

TOC

Diese Zusammenfassungen wurden basierend auf den Vorlesungsfolien erstellt und liegen als **PDF-Dateien** vor. Sie dienen als stichwortartige Gedächtnisstütze und orientieren sich stark an den Inhalten der Folien.

Da der Fokus auf prägnanten Stichpunkten und der direkten Übernahme von Informationen aus den Slides lag, ist dies eher als eine **Stoffsammlung** zu verstehen, die die Struktur und viele Inhalte der Folien widerspiegelt. Meiner Meinung nach sind die **Slides bereits sehr gut aufbereitet**, um den Stoff zu lernen, und diese Zusammenfassungen dienen primär der schnellen Wiederholung und dem Überblick. Es wurde oft eine **1:1-Übernahme von Inhalten der Slides** vorgenommen.

Inhalt

1. Stable Matching Problem
2. Analyse von Algorithmen
3. Graphen
4. Greedy-Algorithmen
5. Divide und Conquer
6. Suchbäume
7. Hashing
7. Praktische Datenstrukturen

1. Stable Matching Problem

Einleitung

Gegeben:

- n Kinder und n Gastfamilien, die an einem Austauschprogramm für Schülerinnen und Schülern teilnehmen.
- Jedes Kind hat eine Präferenzliste von Gastfamilien.
- Jede Gastfamilie hat eine Präferenzliste von Kindern.

Ziel:

Finde eine passende Zuordnung von Kindern zu Gastfamilien.

Beispiel:

Kinder: Xaver, Yvonne, Zola.

Gastfamilien: Abel, Boole, Church.

Präferenzlisten der Kinder:

Name	1. (Höchste Präferenz)	2.	3. (Niedrigste Präferenz)
Xaver	Abel	Boole	Church
Yvonne	Boole	Abel	Church
Zola	Abel	Boole	Church

Präferenzlisten der Familien:

Name	1. Höchste Präferenz	2.	3. (Niedrigste Präferenz)
Abel	Yvonne	Xaver	Zola
Boole	Xaver	Yvonne	Zola
Church	Xaver	Yvonne	Zola

Perfektes Matching:

- Jedem Kind wird genau eine Familie zugewiesen.
- Jedes Kind bekommt genau eine Familie.
- Jede Gastfamilie bekommt genau ein Kind.

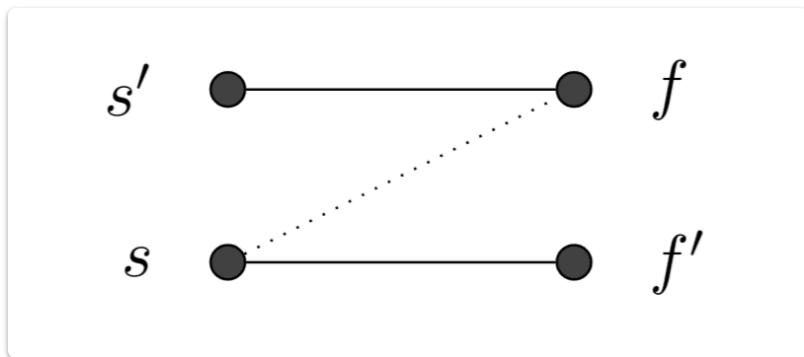
Instabiles Paar:

In einem Matching M ist ein nicht zugewiesenes Paar (s, f) **instabil**, wenn:

- Ein Kind s und eine Familie f sich gegenseitig gegenüber ihren aktuellen Partnern bevorzugen.
- Das instabile Paar (s, f) könnte eine Situation durch Verlassen der aktuellen Partner verbessern.

Notation:

- s für student (Kind)
- f für family (Gastfamilie)

Darstellung eines instabilen Paares:

Hier bevorzugt s die Familie f gegenüber seiner aktuellen Familie f' , und gleichzeitig bevorzugt die Familie f das Kind s gegenüber ihrem aktuellen Kind s' .

Definition:

Stabiles Matching: Ein perfektes Matching ohne instabile Paare. Es besteht daher kein Paar, das einen Anreiz hätte, durch gemeinsames Handeln die Zuteilung zu unterlaufen.

Stable-Matching-Problem: Ausgehend von den Präferenzlisten von n Kindern und n Familien, finde ein stabiles Matching, wenn es existiert.

Stable-Matching-Problem: Fragen

Frage: Gibt es immer ein stabiles Matching?

Hinweis: Das ist nicht von vornherein klar!

Frage: Kann ein stabiles Matching effizient gefunden werden?

Hinweis: Der Brute-Force-Ansatz (alle möglichen Zuordnungen ausprobieren) betrachtet $n!$ viele mögliche Lösungen, was extrem ineffizient ist.

Gale-Shapley-Algorithmus: Algorithmus mit dem wir beide Fragen mit „Ja“ beantworten können.

Gale–Shapley-Algorithmus (GS-Algorithmus)

```

Kennzeichne jede Familie/jedes Kind als frei
while ein Kind ist frei und kann noch eine Familie wählen
    Wähle solch ein Kind s aus
    f ist erste Familie in der Präferenzliste von s,
    die von s noch nicht gewählt wurde
    if f ist frei
        Kennzeichne s und f als einander zugeordnet
    elseif f bevorzugt s gegenüber ihrem aktuellen Partner s'
        Kennzeichne s und f als einander zugeordnet und s' als frei
    else
        f weist s zurück
    
```

Beispiel für Ablauf

Ausgangssituation:

- 4 Kinder (W-Z) mit Präferenzlisten (links).
- 4 Familien (A-D) mit Präferenzlisten (rechts).

	1.	2.	3.	4.
W	B	A	C	D
X	C	B	A	D
Y	B	C	A	D
Z	B	A	D	C

	1.	2.	3.	4.
A	X	W	Y	Z
B	W	Y	X	Z
C	Z	Y	W	X
D	X	W	Y	Z

Erste Zuordnung:

- Wähle erstes freies Kind aus (z.B. W).
- Dieses wählt die erste Familie in seiner Präferenzliste aus (in diesem Fall B).
- Da B frei ist, werden die beiden einstweilig einander zugeordnet.
- Aktuelle Zuordnungen: W-B.

	1.	2.	3.	4.
W	B	A	C	D
X	C	B	A	D
Y	B	C	A	D
Z	B	A	D	C

	1.	2.	3.	4.
A	X	W	Y	Z
B	W	Y	X	Z
C	Z	Y	W	X
D	X	W	Y	Z

Zweite Zuordnung:

- Wähle nächstes freies Kind aus (z.B. X).
- Dieses wählt die erste Familie in seiner Präferenzliste aus (in diesem Fall C).
- Da C frei ist, werden die beiden einstweilig einander zugeordnet.
- Aktuelle Zuordnungen: W-B, X-C.

	1.	2.	3.	4.
W	B	A	C	D
X	C	B	A	D
Y	B	C	A	D
Z	B	A	D	C

	1.	2.	3.	4.
A	X	W	Y	Z
B	W	Y	X	Z
C	Z	Y	W	X
D	X	W	Y	Z

Dritte Zuordnung:

- Wähle nächstes freies Kind aus (z.B. Y).
- Dieses wählt die erste Familie in seiner Präferenzliste aus (in diesem Fall B).
- B ist aber W zugeordnet. In der Präferenzliste von B steht W vor Y, daher lässt B das Y abblitzen.
- Y wählt die nächste Familie in seiner Präferenzliste aus (in diesem Fall C).
- C bevorzugt Y vor ihrem Partner X, daher wird ihre Zuordnung zu X gelöst und statt dessen werden C und Y einander zugeordnet.
- Aktuelle Zuordnungen: W-B, Y-C.

	1.	2.	3.	4.
W	B	A	C	D
X	C	B	A	D
Y	B	C	A	D
Z	B	A	D	C

	1.	2.	3.	4.
A	X	W	Y	Z
B	W	Y	X	Z
C	Z	Y	W	X
D	X	W	Y	Z

Vierte Zuordnung:

- Wähle nächstes freies Kind aus, es ist X, das wieder frei geworden ist.
- X wählt die zweite Familie (B) auf seiner Präferenzliste aus.
- B bevorzugt W vor X.
- X wählt die dritte Familie (A) auf seiner Präferenzliste aus.
- A ist frei und die beiden werden einander zugeordnet.
- Aktuelle Zuordnungen: W-B, X-A, Y-C.

	1.	2.	3.	4.
W	B	A	C	D
X	C	B	A	D
Y	B	C	A	D
Z	B	A	D	C

	1.	2.	3.	4.
A	X	W	Y	Z
B	W	Y	X	Z
C	Z	Y	W	X
D	X	W	Y	Z

Fünfte Zuordnung:

- Wähle nächstes freies Kind aus (nur mehr Z übrig).
- Z wählt die erste Familie (B) auf seiner Präferenzliste aus. B bevorzugt aber W vor Z.
- Z wählt die zweite Familie (A) auf seiner Präferenzliste aus. A bevorzugt aber X vor Z.
- Z wählt die dritte Familie (D) auf seiner Präferenzliste aus.
- D ist frei, also werden Z und D einander zugeordnet.
- Aktuelle Zuordnungen: W-B, X-A, Y-C, Z-D.
- Es ist nun kein Kind mehr frei, und der Algorithmus terminiert. Wir haben ein Stable Matching gefunden.

	1.	2.	3.	4.
W	B	A	C	D
X	C	B	A	D
Y	B	C	A	D
Z	B	A	D	C

	1.	2.	3.	4.
A	X	W	Y	Z
B	W	Y	X	Z
C	Z	Y	W	X
D	X	W	Y	Z

Beweis, dass der Algorithmus korrekt funktioniert:

Beweis, dass er Terminierte

Operation 1: Kinder wählen Familien in absteigender Reihenfolge aus.

Operation 2: Sobald eine Familie zugewiesen wurde, bleibt sie zugewiesen, die Zuteilung kann sich aber ändern.

Behauptung: Algorithmus terminiert nach höchstens n^2 Iterationen der while-Schleife.

Beweis: In jeder Iteration der while-Schleife wählt ein Kind eine Familie aus. Es gibt nur n^2 Möglichkeiten dafür. \square

	1.	2.	3.	4.	5.		1.	2.	3.	4.	5.	
Valentin	A	B	C	D	E		Abel	W	X	Y	Z	V
Werner	B	C	D	A	E		Boole	X	Y	Z	V	W
Xaver	C	D	A	B	E		Church	Y	Z	V	W	X
Yvonne	D	A	B	C	E		Dijkstra	Z	V	W	X	Y
Zola	A	B	C	D	E		Euler	V	W	X	Y	Z

$n(n - 1) + 1$ vorläufige Zuordnungen erforderlich

Beweis, dass alle Kinder und Familien zugewiesen werden:

Behauptung: Alle Kinder und Familien werden zugewiesen.

Beweis: (durch Widerspruch)

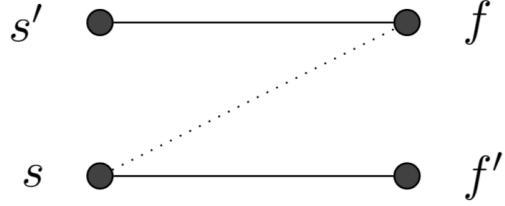
- Angenommen, Kind s wurde nach Terminierung des Algorithmus nicht zugewiesen.
- Dann wurde auch eine Familie (z.B. f) nach Terminierung des Algorithmus nicht zugewiesen.
- Damit wurde f nie ausgewählt.
- Aber s hat jede Familie in seiner Liste ausgewählt, da es ja am Ende nicht zugewiesen wurde.

Beweis: Stabilität

Behauptung: Nachdem der Algorithmus terminiert, existieren keine instabilen Paare.

Beweis: (durch Widerspruch)

Angenommen, (s, f) ist ein instabiles Paar: s bevorzugt f gegenüber seinem aktuellen Partner f' und f bevorzugt s gegenüber seinem aktuellen Partner s' in einem Gale-Shapley-Matching.



Fall 1: s hat f nie ausgewählt.

$\Rightarrow s$ bevorzugt seinen GS-Partner f' gegenüber f .

$\Rightarrow (s, f)$ ist nicht instabil.

Fall 2: s hat f ausgewählt.

$\Rightarrow f$ hat s zurückgewiesen (gleich oder später).

$\Rightarrow f$ bevorzugt s' gegenüber s .

$\Rightarrow (s, f)$ ist nicht instabil.

In jedem Fall ist (s, f) nicht instabil, was ein Widerspruch ist. \square

Wichtige Beobachtungen:

- Kinder wählen Familien in absteigender Reihenfolge der Präferenzen aus.
- Bei Familien kann sich die Situation nur verbessern.

Effiziente Implementierung

Zeitaufwand:

- Das Stable Matching benötigt höchstens n^2 Iterationen.
- Einzelne Schritte in einer Iteration können naiv (z.B. lineare Suche in Listen) implementiert werden, was eine Größenordnung von n Schritten benötigt.
- Dann benötigt man insgesamt höchstens eine Größenordnung von n^3 Schritten.
- Das ist immer noch besser, als ein Brute-Force-Ansatz, der alle möglichen Zuordnungen durchprobiert (es gibt $n!$ mögliche Zuordnungen).

Nächste Vorlesung:

- Mit asymptotischer Analyse können wir das exakt ausdrücken.
- Mit besseren Datenstrukturen können wir das auch schneller lösen.

2. Analyse von Algorithmen

Einleitung

Analyse von Algorithmen

Maße für die Effizienz von Algorithmen:

- Laufzeit
- Speicherplatz
- Besondere charakterisierende Parameter (problemabhängig)

Beispiel für problemabhängige Parameter: Sortieren

- Anzahl der Vergleichsoperationen
- Anzahl der Bewegungen der Datensätze

Laufzeit: Maschinenmodell

Laufzeit: Für die Betrachtung müssen wir eine Maschine festlegen, auf der wir rechnen.

RAM (Random Access Machine):

- Es gibt genau einen Prozessor.
- Alle Daten liegen im Hauptspeicher.
- Die Speicherzugriffe dauern alle gleich lang.

Laufzeit: Primitive Operationen

Zeit für eine Operation: Wenn nichts Genaueres spezifiziert wird, dann zählen wir die primitiven Operationen eines Algorithmus als jeweils eine Zeiteinheit.

Primitive Operationen:

- Zuweisungen: $a \leftarrow b$
- Arithmetische Befehle: $a \leftarrow b \circ c$ mit $\circ \in \{+, -, *, /, mod, \dots\}$
- Logische Operationen: $\wedge, \vee, \neg, \dots$
- Vergleichsoperatoren (z.B. bei Verzweigungen): `if a \circ b else ... mit`
 $\circ \in \{<, \leq, =, \neq, >, \geq\}$

Wir vernachlässigen:

- Indexrechnung
- Typ der Operanden

- Länge der Operanden

Laufzeit eines Algorithmus

Laufzeit:

- Ist die Anzahl der vom Algorithmus ausgeführten primitiven Operationen.
- Die Laufzeit $T(n)$ wird als Funktion der Eingabegröße n beschrieben.

Beispiel: Sortieren von 1000 Zahlen dauert länger als das Sortieren von 10 Zahlen.

Instanz: Eine Eingabe wird auch als Instanz (*instance*) des Problems, das durch den Algorithmus gelöst wird, bezeichnet.

Laufzeitanalyse

Worst-Case-Laufzeit: Ist die *größtmögliche* Laufzeit eines Algorithmus bei einer Eingabe mit Größe n .

- Erfasst allgemein die Effizienz in der Praxis.
- Pessimistische Sichtweise, eine Alternative ist aber schwer zu finden.

Average-Case-Laufzeit: Ist die *durchschnittliche* Laufzeit eines Algorithmus über alle möglichen gültigen Eingaben mit Größe n .

- Es ist allerdings schwer (und oft unmöglich), reale Instanzen durch zufällige Verteilungen exakt zu modellieren.
- Algorithmen, die an bestimmte Eingabeverteilungen angepasst werden, können für andere Eingaben schlechte Ergebnisse liefern.

Best-Case-Laufzeit: Ist die Laufzeit eines Algorithmus bei der *bestmöglichen* Eingabe mit Größe n .

- Untere Schranke, die nur in Spezialfällen erreicht wird.

Wichtig: In allen drei Fällen wird die Laufzeit als *Funktion der Eingabegröße n* angegeben.

Konventionen für Pseudocode

Schlüsselwörter:

- Schlüsselwörter werden **fett** und *hervorgehoben* dargestellt (z.B. **while**, **for**, **if**).

Zuweisung:

- Mit \leftarrow (z.B. $i \leftarrow 1$).

Vergleich:

- Mit $<$, \leq , $=$, \neq , $>$, \geq (z.B. **if** $x = 10$...).

Negation:

- Mit $!$ (z.B. **if** $! \text{finished}$...).

Bedingungen:

- Bedingungen werden nicht geklammert, wenn die Auswertung aus dem Kontext ersichtlich ist.
- Komplexe Bedingungen werden mit ganzen Sätzen beschrieben (z.B. **while** ein Kind ist frei und kann noch eine Gastfamilie auswählen ...).

Blockstruktur:

- Es werden keine Klammern verwendet.
- Alles auf der gleichen Einrückungsebene gehört zum selben Block.
- Wenn notwendig, werden mehrere Anweisungen durch Beistrich getrennt in eine Zeile geschrieben.

Bei Übungen und bei der Prüfung:

- Sie können Zuweisungen oder Vergleiche auch sprachlich beschreiben.
- Es wird empfohlen, zusätzliche Klammern bei Blöcken zu verwenden. Die Zugehörigkeit einer Anweisung zu einem bestimmten Block muss jedenfalls erkennbar sein.

Wichtigster Punkt: Struktur und Ablauf des Algorithmus müssen erkennbar sein!

Funktionen:

- Bei ausgewählten wichtigen Algorithmen werden die Funktionsnamen an den Anfang gestellt (mit etwaigen Parametern) angegeben (z.B. **BFS(s)**: ...).
- Arrays werden immer *per Referenz* übergeben (d.h. Änderungen am Arrayinhalt in einer Funktion sind auch außerhalb der Funktion sichtbar).

- Wenn weitere Parameter per Referenz übergeben werden, dann wird das explizit angegeben.

Speicher:

- Wir gehen davon aus, dass nicht benötigter Speicher automatisch freigegeben wird (*garbage collection*).
- Es wird daher nie explizit die Freigabe von Speicher angegeben.

Beispiel

Beispiel:

- Array A mit n Elementen gegeben.
- Ermittle die Summe aller Elemente, die größer als x sind und gib die Summe aus.

```
Sum(A, x):
sum ← 0
for i ← 0 bis n - 1
    if A[i] > x
        sum ← sum + A[i]
Gib sum aus
```

Anweisung	Konstante Kosten	Wie oft ausgeführt?
$sum \leftarrow 0$	c_1	1
for $i \leftarrow 0$ bis $n - 1$	c_2	n
if $A[i] > x$	c_3	n
$sum \leftarrow sum + A[i]$	c_4	j mit $0 \leq j \leq n$
Gib sum aus	c_5	1

Gesamter Aufwand $T(n)$:

$$\begin{aligned} T(n) &= c_1 \cdot 1 + c_2 \cdot n + c_3 \cdot n + c_4 \cdot j + c_5 \cdot 1 \\ &= (c_2 + c_3) \cdot n + c_4 \cdot j + (c_1 + c_5) \end{aligned}$$

Hierbei stellen c_1, c_2, c_3, c_4, c_5 konstante Kosten für verschiedene primitive Operationen dar. Die Laufzeit $T(n)$ hängt also nicht nur von der Eingabegröße n ab, sondern auch von dem Wert der Variablen j .

Best-Case: $j = 0$, d.h.

$$\begin{aligned} T(n) &= (c_2 + c_3) \cdot n + c_4 \cdot 0 + (c_1 + c_5) \\ &= (c_2 + c_3) \cdot n + (c_1 + c_5) \\ &= a_1 n + b_1 \quad \text{für Konstanten } a_1 \text{ und } b_1 \end{aligned}$$

Im besten Fall ist der Wert von j so, dass er die Laufzeit minimiert. In diesem Beispiel wird angenommen, dass $j = 0$ den geringsten Einfluss auf die Gesamlaufzeit hat. Dadurch reduziert sich der Term $c_4 \cdot j$ auf null, und die Laufzeit ist eine lineare Funktion von n ($a_1 n + b_1$).

Worst-Case: $j = n$, d.h.

$$\begin{aligned}
 T(n) &= (c_2 + c_3) \cdot n + c_4 \cdot n + (c_1 + c_5) \\
 &= (c_2 + c_3 + c_4) \cdot n + (c_1 + c_5) \\
 &= a_2 n + b_2 \quad \text{für Konstanten } a_2 \text{ und } b_2
 \end{aligned}$$

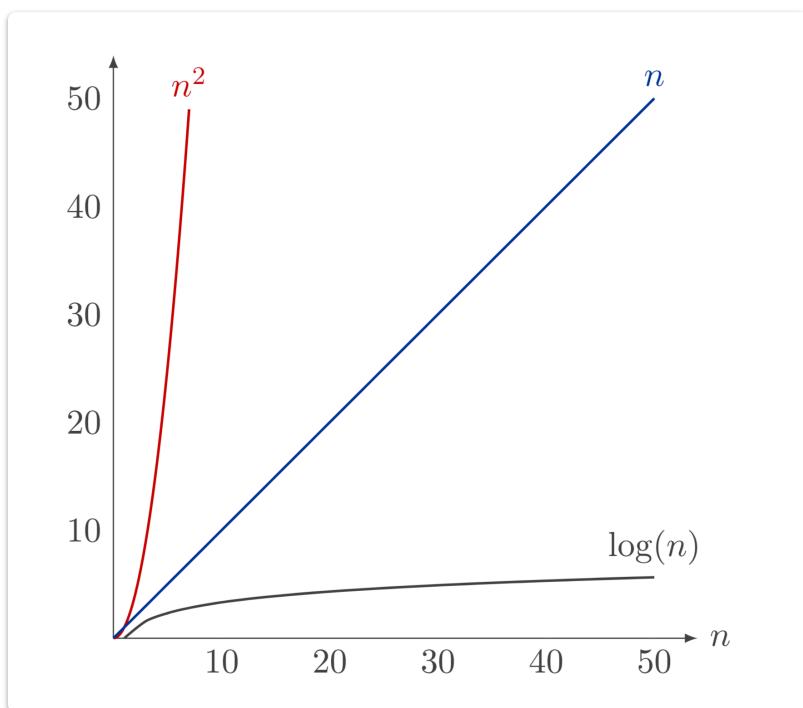
Im schlechtesten Fall ist der Wert von j so, dass er die Laufzeit maximiert. Hier wird angenommen, dass $j = n$ den größten Einfluss hat. Dadurch wird der Term $c_4 \cdot j$ zu $c_4 \cdot n$, und die Laufzeit bleibt eine lineare Funktion von n ($a_2 n + b_2$), wobei die Konstante a_2 möglicherweise größer ist als a_1 .

Zusammenfassend lässt sich sagen: Dieses Beispiel verdeutlicht, dass die genaue Laufzeit eines Algorithmus von der Eingabegröße (n) und möglicherweise von anderen Faktoren (hier j) abhängen kann. Die Best-Case- und Worst-Case-Analysen betrachten die extremen Werte dieser Faktoren, um die minimal und maximal mögliche Laufzeit in Abhängigkeit von der Eingabegröße zu bestimmen. In beiden betrachteten Fällen ist die Laufzeit linear in Bezug auf n , was bedeutet, dass die Laufzeit proportional zur Eingabegröße wächst.

Asymptotisches Wachstum

- Die Laufzeit wird als Funktion $T(n)$ in Abhängigkeit von der Eingabegröße n angegeben.
 - Eine exakte Berechnung der Laufzeit ist meist aber äußerst aufwendig bis praktisch unmöglich, auch wenn ein stark vereinfachtes Maschinenmodell angenommen wird.
 - Wir wollen vor allem Algorithmen unabhängig von konkreten Maschinenparametern vergleichen können.
 - Fokus auf Laufzeit $T(n)$ bei **großer** Eingabegröße n .
 - Unterschiede bei **kleinen** n oft irrelevant.
 - Ziel: Vergleich von Algorithmen **unabhängig** von Hardware.
 - Betrachtung des Wachstums von $T(n)$ für $n \rightarrow \infty$.
 - Vernachlässigung **konstanter Faktoren**.
 - Betrachtung des **dominannten Terms** in $T(n)$.
 - Eingabegröße n ist eine **nicht-negative ganze Zahl**.
-

Asymptotisches Wachstum bekannter Funktionen



Grundidee: Schranken

■ Schranken:

$$g(n) \leq T(n) \leq f(n)$$

■ Ignoriere konstante Faktoren:

$$c_1 \cdot g(n) \leq T(n) \leq c_2 \cdot f(n)$$

■ Ignoriere kleine n :

für $n > n_0$ gilt: $c_1 \cdot g(n) \leq T(n) \leq c_2 \cdot f(n)$

■ Schreibweise:

$$T(n) \in \Omega(g(n)), \quad T(n) \in O(f(n))$$

Obere Schranke (O)

Allgemein: Sei $f(n)$ eine Funktion. Dann bezeichnet $O(f(n))$ die **Menge aller Funktionen**, die asymptotisch durch $c \cdot f(n)$ (c ist eine positive Konstante) von oben beschränkt wird.

Definition: Eine Funktion $T(n)$ ist in $O(f(n))$, wenn **Konstanten** $c > 0$ und $n_0 > 0$ existieren, sodass für alle $n \geq n_0$ gilt:

$$T(n) \leq c \cdot f(n)$$

Notation: Eigentlich müsste man $T(n) \in O(f(n))$ schreiben. In der Praxis schreibt man aber:

- $T(n)$ ist in $O(f(n))$ oder kürzer
- $T(n) = O(f(n))$ (**Achtung:** Hier nicht als Gleichheit zu verstehen!)

Intuitive Erklärung: Die obere Schranke $O(f(n))$ gibt uns eine Vorstellung davon, wie schnell die Laufzeit eines Algorithmus im schlimmsten Fall wächst. Sie sagt aus, dass die Laufzeit für genügend große Eingaben niemals schneller wächst als ein konstantes Vielfaches von $f(n)$.

Beispiel für obere Schranke:

Beispiel: $T(n) = 32n^2 + 17n + 5$. Es ist zu zeigen, dass $T(n)$ in $O(n^2)$ ist.

Beweis:

- Wir zeigen, dass es Konstanten $c > 0$ und $n_0 > 0$ gibt, sodass $32n^2 + 17n + 5 \leq c \cdot n^2$ für alle $n \geq n_0$.
- Wir dividieren die Ungleichung durch n^2 und erhalten: $32 + \frac{17}{n} + \frac{5}{n^2} \leq c$.
- $32 + \frac{17}{n} + \frac{5}{n^2} \leq c$ gilt z.B. für alle $n \geq 18$, wenn $c \geq 34$ (dann sind die beiden Brüche jeweils < 1) oder für alle $n \geq 35$, wenn $c \geq 33$.
- Also können wir z.B. $n_0 = 18$ und $c = 34$ wählen.
- Daher können wir schreiben: $T(n) = O(n^2)$. \square

Untere Schranke (Ω)

Allgemein: Sei $f(n)$ eine Funktion. Dann bezeichnet $\Omega(f(n))$ die **Menge aller Funktionen**, die asymptotisch durch $c \cdot f(n)$ (c ist eine positive Konstante) von unten beschränkt wird.

Definition: Eine Funktion $T(n)$ ist in $\Omega(f(n))$, wenn **Konstanten** $c > 0$ und $n_0 > 0$ existieren, sodass für alle $n \geq n_0$ gilt:

$$T(n) \geq c \cdot f(n)$$

Intuitive Erklärung: Die untere Schranke $\Omega(f(n))$ gibt uns eine Vorstellung davon, wie langsam die Laufzeit eines Algorithmus im besten Fall mindestens wächst. Sie sagt aus, dass die Laufzeit für genügend große Eingaben niemals langsamer wächst als ein konstantes Vielfaches von $f(n)$.

Beispiel für untere Schranke

Beispiel: $T(n) = 32n^2 + 17n + 5$. Es ist zu zeigen, dass $T(n)$ in $\Omega(n^2)$ ist.

Beweis:

- Wir zeigen, dass es Konstanten $c > 0$ und $n_0 > 0$ gibt, sodass $32n^2 + 17n + 5 \geq c \cdot n^2$ für alle $n \geq n_0$.
- Wir dividieren die Ungleichung durch n^2 und erhalten: $32 + \frac{17}{n} + \frac{5}{n^2} \geq c$.
- $32 + \frac{17}{n} + \frac{5}{n^2} \geq c$ gilt z.B. für alle $n \geq 1$, wenn $c \leq 32$.
- Wir wählen daher z.B. $n_0 = 1$ und $c = 32$.
- Daher können wir schreiben: $T(n) = \Omega(n^2)$. \square

Scharfe Schranke (Θ)

Allgemein: Sei $f(n)$ eine Funktion. Dann bezeichnet $\Theta(f(n))$ die **Menge aller Funktionen**, die asymptotisch **gleich großes Wachstum** wie $c \cdot f(n)$ besitzen (c ist eine positive Konstante).

Definition: Eine Funktion $T(n)$ ist in $\Theta(f(n))$, wenn $T(n)$ sowohl in $O(f(n))$ als auch in $\Omega(f(n))$ ist.

Intuitive Erklärung: Die scharfe Schranke $\Theta(f(n))$ beschreibt das genaue asymptotische Wachstumsverhalten der Laufzeit eines Algorithmus. Sie bedeutet, dass die Laufzeit für genügend große Eingaben weder wesentlich schneller noch wesentlich langsamer wächst als ein konstantes Vielfaches von $f(n)$. Die Laufzeit ist in diesem Fall "eingeklemmt" zwischen zwei konstanten Vielfachen von $f(n)$.

Beispiel für scharfe Schranke

Beispiel: $T(n) = 32n^2 + 17n + 5$. Es ist zu zeigen, dass $T(n)$ in $\Theta(n^2)$ ist.

Beweis:

- Wir haben auf den vorherigen Folien schon gezeigt, dass $T(n)$ in $O(n^2)$ und in $\Omega(n^2)$ ist.
- Aus den beiden Aussagen für obere und untere Schranken folgt, dass $T(n)$ in $\Theta(n^2)$ ist. \square

Keine Schranke

Hier geht es darum wie man zeigt, dass etwas nicht in einer gewissen Schranke ist, das macht man mit einem Beweis durch Widerspruch:

Beispiel: $T(n) = 32n^2 + 17n + 5$, $T(n)$ ist **nicht** in $O(n)$.

Beweis: (durch Widerspruch)

- Angenommen, es gibt Konstanten $c > 0$ und $n_0 > 0$, sodass $32n^2 + 17n + 5 \leq c \cdot n$ für alle $n \geq n_0$. Wir formen um:

$$\begin{aligned} 32n^2 + 17n + 5 &\leq c \\ 32n^2 &\leq c - 17 \\ n^2 &\leq \frac{c-17}{32} \\ n &\leq \sqrt{\frac{c-17}{32}} \end{aligned}$$

- Das ist aber falsch für $n > \frac{c-17}{32}$. Widerspruch zur Annahme.
- Daher können wir **nicht** schreiben: $T(n) = O(n)$. \square

Weitere Beispiele: Ähnlich lässt sich z.B. zeigen, dass $T(n)$ nicht in $\Omega(n^3)$, $\Theta(n)$, oder $\Theta(n^3)$ ist.

Richtige Anwendung

Sinnlose Aussage: Jeder vergleichsbasierte Sortieralgorithmus für n Elemente benötigt zumindest $O(n \log n)$ Vergleiche.

- Es sollte Ω für die untere Schranke benutzt werden.

Nicht nur Laufzeit: Wir verwenden die asymptotische Notation nicht nur für die Laufzeit von Algorithmen, sondern auch für die Abschätzung verschiedener anderer Größen, z.B.:

- Speicherbedarf
- Rückgabewert von Funktionen
- Anzahl gewisser Operationen (wie Vertauschung von Elementen beim Sortieren)

Untere und obere Schranken

- **Definitionen:**

- Falls $f = O(g)$, dann gilt $g = \Omega(f)$.
- $f = \Omega(g)$ genau dann, wenn $g = O(f)$.
- $f = \Theta(g)$ genau dann, wenn $f = \Omega(g)$ und $f = O(g)$ gilt. Äquivalent dazu: $f = \Theta(g)$ genau dann, wenn $g = \Theta(f)$.

- **Beweis (Auszug für $f = O(g) \implies g = \Omega(f)$):**

- Annahme: $f = O(g)$.
- Daraus folgt: Es existieren Konstanten $c > 0$ und $n_0 > 0$, sodass für alle $n \geq n_0$ gilt: $f(n) \leq c \cdot g(n)$.
- Wir wählen als neue Konstante $c' = 1/c$. Da $c > 0$, ist auch $c' > 0$.
- Für alle $n \geq n_0$ gilt somit: $g(n) \geq \frac{1}{c} \cdot f(n) = c' \cdot f(n)$.
- Also gilt $g = \Omega(f)$.

Eigenschaft: Additivität

- **Obere Schranke:** Wenn $f = O(h)$ und $g = O(h)$, dann gilt $f + g = O(h)$.
 - $f + g$ bezeichnet die Funktion, die definiert ist durch $(f + g)(n) = f(n) + g(n)$.
- **Beweis (Obere Schranke):**
 - Da $f = O(h)$, existieren Konstanten $c > 0$ und $n_0 > 0$, sodass für alle $n \geq n_0$ gilt: $f(n) \leq c \cdot h(n)$.
 - Da $g = O(h)$, existieren Konstanten $c' > 0$ und $n'_0 > 0$, sodass für alle $n \geq n'_0$ gilt: $g(n) \leq c' \cdot h(n)$.
 - Wir wählen $n''_0 = \max(n_0, n'_0)$. Für alle $n \geq n''_0$ gelten beide Ungleichungen.
 - Daraus ergibt sich: $f(n) + g(n) \leq c \cdot h(n) + c' \cdot h(n) = (c + c') \cdot h(n)$.
 - Somit gilt $f(n) + g(n) \leq c'' \cdot h(n)$ für alle $n \geq n''_0$, wobei $c'' = c + c'$ eine positive Konstante ist.
 - Also ist $f + g = O(h)$.

Beispiel

Beispiel:

- $f(n) = 2n + 3 (= O(n^2))$, $g(n) = 5n^2 (= O(n^2))$
- $f(n) + g(n) = 5n^2 + 2n + 3 (= O(n^2))$

Weiteres Beispiel:

- Angenommen, ein Algorithmus besteht aus zwei Teilen A und B , die hintereinander ausgeführt werden.
- Die Ausführung von A benötigt $O(n^2)$ Zeit.
- Die Ausführung von B benötigt $O(n^3)$ Zeit.
- Der gesamte Algorithmus benötigt dann $O(n^3)$ Zeit.

Hinweis: Das gilt auch für die Summe von mehreren (aber konstant vielen) Funktionen.

Untere und scharfe Schranke

- **Untere Schranken:** Mit ähnlichen Argumenten wie für die obere Schranke kann man zeigen:
 - Wenn $f = \Omega(h)$ oder $g = \Omega(h)$, dann gilt $f + g = \Omega(h)$.
- **Scharfe Schranken:** Aus den beiden Aussagen für obere und untere Schranken folgt:
 - Wenn $f = \Theta(h)$ und $g = \Theta(h)$, dann gilt $f + g = \Theta(h)$.

Anwendung

- **Eine Anwendung:** Wenn $g = O(f)$, dann gilt $f + g = \Theta(f)$.
- **Beweis:**
 - Da laut Annahme $g = O(f)$ und klarerweise $f = O(f)$ (mit $c = 1$ und beliebigem $n_0 \geq 0$), folgt mittels der Additivitätseigenschaft für obere Schranken: $f + g = O(f)$.
 - Es gilt aber auch $f + g = \Omega(f)$, da für alle $n \geq 0$ gilt: $f(n) + g(n) \geq f(n)$ (da Funktionswerte asymptotisch nicht negativ sind). Wir können also $c = 1$ und $n_0 = 0$ wählen.
 - Daraus folgt $f + g = \Theta(f)$, da $f + g = O(f)$ und $f + g = \Omega(f)$.

Summenbeispiel nochmal:

Hier betrachten wir das Beispiel vom Anfang ([Summenbeispiel](#)) nochmal mit neu erworbenen Wissen an:

Beispiel:

- Array A mit n Elementen gegeben.
- Ermittle die Summe aller Elemente, die größer als x sind und gib die Summe aus.

```

Sum(A, x):
sum ← 0
for i ← 0 bis n - 1
    if A[i] > x
        sum ← sum + A[i]
Gib sum aus

```

- Wir haben eine Laufzeit von $T(n) = a_2n + b_2$ für Konstanten a_2 und b_2 bestimmt. Es gilt also $T(n) = \Theta(n)$.
- Diese asymptotische Abschätzung können wir auch einfacher erreichen:
- Die Schleife wird n mal ausgeführt, der Aufwand im Schleifenkörper sowie die Initialisierung sind konstant $\Theta(1)$. Es ergibt sich die Gesamlaufzeit $\Theta(n)$.

Transitivität

- **Obere Schranken:** Wenn $f = O(g)$ und $g = O(h)$, dann gilt $f = O(h)$.
- **Beweis:**
 - Da $f = O(g)$, existieren Konstanten $c > 0$ und $n_0 > 0$, sodass für alle $n \geq n_0$ gilt: $f(n) \leq c \cdot g(n)$.
 - Da $g = O(h)$, existieren Konstanten $c' > 0$ und $n'_0 > 0$, sodass für alle $n \geq n'_0$ gilt: $g(n) \leq c' \cdot h(n)$.
 - Wir wählen $n''_0 = \max(n_0, n'_0)$ und $c'' = c \cdot c'$. Beide sind positive Konstanten.
 - Damit ergibt sich für alle $n \geq n''_0$:
$$f(n) \leq c \cdot g(n) \leq c \cdot (c' \cdot h(n)) = (c \cdot c') \cdot h(n) = c'' \cdot h(n).$$
 - Also gilt $f(n) \leq c'' \cdot h(n)$ für alle $n \geq n''_0$, was bedeutet, dass $f = O(h)$.
- **Beispiel:**
 - $f(n) = n$, $g(n) = n^2$, $h(n) = n^3$
 - Es gilt $f(n) = O(g(n))$ (da $n \leq 1 \cdot n^2$ für $n \geq 1$) und $g(n) = O(h(n))$ (da $n^2 \leq 1 \cdot n^3$ für $n \geq 1$).
 - Folglich gilt auch $f(n) = O(h(n))$ (da $n \leq 1 \cdot n^3$ für $n \geq 1$).
- **Untere Schranken:** Mit ähnlichen Argumenten wie für obere Schranken kann man zeigen:
 - Wenn $f = \Omega(g)$ und $g = \Omega(h)$, dann gilt $f = \Omega(h)$.
- **Scharfe Schranken:** Aus den beiden Aussagen für obere und untere Schranken folgt:
 - Wenn $f = \Theta(g)$ und $g = \Theta(h)$, dann gilt $f = \Theta(h)$.

Asymptotische Äquivalenz

- **Äquivalenz:** Die binäre Relation $f = \Theta(g)$ zwischen Funktionen bildet eine Äquivalenzrelation.

- **Beweis:**

- Wir haben schon gezeigt:
 - $f = \Theta(g)$ gilt genau dann, wenn $g = \Theta(f)$ gilt (Symmetrie).
 - $f = \Theta(f)$ (Reflexivität): Es existieren $c_1 = 1 > 0$ und $c_2 = 1 > 0$ sowie ein $n_0 \geq 0$ (z.B. $n_0 = 0$), sodass für alle $n \geq n_0$ gilt: $c_1 \cdot f(n) \leq f(n) \leq c_2 \cdot f(n)$.
 - Wenn $f = \Theta(g)$ und $g = \Theta(h)$, dann gilt $f = \Theta(h)$ (Transitivität).

Da die Relation Θ reflexiv, symmetrisch und transitiv ist, handelt es sich um eine Äquivalenzrelation.

Asymptotische Dominanz

- **Dominanz von Funktionen:**

- f wird von g dominiert, wenn $f = O(g)$ aber nicht $g = O(f)$ gilt.
- f und g gehören **nicht** zur gleichen Äquivalenzklasse bezüglich asymptotischen Wachstums.
- Wir schreiben dann $f \ll g$.

Alternative Sichtweise:

Dominanz: g dominiert f , wenn $\lim_{n \rightarrow \infty} f(n)/g(n) = 0$.

Beispiel: Funktion $g(n)$ dominiert $f(n)$

- $g(n) = n^3$ und $f(n) = n^2$
- $\lim_{n \rightarrow \infty} f(n)/g(n) = \lim_{n \rightarrow \infty} n^2/n^3 = \lim_{n \rightarrow \infty} 1/n = 0$

Beispiel: Keine Dominanz

- $g(n) = 2n^2$ und $f(n) = n^2$
- $\lim_{n \rightarrow \infty} f(n)/g(n) = \lim_{n \rightarrow \infty} n^2/2n^2 = \lim_{n \rightarrow \infty} 1/2 \neq 0$

Hinweis:

- Wir gehen meist **von stetigen Funktionen** aus.
- Es gibt Paare von Funktionen f und g , sodass weder f die Funktion g dominiert noch g die Funktion f dominiert (z.B. bei nicht stetigen Funktionen).

Allgemein gilt für Polynome: n^a dominiert n^b , wenn $a > b$, da

$$\lim_{n \rightarrow \infty} n^b/n^a = \lim_{n \rightarrow \infty} n^{b-a} = 0.$$

Asymptotische Schranken von gebräuchlichen Funktionen

Polynome:

Polynome: $f(n) = a_0 + a_1n + \dots + a_dn^d$ ist in $\Theta(n^d)$ wenn $a_d > 0$.

Beweis:

- Für jeden Koeffizienten a_j für $j < d$ gilt, dass $a_j n^j \leq |a_j| n^d$ für alle $n \geq 1$.
- Daher ist jeder Term in $O(n^d)$.
- Da $f(n)$ die Summe von einer konstanten Anzahl an Funktionen ist, ist auch $f(n)$ in $O(n^d)$ (wegen Additivitätseigenschaft).
- Da ein Summand in $f(n)$ in $\Omega(n^d)$ ist, ist $f(n)$ in $\Omega(n^d)$ (wegen Additivitätseigenschaft) und daher: $f(n) = \Theta(n^d)$. \square

Polynomialzeit: Laufzeit liegt in $O(n^d)$ für eine Konstante d , wobei d unabhängig von der Eingabegröße n ist.

Logarithmen:

Logarithmen: Für Konstanten $a, b > 0$ gilt $\Theta(\log_a n) = \Theta(\log_b n)$.

Beweis: Wir berücksichtigen folgende Identität

$$\log_a n = \frac{\log_b n}{\log_b a}$$

$\log_b a$ ist aber eine Konstante und daraus folgt, dass $\log_a n = \Theta(\log_b n)$. \square

Hinweis: Daher braucht bei asymptotischen Angaben die Basis von Logarithmen nicht angegeben werden.

Vergleich zu Polynomfunktionen: Für jede Konstante $\varepsilon > 0$ gilt, $\log n \ll n^\varepsilon$.

■ *log wächst asymptotisch langsamer als jede Polynomfunktion*

Exponentiell:

Exponentiell: Für alle Konstanten $c > 1$ und $d > 0$ gilt, $n^d \ll c^n$.

■ *Jede Exponentialfunktion wächst asymptotisch schneller als jede Polynomfunktion*

Hinweis: Für Konstanten $1 < c_1 < c_2$ gilt (im Gegensatz zu den Basen bei den Logarithmen) $c_1^n \ll c_2^n$.

Beweis: $c_1^n = O(c_2^n)$ ist klar. Angenommen $c_2^n = O(c_1^n)$. Daraus würde folgen, dass $c_2^n \leq c \cdot c_1^n$ für eine Konstante $c > 0$ gilt. Umgeformt ergibt das $(c_2/c_1)^n \leq c$, was aber nicht sein kann, da der Ausdruck $(c_2/c_1)^n$ gegen Unendlich geht (da wir ja $1 < c_1 < c_2$ angenommen haben). \square

Verhältnisse:

- **Verhältnisse:** Ordnung der Dominanz von Funktion $f(n)$ für $n \geq 0$ ($a \ll b$ bedeutet b dominiert a)

$$1 \ll \log n \ll \sqrt[k]{n} \ll n \ll n \log n \ll n^{1+\epsilon} \ll n^2 \ll n^3 \ll \dots \ll c^n \ll n! \ll n^n$$

- **Hinweise:**

- 1 bezeichnet die konstante Funktion $f(n) = 1$.
- $n^{1+\epsilon}$ für $0 < \epsilon < 1$.
- $c > 1$ (konstante Basis exponentiellen Wachstums).
- $k > 3$

Laufzeiten einiger gebräuchlicher Funktionen

Konstantes Wachstum: $O(1)$

- **Konstantes Wachstum:** Unabhängig von der Eingabegröße n ist der Aufwand für eine Operation konstant.
- **Hinweis:** Konstant bedeutet hier nicht, dass die Operation auf unterschiedlichen Systemen gleich viel Aufwand benötigt. Vielmehr wird sich der Aufwand auf unterschiedlichen Systemen sehr wohl unterscheiden. Auf einem System wird aber der Aufwand für die Operation unabhängig von n gleich bleiben.
- **Beispiele:**
 - Addition zweier Zahlen
 - Zugriff auf das i -te Element in einem Array der Größe n .

Logarithmisches Wachstum: $O(\log n)$

- **Logarithmisches Wachstum:** Wenn z.B. die Eingabegröße in jedem Schritt halbiert wird.
- **Binäre Suche:** Binäre Suche nach einem gegebenen Wert in einem aufsteigend sortierten Array A

```
Ermittle den mittleren Index  $s$  im Array A
if  $A[s] =$  gesuchter Wert
    Gib aus, dass Wert gefunden wurde
elseif gesuchter Wert ist kleiner als  $A[s]$ 
    Wende binäre Suche auf das Teilarray links von  $s$  an
elseif gesuchter Wert ist größer als  $A[s]$ 
    Wende binäre Suche auf das Teilarray rechts von  $s$  an
```

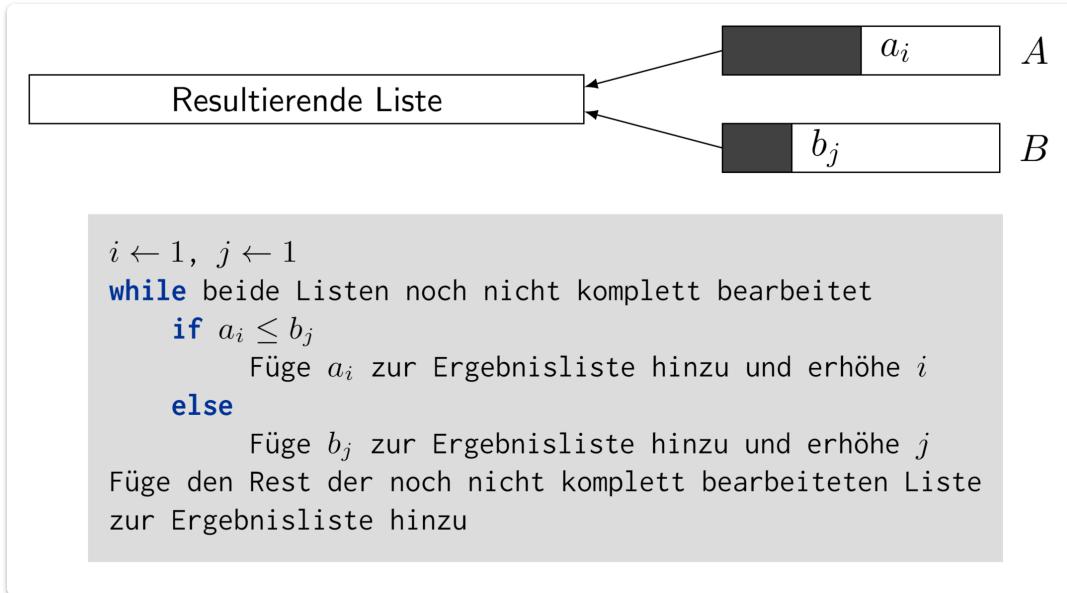
Lineares Wachstum: $O(n)$

- **Lineares Wachstum:** Laufzeit ist proportional zur Eingabegröße.
- **Maximumsuche:** Ermittle das Maximum von n Zahlen a_1, a_2, \dots, a_n .

```
 $max \leftarrow a_1$ 
for  $i \leftarrow 2$  bis  $n$ 
    if  $a_i > max$ 
         $max \leftarrow a_i$ 
```

Die Maximumsuche hat eine lineare Zeitkomplexität, da jedes Element des Arrays einmal betrachtet wird. Die Anzahl der Operationen wächst direkt mit der Anzahl der Elemente n .

- **Merge:** Verschmelze zwei sortierte Listen $A = a_1, a_2, \dots, a_n$ und $B = b_1, b_2, \dots, b_n$ zu einer sortierten Liste.



Der Merge-Prozess hat eine **lineare Zeitkomplexität**, da im schlimmsten Fall jedes Element der beiden Eingabelisten einmal in die Ergebnisliste eingefügt wird. Die Anzahl der Operationen ist proportional zur Summe der Längen der beiden Listen, also $O(n + m)$. Wenn beide Listen die Größe n haben, ist die Komplexität $O(2n) = O(n)$.

$O(n \log n)$ Wachstum

- **$O(n \log n)$ Laufzeit:** Tritt z.B. bei "teile und herrsche" (Divide-and-Conquer)-Algorithmen auf.
 - Wird auch als leicht überlinear bezeichnet.
- **Sortieren:**
 - Naives Sortieren wie z.B. Bubblesort hat eine Laufzeit von $O(n^2)$.
 - Schnelleres Sortieren ist möglich. Mergesort ist ein Sortieralgorithmus, der garantiert nur $O(n \log n)$ Vergleiche durchführt. Wird im Laufe dieser Vorlesung noch besprochen.
- **Beispiel - Größtes leeres Intervall:** Es seien n Zeitpunkte x_1, \dots, x_n gegeben, zu denen Kopien einer Datei am Server abgelegt werden. Wie groß ist das größte Intervall, in dem keine Kopien am Server ankommen?
- **$O(n \log n)$ Lösung:** Sortiere die Zeitpunkte. Durchlaufe die Liste in sortierter Reihenfolge und berechne das größte Intervall zwischen zwei aufeinanderfolgenden Zeitpunkten.
 - **Sortieren:** Das Sortieren der n Zeitpunkte benötigt typischerweise $O(n \log n)$ Zeit (z.B. mit Mergesort).
 - **Durchlaufen und Differenz bilden:** Das anschließende Durchlaufen der sortierten Liste und das Berechnen der Differenzen zwischen aufeinanderfolgenden Zeitpunkten benötigt $O(n)$ Zeit.
 - **Gesamlaufzeit:** Die Gesamlaufzeit ist somit $O(n \log n) + O(n) = O(n \log n)$.

Quadratisches Wachstum: $O(n^2)$

- **Quadratisches Wachstum:** Betrachte alle Paare von Elementen.
- **Dichtestes Punktpaar:** Gegeben sei eine Liste von n Punkten in einer Ebene $(x_1, y_1), \dots, (x_n, y_n)$ und es sollen die zwei am dichtesten beieinander liegenden Punkte gefunden werden.
- **$O(n^2)$ Lösung:** Überprüfe alle Paare von Punkten.

```

min ←  $(x_1 - x_2)^2 + (y_1 - y_2)^2$ 
for i ← 1 bis n - 1
  for j ← i + 1 bis n
    d ←  $(x_i - x_j)^2 + (y_i - y_j)^2$ 
    if d < min
      min ← d

```

□ Es muss nicht die Wurzel gezogen werden

- **Ablauf des Algorithmus:**

Wert für i	Werte für j	Anzahl der Durchläufe
1	2 … n	$n - 1$
2	3 … n	$n - 2$
…	…	…
$n - 1$	n	1

- **Summe:** $(n - 1) + (n - 2) + \dots + 1 = \frac{n(n-1)}{2} = \frac{n^2-n}{2} = \Theta(n^2)$.
- **Anmerkung:** $\Omega(n^2)$ erscheint unvermeidbar, aber es gibt für dieses Problem einen Algorithmus, der effizienter als der offensichtliche ist (z.B. ein $O(n \log n)$ Algorithmus basierend auf Divide and Conquer).

Kubisches Wachstum: $O(n^3)$

- **Kubisches Wachstum:** Zähle alle Dreiergruppen von Elementen.
- **Disjunkte Mengen:** Gegeben seien n Mengen S_1, \dots, S_n , wobei jede Menge eine Teilmenge von $\{1, 2, \dots, n\}$ ist. Existiert ein Paar von Mengen, das disjunkt ist?
- **$O(n^3)$ Lösung:** Bestimme für jedes Paar von Mengen, ob es disjunkt ist.

```

for i ← 1 bis n - 1
  for j ← i + 1 bis n
    foreach Element p von  $S_i$ 
      Bestimme ob p auch in  $S_j$  vorhanden ist
      if kein Element von  $S_i$  ist in  $S_j$  vorhanden
        Gib aus, dass  $S_i$  und  $S_j$  disjunkt sind

```

Die Komplexität ergibt sich, da wir $O(n^2)$ Paare von Mengen betrachten und für jedes Paar im schlimmsten Fall $O(n)$ Elemente vergleichen müssen (wenn die Mengen bis zu Größe n

haben können).

Polynomielle Laufzeit: $O(n^k)$ Wachstum

- **Teilsummenproblem mit k Zahlen:** Gegeben seien n verschiedene ganze Zahlen in einem Array A . Gibt es genau k Zahlen ($k < n$), sodass die Summe dieser Zahlen einer gegebenen Zahl x entspricht?
- **Beispiel: $O(n^4)$ Lösung für $k = 4$ mit Ausgabe**

```
for  $i \leftarrow 0$  bis  $n - 4$ 
    for  $j \leftarrow i + 1$  bis  $n - 3$ 
        for  $k \leftarrow j + 1$  bis  $n - 2$ 
            for  $l \leftarrow k + 1$  bis  $n - 1$ 
                if  $(A[i] + A[j] + A[k] + A[l]) = x$ 
                    Gib  $A[i]$ ,  $A[j]$ ,  $A[k]$  und  $A[l]$  aus
```

Die vier verschachtelten Schleifen iterieren über alle möglichen Kombinationen von vier Indizes i, j, k, l . Im schlimmsten Fall werden alle $O(n^4)$ Kombinationen überprüft.

- **Hinweis:** Dieses Problem kann man in der Regel effizienter lösen (oftmals mit dynamischer Programmierung oder anderen Techniken).

Exponentielles Wachstum: $O(c^n)$

- **Teilsummenproblem (Subset Sum):** Gegeben sei eine Menge von ganzen Zahlen. Gibt es eine Untermenge dieser Zahlen, deren Elementsumme einer vorgegebenen Zahl x entspricht?
- **Lösung:**

```
foreach Teilmenge  $S$  von Zahlen
    Überprüfe, ob die Summe der Elemente in  $S$  gleich  $x$  ist
```
- **Exponentielles Wachstum:** Eine endliche Menge mit n Elementen hat genau 2^n Teilmengen und jede dieser Teilmengen muss im schlimmsten Fall untersucht werden. Daher die exponentielle Laufzeit von $O(2^n)$.

Laufzeiten am Beispiel von Stable Matching

Gale–Shapley–Algorithmus

Hier nochmal der Algorithmus der in [Stable Matching Problem](#) schon vorgekommen ist:

```

Kennzeichne jede Familie/jedes Kind als frei
while ein Kind ist frei und kann noch eine Familie auswählen
    Wähle solch ein Kind  $s$  aus
     $f$  ist erste Familie in der Präferenzliste von  $s$ ,
    die  $s$  noch nicht ausgewählt hat
    if  $f$  ist frei
        Kennzeichne  $s$  und  $f$  als einander zugeordnet
    elseif  $f$  bevorzugt  $s$  gegenüber ihrem aktuellen Partner  $s'$ 
        Kennzeichne  $s$  und  $f$  als einander zugeordnet und  $s'$  als frei
    else
         $f$  weist  $s$  zurück

```

Wir wollen uns jetzt anschauen, wie man so etwas bestenfalls implementiert.

Implementierung von Stable Matching

- **Ausgangssituation:** Wir wissen, dass der Gale-Shapley-Algorithmus ein Stable Matching zwischen n Kindern und n Familien mit $\leq n^2$ Iterationen findet.
- **Ziel:** Wir möchten zeigen, dass der gesamte Algorithmus mit einer Laufzeit in $O(n^2)$ implementiert werden kann.
- **Überlegungen zur Datenstruktur:**
 - Jedes Kind und jede Familie hat eine Präferenzliste aller Mitglieder der anderen Gruppe. Wie repräsentiert man so eine Rangfolge?
 - Außerdem muss in jedem Schritt das aktuelle Matching gespeichert werden.
 - Wir werden zeigen, dass dafür Arrays und Listen ausreichen.

Array

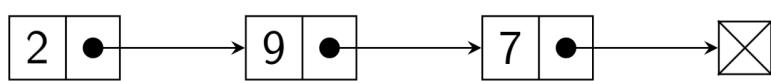
- **Array:**
 - Ist eine statische Datenstruktur mit im Speicher sequentiell abgelegten Elementen.
 - Alle Elemente haben den gleichen Typ.
 - Zugriff über Index in konstanter Zeit möglich ($O(1)$).
- **Beispiel:**

0	1	2	3	4	5	6	7
0	7	2	6	0	3	4	5

- **Zweidimensionale Arrays:** Zweidimensionale (oder mehrdimensionale) Arrays können als Array von Arrays realisiert werden.

Verkettete Liste

- **Einfach verkettete Liste:**
 - Ist eine dynamische Datenstruktur.
 - Speicherung von miteinander in Beziehung stehenden Knoten (durch Zeiger auf nachfolgende Knoten).
 - Anzahl der Knoten muss im Vorhinein nicht bekannt sein.
- **Beispiel:**



- **Implementierungsdetails:** Siehe VU Einführung in die Programmierung 2.

Arrays und Listen

- **Typische Operationen:** Worst-Case-Komplexität von Operationen auf einem unsortierten Array und einer unsortierten einfach verketteten Liste.
- **Tabelle: Vergleich von Operationen auf Arrays und auf Listen**

Operation	Array	Liste
Beliebiges Element suchen	$\Theta(n)$	$\Theta(n)$
Auf Element mit Index i zugreifen	$\Theta(1)$	$\Theta(n)$
Element am Anfang einfügen	$\Theta(n)$	$\Theta(1)$

Genaueres zu Arrays und Listen [Liste](#) und [Array](#)

Grundlegende Datenstruktur

- **Vereinfachung:**
 - Es gibt sowohl n Kinder und n Familien.
 - Kinder und Familien werden jeweils mit einer Nummer ($0, \dots, n - 1$) assoziiert.
 - Damit kann man ein Array anlegen, dessen Indexwerte sich aus diesen Nummern ergeben.
- **Präferenzlisten:**
 - Jedes Kind und jede Familie hat eine Präferenzliste in Form eines Arrays.
 - $SPräf[s, i]$ ist die Familie mit Index i in der Liste von Kind s (diese Familie hat bei Kind s den Rang $i + 1$).
 - $FPräf[f, i]$ ist das Kind mit Index i in der Liste der Familie f (dieses Kind hat bei Familie f den Rang $i + 1$).

- **Platzbedarf:** Es gibt n Kinder und n Familien, beide haben je ein Array der Länge n , daher $\Theta(n^2)$ (wie bei Laufzeit asymptotisch abgeschätzt).

Beispiel

- **Beispiel allgemein:**

	1.	2.	3.
Xaver	Abel	Boole	Church
Yvonne	Boole	Abel	Church
Zola	Abel	Boole	Church

- **Mapping für Array SPräf:** Xavier = 0, Yvonne = 1, Zola = 2, Abel = 0, Boole = 1, Church = 2

	0	1	2
0	0	1	2
1	1	0	2
2	0	1	2

Schritte der Iteration:

Erster Schritt: Ein freies Kind finden.

Lösung:

- Verkettete Liste SFree mit allen freien Kindern. Zu Beginn enthält SFree alle Kinder. (Aufwand der Initialisierung $\Theta(n)$)
- Das erste Element s wird ausgewählt (konstanter Aufwand).
- s wird aus der Liste gelöscht und möglicherweise ein s' (wenn ein anderes Kind s' wieder frei wird) in die Liste an der ersten Stelle eingefügt.
- Dieser Schritt kann in konstanter Zeit ausgeführt werden.

Zweiter Schritt: Man muss für ein Kind jene Familie mit dem höchsten Rang finden, die es noch nicht ausgewählt hat.

Lösung:

- Dazu wird ein Array Next benutzt, das für jedes Kind die Position (Index in einem SPref-Array) der nächsten auszuwählenden Familie angibt.
- Zunächst wird für jedes Kind s das Array mit $\text{Next}[s] = 0$ initialisiert. (Aufwand proportional zu n)
- Wenn ein Kind s eine Familie auswählen möchte, dann wählt es die Familie $f = \text{SPref}[s, \text{Next}[s]]$ aus.
- Wird eine Familie ausgewählt, dann wird $\text{Next}[s]$ um 1 erhöht (unabhängig vom Ergebnis).
- Alle Operationen benötigen konstante Zeit und daher kann dieser Schritt in konstanter Zeit ausgeführt werden.

Dritter Schritt: Für eine Familie f müssen wir entscheiden, ob f schon zugewiesen wurde und wer das zugewiesene Kind ist.

Lösung:

- Wir verwenden ein Array Current der Länge n .
- $\text{Current}[f]$ ist der Partner von f .
- Hat f keinen Partner, dann wird das durch eine spezielle Zahl (z.B. -1) angezeigt.
- Am Anfang werden alle Einträge des Arrays auf die spezielle Zahl gesetzt. (Aufwand proportional zu n)
- Dieser Schritt kann daher auch in konstanter Zeit durchgeführt werden.

Vierter Schritt: Für eine Familie f und zwei Kinder s und s' müssen wir entscheiden, ob s oder s' von f bevorzugt wird.

Lösung:

- Dazu wird am Anfang ein $n \times n$ Array Ranking erstellt.
- $\text{Ranking}[f, s]$ enthält den Rang von Kind s in der sortierten Reihenfolge von f .
- Für jede Familie muss daher nur die Präferenzliste einmal durchlaufen werden.
- Der Aufwand dafür ist proportional zu n^2 , aber die Erstellung von Ranking wird vor(!) dem eigentlichen Algorithmus ausgeführt.
- Bei einer Iteration des Algorithmus müssen daher nur mehr die zwei Einträge $\text{Ranking}[f, s]$ und $\text{Ranking}[f, s']$ verglichen werden.
- Damit ist auch dieser Schritt in konstanter Zeit ausführbar.

Erstellung des Arrays Ranking

Für jede Familie (Zeile): Erzeuge eine **inverse Liste** der Präferenzliste der Familien.

Beispiel: Abel (hat Nummer 0)

Präferenz	0	1	2	3	4	5	6	7
0	7	2	6	0	3	4	5	1
Ranking	0	1	2	3	4	5	6	7
0	3	7	1	4	5	6	2	0

Abel bevorzugt Kind 2 gegenüber 5 da $\underbrace{\text{Ranking}[0, 2]}_1 < \underbrace{\text{Ranking}[0, 5]}_6$

```

for  $i \leftarrow 0$  bis  $n - 1$ 
  for  $j \leftarrow 0$  bis  $n - 1$ 
     $\text{Ranking}[i, \text{FPref}[i, j]] \leftarrow j$ 
  
```

Laufzeit und Implementierung

- **Laufzeit:** In der Initialisierungsphase werden zwei $\Theta(n^2)$ Arrays erstellt. Die Laufzeit dieser Phase liegt in $O(n^2)$. Alle vier Schritte der Iteration (des Gale-Shapley-Algorithmus, vermutlich) lassen sich in konstanter Zeit ausführen. Daher liegt die Gesamlaufzeit für den Algorithmus in $O(n^2)$.
- **Implementierung:** Vom abstrakten Pseudocode zu einer Beschreibung in genauerem Pseudocode ist aber noch Einiges an Arbeit zu leisten!

Pseudocode mit Arrays

Gegeben: Arrays SPref und FPref

```

for  $i \leftarrow 0$  bis  $n - 1$ 
    for  $j \leftarrow 0$  bis  $n - 1$ 
         $\text{Ranking}[i, \text{FPref}[i, j]] \leftarrow j$ 
for  $i \leftarrow 0$  bis  $n - 1$ 
     $\text{Next}[i] \leftarrow 0$ 
     $\text{Current}[i] \leftarrow -1$ 
 $\text{SFree} \leftarrow$  Liste aller Kinder
while  $\text{SFree}$  ist nicht leer
     $s \leftarrow$  erstes Element aus  $\text{SFree}$ , lösche erstes Element aus  $\text{SFree}$ 
     $f \leftarrow \text{SPref}[s, \text{Next}[s]]$ 
     $s' \leftarrow \text{Current}[f]$ 
    if  $s' = -1$ 
         $\text{Current}[f] \leftarrow s$ 
    elseif  $\text{Ranking}[f, s] < \text{Ranking}[f, s']$ 
         $\text{Current}[f] \leftarrow s$ 
        Füge  $s'$  in  $\text{SFree}$  an erster Stelle ein
    else
        Füge  $s$  in  $\text{SFree}$  an erster Stelle ein
     $\text{Next}[s] \leftarrow \text{Next}[s] + 1$ 

```

Sortieren als praktisches Beispiel

Sortieren: Gegeben sind n Elemente die aufsteigend angeordnet werden sollen.

- **Sortiere eine Liste von Namen**
- Organisiere eine MP3-Bibliothek
- Zeige Google PageRank Resultate an
- Liste RSS-Feedelemente in umgekehrt chronologischer Reihenfolge auf
- Finde den Median
- Finde das nächste Paar
- Binäre Suche

Rahmenbedingungen und Annahmen

- **Internes Sortieren:** Alle Daten sind im Hauptspeicher.
- **Datenstruktur:** Array mit n Elementen, $A[0], \dots, A[n - 1]$, auf denen eine Ordnungsrelation " \leq " definiert ist.
- **Hinweis:** Aus Gründen der Einfachheit gehen wir in den folgenden Beispielen von einem Array mit n Werten aus, die sortiert werden sollen. In der Praxis können bei diesen Werten noch zusätzliche Informationen vorhanden sein, d.h. die Werte dienen als Schlüssel zum Auffinden von Informationen.

Elementare Sortierverfahren

- **Bubblesort:**
 - Aus VU Einführung in die Programmierung 1 bekannt.
 - Laufzeit liegt im Worst- und Average-Case in $\Theta(n^2)$. Im Best-Case kann die Laufzeit bei optimierten Implementierungen in $\Theta(n)$ liegen.
- **Weitere Beispiele für elementare Verfahren:**
 - Sortieren durch Minimumssuche (Selectionsort)
 - Sortieren durch Einfügen (Insertionsort)

Selectionsort (Sortieren durch Minimumssuche)

- **Selectionsort:** Selectionsort sucht in jeder Iteration das kleinste Element in der noch unsortierten Teilfolge, entnimmt es und fügt es dann an die sortierte Teilfolge an.
- **Beispiel:** Implementierung mit einem Array A mit n Elementen.

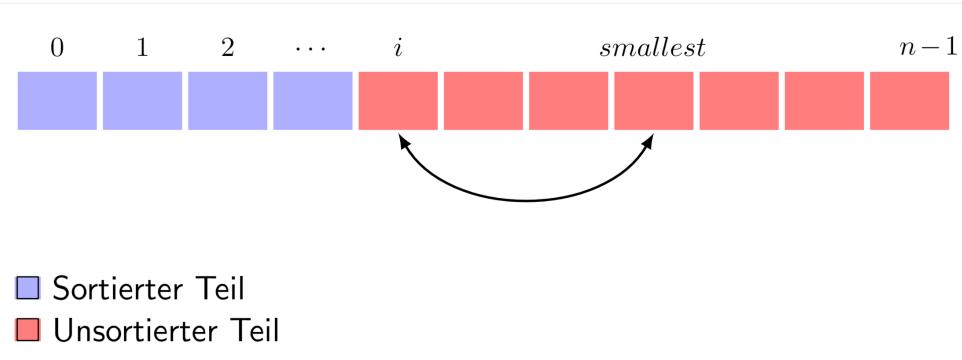
Selectionsort(A):

```

for  $i \leftarrow 0$  bis  $n - 2$ 
     $smallest \leftarrow i$ 
    for  $j \leftarrow i + 1$  bis  $n - 1$ 
        if  $A[j] < A[smallest]$ 
             $smallest \leftarrow j$ 

```

Vertausche $A[i]$ mit $A[smallest]$



Beispiel

- Ausgangssequenz: 5, 2, 4, 3, 1.
- Minimum im jeweiligen Durchlauf eingekreist.
- Die sortierte Teilfolge wird von links her aufgebaut.

$i = 0$	5	2	4	3	1
$i = 1$	1	(2)	4	3	5
$i = 2$	1	2	4	(3)	5
$i = 3$	1	2	3	(4)	5
Ende	1	2	3	4	5

Analyse

- **Laufzeit:** Die Laufzeit liegt immer (Best/Average/Worst-Case) in $\Theta(n^2)$.
- **Analyse:**
 - Die innere Schleife wird beim ersten Durchlauf der äußeren Schleife $n - 1$ -mal ausgeführt.
 - Im nächsten Durchlauf $n - 2$, dann $n - 3$ -mal usw.
 - $(n - 1) + (n - 2) + (n - 3) + \dots + 2 + 1 = \frac{n(n-1)}{2} = \frac{n^2-n}{2} = \Theta(n^2)$.

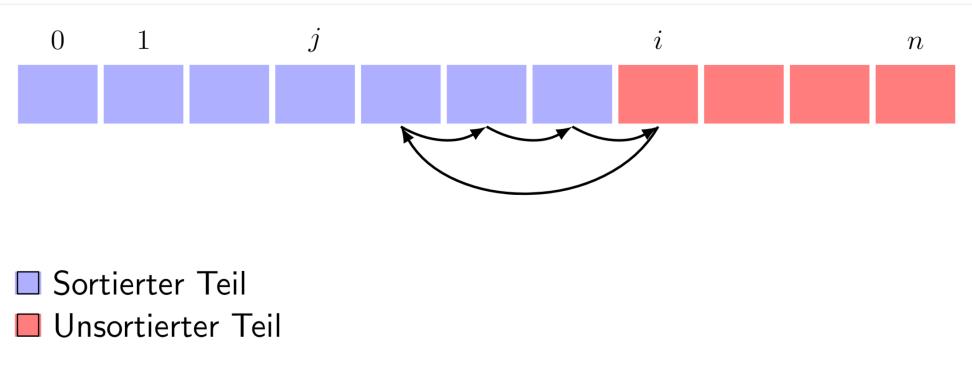
- **Anzahl der Vertauschungen:** Eine Vertauschung pro äußerem Schleifendurchlauf, d.h. $\Theta(n)$.

Insertionsort (Sortieren durch Einfügen)

- **Insertionsort:** Insertionsort entnimmt der unsortierten Teilfolge ein Element und fügt es an der richtigen Stelle in die (anfangs leere) sortierte Teilfolge ein.
- **Beispiel:** Implementierung mit einem Array A mit n Elementen.

Insertionsort(A):

```
for  $i \leftarrow 1$  bis  $n - 1$ 
     $key \leftarrow A[i]$ ,  $j \leftarrow i - 1$ 
    while  $j \geq 0$  und  $A[j] > key$ 
         $A[j + 1] \leftarrow A[j]$ 
         $j \leftarrow j - 1$ 
     $A[j + 1] \leftarrow key$ 
```



Beispiel

- Ausgangssequenz: 5, 2, 4, 6, 1, 3.
- Jede Zeile zeigt das Einordnen des aktuellen Elements in die sortierte Teilfolge.

5	2	4	6	1	3
2	5	4	6	1	3
2	4	5	6	1	3
2	4	5	6	1	3
1	2	4	5	6	3
1	2	3	4	5	6

Analyse

- **Laufzeit:**

- Im Best-Case ist das Array schon sortiert und die Laufzeit liegt in $\Theta(n)$ (die `while`-Schleife wird nie durchlaufen).
- Im Worst-Case muss die innere Schleife i -mal ausgeführt werden, d.h. die Laufzeit ist die Summe von

$$1 + 2 + 3 + \dots + (n-2) + (n-1) = \frac{(n-1)n}{2} = \Theta(n^2)$$

und damit liegt die Laufzeit wieder in $\Theta(n^2)$.

- Es kann gezeigt werden, dass im Average-Case die Laufzeit auch in $\Theta(n^2)$ liegt. Der Beweis ist kompliziert. Er beruht auf der Idee, dass die innere Schleife im Mittel $\frac{i}{2}$ -Mal ausgeführt wird.
- **Anzahl der Vertauschungen:** Wie oben, da Insertionsort in jedem Schritt der inneren Schleife Vertauschungen vornehmen muss, ist die Anzahl der Vertauschungen im Worst- und Average-Case ebenfalls $\Theta(n^2)$. Im Best-Case (sortiertes Array) sind es 0 Vertauschungen, also $\Theta(1)$.

Sortieren: Ausblick

- **Elementare Sortierverfahren:** Die Laufzeit liegt im Worst- und Average-Case immer in $\Theta(n^2)$.
- **Frage:** Kann man im Worst- und Average-Case schneller sortieren?
- **Antwort:** Ja. Die Erklärung folgt im Kapitel über Divide-and-Conquer-Algorithmen.

Polynomialzeit

- **Brute-Force-Methode:** Für viele nicht triviale Probleme gibt es einen einfachen Algorithmus, der jeden möglichen Fall überprüft.
 - In der Praxis häufig zu zeitaufwendig.
 - $n!$ Möglichkeiten für Stable-Matching mit n Kindern und n Familien.
- **Polynomialzeit:**
 - Es existiert eine Konstante $d \geq 1$, sodass die Laufzeit in $O(n^d)$ liegt.
- **Erklärung:** Ein Algorithmus läuft in Polynomialzeit, wenn die Laufzeit höchstens polynomiell mit der Größe der Eingabe n des Problems wächst.

Warum das wichtig ist

- **Tabelle:** Laufzeiten (aufgerundet) von Algorithmen mit unterschiedlichem Laufzeitverhalten für steigende Eingabegrößen auf einem Prozessor, der eine Million primitive Operationen pro Sekunde ausführen kann. Wenn die Laufzeit 10^{25} Jahre überschreitet, dann wird das als *sehr lange* angeführt.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n =$	10	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	4 s
$n =$	30	< 1 s	< 1 s	< 1 s	< 1 s	18 min	10^{25} Jahre
$n =$	50	< 1 s	< 1 s	< 1 s	11 min	36 Jahre	sehr lange
$n =$	100	< 1 s	< 1 s	< 1 s	1 s	12,892 Jahre	sehr lange
$n =$	1,000	< 1 s	< 1 s	1 s	18 min	sehr lange	sehr lange
$n =$	10,000	< 1 s	< 1 s	2 min	12 Tage	sehr lange	sehr lange
$n =$	100,000	< 1 s	2 s	3 Stunden	32 Jahre	sehr lange	sehr lange
$n =$	1,000,000	1 s	20 s	12 Tage	31,710 Jahre	sehr lange	sehr lange

Die Tabelle verdeutlicht den enormen Unterschied zwischen polynomiellen und exponentiellen Laufzeiten. Während Algorithmen mit polynomieller Laufzeit für relativ große Eingaben oft noch praktikabel sind, werden Algorithmen mit exponentieller Laufzeit schnell unbrauchbar, da die Laufzeit mit der Eingabegröße extrem stark ansteigt.

Worst-Case Polynomialzeit

- **Definition:** Wir nennen einen Algorithmus **effizient**, wenn seine Laufzeit polynomiell in der Eingabegröße ist.
- **Cobham-Edmonds Annahme:** Effiziente Lösbarkeit mit Lösbarkeit in Polynomialzeit gleichzusetzen, geht auf Alan Cobham und Jack Edmonds zurück, die das in den 1960er-Jahren vorgeschlagen haben.
- **Rechtfertigung:**
 - In der Praxis haben polynomielle Algorithmen meist kleine Konstanten und kleine Exponenten (man kann natürlich pathologische Fälle konstruieren ...).
 - Das Überwinden der exponentiellen Schranke von Brute-Force-Algorithmen legt meist eine wichtige Struktur des Problems offen.

- Diese Cobham-Edmonds-Annahme hat sich weitgehend durchgesetzt und die Informatikforschung der letzten 50 Jahre geprägt.
- **So effizient wie möglich**
 - Wenn wir ein Problem in Polynomialzeit lösen können, wollen wir natürlich einen Algorithmus mit möglichst kleiner polynomieller Laufzeit finden.
 - Es bedarf oft viel Aufwand, z.B. eine Laufzeit von $O(n^3)$ auf $O(n^2)$ zu reduzieren, oder von $O(n^2)$ auf $O(n \log n)$.
 - In den nächsten Abschnitten werden wir verschiedene algorithmische Probleme betrachten und möglichst effiziente Algorithmen zu ihrer Lösung kennenlernen.

Mehr dazu: [9. Polynominalzeitreduktionen](#)

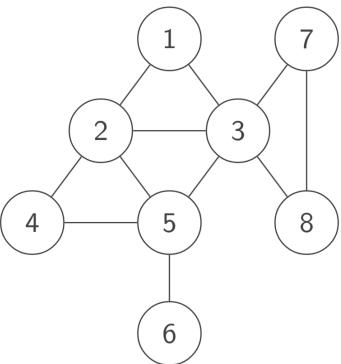
3. Graphen

Arten von Graphen + Definition

Ungerichtete Graphen

- **Ungerichteter Graph:** $G = (V, E)$, wobei
 - V = Menge der Knoten (Vertices, nodes).
 - E = Menge der Kanten zwischen Paaren von Knoten (edges).
 - Notation für Kante zwischen Knoten u und v : $\{u, v\}$ bzw. $\{v, u\}$.
 - Alternativ wird auch $u - v$ bzw. $v - u$ verwendet.
 - Parameter für Größen: $n = |V|$, $m = |E|$.

- **Beispiel:**



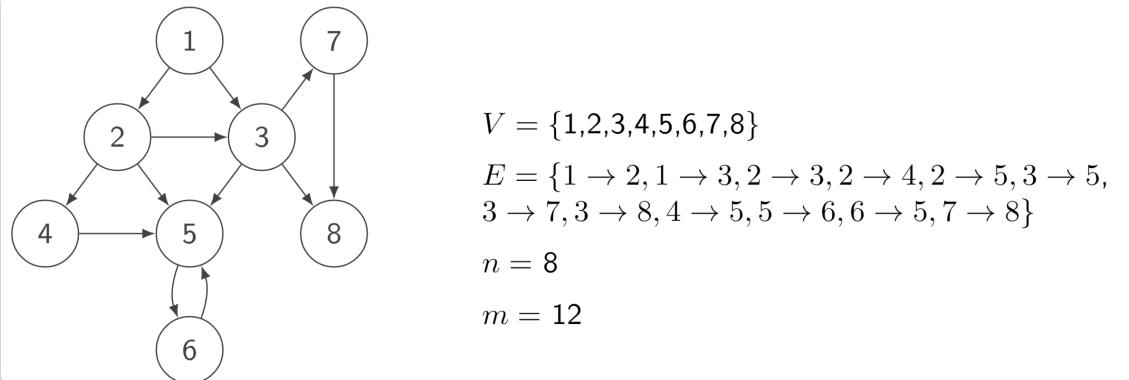
$$\begin{aligned} V &= \{1, 2, 3, 4, 5, 6, 7, 8\} \\ E &= \{1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6, 7-8\} \\ n &= 8 \\ m &= 11 \end{aligned}$$

- **Adjazent, inzident, Nachbarschaft:** Sei $e = \{u, v\}$ eine Kante in E .
 - u und v sind **adjazent**, d.h. u ist Nachbar von v und v ist Nachbar von u .
 - e ist **inzident** zu u (bzw. v), und u (bzw. v) ist inzident zu e .
 - (u, v) bezeichnet die ungeordnete Kante $\{u, v\}$.
- **Knotengrad (degree):** $\deg(v)$ bezeichnet den Knotengrad des Knotens v .
 - $\deg(v)$ entspricht der Anzahl der zu v inzidenten Kanten.
 - Es gilt: $\sum_{v \in V} \deg(v) = 2 \cdot |E|$ (Handshaking-Lemma).
- **Grundlegende Definitionen:**
 - **Mehrfachkante:** Mehrere Kanten zwischen zwei Knoten.
 - **Schleife:** Eine Kante, die einen Knoten mit sich selbst verbindet (z.B. $\{v, v\}$).
- **Schlichter Graph:** Ein ungerichteter Graph ohne Mehrfachkanten und ohne Schleifen.
- **Hinweise:**
 - In dieser Vorlesung werden, wenn nicht anders verlautbart, schlichte Graphen betrachtet.
 - Bei bestimmten Problemstellungen werden gewichtete Graphen verwendet, bei denen Knoten und/oder Kanten eine reelle Zahl zugeordnet bekommen.

Gerichtete Graphen

- **Gerichteter Graph (Digraph):** $G = (V, E)$, wobei
 - V = Menge der Knoten (vertices, nodes).
 - E = Menge der gerichteten Kanten (arcs) zwischen Paaren von Knoten.
 - Notation für Kante von Knoten a zu b : (a, b) bzw. $a \rightarrow b$.
 - $(a, b) \neq (b, a)$.

- **Beispiel:**



- **Hinweis:** Kanten in entgegengesetzter Richtung sind auch in schlichten Digraphen erlaubt.
- **Eingangsknotengrad:** $\deg^-(v)$ ist die Anzahl der eingehenden inzidenten Kanten (Kanten, die in v enden).
- **Ausgangsknotengrad:** $\deg^+(v)$ ist die Anzahl der ausgehenden inzidenten Kanten (Kanten, die von v ausgehen).
- Es gilt: $\deg(v) = \deg^+(v) + \deg^-(v)$ (wobei $\deg(v)$ der Grad im zugrundeliegenden ungerichteten Graphen wäre, falls die Richtung ignoriert wird).

Einige Anwendungen von Graphen

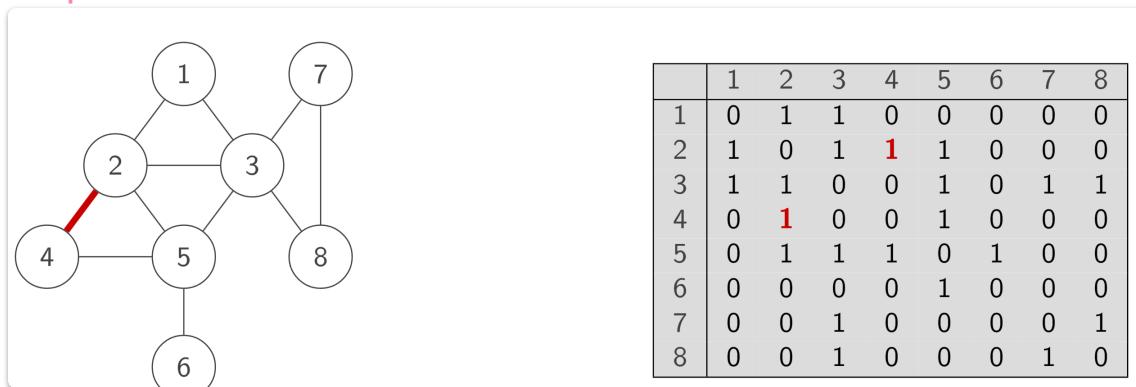
- **Tabelle: Beispiele für Graphen und ihre Komponenten**

<i>Graph</i>	<i>Knoten</i>	<i>Kanten</i>
Verkehr	Kreuzungen	Straßen
Netzwerke	Computer	Glasfaserkabel
World Wide Web	Webseiten	Hyperlinks
Sozialer Bereich	Personen	Beziehungen
Nahrungsnetz	Spezies	Räuber-Beute-Beziehung
Software	Funktionen	Funktionsaufrufe
Scheduling	Aufgaben	Ablaufeinschränkungen
elektronische Schaltungen	Gatter	Leitungen

Repräsentation von Graphen:

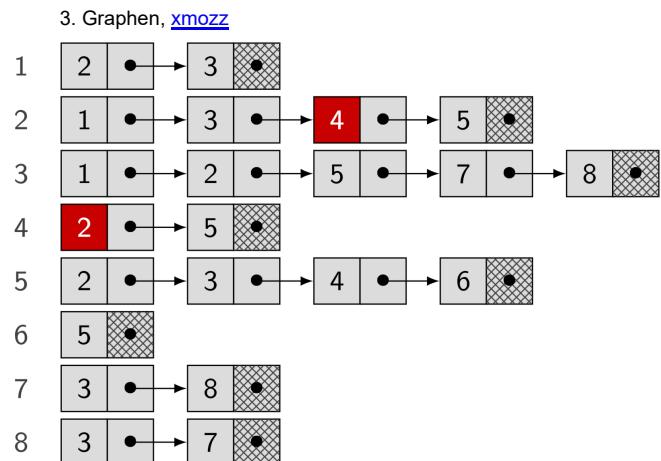
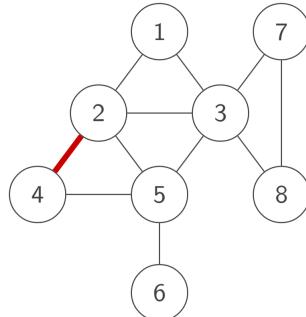
Adjazenzmatrix

- **Adjazenzmatrix:** $n \times n$ -Matrix A , mit $A_{uv} = 1$ wenn $\{u, v\}$ eine Kante ist (bzw. (u, v) für gerichtete Graphen).
 - Knoten: $1, 2, \dots, n$.
 - Zwei Einträge für jede ungerichtete Kante.
 - Für gewichtete Graphen: Reelle Matrix statt Boolesche Matrix (z.B. mit dem Gewicht der Kante).
 - Platzbedarf in $\Theta(n^2)$.
 - Überprüfen, ob $\{u, v\}$ eine Kante ist, hat Laufzeit $\Theta(1)$.
 - Aufzählen aller Kanten hat eine Laufzeit von $\Theta(n^2)$.
- **Beispiel:**



Adjazenzlisten

- **Adjazenzlisten:** Array von Listen. Index ist die Knotennummer.
 - Knoten: $1, 2, \dots, n$.
 - Zwei Einträge für jede ungerichtete Kante (in der Liste von u ist v und in der Liste von v ist u).
 - Für gewichtete Graphen: Speichere Gewicht in der Liste.
 - Platzbedarf in $O(n + m)$.
 - Überprüfen, ob $\{u, v\}$ eine Kante ist, hat eine Laufzeit von $O(\deg(u))$ (im schlimmsten Fall $O(n)$).
 - Aufzählen aller Kanten hat eine Laufzeit von $O(n + m)$.
- **Beispiel:**



Adjazenzmatrix oder Adjazenzlisten

- **Kantenanzahl:**

- Ein Graph kann bis zu $m = \binom{n}{2} = \frac{n(n-1)}{2} = \Theta(n^2)$ viele Kanten enthalten.
- Graphen sind **dicht (dense)** falls $m = \Theta(n^2)$.
- Graphen sind **licht (sparse)** falls $m = O(n)$.
- Für dichte Graphen sind beide Darstellungsformen (Adjazenzmatrix oder Adjazenzlisten) vergleichbar.

- **Praxis:**

- Graphen, die sich aus Anwendungen ergeben, enthalten aber oft erheblich weniger Kanten.
- Typischerweise gilt dann $m = O(n)$.
- In diesem Fall ist die Darstellung mittels Adjazenzlisten günstiger.

- **Hinweis:** Wenn wir sagen, dass ein Algorithmus auf Graphen in Linearzeit läuft ($O(n + m)$), gehen wir von einer Darstellung mit Adjazenzlisten aus und betrachten jeden Knoten und jede Kante einmal.

Kanten ausgeben

Ungerichteter Graph

Adjazenzmatrix: Adjazenzmatrix M gegeben, n Knoten nummeriert von 0 bis $n - 1$

```

for  $u \leftarrow 0$  bis  $n - 2$ 
    for  $v \leftarrow u + 1$  bis  $n - 1$ 
        if  $M[u, v] = 1$ 
            Gib Kante  $(u, v)$  aus

```

Adjazenzliste: n Knoten nummeriert von 0 bis $n - 1$, jeder Knoten besitzt Liste der adjazenten Knoten

```

for  $u \leftarrow 0$  bis  $n - 1$ 
    foreach Kante  $(u, v)$  inzident zu  $u$ 
        if  $u < v$ 
            Gib Kante  $(u, v)$  aus

```

Gerichtete Graphen

Adjazenzmatrix: Adjazenzmatrix M gegeben, n Knoten nummeriert von 0 bis $n - 1$

```

for  $u \leftarrow 0$  bis  $n - 1$ 
    for  $v \leftarrow 0$  bis  $n - 1$ 
        if  $M[u, v] = 1$ 
            Gib Kante  $(u, v)$  aus

```

Adjazenzliste: n Knoten nummeriert von 0 bis $n - 1$, jeder Knoten besitzt Liste der adjazenten Knoten

```

for  $u \leftarrow 0$  bis  $n - 1$ 
    foreach Kante  $(u, v)$  inzident zu  $u$ 
        Gib Kante  $(u, v)$  aus

```

Pfade und Kreise

Kantenzüge und Pfade

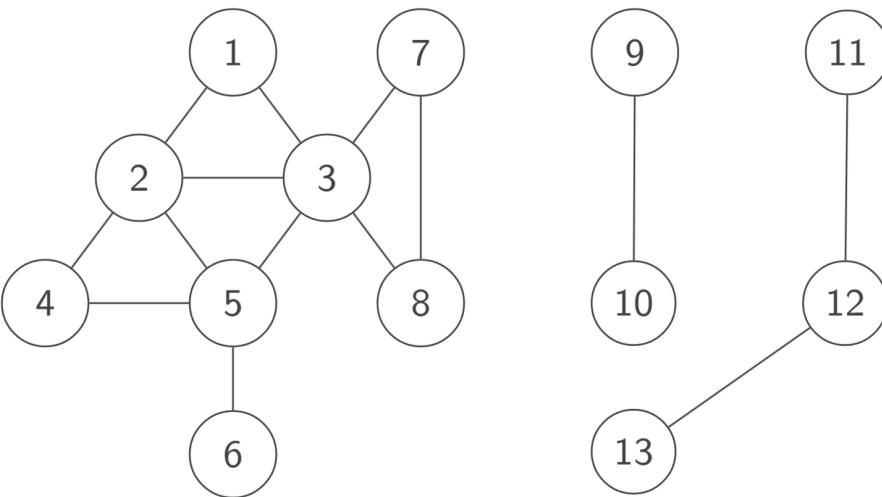
- **Definition:** **Kantenzug** (eng: non-simple path) in einem ungerichteten Graphen $G = (V, E)$ ist eine Folge von Knoten v_1, v_2, \dots, v_k , $k \geq 1$, mit der Eigenschaft, dass jedes aufeinanderfolgende Paar v_i, v_{i+1} durch eine Kante in E verbunden ist. Die **Länge** des Kantenzugs ist $k - 1$.
- **Definition:** **Pfad oder Weg** (eng: simple path) in einem ungerichteten Graphen $G = (V, E)$ ist ein Kantenzug v_1, v_2, \dots, v_k , bei dem sich kein Knoten wiederholt, also bei dem $v_i \neq v_j$ für alle $1 \leq i, j \leq k$ mit $i \neq j$ gilt.
- **Hinweis:** Wir sagen auch: Der Pfad geht von v_1 nach v_k und wir bezeichnen den Pfad als v_1-v_k -Pfad.
- **Achtung:** Die Begriffe Pfad, Weg und Kantenzug werden in der Literatur nicht einheitlich verwendet.

Zusammenhang und Distanz

- **Definition:** Knoten u ist von Knoten v in einem Graph G **erreichbar**, falls G einen $v-u$ -Pfad enthält.
- **Definition:** Ein ungerichteter Graph ist **zusammenhängend**, wenn jedes Paar von Knoten u und v voneinander erreichbar ist.
- **Definition:** Die **Distanz** zwischen Knoten u und v in einem ungerichteten Graphen ist die Länge eines kürzesten $u-v$ -Pfades.
- **Hinweis:** Falls u von v nicht erreichbar ist, nehmen wir die Distanz als ∞ an.

Zusammenhang: Beispiel

- **Nicht zusammenhängender Graph:**

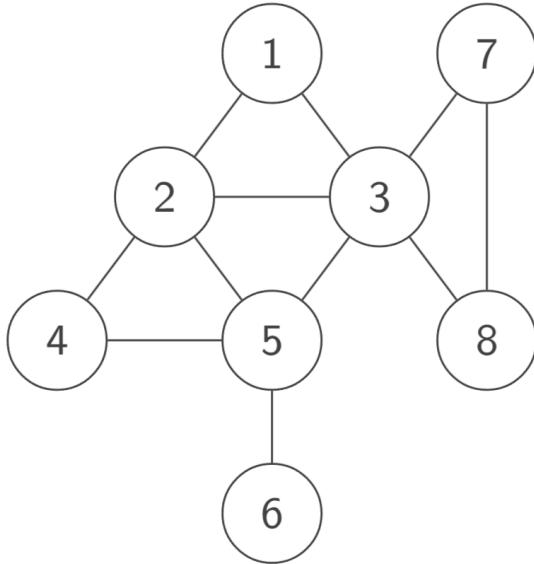


- **Nicht zusammenhängend:** Es gibt zum Beispiel keinen Pfad vom Knoten 1 zu Knoten 10.

- **Beispiel für Zusammenhang:** Die Knoten 1 bis 8 und ihre inzidenten Kanten bilden einen zusammenhängenden Graphen.

Kreis

- **Definition:** Ein Kreis (eng: simple cycle) ist ein Kantenzug v_1, v_2, \dots, v_k , in dem $v_1 = v_k$, $k \geq 3$, und die ersten $k - 1$ Knoten alle unterschiedlich sind. Die Länge des Kreises ist $k - 1$.
- **Beispiel für Kreis:** $C = 1, 2, 4, 5, 3, 1$



Pfade und Kreise in gerichteten Graphen

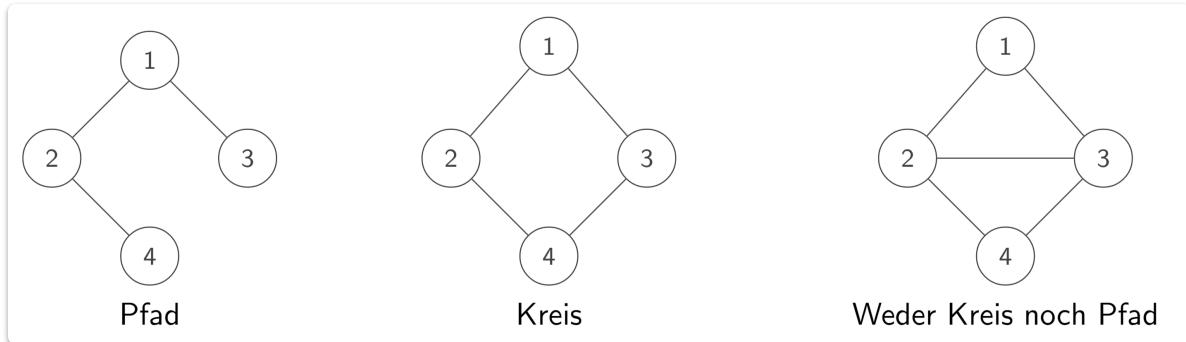
- **Pfad:** Ein Kantenzug in einem gerichteten Graphen $G = (V, E)$ ist eine Folge von Knoten v_1, v_2, \dots, v_k , $k \geq 1$, mit der Eigenschaft, dass jedes aufeinanderfolgende Paar (v_i, v_{i+1}) durch eine gerichtete Kante in E verbunden ist. Ein **Pfad** ist ein Kantenzug, in dem sich kein Knoten wiederholt.
- **Hierbei gilt:**
 - Der Pfad geht von einem Startknoten u zu einem Endknoten v (u - v -Pfad).
 - Die Umkehrung muss aber nicht gelten.
 - v kann von u aus erreicht werden, falls ein u - v -Pfad existiert.
 - Kürzeste u - v -Kantenzüge sind Pfade.
- **Kreis:** Ein gerichteter Kreis ist ein Kantenzug v_1, v_2, \dots, v_k , in dem $v_1 = v_k$, $k \geq 3$, und die ersten $k - 1$ Knoten alle unterschiedlich sind.

Pfade und Kreise als Graphen

- Falls ein Graph G aus nur einem Pfad oder nur einem Kreis besteht, so nennen wir den ganzen Graphen einen Pfad/Kreis. Formal sagen wir:
- **Pfad:** Ein Graph G ist ein **Pfad**, falls es eine Aufzählung v_1, v_2, \dots, v_k der Knoten von G gibt, so dass es in G genau eine Kante zwischen zwei Knoten v_i und v_j gibt, falls $|i - j| = 1$.

- **Kreis:** Ein Graph G ist ein **Kreis**, falls es eine Aufzählung v_1, v_2, \dots, v_k der Knoten von G gibt, so dass es in G genau eine Kante zwischen zwei Knoten v_i und v_j gibt, falls entweder $|i - j| = 1$ oder $\{i, j\} = \{1, k\}$ gilt.

Beispiele

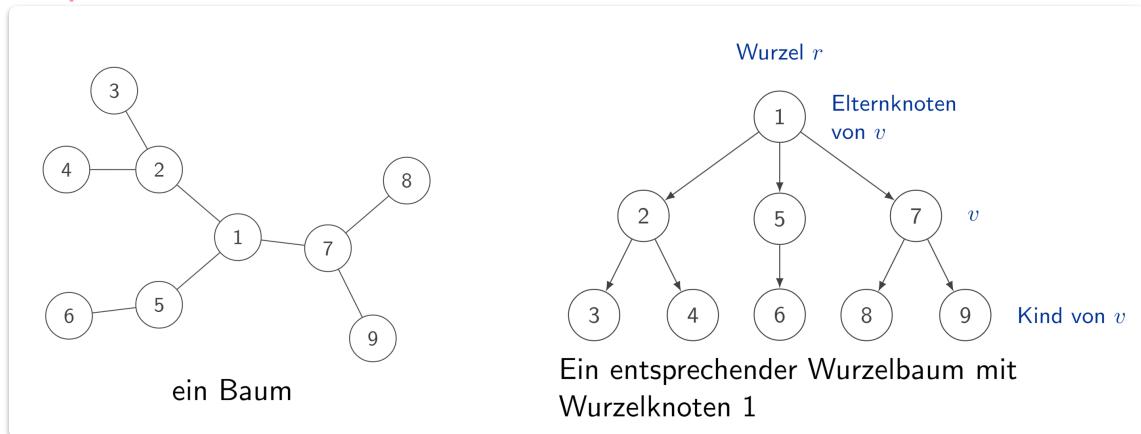


Bäume

- **Theorem:** Sei G ein ungerichteter Graph. G ist ein Baum genau dann, wenn es für jedes Paar von Knoten u und v genau einen Pfad von u nach v gibt.
- **Beweis:**
 - (\Rightarrow) G ist ein Baum, also zusammenhängend. Gäbe es zwei verschiedene Pfade zwischen u und v , so würde deren Vereinigung einen Kreis enthalten, was der Definition eines Baumes widerspricht.
 - (\Leftarrow) Wenn es für jedes Paar von Knoten genau einen Pfad gibt, dann ist G zusammenhängend. Gäbe es einen Kreis in G , so gäbe es mindestens ein Paar von Knoten im Kreis, für die zwei verschiedene Pfade existieren (entlang des Kreises in beide Richtungen), was der Voraussetzung widerspricht. Also enthält G keinen Kreis und ist somit ein Baum.

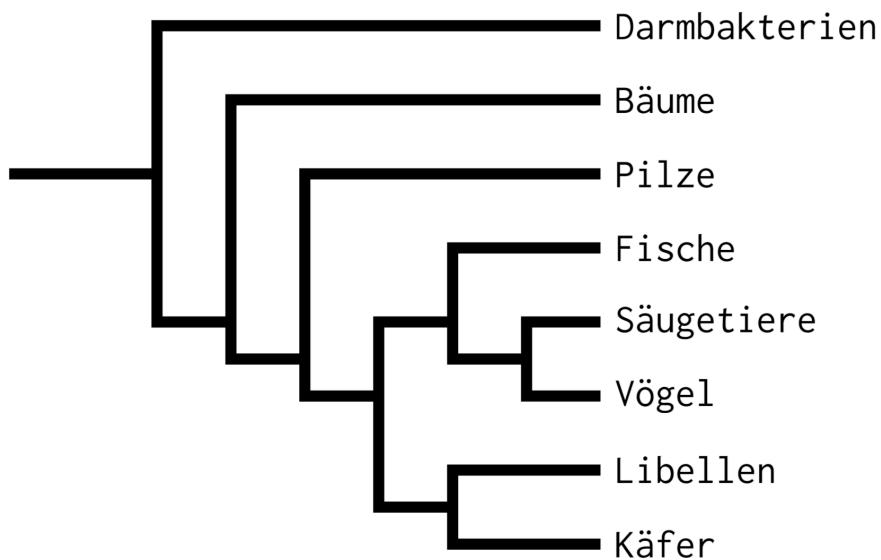
Wurzelbaum (rooted tree, arborescence)

- **Wurzelbaum:** Gegeben sei ein Baum T . Wähle einen Wurzelknoten r und gib jeder Kante eine Richtung von r weg.
- **Bedeutung:** Modelliert hierarchische Strukturen.
- **Beispiele:**



Phylogenetischer Baum

- **Phylogenetischer Baum:** Beschreibt die evolutionären Beziehungen zwischen verschiedenen Arten.
- **Beispiel:**

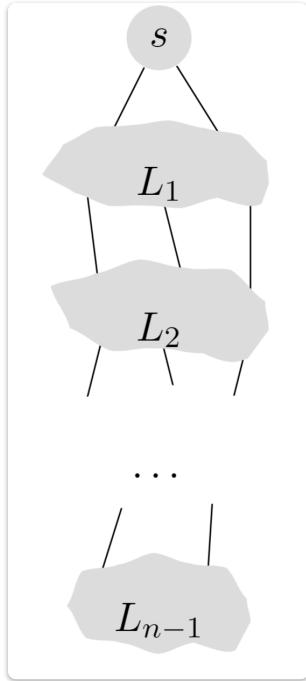


- So was haben wir auch bei **GUI-Hierarchien**

Durchmusterung von Graphen (Graph Traversal)

Breitensuche (Breadth First Search, BFS)

- **BFS Ansatz:** Untersuche alle Knoten der Reihe nach ausgehend vom Startknoten s in aufsteigender Richtung, wobei die Knoten Ebene für Ebene abgearbeitet werden.

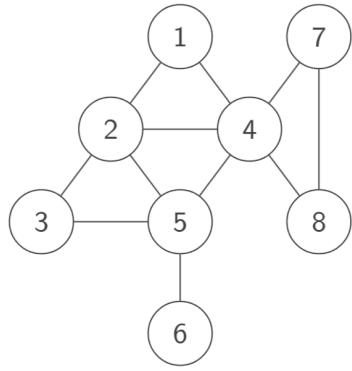


- **BFS Algorithmus:**

- $L_0 = \{s\}$ = alle Knoten mit Distanz 0 von s .
- L_1 = alle Knoten, die nicht zu L_0 gehören und die über eine Kante mit einem Knoten in L_0 verbunden sind.
- L_i = alle Knoten, die zu keiner der vorherigen Ebenen gehören und die über eine Kante mit einem Knoten in L_{i-1} verbunden sind.

Anwendungen der Breitensuche

- **Zusammenhangsproblem:** Existiert zwischen zwei gegebenen Knoten s und t ein Pfad?
- **Kürzester Pfad:** Wie viele Kanten hat ein kürzester Pfad zwischen s und t (= Distanz zwischen s und t)?
- **Anwendungen:**
 - Routenplanung (z.B. Labyrinth durchsuchen).
 - Six Degrees of Kevin Bacon-Zahl.
 - Die kleinste Anzahl an Hops (kürzester Pfad) zwischen zwei Knoten in einem Kommunikationsnetzwerk.



Breitensuche: Theorem

- **Theorem:** Für jede Ebene $i = 0, 1, \dots$ gilt, dass L_i alle Knoten mit Distanz i von s beinhaltet.
- **Beweis (angedeutet):**
 - **Basisfall:** $i = 0$. $L_0 = \{s\}$, und die Distanz von s zu sich selbst ist 0.
 - **Induktiver Schritt:** Angenommen, die Aussage gilt für $i - 1$. Betrachte einen Knoten $v \in L_i$. Da v in L_i liegt, muss er ein Nachbar eines Knotens $u \in L_{i-1}$ sein und nicht in einer vorherigen Ebene gelegen haben. Nach der Induktionsvoraussetzung hat u die Distanz $i - 1$ von s . Daher hat v höchstens die Distanz i von s . Wäre die Distanz kleiner als i , müsste v bereits in einer früheren Ebene entdeckt worden sein.
 - Für alle weiteren Knoten gilt die gleiche Argumentation, d.h. L_i liegt schließlich in der i -ten Ebene.

Implementierung mit einer Queue

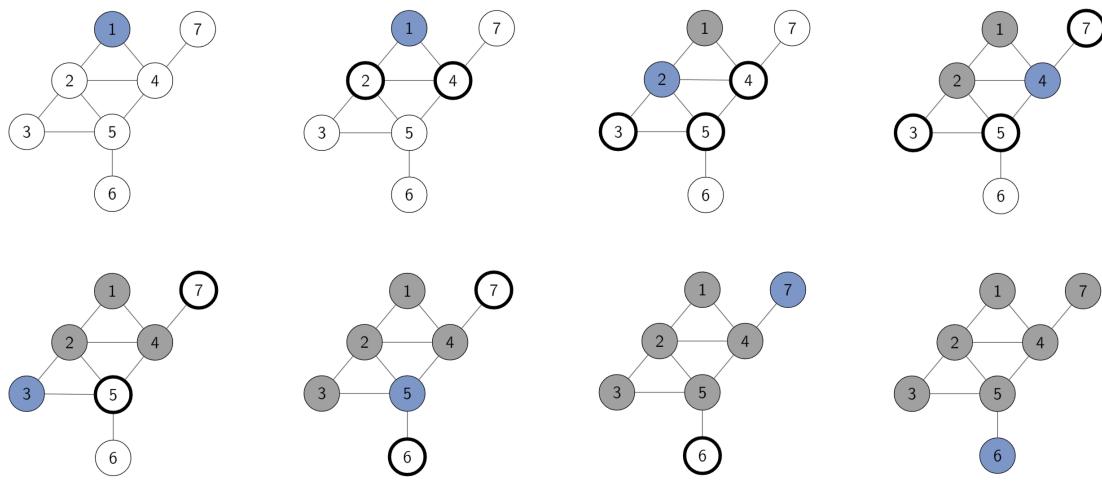
Implementierung: Array Discovered, Queue Q , Graph $G = (V, E)$, Startknoten s .

```

BFS( $G, s$ ):
    Discovered[ $s$ ]  $\leftarrow$  true
    Discovered[ $v$ ]  $\leftarrow$  false für alle anderen Knoten  $v \in V$ 
     $Q \leftarrow \{s\}$ 
    while  $Q$  ist nicht leer
        Entferne ersten Knoten  $u$  aus  $Q$ 
        Führe Operation auf  $u$  aus (z.B. Ausgabe)
        foreach Kante  $(u, v)$  inzident zu  $u$ 
            if !Discovered[ $v$ ]
                Discovered[ $v$ ]  $\leftarrow$  true
                Füge  $v$  zu  $Q$  hinzu
    
```

Beispiel

- **Möglicher Ablauf:** Startknoten = 1, bearbeitete Knoten sind grau, aktiver Knoten ist blau, alle anderen Knoten sind weiß, Knoten in Queue sind mit dicken Rahmen gekennzeichnet.



Analyse

- **Theorem:** BFS hat eine Laufzeit von $O(n + m)$.
- **Laufzeit:** Für die Laufzeitabschätzung müssen wir drei Teile betrachten:
 - Initialisierung vor der `while`-Schleife
 - `while`-Schleife
 - `foreach`-Schleife

Analyse

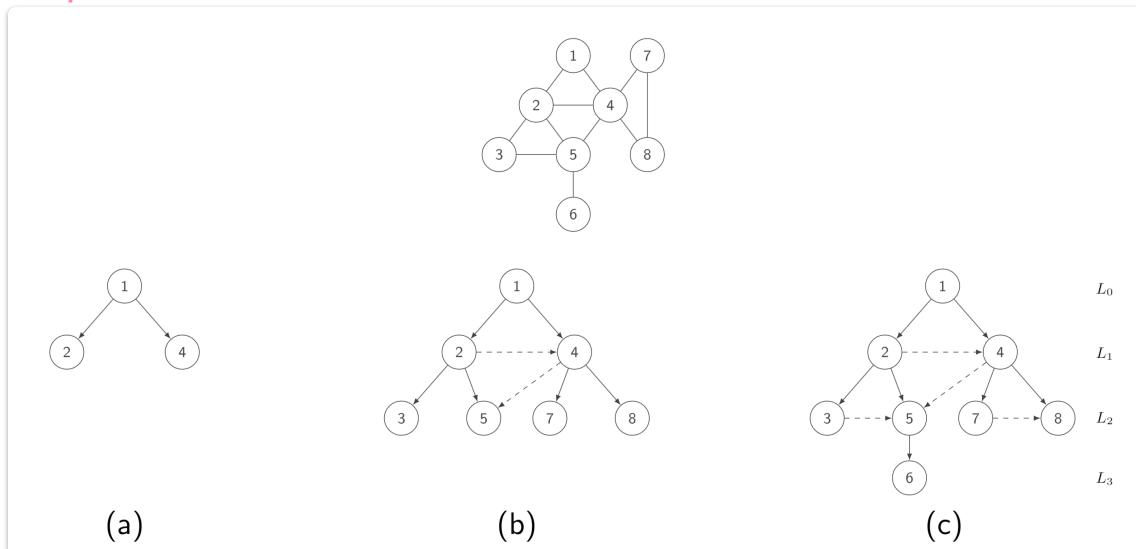
- **Initialisierung vor der `while`-Schleife:**
 - Jeder Knoten wird einmal betrachtet.
 - Pro Knoten können die Anweisungen in konstanter Zeit ausgeführt werden.
 - Daher benötigt die Initialisierung $O(n)$ Zeit.
- **`while`-Schleife:**
 - Jeder Knoten u wird höchstens einmal in Q gegeben, denn nachdem er das erste Mal in Q gegeben wird, wird ja `Discovered[u]` auf `true` gesetzt.
 - Daher wird die `while`-Schleife für jeden Knoten höchstens einmal durchlaufen.
- **`foreach`-Schleife:**
 - Sei u der gerade aktuelle Knoten bevor die `foreach`-Schleife ausgeführt wird.
 - Dann werden in der `foreach`-Schleife alle Knoten v in der Adjazenzliste von u betrachtet.
 - Das sind genau $\deg(u)$ viele. Daher wird die Schleife $\deg(u)$ -mal durchlaufen. Die gesamten Anweisungen in der Schleife benötigen konstante Zeit.
- **Gesamt:**
 - Insgesamt beträgt die Laufzeit also $O(n + \sum_{u \in V} \deg(u))$.
 - Da $\sum_{u \in V} \deg(u) = 2m$, liegt die Laufzeit in $O(n + m)$.

BFS-Baum

- **BFS-Baum:** Breitensuche erzeugt einen Baum (BFS-Baum), dessen Wurzel ein Startknoten s ist und der alle von s erreichbaren Knoten beinhaltet.
- **Aufbau:** Man startet bei s . Wird nun ein Knoten v in der Ebene L_i gefunden, ist er zu mindestens einem Knoten u der Ebene L_{i-1} benachbart. Der Knoten u von dem aus v gefunden wird, wird als Elternknoten von v im BFS-Baum gemacht.

Eigenschaft

- **Eigenschaft:** Sei T ein BFS-Baum von $G = (V, E)$ und sei $\{x, y\}$ eine Kante von G . Dann können die Ebenen der Knoten x und y höchstens um 1 unterschieden sein.
- **Beispiele:**



Ermitteln der Ebenen

- **Anwendung von BFS:** Ermitteln der Ebene jedes einzelnen Knotens.

Implementierung: Array Level, Queue Q , Graph $G = (V, E)$, Startknoten s .

```

BFS( $G, s$ ):
Level[ $s$ ]  $\leftarrow 0$ 
Level[ $v$ ]  $\leftarrow -1$  für alle anderen Knoten  $v \in V$ 
 $Q \leftarrow s$ 
while  $Q$  ist nicht leer
    Entferne ersten Knoten  $u$  aus  $Q$ 
    foreach Kante  $(u, v)$  inzident zu  $u$ 
        if Level[ $v$ ] == -1
            Level[ $v$ ]  $\leftarrow$  Level[ $u$ ] + 1
            Füge  $v$  zu  $Q$  hinzu

```

Tiefensuche (Depth First Search, DFS)

- **DFS Ansatz:** Von einem besuchten Knoten u wird zuerst immer zu einem weiteren noch nicht besuchten Nachbarknoten gegangen (rekursiver DFS-Aufruf), bevor die weiteren Nachbarknoten von u besucht werden.
- **DFS Algorithmus:**

DFS(G, s):

Discovered[v] \leftarrow false für alle Knoten $v \in V$

DFS1(G, s)

DFS1(G, u):

Discovered[u] \leftarrow true

Führe Operation auf u aus (z.B. Ausgabe)

foreach Kante (u, v) inzident zu u

if !Discovered[v]

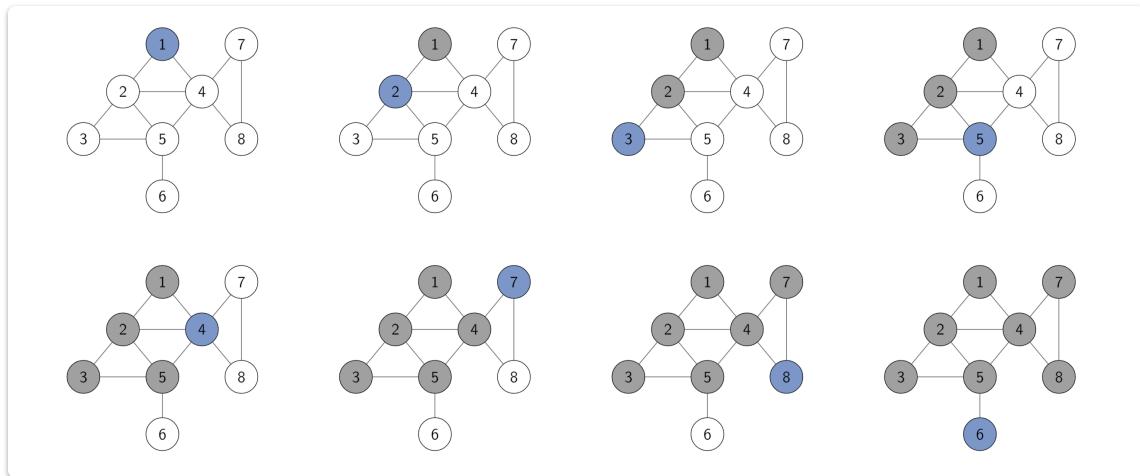
DFS1(G, v)

Analyse

- **Theorem:** DFS hat eine Laufzeit von $O(n + m)$.
- **Laufzeit:** Für die Laufzeitabschätzung betrachten wir:
 - Initialisierung
 - **foreach**-Schleife
 - Rekursive Aufrufe
- **Initialisierung:**
 - Initialisierung vor dem Aufruf von DFS1 in $O(n)$ Zeit.
 - DFS1(G, s) wird für jeden Knoten s höchstens einmal aufgerufen.
- **foreach -Schleife in DFS1(G, u):**
 - Es werden alle Knoten v in der Adjazenzliste von u betrachtet. Das sind genau $\deg(u)$ viele.
 - Daher wird die Schleife $\deg(u)$ -mal durchlaufen.
 - Die einzelnen Anweisungen in der Schleife benötigen konstante Zeit (außer dem rekursiven Aufruf DFS1(G, v), aber dessen Laufzeit wird ja in der Analyse für den Knoten v berücksichtigt).
- **Gesamt:**
 - Insgesamt beträgt die Laufzeit also $O(n + \sum_{u \in V} \deg(u))$.
 - Da $\sum_{u \in V} \deg(u) = 2m$, erhalten wir eine Laufzeit von $O(n + m)$.

Beispiel

- **Möglicher Ablauf:** Startknoten = 1, bearbeitete Knoten sind grau unterlegt, aktiver Knoten ist blau, alle anderen Knoten sind weiß.

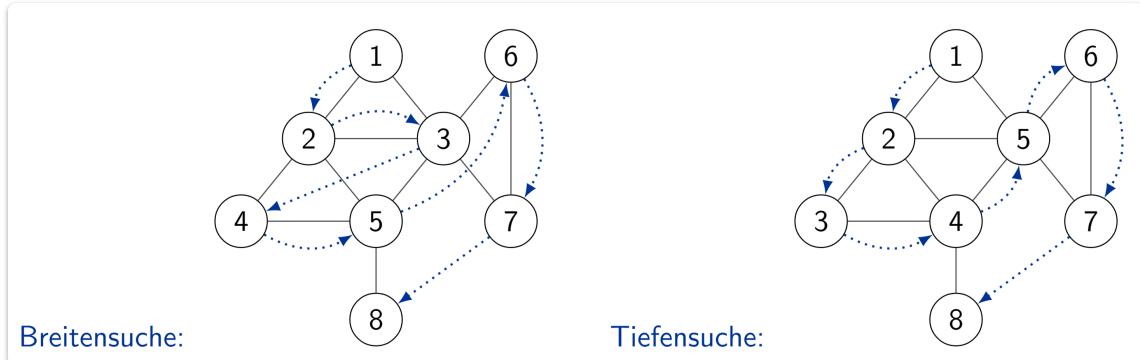


Durchmusterung

- **Durchmusterung:** Durchmusterung bei DFS unterscheidet sich von der bei BFS.
 - Es wird zunächst versucht, möglichst weit vom Startknoten weg zu kommen.
 - Gibt es in der Nachbarschaft keine möglichen Knoten, dann wird durch rekursive Aufstiege bis zu einer möglichen Verzweigung zurückgegangen (Backtracking).

Beispiel

- Vergleich: Tiefensuche und Breitensuche im Vergleich

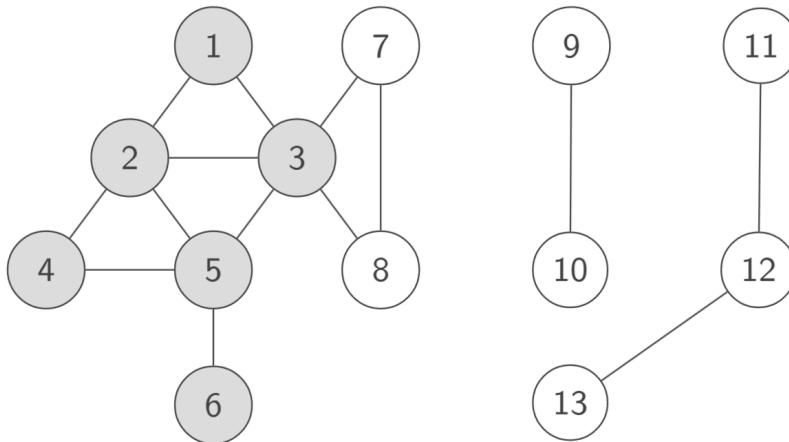


Zusammenhangskomponente

- **Zusammenhang (Wiederholung):** Ein ungerichteter Graph ist **zusammenhangend**, wenn fur jedes Paar von Knoten u und v ein Pfad zwischen u und v existiert.
 - **Nicht zusammenhangend:** Gibt es zwischen einem Paar von Knoten keinen Pfad, dann ist der Graph nicht zusammenhangend.
 - **Teilgraph:** Ein Graph $G_1 = (V_1, E_1)$ heit **Teilgraph** von $G_2 = (V_2, E_2)$, wenn seine Knotenmenge V_1 Teilmenge von V_2 und seine Kantenmenge E_1 Teilmenge von E_2 ist.
 - **Zusammenhangskomponente:** Ein **maximaler** zusammenhangender Teilgraph. Ein nicht zusammenhangender Graph besteht aus mehreren Zusammenhangskomponenten. Ein

zusammenhängender Graph besitzt nur eine Zusammenhangskomponente.

- **Beispiel:** Ein nicht zusammenhängender Graph mit 3 Zusammenhangskomponenten.



- Der durch die grauen Knoten induzierte Teilgraph ist zwar zusammenhängend, aber er ist nicht maximal (da Knoten 7 und 8 können noch hinzugefügt werden). Daher bildet dieser Teilgraph keine Zusammenhangskomponente.
- **Zusammenhangskomponente:** Finde alle Knoten, die von s aus erreicht werden können.
- **Lösung:**
 - Rufe $\text{DFS}(G, s)$ oder $\text{BFS}(G, s)$ auf.
 - Ein Knoten u ist von s genau dann erreichbar, wenn $\text{Discovered}[u] = \text{true}$ ist.

Zusammenhangskomponenten zählen

- **DFSUM Algorithmus:** Startknoten s , globales Array Discovered , Graph $G = (V, E)$.

$\text{DFSUM}(G)$:

$\text{Discovered}[v] \leftarrow \text{false}$ für alle Knoten $v \in V$

$i \leftarrow 0$

foreach Knoten $v \in V$

if $\text{Discovered}[v] = \text{false}$

$i \leftarrow i + 1$

$\text{DFS1}(G, v)$

return i

- **Laufzeit:** Die Laufzeit liegt in $O(n + m)$.

- **Analyse:**

- Sei $G = (V, E)$ der gegebene Graph und $G_1 = (V_1, E_1), \dots, G_r = (V_r, E_r)$ seine Zusammenhangskomponenten. Sei $|V_i| = n_i$ und $|E_i| = m_i$, für $1 \leq i \leq r$.
- Klarerweise gilt $n = n_1 + \dots + n_r$ und $m = m_1 + \dots + m_r$.
- Für jede einzelne Zusammenhangskomponente G_i ($1 \leq i \leq r$) führt der Algorithmus eine Tiefensuche durch. Dies hat eine Laufzeit von $O(n_i + m_i)$.
- Die Initialisierung benötigt $O(n)$ Zeit.
- Insgesamt erhalten wir eine Laufzeit von $O(n + \sum_{i=1}^r (n_i + m_i)) = O(n + (n + m)) = O(2n + m) = O(n + m)$.

Zusammenhang in gerichteten Graphen

Suche in gerichteten Graphen

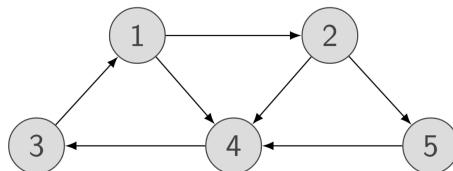
- **Gerichtete Erreichbarkeit:** Gegeben sei ein Knoten s , finde alle Knoten, die von s aus erreicht werden können.
- **Gerichteter kürzester Pfad:** Gegeben seien zwei Knoten s und t , ermittle einen kürzesten Pfad von s nach t .
- **Suche in gerichteten Graphen:** BFS und DFS können auch auf gerichteten Graphen angewendet werden.
- **Beispiel Webcrawler:** Starte von einer Webseite s . Finde alle Webseiten, die von s aus direkt oder indirekt verlinkt sind.

Starker Zusammenhang

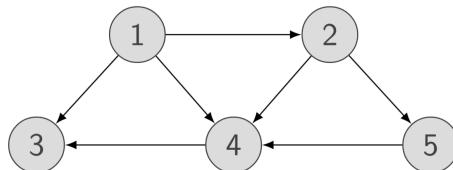
- **Definition:** Knoten u und v in einem gerichteten Graphen sind **gegenseitig erreichbar**, wenn es einen Pfad von u zu v und einen Pfad von v zu u gibt.
- **Definition:** Ein gerichteter Graph ist **stark zusammenhängend**, wenn jedes Paar von Knoten im Graphen gegenseitig erreichbar ist.
- **Hinweis:** Ein gerichteter Graph ist **schwach zusammenhängend**, falls der zugehörige ungerichtete Graph (also der Graph, der entsteht, wenn man jede gerichtete Kante durch eine ungerichtete Kante ersetzt) zusammenhängend ist.

Beispiel

Stark zusammenhängend:



Nicht stark zusammenhängend (aber schwach zusammenhängend): Knoten 1 kann von keinem anderen Knoten erreicht werden, vom Knoten 3 führt kein Pfad weg.



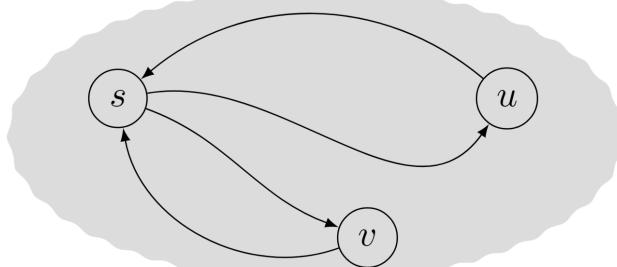
Starker Zusammenhang Fortsetzung

- **Lemma:** Sei s ein beliebiger Knoten in einem gerichteten Graphen G . G ist stark zusammenhängend dann und nur dann, wenn jeder Knoten von s aus und s von jedem Knoten aus erreicht werden kann.

- **Beweis:**

- (\implies) Folgt direkt aus der Definition.
- (\impliedby) Seien u und v beliebige Knoten. Nach Voraussetzung gibt es einen Pfad von s nach v ($s \rightsquigarrow v$) und einen Pfad von u nach s ($u \rightsquigarrow s$). Durch Verkettung erhalten wir einen Pfad von u nach v ($u \rightsquigarrow s \rightsquigarrow v$). Ebenso gibt es einen Pfad von s nach u ($s \rightsquigarrow u$) und einen Pfad von v nach s ($v \rightsquigarrow s$), also auch einen Pfad von v nach u ($v \rightsquigarrow s \rightsquigarrow u$). Somit sind u und v gegenseitig erreichbar, und da u und v beliebig gewählt waren, ist G stark zusammenhängend.

auch ok, wenn Pfade überlappen

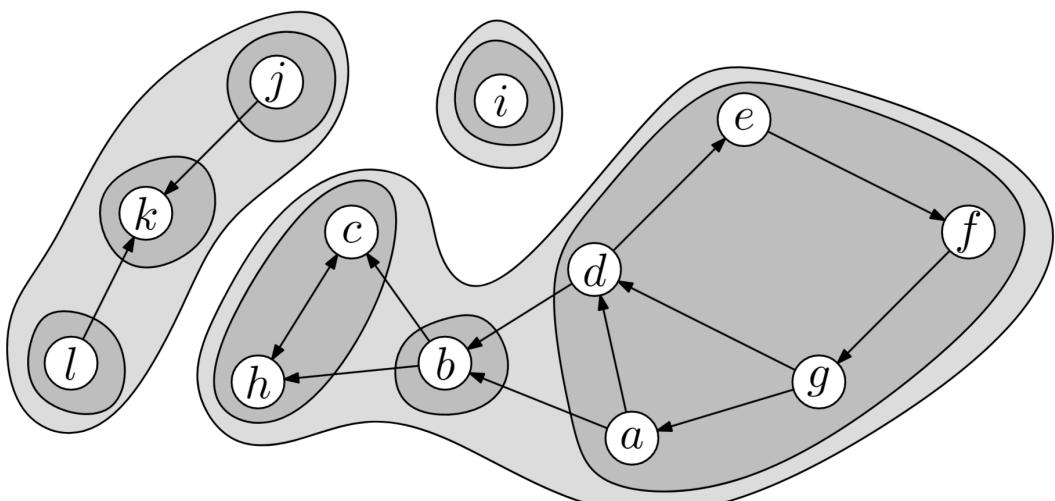


Algorithmus

- **Theorem:** Laufzeit für die Überprüfung, ob G stark zusammenhängend ist, liegt in $O(n + m)$.
- **Beweis:**
 - Wähle einen beliebigen Knoten s .
 - Führe BFS mit Startknoten s in G aus.
 - Gib true zurück dann und nur dann, wenn alle Knoten in beiden BFS-Ausführungen erreicht werden können.
 - Korrektheit folgt unmittelbar aus dem vorherigen Lemma.
 - umgekehrte Orientierung von jeder Kante in G .

Schwache und starke Zusammenhangskomponenten

- Eine **schwache Zusammenhangskomponente** eines gerichteten Graphen ist ein maximaler schwach zusammenhängender gerichteter Teilgraph.
- Eine **starke Zusammenhangskomponente** eines gerichteten Graphen ist ein maximaler stark zusammenhängender gerichteter Teilgraph.
- **Beispiel:** Schwache Zusammenhangskomponenten sind in hellgrau, starke Zusammenhangskomponenten in dunkelgrau gekennzeichnet.



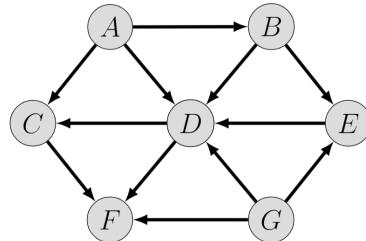
DAGs und Topologische Sortierung

Gerichteter azyklischer Graph (Directed Acyclic Graph, DAG)

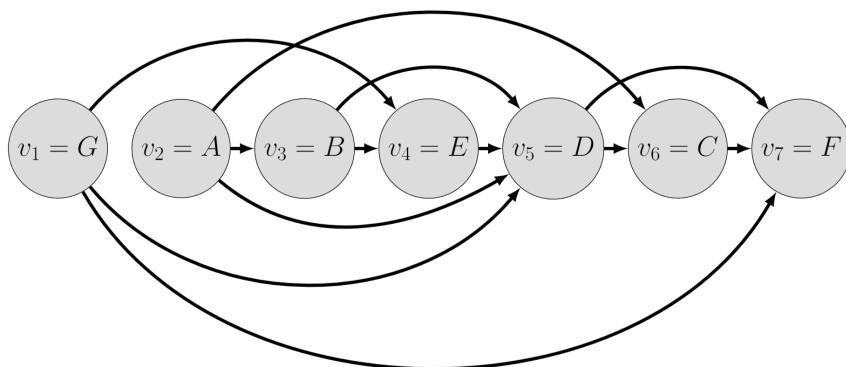
- **Definition:** Ein DAG ist ein gerichteter Graph, der keine gerichteten Kreise enthält.
- **Beispiel:** Knoten: Aufgaben, Kanten: Reihenfolgebeschränkungen. Kante (u, v) bedeutet, Aufgabe u muss vor Aufgabe v erledigt werden.
- **Definition:** Wir nennen einen Knoten v ohne eingehende Kanten in einem gerichteten Graphen (i.e., $\deg^-(v) = 0$) **Quelle**.
- **Definition:** Eine **topologische Sortierung** eines gerichteten Graphen $G = (V, E)$ ist eine lineare Ordnung seiner Knoten, bezeichnet mit v_1, v_2, \dots, v_n , sodass für jede Kante $(v_i, v_j) \in E$ gilt, dass $i < j$.

Topologische Sortierung: Beispiel

Ein DAG:



Eine topologische Sortierung:

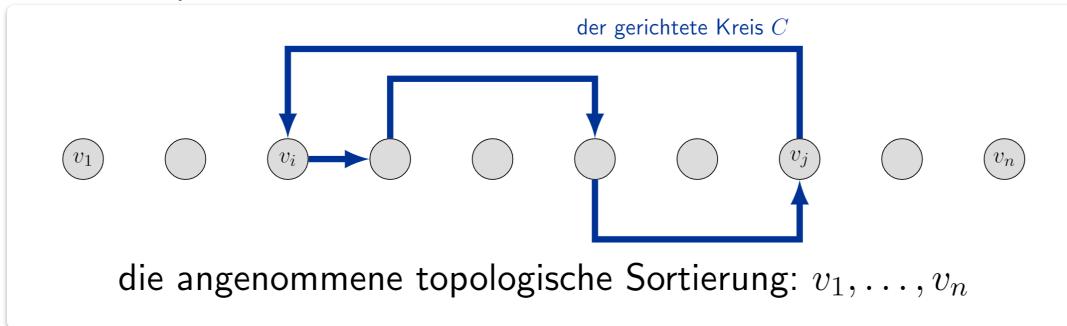


Reihenfolgebeschränkung

- **Reihenfolgebeschränkung:** Kante (u, v) bedeutet, dass Aufgabe u vor Aufgabe v bearbeitet werden muss.
- **Anwendungen:**
 - Voraussetzungen bei Kursen: Kurs u muss vor Kurs v absolviert werden.
 - Übersetzung: Modul u muss vor Modul v übersetzt werden.
 - Pipeline von Prozessen: Ausgabe von Prozess u wird benötigt, um die Eingabe von v zu bestimmen.

Gerichteter azyklischer Graph

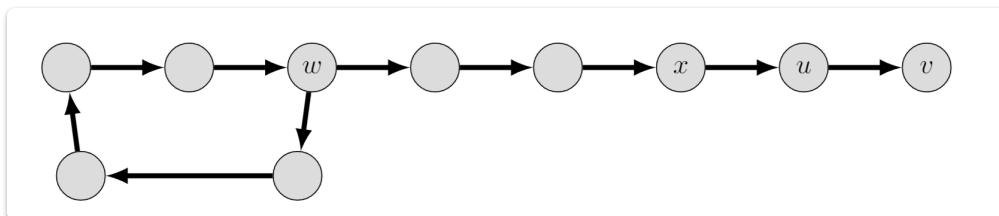
- **Lemma:** Wenn G eine topologische Sortierung hat, dann ist G ein DAG. (Beweis durch Widerspruch)
 - Wir nehmen an, dass G eine topologische Sortierung v_1, \dots, v_n besitzt und auch einen gerichteten Kreis C besitzt.
 - Sei v_i der Knoten mit dem kleinsten Index in C und sei v_j der Knoten direkt vor v_i in C ; daher gibt es die Kante (v_j, v_i) .
 - Durch die Wahl von v_i gilt $j > i$.
 - Andererseits, da v_1, \dots, v_n eine topologische Sortierung ist, müsste eigentlich $j < i$ sein. Widerspruch.



- **Lemma:** Wenn G eine topologische Sortierung hat, dann ist G ein DAG.
- **Frage:** Hat jeder DAG eine topologische Sortierung?
- **Frage:** Wenn ja, wie berechnen wir diese?

Lemma 1:

- Wenn G ein DAG ist, dann hat G eine Quelle. (Beweis durch Widerspruch)
 - Wir nehmen an, G ist ein DAG ohne Quelle.
 - Wähle einen beliebigen Knoten v und folge den Kanten von v aus rückwärts. Da v zumindest eine eingehende Kante (u, v) besitzt, können wir rückwärts zu u gelangen.
 - Da u zumindest eine eingehende Kante (x, u) hat, können wir rückwärts zu x gelangen.
 - Da G endlich ist, wiederholt sich irgendwann ein Knoten.
 - Sei C die Sequenz von Knoten zwischen dem ersten und zweiten Besuch des sich wiederholenden Knotens. C ist ein Kreis. \square



Lemma 2:

- G ist ein DAG genau dann, wenn jeder Teilgraph von G eine Quelle hat.
 - **Beweis:**

- (\implies) Wenn G ein DAG ist, dann ist offensichtlich auch jeder Teilgraph von G ein DAG (Das Entfernen von Knoten kann keine Kreise produzieren). Deswegen hat jeder Teilgraph von G eine Quelle (nach dem vorherigen Lemma).
- (\impliedby) Angenommen G ist kein DAG. Dann enthält G einen Kreis als Teilgraph. Ein Kreis hat keine Quelle. Widerspruch.

Erkennen eines DAG mittels wiederholtem Löschen von Kanten

```

while  $G$  hat mindestens einen Knoten
  if  $G$  hat eine Quelle
    Wähle eine Quelle  $v$  aus
    Gib  $v$  aus
    Lösche  $v$  und alle inzidenten Kanten aus  $G$ 
  else return  $G$  ist kein DAG
return  $G$  ist ein DAG

```

- **Hinweis:**
 - Ein Knoten kann im Lauf des Algorithmus zur Quelle werden.
 - Falls G ein DAG ist, gibt dieser Algorithmus eine topologische Sortierung aus.

Lemma 3:

- Wenn G ein DAG ist, dann hat G eine topologische Sortierung.
- **Beweis:**
 - Falls G ein DAG ist, dann hat G eine Quelle. Wir wählen eine Quelle v_1 als erstes Element der topologischen Sortierung und entfernen v_1 und alle ausgehenden Kanten. Der resultierende Graph G' ist ebenfalls ein DAG und hat somit wieder eine Quelle v_2 . Wir wählen v_2 als zweites Element der topologischen Sortierung und wiederholen diesen Prozess, bis alle Knoten entfernt wurden. Die resultierende Reihenfolge v_1, v_2, \dots, v_n ist eine topologische Sortierung, da für jede Kante (v_i, v_j) gilt, dass v_i vor v_j in der Ordnung gewählt wurde, also $i < j$. Falls G keinen DAG enthält (also einen Kreis), ist in jedem Schritt keine Quelle vorhanden, und es kann keine topologische Sortierung erstellt werden.

Topologische Sortierung

- **Algorithmus:** Effiziente Implementierung des Lösralgorithmus: Löschen von Knoten, deren In-Degree 0 ist. Es wird zusätzlich eine Liste L verwendet.

```

foreach  $v \in V$ 
    count[ $v$ ]  $\leftarrow 0$ 
foreach  $v \in V$ 
    foreach Kante  $(v, w) \in E$ 
        count[ $w$ ]  $\leftarrow$  count[ $w$ ] + 1
foreach  $v \in V$ 
    if count[ $v$ ] = 0
        Gib  $v$  zur Liste  $L$  am Anfang hinzufügen
while  $L$  ist nicht leer
    Sei  $v$  erstes Element in  $L$ , lösche  $v$  aus  $L$ 
    Gib  $v$  aus
    foreach Kante  $(v, w) \in E$ 
        count[ $w$ ]  $\leftarrow$  count[ $w$ ] - 1
        if count[ $w$ ] = 0
            Gib  $w$  zur Liste  $L$  am Anfang hinzufügen

```

Laufzeit

- **Theorem:** Algorithmus findet eine topologische Sortierung in $O(n + m)$ Zeit.
- **Beweis:** Dazu betrachten wir die folgenden Teile:
 - Initialisierung
 - Erste `foreach`-Schleife für `count`.
 - Zweite `foreach`-Schleife für Liste der Quellen.
 - `while`-Schleife (mit `foreach`-Schleife).
- **Initialisierung:**
 - Die erste `foreach`-Schleife für `count` benötigt $O(n)$ Zeit.
 - Bei den verschachtelten `foreach`-Schleifen wird die innere `foreach`-Schleife für jeden Knoten genau $\deg^+(v)$ -mal durchlaufen. Daher benötigt man dafür $O(\sum_{v \in V} \deg^+(v)) = O(m)$ Zeit.
 - Die Generierung der Liste L durch die dritte `foreach`-Schleife benötigt $O(n)$ Zeit.
 - Die Initialisierung hat also eine Laufzeit von $O(n + m)$.

Analyse

- **while -Schleife:**
 - Jeder Knoten v wird höchstens einmal aus L entnommen.
 - Daher wird die `while`-Schleife höchstens n -mal durchlaufen.
- **foreach -Schleife:**
 - Sei u der gerade aktuelle Knoten bevor die `foreach`-Schleife ausgeführt wird.
 - Dann werden in der `foreach`-Schleife alle Knoten v in der Adjazenzliste (Ausgangsnachbarn) von u betrachtet.

- Das sind genau $\deg^+(u)$ viele. Daher wird die Schleife $\deg^+(u)$ -mal durchlaufen.
- Da jede Kante (u, v) genau einmal betrachtet wird (wenn u aus L entnommen wird), ist die Gesamtzahl der Durchläufe der inneren `foreach`-Schleife $\sum_{u \in V} \deg^+(u) = m$.
- Jeder Knoten wird höchstens einmal in L eingefügt.

Die Gesamlaufzeit beträgt somit $O(n + m)$.

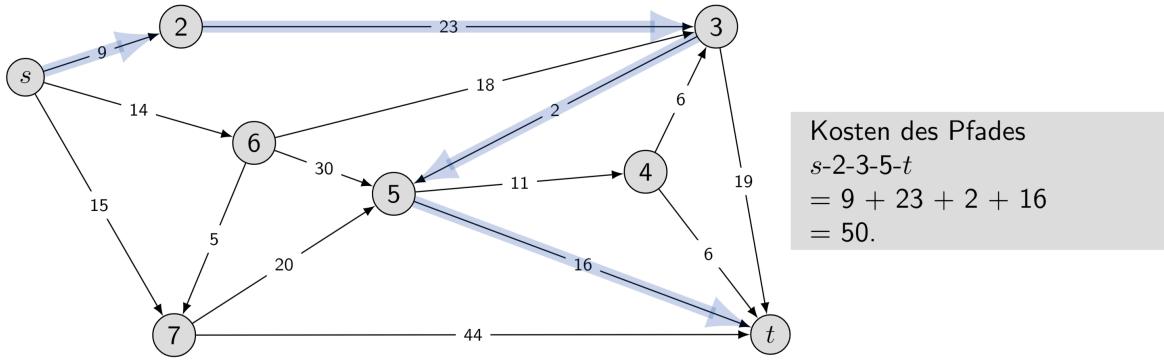
Kürzester Pfad (Shortest Path Problem)

Netzwerk für kürzesten Pfad:

- Gerichteter Graph $G = (V, E)$.
- Start s , Ziel t .
- Länge $\ell_e \geq 0$ ist die Länge der Kante e (Gewicht).

Kürzester Pfad: Finde **kürzesten** gerichteten Pfad von s nach t .

- **Kürzester Pfad = Pfad mit den geringsten Kosten, wobei die Kosten eines Pfades die Summe der Gewichte seiner Kanten sind.**



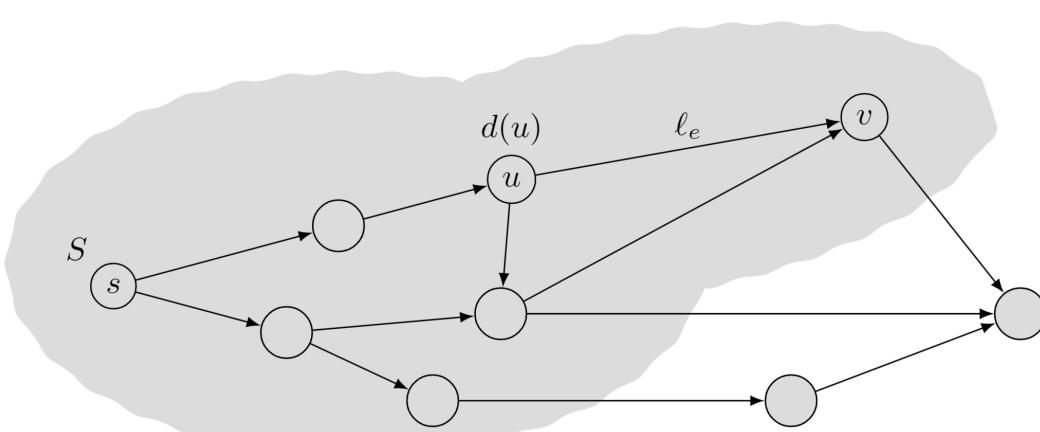
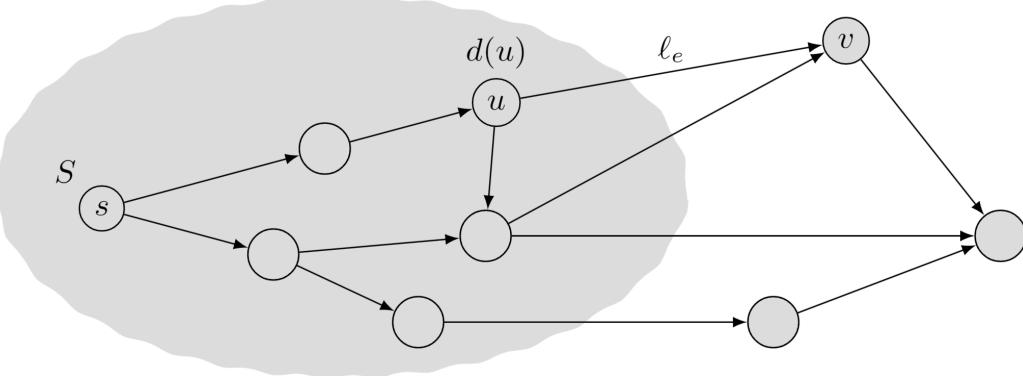
Algorithmus von Dijkstra:

- Verwalte eine Menge S von **untersuchten Knoten**, für die wir die Kosten $d(u)$ eines kürzesten $s-u$ -Pfades ermittelt haben.
- Initialisiere $S = \{s\}$, $d(s) = 0$.
- Wähle wiederholt einen nicht untersuchten Knoten v , für den der folgende Wert am kleinsten ist:

$$\min_{e=(u,v):u \in S} d(u) + l_e,$$

d.h. die Länge eines kürzesten Pfades zu einem u im untersuchten Teil des Graphen, gefolgt von einer einzigen Kante (u, v) .

- Füge v zu S hinzu und setze $d(v) = \min_{e=(u,v):u \in S} d(u) + l_e$.
- Extrahiere den Pfades entweder durch Merken des Vorgängerknotens oder mittels eines eigenen Algorithmus, der nach Dijkstra ausgeführt wird.



Korrektheitsbeweis

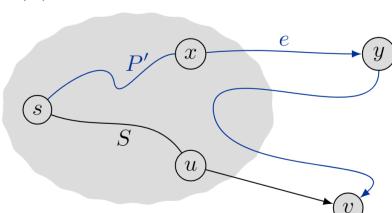
Invariante: Für jeden Knoten $u \in S$, ist $d(u)$ die Länge eines kürzesten $s-u$ Pfades.

Beweis: (durch Induktion nach $|S|$)

Induktionsanfang: $|S| = 1$ ist trivial.

Induktionsbehauptung: Angenommen, wahr für $|S| = k \geq 1$.

- Sei v der nächste zu S hinzugefügte Knoten und sei (u, v) die gewählte Kante.
- Ein kürzester $s-u$ Pfad plus (u, v) ist ein $s-v$ Pfad der Länge $d(v)$.
- Wir betrachten einen beliebigen $s-v$ Pfad P . Wir werden zeigen, dass er nicht kürzer als $d(v)$ ist.
- Sei $e = (x, y)$ die erste Kante in P die S verlässt und sei P' der Teilpfad zu x .
- P ist schon zu lange, wenn er S verlässt.



$$\text{Länge}(P) \geq \text{Länge}(P') + \ell_e \geq d(x) + \ell_e \geq d(y) \geq d(v)$$

█ Nicht-negative Gewichte
 █ Induktionsbehauptung
 █ Definition von $d(y)$
 █ Dijkstra-Algorithmus wählt v anstatt y

Implementierung

- **Implementierung:** Wir werden eine Priority Queue (Vorrangwarteschlange) für S betrachten:

- Eine Vorrangwarteschlange verwaltet (priority queue) für nicht untersuchten Knoten, geordnet nach den Kosten d .
- Ein Eintrag in der Queue besteht aus dem Knotenindex und den dazugehörigen Kosten d .
- **Algorithmus:** Arrays `Discovered` und `d`, Graph $G = (V, E)$, Liste L , Startknoten s .

```
Dijkstra(G, s):
    Discovered[v] ← false für alle Knoten  $v \in V$ 
     $d[s] \leftarrow 0$ 
     $d[v] \leftarrow \infty$  für alle anderen Knoten  $v \in V \setminus \{s\}$ 
     $L \leftarrow V$ 
    while  $L$  ist nicht leer
        wähle  $u \in L$  mit kleinstem Wert  $d[u]$ 
        lösche  $u$  aus  $L$ 
        Discovered[u] ← true
        foreach Kante  $e = (u, v) \in E$ 
            if !Discovered[v]
                 $d[v] \leftarrow \min(d[v], d[u] + \ell_e)$ 
```

Dijkstra-Algorithmus mit Liste

- **Theorem:** Der Dijkstra-Algorithmus, implementiert mit einer Liste, hat eine Worst-Case-Laufzeit von $O(n^2)$.
- **Laufzeiten:**
 - Initialisierung der Arrays benötigt $O(n)$ Zeit.
 - Die `while`-Schleife wird n -mal ausgeführt und darin muss in jeder Iteration der Knoten u mit dem kleinsten Wert für $d[u]$ gefunden werden. Das liegt in $O(n)$ Zeit.
 - Die `foreach`-Schleife wird insgesamt (über alle Iterationen der `while`-Schleife) höchstens m -mal ausgeführt. Für jeden Knoten werden seine ausgehenden Kanten nur einmal betrachtet und insgesamt gibt es nur m Kanten.
 - Daher beträgt die Laufzeit $O(n + n^2 + m)$ und somit $O(n^2)$ (da m im schlimmsten Fall $O(n^2)$ sein kann).

Wir werden sehen, dass der Dijkstra-Algorithmus mit einer Worst-Case-Laufzeit von $O((n + m) \log n)$ implementiert werden kann. Für lichte Graphen ist das effizienter als $O(n^2)$.

Priority Queue (Vorrangwarteschlange)

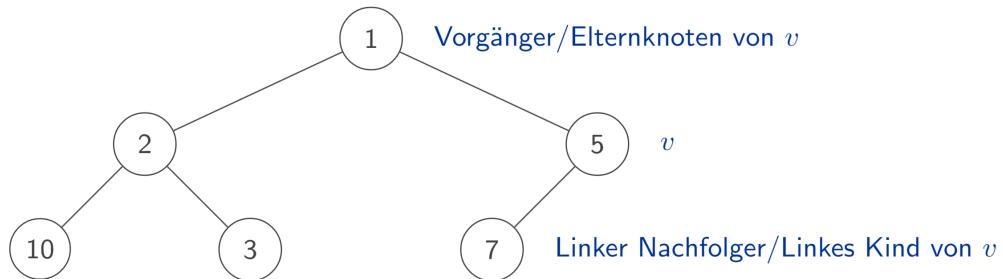
- **Priority Queue:**
 - Eine Priority Queue ist eine Datenstruktur, die eine Menge S von Elementen verwaltet.
 - Jedes Element in S hat einen dazugehörigen Wert, der die Priorität von Elementen beschreibt.

- Kleinere Werte repräsentieren höhere Prioritäten.
 - **Operationen:** Alle mit Laufzeit in $O(\log n)$.
 - Einfügen eines Elements in die Menge S .
 - Erhöhen eines Elements in der Menge S .
 - Finden eines Elements mit dem kleinsten Wert (höchster Priorität).
 - Entfernen eines Elements mit dem kleinsten Wert (höchster Priorität).
 - **Frage:** Wie erreicht man eine Laufzeit in $O(\log n)$?
 - Antwort: Mit einer bestimmten Datenstruktur, dem Heap.

Heap

- **Heap (Min-Heap):** Ein binärer Wurzelbaum, dessen Knoten mit \leq total geordnet sind, sodass gilt:
 - Für jeden Knoten v (außer der Wurzel) mit Elternknoten u , dann gilt $d[u] \leq d[v]$ (Heap-Eigenschaft für Min-Heap).
 - Alle Ebenen von Knoten bis auf die letzte sind vollständig aufgefüllt.
 - Die letzte Ebene des Baumes muss linksbündig aufgefüllt werden.
 - **Beispiel:**

Beispiel:

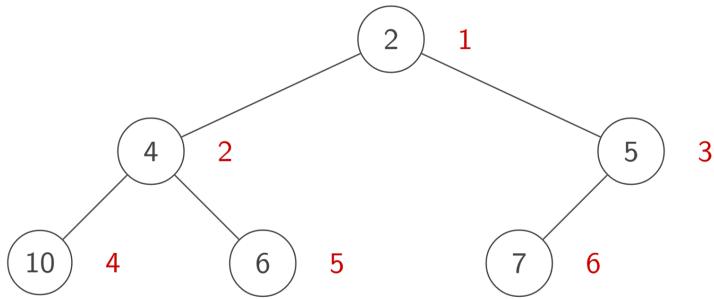


Repräsentation eines Heaps

- **Effiziente Repräsentation:** Knoten des Baums ebenenweise in einem Array speichern.
 - **Effiziente Berechnung:**
 - Die beiden Nachfolgerknoten eines Knotens an der Position k befinden sich an den Positionen $2k + 1$ und $2k + 2$. Sein Elternknoten befindet sich an der Position $\lfloor \frac{k-1}{2} \rfloor$.
 - Damit obige Rechnung immer funktioniert, wird das Array ab Index 0 belegt.
 - Würde man bei Index 1 0 anfangen, würden die Berechnungen folgendermaßen aussehen: Nachfolger links auf $2k$, Nachfolger rechts auf $2k + 1$, Elternknoten auf $\lfloor \frac{k}{2} \rfloor$.

Beispiel

Heap:



Array: 6 Einträge, erster Platz unbelegt (mit 0 initialisiert).

Index	0	1	2	3	4	5	6
Wert	0	2	4	5	10	6	7

Heapify-up

- **Fügen eines neuen Elements:** Bei einem Heap mit n Elementen wird das neue Element an Position $n + 1$ eingefügt. Wir gehen davon aus, dass noch genügend Platz im Array frei sind.
- **Heap-Bedingung:** Die Heap-Bedingung muss für das neue Element wieder hergestellt werden.
- **Reparieren:** Durch Operation Heapify-up (für Heap-Array H an Position i). Aufruf nach Einfügen des neuen Elements: $\text{Heapify-up}(H, n)$.

Heapify-up(H, i):

if $i > 1$

$j \leftarrow \lfloor i/2 \rfloor$

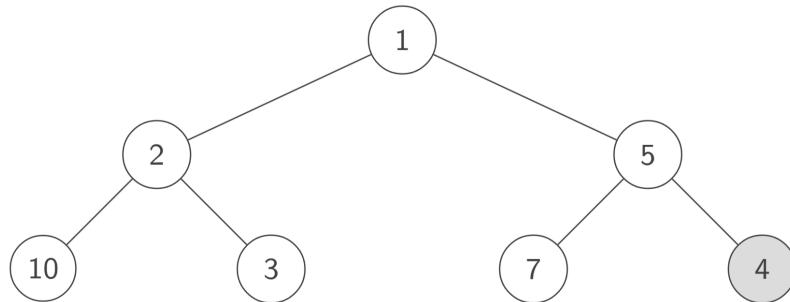
if $H[i] < H[j]$

 Vertausche die Array-Einträge $H[i]$ und $H[j]$

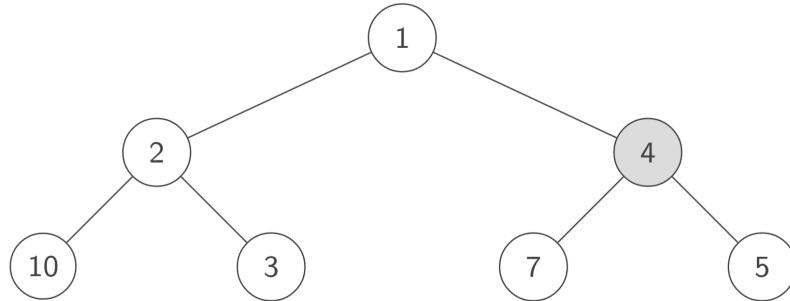
 Heapify-up(H, j)

Beispiel

Einfügen von 4:



Verschieben von 4:



Heapify-down

- **Löschen eines Elements:** Element wird an Stelle i gelöscht. Das Element an Stelle n (bei n Elementen) wird an die freie Stelle verschoben.
- **Heap-Bedingung:** Die Heap-Bedingung kann durch das neue Element an der Stelle i verletzt sein.
- **Reparieren:**
 - Eingefügtes Element ist zu groß: Benutze Heapify-down, um das Element auf eine untere Ebene zu bringen.
 - Eingefügtes Element ist zu klein: Benutze Heapify-up (wie beim Einfügen) von der Stelle i aus.
- **Hinweis:** Beim Heap wird typischerweise die Wurzel entfernt und daher wird nur Heapify-down benutzt.
- **Laufzeit für Löschen:** Für Heap-Array H an Position i in $O(\log n)$ Zeit.

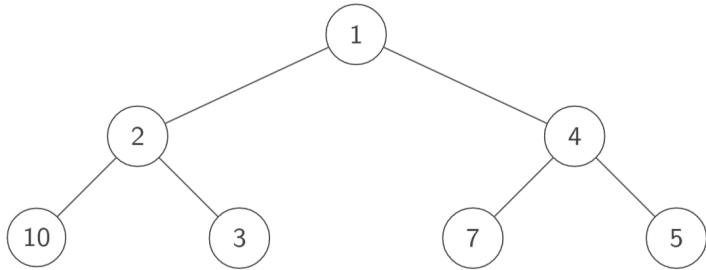
```

Heapify-down(H, i):
n ← length(H)-1
if  $2 \cdot i > n$ 
    return
elseif  $2 \cdot i < n$ 
    left ←  $2 \cdot i$ , right ←  $2 \cdot i + 1$ 
    j ← Index des kleineren Wertes von H[left] und H[right]
else
    j ←  $2 \cdot i$ 
if H[j] < H[i]
    Vertausche die Arrayeinträge H[i] und H[j]
    Heapify-down(H, j)

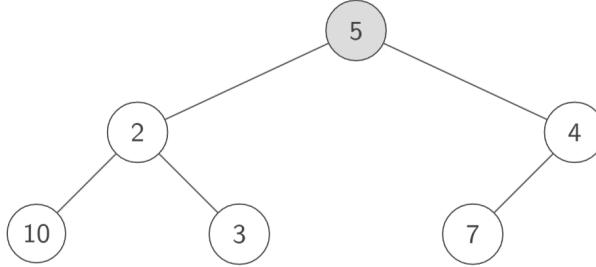
```

Beispiel

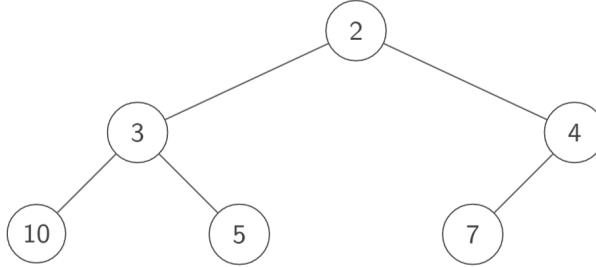
Ursprünglicher Heap:



Löschen von 1, verschieben von 5:



Heapify-down (zwei Mal)



Operationen auf Heap

- **Insert(H, x):** Fügt Element x in den Heap H ein. Hat der Heap n Elemente, dann liegt die Laufzeit in $O(\log n)$.
- **FindMin(H):** Findet das Minimum im Heap H . Laufzeit ist konstant ($O(1)$, da Wurzel).

- **Delete(H, i):** Löscht das Element im Heap H an der Stelle i . Für einen Heap mit n Elementen liegt die Laufzeit in $O(\log n)$.
- **ExtractMin(H):** Kombination von FindMin und Delete und daher in $O(\log n)$.

Erstellen eines Heaps

- **Erstellen:** Das Erstellen eines Heaps aus einem Array A mit Größe n , das noch nicht die Heapeigenschaft erfüllt:

```
Init(A, n):
for  $i = \lfloor n/2 \rfloor$  bis 1
    Heapify-down(A, i)
```

Analyse

Laufzeit: $O(n)$ ergibt sich aus folgender Berechnung:

- Einfachheitshalber nehmen wir an, der Binärbaum ist vollständig und hat n Knoten.
- Es folgt, dass $n = 2^{h+1} - 1$ wobei h die Höhe des Baumes ergibt.
- Wir lassen den Index j über die Ebenen E_j des Baumes laufen, wobei mit E_0 die Ebene mit den Blättern des Baumes bezeichnet und E_h die Ebene mit der Wurzel.
- Es folgt, dass Ebene E_j genau 2^{h-j} Knoten enthält und der Aufwand zum Einfügen eines Elements auf Ebene E_j maximal proportional zu j ist.
- Insgesamt ergibt sich also ein Maximalaufwand von $\sum_{j=0}^h j2^{h-j}$, den wir folgendermaßen abschätzen:

$$\sum_{j=0}^h j2^{h-j} = \sum_{j=0}^h j \frac{2^h}{2^j} = 2^h \sum_{j=0}^h \frac{j}{2^j} \leq 2^h 2 = 2^{h+1} = n + 1 = O(n)$$

■ folgt aus $\sum_{i=1}^{\infty} \frac{i}{2^i} = 2$. ■ da $n = 2^{h+1} - 1$.

Dijkstra-Algorithmus: Effiziente Variante

- **Algorithmus:** Arrays `Discovered` und `d`, Graph $G = (V, E)$, Startknoten s .
- **Verwende Vorrangwarteschlange Q , in der die Knoten v nach dem Wert $d[v]$ geordnet sind.**

```

Dijkstra( $G, s$ ):
  Discovered[ $v$ ]  $\leftarrow$  false für alle Knoten  $v \in V$ 
   $d[s] \leftarrow 0$ 
   $d[v] \leftarrow \infty$  für alle anderen Knoten  $v \in V \setminus \{s\}$ 
   $Q \leftarrow V$ 
  while  $Q$  ist nicht leer
    wähle  $u \in Q$  mit kleinstem Wert  $d[u]$ 
    lösche  $u$  aus  $Q$ 
    Discovered[ $u$ ]  $\leftarrow$  true
    foreach Kante  $e = (u, v) \in E$ 
      if !Discovered[ $v$ ]
        if  $d[v] > d[u] + \ell_e$ 
          lösche  $v$  aus  $Q$ 
           $d[v] \leftarrow d[u] + \ell_e$ 
          füge  $v$  zu  $Q$  hinzu

```

Dijkstra-Algorithmus mit Vorrangwarteschlange

- **Theorem:** Der Dijkstra-Algorithmus, implementiert mit einer Vorrangwarteschlange, hat eine Worst-Case-Laufzeit von $O((n + m) \log n)$.
- **Laufzeiten:**
 - Initialisierung der Arrays benötigt $O(n)$ Zeit.
 - Die `while`-Schleife wird n -mal ausgeführt (da jeder Knoten einmal aus der Queue entfernt wird).
 - Das Finden des Knotens u mit dem kleinsten $d[u]$ in der Queue benötigt $O(\log n)$ Zeit (mittels ExtractMin-Operation).
 - Die `foreach`-Schleife liegt in $O(\deg(u))$ Zeit. Für jeden Knoten werden seine ausgehenden Kanten nur einmal betrachtet und insgesamt gibt es m Kanten.
 - Die Operation des Reduzierens der Distanz $d[v]$ (falls ein kürzerer Pfad gefunden wird) und das Aktualisieren der Priorität in der Queue (DecreaseKey oder ähnliche Operation) benötigt $O(\log n)$ Zeit.
 - Daher beträgt die Laufzeit $O(n + n \log n + m \log n)$ und somit $O((n + m) \log n)$.

Dijkstra Algorithmus Vergleich

Wir vergleichen die Laufzeit des Dijkstra-Algorithmus bei Verwendung von Listen, Vorrangwarteschlange als Heap und Vorrangwarteschlange als Fibonacci-Heap (diese verbesserte Datenstruktur haben wir nicht besprochen).

Tabelle: Vergleich verschiedener Datenstrukturen für Dijkstra-Algorithmus.

Liste	Heap	FibHeap
$O(n^2)$	$O((n + m) \log n)$	$O(m + n \log n)$

4. Greedy-Algorithmen

Einleitung

Einführendes Beispiel

- **Geld wechseln:** Gegeben sei eine Stückelung von Münzen (z.B. Euromünzen in Cent): 1, 2, 5, 10, 20, 50, 100, 200.
- **Gesucht:** Methode, um einen Betrag mit der kleinstmöglichen Anzahl an Münzen herauszugeben.
- **Beispiel:**
 - 37 Cent
 - Optimale Lösung: $1 \times 20, 1 \times 10, 1 \times 5, 1 \times 2$
- **Hinweis:** Es kann auch mehr als eine Lösung geben.
 - Stückelung von Münzen: 1, 5, 10, 20, 25, 50
 - Betrag: 30
 - 1×20 und 1×10 sowie 1×25 und 1×5 sind optimale Lösungen.

Greedy-Algorithmus

- **Greedy-Ansatz:** Für Betrag S .

```
while  $S \neq 0$ 
  Finde die Münze mit größtem Wert  $x$ , sodass  $x \leq S$ 
  Benutze  $\lfloor S/x \rfloor$  Münzen von Wert  $x$ 
   $S \leftarrow S \bmod x$ 
```

- **Greedy-Ansatz konkreter:**

- Werte von m Münzen in einem Array w .
- Es gilt $w[0] > w[1] > \dots > w[m - 1] = 1$.
- Betrag S gegeben.
- Anzahl jeder einzelnen Münze, um S zu wechseln, wird in einem Array `num` gespeichert.
- `num[i]` enthält Anzahl der Münzen von Wert $w[i]$.

```
for  $i \leftarrow 0$  bis  $m - 1$ 
   $\text{num}[i] \leftarrow \lfloor \frac{S}{w[i]} \rfloor$ 
   $S \leftarrow S \bmod w[i]$ 
```

Allgemeines zu Greedy-Algorithmen

- **Greedy-Algorithmus:** Eine Lösung wird schrittweise aufgebaut, in jedem Schritt wird das Problem auf ein kleineres Problem reduziert.
- **Greedy-Prinzip:** Füge jeweils eine lokal am attraktivsten erscheinende Lösungskomponente hinzu.
- Einmal getroffene Entscheidungen werden nicht mehr zurückgenommen.
- Meist einfach zu konstruieren und zu implementieren.
- Kann eine optimale Lösung liefern, muss es i.A. aber nicht.

Optimalität

Optimale Lösung: Für eine Stückelung von 1, 5 und 10 Cent gezeigt, dass der Greedy-Algorithmus eine optimale Lösung liefert.

Beweis:

- Wir gehen von irgendeiner optimalen Lösung aus.
- Die Lösung kann nicht mehr als vier 1er haben, da fünf davon durch einen 5er ersetzt werden könnten.
- Die Lösung kann auch nicht mehr als einen 5er haben, da zwei davon durch einen 10er ersetzt werden könnten.
- Daher muss die Anzahl der 10er im Greedy-Algorithmus und in einem optimalen Algorithmus gleich sein.
- Die Anzahl der restlichen Münzen kann dann maximal 9 ergeben.
- Daher muss man nur den Fall ≤ 9 betrachten.
- Jeder Betrag ≤ 9 kann nur durch 1er abgedeckt werden und der Greedy-Algorithmus und der Greedy-Algorithmus benutzen die gleiche Anzahl von 1er.
- Wenn der Betrag zwischen 5 und 9 (beide inklusive) ist, dann haben der optimale Algorithmus und der Greedy-Algorithmus genau einen 5er und der Rest wird mit 1ern aufgefüllt.
- Der Greedy-Algorithmus liefert daher die gleiche Anzahl an Münzen wie die optimale Lösung.

Hinweis: Für Euromünzen kann ähnlich gezeigt werden, dass der Greedy-Algorithmus optimal ist.

Nicht optimal:

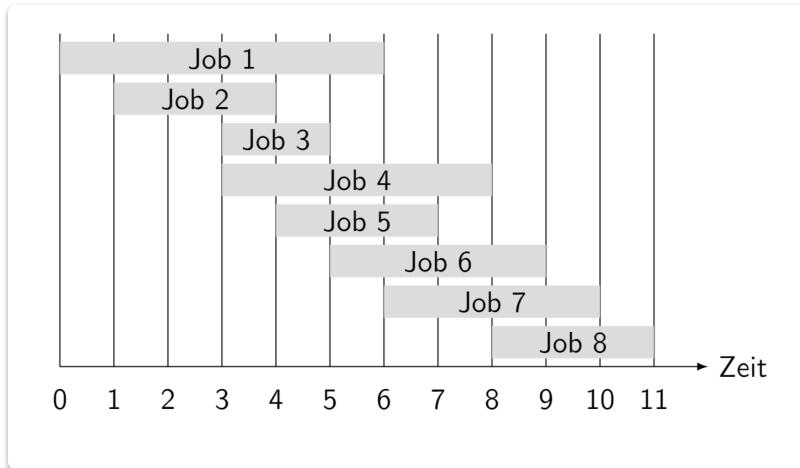
- Gegeben sei eine Stückelung von 1, 5, 10, 20, 25.
- Bei dieser Stückelung liefert der Greedy-Algorithmus nicht immer eine optimale Lösung.

Beispiel: Mit $S = 40$.

- Greedy-Algorithmus liefert $1 \times 25, 1 \times 10, 1 \times 5$. Optimale Lösung ist 2×20 .

Zeitplanung von Jobs (Interval Scheduling)

- **Gegeben:** Jobs $j = 1, \dots, n$.
- Jeder Job j startet zum Zeitpunkt s_j und endet zum Zeitpunkt f_j .
- Zwei Jobs sind **kompatibel**, wenn sie sich nicht überlappen.
- **Ziel:** Finde größte Teilmenge von paarweise kompatiblen Jobs.



Beispiele: Job 2 und 5 sind kompatibel, Job 2 und 3 sind nicht kompatibel.

Interval Scheduling: Greedy-Algorithmus

Greedy-Ansatz: Betrachte die Jobs in einer natürlichen Ordnung. Wähle einen Job wenn er kompatibel (nicht überlappend) mit den bisher gewählten Jobs ist.

Mögliche Greedy-Strategien:

- **Früheste Startzeit:** Berücksichtige Jobs in aufsteigender Reihenfolge von s_j .
- **Kürzestes Intervall:** Berücksichtige Jobs in aufsteigender Reihenfolge von $f_j - s_j$.
- **Wenigste Konflikte:** Zähle für jeden Job j die Anzahl c_j der nicht kompatiblen Jobs. Berücksichtige Jobs in aufsteigender Reihenfolge von c_j .

Greedy-Ansatz: Betrachte die Jobs in einer natürlichen Ordnung. Wähle einen Job wenn er kompatibel (nicht überlappend) mit den bisher gewählten Jobs ist.



Früheste Beendigungszeit: Gegenbeispiel? Nein!

Greedy-Algorithmus: Berücksichtige Jobs in aufsteigender Reihenfolge der Beendigungszeit. Wähle einen Job, wenn er kompatibel mit den bisher gewählten Jobs ist.

```

Sortiere Jobs nach Beendigungszeit, sodass  $f_1 \leq f_2 \leq \dots \leq f_n$ 
 $A \leftarrow \emptyset$ 
for  $j \leftarrow 1$  bis  $n$ 
    if Job  $j$  ist kompatibel zu  $A$ 
         $A \leftarrow A \cup \{j\}$ 
return  $A$ 

```

■ Menge der ausgewählten Jobs

Greedy-Algorithmus: Pseudocode mit angepassten Indexwerten und Array.

```

Sortiere Jobs nach Beendigungszeit, sodass  $f_1 \leq f_2 \leq \dots \leq f_n$ 
 $A \leftarrow \emptyset$ 
 $t \leftarrow 0$ 
for  $j \leftarrow 1$  bis  $n$ 
    if  $t \leq s_j$ 
         $A \leftarrow A \cup \{j\}$ 
         $t \leftarrow f_j$ 
return  $A$ 

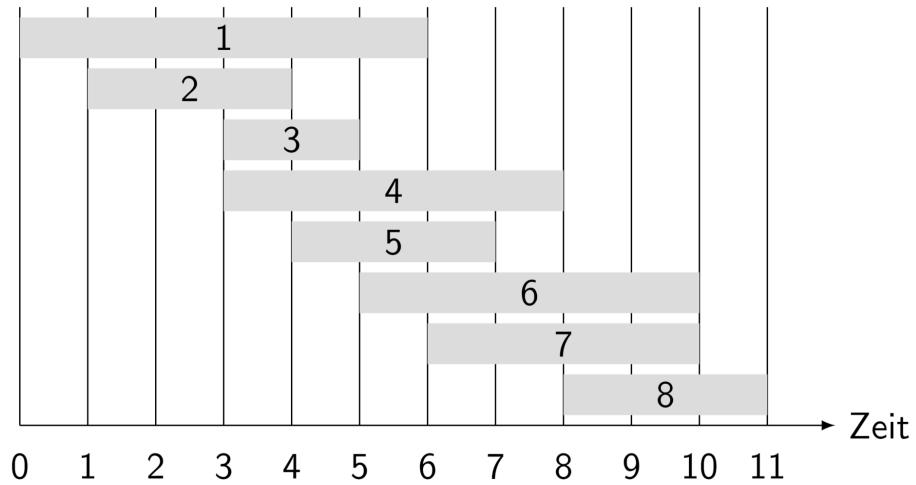
```

Implementierung

Implementierung: Laufzeit in $O(n \log n)$.

- Jobs werden nach Beendigungszeit sortiert und nummeriert. Wenn $f_i \leq f_j$, dann $i < j$. Die Sortierung läuft in $O(n \log n)$.
- Jobs werden vom ersten Job beginnend in der Reihenfolge ansteigender Werte für f_j untersucht.
- Sei S die Beendigungszeit des aktuellen Jobs i :
 - Dann wird in den nachfolgenden Jobs der erste Job j gesucht, für den gilt: $s_j \geq S$. Wenn so ein Job existiert, wird er zur aktuellen Lösung hinzugefügt und die Suche wird von diesem Job aus fortgesetzt.
- Der Greedy-Algorithmus kann in einem Durchlauf realisiert werden, d.h. die Laufzeit ohne Sortieren liegt in $O(n)$.
- Somit liegt die Gesamlaufzeit in $O(n \log n)$.

Beispiel



Jobs: Nach Beendigungszeit sortiert

Job i	2	3	1	5	4	6	7	8
s_i	1	3	0	4	3	5	6	8
f_i	4	5	6	7	8	10	10	11

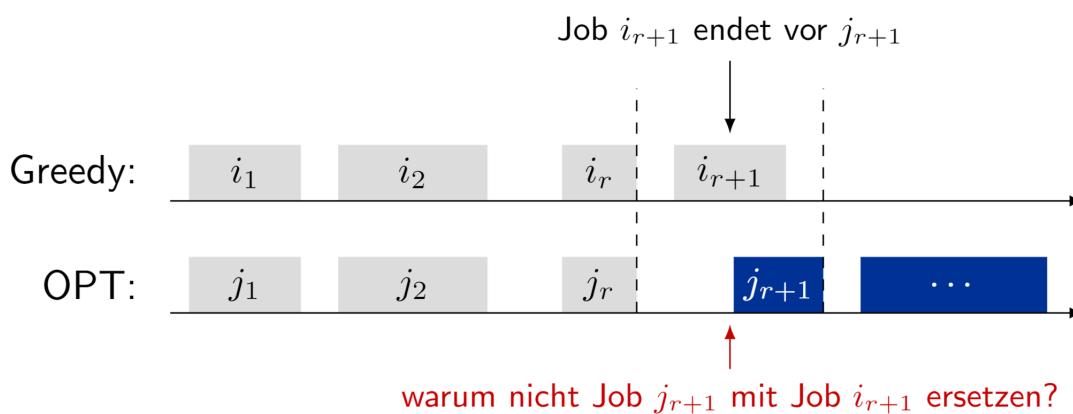
Lösung: Jobs 2, 5 und 8

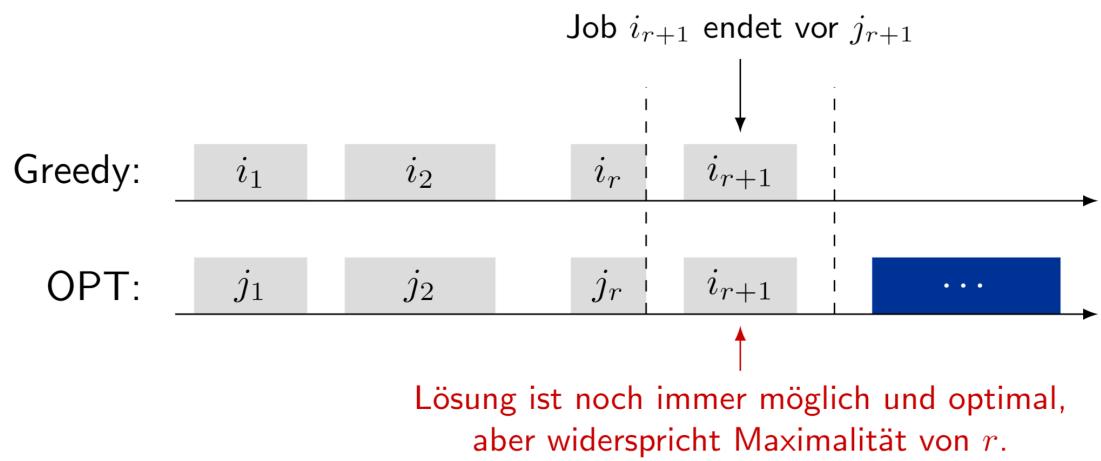
Analyse

Theorem: Der Greedy-Algorithmus liefert immer eine optimale Lösung.

Beweis: (durch Widerspruch)

- Angenommen, der Algorithmus liefert keine optimale Lösung.
- Sei i_1, i_2, \dots, i_k die Menge von Jobs, die vom Algorithmus ausgewählt wird.
- Sei j_1, j_2, \dots, j_m die Menge von Jobs in einer optimalen Lösung mit $k < m$.
- Sei i_r der erste Job in der algorithmischen Lösung, sodass $f_{i_r} \geq f_{j_r}$.

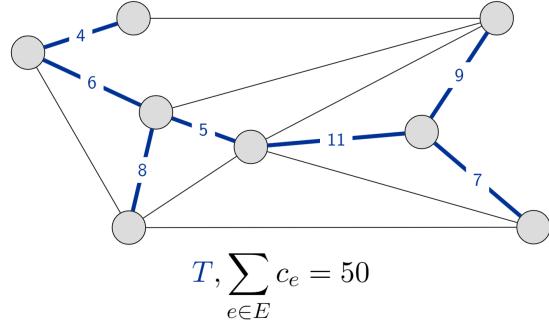
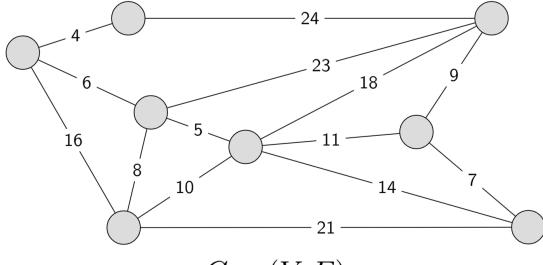




Minimaler Spannbaum

Gegeben: Ein zusammenhängender schlichter Graph $G = (V, E)$ mit reellwertigen Kantengewichten $c_e = c_{uv} = c_{vu}$ für $e = (u, v) \in E$.

Minimaler Spannbaum: Ein minimaler Spannbaum (*Minimum Spanning Tree, MST*) ist ein Teilgraph $G_T = (V, T)$ von G mit gleicher Knotenmenge und einer Teilmenge der Kanten $T \subseteq E$, sodass er ein aufspannender Baum mit minimaler Summe der Kantengewichte ist.



MST-Problem

MST-Problem: Finde in einem zusammenhängenden schlichten Graphen $G = (V, E)$ mit reellwertigen Kantengewichten c_e einen minimalen Spannbaum, d.h. einen zusammenhängenden, zyklenfreien Untergraphen $G_T = (V, T)$ mit $T \subseteq E$, dessen Kanten alle Knoten aufspannen und für den $\text{cost}(T) = \sum_{e \in T} c_e$ so klein wie möglich ist.

Aufwand: Es gibt exponentiell viele Spannbäume und daher wäre eine Brute-Force-Durchprobieren aller Spannbäume nicht effizient.

Lösung: Algorithmen, die in diesem Abschnitt vorgestellt werden.

Anwendungen

Das MST-Problem ist ein fundamentales Problem mit vielen unterschiedlichen Anwendungen:

- Basis für den Entwurf von Netzwerken:
 - Telefonnetz, Elektrizität, Kabelfernsehen, Computernetz, Straßenverkehrsnetz
- Approximationsalgorithmen für schwere Probleme:
 - Problem des Handlungsreisenden (Traveling Salesperson Problem), Steinerbaum Problem

Greedy-Algorithmen

Algorithmen:

- **Algorithmus von Prim:** Starte mit einem beliebigen Startknoten s . Füge in jedem Schritt eine billigste Kante $e = \{u, v\}$ zu T hinzu, die genau einen noch nicht angebundenen

Knoten mit dem bisherigen Baum verbindet.

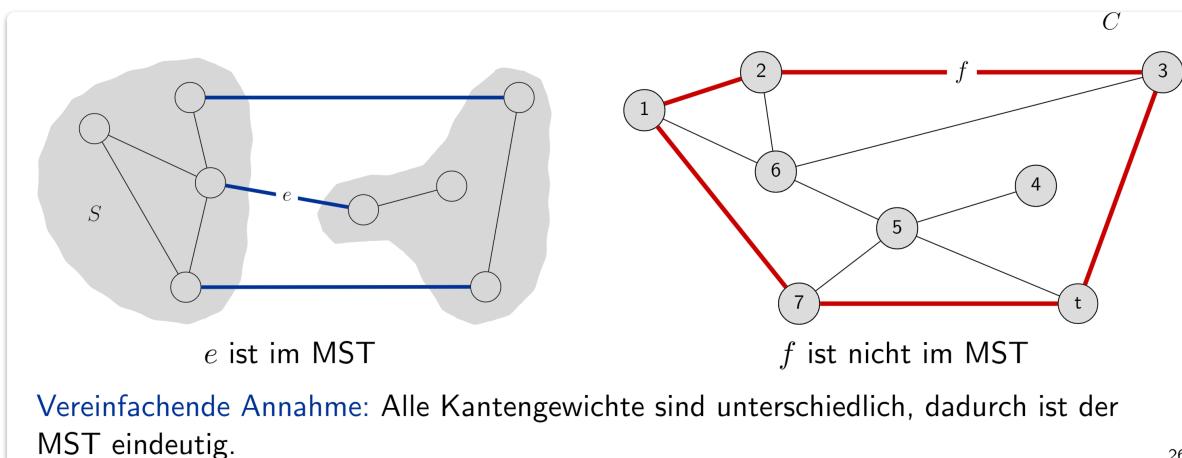
- **Algorithmus von Kruskal:** Starte mit $T = \emptyset$. Betrachte die Kanten in aufsteigender Reihenfolge ihrer Kosten. Füge Kante e nur dann zu T hinzu, wenn dadurch kein Kreis erzeugt wird.

Beide Algorithmen erzeugen immer einen MST.

Lemmata

Kantenschnittlemma: Sei $S \subseteq V$ eine beliebige Teilmenge von Knoten und sei e die minimale gewichtete Kante mit genau einem Endknoten in S . Dann enthält der MST T die Kante e .

Kreislemma: Sei C ein beliebiger Kreis und sei f die maximal gewichtete Kante in C . Dann enthält der MST T die Kante f nicht.



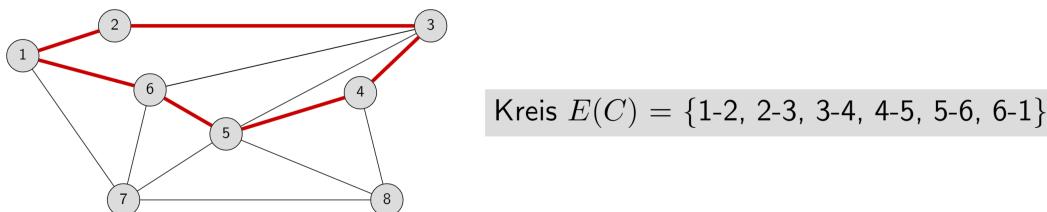
Vereinfachende Annahme: Alle Kantengewichte sind unterschiedlich, dadurch ist der MST eindeutig.

26

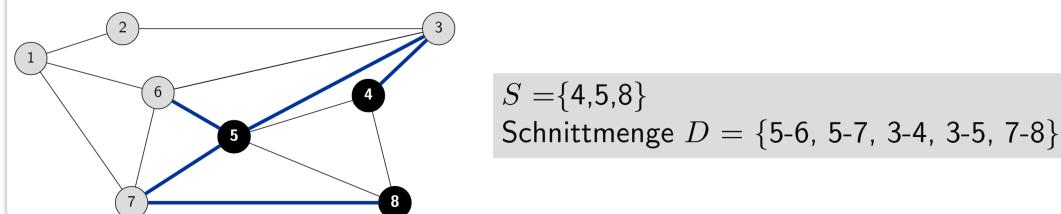
Kreise und Schnitte

Kreise und Schnitte

Kreis: Ein Kreis ist ein Kantenzug $v_1, v_2, \dots, v_{k-1}, v_k$ in dem $v_1 = v_k$, $k \geq 4$, und die ersten $k - 1$ Knoten alle unterschiedlich sind. Alternativ kann ein Kreis als Menge $E(C)$ von Kanten der Form $a-b$, $b-c$, $c-d$, \dots , $y-z$, $z-a$ gesehen werden.



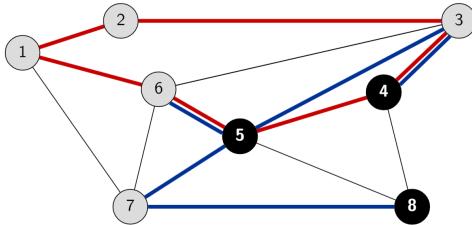
Kantenschnittmenge: Sei S eine Teilmenge der Knoten. Die dazugehörige Kantenschnittmenge D ist die Menge jener Kanten, die genau einen Endpunkt in S haben.



27

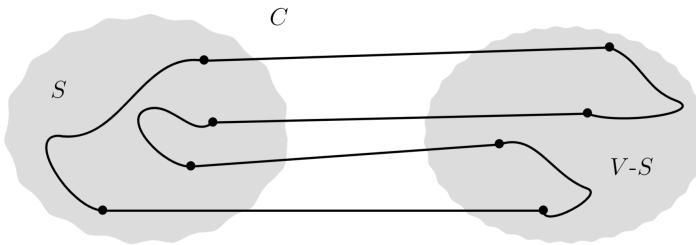
Paritätslemma

Behauptung: Ein beliebiger Kreis und eine beliebige Kantenschnittmenge haben eine gerade Anzahl von Kanten gemeinsam.



Kreis $E(C) = \{1-2, 2-3, 3-4, 4-5, 5-6, 6-1\}$
 Schnittmenge $D = \{3-4, 3-5, 5-6, 5-7, 7-8\}$
 Durchschnitt = $\{3-4, 5-6\}$

Beweis: (durch Bild)



Beweis des Kantenschnittlemmas

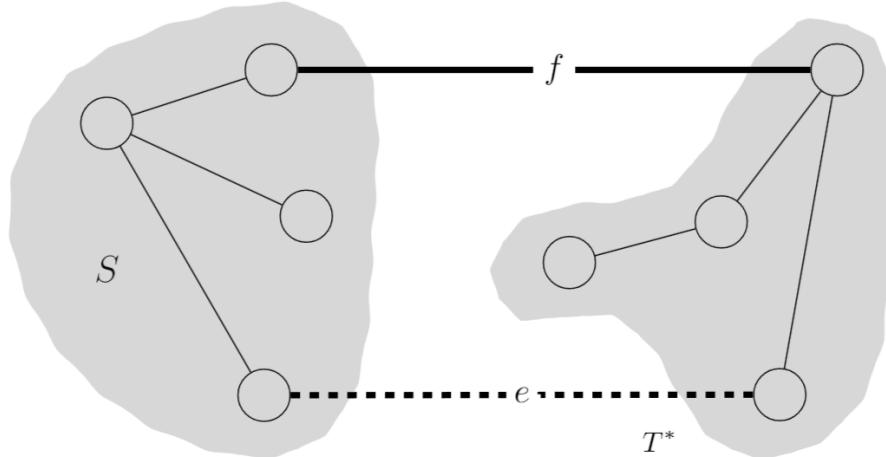
Kantenschnittlemma: Sei $S \subseteq V$ eine beliebige Teilmenge von Knoten und sei e die minimale gewichtete Kante mit genau einem Endknoten in S . Dann enthält der MST T^* die Kante e .

Annahme für Beweis: Alle Kantengewichte c_e sind unterschiedlich, vereinfacht Beweis.

Hinweis: Man kann zu allen Kosten kleine Störwerte hinzufügen, um die Annahme, dass alle Kanten unterschiedliche Gewichte haben müssen, zu vermeiden.

Beweis: (Austauschargument)

- Angenommen $e \notin T^*$.
- Das Hinzufügen von e zu T^* erzeugt einen Kreis C in $T^* \cup \{e\}$.
- Da e sowohl in S als auch in der Schnittmenge D von S liegt.
- Paritätslemma \implies es existiert eine andere Kante e' in C , die sowohl in $S(G)$ als auch in $\overline{S}(G)$ verbindet.
- $T' = T^* \cup \{e\} \setminus \{e'\}$ ist auch ein Spannbaum.
- Da $c_e < c_{e'}$, folgt $cost(T') < cost(T^*)$.
- Das ist ein Widerspruch zur Annahme, dass T^* minimal ist. \square



Beweis des Kreislemmas

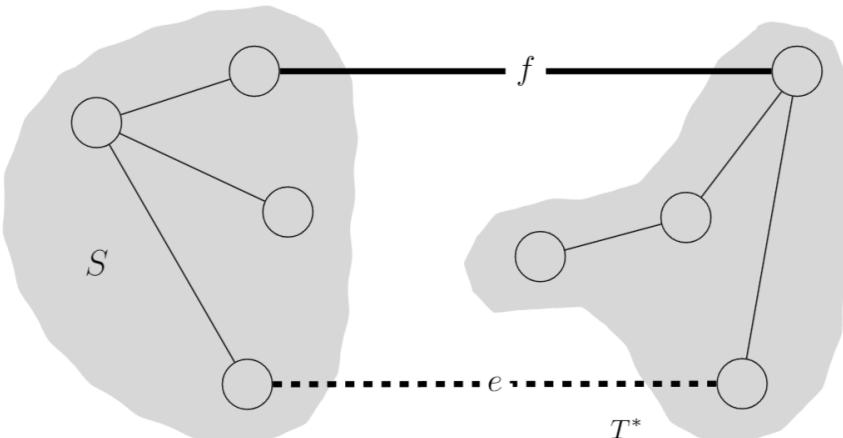
Kreislemma: Sei C ein beliebiger Kreis in G und sei f die maximal gewichtete Kante in C . Dann enthält kein MST T^* die Kante f .

Annahme für Beweis: Alle Kantengewichte c_e sind unterschiedlich, vereinfacht Beweis.

Hinweis: Man kann zu allen Kosten kleine Störwerte hinzufügen, um die Annahme, dass alle Kanten unterschiedliche Gewichte haben müssen, zu vermeiden.

Beweis: (Austauschargument)

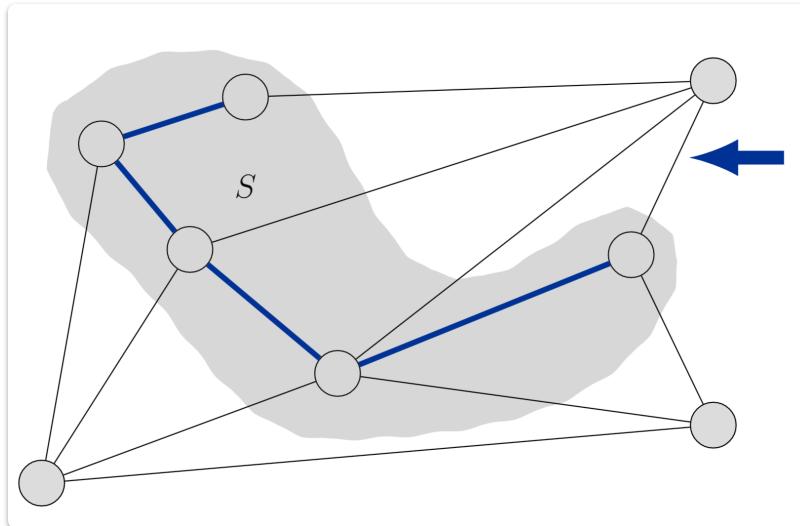
- Angenommen $f \in T^*$.
- Löschen von f aus T^* erzeugt eine Teilmenge S von Knoten in $T^* \setminus \{f\}$.
- Da f sowohl in S als auch in der Schnittmenge D von S liegt.
- Paritätslemma \implies es existiert eine andere Kante e in C , die sowohl in $S(G)$ als auch in $\overline{S}(G)$ verbindet.
- $T' = T^* \setminus \{f\} \cup \{e\}$ ist auch ein Spannbaum.
- Da $c_e < c_f$, folgt $cost(T') < cost(T^*)$.
- Das ist ein Widerspruch zur Annahme, dass T^* minimal ist. \square



Algorithmus von Prim

Algorithmus von Prim: [Jarník 1930, Dijkstra 1957, Prim 1959]

- Initialisiere S mit einem beliebigen Knoten.
- Wende das Kantenschnittlemma auf S an.
- Füge die minimal gewichtete Kante e in der Schnittmenge von S zu T hinzu und füge den Knoten v (Endknoten von e , der sich noch nicht in S befindet) zu S hinzu.



Inuitiv:

Man sucht immer eine Kante zu einem Knoten den man noch nicht in seinem Ergebnis hat und der am wenigsten Weg adden würde.

Implementierung

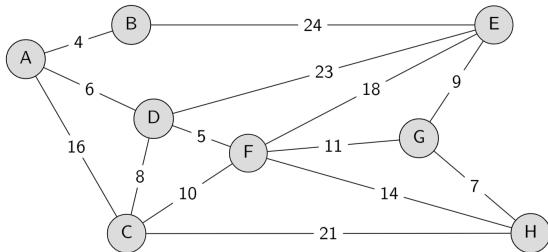
Annahme: Alle Kantengewichte sind unterschiedlich.

```

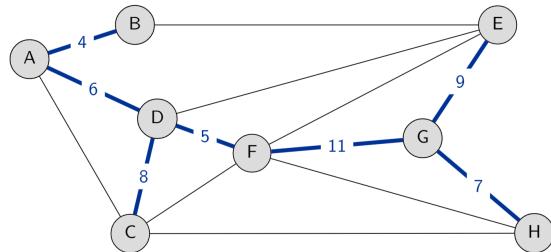
Prim( $G, c$ ):
foreach ( $v \in V$ )
     $A[v] \leftarrow \infty$ 
Initialisiere eine leere Priority Queue  $Q$ 
foreach ( $v \in V$ )
    Füge  $v$  in  $Q$  ein
 $S \leftarrow \emptyset$ 
while  $Q$  ist nicht leer
     $u \leftarrow$  entnehme minimales Element aus  $Q$ 
     $S \leftarrow S \cup \{u\}$ 
    foreach Kante  $e = (u, v)$  inzident zu  $u$ 
        if  $v \notin S$  und  $c_e < A[v]$ 
            Verringere die Priorität  $A[v]$  auf  $c_e$ 

```

Beispiel



$$G = (V, E)$$



$$\textcolor{blue}{T}, \sum_{e \in E} c_e = 50$$

Start:

- Start bei A (willkürlich gewählt, alle Knoten gleiche Priorität)
- Priority Queue zu Beginn: A, B, C, D, E, F, G, H

Ausgewählt	Resultierende Priority Queue	Knotenmenge S	Gewicht
A	B, D, C, E, F, G, H	A	0
B	D, C, E, F, G, H	A, B	4
D	F, C, E, G, H	A, B, D	10
F	C, G, H, E	A, B, D, F	15
C	G, H, E	A, B, D, F, C	23
G	H, E	A, B, D, F, C, G	34
H	E	A, B, D, F, C, G, H	41
E		A, B, D, F, C, G, H, E	50

Analyse

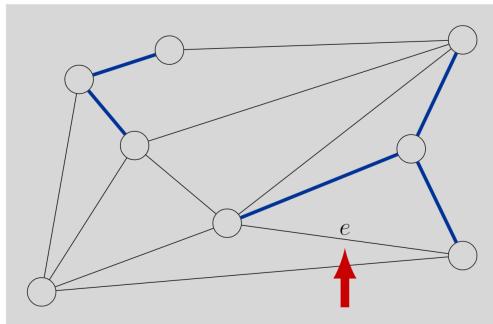
Implementierung: Benutze eine Priority Queue wie bei Dijkstra.

- Verwalte eine Menge von bearbeiteten Knoten S .
- Verwalte jeden unbearbeiteten Knoten v mit Kosten $A[v]$ in der Priority Queue.
- $A[v]$ sind die Kosten der billigsten Kante von v zu einem Knoten in S .
- Laufzeit in $O(|E| \log |V|)$, wenn die Priority Queue mit einem Array implementiert ist.
- Laufzeit in $O(|E| + |V| \log |V|)$ mit einem binären Heap (Min-Heap).

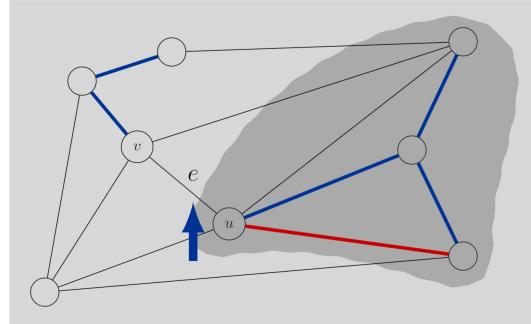
Algorithmus von Kruskal

Algorithmus von Kruskal: [Kruskal, 1956]

- Bearbeite Kanten in aufsteigender Reihenfolge der Kantengewichte.
- **Fall 1:** Wenn das Hinzufügen von $e = \{u, v\}$ zu T einen Kreis erzeugt, verwirfe e gemäß des Kreislemmas.
- **Fall 2:** Sonst füge $e = \{u, v\}$ in T gemäß des Kantenschnittlemmas ein.



Fall 1



Fall 2

Implementierung

Kruskal(G, c):

Sortiere Kantengewichte so, dass $c_1 \leq c_2 \leq \dots \leq c_m$

$T \leftarrow \emptyset$

foreach ($u \in V$) erzeuge eine einelementige Menge mit u

for $i \leftarrow 1$ bis m

$(u, v) = e_i$

if u und v sind in verschiedenen Mengen

$T \leftarrow T \cup \{e_i\}$

 Vereinige die Mengen mit u und v

return T

■ sind u und v in unterschiedlichen Zusammenhangskomponenten?

■ Vereinige zwei Komponenten

Sind u und v in unterschiedlichen Zusammenhangskomponenten?

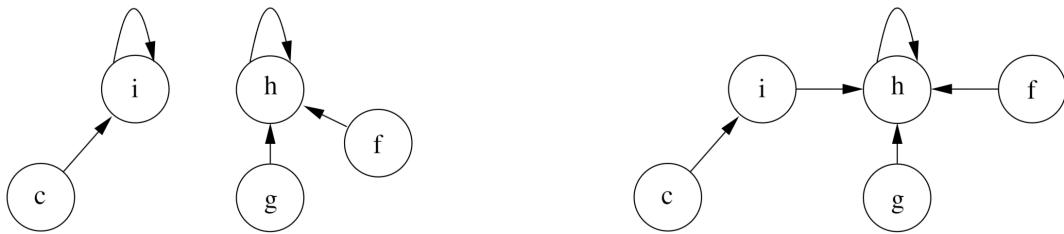
- Einfache Möglichkeit: Verwende Tiefen- oder Breitensuche.
- Effizienter: Benutze die sogenannte Union-Find-Datenstruktur.
- Verwalte die Teilmengen aller Knoten für jede Zusammenhangskomponente.
- $O(|E| \log |E|)$ für die Sortierung ($|E| \leq |V|^2 \implies \log |E| \in O(\log |V|)$).

Union-Find-Datenstruktur: Abstrakter Datentyp

Abstrakter Datentyp: Dynamische Disjunkte Mengen (DDM)

Familie: $S = \{S_1, S_2, \dots, S_k\}$ disjunkten Teilmengen einer Menge M . Jede S_i hat einen Repräsentanten.

- **makeSet(x):** Erzeugt eine Menge $\{x\}$, x ist Repräsentant von S_i .
- **union(u, v):** Vereinigt Mengen S_u und S_v , deren Repräsentanten u und v sind; neuer Repräsentant ist ein beliebiges $w \in S_u \cup S_v$.
- **findSet(x):** Liefert Repräsentanten der Menge S_i mit $x \in S_i$.



v	a	b	c	d	e	f	g	h	i
parent[v]	a	b	i	d	e	h	h	h	h

Implementierung

Einfache Implementierung:

- **makeset (v):**

```
parent[v] = v
```

- **union (v, w):**

```
parent[v] = w
```

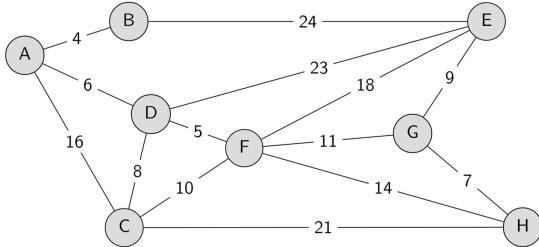
- **findset (v):**

```

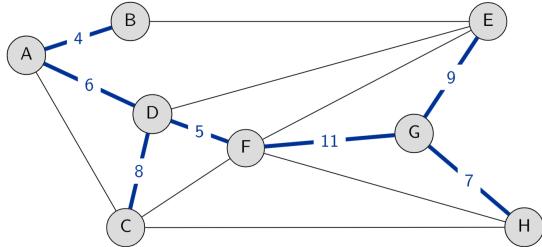
 $h = v$ 
while parent[h]  $\neq h$ 
     $h = \text{parent}[h]$ 
return  $h$ ;
```

Laufzeit: Mit einer verbesserten Implementierung kann eine **in der Praxis nahezu konstante Laufzeit** für jede der drei Operationen erreicht werden.

Beispiel



$$G = (V, E)$$



$$T, \sum_{e \in E} c_e = 50$$

Start: Kanten sortiert nach Gewicht (kleinstes zuerst): (A,B), (D,F), (A,D), (G,H), (C,D), (E,G), (C,F), (F,G), (F,H), (A,C) ...

Mengen	Kante	Hinzu?	T
{A}, {B}, {C}, {D}, {E}, {F}, {G}, {H}	(A,B)	Ja	{(A,B)}
{A,B}, {C}, {D}, {E}, {F}, {G}, {H}	(D,F)	Ja	{(A,B), (D,F)}
{A,B}, {C}, {D,F}, {E}, {G}, {H}	(A,D)	Ja	{(A,B), (D,F), (A,D)}
{A,B,D,F}, {C}, {E}, {G}, {H}	(G,H)	Ja	{(A,B), (D,F), (A,D), (G,H)}
{A,B,D,F}, {C}, {E}, {G,H}	(C,D)	Ja	{(A,B), (D,F), (A,D), (G,H), (C,D)}
{A,B,C,D,F}, {E}, {G,H}	(E,G)	Ja	{(A,B), (D,F), (A,D), (G,H), (C,D), (E,G)}
{A,B,C,D,F}, {E,G,H}	(C,F)	Nein	{(A,B), (D,F), (A,D), (G,H), (C,D), (E,G)}
{A,B,C,D,F}, {E,G,H}	(F,G)	Ja	{(A,B), (D,F), (A,D), (G,H), (C,D), (E,G), (F,G)}
{A,B,C,D,E,F,G,H}

■ Ab jetzt werden keine weiteren Kanten mehr aufgenommen!

43 / 4

Kruskal und Prim im Vergleich

Laufzeit von Kruskal:

- Die Union-Find-Operation ist praktisch in konstanter Zeit möglich, daher hat der zweite Teil des Kruskal-Algorithmus eine nahezu lineare Laufzeit.
- Der Gesamtaufwand wird hauptsächlich durch das Kantensortieren bestimmt und ist somit $O(m \log n)$.

Laufzeit von Prim:

- Wird als Priority Queue ein klassischer Heap verwendet, beträgt der Gesamtaufwand $O(m \log n)$.
- Wird ein Fibonacci-Heap verwendet, reduziert sich die Laufzeit auf $O(m + n \log n)$.

Anwendung in der Praxis:

- Für dichte Graphen ($m = \Theta(n^2)$) ist Prims Algorithmus besser geeignet.
- Für dünne Graphen ($m = \Theta(n)$) ist Kruskals Algorithmus besser geeignet.

5. Divide and Conquer

Divide-and-Conquer: Allgemeines Prinzip für effiziente Problemlösungsstrategien.

Vorgehensweise:

- **Teile:** Zerlege das Problem in mehrere, meist zwei, kleinere Teilprobleme.
- **Herrsche (Conquer):** Löse jeden der Teilprobleme rekursiv.
- **Zusammenführen (Combine):** Fasse die Lösungen der Teilprobleme zu einer Gesamtlösung zusammen.

Zitat:

Divide et impera.

Veni, vidi, vici.

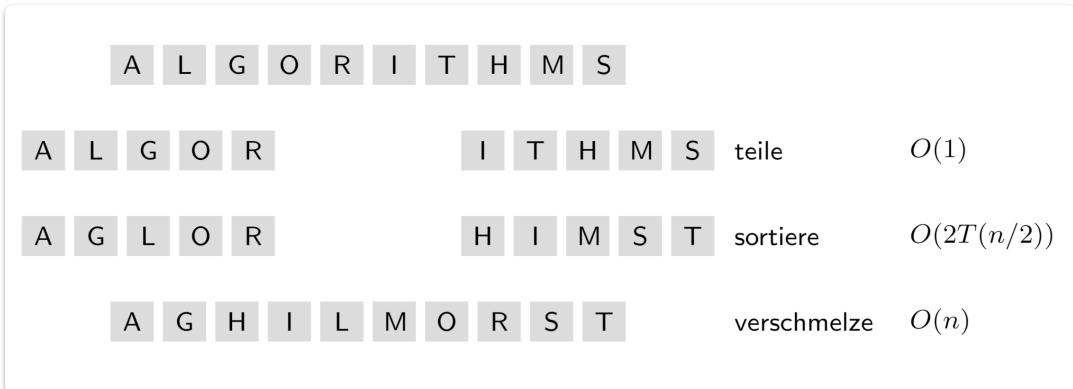
Julius Caesar

Sortieren: Wiederholung

- Primitive Sortierverfahren:
 - Bubblesort
 - Insertionsort
 - Selectionsort
- Laufzeit: Die Laufzeit dieser Verfahren liegt im Worst- und Average-Case immer in $\Theta(n^2)$.
- Frage: Kann man im Worst- und Average-Case schneller sortieren?
- Antwort: Ja. Mergesort ist ein Beispiel dafür.

Mergesort (Sortieren durch Mischen)

- Mergesort:
 - Teile Array in zwei Hälften.
 - Sortiere jede Hälfte rekursiv.
 - Verschmelze zwei Hälften zu einem sortierten Ganzen.

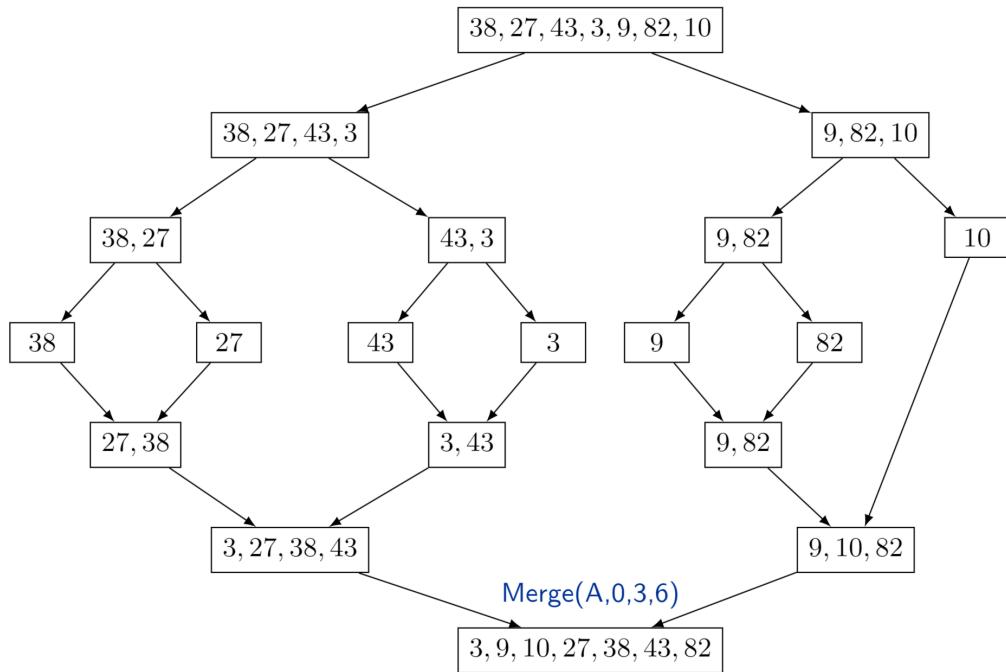


- Pseudocode:
 - Mergesort für ein Array A .
 - Sortiert den Bereich $A[l]$ bis $A[r]$.

```
Mergesort(A, l, r):
  if l < r
    m ← ⌊(l + r)/2⌋
    Mergesort(A, l, m)
    Mergesort(A, m + 1, r)
    Merge(A, l, m, r)
```

- Aufruf: $\text{Mergesort}(A, 0, n - 1)$ für ein Array A mit n Elementen.

Beispiel



Merging (Verschmelzen)

- Vorgehen: Verschmelze zwei sortierte Listen zu einer sortierten Gesamtliste.
- Wie kann man effizient verschmelzen?
 - Benutze temporäres Array.
 - Durchlaufe beide Listen vom Anfang an.
 - Führe die Elemente beider Listen im Reißverschlussverfahren zusammen, übernehme dabei jeweils das kleinste Element der beiden Listen.
 - Hat lineare Laufzeit.



- Pseudocode: Merge auf ein Array A . Verwendet Hilfsarray B .

```

Merge(A,l,m,r):
  i ← l, j ← m + 1, k ← l
  while i ≤ m und j ≤ r
    if A[i] ≤ A[j]
      B[k] ← A[i], i ← i + 1
    else
      B[k] ← A[j], j ← j + 1
      k ← k + 1
    if i > m
      for h ← j bis r
        B[k] ← A[h], k ← k + 1
    else
      for h ← i bis m
        B[k] ← A[h], k ← k + 1
    for h ← l bis r
      A[h] ← B[h]
  
```

Eine nützliche Rekursionsgleichung

- Definition: $C(n)$ = Anzahl der Schlüsselvergleiche (*comparisons*) in Mergesort bei einer Eingabegröße n .
- Mergesort Rekursion:

$$C(n) \leq \begin{cases} 0 & \text{wenn } n = 1 \\ \underbrace{C(\lceil n/2 \rceil)}_{\text{linke Hälfte}} + \underbrace{C(\lfloor n/2 \rfloor)}_{\text{rechte Hälfte}} + \underbrace{n}_{\text{Verschmelzen}} & \text{sonst} \end{cases}$$

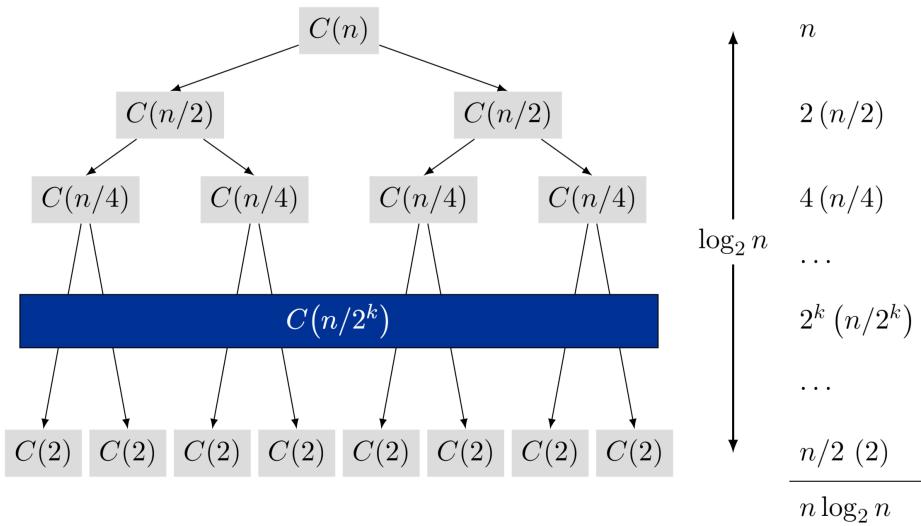
- Lösung: $C(n) = O(n \log_2 n)$.
- Beweis: Wir beschreiben mehrere Wege das $O(n \log_2 n)$ zu beweisen.
- Annahmen:
 - Wir nehmen anfänglich an, dass n eine Zweierpotenz ist.
 - Für ein allgemeines n' mit $\frac{n}{2} < n' < n$ (wobei n eine Zweierpotenz ist) gilt dann:

$$C(n') = O(n \log n)$$

- da $O(\frac{n}{2} \log \frac{n}{2}) = O(n \log n)$ gilt.

Beweis durch Rekursionsbaum

$$C(n) \leq \begin{cases} 0 & \text{wenn } n = 1 \\ \underbrace{2C(n/2)}_{\text{Sortieren beider Hälften}} + \underbrace{n}_{\text{Verschmelzen}} & \text{sonst} \end{cases}$$



Beweis durch Auflösen der Rekurrenz

Behauptung: Wenn $C(n)$ die Rekurrenz erfüllt, dann $C(n) \leq n \log_2 n$.

$$C(n) \leq \begin{cases} 0 & \text{wenn } n = 1 \\ \underbrace{2C(n/2)}_{\text{Sortieren beider Hälften}} + \underbrace{n}_{\text{Verschmelzen}} & \text{sonst} \end{cases}$$

Beweis: Für $n > 1$:

$$\begin{aligned} \frac{C(n)}{n} &\leq \frac{2C(n/2)}{n} + 1 = \frac{C(n/2)}{n/2} + 1 \\ &\leq \frac{2C(n/4)}{n/2} + \frac{n/2}{n/2} + 1 = \frac{C(n/4)}{n/4} + 1 + 1 \\ &\leq \dots \\ &\leq \frac{C(n/n)}{n/n} + \underbrace{1 + \dots + 1}_{\log_2 n} \\ &= \log_2 n \end{aligned}$$

Beweis durch Induktion

Behauptung: Wenn $C(n)$ die Rekursion erfüllt, dann $C(n) \leq n \log_2 n$.

$$C(n) \leq \begin{cases} 0 & \text{wenn } n = 1 \\ \underbrace{2C(n/2)}_{\text{Sortieren beider Hälften}} + \underbrace{n}_{\text{Verschmelzen}} & \text{sonst} \end{cases}$$

Beweis: (durch Induktion auf n)

- Induktionsanfang: $n = 1$.
- Induktionsbehauptung: $C(n) \leq n \log_2 n$.
- Ziel: Zeige, dass $C(2n) \leq 2n \log_2 2n$.

$$\begin{aligned} C(2n) &\leq 2C(n) + 2n \\ &\leq 2n \log_2 n + 2n \\ &= 2n(\log_2 n + 1) \\ &= 2n(\log_2(2n/2) + 1) \\ &= 2n(\log_2 2n - \log_2 2 + 1) \\ &= 2n(\log_2 2n - 1 + 1) \\ &= 2n \log_2 2n \end{aligned}$$

Analyse der Mergesort-Rekursion

Behauptung: Wenn $C(n)$ die folgende Rekursion erfüllt, dann $C(n) \leq n \lceil \log_2 n \rceil$.

$$C(n) \leq \begin{cases} 0 & \text{wenn } n = 1 \\ \underbrace{C(\lceil n/2 \rceil)}_{\text{linke Hälfte}} + \underbrace{C(\lfloor n/2 \rfloor)}_{\text{rechte Hälfte}} + \underbrace{n}_{\text{Verschmelzen}} & \text{sonst} \end{cases}$$

Beweis: (durch Induktion auf n)

- Induktionsanfang: $n = 1$.
- Definiere $n_1 = \lceil n/2 \rceil$, $n_2 = \lfloor n/2 \rfloor$.
- Induktionsschritt: Ist wahr für $1, 2, \dots, n-1$.

$$\begin{aligned} C(n) &\leq C(n_1) + C(n_2) + n \\ &\leq n_1 \lceil \log_2 n_1 \rceil + n_2 \lceil \log_2 n_2 \rceil + n \\ &\leq n_1 \lceil \log_2 n_1 \rceil + n_2 \lceil \log_2 n_1 \rceil + n \\ &= n \lceil \log_2 n_1 \rceil + n \\ &\leq n (\lceil \log_2 n \rceil - 1) + n \\ &= n \lceil \log_2 n \rceil \end{aligned}$$

$$\begin{aligned} n_1 &= \lceil n/2 \rceil \\ &\leq \lceil 2^{\lceil \log_2 n \rceil} / 2 \rceil \\ &= 2^{\lceil \log_2 n \rceil} / 2 \\ &\Rightarrow \log_2 n_1 \leq \lceil \log_2 n \rceil - 1 \end{aligned}$$

- Asymptotische untere Schranke für $C(n)$: Das Verschmelzen für n Elemente benötigt zumindest $C_{best} = \lfloor \frac{n}{2} \rfloor$ Schlüsselvergleiche.
- Daher gilt:

$$C_{best}(n) = C_{worst}(n) = C_{avg}(n) = \Theta(n \log n)$$

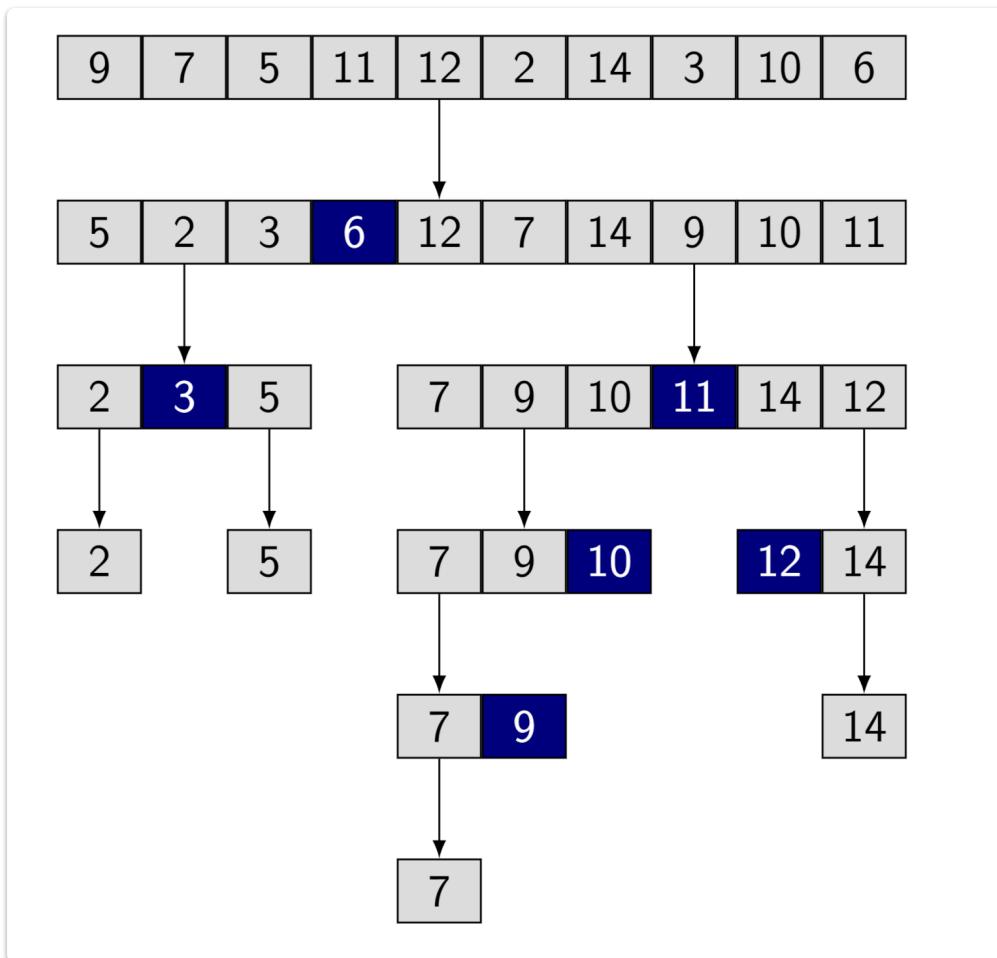
- Allgemein: Die Laufzeit von Mergesort wird durch die Anzahl der notwendigen Vergleiche dominiert.
- Daher gilt:

$$T_{best}(n) = T_{worst}(n) = T_{avg}(n) = \Theta(C(n)) = \Theta(n \log n)$$

Quicksort

- Quicksort: Benutzt auch das Divide-and-Conquer-Prinzip, aber auf eine andere Art und Weise.
- Teile: Wähle „Pivotelement“ x aus Folge A , z.B. das an der letzten Stelle stehende Element. Teile A ohne x so in zwei Teilstücke A_1 und A_2 , dass gilt:
 - A_1 enthält nur Elemente $\leq x$.
 - A_2 enthält nur Elemente $\geq x$.
- Herrsche:
 - Rekursiver Aufruf für A_1 .
 - Rekursiver Aufruf für A_2 .
- Kombiniere: Bilde A durch Hintereinandersetzen von A_1 , x , A_2 .

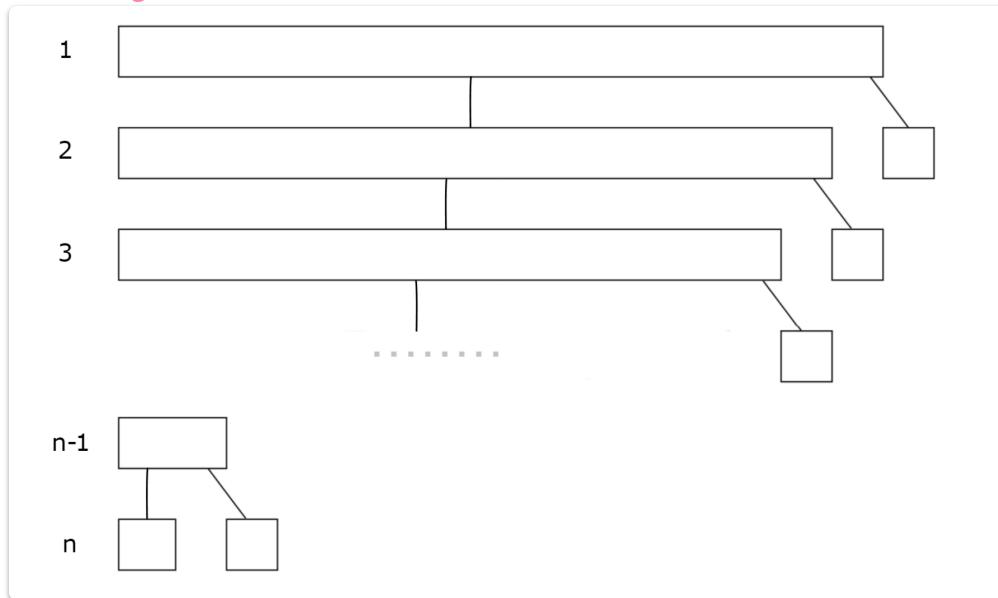
Beispiel



Analyse

- Best-Case:
 - Die beiden Teilstücke haben *immer (ungefähr) die gleiche Länge*.
 - Die Höhe des Aufrufbaumes ist $\Theta(\log n)$.
 - Auf jeder Aufrufebene werden $\Theta(n)$ Vergleiche durchgeführt.
 - Die Anzahl der Vergleiche und die Laufzeit liegen in $\Theta(n \log n)$.

- Worst-Case: Jede (Teil-)Folge wird **immer beim letzten (oder immer beim ersten) Element geteilt.**



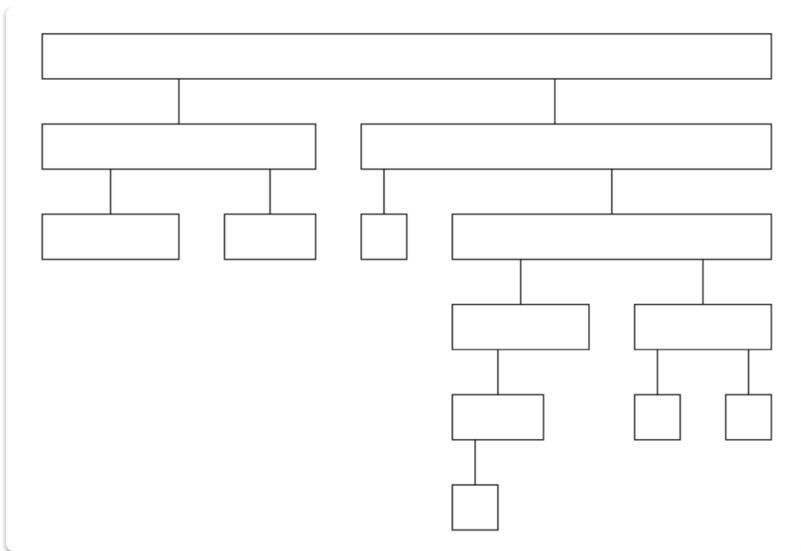
- Die Anzahl der Vergleiche ist $\Theta(n^2)$.

Worst-Case:

- Mögliche Szenario:
 - Aufsteigend sortierte Folge* und es wird **immer das hinterste Element als Pivotelement gewählt.**
 - Alle restlichen Elemente sind kleiner als das Pivotelement und daher wird die Rekursion **nur für die linke Seite** (alle restlichen Elemente außer dem Pivotelement) weitergeführt. Die rechte Seite ist leer (*Terminierung der Rekursion*).
 - Die Größe der Teilfolge in der nächsten Rekursionsstufe verringert sich **immer nur um 1!**
 - $\Theta(n)$ rekursive Aufrufe.
 - Die Laufzeit liegt in $\Theta(n^2)$.
 - Der zum Array zusätzlich benötigte Speicherplatz ist durch die $\Theta(n)$ rekursiven Aufrufe ebenfalls $\Theta(n)$.
- „Vermeiden“ des Worst-Case in der Praxis:
 - Pivotelement **immer zufällig wählen**.
 - Randomisierter Quicksort*.
 - Es ist **nicht mehr** die sortierte Folge das Worst-Case-Szenario.
 - Betrachte **jeweils das erste, letzte und mittlere Element** (an der Position $\lfloor \frac{n}{2} \rfloor$) und nimm den **Median als Pivotelement**.

Average-Case:

- Komplizierter Beweis zeigt, dass die Anzahl der Vergleiche und die Laufzeit dafür auch in $\Theta(n \log n)$ liegen.
- Beispiel für Average-Case: Jede (Teil-)Liste wird nahe der Mitte geteilt.



- Die Anzahl der Vergleiche ist $\Theta(n \log n)$.

Speicherplatzkomplexität

- Speicherplatzkomplexität: Ist ein Maß für das Anwachsen des Speicherbedarfs eines Algorithmus in Abhängigkeit von der Eingabegröße.
- Beispiele für Speicherplatzkomplexität:
 - Quicksort: Worst-Case liegt in $\Theta(n)$, Best-/Average-Case liegt in $\Theta(\log n)$.
 - Mergesort: Best-/Average-/Worst-Case liegt in $\Theta(n)$.
- Praxis: In der Praxis wird daher Quicksort sehr oft Mergesort vorgezogen.

Vergleich von Sortierverfahren

Tabelle: Laufzeit und Vergleiche.

Sortierverfahren	Best-Case	Average-Case	Worst-Case
Insertionsort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Selectionsort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Mergesort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Quicksort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$

Tabelle: Zusätzlicher Speicher.

Sortierverfahren	Best-Case	Average-Case	Worst-Case
Insertionsort	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Selectionsort	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Mergesort	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Quicksort	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(n)$

Einsatz von Sortierverfahren

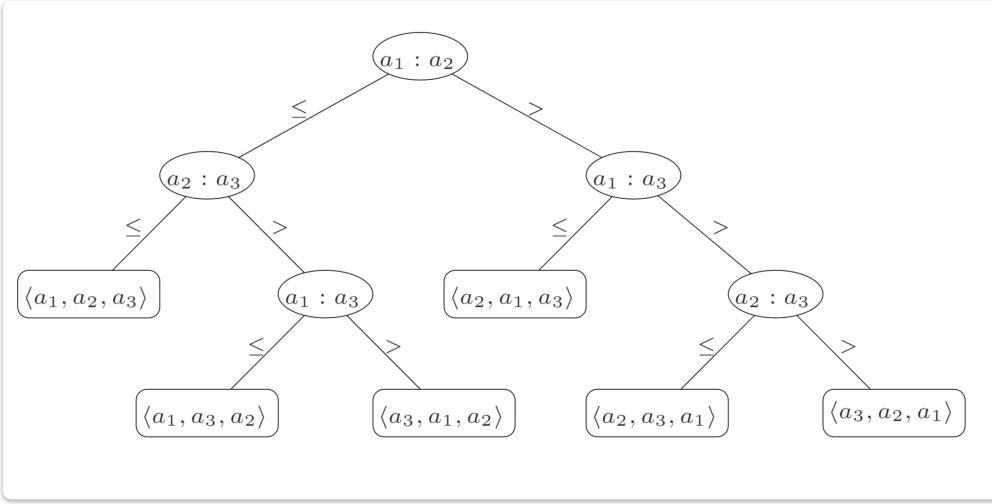
- Quicksort:
 - Wird sehr oft in allgemeinen Sortiersituationen bevorzugt.
- Mergesort:
 - Mergesort wird hauptsächlich für das Sortieren von Listen verwendet.
 - Wird auch für das Sortieren von Dateien auf externen Speichermedien eingesetzt.
 - Dabei wird aber eine iterative Version von Mergesort (*Bottom-up-Mergesort*) verwendet, bei der nur $\log n$ -mal eine Datei sequentiell durchgegangen wird.

Eine untere Schranke

- Ausgangslage: Wir haben verschiedene Sortieralgorithmen kennengelernt. Die Worst-Case Laufzeit liegt im Bereich $O(n \log n)$ bis $O(n^2)$.
- Frage: Geht es besser als in $O(n \log n)$ Zeit, z.B. $O(n)$?
- Antwort: Wir zeigen für das allgemeine Sortierproblem unter der Annahme, dass n verschiedene Schlüssel sortiert werden müssen, dass $O(n \log n)$ optimal ist.

Entscheidungsbaum

- Entscheidungsbaum:
 - Die Knoten entsprechen einem Vergleich von Elementen a_i und a_j .
 - Die Blätter entsprechen den Permutationen.
- Beispiel: Entscheidungsbaum des Insertionsort Algorithmus für (a_1, a_2, a_3) .



- Anzahl der Schlüsselvergleiche: Die Anzahl der Schlüsselvergleiche im Worst-Case C_{worst} entspricht genau der Anzahl der Knoten auf dem längsten Pfad von der Wurzel bis zu einem Blatt minus 1.
- Frage: Wie lautet die untere Schranke für die Höhe eines Entscheidungsbaums?
- Satz: Jeder Entscheidungsbaum für die Sortierung von n paarweise verschiedenen Schlüsseln hat die Höhe $\Omega(n \log_2 n)$.

Beweis

Beweis:

- Betrachte einen Entscheidungsbaum der Höhe h , der n unterschiedliche Schlüssel sortiert.
- Der Baum hat mindestens $n!$ Blätter.
- $n! \leq 2^h$, das impliziert $h \geq \log_2(n!)$.
- Hilfsrechnung:

$$n! \geq n \cdot (n-1) \cdot (n-2) \cdots \lceil \frac{n}{2} \rceil \geq (\lceil \frac{n}{2} \rceil)^{\lceil \frac{n}{2} \rceil} \geq \frac{n}{2}^{\frac{n}{2}}$$
- $h \geq \log_2(n!) \geq \log_2\left(\frac{n}{2}^{\frac{n}{2}}\right) \geq \frac{n}{2} \log_2 \frac{n}{2} = \frac{n}{2}(\log_2 n - 1) = \Omega(n \log_2 n)$

Untere Schranke für allgemeines Sortierproblem

- Satz: Jedes allgemeine Sortierverfahren benötigt zum Sortieren von n paarweise verschiedenen Schlüsseln mindestens $\Omega(n \log n)$ Laufzeit im Worst-Case.
- Folgerung: Mergesort ist ein asymptotisch *zeitoptimaler* Sortieralgorithmus. Es geht (asymptotisch) nicht besser!

Lineare Sortierverfahren

- Beobachtung: Das Ergebnis für die untere Schranke basiert auf der Annahme, dass man keine Eigenschaften der zu sortierenden Elemente ausnutzt, sondern lediglich den Vergleichsoperator.
- Praxis: Tatsächlich sind aber meist Wörter über ein bestimmtes Alphabet (d.h. einer bestimmten Definitionsmenge) gegeben, z.B.:
 - Wörter über $\{a, b, \dots, z, A, B, \dots, Z\}$.
 - Dezimalzahlen.
 - Ganzzahlige Werte aus einem kleinen Bereich.
- Lineare Sortierverfahren: Diese Information über die Art der Werte und ihren Wertebereich kann man zusätzlich ausnutzen und dadurch im Idealfall Verfahren erhalten, die auch im Worst-Case in linearer Zeit sortieren.

Beispiel: Countsort

Eingabe: n Zahlen im Bereich 0 bis z im Array A, Hilfsarray Counts, $z < n$

```

for  $j \leftarrow 0$  bis  $z$ 
  Counts[ $j$ ]  $\leftarrow 0$ 
for  $i \leftarrow 0$  bis  $n - 1$ 
  Counts[A[ $i$ ]]  $\leftarrow$  Counts[A[ $i$ ]] + 1
   $i \leftarrow 0$ 
for  $j \leftarrow 0$  bis  $z$ 
  for  $k \leftarrow 0$  bis Counts[ $j$ ] - 1
    A[ $i$ ]  $\leftarrow j$ 
     $i \leftarrow i + 1$ 
  
```

Komplexität: Erste Schleife in $\Theta(z)$, zweite Schleife in $\Theta(n)$, dritte (mit vierter) Schleife kann nur maximal n Zahlen bearbeiten und ist daher auch in $\Theta(n)$. Damit läuft der Algorithmus in $\Theta(z + n + n) = \Theta(n)$ (da $z = O(n)$).

Beispiele zu Divide and Conquer:

Gute Beispiele für dieses Prinzip findet man in den Slides ab Seite 54.

Da hätte man ein mal [Inversionen zählen](#) und dann noch [Closest Pair of Points](#)

6. Suchbäume

Einleitung

Wörterbuchproblem

Wörterbuch:

- Menge von Elementen eines gegebenen Grundtyps.
- Operationen: Einfügen, Suchen, Entfernen.
- Jedes Element hat einen eindeutigen **Schlüssel** (*key*) und zugehörige Nutzdaten.
- Alle Elemente sind über den Schlüssel identifizierbar.
- Beispiel: Schlüssel sind natürliche Zahlen ($k \in \mathbb{N}$).
- Operationen hängen nur vom Schlüssel ab.
- Vereinfachung: Wörterbuch besteht aus einer Menge ganzzahliger Schlüssel.

Wörterbuchproblem:

- Finde eine geeignete Datenstruktur zusammen mit möglichst effizienten Algorithmen für die Operationen Einfügen, Suchen und Entfernen von Schlüsseln.

Suchverfahren

Suchen:

- Wichtiges Thema der Algorithmik.
- Grundlegende Operation, die oft mit Computern ausgeführt wird:
 - Suchen von Datensätzen in Datenbanken
 - Suchen nach Wörtern in Wörterbüchern
 - Suchen von Stichwörtern im WWW

Array mit n Schlüsseln:

- Lineare Suche in $O(n)$ Zeit.
- Binäre Suche in $O(\log n)$ Zeit (setzt sortiertes Array voraus).
- Einfügen in ein sortiertes Array in $O(n)$ Zeit (Verschieben der Elemente).
- Entfernen aus einem sortierten Array in $O(n)$ Zeit (Verschieben der Elemente).

Frage: Geht es effizienter?

Antwort: Ja, z.B. mit **Suchbäumen**.

Suchbäume

Allgemein:

- Suchbäume existieren in unterschiedlichen Ausprägungen.
- Ermöglichen es, folgende Operationen effizient zu implementieren:
 - Einfügen eines neuen Elements.
 - Suche eines Elements.
 - Entfernen eines Elements.
 - Ausgeben aller gespeicherten Elemente in geordneter Reihenfolge.
 - Finden des kleinsten/größten Elements.
 - Finden eines nächstkleineren/nächstgrößeren Elements.

Wurzelbäume

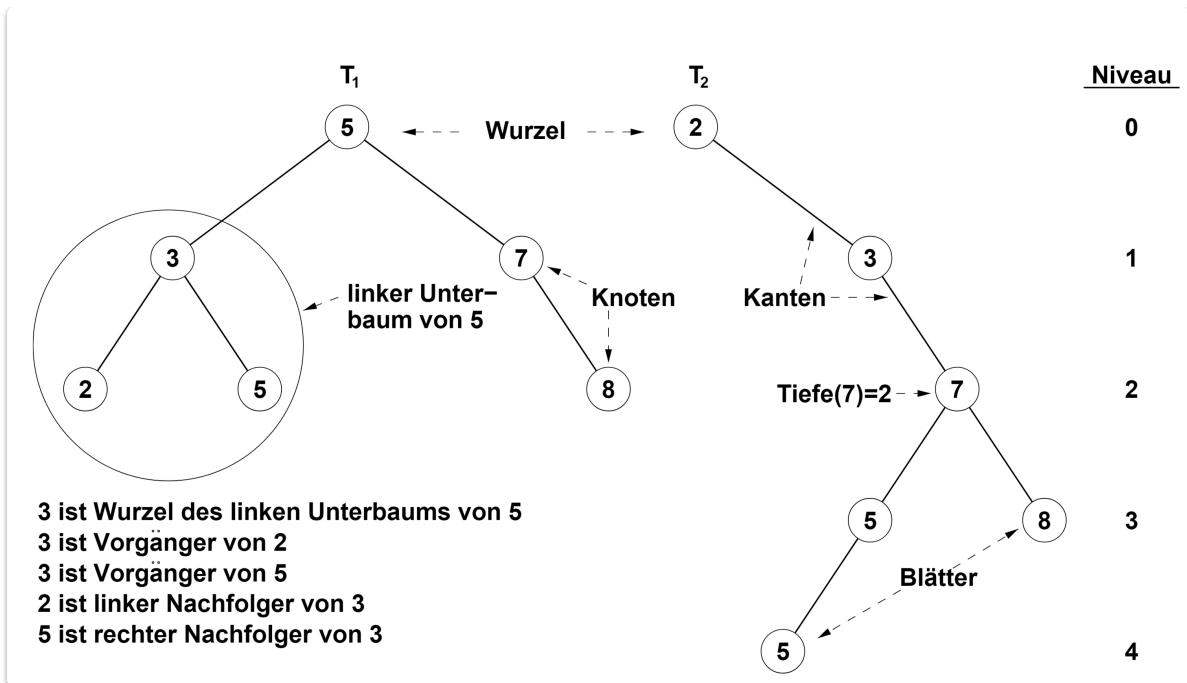
Baum:

- Hier: Spezieller Wurzelbaum (siehe Kapitel Graphen).
- Die Kinder (Nachfolger) eines jeden Knotens sind in einer bestimmten Reihenfolge gegeben.

Wurzel:

- Ist der einzige Knoten ohne Vorgänger.
- Alle anderen Knoten können genau über einen Pfad von der Wurzel erreicht werden.
- Jeder Knoten ist gleichzeitig die Wurzel eines Unterbaumes, der aus ihm selbst und seinen direkten und indirekten Nachfolgern besteht.

Binärer Baum: Jeder Knoten besitzt höchstens zwei Kinder, ein "linkes" und ein "rechtes" Kind.



Tiefe (Niveau) eines Knotens: Tiefe.

- Die Tiefe eines Knotens ist die Anzahl der Kanten auf dem Pfad von der Wurzel zu diesem Knoten.
- Die Tiefe ist eindeutig, da nur ein solcher Pfad existiert.
- Die Tiefe der Wurzel ist 0.
- Die Menge aller Knoten mit gleicher Tiefe im Baum wird auch als Ebene oder Niveau bezeichnet.

Höhe eines (Teil-)Baumes T :

- Die **Höhe** $h(T_r)$ eines (Teil-)Baumes T_r mit dem Wurzelknoten r ist die Länge eines längsten Pfades in dem Teilbaum von r zu einem beliebigen Knoten v in T_r .
- Ein Baum mit nur einem (Wurzel-)Knoten hat die Höhe 0.
- Die Höhe des leeren Baumes definieren wir mit -1.

Blatt:

- Ein Knoten heißt **Blatt**, wenn er keine Kinder besitzt.
- Alle anderen Knoten nennt man **innere Knoten**.

Implementierung von binären Bäumen

Knoten x :

- $x.key$: Schlüssel von x .
- $x.left$: Verweis auf linkes Kind ($null$ wenn nicht vorhanden).
- $x.right$: Verweis auf rechtes Kind ($null$ wenn nicht vorhanden).
- $x.info$: Nutzdaten im Knoten (hier als optional betrachtet).

- $x.parent$: Verweis auf Vorgänger von x (*optional; null* wenn Wurzel).
- Der Zugriff auf den Baum erfolgt über einen Verweis auf den Wurzelknoten, z.B. $root$.
- Ist der Baum leer, so gilt $root = null$.

Durchmusterungen (Traversals): Inorder

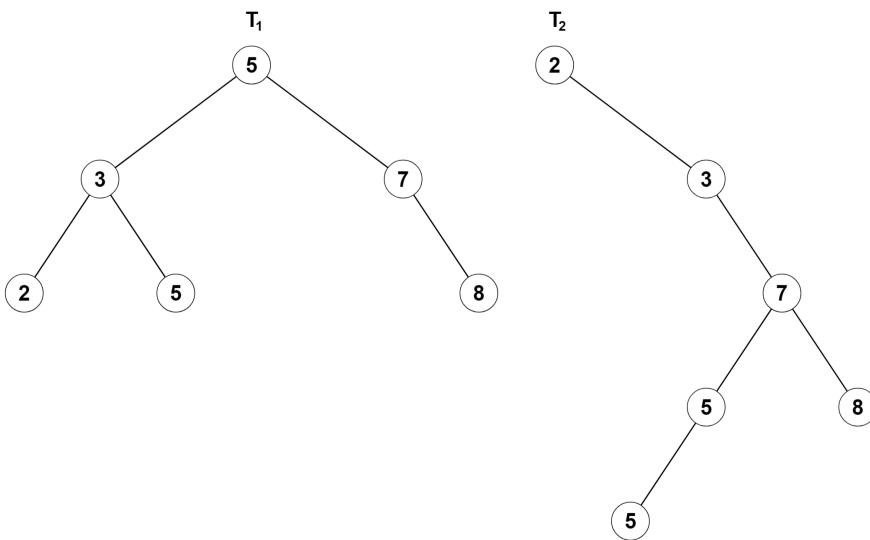
Inorder-Durchmusterung:

- Behandle rekursiv zunächst den linken Unterbaum, dann die Wurzel, dann den rechten Unterbaum.

Aufruf: `Inorder(root)`.

```
Inorder(p)
if p ≠ null
    Inorder(p.left)
    Bearbeite p (z.B. Ausgabe)
    Inorder(p.right)
```

Beispiel



Für beide gezeigten Bäume ist die Inorder-Durchmusterungsreihenfolge **2, 3, 5, 5, 7, 8**.

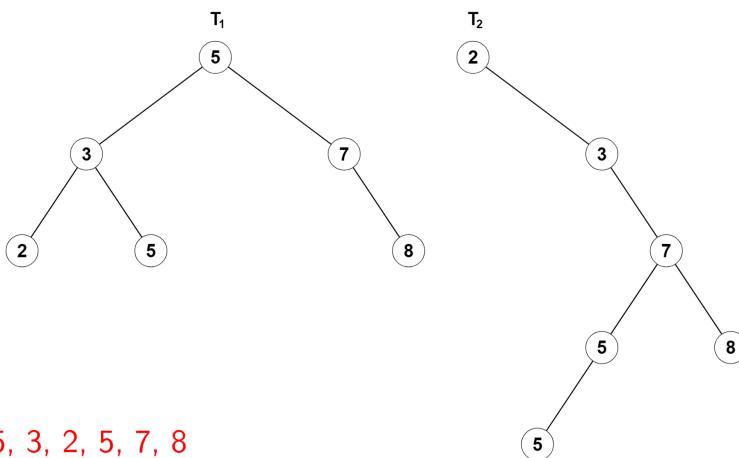
Durchmusterungen: Preorder

Preorder-Durchmusterung:

- Behandle rekursiv zunächst die Wurzel, dann den linken Unterbaum, danach den rechten Unterbaum.

Beispiel:

Beispiel:



- Für T_1 : 5, 3, 2, 5, 7, 8
- Für T_2 : 2, 3, 7, 5, 5, 8

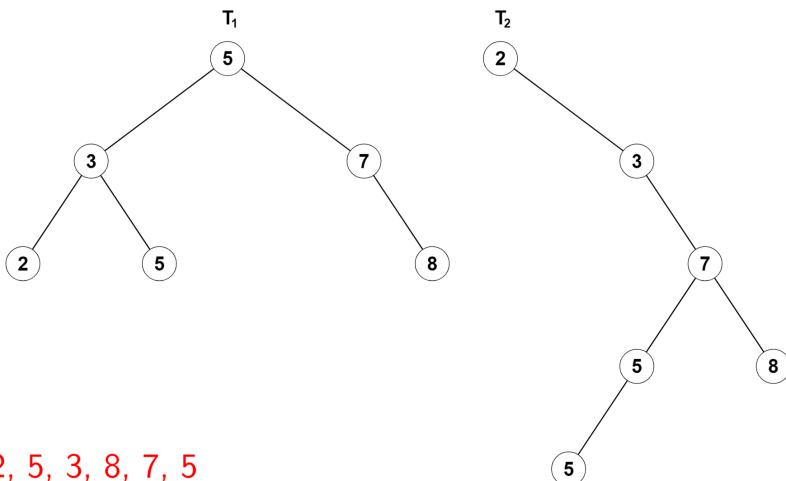
Durchmusterungen: Postorder

Postorder-Durchmusterung:

- Behandle rekursiv zunächst den linken Unterbaum, dann den rechten Unterbaum, danach die Wurzel.

Beispiel:

Beispiel:



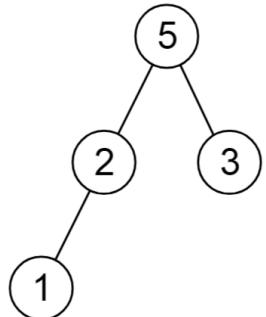
- Für T_1 : 2, 5, 3, 8, 7, 5
- Für T_2 : 5, 5, 8, 7, 3, 2

Durchmusterungen: Theorem

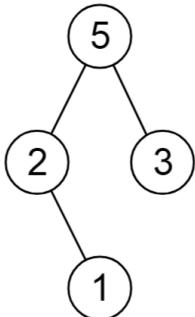
Theorem (ohne Beweis):

- Ein binärer Baum kann immer eindeutig aus der Inorder- und Preorder-Durchmusterungsfolge der Elemente rekonstruiert werden, wenn die Elemente paarweise unterschiedlich sind.
- Gleiches gilt für Inorder zusammen mit Postorder, nicht jedoch für Preorder und Postorder.

Beispiel:



(1)



(2)

- Inorder für (1): 1, 2, 5, 3
- Inorder für (2): 2, 1, 5, 3
- Preorder für beide: 5, 2, 1, 3
- Postorder für beide: 1, 2, 3, 5

Binäre Suchbäume

Ein **binärer Suchbaum** ist ein binärer Baum, der in jedem Knoten x folgende **binäre Suchbaumeigenschaft** erfüllt:

- Ist y ein Knoten des linken Unterbaumes von x , so gilt:

$$y.\text{key} \leq x.\text{key}$$

- Ist z ein Knoten des rechten Unterbaumes von x , so gilt:

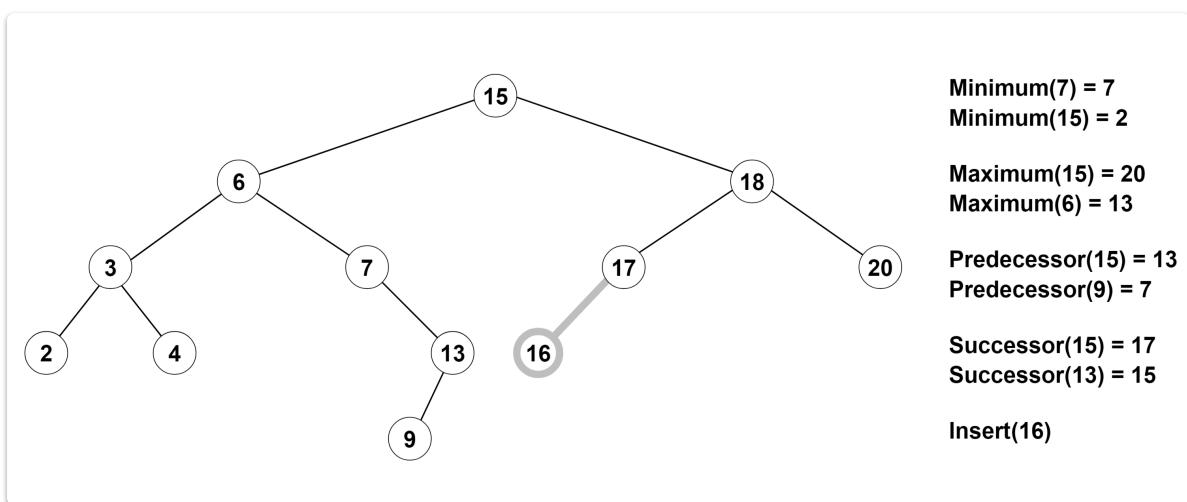
$$z.\text{key} \geq x.\text{key}$$

Hinweis: In einer konkreten Implementierung muss man sich entscheiden, ob man gleiche Schlüssel immer links oder immer rechts speichert.

Operationen auf binären Suchbäumen

Typische Operationen:

- Einfügen / Entfernen eines Elements
- Suchen nach einem Element
- Minimum / Maximum: kleinstes / größtes Element finden
- Predecessor / Successor: voriges / nächstes Element entsprechend der Sortierreihenfolge finden
- Durchmusterung



Suchen

Eingabe: Baum mit Wurzel p und gesuchter Schlüssel s

Rückgabewert: Knoten mit Schlüssel s oder $null$, falls s nicht vorhanden ist.

```
Search( $p, s$ ):
  while  $p \neq null$  und  $p.key \neq s$ 
    if  $s < p.key$ 
       $p \leftarrow p.left$ 
    else
       $p \leftarrow p.right$ 
  return  $p$ 
```

Minimum

Eingabe: Baum mit Wurzel p .

Rückgabewert: Knoten mit dem kleinsten Schlüssel.

```
Minimum( $p$ ):
  if  $p = null$ 
    return  $null$ 
  while  $p.left \neq null$ 
     $p \leftarrow p.left$ 
  return  $p$ 
```

Vorgehen: Solange beim linken Kind weitergehen, bis es keinen linken Nachfolger mehr gibt.

Maximum

Eingabe: Baum mit Wurzel p .

Rückgabewert: Knoten mit dem größten Schlüssel.

```
Maximum( $p$ ):
  if  $p = null$ 
    return  $null$ 
  while  $p.right \neq null$ 
     $p \leftarrow p.right$ 
  return  $p$ 
```

Vorgehen: Solange beim rechten Kind weitergehen, bis es keinen rechten Nachfolger mehr gibt.

Successor

Eingabe: Knoten p in Baum.

Rückgabewert: Nächster Knoten entsprechend der Inorder-Durchmusterungsreihenfolge oder $null$.

```
Successor( $p$ ):
if  $p.right \neq null$ 
    return Minimum( $p.right$ )
else
     $q \leftarrow p.parent$ 
    while  $q \neq null$  und  $p = q.right$ 
         $p \leftarrow q$ 
         $q \leftarrow q.parent$ 
    return  $q$ 
```

Predecessor

Eingabe: Knoten p in Baum.

Rückgabewert: Vorhergehender Knoten entsprechend der Inorder-Durchmusterungsreihenfolge oder $null$.

```
Predecessor( $p$ ):
if  $p.left \neq null$ 
    return Maximum( $p.left$ )
else
     $q \leftarrow p.parent$ 
    while  $q \neq null$  und  $p = q.left$ 
         $p \leftarrow q$ 
         $q \leftarrow q.parent$ 
    return  $q$ 
```

Einfügen

Eingabe: Baum mit Wurzel $root$ und ein neuer Knoten q .

Hinweis: $root$ wird bei leerem Baum verändert (Referenzparameter)

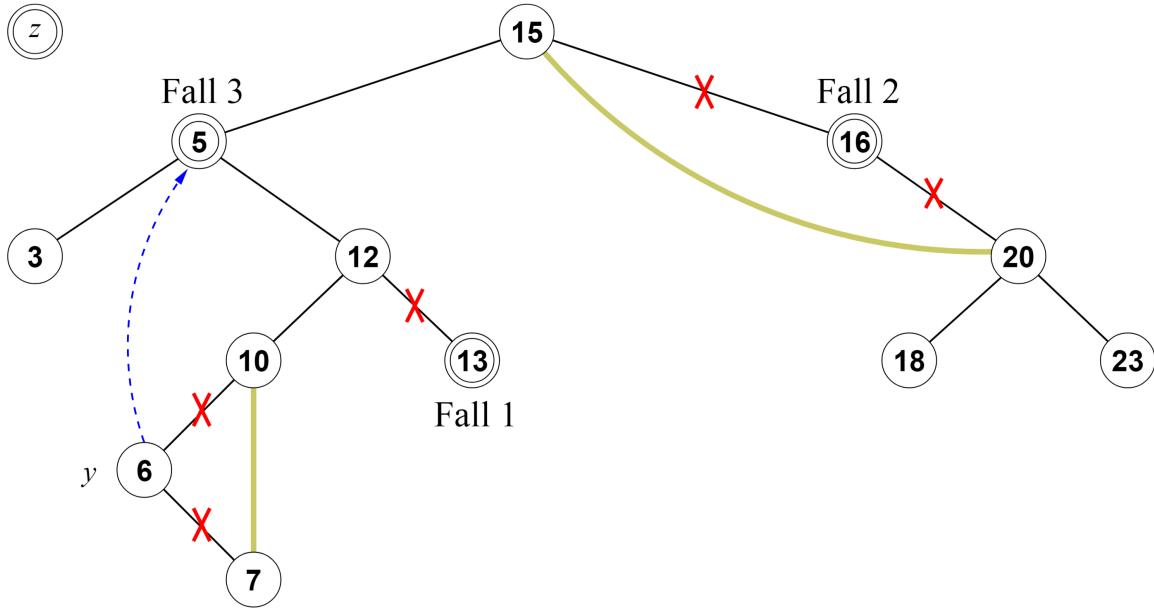
```

Insert( $root$ ,  $q$ ):
 $r \leftarrow null$ ,  $p \leftarrow root$ 
while  $p \neq null$ 
     $r \leftarrow p$ 
    if  $q.key < p.key$ 
         $p \leftarrow p.left$ 
    else
         $p \leftarrow p.right$ 
     $q.parent \leftarrow r$ ,  $q.left \leftarrow null$ ,  $q.right \leftarrow null$ 
    if  $r = null$ 
         $root \leftarrow q$ 
    else
        if  $q.key < r.key$ 
             $r.left \leftarrow q$ 
        else
             $r.right \leftarrow q$ 

```

Entfernen

Entfernen: Knoten z soll entfernt werden. Dabei müssen drei Fälle unterschieden werden:



Fall 1: z hat keine Kinder (z.B. Knoten 13).

- z kann einfach entfernt werden.

Fall 2: z hat ein Kind (z.B. Knoten 16).

- Entspricht dem Entfernen aus einer verketteten linearen Liste.

Fall 3: z hat zwei Kinder (z.B. Knoten 5).

- Wir bringen an die Stelle des zu löschenen Knotens einen Ersatzknoten.
- Löschen des Ersatzknotens an seiner ursprünglichen Position.

Geeigneter Ersatzknoten für z :

- Der Knoten mit dem größten Schlüssel des linken Unterbaumes (Predecessor).
- der Knoten mit dem kleinsten Schlüssel des rechten Unterbaumes (Successor).

Hinweis: Der Ersatzknoten hat in seiner ursprünglichen Position immer maximal einen Nachfolger und wird schließlich gemäß Fall 1 oder 2 entfernt.

Eingabe: Baum mit Wurzel $root$ und ein Knoten q .

Hinweis: $root$ kann verändert werden (Referenzparameter)

```
Remove(root, q):
  if q.left = null oder q.right = null
    r ← q
  else
    r ← Successor(q), q.key ← r.key, q.info ← r.info
    if r.left ≠ null
      p ← r.left
    else
      p ← r.right
    if p ≠ null
      p.parent ← r.parent
    if r.parent = null
      root ← p
    else
      if r = r.parent.left
        r.parent.left ← p
      else
        r.parent.right ← p
```

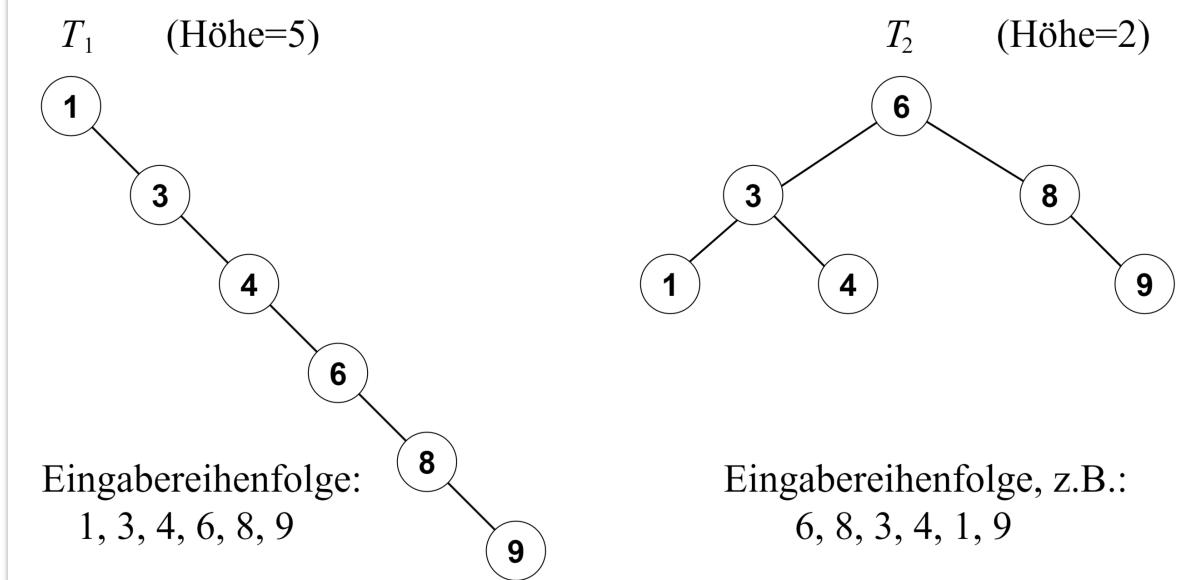
Laufzeiten

Worst-Case:

- Der Zeitaufwand für Suchen, Einfügen, Entfernen, Minimum, Maximum, Predecessor und Successor in einem binären Suchbaum mit der Höhe h liegt in $O(h)$.
- In jeder Operation muss man ungünstigstenfalls einem Pfad von der Wurzel zum tiefsten Blatt folgen.

Strukturen von Suchbäumen

Die Struktur eines binären Suchbaums hängt von der Eingabereihenfolge ab.



Ein binären Suchbaum aus einer sortierten Eingabereihenfolge aufzubauen führt zur **Entartung in eine lineare Liste!**

Balanciert oder Liste

Vollständig balancierter Baum:

- Alle inneren Knoten bis auf jene in der vorletzten Ebene haben 2 Nachfolger.
- Hat die Höhe $h(T) = \lfloor \log_2 n \rfloor$ und daher benötigen alle Operationen bis auf das Durchmustern $O(\log n)$ Zeit.

Zu einer Liste entarteter Baum:

- Ist der Baum jedoch entartet und hat eine Höhe $h(T) = O(n)$, dann benötigen alle Operationen $O(n)$ Zeit!

Average-Case:

- Die erwartete durchschnittliche Suchpfadlänge (über alle möglichen binären Suchbäume für n unterschiedliche Elemente) ist für große n nur ca. 40% länger als im Idealfall, d.h. in $O(\log n)$.
- Für einen ausführlichen Beweis siehe Seite 277ff in:
 - T. Ottmann und P. Widmayer: Algorithmen und Datenstrukturen, 5. Auflage, Spektrum Akademischer Verlag, 2012

Hinreichend balancierter Baum:

Für ein garantiertes logarithmisches Zeitverhalten genügen auch hinreichend balancierte Bäume, die wir im nächsten Abschnitt besprechen werden.

Die bisher betrachteten Suchbäume werden im Gegensatz zu den folgenden Bäumen, bei denen wir die Struktur speziell beeinflussen, auch konkreter **natürliche Suchbäume** genannt.

Balancierte Bäume

Idee: Suchbaum wird durch geeignete Randbedingungen und entsprechende Umordnungsoperationen **effizient ausbalanciert**, um zu garantieren, dass seine Höhe logarithmisch bleibt.

Möglichkeiten:

- **Höhenbalancierte Bäume:** Die Höhe der Unterbäume eines jeden Knotens unterscheidet sich jeweils um höchstens eine Konstante voneinander.
- **Gewichtsbalanceierte Bäume:** Die Anzahl der Knoten in den Unterbäumen jedes Knotens unterscheidet sich höchstens um einen konstanten Faktor.
- **(a, b)-Bäume** ($2 \leq a \leq b$): Jeder Knoten (außer der Wurzel) hat zwischen a und b Kinder und alle Blätter haben den gleichen Abstand zur Wurzel; z.B. $a = 2, b = 4$.

AVL-Bäume

Damit man eben nicht das Problem einer Linearen Liste im Baum hat, gibt es verschiedene Arten von Bäumen die sich selbst balancieren.

Geschichte:

- Erster historischer Vorschlag für selbstbalancierende binäre Suchbäume geht auf AVL-Bäume (Adelson-Velski und Landis, 1962) zurück.

Generelle Idee:

- Durch eine Forderung an die Höhendifferenz der beiden Teilbäume eines jeden Knotens wird ein Degenerieren von Suchbäumen verhindert.

Kritische Operationen:

- Nur Einfügen und Löschen sind kritische Operationen und erfordern eine spezielle Behandlung.
- Alle anderen Operationen (Suchen, ...), die den Baum nicht verändern, funktionieren wie bisher.

Balance

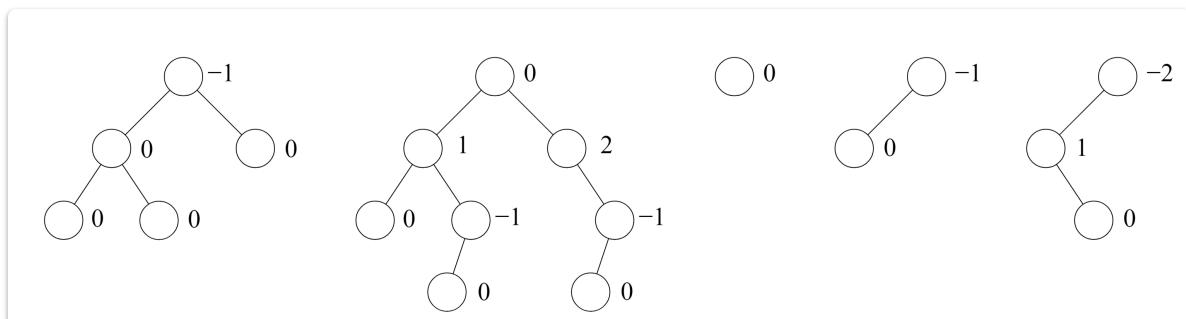
- **Balance eines Knotens v :** $bal(v) = h_2 - h_1$
 - h_1 : Höhe des linken Unterbaumes von v
 - h_2 : Höhe des rechten Unterbaumes von v
- **Balancierter Knoten:** Ein Knoten v heißt **balanciert**, wenn $bal(v) \in \{-1, 0, 1\}$.

AVL-Bedingung

- Ein AVL-Baum ist ein binärer Suchbaum, in dem alle Knoten balanciert sind.
- Für jeden Knoten v gilt: Die Höhe des linken Teilbaums unterscheidet sich von der Höhe des rechten Teilbaums um höchstens 1.

Hinweis: Die Höhe eines leeren Baumes haben wir mit -1 definiert.

Beispiel Balance



AVL-Bäume: Grundlegende Idee

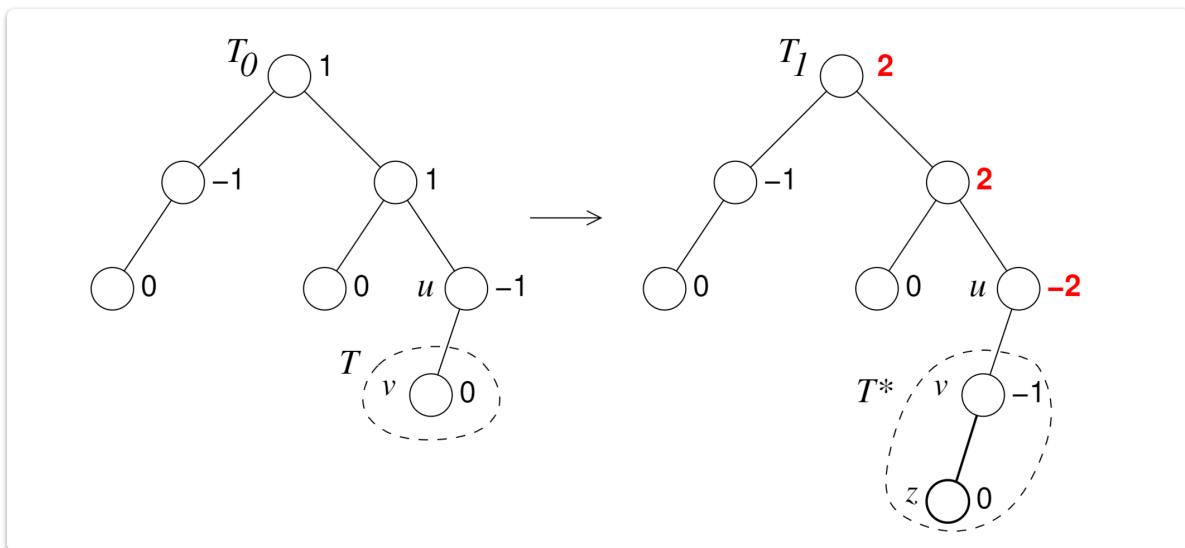
Grundlegende Idee zum Einfügen und Entfernen:

- Führe das Einfügen und Entfernen wie bisher aus.
- Überprüfe danach die **Balance** in möglicherweise betroffenen Knoten und führe gegebenenfalls eine **Rebalancierung** (lokale Umordnung) durch, um die Balance in allen Knoten wieder herzustellen.

AVL-Ersetzung

Definition: Eine AVL-Ersetzung

- ist eine Operation (z.B. Einfügen, Löschen),
- die einen Unterbaum T eines Knotens
- durch einen modifizierten (gültigen) AVL-Baum T^* ersetzt,
- dessen Höhe um höchstens 1 von der Höhe von T abweicht.



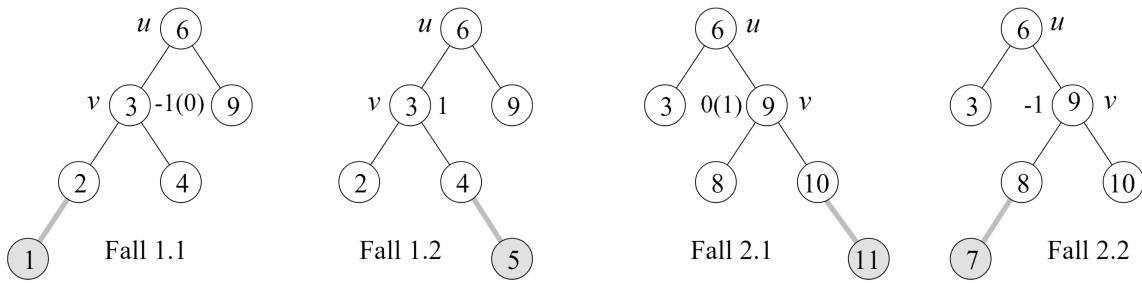
Rebalancierung: Grundlagen

Definitionen:

- Sei T_0 ein gültiger AVL-Baum vor der AVL-Ersetzung und T_1 der unbalancierte Baum hinterher.
- Sei u der unbalancierte Knoten ($\text{bal}(u) \in \{-2, +2\}$) maximaler Tiefe. An diesem wird mit der Rebalancierung gestartet.

Überblick

Rebalancierung: Vier Beispiele für AVL-Bäume, die durch das Einfügen eines Knotens aus der Balance geraten sind.

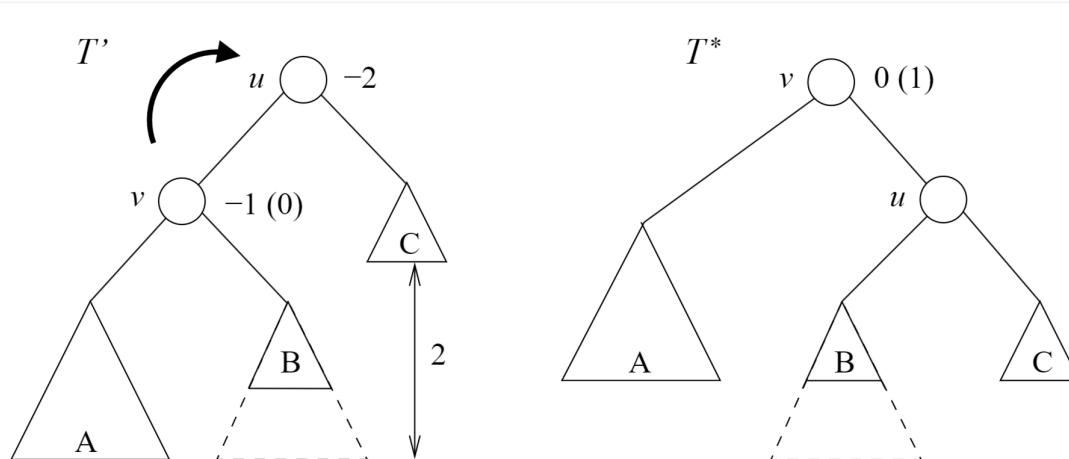


- Wenn $\text{bal}(u) = -2$ und v ist das linke Kind von u und
 - $\text{bal}(v) \in \{-1, 0\} \rightarrow \text{Fall 1.1}$
 - $\text{bal}(v) = 1 \rightarrow \text{Fall 1.2}$
- Wenn $\text{bal}(u) = 2$ und v ist das rechte Kind von u und
 - $\text{bal}(v) \in \{0, 1\} \rightarrow \text{Fall 2.1}$
 - $\text{bal}(v) = -1 \rightarrow \text{Fall 2.2}$

Fall 1.1

Fall 1.1: Sei $\text{bal}(u) = -2$. Sei v das linke Kind von u und $\text{bal}(v) \in \{-1, 0\}$.

- Der linke Unterbaum des linken Kindes von u ist höher als oder gleich hoch wie der rechte.
- Rebalancierung von u durch **einfache Rotation nach rechts** an u , d.h. u wird rechtes Kind von v .
- Das rechte Kind von v wird als linkes Kind an u abgegeben.

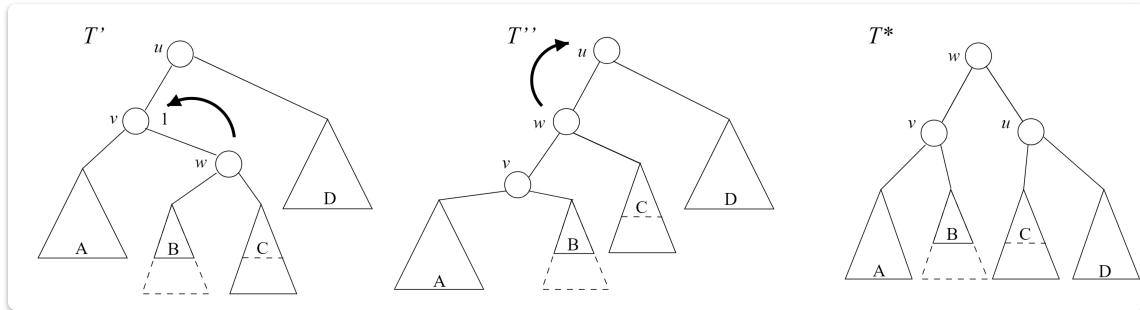


Fall 1.2

Fall 1.2: Sei $\text{bal}(u) = -2$. Sei v das linke Kind von u und $\text{bal}(v) = 1$.

- Der rechte Unterbaum des linken Kindes von u ist höher als der linke.
- Dann existiert das rechte Kind w von v .
- Rebalancierung durch eine **Rotation nach links** an v und eine anschließende **Rotation nach rechts** an u (**Doppelrotation links-rechts**).

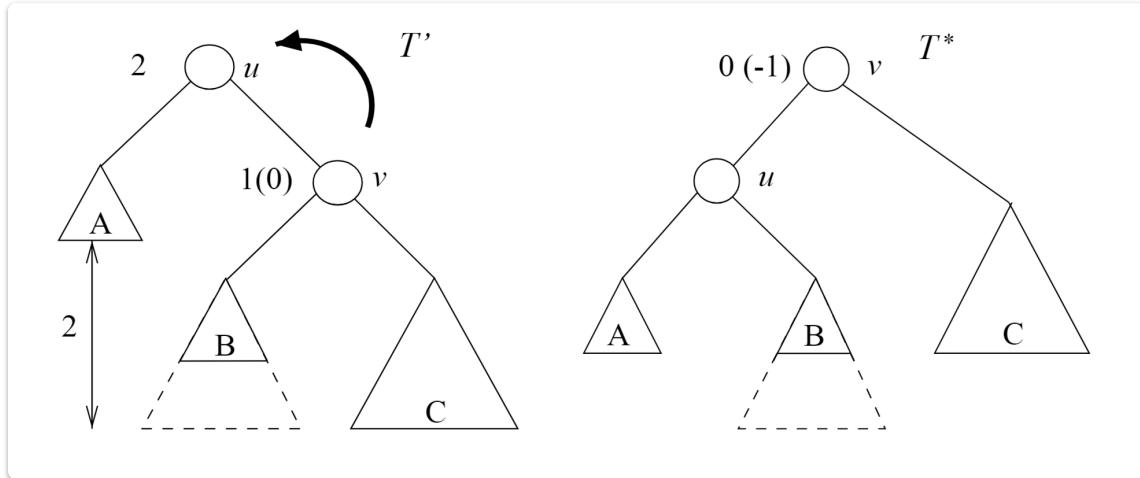
- Also man erreicht mit der ersten Rotation wieder Fall 1.1 und dann kann man von da wie gewohnt weitermachen.



Fall 2.1

Fall 2.1: Sei $\text{bal}(u) = 2$. Sei v das rechte Kind von u und $\text{bal}(v) \in \{0, 1\}$.

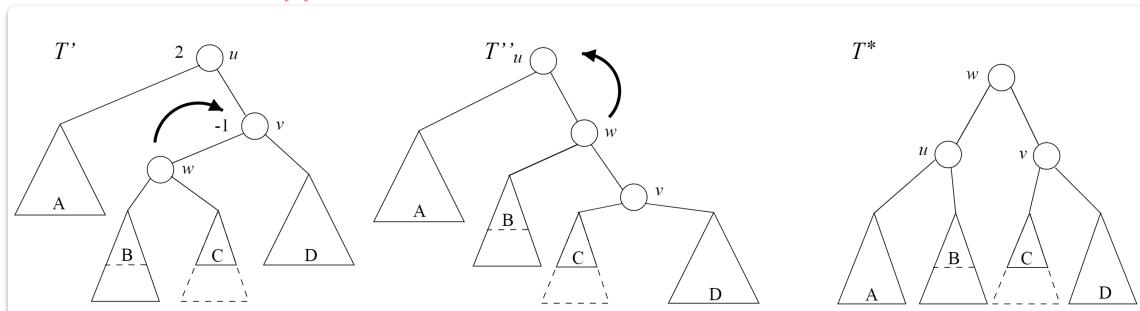
- Der rechte Unterbaum des rechten Kindes von u ist höher als oder gleich hoch wie der linke.
- Rebalancierung von u durch **einfache Rotation nach links** an u , d.h. u wird linkes Kind von v .
- Das linke Kind von v wird als rechtes Kind an u abgegeben.



Fall 2.2

Fall 2.2: Sei $\text{bal}(u) = 2$. Sei v das rechte Kind von u und $\text{bal}(v) = -1$.

- Der linke Unterbaum des rechten Kindes von u ist höher als der rechte.
- Dann existiert das linke Kind w von v .
- Rebalancierung durch eine **Rotation nach rechts** an v und eine anschließende **Rotation nach links an u (Doppelrotation rechts-links)**.



Höhe

Implementierung: Wir ergänzen alle Knoten u um ein Attribut $u.height$, das jeweils die **Höhe** des (Unter-)Baumes mit der Wurzel u angibt.

Hilfsfunktion: Wir definieren folgende Hilfsfunktion zur Ermittlung der Höhe eines Baumes:

- **Eingabe:** Teilbaum mit Wurzel u .
- **Rückgabewert:** Höhe des Teilbaums.

```
Height( $u$ ):  

if  $u = null$   

    return  $-1$   

else  

    return  $u.height$ 
```

Einfache Rotation nach rechts

Eingabe: Teilbaum mit Wurzel u .

Rückgabewert: Neue Wurzel v des rebalancierten Teilbaums.

```
RotateToRight( $u$ ):  

 $v \leftarrow u.left$   

 $u.left \leftarrow v.right$   

 $v.right \leftarrow u$   

 $u.height \leftarrow \max(\text{Height}(u.left), \text{Height}(u.right)) + 1$   

 $v.height \leftarrow \max(\text{Height}(v.left), \text{Height}(u)) + 1;$   

return  $v$ 
```

Doppelrotation links-rechts

Eingabe: Teilbaum mit Wurzel u .

Rückgabewert: Neue Wurzel des rebalancierten Teilbaums.

```
DoubleRotateLeftRight( $u$ ):  

 $u.left \leftarrow \text{RotateToLeft}(u.left)$   

return  $\text{RotateToRight}(u)$ 
```

Einfügen in AVL-Baum

Eingabe: Wurzel p (wird mögl. geändert), neuer Knoten q .

Rückgabewert: Höhe des Teilbaums.

```
Insert( $p, q$ ):  

if  $p = null$   

     $p \leftarrow q, q.left \leftarrow q.right \leftarrow null, q.height \leftarrow 0$   

else  

    if  $q.key < p.key$   

        Insert( $p.left, q$ )  

        if Height( $p.right$ ) – Height( $p.left$ ) = -2  

            if Height( $p.left.left$ ) ≥ Height( $p.left.right$ )  

                 $p \leftarrow \text{RotateToRight}(p)$   

            else  

                 $p \leftarrow \text{DoubleRotateLeftRight}(p)$   

    elseif  $q.key > p.key$   

        ....  

    else  

        Knoten  $q$  schon vorhanden  

 $p.height \leftarrow \max(\text{Height}(p.left), \text{Height}(p.right)) + 1$ 
```

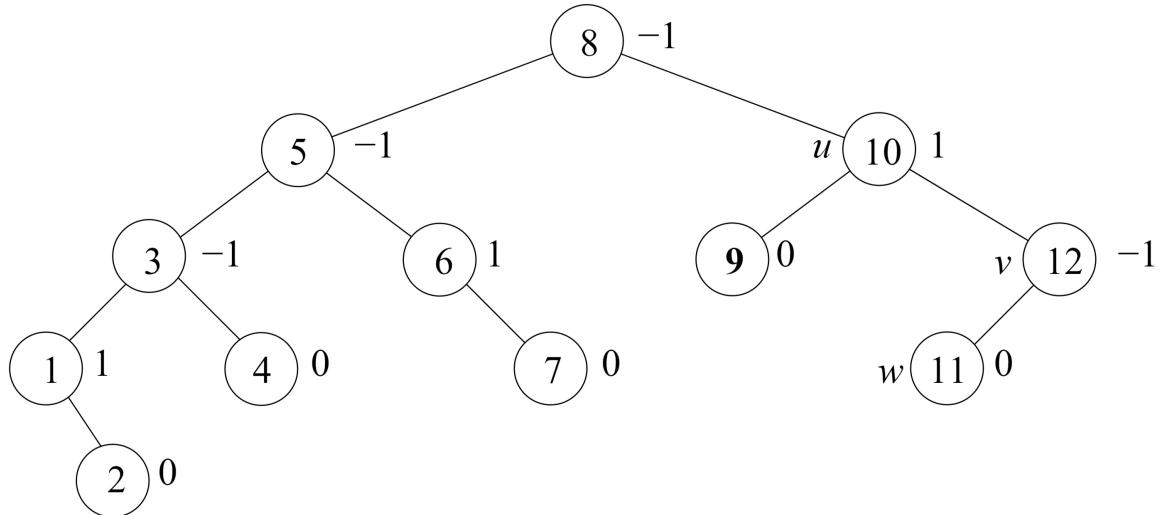
```

Insert(p, q):
if p = null
    p  $\leftarrow$  q, q.left  $\leftarrow$  q.right  $\leftarrow$  null, q.height  $\leftarrow$  0
else
    if q.key < p.key
        ...
    elseif q.key > p.key
        Insert(p.right, q)
        if Height(p.right) – Height(p.left) = 2
            if Height(p.right.right)  $\geq$  Height(p.right.left)
                p  $\leftarrow$  RotateToLeft(p)
            else
                p  $\leftarrow$  DoubleRotateRightLeft(p)
        else
            Knoten q schon vorhanden
            p.height  $\leftarrow$  max(Height(p.left), Height(p.right)) + 1

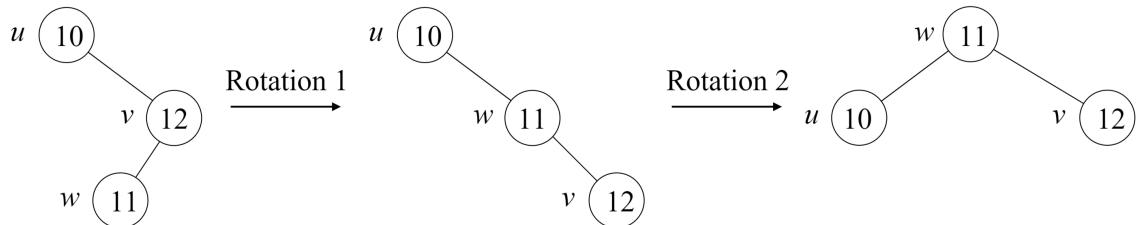
```

Beispiel für Entfernen aus AVL-Baum

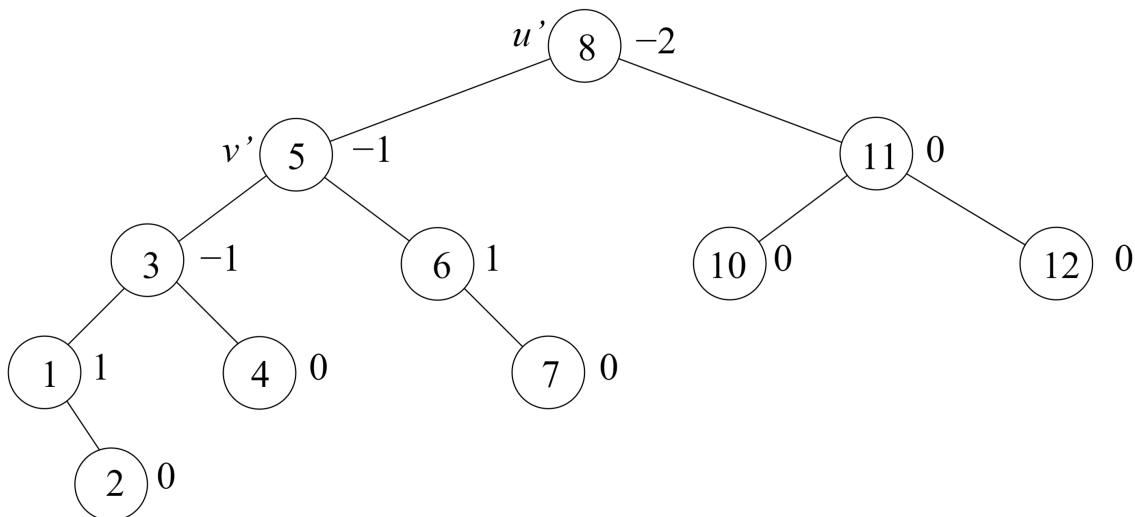
Ausgangssituation: 9 soll entfernt werden.



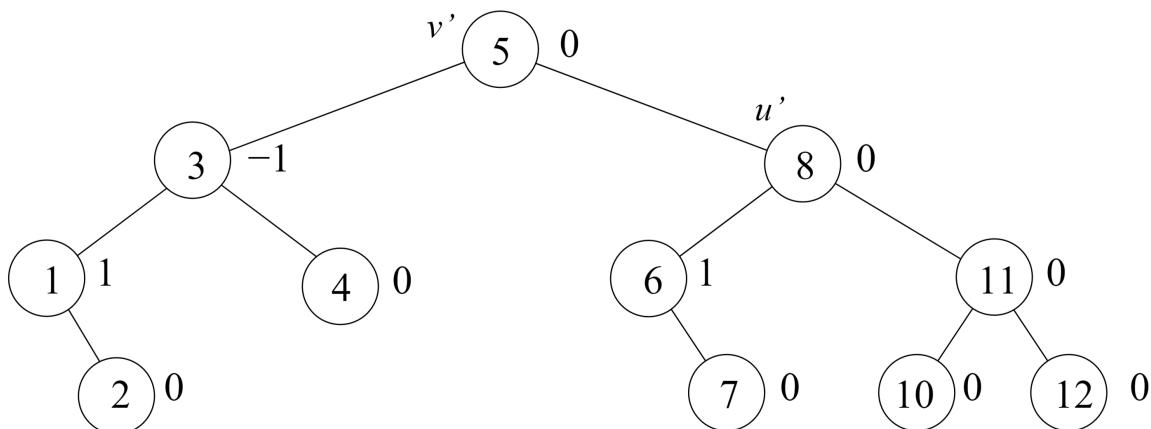
Entfernen: Erfordert Doppelrotation rechts-links.



Ergebnis: 9 wurde entfernt, Unterbaum rotiert, Höhe der Wurzel hat sich verändert.



Abschluss: Einfache Rotation nach rechts über die Wurzel.



Analyse von AVL-Bäumen

Theorem: Die Höhenbedingung eines AVL-Baumes stellt sicher, dass die Höhe eines AVL-Baums mit $n \geq 1$ Knoten durch $O(\log n)$ beschränkt ist.

Beweis: Einen ausführlichen Beweis findet man ab Seite 284ff in:

T. Ottmann und P. Widmayer: *Algorithmen und Datenstrukturen*, 5. Auflage, Spektrum Akademischer Verlag, 2012.

Rotieren: Der Zeitaufwand zum Ausführen einer einzelnen Rotation oder Doppelrotation ist konstant.

Worst-Case:

- Beim Einfügen gibt es maximal eine (Doppel-)Rotation.
- Beim Entfernen werden im Worst-Case auf dem Suchpfad von der betroffenen Stelle weg bis zur Wurzel Rotationen bzw. Doppelrotationen durchgeführt.

Aufwand: Die Laufzeit der Operationen Einfügen und Entfernen liegt daher immer in $O(\log n)$

B-Bäume

Motivation

Bisherige Suchbäume: Gut geeignet, wenn sich die Daten im Hauptspeicher befinden (internes Suchen).

Große Datenmengen:

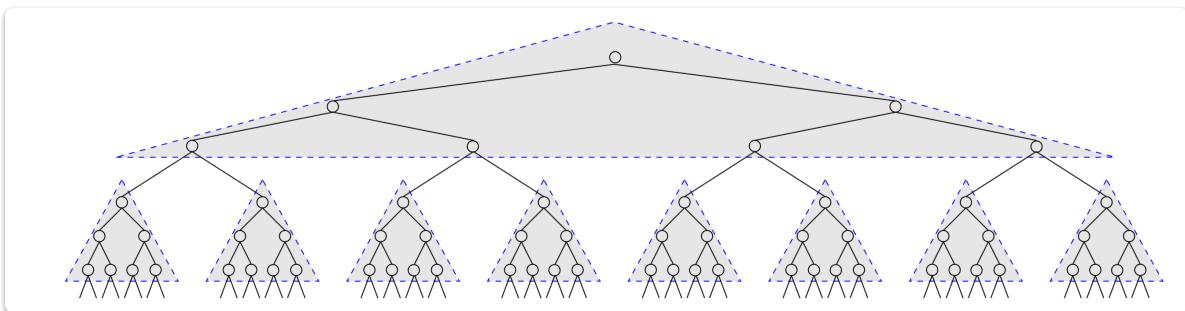
- Bei sehr großen Datenmengen (z.B. Datenbanken) werden die Daten auf Festplatten oder anderen externen Speichern abgelegt.
- Bei Bedarf werden Teile davon in den Hauptspeicher geladen.
- Ein Zugriff auf den externen Speicher benötigt deutlich mehr Zeit als ein Zugriff auf den Hauptspeicher.
- Zugriffe sind **blockorientiert**, d.h. es werden immer ganze Blöcke von z.B. 4KB gelesen/geschrieben.

Beispiel: Binärer Baum mit N Datensätzen, extern gespeichert.

- Verweise auf die linken und rechten Unterbäume sind jeweils Adressen auf einem externen Speichermedium.
- Suche nach einem Schlüssel benötigt daher ungefähr $\log_2 N$ externe Zugriffe.
- Bei $N = 10^9$ sind das ca. 30 externe Zugriffe.

Annahme: Externe Speichermedien weisen i.A. eine blockorientierte Struktur auf. Bei einem externen Speicherzugriff wird immer eine gesamte Speicherseite (page, block) gelesen oder geschrieben.

Grundsätzliche Idee: „Zusammenfassen mehrerer Knoten“, sodass jeweils eine Speicherseite bestmöglich genutzt wird.



Vorteil: Weniger Zugriffe auf Speicherseiten notwendig.

Beispiel:

- Baum mit $N = 10^9$ Schlüssel, 128 Schlüssel pro Speicherseite.
- Man benötigt nur 3 externe Speicherzugriffe.

Definition

B-Bäume: Sie sind eine Verallgemeinerung von binären Suchbäumen und setzen die Idee der Gruppierung in Speicherseiten in die Praxis um.

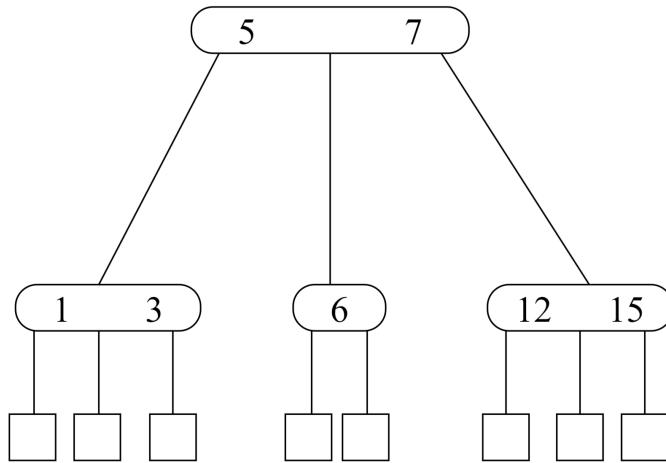
Definition: B-Baum der Ordnung m (Bayer und McCreight, 1972)

1. Alle Blätter haben gleiche Tiefe und sind leere Knoten.
2. Jeder Knoten hat bis zu m Kinder.
3. Jeder innere Knoten außer der Wurzel hat mindestens $\lceil m/2 \rceil$ Kinder. Die Wurzel hat mindestens 2 Kinder.
4. Jeder Knoten mit k Schlüsseln hat $k + 1$ Kinder.
5. Für jeden Knoten mit Schlüsseln s_1, s_2, \dots, s_k und Kindern c_1, c_2, \dots, c_{k+1} gilt:
 - Alle Schlüssel in T_1 sind kleiner gleich s_1 , und
 - alle Schlüssel in T_i (für $1 < i \leq k$) bezeichnen den Teilbaum mit Wurzel c_i , und es gilt $s_{i-1} < (\text{alle Schlüssel in } T_i) \leq s_i$, und
 - alle Schlüssel in T_{k+1} sind größer als s_k .

(T_i bezeichnet den Teilbaum mit Wurzel c_i .)

Beispiel

Beispiel: B-Baum der Ordnung $m = 3$ (2-3 Baum) mit 7 Schlüsseln und 8 Blättern.



Implementierung eines B-Baum-Knotens

Implementierung:

- $p.n$: Anzahl der Schlüssel
- $p.key[1], \dots, p.key[l]$: Schlüssel s_1, \dots, s_l
- $p.info[1], \dots, p.info[l]$: Datenfelder zu Schlüsseln s_1, \dots, s_l
- $p.child[0], \dots, p.child[l]$: Verweis auf Kinderknoten v_0, \dots, v_l

Blätter: Diese markieren wir hier durch $p.n = 0$.

Anmerkung: In einer realen Implementierung brauchen die leeren Blätter nicht explizit gespeichert zu werden. Uns erleichtern sie hier aber die Definitionen und Überlegungen.

Suche

Eingabe: B-Baum mit Wurzel p und ein Schlüssel x .

Rückgabewert: Knoten mit Schlüssel x oder $null$, falls x nicht vorhanden ist.

```
Search( $p, x$ ):
 $i \leftarrow 1$ 
while  $i \leq p.l$  und  $x > p.key[i]$ 
     $i \leftarrow i + 1$ 
if  $i \leq p.l$  und  $x = p.key[i]$ 
    return ( $p, i$ )
if  $p.l = 0$ 
    return  $null$ 
else
    return Search( $p.child[i - 1], x$ )
```

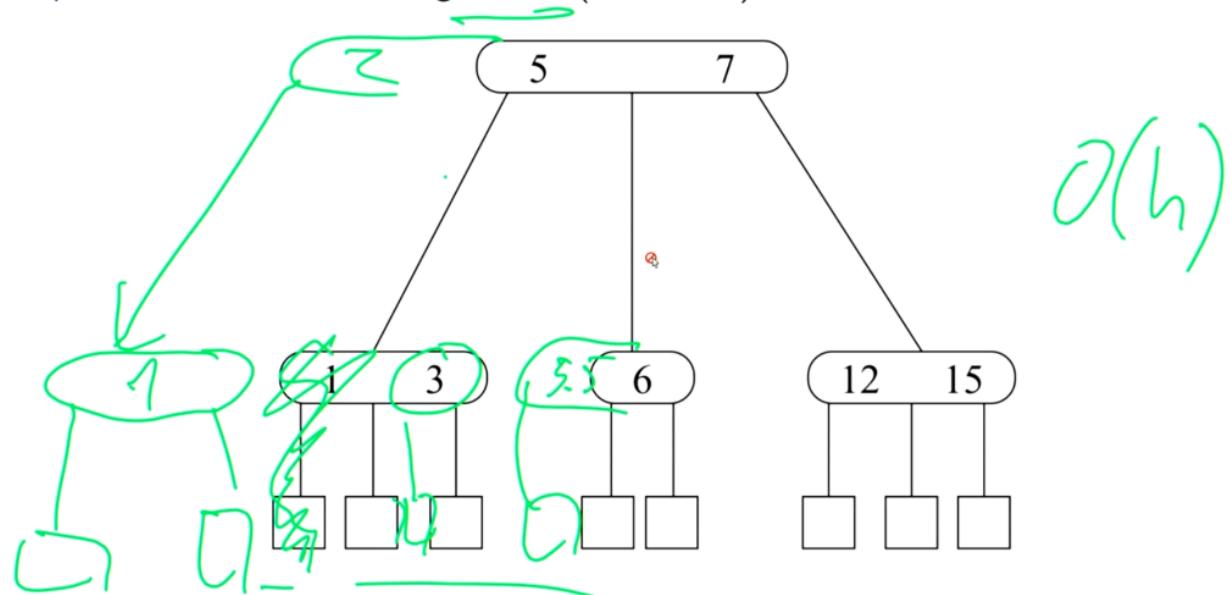
Einfügen

Einfügen im B-Baum:

- Schlüssel suchen → endet in Blatt p_0
- Sei p Vorgänger von p_0 und $p.child[i]$ zeigt auf p_0
- Schlüssel zwischen s_i und s_{i+1} in p einfügen, neues Blatt erzeugen
- Wenn $p.n = m$ fertig sonst: p splitten
 - p' : neuer Knoten
 - $s_{\lceil m/2 \rceil}$: mittlerer Schlüssel
 - Schlüssel $s_1, \dots, s_{\lceil m/2 \rceil - 1}$ in p
 - Schlüssel $s_{\lceil m/2 \rceil + 1}, \dots, s_m$ in p'
 - mittleren Schlüssel in Vorgänger einfügen

Anmerkung: B-Bäume wachsen in ihrer Höhe immer nur durch Splitten der Wurzel!

Beispiel: B-Baum der Ordnung $m = 3$ (2-3 Baum) mit 7 Schlüsseln und 8 Blättern.



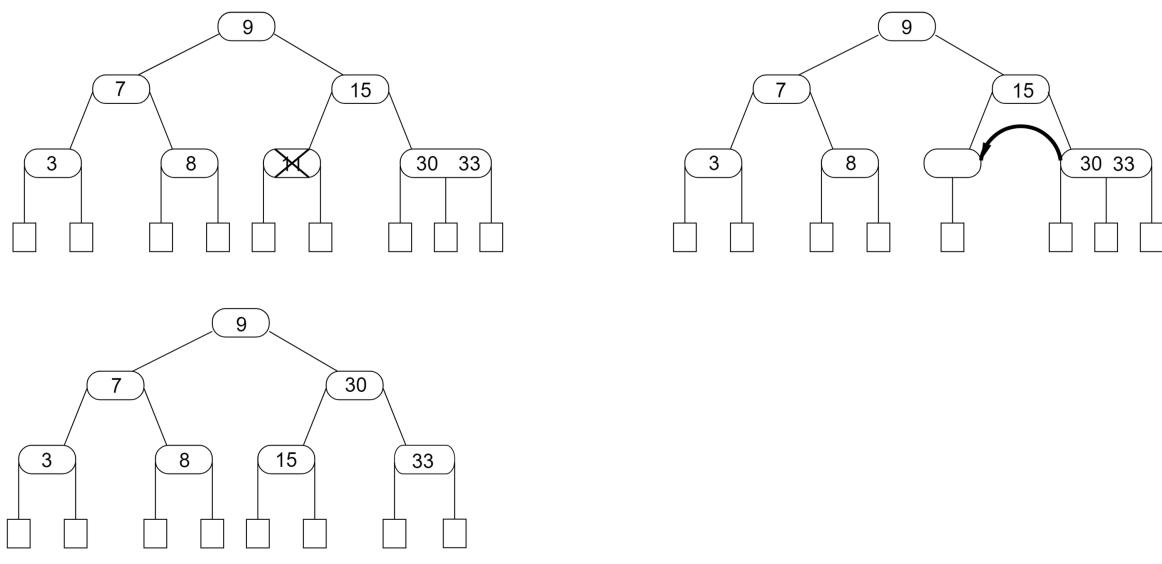
Entfernen

Entfernen aus einem B-Baum:

- Schlüssel suchen
- Falls Schlüssel nicht in unterster Ebene:
 - mit **Ersatzschlüssel** aus unterster Ebene tauschen
 - Ersatzschlüssel:
 - kleinster im rechten Unterbaum oder größter im linken,
 - vgl. Entfernen im binären Suchbaum mit zwei Nachfolgern
- Datensatz mit Blattknoten entfernen
- Falls nun der Knoten zu wenige Schlüssel hat:
 - Übernehme Schlüssel vom linken oder rechten Geschwisterknoten, wenn dieser $> \lceil m/2 \rceil$ Nachfolger hat
 - oder verschmelze mit linkem oder rechtem Geschwisterknoten mit $\lceil m/2 \rceil$ Nachfolgern
 - Dabei Trennelement im Elternknoten mitberücksichtigen!

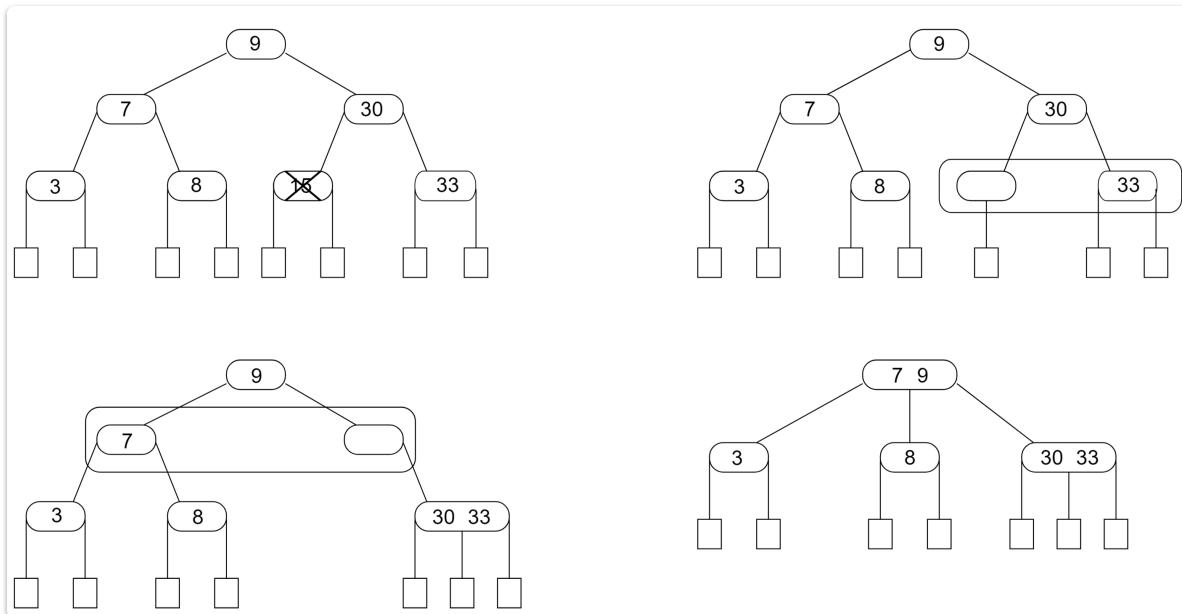
Beispiel 1

11 aus B-Baum (Ordnung 3) entfernen



Beispiel 2

15 aus B-Baum (Ordnung 3) entfernen



Eigenschaften von B-Bäumen

Lemma: Die Anzahl der Blätter in einem B-Baum T ist immer um 1 größer als die Anzahl der Schlüssel.

Beweis: Mit Induktion über die Höhe h

- **Basisfall:** $h = 1$: Wurzel mit $k - 1$ Schlüssel, $2 \leq k \leq m$
 - Lemma gilt für Höhe 1.
- **Induktionsannahme:** Lemma gilt für Höhe h .
- **Induktionsschritt:** Wir zeigen, dass es auch für Höhe $h + 1$ gilt:
 - In einem B-Baum mit Höhe $h + 1$ gibt es k Unterbäume T_1, \dots, T_k mit Höhe h .
 - Diese Unterbäume haben $(n_1 - 1), \dots, (n_k - 1)$ Schlüssel.

- → Insgesamt gibt es $\sum_{i=1}^k n_i$ Blätter und $\sum_{i=1}^k (n_i - 1) = (\sum_{i=1}^k n_i) - k = (\text{Anzahl Blätter}) - k$ Schlüssel.
- Da die Wurzel $k - 1$ Schlüssel hat, ist die Gesamtzahl der Schlüssel $(\sum_{i=1}^k n_i) - k + (k - 1) = (\text{Anzahl Blätter}) - 1$.

Wieviele Blätter kann es in einem B-Baum der Höhe h geben?

- Die minimale Anzahl an Blättern N_{min} wird erreicht, wenn die Wurzel nur 2 und jeder andere innere Knoten nur $\lceil m/2 \rceil$ Nachfolger hat, d.h. $N_{min} = 2 \cdot \lceil m/2 \rceil^{h-1}$.
- Die maximale Anzahl an Blättern N_{max} wird erreicht, wenn jeder innere Knoten die maximal mögliche Anzahl m von Nachfolger hat, d.h. $N_{max} = m^h$.

Korollar: Für die Höhe h eines B-Baumes mit N Schlüsseln und $(N + 1)$ Blättern muss gelten:

$$N_{min} = 2 \lceil m/2 \rceil^{h-1} \leq N + 1 \leq m^h = N_{max}$$

und somit auch

$$\log_m(N + 1) \leq h \leq 1 + \log_{\lceil m/2 \rceil} \left(\frac{N + 1}{2} \right)$$

D.h., wir haben gezeigt, dass B-Bäume immer eine Höhe $h = \Theta(\log N)$ haben, und somit sind die Operationen Suchen, Einfügen und Entfernen auch effizient in $\Theta(\log N)$ Zeit durchführbar.

B*-Bäume

Variante von B-Bäumen

Definition

- **Komplette Datensätze nur in Blättern:**
 - k^* bis $2k^*$ Datensätze, keine Verweise
 - (k^* ist ein von m unabhängiger Parameter)
- In Zwischenknoten nur Schlüssel und Verweise
 - Schlüssel: immer der des größten Elements im vorangehenden Unterbaum

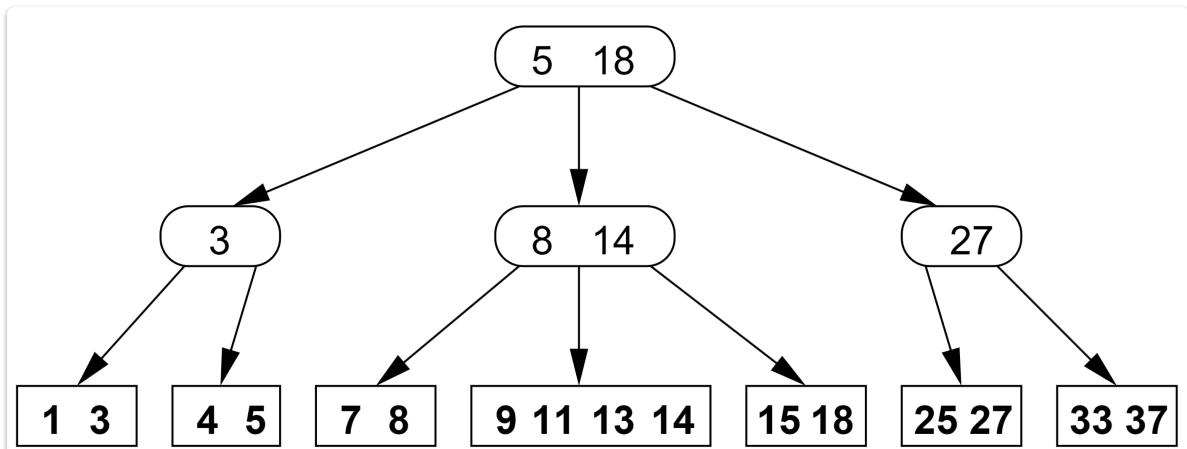
Motivation und Konsequenz: m kann größer gewählt werden, T' hat geringere Höhe!

Eigenschaften

- **Suchen:** Man muss im Unterschied zum B-Baum in jedem Fall bis zu einem Blatt gehen. Das erhöht die mittlere Anzahl von Schritten jedoch kaum, zumal der B*-Baum i.A. ja auch geringere Höhe hat.
- **Einfügen:** Das Prinzip ist das gleiche wie beim B-Baum. Kleine Unterschiede ergeben sich dadurch, dass es nur einen trennenden Schlüssel (keine trennenden Datensätze) gibt.
- **Entfernen:** Der zu entfernende Datensatz liegt immer in einem Blatt. Verweise in darüberliegenden Zwischenknoten müssen ggfs. aktualisiert werden.

Beispiel zu B*-Bäumen

Beispiel: Ein B*-Baum mit $m = 3$ und $k^* = 2$



7. Hashing

Einleitung

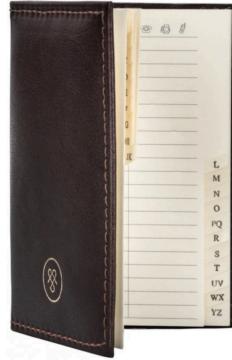
Wie auch [sem_2/AlgoDat/vo/6. Suchbäume](#) ist Hashing eine Lösung für das Wörterbuchproblem

Beispiel

- Vergleich mit einem „Telefonregister“:
Eine Seite für jeden Anfangsbuchstaben.
- Einfache Berechnung der Position in der Hashtabelle mit der Ordinalzahl (`ord`) des ersten Buchstabens im Namen s (z.B. $\text{ord}('A') = 0$, $\text{ord}('B') = 1, \dots$).
- **Hashfunktion h :** hier $h(s) = \text{ord}(s[0])$.

0 ("A")	Anna 123
1 ("B")	Barbara 222
2 ("C")	
3 ("D")	Doris 404, Daniel 343
4 ("E")	
5 ("F")	
6 ("G")	Günther 777
7 ("H")	
...	...
25 ("Z")	

Hashtabelle T



Grundlagen von Hashing

Hashtabelle:

- Wir gehen davon aus, dass die Hashtabelle T mit einer vorgegebenen Tabellengröße m als Array mit den Indizes $0, \dots, m - 1$ realisiert wird.
- Für eine Hashtabelle der Größe m , die aktuell n Schlüssel speichert, ist $\alpha = \frac{n}{m}$ der **Belegungsfaktor** der Tabelle.

Hashfunktion:

- Wir wählen eine Hashfunktion $h : \mathcal{K} \rightarrow \{0, \dots, m - 1\}$, die jedem Schlüssel $k \in \mathcal{K}$ einen eindeutigen – aber i.A. nicht umgekehrt eindeutigen – Hashwert zuordnet.

Vorteil: Laufzeit für die Operationen Suchen, Einfügen und Entfernen liegt im **Idealfall in $O(1)$** , wenn wir annehmen, dass $h(s)$ in konstanter Zeit berechnet werden kann.

Einschränkung:

- Gilt $h(k) = h(k')$ für $k \neq k'$, d.h., zwei verschiedene Schlüssel haben den gleichen Hashwert, so wird dies **Kollision** genannt.
- $O(1)$ gilt nur unter der Annahme, dass die Anzahl der Kollisionen vernachlässigbar klein ist.
- Im Erwartungsfall (bei entsprechender Konfiguration) erreichbar.
- Im Worst-Case liegt der Aufwand in $O(n)$.

Zu klärende Punkte:

- Wie erfolgt die Kollisionsbehandlung?
 - Verkettung der Überläufer
 - Offene Hashverfahren
- Was ist eine gute Hashfunktion?
 - Divisions-Rest-Methode
 - Multiplikationsmethode
- Wie soll die Tabellengröße m gewählt werden?

Alle diese Aspekte beeinflussen die Güte/Effizienz der Hashtabelle.

Hashfunktionen

Was charakterisiert eine gute Hashfunktion?

Vor allem:

- Verwendete Schlüssel sollen möglichst gleichmäßig auf alle Plätze $0, \dots, m - 1$ der Hashtabelle aufgeteilt werden.
- Auch kleinste Änderungen im Schlüssel sollen zu einem anderen, möglichst unabhängigen Hashwert führen.
- Das Beispiel mit der Ordnungszahl des Anfangsbuchstabens ist i.A. keine gute Wahl.

Divisions-Rest-Methode

Annahme: $k \in \mathbb{N}$

Berechnung:

$$h(k) = k \mod m$$

Eigenschaften:

- Die Hashfunktion kann sehr schnell berechnet werden.
- Die richtige Wahl von m ist hier sehr wichtig.
- Eine gute Wahl für m ist eine **Primzahl**.

Schlechte Wahl für m :

- $m = 2^p$: Nur die letzten p Binärziffern spielen eine Rolle!
- $m = 10^p$: Analog bei dezimalen Zahlen.
- $m = r^p$: Analog bei r -adischen Zahlen.
- Aber auch $m = r^p \pm c$ für kleine c kann problematisch sein:
 - z.B. $m = 2^7 - 1 = 127$. Buchstaben als Zahlen interpretieren
 - $h("p") = (112) \mod 127 = 112$
 - $h("t") = (116) \mod 127 = 116$
 - $h("pt") = (112 * 128 + 116) \mod 127 = 14452 \mod 127 = 101$
 - $h("tp") = (116 * 128 + 112) \mod 127 = 14960 \mod 127 = 101$ (Schlüssel in denen zwei Buchstaben vertauscht sind haben hier häufig den gleichen Hashwert)

Divisions-Rest-Methode für Strings

Schlüssel: String $s = (s_1, \dots, s_l)$, wobei $s_i \in \{0, \dots, 127\}$

Berechnung: $k = 128^{l-1}s_1 + 128^{l-2}s_2 + \dots + s_l$

Die sehr großen ganzzahligen Werte sind problematisch!

Berechnung mit Horner-Schema:

$$k = (\dots ((s_1 \cdot 128 + s_2) \cdot 128 + s_3) \cdot 128 + \dots + s_{l-1}) \cdot 128 + s_l$$

Es gilt:

$$\begin{aligned} k \mod m &= (\dots ((s_1 \cdot 128 \mod m + s_2) \mod m \cdot 128 \mod m + s_3) \\ &\quad \mod m \cdot 128 \mod m + \dots + s_{l-1}) \mod m \cdot 128 \mod m + s_l \mod m \end{aligned}$$

Konsequenz: Keine Zwischenresultate $> (m - 1) \cdot 128 + 127$. Berechnung mit üblichen Integer-Typen so gut möglich.

Multiplikationsmethode

Grundlegende Idee:

- Wir nehmen wieder an: Schlüssel $k \in \mathbb{N}$
- Gegeben: irrationale Zahl A
- Berechnung:

$$h(k) = \lfloor m \cdot (k \cdot A \mod 1) \rfloor$$

wobei $(k \cdot A \mod 1) = k \cdot A - \lfloor k \cdot A \rfloor \in [0, 1)$

- Der Schlüssel wird mit A multipliziert, der ganzzahlige Anteil des Resultats wird abgeschnitten.
- Man erhält einen Wert in $[0, 1)$, dieser wird mit der Tabellengröße m multipliziert, das Ergebnis gerundet.
- **Eigenschaft:** die gleichmäßige Streuung bestätigt:
 - Für eine Schlüsselmenge $1, 2, 3, \dots$ liegt $i \cdot A \mod 1$ – bis auf den Faktor $1/m$ – im gleichen Intervall zwischen zuvor ermittelten Werten, 0 und 1.

Wahl für A

Allgemein:

- Die Wahl von m ist hier unkritisch, sofern A eine irrationale Zahl ist.

Beste Wahl für A : der goldene Schnitt

$$A = \Phi^{-1} = \frac{\sqrt{5} - 1}{2} \approx 0.6180339887\dots$$

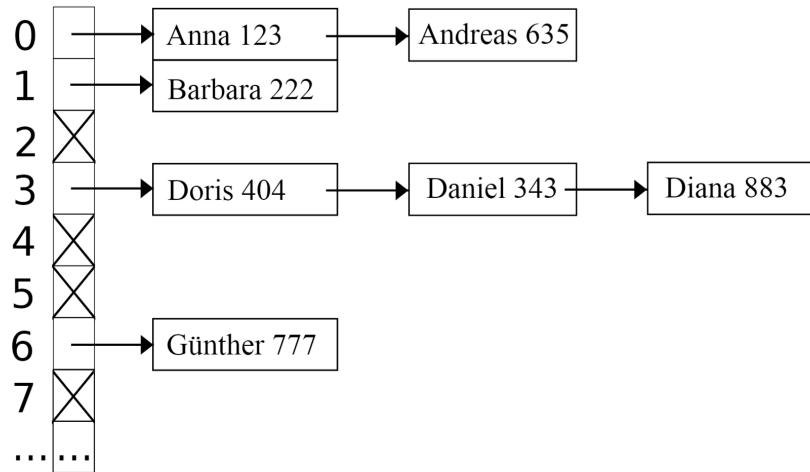
$$\begin{array}{c} a \quad b \\ \text{ca. } 61,8 \% \quad \text{ca. } 38,2 \% \\ \underbrace{}_{a+b} \end{array} \quad \frac{a}{a+b} = \frac{b}{a}$$

- Eine Begründung warum Φ^{-1} die beste Wahl ist findet sich in: D.E. Knuth: *The Art of Computer Programming*, Vol.3: *Sorting and Searching*, Addison-Wesley, 1973.

Kollisionsbehandlung, Verkettung der Überläufer

Idee

Idee: Jedes Element der Hashtabelle ist eine verkettete Liste.



Initialisierung

Eingabe: Hashtabelle T = Array von m Verweisen auf die jeweils ersten Elemente.

Ergebnis: Initialisierte Hashtabelle.

```
Initialize(T, m):
  for i ← 0 bis m - 1
    T[i] = null
```

Erläuterung:

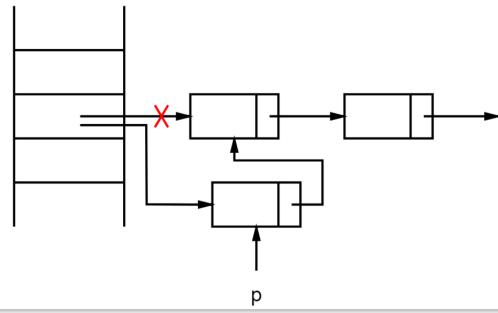
- Es wird ein Array T der Größe m erstellt.
- Jedes Element $T[i]$ dieses Arrays wird auf `null` gesetzt, was bedeutet, dass zu Beginn keine Elemente in der Hashtabelle vorhanden sind.
- m repräsentiert die Größe der Hashtabelle oder die Anzahl der Buckets.

Einfügen

Eingabe: Hashtabelle T und einzufügendes Element p .

Ergebnis: Hashtabelle T mit neu eingetragenem Element p .

```
Insert(T, p):
pos ← h(p.key)
p.next ← T[pos]
T[pos] ← p
```



Erläuterung:

1. Der Hashwert des Schlüssels von Element p ($p.key$) wird mit der Hashfunktion h berechnet. Dieser Hashwert (pos) bestimmt den Index im Array T , an dem das Element eingefügt werden soll.
2. $p.next \leftarrow T[pos]$: Das `next`-Feld des einzufügenden Elements p wird auf das aktuelle Element gesetzt, das sich an der berechneten Position $T[pos]$ befindet. Dies ist wichtig für die Kollisionsbehandlung durch Verkettung. Wenn der Bucket leer ist, wird $p.next$ auf `null` gesetzt.
3. $T[pos] \leftarrow p$: Der Zeiger im Array T an der Position pos wird nun auf das neu eingefügte Element p gesetzt. Dadurch wird p zum ersten Element in der Liste (oder dem einzigen Element, falls der Bucket vorher leer war) an dieser Position.

Hinweis: Der Hashwert (und damit die Position in der Hashtabelle) für $p.key$ wird mit der Funktion h berechnet.

Suchen

Eingabe: Hashtabelle T und gesuchter Schlüssel k .

Rückgabewert: Gesuchtes Element p .

```
Search(T, k):
p = T[h(k)]
while p ≠ null und p.key ≠ k
    p ← p.next
return p
```

Erläuterung:

1. Der Hashwert des gesuchten Schlüssels k wird mit der Hashfunktion h berechnet. Dies ergibt die Startposition $T[h(k)]$ in der Hashtabelle, an der sich das gesuchte Element befinden könnte.
2. $p \leftarrow T[h(k)]$: Der Zeiger p wird auf das erste Element in der Liste an der berechneten Position gesetzt.
3. **while** $p \neq \text{null}$ und $p.key \neq k$: Es wird so lange durch die verkettete Liste gegangen, wie der aktuelle Zeiger p nicht `null` ist (also das Ende der Liste noch nicht erreicht wurde)

und der Schlüssel des aktuellen Elements ($p.key$) nicht mit dem gesuchten Schlüssel k übereinstimmt.

4. $p \leftarrow p.next$: Wenn der Schlüssel des aktuellen Elements nicht der gesuchte Schlüssel ist, wird p zum nächsten Element in der verketteten Liste gesetzt.

Entfernen eines Elements aus einer Hashtabelle

Eingabe: Hashtabelle T und Schlüssel k des zu entfernenden Elements (Annahme: Element mit Schlüssel k ist in T enthalten).

Ergebnis: Element mit dem Schlüssel k wurde aus T entfernt.

Algorithmus Remove(T, k) :

1. Berechne den Hashwert des Schlüssels: $pos \leftarrow h(k)$
2. Initialisiere q als `null` (Zeiger auf den vorherigen Knoten).
3. Setze p auf den Kopf der Liste an der Hash-Position: $p \leftarrow T[pos]$
4. **Suche nach dem Element:**

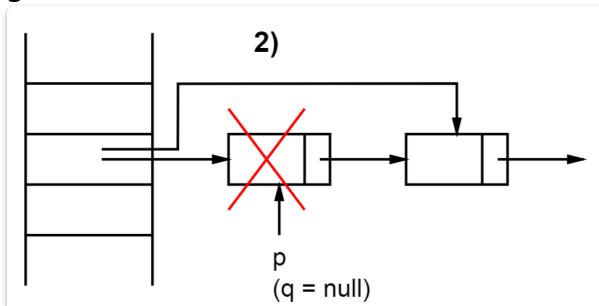
```
while p.key != k:
    q ← p
    p ← p.next
```

- Die Schleife iteriert durch die verkettete Liste an der Position pos , bis das Element mit dem Schlüssel k gefunden wird.
- q folgt p und zeigt immer auf den vorherigen Knoten.

5. **Fall 1: Element am Kopf der Liste (q ist `null`)**

```
if q == null:
    T[pos] ← T[pos].next
```

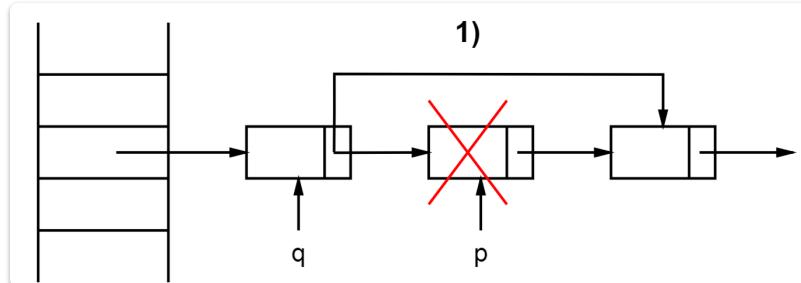
- Wenn q `null` ist, bedeutet das, dass das zu entfernende Element das erste Element in der Liste an der Position pos ist.
- In diesem Fall wird der Zeiger in der Hashtabelle direkt auf das nächste Element gesetzt.



6. **Fall 2: Element nicht am Kopf der Liste (q ist nicht `null`)**

```
else:  
    q.next ← p.next
```

- Wenn q nicht `null` ist, bedeutet das, dass das zu entfernende Element nicht das erste ist.
- Der `next`-Zeiger des vorherigen Elements (q) wird auf das nächste Element nach dem zu entfernenden Element (p) gesetzt. Dadurch wird p aus der Liste entfernt.



`Remove(T, k):`

```
pos ← h(k)  
q ← null  
p ← T[pos]  
while p.key ≠ k  
    q ← p  
    p ← p.next  
if q = null  
    T[pos] ← T[pos].next  
else  
    q.next ← p.next;
```

Kollisionsbehandlung, Offene Hashverfahren

Grundlegende Idee

- Alle Datensätze werden direkt in einem einfachen Array gespeichert.
- Pro Platz ein Flag $f_i \in \{\text{frei}, \text{besetzt}, \text{wieder frei}\}$.

Kollisionsbehandlung: Wenn ein Platz belegt ist, werden weitere Reihenfolge weitere Plätze betrachtet (Sondieren).

Am Anfang sind alle Plätze frei:

0 ("A")		frei
1 ("B")		frei
2 ("C")		frei
3 ("D")		frei
4 ("E")		frei
5 ("F")		frei
6 ("G")		frei
7 ("H")		frei
...		frei
25 ("Z")		frei

- Hashfunktion:** Ordinalzahl des ersten Buchstabens im Namen
- Sondierreihenfolge:** einfach die nächste Position

0 ("A")	Anna 123	besetzt
1 ("B")	Albert 127	besetzt
2 ("C")		frei
3 ("D")	Doris 404	besetzt
4 ("E")		frei
5 ("F")		frei
6 ("G")	Günther 777	besetzt
7 ("H")		frei
...		frei
25 ("Z")		frei

- Anna wurde vor Albert eingetragen, Albert kommt daher auf die nächste freie Position 1.
- Ein weiterer Eintrag für Andreas würde an Position 2 gespeichert werden, Armin danach an Position 4, usw.

- **Entfernen:** Flag wird auf wieder frei gesetzt.
 - Im Beispiel wird Anna entfernt.
- Würde nach $h("Albert") = 1$ gesucht, würde Albert nicht mehr gefunden werden, da die Sondierung bei Position 0 abbricht!
- Ein neuer Eintrag für Andreas würde wieder an Position 0 gespeichert werden.

0 ("A")	Anna 123	wieder frei
1 ("B")	Albert 127	besetzt
2 ("C")		frei
3 ("D")	Doris 404	besetzt
4 ("E")		frei
5 ("F")		frei
6 ("G")	Günther 777	besetzt
7 ("H")		frei
...		frei
25 ("Z")		frei

Im Detail

- Alle Elemente werden – im Gegensatz zur Verkettung der Überläufer – direkt im Array gespeichert.
- Zu jedem Platz $i = 0, \dots, m - 1$ wird ein Flag $f_i \in \{\text{frei}, \text{besetzt}, \text{wieder frei}\}$ gespeichert.
- In der anfangs leeren Tabelle gilt für alle i : $f_i = \text{frei}$.
- Beim Einfügen wird das neue Element am ersten mit frei oder wieder frei markierten Platz eingefügt, das Flag wird auf besetzt gesetzt.
- Die Suche durchmustert alle Plätze bis der gesuchte Schlüssel entweder gefunden wird oder ein Platz als frei markiert ist (Schlüssel nicht enthalten).
- Das Entfernen setzt das Flag für das zu entfernende Element auf wieder frei.

Sondierung

Kollisionsbehandlung: Wenn ein Platz belegt ist, so werden in einer bestimmten Reihenfolge weitere Plätze in Betracht gezogen.

Sondierungsreihenfolge (Probing): Reihenfolge der auszuprobierenden Plätze.

→ Die Hashfunktion wird zu einer **Sondierungsfunktion** $h(k, i)$ für Schlüssel k und Positionen $i = 0, 1, \dots, m - 1$ erweitert, die Sondierungsreihenfolge ist dann $h(k, 0), h(k, 1), \dots, h(k, m - 1)$.

Lineares Sondieren

Gegeben: Eine normale Hashfunktion:

$$h' : K \rightarrow \{0, 1, \dots, m - 1\}$$

Lineares Sondieren: Wir definieren für $i = 0, 1, \dots, m - 1$:

$$h(k, i) = (h'(k) + i) \mod m$$

Beispiel

Beispiel: $m = 8$, $h'(k) = k \mod m$

Schlüssel und Wert der Hashfunktion:

k	10	19	31	22	14	16
$h'(k)$	2	3	7	6	6	0

Belegung der Hashtabelle:

0	1	2	3	4	5	6	7
14	16	10	19			22	31

Durchschnittliche Zeit: Für eine erfolgreiche Suche $\frac{9}{6} = 1,5$ (siehe Tabelle).

k	10	19	31	22	14	16					
	1	+	1	+	1	+	3	+	2	=	9

Probleme

- Nach dem Einfügen ist die Wahrscheinlichkeit für einen neu einzufügenden Schlüssel in der Hashtabelle an einer gewissen Position gespeichert zu werden für die verschiedenen Positionen unterschiedlich.
- Wird ein Platz belegt, dann verändert sich die Wahrscheinlichkeit für das Einfügen an seinem nachfolgenden Platz.

Beispiel:

- In der leeren Tabelle haben alle Plätze die gleiche Wahrscheinlichkeit.
- Nach dem Einfügen von verschiedenen Schlüsseln verändern sich die Wahrscheinlichkeiten. Z.B. werden auf der rechten Seite im Eintrag $T[2]$ nach dem Einfügen von Anna und Barbara alle Schlüssel k mit $h'(k) = 0$ oder $h'(k) = 1$ oder $h'(k) = 2$ gespeichert; im Eintrag $T[5]$ dagegen nur alle Schlüssel k mit $h'(k) = 5$.

0 ("A")	1/26	0 ("A")	Anna 123
1 ("B")	1/26	1 ("B")	Barbara 222
2 ("C")	1/26	2 ("C")	3/26
3 ("D")	1/26	3 ("D")	Doris 404
4 ("E")	1/26	4 ("E")	2/26
5 ("F")	1/26	5 ("F")	1/26
6 ("G")	1/26	6 ("G")	Günther 777
7 ("H")	1/26	7 ("H")	2/26
...
25 ("Z")	1/26	25 ("Z")	1/26

Probleme:

- Lange belegte Teilstücke der Hashtabelle haben eine stärkere Tendenz zu wachsen als kurze.
- Dieser Effekt wird noch verstärkt, weil lange belegte Teilstücke zu größeren Zusammenwachsen, wenn die Lücken zwischen ihnen geschlossen werden.
- Als Folge dieses Phänomens der primären Häufung (primary clustering) verschlechtert sich die Effizienz des linearen Sondierens drastisch, sobald sich der Belegungsfaktor α dem Wert 1 nähert.

Uniform Hashing

- Idealform des Sondierens.
- Jeder Schlüssel erhält mit gleicher Wahrscheinlichkeit eine bestimmte der $m!$ Permutationen von $0, 1, \dots, m - 1$ als Sondierungsreihenfolge zugeordnet.
- Ist in der Praxis schwierig zu implementieren und wird daher mit den nachfolgenden Verfahren approximiert.

Quadratisches Sondieren

Idee: Um die primäre Häufung des linearen Sondierens zu vermeiden, wird beim quadratischen Sondieren für Schlüssel k von $h'(k)$ aus mit quadratisch wachsendem Abstand nach einem freien Platz gesucht.

Gegeben: Eine normale Hashfunktion:

$$h' : K \rightarrow \{0, 1, \dots, m - 1\}$$

Quadratisches Sondieren: Sondierungsfunktion lautet nun:

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mod m$$

Dabei sind c_1 und c_2 geeignete gewählte Konstanten.

Beispiel

Beispiel: $m = 8$, $h'(k) = k \mod m$, $c_1 = c_2 = \frac{1}{2}$, gleiche Schlüssel wie vorhin.

Schlüssel und Wert der Hashfunktion:

k	10	19	31	22	14	16
$h'(k)$	2	3	7	6	6	0

0	1	2	3	4	5	6	7
		10	19			22	31

$$\begin{aligned} 14 \rightarrow 6 &\rightarrow 6 + \frac{1}{2}1 + \frac{1}{2}1^2 \mod 8 = 7 \\ &\rightarrow 6 + \frac{1}{2}2 + \frac{1}{2}2^2 \mod 8 = 1 \end{aligned}$$

0	1	2	3	4	5	6	7
16	14	10	19			22	31

Beispiel und Analyse

0	1	2	3	4	5	6	7
16	14	10	19			22	31

Durchschnittliche Zeit: Für eine erfolgreiche Suche $\frac{8}{6} \approx 1.33$.

k	10	19	31	22	14	16	
	1	+ 1	+ 1	+ 1	+ 3	+ 1	= 8

Probleme: Primäre Häufungen werden vermieden, aber ein anderes Phänomen, die sekundären Häufungen (*secondary clustering*) können auftreten.

Güte von Kollisionsbehandlung

Theoretische Analyseergebnisse:

Durchschnittliche Anzahl der Sondierungen für große m, n :

α	Verkettung		offene Hashverfahren					
			lineares S.		quadr. S.		unif. hashing	
	erfolgreich	erfolglos	er	el	er	el	er	el
0.5	1.250	0.50	1.5	2.5	1.44	2.19	1.39	2
0.9	1.450	0.90	5.5	50.5	2.85	11.40	2.56	10
0.95	1.475	0.95	10.5	200.5	3.52	22.05	3.15	20
1.0	1.500	1.00	—	—	—	—	—	—

er: erfolgreiche Suche, el: erfolglose Suche

Ergebnisse von D.E. Knuth: The Art of Computer Programming, Vol.3: Sorting and Searching, Addison-Wesley, 1973.

Double Hashing

Idee: Die Effizienz des uniformen Sondierens wird bereits annähernd erreicht, wenn man statt einer zufälligen Permutation für die Sondierungsfolge eine zweite Hashfunktion verwendet.

Gegeben: Zwei Hashfunktionen:

$$h_1, h_2 : K \rightarrow \{0, 1, \dots, m - 1\}$$

Double Hashing: Wir definieren für $i = 0, 1, \dots, m - 1$:

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \mod m$$

Wahl von $h_2(k)$: Für alle Schlüssel k muss die Sondierungsfolge alle Plätze $0, \dots, m - 1$ erreichen. Das bedeutet, dass $h_2(k) \neq 0$ sein muss und m nicht teilen darf. m sollte eine Primzahl sein, h_2 sollte unabhängig von h_1 gewählt werden.

Beispiel

$$m = 7, h_1(k) = k \mod 7, h_2(k) = 1 + (k \mod 5)$$

k	10	19	31	22	14	16
$h_1(k)$	3	5	3	1	0	2
$h_2(k)$	1	5	2	3	5	2

0	1	2	3	4	5	6
			10		19	

0	1	2	3	4	5	6
31	22		10		19	

(3) (1) (2)

0	1	2	3	4	5	6
31	22	16	10		19	14

(1) (4) (3) (2) (5)

Durchschnittliche Zeit: Für eine erfolgreiche Suche ist $\frac{12}{6} = 2$. Dies ist jedoch ein untypisch schlechtes Beispiel für Double Hashing.

$$\begin{array}{c|ccccccccc}
k & 10 & 19 & 31 & 22 & 14 & 16 \\
\hline
& 1 & + & 1 & + & 3 & + & 1 & + & 5 & + & 1 & = & 12
\end{array}$$

Praxis

- Im Allgemeinen ist Double Hashing **effizienter als quadratisches Sondieren**.
- In der **Praxis** entsprechen die **Ergebnisse** von Double Hashing nahezu **denen von uniformen Hashing**.

Verbesserung nach Brent [1973]

Idee: Wenn beim Einfügen eines Schlüssels ein sonderter Platz j mit $k' = T[j]$. key belegt ist, setze

$$j_1 = (j + h_2(k)) \bmod m$$

$$j_2 = (j + h_2(k')) \bmod m.$$

Ist nun j_1 besetzt aber j_2 frei, verschiebe k' auf j_2 um Platz für k auf j_1 zu machen.

Angewendet auf unser Beispiel:

0	1	2	3	4	5	6
			10	→	19	

0	1	2	3	4	5	6
			31	10	19	

Rest immer frei

0	1	2	3	4	5	6
14	22	16	10	10	19	

31

Durchschnittliche Zeit: Für eine erfolgreiche Suche ist $\frac{7}{6} \approx 1.17$.

$$\begin{array}{c|ccccccccc}
k & 10 & 19 & 31 & 22 & 14 & 16 \\
\hline
& 2 & + & 1 & + & 1 & + & 1 & + & 1 & = & 7
\end{array}$$

Einfügen nach Brent

Eingabe: Hashtabelle T und neuer Schlüssel k .

```

Insert-Brent(T, k):
j ← h1(k)
while T[j].status = used
    k' ← T[j].key
    j1 ← (j + h2(k)) mod m
    j2 ← (j + h2(k')) mod m
    if T[j1].status ≠ used oder T[j2].status = used
        j ← j1
    else
        T[j] ← k
        k ← k'
        j ← j2
T[j] ← k
T[j].status ← used

```

Analyseergebnis zur Verbesserung nach Brent

Anzahl der Sondierungen: Im Durchschnitt für große m und n :

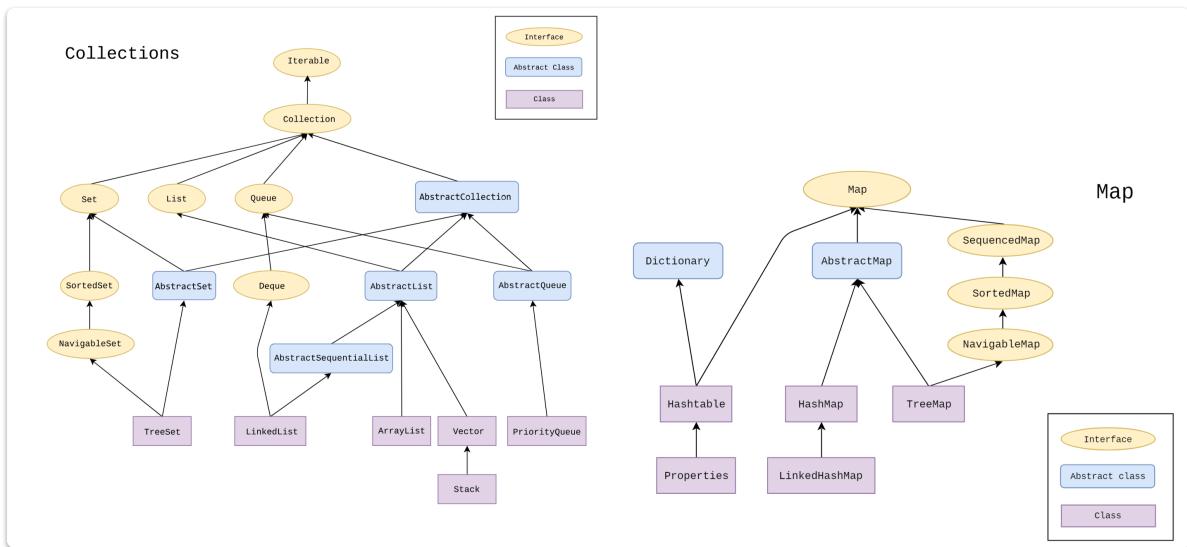
- Erfolgreiche Suche $\approx \frac{1}{1-\alpha}$
- Erfolgreiche Suche < 2.5 (unabhängig von α für $\alpha \leq 1$)

Vorteil: Durchschnittlicher Aufwand einer erfolgreichen Suche liegt selbst im Extremfall $\alpha = 1$ in $\Theta(1)$.

Offene Hashverfahren: Eignung und Reorganisation

- Bei offenen Hashverfahren ist der Belegungsfaktor $\alpha = \frac{n}{m}$ immer kleiner (max. gleich) 1, offensichtlich können nicht mehr als m Elemente gespeichert werden.
- Belegungsfaktoren sehr nahe 1 sind i.A. ungünstig, da die Anzahl der zu sondierenden Positionen sehr groß werden kann!
- Gegebenenfalls ist eine **Reorganisation** notwendig, d.h., dass eine gänzlich neue Hashtabelle z.B. mit doppelter Größe aufgebaut wird, wenn mehr als die ursprünglich erwartete Anzahl an Elementen zu speichern ist (Aufwand i.A. $\Theta(n)$).
- Generell sind offene Hashverfahren besser geeignet, wenn die Anzahl der zu speichernden Elemente vorab bekannt ist und selten oder gar nicht Elemente entfernt werden; häufiges Entfernen bewirkt, dass sich die Sondierungsketten für die Suche verlängern; eine regelmäßige Reorganisation kann dann ebenfalls sinnvoll bzw. notwendig werden.

8. Praktische Datenstrukturen



Interfaces

- Für unterschiedliche Datenstrukturen.
- Schnittstellen für den Zugriff auf die Datenstrukturen.
- Konkrete Implementierungen realisieren diese Interfaces.

Collection:

- Eine Collection verwaltet Objekte (Elemente).
- Interface enthält generelle Methoden (für alle Collection-Typen).
- Es gibt keine direkte Implementierung dieses Interfaces.

Set:

- Eine Collection, die keine Duplikate enthält.

List:

- Eine geordnete Collection (die Duplikate enthalten kann).
- Elemente können über einen Index angesprochen werden (nicht immer effizient).

Queue/Deque:

- Verwalten von Warteschlangen.
- FIFO, Prioritätswarteschlangen.
- Einfügen und Löschen an beiden Enden bei Deque („double ended queue“).

Map:

- Maps verwalten Schlüssel mit dazugehörigen Werten.

SortedSet und SortedMap:

- Sind spezielle Versionen von Set und Map, bei denen die Elemente (Schlüssel) in aufsteigender Reihenfolge verwaltet werden.

Konzept	Interface				
	Set	List	Queue	Deque	Map
Arrays	–	ArrayList	–	ArrayDeque	–
Bäume	TreeSet	–	–	–	TreeMap
Hashtabellen	HashSet	–	–	–	HashMap
Heap	–	–	PriorityQueue	–	–
Verkettete Listen	–	LinkedList	LinkedList	LinkedList	–

Listenimplementierung

Listen

ArrayList:

- Indexzugriff auf Elemente ist überall gleich schnell ($O(1)$).
- Einfügen und Löschen ist am Listenende schnell und wird mit wachsender Entfernung vom Listenende langsamer ($O(n)$).

LinkedList:

- Indexzugriff auf Elemente ist an den Enden schnell und wird mit der Entfernung von den Enden langsamer ($O(n)$).
 - Einfügen und Löschen ohne Indexzugriff ist überall gleich schnell ($O(1)$).
-

Queue

LinkedList ist auch eine Implementierung von Queue.

PriorityQueue:

- Ist als Min-Heap implementiert.
 - Einfügen eines Elements und Löschen des ersten Elements in einer Queue der Größe n sind in $O(\log n)$.
 - Löschen eines beliebigen Elements aus einer Queue der Größe n ist in $O(n)$.
 - Lesen des ersten Elements in einer Queue ist in konstanter Zeit möglich ($O(1)$).
-

Beispiel zu ArrayList, LinkedList und PriorityQueue

```
import java.util.*;  
  
public class Example {  
    public static void main(String[] args) {  
        List<Integer> myAL = new ArrayList<>();  
        myAL.add(42); myAL.add(17);  
        System.out.println(myAL); // [42, 17]  
  
        List<Integer> myLL = new LinkedList<>();  
        myLL.add(42); myLL.add(17);  
        System.out.println(myLL); // [42, 17]  
  
        PriorityQueue<Integer> myPQ = new PriorityQueue<>();  
        myPQ.add(750);  
        myPQ.add(500);  
        myPQ.add(900);  
        myPQ.add(100);  
  
        while (!myPQ.isEmpty()) {  
            System.out.print(myPQ.remove() + " ");  
        } // 100 500 750 900  
    }  
}
```

TreeSet und HashSet

Set-Implementierungen: TreeSet

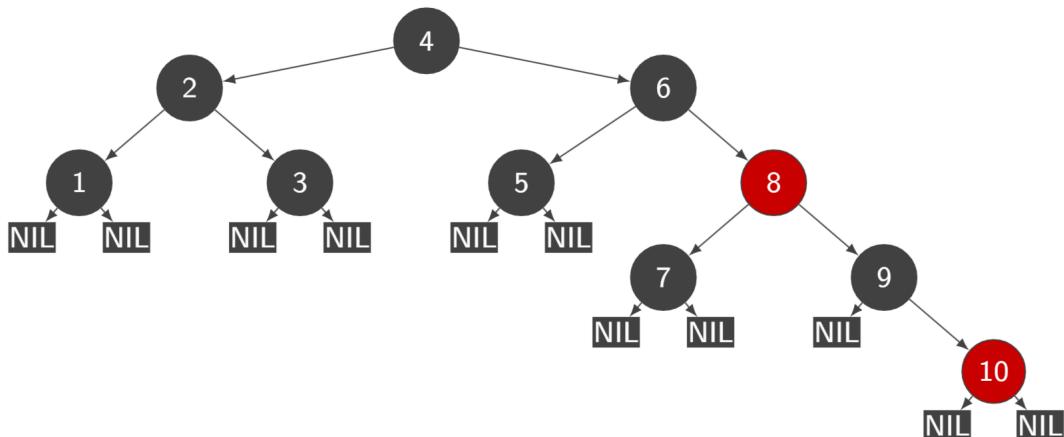
TreeSet:

- Implementiert als Rot-Schwarz-Baum.
- `null`-Elemente sind nicht erlaubt.
- Die Laufzeit von Einfügen, Suchen und Löschen eines Elements liegt bei einem Baum mit n Elementen in $O(\log n)$.
- Auf die Elemente eines `TreeSets` muss eine Ordnung definiert sein (müssen vergleichbar sein, d.h. das Interface `Comparable` implementieren).

Rot-Schwarz-Baum:

- Variante eines balancierten Suchbaums.
- Kann als Spezialfall eines B-Baums der Ordnung 4 betrachtet werden.
- Einfügen und Löschen ist effizienter als in B-Bäumen solange die Datenmenge nicht zu groß ist.
- Selbstbalancierender binärer Suchbaum: jeder Knoten entweder rot oder schwarz.
- Die Wurzel ist schwarz. Jeder Blattknoten (NIL) ist schwarz.
- Kindknoten von einem roten Knoten sind schwarz.
- Für jeden Knoten gilt, dass alle Pfade von diesem Knoten zu nachfolgenden Blattknoten die gleiche Anzahl an schwarzen Knoten beinhalten.
- Einfügen und Löschen können eine Rebalancierung erfordern.

Beispiel:



Set-Implementierungen: HashSet

HashSet:

- `null`-Elemente sind zulässig.
- Einfügen, Suchen und Löschen sind in konstanter Zeit möglich (abhängig von der Verteilung der Einträge in der Hashtabelle).
- Aber: Rehashing kann zu Performance-Problemen führen (siehe auch API-Beschreibung).

Implementierung:

- Hashing mit Verkettung der Überläufer. Falls die Liste zu lange wird, wird sie in eine `TreeMap` umgewandelt.
 - Maximaler Belegungsfaktor = 0.75.
 - Bei Überschreitung des Belegungsfaktors verdoppelt sich die Größe. (Größe der Hashtabelle ist immer eine Zweierpotenz).
-

Beispiel zu TreeSet und HashMap

```
import java.util.*;

public class Example {
    public static void main(String[] args) {
        Set<String> myTS = new TreeSet<>();
        myTS.add("apples");
        myTS.add("bananas");
        myTS.add("strawberries");
        myTS.add("grapes");
        myTS.add("strawberries"); // adding duplicate elements will be ignored
        System.out.println(myTS); // [apples, bananas, grapes, strawberries]

        Set<String> myHS = new HashSet<>();
        myHS.add("apples");
        myHS.add("bananas");
        myHS.add("strawberries");
        myHS.add("grapes");
        myHS.add("strawberries"); // adding duplicate elements will be ignored
        System.out.println(myHS); // [strawberries, bananas, apples, grapes]
    }
}
```

Verschiedene Implementierungen von Hashing

mehr siehe hier

Beispiele für Verkettung der Überläufer:

	Belegungsfaktor	Wachstumsfaktor	Tabellengröße
Java	0.75	2	Zweierpotenzen
C++	1	>2	Primzahlen
Go	6.5	2	Zweierpotenzen

Beispiele für offenes Hashing:

	Belegungsfaktor	Wachstumsfaktor	Tabellengröße	Sondierung
Python	0.66	2	Zweierpotenzen	Quadratisch
Ruby	0.5	2	Zweierpotenzen	Quadratisch

Anmerkung: Die Sondierungsfunktion verwendet in beiden Fällen zusätzlich eine sogenannte Perturbation.

Maps

- Maps sind eine Verallgemeinerung von Arrays mit einem beliebigen Indextyp (nicht nur `int`).
- Eine Map ist eine Menge von Schlüssel-Werte Paaren.
 - Schlüssel müssen innerhalb einer Map eindeutig sein.
 - Werte müssen nicht eindeutig sein.

Arten von Maps: Wie bei Sets gibt es grundsätzlich zwei Versionen.

- **HashMap:** Implementierung wie HashSet.
- **TreeMap:** Implementierung wie TreeSet.

Anmerkung: Genau genommen sind Tree- und HashMap die primitiven Implementierungen die von Tree- und HashSet verwendet werden.

Operationen auf Maps: Bestimmte Methoden wie z.B. `put`, `get`, `containsKey`, `containsValue`, `remove` etc. werden angeboten.

Beispiele zu HashMap und TreeMap

```
import java.util.*;

public class Example {
    public static void main(String[] args) {
        // HashMap Beispiel
        Map<String, Integer> myHM = new HashMap<>();
        myHM.put("one", 1);
        myHM.put("two", 2);
        myHM.put("three", 3);
        myHM.put("four", 4);
        myHM.put("five", 5);
        // only add if key does not exist or is mapped to `null`

        System.out.println("myHM: " + myHM);
        // {one=1, two=2, three=3, four=4, five=5}

        String id = "three";
        if (myHM.containsKey(id)) {
            System.out.println("Found key " + id + ". Value: " + myHM.get(id));
            // Found key three. Value: 3
        }

        // TreeMap Beispiel
        Map<String, Integer> myTM = new TreeMap<>();
```

```
myTM.put("one", 1);
myTM.put("two", 2);
myTM.put("three", 3);
myTM.put("four", 4);
myTM.put("five", 5);
// only add if key does not exist or is mapped to `null`  
  
System.out.println("myTM: " + myTM);
// {five=5, four=4, one=1, three=3, two=2} (sortiert nach Schlüssel)  
  
id = "four";
if (myTM.containsKey(id)) {
    System.out.println("Found key " + id + ". Value: " + myTM.get(id));
    // Found key four. Value: 4
}
}  
}
```

Abstrakte Klassen

- Unterstützen die Entwicklung neuer Klassen im Framework.
- Implementieren einen Teil eines Interfaces und lassen bestimmte Teile noch offen.

Neue Klasse implementieren:

- Auswählen einer geeigneten abstrakten Klasse von der geerbt wird.
- Implementierung aller abstrakten Methoden.
- Sollte die Collection modifizierbar sein, dann müssen auch einige konkrete Methoden überschrieben werden (siehe API).
- Testen der neuen Klasse (inklusive Performance).

Beispiele:

- `AbstractCollection`
- `AbstractSet`
- `AbstractList` (basierend auf Array)
- `AbstractSequentialList` (basierend auf verketteter Liste)
- `AbstractQueue`
- `AbstractMap`

API: API-Dokumentation jeder abstrakten Klasse beschreibt genau, wie man eine Klasse ableiten muss:

- Grundlegende Implementierung.
- Welche Methoden müssen implementiert werden.
- Welche Methoden müssen überschrieben werden, wenn man Modifikationen zulassen möchte.

Algorithmen im Collections-Framework

Algorithmen:

- Das Collections-Framework bietet auch Algorithmen für die Verarbeitung von Container-Klassen an.
- Diese Algorithmen werden als statische Methoden (polymorphe Methoden) in der Hilfsklasse `Collections` gesammelt.

Beispiele:

- `sort` sortiert die Elemente einer generischen Liste nach aufsteigender Größe.
- `binarySearch` sucht ein Element in der sortierten Liste (Voraussetzung) und liefert einen Index zurück, wenn das Element gefunden wurde (ansonsten eine negative Zahl).
- `max` liefert das größte Element einer Collection.
- `shuffle` mischt die Elemente einer generischen Liste zufällig.

Beispiele zu Algorithmen im Collections-Framework

```
import java.util.*;

public class Example {
    public static void main(String[] args) {
        List<Integer> myAL = new ArrayList<>(Arrays.asList(5, 3, 1, 4));
        System.out.println("myAL: " + myAL); // [5, 3, 1, 4]

        Collections.sort(myAL);
        System.out.println("sort(myAL): " + myAL); // [1, 3, 4, 5]

        System.out.println("Collections.max(myAL): " + Collections.max(myAL)); // 5
        System.out.println("Collections.binarySearch(myAL, 3): "
            + Collections.binarySearch(myAL, 3)); // 1
        System.out.println("Collections.binarySearch(myAL, 42): "
            + Collections.binarySearch(myAL, 42)); // -5

        Collections.shuffle(myAL);
        System.out.println("shuffle(myAL): " + myAL); // z.B.: [1, 4, 3, 5]
    }
}
```

Timsort

- Zum Sortieren von Collections wird Timsort verwendet.
- Timsort ist eine Kombination von Mergesort und Insertionsort.
- Timsort wurde 2002 von Tim Peters für die Verwendung in Python entwickelt.
- Heute wird Timsort unter anderem in Python, Java, Android Plattform, V8 (Chrome Browser), Swift, Rust verwendet.

Grundlegende Idee: Finde schon sortierte Stücke (Runs) des Inputs S , die dann paarweise verschmolzen werden (Merging).

```

Timsort( $S$ )
runs  $\leftarrow$  partitioniere  $S$  in Runs
 $\mathcal{R} \leftarrow$  leerer Stack
while runs  $\neq \emptyset$ 
    entferne Run  $r$  von runs and pushe  $r$  auf den Stack  $\mathcal{R}$ 
    merge_collapse( $\mathcal{R}$ ) // ausbalanciertes Verschmelzen
while height( $\mathcal{R}$ )  $\neq 1$ 
    verschmelze die beiden oberen Runs am Stack

```

Berechnen von Runs

Min_Run:

- Runs sind mindestens `Min_Run` lang.
- `Min_Run` wird so gewählt, dass $\frac{\text{Größe des Arrays}}{\text{Min_Run}}$ eine Zweierpotenz oder etwas kleiner als eine Zweierpotenz ist.
- Experimente zeigen, dass ein Wert zwischen 32 und 64 für `Min_Run` optimal ist.

Finden von Runs:

- Startend bei dem ersten Element, das noch nicht in einem Run ist, füge so lange Elemente zum Run hinzu, wie die Sequenz entweder aufsteigend oder strikt absteigend ist.
- Falls die Sequenz strikt absteigend war, invertiere die Reihenfolge der Elemente.
- Falls die Länge der Sequenz größer `Min_Run` ist, ist damit ein Run gefunden.
- Andernfalls füge solange nachfolgende Elemente hinzu bis `Min_Run` erreicht ist. Sortiere den entstandenen Run mittels (binärem) Insertionsort.

Runs mit `Min_Run = 4`:



Verschmelzen der Runs (Merging)

- Nachdem das Array in Runs aufgeteilt und diese sortiert sind, werden diese wie bei Mergesort verschmolzen.

- Merging ist am effizientesten, wenn die Anzahl der Runs eine Zweierpotenz ist und die Runs möglichst gleich groß sind.
- Da Runs im Allgemeinen sehr unterschiedliche Größen haben können, wird versucht mithilfe von Invarianten eine sinnvolle Merge-Reihenfolge zu finden. Beachte dabei nur "benachbarte" Runs werden verschmolzen.
- Zusätzlich wird beim Merging eine Strategie namens Galloping verwendet, um möglicherweise vorhandene Strukturen in den Runs zu nutzen.

Invarianten und merge_collapse

- Stack R von Runs mit Aufbau $R[1], \dots, R[\text{height}(R)]$, wobei $R[1]$ top-of-stack ist.
- Die Länge von Run $R[i]$ wird mit r_i bezeichnet.

```
merge_collapse( $\mathcal{R}$ )
while height( $\mathcal{R}$ ) > 1
  if height( $\mathcal{R}$ ) > 2 und  $r_3 \leq r_2 + r_1$ 
    if  $r_3 < r_1$ 
      verschmelze  $\mathcal{R}[2]$  und  $\mathcal{R}[3]$  am Stack
    else
      verschmelze  $\mathcal{R}[1]$  und  $\mathcal{R}[2]$  am Stack
  elseif height( $\mathcal{R}$ ) > 1 und  $r_2 \leq r_1$ 
    verschmelze  $\mathcal{R}[1]$  und  $\mathcal{R}[2]$  am Stack
  else
    break
```

- Nach Ausführung von `merge_collapse(R)` gelten folgende Invarianten:
 - (1) $r_2 > r_1$
 - (2) $r_3 > r_1 + r_2$

Beispiel zu Invarianten

- Wiederholung Invarianten: (1) $r_2 > r_1$, (2) $r_3 > r_1 + r_2$
- Länge der Runs am Stack werden in Reihenfolge ..., r_3, r_2, r_1 angegeben.
- Stack mit Arrays der Größe: 138, 67, 15. Invarianten erfüllt.
- Nächster Run hat Größe 16. D.h. 138, 67, 15, 16. Invariante (2) nicht erfüllt!
- Mergen ergibt: 138, 67, 31. Invarianten erfüllt.
- Nächster Run hat Größe 17. D.h. 138, 67, 31, 17. Invariante (1) nicht erfüllt.
- Nächster Run hat Größe 15. D.h. 138, 67, 31, 17, 15. Invariante (1) nicht erfüllt!
- Mergen ergibt: 138, 67, 31, 32. Invariante (2) nicht erfüllt!
- Mergen ergibt: 138, 67, 63. Invarianten erfüllt.
- ...

Galloping

- Lineares Merging:
 - Arrays A und B sollen verschmolzen (gemerged) werden.
 - Vergleiche $A[0]$ und $B[0]$ und verschiebe den kleineren Wert in das Merge-Array.

- Ineffizient falls das selbe Array gewinnt.
- Timsort speichert, wie oft dasselbe Array gewinnt und wechselt in den Galloping-Mode falls dasselbe Array mal gewinnt.

$A = \boxed{1} \quad \boxed{3} \quad \boxed{4} \quad \boxed{5} \quad \boxed{7} \quad \boxed{8}$

$B = \boxed{11} \quad \boxed{15} \quad \boxed{16} \quad \boxed{18} \quad \boxed{21} \quad \boxed{23}$

- Galloping:
 - Falls A Min-Galloping gewonnen hat, suche das erste Element $A[i]$ in A das größer als $B[0]$ ist.
 - Kopiere $A[0]$ bis $A[i - 1]$ in das Merge-Array.
 - Suche nach $B[j]$ mittels galoppierender binärer Suche, d.h. vergleiche $B[0]$ mit $A[1], A[3], A[7], \dots, A[2^k - 1]$ bis $B[0]$ kleiner ist, dann binäre Suche.

Vorteile und Nachteile von Timsort

- Vorteile:
 - Extrem schnell auf Arrays, die bereits viel Struktur haben (Best-Case: $O(n)$).
 - Average- und Worst-Case $O(n \log n)$ wie Mergesort.
 - Im Durchschnitt weniger Objekt-Vergleiche als Quicksort.
 - Stabil.
- Nachteile:
 - Zusätzlicher Speicher: Average- und Worst-Case: $O(n)$.
 - (Optimierter Quicksort: $O(\log n)$)
 - Im Durchschnitt langsamer als ein optimierter Quicksort, wenn das Vergleichen von Elementen sehr effizient möglich ist.
- Kompromisslösung in Java: Timsort wird nur auf Objekt-Arrays verwendet. Für das Sortieren von Arrays mit primitiven Datentypen (z.B. int) ist eine spezielle Quicksort Variante als Sort in der Hilfsklasse Arrays vorgesehen.

Sortieren: Stabilität (Wiederholung)

- Definition: Ein Sortierverfahren ist **stabil**, wenn es die Reihenfolge der Datensätze, deren Sortierschlüssel gleich sind, bewahrt.

Beispiel: Liste von Personendaten mit Abteilungsnummer (innerhalb der Abteilung sortiert): 3 - Daniel, 4 - Maria, 2 - Paul, 3 - Erika, 1 - Anton, 3 - Sarah

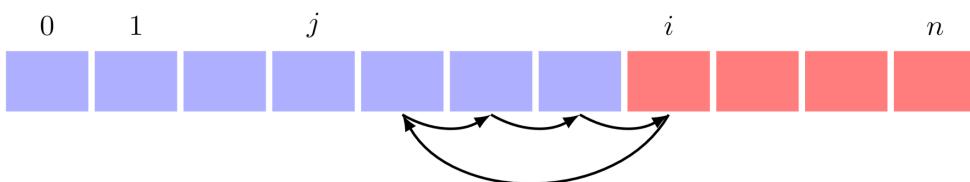
- Sortierung nach Abteilungsnummer mit stabilem Sortierverfahren:
1 - Anton, 2 - Paul, 3 - Daniel, 3 - Erika, 3 - Sarah, 4 - Maria
- Beispiel Sortierung nach Abteilungsnummer mit instabilem Sortierverfahren:
1 - Anton, 2 - Paul, 3 - Sarah, 3 - Daniel, 3 - Erika, 4 - Maria

Stabilität: Insertionsort

Insertionsort ist stabil.

Zur Erinnerung:

```
Insertionsort(A):
  for i ← 1 bis n - 1
    key ← A[i], j ← i - 1
    while j ≥ 0 und A[j] > key
      A[j + 1] ← A[j]
      j ← j - 1
    A[j + 1] ← key
```

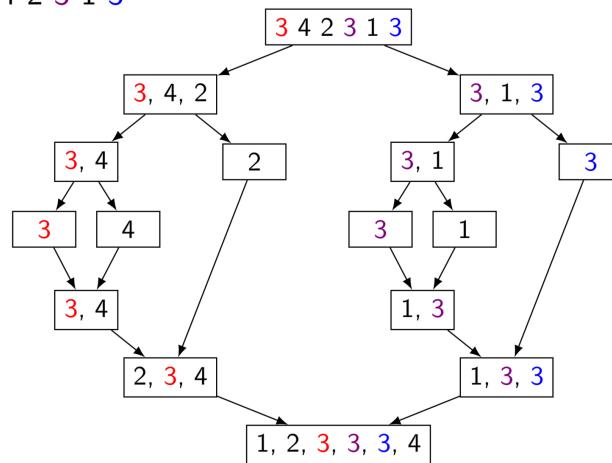


Stabilität: Insertionsort

Mergesort ist stabil.

Idee: Beim Aufteilen linke Hälfte nach links, rechte nach rechts, d.h. Reihenfolge wird nicht geändert. Beim Verschmelzen wird bei Gleichheit das linke Element genommen und daher die Reihenfolge auch nicht geändert.

Beispiel zur Illustration der Idee (kein Beweis) mit vereinfachter Ausgangssequenz 3 4 2 3 1 3



Stabilität Selectionsort

Selectionsort ist nicht stabil.

Beispiel mit vereinfachter Ausgangssequenz 3 4 2 3 1 3

3 4 2 3 1 3

1 4 2 3 3 3

1 2 4 3 3 3

1 2 3 4 3 3

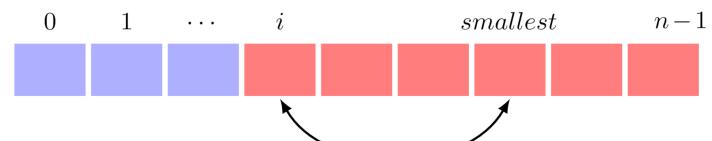
1 2 3 3 4 3

1 2 3 3 3 4

Zur Erinnerung:

Selectionsort(A):

```
for  $i \leftarrow 0$  bis  $n - 2$ 
     $smallest \leftarrow i$ 
    for  $j \leftarrow i + 1$  bis  $n - 1$ 
        if  $A[j] < A[smallest]$ 
             $smallest \leftarrow j$ 
    Vertausche  $A[i]$  mit  $A[smallest]$ 
```



Weitere Bibliotheken

Eclipse Collections

- <https://www.eclipse.org/collections/>
- Optimierte Sets und Maps, Immutable Collections, Collections für primitive Datentypen, Multimaps, Bimaps, Verschiedene Iterationsstile

Google Guava

- <https://github.com/google/guava>
- Unterstützt z.B. Multisets, Multimaps, Bimaps

Apache Commons Collections

- <https://commons.apache.org/proper/commons-collections/>

Brownies Collections

- <http://www.magicwerk.org/page-collections-overview.html>
- z.B. High Performance Listen (GapList, BigList)

JGraphT

- <https://github.com/jgrapht/jgrapht>
- Algorithmen und Datenstrukturen für Graphen