Zusammenfassung

Datenstrukturen & Algorithmen

M. Leonard Haufs

Sommersemester 2012

Wichtige Hinweise:

Letzte Bearbeitung: Sonntag, 9. Dezember 2012

Ihr dürft die Zusammenfassung selbstverständlich gerne nutzen.
Ich habe nur eine Bedingung. Wenn ihr Fehler im Dokument findet, schreibt sie mir bitte an folgende Adresse, damit ich die Zusammenfassung verbessern kann:
martin.leonard.haufs@rwth-aachen.de

Viele der Abbildungen dieser Zusammenfassung sind folgender Vorlesung entnommen:

Datenstrukturen und Algorithmen
Prof. Dr. Joost-Pieter Katoen
(*RWTH Aachen*, Sommersemester 2012)

Inhaltsverzeichnis

1 Komplexitätsklassen **1**

2 Rekursionsgleichungen **2**
1 Rekursionsbäume . . . 2
2 Substitutionsmethode . . . 3
3 Mastertheorem . . . 4

3 Datenstrukturen **5**
1 Listen . . . 5
2 Bäume . . . 6
2.1 Binärbäume . . . 6
2.2 Binäre Suchbäume . . . 7
2.3 Rot-Schwarz-Bäume . . . 9
3 Graphen . . . 11

4 Flüsse **15**
1 Allgemeine Flusseigenschaften . . . 15
1.1 Flussnetzwerke . . . 15
1.2 Fluss in einem Flussnetzwerk . . . 15
1.3 Flüsse zwischen Knotenmengen . . . 16
2 Ford-Fulkerson-Methode . . . 16

5 Algorithmenprinzipien **17**
1 Divide & conquer - Teilen und Beherrschen . . . 17
2 Greedy-Algorithmen . . . 17
3 Dynamische Programmierung . . . 17
3.1 Rucksackproblem . . . 18
3.2 Longest Common Subsequence (LCS) . . . 19

6 Sortieralgorithmen **21**
1 Einfache Sortieralgorithmen . . . 21
1.1 Insertionsort - Sortieren durch einfügen . . . 21
1.2 Selectionsort . . . 22

2 Höhere Sortieralgorithmen . . . 22
2.1 Heapsort . . . 22
2.2 Countingsort . . . 24
2.3 Mergesort . . . 25
2.4 Quicksort . . . 26
2.5 Dutch National Flag- Problem . . . 27
3 Gesamtübersicht über Sortieralgorithmen . . . 29

7 Graphenalgorithmen 30
1 Graphensuche . . . 30
1.1 Tiefensuche . . . 30
1.2 Breitensuche . . . 32
2 Sharir's Algorithmus . . . 33
3 Prims Algorithmus . . . 34
4 Single Source Shortest Path (SSSP) . . . 35
4.1 Dijkstra-Algorithmus . . . 35
4.2 Bellman-Ford-Algorithmus . . . 36
5 Topologische Sortierung . . . 37
6 All Pairs Shortest Paths . . . 39
6.1 Floyd-Warshall Algorithmus . . . 39
7 Algorithmen auf Flussnetzwerken . . . 40
7.1 Ford-Fulkerson-Methode - Maximaler Fluss . . . 40
7.2 Min-cut-max-flow . . . 41

8 Hashing 42
1 Hashfunktionen . . . 42
2 Geschlossenes Hashing . . . 43
3 Offenes Hashing . . . 44

9 Algorithmische Geometrie 47
1 Mathematische Grundlagen . . . 47
2 Graham-Scan . . . 48

1 Komplexitätsklassen

$O(f)$

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

Alternative Definition:

$$g \in O(f) \Leftrightarrow \limsup_{n \to \infty} \frac{g(n)}{f(n)} = c \geq 0, c \neq \infty$$

$\Omega(f)$

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

Alternative Definition:

$$g \in \Omega(f) \Leftrightarrow \liminf_{n \to \infty} \frac{g(n)}{f(n)} = c > 0$$

$\Theta(f)$

$$g \in \Theta(f) \Leftrightarrow \exists c_1, c_2 > 0, n_0 : \forall n \geq n_0 : c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)$$

Alternative Definition:

$$g \in \Theta(f) \Leftrightarrow \lim_{n \to \infty} \frac{g(n)}{f(n)} = c, 0 < c < \infty$$

$o(f)$

$$g \in o(f) \Leftrightarrow \forall c > 0, n_0 : \forall n \geq n_0 : 0 \leq g(n) \leq c < f(n)$$

$\omega(f)$

$$g \in \omega(f) \Leftrightarrow \forall c > 0, n_0 : \forall n \geq n_0 : f(n) < g(n)$$

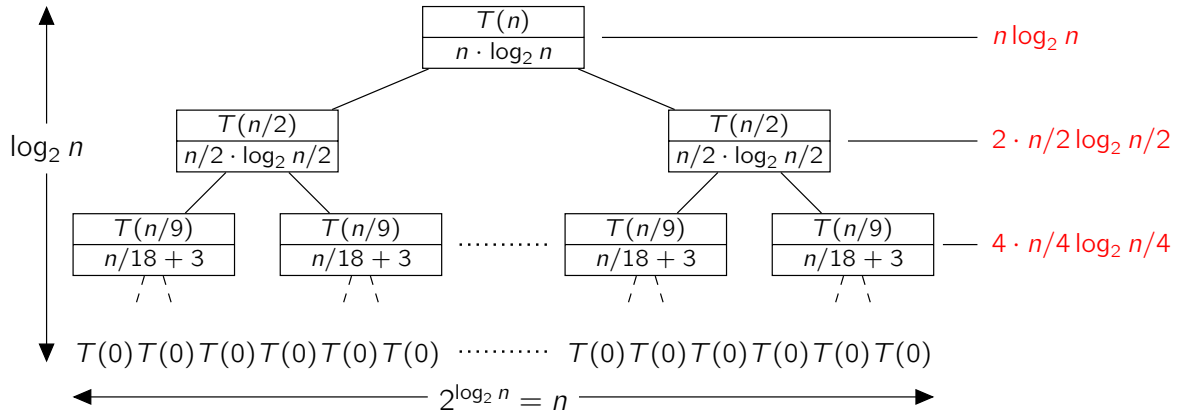
2 Rekursionsgleichungen

1 Rekursionsbäume

Allgemeines Prinzip:

Mithilfe von Rekursionsbäumen lässt sich die Lösung einer Rekursionsgleichung erraten, indem die Aufteilung in Teilprobleme iterativ dargestellt wird.

Beispiel: $T(n) = 2T(\frac{n}{2}) + n \cdot \log_2 n, T(0) = 1$



$$T(n) = \sum_{i=0}^{\log_2 n} 2^i \cdot \frac{n}{2^i} \cdot log_2\left(\frac{n}{2^i}\right) + 2^{log_2 n+1} = \sum_{i=0}^{\log_2 n} 2^i \cdot log_2\left(\frac{n}{2^i}\right) + 2 \cdot n$$

Die Rekursionsgleichung leitet sich wie folgt her:

$$\sum_{i=0}^{h} (\text{Rekursive Kosten auf Ebene i}) + \text{Gesamtkosten der Blätter}(= c \cdot b^{h+1})$$

mit Verästelungsgrad b.

Bei der Bestimmung der Höhe h ist folgendes zu beachten:

- Ist der Basisfall T(0), so ist die Höhe um 1 erhöht, da zusätzlich die Elemente der Ebene T(0) betrachtet werden.

Typische Höhen:

- $T(n) = b \cdot T(\frac{n}{c})$
  $\rightarrow\ h = log_c n$
- $T(n) = b \cdot T(n-c), c \in \{1, 2\}$
  $\rightarrow\ h = \frac{n}{c}$
- $T(n) = b \cdot T(\sqrt{n})$
  $\rightarrow\ h = \log_2 \log_2 n$

2 Substitutionsmethode

Allgemeines Vorgehen:

- Lösung 'raten' (z.B. Rekursionsbaum)
- Betrachte die jeweilige Definition des betrachteten Falls (O, Ω,, Θ, o, ω)
- Setze die geratene Lösung in den rekursiven Teil ($T(n)$) der Rekursionsgleichung ein und beweise entsprechend der Definition des betrachteten Falls.
- Teile durch Umformen den Term in 2 Teile auf:
  - Einem Teil, der einer Konstanten multipliziert mit der beweisenden Schranke entspricht und
  - Einem Restterm, durch Abschätzen über die Konstante die Definition des Falls bestärkt.
- Gegebenenfalls muss durch Variablentransformation die Rekursionsgleichung zunächst vereinfacht werden.
  - Häufigste Transformationen:
    - $n = 2^m \Leftrightarrow m = log_2 n$
    - $T(2^m) = S(m)$

  Beispiel:
  Bestimme die exakte Lösung der Rekursionsgleichung

  $$T(n) = \sqrt[3]{n} + 3 \text{ mit } T(2) = 2.$$

$$
\begin{aligned}
& T(n) = T(\sqrt[3]{n}) + 3 && \text{|Variablentransformation } m = \log_2 n \\
\Leftrightarrow\quad & T(2^m) = T(2^{\frac{m}{3}}) + 3 && \text{| Umbenennung } T(2^m) = S(m) \\
\Leftrightarrow\quad & S(m) = S(\frac{m}{3}) + 3 && \text{| Lösung bekannter Rekursionsgleichung: } S(m) = S(\frac{m}{3}) + 3 \\
\Leftrightarrow\quad & S(m) = 3 \cdot \log_3 m + S(1) && \text{| da } S(1) = T(2) = 2 \text{ (Anfangsbedingung)} \\
\Leftrightarrow\quad & S(m) = 3 \cdot \log_3 m + 2 && \text{| } m = \log_2 n \\
\Leftrightarrow\quad & T(n) = 3 \cdot \log_3 \log_2 n + 2
\end{aligned}
$$

3 Mastertheorem

$$T(n) = b \cdot T(\frac{n}{c}) + f(n)$$

mit $b \leq 1$ und $c > 1$

Anzahl der Blätter im Rekursionsbaum:

$$n^E \text{ mit } E = \tfrac{log(b)}{log(c)} = \log_c b$$

1. $f(n) \in O(n^{E-\epsilon})$ (für ein $\epsilon > 0$)
   $\rightarrow\ T(n) \in \Theta(n^E)$
2. $f(n) \in \Theta(n^E)$
   $\rightarrow\ T(n) \in \Theta(n^E \cdot log(n))$
3. $f(n) \in \Omega(n^{E+\epsilon})$ (für ein $\epsilon > 0$) *und*
   $b \cdot f(\frac{n}{c}) \leq d \cdot f(n)$ (für $d < 1$, n hinreichend groß)
   $\rightarrow\ T(n) \in \Theta(f(n))$

Grenzen des Mastertheorems:

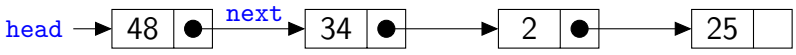
Das Mastertheoren ist *nicht* anwendbar, wenn

- die Funktion $f(n)$ negativ ist.
- b oder c $\notin \mathbb{N}^{>0}$.
- der Unterschied von n^E und $f(n)$ nicht *polynomiell* ist.

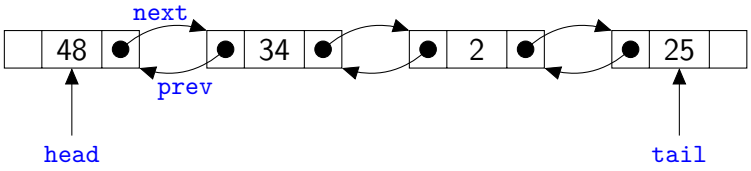
3 Datenstrukturen

1 Listen

Einfach verkettete Listen



Doppelt verkettete Listen



Laufzeiten

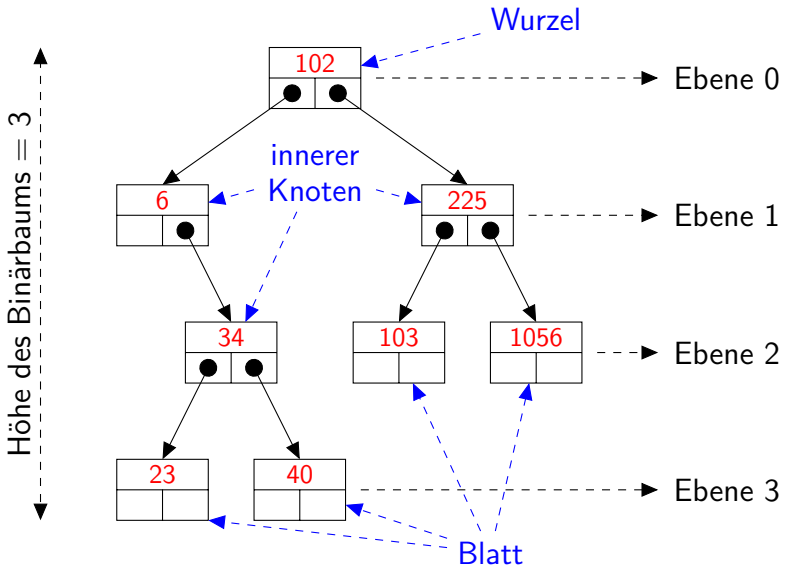
	Implementierung	
Operation	einfach verkette Liste	doppelt verkettete Liste
`insert(L,x)`	$\Theta(1)$	$\Theta(1)$
`remove(L,x)`	$\Theta(n)$	$\Theta(1)$
`search(L,k)`	$\Theta(n)$	$\Theta(n)$
`minimum(L)`	$\Theta(n)$	$\Theta(n)$
`maximum(L)`	$\Theta(n)$	$\Theta(n)$
`search(L,k)`	$\Theta(n)$	$\Theta(n)$
`minimum(L)`	$\Theta(n)$	$\Theta(n)$
`maximum(L)`	$\Theta(n)$	$\Theta(n)$
`successor(L,x)`	$\Theta(1)$	$\Theta(1)$
`predecessor(L,x)`	$\Theta(n)$	$\Theta(1)$

2 Bäume

2.1 Binärbäume

Allgemeiner Aufbau

Ein Binärbaum ist ein gerichteter, zykelfreier Graph mit Knoten (nodes, allgemein: vertices) V und gerichteten Kanten (edges).



Begriffsklärung Binärbäume

- Es gibt *genau* einen ausgezeichneten Knoten, die *Wurzel* (root).
- Alle Kanten zeigen von der Wurzel weg.
- Diese Kanten sind Zeiger. Jedes Element bekommt zwei dieser Zeiger (left und right) zu den nachfolgenden Elementen.
- Der Ausgangsgrad jedes Knotens ist höchstens 2.
- Der Eingangsgrad eines Knoten ist 1, bzw. 0 (bei der Wurzel).
  - Sonderfall: Baum mit $V = E = \emptyset$.
- Ein Knoten mit *leerem* linken und rechten Teilbaum heißt Blatt (leaf).
- Die *Tiefe* (auch: Ebene) eines Knotens ist sein Abstand, d. h. die Pfadlänge, von der Wurzel.
- Die *Höhe* eines Baumes ist die maximale Tiefe seiner Blätter.

Allgemeine Eigenschaften

- Ebene d hat höchstens 2^d Knoten.
- Mit Höhe h kann ein Binärbaum *maximal* $2^{h+1} - 1$ Knoten enthalten.
- Mit n Knoten hat ein Binärbaum mindestens die Höhe $h = \lceil \log_2 n + 1 \rceil - 1$

Traversierungen

Eine Traversierung ist ein Baumdurchlauf mit folgenden Eigenschaften:

1. Die Traversierung beginnt und endet an der Wurzel.
2. Die Traversierung folgt den Kanten des Baumes. Jede Kante wird genau zweimal durchlaufen: Einmal von oben nach unten und danach von unten nach oben.
3. Die Teilbäume eines Knotens werden in festgelegter Reihenfolge (zuerst linker, dann rechter Teilbaum) besucht.
4. Unterschiede bestehen darin, bei welchem Durchlauf man den Knoten selbst (bzw. das dort gespeicherte Element) „besucht“.

   (Die Arraydarstellung einer Traversierung nennt man Linearisierung)

Es gibt 3 verschiedene Formen der Traversierung:

- **Inorder-Traversierung:** “Scannen von links nach rechts”
  Zuerst linken Teilbaum, dann Wurzel, dann rechten Teilbaum ausgeben.
- **Preorder-Traversierung:** “Wie Einfügereihenfolge”
  Zuerst Wurzel, dann linken, dann rechten Teilbaum ausgeben.
- **Postorder:** “Soweit wie möglich runter”
  Zuerst linken, dann rechten Teilbaum, dann Wurzel ausgeben.

2.2 Binäre Suchbäume

Allgemeine Definition

Ein binärer Suchbaum (BST) ist ein Binärbaum, der Elemente mit Schlüsseln enthält, wobei der Schlüssel jedes Knotens

- mindestens so groß ist, wie jeder Schlüssel im linken Teilbaum und
- höchstens so groß ist, wie jeder Schlüssel im rechten Teilbaum.

Führt man auf einem binären Suchbaum eine *Inordertraversierung* durch, so erhält man mit einer Laufzeit von $\Theta(n)$ die Schlüssel als Linearisierung in sortierter Reihenfolge.

Operationen auf Binären Suchbäumen

Suchen

Traversiere (durchsuche) den Baum:

- Falls der gesuchte Schlüssel *gleich* dem aktuell betrachteten Schlüssel ist: Suche beenden
- Falls der gesuchte Schlüssel *kleiner* als der aktuell betrachtete Schlüssel ist: Durchlaufe den linken Teilbaum (linkes Kind ist neuer betrachtete Schlüssel)
- Falls der gesuchte Schlüssel *größer* als der aktuell betrachtete Schlüssel ist: Durchlaufe den rechten Teilbaum (rechtes Kind ist neuer betrachtete Schlüssel)

Komplexität: $\Theta(h)$

Einfügen

1. Suche einen geeigneten freien Platz:

   Wie bei der regulären Suche, außer dass, selbst bei gefundenem Schlüssel, weiter abgestiegen wird, bis ein Knoten ohne entsprechendes Kind erreicht ist.

2. Hänge den neuen Knoten an:

   Verbinde den neuen Knoten mit dem gefundenen Vaterknoten.

Komplexität: $\Theta(h)$ (wegen der notwendigen Suche)

Nachfolger finden

Es gibt 2 mögliche Fälle:

1. Der rechte Teilbaum existiert:
   $\rightarrow$ Der Nachfolger ist der linkeste Knoten (das Minimum) im rechten Teilbaum.
2. Der rechte Teilbaum existiert *nicht*
   $\rightarrow$ Der Nachfolger ist der jüngste Vorfahre, dessen *linker* Teilbaum den Ausgangsknoten enthält.

Komplexität: $\Theta(h)$

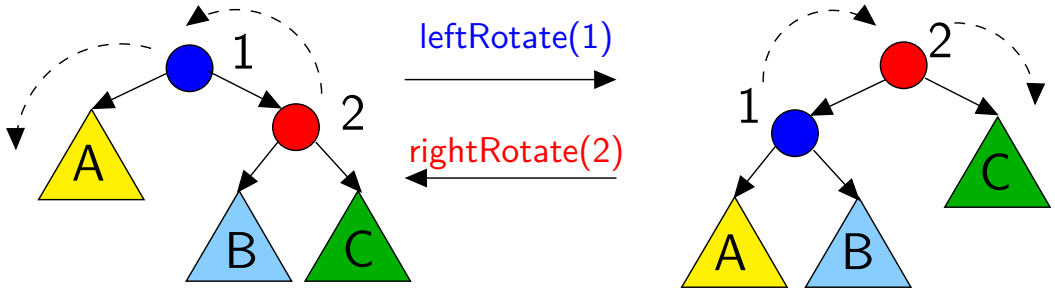
Löschen

Drei mögliche Fälle:

1. Knoten hat *keine* Kinder: Lösche den Knoten
2. Knoten hat *ein* Kind: Schneide den Knoten aus. Verbinde also den Vater- und den Kindknoten direkt miteinander.
3. Knoten hat *zwei* Kinder:
   - Finde den Nachfolger.
     - Lösche den Nachfolger aus dem Baum.
   - Ersetze den Knoten durch dessen Nachfolger.

Komplexität: $\Theta(h)$

Links- und Rechtsrotationen



Komplexität: $\Theta(1)$

2.3 Rot-Schwarz-Bäume

Allgemeine Eigenschaften

Ein binärer Suchbaum, dessen Knoten jeweils zusätzlich eine Farbe haben, hat die Rot-Schwarz-Eigenschaft, falls folgende Bedingungen erfüllt sind:

1. Jeder Knoten ist entweder rot oder schwarz.
2. Die Wurzel ist schwarz.
3. Jedes (externe) Blatt ist schwarz.
4. Ein roter Knoten hat nur schwarze Kinder.
5. Für jeden Knoten enthalten alle Pfade, die an diesem Knoten starten und in einem Blatt des Teilbaumes dieses Knotens enden, die gleiche Anzahl schwarzer Knoten.

Die Schwarzhöhe bh(x) eines Knotens x ist die Anzahl schwarzer Knoten bis zu einem (externen) Blatt, x ausgenommen. Die Schwarzhöhe bh(t) eines RBT t ist die Schwarz-Höhe seiner Wurzel.

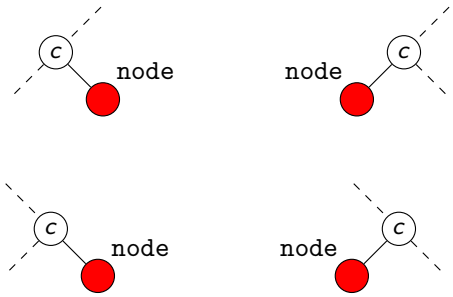
Elementare Eigenschaften

Ein Rot-Schwarzbaum t mit Schwarzhöhe $h = bh(t)$ hat:

- *Mindestens* $2h - 1$ innere Knoten.
- *Höchstens* $4h - 1$ innere Knoten.
- Ein Rot-Schwarz-Baum mit n inneren Knoten hat höchstens die Höhe $2 \cdot log(n+1)$.

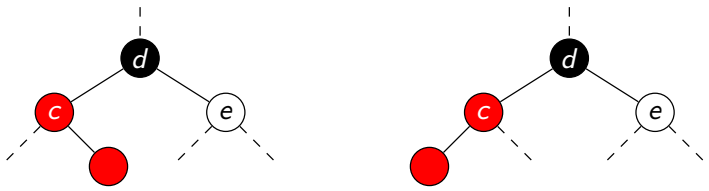
Operationen auf Rot-Schwarzbäumen

Einfügen



- Jeder neu eingefügte Knoten ist *rot*.
- Ist der Vaterknoten c **schwarz**, ist der Einfügevorgang abgeschlossen.
- Ist der Vaterknoten c *rot*, so muss die *Rot-Rot-Verletzung* behoben werden. Es gibt 3 mögliche Fälle:

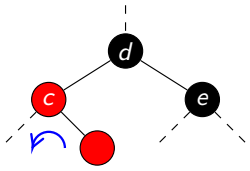
  Fall 1: Onkelknoten e ist rot



  - Umfärben von *c und e* auf **schwarz** und *d* auf **rot**

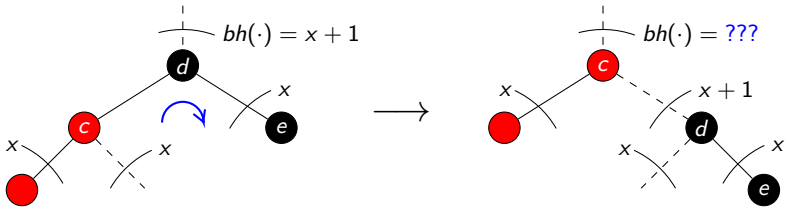
- Löse evtl. höher liegende Rot-Rot-Verletzung iterativ auf.
- ist d die Wurzel, so wird d wieder **schwarz** gefärbt.

Fall 2: Onkelknoten e ist schwarz, neuer Knoten liegt innen



- Überführe zu *Fall 3* durch Rotation um Vater c nach außen.

Fall 3: Onkelknoten e, neuer Knoten liegt außen



- Rotation in Richtung e um Großvaterknoten d.
- d **rot** färben.
- c **schwarz** färben.

Komplexität: $\Theta(\log n)$

3 Graphen

Definitionen

Ein gerichteter Graph G ist ein Paar (V, E) mit

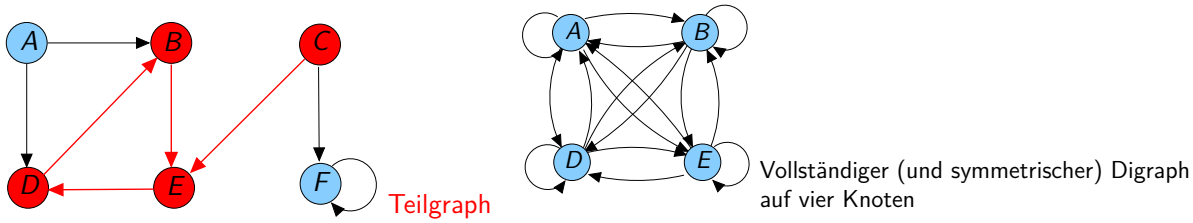
- einer Menge Knoten (vertices) V und
- einer Menge (geordneter) Paare von Knoten $E \subseteq VxV$, die (gerichtete) Kanten (edges) genannt werden.
- Falls E eine Menge ungeordneter Paare ist, heißt G ungerichtet.

Ein Teilgraph (subgraph) eines Graphen G = (V , E) ist ein Graph $G' = (V', E')$ mit:

- $V' \subseteq V$ und $E' \subseteq E$.

– Ist $V' \subset V$ und $E' \subset E$, so heißt G' *echter Teilgraph*.

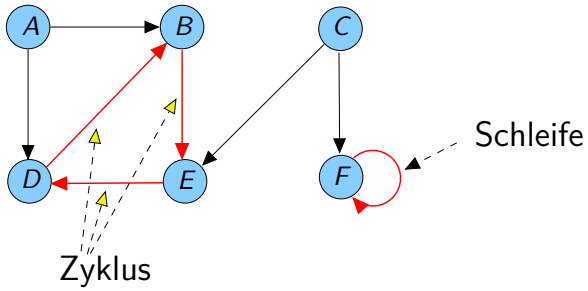
- **Transponierter Graph:** In G^T ist die Richtung der Kanten von G umgedreht.



Terminologie bei Graphen

Grapheigenschaften

- Der Graph G heißt *symmetrisch*, wenn aus (v, w) ∈ E folgt (w, v) ∈ E.
- Der Graph G ist *vollständig*, wenn jedes Paar von Knoten mit einer Kante verbunden ist.
- Knoten w ist *adjazent* zu Knoten v, wenn $(v, w) \in E$.
- *Transponiert* man G, so erhält man $G^T = (V, E')$ mit $(v, w) \in E'$ gdw. $(w, v) \in E$.
- Ein *Weg* von Knoten v nach w ist eine Folge von Kanten $(v_i, v_{i+1}) : v_0 v_1 v_2 ... v_{k-1} v_k$, so dass $v_0 = v$ und $v_k = w$
- Ein *Weg*, bei dem alle Kanten verschieden sind, heißt *Pfad*.
- *Länge* eines Pfades (Weges) ist die Anzahl der durchlaufenen Kanten.
- Knoten w heißt *erreichbar* von v, wenn es einen Pfad von v nach w gibt.
- Ein *Zyklus* ist ein nicht-leerer Weg bei dem der Startknoten auch Endknoten ist.
  - Ein Zyklus der Form *vv* heißt *Schleife* (loop, self-cycle).
  - Ein Graph ist *azyklisch*, wenn er keine Zyklen hat.



Beispiele für Wege: A B E D B und C F F, Beispiele für Pfade: E D B und C F

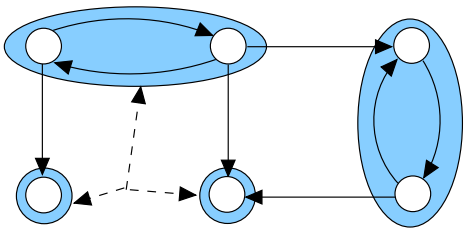
Für ungerichtete Graphen G gilt:

- Ein Graph G heißt *zusammenhängend*, wenn jeder Knoten von jedem anderen Knoten aus erreichbar ist.
- Eine *Zusammenhangskomponente* von G ist ein maximaler zusammenhängender Teilgraph von G.
- In einer Ansammlung von Graphen heißt ein Graph *maximal*, wenn er von keinem anderen dieser Graphen ein echter Teilgraph ist.

Für gerichtete Graphen G gilt:

- G heißt *stark zusammenhängend*, wenn jeder Knoten von jedem anderen aus erreichbar ist.
- G heißt *schwach zusammenhängend*, wenn der zugehörige ungerichtete Graph (wenn alle Kanten ungerichtet gemacht worden sind) zusammenhängend ist.
- Eine *starke Zusammenhangskomponente* von G ist ein maximaler stark zusammenhängender Teilgraph von G.

Ein nicht verbundener Graph kann eindeutig in verschiedene Zusammenhangskomponenten aufgeteilt werden.



Zusammenhangskomponenten

Darstellung als Adjazenzmatrix und Adjazenzliste

Die Adjazenzmatrix-Darstellung eines Graphen ist durch eine nxn Matrix A gegeben, wobei $A(i,j) = 1$, wenn $(v_i, v_j) \in E$, sonst 0.

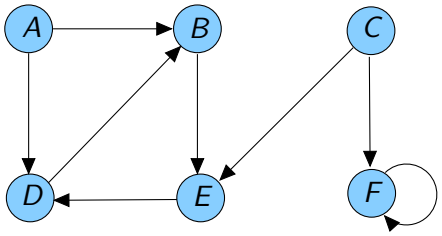
- Wenn G ungerichtet ist, ergibt sich eine symmetrische Adjazenzmatrix A (d.h. $A = A^T$). Dann muss nur die Hälfte der Datenmenge gespeichert werden.

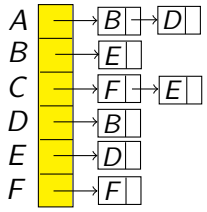
$\rightarrow$ Platzbedarf: $\Theta(n^2)$

Bei der Darstellung als Array von Adjazenzlisten gibt es ein durch die Nummer des Knoten indiziertes Array, das jeweils verkettete Listen (Adjazenzlisten) enthält.

- Der i-te Arrayeintrag enthält alle Kanten von G, die von v_i „ausgehen“.
- Ist G ungerichtet, dann werden Kanten doppelt gespeichert.
- Kanten, die in G nicht vorkommen, benötigen keinen Speicherplatz.

  $\rightarrow$ Platzbedarf: $\Theta(n + m)$





Adjazenzliste

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Adjazenzmatrix

Abbildung 3.1: Beispiel für die Darstellung als Adjazenzmatrix und Adjazenzliste

4 Flüsse

1 Allgemeine Flusseigenschaften

1.1 Flussnetzwerke

Ein Flussnetzwerk ist ein gerichteter Graph, dessen Kanten die Zusatzinformation der Kapazität tragen (Kanten sind wie Wasserrohre).$s, t \in V$ (Quelle s, Senke t) sind ausgezeichnete Knoten des Netzwerkes.

- An der Quelle wird produziert
- An der Senke wird verbraucht
- Kapazität = maximale Durchsatzrate

1.2 Fluss in einem Flussnetzwerk

Ein Fluss f hat folgende Eigenschaften:

Beschränkung: Für $v \in V$ gilt: $f(u, v) \leq c(u, v)$.

Ein Fluss f ist also immer maximal so groß wie die Kapazität c.

Asymmetrie: Für $v \in V$ gilt: $f(u, v) = -f(v, u)$

Wird die Flussrichtung umgekehrt, so ändert sich das Vorzeichen des Flusses.

Flusserhaltung: Für $u \in V - \{s, t\}$ gilt: $\Sigma_{v \in V} f(u, v) = 0$

Was in einen inneren Knoten, der weder Quelle noch Senke ist, hineinfließt kommt auch wieder heraus.

- f(u,v) ist der Fluss vom Knoten u zum Knoten v.
- Der Wert |f| eines Flusses f ist der Gesamtfluss aus der Quelle s:

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

- Ein *maximaler Fluss* ist einen Fluss mit maximalem Wert für ein Flussnetzwerk.

1.3 Flüsse zwischen Knotenmengen

$$
\begin{aligned}
f(x, Y) &= \sum_{y \in Y} f(x, y) && \text{für } Y \subseteq V \\
f(X, y) &= \sum_{x \in X} f(x, y) && \text{für } X \subseteq V \\
f(X, Y) &= \sum_{x \in X} \sum_{y \in Y} f(x, y) && \text{für } X, Y \subseteq V \\
f(X, X) &= 0 && \text{für } X \subseteq V \\
f(X, Y) &= -f(Y, X) && \text{für } X, Y \subseteq V \\
f(X \cup Y, Z) &= f(X, Z) + f(Y, Z) && \text{für } X, Y, Z \subseteq V : X \cap Y = \varnothing \\
f(Z, X \cup Y) &= f(Z, X) + f(Z, Y) && \text{für } X, Y, Z \subseteq V : X \cap Y = \varnothing
\end{aligned}
$$

2 Ford-Fulkerson-Methode

Prinzip

- Überführe den Graphen in ein Flussnetzwerk.
- Suche einen beliebigen Pfad zwischen s und t heraus.
  - Bestimme die minimale Kapazität (maximalen Durchfluss) dieses Pfades.
  - Bestimme die Flusserhöhung, indem der Fluss entlang des Pfades gleich der Kapazität der Kante mit der minimalen Kapazität gesetzt wird.
  - Bilde das Restnetzwerk, indem der Durchfluss als umgekehrte Kanten eingezeichnet wird (eventuell existierende Gegenpfade werden verrechnet).
- Fahre solange fort, neue Pfade zu suchen, bis es keine augmentierende (den Fluss weiter vergrößernde) Pfade mehr gibt, d.h. die Senke noch von der Quelle aus erreichbar ist.

→ Der Maximale Fluss ist die Summe aller Flusserhöhungen.

5 Algorithmenprinzipien

1 Divide & conquer - Teilen und Beherrschen

Das Prinzip hinter *Divide & conquer- Algorithmen* ist, dass sie ein Problem in kleinere Teilprobleme teilen, die zwar dem ursprünglichen Problem ähneln, jedoch eine kleinere Größe besitzen. Die Teilprobleme werden dabei *rekursiv* gelöst. Anschließend werden die Lösungen der Teilprobleme dann kombiniert, um daraus die Lösung des Ursprungsproblems zu erhalten. Das Paradigma umfasst drei Schritte:

- **Teile** das Problem in eine Anzahl von Teilproblemen auf.
- **Beherrsche** die Teilprobleme durch rekursives Lösen. Hinreichend kleine Teilprobleme werden direkt gelöst.
- **Verbinde** die Lösungen der Teilprobleme zur Lösung des Ausgangsproblems.

2 Greedy-Algorithmen

Greedy('gierige') Algorithmen sind dadurch gekennzeichnet, dass sie in jedem Schritt eine *kuzfristig optimale* Lösung wählen. Diese Lösungsfindung sollte eine möglichst geringe Komplexität haben. Nachdem eine Wahl für eine Lösung getroffen wurde ist diese *nicht* mehr rückgängig zu machen.
Die Lösungsstrategie eines Greedy-Algorithmus' ist optimal, wenn sie sich aus optimalen Teilproblemen zusammensetzt, die unabhängig von anderen Teillösungen sind.

3 Dynamische Programmierung

Dynamische Programmierungs-Probleme sind *Optimierungsprobleme*, also Probleme, für die es verschiedene Lösungsmöglichkeiten mit einer *Minimierung* von Kosten oder einer *Maximierung* eines Wertes gibt. Damit Dynamische Programmierung möglich ist, muss die optimale Lösung aus sich überlappenden Teilproblemen bestehen, die sich rekursiv aufeinander beziehen.

Allgemeines Vorgehen

1. Stelle die Rekursionsgleichung (top-down) für den Wert der Lösung auf.
   - Dabei sind folgende Punkte zu klären:
     - Festlegung der Basisfälle
     - Unterscheidung zwischen Maximierungs- und Minimierungsproblem.
     - Festlegung der Abbruchbedingungen
2. Löse die Rekursionsgleichung bottom-up.
3. Bestimme aus dem Wert der Lösung die Argumente der Lösung.
4. Rekonstruiere die Lösung.
   - dies geschieht durch aufstellen einer Lösungstabelle, die entsprechend der Rekursionsgleichung ausgefüllt wird.

3.1 Rucksackproblem

Gegeben sei ein Rucksack, mit maximaler Tragkraft M, sowie n Gegenstände, die sowohl ein Gewicht als auch einen Wert haben. *Nehme möglichst viel Wert mit, ohne den Rucksack zu überladen.*

Rekursionsgleichung

Sei also $C[i, j]$ der maximale Wert des Rucksacks mit Tragkraft j, wenn nur die Gegenstände $0, ..., i-1$ berücksichtigt werden.

$$C[i,j] = \begin{cases} \max(C[i-1,j], c_{i-1} + C[i-1, j-w_{i-1}]) & \\ -\infty & \text{für } j < 0 \\ 0 & \text{für } i = 0,\ j \geqslant 0 \end{cases}$$

```
1  // Eingabe: Gewichte w[i], Werte c[i], Tragkraft M
   int knapDP(int w[n], int c[n], int n, int M) {
3      int C[n+1,M+1];        // Array initialisieren
       for (int j = 0; j <= M; j++)
5          C[0,j] = 0;        // Basisfall
       for (int i = 1; i <= n; i++){
7          for (int j = 0; j <= M; j++)
               if (w[i-1] <= j) {
9                  C[i,j] = max(C[i-1,j], c[i-1] + C[i-1,j-w[i-1]]);
               } else
11                 C[i,j] = C[i-1,j]; // passt nicht
       }
```

```
13      return C[n,M];
}
```

Beispiel

`w[] = { 2, 12, 1, 1, 4 }, c[] = { 2, 4, 2, 1, 10 }, M = 15`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2
2	0	0	2	2	2	2	2	2	2	2	2	2	4	4	6	6
3	0	2	2	4	4	4	4	4	4	4	4	4	4	6	6	8
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

3.2 Longest Common Subsequence (LCS)

Sei $A = (a_1, ..., a_m)$ eine Sequenz. Die Sequenz $A_{ik} = (a_{i1}, ..., a_{ik})$ ist eine Teilsequenz von A wobei $i_1 < i_2 < ... < i_k$ und $ij \in 1, ..., m$.
Beispiel: bca ist eine gemeinsame Teilsequenz von A = a**bc**bd**a**b und B = **b**dc**a**b**a**.

Rekursionsgleichung

Hier lässt sich die Rekursiongleichung für den Wert, also die Länge der LCS aufstellen: $L[i,j] = |LCS(A_i, B_j)|$

$$L[i,j] = \begin{cases} 0 & \text{für } i = 0 \text{ oder } j = 0 \\ L[i-1, j-1] + 1 & \text{falls } a_i = b_j,\ i,j > 0 \\ \max(L[i, j-1], L[i-1, j]) & \text{falls } a_i \neq b_j,\ i,j > 0 \end{cases}$$

```
int lcs(int seq1[], int l1, int seq2[], int l2) {
2     int L[l1+1,l2+1];
      for (int i = 0; i i= l1; i++)
4        L[i,0] = 0;
      for (int j = 0; j i= l2; j++)
6        L[0,j] = 0;

8     for (int i = 1; i i= l1; i++) {
         for (int j = 1; j i= l2; j++) {
10       if (seq1[i] == seq2[j]) {
            L[i,j] = L[i - 1, j - 1] + 1;
12       } else if (L[i - 1, j] > L[i, j - 1]) {
```

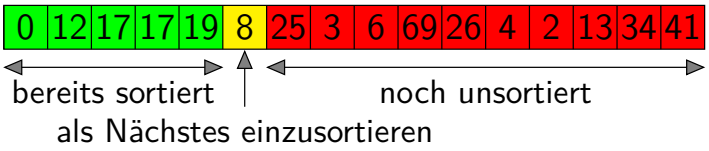
```
            L[i,j] = L[i - 1, j];
14        } else{
            L[i,j] = L[i, j - 1];
16        }
      }
18    return L[l1,l2];
  }
```

6 Sortieralgorithmen

1 Einfache Sortieralgorithmen

1.1 Insertionsort - Sortieren durch einfügen

Prinzip:



- Beginne beim zweiten Element.
- Der unsortierte Bereich des Arrays wird von links nach rechts durchlaufen.
- Gehe zun ersten bisher noch nicht berücksichtigten Element.
- Suche die richtige Position für das Element im bereits sortierten Arrayabschnitt.
- Füge das Element an die richtige Stelle ein.
- Verschiebe alle bereits sortierten Elemente *Rechts* vom eingefügten Element um 1 nach *rechts*.

$\Rightarrow$ Wiederhole solange, bis das Ende des Arays erreicht ist.

```
1  void insertionSort(n){
       for (i = 1; i < n; i++){
3          if (E[i] < E[i-1]){
               int temp = E[i];
5              for (j = i; j>0 && E[j-1] > temp; j--){
                   E[j] = E[j-1];
7              }
               E[j] = temp;
9              }
       }
11 }
```

1.2 Selectionsort

Prinzip:

1. Setze Pointer i auf das erste Element.
2. Suche kleinstes Element rechts von i und speichere die Position dieses Elementes als m.
3. Vertausche i. Element mit dem Wert des Elementes an Position m.
   Setze i um eins hoch.

⇒ Wiederhole 2./3. bis Ende des Arrays erreicht.

```
1  void selectionSort(Array E) {
     int i,j,m;
3    for (i = 0; i < E.length; i++) {
       m = i;
5      for (j = i + 1; j < E.length; j++) {
         if (E[j] <= E[m]) {
7          m = j;
         }
9      }
       int v = E[i];
11     E[i] = E[m];
       E[m] = v;
13   }
   }
```

2 Höhere Sortieralgorithmen

2.1 Heapsort

Prinzip:

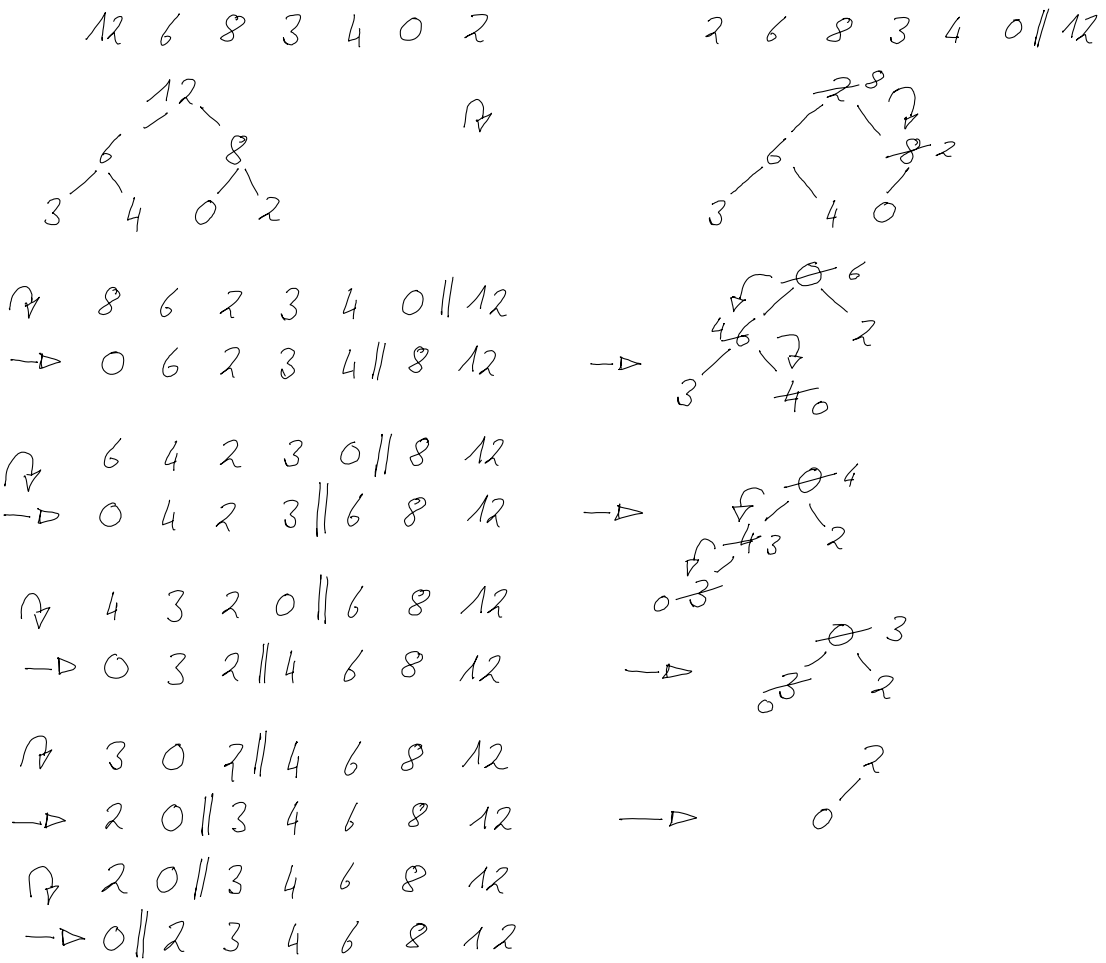
1. Heapsort:
   - *buildHeap:* Heap aus Array aufbauen.
   - Beginne beim letzten Element und gehe nach vorne durch, bis der Arrayanfang erreicht ist:
     - Tausche das erste Arrayelement mit dem aktuell betrachteten.
     - *Heapify:* Stelle die Heapeigenschaft wieder her.
2. Heapify (Heapeigenschaft wiederherstellen):
   - Beginne beim letzten Element und gehe rückwärts bis zum ersten Element:

- Finde das Maximum der Werte A[i] und seiner Kinder
- Is A[i] bereits das größte Element, dann ist der gesamte Teilbaum auch ein Heap.
  - Sonst: Tausche A[i] mit dem größten Element und führe Heapify erneut in diesem Unterbaum aus.

```
void buildHeap(int E[]) {
    for (int i = E.length / 2 - 1; i >= 0; i--) {
        heapify(E, E.length, i);
    }
}


void heapSort(int[] E) {
    buildHeap(E);
    for (int i = E.length - 1; i > 0; i--) {
        swap(E[0], E[i]);
        heapify(E, i, 0);
    }
}

void heapify(int[] E, int n, int pos) {
    int next = 2 * pos + 1;
    while (next < n) {
        if (next + 1 < n && E[next + 1] > E[next]) {
            next = next + 1;
        }
        if (E[pos] > E[next]) {
            break;
        }
        swap(E[pos], E[next]);
        pos = next;
        next = 2 * pos + 1;
    }
}
```



Beispiel für Heapsort mit Eingabearray $E = [12, 6, 8, 3, 4, 0, 2]$

2.2 Countingsort

Prinzip:

- Zunächst ein *Histogramm- Hilfsarray* erstellen, in dem die Häufigkeit jeder Zahl gespeichert wird.
- *Positionsarray* erstellen, in dem von links nach rechts die Werte des Histogramm-Arrays aufaddiert werden.

Beispiel:

Histogramm:	2	0	0	1	2	1
Positionen:	2	2+0	2+0	2+1	3+2	5+1

- Ausgabearray von hinten nach vorne erstellen, indem die neue Position der Zahl X im Eingabearray der Wert an Position $(X - 1)$ des Positionsarrays ist.
  - Der Wert des Positionsarrays wird anschließend um 1 verringert.

Beispiel:

Eingabe:	6	8	0	3	5	4	0	4

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Histogramm:	2	0	0	1	2	1	1	0	1
Positionen	2	2	2	3	5	6	7	7	8

Ausgabearray								Positionsarray								
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
				4				2	2	2	3	**4**	6	7	7	8
	0			4				**1**	2	2	3	4	6	7	7	8
	0		**4**	4				1	2	2	3	**3**	6	7	7	8
	0		4	4	**5**			1	2	2	3	3	**5**	7	7	8
	0	**3**	4	4	5			1	2	2	**2**	3	5	7	7	8
0	0	3	4	4	5			**0**	2	2	2	3	5	7	7	8
0	0	3	4	4	5		8	0	2	2	2	3	5	7	7	**7**
0	0	3	4	4	5	**6**	8	0	2	2	2	3	5	7	**6**	7

```
1   int[n] countingSort(int[] E, int n, int k){
        int[] histogram = new int[k];   // Direkte Adressierungstabelle
3       for(int i = 0; i < 0; i++){
            histogramm[E[i]]++;          // Zähle Häufigkeit
5       }
        for(int i = 1; i < k; i++){       // Berechne Position
7           histogramm[i] = histogram[i] + histogram[i - 1];
        }
9       //Berechne Position
        int ergebnis[n];
11      for (int i = n - 1; i >= 0; i--){
        //läuft nur stabil, wenn rückwärts gezählt wird.
13          histogram[E[i]]--;
            result[histogram[E[i]]] = E[i];
15      }
        return result;
17  }
```

2.3 Mergesort

Prinzip:

- **Teile** das Zahlenarray in zwei (möglichst) gleichgroße Hälften auf.
- **Beherrsche:** Sortiere die Teile durch rekursives *Mergesort*-aufrufen.
- **Verbinde** je zwei bereits sortierte Teilsequenzen zu einen einzigen sortierten Array.

```
1  // Aufruf: mergeSort(E, 0, E.length-1);
   void mergeSort(Array E, int links, int rechts){
3      if (left < right) {
           int mid = (left + right) / 2;     // finde Mitte
5          mergeSort(E, left, mid);          // sortiere linke Haelfte
           mergeSort(E, mid + 1, right);     // sortiere rechte Haelfte
7          // Verschmelzen der sortierten Haelften
           merge(E, left, mid, right);
9      }
   }
11
   void merge(int E[], int left, int mid, int right) {
13     int a = left,
       b = mid + 1;
15     int Eold[] = E;
       for (int left = 0; left <= right; left++) {
17         if (a > mid) { // Wir wissen (Widerspruch): b <= right
               E[left] = Eold[b];
19             b++;
           }else if (b > right || Eold[a] <= Eold[b]) {
21         E[left] = Eold[a];
           a++
23         }else {
               E[left] = Eold[b];
25             b++;
           }
27     }
   }
```

2.4 Quicksort

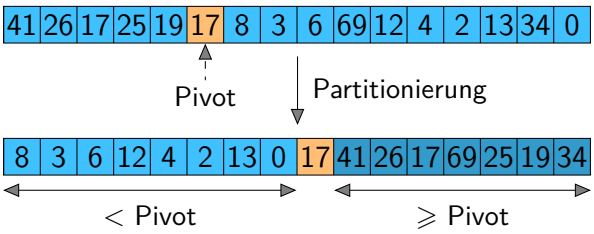
Prinzip

Die Elemente werden zunächst auf die richtige Hälfte des Arrays gebracht, und dann *rekursiv* sortiert.

1. **Teile:** Ein Pivotelement aus dem Array auswählen, Array in zwei Hälften aufteilen:
   - Kleiner als das Pivotelement
   - Mindestens so groß wie das Pivotelement
2. **Beherrsche:** Die Teile *rekursiv* sortieren, das Pivotelement zwischen die sortierten Teile setzen.

Prinzip der Partitionierung:

- Es werden drei Bereiche eingesetzt: “< Pivot”, “> Pivot” und “ungeprüft”.



- Solange das zusätzliche Element "< Pivot" ist, schicke die linke Grenze nach rechts.
- Solange das zusätzliche Element ">= Pivot" ist, schiebe die rechte Grenze nach links.
- Tausche das links gefundene mit dem rechts gefundenen.

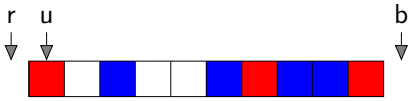
⇒ Fahre fort, bis sich die Grenzen treffen.

```
1  void quickSort(int E[], int left, int right) {
       if (left < right) {
3          int i = partition(E, left, right);
           // i ist Position des Pivotelementes
5          quickSort(E, left, i - 1); // sortiere den linken Teil
           quickSort(E, i + 1, right); // sortiere den rechten Teil
7      }
   }
9
   int partition(int E[], int left, int right) {
11     // Wähle einfaches Pivotelement
       int ppos = right, pivot = E[ppos];
13     while (true) {
           // Bilineare Suche
15         while (left < right && E[left] < pivot) left++;
           while (left < right && E[right] >= pivot) right--;
17         if (left >= right) {
               break;
19         }
           swap(E[left], E[right]);
21     }
       swap(E[left], E[ppos]);
23     return left; // gib neue Pivotposition als Splitpunkt zurück
   }
```

2.5 Dutch National Flag- Problem

Prinzip: Jedes Feld eines Arrays kann drei mögliche Werte haben: Rot, Weiß oder Blau.

- Teile das Array in *vier Regionen* auf:
  1. $0 < i \leq r$
  2. $u < i < b$
  3. $b \leq i \leq n$
- Es gibt 3 Zeiger:
  - **r:** Rote Region
  - **b:** Blaue Region
  - **u:** Unbekannte Region



Zeigerstartposition beim Dutch National Flag- Problem

- Steht u auf einem *roten* Feld, wird $E[u]$ mit $E[r+1]$ vertauscht und der rote Bereich vergrößert und der unbekannte Bereich um 1 verringert *(r++, u++)*
- Steht u auf einem *weißen* Feld, wird der unbekannte Bereich um 1 verkleinert *(u++)*
- Steht u auf einem *blauen* Feld, wird $E[u]$ mit $E[b-1]$ vertauscht. Der blaue Bereich erhöht sich um 1. *(b–, u bleibt gleich!)*

$\Rightarrow$ Der Algorithmus terminiert, wenn $u = b$.
In jedem Schritt wird entweder u erhöht, oder b verringert.

```
void DutchNationalFlag(Color E[], int n) {
    int r = 0, b = n + 1; // rote und blaue Regionen sind leer
    int u = 1; // weiße Region ist leer, die unbekannte == E
    while (u < b) {
        if (E[u] == rot) {
            swap(E[r + 1], E[u]);
            r = r+1; // vergrößere die rote Region
            u = u + 1; // verkleinere die unbekannte Region
        }
        if (E[u] == weiss) {
            u = u + 1;
        }
        if (E[u] == blau) {
            swap(E[b - 1], E[u]);
            b = b-1; // vergrößere die blaue Region
        }
    }
}
```

3 Gesamtübersicht über Sortieralgorithmen

	Laufzeitkomplexität			
Algorithmus	**Worst case**	**Average case**	**Speicher**	**Stabilität**
Insertionsort	$\Theta(n^2)$	$\Theta(n^2)$	in-place	stabil
Selectionsort	$\Theta(n^2)$	$\Theta(n^2)$	in-place	nicht stabil (*)
Quicksort	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(\log n)$	nicht stabil(*)
Mergesort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$	stabil
Heapsort	$\Theta(n \log n)$	$\Theta(n \log n)$	in-place	nicht stabil
Countingsort[1]	$\Theta(n + k)$	$\Theta(n + k)$	O(n + k)	stabil
Dutch National Flag	$\Theta(n)$	$\Theta(n)$	in place	nicht stabil

(*): Vorlesungsversion ist nicht stabil, es existiert jedoch auch eine stabile Version.

7 Graphenalgorithmen

1 Graphensuche

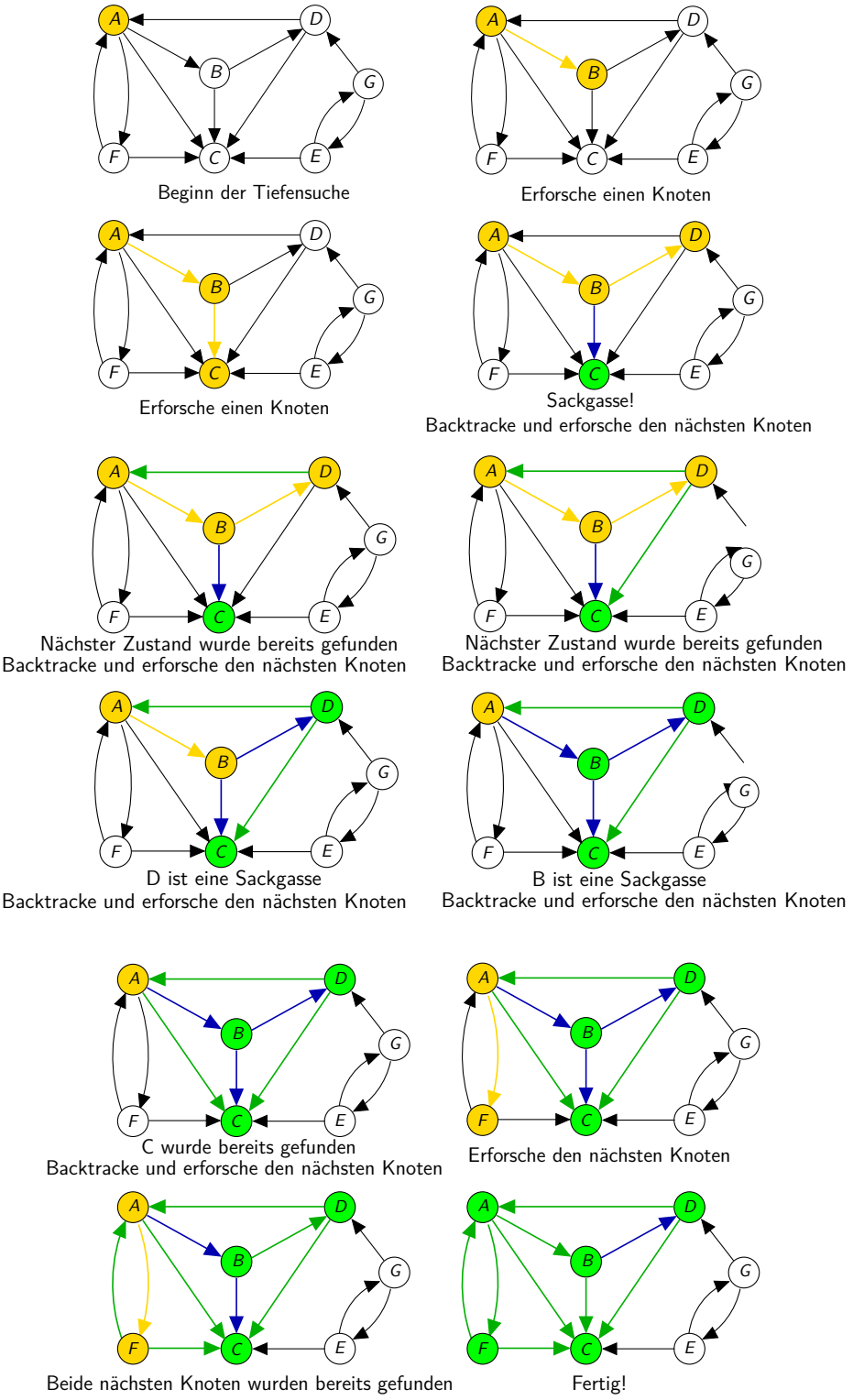
1.1 Tiefensuche

(Tiefensuche: Depth-First Search, DFS)

Prinzip:

- Alle Knoten als *'nicht gefunden'* markieren.
- Aktuellen Knoten als *'gefunden'* markieren.
- Jede Kante mit 'gefundenem' *Nachfolger*:
  - Erforsche Kante, besuche Nachfolger und forsche von dort aus weiter bis es nicht mehr weiter geht.
- Für jede Kante mit gefundenem Nachfolger:
  - Kante überprüfen ohne den Knoten selbst zu besuchen.
- Knoten als *'abgeschlossen'* markieren.

$\Rightarrow$ Man erhält die Menge der vom Startknoten aus erreichbaren Knoten.



Beispiel für eine Tiefensuche

```
void dfsSearch(List adjList[n], int n, int start){
    int color[n];
    for(int i = 0; i < n; i++){    // Farbarray initialisieren
        color[i]  = WHITE;
    }
    dfsRec(adjList, n, start, color);
}

void dfsRec(List adjList[n], int start, int &color[n]){
    color[start] = GRAY;          // Aktuell betrachteten Knoten grau färben
    foreach(next in adjlist[start]){
        if(color[next] == WHITE)
            dfsRec(adjList, n, next, color);
    }
    color[start] = BLACK; // Gefundenen Startknoten schwarz färben
}
```

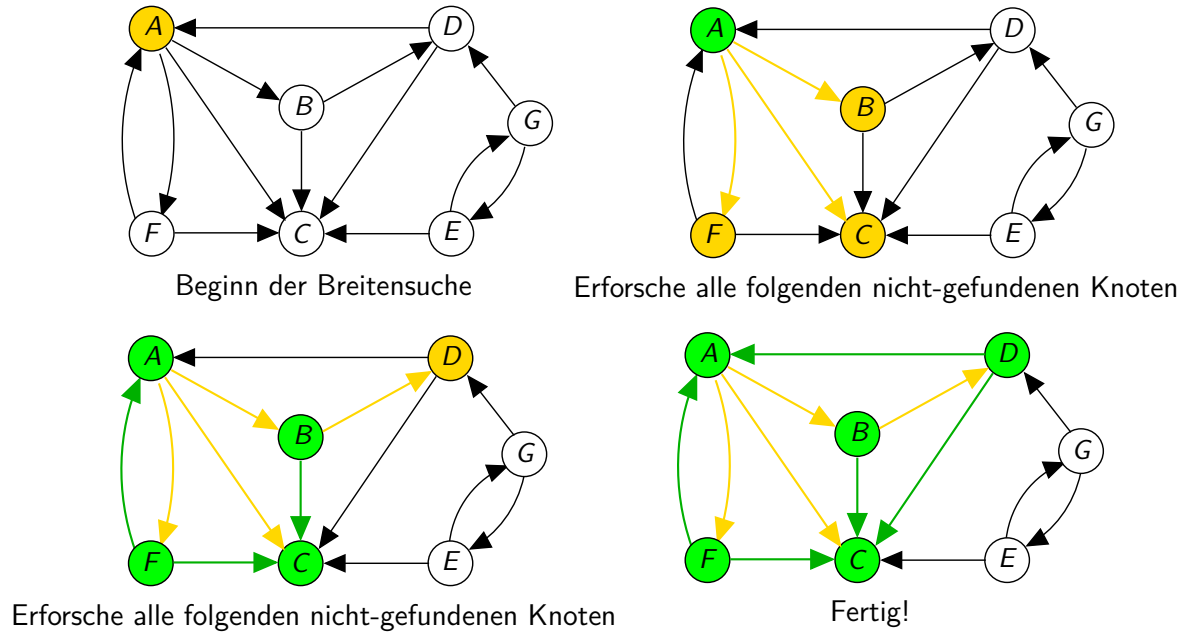
1.2 Breitensuche

Prinzip:

- Alle Knoten als *'nicht gefunden'* markieren
- Aktuell betrachteten Knoten als *'gefunden'* markieren
- Jede Kante mit *'nicht gefundenem'* Nachfolger:
  - *Gleichzeitig* von allen diesen Knoten aus weiter suchen

  → *Kein* Backtracking

⇒ Man erhält die Menge aller Knoten, die vom Stratknoten aus erreichbar ist.



Beispiel für eine Breitensuche

```
void bfsSearch(list adjlist(n), int n, int start){
    int color[];
    Queue wait; // Warteschlange initialisieren
    //Alle Knoten als 'nicht gefunden' markieren
    for (int i = 0; i < n; i++)
        color[i] = WHITE;
    color[start] = GRAY; //Startknoten als 'gefunden' markieren
    wait.enqueue(start)//Startknoten auf Warteschlange setzen
    While(!Wait.isempty()){
        int v = wait.dequeue());//Knoten als bearbeitet von queue nehmen
        foreach(w in adjlist[v]){//Die Nachfolgerknoten von v betrachten
            if color[w] == WHITE {
                color[w] = GRAY;
                wait.enqueue(w);  //Nachfolgerknoten als nächste Startknoten
            }
        }
    }
    color[v] = BLACK;//Gefundenen Knoten als 'abgeschlossen' markieren
}
```

2 Sharir's Algorithmus

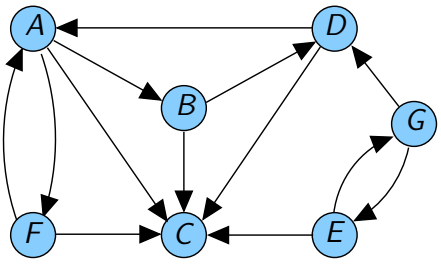
Sharir's Algorithmus findet *starke Zusammenhangskomponenten*.

Prinzip

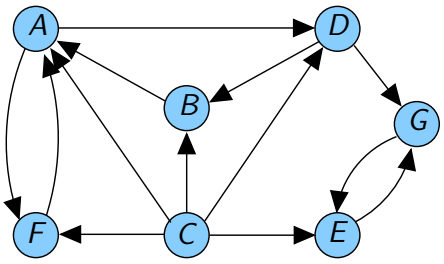
- Tiefensuche auf Graph
  - Alle Knoten, die *'abgeschlossen'* sind, also gefunden wurden, auf einem Stack speichern.
- Graph *transponieren* und auf diesem transponierten Graphen eine weitere Tiefensuche durchführen. Dabei in der Reihenfolge die Knoten besuchen, in der sie auf dem Stapel stehen.
- *Leiter* speichern. Leiter sind Knoten, die als erste bei der 2. Tiefensuche 'entdeckt' wurden.

$\Rightarrow$ Die starken Zusammenhangskomponenten sind die im stapel zwischen zwei *Leitern*.

Beispiel:

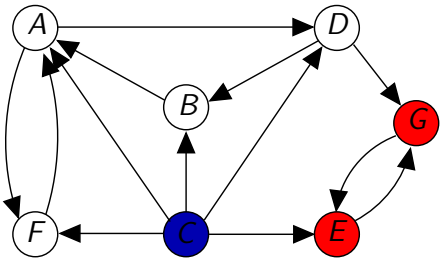


Ursprünglicher Digraph



Transponierter Graph

Stack am Ende der Phase 1



In Phase 2 gefundene starke Komponenten

Zeitkomplexität

Sharir's Algorithmus hat eine Zeitkomplexität von $\Theta(|V| + |E|)$.
Seine Speicherkomplexität beträgt: $\Theta(|V|)$.

3 Prims Algorithmus

Der Algorithmus findet minimale Spannbäume.

Prinzip

- Bei beliebigen Knoten beginnen, Wert dieses Knotens auf **0**, den der anderen auf ∞ setzen.
- Günstigste Kante bestimmen ($\rightarrow$ *Greedy- Algorithmus*), Werte der angrenzenden Knoten updaten, wenn Kantengewicht kleiner als der Wert ist.
- Bei günstigem Knoten fortfahren, bis keine weiteren Randknoten mehr verfügbar sind.
  - Bei gleichem Gewicht wird in *alphabetischer Reihenfolge* vorgegangen.

4 Single Source Shortest Path (SSSP)

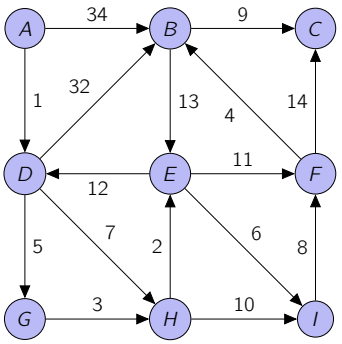
4.1 Dijkstra-Algorithmus

Dijkstra ermittelt die kürzesten Wege vom Startknoten zu allen anderen Knoten des Graphen.
Hier sind (im Gegensatz zu Bellman Ford) ausschließlich *nicht* negative Kantengewichte zugelassen.

- Initialisiere alle Knoten mit ∞
- Aktualisiere ausgehend vom aktuell betrachteten Knoten alle Knoten, die mit diesem verknüpft sind, wenn die Summe aus aktuellem Knoten und der Kante geringer ist als der Wert des Zielknotens. Betrachte den aktuellen Knoten zukünftig nicht mehr.
- Fahre mit dem Knoten mit minimalem Wert fort.
  $\rightarrow$ Greedy-Algorithmus

$\Rightarrow$ Führe diese Schritte in einer (großen) Tabelle aus (siehe Beispiel).

Beispiel



Schritt	0	1	2	3	4	5	6	7	8
Knoten	A	D	G	H	E	I	F	B	C
$D[A]$	0	X	X	X	X	X	X	X	X
$D[B]$	34	33	33	33	33	33	25	25	X
$D[C]$	∞	∞	∞	∞	∞	∞	35	34	34
$D[D]$	1	1	X	X	X	X	X	X	X
$D[E]$	∞	∞	∞	10	10	X	X	X	X
$D[F]$	∞	∞	∞	∞	21	21	21	X	X
$D[G]$	∞	6	6	X	X	X	X	X	X
$D[H]$	∞	8	8	8	X	X	X	X	X
$D[I]$	∞	∞	∞	18	16	16	X	X	X

4.2 Bellman-Ford-Algorithmus

Bellman Ford bestimmt *den Kürzesten Pfad* von einem Startknoten zu allen anderen Knoten nach dem SSSP (*single source shortest path*)- Prinzip.
Wichtigster Unterschied zu *Dijkstra* ist, dass er auch mit negativen Kantengewichten umgehen kann.

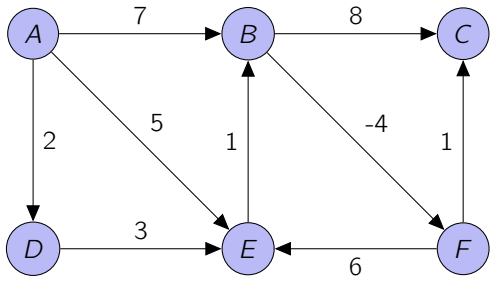
Prinzip

1. Initialisiere alle Knoten mit ∞
2. Initialisiere den Startknoten mit **0**
3. Aktualisiere die Kosten aller direkt durch Kanten erreichbaren Knoten, wenn durch die Summe des Ursrungsknotens und der Kosten der Kante ein *geringerer* Wert erzielt wird, als der Zielknoten bislang hatte.
4. Wiederhole 3), bis sich die Kosten nicht mehr ändern.

$\rightarrow$ Breche ab, wenn ein negativer Zyklus vorliegt. Dies ist der Fall, wenn nach (Knotenanzahl - 1)- Wiederholungen noch Verbesserungen möglich sind.

$\Rightarrow$ Führe diese Schritte in einer Tabelle aus (siehe Beispiel).

Beispiel



			1/A	1/B	1/E	1/F	2/B	2/F	
A:	∞	0							A: ∞, 0
B:	∞		7		6				B: ∞, 7, 6
C:	∞			15		4		3	C: ∞, 15, 4, 3
D:	∞		2						D: ∞, 2
E:	∞		5						E: ∞, 5
F:	∞			3			2		F: ∞, 3, 2

Der Graph enthält keinen Zyklus mit negativem Gewicht.

5 Topologische Sortierung

Topologische Sortierung bezeichnet eine Reihenfolge von Dingen, bei der vorgegebene Abhängigkeiten erfüllt sind. Anstehende Tätigkeiten einer Person etwa unterliegen einer Halbordnung: es existieren Bedingungen wie „Tätigkeit A muss vor Tätigkeit B erledigt werden“. Eine Reihenfolge, welche alle Bedingungen erfüllt, nennt man topologische Sortierung der Menge anstehender Tätigkeiten. Eine Topologische Sortierung ist nur möglich, wenn es keinen Zyklus gibt, d.h. wenn keine gegenseitigen Abhängigkeiten vorliegen.

Prinzip:

- Stelle die Abhängigkeiten als Graph dar.
  - Die gerichteten Kanten zeigen jeweils zum Knoten, von dem der Ausgangsknoten abhängig ist
- Sortiere die Knoten nach ihrer Abhängigkeiten so dass rechts diejenigen Knoten stehen, von denen nichts abhängt und links diejenigen Knoten stehen, die *keine* eigenen Abhängigkeiten haben
- Versehe alle Knoten mit den jeweiligen Kosten.

⇒ Der Kritische Pfad ist der Pfad mit den maximalen Kosten (betrachtet von links nach rechts).

Der *kritische Pfad* ist eine Folge von Aufgaben $v_0 \dots v_k$, sodass

- v_0 keine Abhängigkeiten hat
- v_i abhängig von v_{i-1} ist, wobei $est(v_i) = eft(v_{i-1})$ (Keine Verzögerung!)

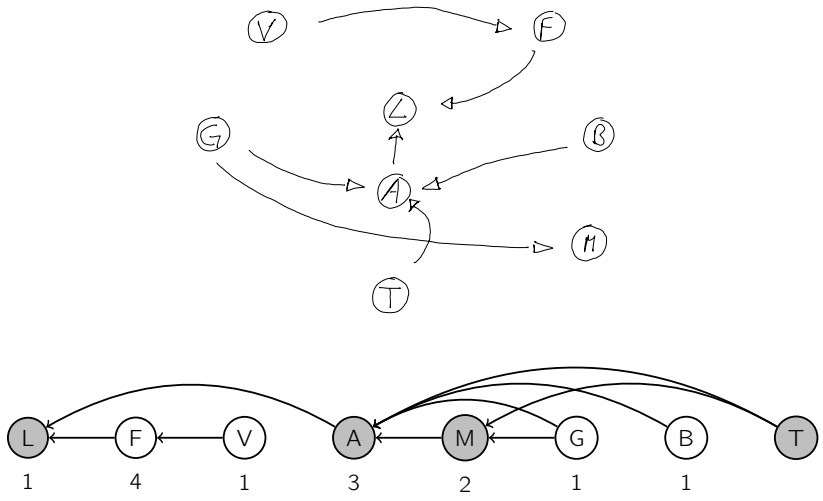
est: earliest starting-time

eft: earliest finish-time

- $eft(v_k)$ das Maximum über alle Aufgaben ergibt.

Beispiel

Arbeitsschritt	Abkürzung	Dauer	Abhängig von
Vorlesung	V	1	F
Folien	F	4	L
Lehrstoff	L	1	-
Globalübung	G	1	A, M
Musterlösung	M	2	A
Aufgaben	A	3	L
Blatt	B	1	A
Tutorenbesprechung	T	1	A, M



Darstellung der Abhängigkeiten als gerichteter Graph und als Topologische Sortierung mit kritischem Pfad.

– Vollständige Tiefensuche, bei der zusätzlich ein Array *'kritische Länge'* und *'earliest finish-time'* übergeben wird.

– Nach der Abfrage der Knotenfarbe wird *jedes mal* folgende Abfrage durchgeführt:

```
1        if(eft[next] >= est){
             est = eft[next]
3            critDep[start] = next;
```

```
}
```

– Bevor der Knoten schwarz gefärbt wird, wird

```
eft[start] = est + duration[start]
```

durchgeführt.

6 All Pairs Shortest Paths

Betrachtet wird die Länge des Pfades jedes Knotens mit jedem anderen Knoten. Hier sind negative Kantengewichte, jedoch keine negativen Zyklen zugelassen.

6.1 Floyd-Warshall Algorithmus

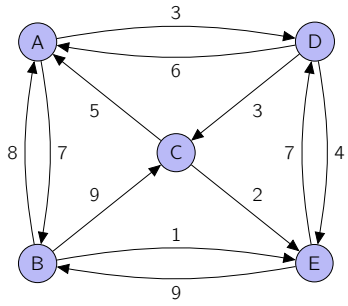
Prinzip

Verbessert sich die Pfad*länge*, wenn der Weg über Zwischenknoten führt?

– Floyd-Warshall ist eine klassische Aufgabe der Dynamischen Programmierung:

$$d_{ij}^{(k)} = \begin{cases} W(i,j) & \text{für } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{für } k > 0 \end{cases}$$

Beispiel



	A	B	C	D	E
A	0	7	∞	3	∞
B	8	0	9	∞	1
C	5	∞	0	∞	2
D	6	∞	3	0	4
E	∞	9	∞	7	0

	A	B	C	D	E
A	0	7	∞	3	∞
B	8	0	9	11	1
C	5	12	0	8	2
D	6	13	3	0	4
E	∞	9	∞	7	0

	A	B	C	D	E
A	0	7	16	3	8
B	8	0	9	11	1
C	5	12	0	8	2
D	6	13	3	0	4
E	17	9	18	7	0

	A	B	C	D	E
A	0	7	16	3	8
B	8	0	9	11	1
C	5	12	0	8	2
D	6	13	3	0	4
E	17	9	18	7	0

	A	B	C	D	E
A	0	7	6	3	7
B	8	0	9	11	1
C	5	12	0	8	2
D	6	13	3	0	4
E	13	9	10	7	0

	A	B	C	D	E
A	0	7	6	3	7
B	8	0	9	8	1
C	5	11	0	8	2
D	6	13	3	0	4
E	13	9	10	7	0

7 Algorithmen auf Flussnetzwerken

7.1 Ford-Fulkerson-Methode - Maximaler Fluss

Prinzip

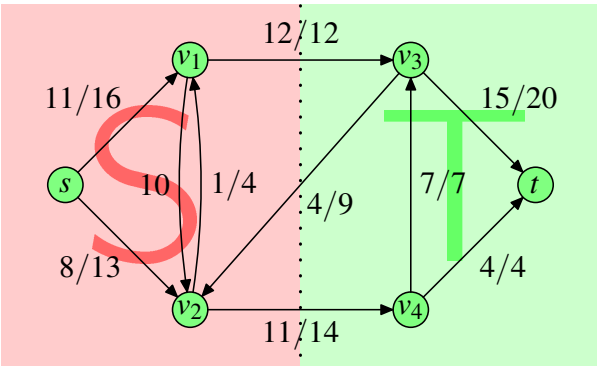
- Überführe den Graphen in ein Flussnetzwerk.
- Suche einen beliebigen Pfad zwischen s und t heraus.
  - Bestimme die minimale Kapazität (maximalen Durchfluss) dieses Pfades.
  - Bestimme die Flusserhöhung, indem der Fluss entlang des Pfades gleich der Kapazität der Kante mit der minimalen Kapazität gesetzt wird.
  - Bilde das Restnetzwerk, indem der Durchfluss als umgekehrte Kanten eingezeichnet wird (eventuell existierende Gegenpfade werden verrechnet).

– Fahre solange fort, neue Pfade zu suchen, bis es keine augmentierende (den Fluss weiter vergrößernde) Pfade mehr gibt, d.h. die Senke noch von der Quelle aus erreichbar ist.

→ Der Maximale Fluss ist die Summe aller Flusserhöhungen.

7.2 Min-cut-max-flow

Ein Schnitt (S, T) in einem Flussnetzwerk $G = (V, E, c)$ ist eine Partition $S \cup T = V, S \cap T \neq \emptyset$ mit $s \in S$ und $t \in T$. Die Knoten werden also in zwei Gruppen aufgeteilt: Eine, die die Senke- und eine, welche die Quelle enthält.



(Schnitt $(S = /s, v_1, v_2/, T = /t, v_3, v_4/)$ hier 26)

Max-flow Min-cut Theorem

Ist f ein Fluss im Flussnetzwerk G, dann sind äquivalent (gleichbedeutend):

– f ist ein maximaler Fluss.

– In G_f gibt es keinen augmentierenden Pfad.

– $|f| = c(S, T)$ für einen Schnitt (S, T), d.h. der Schnitt (S, T) ist minimal.

Daraus folgt:

→ Die Kapazität eines minimalen Schnittes ist gleich dem Wert eines maximalen Flusses. (Kapazität des minimalen Schnittes entspricht dem maximalen Flusses)

→ Falls die Ford–Fulkerson–Methode terminiert, berechnet sie einen maximalen Fluss.

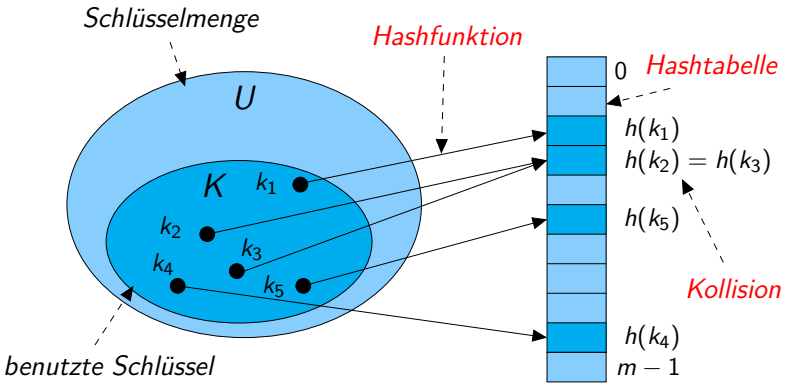
8 Hashing

Allgemeines Prinzip

Das Ziel von Hashing ist es, einen großen Schlüsselraum auf einen kleineren Bereich von ganzen Zahlen abzubilden. Dabei soll möglichst unwahrscheinlich sein, dass zwei Schlüssel auf die selbe Zahl abgebildet werden. Eine *Hashfunktion* bildet einen *Schlüssel* auf einen *Index* der Hashtabelle T ab:

$$h : U \to \{0, 1, ..., m-1\} \text{ für Tabellengröße m und } |U| = n.$$

Wir sagen, dass h(k) der Hashwert des Schlüssels k ist. Das Auftreten von $h'(k)$ für $k \neq k'$ nennt man *Kollision*.



1 Hashfunktionen

Eine Hashfunktion bildet einen Schlüssel auf einen Index (eine ganze Zahl) ab.

Gute Hashfunktionen

Eine gute Hashfunktion ist charakterisiert durch folgende Eigenschaften:

  - Effizient
    - Geringer Speicheraufwand
    - Einfach zu berechnen
    - Eingabewerte müssen nur einmal ausgelesen werden
  - Ähnliche Schlüssel sollten zu unterschiedlichen und breit verteilten Hashwerten führen.
  - Die Hashfunktion sollte *surjektiv* sein, d.h möglichst jeder Hashwert sollte genutzt werden.
    - Dabei sollte alle Indizes in etwa die gleiche Häufigkeit besitzen.

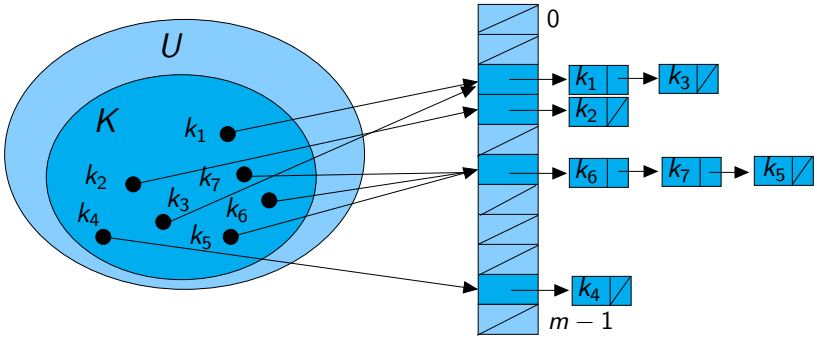
Es gibt verschiedene Methoden, um eine *gute* Hashfunktion zu erhalten:

  - **Divisionsmethode:** Hashfunktion: $h(k) = k \mod m$
    - Dabei sollte m möglichst eine Primzahl sein und *nicht* zu nahe an einer Zweierpotenz liegen.
  - **Multiplikationsmethode:** Hashfunktion: $h(k) = \lfloor m(k \cdot c \mod 1) \rfloor$ für $0 < c < 1$

    $k \cdot c \mod 1$ ist hierbei der Nachkommateil von $k \cdot c$, d.h. $k \cdot c - \lfloor k \cdot c \rfloor$.
  - **Universelles Hashing:** Löst das Problem, dass es immer eine ungünstige Sequenz von Schlüsseln geben kann, sodass alle Schlüssel auf einen Hashwert abgebildet werden.
    - Dazu wird beim universellen Hashing eine *zufällige* Hashfunktion aus einem Pool H von Hashfunktionen ausgewählt.
    - Eine Menge H der Hashfunktionen wird als universell betrachtet, wenn mit einer zufällig aus H ausgewählten Hashfunktion die Wahrscheinlichkeit für eine Kollision zwischen den Schlüsseln k und l nicht größer als die Wahrscheinlichkeit $\frac{1}{m}$ einer Kollision ist, wenn $h(k)$ und $h(l)$ *zufällig* und *unabhängig* aus der Menge $\{0, 1, ..., m-1\}$ gewählt wurden.
    - Für universelles Hashing ist die erwartete Länge der Liste $T[k]$
      - gleich α, wenn k nicht in T enthalten ist.
      - gleich $1 + \alpha$, wenn k in T enthalten ist.

2 Geschlossenes Hashing

Beim geschlossenen Hashing, auch Kollisionsauflösung durch Verkettung genannt, werden alle Schlüssel, die zum gleichen Hashwert führen, innerhalb einer verketteten Liste gespeichert, auf die ein Arrayeintrag verweist:



Idee der Kollisionsauflösung durch Verkettung

Komplexität

- Im **Worst-Case** haben alle Schlüssel den selben Hashwert. Suchen und Löschen hat die selbe Worst-Case Komplexität wie bei Listen: $W(n) = \Theta(n)$.
- Im **Average-Case** jedoch beträgt die Laufzeit zum (erfolgreichen und erfolglosen) Suchen $A(n) = \Theta(1 + \alpha)$.
  α ist hierbei der *Füllgrad* der Hashtabelle: $\alpha = \frac{n}{m}$ (n: Anzahl der eingegebenen Schlüssel, m: Größe der Hashtabelle)

3 Offenes Hashing

Allgemeines Prinzip:

Beim offenen Hashing, auch Hashing mit offener Adressierung genannt, werden alle Schlüssel direkt auf eine einzige Hashtabelle abgebildet. Dabei können höchstens n Schlüssel gespeichert werden:

$$\alpha(n, m) = \frac{n}{m} \leq 1$$

Einfügen eines Schlüssels k

- *Sondiere* die Positionen der Hashtabelle, bis ein leerer Slot gefunden wird
  - Die zu überprüfenden Positionen werden mittels einer *Sondierungsfunktion* in Abhängigkeit des Schlüssels k bestimmt.

Die Hashfunktion hängt also sowohl vom Schlüssel k, als auch von der *Nummer der Sondierung i* ab.

Sondierungsfunktionen

Sondierungsfunktionen ordnen einen Schlüssel k auf eine Position der Hashtabelle zu. Dazu wird bei jeder fehlgeschlagenen Zuordnung eines Schlüssels dessen Zähler i um 1 hochgesetzt und die Sondierungsfunktion mit diesen modifizierten Parametern erneut aufgerufen.

- **Lineares Sondieren:** $h(k,i) = (h'(k) + i) \mod m$ (für $i < m$)

  mit: k: Schlüssel

  i: Sondierungsschritt

  h': Hashfunktion

  - Die Verschiebung der nachfolgende Sondierungen ist linear von i abhängig, die erste Sondierung bestimmt bereits die gesamte Sequenz.

    → Es können insgesamt m verschiedene Sequenzen erzeugt werden.

- **Quadratisches Sondieren:** $h(k,i) = (H'(k) + c_1 \cdot i + c_2 \cdot i^2) \mod m$ (für $i < m$)

  mit: k: Schlüssel

  i: Sondierungsschritt

  h': Hashfunktion

  Konstanten $c_1, c_2 \neq 0$

  - Die Verschiebung der nachfolgende Sondierungen ist quadratisch von i abhängig, die erste Sondierung bestimmt auch hier bereits die gesamte Sequenz.

    → Bei geeignet gewähltem c_1, c_2 können auch hier insgesamt m verschiedene Sequenzen erzeugt werden.

    → Clustering (Aneinanderreihung), welches bei linearem Sondieren auftritt wird jedoch vermieden. Jedoch tritt *sekundäres Clustering* immer noch auf:

      Beispiel: $h(k,0) = h(k',0)$ verursacht $h(k,i) = h(k',i)$ (für alle i)

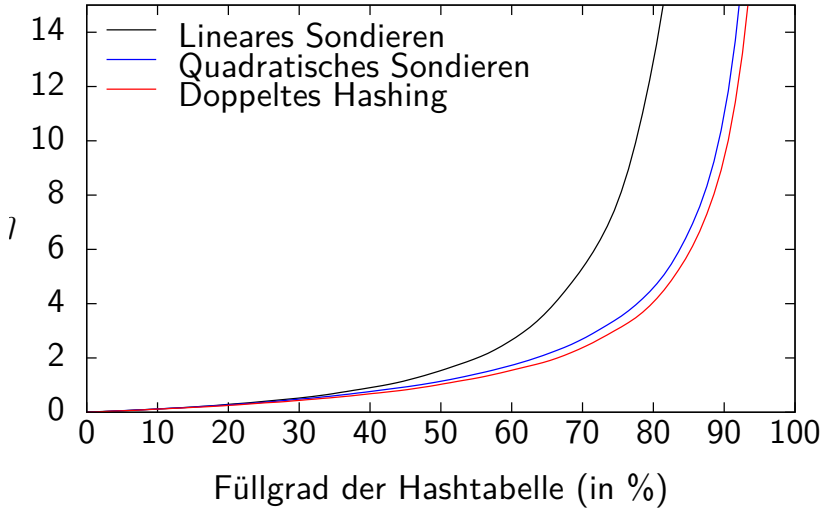
  - **Doppeltes Hashing:** $h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod m$ (für i < m)

    mit: h_1, h_2: Übliche Hashfunktionen.

- Die Verschiebung der nachfolgenden Sondierungen ist von $h_2(k)$ abhängig. Die erste Sondierung bestimmt also *nicht* die gesamte Sequenz.

  → Dies führt zu einer besseren Verteilung der Schlüssel in der Hashtabelle. Dies kommt gleichverteiltem Hashing nahe.

  - Sind h_2 und m *relativ prim* (Teilerfremd), wird die gesamte Hashtabelle abgesucht.



Vergleich der drei Sondierungsmethoden

Komplexität

Average-Case:

- Die erfolgreiche Suche benötigt $O\left(\frac{1}{\alpha} \cdot \ln \frac{1}{1-\alpha}\right)$
- Die erfolglose Suche benötigt $O(\frac{1}{1-\alpha})$

Löschen eines Schlüssels k

Problematik

Im Gegensatz zum Geschlossenen Hashing kann ein zu löschender Schlüssel hier nicht einfach gelöscht (durch *null* ersetzt) werden, da ansonsten das beim Einfügen geschehene Sondieren nicht berücksichtigt wird.

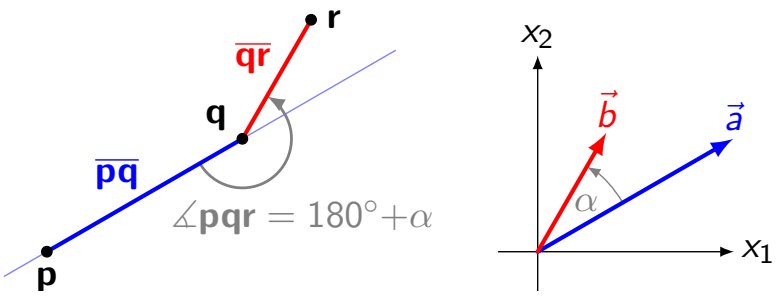
Daher muss hier statt einem Löschen (ersetzen mit *null*) der Schlüssel durch einen *speziellen Wert DELETED* ersetzt werden.

9 Algorithmische Geometrie

1 Mathematische Grundlagen

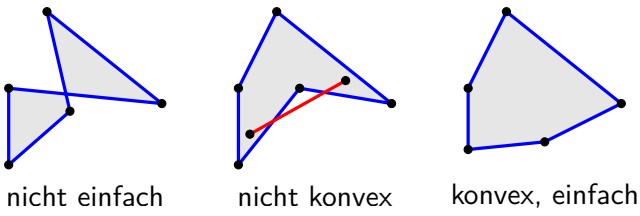
Winkelbestimmung mit Determinanten

Gegeben der Streckenzug (p, q, r):

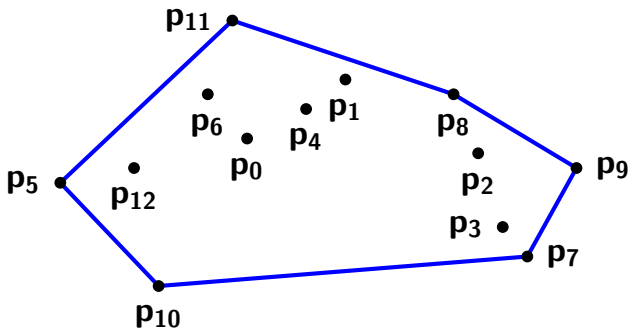


- $\overrightarrow{a} = \overrightarrow{d_{pq}} = q - p$ und $\overrightarrow{b} = \overrightarrow{d_{pr}} = r - q$
- **Linksdrehung**, wenn $\det(\overrightarrow{a}, \overrightarrow{b}) > 0$
- **Rechtsdrehung**, wenn $\det(\overrightarrow{a}, \overrightarrow{b}) < 0$
- Wenn $\det(\overrightarrow{a}, \overrightarrow{b}) = 0$ liegt r auf der Verlängerung von $\overline{pq}$. ($\measuredangle pqr = 0°$ oder $180°$)

Polygone



Konvexe Hülle



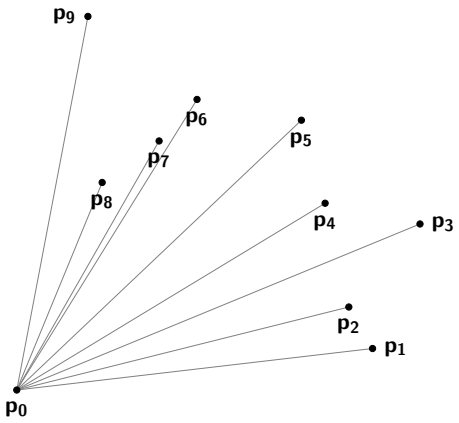
Die konvexe Hülle einer Menge Q von Punkten ist das kleinste konvexe Polygon P, für das sich jeder Punkt in Q entweder auf dem Rand von P oder in seinem Inneren befindet. Hilfreich ist hierbei die Vorstellung der Punktemenge als Nägel auf einem Brett, um die außen ein Gummiband gespannt wird. Die konvexe Hülle hat dann die Form, die durch das straffe Gummiband gebildet wird, das alle Nägel umschließt.

2 Graham-Scan

Der Graham-Scan gibt die konvexe Hülle einer Menge Q von Punkten in einem zweidimensionalen Raum aus.

Prinzip

1. Suche den Startpunkt, der auf jeden Fall auf der Hülle liegt.
   - Konvention: Gewählt wird der Punkt mit der niedrigsten x_2- Koordinate. Haben mehrere Punkte die gleiche x_2-Koordinate, so GIBT zusätzlich die geringste x_1-Koordinate den Ausschlag
2. Von Punkt diesem ausgehend sortiere die Punkte nach zunehmendem (Polar-) Winkel zur x_1- Achse (entspricht einer *rotierenden Sweepline*):
   - Dies geschieht mittels Determinantenberechnung: Die Determinanten der Strecken vom Startpunkt zum betrachteten Punkt und der x_1-Achse werden berechnet. Die Reihenfolge der Determinanten gibt die Polarwinkelreihenfolge an.

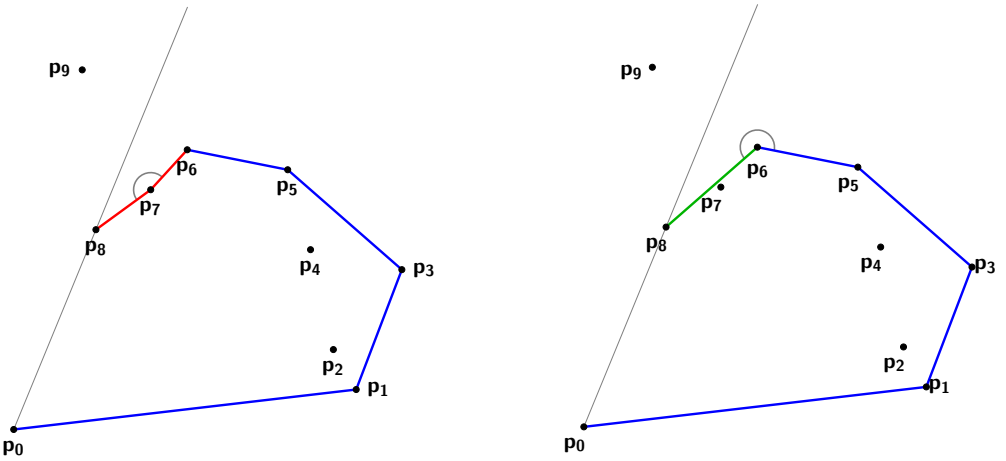


3. Die ersten drei Punkte werden nun auf einen Stapel gelegt (und somit als 'auf der konvexen Hülle liegend' betrachtet).
4. Die weiteren Punkte werden einzeln auf ihren Winkel zu den beiden Vorgängerpunkten auf dem Stapel überprüft:
   - Die Winkelüberprüfung geschieht mittels Determinante:

     $$\det(stack.top - (stack.top - 1), NeuesElement - stack.top)$$

     → Es wird also die Determinante von (oberstem Stackelement *abzüglich* zweitoberstem Stackelement) und (neu hinzukommendem Element *abzüglich* oberstem Stackelement) berechnet.
   - Hier sind *zwei* Fälle möglich:
     a) Ist die berechnete Determinante ≤ 0, knickt die Verbindungsstrecke der drei Punkte bei *Stack.top* nach *rechts* ab. Das oberste Stapelelement wird entfernt, da es sich *nicht* auf der Hülle, sondern *im inneren* des Polygons befindet.
        - Das neu hinzukommende Element wird nun erneut mit den restlichen auf dem Stack befindlichen Elementen überprüft.

     b) Ist die berechnete Determinante > 0, knickt die Verbindungsstrecke der drei Punkte bei *Stack.top* nach *links* ab. Das neu hinzukommende Element wird oben auf den bestehenden Stapel gelegt.



Rot: p_7 wird aus konvexer Hülle entfernt, Grün: p_8 wird anschließend überprüft.

Beispiel

Konvexe Hülle für folgende Punkte in einem Zweidimensionalen Koordinatensystem:
(1, 0); (4, 1); (4, 3); (3, 3); (2, 2); (1, 4); (0, 2)

Knoten	Vektoren und Determinante	Stack
$(3,3)$		
$(4,3)$	$\det(\begin{pmatrix} 0 \\ 2 \end{pmatrix}, \begin{pmatrix} -1 \\ 0 \end{pmatrix}) = 0 + 2 = 2 > 0$	$(3,3)$
$(4,1)$		$(4,3)$
		$(4,1)$
		$(1,0)$
$(2,2)$		
$(3,3)$	$\det(\begin{pmatrix} -1 \\ 0 \end{pmatrix}, \begin{pmatrix} -1 \\ -1 \end{pmatrix}) = 1 - 0 = 1 > 0$	$(2,2)$
$(4,3)$		$(3,3)$
		$(4,3)$
		$(4,1)$
		$(1,0)$

Knoten	Vektoren und Determinante	Stack
$(1,4)$ $(2,2)$ $(3,3)$	$\det(\begin{pmatrix} -1 \\ -1 \end{pmatrix}, \begin{pmatrix} -1 \\ 2 \end{pmatrix}) = -2 - 1 = -3 \leq 0$	$(3,3)$ $(4,3)$ $(4,1)$ $(1,0)$
$(1,4)$ $(3,3)$ $(4,3)$	$\det(\begin{pmatrix} -1 \\ 0 \end{pmatrix}, \begin{pmatrix} -2 \\ 1 \end{pmatrix}) = -1 - 0 = -1 \leq 0$	$(4,3)$ $(4,1)$ $(1,0)$
$(1,4)$ $(4,3)$ $(4,1)$	$\det(\begin{pmatrix} 0 \\ 2 \end{pmatrix}, \begin{pmatrix} -3 \\ 1 \end{pmatrix}) = 0 + 6 = 6 > 0$	$(1,4)$ $(4,3)$ $(4,1)$ $(1,0)$
$(0,2)$ $(1,4)$ $(4,3)$	$\det(\begin{pmatrix} -3 \\ 1 \end{pmatrix}, \begin{pmatrix} -1 \\ -2 \end{pmatrix}) = 6 + 1 = 7 > 0$	$(0,2)$ $(1,4)$ $(4,3)$ $(4,1)$ $(1,0)$