2 \\ + 101_2 \\ \hline 1010_2 \end{array} = 10_{10} \text{ Knoten}$$

- min: trivial $O(1)$
- insert: merge mit Binomial-Tree vom Rang 0 (ein Element) $O(\log n)$
- deleteMin: Min-zeiger-Wurzel löschen $\rightarrow$ Baum zerfällt in Bäume vom Rang $r-1, r-2, \dots, 0 \rightarrow$ merge $O(\log n)$
- decreaseKey: Key von Element ändern $\rightarrow$ siftUp $\rightarrow$ evtl. Min-Pointer aktualisieren $O(\log n)$
- remove: decreaseKey zu $-\infty$, danach deleteMin $O(\log n)$

Vergleich anhand Priority Queue:

Priority Queue als ...	Binomial Heap	Binary Heap
insert	$O(\log n)$	$O(\log n)$
min	$O(1)$	$O(1)$
delete Min	$O(\log n)$	$O(\log n)$
decreaseKey	$O(\log n)$	$O(\log n)$
merge	$O(\log n)$	$O(n)$

Binary Search Tree:

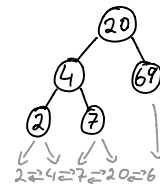
• Binary Tree, für den gilt:

- Jeder Knoten hat einen eindeutigen Key
- Für jeden Knoten x : alles links $\leq x <$ alles rechts
- Schlüssel-Invariante: jeder Schlüssel ist eindeutig
- Grad-Invariante: jeder Knoten hat max. 2 Nachfolger
- Suchbaum-Invariante: für jeden Knoten v mit $k := \text{key}(v)$ gilt:

$$k_l \leq k < k_r \quad \forall k_l \text{ aus linkem Teilbaum, } \forall k_r \text{ aus rechtem Teilbaum}$$

• Entweder ...

- intern: Nutzdaten in inneren Knoten
- oder extern: Baumknoten $\hat{=}$ Navigationsinformationen, Nutzdaten in Blättern (Pointer auf Elemente in sortierter Liste).



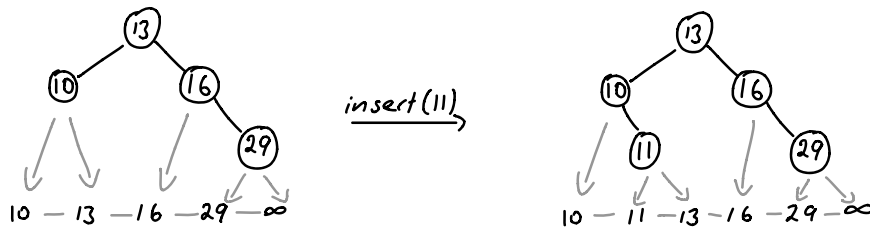
Wichtige Operationen:

• locate(k):

- Starte in Wurzel
- $\forall v: k \leq \text{key}(v)$? gehe zu $v.\text{left}$: gehe zu $v.\text{right}$
- Liefert das Element, oder das nächst-größere

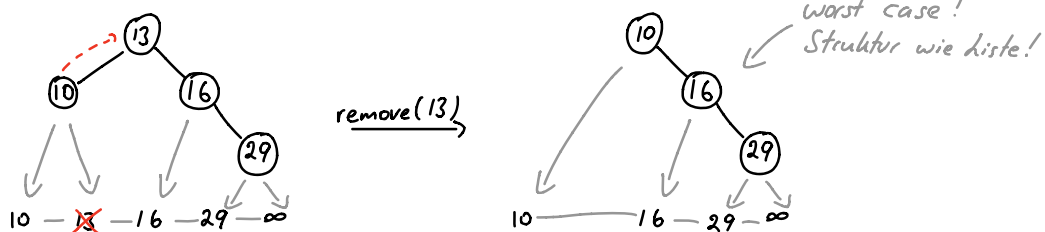
• insert(e):

- Mit locate e' finden
- $\text{key}(e') > \text{key}(e)$? in Liste davor einfügen + neues Blatt : Fehler



• remove(k):

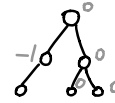
- Mit locate e' in Liste finden
- $\text{key}(e') = k$:
 - e' löschen
 - $v := e'.\text{parent}$ löschen
 - Knoten w mit $\text{key}(w) = k$ bekommt $\text{key}(w) = \text{key}(v)$
- Sonst: Fehler



AVL-Tree:

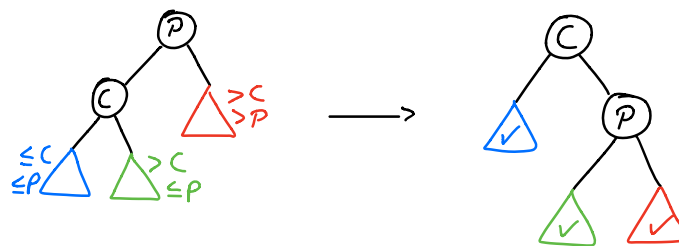
- Binary Search Tree mit AVL-Bedingung:

$$\forall v: v.\text{left.height} - v.\text{right.height} \in \{-1, 0, 1\}$$

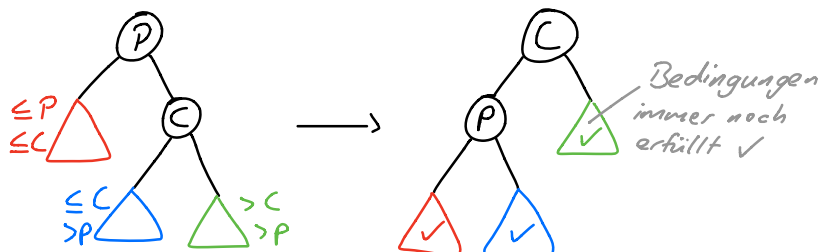


Wichtige Operationen:

- insert: $O(\log n)$
 - Erst insert wie in Binary Search Tree, anhängen an Knoten v
 - Gehe von v rückwärts zur Wurzel, re-kalkuliere Höhendifferenzen
 - Falls Differenz ± 2 : Re-Balancierung
 - Vater & Kind gleiches Vorzeichen:
 - $+$: Links-Rotation
 - $-$: Rechts-Rotation
 - Verschiedene Vorzeichen
 - $+2/-1$: Links-R. am Kind, Rechts-R. am Vater
 - $-2/+1$: Rechts-R. am Kind, Links-R. am Vater
- rotateRight:



- rotateLeft:

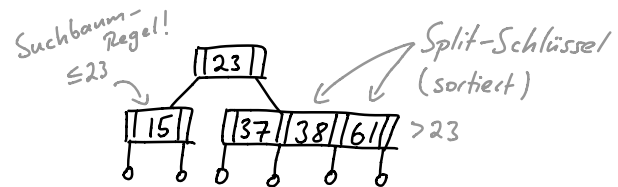


· remove : $O(\log n)$

- Suche zu entfernenden Knoten v
- Falls v Blatt: lösche v
- Falls v genau einen Nachfolger hat: ersetze v durch Kind
- Falls v 2 Nachfolger hat: tausche v mit rechtestem Knoten im linken Unterbaum, lösche v dort
- v rückwärts zur Wurzel: Höhendifferenzen aktualisieren, ggf. rotieren (analog insert)

(a, b) -Trees:

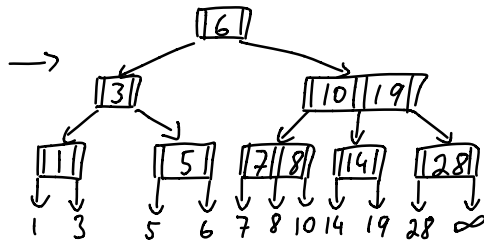
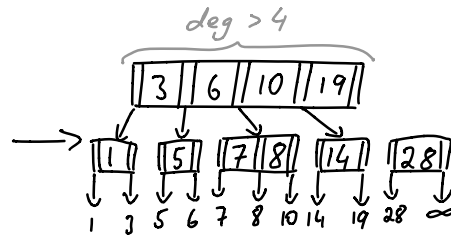
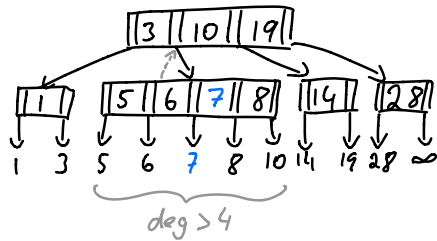
- Externer Suchbaum
- $a \geq 2$, $b \geq 2a - 1$
- Form-Invariante: alle Blätter selbe Tiefe
- Grad-Invariante: Für alle Knoten (außer Wurzel) v : $a \leq \deg(v) \leq b$
- Wurzel w : $2 \leq \deg(w) \leq b$ (außer nur ein Blatt)
- Tiefe $\leq 1 + \lfloor \log_a \frac{n+1}{2} \rfloor$ für $n > 1$ Blätter



Wichtige Operationen:

- locate: wie in jedem Suchbaum $\Theta(\log n)$
- insert(e): $\Theta(\log n)$
 - mit locate Element e' in Liste finden
 - $\text{key}(e) < \text{key}(e')$? vor e' einfügen : Fehler
 - $\text{key}(e)$ in Baumknoten v über e einfügen (mit Handle)
 - $\deg(v) \leq b$? fertig : v aufteilen, größte Key im linken Teil $\rightarrow$ Vaterknoten (usw. bis Wurzel)

(2,4)-Baum: insert (7)



- remove(k): $\Theta(\log n)$
 - mit locate(k) Element e finden
 - e aus Liste entfernen
 - in Baumknoten v über e Schlüssel (und Handle) entfernen
 - $d(v) \geq \alpha$? fertig: Kante stehen
 - Kante stehen:

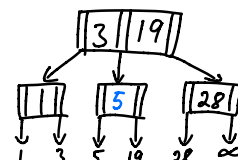
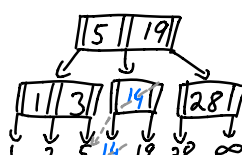
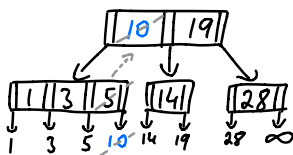
Falls Nachbar v' in gleicher Ebene mit $\deg(v') > \alpha$: von v' stehen,

Sonst: merge v mit Nachbar in gleicher Ebene.

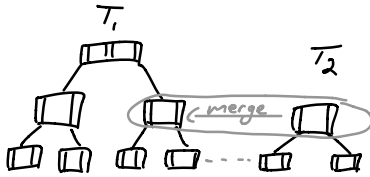
Merge ggf. bis zur Wurzel w . Falls dann $\deg(w) < 2$:

lösche Wurzel

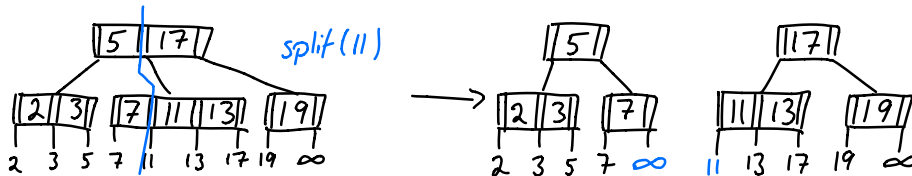
(2,4)-Baum:



- min / max : $O(1)$
über first-/last-Methode der Liste
- range query $[x, y]$: $O(\log n + \text{Ausgabegröße})$
 - locate(x)
 - durchlaufe Liste, bis Element $> y$
- Konkatination: $O(1 + |h_1 - h_2|)$ bzw. $O(1 + \log(\max\{h_1, h_2\}))$ mit Höhenbestimmung
 - Bedingung: Schlüssel in $T_1 \leq$ Schlüssel in T_2
 - ∞ -Dummy in T_1 löschen (ggf. Behandlung wie bei remove)
 - Wurzel des Baums mit niedrigerer Höhe mit äußerstem Knoten in gleicher Ebene des anderen Baumes (Blätter beider Bäume auf gleicher Ebene) verschmelzen (ggf. Behandlung wie bei insert)



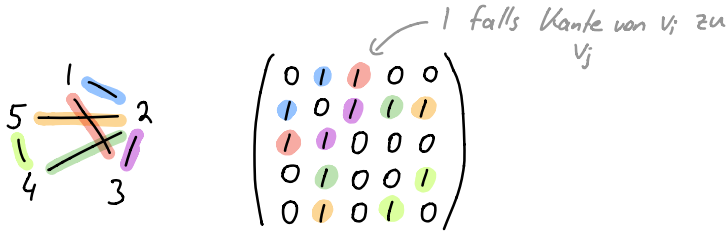
- Aufspaltung am Schlüssel k : $O(\log n)$
 - Pfad von locate(k) speichern
 - jeden Knoten v auf dem Pfad in v_l und v_r aufsplitten (evtl. gibt es dann Knoten ohne Kinder)
 - Knoten mit Kindern werden als Wurzeln interpretiert:
Konkatination linke Bäume (und neues ∞ -Dummy),
Konkatination rechte Bäume (neuer Baum mit Elementen ab k)



Graphen

Graphrepräsentationen: Im Folgenden $n := \text{Anzahl Knoten}$, $m := \text{Anzahl Kanten}$

Adjazenzmatrix:



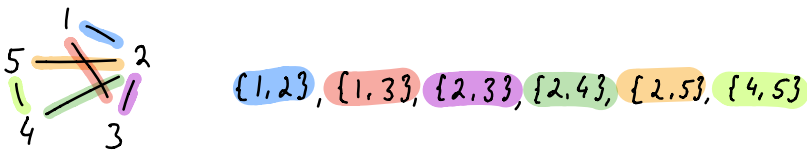
Vorteile:

- Feststellen, ob 2 Knoten benachbart sind in $O(1)$
- Einfügen und Löschen von Kanten in $O(1)$

Nachteile:

- $\Theta(n^2)$ Speicher, auch mit weniger als n Kanten
- Alle Nachbarn eines Knotens finden in $O(n)$
- Neue Matrix für jeden neuen Knoten

Kantenliste:



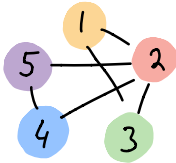
Vorteile:

- $O(m)$ Speicher
- Knoten und Kanten einfügen in $O(1)$
- Löschen von Kanten per Handle in $O(1)$

Nachteile:

- Kante finden/löschen: worst case $\Theta(m)$
- Nachbarn feststellen in $O(m)$

Adjazenzarrays:



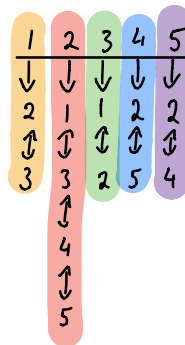
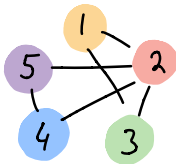
Vorteile:

- Gerichtete Graphen $n+m+\Theta(1)$ Speicher
- Ungerichtete Graphen $n+2m+\Theta(1)$ Speicher

Nachteile:

- Nur für statische Graphen geeignet (Einfügen teuer)

Adjazenzlisten:



doppelt-verkettete Variante

Vorteile:

- Einfügen von Kanten in $O(1)$
- Löschen von Kanten in $O(d)$ (per Handle $O(1)$)
 degree

Nachteile:

- Zeigerstrukturen: Speicher, Zugriffszeit

Adjazenzliste + Hashtabelle:

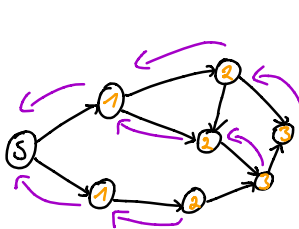
- Hashtabelle bildet 2 Knoten auf Zeiger auf (eventuelle) Kante zwischen den 2 Knoten ab

Vorteile:

- Kante finden / löschen: $O(1)$ worst case
- Kante einfügen: $O(1)$ amortisiert
- Speicheraufwand $O(n+m)$

Breitensuche (BFS):

Zuerst in die Breite gehen (alle Nachbarn abarbeiten), jew. **Distanz** zum Startknoten s vermerken, jew. **parent** vermerken.



↑
"Von welchem Knoten wurde der Knoten entdeckt?"

Kantentypen:

- Baumkanten zum Kind
- Rückwärtskanten zu einem Vorfahren
- Kreuzkanten: sonstige

Anwendungen:

- Single Source Shortest Path (SSSP) in ungewichteten Graphen

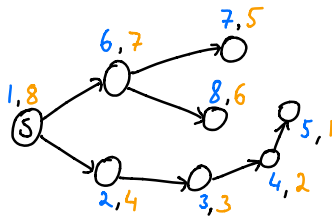
- Bestimmung des nächsten Zugs bei Spielen

Umsetzung:

- Standard-BFS mit FIFO-Queue (ebenenweise Erkundung, teuer!)
- Best-First-Search mit Priority Queue (z.B. bei Spielen: Priorität eines Knotens bestimmt durch Güte-Heuristik des Spielzustands)

Tiefensuche (DFS):

Zuerst in die Tiefe gehen (Rekursions-Stack), Explorationsreihenfolge speichern ($dfsNum$), Fertigstellungsreihenfolge speichern ($finishNum$).



Kantentypen:

- Baumkanten zum Nachbarn
- Vorwärtskanten zu einem Nachfolger
- Rückwärtskanten zu einem Vorfahren
- Kreuzkanten sonstige

Kante (v, w) ist...	$dfsNum[v] < dfsNum[w]$	$finishNum[v] > finishNum[w]$
...Baum- & Vorwärtskante	✓	✓
...Rückwärtskante	✗	✗
...Kreuzkante	✗	✓

Anwendung:

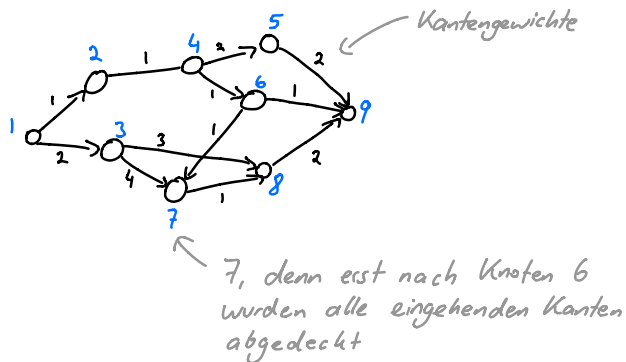
- DAG-Erkennung: Graph ist DAG $\iff$ DFS enthält keine Rückwärtskante
- (Starke) Zusammenhangskomponenten bestimmen

Zusammenhang von Graphen:

- Zusammenhängend $\iff$ Es gibt von jedem Knoten aus einen Pfad zu jedem anderen Knoten
- Stark zusammenhängend $\iff$ Es gibt von jedem Knoten aus einen gerichteten Pfad zu jedem anderen Knoten
- Starke Zusammenhangskomponente: induzierter Teilgraph ist stark zusammenhängend und (inklusions-)maximal

Topologische Sortierung in DAGs: $O(n+m)$ DAG: Directed Acyclic Graph

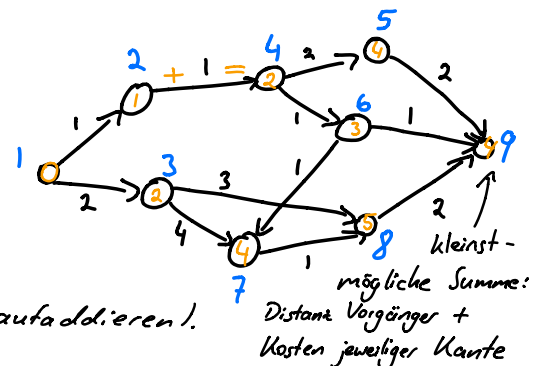
Erst, wenn alle Vorgänger eines Knotens nummeriert wurden, den Knoten nummerieren. Für alle Kanten (v, w) soll gelten: $\text{topoNum}(v) < \text{topoNum}(w)$.



Kürzeste Wege finden: $O(n+m)$

Gemäß Reihenfolge der topologischen Sortierung die Distanzen an den Knoten aktualisieren

(Kantengewichte, die zum Knoten hinführen aufaddieren).

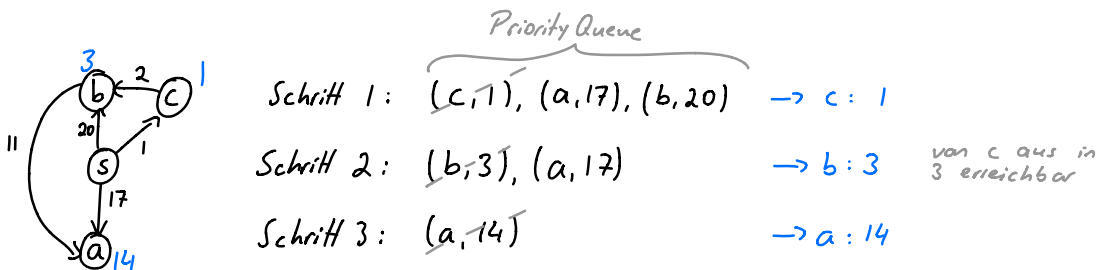


Realisierung:

- Counter für die Anzahl eingehender Kanten jedes Knotens
- FIFO-Queue $O(n+m)$
 - initialisieren mit Knoten ohne eingehende Kanten
 - Knoten v aus Queue nehmen, bei allen Nachfolgern Counter dekrementieren
 - Counter == 0 ? Nachfolger $\rightarrow$ Queue
 - Wiederholen bis Queue leer

Dijkstra - Algorithmus: $O(m + n \log n)$

- Graph gerichtet oder ungerichtet
- Kantengewichte ≥ 0
- SSSP
- Distanz initial ∞ für alle Knoten, 0 für Startknoten: $d[s] = 0$
- Knoten inklusive Distanzen in Priority Queue
- Greedy nächsten Knoten mit kleinster Distanz nehmen, endgültige Distanz setzen
- Alle Nachbarn des Knotens in Priority Queue aufnehmen / aktualisieren



- Problem: kommt nicht mit Negativkreisen klar

Bellman-Ford-Algorithmus: $O(m \cdot n)$

- beliebiger Graph, beliebige Kantengewichte
- SSSP
- $(n-1)$ -Mal alle Kanten durchlaufen, Distanzen Aktualisieren
→ alle kürzesten Wege berücksichtigt
- erneut alle Kanten durchlaufen → falls Distanzverringern: Negativkreis erkannt
→ alle erreichbaren Knoten „infizieren“ (markieren mit $-\infty$).

Floyd-Warshall-Algorithmus: $O(n^3)$

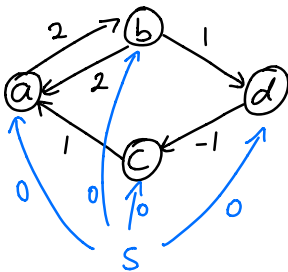
- Graph beliebig
- Negativkreise verfälschen Ergebnis, Negativkanten erlaubt
- All Pairs Shortest Paths (APSP): kürzeste Pfade zwischen allen Knotenpaaren
- Distanzmatrix mit allen kürzesten Pfaden (0 auf Diagonale) existiert
- alle Knoten durchlaufen, und für jeden Knoten:
 - gesamte Distanzmatrix durchlaufen, prüfen ob Weg über aktuellen Knoten kürzer ist als aktueller Weg („Umweg“ schneller als direkter Weg)
 - ggf. Distanzmatrix aktualisieren

Modifizierte Kantenkosten:

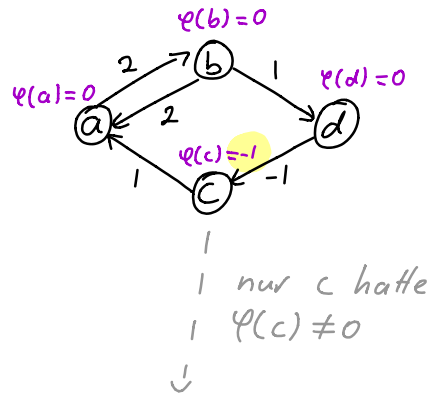
- Ziel: negative Kantenkosten eliminieren
 - Vorgehen:
 - Sei $e = (v, w)$ Kante, $\varphi: V \rightarrow \mathbb{R}$ Abbildung (Zuordnung Knotenpotential)
- cost · $c'(e) = \varphi(v) + c(e) - \varphi(w)$
- Seien p, q Pfade. Es gilt: $c(p) < c(q) \iff c'(p) < c'(q)$

Johnson - Algorithmus: $O(m \cdot n + n^2 \log n)$

- Graph beliebig
- APSP
- Neuen „künstlichen“ Knoten s hinzufügen, mit allen anderen Knoten v verbinden s.d. $c(s, v) = 0 \forall v$
- Knotenpotenziale berechnen mit Bellman-Ford (SSSP von $s \rightarrow v \forall v$),
- modifizierte Kantenkosten c' berechnen (über Knotenpotenziale)
- mit Dijkstra (SSSP) alle Distanzen $d'(v, w)$ (v, w Knoten) mit modifizierten Kantenkosten berechnen (für alle Knoten außer s)
- Distanzen korrigieren: $d(v, w) = d'(v, w) + \varphi(w) - \varphi(v)$



Bellman-Ford
von s aus
liefert Knoten-
potenziale φ



Dijkstra liefert Distanzmatrix d' :

d'	a	b	c	d
a	0	2	3	3
b	1	0	1	1
c	0	2	0	3
d	0	2	0	0

korrigieren $\rightarrow$

d	a	b	c	d
a	0	2	2	3
b	1	0	0	1
c	1	3	0	4
d	0	2	-1	0

Algorithmus von Kruskal: $O(m \log m)$

- Ziel: minimalen Spannbaum finden
- alle Kanten aufsteigend nach Kosten sortieren
- alle Kanten $e = \{u, v\}$ durchgehen
 - falls u und v in versch. Bäumen: $\leftarrow$ ^{Union - Find} if (find(u) $\neq$ find(v))...
Spannbaum $\cup e$ und union(u, v)

Union - Find:

```
int find(int i){  
    if (parents[i] == i) return i;  
    int k = find(parents[i]);  
    parents[i] = k;  
    return k;  
}  
  
void union(int i, int j) {  
    int r_i = find(i);  
    int r_j = find(j);  
    if (r_i != r_j) parents[r_i] = r_j;  
}
```

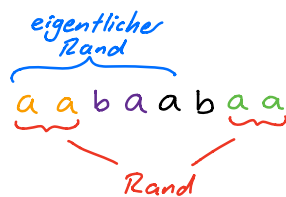
Algorithmus von Jarník-Prim: $O(m + n \log n)$

- Ziel: minimalen Spannbaum finden
- Greedy-Vorgehen wie bei Dijkstra
- PriorityQueue mit allen Knoten abarbeiten, alle Knoten v anfangs Distanz ∞ (außer Startknoten)
- Rest wie Dijkstra

Pattern Matching

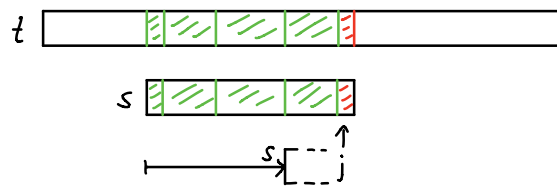
Pattern-Matching:

- Σ : Alphabet
- w : Wort $(w_0, \dots, w_{n-1})$ s.d. $\forall i \in \{0, \dots, n-1\}: w_i \in \Sigma$
- $|w|$: Länge des Worts (Anzahl Symbole)
- ϵ : leeres Wort
- Σ^* : $\{w \mid w \in \Sigma\}$
- Σ^+ : $\Sigma^* \setminus \{\epsilon\}$
- Σ^k : $\{w \in \Sigma \mid |w| = k\}$
- w' **Präfix** von w , wenn w mit w' beginnt
- w' **Suffix** von w , wenn w mit w' endet
- w' **Infix** von w , wenn w' in w vorkommt, aber nicht Präfix / Suffix ist
- r heißt **Rand** von w , wenn r Präfix und Suffix ist
- r heißt **eigentlicher Rand** von w , wenn $r \neq w$ und es sonst keinen längeren Rand gibt



Knuth-Morris-Pratt-Algorithmus: $O(n)$

- Pattern s in String t finden
- Präfix des Patterns wird mit t verglichen bis Mismatch an Stelle j
- Pattern um $j - (\text{Länge des Präfix})$ weiter „shiften“:
 $\nwarrow$
 eigentlicher Rand



- Border - Tabelle für Länge des eigentlichen Rands $|r|$ der jeweiligen Präfixe