SS 2021

Algorithmen und Datenstrukturen

0. Kapitel

Vorbemerkungen

Martin Dietzfelbinger

April 2021

Team:

Folien und verantwortlich: Univ.-Prof. Dr. **Martin Dietzfelbinger**

Vorlesung & Übung & Praktikum:

M. Sc. **Philipp Schlag**

FG Komplexitätstheorie und Effiziente Algorithmen
`http://www.tu-ilmenau.de/ktea/`

Hörer/innen:

Informatikstudierende im 2. Semester.

Andere Interessierte *sehr* willkommen!

Material:

Material und Kommunikation über

`https://moodle2.tu-ilmenau.de/course/view.php?id=3530`

Datenstrukturen!

- Stacks, Queues
- Arrays
- Lineare Listen
- Suchbäume (implementieren Wörterbücher)
- Balancierte Suchbäume
- Hashtabellen (implementieren Wörterbücher)
- Heaps (implementieren Prioritätswarteschlangen)
- Union-Find-Datenstruktur
- Datenstrukturen für Graphen

Algorithmen!

- Sortieren: Mergesort, Quicksort, Heapsort
- Graphdurchläufe: Breitensuche, einfache Tiefensuche
- Volle Tiefensuche, starke Zusammenhangskomponenten
- Divide-and-Conquer: Multiplikation von ganzen Zahlen und von Matrizen, Auswahlproblem, Schnelle Fourier-Transformation (eventuell)
- Greedy-Ansatz: Kürzeste Wege, Huffman-Codes, Minimale Spannbäume, fraktionales Rucksackproblem
- Dynamische Programmierung: Kürzeste Wege, Editierdistanz, 0-1-Rucksackproblem

Algorithmen und Datenstrukturen: Ziele

Studierende sollen beherrschen:

- Grundlage für Analyse: Umgang mit **O-Notation**
- Eine **Spezifikation**stechnik für **Datentypen**
- Relevante **Implementierungen** fundamentaler **Datentypen**
- **Auswahlkriterien** für elementare **Datenstrukturen** kennen und anwenden
- Kenntnis grundlegender **Entwurfsstrategien** („Algorithmenparadigmen“)
- Kenntnis **grundlegender Algorithmen** und ihrer Eigenschaften
- **Auswahlkriterien** für **Algorithmen**, insbesondere Ressourcenverbrauch, kennen und anwenden
- Standard-**Analysemethoden** für Ressourcenverbrauch (Rechenzeit, Speicherplatz)
- **Korrektheitsbeweise** für Algorithmen verstehen und darstellen

Zur Arbeitsweise:

Stoff ist **viel** zu **kompliziert** und **umfangreich**, um durch reines Zuhören (bzw. einfaches Durchlesen) verstanden zu werden.

- Regelmäßig **Vorlesung nacharbeiten**. **Semesterbegleitend!**
- Begleitend Bücher ansehen.
- Übungsblätter drucken, lesen, zur Übung mitbringen, **vorher Lösung ausdenken**, Lösungsweg aufschreiben, an Lösungen **mitarbeiten**, Lösungen **vortragen**.
- Regelmäßig **Übungen nacharbeiten**. **Semesterbegleitend!**
- Praktikumsaufgaben **vorbereiten**. **Semesterbegleitend!**
- Bei Verständnisproblemen **frühzeitig fragen!**

Zeitaufwand? – Beispielrechnung

Leistungspunkte: 8 LP – Entspricht 240 Zeitstunden.

Vorlesungszeit: 15 Wochen.

12 Stunden pro Woche

Davon: **6** in Vorlesung/Übung/Praktikum

Zeitaufwand: **6–7** Zeitstunden pro Woche
neben Vorlesung/Übung!

ergibt: **180–195** Stunden.

Bleiben: **45–60 Stunden Prüfungsvorbereitung!**

Vorlesung/Übung

(Donnerstag 15:00 – 16:30 Audimax)
(Freitag 13:00 – 14:30 Audimax)

Vorlesung bis auf Weiteres ersetzt durch Online-Angebot.

(Dienstag 17:00 – 18:30 Sr HU 013)
(Mittwoch 9:00 – 10:30 Sr H 2509)
(Mittwoch 11:00 – 12:30 Sr H 1520a)

Übung vorläufig ersetzt durch Online-Angebot.

Übungsblätter stehen im Netz.
Übung **vor-** und **nach**bereiten!

Prüfungsstoff: Vorlesung + Übungsaufgaben.

Prüfungsklausur: 150 Minuten, **150 Punkte.**

Achtung: Angaben zu Klausuren
unter dem Vorbehalt der Durchführbarkeit im Juni bzw. Juli/August.

Praktikum

Verpflichtend nur für Informatikstudierende!

(Donnerstag, 7:00 – 8:30, gerade (G) Wochen, RTK 6)
(Donnerstag, 9:00 – 10:30, gerade (G) Wochen, RTK 6)
Präsenzangebot beginnt, sobald dies möglich ist.

Live-Programmierung in **C++:**
Implementierung und Zeitmessung (hauptsächlich) zu Algorithmen und Datenstrukturen der Vorlesung.
Praktikumsaufgaben/-anleitungen auf der Webseite.

Abschluss (Modulverpflichtung, ohne Note):
Präsentation ausgewählter Praktikumsaufgaben am Semesterende

Grundlagen, Voraussetzungen

- Prof. Schäfer, Algorithmen und Programmierung für (Ingenieur-)Informatiker, Vorlesung, WS 2019/20
  `https://www.tu-ilmenau.de/telematik/lehre/wintersemester-2019-2020/algorithmen-und-programmierung-fuer-ingenieur-informatiker/`
  (wird vorausgesetzt).
- G. Saake, K. U. Sattler. Algorithmen und Datenstrukturen – Eine Einführung mit Java. dpunkt.verlag, 5. Auflage, 2013 (Referenz für Grundlagen, Hintergrund).
- Prof. Kriesell, Grundlagen und Diskrete Strukturen, Vorlesung, WS 2020/21
  (wird vorausgesetzt).
- Dr. Schreyer, Mathematik für Informatiker 1, Vorlesung, WS 2020/21
  (wird vorausgesetzt).

Literaturvorschläge:

- R. H. **Güting** und S. **Dieker**, **Datenstrukturen und Algorithmen**, 4. Auflage (erweitert), SpringerVieweg 2018.
  Einführend, kompakt, für Bachelorstudium. Themen überlappen sich mit Vorlesung, sind aber *nicht deckungsgleich*. Gut zum „Reinkommen“.
  eBook, Universitätsbibliothek, sogar Download. Im Katalog nach PPN 1028030827 suchen.
- T. H. **Cormen**, C. E. **Leiserson**, R. L. **Rivest**, C. **Stein**,
  Introduction to Algorithms, 3rd ed., MIT Press, 2009.
  Reicht für das ganze Studium und darüber hinaus.
  Deutsche Ausgabe, 4. Auflage, 2013 bei Oldenbourg.
  Englisches eBook, Universitätsbibliothek: Im Katalog nach PPN 686807499 suchen.
- T. **Ottmann**, P. **Widmayer**, **Algorithmen und Datenstrukturen**,
  6. Aufl., SpringerVieweg, 2017. (Deutsches Standardwerk, sehr umfassend.)

Literaturvorschläge (Forts.):

- M. **Dietzfelbinger**, K. **Mehlhorn**, P. **Sanders**,
  Algorithmen und Datenstrukturen – Die Grundwerkzeuge, Springer-Verlag, 2014
  (Ist im Stil verschieden von der Vorlesung, Themen überschneiden sich, das Buch führt bei einigen Themen deutlich weiter; für Interessierte. Originalausgabe auf englisch von Mehlhorn und Sanders. Neues englisches Buch: Fortgeschrittener, hat auch parallele Algorithmen zum Thema.)
- R. **Sedgewick** and K. **Wayne**, **Algorithms**, 4th ed., Addison-Wesley, 2012. (Amerikanisches Standardwerk, umfassend, programmiernah. Wie Cormen et al.: reicht für das ganze Studium. Ressourcen im Web: Video-Lectures, Aufgaben, Programmcode. Deutsch: 4. Auflage, Pearson, 2014, ebenfalls mit Web-Ressourcen, insbesondere Programmcode.)
- J. **Kleinberg**, É. **Tardos**, **Algorithm Design**, Pearson Education, 2005 (weiterführend).

Vorlesung folgt eigenem Plan, nicht direkt einem Buch.

SS 2021

Algorithmen und Datenstrukturen

1. Kapitel

Einführung und Grundlagen

Martin Dietzfelbinger

April/Mai 2021

Algorithmen!

Beispiel: **Sortieren**.

Vorher: $(\binom{\texttt{Pam}}{f}, \binom{\texttt{Ida}}{f}, \binom{\texttt{Al}}{m}, \binom{\texttt{Liz}}{f}, \binom{\texttt{Lex}}{m}, \binom{\texttt{Tom}}{m}, \binom{\texttt{Sue}}{f}, \binom{\texttt{Ed}}{m})$

Nachher: $(\binom{\texttt{Al}}{m}, \binom{\texttt{Ed}}{m}, \binom{\texttt{Ida}}{f}, \binom{\texttt{Lex}}{m}, \binom{\texttt{Liz}}{f}, \binom{\texttt{Pam}}{f}, \binom{\texttt{Sue}}{f}, \binom{\texttt{Tom}}{m})$

Allgemeine **Aufgabe:** Für jede beliebige gegebene Folge

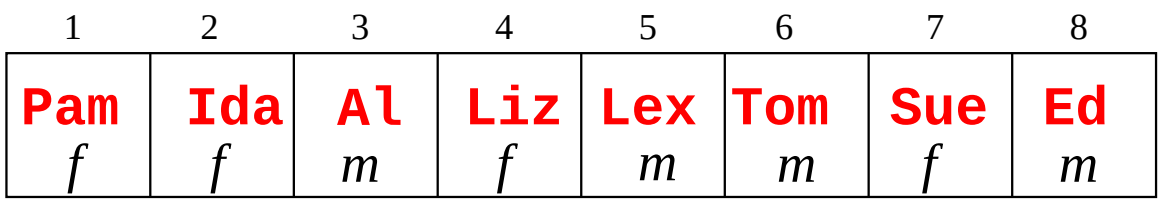
$$(a_1, a_2, \ldots, a_n)$$

von Objekten, die mit einem „Namen“ („Schlüssel“) aus einem total geordneten Bereich $(U, <)$ (Zahlen, Strings/Zeichenreihen, usw.) versehen sind,

sortiere $(a_1, a_2, \ldots, a_n)$, d. h. bringe die Objekte in aufsteigende Reihenfolge.

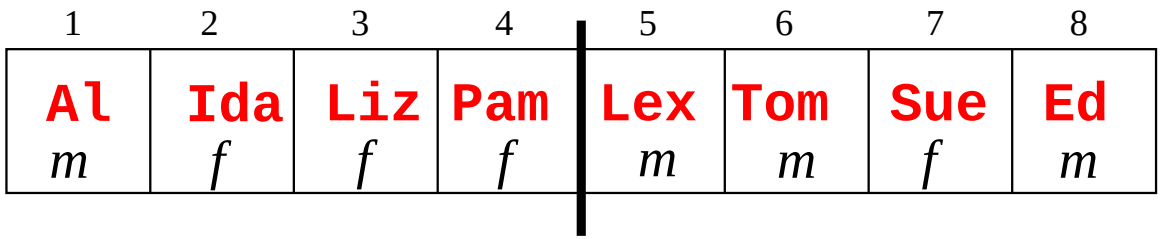
Notation: `x.key`: Schlüssel des Objekts `x`.

Sortieren durch Einfügen (Insertionsort)



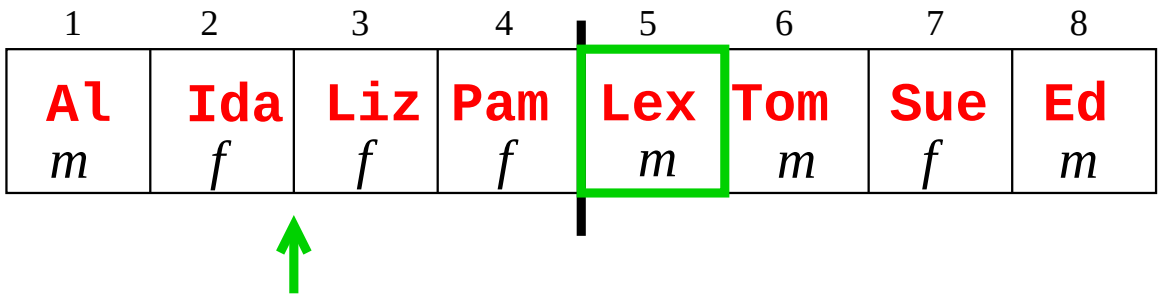
Zu Beginn, vor Runde 1.

Sortieren durch Einfügen (Insertionsort)



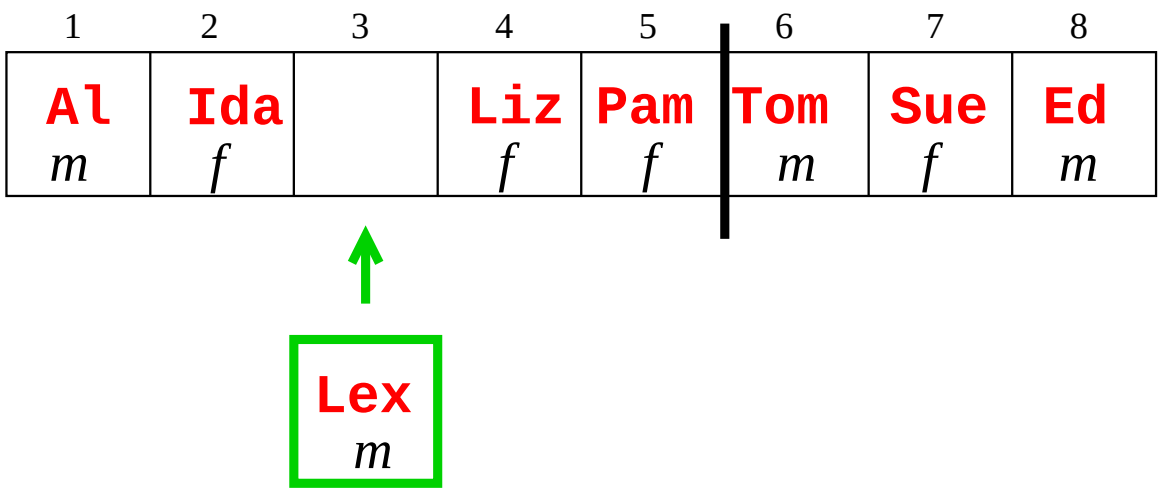
Vor Runde 5: `A[1..4]` sortiert.

Sortieren durch Einfügen (Insertionsort)



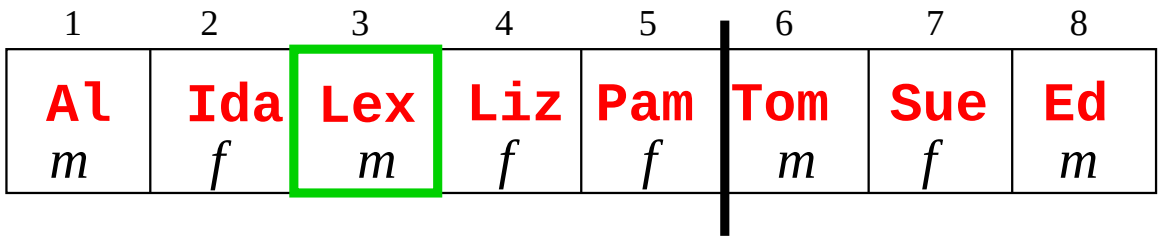
Runde 5: Bestimme Position von `A[5]` in `A[1..4]`.

Sortieren durch Einfügen (Insertionsort)



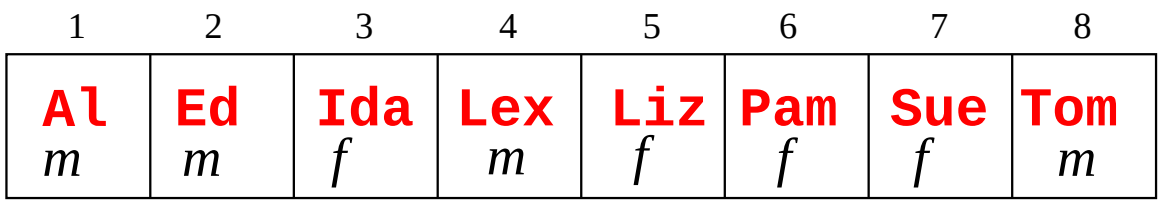
Runde 4: Entnehme `A[5]`, verschiebe `A[4]` und `A[3]`, um für `A[5]` Platz zu machen.

Sortieren durch Einfügen (Insertionsort)



Nach Runde 5: `A[1..5]` sortiert.

Sortieren durch Einfügen (Insertionsort)



Am Ende: `A[1..8]` sortiert.

Sortieren durch Einfügen (Insertionsort)

Pseudocode:

Algorithmus Insertionsort(`A[`$1..n$`]`**)**

```
(1)  for i from 2 to n do                  // Runden i = 2, ..., n
(2)    x ← A[i] ;                          // entnehme A[i]
(3)    j ← i ;
(4)    while j > 1 and* x.key < A[j-1].key do
(5)      A[j] ← A[j-1] ;                   // Einträge größer als A[i] werden einzeln
(6)      j ← j-1 ;                         // um 1 Position nach rechts verschoben
(7)    A[j] ← x ;                          // Einfügen
(8)  return A[1..n] .                      // Ausgabe
```

*: sequenzielles **and**

Was ist ein Algorithmus?

Erklärung:

Ein **Algorithmus** $\mathcal{A}$ ist eine durch einen endlichen Text gegebene Verarbeitungsvorschrift, die zu jeder gegebenen Eingabe x aus einer Menge $\mathcal{I}$ (der Menge der möglichen Eingaben) eindeutig eine Abfolge von auszuführenden Schritten („Berechnung“) angibt.
Wenn diese Berechnung nach endlich vielen Schritten anhält, wird ein Ergebnis $\mathcal{A}(x)$ aus einer Menge $\mathcal{O}$ (der Menge der möglichen Ausgaben) ausgegeben.

Beschreibungsformen: Umgangssprache, Pseudocode,
Programm in Programmiersprache, Programm für formales Maschinenmodell,
Programm in Maschinensprache, Turingmaschinenprogramm, . . .

Merkmale eines Algorithmus

- Endlicher Text
- **Ein** Algorithmus für eine **große** (oft: unendliche!) Menge $\mathcal{I}$ von Eingaben x

  (Beachte: **Zeitliche Trennung** der Formulierung des
  Algorithmus bzw. der Programmierung und der Anwendung auf eine Eingabe.)
- Eindeutig* vorgeschriebene Abfolge von Schritten,
  wenn Eingabe x vorliegt *(im Rahmen des benutzten Modells)
- **Wenn** Verarbeitung anhält: Ausgabe $\mathcal{A}(x) \in \mathcal{O}$ ablesbar

Nicht verlangt: Terminierung auf allen $x \in \mathcal{I}$.

Erweiterungen des Algorithmusbegriffs

- **Online-Algorithmen**: Die Eingabe wird nicht am Anfang der Berechnung, sondern nach und nach zur Verfügung gestellt, Ausgaben müssen nach Teileingabe erfolgen
  $\rightarrow$ Häufige Situation bei **Datenstrukturen**
  $\rightarrow$ Standard bei **Betriebssystemen**, **Systemprogrammen**, **Anwendungsprogrammen** mit **Benutzerdialog** usw.
- **Randomisierung**: Algorithmus führt **Zufallsexperimente** durch, keine Eindeutigkeit der Berechnung gegeben.
- Verteilte Algorithmen*: Mehrere Rechner arbeiten zusammen, kommunizieren, zeitliche Abfolge nicht vorhersagbar

* in dieser Vorlesung nicht oder nur am Rande diskutiert

Algorithmus Insertionsort (InsS)

```
(1)  for i from 2 to n do
(2)     x ← A[i] ;
(3)     j ← i ;
(4)     while j > 1 and x.key < A[j-1].key do
(5)        A[j] ← A[j-1] ;
(6)        j ← j-1 ;
(7)     A[j] ← x;
(8)  return A[1..n] .
```

Fragen:

(a) Ist dieser Algorithmus „**korrekt**“?

(b) **Wie lange** dauern Berechnungen mit diesem Algorithmus?

(a) Korrektheitsbeweise

Damit die Frage nach einem Beweis der Korrektheit überhaupt sinnvoll ist, müssen wir zunächst die zu lösende Aufgabe

spezifizieren.

Bemerkung: Aus einer **vagen Problemstellung** eine **präzise Spezifikation** des zu lösenden Problems zu gewinnen ist ein in der Praxis extrem wichtiger Schritt, dessen Bedeutung oft unterschätzt wird.

Berechnungsprobleme

Erklärung: Ein **Berechnungsproblem $\mathcal{P}$** besteht aus

(i) einer Menge $\mathcal{I}$ von möglichen *Eingaben* („*Inputs*“, „*Instanzen*“)
($\mathcal{I}$ kann unendlich sein!)

(ii) einer Menge $\mathcal{O}$ von möglichen *Ausgaben*

(iii) einer Funktion $f : \mathcal{I} \to \mathcal{O}$

Ein Algorithmus $\mathcal{A}$ **„löst $\mathcal{P}$“**, wenn $\mathcal{A}$ genau die Eingaben aus $\mathcal{I}$ verarbeitet und zu Eingabe $x \in \mathcal{I}$ Ausgabe $\mathcal{A}(x) = f(x)$ liefert.

Spezialfall $\mathcal{O} = \{1, 0\}$ „=“ $\{\mathit{true}, \mathit{false}\}$ „=“ $\{\text{Ja}, \text{Nein}\}$:
$\mathcal{P}$ heißt **Entscheidungsproblem**.

Varianten

- Zu einer Eingabe $x \in \mathcal{I}$ könnte es mehrere legale Ergebnisse geben.

  Formal: In (iii) haben wir statt einer Funktion f eine **Relation $R \subseteq \mathcal{I} \times \mathcal{O}$** mit $\forall x \in \mathcal{I}\, \exists y \in \mathcal{O}\colon R(x, y)$.

  $\mathcal{A}$ löst $\mathcal{P}$, wenn $\mathcal{A}$ genau die Eingaben aus $\mathcal{I}$ verarbeitet und $R(x, \mathcal{A}(x))$ für jedes $x \in \mathcal{I}$ gilt.

  Beispiel „nicht-stabiles Sortieren“: Beim Sortieren von Objekten/Datensätzen, wobei Schlüssel wiederholt vorkommen dürfen, kann *jede* sortierte Anordnung der Objekte eine legale Lösung sein. Dann ist die Lösung nicht eindeutig.

- Online-Situation: Eingabe wird „in Raten“ geliefert, Zwischenausgaben werden verlangt.

Spezifikation des Sortierproblems

Parameter: Die Grundbereiche (Klassen), aus denen die zu sortierenden Objekte und die Schlüssel stammen, sind Parameter der Spezifikation.

$(U, <)$: total geordnete Menge (endlich/unendlich, Elemente heißen „Schlüssel“).

Beispiele: Zahlenmengen $\{1, \ldots, n\}$, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$; Mengen von Strings.

D: Menge (Elemente heißen „Datensätze“).

$key : D \to U$ gibt zu jedem Datensatz einen „Sortierschlüssel“ an.

(i), (ii)

$$\mathcal{I} = \mathcal{O} = \mathbf{Seq}(D) := D^{<\infty} := \bigcup_{n \geq 0} D^n$$
$$= \{(a_1 \ldots, a_n) \mid n \in \mathbb{N}, a_1, \ldots, a_n \in D\},$$

(die Menge aller endlichen Folgen von Datensätzen)

Beispiel: $U = \mathbb{N}$, $D = \mathbb{N} \times \{\mathtt{a}, \ldots, \mathtt{z}\}$

$key\colon D \ni (i, b) \mapsto i \in U$

Ein möglicher Input:

$$x = ((5, \mathtt{r}), (3, \mathtt{s}), (11, \mathtt{t}), (6, \mathtt{e}), (9, \mathtt{r}), (5, \mathtt{t}), (4, \mathtt{o}), (5, \mathtt{i})) \in \mathsf{Seq}(D).$$

Das Ergebnis beim Sortieren sollte z. B. sein:

$$y = ((3, \mathtt{s}), (4, \mathtt{o}), (5, \mathtt{r}), (5, \mathtt{t}), (5, \mathtt{i}), (6, \mathtt{e}), (9, \mathtt{r}), (11, \mathtt{t})) \in \mathsf{Seq}(D).$$

(iii) Wir definieren eine geeignete Relation $R \subseteq \mathcal{I} \times \mathcal{O}$.

Vorher/Nachher dieselbe Objektmenge: Ausgabe ist **Permutation** der Eingabe.

$x = (a_1, \ldots, a_n) \in \mathcal{I}$ und $y = (b_1, \ldots, b_m) \in \mathcal{O}$ stehen in Relation R, d. h. $R(x, y)$ gilt

$:\Leftrightarrow$

$n = m$ und es gibt eine Permutation π von $\{1, \ldots, n\}$ mit

$b_i = a_{\pi(i)}$, $1 \leq i \leq n$, und $key(b_1) \leq key(b_2) \leq \cdots \leq key(b_n)$.

Beispiel:

Für $x = ((5, \mathtt{r}), (3, \mathtt{s}), (11, \mathtt{t}), (6, \mathtt{e}), (9, \mathtt{r}), (5, \mathtt{t}), (4, \mathtt{o}), (5, \mathtt{i}))$ und
$y = ((3, \mathtt{s}), (4, \mathtt{o}), (5, \mathtt{r}), (5, \mathtt{t}), (5, \mathtt{i}), (6, \mathtt{e}), (9, \mathtt{r}), (11, \mathtt{t}))$ ist z. B.

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 2 & 7 & 1 & 6 & 8 & 4 & 5 & 3 \end{pmatrix}$$

passend. ($\pi(2) = 7$: Ausgabepos. 2 $\widehat{=}$ Eingabepos. 7.)

Alternative Formulierung des Sortierproblems:

Die Ausgabe ist nicht die sortierte Folge, sondern die Permutation π, die die Eingabefolge in die richtige Reihenfolge bringt, wie oben beschrieben.

$$\mathcal{O}' = \{\pi \mid \pi \text{ Permutation einer Menge } \{1, \ldots, n\}, n \in \mathbb{N}\}$$

Definition:

$(a_1, \ldots, a_n)$ **wird durch π sortiert** $:\Leftrightarrow key(a_{\pi(1)}) \leq key(a_{\pi(2)}) \leq \cdots \leq key(a_{\pi(n)})$.

$x = (a_1, \ldots, a_n) \in \mathcal{I}$ und $\pi \in \mathcal{O}'$ **stehen in der Relation R'**, d. h.:

$R'(x, \pi)$ gilt $:\Leftrightarrow$ x wird durch π sortiert.

Dann ist $(\mathcal{I}, \mathcal{O}', R')$ ebenfalls eine Spezifikation des Sortierproblems.

Löst Insertionsort das Sortierproblem?

Gilt für jede Eingabe $(a_1, \ldots, a_n)$ in `A[1..n]`, dass nach Ausführung von Insertionsort dieselben Elemente im Array stehen, aber nach Schlüsseln aufsteigend sortiert?

Korrektheitsbeweis an *Beispiel*:

Durch **vollständige Induktion** über $i = 1, 2, \ldots, n$ zeigt man die folgende (Induktions-)Behauptung:

(IB_i)

> Nach Durchlauf Nummer i (in `i`) der äußeren Schleife (1)–(7) stehen in `A[1..i]` dieselben Elemente wie am Anfang, aber aufsteigend sortiert.

Nach Durchlauf Nummer n, also am Ende, gilt dann (IB_n). Diese Behauptung sagt gerade, dass die Einträge aus der ursprünglichen Eingabe nun in aufsteigender Reihenfolge in `A[1..n]` stehen.

Korrektheitsbeweis für Insertionsort

Zunächst erledigen wir zwei (triviale) Randfälle: Wenn $n = 0$ ist, also das Array leer ist, oder wenn $n = 1$ ist, es also nur einen Eintrag gibt, dann tut der Algorithmus gar nichts, und die Ausgabe ist korrekt.

Sei nun $n \geq 2$ beliebig, und der Input sei im Array `A[1..n]` gegeben.
Wir beweisen die Induktionsbehauptung (IB_i) durch Induktion über $i = 1, 2, \ldots, n$.

I.A.: $i = 1$: „Nach Durchlauf für $i = 1$“ ist vor Beginn.
Anfangs ist das Teilarray `A[1..1]` (nur ein Eintrag) sortiert, also gilt (IB_1).

I.V.: Wir nehmen an: $1 < i \leq n$ und (IB_{i-1}) gilt, d. h. `A[1..i − 1]` ist sortiert, enthält dieselben Objekte wie am Anfang, und Schleifendurchlauf i beginnt.

I.-Schritt: Betrachte Schleifendurchlauf i (Zeilen (2)–(7)).
$x :=$ `A[`i`]` ist das aktuelle Objekt; es wird in Variable `x` kopiert.
Position `A[`i`]` kann man sich jetzt als leer vorstellen.

Sei $m = m_i$ die größte Zahl in $\{0, \ldots, i-1\}$, so dass $\boxed{key(\texttt{A[}j\texttt{]}) \leq key(x) \text{ für } 1 \leq j \leq m}$.

(Wenn es gar kein $j \leq i-1$ mit $key(\texttt{A}[j]) \leq key(x)$ gibt, ist $m = 0$.
Wenn für alle $1 \leq j \leq i-1$ die Ungleichung erfüllt ist, ist $m = i-1$.)

Weil nach I.V. $\texttt{A}[1..i-1]$ sortiert ist, gilt: $\boxed{key(x) < key(\texttt{A}[j]) \text{ für } m+1 \leq j \leq i-1}$.
Die innere Schleife (4)–(7) mit der Initialisierung in Zeile (3) testet, durch direkten Vergleich in Zeile (4), die Zahlen $j-1 = i-1, i-2, i-3, \ldots, 0$ nacheinander darauf, ob $j-1 = m$ ist. Entweder wird dabei $j = 1$ erreicht, dann ist $m = 0 = j-1$, oder man stellt für ein $j \geq 2$ fest, dass $key(\texttt{A}[j-1]) \leq key(x)$ gilt. Dann ist offenbar ebenfalls $m = j-1$.

Schon während der Suche nach m werden (in Zeile (5)) die Einträge
$\texttt{A}[i-1]$,$\texttt{A}[i-2]$,. . . ,$\texttt{A}[m+1]$, in dieser Reihenfolge, um eine Position nach rechts verschoben, ohne die Sortierung dieser Elemente zu ändern.
Nun stehen $\texttt{A}[1], \ldots, \texttt{A}[m]$ an ihrer ursprünglichen Position, $\texttt{A}[m+1]$,$\texttt{A}[m+2]$,. . . ,$\texttt{A}[i-1]$ (vorher) stehen jetzt in $\texttt{A}[m+2..i]$.
In Zeile (7) wird x (aus x) an die Stelle $\texttt{A}[m+1]$ geschrieben. Nach den Feststellungen in den Boxen und aus der I.V. ergibt sich, dass nun $\texttt{A}[1..i]$ aufsteigend sortiert ist. Also gilt (IB_i).

Nach Durchlaufen der äußeren Schleife Schleife für $i = 2, \ldots, n$ gilt die Aussage (IB_n), also ist nach Abschluss des Ablaufs das Array sortiert.

Methoden für Korrektheitsbeweise

- **Vollständige Induktion** (wie soeben gesehen)
  Damit wird die Korrektheit von **Schleifen** bewiesen.
  Meist ist dabei die Formulierung einer geeigneten **Induktionsbehauptung** zentral, die dann **„Schleifeninvariante“** genannt wird.
- **Verallgemeinerte Induktion** (d.h. man schließt von (IB_j) *für alle* $j < i$ auf (IB_i)).
  Damit werden **rekursive Prozeduren** behandelt. (Siehe: Divide-and-Conquer.)
- **Diskrete Mathematik**: Ad-hoc-Beweise.
  (Beispiele werden wir bei Graphalgorithmen sehen.)

PAUSE

(b) Laufzeitanalysen

Sprechweise: Einen Algorithmus **analysieren** heißt, Kosten, Aufwand, „Rechenzeit“ (synonym!) zu bestimmen oder zumindest abzuschätzen, die sich einstellt, wenn man eine Implementierung des Algorithmus $\mathcal{A}$ auf Eingabe x ablaufen lässt.

Natürlich geht es schneller, kurze Arrays zu sortieren als lange. Daher ist ein zentraler Parameter einer solchen Analyse die

Größe der Eingabe $x \in \mathcal{I}$, bezeichnet mit $|x|$ oder $\mathbf{size}(x)$ (oft, nicht immer: n).

Die Bedeutung von $\mathrm{size}(x)$ ist abhängig vom Problem $\mathcal{P}$.
Beispiele: Beim Sortierproblem ist $\mathrm{size}((a_1, \ldots, a_n)) = n$.
Wenn $x \in \mathcal{I}$ ein String $a_1 \ldots a_n$ ist: $\mathrm{size}(x) = n = \#(\text{Buchstaben in } x)$.
Wenn Eingabe $G = (V, E)$ ein Graph mit $n = |V|$ Knoten (Ecken) und m Kanten ist:
Oft $\mathrm{size}(G) = n + m$.

$$\mathcal{I}_n := \{x \in \mathcal{I} \mid \mathrm{size}(x) = n\} \text{ – „Größenklasse“.}$$

Kosten – Rechenzeiten

Kosten für Elementaroperationen in Programm:

Addition, Vergleich, Test, Sprung, Zuweisung,
Auswertung eines Attributs, Zeiger-/Indexauswertung

Jede Elementaroperation op ist mit einer Konstanten c_{op} bewertet, abhängig von **Programmiersprache**, **Compiler**, **Prozessor**, **Hauptspeichertyp**, **Cachestruktur**, . . .

Idee: op benötigt **höchstens** „Zeit“ c_{op} (Takte oder ns oder μs oder . . .)

Unabhängig vom konkreten Input x.

Die Konstanten c_{op} werden nie explizit genannt;
vereinfachend oft sogar: $\boldsymbol{c_{\text{op}} = 1}$ („1 Maschinenschritt“, „1 Operation“).

Kosten – Rechenzeiten

Wir schreiben

$$O(1)$$

für: „ein Wert, nicht größer als eine (von x unabhängige) Konstante c“
und wollen Laufzeit des Algorithmus anhand von Pseudocode o. ä. analysieren.
Technische Besonderheiten von konkreten Rechnern
(Cachearchitekturen, Pipelining, Mehrkernprozessoren, Coprozessoren, . . .),
Implementierungen, Compileroptimierungen, usw.
zu berücksichtigen wäre viel zu komplex!

Wir definieren als Kosten/Rechenzeit von Algorithmus $\mathcal{A}$ auf Eingabe x:

$t_{\mathcal{A}}(x) := \#$(Operationen, die $\mathcal{A}$ auf Input x ausführt). (# bedeutet „Anzahl“)

Beispiel: Insertionsort auf Input $(a_1, \ldots, a_n)$, gegeben in `A[1..n]`

Zeilen (2)–(3) und (7): Werden $(n-1)$-mal ausgeführt.
Es gibt n Schleifenende-Tests (Zeile (1)) $\Rightarrow$

Kosten von Zeilen (1)–(3), (7): $n \cdot \boldsymbol{O}(1) =$ „$\boldsymbol{O}(n)$“ (genaue Erklärung später).

Zeilen(4)–(6) für ein festes i (in `i`): Entscheidend: Anzahl der Tests in (4).

Der Test „`j` > 1“ wird zwischen 1-mal und i-mal durchgeführt,
der Test „`x.key` $<$ `A[j-1].key`“ zwischen 1-mal und $(i-1)$-mal, **inputabhängig**.

Setze* $\boldsymbol{k_i} := k_i(x) := \#\{l \mid 1 \leq l \leq i-1 \text{ und } key(a_i) < key(a_l)\}$ („verkehrt herum“).
(Nicht schwer zu sehen: $k_i = i - 1 - m_i$ mit m_i aus dem Korrektheitsbeweis von Insertionsort.)

Kosten für Zeilen (4)–(6) bei Schleifendurchlauf für i: $(k_i+1) \cdot \boldsymbol{O}(1) =$ „$\boldsymbol{O}(k_i+1)$“.

---

* **Notation:** $\#A$ oder $|A|$: Kardinalität von A.

Kosten insgesamt: Summiere über $2 \leq i \leq n$.

$$\boldsymbol{O}(n) + (k_2 + 1) \cdot \boldsymbol{O}(1) + (k_3 + 1) \cdot \boldsymbol{O}(1) + \cdots + (k_n + 1) \cdot \boldsymbol{O}(1) =$$

$$= \boldsymbol{O}(n + \sum_{2 \leq i \leq n} k_i).$$

(„Ausklammern des O“ und unkonventionelle Benutzung von „$=$“ wird später gerechtfertigt.)

Typisch für solche Analysen: Nicht jede Operation zählen, sondern Zurückziehen auf das Zählen von **„charakteristischen Operationen“**, die den Rest bis auf einen konstanten Faktor dominieren.

Hier: Vergleiche zwischen Schlüsseln (Zeile (4)), Anzahl $\leq k_i + 1$.

(Oder: Zuweisungen von Elementen (Zeilen (5) und (7)), Anzahl exakt $k_i + 1$.)

Definition:

$$\boldsymbol{F_x} := \sum_{2 \leq i \leq n} k_i$$
$$= \#\{(l,i) \mid 1 \leq l < i \leq n, key(a_l) > key(a_i)\}$$

Anzahl der **„Fehlstände“** in $x = (a_1, \ldots, a_n)$

Z. B. gibt es in $x = (4, 7, 5, 3)$ vier Fehlstände,
nämlich $(1, 4), (2, 3), (2, 4), (3, 4)$.

Gesamtaufwand von Insertionsort auf Eingabe x der Länge n ist „$\boldsymbol{O(n + F_x)}$“.

„Worst case“ (Schlechtester Fall):

Wegen $F_x \leq \sum_{2 \leq i \leq n}(i-1) = \sum_{j=1}^{n-1} j \overset{\text{(Mathem.)}}{=} \frac{n(n-1)}{2} < \frac{n^2}{2}$ gilt:

Gesamtaufwand/-rechenzeit/-kosten für Insertionsort auf Inputs mit n Komponenten beträgt „$\boldsymbol{O}(n + n^2/2)$“ = „$\boldsymbol{O}(n^2)$“.

Es können auch solch große Kosten auftreten:
$k_i = i - 1$ für alle i gilt für absteigend sortierte Eingaben, z. B.
$x = ((11, \mathtt{a}), (9, \mathtt{q}), (8, \mathtt{p}), (7, \mathtt{a}), (5, \mathtt{z}), (4, \mathtt{u}), (2, \mathtt{q}))$.

Für solche Inputs sind die Kosten mindestens $n + \frac{1}{2}n(n-1) > \frac{1}{2}n^2$.

Wir schreiben: Worst-case-Kosten sind „$\boldsymbol{\Theta}(n^2)$“. (Details zur Notation später).

Worst-case-Kosten, allgemein

Sei $\mathcal{A}$ Algorithmus für Inputs aus $\mathcal{I}$. Wir hatten: $\mathcal{I}_n = \{x \in \mathcal{I} \mid \text{size}(x) = n\}$.

Wir definieren **worst-case-Kosten** oder **Kosten im schlechtesten Fall** auf Inputs der Größe n:

$$\boldsymbol{T_{\mathcal{A}}(n)} := \max\{t_{\mathcal{A}}(x) \mid x \in \mathcal{I}_n\}.$$

Wenn man betonen will, dass es um die worst-case-Kosten geht: $\boldsymbol{T_{\mathcal{A},\text{worst}}(n)}$.

Im *Beispiel* Insertionsort haben wir: $T_{\text{IS,worst}}(n) =$ „$\Theta(n^2)$“.

Best-case-Kosten, allgemein

Wir definieren **best-case-Kosten** oder **Kosten im besten Fall**:

$$T_{\mathcal{A},\text{best}}(n) := \min\{t_{\mathcal{A}}(x) \mid x \in \mathcal{I}_n\}.$$

Beispiel Insertionsort:
$k_i = 0$ für alle i gilt für aufsteigend sortierte Eingabe x, z. B.
$x = ((2, \mathtt{q}), (4, \mathtt{u}), (5, \mathtt{z}), (7, \mathtt{a}), (8, \mathtt{p}), (9, \mathtt{q}), (11, \mathtt{a}))$.
Für solche Inputs ist $F_x = 0$, also $n + F_x = n + 0 = n$.
Für jeden Input gilt $n + \sum_{2 \le i \le n} k_i \ge n$.
Gesamtaufwand für Insertionsort im besten Fall ist daher: $T_{\text{IS,best}}(n) = \boldsymbol{O}(n + 0) = \boldsymbol{O}(n)$.

„Average case“ (Mittlerer Fall):

Wir betrachten wieder das Beispiel Insertionsort, mit Eingabe $X = (a_1, \ldots, a_n)$.

Vereinfachende Annahme: Die Schlüssel $key(a_i)$, $1 \leq i \leq n$, sind verschieden.

Annahme: Jede der $n!$ möglichen Anordnungen der n Objekte in der Eingabe $(a_1, \ldots, a_n)$ tritt mit derselben Wahrscheinlichkeit $1/n!$ auf.

(Vgl.: Laplace'sches Indifferenzprinzip, `https://de.wikipedia.org/wiki/Indifferenzprinzip`)

Beispiel: Schlüssel $2, 3, 7$:
Anordnungen $(2,3,7), (2,7,3), (3,2,7), (3,7,2), (7,2,3), (7,3,2)$, jede mit W.-keit $\frac{1}{6}$

Bilde **Mittelwert/Erwartungswert** für $F_x = \sum_{2 \leq i \leq n} k_i$ und berechne:

$$\mathbf{E}(F_x) \overset{(*)}{=} \frac{1}{2} \cdot \binom{n}{2} = \frac{1}{2} \cdot \frac{n(n-1)}{2} = \frac{1}{4} n(n-1) < \frac{n^2}{4}.$$

Zu $(*)$: Wir betrachten der Einfachheit halber Inputs, die aus den Zahlen $1, \ldots, n$ bestehen. Dann ist jeder mögliche Input x einfach eine Permutation $(a_1, \ldots, a_n)$ von $\{1, \ldots, n\}$, und $\mathcal{I}_n$ ist die Menge aller dieser Inputs. Es gibt $n!$ viele davon. Nun betrachten wir die Menge P_n aller Paare $(x, (l, i))$, für x einen solchen Input und $1 \leq l < i \leq n$. Wir haben offenbar $|P_n| = n! \cdot \binom{n}{2}$. Uns interessiert

$$P_n^{\times} := \{(x, (l, i)) \in P_n \mid (l, i) \text{ ist Fehlstand in } x\}.$$

Zu Input $x = (a_1, \ldots, a_n)$ und Paar (l, i) betrachten wir den Input $x^{(l,i)}$, der sich durch Vertauschen von a_l und a_i in x ergibt. (*Beispiel*: $(5, \underline{2}, 3, 1, \underline{4})^{(2,5)}$ ist $(5, \underline{4}, 3, 1, \underline{2})$.)
Dann ist folgendes offensichtlich: Wenn wir in $x^{(l,i)}$ wieder Positionen l und i vertauschen, erhalten wir x zurück. Und: Von Inputs x und $x^{(l,i)}$ hat genau einer bei (l, i) einen Fehlstand. Das heißt, dass es in P_n genau $\frac{1}{2}|P_n| = \frac{1}{2}n! \cdot \binom{n}{2}$ viele Paare $(x, (l, i))$ gibt, für die x bei (l, i) einen Fehlstand hat. Kurz: $\left|P_n^{\times}\right| = \frac{1}{2}n! \cdot \binom{n}{2}$.

Also:

$$\sum_{x \in \mathcal{I}_n} F_x = \sum_{x \in \mathcal{I}_n} \sum_{\substack{1 \leq l < i \leq n \\ (l,i) \text{ ist Fehlstand in } x}} 1 \quad = \left|P_n^{\times}\right| = \frac{1}{2}n! \cdot \binom{n}{2}.$$

Durch Mittelung erhalten wir die mittlere/erwartete Anzahl von Fehlständen in einem $x \in \mathcal{I}_n$ als

$$\mathbf{E}(F_x) = \frac{1}{n!} \cdot \frac{1}{2} n! \cdot \binom{n}{2} = \frac{1}{2}\binom{n}{2} = \frac{n(n-1)}{4}.$$

Also ist der Aufwand von Insertionsort **im mittleren Fall**, auf Inputs der Länge n: $\boldsymbol{O}(n^2)$, aber auch $\boldsymbol{\Omega}(n^2)$. Zusammengefasst: $\boldsymbol{\Theta}(n^2)$. (Notation wird später erklärt.)

Average-case-Kosten, allgemein

Average-case-Kosten oder **durchschnittliche Kosten**:

Gegeben sei, für jedes $n \in \mathbb{N}$, eine Wahrscheinlichkeitsverteilung $\mathbf{Pr}_n$ auf $\mathcal{I}_n$.

D.h.: Für jedes $x \in \mathcal{I}_n$ ist eine Wahrscheinlichkeit $\mathbf{Pr}_n(x)$ festgelegt.

(Woher kommt $\mathbf{Pr}_n$? Aus empirischer Beobachtung, oder per Festlegung, z. B. auf der Basis des Laplace'schen Indifferenzprinzips.) Dann definieren wir:

$$T_{\mathcal{A},\mathsf{av}}(n) := \mathbf{E}(t_{\mathcal{A}}) = \sum_{x \in \mathcal{I}_n} \mathbf{Pr}_n(x) \cdot t_{\mathcal{A}}(x).$$

(**E**: Erwartungswert, berechnet mit $\mathbf{Pr}_n$; „Mittelwert".)

Beispiel Insertionsort: $T_{\mathsf{IS},\mathsf{av}}(n) = \text{„}\mathbf{\Theta}(n + \frac{1}{4}n(n-1))\text{“} = \text{„}\mathbf{\Theta}(n^2)\text{“}$.

Analyse von Kosten/Rechenzeit

Feststellung:

Selbst für feste Implementierung eines Algorithmus $\mathcal{A}$ auf fester Maschine und feste

„Eingabegröße“ $n = \text{size}(x) = |x|$

kann der Berechnungsaufwand auf verschiedenen Inputs sehr unterschiedlich sein.

D. h. der Begriff „**Die** Laufzeit von Algorithmus $\mathcal{A}$ auf Eingaben x vom Umfang $n = \text{size}(x)$“

ist oft sinnlos.

Sinnvoll: **Schlechtesten** und besten Fall, eventuell **durchschnittlichen** Fall betrachten.

Analyse von Kosten/Rechenzeit

In allen drei Situationen (worst case, best case, average case):

Konstante Faktoren ignorieren!

Technisch:

Obere Schranken (d. h. „Kosten sind höchstens . . . “) $\rightarrow$ „O-Notation“

Untere Schranken (d. h. „Kosten sind mindestens . . . “) $\rightarrow$ „Ω-Notation“ (s. unten)

Asymptotisches Verhalten, **präzise**
(d. h. „Kosten sind bis auf konstanten Faktor gleich . . . “) $\rightarrow$ „Θ-Notation“

Ende 1. Vorlesung

Größenordnungen von Funktionen

Notation:

$\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ (bei Bedarf: $\mathbb{N}^+ = \{n \in \mathbb{N} \mid n > 0\} = \{1, 2, 3, \ldots\}$)

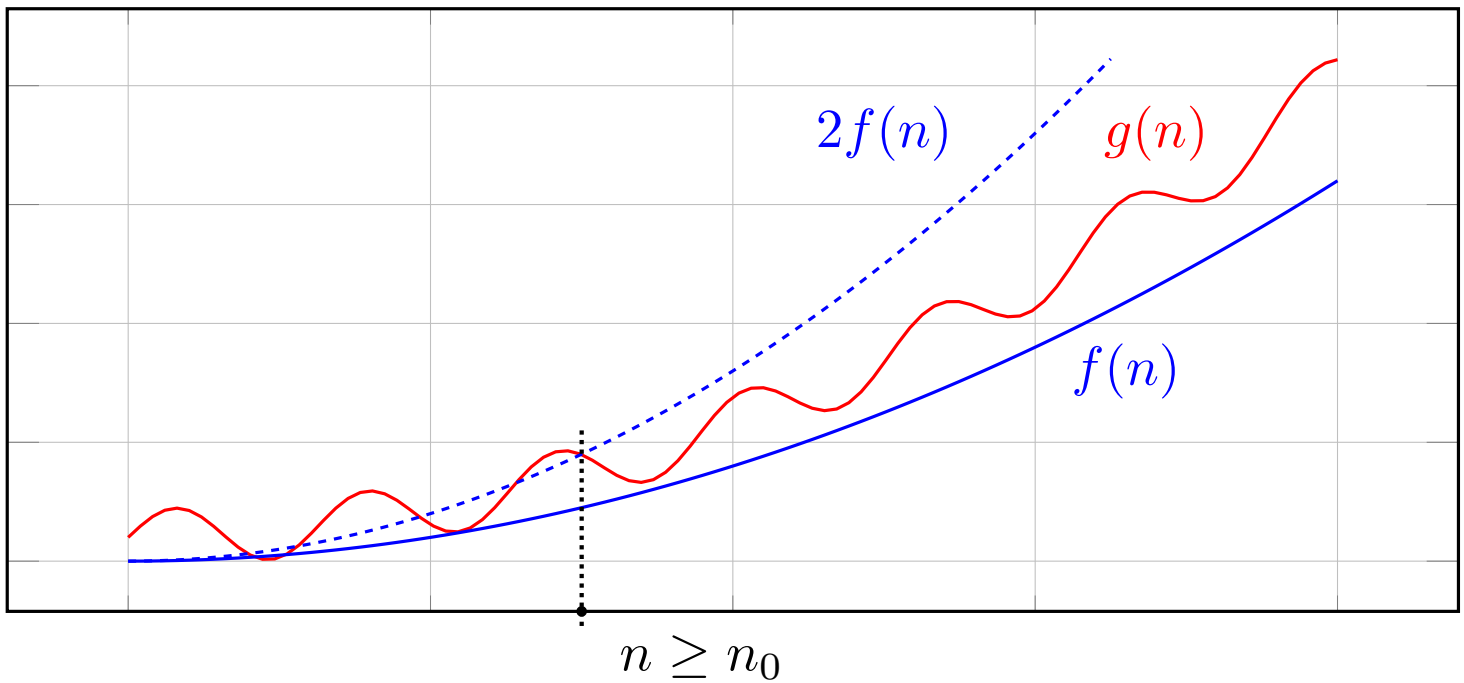
$\mathbb{R}_0^+ = \{x \in \mathbb{R} \mid x \geq 0\}$ und $\mathbb{R}^+ = \{x \in \mathbb{R} \mid x > 0\}$

Wir betrachten Funktionen $f : \mathbb{N} \to \mathbb{R}$. Das Argument $n \in \mathbb{N}$ ist oft die Eingabegröße, $f(n)$ schätzt eine Rechenzeit oder eine Wahrscheinlichkeit ab. Uns interessieren praktisch nur Funktionen, die für alle genügend großen Argumente n positiv oder nichtnegativ sind. – Abkürzungen:

$$\mathcal{F}^+ := \{f \mid f : \mathbb{N} \to \mathbb{R}, \exists n_0 \forall n \geq n_0 : f(n) > 0\}.$$

$$\mathcal{F}_0^+ := \{f \mid f : \mathbb{N} \to \mathbb{R}, \exists n_0 \forall n \geq n_0 : f(n) \geq 0\}.$$

„g **wächst asymptotisch höchstens so schnell wie** f“ soll heißen:
Es gibt eine Konstante $c > 0$, so dass für alle genügend großen n gilt: $g(n) \leq c \cdot f(n)$.



Horizontale Achse: n in $\mathbb{N}$. – Im Bild ist $g(n) \leq c \cdot f(n)$ für $n \geq n_0$ und $c = 2$.

$O(f)$: Funktionenmenge

Wir fassen alle Funktionen g, die asymptotisch höchstens so schnell wie f wachsen, in einer Menge zusammen.

Definition Für $f \in \mathcal{F}^+$ sei

$$\boldsymbol{O(f)} := \{\, g \in \mathcal{F}_0^+ \mid \exists n_0 \in \mathbb{N}\ \ \exists c > 0\ \ \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}.$$

Beispiel: Für $g(n) = 2n + 7\log n$ und $f(n) = n$ gilt $g \in O(f)$.
Wir wählen dazu $c = 4$ und $n_0 = 16$. Für $n \geq n_0$ gilt (weil $\frac{\log n}{n}$ monoton fällt):

$$\begin{aligned} g(n) = 2n + 7\log n &= 2n + \tfrac{7\log n}{n} \cdot n \leq 2n + \tfrac{7\log n_0}{n_0} \cdot n \\ &= 2n + \tfrac{7\cdot 4}{16} \cdot n < 2n + 2n = 4n = c \cdot f(n). \end{aligned}$$

$O(f)$: Funktionenmenge

Beispiele: $g \in O(f)$ gilt für folgende Paare (f, g):

(a) $g(n) = 2n + 7 \log n$ und $f(n) = n$.

(b) $g(n) = 2n + 7 \log n$ und $f(n) = n^2$.

(c) $g(n) = 4n^3 - 13n^2 + 10$ und $f(n) = n^3 \log n$.

(d) $g(n) = 3 + 2 \sin n$ und $f(n) = 1$.

(e) $g(n) = 2n$ für gerade n und $g(n) = 2/n$ für ungerade n, und $f(n) = n$.

Beispiel (b) zeigt, dass es nicht verboten ist, eine „viel zu große“ Funktion $f(n)$ zu verwenden. Beispiel (d) zeigt, dass die konstante Funktion $f(n) = 1$ für $n \in \mathbb{N}$ auch zugelassen ist.

Übung: Finde für Fälle (b)–(e) passende Zahlen c und n_0. (Müssen nicht so klein wie möglich sein.)

Notation „$\cdots = O(f(n))$“

Schreib- und Sprechweisen:

(i) „$g(n) = O(f(n))$“ hat exakt dieselbe Bedeutung wie „$g \in O(f)$“.

(ii) Man spricht: „$g(n)$ **ist (in) Groß-Oh von** $f(n)$“. Achtung: **Nicht** „. . . ist gleich . . . “

(!!) „$=$“ bedeutet **nicht Gleichheit**, die Relation ist nicht symmetrisch.

(iii) Spezialfall: „$g(n) = O(1)$“. (1 steht für f mit $f(n) = 1$ für $n \in \mathbb{N}$.)

Hilfreiche Intuition:

$O(f(n))$ steht für **irgendein Element** von $O(f)$. (Eine anonyme Funktion $g \in \mathcal{F}_0^+$, von der man **nur** weiß, dass sie für genügend große n durch $c \cdot f(n)$ beschränkt ist.)

Damit ist z. B. auch Folgendes sinnvoll: $3n^3 + 2n^2 - 5n + 4 = 3n^3 + O(n^2)$.

Notation „$\cdots = O(f(n))$“

Beispiele:

(a) $2n + 7 \log n = O(n)$.

(b) Für jede Funktion g mit $0 \leq g(n) \leq 4n^2 / \log n$ für alle $n \geq 10$ gilt: $g(n) = O(n^2)$.

(c) $2n^2 - 30n + 3 = O(n^2)$.

(d) $4 \cdot 2^n + 3 \cdot n^2 = 4 \cdot 2^n + O(n^2) = O(2^n)$.

(e) $4n^2 \log n + 2n(\log n)^3 = 4n^2 \log n + O(n^2) = O(n^3)$.

(f) $4n^2 \log n + 2n(\log n)^3 = O(n^2 \log n)$.

(g) $3 + 2 \sin n = O(1)$.

Übung: Finde für Fälle (b)–(f) passende Zahlen c und n_0.

Verwendung der Notation „$g(n) = O(f(n))$“ im Kontext der Algorithmenanalyse:

$g(n) = T_{\mathcal{A}}(n)$ für einen Algorithmus $\mathcal{A}$. Oft sind solche Funktionen in ihren Details gar nicht zu ermitteln. Daher sucht man eine Abschätzung nach oben in der Form $T_{\mathcal{A}}(n) = O(f(n))$. Angenehm dabei: Man muss sich nicht um die konstanten Faktoren kümmern, also auch nicht um die Faktoren, die möglicherweise in der Definition von $T_{\mathcal{A}}(n)$ vorkommen.
Die Funktion $f(n)$ sollte einerseits eine einfache Gestalt haben, andererseits auch $T_{\mathcal{A}}(n)$ nicht unnötig überschätzen.

Konvention:

Innerhalb des $O(\ldots)$-Terms verwendet man möglichst **kleine** und/oder **möglichst einfache** Funktionen.

$17n^2/(\log n)^3 + 13n^{3/2} = O(n^2/(\log n)^3)$ (klein) oder

$17n^2/(\log n)^3 + 13n^{3/2} = O(n^2)$ (besonders einfach)

O-Notation ist auch für Funktionen definiert, die für große n eher klein sind: Konstante Funktionen, sogar Funktionen, die gegen 0 gehen.

Verwendung: Abschätzung von „kleinen“ Fehlertermen (absolut oder relativ), Abschätzung von Wahrscheinlichkeiten.

Beispiele:

(a) $\sum_{i=0}^{n} c^i < \frac{1}{1-c}$ (geometrische Reihe), also $\sum_{i=0}^{n} c^i = O(1)$, falls $0 < c < 1$ fest.

(b) $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \ln n + O(1)$. (Die n-te „harmonische Zahl“.)

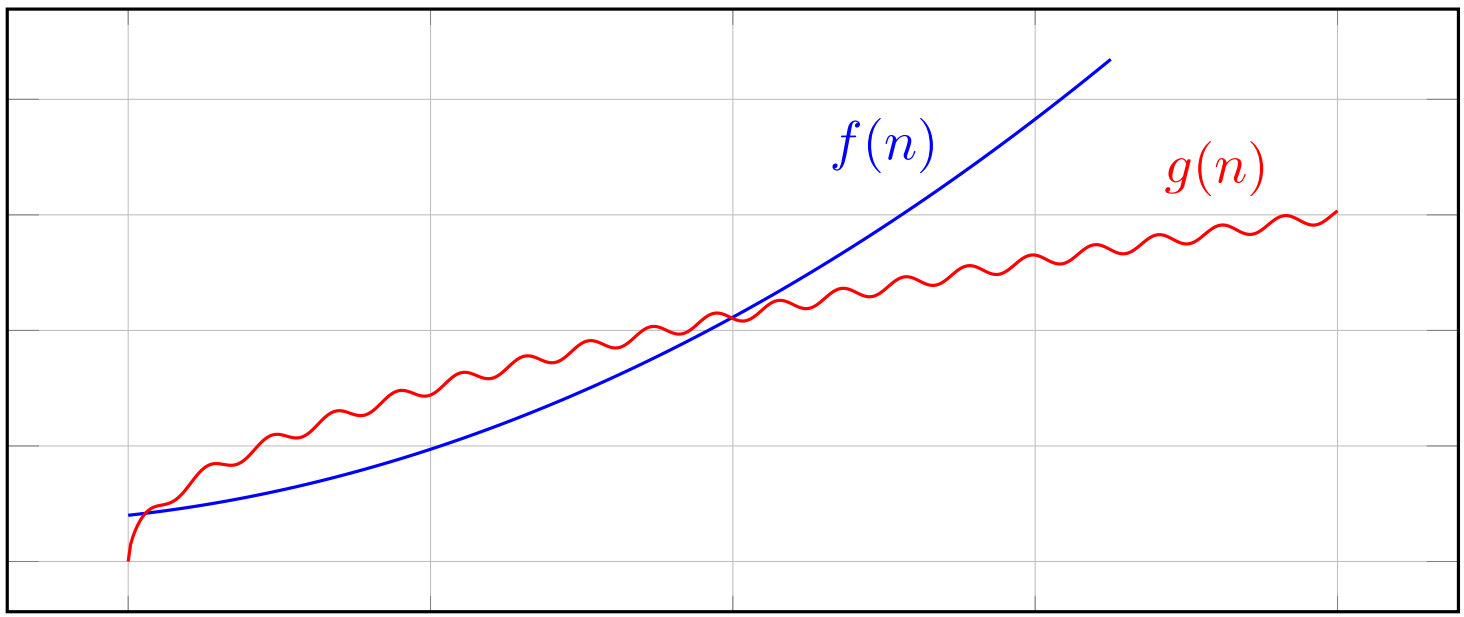
(c) $\binom{n}{2}^{-1} = \frac{2}{n(n-1)} = O(1/n^2)$.

(d) $2/n^2 + 3/n^3 = O(1/n^2)$.

(e) $2^{-n} + 3^{-n} = O(2^{-n})$.

„$o(f)$“ und „$\cdots = o(f(n))$“

„g **wächst asymptotisch strikt langsamer als** f“



Horizontale Achse: n aus $\mathbb{N}$.

„$o(f)$“ und „$\cdots = o(f(n))$“

Definition Für $f \in \mathcal{F}^+$ sei

$$o(f) := \left\{ g \in \mathcal{F}_0^+ \;\middle|\; \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0 \right\}$$

Schreibe „$g(n) = o(f(n))$“ für $g \in o(f)$. (Sprich: „$g(n)$ **ist (in) klein-oh von** $f(n)$“.)

Beispiele:

(a) $n/\log n = o(n)$.

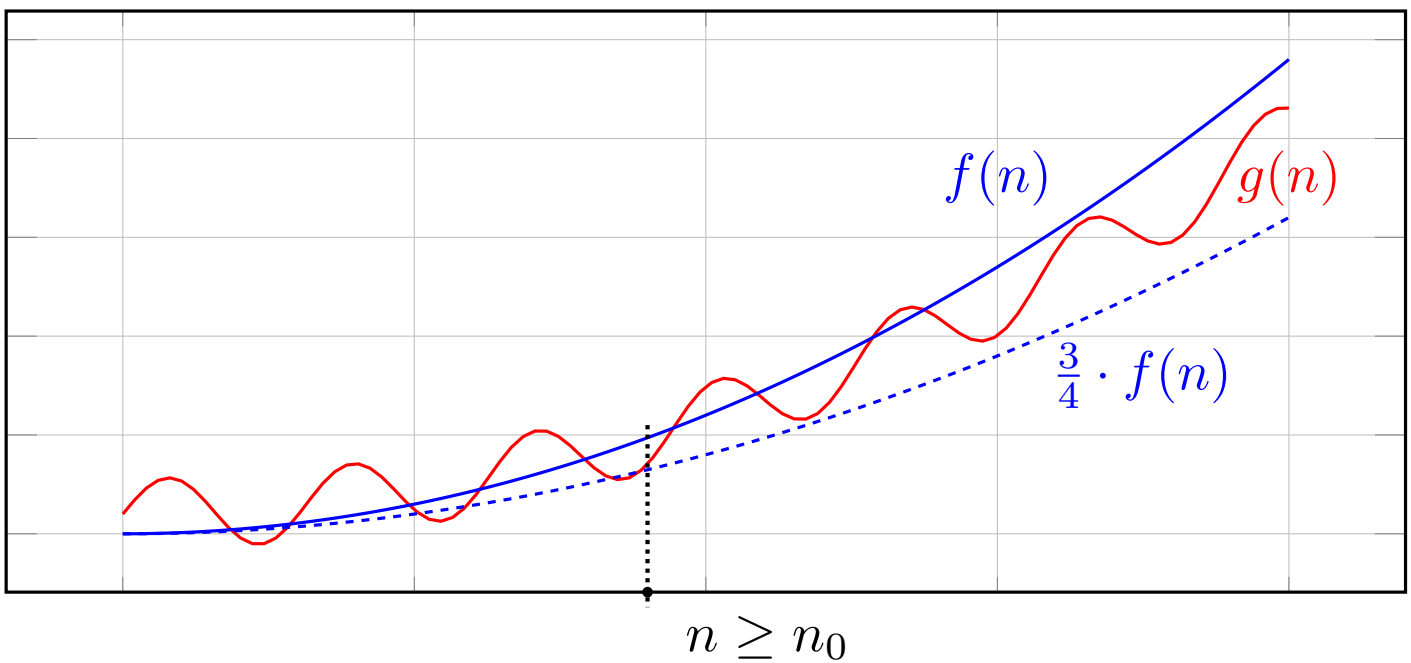
(b) $5n^2 \log n + 7n^{2.1} = o(n^{5/2})$.

(c) $n^{10} = o(2^n)$.

(d) $1/(n \log n) = o(1/n)$.

„$\Omega(f)$“ und „$\cdots = \Omega(f(n))$“

„g **wächst asymptotisch mindestens so schnell wie** f“ heißt: Es gibt eine Konstante $d > 0$, so dass $g(n) \geq d \cdot f(n)$ gilt, für genügend große n. – Im Bild: $g(n) \geq d \cdot f(n)$ für $d = \frac{3}{4}$, $n \geq n_0$.



„$\Omega(f)$“ und „$\cdots = \Omega(f(n))$“

Definition Für $f \in \mathcal{F}^+$ sei

$$\Omega(f) := \{g \in \mathcal{F}^+ \mid \exists n_0 \in \mathbb{N}\, \exists d > 0\, \forall n \geq n_0 \colon g(n) \geq d \cdot f(n)\}$$

Schreibe „$g(n) = \Omega(f(n))$“ für $g \in \Omega(f)$. (Sprich: „$g(n)$ **ist Groß-Omega von** $f(n)$“.)

Beispiele:

(a) $0.3n \log n - 20n = \Omega(n \log n)$.

(b) $\frac{1}{10}n^2 - 20n + 1 = \Omega(n^2)$.

(c) $(1 - 1/n)^n = \Omega(1)$.

(d) $\frac{1}{3n^2} - \frac{5}{n^3} = \Omega(1/n^2)$.

Nur manchmal wird benutzt: Für $f \in \mathcal{F}^+$:

$\boldsymbol{\omega(f)} := \{g \in \mathcal{F}^+ \mid \lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty\}$

„$g(n)$ **wächst asymptotisch echt schneller als** $f(n)$.“

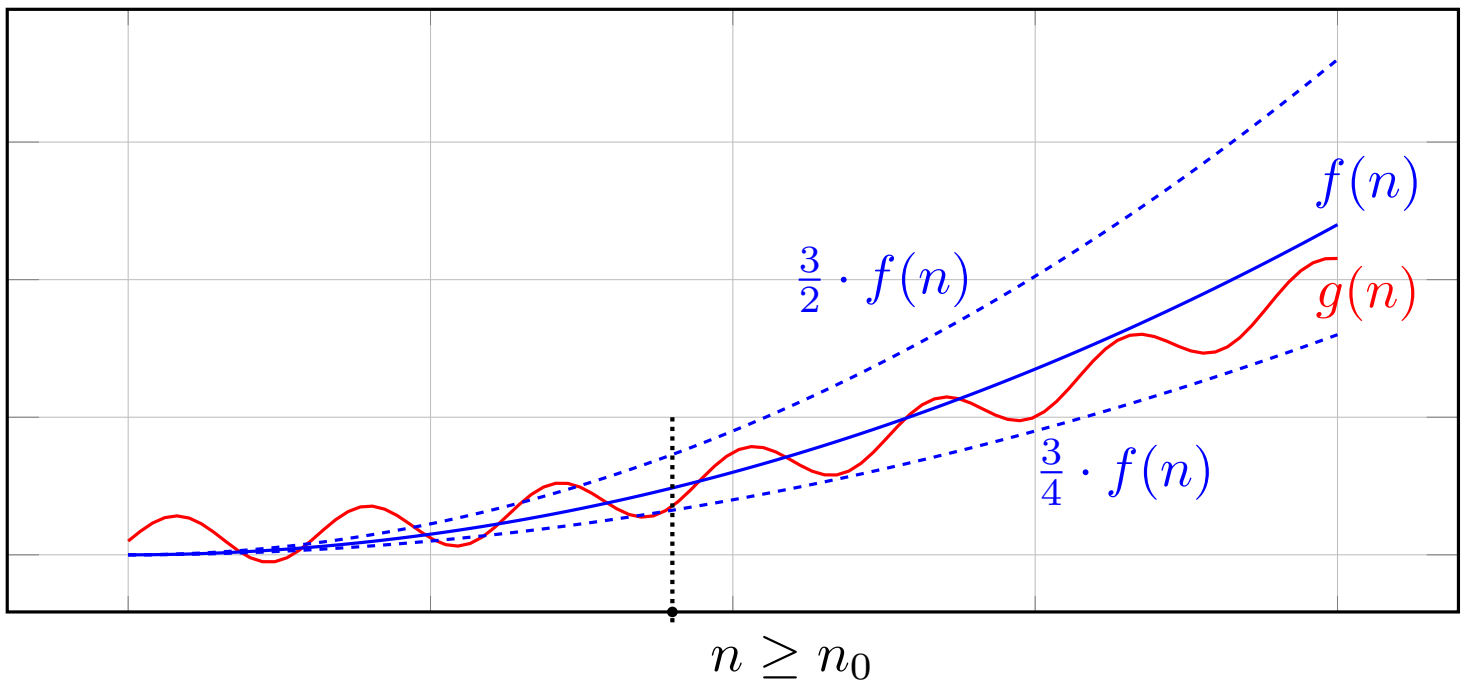
Schreibe „$\boldsymbol{g(n) = \omega(f(n))}$“ für $g \in \omega(f)$.

(Sprich: „$g(n)$ ist **klein-omega von** $f(n)$“.)

Klar: Für $f, g \in \mathcal{F}^+$ gilt: $g \in \omega(f) \Leftrightarrow f \in o(g)$.

„$\Theta(f)$“ und „$\cdots = \Theta(f(n))$“

„g **ist asymptotisch von derselben Größenordnung wie** f“.



„$\Theta(f)$“ und „$\cdots = \Theta(f(n))$“

Für $f \in \mathcal{F}^+$ definiere:

$\Theta(f) := O(f) \cap \Omega(f)$, d.h.

$\Theta(f) = \{g \in \mathcal{F}^+ \mid \exists n_0 \in \mathbb{N}\ \exists c, d > 0\ \forall n \geq n_0 \colon d \cdot f(n) \leq g(n) \leq c \cdot f(n)\}$

Beobachtung: $g \in \Theta(f) \Leftrightarrow O(g) = O(f)$.

Schreibe „$g(n) = \Theta(f(n))$“ für $g \in \Theta(f)$. (Sprich: „$g(n)$ **ist Theta von** $f(n)$“.)

Beispiele: (a) $\binom{n}{2} = \Theta(n^2)$.

(b) $n \log_2 n = \Theta(n \log_{10} n) = \Theta(n \ln n)$.

(c) $(1 + 1/n)^n = \Theta(1)$.

Nur manchmal benutzt: Für $f \in \mathcal{F}^+$ definiere:

$\boldsymbol{\omega(f)} := \{g \in \mathcal{F}^+ \mid \lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty\}$

Schreibe „$\boldsymbol{g(n) = \omega(f(n))}$“ für $g \in \omega(f)$. (Sprich: „$g(n)$ ist **klein-omega von** $f(n)$“.)

Bedeutung: „$g(n)$ **wächst asymptotisch echt schneller als** $f(n)$.“

Klar: Für $f, g \in \mathcal{F}^+$ gilt: $g \in \omega(f) \Leftrightarrow f \in o(g)$.

PAUSE

Rechenregeln

Lemma 1 (Grenzwertregel)

Wenn $f \in \mathcal{F}^+$, $g \in \mathcal{F}_0^+$ und $\lim_{n\to\infty} \frac{g(n)}{f(n)} = c$ für ein $c > 0$, dann ist $g(n) = \Theta(f(n))$.

Beweis von Lemma 1: Aus $\lim_{n\to\infty} \frac{g(n)}{f(n)} = c$ folgt, dass es ein n_0 gibt, so dass für alle $n \geq n_0$ gilt: $c/2 \leq \frac{g(n)}{f(n)} \leq 2c$.

Also: $\frac{1}{2}f(n) \leq g(n) \leq 2f(n)$ für $n \geq n_0$, also $g(n) = \Omega(f(n))$ und $g(n) = O(f(n))$.

Beispiel: $g(n) = 2n + 7\log n$ und $f(n) = n$.
Es ist $\lim_{n\to\infty} \frac{2n+7\log n}{n} = \lim_{n\to\infty}(2 + \frac{7\log n}{n}) = 2$, also $2n + 7\log n = \Theta(n)$.

Beachte auch den etwas anders gelagerten Fall $c = 0$:

$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$ bedeutet: $g(n) = o(f(n))$.

(Daraus folgt $g(n) = O(f(n))$, aber es gilt **nicht** $g(n) = \Theta(f(n))$).

[Variante für Spezialist/inn/en: Wenn $\sup_{n \in \mathbb{N}} \frac{g(n)}{f(n)} < \infty$, dann folgt $g(n) = O(f(n))$.]

Rechenregeln

Wichtiger Spezialfall: Wenn g ein **Polynom** ist:

$$g(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$$

für $a_k, \ldots, a_1, a_0 \in \mathbb{R}$, wobei $a_k > 0$, und $f(n) = n^k$, dann

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \left(a_k + \frac{a_{k-1}}{n} + \cdots + \frac{a_1}{n^{k-1}} + \frac{a_0}{n^k} \right) = a_k,$$

also gilt $g(n) = \Theta(n^k)$.

Beispiel: Man erhält „durch bloßes Hinschauen“: $20n^5 - 30n^3 - 210n^2 + 15 = \Theta(n^5)$.

Rechenregeln

Lemma 2 (Transitivität)

Es seien $f, g \in \mathcal{F}^+$. Dann gilt für jedes $h \in \mathcal{F}_0^+$:

Wenn $h(n) = O(g(n))$ und $g(n) = O(f(n))$, dann gilt $h(n) = O(f(n))$.

D.h.: Wenn $g \in O(f)$, dann ist $O(g) \subseteq O(f)$.

Kurz: $h(n) = O(g(n)) = O(f(n))$; man kann schrittweise abschätzen und vereinfachen.

Achtung: In der $O(\cdot)$-Notation bedeutet „$=$“ keine Gleichheit,
die Relation ist **nicht symmetrisch**. Ihr Verhalten ist eher dem „$\leq$“ ähnlich!

Rechenregeln

Beweis von Lemma 2:
Wähle $c_1 > 0$ und n_1 so groß, dass gilt: $\forall n \geq n_1 : g(n) \leq c_1 \cdot f(n)$.
Wähle $c_2 > 0$ und n_2 so groß, dass gilt: $\forall n \geq n_2 : h(n) \leq c_2 \cdot g(n)$.
Setze $n_0 := \max\{n_1, n_2\}$.
Dann gilt für $n \geq n_0$: $h(n) \leq c_2 g(n) \leq c_2(c_1 f(n)) = (c_1 c_2) f(n)$.
Also: $h(n) = O(f(n))$.

Rechenregeln

Lemma 3 („Konstante Faktoren weglassen“)

Ist $g(n) = O(d \cdot f(n))$ und $d > 0$ beliebig, dann gilt $g(n) = O(f(n))$.

D.h.: Wenn $d > 0$, dann ist $O(d \cdot f) = O(f)$ (für die Funktionenmengen).

Analoges gilt bei $o(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$, $\omega(\cdot)$.

Beispiele:

- Nicht ~~$g(n) = O(3n \log_{2.8} n)$~~ schreiben, sondern $g(n) = O(n \log n)$;
- nicht ~~$g(n) = O(n/3)$~~ schreiben, sondern $g(n) = O(n)$;
- nicht ~~$g(n) = O(2.5)$~~ schreiben, sondern $g(n) = O(1)$.

Achtung: Bei 2^n und 3^n kommt es auf die unterschiedliche Basis an!
($2^n = o(3^n)$, also ist 3^n **nicht** in $O(2^n)$, und $O(2^n) \neq O(3^n)$.)

Rechenregeln

Beweis von Lemma 3:
Wähle $c_0 > 0$ und n_0 so groß, dass gilt: $\forall n \geq n_0 : g(n) \leq c_0 \cdot f(n)$.
Dann gilt für alle $n \geq 0$ auch $g(n) \leq (c_0/d) \cdot (d \cdot f(n))$,
d. h. $g(n) = O(d \cdot f(n))$.

Rechenregeln

Lemma 4 (Linearität und Maximumsregel)

Ist $g_1(n) = O(f_1(n))$ und $g_2(n) = O(f_2(n))$, so folgt

$$g_1(n) + g_2(n) = O(f_1(n) + f_2(n)) = O(\max\{f_1(n), f_2(n)\}).$$

Hintereinanderausführung von Algo'en: Rechenzeiten addieren, auch in O-Notation.

Kosten teurerer Teile dominieren die Kosten billigerer Teile. Analog bei $o(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$, $\omega(\cdot)$.

Beispiele: $10n \log n + 5n^2 = O[\Theta, \Omega](n \log n + n^2) = O[\Theta, \Omega](n^2)$, und
$5n/\log n + 10\sqrt{n} = o(n) + o(n) = o(n)$.

Rechenregeln

Beweis von Lemma 4:

Für $i = 1, 2$: Wähle $c_i > 0$ und n_i so groß, dass gilt: $\forall n \geq n_i : g_i(n) \leq c_i \cdot f_i(n)$.

Setze $n_0 := \max\{n_1, n_2\}$ und $c_0 := \max\{c_1, c_2\}$. Dann gilt für $n \geq n_0$:

$g_1(n) + g_2(n) \leq c_1 f_1(n) + c_2 f_2(n) \leq c_0(f_1(n) + f_2(n))$,
also $g_1(n) + g_2(n) = O(f_1(n) + f_2(n))$.

Zudem für $n \geq n_0$: $f_1(n) + f_2(n) \leq 2 \max\{f_1(n), f_2(n)\}$.

Also $g_1(n) + g_2(n) \leq 2c_0 \max\{f_1(n), f_2(n)\}$.

Rechenregeln

Lemma 5 (Weglassen von Termen „kleinerer Ordnung“)

Ist $f \in \mathcal{F}^+$ und $f(n) = \Theta(h(n))$ und $|g(n)| = o(f(n))$, so ist $f(n)+g(n) = \Theta(h(n))$.

Analog bei $O(\cdot)$, $\Omega(\cdot)$, $o(\cdot)$, $\omega(\cdot)$.

Fazit: In Summen suche den „dominierenden Term“; lasse Rest weg.

Beispiele:

$5n^2 - 10n \log n = \Theta(n^2)$.
$n^2/\log n - 10n = o(n^2)$ und $n^2/\log n - 10n = \omega(n)$.
$1/(2n) - 10/n^2 = \Theta(1/n)$.

Rechenregeln

Beweis von Lemma 5:

Wähle $C > c > 0$ und n_1 so groß, dass gilt: $\forall n \geq n_1 : c \cdot h(n) \leq f(n) \leq C \cdot h(n)$.

Wähle n_2 so groß, dass gilt: $\forall n \geq n_2 : |g(n)| \leq \frac{1}{2} \cdot f(n)$.

Setze $n_0 := \max\{n_1, n_2\}$.

Dann gilt für $n \geq n_0$: $\frac{c}{2} \cdot h(n) \leq \frac{1}{2} \cdot f(n) \leq f(n) + g(n) \leq \frac{3}{2} \cdot f(n) \leq \frac{3C}{2} \cdot h(n)$.

Rechenregeln

Lemma 6 (Multiplikationsregel)

Ist $g_1(n) = O(f_1(n))$ und $g_2(n) = O(f_2(n))$, so folgt $g_1(n)g_2(n) = O(f_1(n)f_2(n))$.

Analog bei $o(\cdot)$, $\Omega(\cdot)$, $\omega(\cdot)$, $\Theta(\cdot)$.

(Bei $o(\cdot)$ genügt als Voraussetzung: $g_1(n) = O(f_1(n))$ und $g_2(n) = o(f_2(n))$.)

Anwendung bei Algorithmenanalyse: Sei $g_1(n) = O(f_1(n))$ und $g_2(n) = O(f_2(n))$.

Wird eine **Schleife** $g_1(n)$-mal ausgeführt, mit $g_1(n) = O(f_1(n))$,
wobei die Kosten für die einmalige Ausführung höchstens $g_2(n) = O(f_2(n))$ betragen,
dann sind die Gesamtkosten $O(f_1(n)f_2(n))$.

Rechenregeln

Beweis von Lemma 6:

Für $i = 1, 2$: Wähle $c_i > 0$ und n_i so groß, dass gilt: $\forall n \geq n_i : g_i(n) \leq c_i \cdot f_i(n)$.

Setze $n_0 := \max\{n_1, n_2\}$ und $c_0 = c_1 c_2$. Dann gilt für $n \geq n_0$:

$g_1(n)g_2(n) \leq (c_1 \cdot f_1(n))(c_2 \cdot f_2(n)) \leq c_0(f_1(n)f_2(n))$,
also $g_1(n)g_2(n) = O(f_1(n)f_2(n))$.

Nun betrachte den Fall $g_1(n) = O(f_1(n))$ und $g_2(n) = o(f_2(n))$.
Das heißt: $g_1(n) \leq c_1 \cdot f_1(n)$ für $n \geq n_1$ und $\lim_{n\to\infty} \frac{g_2(n)}{f_2(n)} = 0$.
Dann haben wir: $\frac{g_1(n)g_2(n)}{f_1(n)f_2(n)} \leq c_1 \frac{g_2(n)}{f_2(n)} \to 0$, also $g_1(n)g_2(n) = o(f_1(n)f_2(n))$.

Rechenregeln

O-Notation in **Summen**

Beispiel: Wir wollen die Rechenzeit einer Schleife analysieren, wobei der Schleifenrumpf bei verschiedenen Durchläufen unterschiedlich lange braucht, etwa:

Input der Größe n erzeugt $\leq n$ Durchläufe, Kosten beim i-ten Durchlauf: $a_{n,i} \leq \frac{10n}{i}$.

Maximum der Kosten: $\leq 10n/1 = O(n)$ im ersten Durchlauf.

Wenn wir einheitlich $a_{n,i} = O(n)$ abschätzen, liefert die Multiplikationsregel Gesamtkosten $O(n) \cdot O(n) = O(n \cdot n) = O(n^2)$.

Aber: $\frac{n}{1} + \frac{n}{2} + \cdots + \frac{n}{n} = n \cdot (\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n}) = nH_n$, für $H_n = n$-te harmonische Zahl.

Wir werden später sehen: $H_n \leq 1 + \ln n = O(\log n)$.

Daher Vermutung: Gesamtkosten sind nur $O(n \log n)$.

Rechenregeln

Lemma 7 (O-Notation in Summen)

Sei $g\colon \mathbb{N} \times \mathbb{N} \to \mathbb{R}_0^+$. Für jedes $n \in \mathbb{N}$ sei $a_{n,1}, \ldots, a_{n,t(n)}$ Zahlenfolge mit

$$\exists \boldsymbol{c} > 0\ \exists n_0\ \forall n \geq n_0\ \forall 1 \leq i \leq t(n):\ 0 \leq a_{n,i} \leq \boldsymbol{c} \cdot g(n,i).$$

(„$a_{n,i}$ ist **uniform** durch ein Vielfaches von $g(n,i)$ beschränkt“; kurz: $a_{n,i} = O(g(n,i))$.)

Dann gilt (die Variable ist n):

$$S(n) := \sum_{1 \leq i \leq t(n)} a_{n,i} = O\Big(\sum_{1 \leq i \leq t(n)} g(n,i) \Big).$$

Beispiel: Im oben diskutierten Beispiel hat bei Eingabegröße n der i-te Schleifendurchlauf Kosten $a_{n,i} \leq 10n/i$, für $1 \leq i \leq t(n) = n$. Wir wählen $c = 10$, $n_0 = 1$ und $g(n,i) = n/i$. Dann sind nach der Summationsformel die Gesamtkosten

$$\sum_{1 \leq i \leq t(n)} a_{n,i} = O\Big(\sum_{1 \leq i \leq n} \frac{n}{i} \Big) = O\Big(n \cdot \Big(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \Big) \Big) = O(n \log n).$$

Achtung! Wenn es für jedes $n \geq n_0$ ein (eigenes) c_n gibt mit

$$\forall 1 \leq i \leq t(n) : 0 \leq a_{n,i} \leq c_n \cdot g(n,i),$$

($a_{n,i}$ **nicht-uniform** beschränkt),dann kann man **nicht** folgern, dass gilt:

$$\sum_{1 \leq i \leq t(n)} a_{n,i} = O\Big(\sum_{1 \leq i \leq t(n)} g(n,i) \Big).$$

PAUSE

Verwendung der O-Notation bei der Analyse eines Algorithmus

Beispiel Insertionsort, Eingabe $x = (a_1, \ldots, a_n)$, Größe n:

Mit Multiplikationsregel: Im Durchlauf für i hat die Schleife in Zeilen (4)–(6) Kosten $\Theta(1) + k_i \cdot \Theta(1) = \Theta(1 + k_i)$ (wobei $k_i = \#\{(l, i) \mid 1 \leq l < i \leq n, key(a_l) > key(a_i)\}$).

Mit Additionsregel: Zeilen (1)–(7) für den Durchlauf für i haben Kosten $\Theta(1) + \Theta(1 + k_i) + \Theta(1) = \Theta(1 + k_i)$.

Mit Summationsregel: Der gesamte Ablauf hat Kosten

$$t_{\mathsf{IS}}(x) = \sum_{2 \leq i \leq n} \Theta(1 + k_i) = \Theta(n + \sum_{\mathbf{2 \leq i \leq n}} \boldsymbol{k_i}) = \Theta(n + \boldsymbol{F_x}) = O(n^2).$$

$\boldsymbol{F_x}$: Anzahl der „Fehlstände“ (l, i), wo $l < i$ und $key(a_l) > key(a_i)$. Wissen schon: $F_x \leq \binom{n}{2}$.

Ignorieren konstanter Faktoren: Bedeutung

Beispiel:

Ein anderer Sortieralgorithmus ist **Mergesort** (siehe AuP-Vorlesung).

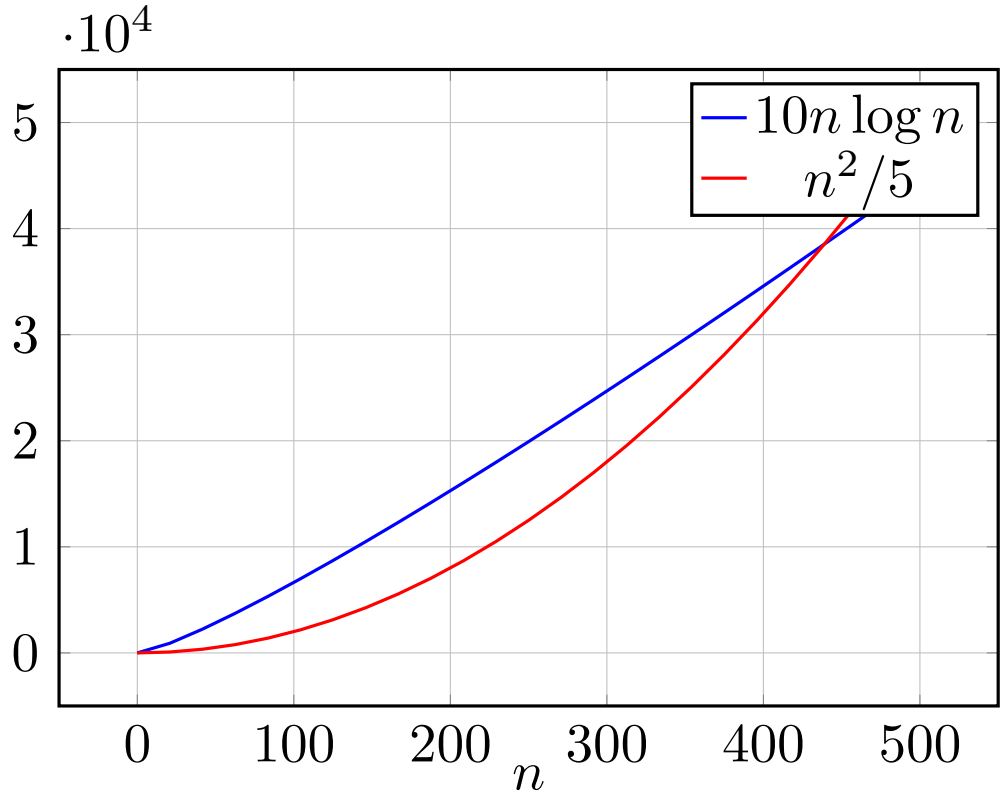
Rechenzeit/Kosten: $T_{\text{Mergesort}}(n) = \Theta(n \log n)$.

Auch wenn wir die Konstanten nicht kennen, können wir Folgendes sagen:

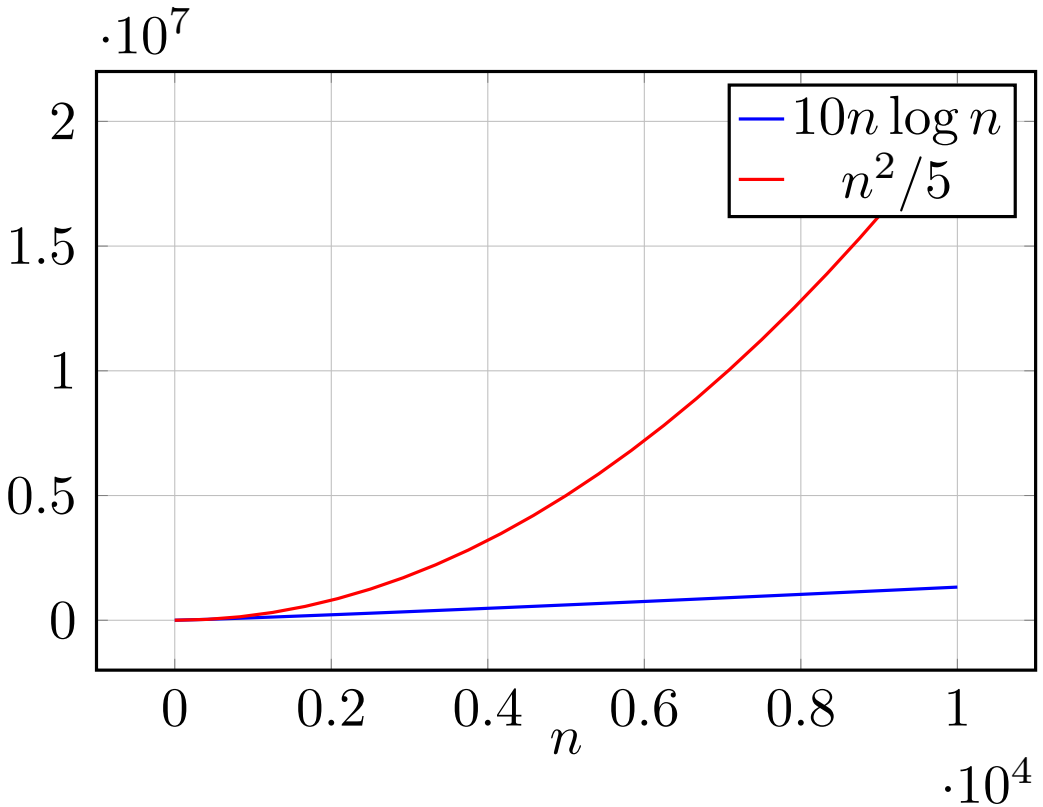
Mergesort wird „für genügend große Inputs“ schneller sein
als Insertionsort (mit $T_{\text{IS,worst}}(n) = \Theta(n^2)$).

(Die Praxis zeigt: Für viele Algorithmenpaare tritt ein solcher „asymptotischer“ Unterschied schon für nicht allzugroße Werte von n zu Tage.)

Die folgenden Bilder vergleichen Funktionen in $\Theta(n^2)$ und $\Theta(n \log n)$ für bestimmte versteckte Konstanten. Für kleine n scheint die $\Theta(n^2)$-Funktion im Vorteil zu sein, aber für größere n gewinnt die $\Theta(n \log n)$-Funktion klar. Der Überschneidungspunkt hängt von den versteckten Konstanten ab.



Für $n \leq 430$ ist $n^2/5$ „besser als“ $10n \log n$.



Für n in $[10^3, 10^4]$ übernimmt das asymptotische Verhalten: $10n \log n$ **viel besser** als $n^2/5$.

Wachstumsordnungen

„Einfache“ Funktionen mit unterschiedlich schnellem Wachstum bilden **Hierarchie**.
Beispiele (s. Übung):

$$
\begin{array}{lll}
f_0(n) = 1, & & \\
f_1(n) = \log\log n, & f_2(n) = \log n, & f_3(n) = (\log n)^2, \\
f_4(n) = \sqrt{n}, & f_5(n) = \frac{n}{\log n}, & f_6(n) = n, \\
f_7(n) = n \log n, & f_8(n) = n(\log n)^2, & f_9(n) = n^{3/2}, \\
f_{10}(n) = n^2, & f_{11}(n) = n^3, & f_{12}(n) = 2^n, \\
f_{13}(n) = n^5 \cdot 2^n, & f_{14}(n) = e^n, & f_{15}(n) = 3^n \\
f_{16}(n) = (n/e)^n, & f_{17}(n) = n!, & f_{18}(n) = n^n.
\end{array}
$$

Es gilt: $\lim_{n\to\infty} f_i(n)/f_{i+1}(n) = 0$, also $f_i(n) = o(f_{i+1}(n))$, für $0 \leq i \leq 17$.

Mit $f_{-i}(n) := 1/f_i(n)$, für $1 \leq i \leq 18$, gilt dies sogar für $-18 \leq i \leq 17$.

Vergleich einiger Wachstumsordnungen

	n							
$t_{\mathcal{A}}(n)$	10	100	1000	10^4	10^5	10^6	10^7	10^8
$\log n$	33ns	66ns	0.1μs	0.1μs	0.2μs	0.2μs	0.2μs	0.3μs
$\sqrt{n}$	32ns	0.1μs	0.3μs	1μs	3.1μs	10μs	31μs	0.1ms
n	100ns	1μs	10μs	0.1ms	1ms	10ms	0.1s	1s
$n \log n$	0.3μs	6.6μs	0.1ms	1.3ms	16ms	0.2s	2.3s	27s
$n^{3/2}$	0.3μs	10μs	0.3ms	10ms	0.3s	10s	5.2m	2.7h
n^2	1μs	0.1ms	10ms	1s	1.7m	2.8h	11d	3.2y
n^3	10μs	10ms	10s	2.8h	115d	317y	$3.2{\cdot}10^5$y	
1.1^n	26ns	0.1ms	$7.8{\cdot}10^{25}$y					
2^n	10μs	$4.0{\cdot}10^{14}$y						
$n!$	36ms	$3.0{\cdot}10^{142}$y						
n^n	1.7m	$3.2{\cdot}10^{184}$y						

Zeit für eine (Elementar-)Operation: 10 ns

Hilft ein schnellerer Rechner?

Wie wächst die maximal mögliche Eingabegröße, wenn der Rechner um den Faktor 10 schneller wird?

$T_{\mathcal{A}}(n)$	Max_{alt}	Max_{NEU}
$\log n$	n	n^{10}
$\sqrt{n}$	n	$100n$
n	n	$10n$
$n \log n$	n	$10n \cdot \frac{\log n}{\log n + \log 10}$
$n^{3/2}$	n	$4.64n$
n^2	n	$3.16n$
n^3	n	$2.15n$
n^k	n	$10^{1/k}n$
c^n	n	$n + \log_c 10$
$n!$	um von n auf $n+1$ zu kommen, braucht man einen um den Faktor n schnelleren Rechner	
n^n		

Hilft ein schnellerer Rechner?

Angetrieben durch die höheren **Rechengeschwindigkeiten** und die noch rascher wachsenden **Speicherkapazitäten** wächst in vielen Bereichen die Menge der zu bewältigenden Daten noch schneller als die Rechnergeschwindigkeit.

Beispiele:
Indizes der Internet-Suchmaschinen.
Routenplaner, Fahrplan-, Ticketsysteme der Bahn.
Computergraphik (Szenen aus zig Milliarden von Elementen!)

Um die steigende Rechnergeschwindigkeit wenigstens einigermaßen ausnutzen zu können, sind Algorithmen mit polynomieller Laufzeit (kleine Exponenten) wesentlich.

Unser Ziel in dieser Vorlesung: Bekannte hocheffiziente Algorithmen kennenlernen.

Techniken für den Entwurf und die Analyse effizienter Algorithmen bereitstellen.

Ende 2. Vorlesung

SS 2021

Algorithmen und Datenstrukturen

2. Kapitel

Fundamentale Datentypen und Datenstrukturen

Martin Dietzfelbinger

Mai 2021

Thema: Datentypen und Datenstrukturen

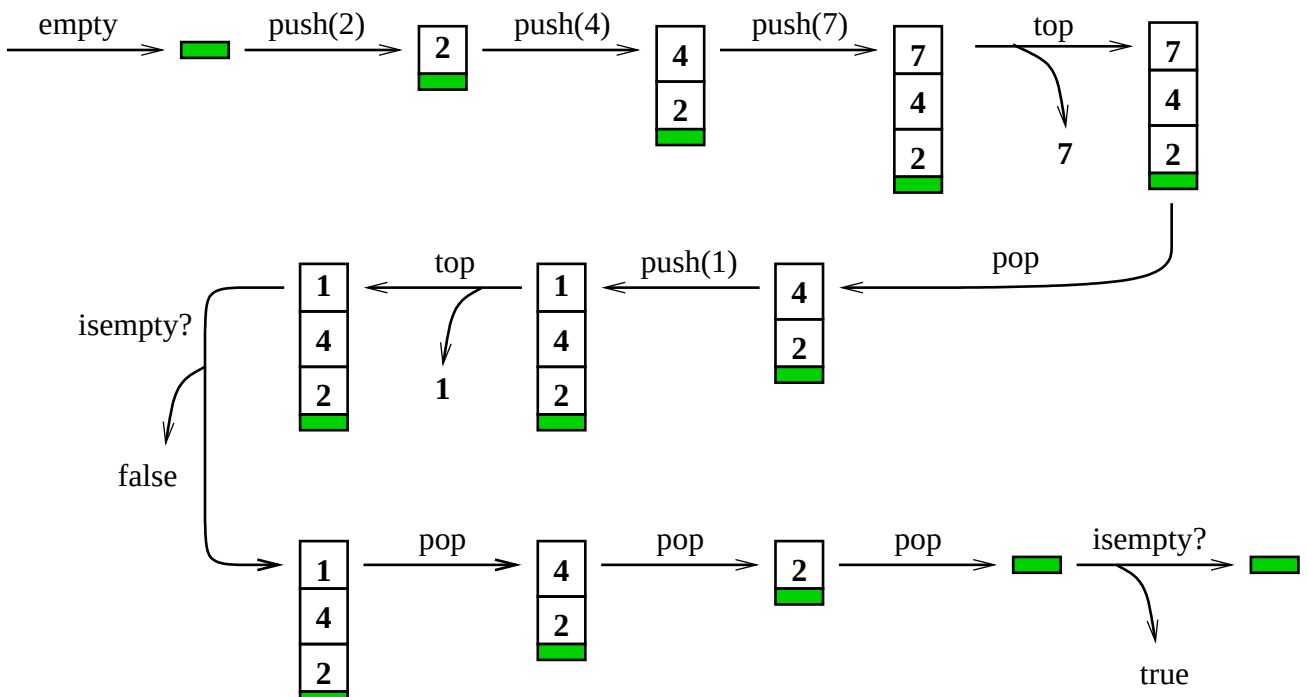
Einfache Datentypen (werden hier diskutiert)

- **Stacks** (Keller, Stapel, LIFO-Speicher) (Prinzipien bekannt)
- **Dynamische Arrays** (Indizierte dynamische Folgen) (Erweiterung von Stacks)
- Queues (Warteschlangen, FIFO-Speicher) (erwähnt in AuP)
- Mengen
- Wörterbücher oder Assoziative Arrays

Basis-Datenstrukturen (als bekannt vorausgesetzt)

- Statische Arrays (Kap. 2.3.5 im Buch von Saake und Sattler)
- Einfach und doppelt verkettete Listen (Kap. 13.2 und 13.3 im Buch von Saake und Sattler)

2.1 Stacks* und dynamische Arrays



* Aliasse: Keller, Stapel, **L**ast**I**n**F**irst**O**ut-Speicher, Pushdown-Speicher

Informale Beschreibung der Operationen; Vorstellung ist ein **Stapel** (Alltagsbegriff).

Einträge: Elemente einer **Menge D (Parameter)**.

empty$()$:	erzeuge leeren Stapel („Konstruktor“).
push(s, x):	lege Element $x \in D$ oben auf den Stapel s.
pop(s):	entferne oberstes Element von Stapel s (falls s leer: Fehler).
top(s):	gib oberstes Element des Stapels s aus (falls s leer: Fehler).
isempty(s):	Ausgabe „$true$“ falls Stapel s leer, „$false$“ sonst.

Eine Beschreibung in Umgangssprache und durch ein Beispiel genügt aber nicht. Um die Frage nach der **Korrektheit einer Implementierung** überhaupt sinnvoll stellen zu können, benötigen wir eine präzise Beschreibung des Verhaltens, also eine **Spezifikation**.

Spezifikation des Datentyps „Stack über D“

Namen von „Sorten“, Namen von „Operationen“ mit Angabe der Namen der Argumentsorten und Resultatsorte für jede Operation.

1. Signatur:

Sorten:	*Elements*
	Stacks
	Boolean
Operationen:	*empty*: $\rightarrow$ *Stacks*
	push: *Stacks* $\times$ *Elements* $\rightarrow$ *Stacks*
	pop: *Stacks* $\rightarrow$ *Stacks*
	top: *Stacks* $\rightarrow$ *Elements*
	isempty: *Stacks* $\rightarrow$ *Boolean*

Rein syntaktische Vorschriften! Verhalten (noch) ungeklärt!

Spezifikation des Datentyps „Stack über D“

2. Mathematisches Modell ([Saake/Sattler]-Terminologie (Kap. 11): „Algebra“)

// **Sortennamen** werden durch **Mengen** unterlegt.

Sorten: *Elements*: Menge D, nichtleer **(Parameter)**

Stacks: $D^{<\infty} = \mathsf{Seq}(D)$

mit $\mathsf{Seq}(D) = \{(a_1, \ldots, a_n) \mid n \in \mathbb{N}, a_1, \ldots, a_n \in D\}$

Boolean: $\{true, false\}$ bzw. $\{1, 0\}$

Ein Stack wird also als Folge $(\boldsymbol{a_1}, a_2, \ldots, a_n)$, mit $n \geq 0$, dargestellt.

Das **linke Ende** der Folge, Element $\boldsymbol{a_1}$, steht im Stack **oben**.

Die leere Folge ( ) stellt den leeren Stack dar.

Spezifikation des Datentyps „Stack über D“

2. Mathematisches Modell (Forts.)

// **Operationen** werden durch (mathematische) **Funktionen** modelliert.

Operationen: *empty*() := ()

$$\mathit{push}((a_1, \ldots, a_n), x) := (x, a_1, \ldots, a_n)$$

$$\mathit{pop}((a_1, \ldots, a_n)) := \begin{cases} (a_2, \ldots, a_n), & \text{falls } n \geq 1 \\ \text{undefiniert}, & \text{falls } n = 0 \end{cases}$$

$$\mathit{top}((a_1, \ldots, a_n)) := \begin{cases} a_1, & \text{falls } n \geq 1 \\ \text{undefiniert}, & \text{falls } n = 0 \end{cases}$$

$$\mathit{isempty}((a_1, \ldots, a_n)) := \begin{cases} \mathit{false}, & \text{falls } n \geq 1 \\ \mathit{true}, & \text{falls } n = 0 \end{cases}$$

Spezifikation des Datentyps (ADT) „Stack über D“

Alternative: (siehe [AuP], Kap. 8 und [Saake/Sattler], Kap. 11)

„1. Signatur“: wie oben.

2. Axiome:

$\forall\, s$: *Stacks*, $\forall\, x$: *Elements*

$$\mathit{pop}(\mathit{push}(s, x)) = s$$
$$\mathit{top}(\mathit{push}(s, x)) = x$$
$$\mathit{isempty}(\mathit{empty}()) = \mathit{true}$$
$$\mathit{isempty}(\mathit{push}(s, x)) = \mathit{false}$$

Vorteil: Zugänglich für automatisches Beweisen von Eigenschaften des Datentyps.

Nachteil: Finden eines geeigneten Axiomensystems für eine geplante Semantik evtl. nicht offensichtlich.

Implementierung von Stacks

Grundsätzlich, in beliebiger objektorientierter Sprache: als Klasse mit Methoden.

Klasse für Elemente (ein Parameter, bereitgestellt über Generic- bzw. Template-Mechanismus): `elements`. Dazu: `boolean` aus der Programmiersprache.

Prinzip: Kapselung – Information Hiding

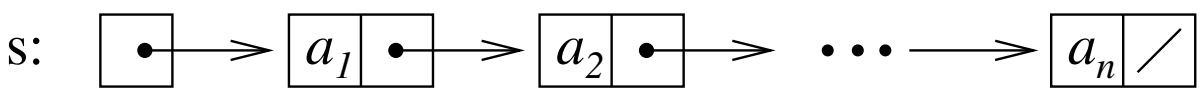
Der Benutzer sieht die Details der Implementierung nicht. Zugriff auf die Objekte der Datenstruktur erfolgt ausschließlich über die (öffentlichen) Methoden der Klassen, entsprechend den Operationen des Datentyps.

Zwei strukturell unterschiedliche Möglichkeiten (mindestens!) für Implementierung von Stackobjekten:

(Einfach verkettete) **Lineare Liste** und **Array**.

Listenimplementierung von Stacks

Ein Stack $(a_1, \ldots, a_n)$ („oben“ ist bei a_1. Die Einträge stammen aus der Klasse `elements`) wird durch eine einfach verkettete Liste s dargestellt. Die Einträge stehen in der Liste in Reihenfolge $a_1, \ldots, a_n$:



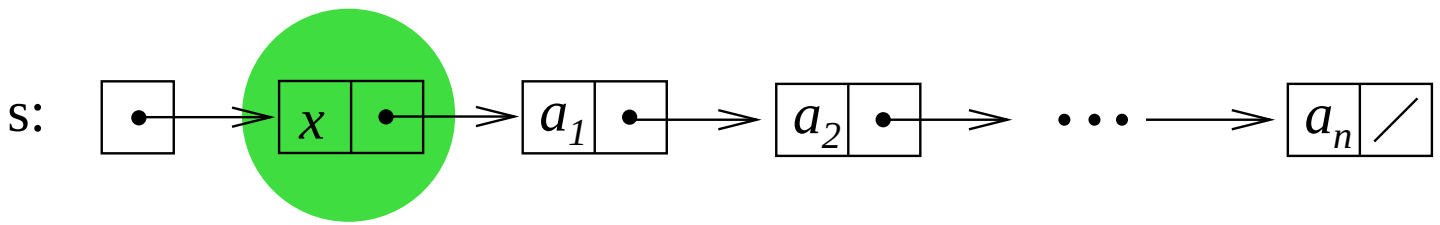
(Das Symbol ⁄ steht für den Nullzeiger/die Nullreferenz)

Zur Stack-Klasse gehören Methoden, die den Operationen entsprechen.

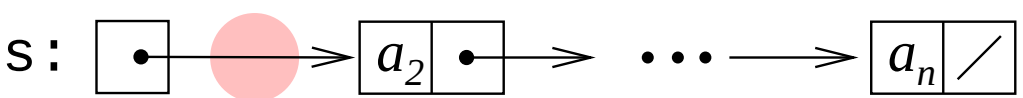
`s.empty`: Erzeugt neue leere Liste.

Listenimplementierung von Stacks

`s.push(`x`)`: Setze neuen Listenknoten mit Eintrag x an den Anfang der Liste. Aus der Liste für $(a_1, \ldots, a_n)$ und einem Objekt x vom Typ `elements` wird also



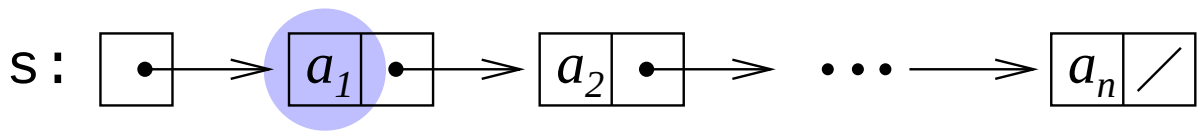
`s.pop`: Entferne ersten Listenknoten (falls vorhanden). Aus der Liste für $(a_1, \ldots, a_n)$ mit $n \geq 1$ wird also (an der Stelle des roten Kreises stand vorher a_1):



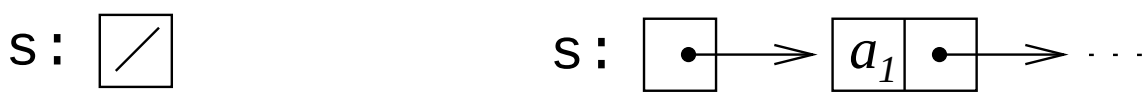
Wenn `s.pop` für eine leere Liste aufgerufen wird, muss eine Fehlermeldung (exception o. ä.) erfolgen.

Listenimplementierung von Stacks

`s.top`: Gib Inhalt des ersten Listenknotens aus (falls vorhanden, sonst Fehler)



`s.isempty`: Falls Liste leer: Ausgabe *true*, sonst *false*, Ausgabetyp `boolean`.



Listenimplementierung von Stacks

Beobachtung

Wenn der Benutzer nie den Fehler macht, *pop* oder *top* für den leeren Stack (die leere Liste) aufzurufen, dann ist der einzige Fehler, der bei der Benutzung dieser Implementierung auftreten kann, ein Speicherüberlauf bei der Operation `s.push(`x`)`, weil für das neue Listenelement im für das Programm verfügbaren Speicher kein Platz mehr ist.

Beobachtung

Jede der Methoden hat Rechenzeit $O(1)$ (also konstant, unabhängig von n).

Wir bemerken aber, dass in vielen Programmiersprachen die Aktion, ein neues Listenelement zu kreieren, wie es in `s.push` benötigt wird, eine deutlich größere (konstante) Rechenzeit erfordert als etwa ein einfacher Speicherzugriff oder der Zugriff auf den ersten Eintrag einer linearen Liste. – Eine Möglichkeit, diesen Effekt abzuschwächen, wäre es, von Zeit zu Zeit ein ganzes Array von Listenelementen auf einen Schlag zu allozieren und die Verwaltung der Listenelemente direkt, als Teil der Datenstruktur, durchzuführen. Spart Rechenzeit, erhöht aber den Programmieraufwand.

PAUSE

Korrektheit der Listenimplementierung

Eine beliebige Menge D sei gegeben. Wir betrachten eine Folge

$$\mathrm{Op}_0 = \textit{empty}, \mathrm{Op}_1, \ldots, \mathrm{Op}_N \quad (\textit{push} \text{ hat immer ein Argument aus } D)$$

von Stackoperationen, wobei *empty* nur als Op_0 vorkommt.
Wenn man die Operationen dieser Folge im mathematischen Modell der Reihe nach anwendet, und es keinen Fehler gibt, entsteht eine Folge

$$s_0, s_1, s_2, \ldots, s_N$$

von Stacks (Tupeln), die man als Zustände auffasst, und eine Folge

$$z_0, z_1, z_2, \ldots, z_N$$

von Ausgaben (Werte in D bzw. $\{true, false\}$).

Beispiel (mit $D = \mathbb{N}$) :

i	OP_i	s_i	z_i
0	*empty*	$()$	–
1	*push*(2)	(2)	–
2	*push*(4)	$(4, 2)$	–
3	*push*(7)	$(7, 4, 2)$	–
4	*top*	$(7, 4, 2)$	7
5	*pop*	$(4, 2)$	–
6	*push*(1)	$(1, 4, 2)$	–
7	*top*	$(1, 4, 2)$	1
8	*isempty*	$(1, 4, 2)$	$false$
9	*top*	$(1, 4, 2)$	1
10	*pop*	$(4, 2)$	–
11	*pop*	(2)	–
12	*pop*	$()$	–
13	*isempty*	$()$	$true$
14	*pop*	$\lightning$	$\lightning$
15	*push*(3)	$\lightning$	$\lightning$
⋮	⋮	⋮	⋮

Korrektheit der Listenimplementierung

$\mathrm{Op}_0 = \mathit{empty}()$ liefert $s_0 = ()$.
$\mathrm{Op}_i = \mathit{push}(x)$ angewendet auf $s_{i-1} \in \mathrm{Seq}(D)$ liefert $s_i = \mathit{push}(s_{i-1}, x)$. (1).
$\mathrm{Op}_i = \mathit{pop}(x)$ angewendet auf $s_{i-1} \in \mathrm{Seq}(D)$ liefert $s_i = \mathit{pop}(s_{i-1})$. (1).
$\mathrm{Op}_i = \mathit{top}(x)$ angewendet auf $s_{i-1} \in \mathrm{Seq}(D)$ liefert $s_i = s_{i-1}$ und $z_i = \mathit{top}(s_{i-1})$. (1).
$\mathrm{Op}_i = \mathit{isempty}()$ angewendet auf $s_{i-1} \in \mathrm{Seq}(D)$ liefert $s_i = s_{i-1}$ und $z_i = \mathit{isempty}(s_{i-1})$. (1).

(1): jeweils solange $s_{i-1} \neq \lightning$ und in Op_i kein Fehler auftritt.

Fehlerfall: Wenn $\mathrm{Op}_i = \mathit{top}$ oder $\mathrm{Op}_i = \mathit{pop}$ und $s_{i-1} = ()$, oder wenn $s_{i-1} = \lightning$, dann ist $s_i = \lightning$ und $z_i = \lightning$.

(Keine „Erholung“ nach dem ersten Fehler!)

Beispiel für Ein-/Ausgabeverhalten

Korrektheit der Implementierung bedeutet, dass das Ein-/Ausgabeverhalten des mathematischen Modells auf beliebigen Operationsfolgen nachgebaut wird.

OP_i	z_i
empty	–
push(2)	–
push(4)	–
push(7)	–
top	7
pop	–
push(1)	–
top	1
isempty	$false$
top	1
pop	–
pop	–
pop	–
isempty	$true$
pop	$\lightning$
push(3)	$\lightning$
$\vdots$	$\vdots$

Satz 2.1.1

Die Listenimplementierung ist **korrekt**, d. h. sie erzeugt auf jeder Eingabefolge $\mathrm{Op}_0, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$ genau dieselbe Ausgabefolge wie das mathematische Modell, solange **weder** im mathematischen Modell ein Fehler auftritt (*top* oder *pop* auf leerem Stack) **noch** in der Implementierung ein Laufzeitfehler **„Speicherüberlauf“** eintritt (bei einem *push*(x)).

Beweis: Man zeigt per Induktion über $i = 0, \ldots, N$ folgende Induktionsbehauptung, für die Ausführung von $\mathrm{Op}_0, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$ als Schritte $0, \ldots, N$:

(IB_i) Wenn in Schritten $0, \ldots, i$ im mathematischen Modell und in der Implementierung kein Fehler aufgetreten ist, dann sind die Einträge in der Liste (der Datenstruktur) genau die Einträge in s_i (dem Zustand des mathematischen Modells), und die Ausgabe der Datenstruktur in Schritt i ist z_i.

Beweis von (IB$_i$) durch Induktion über i.

I.A.: $i = 0$. Wir haben Op$_0$ = *empty*. Also: $s_0 = ()$ und Op$_0$ liefert die leere Liste, und (IB$_0$) gilt.

I.V.: $i > 0$ und (IB$_{i-1}$) gilt.

I.-Schritt: Wenn $s_{i-1} = \lightning$, müssen wir nichts zeigen. Nehmen wir also $s_{i-1} = (a_1, \ldots, a_n)$ an. Nach I.V. enthält die Datenstruktur eine Liste mit Einträgen $a_1, \ldots, a_n$.

Nun betrachtet man Op$_i$. Wenn dies *top* ist, gibt das mathematische Modell $z_i = a_1$ aus, ebenso wie die Implementierung, vorausgesetzt, es ist $n \geq 1$. Wenn $n = 0$ ist, wechselt das mathematische Modell in den Fehlerzustand $\lightning$, gibt „Fehler“ aus, und auch die Implementierung meldet „Fehler“. Es folgt (IB$_i$).

Wenn Op$_i$ = *pop* ist, liefert das mathematische Modell $s_i = (a_2, \ldots, a_n)$, und die Implementierung entfernt den ersten Eintrag a_1 aus der Liste, vorausgesetzt, es gilt $n \geq 1$. Sonst wechseln beide in einen Fehlerzustand. Es folgt (IB$_i$).

Wenn Op$_i$ = *push*(x), dann liefert das mathematische Modell $s_i = (x, a_1, \ldots, a_n)$, und die Implementierung fügt x vorne in die Liste ein. Es folgt (IB$_i$).

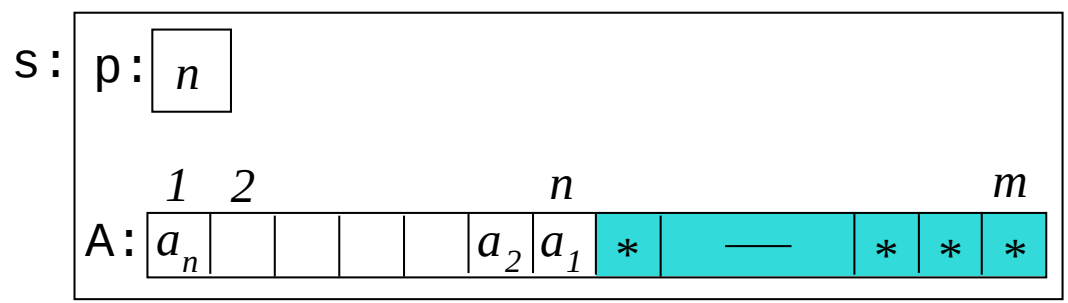
Wenn Op$_i$ = *isempty*, liefert das mathematische Modell $z_i = true$ bzw. $z_i = false$, je nachdem ob $n \geq 1$ oder $n = 0$ gilt. Genau dieselbe Ausgabe wird von der Implementierung erzeugt. Also gilt (IB$_i$).

Kommentar

- Wenn man diesen Beweis betrachtet, drängt sich der Gedanke auf, dass es sich um eine gewaltige Banalität handelt. Man überprüft ja nur, was bei der Definition des mathematischen Modells und bei der Programmierung der Datenstruktur geplant war, nämlich dass die Implementierung genau das mathematische Modell nachahmt.
- Diese Situation tritt bei der Implementierung von nicht zu komplexen Datenstrukturen recht häufig ein.
- Die Korrektheitsbeweise werden dann „trivial“ oder „banal“, weil sie nur das Offensichtliche nachkontrollieren.
- Wir werden daher solche Beweise im weiteren Verlauf nicht mehr durchführen, sondern werden uns üblicherweise darauf beschränken, die (sorgfältig formulierte) Induktionsbehauptung anzugeben. Das muss man aber wirklich auch machen! Auf das mechanische, langweilige Durchspielen von Fällen werden wir verzichten.
- Beweise dieser Art muss man in der Prüfung nicht durchführen können, aber man muss gegebenenfalls erklären können, was zu tun ist (und die Induktionsbehauptung aufschreiben können).

Arrayimplementierung von Stacks

(Vgl. [Saake/Sattler], Kap. 13.1, oder [AuP], Kap. 10.) Das Stackobjekt heißt `s`; es hat als Attribute einen Zeiger/eine Referenz auf ein Array `A[1..`m`]` und eine Integervariable `p` („Pegel“).



Idee: Wenn der Stack im mathematischen Modell momentan $(a_1, \ldots, a_n)$ ist, steht n in `p` und in `A[1]` steht a_n (unterster Eintrag), . . . , in `A[`n`]` steht a_1 (oberster Eintrag). **Achtung: Reihenfolge umgedreht!**

„$*$“ steht für einen beliebigen Arrayeintrag, in `A[`i`]`, für $n + 1 \leq i \leq m$.

Arrayimplementierung von Stacks: Methoden

`s.empty`: wird durch Konstruktor realisiert; Option dabei: m als Parameter. (Sonst: m fix gewählt.)
  Erzeuge neues Objekt `s` mit `A[1..`m`] of elements` und `p: int` ;
  `p` $\leftarrow 0$;

`s.push(x)`:
  if `p` $= $ `A.length` **then „Fehler“** (z.B. *OverflowException*)
  else `p` $\leftarrow$ `p+1`; `A[p]` $\leftarrow$ `x`;

`s.pop`:
  if `p` $= 0$ **then „Fehler“** (z.B. *StackemptyException*)
  else `p` $\leftarrow$ `p-1`;

`s.top`:
  if `p` $= 0$ **then „Fehler“** (z.B. *StackemptyException*)
  else return `A[p]`;

`s.isempty`:
  if `p` $= 0$ **then return** „$true$“ **else return** „$false$“;

Korrektheit der Arrayimplementierung

Wir betrachten wieder beliebige Folgen $\mathrm{Op}_0 =$ *empty*, $\mathrm{Op}_1, \ldots, \mathrm{Op}_N$ von Stackoperationen (wobei *empty* nur als Op_0 vorkommen darf). Im mathematischen Modell wird davon eine Folge $s_0, s_1, s_2, \ldots, s_N$ von Stacks (Tupeln) und eine Folge $z_0, z_1, z_2, \ldots, z_N$ von Ausgaben erzeugt.

Satz 2.1.2

Die Arrayimplementierung ist **korrekt**, d. h. sie erzeugt auf Eingabe $\mathrm{Op}_0 =$ *empty*, $\mathrm{Op}_1, \ldots, \mathrm{Op}_N$ genau dieselbe Ausgabefolge wie das mathematische Modell, solange **weder** im mathematischen Modell ein Fehler (*top* oder *pop* auf leerem Stack) auftritt **noch** die Höhe des Stacks s_i (d. h. die Länge der Folge s_i) die Arraygröße m überschreitet.

Beweis: Man beweist durch Induktion über $i = 0, 1, \ldots, N$ die folgende Induktionsbehauptung für die Ausführung von $\mathsf{Op}_0, \mathsf{Op}_1, \ldots, \mathsf{Op}_N$ als Schritte $0, \ldots, N$:

(IB_i)

Wenn in Schritten $0, \ldots, i$ im mathematischen Modell kein Fehler aufgetreten ist und die Folgen (Stacks) $s_1, \ldots, s_i$ alle höchstens Länge m haben, dann gilt nach Schritt i:

- p enthält die Länge n_i der Folge s_i und
- $s_i = (\mathtt{A}[n_i], \ldots, \mathtt{A}[1])$ und
- die Implementierung gibt in Schritt i genau z_i aus.

D. h.: $\mathtt{A}[1..\mathtt{p}]$ stellt genau s_i dar, in umgekehrter Reihenfolge.

Beachte die folgende Subtilität, die die Arrayimplementierung von der Listenimplementierung unterscheidet: Der Teil $\mathtt{A}[n_i + 1..m]$ des Arrays kann völlig beliebige Einträge enthalten. (Das sind die $*$-Einträge im Bild auf Folie 19.) D. h.: Ein und derselbe Stack s im Sinn des mathematischen Modells kann durch verschieden beschriftete Arrays $\mathtt{A}[1..m]$ dargestellt werden.

Details des Induktionsbeweises von Satz 2.1.2:
I.A.: $i = 0$. Anfänglich hat p den Inhalt $n_0 = 0$. $\Rightarrow (\mathtt{A}[n_0], \ldots, \mathtt{A}[1]) = () = s_0$.
(gleichgültig welchen Inhalt $\mathtt{A}[1..m]$ hat).
I.V.: $i > 0$, und (IB_{i-1}) gilt.
I.Schritt.: Wenn $s_{i-1} = \lightning$ (d. h. es gab einen Fehler im mathematischen Modell) oder wenn eines der s_j mit $j < i$ Länge $> m$ hat, ist nichts zu zeigen. Also nehmen wir $s_{i-1} \in \mathrm{Seq}(D)$ an, und keinen Überlauf in der Implementierung.
Nach I.V. haben wir: $s_{i-1} = (a_1, \ldots, a_{n_{i-1}})$, wobei n_{i-1} der Inhalt von p ist, und
$s_{i-1} = (\mathtt{A}[n_{i-1}], \ldots, \mathtt{A}[1])$.
Fall 1: $\mathrm{Op}_i = \mathtt{s.push}(x)$. – Dann ist $s_i = (x, a_1, \ldots, a_{n_{i-1}})$.
Wenn $n_{i-1} < m$ ist, wird von der Prozedur $\mathtt{s.push}(x)$ das Objekt x an die Stelle $\mathtt{A}[n_{i-1} + 1]$ gesetzt und p auf den Wert $n_{i-1} + 1 = n_i$ erhöht. Daraus folgt (IB_i). Wenn aber $n_{i-1} = m$ ist, tritt in der Implementierung ein Fehler ein, und (IB_i) ist trivialerweise erfüllt.
Fall 2: $\mathrm{Op}_i = \mathtt{s.top}()$. – Dann ist $s_i = (a_1, \ldots, a_{n_{i-1}})$, und auch in der Implementierung ändert sich nichts im Stackobjekt. Wenn $n_{i-1} \geq 1$ ist, wird in der Implementierung $\mathtt{A}[n_{i-1}] = a_1$ ausgegeben, ebenso wie im mathematischen Modell. Wenn $n_{i-1} = 0$, erzeugt die top-Operation im mathematischen Modell einen Fehler, und wieder gilt (IB_i). (Mathematisches Modell und Implementierung melden den Fehler.)
Die anderen Fälle, entsprechend den anderen Operationen, behandelt man analog.

Zeitaufwand der Arrayimplementierung

Behauptung 2.1.3

Bei der Arrayimplementierung von Stacks hat jede einzelne Operation Kosten $O(1)$.

Das sieht man durch Betrachtung der einzelnen Operationen.

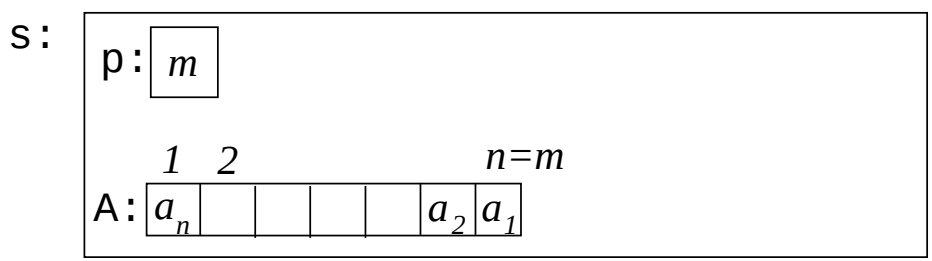
Bei der Initialisierung `empty()` (Aufruf des Konstruktors) muss man aber auf Folgendes achten: Wenn man m, die Arraygröße, vom Benutzer als Parameter angeben lässt (was sehr sinnvoll ist!), dann darf das Array `A[`$1..m$`]` vom Konstruktor nur **alloziert**, nicht initialisiert werden. In C++ ist dies möglich. Andere Programmiersprachen wie etwa Java initialisieren ein neu angelegtes Array automatisch. In diesem Fall hat `empty()` natürlich Rechenzeit $\Theta(m)$.

Man beobachte, dass die Korrektheit nicht beeinträchtigt wird, wenn das Array anfangs einen völlig beliebigen Inhalt hat.

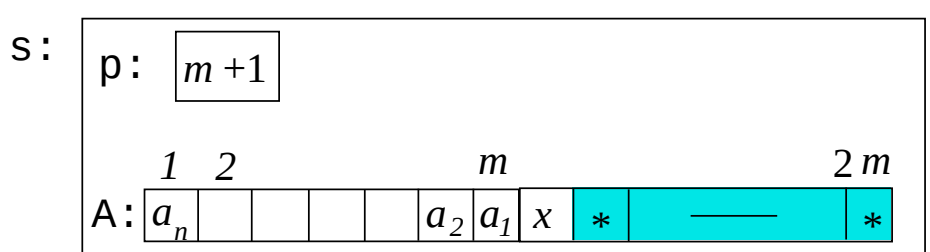
Vermeidung von Platzproblemen: Verdoppelungsstrategie

`s.push(`x`)` bei voller Tabelle.

Vorher:



Nachher:



Vermeidung von Platzproblemen: Verdoppelungsstrategie

Verdoppelungsstrategie

In `push`(x): Bei Überlaufen des Arrays `A`$[1..m]$ (d. h. wenn der Pegelstand n gleich der Arraylänge m ist) **nicht** einen Laufzeitfehler (oder eine *exception*) generieren, sondern:

1. Ein neues, doppelt so großes Array `AA` allozieren.
2. m Einträge aus `A`$[1..m]$ nach `AA`$[1..m]$ kopieren.
3. x an Stelle $n+1$ schreiben, Pegel auf $n+1$ setzen.
4. `AA` in `A` umbenennen/Referenz umsetzen.

   (Altes Array freigeben, falls in der Programmiersprache sinnvoll. (C++ ja, Java nein.))

Kosten: $\Theta(m)$, wo $m = n =$ aktuelle Größe des Stacks.

Analyse der Gesamtkosten bei Verdoppelungsstrategie

Betrachte eine beliebige Operationenfolge $\mathrm{Op}_0 = \textit{empty}, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$.
Arraygröße am Anfang: m_0. (Von Implementierung fix gewählt oder vom Benutzer bestimmt.)
Jede Operation hat Kosten $O(1)$, außer wenn die Arraygröße verdoppelt wird.

Die Arraygröße wächst erst von m_0 auf $2m_0$, dann von $2m_0$ auf $4m_0$, usw., allgemein: von $m_0 \cdot 2^{\ell-1}$ auf $m_0 \cdot 2^{\ell}$. Bei der ℓ-ten Verdopplung entstehen Kosten $\Theta(m_0 \cdot 2^{\ell-1})$.

$k :=$ Anzahl der push-Operationen in $\mathrm{Op}_1, \ldots, \mathrm{Op}_N \Rightarrow$ Zahl der Einträge ist immer $\leq k$, also gilt, wenn die Verdopplung von $m_0 \cdot 2^{\ell-1}$ auf $m_0 \cdot 2^{\ell}$ tatsächlich stattfindet: $m_0 \cdot 2^{\ell-1} < k$.

Sei L maximal mit $m_0 \cdot 2^{L-1} < k$.

Dann sind die Gesamtkosten für alle Verdopplungen zusammen höchstens:

$$\sum_{1 \leq \ell \leq L} O(m_0 \cdot 2^{\ell-1}) \overset{(*)}{=} O\big(m_0 \cdot (1 + 2 + 2^2 + \cdots + 2^{L-1})\big) \overset{(**)}{=} O\big(m_0 \cdot 2^L\big) = O(k).$$

Die **Gesamtkosten** für alle Verdopplungen zusammen sind also $N \cdot \Theta(1) + O(k) = \Theta(N)$.

Bemerkung: Wir haben für $\overset{(*)}{=}$ die Summationsregel für die O-Notation und für $\overset{(**)}{=}$ die Summenformel für geometrische Reihen benutzt.

Satz 2.1.4

Wenn ein Stack mit **Arrays** und der **Verdoppelungsstrategie** implementiert wird, gilt:
(a) Die Gesamtkosten für N Operationen betragen $\Theta(N)$.
(b) Der gesamte Platzbedarf ist $\Theta(k)$, wenn k die maximale je erreichte Stackhöhe ist.

Bemerkung: Selbst wenn der Speicher nie bereinigt wird (d. h. wenn kleinere, nicht mehr benutzte Arrays nicht freigegeben werden), tritt ein Speicherüberlauf erst ein, wenn die Zahl der Stackeinträge zu einem bestimmten Zeitpunkt mehr Speicher beansprucht als $\frac{1}{4}$ des gesamten zur Verfügung stehenden Speichers.

Bemerkung: Was wir gemacht haben, nämlich die *Summe* der Kosten einer Reihe von Operationen abzuschätzen (anstatt einfach die Anzahl der Operationen mit den maximalen Kosten zu multiplizieren), nennt man **„amortisierte Analyse“**. Dieser Ansatz wird uns noch mehrfach begegnen.

Vergleich Listen-/Arrayimplementierung:

Arrayimplementierung

- $O(1)$ Kosten pro Operation, **„amortisiert“** (im Durchschnitt über alle Operationen),
- Sequentieller, indexbasierter Zugriff im Speicher (cachefreundlich),
- Höchstens 50% des allozierten Speicherplatzes bleibt ungenutzt.

Listenimplementierung

- $O(1)$ Kosten pro Operation **im schlechtesten Fall**
  (jedoch: relativ hohe Allokationskosten bei *push*-Operationen),
- Zusätzlicher Platz für Zeiger benötigt.

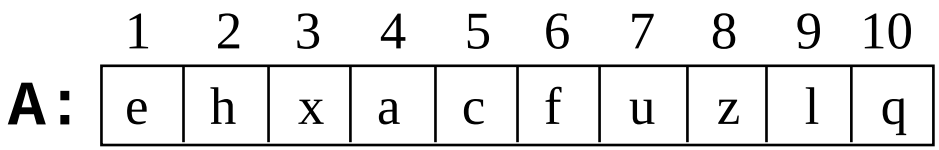
Experimente zeigen, dass die Arrayimplementierung spürbar kleinere Rechenzeiten aufweist.

PAUSE

Datentyp Dynamische Arrays

Stacks + „wahlfreier Zugriff“.

Neuer Grunddatentyp: *Nat* – durch $\mathbb{N} = \{0, 1, 2, \ldots\}$ interpretiert. Anschaulich:



Aktuelle Länge: $n = 10$

Operationen, informal beschrieben:

empty():	liefert leeres Array, Länge 0
read(A, i):	liefert Eintrag A[i] an Position i, falls $1 \leq i \leq n$.
write(A, i, x):	ersetzt Eintrag A[i] an Position i durch x, falls $1 \leq i \leq n$.
length(A):	liefert aktuelle Länge n.
addLast(A, x):	verlängert Array um 1 Position, schreibt x an letzte Stelle ($\hat{=}$ push).
removeLast(A):	verkürzt Array um eine Position ($\hat{=}$ pop).

Wieso gibt es kein „isempty“? (Teste *length*(A) auf 0.)

Dynamische Arrays

1. Signatur:

Sorten:	*Elements*
	Arrays
	Nat
Operationen:	$empty: \rightarrow Arrays$
	$addLast: Arrays \times Elements \rightarrow Arrays$
	$removeLast: Arrays \rightarrow Arrays$
	$read: Arrays \times Nat \rightarrow Elements$
	$write: Arrays \times Nat \times Elements \rightarrow Arrays$
	$length: Arrays \rightarrow Nat$

2. Mathematisches Modell

Sorten:	*Elements*:	(nichtleere) Menge D	(Parameter)
	Arrays:	Seq(D)	
	Nat:	$\mathbb{N}$	

Dynamische Arrays

2. Mathematisches Modell (Forts.)

Operationen:

$$\mathit{empty}() := ()$$

$$\mathit{addLast}((a_1, \ldots, a_n), x) := (a_1, \ldots, a_n, x)$$

$$\mathit{removeLast}((a_1, \ldots, a_n)) := \begin{cases} (a_1, \ldots, a_{n-1}), & \text{falls } n \geq 1 \\ \text{undefiniert}, & \text{falls } n = 0 \end{cases}$$

$$\mathit{read}((a_1, \ldots, a_n), i) := \begin{cases} a_i, & \text{falls } 1 \leq i \leq n \\ \text{undefiniert}, & \text{sonst} \end{cases}$$

$$\mathit{write}((a_1, \ldots, a_n), i, x) := \begin{cases} (a_1, \ldots, a_{i-1}, x, a_{i+1}, \ldots, a_n), & \text{falls } 1 \leq i \leq n \\ \text{undefiniert}, & \text{sonst} \end{cases}$$

$$\mathit{length}((a_1, \ldots, a_n)) := n$$

Dynamische Arrays

Implementierung: Wir erweitern die Arrayimplementierung von Stacks in der offensichtlichen Weise, gleich mit Verdoppelungsstrategie.

Kosten: *empty*, *read*, *write*, *length*: $O(1)$.
addLast: k *addLast*-Operationen kosten Zeit $O(k)$.
removeLast: Einfache Version (Pegel dekrementieren): $O(1)$.

Verfeinerung (auch bei Stacks möglich): **Halbierung**, wenn Array (Länge m) zu groß für die Anzahl n der Einträge wird, z. B. wenn durch eine *removeLast*-Operation $n < \frac{1}{4} \cdot m$ wird.
Nachteil: Diese Operation kostet $\Theta(n)$ Zeit.
Vorteil: Umfang der Datenstruktur ist **stets** $\leq 4n$, für aktuelle Arraylänge n.

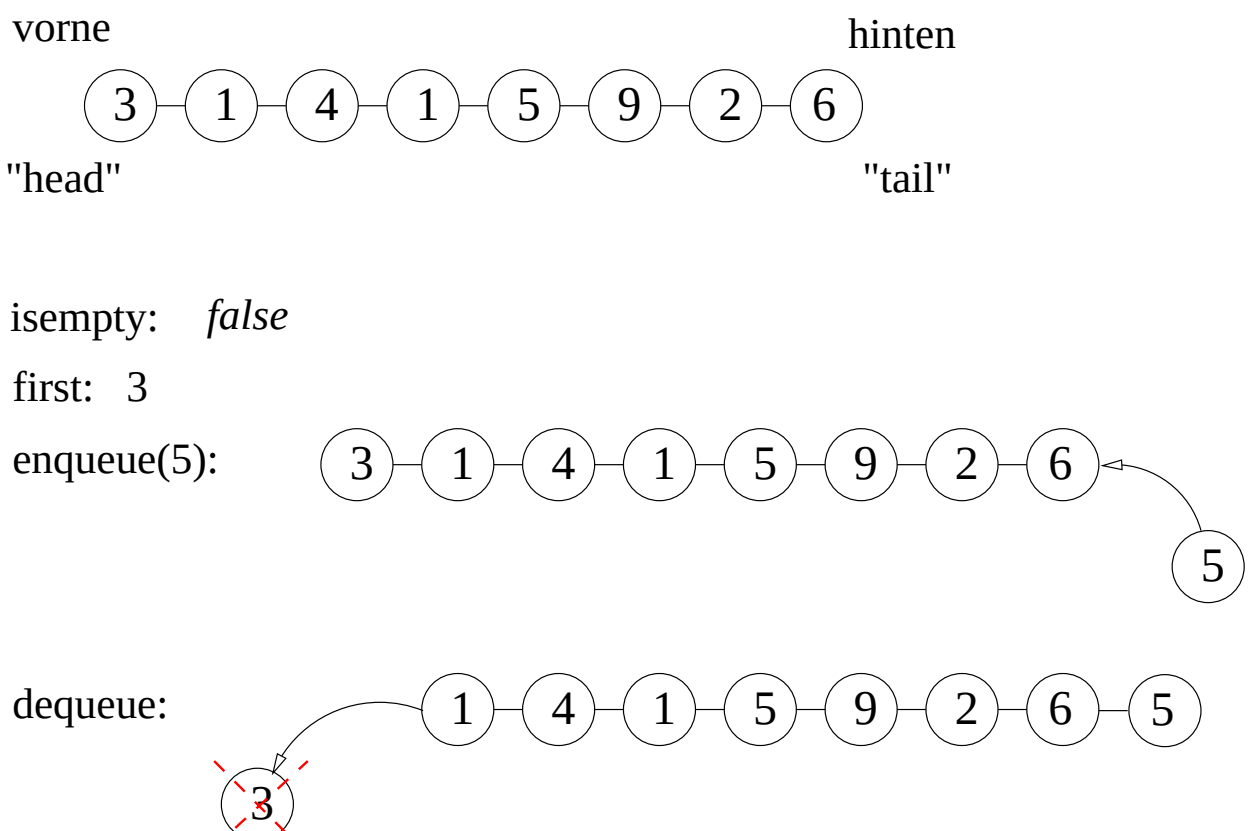
Mitteilung

Startend mit einem leeren dynamischen Array, kosten k *addLast*- und *removeLast*-Operationen bei der Arrayimplementierung mit Verdoppelung und Halbierung insgesamt Zeit $O(k)$.
Der Beweis benutzt etwas fortgeschrittenere Techniken zur „amortisierten Analyse“.

2.2 Queues (Warteschlangen, FIFO*-Listen)

Beispiel:

* **F**irst-**I**n-**F**irst-**O**ut-Listen



Spezifikation des Datentyps „Queue“ über D

1. Signatur:

Sorten:	*Elements*
	Queues
	Boolean
Operationen:	*empty*: $\rightarrow$ *Queues*
	enqueue: *Queues* $\times$ *Elements* $\rightarrow$ *Queues*
	dequeue: *Queues* $\rightarrow$ *Queues*
	first: *Queues* $\rightarrow$ *Elements*
	isempty: *Queues* $\rightarrow$ *Boolean*

Beachte: Die Signatur ist **identisch** zur **Signatur von Stacks** – bis auf Umbenennungen.
Rein syntaktische Vorschriften! Verhalten (noch) ungeklärt!

Spezifikation des Datentyps „Queue“ über D

2. Mathematisches Modell

Sorten: *Elements*: (nichtleere) Menge D (Parameter)
Queues: Seq(D)
Boolean: $\{true, false\}$

Die leere Folge $()$ stellt die leere Warteschlange dar.

Spezifikation des Datentyps „Queue“ über D

2. Mathematisches Modell (Forts.)

Operationen:

$$\mathit{empty}() := ()$$

$$\mathit{enqueue}((a_1, \ldots, a_n), x) := (a_1, \ldots, a_n, x)$$

$$\mathit{dequeue}((a_1, \ldots, a_n)) := \begin{cases} (a_2, \ldots, a_n), & \text{falls } n \geq 1 \\ \text{undefiniert}, & \text{falls } n = 0 \end{cases}$$

$$\mathit{first}((a_1, \ldots, a_n)) := \begin{cases} a_1, & \text{falls } n \geq 1 \\ \text{undefiniert}, & \text{falls } n = 0 \end{cases}$$

$$\mathit{isempty}((a_1, \ldots, a_n)) := \begin{cases} \mathit{false}, & \text{falls } n \geq 1 \\ \mathit{true}, & \text{falls } n = 0 \end{cases}$$

Spezifikation des ADTs „Queue“ über D

Alternative: (siehe [AuP], Kap. 8 und [Saake/Sattler], Kap. 11)

„1. Signatur“: wie oben.

2. Axiome:

$\forall\, q$: *Queue*, $\forall\, x, y$: *Elements*

$$\mathit{dequeue}(\mathit{enqueue}(\mathit{empty}(), x)) = \mathit{empty}()$$

$$\mathit{first}(\mathit{enqueue}(\mathit{empty}(), x)) = x$$

$$\mathit{dequeue}(\mathit{enqueue}(\mathit{enqueue}(q, x), y)) = \mathit{enqueue}(\mathit{dequeue}(\mathit{enqueue}(q, x)), y)$$

$$\mathit{first}(\mathit{enqueue}(\mathit{enqueue}(q, x), y)) = \mathit{first}(\mathit{enqueue}(q, x))$$

$$\mathit{isempty}(\mathit{empty}()) = \mathit{true}$$

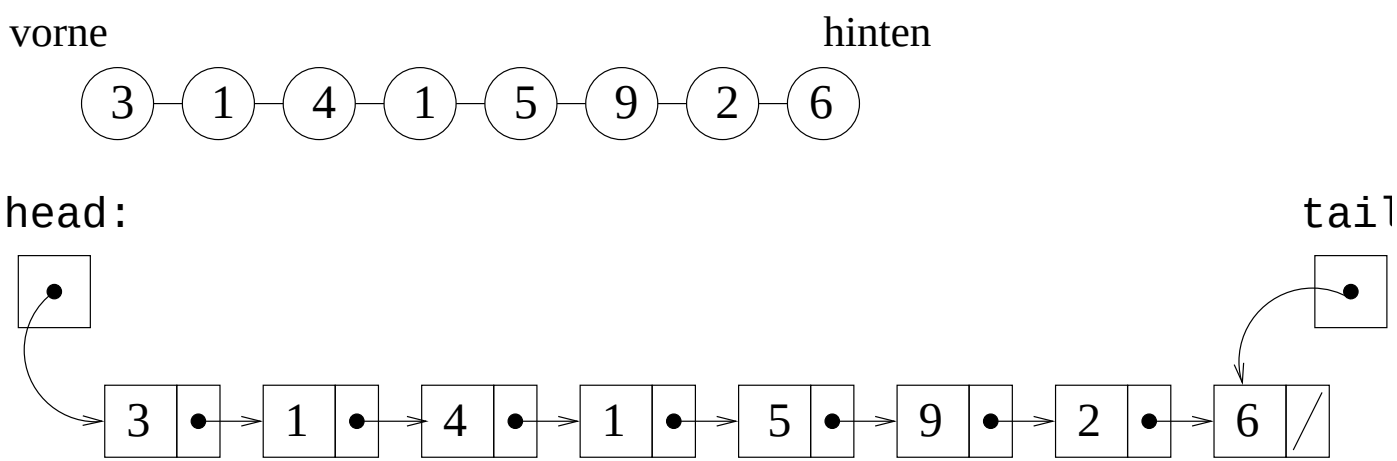
$$\mathit{isempty}(\mathit{enqueue}(q, x)) = \mathit{false}$$

Kommentar

- Das mathematische Modell ist fast identisch zu dem für Stacks. Nur fügt *enqueue*(x) das Element x hinten an die Folge $(a_1, \ldots, a_n)$ an, während *push*(x) das Element x vorne anfügt.
- Wir sagen noch etwas zur Spezifikation mit Axiomen. Nächste Folie: Sechs Axiome für „Queues“.
- Die ersten beiden Axiome beschreiben das Verhalten, wenn man an eine leere Queue ein Element anhängt: Entnehmen des ersten Elements liefert die leere Queue, Lesen des ersten Elements liefert, was gerade angehängt wurde.
- Die mittleren Axiome beschreiben das Ergebnis q'', wenn man an eine nichtleere Queue $q' =$*enqueue*(q, x) ein Element y anhängt. Der erste Eintrag von q'' ist einfach der erste Eintrag von q'. Entnehmen des ersten Eintrags aus q'' liefert dieselbe Queue, die sich ergibt, wenn man erst aus q' den ersten Eintrag entnimmt und dann y anhängt.
- Die letzten beiden Axiome beschreiben, wie bei Stacks, das korrekte Verhalten von *isempty*.
- Die Axiome sind natürlich, wenn man sie liest und überlegt, was sie bedeuten, aber es gibt zwei Fragen, für Queues und für allgemeine (neue) Datentypen:
  - Wie findet man geeignete Axiome?
  - Woher weiß man, dass die Axiome den geplanten Datentyp komplett beschreiben?
- Diese Fragen stellen sich für jeden neuen Datentyp neu und sind i. a. nicht leicht zu beantworten.

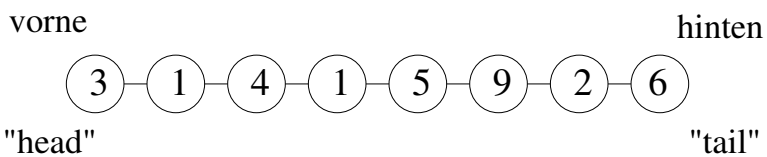
Listenimplementierung von Queues

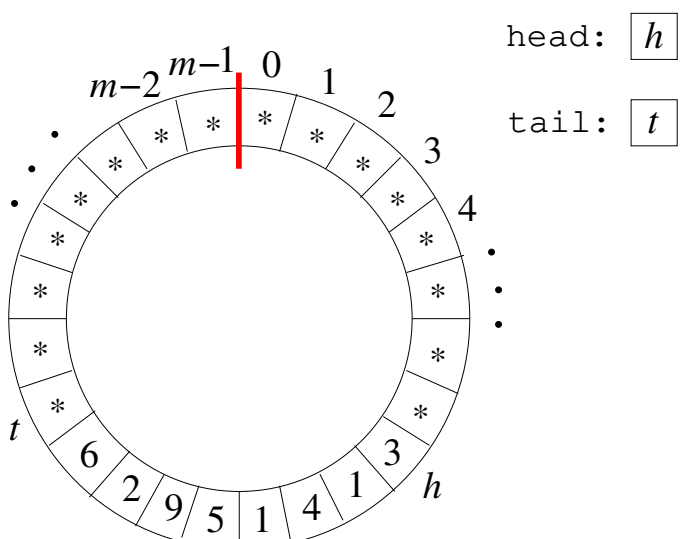
Implementierung mittels einfach verketteter Listen
mit Listenendezeiger/-referenz: Skizze.



Zu Übung überlege man sich, wie die Operationen zu implementieren wären.
Etwas knifflig: Umgang mit der Situation, dass Liste leer wird. (Übung!)

Arrayimplementierung von Queues





Arrayimplementierung von Queues (Forts.)

Implementierung mittels eines Arrays, „zirkulär“ benutzt:
Das Queueobjekt besteht aus einem Array $\texttt{A}[0\,..\,m-1]$ und zwie Integervariablen `head` und `tail`. Man stellt sich das Array $\texttt{A}[0\,..\,m-1]$ zu einem Ring gebogen vor, so dass man Anfang und Ende zusammenkleben kann (im Bild: rote Linie). Durch den Inhalt h von `head` und den Inhalt t von `tail` sind in `A` zwei Positionen bestimmt. Die Idee ist, dass die Einträge $a_1, \ldots, a_n$ der Queue in den Positionen

$$\texttt{A}[h], \texttt{A}[h+1], \ldots, \texttt{A}[t-1]$$

des Arrays stehen (vgl. Bild). Dabei werden die Indizes modulo m gerechnet: auf $m-1$ folgt 0.
Die Queue ist leer genau dann wenn $h = t$ gilt.

Die Zellen $\texttt{A}[t], \texttt{A}[t+1], \ldots, \texttt{A}[h-1]$ sind leer (im Bild mit $*$ markiert). Mindestens $\texttt{A}[t]$ ist leer. D. h.: Die maximale Anzahl n der Einträge im Array **(die Kapazität)** ist $m-1$.

Um x einzufügen, wird (sofern Platz ist), x nach $\texttt{A}[t]$ geschrieben und dann `tail` um 1 erhöht.

Ein Überlauf (oder eine Verdopplung) wird ausgelöst, wenn beim Einfügen die Variable `tail` denselben Wert erhalten würde wie `head`. Dieses Kriterium macht es überflüssig, die Beladung n mitzuführen. – Wir beschreiben die Operationen im Einzelnen.

Implementierung der Operationen auf einer Queue q:

`q.empty()`: // Konstruktor; Option: m als Argument
  Erzeuge Array `A[`$0..m-1$`] of elements` und zwei Variable `head, tail: int` ;
  `head` $\leftarrow 0$; `tail` $\leftarrow 0$;

`q.isempty`:
  if `head = tail` **then return** „$true$“ **else return** „$false$“ ;

`q.first`:
  if `head = tail` **then „Fehler“** (z.B. *QEmptyException*)
  else return `A[head]` ;

`q.dequeue`:
  if `head = tail` **then „Fehler“** (z.B. *QEmptyException*)
  else `head` $\leftarrow (\texttt{head} + 1) \bmod m$;

`q.enqueue(`x`)`:
  `tt` $\leftarrow (\texttt{tail} + 1) \bmod m$;
  if `tt` $=$ `head` **then „Überlauf“** (z.B. *QOverflowException* (oder Verdopplung))
    else `A[tail]` $\leftarrow x$; `tail` $\leftarrow$ `tt` ;

Satz 2.2.1

Die angegebene Arrayimplementierung einer Queue über D ist korrekt, ohne Verdoppelungsstrategie, solange kein Fehler im Modell und kein Überlauf im Array eintritt; mit Verdoppelungsstrategie, solange kein Fehler im Modell und kein Speicherüberlauf auftritt.

„Korrektheit“ heißt natürlich, genau wie bei Stacks: Das Ein-/Ausgabeverhalten der Implementierung ist das gleiche wie beim mathematischen Modell.

Beweis: Betrachte eine Folge $\mathrm{Op}_0 = \mathit{empty}, \mathrm{Op}_1, \ldots, \mathrm{Op}_N$ von Queueoperationen, wobei *empty* nur ganz am Anfang vorkommen darf. Zeige dann durch Induktion über $i = 0, \ldots, N$ unter der Annahme, dass **im mathematischen Modell kein Fehler** auftritt, folgende **Behauptung** $(\mathrm{IB})_i$:
Wenn nach Operationen $\mathrm{Op}_0, \mathrm{Op}_1, \ldots, \mathrm{Op}_i$ der Inhalt der Queue $q_i = (a_1, \ldots, a_{n_i})$ ist, dann stehen nach Ausführung der Operationen in der Implementierung die Elemente

$$a_1, \ldots, a_{n_i} \text{ in den Positionen } \texttt{A[}h\texttt{]}, \texttt{A[}h+1\texttt{]}, \ldots, \texttt{A[}t-1\texttt{]}$$

(Indizes modulo m gerechnet), wobei h der Inhalt von `head` und t der Inhalt von `tail` ist.
Insbesondere ist die Warteschlange leer genau dann wenn $h = t$ ist.

Satz 2.2.2

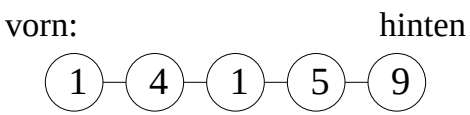
Bei einer Queue mit Arrays und der Verdoppelungsstrategie sind die Gesamtkosten für N Operationen $O(N)$.
Platzbedarf: $O(k)$, wenn k die maximale je erreichte Länge der Queue ist.

Beweise: Analog zu Stacks.

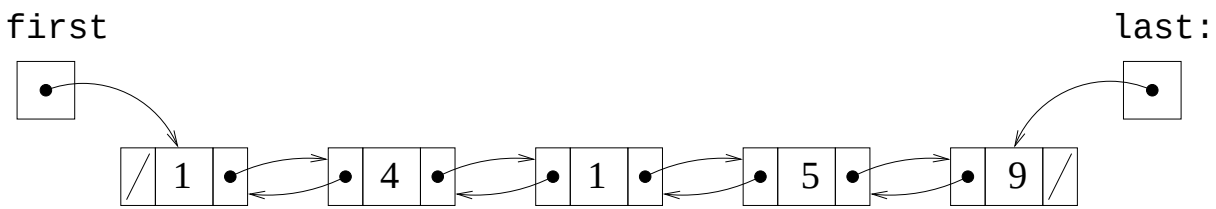
Datentyp Doppelschlange – Deque

Schlange mit zwei Enden; Einfügen und Entnehmen an beiden Enden möglich.

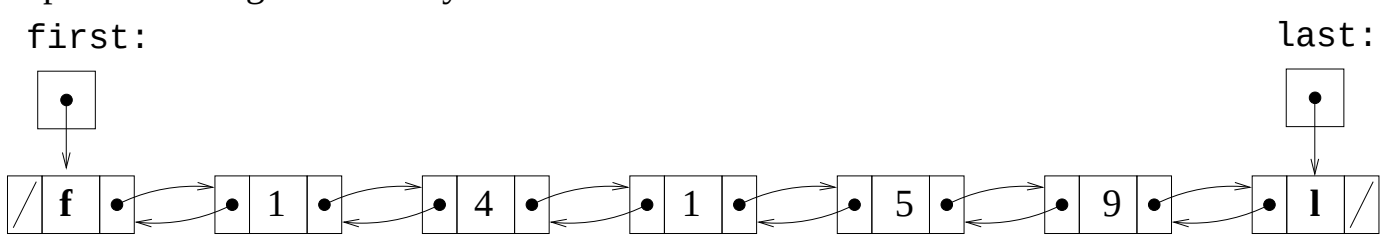
Schema:



Einfache Implementierung:



Implementierung mit Dummyelementen:



1. Signatur:

Sorten:	*Elements*	*Operationen*:	*empty*: . . .
	Deques		*pushFront*: . . .
	Boolean		*popFront*: . . .
			first: . . .
			pushBack: . . .
			popBack: . . .
			last: . . .
			isempty: . . .

Details
2. Mathematisches Modell
Implementierung mit Arrays
Implementierung mit doppelt verketteten Listen

} Übung.

PAUSE

Kommentar

- Ein neuer Datentyp, geliehen aus der Mathematik: „Endliche Mengen“.
- Wie üblich, gibt es eine Grundmenge D, die endlich oder unendlich sein kann. Denken Sie an natürliche oder ganze Zahlen, an Strings beliebiger Länge, aber auch an die Menge aller 64-Bit-Wörter.
- Anhand von Mengen diskutieren wir Techniken, die später für den sehr wichtigen allgemeineren Datentyp „Wörterbuch“ (alias „assoziatives Array“) verwendet werden.
- Zunächst gibt es nur sehr einfache Operationen. Man kann eine Menge S erzeugen (die als leere Menge initialisiert wird), man kann ein Element $x \in U$ zu S hinzufügen, man kann ein Element aus S streichen, und man kann ein beliebiges $x \in U$ darauf testen, ob es Element von S ist oder nicht.
- Kompliziertere Situationen folgen später.
- Durch eine Folge solcher Operationen können nur endliche Mengen S entstehen.

2.3 Einfache Datenstrukturen für Mengen

Datentyp: Endliche Menge über D.

Intuitive Aufgabe:

Speichere eine (i. a. veränderliche) endliche Menge $S \subseteq D$, mit Operationen:

Initialisierung:	$S \leftarrow \emptyset$;	// anfangs ist S leer
Suche:	$x \in S$?;	// für $x \in U$ teste, ob x Element von S ist
Hinzufügen:	$S \leftarrow S \cup \{x\}$;	// füge $x \in U$ zu S hinzu
Löschen:	$S \leftarrow S - \{x\}$.	// entferne $x \in U$ aus S

Datentyp Dynamische Menge: DynSet

1. Signatur:

Sorten:	*Elements*
	Sets
	Boolean
Operationen:	$empty: \rightarrow Sets$
	$insert: Sets \times Elements \rightarrow Sets$
	$delete: Sets \times Elements \rightarrow Sets$
	$member: Sets \times Elements \rightarrow Boolean$

Datentyp Dynamische Menge: DynSet

2. Mathematisches Modell:

Sorten:

Elements: (nichtleere) Menge D **(Parameter)**

Sets: $\mathcal{P}^{<\infty}(D) = \{S \subseteq D \mid S \text{ endlich}\}$

Boolean: $\{true, false\}$

Operationen:

$empty() := \emptyset$

$insert(S, x) := S \cup \{x\}$

$delete(S, x) := S - \{x\}$

$$member(S, x) := \begin{cases} true, & \text{falls } x \in S \\ false, & \text{falls } x \notin S \end{cases}$$

Datentyp Dynamische Menge: DynSet

Implementierungsmöglichkeiten:

(A) Einfach verkettete Liste oder Array **mit Wiederholung**

(B) Einfach verkettete Liste oder Array **ohne Wiederholung**

(C) Einfach verkettete Liste oder Array, **aufsteigend sortiert**

Nur möglich, wenn es auf D eine totale Ordnung $<$ gibt.

Später:

(D) Suchbäume

(E) Hashtabellen

Zusammenfassung

(A) Einfach verkettete Liste oder Array **mit Wiederholung**:

Initialisierung, Einfügen in Zeit $O(1)$.

Platzbedarf, Länge der Liste: $h = O(\#(\text{bisherige Einfügungen}))$.

Suchen, Löschen in Zeit $\Theta(h)$.

(B) Einfach verkettete Liste oder Array **ohne Wiederholung**

$n = |S| = \#(\text{Einträge})$.

Platzbedarf: $O(n)$.

Einfügen, Suchen, Löschen in Zeit $\Theta(n)$.

Achtung: Wegen der hohen Suchzeiten sind lineare Listen und Arrays als grundsätzlicher Ansatz zur Implementierung des Mengen-Datentyps DynSet **nicht geeignet** (außer es handelt sich um sehr kleine Strukturen).

(C) Array, ohne Wiederholung, **aufsteigend sortiert**:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A:	2	5	7	10	17	19	26	50	54	59	67	$*$	$*$	$*$	$*$

„Pegelvariable“ n enthält $n = \#(\text{Einträge})$, hier $n = 11$.

Nachteil:
Bei *insert*, *delete* ist **Verschieben** vieler Elemente nötig, um die Ordnung zu erhalten und dabei das Entstehen von Lücken zu vermeiden.

Zeitaufwand für diese Operationen: $\Theta(n) = \Theta(|S|)$.

Vorteil:

member: Kann **binäre Suche** benutzen.

Binäre Suche, rekursiv

Aufgabe: Suche Objekt $x \in D$ in Teilarray $\texttt{A}[a\,..\,b]$

Überlege:

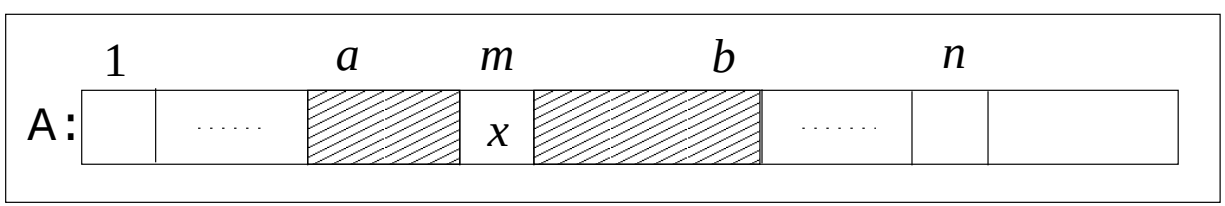
0. Falls $b < a$, ist x nicht im Teilarray.

1. Falls $a = b$, ist x in $\texttt{A}[a\,..\,b]$ genau dann wenn $x = \texttt{A}[a]$.

2. Falls $a < b$, berechne $m = a + \lfloor (b-a)/2 \rfloor$ // Mitte des Teilarrays

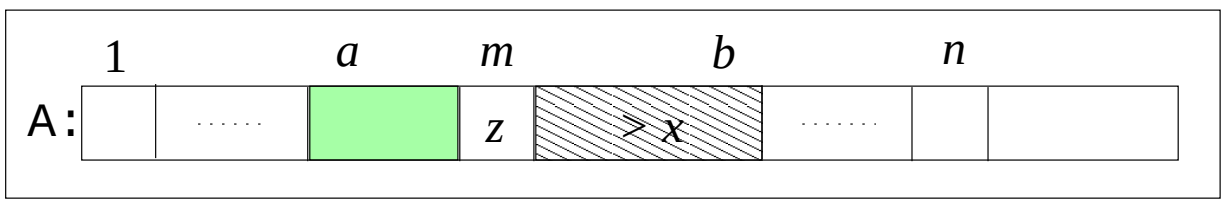
3 Fälle:

$x = \texttt{A}[m]$: x gefunden, Antwort $true$.



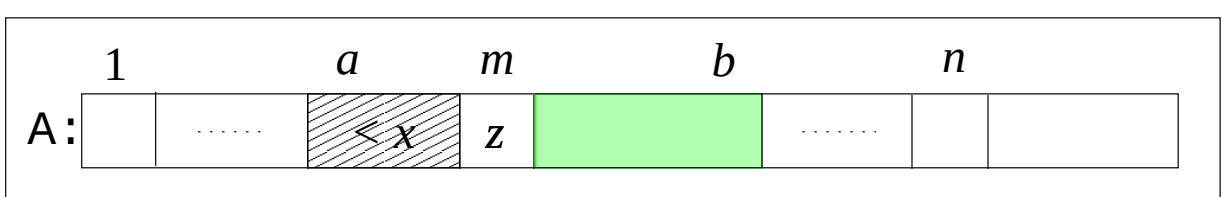
Binäre Suche, rekursiv

$x < \texttt{A}[m]$: Wende **(rekursiv)** binäre Suche auf $\texttt{A}[a\,..\,m-1]$ an.



$x < z$

$x > \texttt{A}[m]$: Wende **(rekursiv)** binäre Suche auf $\texttt{A}[m+1\,..\,b]$ an.



$z < x$

Um x im Gesamtarray $\texttt{A}[1\,..\,n]$ zu suchen, wenn $n \geq 1$:
Binäre Suche nach x in $\texttt{A}[1\,..\,n]$.

Gegeben: Array `A[1..n]`, x: `elements`; (unveränderlich, global bekannt)

Prozedur RecBinSearch(a, b) // **rekursiv!**
Eingabe: a, b: `int`; // $1 \leq a$ und $b \leq n$
// Sucht global bekanntes x unter den `A[`i`]`, $a \leq i \leq b$

(1) **if** $b < a$ **then return** $false$;
(2) **if** $a = b$ **then return** $(x =$ `A[`a`]`$)$; //boolescher Wert
(3) `m` $\leftarrow a + (b - a)$ `div` 2; //ganzzahlige Division
(4) **if** $x =$ `A[m]` **then return** $true$;
(5) **if** $x <$ `A[m]` **then return RecBinSearch**$(a,$`m`$ - 1)$;
(6) **if** $x >$ `A[m]` **then return RecBinSearch**$($`m`$ + 1, b)$.

Algorithmus Binäre Suche (rekursiv)
Eingabe: Array `A[1..n]`; x: `elements`;
mit `A[1]` $< \cdots <$ `A[`n`]`;
(1) **return RecBinSearch**$(1, n)$.

Proposition 2.3.1

Es gelte $1 \leq a$ und $b \leq n$, und das Teilarray $\texttt{A}[a..b]$ von $\texttt{A}[1..n]$ sei strikt aufsteigend sortiert. Dann gilt: Der Aufruf **RecBinSearch**(a, b) liefert das Resultat $true$, falls x in $\texttt{A}[a..b]$ vorkommt, $false$, falls nicht.

Satz 2.3.2

Binäre Suche angewendet auf $\texttt{A}[1..n]$ und x liefert $true$, falls x in $\texttt{A}[1..n]$ vorkommt, $false$, falls nicht. (Folgt sofort aus der Proposition.)

Beweis von Prop. 2.3.1: Setze $i := \max\{b - a + 1, 0\}$, die Länge des betrachteten Teilarrays.

Wir benutzen **verallgemeinerte Induktion** über i.

I.A.: Fall $i = 0$. D. h. $b < a$, d. h. das Teilarray $\texttt{A}[a..b]$ ist leer.
Die Antwort $false$, die in (Zeile (1)) ausgegeben wird, ist also korrekt.
Fall $i = 1$: Dann ist $1 \leq a = b \leq n$, und die Ausgabe $(x = \texttt{A}[a])$, die in (Zeile (2)) ausgegeben wird, ist korrekt.

Sei nun $i > 1$, also $1 \leq a < b \leq n$.

I.V.: Prop. 2.3.1 gilt für Aufrufe **RecBinSearch**(c, d) mit $d - c + 1 < i$.

Ind.-Schritt: Es wird $m = a + \lfloor (b-a)/2 \rfloor = \lfloor (a+b)/2 \rfloor$ berechnet.

Beobachte: $(*)$ $a \leq m < b$. (Wir haben $a < (a+b)/2 < b$.)

Fall 1: $x =$ `A[`m`]`. – Die in Zeile (4) erzeugte Ausgabe $true$ ist korrekt.

Fall 2: $x <$ `A[`m`]`. – Analog zu Fall 3.

Fall 3: $x >$ `A[`m`]`. – Beobachte: x kommt auf keinen Fall in `A[`$a..m$`]` vor, denn für $a \leq i \leq m$ gilt `A[`i`]` $\leq$ `A[`m`]` $< x$, weil das Array sortiert ist.

Also kommt x in `A[`$a..b$`]` vor genau dann wenn x in `A[`$m+1..b$`]` vorkommt.

Die Länge dieses Abschnittes ist $b - m \overset{(*)}{\leq} b - a < i$. Nach I.V. liefert der rekursive Aufruf **RecBinSearch**$(m+1, b)$ in Zeile (5) das korrekte Resultat für diesen Abschnitt, also auch insgesamt. □

Zur **Rechenzeit** von Binärer Suche:

Aus dem Korrektheitsbeweis folgt, dass die rekursive Prozedur immer terminiert. Aber wie lange dauert es?

$\boldsymbol{T_{RBS}(i)}$ seien die **worst-case-Kosten** für einen Aufruf **RecBinSearch**(a, b) mit $i = b - a + 1$ (maximiert über alle möglichen Arrays $\mathtt{A}[a\,..\,b]$ und Indexpaare (a, b)).

Jeder Aufruf von **RecBinSearch** verursacht Kosten $\leq C$ für eine Konstante C, wenn man die Kosten der hiervon ausgelösten rekursiven Aufrufe nicht zählt.

Gesucht also, für alle i:

$r(i) :=$ maximale Anzahl der Aufrufe (inklusive des ersten Aufrufs), wenn $b - a + 1 = i$.

Klar nach Zeilen (1), (2) von **RecBinSearch**: $r(0) = r(1) = 1$.

Für $i \geq 2$: Die rekursiven Aufrufe in Zeilen (5) bzw. (6) beziehen sich auf Teilarrays der Länge $\lfloor (i-1)/2 \rfloor$ bzw. $\lceil (i-1)/2 \rceil$.
$\Rightarrow$ **„Rekurrenzungleichung“:**

$$r(i) \leq 1 + \max\{r(\lfloor (i-1)/2 \rfloor), r(\lceil (i-1)/2 \rceil)\}.$$

Behauptung: $r(i) \leq 1 + \log i$, für $i \geq 1$. (Wissen sowieso: $r(0) = 1$.)

Beweis durch verallgemeinerte Induktion über $i \geq 1$.

I.A.: Beh. stimmt für $i = 1$, weil $r(1) = 1$ und $\log 1 = 0$.

Nun sei $i \geq 2$.

I.V.: Beh. stimmt für alle $j \in \{1, \ldots, i-1\}$.

I.-Schritt: Die Rekurrenzungleichung sagt:

$$r(i) \leq 1 + \max\{r(\lfloor (i-1)/2 \rfloor), r(\lceil (i-1)/2 \rceil)\}.$$

Für $i = 2$ heißt das $r(2) \leq 1 + \max\{r(0), r(1)\} = 1 + 1 = 1 + \log 2$.

Falls $i \geq 3$, folgt mit der **I.V.**:

$$\begin{aligned} r(i) &\leq 1 + \max\{1 + \log(\lfloor (i-1)/2 \rfloor), 1 + \log(\lceil (i-1)/2 \rceil)\} \\ &\leq 2 + \log(i/2) = 2 + \log i - 1 = 1 + \log i \ . \end{aligned}$$

Das ist die Induktionsbehauptung. □

Satz 2.3.3

Der Algorithmus **Binäre Suche** benötigt auf Eingabe $\mathtt{A}[1\,..\,n]$ höchstens $1 + \lfloor \log n \rfloor$ rekursive Aufrufe und Kosten (Rechenzeit) $O(\log n)$.

Beispiel: Binäre Suche in Array der Größe $10\,000\,000$ erfordert nicht mehr als $1 + \lfloor \log_2(10^7) \rfloor = 24$ rekursive Aufrufe. (Extrem wenig im Vergleich zu linearer Zeit in Fällen (A) und (B)!)

PAUSE

Iterative binäre Suche in schwach geordneten Arrays

Beispiel: Array `A[1..11]` schwach aufsteigend sortiert, Suche nach x.
Ausgabe $i(x)$: „Position“ von Suchschlüssel x.

	1	2	3	4	5	6	7	8	9	10	11=n
A:	4	4	7	7	7	9	9	10	12	15	15

$i(7) = 3$ erstes Vorkommen von x

$i(8) = 6$ Position des ersten Eintrags größer als x

$i(20) = 12$ x ist größer als alle Arrayeinträge

Allgemein:

Iterative binäre Suche in schwach geordneten Arrays

Variante der Aufgabenstellung „suche x in Array $\texttt{A}[1..n]$“ (häufig in Anwendungen): $\texttt{A}[1..n]$ enthält eventuell Einträge mehrfach und ist nur schwach aufsteigend sortiert, d. h. es gilt $\texttt{A}[1] \leq \cdots \leq \texttt{A}[n]$.

Die Ausgabe ist informativer als die bisher betrachtete (von der *member*-Operation verlangte) $true/false$-Entscheidung. Sie besteht aus zwei Teilen w und $i(x)$.

Der Wahrheitswert $w \in \{true, false\}$ hat dieselbe Bedeutung wie bisher. Zudem definieren wir:

$$i(x) := \min(\{i \mid 1 \leq i \leq n, x \leq \texttt{A}[i]\} \cup \{n+1\}).$$

Wenn x im Array vorkommt, ist dies die Position des ersten (am weitesten links stehenden) Vorkommens von x.
Wenn x nicht vorkommt, ist es die Position des ersten Eintrags, der größer als x ist.
Wenn kein Eintrag im Array größer als x ist, ist $i(x) = n + 1$.

Iterative binäre Suche in schwach geordneten Arrays

Wir ändern auch die Algorithmennotation. (Unabhängig von der geänderten Aufgabe!)

In einer einfachen Schleife (ohne Rekursion) wird die Position $i(x)$ bestimmt.
Danach wird in einem „=“-Vergleich entschieden, ob x vorkommt.

Algorithmus Binäre Suche (iterativ)
Eingabe: Array `A[1..n]`, x: `elements`;
mit `A[1]` $\leq \cdots \leq$ `A[n]`.

(1) `a` $\leftarrow 1$; `b` $\leftarrow n$;
(2) **while** `a` $<$ `b` **do**
(3) `m` $\leftarrow$ `a + (b-a) div 2` ;
(4) **if** $x \leq$ `A[m]` **then** `b` $\leftarrow$ `m` **else** `a` $\leftarrow$ `m` $+ 1$;
(5) **if** $x >$ `A[a]` **then return** $(\mathit{false}$, `a` $+ 1)$;
(6) **if** $x =$ `A[a]` **then return** $(\mathit{true}$, `a`$)$
(7) **else return** $(\mathit{false}$, `a`$)$.

Satz 2.3.4

Der Algorithmus Binäre Suche (iterativ) auf einem schwach aufsteigend geordneten Array `A[1..`n`]` liefert das korrekte Ergebnis (wie oben beschrieben).

Die Anzahl der Durchläufe durch den Rumpf der Schleife ist höchstens $\lceil \log n \rceil$.

Die Laufzeit ist $\Theta(\log n)$ im schlechtesten und im besten Fall.

Beweis: Übung.

Vorsicht! Die Korrektheit ist alles andere als offensichtlich. Sorgfältiges Argumentieren ist nötig.

Mehrere Mengen: Datentyp DynSets

Intuitive Aufgabe: Speichere **mehrere** Mengen $S_1, \ldots, S_r \subseteq D$, so dass für jede einzelne von ihnen die Operationen des Datentyps **Dynamische Menge** ausführbar sind und zudem

Vereinigung, **Durchschnitt**, **Differenz**, **symmetrische Differenz**, **Leerheitstest**

von beliebigen dieser Mengen.

(Vereinigung: $S \cup S'$; Durchschnitt: $S \cap S'$; Differenz: $S - S'$;
symmetrische Differenz: $S \oplus S' = S \triangle S' = (S - S') \cup (S' - S)$; Leerheitstest: Ist $S = \emptyset$?)

Mehrere Mengen: Datentyp DynSets

Signatur: Wie *Dynamische Menge*, zusätzlich Sorte *Boolean* sowie:

union : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
intersection : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
diff : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
symdiff : *Sets* $\times$ *Sets* $\rightarrow$ *Sets*
isempty : *Sets* $\rightarrow$ *Boolean*

Mathematisches Modell:
Sorten und Operationen wie *Dynamische Menge*, und zusätzlich $\{true, false\}$ als *Boolean* und

union$(S_1, S_2) := S_1 \cup S_2$
intersection$(S_1, S_2) := S_1 \cap S_2$
diff$(S_1, S_2) := S_1 - S_2$
symdiff$(S_1, S_2) := (S_1 \cup S_2) - (S_1 \cap S_2)$

$$\textit{isempty}(S) := \begin{cases} true, & \text{falls } S = \emptyset \\ false, & \text{sonst.} \end{cases}$$

Mehrere Mengen: Datentyp DynSets

Weitere interessante Operationen auf Mengen erhält man durch Kombination:

Gleichheitstest: Ist $S_1 = S_2$?

$\mathit{equal}(S_1, S_2) = \mathit{isempty}(\mathit{symdiff}(S_1, S_2))$

Teilmengentest: Ist $S_1 \subseteq S_2$?

$\mathit{subset}(S_1, S_2) = \mathit{isempty}(\mathit{diff}(S_1, S_2))$

Disjunktheitstest: Ist $S_1 \cap S_2 = \emptyset$?

$\mathit{disjoint}(S_1, S_2) = \mathit{isempty}(\mathit{intersection}(S_1, S_2))$

Natürlich könnte man diese Operationen auch in die Spezifikation aufnehmen.

Implementierung des Datentyps *DynSets*

Einfach: Ungeordnete Strukturen (Liste/Array) **mit** oder **ohne** Wiederholung.

Implementierung von *isempty*: teste, ob Liste leer bzw. Pegelstand gleich 0.

union(S_1, S_2): **Mit Wiederholung**: Darstellung von S_i hat Länge h_i, $i = 1, 2$.
Bei (einfach verketteten) Listen, mit Zeiger auf Listenende: **Kosten:** $O(1)$
Bei Arrays muss man beide in ein neues Array kopieren **Kosten:** $\Theta(h_1 + h_2)$.

Für *intersection*(S_1, S_2), *diff*(S_1, S_2), *symdiff*(S_1, S_2): **Mit Wiederholung**:
Für *jedes* Element der Darstellung von S_1 durchsuche Darstellung von S_2.
Kosten: $\Theta(h_1 \cdot h_2)$.

Ohne Wiederholung: Darstellung von S_i hat Länge $n_i = |S_i|$.
Alle Operationen haben **Kosten:** $\Theta(n_1 \cdot n_2)$.

Implementierung des Datentyps *DynSets*

Geschicktere Darstellung für Mengen: (C) Aufsteigend sortierte Arrays/Listen.

Implementierung der Operationen
union(S_1, S_2), *intersection*(S_1, S_2), *diff* (S_1, S_2), *symdiff* (S_1, S_2) durch

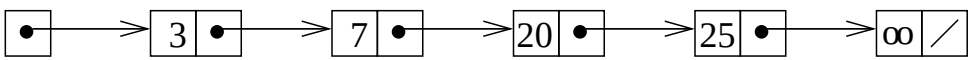
quasiparallelen Durchlauf durch zwei Arrays/Listen, analog zum Mischen (**Merge**) bei **Mergesort** (Vgl. [AuP], Kapitel 7, oder diese Vorlesung, Kap. 6).

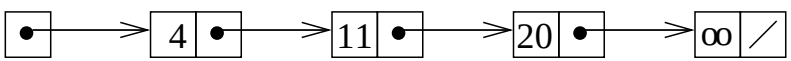
Beginnend mit den beiden ersten Einträgen der Listen bestimmt man immer den kleineren Eintrag der beiden „aktuellen“ Einträge und überträgt diesen in die Ergebnisliste.
Bei zwei gleichen Einträgen in den beiden Listen wird nur eine Kopie in das Ergebnis übernommen.

Für die anderen Operationen: analog. Bei Arrays: Analog.

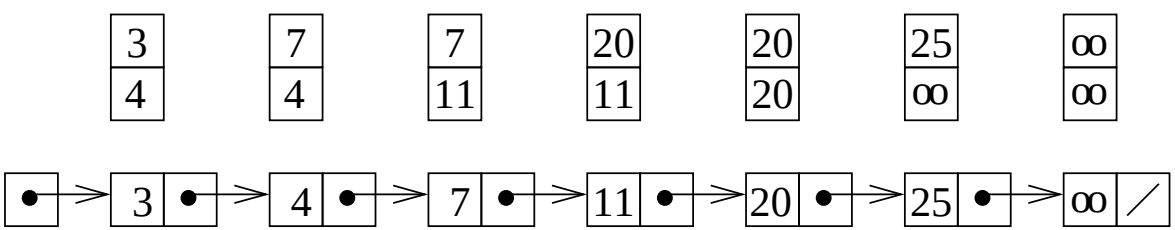
Implementierung des Datentyps *DynSets*

Hier: Mit uneigentlichem „Wächterelement“ (engl.: „sentinel“), das ist ein Eintrag ∞, größer als alle Elemente von U. – Eingabe: Listen für S_1 und S_2.





Ausgeführte Vergleiche und Ergebnisliste bei *union*(S_1, S_2):



Bei Array-/Listenlängen $|S_1| = n_1$, $|S_2| = n_2$: **Kosten:** $\Theta(n_1 + n_2)$.

Für „kleine“ Mengen: Bitvektor-Darstellung

Grundmenge D ist gegeben als konstantes Tupel $(x_1, \ldots, x_N)$

Wir identifizieren x_i mit i, d.h. $D = \{1, \ldots, N\}$.

Beispiel: In Pascal:

```
type
Farbe = (rot, gruen, blau, gelb, lila, orange);
Palette = set of Farbe;
```

Hier: $N = 6$.

`rot` $\hat{=}$ 1, `gruen` $\hat{=}$ 2, `blau` $\hat{=}$ 3,
`gelb` $\hat{=}$ 4, `lila` $\hat{=}$ 5, `orange` $\hat{=}$ 6.

Für „kleine“ Mengen: Bitvektor-Darstellung

Repräsentation: Durch Array `A[`$1..N$`] of Boolean` wird

$$S = \{i \mid \texttt{A}[i] = 1\}$$

dargestellt.

Wie immer: 0 für $false$, 1 für $true$.

Beispiel: Darstellung der Menge $\{2, 3, 5\} \mathrel{\hat{=}} \{\texttt{gruen}, \texttt{blau}, \texttt{lila}\}$:

0	1	1	0	1	0

Platzbedarf:
N Bits, d. h. $\lceil N/8 \rceil$ Bytes oder $\lceil N/64 \rceil$ Speicherworte à 64 Bits.

Für „kleine“ Mengen: Bitvektor-Darstellung

empty: Lege Array `A[`$1..N$`]` an;
for `i` **from** 1 **to** N **do** `A[i]` $\leftarrow 0$; **Kosten:** $\boldsymbol{\Theta(N)}$

insert(S, i):
`A[`i`]` $\leftarrow 1$; **Kosten:** $\boldsymbol{O(1)}$

delete(S, i):
`A[`i`]` $\leftarrow 0$; **Kosten:** $\boldsymbol{O(1)}$

member(S, i):
return `A[`i`]`; **Kosten:** $\boldsymbol{O(1)}$

DynSets-Op.en *union*(S_1, S_2), *intersection*(S_1, S_2), *diff*(S_1, S_2), *symdiff*(S_1, S_2):
Jeweils paralleler Durchlauf durch zwei Arrays. **Kosten:** $\boldsymbol{\Theta(N)}$

isempty(S):
Durchlauf durch ein Array. **Kosten:** $\boldsymbol{\Theta(N)}$

Für „kleine“ Mengen: Bitvektor-Darstellung

Zusatz 1: Wenn man zur Bitvektor-Darstellung von S eine `int`-Variable hinzufügt, die $|S|$ enthält, dauert der Leerheitstest nur $O(1)$, die anderen Operationen nur um einen konstanten Faktor länger.

Übungsfrage: Wie ist die Implementierung von *empty*, *insert*, *delete*, *union*, *symdiff* zu modifizieren, wenn ein solcher Zähler mitgeführt wird?

Zusatz 2: Im Fall der Bitvektor-Darstellung lässt sich auch die **Komplement**-Operation *compl*: *Sets* $\to$ *Sets* mit der Semantik *compl*$(S) = U - S$ (im math. Modell) leicht implementieren. **Kosten:** $\boldsymbol{\Theta(N)}$

Zusatz 3: Eine interessante Variante: Packe 64 Bits in ein Wort (lange vorzeichenlose Zahl). Dann nimmt jedes Element von U nur 1 Bit im Speicher in Anspruch.
(Übung: Programmiere die Operationen.)

Kosten für *DynSets*-Operationen, einfache Implementierungen:

	BV	AmW	LmW	{A,L}oW	A-sort	L-sort
empty	$\Theta(N)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
insert	$\boldsymbol{O(1)}$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
delete	$\boldsymbol{O(1)}$	$O(h)$	$O(h)$	$O(n)$	$O(n)$	$O(n)$
member	$\boldsymbol{O(1)}$	$O(h)$	$O(h)$	$O(n)$	$\boldsymbol{O(\log n)}$	$O(n)$
isempty	$\boldsymbol{O(1)}$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
union	$\Theta(N)$	$O(h_1+h_2)$	$\boldsymbol{O(1)}$	$O(n_1n_2)$	$O(n_1+n_2)$	$O(n_1+n_2)$
intersection	$\Theta(N)$	$O(h_1h_2)$	$O(h_1h_2)$	$O(n_1n_2)$	$O(n_1+n_2)$	$O(n_1+n_2)$
diff	$\Theta(N)$	$O(h_1h_2)$	$O(h_1h_2)$	$O(n_1n_2)$	$O(n_1+n_2)$	$O(n_1+n_2)$
symdiff	$\Theta(N)$	$O(h_1h_2)$	$O(h_1h_2)$	$O(n_1n_2)$	$O(n_1+n_2)$	$O(n_1+n_2)$
compl	$\Theta(N)$	–	–	–	–	–

BV: Bitvektor; A: Array; o: ohne; m: mit; W: Wiederholung; A/L-sort: sortierte(s) Array/Liste

N: Länge des Bitvektors; n, n_1, n_2: Größe von S, S_1, S_2

h, h_1, h_2: Länge der Listen-/Arraydarstellung von S, S_1, S_2

PAUSE

2.4. Der Datentyp Wörterbuch

Synonyme:

„Dictionary“, **„(Dynamische) Abbildung“**, **„Map“**, **„Assoziatives Array“**.

Abstrakte Klasse in Java: `java.util.Dictionary`. Klasse in C++: `std::map`.

Gegeben ist Schlüsselmenge U und Wertemenge R („*range*“).

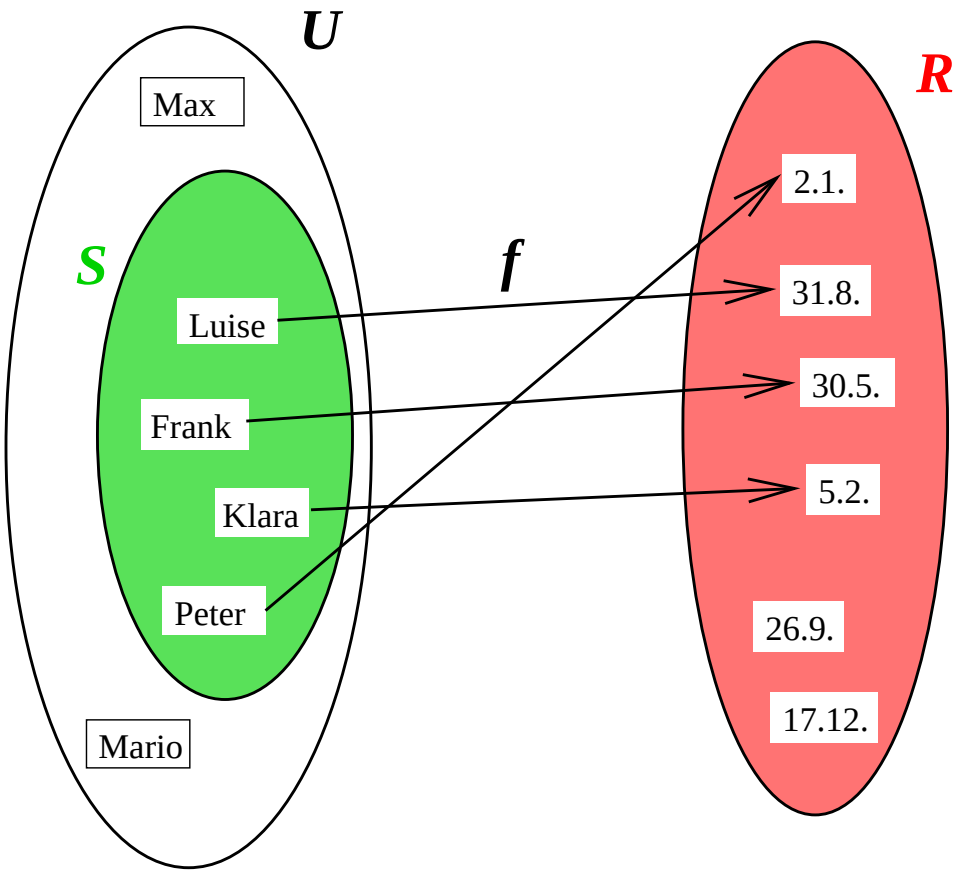
Manchen Schlüsseln $x \in U$ soll ein Wert $f(x) \in R$ zugeordnet sein.

Menge der möglichen Datensätze: $D = U \times R$ (Menge von Paaren (x, r)).

Man möchte:

empty$()$:	ein leeres Wörterbuch erzeugen
lookup(x):	den dem Schlüssel x zugeordneten Wert $f(x)$ finden
insert(x, r):	ein neues (Schlüssel, Wert)-Paar (x, r) einbauen
delete(x):	den Eintrag zu Schlüssel x löschen

Beispiel für eine solche Zuordnung mit $U =$ Menge der endlichen Folgen von Buchstaben A, B, ..., Z, a, b, ..., z und $R = \{1, \ldots, 31\} \times \{1, \ldots, 12\}$:



i	Op_i	Wörterbuch	Ausgabe
0	*empty*()	$\emptyset$	
1	*insert*(Frank, 30.5.)	{(Frank, 30.5.)}	„ok“
2	*insert*(Klara, 17.12.)	{(Frank, 30.5.), (Klara, 17.12.)}	„ok“
3	*insert*(Peter, 2.1.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.)}	„ok“
4	*lookup*(Klara)	” ”	„17.12.“
5	*lookup*(Luise)	” ”	„$\perp$“
6	*insert*(Luise, 31.8.)	{(Frank, 30.5.), (Klara, 17.12.), (Peter, 2.1.), (Luise, 31.8.)}	„ok“
7	*insert*(Klara, 5.2.)	{(Frank, 30.5.), (Klara, 5.2.), (Peter, 2.1.), (Luise, 31.8.)}	„!!“
8	*lookup*(Klara)	” ”	„5.2.“
9	*delete*(Peter)	{(Frank, 30.5.), (Klara, 5.2.), (Luise, 31.8.)}	„ok“
10	*lookup*(Peter)	” ”	„$\perp$“
11	*delete*(Mario)	” ”	„!!“

Datentyp Wörterbuch

1. Signatur:

Sorten:	*Keys*
	Values
	Maps
Operationen:	$empty: \rightarrow Maps$
	$insert: Maps \times Keys \times Values \rightarrow Maps$
	$delete: Maps \times Keys \rightarrow Maps$
	$lookup: Maps \times Keys \rightarrow Values$

2. Mathematisches Modell:

Sorten: *Keys*: Menge U (Menge aller Schlüssel, ein Parameter)
Values: Menge R (Menge aller Werte, ein Parameter)
Maps: $\{f \mid f\colon S \to R \text{ für ein } S \subseteq U,\ |S| < \infty\}$

Erinnerung:

$f\colon S \to R$ heißt „f ist **Abbildung/Funktion** von S nach R“, und das heißt:

$f \subseteq S \times R$ und $\forall x \in S\ \exists! r \in R\colon (x, r) \in f$.

(„$\exists!$“: „es gibt genau ein“.)

Für S schreiben wir **Def(f)** (Definitionsbereich)

Falls $x \in \text{Def}(f)$, ist $\boldsymbol{f(x)}$ das eindeutig bestimmte $r \in R$ mit $(x, r) \in f$.

Operationen:

$\mathit{empty}() := \emptyset$.

$$\mathit{lookup}(f, x) := \begin{cases} f(x), & \text{falls } x \in \mathsf{Def}(f) \\ \text{„}\perp\text{“}, & \text{sonst.} \end{cases}$$

$$\mathit{delete}(f, x) := \begin{cases} f - \{(x, f(x))\}, & \text{falls } x \in \mathsf{Def}(f) \\ f, & \text{sonst.} \end{cases}$$

$$\mathit{insert}(f, x, r) := \begin{cases} (f - \{(x, f(x))\}) \cup \{(x, r)\}, & \text{falls } x \in \mathsf{Def}(f) \\ f \cup \{(x, r)\}, & \text{sonst.} \end{cases}$$

Achtung: „$\perp$“ („*undefiniert*“) ist ein reguläres Ergebnis. Natürlich sollte $\perp$ kein Element von R sein.

Implementierungsmöglichkeiten

- Listen/Arrays, Einträge aus $U \times R$, mit Wiederholung von Schlüsseln.
  (Bei Einfügen vorne einhängen, das erste Vorkommen eines Schlüssels zählt.)
- Listen/Arrays, Einträge aus $U \times R$, ohne Wiederholung von Schlüsseln, unsortiert oder sortiert.
- Wenn $U = \{1, \ldots, N\}$: Array `f[1..N]` mit Werten in $R \cup \{\perp\}$, wobei $\perp \mathrel{\hat{=}}$ „*undefiniert*“.
- **Suchbäume** und **Hashtabellen** (später).

Zeiten für Operationen wie bei Implementierungen des Datentyps *DynSet* (Folie 33).

SS 2021

Algorithmen und Datenstrukturen

3. Kapitel

Binärbäume

Martin Dietzfelbinger

Mai 2021

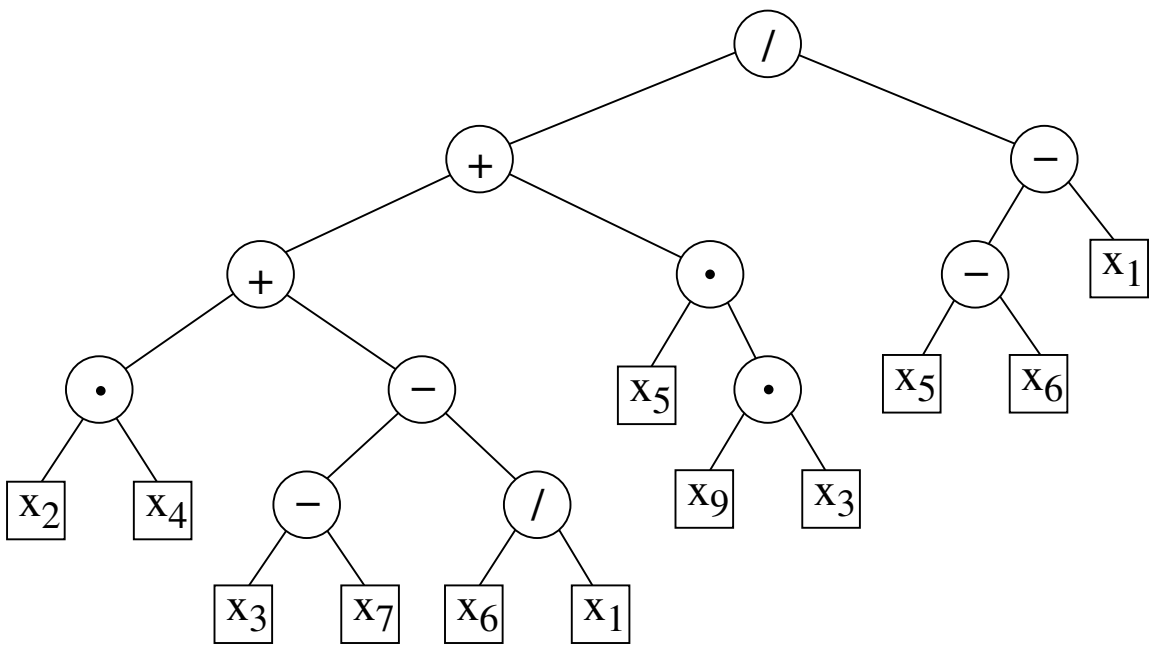
3.1 Binärbäume: Grundlagen

Der Datentyp (und die Datenstruktur) **Binärbaum** tritt überall in der Informatik auf:

- als binärer Suchbaum
- als Repräsentation von binären Ausdrücken (logisch, arithmetisch)
- als Entscheidungsbaum
- als Codierungs-/Decodierungsbaum
- als binärer **Heap** (später)
- ...

Beispiel 1: **Arithmetischer Ausdruck** mit zweistelligen Operatoren:

$((((x_2 \cdot x_4) + ((x_3 - x_7) - (x_6/x_1))) + (x_5 \cdot (x_9 \cdot x_3)))/((x_5 - x_6) - x_1))$

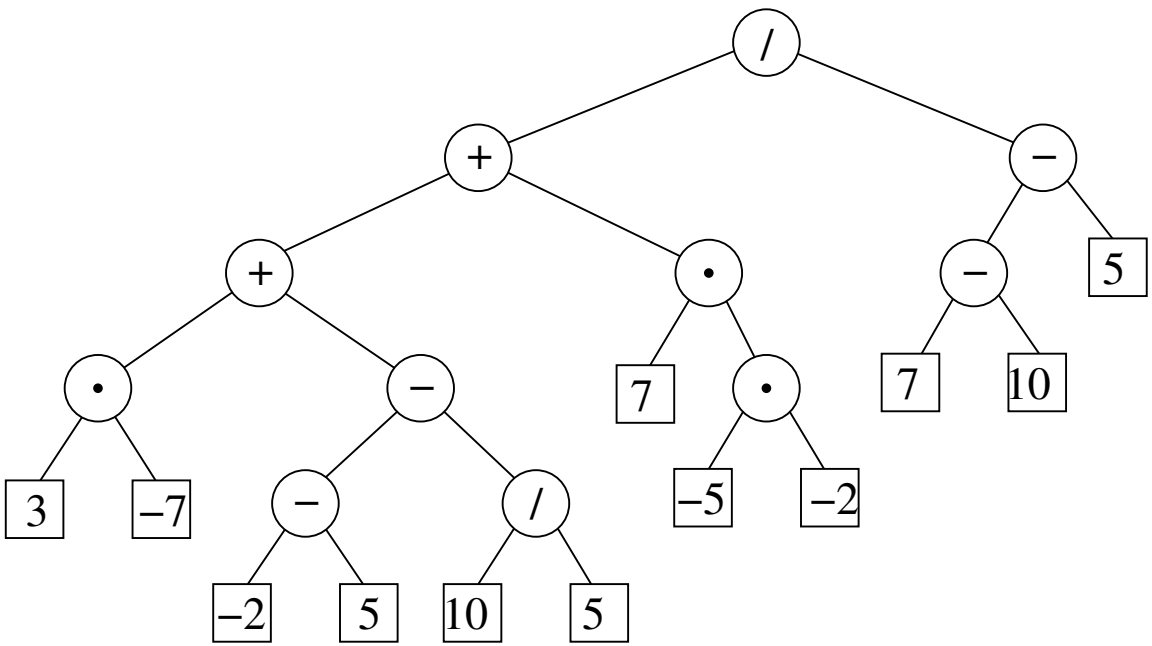


Auswertung:

1) Ordne den Blättern konkrete Werte zu,

z.B. $x_1 = 5$, $x_2 = 3$, $x_3 = -2$, $x_4 = -7$, $x_5 = 7$, $x_6 = 10$, $x_7 = 5$, $x_9 = -5$.

2) Werte aus, von den Blättern startend („bottom-up“).

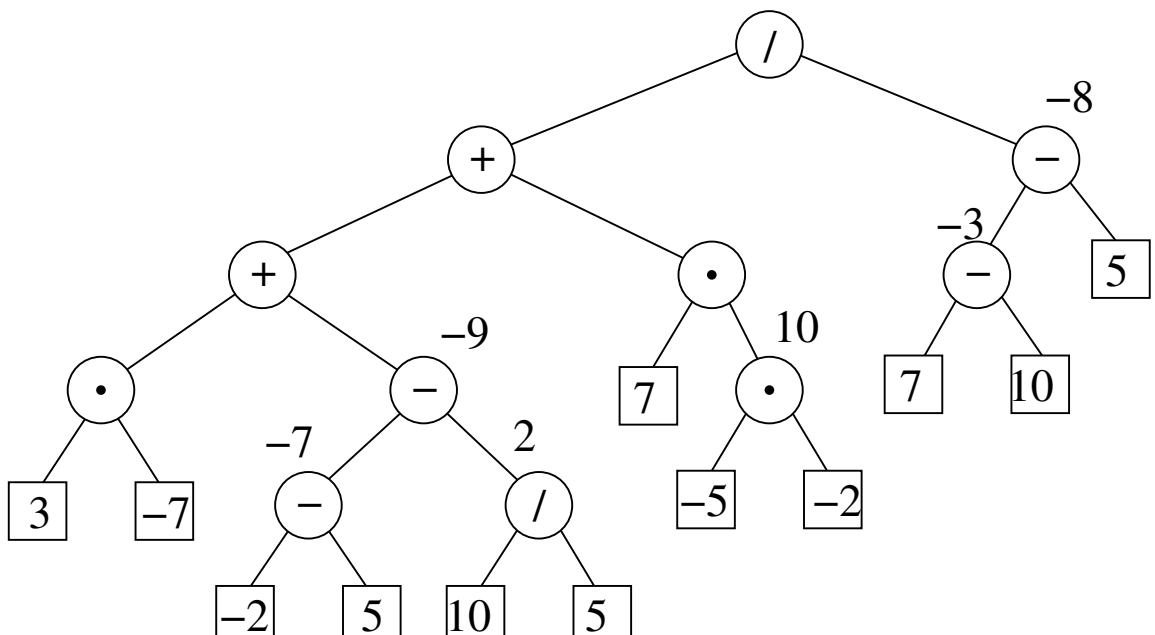


Auswertung:

1) Ordne den Blättern konkrete Werte zu,

z.B. $x_1 = 5$, $x_2 = 3$, $x_3 = -2$, $x_4 = -7$, $x_5 = 7$, $x_6 = 10$, $x_7 = 5$, $x_9 = -5$.

2) Werte aus, von den Blättern startend („bottom-up“).



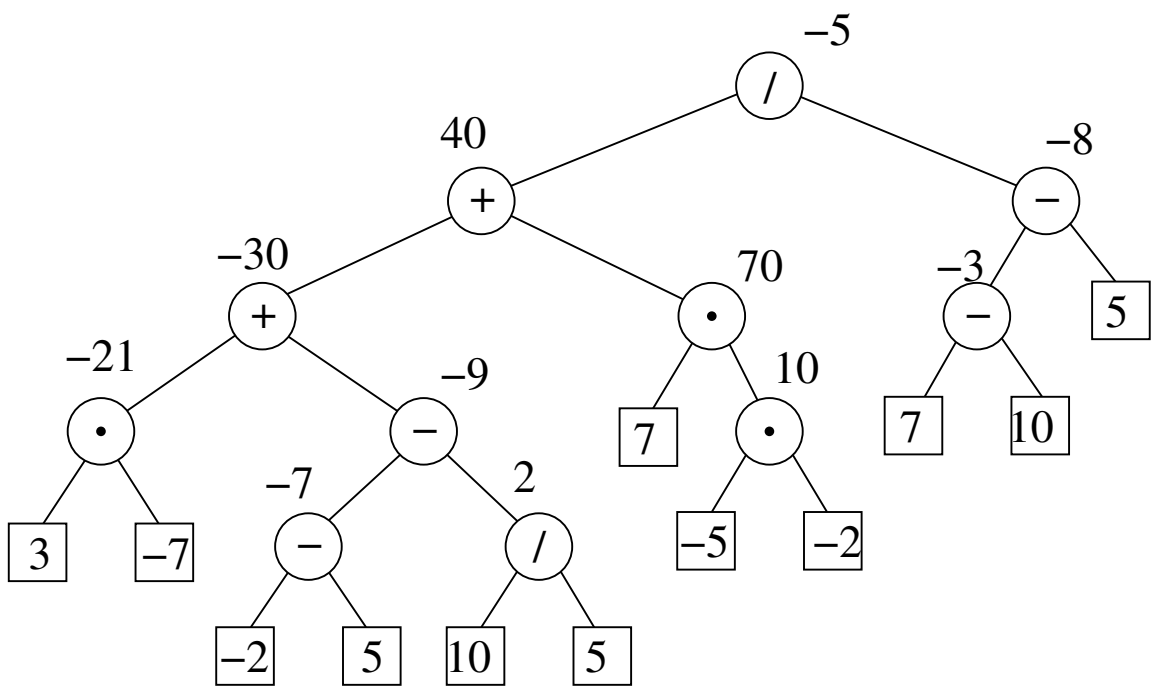
Nach sechs Schritten.

Auswertung:

1) Ordne den Blättern konkrete Werte zu,

z.B. $x_1 = 5$, $x_2 = 3$, $x_3 = -2$, $x_4 = -7$, $x_5 = 7$, $x_6 = 10$, $x_7 = 5$, $x_9 = -5$.

2) Werte aus, von den Blättern startend („bottom-up“).



Fertig. Resultat: -5, an der Wurzel abzulesen.

Beispiel 2: **Codierungs-/Decodierungsbaum**

Zeichen aus Alphabet Σ erhalten binäre Codes.

Diese können auch *verschiedene Länge* haben:

„Wenige Bits für häufige Buchstaben, viele Bits für seltene.“

Mini-Beispiel:

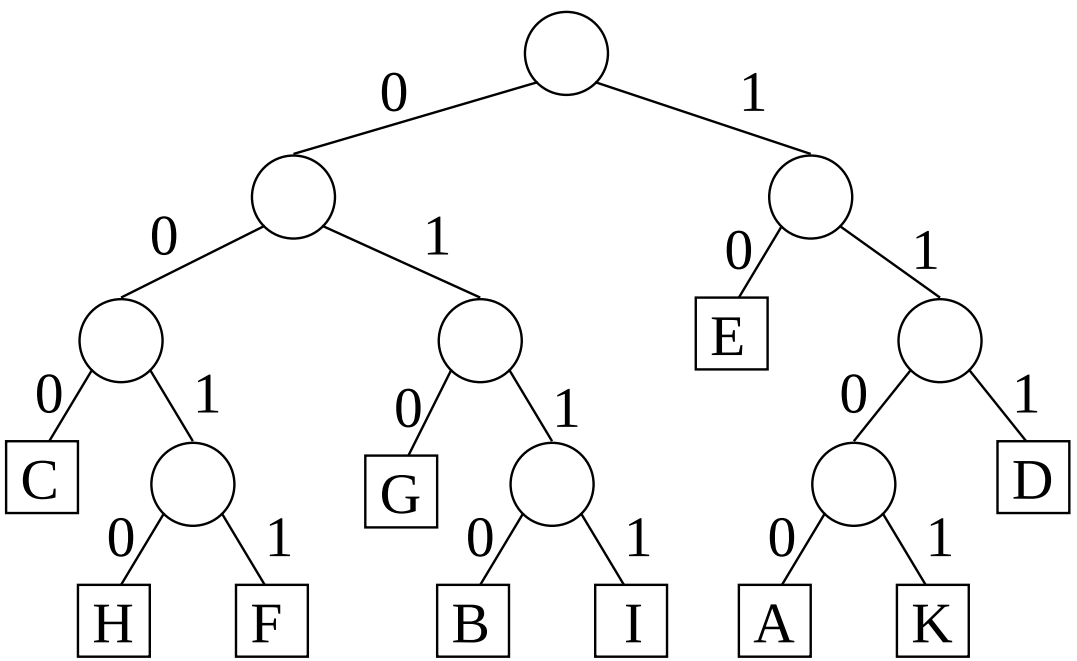
A	B	C	D	E	F	G	H	I	K
1100	0110	000	111	10	0011	010	0010	0111	1101

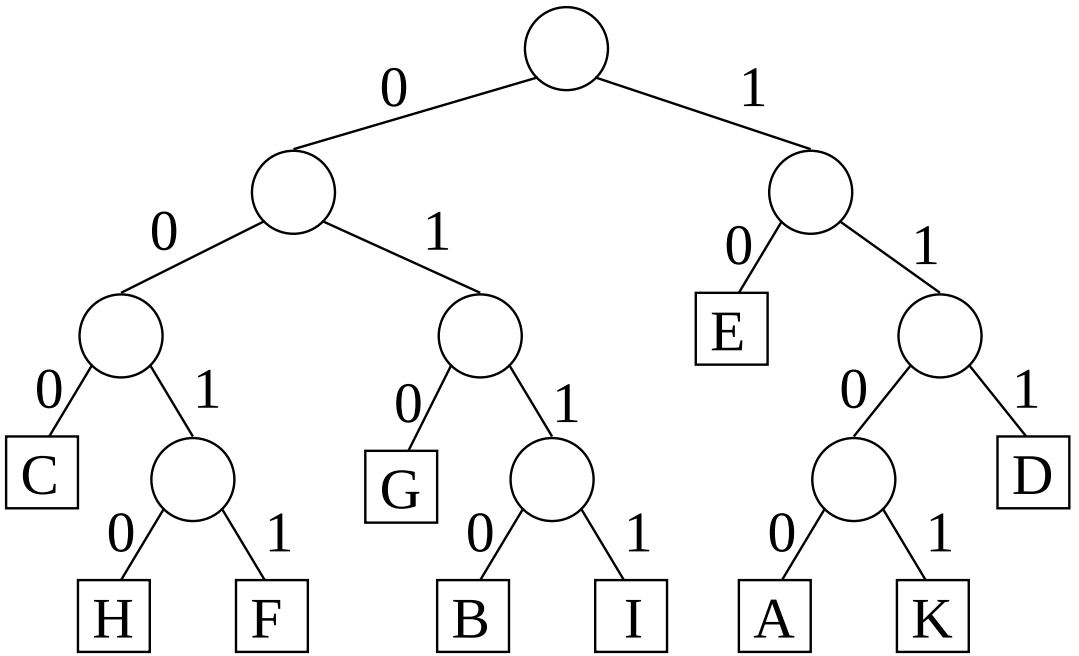
Zur Codierung von Zeichenreihen benutzt man direkt die Tabelle.

Beispiel:
`FEIGE` hat Codierung $0011\,10\,0111\,010\,10$.

„Präfixfreier“ Code: Kein Codewort ist Anfangsstück eines anderen.
Damit: „eindeutig decodierbar“; Fuge zwischen Codes für Buchstaben muss nicht markiert werden: Schreibe $00111001110101 0$ statt $0011\,10\,0111\,010\,10$.

Kompakte Repräsentation des Codes als Binärbaum:



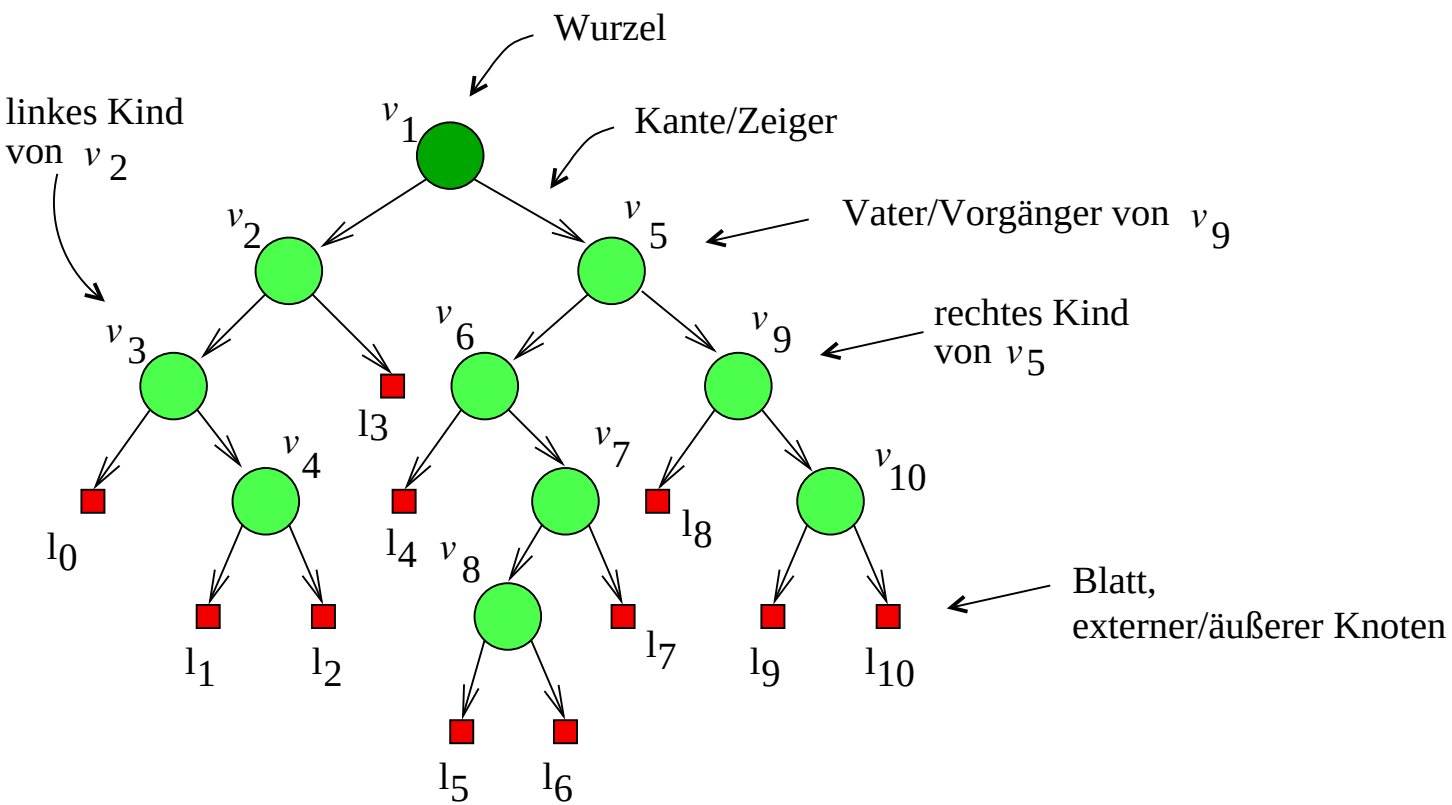


Blätter sind mit Buchstaben markiert; Weg von der Wurzel zum Blatt gibt das Codewort wieder (links: 0, rechts: 1).

Decodierung: Laufe Weg im Baum, vom Codewort gesteuert, bis zum Blatt.
Wegen Präfixeigenschaft: im Code keine Zwischenräume nötig.
Beispiel: 0000010100011 liefert „`CHEF`“.

Teil 2: Definitionen

Binärbaum-Terminologie: Überblick



Definition von Binärbäumen: „flach“

Definition 3.1.1

Ein **Binärbaum** T besteht aus einer endlichen Menge V von „inneren“ Knoten, einer dazu disjunkten endlichen Menge L von „äußeren“ Knoten sowie einer **injektiven** Funktion

$$\mathit{child}\colon V \times \{\mathit{left}, \mathit{right}\} \to V \cup L,$$

wobei Folgendes gilt:

(i) In $V \cup L$ gibt es genau einen Knoten r, der nicht als Wert $\mathit{child}(v, \mathit{left})$ oder $\mathit{child}(v, \mathit{right})$ vorkommt. Dieser Knoten r heißt die **Wurzel**.

Definition von Binärbäumen: „flach“

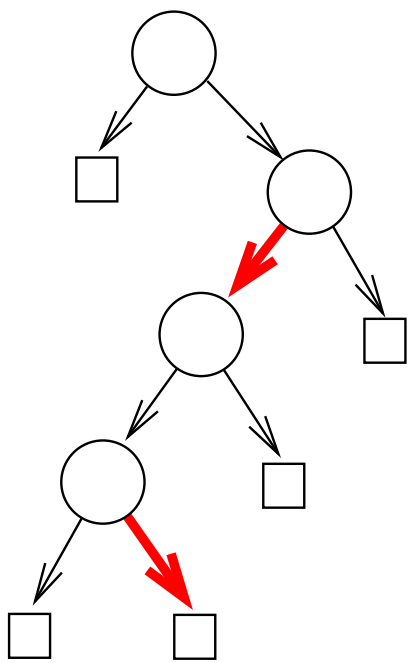
child(v, left) [*child*(v, right)] heißt
das **linke** [**rechte**] **Kind** von $v \in V$.

“*child* injektiv” heißt:
Für jeden Knoten $w \in V \cup L$ mit $w \neq r$ gibt es **genau einen** Knoten $u \in V$ mit
$w =$ *child*(u, left) oder $w =$ *child*(u, right)
(aber nicht beides).

Dieses u heißt $p(w)$, der **Vorgänger**knoten oder **Vater**knoten von w.

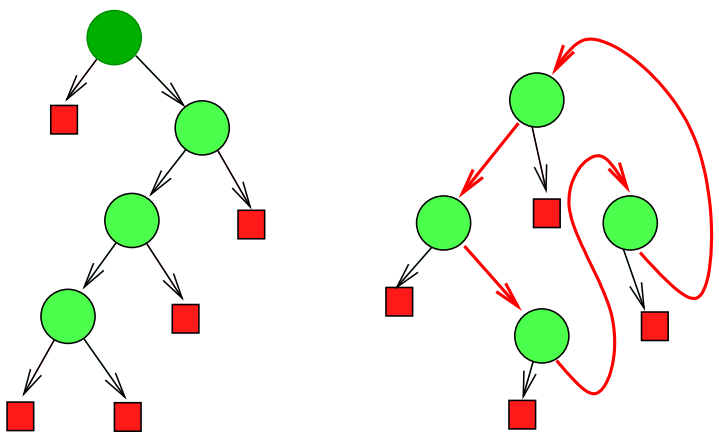
Wir sagen:

Von $p(w)$ nach w verläuft eine **Kante**.



Definition von Binärbäumen: „flach“

Beispiel: Die folgende (zweiteilige) Struktur erfüllt (i), ist aber kein Binärbaum:



(ii) **„Kreisfreiheit“:** Für jeden Knoten $w \in V \cup L$ gilt:
Die Folge $v_0 = w, v_1 = p(w), v_2 = p(v_1), v_3 = p(v_2), \ldots$ bricht nach endlich vielen Schritten mit einem $v_d = r$ ab.

(Dadurch ist ein **eindeutiger Weg** von w zur Wurzel festgelegt. Also ist jeder Knoten des Binärbaums von der Wurzel aus entlang einer Kantenfolge („Weg“) erreichbar.)

Definition von Binärbäumen: „flach“

Markierungen (oder **Beschriftungen**) von Knoten und Kanten werden als Funktionen dargestellt: (D, Z, X sind beliebige Mengen.)

Knotenmarkierungen: Funktionen $m_V \colon V \to D$ und $m_L \colon L \to Z$.

Kantenmarkierungen: Funktion $m_K \colon V \times \{\mathit{left}, \mathit{right}\} \to X$.

Beispiel: In einem Codierungsbaum sind die Blätter mit Buchstaben markiert, die beiden Kanten, die aus einem inneren Knoten herausführen, mit 0 und 1.

Definition von Binärbäumen: rekursiv

D sei Menge von (möglichen) Knoten**markierungen**

Z sei Menge von (möglichen) Blatt**markierungen**

(Induktive) Definition 3.1.2

(i) Wenn $z \in Z$, dann ist (z) ein (D, Z)-Binärbaum.

// Blatt mit Markierung z. Auch: z (ohne Klammern), $\boxed{z}$

(ii) Wenn T_1, T_2 (D, Z)-Binärbäume sind und $x \in D$ ist,
dann ist auch (T_1, x, T_2) ein (D, Z)-Binärbaum.

// Wurzel mit Markierung x, linker Unterbaum T_1, rechter Unterbaum T_2.

(iii)* Nichts sonst ist (D, Z)-Binärbaum.

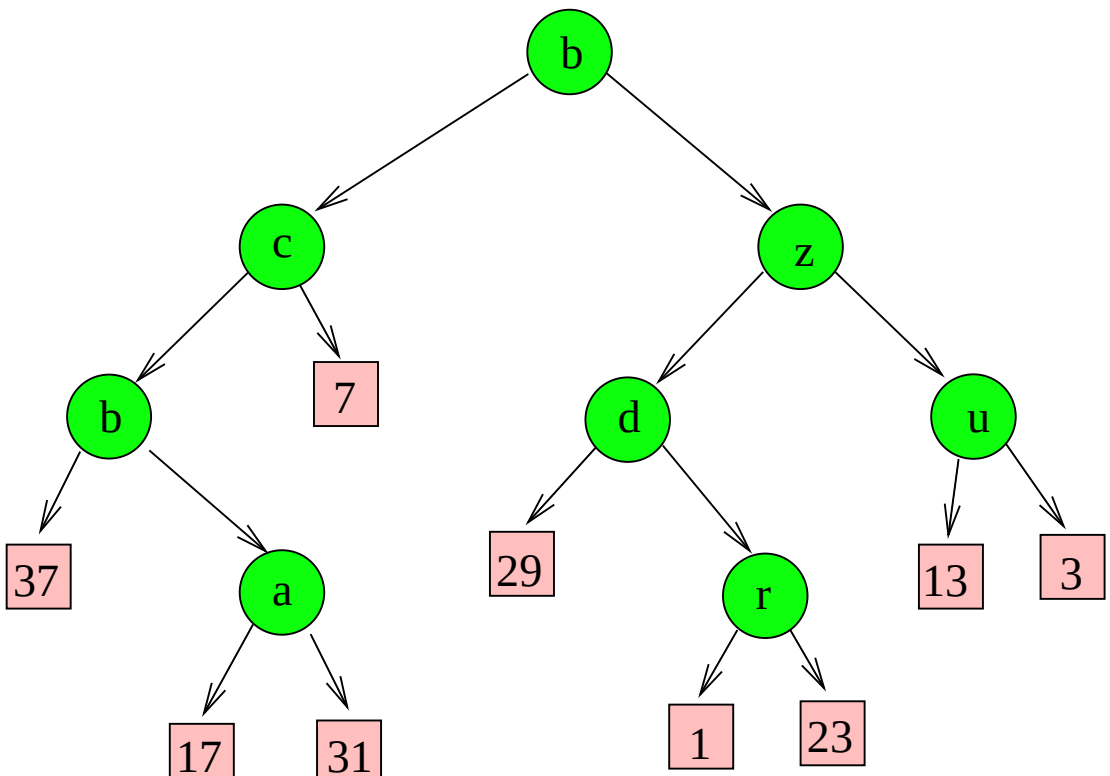
Definition ist äquivalent zur "flachen" Auffassung.

*(iii) wird bei induktiven Definitionen meist weggelassen.

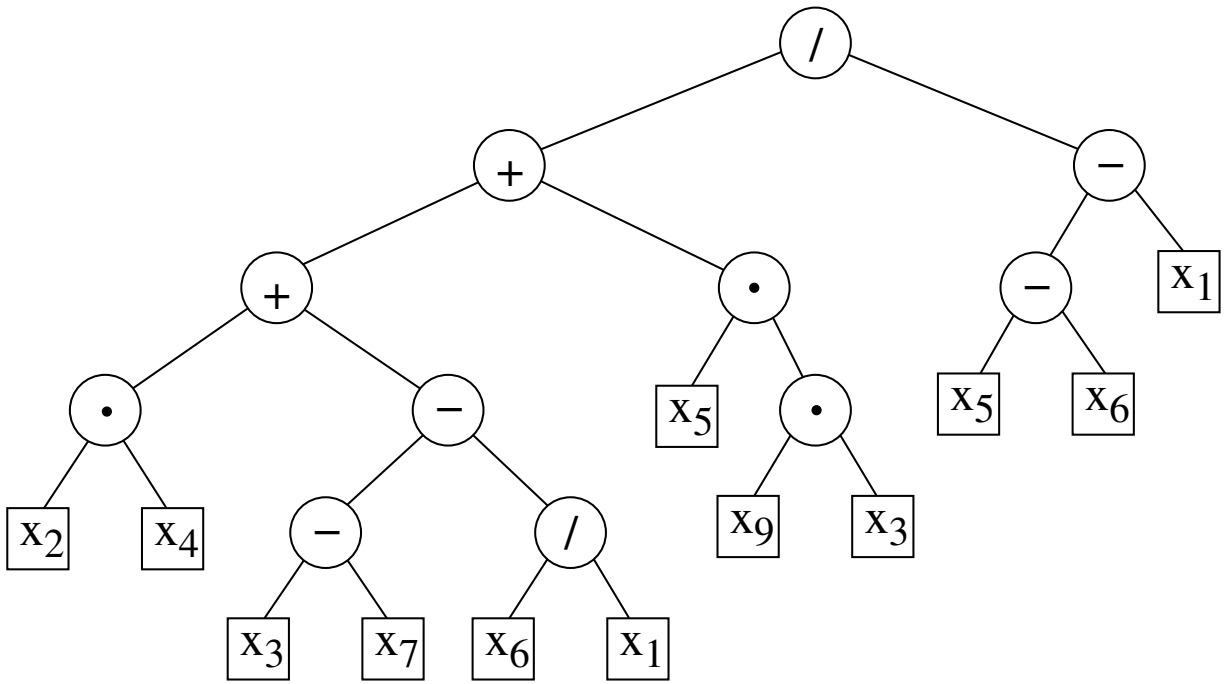
Beispiel: $D = \{\mathrm{a}, \ldots, \mathrm{z}\}$ und $Z = \mathbb{N}$. – Tupeldarstellung (rekursiv):

$(((37, \mathrm{b}, (17, \mathrm{a}, 31)), \mathrm{c}, 7), \mathrm{b}, ((29, \mathrm{d}, (1, \mathrm{r}, 23)), \mathrm{z}, (13, \mathrm{u}, 3)))$

Graphische Darstellung („flach“):



Beispiel: **Arithmetischer Ausdruck.** $D = \{+, -, \cdot, /\}$ und $Z = \{x_1, x_2, x_3, \ldots\}$.



$$((((x_2 \cdot x_4) + ((x_3 - x_7) - (x_6/x_1))) + (x_5 \cdot (x_9 \cdot x_3)))/((x_5 - x_6) - x_1))$$

Der Ausdruck ist genau die (rekursive) Tupeldarstellung, nur ohne Kommas!

Teil 3: Operationen

Spezifikation des Datentyps „Binärbaum“

Datentyp Binärbaum über D: (**leere** äußere Knoten)

1. Signatur:

Sorten: *Elements*
Bintrees
Boolean

Operationen: *empty*: $\rightarrow$ *Bintrees*
buildTree: *Elements* $\times$ *Bintrees* $\times$ *Bintrees* $\rightarrow$ *Bintrees*
leftSubtree: *Bintrees* $\rightarrow$ *Bintrees*
rightSubtree: *Bintrees* $\rightarrow$ *Bintrees*
data: *Bintrees* $\rightarrow$ *Elements*
isempty: *Bintrees* $\rightarrow$ *Boolean*

2. Mathematisches Modell: (nächste Folie)

Sorten: *Elements*: (nichtleere) Menge D

Bintrees: $\{T \mid T$ ist $(D, \{-\})$-Binärbaum$\}$

Boolean: $\{true, false\}$

Op'en: *empty*$() := \square := (-)$ // „leerer Baum“

buildTree$(x, T_1, T_2) := (T_1, x, T_2)$

$$\textit{leftSubtree}(T) := \begin{cases} T_1, & \text{falls } T = (T_1, x, T_2) \\ \text{undefiniert}, & \text{falls } T = \square \end{cases}$$

$$\textit{rightSubtree}(T) := \begin{cases} T_2, & \text{falls } T = (T_1, x, T_2) \\ \text{undefiniert}, & \text{falls } T = \square \end{cases}$$

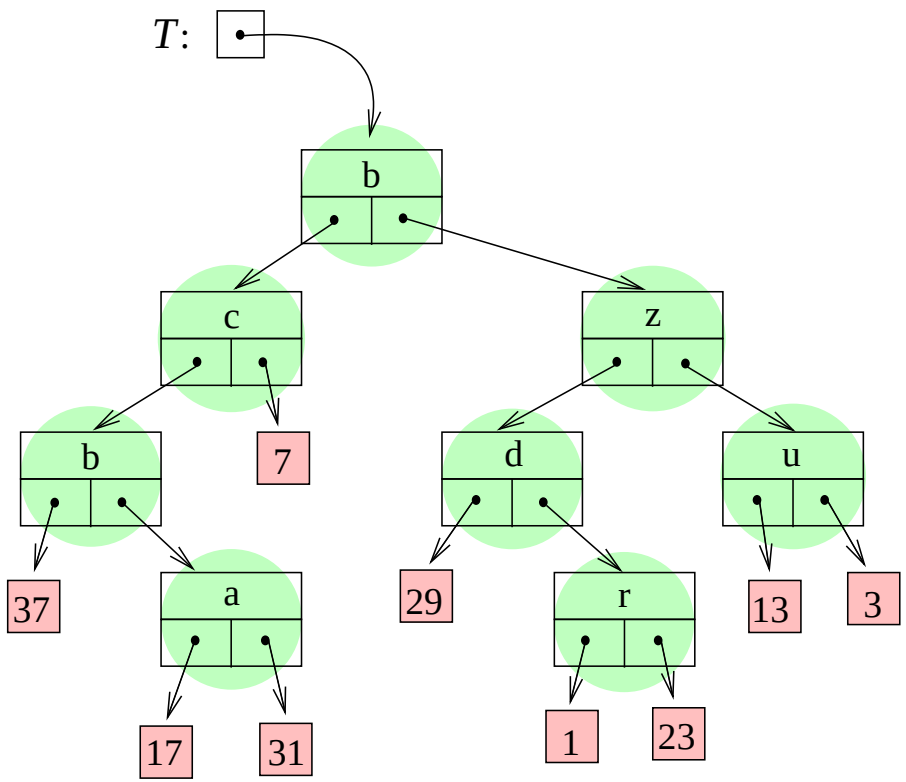
$$\textit{data}(T) := \begin{cases} x, & \text{falls } T = (T_1, x, T_2) \\ \text{undefiniert}, & \text{falls } T = \square \end{cases}$$

$$\textit{isempty}(T) := \begin{cases} false, & \text{falls } T = (T_1, x, T_2) \\ true, & \text{falls } T = \square \end{cases}$$

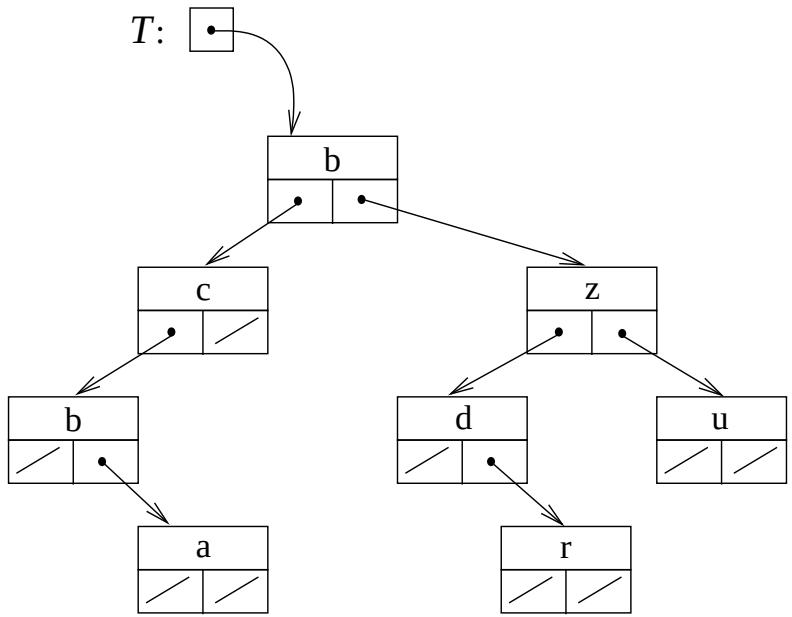
Binärbäume, Implementierung I:

Zeigerstruktur bzw. Referenzstruktur, hier mit externen Knoten, die Inhalt haben.

Für Implementierung in Java vgl. [AuP], Kap. 10 und [Saake/Sattler], Kap. 14.2.



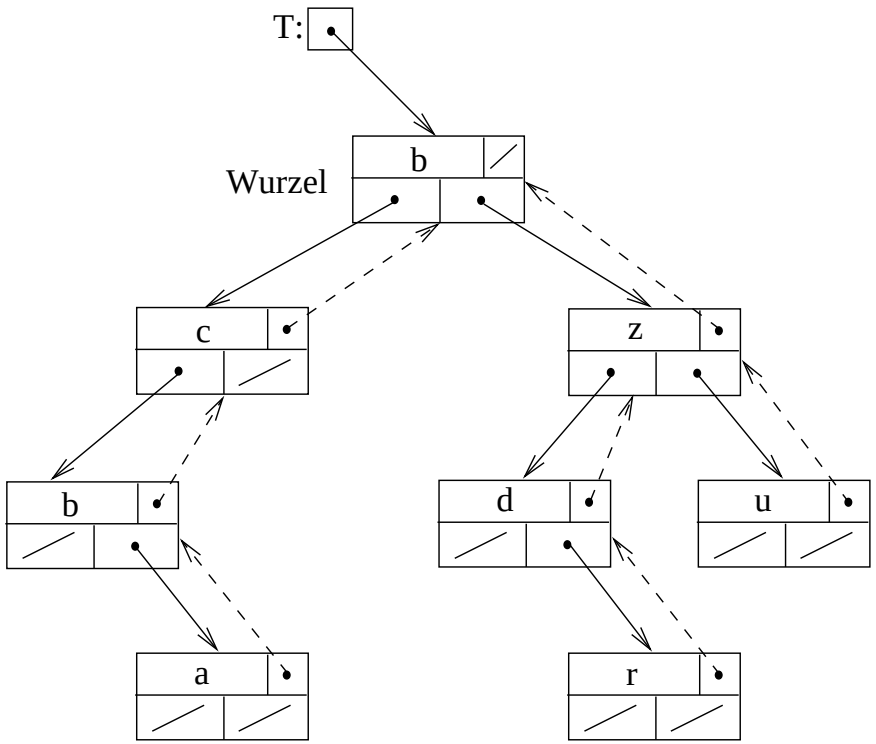
Binärbäume, Implementierung II: Leere externe Knoten



Externe Knoten werden nicht repräsentiert (`null`-Zeiger/-Referenz). – Oder: Ein externer Knoten ℓ, Referenz auf ℓ statt `null` (`nullNode` in [Saake/Sattler], Kap. 14.2.)

Speicherplatzbedarf für n Knoten: n Dateneinträge, $2n$ Zeiger/Referenzen, davon $n + 1$ mal `null`.

Mögliche Erweiterung, konzeptuell nur in „flacher“ Auffassung: Vorgängerzeiger.



Erleichtert Navigieren im Baum, besonders bei Verwenden von iterativen Verfahren.

Implementierung der Operationen des mathematischen Modells (leere externe Knoten):

Baum wird immer als Zeiger/Referenz auf den Wurzelknoten dargestellt.

empty(): Generiere einen Zeiger/eine Referenz `T` mit Wert `null`.
(Bzw.: Generiere `nullNode` und generiere Zeiger/Referenz `T` auf diesen Knoten.)

isempty(`T`): klar.

data(`T`): auch klar; nur definiert, wenn `T` nicht `null` ist.

leftSubtree(`T`) liefert Zeiger/Referenz `T.left`; nur definiert, wenn `T` nicht `null` ist.

rightSubtree(`T`): analog.

buildTree(x, T_1, T_2) erzeugt einen neuen Knoten (Wurzel) r mit Eintrag x, in die Kindplätze werden die Zeiger/Referenzen auf T_1 und T_2 eingetragen.

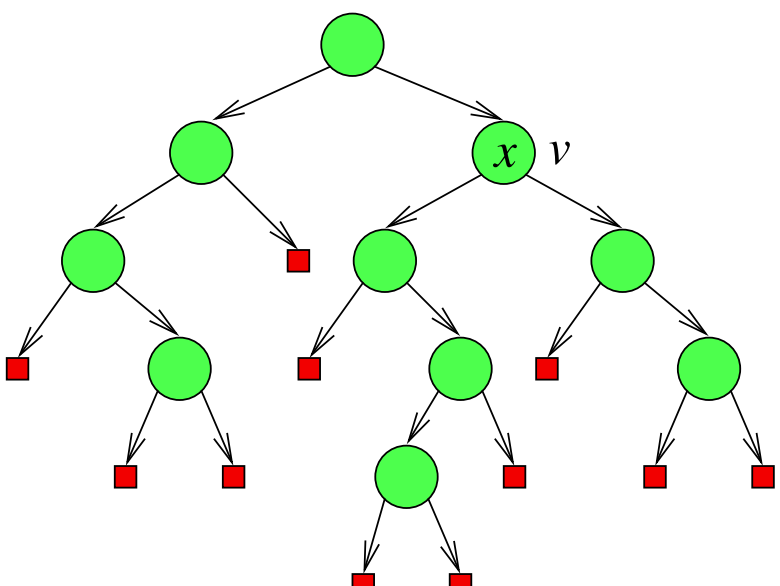
Achtung: Der **Benutzer** der *buildTree*-Operation muss sicherstellen, dass die Knotenmengen von T_1 und T_2 disjunkt sind!

Details: [Saake/Sattler], Kap. 14.2., wenn auch mit leicht anderen Bezeichnungen.

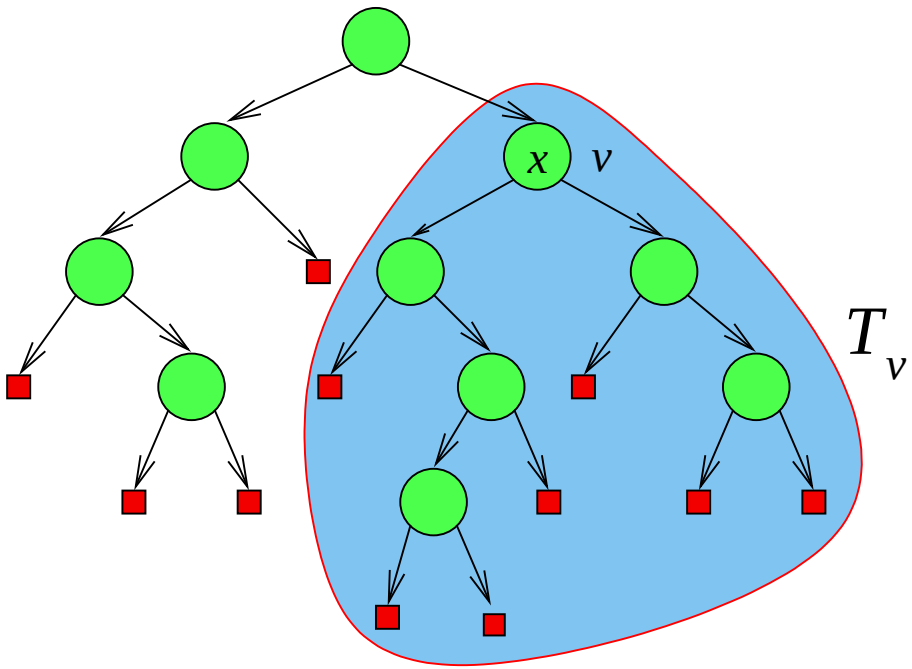
Teil 4: Terminologie

3.2 Binärbäume: Terminologie

- **Binärbaum** T, engl: „*binary tree*“
- (innerer) **Knoten** v, engl: „*node*“
- Knoteneintrag: $x = \mathit{data}(v) = \mathit{data}(T') \qquad (T' = T_v)$

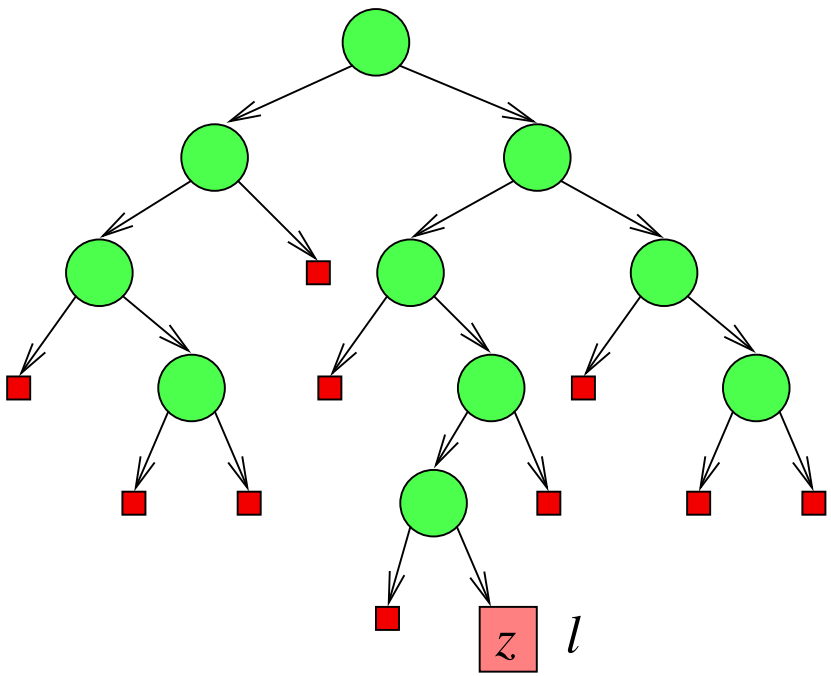


- T_v: Unterbaum mit Wurzel v



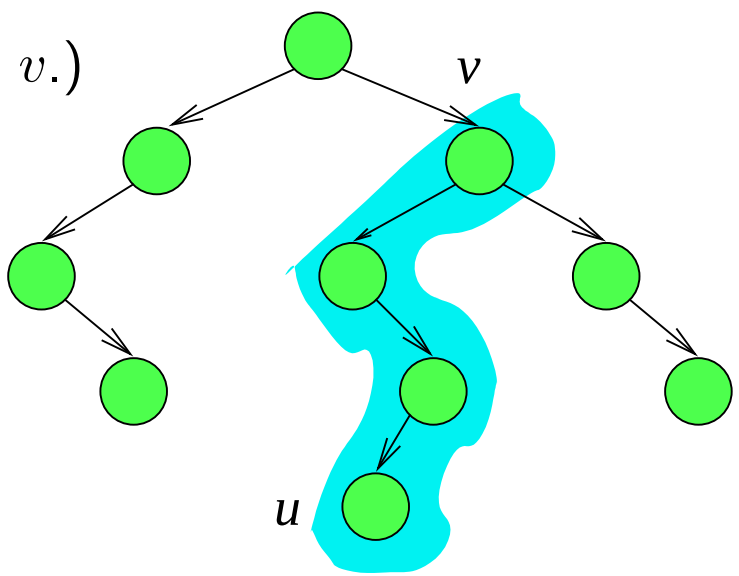
T_v ist der Teil von T, der aus den Knoten besteht, deren Weg zur Wurzel durch v verläuft. Die Wurzel von T_v ist v.
Dabei kann v ein beliebiger innerer oder äußerer Knoten sein.

- **(äußerer) Knoten** $\equiv$ **Blatt** l, engl: „*leaf*“, Plural „*leaves*“
- Wurzel, engl: „*root*“



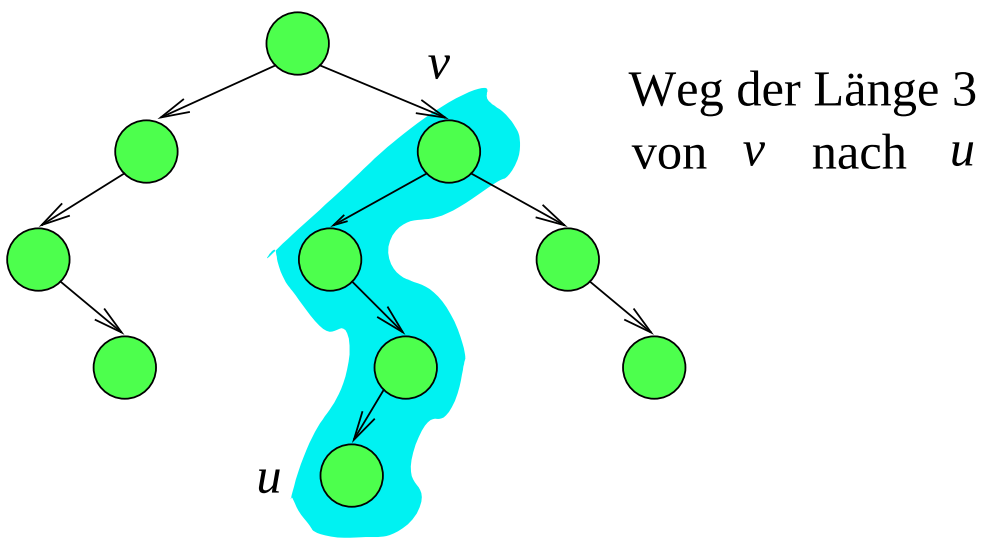
- Blatteintrag: $z = \mathit{data}(l)$

- **Vater/Vorgänger**, engl: „*parent*“, von v: $\boldsymbol{p(v)}$
- linkes **Kind**, engl: „*left child*“, von v: von v: $child(v, left)$
- rechtes **Kind**, engl: „*right child*“, von v: von v: $child(v, right)$
- **Vorfahr** (Vater von Vater von Vater . . . von u),
  engl: „*ancestor*“ (Im Bild: v ist Vorfahr von u.)
- **Nachfahr** (Kind von Kind von . . . von v),
  engl: „*descendant*“ (Im Bild: u ist Nachfahr von v.)



Achtung: Oft gilt v als (uneigentlicher) Vorfahr/Nachfahr von v.

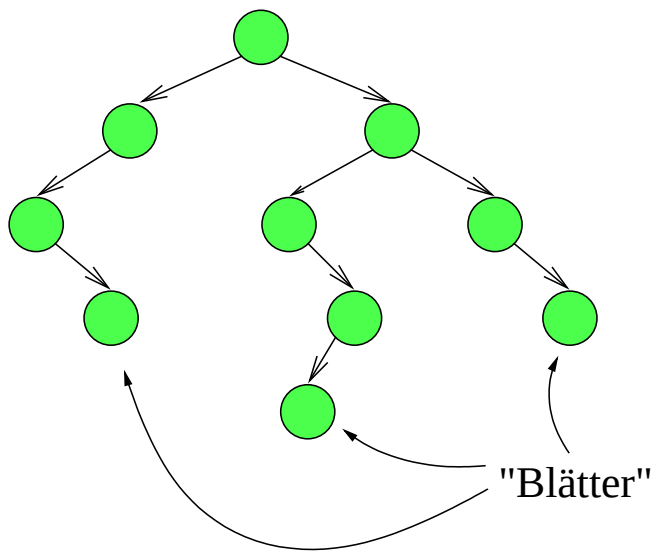
- **Weg** in T: Knotenfolge oder Kantenfolge, die von v zu einem Nachfahren u führt
- **Länge** eines Wegs $(v_0, v_1, \ldots, v_s)$: Anzahl s der **Kanten** auf dem Weg



Häufiger Spezialfall: Die externen Knoten □ haben keinen Eintrag.

(Bzw. jeder hat denselben nichtssagenden Eintrag, z.B. „−“ .)

Ein solcher externer Knoten wird auch als „**leerer Baum**“ bezeichnet.

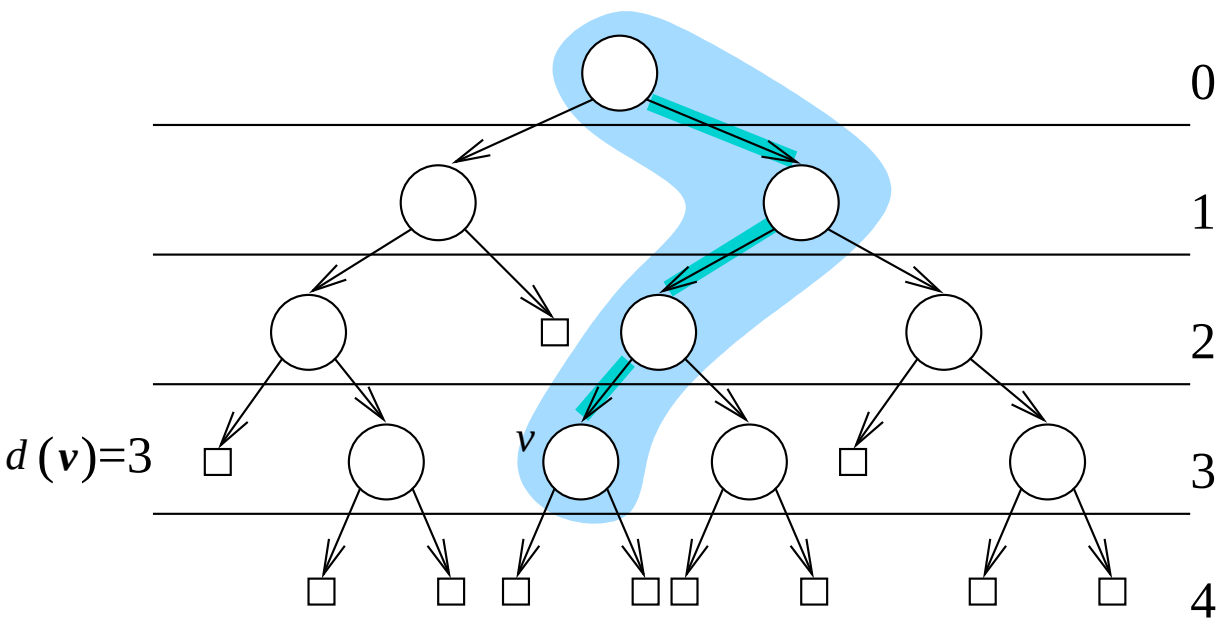


Oft bezeichnet man dann die inneren Knoten, die keinen inneren Knoten als Nachfolger haben, als „Blätter“.

Tiefe/Level eines Knotens

Die Länge (das ist die Kantenzahl) des Wegs $(v_0, v_1, \ldots, v_d)$ von der Wurzel $r = v_0$ zum Knoten $v = v_d$ heißt die **Tiefe** oder das **Level** $d(v)$ von Knoten v.
Die Knoten mit Tiefe l bilden das **Level** l.

Auch externen Knoten wird so eine Tiefe zugeordnet.



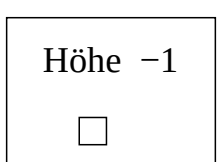
Tiefe (oder synonym: Höhe) $d(T)$ (oder: $h(T)$) eines Binärbaums:

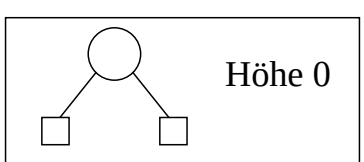
1. Fall: Die externen Knoten sind leer ($\square$):

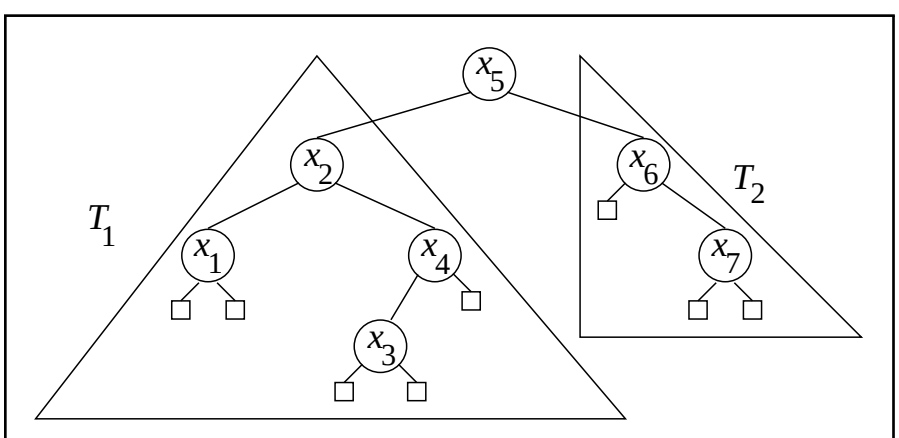
$d(T) := \max(\{-1\} \cup \{d(v) \mid v$ **innerer** Knoten in $T\})$.

Rekursive Formel: $d(T) = -1$ für $T = \square$ und $d((T_1, x, T_2)) = 1 + \max\{d(T_1), d(T_2)\}$.

Beispiele: Höhe/Tiefe -1, 0 und 3.





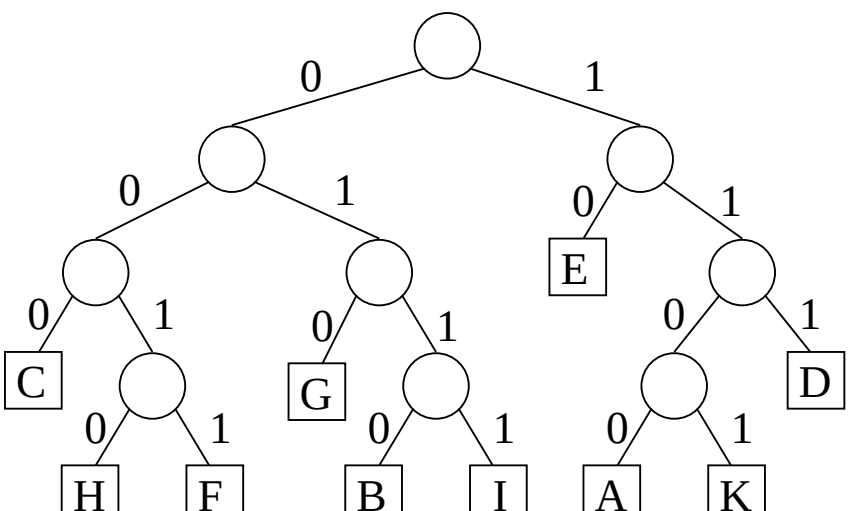


Tiefe (oder synonym: Höhe) $d(T)$ (oder: $h(T)$) eines Binärbaums:

2. Fall: Die externen Knoten sind nicht leer, also von der Form $\boxed{z}$, $z \in Z$:

$d(T) := \max\{d(v) \mid v$ **äußerer** Knoten in $T\}$.

Rekursive Formel: $d(\square) = 0$; $d((T_1, x, T_2) = 1 + \max\{d(T_1), d(T_2)\}$.

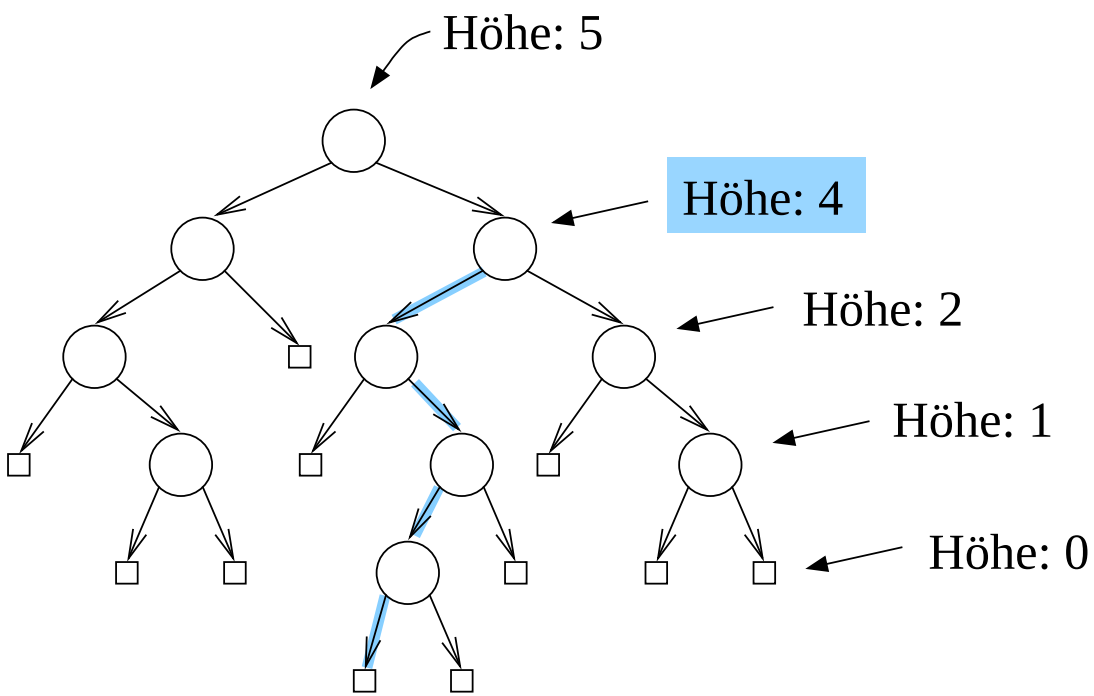


Ein Baum mit Tiefe 4, gleich der maximalen Länge eines Weges zu einem Blatt.

Höhe eines Knotens:

Die **Höhe** $h(v)$ eines Knotens in T ist die **Tiefe** $d(T_v)$ des Teilbaums mit Wurzel v.

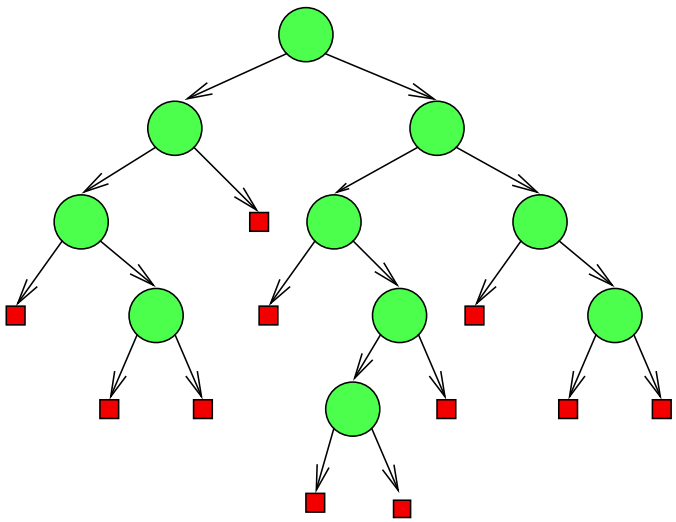
Beispiel:



Hier angegeben: Fall „Blätter nicht leer“.

Teil 5: TIPL und TEPL

3.3 Binärbäume: Eigenschaften



Proposition 3.3.1

Für einen Binärbaum T mit $n \geq 0$ inneren Knoten und **leeren** äußeren Knoten gilt:

(a) T hat $2n$ Zeiger/Kanten. (Im Beispiel: $n = 10$, $2n = 20$.)

(b) T hat $n + 1$ äußere Knoten. (Im Beispiel: $n + 1 = 11$.)

(c) Auf Level l liegen höchstens 2^l Knoten, für $l = 0, 1, 2, \ldots$.

(d) Auf Level $0, 1, \ldots, l$ zusammen liegen höchstens

$$1 + 2 + 4 + \cdots + 2^l = 2^{l+1} - 1$$

innere Knoten.
Daraus: $\log(n+1) \leq l + 1$, für das Level l mit dem tiefsten inneren Knoten.

(e)* $\lceil \log(n+1) \rceil - 1 \leq d(T) \leq n - 1$.

Merke: Die Tiefe eines Binärbaums mit n inneren Knoten ist

mindestens $\log n - 1$ und höchstens $n - 1$.

* $\log n = \log_2 n$.

Beweis von Proposition 3.3.1:

(a) Jeder innere Knoten hat zwei Zeiger/Kanten zu Kindern.

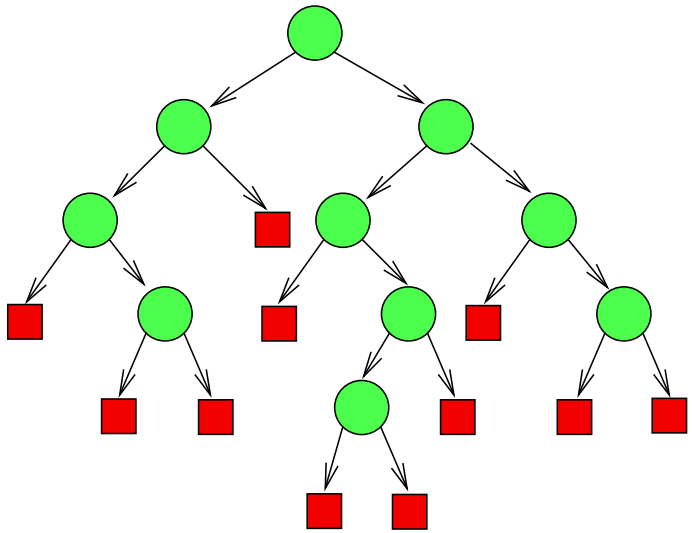
(b) Jeder Knoten außer der Wurzel hat eine eingehende Kante. Nach (a) gibt es $2n$ Kanten, und es gibt $n-1$ innere Knoten, die nicht die Wurzel sind. Daher muss es $2n-(n-1)=n+1$ äußere Knoten geben.

(c) Beweis durch Induktion über l.
$l=0$: Nur die Wurzel hat Level 0, und $2^0=1$.
„$l-1 \to l$“: Nach I.V. liegen auf Level $l-1$ höchstens 2^{l-1} Knoten. Jeder davon hat höchstens zwei Kinder. Also liegen auf Level l höchstens $2^{l-1}\cdot 2 = 2^l$ Knoten.

(d) Sei l das maximale Level, das einen inneren Knoten enthält. Alle Levels bis l zusammen enthalten n Knoten; nach (c) ist $n \leq 2^0+2^1+\cdots+2^l$. Die Schranke $n \leq 2^{l+1}-1$ folgt dann mit Hilfe der Summenformel $\sum_{0\leq j<k} q^j = (q^k-1)/(q-1)$ für geometrische Reihen.

Daraus folgt $n+1 \leq 2^{l+1}$, also durch Logarithmieren: $\log(n+1) \leq l+1$.

(e) In der Situation von (d) gilt $l \geq \log(n+1)-1$; weil l ganzzahlig ist, folgt $l \geq \lceil \log(n+1) \rceil - 1$.
□

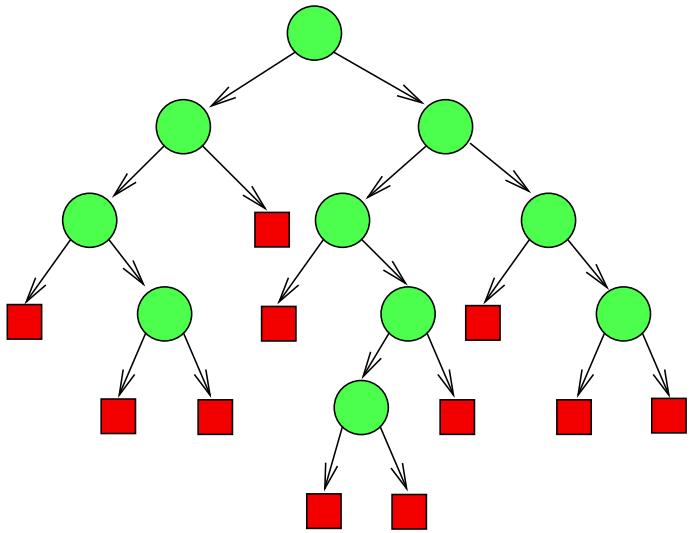


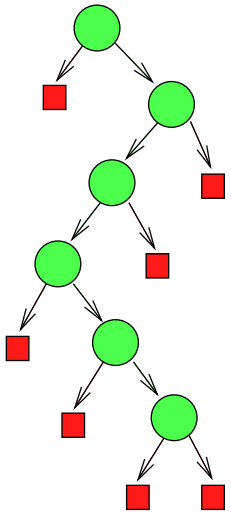
Proposition 3.3.2

Für einen Binärbaum T mit $N \geq 1$ **nichtleeren** äußeren Knoten gilt:

(a) T hat $n = N - 1$ innere Knoten. (Im Beispiel: $N = 11$, $n = 10$.)

(b) T hat $2N - 2$ Zeiger/Kanten.





(c) $N \leq 2^{d(T)}$. (*Beweis*: Induktion über Binärbaume, rekursive Struktur.)

(d) $N - 1 \geq d(T) \geq \lceil \log N \rceil$.

Merke:

Die Tiefe eines Binärbaums mit N **nichtleeren** äußeren Knoten ist $\geq \log N$.

Beweis von 3.3.2:

(a) Dies ist eine Umformulierung von Prop. 3.3.1(b).

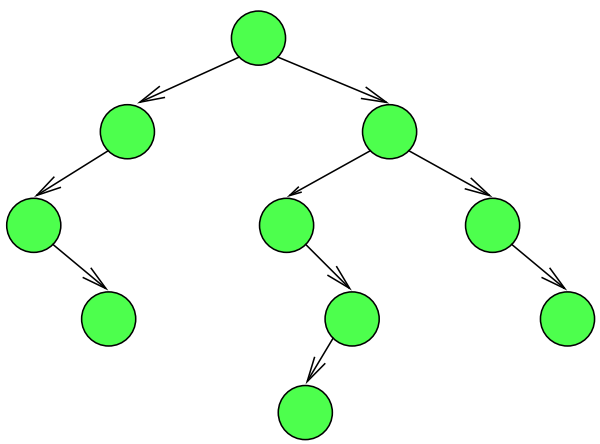
(b) Dies ist eine Umformulierung von Prop. 3.3.1(a).

(c) Durch (verallgemeinerte) Induktion über $d = d(T)$ zeigen wir, dass die Anzahl N_T der Blätter von T nicht größer als $2^{d(T)}$ ist.
I.A.: $d = 0$: Dann besteht T nur aus einem Blatt, also $N_T = 1$, und $2^0 = 1$.
Sei nun $d = d(T) > 0$.
I.V.: Behauptung ist richtig für alle Bäume mit Tiefe kleiner als d.
I.-Schritt: T hat eine Wurzel, die innerer Knoten ist, und zwei Unterbäume T_1 und T_2 mit $\max\{d(T_1), d(T_2)\} = d - 1$.
Klar: Baum T hat $N_T = N_{T_1} + N_{T_2}$ Blätter.
Nach I.V. gilt $N_{T_i} \leq 2^d$, also $N_{T_i} \leq 2^{d-1}$, für $i = 1, 2$.
Also: $N_T = N_{T_1} + N_{T_2} \leq 2 \cdot 2^{d-1} = 2^d$.

(d) Ergebnis (c) bedeutet $N_T \leq 2^{d(T)}$. Logarithmieren liefert $\log(N_T) \leq d(T)$. □

Totale innere Weglänge

Laufe nacheinander von der Wurzel zu jedem **inneren** Knoten v.
Wieviele Kanten werden **insgesamt** durchlaufen?



$$0 + 2 \cdot 1 + 3 \cdot 2 + 3 \cdot 3 + 1 \cdot 4 = 21$$

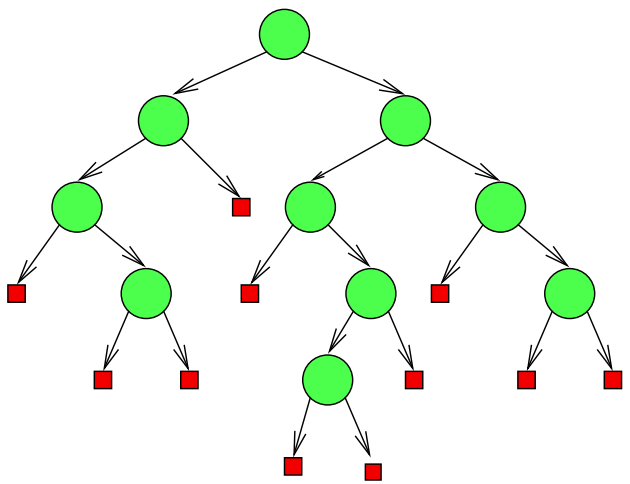
Totale innere Weglänge („**t**otal **i**nternal **p**ath **l**ength“):

TIPL$(T) := \sum_{v \in V} d(v)$. (Im Beispiel: 21.)

Mittlere innere Weglänge: $\frac{1}{n} \cdot \mathsf{TIPL}(T)$. (Im Beispiel: $\frac{21}{10}$.)

Totale äußere Weglänge

Laufe nacheinander von der Wurzel zu jedem **äußeren** Knoten ℓ.
Wieviele Kanten werden **insgesamt** durchlaufen?



$$2 + 3 \cdot 3 + 5 \cdot 4 + 2 \cdot 5 = 41.$$

Totale äußere Weglänge („**t**otal **e**xternal **p**ath **l**ength“):

TEPL$(T) := \sum_{l \in L} d(l)$. (Im Beispiel: 41.)

Mittlere äußere Weglänge: TEPL$(T)/(n+1)$. (Im Beispiel $\frac{41}{11}$.)

Proposition 3.3.3

Für jeden Binärbaum mit n inneren Knoten gilt: $\mathsf{TEPL}(T) = \mathsf{TIPL}(T) + 2n$.

(Im Beispiel: $41 = 21 + 2 \cdot 10$.)

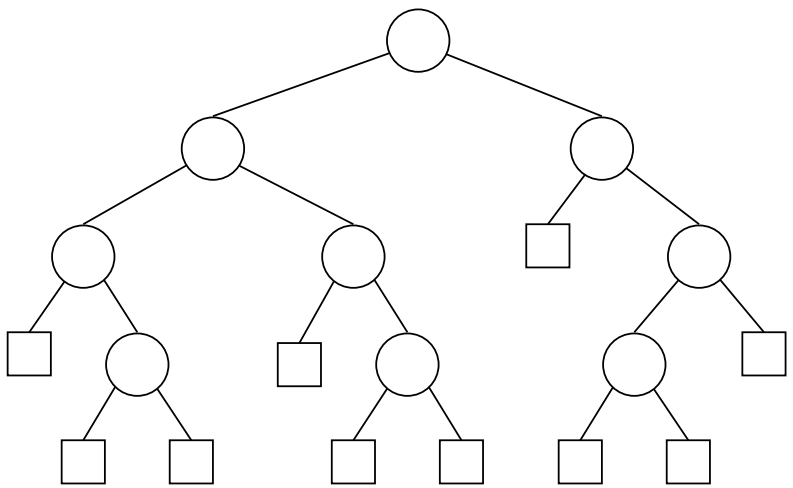
Beweis: Induktion über die Knotenanzahl n.

I.A.: $n = 0$: $\mathsf{TEPL}(T) = 0$ (ein Weg der Länge 0) und
$\mathsf{TIPL}(T) = 0$ (leere Summe, da kein innerer Knoten vorhanden).
Sei nun T Baum mit $n > 0$ Knoten. **I.V.:** Die Behauptung gilt für Bäume mit $n - 1$ Knoten.
I.-Schritt: In T gibt es einen inneren Knoten v, der zwei externe Kinder hat. Sei k die Tiefe von v.
Ersetze v und seine beiden Kinder durch neuen externen Knoten ℓ. Dies liefert T' mit $n - 1$ Knoten.
Dann:

$$\begin{aligned} \mathsf{TEPL}(T') &= \mathsf{TEPL}(T) - 2(k+1) + k &&= \mathsf{TEPL}(T) - k - 2 && [2 \text{ Kinder von } v \text{ weg, } \ell \text{ dazu}];\\ \mathsf{TIPL}(T') & && = \mathsf{TIPL}(T) - k && [v \text{ weg}]. \end{aligned}$$

Durch Subtrahieren: $\mathsf{TEPL}(T') - \mathsf{TIPL}(T') = \mathsf{TEPL}(T) - 2 - \mathsf{TIPL}(T)$.
Nach I.V.: $\mathsf{TEPL}(T') - \mathsf{TIPL}(T') = 2(n-1)$.
Es folgt $\mathsf{TEPL}(T) - \mathsf{TIPL}(T) = 2n$, das ist die I.-Beh. □

Nächstes Ziel: Auch die **mittlere Weglänge** in Binärbäumen wächst mindestens **logarithmisch** mit der Knotenanzahl.

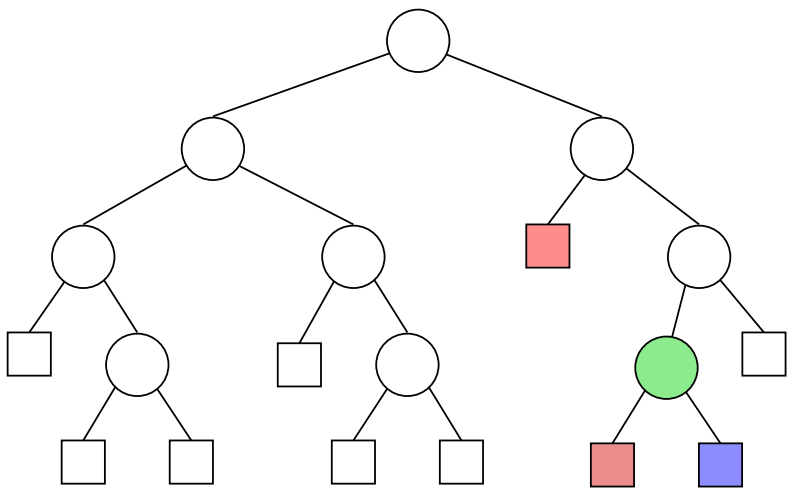


Lemma

Sei T Binärbaum mit N Blättern und **kleinstmöglicher** totaler externer Weglänge. Dann gibt es in T keine zwei Blätter ℓ und ℓ' mit $d(\ell) \geq d(\ell') + 2$.

Beweis hiervon: Sonst könnte man TEPL(T) durch Umbauen verkleinern, wie im Bild.

Nächstes Ziel: Auch die **mittlere Weglänge** in Binärbäumen wächst mindestens **logarithmisch** mit der Knotenanzahl.

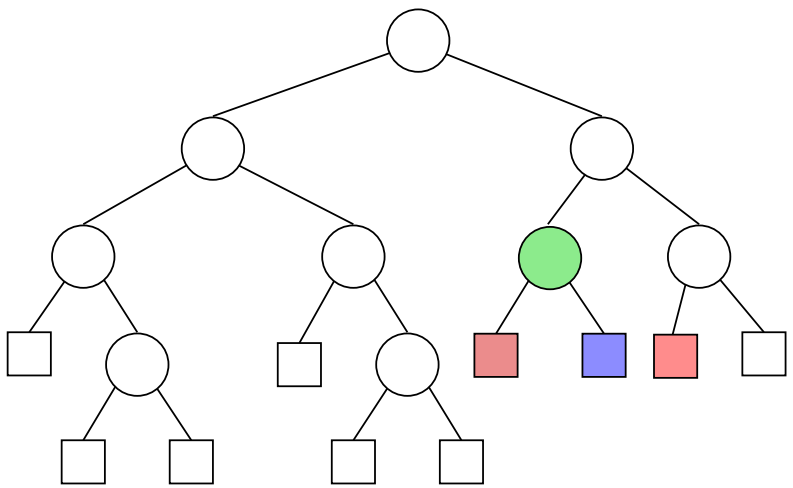


Lemma

Sei T Binärbaum mit N Blättern und **kleinstmöglicher** totaler externer Weglänge. Dann gibt es in T keine zwei Blätter ℓ und ℓ' mit $d(\ell) \geq d(\ell') + 2$.

Beweis hiervon: Sonst könnte man TEPL(T) durch Umbauen verkleinern, wie im Bild.

Nächstes Ziel: Auch die **mittlere Weglänge** in Binärbäumen wächst mindestens **logarithmisch** mit der Knotenanzahl.



Lemma

Sei T Binärbaum mit N Blättern und **kleinstmöglicher** totaler externer Weglänge. Dann gibt es in T keine zwei Blätter ℓ und ℓ' mit $d(\ell) \geq d(\ell') + 2$.

Beweis hiervon: Sonst könnte man TEPL(T) durch Umbauen verkleinern, wie im Bild.

Proposition 3.3.5 Binärbaum T hat N äußere Knoten $\Rightarrow$ TEPL$(T) \geq N \log N$.

Beweis: Wähle T mit kleinstmöglichem Wert TEPL(T) unter allen Bäumen mit N Blättern.

Sei $d = d(T)$ die Tiefe von T. Nach dem Lemma gibt es in T Blätter auf Level d und vielleicht auch auf Level $d-1$. Sei $a := \#$(Blätter auf Level $d-1$) und $b := \#$(Blätter auf Level d).

Klar: $a + b = N$. Weiter gilt: $2a + b = 2^d$. (Ersetze Blätter auf Level $d-1$ durch einen inneren Knoten mit zwei Blättern als Kindern. Dann hat Level d genau 2^d Blätter.) Umstellen ergibt:

$$a = 2^d - N \text{ und } b = 2N - 2^d.$$

Es gilt: $(*)$ TEPL$(T) = a(d-1) + bd = (2^d - N)(d-1) + (2N - 2^d)d = N(d+1) - 2^d$.

Beobachte: Weil $a \geq 0$, gilt $N \leq 2^d$, und weil $b > 0$, gilt $N > 2^{d-1}$.

Wir diskutieren die Funktion

$$f\colon [2^{d-1}, 2^d] \to \mathbb{R},\ f(x) = x \log x - (x(d+1) - 2^d).$$

Wir haben:

$f(2^{d-1}) = 2^{d-1}(d-1) - (2^{d-1}(d+1) - 2^d) = 0$ und $f(2^d) = 2^d d - (2^d(d+1) - 2^d) = 0$.

f ist **konvex** in $[2^{d-1}, 2^d]$ (denn $f'(x) = \log x + \log e - (d+1)$ und $f''(x) = (\log e)/x > 0$).

Zwischen seinen beiden Nullstellen muss f negativ sein, also gilt $(x(d+1) - 2^d) \geq x \log x$ für $x \in [2^{d-1}, 2^d]$. Mit $(*)$ folgt TEPL$(T) = (N(d+1) - 2^d) \geq N \log N$. □

Korollar 3.3.6

Für die **mittlere äußere Weglänge** eines Binärbaums T mit N Blättern gilt:

$$\boxed{\frac{1}{N}\mathsf{TEPL}(T) \geq \log N.}$$

Nach Proposition 3.3.3: $\mathsf{TEPL}(T) = \mathsf{TIPL}(T) + 2n$.
Also: $\mathsf{TIPL}(T) \geq \mathsf{TEPL}(T) - 2n \geq (n+1)\log(n+1) - 2n > n(\log n - 2)$.

Korollar 3.3.7

Für die **mittlere innere Weglänge** eines Binärbaums T mit n Knoten gilt:

$$\boxed{\frac{1}{n}\mathsf{TIPL}(T) > \log n - 2.}$$

Teil 6: Baumdurchläufe

3.4 Durchläufe durch Binärbäume

(D, Z)-Binärbäume kann man systematisch durchlaufen, die Knoten nacheinander besuchen und (anwendungsabhängige) Aktionen an den Knoten ausführen.

Abstrakt gefasst: `visit`-Operation.
Organisiere *data*-Teil der Knoten als Objekte einer Klasse;
`visit`(T) löst Abarbeitung einer Methode von *data*(T) aus.

Möglichkeiten für `visit` bei Knoten v:

– Daten *data*(v) ausgeben;

– laufende Nummer für v vergeben;

– Zeiger auf v an Liste anhängen;

– zu v gehörende Aktion ausführen (z. B. arithmetische Operation)

usw.

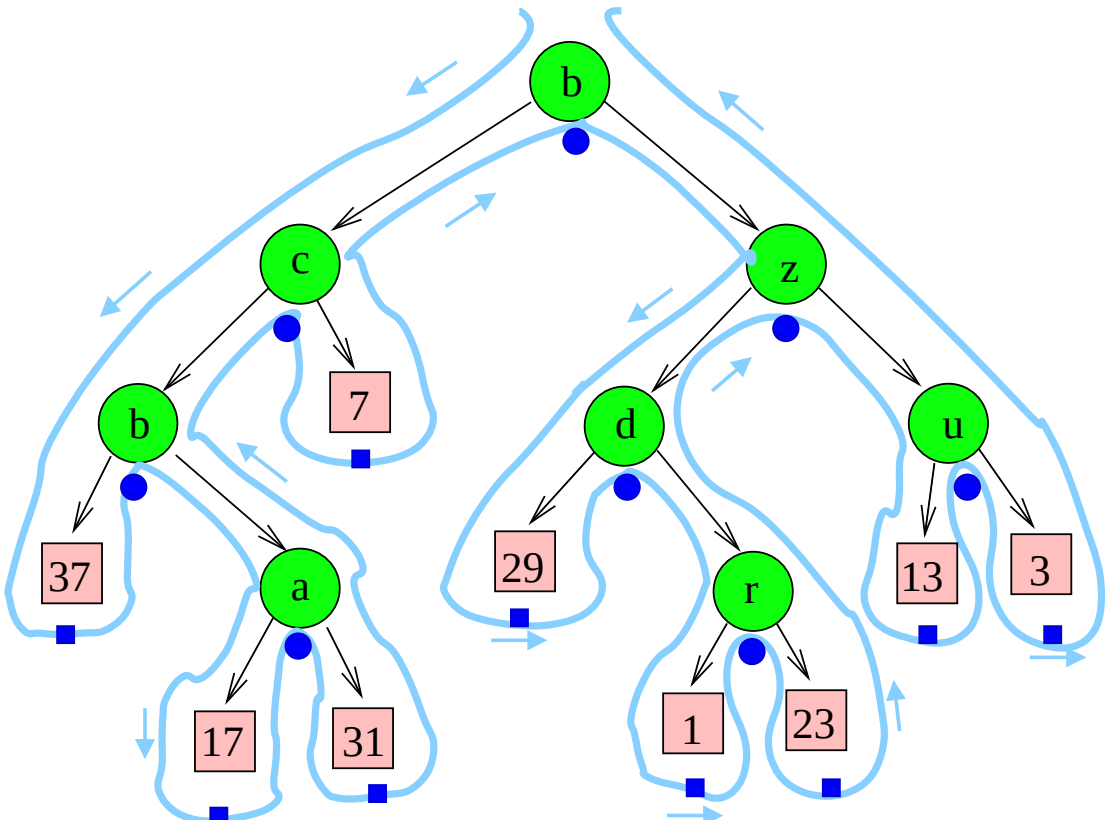
Inorder-Durchlauf durch T

falls $T = \boxed{z}$:
    `e-visit`(z) ; // besuche äußeren Knoten

falls $T = (T_1, x, T_2)$:
    Inorder-Durchlauf durch T_1;
    `i-visit`(x) ; // besuche inneren Knoten
    Inorder-Durchlauf durch T_2.

Inorder: „Erst den linken Unterbaum, dann die Wurzel, dann den rechten Unterbaum.“

Inorder-Durchlauf durch T



Ausgabe: $37, \mathrm{b}, 17, \mathrm{a}, 31, \mathrm{c}, 7, \mathrm{b}, 29, \mathrm{d}, 1, \mathrm{r}, 23, \mathrm{z}, 13, \mathrm{u}, 3.$

Beobachte: Man besucht abwechselnd externe und interne Knoten.

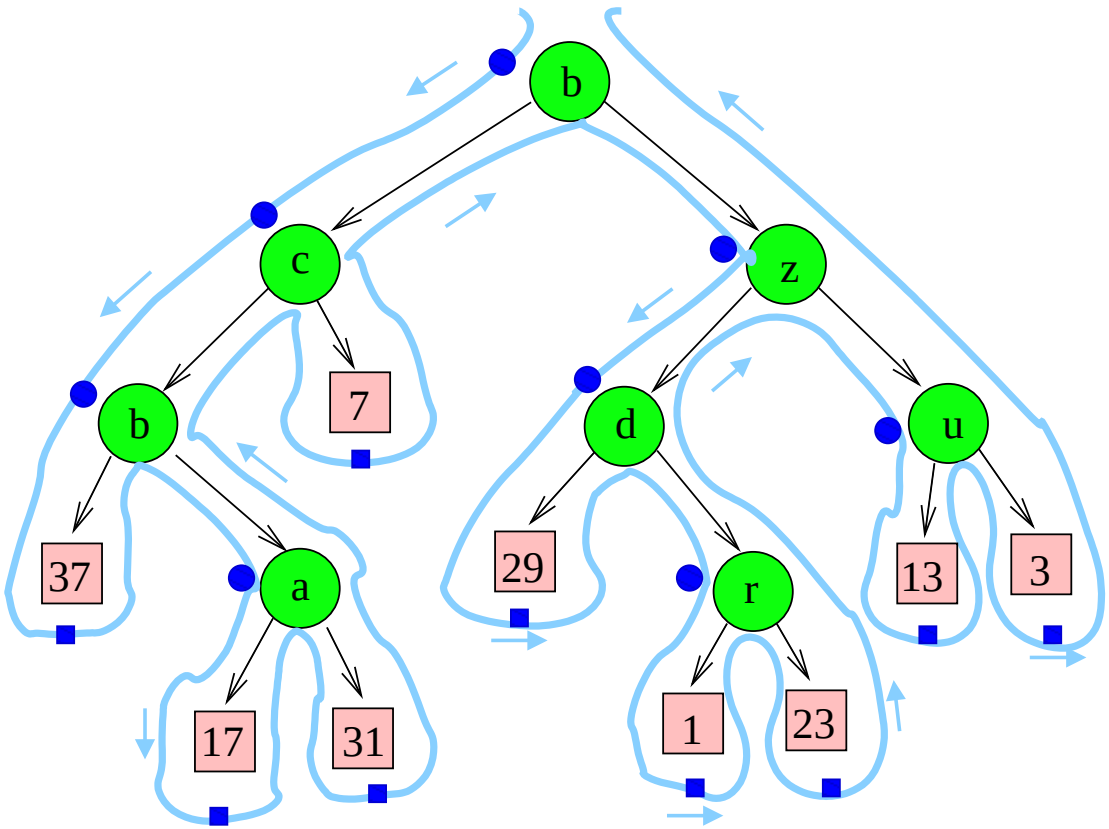
Präorder-Durchlauf durch T

falls $T = \boxed{z}$:
    `e-visit`(z) ;

falls $T = (T_1, x, T_2)$:
    `i-visit`(x) ;
    Präorder-Durchlauf durch T_1;
    Präorder-Durchlauf durch T_2;

Präorder:	„Erst die Wurzel, dann den linken Unterbaum, dann den rechten Unterbaum.“

Präorder-Durchlauf durch T



Ausgabe: $\mathrm{b}, \mathrm{c}, \mathrm{b}, 37, \mathrm{a}, 17, 31, 7, \mathrm{z}, \mathrm{d}, 29, \mathrm{r}, 1, 23, \mathrm{u}, 13, 3$

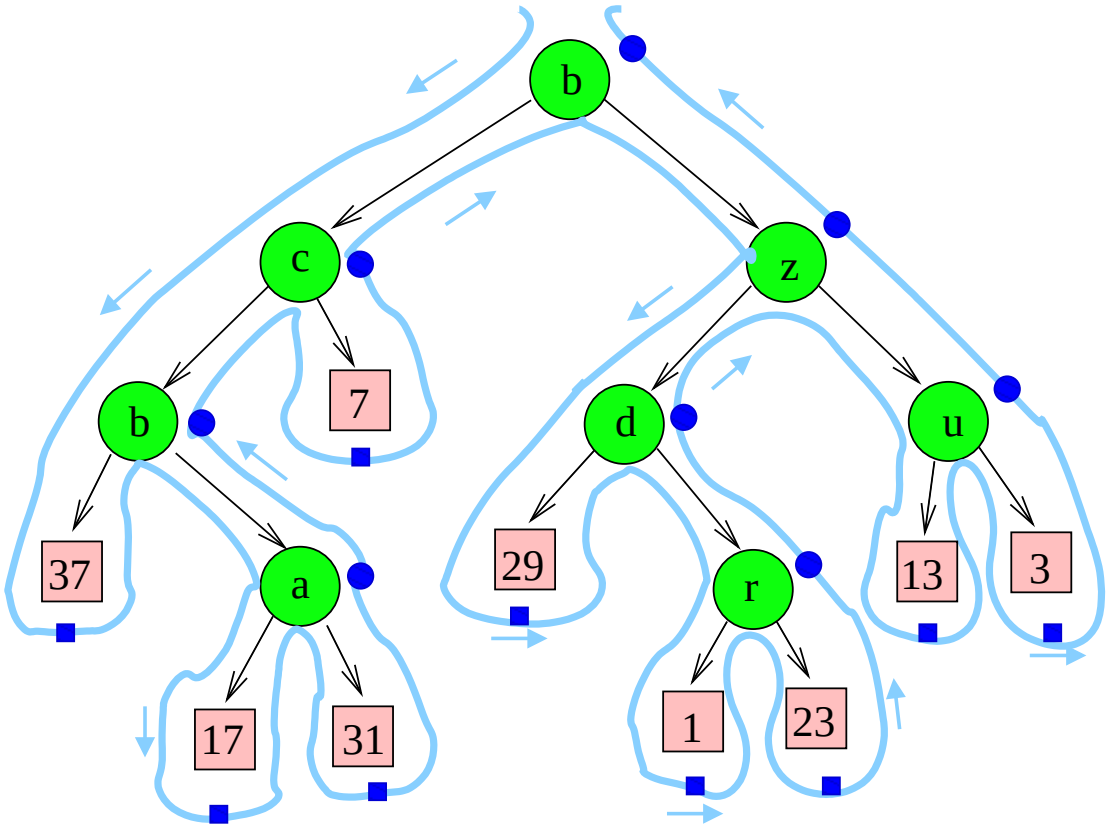
Postorder-Durchlauf durch T

falls $T = \boxed{z}$:
    `e-visit`(z) ;

falls $T = (T_1, x, T_2)$:
    Postorder-Durchlauf durch T_1;
    Postorder-Durchlauf durch T_2;
    `i-visit`(x) ;

Postorder: „Erst den linken Unterbaum, dann den rechten Unterbaum, zuletzt die Wurzel.“

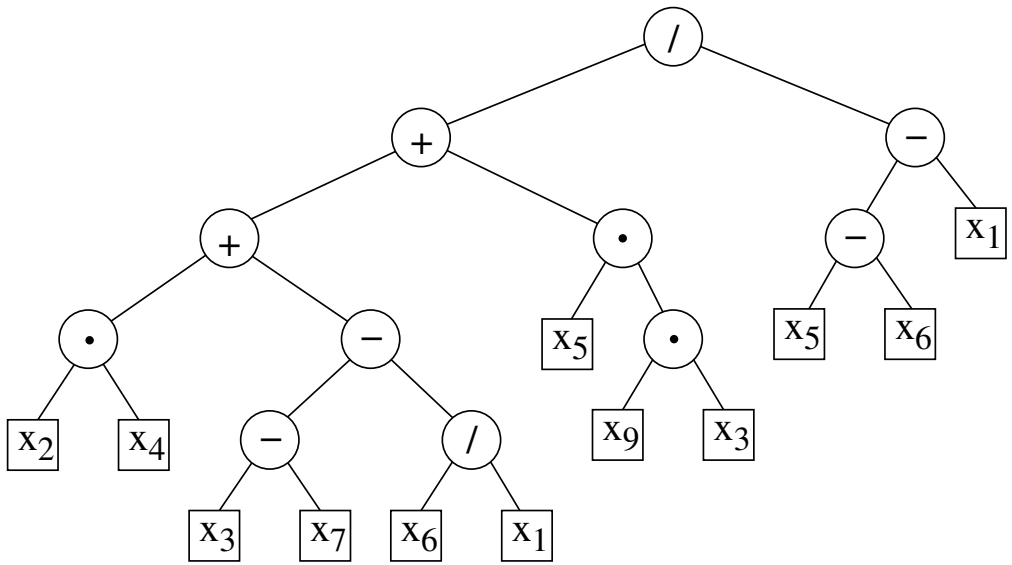
Postorder-Durchlauf durch T



Ausgabe: $37, 17, 31, \mathrm{a}, \mathrm{b}, 7, \mathrm{c}, 29, 1, 23, \mathrm{r}, \mathrm{d}, 13, 3, \mathrm{u}, \mathrm{z}, \mathrm{b}.$

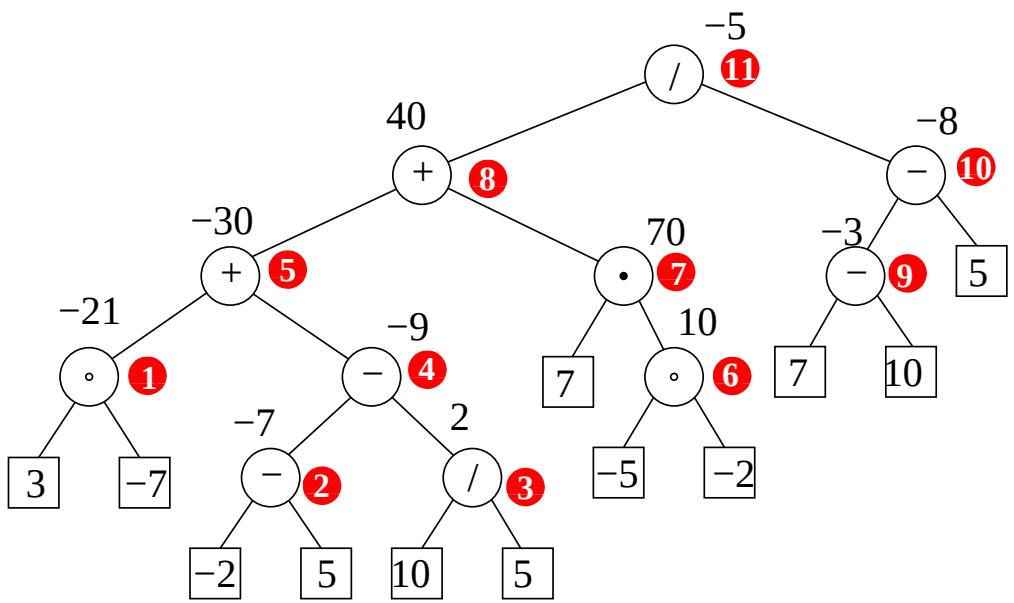
Beispiel: **Arithmetischer Ausdruck als Baum**

$D = \{+, -, \cdot, /\}$ und $Z = \{x_1, x_2, x_3, \ldots\}$.



Beispiel: **Arithmetischer Ausdruck als Baum**

$D = \{+, -, \cdot, /\}$ und $Z = \{x_1, x_2, x_3, \ldots\}$.



Ordne den Blättern konkrete Werte zu, z.B. $x_1 = 5$, $x_2 = 3$, $x_3 = -2$, $x_4 = -7$, $x_5 = 7$, $x_6 = 10$, $x_7 = 5$, $x_9 = -5$. Dann ordne jedem Knoten einen Wert zu. (Ergebnis des Unterbaums.)
Geeignete Reihenfolge: Postorder-Durchlauf. **(Rote Kreise.)**

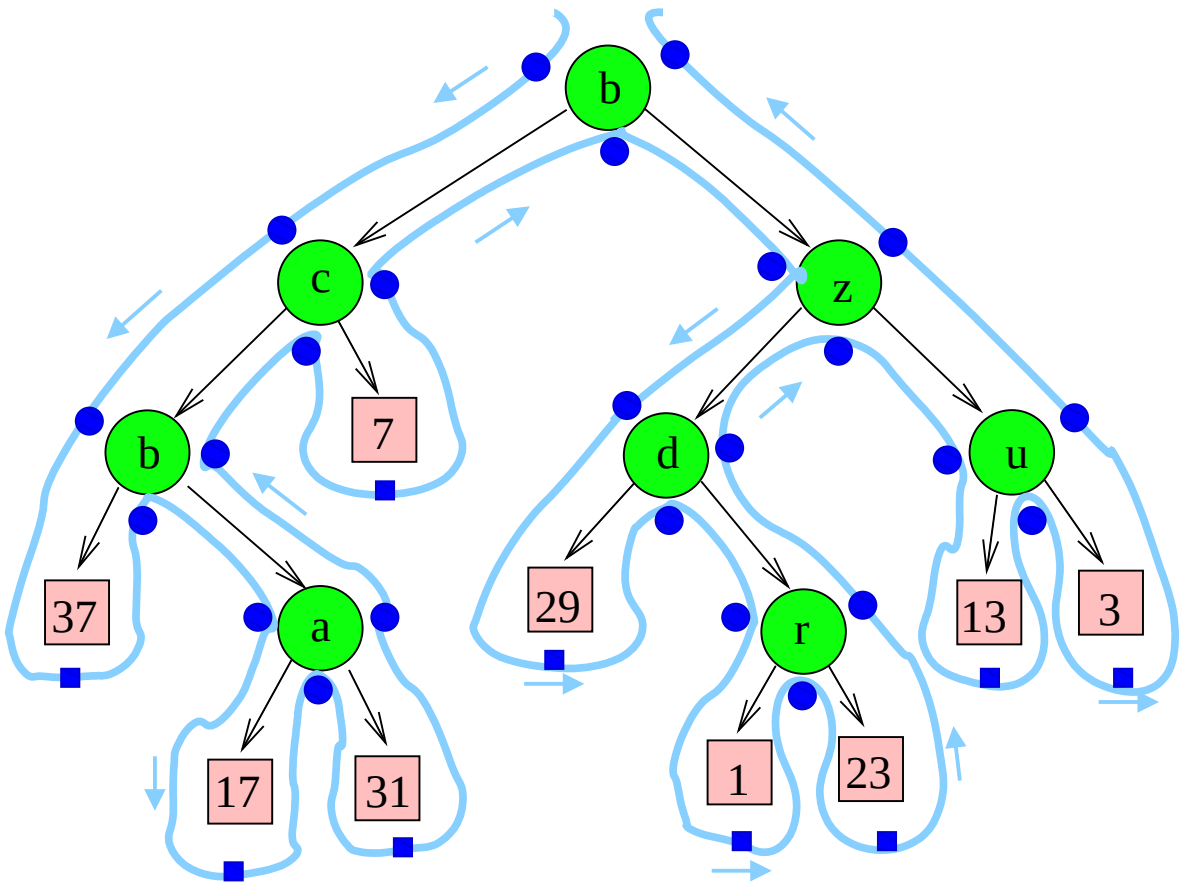
Auch möglich: Kombi-Durchlauf:

falls $T = \boxed{z}$:
    `e-visit`(z) ;

falls $T = (T_1, x, T_2)$:
    `preord-i-visit`(x) ;
    Kombi-Durchlauf durch T_1;
    `inord-i-visit`(x) ;
    Kombi-Durchlauf durch T_2;
    `postord-i-visit`(x) ;

Ermöglicht Datentransport von der Wurzel nach unten in den Baum hinein und wieder zurück.

Kombi-Durchlauf durch T



Beispiel für Anwendung eines Kombi-Durchlaufs:

Gegeben: Binärbaum T als verzeigerte Struktur (Folie 16).

Aufgabe: Generiere die rekursive Darstellung von T, aufgeschrieben als ein String (Folien 11–12).

Zeichen sollen sequenziell in die Ausgabe geschrieben werden („**print**“).

Dazu muss man nur die vier Besuchsmethoden geeignet spezifizieren:

`e-visit`(z):	**print**("(",z,")").
`preord-i-visit`(x):	**print**("(").
`inord-i-visit`(x):	**print**(",",x,",").
`postord-i-visit`(x):	**print**(")").

Man sollte dies an einem Beispiel testen.

Zeitanalyse für Baumdurchlauf

Die Baumdurchläufe können mittels rekursiver Prozeduren implementiert werden.

Beispiel: Inorder-Durchlauf. Prozedur: `inorder(T)`

Angewendet auf einen Baum T mit n inneren Knoten.

Für jeden inneren und äußeren Knoten wird die Prozedur `inorder(.)` einmal aufgerufen, insgesamt $(2n+1)$-mal.

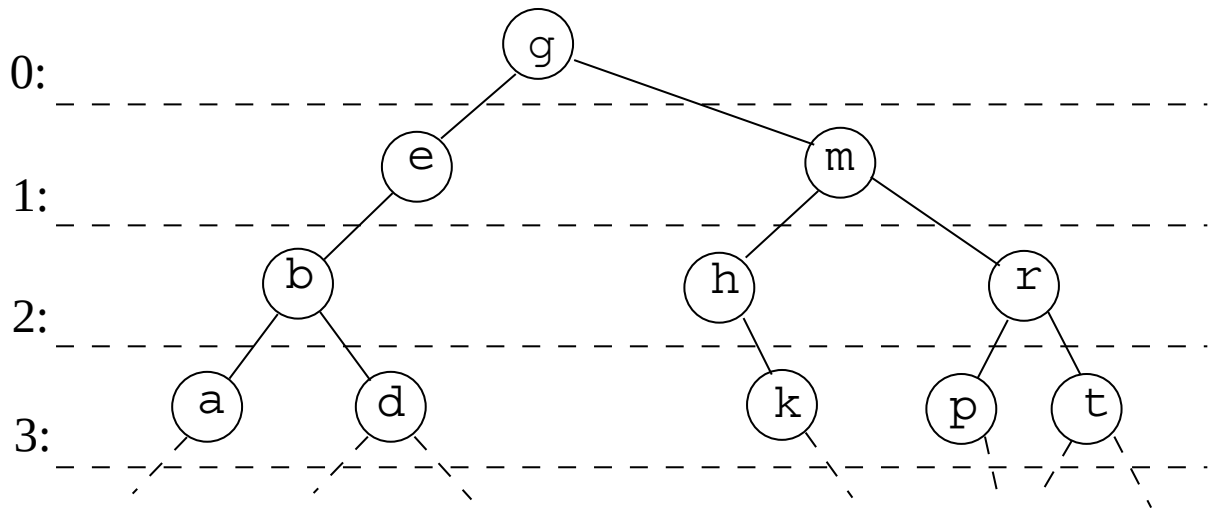
Kosten pro Aufruf: $O(1)$ plus Kosten für `i-visit`/`e-visit`-Operation.

Insgesamt: $(2n+1) \cdot (O(1) + C_{\texttt{visit}})$,
wobei $C_{\texttt{visit}}$ eine Schranke für die Kosten der `visit`-Operationen ist.

Behauptung 3.4.1

Baumdurchläufe haben lineare Laufzeit.

Levelorder-Durchlauf



Wunschausgabe: `g em bhr adkpt` , „levelweise von links nach rechts“.

Datenstruktur:
Queue `Q`, die Zeiger/Referenzen auf Binärbaumknoten speichern kann.

Prozedur Leveldurchlauf (T: Baum mit leeren äußeren Knoten)
Eingabe: v: node; // Wurzel von T

```
(1)   Q: Queue für node;
(2)   count ← 0;
(3)   if v ≠ NULL then // entdecke Wurzel
(4)     count++; v.lnum ← count;
(5)     v.level ← 0; enqueue(v);
(6)   while not Q.isempty do
(7)     w ← Q.first; Q.dequeue;
(8)     visit(w);  // bearbeite w
(9)     p ← w.leftchild;
(10)    if p ≠ NULL then
(11)      count++; p.lnum ← count; // entdecke p
(12)      p.level ← w.level + 1; enqueue(p);
(13)    p ← w.rightchild;
(14)    if p ≠ NULL then
(15)      count++; p.lnum ← count; // entdecke p
(16)      p.level ← w.level + 1; enqueue(p);
```

Behauptung 3.4.2 „Lineare Zeit, Korrektheit“

(a) Levelorder-Durchlauf benötigt Zeit $n \cdot t_{\texttt{visit}} + O(n)$,
wobei $t_{\texttt{visit}}$ die Zeit für `visit` angibt.

(b) Der Levelorder-Durchlauf ordnet jedem Binärbaumknoten v (in v`.level`)
korrekt seine Tiefe zu.

Beweis: Man zeigt durch Induktion über das Level von v:

Jeder innere Knoten v des Binärbaums wird genau einmal in die Queue **eingefügt**,
genau einmal **entnommen** und im Schleifenrumpf `(7)-(16)` **bearbeitet**.
Dabei wird die korrekte Levelnummer vergeben.

Weil Queue-Operationen Zeit $O(1)$ benötigen, ist der restliche Zeitbedarf $O(n)$. □

Ergänzung: (Angeordnete) Bäume mit beliebiger Anzahl von Kindern

Anstelle von Binärbäumen kann man auch Bäume betrachten, bei denen ein (innerer) Knoten v eine beliebige Anzahl $d_v \geq 0$ von Kindern haben kann. Wenn $d_v = 0$ ist, ist v ein Blatt. Dabei hat ein Knoten v nicht ein „linkes“ und ein „rechtes“ Kind, sondern Kind 1, . . . , Kind d_v (ohne die Möglichkeit eines „leeren“ Kindplatzes).

Induktive Definition von „Mehrwegbaum mit Knotenmarkierungen aus D“:
(i) Für $x \in D$ ist (x) ein Mw-D-Baum.
(ii) Wenn $d \geq 1$ ist und $T_1, \ldots, T_d$ Mw-D-Bäume sind und $x \in D$ ist,
dann ist auch $(x, T_1, \ldots, T_d)$ ein Mw-D-Baum.

Implementierung solcher Bäume: Ein Knoten enthält einen Dateneintrag aus D und eine (lineare) Liste oder ein Array von Zeigern auf Kindknoten. In [AuP], Kap. 10, wird skizziert, wie sich solche Bäume mit Hilfe von Binärbaumknoten darstellen lassen.

Für solche Mehrwegbäume sind Präorder- und Postorder-Durchlauf sowie Levelorder-Durchlauf sinnvoll (und benötigen lineare Zeit).

Uns werden solche Bäume später als **Mehrweg-Suchbäume** und im Kontext von **Graphdurchläufen** („Breitensuchbaum“, „Tiefensuchbaum“) begegnen.

ENDE

3. Kapitel

SS 2021

Algorithmen und Datenstrukturen

4. Kapitel

Suchbäume

Martin Dietzfelbinger

Mai 2021

4.1 Binäre Suchbäume

Binäre Suchbäume **implementieren den Datentyp Wörterbuch**.

Zu diesem Datentyp gehören:

Schlüsselmenge U und Wertemenge R

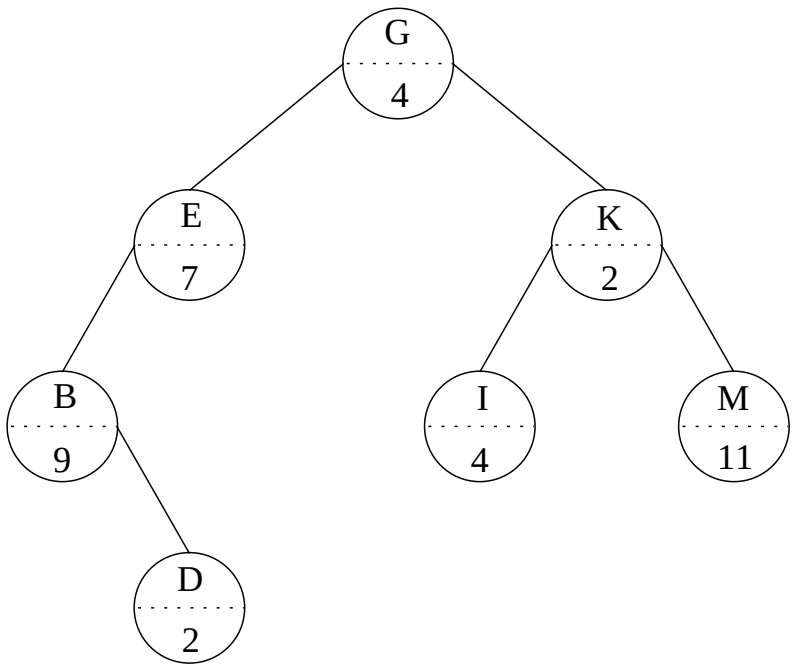
Wörterbuch: $f : S \to R$, wobei $S \subseteq U$ endlich.

- *empty*(): leeres Wörterbuch erzeugen
- *lookup*(f, x): falls $x \in S$: $f(x)$ ausgeben (sonst: „undefined“)
- *insert*(f, x, r): $f(x) := r$ setzen (für x neu in S, oder $f(x)$ auf r aktualisieren)
- *delete*(f, x): x aus $S = \mathrm{Def}(f)$ löschen, falls $x \in S$

Hier **zusätzlich** nötig: U ist durch $<$ **(total geordnet)**.

Beispiel: $U = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \ldots, \mathrm{Z}\}$ (Standardsortierung), $R = \mathbb{N}$.

Ein binärer Suchbaum T:



Dargestellte Funktion $f = f_T$:

$S = \mathrm{Def}(f) = \{\mathrm{B}, \mathrm{D}, \mathrm{E}, \mathrm{G}, \mathrm{I}, \mathrm{K}, \mathrm{M}\}$, $f(\mathrm{B}) = 9$, $f(\mathrm{D}) = 2$, usw.

Binärer Suchbaum

Knoteneinträge (hier in inneren Knoten v):

- *key*(v): Schlüssel (aus U, mit $<$ Totalordnung auf U)
- *data*(v): Daten, Wert (aus R) (oft: Zeiger, Referenz)

Definition 4.1.1

Ein Binärbaum T mit Knoteneinträgen aus $U \times R$ heißt ein **binärer Suchbaum***, falls **für jeden Knoten v** in T gilt:

für alle Knoten w im **linken** Unterbaum von T_v: *key*$(w) <$ *key*(v)

für alle Knoten w im **rechten** Unterbaum von T_v: *key*$(v) <$ *key*(w)

---

engl.: *binary search tree*. Abk.: **(U, R)-BSB** oder **(U, R)-BST**

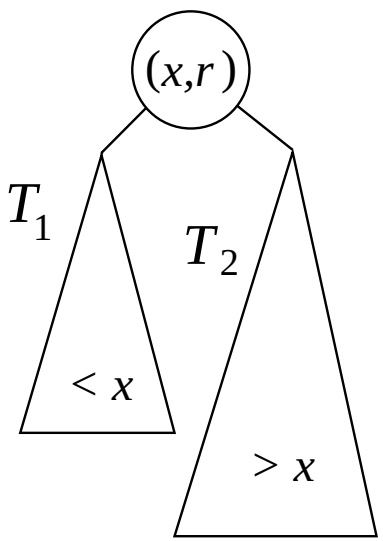
Alternative Definition: Rekursive Auffassung.

(i) $\square$ ist (U, R)-BSB, mit Schlüsselmenge $S = \emptyset$. $\square$

(ii) **Wenn** $x \in U$ und $r \in R$ **und**
T_1, T_2 sind (U, R)-BSB
mit Schlüsselmengen S_1 bzw. $S_2 \subseteq U$,

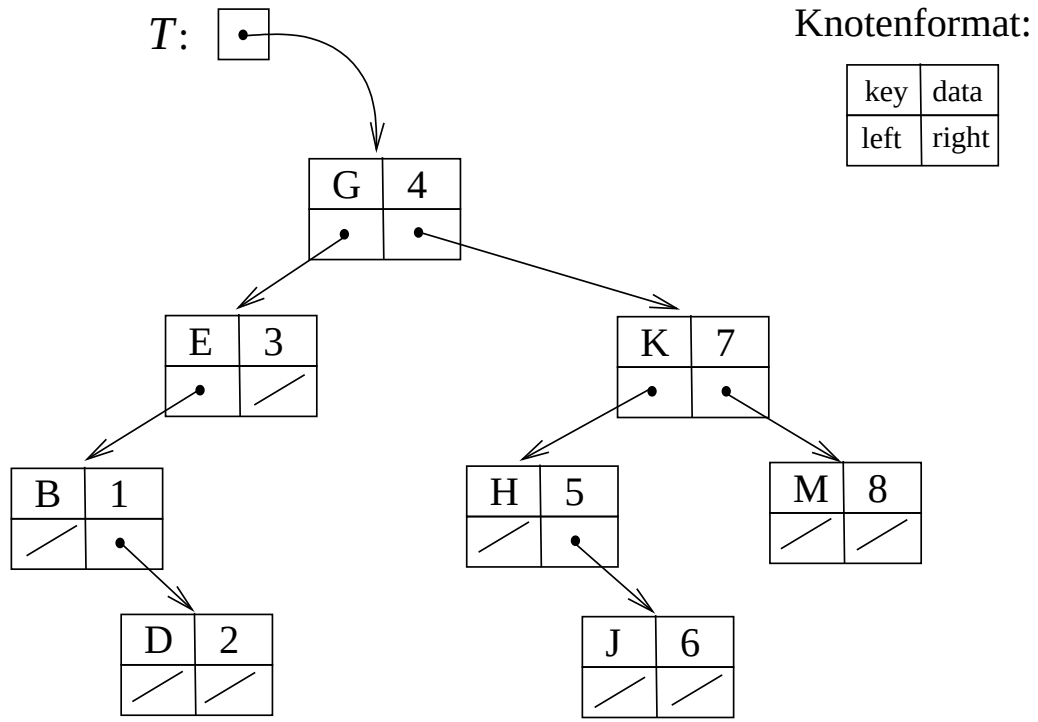
und wenn $y < x$ für alle $y \in S_1$
und $x < z$ für alle $z \in S_2$,

dann ist $(T_1, (x, r), T_2)$ ein (U, R)-BSB
mit Schlüsselmenge $S_1 \cup \{x\} \cup S_2$.



[(iii) Nichts sonst ist ein (U, R)-BSB.]

Falls (ii): $\mathit{key}(T) := x$ und $\mathit{data}(T) := r$.



Implementierung von binären Suchbäumen: Verzeigerte Struktur.

Knotenobjekt hat Komponenten (Attribute) für Schlüssel, Daten und Referenzen/ Zeiger auf linkes und rechtes Kind: $T.key$, $T.data$, $T.left(child)$, $T.right(child)$

Baum ist gegeben durch Zeiger/Referenz auf Wurzel.

Suchen, rekursiv

lookup(T, x), für BSB T, $x \in U$.

// Ergebnis: r, wenn (x, r) im Baum T vorhanden
// „nicht vorhanden“, wenn es kein solches Paar gibt.

1. Fall: $T = \square$: **return** „nicht vorhanden“

// Ab hier: $T \neq \square$, also $T = (T_1, (y, r), T_2)$.

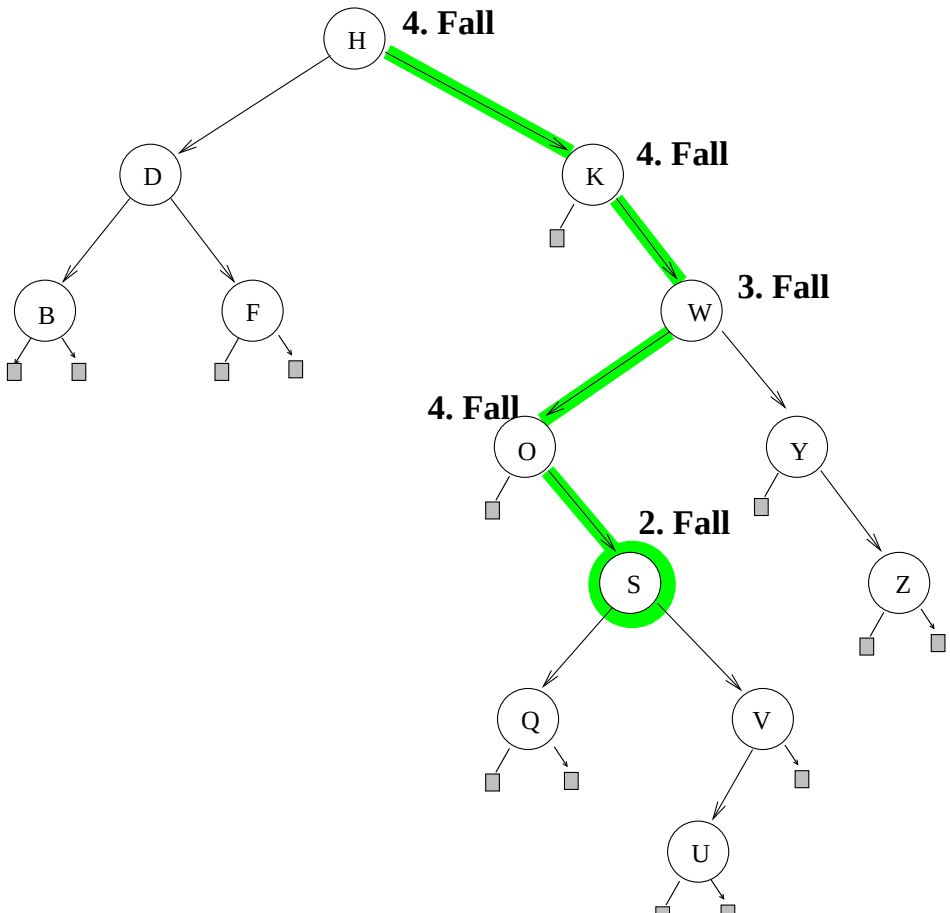
2. Fall: $x = y$: **return** r.

3. Fall: $x < y$: **return lookup**$(T.\mathit{left}, x)$ // rekursiver Aufruf.

4. Fall: $x > y$: **return lookup**$(T.\mathit{right}, x)$ // rekursiver Aufruf.

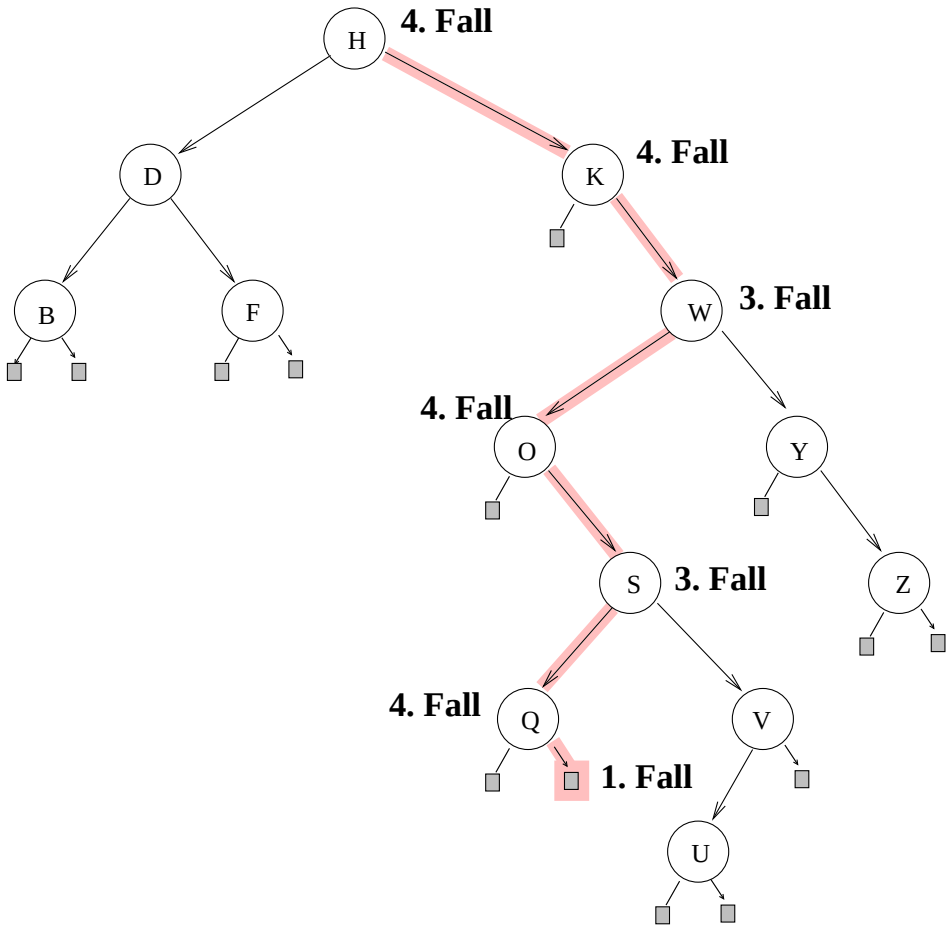
(**Beachte** die einfache Programmstruktur, die sich an rekursiver Definition von BSBen orientiert!)

Beispiel:



Suche: $\mathrm{S} \in S$.

Beispiel:



Suche: $\mathrm{R} \notin S$.

Korrektheit: Für BSB T definieren wir $\boldsymbol{f_T(x)} := r$, wenn es in T einen Knoten v mit $v.\mathit{key} = x$ und $v.\mathit{data} = r$ gibt; wenn es keinen Knoten v mit $v.\mathit{key} = x$ gibt, ist $x \notin \mathrm{Def}(f_T)$. D. h.: f_T ist die von T dargestellte Funktion. Per Induktion über den Aufbau von Binärbäumen zeigt man: Die Ausgabe von **lookup**(T, x) ist $f_T(x)$.

(Später gezeigt: **insert** und **delete** verändern f_T wie vom mathematischen Modell vorgeschrieben.)

Zeitaufwand: Wenn x in Knoten v_x sitzt:
Für jeden Knoten v auf dem Weg von der Wurzel zu v_x Kosten $O(1)$. Also:

$$\boxed{\text{Kosten}^* \quad O(d(v_x)) = O(d(T)).}$$

Wenn x in T nicht vorkommt: Suche endet in externem Knoten l_x. Kosten $O(1)$ für jeden Knoten auf dem Weg. Also:

$$\boxed{\text{Kosten } O(d(l_x)) = O(d(T)).}$$

* Lies (überall) $O(1 + d(v_x)) = O(1 + d(T))$ usw. – **Übung:** Suche **iterativ** programmieren.

Einfügen, rekursiv

insert(T, x, r), für BSB T, $x \in U, r \in R$.

// Ergebnis: Baum T' mit $f_{T'} = \mathit{insert}(f_T, x, r)$.

1. Fall: $T = \square$: Erzeuge neuen (Wurzel-)Knoten v mit Eintrag (x, r);
$T \leftarrow v$; **return** T.

// Ab hier: $T \neq \square$, also $T = (T_1, (y, r'), T_2)$.

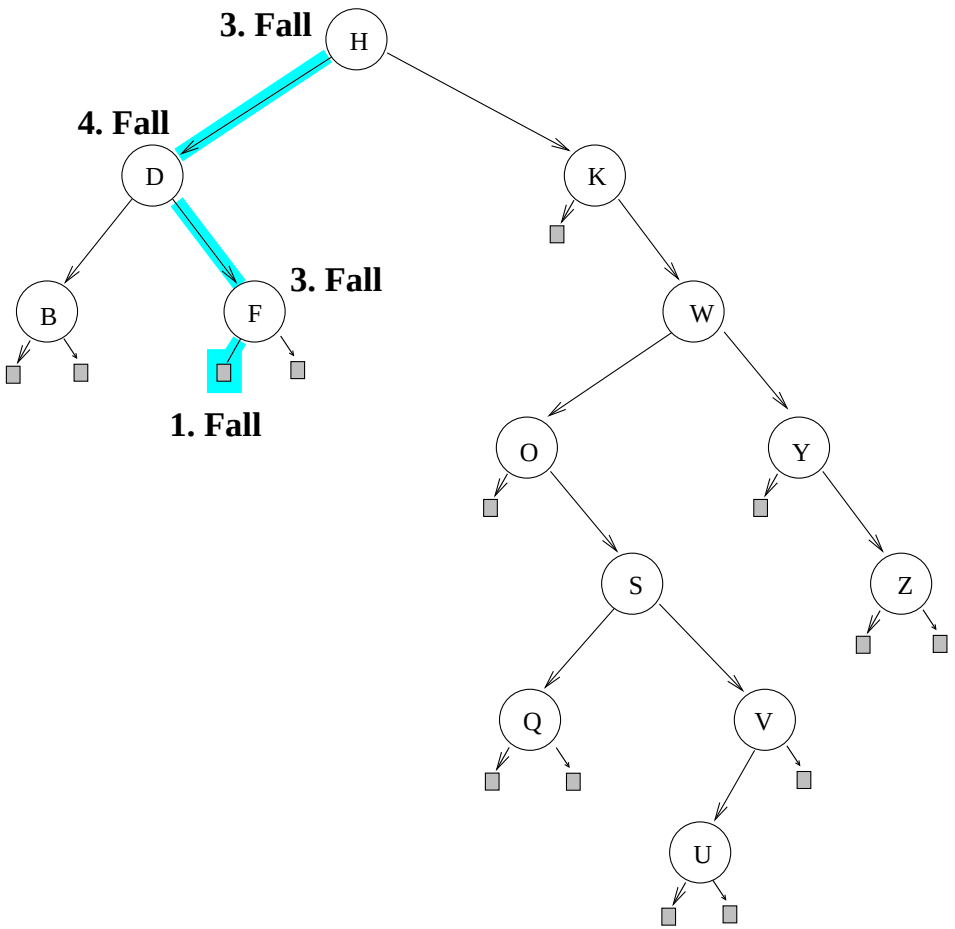
2. Fall: $x = y$: $T.\mathit{data} \leftarrow r$; **return** T. // **Update-Situation!**

3. Fall: $x < y$: $T.\mathit{left} \leftarrow$ **insert**$(T.\mathit{left}, x, r)$; **return** T.

4. Fall: $y < x$: $T.\mathit{right} \leftarrow$ **insert**$(T.\mathit{right}, x, r)$; **return** T.

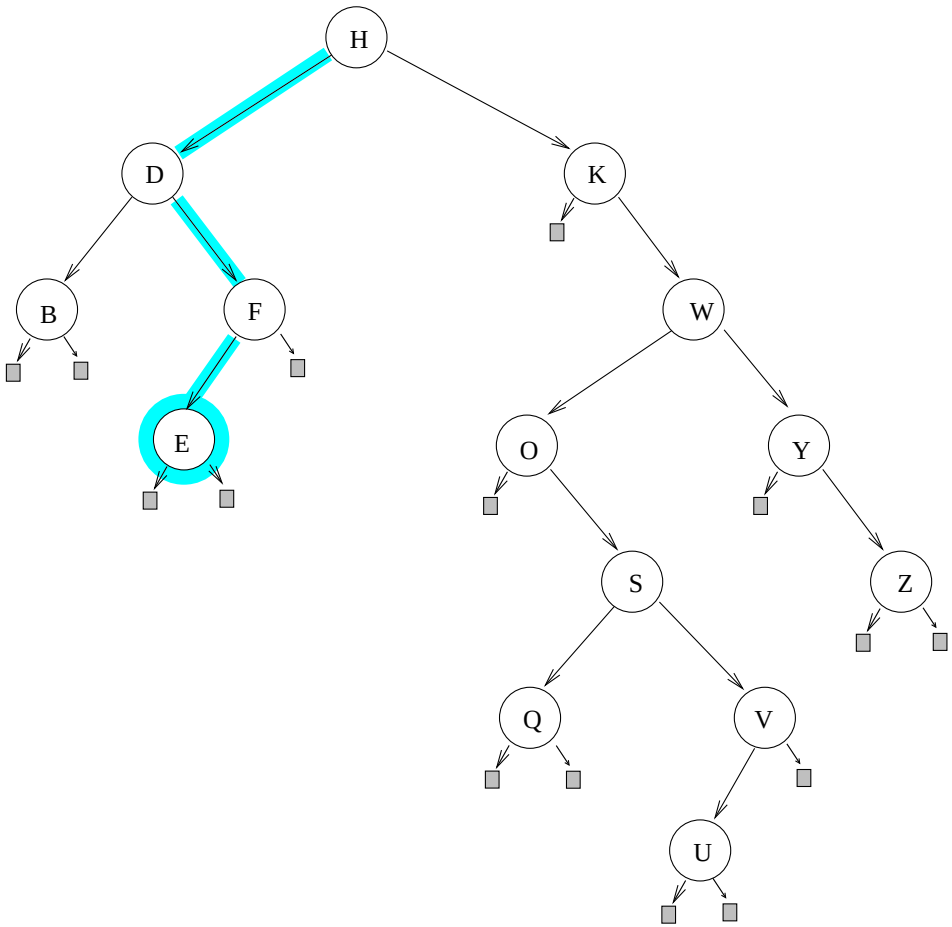
Beachte im 3. und im 4. Fall, dass implizit der Unterbaum „abgehängt“ und der Behandlung durch den rekursiven Aufruf übergeben wird. Durch den rekursiven Aufruf kann sich die Wurzel des Unterbaums verändern (oder aus einem `null`-Zeiger ein Zeiger auf einen echten Knoten werden). Der veränderte Unterbaum wird wieder an die richtige Stelle angehängt.

Beispiel:



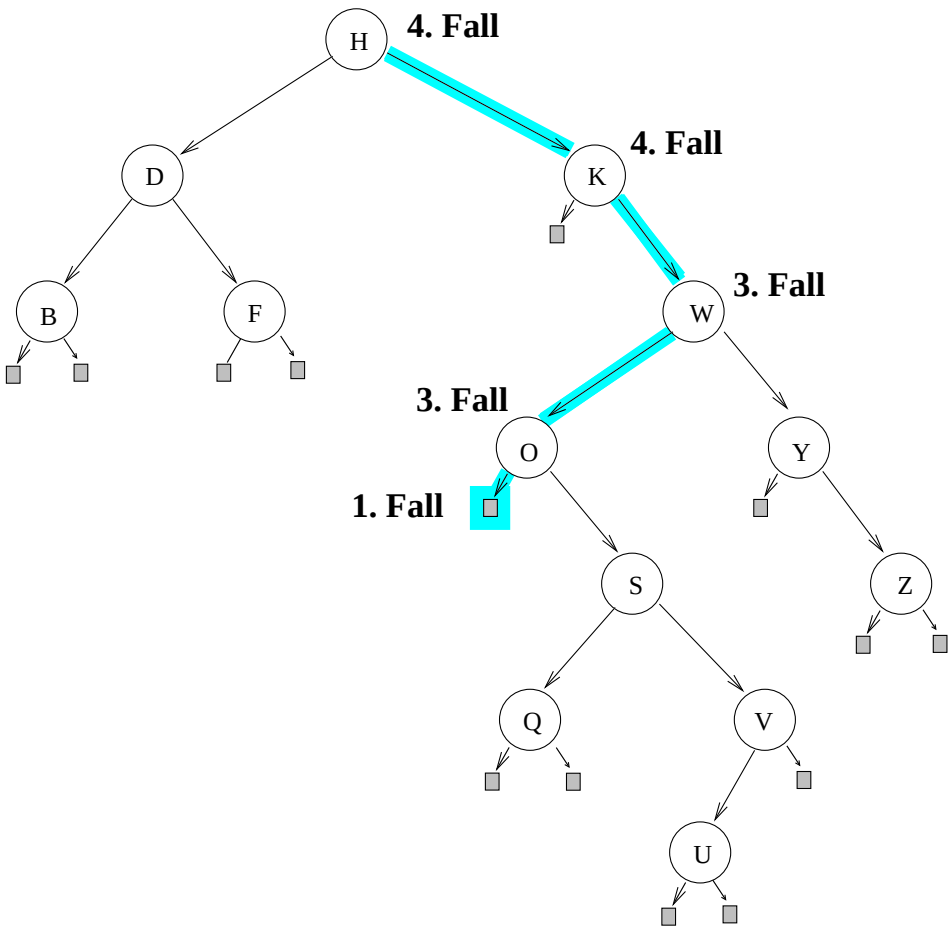
Füge ein: $\mathrm{E} \notin S$.

Beispiel:



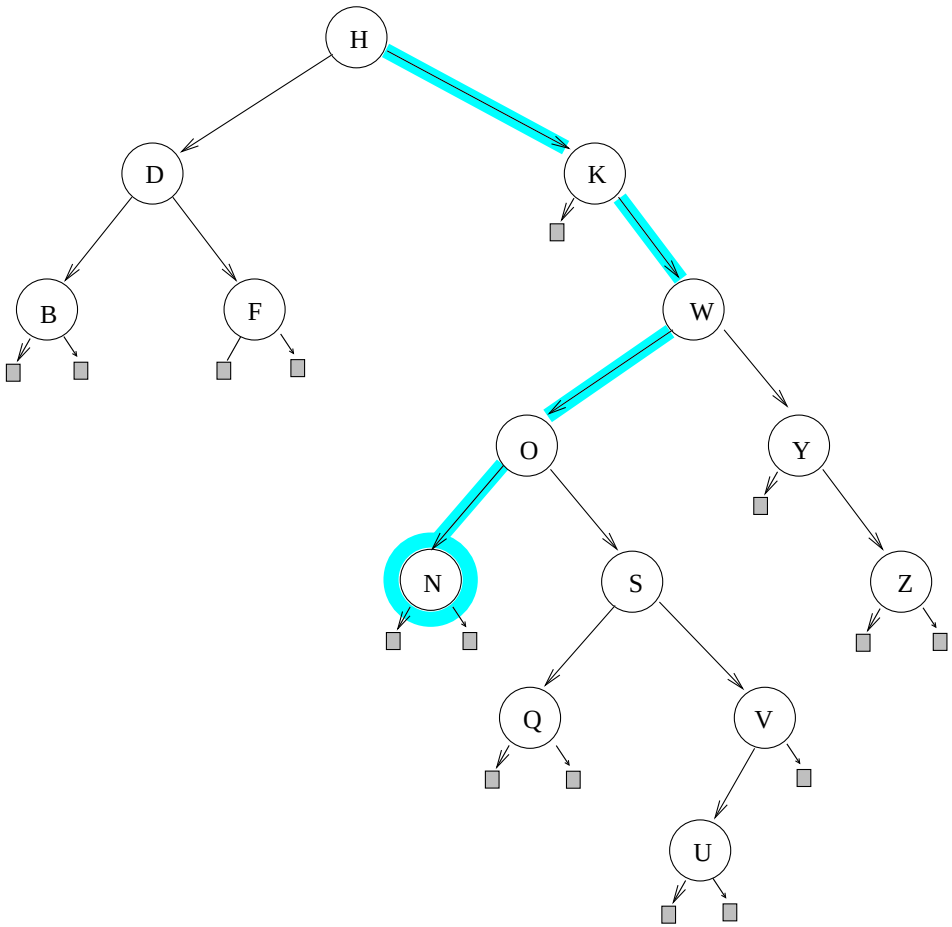
Füge ein: $\mathrm{E} \notin S$.

Beispiel:



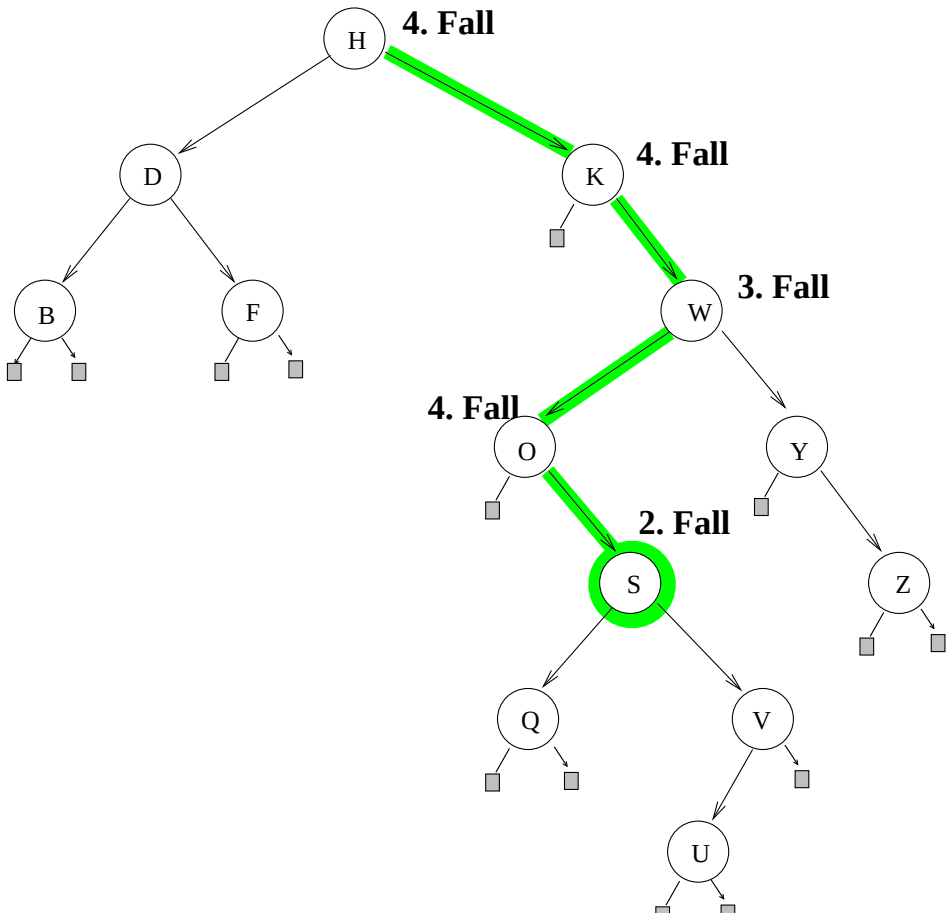
Füge ein: $\mathrm{N} \notin S$.

Beispiel:



Füge ein: $\mathrm{N} \notin S$.

Beispiel:



Füge ein: $\mathrm{S} \in S$.

Korrektheit:

Behauptung: Aufruf **insert**(T, x, r) erzeugt einen binären Suchbaum T', der das modifizierte Wörterbuch *insert*(f_T, x, r) darstellt. (f_T: die von T dargestellte Funktion.)

Beweis: Induktion über den Aufbau von Binärbäumen. (Siehe Druckfolien.)

Zeitaufwand: Wenn x im Knoten v_x schon vorhanden:

$$\boxed{\text{Kosten } O(d(v_x)) = O(d(T)).}$$

Wenn x nicht in T: $O(1)$ für jeden Knoten auf dem Weg von Wurzel zu l_x. Also:

$$\boxed{\text{Kosten } O(d(l_x)) \ = O(d(T)).}$$

Übung: Einfügung **iterativ** programmieren. *Hinweis*: Man muss immer auch den Vorgänger des aktuellen Unterbaums kennen.

(Für Interessierte.)
Beweis der Behauptung „Aufruf **insert**(T, x, r) erzeugt einen BST T', der *insert*(f_T, x, r) darstellt.“ durch **Induktion über den rekursiven Aufbau** von T.
(i) Wenn $T = \square$, dann ist f_T die leere Funktion. Der Algorithmus landet im 1. Fall. Es wird ein neuer Knoten v mit Eintrag (x, r) erzeugt und als T' zurückgegeben. Offenbar stellt dieser Baum dir Funktion *insert*$(f_T, x, r) = \{(x, r)\}$ dar.
(ii) Wenn $T = (T_1, (y, r'), T_2)$, gibt es mehrere Fälle.

2. Fall: $x = y$. Dann sieht T' genauso aus wie T, nur dass in der Wurzel der Wert r' durch r ersetzt wurde. Daher ist T' offenbar ein BSB und stellt *insert*(f_T, x, r) dar.
3. Fall: $x < y$. Dann erhält man T', indem man T_1 durch $T_1' =$ **insert**(T_1, x, r) ersetzt. Nach I.V. (kleinerer Baum!) ist T_1' ein BSB, und $T' = (T_1', (y, r'), T_2)$. Ist T' ein BSB? Unterbaum T_2 ist unverändert, also $> y$, in T_1' kommt eventuell Schlüssel $x < y$ hinzu. Also ist die Schlüsselanordnung korrekt. Stellt T' die Funktion *insert*(f_T, x, r) dar? In der Wurzel und in T_2 ist alles beim Alten. Durch T_1' wird nach I.V. $f_1' =$ *insert*(f_{T_1}, x, r) dargestellt. Egal ob nun (x, r) ein neues Paar ist oder im linken Unterbaum nur ein Update erfolgte, die Funktion f_1' zusammen mit (y, r') (Wurzel) und f_{T_2} bildet *insert*(f_T, x, r).
4. Fall: $y < x$: Wie 3. Fall. $\square$

Löschen, rekursiv

delete(T, x), für BSB T, $x \in U$.

// Ergebnis: Baum T' mit $f_{T'} = \mathit{delete}(f_T, x)$.

1. Fall: $T = \square$: **return** T. // Nichts zu tun (x nicht vorhanden)

// Ab hier: $T \neq \square$, also $T = (T_1, (y, r), T_2)$.

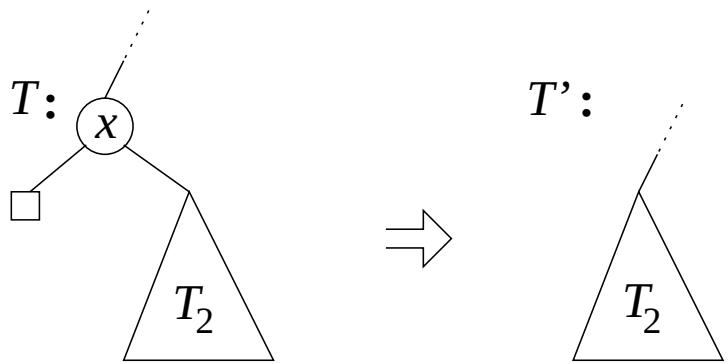
2. Fall: $x < y$: $T.\mathit{left} \leftarrow$ **delete**$(T.\mathit{left}, x)$; **return** T;

3. Fall: $y < x$: $T.\mathit{right} \leftarrow$ **delete**$(T.\mathit{right}, x)$; **return** T;

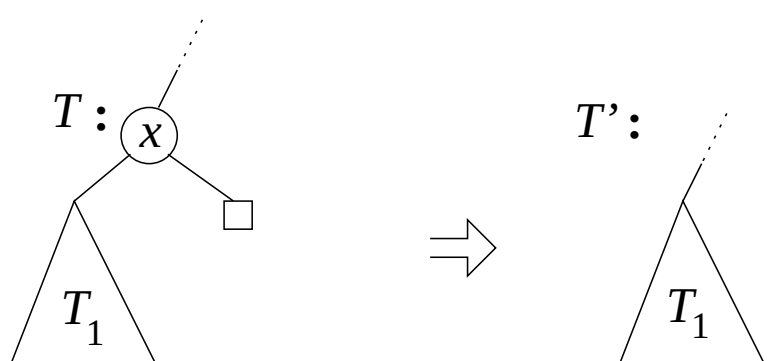
4. Fall: $x = y$: **Lösche Wurzel** von T (Unterfälle **4a–4c**, s. u.).

Lösche Wurzel von $T = (T_1, (x, r), T_2)$!

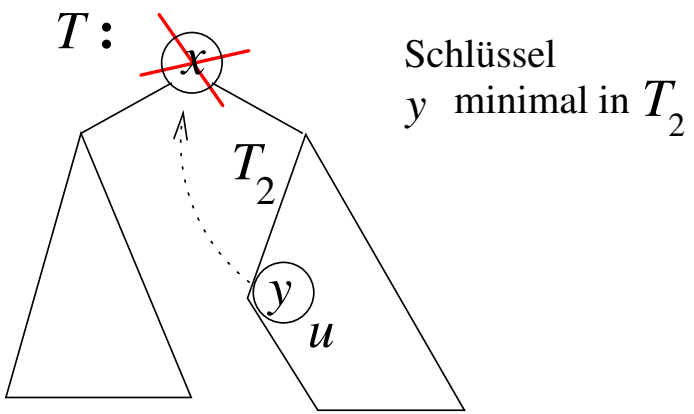
Fall 4a: $T.left = \square$: **return** $T.right$ (auch wenn $T.right = \square$ ist!)



Fall 4b: $T.right = \square$: **return** $T.left$



Fall 4c: $T.\mathit{left} \neq \square$ und $T.\mathit{right} \neq \square$.



Suche in T_2 Eintrag (y, r') mit minimalem Schlüssel y, in Knoten u.

(Dies ist der **Inorder-Nachfolger** der Wurzel von T.)

$T_2' := T_2$ nach Entfernen von Knoten u; // siehe unten **ExtractMin**

$u.\mathit{left} \leftarrow T.\mathit{left}$; $u.\mathit{right} \leftarrow T_2'$; **return** u.

(Zeiger/Referenzen umhängen, nicht Daten/Schlüssel kopieren!)

Extrahieren des Minimums, rekursiv

extractMin(T) // zieht aus einem (nichtleeren) BSB $T = (T_1, (z, r), T_2)$ den Knoten u mit dem kleinsten Schlüssel heraus.

Dies ist ebenfalls rekursiv zu realisieren.

Resultat: Kleinerer Baum T' und ein separater Baumknoten u (via Zeiger auf u).

Fall „leer“: $T.\mathit{left} = \square$. // Schlüssel z in der Wurzel ist minimal in T.

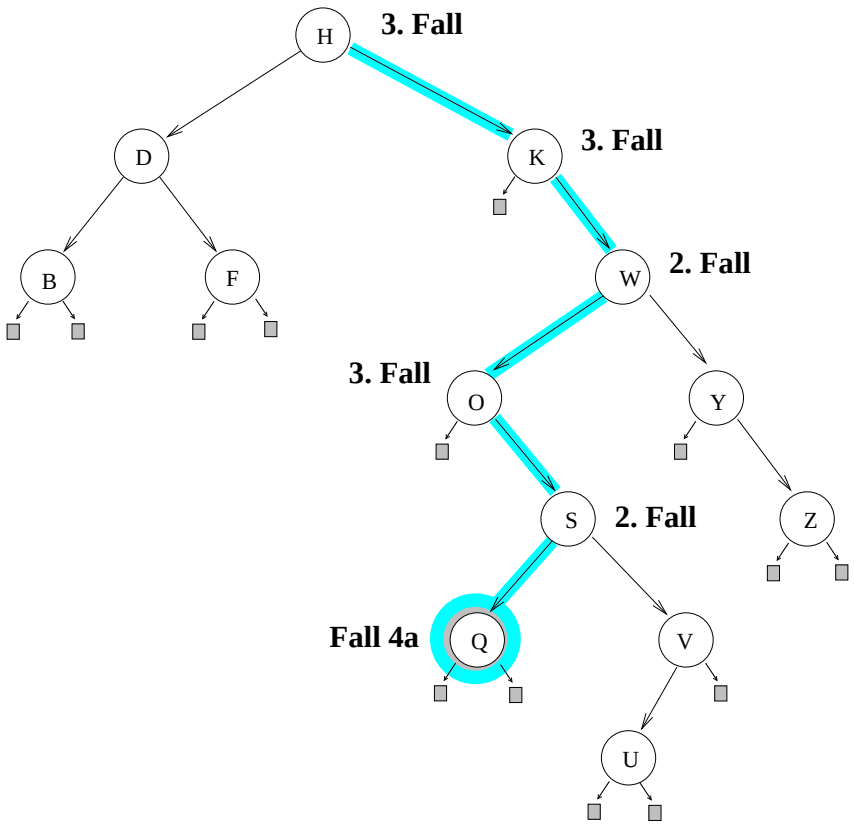
$T' \leftarrow T.\mathit{right}$; $T.\mathit{right} \leftarrow \square$; $u \leftarrow T$; **return**(T', u) // Knoten u war Wurzel.

Fall „nichtleer“: $T.\mathit{left} \neq \square$.

$(T', u) \leftarrow$ **extractMin**$(T.\mathit{left})$; $T.\mathit{left} \leftarrow T'$; **return**$(T,\ u)$.

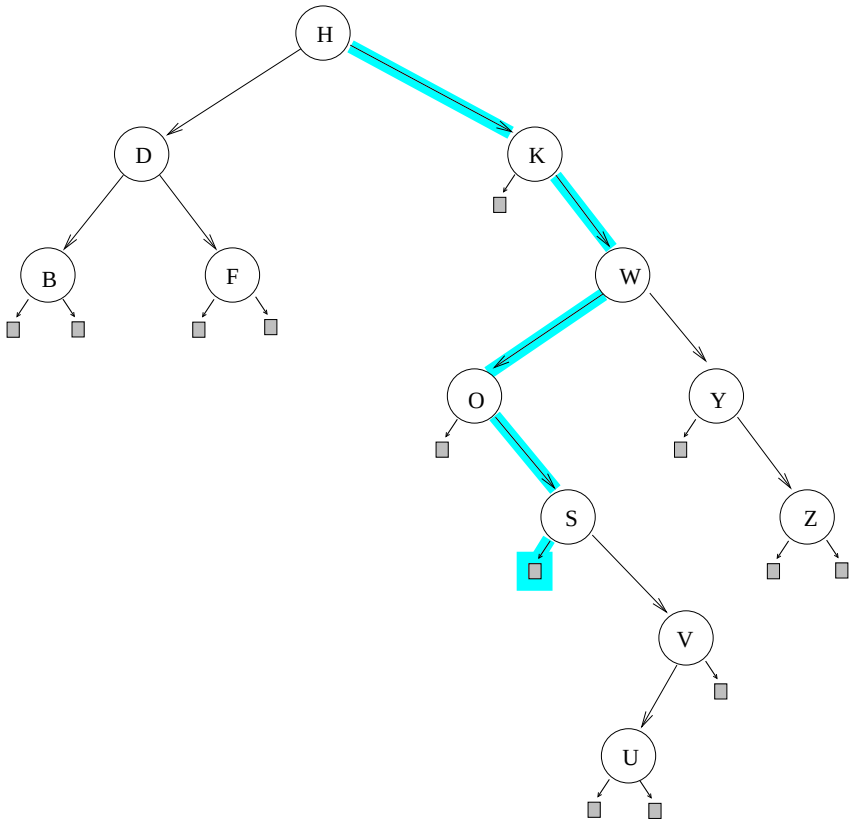
Anschaulich, iterativ: Starte in der Wurzel von T. Laufe den „Weg der linken Kinder“ hinab, bis ein Knoten u erreicht ist, der kein linkes Kind hat. Knoten u enthält den kleinsten Schlüssel in T. Lenke Zeiger auf u auf den rechten Unterbaum von u um, und gib den so modifizierten Baum und u zurück.

Beispiel:



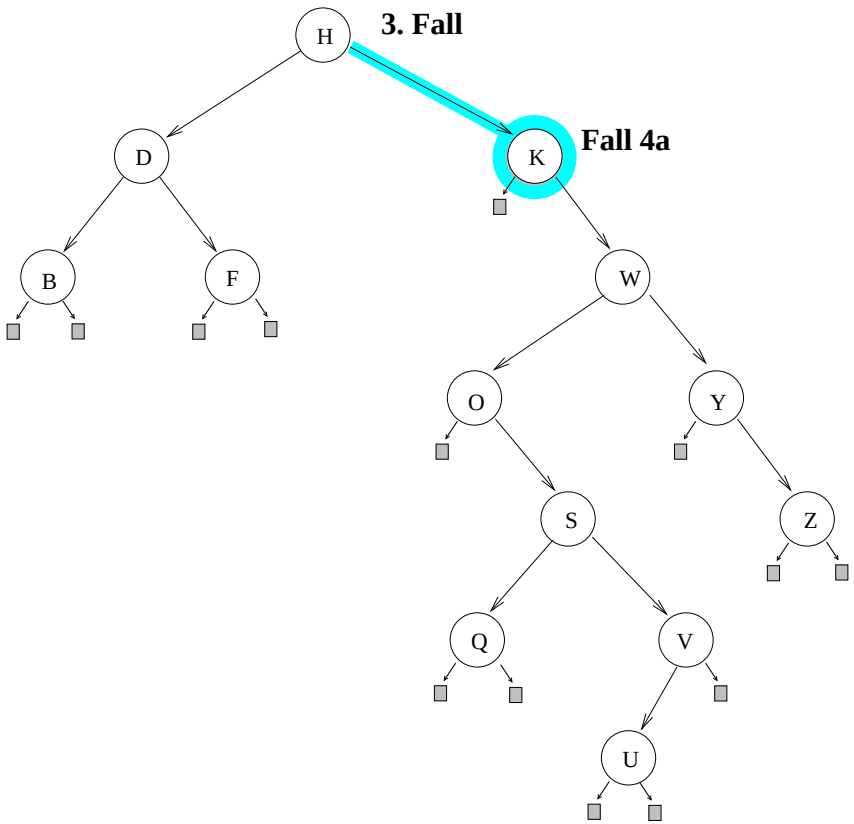
Lösche: Q.

Beispiel:



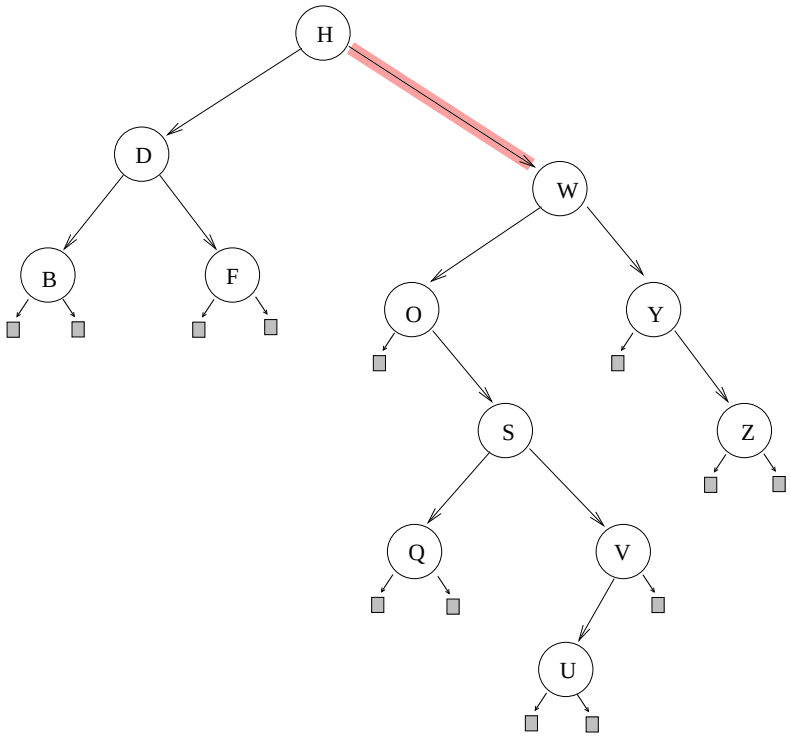
Lösche: Q.

Beispiel:



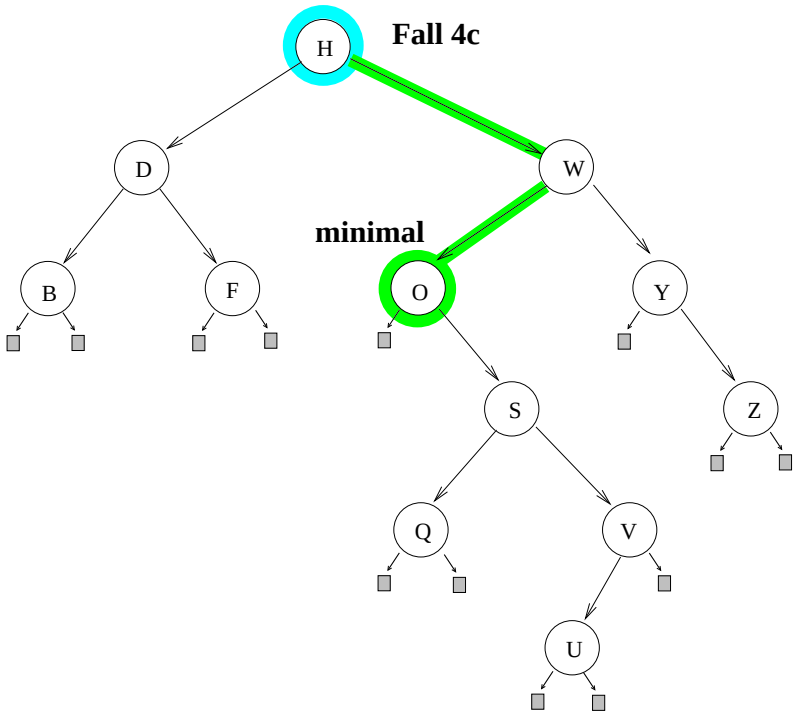
Lösche: K.

Beispiel:



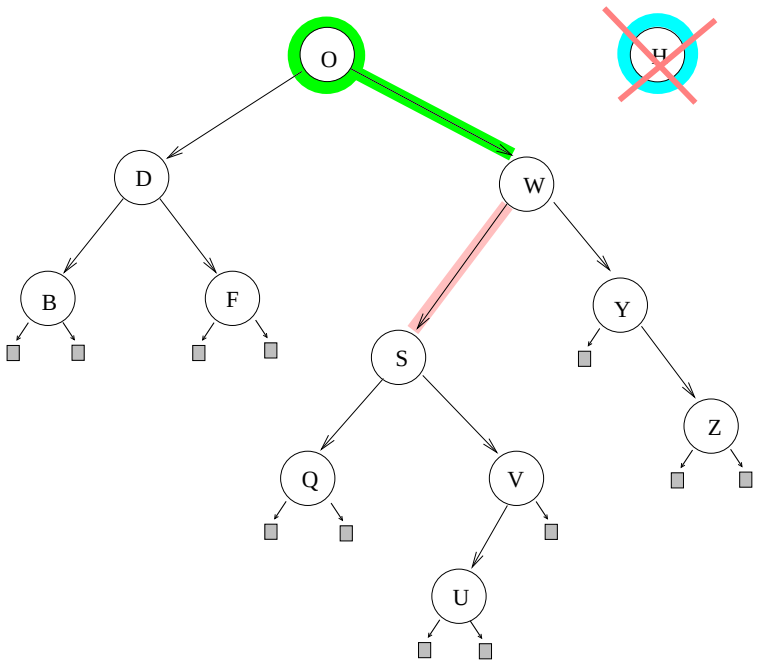
Lösche: K.

Beispiel:



Lösche: H.

Beispiel:



Lösche: H.

Intuitive, **iterative** Beschreibung des Löschvorgangs:
Suche Schlüssel x in T. Falls nicht gefunden, ist nichts zu tun. Falls x in Knoten v, drei Fälle:

a) v hat leeren linken Unterbaum.
„Biege Zeiger vom Vorgänger von v zu v auf den rechten Unterbaum von v um.“
(**Achtung!** Der Fall, dass v Blatt ist, ist hier inbegriffen.)

b) v hat leeren rechten Unterbaum.
„Biege Zeiger vom Vorgänger von v zu v auf den linken Unterbaum von v um.“

c) Beide Unterbäume von v sind nicht leer.
Suche **Inorder-Nachfolger** u von v, d. h. den Knoten mit kleinstem Schlüssel im rechten Unterbaum von v. (Gehe zum rechten Kind von v und dann „immer links“, bis Knoten u erreicht, der kein linkes Kind hat.) **Der linke Unterbaum von u ist immer leer!**
Lenke den Zeiger vom Vorgänger von u auf den rechten Unterbaum von u um, behalte u als einzelnen Knoten.
Ersetze in T den Knoten v durch u (Zeiger zu den Kindern kopieren, Zeiger auf v nach u umlenken).

Übung:
Löschung **iterativ** programmieren. *Hinweis*: Erst v mit $v.key = x$ finden, aber man benötigt auch den Vorgänger von v. Im Fall c): u finden, aber man benötigt auch den Vorgänger von u.

Korrektheit:

Behauptung: Aufruf **delete**(T, x) erzeugt einen binären Suchbaum für das modifizierte Wörterbuch *delete*(f_T, x).

Beweis: Induktion über den Aufbau von Binärbäumen.

Dabei muss man gesondert beweisen, und dann benutzen, dass die Prozedur **extractMin**(T), angewendet auf einen nichtleeren BSB, das verlangte Resultat liefert.

Zeitaufwand:

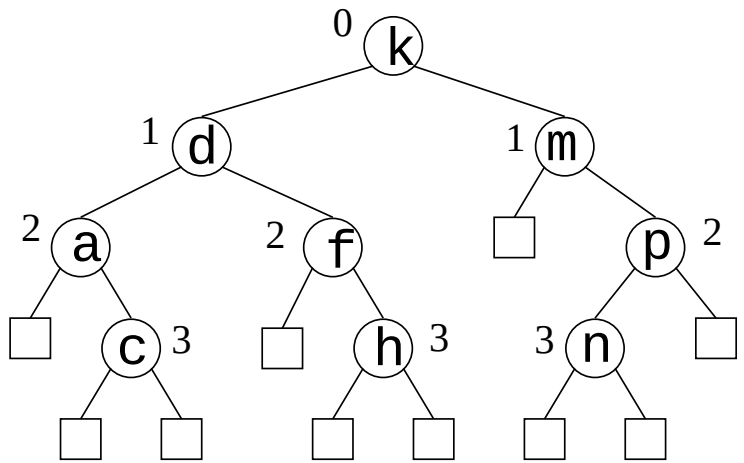
- x im Knoten v_x, Fälle 2/3: $O(d(v_x)) = O(d(T))$.
- x im Knoten v_x, Fall 4:
  $O(d(v_x^+)) = O(d(T))$, für Inorder-Nachfolger v_x^+ von v_x.
- x in T nicht vorhanden:
  $O(d(l_x)) = O(d(T))$, für externen Knoten l_x.

Vorlesungsvideo:

Erwartete mittlere Tiefe in zufälligen BSBs

4.2 Erwartete mittlere Tiefe in zufälligen BSBs

Schon gesehen:
Summe der Knotentiefen.



Totale innere Weglänge TIPL$(T) = \sum_{v \in V} d(v) (= 0 + 2 \cdot 1 + 3 \cdot 2 + 3 \cdot 3 = 17)$.

Mittlere innere Weglänge: $\frac{1}{n}\mathrm{TIPL}(T)$ (hier: $\frac{17}{9}$).

Totale Anzahl von durchlaufenen Knoten, wenn man zu jedem einmal hinläuft: $n + \mathrm{TIPL}(T)$.

Mittelwert $1 + \frac{1}{n}\mathrm{TIPL}(T)$: erwartete Kosten, wenn jeder Schlüssel mit Wahrscheinlichkeit $\frac{1}{n}$ gesucht wird.

Zufällig erzeugte binäre Suchbäume

T **„zufällig erzeugt“**: Gegeben: $S = \{x_1, \ldots, x_n\} \subseteq U$ mit $x_1 < \cdots < x_n$.

Starte mit leerem Baum, füge Schlüssel aus S nacheinander ein. Dabei:

Eingabereihenfolge der Schlüssel ist „rein zufällig“, d. h. für jede Permutation π hat die Reihenfolge $x_{\pi(1)}, \ldots, x_{\pi(n)}$ Wahrscheinlichkeit $1/n!$. Definiere:

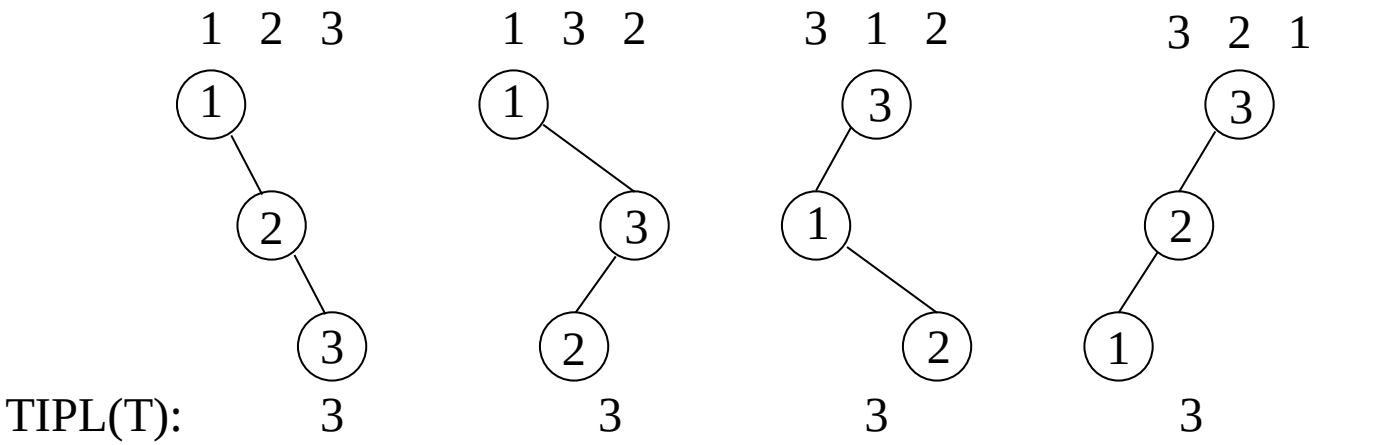
$$\boldsymbol{A(n)} := \mathbf{E}(\mathrm{TIPL}(T)) \quad \text{(Erwartungswert)}.$$

Dann ist der **erwartete** (über **Eingabereihenfolge zufällig**) **mittlere** (**über n Schlüssel $x \in S$ gemittelte**) Aufwand für **lookup**(x):

$$\mathbf{E}\left(O\left(1 + \frac{1}{n}\mathrm{TIPL}(T)\right)\right) = O\left(1 + \frac{1}{n}A(n)\right).$$

Was ist $A(n)$? $\quad A(0) = 0, \;\; A(1) = 0, \;\; A(2) = 1, \ldots$

$A(3) =$? – 6 Einfügereihenfolgen für $(x_1, x_2, x_3) = (1, 2, 3)$:

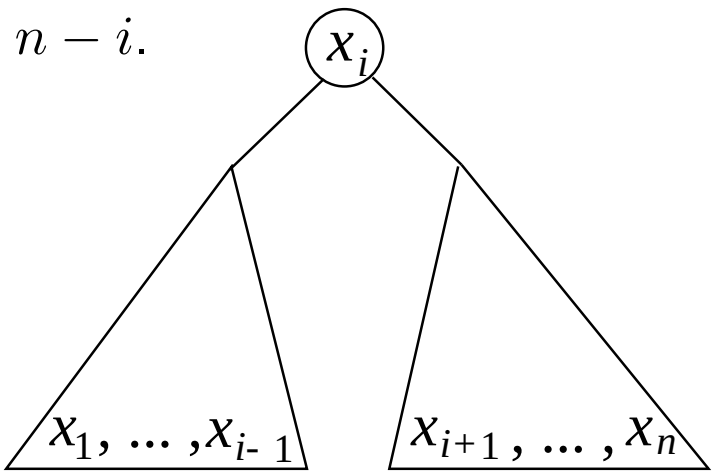


! Derselbe Baum !

$A(3) = \mathbf{E}(\mathrm{TIPL}(T)) = \frac{1}{6}(3 + 3 + 3 + 3 + 2 + 2) = \frac{8}{3}$.

Rekursionsformel

Wenn x_i der erste eingefügte Schlüssel ist, dann hat der linke Unterbaum von T genau $i-1$ Schlüssel, der rechte $n-i$.



Die Unterbäume sind selbst zufällig erzeugte binäre Suchbäume. Also:

$$\mathbf{E}(\text{TIPL}(T) \mid \textbf{falls } x_i \text{ erster}) = \ldots = ((i-1) + A(i-1)) + ((n-i) + A(n-i)).$$

Begründung: Der Weg in T zu jedem der $n-1$ Knoten in den Teilbäumen ist um 1 länger als der Weg im Teilbaum; daher die Summanden $(i-1)$, $(n-i)$.

Was wir hatten, etwas vereinfacht:

$$\mathbf{E}(\text{TIPL}(T) \mid \textbf{falls } x_i \text{ erster}) = n - 1 + A(i-1) + A(n-i).$$

Jeder der n Schlüssel $x_1, \ldots, x_n$ hat dieselbe Wahrscheinlichkeit $\frac{1}{n}$, als erster eingefügt zu werden und die Wurzel zu bilden. Mittelung liefert den E.-wert:

$$A(n) = \frac{1}{n} \cdot \sum_{1 \le i \le n} (n - 1 + A(i-1) + A(n-i)).$$

Das heißt:

$$A(n) = n - 1 + \frac{1}{n} \cdot \sum_{1 \le i \le n} (A(i-1) + A(n-i)) = n - 1 + \frac{2}{n} \cdot \sum_{0 \le j \le n-1} A(j).$$

(Begründung: $i - 1$ durchläuft $j = 0, 1, \ldots, n-1$; $n - i$ durchläuft $j = n-1, \ldots, 1, 0$.)

Wir haben also die folgende Rekurrenz**gleichung**:

$$A(0) = 0 \quad \text{und} \quad A(n) = n - 1 + \frac{2}{n} \cdot \sum_{1 \le j \le n-1} A(j) \text{ für } n \ge 1.$$

Damit können wir nacheinander Werte berechnen:

$$\begin{aligned}
A(1) &= 0 + \tfrac{2}{1} \cdot \textstyle\sum_{1 \le j \le 0} A(j) && = 0; \\
A(2) &= 1 + \tfrac{2}{2} \cdot (0) && = 1; \\
A(3) &= 2 + \tfrac{2}{3} \cdot (0 + 1) && = \tfrac{8}{3}; \\
A(4) &= 3 + \tfrac{2}{4} \cdot (0 + 1 + \tfrac{8}{3}) && = \tfrac{29}{6}; \\
A(5) &= 4 + \tfrac{2}{5} \cdot (0 + 1 + \tfrac{8}{3} + \tfrac{29}{6}) && = \tfrac{37}{5}; \text{ usw.}
\end{aligned}$$

Finden wir eventuell eine geschlossene Form?

$$A(n) = (n-1) + \frac{2}{n} \cdot \sum_{1 \le j \le n-1} A(j)$$

Multiplizieren mit n:

$$nA(n) = n(n-1) + 2 \cdot \sum_{1 \le j \le n-1} A(j)$$

Dasselbe für $n-1$:

$$(n-1)A(n-1) = (n-1)(n-2) + 2 \cdot \sum_{1 \le j \le n-2} A(j)$$

Subtrahieren (dieser Trick eliminiert die Summe!):

$$\begin{aligned} nA(n) - (n-1)A(n-1) &= n(n-1) - (n-1)(n-2) + 2A(n-1) \\ &= 2(n-1) + 2A(n-1). \end{aligned}$$

Wir haben: $nA(n) - (n-1)A(n-1) = 2A(n-1) + 2(n-1)$, d. h.

$$nA(n) - (n+1)A(n-1) = 2(n-1). \qquad \Big| : n(n+1)$$

$$\frac{A(n)}{n+1} - \frac{A(n-1)}{n} = \frac{2(n-1)}{n(n+1)}.$$

Mit der Abkürzung $Z(n) := A(n)/(n+1)$ gilt

$$Z(0) = 0 \quad \text{und} \quad Z(n) - Z(n-1) = \frac{2(n-1)}{n(n+1)}, \text{ für } n \geq 1.$$

Also, für $n \geq 0$:

$$Z(n) = \sum_{1 \leq j \leq n} \frac{2(j-1)}{j(j+1)} = \sum_{1 \leq j \leq n} \left(\frac{2}{j} - \frac{4}{j(j+1)} \right).$$

Wir haben:

$$Z(n) = 2 \sum_{1 \le j \le n} \frac{1}{j} - 4 \sum_{1 \le j \le n} \frac{1}{j(j+1)}.$$

Mit $\boldsymbol{H_n} := 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}$ **(n-te harmonische Zahl)** und $\frac{1}{1\cdot 2} + \frac{1}{2\cdot 3} + \cdots + \frac{1}{n\cdot(n+1)} = (1-\frac{1}{2})+(\frac{1}{2}-\frac{1}{3})+\cdots+(\frac{1}{n}-\frac{1}{n+1}) = 1-\frac{1}{n+1}$ ergibt das:

$$Z(n) = 2H_n - \frac{4n}{n+1}.$$

Mit $A(n) = Z(n) \cdot (n+1)$ erhalten wir die „geschlossene Form“

$$A(n) = 2(n+1)H_n - 4n.$$

Erwartete mittlere Knotentiefe (Einfügereihenfolge **zufällig**; **Mittelung** über $x_1, \ldots, x_n$):

$$\boxed{A(n)/n = 2H_n - 4 + O(\tfrac{H_n}{n}).}$$

Was ist H_n? Für jedes $i \geq 2$:

$$\frac{1}{i} < \int_{i-1}^{i} \frac{dt}{t} \, .$$

Für jedes $i \geq 1$:

$$\int_{i}^{i+1} \frac{dt}{t} < \frac{1}{i} \, .$$

Daraus:

$$H_n - 1 = \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} < \int_{1}^{n} \frac{dt}{t} = \ln n \leq 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-1} < H_n.$$

Also:

$$\boxed{\ln n < H_n < 1 + \ln n, \text{ für } n \geq 2.}$$

Genauer (ohne Bew.): $(H_n - \ln n) \nearrow \gamma$, für $n \to \infty$, wobei $\gamma = 0{,}57721\ldots$ („Euler-Konstante“).

Hatten schon:

$$\frac{1}{n}A(n) = 2H_n - 4 + O(\tfrac{H_n}{n}).$$

Eben gesehen: $\gamma = \lim_{n\to\infty}(H_n - \ln n) = \gamma \approx 0{,}57721.$

Satz 4.2.1

$$\boxed{\frac{1}{n}A(n) = 2\ln n - (4 - 2\gamma) + o(1)}$$

Wenn man Zweierlogarithmen bevorzugt:

$$\frac{1}{n}A(n) = (\mathbf{2\ln 2})\log n - 2{,}846\ldots + o(1), \text{ mit } 2\ln 2 = \mathbf{1{,}386\ldots}.$$

Satz 4.2.1 (Kurzform) In zufällig erzeugten binären Suchbäumen ist die **erwartete mittlere** Knotentiefe etwa $1{,}39\log n$.

Mitteilungen:

Erwartete **Baumtiefe** $d(T)$ für zufällig erzeugtes T ist ebenfalls $O(\log n)$, genauer: $\mathbf{E}(d(T)) \leq 3 \log n + O(1/n)$.

Leider ist sehr oft die **Einfügereihenfolge nicht zufällig**.

Ungünstig: Daten perfekt oder partiell sortiert. (Und das ist nicht ungewöhnlich.)

Zudem: Häufiges Löschen mit **extractMin** zerstört die Balance, mittlere Suchzeit wird $\Omega(\sqrt{n})$.

Abhilfe hierfür: Bei Löschungen von Knoten mit nichtleeren Unterbäumen **abwechselnd kleinsten Schlüssel aus rechtem Unterbaum und größten Schlüssel aus linkem Unterbaum** entnehmen.

Rest dieses Kapitels: **Balancierte** Suchbäume. (Erzwingen logarithmische Tiefe.)

Vorlesungsvideo:

AVL-Bäume (Einführung)

4.3 Balancierte binäre Suchbäume

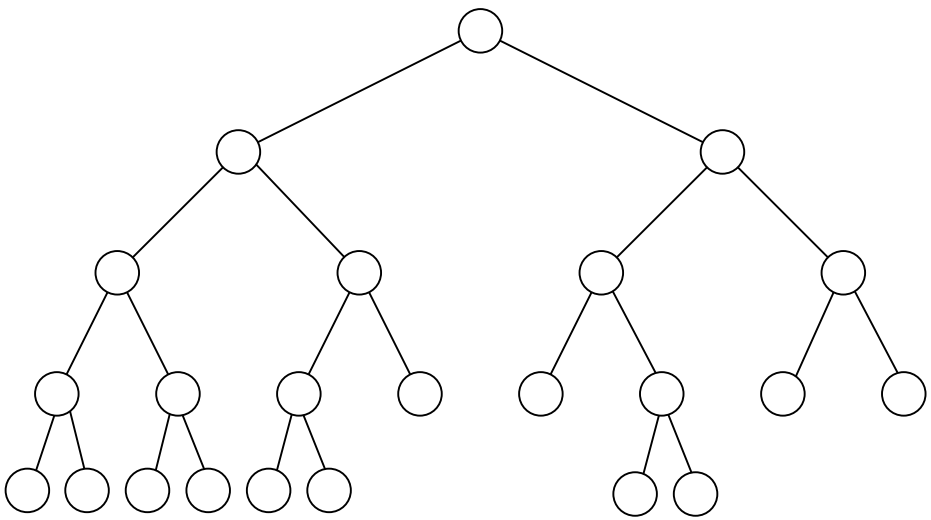
Idee:

- Lasse nur Bäume zu, die bestimmte **Strukturbedingungen** erfüllen.
- Strukturbedingungen **erzwingen geringe Tiefe** (z. B. $O(\log n)$ bei n Einträgen).
- **Implementiere** Wörterbuch-Operationen *insert* und *delete* so, dass

  (i) sie die **Strukturbedingungen erhalten** und

  (ii) ihre **Kosten O(Baumtiefe)** sind.

?? Attraktiv: Perfekte Balance.
D. h.: Alle Blätter auf zwei benachbarten Levels:



Tiefe d erfüllt $2^d - 1 < n \leq 2^{d+1} - 1$, also $d < \log(n+1) \leq d+1$, also

$d = \lceil \log(n+1) \rceil - 1$. (Bemerkung: $\lceil \log(n+1) \rceil = |\mathrm{bin}(n)|$, Länge der Binärdarst.)

Attraktiv?? Nein! Einfüge- und Lösch-Operationen zu teuer.

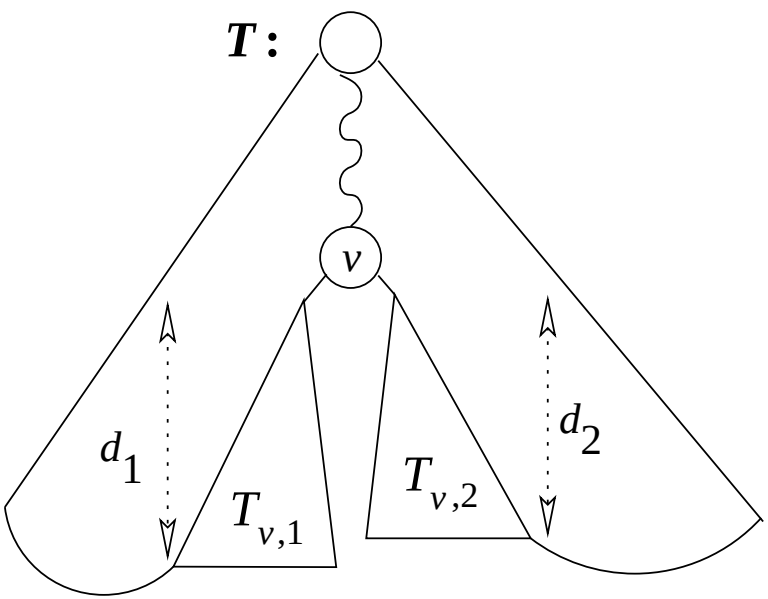
Häufig benutzte Strukturbedingungen:

- **AVL-Bäume** (behandeln wir ausführlich)
- **Rot-Schwarz-Bäume** ([Cormen et al.], [Sedgewick])
- **2-3-Bäume** (behandeln wir prinzipiell)
- **2-3-4-Bäume** ([Cormen et al.], [Sedgewick],[D./Mehlhorn/Sanders])
- **B-Bäume** ([Cormen et al.], Vorlesungen zu Datenbanktechnologie)

4.3.1 AVL-Bäume

Höhenbalancierte binäre Suchbäume

[G. M. **A**delson-**V**elski und J. M. **L**andis 1962]



$$|d_2 - d_1| \leq 1$$

Definition 4.3.1

Ein **Binärbaum** T heißt **höhenbalanciert**, falls in **jedem Knoten** v in T für den Teilbaum $T_v = (T_{v,1}, v, T_{v,2})$ mit Wurzel v gilt:

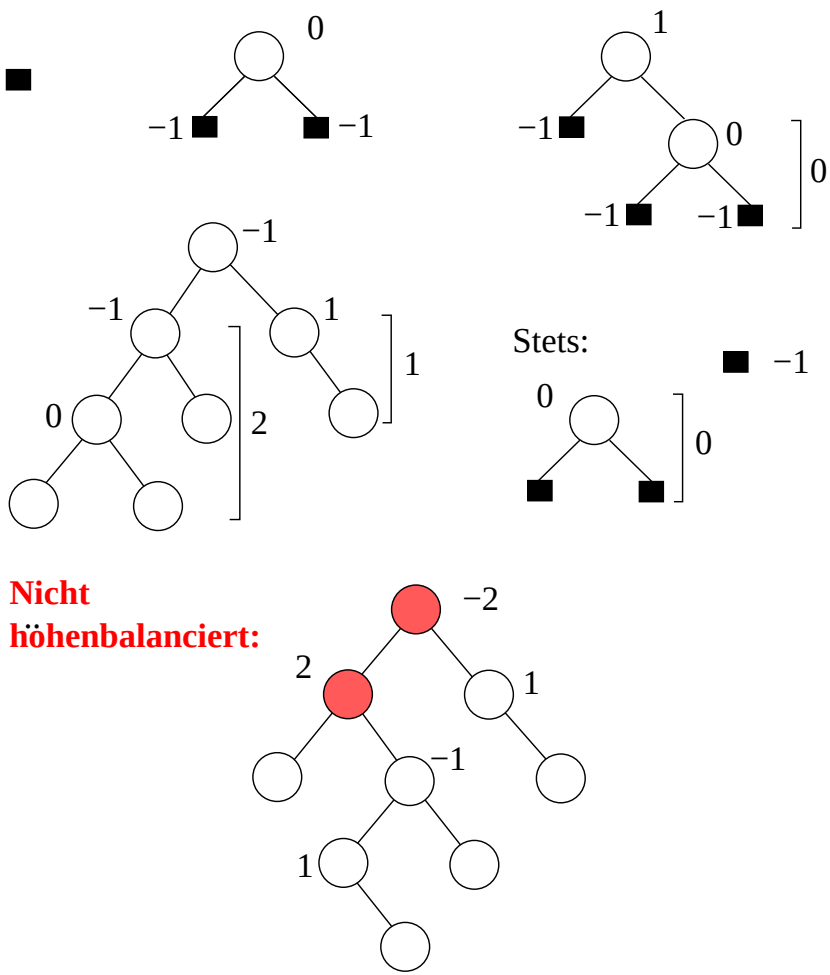
$$\underbrace{d(T_{v,2}) - d(T_{v,1})}_{\text{„Balancefaktor } bal \text{ in } v\text{“}} \in \{-1, 0, 1\} \; .$$

Äquivalent ist die folgende **rekursive** Charakterisierung:

(i) Der leere Baum $\square$ ist höhenbalancierter BB über U.

(ii) Sind T_1 und T_2 höhenbalancierte BB, $x \in U$, und ist
$bal = d(T_2) - d(T_1) \in \{-1, 0, 1\}$,
so ist (T_1, x, T_2) höhenbalancierter BB über U (mit **Balancefaktor** *bal*).

Beispiele:



Definition 4.3.2

Ein **AVL-Baum** ist ein **höhenbalancierter binärer Suchbaum**.

Für die **Implementierung** von AVL-Bäumen muss man in jedem inneren Knoten v seinen Balancefaktor **bal(v)** speichern.

Hoffnung: AVL-Bäume sind nicht allzu tief. Tatsächlich: „. . . logarithmisch tief“!

Satz 4.3.3

Ist T ein höhenbalancierter Baum mit $n \geq 1$ Knoten, so gilt

$$\boxed{d(T) \leq 1{,}4405 \cdot \log_2 n.}$$

D. h.: AVL-Bäume sind höchstens um den Faktor $1{,}4405$ tiefer als vollständig balancierte binäre Suchbäume mit derselben Knotenzahl.

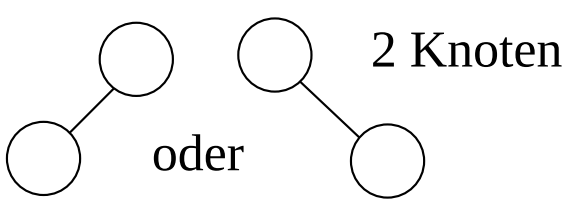
Beweisansatz: Wir zeigen, dass ein AVL-Baum mit Tiefe d exponentiell in d viele Knoten haben muss, nämlich mindestens Φ^d viele für eine Konstante $\Phi > 1$, mit $\log_\Phi 2 \leq 1{,}4405$.

Beweis von Satz 4.3.3: Für $d = 0, 1, 2, \ldots$ setze

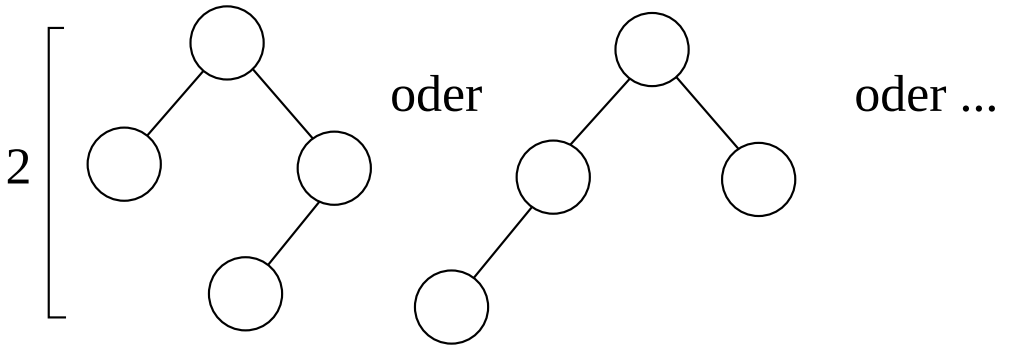
$N(d) :=$ **minimale Anzahl innerer Knoten** in einem AVL-Baum mit Tiefe d.

$N(0) = 1$ 0 1 Knoten

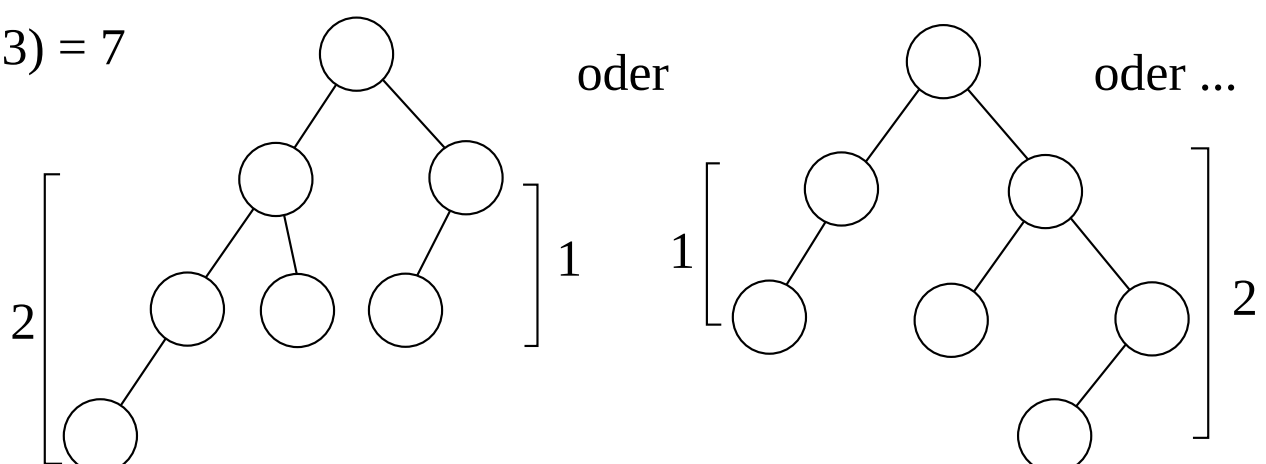
$N(1) = 2$ 1 2 Knoten oder



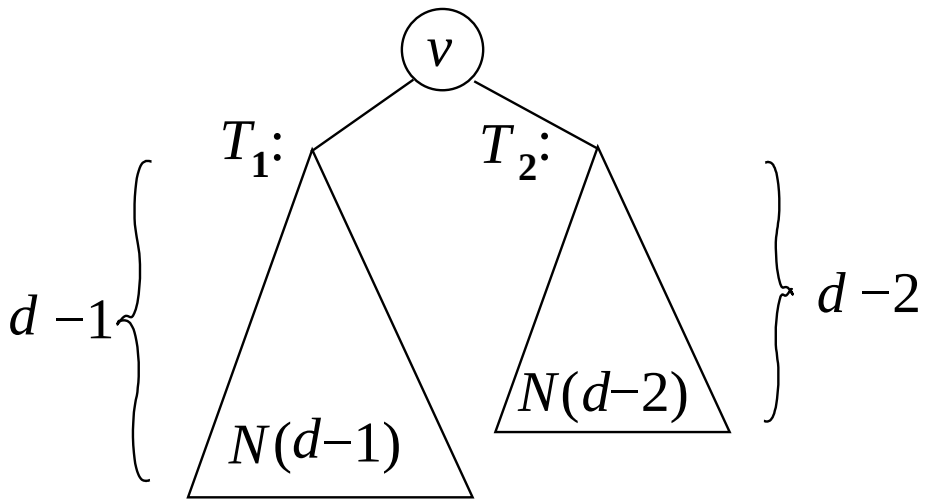
$N(2) = 4$



$N(3) = 7$



Behauptung 1: $N(d) = 1 + N(d-1) + N(d-2)$, für $d \geq 2$.



Beweis: Damit der höhenbalancierte Baum T der Tiefe d minimale Knotenzahl hat, muss ein Unterbaum von T Tiefe $d-1$ und der andere Tiefe $d-2$ haben. (Es ist klar, dass $N(d-1) > N(d-2)$ gilt.) Weiter müssen die Unterbäume minimale Knotenzahl für ihre jeweilige Tiefe haben. □

$N(d) = 1 + N(d-1) + N(d-2)$, für $d \geq 2$. – $N(d)$ für kleine d:

d	$N(d)$
0	1
1	2
2	4
3	7
4	12
5	20
6	33
7	54
8	88
9	143

Eindruck: exponentielles Wachstum.

Man kann zeigen: $N(d) = F_{d+3} - 1$, für die **Fibonacci-Zahlen** $F_0, F_1, F_2, \ldots$.

Behauptung 2:

$$N(d) \geq \Phi^d \text{ , für } d \geq 0,$$

wobei $\Phi = \frac{1}{2}(1+\sqrt{5}) \approx 1{,}618\ldots$ **(„goldener Schnitt“).**

$(*)$ Φ ist (eine, nämlich die positive) Lösung der Gleichung $x^2 = x + 1$.

Beweis von Beh. 2: Vollständige Induktion.

I.A.: Für $d = 0$: $N(0) = 1 = \Phi^0$; für $d = 1$: $N(1) = 2 > \Phi^1$.

Sei nun $d \geq 2$. **I.V.**: Behauptung gilt für $d' < d$. **I.-Schritt:**

$$N(d) \overset{\text{Beh. 1}}{=} 1 + N(d-1) + N(d-2)$$

$$\overset{\text{I.V.}}{\geq} \Phi^{d-1} + \Phi^{d-2} = \Phi^{d-2}(\Phi + 1) \overset{(*)}{=} \Phi^{d-2} \cdot \Phi^2 = \Phi^d.$$

□

Nun sei T ein höhenbalancierter Baum mit Höhe $d(T) \geq 0$ und n Knoten.

Nach Behauptung 2: $\Phi^{d(T)} \leq n$.

Logarithmieren liefert $d(T) \leq \log_\Phi n$, d. h.

$$d(T) \leq (\log_\Phi 2) \cdot \log_2 n.$$

Es gilt: $\log_\Phi 2 = (\ln 2)/(\ln \Phi) = 1{,}440420\ldots < 1{,}4405$.

Damit ist Satz 4.3.3 bewiesen. □

AVL-Bäume: höhenbalancierte binäre Suchbäume.
Wissen: haben logarithmische Tiefe.

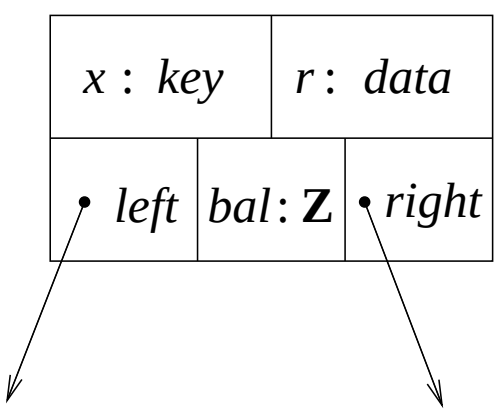
Müssen noch zeigen:

Man kann die Wörterbuchoperationen so implementieren,

dass die AVL-Eigenschaft erhalten bleibt,

und der Zeitbedarf proportional zur Tiefe ist.

Knotenformat:



x: `key`;
r: `data`;
bal: `integer`; // Legal: $\{-1, 0, 1\}$
left, *right*: `AVL_Tree`
// Zeiger auf Baumknoten

Vorlesungsvideo:

AVL Rotationen

4.3.2 Implementierung der Operationen bei AVL-Bäumen

AVL_empty: Erzeuge *NULL*-Zeiger. (Wie bei gewöhnlichem BSB.)

AVL_lookup: Wie bei gewöhnlichem BSB.

Brauchen nur: **AVL_insert**(T, x, r) und **AVL_delete**(T, x).

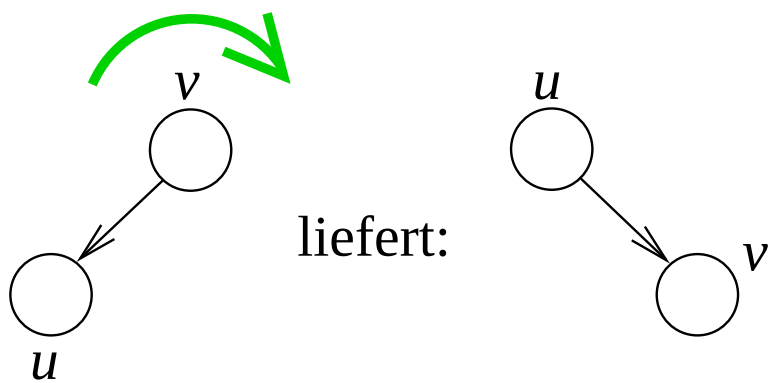
Grundansatz: Führe Operationen aus wie bei gewöhnlichem BSB.

Eventuell wird dadurch Bedingung „Höhenbalancierung“ verletzt.

„Reparatur“: **Rebalancierung**, erfolgt rekursiv.

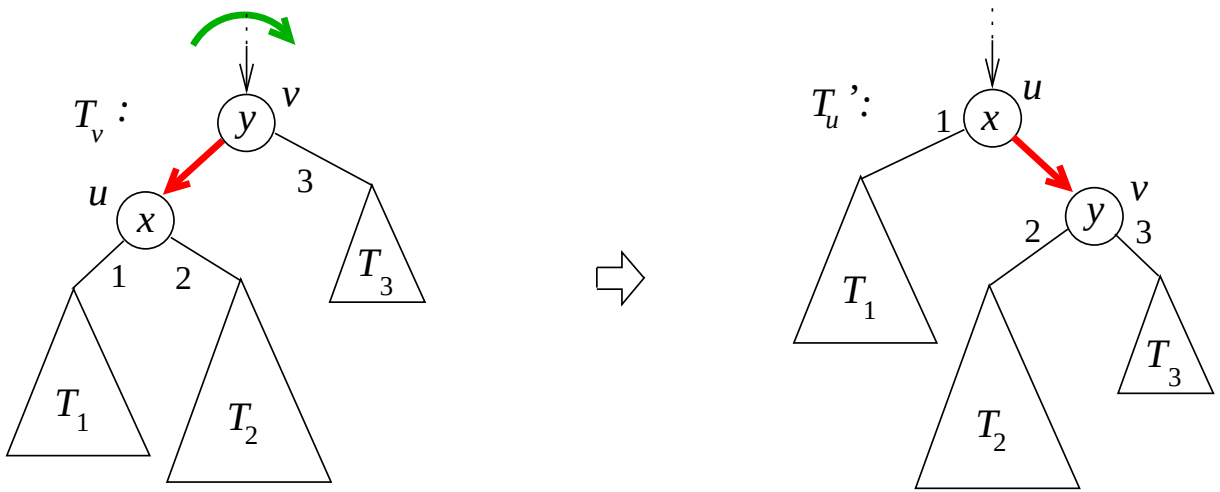
Rotationen: Hilfsoperationen für Rebalancierung

Rechtsrotation: „kippe 1 Kante nach rechts“:



Rotationen: Hilfsoperationen für Rebalancierung

Rechtsrotation: Mit Unterbäumen:



Beobachte: Knotenmengen von T_v und T'_u sind gleich $\wedge$ T'_u ist binärer Suchbaum.

Denn: Knotenmengen sind klar. Weil T_v binärer Suchbaum ist, gilt für v_1 in T_1, v_2 in T_2, v_3 in T_3: $\mathit{key}(v_1) < x < \mathit{key}(v_2) < y < \mathit{key}(v_3)$. Also ist T'_u auch BSB.

Beim Umbau ändert sich nur: rechtes Kind von u und linkes Kind von v, sowie der Zeiger auf Wurzelknoten (von v auf u umsetzen).

Rotationen: Hilfsoperationen für Rebalancierung

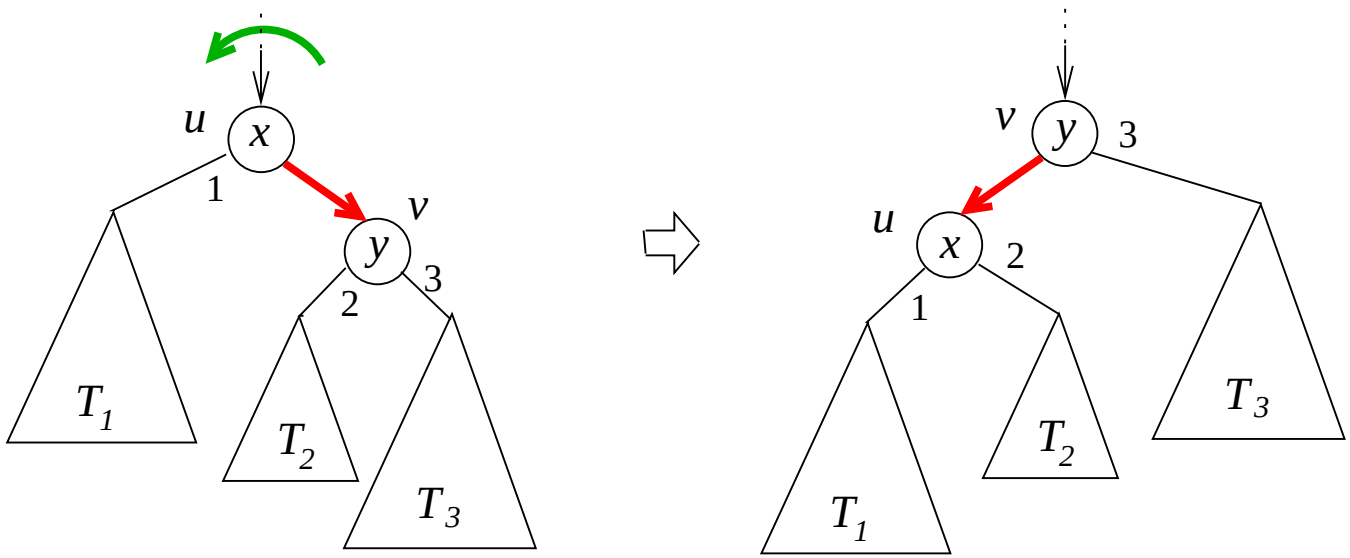
Rechtsrotation als Programm:

Funktion rotateR(v : AVL_Tree): AVL_Tree

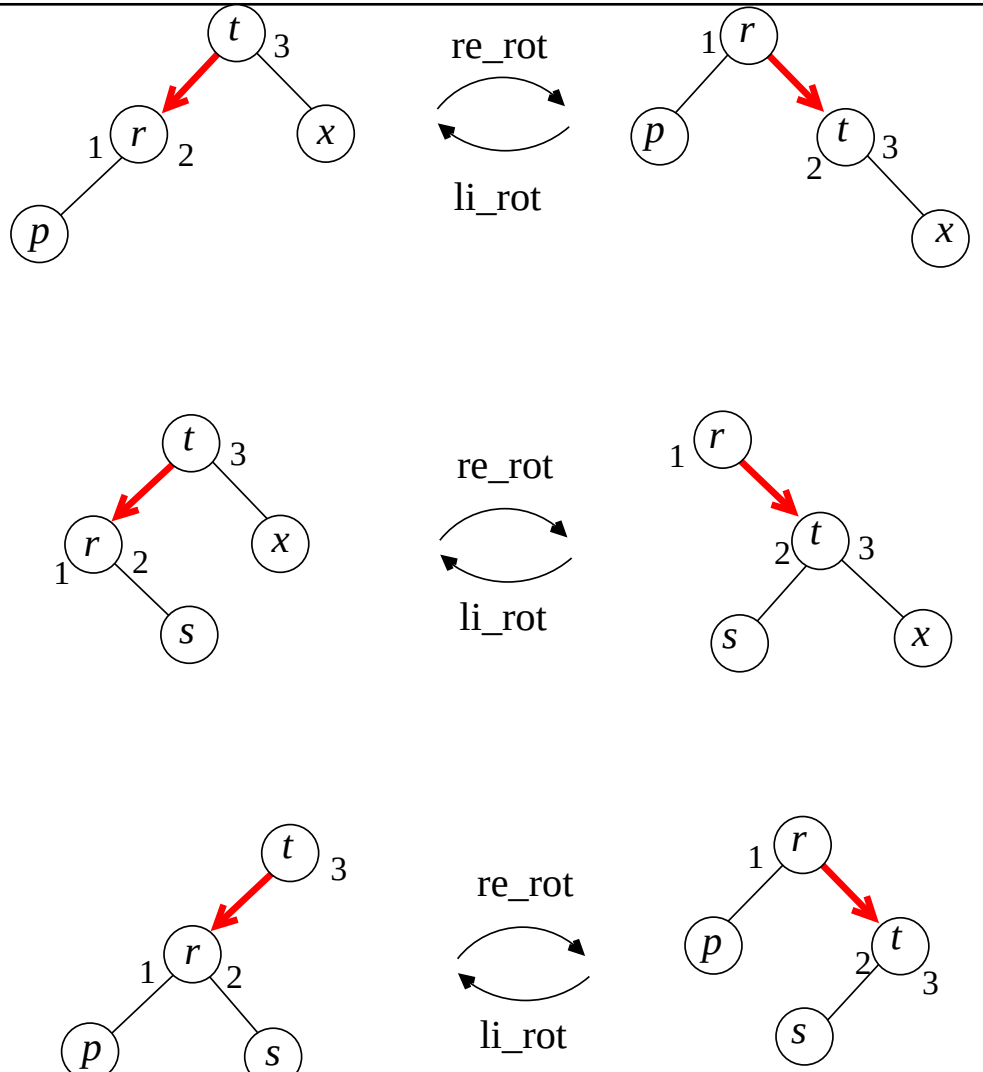
```
(1)       // v ≠ □, v.left ≠ □
(2)   u : AVL_Tree ;  // Hilfsvariable
(3)   u ← v.left ;
(4)   v.left ← u.right ;
(5)   u.right ← v ;
(6)   return u
(7)    // !! Balancefaktoren in u und v sind falsch!
```

Rotationen: Hilfsoperationen für Rebalancierung

Linksrotation: Umkehrung von Rechtsrotation, Implementierung analog.

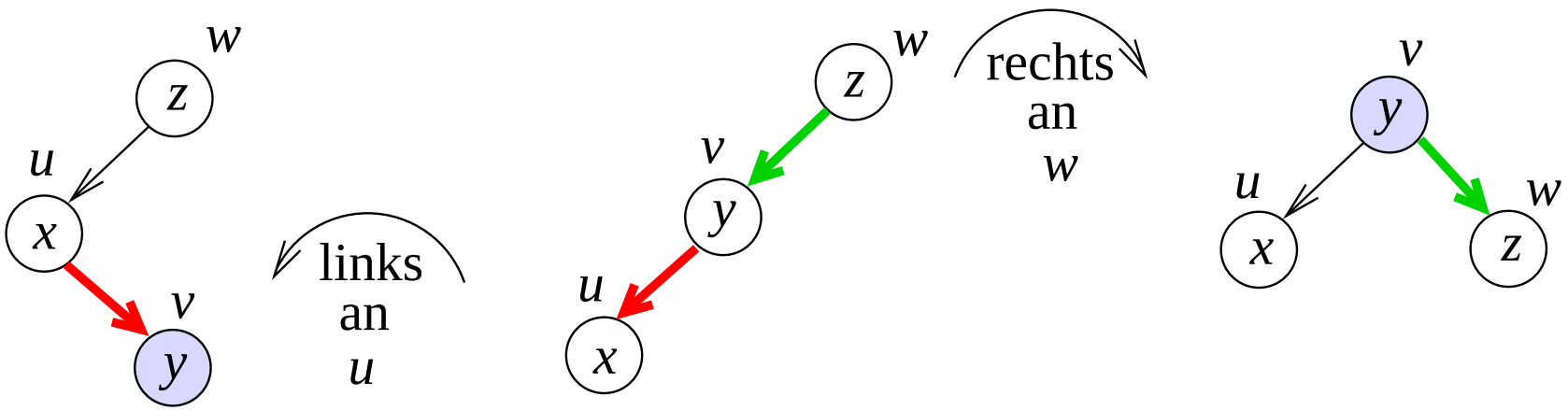


Hier und in den folgenden *Beispielen*:
Nummern bezeichnen Anschlussstellen für die Unterbäume.


t
3
r
1
2
x
p
re_rot
li_rot
r
1
p
t
2
3
x
t
3
r
1
2
x
s
re_rot
li_rot
r
1
t
2
3
s
x
t
3
r
1
2
p
s
re_rot
li_rot
r
1
p
t
2
3
s

Doppelrotationen: Links-Rechts, Rechts-Links

Beispiel: **Links-Rechts-Doppelrotation**

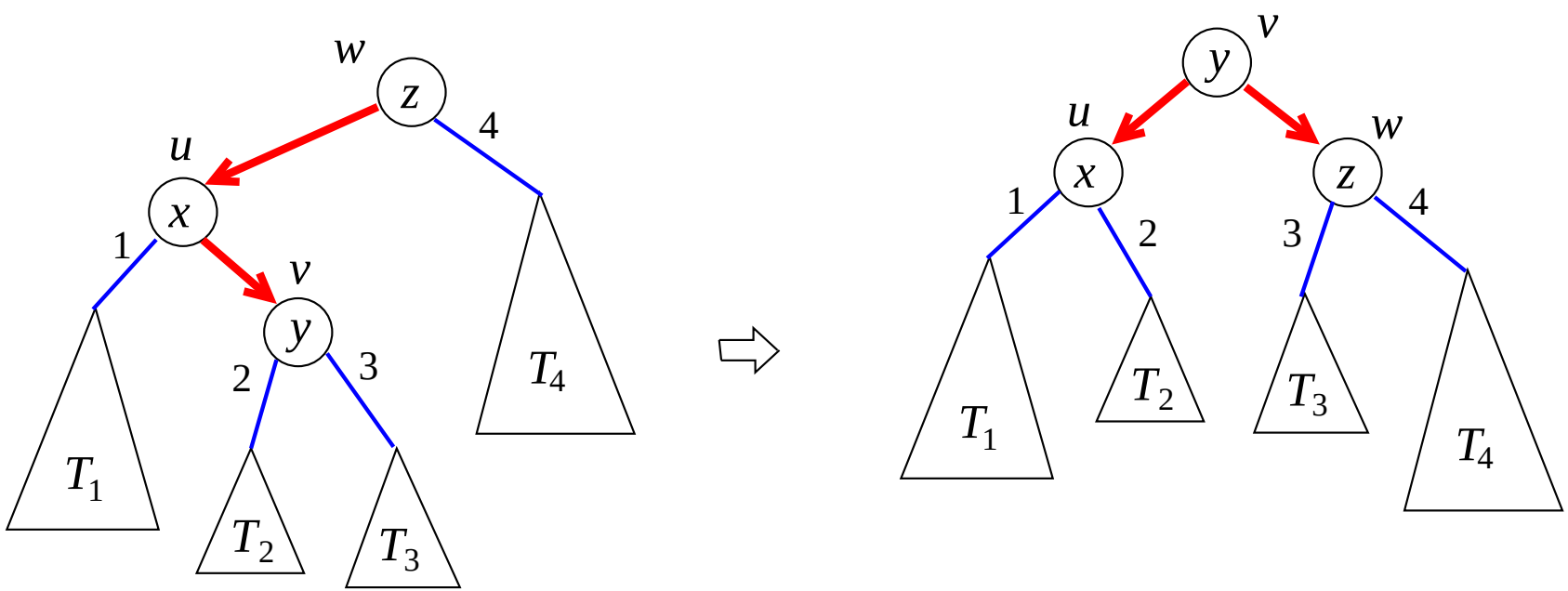


Anzuwenden auf Zick-Zack-Weg aus zwei Kanten, Form „**links-rechts**“.

Erst eine **Links**rotation, dann eine **Rechts**rotation.

Effekt: **Tiefster** Knoten v wird Wurzel.

Links-Rechts-Doppelrotation: Mit Unterbäumen



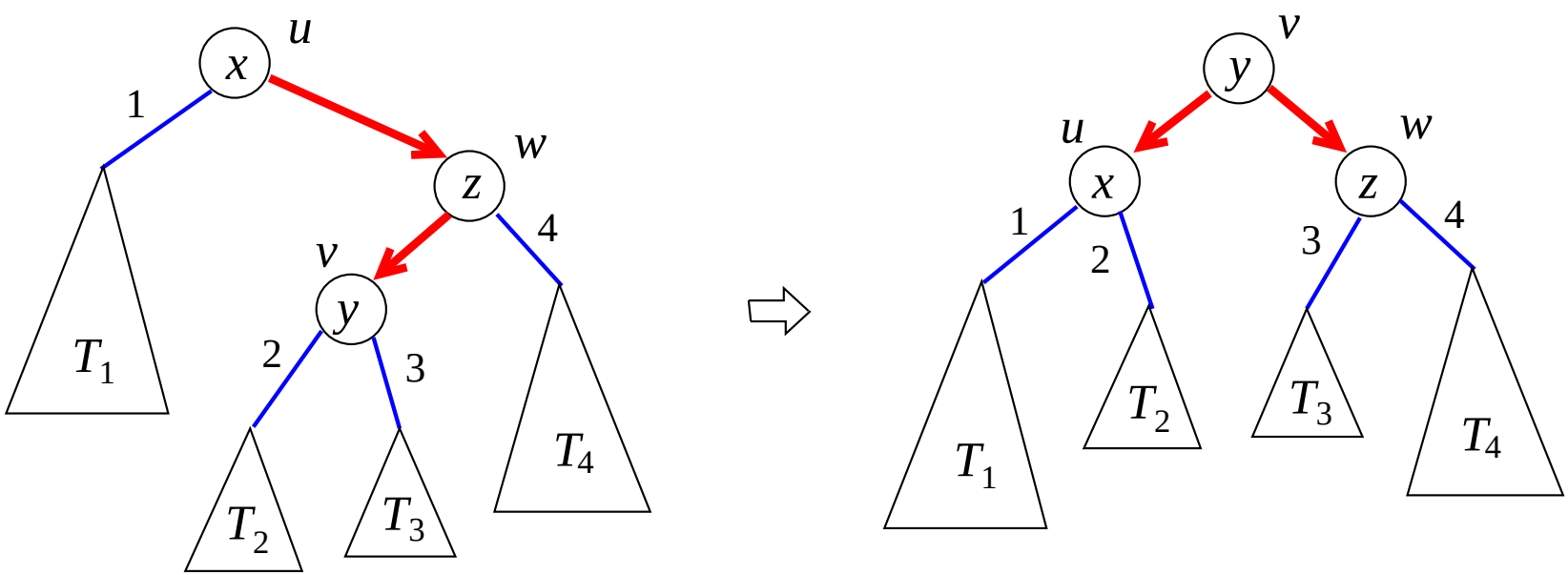
Gesamteffekt: Der **unterste** Knoten v des Zick-Zack-Wegs („links-rechts“) wandert nach oben, wird **Wurzel**; die beiden anderen Knoten u und w werden linkes und rechtes Kind. (Vier Teilbäume werden umgehängt, zwei bleiben gleich.)

LR-Doppelrotation als Programm

Funktion rotateLR(w : AVL_Tree) : AVL_Tree

```
(1)  // Eingabe: w ≠ □ mit  w.left ≠ □  und  w.left.right ≠ □
(2)  u, v : AVL_Tree ;
(3)  u ← w.left ;
(4)  v ← u.right ;
(5)  w.left ← v.right ;
(6)  u.right ← v.left ;
(7)  v.left ← u ;
(8)  v.right ← w ;
(9)  return v
(10)  // !! Balancefaktoren in u, v, w sind falsch!
⋮     ⋮
```

Rechts-Links-Doppelrotation:
Symmetrisch zu Links-Rechts-Doppelrotation



Implementierung: Analog zu Links-Rechts-Doppelrotation.

Vorlesungsvideo:

AVL_Insert

AVL_insert(T, x, r)

Anschauliche Beschreibung:

1. Füge ein wie bei gewöhnlichem BSB. Dies erzeugt neuen Knoten (außer Update).

2. Laufe dann **Einfügeweg von unten nach oben** ab, kontrolliere Balancebedingung. Wenn diese nirgendwo verletzt ist: fertig.

3. Sonst: $v :=$ **tiefster Knoten** auf Einfügeweg mit $bal(v) \in \{-2, 2\}$.

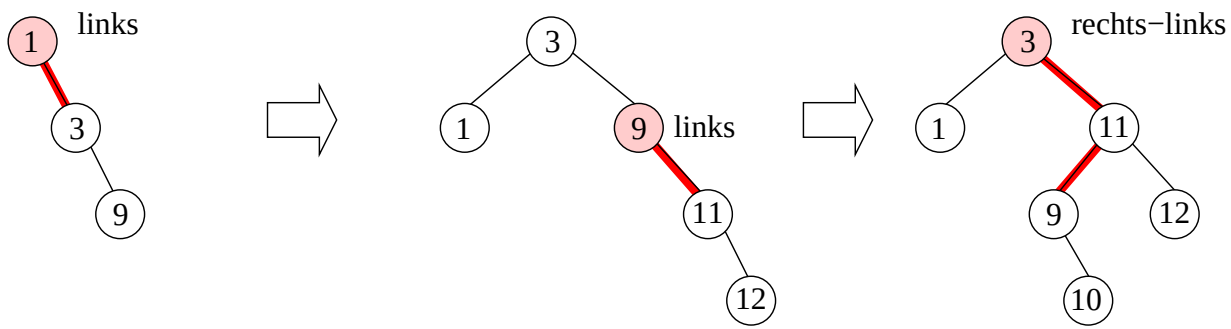
An v: Einfach- oder Doppelrotation. (Dann Rebalancierung beendet.)

3a) Wenn ein **äußerer** Teilbaum unter v (links-links oder rechts-rechts) zu tief ist:
Einfache Rotation hebt diesen Teilbaum ein Level höher.

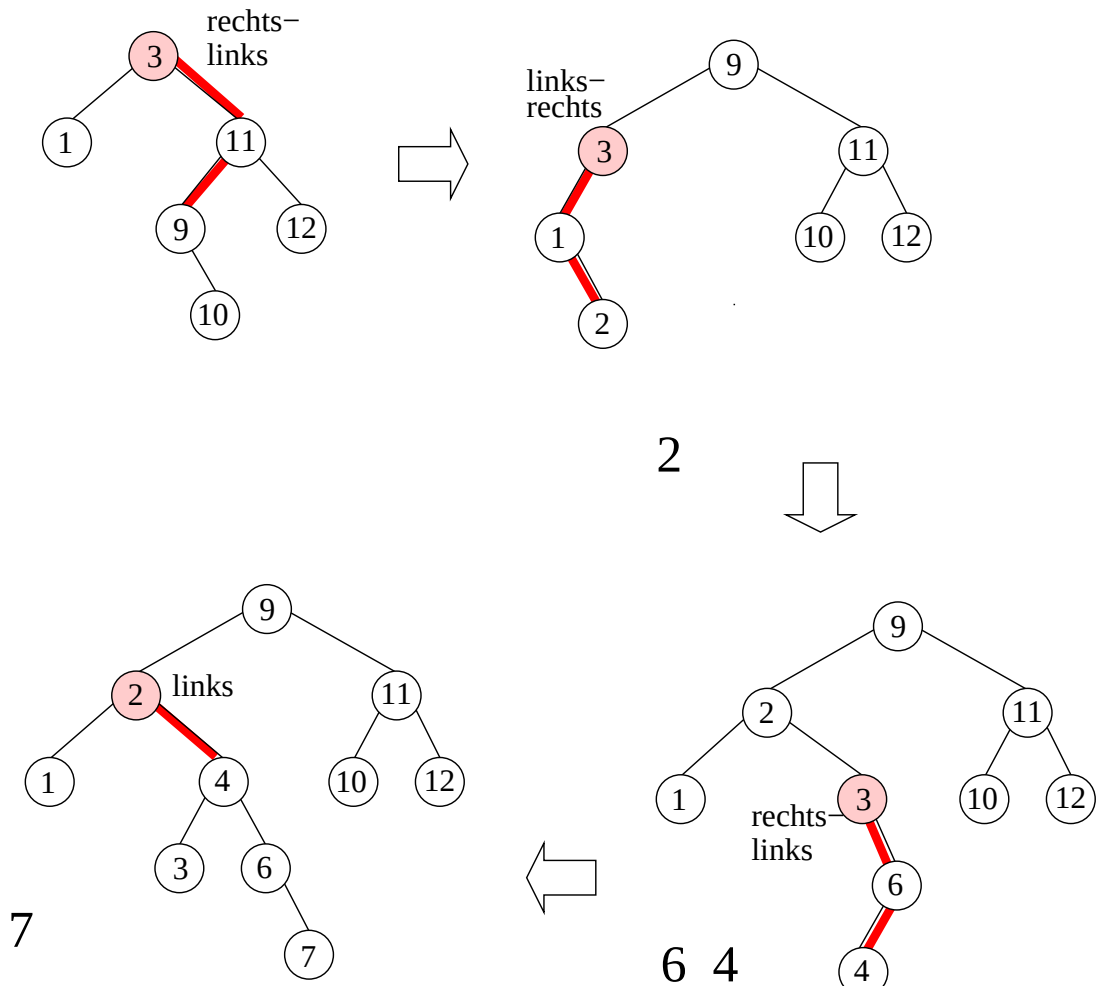
3b) Wenn ein **mittlerer** Teilbaum unter v (links-rechts oder rechts-links) zu tief ist:
Doppelrotation hebt den mittleren Teilbaum ein Level höher.

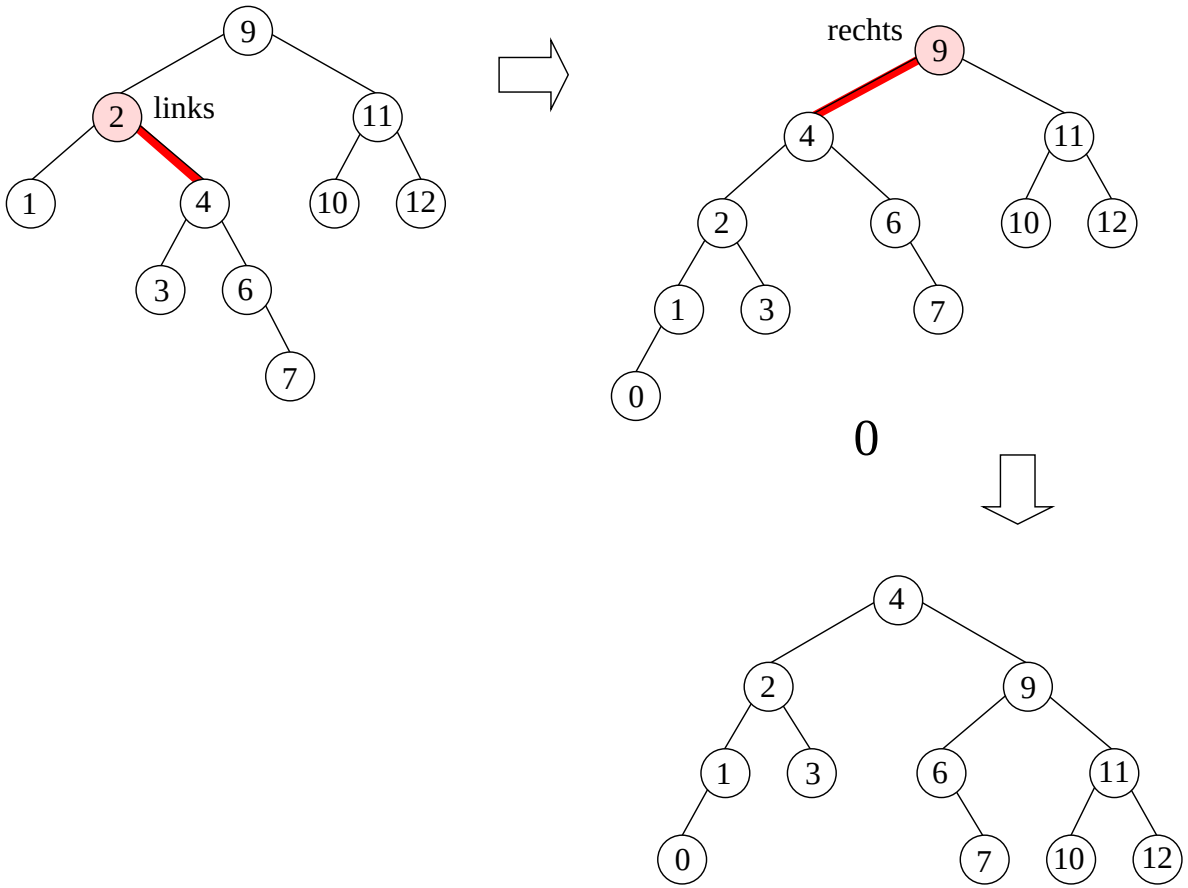
Zum Ausprobieren:
`https://www.cs.usfca.edu/~galles/visualization/AVLtree.html`

Beispiel: Einfügen von 1, 3, 9, 11, 12, 10, 2, 6, 4, 7, 0 in anfangs leeren AVL-Baum. Wir zeichnen nur Situationen, in denen **Rotationen stattfinden**. Der tiefste Knoten, an dem die Balancebedingung verletzt ist, ist **rot** gezeichnet.



1 3 9 11 12 10


rechts−
links
3
1
11
9
12
10
links−
rechts
9
3
11
1
10
12
2
2
9
2
links
11
1
4
10
12
3
6
7
7
9
2
11
1
3
10
12
rechts−
links
6
4
6 4


9
2
links
11
1
4
10
12
3
6
7
rechts
9
4
11
2
6
10
12
1
3
7
0
0
4
2
9
1
3
6
11
0
7
10
12

Etwas knifflig: Rekursive Programmierung des Ablaufs.

Programm kann nicht verwenden: „Globale Sicht“, Vergleichen von Tiefen, weder durch „Hinschauen“ noch rechnerisch (Tiefen von Teilbäumen werden nicht mitgeführt!).

Benutzt werden:

- Balancefaktoren `T.bal` in den Knoten,

- Flagbit „`deeper`“, das als Resultat eines rekursiven Aufrufs **AVL_insert**(T, x, r) mitteilt, ob der Baum dadurch **tiefer** geworden ist.

Wenn **AVL_insert**(T', x, r) für einen Unterbaum T' von T aufgerufen wurde, gibt es zur Rebalancierung an der Wurzel von T eine große Fallunterscheidung, gesteuert durch (alte) Balancefaktoren und die „deeper“-Meldung aus dem rekursivem Aufruf.

Wir entwickeln das Programm systematisch durch Erweitern der gewöhnlichen BSB-Einfügung.

Funktion AVL_insert(T,x,r): (AVL_Tree,boolean)

(1) // **Eingabe:** T: AVL_Tree, x: key, r: data

(2) // **Ausgabe:** T: AVL_Tree, deeper: boolean

(3) **1. Fall:** $T = \square$

(4) T: new AVL_Tree // Erzeuge neuen AVL-Baum-Knoten

(5) T.key $\leftarrow$ x ;

(6) T.data $\leftarrow$ r ;

(7) T.left $\leftarrow \square$;

(8) T.right $\leftarrow \square$;

(9) T.bal $\leftarrow 0$;

(10) **return** (T, true) .

(11) // Baumhöhe hat sich von -1 auf 0 erhöht.

(12) **2. Fall:** `T` $\neq \square$ **and** `T.key = x`.
(13) // Update-Situation!
(14) `T.data` $\leftarrow$ `r` ;
(15) **return** (`T`, *false*) .
(16) // Baumstruktur hat sich nicht verändert.

(17) **3. Fall:** `T` $\neq \square$ **and** `x < T.key`.
(18) (`T.left`, `left_deeper`) $\leftarrow$ **AVL_insert**(`T.left`, `x`, `r`) ;
(19) // Rekursives Einfügen in linken Unterbaum, liefert AVL-Baum und Flagbit
(20) (`T`, `deeper`) $\leftarrow$ **RebalanceInsLeft**(`T`, `left_deeper`) ;
(21) // Rebalancierung in der Wurzel von `T`, falls nötig, **s. unten**
(22) **return** (`T`, `deeper`) .

(23) **4. Fall:** $\texttt{T} \neq \square$ **and** $\texttt{T.key} < \texttt{x}$.

(24) $(\texttt{T.right}, \texttt{right_deeper}) \leftarrow$ **AVL_insert**$(\texttt{T.right}, \texttt{x}, \texttt{r})$;

(25) // Rekursives Einfügen in rechten Unterbaum, liefert AVL-Baum und Flagbit

(26) $(\texttt{T}, \texttt{deeper}) \leftarrow$ **RebalanceInsRight**$(\texttt{T}, \texttt{right_deeper})$;

(27) // Rebalancierung in der Wurzel von T, falls nötig,
symmetrisch zum **3. Fall**.

(28) **return** $(\texttt{T}, \texttt{deeper})$.

Zeile (18): **AVL_insert** wird rekursiv auf `T.left`, `x`, `r` angewendet.

Ergebnis: Unterbaum `T.left` geändert. Flagbit `left_deeper` bedeutet:

`left_deeper` = *false*: `T.left` hat gleiche Höhe wie vorher;
`left_deeper` = *true*: `T.left` ist um 1 Level tiefer geworden.

Wir beweisen die Korrektheit durch Induktion über rekursive Aufrufe. Daher können wir als **I.V.** verwenden: `T.left` ist AVL-Baum (mit korrekten Balancefaktoren).

Verbleibende **Probleme**:

- Ist die Balancebedingung in der Wurzel von `T` erfüllt?
- Der Balancefaktor `T.bal` in der Wurzel von `T` könnte falsch sein.

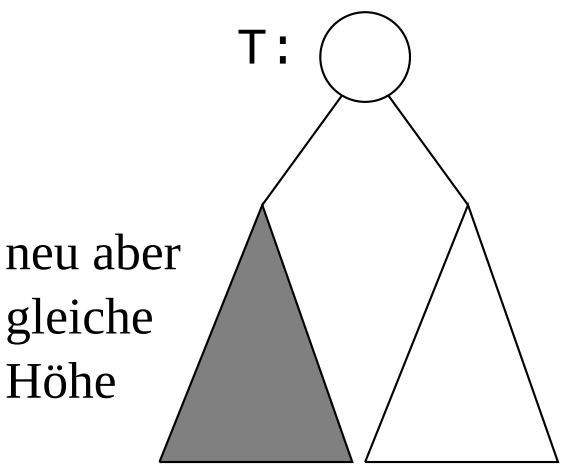
Zeile (20): **RebalanceInsLeft** prüft, ändert Struktur, korrigiert Balancefaktoren;
Ergebnis: Neuer AVL-Baum `T` und Flagbit `deeper`.

RebalanceInsLeft(T,left_deeper)

Fall RebIL-1

Aktion:

left_deeper = *false*



deeper ← *false* ;
// T.bal stimmt noch

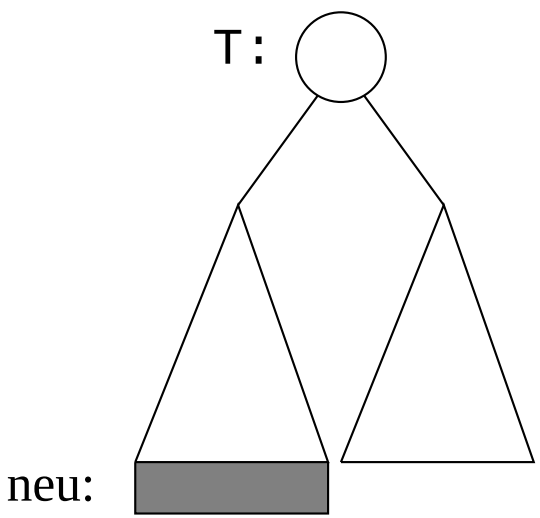
// Wenn dieser Fall einmal eingetreten ist,
// setzt er sich nach oben immer weiter fort.
// D. h.: Rebalancierung ist abgeschlossen.

RebalanceInsLeft(T,left_deeper)

Fall RebIL-2

Aktion:

left_deeper $=$ *true* $\wedge$
T.bal $=$ 0 // alter Wert!



deeper $\leftarrow$ *true* ;
T.bal $\leftarrow -1$;

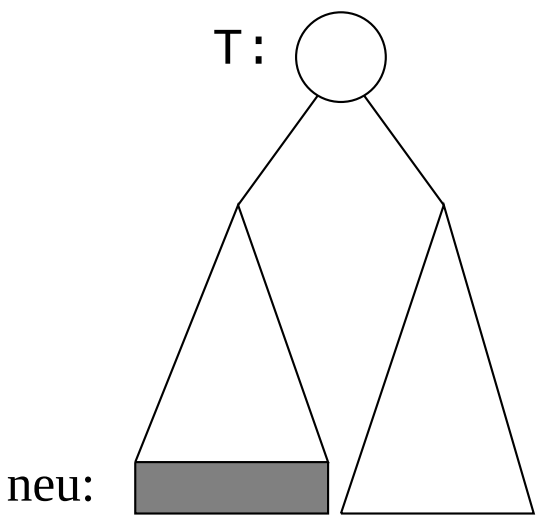
// Die Rebalancierungsaufgabe wird nach oben weitergereicht. Balancefaktor in Wurzel ist $\neq 0$.

RebalanceInsLeft(T,left_deeper)

Fall RebIL-3

Aktion:

left_deeper $=$ *true* $\wedge$
T.bal $=$ 1 // alter Wert!



deeper $\leftarrow$ *false* ;
T.bal $\leftarrow 0$;

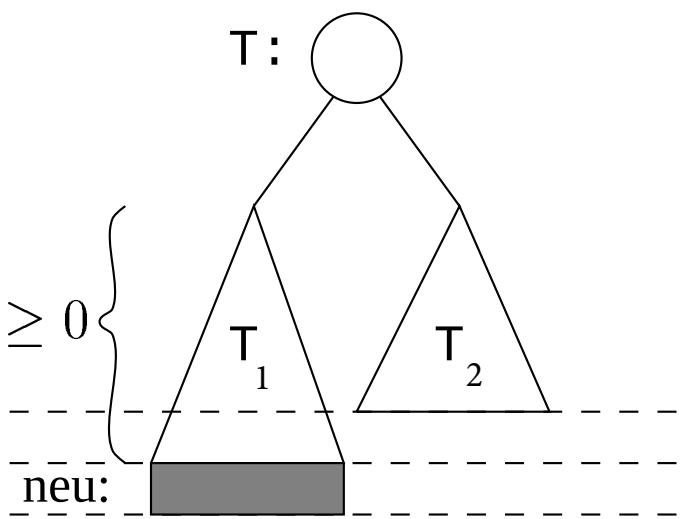
// Rebalancierung ist abgeschlossen, ohne Strukturänderung.

RebalanceInsLeft(T,left_deeper)

Fall RebIL-4

Aktion:

left_deeper $=$ *true* $\wedge$
T.bal $=$ -1 // alter Wert!



???

// Strukturänderung nötig.

Beobachte: Schon **vor** dem rekursiven Aufruf **AVL_insert**(`T.left`, `x`, `r`) muss der linke Unterbaum $T_1 =$ `T.left` von T nicht leer gewesen sein.

Schon gesehen: **RebIL-1**, **RebIL-3** (analog: **RebIR-1**, **RebIR-3**) liefern `deeper` = *false*.

Werden sehen: **RebIL-4** und **RebIR-4** liefern ebenfalls `deeper` = *false*.

$\Rightarrow$ beim rekursiven Aufruf der Rebalancierung für `T.left`
ist Fall **RebIL-2** oder **RebIR-2** eingetreten (Baum ist tiefer, durch neues Blatt).

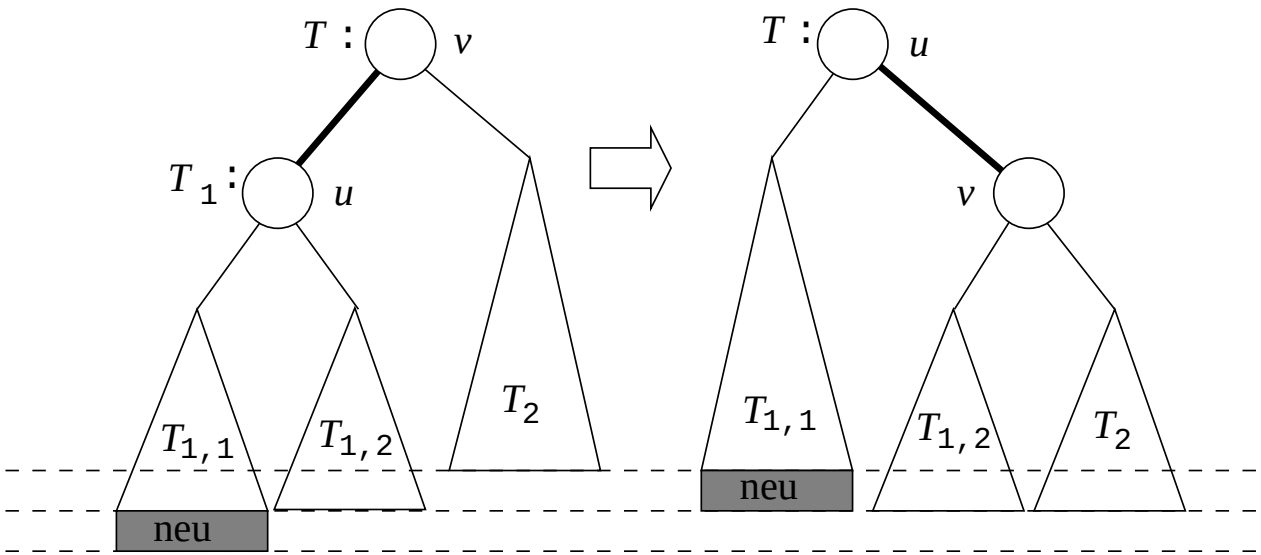
$\Rightarrow$ Balancefaktor `T.left.bal` in der Wurzel von T_1 ist -1 oder 1.

Unterfall RebIL-4.1

$\texttt{T.left.bal} = -1$

Aktion:

Rechtsrotation



$\texttt{T} \leftarrow \textbf{rotateR}(\texttt{T})$;

$\texttt{deeper} \leftarrow \mathit{false}$;

$\texttt{T.bal} \leftarrow 0$;

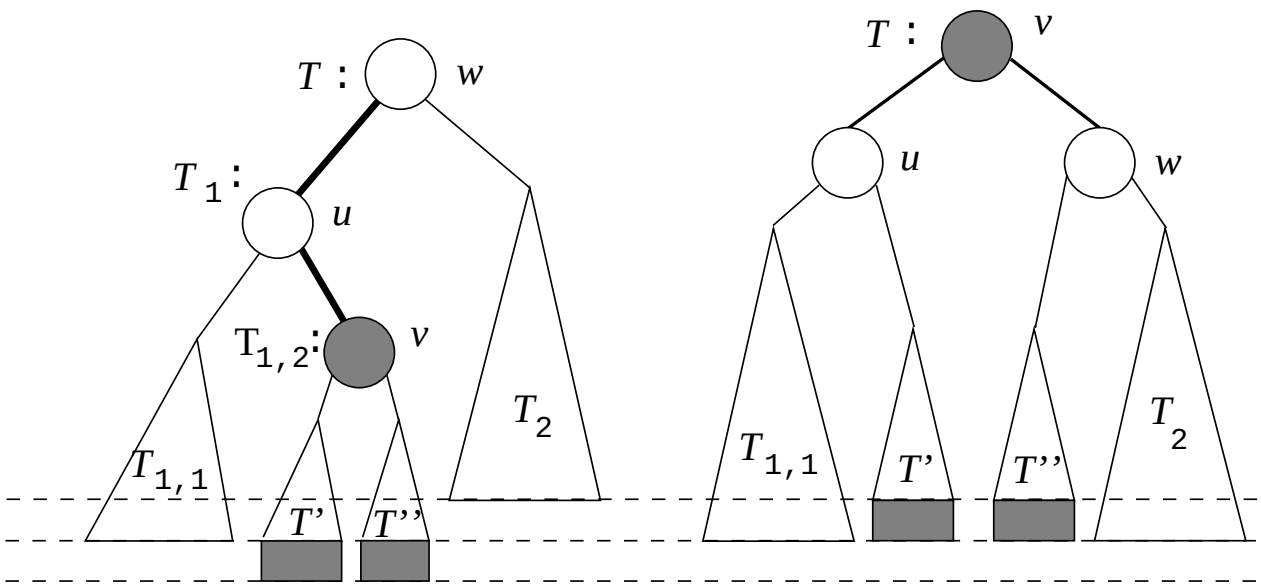
$\texttt{T.right.bal} \leftarrow 0$;

Unterfall RebIL-4.2

`T.left.bal` $= 1$

Aktion:

Links-Rechts-Doppelrotation



`T` $\leftarrow$ **rotateLR**(`T`) ;

`deeper` $\leftarrow$ *false* ;

Neue `bal`-Werte: s. Tabelle.

Neue `bal`-Werte im Fall RebIL-4.2:

(alt)	(neu)		
v`.bal`	u`.bal`	w`.bal`	v`.bal`
-1	0	1	0
0	0	0	0
1	-1	0	0

u ist Wurzel von `T.left`, w ist Wurzel von `T.right`, v ist die Wurzel von T.

Die Zahl v`.bal` (alt) stand schon **vor der LR-Rotation** in v`.bal`, gibt also den Balancefaktor des alten Unterbaums $T_{1,2} =$ `T.left.right` an.

Überlege: Kann es überhaupt passieren, dass der alte Baum $T_{1,2}$ Balancefaktor 0 hatte?

Antwort: **Ja!** Im rekursiven Aufruf **AVL_insert**`(T.left,x,r)` bestand `T.left` nur aus einem Knoten, $T_{1,2}$ ist der neu eingefügte Knoten; T', T'', $T_{1,1}$ und T_2 sind leer, haben also alle Tiefe -1.

Proposition 4.3.4

Die rekursive Prozedur **AVL_insert**(T, x, r) führt die Wörterbuchoperation *insert* korrekt durch, d. h.: Aus T entsteht ein AVL-Baum für *insert*(f_T, x, r).

Die Prozedur hat Laufzeit $O(\log n)$ und führt höchstens eine Einfach- oder Doppelrotation durch.

Man überlege (an Beispielen), wie die beschriebene rekursive Realisierung von **AVL_insert** die intuitive Beschreibung von Folie 60 umsetzt.

4.3.5 Folgerung (aus Algorithmus **AVL_insert**):

Für jedes $n \geq 0$ gibt es einen höhenbalancierten Baum mit n Knoten.

Beweis: Man fügt $1, \ldots, n$ (in beliebiger Reihenfolge) mit **AVL_insert** in einen anfangs leeren AVL-Baum ein. **Gute Übung:** Führe dies für $n = 16$ (z. B.) von Hand aus.

Bemerkung: Wenn T ein beliebiger höhenbalancierter Baum mit n Knoten ist, dann gibt es eine Einfügereihenfolge für Schlüssel $1, \ldots, n$, die genau diesen Baum erzeugt – sogar ohne jede Rotation **(Übung)**.

Vorlesungsvideo:

AVL_Delete

AVL_delete(T, x)

Anschauliche Beschreibung:

1. Suche Knoten mit x und lösche ihn wie bei gewöhnlichem BSB. Dadurch wird ein Knoten u mit $T_u.\mathit{left} = \square$ oder $T_u.\mathit{right} = \square$ entfernt. Sei w der Vorgängerknoten von u. Einer der Unterbäume von w ist nun „flacher“ als vorher.

2. Laufe Weg von w zur Wurzel **von unten nach oben**, kontrolliere an jedem Knoten v dieses Weges die Balancebedingung.

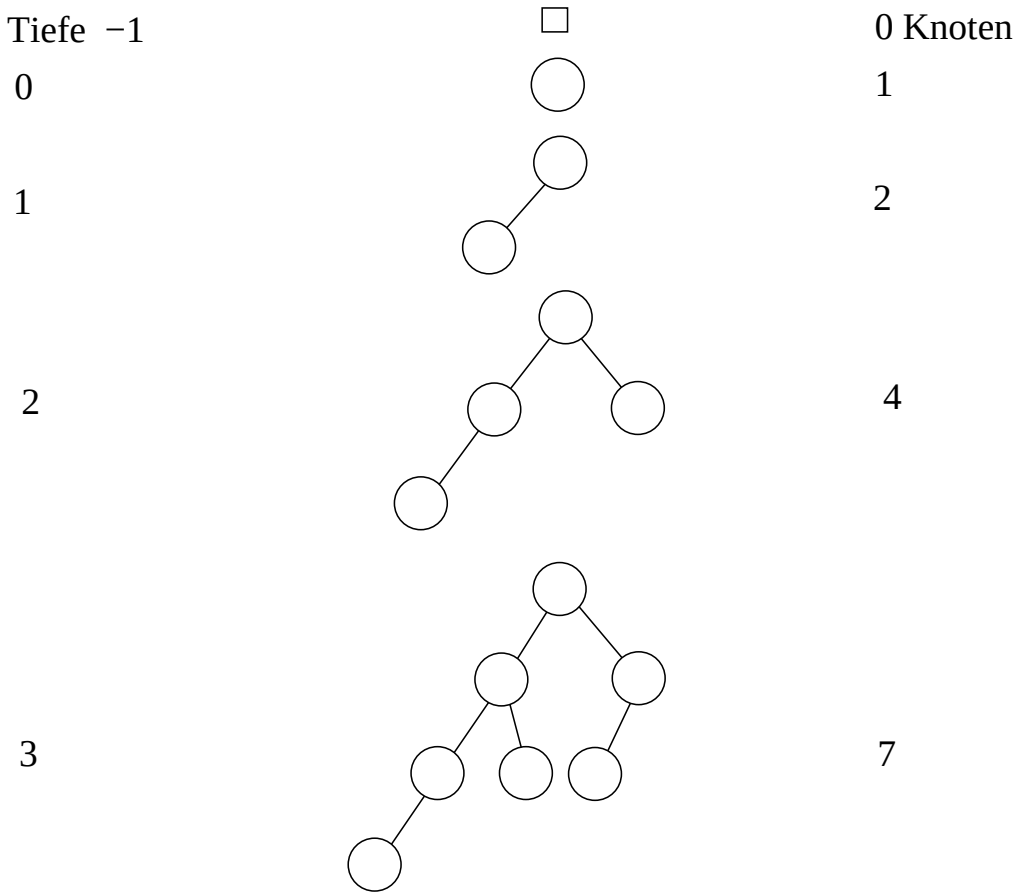
3. Wenn an v die Balancebedingung verletzt ist, d. h. $bal(v) \in \{-2, 2\}$, führe Einfach- oder Doppelrotation aus. **Aber welche?**

3a) Wenn ein **äußerer** Teilbaum unter v (links-links oder rechts-rechts) der tiefste ist:
Einfache Rotation, um diesen Teilbaum ein Level höher zu heben.
(Möglich ist: Der Geschwister-Teilbaum ist genauso tief. Dann ist die Rebalancierung beendet.)

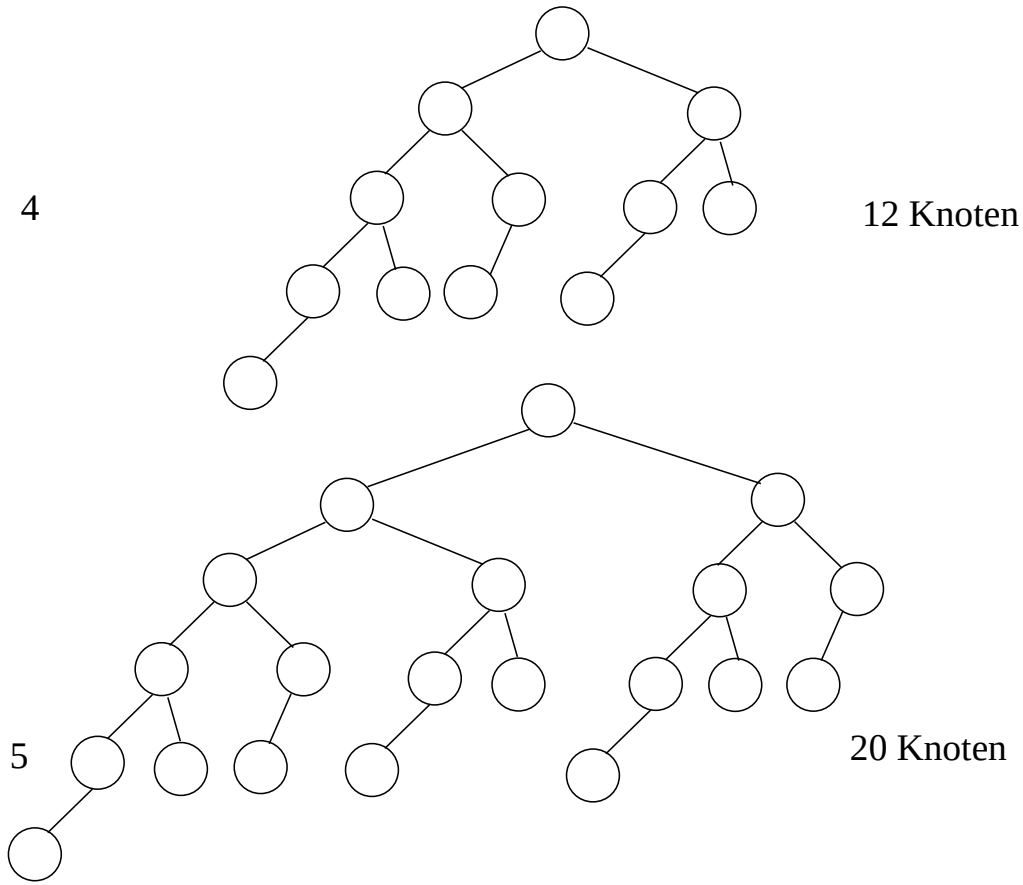
3b) Wenn ein **mittlerer** Teilbaum unter v (links-rechts oder rechts-links) der (strikt) tiefste ist:
Die entsprechende **Doppelrotation** hebt diesen mittleren Teilbaum ein Level höher.

Zum Ausprobieren: `https://www.cs.usfca.edu/~galles/visualization/AVLtree.html`

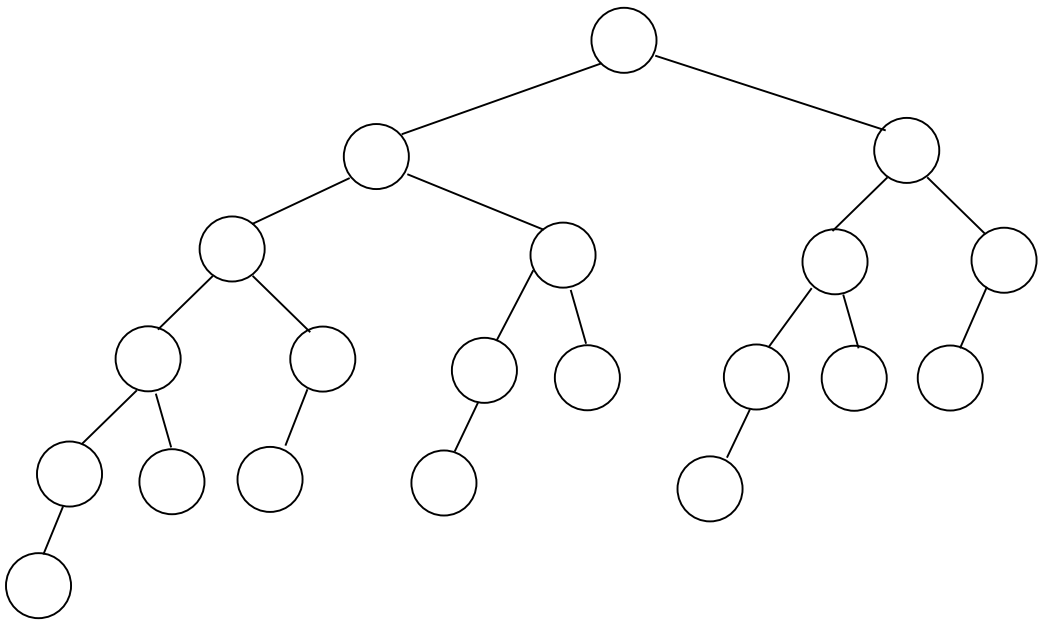
Beispiel: Fibonacci-Bäume . . .



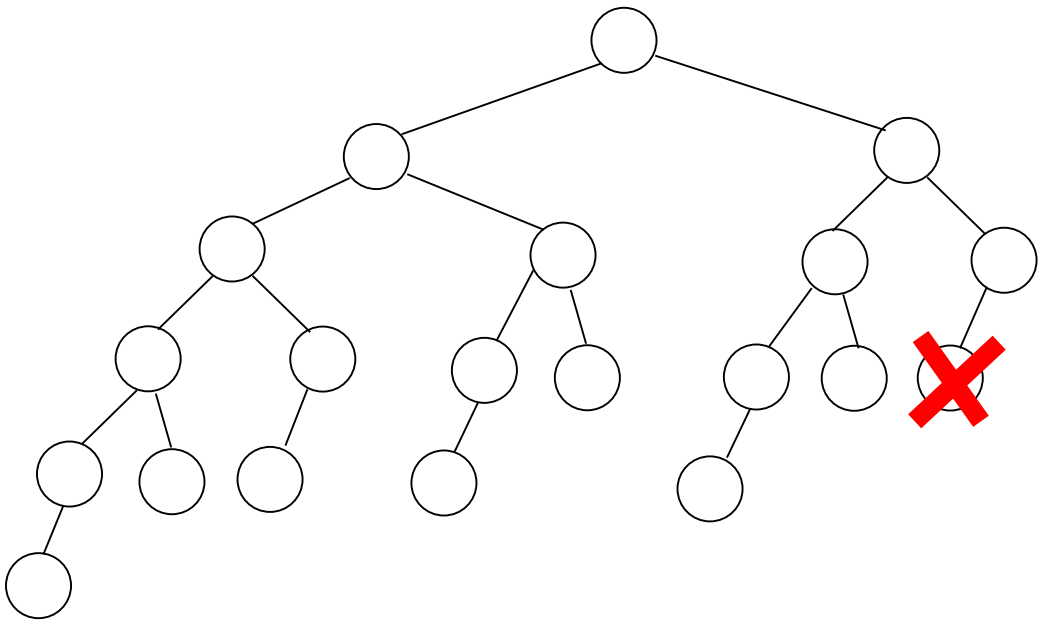
Fibonacci-Bäume der Tiefen $-1, 0, 1, 2, 3$

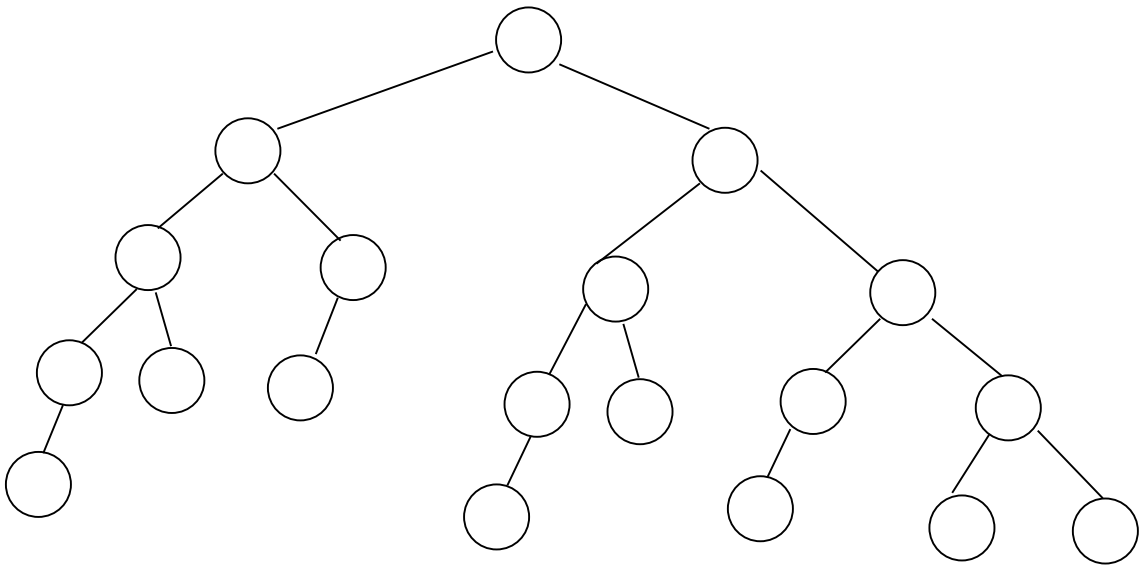


Fibonacci-Bäume der Tiefen 4 und 5



Blatt v:
Starte bei Wurzel, gehe rekursiv in flacheren Unterbaum.
Entfernen von v löst Rotations**kaskade** aus.





Fertig!

Programmierung von **AVL_delete**: Nicht prüfungsrelevant!

Können muss man aber: Ausführen der Löschung und Rebalancierung, analog zur Einfügung.

In Prozeduren: `shallower` ist ein Flagbit, das anzeigt, ob der soeben bearbeitete Teilbaum durch die Löschung flacher geworden ist.

Funktion AVL_delete`(T,x): (AVL_Tree,boolean)`

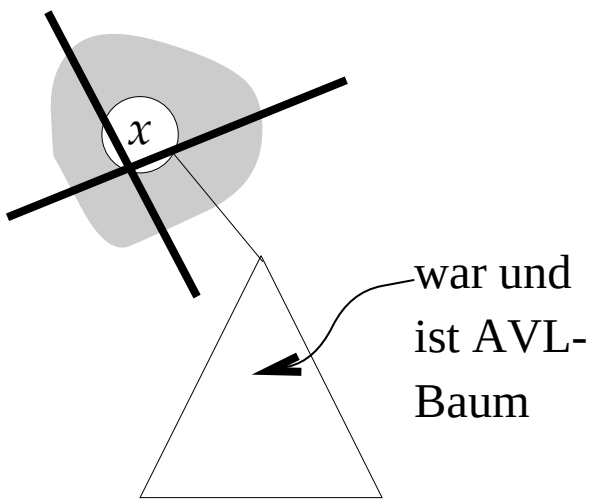
```
(1)  // Eingabe: T: AVL_Tree, x: key
(2)  // Ausgabe: T: AVL_Tree, shallower: boolean

(3)  1. Fall: T = □   // x nicht da
(4)     return (T, false).
(5)        // Keine Rebalancierung nötig!

(6)  2. Fall: T ≠ □ and x < T.key.
(7)     (T.left, left_shallower) ← AVL_delete(T.left, x);
(8)        // Rekursives Löschen im linken Unterbaum
(9)     (T, shallower) ← RebalanceDelLeft(T,left_shallower);
(10)       // Rebalancierung in der Wurzel von T, s. unten
(11)    return (T, shallower).
```

(12) **3. Fall:** `T` $\neq \square$ **and** `T.key < x`.
(13) $(\texttt{T.right}, \texttt{right_shallower}) \leftarrow$ **AVL_delete**$(\texttt{T.right}, \texttt{x})$;
(14) // Rekursives Löschen im rechten Unterbaum
(15) $(\texttt{T}, \texttt{shallower}) \leftarrow$ **RebalanceDelRight**$(\texttt{T}, \texttt{right_shallower})$;
(16) // Rebalancierung in der Wurzel von `T`, **symmetrisch zu 2. Fall**
(17) **return** $(\texttt{T}, \texttt{shallower})$.

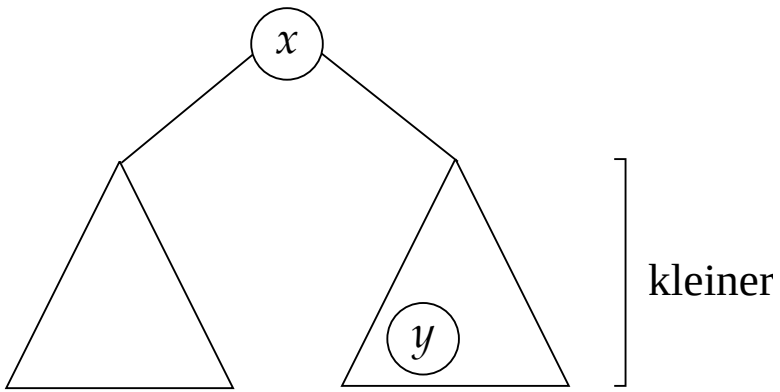
(18) **4. Fall:** `T` $\neq \square$ **and** `T.key = x`.
(19) // Entferne Wurzelknoten!
(20) **Fall 4a:** `T.left` $= \square$



(21) **return** $(\texttt{T.right}, \textit{true})$.
(22) // Baum ist flacher als vorher.

(23) **Fall 4b:** `T.right` $= \square$
(24) **return** $(\texttt{T.left}, \textit{true})$.

(25) **Fall 4c:** T $\neq \square$ **and** beide Unterbäume nicht leer




```
(26)  (T.right, v, right_shallower) ← AVL_extractMin(T.right) ;
(27)  v.left ← T.left; v.right ← T.right;
(28)  v.bal ← T.bal;
(29)  T ← v;  // v ersetzt die Wurzel von T
(30)  (T, shallower) ← RebalanceDelRight(T,right_shallower) ;
(31)  return (T, shallower) .
```

Funktion AVL_extractMin(T): (AVL_Tree,boolean)

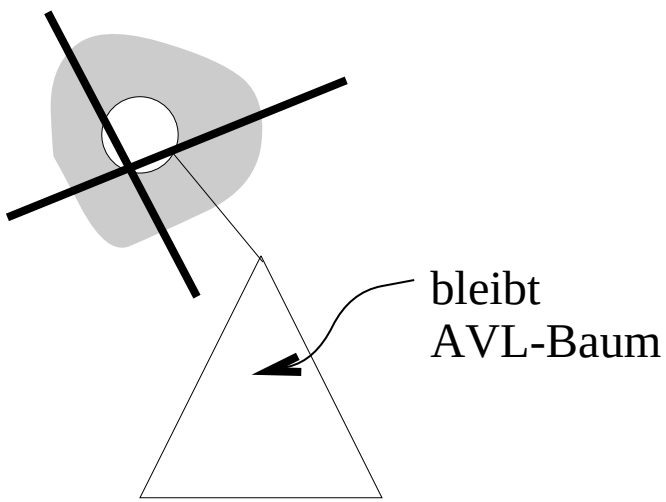
```
(1)  // Eingabe: T: AVL_Tree mit T ≠ □
(2)  // Ausgabe: T, v: AVL_Tree; shallower: boolean

(3)     // Knoten v mit minimalem Eintrag aus T ausgeklinkt
(4)     // shallower = true, falls T flacher geworden ist
     ...
```

In **AVL_extractMin**(T):

Fall „Schluss“:

$\texttt{T.left} = \square$



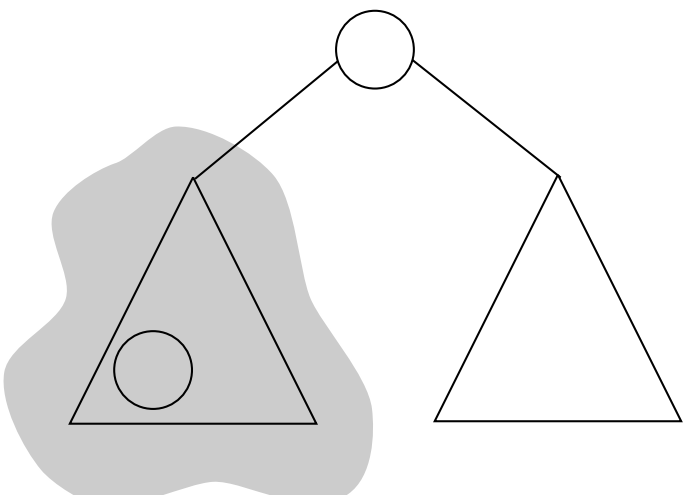
Wurzel abhängen:

$\texttt{v} \leftarrow \texttt{T}$;
$\texttt{T} \leftarrow \texttt{T.right}$
return $(\texttt{T}, \texttt{v}, \mathit{true})$.

In **Prozedur AVL_extractMin**(T):

Fall „Rekursion“:

T.left $\neq \square$



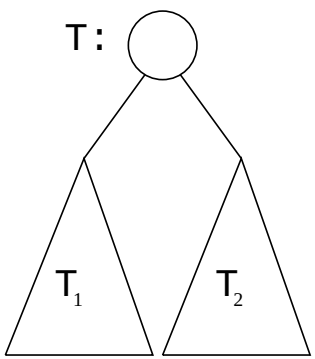
(T, v, left_shallower) $\leftarrow$ **AVL_extractMin**(T.left);
(T, shallower) $\leftarrow$ **RebalanceDelLeft**(T,left_shallower);
return(T, v, shallower).

RebalanceDelLeft(`T,left_shallower`)

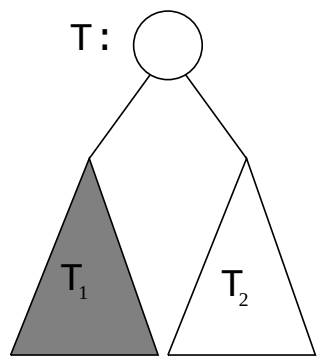
// prüft Balancebedingung in der Wurzel von `T`, korrigiert

// (**RebalanceDelRight**: symmetrisch)

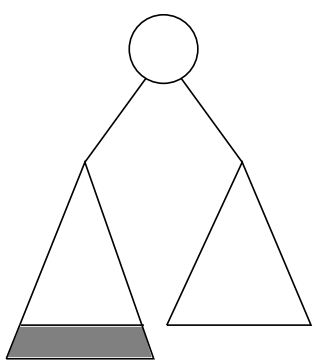
Situation:



durch rekursives
Löschen in T_1
umgebaut zu



$T_1 =$ `T.left` verändert; „`left_shallower`“ teilt mit, ob T_1 flacher geworden ist.
`T.bal` ist bislang unverändert.

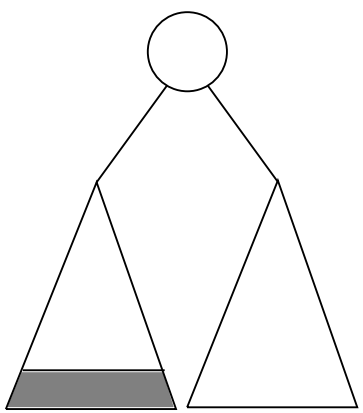
Fall	Aktion
Fall RebDL-1: `left_shallower` $=$ *false*	`shallower` $\leftarrow$ *false*;
Fall RebDL-2: `left_shallower` $=$ *true* $\wedge$ `T.bal` $=$ -1	
 geschrumpft	`shallower` $\leftarrow$ *true*; `T.bal` $\leftarrow$ 0;

Fall

Aktion

Fall RebDL-3:

$\texttt{left_shallower} = \mathit{true} \wedge \texttt{T.bal} = 0$

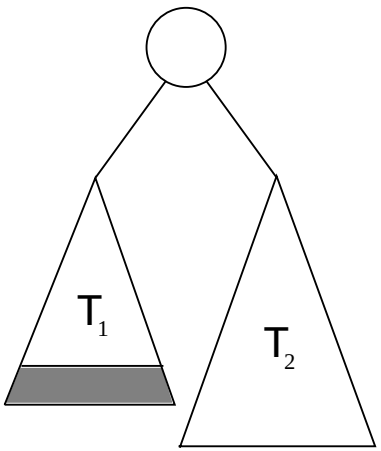


$\texttt{shallower} \leftarrow \mathit{false};$
$\texttt{T.bal} \leftarrow 1;$

geschrumpft

Fall RebDL-4:

$\texttt{left_shallower} = \textit{true} \wedge \texttt{T.bal} = 1$



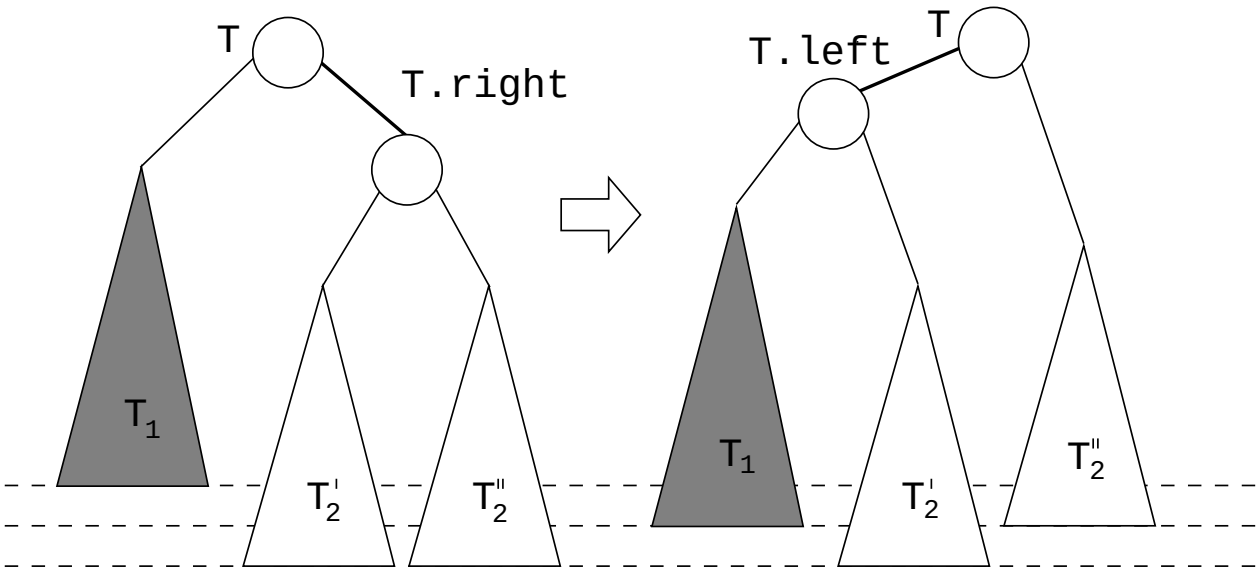
geschrumpft

Unterfälle je nach Aussehen von T_2.

Wir wissen: T_2 hat Tiefe mindestens 1.

(T_1 kann vor der Löschung nicht leer gewesen sein, hatte also Tiefe mindestens 0.)

Fall RebDL-4.1: `T.right.bal` $= 0$; // Zwei gleich tiefe Unterbäume in `T.right`

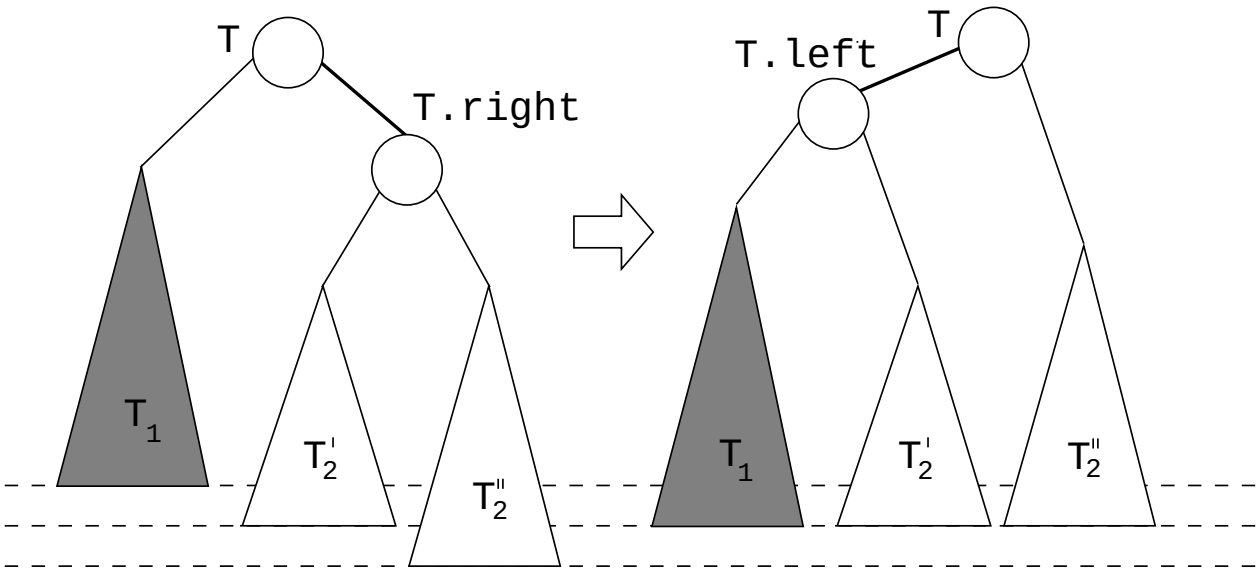


Linksrotation!

$\texttt{T} \leftarrow$ **rotateL**$(\texttt{T})$;
`T.left.bal` $\leftarrow 1$;
`T.bal` $\leftarrow$ -1;
`shallower` $\leftarrow$ *false*;

// Rebalancierung beendet.

Fall RebDL-4.2: `T.right.bal` $= 1$; // der äußere Unterbaum in `T.right` ist tiefer

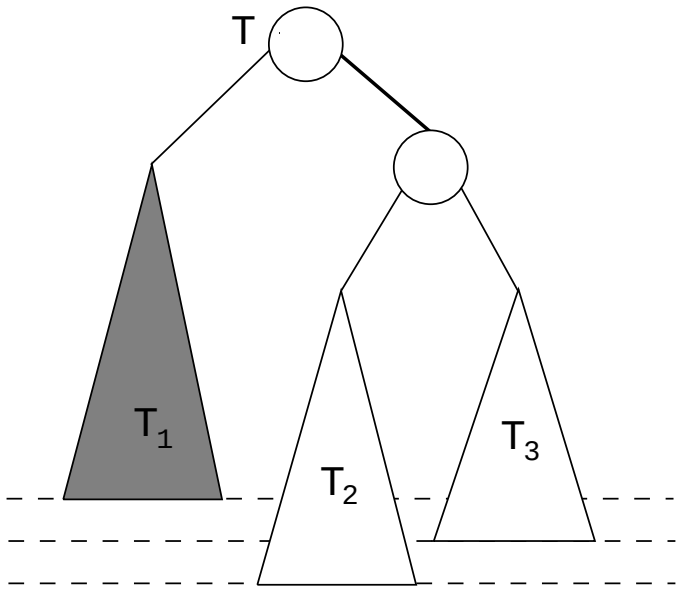


Linksrotation!

```
T ← rotateL(T);
T.left.bal ← 0;
T.bal ← 0;
shallower ← true;
```

// Rebalancierung muss weitergehen!

Fall RebDL-4.3: `T.right.bal` = -1; // der innere Unterbaum in `T.right` ist tiefer

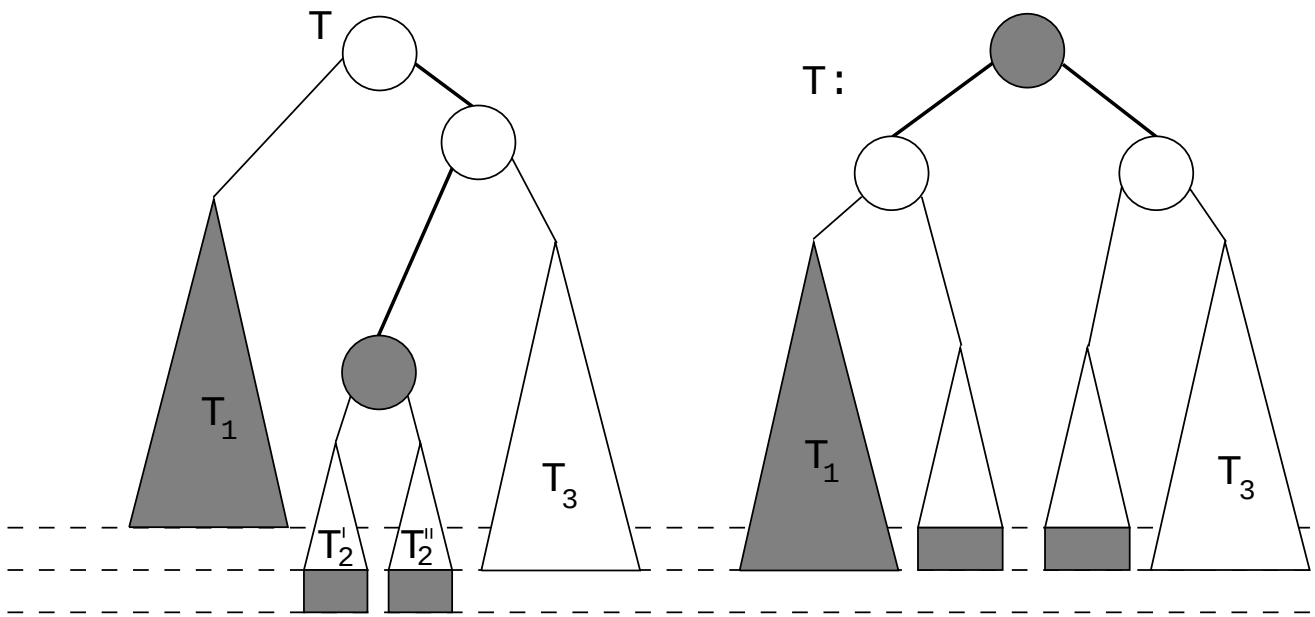


// Beobachte: Der zu tief hängende Teilbaum ist auf dem Weg „rechts-links“ zu erreichen.

// → Doppelrotation!

Fall RebDL-4.3: (Forts.)

Betrachte Teilbäume von $T_2 =$ `T.right.left`.



Doppelrotation! `T` $\leftarrow$ **rotateRL**(`T`);

`shallower` $\leftarrow$ *true*;

Neue Werte der Balancefaktoren:

Neue Werte der Balancefaktoren:

altes	neues		
`T.bal`	`T.left.bal`	`T.right.bal`	`T.bal`
-1	0	1	0
0	0	0	0
1	-1	0	0

(Das alte `T.bal` enthält (nach der Doppelrotation) den Balancefaktor, der ursprünglich in der Wurzel des „mittleren“ Unterbaums T_2 stand.)

Ende des nicht prüfungsrelevanten Lösch-Programms.

Achtung: Man muss Löschungen an Beispielen durchführen können!

Proposition 4.3.6

Die rekursive Prozedur `AVL_delete`(T, x) führt die Wörterbuchoperation *delete* korrekt durch. D. h.: es entsteht ein AVL-Baum für das Wörterbuch *delete*(f_T, x).

Die Prozedur hat Laufzeit $O(\log n)$ und führt **an jedem Knoten auf dem Weg** vom gelöschten Knoten zur Wurzel höchstens eine Einfach- oder Doppelrotation durch.

Satz 4.3.7

In **AVL-Bäumen** kostet jede **Wörterbuchoperation** Zeit

$$\boldsymbol{O(\log n)},$$

wenn n die aktuelle Anzahl der Einträge ist.

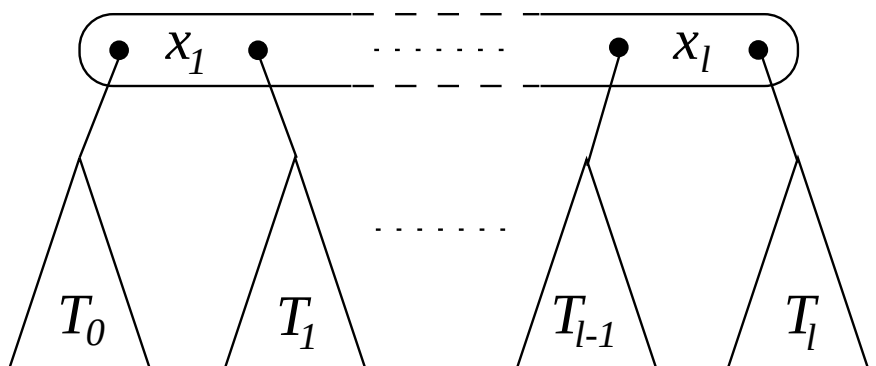
Vorlesungsvideo:

2-3-Bäume

4.4 Mehrweg-Suchbäume

Bäume aus Knoten mit variablem (Ausgangs-)Grad.

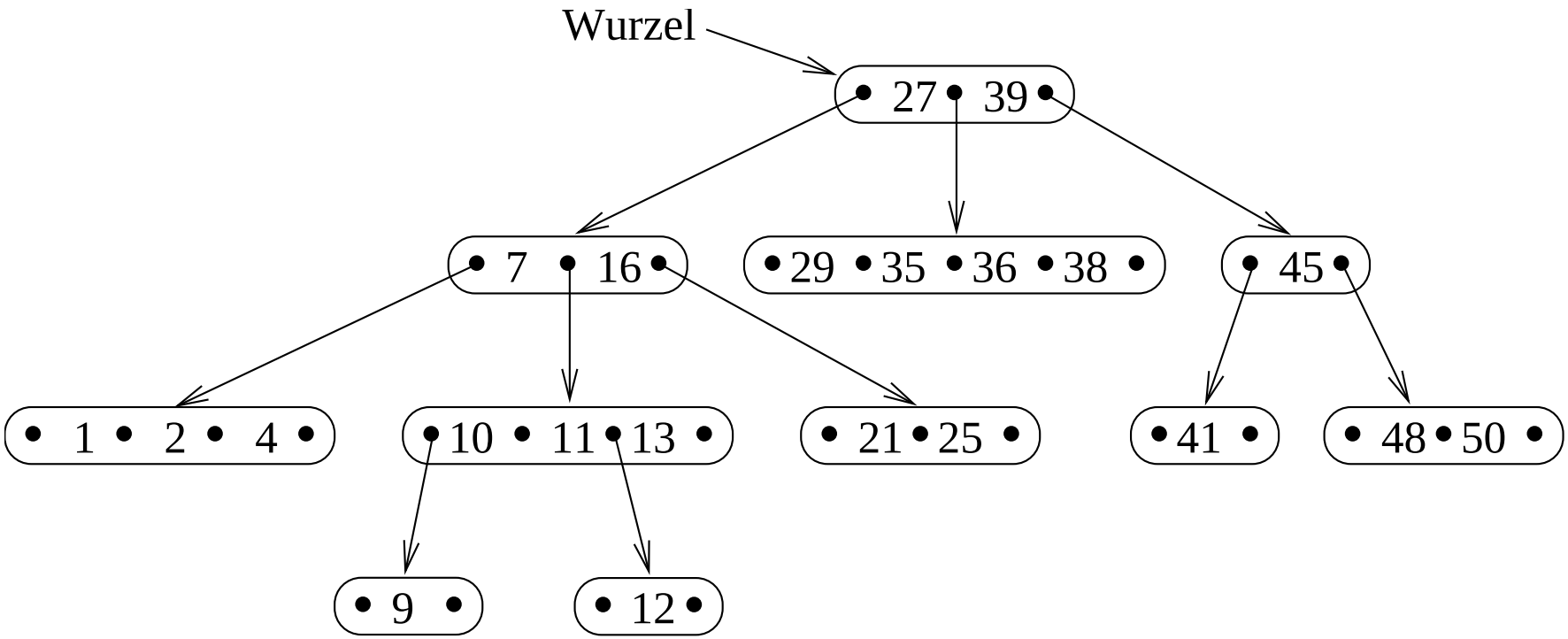
Ein Knoten mit $l \geq 1$ Schlüsseln und $l + 1 \geq 2$ Unterbäumen:



l Schlüssel: $x_1 < \cdots < x_l$. $\quad$ $l + 1$ Unterbäume: $T_0, \ldots, T_l$.

Für $\boldsymbol{y_i}$ in T_i, $0 \leq i \leq l$ gilt: $\boldsymbol{y_0} < x_1 < \boldsymbol{y_1} < x_2 < \cdots < \boldsymbol{y_{l-1}} < x_l < \boldsymbol{y_l}$

Beispiel eines $\mathbb{N}$-MwSB:



„•“ ohne Pfeil steht für einen Zeiger auf einen leeren Baum $\square$.

Man überprüfe die Anordnungsbedingung in diesem Beispiel!

Definition 4.4.1

Mehrweg-Suchbäume (MwSBe) über dem Universum $(U, <)$ sind wie folgt **induktiv definiert**:

(0) Der leere Baum $\square$ ist ein U-MwSB.

(1) Ist $l \geq 1$ und sind $x_1 < x_2 < \cdots < x_l$ Schlüssel in U und sind $T_0, T_1, \ldots, T_l$ U-MwSBe mit:

$$y_0 < x_1 < y_1 < x_2 < \cdots < y_{l-1} < x_l < y_l, \text{ für alle } y_i \text{ in } T_i,\ 0 \leq i \leq l,$$

dann ist auch $(T_0, x_1, T_1, x_2, \ldots, x_{l-1}, T_{l-1}, x_l, T_l)$ ein U-Mehrweg-Suchbaum.

Analog: (U, R)-Mehrweg-Suchbäume, mit Menge R von Datensätzen.

Knotenformat ist dann: $(T_0, (x_1, r_1), T_1, (x_2, r_2), \ldots, (x_{l-1}, r_{l-1}), T_{l-1}, (x_l, r_l), T_l)$.

Rekursive **Suche** in einem Mehrwegsuchbaum T nach Schlüssel x aus U.

(Abstrakt, aber auch die Implementierungen arbeiten nach diesem Muster.)

Wenn $T = \square$, dann gibt es keinen Eintrag mit Schlüssel x.

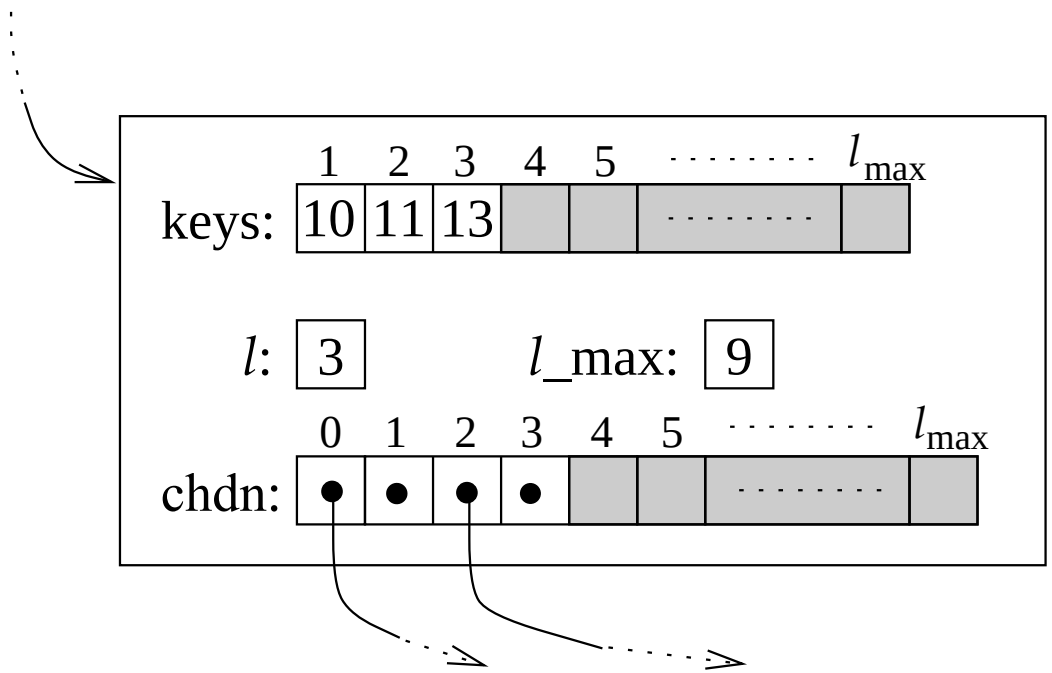
Wenn $T = (T_0, x_1, T_1, x_2, \ldots, x_{l-1}, T_{l-1}, x_l, T_l)$, dann:

- Falls $x = x_i$ für ein i, dann sitzt Eintrag mit Schlüssel x im Wurzelknoten von T.
- Sonst: Bestimme das eindeutige i mit $x_i < x < x_{i+1}$ und suche rekursiv in T_i.

  (Dabei sollen x_0 und x_{n+1} die „virtuellen Schlüssel“ $-\infty$ bzw. $+\infty$ sein.)

Implementierung der Knoten in einem Mehrwegsuchbaum:

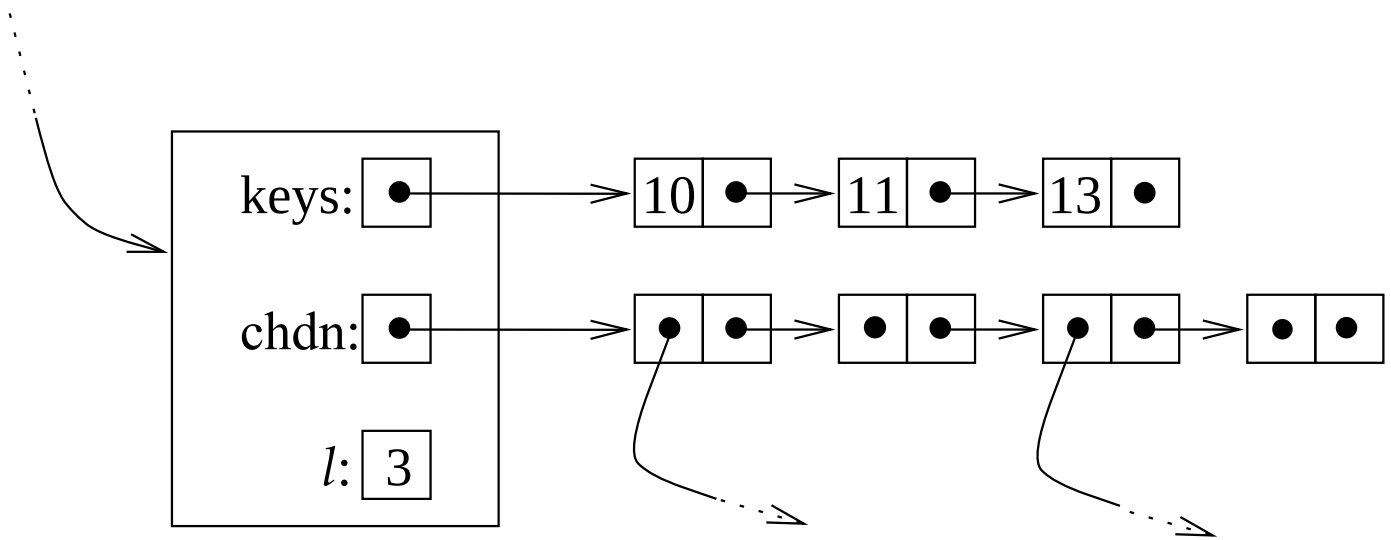
Variante 1: Jeder Knoten enthält zwei Arrays (oder „vectors“), *keys* und *chdn* (für „children“), bzw. noch ein Array *data* für die Daten zu den Schlüsseln.



Implementierung der Knoten in einem Mehrwegsuchbaum:

Variante 2: Schlüssel und Kinder als Listen *keys* und *chdn*.

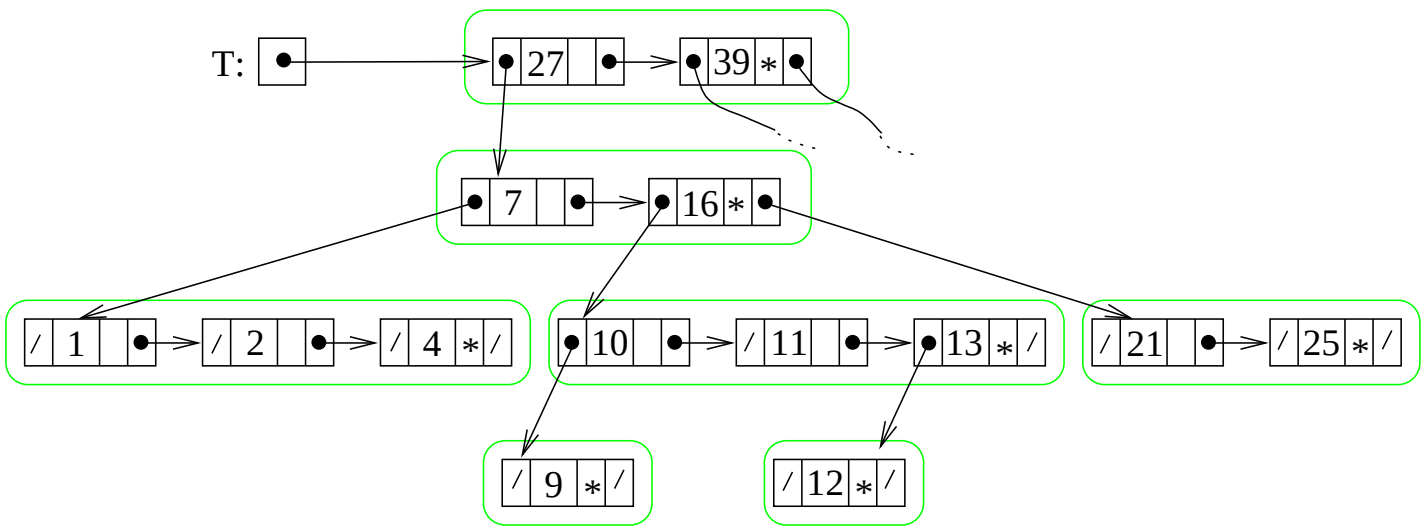
(Daten sind gegebenenfalls in die Schlüsselliste zu integrieren.)



Vorteil: Kein Platzverbrauch für nicht existierende Schlüssel/Unterbäume.

Implementierung der Knoten in einem Mehrwegsuchbaum:

Variante 3: Ein MwSB-Knoten wird durch eine aufsteigend sortierte lineare Liste von **Binärbaumknoten** dargestellt.



Jeder Knoten hat Platz für Schlüssel, (Daten,) Zeiger auf Unterbaum, Zeiger auf nächsten in Liste. Im Knoten muss als Boolescher Wert vermerkt sein, ob es sich um den Knoten am Listenende handelt. (Im Bild: „*“.) Ein solcher Knoten hat dann zwei Zeiger auf Unterbäume.
Elegant: Für die Suche kann man die gewöhnliche Suchprozedur für binäre Suchbäume benutzen.

Spezialfall: 2-3-Bäume

Definition 4.4.2

Ein Mehrweg-Suchbaum T heißt ein **2-3-Baum**, wenn gilt:

(a) Jeder Knoten enthält 1 oder 2 Schlüssel
(also hat jeder Knoten **2** oder **3** Unterbäume, was der Struktur den Namen gibt);

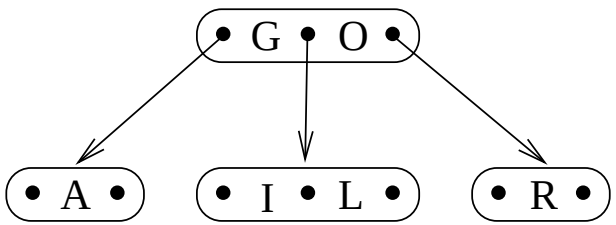
(b) Für jeden Knoten v in T gilt: Wenn v **einen** leeren Unterbaum hat, so sind **alle** Unterbäume unter v leer, d.h. v ist dann **Blatt**;

(c) Alle Blätter von T haben dieselbe Tiefe.

Beispiele:

Leerer 2-3-Baum: □

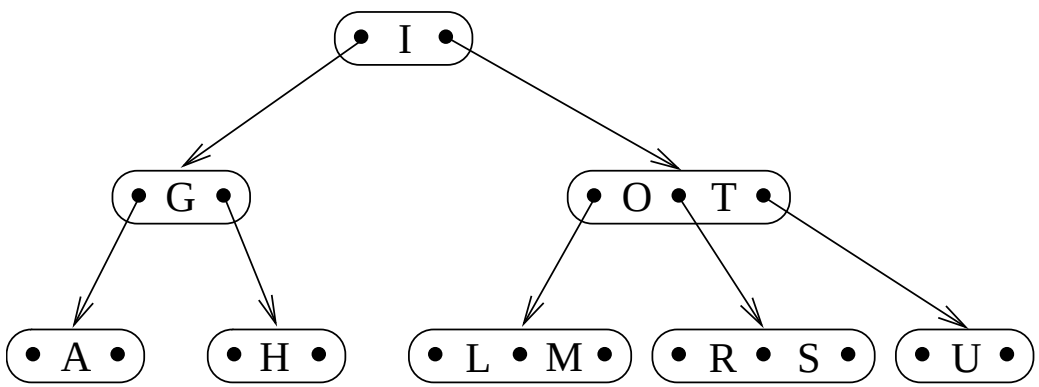
2-3-Baum mit zwei Ebenen:



Level 1

Level 0 = Blattebene

2-3-Baum mit drei Ebenen:

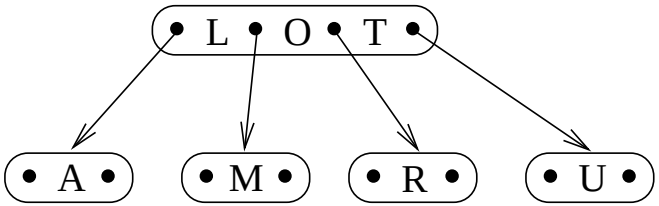


Level 2

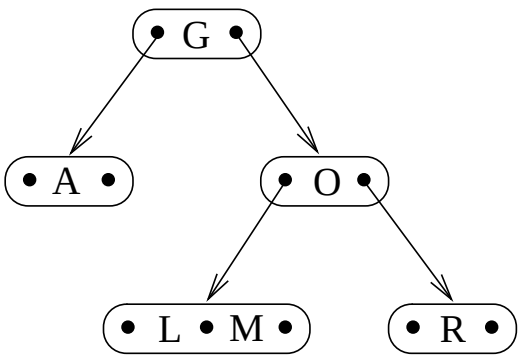
Level 1

Level 0 = Blattebene

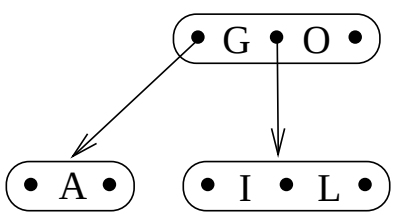
NB: Die Ebenen werden von den Blättern her nummeriert.



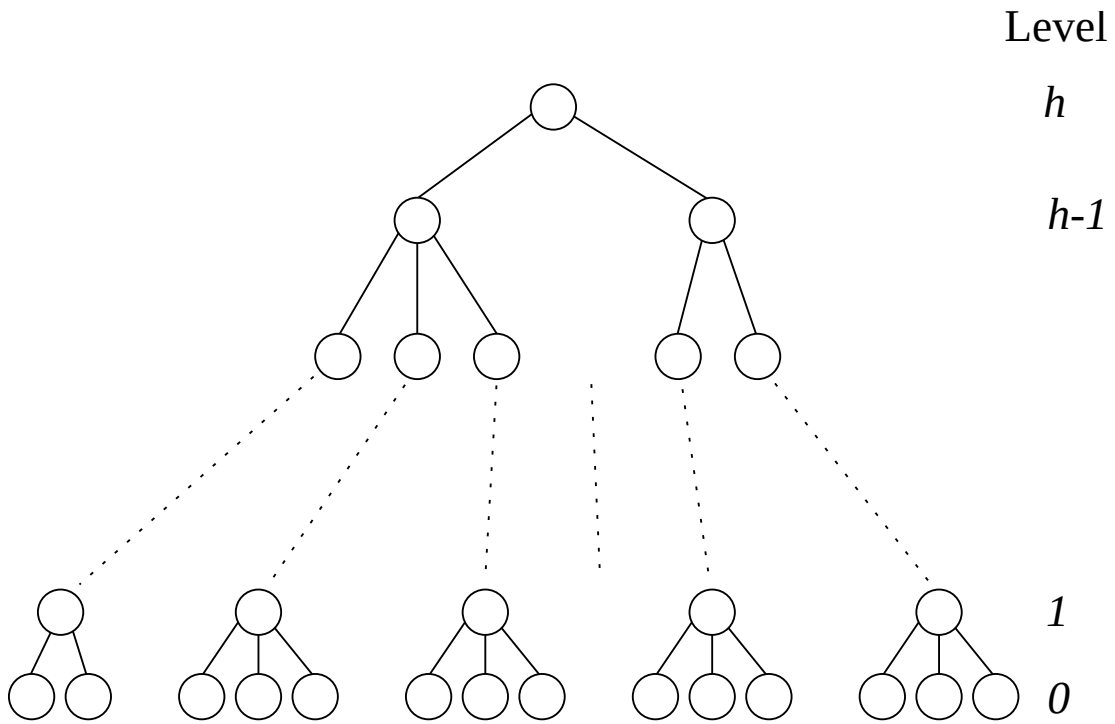
Kein 2-3-Baum (zu großer Grad).



Kein 2-3-Baum (Blätter in verschiedenen Tiefen).



Kein 2-3-Baum (Leerer Unterbaum in Nicht-Blatt).



Proposition 4.4.3

Die Tiefe eines 2-3-Baums mit n Einträgen ist mindestens $\lceil \log_3(n+1) \rceil - 1$ und höchstens $\lfloor \log_2(n+1) \rfloor - 1$. $[\log_3 x \approx 0.63 \log_2 x]$ *Beweis*: Übung.

Suche in 2-3-Bäumen

lookup(T, x), für 2-3-Baum T, $x \in U$.

1. Fall: $T = \square$. **return** „nicht vorhanden“.

2. Fall: $T = (T_0, x_1, T_1)$ oder $T = (T_0, x_1, T_1, x_2, T_2)$, evtl. mit Datensatz r_1 (und r_2).

Sequentielle Abarbeitung der folgenden Tests:

2a: $x < x_1$: **return lookup**(T_0, x).

2b: $x = x_1$: **return** „gefunden“ bzw. r_1.

2c: (x_2 existiert nicht $\vee$ $x < x_2$): **return lookup**(T_1, x).

2d: $x = x_2$: **return** „gefunden“ bzw. r_2.

2e: $x_2 < x$: **return lookup**(T_2, x).

Zeitaufwand: Wenn x im Knoten v_x sitzt: Für jeden Knoten v auf dem Weg von der Wurzel zu v_x entstehen Kosten $O(1)$. Erfolglose Suche endet auf Blattebene.

$\Rightarrow$ Zeit für Suche in 2-3-Bäumen mit n Einträgen ist $O(\log n)$.

Einfügen in 2-3-Bäumen

insert(T, x, r), für 2-3-Baum T, $x \in U$.

Resultat: Neuer Baum T', Boolescher Wert `higher` (gibt an, ob T' höher als T ist).

Invariante (I): (Diese werden wir benutzen und kontrollieren!)

T' höher als $T \;\Rightarrow\; T'$ hat in der Wurzel *nur einen* Schlüssel.

1. Fall: $T = \square$.

Erzeuge neuen 2-3-Baum-Knoten `T` mit Eintrag (x, r) und zwei leeren Unterbäumen.

return (`T`, *true*)

(I) stimmt.

2. Fall: $T = (T_0, (x_1, r_1), T_1)$ oder $T = (T_0, (x_1, r_1), T_1, (x_2, r_2), T_2)$

2a: $x < x_1$: **insert**(T_0, x, r) liefert T_0' (an der Stelle von T_0) und `sub_higher`.
Dann: **Rebalancierung** (s. u.).

2b: $x = x_1$: Update: Ersetze r_1 durch r. **return** (T, false);

2c: (x_2 existiert nicht $\vee$ $x < x_2$): **insert**(T_1, x, r) liefert T_1' (an Stelle von T_1) und `sub_higher`.
Dann: **Rebalancierung** (s. u.).

2d: $x = x_2$: Update: Ersetze r_2 durch r. **return** (T, false);

2e: $x_2 < x$: **insert**(T_2, x, r) liefert T_2' (an Stelle von T_2) und `sub_higher`.
Dann: **Rebalancierung** (s. u.).

Rebalancierung

Fälle 2a/2c/2e:

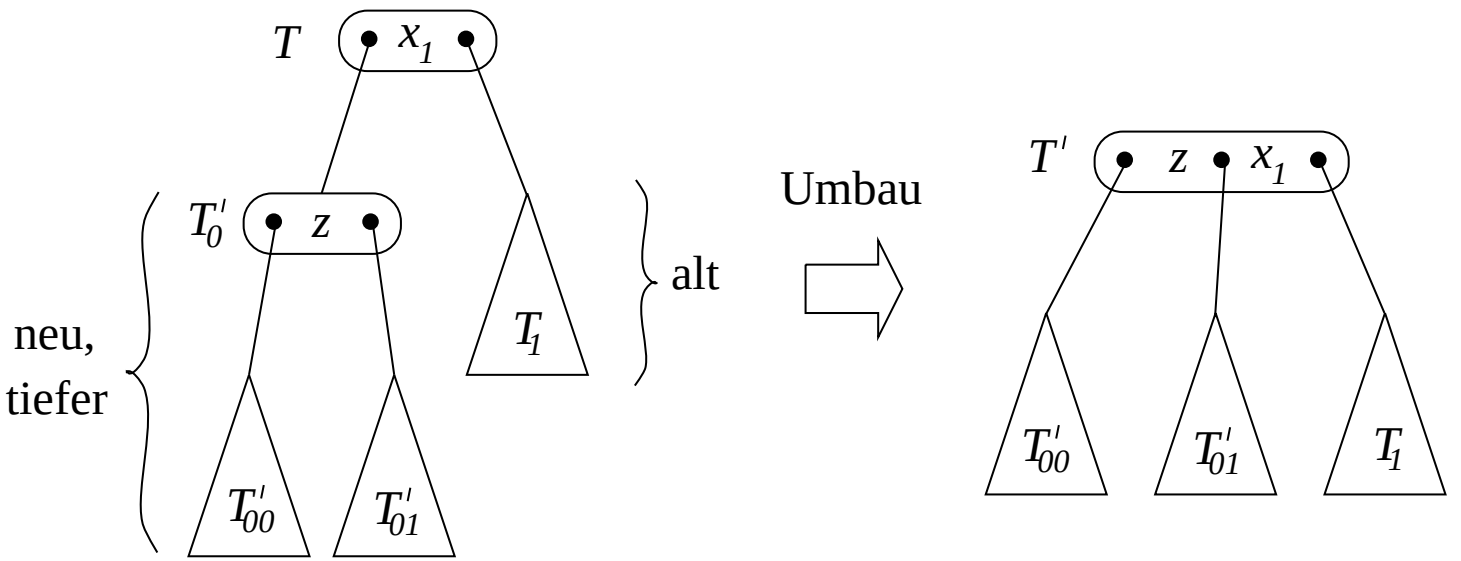
Fall 2a(i)/2c(i)/2e(i): `sub_higher` $=$ *false*.

// Der veränderte Teilbaum ist **nicht tiefer** geworden.

return$(T,$ *false*$)$.

Fall 2a(ii)/2c(ii): x_2 existiert nicht und `sub_higher` $=$ *true*

2a(ii): Benutze **(I)**!

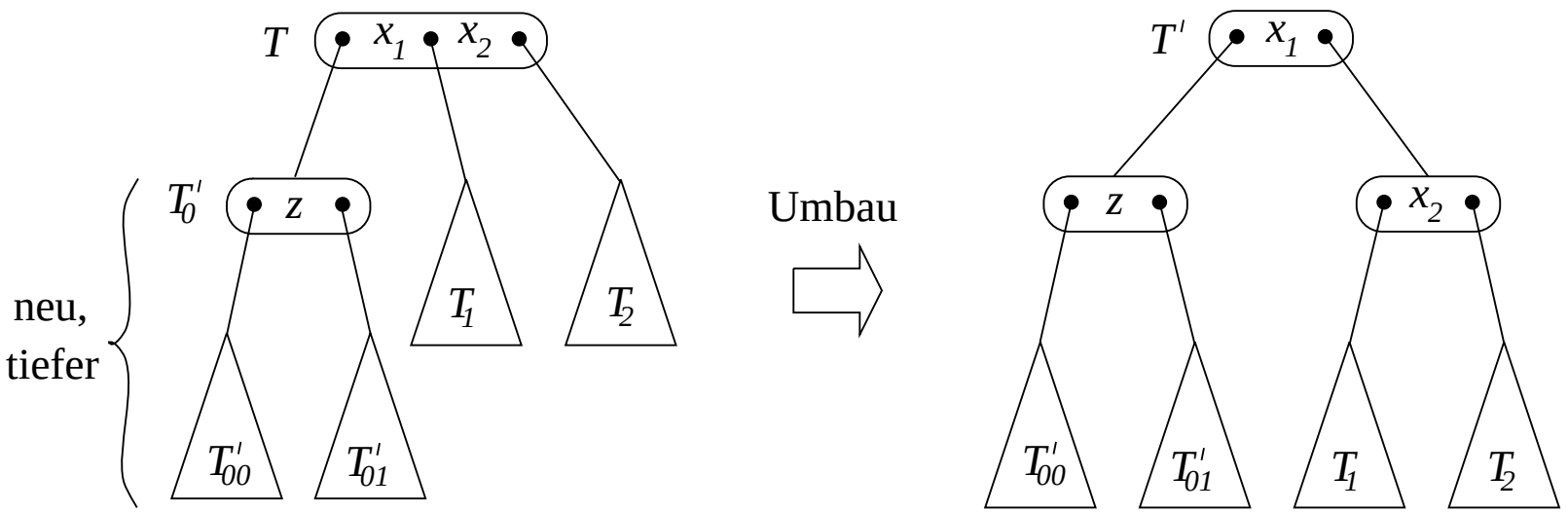


return(T', false).

2c(ii): analog. Fall 2a(iii)/2c(iii)/2e(iii): x_2 **existiert** und `sub_higher` $=$ *true*

Benutze jeweils **(I)**!

2a(iii):



return(T', true).

Prüfe: **(I)** erfüllt.

2c(iii)/2e(iii): analog.

Beobachtung: Die Tiefe von 2-3-Bäumen wächst durch „Spalten der Wurzel“.

Proposition 4.4.4

(a) Die Prozedur **insert** ist korrekt, d. h. der Aufruf **insert**(T, x, r) liefert einen 2-3-Baum für *insert*(f_T, x, r).

(b) Das Einfügen eines Schlüssels in einen 2-3-Baum mit n Schlüsseln benötigt Zeit $O(\log n)$ und bewirkt die Erzeugung von höchstens $\log_2 n$ neuen Knoten.

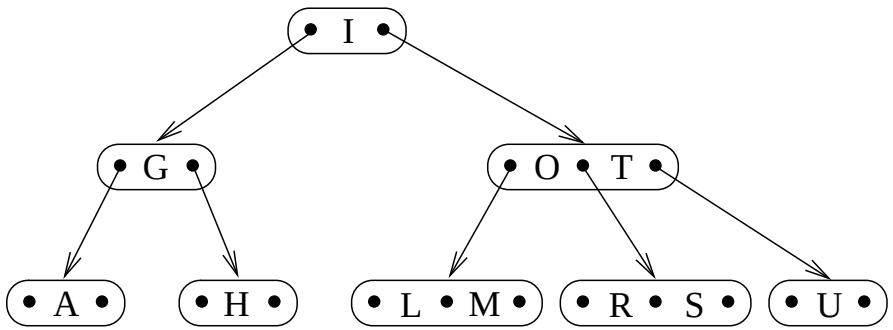
Beweis:

(a) Man kontrolliert in jedem Fall des Algorithmus nach, dass die 2-3-Baum-Struktur wiederhergestellt wird.

(b) Für die Bearbeitung **eines Levels** in der *insert*-Prozedur wird Zeit $O(1)$ benötigt.
Auf jedem der $\leq \log n$ Levels wird höchstens 1 Knoten neu gebildet.

Folgerung: Für jedes $n \geq 0$ existiert ein 2-3-Baum mit n Schlüsseln. (Man füge in einen anfangs leeren Baum die Schlüssel $1, \ldots, n$ ein.)

Löschen in 2-3-Bäumen

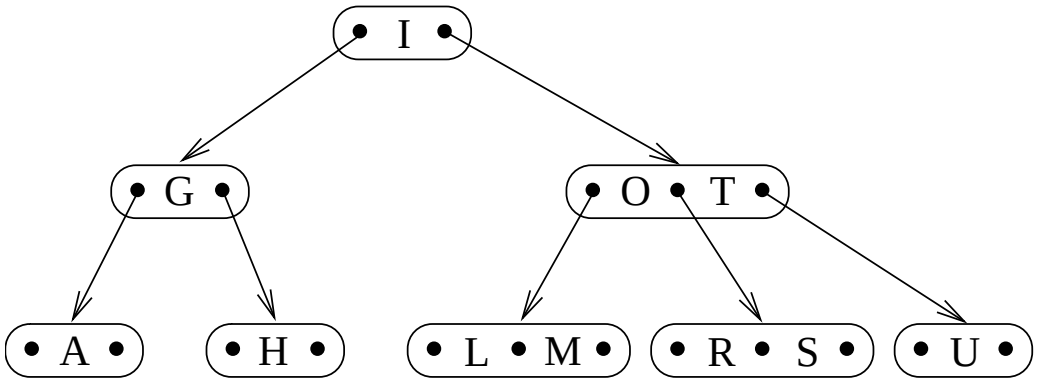


Prinzip: Suche kleinsten Schlüssel $y \geq x$, der in einem Blatt gespeichert ist. (x oder Inorder-Nachfolger von x.)

Setze y an die Stelle von x und entferne y.

Beispiele: U wird einfach gelöscht. Um I zu löschen: L in die Wurzel ziehen.

Daher: Nur Löschen eines Eintrags y in einem **Blatt** ist relevant.



• Einfacher Fall: Neben y hat das Blatt noch andere Einträge (Beispiel: M).

Keine Umstrukturierung nötig.

• Rebalancierungsfall: Durch das Streichen von y wird das Blatt leer, verliert also an Höhe (Beispiel: A).

Ähnlich wie bei den Einfügungen werden **flacher gewordene Teilbäume** durch lokale Umstellungen, von unten nach oben, wieder **auf die Höhe der anderen Teilbäume gebracht**. (Details weggelassen.)

2-3-4-Bäume

Definition Ein **2-3-4-Baum** ist ein **Mehrweg-Suchbaum** mit:

(a) Jeder Knoten hat 2, 3 oder 4 Kinder (1, 2 oder 3 Schlüssel)

(b) Für jeden Knoten v in T gilt: v hat **einen** leeren Unterbaum $\Rightarrow$ **alle** Unterbäume unter v sind leer, d. h. v ist **Blatt**;

(c) Alle Blätter von T haben dieselbe Tiefe.

Mitteilung Die Wörterbuchoperationen können auch mit 2-3-4-Bäumen so implementiert werden, dass sie **logarithmische Zeit** benötigen. Die Rebalancierung lässt sich alternativ „top-down“ implementieren, auf dem Weg zur Einfügestelle bzw. Löschstelle, ohne Zurücklaufen.
Lit.: [D./Mehlhorn/Sanders]. (Dort: Einträge nur in Blättern.)

Man erhält den allgemeineren Begriff des **(a, b)-Baums**, wenn man in der obigen Definition „2, 3 oder 4 Kinder“ durch „zwischen a und b Kindern, für $2 \leq a \leq (b+1)/2$ (Ausnahme: die Wurzel hat zwischen 2 und b Kinder)“ ersetzt.

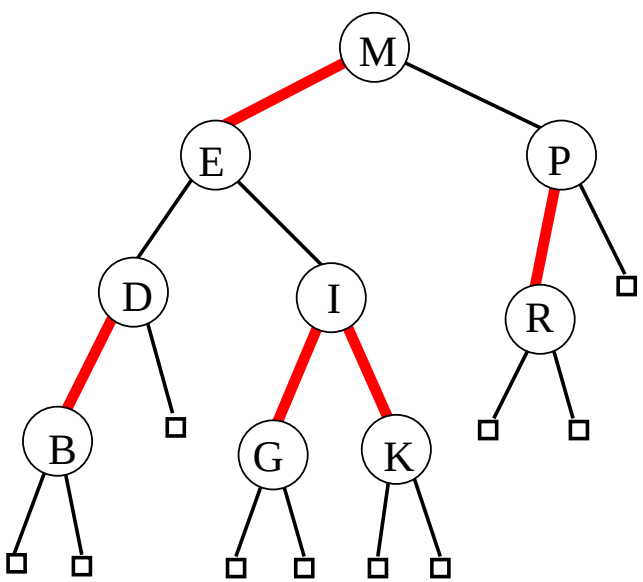
Vorlesungsvideo:

Rot-Schwarz-Bäume und B-Bäume

Rot-Schwarz-Bäume

Definition

Ein **(linksgeneigter) Rot-Schwarz-Baum** ist ein **binärer Suchbaum** mit:



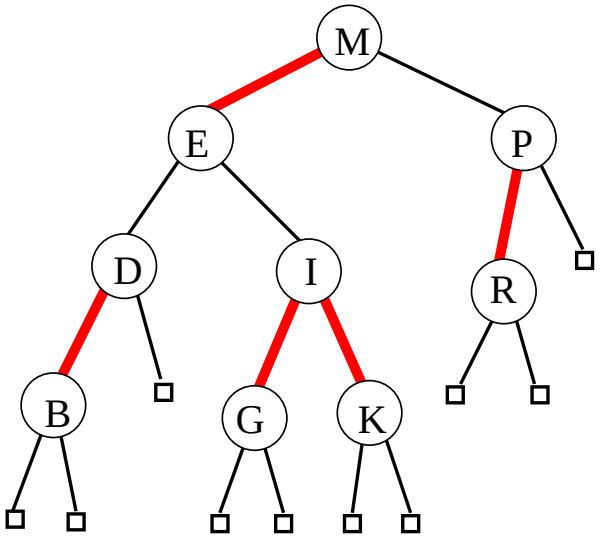
(a) Jede Kante hat eine Farbe **rot** oder **schwarz** (1 Flagbit!).

(b) Kanten zu (leeren, externen!) Blättern sind **schwarz**.

(c) Auf keinem Weg folgen zwei rote Kanten aufeinander.

(d) Wenn ein Knoten nur eine rote Ausgangskante hat, so führt diese zum linken Kind.

(e) Auf jedem Weg von der Wurzel zu einem Blatt liegen gleich viele schwarze Kanten.



Schwarz-Tiefe: 2.

Die **Schwarz-Tiefe** $b(v)$ eines Knotens v ist die Anzahl der schwarzen Kanten auf einem Weg von v zu einem Blatt.

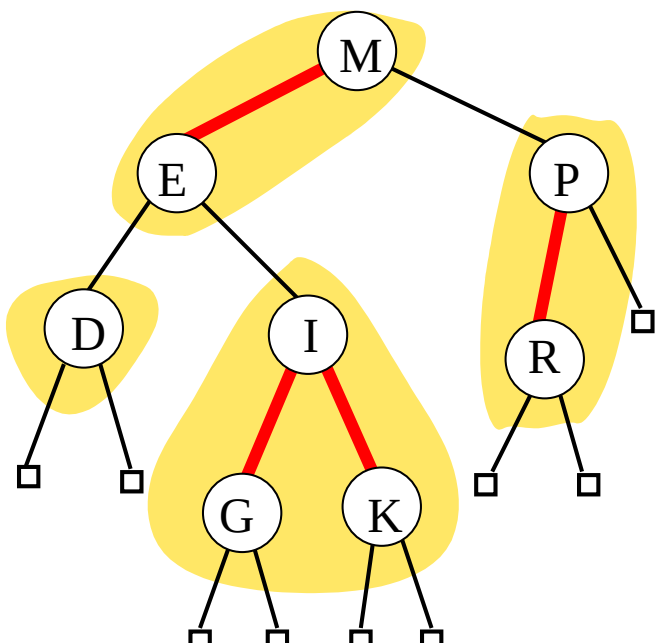
Die **Schwarz-Tiefe** $b(T)$ des Rot-Schwarz-Baums T ist die Schwarz-Tiefe des Wurzelknotens von T.

Mitteilung

Die Wörterbuchoperationen können auch mit Rot-Schwarz-Bäumen so implementiert werden, dass sie **logarithmische Zeit** benötigen. Weiterführende Operationen (wie das Spalten von Bäumen oder die Vereinigung von Bäumen) lassen sich mit Rot-Schwarz-Bäumen gut implementieren.
Lit.: [Cormen, Leiserson, Rivest, Stein] und [Sedgewick].

Wenn man einen schwarzen Knoten mit seinen roten Kindern (0, 1 oder 2) zu einem „Super-Knoten“ zusammenfasst, erhält man einen 2-3-4-Baum.

D. h.: Rot-Schwarz-Bäume sind spezielle Darstellungen von 2-3-4-Bäumen, die nur Binärbaumknoten benutzen.



B-Bäume

Ein **B-Baum** zum **Parameter $t \geq 2$** ist ein $(t, 2t)$-Baum, also ein **Mehrweg-Suchbaum** mit:

(a) Jeder Knoten hat maximal $2t$ Kinder ($\leq 2t - 1$ Schlüssel);
(b) Jeder Knoten außer der Wurzel hat mindestens t Kinder ($\geq t - 1$ Schlüssel);
die Wurzel hat mindestens 2 Kinder (≥ 1 Schlüssel);
(c) v hat **einen** leeren Unterbaum $\Rightarrow v$ ist **Blatt**;
(d) Alle Blätter von T haben dieselbe Tiefe.

Mitteilung Die Tiefe eines B-Baums mit Parameter t für n Einträge ist $\Theta(\log_t n) = \Theta((\log n)/(\log t))$, genauer mindestens $(\log n)/\log(2t) - 1$, höchstens $(\log n)/\log t$.

Beispiel: Die Tiefe eines B-Baums zu $t = 32$ ist höchstens $(\log n)/5$.

Jede Wörterbuchoperation betrifft höchstens zwei Knoten auf jedem Level, also höchstens $2(\log n)/\log t$ viele.
B-Bäume werden für die Implementierung von Wörterbüchern auf externen Speichermedien (Festplatte, SSD) benutzt, die eine höhere Latenzzeit haben und Lesen in größeren Blöcken erlauben. Details: Vorlesungen zu Datenbanktechniken.

SS 2021

Algorithmen und Datenstrukturen

5. Kapitel

Hashverfahren

Martin Dietzfelbinger

Mai 2021

5.1 Grundbegriffe

Gegeben: Universum U (auch *ohne Ordnung*) und Wertebereich R.

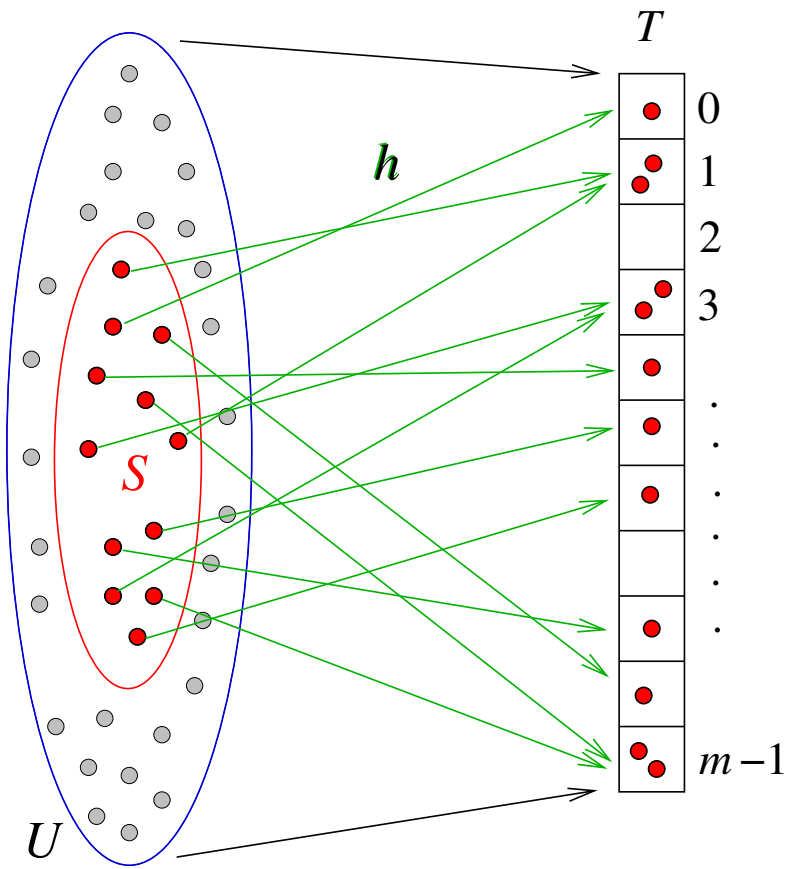
Hashverfahren (oder **Schlüsseltransformationsverfahren**) implementieren ein **Wörterbuch** (alias **dynamische Abbildung**, **assoziatives Array**)

$$f : S \to R, \text{ mit } S \subseteq U \text{ endlich.}$$

Ziel:

Zeit $O(1)$ pro Operation
„im Durchschnitt“
unabhängig vom Umfang der Datenstruktur!

Gegensatz **Suchbäume:** Zeit $O(\log(n))$ pro Operation.



Hashfunktion* $h: U \to [m]$

U : Universum der Schlüssel

$[m] = \{0, \ldots, m-1\}$:
Indexbereich für Tabelle T

$S \subseteq U$, $|S| = n$

Definitionsbereich von f.

* to hash (*engl.*): kleinhacken, kleinschneiden

Grundansatz:

Speicherung von $f\colon S \to R$ in Tabelle $\texttt{T}[0\,..\,m-1]$.

Aus Schlüssel $x \in U$ wird ein Index $h(x) \in [m]$ **berechnet**.

„x wird in $h(x)$ umgewandelt/**transformiert**.“

Idealvorstellung:

Speichere Paar $(x, f(x))$ in Zelle $\texttt{T}[h(x)]$.

Beispiel: $U = \{\mathtt{A}, \ldots, \mathtt{Z}, \mathtt{a}, \ldots, \mathtt{z}\}^{3..20}$

Nichtleere Wörter aus lateinischen Buchstaben mit mindestens drei und höchstens 20 Buchstaben.

$h\colon U \to [13]$ definiert durch

$$h(c_1 c_2 c_3 \cdots c_r) = \mathsf{num}(c_3) \bmod 13\,,$$

wobei

$$\begin{array}{ccccc} \mathsf{num}(\mathtt{A}) & = & \mathsf{num}(\mathtt{a}) & = & 0 \\ \mathsf{num}(\mathtt{B}) & = & \mathsf{num}(\mathtt{b}) & = & 1 \\ & \vdots & & \vdots & \\ \mathsf{num}(\mathtt{Z}) & = & \mathsf{num}(\mathtt{z}) & = & 25 \end{array}$$

* $x \bmod m$: Rest bei der Division von x durch m.

In Tabelle:

x	$f(x)$	$\mathsf{num}(c_3)$	$h(x)$
Januar	31	13	0
Februar	28	1	1
Maerz	31	4	4
April	30	17	4
Mai	31	8	8
Juni	30	13	0
Juli	31	11	11
August	31	6	6
September	30	15	2
Oktober	31	19	6
November	30	21	8
Dezember	31	25	12

$h(c_1 \dots c_r) = \mathsf{num}(c_3) \bmod 13$

j	$(x, f(x))$ mit $h(x) = j$
0	(Januar, 31); (Juni, 30)
1	(Februar, 28)
2	(September, 30)
3	
4	(Maerz, 31); (April, 30)
5	
6	(August, 31); (Oktober, 31)
7	
8	(Mai, 31); (November, 30)
9	
10	
11	(Juli, 31)
12	(Dezember, 31)

Kollisionen bei $j = 0, 4, 6, 8$.

Definition 5.1.1

Eine Funktion $h\colon U \to [m]$ heißt eine **Hashfunktion**.

Wenn $S \subseteq U$ gegeben ist (oder ein $f\colon S \to R$), verteilt h die Schlüssel in S auf $[m]$.

$\boldsymbol{B_j} := \{x \in S \mid h(x) = j\}$ heißt **Behälter** („bucket“).

Idealfall: Die Einschränkung $h \restriction S$ ist **injektiv**, d. h. für jedes $j \in [m]$ existiert höchstens ein $x \in S$ mit $h(x) = j$.

Beispiel: Auf $S_1 = \{\texttt{Februar}^{(1)}, \texttt{Maerz}^{(4)}, \texttt{Mai}^{(8)}, \texttt{Juni}^{(0)}, \texttt{Juli}^{(11)}, \texttt{Dezember}^{(12)}\}$ ist die Funktion h von oben injektiv,
aber nicht auf $S_2 = \{\texttt{Januar}^{(0)}, \texttt{Mai}^{(8)}, \texttt{Juli}^{(11)}, \texttt{November}^{(8)}\}$.

Sprechweise: h **perfekt für** S $:\Leftrightarrow$ $h \restriction S$ injektiv.

Falls h perfekt für S:
Speichere x bzw. (x, r) in Tabellenplatz $\texttt{T}[h(x)]$, für $x \in S$.

lookup(x):

- Berechne $j = h(x)$.
  (Dies sollte **effizient** möglich sein.)
- Prüfe, ob in $\texttt{T}[j]$ Schlüssel x steht.
  Falls ja: Inhalt (x, r) von $\texttt{T}[j]$ ausgeben, sonst „$x \notin S$“.

Kollision: $x, y \in S$, $x \neq y$, $h(x) = h(y)$.

Leider sind Kollisionen „praktisch unvermeidlich“.

(Ausnahme: Speziell auf S abgestimmte Hashfunktion h,
fortgeschrittene Konstruktionen.)

Fall: h „rein zufällig“, $|S| = n$, $S = \{x_1, \ldots, x_n\}$. D. h.:

Uniformitätsannahme (UF*):

> Werte $h(x_1), \ldots, h(x_n)$ in $[m]^n$ sind „rein zufällig“:
> Jeder der m^n Vektoren $(j_1, \ldots, j_n) \in [m]^n$
> kommt mit derselben Wahrscheinlichkeit $1/m^n$
> als $(h(x_1), \ldots, h(x_n))$ vor.

(UF*) tritt z. B. (näherungsweise) ein, wenn

(i) U endlich ist und

(ii) h das Universum in gleich große Teile zerlegt, d. h.[*]

$$\left|h^{-1}(\{j\})\right| = \lfloor |U|/m \rfloor \quad \text{oder} \quad = \lceil |U|/m \rceil, \text{ für } j = 0, \ldots, m-1, \quad \text{und}$$

(iii) $S \subseteq U$ eine rein zufällige Menge der Größe n ist. (**(!) Sehr idealisierend!**)

[*] $\lfloor y \rfloor$ (bzw. $\lceil y \rceil$): größte (kleinste) ganze Zahl $\leq$ (bzw. $\geq$) y.

Mit (UF*):

$$\textbf{Pr}(\text{keine Kollision}) = \textbf{Pr}(h \text{ ist injektiv auf } \{x_1, \ldots, x_n\}) =$$
$$= \frac{m}{m} \cdot \frac{m-1}{m} \cdot \frac{m-2}{m} \cdot \ldots \cdot \frac{m-n+1}{m} = \ldots$$

(Für $i = 1, \ldots, n$ ist die Wahrscheinlichkeit, dass $h(x_i)$ in $[m] - \{h(x_1), \ldots, h(x_{i-1})\}$ liegt, genau $\frac{m-i+1}{m}$.)

$$\begin{aligned} \ldots &= 1 \cdot \left(1 - \frac{1}{m}\right) \cdot \left(1 - \frac{2}{m}\right) \cdot \ldots \cdot \left(1 - \frac{n-1}{m}\right) \\ &< e^0 \cdot e^{-\frac{1}{m}} \cdot e^{-\frac{2}{m}} \cdot \ldots \cdot e^{-\frac{n-1}{m}} = e^{-\frac{n(n-1)}{2m}}. \end{aligned}$$

(denn für $x \neq 0$ ist $1 + x < e^x$), also . . .

$$\boxed{\textbf{Pr}(h \text{ injektiv auf } S) < e^{-\frac{n(n-1)}{2m}}.}$$

Wunsch: Linearer Platzverbrauch, also

$$\text{„}\textbf{Auslastungsfaktor}\text{“} \quad \alpha := \frac{n}{m}$$

nicht kleiner als eine Konstante α_0, z. B. $\alpha \geq \frac{1}{2}$ stets. Dann:

$$\textbf{Pr}(h \text{ injektiv auf } S) \leq e^{-\alpha_0(n-1)/2}.$$

Winzig für nicht ganz kleine n! D. h.: Falls h nicht auf S abgestimmt ist, kommen Kollisionen praktisch immer vor.

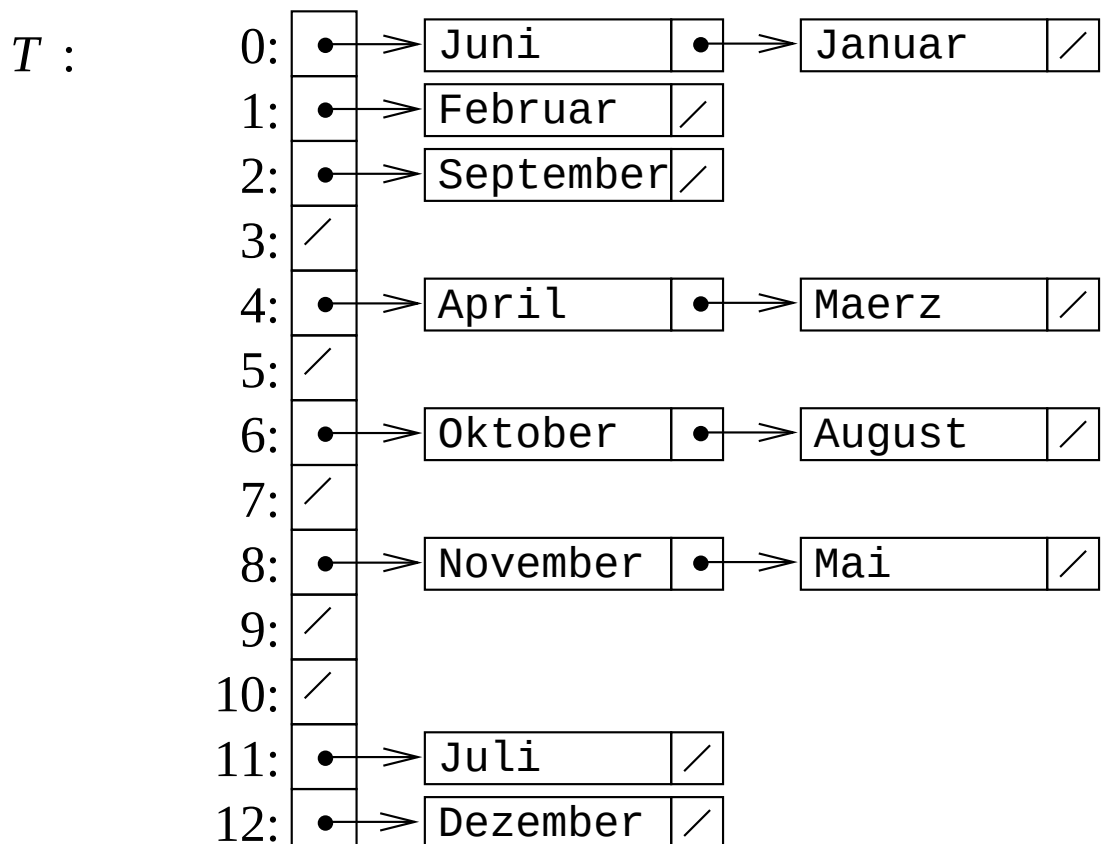
$\rightarrow$ **Kollisionsbehandlung** ist notwendig.

Vorlesungsvideo:

Hashing mit verketteten Listen

5.2 Hashing „mit verketteten Listen“

Beispiel:



5.2 Hashing „mit verketteten Listen“

engl.: **chained hashing**

Auch: **„offenes Hashing“**, weil man Platz außerhalb von T benutzt.

In $\texttt{T}[0\,..\,m-1]$: **Zeiger** auf die Anfänge
von m **einfach verketteten linearen Listen** $L_0, \ldots, L_{m-1}$.

(Oder auch: Zeiger auf „kleine“ Arrays/Vektoren mit variabler Länge.)

Liste L_j enthält die Einträge mit Schlüsseln x aus Behälter/Bucket

$$\boldsymbol{B_j} = \{x \in S \mid h(x) = j\}.$$

Bemerkung: Hier, als Unterstruktur, sind lineare Listen perfekt geeignet!

Wir implementieren die Wörterbuchoperationen: **Algorithmen(skizzen)**.

empty(m): Erzeugt Array `T[0..`$m-1$`]` mit m NULL-Zeigern/ NULL-Referenzen **und legt $h: U \to [m]$ fest**.

Kosten: $\Theta(m)$.

Beachte: Man wählt h, ohne S zu kennen!

lookup(x): Berechne $j = h(x)$;
suche Schlüssel x durch Durchlaufen der Liste L_j bei `T[`j`]`;
falls Eintrag (x, r) gefunden: **return** r;
// **erfolgreiche Suche**
sonst: **return** „*nicht vorhanden*“
// **erfolglose Suche**

Kosten/Zeit: $O(1 +$ Zahl der Schlüsselvergleiche „$x = y$?“$)$

. . . falls die Auswertung von $h(x)$
und ein Schlüsselvergleich Zeit $O(1)$ kostet.

Erfolglose Suche: Exakt $|B_j|$ Schlüsselvergleiche.

Erfolgreiche Suche: Höchstens $|B_j|$ Schlüsselvergleiche.

insert(x, r): Berechne $j = h(x)$;
suche Schlüssel x in Liste L_j bei `T[`j`]`;
falls gefunden: ersetze Daten bei x durch r;
// **erfolgreiche Suche**
sonst: füge (x, r) in Liste `T[`j`]` ein.
// **erfolglose Suche, Normalfall!**

delete(x): Berechne $j = h(x)$;
suche Schlüssel x in Liste L_j bei `T[`j`]`;
falls gefunden: lösche Eintrag $(x, f(x))$ aus L_j;
// **erfolgreiche Suche, Normalfall!**
sonst: tue nichts. // **erfolglose Suche**

Kosten jeweils: wie bei *lookup*.

Ziel: Analysiere **erwartete Kosten** für *lookup*, *insert*, *delete*.

Hierfür genügt: Analysiere **erwartete Anzahl der Schlüsselvergleiche**.

Vorausgesetzt: Irgendeine Quelle für **„Zufall“**. (Dazu später mehr.)

Pr(A): Wahrscheinlichkeit für **Ereignis** A.

E(X): Erwartungswert der **Zufallsvariablen** X.

Abgeschwächte „Uniformitätsannahme ($\mathbf{UF}_2$)“:

> Für je 2 Schlüssel $x \neq z$ in U gilt: $\mathbf{Pr}(h(x) = h(z)) \leq \frac{1}{m}$.

Später: Wann ist diese Annahme erfüllt?
Wenn die Hashwerte rein zufällig sind, gilt: $\mathbf{Pr}(h(x) = h(z)) = \frac{1}{m}$.

Szenario: Menge $S = \{x_1, \ldots, x_n\}$ von n Schlüsseln in Struktur gespeichert. Nun wird nach $y \in U$ gesucht.

$\alpha = \frac{n}{m}$: der (aktuelle) **Auslastungsfaktor**.

Erfolglose Suche: $y \notin S$.

Definiere **Zufallsgrößen/Zufallsvariable**, für alle $x, z \in U$:

$$Y_{x,z} := \begin{cases} 1\ , & \text{falls } h(x) = h(z), \\ 0\ , & \text{falls } h(x) \neq h(z). \end{cases}$$

Nach (UF_2): $\mathbf{E}(Y_{x,z}) = \mathbf{Pr}(h(x) = h(z)) \leq \frac{1}{m}$.

Anzahl der Schlüsselvergleiche/Listenlänge $=$

$$|B_{h(y)}| = |\{x \in S \mid h(x) = h(y)\}| = \sum\nolimits_{x \in S} Y_{x,y}.$$

Eine Zufallsvariable!

Mit „Linearität des Erwartungswertes“:

$$\mathbf{E}(|B_{h(y)}|) = \sum_{x \in S} \mathbf{E}(Y_{x,y}) \overset{(\mathsf{UF}_2)}{\leq} \sum_{x \in S} \frac{1}{m} = \frac{n}{m} = \alpha.$$

Satz 5.2.3

Bei Hashing mit verketteten Listen gilt unter der Annahme (UF_2):

Die **erwartete Anzahl** von Schlüsselvergleichen bei **erfolgloser** Suche ist höchstens α, bei **erfolgreicher** Suche höchstens $1 + (n-1)/m < 1 + \alpha$.
(Analog; Summand $\mathbf{E}(Y_{y,y}) = 1$.)

Die **erwartete Zeit** für eine erfolglose/erfolgreiche Suche ist $O(1 + \alpha)$.

Fazit: Wenn $\alpha \leq \alpha_1$, für α_1 konstant (z.B. $\alpha_1 = 2$), dann ist die **erwartete Zeit** für eine Suche $O(1)$.

Verdoppelungsstrategie

. . . hilft, die Bedingung $\alpha \leq \alpha_1$ zu erzwingen, auch wenn anfangs die Schlüsselanzahl unbekannt ist.

(Analog zu Verdoppelungsstrategie bei Stacks, Kap. 2)

empty(m): Anfangs-Tabellengröße m oder
empty$()$: Anfangs-Tabellengröße m_0 (fester Wert)

Variable `load` enthält aktuelle Anzahl von Schlüsseln.

empty(m):	$\texttt{load} \leftarrow 0$,
insert(x, r) mit x neu:	$\texttt{load} \leftarrow \texttt{load} + 1$,
delete(x) mit x vorhanden:	$\texttt{load} \leftarrow \texttt{load} - 1$.

Verdoppelungsstrategie (Fortsetzung)

Wenn nach *insert*(x, r) (mit x neu) der Wert `load` die Grenze $\alpha_1 m$ überschreitet:

- Erzeuge **neue Tabelle** $\mathtt{T}_{\text{neu}}$ der Größe $2m$.
- Erzeuge **neue Hashfunktion** $h_{\text{neu}} : U \to [2m]$.
- Übertrage Einträge aus den alten Listen
  in neue Listen zu Tabelle $\mathtt{T}_{\text{neu}}$, gemäß neuer Hashfunktion.
  Dabei Einfügen **am Listenanfang ohne Suchen**.
  (Wissen: Alle Schlüssel sind verschieden!)
  Kosten: $O(m)$ im schlechtesten Fall.
- Benenne $\mathtt{T}_{\text{neu}}$ in `T` und h_{neu} in h um.

Satz 5.2.4

Sei α_1 fest. Wenn die verwendeten Hashfunktionen (UF_2) erfüllen, dann gilt für das skizzierte Verfahren für Hashing mit Listen mit Verdoppelungsstrategie:

(a) Die erwartete Zeit für erfolglose bzw. erfolgreiche Suche ist $O(1 + \alpha_1)$.

(b) Die Gesamtkosten für sämtliche Verdopplungen sind durch $O(\tilde{n})$ (schlechtester Fall) beschränkt; dabei ist $\tilde{n}$ die maximale Anzahl der jemals gleichzeitig in der Datenstruktur vorhandenen Schlüssel (maximaler Wert in `load`).

(c) Der Gesamtplatzbedarf ist $O(\tilde{n})$.

(Zu (b): **Analyse** der Verdoppelungsstrategie wie bei Stacks.)

Vorlesungsvideo:

Hashfunktionen

5.3 Hashfunktionen

5.3.1 Einfache Hashfunktionen

Kriterien für Qualität:

1) **Zeit für Auswertung** von $h(x)$ (*Schnell*!)

2) **Wahrscheinlichkeit** für **Kollisionen**
(unter „realistischen Annahmen“ über Schlüsselmenge S).
(*Klein*, am besten $\leq c/m$ für c konstant.)

Für 1) entscheidend: **Schlüsselformat**.

1. Fall: Schlüssel sind natürliche Zahlen: $U \subseteq \mathbb{N}$.

Divisionsrestmethode: $\boxed{h(x) = x \bmod m}$

Sehr einfach, recht effizient.

Nachteile:

- Unflexibel: Nur eine feste Funktion für eine Tabellengröße.
- Bei strukturierten Schlüsselmengen (z. B. aus Strings übersetzte Schlüssel) deutlich erhöhtes Kollisionsrisiko.

Für Produktionssysteme darf diese Methode nicht benutzt werden!

Multiplikationsmethode: $U \subseteq \mathbb{N}$. *Idee:* Für $0 < \vartheta < 1$ (idealisiert: ϑ **irrational**)

$h(x) := \lfloor ((\vartheta \cdot x) \bmod 1) \cdot m \rfloor$. (Dabei: $(\vartheta \cdot x) \bmod 1 = \vartheta \cdot x - \lfloor \vartheta \cdot x \rfloor \in [0, 1)$.)

(Gebrochener Anteil von ϑx wird auf Wertebereich $[0, m)$ „aufgeblasen“, dann auf ganze Zahl abgerundet. Für $\theta = \frac{1}{2}(\sqrt{5} - 1) \approx 1{,}618$ weiß man: $x, x + 1, \ldots, x + n - 1$ werden gut verteilt.)

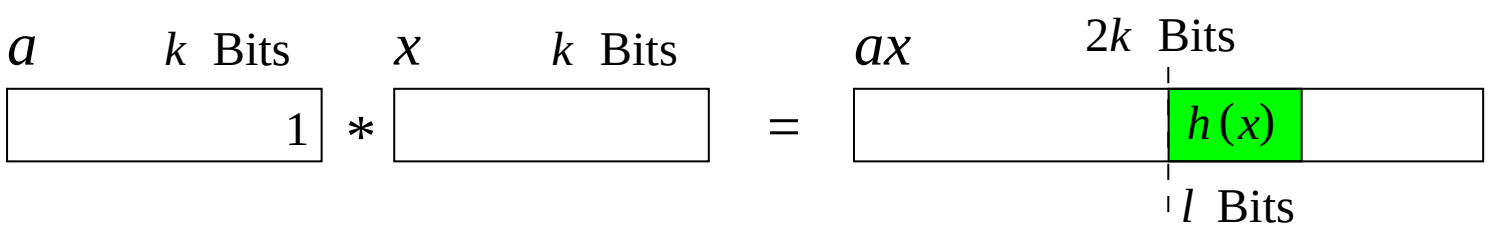
Diskrete Multiplikationsmethode: $U = [2^k]$ und $m = 2^l$.

$$h_a(x) = (ax \bmod 2^k) \operatorname{div} 2^{k-l}.$$

mit $a \in [2^k]$ ungerade und z. B. $a/2^k \approx \frac{1}{2}\left(\sqrt{5} - 1\right) = 0{,}618\ldots$.

Oder: $a \in [2^k]$ ungerade, **zufällig**.

Sehr effiziente Auswertung ohne Division:



(Wenn $k =$ Wortbreite w: Multiplikation + Shift.)

2. Fall: Schlüssel sind Strings/Wörter: $U \subseteq \text{Seq}(\Sigma)$

Σ: **„Alphabet“** – endliche nichtleere Menge.

$\mathbf{Seq(\Sigma)}$: Menge aller endlichen Folgen von Elementen von Σ: „Wörter über Σ“.

Beispiel: $\Sigma =$ ISO-8859-1-Alphabet $\widehat{=} \{0,1\}^8 \widehat{=} \{0,1,\ldots,255\}$.

(Siehe `https://de.wikipedia.org/wiki/ISO_8859-1#ISO/IEC_8859-1`.)

Schlüssel: $x = (c_1, \ldots, c_r) = c_1 \ldots c_r$ mit $c_1, \ldots, c_r \in \Sigma$.

Möglich, aber nicht zu empfehlen:
Transformation von Schlüssel x in Zahl $\hat{x} = n(x)$ als neuen Schlüssel, dann: $x \mapsto h(\hat{x})$ für Hashfunktion h auf Zahlen.

- Kollisionen $n(x) = n(y)$ nie auflösbar!
- Selbst wenn n injektiv: Ineffiziente Auswertung von $h(\hat{x})$.

Besser: An Schlüsselformat $x = (c_1, \ldots, c_r)$ angepasste Hashfunktionen.

Lineare Funktionen über Körper $\mathbb{Z}_p = [p] = \{0, \ldots, p-1\}$:

$\Sigma \subseteq [p]$ für p Primzahl, $m = p$. (Beispiel: ISO-8859-1 $\widehat{=}\ [256] \subseteq [p] = [1009]$.)

Wähle Koeffizienten $a_1, \ldots, a_r \in [p]$.

$$h(c_1 \cdots c_r) = \left(\sum_{i=1}^{r} a_i \cdot c_i \right) \bmod p.$$

Vorteile:

- Rechnung nur mit Zahlen $< p^2 = m^2$ . . . (wenn man nach jeder Multiplikation/Addition Rest modulo p bildet);
- sehr effizient;
- theoretisch nachweisbares gutes Verhalten, wenn $a_1, \ldots, a_r$ aus $[p]$ **zufällig** gewählt werden.

5.3.2 Anspruchsvollere Hashfunktionen

Qualitätskriterien:

1) Zeit für **Auswertung** von $h(x)$ (*klein*!)
2) geringe **Wahrscheinlichkeit** für **Kollisionen**
   ohne Annahmen über die Schlüsselmenge S (ähnlich der zufälligen Situation!)
3) Effizientes **Generieren** einer neuen Hashfunktion
   (benötigt z. B. bei der Verdoppelungsstrategie)
4) **Platzbedarf** für die Speicherung von h (Programmtext, Parameter)
   (*gering* gegenüber n!)

Universelles Hashing (Idee): Bei der Festlegung der Hashfunktion h wird (kontrolliert) **Zufall** eingesetzt („Randomisierter Algorithmus“).
(Bei Programmierung: „Pseudozufallszahlengenerator“, in jeder Prog.-Sprache bereitgestellt.)
Analyse beruht auf dadurch **vom Algorithmus erzwungenen** Zufallseigenschaften von h, nicht auf Annahmen über S.

Beispiel: Alphabet $\Sigma = [s]$, $U \subseteq [s]^r$, und $m = p \geq s$ eine **Primzahl**.

[*Konkret*: $U = (\text{ISO 8859-1})^{20} = \{0, \ldots, 255\}^{20}$, $s = 256$, $m = p = 1009$.]

$a_1, \ldots, a_r$ seien (uniform) **zufällig gewählt** aus $[p]$. (In *empty*(p).)

Definiere

$$h(\underbrace{c_1 \cdots c_r}_{x}) = (a_1 \cdot c_1 + \cdots + a_r \cdot c_r) \bmod p.$$

Proposition 5.3.1

In der beschriebenen Situation gilt für $x, z \in U$ mit $x \neq z$:

$$\mathbf{Pr}(h(x) = h(z)) = \frac{1}{p}.$$

(Zufall in der Wahrscheinlichkeit stammt aus dem Algorithmus.) **$(\mathbf{UF_2})$ ist erfüllt!**

Beweis: Sei $x = (c_1, \ldots, c_r)$, $z = (d_1, \ldots, d_r)$.

Rechne im **Körper** $\mathbb{Z}_p = \mathbb{Z}/p\mathbb{Z}$ „=“ $[p]$ (also modulo p).

Es gibt p^r viele Möglichkeiten für $a = (a_1, \ldots, a_r)$.

Wie viele davon erfüllen
$a_1c_1 + \ldots + a_rc_r = a_1d_1 + \ldots + a_rd_r$, d. h. $a_1(c_1 - d_1) + \ldots + a_r(c_r - d_r) = 0$?

Beispielsweise sei $c_r - d_r \neq 0$.

Für jede Wahl von $a_1, \ldots, a_{r-1}$ gibt es genau ein passendes a_r, nämlich

$$a_r = -(a_1(c_1 - d_1) + \ldots + a_{r-1}(c_{r-1} - d_{r-1})) \cdot (c_r - d_r)^{-1}.$$

(Wir rechnen modulo p, mit Inversen im Körper $\mathbb{Z}_p$.) Also ist die Wahrscheinlichkeit, einen passenden Vektor $(a_1, \ldots, a_r)$ zu wählen, genau $\dfrac{p^{r-1}}{p^r} = \dfrac{1}{p}$. □

Variante: $U \subseteq [p]$, p Primzahl.

Tabellengröße $m \leq p$ beliebig.

Wähle $\boldsymbol{a}$ aus $\{1, \ldots, p-1\}$ (uniform) **zufällig**.

Definiere

$$h(x) = ((\boldsymbol{a} \cdot x) \bmod p) \bmod m.$$

Proposition 5.3.2

In dieser Situation gilt für $x, z \in U$ mit $x \neq z$:

$$\mathbf{Pr}(h(x) = h(z)) \leq \frac{2}{m}.$$

(D. h. eine Abschwächung von (UF_2) ist erfüllt.)

Beweis (nicht prüfungsrelevant): Druckfolien.

Beweis **(nicht prüfungsrelevant)**: Für $a \in \{1, \ldots, p-1\}$ setze:

$$h_a(x) := (ax \bmod p) \bmod m, \text{ für } x \in U.$$

Seien $x, z \in U$ verschieden mit $h_a(x) = h_a(z)$. Setze $y := x - z$.

1. Fall: $ax \bmod p > az \bmod p$.
Dann ist $ax \bmod p - az \bmod p = ay \bmod p \in \{1, \ldots, p-1\}$ durch m teilbar.
2. Fall: $ax \bmod p < az \bmod p$.
Dann ist $ax \bmod p - az \bmod p = ay \bmod p - p \in \{-1, \ldots, -(p-1)\}$ durch m teilbar.

Also gilt: $ay \bmod p \in \{i \cdot m \mid 1 \leq i \leq \frac{p-1}{m}\} \cup \{p - i \cdot m \mid 1 \leq i \leq \frac{p-1}{m}\}$.
Weil $y \neq 0$, ist $ay \bmod p$ uniform zufällig in $\{1, \ldots, p-1\}$ verteilt. – Also:

$$\begin{aligned}\mathbf{Pr}(h_a(x) = h_a(z)) &\leq \mathbf{Pr}\big(ay \bmod p \in \{i \cdot m \mid 1 \leq i \leq \tfrac{p-1}{m}\} \cup \{p - i \cdot m \mid 1 \leq i \leq \tfrac{p-1}{m}\}\big) \\ &\leq \frac{2(p-1)/m}{p-1} = \frac{2}{m}.\end{aligned}$$

□

Proposition 5.3.3

Wenn man beim **multiplikativen Hashing** (wobei $U = [2^k]$, $m = 2^l$), mit

$$h(x) = (\boldsymbol{a} \cdot x \bmod 2^k) \operatorname{div} 2^{k-l}$$

als $\boldsymbol{a}$ eine **zufällige** **ungerade** Zahl in $[2^k]$ wählt, dann gilt für beliebige $x, z \in U$, $x \neq z$:

$$\mathbf{Pr}(h(x) = h(z)) \leq \frac{2}{2^l} = \frac{2}{m}.$$

D. h.: Eine Abschwächung von (UF_2) ist erfüllt.

(*Beweis*: Originalliteratur [M. D., T.Hagerup, J.Katajainen, M.Penttonen, A Reliable Randomized Algorithm for the Closest-Pair Problem, J. Algorithms **25**(1): 19-51 (1997)].)

Tabellenhashing [Zobrist, 1970]

Universum: $U = \Sigma^r$, mit $\Sigma = \{0, \ldots, s-1\}$.

$m = 2^l$ mit $l \leq w$ (w: Wortlänge, z. B. $w = 32$).

Wertebereich: $[m]$, entspricht $\{0,1\}^l$ (via Binärdarstellung!).

Repräsentation der Hashfunktion h:

Array $\mathtt{A}[1\,..\,r\,, 0\,..\,s-1]$ mit Einträgen aus $\{0,1\}^w$.

$h(c_1 \cdots c_r) =$ die ersten l Bits von $\mathtt{A}[1, c_1] \oplus \cdots \oplus \mathtt{A}[r, c_r]$

Dabei: $\oplus$ ist bitweises XOR, auf ganzes Wort angewendet.
(Maschinenoperation – effizient!)

Beispiel: $\Sigma = \{0, 1, \ldots, 7\}$, $r = 6$ (1 Schlüssel = 6 Oktalziffern), $w = 4$, $l = 3$.

Array `A`:

	Buchstaben in Σ (Oktalziffern)							
	0	1	2	3	4	5	6	7
Position $i = 1$	0010	1001	0011	1010	0111	0000	1101	0111
2	0011	1000	1111	0100	1001	1101	0110	0110
3	1000	0110	0101	1001	1110	1111	0000	1110
4	1101	0100	1110	0111	1101	0100	1010	0110
5	1100	1001	1100	1011	0101	1011	1110	0000
6	0110	1101	1001	1111	1000	1110	0011	0111

$h(3\,5\,3\,7\,4\,3)$
$=$ die ersten 3 Bits von $1010 \oplus 1101 \oplus 1001 \oplus 0110 \oplus 0101 \oplus 1111$
$=$ die ersten 3 Bits von $0010 = 001$.

Nachteil: Array $\texttt{A}[1\,..\,r\,,0\,..\,s-1]$ benötigt **Platz** $r \cdot s$ (Wörter).

Beispiel: $r = 20$ ISO-8859-1-Zeichen pro Wort, $\Sigma = \{0,1\}^8$, $w = 32$.
$\texttt{A}[1\,..\,r\,,0\,..\,s-1]$ benötigt
$rs(w/8) = 20 \cdot 256 \cdot 4 = 20480$ Bytes, d. h. 20 KB.
Sinnvoll nur für **große** Hashtabellen.

Platzsparend, etwas langsamer:
$\Sigma = \{0,1\}^4 \mathrel{\widehat{=}} \{\texttt{0}, \texttt{1}, \ldots, \texttt{9}, \texttt{A}, \ldots, \texttt{F}\}$. („Tetraden“ bzw. Hexadezimalziffern.)

Platz im Beispiel: $rs(w/8) = 40 \cdot 16 \cdot 4 = 2560$ Bytes.

Vorteile von Tabellenhashing:

- **Extrem schnelle Auswertung** ($\oplus$ ebenso schnell wie $+$).
- Dasselbe `A` kann für $m = 2^4, 2^5, 2^6, \ldots, 2^w$ benutzt werden.

  Ideal für die Kombination mit der Verdoppelungsstrategie: Verdopple Tabellengröße $\widehat{=}$ erhöhe l um 1; Einträge in `A` bleiben unverändert.
- Beweisbar **gutes Verhalten**, wenn Einträge von `A` zufällig.

Proposition 5.3.4

Wenn die Einträge in Array `A[1..r,0..s-1]` uniform **zufällig** gewählt werden, dann gilt für $x, y, z \in U$ mit x, y, z verschieden und $a, b, c \in \{0,1\}^l$ beliebig:

$$\mathbf{Pr}(h(x) = a \wedge h(y) = b \wedge h(z) = c) = \frac{1}{m^3}.$$

Insbesondere gilt (UF_2).

Beweis: Übung.

Leichte Modifikation des Beweises von Satz 5.2.3. liefert:

Proposition 5.3.5

Sei c konstant. Wenn $\mathbf{Pr}(h(x) = h(z)) \leq \frac{c}{m}$, für beliebige $x, z \in U$, $x \neq z$, und wenn wir **Hashing mit verketteten Listen** benutzen, dann ist die **erwartete** Zahl von Schlüsselvergleichen

- höchstens $c \cdot \alpha$ bei der **erfolglosen Suche** und
- höchstens $1 + c \cdot \alpha$ bei der **erfolgreichen Suche**.

Wenn $\alpha \leq \alpha_1$ sichergestellt ist, für α_1 konstant (Verdoppelungsstrategie):
Erwartete Suchzeit ist $\boldsymbol{O(1)}$.

Stichwort: „Universelles Hashing“: siehe Druckfolien.

Universelles Hashing (Formales Konzept)

SS 2021: Nicht prüfungsrelevant.

Eine Menge $\mathcal{H}$ von Hashfunktionen, d.h. $\mathcal{H} \subseteq \{h \mid h\colon U \to [m]\}$,

heißt **c-universelle** Klasse von Hashfunktionen,

wenn für beliebige $x, z \in U$ mit $x \neq z$ gilt:

$$\frac{|\{h \in \mathcal{H} \mid h(x) = h(z)\}|}{|\mathcal{H}|} \leq \frac{c}{m}.$$

In anderen Worten:

. . . wenn für für beliebige $x, z \in U$ mit $x \neq z$ und für zufällig gewähltes h aus $\mathcal{H}$ gilt:

$$\mathbf{Pr}(h(x) = h(z)) \leq \frac{c}{m}.$$

Beispiele:

- $U \subseteq [p]$, p Primzahl, Tabellengröße $m \leq p$ beliebig.

$$h_a(x) = ((a \cdot x) \bmod p) \bmod m, \text{ für } a \in \{1, \ldots p-1\}.$$

  Dann ist $\boldsymbol{\mathcal{H}_{p,m}} = \{h_a \mid 1 \leq a < p\}$ eine 2-universelle Klasse.

- $U = (\text{ISO 8859-1})^r \mathrel{\widehat{=}} \{0, \ldots, 255\}^r$, $m = p > 256$ Primzahl.

  $h_{a_1,\ldots,a_r}(c_1 \ldots c_r) = \left(\sum_{i=1}^{r} a_i c_i \right) \bmod p$, für $c_1 \ldots c_r \in U$.

  Dann ist $\boldsymbol{\mathcal{H}_{p,r}} = \{h_{a_1,\ldots,a_r} \mid a_1, \ldots, a_r \in [p]\}$ eine 1-universelle Klasse.

Beispiele:

- **Multiplikatives Hashing** mit zufälligem (ungeradem) Faktor a führt zu einer 2-universellen Klasse.
- **Tabellenhashing** mit zufälligen Tabelleneinträgen führt zu einer 1-universellen Klasse.

Die obige **Proposition 5.3.5** liest sich nun so:

Wenn $\mathcal{H}$ c-universell ist, wenn man Hashing mit verketteten Listen benutzt und (in *empty*(m)) ein h aus $\mathcal{H}$ **zufällig** wählt,

dann gelten die oberen Schranken $\boldsymbol{c \cdot \alpha}$ bzw. $\boldsymbol{1 + c \cdot \alpha}$ für die erwartete Anzahl der Schlüsselvergleiche bei **erfolgloser** bzw. **erfolgreicher** Suche.

Vorlesungsvideo:

Geschlossenes Hashing mit Sondierungsstrategien

5.4 Geschlossenes Hashing

oder **„offene Adressierung“**: Kollisionsbehandlungs-Methode.

Idee: Für jeden Schlüssel $x \in U$ gibt es eine
Sondierungsfolge $h(x,0), h(x,1), h(x,2), \ldots$ in $[m]$.

Beim **Einfügen** von x und **Suchen** nach x werden die Zellen von `T` in dieser Reihenfolge untersucht, bis eine leere Zelle oder der Eintrag x gefunden wird. – Günstig:

(∗) $h(x,0), h(x,1), \ldots, h(x,m-1)$ erreicht jede Zelle, d. h.:
$(h(x,0), h(x,1), h(x,2), \ldots, h(x,m-1))$
ist eine **Permutation** von $[m]$.

Wenn (∗) gilt und mindestens eine Zelle in `T` leer ist, dann endet jede erfolglose Suche in einer leeren Zelle.

A – Lineares Sondieren $h(x,k) = (h(x) + k) \bmod m,\ k = 0, 1, 2, \ldots$

Offensichtlich: $(*)$ erfüllt.

Beispiel:

Nächste Tabelle: „Wunschplätze“ für die Schlüssel.

Übernächste Tabelle: Sequenzielles Einfügen in der natürlichen Reihenfolge gemäß „Linearem Sondieren“.

$h(c_1 \dots c_r) = (\mathsf{num}(c_3) + 11) \bmod 13$

j	x
0	$\texttt{September}^{(0)}$
1	
2	$\texttt{Maerz}^{(2)}$ $\texttt{April}^{(2)}$
3	
4	$\texttt{August}^{(4)}$ $\texttt{Oktober}^{(4)}$
5	
6	$\texttt{Mai}^{(6)}$ $\texttt{November}^{(6)}$
7	
8	
9	$\texttt{Juli}^{(9)}$
10	$\texttt{Dezember}^{(10)}$
11	$\texttt{Januar}^{(11)}$ $\texttt{Juni}^{(11)}$
12	$\texttt{Februar}^{(12)}$

Tabelle `T`$[0..12]$:

Platz	Eintrag
0	`Juni`$^{(11)}$
1	`Maerz`$^{(1)}$
2	`April`$^{(1)}$
3	`September`$^{(0)}$
4	`August`$^{(4)}$
5	
6	`Mai`$^{(6)}$
7	
8	
9	`Juli`$^{(9)}$
10	
11	`Januar`$^{(11)}$
12	`Februar`$^{(12)}$

Tabelle `T`$[0..12]$:

Platz	Eintrag
0	`Juni`$^{(11)}$
1	`Maerz`$^{(1)}$
2	`April`$^{(1)}$
3	`September`$^{(0)}$
4	`August`$^{(4)}$
5	`Oktober`$^{(4)}$
6	`Mai`$^{(6)}$
7	`November`$^{(6)}$
8	
9	`Juli`$^{(9)}$
10	`Dezember`$^{(10)}$
11	`Januar`$^{(11)}$
12	`Februar`$^{(12)}$

empty(m): Lege leeres Array `T`$[0\,..\,m-1]$ an.

// Arrayeintrag: (Schlüssel,Wert (in R)) oder *leer*.

Wähle Hashfunktion h.

`T`$[j] \leftarrow$ *leer*, für $j \in [m]$.

`load` $\leftarrow 0$, initialisiere `maxload` mit Wert $< m$.

insert(x, r): $j \leftarrow h(x)$;

Finde erstes l in der Folge j, $(j+1) \bmod m$, $(j+2) \bmod m$, ...

mit `T`$[l]$`.key` $= x$ **oder** `T`$[l] =$ *leer*.

Im Fall `T`$[l]$`.key` $= x$: // **erfolgreiche Suche**, Update-Situation

`T`$[l]$`.data` $\leftarrow r$;

Im Fall `T`$[l] =$ *leer*: // **erfolglose Suche**, neuer Eintrag

Falls `load` $=$ `maxload`: „Überlaufbehandlung“

// z. B. Übertragen der Einträge von `T` in neue, doppelt so große Tabelle

// Neustart von **insert**(x, r)

Sonst: `T`$[l] \leftarrow (x, r)$; `load++`.

lookup(x): $j \leftarrow h(x)$;
Finde erstes l in der Folge $h(x, j)$, $j = 0, 1, 2, \ldots$, mit `T[`l`].key` $= x$ **oder** `T[`l`]` $=$ *leer*.
Im Fall `T[`l`].key` $= x$: **return** `T[`l`].data`; // **erfolgreiche Suche**
Im Fall `T[`l`]` $=$ *leer*: **return** „*nicht vorhanden*“. // **erfolglose Suche**

Um die **Korrektheit** (des Ein-/Ausgabeverhaltens) zu zeigen, betrachten wir eine beliebige Operationenfolge und zeigen die folgende Invariante, für jeden Schlüssel x:

Invariante: Solange x nicht eingefügt wurde, gibt es in der Tabelle keinen Eintrag mit Schlüssel x. Nachdem x eingefügt wurde, gilt: Es gibt genau einen solchen Eintrag, in einer Zelle `T[`i`]`. Diese enthält immer den Datensatz r von der *letzten* Operation **insert**(x, r). Wenn $i = h(x, l)$, enthalten alle Zellen `T[`$h(x, j)$`]`, $0 \leq j < l$, einen Schlüssel $\neq x$. (Beweis durch Induktion über die ausgeführten Einfügungen, auf den Druckfolien.)

Korrektheit der Suche: Aus der Invarianten folgt sofort durch Betrachten der Algorithmen, dass **lookup**(x) einen Eintrag mit Schlüssel x genau dann findet, wenn es einen gibt, und dass dieser immer den Datensatz r enthält, der von der neuesten Operation **insert**(x, r) herrührt. (Details: Druckfolien.)

Beweis der Invarianten: Dass es keinen Eintrag mit Schlüssel x gibt, wenn x noch nicht eingefügt wurde, ist klar. Da Schlüssel nie gelöscht werden, sondern nur Datensätze überschrieben werden, bleibt ein einmal generierter Eintrag mit Schlüssel x immer erhalten.
Für den zweiten Teil benutzen wir Induktion. Betrachte die erste Einfügeoperation **insert**(x, r) für x. Die Suche nach x endet in einer leeren Zelle `T[`i`]`, mit $i = h(x, l)$, wobei alle Zellen `T[`$h(x, j)$`]`, $0 \leq j < l$, einen Schlüssel $\neq x$ enthalten. Hier wird (x, r) gespeichert. Wenn später in **insert**(x, r') erneut nach x gesucht wird, dann folgt aus der Invarianten und dem Suchverfahren, dass die eindeutig bestimmte Zelle `T[`i`]` mit Schlüssel x gefunden wird. Also aktualisiert **insert**(x, r') den richtigen Datensatz (und erzeugt kein Duplikat).

Zur **Korrektheit der Suche**: Aus der Invarianten folgt, dass **lookup**(x) nur dann einen Datensatz zurückgibt, wenn x vorher eingefügt wurde. Wenn dies der Fall ist, stellt die Invariante sicher, dass **lookup**(x) die eindeutig bestimmte Zelle `T[`i`]` mit Schlüssel x findet, und dass diese den aktuellen Datensatz für diesen Schlüssel enthält.

Satz 5.4.1 (1962 bewiesen von Donald E. Knuth (*1938, TAOCP, TEX))

Bei linearem Sondieren in einer Tabelle der Größe m mit Schlüsselmenge $S = \{x_1, \ldots, x_n\}$ gilt, mit festem $\boldsymbol{\alpha = \frac{n}{m}}$ (und $n, m \to \infty$):

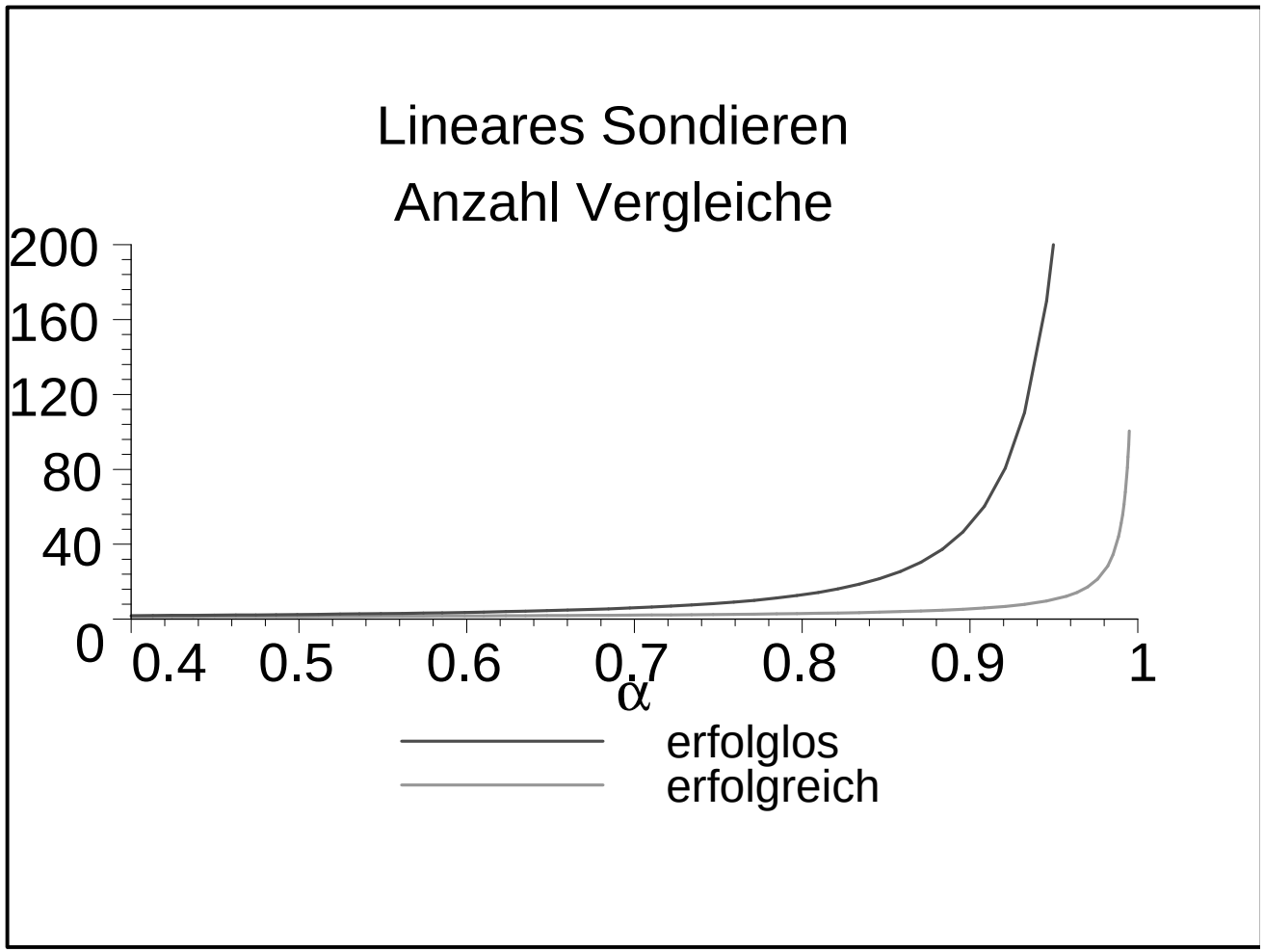
Die **erwartete** (nach (UF*): $h(x_1), \ldots, h(x_n)$ **rein zufällig**) Anzahl von Schlüsselvergleichen (besuchte Arrayzellen) bei **erfolgloser Suche** nach $y \notin S$ ist

$$\approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right).$$

Die **erwartete** (nach (UF*)) **mittlere** (über $x_1, \ldots, x_n$) Anzahl von Schlüsselvergleichen bei **erfolgreicher Suche** nach $y \in S$ ist

$$\approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right).$$

Der erwartete Aufwand ist also $\boldsymbol{O(1)}$, falls $\boldsymbol{\alpha \leq \alpha_0}$ für eine Konstante $\alpha_0 < 1$.


Lineares Sondieren
Anzahl Vergleiche
200
160
120
80
40
0
0.4
0.5
0.6
0.7
0.8
0.9
1
α
erfolglos
erfolgreich

Erwartete Anzahl von Schlüsselvergleichen bei linearem Sondieren:

	erfolglose Suche	erfolgreiche Suche (Mittel über $x_1, \ldots, x_n$)
α	$\frac{1}{2} \cdot (1 + \frac{1}{(1-\alpha)^2})$	$\frac{1}{2} \cdot (1 + \frac{1}{1-\alpha})$
0,5	2,5	1,5
0,6	3,625	1,75
0,7	6,06	2,16
0,75	8,5	2,5
0,8	13	3
0,9	**50,5**	5,5
0,95	**200,5**	**10,5**
0,98	**1250,5**	**25,5**

Ergebnisse für lineares Sondieren:

Eine einzelne erfolglose Suche wird z. B. bei zu 90% gefüllter Tabelle durchaus teuer.

Bei vielen erfolglosen Suchen in einer nicht veränderlichen Tabelle (anwendungsabhängig!) sind diese Kosten nicht tolerierbar.

$\alpha \leq 0{,}6$ oder $\alpha \leq 0{,}7$ liefert jedoch akzeptable Suchkosten.

Die (über $x_i \in S$) **gemittelte** Einfüge-/Suchzeit ist dagegen sogar bei 90% gefüllter Tabelle erträglich.

!!! Lineares Sondieren passt perfekt zu **Cache-Architekturen** und ist deswegen auch in der Praxis sehr schnell.
(Bei einem Cachefehler wird eine ganze „Zeile“ in den Cache geholt, also mehrere aufeinanderfolgende Wörter aus `T`.)

Verdoppelungsstrategie bei linearem Sondieren:

Verdopple die Tabelle, sobald der Auslastungsfaktor $\alpha = n/m$ einen Wert von α_0 übersteigt – z.B. $\alpha_0 = 0{,}75$ oder $\alpha_0 = 0{,}8$.

Empfehlung: Kombiniere lineares Sondieren mit einer guten universellen Hashklasse, z. B. Tabellenhashing.

Nicht auf die Zufälligkeit von S vertrauen!

Alternative: **MurmurHash3** (32-Bit- oder 128-Bit-Werte).

`https://en.wikipedia.org/wiki/MurmurHash`
Lässt „seed“ zu, d. h. unterschiedliche Funktionen.
Bisherige experimentelle Untersuchungen: schnelle Auswertung, gute Verteilung.
Jedoch: Keine mathematische Aussage bekannt.

Achtung! Kryptographische Hashfunktionen (z.B. MD4, MD5, SHA-1, SHA-3, . . .) haben andere Anforderungen und **viel höhere Auswertezeiten**!
Sie sind für Anwendungen mit Hashtabellen nicht geeignet.

Andere Sondierungsfolgen sind möglich, zum Beispiel . . .

B – Quadratisches Sondieren

$h\colon U \to [m]$ sei beliebige Hashfunktion.

$$
\begin{aligned}
h(x,0) &= h(x) \\
h(x,1) &= (h(x) + \mathbf{1}) \bmod m \\
h(x,2) &= (h(x) - \mathbf{1}) \bmod m \\
h(x,3) &= (h(x) + \mathbf{4}) \bmod m \\
h(x,4) &= (h(x) - \mathbf{4}) \bmod m \\
h(x,5) &= (h(x) + \mathbf{9}) \bmod m \\
h(x,6) &= (h(x) - \mathbf{9}) \bmod m \\
&\vdots
\end{aligned}
$$

Allgemein: $\boxed{h(x,k) = \left(h(x) + \lceil k/2 \rceil^2 \cdot (-1)^{k+1}\right) \bmod m, \text{ für } k = 0, 1, 2, \ldots.}$

(Man kann die Folge auch inkrementell nur mit Additionen/Subtraktionen modulo m berechnen.)

Beispiel: $m = 19$, $h(x) = 8$,
liefert Sondierungsfolge

$$\mathbf{8}, \mathbf{9}, \mathbf{7}, \mathbf{12}, \mathbf{4}, \mathbf{17}, \mathbf{18}, \mathbf{5}, \mathbf{11}, \mathbf{14}, \mathbf{2}, \mathbf{6}, \mathbf{10}, \mathbf{0}, \mathbf{16}, \mathbf{15}, \mathbf{1}, \mathbf{13}, \mathbf{3}.$$

Fakt 5.4.2

Ist m Primzahl, so dass $m + 1$ durch 4 teilbar ist, dann gilt bei der Verwendung von quadratischem Sondieren:

$$(*) \qquad \{h(x, k) \mid 0 \leq k < m\} = [m].$$

(*Beweis*: Elementare Zahlentheorie.)

Beobachtung (empirisch):

Quadratisches Sondieren verhält sich sehr gut.

(**Vor.:** h gut verteilend, Auslastungsfaktor $\alpha \leq 0{,}9$.)

C – Doppel-Hashing

Benutze zwei (unabhängig berechnete) Hashfunktionen

$$h_1 \colon U \to [m]$$
$$h_2 \colon U \to [m-1]$$

und setze, für $\boldsymbol{k} = 0, 1, 2, \ldots$:

$$\boxed{h(x, \boldsymbol{k}) := (h_1(x) + \boldsymbol{k} \cdot (1 + h_2(x))) \bmod m, \text{ für } k = 0, 1, 2, \ldots.}$$

Beispiel: $m = 13$, $h_1(x) = \mathbf{3}$, $1 + h_2(x) = \mathbf{7}$ liefert Sondierungsfolge

$$\mathbf{3}, \mathbf{10}, \mathbf{4}, \mathbf{11}, \mathbf{5}, \mathbf{12}, \mathbf{6}, \mathbf{0}, \mathbf{7}, \mathbf{1}, \mathbf{8}, \mathbf{2}, \mathbf{9}.$$

Übung: Berechne die Folgenglieder **inkrementell** (Runden $k = 0, 1, 2, \ldots$) ohne Multiplikationen und Divisionen („$\ldots \bmod m$“). *Beispiel*: $\mathbf{3}, 3 + \mathbf{7} = 10, (10 + \mathbf{7}) \bmod 13 = 4, \ldots$

$h(x,k) := (h_1(x) + k \cdot (1 + h_2(x))) \bmod m$, für $k = 0, 1, 2, \ldots$.

Proposition 5.4.3

Wenn m Primzahl ist, dann gilt für **Doppel-Hashing**:

(∗) $\quad \{h(x,k) \mid 0 \leq k < m\} = [m].$

Beweis: Sei $z = 1 + h_2(x)$. Dann: $1 \leq z \leq m-1$. Wir müssen nur zeigen, dass die Abbildung $k \mapsto (h_1(x) + k \cdot z) \bmod m$ auf $\{0, 1, \ldots, m-1\}$ injektiv ist.
Dazu: Sei $h(x,k) = h(x,l)$ mit $0 \leq k, l < m$. Dann ist

$$(h_1(x) + k \cdot z) \bmod m = (h_1(x) + l \cdot z) \bmod m,$$

also $(l-k) \cdot z \equiv 0 \pmod m$, d.h. m teilt $(l-k) \cdot z$.
Da m Primzahl ist und $0 < z < m$ gilt, teilt m die Zahl $l - k$.
Weil $0 \leq |l-k| < m$, folgt $l - k = 0$, also $l = k$. □

Man kann beweisen: Wenn $h_1(x), h_2(x)$, $x \in S$, rein zufällig sind, verhält sich **Doppel-Hashing** sehr gut, nämlich im Wesentlichen wie „Uniformes Sondieren“.

Vorlesungsvideo:

Uniformes Sondieren

D – Uniformes Sondieren/Ideales Hashing

Keine Methode, sondern eine **Wahrscheinlichkeitsannahme** für Verfahren mit offener Adressierung:

(IH): $(h(x,0), h(x,1), \ldots, h(x,m-1))$ ist eine **rein zufällige** Permutation von $[m]$, unabhängig für jedes $x \in U$.

(Erinnerung: Es gibt $m!$ verschiedene Permutationen von $[m]$.)

Effekt von (IH): $x_1, \ldots, x_n$ seien gespeichert; y kommt neu hinzu.
Ganz egal was die Hashwerte von $x_1, \ldots, x_n$ und was $h(y,0), \ldots, h(y,k-1)$ sind:
$h(y,k)$ nimmt jeden Wert $j \in [m] - \{h(y,0), \ldots, h(y,k-1)\}$
mit derselben Wahrscheinlichkeit $1/(m-k)$ an.

n: Anzahl der Schlüssel in $\mathtt{T}[0\,..\,m-1]$; $\alpha = \frac{n}{m}$.

$T'_{m,n}$:= #(untersuchte Fächer bei **erfolgloser Suche**) (eine Zufallsvariable)

$C'_{m,n} := \mathbf{E}(T'_{m,n})$.

$T_{m,n}$:= #(untersuchte Fächer bei **erfolgreicher Suche**), gemittelt über $x_1, \ldots, x_n$ (eine Zufallsvariable).

$C_{m,n} := \mathbf{E}(T_{m,n})$.

Satz 5.4.5

Unter der Annahme (IH) gilt, für $\alpha = n/m$:

(i) $C'_{m,n} = \frac{m+1}{m+1-n} = \frac{1}{1-\frac{n}{m+1}} \quad \left(< \frac{1}{1-n/m} = \frac{1}{1-\alpha} \right)$.

Wenn α fest und $n, m \to \infty$, dann gilt $C'_{m,n} \to \frac{1}{1-\alpha}$.

(Immer wieder **rein zufällige** Positionen zu testen liefert Erfolgswahrscheinlichkeit $\frac{m-n}{m}$ in einem Versuch, also erwartete Versuchsanzahl $\frac{m}{m-n} = \frac{1}{1-\alpha}$.)

Beweis von (i): $x_1, \ldots, x_n$ sind gespeichert, $y \notin \{x_1, \ldots, x_n\}$ wird gesucht.

In jedem Fall wird $\texttt{T}[h(y,0)]$ untersucht, Kosten 1.

1. *Fall*: $\texttt{T}[h(y,0)]$ ist leer.

Wahrscheinlichkeit: $\frac{m-n}{m}$. Weitere Kosten: 0.

2. *Fall*: $\texttt{T}[h(y,0)]$ ist **nicht leer**.

Wahrscheinlichkeit: $\frac{n}{m}$. Weitere Kosten: $\boldsymbol{C'_{m-1,n-1}}$.

(Suche in $m-1$ Plätzen, von denen $n-1$ besetzt sind.)

Rekursionsformel, für $0 \leq n < m$:

$$C'_{m,n} = \begin{cases} 1 & \text{, falls } n = 0 \\ 1 + \frac{n}{m} \cdot C'_{m-1,n-1} & \text{, falls } n \geq 1. \end{cases}$$

Wir haben: $C'_{m,n} = \begin{cases} 1 & \text{, falls } n = 0 \\ 1 + \frac{n}{m} \cdot C'_{m-1,n-1} & \text{, falls } n \geq 1. \end{cases}$ Also:

$$\begin{aligned} C'_{m,0} &= 1\,; \\ C'_{m,1} &= 1 + \frac{1}{m} \cdot 1 = \frac{m+1}{m}\,; \\ C'_{m,2} &= 1 + \frac{2}{m} \cdot \frac{m}{m-1} = \frac{m+1}{m-1}\,; \\ C'_{m,3} &= 1 + \frac{3}{m} \cdot \frac{m}{m-2} = \frac{m+1}{m-2}\,; \end{aligned}$$

und so weiter. Dies liefert:

$$C'_{m,n} = \frac{m+1}{m+1-n}.$$

(Formaler Beweis durch vollständige Induktion.) □

Satz 5.4.5

Unter der Annahme (IH) gilt bei $\alpha = n/m$:

(i) $C'_{m,n} = \frac{m+1}{m+1-n} = \frac{1}{1-\frac{n}{m+1}}$.

Dies ist $< \frac{1}{1-n/m} = \frac{1}{1-\alpha}$.

Wenn α fest und $n, m \to \infty$, dann gilt $C'_{m,n} \to \frac{1}{1-\alpha}$.

(ii) Für $\boldsymbol{T_{m,n}}$ = #(untersuchte Fächer bei **erfolgreicher Suche**), gemittelt über $x_1, \ldots, x_n$, gilt:

$\boldsymbol{C_{m,n}} = \mathbf{E}(T_{m,n}) < \frac{1}{\alpha} \cdot \ln(\frac{1}{1-\alpha})$.

Beweis von (ii): Beachte: Suchaufwand für $x_i \in S$ = Einfügeaufwand für x_i.

Also Mittelung über $C'_{m,0}, C'_{m,1}, \ldots, C'_{m,n-1}$:

$$C_{m,n} = \frac{1}{n} \cdot \sum_{1 \le i \le n} \frac{m+1}{m+1-(i-1)}$$

$$= \frac{m+1}{n} \cdot \left(\frac{1}{m+1} + \frac{1}{m} + \cdots + \frac{1}{m+1-(n-1)} \right).$$

Leicht zu sehen (Untersumme!):

$$\frac{1}{m+1} + \frac{1}{m} + \cdots + \frac{1}{m+1-(n-1)} \le \int_{m+1-n}^{m+1} \frac{dt}{t} =$$

$$= \ln(m+1) - \ln(m+1-n) = \ln\left(\frac{m+1}{m+1-n}\right).$$

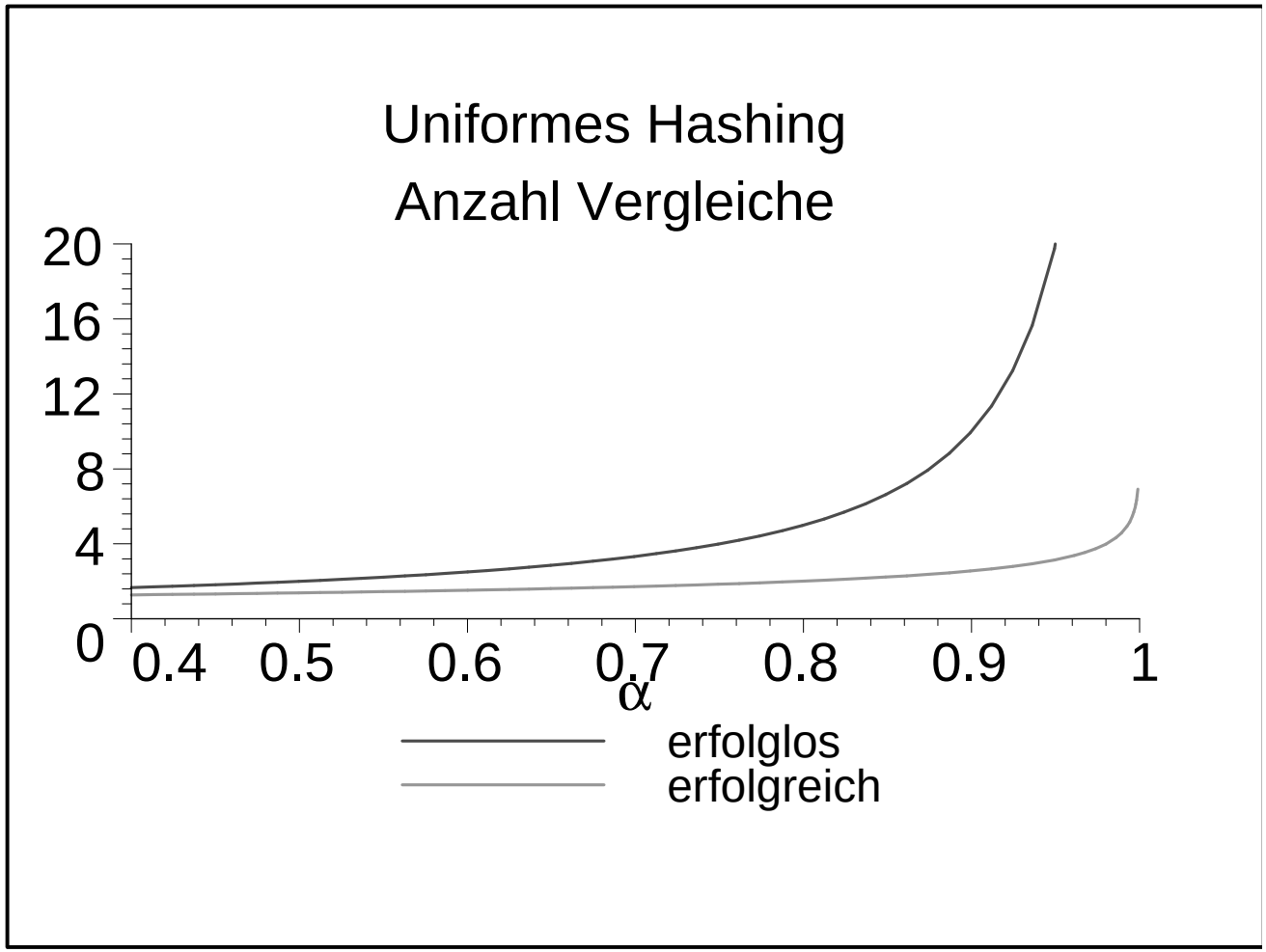
Also:

$$C_{m,n} \leq \frac{m+1}{n} \cdot \ln\left(\frac{1}{1-\frac{n}{m+1}}\right) \overset{(!)}{<} \frac{m}{n} \cdot \ln\left(\frac{1}{1-\frac{n}{m}}\right).$$

Für $\alpha = n/m$, α fest, $n, m \to \infty$, ist $C_{m,n} \approx \frac{1}{\alpha} \ln(\frac{1}{1-\alpha})$. □

Für „$\overset{(!)}{<}$“ beweist man durch Kurvendiskussion: $t \mapsto \frac{1}{t} \cdot \ln(\frac{1}{1-t})$ ist für $0 < t < 1$ monoton wachsend.

Bemerkung: Für $\alpha \nearrow 1$ geht $\frac{1}{\alpha} \cdot \ln(\frac{1}{1-\alpha})$ **extrem langsam** gegen ∞.


Uniformes Hashing
Anzahl Vergleiche
20
16
12
8
4
0
0.4
0.5
0.6
0.7
0.8
0.9
1
α
erfolglos
erfolgreich

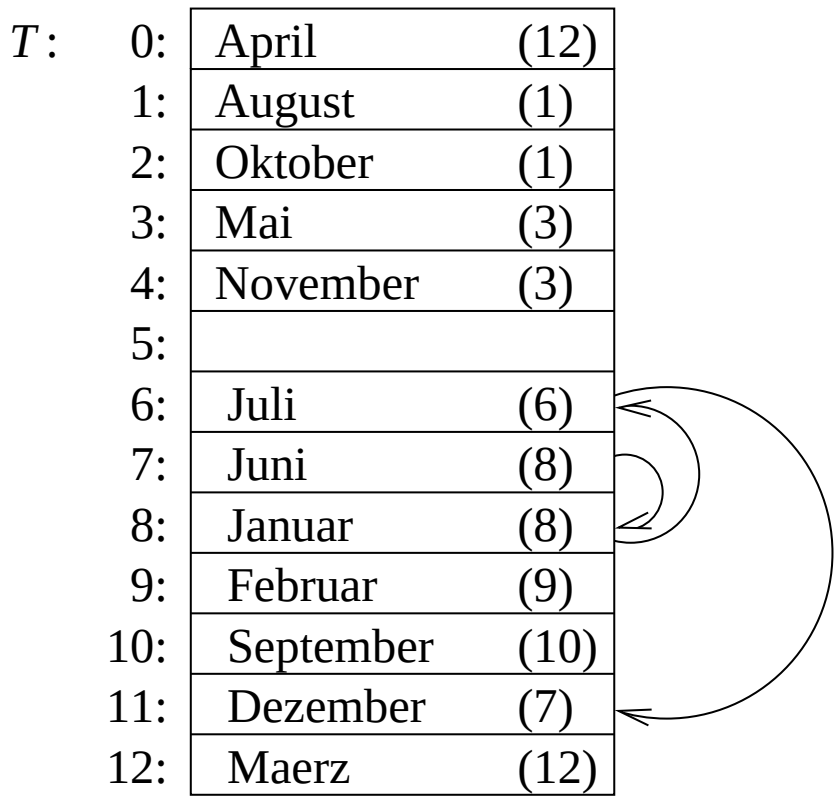
Erwartete Anzahl von Schlüsselvergleichen bei uniformem Hashing:

	erfolglose Suche	erfolgreiche Suche (Mittel über $x_1, \ldots, x_n$)
α	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha}\ln(\frac{1}{1-\alpha})$
0,5	2	1,39
0,6	2,5	1,53
0,7	3,33	1,72
0,75	4	1,96
0,8	5	2,01
0,9	10	2,56
0,95	**20**	3,15
0,98	**50**	3,99
0,99	**100**	4,65

Vorlesungsvideo:

Löschen bei geschlossenem Hashing

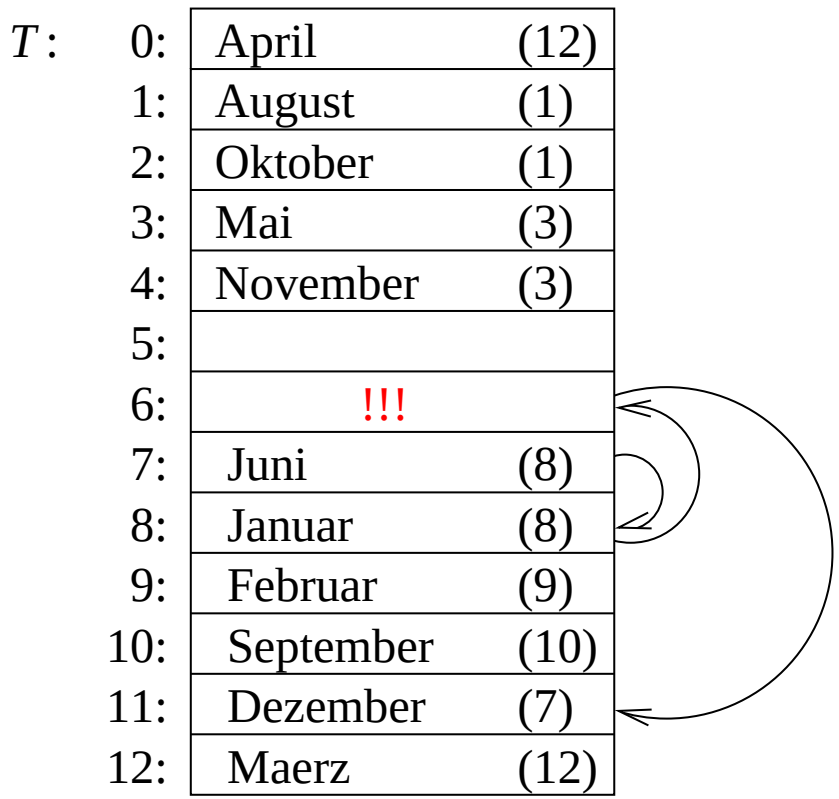
Löschungen? Problem z. B. bei Quadratischem Sondieren.



Suchfolge „Dezember“ mit
$h(\text{Dezember}) = 7$: $7, 8, 6, 11$.

Lösche „Juli“!

Löschungen? Problem z. B. bei Quadratischem Sondieren.



Suchfolge „Dezember“ mit
$h(\text{Dezember}) = 7$: $7, 8, 6, 11$.

Lösche „Juli“!

„Dezember“ wird nicht mehr gefunden.

Ausweg: Jede Zelle hat Statusvariable `status` (*voll*, *leer*, *gelöscht*).

Löschen: Zelle nicht auf *leer*, sondern auf *gelöscht* setzen (engl.: „*tombstone*“).

Suchen: Immer bis zur ersten *leer*en Zelle gehen.

Zellen, die *gelöscht* sind, dürfen mit einem neuen Schlüssel überschrieben werden.

Überlauf tritt **spätestens** ein, wenn die letzte *leer*e Zelle beschrieben werden würde.

Sinnvoller: Überlauf tritt ein, wenn die Anzahl der *voll*en und *gelöscht*en Zellen zusammen eine feste Schranke `bound` überschreitet.

Z.B. $m-1$ oder $\alpha_1 m$, α_1 passend zum Sondierungsverfahren.

Alle Operationen bei offener Adressierung

empty(m):

Lege leeres Array `T[`$0\,..\,m-1$`]` an.

Initialisiere alle Zellen `T[`j`]` mit $(-,-,$ *leer*$)$.

Wähle Hashfunktion $h\colon U \times [m] \to [m]$.

Initialisiere `inUse` mit 0. (Zählt **nicht-leere** Zellen.)

Initialisiere `load` mit 0. (Zählt Einträge.)

Initialisiere `bound` (z. B. mit $\alpha_0 m$, maximal $m-1$).

lookup(x):

Finde erstes l in der Folge $h(x,0), h(x,1), h(x,2), \ldots$ mit

1. Fall: `T[`l`].status` $=$ *leer*: **return** „*not found*“;

oder

2. Fall: (`T[`l`].status` $=$ *voll* **und** `T[`l`].key` $= x$): **return** `T[`l`].data`.

delete(x):

Finde erstes l in der Folge $h(x,0), h(x,1), h(x,2), \ldots$ mit

1. Fall: `T[`l`].status` $=$ *leer*: tue nichts; // evtl. Warnung: *not found*

oder

2. Fall: (`T[`l`].status` $=$ *voll* **und** `T[`l`].key` $= x$): `T[`l`]` $\leftarrow (-, -,$ *gelöscht*$)$; `load--`.

insert(x, r):

Finde erstes l in der Folge $h(x,0), h(x,1), h(x,2), \ldots$
mit `T[`l`].status` $=$ *leer* **oder** (`T[`l`].status` $=$ *voll* **und** `T[`l`].key` $= x$);

Merke dabei das erste l' in dieser Folge mit `T[`l'`].status` $\in \{$*gelöscht*, *leer*$\}$;

1. **Fall:** `T[`l`].status` $=$ *voll*: `T[`l`].data` $\leftarrow r$; // Update
2. **Fall:** `T[`l'`].status` $=$ *gelöscht*: `T[`l'`]` $\leftarrow (x, r,$ *voll* $)$; `load++`;
   // überschreibe gelöschtes Feld
3. **Fall:** `T[`l'`].status` $=$ *leer*: // $l = l'$, neue Zelle
   if `inUse` $+ 1 >$ `bound` **then** „Überlaufbehandlung“
   else `inUse++`; `load++`; `T[`l`]` $\leftarrow (x, r,$ *voll* $)$;

Überlaufbehandlung, Verdopplung:

Sinnvoll: **Überlauf** tritt ein, wenn die Anzahl der *voll*en und *gelöscht*en Zellen zusammen (in `inUse`) eine Schranke $\alpha_0 m$ (in `bound`) überschreitet.

(α_0 passend zum Sondierungsverfahren festlegen.)

Anzahl der *voll*en Zellen: `load`.

Falls $\texttt{load} + 1 > \alpha_1 m$ für (z. B.) $\alpha_1 = 0.8\alpha_0$, **verdopple** Tabellengröße, und trage Schlüssel aus `T` **neu** in die größere Tabelle ein. Setze $\texttt{inUse} \leftarrow \texttt{load}$.

Sonst: Trage alle Schlüssel **neu** in die **alte Tabelle** `T` ein,
eliminiere alle „*gelöscht*“-Markierungen, wie folgt . . .

Trick für Umorganisieren: (Ohne zusätzlichen Platz.)

1) `inUse` $\leftarrow$ `load`

2) Für $i = 0, 1, \ldots, m-1$ tue folgendes:
Wenn $\texttt{T}[i]\texttt{.status} =$ *voll*, setze $\texttt{T}[i]\texttt{.status} \leftarrow$ *alt*.
Wenn $\texttt{T}[i]\texttt{.status} =$ *gelöscht*, setze $\texttt{T}[i]\texttt{.status} \leftarrow$ *leer*.

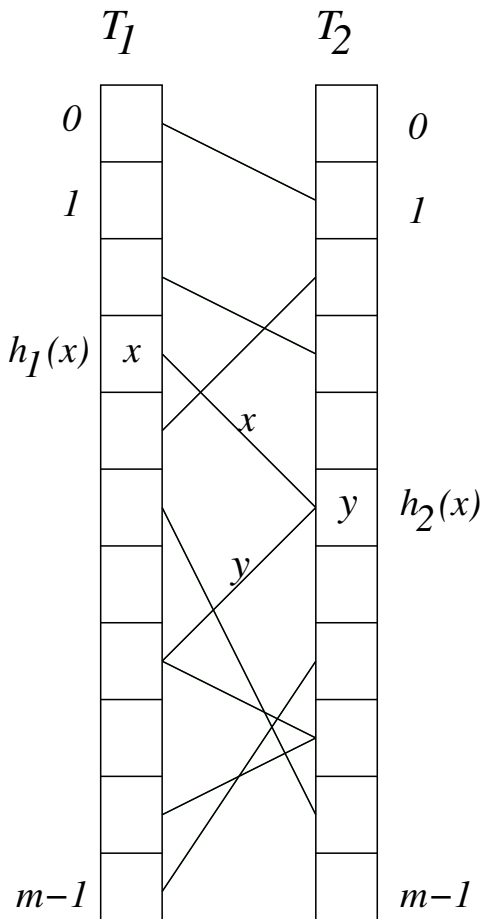
3) Für $i = 0, 1, \ldots, m-1$ tue folgendes: // checke jeden Platz
Wenn $\texttt{T}[i]\texttt{.status} =$ *alt*, dann
    setze $(\texttt{x}, \texttt{r}) \leftarrow (\texttt{T}[i]\texttt{.key}, \texttt{T}[i]\texttt{.data})$;
    setze $\texttt{T}[i]\texttt{.status} \leftarrow$ *leer*;
    insert$^*(\texttt{x}, \texttt{r})$.

    Dabei ist **insert***(x,r) folgende rekursive Variante von **insert**(x,r):
    Finde erstes l in der Folge $h(x, j), j = 0, 1, 2, \ldots$ mit $\texttt{T}[l]\texttt{.status} \in \{$*leer*, *alt*$\}$;
    Falls $\texttt{T}[l]\texttt{.status} =$ *leer*: $\texttt{T}[l] \leftarrow (x, r,$ *voll*$)$.
    Falls $\texttt{T}[l]\texttt{.status} =$ *alt*: $(\texttt{y}, \texttt{s}) \leftarrow (\texttt{T}[l]\texttt{.key}, \texttt{T}[l]\texttt{.data})$;
    $\texttt{T}[l] \leftarrow (x, r,$ *voll*$)$;
    insert$^*(\texttt{y}, \texttt{s})$.

Vorlesungsvideo:

Cuckoo Hashing

5.5 Cuckoo Hashing [Pagh/Rodler 2001/04]



Implementierung eines dynamisches Wörterbuchs

Zwei Tabellen T_1, T_2, jeweils Größe m

Zwei Hashfunktionen h_1 und h_2

(∗) $x \in S$ sitzt in $T_1[h_1(x)]$ oder in $T_2[h_2(x)]$.

$\Rightarrow$ **Konstante Suchzeit** garantiert.

Auch *delete*(x) in $O(1)$ Zeit.

Cuckoo-Hashing

Definition 5.5.1

h_1, h_2 **passen zu** S, falls die Schlüssel aus S gemäß der Regel $(*)$ gespeichert werden können, d. h. man kann S in S_1 und S_2 partitionieren, so dass h_1 auf S_1 und h_2 auf S_2 injektiv ist.

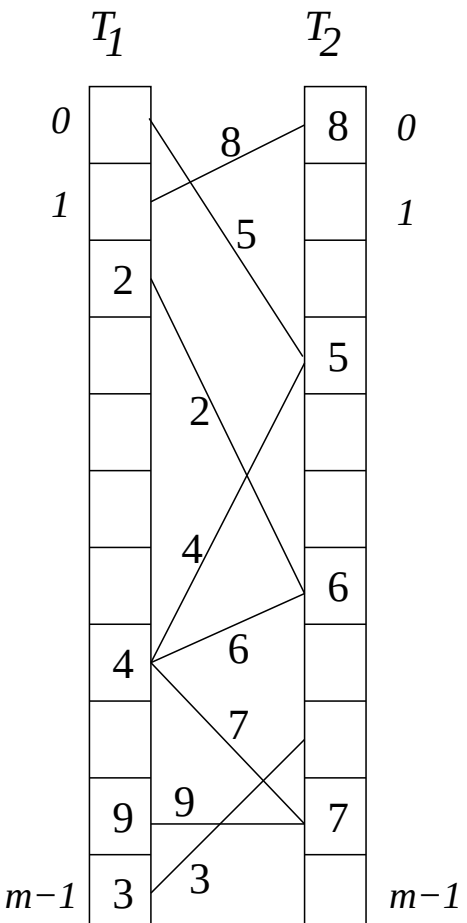
Das Verfahren heißt **"Kuckucks-Hashing"** wegen seiner Einfügeprozedur:

Schlüssel x, der eingefügt werden soll, kann Schlüssel y, der in $T_1[h_1(x)]$ oder $T_2[h_2(x)]$ („im Nest") sitzt,

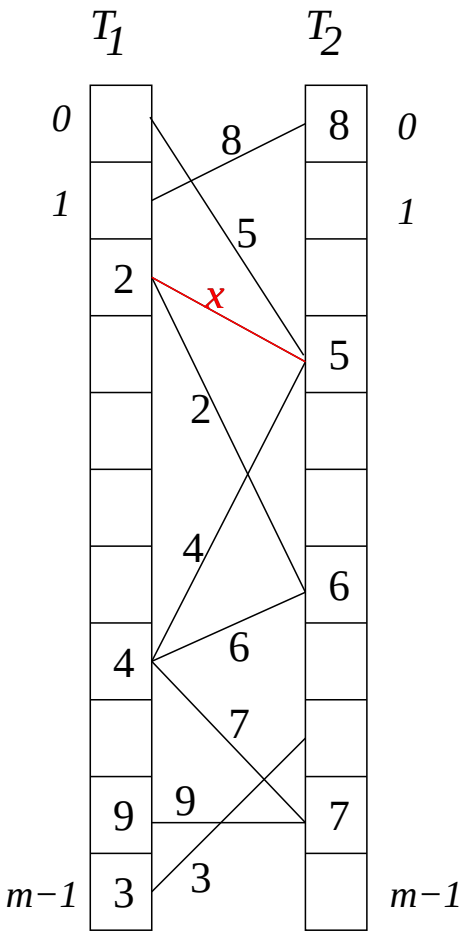
hinauswerfen (verdrängen).

Der verdrängte Schlüssel geht zu seinem alternativen Platz.

Cuckoo-Hashing

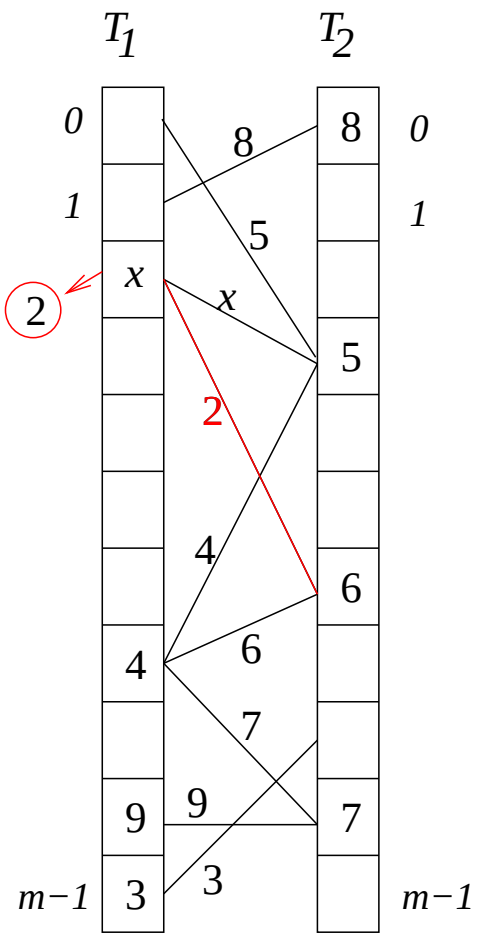


Cuckoo-Hashing



Füge x ein.
Versuche $T_1[h_1(x)]$.
Besetzt!

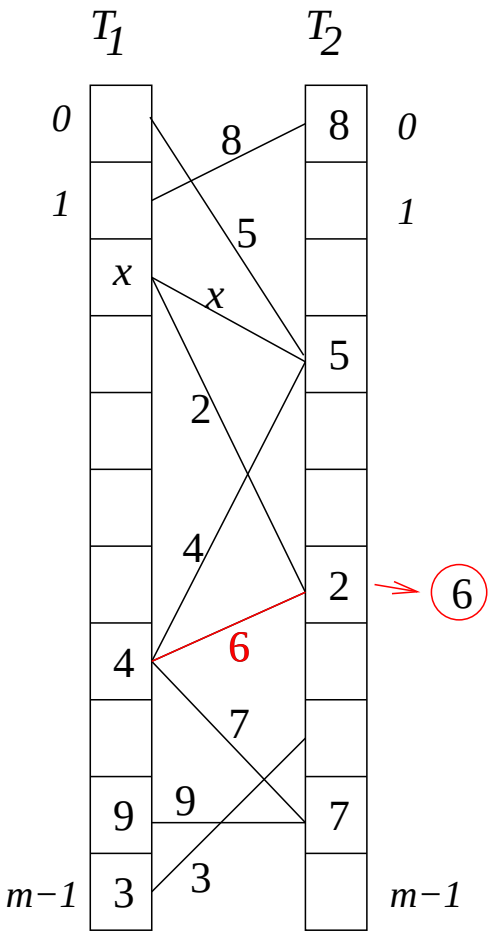
Cuckoo-Hashing



Schlüssel x in T_1 einfügen.

Schlüssel 2 aus T_1 hinauswerfen, soll nach T_2 gehen.

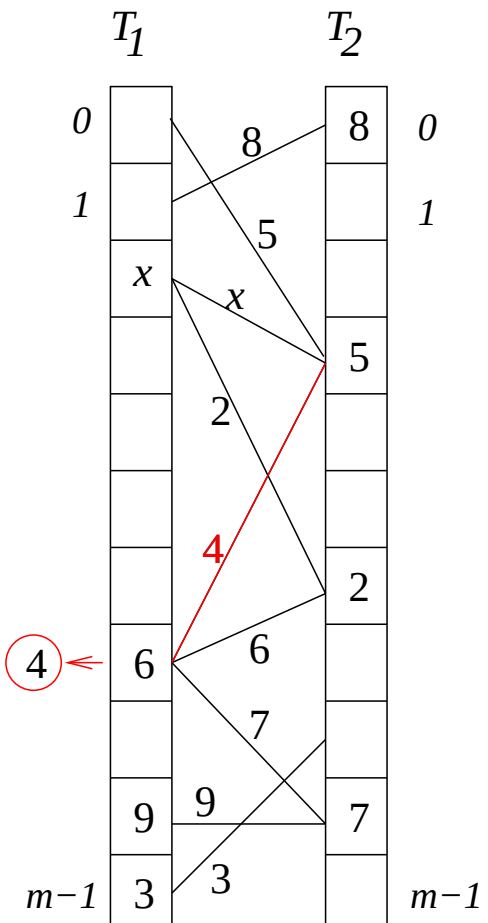
Cuckoo-Hashing



Schlüssel 2 in T_2 einfügen.

Schlüssel 6 aus T_2 hinauswerfen,
soll nach T_1 gehen.

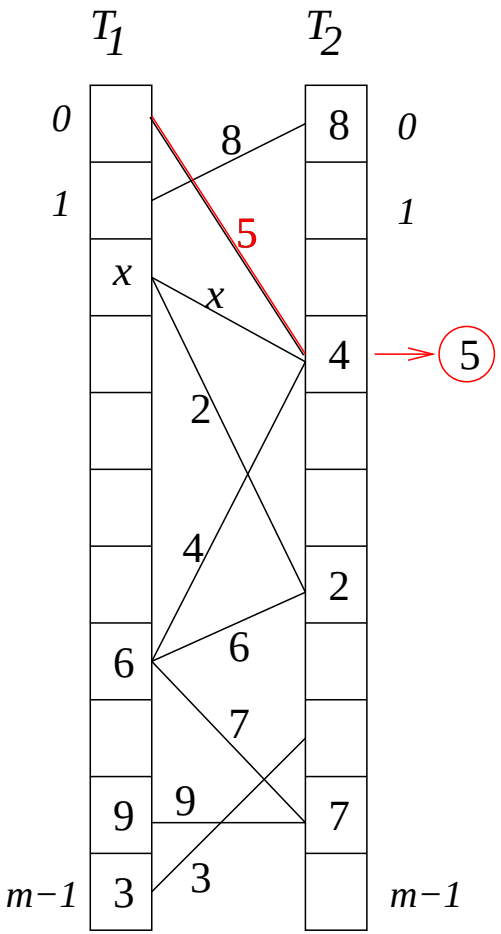
Cuckoo-Hashing



Schlüssel 6 in T_1 einfügen.

Schlüssel 4 aus T_1 hinauswerfen,
soll nach T_2 gehen.

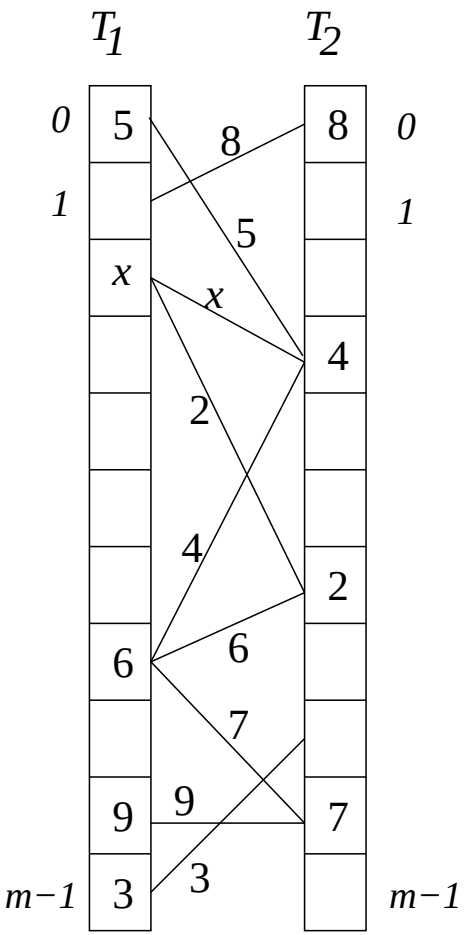
Cuckoo-Hashing



Schlüssel 4 in T_2 einfügen.

Schlüssel 5 aus T_2 hinauswerfen,
soll nach T_1 gehen.

Cuckoo-Hashing



Schlüssel 5 in T_1 einfügen.

Funktioniert, weil $T_1[h_1(5)]$ leer ist.

Fertig!

Cuckoo-Hashing: Teil-Analyse

Grundfrage: Wann funktioniert das überhaupt?

Hier zunächst: Wie lange dauert eine Einfügung?

Wie viele Verdrängungsschritte gibt es bei Einfügung von x?

Wir beobachten: Wenn die Einfügung von x zur Verdrängung von y führt, dann gibt es eine Folge von Schlüsseln $x_1, \ldots, x_k$ mit:

x verdrängt x_1, x_1 verdrängt x_2, . . . , x_k verdrängt y.

Damit das passieren kann, muss es eine einfache „$x..y$-Kette“ $x, x_1, x_2, \ldots, x_k, y$ aus *verschiedenen* Schlüsseln geben, die über die Hashfunktionen eine Verbindung herstellt:

$$\begin{aligned} h_1(x) &= h_1(x_1), \\ h_2(x_1) &= h_2(x_2) \\ h_1(x_2) &= h_1(x_3) \\ &\vdots \\ h_1(x_\ell) &= h_1(y). \end{aligned}$$

(Auch möglich: Die Folge beginnt mit $h_2(x) = h_2(x_1)$ oder sie endet mit $h_2(x_\ell) = h_2(y)$.)

Wir sammeln alle in Frage kommenden y in eine (Zufalls-)Menge

$$\boldsymbol{B_x} := \{y \in S \mid \text{es gibt eine (}\textit{einfache}\text{) } x \ldots y\text{-Kette}\}.$$

Wir wollen die erwartete Größe $\mathbf{E}(|B_x|)$ abschätzen.

Dazu schreiben wir $|B_x| = \sum_{y \in S} X_y$ mit

$$\boldsymbol{X_y} := \begin{cases} 1 & \text{wenn } y \in B_x \\ 0 & \text{wenn } y \notin B_x. \end{cases}$$

Dann gilt (Linearität des Erwartungswertes):

$$\mathbf{E}(|B_x|) = \sum_{y \in S} \mathbf{E}(X_y) = \sum_{y \in S} \mathbf{Pr}(\text{es gibt } x \ldots y\text{-Kette}).$$

Wir schätzen $\mathbf{E}(X_y)$ mit Hilfe der „Vereinigungsschranke“ $\mathbf{Pr}(A_1 \cup \cdots \cup A_\ell) \leq \mathbf{Pr}(A_1) + \cdots + \mathbf{Pr}(A_\ell)$ ab:

$$\begin{aligned}\mathbf{E}(X_y) &= \mathbf{Pr}(\exists k \geq 1\text{: es gibt einfache } x \ldots y\text{-Kette der Länge } k) \\ &\leq \sum_{k\geq 1} \sum_{x_1,\ldots,x_k \in S} \mathbf{Pr}(x, x_1, \ldots, x_k, y \text{ ist } x \ldots y\text{-Kette}) \\ &\leq \sum_{k\geq 1} 4n^k \frac{1}{m^{k+1}}\end{aligned}$$

($k+1$ Hashwerte sind gleich; Faktor 4 wg. h_1/h_2-Auswahl)

$$\cdots = \frac{4}{m} \cdot \sum_{k\geq 1} \left(\frac{n}{m}\right)^k \leq \frac{4}{m} \cdot \sum_{k\geq 0} (1-\beta)^k = \frac{4}{m} \cdot \frac{1}{\beta},$$

falls $n/m \leq 1-\beta$. Damit: $\mathbf{E}(|B_x|) \leq 4n/(m\beta) \leq \frac{4(1-\beta)}{\beta} < 4/\beta$.

Bei einer erfolgreichen Einfügung wird kein Schlüssel mehr als zweimal verdrängt.

Wir erhalten also

$$\mathbf{E}(\text{Einfügezeit}) = O(\mathbf{E}(|B_x|)) = O(1/\beta).$$

Leider ist die Analyse nicht ganz vollständig. Man muss noch die Wahrscheinlichkeit abschätzen, dass h_1 und h_2 nicht zu S passen. Diese ergibt sich (mit etwas komplizierteren Überlegungen) zu $\Theta(1/n)$.

Damit kann man dann die Analyse vervollständigen: Erwartete Einfügezeit ist $O(1)$, selbst wenn man den mit Wahrscheinlichkeit $O(1/n)$ nötigen Neuaufbau mit in Rechnung stellt.

Literatur: Rasmus Pagh, Cuckoo Hashing for Undergraduates, March 27, 2006, `https://www.itu.dk/people/pagh/papers/cuckoo-undergrad.pdf`

Cuckoo-Hashing: Analyse

Beobachtung 1 (trivial):

Wiederholte Anwendung der
Einfügeprozedur platziert alle Schlüssel aus S in T_1, T_2

$\Rightarrow$

h_1, h_2 passen zu S.

D.h.: **Wenn** h_1, h_2 **nicht** zu S passen, **dann** muss die Einfügung eines Schlüssels **fehlschlagen**.

Wie sieht ein Fehlschlag aus?

Cuckoo-Hashing: Analyse

Wann gehen die Einfügungen gut?

Wie lange dauert eine Einfügung?

Beobachtung 1 (trivial):

Wiederholte Anwendung der
Einfügeprozedur platziert alle Schlüssel aus S in T_1, T_2

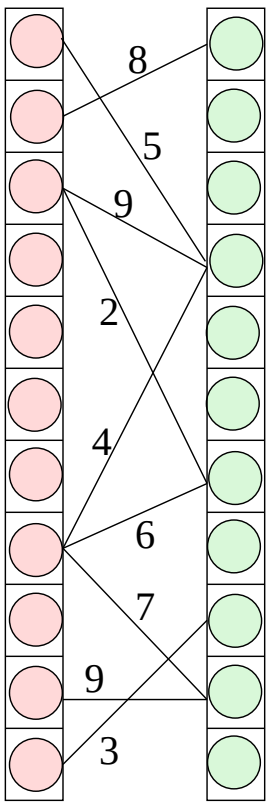
$\Rightarrow$

h_1, h_2 passen zu S.

D.h.: **Wenn** h_1, h_2 **nicht** zu S passen, **dann** muss die Einfügung eines Schlüssels **fehlschlagen**.

Wie sieht ein Fehlschlag aus?

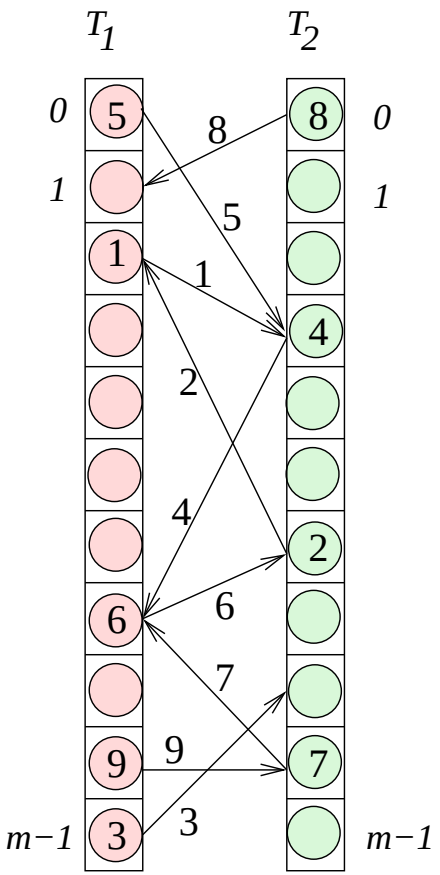
Cuckoo-Hashing: Graph



Bipartiter
Cuckoo-Graph
$G(S, h_1, h_2)$

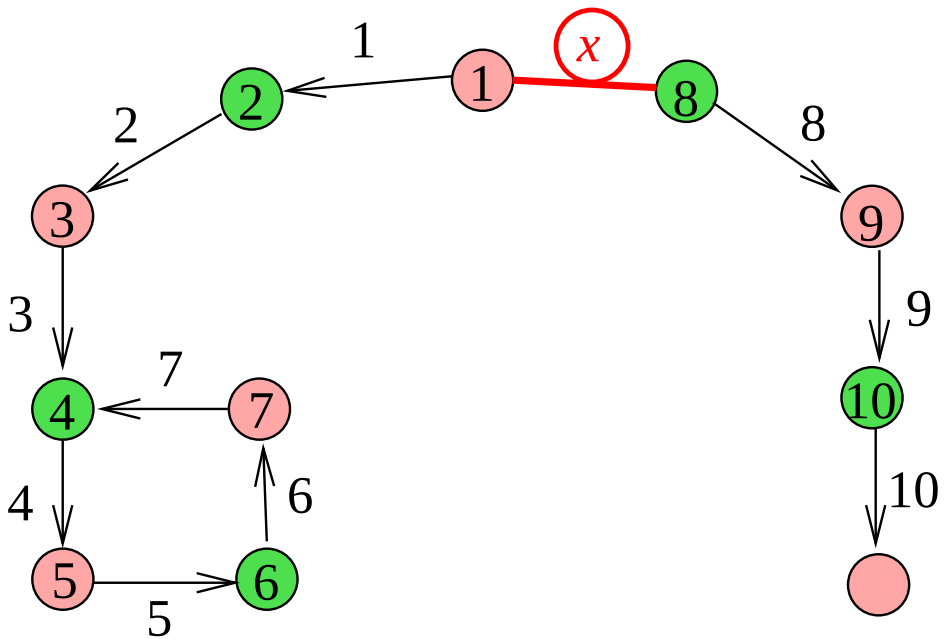
$V = W = [m]$
$E = \{(h_1(x), h_2(x)) \mid x \in S\}$

Cuckoo-Hashing: Graph



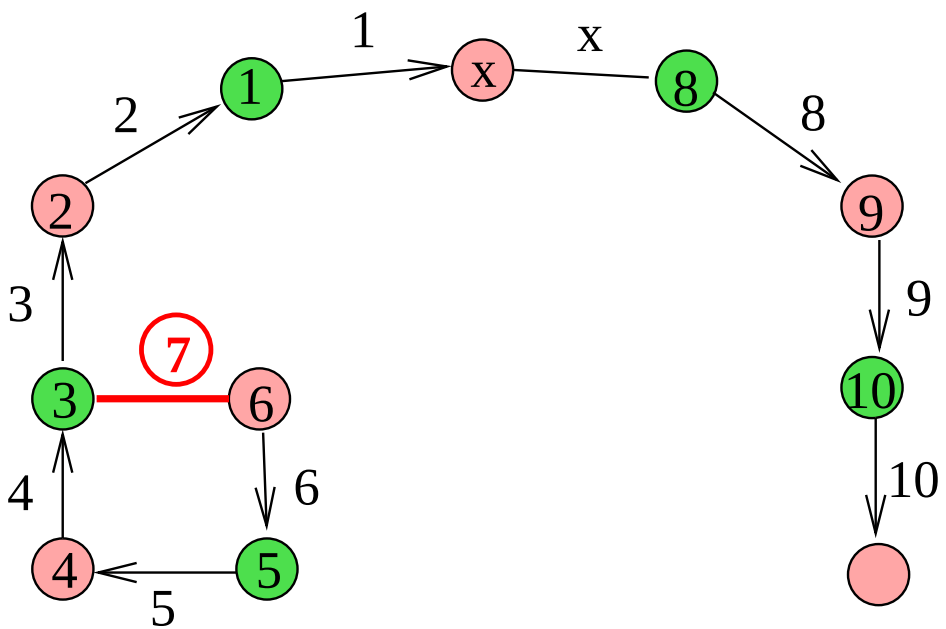
h_1, h_2 passen zu S
$\Leftrightarrow$
Man kann die Kanten von $G(S, h_1, h_2)$ so richten, dass jeder Knoten Ausgrad ≤ 1 hat.

Cuckoo-Hashing: Fehlschlag?



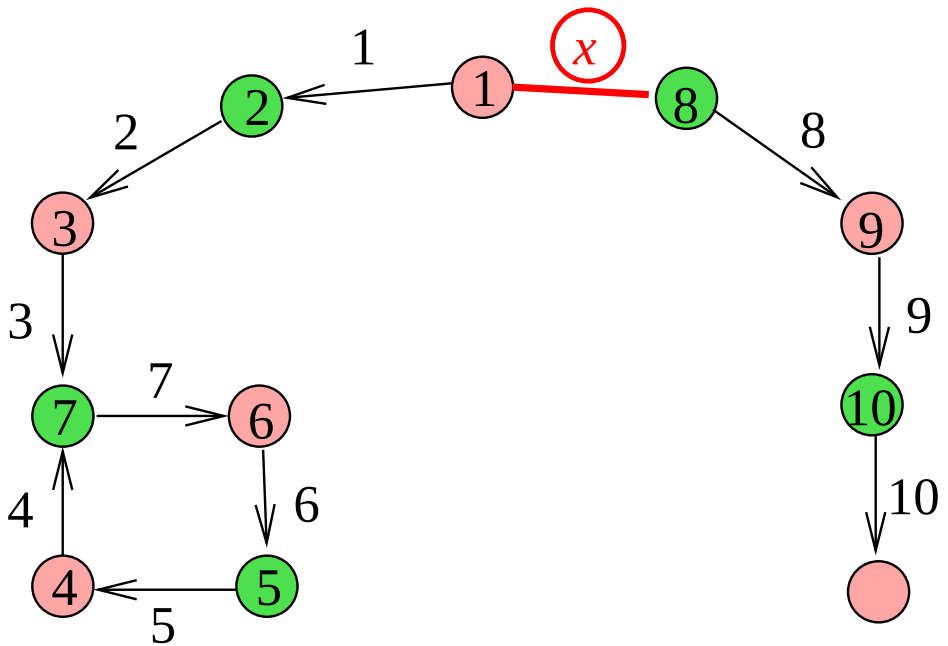
Neuer Schlüssel x.

Cuckoo-Hashing: Fehlschlag?



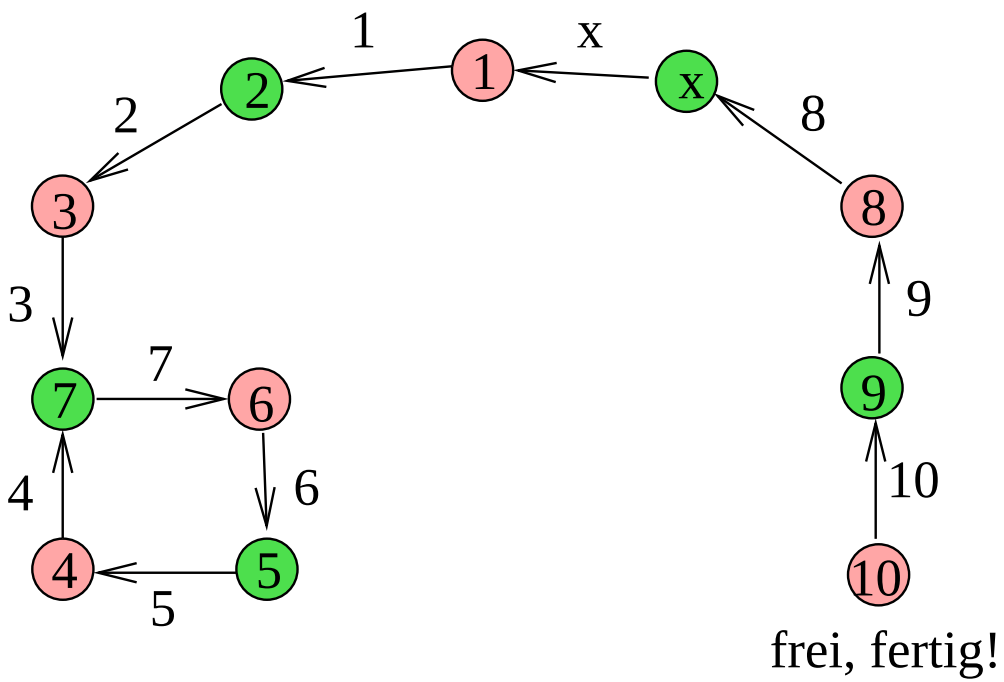
Nach 6 Runden. Schlüssel 7 ist draußen.

Cuckoo-Hashing: Fehlschlag?

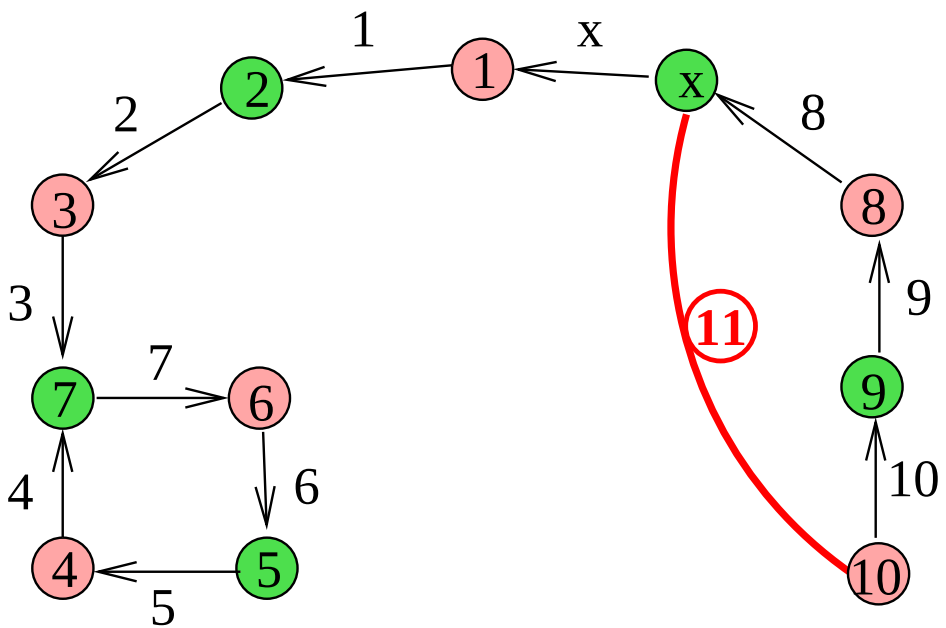


Nach weiteren 4 Runden.

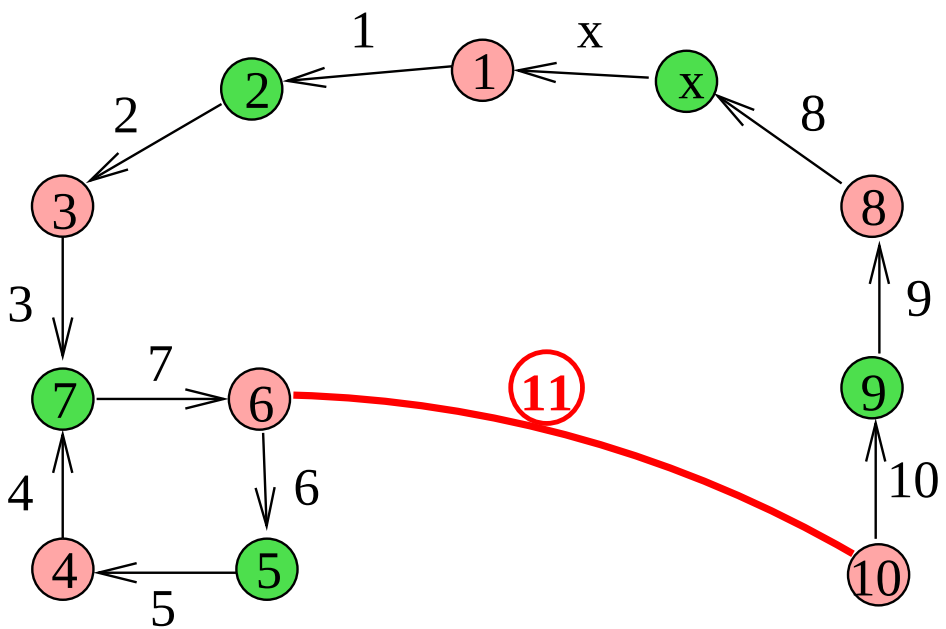
Cuckoo-Hashing: Fehlschlag?



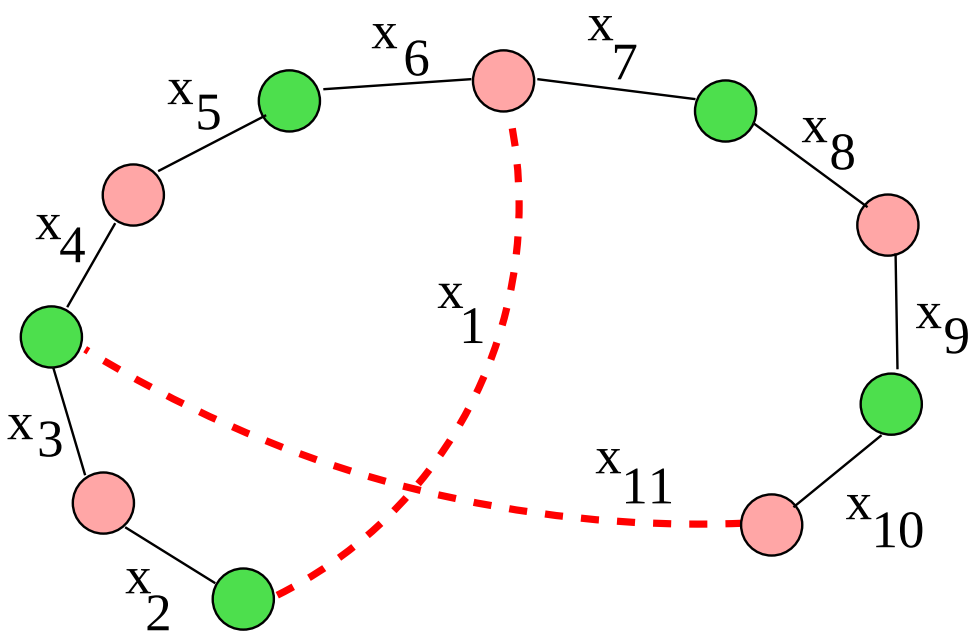
Cuckoo-Hashing: Fehlschlag!



Cuckoo-Hashing: Fehlschlag!

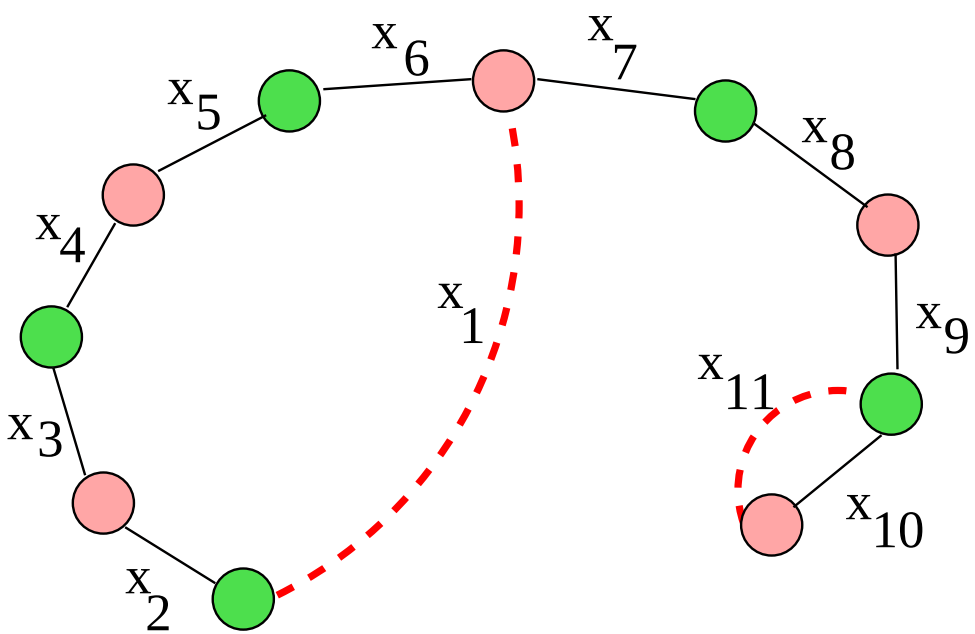


Cuckoo-Hashing: Fehlschlag



„Sperrstruktur“

Cuckoo-Hashing: Fehlschlag



„Sperrstruktur“

Cuckoo-Hashing: Analyse

Beobachtung 2: Aus dem Ablauf der Einfügung folgt:
Wenn h_1, h_2 **nicht** zu S passen, **dann** existiert in $G(S, h_1, h_2)$ eine **Sperrstruktur**, d. h. es gibt eine Folge von Schlüsseln $x_1, \ldots, x_k$ in S, deren Hashwerte wie folgt kollidieren (z. B., falls k ungerade):

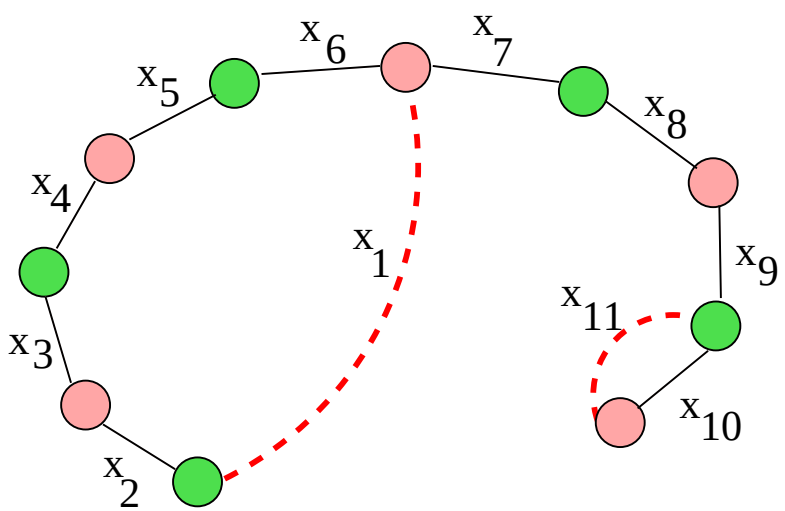
$$h_2(x_1) = h_2(x_2), \qquad h_1(x_2) = h_1(x_3),$$
$$h_2(x_3) = h_2(x_4), \qquad h_1(x_4) = h_1(x_5),$$
$$\vdots \qquad\qquad \vdots$$
$$h_2(x_{k-2}) = h_2(x_{k-1}), \qquad h_1(x_{k-1}) = h_1(x_k),$$

$$h_1(x_1) \in \{h_1(x_3), \ldots, h_1(x_k)\},$$
$$h_2(x_k) \in \{h_2(x_2), \ldots, h_2(x_{k-1})\}.$$

Cuckoo-Hashing: Analyse

Beobachtung 2': Sogar Äquivalenz!

Wenn in $G(S, h_1, h_2)$ eine **Sperrstruktur** existiert,
dann passen h_1, h_2 nicht zu S.



Mehr Kanten/Schlüssel als Knoten/Tabellenplätze!

Cuckoo-Hashing: Analyse

Sei $n/m \leq 1 - \beta$. Dann gilt:

$$\begin{aligned}
&\textbf{Pr}(h_1, h_2 \text{ passen nicht zu } S) \\
&\leq \sum_{3 \leq k \leq n} n^k \cdot 2 \cdot \left(\frac{1}{m}\right)^{k-1} \cdot \left(\frac{k}{2m}\right)^2 \\
&< \frac{1}{2m} \cdot \sum_{k \geq 3} k^2 \cdot \left(\frac{n}{m}\right)^k \\
&\leq \frac{1}{2m} \cdot \sum_{k \geq 3} \cdot k^2 \cdot (1-\beta)^k = O\left(\frac{1}{\beta^3 \cdot m}\right).
\end{aligned}$$

Cuckoo-Hashing: Analyse

(Es gibt $\leq n^k$ Folgen $(x_1, \ldots, x_k)$ von Schlüsseln.
Die erste Kante kann einen h_1- oder einen h_2-Endpunkt haben.
Dass die Hashwerte entlang des Weges übereinstimmen, hat Wahrscheinlichkeit $1/m^{k-1}$.
Die Anzahl der möglichen Trefferknoten für jede der beiden „zurückschlagenden“ Kanten ist $\leq k/2$.)

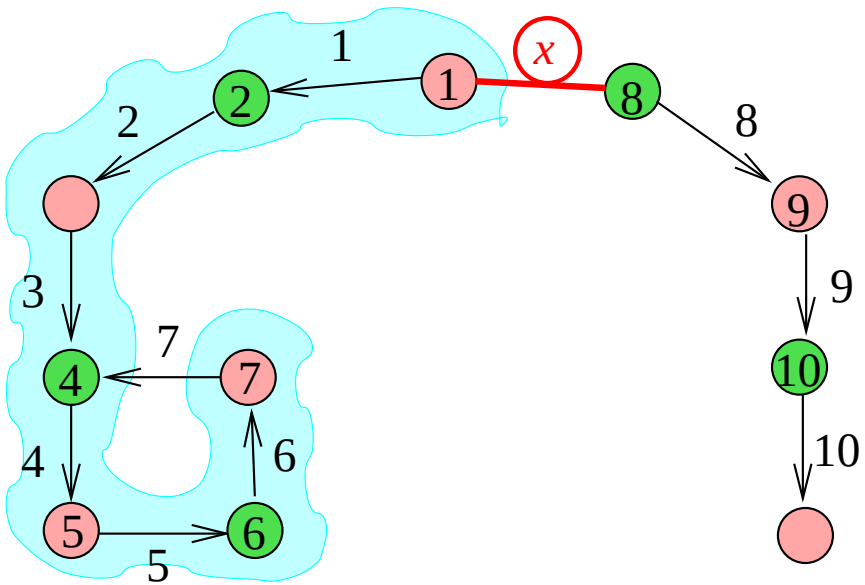
Cuckoo-Hashing: Analyse

Benötige: Mechanismus, um Endlosschleifen (Einfügung, die Sperrstruktur erzeugt) abzubrechen.

Lasse Einfügezyklus für **maximal** $C \log n$ Schritte laufen,
C eine Konstante.
Danach: „Rehash“ (neue Hashfunktion, Neuaufbau).

Will aber keine erfolgreiche Einfügung verlieren, nur weil sie lange dauert!

Cuckoo-Hashing: Analyse



Beobachtung 3:
Wenn die **erfolgreiche** Einfügung von x mindestens t Verdrängungen durchführt, dann gibt es in $G(S, h_1, h_2)$ einen einfachen Weg der Länge $\geq \lceil (t-1)/3 \rceil$, der an Knoten $h_1(x)$ oder $h_2(x)$ beginnt.

Cuckoo-Hashing: Analyse

$$
\begin{aligned}
&\textbf{Pr}(\geq t \text{ Verdrängungsschritte}) \\
&\quad \leq 2 \cdot n^{\lceil (t-1)/3 \rceil} \left(\frac{1}{m}\right)^{\lceil (t-1)/3 \rceil} \\
&\quad \leq 2 \cdot (1-\beta)^{(t-1)/3} \\
&\quad \leq 2^{c-t/d} \text{ , für Konstante } c, d.
\end{aligned}
$$

$\Rightarrow$ **Pr**$(\geq C \log n$ Verdrängungsschritte für $x) = O(1/m^3)$
für Konstante $C \geq \frac{3 \ln 2}{\beta}$ $[\geq 3/\log(1/(1-\beta))]$.

$\Rightarrow$ **Pr**$(\exists x \in S\colon \geq C \log n$ Verdrängungsschritte für $x) = O(1/m^2)$.

Cuckoo-Hashing: Analyse

Einfügung von x kostet:

$$
\begin{aligned}
&\mathbf{E}(\text{Anzahl Verdrängungsschritte}) \\
&\overset{*}{\leq} \sum_{t \geq 1} \mathbf{Pr}(\geq t \text{ Verdrängungsschritte}) \\
&\leq \sum_{t \geq 1} 2^{c - t/d} \\
&= O(1).
\end{aligned}
$$

$*$ Standardformel für Erwartungswerte.

Cuckoo-Hashing: Zusammenfassung

Mitteilung 5.5.2 Vor.: **(UF*)** Hashfunktionen verteilen Schlüssel **rein zufällig**. Wenn man Cuckoo-Hashing mit n Schlüsseln in zwei Tabellen mit je m Plätzen durchführt, wobei $\boldsymbol{n/m \leq 1 - \beta}$ (essenziell!), dann passen h_1, h_2 mit Wahrscheinlichkeit $1 - O(1/m)$ zur Schlüsselmenge S.

Falls h_1, h_2 passen, ist die erwartete Einfügezeit $O(1/\beta)$.

Bei Auftreten eines Fehlers (z. B. Einfügung nach $10 \log n$ Runden nicht beendet): „Rehash“ (neue Hashfunktionen wählen) mit erwarteten Kosten $O(n)$. „Amortisierte“ erwartete Einfügezeit: $O(1)$.

Literatur: [Rasmus Pagh, Cuckoo Hashing for Undergraduates],
`https://www.itu.dk/people/pagh/papers/cuckoo-undergrad.pdf`

Nachteil: Auslastung der Tabelle **unter 50%**.
Abhilfe: Mehr als 2 Hashfunktionen oder größere Behälter. (Details in Spezialvorlesungen.)
Die verfeinerten Verfahren sind auch **praktisch** sehr effizient (auch mit Tabellenhashing).

Multiplikative/lineare Hashfunktionen nicht benutzen!

SS 2021

Algorithmen und Datenstrukturen

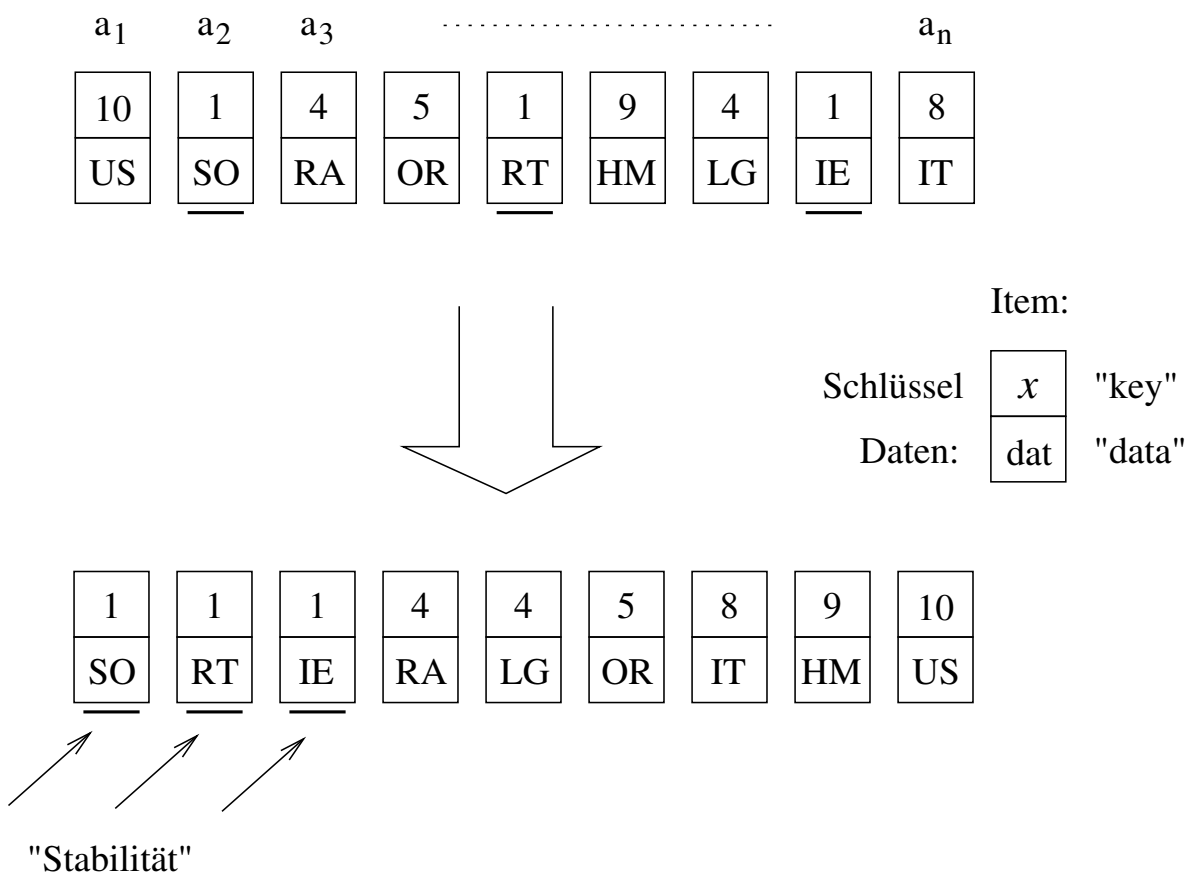
6. Kapitel

Sortieralgorithmen

Martin Dietzfelbinger

Juni 2021

6.1 Erinnerung, Grundbegriffe



Sortierproblem, abstrakt, Spezifikation: Siehe Kapitel 1.

Input: Tupel $x = (a_1, \ldots, a_n)$ von Objekten a_i
mit Schlüssel- und Datenteil: a_i`.key` und a_i`.data`.

Gegeben als **Array** `A[`$1..n$`]` oder als **lineare Liste**.

Achtung: In Bildern unterdrücken wir oft den Datenteil. Objekt a_i wird also einfach als Schlüssel dargestellt.

Schon gesehen: Sortieralgorithmus **Insertionsort**.

Bei Vorkommen gleicher Sortierschlüssel: Sortierverfahren heißt **stabil**, wenn Objekte mit identischen Schlüsseln in der Ausgabe in derselben Reihenfolge stehen wie in der Eingabe.

Maßzahlen für Sortieralgorithmen (in Abhängigkeit von n)**:**

1. **Laufzeit** (in O-Notation);
2. **Anzahl** der **Vergleiche** („*wesentliche Vergleiche*“)

$$\text{„}a_i\texttt{.key} < a_j\texttt{.key}\text{“} \quad \text{bzw. „}a_i\texttt{.key} \leq a_i\texttt{.key}\text{“};$$

3. Anzahl der Daten**verschiebungen** oder **Kopiervorgänge**
   (wichtig bei sehr umfangreichen Objekten);
4. der *neben* dem Array (bzw. der linearen Liste) benötigte **Speicherplatz**;
   wenn dieser nur $O(1)$, also von n unabhängig, ist:
   Algorithmus arbeitet „**in situ**“ oder „**in-place**“.

Insertionsort: Schlechtester Fall: $\frac{1}{2}n(n-1) \approx \frac{n^2}{2}$ Vergleiche, Zeit $\Theta(n^2)$.
Bester Fall: $n-1$ Vergleiche, Zeit $\Theta(n)$. **Durchschnittlicher** Fall: Zeit $\Theta(n^2)$. **Stabil, in situ.**

6.2 Mergesort

(engl. *merge* = vermengen, reißverschlussartig zusammenfügen)

Algorithmenparadigma oder **-schema**:

„Divide-and-Conquer“ (D-a-C, „teile und herrsche“)

Vorgehen eines D-a-C-Algorithmus $\mathcal{A}$ für Berechnungsproblem $\mathcal{P} = (\mathcal{I}, \mathcal{O}, f)$, auf Instanz $x \in \mathcal{I}$:

0) Trivialitätstest: Wenn $\mathcal{P}$ für x einfach direkt zu lösen ist, tue dies. – Sonst:

1) Teile: Zerteile x in (zwei oder mehr) Stücke $x_1, \ldots, x_a \in \mathcal{I}$.
(Wesentlich für Korrektheit/Terminierung: Jedes der Stücke ist „kleiner“ als x.)

2) Rekursion: Löse $\mathcal{P}$ für $x_1, \ldots, x_a$, separat, durch **rekursive** Verwendung von $\mathcal{A}$.
Teillösungen: $r_1, \ldots, r_a$.

3) Kombiniere: Baue aus x und $r_1, \ldots, r_a$ und $x_1, \ldots, x_a$ eine Lösung r für $\mathcal{P}$ auf x auf.

Mergesort ist D-a-C-Algorithmus für das Sortierproblem. – Input: $x = (a_1, \ldots, a_n)$.

Eine Trivialitätsschranke $n_0 \geq 1$ ist festgelegt.

0) Trivialitätstest: Falls $n \leq n_0$, sortiere x mit einem einfachen Algorithmus (z. B. Insertionsort). – Sonst:

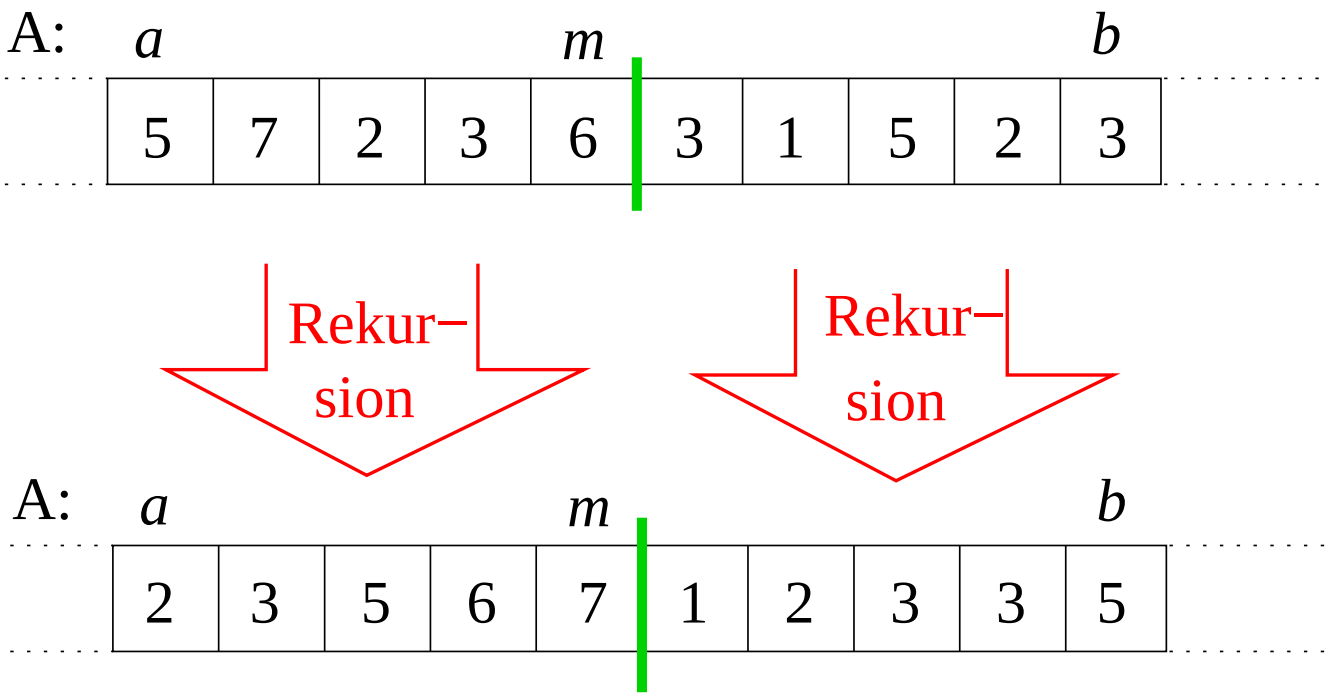
1) Teile: Setze $m := \lceil n/2 \rceil$.
Zerlege x in zwei Teilinputs x_1 (m Einträge) und x_2 ($n - m$ Einträge).

2) Rekursion: Sortiere x_1 **rekursiv**, Ergebnis r_1; sortiere x_2 **rekursiv**, Ergebnis r_2.

3) Kombiniere: „Mische“ die sortierten Folgen r_1 und r_2 zu einer sortierten Folge r zusammen. (Werden sehen: Aufwand ist $\leq n - 1$ Vergleiche.)

Für eine konkrete Formulierung nehmen wir an, dass Ein- und Ausgabe über ein Array `A[1..n]` erfolgen, und geben eine **rekursive** Prozedur **r_Mergesort** an, die ein durch seine Grenzen a und b ($1 \leq a \leq b \leq n$) gegebenes Teilarray von `A[1..n]` sortiert.

Rekursive **Prozedur r_Mergesort**(a, b):



1) Aufteilen von $\texttt{A}[a \ldots b]$ in $\texttt{A}[a \ldots m]$ und $\texttt{A}[m+1 \ldots b]$.

2) Rekursives Sortieren der beiden Segmente.

3) **Merge**(a, m, b).

Datenstruktur: Arrays $\texttt{A}[1..n]$ (Ein-/Ausgabe), $\texttt{B}[1..n]$ (Hilfsarray)

Prozedur r_Mergesort(a, b) // $1 \leq a \leq b \leq n$, sortiert $\texttt{A}[a..b]$

(1) **if** $b - a \leq n_0$ // $n_0 \geq 0$ ist vorausgesetzt

(2) **then** Insertionsort(a, b)

(3) **else** // Wissen: Länge von $\texttt{A}[a..b]$ ist ≥ 2

(4) $\texttt{m} \leftarrow a + \lfloor (b - a)/2 \rfloor$;

(5) **r_Mergesort**$(a, \texttt{m})$;

(6) **r_Mergesort**$(\texttt{m} + 1, b)$;

(7) **Merge**$(a, \texttt{m}, b)$ // fügt sortierte Teilfolgen zusammen;
// benutzt Hilfsarray $\texttt{B}[1..n]$

Prozedur Mergesort$(\texttt{A}[1..n])$ // sortiert $\texttt{A}[1..n]$

(1) **r_Mergesort**$(1, n)$.

Benötigt: Prozedur **Merge**(a, m, b), $1 \leq a \leq m < b \leq n$,
die folgendes tut:

Voraussetzung: $\texttt{A}[a\,..\,m]$ und $\texttt{A}[m+1\,..\,b]$ sind sortiert.

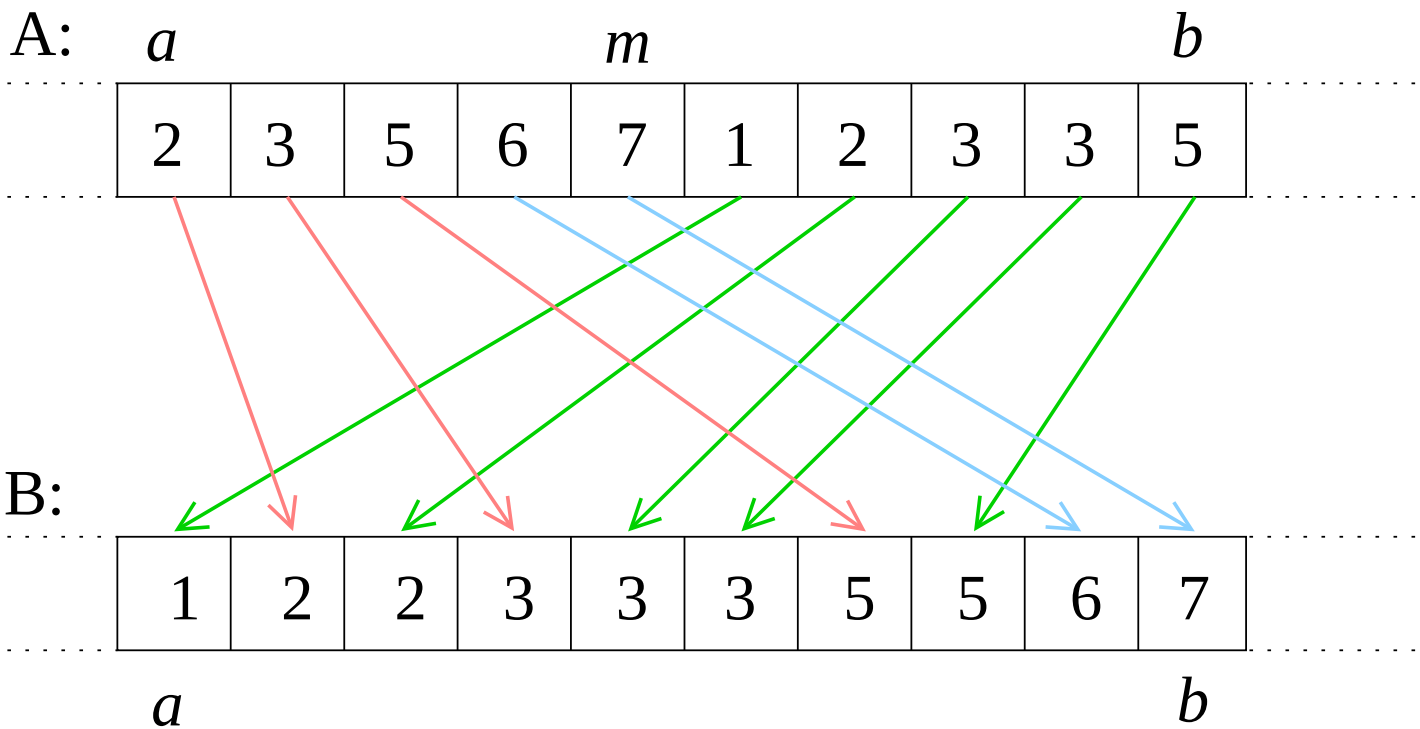
Resultat: $\texttt{A}[a\,..\,b]$ ist sortiert.

Ansatz: Quasiparalleler Durchlauf durch die Teilarrays:

„Reißverschlussverfahren“.

Benutze Hilfsarray $\texttt{B}[a\,..\,b]$ für das Ergebnis.

Vergleiche Kapitel 2, Folie 29: **union**(S_1, S_2) bei der Implementierung von **DynSets** mit sortierten Listen/Arrays.



„Mischen“ von $\texttt{A}[a \ldots m]$ und $\texttt{A}[m+1 \ldots b]$.

Rote Pfeile: `i`–`k`-Paare in Zeile (6).

Grüne Pfeile: `j`–`k`-Paare in Zeile (7).

Blaue Pfeile: `i`–`k`-Paare in Zeile (8), Kopieren.

Prozedur Merge(a, m, b)

```
          // für 1 ≤ a ≤ m < b ≤ n, sortiert A[a..b],
          // nimmt an, dass A[a..m] und A[m+1..b] sortiert sind.
(1)   i ← a;        // Index in A[a..m]
(2)   j ← m+1;      // Index in A[m+1..b]
(3)   k ← a;        // Index in B[a..b]
(4)   while i ≤ m and j ≤ b do
(5)     if A[i] ≤ A[j]
(6)       then B[k] ← A[i]; ++i; ++k
(7)       else B[k] ← A[j]; ++j; ++k
                  // verbleibendes Schlussstück kopieren:
(8)   while i ≤ m do B[k] ← A[i]; ++i; ++k;
(9)   while j ≤ b do B[k] ← A[j]; ++j; ++k;
                  // B[a..b] nach A[a..b] kopieren:
(10)  for k from a to b do A[k] ← B[k].
```

Eigenschaften von **Merge**(a, m, b):

- Das Verfahren ist korrekt, d. h.: Wenn die beiden Teile sortiert sind, enthält $\texttt{A}[a..b]$ dieselben Einträge wie vorher, in aufsteigend sortierter Reihenfolge.
- Es werden höchstens $b - a$ Schlüsselvergleiche ausgeführt. (Bei jedem Vergleich wird ein Schlüssel transportiert, nach Schleife (4)–(7) ist mindestens ein Eintrag übrig.)
- Der Zeitbedarf ist $\Theta(b - a)$;
- **Merge** ist **stabil**. (Vergleichsmodus Zeile (5) bevorzugt linkes Teilarray.)

Satz 6.2.1

Mergesort$(\texttt{A}[1..n])$ sortiert $\texttt{A}[1..n]$ stabil.

Beweis: Durch Induktion über rekursive Aufrufe von **r_Mergesort**(a, b) zeigt man, dass die Prozedur das Teilarray $\texttt{A}[a..b]$ stabil sortiert. Im Induktionsschritt benutzt man die eben aufgelisteten Eigenschaften von **Merge**. □

Zeitanalyse von Mergesort über „Rekurrenzgleichungen“. Zähle nur Vergleiche. Vor.: $n_0 = 0$.

$C(n) :=$ maximale Anzahl von Vergleichen bei Eingabelänge n.

$$C(1) = 0, C(2) = 1, C(3) = 3.$$
$$C(n) = C(\lceil n/2 \rceil) + C(\lfloor n/2 \rfloor) + n - 1. \qquad (*)$$

(Längen der Teilarrays: $\lceil n/2 \rceil$ und $\lfloor n/2 \rfloor$, **Merge** benötigt $\leq n - 1$ Vergleiche.)

Ausrechnen von Hand liefert:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
$C(n)$	0	1	3	5	8	11	14	17	21	25	29	33	37	41	...

Zunächst: Zweierpotenzen $n = 2^k$, $k \geq 0$.

n	1	2	4	8	16	32	64	...
$C(n)$	0	1	5	17	49	129	321	...

Geratene **Beobachtung:** $k \geq 0 \Rightarrow C(2^k) = (k-1) \cdot 2^k + 1$.

Beweis durch Induktion über k:
Für $k = 0$: $C(2^0) = (0-1) \cdot 2^0 + 1$.
Sein nun $k \geq 1$. Die I.V. sagt: $C(2^{k-1}) = (k-2) \cdot 2^{k-1} + 1$.
I.-Schritt: Nach Rekurrenz: $C(2^k) = 2C(2^{k-1}) + 2^k - 1$.
Nach I.V.: $C(2^k) = 2((k-2) \cdot 2^{k-1} + 1) + 2^k - 1 = (k-1) \cdot 2^k + 1$.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
$C(n)$	0	1	3	5	8	11	14	17	21	25	29	33	37	41	...

Zwischen 2^{k-1} und 2^k jeweils Zuwachs um k.

Damit **Vermutung**: Für $2^{k-1} < n \leq 2^k$, d. h. $k = \lceil \log n \rceil$:

$$C(n) = (k-2)2^{k-1} + 1 + (n - 2^{k-1})k = nk - 2^k + 1.$$

Satz 6.2.2

$C(n) = n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1$, für $n \geq 1$.

Beweis: Durch Induktion über $k \geq 1$ zeigen wir: Für $2^{k-1} \leq n \leq 2^k$ gilt $C(n) = nk - 2^k + 1$.

I.A.: $k = 1$. Für $n = 1$ gilt $C(1) = 0 = 1 \cdot 1 - 2^1 + 1$ und $C(2) = 1 = 2 \cdot 1 - 2^1 + 1$.
Nun sei $k \geq 2$ gegeben; **I.V.:** Beh. gilt für $k - 1$.

I.-Schritt:

1. Fall: $n = 2^k$. Nach Beobachtung: $C(n) = 2^k(k-1) + 1 = nk - 2^k + 1$.

2. Fall: $n = 2^{k-1}$. Nach I.V. gilt $C(n) = 2^{k-1}(k-1) - 2^{k-1} + 1 = 2^{k-1} \cdot k - 2^k + 1 = nk - 2^k + 1$.

3. Fall: $2^{k-1} < n < 2^k$.

Es gilt $2^{k-2} < n/2 < 2^{k-1}$, also $2^{k-2} \leq \lfloor n/2 \rfloor \leq \lceil n/2 \rceil \leq 2^{k-1}$.

Nach **I.V.** haben wir also

$$C(\lceil n/2 \rceil) = \lceil n/2 \rceil \cdot (k-1) - 2^{k-1} + 1 \quad \text{und} \quad C(\lfloor n/2 \rfloor) = \lfloor n/2 \rfloor \cdot (k-1) - 2^{k-1} + 1.$$

Aus der Rekurrenzgleichung $(*)$ erhalten wir damit:

$$\begin{aligned} C(n) &= C(\lceil n/2 \rceil) + C(\lfloor n/2 \rfloor) + n - 1 \\ &= \lceil n/2 \rceil (k-1) - 2^{k-1} + 1 + \lfloor n/2 \rfloor (k-1) - 2^{k-1} + 1 + n - 1 \\ &= n(k-1) + n - 2^k + 1 = nk - 2^k + 1. \end{aligned}$$

□

Korollar 6.2.3

$C(n) \leq n \log n$, für $n \geq 1$.

Beweis: Wenn $n = 2^k$ ist, gilt nach Satz 6.2.2:

$C(n) = 2^k \cdot k - 2^k + 1 \leq nk = n \log n$.

Wenn $2^{k-1} < n < 2^k$ ist für ein $k \geq 1$, dann gilt:

$nk - 2^k + 1 = n(k-1) + (n - 2^k + 1) \leq n \log n$. □

Korollar 6.2.4

Die Rechenzeit von Mergesort ist $\Theta(n \log n)$.

Variante 1: (Sollte man immer anwenden!)

In **r_Mergesort** kleine Teilarrays ($b - a \leq n_0$) durch (z. B.) **Insertionsort** sortieren.

Vermeidet überproportional hohen Zeitaufwand für viele rekursive Aufrufe bei kleinen Teilarrays.
Das optimale n_0 auf einer konkreten Maschine kann ein Experiment liefern.
($n_0 = 8$ ist keinesfalls zu groß.)

Variante 2: Mergesort für lineare Listen.

Wenn Liste L Länge 1 hat: fertig; sonst teile L in Teillisten L_1, L_2 der halben Länge.

(Durchlaufe L, hänge die Einträge abwechselnd an L_1 und L_2 an. Kosten: Θ(Länge von L).)

Auf L_1 und L_2 wird der Algorithmus rekursiv angewendet.

Dann werden die beiden nun sortierten Listen „gemischt“, analog **Merge**.

Analyse der Vergleiche: Identisch zum Obigen.

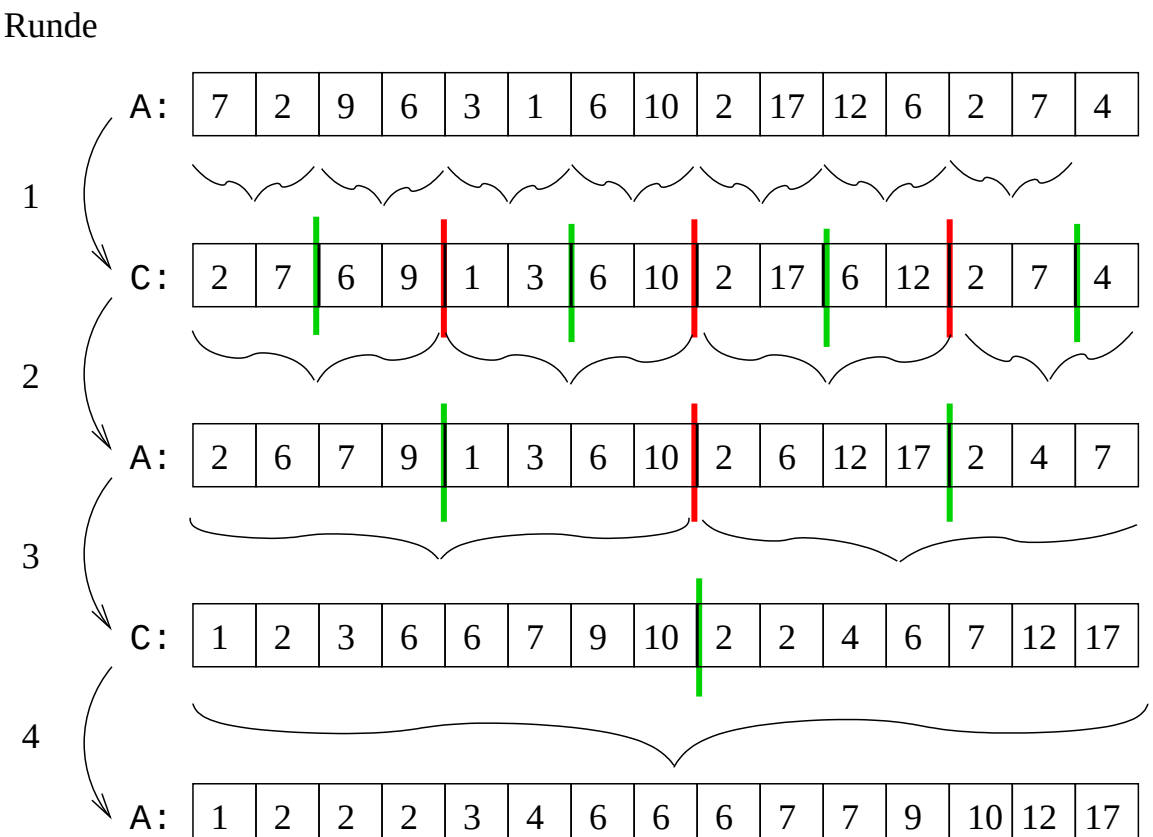
Laufzeit: Proportional zur Anzahl der Vergleiche, also auch $\Theta(n \log n)$.

Kein zusätzlicher Platz (aber Zeiger für die Liste).

Variante 3: Iteratives Mergesort, „bottom-up“

Idee: Arbeite in Runden $i = 1, 2, 3, \ldots, \lceil \log n \rceil$.

Beispiel:



Variante 3: Iteratives Mergesort, „bottom-up“

Runden $i = 1, \ldots, \lceil \log n \rceil$.

Vor Runde i: Teilarrays $\mathtt{A}[(l-1) \cdot 2^{i-1} + 1 \ldots l \cdot 2^{i-1}]$ der Länge 2^{i-1} sind sortiert. (Eventuell ist das am weitesten rechts stehende Teilarray unvollständig.)

Runde i: **merge**$((l-2) \cdot 2^{i-1} + 1, (l-1)2^{i-1}, l \cdot 2^{i-1})$ für $l = 2, 4, 6, \ldots, 2\lceil n/2^i \rceil$,

mit Hilfe eines zweiten Arrays B.

(Mögliche Sonderrolle für unvollständige Teilarrays ganz rechts.)

Trick, spart Kopierarbeit: Benutze 2 Arrays, die in jeder Runde ihre Rolle wechseln.

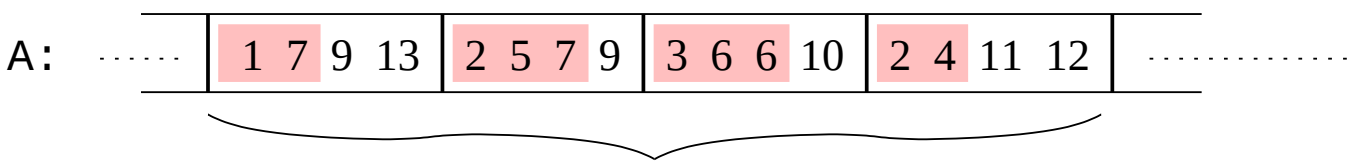
Zeitverhalten von iterativem Mergesort: wie bei rekursivem Mergesort.

Alternative, einfache Zeitanalyse für diese Variante:
Jede Runde kostet offenbar $< n$ Vergleiche und Zeit $\Theta(n)$, und es gibt $\lceil \log n \rceil$ Runden. Also ist die die Zahl der Vergleiche $< n\lceil \log n \rceil$ und die Gesamtzeit $\Theta(n\lceil \log n \rceil) = \Theta(n \log n)$.

Variante 4: Mehrweg-Mergesort

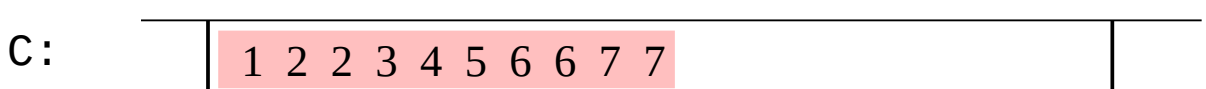
Besonders in Kombination mit dem iterativen Verfahren interessant ist die Möglichkeit, in einer `merge`-Runde nicht nur ein Array, sondern 3, 4, oder viel mehr (k viele) Teilarrays zusammenzumischen.

(Unrealistisch kleines) *Beispiel*:



4−way merge: wähle kleinstes Element aus 4 (Rest−)Arrays.

fertig



Die nächsten gewählten Elemente: "9" aus 1.Teilarray, "9" aus 2. Teilarray, "10", "11", etc.

Benötigt: Auswahl des kleinsten „aus k“. Dafür: Heap (später).

Beim Sortieren von Daten auf Hintergrundspeichern wie **Festplatten** oder **Solid-State-Disks** hat diese Variante gegenüber dem Mischen von zwei Teilfolgen Laufzeitvorteile, da die Anzahl von (zeitaufwendigen) Seitenfehlern (*page faults*) verringert wird.

Daher Empfehlung: Für Sortieren von großen Datenmengen auf Hintergrundspeichern: Mergesort-artige Verfahren benutzen!

Ein ähnlicher Effekt tritt auch beim Sortieren im Hauptspeicher mit k-Wege-Mischen ein: Die Zahl der Cachefehler (*cache misses*) sinkt.

Details, für Interessierte: Ein (Riesen-)Array (n Einträge) im Hintergrundspeicher soll sortiert werden.
Wir nehmen an, dass im Hauptspeicher für den Sortieralgorithmus Platz für M viele Elemente zur Verfügung steht. Mit einem Transport können wir B viele Elemente vom Hintergrundspeicher holen oder dorthin schreiben. B heißt die Seiten- oder Blockgröße, in der „Einheit“ Arrayeinträge.
Wir sortieren zunächst Arraysegmente der Länge M mit einem „internen“ Verfahren, z. B. gewöhnlichem Mergesort, schreiben diese wieder auf den Hintergrundspeicher, das ergibt $2n/B$ viele Blocktransporte.
Dann wählen wir $k = M/B$, mischen also in einer Runde immer M/B viele sortierte Arraysegmente derselben Länge L zu einem Segment der Länge $L(M/B)$ zusammen. (k kann nicht größer als M/B sein, weil für jedes der bearbeiteten k Segmente mindestens ein Block im Hauptspeicher liegen muss.) In jeder Runde werden n/B Blöcke aus dem Hintergrundspeicher geholt und wieder dorthin geschrieben.
Was ist die Anzahl r der Runden im zweiten Teil? Es muss $M \cdot (M/B)^r \approx n$ sein, das heißt $r \approx \log_{M/B}(n/M) = \mathbf{\log(n/M)/\log(M/B)}$.
Dieser $(\log(n/M)/\log(M/B))$-malige Transport aller Daten aus dem Hintergrundspeicher in den Hauptspeicher und zurück kostet die Übertragung von $2(n/B)\log(n/M)/\log(M/B)$ vielen Blöcken. Wenn man naiv wie in gewöhnlichem Mergesort immer nur zwei Segmente mischt, erhält man $\mathbf{\log(n/M)}$ Runden und $2(n/B)\log(n/M)$ übertragene Blöcke. Der eingesparte Faktor $\log(M/B)$ kann erheblich sein! ($M = 10^6$ und $B = 1000$ ist nicht unrealistisch. Dann ist $\log(M/B) = 10$.) Der eingesparte Faktor hängt nicht von n ab.

Vorlesungsvideo:

Quicksort

6.3 Quicksort

Schnelles Sortieren im Durchschnitt/Erwartungswert [Hoare, 1960]

Standard-Sortierverfahren, bereitgestellt in praktisch allen Programmiersprachen, für Sortieren von Daten mit zahlenwertigen Schlüsseln im Hauptspeicher vorzugsweise zu benutzen.

Folgt ebenso wie Mergesort dem

Divide-and-Conquer-Ansatz

Zentrale Teilaufgabe:

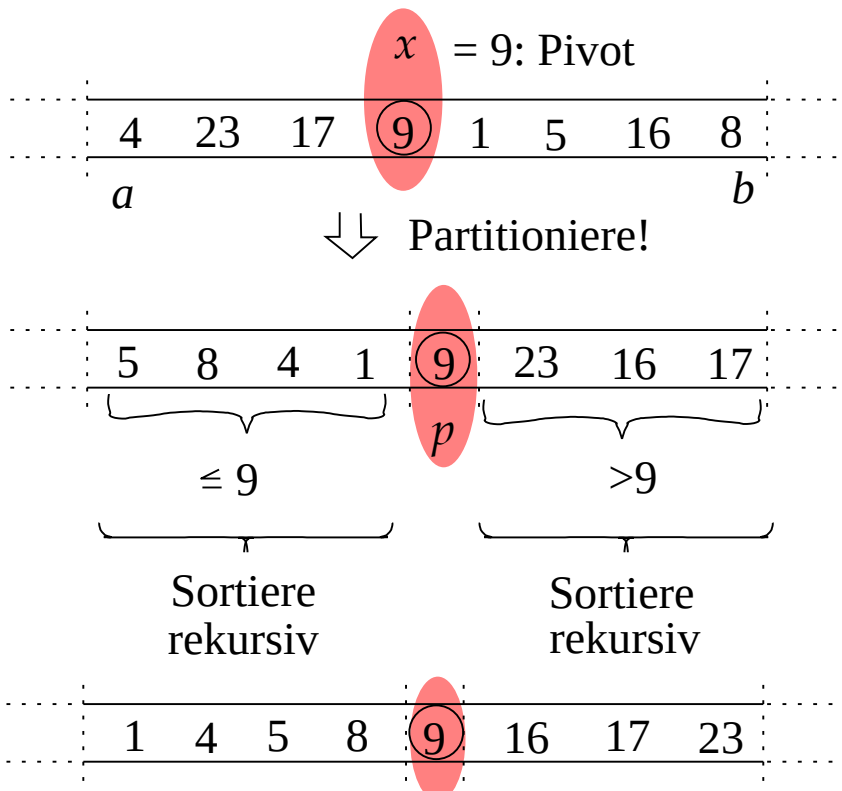
Sortiere (Teil-)Array $\texttt{A}[a\,..\,b]$, wo $1 \leq a \leq b \leq n$.

Zu erledigen von einer **(rekursiven)** Prozedur **r_qsort**(a, b).

Kernstück: Partitionieren und rekursiv sortieren.

Quicksort−Schema:

Teilaufgabe: Sortiere A[*a*...*b*]



Ansatz für **r_qsort**(a, b), gleichzeitig Korrektheitsbetrachtung:

0) Trivialitätstest: Wenn $a \geq b$, ist nichts zu tun.

(Oder: Wenn $b - a < n_0$: **Insertionsort**(a, b).)

1) Teile: (Hier „**partitioniere**“)
Wähle „**Pivotelement**“ $x \in \{\texttt{A}[a]\texttt{.key}, \ldots, \texttt{A}[b]\texttt{.key}\}$.
Arrangiere die Einträge von $\texttt{A}[a\,..\,b]$ so um, dass für ein p mit $a \leq p \leq b$ gilt:
$\texttt{A}[p]\texttt{.key} = x$ und $\texttt{A}[a]\texttt{.key}, \ldots, \texttt{A}[p-1]\texttt{.key} \leq x < \texttt{A}[p+1]\texttt{.key}, \ldots, \texttt{A}[b]\texttt{.key}$.

2) Rekursion auf Teilinstanzen $x_1 = \texttt{A}[a\,..\,p-1]$ und $x_2 = \texttt{A}[p+1\,..\,b]$:
r_qsort$(a, p-1)$; **r_qsort**$(p+1, b)$.

Nach I.V. gilt nun: $\texttt{A}[a\,..\,p-1]$, $\texttt{A}[p+1\,..\,b]$ sortiert.

3) Kombiniere: Es ist nichts zu tun: $\texttt{A}[a\,..\,b]$ ist sortiert.

Kernstück: 2) Partitionieren.

Möglichkeiten für Wahl des „**Pivotelements**“ x:

- `A[`a`].key` (**erstes** Element des Teilarrays)
- `A[`b`].key` (**letztes** Element des Teilarrays)
- `A[`$a + \lfloor (b-a)/2 \rfloor$`].key` (**Mitte** des Teilarrays)
- „**Clever Quicksort**“: Das Pivotelement x ist der **Median** der drei Schlüssel `A[`a`].key`, `A[`b`].key`, `A[`$a + \lfloor (b-a)/2 \rfloor$`].key`
- „**Randomisiertes Quicksort**“:
  Wähle einen der $b - a + 1$ Einträge in `A[`a`..`b`]` **zufällig**.

Allgemein: Wähle ein s mit $a \leq s \leq b$, **vertausche** `A[`a`]` mit `A[`s`]`.

Das Pivotelement ist danach immer `A[`a`].key`.

Partitionieren mit Pivotelelement $x = \texttt{A}[a]\texttt{.key}$

Klar: $b - a$ Vergleiche genügen, um Schlüssel $\leq x$ „nach links“ und Schlüssel $> x$ „nach rechts“ zu schaffen und den Eintrag mit dem Pivotelement an die Stelle zwischen den Bereichen zu schreiben.

Eine sehr geschickte Methode wurde schon von **C. A. R. Hoare**[1] angegeben:
kein zusätzlicher Platz, keine unnötige Datenbewegung.

Hier: Variante des in der AuP-Vorlesung angegebenen Verfahrens.

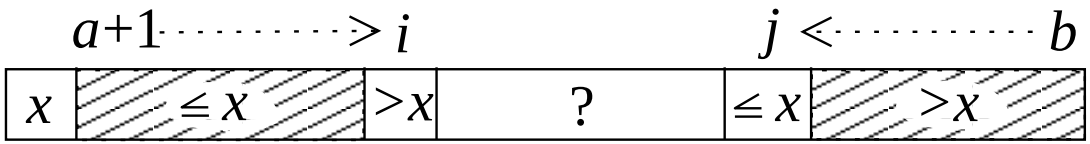
---

[1] **C. A. R. Hoare** (*1934), brit. Informatiker, erfand Quicksort & Korrektheitskalkül („Hoare-Kalkül“).

„I conclude that there are two ways of constructing a software design: One way is to make it *so simple* that there are *obviously no* deficiencies and the other way is to make it *so complicated* that there are *no obvious* deficiencies. The first method is far more difficult.“ (Dankesrede für den Turingpreis 1980)

„I think Quicksort is the only really interesting algorithm that I've ever developed.“

Quelle: Wikipedia, `https://de.wikipedia.org/wiki/Tony_Hoare#Zitate`

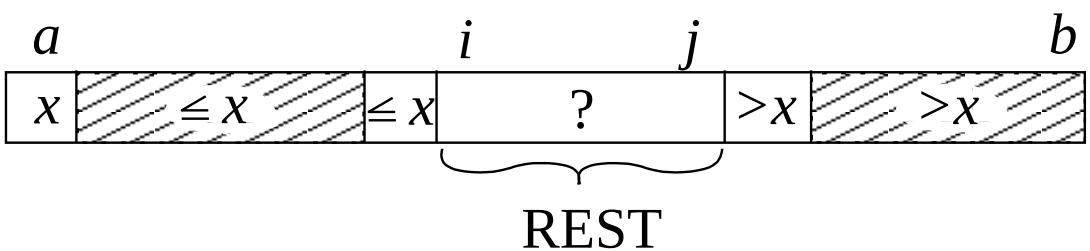


Ignoriere `A[`a`]`.

Lasse ab $a + 1$ einen „Zeiger“ (Index) i nach rechts wandern, bis ein Schlüssel $> x$ erreicht wird.

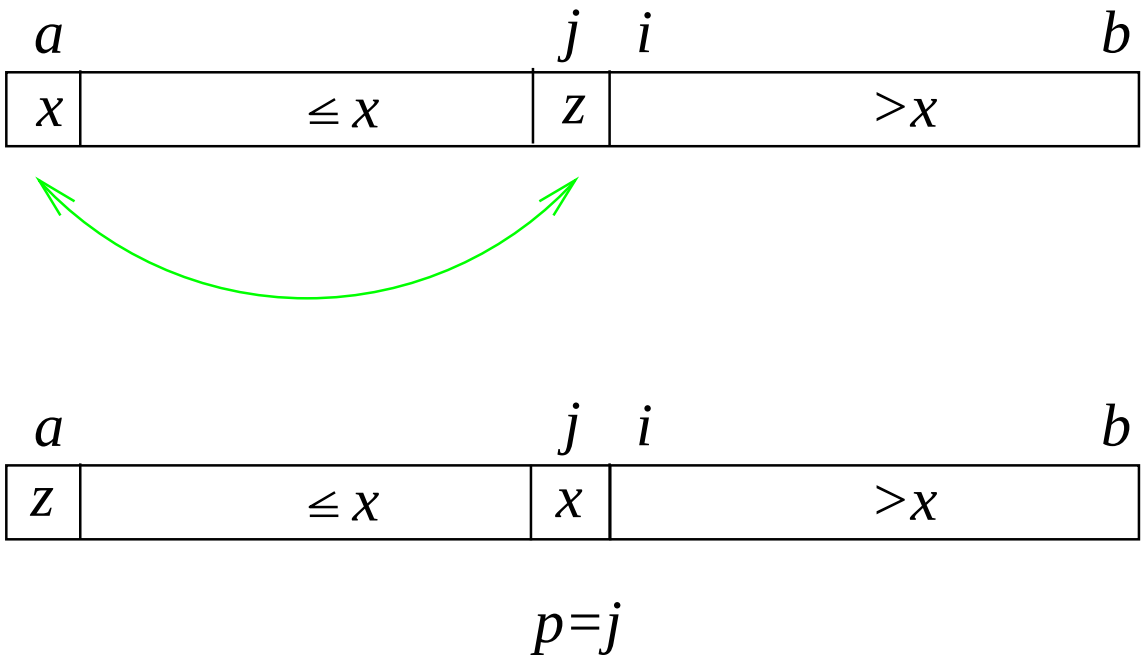
Lasse ab b einen „Zeiger“ j (Index) nach links wandern, bis ein Schlüssel $\leq x$ erreicht wird.

Falls $i < j$: Vertausche `A[`i`]` und `A[`j`]`.



Erhöhe i um 1, erniedrige j um 1. Wiederhole . . .

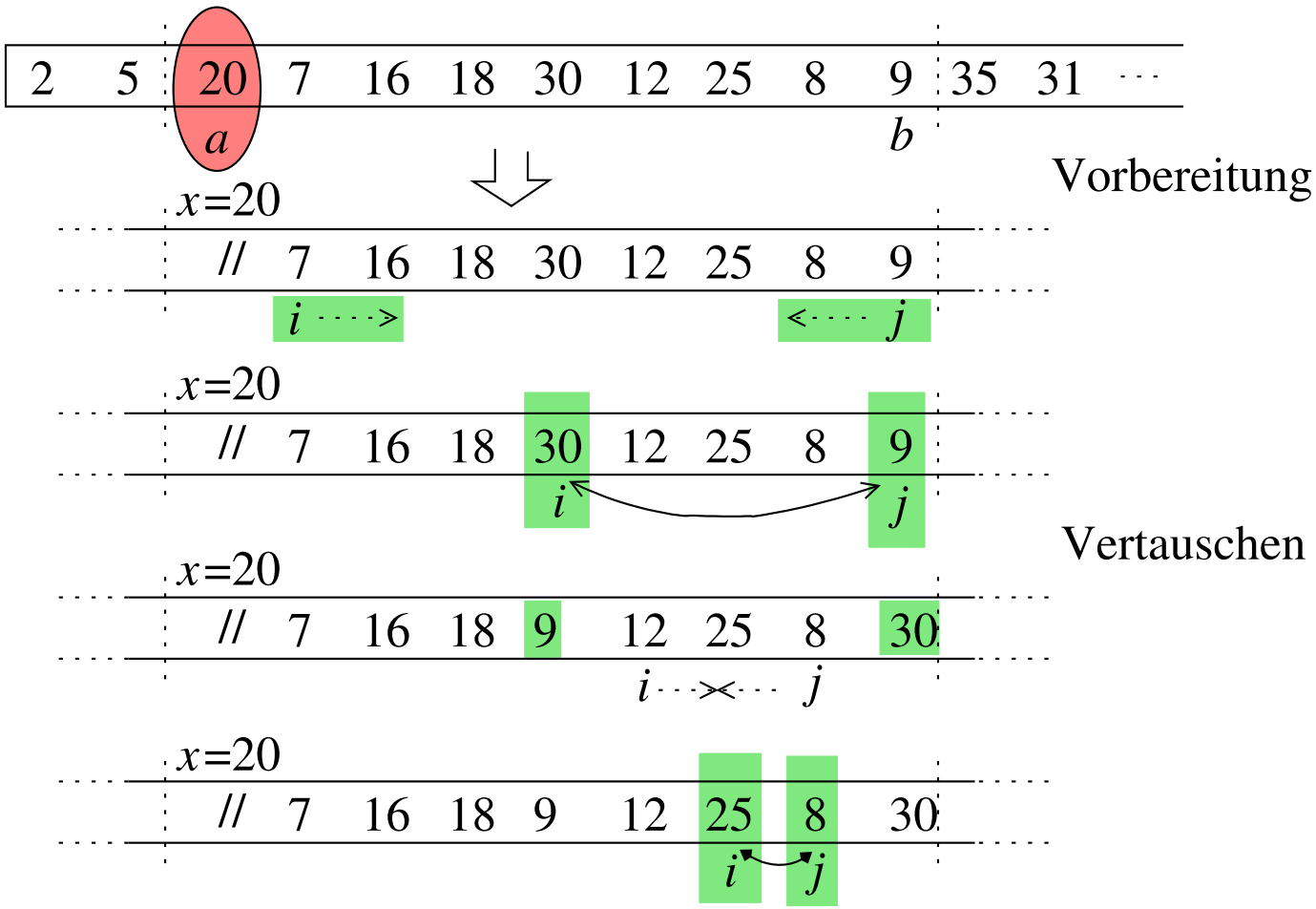
. . . bis sich i und j „treffen“ und „überkreuzen“, d. h. bis $i = j + 1$ gilt.

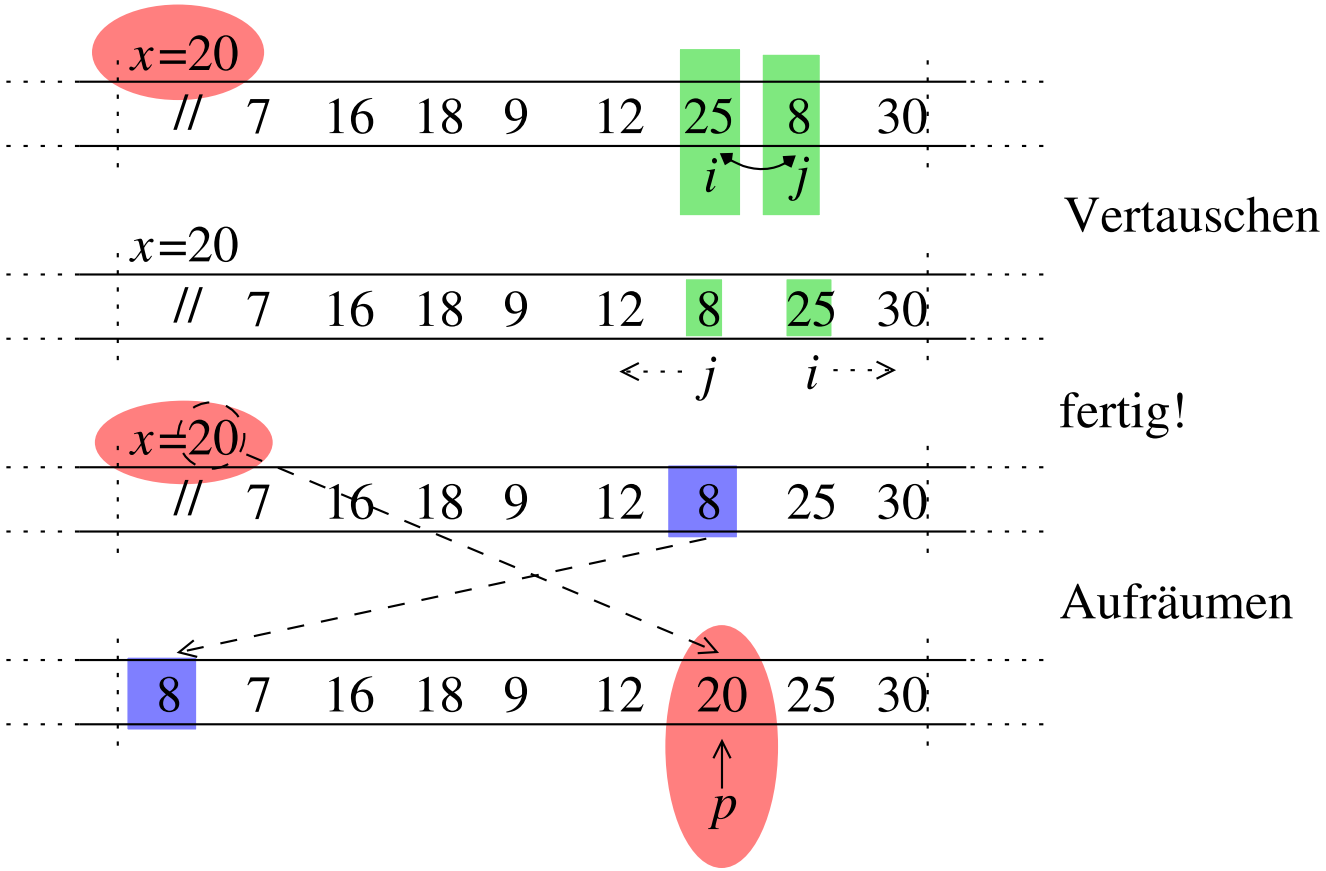


Es gilt jetzt: $\texttt{A}[a+1]\texttt{.key}, \ldots, \texttt{A}[j]\texttt{.key} \leq x < \texttt{A}[j+1]\texttt{.key}, \ldots, \texttt{A}[b]\texttt{.key}$.

Nun vertausche $\texttt{A}[a]$ mit $\texttt{A}[j]$. Position p des Pivotelements x ist jetzt $\texttt{A}[j]$.

Beispiel für partition:





Sorgfalt bei der Implementierung der Idee ist nötig, um Randfälle und den Abschluss korrekt zu behandeln und um unnötige Vergleiche zu verhindern. Nach einigem Tüfteln ergibt sich Folgendes.

Prozedur partition$(a, b, \mathtt{p})$ // Partitionierungsprozedur in **r_qsort**(a, b)

// Voraussetzung: $a < b$; Pivotelement: $\mathtt{A}[a]$

```
(1)   x ← A[a].key;
(2)   i ← a + 1;  j ← b;
(3)   while i ≤ j do     // noch nicht fertig; Hauptschleife
(4)     while i ≤ j and A[i].key ≤ x do ++i;
(5)     while j > i and A[j].key > x do --j;
(6)     if i = j then --j;      // Wissen: A[i].key > x
(7)     if i < j then
(8)       vertausche A[i] und A[j];
(9)       ++i;  --j;
(10)  if a < j then vertausche A[a] und A[j];
(11)  p ← j;    // Rückgabewert
```

Achtung: Diese Implementierungsdetails vermeiden Mehrfachvergleiche zwischen Schlüsseln.

Lemma 6.3.1 (Korrektheit von **partition**)

Ein Aufruf **partition**$(a, b, \mathtt{p})$, für $1 \leq a < b \leq n$, bewirkt Folgendes:

Sei $x = \mathtt{A}[a]\,\mathtt{.key}$ (vor Aufruf), p der Inhalt von $\mathtt{p}$ (nach Aufruf).

(a) $\mathtt{A}[a\,..\,b]$ wird so umarrangiert, dass $x = \mathtt{A}[p]\,\mathtt{.key}$ und
$\mathtt{A}[a+1]\,\mathtt{.key}, \ldots, \mathtt{A}[p-1]\,\mathtt{.key} \leq x < \mathtt{A}[p+1]\,\mathtt{.key}, \ldots, \mathtt{A}[b]\,\mathtt{.key}$.

(b) An welche Stelle die Einträge $\mathtt{A}[j]$, $a \leq j \leq b$, gebracht werden, hängt nur von der Menge $\{i \in [a+1, b] \mid \mathtt{A}[i]\,\mathtt{.key} > x\}$ ab, nicht von den konkreten Schlüsselwerten.

Beweis: (a) Sorgfältige, etwas mühselige Fallunterscheidung, nur für Unerschrockene auf den nächsten Folien durchexerziert. (b) Alle Entscheidungen werden nur aufgrund von Vergleichen mit x getroffen.□

Beweis von (a): **(Nicht prüfungsrelevant.)**
Wir beobachten die folgende **Schleifeninvariante** für die **Hauptschleife** (3)–(9)

für a, b und die Inhalte i, j, x von `i`, `j`, `x`:

Unmittelbar vor Beginn von Zeile (3) gilt:

(I) $a \leq i - 1 \leq j \leq b$,
(IIa) $\texttt{A}[a+1]\texttt{.key}, \ldots, \texttt{A}[i-1]\texttt{.key} \leq x$ und
(IIb) $\texttt{A}[j+1]\texttt{.key}, \ldots, \texttt{A}[b]\texttt{.key} > x$.

Beweis durch Induktion über Schleifendurchläufe, gleich.

Beweis der Korrektheit der Prozedur **partition**:

Die Hauptschleife bricht ab, wenn in Zeile (3) $j < i$ gilt.

Wegen (I) gilt dann $j = i - 1$ und damit wegen (IIa,b) auch
$\texttt{A}[a+1], \ldots, \texttt{A}[j] \leq x$ und $\texttt{A}[j+1], \ldots, \texttt{A}[b] > x$.
Wenn $j = a$ ist, gibt es also außer $\texttt{A}[a]$ keine Einträge mit Schlüssel $\leq x$, es ist nichts zu tun.
Sonst stellt die Vertauschung in Zeile (10) die verlangte Ausgabe her.

Beweis der **Invariante**: Induktion über Schleifendurchläufe.

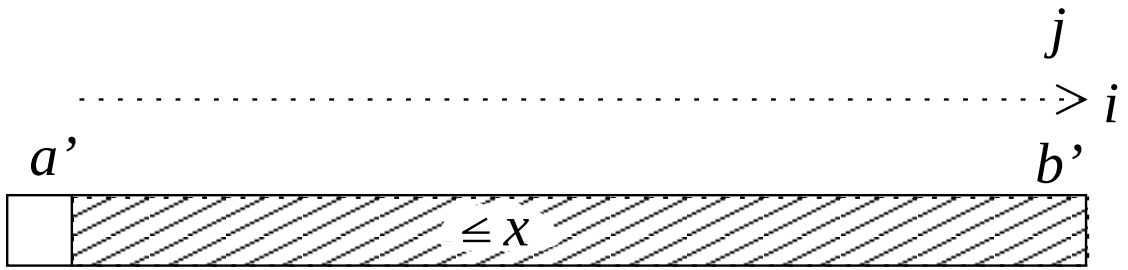
I.A.: Zu Beginn gilt die Invariante: (I) ist klar, und (IIa,b) sind Aussagen über die leere Menge.

I.-Schritt: Betrachte einen kompletten Schleifendurchlauf, unter der Annahme, dass anfangs die Invariante gilt.

Wir können $i \leq j$ annehmen (sonst bricht die Schleife ab).

Setze $a' := i - 1$ und $b' := j$. Dann gilt $a \leq a' < b' \leq b$.
Aus der Art, wie in Zeilen (4) und (5) die Variablen `i` und `j` behandelt werden, sieht man, dass die Einträge in
$\texttt{A}[a+1\,..\,a']$ und in $\texttt{A}[b'+1\,..\,b]$ in der Schleife weder gelesen noch verändert werden.

Wir können also so tun, als ob wir im ersten Schleifendurchlauf wären, mit Teilarray $\texttt{A}[a'+1\,..\,b']$.



1. Fall: Die Schleife (4) bricht ab, weil $i > j$ wird.
Dann ist offenbar jetzt $i - 1 = j = b'$
(also gilt (I)) und $\texttt{A}[a' + 1], \ldots, \texttt{A}[b'] \leq x$,
weil dies in (4) getestet wurde.

Also gilt (IIa,b) für die neuen Werte i und j.

Die Tests in Zeilen (5), (6), (7) ergeben jeweils $false$, die Anweisungen in Zeilen (5)–(9) werden nicht ausgeführt.

(Die Hauptschleife wird nun nicht mehr ausgeführt.)

2. Fall:

Die Schleife (4) bricht mit $i \leq j$ ab, weil

$$\texttt{A}[i]\texttt{.key} > x \tag{$*1$}$$

ist.

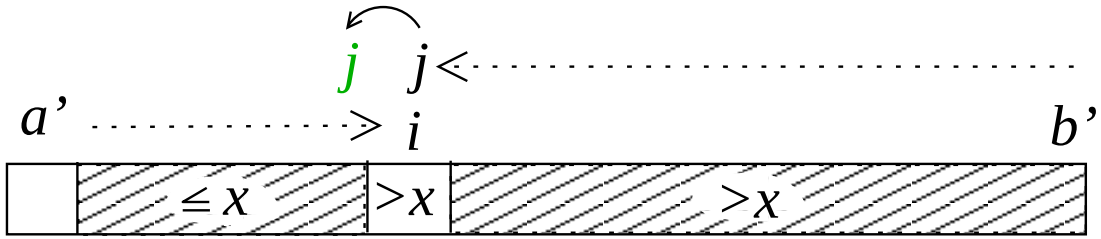
Dann gilt

$$\texttt{A}[a'+1]\texttt{.key}, \ldots, \texttt{A}[i-1]\texttt{.key} \leq x, \tag{$*2$}$$

weil dies getestet worden ist.

Beobachte auch:

$$a'+1 \leq i \leq j \leq b'. \tag{$*3$}$$



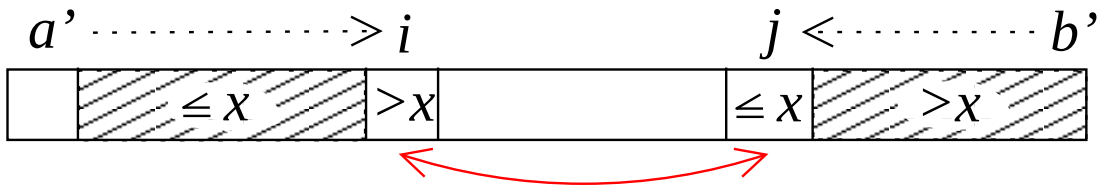
Unterfall 2a: Die Schleife in (5) bricht ab, weil $i = j$ wird.
Dann gilt (weil dies getestet wurde):

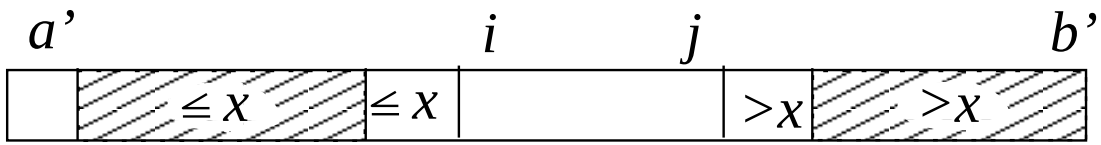
(∗4) $$\texttt{A}[j+1]\texttt{.key}, \ldots, \texttt{A}[b']\texttt{.key} > x$$

und $\texttt{A}[j]\texttt{.key} > x$ (wegen **(∗1)**). Also gilt (IIb) auch nach Zeile (6).
Weil sich i nicht ändert, haben wir (IIa) wegen **(∗2)**.

Zeile (6) verringert j um 1, so dass nun $a' \leq j$, also (I) gilt.

(Der nächste Test in Zeile (3) beendet die Hauptschleife.)





Unterfall 2b: Die Schleife in (5) bricht mit $i < j$ ab, weil

(∗5) $\qquad$ `A[`j`].key` $\leq x$

gilt. Wegen **(∗3)** gilt nach der Veränderung der Indizes in Zeile (9):

$a' \leq j$ und $i - 1 \leq j < b'$, also (I).

(IIa), (IIb) folgen ebenfalls unter Verwendung von **(∗1)**, **(∗2)**, **(∗4)**.

Dies beendet den Beweis der Schleifeninvarianten. □

Lemma 6.3.2 (Aufwand von **partition**)

Bei Aufruf **partition**$(a, b, \mathtt{p})$ gilt:

(a) Die Anzahl der Schlüsselvergleiche ist exakt $b - a$;

(b) der (Zeit-)Aufwand ist $\Theta(b - a)$.

Beweis: Sei i der Inhalt von `i`, j der Inhalt von `j`.

(a) Schlüsselvergleiche finden nur statt, so lange $i < j$ gilt. Nach jedem Vergleich „`A[`i`].key` $\leq x$“ wird i um 1 erhöht (Zeile (4) oder (9)),
nach jedem Vergleich „`A[`j`].key` $> x$“ wird j um 1 erniedrigt (Zeile (5) oder (9)).

In der Situation $i \geq j$ wird kein Vergleich mehr ausgeführt. Daher kann kein Eintrag zweimal mit x verglichen werden.

Prozedur r_qsort(a, b)

// Rekursive Prozedur im Quicksort-Algorithmus, $1 \leq a \leq n+1$, $0 \leq b \leq n$.

(1) **if** $a < b$ **then**

(2) $\mathtt{s} \leftarrow$ ein Element von $\{a, \ldots, b\}$;
// Es gibt verschiedene Methoden, um s zu wählen; s. Folie 22

(3) vertausche $\mathtt{A}[a]$ und $\mathtt{A}[\mathtt{s}]$;

(4) **partition**$(a, b, \mathtt{p})$; // p: Ausgabeparameter

(5) **r_qsort**$(a, \mathtt{p}-1)$;

(6) **r_qsort**$(\mathtt{p}+1, b)$.

Effiziente Variation zum Abbruchkriterium in Zeile (1):
Für ein passendes $n_0 \geq 2$ Teilarrays der Länge $\leq n_0$ mit **Insertionsort**$(\mathtt{A}[a..b])$ sortieren.

Lemma 6.3.3 Ein Aufruf von **r_qsort**(a, b) sortiert $\mathtt{A}[a\,..\,b]$. (Beweis: s. Folie 21.)

Um $\mathtt{A}[1\,..\,n]$ zu sortieren:

Prozedur Quicksort$(\mathtt{A}[1\,..\,n])$

(1) **r_qsort**$(1, n)$.

Rechenzeit von Quicksort(A[1..n]):

Aufwand für einen Aufruf **r_qsort**(a, b) ist $O(1)$

plus der Aufwand für **partition**(a, b, p), proportional zu $b - a$ (falls $a < b$)

plus der Aufwand für die rekursiven Aufrufe (falls $a < b$).

$b - a$ ist genau die Anzahl der Schlüsselvergleiche, die von **partition**(a, b, p) ausgeführt werden.

Die Aufrufe **partition**(a, b, p) mit $a < b$ haben verschiedene Pivotelemente, also gibt es davon $\leq n$ viele. Jeder Aufruf mit $a < b$ generiert höchstens zwei Aufrufe für ein leeres oder ein einelementiges Teilarray, also gibt es davon $\leq 2n$ viele.

Die Gesamtrechenzeit für **Quicksort**(A[1..n]) ist also:

$$O(n) + O(\underbrace{\text{Gesamtanzahl aller Schlüsselvergleiche}}_{=: V}).$$

V hängt von der Anordnung der Eingabe ab! (Bei randomisiertem QS zusätzlich vom Zufall.)

Schlechtester Fall („worst case“):

Wir überlegen: Wenn zwei Schlüssel verglichen werden, ist einer das Pivotelement, der andere wandert in eines der beiden Teilarrays.

$\Rightarrow$ Es werden niemals dieselben Schlüssel später nochmals verglichen.

Also gilt immer: $V \leq \binom{n}{2} = \frac{1}{2}n(n-1)$.

Der Wert $V = \binom{n}{2}$ entsteht zum Beispiel, wenn die Eingabe schon sortiert ist und man in **r_qsort**(a, b) immer `A[`a`]` als Pivotelement wählt.

(Dann ist das linke Teilarray leer, das rechte enthält $b - a$ Elemente.
Die Rekursionstiefe ist n, und $V = (n-1) + (n-2) + \cdots + 3 + 2 + 1 = \binom{n}{2}$.)

Achtung: Ein ähnlicher Effekt tritt ein, wenn das Array „fast“ sortiert ist.

Durchschnittlicher/erwarteter **Fall**:

Zur Vereinfachung nehmen wir an:
Eingabeschlüssel sind n verschiedene Zahlen $x_1 < \cdots < x_n$.

Wir machen die folgende **Wahrscheinlichkeitsannahme**:

> Jede Anordnung der Eingabeschlüssel hat dieselbe Wahrscheinlichkeit.

Es gibt $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$ viele solche Anordnungen.

Als Pivotelement wählen wir den Eintrag an einer festen Stelle, z. B. `A[`a`]`.

Unter dieser Wahrscheinlichkeitsannahme wird V zu einer **Zufallsgröße**.

Wir untersuchen den Erwartungswert/Mittelwert von V.

(**Vorsicht!** Die folgende Überlegung gilt nicht für „Clever-Quicksort“ und nicht, wenn die Schlüssel nicht alle verschieden sind.)

$C_n :=$ die erwartete Anzahl der Vergleiche beim Sortieren eines Teilarrays mit n Einträgen mit **r_qsort** (unter der Annahme „zufällige Anordnung von $x_1 < \cdots < x_n$ in der Eingabe“).

$C_0 = 0$, (klar)
$C_1 = 0$, (klar) $C_2 = 1$, (ein Vergleich mit einem Pivot, dann fertig).

C_3: Wenn man beim ersten **r_qsort**(a, b)-Aufruf mit $b = a + 2$ das mittlere Element x_2 als Pivot hat, gibt es 2 Vergleiche insgesamt.
Wahrscheinlichkeit hierfür: $\frac{1}{3}$.

Wenn man x_1 oder x_3 als Pivot hat, hat man zunächst 2 Vergleiche, und einer der rekursiven Aufrufe führt zu einem dritten.
Insgesamt: 3 Vergleiche. Wahrscheinlichkeit hierfür: $\frac{2}{3}$.

Mittlere Vergleichsanzahl: $C_3 = \frac{1}{3} \cdot 2 + \frac{2}{3} \cdot 3 = \frac{8}{3}$.

C_n: Wenn man beim ersten **r_qsort**(a, b)-Aufruf das i-t kleinste Element x_i als Pivotelement wählt, dann erzeugt dieser Aufruf selbst zunächst genau $n - 1$ Vergleiche,

es ergibt sich als partitionierende Position p (in p) genau $a + i - 1$,

und es entstehen die rekursiven Aufrufe

r_qsort$(a, a + i - 2)$ (Teilarray der Länge $i - 1$) und
r_qsort$(a + i, b)$ (Teilarray der Länge $n - i$).

Diese kosten im Durchschnitt C_{i-1} und C_{n-i} Vergleiche.

Beachte: Da anfangs die Schlüssel zufällig angeordnet waren, und bei **partition** nur die Beziehung zwischen Schlüsselwert und dem Pivotelement x relevant ist (Lemma 6.3.1(b)), ist nachher die Anordnung in den beiden Teilarrays ebenfalls zufällig.

Falls x_i Pivotelement ist: Durchschnittliche Kosten sind $\underbrace{n-1}_{\textbf{partition}} + \underbrace{C_{i-1}}_{\text{1. rek. Aufruf}} + \underbrace{C_{n-i}}_{\text{2. rek. Aufruf}}$.

Die Wahrscheinlichkeit, dass x_i Pivotelement ist, ist gleich $1/n$, weil nach der Wahrscheinlichkeitsannahme jede Anordnung der Einträge $x_1, \ldots, x_n$ in $\texttt{A}[a\,..\,b]$ gleichwahrscheinlich ist (Lemma 6.3.1(b)).

Also:

$$C_n = \frac{1}{n} \cdot \sum_{i=1}^{n} (n - 1 + C_{i-1} + C_{n-i})$$

$$= n - 1 + \frac{1}{n} \cdot \sum_{i=1}^{n} (C_{i-1} + C_{n-i}).$$

$C_0 = C_1 = 0$, $C_2 = 1$, $C_n = n - 1 + \frac{1}{n} \cdot \sum_{i=1}^{n}(C_{i-1} + C_{n-i})$ für $n \geq 3$.

Abschnitt 4.2 in Kap. 4, Folien 24–32 :

Rekursionsformel für die mittlere totale Weglänge $A(n)$ in zufällig erzeugten binären Suchbäumen.

$A(0) = A(1) = 0$, $A(2) = 1$,

$A(n) = n - 1 + \frac{1}{n} \cdot \sum_{i=1}^{n}(A(i-1) + A(n-i))$.

Das ist genau die gleiche Rekursion wie die für C_n!

Bemerkung: Dies ist kein Zufall, sondern es besteht eine recht enge Verbindung zwischen Quicksort und dem randomisierten Erzeugen von Suchbäumen.

Also sind auch die Lösungen identisch.

Satz 6.3.4

Die erwartete Vergleichsanzahl bei der Anwendung von Quicksort auf ein Array mit n verschiedenen Einträgen, die zufällig angeordnet sind, ist

$$2(n+1)H_n - 4n \approx \underline{(2\ln 2)n\log n} - 2{,}846\ldots n.$$

Dabei ist $2\ln 2 = \mathbf{1{,}386294\ldots}$ (die „Quicksort-Konstante“). □

Korollar 6.3.5

Für die randomisierte Version, bei der das Pivotelement zufällig gewählt wird, gilt dieselbe Formel für die erwartete Anzahl von Vergleichen, **und zwar für jedes beliebige feste Eingabearray** $\texttt{A}[1\,..\,n]$.

Randomisierung verhindert worst-case-Eingaben bei Quicksort!

Bemerkung:
Eine $O(n\log n)$-Laufzeit stellt sich bei zufälliger Eingabeanordnung bzw. bei randomisiertem Quicksort sogar mit hoher Wahrscheinlichkeit ein. („Zuverlässigkeitsaussage“.)

Bemerkungen:

- Quicksort ist in der Praxis sehr schnell. (Sequenzielle Zugriffe in **partition** sind Cache-freundlich.)
- Vermeide rekursive Aufrufe für allzu kleine Teilarrays: besser direkt mit **Insertionsort** sortieren.
- **Warnung 1:** Wenn `A[1 . . n]` teilweise sortiert ist und man das Pivotelement naiv wählt, erhält man schlechte Laufzeiten, bis zu $\Omega(n^2)$. Abhilfe: Clever Quicksort oder Randomisiertes Quicksort.
- **Warnung 2:** Wenn `A[1 . . n]` (viele) identische Schlüssel enthält, ist das beschriebene Verfahren nicht günstig.

  Einfache Abhilfe: Ersetze Schlüssel `A[`i`].key` durch das Paar (`A[`i`].key`,i), und sortiere lexikographisch mit randomisiertem Quicksort. Oder: **3-Wege-Partition**. (Für Pivot x bilde drei Blöcke mit Einträgen $< x$, $= x$ und $> x$ und wende Rekursion nur auf den ersten und den dritten Block an.)
- Quicksort ist **nicht stabil**.
- Platz (für den Rekursionsstack) kann im schlechtesten Fall bis zu $O(n)$ groß werden.

  Ausweg: Formuliere **r_qsort** um, so dass Rekursion nur für das kleinere der beiden Teilarray angewendet wird. Das größere wird in einer Iteration weiterbehandelt.

  Effekt: Deutlich weniger Prozeduraufrufe. Der Rekursionsstack hat höchstens Tiefe $\log n$ – fast „in situ“. (Siehe nächste Folie.)

Prozedur hr_qsort(a, b) // Halb-rekursive Variante

```
(1)  a ← a; b ← b;
(2)  while a < b do
(3)     s ← ein (z. B. zufälliges) Element von {a, ..., b};
(4)     vertausche A[a] und A[s];
(5)     partition(a, b, p);
(6)     if  p − a < b − p
(7)        then
(8)           hr_qsort(a, p − 1);
(9)           a ← p + 1;        // while-Schleife bearbeitet rechtes Teilarray weiter
(10)       else  // p − a ≥ b − p
(11)          hr_qsort(p + 1, b)
(12)          b ← p − 1;        // while-Schleife bearbeitet linkes Teilarray weiter
```

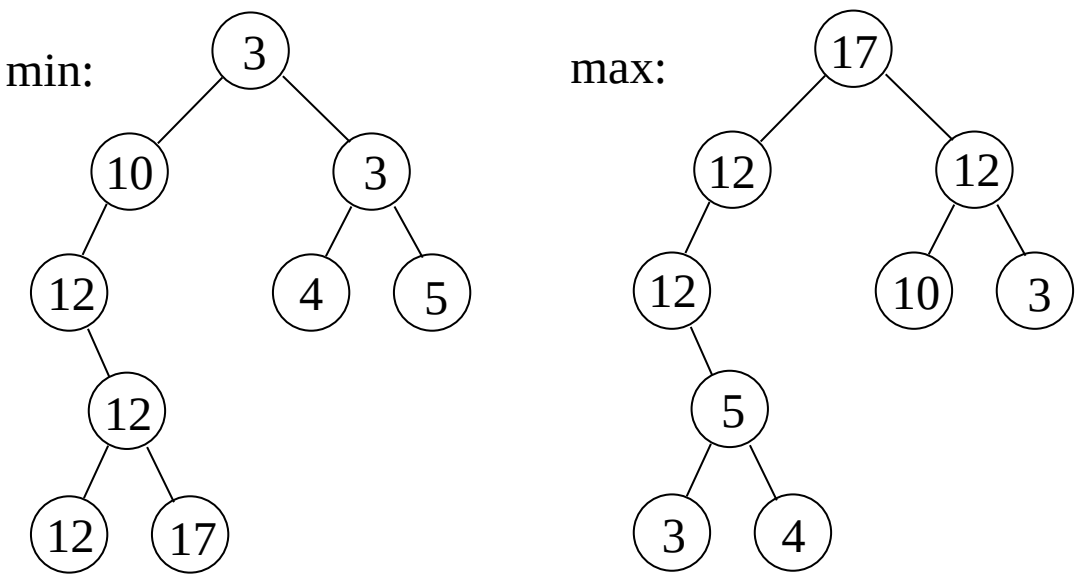
Vorlesungsvideo:

Heaps

6.4 Heapsort

Heap-Ordnung in Binärbäumen:

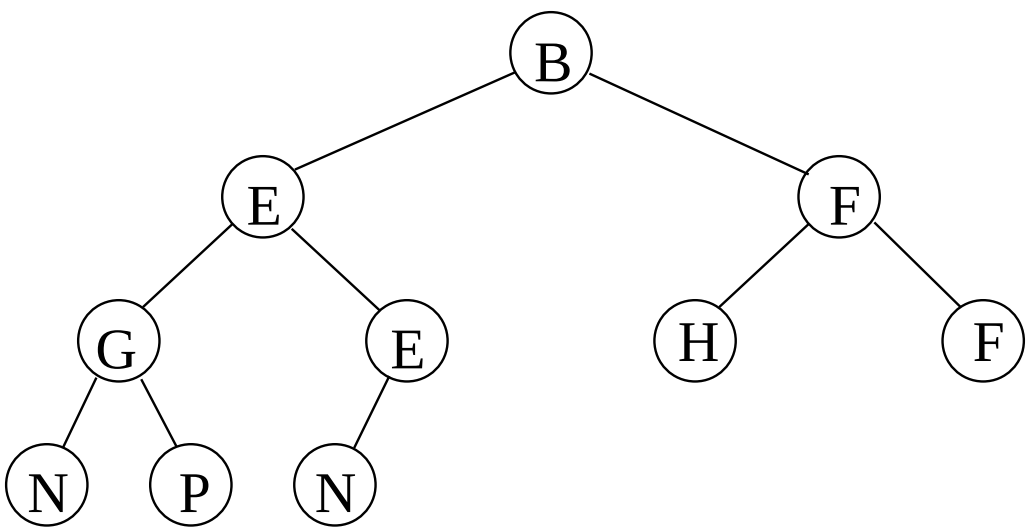
Beispiel: Hier nur interne Knoten relevant. Blätter: Knoten ohne (interne) Kinder.

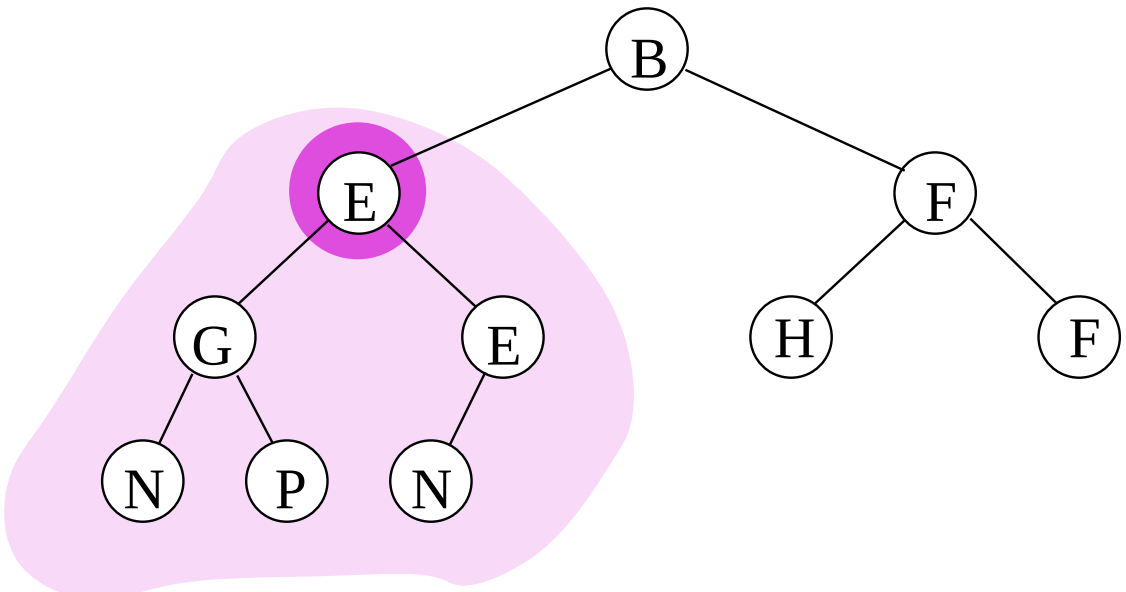


Links/Rechts (Min/Max-Heap): Entlang Blatt-Wurzel-Wegen fallen/steigen die Schlüssel.

Definition 6.4.1

Ein Binärbaum T mit Knotenbeschriftungen $m(v)$ aus $(U, <)$ heißt **min-heapgeordnet** (kurz oft: **heapgeordnet**), wenn für alle Knoten v und w gilt:
v Vorgänger von $w \Rightarrow m(v) \leq m(w)$.



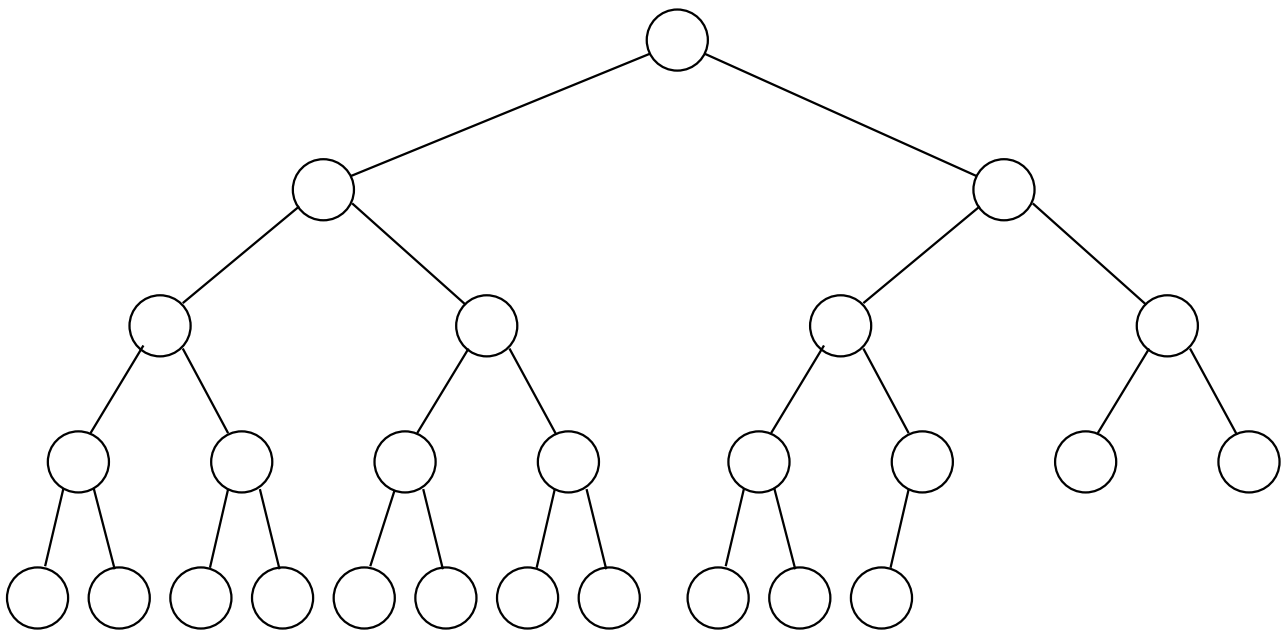


Beachte: T heapgeordnet $\Rightarrow$ in Knoten v steht der minimale Eintrag des Teilbaums T_v mit Wurzel v.

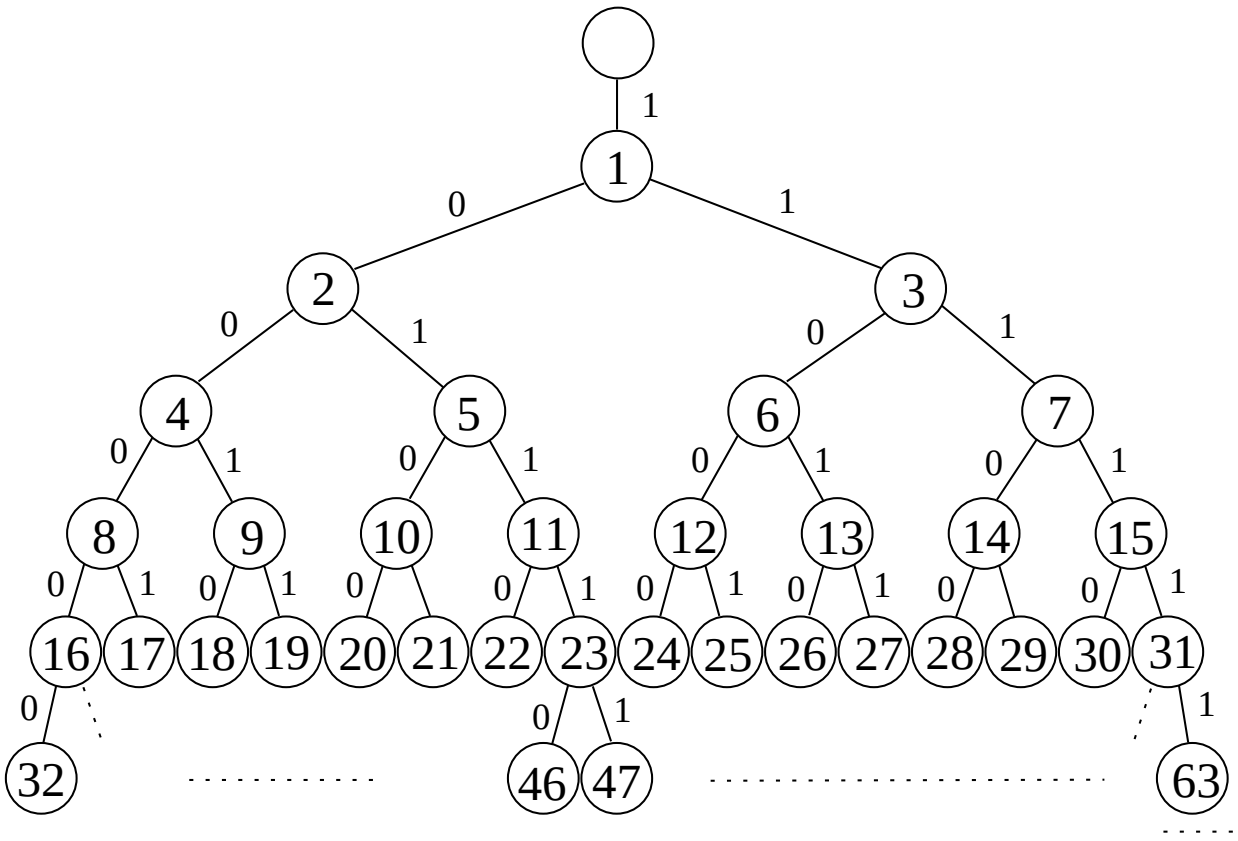
(Variante:

T heißt **max-heapgeordnet**, wenn für alle v, w gilt: v Vorgänger von w $\Rightarrow$ $m(v) \geq m(w)$.)

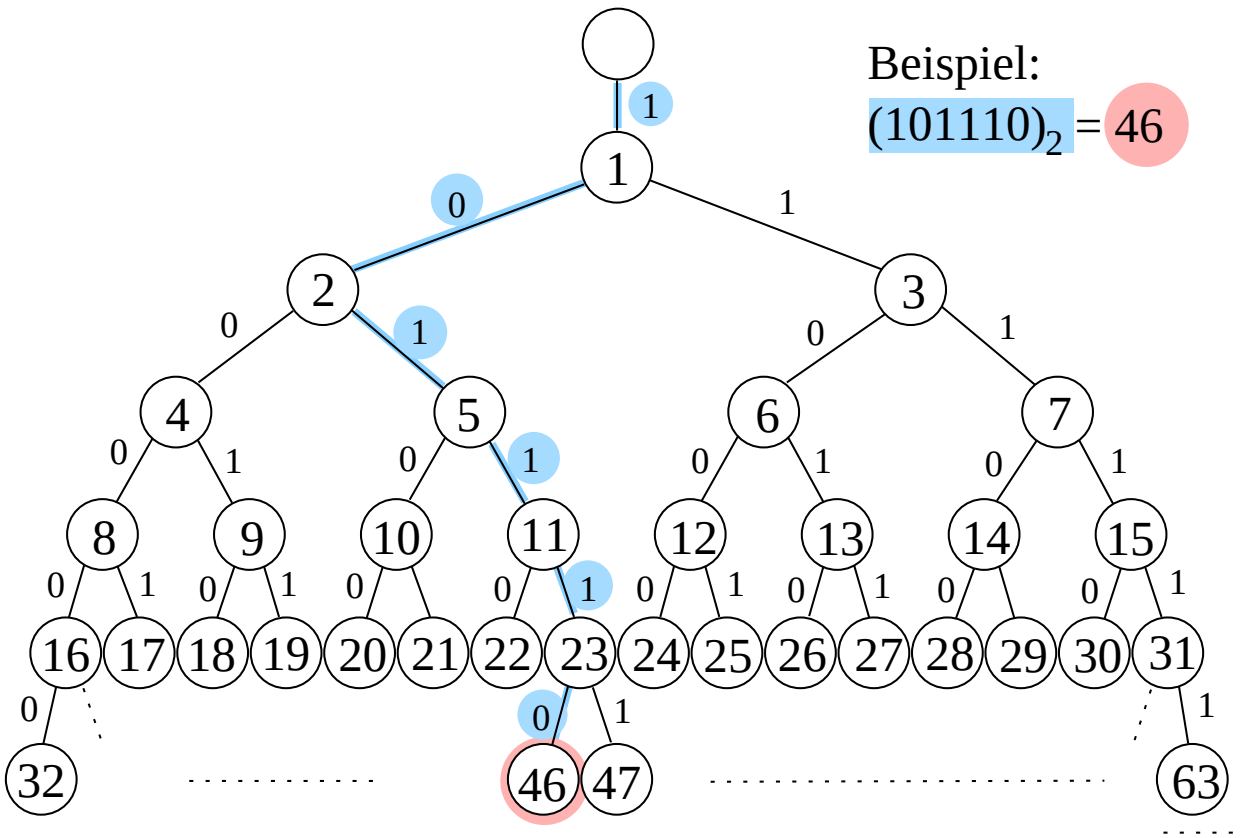
Ein **„linksvollständiger“** Binärbaum:
Alle Levels $j = 0, 1, \ldots$ voll (jeweils 2^j Knoten) bis auf das tiefste.
Das tiefste Level l hat von links her gesehen eine ununterbrochene Folge von Knoten.



(Unendlicher) vollständiger Binärbaum (mit Kunstwurzel), Kante zu linkem/rechtem Kind mit $0/1$ beschriftet, Knoten gemäß **Leveldurchlauf-Anordnung** nummeriert:



(Unendlicher) vollständiger Binärbaum (mit Kunstwurzel), Kante zu linkem/rechtem Kind mit $0/1$ beschriftet, Knoten gemäß **Leveldurchlauf-Anordnung** nummeriert:



Ist $s_1 \cdots s_l \in \{0,1\}^l$ die Markierung auf dem Weg von der Wurzel zu Knoten i, so ist $1s_1 \cdots s_l$ die **Binärdarstellung** von i.

Damit können linksvollständige Binärbaume **ohne Zeiger** als Arrays dargestellt werden: Speichere Eintrag zu Knoten Nummer i im Baum in `A[`i`]`!

Dies spart den Speicherplatz für die Zeiger ein.

Dennoch können wir leicht zu Kindern und Vorgängern navigieren:

- Knoten 1 ist die Wurzel;
- Knoten $2i$ ist linkes Kind von i;
- Knoten $2i + 1$ ist rechtes Kind von i;
- Knoten $i \geq 2$ hat Vorgänger $\lfloor i/2 \rfloor$.

Kombiniere Heapbedingung, Array-Darstellung, Daten:

Definition 6.4.2

Sei $(U, <)$ Totalordnung. Ein (Teil-)Array $\texttt{A}[1\,..\,k]$ mit Einträgen aus $U \times D$ heißt ein **Min-Heap** (oder **Heap**), wenn der entsprechende linksvollständige Binärbaum mit k Knoten (Knoten i mit $\texttt{A}[i]\,.\texttt{key} \in U$ und $\texttt{A}[i]\,.\texttt{data} \in D$ beschriftet) bezüglich der `key`-Komponenten (min-)heapgeordnet ist, d. h. wenn gilt:

$$\boxed{\texttt{A}[\lfloor i/2 \rfloor]\,.\texttt{key} \leq \texttt{A}[i]\,.\texttt{key}, \text{ für } 1 < i \leq k.}$$

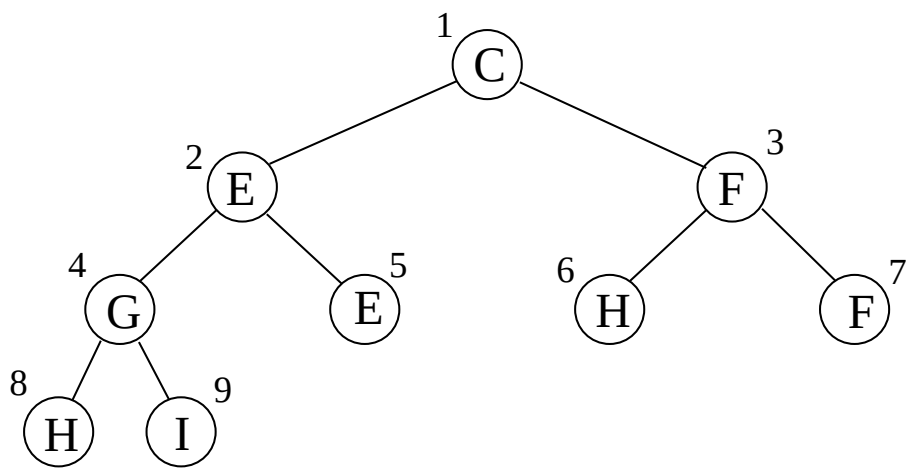
Die Charakterisierung im Kasten erwähnt den (gedachten) Binärbaum nicht mehr explizit.

Analog: **Max-Heap** (benutze Maxheap-Ordnung).

Beispiel: $U = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \ldots, \mathrm{Z}\}$ mit der Standardordnung. (Daten weggelassen.)

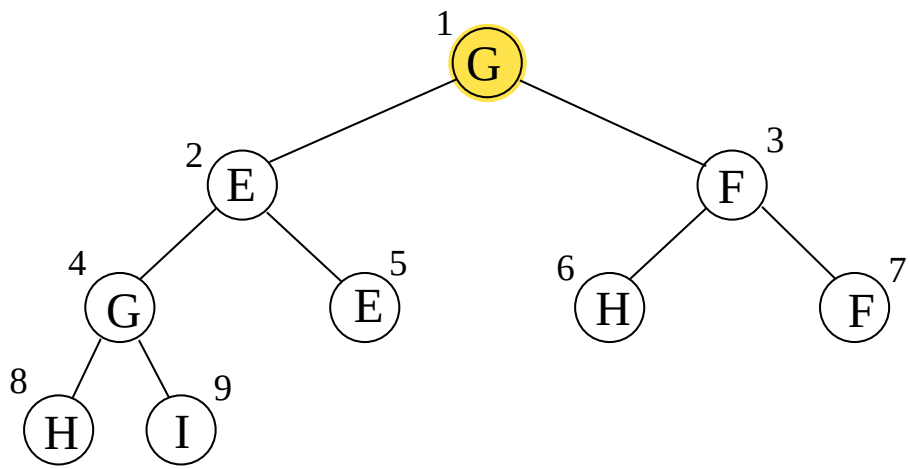
Ein Min-Heap und der zugehörige Baum ($k = 9$):

1	2	3	4	5	6	7	8	9	10	11	12
C	E	F	G	E	H	F	H	I	*	*	*



Aufgabe: Heaps reparieren

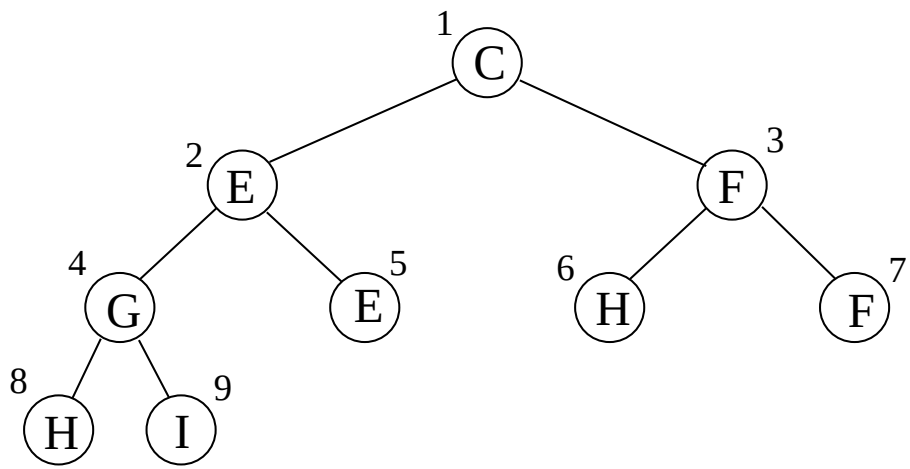
1	2	3	4	5	6	7	8	9	10	11	12
G	E	F	G	E	H	F	H	I	*	*	*



Beobachte: Wurzelschlüssel G z. B. durch C ersetzen liefert Heap.

Aufgabe: Heaps reparieren

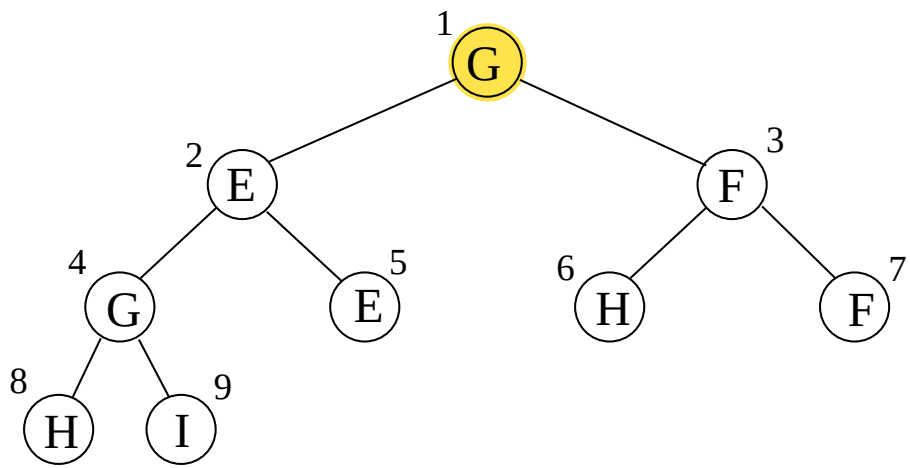
1	2	3	4	5	6	7	8	9	10	11	12
C	E	F	G	E	H	F	H	I	*	*	*



Beobachte: Wurzelschlüssel G z. B. durch C ersetzen liefert Heap.

Aufgabe: Heaps reparieren

1	2	3	4	5	6	7	8	9	10	11	12
G	E	F	G	E	H	F	H	I	*	*	*

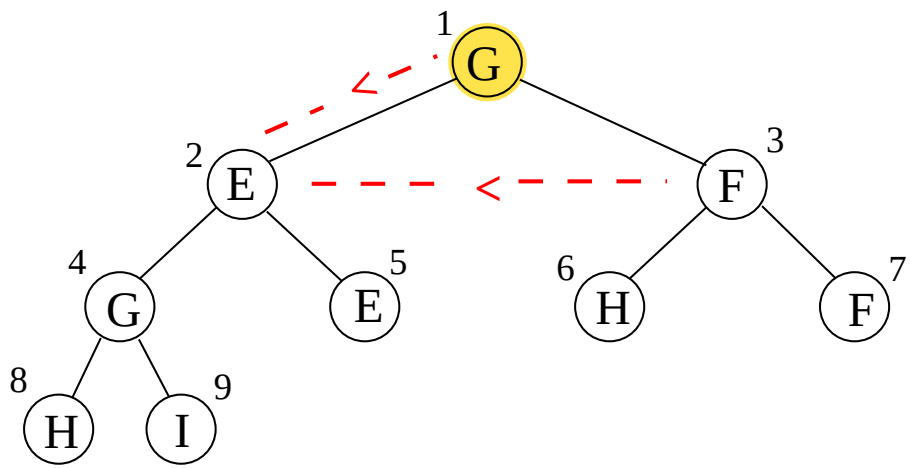


Beobachte: Wurzelschlüssel G z. B. durch C ersetzen liefert Heap.

Definition: Höchstens $\texttt{A}[j]$ ist zu groß :$\Leftrightarrow$ es gibt einen Schlüssel $x \leq \texttt{A}[j]\texttt{.key}$, so dass ein Heap entsteht, wenn man $\texttt{A}[j]\texttt{.key}$ durch x ersetzt.

Heaps reparieren

1	2	3	4	5	6	7	8	9	10	11	12
G	E	F	G	E	H	F	H	I	*	*	*



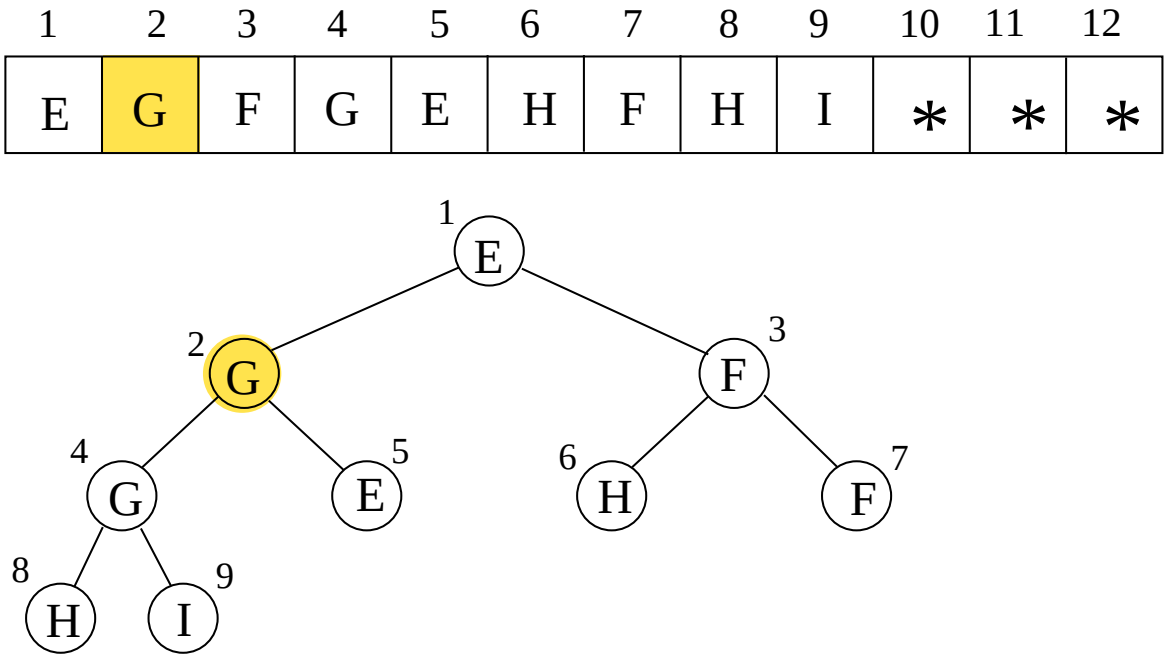
Höchstens `A[1]` ist zu groß: 1 ist „aktueller Knoten“.

Vergleiche die Kinder des aktuellen Knotens.

Vergleiche „kleineres“ Kind mit aktuellem Knoten.

Falls kleiner: **Vertausche** „kleineres“ Kind (E) mit dem aktuellen Knoten (G).

Heaps reparieren



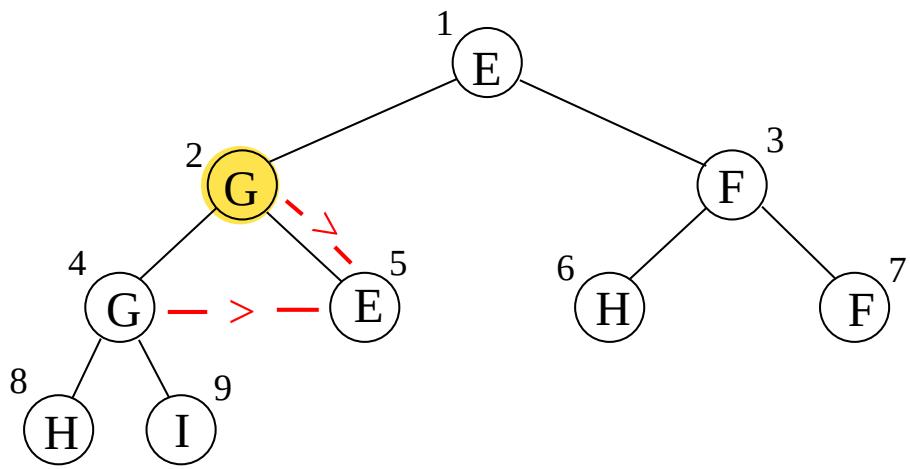
Vergleiche „kleineres“ Kind mit aktuellem Knoten.

Falls kleiner: **Vertausche** „kleineres“ Kind (E) mit dem aktuellen Knoten (G).

Nun: Höchstens `A[2]` ist zu groß. (Ersetze G durch E!)

Heaps reparieren

1	2	3	4	5	6	7	8	9	10	11	12
E	G	F	G	E	H	F	H	I	*	*	*



Höchstens $\texttt{A}[2]$ ist zu groß. (Ersetze G durch E!)

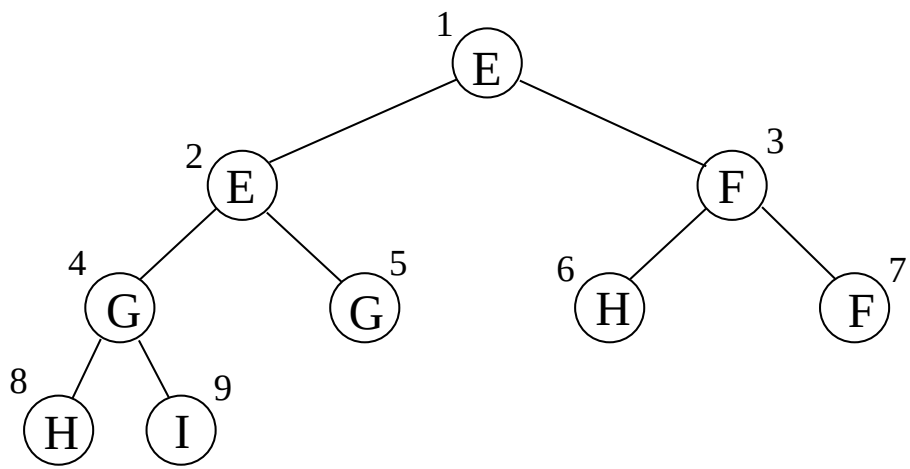
Vergleich der beiden Kinder von Knoten 2.

Vergleich des „kleineren“ Kinds (Knoten 5) mit dem aktuellen Knoten 2.

Vertauschen des „kleineren“ Kinds (Knoten 5) mit dem „aktuellen“ Knoten 2.

Heaps reparieren

1	2	3	4	5	6	7	8	9	10	11	12
E	E	F	G	G	H	F	H	I	*	*	*



Höchstens $\texttt{A}[5]$ ist zu groß. (Ersetze G durch E!)

Aktueller Knoten 5 hat keine Kinder $\Rightarrow$ Kein Fehler mehr: Reparatur beendet.

Andere Möglichkeit für Ende:

Eintrag im aktuellen Knoten ist nicht größer als die Einträge in seinen Kindern.

```
bubbleDown(1, k)  // Heap-Reparatur in A[1 .. k]
                  // Vorbedingung: Höchstens A[1] ist zu groß
(1)   j ← 1;      // „aktueller Knoten“
(2)   done ← (2 · j > k);
(3)   while not done do   // (∗) höchstens A[j] ist zu groß, A[j] hat Kind A[2 · j] im Baum
(4)       m ← 2 · j;
(5)       if m + 1 ≤ k and A[m+1].key < A[m].key
(6)           then m ← m + 1;   // nun gilt: A[m] ist einziges oder „kleineres“ Kind von A[j]
(7)       if A[m].key < A[j].key
(8)           then vertausche A[j] mit A[m];
(9)                j ← m;       // neuer „aktueller Knoten“
(10)               done ← (2 · j > k);   // wenn done = false, gilt wieder (∗)
(11)          else done ← true.  // fertig, kein Fehler mehr.
```

Anmerkung: Im Programmtext wird auf die Binärbaumstruktur nicht Bezug genommen (außer indirekt über die Indexberechnungen).

Korrektheit:

Wir wollen zeigen: Wenn höchstens `A[1]` zu groß ist und **bubbleDown**$(1, k)$ durchgeführt wird, dann entsteht ein Heap mit denselben Einträgen wie vorher.

Dass sich die Menge der Einträge nicht ändert, folgt daraus, dass Elemente nur vertauscht werden.

Für die Heapeigenschaft zeigen wir die folgende Induktionsbehauptung **durch Induktion über die Schleifendurchläufe**:

(IB$_j$) Zu Beginn des Durchlaufs, in dem `j` die Zahl j enthält, ist **höchstens** `A[`j`]` **zu groß**.

Wir formulieren den Beweis in der Baum-Sprechweise.

Der Ind.-Anfang ($j = 1$) gilt nach Voraussetzung.

Betrachte Schleifendurchlauf für Knoten j.

1. Fall: In Zeile (8) wird $\texttt{A}[j]$ mit $\texttt{A}[2j]$ vertauscht.

Es seien u, v, w die Schlüssel in $\texttt{A}[j]$, $\texttt{A}[2j]$ bzw. $\texttt{A}[2j+1]$ **vor** der Vertauschung.
Nach (IB_j) können wir ein $x \leq u$ wählen, so dass

$(*)$ in $\texttt{A}[1..k]$ ein Heap entsteht, wenn man das u in $\texttt{A}[j]$ durch x ersetzt.

Wir wissen nach dem Algorithmus: $v \leq w$ und $v < u$.

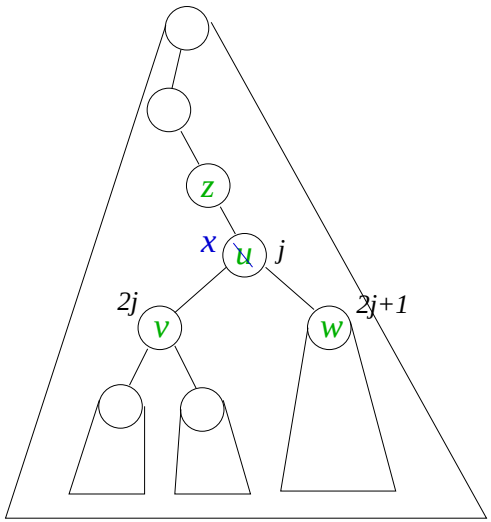
Nach der Vertauschung steht u in $\texttt{A}[2j]$ und v in $\texttt{A}[j]$.

Für **I.-Schritt** zu zeigen: Es entsteht ein Heap, wenn wir u in $\texttt{A}[2j]$ durch v ersetzen.

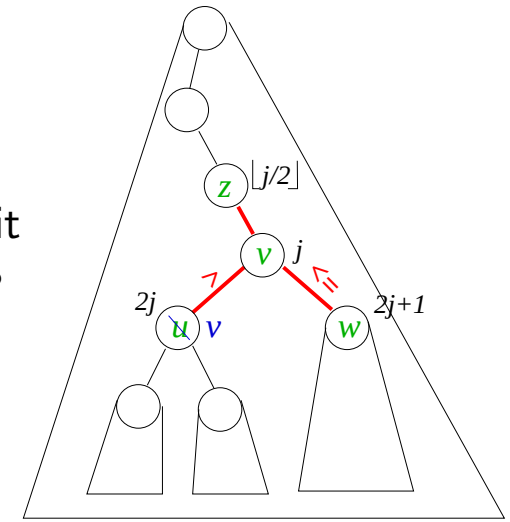
(Weil $v < u$ gilt, liefert das (IB_{2j}).)

Behauptung: Aus $\texttt{A}[1..k]$ (nach der Vertauschung) entsteht ein Heap, wenn wir $\texttt{A}[2j]$ Schlüssel u durch v ersetzen.

(A)
Vorher, nach I.V.:
Ein Heap
(mit x in $\texttt{A}[j]$).



(B)
Nachher:
Ein Heap (mit
v in $\texttt{A}[2j]$)?



Situationen (A) und (B) (mit Ersetzungen $u \to x$ bzw. $u \to v$) unterscheiden sich nur in Knoten j.
$\to$ Müssen in (B) nur Relationen zwischen j, seinem Vorgänger und seinen Kindern überprüfen.

(1) „j vs. $\lfloor j/2 \rfloor$“: Nach I.V. in (A) gilt $z \leq x \leq v$.

(2) „j vs. $2j$“: Die Schlüssel sind beide gleich v.

(3) „j vs. $2j+1$“: Im Algorithmus, Zeile (8), wurden v und w verglichen und es wurde $2j$ als „kleineres Kind“ gewählt. Also gilt $v \leq w$.

2. Fall: In Zeile (7) wird `A[`j`]` mit `A[`$2j+1$`]` vertauscht: analog zum 1. Fall.

In jedem Schleifendurchlauf wird der Inhalt von `j` (mindestens) verdoppelt. Daher gilt:
Nach einer Reihe von Schleifendurchläufen endet die Schleife. Dann steht ein j in `j`, für das gilt:

(IB_j) und Knoten j hat keine Kinder oder die Schlüssel in den Kindern sind $\geq u = \texttt{A[}j\texttt{].key}$.

Sei $x \leq u$ so, dass ein Heap entsteht, wenn u durch x ersetzt wird.
Dann ist auch `A[`$1 \,..\, k$`]` ohne diese Ersetzung ein Heap, weil die Schlüssel in den Kindern von Knoten j nicht kleiner als u sind und im Fall $j > 1$ der Schlüssel in Knoten $\lfloor j/2 \rfloor$ höchstens $x \leq u$ ist.

Kosten: Der Inhalt j der Variablen `j` ist anfangs 1 und verdoppelt sich in jeder Runde (mindestens); ein Schleifendurchlauf findet nur statt, wenn $2j \leq k$ ist.
Daher: Wenn es s Durchläufe durch die **while**-Schleife gibt, muss $2^s \leq k$ sein, d. h. $s \leq \log k$.
In jedem Durchlauf gibt es maximal zwei Schlüsselvergleiche.
Also ist der Zeitaufwand $O(\log k)$, bei nicht mehr als $2 \log k$ Vergleichen. □

Wir haben damit das folgende Ergebnis gezeigt.

Lemma 6.4.3

Wenn anfangs höchstens `A[1]` zu groß ist, stellt **bubbleDown**$(1, k)$ die Heapeigenschaft her. Die Prozedur führt maximal $2 \log k$ Vergleiche durch und hat Laufzeit $O(\log k)$.

Vorlesungsvideo:

Heapsort

Die Idee von Heapsort für ein Array $\texttt{A}[1..n]$ ist nun folgende:

1. (Heapaufbau) Arrangiere $\texttt{A}[1..n]$ so um, dass ein Heap entsteht.

2. (Auswahlphase) Für k von n abwärts bis 2:

Entnimm kleinsten Eintrag $\texttt{A}[1]$ aus dem Heap $\texttt{A}[1..k]$. Stelle mit den in $\texttt{A}[2..k]$ verbliebenen Einträgen in $\texttt{A}[1..k-1]$ wieder einen Heap her.
(In $\texttt{A}[k]$ ist dann Platz für den entnommenen Eintrag.)

Dadurch werden die Arrayeinträge in steigender Reihenfolge entnommen.

Wenn wir den in Runde k entnommenen Eintrag in $\texttt{A}[k]$ speichern, ist am Ende das Array $\texttt{A}[1..n]$ fallend sortiert.

Auswahlphase als Programm:

Prozedur HeapSelect$(1, n)$

(1) **for** $\mathtt{k}$ **from** n **downto** 2 **do**

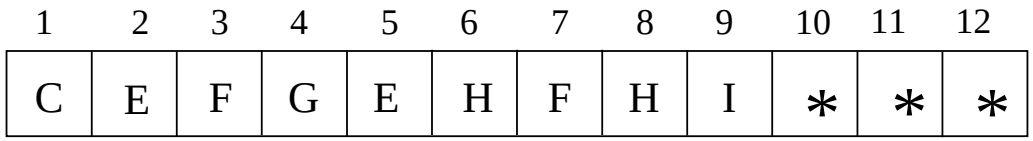
// in $\mathtt{A}[1]$ steht das Minimum von $\mathtt{A}[1..k]$

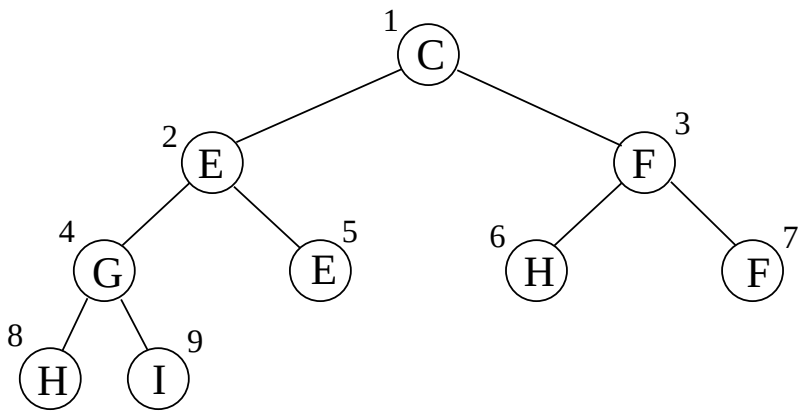
(2) vertausche $\mathtt{A}[1]$ mit $\mathtt{A}[\mathtt{k}]$;

// in $\mathtt{A}[1..k-1]$ ist höchstens $\mathtt{A}[1]$ zu groß

(3) **bubbleDown**$(1, \mathtt{k}-1)$; // Heapreparatur

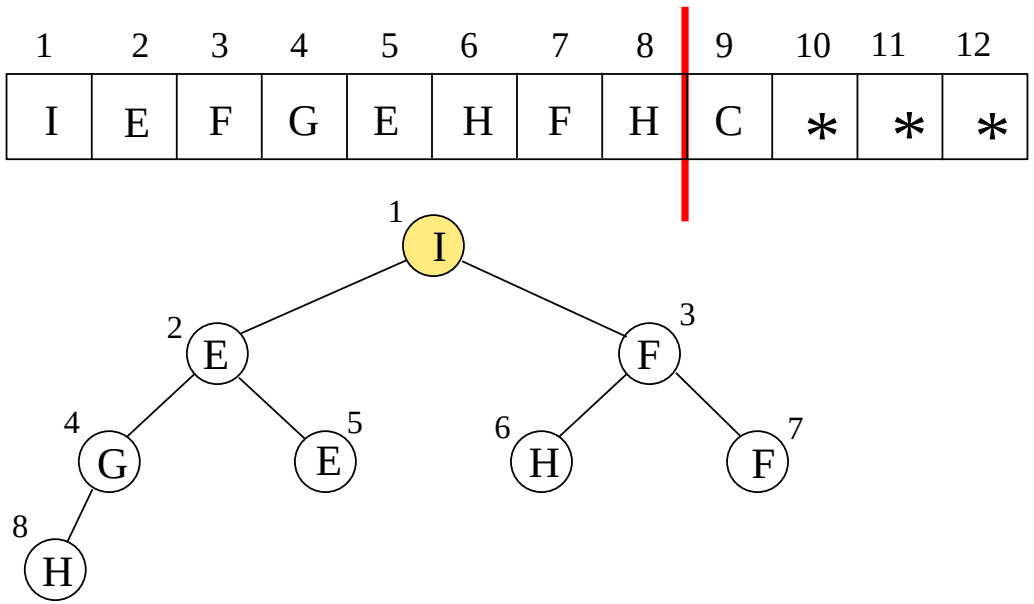
Auswahlphase





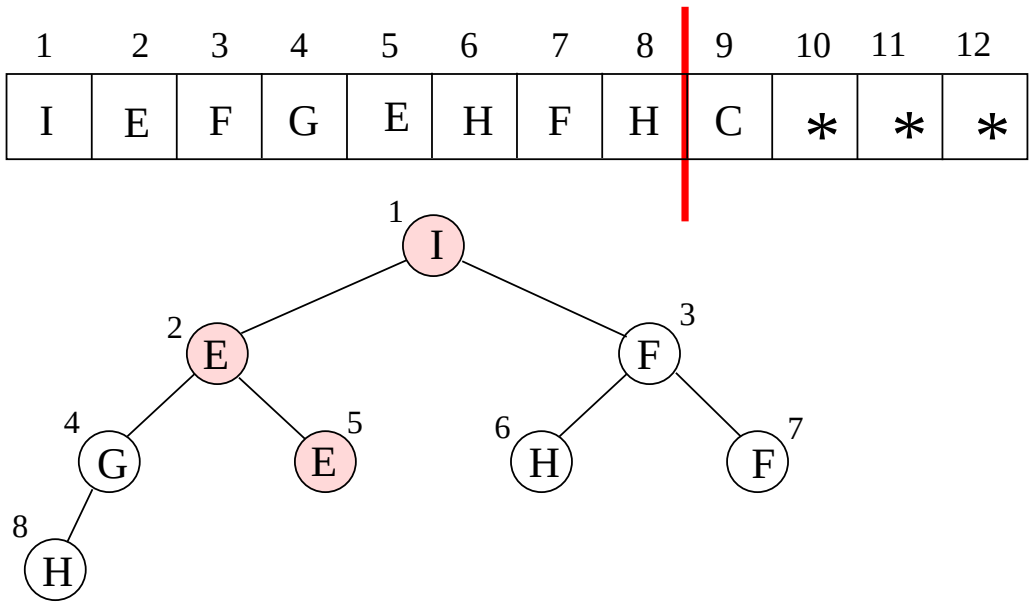
Ein Heap: $\mathtt{A}[1..9]$.

Auswahlphase



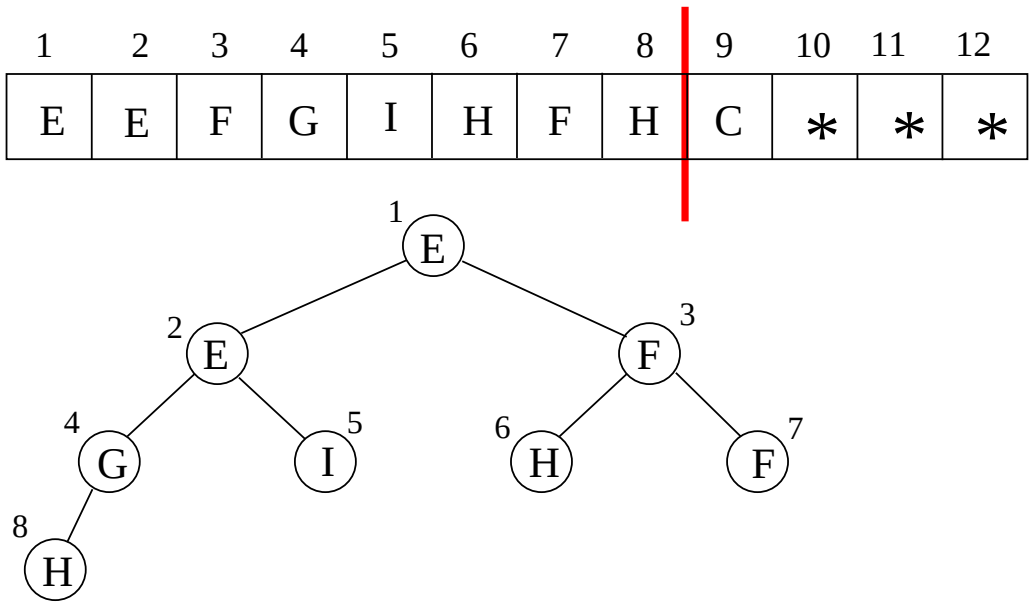
Minimum aus `A[1]` nach `A[9]`, altes `A[9]` in die Wurzel.
In `A[1..8]` ist höchstens `A[1]` zu groß.

Auswahlphase



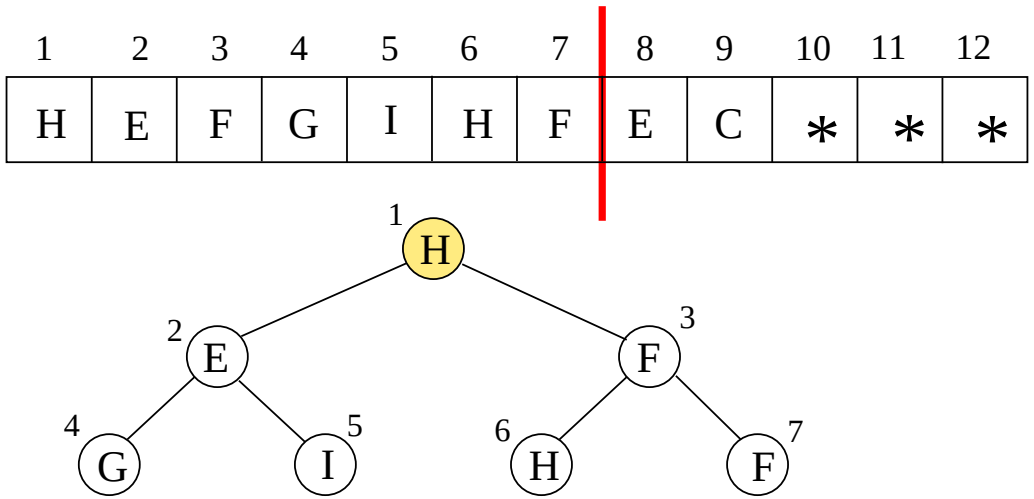
Von **bubbleDown** verfolgter Weg.

Auswahlphase



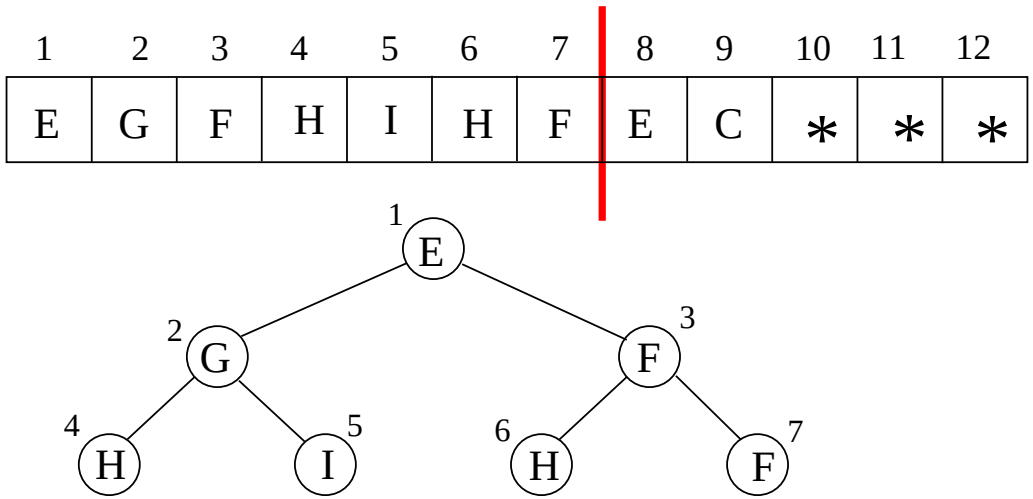
A[1..8] ist Heap.

Auswahlphase



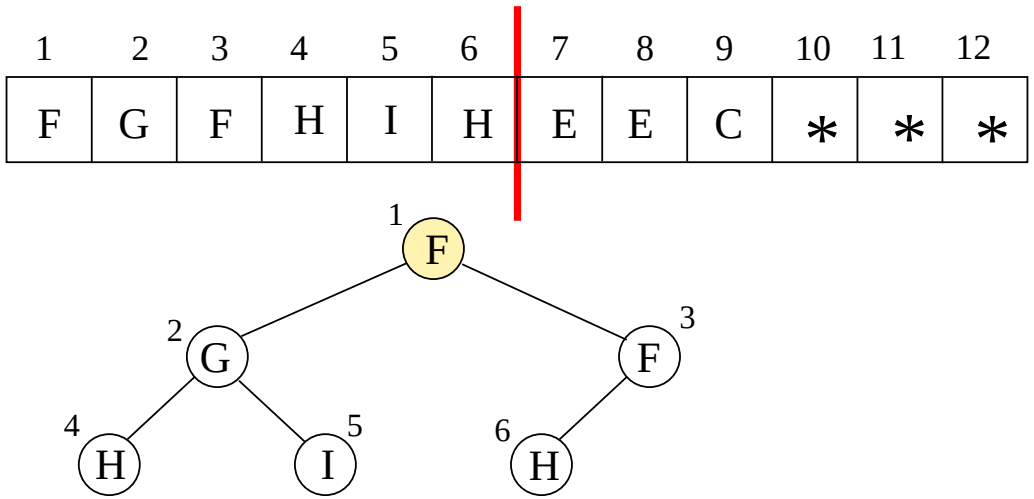
Minimum aus `A[1]` nach `A[8]`, altes `A[8]` in die Wurzel.
In `A[1..7]` ist höchstens `A[1]` zu groß.

Auswahlphase



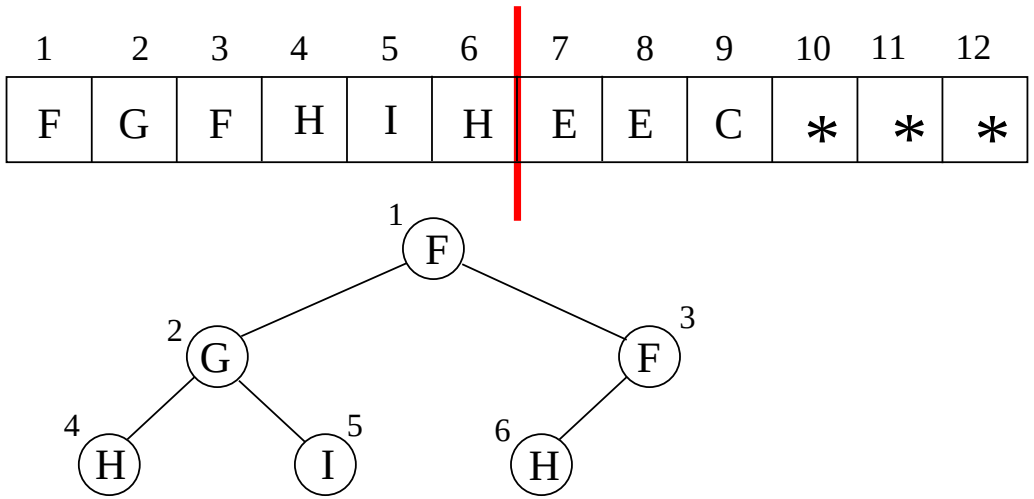
`A[1..7]` ist Heap.

Auswahlphase



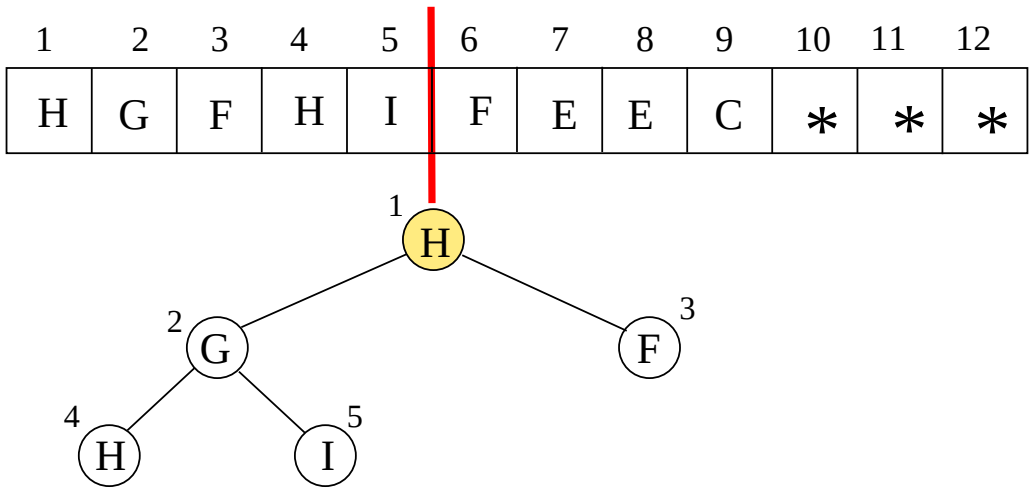
Minimum aus `A[1]` nach `A[7]`, altes `A[7]` in die Wurzel.
In `A[1..6]` ist höchstens `A[1]` zu groß.

Auswahlphase



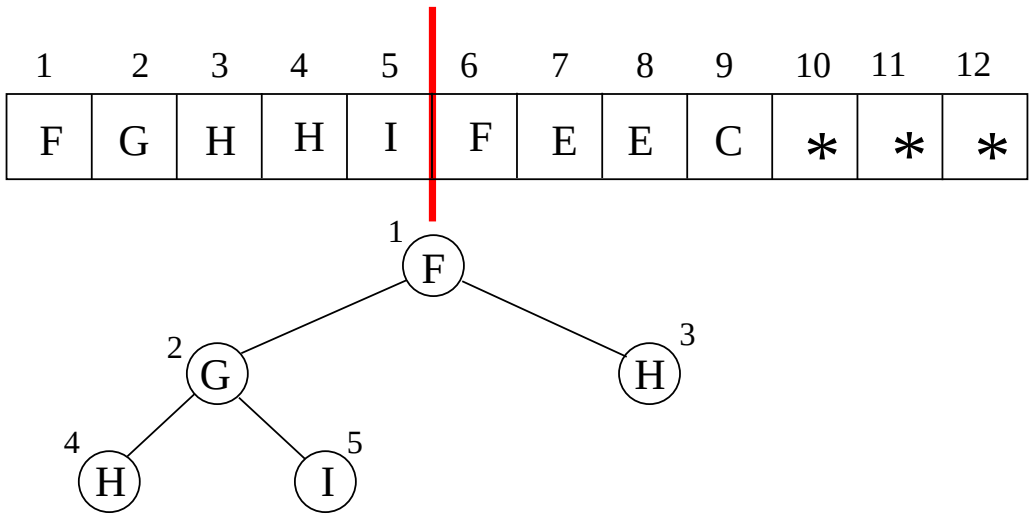
$\texttt{A}[1..6]$ ist Heap.

Auswahlphase



Minimum aus $\texttt{A}[1]$ nach $\texttt{A}[6]$, altes $\texttt{A}[6]$ in die Wurzel.
In $\texttt{A}[1..5]$ ist höchstens $\texttt{A}[1]$ zu groß.

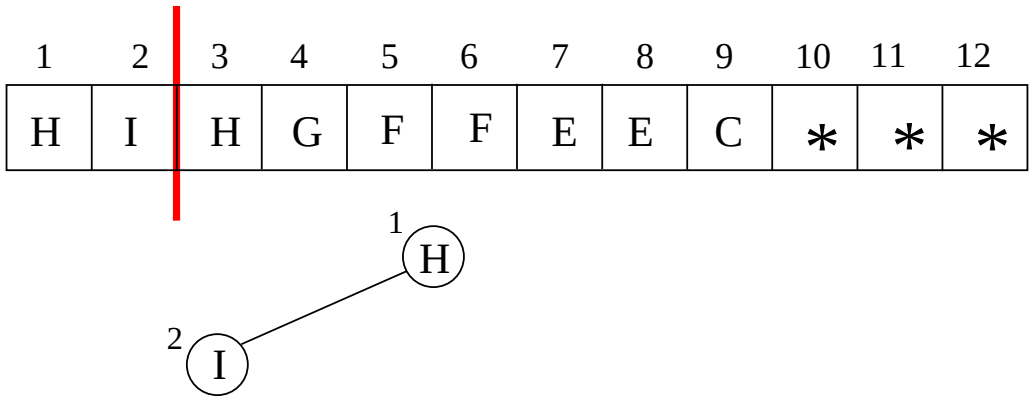
Auswahlphase



$\texttt{A}[1..5]$ ist Heap.

usw.

Auswahlphase



A[1..2] ist Heap.

Auswahlphase

1	2	3	4	5	6	7	8	9	10	11	12
I	H	H	G	F	F	E	E	C	*	*	*

Vertauschen von `A[1]` und `A[2]` vollendet fallende Sortierung.

bubbleDown$(1, 1)$ hat keinen Effekt.

Satz 6.4.4

Die Prozedur **HeapSelect**$(1, n)$ führt höchstens $2n \log n$ Vergleiche durch und hat Rechenzeit $O(n \log n)$.

Beweis:

Anzahl Vergleiche: höchstens $\sum_{2 \leq k \leq n} 2 \log k < 2n \log n$.

Kosten: $\sum_{2 \leq k \leq n} (O(1) + O(\log k)) = O(n) + O(n \log n) = O(n \log n)$. □

Es fehlt noch: **„1. Heapaufbau“.**

Wie verwandelt man ein beliebiges Array `A[1..n]` in einen Heap?

Idee: Betrachte im linksvollständigen Baum mit n Knoten die Teilbäume T_ℓ mit Wurzel ℓ, und stelle in diesen „von unten nach oben“, also in der Reihenfolge $\ell = n, n-1, \ldots, 3, 2, 1$, die Heapbedingung her.

Beobachtung: Wenn $n \geq \ell > n/2$, dann ist $2\ell > n$, also besteht der Teilbaum T_ℓ mit Wurzel ℓ nur aus dem Knoten ℓ und ist daher heapgeordnet.

Nun arbeitet man iterativ die Werte $\ell = \lfloor n/2 \rfloor, \lfloor n/2 \rfloor - 1, \ldots, 2, 1$ ab.

Sei ein solches ℓ gegeben, und seien $T_{2\ell}$ und $T_{2\ell+1}$, die Unterbäume an den Kindern von ℓ, schon heapgeordnet. Das bedeutet: Teilbaum T_ℓ ist heapgeordnet, außer dass eventuell im Wurzelknoten ℓ ein zu großer Schlüssel steht.

Wir wissen schon, wie man das korrigiert: Man wendet **bubbleDown** an, startend bei der Wurzel, die hier ℓ ist.

Also löst die folgende Variante von **bubbleDown** (s. Folie 52) die Aufgabe.

Prozedur bubbleDown(ℓ, n) // Heap-Reparatur im Teilbaum T_ℓ

// Vorbedingung: T_ℓ ist heapgeordnet, außer dass höchstens $\mathtt{A}[\ell]$ zu groß ist

(1) $\mathtt{j} \leftarrow \ell$;

(2) $\mathtt{done} \leftarrow (2 \cdot \mathtt{j} > n)$;

(3) **while not** done **do**

(4) ...

Lemma 6.4.6

Wenn in $\mathtt{A}[\ell + 1\,..\,n]$ (d. h. in allen Teilbäumen mit Wurzel $> \ell$) die Heapbedingung gilt, dann gilt sie nach Ausführung von **bubbleDown**(ℓ, n) in $\mathtt{A}[\ell\,..\,n]$. Die Anzahl der Vergleiche ist nicht größer als $2\log(n/\ell)$, die Laufzeit $O(\log(n/\ell))$.

Beweis: Korrektheit wie bei Lemma 6.4.3. Anzahl Schlüsselvergleiche ähnlich wie bei Lemma 6.4.3: Wenn es in **bubbleDown**(ℓ, n) s Schleifendurchläufe gibt, muss $2^s \cdot \ell \leq n$ sein, d. h. $s \leq \log(n/\ell)$. Also gibt es maximal $2\log(n/\ell)$ Vergleiche bei einem Zeitaufwand von $O(\log(n/\ell))$. □

Prozedur makeHeap$(1, n)$ // Transformiert Array `A[1..n]` in einen Heap

(1) **for** `ell` **from** $\lfloor n/2 \rfloor$ **downto** 1 **do**

(2) **bubbleDown**(`ell`, n);

Lemma 6.4.7

Ein Aufruf **makeHeap**$(1, n)$ wandelt `A[1..n]` in einen Heap um. Die Anzahl der Vergleiche ist nicht größer als $2n$; die Rechenzeit ist $O(n)$.

Beweis der Korrektheit: Mit Induktion über $\ell = \lfloor n/2 \rfloor, \lfloor n/2 \rfloor - 1, \ldots, 2, 1$:
Nach Durchlauf mit ℓ in `ell` gilt Heapbedingung in `A[`ℓ`..n]`. □

Kosten: Die Anzahl der Vergleiche ist höchstens (mit Lemma 6.4.6):

$$2 \cdot \sum_{1 \le \ell \le n/2} \lfloor \log(n/\ell) \rfloor = 2 \cdot \sum_{1 \le \ell \le n/2} \sum_{j\,:\, j \ge 1 \wedge \ell \cdot 2^j \le n} 1$$

(weil $\lfloor \log(n/\ell) \rfloor = |\{j \mid j \le \log(n/\ell)\}| = |\{j \mid 2^j \le n/\ell\}|$)

$$= 2 \cdot \sum_{1 \le j \le \log n} \sum_{1 \le \ell \le n/2^j} 1$$

(Vertauschung der Summationsreihenfolge)

$$\le 2 \cdot \sum_{1 \le j \le \log n} \frac{n}{2^j} \quad < \quad 2n \cdot \sum_{1 \le j \le \log n} \frac{1}{2^j}$$

(Geometrische Reihe: $\sum_{j \ge 1} 2^{-j} = 1$.)

$$< 2n.$$

Algorithmus HeapSort(A[1 . . n])
(1) **makeHeap**$(1, n)$;
(2) **HeapSelect**$(1, n)$;

Satz 6.4.8

Der Aufruf **HeapSort**(A[1 . . n]) sortiert A[1 . . n] in fallende Reihenfolge. Die gesamte Rechenzeit ist $O(n \log n)$, die Gesamtzahl der Vergleiche ist kleiner als $2n(\log n + 1)$. **HeapSort** arbeitet **in situ/in-place** (ist aber **nicht stabil**).

Beweis: Korrektheit und Zeitbedarf folgt aus der Analyse der Teilprozeduren. Die Gesamtzahl der Vergleiche ist nach Satz 6.4.4 und Lemma 6.4.7 höchstens:

$2n + \sum_{2 \le k \le n} 2 \log k \le 2n + 2 \cdot \sum_{2 \le k \le n} \log k \le 2n + 2n \log n.$ □

Anmerkung: Man kann sogar zeigen: $< 2n \log n$ Vergleiche genügen, für nicht ganz kleine n.

Bemerkung 1: Wenn Sortierung **in aufsteigende Reihenfolge** gewünscht wird, ersetze in allen Prozeduren und Programmen Schlüsselvergleiche „$\texttt{A}[i] \leq \texttt{A}[j]$“ durch „$\texttt{A}[i] \geq \texttt{A}[j]$“ und umgekehrt. Die in $\texttt{A}[1\,..\,n]$ entstehende Struktur ist dann ein **Max-Heap**: in der Wurzel des Binärbaumes steht immer das Maximum.

Bemerkung 2: Die **mittlere** Anzahl von Vergleichen, unter der Annahme, dass jede Anordnung der Eingabeobjekte in $\texttt{A}[1\,..\,n]$ dieselbe Wahrscheinlichkeit $1/n!$ hat, ist $2n \log n - O(n)$.

Bemerkung 3: Es gibt eine Variante namens „Bottom-Up-Heapsort“, die $n \log n + O(n)$ Vergleiche im **mittleren** Fall hat. – **Idee:** Bei **bubbleDown**$(1, k)$ laufe gesamten „Weg der kleineren Kinder“ bis zum Blatt, ziehe dabei die „kleineren Kinder“ um eine Ebene nach oben. (Nur ein Vergleich pro Ebene.) Füge am Ende das kritische Element aus der Wurzel *von unten nach oben* an die richtige Stelle auf diesem Weg ein. Im durchschnittlichen Fall sind hier nur konstant viele Schritte nach oben nötig.

Bemerkung 4: Bei Variante **„d-Way-Heapsort“**, für $d \geq 3$, sind die Arraypositionen mit $0, 1, \ldots, k-1$ nummeriert. Jeder Knoten $i \in \{0, 1 \ldots, k-1\}$ hat bis zu d viele Kinder, nämlich $di+1, di+2, \ldots, di+d$ (soweit diese Zahlen $< k$ sind).
Der Vorgänger des Knotens $i > 0$ ist $\lfloor (i-1)/d \rfloor$.

Die Heapbedingung in $\texttt{A}[0..k-1]$ wird modifiziert:

$$\boxed{\texttt{A}[\lfloor (i-1)/d \rfloor] \leq \texttt{A}[i], \text{ für } 1 \leq i < k.}$$

Bei **bubbleDown**$(0, k-1)$ wird $\texttt{A}[j]$ mit dem *kleinsten Eintrag in einem Kind* von Knoten j vertauscht, wenn der Schlüssel in diesem Kind kleiner ist als $\texttt{A}[j]\texttt{.key}$.
(Die Entscheidung erfordert bis zu d viele Vergleiche pro Ebene.)

Vorteil: Der Baum hat nur Tiefe $\log_d n = (\log n)/(\log d)$. Daher gibt es bei **bubbleDown** Zugriff nur auf $(\log n)/(\log d)$ zusammenhängende Segmente im Array. Das führt zu viel weniger Cache-Fehlern als bei gewöhnlichem Heapsort und ist daher schneller. (Experimentell stellt man fest, dass besonders $d = 4$ zu attraktiven Verbesserungen führt.)

Übung: Man formuliere die Details und beweise die Korrektheit von **bubbleDown**$(0, k-1)$.

Vorlesungsvideo:

Prioritätswarteschlangen

6.5 Datentyp: Prioritätswarteschlange

oder Vorrangwarteschlangen

In einer Prioritätswarteschlange werden Objekte aufbewahrt, die mit einem Schlüssel aus einem totalgeordneten Universum $(U, <)$ versehen sind. (Oft sind die Schlüssel Zahlen.) Objekte werden **eingefügt** und **entnommen**.

Einfügungen sind beliebig möglich.

Beim Entnehmen wird immer ein Eintrag mit **minimalem Schlüssel** gewählt.

Idee: Schlüssel entsprechen hier **„Prioritäten“** – je *kleiner* der Schlüssel, desto *höher* die Priorität. Wähle stets einen Eintrag mit höchster Priorität!

Der Datentyp Prioritätswarteschlage ist mit **Heaps** effizient zu realisieren.

Wir diskutieren hier nur **Binärheaps**, aber man könnte genauso gut auch Bäume mit größerem Ausgangsgrad benutzen, etwas 4-Wege-Heaps oder allgemein k-Wege-Heaps.

Beispiel: Datensatz: (L, k), $L \in \{\mathtt{A}, \ldots, \mathtt{Z}\}$, $k \in \mathbb{N}$, Schlüssel ist der Buchstabe L.

Operation	innerer Zustand	Ausgabe
empty	$\emptyset$	–
isempty	$\emptyset$	$true$
insert$(\mathtt{M}, 1)$	$\{(\mathtt{M}, 1)\}$	–
insert$(\mathtt{K}, 2)$	$\{(\mathtt{M}, 1), (\mathtt{K}, 2)\}$	–
insert$(\mathtt{M}, 3)$	$\{(\mathtt{M}, 1), (\mathtt{K}, 2), (\mathtt{M}, 3)\}$	–
insert$(\mathtt{R}, 4)$	$\{(\mathtt{M}, 1), (\mathtt{K}, 2), (\mathtt{M}, 3), (\mathtt{R}, 4)\}$	–
insert$(\mathtt{T}, 5)$	$\{(\mathtt{M}, 1), (\mathtt{K}, 2), (\mathtt{M}, 3), (\mathtt{R}, 4), (\mathtt{T}, 5)\}$	–
extractMin	$\{(\mathtt{M}, 1), (\mathtt{M}, 3), (\mathtt{R}, 4), (\mathtt{T}, 5)\}$	$(\mathtt{K}, 2)$
extractMin	$\{(\mathtt{M}, 1), (\mathtt{R}, 4), (\mathtt{T}, 5)\}$	$(\mathtt{M}, 3)$
insert$(\mathtt{R}, 6)$	$\{(\mathtt{M}, 1), (\mathtt{R}, 4), (\mathtt{T}, 5), (\mathtt{R}, 6)\}$	–
insert$(\mathtt{M}, 2)$	$\{(\mathtt{M}, 1), (\mathtt{R}, 4), (\mathtt{T}, 5), (\mathtt{R}, 6), (\mathtt{M}, 2)\}$	–
extractMin	$\{(\mathtt{R}, 4), (\mathtt{T}, 5), (\mathtt{R}, 6), (\mathtt{M}, 2)\}$	$(\mathtt{M}, 1)$
extractMin	$\{(\mathtt{R}, 4), (\mathtt{T}, 5), (\mathtt{R}, 6)\}$	$(\mathtt{M}, 2)$
isempty	$\{(\mathtt{R}, 4), (\mathtt{T}, 5), (\mathtt{R}, 6)\}$	$false$

Beispiel: Datensatz: (L, k), $k \in \mathbb{N}$, $L \in \{\mathtt{A}, \ldots, \mathtt{Z}\}$, Schlüssel ist der Buchstabe L.

Operation	innerer Zustand	Ausgabe
empty	$\emptyset$	–
isempty	$\emptyset$	$true$
insert$(\mathtt{M}, 1)$	$\{(\mathtt{M}, 1)\}$	–
insert$(\mathtt{K}, 2)$	$\{(\mathtt{M}, 1), (\mathtt{K}, 2)\}$	–
insert$(\mathtt{M}, 3)$	$\{(\mathtt{M}, 1), (\mathtt{K}, 2), (\mathtt{M}, 3)\}$	–
insert$(\mathtt{R}, 4)$	$\{(\mathtt{M}, 1), (\mathtt{K}, 2), (\mathtt{M}, 3), (\mathtt{R}, 4)\}$	–
insert$(\mathtt{T}, 5)$	$\{(\mathtt{M}, 1), (\mathtt{K}, 2), (\mathtt{M}, 3), (\mathtt{R}, 4), (\mathtt{T}, 5)\}$	–
extractMin	$\{(\mathtt{M}, 1), (\mathtt{M}, 3), (\mathtt{R}, 4), (\mathtt{T}, 5)\}$	$(\mathtt{K}, 2)$
extractMin	$\{(\mathtt{M}, 1), (\mathtt{R}, 4), (\mathtt{T}, 5)\}$	$(\mathtt{M}, 3)$
insert$(\mathtt{R}, 6)$	$\{(\mathtt{M}, 1), (\mathtt{R}, 4), (\mathtt{T}, 5), (\mathtt{R}, 6)\}$	–
insert$(\mathtt{M}, 2)$	$\{(\mathtt{M}, 1), (\mathtt{R}, 4), (\mathtt{T}, 5), (\mathtt{R}, 6), (\mathtt{M}, 2)\}$	–
extractMin	$\{(\mathtt{R}, 4), (\mathtt{T}, 5), (\mathtt{R}, 6), (\mathtt{M}, 2)\}$	$(\mathtt{M}, 1)$
extractMin	$\{(\mathtt{R}, 4), (\mathtt{T}, 5), (\mathtt{R}, 6)\}$	$(\mathtt{M}, 2)$
isempty	$\{(\mathtt{R}, 4), (\mathtt{T}, 5), (\mathtt{R}, 6)\}$	$false$

Einfügereihenfolge hat keine festgelegte Beziehung zur Ausgabereihenfolge.

Anwendungen:

1) **Prozessverwaltung in Multitask-System**
Jeder Prozess hat einen Namen (eine „ID“) und eine Priorität (Zahl in $\mathbb{N}$).
(**Kleinere Zahlen** bedeuten **höhere Prioritäten**.)
Aktuell rechenbereite Prozesse befinden sich in der **Prozesswarteschlange**. Bei Freiwerden eines Prozessors (z. B. durch Unterbrechen eines Prozesses) soll einer der Prozesse **mit höchster Priorität** aus der Warteschlange entnommen und weiter ausgeführt werden.

2) **„Discrete Event Simulation“**
System von „Aktionen“ oder „Ereignissen“ soll auf dem Rechner simuliert werden.
(Anwendungsbeispiel: Technisches System mit mehreren kommunizierenden Komponenten.)
Jeder **ausführbaren** Aktion A ist ein Zeitpunkt $t_A \in [0, \infty)$ zugeordnet.
Die Ausführung einer Aktion A kann neue Aktionen A' erzeugen oder auch schon vorhandene Aktionen A' früher ausführbar machen, mit neuen Zeitpunkten $t_{A'} > t_A$.
Ein Schritt: Wähle diejenige noch nicht ausgeführte Aktion *mit dem frühesten Ausführungszeitpunkt* und führe sie aus.

3) Innerhalb von **Algorithmen** (Dijkstra, Jarník/Prim (später), geometrische Algorithmen („scan-line“)).

Datentyp: SimplePriorityQueue (Einfache Prioritätswarteschlange)

1. Signatur:

Sorten:	*Keys*
	Data
	PrioQ // „Priority Queues“
	Boolean
Operationen:	*empty*: $\rightarrow$ *PrioQ*
	isempty: *PrioQ* $\rightarrow$ *Boolean*
	insert: *PrioQ* $\times$ *Keys* $\times$ *Data* $\rightarrow$ *PrioQ*
	extractMin: *PrioQ* $\rightarrow$ *PrioQ* $\times$ *Keys* $\times$ *Data*

Mathematisches Modell:

Sorten:

Keys	:	$(U, <)$ // totalgeordnetes „Universum“
Data	:	D // Menge von „Datensätzen“
Boolean	:	$\{false, true\}$
PrioQ	:	die Menge aller endlichen **Multimengen*** $P \subseteq U \times D$

Operationen:

empty$( ) = \emptyset$ // die *leere Multimenge*

$$\textit{isempty}(P) = \begin{cases} true, & \text{für } P = \emptyset \\ false, & \text{für } P \neq \emptyset \end{cases}$$

* **Multimenge**: Elemente dürfen mehrfach vorkommen, wie z. B. in $\{1, 1, 4, 5, 5, 5, 9\}$.

Mathematisches Modell (Forts.):

insert$(P, x, d) = P \cup \{(x, d)\}$ (als Multimenge),

$$\textit{extractMin}(P) = \begin{cases} \textit{undefiniert}, & \text{wenn } P = \emptyset \\ (P', x_0, d_0), & \text{wenn } P \neq \emptyset, \end{cases}$$

für ein Element (x_0, d_0) in P mit minimalem Schlüssel x_0, und $P' = P - \{(x_0, d_0)\}$ (als Multimenge).

Standardimplementierung: (Binäre) Heaps

Andere Implementierungen sind möglich!

Implementierung: Objekt mit Array `A[`$1\,..\,m$`]` (für die Einträge) und Pegel `n` für die aktuelle Zahl n von Einträgen.

Überlaufprobleme werden ausgeklammert. (Verdoppelungsstrategie, falls nötig.)

empty(m):
Lege Array `A[`$1\,..\,m$`]` an;
Jeder Eintrag kann ein Paar $(\texttt{key}, \texttt{data})$ aufnehmen.
$\texttt{n} \leftarrow 0$.

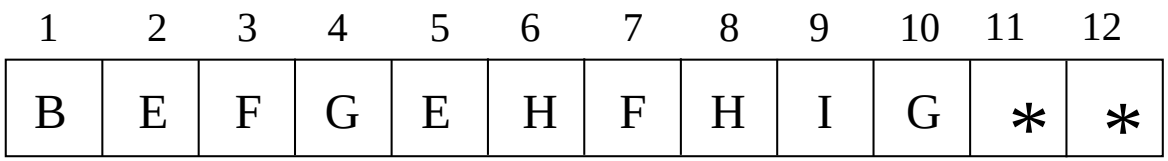
// Zeitaufwand: $O(1)$ oder $O(m)$.

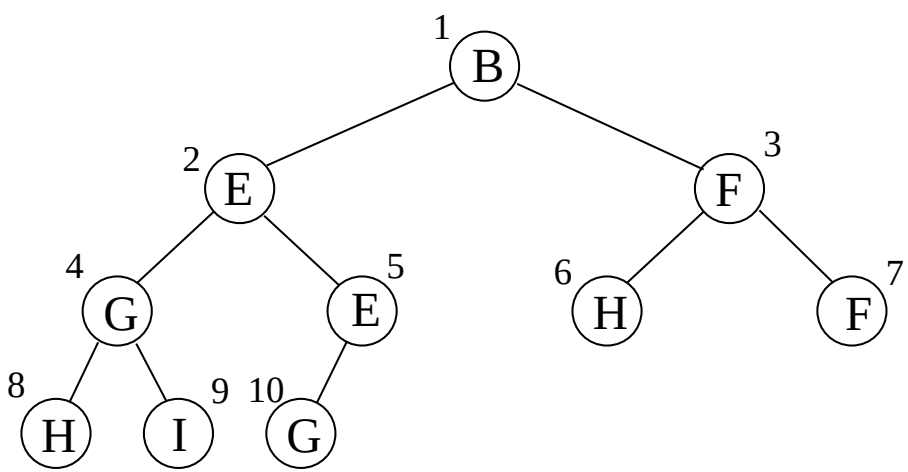
isempty$()$: **return** $(\texttt{n} = 0)$; // Zeitaufwand: $O(1)$

extractMin: Implementierung von *extractMin*(P):

Ein Eintrag mit minimalem Schlüssel steht in der Wurzel, d. h. in Arrayposition 1.

Entnehme `A[1]` (hier „B“) und gib es aus.

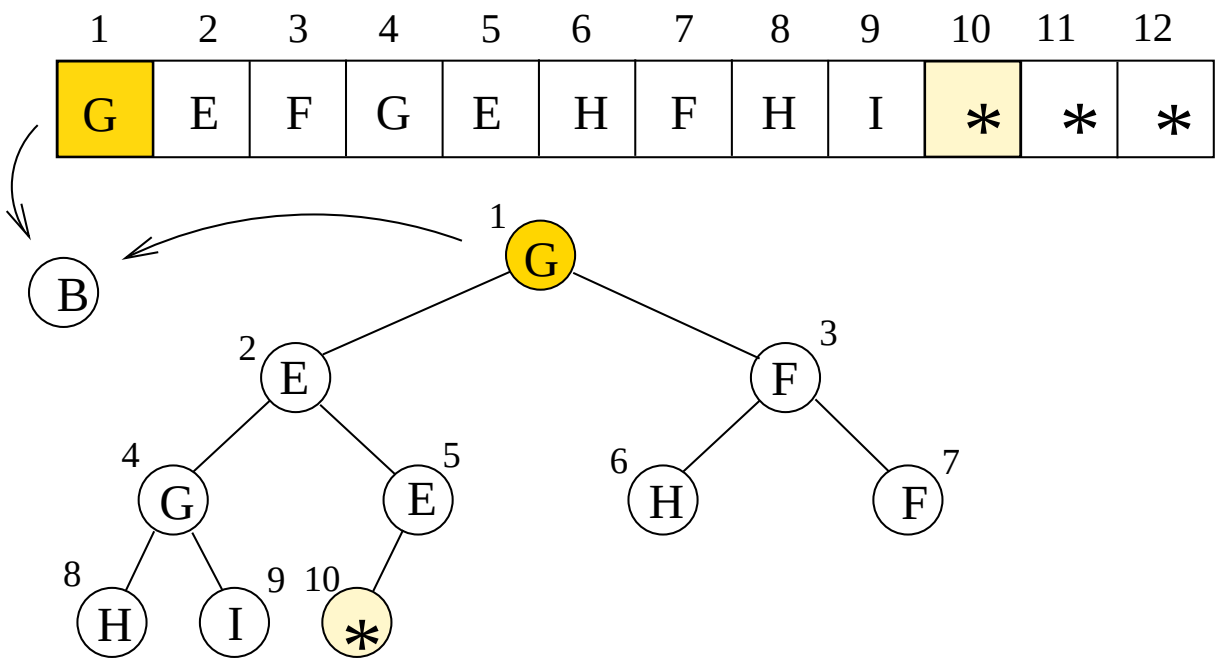




extractMin: Implementierung von *extractMin*(P):

Ein Eintrag mit minimalem Schlüssel steht in der Wurzel, d. h. in Arrayposition 1.

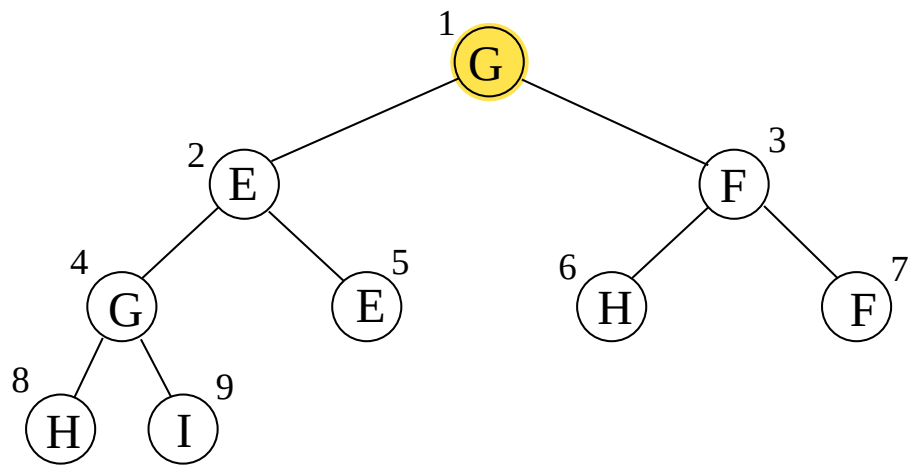
Entnehme `A[1]` (hier „B“) und gib es aus.



„Loch“ im Binärbaum – „Stopfen“ mit dem letzten Eintrag.

Bleibt Aufgabe: Heap reparieren

1	2	3	4	5	6	7	8	9	10	11	12
G	E	F	G	E	H	F	H	I	*	*	*



Höchstens `A[1]` ist zu groß, also: **bubbleDown** hilft hier!

Prozedur extractMin()

// Entnehmen eines minimalen Eintrags aus Priority-Queue
// Ausgangspunkt: Pegelstand n (in `n`), `A[`$1\,..\,n$`]` ist Heap, $n \geq 1$

(0) **if** `n` $= 0$ **then** „Fehlerbehandlung“;
(1) `x` $\leftarrow$ `A[1].key`; `d` $\leftarrow$ `A[1].data`;
(2) `A[1]` $\leftarrow$ `A[n]`;
(3) `n--`;
(4) **if** `n` > 1 `then` **bubbleDown**$(1,$`n`$)$;
(5) **return** $($`x`$,$ `d`$)$;

Korrektheit: klar wegen Korrektheit von **bubbleDown**$(1,$`n`$)$.

Zeitaufwand: $O(\log(n))$, wobei n die aktuelle Größe des Heaps ist.

Beachte: Benutzer der Datenstruktur hat keinen Einfluss auf die Reihenfolge der Ausgabe von Objekten mit identischem Schlüssel.

Implementierung von **insert**(x, d):

Voraussetzung: `A[1..n]` ist Heap; $1 \leq n < m$; `n` enthält n.

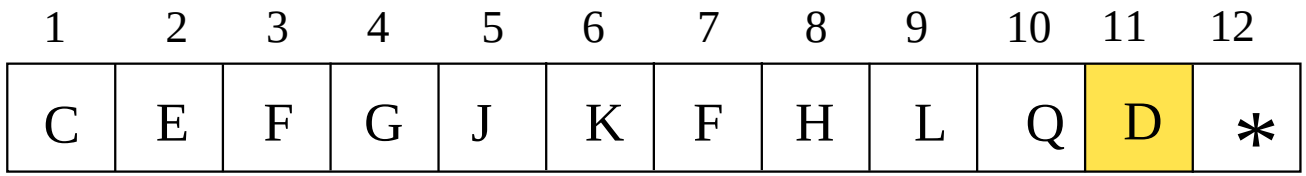
`n++`;
`A[n]` $\leftarrow (x, d)$.

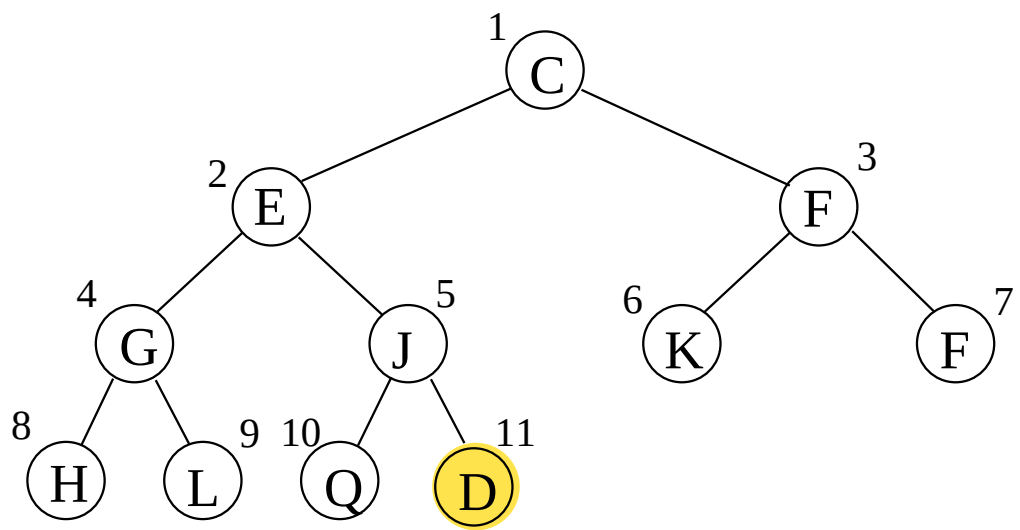
An der Stelle $n' = n + 1$ (neuer Pegelstand) ist nun eventuell die Heapeigenschaft gestört, **weil x zu klein ist**.

Definition

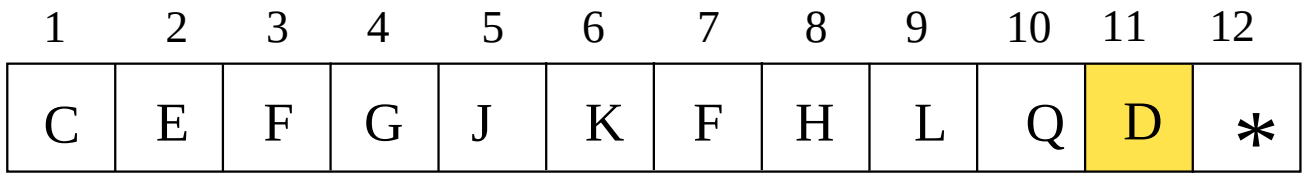
Höchstens `A[j]` ist zu klein (in `A[1..n]`) :$\Leftrightarrow$
in `A[1..n]` entsteht ein Heap, wenn man $u =$ `A[j].key` durch einen geeigneten Schlüssel $x \geq u$ ersetzt.

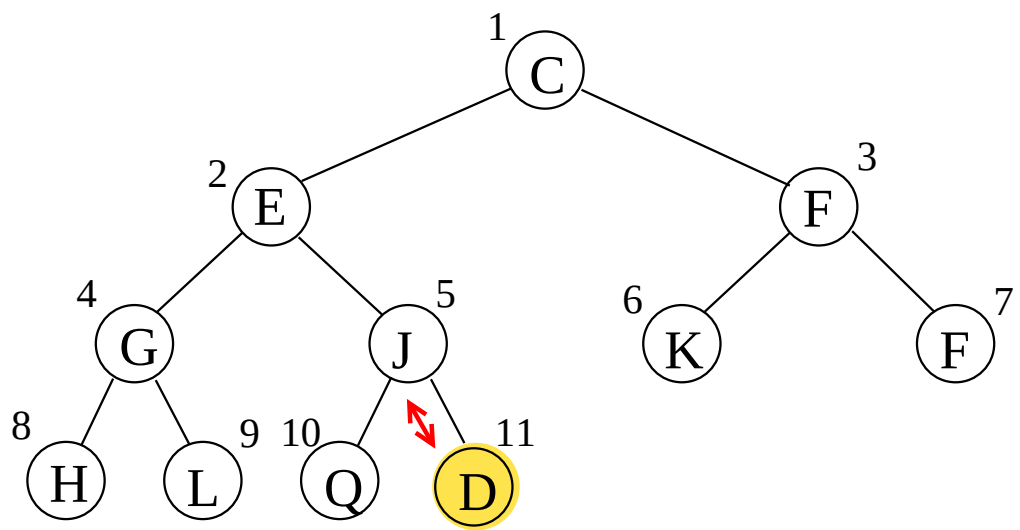
Einfügen von „D“ an Stelle $n = 11$.



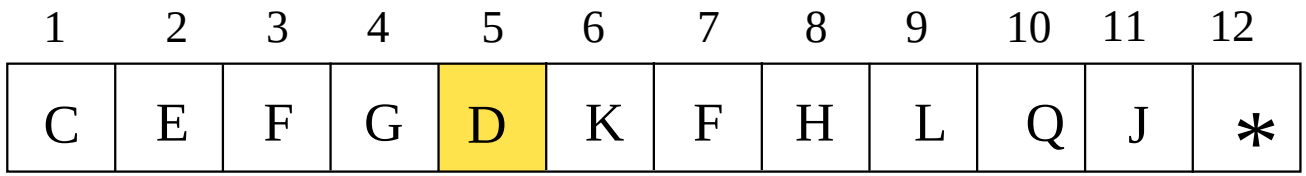


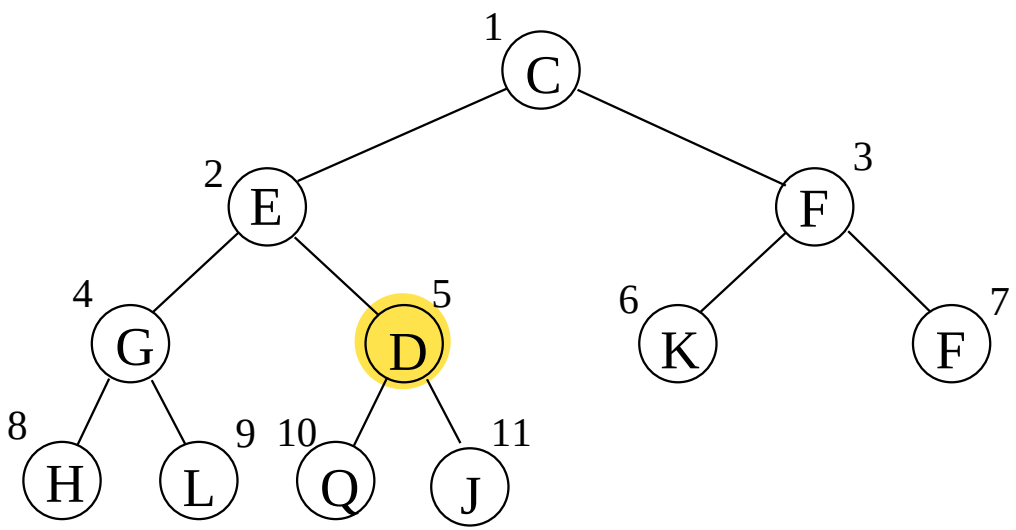
Reparatur, wenn höchstens ein Schlüssel zu klein: **bubbleUp**.





Reparatur, wenn höchstens ein Schlüssel zu klein: **bubbleUp**.

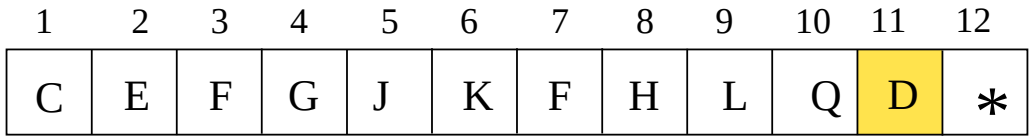


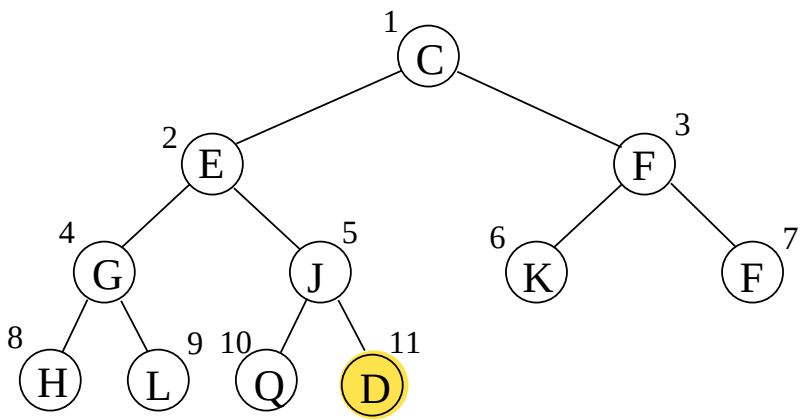


und so weiter . . .

Prozedur bubbleUp(i) // Heapreparatur ab `A[`i`]` nach oben

(1) `j` $\leftarrow i$;
(2) `h` $\leftarrow \lfloor$`j`$/2 \rfloor$;
(3) **while** `h` ≥ 1 **and** `A[h].key` $>$ `A[j].key` **do**
(4) vertausche `A[j]` mit `A[h]`;
(5) `j` $\leftarrow$ `h`;
(6) `h` $\leftarrow \lfloor$`j`$/2 \rfloor$.

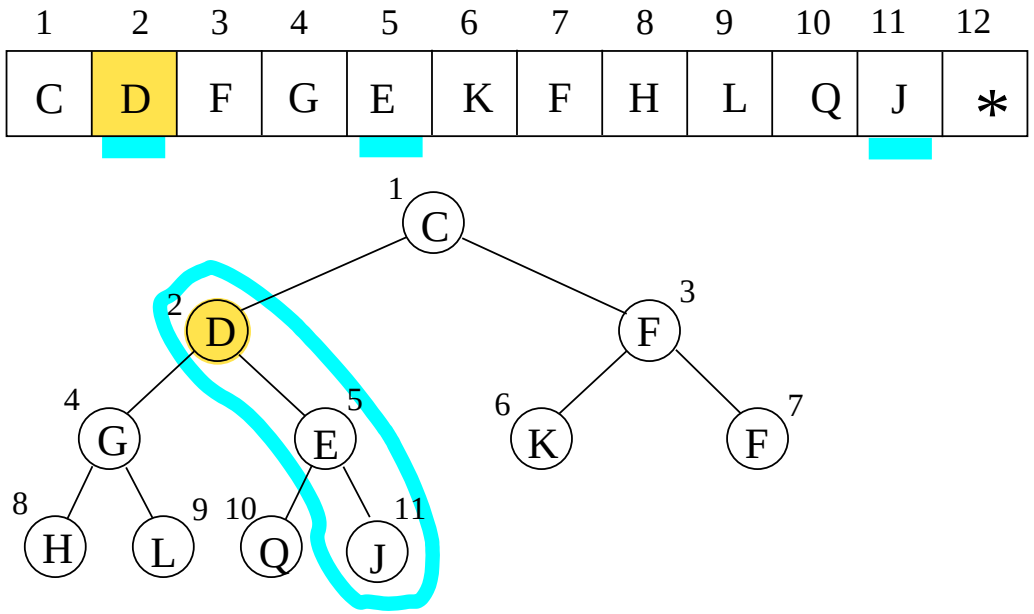




Anschaulich: Auf dem Weg von $\texttt{A}[i]$ zur Wurzel werden alle Elemente, deren Schlüssel größer als der (alte) Eintrag in $\texttt{A}[i]$) ist, um eine Position (auf dem Weg) nach unten gezogen.

Eintrag $\texttt{A}[i]$ landet in der freigewordenen Position.

Man kann dies auch effizienter programmieren (wie in Insertionsort, Folie 7 in Kapitel 1).



Anschaulich: Auf dem Weg von $\texttt{A}[i]$ zur Wurzel werden alle Elemente, deren Schlüssel größer als der (alte) Eintrag in $\texttt{A}[i]$ ist, um eine Position (auf dem Weg) nach unten gezogen.

Eintrag $\texttt{A}[i]$ landet in der freigewordenen Position.

Man kann dies auch effizienter programmieren (wie in Insertionsort, Folie 7 in Kapitel 1).

Behauptung 6.5.1

In `A[`$1..n$`]` ist höchstens `A[`i`]` zu klein

$\Rightarrow$ nach Ausführung von **bubbleUp**(i) ist `A[`$1..n$`]` ein Heap.

Der Zeitaufwand ist $O(\log i)$, die Anzahl der Schlüsselvergleiche ist höchstens das Level von i im Baum, also die Anzahl der Bits in der Binärdarstellung $\mathrm{bin}(i)$, d. h. $\lceil \log(i+1) \rceil$.

Beweis: Ähnlich wie bei **bubbleDown** (siehe Druckfolien).

Beweis der Korrektheit von **bubbleUp**(i): Durch Induktion über die Runden zeigt man:

(IB$_j$) Zu Beginn des Schleifendurchlaufs für j ist höchstens $\texttt{A}[j]$ zu klein.

Zu Beginn, für $j = i$, stimmt dies nach Annahme über die Eingabe.
Nun betrachte einen Schleifendurchlauf der **while**-Schleife. Die I.V. sagt, dass ein Heap entsteht, wenn man u in Knoten j durch einen geeigneten Schlüssel $x \geq u$ ersetzt.

1. Fall: $j \geq 2$ und $u = \texttt{A}[j]\texttt{.key} < v = \texttt{A}[\lfloor j/2 \rfloor]\texttt{.key}$; es wird getauscht.
Nun steht u in Knoten $\lfloor j/2 \rfloor$ und v im Kind j.

Beh.: Wenn man den Schlüssel u in $\lfloor j/2 \rfloor$ auf v erhöht, entsteht ein Heap (also gilt (IB$_{\lfloor j/2 \rfloor}$)).

„Alte Situation“: x in Knoten j und v in Knoten $\lfloor j/2 \rfloor$: Ist Heap nach I.V., also gilt $x \geq v$.
„Neue Situation“: v in Knoten j und v in Knoten $\lfloor j/2 \rfloor$. (Zu zeigen: Ist Heap.)

Kann durch die Änderung von x auf v in Knoten j die Heapeigenschaft verloren gehen? Sicher nicht zwischen j und $\lfloor j/2 \rfloor$ (derselbe Schlüssel v!). Die Schlüssel in Knoten $2j$ und $2j + 1$ (falls sie existieren) sind $\geq x$, und es gilt $x \geq v$ (nach I.V.), also gilt die Heapbedingung auch hier.

2. Fall: $j = 1$ oder $u = \texttt{A}[j]\texttt{.key} \geq v = \texttt{A}[\lfloor j/2 \rfloor]\texttt{.key}$, der Algorithmus endet.
(IB$_j$) besagt, dass wir $u = \texttt{A}[j]\texttt{.key}$ durch ein $x \geq u$ ersetzen können, so dass $\texttt{A}[1\,..\,k]$ Heap wird. Dann liegt aber auch mit Schlüssel u in Knoten j schon ein Heap vor, weil es keinen Konflikt mit dem Vorgänger von Knoten j gibt. □

Prozedur insert(x, d)

// Einfügen eines neuen Eintrags in Priority-Queue

// Ausgangspunkt: Pegel `n` enthält n, `A[`$1 \,..\, n$`]` ist Heap.

(1) **if** `n` $= m$ **then** „Überlaufbehandlung“; // z. B. Verdopplung

(2) `n++`;

(3) `A[n]` $\leftarrow (x, d)$;

(4) **bubbleUp**(`n`).

Korrektheit: Klar wegen Korrektheit von **bubbleUp**. **Zeitaufwand:** $O(\log n)$.

Wir können **bubbleUp**(i) sogar für eine etwas allgemeinere Operation verwenden (Korrektheitsbeweis gilt weiter):

Wir ersetzen einen beliebigen Schlüssel im Heap (Position i) durch einen kleineren.

Mit **bubbleUp**(i) kann die Heapeigenschaft wieder hergestellt werden.

Prozedur decreaseKey(x, i)

// (Erniedrigen des Schlüssels an Arrayposition i auf x)

(1) **if** `A[`i`].key` $< x$ **then** Fehlerbehandlung;

(2) `A[`i`].key` $\leftarrow x$;

(3) **bubbleUp**(i).

// Zeitaufwand: $O(\log(i))$

SimplePriorityQueue plus „decreaseKey“:

Prioritätswarteschlange

Satz 6.5.2

Der Datentyp “Prioritätswarteschlange” kann mit Hilfe eines Heaps implementiert werden.

Dabei erfordern *empty* und *isempty* (und das **Ermitteln** des kleinsten Eintrags) konstante Zeit

und *insert*, *extractMin* und *decreaseKey* benötigen jeweils Zeit $O(\log n)$.

Dabei ist n jeweils der aktuelle Pegelstand, also die Anzahl der Einträge in der Prioritätswarteschlange.

Technisches Problem, noch zu klären:

Wie soll man der Datenstruktur „mitteilen“, **welches Objekt gemeint ist**, wenn *decreaseKey* auf einen Eintrag angewendet werden soll?

Bei (binärem) Heap: Die Positionen der Einträge im Array ändern sich die ganze Zeit, durch die durch **insert** und **extractMin** verursachten Verschiebungen im Array.

Technisch unsauber wäre es, dem Benutzer der Datenstruktur stets mitzuteilen, an welcher Stelle im Array ein Eintrag sitzt.
Dies widerspräche eklatant dem Prinzip der Kapselung eines Datentyps. (Im Kontext von Algorithmen wird allerdings manchmal so verfahren, dass der benutzende Algorithmus Zugriffsrechte in die Datenstruktur hinein hat.)

Damit zusammenhängend: Die **Spezifikation** des Datentyps „Prioritätswarteschlange“ ist nicht ganz offensichtlich.

Besser: Erweiterung der Datenstruktur: Objekt (x, d) erhält bei Ausführung von **insert**(x, d) eine **eindeutige** und **unveränderliche „Identität“** p zugeteilt, über die dieses Objekt angesprochen werden kann.
(Damit können sogar **verschiedene Kopien** ein und desselben Datensatzes **unterschieden** werden!)

p wird dem Benutzer als Reaktion auf die Einfügung mitgeteilt.

Beispiel: Datensatz = Schlüssel = Eintrag aus $\{\mathtt{A}, \ldots, \mathtt{Z}\}$.

Wir vergeben **Nummern** $1, 2, 3, \ldots$ als Identitäten. Der **Benutzer** der Datenstruktur muss sicherstellen, dass Identitäten **nicht mehr benutzt** werden, nachdem das entsprechende Objekt (durch ein *extractMin*) wieder aus der Datenstruktur verschwunden ist.

Runde	Operation	innerer Zustand	Ausgabe
0	*empty*	$\emptyset$	–
1	*insert*(D)	$\{(1,\mathtt{D})\}$	1
2	*insert*(B)	$\{(1,\mathtt{D}), (2,\mathtt{B})\}$	2
3	*insert*(D)	$\{(1,\mathtt{D}), (2,\mathtt{B}), (3,\mathtt{D})\}$	3
4	*isempty*	$\{(1,\mathtt{D}), (2,\mathtt{B}), (3,\mathtt{D})\}$	*false*
5	*insert*(E)	$\{(1,\mathtt{D}), (2,\mathtt{B}), (3,\mathtt{D}), (4,\mathtt{E})\}$	4
6	*insert*(G)	$\{(1,\mathtt{D}), (2,\mathtt{B}), (3,\mathtt{D}), (4,\mathtt{E}), (5,\mathtt{G})\}$	5
7	*decreaseKey*(5,D)	$\{(1,\mathtt{D}), (2,\mathtt{B}), (3,\mathtt{D}), (4,\mathtt{E}), (5,\mathtt{D})\}$	–
⋮	⋮	⋮	⋮

Runde	Operation	innerer Zustand	Ausgabe
⋮	⋮	⋮	⋮
		$\{(1,\mathtt{D}),(2,\mathtt{B}),(3,\mathtt{D}),(4,\mathtt{E}),(5,\mathtt{D})\}$	
8	*extractMin*	$\{(1,\mathtt{D}),(3,\mathtt{D}),(4,\mathtt{E}),(5,\mathtt{D})\}$	$(2,\mathtt{B})$
9	*extractMin*	$\{(1,\mathtt{D}),(4,\mathtt{E}),(5,\mathtt{D})\}$	$(3,\mathtt{D})$
10	*decreaseKey*$(4,\mathtt{B})$	$\{(1,\mathtt{D}),(4,\mathtt{B}),(5,\mathtt{D})\}$	–
11	*extractMin*	$\{(1,\mathtt{D}),(5,\mathtt{D})\}$	$(4,\mathtt{B})$
12	*insert*$(\mathtt{F})$	$\{(1,\mathtt{D}),(5,\mathtt{D}),(3,\mathtt{F})\}$	3
13	*decreaseKey*$(3,\mathtt{C})$	$\{(1,\mathtt{D}),(5,\mathtt{D}),(3,\mathtt{C})\}$	–
14	*extractMin*	$\{(1,\mathtt{D}),(5,\mathtt{D})\}$	$(3,\mathtt{C})$
15	*extractMin*	$\{(1,\mathtt{D})\}$	$(5,\mathtt{D})$
16	*extractMin*	$\emptyset$	$(1,\mathtt{D})$
17	*isempty*	$\emptyset$	$true.$

Die in Runde 9 freigewordene Identität 3 wird in Runde 12 wieder vergeben. In Runde 13 wäre zum Beispiel die Operation *decreaseKey*$(4,\mathtt{A})$ **nicht legal**, da das (vorher existierende) Objekt mit Identität 4 in Runde 11 aus der Datenstruktur verschwunden ist.

Das Beispiel ist als Vorbild für ein mathematisches Modell geeignet.

Technische Realisierung (für Interessierte):

Allgemein: Jedes im Heap gespeicherte Objekt ((Daten, Schlüssel)-Paar plus Zusatzinformationen) steht während seiner gesamten Lebensdauer an einer festen, unveränderlichen Stelle im Speicher. Zugriffe auf ein solches Objekt erfolgen über eine Referenz (bzw. einen Zeiger). Im Heaparray werden nur Referenzen/Zeiger auf diese Objekte gehalten.

Version 1: Die Identität p **ist** die Referenz/der Zeiger auf das „echte“ Objekt im Speicher.
Das Objekt enthält seine aktuelle Position im Heap-Array als Komponente. (Diese muss bei **bubbleUp** und **bubbleDown** stets mit aktualisiert werden.) – **Nachteil:** Die Kapselung der Datenstruktur ist durchbrochen, da der Benutzer über den Zeiger/die Referenz direkt auf die Objekte zugreifen kann.

Version 2: Jedes Objekt erhält eine beliebige (laufende) Nummer (z. B. durch Durchzählen der Einfügungen). Diese Nummer ist dann die Identität p des Objekts. Die Datenstruktur hält in einem („privaten“, also internen) Hilfsarray für jede dieser Nummern die Referenz auf das entsprechende Objekt. Das Objekt selbst enthält neben Daten und Schlüssel auch seine aktuelle Position im Heaparray. – **Nachteil:** Der Platzbedarf des Hilfsarrays entspricht der Anzahl der Einfügungen, nicht der maximalen Größe der Prioritätswarteschlange.

Version 3: Wie Version 2, aber die Identität p eines durch **extractMin** aus dem Heap gelöschten Eintrags kann wieder vergeben werden. Dann muss die Datenstruktur zusätzlich die verfügbaren Identitäten verwalten, zum Beispiel mit einem Stack, und Heapeinträge enthalten die Identität.

Datentyp: PriorityQueue (Prioritätswarteschlange)

1. Signatur:

Sorten: *Keys*
Data
Id // „Identitäten“
PrioQ
Boolean

Operationen: $empty: \rightarrow PrioQ$
$isempty: PrioQ \rightarrow Boolean$
$insert: PrioQ \times Keys \times Data \rightarrow PrioQ \times Id$
$extractMin: PrioQ \rightarrow PrioQ \times Id \times Keys \times Data$
$decreaseKey: PrioQ \times Id \times Keys \rightarrow PrioQ$

Mathematisches Modell:

Sorten:

Keys	:	$(U, <)$	// totalgeordnetes „Universum“
Data	:	D	// Menge von „Datensätzen“
Id	:	$\mathbb{N} = \{0, 1, 2, \ldots\}$	// unendliche Menge möglicher „Identitäten“
Boolean	:	$\{false, true\}$	
PrioQ	:	die Menge aller Funktionen $f : X \to U \times D$, wobei $X = \mathit{Def}(f) \subseteq \mathbb{N}$ endlich ist	

Operationen:

empty$() = \emptyset$ // die *leere Funktion*

$$\textit{isempty}(f) = \begin{cases} true, & \text{für } f = \emptyset \\ false, & \text{für } f \neq \emptyset \end{cases}$$

$\textit{insert}(x, d, f) = (f \cup \{(p, (x, d))\}, p)$, für ein $p \in \mathbb{N} - \mathrm{Def}(f)$

$$\textit{extractMin}(f) = \begin{cases} \textit{undefiniert}, & \text{für } f = \emptyset \\ (f', p_0, x_0, d_0), & \text{für } f \neq \emptyset, \end{cases}$$

wobei im zweiten Fall
$x_0 = \min\{x \mid \exists p\, \exists d\colon f(p) = (x, d)\}$
und $f(p_0) = (x_0, d_0)$
und $f' := f - \{(p_0, (x_0, d_0))\}$.

$\textit{decreaseKey}(p, x) = (f - \{(p, (y, d))\}) \cup \{(p, (x, d))\}$,
falls $f(p) = (y, d)$ mit $x \leq y$ (sonst Fehler)

Implementierung: Größere Übungsaufgabe.

Vorlesungsvideo:

Untere Schranken

6.6 Untere Schranke für Sortieren

Ein Sortieralgorithmus $\mathcal{A}$ heißt ein **Schlüsselvergleichsverfahren**, wenn

in $\mathcal{A}$ auf Schlüssel x, y aus U nur Operationen

$$x < y \,? \qquad x \leq y \,? \qquad x = y \,?$$

sowie Verschieben und Kopieren angewendet werden. Schlüssel sind also „atomare Objekte“, die als Semantik nur ihre Rolle in der Ordnung $(U, <)$ haben.

Gegensatz wäre: Die Binärdarstellung eines zahlenwertigen Schlüssels wird gelesen und verwendet.

Mergesort, Quicksort, Heapsort, Insertionsort sind Schlüsselvergleichsverfahren.

Wir haben gesehen: Diese Verfahren benötigen $O(n \log n)$ oder $O(n^2)$ Vergleiche.

Ziel in diesem Abschnitt:

Schlüsselvergleichsverfahren können **nicht schneller als** in $\boldsymbol{\Omega(n \log n)}$ Vergleichen sortieren.

Sei $\mathcal{A}$ ein Schlüsselvergleichsverfahren.

Wir denken uns $\mathcal{A}$ auf die $n!$ verschiedenen Eingaben $(a_1, 1), \ldots, (a_n, n)$ angewendet, wobei $\sigma = (a_1, \ldots, a_n)$ eine Permutation von $\{1, \ldots, n\}$ ist.

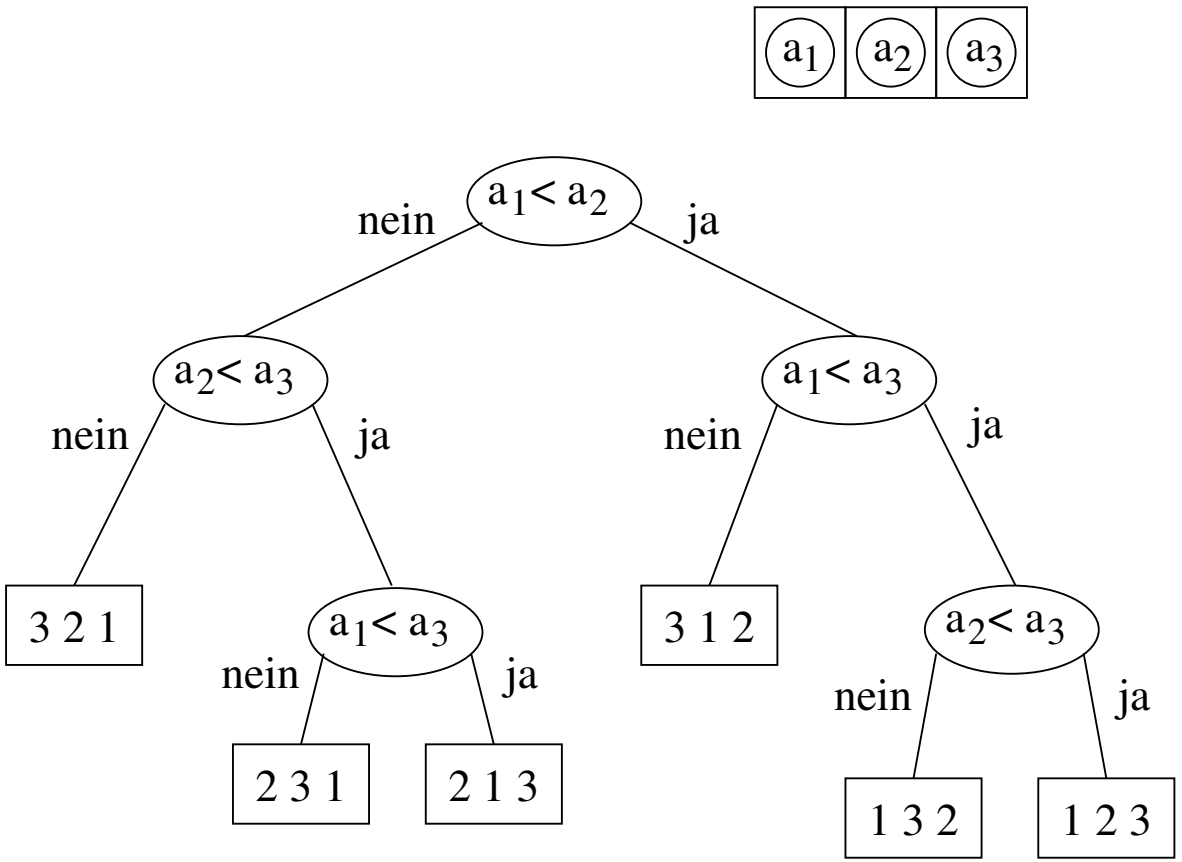
Der Sortierschlüssel ist dabei die erste Komponente. (Die zweite wird nur „mitgeschleift“.)

Beispiel: Aus $(2, 1), (4, 2), (1, 3), (3, 4)$ wird $(1, 3), (2, 1), (3, 4), (4, 2)$.

Effekt: In den zweiten Komponenten steht die Permutation $\pi = \sigma^{-1}$ (im Beispiel $\pi = (3, 1, 4, 2)$), die die Eingabe $(a_1, \ldots, a_n)$ sortiert: $a_{\pi(1)} < \cdots < a_{\pi(n)}$.

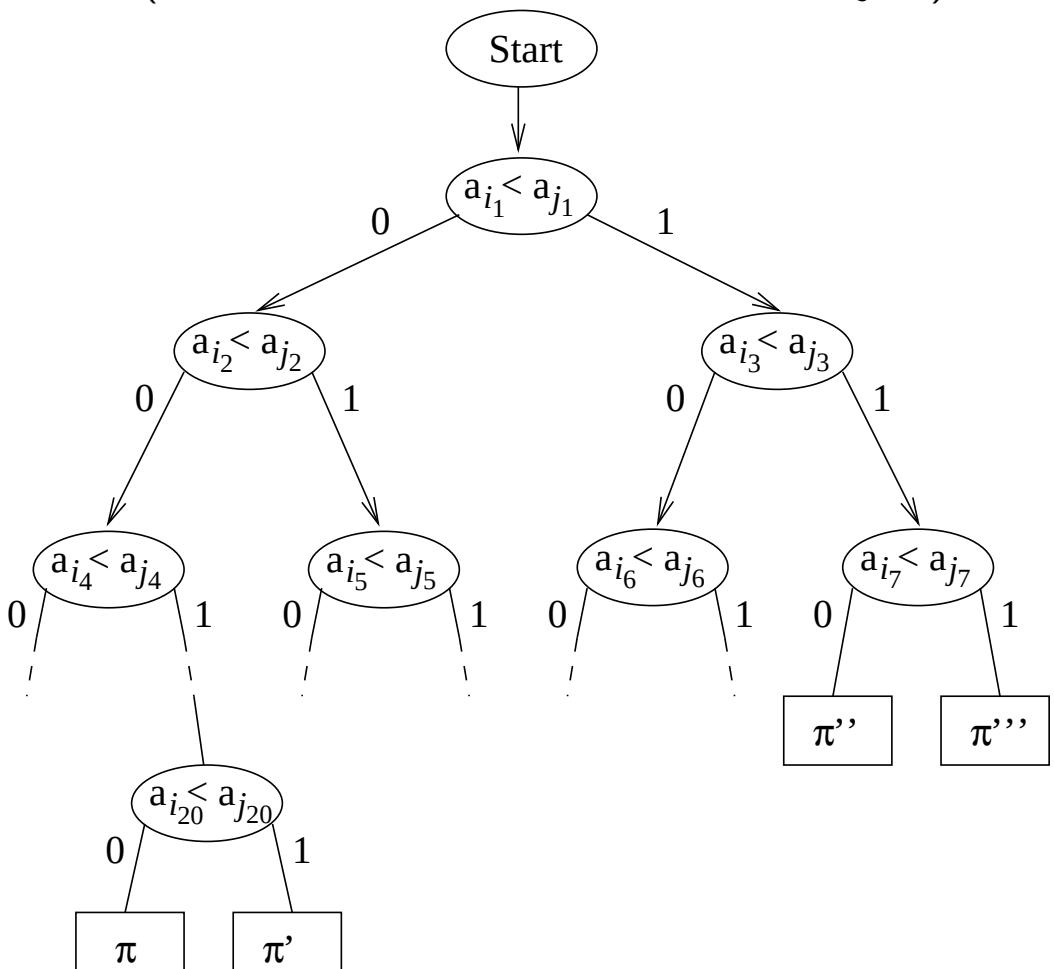
Daraus folgt: Algorithmus $\mathcal{A}$ erzeugt auf den genannten $n!$ verschiedenen Eingaben $n!$ verschiedene Ausgaben, eine für jede Anordnung σ der Eingabeobjekte.

Beispiel: Alle Verläufe für **Mergesort** mit drei Elementen.



In Blättern: die zweiten Komponenten

Allgemein: Ausführen von Algorithmus $\mathcal{A}$ auf allen diesen Inputs führt zu einem **Vergleichsbaum** $T_{\mathcal{A},n}$ mit genau $n!$ Blättern. („0“ bedeutet „nein“, „1“ bedeutet „ja“.)



Allgemein: Jeder vergleichsbasierte Sortieralgorithmus $\mathcal{A}$ erzeugt für jedes n einen **Vergleichsbaum** $T_{\mathcal{A},n}$ mit genau $n!$ Blättern. Die Wege im Baum von der Wurzel zu den Blättern entsprechen den Berechnungen von $\mathcal{A}$ auf den $n!$ genannten Eingaben; die Knoten auf einem solchen Weg entsprechen den in der Berechnung ausgeführten Vergleichen.

Nur Vergleiche „$a_i < a_j$?“ sind überhaupt relevant, wenn wir uns auf die $n!$ vielen vorher beschriebenen Eingaben mit n verschiedenen Komponenten konzentrieren.
Für jede Teilmenge J aller dieser Inputs definieren wir durch Induktion einen Baum T_J, wie folgt.
Wenn J nur ein Element $((a_1, 1), \ldots, (a_n, n))$ hat, dann ist T_J ein mit π beschriftetes Blatt, wobei π die Permutation ist, die $(a_1, \ldots, a_n)$ sortiert.
Wenn J zwei oder mehr Elemente hat, lassen wir $\mathcal{A}$ auf allen Inputs aus J ablaufen, bis zum ersten Vergleich „$a_i < a_j$?“, der nicht bei allen Eingaben in J zu demselben Ergebnis in $\{\text{ja}, \text{nein}\}$ führt. Weil $\mathcal{A}$ vergleichsbasiert ist, gibt es keine anderen Verzweigungen, also arbeitet $\mathcal{A}$ auf allen Eingaben in J bis zu diesem Punkt genau gleich.
Wir bilden $J_0 := \{x \in J \mid \text{Test „}a_i < a_j\text{?“ bei Input } x \text{ liefert „nein“}\}$ und $J_1 := J - J_0$. Baum T_J besteht aus einer Wurzel, die mit „$a_i < a_j$?“ beschriftet ist, sowie T_{J_0} als linkem Unterbaum und T_{J_1} als rechtem Unterbaum.
Man sieht dann, dass jeder Input $(a_1, 1), \ldots, (a_n, n)$ zu genau einem Blatt gehört, und die Knoten auf dem Weg von der Wurzel zu diesem Blatt den Vergleichen entsprechen, die auf diesem Input ausgeführt werden.

Satz 6.6.1

Wenn $\mathcal{A}$ ein Schlüsselvergleichsverfahren ist, das das Sortierproblem für beliebige Inputs mit n Komponenten löst, dann gibt es eine Eingabe $(a_1, \ldots, a_n)$ (eine Permutation von $\{1, \ldots, n\}$), auf der $\mathcal{A}$ mindestens $\lceil \log(n!) \rceil$ Vergleiche ausführt.
Dies sind $n \log n - O(n)$ viele.

Beweis: Der Vergleichsbaum, den $\mathcal{A}$ erzeugt, hat $n!$ (externe) Blätter, also hat er Tiefe $\geq \lceil \log(n!) \rceil$ (s. Prop. 3.3.2(d)). Wir zeigen gleich: $\log(n!) = n \log n - O(n)$.□

Bemerkung:

$n! \geq (n/e)^n$. *Beweis*: $n^n/n! < \sum_{i \geq 0} \frac{n^i}{i!} = e^n$. □

Also: $\log(n!) \geq \log((n/e)^n) = n \log n - n \log e = \boldsymbol{n \log n} - 1{,}44269 \ldots n$.

Andersherum: $\ln(n!) = \sum_{1 \leq i \leq n} \ln i \leq \ln n + \int_1^n \ln x \, dx =$
$\ln n + [x(\ln x - 1)]_1^n = \ln n + n \ln n - n + 1$, also $n! \leq (en)(n/e)^n$.

Mitteilung:

Für $n \geq 1$ gilt (**Stirlingsche Formel**):

$$\sqrt{2\pi n} \left(\frac{n}{e}\right)^n \leq n! \leq \sqrt{2\pi n} \left(\frac{n}{e}\right)^n e^{\frac{1}{12n}}$$

Beispiel: $3598695{,}62 < 10! = 3628800 < 3628810{,}05$.

Bemerkung: Die obere Schranke ist der bessere Schätzwert.

Satz 6.6.2

Sei $\mathcal{A}$ ein Schlüsselvergleichsverfahren, das das Sortierproblem für alle Inputs mit n Komponenten löst, dann gilt für die **mittlere Vergleichsanzahl $\overline{V}_n$** (gemittelt über die $n!$ verschiedenen Anordnungen von $\{1, \ldots, n\}$, jede gleich wahrscheinlich):

$$\overline{V}_n \geq \log(n!).$$

Beweis: Der Vergleichsbaum $T_{\mathcal{A},n}$, den $\mathcal{A}$ erzeugt, hat $N = n!$ (externe) Blätter.

$T_{\mathcal{A},n}$ hat N externe Knoten $\Rightarrow$ **TEPL**$(T_{\mathcal{A},n}) \geq N \log N$, nach Prop. 3.3.5.

Also ist die **mittlere äußere Weglänge** $\frac{1}{N}$TEPL$(T_{\mathcal{A},n})$ mindestens $\log N$.

Daraus: Die mittlere Anzahl von Vergleichen ist $\geq \log N = \log(n!)$. □

Der nächste Satz besagt, dass randomisierte Sortierverfahren wie randomisiertes Quicksort zwar Vorteile haben, insbesondere worst-case-Inputs vermeiden können, aber keine Handhabe bilden, um an der unteren Schranke $\log(n!)$ vorbeizukommen.

Satz 6.6.3

Wenn $\mathcal{A}$ ein Schlüsselvergleichsverfahren ist, das das Sortierproblem für alle Inputs mit n Komponenten löst und dabei Randomisierung benutzt (d. h. Zufallsexperimente ausführt wie etwa Randomisiertes Quicksort), dann **gibt es eine Eingabe** $a = (a_1, \ldots, a_n)$ (nämlich eine Permutation von $\{1, \ldots, n\}$), so dass die **erwartete** Vergleichsanzahl von $\mathcal{A}$ auf a (**gemittelt über die Zufallsexperimente** von $\mathcal{A}$) mindestens $\log(n!)$ ist.

Beweis: (Nicht prüfungsrelevant, für Interessierte.) Man kann sich vorstellen, dass man alle Zufallsexperimente, die in $\mathcal{A}$ überhaupt vorkommen können, vor Beginn des Ablaufs in einem großen zusammenfassenden Zufallsexperiment ausführt. (Beispiel **randomisiertes Quicksort**: Für jedes Paar (a, b), $1 \leq a < b \leq n$, wird **vorab** zufällig eine Position $s_{(a,b)} \in [a..b]$ gewählt. Wenn dann im Ablauf des Algorithmus im Teilarray $\mathtt{T}[a..b]$ ein Pivot gewählt werden muss, benutzt man $s_{(a,b)}$.)

Ergebnis ist ein (möglicherweise sehr langes) Zufallswort α.
Jedes solche α hat eine Wahrscheinlichkeit p_α. Dabei gilt $\sum_\alpha p_\alpha = 1$.

$\mathcal{A}_\alpha$: Algorithmus $\mathcal{A}$, der mit dem Ergebnis α des anfänglichen Experiments läuft.
$\mathcal{A}_\alpha$ ist ein gewöhnlicher deterministischer Algorithmus!

$T_\alpha(\sigma) :=$ #(Vergleiche bei $\mathcal{A}_\alpha$ auf Input σ). ($\sigma = (a_1, \ldots, a_n)$: Permutation von $\{1, \ldots, n\}$.)
Nach Satz 6.6.2 gilt: $\frac{1}{n!} \sum_\sigma T_\alpha(\sigma) \geq \log(n!)$.
Summiere über alle α's: $\displaystyle\sum_\alpha p_\alpha \cdot \frac{1}{n!} \sum_\sigma T_\alpha(\sigma) \geq \log(n!)$.

Umstellen der Summation: $\displaystyle\frac{1}{n!} \sum_\sigma \Big(\sum_\alpha p_\alpha \cdot T_\alpha(\sigma) \Big) \geq \log(n!)$.

Daraus folgt: **Es gibt ein** σ mit $\sum_\alpha p_\alpha \cdot T_\alpha(\sigma) \geq \log(n!)$.
$\sum_\alpha p_\alpha \cdot T_\alpha(\sigma)$ ist aber gerade die erwartete Anzahl von Vergleichen von $\mathcal{A}$ auf Eingabe σ. □

Vorlesungsvideo:

Sortieren in Linearzeit

6.7. Sortieren in Linearzeit

Wenn wir Schlüssel nicht als atomares Objekt, sondern z. B. als **Index** benutzen, dann gilt die untere Schranke aus 6.6 nicht.

Wir wollen (für besondere Schlüsselformate, ganze Zahlen oder Strings) in Linearzeit sortieren!

Sei zunächst $U = \{0, 1, \ldots, m-1\} = [m]$.

Gegeben: n Objekte mit Schlüsseln $a_1, \ldots, a_n$ aus U. Sortiere!

Präziser: Gegeben sind in `A[1..n]` n Datenobjekte

$$(a_1, d_1), \ldots, (a_n, d_n)$$

mit Schlüsseln $a_1, \ldots, a_n$ aus U. Sortiere **(stabil)**!

(A) Countingsort – Sortieren durch Zählen

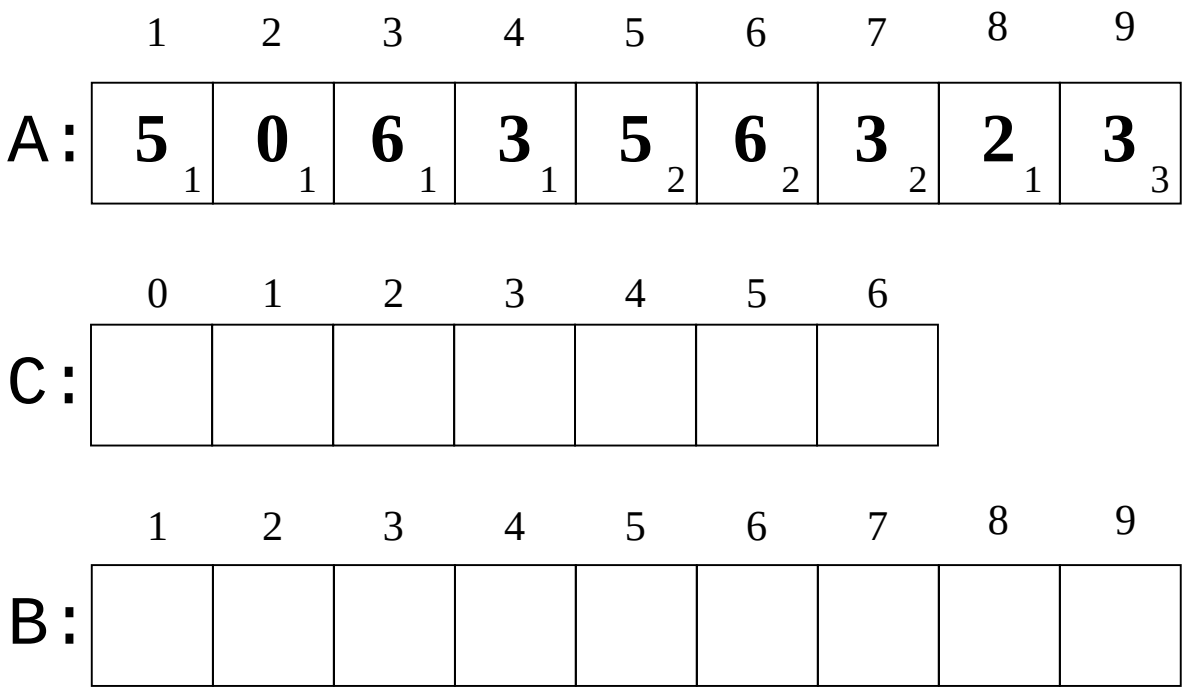
Idee, drei Phasen:

1. **Zähle** in Komponente `C`$[i]$ eines Arrays `C`$[0\,..\,m-1]$, wie oft der Schlüssel i vorkommt.

2. Benutze das Ergebnis von 1., um (wieder in `C`$[i]$) zu berechnen, wie viele Einträge in `A` einen Schlüssel $\leq i$ haben.

3. Übertrage die Einträge mit Schlüssel i aus `A` in die Positionen `C`$[i-1]+1,\ldots,$`C`$[i]$ in Array B, **stabil**.

Beispiel:



1. **Zähle** in Komponente `C[`i`]` eines Arrays `C[`$0..m-1$`]`, wie oft der Schlüssel i vorkommt.

(1) **for** `i` **from** 0 **to** $m-1$ **do** `C[i]` $\leftarrow 0$; // Zeitaufwand: $\Theta(m)$
(2) **for** `j` **from** 1 **to** n **do** `C[A[j].key]++`; // Zeitaufwand: $\Theta(n)$

2. Benutze das Ergebnis von 1., um (wieder in `C[`i`]`) zu berechnen, wie viele Einträge in `A` einen Schlüssel $\leq i$ haben.

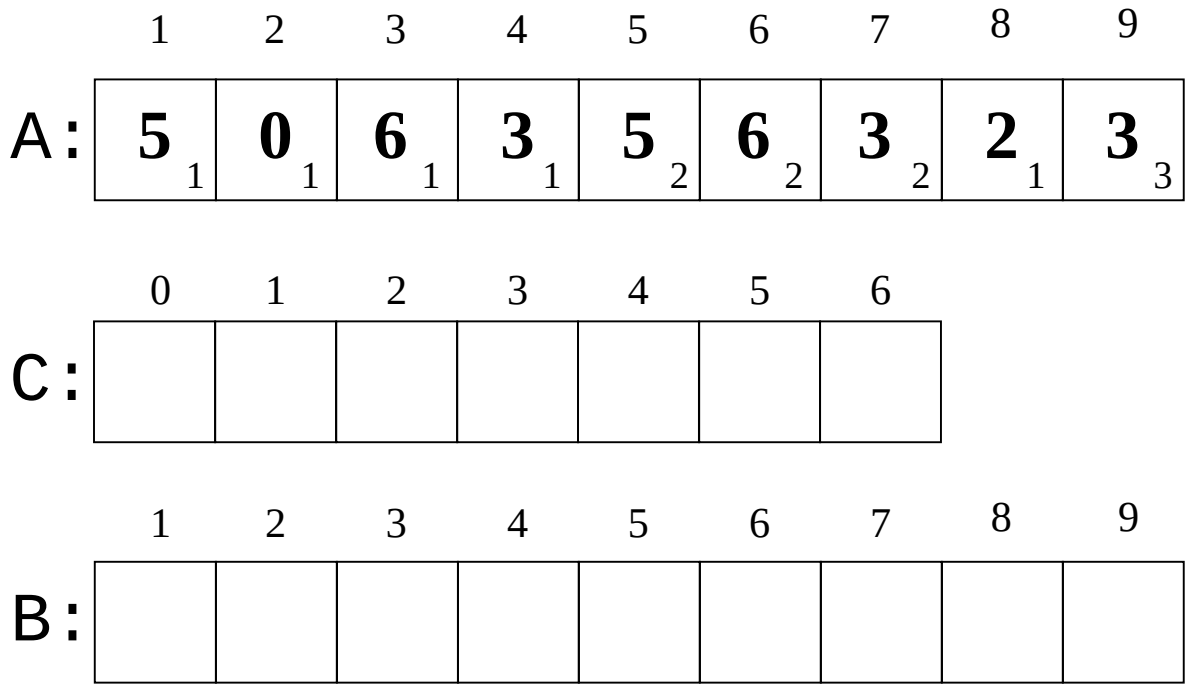
(3) **for** `i` **from** 1 **to** $m-1$ **do** `C[i]` $\leftarrow$ `C[i-1]` $+$ `C[i]`; // Zeitaufwand: $\Theta(m)$

3. Übertrage die Einträge mit Schlüssel i aus `A` in die Positionen `C`$[i-1]+1,\ldots,$`C`$[i]$ in Array B, **stabil**.

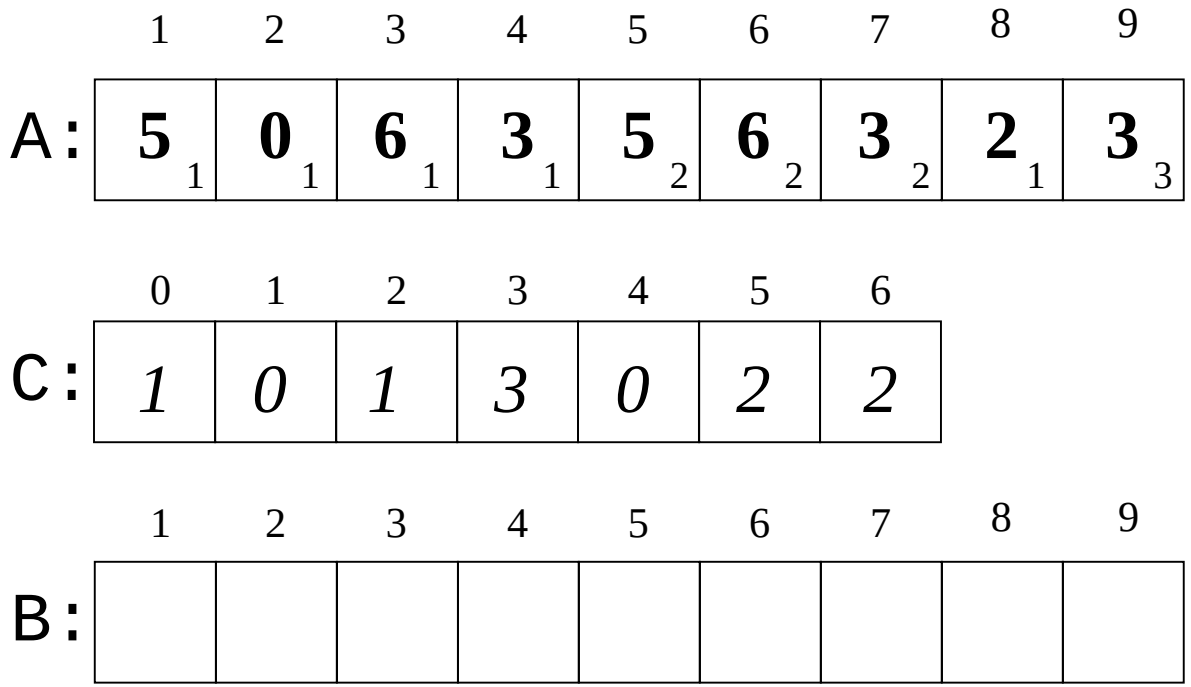
Geschickt: In `A`$[1..n]$ einmal von hinten nach vorne laufen.

Zum Verständnis beobachte: Wenn Eintrag mit Schlüssel i, der in `A` am weitesten rechts steht, an Stelle `A`$[j]$ steht, muss dieser an die Stelle `B[C`$[i]$`]` geschrieben werden. Der **nächste** Eintrag mit diesem Schlüssel muss in das Fach unmittelbar links daneben – daher: Eintrag übertragen, dann `C`$[i]$ herunterzählen. Dies wird iteriert. Man erhält folgendes Vorgehen.

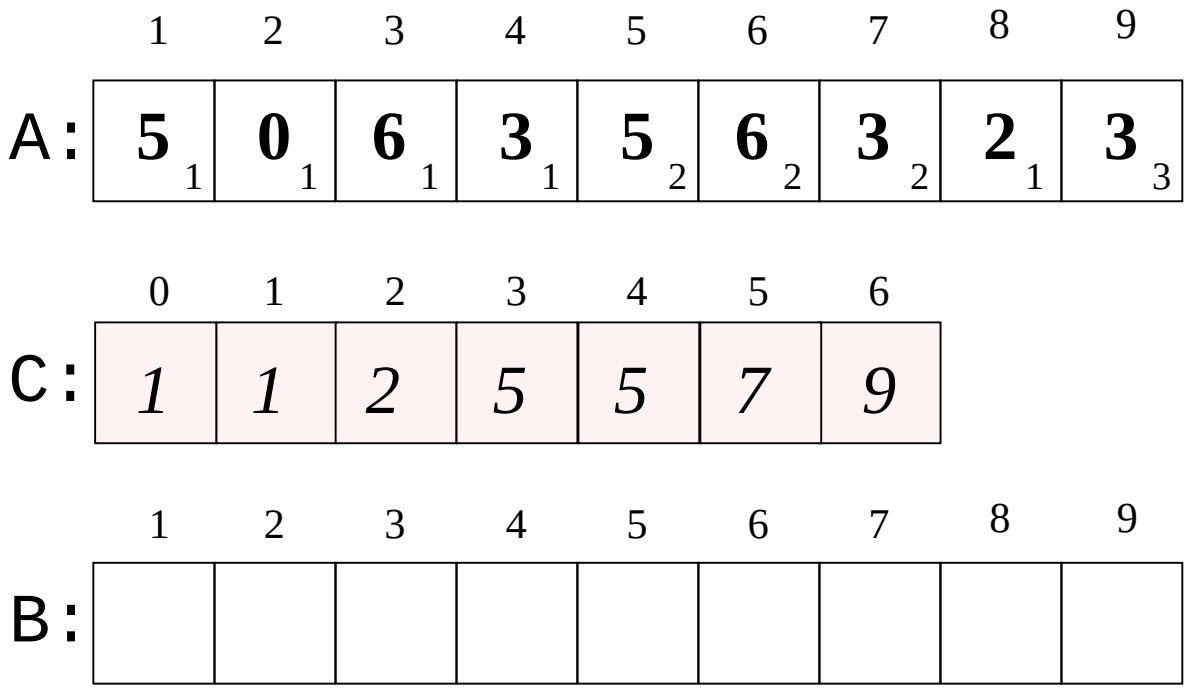
```
(4)  for j from n downto 1 do
(5)      i ← A[j].key;
(6)      B[C[i]] ← A[j];        // aktueller Platz für Schlüssel i (in i)
(7)      C[i]--;                // auf linken Nachbarn umsetzen
                                                  // Zeitaufwand: Θ(n)
```



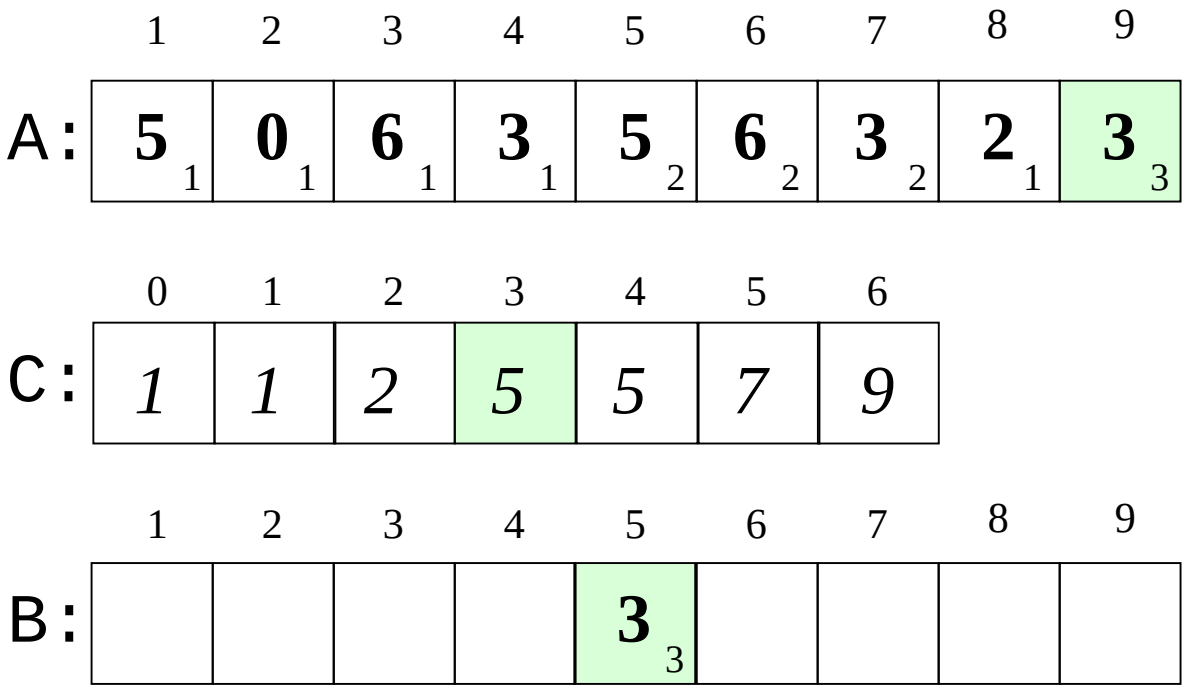
Eingabe, Datenstruktur. Kleine Ziffern in der Eingabe: Reihenfolge identischer Schlüssel.



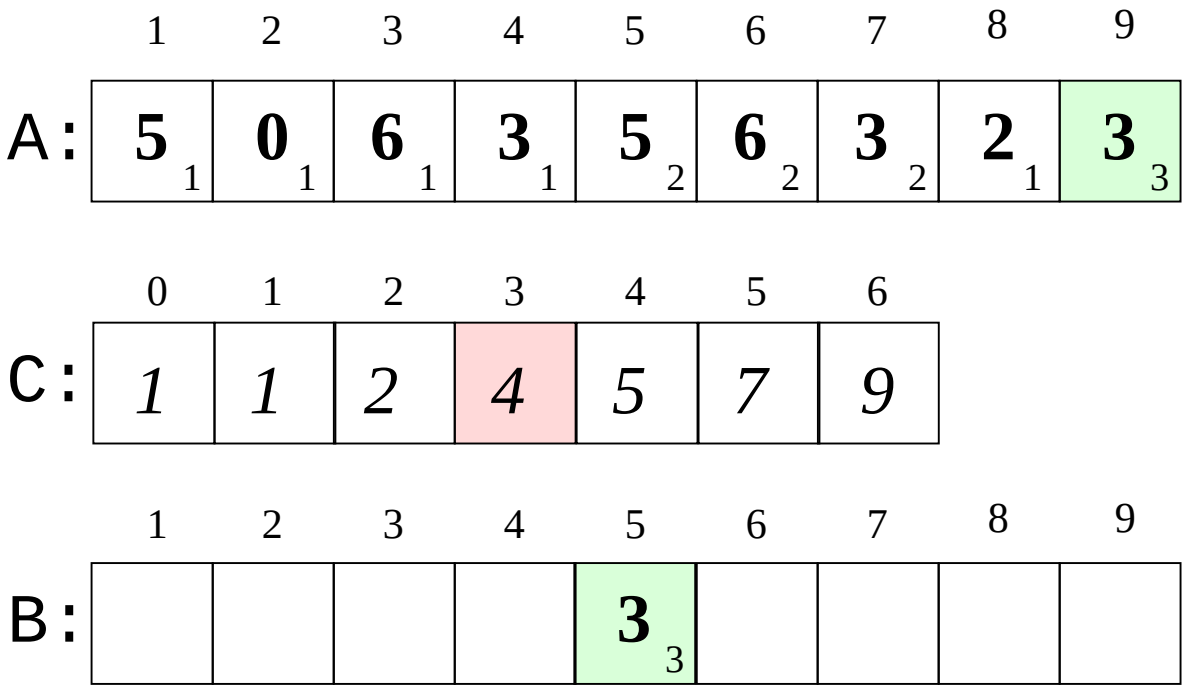
Nach Zeilen (1)–(2): Einträge gezählt.



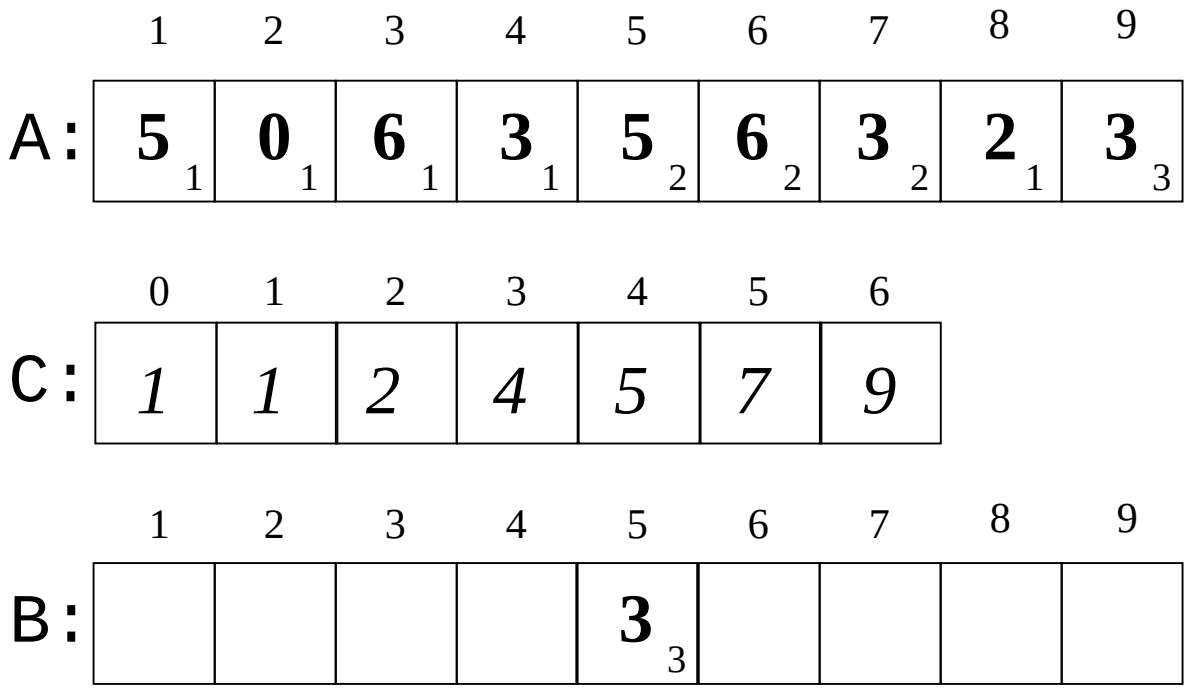
Nach Zeile (3): Endpunkte der Segmente gleicher Schlüssel ermittelt.

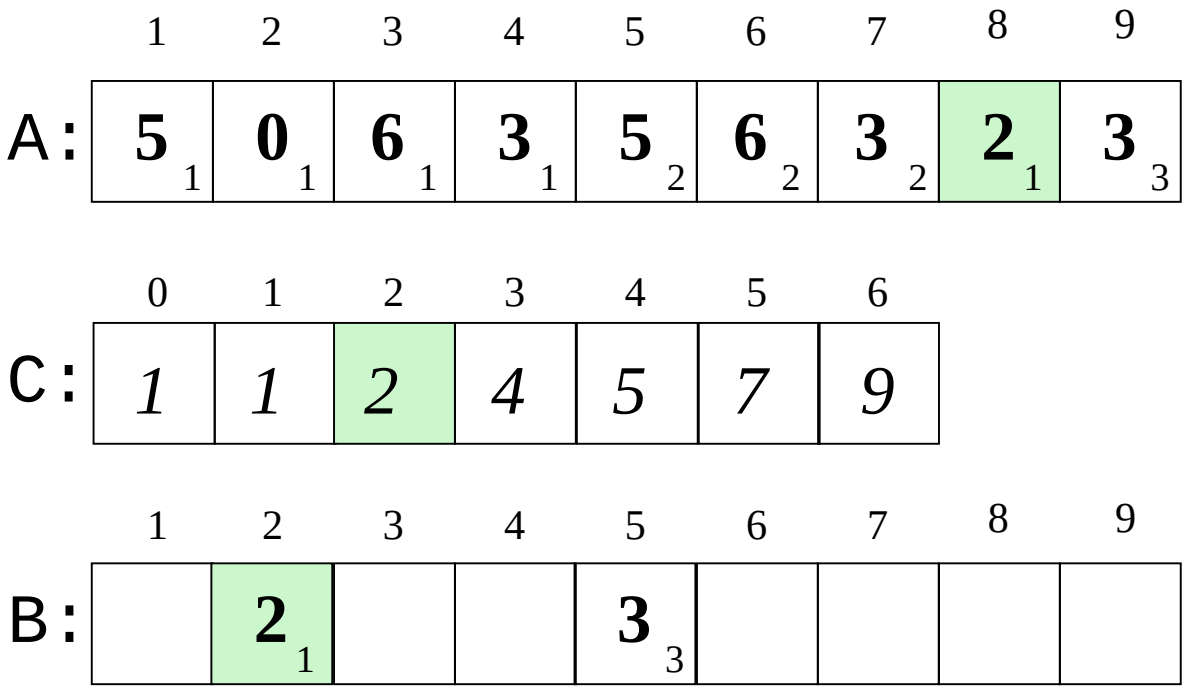


Zeile (6): Eintrag kopieren.

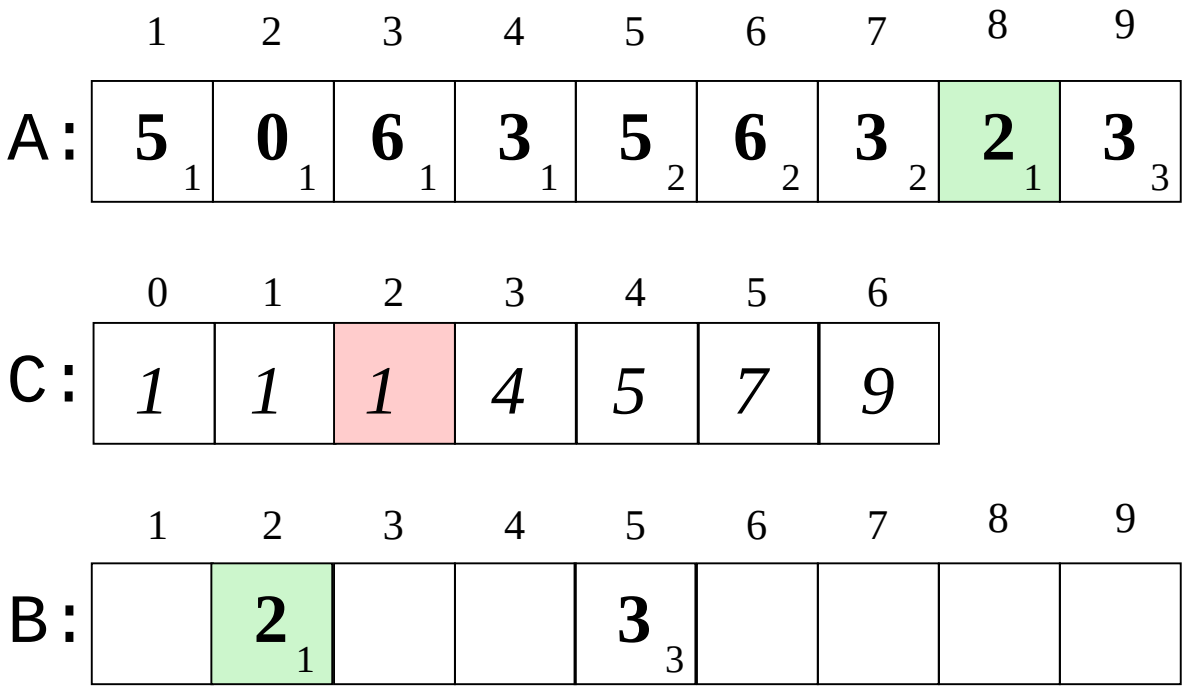


Zeile (7): Index heruntersetzen.

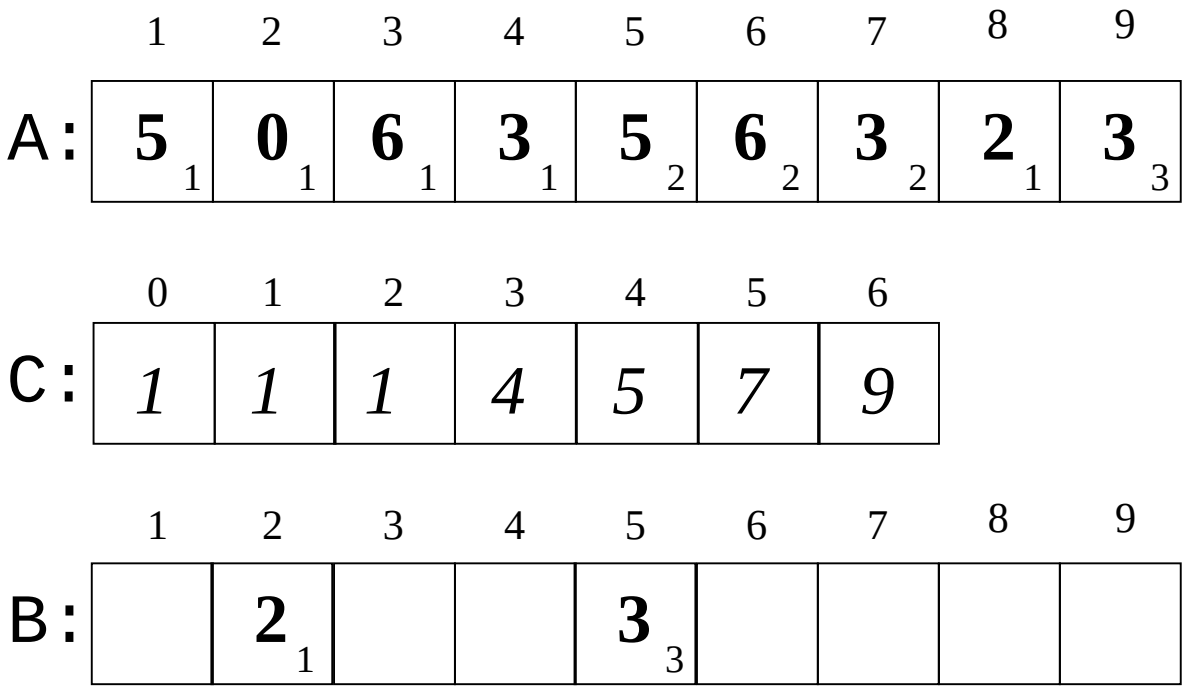

1 2 3 4 5 6 7 8 9
A: 5 1 0 1 6 1 3 1 5 2 6 2 3 2 2 1 3 3
0 1 2 3 4 5 6
C: 1 1 2 4 5 7 9
1 2 3 4 5 6 7 8 9
B: 3 3

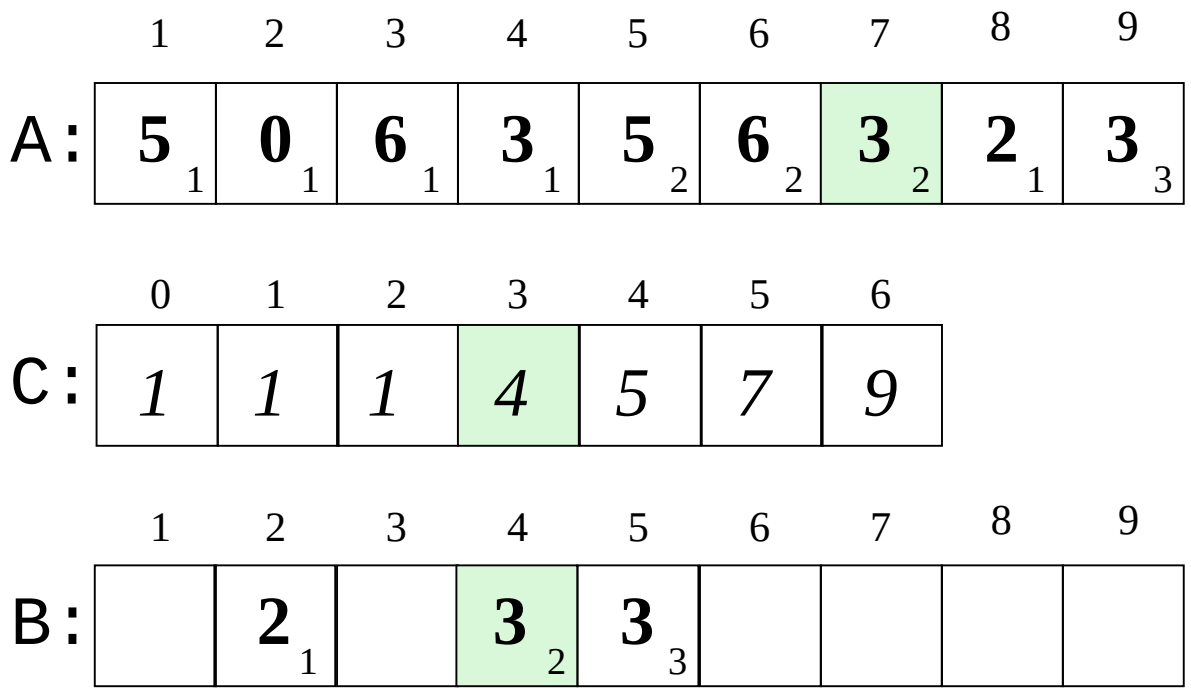


Zeile (6): Eintrag kopieren.

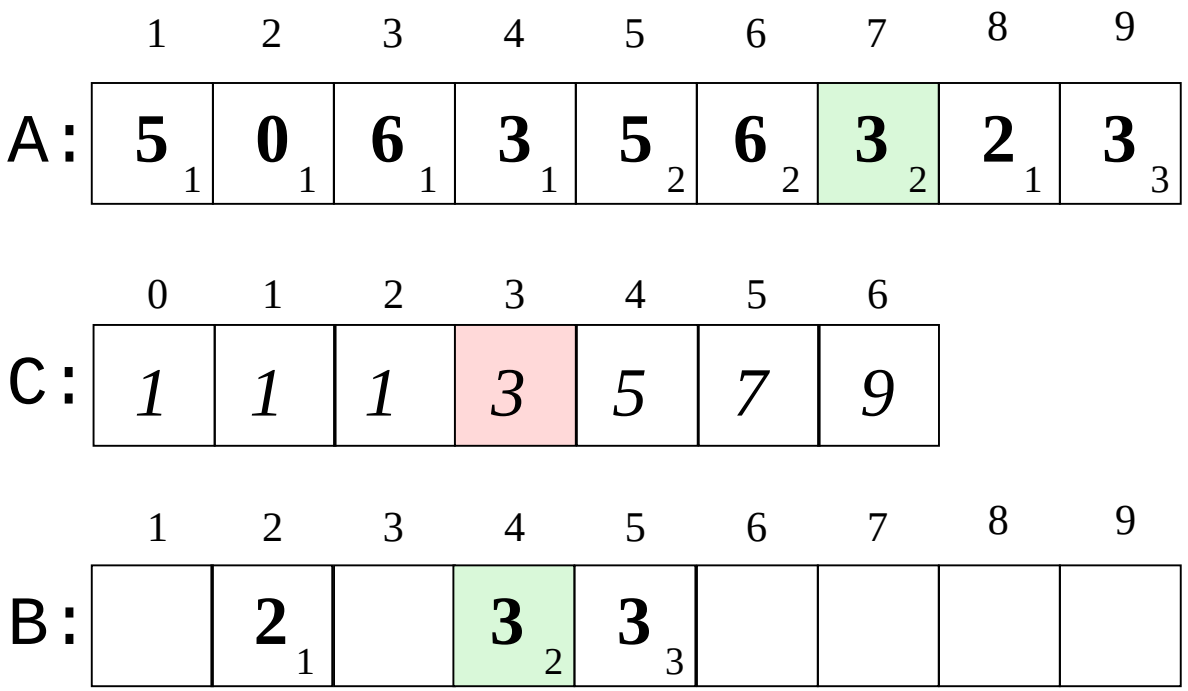


Zeile (7): Index heruntersetzen.

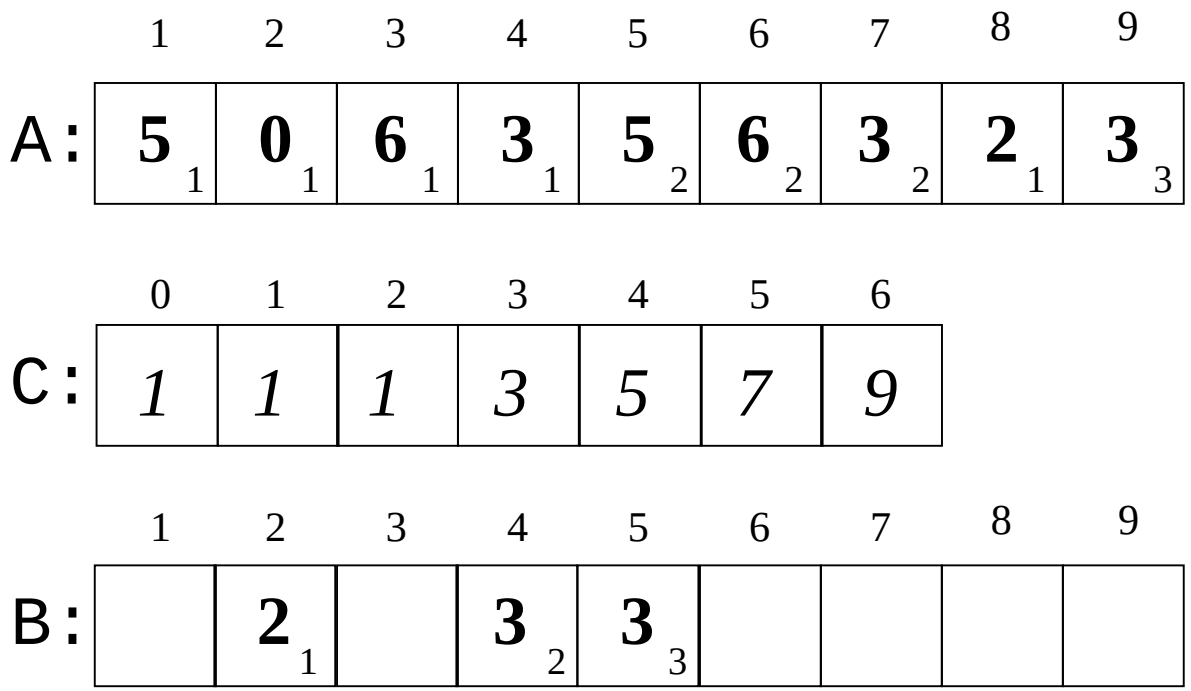

1 2 3 4 5 6 7 8 9
A: 5 1 0 1 6 1 3 1 5 2 6 2 3 2 2 1 3 3
0 1 2 3 4 5 6
C: 1 1 1 4 5 7 9
1 2 3 4 5 6 7 8 9
B: 2 1 3 3

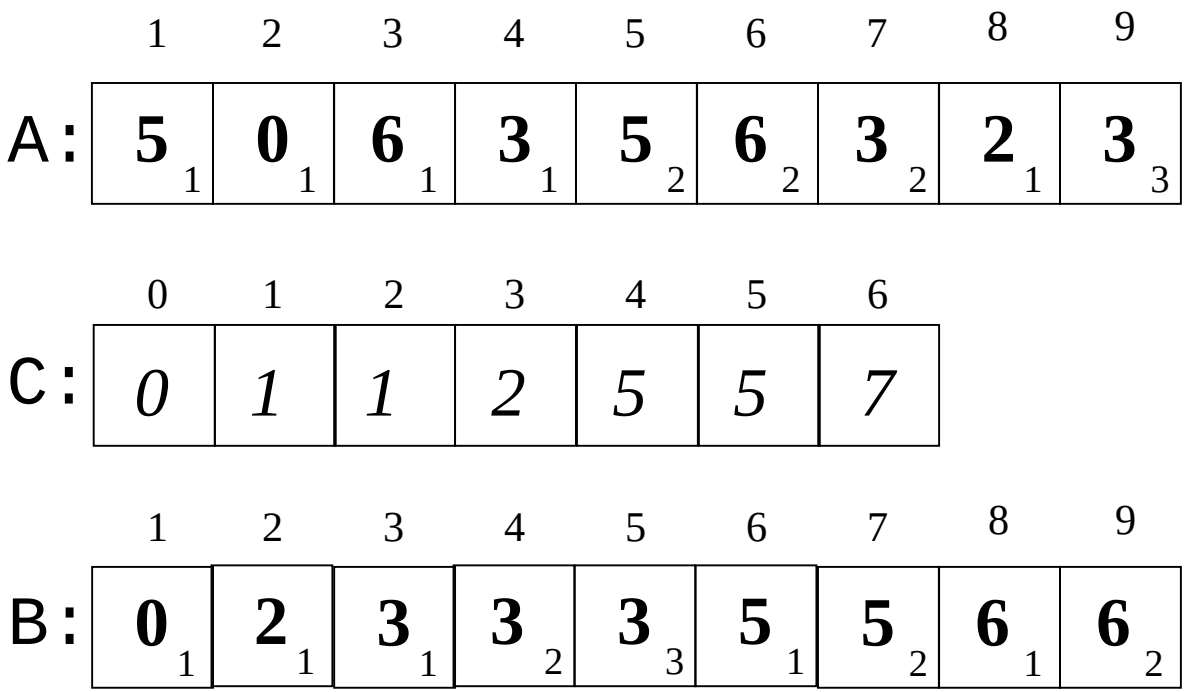


Zeile (6): Eintrag kopieren.



Zeile (7): Index heruntersetzen.


1 2 3 4 5 6 7 8 9
A: 5 1 0 1 6 1 3 1 5 2 6 2 3 2 2 1 3 3
0 1 2 3 4 5 6
C: 1 1 1 3 5 7 9
1 2 3 4 5 6 7 8 9
B: 2 1 3 2 3 3



Resultat nach Beendigung der Schleife (6)–(9).

Satz 6.7.1 (Countingsort – Sortieren durch Zählen)

n Objekte mit Schlüsseln aus $[m]$ (gegeben in Array) können in Zeit $\Theta(n+m)$ und mit Zusatzplatz $\Theta(n+m)$ sortiert werden.

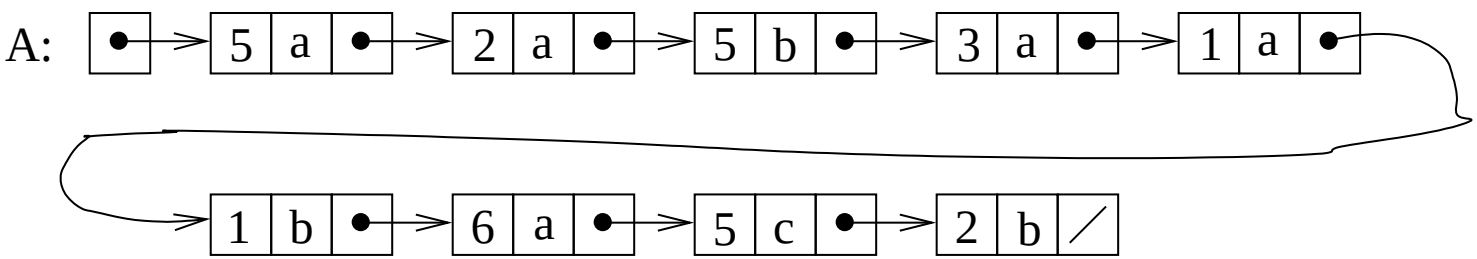
Das Sortierverfahren ist stabil.

Linearzeit und linearer Platz, falls $m \leq cn$, c konstant.

Countingsort ist auch in der **praktischen** Anwendung **sehr schnell**
(wenn auch überhaupt nicht Cache-freundlich).

(B) Bucketsort – Fachsortieren

Nun seien die zu sortierenden Objekte mit Schlüsseln aus $[m]$ als (einfach verkettete) lineare **Liste** gegeben.

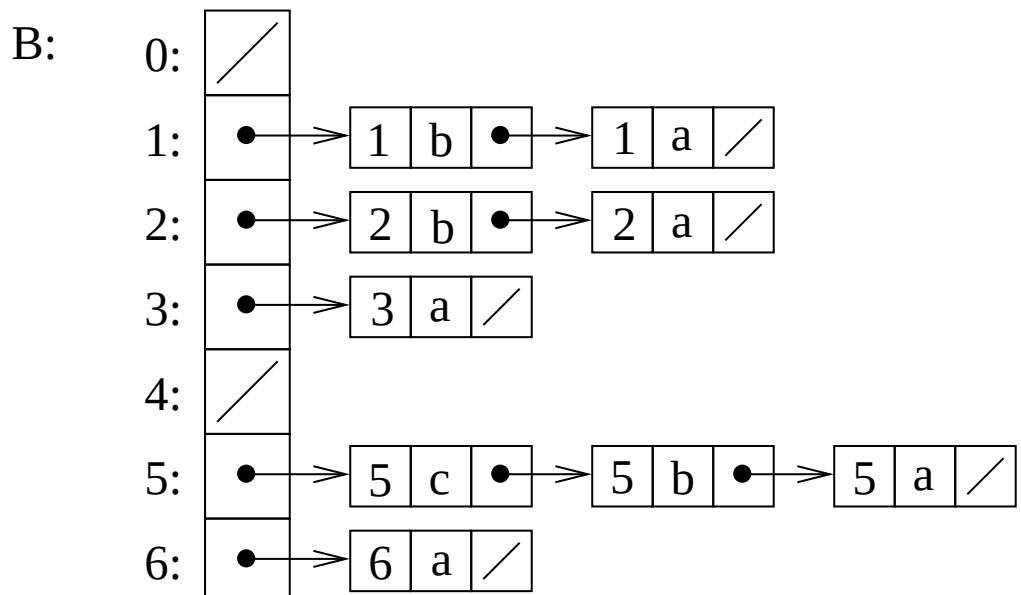


Datenstruktur: Array $\mathtt{B}[0\,..\,m-1]$ von Listen $L_0 \ldots, L_{m-1}$.

Initialisierung mit NULL-Zeigern: Zeit $\Theta(m)$.

Durchlaufe Eingabeliste, übertrage Einträge mit Schlüssel i an den Anfang der Liste L_i (angehängt in `B[`i`]`): Zeit $\Theta(n)$.

Nach Transport in das Listenarray:



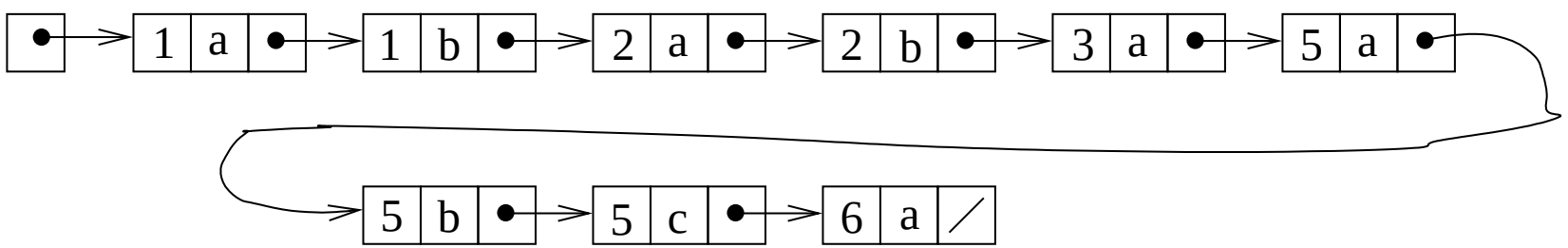
Bemerke: Identische Schlüssel in umgekehrter Reihenfolge.

Durchlaufe Array `B`$[0..m-1]$ **von hinten nach vorn**, baue neue sortierte Liste `A` (anfangs leer).

Für i: lies Elemente der Liste L_i nacheinander aus, hänge sie vorne in die Liste `A` ein.

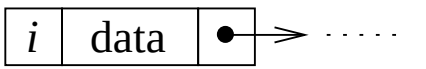
Nach Auslesen:

A:



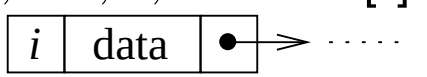
Algorithmus Bucketsort (Fachsortieren)

(1) Bearbeite Elemente der Liste A nacheinander.

Füge Listenelement 

vorn in Liste B[i] ein.

NB: Reihenfolge von Elementen mit demselben Schlüssel wird umgedreht.

(2) Für $i = m-1, \ldots, 1, 0$: lies B[i] von vorn nach hinten.
Füge Element 

vorn in Ausgabeliste ein.

NB: Reihenfolge wird erneut umgedreht $\rightarrow$ **Stabilität**.

Korrektheit: klar.

Laufzeit:

(0) Anlegen von B, Initialisieren mit NULL-Zeigern: $\Theta(m)$.

(1) Verteilen auf die Listen: $\Theta(n)$

(2) Zusammenfügen: $\Theta(m) + \Theta(n)$
(Jeder Listenanfang $B[i]$ muss inspiziert werden.)

Gesamt: $\Theta(m + n)$.

Linearzeit und linearer Platz, falls $m \leq cn$, c konstant.

Alternative:

Bei $\texttt{B}[i]$-Listen Endezeiger benutzen, in Phase (1) **hinten** anfügen.

In Phase (2): Teillisten konkatenieren.

Vorteil: in Phase (2) nur $O(m)$ Zeigerbewegungen, Zeit $O(m)$.

Nachteil: Zusätzlicher Speicherplatz $O(m)$ für Endezeiger.

Günstig insbesondere dann, wenn $m \ll n$ ist.

Satz 6.7.2 (Bucketsort/Fachsortieren)

n Objekte mit Schlüsseln aus $[m]$ (gegeben als lineare Liste) können in Zeit $\Theta(n+m)$ und mit Zusatzplatz für m Zeiger **stabil** sortiert werden.

Was tun, wenn die Anzahl m der möglichen Schlüssel begrenzt, aber deutlich größer als n ist? (Beispiel: $m = n^2$.)

(C) Radixsort – Mehrphasen-Counting-/Bucketsort

Anwendbar in verschiedenen Situationen:

1. Schlüssel sind **Folgen** $x = (x_1, \ldots, x_k) \in U^k$, $U = [m]$, $m \leq n$.

   Anordnungskriterium: **lexikographisch**, d. h.

$$(x_1, \ldots, x_k) < (y_1, \ldots, y_k)$$
$$\Leftrightarrow \exists j \in \{1, \ldots, k\} : (x_1, \ldots, x_{j-1}) = (y_1, \ldots, y_{j-1}) \wedge x_j < y_j.$$

2. Schlüssel sind **Zahlen** x in $U \subseteq \{0, 1, \ldots, m^k - 1\}$, $m \leq n$.

3. k (möglicherweise verschiedene) geordnete Mengen $(U_1, <_1), \ldots, (U_k, <_k)$

   Schlüssel sind **Folgen** $x = (x_1, \ldots, x_k) \in U_1 \times \cdots \times U_k$,
   mit lexikographischer Ordnung.

   Beispiele:
   (Kalender-)Daten: $\{1, \ldots, 31\} \times \{1, \ldots, 12\} \times \{1801, \ldots, 2090\}$.

   Besser: $\{1801, \ldots, 2090\} \times \{1, \ldots, 12\} \times \{1, \ldots, 31\}$.

   Spielkarten: $\{\diamondsuit, \heartsuit, \spadesuit, \clubsuit\} \times \{2, \ldots, 10, \text{Bube}, \text{Dame}, \text{König}, \text{Ass}\}$.

4. $\Sigma^{<\infty}$. Dabei: Σ ist „Alphabet“, d. h. nichtleere endliche Menge (z. B. ein ISO/IEC 8859-Alphabet) mit Ordnung $<$.

(4a) **Lexikographische Ordnung** auf $\Sigma^{<\infty}$
(wie im Lexikon, `ab` $<$ `abacus` $<$ `abend` $<$ `aber` $<$ `ar` $<$ `ararat`):

$$
\begin{aligned}
&(x_1, \ldots, x_k) <_{\text{lex}} (y_1, \ldots, y_\ell) \\
&\quad \Leftrightarrow \quad \big(k < \ell \wedge (x_1, \ldots, x_k) = (y_1, \ldots, y_k)\big) \quad \vee \\
&\qquad\qquad \big(\exists j \in \{1, \ldots, \min\{k, \ell\}\} : (x_1, \ldots, x_{j-1}) = (y_1, \ldots, y_{j-1}) \wedge x_j < y_j\big).
\end{aligned}
$$

(4b) **Kanonische Ordnung** auf $\Sigma^{<\infty}$
(kürzere Strings zuerst, `ab` $<$ `ar` $<$ `aber` $<$ `abend` $<$ `abacus` $<$ `ararat`):

$$
\begin{aligned}
&(x_1, \ldots, x_k) <_{\text{kan}} (y_1, \ldots, y_\ell) \\
&\quad \Leftrightarrow \quad k < \ell \quad \vee \\
&\qquad\qquad \big(k = \ell \wedge \exists j \in \{1, \ldots, k\} : (x_1, \ldots, x_{j-1}) = (y_1, \ldots, y_{j-1}) \wedge x_j < y_j\big).
\end{aligned}
$$

Zunächst: Situation 1, Schlüssel $x = (x_1, \ldots, x_k) \in U^k$, $U = [m]$.

Idee: **sortiere k-mal mittels Counting-/Bucketsort**
und zwar gemäß Komponente k, dann $k-1$, . . . , dann 1
(die „**am wenigsten signifikante**“ Komponente **zuerst**).

Eingabe:
Array/Liste `A`: Einträge mit Schlüsseln $(x_1, \ldots, x_k)$ aus $[m]^k$.

Methode:
für $l = k, k-1, \ldots, 1$ tue:
sortiere `A` mittels **Counting-** bzw. **Bucketsort** für $U = [m]$
mit Komponente x_l aus $(x_1, \ldots, x_k)$ als Sortierschlüssel.

Zeit insgesamt: $\Theta(k \cdot (n+m))$

Zusatzplatz: $\Theta(m+n)$ bzw. $\Theta(m)$.

Beispiel:

$U = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}\}$

Eingabe `A`: | BCC | | ABC | | BAC | | CAD | | ABA |

Nach Runde $l = 3$: | ABA | | BCC | | ABC | | BAC | | CAD |

Nach Runde $l = 2$: | BAC | | CAD | | ABA | | ABC | | BCC |

Nach Runde $l = 1$: | ABA | | ABC | | BAC | | BCC | | CAD |

Beobachte: Nach Runde für l sind die Objekte gemäß $x_l, \ldots, x_k$ sortiert.

Satz 6.7.3

Der skizzierte Algorithmus **Radixsort** sortiert n Objekte mit Schlüsseln aus $[m]^k$ in Zeit $\Theta(k(n+m))$ und Platz $\Theta(m)$ (bzw. $\Theta(n+m)$ für Arrays). Das Verfahren ist stabil.

Beweis: **(Nicht prüfungsrelevant.)** Nur Korrektheit. Man zeigt durch Induktion über $l = k, k-1, \ldots, 1$ die folgende Schleifeninvariante:

Nach Schleifendurchlauf für l ist das Array/die Liste `A`
gemäß den Teilschlüsseln $(x_l, x_{l+1}, \ldots, x_k) \in [m]^{k-l+1}$ lexikographisch sortiert.

Für Induktionsschritt $l+1 \to l$ benutzt man, dass Counting-/Bucketsort **stabil** ist, und die schon hergestellte Ordnung bezüglich der weniger signifikanten Komponenten nicht zerstört.

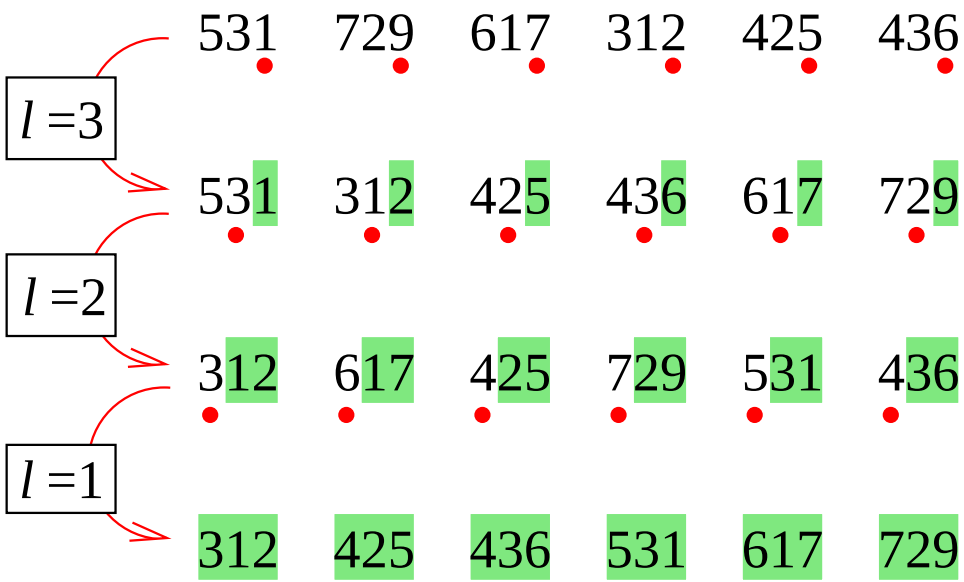
Das Verfahren für $U_1 \times \cdots \times U_k$ (Situation 3) ist im Wesentlichen dasselbe, nur benötigt man verschiedene B-Arrays (oder Arrayabschnitte) in den einzelnen Durchgängen.

Zeit: $O(kn + |U_1| + \cdots + |U_k|)$; Platz: $O(n + \max_{1 \leq i \leq k} |U_i|)$.

Situation 2: Schlüssel sind Zahlen in $\{0, 1, \ldots, m^k - 1\}$:

Repräsentiere Zahlen in m-ärer Darstellung (mit k Ziffern); wende Radixsort an.

Beispiel: $m = 10$ (Dezimaldarstellung)



So kleine m sind **nicht typisch**.
Sinnvoll: $m \approx n$, m Zweierpotenz, z. B. $m = 2^8$ oder 2^{16}.

Dann ist eine „Ziffer“ x_l ein Segment der Binärdarstellung von x, die leicht aus x zu extrahieren ist.

Satz 6.7.4 **Radixsort** für Zahlen sortiert n Objekte mit Schlüsseln aus $[m^k]$ in Zeit $\Theta(k(n+m))$ und Platz $\Theta(m)$ (bzw. $\Theta(n+m)$ für Arrays). Das Verfahren ist stabil.

SS 2021

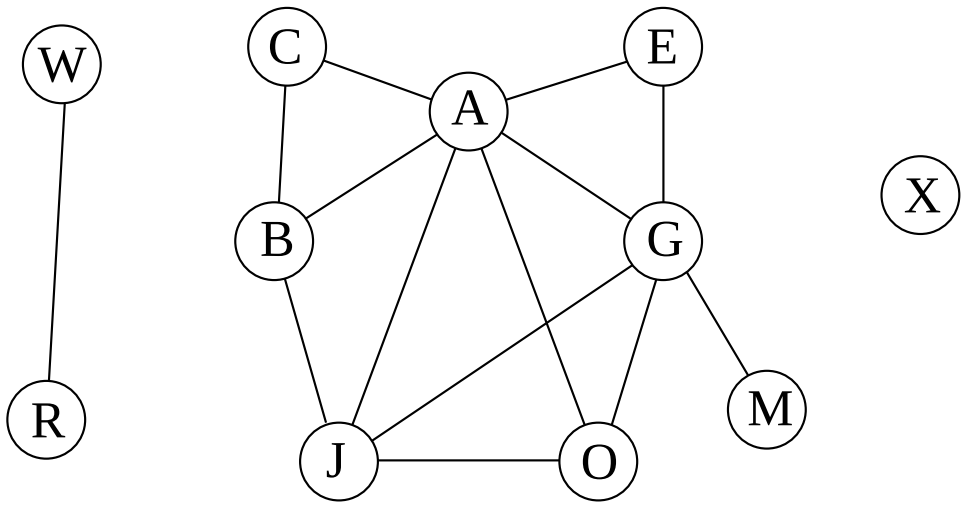
Algorithmen und Datenstrukturen

7. Kapitel

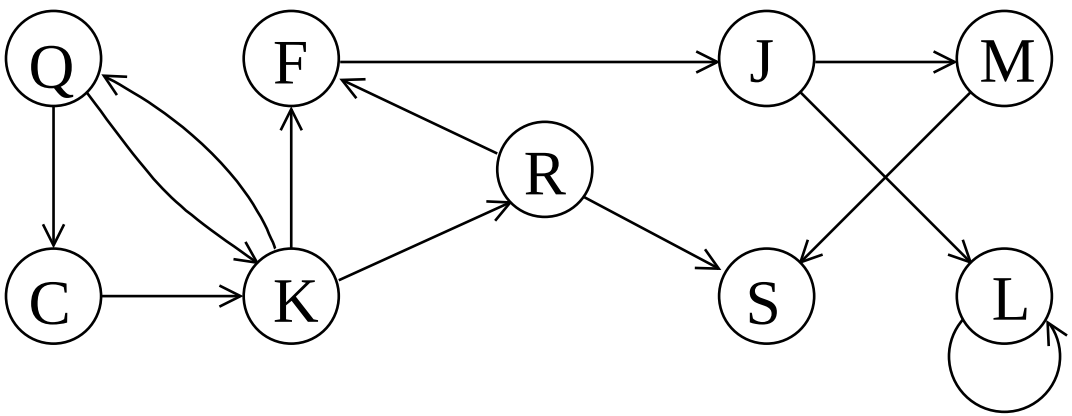
Graphen, Digraphen und Breitensuche

Martin Dietzfelbinger

Juni 2021



(Ungerichteter) Graph



Gerichteter Graph (Digraph)

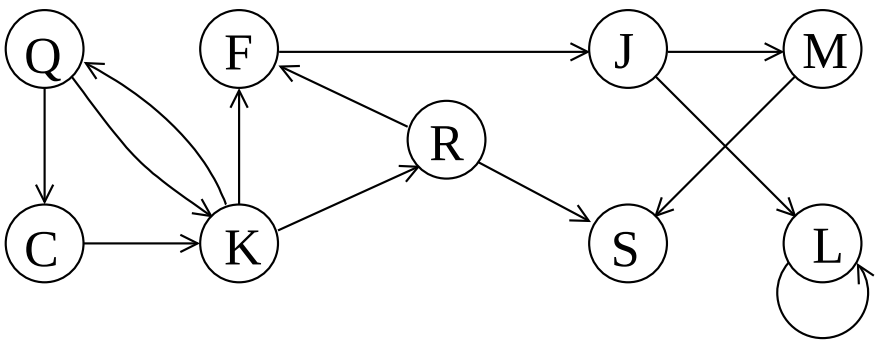
Graphen und **Digraphen** sind eine in der Informatik und zur Modellierung von Anwendungssituationen allgegenwärtige (Daten-)Struktur.

Sie modellieren:

- Orte und Verbindungsstraßen, Kreuzungen/Einmündungen und Straßen
- Gatter und Leitungen auf einem Chip
- Systemkomponenten und Verbindungen
- Zustände eines Systems und Übergänge
- Flussdiagramme für Programmentwurf
- Datenflussdiagramme für Programmanalyse
- Vorgänge mit Unverträglichkeitsbeziehungen
- Stationen eines Transportsystems mit Kapazitäten der Verbindungen
- Soziale Beziehungen
- Das Internet
- u. v. a. m.

In dieser Vorlesung nur: Algorithmen zum Umgang mit den abstrakten Strukturen.
(Modellierung für Anwendungen und Beispiele nur am Rande.)

7.1 Grundbegriffe



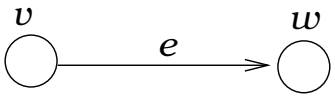
Definition 7.1.1

Ein **gerichteter Graph** oder **Digraph** G (wegen „*directed graph*“)

ist ein Paar (V, E), wobei V eine endliche Menge und
E eine Teilmenge von $V \times V = \{(v, w) \mid v, w \in V\}$ ist.

Die Elemente von V heißen **Knoten**[1], die Elemente von E heißen **Kanten**.

[1] In der Mathematik oft auch: „Ecken“. Wir benutzen diese Bezeichnung nicht.



Knoten zeichnet man als Kreise („*node*“ oder „*vertex*“),

Kanten als Pfeile („*edge*“ oder „*arc*“).

Ist $e = (v, w)$ eine Kante (von G), so heißt

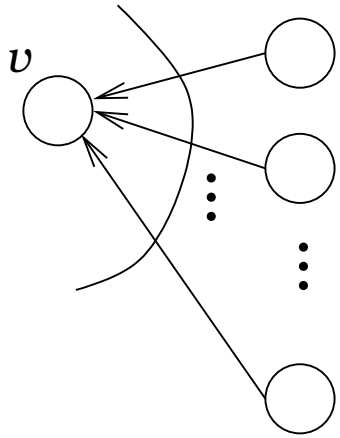
v der **Anfangsknoten** und w der **Endknoten** von e,

v bzw. w heißen **inzident** zu e (v, w „**liegen auf**“ e),

w heißt auch (ein) **Nachfolger** von v,

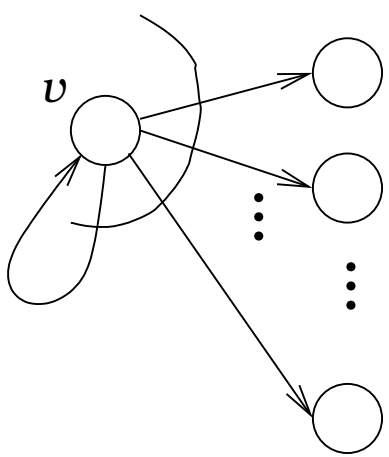
v heißt (ein) **Vorgänger** von w.

Eine Kante (v, v) heißt **Schleife** („*loop*“).



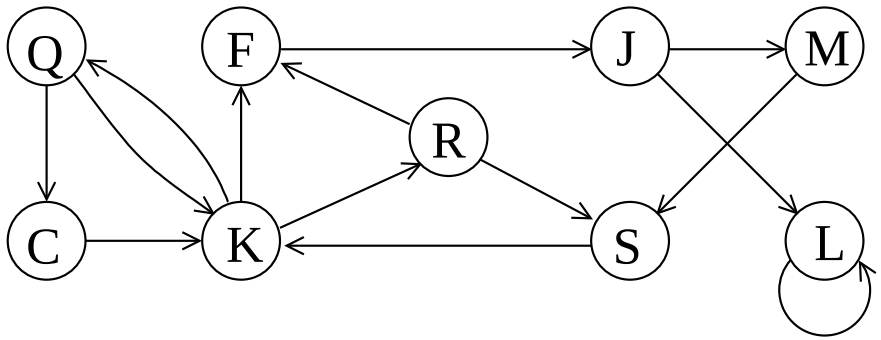
Der **Eingangsgrad** („*indegree*“) eines Knotens v ist die Anzahl der Kanten, die in v hineinführen:

$$\textbf{indeg}(v) = |\{u \in V \mid (u, v) \in E\}|.$$



Der **Ausgangsgrad** („*outdegree*“) eines Knotens v ist die Anzahl der Kanten, die aus v herausführen:

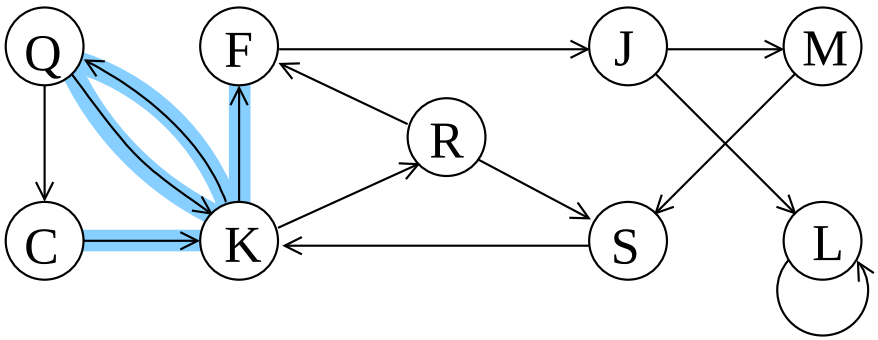
$$\textbf{outdeg}(v) = |\{w \in V \mid (v, w) \in E\}|.$$



Lemma 7.1.1

$$\sum_{v \in V} \text{indeg}(v) = \sum_{v \in V} \text{outdeg}(v) = |E|.$$

Beweis: In beiden Summen wird jede Kante genau einmal gezählt. □



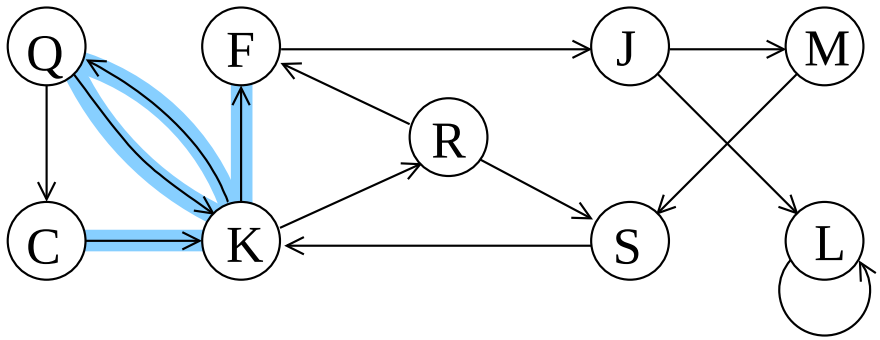
Definition 7.1.2

Sei $G = (V, E)$ ein Digraph.

(a) Ein **Kantenzug** in G ist eine Folge $p = (v_0, v_1, \ldots, v_k)$ von Knoten, wobei $(v_{i-1}, v_i) \in E$ für $1 \leq i \leq k$.

Gleichbedeutend: Eine Folge $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ von Kanten.

Beispiele für Kantenzüge im Bild: $(\mathrm{C}, \mathrm{K}, \mathrm{R}, \mathrm{S})$, **$(\mathrm{C}, \mathrm{K}, \mathrm{Q}, \mathrm{K}, \mathrm{F})$**, $(\mathrm{F}, \mathrm{J}, \mathrm{L}, \mathrm{L}, \mathrm{L}, \mathrm{L})$.



(b) Die **Länge** von $(v_0, v_1, \ldots, v_k)$ ist k (die **Kantenzahl**).

(Bsp.: $\mathbf{(C, K, Q, K, F)}$ hat Länge 4.)

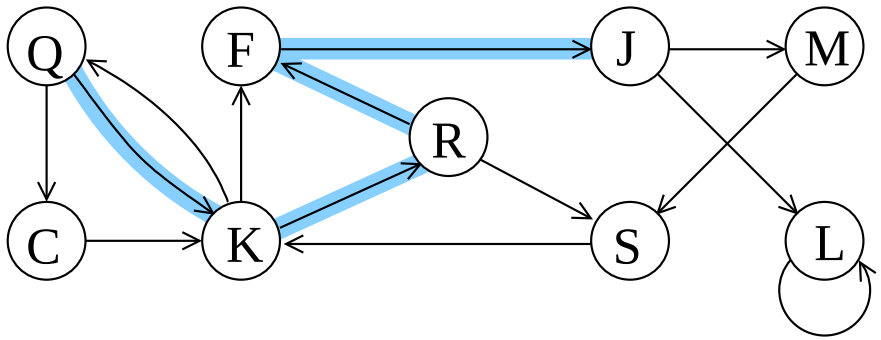
Kantenzug (v) hat keine Kante, also Länge 0.

(c) Wir schreiben $\boldsymbol{v \rightsquigarrow_G w}$ oder $\boldsymbol{v \rightsquigarrow w}$, wenn in G ein Kantenzug $(v_0, v_1, \ldots, v_k)$ mit $v = v_0$ und $w = v_k$ („ein Kantenzug von v nach w“) existiert.

Beispiel: $\mathrm{K} \rightsquigarrow \mathrm{S}$, $\mathrm{M} \rightsquigarrow \mathrm{L}$, $\mathrm{M} \rightsquigarrow \mathrm{M}$, aber **nicht** $\mathrm{L} \rightsquigarrow \mathrm{M}$.

Beobachtung Die Relation $\rightsquigarrow$ ist **reflexiv** und **transitiv**.

((v) ist Kantenzug; Kantenzüge kann man **aneinanderhängen**.)



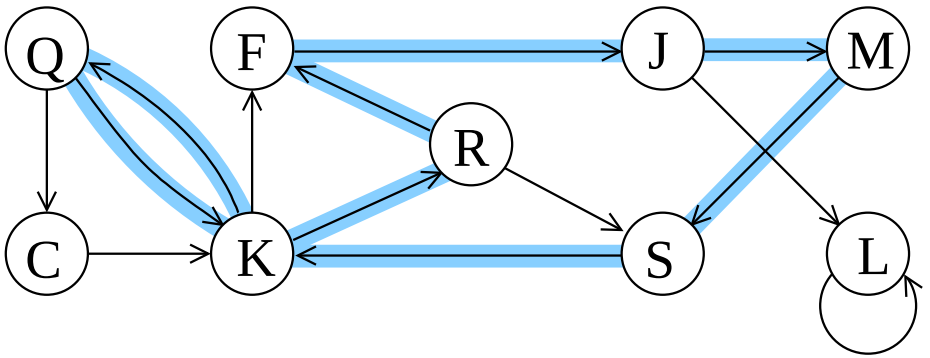
Definition 7.1.3

Ein Kantenzug $(v_0, v_1, \ldots, v_k)$ in einem Digraphen G heißt **Weg** oder **Pfad** (in dieser Vorlesung *synonym*), wenn $v_0, v_1, \ldots, v_k$ verschieden sind.

Beispiel: $\mathbf{(Q, K, R, F, J)}$.

Beobachtung Wenn $v \rightsquigarrow w$, dann existiert ein Weg $(v_0, v_1, \ldots, v_l)$ mit $v = v_0$ und $w = v_l$.

(Beweisidee: Wenn Kantenzug $(v_0, v_1, \ldots, v_k)$ Knoten u doppelt enthält, ersetze Teilfolge $\ldots, u, \ldots, u, \ldots$ durch $\ldots, u, \ldots$. Falls nötig, wiederhole diesen Schritt.)



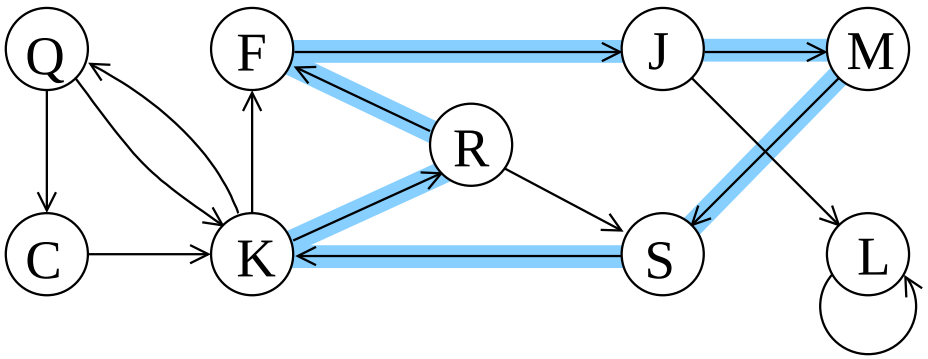
Definition 7.1.4

(a) Ein Kantenzug $(v_0, v_1, \ldots, v_k)$ in einem Digraphen G heißt ein **Kreis** oder **Zyklus** („*cycle*“), wenn $k \geq 1$ und $v_0 = v_k$. Die Länge des Kreises ist k.

Anm.: Eine Schleife $(v, v) \in E$ ist also ein Kreis der Länge 1.

Beispiel: $(\mathrm{K}, \mathrm{Q}, \mathrm{C}, \mathrm{K})$, (L, L), $(\mathrm{L}, \mathrm{L}, \mathrm{L})$, $(\mathbf{Q}, \mathbf{K}, \mathbf{R}, \mathbf{F}, \mathbf{J}, \mathbf{M}, \mathbf{S}, \mathbf{K}, \mathbf{Q})$ sind Kreise.

Bemerkung: Kreise, die durch einen zyklischen Shift auseinander hervorgehen, wie $(\mathrm{K}, \mathrm{Q}, \mathrm{C}, \mathrm{K})$ und $(\mathrm{Q}, \mathrm{C}, \mathrm{K}, \mathrm{Q})$, werden als gleich betrachtet.



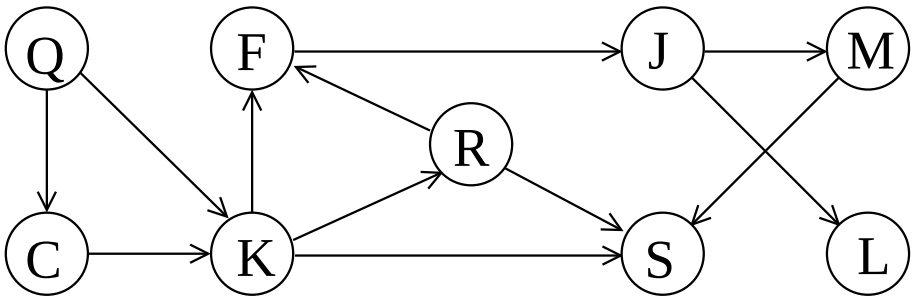
(b) Ein Kreis $(v_0, v_1, \ldots, v_{k-1}, v_0)$ heißt **einfach**, wenn $v_0, \ldots, v_{k-1}$ verschieden sind.

Beispiel: **(J, M, S, K, R, F, J)**, $(\mathrm{K}, \mathrm{C}, \mathrm{Q}, \mathrm{K})$, (L, L) sind einfache Kreise.

Beobachtung:

Wenn ein Digraph G einen Kreis enthält, dann enthält G auch einen einfachen Kreis.

(Beweisidee: Herausstreichen von Teilwegen $v, \ldots, v$, genau wie bei Wegen, Folie 10.)



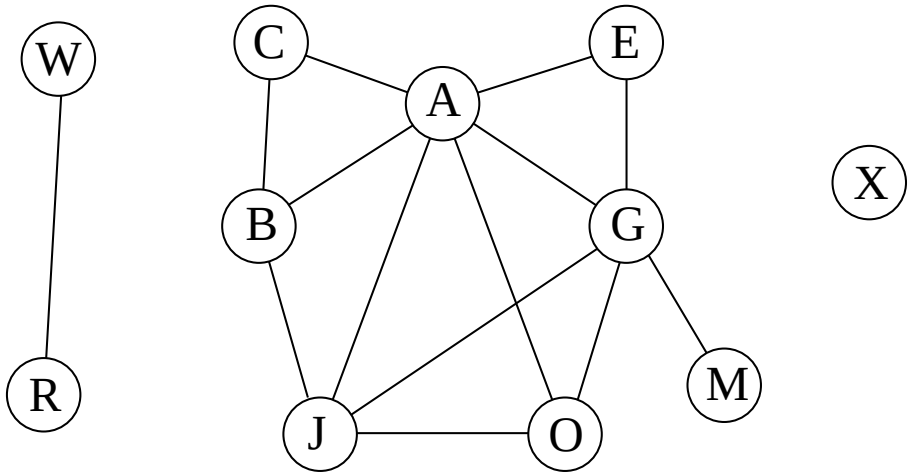
Definition 7.1.5

Ein Digraph G heißt **kreisfrei** oder **azyklisch**, wenn es in G keinen Kreis gibt, sonst heißt G **zyklisch**.

Diese **azyklischen gerichteten Graphen** bilden eine wichtige Teilklasse der gerichteten Graphen.

(Englisch, mit verdrehter Wortreihenfolge: **„Directed Acyclic Graphs“**, daher **DAG**s oder **dag**s.)

Offensichtliche algorithmische Grundaufgabe: Testen, ob G ein DAG ist. (Später.)

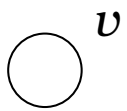


Definition 7.1.6

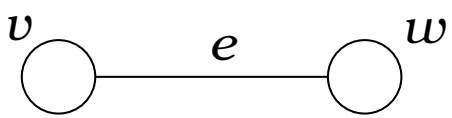
Ein **ungerichteter Graph**, oft auch: ein **Graph G** ist ein Paar (V, E), wobei
V eine endliche Menge ist und
E eine Teilmenge von $[V]_2 := \binom{V}{2} := \{\{v, w\} \mid v, w \in V, v \neq w\}$ ist.

Schreibweise: (v, w) für $\{v, w\}$.

Im Bild: $V = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{E}, \mathrm{G}, \mathrm{J}, \mathrm{M}, \mathrm{O}, \mathrm{R}, \mathrm{W}, \mathrm{X}\}$,
$E = \{(\mathrm{A}, \mathrm{B}), (\mathrm{A}, \mathrm{C}), (\mathrm{C}, \mathrm{B}), (\mathrm{A}, \mathrm{E}), (\mathrm{A}, \mathrm{G}), (\mathrm{G}, \mathrm{E}), (\mathrm{A}, \mathrm{J}), (\mathrm{A}, \mathrm{O}), (\mathrm{B}, \mathrm{J}),$
$(\mathrm{J}, \mathrm{O}), (\mathrm{J}, \mathrm{G}), (\mathrm{O}, \mathrm{G}), (\mathrm{G}, \mathrm{M}), (\mathrm{R}, \mathrm{W})\}$.



Die Elemente von V heißen **Knoten**[1] („*node*“ oder „*vertex*“).
Knoten zeichnet man als Kreise.



Die Elemente von E heißen **Kanten**.
Kanten zeichnet man als
(ungerichtete) Linien („*edge*“).

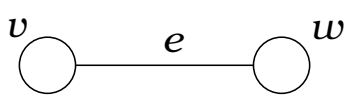
Konvention: Kante $\{u, v\}$ $(= \{v, u\})$ wird als $\boldsymbol{(u, v)}$ geschrieben.

(**Nur!**) Bei Kanten von (ungerichteten) Graphen gilt also $(u, v) = (v, u)$.

[1] In der Mathematik oft auch: „Ecken“.

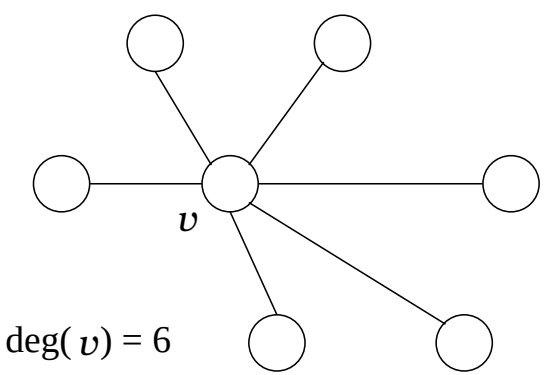
Definition 7.1.7

Sei $G = (V, E)$ ein ungerichteter Graph.



Ist $e = (v, w)$ eine Kante (von G), so heißt v bzw. w **inzident** zu e, und v und w heißen **adjazent** oder **benachbart**; v heißt **Nachbar** von w und umgekehrt.

„Schleifen“, d.h. „Kanten“ (v, v), sind bei (ungerichteten) Graphen normalerweise nicht zugelassen.



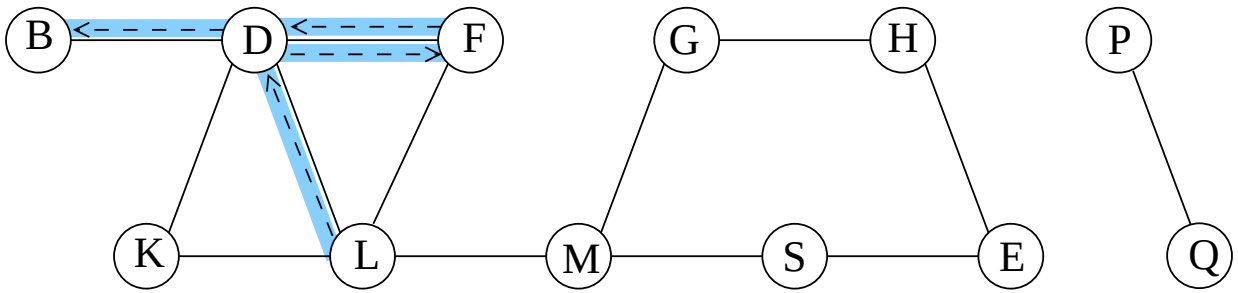
Der **Grad** (Valenz, „*degree*“) eines Knotens v ist $\mathbf{deg}(v) := |\{u \in V \mid (u, v) \in E\}|$.
Knoten v mit Grad 0 heißen **isoliert**. (Keine Nachbarn.)

Lemma 7.1.8 („Handshaking-Lemma“)

Wenn $G = (V, E)$ ein Graph ist, dann gilt:

$$\sum_{v \in V} \deg(v) = 2|E|.$$

Beweis: Jede Kante (u, v) in E trägt 1 zu $\deg(u)$ und 1 zu $\deg(v)$ bei. □



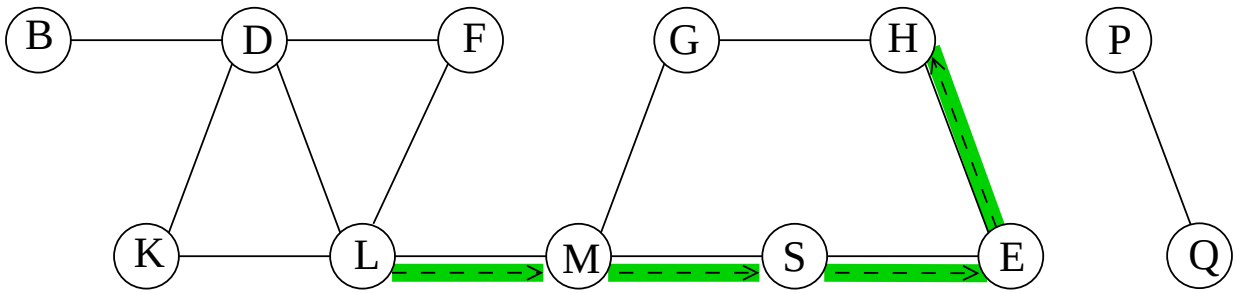
Definition 7.1.9

Sei $G = (V, E)$ ein Graph.

(a) Ein **Kantenzug** in G ist eine Folge $(v_0, v_1, \ldots, v_k)$ von Knoten,
d. h. Elementen von V, wobei $(v_{i-1}, v_i) \in E$ für $1 \leq i \leq k$.

(b) Die **Länge** eines Kantenzugs $(v_0, v_1, \ldots, v_k)$ ist die **Kantenzahl k**.
($k = 0$ ist erlaubt.)

Kantenzüge im Beispielgraphen: $(\mathrm{L}, \mathrm{D}, \mathrm{F}, \mathrm{D}, \mathrm{B})$ (Länge 4), $(\mathrm{L}, \mathrm{F}, \mathrm{D}, \mathrm{L}, \mathrm{M}, \mathrm{S})$, (Länge 5).



(c) Ein Kantenzug $(v_0, v_1, \ldots, v_k)$ in einem Graphen G heißt **Weg** oder **Pfad**, wenn $v_0, v_1, \ldots, v_k$ verschieden sind.

Weg im Beispiel: $(\mathbf{L}, \mathbf{M}, \mathbf{S}, \mathbf{E}, \mathbf{H})$, Länge 4.

Kantenzüge, aber keine Wege: Beispiele auf vorheriger Folie.

Lemma 7.1.10

Sei G ein Graph. Wenn es einen Kantenzug $(v_0, \ldots, v_k)$ mit $v_0 = v$ und $v_k = w$ („von v nach w“) gibt, dann gibt es auch ein Weg von v nach w.

(*Beweis* wie bei Digraphen, Folie 10.)

Definition 7.1.11

Sei G ein Graph. Wenn $v, w \in V$ durch einen Kantenzug $p = (v_0, v_1, \ldots, v_k)$ mit $v_0 = v$ und $v_k = w$ verbunden sind, schreiben wir $\boldsymbol{v \sim_G w}$ oder $\boldsymbol{v \sim w}$.

Lemma 7.1.12

Die zweistellige Relation $\sim_G$ auf V ist eine **Äquivalenzrelation**, d. h. sie ist

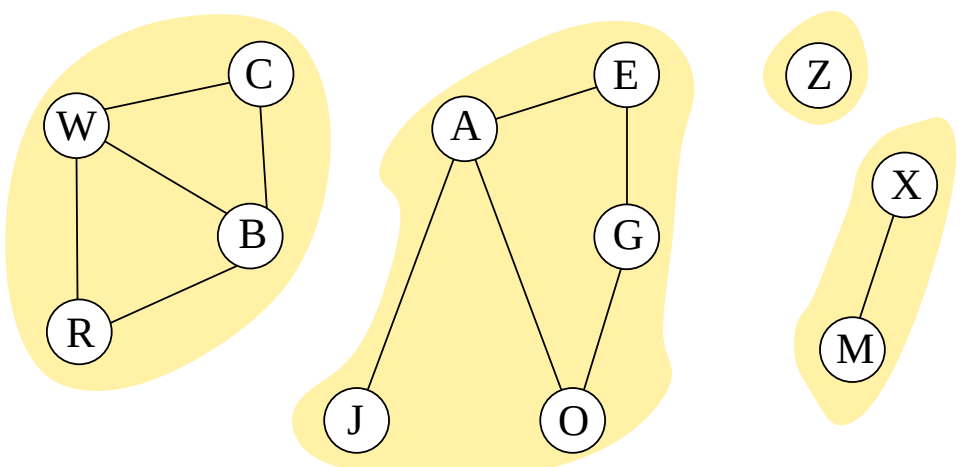
reflexiv: $v \sim_G v$,
symmetrisch: $v \sim_G w \Rightarrow w \sim_G v$,
transitiv: $u \sim_G v \wedge v \sim_G w \Rightarrow u \sim_G w$.

Beweis:
Reflexivität: (v) ist ein Weg von v nach v, Länge 0;
Symmetrie: durchlaufe Weg von v nach w in umgekehrter Richtung;
Transitivität: Konkatenieren der Wege von u nach v und von v nach w liefert zunächst Kantenzug von u nach w, dann einen Weg durch Wegstreichen von Teilstücken. □

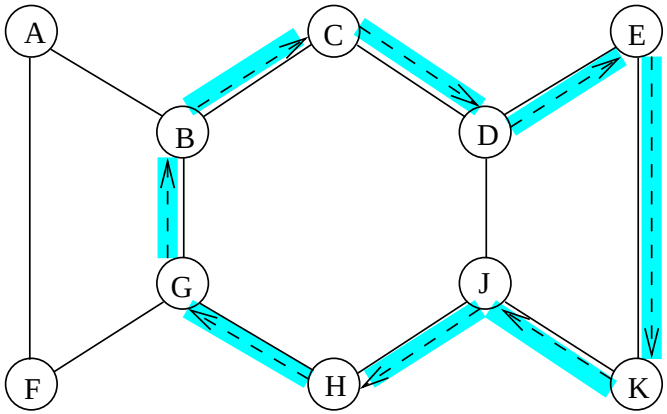
Definition 7.1.13

(a) Die Äquivalenzrelation $\sim_G$ zerlegt die Knotenmenge V in Äquivalenzklassen, die **Zusammenhangskomponenten** von G.

Beispiel: Graph mit vier Zsh.-Komp. $\{\mathrm{B}, \mathrm{C}, \mathrm{R}, \mathrm{W}\}$, $\{\mathrm{A}, \mathrm{G}, \mathrm{E}, \mathrm{J}, \mathrm{O}\}$, $\{\mathrm{M}, \mathrm{X}\}$, $\{\mathrm{Z}\}$:



(b) Ein Graph G, der nur eine Zusammenhangskomponente hat (d. h., in dem $u \sim_G v$ für alle $u, v \in V$ gilt), heißt **zusammenhängend**.



Einfache Kreise: (B,C,D,E,K,J,H,G,B) und (K,J,D,E,K)

Definition 7.1.14

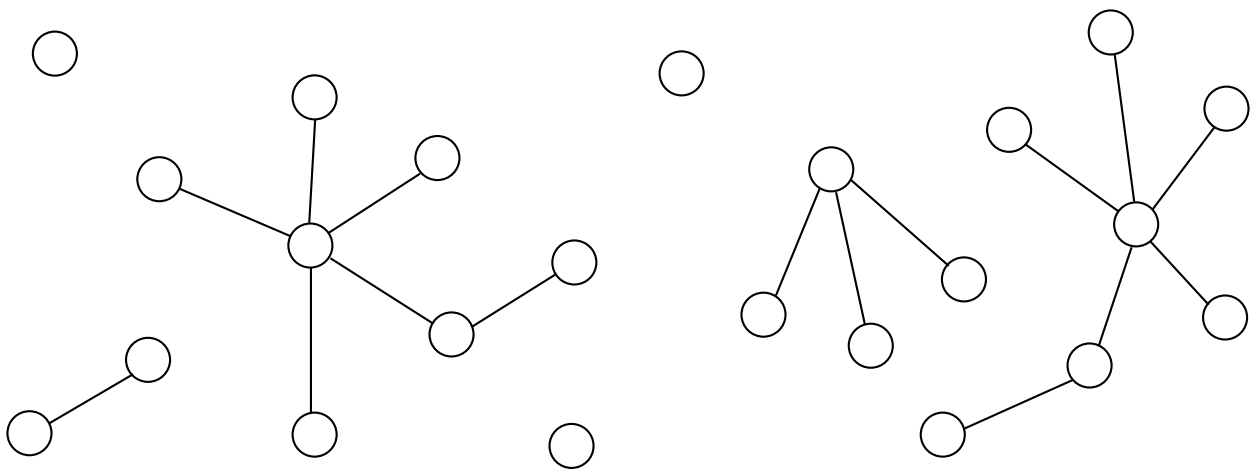
Ein **(einfacher) Kreis** in einem Graphen G ist ein Kantenzug $(v_0, v_1, \ldots, v_k)$ mit $k \geq 3$ und $v_0 = v_k$ und $v_0, v_1, \ldots, v_{k-1}$ verschieden. (Die Länge des Kreises ist k.)

Der Anfangspunkt eines Kreises ist unerheblich: $(\mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{K}, \mathrm{J}, \mathrm{H}, \mathrm{G}, \mathrm{B})$ und $(\mathrm{K}, \mathrm{J}, \mathrm{H}, \mathrm{G}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{K})$ werden als „derselbe Kreis“ angesehen.
Oft auch: Die Durchlaufrichtung ist unerheblich, d. h. $(\mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{K}, \mathrm{J}, \mathrm{H}, \mathrm{G}, \mathrm{B})$ und $(\mathrm{B}, \mathrm{G}, \mathrm{H}, \mathrm{J}, \mathrm{K}, \mathrm{E}, \mathrm{D}, \mathrm{C}, \mathrm{B})$ gelten als „derselbe Kreis“.

Definition 7.1.15

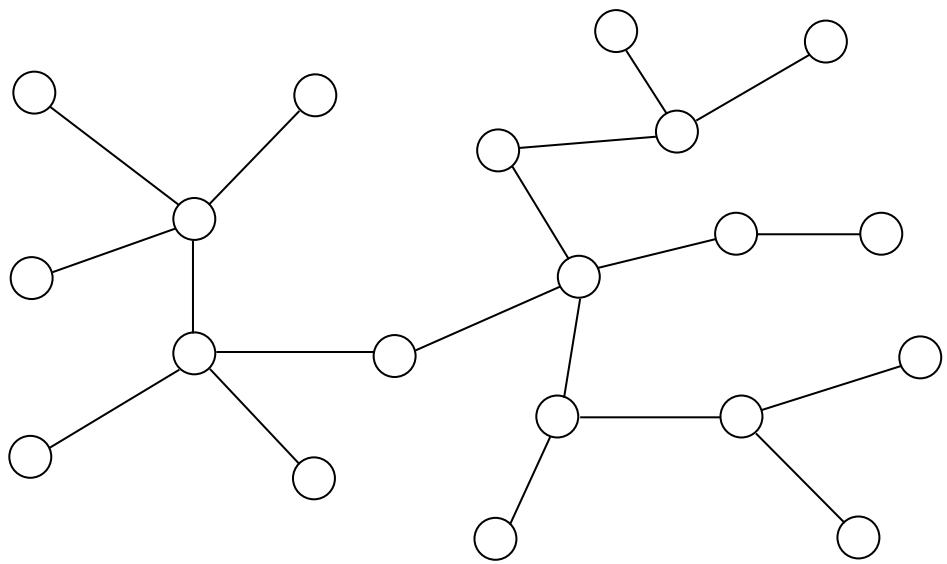
(a) Ein Graph $G = (V, E)$ heißt **kreisfrei**, wenn er **keinen Kreis** besitzt.

Beispiel: Die Abbildung zeigt einen kreisfreien Graphen.



(b) Ein Graph heißt ein **freier Baum** (oder nur **Baum**),
wenn er zusammenhängend und kreisfrei ist.

Beispiel: Ein (freier) Baum mit 20 Knoten und 19 Kanten:



Anmerkung: Die Zusammenhangskomponenten eines kreisfreien Graphen sind freie Bäume, wie auf der vorigen Folie. Kreisfreie Graphen heißen daher auch **(freie) Wälder**.

Lemma 7.1.16 (Fundamentallemma über Bäume)

Wenn $G = (V, E)$ ein Graph mit $n = |V|$ Knoten und $m = |E|$ Kanten ist, dann sind die folgenden Aussagen **äquivalent**:

(a) G ist ein Baum.

(b) G ist kreisfrei und $m \geq n - 1$.

(c) G ist zusammenhängend und $m \leq n - 1$.

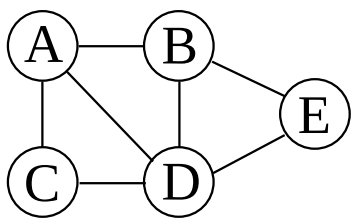
(d) Zu je zwei Knoten u und v gibt es genau einen (einfachen) Weg von u nach v.

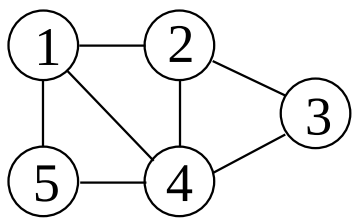
(e) G ist kreisfrei, aber das Hinzufügen einer beliebigen Kante zu G erzeugt einen Kreis. (G ist „maximal kreisfrei“.)

(f) G ist zusammenhängend, aber das Entfernen einer beliebigen Kante aus E erzeugt einen nicht zusammenhängenden Graphen.

Beweis: Für Interessierte in einer eigenen Notiz (s. Webseite und Moodle).

7.2 Datenstrukturen für Digraphen und Graphen





nodes:

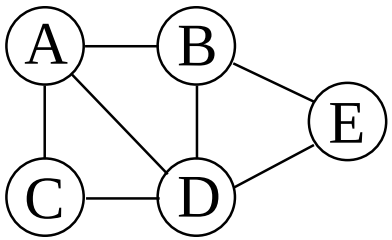
1:	A
2:	B
3:	E
4:	D
5:	C

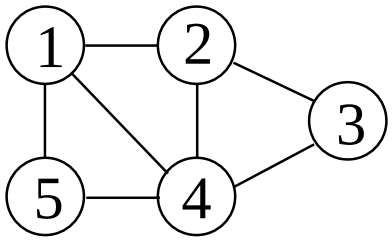
Sei $G = (V, E)$ Graph oder Digraph.

V kann irgendeine endliche Menge sein. (Hier: $\{\mathrm{A},\mathrm{B},\mathrm{C},\mathrm{D},\mathrm{E}\}$.)

Ordne die $n = |V|$ Knoten beliebig an, z. B. als $V = \{v_1, \ldots, v_n\}$
und stelle sie durch ein Array dar:

`nodes: array [1..`n`] of nodetype`





nodes:

1:	A
2:	B
3:	E
4:	D
5:	C

Der Knotenname v_i und andere Attribute der Knoten („Markierungen“, „Beschriftungen“) werden als Komponenten in `nodes` gehalten.

Wir nehmen meistens an, dass die Knoten mit $1, 2, \ldots, n$ durchnummeriert sind und ein `nodes`-Array vorhanden ist.

In dieser Darstellung ist (i, j) eine Kante genau dann, wenn im Originalgraphen G das Paar (v_i, v_j) eine Kante ist.

Definition 7.2.1

Ist $G = (V, E)$ Graph oder Digraph mit $V = \{1, \ldots, n\}$, dann ist die **Adjazenzmatrix** von G die $n \times n$-Matrix

$$A = A_G = \begin{pmatrix} a_{11} & \ldots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \ldots & a_{nn} \end{pmatrix},$$

mit

$$a_{ij} = [(i,j) \in E] := \begin{cases} 1, & \text{falls } (i,j) \in E \\ 0, & \text{falls } (i,j) \notin E. \end{cases}$$

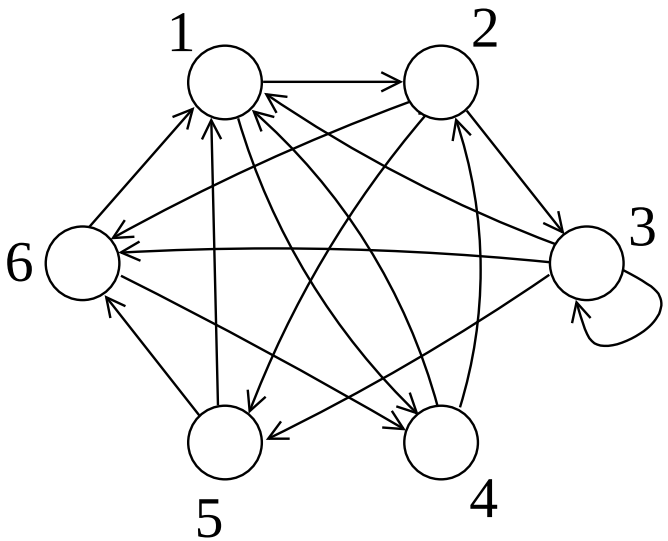
In den meisten Programmiersprachen:
Matrix wird als 2-dimensionales Array `A[1 . . n, 1 . . n]` realisiert, mit Einträgen aus $\{0, 1\}$.

Klar: Zugriff (Lesen, Schreiben) auf a_{ij} in Zeit $O(1)$.

Beispiel-Digraph:

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

$\widehat{=}$

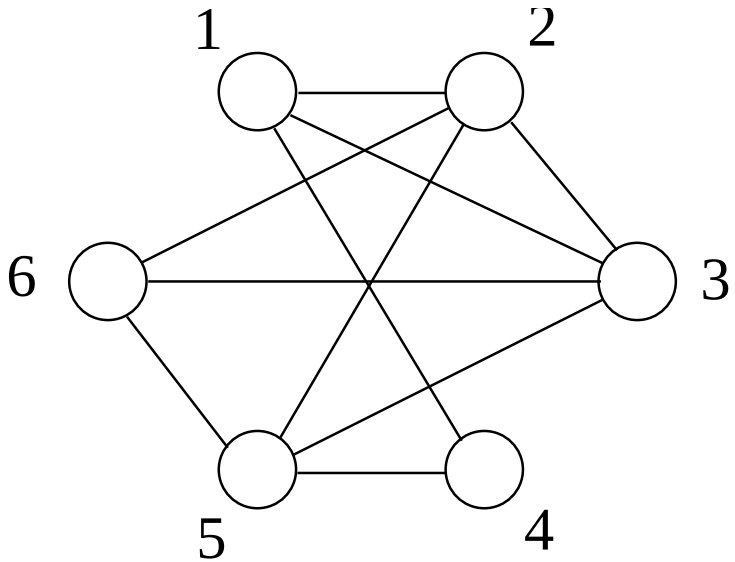


Die Anzahl der 1en in Zeile i ist outdeg(i);
die Anzahl der 1en in Spalte j ist indeg(j).

Beispiel-Graph:

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

$\widehat{=}$



Die Adjazenzmatrix eines **Graphen** ist **symmetrisch**: $\mathtt{A}[i,j] = \mathtt{A}[j,i]$.

Die Anzahl der 1en in Zeile/Spalte i ist $\deg(i)$.

Beobachtungen

Verwendet man zur Darstellung eines Graphen bzw. Digraphen mit n Knoten eine **Adjazenzmatrix**, so gilt:

(a) Der Speicherplatzbedarf ist $\Theta(n^2)$ [Bits];

(b) a_{ij} kann in Zeit $O(1)$ ermittelt oder geändert werden;

(c) die Ermittlung **aller** Nachfolger, Vorgänger oder Nachbarn eines Knotens erfordert Zeit $\Theta(n)$. (Zeilen/Spaltendurchlauf; Reihenfolge: $1, \ldots, n$).

Der Speicherplatzbedarf ist im Vergleich zur vorhandenen Information relativ hoch, wenn $|E| \ll n^2$ („dünn besetzte“ Graphen).

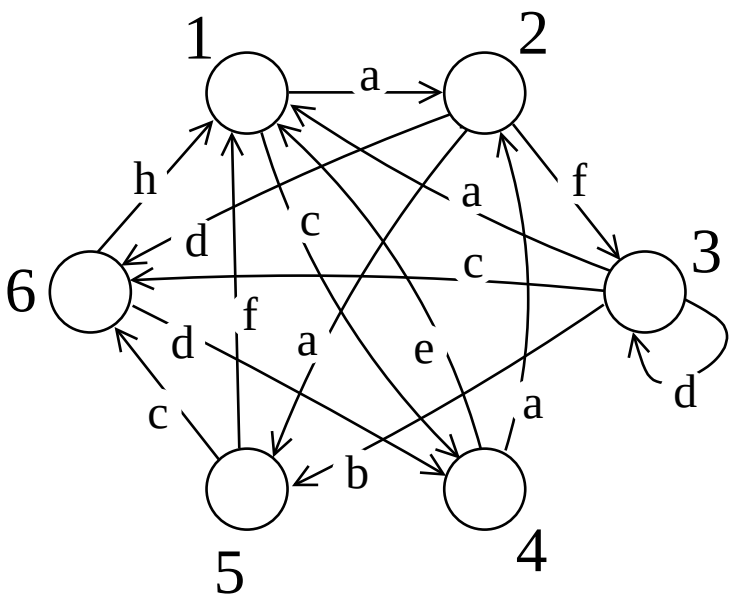
Reduzierung des Speicherplatzbedarfs auf etwa n^2/w Maschinenwörter:
Speichere w Bits der Matrix in Maschinenwörtern (z. B. „long long int“) der Länge w, als *Bitvektor*.

Erweiterung: Auch Kanten können mit Objekten aus einer Menge M beschriftet („markiert“) sein (Längen, Gewichte, Kosten, Kapazitäten). Dann benutzt man ein Array mit Einträgen aus $M \cup \{-\}$ oder $M \cup \{\infty\}$ („$-$“ bzw. „∞“ bedeutet „nicht vorhanden“)

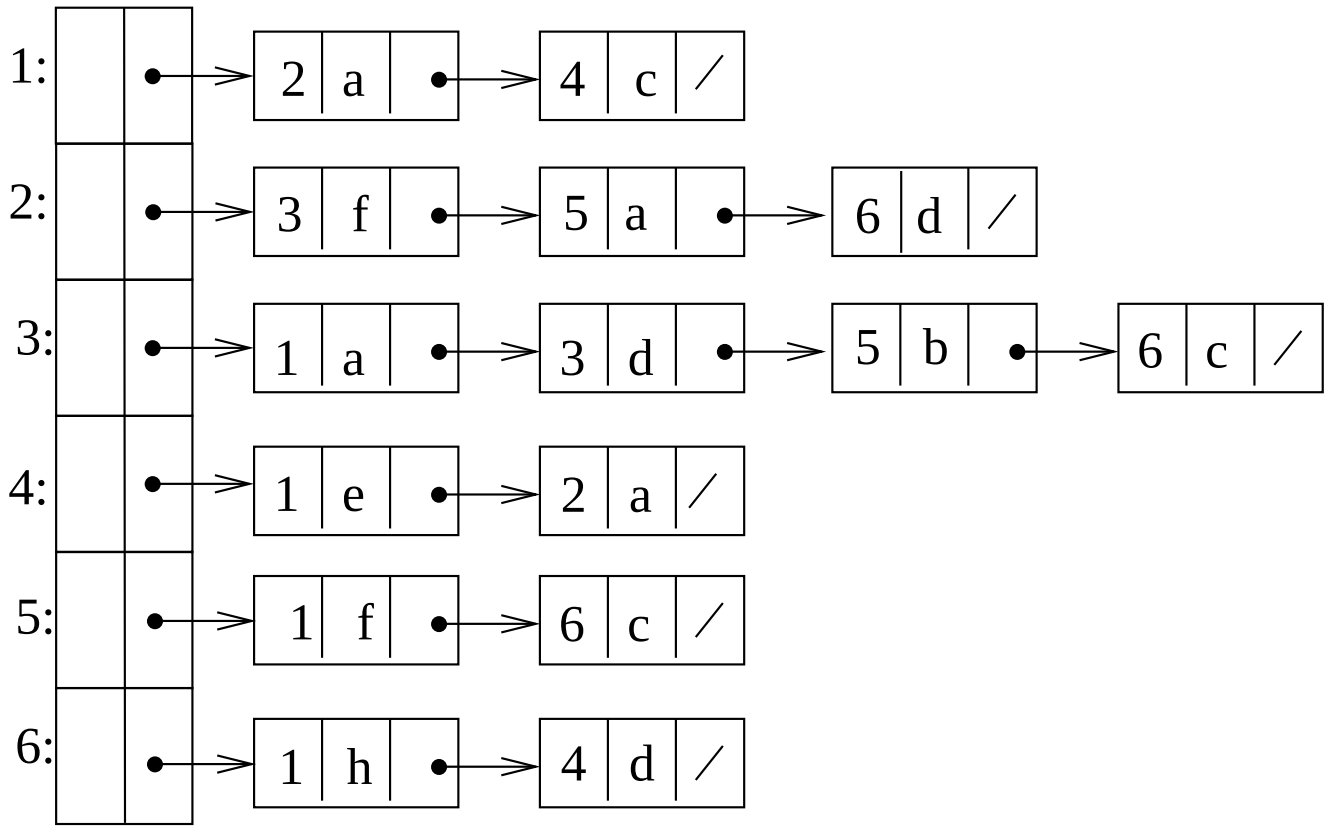
Beispiel mit $M = \{\mathrm{a}, \mathrm{b}, \ldots, \mathrm{z}\}$:

	1	2	3	4	5	6
1	–	a	–	c	–	–
2	–	–	f	–	a	d
3	a	–	d	–	b	c
4	e	a	–	–	–	–
5	f	–	–	–	–	c
6	h	–	–	d	–	–

$\widehat{=}$



Adjazenzlistendarstellung:



Adjazenzlistendarstellung:

Für jeden Knoten i gibt es Liste L_i, in der die

- *Nachfolger von* i (bei Digraphen) bzw. die
- *Nachbarn von* i (bei Graphen)

gespeichert sind.

Realisierung der Listen: L_i ist (z. B. einfach verkettete) lineare Liste, mit Kopf im Eintrag $\texttt{nodes}[i]$, für $1 \leq i \leq n$.

Beobachtungen

(i) Länge von L_i: outdeg(i) bei Digraphen, deg(i) bei Graphen.

(ii) Bei Graphen gilt: Eintrag i kommt in L_j vor $\Leftrightarrow$ Eintrag j kommt in L_i vor.

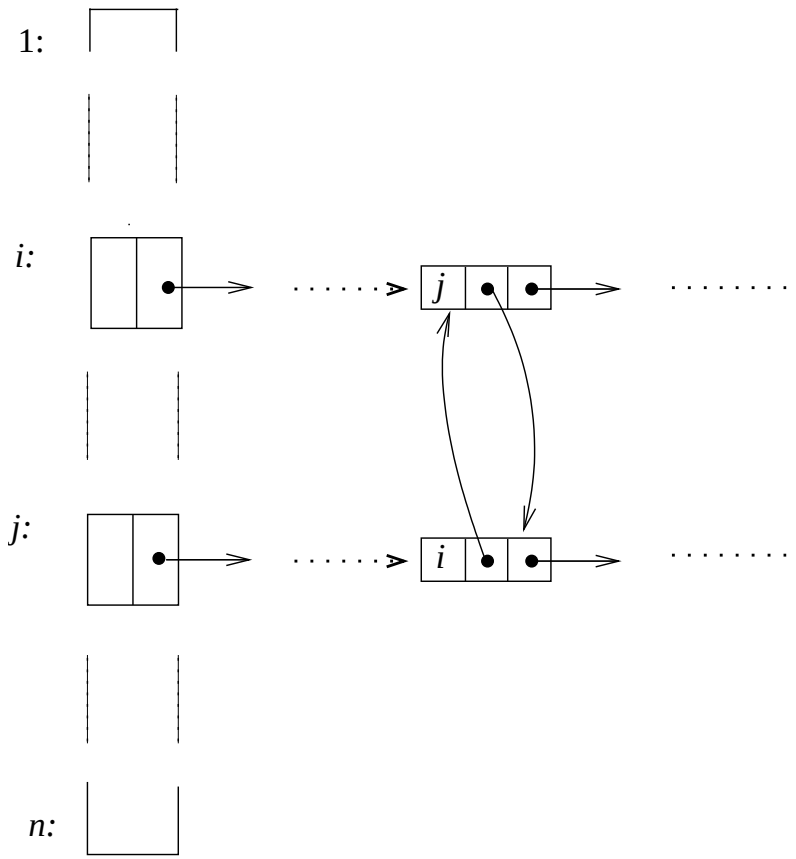
(iii) Durch die **Reihenfolge** der Einträge in Liste L_i sind die Nachbarn/Nachfolger von Knoten i **implizit angeordnet**.

Erweiterungen der Adjazenzlistenstruktur:

1) **Knotenbeschriftungen**: im `nodes`-Array zu platzieren.
2) **Kantenbeschriftungen**: in zusätzlichen Komponenten (Attributen) in den Elementen für die Adjazenzlisten.
3) In **Graphen**: Der Listeneintrag für j in L_i kann einen **Zeiger** zum Listeneintrag für i in L_j (die **„Gegenkante“**) enthalten.
4) Bei **Digraphen**: Der **Umkehrgraph** G^{R} enthält in der Adjazenzliste L_i^{R} für i die Knoten j, die *Vorgänger* von i in G sind.

Übung: Erstellen der Darstellung von G^{R} aus der von G in Zeit $O(|V| + |E|)$.

Querverweise `reverse_edge` bei Graphen:



Beobachtungen

Wenn ein Graph oder Digraph $G = (V, E)$ mittels Adjazenzlisten dargestellt wird, gilt:

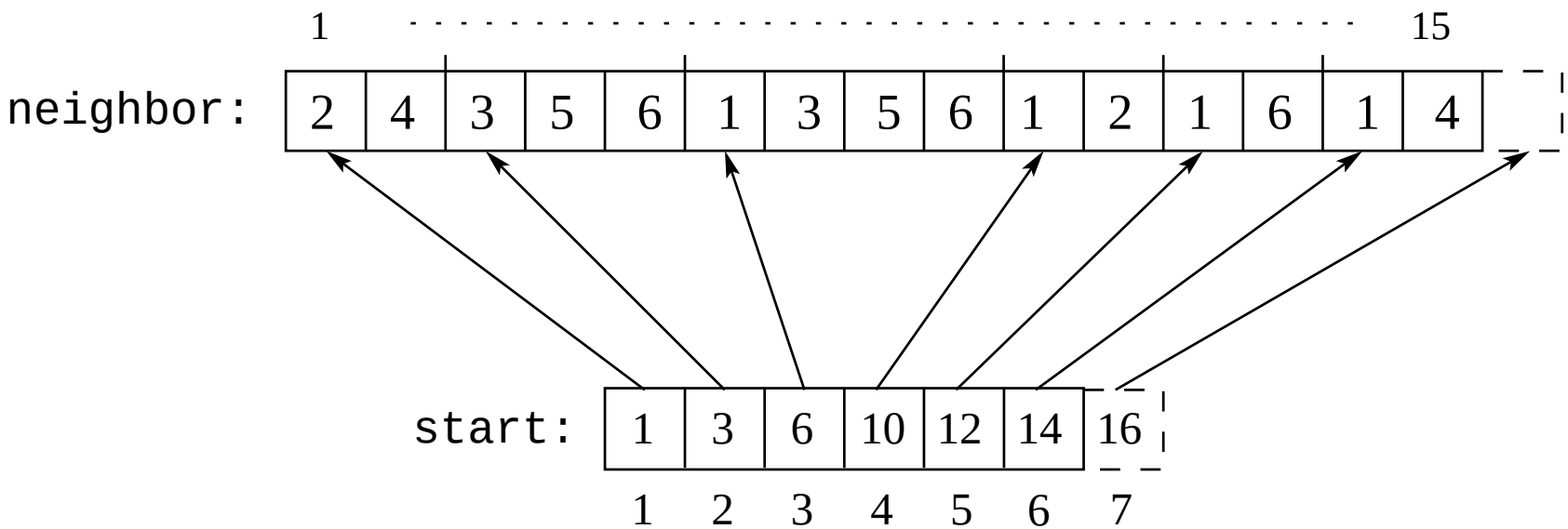
(a) Speicherplatz $O(|V| + |E|)$ wird benötigt;

(b) das Durchlaufen *aller* Kanten kann in Zeit $O(|V| + |E|)$ ausgeführt werden;

(c) das Durchlaufen der Adjazenzliste zu Knoten i benötigt Zeit $O(\deg(i))$ bzw. $O(\text{outdeg}(i))$;

(d) (nur) falls Vorgängerlisten vorliegen, können auch die Vorgänger von Knoten i in Zeit $O(\text{indeg}(i))$ durchlaufen werden.

Adjazenzarraydarstellung:



Die **Adjazenzarraydarstellung** besteht aus Arrays `neighbor`$[1..m]$ und `start`$[1..n+1]$.

In `neighbor`$[1..m]$ stehen nacheinander die Nachfolger/Nachbarn der Knoten $1, \ldots, n$, in dieser Reihenfolge.

Genauer: Sei $s_i = 1 + \sum_{1 \leq j < i} [\text{out}]\deg(j)$, für $1 \leq i \leq n+1$. Dann stehen im Segment `neighbor`$[s_i..s_{i+1} - 1]$ die (Indizes der) Nachfolger/Nachbarn von Knoten i.

In `start`$[1..n+1]$: `start`$[i] = s_i$, für $1 \leq i \leq n+1$. (Zum Navigieren benötigt.)

Verwendung: Wie Adjazenzlisten.

Die **Adjazenzarraydarstellung** ist günstig, wenn G über die Zeit unveränderlich ist.

Vorteile:
- Platzersparnis (keine Listenzeiger).
- Deutlich schnellerer Zugriff auf die Namen der Nachfolger bzw. der Nachbarn eines Knotens als bei der Adjazenzlistendarstellung.

(Bei einem Zugriff auf eine Position im Adjazenzarray wird ein ganzer Block in den Cache geladen.)

Übungsaufgabe: Geben Sie ein Verfahren an, mit dem sich in Linearzeit aus der Adjazenzarraydarstellung eines Graphen die entsprechende Darstellung von G^{R} konstruieren lässt.

7.3 Breitensuche* in Digraphen

1. Breitensuche von einem Knoten v_0 aus:

Finde alle Knoten v, die von v_0 aus **erreichbar** sind.

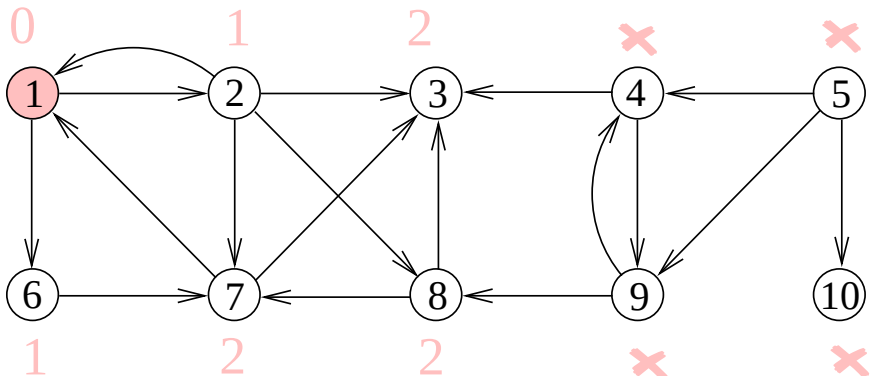
Nummeriere die entdeckten Knoten.

Bestimme für jeden erreichbaren Knoten v einen **Weg** von v_0 nach v mit **minimaler Länge**, und notiere diese Länge.

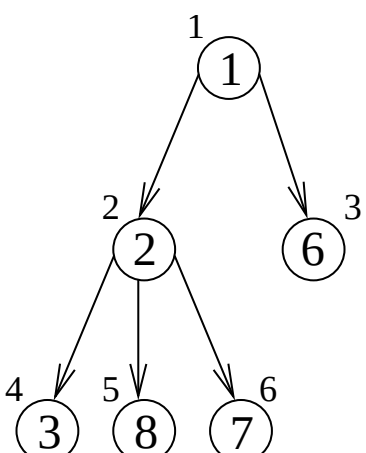
Diese Länge heißt der **Abstand** von v_0 nach v, kurz $d(v_0, v)$, oder das **Level** von v (von v_0 aus gemessen).

---

* englisch: **breadth-first-search**, kurz **BFS**.



Die von $v_0 = 1$ aus erreichbaren Knoten, levelweise angeordnet und nummeriert, mit kürzesten Wegen von $v_0 = 1$ aus:



Ansatz: Knoten werden **„entdeckt“**

(nummeriert, als „entdeckt“ markiert, mit Levelnummer versehen)

und später **„bearbeitet“**

(d. h. die Menge der Nachfolger – Adjazenzliste/Adjazenzarrayabschnitt – wird durchlaufen, um neue Knoten zu entdecken).

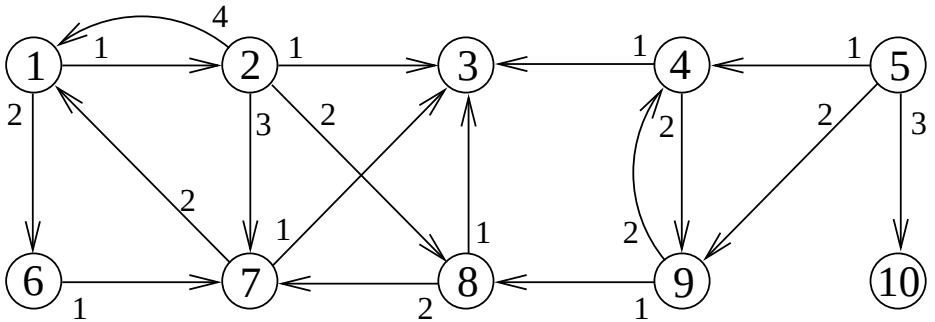
Reihenfolge der Bearbeitung wird über eine **Queue** Q organisiert. (Enthält anfangs v_0.)

Information an Knoten v (z. B. gespeichert in `nodes`-Array):

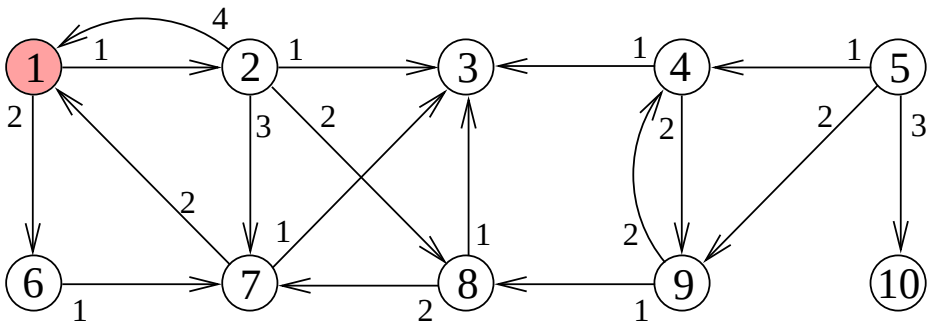
- **Nummer** $\mathit{bfs_num}(v)$
- **Level** $\mathit{level}(v)$ (Abstand von v_0 nach v)
- $p(v)$ (**Vorgänger** von v auf kürzestem Weg $v_0 \rightsquigarrow v$)

Zähler `bfs_count`: Anzahl der bisher entdeckten Knoten.

Hinweis: Kleine Ziffern an ausgehenden Kanten geben die Reihenfolge in der Adjazenzliste an.

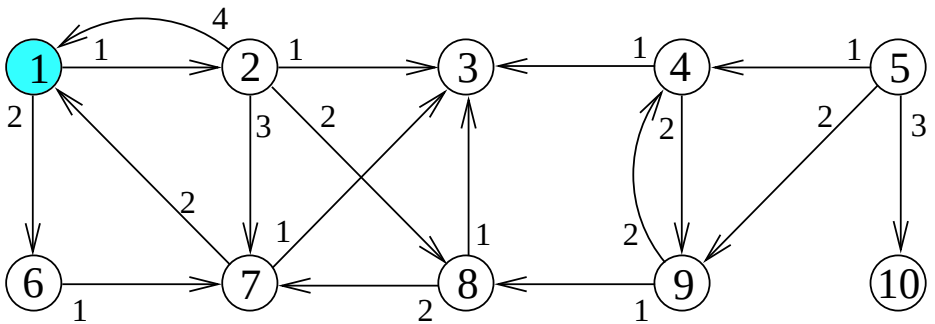


entdeckte Knoten
bfs-Nummer
Queue
Level
Vorgänger



in Queue

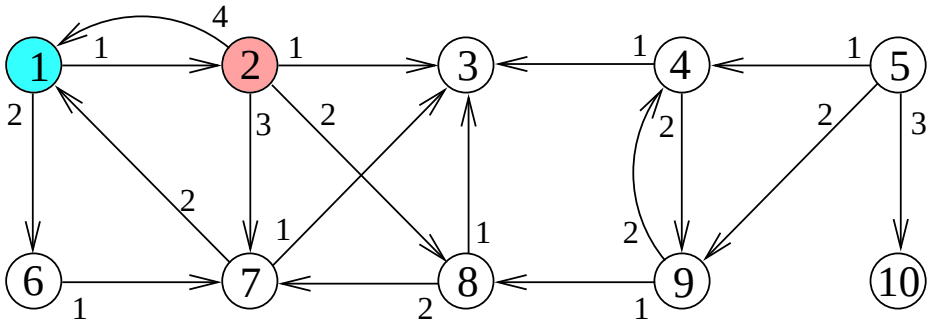
entdeckte Knoten	1
bfs-Nummer	1
Queue	1
Level	0
Vorgänger	(1)



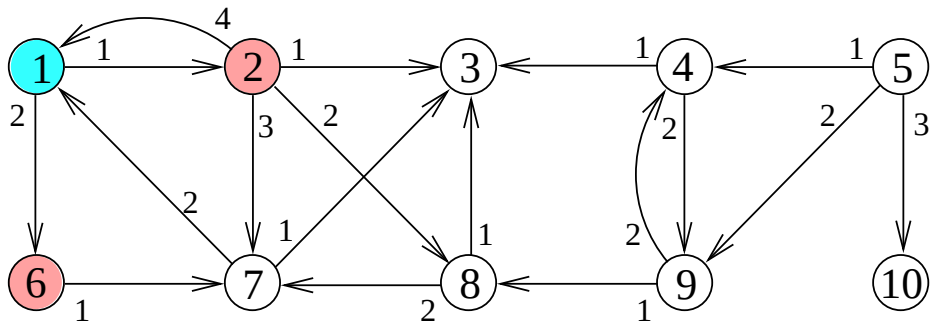
in Queue

in Bearbeitung

entdeckte Knoten	1
bfs-Nummer	1
Queue	
Level	0
Vorgänger	(1)



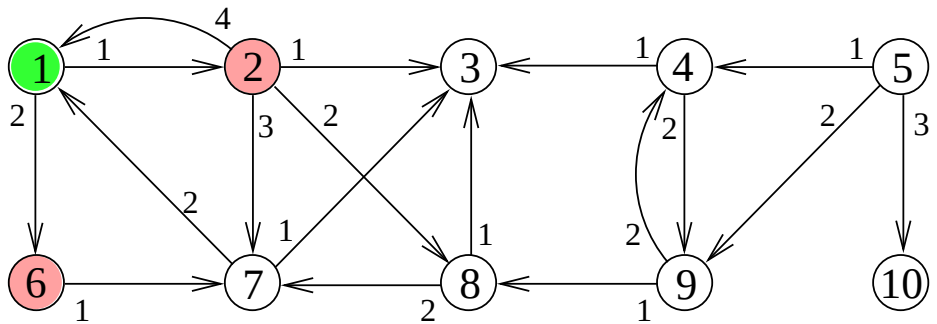
entdeckte Knoten	1	2
bfs-Nummer	1	2
Queue		2
Level	0	1
Vorgänger	(1)	1



in Queue

in Bearbeitung

entdeckte Knoten	1	2	6
bfs-Nummer	1	2	3
Queue		2	6
Level	0	1	1
Vorgänger	(1)	1	1

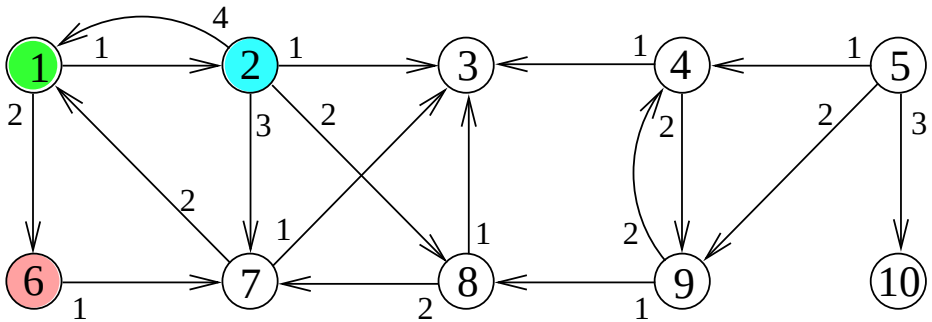


in Queue

in Bearbeitung

fertig

entdeckte Knoten	1	2	6
bfs-Nummer	1	2	3
Queue		2	6
Level	0	1	1
Vorgänger	(1)	1	1

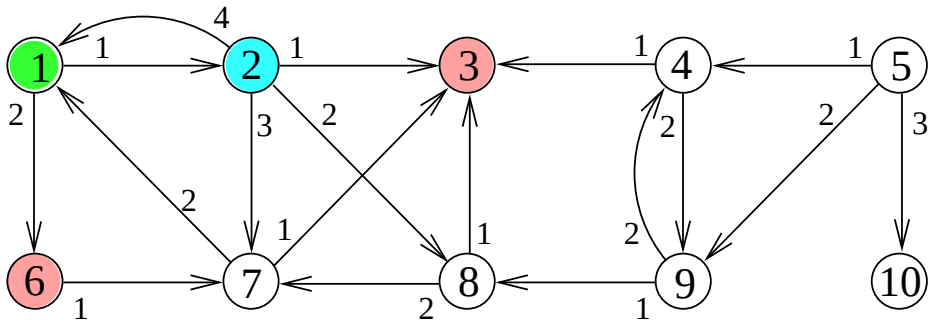


in Queue

in Bearbeitung

fertig

entdeckte Knoten	1	2	6
bfs-Nummer	1	2	3
Queue			6
Level	0	1	1
Vorgänger	(1)	1	1

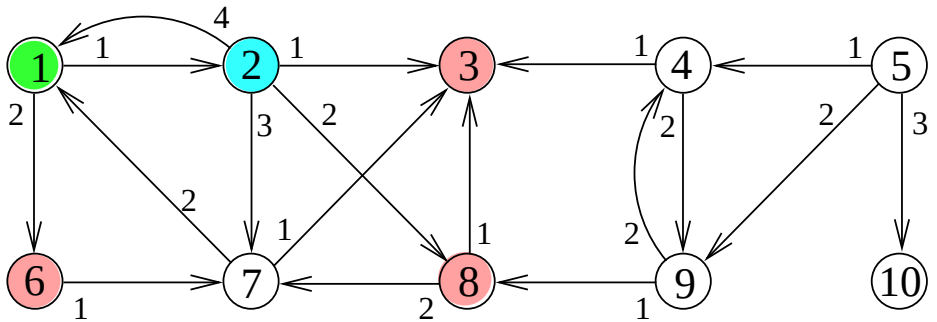


in Queue

in Bearbeitung

fertig

entdeckte Knoten	1	2	6	3
bfs-Nummer	1	2	3	4
Queue			6	3
Level	0	1	1	2
Vorgänger	(1)	1	1	2

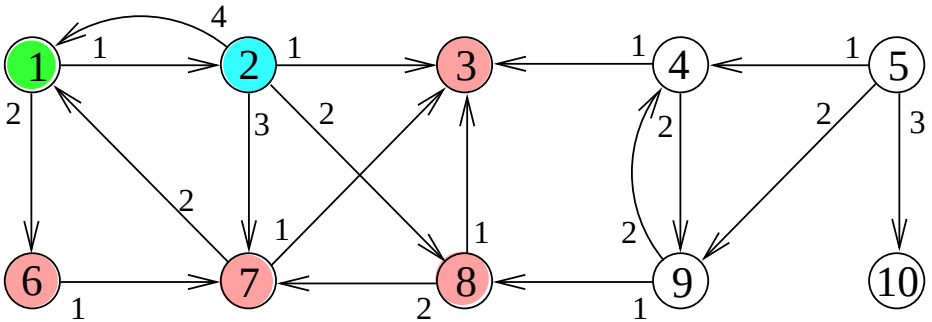


in Queue

in Bearbeitung

fertig

entdeckte Knoten	1	2	6	3	8
bfs-Nummer	1	2	3	4	5
Queue			6	3	8
Level	0	1	1	2	2
Vorgänger	(1)	1	1	2	2

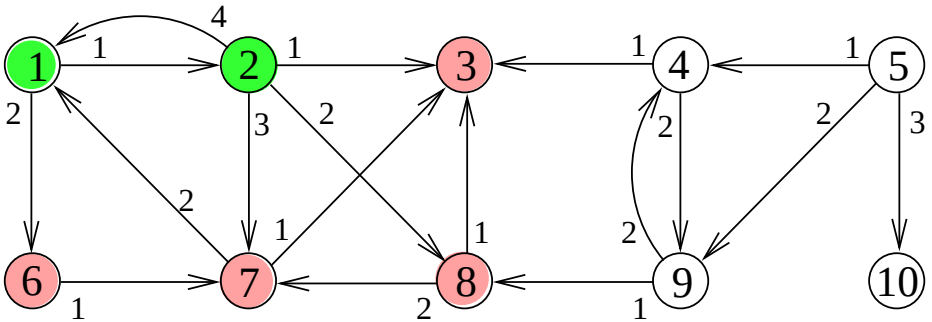


in Queue

in Bearbeitung

fertig

entdeckte Knoten	1	2	6	3	8	7
bfs-Nummer	1	2	3	4	5	6
Queue			6	3	8	7
Level	0	1	1	2	2	2
Vorgänger	(1)	1	1	2	2	2

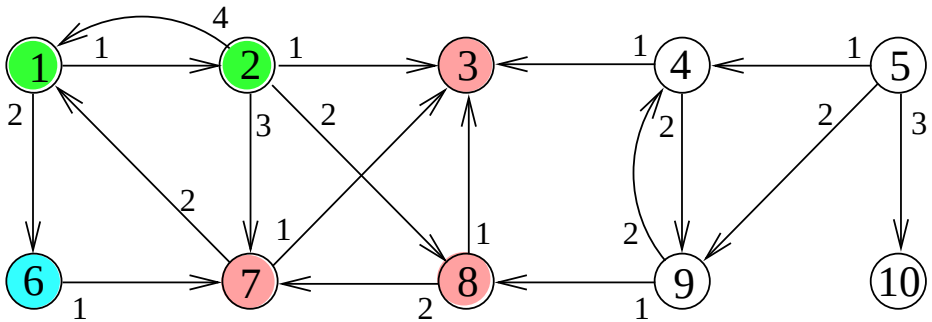


in Queue

in Bearbeitung

fertig

entdeckte Knoten	1	2	6	3	8	7
bfs-Nummer	1	2	3	4	5	6
Queue			6	3	8	7
Level	0	1	1	2	2	2
Vorgänger	(1)	1	1	2	2	2

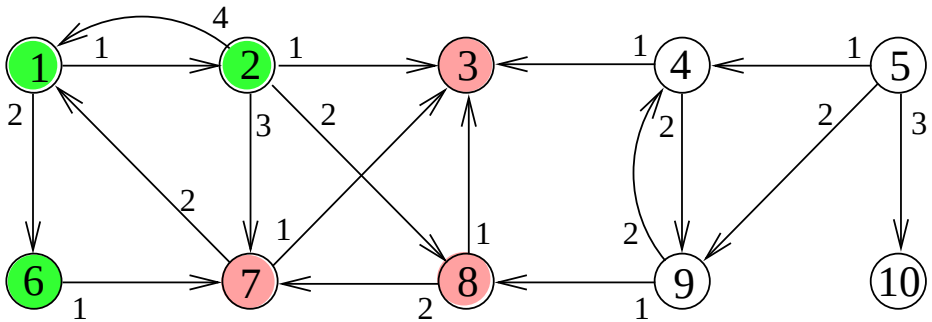


in Queue

in Bearbeitung

fertig

entdeckte Knoten	1	2	6	3	8	7
bfs-Nummer	1	2	3	4	5	6
Queue				3	8	7
Level	0	1	1	2	2	2
Vorgänger	(1)	1	1	2	2	2

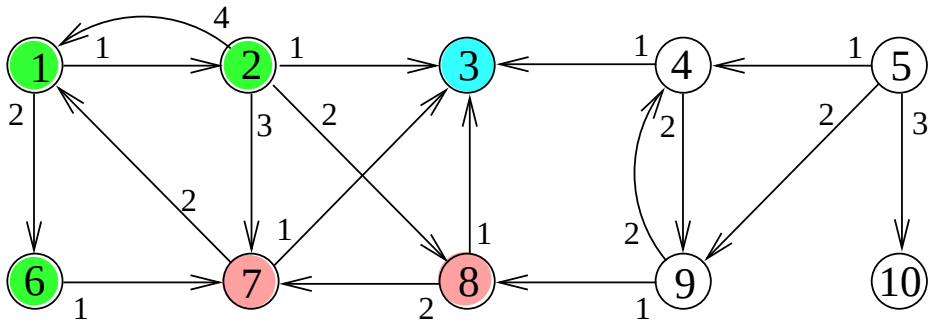


in Queue

in Bearbeitung

fertig

entdeckte Knoten	1	2	6	3	8	7
bfs-Nummer	1	2	3	4	5	6
Queue				3	8	7
Level	0	1	1	2	2	2
Vorgänger	(1)	1	1	2	2	2



in Queue

in Bearbeitung

fertig

entdeckte Knoten	1	2	6	3	8	7
bfs-Nummer	1	2	3	4	5	6
Queue					8	7
Level	0	1	1	2	2	2
Vorgänger	(1)	1	1	2	2	2

. . . und so weiter

Initialisierung: `bfs_count` $\leftarrow 0$;
Alle *bfs_num*(v) werden auf 0 gesetzt. `Q`: Queue.

Prozedur bfs(v_0) // Breitensuche von v_0 aus, v_0 wird **entdeckt**

```
(1)  bfs_count++;
(2)  bfs_num[v_0] ← bfs_count;
(3)  level[v_0] ← 0;
(4)  p[v_0] ← v_0; // Kennzeichne Startknoten
(5)  Q.enqueue(v_0); // Startknoten in Warteschlange stecken
(6)  while not Q.isempty do
(7)    v ← Q.dequeue; // nun bearbeite v:
(8)    für jeden Nachfolger w von v (Adjazenziste!) tue:
(9)      if bfs_num[w] = 0 then // w wird entdeckt:
(10)       bfs_count++;
(11)       bfs_num[w] ← bfs_count;
(12)       level[w] ← level[v] + 1;
(13)       p[w] ← v; // w von v aus erreicht
(14)       Q.enqueue(w);
```

Lemma 7.3.1 Genau diejenigen Knoten, die von v_0 aus erreichbar sind, werden **entdeckt**, erhalten also eine bfs-Nummer, eine Levelnummer und einen Vorgänger, und sie werden **bearbeitet**, d. h., ihre Adjazenzliste wird durchlaufen.

Beweis: Alle entdeckten Knoten sind erreichbar, da man sich entlang von Kanten voranbewegt. – Umgekehrt: Betrachte Weg $(v_0, v_1, \ldots, v_k)$ mit $v = v_k$. Durch Induktion über i zeigt man, dass jedes v_i vom Algorithmus entdeckt wird. Dann landet v_i in Q, wird also irgendwann bearbeitet.

Beobachtungen

Setze $\boldsymbol{R_v} := \{w \in V \mid v \rightsquigarrow w\}$, $\boldsymbol{E_v} := \{(w,u) \in E \mid v \rightsquigarrow w\}$.

1) Die Laufzeit eines Aufrufs **bfs**(v_0) ist $\Theta(|R_{v_0}| + |E_{v_0}|)$.

Beweis: Jeder Knoten $w \in R_{v_0}$ wird einmal entdeckt (Zeit $O(1)$) und einmal bearbeitet (Zeit $O(\mathrm{outdeg}(w))$). Insgesamt ist das $O(|R_{v_0}|) + O(\sum_{w \in R_{v_0}} \deg(w)) = O(|R_{v_0}|) + O(|E_{v_0}|)$.

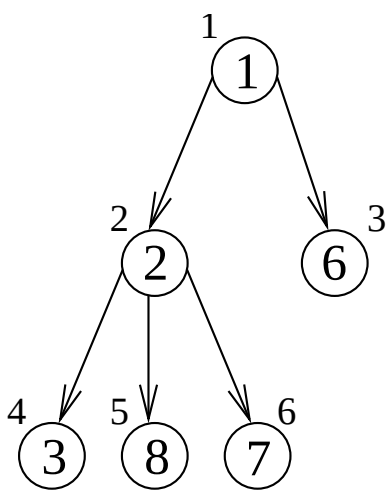
2) Der Speicherplatzbedarf für die Warteschlange ist $O(|R_{v_0}|)$.

Beobachtungen

3) Für $v \neq v_0$ ist $p(v)$ ist der Knoten, von dem aus v entdeckt wird.
Knoten $p(v)$ erfüllt $\mathit{level}(p(v)) = \mathit{level}(v) - 1$.

Also: Wenn man von v zu $v_1 = p(v)$ geht, dann zu $v_2 = p(v_1)$, usw., erreicht man nach genau $\mathit{level}(v)$ Schritten die Wurzel.

Also: Die entdeckten Knoten mit den Kanten $(p(v), v)$, $v \neq v_0$, bilden einen Baum,
den **Breitensuchbaum** oder **BFS-Baum**.



Lemma 7.3.2

Zu jedem Zeitpunkt gilt: Es gibt ein $\ell \geq 0$, so dass sich in der Queue nur Knoten mit Levelnummern ℓ (vorne) und $\ell + 1$ (hinten) befinden. Dieses ℓ ist monoton nichtfallend. Insbesondere sind die Levelnummern der nacheinander aus der Queue entnommenen Knoten nichtfallend.

Beweis: Einfacher Induktionsbeweis über die Runden des Algorithmus.

Lemma 7.3.3

Wenn $(v, w) \in E$, dann gilt *level*$(w) \leq$ *level*$(v) + 1$.

Beweis: Aus Lemma 7.3.2 folgt: Bevor Bearbeitung von v beginnt, wurde noch kein Knoten mit Level $> \ell =$ *level*(v) bearbeitet.
$\Rightarrow$ es wurde noch keine Levelnummer $>$ *level*$(v) + 1$ vergeben. Also:
Fall 1: w zu diesem Zeitpunkt **schon entdeckt**: *level*$(w) \leq \ell + 1$.
Fall 2: w **noch nicht entdeckt**: w wird beim Durchlauf durch die Adjazenzliste von v entdeckt und erhält Levelnummer $\ell + 1$.

Proposition 7.3.4

Nach der Durchführung von **bfs**(v_0) gilt für jeden dabei entdeckten Knoten v:

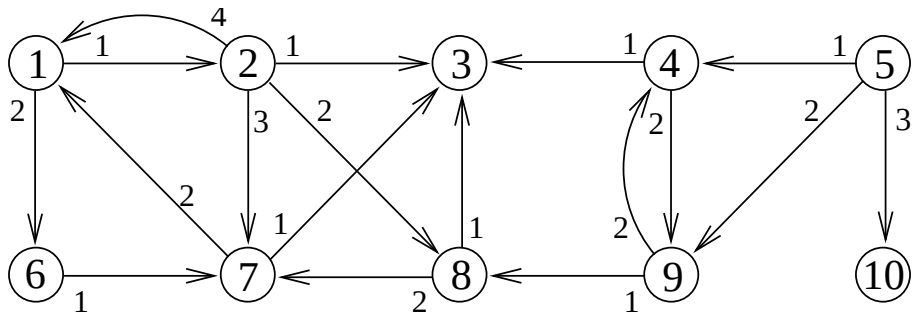
level(v) ist die Länge eines kürzesten Weges von v_0 nach v.

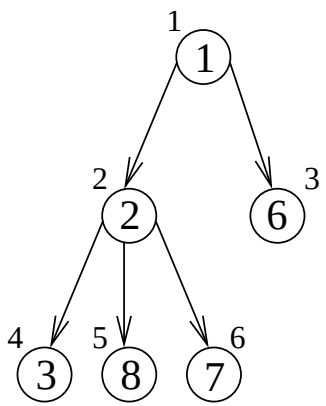
Beweis: Betrachte den Weg von v_0 nach v, der sich durch
Rückwärtsverfolgen der p-Zeiger von v aus ergibt. Länge: *level*(v).

Sei $p = (v_0, v_1, \ldots, v_r)$ ein **beliebiger** Weg mit $v_r = v$.

Nach Lemma 7.3.3 gilt *level*$(v_i) \leq$ *level*$(v_{i-1}) + 1$,
für $1 \leq i \leq r$, und *level*$(v_0) = 0$, also *level*$(v) =$ *level*$(v_r) \leq r$.

Weil p beliebig war, hat **jeder** Weg von v_0 nach v mindestens Länge *level*(v). □





Wie erreichen wir die restlichen Knoten in G?

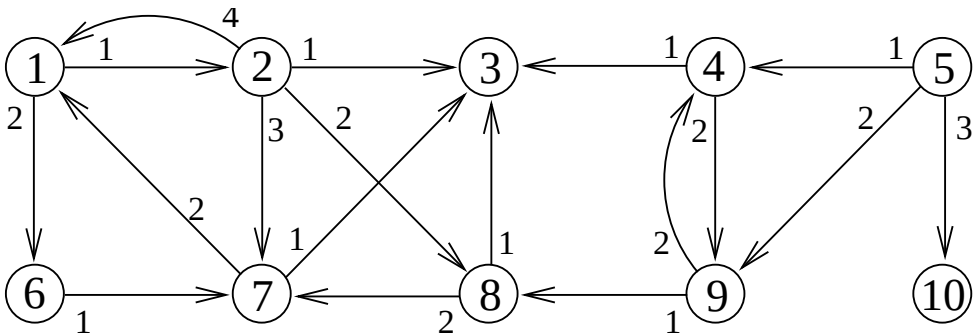
Starte eine Breitensuche bei jedem Knoten v, der nicht bei einer vorherigen Breitensuche entdeckt worden ist. Behalte dabei die vorher vergebenen Nummern bei.

2. Globale Breitensuche in G:

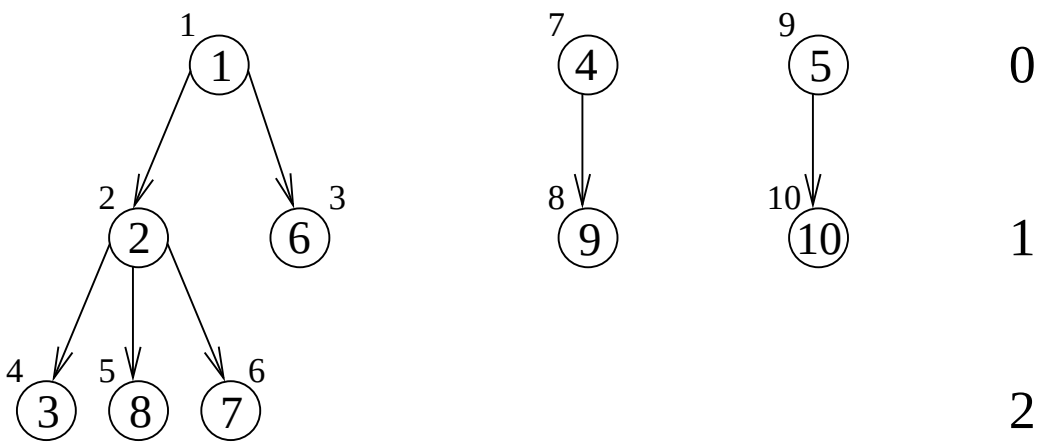
Algorithmus BFS(G) // Breitensuche in $G = (V, E)$

(1) $\texttt{bfs_count} \leftarrow 0$;
(2) **for** v **from** 1 **to** n **do** *bfs_num*$(\mathrm{v}) \leftarrow 0$;
(3) **for** v **from** 1 **to** n **do**
(4) **if** $\texttt{bfs_num}[\mathrm{v}] = 0$ **then**
(5) **bfs**(v); // starte Breitensuche von v aus

Anmerkung zu Zeilen (2), (4):
Die *bfs_num*-Werte werden von den **bfs**-Aufrufen verändert;
sie haben also beim Wieder-Lesen in der zweiten Schleife nicht unbedingt immer noch den Initialisierungs-Wert 0.



Entstehende BFS-Bäume mit BFS-Nummern und Levels:



Bei **bfs**-Aufruf für neuen Knoten `bfs_count` einfach weiterzählen.

Welche Knotenmengen werden bei den einzelnen Aufrufen entdeckt?

Achtung: Im Baum mit Wurzel v stehen nicht alle Knoten, die von v aus erreichbar sind, sondern . . .

Für $v \in V$ hatten wir: $R_v = \{w \in V \mid v \rightsquigarrow w\}$. Wir setzen

$W_v := \bigcup_{u \leq v} R_u = \{w \in V \mid \exists u \leq v \colon u \rightsquigarrow w\}$.

Dann ist $W_1 \subseteq W_2 \subseteq \cdots \subseteq W_n = V$.

Behauptung 7.3.5

(a) v ist Wurzel eines Breitensuchbaums $\Leftrightarrow v \notin W_{v-1}$.

(b) Wenn v Wurzel ist, dann befinden sich genau die Knoten aus $R_v - W_{v-1}$ im Breitensuchbaum unter v.

Beweis: „(a) $\Leftarrow$“: Indirekt. *Annahme*: $v \notin W_{v-1}$ und v ist keine Wurzel eines BFS-Baums. Da v keine Wurzel eines BFS-Baums ist, gilt bei der Betrachtung von v in Zeilen (3)–(5) $\texttt{bfs_num}[v] > 0$. Das heißt, dass vorher v in einen BFS-Baum mit Wurzel $u < v$ eingebaut wurde. Also gibt es einen Weg von u nach v, d. h. $v \in W_u$. Nun ist $W_u \subseteq W_{v-1}$, also auch $v \in W_{v-1}$, Widerspruch.

„(a) $\Rightarrow$“: Indirekt. *Annahme*: v ist Wurzel eines BFS-Baums und $v \in W_{v-1}$. Da $v \in W_{v-1}$ ist, gibt es $u < v$, so dass ein Weg $u = u_0, \ldots, u_k = v$ von u nach v existiert. Durch vollständige Induktion über i zeigt man: u_i wird entdeckt, bevor v in der Schleife in Zeilen (3)–(5) getestet wird. Es folgt, dass auch $u_k = v$ vor Runde v der Schleife in Zeilen (3)–(5) entdeckt wird und eine BFS-Nummer erhält. Also gilt $\texttt{bfs_num}[v] > 0$ in der Runde, wo v getestet wird, und v wird keine Wurzel eines BFS-Baums, Widerspruch.

(b) Man zeigt, dass alle Knoten aus W_v in den Bäumen mit Wurzeln $\leq v$ sitzen. □

2. Breitensuche von einer Knotenmenge V_0 aus:

Für eine beliebige Menge $V_0 \subseteq V$, $V_0 \neq \emptyset$, lade anfangs **alle** $v \in V_0$ in die Queue, setze *level*$(v) = 0$ für $v \in V_0$.

Die modifizierte Prozedur findet dann alle von V_0 aus erreichbaren Knoten.

level(w) ist die minimale Länge eines Weges von einem Knoten $v \in V_0$ zu w.

Beweis: Wie im Fall des Startens von einem Knoten. □

3. Breitensuche in (ungerichteten) Graphen: ubfs, UBFS

Verläuft analog **bfs**. – Beachte: Bei der Bearbeitung von v wird der Knoten $p(v)$, obwohl er in der Adjazenzliste von v vorkommt, niemals berücksichtigt (hat schon eine Nummer).

Wenn man in v_0 startet, werden genau die Knoten in der Zusammenhangskomponente von v_0 entdeckt.

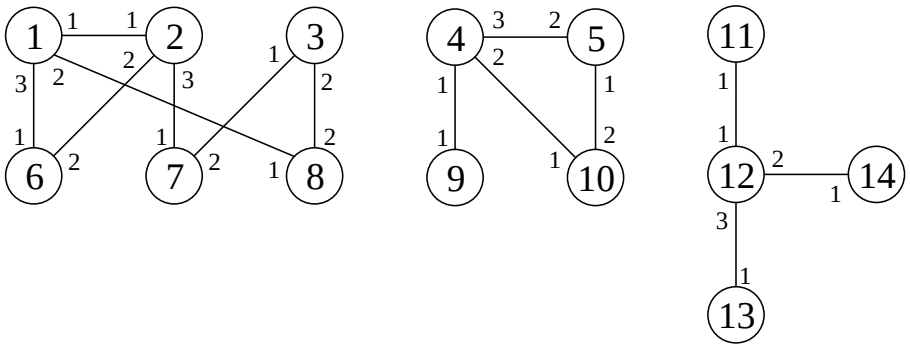
Es ist klar, dass keine anderen Knoten entdeckt werden können. Wenn $(v_0, v_1, \ldots, v_r)$ ein Weg ist, dann sieht man durch Induktion über i ein, dass jeder Knoten v_i auf dem Weg entdeckt wird.

level(v) entspricht dabei dem Abstand von v zu v_0 – der Länge eines kürzesten Wegs von v_0 zu v.

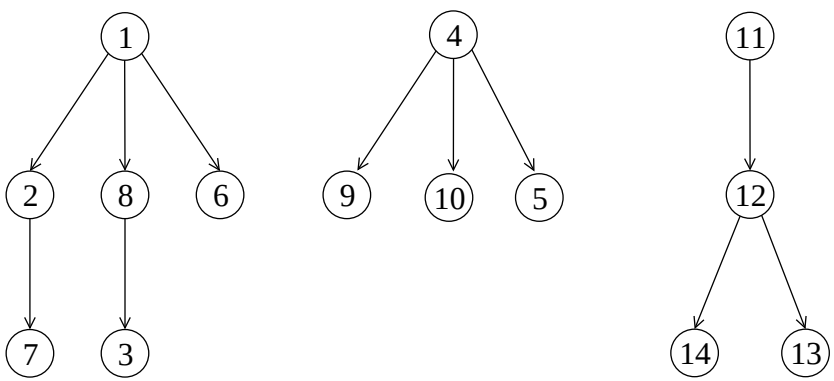
Wenn man in allen noch nicht entdeckten Knoten eine Breitensuche startet (**UBFS**(G)), findet man (in Zeit $O(|V| + |E|)$) alle Zusammenhangskomponenten.

Für diesen Zweck wird BFS in ungerichteten Graphen bevorzugt benutzt.

Beispiel: Ungerichtete BFS in



liefert einen Breitensuchbaum für jede Zusammenhangskomponente:



Wenn eine Zusammenhangskomponente ein Baum ist: BFS installiert den Startknoten als Wurzel und berechnet Abstände von dieser Wurzel.

Kosten für BFS in ungerichteten Graphen: $O(|V| + |E|)$.

Speicherplatz für die Queue: $O(|V|)$.

Genauer: $O(\text{Anzahl Knoten in der größten Zusammenhangskomponente})$.

Wenn man nur eine Zusammenhangskomponente absucht: Kosten entsprechend der Größe dieser Komponente.

Video: Beweise zu Breitensuche

SS 2021

Algorithmen und Datenstrukturen

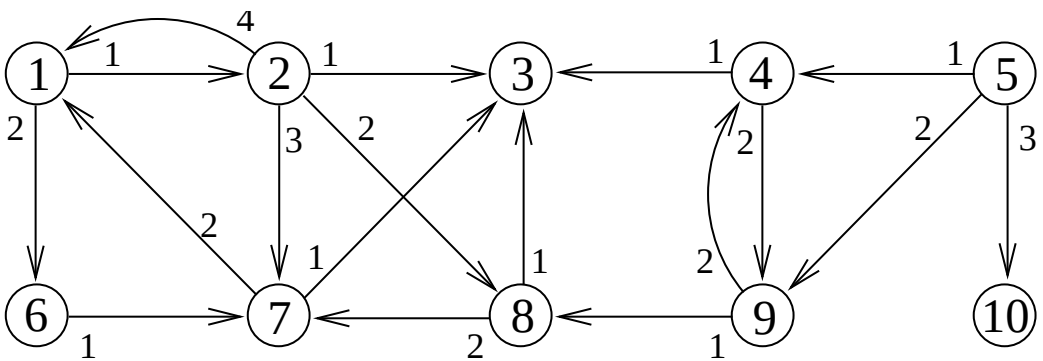
8. Kapitel

Tiefensuche

Martin Dietzfelbinger

Juni 2021

8.1 Einfache Tiefensuche in Digraphen



Eingabeformat:
Digraph $G = (V, E)$, Knotenmenge $V = \{1, \ldots, n\}$ (o. B. d. A.),
in Adjazenzlisten-/array- oder Adjazenzmatrixdarstellung,
aus v ausgehende Kanten sind (implizit) angeordnet.

Ziele:

- Besuche **alle** Knoten und Kanten.
- Sammle einfache **Strukturinformation**.

Tiefensuche (Depth-First-Search)

Zunächst: Gehe von Knoten v_0 aus, finde alle auf Wegen von v_0 aus erreichbaren Knoten und Kanten.

Vorwärtsgehen hat **Vorrang**! **Effekt:**
Die Folge der entdeckten Knoten geht immer so weit wie möglich „in die Tiefe“.

Realisierung: Rekursive Prozedur „**dfs(v)**“:

- „Besuche“ Knoten v (Aktion an diesem Knoten).
- Dann betrachte die Nachfolger $w_1, \ldots, w_{\mathrm{outdeg}(v)}$ von v nacheinander, aber:

> Sobald **neuer** Knoten w „entdeckt“ wird, starte **sofort** dfs(w).

(Weitere Nachfolger von v werden später betrachtet.)

Man muss verhindern, dass Knoten v mehrfach besucht wird.

Dazu: „Statusinformation“, in Array `status`$[1\,..\,n]$ (oder im `nodes`Array):

v **neu** – noch nie gesehen

v **aktiv** – dfs(v) gestartet, noch nicht beendet

v **fertig** – dfs(v) beendet

Initialisierung: Alle Knoten sind **neu**.

Einfachversion:

Knoten werden in der Reihenfolge der Entdeckung durchnummeriert:

`dfs_num`$[1\,..\,n]$ speichert **Tiefensuch-Nummerierung**.

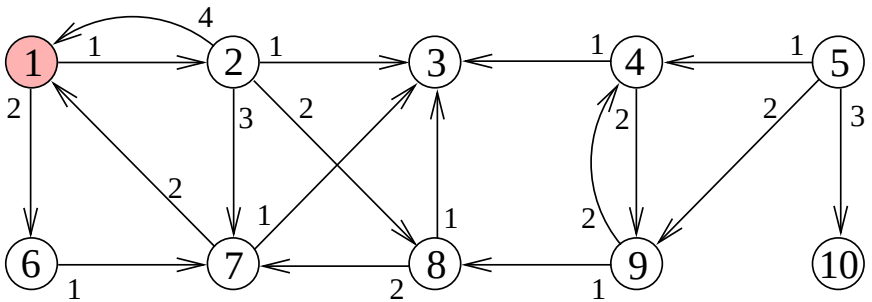
Mitzählen in `dfs_count` (globale Variable), mit 0 initialisiert.

dfs-visit(v): Aktion an v bei Entdeckung (anwendungsabhängig).

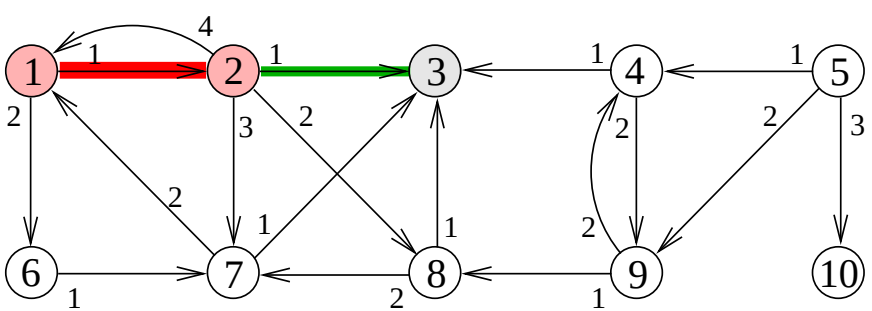
Prozedur dfs(v) // Tiefensuche an v, **rekursiv**,
// nur für v mit `status[`v`]` $=$ **neu** aufzurufen

(1) `dfs_count++;`
(2) `dfs_num[`v`]` $\leftarrow$ `dfs_count;`
(3) `status[`v`]` $\leftarrow$ **aktiv** ;
(4) dfs-visit(v); // Aktion an v bei Erstbesuch
(5) für jeden Nachfolger `w` von v (Adjazenzliste!) tue:
(6) **if** `status[w]` $=$ **neu** **then** // `w` wird entdeckt!
(7) dfs(`w`);
(8) `status[`v`]` $\leftarrow$ **fertig** .

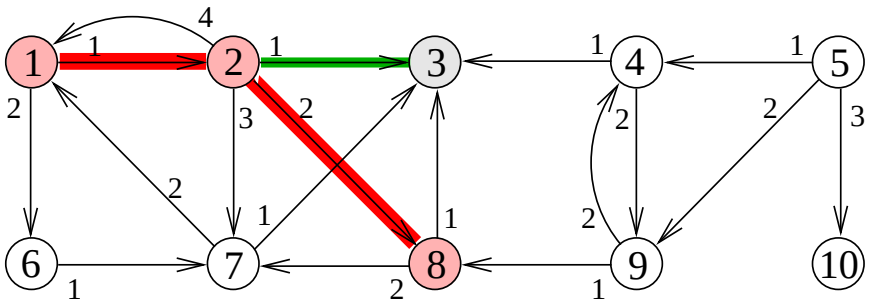
Tiefensuche von v_0 aus:
Initialisiere alle Knoten als „**neu**“; setze `dfs_count` $\leftarrow 0$.
Rufe dfs(v_0) auf.



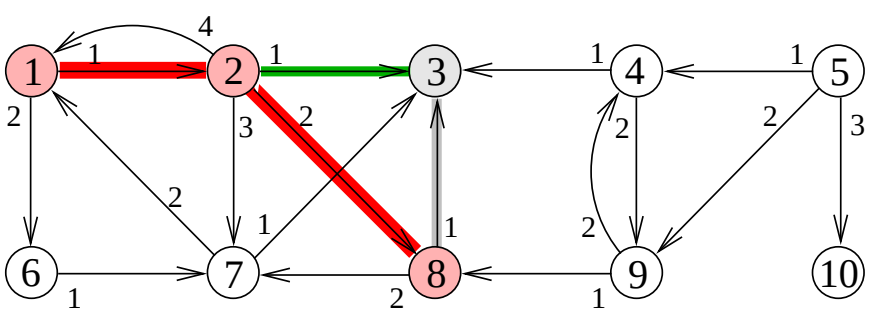
Start von dfs(1), „Roter Weg“: $\mathbf{(1)}$.



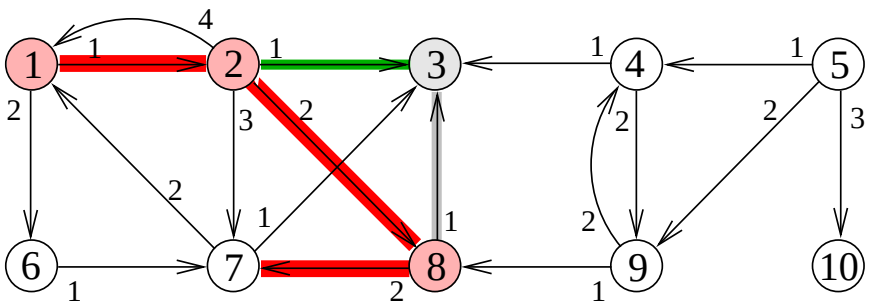
Ende von dfs(3), „Roter Weg“: $\mathbf{(1, 2)}$.



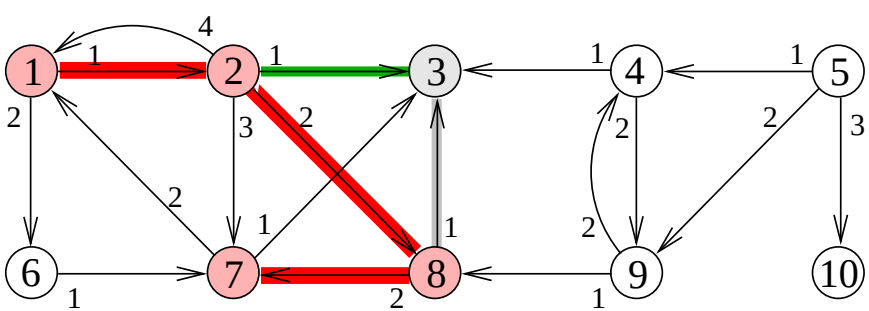
Start von dfs(8), „Roter Weg“: $\mathbf{(1, 2, 8)}$.



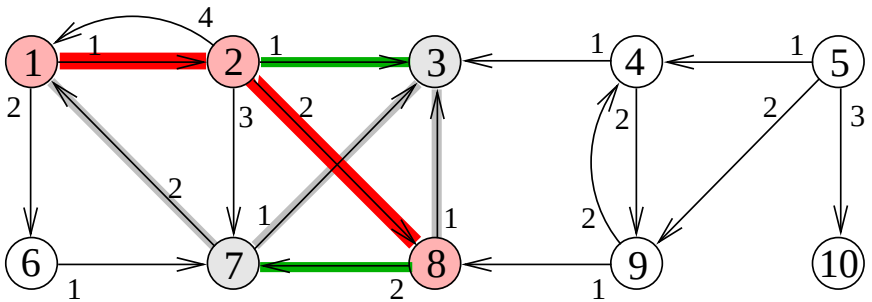
Kante $(8, 3)$ wird angesehen, Knoten 3 ist **fertig** .



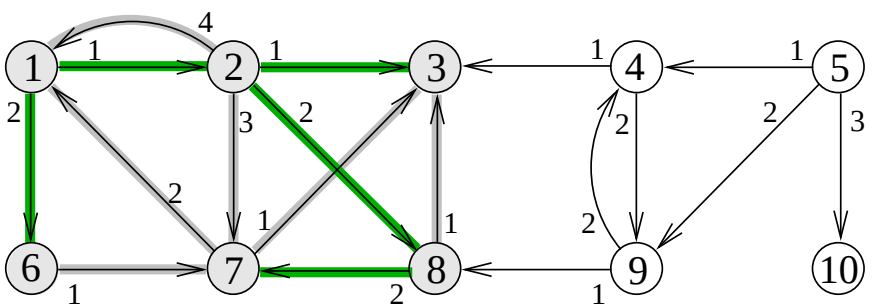
Kante $(8, 7)$ wird angesehen, Knoten 7 ist **neu** .



Start von dfs(7), „Roter Weg“: $\mathbf{(1, 2, 8, 7)}$.



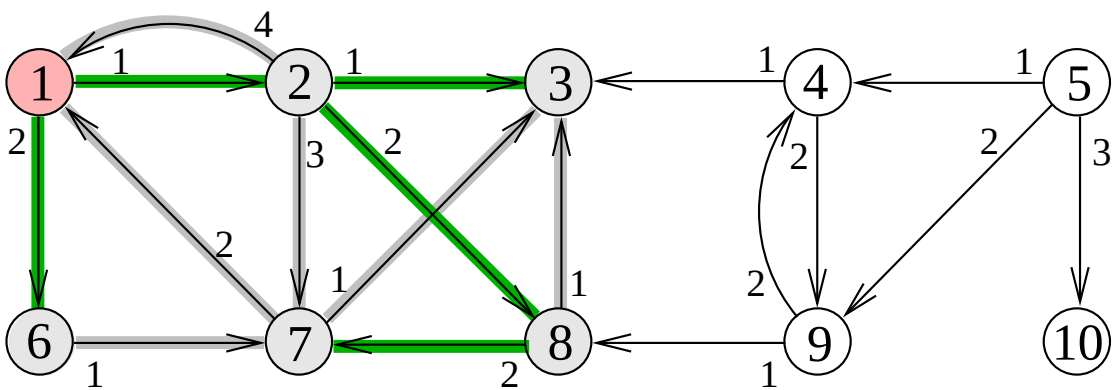
dfs(7) endet, „Roter Weg“: $\mathbf{(1, 2, 8)}$.



Fertig! Erreichbare Knoten und Kanten, **DFS-Baum** zu Startknoten 1.

$\boldsymbol{R_{v_0}} = \{v \in V \mid v_0 \rightsquigarrow v\}$. (Von v_0 aus erreichbare **Knoten**.)

$\boldsymbol{E_{v_0}} = \{(v, w) \in E \mid v_0 \rightsquigarrow v\}$. (Von v_0 aus erreichbare **Kanten**.)



Satz 8.1.1

Bei Tiefensuche von v_0 aus wird für jeden Knoten $v \in R_{v_0}$ dfs(v) (genau einmal) aufgerufen, für Knoten $v \notin R_{v_0}$ wird dfs(v) nicht aufgerufen. Jede Kante $(v, w) \in E_{v_0}$ wird genau einmal betrachtet, andere Kanten nicht.

Anm.: Auch Breitensuche von v_0 aus besucht genau diese Knoten/Kanten.

Beweis:

(1) Da Aufrufe nur entlang von Kanten von G erfolgen,
kann dfs(v) nur aufgerufen werden, wenn $v_0 \rightsquigarrow v$ gilt.

(2) Sobald dfs(v) aufgerufen wird, wird `status`$[v]$ auf „aktiv“ gesetzt, später auf „fertig“, aber nie mehr zurück auf „neu“. Ein Aufruf erfolgt nur für „neue“ Knoten. Daher: **maximal** ein Aufruf für v.

(3) Sei $(v_0, v_1, \ldots, v_t = v)$ ein Weg von v_0 nach v.
Annahme: Es gibt $i > 0$, für das dfs(v_i) nicht aufgerufen wird.
Betrachte das kleinste solche i. Dann wird dfs(v_{i-1}) aufgerufen.
Dabei wird die Kante (v_{i-1}, v_i) untersucht und festgestellt, dass v_i „neu“ ist. Also wird dfs(v_i) aufgerufen, **Widerspruch**.

(4) Eine Kante $(v, w) \in E_{v_0}$ wird im Aufruf dfs(v) untersucht (und in keinem anderen Aufruf).
Kanten (v, w) mit $v \notin R_{v_0}$ werden nie betrachtet. □

Beobachtung 1:

Der Zeitaufwand für die Bearbeitung eines Aufrufs dfs(v) ist
(ohne die ausgelösten rekursiven Aufrufe) $O(1) + O(\text{outdeg}(v))$.

Der **Zeitaufwand** für den gesamten Aufruf dfs(v_0) ist **linear**, nämlich

$$O\Big(\sum_{v \in R_{v_0}} (1 + \text{outdeg}(v))\Big) = O(|E_{v_0}|).$$

(Beachte: $|R_{v_0}| \leq |E_{v_0}| + 1$.)

Der Zeitaufwand für die Initialisierung ist $O(n)$. – Also:

Mit Tiefensuche in G von v_0 aus lassen sich in Zeit $O(|V| + |E_{v_0}|)$
die Mengen R_{v_0} und E_{v_0} ermitteln.

Definition 8.1.2

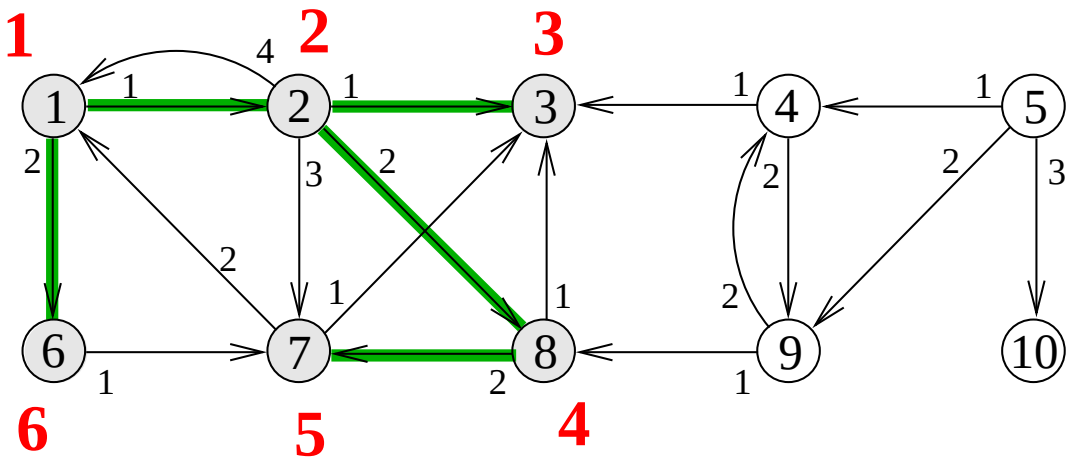
Die (**reflexive** und) **transitive Hülle** eines Digraphen G ist der Digraph $\mathbf{TH}(G) := (V, E^*)$ mit Kantenmenge

$$E^* = \{(v, w) \mid v, w \in V, v \rightsquigarrow_G w\} = \bigcup_{v \in V} (\{v\} \times R_v).$$

Tiefensuche in G, gestartet von jedem Knoten v nacheinander, ermittelt alle Mengen R_v, $v \in V$, also $\mathbf{TH}(G) = (V, E^*)$, in Zeit

$$O\Big(\sum_{v \in V} |E_{R_v}|\Big) = O(|E^*|) = O(|V| \cdot |E|).$$

(Nach jedem Aufruf „dfs(v)“ wird mit Hilfe von R_v in Zeit $O(|R_v|)$ das `status`-Array auf „**neu**“ zurückgesetzt.)

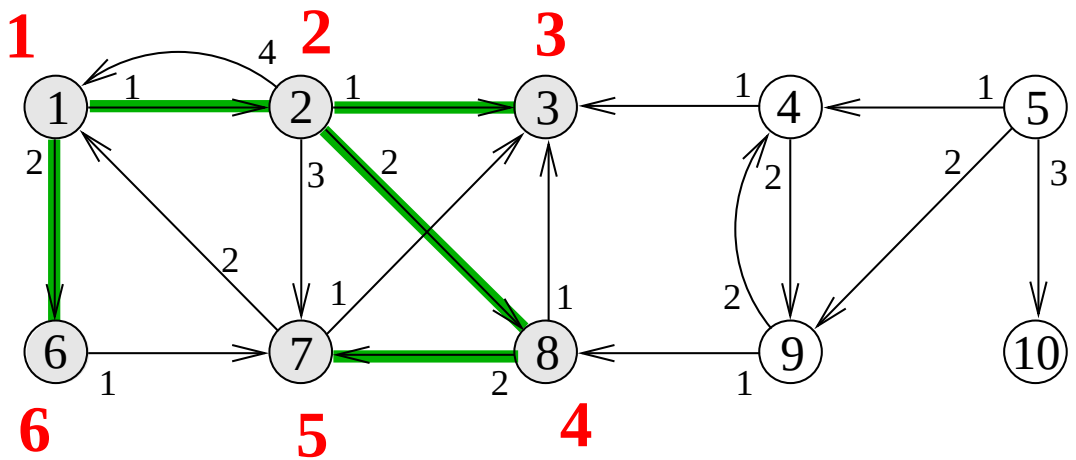


Beobachtung 2:

Jeder Knoten $v \in R_{v_0} - \{v_0\}$ wird von einem **eindeutig bestimmten** Knoten w aus „entdeckt“ (Inspektion der Kante (w, v) zeigt v als **neu**);
dieses w nennen wir $p(v)$, den **Vorgänger** von v.

Kanten von $p(v)$ zu v: „Entdeckungsrichtung“.

Man beachte: Aufruf „dfs(v)“ **beginnt nach** und **endet vor** Aufruf „dfs$(p(v))$“.

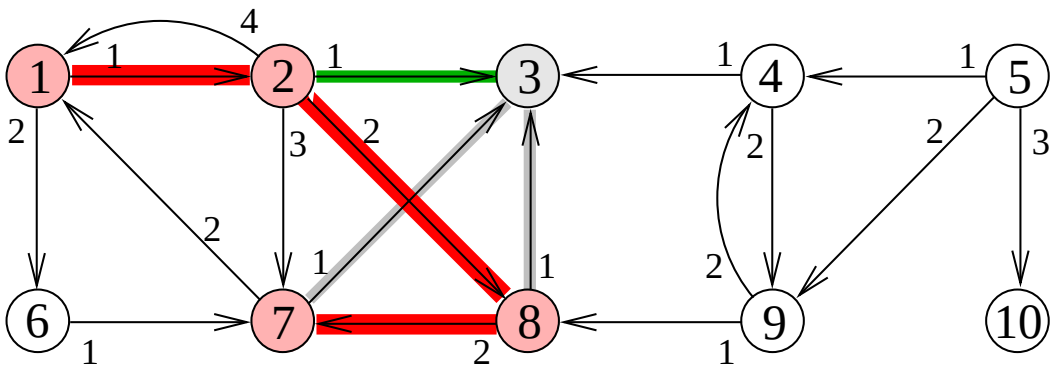


Die Kanten $(p(v), v)$ ($\widehat{=}$ direkter Aufruf dfs(v) aus dfs$(p(v))$) bilden einen **(gerichteten)** **Baum** mit Wurzel v_0 und Knotenmenge R_{v_0}, den

Tiefensuch-Baum/DFS-Baum $T_{\text{dfs}}(v_0)$.

Ein **Weg** im DFS-Baum entspricht einer indirekten Aufrufbeziehung.

Die DFS-Nummern **(rot)** zählen die Knoten in $T_{\text{dfs}}(v_0)$ in **Präorder**-Reihenfolge auf.
Die Anordnung der Kinder eines Knotens wird durch die Anordnung der Nachfolger in den Adjazenzlisten bestimmt.



Zu jedem Zeitpunkt bilden die **aktiven** Knoten einen Weg (den **„roten Weg“**) im DFS-Baum.

Startpunkt ist v_0, Endpunkt ist der Knoten v, in dessen „dfs(v)“-Aufruf eben gearbeitet wird. (Hier: Knoten 7.)

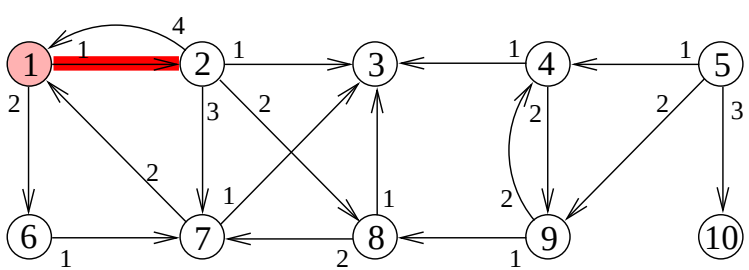
Die Kanten des roten Weges sind Kanten im DFS-Baum.

Wie entscheidet sich, ob w in $T_{\text{dfs}}(v_0)$ Nachfahr von v ist? D.h.: Wie kann man sehen, ob dfs(w) aufgerufen wird, während v **aktiv** ist?

Satz 8.1.3 („Satz vom weißen Weg“)

w ist ein Nachfahr von v im Tiefensuchbaum $T_{\text{dfs}}(v_0)$ $\Leftrightarrow$

in dem Moment, in dem dfs(v) aufgerufen wird, existiert ein Weg von v nach w aus Knoten, die alle **neu** („weiß“) sind.



Beispiel: dfs(2) startet.

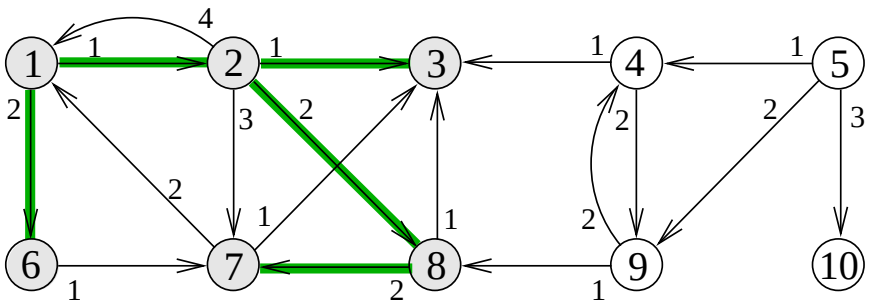
Knoten 3, 7, 8 sind auf „weißen Wegen“ von 2 aus erreichbar

$\Rightarrow$ diese Knoten sind im DFS-Baum Nachfahren von 2.

Knoten 6 ist von 2 aus erreichbar, aber nicht auf einem „weißen Weg“

$\Rightarrow$ Knoten 6 ist im DFS-Baum nicht Nachfahr von 2.

Beweis von Satz 8.1.3: „⇒“:



Sei w in $T_{\text{dfs}}(v_0)$ Nachfahr von v.

Betrachte den Weg, der in $T_{\text{dfs}}(v_0)$ von v nach w führt: $(v = u_0, u_1, u_2, \ldots, u_t = w)$.

(Beispiel im Bild: Weg $(2, 8, 7)$.)

Nach Beobachtung 2 wird erst dfs(v) aufgerufen, dann dfs(u_1), dann dfs(u_2), usw.;

also sind in dem Moment, in dem der Aufruf dfs(v) erfolgt,
alle Knoten $v = u_0, u_1, u_2, \ldots, u_t = w$ noch **neu** .

Beweis von Satz 8.1.3: „$\Leftarrow$“:

Sei $(v = u_0, u_1, u_2, \ldots, u_t = w)$, mit $t > 0$, ein Weg aus Knoten, die im Moment des Aufrufs dfs(v) alle **neu** sind.

Durch Induktion über i zeigen wir, dass für jedes i, $0 \leq i \leq t$, der Aufruf dfs(u_i) erfolgt, bevor dfs(v) endet.

I.A.: Die Beh. ist für $i = 0$ klar.

I.Schritt: Sei nun $1 \leq i \leq t$.

Nach I.V. beginnt dfs(u_{i-1}), bevor dfs(v) endet.

Während des Aufrufs dfs(u_{i-1}) wird die Kante (u_{i-1}, u_i) inspiziert.

1. Fall: In diesem Moment ist u_i noch **neu**. Dann erfolgt der Aufruf dfs(u_i) jetzt.

2. Fall: In diesem Moment ist u_i **aktiv** oder **fertig**. Dann ist der Aufruf dfs(u_i) schon vorher passiert.

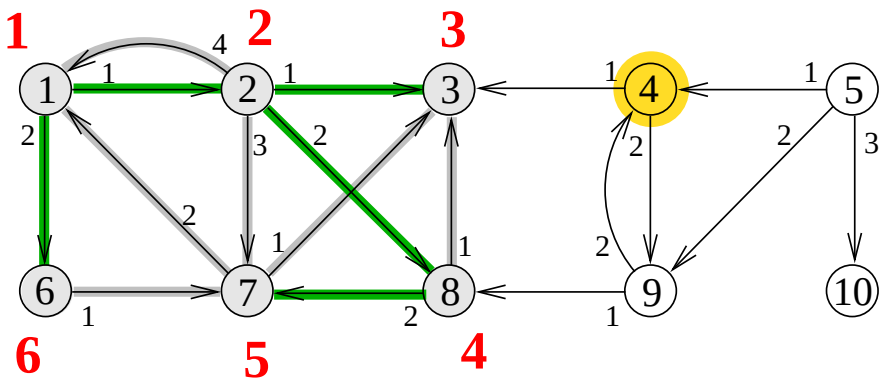
Demnach wird dfs(w) (direkt oder indirekt) aus dfs(v) aufgerufen,
also ist w Nachfahr von v im DFS-Baum. $\square$

Globale Tiefensuche in Digraphen: Alle Knoten und Kanten werden besucht.

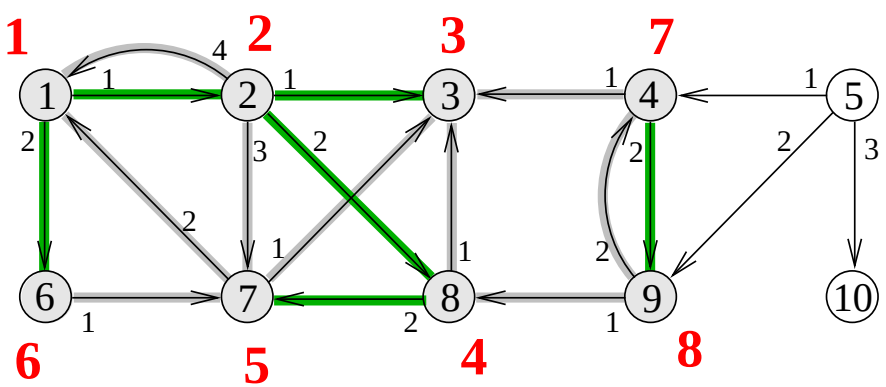
Algorithmus DFS(G) // Tiefensuche in $G = (V, E)$ mit $V = \{1, \ldots, n\}$

(1) `dfs_count` $\leftarrow 0$;
(2) **for** `v` **from** 1 **to** n **do** `status[v]` $\leftarrow$ **neu** ;
(3) **for** `v` **from** 1 **to** n **do**
(4) **if** `status[v]` $=$ **neu** **then**
(5) dfs(`v`); // starte Tiefensuche von `v` aus

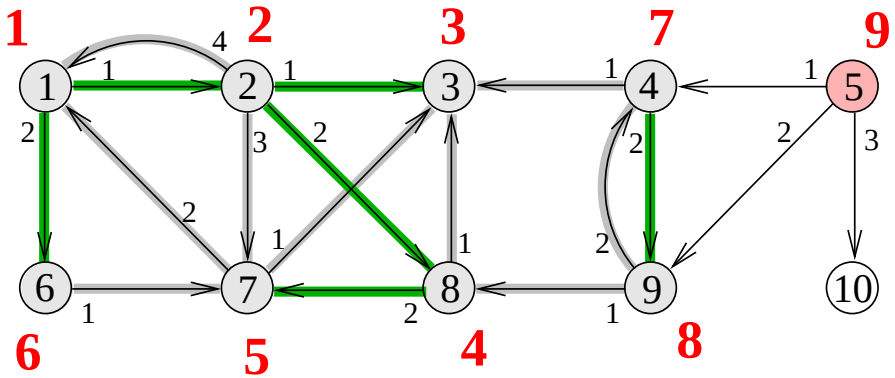
Zu Zeilen (2), (4): Die `status`$[v]$-Werte werden von den dfs-Aufrufen in Zeile (5) verändert;
sie haben also beim Wieder-Lesen in der zweiten Schleife nicht unbedingt immer noch den Initialisierungs-Wert **neu**.



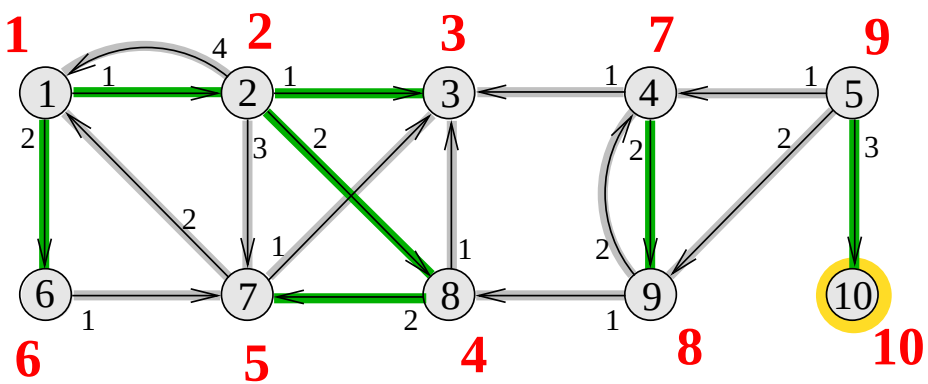
Knoten 4 ist **neu** , also starte dfs(4).



dfs(4) endet.



dfs(5) startet.



Letzter Test: Ist Knoten 10 **neu** ? Nein, also fertig.

DFS(G) erzwingt, dass für **jeden** Knoten v in V irgendwann einmal dfs(v) aufgerufen wird. (Dass es höchstens einen Aufruf für v gibt, sieht man wie vorher.)

Daher werden auch alle Kanten, die aus v herausführen, angesehen.

Gesamt-(Zeit-)Aufwand: $O(1 + \text{outdeg}(v))$ für Knoten v. Insgesamt:

$$O\Big(\sum_{v \in V} (1 + \text{outdeg}(v))\Big) = O(|V| + |E|) = O(n + m).$$

Satz 8.1.4

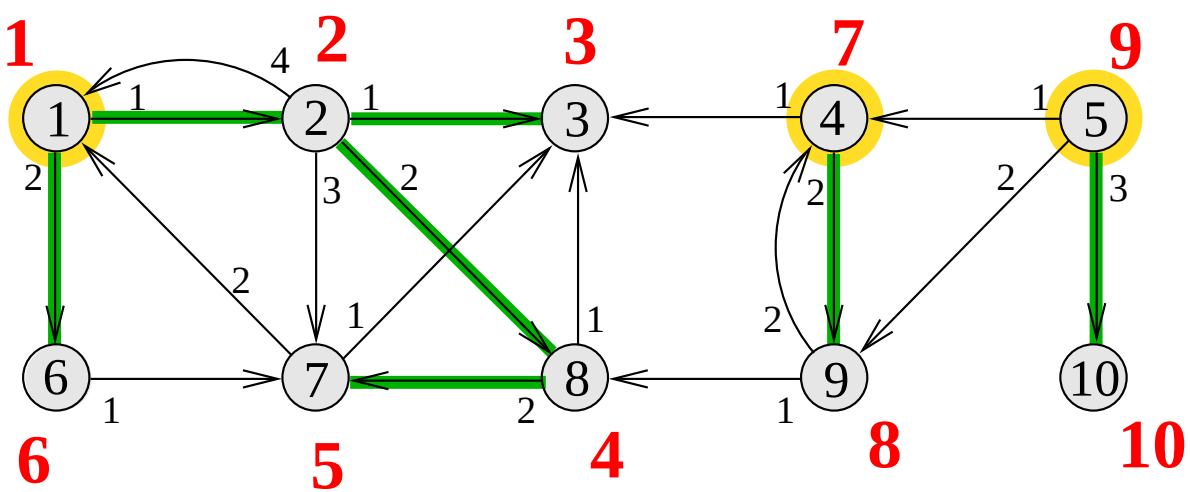
Der Zeitaufwand von DFS(G) ist $O(|V| + |E|)$.

(Sogar: $\Theta(|V| + |E|)$, da jeder Knoten und jede Kante betrachtet wird.)

Bemerkung:
Bei DFS entstehen mehrere Bäume, die zusammen den **„Tiefensuchwald“** bilden.

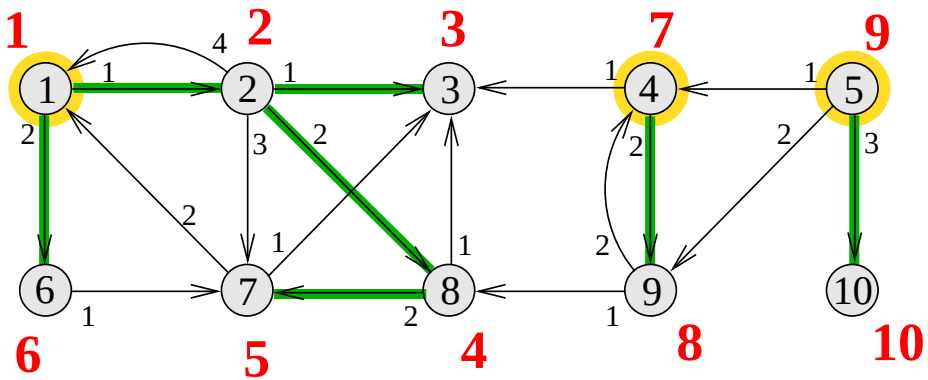
(Kante (w, v) entspricht dem Aufruf dfs(v) direkt aus dfs(w).)

Vorsicht: Wenn ein Baum im Tiefensuchwald Wurzel v hat, so ist nicht gesagt, dass dieser Baum alle von v aus erreichbaren Knoten enthält. (Dies gilt nur für den **ersten** der Bäume.)



Beispiel: Es gilt $5 \rightsquigarrow 1$, aber 1 ist nicht im Baum mit Wurzel 5.

Satz 8.1.3, der „Satz vom weißen Weg“, gilt auch für den Tiefensuchwald, mit identischem Beweis.



Für $v \in V = \{1, \ldots, n\}$: $\boldsymbol{W_v} := \{u \in V \mid \exists w \leq v : w \rightsquigarrow u\}$.
Beispiel: $W_1 = W_2 = W_3 = \{1, 2, 3, 6, 7, 8\}$, $W_4 = \{1, 2, 3, 4, 6, 7, 8, 9\}$, also $4 \notin W_3$ und $W_4 - W_3 = \{4, 9\}$.

Satz 8.1.5 (vgl. Beh. 7.3.5 zu BFS-Bäumen)
(a) v wird Wurzel eines Baums im Tiefensuch-Wald $\Leftrightarrow v \notin W_{v-1}$.

(b) Wenn v Wurzel ist, dann hat der Baum mit Wurzel v Knotenmenge $W_v - W_{v-1}$ (erreichbar von v aus, aber von keinem vorherigen Knoten). (Beweis auf Druckfolien.)

Beweis: (a) „$\Leftarrow$“: Sei $v \notin W_{v-1}$. v kann nicht in einem Baum mit einer Wurzel $w < v$ sitzen, weil es überhaupt keinen Weg von w nach v gibt.

Also ist v selbst eine Wurzel.

„$\Rightarrow$“: Sei $v \in W_{v-1}$.
Betrachte einen Weg $(w = v_0, v_1, \ldots, v_t = v)$ mit $w < v$.

Der Aufruf dfs(w) erfolgt spätestens, wenn w in der äußeren Schleife von DFS(G) getestet wird, also bevor v getestet wird.

Betrachte den Knoten v_i auf dem Weg, dessen Aufruf dfs(v_i) **zuerst** erfolgt.

Wenn dieser Aufruf erfolgt, sind $v_i, v_{i+1}, \ldots, v_t$ noch neu.

Nach Satz 8.1.3, dem „Satz vom weißen Weg“, wird v Nachfahr von v_i, also ist v keine Wurzel.

(b) Sei v Wurzel und sei $v = v_0, v_1, \ldots, v_t = w$ ein Weg von v zu $w \in R_v - W_{v-1}$. Dann sind offenbar alle Knoten v_i auf dem Weg in $R_v - W_{v-1}$.

Wenn also der Aufruf dfs(v) erfolgt, sind alle Knoten $v = v_0, v_1, \ldots, v_t = w$ neu.
Nach dem Satz vom weißen Weg wird w Nachfahr von v. □

8.2 Volle Tiefensuche in Digraphen

Wir bauen dfs(v) und DFS(G) aus, so dass die Kanten (v, w) von G in vier Klassen $\boldsymbol{T}$, $\boldsymbol{B}$, $\boldsymbol{F}$, $\boldsymbol{C}$ eingeteilt werden:

- $\boldsymbol{T}$ **(„Tree“):** **Baumkanten**, die Kanten des Tiefensuchwaldes.

  Kriterium: Wenn dfs(v) die Kante (v, w) betrachtet, ist w noch **neu** .

- $\boldsymbol{B}$ **(„Back“):** **Rückwärtskanten**.
  (v, w) ist Rückwärtskante, wenn w Vorfahr von v im Tiefensuchwald ist.

  Kriterium: Wenn dfs(v) die Kante (v, w) betrachtet, ist w **aktiv** .

- F: **Vorwärtskanten** („Forward“).
  (v, w) ist Vorwärtskante, wenn w Nachfahr von v im Tiefensuchwald ist, und (v, w) keine Baumkante ist.

  Kriterium: Wenn dfs(v) die Kante (v, w) betrachtet, ist w **fertig** und *dfs_num*$(v) <$ *dfs_num*(w).

- C: **Querkanten** („Cross“).
  (v, w) ist Querkante, wenn der gesamte Unterbaum von w schon fertig abgearbeitet ist, wenn dfs(v) aufgerufen wird.

  Kriterium: Wenn dfs(v) die Kante (v, w) betrachtet, ist w **fertig** und *dfs_num*$(v) >$ *dfs_num*(w).

Außerdem nummerieren wir die Knoten $v \in V$ auch **in der Reihenfolge** durch, in der die dfs(v)-Aufrufe **beendet** werden:

f(in)-Nummern $\texttt{f_num}(v)$, $v \in \{1, \ldots, n\}$, mit Wertebereich $\{1, 2, \ldots, |V|\}$.

Prozedur dfs(v) // „volle“ Tiefensuche in v, nur für v **neu** erlaubt

(1) `dfs_count++;`

(2) `dfs_num[`v`]` $\leftarrow$ `dfs_count;`

(3) `status[`v`]` $\leftarrow$ **aktiv** ;

(4) **dfs-visit**(v); // Aktion an v bei Entdeckung

(5) für jeden Nachfolger `w` von v (Adjazenzliste!) tue:

(6) **1. Fall:** `status[w]` $=$ **neu** : $T \leftarrow T \cup \{(v, \texttt{w})\}$; dfs(`w`);

(7) **2. Fall:** `status[w]` $=$ **aktiv** : $B \leftarrow B \cup \{(v, \texttt{w})\}$;

(8) **3. Fall:** `status[w]` $=$ **fertig** $\wedge$ `dfs_num[`v`]` $<$ `dfs_num[w]`:
$F \leftarrow F \cup \{(v, \texttt{w})\}$;

(9) **4. Fall:** `status[w]` $=$ **fertig** $\wedge$ `dfs_num[`v`]` $>$ `dfs_num[w]`:
$C \leftarrow C \cup \{(v, \texttt{w})\}$;

(10) `f_count++;`

(11) `f_num[`v`]` $\leftarrow$ `f_count;`

(12) **fin-visit**(v); // Aktion an v bei Abschluss

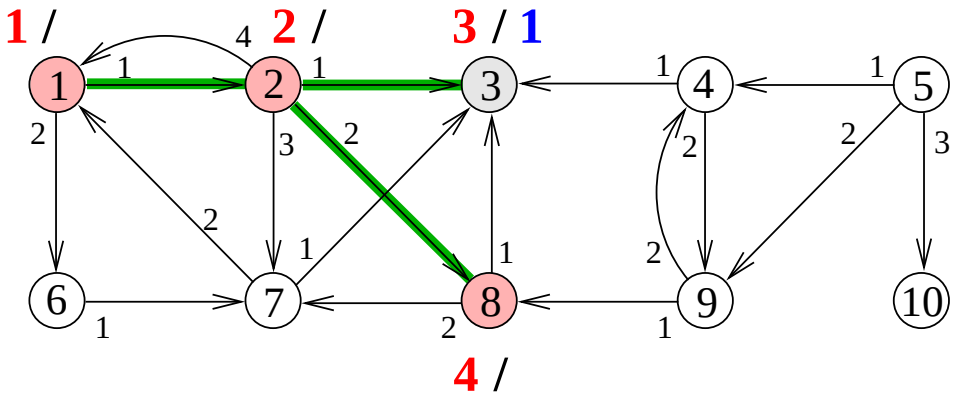
(13) `status[`v`]` $\leftarrow$ **fertig** .

Initialisierung: $\texttt{dfs_count} \leftarrow 0$; $\texttt{f_count} \leftarrow 0$.
Alle Knoten sind **neu** ; Mengen T, B, F, C sind leer.
Tiefensuche von v_0 aus („dfs(v_0)“) entdeckt R_{v_0}.

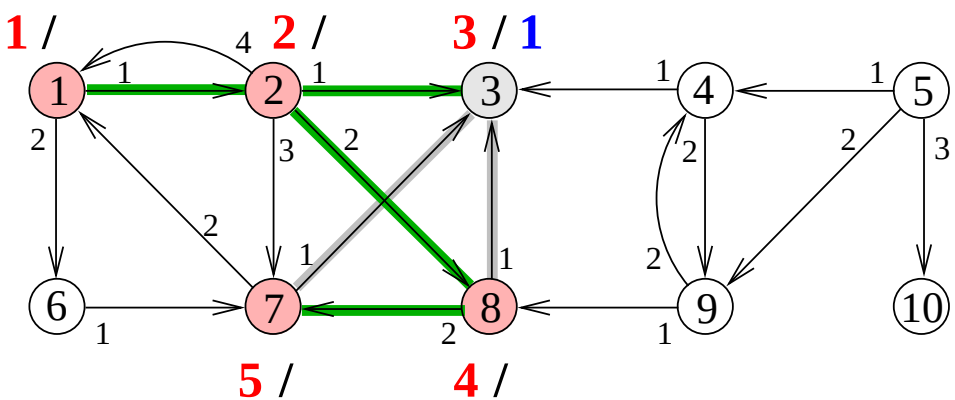
Tiefensuche **für den ganzen Graphen**: DFS(G) wie vorher, nur mit der voll ausgebauten dfs-Prozedur, und der Initialisierung von $\texttt{f_count}$ und T, B, F, C.

Algorithmus DFS(G) // Tiefensuche in $G = (V, E)$, voll ausgebaut

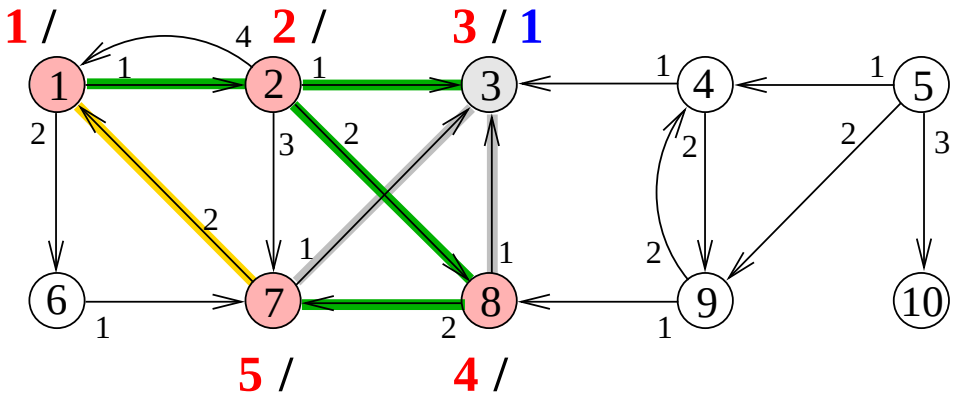
```
(1)  dfs_count ← 0;
(2)  f_count ← 0;
(3)  T ← ∅; B ← ∅; F ← ∅; C ← ∅;
(3)  for v from 1 to n do status[v] ← neu ;
(4)  for v from 1 to n do
(5)    if status[v] = neu then
(6)      dfs(v); // starte (volle) Tiefensuche von v aus
```



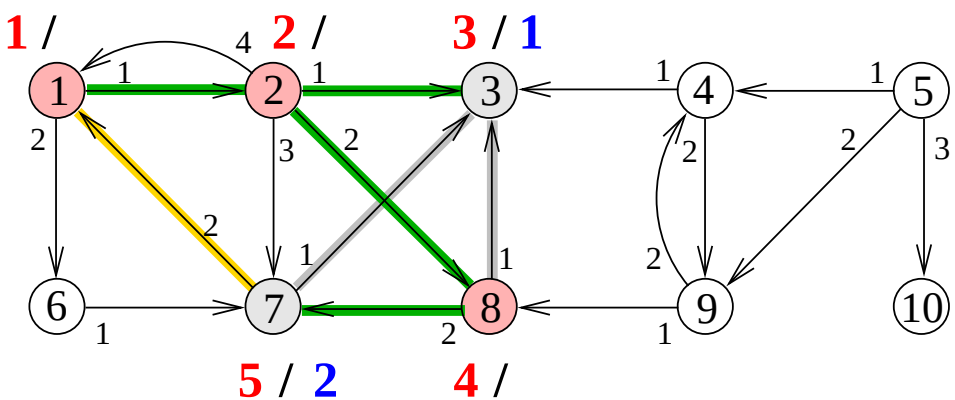
dfs(8) startet, dfs-Nummer $\mathbf{4}$,
$(2, 8)$ wird Baumkante.



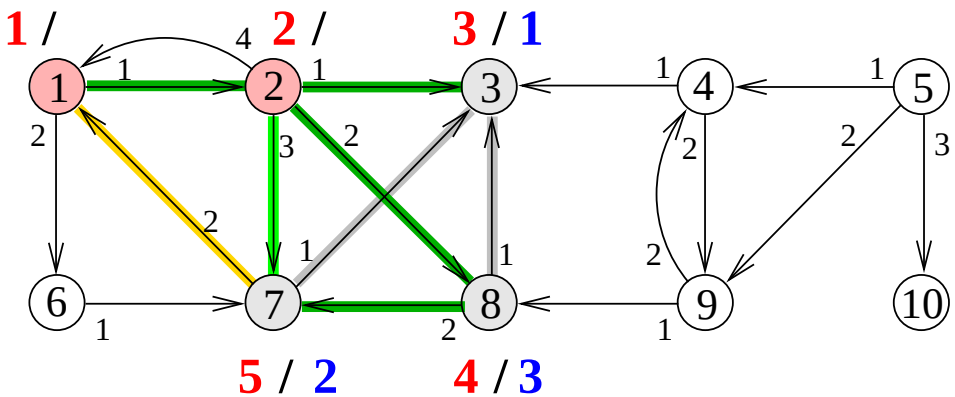
$(7, 3)$ wird **Querkante**,
weil 3 **fertig** ist. und
$\mathbf{5} > \mathbf{3}$ gilt.



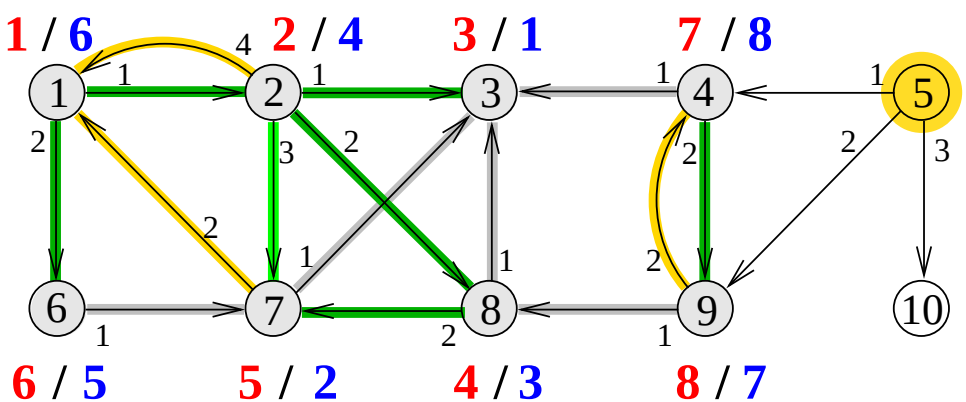
$(7, 1)$ wird **Rückwärtskante**, weil 1 **aktiv** ist.



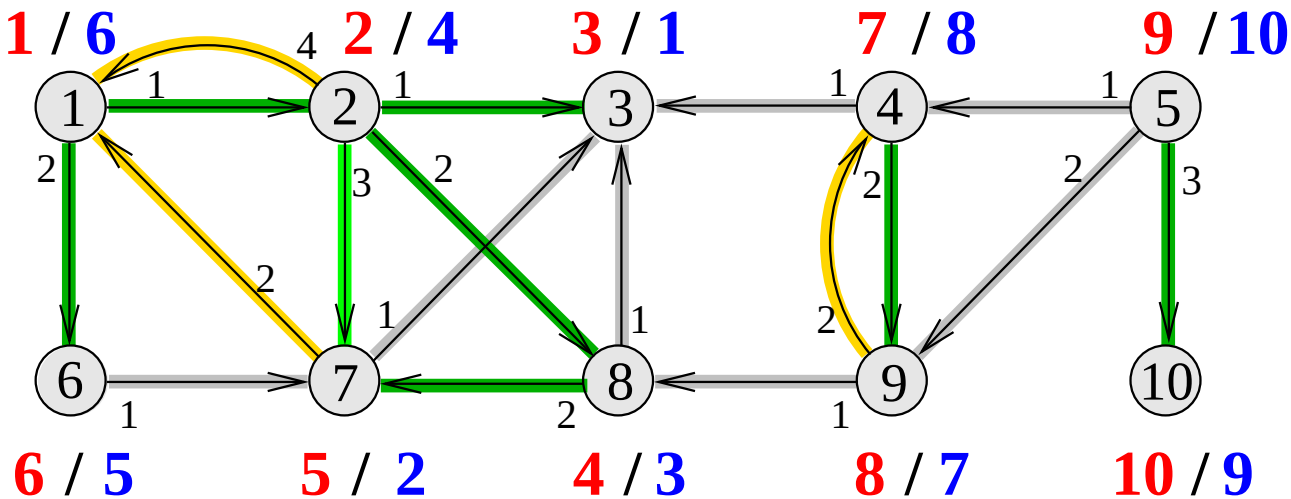
dfs(7) endet, f-Nummer **2**.



$(2, 7)$ wird **Vorwärtskante**,
weil 7 **fertig** ist und $2 < 5$ gilt.



5 wird getestet,
ist noch **neu**,
dfs(5) wird gestartet.



Klar: Alle Aussagen für die einfache dfs-Prozedur bleiben gültig.

Beobachtung 1: Jede Kante (v, w) wird genau einmal betrachtet und daher genau in eine der Mengen T, B, F und C eingeordnet.

Beobachtung 2: Die **dfs-Nummern** entsprechen einem **Präorder**-Durchlauf, die **f-Nummern** einem **Postorder**-Durchlauf durch die Bäume des Tiefensuch-Waldes.

Beobachtung 3: Folgende Aussagen sind äquivalent:

(a) dfs(w) beginnt nach dfs(v) und endet vor dfs(v)
(d. h. dfs(w) wird (direkt oder indirekt) von dfs(v) aus aufgerufen);

(b) $\texttt{dfs_num}[v] < \texttt{dfs_num}[w] \wedge \texttt{f_num}[v] > \texttt{f_num}[w]$;

(c) v ist Vorfahr von w im Tiefensuch-Wald.

Konsequenz: Die **Vorfahr-Nachfahr-Relation** im DFS-Wald lässt sich in Zeit $O(1)$ testen, wenn die dfs- und die f-Nummern aller Knoten bekannt sind.

Satz 8.2.1

Der Algorithmus DFS(G) klassifiziert die Kanten korrekt.

Beweis: Sei (v, w) eine beliebige Kante in G.

1. Fall: dfs(w) wird direkt aus dfs(v) aufgerufen.

Dann ist (v, w) **Baumkante** und wird richtig als T klassifiziert.

2. Fall: Die Untersuchung der Kante (v, w) in dfs(v) ergibt, dass w **aktiv** ist.

Dann liegt w auf dem **roten Weg** von der Wurzel des Tiefensuchbaums zu v, ist also Vorfahr von v (oder gleich v – Schleifen sind ja nicht verboten).
Daher ist (v, w) **Rückwärtskante** und wird richtig als B klassifiziert.

3. Fall: Die Untersuchung der Kante (v, w) in dfs(v) ergibt, dass w **fertig** ist und $\texttt{dfs_num}[v] < \texttt{dfs_num}[w]$ gilt.

Weil dfs(v) noch aktiv ist, wird $\texttt{f_num}[v] > \texttt{f_num}[w]$.
Dann ist w nach Beobachtung 3 Nachfahr von v im Tiefensuchwald,
also ist (v, w) **Vorwärtskante** und wird richtig als $\boldsymbol{F}$ klassifiziert.

4. Fall: Die Untersuchung der Kante (v, w) in dfs(v) ergibt, dass w **fertig** ist und $\texttt{dfs_num}[v] > \texttt{dfs_num}[w]$ gilt.

Weil dfs(v) noch aktiv ist, wird $\texttt{f_num}[v] > \texttt{f_num}[w]$.
Nach Beobachtung 3 ist w weder Nachfahr noch Vorfahr von v im Tiefensuchwald.
(Bei der Auffassung der dfs-Aufrufe als Präorder- oder Postorderdurchlauf im Tiefensuchwald ist die Bearbeitung des Unterbaums unter w komplett abgeschlossen, bevor v erreicht wird.)
Daher ist (v, w) **Querkante** und wird richtig als $\boldsymbol{C}$ klassifiziert.

8.3 Tiefensuche in ungerichteten Graphen

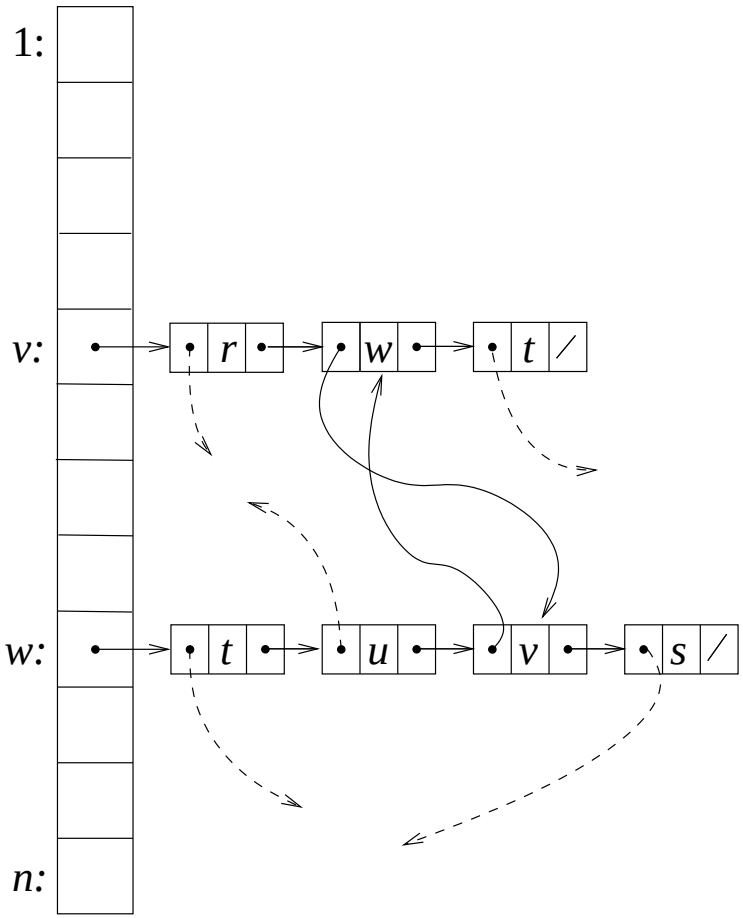
Einfacher als Tiefensuche in gerichteten Graphen.

Zweck:

- Zusammenhangskomponenten finden
- Spannbaum für jede Zusammenhangskomponente finden
- Kreisfreiheitstest

(Diese einfachen Aufgaben wären auch von Breitensuche zu erledigen.)

Komplexere Erweiterungen, z. B. „Zweifach-Zusammenhangskomponenten“, **erfordern** DFS. (Literatur z. B.: [Sedgewick, Algorithms, Part 5: Graph Algorithms].)



Adjazenzlistendarstellung mit Querverweisen.

Erinnerung: **Adjazenzlistendarstellung** mit Querverweisen.

Knotenarray: `nodes[1..`n`]`.

Zu Knoten v gibt es eine lineare Liste L_v mit einem Eintrag für jede Kante (v, w).

Von Eintrag für Kante (v, w) in L_v führt ein Verweis auf die „Gegenkante“ (w, v) in L_w (und natürlich auch umgekehrt).

Wir können an Kanten Markierungen anbringen.

DFS(G): Anfangs markieren wir beide Richtungen aller Kanten mit „neu“.

Wir stellen sicher, dass nach Benutzung der Kante (v, w) in Richtung von v nach w die umgekehrte Richtung nicht mehr benutzt wird, indem wir (w, v) mit **„gesehen“** markieren.

Dank der Gegenkanten ist dies in Zeit $O(1)$ möglich.

Klassifizierung in T-Kanten und B-Kanten.

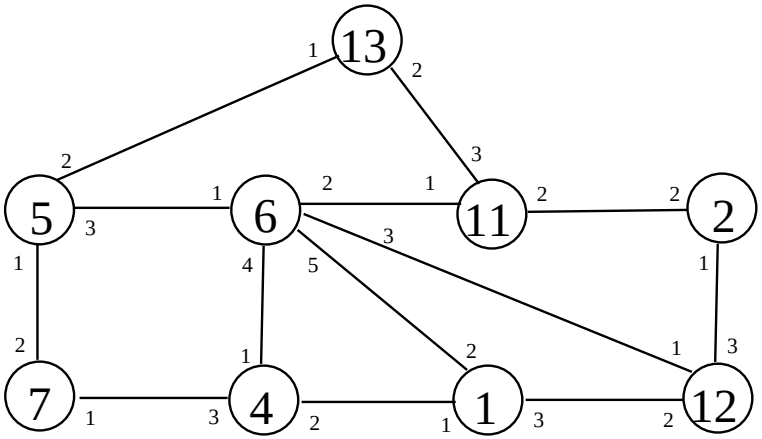
Algorithmus udfs(v) // Tiefensuche von v aus, rekursiv
// nur für v mit `status[`v`]` $=$ **neu** aufrufen

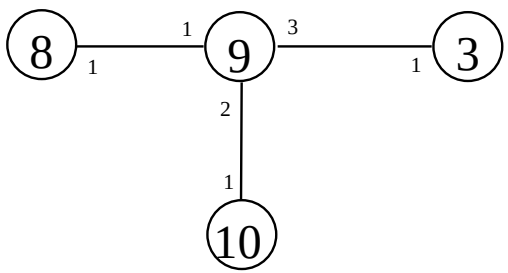
(1) `dfs_count++;`
(2) `dfs_num[`v`]` $\leftarrow$ `dfs_count;`
(3) dfs-visit(v); // Aktion an v bei Erstbesuch
(4) `status[`v`]` $\leftarrow$ **aktiv** ;
(5) für jede **„neue“** Kante $(v, \texttt{w})$ (Adjazenzliste!) tue:
(6) setze Gegenkante $(\texttt{w}, v)$ auf **„gesehen“**;
(7) **1. Fall:** `status[w]` $=$ **neu**: $T \leftarrow T \cup \{(v, \texttt{w})\}$; **udfs**$(\texttt{w})$;
(8) **2. Fall:** `status[w]` $=$ **aktiv**: $B \leftarrow B \cup \{(v, \texttt{w})\}$;
(9) `f_count++;`
(10) `f_num[`v`]` $\leftarrow$ `f_count;`
(11) fin-visit(v); // Aktion an v bei letztem Besuch
(12) `status[`v`]` $\leftarrow$ **fertig**.

Globale Tiefensuche in (ungerichtetem) Graphen G:

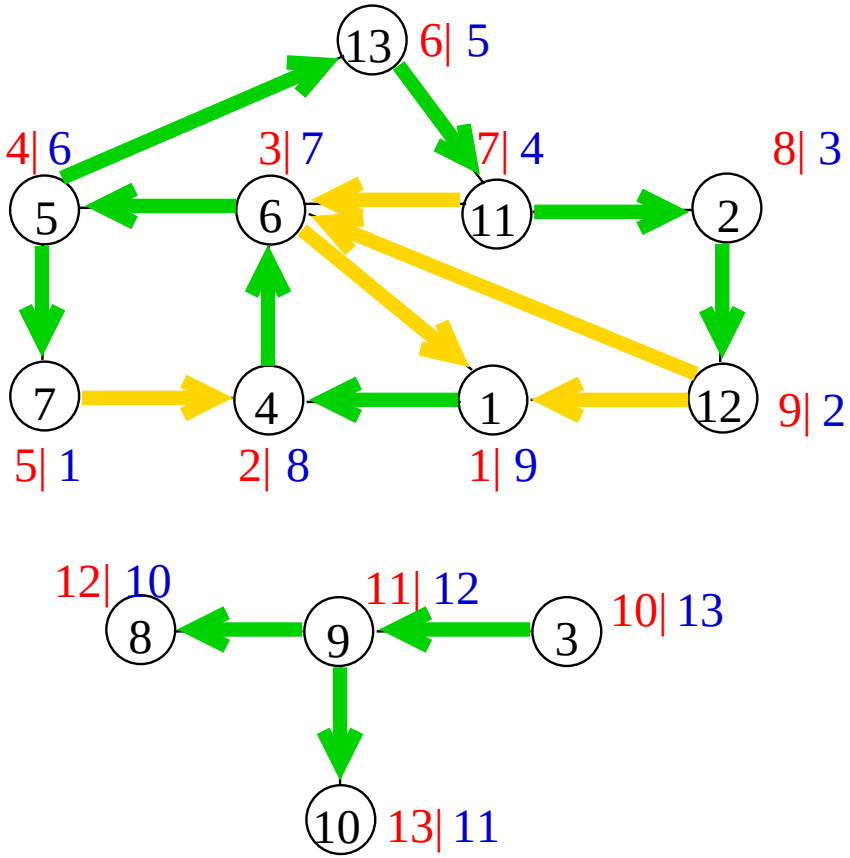
Algorithmus UDFS(G) // Tiefensuche in $G = (V, E)$

(1) `dfs_count` $\leftarrow 0$;
(2) `f_count` $\leftarrow 0$;
(3) **for** `v` **from** 1 **to** n **do** `status[v]` $\leftarrow$ **neu**;
(4) $T \leftarrow \emptyset$; $B \leftarrow \emptyset$;
(5) **for** `v` **from** 1 **to** n **do**
(6) **if** `status[v]` $=$ **neu** **then**
(7) **udfs**(`v`); // starte Tiefensuche von `v` aus





Ungerichteter Graph.



Ungerichteter Graph mit Baum- und Rückwärtskanten sowie dfs- und f-Nummern.

Satz 8.3.1 **udfs**(v_0) werde aufgerufen. Dann gilt:

(a) **udfs**(v_0) entdeckt genau die Knoten in der Zusammenhangskomponente von v_0.

(b) Die Kanten (v, w), für die **udfs**(w) unmittelbar aus **udfs**(v) aufgerufen wird, bilden einen gerichteten Baum mit Wurzel v_0 **(„Tiefensuchbaum“)**.

(c) (Satz vom weißen Weg) u ist im DFS-Baum ein Nachfahr von v genau dann wenn zum Zeitpunkt des Aufrufs **udfs**(v) ein Weg von v nach u existiert, der nur neue (weiße) Knoten enthält.

Rechenzeit (mit Initialisierung): $O(|V| + |E_{v_0}|)$, wo E_{v_0} die Menge der Kanten in der Zusammenhangskomponente von v_0 ist.
Beweis: Ähnlich zum gerichteten Fall.

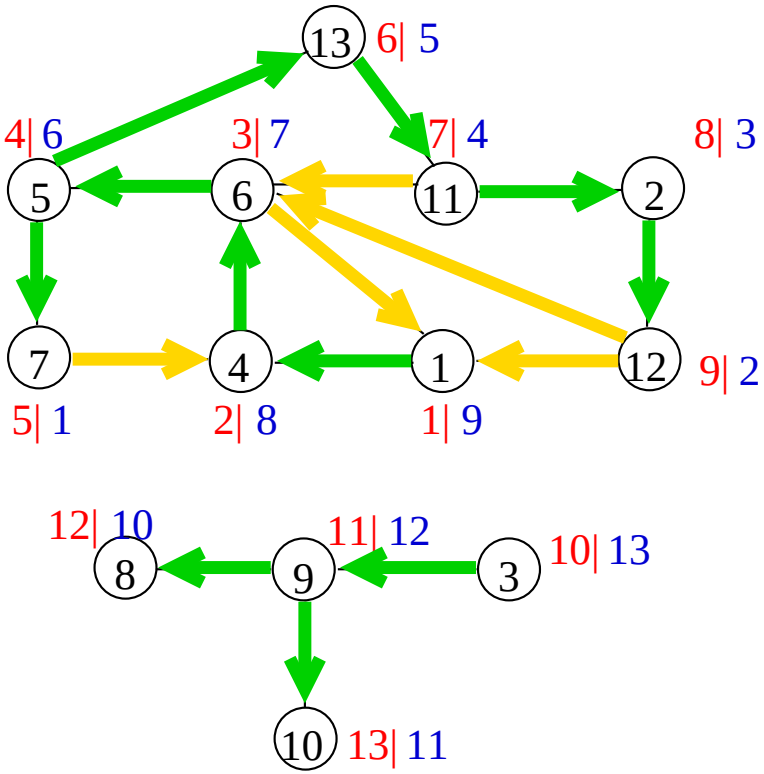
Satz 8.3.2

UDFS(G) werde aufgerufen. Dann gilt:

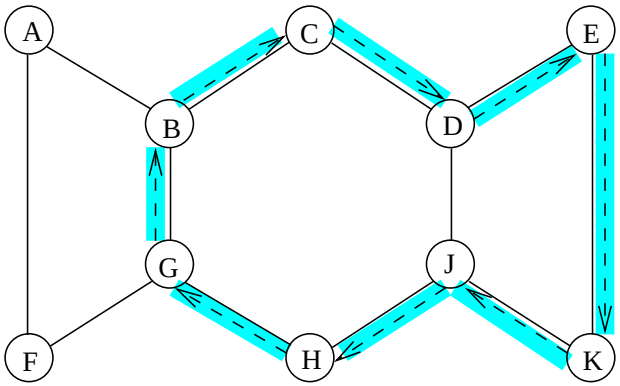
(a) Die T-Kanten bilden eine Reihe von Bäumen (die Tiefensuch-Bäume).
Die Knotenmengen dieser Bäume sind die Zusammenhangskomponenten von G.

(b) Die Wurzeln der Bäume sind die jeweils ersten Knoten einer Zusammenhangskomponente in der Reihenfolge, die in Zeile (2) benutzt wird.

Gesamtrechenzeit: $O(|V| + |E|)$: linear!



Beobachtung: Jeder Knoten wird besucht; jede Kante wird in genau einer Richtung betrachtet und als Baumkante (T) oder Rückwärtskante (B) klassifiziert; die jeweiligen Gegenkanten werden nicht betrachtet (weil sie auf „gesehen“ gesetzt werden).



Einfache Kreise: (B,C,D,E,K,J,H,G,B) und (K,J,D,E,K)

(Erinnerung an Definition 7.1.14)

Ein Kantenzug $(v_0, v_1, \ldots, v_k)$ mit $k \geq 3$ in einem (ungerichteten) Graphen G heißt ein **(einfacher) Kreis**, wenn $v_0 = v_k$ gilt und $v_0, v_1, \ldots, v_{k-1}$ verschieden sind.

Kreisfreiheitstest in Graphen und Finden eines Kreises, wenn es einen gibt.

Satz 8.3.3

Nach Ausführung von **UDFS**(G) gilt: $B \neq \emptyset \Leftrightarrow G$ enthält einen Kreis.

Beweis:

„$\Rightarrow$“: Wenn in Zeile (8) eine „neue“ Kante (v, w) gefunden wird, die zu einem aktiven (roten) Knoten w führt, dann bildet diese Kante mit dem Abschnitt des roten Weges von w nach v zusammen einen Kreis der Länge ≥ 3.
(Es kann nicht sein, dass (w, v) eine Baumkante ist, da sonst (v, w) den Status „**gesehen**“ hätte.)

In diesem Moment kann man auch leicht diesen Kreis ausgeben.

„$\Leftarrow$“: Wenn $B = \emptyset$ ist, dann wird jede Kante von G in einer Richtung Baumkante. Die Kanten der UDFS-Bäume bilden ungerichtete Bäume und enthalten alle Kanten von G, also kann G keinen Kreis haben. □

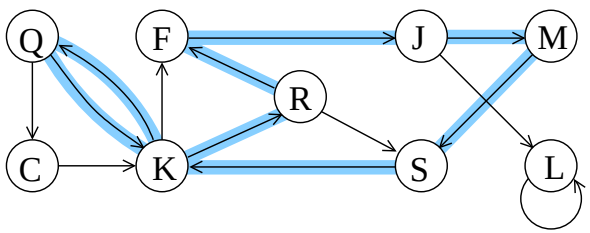
Bemerkung

UDFS auf einen ungerichteten Wald (Vereinigung disjunkter Bäume) angewendet gibt jeder Komponente eine Wurzel und richtet die Kanten von dieser Wurzel weg.

Dies ist eine sehr einfache und schnelle Methode (Linearzeit!), einen ungerichteten Wald zu „orientieren“.

(Dies ist natürlich auch mit Breitensuche zu realisieren.)

8.4 Kreisfreiheitstest und topologische Sortierung

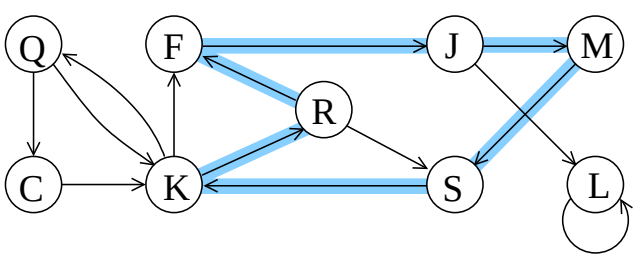


(Erinnerung an Definition 7.1.4)

(a) Ein Kantenzug $(v_0, v_1, \ldots, v_k)$ in einem Digraphen G heißt ein **Kreis** oder **Zyklus**, wenn $k \geq 1$ und $v_0 = v_k$.

Beispiel: $(\mathrm{Q}, \mathrm{K}, \mathrm{R}, \mathrm{F}, \mathrm{J}, \mathrm{M}, \mathrm{S}, \mathrm{K}, \mathrm{Q})$ ist ein Kreis.

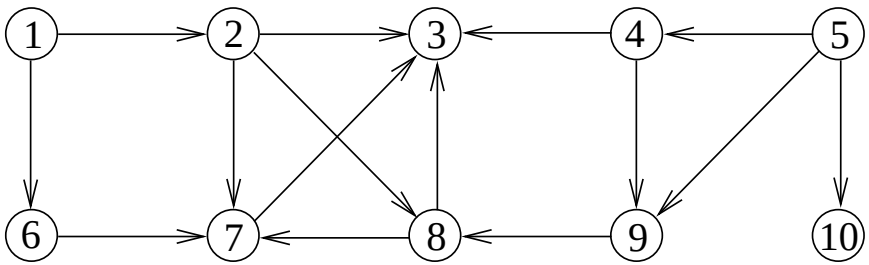
Kreise, die durch einen zyklischen Shift auseinander hervorgehen, werden als gleich betrachtet.



(b) Ein Kreis $(v_0, v_1, \ldots, v_{k-1}, v_0)$ heißt **einfach**, wenn $v_0, \ldots, v_{k-1}$ verschieden sind.

Beispiel: $(\mathrm{J}, \mathrm{M}, \mathrm{S}, \mathrm{K}, \mathrm{R}, \mathrm{F}, \mathrm{J})$ ist ein einfacher Kreis.

Wir wissen: Wenn G einen Kreis enthält, dann auch einen einfachen Kreis.



(Erinnerung an Definition 7.1.5)

Ein Digraph G heißt **kreisfrei** oder **azyklisch**, wenn es in G keinen Kreis gibt, sonst heißt G **zyklisch**.

Azyklische gerichtete Graphen: **„Directed Acyclic Graphs“**, daher **DAG**s.

Ob ein Digraph zyklisch ist oder nicht, kann man nach Durchführung einer Tiefensuche sofort ablesen.

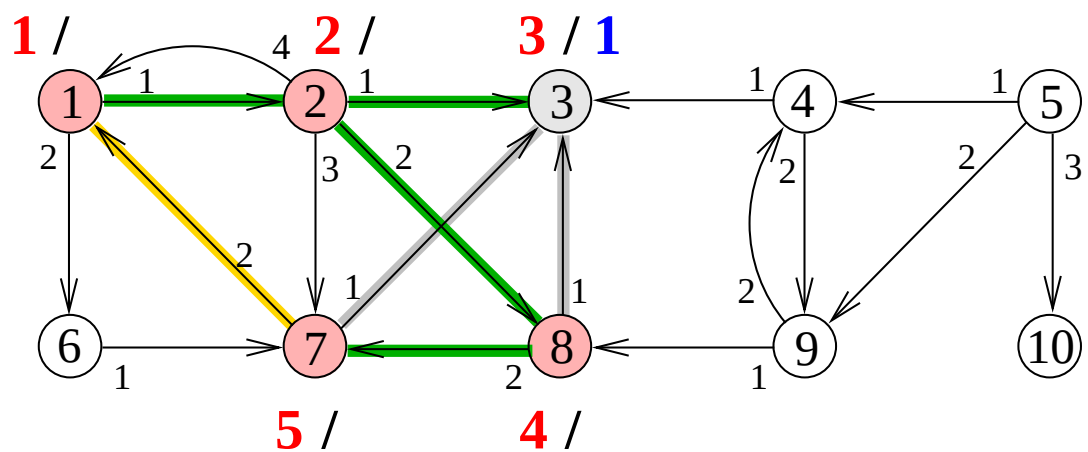
Satz 8.4.1

Nach Aufruf von DFS(G) gilt: G enthält einen Kreis $\Leftrightarrow$ $B \neq \emptyset$.

Beweis: „$\Leftarrow$“: Wenn DFS(G) eine B-Kante (v, w) findet, dann ist in diesem Moment v der vorderste Knoten des **roten Weges**, und w liegt irgendwo auf dem roten Weg. Also gibt es in G einen Weg von w nach v aus Baumkanten. Zusammen mit der Kante (v, w) ergibt sich ein Kreis.

Spezialfall: $v = w$, $t = 0$: **Schleife.** Dies wird als Kreis erkannt.

Beispiel:



Roter Weg: $(1, 2, 8, 7)$; Rückwärtskante $(7, 1)$ schließt den Kreis.

„⇒“: Nun sei $\boldsymbol{B = \emptyset}$.

Für eine Kante (v, w) gibt es dann nur noch drei Fälle:

1. Fall: (v, w) ist Baumkante. – Dann ist $\texttt{f_num}[v] > \texttt{f_num}[w]$.

2./3. Fall: (v, w) ist Vorwärtskante oder Querkante. – Dann ist w „fertig“, während an v noch gearbeitet wird, also ist wieder $\texttt{f_num}[v] > \texttt{f_num}[w]$.

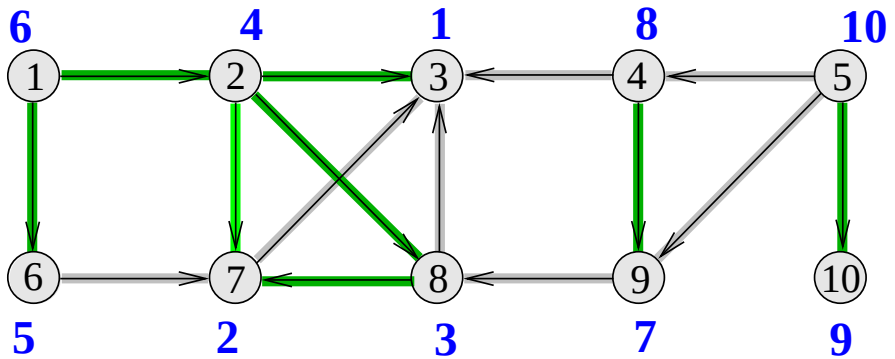
Für jede Kante (v, w) in E ist $\texttt{f_num}[v]$ größer als $\texttt{f_num}[w]$.

Wenn $(u_0, \ldots, u_t)$ ein Weg in G ist, so **nehmen** entlang dieses Weges **die f-Nummern strikt ab**.

Daher kann auf keinen Fall $t \geq 1$ und $u_0 = u_t$ sein.

Also enthält G keinen Kreis. □

Beispiel: (Azyklischer) Digraph G mit f-Nummern:



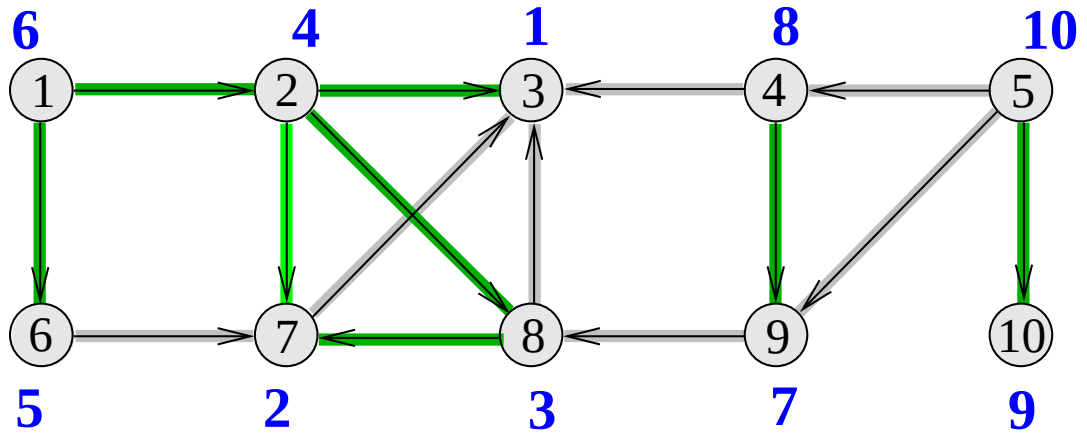
Wie im Beweis von Satz 8.4.1: f-Nummern fallen entlang jeder Kante!

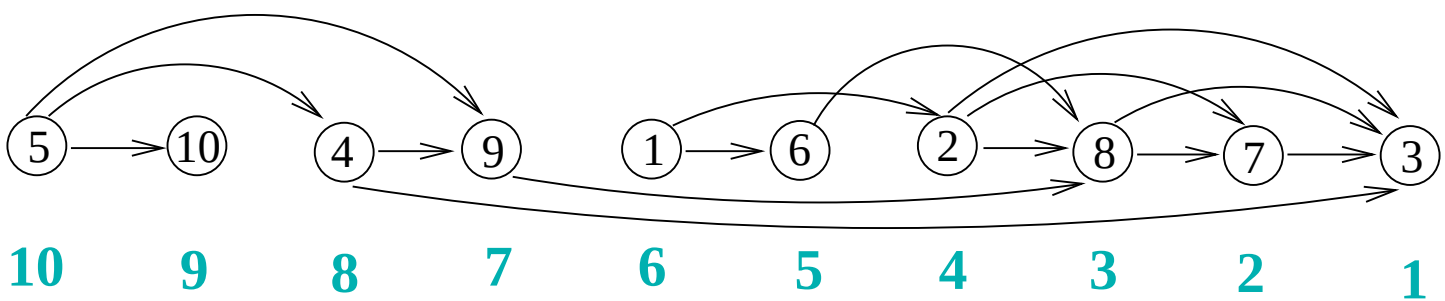
Definition 8.4.2

Sei $G = (V, E)$ ein azyklischer Digraph.

Eine **topologische Sortierung** (oder topologische Anordnung) der Knoten in V ist eine Bijektion $\pi\colon V \to \{1, \ldots, n\}$, so dass für alle Kanten $(v, w) \in E$ die Beziehung $\pi(v) < \pi(w)$ gilt.

Anschauung: Man setzt Knoten v an Position $\pi(v)$; dann laufen alle Kanten „von links nach rechts“.





Derselbe Graph, linear angeordnet. – Permutation π dazu:

v	1	2	3	4	5	6	7	8	9	10
$\pi(v)$	5	7	10	3	1	6	9	8	4	2

DFS(G) findet sofort eine topologische Sortierung!
(Natürlich nur in **azyklischen** Digraphen.)

Idee: Wir haben im Beweis von 8.4.1 gesehen, dass die f-Nummern entlang jeder Kante (v, w) **strikt fallen**; außerdem sind die f-Nummern eine **Bijektion**.

Definiere: $\boldsymbol{\pi(v)} := (n+1) - \mathit{f_num}(v)$.

Laufzeit/Kosten zur Ermittlung einer topologischen Sortierung: $O(|V|+|E|)$, **linear**.

Satz 8.4.3

DFS(G) (in der ausführlichen Version) testet Digraphen auf Azyklizität und findet gegebenenfalls eine topologische Sortierung von G, in Zeit $O(|V| + |E|)$.

Anwendung 1:

V: Menge von Tasks (für einen Computer, für eine Maschine).

$E \subseteq V \times V$: Zeitliche Abhängigkeiten.

$(v, w) \in E$ heißt: Task v muss beendet sein, bevor Task w beginnt.

Der resultierende Graph $G = (V, E)$ sollte azyklisch sein, sonst . . .
ist die Aufgabe überhaupt nicht durchführbar!

Eine topologische Sortierung ist dann eine mögliche Ausführungsreihenfolge für die Tasks auf der Maschine.

Bei Parallelverarbeitung: „Scheduling“ ist eine viel komplexere Aufgabe!

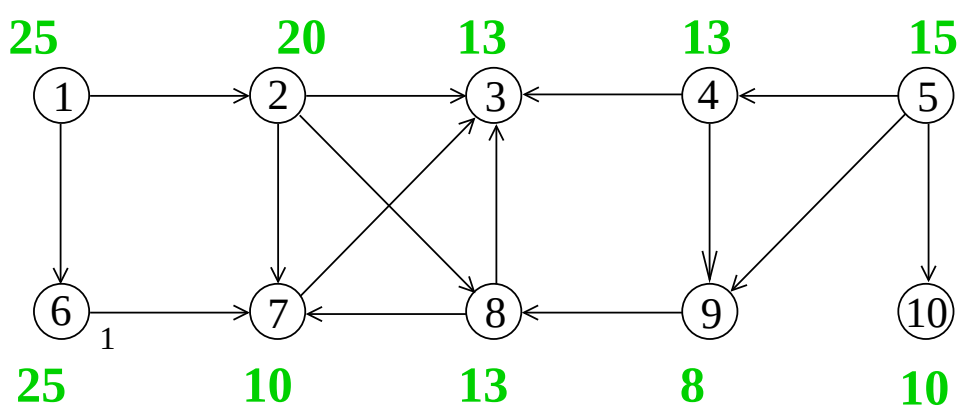
Anwendung 2:

V: Menge von Tasks (für einen Computer, für eine Maschine),
mit **Verarbeitungszeiten $c(v)$**, für $v \in V$.

$E \subseteq V \times V$: Zeitliche Abhängigkeiten wie vorher, azyklisch.

Es stehen genügend Maschinen zur Verfügung, um beliebig viele Tasks parallel auszuführen.

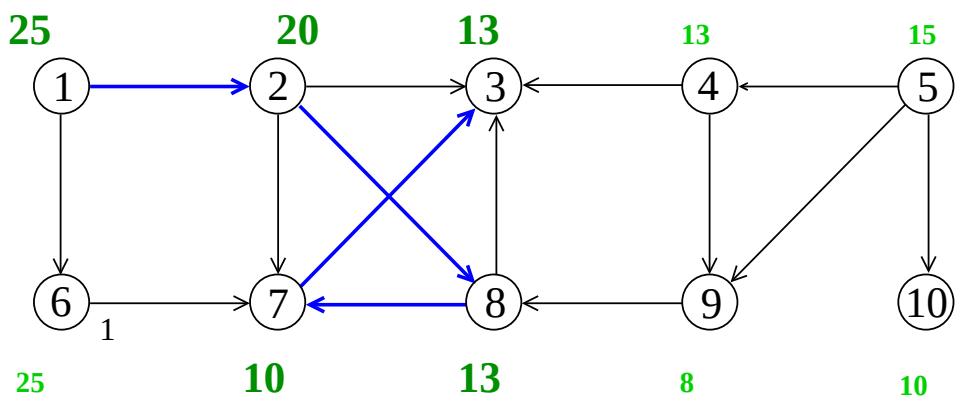
Frage: Wie lange dauert die Ausführung aller Tasks (mindestens) bei optimaler Planung?
Weitergehend: Finde einen optimalen Plan.



Definition: Ein **„kritischer Pfad“** in $G = (V, E)$ ist eine Folge $p = (v_0, \ldots, v_r)$ von $r \geq 1$ verschiedenen Tasks mit $(v_0, v_1), \ldots, (v_{r-1}, v_r) \in E$ und maximalen „Kosten“ $c(p) := c(v_0) + \cdots + c(v_r)$.

Klar: Die Gesamt-Ausführungszeit ist mindestens $c(p)$. (Man kann sogar Gleichheit zeigen.)

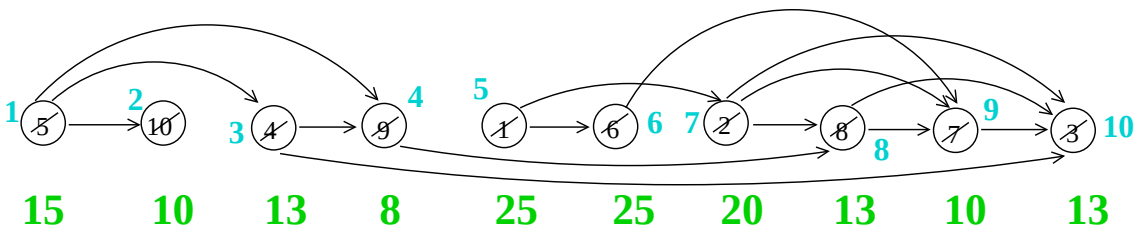
Aufgabe: Gegeben DAG $G = (V, E)$, bestimme kritischen Pfad, in Zeit $O(|V|+|E|)$.



Ein kritischer Pfad mit Kosten $25 + 20 + 13 + 10 + 13 = 81$.

Aber wie berechnen?

Lösung: Ermittle zunächst eine topologische Sortierung von $G = (V, E)$.
Dann benenne Knoten um, so dass („o.B.d.A.“) gilt: $(v, w) \in E \Rightarrow v < w$.



Definiere, für $1 \leq v \leq n$:

$\bar{c}(v)$:= Kosten eines teuersten Pfades in $\{v, \ldots, n\}$, der mit v beginnt.

$s(v)$:= erster Knoten nach v auf einem solchen Pfad bzw. undefiniert,
wenn v in G keinen Nachfolger hat.

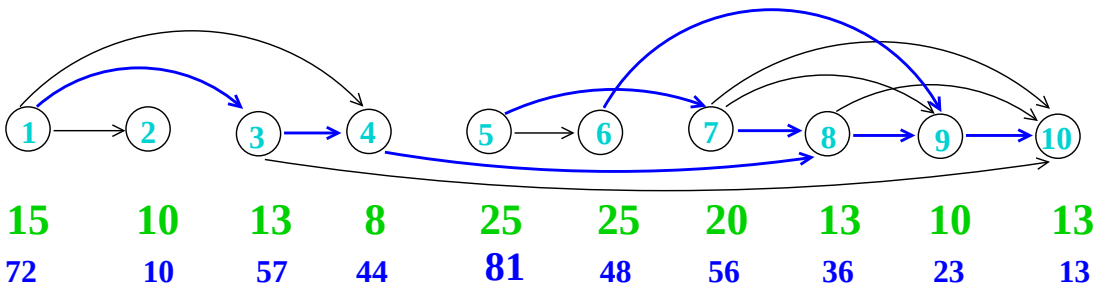
Berechne $\bar{c}(v)$ und $s(v)$ iterativ, für $v = n, \ldots, 1$, wie folgt.

Weil n keinen Nachfolger hat, gilt $\bar{c}(n) = c(n)$, und $s(n)$ ist undefiniert.

Nun sei $v < n$ und $t(w)$ für $v < w \leq n$ schon definiert. Dann gilt:

$$\bar{c}(v) = \begin{cases} c(v) + \max\{\bar{c}(w) \mid (v, w) \in E\} & \text{, falls eine Kante } (v, w) \text{ existiert } (\textbf{(!)}\ v < w), \\ c(v) & \text{, sonst.} \end{cases}$$

Ergebnis der Berechnung:



Ausgabe: Suche v^* mit maximalem $\bar{c}(v^*)$ (hier: $v^* = 5$ mit $\bar{c}(v^*) = 81$).

Kritischer Pfad: $(v^*, s(v^*), s(s(v^*)), \dots)$ (bis zum ersten Knoten ohne Nachfolger).

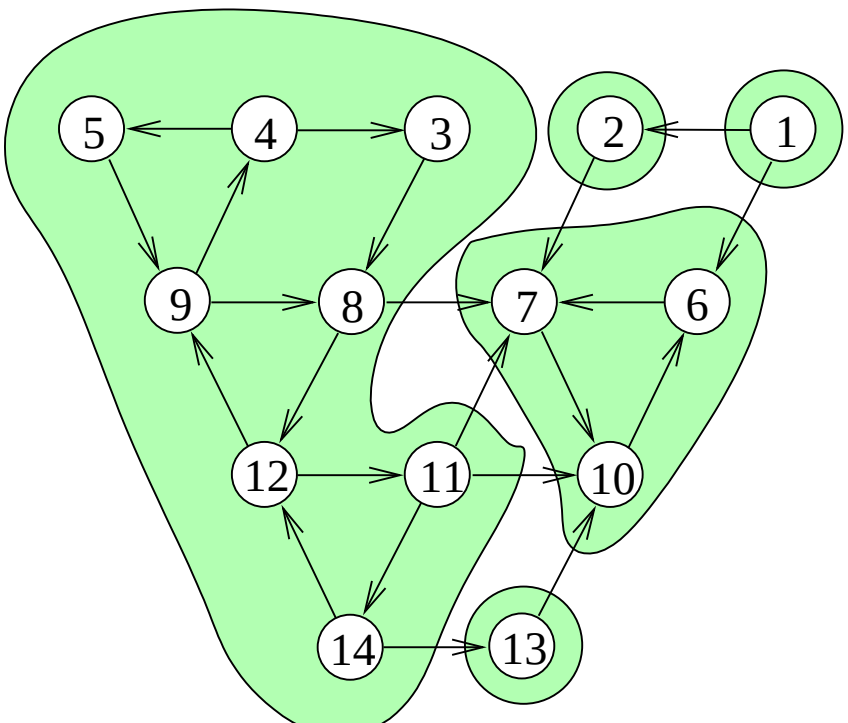
Hier: $(5, 7, 8, 9, 10)$.

Alternative: Berechnung in der Reihenfolge $1, \dots, n$. Nachteil: Benötigt Umkehrgraphen G^{R}.

Eine simple Erweiterung liefert sogar einen Ablaufplan, der optimale Zeit $\bar{c}(v^*)$ braucht:
Starte Task v zum Zeitpunkt $t(v) := \bar{c}(v^*) - \bar{c}(v)$.
(Das ist der „Aufschieberitis-Ablaufplan“: Alle Tasks werden so spät wie nur irgend möglich gestartet, ohne die bestmögliche Zeitschranke $\bar{c}(v^*)$ zu gefährden.)
Wenn (v, w) eine Kante ist, gilt $t(w) - t(v) = \bar{c}(v) - \bar{c}(w) \geq c(v)$ nach Konstruktion, also reicht die Zeit zwischen $t(v)$ und $t(w)$ für die Ausführung von Task v.

8.5 Starke Zusammenhangskomponenten in Digraphen

Beispiel: 5 starke Zusammenhangskomponenten



8.5 Starke Zusammenhangskomponenten in Digraphen

Definition 8.5.1

Zwei Knoten v und w in einem Digraphen G heißen **äquivalent**,
wenn $v \rightsquigarrow w$ und $w \rightsquigarrow v$ gilt. – Notation: $v \leftrightsquigarrow w$.

Überlege:

(a) Die so definierte Relation ist reflexiv, transitiv und symmetrisch, also eine **Äquivalenzrelation**.

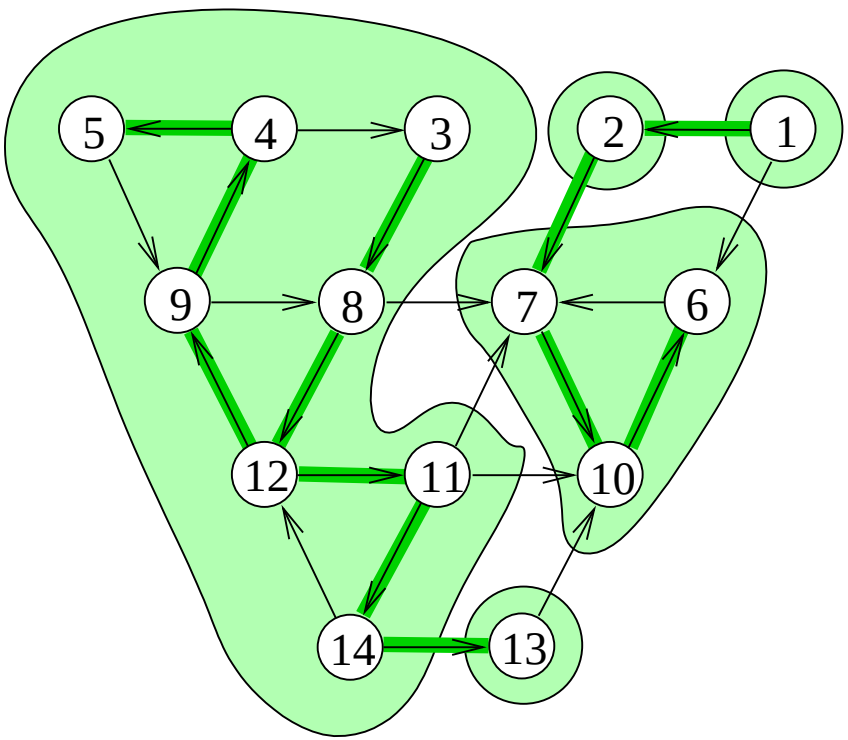
(b) Zwei Knoten v und w sind äquivalent genau dann wenn sie identisch sind **oder** es in G einen (gerichteten) Kreis gibt, auf dem beide Knoten liegen.

Definition 8.5.2

Die Äquivalenzklassen zur Relation $\leftrightsquigarrow$ heißen
starke Zusammenhangskomponenten von G.

Starke Zusammenhangskomponenten in Digraphen

Beispiel: 5 starke Zusammenhangskomponenten, 2 DFS-Bäume.



Lemma 8.5.3

Für jede starke Zusammenhangskomponente („**Komponente**“) $C \subseteq V$ gilt: Wenn DFS(G) ausgeführt wird, liegen die Knoten von C alle in einem Tiefensuchbaum.

Mit anderen Worten: Jeder Tiefensuchbaum ist selbst eine Komponente oder lässt sich (disjunkt) in mehrere Komponenten zerlegen.

Beweis: Es sei v_C der erste Knoten in C, für den dfs aufgerufen wird. Sei $v \in C$ beliebig.
Dann ist v von v_C aus auf einem Weg $(v_C = v_0, v_1, \ldots, v_t = v)$ **in G** erreichbar.

Behauptung: Alle Knoten v_i dieses Weges liegen sogar **in C**.

(*Beweis* hierfür: Weil C Komponente ist, gilt auch $v \rightsquigarrow v_C$. Zusammen mit $v_C \rightsquigarrow v_i$ und $v_i \rightsquigarrow v$ ergibt sich $v_i \leftrightsquigarrow v_C$, also $v_i \in C$.)

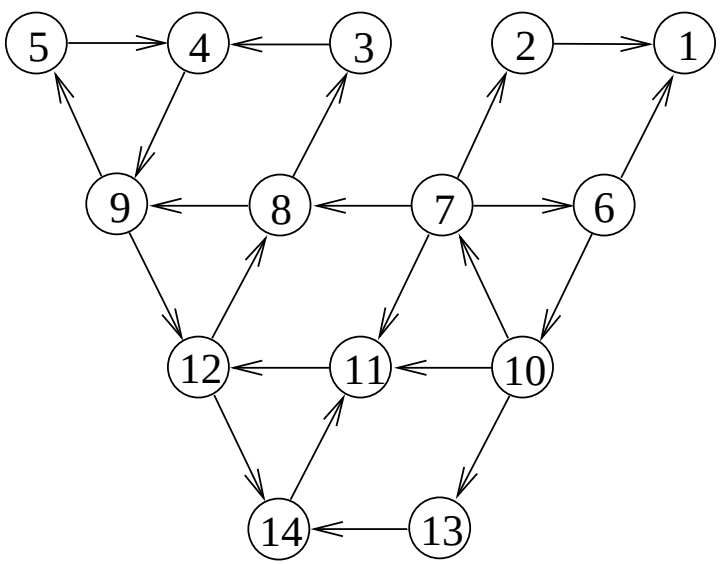
$\Rightarrow$ (nach Beh. und Wahl von $v_0 = v_C$) Zu dem Zeitpunkt, zu dem dfs(v_C) aufgerufen wird, sind die Knoten auf dem Weg $(v_0, v_1, \ldots, v_t = v)$ alle noch neu

$\Rightarrow$ (nach Satz 8.1.3, dem „Satz vom weißen Weg“) v wird Nachfahr von v_C im Tiefensuchbaum. □

Folgerung (aus dem Beweis): v_C hat die kleinste dfs-Nummer und die größte f-Nummer in C.

Um mit Tiefensuche die starken Zusammenhangskomponenten zu finden, muss man verhindern, dass in einem Baum mehrere solche Komponenten sitzen (wie im Bild auf Folie 58).

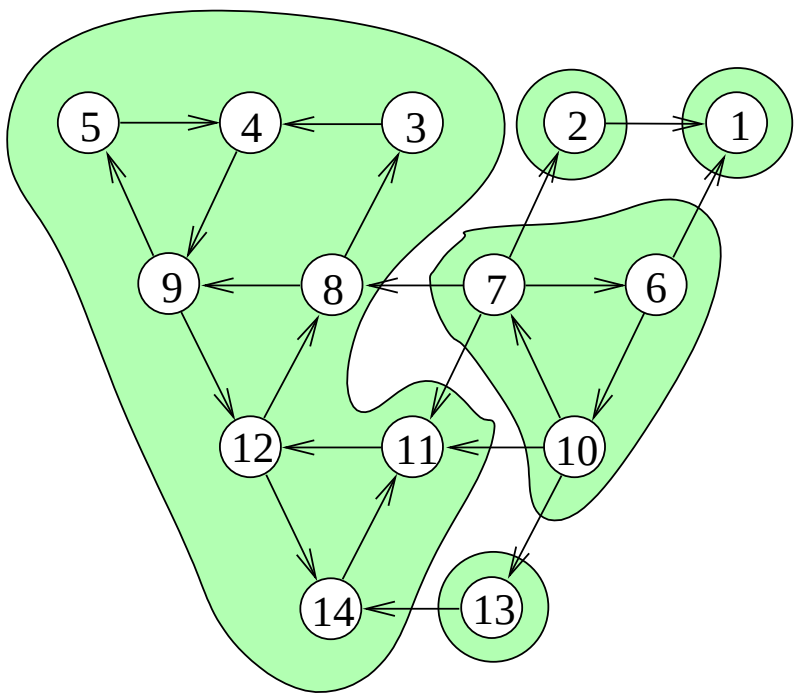
Trick 1: Führe DFS im „Umkehrgraphen“ G^{R} durch, der durch Umkehren aller Kanten in G entsteht.



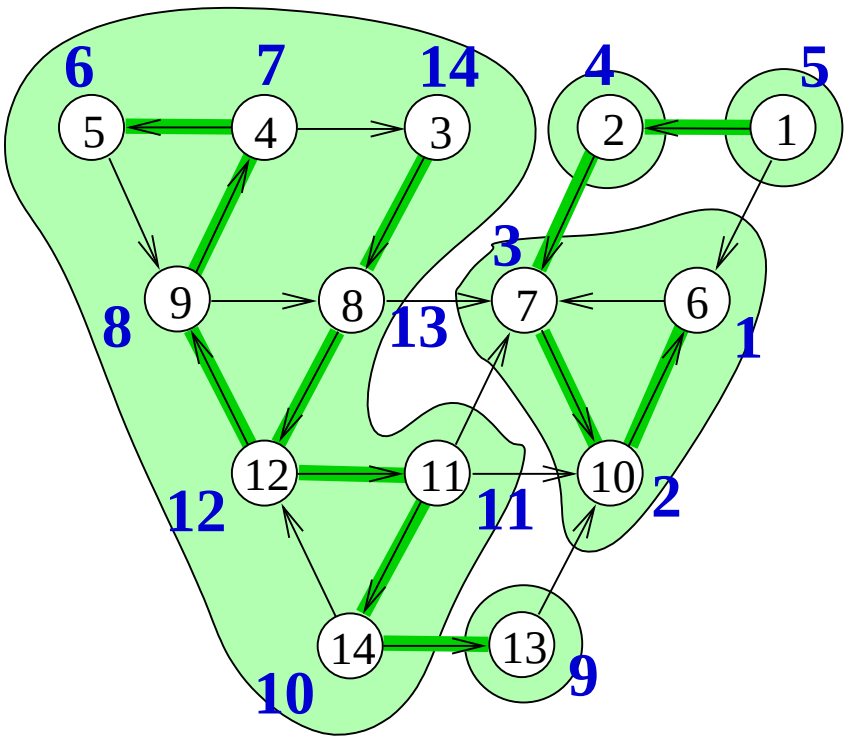
Umkehrgraph G^{R} von G.

Beobachtung: G^{R} hat dieselben starken Zusammenhangskomponenten wie G.

(Die Kreise in den beiden Graphen sind dieselben, nur mit umgekehrter Durchlaufreihenfolge.)

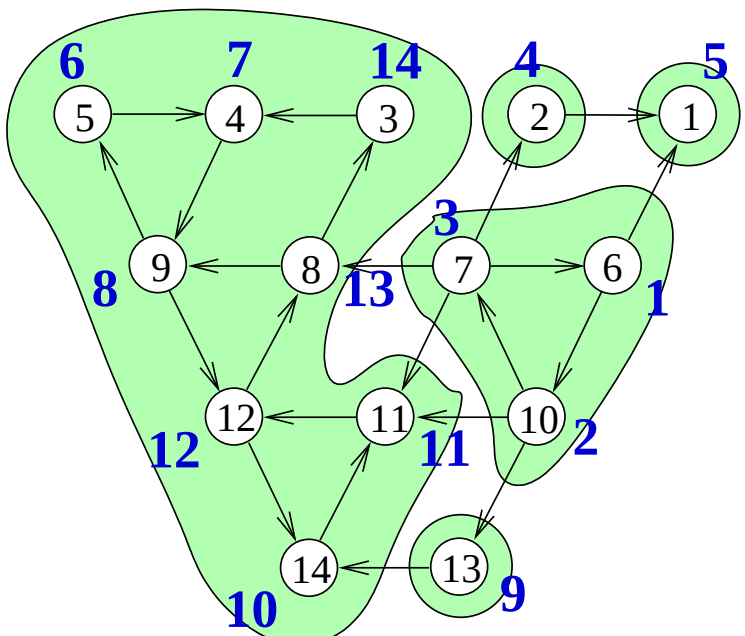


Trick 2: Verwende die **f-Nummern** einer ersten DFS in G, um zu bestimmen, in welcher Reihenfolge die Knoten in der Hauptschleife der zweiten DFS (in G^R) betrachtet werden.



Graph G mit starken Zusammenhangskomponenten, Tiefensuchwald und f-Nummern aus einem ersten DFS-Aufruf.

Beispiel:



Graph G^{R} mit Komponenten und f-Nummern für G.

Führe nun DFS in G^{R} durch, untersuche dabei Knoten gemäß **fallenden f-Nummern** auf Eigenschaft „neu“.

Im Beispiel: Von 3 **(14)** aus wird eine Komponente erreicht, von 13 **(9)** aus die nächste, von (1) **(5)** aus die nächste, von (2) **(4)** aus die nächste, von 7 **(3)** aus die letzte.

Keine Komponente wird „versehentlich“ betreten.

Algorithmus Strong Components(G) (von **S. R. Kosaraju**, 1978)

Eingabe: Digraph $G = (V, E)$

Ausgabe: Starke Zusammenhangskomponenten von G

(1) Berechne die f-Nummern mittels DFS(G);
(2) Bilde „Umkehrgraphen“ G^{R} durch Umkehren aller Kanten in G;
(3) Führe DFS(G^{R}) durch; Knotenreihenfolge: f-Nummern aus (1) absteigend
(4) **Ausgabe:** Die Knotenmengen der Tiefensuchbäume aus (3).

Satz 8.5.4

(a) Der Algorithmus **Strong Components** gibt die starken Zusammenhangskomponenten von G aus.

(b) Die Laufzeit ist $O(|V| + |E|)$.

Beweis von Teil (b): Die DFS-Aufrufe haben lineare Laufzeit, und auch der Umkehrgraph lässt sich in Zeit $O(|V| + |E|)$ berechnen.

Beweis von Teil (a): Nach Lemma 8.5.3 wissen wir, dass jeder DFS-Baum der Tiefensuche in G^{R} Vereinigung von Komponenten (von G^{R}, also von G) ist.

Wir zeigen im Folgenden, dass jeder solche Baum nur eine Komponente enthält.

Betrachte dazu eine beliebige starke Zusammenhangskomponente C von G.
$\boldsymbol{v_C} \in C$ sei **der Knoten mit maximaler f-Nummer** in C.

(Aus „Folgerung“ nach Lemma 8.5.3: In der ersten Tiefensuche in G ist v_C der erste Knoten von C, der besucht wird, also auch der mit minimaler DFS-Nummer in C.)

Wir wollen zeigen, dass in der Tiefensuche in G^R (Teil (3))
Knoten v_C Wurzel eines DFS-Baums wird.

(D. h.: In DFS(G^R) in Teil (3) enthält jede Komponente eine Wurzel eines DFS-Baums.
Mit Lemma 8.5.3 folgt: Jeder solche DFS-Baum enthält genau eine Komponente, und wir sind fertig.)

Wegen der besonderen Reihenfolge der Knoten in der äußeren Schleife in Teil (3) gilt nach Satz 8.1.5:

In Teil (3) wird v_C Wurzel eines DFS-Baums

$\Leftrightarrow$ es gibt kein v mit *f-num*$(v) >$ *f-num*(v_C) und $v \rightsquigarrow_{G^R} v_C$

$\Leftrightarrow$ es gibt kein v mit *f-num*$(v) >$ *f-num*(v_C) und $v_C \rightsquigarrow_G v$

$\Leftrightarrow$ für alle v mit $v_C \rightsquigarrow_G v$ gilt *f-num*$(v) \leq$ *f-num*(v_C).

Es genügt also, Folgendes zu zeigen:

Lemma 8.5.5

Aus $v_C \rightsquigarrow_G v$ folgt *f-num*$(v) \leq$ *f-num*(v_C) (in DFS in Teil (1)!).

Lemma 8.5.5 Aus $v_C \rightsquigarrow_G v$ folgt *f-num*$(v) \leq$ *f-num*(v_C).

Beweis: O. B. d. A. gilt $v_C \neq v$.

Sei $\underbrace{v_C = v_0, v_1, \ldots, v_{t-1}, v_t = v}_{=: p}$ ein Weg in G.

1. Fall: Wenn dfs(v_C) startet, sind alle Knoten auf p **neu** .
Nach Satz 8.1.3 („Satz vom weißen Weg“) wird v Nachfahr von v_C im DFS-Baum, also ist *f-num*$(v) <$ *f-num*(v_C).

2. Fall: Wenn dfs(v_C) startet, ist ein Knoten v_i, $0 < i \leq t$, **aktiv** .
Dann führt ein Schluss-Stück des roten Weges (Folie 12) von v_i zu v_C.
Von v_C nach v_i führt der erste Teil des Weges p.
Daraus: $v_C \leftrightsquigarrow v_i$, also $v_i \in C$.
Andererseits ist *f-num*$(v_i) >$ *f-num*(v_C), im **Widerspruch** zur Wahl von v_C.
Der 2. Fall kann also gar nicht eintreten.

3. Fall: Wenn dfs(v_C) startet, ist ein Knoten v_i, $0 < i \leq t$, **fertig**.

Wegen des Vorgehens der dfs-Prozedur und weil Fall 2 nicht eintritt, gilt für $i \leq j < t$, zu dem Zeitpunkt, an dem dfs(v_C) startet:

Wenn v_j „fertig“ ist, dann ist auch v_{j+1} „fertig“.
(Bevor dfs(v_j) endete, wurde Kante (v_j, v_{j+1}) betrachtet; also kann v_{j+1} nicht „neu“ sein.)

Also ist v „fertig“, wenn dfs(v_C) startet.

Daraus folgt: *f-num*$(v) <$ *f-num*(v_C). □

SS 2021

Algorithmen und Datenstrukturen

9. Kapitel

Divide-and-Conquer-Algorithmen

Martin Dietzfelbinger

Juli 2021

Kapitel 9 Divide-and-Conquer (D-a-C)

Das **Algorithmenparadigma** hinter **MergeSort**

Vorgehen eines D-a-C-Algorithmus $\mathcal{A}$ für Berechnungsproblem $\mathcal{P} = (\mathcal{I}, \mathcal{O}, f)$
auf Instanz $x \in \mathcal{I}$:

0) Trivialitätstest: Wenn $\mathcal{P}$ für x einfach direkt zu lösen ist, tue dies. – Sonst:

1) Teile: Bilde aus x ($a \geq 1$ viele) Instanzen $x_1, \ldots, x_a \in \mathcal{I}$.

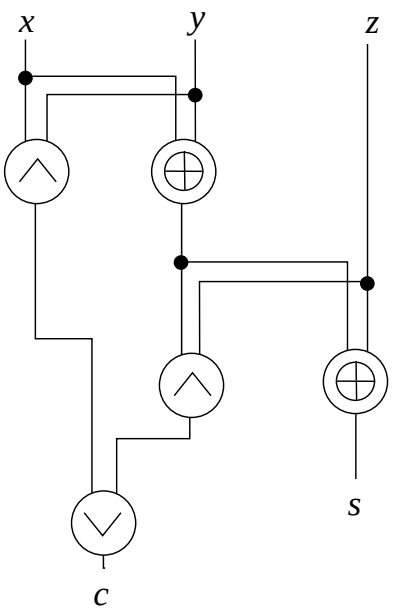
2) Rekursion: Löse $\mathcal{P}$ für $x_1, \ldots, x_a$, separat, durch
rekursive Verwendung von $\mathcal{A}$. Teillösungen: $r_1, \ldots, r_a$.

3) Kombiniere: Baue aus x und $r_1, \ldots, r_a$ und $x_1, \ldots, x_a$
eine Lösung r für $\mathcal{P}$ auf x auf.

9.1 Multiplikation ganzer Zahlen

Zahlen in Binärdarstellung.
(Methoden funktionieren im Prinzip für jede beliebige Basis.)

Bekannt:



Volladdierer (5 zweistellige Bitoperationen)
liefert zu 3 Bits x, y, z die zwei Bits
$(c, s) = \text{fulladd}(x, y, z)$ mit

$$c = (x \wedge y) \vee (y \wedge z) \vee (z \wedge x)$$

(Übertragsbit, **Carry**bit) und

$$s = d \oplus e \oplus f$$

(**Summen**bit).

Bekannt: Serielle **Binäraddition**.

Input: Binärzahlen/-strings $a_{n-1} \dots a_0$ und $b_{n-1} \dots b_0$, ≥ 0

$\mathtt{c}_0 \leftarrow 0$; // Carry, Übertrag
for $\mathtt{i}$ **from** 0 **to** $n-1$ **do** $(\mathtt{c}_{\mathtt{i}+1}, \mathtt{s}_\mathtt{i}) \leftarrow \text{fulladd}(a_\mathtt{i}, b_\mathtt{i}, \mathtt{c}_\mathtt{i})$;
$\mathtt{s}_n \leftarrow \mathtt{c}_n$;
Ergebnis: $\mathtt{s}_n \dots \mathtt{s}_0$.

Kosten: Nicht mehr als $5n = O(n)$ Bitoperationen.

Bekannt: Ganze Zahlen: Notation als Paar (Vorzeichen, Betrag (in Binärdarst.)),
z. B. $10, -1001, 101010, -11110$.

Rechnerintern: **Zweierkomplementdarstellung**.
Addition und Subtraktion auf die **Addition vorzeichenloser Zahlen** zurückführbar.
Kosten für n-Bit-Zahlen: $\leq 10n$ Bitoperationen.

Multiplikation zweier natürlicher Zahlen

„Schulmethode“ SM(x, y)

Faktoren x, y:	1	0	0	1	1	0	·	1	1	0	0	1	1
			1	0	0	1	1	0	0	0	0	0	0
				1	0	0	1	1	0	0	0	0	0
					0	0	0	0	0	0	0	0	0
						0	0	0	0	0	0	0	0
							1	0	0	1	1	0	0
		+						1	0	0	1	1	0
Produkt:			1	1	1	1	0	0	1	0	0	1	0

Multiplikation $\widehat{=}$ Addition von n Binärzahlen.

Allgemein:

Input: Binärzahlen $x = a_{n-1} \dots a_0$ und $y = b_{n-1} \dots b_0$

Bilde n Binärzahlen $d^{(0)}, \dots, d^{(n-1)}$:

$$d^{(i)} = (a_{n-1} \cdot b_i) \dots (a_0 \cdot b_i) \underbrace{0 \dots 0}_{i \text{ Nullen}}$$

und addiere alle diese.

$\leq n-1$ Additionen von Zahlen mit nicht mehr als $2n$ Bits:

$O(n^2)$ Bitoperationen.

Überlege: Was ändert sich bei Ziffernsatz $\{0, 1, \dots, b-1\}$ statt $\{0, 1\}$? (Z. B. $b = 10$.)

Geht es billiger als in Zeit $O(n^2)$?

Multiplikation mit **Divide-and-Conquer**-Strategie:

Eingabe: n-Bit-Binärzahlen x und y, eventuell Vorzeichen.

Falls $n \leq n_0$: Benutze Schulmethode.

Falls $n > n_0$: (**„Teile“**)

Setze $k = \lceil n/2 \rceil$.

Schreibe $x = \underbrace{a_{n-1} \dots a_k}_{A} \underbrace{a_{k-1} \dots a_0}_{B}$

und $y = \underbrace{a_{n-1} \dots a_k}_{C} \underbrace{a_{k-1} \dots a_0}_{D}$

Dann $x = A \cdot 2^k + B$ und $y = C \cdot 2^k + D$.

Also

$$x \cdot y = (A \cdot 2^k + B) \cdot (C \cdot 2^k + D) = A \cdot C \cdot 2^{2k} + (A \cdot D + B \cdot C) \cdot 2^k + B \cdot D.$$

Erste Idee: Berechne **rekursiv** $A \cdot C$, $A \cdot D$, $B \cdot C$, $B \cdot D$,

und füge die Produkte durch einige Additionen zum Resultat $x \cdot y$ zusammen.

Kosten für n-Bit-Zahlen (für eine Konstante c):

$$C(n) \leq \begin{cases} 1 & \text{für } n = 1 \\ 4 \cdot C(n/2) + c \cdot n & \text{für } n > 1. \end{cases}$$

Der Summand $4 \cdot C(n/2)$ erfasst die Kosten der vier rekursiven Aufrufe, der Summand cn die Kosten der Additionen.

Man **kann** zeigen (machen wir später, „Master-Theorem“):

Die Anzahl der Bitoperationen ist wieder $\Theta(n^2)$, nicht besser als Schulmethode.

Wir haben: $x \cdot y = A \cdot C \cdot 2^{2k} + (A \cdot D + B \cdot C) \cdot 2^k + B \cdot D$.

Trick:

$E := A - B$ und $F := C - D$ (sieht sinnlos aus . . .)

Bemerke: $|E|$ und $|F|$ haben als Betrag der Differenz von zwei nichtnegativen k-Bit-Zahlen höchstens k Bits. – Dann:

$E \cdot F = (A - B) \cdot (C - D) = A \cdot C + B \cdot D - (A \cdot D + B \cdot C)$.

Also:

$$A \cdot D + B \cdot C = A \cdot C + B \cdot D - E \cdot F.$$

Eingesetzt:

$$x \cdot y = A \cdot C \cdot 2^{2k} + (A \cdot C + B \cdot D - E \cdot F) \cdot 2^k + B \cdot D.$$

Nur noch **drei** rekursive Multiplikationen von k-Bit-Zahlen nötig!

Algorithmus Karatsuba(x, y)

Eingabe: Zwei n-Bit-Zahlen x und y, nichtnegativ
if $n \leq n_0$ **then return** SM(x, y) // Multiplikation mit Schulmethode
else
$k \leftarrow \lceil n/2 \rceil$;
zerlege $x = A \cdot 2^k + B$ und $y = C \cdot 2^k + D$; // vier k-Bit-Zahlen
$E \leftarrow A - B$ und $F \leftarrow C - D$; // k-Bit-Zahlen, mit Vorzeichen
$G \leftarrow$ Karatsuba(A, C); // Rekursion
$H \leftarrow$ Karatsuba(B, D); // Rekursion
$I \leftarrow$ Karatsuba$(|E|, |F|)$; // Rekursion
return $G \cdot 2^{2k} + (G + H - \text{sign}(E) \cdot \text{sign}(F) \cdot I) \cdot 2^k + H$.

(sign(a) ist das Vorzeichen der ganzen Zahl a.
Multiplikationen mit ± 1 und Zweierpotenzen sind „umsonst“.)

Beispiel: Mit Dezimalzahlen, $n_0 = 2$.

(Methode funktioniert mit jeder Basis $b \geq 2$. In der Informatik interessante Basiszahlen b:
2, 8, 10, 16 (Hexzahlen), 256 (Ziffern sind Bytes), 2^{16}, 2^{32} (Ziffern sind (Halb-)Wörter), . . .)

$n = 8$, $x = 76490358$, $y = 35029630$.

$A = 7649$, $B = 0358$, $C = 3502$, $D = 9630$.

$E = A - B = 7291$, $F = C - D = -6128$, $\mathsf{sign}(E) \cdot \mathsf{sign}(F) = 1 \cdot (-1) = -1$.

Jeweils ≤ 4 Dezimalziffern.

Rekursion für $A \cdot C$:
$a = 76, b = 49, c = 35, d = 02$.
$e = a - b = 27$, $f = c - d = 33$.

Weil z.B. $n_0 = 2$ ist, wird direkt multipliziert:
$g = a \cdot c = 76 \cdot 35 = 2660$, $h = b \cdot d = 98$, $i = |e| \cdot |f| = 27 \cdot 33 = 891$.

3 Multiplikationen von 2-Bit-Zahlen!

Ergebnis:

$$G = A \cdot C = 2660 \cdot 10^4 + (2660 + 98 - 891) \cdot 10^2 + 98 = 26786798.$$

Analog, jeweils rekursiv:
$H = B \cdot D = 03447540$, $I = |E| \cdot |F| = 44679248$.

Ergebnis:

$$x \cdot y = 26786798 \cdot 10^8 + (26786798 + 03447540 - (-1) \cdot 44679248) \cdot 10^4 + 03447540 = 267942893939307540$$

Multiplikation mit 10^k: Anhängen von Nullen. $\Rightarrow$ Beim Kombinationsschritt gibt es nur Additionen!

Laufzeitanalyse:

Es sei $n_0 = 1$ und n sei eine Zweierpotenz: $\boldsymbol{n = 2^\ell}$.

$T_{\mathsf{Ka}}(n) :=$ Anzahl der Bit-Operationen, die der Algorithmus von Karatsuba auf einer Eingabe aus zwei n-Bit-Zahlen macht, mit $n_0 = 1$. Klar: $T_{\mathsf{Ka}}(1) = 1$.

Für einen Input der Größe $n > n_0$ müssen wir (rekursiv) $a = 3$ Teilinstanzen bearbeiten, für Parametergröße $\lceil n/b \rceil = \lceil n/2 \rceil$, also $b = 2$,
und im Kombinationsschritt einige Additionen und Subtraktionen von Zahlen mit maximal $2n$ Bits durchführen, was zusätzlich $O(n)$ Bitoperationen erfordert.

Rekurrenzungleichung:

$$T_{\mathsf{Ka}}(n) \leq \begin{cases} 1 & \text{für } n = 1, \\ 3 \cdot T_{\mathsf{Ka}}(n/2) + c \cdot n & \text{für } n > 1, \end{cases}$$

wobei c konstant ist.

Direkte Rechnung,

wobei die Rekurrenzungleichung für n, $n/2$, $n/4$, $n/8$, ... an der Stelle von n benutzt wird:

$$
\begin{aligned}
T_{\mathsf{Ka}}(n) &\leq 3 \cdot T_{\mathsf{Ka}}(n/2) + c \cdot n \\
&\leq 3 \cdot (3 \cdot T_{\mathsf{Ka}}((n/2)/2) + c \cdot n/2) + c \cdot n \\
&= 3^2 \cdot T_{\mathsf{Ka}}(n/2^2) + c \cdot 3 \cdot n/2 + c \cdot n \\
&\leq 3^3 \cdot T_{\mathsf{Ka}}(n/2^3) + c \cdot 3^2 \cdot n/2^2 + c \cdot 3 \cdot n/2 + c \cdot n \\
&\;\;\vdots \\
&\leq 3^\ell \cdot T_{\mathsf{Ka}}(n/2^\ell) + c \cdot \sum_{0 \leq j < \ell} 3^j \cdot n/2^j. \\
&= 3^\ell \cdot T_{\mathsf{Ka}}(1) + cn \cdot \sum_{0 \leq j < \ell} (3/2)^j.
\end{aligned}
$$

(Fortsetzung folgt)

Wir setzen fort, unter Benutzung der Formel für geometrische Reihen:

$$T_{\mathsf{Ka}}(2^\ell) \leq 3^\ell \cdot \underbrace{T_{\mathsf{Ka}}(1)}_{=1} + cn \cdot \sum_{0 \leq j < \ell} (3/2)^j$$

$$= 3^\ell + c \cdot 2^\ell \cdot \frac{(\frac{3}{2})^\ell - 1}{\frac{3}{2} - 1}$$

$$< 3^\ell \cdot (1 + 2c).$$

Nun gilt $3^\ell = (2^{\log_2 3})^\ell = 2^{(\log_2 3)\ell} = 2^{\ell \log_2 3} = (2^\ell)^{\log_2 3} = n^{\log_2 3}$.

$\Rightarrow T_{\mathsf{Ka}}(n) \leq (1 + 2c) n^{\log_2 3} = O(n^{1{,}585})$.

Satz 9.1.1

Beim Karatsuba-Multiplikationsalgorithmus beträgt die Anzahl der Bitoperationen und die Rechenzeit $O(n^{\log_2 3}) = O(n^{1{,}585})$.

Bemerkungen

$n^{\log_2 3}$ ist viel kleiner als n^2!

Dieselbe Rechnung mit „4“ an der Stelle von „3“ beweist, dass das naive rekursive Verfahren mit vier rekursiven Aufrufen (Folie 7) Rechenzeit $O(n^{\log_2 4}) = O(n^2)$ hat.

In der Praxis (Implementierung in Software): für $w =$ Wortlänge des Rechners (z. B. $w = 32$) für Operationen auf Ziffern die eingebaute Multiplikations-Hardware benutzen. (D. h.: mit Basis $b = 2^w$ rechnen.)

Für Zahlen bis zu einer Länge von n_0 Worten: Schulmethode. (Auch in der Rekursion!)

Nur für noch längere Zahlen D-a-C-Verfahren benutzen.
Welches n_0 optimal ist, hängt von der Hardware, vom Compiler und von Programmierdetails ab.
Ein gutes n_0 kann man experimentell bestimmen. (Oft gut: $24 \leq n_0 \leq 32$.)

(Studie hierzu: Buch [D./Mehlhorn/Sanders].)

Beispiele: 1024 Binärziffern, gut 300 Dezimalziffern.

Will man mit so riesigen Zahlen rechnen?

Ja! – Kryptographie!

Ignoriere Additionen, setze $n_0 = 32$.
$M'_{\mathsf{Ka}}(n) = \#$(32-Bit-Mult. bei Karatsuba für zwei n-Bit-Zahlen), mit $n_0 = 32$.

Rekurrenzgleichung:

$$M'_{\mathsf{Ka}}(n) = \begin{cases} 1 & \text{für } n \leq n_0, \\ 3 \cdot M'_{\mathsf{Ka}}(n/2) & \text{für } n > n_0, \end{cases}$$

Daraus: $M'_{\mathsf{Ka}}(2^\ell) = 3^{\ell-5} \cdot M'_{\mathsf{Ka}}(2^5) = 3^{\ell-5}$.

$2^{10} = 1024$ Binärziffern, $\ell = 10$:

Karatsuba: $3^5 = 243$ Multiplikationen$_{32}$;
Schulmethode: $(2^{\ell-5})^2 = 1024$ Multiplikationen$_{32}$.

$2^{15} = 32768$ Binärziffern, $\ell = 15$, ca. 9900 Dezimalziffern:

Karatsuba: $3^{10} = 59049$ Multiplikationen$_{32}$;
Schulmethode:
$(2^{15-5})^2 = 2^{20}$ Multiplikationen$_{32}$, mehr als 1 Million!

Ersparnis: Faktor 18.

Geht es noch besser? Theoretisch ja, praktisch eigentlich kaum.

Mitteilung:

- Schönhage-Strassen (1971): Multiplikation zweier n-Bit-Zahlen in Zeit $O((n \log n) \cdot \boldsymbol{\log \log n})$.

  [A. Schönhage und V. Strassen: Schnelle Multiplikation großer Zahlen, Computing 7, 1971, Springer-Verlag, S. 281–292]
- Fürer (2007), De *et al.* (2008): Multiplikation zweier n-Bit-Zahlen in Zeit $O((n \log n) \cdot \boldsymbol{2^{\log^* n}})$. ($\log^* n$ wächst extrem langsam.)

  [M. Fürer: Faster integer multiplication, STOC 2007, S. 57–66].
  [A. De, P. P. Kurur, C. Saha, R. Saptharishi: Fast integer multiplication using modular arithmetic. STOC 2008, S. 499–506 und SIAM J. Comput. 42(2): 685-699 (2013)].
- David Harvey, Joris Van Der Hoeven (2019/21): Multiplikation zweier n-Bit-Zahlen in Zeit $\boldsymbol{O(n \log n)}$.
  [Annals of Mathematics Vol. 193, No. 2 (March 2021), pp. 563–617. `https://doi.org/10.4007/annals.2021.193.2.4`]

Dabei ist $\log^* n$ definiert als die kleinste Zahl i mit

$$\underbrace{\log \log \ldots \log}_{i\text{-mal}} n \leq 1.$$

Also: $\log^* 2 = 1$, $\log^* 4 = 2$, $\log^* 16 = 3$,

$\log^*(65536) = 4$, $\log^*(2^{65536}) = 5$, $\log^*(2^{2^{65536}}) = 6$.

$2^{2^{65536}}$ ist schon eine **sehr** große Zahl.

$\Rightarrow$ Zahlen n mit $\log^* n > 6$ kommen in der Praxis nicht vor.

9.2 Matrixmultiplikation

Es sei R irgendein **Ring**.[1]

$A = (a_{ij})_{1 \le i,j \le n}$, $B = (b_{ij})_{1 \le i,j \le n}$ seien $n \times n$-Matrizen über R.

Aufgabe: Berechne die Produktmatrix $P = A \cdot B$, d.h. $P = (p_{ij})_{1 \le i,j \le n}$ mit

$$p_{ij} = \sum_{1 \le k \le n} a_{ik} b_{kj}.$$

Eine **naive** Implementierung gemäß dieser Formel kostet n^3 Ring-Multiplikationen und $n^2(n-1)$ Ring-Additionen, hat also Rechenzeit $\boldsymbol{O(n^3)}$.

Strassen (1969): Es geht mit weniger Multiplikationen! Ansatz: Divide-and-Conquer.

[1] Man kann addieren, subtrahieren, multiplizieren, gemäß Standard-Rechenregeln. Bsp.: $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$.

Wir nehmen an: $n = 2^\ell$, Zweierpotenz.

Eingabe: Zwei $n \times n$-Matrizen A und B.

Falls $n \leq n_0$: Berechne $A \cdot B$ mit der direkten Methode. Kosten n_0^3 Multiplikationen.

Falls $n > n_0$, zerlege A, B in jeweils 4 quadratische $(\frac{n}{2} \times \frac{n}{2})$-Teilmatrizen:

$$A = \left(\begin{array}{c|c} C & D \\ \hline E & F \end{array}\right), \qquad B = \left(\begin{array}{c|c} G & H \\ \hline K & L \end{array}\right).$$

Dann gilt (wie nicht schwer nachzukontrollieren ist):

$$A \cdot B = \left(\begin{array}{c|c} C \cdot G + D \cdot K & C \cdot H + D \cdot L \\ \hline E \cdot G + F \cdot K & E \cdot H + F \cdot L \end{array}\right).$$

Haben:

$$A \cdot B = \left(\begin{array}{c|c} C \cdot G + D \cdot K & C \cdot H + D \cdot L \\ \hline E \cdot G + F \cdot K & E \cdot H + F \cdot L \end{array} \right).$$

Dies suggeriert einen rekursiven Ansatz, in dem 8 Multiplikationen von $(\frac{n}{2} \times \frac{n}{2})$-Teilmatrizen durchgeführt werden.

Einfache Analyse ergibt: Dies führt zu n^3 Multiplikationen in R, kein Gewinn.

(Unten zeigen wir mit dem Master-Theorem: $O(n^3)$.)

Strassen-Trick: 7 Multiplikationen genügen (mit einigen Additionen/Subtraktionen).

$$P_1 = C \cdot (H - L)$$
$$P_2 = (C + D) \cdot L$$
$$P_3 = (E + F) \cdot G$$
$$P_4 = F \cdot (K - G)$$
$$P_5 = (C + F) \cdot (G + L)$$
$$P_6 = (D - F) \cdot (K + L)$$
$$P_7 = (C - E) \cdot (G + H)$$

Dann:

$$A \cdot B = \left(\begin{array}{c|c} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ \hline P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{array} \right)$$

Von Hand nachzukontrollieren!

18 Additionen von $(\frac{n}{2} \times \frac{n}{2})$-Matrizen.

(Alternative Methode, etwas komplizierter, benötigt nur 15 Additionen.)

Aufwandsanalyse:

Rekurrenzungleichung für die Anzahl der Operationen:

$$T_{\mathsf{Str}}(n) \leq \begin{cases} 1 & \text{für } n = 1, \\ 7 \cdot T_{\mathsf{Str}}(n/2) + c \cdot n^2 & \text{für } n > 1. \end{cases}$$

$n = 1$: Eine Ringmultiplikation.

$n > 1$: Neben den rekursiven Aufrufen genau 18 Additionen von $(\frac{n}{2} \times \frac{n}{2})$-Matrizen, mit Kosten $18 \cdot (n/2)^2 = 4{,}5 \cdot n^2$.

Also: $c = 4{,}5$ ist geeignete Konstante.

Nun: Rechnung für $n = 2^\ell$ wie bei der Analyse des Karatsuba-Algorithmus.

$$
\begin{aligned}
T_{\mathsf{Str}}(n) &\leq 7 \cdot T_{\mathsf{Str}}(n/2) + c \cdot n^2 \\
&\leq 7^2 \cdot T_{\mathsf{Str}}(n/4) + c \cdot 7 \cdot (n/2)^2 + c \cdot n^2 \\
&\leq 7^3 \cdot T_{\mathsf{Str}}(n/8) + c \cdot 7^2 \cdot (n/4)^2 + c \cdot 7 \cdot (n/2)^2 + c \cdot n^2 \\
&\quad \vdots \\
&\leq 7^\ell \cdot \underbrace{T_{\mathsf{Str}}(n/2^\ell)}_{=1} + c \cdot n^2 \cdot \sum_{0 \leq j < \ell} \frac{7^j}{(2^j)^2} \\
&= 7^\ell + c \cdot n^2 \cdot \sum_{0 \leq j < \ell} \left(\frac{7}{4}\right)^j < 7^\ell + c \cdot (2^\ell)^2 \cdot \frac{(\frac{7}{4})^\ell}{\frac{7}{4} - 1} \\
&= 7^\ell \cdot (1 + 4c/3).
\end{aligned}
$$

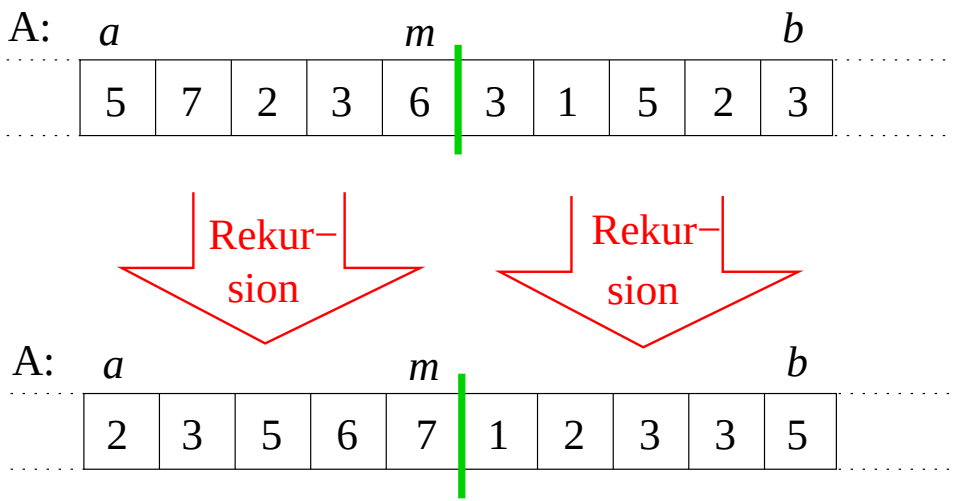
Wie beim Karatsuba-Algorithmus: $T_{\mathsf{Str}}(n) = T_{\mathsf{Str}}(2^\ell) = O(7^\ell)$.

Dabei: $7^\ell = 2^{\ell \log_2 7} = n^{\log_2 7}$ mit $\log_2 7 \approx 2{,}81$.

Satz 9.1.2

Der Algorithmus von **Strassen** für die Matrixmultiplikation führt bei $n \times n$-Matrizen über dem Ring R als Eingabe höchstens $n^{\log_2 7}$ Ringmultiplikationen und $O(n^{\log_2 7})$ Ringadditionen und -subtraktionen aus. Dabei ist $\log_2 7 \approx 2{,}81$.

9.3 Erinnerung: Mergesort (Kap. 6.2)



Prozedur r_MergeSort(a, b):

0) Falls $\boldsymbol{n} := b - a + 1 \leq n_0$: Insertionsort.

1) $m := \lfloor (a+b)/2 \rfloor$; Aufteilen von $\mathtt{A}[a \ldots b]$ in $\mathtt{A}[a \ldots m]$ und $\mathtt{A}[m+1 \ldots b]$ (Längen $\lceil n/2 \rceil$ und $\lfloor n/2 \rfloor$).

2) Diese beiden Segmente werden **rekursiv** sortiert.

3) **Mischen** von $\mathtt{A}[a \ldots m]$ und $\mathtt{A}[m+1 \ldots b]$: **Merge**(a, m, b). ($\leq n-1$ Vergleiche.)

Rekurrenzgleichung für Vergleichsanzahl $C(n)$ für Sortieren eines Teilarrays der Länge n:

$$C(n) = \begin{cases} 0, & \text{für } n = 0, \\ C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + n - 1, & \text{für } n \geq 1. \end{cases}$$

Gezeigt in 6.2, mit ad-hoc-Beweis: $C(n) = n\lceil \log n \rceil - (2^{\lceil \log n \rceil} - 1) \leq n \log n$.

Worst-Case-Rechenzeit: $O(n \log n)$.

9.4 Das Master-Theorem

Wir betrachten Rekurrenzungleichungen der folgenden Form:

$$B(n) \leq \begin{cases} g & \text{, falls } n \leq n_0 \\ a \cdot B(n/b) + f(n) & \text{, sonst.} \end{cases}$$

Dabei: $a \geq 1$ und $b > 1$ und g und n_0 sind Konstante,
$f : \mathbb{N} \to \mathbb{N}$ ist eine monoton wachsende Funktion.

Falls n/b keine ganze Zahl ist, sollte man sich an Stelle von $B(n/b)$ z. B. $B(\lceil n/b \rceil)$ geschrieben denken. Es muss dann gelten: $n > n_0 \Rightarrow \lceil n/b \rceil < n$.

Ziel: „Geschlossene“ Abschätzung von $B(n)$ nach oben.

Gesehen: Eine solche Rekurrenzungleichung ergibt sich bei der Analyse eines Divide-and-Conquer-Algorithmus mit:

- Basisfall (Größe $\leq n_0$) hat höchstens Kosten g,
- aus Instanz der Größe $n > n_0$ werden a Teilinstanzen der Größe n/b (passend gerundet) gebildet („**teile**“),
- es erfolgen a **rekursive Aufrufe**,
- und die a Lösungen werden zusammengesetzt („**kombiniere**“).
- Kosten für das Aufspalten und das Kombinieren: $f(n)$.

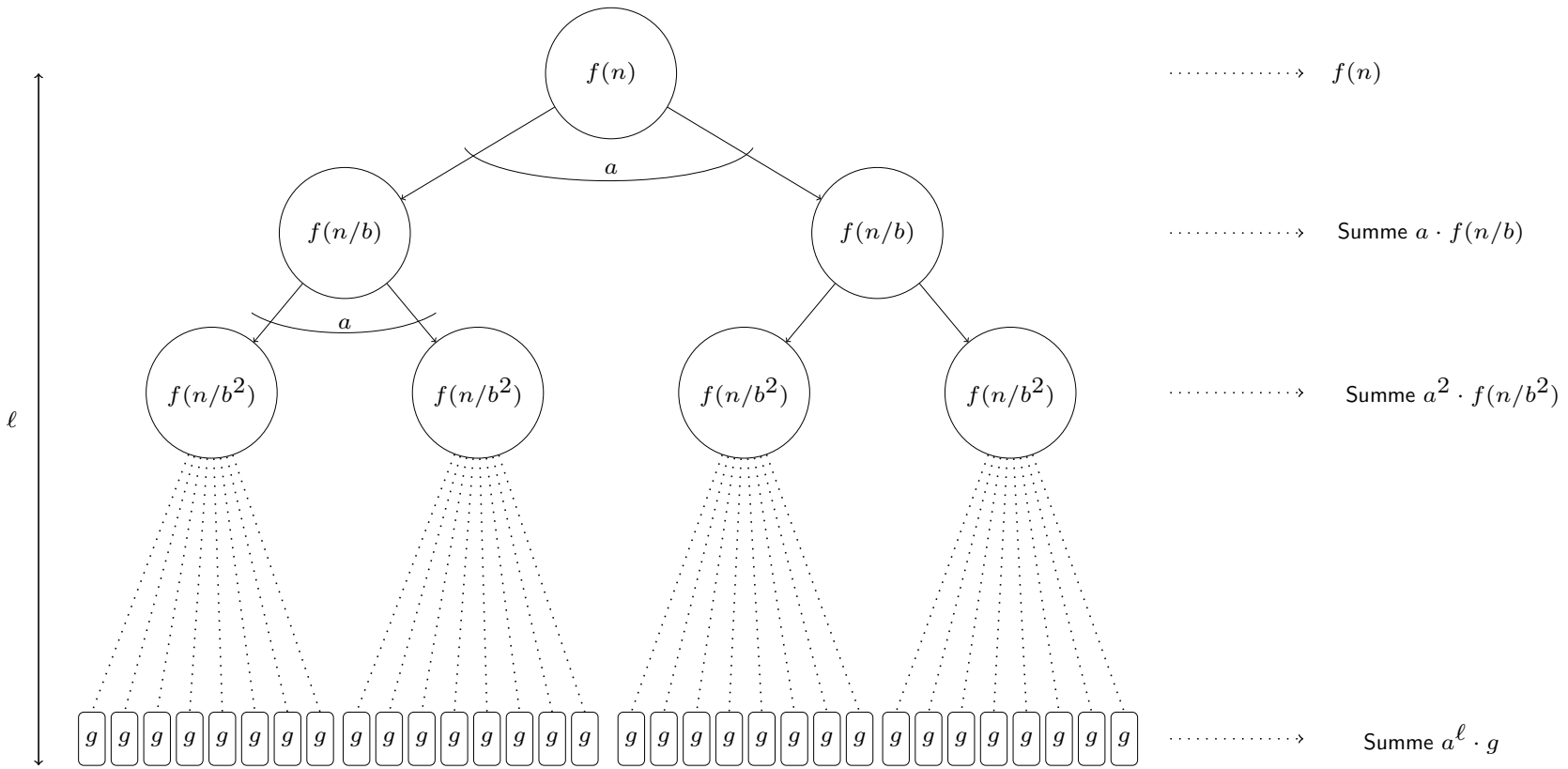
O.B.d.A.: $B(n)$ monoton wachsend. – Sonst definiere:

$$\hat{B}(n) := \max\{B(i) \mid 1 \leq i \leq n\}, \text{ für } n \geq 0.$$

$\Rightarrow \hat{B}(n)$ ist monoton und erfüllt dieselbe Rekurrenzungleichung, und $B(n) \leq \hat{B}(n)$.

Vereinfachende Annahmen (nicht wesentlich): $n_0 = 1$ und $n = b^\ell$.
$b > 1$ und $a \geq 1$, beide sind ganzzahlig.

„Rekursionsbaum“: Veranschaulicht Kostenaufteilung, (mathematisch nicht notwendig).
Level 0: Wurzel, hat Eintrag $f(n)$ und hat a Kinder auf Level 1.
Knoten v auf Level $i < \ell$ hat Eintrag $f(n/b^i)$ und hat a Kinder auf Level $i+1$.
Knoten auf Level ℓ sind Blätter, sie haben Eintrag g.



Lemma 9.4.1

Wenn v ein Knoten auf Level i ist, dann gilt:

$B(n/b^i) \leq$ Summe der Einträge im Unterbaum unter v.

(Beweis durch Induktion über $i = \ell, \ell - 1, \ldots, 1$.)

Also: $B(n) \leq$ Summe aller Einträge im Baum.

Auf Level i gibt es a^i Knoten mit Eintrag $f(n/b^i)$. Summation liefert:

$$\boxed{B(n) \leq \sum_{0 \leq i < \ell} a^i \cdot f(n/b^i) + a^\ell \cdot g.}$$

Erster Term $B_1(n)$: Beitrag zu Gesamtkosten aus dem Inneren des Baums.

Zweiter Term $B_2(n)$: Beitrag von den Blättern. (Algorithmisch: Die a^ℓ Basisfälle.)

Beobachte: $a^\ell = (b^{\log_b a})^\ell = (b^\ell)^{\log_b a} = n^{\log_b a}$, also $B_2(n) = O(n^{\log_b a})$.

Herleitung der Ungleichung in der Box durch iterierte Anwendung der Rekurrenzungleichung, ohne Veranschaulichung durch Baum (vgl. auch die konkreten Fälle in 6.1 und 6.2):

$$
\begin{aligned}
B(n) &\leq a \cdot B(n/b) + f(n) \\
&\leq a \cdot (a \cdot B((n/b)/b) + f(n/b)) + f(n) \\
&= a^2 \cdot B(n/b^2) + a \cdot f(n/b) + f(n) \\
&\leq a^3 \cdot B(n/b^3) + a^2 \cdot f(n/b^2) + a \cdot f(n/b) + f(n) \\
&\quad \vdots \\
&\leq a^j \cdot B(n/b^j) + a^{j-1} \cdot f(n/b^{j-1}) + \cdots + a \cdot f(n/b) + f(n) \\
&\quad \vdots \\
&\leq a^\ell \cdot \underbrace{B(n/b^\ell)}_{=B(1)=g} + \sum_{0 \leq i < \ell} a^i f(n/b^i) = B_2(n) + B_1(n).
\end{aligned}
$$

Man erkennt hier, dass man nicht annehmen muss, dass a ganzzahlig ist.

Erster Term: $B_1(n) = \sum_{0 \leq i < \ell} a^i \cdot f(n/b^i)$.

3 Fälle (je nach Verhalten des Gesamtaufwandes $a^i \cdot f(n/b^i)$ auf Level i, für $i = 0, \ldots, \ell - 1$):

Intuitiv:

1. Fall: $a^i \cdot f(n/b^i)$ wächst mit i an.

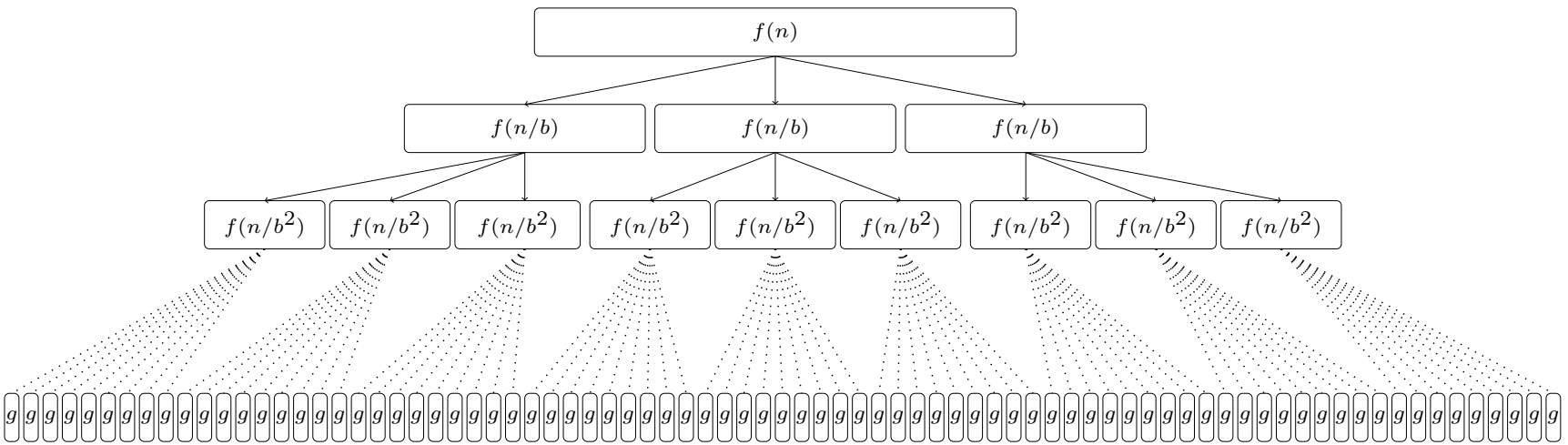
2. Fall: $a^i \cdot f(n/b^i)$ bleibt in etwa gleich über alle i.

3. Fall: $a^i \cdot f(n/b^i)$ schrumpft mit i.

Genaueres folgt.

1. Fall: $f(n) = O(n^\alpha)$ mit $b^\alpha < a$, oder, äquivalent $\alpha < \log_b a$.

Die Beiträge aus den unteren Baumebenen (kleine Instanzen) dominieren.
(Fallende Größe wird durch wachsende Anzahl überkompensiert. Im Bild: Die Breite des $f(n/b^i)$-Knotens ist $f(n/b^i)$.)



Wir benutzen mehrfach die Summenformel für geometrische Reihen: $\sum_{0 \le i < \ell} q^i = \frac{q^\ell - 1}{q-1}$, für $q \neq 1$, s. Kapitel 1, Folie 37. Für $q > 0$, $q \neq 1$, folgt:

$$\sum_{0 \le i < \ell} q^i < \frac{q^\ell}{q-1}. \qquad (*)$$

$$B_1(n) = \sum_{0 \le i < \ell} a^i \cdot f(n/b^i)$$

$$= O\Big(\sum_{0 \le i < \ell} a^i \cdot \Big(\frac{n}{b^i}\Big)^\alpha\Big) = O\Big(n^\alpha \cdot \sum_{0 \le i < \ell} \Big(\frac{a}{b^\alpha}\Big)^i\Big)$$

$$\overset{(*)}{=} O\left(n^\alpha \cdot \frac{(a/b^\alpha)^\ell}{(a/b^\alpha) - 1}\right) \qquad \text{(geom. Reihe)}$$

$$= O\left(n^\alpha \cdot a^\ell \cdot \frac{1}{(b^\ell)^\alpha}\right) \qquad \text{(Nenner ist konstant)}$$

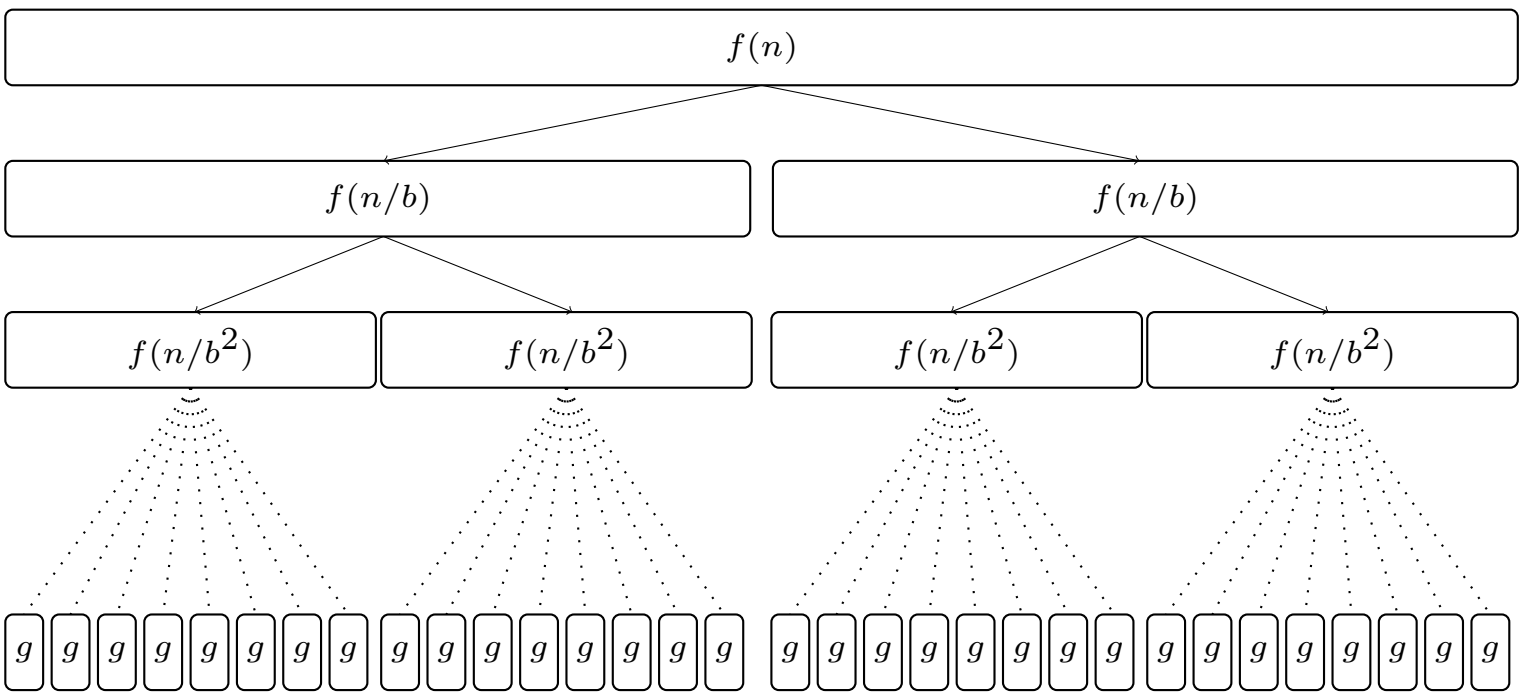
$$= O(a^\ell). \qquad (b^\ell = n, \text{ Kürzen})$$

Also: $B(n) \le B_1(n) + B_2(n) = O(a^\ell) = O(n^{\log_b a})$.

Beispiele: **Karatsuba**-Algorithmus ($b = 2$, $a = 3$, $f(n) = O(n)$),
Strassen-Algorithmus ($b = 2$, $a = 7$, $f(n) = O(n^2)$).

2. Fall: $f(n) = O(n^{\log_b a})$, oder, äquivalent, $f(n) = O(n^\alpha)$ mit $\alpha = \log_b a$.

$f(n/b^i)$ wächst mit n/b^i, $i = \ell - 1, \ldots, 0$, höchstens mit einer Rate, die durch das Schrumpfen der Größe der Baumebene ausgeglichen wird. Der Gesamtaufwand ist beschränkt durch den Aufwand für die Ebene direkt über den Blättern, multipliziert mit der Anzahl der Levels.



$$
\begin{aligned}
B_1(n) &= \sum_{0 \le i < \ell} a^i \cdot f(n/b^i) \\
&= O\left(\sum_{0 \le i < \ell} a^i \cdot \left(\frac{n}{b^i}\right)^{\log_b a} \right) \\
&= O\left(\sum_{0 \le i < \ell} a^i \cdot \frac{n^{\log_b a}}{a^i} \right) \\
&= O\left(n^{\log_b a} \cdot \ell\right) \\
&= O(n^{\log_b a} \cdot \log n).
\end{aligned}
$$

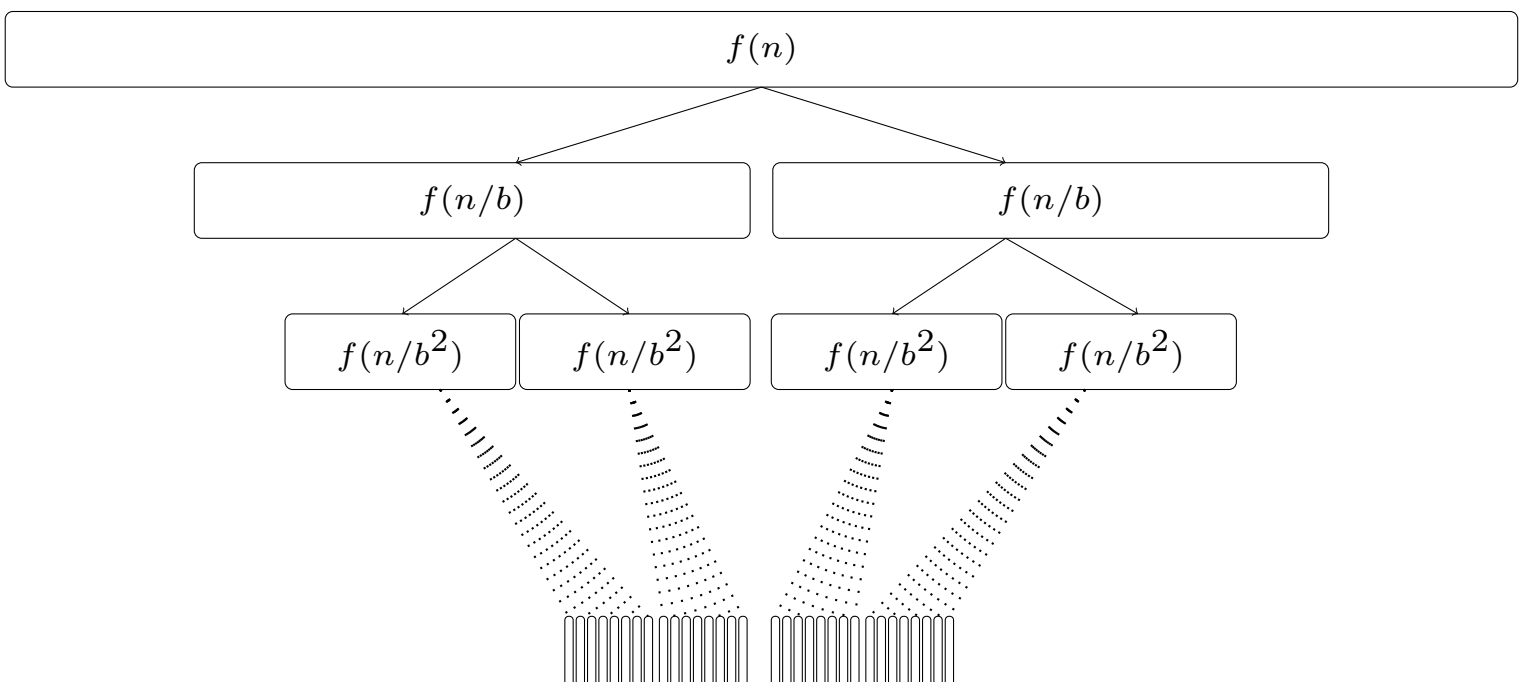
Also: $B(n) \le B_1(n) + B_2(n) = O(n^{\log_b a} \cdot \log n) + O(n^{\log_b a}) = O(n^{\log_b a} \cdot \log n)$.

Beispiele: **Mergesort** ($b = a = 2$), Binäre Suche ($b = 2$, $a = 1$, $\log_b a = 0$, $n^0 = 1$).

3. Fall: $f(n) = \Omega(n^\alpha)$, mit $b^\alpha > a$

und es gibt $c < 1$ mit: $f(n/b) \leq (c/a) \cdot f(n)$, für alle $n \geq 1$ („Regularitätsbedingung“).

Wenn die Größe des Inputs von n auf n/b sinkt, fallen die Kosten im Baumknoten garantiert mindestens um den Faktor c/a. $f(n/b^i)$ fällt dann so rasch mit wachsendem i, dass die wachsende Anzahl a^i der Baumknoten (über)kompensiert wird, und die unteren Baumebenen vernachlässigbares Gewicht haben.



Mit der Regularitätsbedingung erhalten wir nacheinander:
$f(n/b) \leq \frac{c}{a} \cdot f(n)$, $f(n/b^2) \leq \left(\frac{c}{a}\right)^2 \cdot f(n)$, $\ldots$, $f(n/b^i) \leq \left(\frac{c}{a}\right)^i \cdot f(n)$. Damit:

$$
\begin{aligned}
B_1(n) &= \sum_{0 \leq i < \ell} a^i \cdot f(n/b^i) \\
&\leq \sum_{0 \leq i < \ell} a^i \cdot \left(\frac{c}{a}\right)^i \cdot f(n) \\
&= \sum_{0 \leq i < \ell} c^i \cdot f(n) \\
&= f(n) \cdot \sum_{0 \leq i < \ell} c^i \\
&= O(f(n)),
\end{aligned}
$$

weil $\sum_{0 \leq i < \ell} c^i = \frac{1-c^\ell}{1-c} = O(1)$.

Satz 9.4.2 Das Master-Theorem (Einfache Form)

Es gelte $B(n) \leq \begin{cases} g & \text{, falls } n = 1 \\ a \cdot B(n/b) + f(n) & \text{, sonst,} \end{cases}$

für ganzzahlige konstante $b > 1$ und $a > 0$. Dann gilt für $n = b^\ell$:

1. Fall: $f(n) = O(n^\alpha)$ mit $\alpha < \log_b a$. Dann: $B(n) = O(n^{\log_b a})$.

2. Fall: $f(n) = O(n^\alpha)$ mit $\alpha = \log_b a$. Dann: $B(n) = O(n^{\log_b a} \cdot \log n)$.

3. Fall: $f(n) = \Omega(n^\alpha)$ mit $\alpha > \log_b a$ **und** $f(n/b) \leq \frac{c}{a} \cdot f(n)$, für $c < 1$ konstant.
Dann gilt $B(n) = O(f(n))$.

Verschiedene Einschränkungen, die wir im Beweis benutzt haben, sind nicht wirklich nötig. Dieselben Formeln gelten in folgenden Fällen:

- Beliebige $n \in \mathbb{N}$, nicht nur $n = b^\ell$.
- Verallgemeinerte Relation
  $B(n) \leq a \cdot B(n') + f(n)$, $n' \leq \lceil n/b \rceil + d$, für d konstant.
- $b > 1$, $a > 0$ beliebig reell, Schranke n_0 in der Basisbedingung,
  wenn nur $\lceil n/b \rceil + d < n$ für $n > n_0$.
- Analoge untere Schranken.

Genaueres dazu kann man in den Büchern von [D./Mehlhorn/Sanders] oder [Cormen, Leiserson, Rivest und Stein] nachlesen.

9.5 Das Auswahlproblem („selection“)

Beispiel: Finde den drittkleinsten Eintrag in $(7, 3, 8, 5, 4, 9, 2, 10, 4, 8, 7, 12, 9, 5)$!

Gegeben ist eine Folge $(a_1, \ldots, a_n)$ von n Objekten aus einer totalen Ordnung $(D, <)$ (in Array oder als Liste), sowie eine Zahl k, $1 \leq k \leq n$.

Aufgabe: Finde das Element x der Folge, das **Rang k** hat, d. h. das Objekt x in der Liste mit $|\{i \mid a_i < x\}| < k \leq |\{i \mid a_i \leq x\}|$.

Spezialfall:
Der **Median** einer Folge mit n Einträgen ist das Element mit Rang $\lceil n/2 \rceil$.
(Median$((2, 4, 7, 9)) = 4$, Median$((4, 7, 9)) = 7$.)

Einfache Lösung: Sortiere die Folge, mit Ergebnis $(b_1, \ldots, b_n)$, dann wähle $x = b_k$.
Kosten hierfür: $n \log n$ Vergleiche, Zeit $\Theta(n \log n)$.

Wir betrachten einen **randomisierten Algorithmus** für das Auswahlproblem:

Quickselect (C. A. R. („Tony“) Hoare, wie Quicksort)

Ansatz: Wie bei Quicksort.

Input: Folge $(a_1, \ldots, a_n)$, Zahl k, $1 \leq k \leq n$.

Vereinfachende Annahme, vorerst: Die a_i sind verschieden.

Falls $n = 1$, ist nichts zu tun.

Falls $n = 2$, sortiere mit einem Vergleich, Ergebnis (b_1, b_2), gib Element b_k zurück.

Falls $n \geq 3$:

Wähle einen Eintrag x aus $(a_1, \ldots, a_n)$ als partitionierendes Element **zufällig**.

Zerlege $(a_1, \ldots, a_n)$ mit $n-1$ Vergleichen in eine Teilfolge $(b_1, \ldots, b_{p-1})$, alle $< x$, in das Element x, und eine Teilfolge $(c_{p+1}, \ldots, c_n)$, alle $> x$.

1. Fall: $p = k$.
Das Ergebnis ist x.

2. Fall: $p > k$.
Finde (rekursiv) in $(b_1, \ldots, b_{p-1})$ das Element vom Rang k.

3. Fall: $p < k$.
Finde (rekursiv) in $(c_{p+1}, \ldots, c_n)$ das Element vom Rang $k - p$.

Prozedur rqSelect(a, b, k)

// Rekursive Prozedur im Quickselect-Algorithmus, $1 \leq a \leq b \leq n$, $a \leq k \leq b$.
// Vorbedingung: Alle Einträge vom Rang $< a$ [$> b$] links [rechts] von A$[a..b]$
// Nachbedingung: Eintrag vom Rang k in A$[k]$, kleinere links, größere rechts davon.

(1) **if** $a = b\ (= k)$ **then return**;
(2) s $\leftarrow$ ein zufälliges Element von $\{a, \ldots, b\}$;
(3) **if** $(a <$ s$)$ **then** vertausche A$[a]$ und A[s];
(4) **partition**$(a, b,$ p$)$; // wie bei rqsort, Abschnitt 6.3, Folie 30
(5) **if** $k =$ p **then return**;
(6) **if** $k <$ p **then rqSelect**$(a,$ p $- 1, k)$
(7) **else rqSelect**$($p $+ 1, b, k)$.

Mögliche Anpassungen:
(a) Sortiere z.B. mit Insertionsort, wenn $b - a$ sehr klein ist.
(b) Anstelle von Rekursion benutze Iteration.

Korrektheit: Klar.

Zu analysieren: (Erwartete) Rechenzeit.

Sei n die Eingabelänge, $k \in \{1, \ldots, n\}$ der gegebene Rang.

Klar: Rechenzeit ist proportional zur erwarteten Anzahl $\mathbf{E}(\boldsymbol{C_{n,k}})$ von Vergleichen in dieser Situation.

(Wahrscheinlichkeitsmodell: Zufällige Pivotwahl.)

Vernachlässige die Abhängigkeit von k:

$\boldsymbol{U_n} := \max\{\mathbf{E}(C_{n,k}) \mid 1 \leq k \leq n\}$.

Klar: $U_1 = 0$, $U_2 = 1$.

Behauptung: $U_n \leq c \cdot n$, für eine Konstante $c > 0$.

Beweis: Induktion über n.

(Trick: Wir wählen c **später** so, dass der Beweis funktioniert. Der Induktionsanfang ist mit der Bedingung $c \geq 1$ schon gesichert.)

Sei $n \geq 2$.

Das Pivotelement hat Rang $p \in \{1, \ldots, n\}$, jedes p mit Wahrscheinlichkeit $\frac{1}{n}$.

Das Partitionieren kostet $n - 1$ Vergleiche.

1. Fall: $p = k$. Fertig.

2. Fall: $k < p$. – Rekursion in $(b_1, \ldots, b_{p-1})$.

Erwartete Vergleichsanzahl ist $\leq U_{p-1} \leq c(p-1)$, nach I.V.

3. Fall: $k > p$. – Rekursion in $(c_{p+1}, \ldots, c_n)$.

Erwartete Vergleichsanzahl ist $\leq U_{n-p} \leq c(n-p)$, nach I.V.

Also . . .

$$\begin{aligned}\mathbf{E}(C_{n,k}) &< n + \frac{1}{n}\sum_{k<p\leq n} c(p-1) + \frac{1}{n}\sum_{1\leq p<k} c(n-p) \\ &< n + \frac{c}{n}\left((n-k)\cdot\frac{k+(n-1)}{2} + (k-1)\frac{(n-1)+(n-(k-1))}{2}\right).\end{aligned}$$

Benutzt wurde die Formel für arithmetische Reihen:
$a + (a+d) + (a+2d) + \cdots + (a+(r-1)d) = r \cdot \frac{1}{2}(a + (a+(r-1)d))$.
(Anzahl Summanden mal Mittelwert des ersten und letzten Summanden!)

$$\mathbf{E}(C_{n,k}) < n + \frac{c((n-k)(k+n-1) + (k-1)(2n-k))}{2n}.$$

Ausmultiplizieren:

$$\mathbf{E}(C_{n,k}) \leq n + \frac{c(n^2 - 3n - 2k^2 + 2k + 2nk)}{2n}.$$

Wähle k so, dass rechte Seite maximal wird, nämlich $k = \frac{n+1}{2}$:

$$U_n \leq n + \frac{c \cdot (\frac{3}{2}n^2 - 2n + \frac{1}{2})}{2n} < (1 + (3c/4))n.$$

Ohne c zu kennen, geht es nicht weiter.

Wähle $c = 4$, verstärke Ind.-Beh. zu: $U_n \leq 4n$.

Trivial für $n = 1$ und $n = 2$.

Induktionsschritt: $U_n \overset{\text{I.V.}}{\leq} (1 + (3c/4))n = (1 + 3)n = 4n = cn$. Fertig! □

Mitteilungen:

(a) Eine genauere Analyse ergibt für $\alpha = k/n$ konstant eine erwartete Vergleichsanzahl von $2(1 + H(\alpha)\ln 2)n < (3{,}3863 + o(1)) \cdot n$.

Dabei ist $H(\alpha) = -\alpha \log \alpha - (1-\alpha)\log(1-\alpha)$ die **„binäre Entropie“** der Wahrscheinlichkeitsverteilung $(\alpha, 1-\alpha)$.
$H(\alpha)$ liegt zwischen 0 und 1; das Maximum 1 ist bei $\alpha = \frac{1}{2}$, was der Suche nach dem Median entspricht.
$2(1 + \ln 2) \approx 3{,}386294.$

(b) Quickselect ist in der Praxis sehr, sehr schnell.
Achtung! Falls es identische Einträge gibt, muss man entsprechende Vorkehrungen treffen (3-Wege-Partition, s. Kapitel 6, Folie 42).

(c) Die beste theoretische Schranke für die erwartete Vergleichsanzahl bei Algorithmen für das Auswahlproblem, nämlich $\frac{3}{2}n + o(n)$, erreicht ein anderer randomisierter Algorithmus. („Random Sampling“, siehe Vorlesung „Randomisierte Algorithmen“.)

Nun: Ein **deterministischer** Algorithmus mit Aufwand $O(n)$.

(Erfinder: M. **Blum**, R. W. **Floyd**, V. R. **Pratt**, R. L. **Rivest**, R. E. **Tarjan**: lauter Pioniere der Algorithmik!)

„Stufenweises“ Divide-and-Conquer. – Wie bei Quickselect:

- Finde ein partitionierendes Element x.
- Verschiebe Einträge im Array, so dass alle Elemente $< x$ links von x stehen, alle Elemente $> x$ rechts.
- Lese ab, in welchem Teil das Element vom Rang k sitzt.
- Rufe den Algorithmus rekursiv auf diesem Teil auf.

Zentral: **Deterministisch**, unter Benutzung von Rekursion ein „günstiges“ Element x best

Algorithmus BFPRT$(a_1, \ldots, a_n, k)$ // O.B.d.A.: $a_1, \ldots, a_n$ verschieden

(0) Falls $k = 1$ oder $k = n$: Bestimme Minimum/Maximum direkt. **Sonst**:
(1) Falls $n \leq n_0$: Sortiere mit Mergesort, fertig. **Sonst**:
(2) Teile $(a_1, \ldots, a_n)$ in $m = \lceil n/5 \rceil$ Gruppen mit 4 bzw. 5 Elementen auf.
(3) Bestimme in jeder Gruppe den Median (z. B. mit Mergesort).
Sei $(a_1^*, \ldots, a_m^*)$ die Liste dieser Mediane.
(4) Suche mit **BFPRT rekursiv** den **Median** x von $(a_1^*, \ldots, a_m^*)$.
(5) Zerlege $(a_1, \ldots, a_n)$ in eine Teilfolge $b_1, \ldots, b_{p-1}$, alle $< x$, in das Element x, und eine Teilfolge $c_{p+1}, \ldots, c_n$, alle $> x$.
(6) Falls $k = p$: Rückgabe x.
(7) Falls $k < p$: **BFPRT**$(b_1, \ldots, b_{p-1}, k)$. // **Rekursion**
(8) Falls $k > p$: **BFPRT**$(c_{p+1}, \ldots, c_n, k - p)$ // **Rekursion**

Das Pivotelement x bezeichnet man als „*Median der Mediane*“.
Dieses partitionierende Element wird in den Schritten (2)–(4) gefunden.

Korrektheit: Klar, durch Induktion über rekursive Aufrufe.

Laufzeit: Die Laufzeit ist proportional zur Anzahl der durchgeführten Vergleiche.

Wir definieren: $\boldsymbol{C(n)}$:= maximale Anzahl der Vergleiche bei Aufruf

BFPRT$(a_1, \ldots, a_\ell, k)$, $\ell \leq n$, $1 \leq k \leq \ell$.

(Durch Maximieren über „$\ell \leq n$“ wird die Funktion $C(n)$ monoton.)

Zeile (0) (Fall $k \in \{1, n\}$) benötigt $n - 1$ Vergleiche, und wir sind fertig.

Zeile (1) (Fall $n \leq n_0$) benötigt höchstens $n \log n$ Vergleiche.

Zeile (3): Median von 5 Einträgen: mit Mergesort in 8 Vergleichen zu ermitteln.

Direktes Verfahren (Übung) findet Median mit 6 Vergleichen.

Für 4 Einträge genügen 4 Vergleiche.

Zeile (3) benötigt daher $\leq 6n_5$ Vergleiche für die n_5 Fünfergruppen und $4n_4$ Vergleiche für die $n_4 \leq 4$ Vierergruppen.

Da $n = 5n_5 + 4n_4$, ist dies zusammen $\leq 6n/5$.

Zeile (4): Höchstens $C(\lceil n/5 \rceil)$ Vergleiche.

Zeile (5): Exakt $n - 1$ Vergleiche, ebenso wie bei Quickselect.

Zeilen (7)/(8): Es wird nur eine dieser beiden Zeilen ausgeführt. Wir zeigen, dass in beiden Fällen die Anzahl der beteiligten Einträge nicht größer als $7n/10 + 4$ ist.

1. Fall: $p > k$. Bei der rekursiven Suche nach dem Eintrag mit Rang k werden alle Einträge $\geq x$ weggelassen.

Wie viele Einträge sind dies **mindestens**?

Mit $G_1, \ldots, G_m$ bezeichnen wir die Gruppen (4 oder 5 Elemente).
a_j^* ist der Median von G_j. Dann definieren wir:

$$A := \{a_i \mid \exists j : a_i \text{ in Gruppe } G_j \text{ und } a_j^* \geq x \text{ und } a_i \geq a_j^*\}.$$

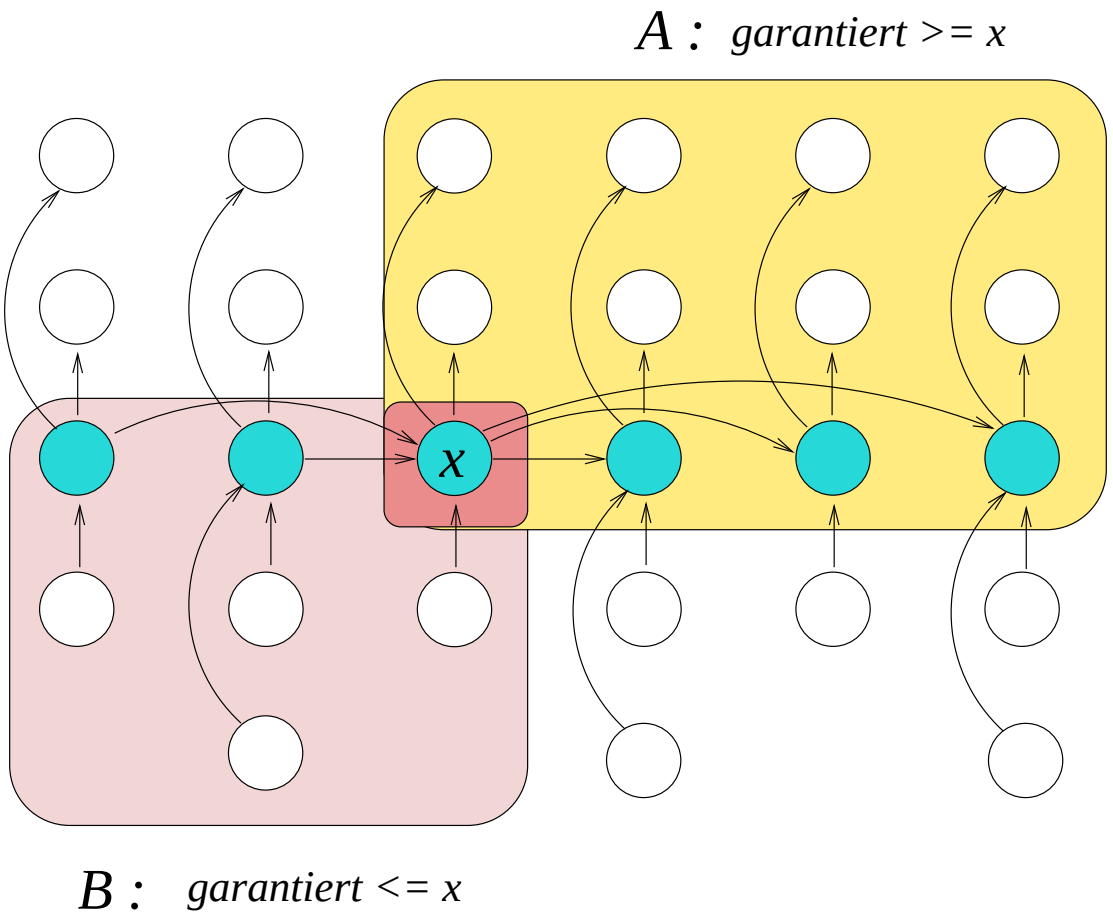
Dann sind alle Elemente von A mindestens so groß wie x (s. Bild).

Also $n - p + 1 \geq |A| \geq 3(\lfloor m/2 \rfloor + 1) \geq 3n/10$, also $p - 1 \leq 7n/10$.

Die Kosten für den rekursiven Aufruf in Zeile (7) sind also maximal $C(\lfloor 7n/10 \rfloor)$.

Spalten: Gruppen nach Umsortieren, Kriterium: a_j^* $</=/>x$

Innerhalb jeder Spalte: unten kleiner als der Median, oben größer



2. Fall: $p < k$. Bei der rekursiven Suche nach dem Eintrag mit Rang k werden alle Einträge $\leq x$ weggelassen.

Definiere

$$B := \{a_i \mid \exists j : a_i \text{ in Gruppe } G_j \text{ und } a_j^* \leq x \text{ und } a_i \leq a_j^*\}.$$

Dann sind alle Elemente von B höchstens so groß wie x (s. Bild).

Weil es mindestens $\lceil m/2 \rceil$ Gruppen G_j mit $a_j^* \leq x$ und maximal 4 Gruppen mit 4 Elementen gibt, folgt:

$p \geq |B| \geq 3\lceil m/2 \rceil - 4 \geq 3n/10 - 4$,

also betrifft der rekursive Aufruf in Zeile (8) höchstens $n - p \leq 7n/10 + 4$ Einträge.

Die Kosten sind also maximal $C(\lfloor 7n/10 \rfloor + 4)$.

Wir erhalten die folgende Rekurrenzungleichung:

$$C(n) \leq \begin{cases} n \log n & \text{für } n \leq n_0, \\ C(\lceil n/5 \rceil) + C(\lfloor 7n/10 \rfloor + 4) + 11n/5 & \text{für } n > n_0. \end{cases}$$

Dabei schätzt der Term $11n/5$ die Beiträge von Zeilen (3) und (5) zusammen ab.

Leider: Unser Mastertheorem nicht anwendbar.

Wir lösen die Rekurrenz direkt, indem wir eine passende Induktionsbehauptung beweisen.

Behauptung:
$C(n) \leq cn$ für alle n und eine passende Konstante c.

Die $n \leq n_0$ werden erledigt, indem man $c \geq \log n_0$ wählt.

Konkret: $n_0 = 500$; jedes $c \geq 9$ erfüllt die Behauptung in diesem Fall.

Nun sei $n > n_0$. Wir rechnen:

$$\begin{aligned} C(n) &\leq C(\lceil n/5 \rceil) + C(\lfloor 7n/10 \rfloor + 4) + 11n/5, \\ &\overset{\text{I.V.}}{\leq} c\lceil n/5 \rceil + c(7n/10 + 4) + 11n/5 \\ &\leq cn/5 + c + 7cn/10 + 4c + 11n/5 \\ &\leq cn + (-cn/10 + 5c + 11n/5). \end{aligned}$$

Entscheidend: $C(n) \leq \frac{9}{10}cn + O(n)$.

Wir wählen c so, dass $cn/10 \geq 5c + 11n/5$ ist, was für $c \geq 25$ und $n \geq n_0 = 500$ der Fall ist (nachrechnen!).

Für ein solches c lässt sich der Induktionsschritt durchführen; damit gilt die Behauptung $C(n) \leq cn$ für alle n.

Wir haben gezeigt:

Satz 9.5.2

Der BFPRT-Algorithmus löst das Auswahlproblem und hat eine Laufzeit von $O(n)$ im schlechtesten Fall.

Bemerkung: (a) Durch eine viel genauere Analyse kann die Konstante in der Vergleichsanzahl noch verbessert werden.
(b) Der beste bekannte deterministische Algorithmus für das Auswahlproblem (anderer Ansatz!) benötigt $(2{,}95 + o(1))n$ Vergleiche. Es ist bekannt, dass jeder deterministische Algorithmus $\geq 2n$ Vergleiche benötigt.

9.6 Die schnelle Fourier-Transformation (FFT)

Aufgabe: Multiplikation von **Polynomen**, z. B.

$(5x^4 - 7x^3 - 4x + 2) \cdot (3x^2 + x + 1) = \ldots$

Polynom in Koeffizientendarstellung:

$$A(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1} = \sum_{0 \le i \le n-1} a_i x^i,$$

mit **Koeffizienten** $a_0, \ldots, a_{n-1} \in$ „Zahlenbereich“ $\mathbb{Z}$ oder $\mathbb{R}$ oder $\mathbb{C}$.

Für die Durchführung benötigen wir **komplexe Zahlen**. (Siehe Mathematikvorlesung.)

Am Schluss wird eine Alternative skizziert, die man für Berechnungen über $\mathbb{Z}$ benutzen kann, ohne dabei zu komplexen Zahlen überzugehen.

Aufgabe: Gegeben zwei Polynome $A(x)$ und $B(x)$, als

$$A(x) = \sum_{0 \le i \le d-1} a_i x^i \quad \text{und} \quad B(x) = \sum_{0 \le j \le d-1} b_j x^j,$$

berechne das **Polynomprodukt**

$$A(x) \cdot B(x) = C(x) = \sum_{0 \le k \le 2d-2} c_k x^k,$$

d. h. berechne die Koeffizienten $c_k = \sum_{i,j \,:\, i+j=k} a_i b_j$, für $0 \le k \le 2d-2$.

Beispiel: $(a_0 + a_1 x + a_2 x^2) \cdot (b_0 + b_1 x + b_2 x^2) =$
$\underline{a_0 b_0} + (\underline{a_0 b_1 + a_1 b_0}) x + (\underline{a_0 b_2 + a_1 b_1 + a_2 b_0}) x^2 + (\underline{a_1 b_2 + a_2 b_1}) x^3 + \underline{a_2 b_2} x^4.$

Die Folge $(c_0, \ldots, c_{2d-2})$ heißt auch **Konvolution** $(a_0, \ldots, a_{d-1}) \circ (b_0, \ldots, b_{d-1})$.

Naive Benutzung der Formel für die c_k liefert $\Theta(d^2)$-Algorithmus.

Ziel: $O(d \log d)$. – Methode: Divide-and-Conquer.

Zentraler Trick: Benutze eine weitere Darstellung von Polynomen, nämlich die

Stützstellen-Darstellung.

Betrachte Folge $(x_0, \ldots, x_{n-1}) \in \mathbb{C}^n$ von beliebigen **verschiedenen** „Stützstellen“, für $n \geq 1$.

Zu einem Polynom $A(x) = \sum_{0 \leq i \leq n-1} a_i x^i$ betrachten wir den zugehörigen „**Wertevektor**“ $(A(x_0), \ldots, A(x_{n-1}))$. – Umkehrung:

Fakt 9.6.1 („**Interpolation**“ von Polynomen über Körpern)

Zu jedem beliebigen Wertevektor $(r_0, \ldots, r_{n-1}) \in \mathbb{C}^n$ **gibt es genau ein** Polynom $A(x) = \sum_{0 \leq i \leq n-1} a_i x^i$, also genau einen Koeffizientenvektor $(a_0, \ldots, a_{n-1})$, mit $A(x_k) = r_k$ für $0 \leq k \leq n-1$.

Es ist also gleichgültig, ob man für die Darstellung eines Polynoms (vom Grad $< n$) seine Koeffizienten oder einen Wertevektor benutzt.

Beweis von Fakt 9.6.1: Betrachte die „**Vandermonde-Matrix**“

$$V(x_0, \ldots, x_{n-1}) = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix}.$$

Offensichtlich ist

$$\begin{pmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{pmatrix} = V(x_0, \ldots, x_{n-1}) \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}.$$

Die Matrix $V(x_0, \ldots, x_{n-1})$ hat Determinante $\prod_{0 \le k < \ell \le n-1} (x_\ell - x_k) \neq 0$, ist also regulär. Daher hat das Gleichungssystem

$$V(x_0, \ldots, x_{n-1}) \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} r_0 \\ r_1 \\ \vdots \\ r_{n-1} \end{pmatrix}$$

genau eine Lösung

$$\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = V(x_0, \ldots, x_{n-1})^{-1} \cdot \begin{pmatrix} r_0 \\ r_1 \\ \vdots \\ r_{n-1} \end{pmatrix}.$$

Bemerkung: Aus der Darstellung der n-fachen Polynomauswertung als Matrix-Vektor-Produkt folgt auch:

Die Umrechnung von $(a_0, \ldots, a_{n-1})$ in $(r_0, \ldots, r_{n-1})$ ist eine (bijektive) **lineare Abbildung** von $\mathbb{C}^n$ nach $\mathbb{C}^n$.

Zurück zu Polynommultiplikation: Um durch Interpolation die $2d-1$ Koeffizienten des Produktpolynoms

$$A(x) \cdot B(x) = C(x) = \sum_{0 \leq k \leq 2d-2} c_k x^k$$

zu erhalten, müssen wir mit mindestens $2d-1$ Argument-Werte-Paaren arbeiten. Aus technischen Gründen verwenden wir $n = 2^{\lceil \log(2d) \rceil} \geq 2d$ viele.

Algorithmenplan für die Polynommultiplikation:

Eingabe: Zwei Polynome $A(x)$ und $B(x)$ vom Grad $< d$ als Koeffizientenvektoren $(a_0, \ldots, a_{d-1})$ und $(b_0, \ldots, b_{d-1})$.

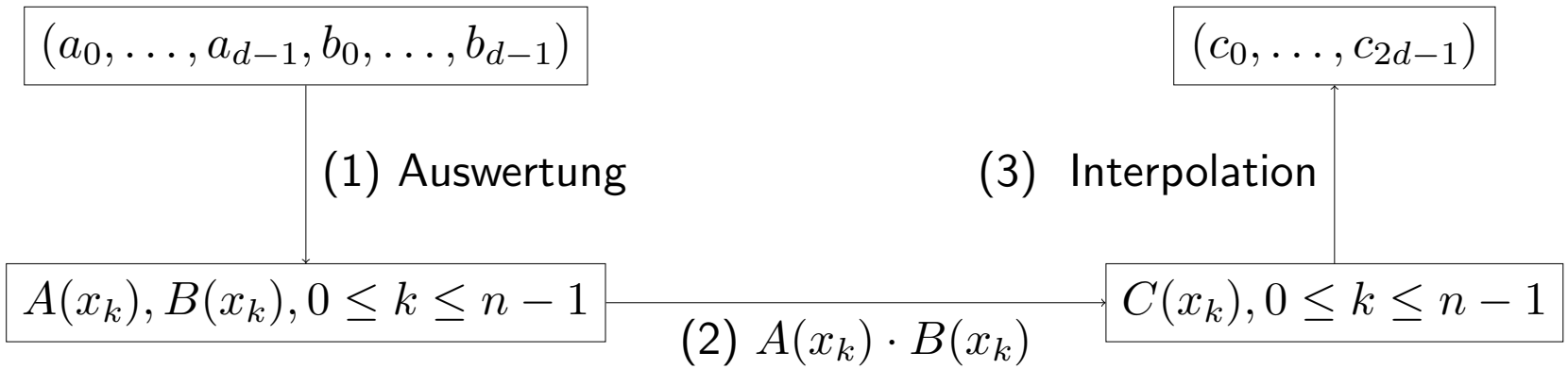
Setze $n = 2^{\lceil \log(2d) \rceil} \geq 2d$, Zweierpotenz, $2d \leq n < 4d$. Also: $O(n) = O(d)$.

$x_0, x_1, \ldots, x_{n-1}$ seien verschiedene Argumentwerte.

(1) Auswertung:
Berechne $A(x_k)$ und $B(x_k)$, für $k = 0, \ldots, n-1$.

(2) Berechne durch **punktweise Multiplikation** die n Werte des Produktpolynoms $C(x)$ an den Stützstellen:
$C(x_k) := A(x_k) \cdot B(x_k)$, für $k = 0, \ldots, n-1$.

(3) Interpolation:
Berechne aus $(C(x_0), C(x_1), \ldots, C(x_{n-1}))$ die Koeffizienten $(c_0, \ldots, c_{2d-1})$ von $C(x)$.



Kosten:

(1) **??** Naiv: Jeden Wert $A(x_k)$ separat berechnen, z. B. mit dem Horner-Schema: $A(x_k) = ((\ldots(a_{n-1} \cdot x_k + a_{n-2}) \cdot x_k \ldots) \cdot x_k + a_1) \cdot x_k + a_0$. Kosten: $O(d^2)$.

(2) Kosten: $O(d)$.

(3) **??** (Auch hier: $O(d^2)$ recht leicht zu erreichen.)

Unser Ziel: $O(d \log d)$ für (1) und (3).

Zunächst: **Auswertung**, für Grad $< n$ und $\boldsymbol{n = 2^L}$ Stützstellen.

Input: $A(x)$ als $(a_0, \ldots, a_{n-1})$, Stützstellen $(x_0, \ldots, x_{n-1})$.

Output: $(r_0, \ldots, r_{n-1}) = (A(x_0), \ldots, A(x_{n-1}))$.

Ansatz: **Divide-and-Conquer**.

Wenn $n = 1$, ist das Ergebnis $(r_0) = (a_0)$.

Wenn $n > 1$, teilen wir $A(x)$ in zwei Teilpolynome auf:

$$A(x) = (a_0 + a_2x^2 + a_4x^4 + \cdots + a_{n-2}x^{n-2}) + x(a_1 + a_3x^2 + a_5x^4 + \cdots + a_{n-1}x^{n-2}),$$

mit den Abkürzungen („**g**erade“, “**u**ngerade“)

$A_{\mathsf{g}}(x) := a_0 + a_2x + \cdots + a_{n-2}x^{n/2-1}$ und $A_{\mathsf{u}}(x) := a_1 + a_3x + \cdots + a_{n-1}x^{n/2-1}$
also

$$A(x) = A_{\mathsf{g}}(x^2) + xA_{\mathsf{u}}(x^2). \tag{1}$$

$$A(x) = A_{\mathsf{g}}(x^2) + xA_{\mathsf{u}}(x^2).$$

$A_{\mathsf{g}}(x^2)$ und $A_{\mathsf{u}}(x^2)$ haben jeweils nur noch $n/2$ Koeffizienten.
(Inputgröße halbiert!)

Originalproblem: Werte das Polynom $A(x)$ an den Stützstellen $x_0, \ldots, x_{n-1}$ aus.

Müssen nun rekursiv $A_{\mathsf{g}}(x_j^2)$ und $A_{\mathsf{u}}(x_j^2)$ berechnen, für $0 \leq j < n$.

Das passt aber nicht mit der Rekursion zusammen.
(Ein Polynom mit $n/2$ Koeffizienten muss an $n/2$ Stellen ausgewertet werden.)

Zentraler Trick: Sorge dafür, dass

$$x_0^2, \ldots, x_{n-1}^2$$

nur $n/2$ verschiedene Stützstellen sind.

Wie kann das funktionieren?

Wähle $x_0, \ldots, x_{n/2-1}$ verschieden und $\neq 0$ und betrachte als Stützstellen

$$\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1},$$

mit folgende Anordnung im Stützstellenvektor:

$$(x_0, \ldots, x_{n/2-1}, -x_0, \ldots, -x_{n/2-1}).$$

Vorerst nehmen wir an, dass die Stützstellen auf jeder Ebene der Rekursion diese Eigenschaft erfüllen.

Auswertung des Polynoms $A(x) = \sum_{0 \le j < n} a_j x^j$ auf Stützstellen $(x_0, \ldots, x_{n-1})$:

Teile: Stelle Koeffizientenvektoren $(a_0, a_2, \ldots, a_{n-2})$ von $A_{\mathsf{g}}(x)$ und $(a_1, a_3, \ldots, a_{n-1})$ von $A_{\mathsf{u}}(x)$ zusammen
und finde $(x'_0, x'_1, \ldots, x'_{n/2-1}) = (x_0^2, x_1^2, \ldots, x_{n/2-1}^2)$.

Rekursion: Werte $A_{\mathsf{g}}(x)$ mit $(x'_0, x'_1, \ldots, x'_{n/2-1})$ und $A_{\mathsf{u}}(x)$ mit $(x'_0, x'_1, \ldots, x'_{n/2-1})$ rekursiv aus. Resultat: Zwei Wertevektoren

$$A_{\mathsf{g}}(x_0^2), A_{\mathsf{g}}(x_1^2), \ldots, A_{\mathsf{g}}(x_{n/2-1}^2) \quad \text{und}$$

$$A_{\mathsf{u}}(x_0^2), A_{\mathsf{u}}(x_1^2), \ldots, A_{\mathsf{u}}(x_{n/2-1}^2).$$

Kombiniere: Berechne nun:

$$A(x_j) = A_{\mathsf{g}}(x_j^2) + x_j \cdot A_{\mathsf{u}}(x_j^2), \text{ für } 0 \le j < n/2;$$

$$A(x_{n/2+j}) = A_{\mathsf{g}}(x_j^2) - x_j \cdot A_{\mathsf{u}}(x_j^2), \text{ für } 0 \le j < n/2.$$

Beobachte: $n/2$ Multiplikationen und n Additionen im Kombinierschritt.

Unter der Annahme, dass auf jeder Rekursionsebene geeignete Stützstellen gefunden werden, schätzen wir die Laufzeit ab:

$C(n)$: Rechenzeit bei Eingaben der Länge n, vernachlässige konstante Faktoren.

Rekurrenzungleichung:

$$C(n) \leq \begin{cases} 1 & \text{, falls } n = 1 \\ 2 \cdot C(n/2) + cn & \text{, sonst,} \end{cases}$$

für eine Konstante c. Mit dem Master-Theorem, 2. Fall, ergibt sich

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

Problem: Auf oberster Rekursionsstufe: $\pm x_0, \ldots, \pm x_{n/2-1}$ als „Plus-Minus“-Paare gewählt.

Im rekursiven Aufruf: $x_0^2, \ldots, x_{n/2-1}^2$ müssen wieder in „Plus-Minus“-Paare aufgeteilt werden.

Wenn die Stützstellen reell sind, kann dies nicht funktionieren: Quadrate sind nie negativ.

Trick: Nutze komplexe Zahlen!

ω sei **primitive n-te Einheitswurzel** in $\mathbb{C}$, d. h.

(i) $\omega^n = 1$,

(ii) für $1 \leq k \leq n-1$ gilt $\sum_{0 \leq j \leq n-1} (\omega^k)^j = 0$.

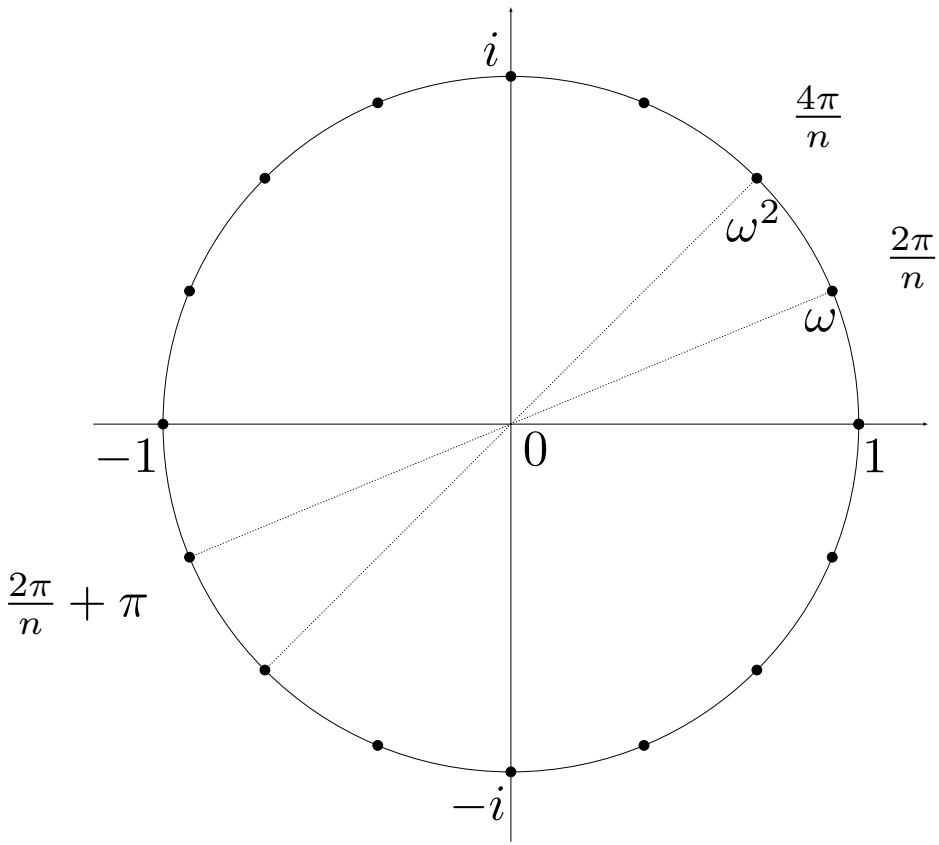
(iii) $\omega^k \neq 1$, für $1 \leq k \leq n-1$.

Unter einfachen weiteren Voraussetzungen sind (ii) und (iii) äquivalent:

– Wenn $\omega^k = 1$ ist, dann folgt $\sum_{0 \leq j \leq n-1} (\omega^k)^j = n$. Daher folgt (iii) aus (ii) (und $n \neq 0$).

– Wenn $\omega^k \neq 1$ gilt, schreiben wir: $(\omega^k - 1)(\sum_{0 \leq j \leq n-1} (\omega^k)^j) = (\omega^k)^n - 1 = 0$.

Daher folgt (ii), wenn (iii) gilt und zudem die Inversen $(\omega^k - 1)^{-1}$ existieren.



In $\mathbb{C}$: Eine primitive n-te Einheitswurzel ist

$$\omega := e^{\frac{2\pi i}{n}} = \cos\left(\frac{2\pi}{n}\right) + i \sin\left(\frac{2\pi}{n}\right),$$

wobei hier i die imaginäre Einheit ist. In der komplexen Zahlenebene liegt der Punkt ω also auf dem Einheitskreis, von 1 aus um den Winkel $2\pi/n$ gegen den Uhrzeigersinn verdreht.
Die Potenzen ω^k, $0 \leq k \leq n-1$, liegen in gleichen Abständen auf dem Einheitskreis.

Wieso ist ω primitive n-te Einheitswurzel?

(i) $\omega^n = (e^{\frac{2\pi i}{n}})^n = e^{2\pi i} = 1.$

(ii) Für jedes beliebige y gilt

$$\sum_{0 \le j \le n-1} y^j = (1+y)(1+y^2)(1+y^4)\cdots(1+y^{2^{L-1}}). \tag{2}$$

(Ausmultiplizieren des Produkts ergibt $n = 2^L$ Summanden y^j, bei denen jeder Exponent $j \in \{0, \ldots, 2^L - 1\}$ genau einmal vorkommt, wegen der Eindeutigkeit der Binärdarstellung.)

Wir betrachten (2) für $y = \omega^k$, für $1 \le k \le n-1$ beliebig.

Schreibe $k = u \cdot 2^\ell$, mit u ungerade und $0 \le \ell < L$. Dann ist

$$(\omega^k)^{2^{L-\ell-1}} = \omega^{u(2^{L-1})} = (\omega^{n/2})^u = (-1)^u = -1,$$

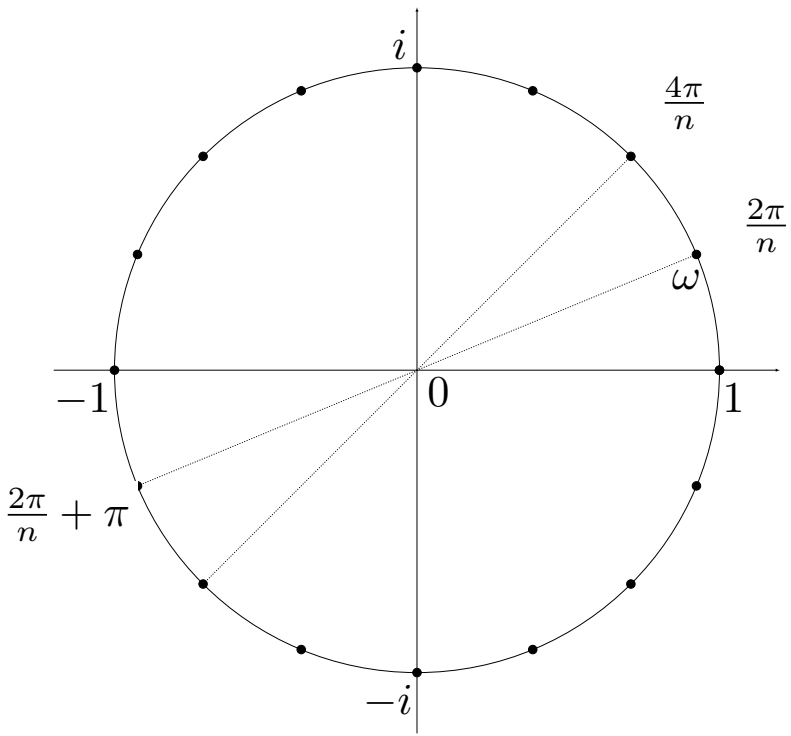
weil u ungerade ist. Also ist der Faktor $(1 + (\omega^k)^{2^{L-\ell-1}})$ in (2) gleich 0, also ist $\sum_{0 \le j \le n-1} (\omega^k)^j = 0$.

Sei $n = 2^L$ und sei ω eine beliebige primitive n-te Einheitswurzel. Dann ist unser Problem mit den $\pm$-Paaren gelöst:

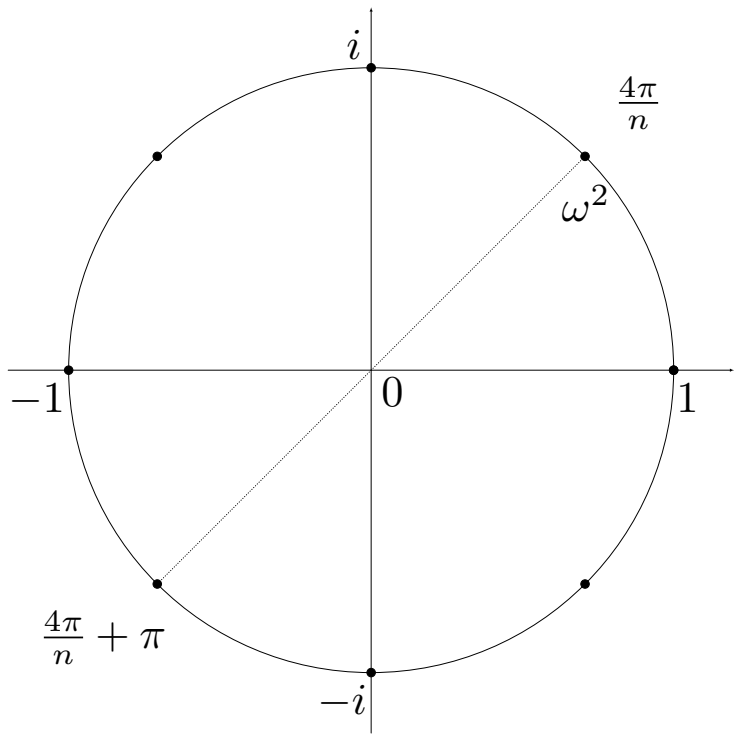
- $\omega^0 = 1, \omega^{n/2} = -1$.
- Für $j \in \{0, \ldots, n/2 - 1\}$ gilt: $\omega^j = -\omega^{n/2+j}$. Wir finden also $\pm$-Paare!
- ω^2 ist selbst eine primitive Einheitswurzel für $n/2$.
- Die Potenzen $(\omega^0)^2, \ldots, (\omega^{n/2-1})^2$ sind $(\omega^2)^0, \ldots, (\omega^2)^{n/2-1}$.
  Im Rekursionsschritt werden wir also wieder $\pm$-Paare finden.

Beispiel: $n = 16$:

16 Stützstellen.



Rekursion: 8 Stützstellen.



Als $(x_0, x_1, \ldots, x_{n-1})$ wählen wir $(\omega^0, \omega^1, \ldots, \omega^{n-1})$.

(Beachte: $\omega^0 = 1$, $\omega^1 = \omega$.)

Gegeben $A(x) = \sum_{0 \leq i \leq n-1} a_i x^i$ als Koeffizientenvektor $(a_0, a_1, \ldots, a_{n-1})$, wollen wir dann die „**diskrete Fourier-Transformierte**“

$(r_0, r_1, \ldots, r_{n-1}) = (A(1), A(\omega), A(\omega^2), \ldots, A(\omega^{n-1}))$

von $(a_0, \ldots, a_{n-1})$ berechnen.

Die Operation

$$\boxed{(a_0, \ldots, a_{n-1}) \mapsto (A(1), A(\omega), A(\omega^2), \ldots, A(\omega^{n-1}))}$$

heißt die **diskrete Fourier-Transformation**.

Es handelt sich dabei um eine bijektive lineare Abbildung von $\mathbb{C}^n$ nach $\mathbb{C}^n$.

(Es gibt auch eine Fourier-Transformation für Funktionen, die auf Integralen beruht.)

Wir berechnen zunächst **rekursiv** die n Werte

$$(s_0, \ldots, s_{n/2-1}) = (A_{\mathsf{g}}((\omega^2)^0), A_{\mathsf{g}}((\omega^2)^1), \ldots, A_{\mathsf{g}}((\omega^2)^{n/2-1}))$$

und

$$(t_0, \ldots, t_{n/2-1}) = (A_{\mathsf{u}}((\omega^2)^0), A_{\mathsf{u}}((\omega^2)^1), \ldots, A_{\mathsf{u}}((\omega^2)^{n/2-1})).$$

Wie schon festgestellt, ist ω^2 eine primitive $(n/2)$-te Einheitswurzel, so dass wir tatsächlich rekursiv vorgehen können.

Wie sollen wir jetzt $(r_0, r_1, \ldots, r_{n-1})$ berechnen?

Für $j = 0, \ldots, n/2 - 1$, mit (1):

$$r_j = A(\omega^j) = A_{\mathsf{g}}((\omega^j)^2) + \omega^j \cdot A_{\mathsf{u}}((\omega^j)^2) = s_j + \omega^j \cdot t_j.$$

(Jetzt kommt der Clou, die Anwendung des Plus-Minus-Tricks!)

Für $j = 0, \ldots, n/2 - 1$ gilt $\omega^{n/2+j} = \underbrace{\omega^{n/2}}_{=-1} \cdot \omega^j = -\omega^j$.

Also:

$$(\omega^{n/2+j})^2 = (-\omega^j)^2 = (\omega^j)^2.$$

Daher:

$$\begin{aligned} r_{n/2+j} &= A(\omega^{n/2+j}) = A_{\mathsf{g}}((\omega^{n/2+j})^2) + \omega^{n/2+j} \cdot A_{\mathsf{u}}((\omega^{n/2+j})^2) \\ &= A_{\mathsf{g}}((\omega^2)^j) - \omega^j \cdot A_{\mathsf{u}}((\omega^2)^j) \\ &= s_j - \omega^j \cdot t_j. \end{aligned}$$

Wir können die Ergebnisse der rekursiven Aufrufe also ein zweites Mal benutzen, um $(r_{n/2}, \ldots, r_{n-1})$ zu berechnen!

Algorithmus FFT (Schnelle Fourier-Transformation)

Eingabe: (Koeffizienten-)Vektor $(a_0, \ldots, a_{n-1})$, für Zweierpotenz n;
primitive n-te Einheitswurzel ω.

if $n = 1$ **then return** (a_0);

$(s_0, \ldots, s_{n/2-1}) \leftarrow \mathsf{FFT}((a_0, a_2, \ldots, a_{n-2}), \omega^2)$;

$(t_0, \ldots, t_{n/2-1}) \leftarrow \mathsf{FFT}((a_1, a_3, \ldots, a_{n-1}), \omega^2)$;

for j **from** 0 **to** $n/2 - 1$ **do**

$r_j \leftarrow s_j + \omega^j \cdot t_j$;

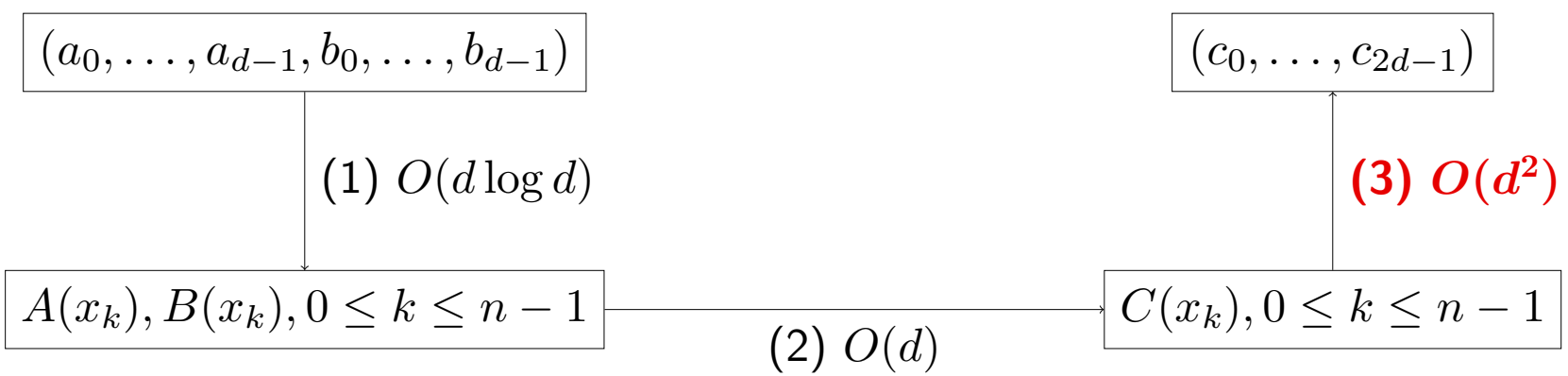
$r_{n/2+j} \leftarrow s_j - \omega^j \cdot t_j$;

return $(r_0, r_1, \ldots, r_{n-1})$.

Satz 9.6.2

Algorithmus FFT berechnet die diskrete Fouriertransformierte eines Koeffizientenvektors im Bezug auf die Argumente $(1, \omega, \omega^2, \ldots, \omega^{n-1})$, wobei ω eine primitive n-te Einheitswurzel ist, in Zeit $O(n \log n)$.

Aktueller Stand bei der Polynommultiplikation $(4d > n \geq 2d)$:



Es fehlt noch: **(3) Interpolation**.

Es fehlt noch: **Interpolation**. Auch hier ein schöner **Trick**.

Gegeben ist $(r_0, \ldots, r_{n-1})$, gesucht der Koeffizientenvektor $(a_0, \ldots, a_{n-1})$, der $M(\omega) \cdot (a_0, \ldots, a_{n-1})^{\mathsf{T}} = (r_0, \ldots, r_{n-1})^{\mathsf{T}}$ erfüllt, für die Matrix $M(\omega)$ der diskreten Fourier-Transformation (die Vandermonde-Matrix von $(\omega^0, \omega, \ldots, \omega^{n-1})$):

$$
\begin{aligned}
M(\omega) &= ((\omega^i)^j)_{0 \le i,j \le n-1} \\
&= \begin{pmatrix}
1 & 1 & 1 & \cdots & 1 \\
1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\
1 & \omega^2 & (\omega^2)^2 & \cdots & (\omega^2)^{n-1} \\
\vdots & \vdots & \vdots & & \vdots \\
1 & \omega^{n-1} & (\omega^{n-1})^2 & \cdots & (\omega^{n-1})^{n-1}
\end{pmatrix} .
\end{aligned}
$$

Man kann die zu $M(\omega)$ inverse Matrix direkt bestimmen.

Setze $\hat{\omega} := \omega^{n-1}$.

Dann gilt $\omega \cdot \hat{\omega} = \omega^n = 1$, also ist $\hat{\omega} = \omega^{-1}$.

Betrachte

$$
\begin{aligned}
M(\hat{\omega}) &= ((\hat{\omega}^i)^j)_{0 \le i,j \le n-1} \\
&= \begin{pmatrix}
1 & 1 & 1 & \cdots & 1 \\
1 & \hat{\omega} & \hat{\omega}^2 & \cdots & \hat{\omega}^{n-1} \\
1 & \hat{\omega}^2 & (\hat{\omega}^2)^2 & \cdots & (\hat{\omega}^2)^{n-1} \\
\vdots & \vdots & \vdots & & \vdots \\
1 & \hat{\omega}^{n-1} & (\hat{\omega}^{n-1})^2 & \cdots & (\hat{\omega}^{n-1})^{n-1}
\end{pmatrix}
\end{aligned}
$$

Wir berechnen den Eintrag z_{ij} an Stelle (i,j) der Produktmatrix $M(\omega) \cdot M(\hat{\omega})$:

$$z_{ij} = \sum_{0 \le k \le n-1} (\omega^i)^k \cdot (\hat{\omega}^k)^j = \sum_{0 \le k \le n-1} (\omega^{i+(n-1)j})^k.$$

Es gibt zwei Fälle. Wenn $i = j$ gilt, ist $\omega^{i+(n-1)j} = \omega^n = 1$, und die Summe ist n.

Wenn $i > j$ gilt, ist $\omega^{i+(n-1)j} = \omega^\ell$ für $1 \le \ell = i - j \le n-1$.
Wenn $i < j$ gilt, ist $\omega^{i+(n-1)j} = \omega^\ell$ für $1 \le \ell = n - (j-i) \le n-1$.

Wegen Bedingung (ii):

$$z_{ij} = \sum_{0 \le k \le n-1} (\omega^\ell)^k = 0.$$

Also: $M(\omega) \cdot M(\hat{\omega}) = n \cdot I_n$, für die $n \times n$-Einheitsmatrix I_n.

Das heißt: $M(\omega)^{-1} = n^{-1} \cdot M(\hat{\omega})$.

Wenn also $(r_0, \ldots, r_{n-1})$ der für die Interpolation gegebene Wertevektor ist, so erhalten wir den Koeffizientenvektor als

$$\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = n^{-1} \cdot M(\hat{\omega}) \cdot \begin{pmatrix} r_0 \\ r_1 \\ \vdots \\ r_{n-1} \end{pmatrix}.$$

Für die Zahl $\hat{\omega}$ beobachten wir:

(i) $\hat{\omega}^n = (\omega^{n-1})^n = (\omega^n)^{n-1} = 1^{n-1} = 1$,

(ii) für $1 \leq k \leq n-1$ gilt $\hat{\omega}^k = (\omega^{n-1})^k = \omega^{n(k-1)}\omega^{n-k} = \omega^{n-k}$,
mit $1 \leq n-k \leq n-1$, also $\sum_{0 \leq j \leq n-1} (\hat{\omega}^k)^j = 0$ wegen (ii) für ω.

Das heißt: Auch $\hat{\omega}$ ist eine primitive n-te Einheitswurzel.

Multiplikation mit $M(\hat{\omega})$ entspricht der FFT-Operation mit $\hat{\omega}$ an Stelle von ω.

Wir erhalten für die Interpolation:

$$(a_0, \ldots, a_{n-1}) = n^{-1} \cdot \mathsf{FFT}((r_0, \ldots, r_{n-1}), \hat{\omega}).$$

Zeitaufwand: $O(n \log n)$, also $O(d \log d)$. Damit: Plan von Folie 68 vollständig!

Algorithmus FFT-PM (Polynommultiplikation)

Eingabe: (Koeffizienten-)Vektoren $(a_0, \ldots, a_{d-1})$, $(b_0, \ldots, b_{d-1})$,
Setze $n = 2^{\lceil \log(2d) \rceil}$ $(\geq 2\,d)$.

Berechne n-te Einheitswurzeln $\omega \leftarrow e^{2\pi i/n}$ und $\hat{\omega} \leftarrow \omega^{n-1}$.

$(r_0^A, \ldots, r_{n-1}^A) \leftarrow \mathsf{FFT}((a_0, \ldots, a_{d-1}, 0, \ldots, 0), \omega)$; // auf Länge n auffüllen

$(r_0^B, \ldots, r_{n-1}^B) \leftarrow \mathsf{FFT}((b_0, \ldots, b_{d-1}, 0, \ldots, 0), \omega)$; // auf Länge n auffüllen

for j **from** 0 **to** $n-1$ **do** $r_j^C \leftarrow r_j^A \cdot r_j^B$;

$(c_0, c_1, \ldots, c_{n-1}) \leftarrow \frac{1}{n} \cdot \mathsf{FFT}((r_0^C, \ldots, r_{n-1}^C), \hat{\omega})$;

return $(c_0, c_1, \ldots, c_{2d-2})$.

Satz 9.6.3

Algorithmus FFT-PM berechnet die Koeffizienten des Produktes zweier durch Koeffizientenvektoren gegebener Polynome vom Grad $< d$ in Zeit $O(d \log d)$.

Beweis: Siehe vorherige Überlegungen.
(Rechenaufwand: $O(2d \log(2d)) = O(d \log d)$.)

Anmerkung: Um Polynome vom Grad $k-1$ und vom Grad $\ell-1$ zu multiplizieren, genügt $n=2^L$ für $L=\lceil \log_2(k+\ell-1)\rceil$.

Für Zuhause: Es folgen weitere Anmerkungen zur Geschichte und Anwendung der FFT. **(Nicht prüfungsrelevant.)**

Zum Üben: Berechnen Sie $(x+1)\cdot(x^2+1)$ mittels FFT!

Der FFT-Ansatz ermöglicht es auch, die diskrete Fourier-Transformatierte eines Vektors sehr effizient parallel zu berechnen, entweder mit mehreren Prozessoren oder sogar in Hardware.
In Algorithmus FFT können die beiden rekursiven Aufrufe unabhängig voneinander parallel durchgeführt werden (in Hardware: zwei Kopien der gleichen Schaltung nötig). Die Berechnung des Resultates $(r_0,\ldots,r_{n-1})$ kann sogar für alle n Werte gleichzeitig erfolgen.
Eine Beispielschaltung findet man auf Seite 69 im Buch von S. Dasgupta, C. Papadimitriou, U. Vazirani, **Algorithms**, McGraw-Hill, 2007.

Bemerkung für $\mathbb{C}$-Vektorraum-Spezialist/inn/en:

Die DFT ist eine Koordinatentransformation für einen Basiswechsel im $\mathbb{C}$-Vektorraum P_n der Polynome vom Grad bis zu $n-1$.

Man betrachtet ein inneres Produkt in P_n:

$$\left\langle \sum_i a_i X^i, \sum_i b_i X^i \right\rangle = \sum_{0 \le i \le n-1} a_i \overline{b_i}.$$

Definiere $f_j(X)$ als das (eindeutig bestimmte) Polynom, das $f_j(\omega^k) = [j = k]$ erfüllt, für $0 \le k \le n-1$. Dann ist die „Standardbasis“ $B_n = (1, X, X^2, \ldots, X_{n-1})$ Orthonormalbasis von P_n und die „Fourierbasis“ $F_n = (f_0(X), f_1(X), \ldots, f_{n-1}(X))$ Orthogonalbasis von P_n.

Die Matrix $M(\omega)$ ist die Matrix für den Basiswechsel von B_n (Koeffizienten $(a_0, \ldots, a_{n-1})$ nach F_n (Koeffizienten $(A(\omega^0), \ldots, A(\omega^{n-1}))$), die Matrix $M(\omega)^{-1} = \frac{1}{n} M(\hat{\omega})$ ist für den umgekehrten Basiswechsel zuständig.

Wozu eigentlich Polynommultiplikation oder „Faltung“

$$c_k = \sum_{i,j\,:\,i+j=k} a_i b_j, \quad \text{für } 0 \leq k \leq 2n-1 \text{ ?}$$

Zentrales Hilfsmittel bei der „Digitalen Signalverarbeitung“.

Offizielle Publikation: [Cooley/Tukey 1965].

Form des Algorithmus 1805 von C. F. Gauß entworfen (und zur Berechnung von Asteroidenbahnen benutzt).

Erstmalig publiziert wurde eine Variante des Algorithmus von C. Runge (1903/05).

(Quelle hierfür: Wikipedia – kann stimmen, muss aber nicht.)

Wir stellen uns ein System vor, in dem regelmäßig Signale anfallen und (zu Ausgabesignalen) verarbeitet werden.
Abtastzeitpunkte $t_0, t_1, \ldots$ mit festem Abstand $\Delta = t_{i+1} - t_i$ liefern (reelle, komplexe) Signalwerte $a_0, a_1, a_2, \ldots$.
Ein Verarbeitungsmechanismus soll diese Messwertfolge in eine Ausgabefolge $c_0, c_1, c_2, \ldots$ umsetzen, mit Ausgabezeiten $t'_0, t'_1, t'_2, \ldots$, ebenso mit Abstand Δ.

Die einfachsten Mechanismen sind **linear** (wenn man zwei Signalfolgen addiert, addieren sich die Ausgabefolgen, ebenso bei Multiplikation mit konstanten Faktoren)
und **zeitinvariant** (wenn die gleiche Signalfolge um einen Zeitschritt Δ versetzt auftritt, ergibt sich die gleiche um Δ versetzte Ausgabefolge).
Leicht zu sehen: Bei linearen, zeitinvarianten Signalverarbeitungs-Systemen ist durch die Ausgabefolge $(b_0, b_1, b_2, \ldots)$, die von einem Signal der Größe 1 bei t_0 und sonst nur Nullsignalen ausgelöst wird, die Ausgabefolge $(c_0, c_1, c_2, \ldots)$ auf einer beliebigen Signalfolge $(a_0, a_1, \ldots)$ eindeutig bestimmt, durch:

$$c_k = \sum_{i+j=k} a_i b_j.$$

Das ist gerade die Folge der Koeffizienten des Produktpolynoms! D.h.: Lineare, zeitinvariante Reaktion auf Messsignale führt unmittelbar zum Problem „Polynommultiplikation“.

„Multiplikation ganzer Zahlen“:
Wir wollen zwei Zahlen x, y mit je N Bits multiplizieren, N ist extrem groß.
Wir zerlegen die Binärdarstellung von x und y in Blöcke der Länge $\sqrt{N}$. Indem wir jeweils jeden dritten Block nehmen, bilden wir aus x und y jeweils drei Zahlen, so dass

$$x = x_0 + x_1 + x_2 \text{ und } y = y_0 + y_1 + y_2$$

gilt und so dass $x_i \cdot y_j$ als Multiplikation von Zahlen mit $\sqrt{N}$ Ziffern im Bereich $[\sqrt{N}]$ aufgefasst werden kann. Durch die großen eingestreuten Blöcke von Nullen gibt es keine Überträge.
(Bild: Tafel/Handzeichnung.)
Die „Ziffern“ der Produkte $x_i \cdot y_j$ stellen sich als die Koeffizienten eines Produktpolynoms heraus.
Mit der FFT könnte man diese Ziffern der Teilprodukte in $O(\sqrt{N} \log N)$ Multiplikationen von komplexen Zahlen ermitteln.

Unter der Annahme, dass eine solche Multiplikation durch geeignete Rundung auf komplexe Zahlen mit Darstellungslänge $O(\sqrt{N})$ in Zeit $O(N)$ möglich ist, erhalten wir einen Gesamtzeitaufwand von $O(N^{3/2} \log N)$, sogar besser als der Zeitbedarf des Karatsuba-Multiplizierers.
Durch Einziehen von Rekursionsstufen lässt sich die Zeit weiter drücken; wenn man es konsequent durchführt, auf $O(N^{1+\varepsilon})$ für beliebige konstante $\varepsilon > 0$ (Verfahren von Toom/Cook, z. B. im Buch „The Art of Computer Programming, Vol. 2: Seminumerical Algorithms“ von D. Knuth beschrieben).
Für den Multiplikationsalgorithmus von Schönhage und Strassen mit Kosten $O(N \log N \log\log N)$ sind neben der FFT noch weitere Ideen nötig.

Für die Arbeit mit ganzen Zahlen ist die oben beschriebene FFT sehr unbequem, weil man mit komplexen Zahlen rechnen muss.
In gewissen Situationen kann man FFT auch „modulo m“ durchführen, für geeignete „Moduli“ m, und ganz im diskreten Bereich bleiben.

Fakt 9.6.4
Wenn $w = 2$ und n eine Zweierpotenz ist, dann gilt für $m = 2^{n/2} + 1$:

(i) $w^n \bmod m = 1$ und $w^{n/2} \bmod m = -1$.

(ii) Für $1 \leq k \leq n-1$ gilt $\sum_{0 \leq j \leq n-1} (w^k)^j = 0$.

(iii) Für $1 \leq k \leq n-1$ gilt $w^k \neq 1$.

(iv) Es gibt $\hat{n}, \hat{w}$ mit $(n \cdot \hat{n}) \bmod m = 1$ und $(w \cdot \hat{w}) \bmod m = 1$.

Die Zahl w spielt also im Ring $\mathbb{Z}_m$ die Rolle einer primitiven n-ten Einheitswurzel, sie besitzt eine Inverse $\hat{w}$, und auch n hat eine Inverse $\hat{n}$.

Beispiel: $w = 2, n = 32, m = 2^{16} + 1 = 65537, \hat{w} = 32769, \hat{n} = 63489$.
Man kann sich überlegen, dass in dieser Situation alle Überlegungen, die wir oben für $\omega, \hat{\omega}, n, n^{-1}$ angestellt haben, für $w, \hat{w}, n, \hat{n}$ ebenso funktionieren, wir also den FFT-Algorithmus auch mit Addition und Multiplikation „modulo m“ benutzen können, obwohl $\mathbb{Z}_m$ normalerweise kein Körper ist. Damit lassen sich die Koeffizienten von Produkten ganzzahliger Polynome mit FFT ohne Umweg über die komplexen Zahlen berechnen, und auch in der Situation der Multiplikation von ganzen Zahlen ist diese FFT-Version günstiger. Einzige Voraussetzung: m muss so groß sein, dass die Koeffizienten c_k des Produktpolynoms schon durch den Wert $c_k \bmod m$ eindeutig bestimmt sind.

9.7 Die schnelle Fourier-Transformation

9.7 Die schnelle Fourier-Transformation

Aufgabe: Gegeben zwei Polynome $A(x)$ und $B(x)$, als

$$A(x) = \sum_{0 \le i \le n-1} a_i x^i \quad \text{und} \quad B(x) = \sum_{0 \le i \le n-1} b_i x^i,$$

9.7 Die schnelle Fourier-Transformation

Aufgabe: Gegeben zwei Polynome $A(x)$ und $B(x)$, als

$$A(x) = \sum_{0 \le i \le n-1} a_i x^i \quad \text{und} \quad B(x) = \sum_{0 \le i \le n-1} b_i x^i,$$

berechne das **Polynomprodukt**

$$A(x) \cdot B(x) = C(x) = \sum_{0 \le k \le 2n-2} c_k x^k,$$

9.7 Die schnelle Fourier-Transformation

Aufgabe: Gegeben zwei Polynome $A(x)$ und $B(x)$, als

$$A(x) = \sum_{0 \le i \le n-1} a_i x^i \quad \text{und} \quad B(x) = \sum_{0 \le i \le n-1} b_i x^i,$$

berechne das **Polynomprodukt**

$$A(x) \cdot B(x) = C(x) = \sum_{0 \le k \le 2n-2} c_k x^k,$$

d. h. berechne die Koeffizienten $c_k = \sum_{i,j \,:\, i+j=k} a_i b_j$.

Die Folge $(c_0, \ldots, c_{2n-2})$ heißt auch **Konvolution** $(a_0, \ldots, a_{n-1}) \circ (b_0, \ldots, b_{n-1})$.

9.7 Die schnelle Fourier-Transformation

Aufgabe: Gegeben zwei Polynome $A(x)$ und $B(x)$, als

$$A(x) = \sum_{0 \le i \le n-1} a_i x^i \quad \text{und} \quad B(x) = \sum_{0 \le i \le n-1} b_i x^i,$$

berechne das **Polynomprodukt**

$$A(x) \cdot B(x) = C(x) = \sum_{0 \le k \le 2n-2} c_k x^k,$$

d. h. berechne die Koeffizienten $c_k = \sum_{i,j \,:\, i+j=k} a_i b_j$.

Die Folge $(c_0, \ldots, c_{2n-2})$ heißt auch **Konvolution** $(a_0, \ldots, a_{n-1}) \circ (b_0, \ldots, b_{n-1})$.

Naive Benutzung der Formel liefert $\Theta(n^2)$-Algorithmus.

9.7 Die schnelle Fourier-Transformation

Aufgabe: Gegeben zwei Polynome $A(x)$ und $B(x)$, als

$$A(x) = \sum_{0 \le i \le n-1} a_i x^i \quad \text{und} \quad B(x) = \sum_{0 \le i \le n-1} b_i x^i,$$

berechne das **Polynomprodukt**

$$A(x) \cdot B(x) = C(x) = \sum_{0 \le k \le 2n-2} c_k x^k,$$

d. h. berechne die Koeffizienten $c_k = \sum_{i,j \,:\, i+j=k} a_i b_j$.

Die Folge $(c_0, \ldots, c_{2n-2})$ heißt auch **Konvolution** $(a_0, \ldots, a_{n-1}) \circ (b_0, \ldots, b_{n-1})$.

Naive Benutzung der Formel liefert $\Theta(n^2)$-Algorithmus.

Ziel: $O(n \log n)$. – Methode: Divide-and-Conquer.

Zentraler Trick: Benutze eine weitere Darstellung von Polynomen, die

Stützstellen-Darstellung.

Zentraler Trick: Benutze eine weitere Darstellung von Polynomen, die

Stützstellen-Darstellung.

Betrachte eine Folge $(x_0, \ldots, x_{n-1}) \in \mathbb{C}^n$ von **verschiedenen** Stützstellen.

Zentraler Trick: Benutze eine weitere Darstellung von Polynomen, die

Stützstellen-Darstellung.

Betrachte eine Folge $(x_0, \ldots, x_{n-1}) \in \mathbb{C}^n$ von **verschiedenen** Stützstellen.

Fakt 9.7.1 („Interpolation“ von Polynomen über Körpern)

Zu jedem beliebigen **Werte**vektor $(r_0, \ldots, r_{n-1}) \in \mathbb{C}^n$ gibt es genau ein Polynom $A(x) = \sum_{0 \leq i \leq n-1} a_i x^i$, also genau einen **Koeffizienten**vektor $(a_0, \ldots, a_{n-1})$, mit $A(x_k) = r_k$ für $0 \leq k \leq n-1$.

Zentraler Trick: Benutze eine weitere Darstellung von Polynomen, die

Stützstellen-Darstellung.

Betrachte eine Folge $(x_0, \ldots, x_{n-1}) \in \mathbb{C}^n$ von **verschiedenen** Stützstellen.

Fakt 9.7.1 („Interpolation“ von Polynomen über Körpern)

Zu jedem beliebigen **Werte**vektor $(r_0, \ldots, r_{n-1}) \in \mathbb{C}^n$ gibt es genau ein Polynom $A(x) = \sum_{0 \leq i \leq n-1} a_i x^i$, also genau einen **Koeffizienten**vektor $(a_0, \ldots, a_{n-1})$, mit $A(x_k) = r_k$ für $0 \leq k \leq n-1$.

Es ist also im Prinzip egal, ob man für die Darstellung eines Polynoms seine Koeffizienten oder einen Wertevektor benutzt.

Beweis von Fakt 9.7.1: Betrachte die „**Vandermonde-Matrix**“

$$V(x_0, \ldots, x_{n-1}) = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix}.$$

Beweis von Fakt 9.7.1: Betrachte die „**Vandermonde-Matrix**“

$$V(x_0, \ldots, x_{n-1}) = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix}.$$

Offensichtlich ist

$$\begin{pmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{pmatrix} = V(x_0, \ldots, x_{n-1}) \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}.$$

Die Matrix $V(x_0, \ldots, x_{n-1})$ hat Determinante $\prod_{0 \le k < \ell \le n-1} (x_\ell - x_k) \neq 0$

(siehe Mathematik), ist also **regulär**.

Die Matrix $V(x_0, \ldots, x_{n-1})$ hat Determinante $\prod_{0 \le k < \ell \le n-1} (x_\ell - x_k) \neq 0$

(siehe Mathematik), ist also **regulär**. Daher hat das Gleichungssystem

$$V(x_0, \ldots, x_{n-1}) \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} r_0 \\ r_1 \\ \vdots \\ r_{n-1} \end{pmatrix}$$

Die Matrix $V(x_0, \ldots, x_{n-1})$ hat Determinante $\prod_{0 \le k < \ell \le n-1} (x_\ell - x_k) \neq 0$

(siehe Mathematik), ist also **regulär**. Daher hat das Gleichungssystem

$$V(x_0, \ldots, x_{n-1}) \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} r_0 \\ r_1 \\ \vdots \\ r_{n-1} \end{pmatrix}$$

genau eine Lösung

$$\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = V(x_0, \ldots, x_{n-1})^{-1} \cdot \begin{pmatrix} r_0 \\ r_1 \\ \vdots \\ r_{n-1} \end{pmatrix}.$$

Aus der Darstellung der n-fachen Polynomauswertung als Matrix-Vektor-Produkt folgt auch:

Bemerkung: Die Umrechnung von $(a_0, \ldots, a_{n-1})$ in $(r_0, \ldots, r_{n-1})$ ist eine (bijektive) **lineare Abbildung** von $\mathbb{C}^n$ nach $\mathbb{C}^n$.

Aus der Darstellung der n-fachen Polynomauswertung als Matrix-Vektor-Produkt folgt auch:

Bemerkung: Die Umrechnung von $(a_0, \ldots, a_{n-1})$ in $(r_0, \ldots, r_{n-1})$ ist eine (bijektive) **lineare Abbildung** von $\mathbb{C}^n$ nach $\mathbb{C}^n$.

Um durch Interpolation die $2n-1$ Koeffizienten des Produktpolynoms $C(x)$ zu erhalten, müssen wir mit mindestens $2n-1$ Stützstellen-Werte-Paaren arbeiten.
Aus technischen Gründen verwenden wir $2n$ viele.

Algorithmenplan für die Polynommultiplikation:

Algorithmenplan für die Polynommultiplikation:

Eingabe: Zwei Polynome $A(x)$ und $B(x)$ als Koeffizientenvektoren $(a_0, \ldots, a_{n-1})$ und $(b_0, \ldots, b_{n-1})$.

Algorithmenplan für die Polynommultiplikation:

Eingabe: Zwei Polynome $A(x)$ und $B(x)$ als Koeffizientenvektoren $(a_0, \ldots, a_{n-1})$ und $(b_0, \ldots, b_{n-1})$.

$x_0, x_1, \ldots, x_{2n-1} \in \mathbb{C}$ seien verschiedene Stützstellen.

Algorithmenplan für die Polynommultiplikation:

Eingabe: Zwei Polynome $A(x)$ und $B(x)$ als Koeffizientenvektoren $(a_0, \ldots, a_{n-1})$ und $(b_0, \ldots, b_{n-1})$.

$x_0, x_1, \ldots, x_{2n-1} \in \mathbb{C}$ seien verschiedene Stützstellen.

(1) Polynom**auswertung**:
Berechne Werte $A(x_k)$ und $B(x_k)$, für $k = 0, \ldots, 2n-1$.

Algorithmenplan für die Polynommultiplikation:

Eingabe: Zwei Polynome $A(x)$ und $B(x)$ als Koeffizientenvektoren $(a_0, \ldots, a_{n-1})$ und $(b_0, \ldots, b_{n-1})$.

$x_0, x_1, \ldots, x_{2n-1} \in \mathbb{C}$ seien verschiedene Stützstellen.

(1) Polynom**auswertung**:
Berechne Werte $A(x_k)$ und $B(x_k)$, für $k = 0, \ldots, 2n-1$.

(2) Berechne durch **punktweise Multiplikation**
Werte des Produktpolynoms $C(x)$:
$C(x_k) := A(x_k) \cdot B(x_k)$, für $k = 0, \ldots, 2n-1$.

Algorithmenplan für die Polynommultiplikation:

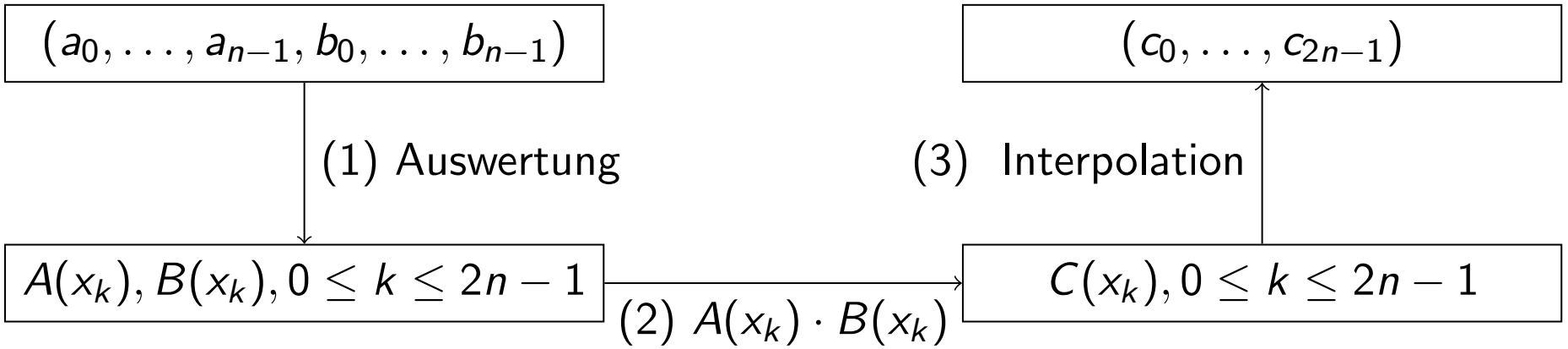
Eingabe: Zwei Polynome $A(x)$ und $B(x)$ als Koeffizientenvektoren $(a_0, \ldots, a_{n-1})$ und $(b_0, \ldots, b_{n-1})$.

$x_0, x_1, \ldots, x_{2n-1} \in \mathbb{C}$ seien verschiedene Stützstellen.

(1) Polynom**auswertung**:
Berechne Werte $A(x_k)$ und $B(x_k)$, für $k = 0, \ldots, 2n-1$.

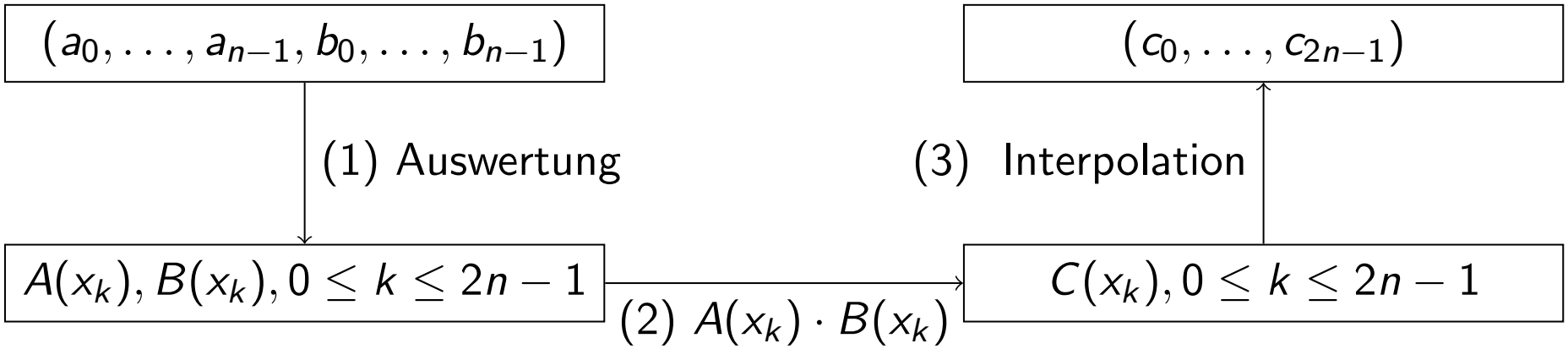
(2) Berechne durch **punktweise Multiplikation**
Werte des Produktpolynoms $C(x)$:
$C(x_k) := A(x_k) \cdot B(x_k)$, für $k = 0, \ldots, 2n-1$.

(3) **Interpolation**:
Berechne aus $(C(x_0), C(x_1), \ldots, C(x_{2n-1}))$ die Koeffizienten $(c_0, \ldots, c_{2n-1})$ von $C(x)$.


$(a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1})$
$(c_0, \ldots, c_{2n-1})$
(1) Auswertung
(3) Interpolation
$A(x_k), B(x_k), 0 \leq k \leq 2n-1$
$C(x_k), 0 \leq k \leq 2n-1$
(2) $A(x_k) \cdot B(x_k)$

Kosten:

(1) **??**

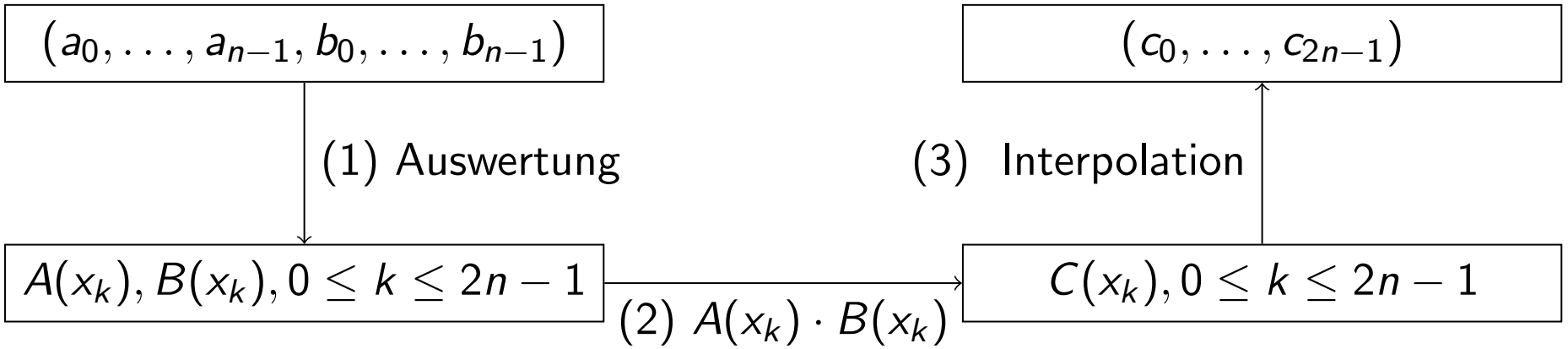


Kosten:

(1) **??** Naiv: Jeden Wert $A(x_k)$ separat berechnen, z. B. mit dem Horner-Schema:

$$A(x_k) = ((\ldots(a_{n-1} \cdot x_k + a_{n-2}) \cdot x_k \ldots) \cdot x_k + a_1) \cdot x_k + a_0$$

Kosten: $O(n^2)$.



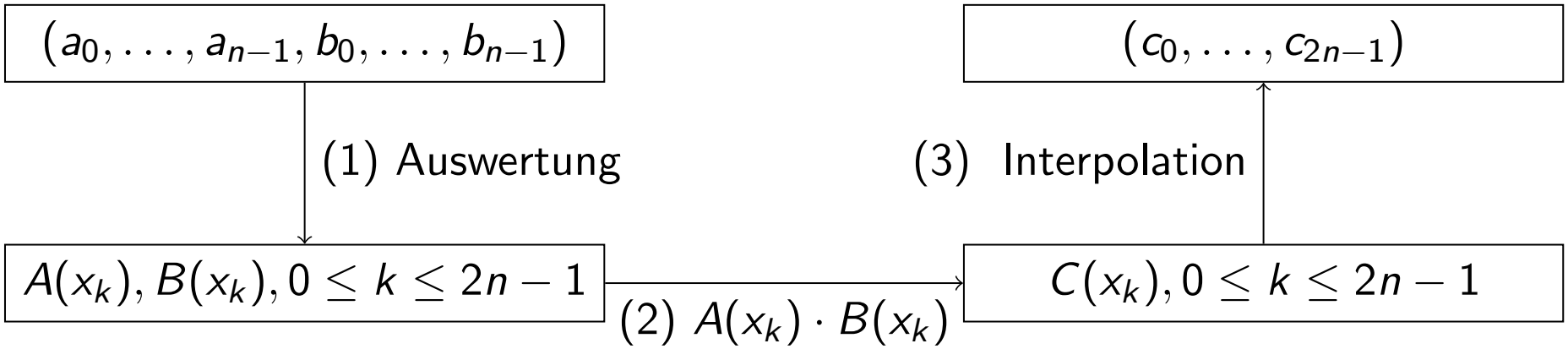
Kosten:

(1) **??** Naiv: Jeden Wert $A(x_k)$ separat berechnen, z. B. mit dem Horner-Schema:

$$A(x_k) = ((\ldots(a_{n-1} \cdot x_k + a_{n-2}) \cdot x_k \ldots) \cdot x_k + a_1) \cdot x_k + a_0$$

Kosten: $O(n^2)$.

(2) $O(n)$.



Kosten:

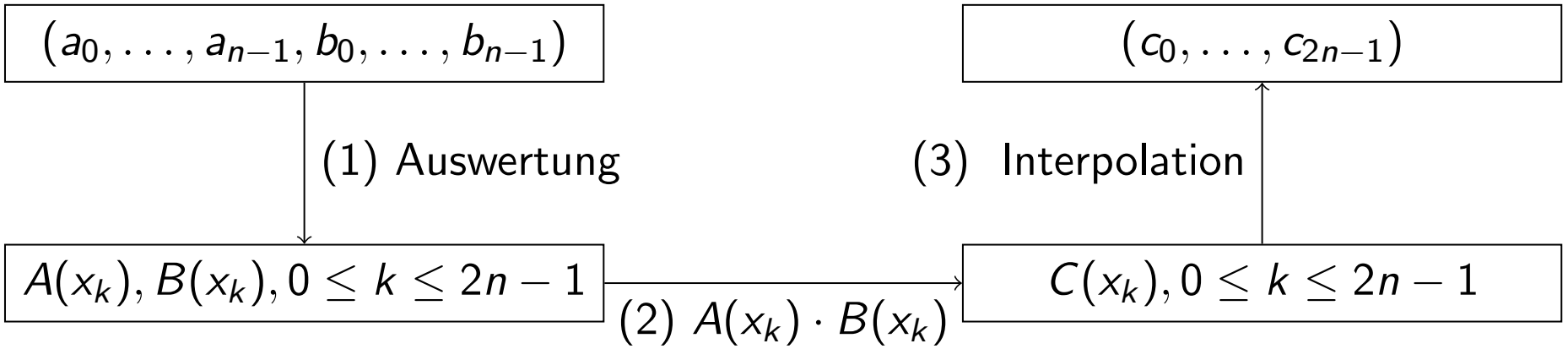
(1) **??** Naiv: Jeden Wert $A(x_k)$ separat berechnen, z. B. mit dem Horner-Schema:

$$A(x_k) = ((\ldots(a_{n-1} \cdot x_k + a_{n-2}) \cdot x_k \ldots) \cdot x_k + a_1) \cdot x_k + a_0$$

Kosten: $O(n^2)$.

(2) $O(n)$.

(3) **??**



Kosten:

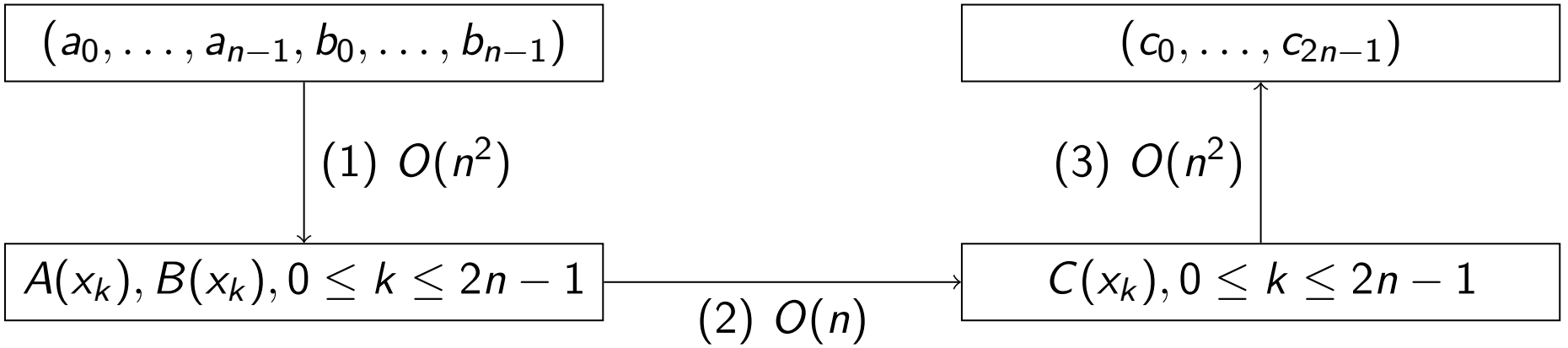
(1) **??** Naiv: Jeden Wert $A(x_k)$ separat berechnen, z. B. mit dem Horner-Schema:

$$A(x_k) = ((\ldots(a_{n-1} \cdot x_k + a_{n-2}) \cdot x_k \ldots) \cdot x_k + a_1) \cdot x_k + a_0$$

Kosten: $O(n^2)$.

(2) $O(n)$.

(3) **??** (Auch hier: $O(n^2)$ recht leicht zu erreichen.)



Kosten:

Unser Ziel: $O(n \log n)$ für (1) und (3).

Zunächst: **Auswertung, für $n = 2^L$ Stützstellen.**

Input: $A(x)$ als $(a_0, \ldots, a_{n-1})$, Stützstellen $(x_0, \ldots, x_{n-1})$.

Output: $(r_0, \ldots, r_{n-1}) = (A(x_0), \ldots, A(x_{n-1}))$.

Ansatz: **Divide-and-Conquer**.

Ansatz: **Divide-and-Conquer**.

Wenn $n = 1$, ist das Ergebnis $(r_0) = (a_0)$.

Ansatz: **Divide-and-Conquer**.

Wenn $n = 1$, ist das Ergebnis $(r_0) = (a_0)$.

Wenn $n > 1$, teilen wir $A(x)$ in zwei Teilpolynome auf:

$$\begin{aligned} A(x) &= (a_0 + a_2x^2 + a_4x^4 + \cdots + a_{n-2}x^{n-2}) \\ &\quad + x(a_1 + a_3x^2 + a_5x^4 + \cdots + a_{n-1}x^{n-2}), \end{aligned}$$

mit den Abkürzungen („**g**erade“, “**u**ngerade“)

Ansatz: **Divide-and-Conquer**.

Wenn $n = 1$, ist das Ergebnis $(r_0) = (a_0)$.

Wenn $n > 1$, teilen wir $A(x)$ in zwei Teilpolynome auf:

$$\begin{aligned} A(x) &= (a_0 + a_2x^2 + a_4x^4 + \cdots + a_{n-2}x^{n-2}) \\ &\quad + x(a_1 + a_3x^2 + a_5x^4 + \cdots + a_{n-1}x^{n-2}), \end{aligned}$$

mit den Abkürzungen („**g**erade“, “**u**ngerade“)

$A_{\mathrm{g}}(x) := a_0 + a_2x + \cdots + a_{n-2}x^{n/2-1}$ und
$A_{\mathrm{u}}(x) := a_1 + a_3x + \cdots + a_{n-1}x^{n/2-1}$

Ansatz: **Divide-and-Conquer**.

Wenn $n = 1$, ist das Ergebnis $(r_0) = (a_0)$.

Wenn $n > 1$, teilen wir $A(x)$ in zwei Teilpolynome auf:

$$\begin{aligned} A(x) &= (a_0 + a_2x^2 + a_4x^4 + \cdots + a_{n-2}x^{n-2}) \\ &\quad + x(a_1 + a_3x^2 + a_5x^4 + \cdots + a_{n-1}x^{n-2}), \end{aligned}$$

mit den Abkürzungen („**g**erade“, “**u**ngerade“)

$A_{\mathrm{g}}(x) := a_0 + a_2x + \cdots + a_{n-2}x^{n/2-1}$ und
$A_{\mathrm{u}}(x) := a_1 + a_3x + \cdots + a_{n-1}x^{n/2-1}$ also

$$A(x) = A_{\mathrm{g}}(x^2) + xA_{\mathrm{u}}(x^2). \tag{1}$$

$$A(x) = A_{\mathrm{g}}(x^2) + xA_{\mathrm{u}}(x^2).$$

$A_{\mathrm{g}}(x^2)$ und $A_{\mathrm{u}}(x^2)$ haben jeweils nur noch $n/2$ Koeffizienten.
(Inputgröße halbiert!)

$$A(x) = A_{\mathrm{g}}(x^2) + xA_{\mathrm{u}}(x^2).$$

$A_{\mathrm{g}}(x^2)$ und $A_{\mathrm{u}}(x^2)$ haben jeweils nur noch $n/2$ Koeffizienten. (Inputgröße halbiert!)

Originalproblem: Werte das Polynom $A(x)$ an den Stützstellen $x_0, \ldots, x_{n-1}$ aus.

$$A(x) = A_{\mathrm{g}}(x^2) + xA_{\mathrm{u}}(x^2).$$

$A_{\mathrm{g}}(x^2)$ und $A_{\mathrm{u}}(x^2)$ haben jeweils nur noch $n/2$ Koeffizienten. (Inputgröße halbiert!)

Originalproblem: Werte das Polynom $A(x)$ an den Stützstellen $x_0, \ldots, x_{n-1}$ aus.

Müssen nun rekursiv $A_{\mathrm{g}}(x_j^2)$ und $A_{\mathrm{u}}(x_j^2)$ berechnen, für $0 \leq j < n$.

$$A(x) = A_{\mathrm{g}}(x^2) + xA_{\mathrm{u}}(x^2).$$

$A_{\mathrm{g}}(x^2)$ und $A_{\mathrm{u}}(x^2)$ haben jeweils nur noch $n/2$ Koeffizienten.
(Inputgröße halbiert!)

Originalproblem: Werte das Polynom $A(x)$ an den Stützstellen $x_0, \ldots, x_{n-1}$ aus.

Müssen nun rekursiv $A_{\mathrm{g}}(x_j^2)$ und $A_{\mathrm{u}}(x_j^2)$ berechnen, für $0 \leq j < n$.

Das passt aber nicht mit der Rekursion zusammen.
(Ein Polynom mit $n/2$ Koeffizienten wird an $n/2$ Stellen ausgewertet.)

Zentraler Trick:

Zentraler Trick: Sorge dafür, dass

$$x_0^2, \ldots, x_{n-1}^2$$

nur $n/2$ verschiedene Stützstellen sind.

Zentraler Trick: Sorge dafür, dass

$$x_0^2, \ldots, x_{n-1}^2$$

nur $n/2$ verschiedene Stützstellen sind.

Wie kann das funktionieren?

Zentraler Trick: Sorge dafür, dass

$$x_0^2, \ldots, x_{n-1}^2$$

nur $n/2$ verschiedene Stützstellen sind.

Wie kann das funktionieren?

Wähle $x_0, \ldots, x_{n/2-1}$ verschieden und $\neq 0$ und betrachte als Stützstellen

$$\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1},$$

Zentraler Trick: Sorge dafür, dass

$$x_0^2, \ldots, x_{n-1}^2$$

nur $n/2$ verschiedene Stützstellen sind.

Wie kann das funktionieren?

Wähle $x_0, \ldots, x_{n/2-1}$ verschieden und $\neq 0$ und betrachte als Stützstellen

$$\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1},$$

mit folgende Anordnung im Stützstellenvektor:

$$(x_0, \ldots, x_{n/2-1}, -x_0, \ldots, -x_{n/2-1}).$$

Zentraler Trick: Sorge dafür, dass

$$x_0^2, \ldots, x_{n-1}^2$$

nur $n/2$ verschiedene Stützstellen sind.

Wie kann das funktionieren?

Wähle $x_0, \ldots, x_{n/2-1}$ verschieden und $\neq 0$ und betrachte als Stützstellen

$$\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1},$$

mit folgende Anordnung im Stützstellenvektor:

$$(x_0, \ldots, x_{n/2-1}, -x_0, \ldots, -x_{n/2-1}).$$

Vorerst nehmen wir an, dass die Stützstellen auf jeder Ebene der Rekursion diese Eigenschaft erfüllen.

Auswertung des Polynoms $A(x)$ auf Stützstellen $(x_0, \ldots, x_{n-1})$:

Auswertung des Polynoms $A(x)$ auf Stützstellen $(x_0, \ldots, x_{n-1})$:

Teile: Berechne Koeffizienten von $A_{\mathrm{g}}(x)$ und $A_{\mathrm{u}}(x)$, sowie $(x'_0, x'_1, \ldots, x'_{n/2-1}) = (x_0^2, x_1^2, \ldots, x_{n/2-1}^2)$.

Auswertung des Polynoms $A(x)$ auf Stützstellen $(x_0, \ldots, x_{n-1})$:

Teile: Berechne Koeffizienten von $A_{\mathrm{g}}(x)$ und $A_{\mathrm{u}}(x)$, sowie $(x'_0, x'_1, \ldots, x'_{n/2-1}) = (x_0^2, x_1^2, \ldots, x_{n/2-1}^2)$.

Rekursion: Werte $A_{\mathrm{g}}(x)$ mit $(x'_0, x'_1, \ldots, x'_{n/2-1})$ und $A_{\mathrm{u}}(x)$ mit $(x'_0, x'_1, \ldots, x'_{n/2-1})$ rekursiv aus.

Auswertung des Polynoms $A(x)$ auf Stützstellen $(x_0, \ldots, x_{n-1})$:

Teile: Berechne Koeffizienten von $A_{\mathrm{g}}(x)$ und $A_{\mathrm{u}}(x)$, sowie $(x'_0, x'_1, \ldots, x'_{n/2-1}) = (x_0^2, x_1^2, \ldots, x_{n/2-1}^2)$.

Rekursion: Werte $A_{\mathrm{g}}(x)$ mit $(x'_0, x'_1, \ldots, x'_{n/2-1})$ und $A_{\mathrm{u}}(x)$ mit $(x'_0, x'_1, \ldots, x'_{n/2-1})$ rekursiv aus. Resultat: Zwei Wertevektoren

$$A_{\mathrm{g}}(x_0^2), A_{\mathrm{g}}(x_1^2), \ldots, A_{\mathrm{g}}(x_{n/2-1}^2) \quad \text{und}$$
$$A_{\mathrm{u}}(x_0^2), A_{\mathrm{u}}(x_1^2), \ldots, A_{\mathrm{u}}(x_{n/2-1}^2).$$

Auswertung des Polynoms $A(x)$ auf Stützstellen $(x_0, \ldots, x_{n-1})$:

Teile: Berechne Koeffizienten von $A_{\mathrm{g}}(x)$ und $A_{\mathrm{u}}(x)$, sowie $(x'_0, x'_1, \ldots, x'_{n/2-1}) = (x_0^2, x_1^2, \ldots, x_{n/2-1}^2)$.

Rekursion: Werte $A_{\mathrm{g}}(x)$ mit $(x'_0, x'_1, \ldots, x'_{n/2-1})$ und $A_{\mathrm{u}}(x)$ mit $(x'_0, x'_1, \ldots, x'_{n/2-1})$ rekursiv aus. Resultat: Zwei Wertevektoren

$$A_{\mathrm{g}}(x_0^2), A_{\mathrm{g}}(x_1^2), \ldots, A_{\mathrm{g}}(x_{n/2-1}^2) \quad \text{und}$$
$$A_{\mathrm{u}}(x_0^2), A_{\mathrm{u}}(x_1^2), \ldots, A_{\mathrm{u}}(x_{n/2-1}^2).$$

Kombiniere: Berechne nun:

$$A(x_j) = A_{\mathrm{g}}(x_j^2) + x_j \cdot A_{\mathrm{u}}(x_j^2), \text{ für } 0 \leq j < n/2;$$
$$A(x_{n/2+j}) = A_{\mathrm{g}}(x_j^2) - x_j \cdot A_{\mathrm{u}}(x_j^2), \text{ für } 0 \leq j < n/2.$$

Auswertung des Polynoms $A(x)$ auf Stützstellen $(x_0, \ldots, x_{n-1})$:

Teile: Berechne Koeffizienten von $A_{\mathrm{g}}(x)$ und $A_{\mathrm{u}}(x)$, sowie $(x'_0, x'_1, \ldots, x'_{n/2-1}) = (x_0^2, x_1^2, \ldots, x_{n/2-1}^2)$.

Rekursion: Werte $A_{\mathrm{g}}(x)$ mit $(x'_0, x'_1, \ldots, x'_{n/2-1})$ und $A_{\mathrm{u}}(x)$ mit $(x'_0, x'_1, \ldots, x'_{n/2-1})$ rekursiv aus. Resultat: Zwei Wertevektoren

$$A_{\mathrm{g}}(x_0^2), A_{\mathrm{g}}(x_1^2), \ldots, A_{\mathrm{g}}(x_{n/2-1}^2) \quad \text{und}$$
$$A_{\mathrm{u}}(x_0^2), A_{\mathrm{u}}(x_1^2), \ldots, A_{\mathrm{u}}(x_{n/2-1}^2).$$

Kombiniere: Berechne nun:

$$A(x_j) = A_{\mathrm{g}}(x_j^2) + x_j \cdot A_{\mathrm{u}}(x_j^2), \text{ für } 0 \leq j < n/2;$$
$$A(x_{n/2+j}) = A_{\mathrm{g}}(x_j^2) - x_j \cdot A_{\mathrm{u}}(x_j^2), \text{ für } 0 \leq j < n/2.$$

Zentral: $n/2$ Multiplikationen und n Additionen im Kombinierschritt.

Unter der Annahme, dass auf jeder Rekursionsebene geeignete Stützstellen gefunden werden, schätzen wir die Laufzeit ab:

Unter der Annahme, dass auf jeder Rekursionsebene geeignete Stützstellen gefunden werden, schätzen wir die Laufzeit ab:

$C(n)$: Rechenzeit bei Eingaben der Länge n, vernachlässige konstante Faktoren.

Unter der Annahme, dass auf jeder Rekursionsebene geeignete Stützstellen gefunden werden, schätzen wir die Laufzeit ab:

$C(n)$: Rechenzeit bei Eingaben der Länge n, vernachlässige konstante Faktoren.

Rekurrenz:

$$C(n) \leq \begin{cases} 1 & \text{, falls } n = 1 \\ 2 \cdot C(n/2) + cn & \text{, sonst,} \end{cases}$$

für eine Konstante c.

Unter der Annahme, dass auf jeder Rekursionsebene geeignete Stützstellen gefunden werden, schätzen wir die Laufzeit ab:

$C(n)$: Rechenzeit bei Eingaben der Länge n, vernachlässige konstante Faktoren.

Rekurrenz:

$$C(n) \leq \begin{cases} 1 & \text{, falls } n = 1 \\ 2 \cdot C(n/2) + cn & \text{, sonst,} \end{cases}$$

für eine Konstante c. Mit dem Master-Theorem, 2. Fall, ergibt sich

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

Problem: Auf oberster Rekursionsstufe: $\pm x_0, \ldots, \pm x_{n/2-1}$ als „Plus-Minus“-Paare gewählt.

Problem: Auf oberster Rekursionsstufe: $\pm x_0, \ldots, \pm x_{n/2-1}$ als „Plus-Minus“-Paare gewählt.

Im rekursiven Aufruf: $x_0^2, \ldots, x_{n/2-1}^2$ müssen wieder in „Plus-Minus“-Paare aufgeteilt werden.

Problem: Auf oberster Rekursionsstufe: $\pm x_0, \ldots, \pm x_{n/2-1}$ als „Plus-Minus“-Paare gewählt.

Im rekursiven Aufruf: $x_0^2, \ldots, x_{n/2-1}^2$ müssen wieder in „Plus-Minus“-Paare aufgeteilt werden.

Wenn die Stützstellen reell sind, kann dies nicht funktionieren: Quadrate sind nie negativ.

Problem: Auf oberster Rekursionsstufe: $\pm x_0, \ldots, \pm x_{n/2-1}$ als „Plus-Minus“-Paare gewählt.

Im rekursiven Aufruf: $x_0^2, \ldots, x_{n/2-1}^2$ müssen wieder in „Plus-Minus“-Paare aufgeteilt werden.

Wenn die Stützstellen reell sind, kann dies nicht funktionieren: Quadrate sind nie negativ.

Trick: Nutze komplexe Zahlen!

ω sei **primitive *n*-te Einheitswurzel**, d. h.

ω sei **primitive *n*-te Einheitswurzel**, d. h.

(i) $\omega^n = 1$,

ω sei **primitive *n*-te Einheitswurzel**, d. h.

(i) $\omega^n = 1$,

(ii) für $1 \leq k \leq n-1$ gilt $\sum_{0 \leq j \leq n-1} (\omega^k)^j = 0$.

ω sei **primitive *n*-te Einheitswurzel**, d. h.

(i) $\omega^n = 1$,

(ii) für $1 \leq k \leq n-1$ gilt $\sum_{0 \leq j \leq n-1} (\omega^k)^j = 0$.

(iii) $\omega^k \neq 1$, für $1 \leq k \leq n-1$.

ω sei **primitive *n*-te Einheitswurzel**, d. h.

(i) $\omega^n = 1$,

(ii) für $1 \leq k \leq n-1$ gilt $\sum_{0 \leq j \leq n-1} (\omega^k)^j = 0$.

(iii) $\omega^k \neq 1$, für $1 \leq k \leq n-1$.

Unter einfachen weiteren Voraussetzungen sind (ii) und (iii) äquivalent:

– Wenn $\omega^k = 1$ ist, dann folgt $\sum_{0 \leq j \leq n-1} (\omega^k)^j = n$.

Daher folgt (iii), wenn (ii) und $n \neq 0$ gelten.

– Wenn $\omega^k \neq 1$ gilt, schreiben wir:

$(\omega^k - 1)(\sum_{0 \leq j \leq n-1} (\omega^k)^j) = (\omega^k)^n - 1 = 0$.

Daher folgt (ii), wenn (iii) gilt und zudem

die Inversen $(\omega^k - 1)^{-1}$ existieren.

In $\mathbb{C}$: Eine primitive n-te Einheitswurzel ist

$$\omega := e^{2\pi i/n} = \cos(2\pi/n) + i\sin(2\pi/n),$$

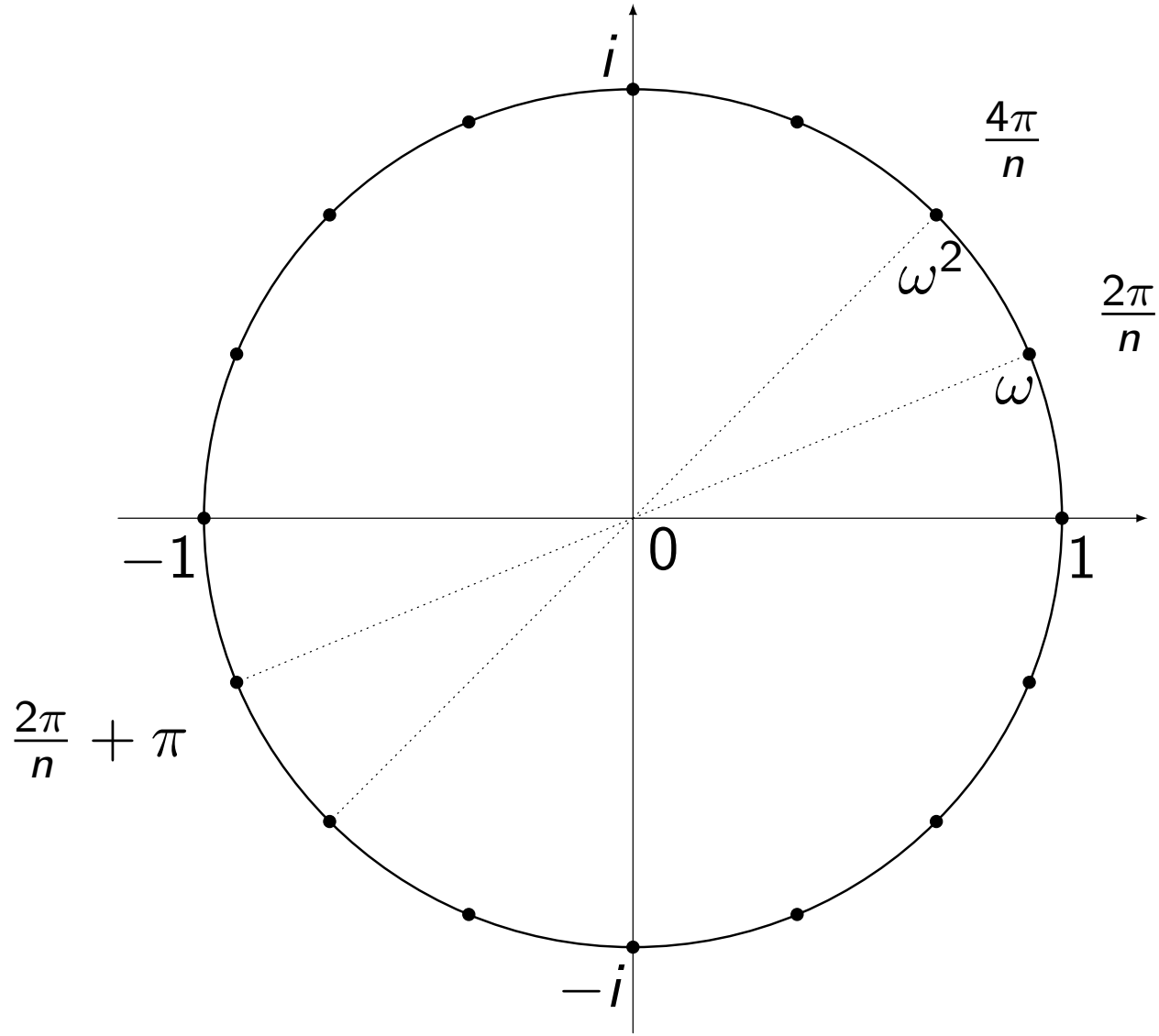
wobei hier i die imaginäre Einheit ist. (Beweis dieser Eigenschaft: später.)
In der komplexen Zahlenebene liegt der Punkt ω also auf dem Einheitskreis, von 1 aus um den Winkel $2\pi/n$ gegen den Uhrzeigersinn verdreht.

In $\mathbb{C}$: Eine primitive n-te Einheitswurzel ist

$$\omega := e^{2\pi i/n} = \cos(2\pi/n) + i\sin(2\pi/n),$$

wobei hier i die imaginäre Einheit ist. (Beweis dieser Eigenschaft: später.)
In der komplexen Zahlenebene liegt der Punkt ω also auf dem Einheitskreis, von 1 aus um den Winkel $2\pi/n$ gegen den Uhrzeigersinn verdreht.

Die Potenzen ω^k, $0 \leq k \leq n-1$, liegen in gleichen Abständen auf dem Einheitskreis.


i
$\frac{4\pi}{n}$
ω^2
$\frac{2\pi}{n}$
ω
-1
0
1
$\frac{2\pi}{n} + \pi$
$-i$

Für $n = 2^L$ gilt:

Für $n = 2^L$ gilt:

- $\omega^0 = 1, \omega^{n/2} = -1.$

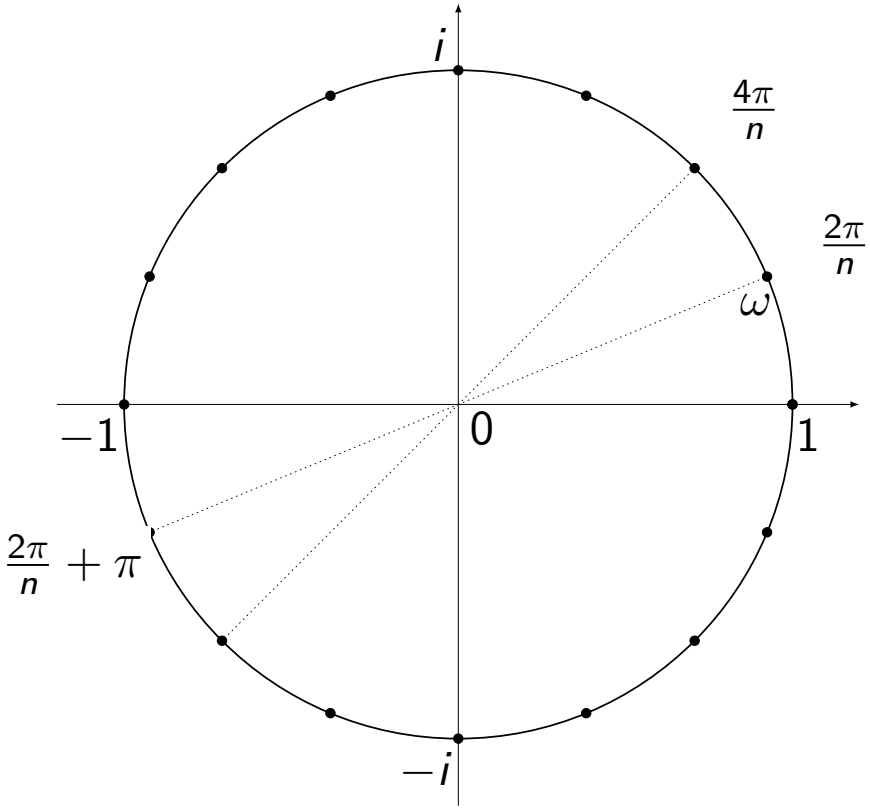
Für $n = 2^L$ gilt:

- $\omega^0 = 1, \omega^{n/2} = -1$.
- Also $j \in \{0, \ldots, n/2 - 1\} : \omega^j = -\omega^{n/2+j}$. Wir finden also $\pm$-Paare.

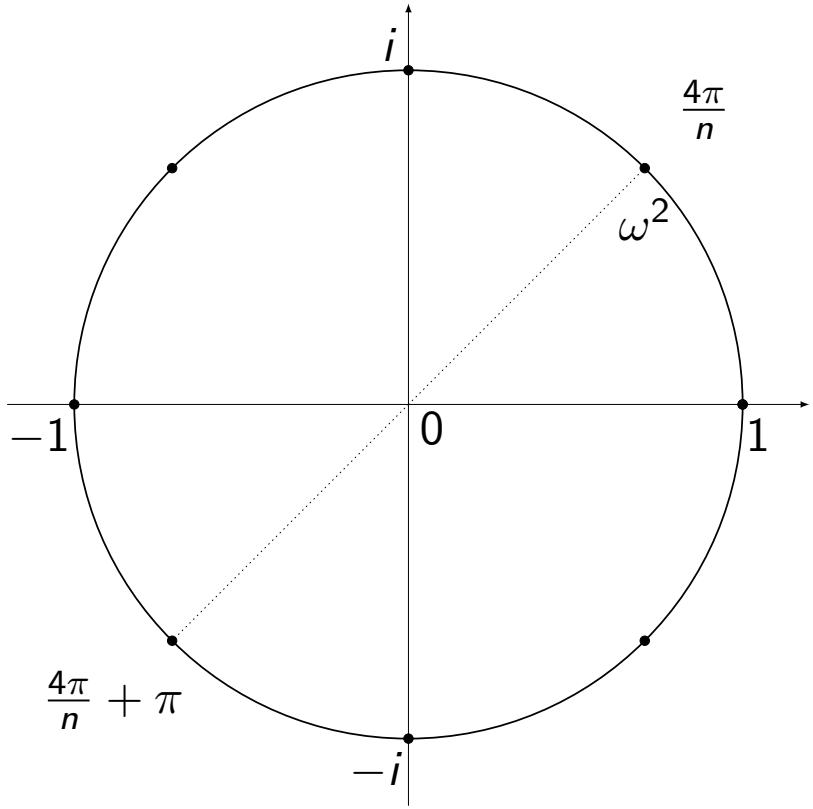
Für $n = 2^L$ gilt:

- $\omega^0 = 1, \omega^{n/2} = -1$.
- Also $j \in \{0, \ldots, n/2 - 1\} : \omega^j = -\omega^{n/2+j}$. Wir finden also $\pm$-Paare.
- ω^2 ist selbst eine primitive Einheitswurzel für $n/2$.
  Im Rekursionsschritt werden wir wieder $\pm$-Paare finden.

16 Stützstellen.



Rekursion: 8 Stützstellen.



Als $(x_0, x_1, \ldots, x_{n-1})$ wählen wir $(\omega^0, \omega^1, \ldots, \omega^{n-1})$.

(Beachte: $\omega^0 = 1$, $\omega^1 = \omega$.)

Als $(x_0, x_1, \ldots, x_{n-1})$ wählen wir $(\omega^0, \omega^1, \ldots, \omega^{n-1})$.

(Beachte: $\omega^0 = 1$, $\omega^1 = \omega$.)

Gegeben $A(x) = \sum_{0 \leq i \leq n-1} a_i x^i$ als Koeffizientenvektor $(a_0, a_1, \ldots, a_{n-1})$, wollen wir also

$$(r_0, r_1, \ldots, r_{n-1}) = (A(1), A(\omega), A(\omega^2), \ldots, A(\omega^{n-1}))$$

berechnen.

Als $(x_0, x_1, \ldots, x_{n-1})$ wählen wir $(\omega^0, \omega^1, \ldots, \omega^{n-1})$.

(Beachte: $\omega^0 = 1$, $\omega^1 = \omega$.)

Gegeben $A(x) = \sum_{0 \leq i \leq n-1} a_i x^i$ als Koeffizientenvektor $(a_0, a_1, \ldots, a_{n-1})$, wollen wir also

$$(r_0, r_1, \ldots, r_{n-1}) = (A(1), A(\omega), A(\omega^2), \ldots, A(\omega^{n-1}))$$

berechnen.

Die Operation $(a_0, \ldots, a_{n-1}) \mapsto (A(1), A(\omega), A(\omega^2), \ldots, A(\omega^{n-1}))$ auf Vektoren heißt die

diskrete Fourier-Transformation (DFT).

Als $(x_0, x_1, \ldots, x_{n-1})$ wählen wir $(\omega^0, \omega^1, \ldots, \omega^{n-1})$.

(Beachte: $\omega^0 = 1$, $\omega^1 = \omega$.)

Gegeben $A(x) = \sum_{0 \leq i \leq n-1} a_i x^i$ als Koeffizientenvektor $(a_0, a_1, \ldots, a_{n-1})$, wollen wir also

$$(r_0, r_1, \ldots, r_{n-1}) = (A(1), A(\omega), A(\omega^2), \ldots, A(\omega^{n-1}))$$

berechnen.

Die Operation $(a_0, \ldots, a_{n-1}) \mapsto (A(1), A(\omega), A(\omega^2), \ldots, A(\omega^{n-1}))$ auf Vektoren heißt die

diskrete Fourier-Transformation (DFT).

(Es gibt auch eine Fourier-Transformation für Funktionen, die auf Integralen beruht.)

Algorithmus FFT (Schnelle Fourier-Transformation)

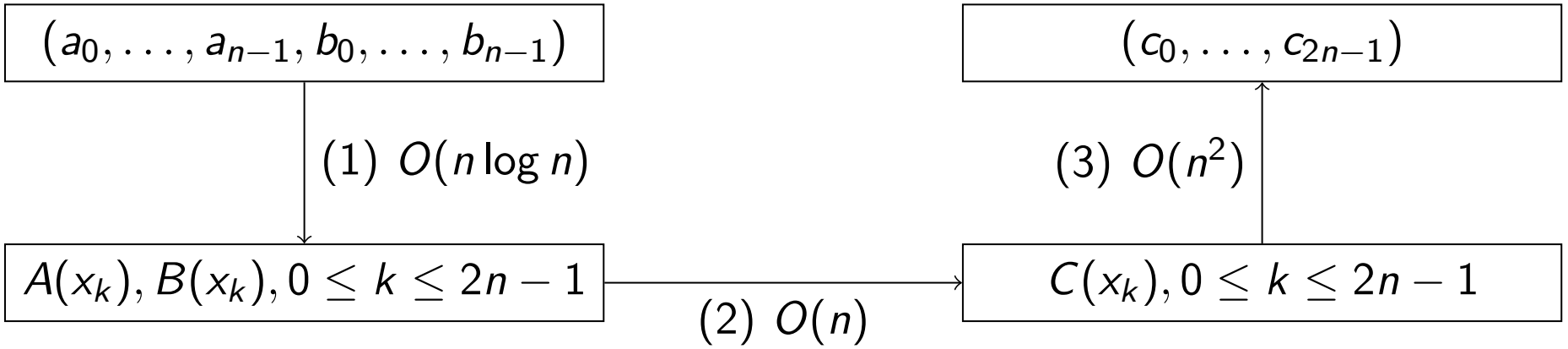
Eingabe: (Koeffizienten-)Vektor $(a_0, \ldots, a_{n-1})$,
für Zweierpotenz n; primitive n-te Einheitswurzel ω.

if $n = 1$ **then return** (a_0);
$(s_0, \ldots, s_{n/2-1}) \leftarrow \mathrm{FFT}((a_0, a_2, \ldots, a_{n-2}), \omega^2)$;
$(t_0, \ldots, t_{n/2-1}) \leftarrow \mathrm{FFT}((a_1, a_3, \ldots, a_{n-1}), \omega^2)$;
for j **from** 0 **to** $n/2 - 1$ **do**
  $r_j \leftarrow s_j + \omega^j \cdot t_j$;
  $r_{n/2+j} \leftarrow s_j - \omega^j \cdot t_j$;
return $(r_0, r_1, \ldots, r_{n-1})$.

Satz 9.7.2

Algorithmus FFT berechnet die diskrete Fouriertransformierte eines Koeffizientenvektors im Bezug auf die Stützstellen $(1, \omega, \omega^2, \ldots, \omega^{n-1})$, wobei ω eine primitive n-te Einheitswurzel ist, in Zeit $O(n \log n)$.

Aktueller Stand:



Es fehlt noch: **Interpolation**.

Auch hier ein schöner **Trick**.

Auch hier ein schöner **Trick**.

Gegeben ist $(r_0, \ldots, r_{n-1})$, gesucht der Koeffizientenvektor $(a_0, \ldots, a_{n-1})$, der $M(\omega) \cdot (a_0, \ldots, a_{n-1})^{\mathsf{T}} = (r_0, \ldots, r_{n-1})^{\mathsf{T}}$ erfüllt, für die Matrix $M(\omega)$ der diskreten Fourier-Transformation:

$$
\begin{aligned}
M(\omega) &= V(1, \omega, \omega^2, \ldots, \omega^{n-1}) = ((\omega^i)^j)_{0 \le i,j \le n-1} \\
&= \begin{pmatrix}
1 & 1 & 1 & \cdots & 1 \\
1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\
1 & \omega^2 & (\omega^2)^2 & \cdots & (\omega^2)^{n-1} \\
\vdots & \vdots & \vdots & & \vdots \\
1 & \omega^{n-1} & (\omega^{n-1})^2 & \cdots & (\omega^{n-1})^{n-1}
\end{pmatrix}.
\end{aligned}
$$

Man kann die zu $M(\omega)$ inverse Matrix direkt bestimmen.

Man kann die zu $M(\omega)$ inverse Matrix direkt bestimmen.

Setze $\bar{\omega} := \omega^{n-1}$.

Man kann die zu $M(\omega)$ inverse Matrix direkt bestimmen.

Setze $\bar{\omega} := \omega^{n-1}$.

Dann gilt $\omega \cdot \bar{\omega} = \omega^n = 1$, also ist $\bar{\omega} = \omega^{-1}$.

Man kann die zu $M(\omega)$ inverse Matrix direkt bestimmen.

Setze $\bar{\omega} := \omega^{n-1}$.

Dann gilt $\omega \cdot \bar{\omega} = \omega^n = 1$, also ist $\bar{\omega} = \omega^{-1}$.

Betrachte

$$M(\bar{\omega}) \quad = \quad ((\bar{\omega}^i)^j)_{0 \leq i,j \leq n-1}$$

Man kann die zu $M(\omega)$ inverse Matrix direkt bestimmen.

Setze $\bar{\omega} := \omega^{n-1}$.

Dann gilt $\omega \cdot \bar{\omega} = \omega^n = 1$, also ist $\bar{\omega} = \omega^{-1}$.

Betrachte

$$
\begin{aligned}
M(\bar{\omega}) &= ((\bar{\omega}^i)^j)_{0 \le i,j \le n-1} \\
&= \begin{pmatrix}
1 & 1 & 1 & \cdots & 1 \\
1 & \bar{\omega} & \bar{\omega}^2 & \cdots & \bar{\omega}^{n-1} \\
1 & \bar{\omega}^2 & (\bar{\omega}^2)^2 & \cdots & (\bar{\omega}^2)^{n-1} \\
\vdots & \vdots & \vdots & & \vdots \\
1 & \bar{\omega}^{n-1} & (\bar{\omega}^{n-1})^2 & \cdots & (\bar{\omega}^{n-1})^{n-1}
\end{pmatrix}
\end{aligned}
$$

Wir berechnen den Eintrag z_{ij} an Stelle (i, j) der Produktmatrix $M(\omega) \cdot M(\bar{\omega})$:

Wir berechnen den Eintrag z_{ij} an Stelle (i, j) der Produktmatrix $M(\omega) \cdot M(\bar{\omega})$:

$$z_{ij} = \sum_{0 \leq k \leq n-1} (\omega^i)^k \cdot (\bar{\omega}^k)^j$$

Wir berechnen den Eintrag z_{ij} an Stelle (i, j) der Produktmatrix $M(\omega) \cdot M(\bar{\omega})$:

$$z_{ij} = \sum_{0 \leq k \leq n-1} (\omega^i)^k \cdot (\bar{\omega}^k)^j = \sum_{0 \leq k \leq n-1} (\omega^{i+(n-1)j})^k.$$

Wir berechnen den Eintrag z_{ij} an Stelle (i, j) der Produktmatrix $M(\omega) \cdot M(\bar{\omega})$:

$$z_{ij} = \sum_{0 \le k \le n-1} (\omega^i)^k \cdot (\bar{\omega}^k)^j = \sum_{0 \le k \le n-1} (\omega^{i+(n-1)j})^k.$$

Es gibt zwei Fälle. Wenn $i = j$ gilt, ist $\omega^{i+(n-1)j} = (\omega^n)^i = 1$, und die Summe ist n.

Wir berechnen den Eintrag z_{ij} an Stelle (i, j) der Produktmatrix $M(\omega) \cdot M(\bar{\omega})$:

$$z_{ij} = \sum_{0 \le k \le n-1} (\omega^i)^k \cdot (\bar{\omega}^k)^j = \sum_{0 \le k \le n-1} (\omega^{i+(n-1)j})^k.$$

Es gibt zwei Fälle. Wenn $i = j$ gilt, ist $\omega^{i+(n-1)j} = (\omega^n)^i = 1$, und die Summe ist n.

Wenn $i > j$ gilt, ist $\omega^{i+(n-1)j} = \omega^\ell$ für $1 \le \ell = i - j \le n - 1$.
Wenn $i < j$ gilt, ist $\omega^{i+(n-1)j} = \omega^\ell$ für $1 \le \ell = n - (j - i) \le n - 1$.

Wir berechnen den Eintrag z_{ij} an Stelle (i, j) der Produktmatrix $M(\omega) \cdot M(\bar{\omega})$:

$$z_{ij} = \sum_{0 \le k \le n-1} (\omega^i)^k \cdot (\bar{\omega}^k)^j = \sum_{0 \le k \le n-1} (\omega^{i+(n-1)j})^k.$$

Es gibt zwei Fälle. Wenn $i = j$ gilt, ist $\omega^{i+(n-1)j} = (\omega^n)^i = 1$, und die Summe ist n.

Wenn $i > j$ gilt, ist $\omega^{i+(n-1)j} = \omega^\ell$ für $1 \le \ell = i - j \le n - 1$.
Wenn $i < j$ gilt, ist $\omega^{i+(n-1)j} = \omega^\ell$ für $1 \le \ell = n - (j - i) \le n - 1$.
Wegen Bedingung (ii):

$$z_{ij} = \sum_{0 \le k \le n-1} (\omega^\ell)^k = 0.$$

Also: $M(\omega) \cdot M(\bar{\omega}) = n \cdot I_n$, für die $n \times n$-Einheitsmatrix I_n.

Also: $M(\omega) \cdot M(\bar{\omega}) = n \cdot I_n$, für die $n \times n$-Einheitsmatrix I_n.

Das heißt: $M(\omega)^{-1} = n^{-1} \cdot M(\bar{\omega})$.

Also: $M(\omega) \cdot M(\bar{\omega}) = n \cdot I_n$, für die $n \times n$-Einheitsmatrix I_n.

Das heißt: $M(\omega)^{-1} = n^{-1} \cdot M(\bar{\omega})$.

Wenn also $(r_0, \ldots, r_{n-1})$ der für die Interpolation gegebene Wertevektor ist, so erhalten wir den Koeffizientenvektor als

$$\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = n^{-1} \cdot M(\bar{\omega}) \cdot \begin{pmatrix} r_0 \\ r_1 \\ \vdots \\ r_{n-1} \end{pmatrix}.$$

Für die Zahl $\bar{\omega}$ beobachten wir:

(i) $\bar{\omega}^n = (\omega^{n-1})^n = (\omega^n)^{n-1} = 1^{n-1} = 1$,

(ii) für $1 \leq k \leq n-1$ gilt $\bar{\omega}^k = (\omega^{n-1})^k = \omega^{n(k-1)}\omega^{n-k} = \omega^{n-k}$, mit $1 \leq n-k \leq n-1$, also $\sum_{0 \leq j \leq n-1} (\bar{\omega}^k)^j = 0$.

Für die Zahl $\bar{\omega}$ beobachten wir:

(i) $\bar{\omega}^n = (\omega^{n-1})^n = (\omega^n)^{n-1} = 1^{n-1} = 1$,

(ii) für $1 \leq k \leq n-1$ gilt $\bar{\omega}^k = (\omega^{n-1})^k = \omega^{n(k-1)}\omega^{n-k} = \omega^{n-k}$, mit $1 \leq n-k \leq n-1$, also $\sum_{0 \leq j \leq n-1} (\bar{\omega}^k)^j = 0$.

Das heißt: Auch $\bar{\omega}$ ist eine primitive n-te Einheitswurzel.

Für die Zahl $\bar{\omega}$ beobachten wir:

(i) $\bar{\omega}^n = (\omega^{n-1})^n = (\omega^n)^{n-1} = 1^{n-1} = 1$,

(ii) für $1 \leq k \leq n-1$ gilt $\bar{\omega}^k = (\omega^{n-1})^k = \omega^{n(k-1)}\omega^{n-k} = \omega^{n-k}$, mit $1 \leq n-k \leq n-1$, also $\sum_{0 \leq j \leq n-1} (\bar{\omega}^k)^j = 0$.

Das heißt: Auch $\bar{\omega}$ ist eine primitive n-te Einheitswurzel.

Multiplikation mit $M(\bar{\omega})$ entspricht der FFT-Operation mit $\bar{\omega}$ an Stelle von ω. – Wir erhalten für die Interpolation:

$$(a_0, \ldots, a_{n-1}) = n^{-1} \cdot \mathrm{FFT}((r_0, \ldots, r_{n-1}), \bar{\omega}).$$

Zeitaufwand: $O(n \log n)$.

Wir können nun unseren Algorithmenplan umsetzen:

Algorithmus FFT-PM (Polynommultiplikation)

Eingabe: (Koeffizienten-)Vektoren $(a_0, \ldots, a_{n-1})$, $(b_0, \ldots, b_{n-1})$, für Zweierpotenz n.

Berechne $2n$-te Einheitswurzeln $\omega \leftarrow e^{\pi i/n}$ und $\bar{\omega} \leftarrow \omega^{2n-1}$.

$(r_0^A, \ldots, r_{2n-1}^A) \leftarrow \mathrm{FFT}((a_0, \ldots, a_{n-1}, 0, \ldots, 0), \omega)$;

$(r_0^B, \ldots, r_{2n-1}^B) \leftarrow \mathrm{FFT}((b_0, \ldots, b_{n-1}, 0, \ldots, 0), \omega)$;

for j **from** 0 **to** $2n-1$ **do** $r_j^C \leftarrow r_j^A \cdot r_j^B$;

$(c_0, c_1, \ldots, c_{2n-1}) \leftarrow \frac{1}{2n} \cdot \mathrm{FFT}((r_0^C, \ldots, r_{2n-1}^C), \bar{\omega})$;

return $(c_0, c_1, \ldots, c_{2n-1})$.

Satz 9.7.3

Algorithmus FFT-PM berechnet die Koeffizienten des Produktes zweier durch Koeffizientenvektoren gegebener Polynome in Zeit $O(n \log n)$.

Satz 9.7.3

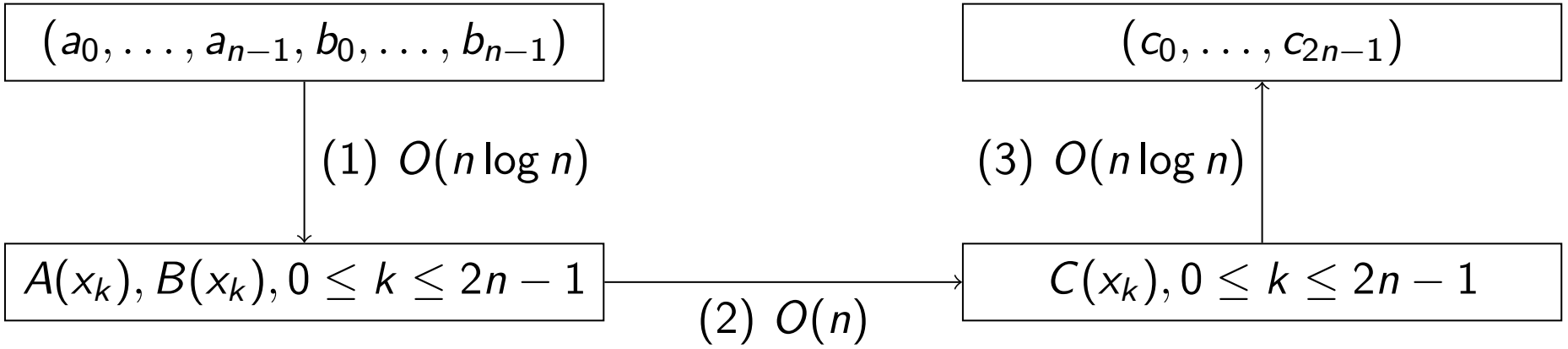
Algorithmus FFT-PM berechnet die Koeffizienten des Produktes zweier durch Koeffizientenvektoren gegebener Polynome in Zeit $O(n \log n)$.

Beweis: Siehe vorherige Überlegungen.
(Rechenaufwand: $O(2n \log(2n)) = O(n \log n)$.)

Satz 9.7.3

Algorithmus FFT-PM berechnet die Koeffizienten des Produktes zweier durch Koeffizientenvektoren gegebener Polynome in Zeit $O(n \log n)$.

Beweis: Siehe vorherige Überlegungen.
(Rechenaufwand: $O(2n \log(2n)) = O(n \log n)$.)



Anmerkung: Um Polynome vom Grad $k-1$ und vom Grad $\ell-1$ zu multiplizieren, genügt $n=2^L$ für $L=\lceil \log_2(k+\ell-1)\rceil$.

Für Zuhause: Es folgen weitere Anmerkungen zur Geschichte und Anwendung der FFT. **(Nicht prüfungsrelevant.)**

Zum Üben: Berechnen Sie $(x+1)\cdot(x^2+1)$ mittels FFT!

Der FFT-Ansatz ermöglicht es auch, die diskrete Fourier-Transformierte eines Vektors sehr effizient parallel zu berechnen, entweder mit mehreren Prozessoren oder sogar in Hardware.

Der FFT-Ansatz ermöglicht es auch, die diskrete Fourier-Transformierte eines Vektors sehr effizient parallel zu berechnen, entweder mit mehreren Prozessoren oder sogar in Hardware.

Die Überlegungen hierzu führen auch zu einer sehr effizienten iterativen Formulierung des Algorithmus.

Der FFT-Ansatz ermöglicht es auch, die diskrete Fourier-Transformierte eines Vektors sehr effizient parallel zu berechnen, entweder mit mehreren Prozessoren oder sogar in Hardware.

Die Überlegungen hierzu führen auch zu einer sehr effizienten iterativen Formulierung des Algorithmus.

In Algorithmus FFT können die beiden rekursiven Aufrufe unabhängig voneinander parallel durchgeführt werden (in Hardware: zwei Kopien der gleichen Schaltung nötig). Die Berechnung des Resultates $(r_0, \ldots, r_{n-1})$ kann sogar für alle n Werte gleichzeitig erfolgen.

Der FFT-Ansatz ermöglicht es auch, die diskrete Fourier-Transformierte eines Vektors sehr effizient parallel zu berechnen, entweder mit mehreren Prozessoren oder sogar in Hardware.

Die Überlegungen hierzu führen auch zu einer sehr effizienten iterativen Formulierung des Algorithmus.

In Algorithmus FFT können die beiden rekursiven Aufrufe unabhängig voneinander parallel durchgeführt werden (in Hardware: zwei Kopien der gleichen Schaltung nötig). Die Berechnung des Resultates $(r_0, \ldots, r_{n-1})$ kann sogar für alle n Werte gleichzeitig erfolgen.

Eine Beispielschaltung findet man auf Seite 69 im Buch von S. Dasgupta, C. Papadimitriou, U. Vazirani, **Algorithms**, McGraw-Hill, 2007.

Wozu eigentlich Polynommultiplikation oder „Faltung“

$$c_k = \sum_{i,j\,:\,i+j=k} a_i b_j, \quad \text{für } 0 \leq k \leq 2n-1 \text{ ?}$$

Wozu eigentlich Polynommultiplikation oder „Faltung“

$$c_k = \sum_{i,j\,:\,i+j=k} a_i b_j, \quad \text{für } 0 \leq k \leq 2n-1 \text{ ?}$$

Zentrales Hilfsmittel bei der „Digitalen Signalverarbeitung“.

Offizielle Publikation: [Cooley/Tukey 1965].

Form des Algorithmus 1805 von C. F. Gauß entworfen (und zur Berechnung von Asteroidenbahnen benutzt).

Erstmalig publiziert wurde eine Variante des Algorithmus von C. Runge (1903/05).

(Quelle hierfür: Wikipedia – kann stimmen, muss aber nicht.)

Wir stellen uns ein System vor, in dem regelmäßig Signale anfallen und (zu Ausgabesignalen) verarbeitet werden.

Wir stellen uns ein System vor, in dem regelmäßig Signale anfallen und (zu Ausgabesignalen) verarbeitet werden.

Abtastzeitpunkte $t_0, t_1, \ldots$ mit festem Abstand $\Delta = t_{i+1} - t_i$ liefern (reelle, komplexe) Signalwerte $a_0, a_1, a_2, \ldots$.

Wir stellen uns ein System vor, in dem regelmäßig Signale anfallen und (zu Ausgabesignalen) verarbeitet werden.

Abtastzeitpunkte $t_0, t_1, \ldots$ mit festem Abstand $\Delta = t_{i+1} - t_i$ liefern (reelle, komplexe) Signalwerte $a_0, a_1, a_2, \ldots$.

Ein Verarbeitungsmechanismus soll diese Messwertfolge in eine Ausgabefolge $c_0, c_1, c_2, \ldots$ umsetzen, mit Ausgabezeiten $t'_0, t'_1, t'_2, \ldots$, ebenso mit Abstand Δ.

Die einfachsten Mechanismen sind **linear** ...

(wenn man zwei Signalfolgen addiert, addieren sich die Ausgabefolgen, ebenso bei Multiplikation mit konstanten Faktoren)

Die einfachsten Mechanismen sind **linear** ...

(wenn man zwei Signalfolgen addiert, addieren sich die Ausgabefolgen, ebenso bei Multiplikation mit konstanten Faktoren)

und **zeitinvariant** ...

(wenn die gleiche Signalfolge um einen Zeitschritt Δ versetzt auftritt, ergibt sich die gleiche um Δ versetzte Ausgabefolge).

Leicht zu sehen: Bei linearen, zeitinvarianten Signalverarbeitungs-Systemen ist durch die Ausgabefolge $(b_0, b_1, b_2, \ldots)$, die von einem Signal der Größe 1 bei t_0 und sonst nur Nullsignalen ausgelöst wird, die Ausgabefolge $(c_0, c_1, c_2, \ldots)$ auf einer beliebigen Signalfolge $(a_0, a_1, \ldots)$ eindeutig bestimmt, durch:

$$c_k = \sum_{i+j=k} a_i b_j.$$

Das ist gerade die Folge der Koeffizienten des Produktpolynoms!

D.h.: Lineare, zeitinvariante Reaktion auf Mess-Signale führt unmittelbar zum Problem „Polynommultiplikation“.

„Multiplikation ganzer Zahlen“:

Wir wollen zwei Zahlen x, y mit je N Bits multiplizieren, N ist extrem groß.

„Multiplikation ganzer Zahlen“:

Wir wollen zwei Zahlen x, y mit je N Bits multiplizieren, N ist extrem groß.

Wir zerlegen die Binärdarstellung von x und y in Blöcke der Länge $\sqrt{N}$. Indem wir jeweils jeden dritten Block nehmen, bilden wir aus x und y jeweils drei Zahlen, so dass

$$x = x_0 + x_1 + x_2 \text{ und } y = y_0 + y_1 + y_2$$

gilt und so dass $x_i \cdot y_j$ als Multiplikation von Zahlen mit $\sqrt{N}$ Ziffern im Bereich $[\sqrt{N}]$ aufgefasst werden kann. Durch die großen eingestreuten Blöcke von Nullen gibt es keine Überträge.

Die „Ziffern“ der Produkte $x_i \cdot y_j$ stellen sich als die Koeffizienten eines Produktpolynoms heraus.

Mit der FFT könnte man diese Ziffern der Teilprodukte in $O(\sqrt{N} \log N)$ Multiplikationen von komplexen Zahlen ermitteln.

Unter der Annahme, dass eine solche Multiplikation (durch geeignete Rundung auf komplexe Zahlen mit Darstellungslänge $O(\sqrt{N})$) in Zeit $O(N)$ möglich ist, erhalten wir einen Gesamtzeitaufwand von $O(N^{3/2} \log N)$, sogar besser als der Zeitbedarf des Karatsuba-Multiplizierers.

Unter der Annahme, dass eine solche Multiplikation (durch geeignete Rundung auf komplexe Zahlen mit Darstellungslänge $O(\sqrt{N})$) in Zeit $O(N)$ möglich ist, erhalten wir einen Gesamtzeitaufwand von $O(N^{3/2} \log N)$, sogar besser als der Zeitbedarf des Karatsuba-Multiplizierers.

Durch Einziehen von Rekursionsstufen lässt sich die Zeit weiter drücken; wenn man es konsequent durchführt, auf $O(N^{1+\varepsilon})$ für beliebige konstante $\varepsilon > 0$ (Verfahren von Toom/Cook, z. B. im Buch „The Art of Computer Programming, Vol. 2: Seminumerical Algorithms“ von D. Knuth beschrieben).

Unter der Annahme, dass eine solche Multiplikation (durch geeignete Rundung auf komplexe Zahlen mit Darstellungslänge $O(\sqrt{N})$) in Zeit $O(N)$ möglich ist, erhalten wir einen Gesamtzeitaufwand von $O(N^{3/2} \log N)$, sogar besser als der Zeitbedarf des Karatsuba-Multiplizierers.

Durch Einziehen von Rekursionsstufen lässt sich die Zeit weiter drücken; wenn man es konsequent durchführt, auf $O(N^{1+\varepsilon})$ für beliebige konstante $\varepsilon > 0$ (Verfahren von Toom/Cook, z. B. im Buch „The Art of Computer Programming, Vol. 2: Seminumerical Algorithms“ von D. Knuth beschrieben).

Für den Multiplikationsalgorithmus von Schönhage und Strassen mit Kosten $O(N \log N \log \log N)$ sind neben der FFT noch weitere Ideen nötig.

Bemerkung für $\mathbb{C}$-Vektorraum-Spezialisten:

Die DFT ist eine Koordinatentransformation für einen Basiswechsel im $\mathbb{C}$-Vektorraum P_n der Polynome vom Grad bis zu $n-1$.

Man betrachtet ein inneres Produkt in P_n:

$$\left\langle \sum_i a_i X^i, \sum_i b_i X^i \right\rangle = \sum_{0 \le i \le n-1} a_i \overline{b_i}.$$

Definiere $f_j(X)$ als das (eindeutig bestimmte) Polynom, das $f_j(\omega^k) = [j = k]$ erfüllt, für $0 \le k \le n-1$. Dann ist die „Standardbasis“ $B_n = (1, X, X^2, \ldots, X_{n-1})$ Orthonormalbasis von P_n und die „Fourierbasis“ $F_n = (f_0(X), f_1(X), \ldots, f_{n-1}(X))$ Orthogonalbasis von P_n.

Die Matrix $M(\omega)$ ist die Matrix für den Basiswechsel von B_n (Koeffizienten $(a_0, \ldots, a_{n-1})$ nach F_n (Koeffizienten $(A(\omega^0), \ldots, A(\omega^{n-1}))$), die Matrix $M(\omega)^{-1} = \frac{1}{n} M(\bar{\omega})$ ist für den umgekehrten Basiswechsel zuständig.

Für die Arbeit mit ganzen Zahlen ist die oben beschriebene FFT sehr unbequem, weil man mit komplexen Zahlen rechnen muss.

In gewissen Situationen kann man FFT auch „modulo m“ durchführen, für geeignete Teiler m, und ganz im diskreten Bereich bleiben.

Für die Arbeit mit ganzen Zahlen ist die oben beschriebene FFT sehr unbequem, weil man mit komplexen Zahlen rechnen muss.

In gewissen Situationen kann man FFT auch „modulo m“ durchführen, für geeignete Teiler m, und ganz im diskreten Bereich bleiben.

Fakt 9.7.4

Wenn $w = 2$ und n eine Zweierpotenz ist, dann gilt für $m = 2^{n/2} + 1$:

(i) $w^n \bmod m = 1$ und $w^{n/2} \bmod m = -1$.

(ii) Für $1 \leq k \leq n-1$ gilt $\sum_{0 \leq j \leq n-1} (w^k)^j = 0$.

(iii) Für $1 \leq k \leq n-1$ gilt $w^k \neq 1$.

(iv) Es gibt $\hat{n}, \hat{w}$ mit $(n \cdot \hat{n}) \bmod m = 1$ und $(w \cdot \hat{w}) \bmod m = 1$.

Die Zahl w spielt also im Ring $\mathbb{Z}_m$ die Rolle einer n-ten Einheitswurzel, sie besitzt eine Inverse $\hat{w}$, und auch n hat eine Inverse $\hat{n}$.

Beispiel: $w = 2, n = 32, m = 2^{16} + 1 = 65537, \hat{w} = 32769, \hat{n} = 63489$.

Man kann sich überlegen, dass in dieser Situation alle Überlegungen, die wir oben für $\omega, \bar{\omega}, n, n^{-1}$ angestellt haben, für $w, \hat{w}, n, \hat{n}$ ebenso funktionieren, wir also den FFT-Algorithmus auch mit Addition und Multiplikation „modulo m“ benutzen können, obwohl $\mathbb{Z}_m$ normalerweise kein Körper ist. Damit lassen sich die Koeffizienten von Produkten ganzzahliger Polynome mit FFT ohne Umweg über die komplexen Zahlen berechnen, und auch in der Situation der Multiplikation von ganzen Zahlen ist diese FFT-Version günstiger. Einzige Voraussetzung: m muss so groß sein, dass die Koeffizienten c_k des Produktpolynoms schon durch den Wert c_k mod m eindeutig bestimmt sind.

SS 2021

Algorithmen und Datenstrukturen

10. Kapitel

Greedy-Algorithmen: Prinzipien

Martin Dietzfelbinger

Juli 2021

Kapitel 10: Greedy-Algorithmen

Greedy*-Algorithmen sind anwendbar bei **Konstruktionsaufgaben**, deren Ziel es ist, eine **optimale Struktur zu finden**, die aus mehreren Komponenten besteht.

Sie finden eine Lösung, die sie schrittweise aufbauen.

In jedem Schritt wird eine „lokal“ oder aus der aktuellen Sicht optimale Entscheidung getroffen, ohne „an die Zukunft zu denken“.
(Was dies konkret bedeutet, versteht man besser anhand der später betrachteten Beispiele.)

Es werden dabei nicht mehr Teillösungen konstruiert als unbedingt nötig.

Es werden nie Entscheidungen korrigiert oder zurückgesetzt.

---

* greedy (*engl.*): gierig.

10.1 Zwei Beispiele

Beispiel 1: Hörsaalbelegung

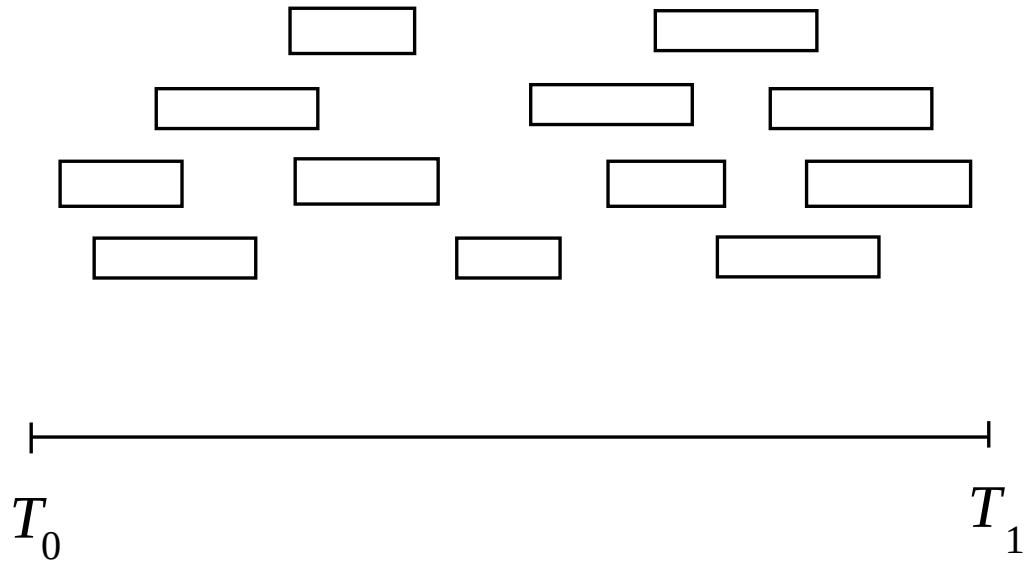
Gegeben: Veranstaltungsort (Hörsaal), Zeitspanne $[T_0, T_1)$ (z. B. 7 Uhr bis 21 Uhr) und eine Menge von n Aktionen (Vorlesungen oder ähnliches), die durch Start- und Endzeit spezifiziert sind:

$$[s_i, f_i) \quad \text{mit } T_0 \leq s_i < t_i \leq T_1, \text{ für } 1 \leq i \leq n.$$

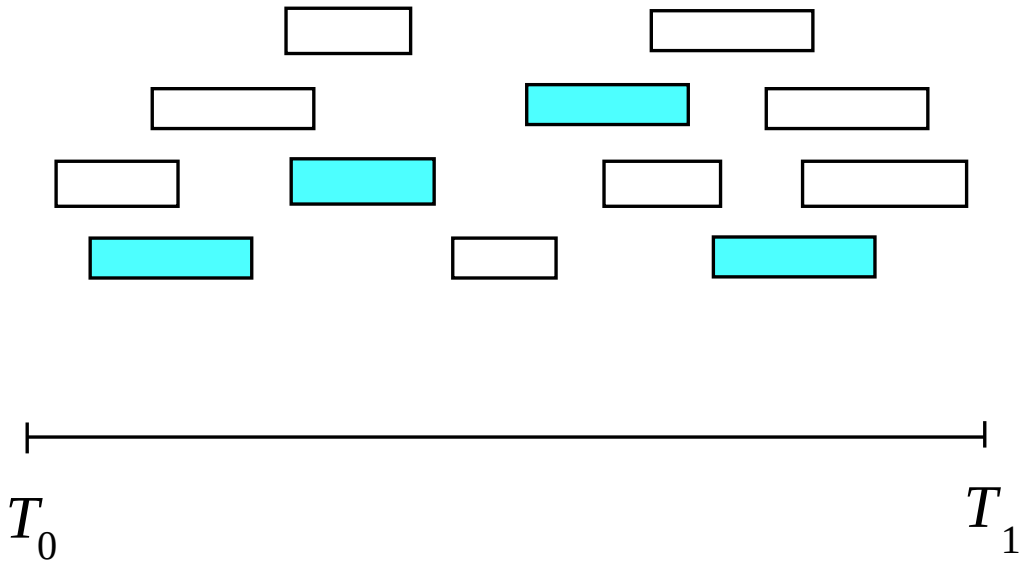
Gesucht: Belegung des Hörsaals, die **möglichst viele** Ereignisse mit disjunkten Zeitspannen stattfinden lässt.

$A \subseteq \{1, \ldots, n\}$ heißt **zulässig**, wenn alle $[s_i, f_i)$, $i \in A$, disjunkt sind.

Aufgabe, formal: Finde eine zulässige Menge A mit $|A|$ so groß wie möglich.

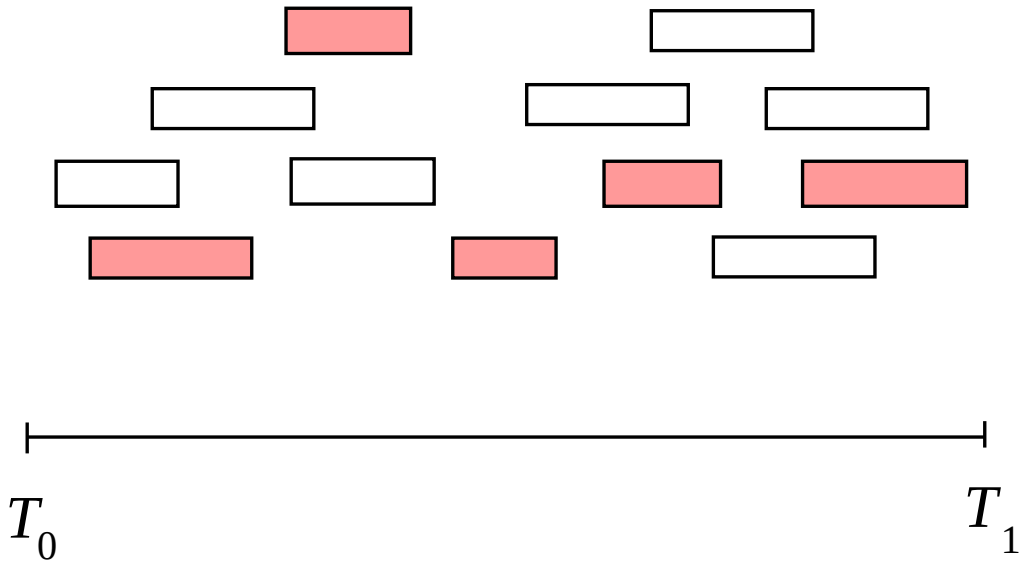


Eingabe: Aktionen mit Beginn und Ende.



Eingabe: Aktionen mit Beginn und Ende.

Zulässige Lösung mit **4** Aktionen. Ist sie optimal?

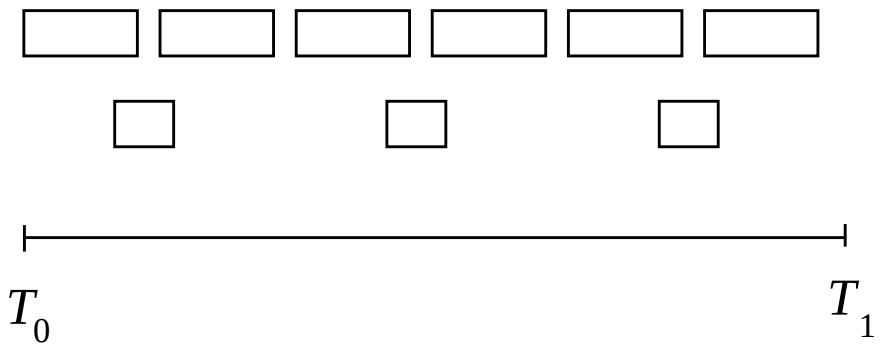


Eingabe: Aktionen mit Beginn und Ende.

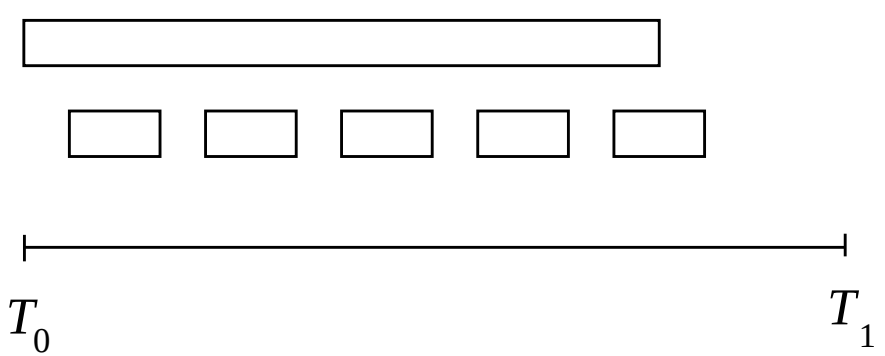
Zulässige Lösung mit **5** Aktionen. Ist sie optimal?

Ansätze, die nicht funktionieren:

- Zuerst kurze Ereignisse planen



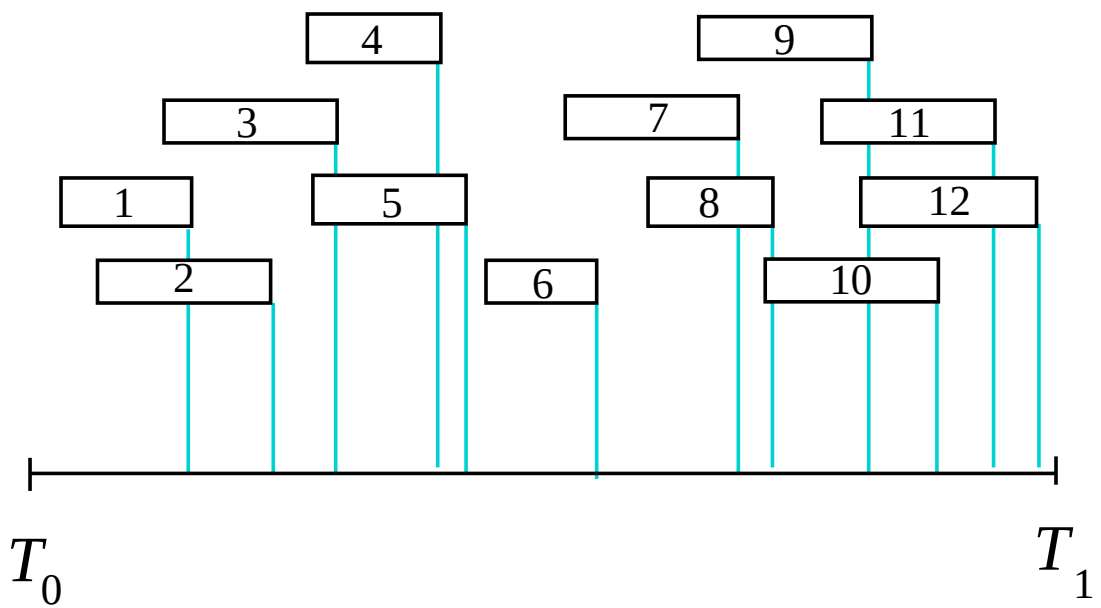
- Immer ein Ereignis mit möglichst früher Anfangszeit wählen



Trick: Bearbeite Ereignisse in der Reihenfolge **wachsender Schlusszeiten**.

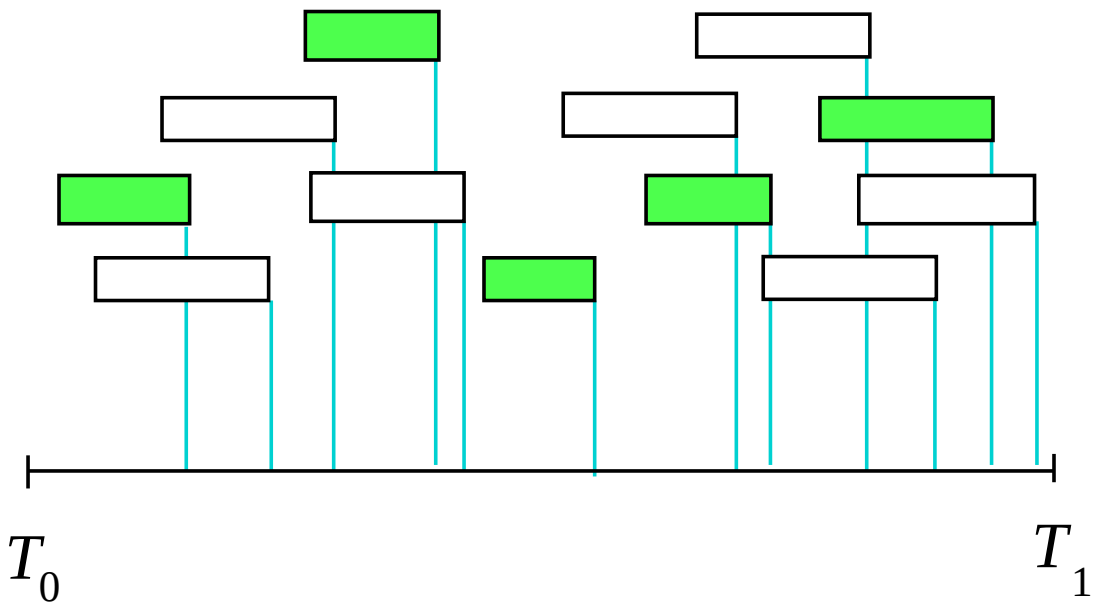
O.B.d.A.: Veranstaltungen nach Schlusszeiten aufsteigend sortiert (Zeitaufwand $O(n \log n)$), also:

$$f_1 \leq f_2 \leq \cdots \leq f_n.$$



Wiederhole: Wähle die **wählbare** Aktion mit der kleinsten Schlusszeit und füge sie zum Belegungsplan hinzu.

Eine Aktion ist **wählbar**, wenn ihre Startzeit mindestens so groß wie die Schlusszeit der letzten schon geplanten Veranstaltung ist.



Ausgabe von GS: Zulässige Lösung.

Algorithmus Greedy Scheduling (GS)

Eingabe: Reelle Zahlen $T_0 < T_1$, Paare $(s_1, t_1), \ldots, (s_n, f_n)$, mit
$T_0 \leq s_i < f_i \leq T_1$ für $1 \leq i \leq n$

Ausgabe: Maximal großes $A \subseteq \{1, \ldots, n\}$ mit $[s_i, f_i), i \in A$, disjunkt

(1) Sortiere Paare gemäß $f_1, \ldots, f_n$, nummeriere um, so dass $f_1 \leq \cdots \leq f_n$;
(2) $A \leftarrow \{1\}$;
(3) $f_{\text{last}} \leftarrow f_1$;
(4) **for** i **from** 2 **to** n **do**
(5) **if** $s_{\mathtt{i}} \geq f_{\text{last}}$ **then** // $[s_{\mathtt{i}}, f_{\mathtt{i}})$ wählbar
(6) $A \leftarrow A \cup \{\mathtt{i}\}$;
(7) $f_{\text{last}} \leftarrow f_{\mathtt{i}}$;
(8) **return** A

Satz 10.1.1

Der Algorithmus Greedy Scheduling (GS) hat lineare Rechenzeit (bis auf die Sortierkosten von in Zeile (1), höchstens $O(n \log n)$) und löst das Hörsaalplanungsproblem optimal.

Beweis: **Rechenzeit:** Sortieren kostet Zeit $O(n \log n)$; der restliche Algorithmus hat offensichtlich Laufzeit $O(n)$.

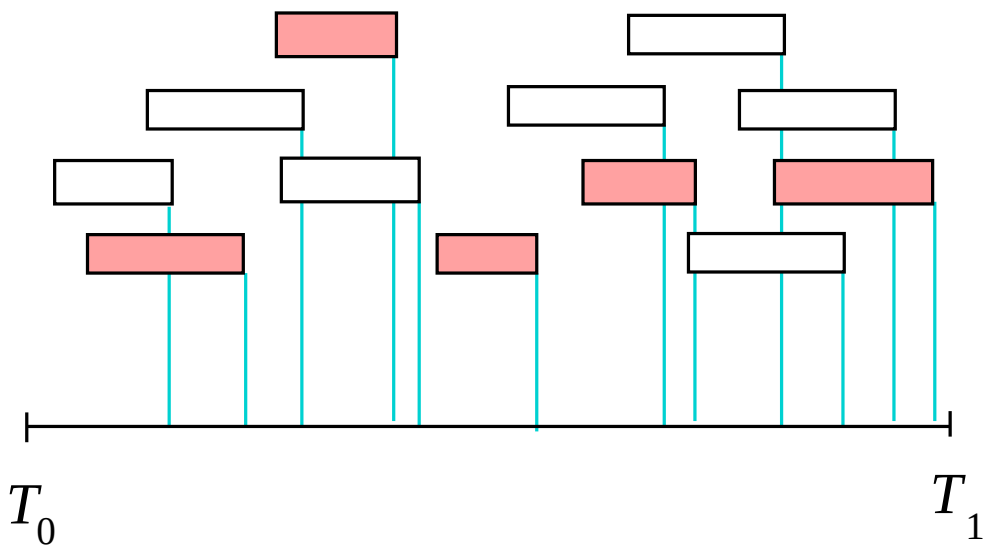
Korrektheit: Wir behaupten: Algorithmus GS liefert auf Inputs mit Größe n eine optimale Lösung. Dies beweisen wir durch **vollständige Induktion**.

Der Fall $n = 1$ ist trivial.

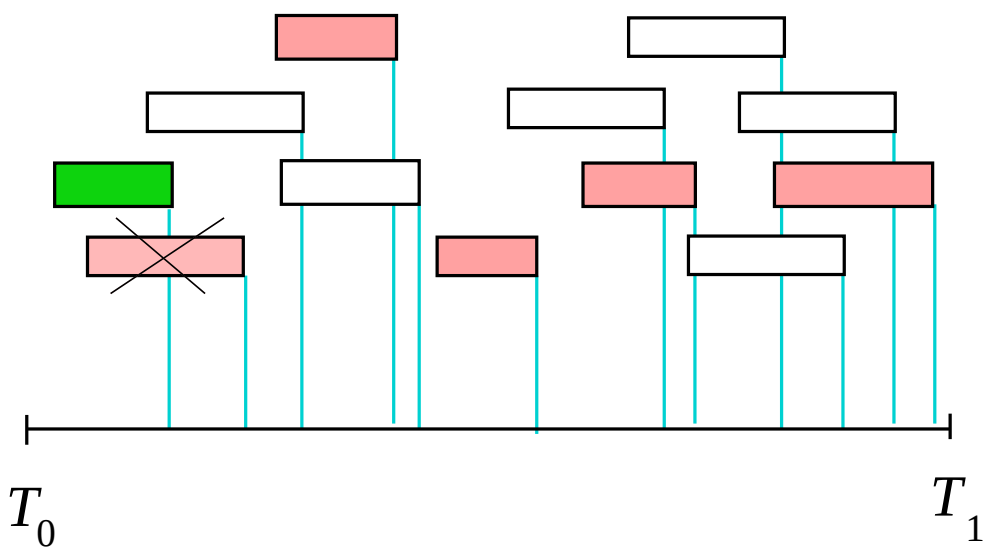
Sei nun $n > 1$.

I.V.: GS liefert für Inputs der Länge $n' < n$ eine optimale Lösung.

Ind.-Schritt: Sei $B \subseteq \{1, \ldots, n\}$ eine **optimale** Lösung, $|B| = r$.
(Im Beispiel: $r = 5$.)



1. Beobachtung:
Es gibt eine Lösung B', die mit dem Intervall $[s_1, f_1)$ (dem ersten Schritt des Greedy-Algorithmus) startet und ebenfalls r Ereignisse hat. (B' ist also auch optimal.)



Setze $B' := (B - \{\min(B)\}) \cup \{1\}$.
Intervalle aufsteigend sortiert $\Rightarrow f_1 \leq f_{\min(B)}$.

2. Beobachtung:

Die Menge $B - \{\min(B)\}$ **löst das Teilproblem**

$$\boldsymbol{f_1, T_1, (s_i, f_i), s_i \geq f_1} \qquad (*)$$

optimal. Anschaulich: In $[f_1, T_1)$ können maximal $r - 1$ dieser Ereignisse untergebracht werden.

Wieso? Sonst würden wir $[s_1, f_1)$ mit einer besseren Lösung für $(*)$ zu einer besseren Lösung für das Gesamtproblem kombinieren: Widerspruch zur Optimalität von B.

Wenn wir also eine beliebige optimale Lösung C für $(*)$ finden, so hat diese $r - 1$ Elemente.

Wenn wir zu einem solchen optimalen C das Ereignis (s_1, f_1) hinzufügen, also $C \cup \{1\}$ bilden, erhalten wir eine Lösung der Größe r für die ursprüngliche Eingabe. Diese ist optimal.

3. Beobachtung:

Algorithmus GS auf Eingabe $T_0, T_1, (s_i, f_i), 1 \leq i \leq n$, hat ab Iteration $i = 2$ **genau dasselbe Verhalten** wie wenn man GS auf f_1, T_1, $(s_i, f_i), s_i \geq f_1$, starten würde.

Nach I.V. liefert also dieser Teil des Algorithmus eine optimale Lösung C mit $r - 1$ Ereignissen für $(*)$.

(mit 2. Beob.) $\Rightarrow$ Greedy Scheduling liefert insgesamt eine optimale Lösung der Größe r. $\square$

Beispiel 2: Fraktionales („teilbares“) Rucksackproblem

Veranschaulichung: Ein Dieb stiehlt Säckchen mit Edelmetallkrümeln (Gold, Silber, Platin) und Edelsteinen. Er hat einen Rucksack mit Volumen b dabei. Durch teilweises Ausleeren der Säckchen kann er beliebige Teilvolumina herstellen. Der Wert pro Volumeneinheit ist unterschiedlich für unterschiedliche Materialien. Was soll er in seinen Rucksack mit Volumen b packen, um den Wert der Beute zu maximieren?

Dazu: Benenne den Bruchteil $\lambda_i \in [0, 1]$, den er von Säckchen i mitnehmen soll.

Gegeben: (n Objekte mit:)
Volumina $a_1, \ldots, a_n > 0$, **Nutzenwerte** $c_1, \ldots, c_n > 0$, **Volumenschranke** b.

Gesucht: Vektor $(\lambda_1, \ldots, \lambda_n) \in [0, 1]^n$, so dass

$$\lambda_1 a_1 + \ldots + \lambda_n a_n \leq b \quad \textbf{(„zulässig“)}$$

und

$$\lambda_1 c_1 + \ldots + \lambda_n c_n \text{ maximal.}$$

Beim **„0-1-Rucksackproblem“** werden nur 0-1-Vektoren mit $\lambda_i \in \{0, 1\}$ zugelassen.

Mitteilung (im Vorgriff auf „Automaten, Sprachen und Komplexität“, 3. Sem.): Die $\{0, 1\}$-Version ist **NP-vollständig**, besitzt also wahrscheinlich keinen effizienten Algorithmus.

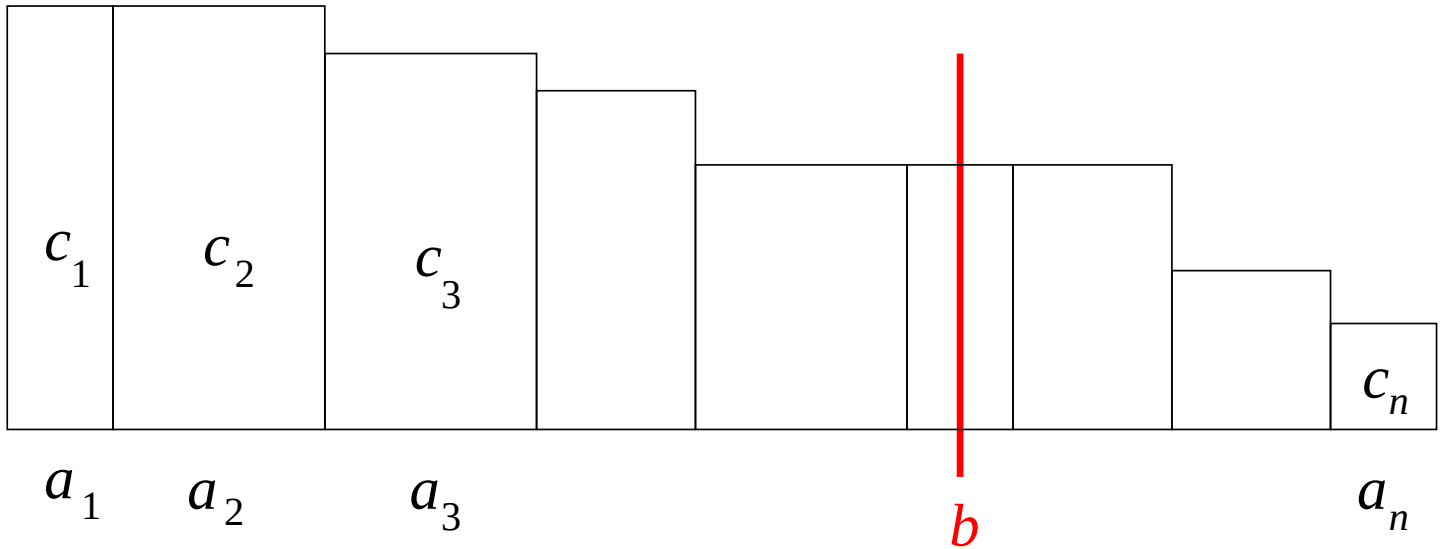
Das fraktionale Rucksackproblem ist mit einem Greedy-Algorithmus in Zeit $O(n \log n)$ lösbar.

Kern der Lösungsidee: Berechne **„Nutzendichten“**

$$d_i = \frac{c_i}{a_i}, 1 \leq i \leq n,$$

und sortiere die Objekte gemäß d_i fallend.

Nehme von vorne beginnend möglichst viele ganze Objekte, bis schließlich das letzte Objekt teilweise genommen wird, so dass die Volumenschranke vollständig ausgeschöpft wird.



Input, sortiert nach Nutzendichte $d_i = c_i/a_i$.

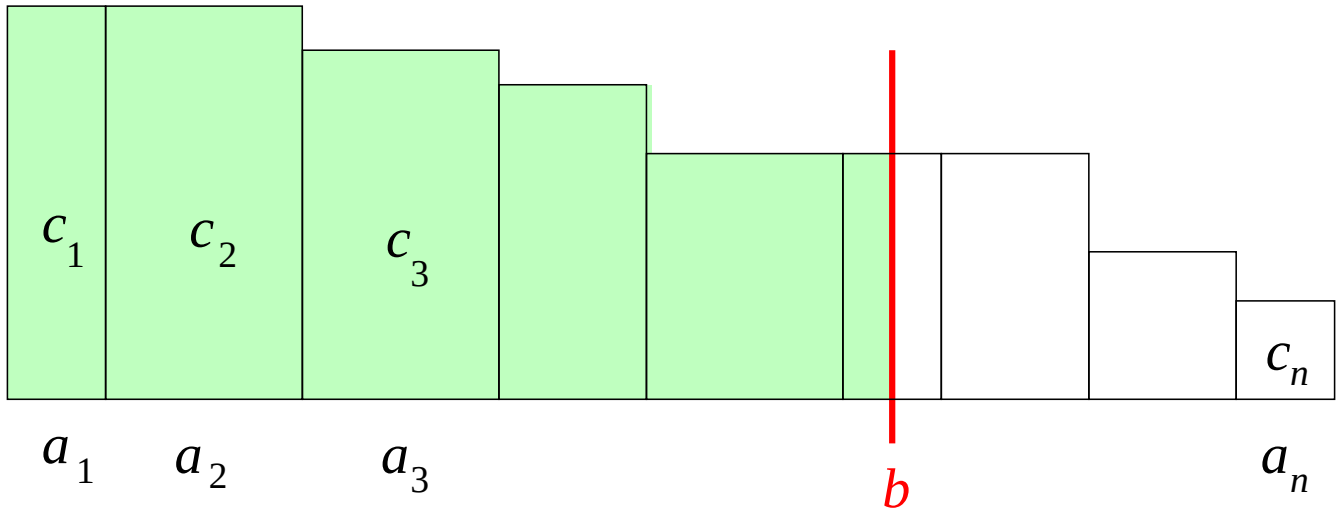
Höhe von Kasten Nummer i ist $\boldsymbol{d_i} = c_i/a_i$.

Breite ist $\boldsymbol{a_i}$.

Fläche ist $\boldsymbol{c_i}$.

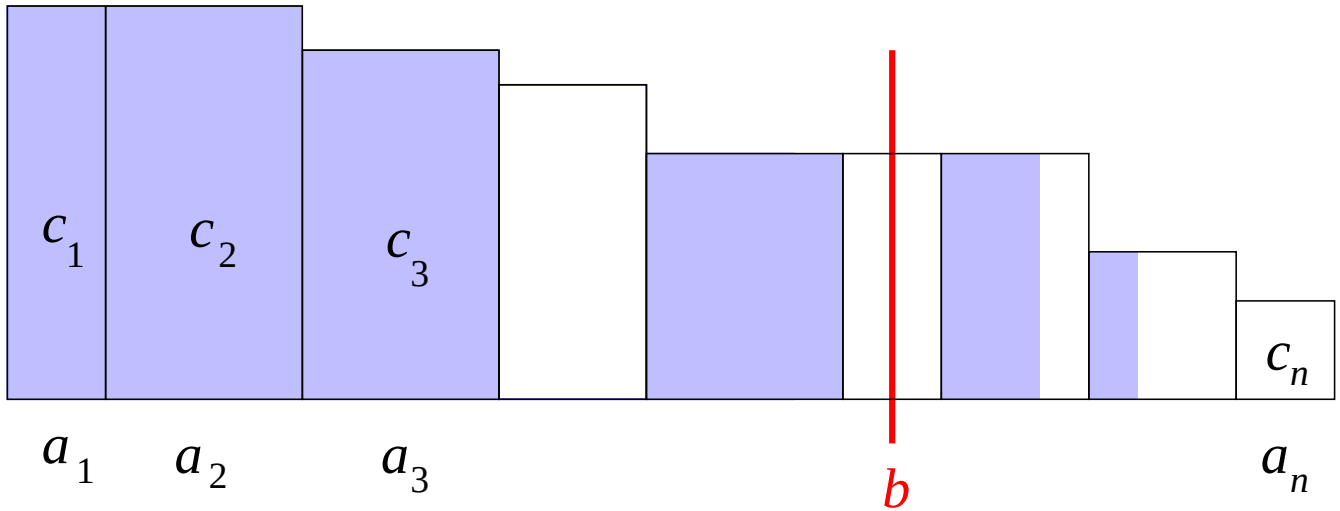
Algorithmus Greedy Fractional Knapsack (GFKS)

(1) **for** $\mathtt{i}$ **from** 1 **to** n **do**
(2) $\quad d_{\mathtt{i}} \leftarrow c_{\mathtt{i}}/a_{\mathtt{i}}$;
(3) $\quad \lambda_{\mathtt{i}} \leftarrow 0$;
(4) Sortiere Objekte gemäß d_i fallend;
(5) $\mathtt{i} \leftarrow 0$;
(6) $\mathtt{r} \leftarrow b$; // Inhalt r ist das verfügbare Rest-Volumen
(7) **while** $\mathtt{r} > 0$ **do**
(8) $\quad \mathtt{i}$++;
(9) $\quad$ **if** $a_{\mathtt{i}} \leq \mathtt{r}$
(10) $\quad\quad$ **then** $\lambda_{\mathtt{i}} \leftarrow 1$; $\mathtt{r} \leftarrow \mathtt{r} - a_{\mathtt{i}}$;
(11) $\quad\quad$ **else** $\lambda_{\mathtt{i}} \leftarrow \mathtt{r}/a_{\mathtt{i}}$;
(12) $\quad\quad\quad\quad \mathtt{r} \leftarrow 0$;

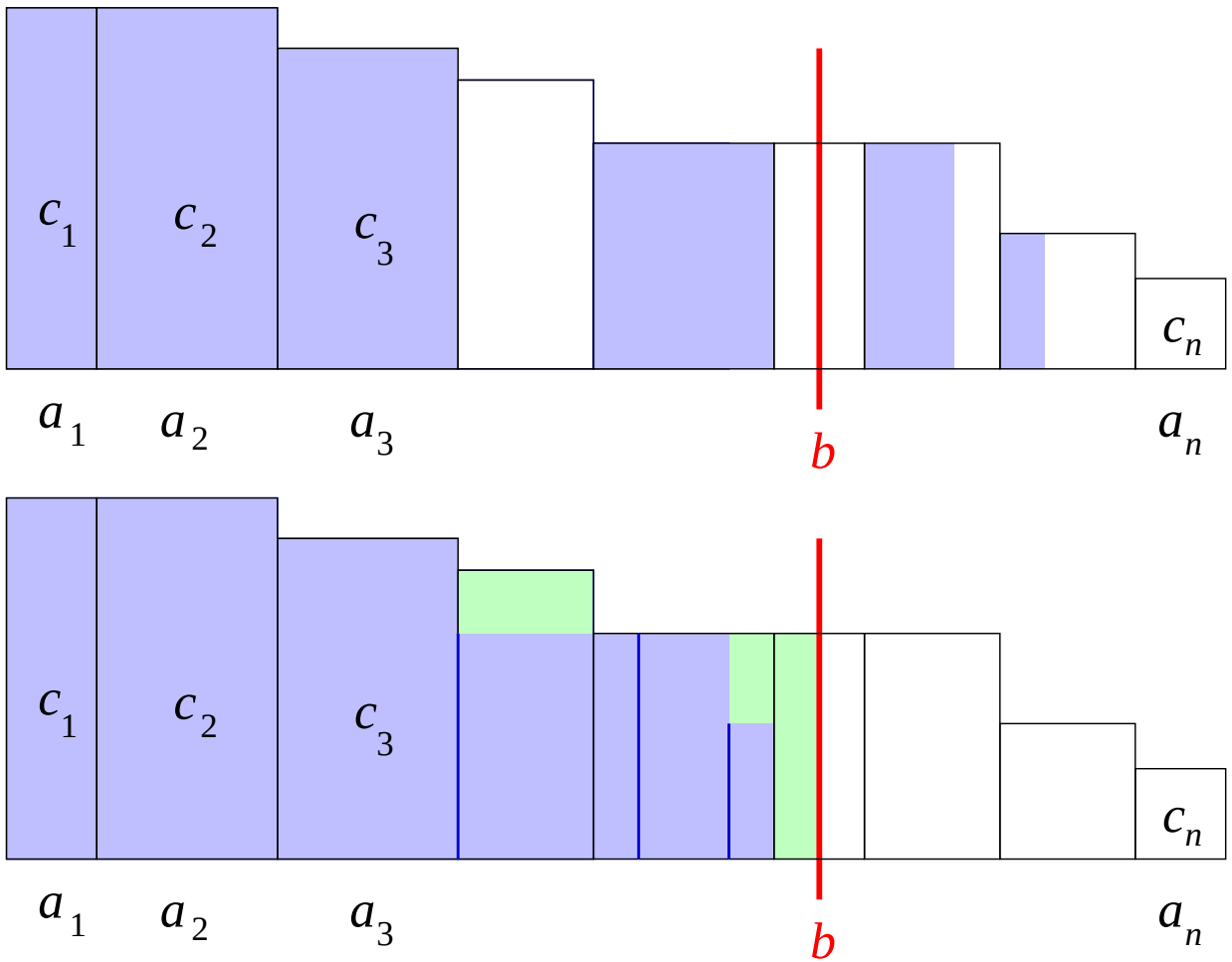


Vom Greedy-Algorithmus gelieferte Lösung.

Gesamtnutzen: grün.



Zulässige Lösung, nicht optimal. Gesamtnutzen: blau.



Anschaulich: **Greedy-Lösung** ist nie schlechter als **beliebige Lösung**.

Satz 10.1.2

Der Algorithmus GFKS ist korrekt (liefert eine zulässige Lösung mit maximalem Gesamtnutzen) und hat Rechenzeit $O(n \log n)$.

Für Interessierte: Vollständiger Beweis, der die anschauliche Argumentation in Formeln übersetzt, steht auf den Druckfolien.

Beweis: Sei $x = (a_1, \ldots, a_n, c_1, \ldots, c_n, b)$ die Eingabe.

Rechenzeit: klar.

Korrektheit: Dass die Lösung zulässig ist, ist klar. **Zu zeigen:** Optimalität.

O.B.d.A.: $\sum_i a_i > b$ (sonst ist $(\lambda_1, \ldots, \lambda_n) = (1, \ldots, 1)$ optimal und wird von GFKS gefunden).

Sei $(\lambda_1, \ldots, \lambda_n) \in [0, 1]^n$ die **Ausgabe** des Algorithmus.

Sei $(\lambda_1^*, \ldots, \lambda_n^*) \in [0, 1]^n$ eine **optimale** Lösung.

Offensichtlich: $\sum_i \lambda_i^* a_i = b$ (sonst könnten wir die Lösung durch Hinzufügen eines Bruchteil eines nicht ganz aufgebrauchten Objekts verbessern).

Durch Induktion über n zeigen wir: $\sum_i \lambda_i c_i = \sum_i \lambda_i^* c_i$.

I.A.: $n = 1$. Dann liefert der Algorithmus offensichtlich die optimale Lösung: Packe genau den Bruchteil $\lambda_1 = b/a_1$, der in den Rucksack passt.

Ind.-Schritt: Sei nun $n > 1$.

1. Fall: $a_1 > b$.

GFKS wählt $\lambda_1 = b/a_1$, und das ist optimal: Weil $d_1 \geq d_i$ für alle i, gilt:

$$\lambda_1 c_1 = b d_1 = \Big(\sum_i \lambda_i^* a_i\Big) d_1 \geq \sum_i \lambda_i^* a_i d_i = \sum_i \lambda_i^* c_i.$$

2. Fall: $a_1 \leq b$.

Behauptung: Wir können o.B.d.A. $\lambda_1^* = 1$ annehmen.
(„Der erste Schritt des Greedy-Algorithmus ist nicht falsch.“)
Denn: Wenn $\lambda_1^* < 1$, kann man wegen $\sum_{1 \leq i \leq n} \lambda_i^* a_i = b$ Werte $\lambda_1' = 1$, $0 \leq \lambda_2' \leq \lambda_2^*, \ldots, 0 \leq \lambda_n' \leq \lambda_n^*$ finden, so dass

$$(1 - \lambda_1^*) a_1 = \sum_{2 \leq i \leq n} (\lambda_i^* - \lambda_i') a_i. \qquad (*)$$

(Verringere Gewicht bei „späteren“ Objekten zugunsten von Objekt 1.)
Dann ist auch $(\lambda_1', \ldots, \lambda_n')$ zulässig und

$$\sum_i \lambda_i' c_i = \sum_i \lambda_i^* c_i + \Big((1 - \lambda_1^*) a_1 d_1 - \sum_{2 \leq i \leq n} (\lambda_i^* - \lambda_i') a_i d_i \Big).$$

Aus $(*)$ und $d_1 \geq d_i$ folgt, dass die letzte Klammer nichtnegativ ist, also $\sum_i \lambda_i' c_i \geq \sum_i \lambda_i^* c_i$ gilt, dass also auch $(\lambda_1', \ldots, \lambda_n')$ optimal ist.

Wir wissen nun: $1 = \lambda_1 = \lambda_1^*$.

Wir wenden die Induktionsvoraussetzung auf den modifizierten Input $x^- = (a_2, \ldots, a_n, c_2, \ldots, c_n, b - a_1)$ an: GFKS auf x^- liefert optimale Lösung $(\lambda_2, \ldots, \lambda_n)$. GFKS läuft in Schleifendurchläufen 2 bis n ebenso wie GFKS auf x^-, liefert also ebenfalls $(\lambda_2, \ldots, \lambda_n)$.

Klar: $(\lambda_2^*, \ldots, \lambda_n^*)$ muss für x^- optimal sein.

(Wenn $(\lambda_2', \ldots, \lambda_n')$ besser wäre, dann wäre $(1, \lambda_2', \ldots, \lambda_n')$ eine bessere Lösung für x als $(1, \lambda_2^*, \ldots, \lambda_n^*)$. Das kann aber nicht sein.)

Also gilt $\sum_{2 \leq i \leq n} \lambda_i c_i = \sum_{2 \leq i \leq n} \lambda_i^* c_i$.

Also haben auch die Lösungen $(1, \lambda_2^*, \ldots, \lambda_n^*)$ und $(1, \lambda_2, \ldots, \lambda_n)$ für x denselben Nutzenwert, und $(\lambda_1, \lambda_2, \ldots, \lambda_n) = (1, \lambda_2, \ldots, \lambda_n)$ ist optimal. □

Charakteristika der Greedy-Methode:

1. Der **erste Schritt** der Greedy-Lösung ist **nicht falsch**. Es gibt eine optimale Lösung, die als **Fortsetzung des ersten Schrittes** konstruiert werden kann.
2. **„Prinzip der optimalen Substruktur“:** Kombiniert man das Ergebnis des ersten „Greedy-Schritts“ mit einer beliebigen optimalen Lösung für die reduzierte Instanz (Wegnehmen des Teils, der durch den ersten Schritt erledigt ist), dann ergibt sich eine optimale Lösung für die Originaleingabe.
3. Der Algorithmus löst das verbleibende Teilproblem rekursiv (oder mit einem zur Rekursion äquivalenten iterativen Verfahren).

Die Korrektheit ergibt sich dann durch vollständige Induktion.

10.2 Huffman-Codes

Gegeben: **Alphabet A** mit $2 \leq |A| < \infty$ und „Wahrscheinlichkeiten“ oder „relativen Häufigkeiten“ $p(a) \in [0,1]$ für jedes $a \in A$.

Also: $\sum_{a \in A} p(a) = 1$.

Beispiel: $A = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}, \mathtt{F}, \mathtt{G}, \mathtt{H}, \mathtt{I}, \mathtt{K}\}$, $|A| = 10$:

a	A	B	C	D	E	F	G	H	I	K
$p(a)$	0,15	0,08	0,07	0,10	0,21	0,08	0,07	0,09	0,06	0,09

Herkunft der Wahrscheinlichkeiten:
(1) Buchstabenhäufigkeit in natürlicher Sprache *oder*
(2) empirische relative Häufigkeiten in einem gegebenen Text $w = a_1 \dots a_m \in A^*$:
Anteil des Buchstabens a an w ist $(p(a) \cdot 100)\%$.

Gesucht: ein **„guter“ binärer Präfixcode** für (A, p). — *Beispiel*:

A	B	C	D	E	F	G	H	I	K
1100	0110	000	111	10	0011	010	0010	0111	1101

Definition 10.2.1 Präfixcode:

Jedem $a \in A$ ist binärer „Code“ $c(a) \in \{0,1\}^*$ (Menge aller Binärstrings) zugeordnet, mit Eigenschaft

Präfixfreiheit: Für $a, b \in A$, $a \neq b$ ist $c(a)$ kein Präfix von $c(b)$.

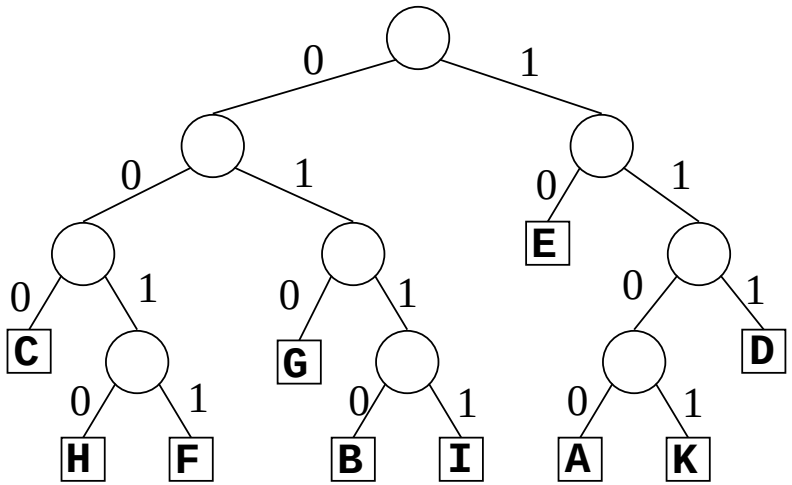
Codierung von Wörtern (Zeichenreihen):

$c(a_1 \dots a_n) = c(a_1) \cdots c(a_n) \in \{0,1\}^*$.

Zur **Codierung** benutzt man (konzeptuell) direkt die Tabelle.

Beispiel: $c(\texttt{F E I G E}) = 001110011101010$.

Kompakte Repräsentation des Codes als Binärbaum:



Blätter sind mit Buchstaben markiert; Weg von der Wurzel zum Blatt gibt das Codewort wieder (links: 0, rechts: 1).

Decodierung: Laufe Weg im Baum, vom Codewort gesteuert, bis zum Blatt. Wiederhole mit dem Restwort, bis nichts mehr übrig ist.

Präfixeigenschaft $\Rightarrow$ keine Zwischenräume nötig.

Beispiel: 001111000000010 liefert „`FACH`“.

1. Idee: Mache alle Codewörter $c(a)$ gleich lang.
Am besten ist dann eine Länge von $\lceil \log_2 |A| \rceil$ Bits.

$\Rightarrow c(a_1 \dots a_m)$ hat Länge $\lceil \log_2 |A| \rceil \cdot m$.

(Beispiele: 52 Groß- und Kleinbuchstaben plus Leerzeichen und Satzzeichen:
$\log 64 = 6$ Bits pro Codewort. Latin-1-Code: 8 Bits pro Codewort.)

2. Idee: Einsparmöglichkeit: Häufige Buchstaben mit kürzeren Wörtern codieren als seltenere Buchstaben.

Ein erster Ansatz zur **Datenkompression** (platzsparendes Speichern, zeitsparendes Übermitteln)!

Hier: **„verlustfreie Kompression“** – Information ist unverändert vorhanden.

Gegensatz: z. B. MP3: Informationsverlust bei der Kompression.

a	A	B	C	D	E	F	G	H	I	K
$p(a)$	0,15	0,08	0,07	0,10	0,21	0,08	0,07	0,09	0,06	0,09
$c_1(a)$	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
$c_2(a)$	1100	0110	000	111	10	0011	010	0010	0111	1101

Wir codieren eine Datei T mit 100000 Buchstaben aus A, wobei die relative Häufigkeit von $a \in A$ durch $p(a)$ gegeben ist.

Mit c_1 (fixe Codewortlänge): **400000** Bits.

Mit c_2 (variable Codewortlänge):

$$\big(4 \cdot (0,15 + 0{,}08 + 0{,}08 + 0{,}09 + 0{,}06 + 0{,}09) + 3 \cdot (0{,}07 + 0{,}10 + 0{,}07) + 2 \cdot 0{,}21\big) \cdot 100000 = \mathbf{334000} \text{ Bits.}$$

Das ist eine Ersparnis von 16%. Bei langen Dateien und wenn die Übertragung teuer oder langsam ist, lohnt es sich, die Buchstaben abzuzählen, die relativen Häufigkeiten $p(a)$ zu bestimmen und einen guten Code mit unterschiedlichen Codewortlängen zu suchen.

Definition 10.2.2

Ein **Codierungsbaum** für A ist ein Binärbaum T, in dem

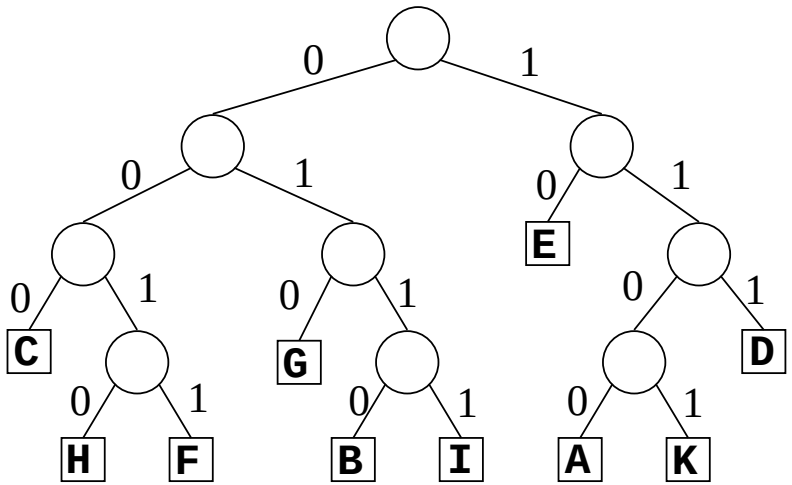
- die Kante in einem inneren Knoten zum linken bzw. rechten Kind (implizit) mit 0 bzw. 1 markiert ist;
- jedem Buchstaben $a \in A$ ein Blatt (externer Knoten) von T exklusiv zugeordnet ist.

Der Code $c_T(a)$ für a ist die Kanteninschrift auf dem Weg von der Wurzel zum Blatt mit Inschrift a.

Die **Kosten** von T unter p sind definiert als:

$$\boldsymbol{B(T,p)} = \sum_{a \in A} p(a) \cdot d_T(a),$$

wobei $d_T(a) = |c_T(a)|$ die Tiefe des a-Blatts in T ist.



Beispiel: Wenn T unser Beispielbaum ist und p die Beispielverteilung von oben, dann ist $B(T,p) = 3{,}34$ ($\ldots = 334\,000/100\,000$, s. Folie 28).

Leicht zu sehen:
$B(T,p) = |c_T(a_1 \ldots a_m)|/m$, wenn $p(a)$ die relative Häufigkeit von a in $w = a_1 \ldots a_m$ ist, **oder**
$B(T,p) =$ die **erwartete** **Bitzahl** pro Buchstabe, wenn $p(a)$ die **Wahrscheinlichkeit** von a ist.

Definition 10.2.3

Ein Codierungsbaum T für A heißt **optimal** oder **redundanzminimal** für p, wenn für alle Codierungsbäume T' für A gilt: $B(T,p) \leq B(T',p)$.

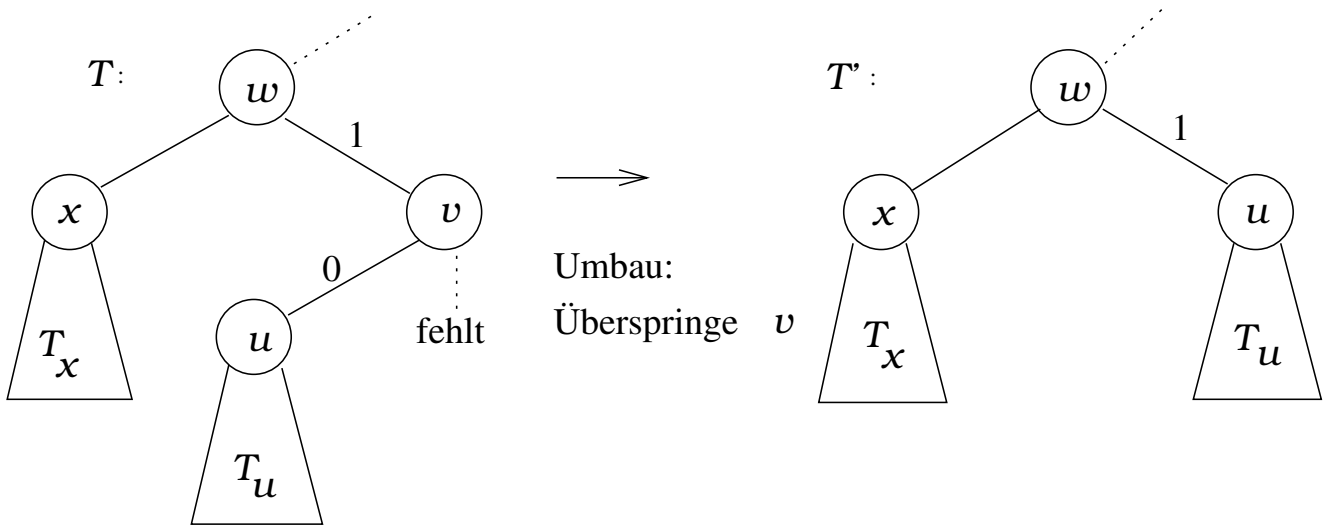
Aufgabe: Zu gegebenem $p\colon A \to [0,1]$ finde einen optimalen Baum T.

Existiert immer ein optimaler Baum? (Zu A gibt es unendlich viele Codierungsbäume!)

Lemma 10.2.4

Wenn T Codierungsbaum und p Verteilung für A ist, dann gibt es einen Codierungsbaum T' mit $B(T', p) \leq B(T, p)$, so dass in T' jeder innere Knoten zwei Kinder hat.

Beweisidee:



Resultat: T' für A mit denselben markierten Blättern wie T, und $B(T', p) \leq B(T, p)$.

Folgerung: Man kann sich bei der Suche nach optimalen Bäumen auf solche beschränken, in denen jeder innere Knoten zwei Kinder hat.

Weil es für festes A nur endlich viele Codierungsbäume gibt, in denen jeder innere Knoten zwei Kinder hat, gibt es für (A, p) optimale Codierungsbäume.

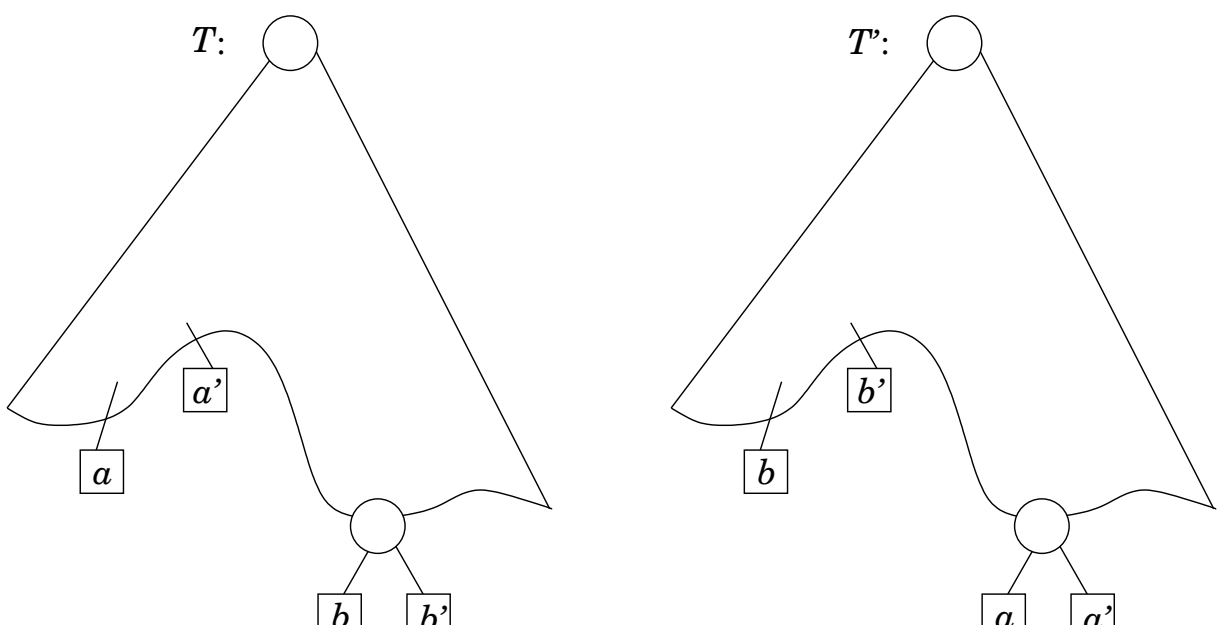
Optimale Bäume sind i. A. nicht eindeutig bestimmt.
(Man kann Knoten auf einem Level beliebig vertauschen.)

Aufgabe: Gegeben (A, p), finde einen optimalen Baum. **Methode: „Greedy“.**

Lemma 10.2.5

Es seien a, a' zwei Buchstaben mit $p(a) \leq p(a') \leq p(b)$ für alle $b \in A - \{a, a'\}$.
(a, a' sind zwei **„seltenste“** Buchstaben.)
Dann gibt es einen optimalen Baum, in dem die Blätter für a und a' „Geschwister“ sind, also Kinder desselben inneren Knotens.

Beweis:



Starte mit beliebigem optimalen Baum T.

O.B.d.A. (Lemma 10.2.4): Alle inneren Knoten haben zwei Kinder.

Es seien $d(a)$ und $d(a')$ die Tiefen der Knoten für a und a'.

Suche zwei Geschwisterknoten mit maximaler Tiefe $d = d(T)$, etwa mit Buchstaben b und b', wobei $p(b) \leq p(b')$.

Vertausche a mit b und a' mit b'!

Dies liefert Baum T', in dem a und a' in Geschwisterknoten sitzen.

Es gilt $B(T, p) \geq B(T', p)$, weil $p(a), p(a') \leq p(b) = p(b')$.

Also ist auch T' optimal für (A, p). □

Damit ist der erste Schritt zur Realisierung eines Greedy-Ansatzes getan!

Man beginnt den Algorithmus mit

„Mache die beiden seltensten Buchstaben zu Geschwistern.“

Dann ist sicher, dass dies stets zu einer optimalen Lösung ausgebaut werden kann.

Dann werden diese beiden Buchstaben „zusammengeklebt“, so dass man ein Alphabet erhält, das eine um 1 kleinere Größe hat.

Konzeptuell wendet man dann **Rekursion** an.

In der Realisierung wird der Algorithmus **iterativ** programmiert.

Algorithmus wurde 1951 von D. A. Huffman (1925–1999), gefunden.

Nette Geschichte: `http://www.huffmancoding.com/my-uncle/scientific-american`

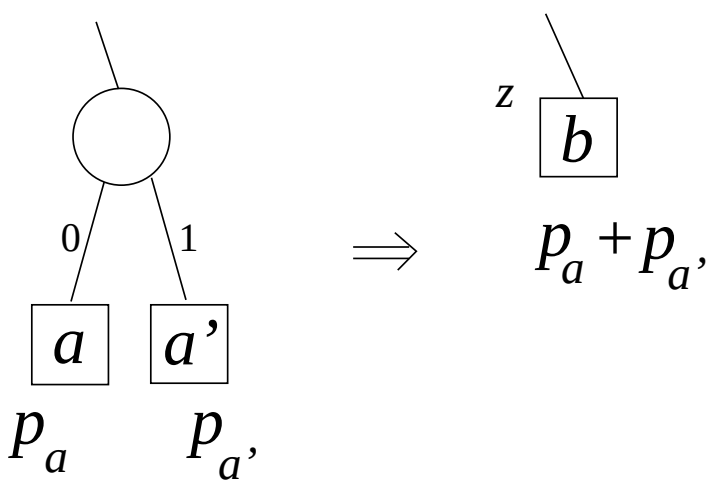
Huffman-Algorithmus (rekursiv):

Wir bauen **„bottom-up“** einen Baum auf.

Wenn $|A| = 1$: Baum hat nur einen (externen) Knoten.
(Code für den einen Buchstaben ist das leere Wort. Seltsam, aber als Rekursionsbasis geeignet.)

Die Kosten sind 0. (Optimalität ist klar.)

Wenn $|A| > 1$, werden zwei „seltenste“ Buchstaben a, a' aus A zu benachbarten Blättern gemacht.

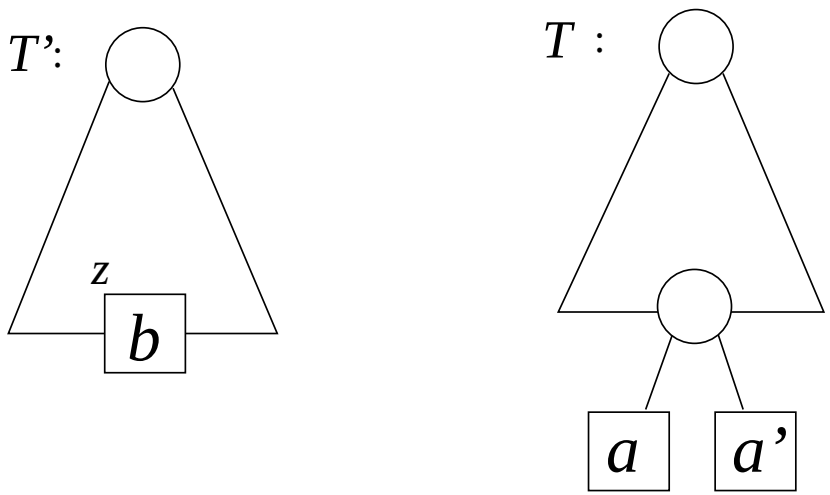


Die Wurzel des so erzeugten Mini-Baums wird als ein neuer Knoten z mit „Kunstbuchstaben“ b aufgefasst, mit Wahrscheinlichkeit $p(b) := p(a) + p(a')$.

Neues Alphabet: $A' := (A - \{a, a'\}) \cup \{b\}$; neue Verteilung:

$$p'(d) := \begin{cases} p(d), & \text{falls } d \neq b, \\ p(a) + p(a'), & \text{falls } d = b. \end{cases}$$

Wir rufen den Algorithmus **rekursiv** auf, um einen Baum T' für A' und p' zu erhalten.



In T' fügen wir an der Stelle des b-Knotens z den a, a'-Baum ein. Dies liefert einen **Codierungsbaum** T für A, mit

$$B(T,p) = B(T',p') + p(a) + p(a').$$

(Checken!)

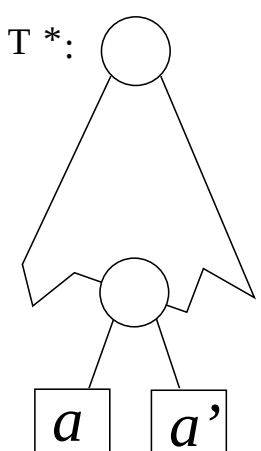
Lemma 10.2.6 T ist **optimaler** Baum für (A, p).

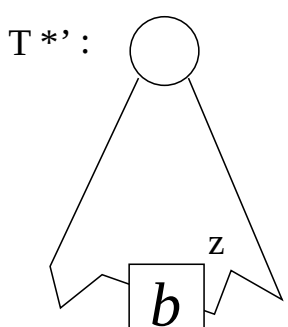
Beweis: Durch Induktion über die rekursiven Aufrufe. Der I.A. ($|A| = 1$) ist klar.

Sei nun $|A| \geq 2$, und seien a und a' seltenste Buchstaben in A mit p.

Nach Lemma 10.2.5. gibt es einen **optimalen** Baum T^* für (A, p), in dem die Knoten für a und a' Geschwister sind.

Aus T^* bilden wir $T^{*\prime}$ durch Ersetzen des a, a'-Teilbaums durch den neuen Knoten z mit dem Kunstbuchstaben b. Dann ist $T^{*\prime}$ **Codierungs**baum für A'.





Wir haben

$$B(T^*, p) = B(T^{*\prime}, p') + p(a) + p(a').$$

Nach I.V. für den rekursiven Aufruf ist T' optimal für (A', p'), also

$$B(T', p') \leq B(T^{*\prime}, p').$$

Daher:

$$\begin{aligned} B(T, p) &= B(T', p') + p(a) + p(a') \\ &\leq B(T^{*\prime}, p') + p(a) + p(a') \\ &= B(T^*, p). \end{aligned}$$

Weil T^* optimaler Baum für (A, p) ist, folgt $B(T, p) = B(T^*, p)$, und auch T ist optimal. □

Man *könnte* nach dem angegebenen Muster eine rekursive Prozedur programmieren.

PQ: Datenstruktur Prioritätswarteschlange,
Einträge: Buchstaben und Kunstbuchstaben;
Schlüssel: die Gewichte $p(b)$, b (Kunst-)Buchstabe.

Operationen: **PQ.insert**: Einfügen eines neuen Eintrags;
PQ.extractMin: Entnehmen des Eintrags mit kleinstem Schlüssel.

Beide Operationen benötigen **logarithmische Zeit** (s. Kap. 6).

Anfangs in **PQ**: Buchstaben $a \in A$ mit Gewichten $p(a)$ als Schlüssel.

Ermitteln und Entfernen der beiden „seltensten“ Buchstaben a, a' durch zwei Aufrufe **PQ.extractMin**;

Einfügen des neuen Kunstbuchstabens b durch **PQ.insert**(b).

Resultierende Rechenzeit: $O(n \log n)$, für $|A| = n$.

Iterative Implementierung mit ein paar Tricks wird **viel effizienter**.

Sei $A = \{a_1, \ldots, a_n\}$. Namen der Buchstaben spielen gar keine Rolle! O.B.d.A.: $1, \ldots, n$.

Wahrscheinlichkeiten: $p_1, \ldots, p_n$.

- Sortieren am Anfang sorgt dafür, dass $p(a_1) \leq \cdots \leq p(a_n)$ (benötigt Zeit $O(n \log n)$).
- Nummern $1, \ldots, n$ für Blätter (Buchstaben aus A),
  Nummern $n+1, \ldots, 2n-1$ für innere Knoten (Kunstbuchstaben).
- Spezielle Darstellung des Codierungsbaums (nur Indizes anstelle von Vorgängerzeigern).
- Beobachtung: Neu erzeugte Häufigkeiten $p(b)$ sind (schwach) monoton wachsend.

Datenstrukturen: Drei Arrays

$\texttt{p}[1..2n-1]$:	Wahrscheinlichkeiten der (Kunst-)Buchstaben,
$\texttt{pred}[1..2n-2]$:	Vorgängerknoten, $2n-1$ ist die Wurzel,
$\texttt{mark}[1..2n-2]$:	Eingangskante mit 0 oder mit 1 markiert?

Algorithmus Huffman(p[1..n])

Input: Gewichts-/Wahrscheinlichkeitsvektor p[1..n], sortiert: p[1] $\leq \cdots \leq$ p[n].

Output: Optimaler Baum T, dargestellt als pred[1..$2n-2$] und mark[1..$2n-2$]

(0) Erweitere den Vektor p[1..n] auf Länge $2n-1$;
(1) pred[1] $\leftarrow n+1$; pred[2] $\leftarrow n+1$;
(2) mark[1] $\leftarrow 0$; mark[2] $\leftarrow 1$;
(3) p[$n+1$] $\leftarrow$ p[1] + p[2];
(4) k $\leftarrow 3$ // erster nicht verarbeiteter Knoten in [1..n]
(5) h $\leftarrow n+1$ // erster nicht verarbeiteter Knoten in [$n+1$..$2n-2$]
(6) **for** b **from** $n+2$ **to** $2n-1$ **do**
(7) // Die folgenden beiden Zeilen finden, in i und j, die Positionen der
(8) // beiden noch nicht verarbeiteten Buchstaben mit kleinsten Gewichten
(9) **if** k $\leq n$ and p[k] $\leq$ p[h] **then** i $\leftarrow$ k; k++ **else** i $\leftarrow$ h; h++;
(10) **if** k $\leq n$ and (h = b or p[k] $\leq$ p[h]) **then** j $\leftarrow$ k; k++ **else** j $\leftarrow$ h; h++;
(11) pred[i] $\leftarrow$ b; pred[j] $\leftarrow$ b;
(12) mark[i] $\leftarrow 0$; mark[j] $\leftarrow 1$;
(13) p[b] $\leftarrow$ p[i] + p[j];
(14) **Ausgabe:** pred[1..$2n-2$] und mark[1..$2n-2$].

Aus pred$[1..2n-2]$ und mark$[1..2n-2]$ baut man den optimalen Huffman-Baum wie folgt:

Alloziere ein Array leaf$[1..n]$ mit Blattknoten-Objekten
und ein Array inner$[n+1..2n-1]$ mit Objekten für innere Knoten.

```
(1)   for i from 1 to n do
(2)     leaf[i].letter ← Buchstabe a_i.
(3)     if mark[i] = 0
(4)       then inner[pred[i]].left ← leaf[i]
(5)       else inner[pred[i]].right ← leaf[i]
(6)   for i from n + 1 to 2n − 2 do
(7)     if mark[i] = 0
(8)       then inner[pred[i]].left ← inner[i]
(9)       else inner[pred[i]].right ← inner[i]
(10)  return inner[2n − 1] // Wurzelknoten
```

Achtung: Wenn man den Algorithmus von Hand ausführt, ist es viel einfacher, den Baum von der Wurzel beginnend zu zeichnen.

a_i	A	B	C	D	E	F	G	H	I	K
i	1	2	3	4	5	6	7	8	9	10
p_i	0,15	0,08	0,07	0,10	0,21	0,08	0,07	0,09	0,06	0,09
pred										
mark										

Wir sortieren und nummerieren um.

(I.A.: $O(n \log n)$ Rechenzeit.)

Dann: Verlängerung der Arrays.

a_i	I	G	C	F	B	K	H	D	A	E
i	1	2	3	4	5	6	7	8	9	10
p_i	0,06	0,07	0,07	0,08	0,08	0,09	0,09	0,10	0,15	0,21
pred										
mark										

i	11	12	13	14	15	16	17	18	19
p_i									
pred									—
mark									—

a_i	I	G	C	F	B	K	H	D	A	E
i	1	2	3	4	5	6	7	8	9	10
p_i	0,06	0,07	0,07	0,08	0,08	0,09	0,09	0,10	0,15	0,21
pred	**11**	**11**								
mark	**0**	**1**								

i	11	12	13	14	15	16	17	18	19
p_i	**0,13**								
pred									—
mark									—

a_i	I	G	C	F	B	K	H	D	A	E
i	1	2	3	4	5	6	7	8	9	10
p_i	0,06	0,07	0,07	0,08	0,08	0,09	0,09	0,10	0,15	0,21
pred	11	11	12	12	13	13	**14**	**14**		
mark	0	1	0	1	0	1	**0**	**1**		

i	11	12	13	14	15	16	17	18	19
p_i	0,13	0,15	0,17	**0,19**					
pred									—
mark									—

a_i	I	G	C	F	B	K	H	D	A	E
i	1	2	3	4	5	6	7	8	9	10
p_i	0,06	0,07	0,07	0,08	0,08	0,09	0,09	0,10	0,15	0,21
pred	11	11	12	12	13	13	14	14	**15**	
mark	0	1	0	1	0	1	0	1	**1**	

i	11	12	13	14	15	16	17	18	19
p_i	0,13	0,15	0,17	0,19	**0,28**				
pred	**15**								—
mark	**0**								—

a_i	I	G	C	F	B	K	H	D	A	E
i	1	2	3	4	5	6	7	8	9	10
p_i	0,06	0,07	0,07	0,08	0,08	0,09	0,09	0,10	0,15	0,21
pred	11	11	12	12	13	13	14	14	15	
mark	0	1	0	1	0	1	0	1	1	

i	11	12	13	14	15	16	17	18	19
p_i	0,13	0,15	0,17	0,19	0,28	**0,32**			
pred	15	**16**	**16**						—
mark	0	**0**	**1**						—

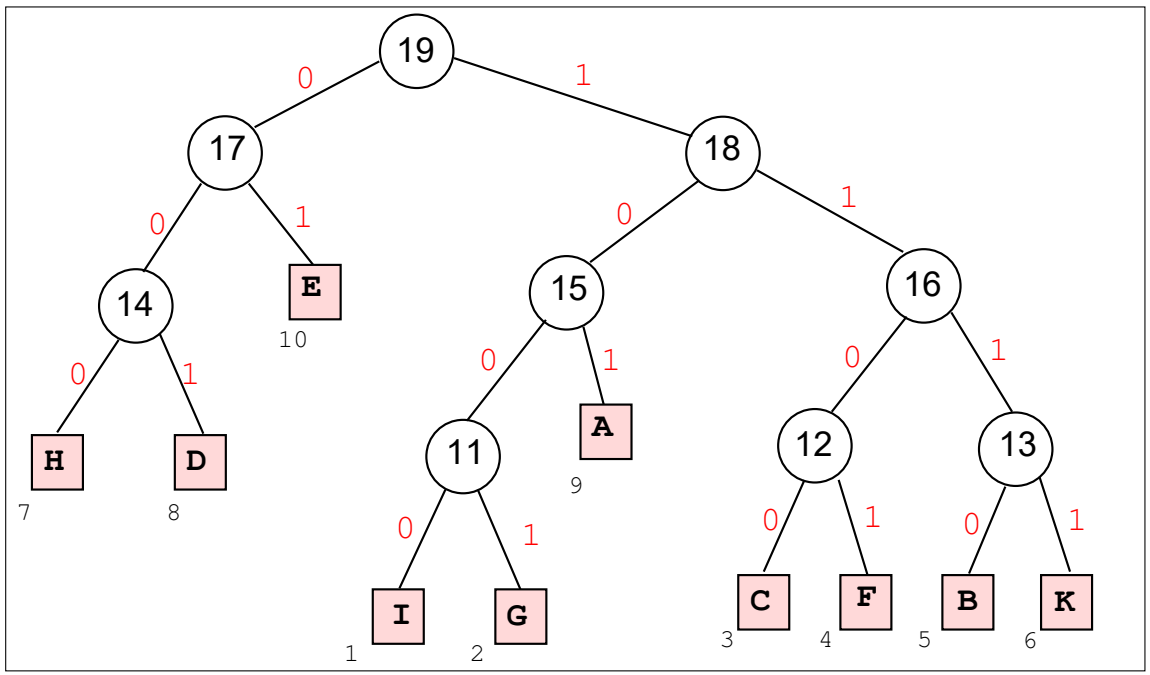
a_i	I	G	C	F	B	K	H	D	A	E
i	1	2	3	4	5	6	7	8	9	10
p_i	0,06	0,07	0,07	0,08	0,08	0,09	0,09	0,10	0,15	0,21
pred	11	11	12	12	13	13	14	14	15	17
mark	0	1	0	1	0	1	0	1	1	1

i	11	12	13	14	15	16	17	18	19
p_i	0,13	0,15	0,17	0,19	0,28	0,32	0,40	0,60	1,00
pred	15	16	16	17	18	18	19	19	—
mark	0	0	1	0	0	1	0	1	—

a_i	I	G	C	F	B	K	H	D	A	E
i	1	2	3	4	5	6	7	8	9	10
pred	11	11	12	12	13	13	14	14	15	17
mark	0	1	0	1	0	1	0	1	1	1

i	11	12	13	14	15	16	17	18	19
pred	15	16	16	17	18	18	19	19	—
mark	0	0	1	0	0	1	0	1	—

Resultierender optimaler Codierungsbaum T, Kodierungsfunktion c_3:



	A	B	C	D	E	F	G	H	I	K
$p(a)$	0,15	0,08	0,07	0,10	0,21	0,08	0,07	0,09	0,06	0,09
c_3	101	1110	1100	001	01	1101	1001	000	1000	1111

$$B(T,p) = 0{,}21 \cdot 2 + (0{,}1 + 0{,}09 + 0{,}15) \cdot 3 + 0{,}45 \cdot 4 = \mathbf{3{,}24}.$$

Satz 10.2.7

Der Algorithmus **Huffman** ist korrekt und hat Laufzeit $O(n \log n)$, wenn n die Anzahl der Buchstaben des Alphabets A bezeichnet.

Das Material in diesem Schlussabschnitt ist interessant, aber **nicht prüfungsrelevant**.

Kann man $B(T,p)$ für einen optimalen Baum einfach aus den Häufigkeiten $p(a_1),\ldots,p(a_n)$ **berechnen**, ohne T zu konstruieren?

Antwort: Ja, zumindest näherungsweise.

Definition 10.2.8

Für $p_1,\ldots,p_n \geq 0$ mit $\sum_{i=1}^{n} p_i = 1$ setze

$$\boldsymbol{H(p_1,\ldots,p_n)} := \sum_{i=1}^{n} p_i \cdot \log(1/p_i).$$

$H(p_1,\ldots,p_n)$ heißt die (binäre) **Entropie** der Verteilung $p_1,\ldots,p_n$.

Beispiele: $H(\frac{1}{2},\frac{1}{4},\frac{1}{4}) = \frac{1}{2}\cdot 1 + 2\cdot\frac{1}{4}\cdot 2 = \frac{3}{2}$, und $H(\frac{1}{8},\ldots,\frac{1}{8}) = 8\cdot\frac{1}{8}\cdot\log(1/\frac{1}{8}) = \log 8 = 3$.

Wenn $p_i = 0$ ist, setzt man $p_i \log(1/p_i) = 0$, was vernünftig ist, weil $\lim_{x\searrow 0} x\cdot\log(1/x) = 0$.

Unser nächstes Ziel: Zusammenhang zwischen **Entropie** $H(p_1, \ldots, p_n)$ und der **erwarteten Bitlänge** der Codierung eines Textes, in dem $n = |A|$ Buchstaben mit Wahrscheinlichkeiten $p_1, \ldots, p_n$ auftreten.

Klassisches Resultat: (Minimaleigenschaft der Entropie.)

Lemma 10.2.9 (Ungleichung von Gibbs)

Es seien $p_1, \ldots, p_n \geq 0$ und $q_1, \ldots, q_n > 0$ mit $\sum_{i=1}^{n} q_i \leq 1 = \sum_{i=1}^{n} p_i$. Dann gilt:

$$\sum_{i=1}^{n} p_i \log(1/q_i) \geq \sum_{i=1}^{n} p_i \log(1/p_i) = H(p_1, \ldots, p_n).$$

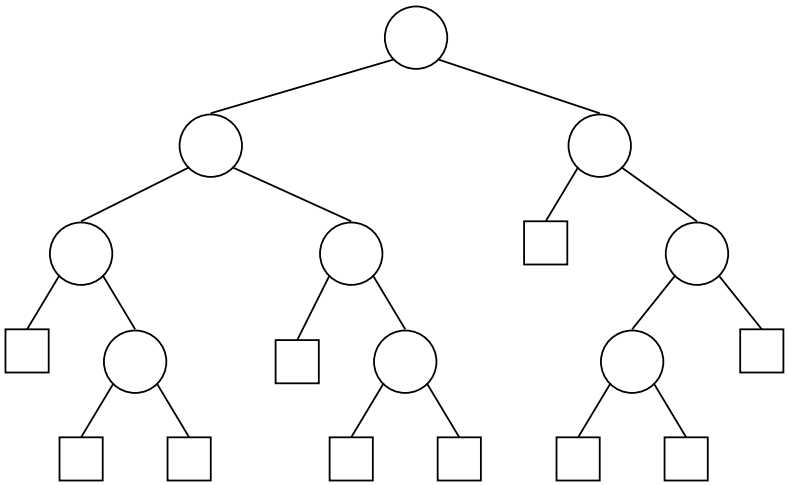
(Die Voraussetzung kann von $\forall i\colon\ q_i > 0$ zu $\forall i\colon\ p_i > 0 \Rightarrow q_i > 0$ abgeschwächt werden.)

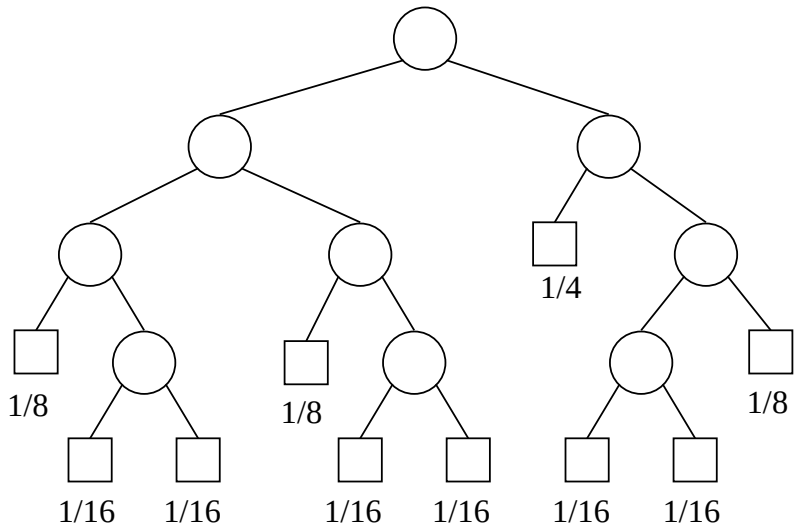
Beweis:

Weil $\log_2 x = (\ln x)/\ln 2$ ist, darf man mit dem natürlichen Logarithmus rechnen.
(Summanden für i mit $p_i = q_i = 0$ lässt man einfach weg.)

$$\begin{aligned}
&\sum_{i=1}^{n} p_i \ln\left(\frac{1}{p_i}\right) - \sum_{i=1}^{n} p_i \ln\left(\frac{1}{q_i}\right) \\
&\quad = \sum_{i=1}^{n} p_i \ln\left(\frac{q_i}{p_i}\right) \overset{(*)}{\leq} \sum_{i=1}^{n} p_i \left(\frac{q_i}{p_i} - 1\right) \\
&\quad = \sum_{i=1}^{n} (q_i - p_i) = \sum_{i=1}^{n} q_i - \sum_{i=1}^{n} p_i \\
&\quad \leq 0.
\end{aligned}$$

$(*)$: Es gilt $\ln x \leq x - 1$, für alle $x > 0$.





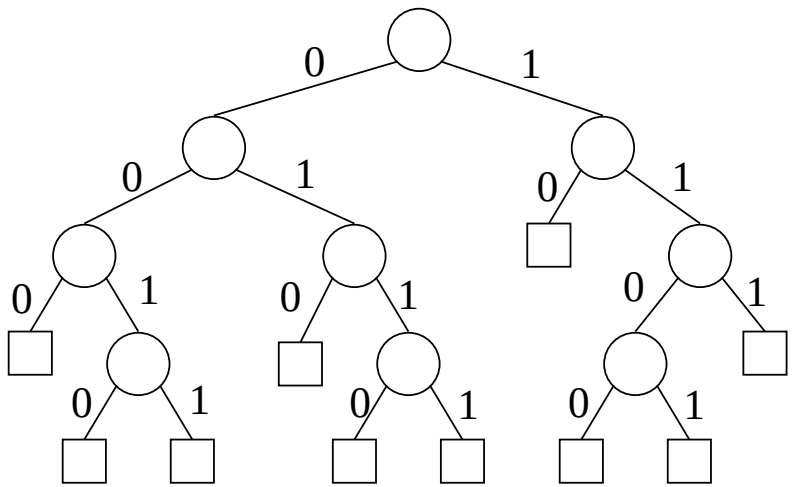
$$\frac{1}{4} + 3 \cdot \frac{1}{8} + 6 \cdot \frac{1}{16} = 1$$

Lemma 10.2.10

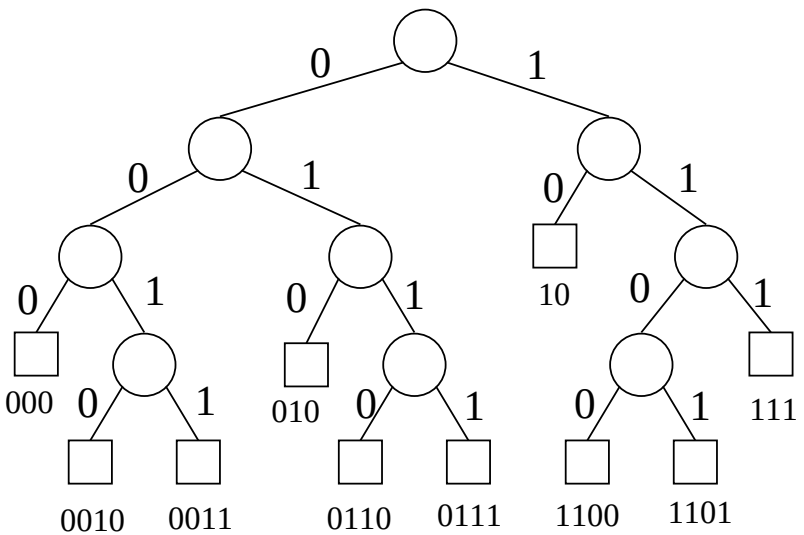
Es sei L die Menge der äußeren Knoten in einem Binärbaum T.

Dann gilt: $$\sum_{l \in L} 2^{-d(l)} = 1.$$

Beweis: Wie in einem Kodierungsbaum markiere in jedem inneren Knoten die Kante zum linken Kind mit 0, die zum rechten Kind mit 1.



Beweis: Wie in einem Kodierungsbaum markiere in jedem inneren Knoten die Kante zum linken Kind mit 0, die zum rechten Kind mit 1.



Jedem Blatt l entspricht dann eine Bitfolge $w_l = b_1 \cdots b_{d(l)}$.

Sei $D = d(T)$ die Tiefe des tiefsten Blatts.

Jedes w_l kann man auf $2^{D-d(l)}$ Arten zu einer Bitfolge der Länge D verlängern.

Alle so erzeugten Bitfolgen sind verschieden; alle 2^D Bitfolgen werden so erzeugt.

$$\Rightarrow \quad \sum_{l \in L} 2^{D-d(l)} = 2^D \quad \Rightarrow \quad \text{Behauptung.} \qquad \square$$

Satz 10.2.11 (Ungleichung von Kraft/McMillan)

Es seien $n \geq 1$, $\ell_1, \ldots, \ell_n \in \mathbb{N}$. Dann gilt:
Es gibt einen Präfixcode mit Codewortlängen $\ell_1, \ldots, \ell_n$ genau dann wenn

$$\sum_{i=1}^{n} 2^{-\ell_i} \leq 1.$$

Nach den Bemerkungen am Anfang von Abschnitt 10.2 kann man statt „Existenz eines Präfixcodes“ auch „Existenz eines Binärbaums mit verschiedenen Blättern auf Niveau $\ell_1, \ldots, \ell_n$ (und eventuell weiteren Blättern)“ sagen.

„$\Rightarrow$“: Sei ein Binärbaum mit n verschiedenen (externen) Blättern auf Tiefe $\ell_1, \ldots, \ell_n$ gegeben. (Eventuell gibt es weitere Blätter.) Aus Proposition 10.2.10 folgt $\sum_{i=1}^{n} 2^{-\ell_i} \leq 1$.

„$\Leftarrow$“: Nun seien $\ell_1, \ldots, \ell_n \geq 0$ gegeben, mit $\sum_{i=1}^{n} 2^{-\ell_i} \leq 1$.

O. B. d. A. gilt $\ell_1 \leq \cdots \leq \ell_n$ (sonst umordnen und umbenennen).

Wir bauen einen Binärbaum mit Blättern auf Niveaus $\ell_1, \ldots, \ell_n$.

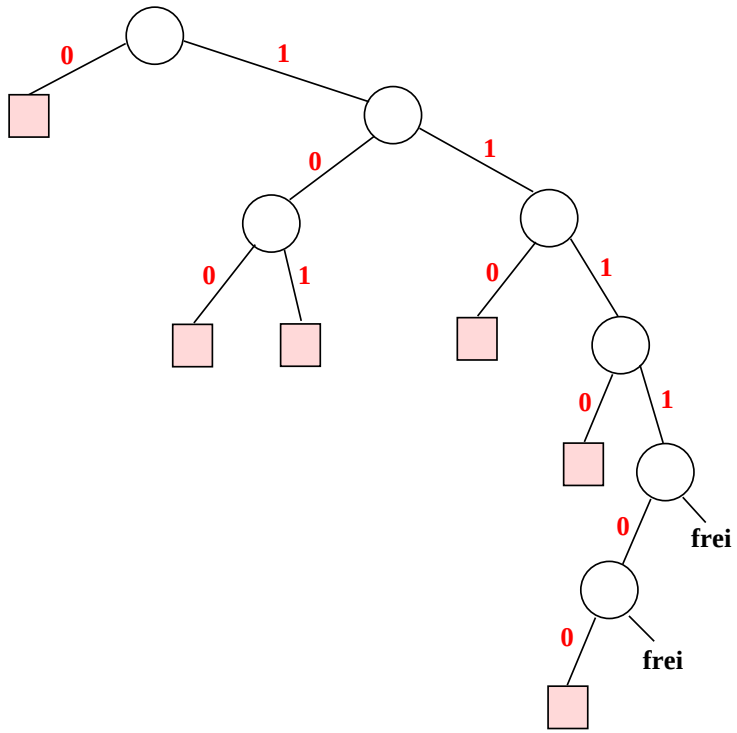
Falls $\ell_1 = 0$ ist, ist $n = 1$. Im Baum gibt es nur ein Blatt auf Niveau 0.

Sonst bauen wir iterativ einen Baum. Anfangs besteht er nur aus einer Wurzel. (Diese Wurzel hat zwei „freie Plätze“ für Kinder auf Niveau 1.)

Dann führen wir Runde k aus, für $k = 1, \ldots, n$:
Suche im Baum einen freien Platz für ein Kind auf Niveau $\ell \leq \ell_k$ und hänge an diese Stelle einen Weg der Länge $\ell_k - \ell + 1$ zu einem neuen Blatt auf Niveau ℓ_k.

Falls $\ell < \ell_k$, entstehen entlang des Wegs neue freie Plätze für Kinder auf Niveaus $\ell + 1, \ldots, \ell_k$.

Beispiel: $(\ell_1, \ldots, \ell_8) = (1, 3, 3, 3, 4, 6, 6, 6, 8)$.



Es werden nacheinander die Blätter mit folgenden Pfaden/Codewörtern angehängt: 0, 100, 101, 110, 1110, 111100.

Man überlege: Wo kommen die nächsten beiden Blätter mit Tiefe 6 hin?

Behauptung: Im Iterationsschritt wird immer ein freier Platz für ein Kind auf Niveau $\ell \leq \ell_k$ gefunden.

Beweis dafür: **Indirekt**. Da vor Runde k nur Blätter auf Niveaus $\leq \ell_k$ eingefügt wurden, sitzen innere Knoten nur auf Niveaus $0, 1, \ldots, \ell_k - 1$.

Annahme: Keiner von diesen inneren Knoten hat einen freien Platz für ein Kind.

Dann liegt ein Binärbaum mit Blättern auf Niveaus $\ell_1, \ldots, \ell_{k-1}$ (alle $\leq \ell_k$) vor, bei dem jeder innere Knoten zwei Kinder hat.

Nach Lemma 10.2.10 folgt $\sum_{1 \leq i \leq k-1} 2^{-\ell_i} = 1$.

Weil $2^{-\ell_k} > 0$, ist dann $\sum_{1 \leq i \leq n} 2^{-\ell_i} > 1$, **Widerspruch**. □

Satz 10.2.12 (Huffman versus Entropie)

Ist $p\colon A \to [0,1]$ mit $\sum_{a\in A} p(a) = 1$ gegeben, so gilt für einen **optimalen** Codierungsbaum T zu (A,p):

$$H(p_1,\ldots,p_n) \leq B(T,p) \leq H(p_1,\ldots,p_n) + 1.$$

(Informal: Setze $p_i = p(a_i)$, für $A = \{a_1,\ldots,a_n\}$. Dann gilt für die Codierung mit einem optimalen Cdierungsbaum: Die erwartete Zahl von Bits, die man braucht, um einen Text $t_1 \ldots t_m$ über A zu codieren, liegt zwischen $m \cdot H(p_1,\ldots,p_n)$ und $m \cdot (H(p_1,\ldots,p_n)+1)$.)

Beweis: 1. Ungleichung:

Es seien $l_1, \ldots, l_n$ die Tiefen der Blätter in T zu den Buchstaben $a_1, \ldots, a_n$.

Dann gilt $\sum_{i=1}^{n} 2^{-l_i} \leq 1$ (nach Satz 10.2.11, Satz von Kraft/McMillan).

Damit können wir Lemma 10.2.9 (Lemma von Gibbs) mit $q_i = 2^{-l_i}$ anwenden und erhalten

$$B(T,p) = \sum_{i=1}^{n} p_i \cdot l_i = \sum_{i=1}^{n} p_i \cdot \log(1/2^{-l_i}) \geq H(p_1, \ldots, p_n).$$

2. Ungleichung: Es genügt zu zeigen, dass ein Codierungsbaum T' für $a_1, \ldots, a_n$ existiert, in dem $B(T', p) \leq H(p_1, \ldots, p_n) + 1$ gilt.

(Ein optimaler Baum T erfüllt ja $B(T, p) \leq B(T', p)$.)

Wir setzen $l_i := \lceil \log(1/p_i) \rceil$, für $1 \leq i \leq n$, und beobachten:

$$\begin{aligned} \sum_{i=1}^{n} 2^{-l_i} &= \sum_{i=1}^{n} 2^{-\lceil \log(1/p_i) \rceil} \leq \sum_{i=1}^{n} 2^{-\log(1/p_i)} \\ &= \sum_{i=1}^{n} p_i = 1. \end{aligned}$$

Nach Satz 10.2.10 („$\Leftarrow$“) existiert also ein Präfixcode $\{x_1, \ldots, x_n\} \subseteq \{0,1\}^*$ mit Codewortlängen $l_1, \ldots, l_n$; im entsprechenden Codierungsbaum T' ordnen wir dem Blatt zu Codewort x_i den Buchstaben a_i zu. Dann ist

$$\begin{aligned} B(T', p) &= \sum_{i=1}^{n} p_i \cdot l_i \leq \sum_{i=1}^{n} p_i \cdot (\log(1/p_i) + 1) \\ &= H(p_1, \ldots, p_n) + \sum_{i=1}^{n} p_i \\ &= H(p_1, \ldots, p_n) + 1. \end{aligned}$$

Bemerkung: Es gibt bessere Kodierungsverfahren als das von Huffman (z. B. „arithmetische Kodierung“; diese vermeiden den Verlust von bis zu einem Bit pro Buchstabe), aber Huffman-Kodierung ist ein guter Anfang, der auch technisch noch eine große Rolle spielt.

SS 2021

Algorithmen und Datenstrukturen

11. Kapitel

Greedy-Algorithmen für Graphprobleme

Martin Dietzfelbinger

Juli 2021

11.1 Kürzeste Wege mit einem Startknoten: Der Algorithmus von Dijkstra

Definition 11.1.1

1. Ein **gewichteter** Digraph $\boldsymbol{G = (V, E, c)}$ besteht aus einem Digraphen (V, E) und einer Funktion $c\colon E \to \mathbb{R}$, die jeder Kante (v, w) einen Wert $c(v, w)$ zuordnet.

   c steht für „**cost**“; $c(v, w)$ kann als „Kosten“ oder „Länge“ oder „Gewicht“ der Kante (v, w) interpretiert werden.

2. Ein gerichteter Kantenzug $p = (v_0, v_1, \ldots, v_k)$ in G hat Kosten/Länge/Gewicht

$$\boldsymbol{c(p)} = \sum_{1 \leq i \leq k} c(v_{i-1}, v_i).$$

3. Der **(gerichtete) Abstand** von $v, w \in V$ ist

$$\boldsymbol{d(v,w)} := \min\{c(p) \mid p \text{ Kantenzug von } v \text{ nach } w\}$$

($= \boldsymbol{\infty}$, falls kein Kantenzug von v nach w existiert;
$= \boldsymbol{-\infty}$, falls es von v nach w Kantenzüge mit beliebig stark negativen Kosten gibt.)

Klar: $d(v,v) \leq 0$ (wegen des Kantenzugs (v) mit Kosten 0).

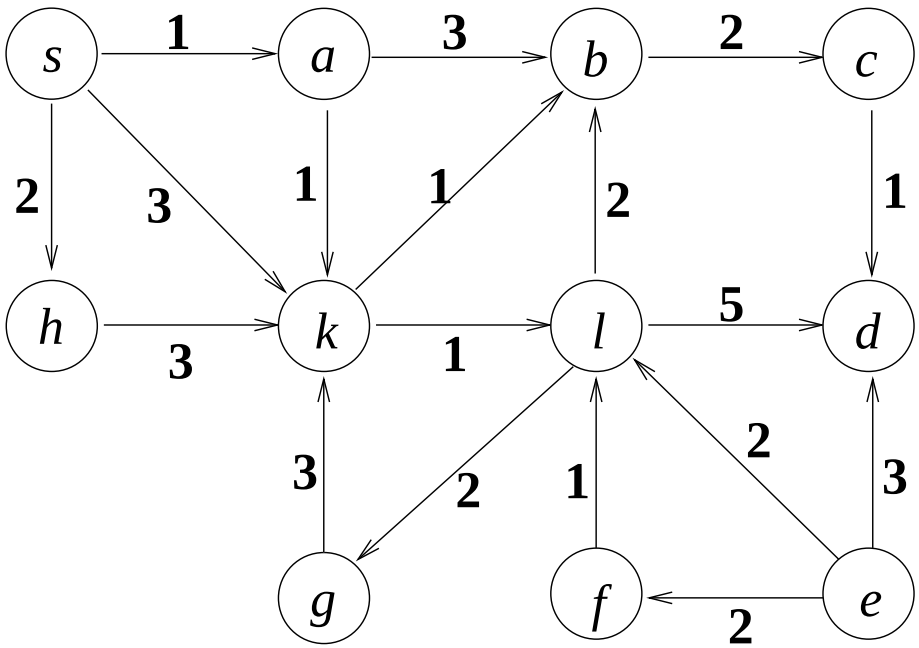
Bemerkung

Wenn alle Kantengewichte ≥ 0 sind, gilt:

$$d(v,w) = \text{minimale Länge eines (einfachen) } \textbf{Weges} \text{ von } v \text{ nach } w.$$

(Man kann aus einem Kantenzug von v nach w Kreise herausschneiden, ohne die Länge zu vergrößern.)

Beispiel: Digraph mit nichtnegativen Kantenkosten.



$c((s,a,b,c)) = 6, \quad c((s,a,k,b,c)) = 5, \quad d(s,c) = 5;$
$d(s,s) = 0;$
$d(s,e) = d(s,f) = \infty.$

Hier betrachten wir einen Algorithmus für das Problem

„Single-Source-Shortest-Paths (SSSP)“

(Kürzeste Wege von einem Startknoten aus).

Gegeben:

Gewichteter Digraph $G = (V, E, c)$ mit Kantenkosten $c(v, w) \geq 0$ und $s \in V$.

Gesucht:

Für jedes $v \in V$ der Abstand $d(s, v)$ und im Fall $d(s, v) < \infty$ ein Weg von s nach v der Länge $d(s, v)$.

Der **Algorithmus von Dijkstra** löst dieses Problem.

(Aussprache: „Daik-stra“.)
Edsger W. Dijkstra, 1930–2002, niederländischer Informatiker, Pionier der „Strukturierten Programmierung“, Erfinder des Semaphorkonzepts für die Synchronisation von Prozessen, Turingpreis 1972.

(Zündende) Idee:

Eine Kante (v, w) wird als „Einbahnstraßen-Zündschnur“ mit *Länge* $c(v, w)$ gedacht.

Die Zündschnüre brennen mit konstanter Geschwindigkeit 1 pro Sekunde.

Wenn das Feuer einen Knoten v erstmals erreicht, zum Zeitpunkt $t = t_v$, fangen alle Zündschnüre, die von v ausgehen, an zu brennen.

Wenn später oder gleichzeitig das Feuer über andere Schnüre nochmals bei v ankommt, passiert nichts weiter.

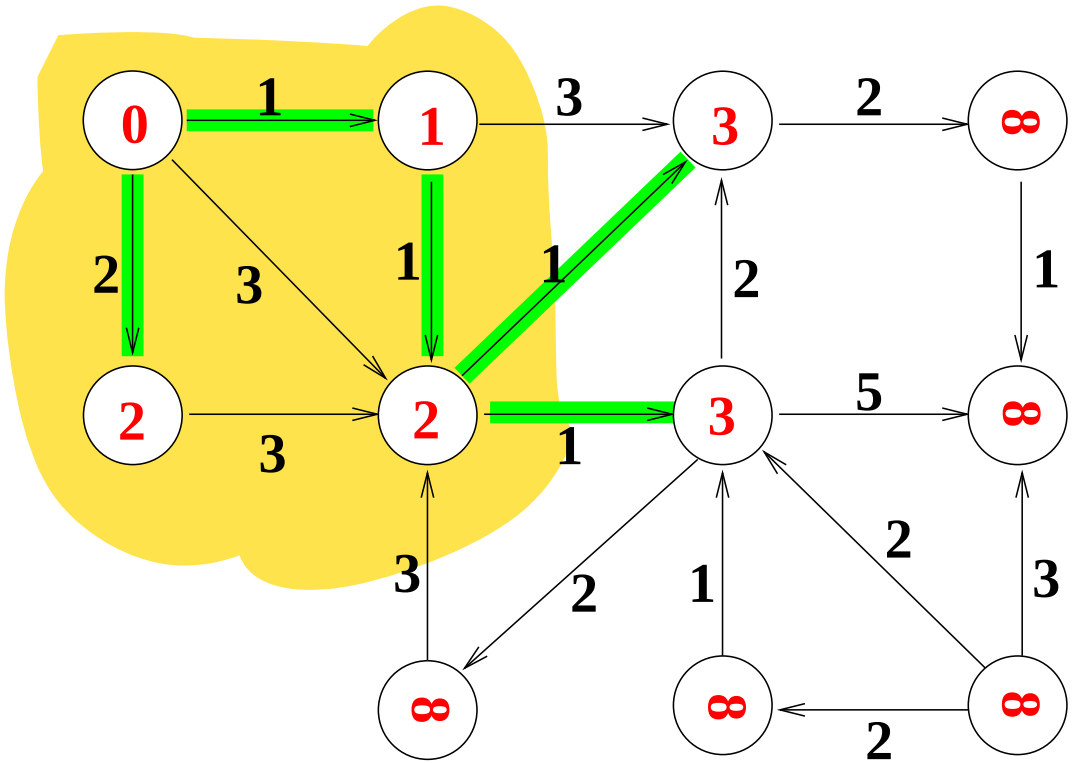
Zum Zeitpunkt $t_s = 0$ halten wir ein Zündholz an den Knoten s.

Veranschaulichung: Auf späteren Folien. **Orange**: Schon erreichte Knoten.

Zahlen in Knoten: Wann erreicht das Feuer den Knoten, *nach aktuellem Stand*?

Über **grüne** Kanten werden bisher unerreichte Knoten **nach aktuellem Stand erstmals** erreicht. (Diese Kanten muss man also beobachten.)

Ablauf des Algorithmus von Dijkstra



Zeitpunkt 2, zwei neue Knoten erreicht, neue Schnüre.

„Klar“ **(??)**:

Das Feuer erreicht den Knoten v genau zum Zeitpunkt $t_v = d(s, v)$.

Denn: Der Zeitraum $[0, d(s, v)]$ genügt, damit das Feuer einen kürzesten Weg von s nach v durchwandern kann, und es kann nicht schneller gehen.

Wir bilden diese Idee nun algorithmisch nach.

Wichtige Beobachtung: Nur die n Zeitpunkte $t_v = d(s, v)$, $v \in V$, sind interessant.

Dazwischen wandert das Feuer irgendwelche Kanten entlang, ohne dass im Prinzip viel passiert (selbst wenn ein schon erreichter Knoten nochmals erreicht wird).

O.B.d.A.: $V = \{1, \ldots, n\}$.

Der Algorithmus arbeitet in bis zu n Runden, eine für jeden erreichbaren Knoten.

V ist stets in zwei disjunkte Mengen S (**orange**) und $V - S$ (weiß) zerlegt.

In jeder Runde wächst S um einen Knoten.

Anfangs: $S \leftarrow \emptyset$.

Array $\texttt{dist}[1..n]$ speichert Zeiten.

Für $v \in S$: $\texttt{dist}[v] = d(s,v) \quad (= t_v)$.

Für $w \notin S$: $\texttt{dist}[w] = \min\{\texttt{dist}[v] + c(v,w) \mid v \in S, (v,w) \in E\}$.

(Der Zeitpunkt, zu dem das Feuer *nach gegenwärtigem Stand der Dinge* Knoten w erreichen wird. Wenn es keine Kante von S nach w gibt, ist $\texttt{dist}[w] = \infty$.)

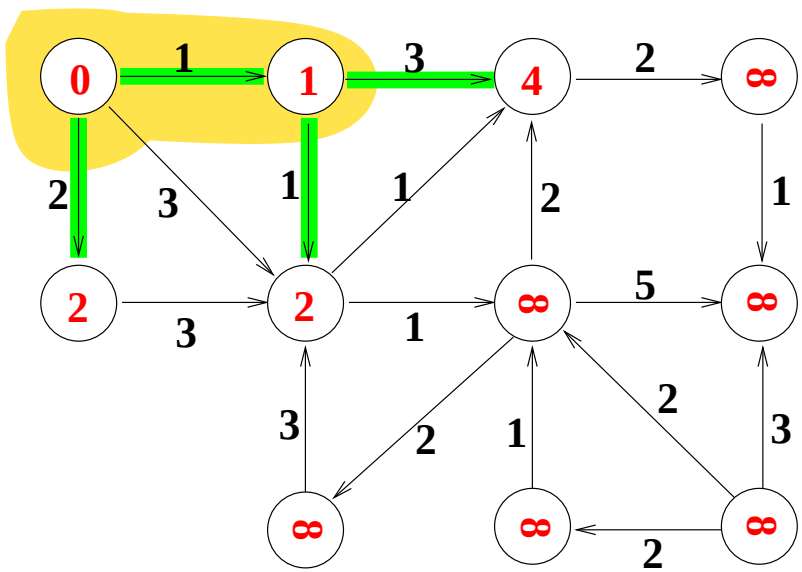
Initialisierung:

$S \leftarrow \emptyset$.

$\texttt{dist}[s] \leftarrow 0$.

Für alle $w \neq s$: $\texttt{dist}[w] \leftarrow \infty$.

Beispiel: **Orange:** S.



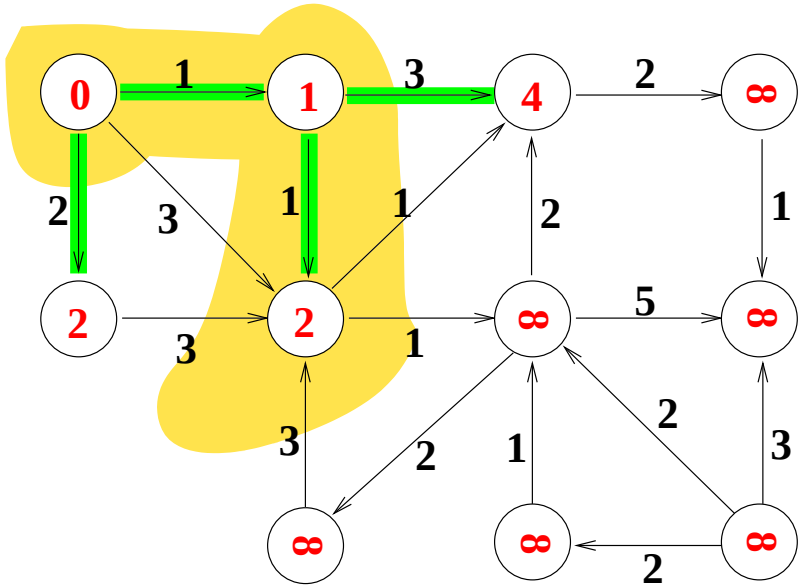
Rote Zahlen in den Knoten sind die `dist[·]`-Werte.

Runde: Bestimme den Knoten $u \in V - S$, der `dist[`w`]`, $w \in V - S$, minimiert.

(Hier: Einer der Knoten mit dist-Wert 2. Wenn es mehrere gibt, wähle einen beliebigen davon.)

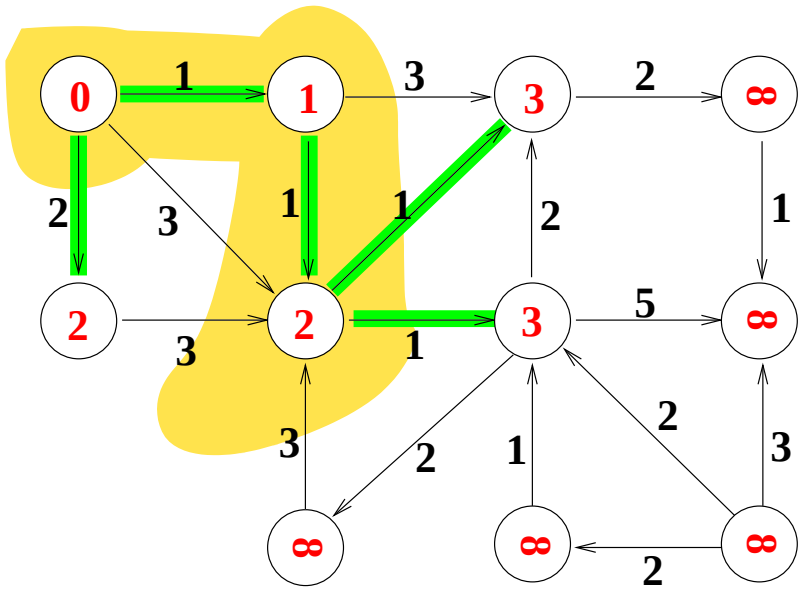
Füge Knoten u zu S hinzu. (Er „brennt an“. Der aktuelle Wert `dist[`u`]` wird „eingefroren“.)

Beispiel:



Was ist noch zu tun?

Beispiel:



Was ist noch zu tun? Auf Kante (u, w) kann neuer Knoten $w \in V - S$ erreicht werden, oder $w \in V - S$ kann schneller erreicht werden. Also: Eventuell $\texttt{dist}[w]$ aktualisieren:

$$\texttt{dist}[w] \leftarrow \min\{\texttt{dist}[w], \texttt{dist}[u] + c(u, w)\}.$$

Der bisher entwickelte Algorithmus berechnet schon die **Länge** der kürzesten Wege (also die **Zeitpunkte**, zu denen das Feuer die Knoten erreicht).

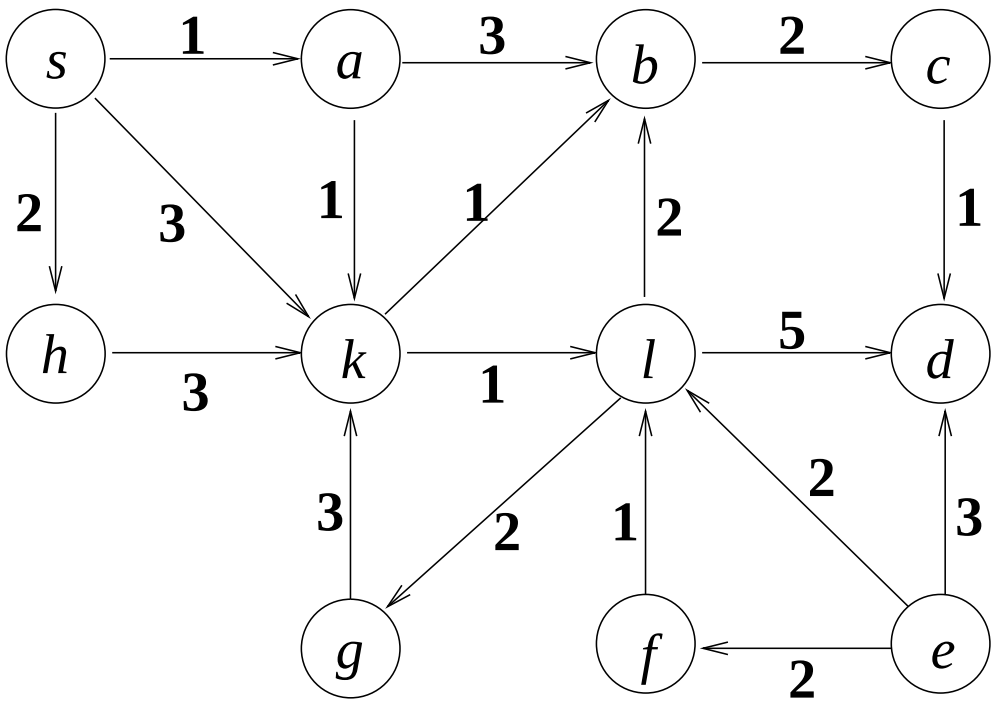
Algorithmus Dijkstra-Distanzen(G, s)

Eingabe: gewichteter Digraph $G = (V, E, c)$, Startknoten $s \in V$
Ausgabe: Länge der kürzesten Wege von s zu den Knoten in G

(1) $\mathtt{S} \leftarrow \emptyset$;
(2) $\mathtt{dist}[s] \leftarrow 0$;
(3) **for** $\mathtt{w} \in V - \{s\}$ **do** $\mathtt{dist}[\mathtt{w}] \leftarrow \infty$;
(4) **while** $\exists u \in V - \mathtt{S}$: $\mathtt{dist}[u] < \infty$ **do** // „Runde“
(5) $\quad \mathtt{u} \leftarrow$ ein solcher Knoten u mit minimalem $\mathtt{dist}[u]$;
(6) $\quad \mathtt{S} \leftarrow \mathtt{S} \cup \{\mathtt{u}\}$;
(7) $\quad$ **for** $\mathtt{w}$ mit $(\mathtt{u}, \mathtt{w}) \in E$ und $\mathtt{w} \notin \mathtt{S}$ **do** // Nachfolger von u, nicht bearbeitet
(8) $\quad\quad \mathtt{dist}[\mathtt{w}] \leftarrow \min\{\mathtt{dist}[\mathtt{w}], \mathtt{dist}[\mathtt{u}] + c(\mathtt{u}, \mathtt{w})\}$;
(9) **Ausgabe:** das Array $\mathtt{dist}[1..n]$.

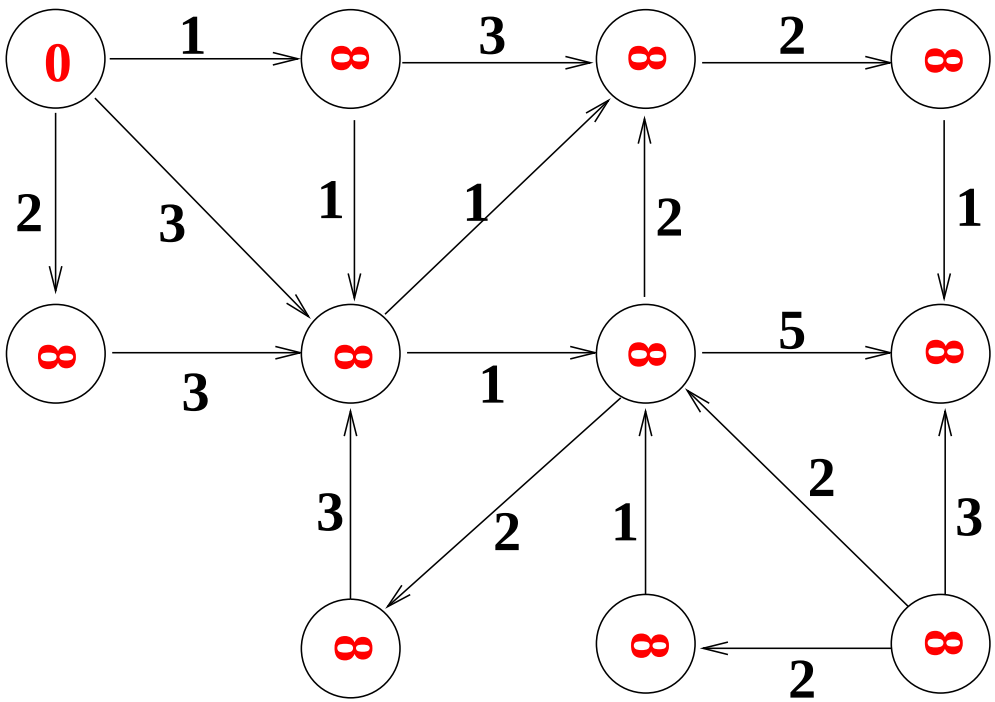
(Technisch wird S durch einen Bitvektor $\mathtt{inS}[1..n]$ realisiert.)

Ablauf des Algorithmus von Dijkstra



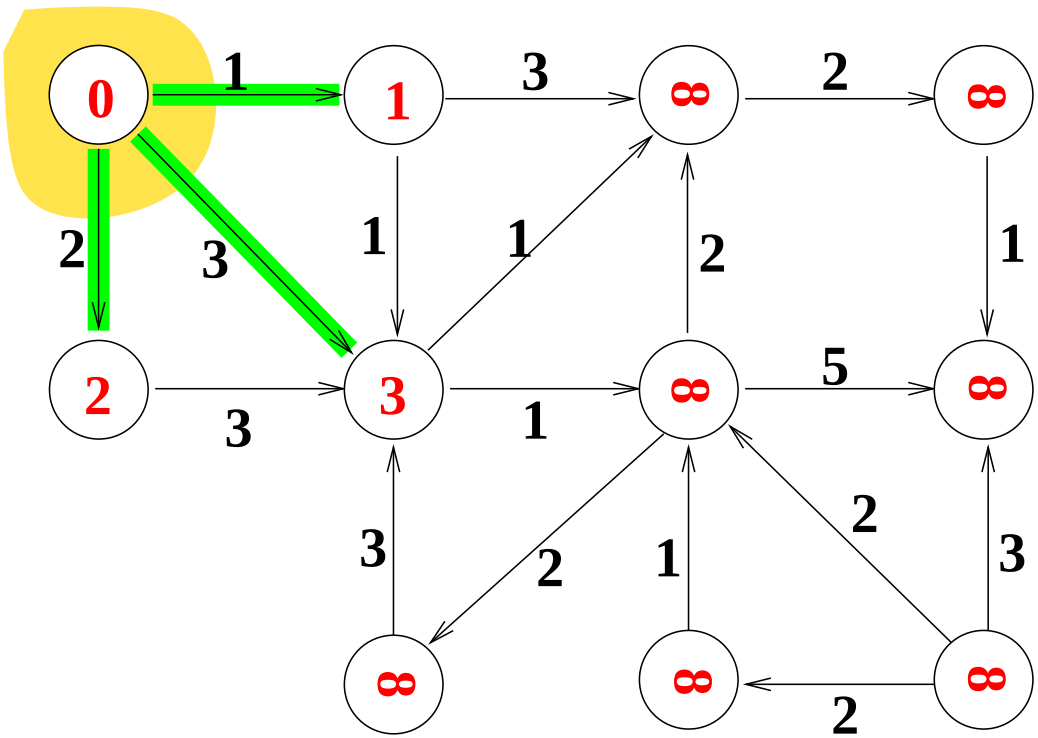
Der Ausgangsgraph.

Ablauf des Algorithmus von Dijkstra



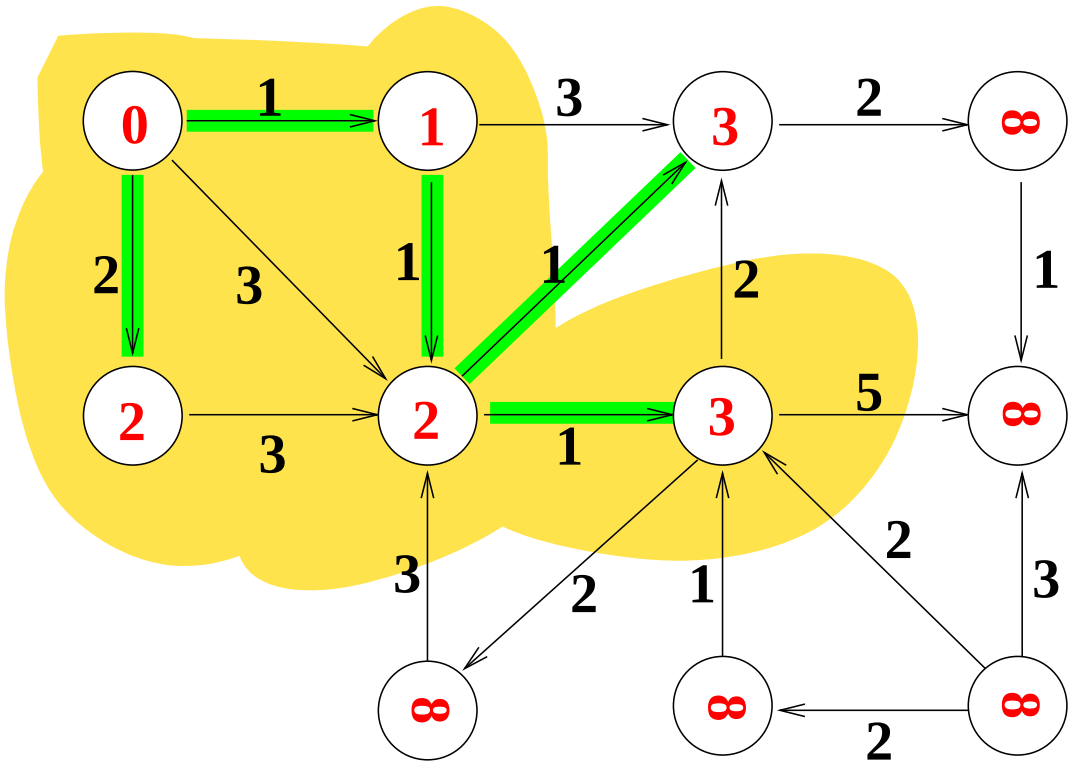
Nach der Initialisierung (Zeilen (1)–(3+)) (Folie 19).

Ablauf des Algorithmus von Dijkstra



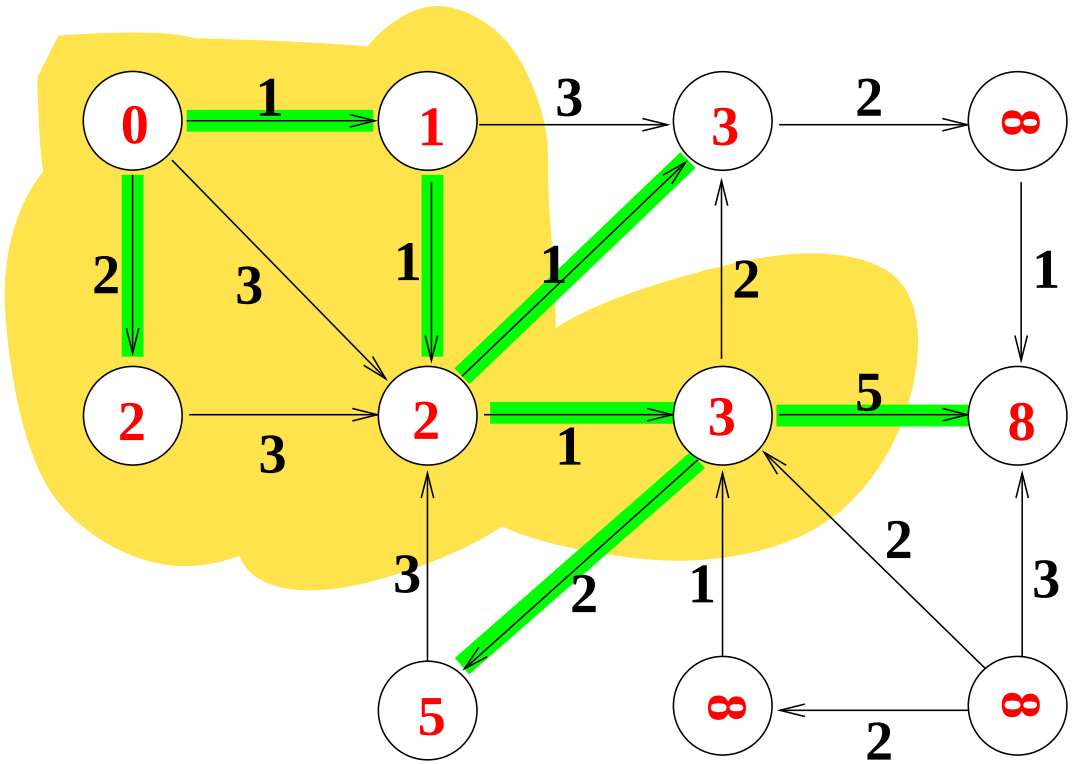
Nach Zeilen (5)–(8d) für $u = s$ (Folie 19).

Ablauf des Algorithmus von Dijkstra



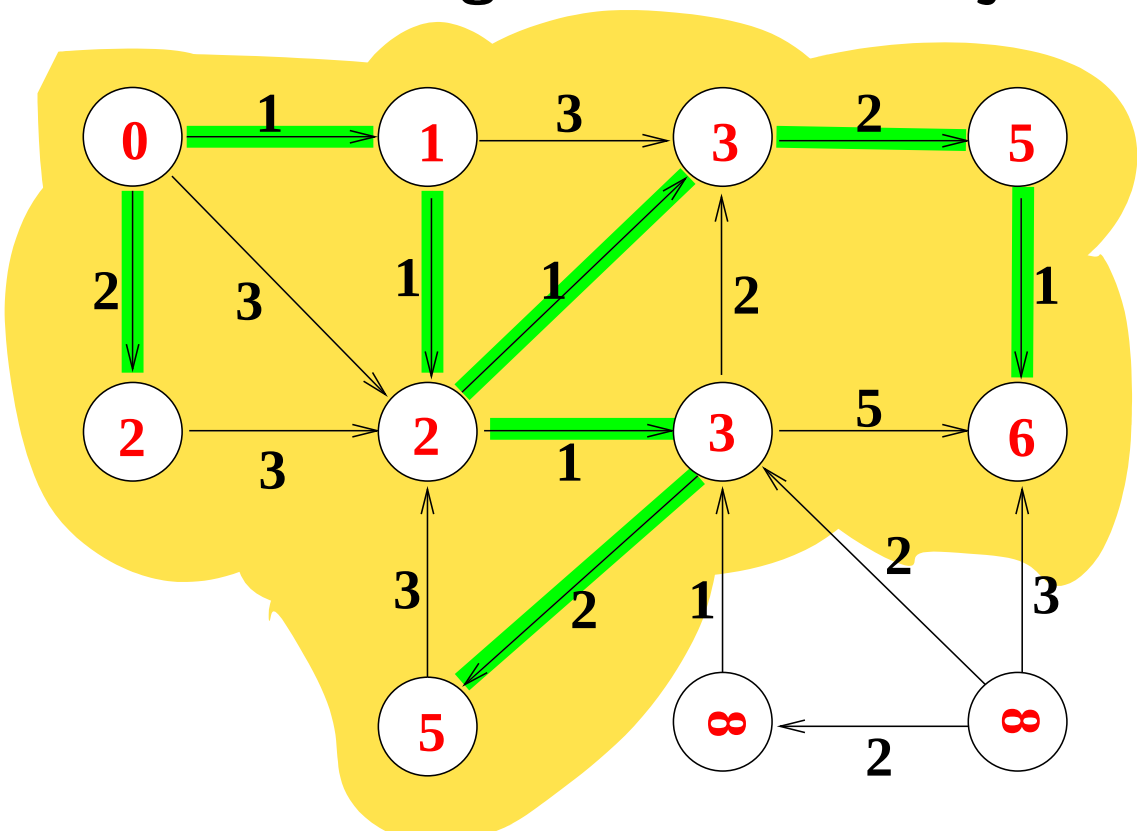
$u = l$, Zeile (6) (Folie 19).

Ablauf des Algorithmus von Dijkstra



$u = l$, Zeilen (7)–(8d) (Folie 19).

Ablauf des Algorithmus von Dijkstra



Alle $v \in V - \mathtt{S}$ erfüllen $\mathtt{dist}[v] = \infty$: **Ende**.

Lemma 11.1.2

Der Algorithmus **Dijkstra-Distanzen** gibt in `dist[`v`]` für alle $v \in V$ den Wert $d(s, v)$ aus.

Beweis:

Wenn Knoten v (mit v in `u`) in Zeilen (5)–(8) betrachtet wird, sagen wir, dass v **bearbeitet** wird.

Wenn `dist[`v`]` in Zeile (2) oder (8) (mit v in `w`) erstmals auf einen Wert $< \infty$ gesetzt wird, sagen wir, dass Knoten v **gefunden** wird.

Unerreichbare Knoten: Durch eine einfache Induktion (wie bei BFS oder DFS) zeigt man, dass jeder Knoten v, der von s aus erreichbar ist, irgendwann gefunden und irgendwann danach bearbeitet wird.

Genau diejenigen Knoten v, die nicht von s aus erreichbar sind, behalten also den Wert `dist[`v`]` $= \infty$.

Nun zeigen wir durch Induktion über Runden die folgenden **Invarianten**, gültig am Ende jeder Runde:

(I1) Für alle $v \in V$ gilt:

$$\texttt{dist}[v] < \infty \Rightarrow \ \exists \text{ Weg von } s \text{ nach } v \text{ mit Länge} \leq \texttt{dist}[v].$$

(I2) Für alle $v \in S$ gilt $\texttt{dist}[v] = d(s, v)$.

Beweis von (I1) durch Induktion über Runden:

Nach der Initialisierung gilt $\texttt{dist}[v] < \infty$ nur für $v = s$; es gibt einen Weg von s nach s der Länge 0.

Betrachte nun eine Runde, in der u bearbeitet wird, und einen Knoten $v \in V - S$.

Wenn sich $\texttt{dist}[v]$ in dieser Runde nicht ändert, ist nichts zu zeigen.

Wenn $\texttt{dist}[v]$ in dieser Runde geändert wird, dann auf den Wert $\texttt{dist}[u] + c(u, v)$.
Nach I.V. gibt es einen Weg p_u von s nach u der Länge höchstens $\texttt{dist}[u]$. Wenn wir p_u um die Kante (u, v) verlängern, erhalten wir einen Weg von s nach v der Länge höchstens $\texttt{dist}[u] + c(u, v) = \texttt{dist}[v]$.

Beweis von (I2) durch Induktion über Runden:

I.A.: Am Anfang ist S leer. In der ersten Runde wird offensichtlich der Startknoten s in S aufgenommen, und $d(s, s) = \texttt{dist}[s] = 0$.

I.V.: Sei $u \neq s$.

(I2) stimmt für alle Runden vor derjenigen, in der u bearbeitet, also in S aufgenommen wird.

I.S.: Betrachte Runde, in der $u \neq s$ bearbeitet wird.

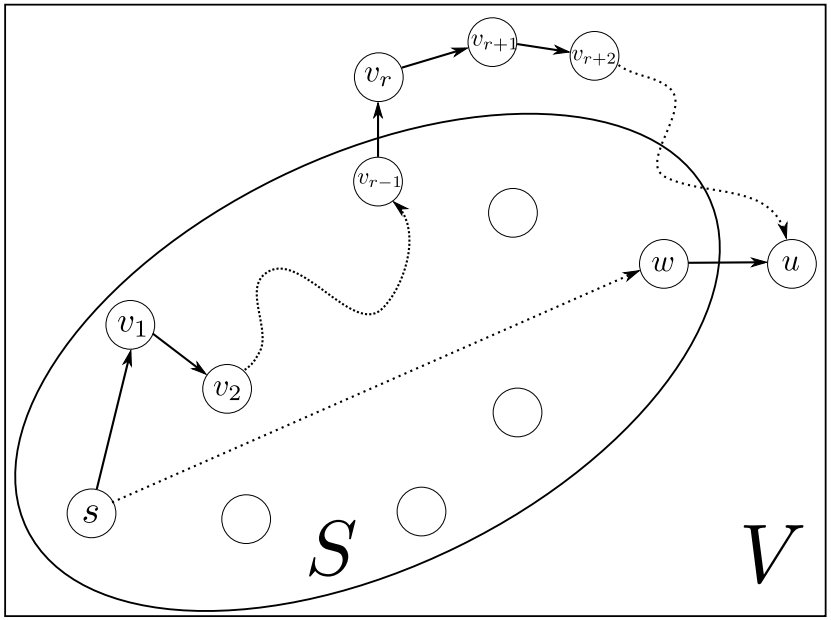
Nach (I1) gibt es einen Weg von s nach u der Länge $\leq \texttt{dist}[u]$.

Zu zeigen bleibt: Es gibt keinen Weg von s nach u, der kürzer als $\texttt{dist}[u]$ ist.

Sei $p = (s = v_0, v_1, \ldots, v_t = u)$ irgendein Weg von s nach u.

p beginnt in $v_0 = s \in$ S und endet in $v_t = u \in V -$ S.

Also gibt es ein r mit $s = v_0, v_1, \ldots, v_{r-1} \in$ S und $v_r \notin$ S.



Nach der Definition der Distanzfunktion gilt für das Anfangsstück $p_{r-1} = (v_0, \ldots, v_{r-1})$ von p:
$c(p_{r-1}) \geq d(s, v_{r-1})$.
Nach I.V. für $v_{r-1} \in \mathtt{S}$ gilt $d(s, v_{r-1}) = \mathtt{dist}[v_{r-1}]$.
Also haben wir $c(p_r) \geq \mathtt{dist}[v_{r-1}] + c(v_{r-1}, v_r)$,
für das Anfangsstück $p_r = (v_0, \ldots, v_{r-1}, v_r)$ von p.
Weil nach Vor. alle **Kantengewichte nichtnegativ** sind, insbesondere also $c(p) \geq c(p_r)$ gilt, folgt

$(*)$ $\quad c(p) \geq \mathtt{dist}[v_{r-1}] + c(v_{r-1}, v_r)$.

Weiter gilt: $(**)$ $\quad \mathtt{dist}[v_{r-1}] + c(v_{r-1}, v_r) \geq \mathtt{dist}[v_r]$.
(Denn: In der Runde, in der v_{r-1} bearbeitet wurde, wurden $\mathtt{dist}[v_{r-1}] + c(v_{r-1}, v_r)$ und $\mathtt{dist}[v_r]$ verglichen, und $(**)$ wurde erzwungen. Wenn sich $\mathtt{dist}[v_r]$ danach noch geändert hat, ist es nur kleiner geworden.)

Schließlich gilt in der aktuellen Runde: $(***)$ $\quad \mathtt{dist}[v_r] \geq \mathtt{dist}[u]$.

Dies liegt daran, dass der Algorithmus einen Knoten mit kleinstem $\mathtt{dist}[v]$-Wert als u wählt.

Kombinieren von $(*)$, $(**)$ und $(***)$ liefert $c(p) \geq \mathtt{dist}[u]$, wie gewünscht. □

Wir wollen aber eigentlich nicht nur die Distanzen $d(s, v)$, sondern **kürzeste Wege** berechnen.

Idee: Für jeden Knoten v merken wir uns, über welche Kante (u, v) Knoten v „vom Feuer erreicht wurde“.

Wenn wir diese „Vorgänger-Information“ benutzen, um von v ausgehend immer weiter rückwärts zu laufen, bis wir s erreichen, erhalten wir einen kürzesten Weg.

Technisch: Für jeden gefundenen Knoten $w \notin \mathtt{S}$ notiere $p(w) \in \mathtt{S}$ mit $(p(w), w) \in E$ und $\mathtt{dist}[w] = \mathtt{dist}[p(w)] + c(p(w), w)$ $(= d(s, p(w)) + c(p(w), w))$.

Ab dem Moment, in dem w bearbeitet wird, ändert sich $p(w)$ nicht mehr.

Datenstruktur für die Vorgänger: $\mathtt{p}[1..n]$.

Notwendige Ergänzungen:

(2+). . . $\texttt{p}[s] \leftarrow -2$; // Sonderfall Wurzel

(3+) **for** $\texttt{w} \in V - \{s\}$ **do** . . . $\texttt{p}[\texttt{w}] \leftarrow -1$; // „undefiniert“

Aktualisierung in den späteren Runden:

(7) **for** $(\texttt{u}, \texttt{w}) \in E$ mit $\texttt{w} \notin \texttt{S}$ **do** // Nachfolger von u, nicht bearbeitet: „update(u, w)“
(8a) $\texttt{dd} \leftarrow \texttt{dist}[\texttt{u}] + c(\texttt{u}, \texttt{w})$;
(8b) **if** $\texttt{dd} < \texttt{dist}[\texttt{w}]$ **then**
(8c) $\texttt{dist}[\texttt{w}] \leftarrow \texttt{dd}$;
(8d) $\texttt{p}[\texttt{w}] \leftarrow \texttt{u}$;

Algorithmus Dijkstra(G, s)

Eingabe: gewichteter Digraph $G = (V, E, c)$, Startknoten $s \in V$

Ausgabe: Länge $d(s, v)$ der kürzesten Wege, Vorgängerknoten $p(v)$

(1) $\mathtt{S} \leftarrow \emptyset$;

(2+) $\mathtt{dist}[s] \leftarrow 0$; $\mathtt{p}[s] \leftarrow -2$;

(3+) **for** $\mathtt{w} \in V - \{s\}$ **do** $\mathtt{dist}[\mathtt{w}] \leftarrow \infty$; $\mathtt{p}[\mathtt{w}] \leftarrow -1$;

(4) **while** $\exists u \in V - \mathtt{S}$: $\mathtt{dist}[u] < \infty$ **do**

(5) $\quad \mathtt{u} \leftarrow$ ein solcher Knoten u mit minimalem $\mathtt{dist}[u]$;

(6) $\quad \mathtt{S} \leftarrow \mathtt{S} \cup \{\mathtt{u}\}$;

(7) $\quad$ **for** $(\mathtt{u}, \mathtt{w}) \in E$ mit $\mathtt{w} \notin \mathtt{S}$ **do** // Nachfolger von u, nicht bearbeitet: „update(u, w)“

(8a) $\quad \mathtt{dd} \leftarrow \mathtt{dist}[\mathtt{u}] + c(\mathtt{u}, \mathtt{w})$;

(8b) $\quad$ **if** $\mathtt{dd} < \mathtt{dist}[\mathtt{w}]$ **then**

(8c) $\quad\quad \mathtt{dist}[\mathtt{w}] \leftarrow \mathtt{dd}$;

(8d) $\quad\quad \mathtt{p}[\mathtt{w}] \leftarrow \mathtt{u}$;

(9+) **Ausgabe:** $\mathtt{dist}[1..n]$ und $\mathtt{p}[1..n]$.

Nach dem Algorithmus ist klar, dass $\texttt{p}[v] \neq -1$ („undefiniert“) genau dann gilt, wenn $\texttt{dist}[v] < \infty$.

$\texttt{p}[v] = -2$ („Startknoten, hat keinen Vorgänger“) gilt nur für $v = s$.

Definition Ein (einfacher) Weg $(s = v_0, v_1, \ldots, v_t)$ in G heißt ein ***S*-Weg**, wenn alle Knoten außer eventuell v_t in S liegen.

Behauptung: Neben (I1) und (I2) gelten nach jeder Runde die folgenden Invarianten:

(I3) Wenn $v \in \texttt{S}$ und $v \neq s$, dann gilt $\texttt{p}[v] \neq -1$ und $\texttt{p}[v]$ ist vorletzter Knoten auf einem S-Weg von s nach v der Länge $d(s, v)$.

(I4) Wenn $w \notin \texttt{S}$ und $w \neq s$ und $\texttt{dist}[w] < \infty$, dann gilt: $\texttt{p}[w] \in \texttt{S}$ und $\texttt{p}[w]$ ist letzter S-Knoten auf einem S-Weg von s nach w der Länge $\texttt{dist}[w]$.

Beweis von (I3) und (I4), für Interessierte: Induktion über Runden.
Nach Runde 0 (also vor Beginn) sind Aussagen (I3) und (I4) trivialerweise wahr, weil $S = \emptyset$ ist und $\texttt{dist}[w] < \infty$ für alle $w \neq s$ gilt.
Nun betrachte das Ende von Runde 1, in der also s bearbeitet wird. Da $S = \{s\}$ gilt, ist (I3) trivial. Wenn $w \neq s$ und $\texttt{dist}[w] < \infty$, gibt es eine Kante (s, w) und $\texttt{dist}[w] = c(s, w)$, und es gilt $\texttt{p}[w] = s$. Also ist auch (I4) richtig.
Nun betrachten wir Runde $t \geq 2$ durch die **while**-Schleife. Damit diese Runde überhaupt stattfindet, muss es einen Knoten $u \in V - S$ mit $\texttt{dist}[u] < \infty$ geben.
Der Algorithmus wählt ein solches u mit minimalem $\texttt{dist}[u]$ und bearbeitet u. Die Menge S wird dabei auf $S' = S \cup \{u\}$ vergrößert.
Nach I.V. (I4) war vor Runde t Knoten $w = \texttt{p}[u] \in S$ der letzte Knoten auf einem S-Weg kürzester Länge $\texttt{dist}[u]$. Nach (I1), angewendet auf u, ist dies ein kürzester Weg von s nach u (der ganz in S' verläuft). Damit gilt (I3) nun auch für u. Für die Knoten $v \in S$ ändert sich nichts.
Für (I4) muss man nur die Knoten w mit $(u, w) \in E$ betrachten, für die sich in Zeile (8b) die Beziehung $\texttt{dd} < \texttt{dist}[w]$ ergibt. Nach I.V. (I4) enthält $\texttt{dist}[w]$ entweder ∞ oder die Länge eines kürzesten S-Weges nach w.
dd enthält dann die Länge eines kürzeren Weges, der erst (wegen (I1) für u) mit Kosten $d(s, u)$ nach u und dann über die Kante (u, w) verläuft. Dies ist ein kürzester S'-Weg nach w, und er hat $u = \texttt{p}[w]$ als letzten Knoten in S'. □

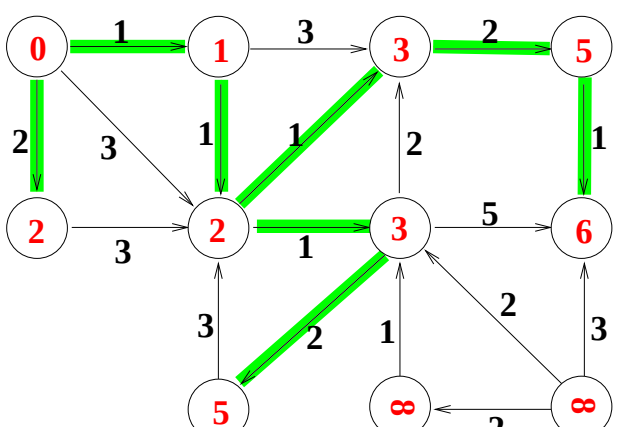
Ergebnis:

Wenn der Algorithmus von Dijkstra anhält, führen die $\mathtt{p}[v]$-Verweise von jedem Knoten v aus entlang eines kürzesten Weges (zurück) zu s.

Da die $\mathtt{p}[v]$-Verweise keinen Kreis bilden können (mit dem Schritt $\mathtt{S} \leftarrow \mathtt{S} \cup \{u\}$ wird der Verweis vom neuen S-Knoten u auf den S-Knoten $\mathtt{p}[u]$ endgültig fixiert), bilden die Kanten $(\mathtt{p}[v], v)$ einen **Baum**, den sogenannten

Baum der kürzesten Wege.

Beispiel:



Wieso steht der Algorithmus von Dijkstra unter der Überschrift „Greedy-Algorithmen“?
Er hält eine sehr geschickte Datenstruktur bereit, wählt dann aber in jeder Runde nach dem Greedy-Muster als nächsten einzubeziehenden Knoten einen, dessen `dist`$[v]$-Wert minimal ist.
Weiter unten lernen wir den Algorithmus von Jarník/Prim für das Problem „Minimaler Spannbaum“ kennen, der ein Paradebeispiel für einen „Greedy“-Algorithmus ist und der sich nur minimal vom Algorithmus von Dijkstra unterscheidet.

Implementierungsdetails:

Noch zu klären:
Wie findet man **effizient** einen Knoten u mit kleinstem Wert `dist`$[u]$?

Einfache Lösung:
Durchsuche in jeder Runde das `dist`-Array, um das Minimum zu finden.

Dann benötigt jede Runde Zeit $\Theta(n)$, und die Gesamtrechenzeit des Algorithmus von Dijkstra ergibt sich zu $\Theta(n^2)$.

Dies ist für „dichte“ Graphen, also Graphen mit sehr vielen Kanten, akzeptabel, **nicht** aber für Graphen mit $|E| \ll |V|^2$ („dünn besetzte“ Graphen).

Effiziente Alternative:

Verwalte die Knoten $w \in V - \mathtt{S}$ mit Werten $\mathtt{dist}[w] < \infty$ mit den $\mathtt{dist}[w]$-Werten als **Schlüssel** in einer

Prioritätswarteschlange PQ.

Wenn $\mathtt{dist}[w] = \infty$, ist w (noch) nicht in **PQ**.

extractMin liefert den nächsten Knoten u, der zu S hinzugefügt werden soll.

`inS: array[1..`n`] of Boolean` (Knoten in S).

Dijkstra(G, s) // (Vollversion mit Prioritätswarteschlange)

Eingabe: gewichteter Digraph $G = (V, E, c)$, $V = \{1, \ldots, n\}$, Startknoten s;
Ausgabe: Länge $d(s, v)$ der kürzesten Wege, Vorgängerknoten $p(v)$
Hilfsdatenstrukturen: **PQ**: eine (anfangs leere) Prioritäts-WS; `inS`, `p`, `dist`: s.o.

```
(1)   for w from 1 to n do
(2)     dist[w] ← ∞; inS[w] ← false; p[w] ← −1;
(3)   dist[s] ← 0; p[s] ← −2; PQ.insert(s);
(4)   while not PQ.isempty do
(5)     u ← PQ.extractMin; inS[u] ← true;   // u wird bearbeitet
(6)     for Knoten w mit (u, w) ∈ E and not inS[w] do
(7)       dd ← dist[u] + c(u, w);
(8)       if p[w] ≥ 0 and dd < dist[w] then
(9)         PQ.decreaseKey(w,dd); p[w] ← u; dist[w] ← dd;
(10)      if p[w] = −1 then   // w wird soeben entdeckt
(11)        dist[w] ← dd; p[w] ← u; PQ.insert(w);
(12)  Ausgabe: dist[1..n] und p[1..n].
```

Aufwandsanalyse:

Die Prioritätswarteschlange realisieren wir als (binären) Heap (s. Abschnitt 6.4).

Maximale Anzahl von Einträgen: n.

Initialisierung: Zeit $O(1)$ für **PQ**, $O(n)$ für den Rest.

Es gibt maximal n Durchläufe durch die **while**-Schleife mit Organisationsaufwand jeweils $O(1)$, zusammen also Kosten $O(n)$ für die Schleifenorganisation.

In Schleifendurchlauf Nummer t, in dem u_t bearbeitet wird:
PQ.extractMin kostet Zeit $O(\log n)$.
Durchmustern der $\deg(u_t)$ Nachbarn von u_t:
Jedes **PQ.insert** oder **PQ.decreaseKey** kostet Zeit $O(\log n)$.

Insgesamt:

$$n \cdot O(\log n) + \sum_{1 \le t \le n} O(\deg(u_t) \cdot \log n)$$

$$= \quad O\Big(n \log n + \log n \cdot \Big(\sum_{1 \le t \le n} \deg(u_t)\Big)\Big) = O(n \log n + m \log n),$$

wobei $n = |V|$ (Knotenzahl), $m = |E|$ (Kantenzahl).

Satz 11.1.3

Der **Algorithmus von Dijkstra** mit Verwendung einer Prioritätswarteschlange, die als Binärheap realisiert ist, ermittelt kürzeste Wege von Startknoten s aus in einem Digraphen $G = (V, E, c)$ in Zeit $O((n + m) \log n)$.

Mitteilung 11.1.4

Es gibt eine Implementierung der Datenstruktur **Prioritätswarteschlange** mit folgenden Rechenzeiten:

empty und *isempty* (und das Ermitteln des kleinsten Eintrags) kosten konstante Zeit;
extractMin kostet Zeit $O(\log n)$; n Einfügungen kosten zusammen Zeit $O(n \log n)$;
m *decreaseKey*-Operationen benötigen **zusammen** Zeit $O(m)$.

„Fibonacci-Heaps“

(Fortgeschritten, siehe Master-Vorlesung „Effiziente Algorithmen“ oder das Buch [Cormen et al., Introduction to Algorithms].)

Resultierende **Rechenzeit für Algorithmus von Dijkstra:**

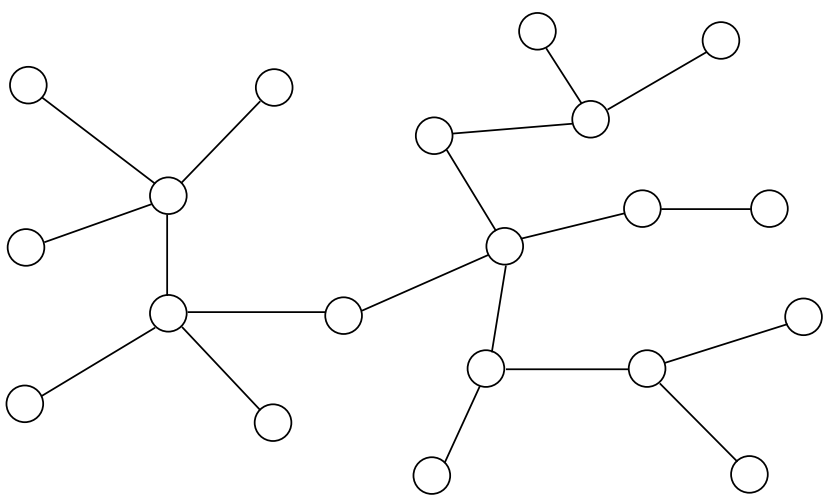
$$O(m + n \log n)$$

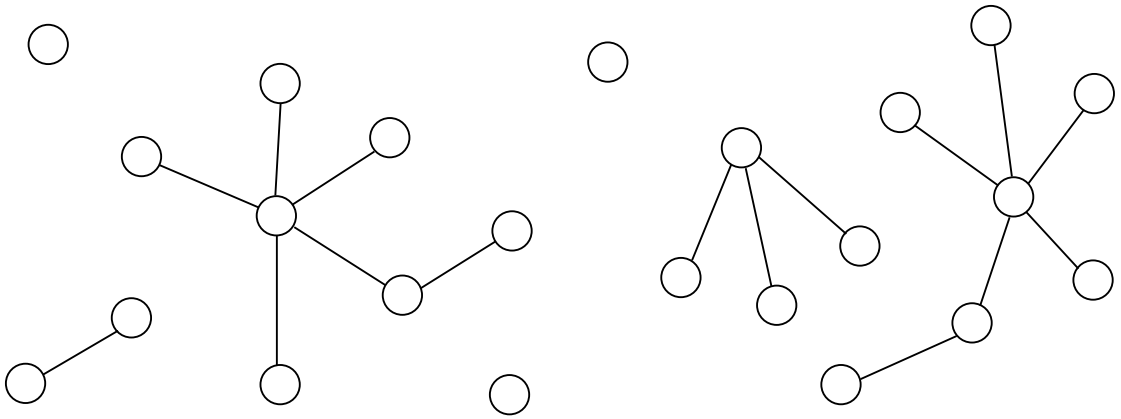
Lineare Rechenzeit für Graphen mit $m = \Omega(n \log n)$ Kanten.

11.2 Minimale Spannbäume: Der Algorithmus von Jarník+Prim

Erinnerung (a) Ein (ungerichteter) Graph $G = (V, E)$ heißt **kreisfrei**, wenn er **keinen Kreis** besitzt.

(b) Ein Graph G heißt ein **freier Baum** (oder nur **Baum**), wenn er zusammenhängend und kreisfrei ist. – *Beispiel*:





Kreisfreie Graphen heißen auch **(freie) Wälder**.

Lemma 7.1.16 über Bäume

Wenn $G = (V, E)$ ein Graph mit n Knoten und m Kanten ist, dann sind folgende Aussagen äquivalent:

(a) G ist ein Baum.

(b) G ist kreisfrei und $m \geq n - 1$.

(c) G ist zusammenhängend und $m \leq n - 1$.

(d) Zu jedem Paar u, v von Knoten gibt es genau einen einfachen Weg von u nach v.

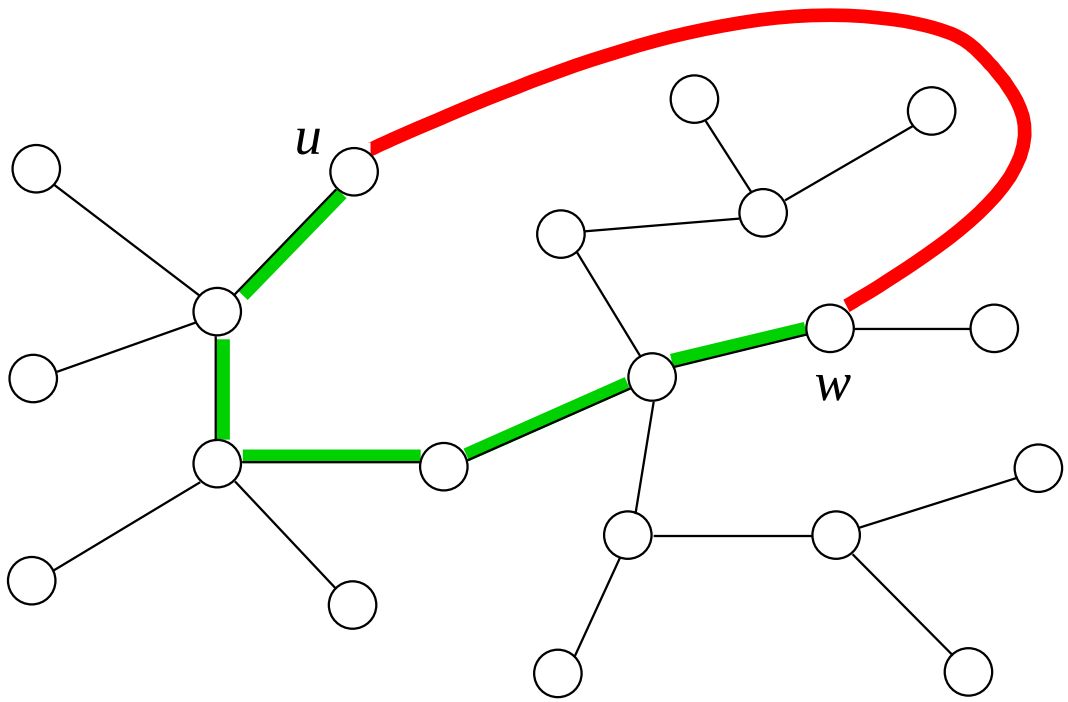
(e) G ist kreisfrei, aber das Hinzufügen einer beliebigen weiteren Kante erzeugt einen Kreis (G ist „maximal kreisfrei“).

(f) G ist zusammenhängend, aber das Entfernen einer beliebigen Kante erzeugt einen nicht zusammenhängenden Restgraphen (G ist „minimal zusammenhängend“).

Aus dem Fundamental-Lemma für Bäume folgt, für einen Baum G mit n Knoten:

(1) G hat $n - 1$ Kanten.

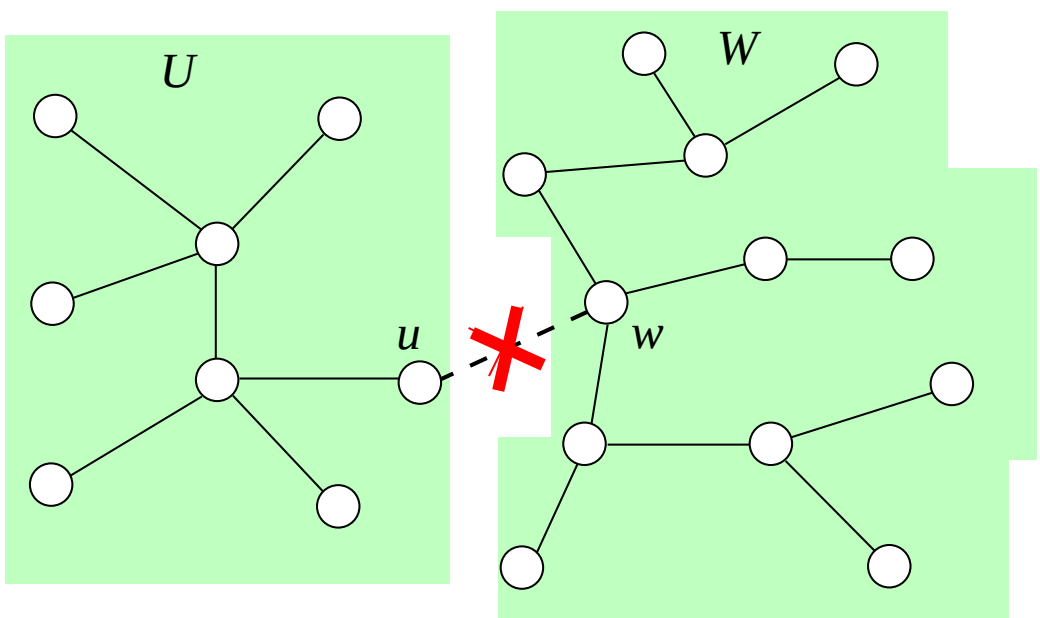
(2) Wenn man zu G eine Kante (u, w) hinzufügt, entsteht genau ein Kreis (aus (u, w) und dem eindeutigen **Weg** von u nach w in G).



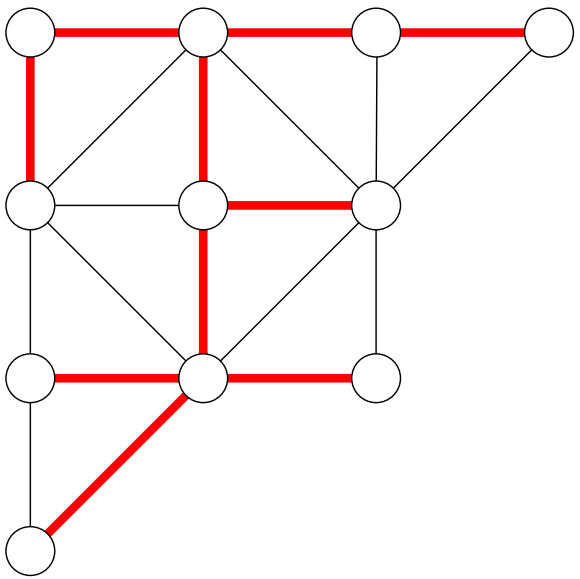
(3) Wenn man aus G eine Kante (u, w) streicht, zerfällt der Graph in 2 Komponenten

$U = \{v \in V \mid \ v$ von u aus über Kanten aus $E - \{(u, w)\}$ erreichbar$\}$;

$W = \{v \in V \mid \ v$ von w aus über Kanten aus $E - \{(u, w)\}$ erreichbar$\}$.



Beispiel:



Definition 11.2.1

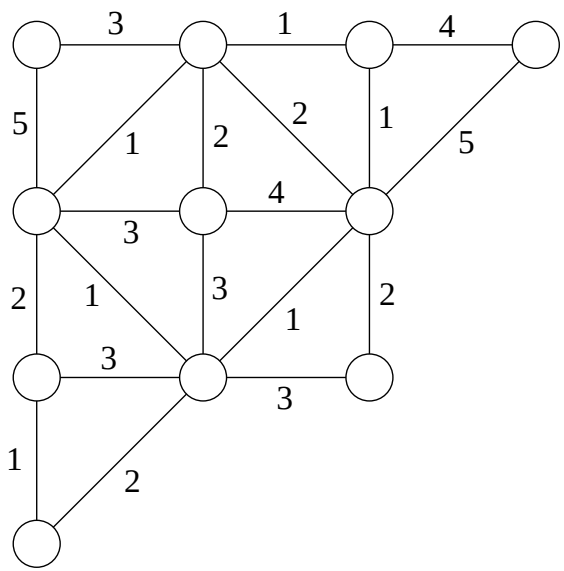
Es sei $G = (V, E)$ ein zusammenhängender Graph. Eine Menge $T \subseteq E$ von Kanten heißt ein **Spannbaum** für G, wenn (V, T) ein Baum ist.

Klar: Jeder zusammenhängende Graph hat einen Spannbaum.

(Iterativer Prozess: Starte mit E. Solange es Kreise gibt, entferne eine beliebige Kante auf einem Kreis. Der Zusammenhang bleibt stets erhalten; der verbleibende Graph muss kreisfrei sein.)

Definition 11.2.2

Es sei $G = (V, E, c)$ ein **gewichteter Graph**, d.h. $c\colon E \to \mathbb{R}$ ist eine „*Gewichtsfunktion*“ oder „*Kostenfunktion*“.



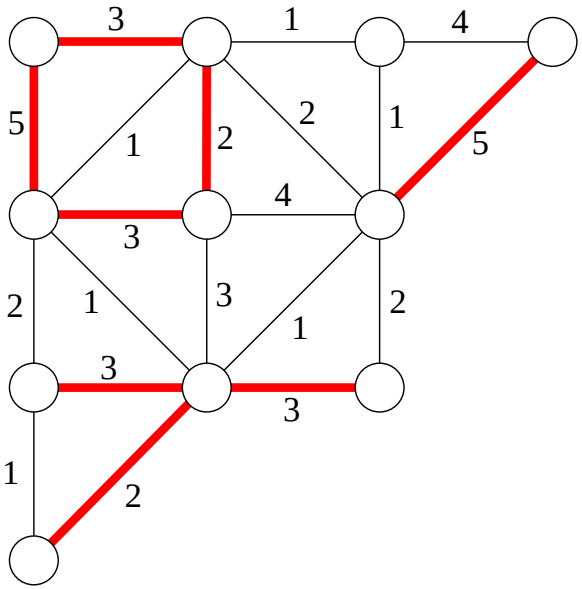
Graph modelliert: Straßennetz – Computernetzwerk – Leitungsnetz – . . . **Kantenkosten** modellieren **Herstellungskosten** (Straße, Unterwasserkabel, Pipeline, . . .) oder **Leitungsmiete** oder . . .

Ziel: Finde **Spannbaum** (zus.-hängend, kreisfrei) von G mit möglichst geringen Kosten.

(a) Jeder Kantenmenge $E' \subseteq E$ wird durch

$$c(E') := \sum_{e \in E'} c(e)$$

ein **Gesamtgewicht** zugeordnet.



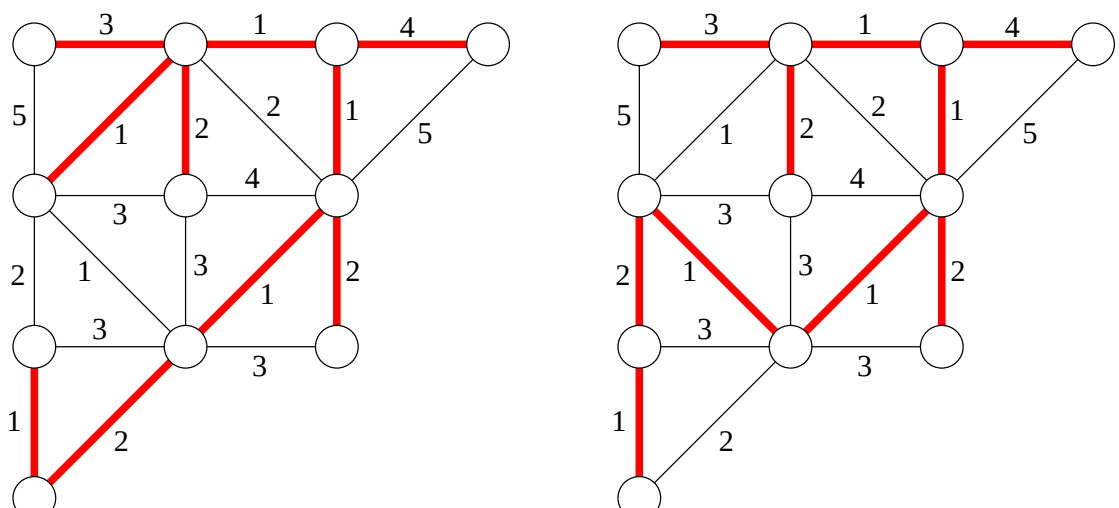
Gesamtgewicht

$c(\boldsymbol{E'}) = 3+5+2+5+3+3+3+2 = 26.$

(b) Sei G zusammenhängend. Ein Spannbaum $T \subseteq E$ für G heißt ein **minimaler Spannbaum**, wenn er minimale Kosten unter allen Spannbäumen hat, d. h. wenn

$$c(T) = \min\{c(T') \mid T' \text{ Spannbaum für } G\}.$$

Abkürzung: **MST** („**M**inimum **S**panning **T**ree“).



Zwei minimale Spannbäume, jeweils mit Gesamtgewicht 18.

Klar: Jeder Graph besitzt einen MST. (Es gibt nur endlich viele Spannbäume.)

Achtung: Es kann mehrere verschiedene MSTs geben (die alle dasselbe Gesamtgewicht haben).

Aufgabe: Zu gegebenem $G = (V, E, c)$ finde einen MST T.

Hier: **„Algorithmus von Jarník/Prim“** und **„Algorithmus von Kruskal“**

Algorithmenparadigma: **„greedy“**.

Baue Lösung **Schritt für Schritt** auf.

(Hier: Wähle eine Kante für T nach der anderen.)

Treffe in jedem Schritt die **(lokal) günstigste Entscheidung**.

Algorithmus von Jarník/Prim ist sehr ähnlich zum Algorithmus von Dijkstra.

Algorithmus von Jarník/Prim: (Programmierdetails später)

S: Menge von Knoten. Enthält die „bisher erreichten“ Knoten.
R: Menge von Kanten. Enthält die „bisher gewählten“ Kanten.

(1) Wähle einen beliebigen (Start-)Knoten $s \in V$.
$\mathtt{S} \leftarrow \{s\}$; $\mathtt{R} \leftarrow \emptyset$;

(2) Wiederhole $(n-1)$-mal:

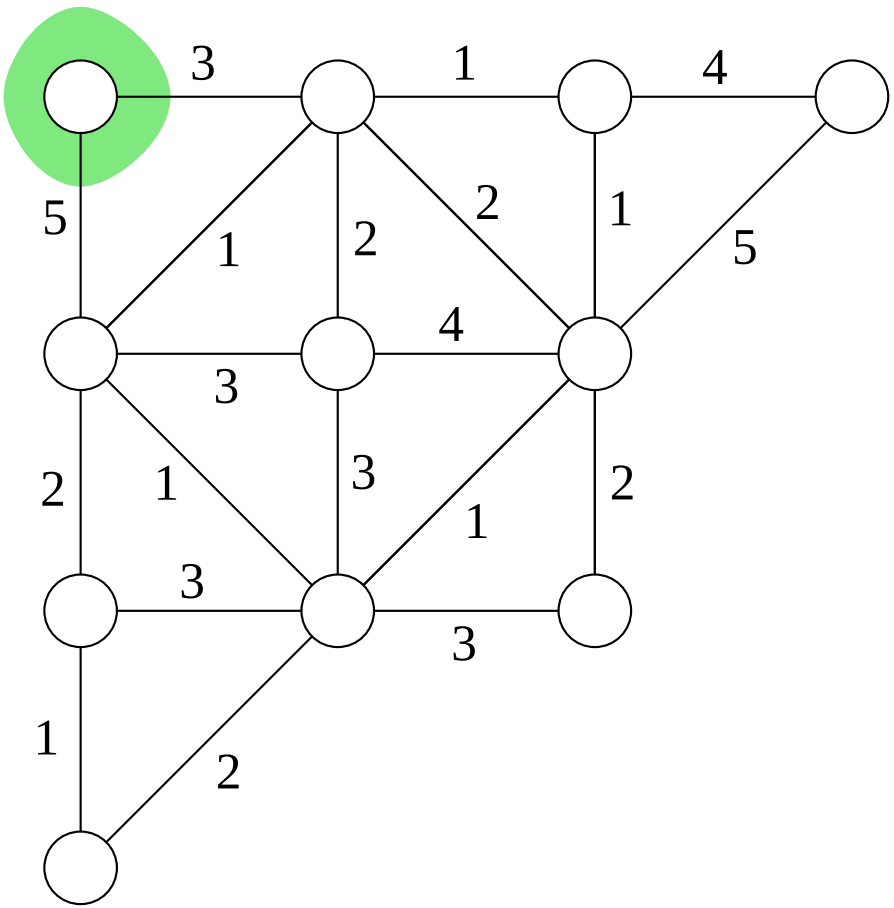
Wähle eine billigste Kante (v,u), die ein $v \in \mathtt{S}$ mit einem $u \in V - \mathtt{S}$ verbindet,

d. h. finde $v \in \mathtt{S}$ und $u \in V - \mathtt{S}$, so dass
$c(v,u)$ **minimal** unter allen Werten $c(v',u')$, $v' \in \mathtt{S}$, $u' \in V - \mathtt{S}$, ist.

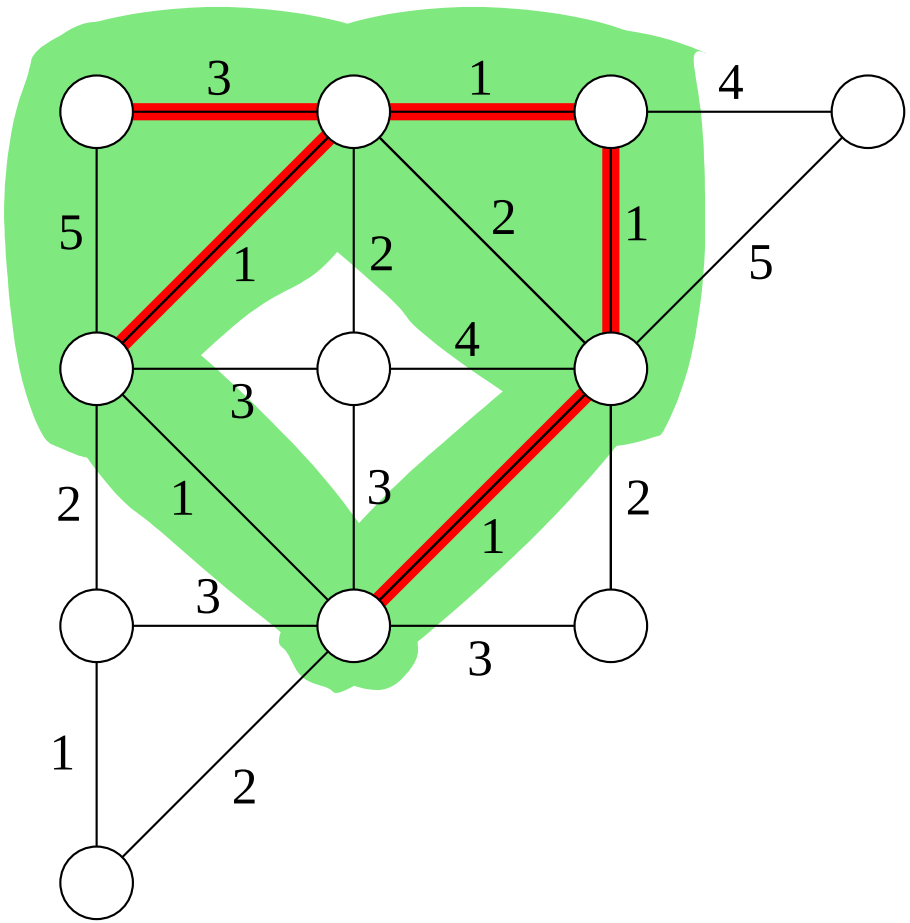
$\mathtt{S} \leftarrow \mathtt{S} \cup \{u\}$;
$\mathtt{R} \leftarrow \mathtt{R} \cup \{(v,u)\}$;

(3) Ausgabe: R.

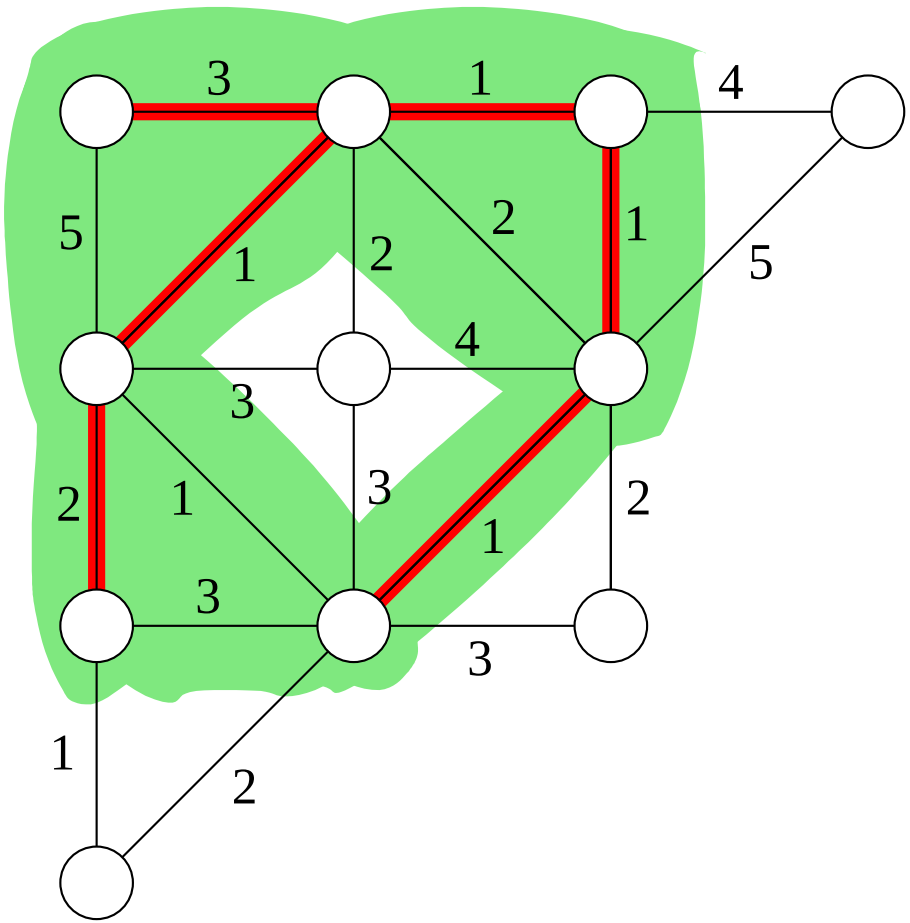
Beispiel (Jarník/Prim):



Beispiel (Jarník/Prim):



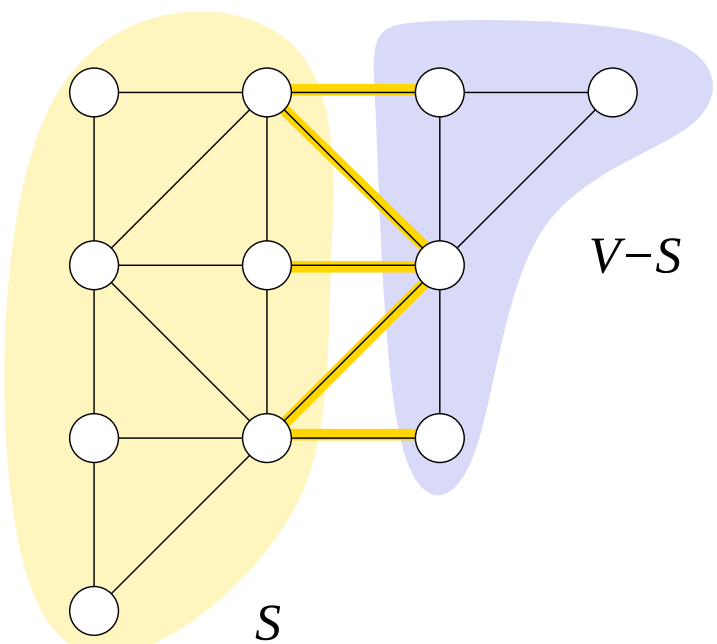
Beispiel (Jarník/Prim):



Die Schnitteigenschaft

Für den Korrektheitsbeweis des Algorithmus von Jarník/Prim:
„Cut property“ – Schnitteigenschaft

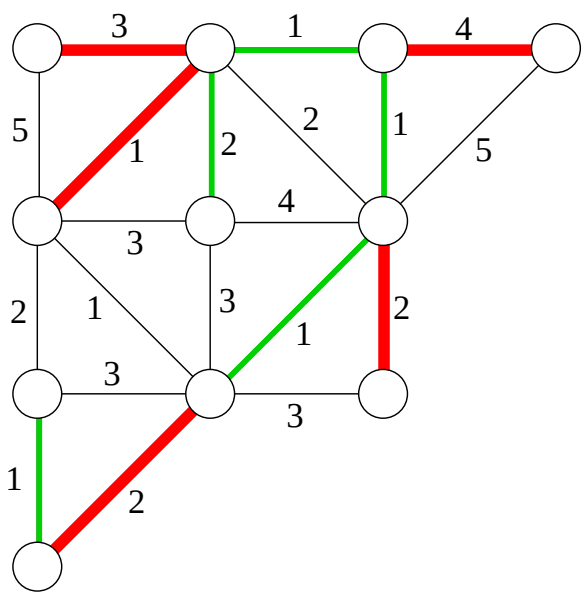
Eine Partition $(S, V - S)$ von V mit $\emptyset \neq S \neq V$ heißt ein **Schnitt**.



Die Schnitteigenschaft

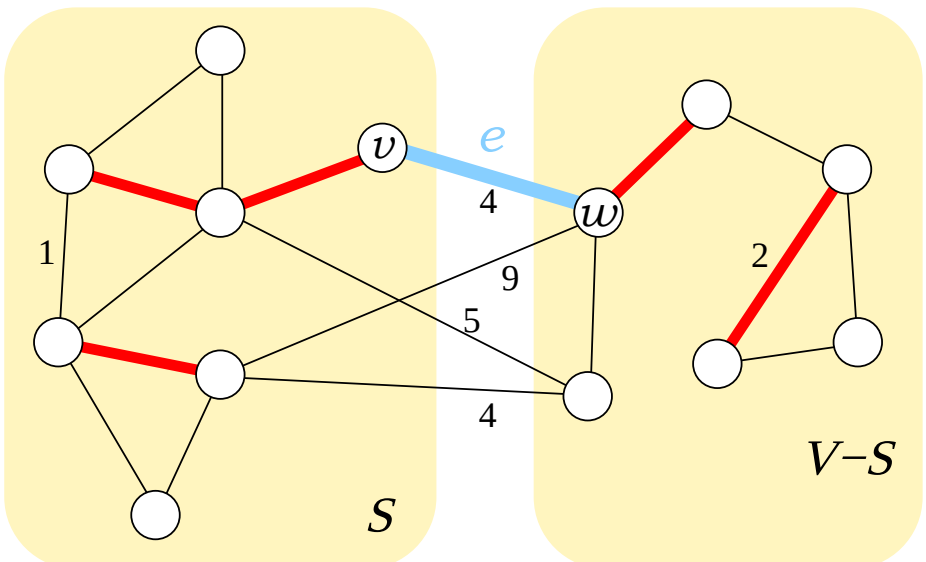
Definition 11.2.4

Eine Menge $R \subseteq E$ heißt **erweiterbar** (zu einem MST), wenn es einen MST T mit $R \subseteq T$ gibt.



$\boldsymbol{R}$ ist erweiterbar, denn es gibt einen MST $\boldsymbol{T} \supseteq \boldsymbol{R}$.

Die Schnitteigenschaft



Behauptung (Schnitteigenschaft):

Sei $\boldsymbol{R} \subseteq E$ erweiterbar **und** sei $(S, V - S)$ ein Schnitt, so dass es keine $\boldsymbol{R}$-Kante von S nach $V - S$ gibt,

und sei $\boldsymbol{e} = (v, w)$, $v \in S$, $w \in V - S$ eine Kante, die den Wert $c((v', w'))$, $v' \in S$, $w' \in V - S$, **minimiert**.

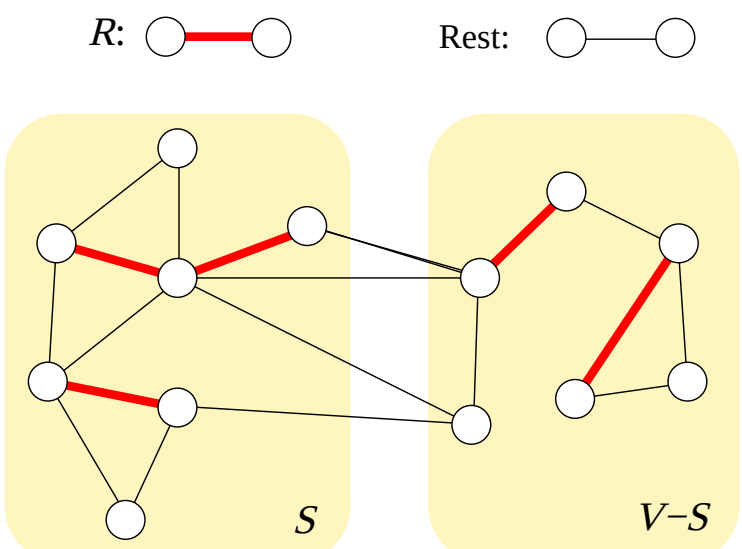
Dann ist auch $\boldsymbol{R} \cup \{\boldsymbol{e}\}$ erweiterbar.

Die Schnitteigenschaft

Beweis der Schnitteigenschaft: Sei $R \subseteq E$, sei $T \supseteq R$ ein MST;
sei $(S, V - S)$ ein Schnitt, e wie in der Voraussetzung.

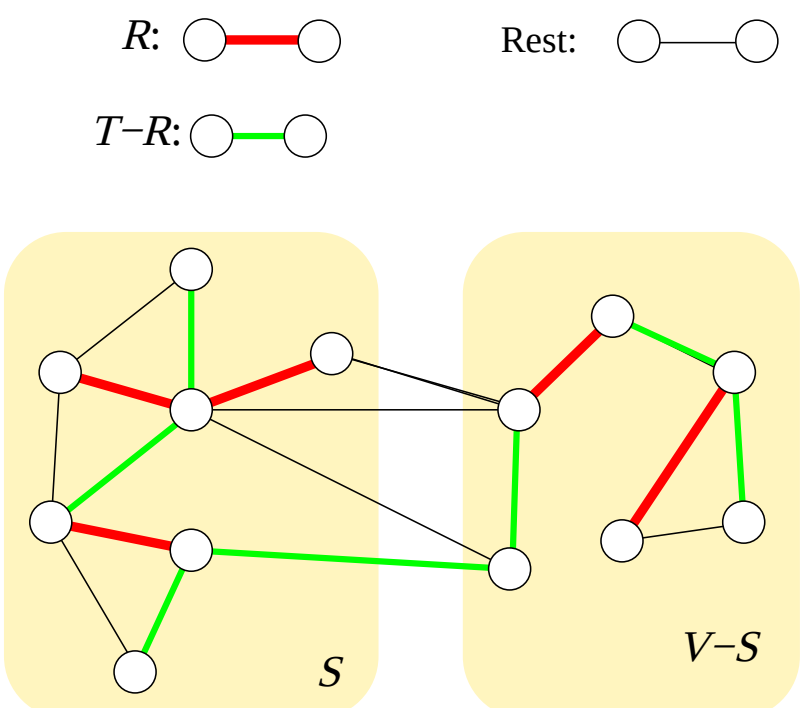
Wenn $e \in T$, dann gilt $R \cup \{e\} \subseteq T$, also ist $R \cup \{e\}$ erweiterbar.

Ab hier: $e \notin T$.



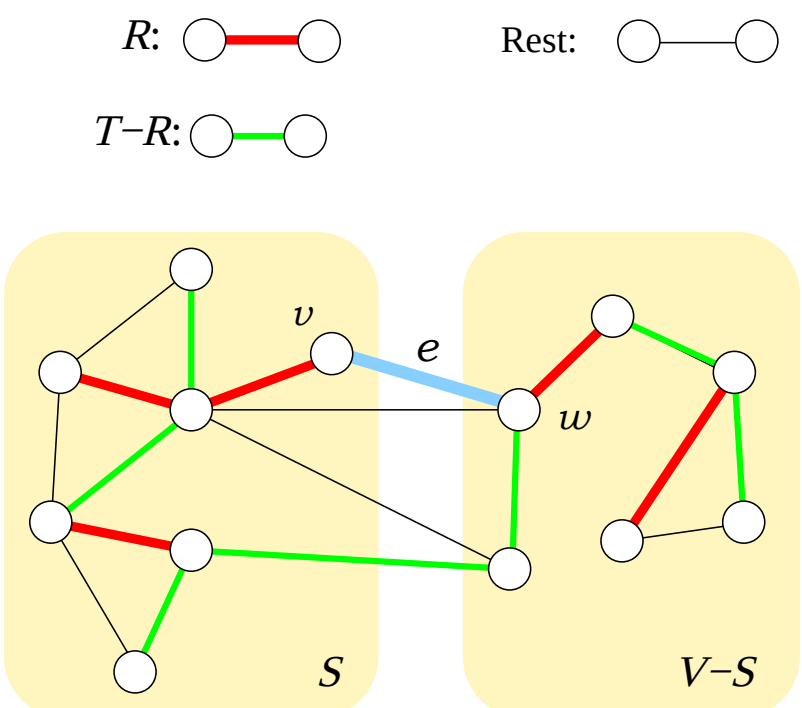
Eine erweiterbare Menge R.

Die Schnitteigenschaft



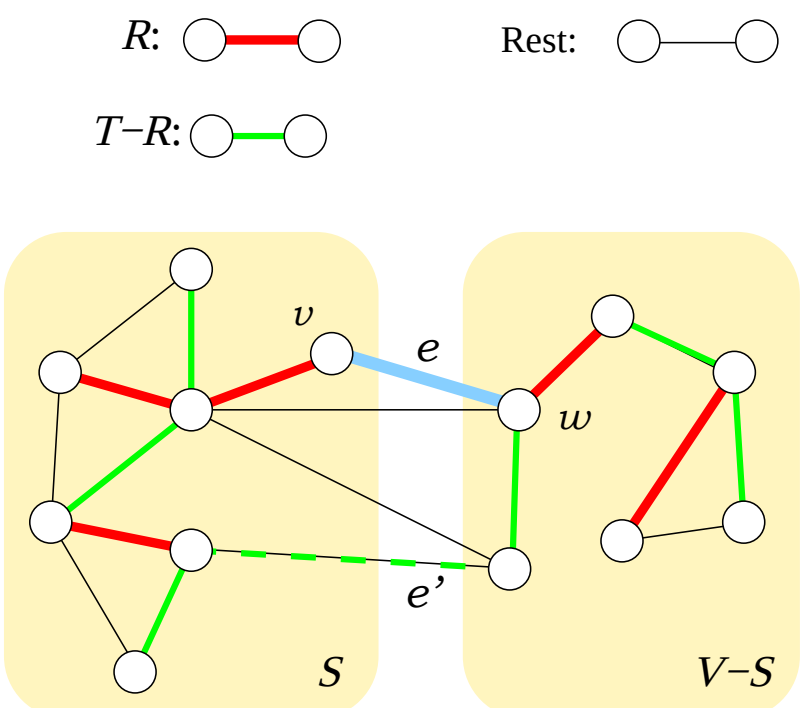
MST T mit $R \subseteq T$.

Die Schnitteigenschaft



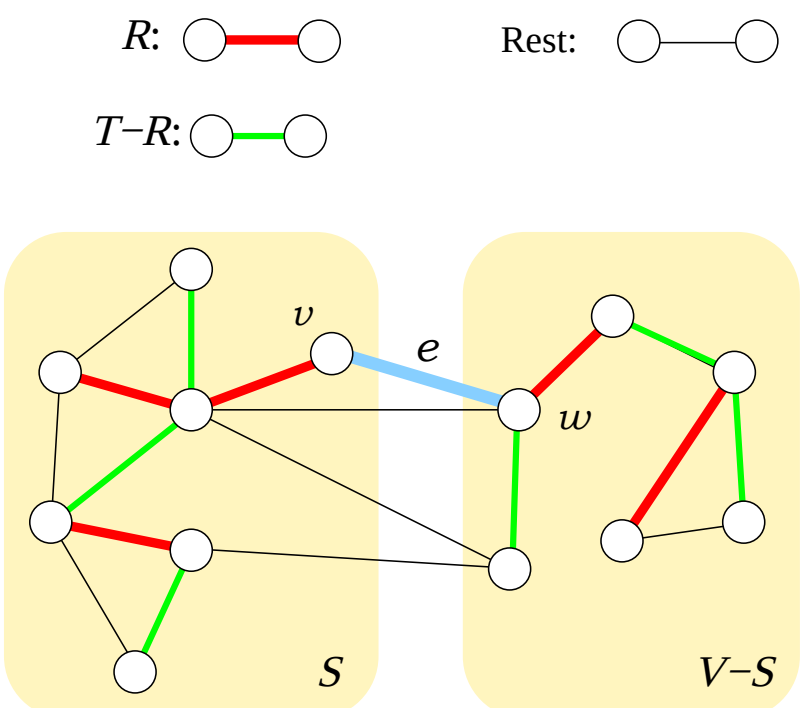
$e = (v, w)$ minimiert $c((v', w'))$, $v' \in S$, $w' \in V - S$.

Die Schnitteigenschaft



Weg in T von v nach w wechselt von S nach $V - S$ bei Kante e'.
Es entsteht ein Kreis in $T \cup \{e\}$, auf dem e und e' liegen.

Die Schnitteigenschaft



$T_e := (T - \{e'\}) \cup \{e\}$ ist Spannbaum und enthält $R \cup \{e\}$.
$c(T_e) - c(T) = c(e) - c(e') \leq 0$, also ist T_e optimal. Also ist $R \cup \{e\}$ erweiterbar.□

Korrektheit des Algorithmus von Jarník/Prim:

R_i: Kantenmenge (Größe i), die nach Runde i in R steht.
S_i: Knotenmenge (Größe $i+1$), die nach Runde i in S steht.

Weil in jeder Runde i ein neuer Knoten zu S_{i-1} hinzukommt, um S_i zu bilden, und die neue Kante in R_i diesen Knoten an S_{i-1} anschließt, ist jeder Graph (S_i, R_i) zusammenhängend. Da durch keine der neuen Kanten ein Kreis geschlossen wird, ist (S_i, R_i) sogar ein Baum.

Der Prozess kann nicht steckenbleiben:
Wenn es keine Kante von S_{i-1} nach $V - S_{i-1}$ gäbe, wäre G nicht zusammenhängend.

Zu zeigen: R_{n-1} ist ein minimaler Spannbaum für G.

Weil R_{n-1} kreisfrei ist und $n-1$ Kanten hat, ist R_{n-1} Spannbaum.

Induktionsbehauptung IB(i): R_i ist erweiterbar.

(Dies beweisen wir gleich durch Induktion über $i = 0, 1, \ldots, n-1$.)

Dann besagt IB$(n-1)$, dass es einen MST T mit $T \supseteq R_{n-1}$ gibt.

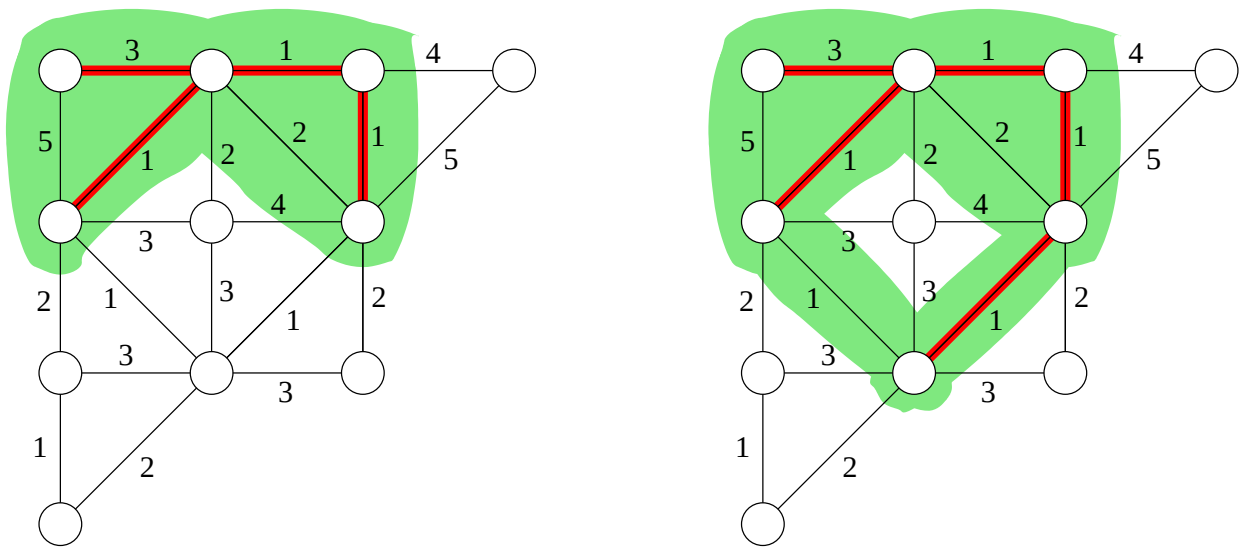
Weil T als Spannbaum $n-1$ Kanten hat und R_{n-1} auch $n-1$ Kanten hat, ist $T = R_{n-1}$, also ist die Ausgabe R_{n-1} ein MST.

IB(i): R_i ist erweiterbar.

I.A.: $i = 0$. – Es gibt einen MST T für G, und $R_0 = \emptyset \subseteq T$ gilt trivialerweise.

I.V.: $1 \leq i \leq n - 1$ und R_{i-1} ist erweiterbar.

I.S.: $(S_{i-1}, V - S_{i-1})$ ist ein Schnitt, und keine Kante von R_{i-1} überquert diesen Schnitt.



Der Algorithmus von Jarník/Prim wählt unter allen Kanten, die den Schnitt überqueren, eine billigste Kante e, und bildet $R_i := R_{i-1} \cup \{e\}$. Mit der Schnitteigenschaft folgt: R_i ist erweiterbar. □

Implementierungsdetails im Algorithmus von Jarník/Prim:

Wir nehmen an, dass $G = (V, E, c)$ mit $V = \{1, \ldots, n\}$ in Adjazenzlistendarstellung gegeben ist. Die Kantengewichte $c(e)$ stehen in den Adjazenzlisten bei den Kanten.

Für jeden Knoten $w \in V - \mathtt{S}$ wollen wir immer wissen:

1) die Länge der billigsten Kante (v, w), $v \in \mathtt{S}$, falls es eine gibt:
   in $\mathtt{dist}[w]$ („Abstand von S“), für Array $\mathtt{dist}[1..n]$.
2) den (einen) Knoten $p(w) \in \mathtt{S}$ mit $c(p(w), w) = \mathtt{dist}[w]$, falls ein solcher existiert:
   in $\mathtt{p}[w]$ („Vorgänger in S“), für Array $\mathtt{p}[1..n]$.

Solange es von S keine Kante nach w gibt, gilt $\mathtt{dist}[w] = \infty$ und $\mathtt{p}[w] = -1$.

Verwalte die Knoten $w \in V - \mathtt{S}$ mit Werten $\mathtt{dist}[w] < \infty$ mit den $\mathtt{dist}[w]$-Werten als **Schlüssel** in einer **Prioritätswarteschlange** **PQ**.

Wenn $\mathtt{dist}[w] = \infty$, ist w (noch) nicht in der **PQ**.

Jarnik/Prim(G, s) // (Vollversion mit Prioritätswarteschlange)

Eingabe: gewichteter **Graph** $G = (V, E, c)$, $V = \{1, \ldots, n\}$, Startknoten $s \in V$ (ist beliebig);

Ausgabe: Ein MST für G.

Hilfsstrukturen: PQ: eine (anfangs leere) Prioritäts-WS; `inS`, `p`: wie oben

```
(1)   for w from 1 to n do
(2)     dist[w] ← ∞; inS[w] ← false; p[w] ← −1;
(3)   dist[s] ← 0; p[s] ← −2; PQ.insert(s);
(4)   while not PQ.isempty do
(5)     u ← PQ.extractMin; inS[u] ← true;
(6)     for Knoten w mit (u, w) ∈ E and not inS[w] do
(7)       dd ← c(u, w);    // einziger Unterschied zu Dijkstra!
(8)       if p[w] ≥ 0 and dd < dist[w] then
(9)         PQ.decreaseKey(w,dd); p[w] ← u; dist[w] ← dd;
(10)      if p[w] = −1 then   // w vorher nicht zu S benachbart
(11)        dist[w] ← dd; p[w] ← u; PQ.insert(w);
(12)  Ausgabe: T = {(w, p[w]) | inS[w] = true, w ≠ s}.   // Menge der gewählten Kanten
```

Zeitanalyse: Exakt wie für den Dijkstra-Algorithmus.

Satz 11.2.5

Der Algorithmus von Jarník/Prim mit Verwendung einer Prioritätswarteschlange, die als Binärheap realisiert ist, ermittelt einen minimalen Spannbaum für $G = (V, E, c)$ in Zeit $\boldsymbol{O((n+m)\log n)}$.

Wie beim Algorithmus von Dijkstra: Wenn man Fibonacci-Heaps o. ä. verwendet, bei denen m decreaseKey-Operationen Zeit $O(m)$ benötigen:
Rechenzeit für Jarník/Prim: $\boldsymbol{O(m + n\log n)}$.

11.3 Der Algorithmus von Kruskal

Auch dieser Algorithmus löst das **MST-Problem**.

Anderer Ansatz als bei Jarník/Prim, aber auch „greedy“:
Starte mit $\boldsymbol{R} = \emptyset$. Dann folgen $n - 1$ Runden.

In jeder Runde:

> Wähle eine Kante $\boldsymbol{e} \in E - \boldsymbol{R}$ von kleinstem Gewicht,
> die mit $(V, \boldsymbol{R})$ keinen Kreis schließt, und füge $\boldsymbol{e}$ zu $\boldsymbol{R}$ hinzu.

Eine offensichtlich korrekte Methode, dies zu organisieren:

> Durchmustere Kanten in **aufsteigender Reihenfolge** des Kantengewichts, und nimm eine Kante genau dann in $\boldsymbol{R}$ auf, wenn sie mit $\boldsymbol{R}$ keinen Kreis bildet.

Algorithmus von Kruskal, informal

1. Schritt:
Sortiere die Kanten $e_1, \ldots, e_m$ nach den Gewichten $c(e_1), \ldots, c(e_m)$ aufsteigend.
Also o.B.d.A.: $c(e_1) \leq \ldots \leq c(e_m)$.

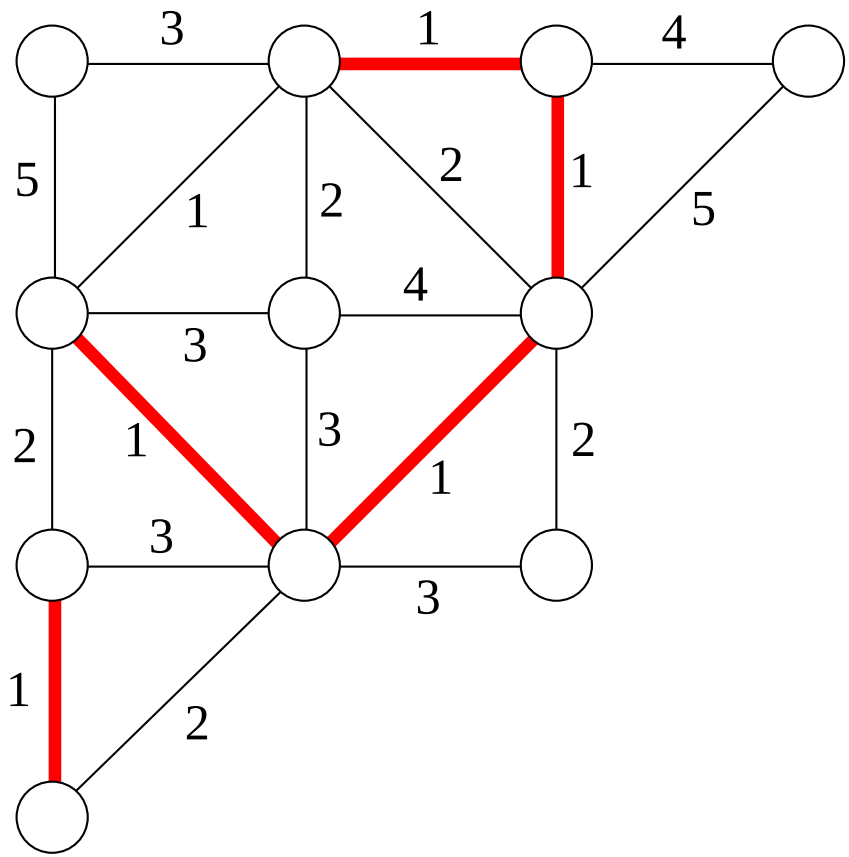
2. Schritt: Setze $\mathtt{R} \leftarrow \emptyset$.

3. Schritt: Für $i = 1, 2, \ldots, m$ tue folgendes:
Falls $\mathtt{R} \cup \{e_i\}$ kreisfrei ist, setze $\mathtt{R} \leftarrow \mathtt{R} \cup \{e_i\}$
// sonst, d.h. wenn e_i einen Kreis schließt, bleibt $\mathtt{R}$ unverändert

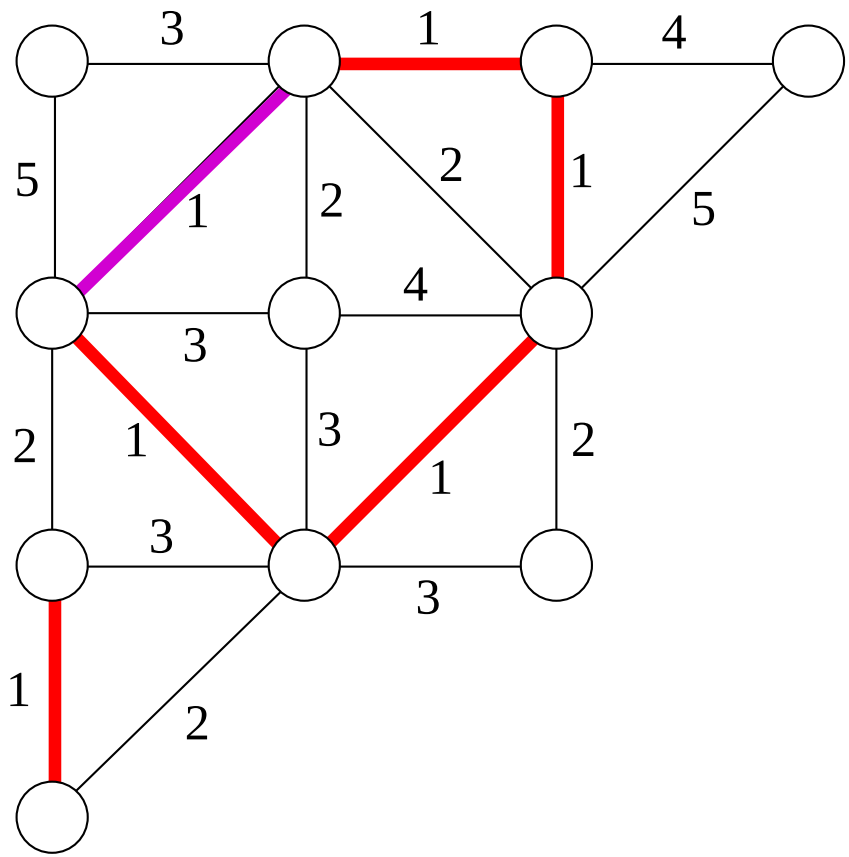
// Optional: Beende Schleife, wenn $|\mathtt{R}| = n - 1$.

4. Schritt: Die Ausgabe ist $\mathtt{R}$.

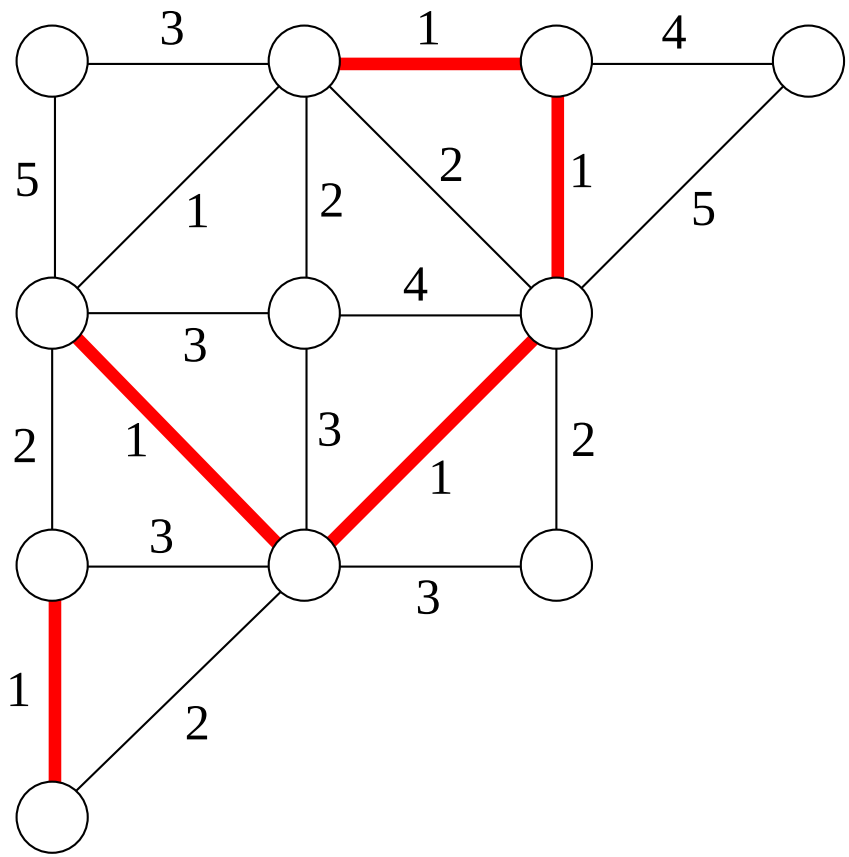
Beispiel (Kruskal):



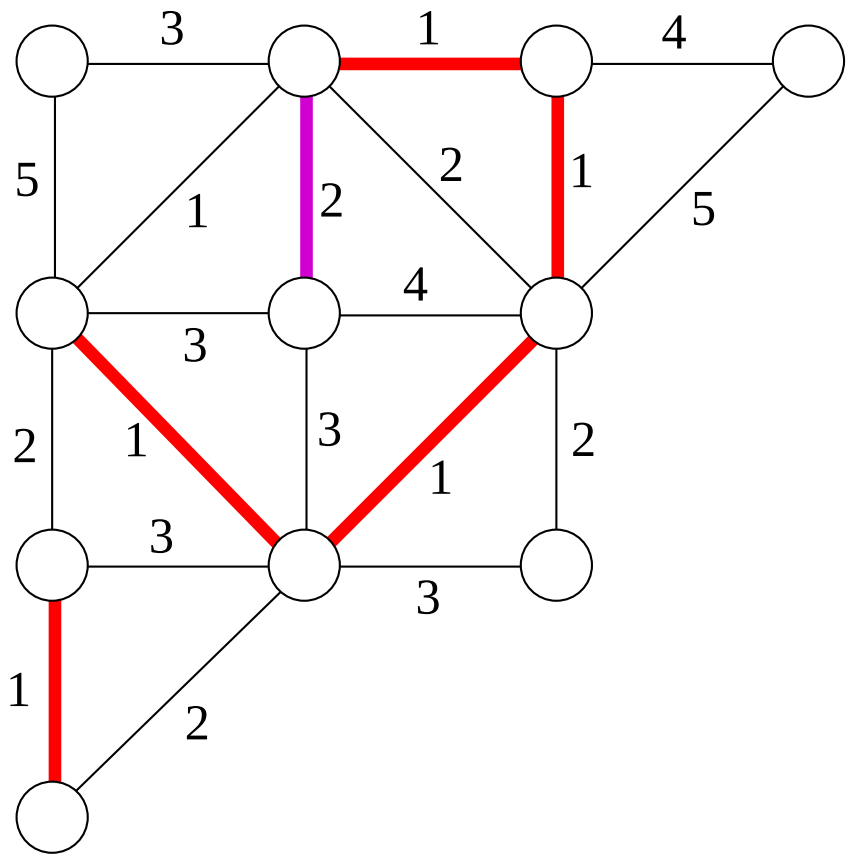
Beispiel (Kruskal):



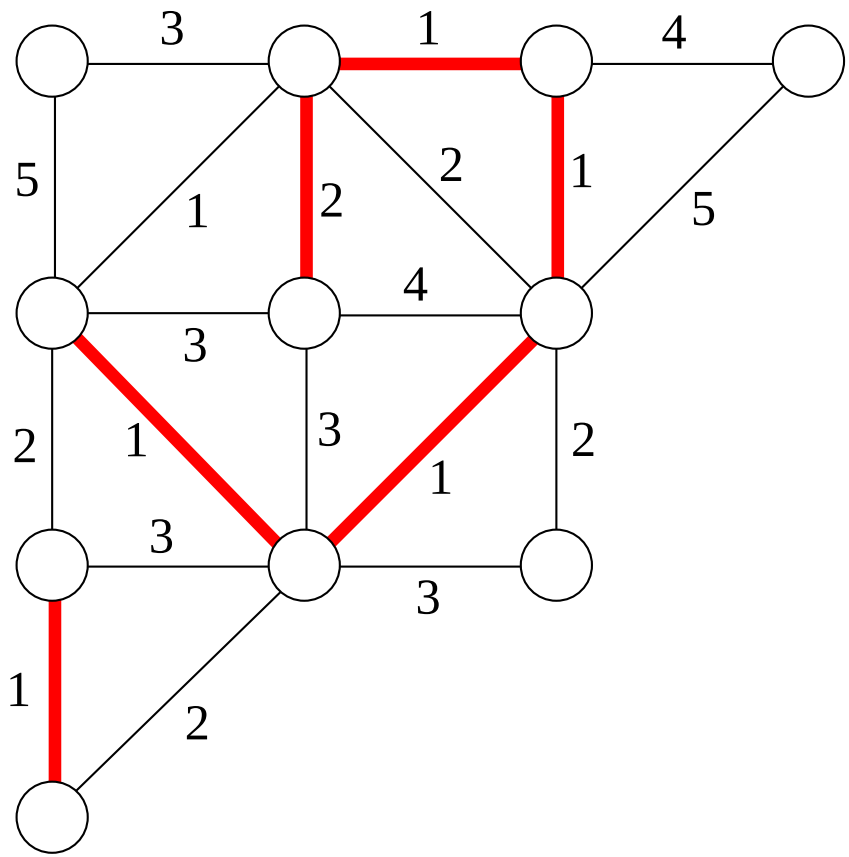
Beispiel (Kruskal):



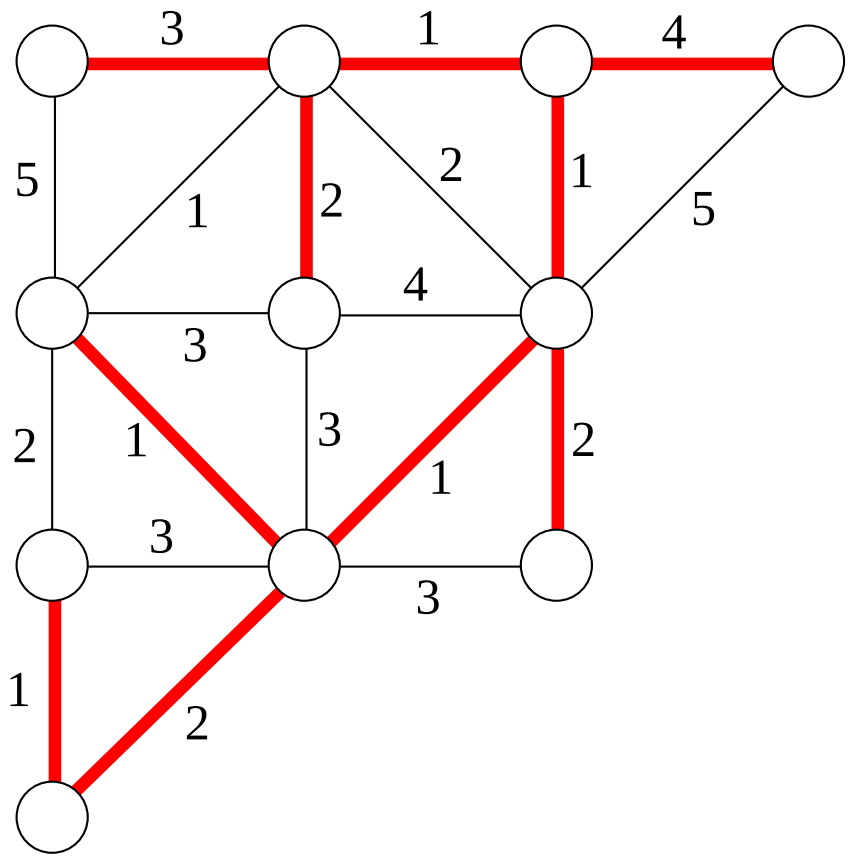
Beispiel (Kruskal):



Beispiel (Kruskal):



Beispiel (Kruskal):



Uns interessieren: 1) Korrektheit; 2) Rechenzeit (später).

Korrektheit des Algorithmus von Kruskal:

R_i sei die Kantenmenge, die nach Runde i, der Bearbeitung von e_i, in R steht.

Zu zeigen ist: R_m ist ein minimaler Spannbaum für G.

Erinnerung: $R \subseteq E$ heißt **erweiterbar**, wenn $R \subseteq T$ gilt, für einen MST T.

Ind.-Behauptung IB(i), $0 \leq i \leq m$**:** R_i ist erweiterbar. (Beweis per Induktion über i.)

Dann besagt IB(m), dass $R_m \subseteq T$ ist, für einen MST T.

Beh.: Es gilt auch $T \subseteq R_m$. (Also gilt Gleichheit, und R_m ist ein MST.)

(*Beweis* der Beh.: Sei $e \in T$. Dann ist $e = e_i$ für ein i, und e_i wurde in Runde i getestet. Weil $R_{i-1} \subseteq R_m \subseteq T$ und $e_i \in T$, ist $R_{i-1} \cup \{e_i\} \subseteq T$, also ist $R_{i-1} \cup \{e_i\}$ kreisfrei, also fügt der Algorithmus die Kante e_i in Runde i zu R hinzu, also gilt $e_i \in R_m$.)

Induktionsbehauptung IB(i): R_i ist erweiterbar.

Beweis, durch Induktion über i:

I.A.: $R_0 = \emptyset$ ist erweiterbar.

I.V.: $1 \leq i \leq m$ und R_{i-1} ist erweiterbar.

I.S.: Nun wird Runde i mit Kante e_i ausgeführt.

1. Fall: $R_{i-1} \cup \{e_i\}$ enthält einen Kreis. Dann ist $R_i = R_{i-1}$, also erweiterbar nach I.V.

2. Fall: $R_{i-1} \cup \{e_i\}$ ist kreisfrei. – Sei $e_i = (v, w)$. Definiere
$S := \{u \in V \mid \text{im Wald } (V, R_{i-1}) \text{ ist } u \text{ von } v \text{ aus erreichbar}\}$.
(S ist die Zusammenhangskomponente von v in (V, R_{i-1}).)
Weil $R_{i-1} \cup \{(v, w)\}$ kreisfrei ist, folgt $w \in V - S$.
Nach Definition von S ist klar, dass keine R_{i-1}-Kante S und $V - S$ verbindet.
Betrachte nun eine beliebige Kante $e = e_j$, die zwischen S und $V - S$ verläuft. Wie eben gesagt, ist dann $e_j \notin R_{i-1}$. Es gilt sogar $j \geq i$. (e_j schließt mit R_{i-1} keinen Kreis, also erst recht nicht mit $R_{j-1} \subseteq R_{i-1}$. Wäre $j < i$, hätte der Algorithmus e_j in Runde j zu R hinzugefügt.)
Weil wir die Kanten anfangs nach Gewicht sortiert haben, folgt $c(e_j) \geq c(e_i)$.
Also ist $c(e_i)$ minimal unter allen $c((v', w'))$ mit $(v', w') \in E$, $v' \in S$, $w' \in V - S$.
Nach der **Schnitteigenschaft** folgt: $R_i = R_{i-1} \cup \{e_i\}$ ist erweiterbar, d. h. die Induktionsbehauptung.

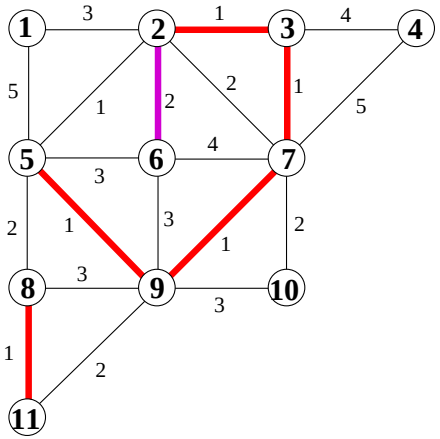
Damit ist der Korrektheitsbeweis für den Algorithmus von Kruskal vollständig. □

11.4 Hilfsstruktur: Union-Find

Union-Find-Datenstrukturen dienen als **Hilfsstruktur** für verschiedene Algorithmen, insbesondere für den Algorithmus von Kruskal.

Zwischenkonfiguration im Algorithmus von Kruskal: Menge $R_{i-1} \subseteq E$, die Wald bilden, und Folge $e_i, \ldots, e_m$ von noch zu verarbeitenden Kanten.

11.4.1 Algorithmus von Kruskal mit Union-Find



Die im Algorithmus von Kruskal zu lösende Aufgabe:
Zu zwei Knoten v und w entscheide, ob es in (V, R_{i-1}) einen Weg von v nach w gibt.

(Hier: zwischen den Endpunkten 2 und 6 der violetten Kante.)

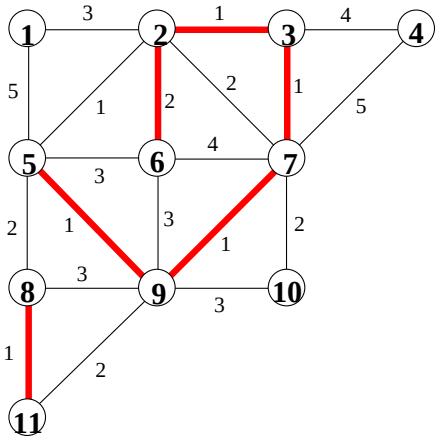
Möglich, aber ungeschickt: Jedesmal Tiefensuche o. ä.

Ansatz: Repräsentiere die **Knotenmengen**, die den Zusammenhangskomponenten von $(V, \boldsymbol{R})$ entsprechen, in einer Datenstruktur. – Im Beispielbild: $\{1\}, \{2, 3, 5, 7, 9\}, \{4\}, \{6\}, \{8, 11\}, \{10\}$.

Es soll **schnell zu ermitteln** sein, ob **zwei Knoten in derselben Komponente/Menge liegen**.

Wenn wir einen Schritt des Kruskal-Algorithmus ausführen, bei dem eine Kante akzeptiert, also in R aufgenommen wird, müssen wir zwei der disjunkten Mengen **vereinigen**.

11.4.1 Algorithmus von Kruskal mit Union-Find



Die im Algorithmus von Kruskal zu lösende Aufgabe:
Zu zwei Knoten v und w entscheide, ob es in (V, R_{i-1}) einen Weg von v nach w gibt.

(Hier: zwischen den Endpunkten 2 und 6 der violetten Kante.)

Möglich, aber ungeschickt: Jedesmal Tiefensuche o. ä.

Ansatz: Repräsentiere die **Knotenmengen**, die den Zusammenhangskomponenten von $(V, \boldsymbol{R})$ entsprechen, in einer Datenstruktur. – Im Beispielbild: $\{1\}, \{2, 3, 5, 7, 9\}, \{4\}, \{6\}, \{8, 11\}, \{10\}$.

Es soll **schnell zu ermitteln** sein, ob **zwei Knoten in derselben Komponente/Menge liegen**.

Wenn wir einen Schritt des Kruskal-Algorithmus ausführen, bei dem eine Kante akzeptiert, also in R aufgenommen wird, müssen wir zwei der disjunkten Mengen **vereinigen**.

Neue Mengen: $\{1\}, \{2, 3, 5, 6, 7, 9\}, \{4\}, \{8, 11\}, \{10\}$.
Auch diese Operation sollte schnell durchführbar sein.

Eine **Partition*** von $\{1, 2, \ldots, n\}$ ist eine **Zerlegung** in Mengen (hier: „Klassen“)

$$\{1, 2, \ldots, n\} = S_1 \cup S_2 \cup \cdots \cup S_\ell,$$

wobei $S_1, S_2, \ldots, S_\ell$ **disjunkt** sind.

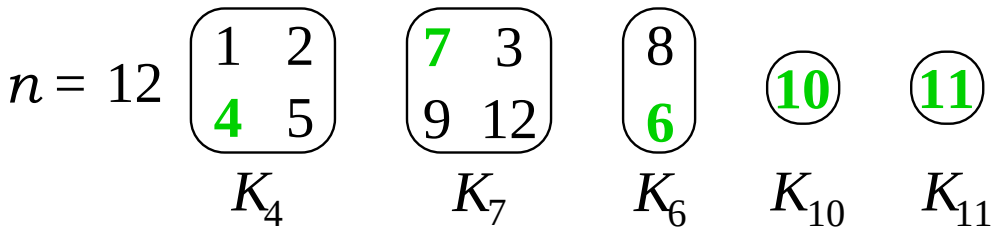
Abstrakte Aufgabe:

Verwalte eine **dynamische** (d.h. veränderliche) **Partition** der Menge $\{1, 2, \ldots, n\}$ unter Operationen

init	(Initialisierung)
union	(Vereinigung von Klassen der Partition)
find	(„In welcher Klasse liegt i?“).

* In der Mathematik heißt die gesamte Aufteilung Partition, die Teile Klassen.
Bei der Speicheraufteilung in Computern bedeutet „Partition“ einen Teil. Nicht verwechseln!

Beispiel: $n = 12$.



In jeder Klasse K der Partition ist ein **Repräsentant** $r \in K$ ausgezeichnet. Dieser fungiert als **Name** von K. Wir schreiben $\boldsymbol{K_r}$ für die Klasse mit Repräsentanten r.

Kompakte Darstellung:
Funktion/Array r mit $r(i) =$ Repräsentant der Klasse von i:

i	1	2	3	4	5	6	7	8	9	10	11	12
$r(i)$	4	4	7	4	4	6	7	6	7	10	11	7

Operationen:

init(n): Erzeugt zu $n \geq 1$ die „diskrete Partition“ mit den n einelementigen Klassen $\{1\}, \{2\}, \ldots, \{n\}$, also $K_i = \{i\}$.

find(i): Gibt zu $i \in \{1, \ldots, n\}$ den Namen $r(i)$ der Klasse $K_{r(i)}$ aus, in der sich i (gegenwärtig) befindet.

union(s, t): Die Argumente s und t müssen Repräsentanten **verschiedener Klassen** K_s bzw. K_t sein. Die Operation ersetzt in der Partition K_s und K_t durch die Vereinigung $K_s \cup K_t$.
Als Repräsentant von $K_s \cup K_t$ kann ein beliebiges Element verwendet werden. (Meistens: s oder t.)

Im Beispiel entfernt **union**$(4, 10)$ die Klassen $K_4 = \{1, 2, \mathbf{4}, 5\}$ und $K_{10} = \{\mathbf{10}\}$ und fügt $K'_{10} = \{1, 2, 4, 5, \mathbf{10}\}$ hinzu.

Aus Sicht der r-Funktion entspricht diese Operation der Änderung der Funktionswerte auf der Klasse K_s:

$$r'(i) := \begin{cases} t & , \text{ falls } r(i) = s, \\ r(i) & , \text{ sonst.} \end{cases}$$

Im Beispiel: Die r-Werte in K_4 werden auf 10 geändert, Rest bleibt gleich.

i	1	2	3	4	5	6	7	8	9	10	11	12
$r'(i)$	**10**	**10**	7	**10**	**10**	6	7	6	7	10	11	7

Zwei Implementierungsmöglichkeiten:

1) **Arrays (mit Listen)** (erlaubt schnelle **find**s)
2) **Bäume** (erlaubt schnelle **union**s)

Algorithmus von Kruskal mit Union-Find

Input: Gewichteter zusammenhängender Graph $G = (V, E, c)$ mit $V = \{1, \ldots, n\}$.

1. Schritt:
Sortiere Kanten $e_1, \ldots, e_m$ nach Gewichten $c_1 = c(e_1), \ldots, c_m = c(e_m)$ aufsteigend.

Resultat: Kantenliste $(v_1, w_1, c_1), \ldots, (v_m, w_m, c_m)$, $c_1 \leq \cdots \leq c_m$.

2. Schritt: Initialisiere Union-Find-Struktur für $\{1, \ldots, n\}$.

3. Schritt: Für $i = 1, 2, \ldots, m$ tue folgendes:

$\mathtt{s} \leftarrow$ **find**(v_i); $\mathtt{t} \leftarrow$ **find**(w_i);
if $\mathtt{s} \neq \mathtt{t}$ **then begin** $\mathtt{R} \leftarrow \mathtt{R} \cup \{e_i\}$; **union**$(\mathtt{s}, \mathtt{t})$ **end**;
// Optional: Beende Schleife, wenn $|\mathtt{R}| = n - 1$.

4. Schritt: Die Ausgabe ist R.

Bemerkung: Im Algorithmus kommt das Wort „Kreis“ nicht mehr vor.
Es wird vielmehr getestet, ob (v_i, w_i) zwei Zusammenhangskomponenten verbindet.

Satz 11.4.1

(a) Der Algorithmus von Kruskal in der Implementierung mit Union-Find ist korrekt.

(b) Die Rechenzeit des Algorithmus ist $O(m \log n)$, wenn man die Union-Find-Datenstruktur mit **Arrays** implementiert.

(c) Die Rechenzeit des Algorithmus ist $O(m \log n)$, wenn man die Union-Find-Datenstruktur mit mit **wurzelgerichteten Bäumen** (mit **Pfadkompression**) implementiert.

Bem.: Teil (c) wird noch präzisiert.

Beweis: (a) Man zeigt durch Induktion über die Runden, dass nach Runde i die Klassen der Union-Find-Struktur genau die Zusammenhangskomponenten des Graphen (Waldes) (V, R_i) sind, für $R_i =$ Inhalt von `R` nach Runde i.

Daher testet „`s` $\leftarrow$ **find**(v_i); `t` $\leftarrow$ **find**(w_i); **if** `s` $\neq$ `t` . . . “ im 3. Schritt korrekt die Kreiseigenschaft.

(b), (c): Siehe unten.

11.4.2 Arrayimplementierung von Union-Find

Array `r[1..n]` enthält in `r[i]` den Repräsentanten $r(i)$ der Klasse $K_{r(i)}$, in der i **gegenwärtig** liegt. In unserem Beispiel sähe `r` folgendermaßen aus:

	1	2	3	4	5	6	7	8	9	10	11	12
`r`:	4	4	7	4	4	6	7	6	7	10	11	7

Offensichtlich ist **find**(i) in Zeit $O(1)$ ausführbar.

Naiver Ansatz für **union**(s, t):

Durchlaufe das Array `r`; ändere dabei alle „s“ in „t“ (oder umgekehrt).

Kosten: $\Theta(n)$.

Dies kann man verbessern.

Trick 1:

Halte die Elemente jeder Klasse K_s (mit Repräsentant s) in einer linearen Liste L_s.

Dann muss man bei der Umbenennung von „s“ in „t“ nur L_s durchlaufen – die Kosten betragen $\Theta(|K_s|)$. Aus Listen L_s und L_t wird eine neue Liste L'_t.

Trick 2:

Bei der Vereinigung von Klassen sollte man den Repräsentanten der **größeren** Klasse übernehmen und den der **kleineren** ändern, da damit die Zahl der Änderungen kleiner gehalten wird. – Hierzu muss man die **Listenlängen/Klassengrößen** in einem zweiten Array `size`$[1..n]$ mitführen.

Im Beispiel:

	1	2	3	4	5	6	7	8	9	10	11	12
`r`:	4	4	7	4	4	6	7	6	7	10	11	7

	1	2	3	4	5	6	7	8	9	10	11	12
`size`:	–	–	–	4	–	2	4	–	–	1	1	–

Nur die Einträge `size[`s`]` mit $r(s) = s$, also für Repräsentanten s, sind relevant. Die anderen Einträge (mit „$-$“ gekennzeichnet) sind unwesentlich.

Listen: $L_4 = (4, 2, 1, 5)$, $L_7 = (7, 3, 12, 9)$, $L_6 = (6, 8)$, $L_{10} = (10)$, $L_{11} = (11)$.

Für eine besonders sparsame Implementierung der Listen ist es nützlich, wenn L_s mit s beginnt.

Beispiel: **union(7, 6)**.

Weil `size[6]` $<$ `size[7]`, werden die Einträge `r[`i`]` für i in L_6 auf **7** geändert und $L_6 = (\mathbf{6}, \mathbf{8})$ wird unmittelbar nach dem ersten Element **7** in L_7 eingefügt:

	1	2	3	4	5	6	7	8	9	10	11	12
`r`:	4	4	7	4	4	7	7	7	7	10	11	7

	1	2	3	4	5	6	7	8	9	10	11	12
`size`:	–	–	–	4	–	–	6	–	–	1	1	–

Aus $L_7 = (7, 3, 12, 9)$ und $L_6 = (6, 8)$ wird die neue Liste $L_7 = (7, \mathbf{6}, \mathbf{8}, 3, 12, 9)$.

Zeitaufwand: Θ(Länge der kleineren Liste) für die Umbenennungen im Array `r` und $O(1)$ für das Ändern des Eintrags `size`$[t]$ sowie das Kombinieren der Listen.

Zur Effizienzverbesserung (um konstanten Faktor) und zur Platzersparnis gibt es noch einen Trick.
Da alle Listen zusammen stets die Einträge $1, 2, \ldots, n$ haben, brauchen wir keine dynamisch erzeugten Listenelemente, sondern können alle Listen kompakt in *einem* Array speichern.

`next`$[1..n]$`: integer` mit

$$\texttt{next}[i] = \begin{cases} j, & \text{falls } j \text{ Nachfolger von } i \text{ in einer der Listen } L_s, \\ 0, & \text{falls } j \textbf{ letzter} \text{ Eintrag in seiner Liste } L_{r(j)}. \end{cases}$$

Beispiel:

	1	2	3	4	5	6	7	8	9	10	11	12
`next:`	–	$\cdots$	12	$\cdots$	$\cdots$	8	6	3	0	$\cdots$	$\cdots$	9

Darstellung von $L_7 = (7, 6, 8, 3, 12, 9)$. (Hier sieht man, wieso L_s mit s beginnen muss.)

Implementierung der Operationen im Detail:

Prozedur init(n) // Initialisierung einer Union-Find-Struktur

(1) Erzeuge `r, size, next`: Arrays der Länge n für `int`-Einträge
(2) **for** `i` **from** 1 **to** n **do**
(3) `r[i]` $\leftarrow$ `i`;
(4) `size[i]` $\leftarrow 1$;
(5) `next[i]` $\leftarrow 0$.

Zeitaufwand: $\Theta(n)$.

Prozedur find(i)

(1) **return** `r[`i`]`.

Zeitaufwand: $O(1)$.

Prozedur union(s, t)

```
    // Ausgangspunkt: s, t sind verschiedene Repräsentanten
(1) if size[s] > size[t] then vertausche s, t;
(2)    // nun: size[s] ≤ size[t]
(3) z ← s;
(4) r[z] ← t;
(5) while next[z] ≠ 0 do        // durchlaufe L_s, setzt r-Werte um
(6)    z ← next[z];
(7)    r[z] ← t;
(8) // nun: z enthält letztes Element von L_s; next[z] = 0
(9) // L_s nach dem ersten Eintrag in L_t einhängen:
(10) next[z] ← next[t]; next[t] ← s;
(11) size[t] ← size[t] + size[s].
```

Aufwand: $O(\text{Länge der kürzeren der Listen } L_s, L_t)$.

Behauptung:

$n-1$ **union**-Aufrufe (mehr kann es nicht geben!) benötigen Zeit $O(n \log n)$.

Wir behaupten **nicht**, dass jeder einzelne **union**-Aufruf Zeit $O(\log n)$ benötigt (hierfür kann man leicht Gegenbeispiele konstruieren) – aber in der Summe entfällt auf jeden solchen Aufruf ein Anteil von $O(\log n)$:

„Amortisierte Analyse“

Wir nummerieren die **union**-Aufrufe mit $p = 1, 2, \ldots, n-1$ durch. Die beiden in Aufruf Nummer p vereinigten Klassen seien $\boldsymbol{K_{p,1}}$ und $\boldsymbol{K_{p,2}}$, wobei $\boldsymbol{|K_{p,1}| \leq |K_{p,2}|}$ gelten soll.

Für den p-ten **union**-Aufruf veranschlagen wir Kosten $\boldsymbol{|K_{p,1}|}$.

Dann ist der Gesamt-Zeitaufwand für alle **union**s zusammen

$$O(n) + O\Big(\sum_{1 \leq p < n} |K_{p,1}| \Big).$$

Wir wollen $\sum_{1 \leq p < n} |K_{p,1}|$ abschätzen.

Trick: Ermittle den Beitrag zu dieser Summe aus Sicht der einzelnen Elemente i.

Für $1 \leq i \leq n$, $1 \leq p < n$ definiere:

$$a_{i,p} := \begin{cases} 1 & \text{falls } i \in K_{p,1}, \\ 0 & \text{sonst.} \end{cases}$$

Dann erhält man durch Umordnen der Summation:

$$\sum_{1 \leq p < n} |K_{p,1}| = \sum_{1 \leq p < n} \sum_{1 \leq i \leq n} a_{i,p} = \sum_{1 \leq i \leq n} \Big(\sum_{1 \leq p < n} a_{i,p} \Big).$$

In der letzten inneren Summe $c_i = \sum_{1 \leq p < n} a_{i,p}$ wird für jedes $i \in \{1, \ldots, n\}$ gezählt, wie oft es bei **union**-Operationen in der kleineren Menge $K_{p,1}$ enthalten ist (d. h. wie oft insgesamt der Repräsentant seiner Klasse gewechselt hat).

Bei jeder **union**-Operation, bei der i in der kleineren Klasse ist, wird die Größe der Klasse mindestens **verdoppelt**, startend mit der Klasse $\{i\}$. Da Klassen nicht größer als n werden können, gilt $2^{c_i} \leq n$, also $c_i \leq \log n$, also

$$\sum_{1 \leq p < n} |K_{p,1}| = \sum_{1 \leq i \leq n} c_i \leq n \log n.$$

Damit ist die Behauptung bewiesen. □

Satz 11.4.2

In der Implementierung der Union-Find-Struktur mit Arrays hat jede **find**-Operation Rechenzeit $O(1)$, $n-1$ **union**-Operationen haben Rechenzeit $O(n \log n)$.

Satz 11.4.1 (Vollversion)

(a) . . .

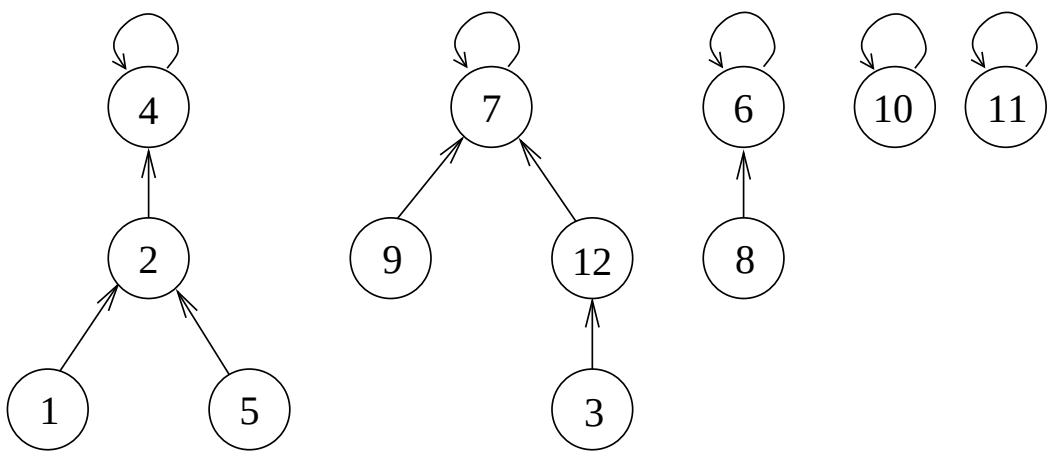
(b) Die Rechenzeit des Algorithmus von Kruskal ist $O(m \log n) + O(m) + O(n \log n)$, also $O(m \log n)$, wenn man die Union-Find-Struktur mit **Arrays** implementiert.

Beweis:
Sortieren der Kanten hat Kosten $O(m \log m) = O(m \log n)$.
$2m$ **find**-Operationen haben Kosten $O(m)$.
$n-1$ **union**-Operationen haben Kosten $O(n \log n)$. □

11.4.3 Baumimplementierung von Union-Find

Eine attraktive Implementierung der Union-Find-Datenstruktur verwendet einen **„wurzelgerichteten Wald“**.

Beispiel: Partition $\{1, 2, \mathbf{4}, 5\}, \{3, \mathbf{7}, 9, 12\}, \{\mathbf{6}, 8\}, \{\mathbf{10}\}, \{\mathbf{11}\}$ wird dargestellt durch:



Für jede Klasse K_s gibt es genau einen Baum B_s. Jedes $i \in K_s$ ist ein Knoten im Baum B_s.
Es gibt nur Zeiger in Richtung auf die Wurzel zu. $p(i)$ ist der Vorgänger von i.
Die Wurzel ist der Repräsentant s.
Sie zeigt auf sich selbst als „Vorgänger“: $p(i) = i$ genau dann wenn i Repräsentant ist.

Eine kostengünstige Darstellung eines solchen Waldes benutzt nur ein Integerarray `p[1..n]`. Für Knoten i gibt der Eintrag `p[`i`]` den Vorgängerknoten $p(i)$ an.

Das dem Wald im Beispiel entsprechende Array sieht so aus:

	1	2	3	4	5	6	7	8	9	10	11	12
p:	2	4	12	4	2	6	7	6	7	10	11	7

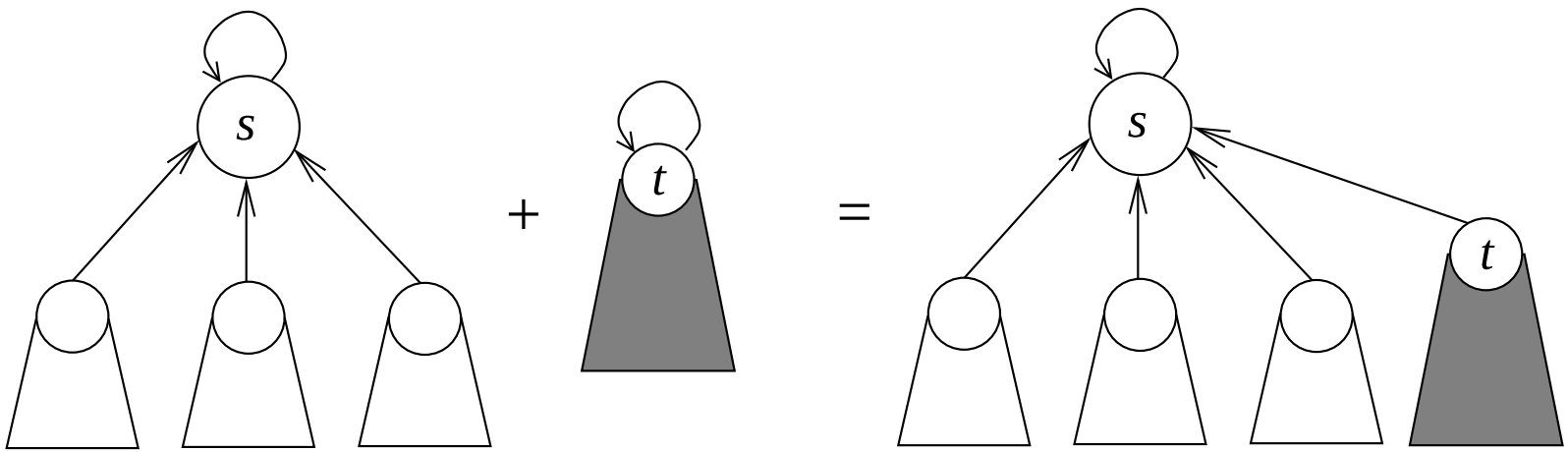
Prozedur find(i)

```
(1)  j ← i;
(2)  jj ← p[j];            // (2)–(6): verfolge Vorgängerzeiger bis zur Wurzel
(3)  while jj ≠ j do
(4)     j ← jj;
(5)     jj ← p[j] ;
(6)  return j.
```

Zeitaufwand: $\Theta(\mathit{depth}(i)) = \Theta(\text{Tiefe von } i \text{ in seinem Baum})$.

union(s, t): Gegeben: Verschiedene Repräsentanten s und t.

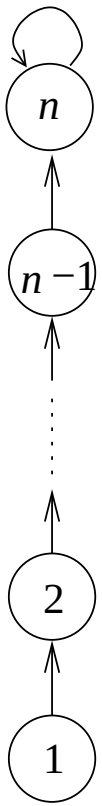
Man macht einen der Repräsentanten zum Kind des anderen, setzt also $\mathrm{p}(s) := t$ **oder** $\mathrm{p}(t) := s$.



? Welche der beiden Optionen sollte man nehmen?

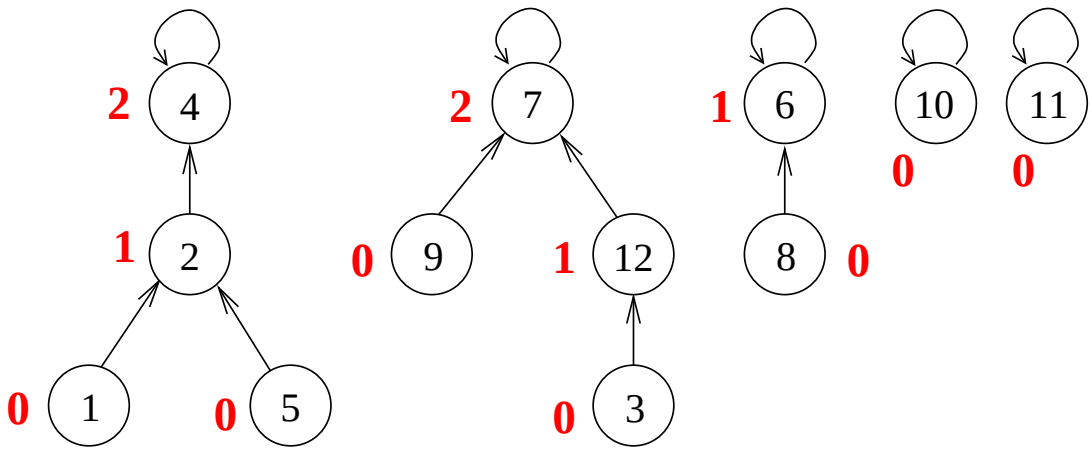
Ungeschickt: union$(i, i+1)$, $i = 1, \ldots, n-1$, dadurch ausführen, dass man nacheinander $\mathrm{p}(i) := i + 1$ ausführt.

Dies führt zu dem Baum



find-Operationen sind nun sehr teuer!
Ein **find**(i) für jedes $i \in \{1, \ldots, n\}$ führt zu Gesamtkosten $\Theta(n^2)$!

Trick: Führe für jeden Knoten i eine Zahl *rank*(i) **(Rang)** mit, in einem Array `rank[1..n]: array of int`, mit `rank[`i`]` $=$ Tiefe des Teilbaums mit Wurzel i.



Für **union**(s, t) benutzen wir die Regel „*union by rank*“:
Die Wurzel mit dem größeren Rang wird Wurzel des neuen Baums, keine Änderung der Ränge.

Bei gleichen Rängen ist die Wahl der Wurzel gleichgültig.
In diesem Fall erhöht sich der Rang des Knotens, der Wurzel bleibt, um 1.

Operationen:

Prozedur init(n) // Initialisierung

(1) Erzeuge p, rank: Arrays der Länge n für int-Einträge
(2) **for** i **from** 1 **to** n **do**
(3) p[i] $\leftarrow$ i; rank[i] $\leftarrow 0$; // n Bäume mit je einem (Wurzel-)Knoten vom Rang 0

Zeitaufwand: $\Theta(n)$.

Prozedur union(s, t) // „union by rank“

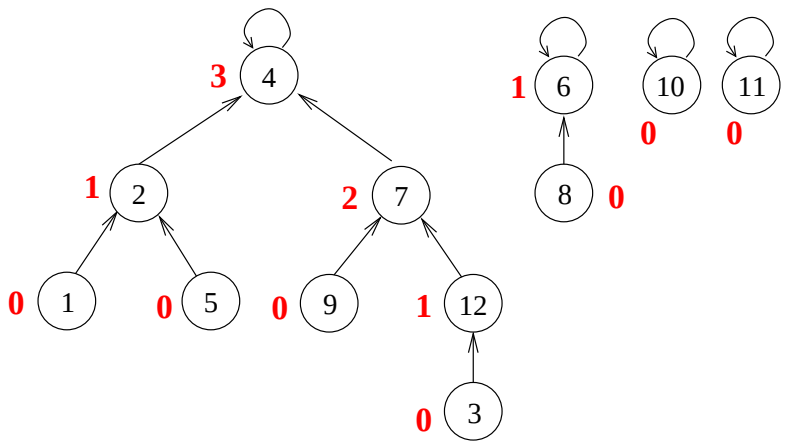
// Ausgangspunkt: s, t sind verschiedene Repräsentanten von Klassen
// D.h.: p[s] $= s$ und p[t] $= t$

(1) **if** rank[s] $>$ rank[t]
(2) **then** p[t] $\leftarrow s$
(3) **elseif** rank[t] $>$ rank[s]
(4) **then** p[s] $\leftarrow t$
(5) **else** p[t] $\leftarrow s$; rank[s] $\leftarrow$ rank[s] $+ 1$.

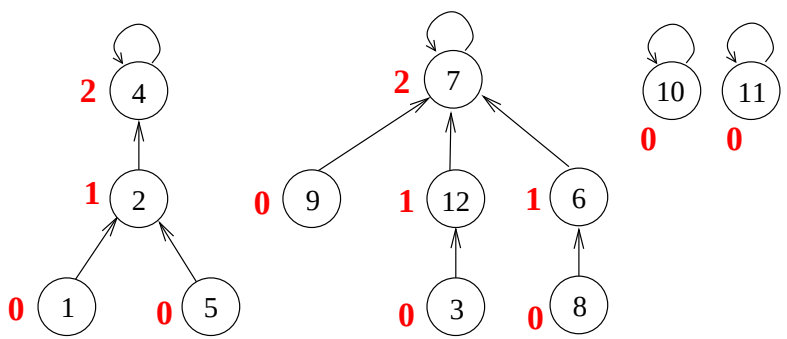
Zeitaufwand: $O(1)$.

Beispiele:

union$(4, 7)$ würde liefern:



union$(6, 7)$ würde liefern:



Satz 11.4.3

Die eben beschriebene Implementierung einer Union-Find-Struktur mit einem wurzelgerichteten Wald hat folgende Eigenschaften:

a) Sie ist korrekt (d. h. hat das vorgesehene Ein-/Ausgabeverhalten).

b) **init**(n) benötigt Zeit $\Theta(n)$; **find**(i) benötigt Zeit $O(\log n)$; **union**(s,t) benötigt Zeit $O(1)$.

Beweis: (a) ist unmittelbar klar.

(b) Wir beobachten:

Fakt 1: $i \neq p(i) \Rightarrow \mathit{rank}(i) < \mathit{rank}(p(i))$.

Beweis: Wenn s bei einer **union**-Operation Kind von t wird, dann ist entweder $\mathit{rank}(s) < \mathit{rank}(t)$ (und das bleibt so) oder es ist $\mathit{rank}(s) = \mathit{rank}(t)$, und der Rang von t erhöht sich nun um 1.
Nachher ist s keine Wurzel mehr. Daher ändert sich $\mathit{rank}(s)$ nicht mehr, und auch $p(s)$ bleibt gleich. Der Rang von $p(s)$ kann im weiteren Verlauf nur noch wachsen. Daher bleibt die behauptete Ungleichung erhalten.

Fakt 2: Wenn s Wurzel des Baums B_s ist und $h = \mathit{rank}(s)$ gilt, dann enthält B_s mindestens 2^h Knoten.

Beweis: Für $h = 0$ ist die Aussage richtig. Eine Wurzel vom Rang $h \geq 1$ entsteht, wenn zwei Bäume vereinigt werden, deren Wurzeln beide Rang $h - 1$ haben. Daraus folgt Fakt 2 leicht durch Induktion über $h = 0, 1, \ldots$.

Fakt 3: Bei n Knoten insgesamt gibt es höchstens $n/2^h$ Knoten von Rang h.

Beweis: Wegen Fakt 1 können Knoten mit Rang h nicht Nachfahren voneinander sein. Also sind alle Unterbäume, deren Wurzeln Rang h haben, disjunkt. Aus Fakt 2 folgt leicht, dass auch für Knoten i im Inneren von Bäumen mit $\mathit{rank}(i) = h$ gilt, dass ihr Unterbaum mindestens 2^h Knoten hat. Also kann es nicht mehr als $n/2^h$ solche Knoten geben.

Aus Fakt 3 folgt, dass es keine Knoten mit Rang größer als $\log n$ geben kann.
Also hat kein Baum Tiefe größer als $\log n$, also kosten **find**-Operationen maximal Zeit $O(\log n)$. □

Pfadkompression

Eine interessante Variante der Union-Find-Datenstruktur, die mit einem wurzelgerichteten Wald implementiert ist, ist der Ansatz der „**Pfadkompression**“ (oder „**Pfadverkürzung**“).

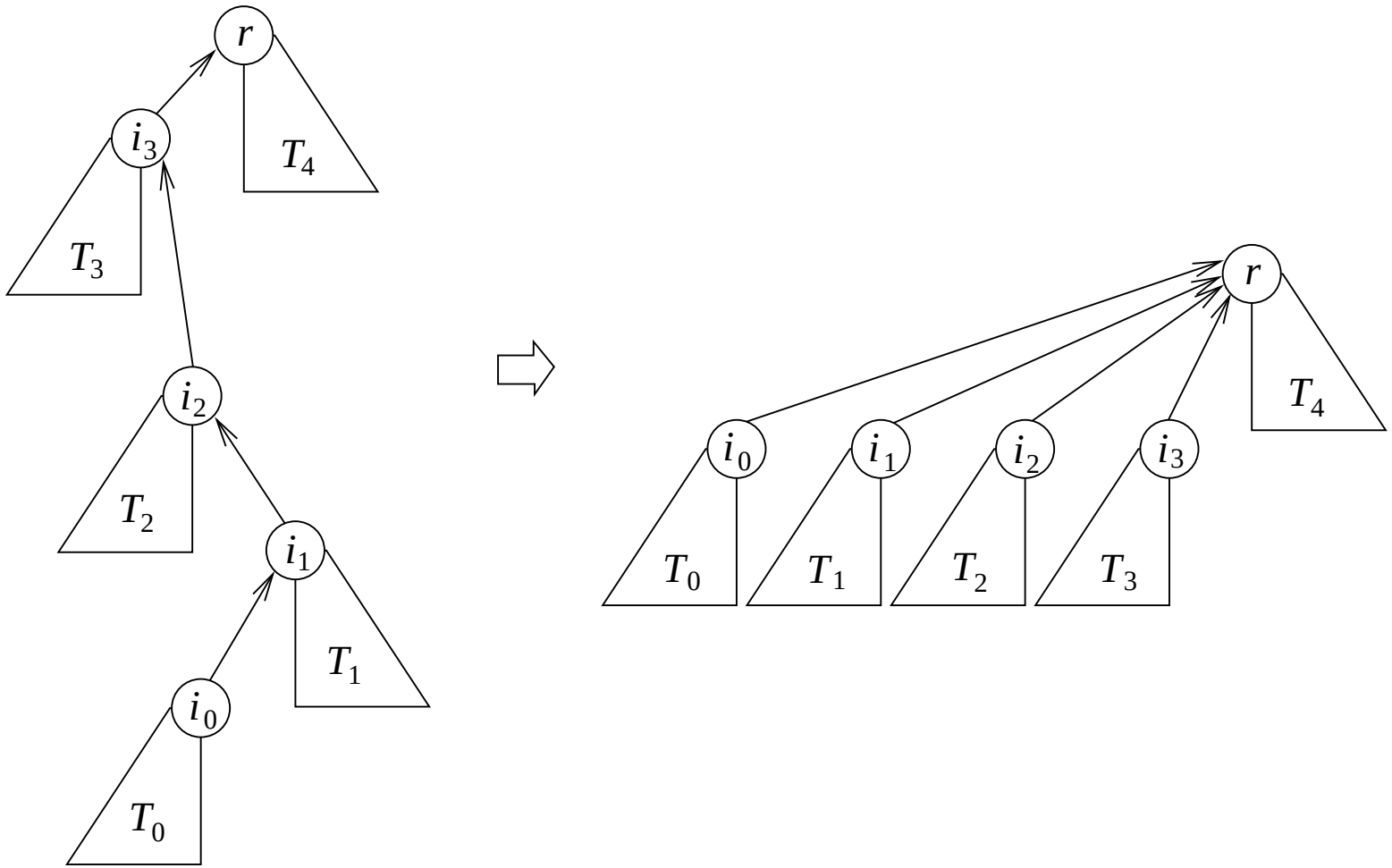
Bei einem **find**(i) muss man den ganzen Weg von Knoten i zu seiner Wurzel $r(i)$ ablaufen; Aufwand $O(\mathit{depth}(i))$.

Ein gutes strategisches Ziel ist also, diese Wege möglichst kurz zu halten.

Idee: Man investiert bei **find**(i) etwas mehr Arbeit, jedoch immer noch im Rahmen $O(\mathit{depth}(i))$, um dabei einige **Wege zu verkürzen** und damit spätere **find**s billiger zu machen.

Jeder Knoten $i = i_0, i_1, \ldots, i_{d-1}$ auf dem Weg von i zur Wurzel $i_d = r(i)$ wird direkt als Kind an die Wurzel gehängt.
Kosten pro Knoten/Ebene: $O(1)$, insgesamt also $O(\mathit{depth}(i))$.

Beispiel: $i_0 = i$ mit $d = \mathit{depth}(i) = 4$.

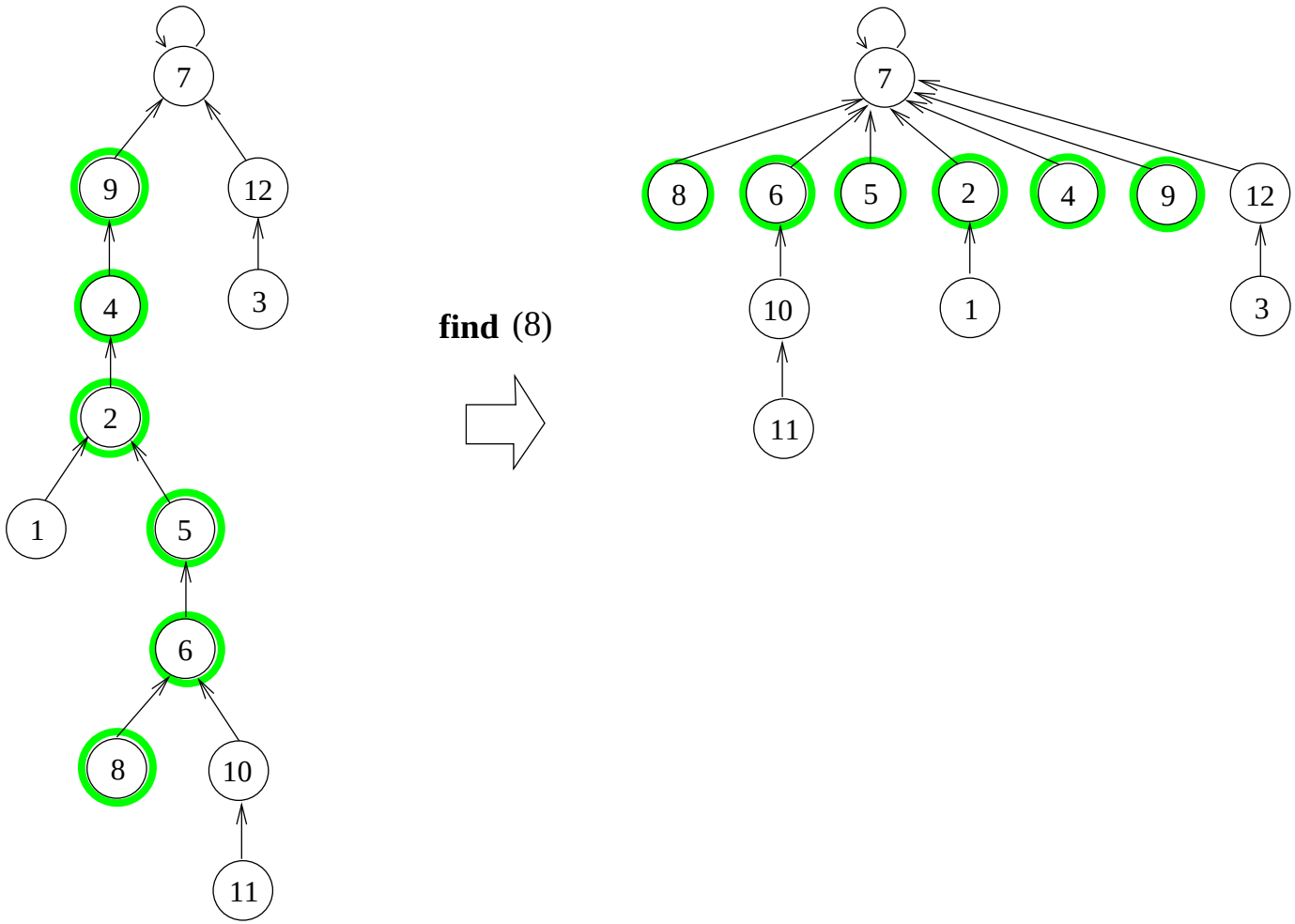


Die neue Version der **find**-Operation:

Prozedur find(i) // Pfadkompressions-Version

```
(1)   j ← i;
      // verfolge Vorgängerzeiger bis zur Wurzel r(i):
(2)   jj ← p[j];
(3)   while jj ≠ j do
(4)     j ← jj;
(5)     jj ← p[j] ;
(6)   r ← j;   // r enthält nun Wurzel r(i)
      // Erneuter Lauf zur Wurzel, Vorgängerzeiger umhängen:
(7)   j ← i;
(8)   jj ← p[j];
(9)   while jj ≠ j do
(10)    p[j] ← r;
(11)    j ← jj;
(12)    jj ← p[j];
(13)  return r.
```

Beispiel:



2019 nicht prüfungsrelevant: Folien 90–91, 93–100

Die **union**-Operationen werden exakt wie bisher durchgeführt.

Beobachtungen: (1) Die Werte im `rank`-Array werden aktualisiert wie vorher.
Man kann sie aber nicht mehr als Baumtiefen interpretieren.

(2) Nach (1) fällt bei einer **union**-Operationen die Entscheidung, welcher Knoten Kind eines anderen wird, exakt wie im Verfahren ohne Pfadkompression. Daher verändert die Pfadkompression die Knotenmengen der Bäume (also die Klassen) und die Repräsentanten nicht.

Aus (1) und (2) folgt, dass **Fakt 2** weiterhin gilt.

Bem.: Nicht-Wurzeln vom Rang h können – weil sie bei der Pfadkompression Kindknoten verlieren können – weniger als 2^h Nachfahren haben.

(3) Wenn ein Knoten i Kind eines anderen wird (also aufhört, Repräsentant zu sein), dann **ändert** sich sein **Rang**, wie in `rank`$[i]$ gespeichert, **nie mehr**. Dies gilt in der Version ohne und in der Version mit Pfadkompression.

Diesen Wert werden wir als den „endgültigen“ Rang von i ansehen.

Die Rang-Werte der Nicht-Wurzeln sind dann in der Version mit und in der Version ohne Pfadkompression **identisch**.

Solange ein Knoten Wurzel ist, ist sein Rang nicht endgültig, aber in beiden Versionen gleich.

Insbesondere: **Fakt 3** gilt weiterhin.

(4) Weder **union**- noch **find**-Operationen (mit Pfadkompression) ändern etwas an **Fakt 1**: Ränge wachsen strikt von unten nach oben entlang der Wege in den Bäumen.
(Beweis durch Induktion über ausgeführte Operationen, Übung.)

Definition: $F(0) := 1, \qquad F(i) := 2^{F(i-1)}$, für $i \geq 1$.

$F(i)$ ist die Zahl, die von einem „Zweierpotenzturm“ der Höhe i berechnet wird:

$$F(i) = \; 2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}} \; i \text{ Zweien}$$

Beispielwerte:

i	0	1	2	3	4	5	6
$F(i)$	1	2	4	16	65536	2^{65536}	$2^{2^{65536}}$

Definition: $\log^* n := \min\{k \mid F(k) \geq n\}$.

Leicht zu sehen: $\log^* n = \min\{k \mid \underbrace{\log \log \ldots \log}_{k} n \leq 1\}$.

Nun teilen wir die Knoten j mit (endgültigem) Rang > 0 (keine Blätter, keine Wurzeln) in **„Ranggruppen“** $G_0, G_1, G_2, \ldots$ ein:

Knoten j gehört zu Ranggruppe G_k, wenn $F(k-1) < \mathit{rank}(j) \leq F(k)$ gilt.

G_0: Rang 1; G_1: Rang 2; G_2: Ränge $3, 4$; G_3: Ränge $5, \ldots, 16$; G_4: Ränge $17, \ldots, 65536$; etc.

Wenn j in Ranggruppe G_k ist, folgt (mit Fakt 3):

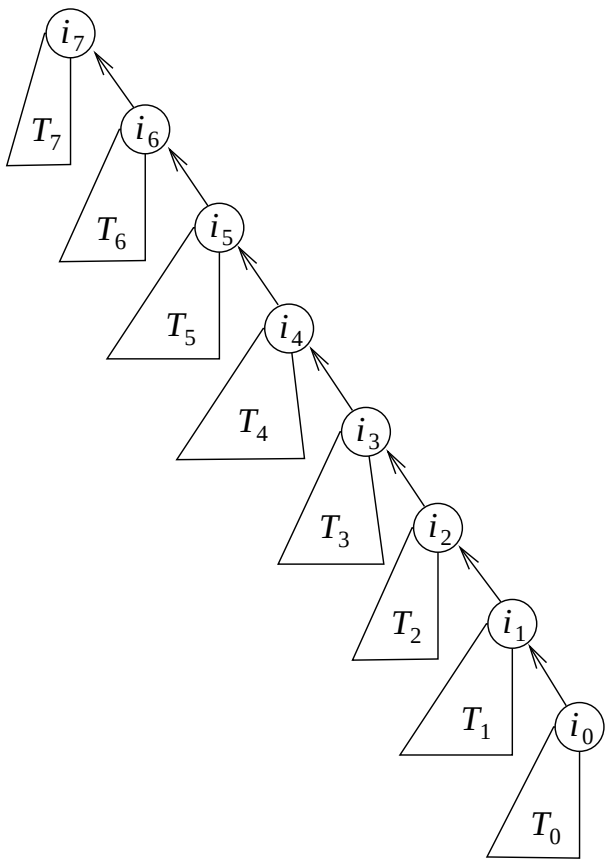
$F(k) = 2^{F(k-1)} < 2^{\mathit{rank}(j)} \leq 2^{\log n} = n.$

Also: $k < \log^* n$, d. h. es gibt maximal $\log^* n$ nichtleere Ranggruppen.

Beachte: $2^{65536} > 1000^{6553} > 10^{19500}$; Werte n mit $\log^* n > 5$, in denen also Ranggruppe G_5 eventuell nicht leer ist, kommen in wirklichen algorithmischen Anwendungen nicht vor $\Rightarrow$ Man wird nie mehr als fünf nichtleere Ranggruppen sehen.

(Es gilt aber: $\lim_{n\to\infty} \log^* n = \infty$.)

Wir wollen nun die Kosten von m **find**-Operationen berechnen. Bei **find**(i) läuft man einen Weg $i = i_0, i_1 = p(i_0), i_2 = p(i_1), \ldots, i_d = p(i_{d-1})$ entlang, von i bis zur Wurzel $r(i) = i_d$.
Beispiel: $d = 7$.



Wir veranschlagen Kosten $d + 1$ für diese Operation, also 1 für jeden Knoten j auf dem Weg.
Die **Rechenzeit** für alle m **find**s zusammen ist dann $O(\text{Summe aller „Kosten“})$.
Für die Kostenanalyse benutzen wir die sogenannte **Bankkontomethode**, in einer anschaulichen Form.

Ein Knoten in Ranggruppe G_k bekommt $F(k)$ € Taschengeld, in dem Moment, in dem er aufhört, Repräsentant/Baumwurzel zu sein, sein Rang also endgültig feststeht.

In G_k sind Knoten mit Rängen $F(k-1)+1, \ldots, F(k)$. Nach Fakt 3 gilt:

$$|G_k| \leq \frac{n}{2^{F(k-1)+1}} + \frac{n}{2^{F(k-1)+2}} + \cdots + \frac{n}{2^{F(k)}} < \frac{n}{2^{F(k-1)}} = \frac{n}{F(k)}.$$

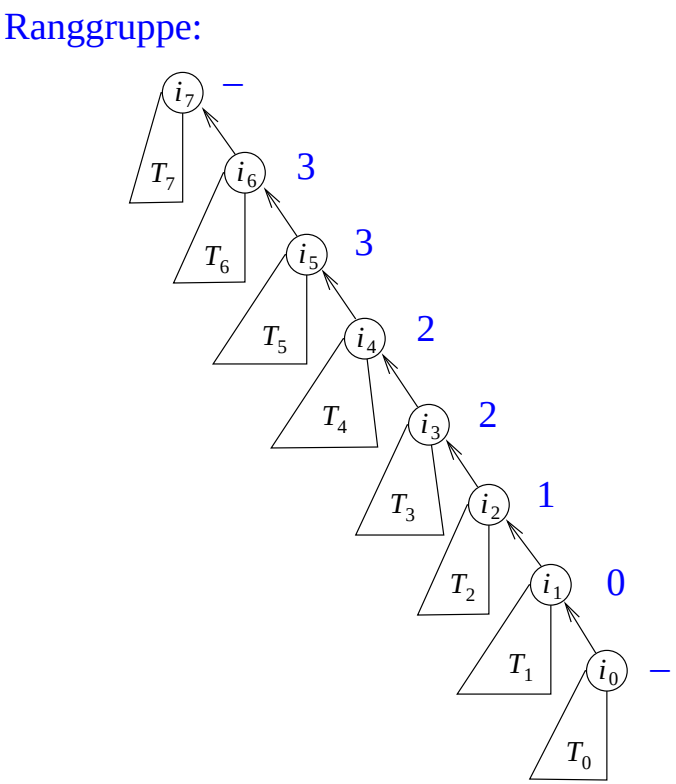
Also bekommen alle Knoten in G_k zusammen höchstens n €, und in allen höchstens $\log^* n$ Ranggruppen zusammen werden nicht mehr als $n \log^* n$ € als Taschengeld ausgezahlt.

Dies ist ein entscheidender Punkt in der Analyse: Die Ranggruppen sind sehr klein, daher kann man sich ein großzügiges Taschengeld leisten.

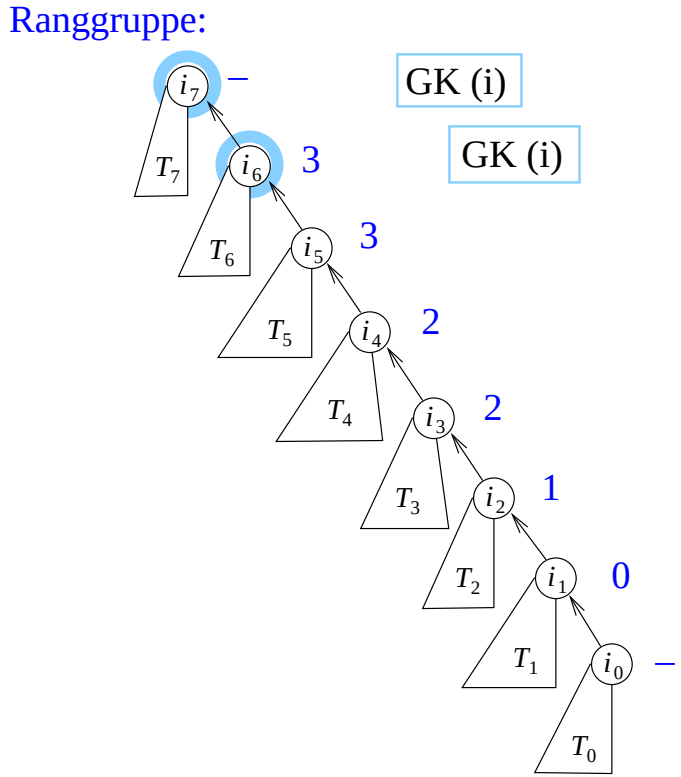
Weiter legen wir $m(2 + \log^* n)$ € in einer „Gemeinschaftskasse“ bereit.

Ziel: Wir wollen zeigen, dass das Taschengeld und das Geld in der Gemeinschaftskasse zusammen ausreichen, um die Kosten aller m **find**-Operationen zu bestreiten.

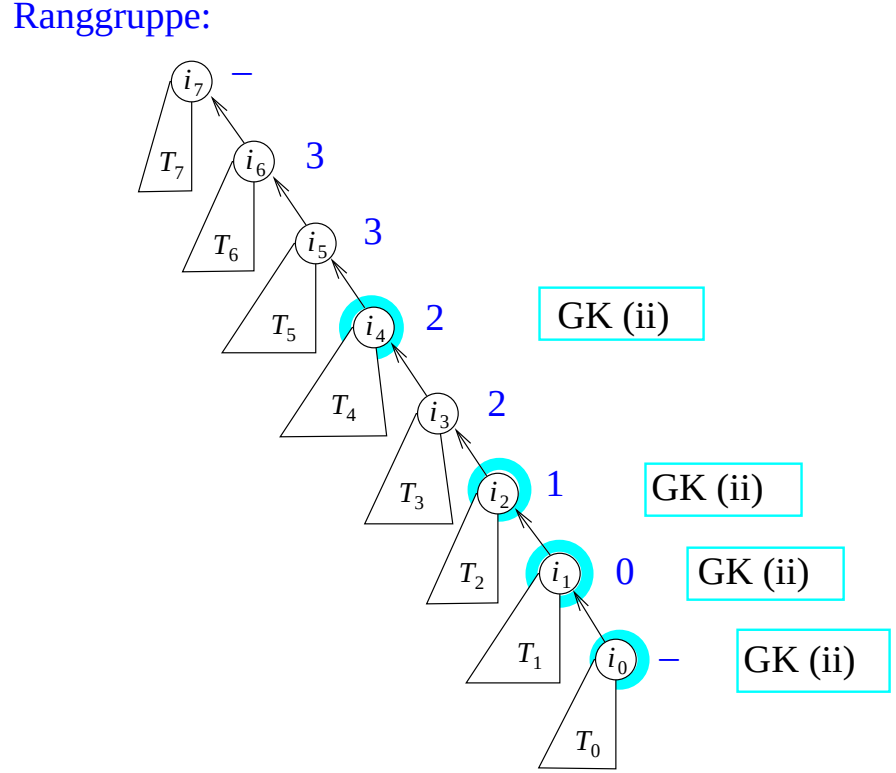
Wir betrachten eine **find**(i)-Operation. Jeder Knoten j auf dem Weg verursacht Kosten 1.



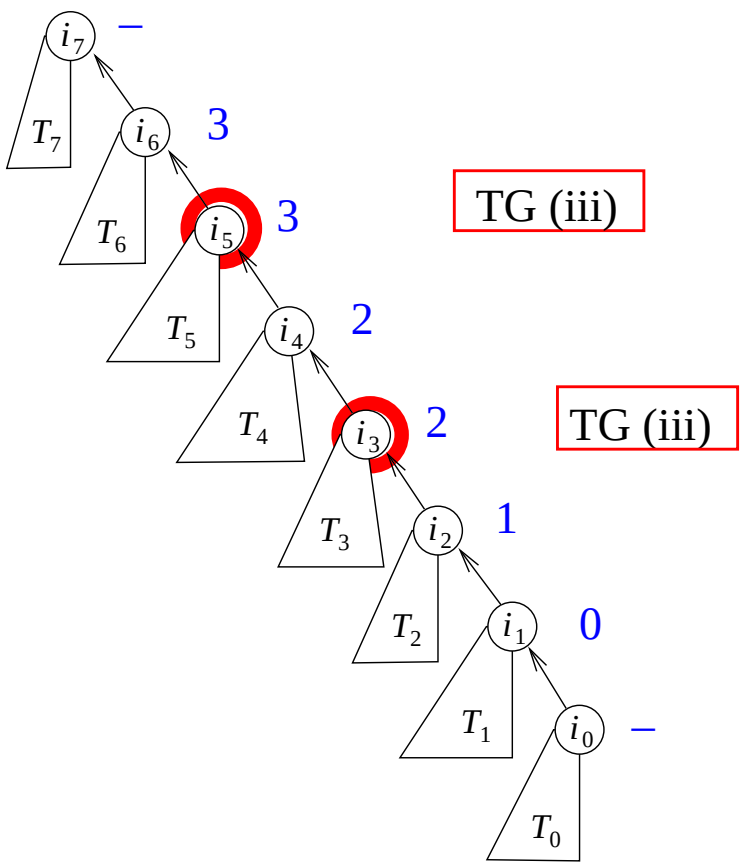
Es gibt drei Fälle.



(i) j ist **Wurzel** oder ist der **vorletzte** Knoten auf dem Weg von i nach $r(i)$.
Die Kosten von j bestreiten wir aus der Gemeinschaftskasse.
Dieser Fall kostet maximal 2€ für die **find**(i)-Operation.
Ab hier: j ist nicht Wurzel oder vorletzter Knoten.



(ii) *rank*$(j) = 0$ oder $p(j)$ ist in einer höheren Ranggruppe als j.
Es gibt höchstens $\log^* n$ viele solche j's, weil Ränge entlang des Weges strikt wachsen (Fakt 1).
Kosten dieser j sind $\leq \log^* n$ €, werden aus Gemeinschaftskasse bezahlt.



(iii) $p(j)$ ist in derselben Ranggruppe wie j.
In diesem Fall muss j mit $1\,€$ aus seinem Taschengeld bezahlen.

Nun wird abgerechnet:

(A) Die Gesamtkosten aus Fällen (i) und (ii), summiert über alle m **find**s, betragen höchstens $m(2 + \log^* n)$; der Inhalt der Gemeinschaftskasse reicht.

(B) Wie steht es mit Fall (iii) und dem Taschengeld?

Wir betrachten nun **(Trick!)** einen festen Knoten j.

Immer wenn j bezahlen muss, nimmt er an einer Pfadverkürzung teil. Sein neuer Vorgängerknoten $p'(j)$ wird seine aktuelle Wurzel (verschieden vom bisherigen Vorgänger $p(j)$).

Wegen Fakt 1 hat der **neue Vorgänger** $p'(j)$ einen **höheren Rang** als der **alte**.

Sei G_k die Ranggruppe von j. In dieser Ranggruppe gibt es nicht mehr als $F(k)$ verfügbare Rang-Werte. Daher ändert sich der Vorgänger von j weniger als $F(k)$-mal, bevor ein Knoten aus einer höheren Ranggruppe Vorgänger von j wird (und alle späteren Vorgänger ebenfalls aus höheren Ranggruppen kommen).

Also tritt Situation (iii) für Knoten j weniger als $F(k)$-mal ein. Daher reicht das Taschengeld von Knoten j aus, um für diese Situationen zu bezahlen.

Die Gesamtkosten aus Fall (iii), summiert über alle j's, betragen also nicht mehr als das gesamte Taschengeld, also höchstens $n \log^* n$.

Satz 11.4.4

In der Implementierung der Union-Find-Struktur mit wurzelgerichteten Wäldern und Pfadkompression haben m **find**-Operationen insgesamt Kosten von maximal $(2 + m + n) \log^* n$, benötigen also Rechenzeit $O((m + n) \log^* n)$.
Jede einzelne **union**-Operation benötigt Zeit $O(1)$.

Bemerkung:
(a) Da für real vorkommende n die Zahl $\log^* n$ nicht größer als 5 ist, wird man in der Praxis eine Rechenzeit beobachten, die linear in $n + m$ ist.
(b) Implementierungen von Union-Find-Strukturen als wurzelgerichteter Wald mit Pfadkompression verhalten sich in der Praxis sehr effizient.

Satz 11.4.1 (Vollversion)

(a) Der Algorithmus von Kruskal in der Implementierung mit Union-Find (Folie 64) ist korrekt.

(b) Die Rechenzeit des Algorithmus ist $O(m \log m) + O(m) + O(n \log n)$, also $O(m \log n)$, wenn man die Union-Find-Struktur mit **Arrays** implementiert.

(c) Die Rechenzeit des Algorithmus ist $O(m \log m) + O(m \log^* n)$, also $O(m \log n)$, wenn die Union-Find-Struktur mit **wurzelgerichteten Bäumen** mit **Pfadkompression** implementiert wird.

Beweis: (a) haben wir schon gesehen.

(b), (c): Der erste Term $O(m \log m)$ benennt die Kosten für das Sortieren. Danach sind $2m$ **find**-Operationen und $n-1$ **union**-Operationen durchzuführen. Die Zeiten hierfür in beiden Implementierungen wurden in den Sätzen 11.4.2, 11.4.3, 11.4.4 begründet. □

Bemerkung

Wenn die Kanten schon sortiert sind oder billig sortiert werden können ($O(m)$ Zeit, z. B. mittels Radixsort, siehe Abschn. 6.7), und $G = (V, E)$ sehr wenige Kanten hat, ergeben sich noch günstigere Rechenzeiten:

- Union-Find mit Arrays und Listen liefert Rechenzeit $O(m + n \log n)$: das ist linear, sobald $m = \Omega(n \log n)$ ist, also für nicht ganz dünn besetzte Graphen;
- Union-Find mit Pfadkompression liefert Rechenzeit $O(m \log^* n)$: fast linear.

Beachte aber: Für $m = \Omega(n \log n)$ bietet der Algorithmus von Jarník/Prim mit Fibonacci-Heaps lineare Rechenzeit ohne Annahmen über die Sortierkosten.

SS 2021

Algorithmen und Datenstrukturen

12. Kapitel

Dynamische Programmierung

Martin Dietzfelbinger

Juli 2021

Kapitel 12: Dynamische Programmierung (DP)

Algorithmenparadigma für **Optimierungsprobleme**. Typische Schritte:

- Definiere (viele) **„Teilprobleme“** (einer Instanz)
- Identifiziere einfache **Basisfälle**
- Formuliere eine Version der Eigenschaft

> **Substrukturen optimaler Strukturen sind optimal**

- Finde Rekursionsgleichungen für Werte optimaler Lösungen:

> **Bellmansche Optimalitätsgleichungen**

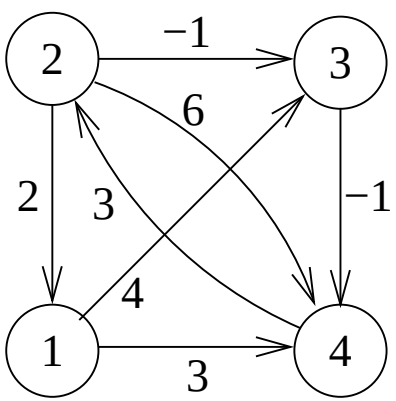
- Berechne optimale Werte (und Strukturen) **iterativ**.

12.1 Das All-Pairs-Shortest-Paths-Problem

Das „**APSP-Problem**“ ist zentrales Beispiel für die Strategie „Dynamische Programmierung“.

„Kürzeste Wege zwischen allen Paaren von Knoten“.

Eingabe:



Digraph $G = (V, E, c)$ mit $V = \{1, \ldots, n\}$ und $E \subseteq \{(v, w) \mid 1 \leq v, w \leq n, v \neq w\}$,

$c\colon E \to \mathbb{R} \cup \{+\infty\}$: **Gewichts-/Kosten-/Längenfunktion**.

Die **Länge** eines Kantenzugs $p = (v = v_0, v_1, \ldots, v_r = w)$ ist

$$c(p) = \sum_{1 \leq s \leq r} c(v_{s-1}, v_s).$$

Gesucht: für alle v, w, $1 \leq v, w \leq n$:
Kantenzug p von v nach w mit minimalem $c(p)$ („kürzester Weg“).

Erinnerung: **Single-Source-Shortest-Paths-Problem (SSSP-Problem)**.
Im Fall **nichtnegativer** Kantengewichte löst n-maliges Aufrufen des Algorithmus von **Dijkstra** das APSP-Problem. Rechenzeit: $n \cdot O(m \log n) = O(nm \log n)$.

Der im Folgenden beschriebene **Algorithmus von Floyd–Warshall*** kann auch mit negativen Kantengewichten umgehen.

* Drei unabh. Publikationen: 1962 R. W. Floyd (1936–2001), amer. Informatiker; 1959 Stephen Warshall (1935–2006), amer. Informatiker; 1959 Bernard Roy (1934–2017), frz. Mathematiker

Wir verlangen aber: Es gibt keine **Kreise** mit (strikt) negativer Gesamtlänge, d. h.

$$(*) \qquad v = v_0, v_1, \ldots, v_r = v \text{ Kreis } \Rightarrow \sum_{1 \le s \le r} c(v_{s-1}, v_s) \ge 0.$$

Grund für diese Annahme: Wenn es einen solchen Kreis von v nach v gibt, und irgendein Weg von v nach w existiert, dann gibt es Kantenzüge von v nach w mit beliebig stark negativer Länge. Das heißt: Die Frage nach einem „kürzesten Weg“ von v nach w ist sinnlos.

Konsequenzen aus $(*)$:

(1) Wenn p Kantenzug von v nach w ist, dann existiert auch ein (einfacher) **Weg** p' von v nach w mit $c(p') \le c(p)$.

(Begründung: Wenn p Segment $(u, \ldots, u)$ enthält, ersetze es durch (u). Hierdurch kann sich die Länge des Kantenzugs nicht vergrößern, weil es keine negativen Kreise gibt. Iteriere dies, bis alle Knotenwiederholungen beseitigt sind.)

(2) Wenn es einen Kantenzug von v nach w gibt, dann auch einen mit minimaler Länge (einen **„kürzesten Weg“**).

(Begründung: Wegen (1) muss man nur unter den (einfachen) **Wegen** von v nach w (mit nicht mehr als $n - 1$ Kanten) einen kürzesten finden; davon gibt es nur endlich viele.)

Bemerkungen:

- Es kann mehrere „kürzeste Wege“ von v nach w geben.
  (Selbst wenn alle Kantenlängen 1 sind, kann es mehrere Wege von v nach w mit gleich vielen „hops“ geben.)
- Deshalb: „Ein“ kürzester Weg, nicht „der“ kürzeste Weg.
- Wenn $G = (V, E, c)$ einen Kreis $(v, \ldots, v)$ mit Kosten 0 hat, der mindestens eine Kante hat, dann gibt es sogar unendlich viele Kantenzüge von v nach w von minimaler Länge.
  (Durchlaufe den Kreis mehrfach.)

Für Digraphen $G = (V, E, c)$, mit $(*)$, definieren wir:

$$\boldsymbol{S(v, w)} := \text{die Länge eines kürzesten Weges von } v \text{ nach } w.$$

(Wenn $v \not\rightsquigarrow w$, setzen wir $S(v, w) := \infty$.)

APSP-Problem, vereinfacht: Bestimme $S(v, w)$, für alle $v, w \in V$.

Gewünschte **Ausgabe:** Matrix $S = (S(v, w))_{1 \leq v, w \leq n}$.

O. B. d. A.: $E = V \times V - \{(v,v) \mid v \in V\}$. (Wenn $(v,w) \notin E$, setzen wir $c(v,w) = \infty$.)

Erster Schritt: Identifiziere geeignete **Teilprobleme**.

Gegeben ein k, $0 \leq k \leq n$, betrachte nur (einfache) Wege von v nach w, die **unterwegs** (also vom Start- und Endpunkt abgesehen) nur Knoten in $\{1, \ldots, k\}$ besuchen (nur Wege $(v, v_1, \ldots, v_{r-1}, w)$ mit $v_1, \ldots, v_{r-1} \in \{1, \ldots, k\}$).

$k = 0$: Kein Knoten darf unterwegs besucht werden; es handelt sich nur um die Kanten (v,w) und um Wege (v) (keine Kante, Kosten 0), wenn $v = w$.

$k = 1$: Die Wege aus dem Fall „$k = 0$“ und zusätzlich Wege $(v, 1, w)$.

$k = 2$: Die Wege aus dem Fall „$k = 1$“ und Wege $(v, 1, 2, w)$, $(v, 2, 1, w)$, $(v, 2, w)$.

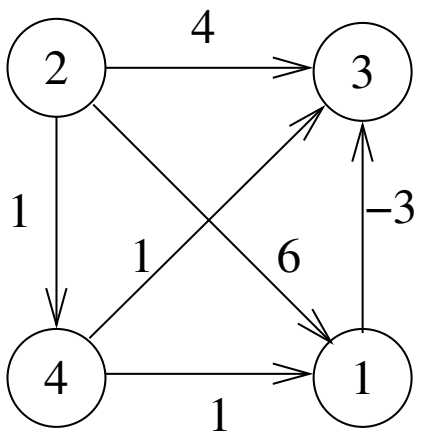
Solche Wege nennen wir ***k*-Wege** von v nach w.

Für $1 \leq v, w \leq n$ und $0 \leq k \leq n$ definieren wir:

$$\boldsymbol{S(v,w,k)} := \text{minimale Länge } \sum_{1 \leq i \leq r} c(v_{i-1}, v_i) \text{ eines } k\text{-Weges } (v = v_0, \ldots, v_r = w) \text{ von } v \text{ nach } w.$$

Klar ist: $S(v,w) = S(v,w,n)$, für alle v, w.

Beispiel: (Die nicht eingezeichneten Kanten haben Länge ∞.)



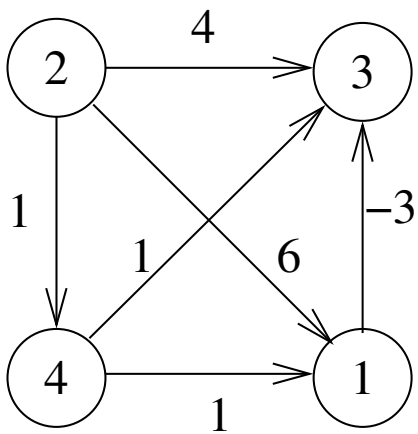
0-Weg von 2 nach 3: $(2, 3)$.

1-Wege: zusätzlich $(2, 1, 3)$.

2-Wege und 3-Wege: Keine Änderung; Endpunkte können auf Wegen nicht im Innern vorkommen.

4-Wege: zusätzlich $(2, 4, 3)$ und $(2, 4, 1, 3)$.

Beispiel: (Die nicht eingezeichneten Kanten haben Länge ∞.)



$S(2,3,0) = 4$
$S(2,3,1) = 3$
$S(2,3,2) = 3$
$S(2,3,3) = 3$
$S(2,3,4) = -1$

monoton fallend

0-Weg von 2 nach 3: $(2, 3)$.

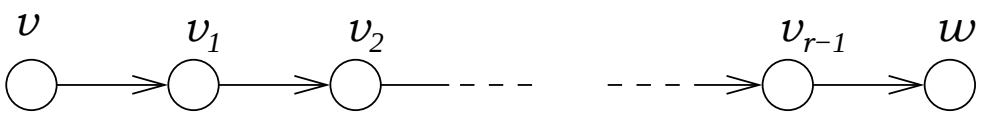
1-Wege: zusätzlich $(2, 1, 3)$.

2-Wege und 3-Wege: Keine Änderung; Endpunkte können auf Wegen nicht im Innern vorkommen.

4-Wege: zusätzlich $(2, 4, 3)$ und $(2, 4, 1, 3)$.

Wir entwickeln jetzt „Bellmansche Optimalitätsgleichungen“ für dieses Problem.

Für $k \geq 1$ betrachte k-Weg p von v nach w mit minimalen Kosten („**kürzester k-Weg**“):



Knoten k kommt im Inneren von p entweder einmal oder gar nicht vor.

1) Falls k in p **nicht** vorkommt, ist p kürzester $(k-1)$-Weg von v nach w.
2) Falls k im Inneren von p **vorkommt**, zerfällt p in zwei Teile

die beide kürzeste $(k-1)$-Wege sind. (Sonst könnte ein Teilweg durch einen billigeren ersetzt werden, im Widerspruch zur Optimalität von p.)

Punkte 1) und 2) drücken aus, dass Substrukturen (Teilwege) einer optimalen Lösung für ein Teilproblem (hier: kürzester k-Weg) selbst wieder optimal sind, für ein anderes Teilproblem.

Um beide Fälle zu erfassen, nehmen wir das Minimum der beiden Möglichkeiten.

Die **Bellmanschen Optimalitätsgleichungen** für den Algorithmus von Floyd-Warshall lauten dann:

$$S(v,w,k) = \min\{S(v,w,k-1), S(v,k,k-1) + S(k,w,k-1)\},$$
$$\text{für } 1 \leq v,w \leq n, 1 \leq k \leq n.$$

Basisfälle:

$S(v,v,0) = 0.$ (Weg (v), Länge 0.)

$S(v,w,0) = c(v,w)$, für $v \neq w.$ (Kante (v,w).)

Obgleich die Bellmanschen Optimalitätsgleichungen scheinbar ein rekursives Programm nahelegen, wird das Programm iterativ aufgeschrieben. Zur Aufbewahrung der Zwischenergebnisse benutzen wir ein Array $\mathtt{S}[1..n,1..n,0..n]$.

Eine iterative Programmstruktur mit einem Array als Datenstruktur für die Zwischenergebnisse ist ganz typisch für das Paradigma „dynamische Programmierung“. Naive Umsetzung in ein rekursives Programm würde dazu führen, dass Zwischenergebnisse wiederholt ausgerechnet werden, mit katastrophalen Folgen für die Rechenzeit.

Initialisierung:

$$\begin{aligned} &\mathtt{S}[v,v,0] \leftarrow 0, && 1 \le v \le n; \\ &\mathtt{S}[v,w,0] \leftarrow c(v,w), && 1 \le v,w \le n, \quad v \neq w. \end{aligned}$$

Der Algorithmus füllt dann das Array S gemäß wachsendem k aus:

```
for k from 1 to n do
  for v from 1 to n do
    for w from 1 to n do
      S[v,w,k] ← min{S[v,w,k-1], S[v,k,k-1] + S[k,w,k-1]}.
```

Korrektheit: Folgt aus der Vorüberlegung. Im Array S werden genau die Werte $S(v, w, k)$ gemäß den Basisbedingungen und den Bellmanschen Optimalitätsgleichungen berechnet.

Rechenzeit: Drei geschachtelte Schleifen: $\Theta(n^3)$.

Bei dieser simplen Implementierung wird noch Platz verschwendet, da für den k-ten Schleifendurchlauf offenbar nur die $(k-1)$-Komponenten des Arrays S benötigt werden.

Man überlegt sich leicht, dass man mit zwei Matrizen auskommt, einer mit den „alten“ Werten ($(k-1)$-Version) und einer mit den „neuen“ Werten (k-Version), zwischen denen dann immer passend umzuschalten ist.

Der Platzaufwand beträgt dann $O(n^2)$, der Zeitbedarf $O(n^3)$.

Wir können den Speicherplatzbedarf und die Berechnung selbst noch vereinfachen. Es gilt nämlich

$$S(v,k,k) = S(v,k,k-1) \text{ und } S(k,w,k) = S(k,w,k-1), \text{ für } 1 \leq v,w,k \leq n.$$

(k kann nicht im Inneren eines k-Weges von v nach k bzw. von k nach w vorkommen.)

Es ist also gleichgültig, ob man in der Iteration die alten Werte $S(v,k,k-1)$ und $S(k,w,k-1)$ oder die neuen Werte $S(v,k,k)$ und $S(k,w,k)$ benutzt. Man braucht also gar kein „altes“ und „neues“ Array, und erhält mit einem zweidimensionalen Array `S[1..n,1..n]`:

```
S[v,v] ← 0, für 1 ≤ v ≤ n;
S[v,w] ← c(v,w), für 1 ≤ v,w ≤ n, v ≠ w;
for k from 1 to n do
  for v from 1 to n do
    for w from 1 to n do
      S[v,w] ← min{S[v,w], S[v,k] + S[k,w]}.
```

Satz 12.1.1

Der bislang dargestellte Algorithmus berechnet in Zeit $O(n^3)$ und Platz $O(n^2)$ die Längen der kürzesten Wege zwischen allen Paaren von Knoten des gewichteten Digraphen $G = (V, E, c)$.

Wir wollen aber nicht nur die **Kosten** eines kürzesten Weges wissen, sondern auch zu gegebenen v, w einen **kürzesten Weg konstruieren** können.

Hierfür: Zusätzliches Array `I[1..n,1..n]`, in dem durch die Runden $k = 0, 1, \ldots, n$ hindurch für jedes Paar (v, w) die folgende Information gehalten wird:
Was ist der Knoten mit der größten Nummer, der für die Konstruktion eines kürzesten k-Weges von v nach w verwendet worden ist?

In Runde k: Wenn sich `S[`v, w`]` nicht ändert, bleibt `I[`v, w`]` gleich.
Wenn $S(v, w, k) < S(v, w, k-1)$, wird `I[`$v, w$`]` auf k gesetzt, weil Knoten k für einen kürzesten k-Weg notwendig ist.

Diese Erweiterung liefert den „vollen“ Floyd-Warshall-Algorithmus.

12.1.2 Algorithmus Floyd-Warshall(C[1..n,1..n])

Eingabe: C[1..n,1..n]: Matrix der Kantenkosten/-längen $c(v,w)$ in $\mathbb{R} \cup \{+\infty\}$

Ausgabe: S[1..n,1..n]: Kosten $S(v,w)$ eines kürzesten v-w-Weges

I[1..n,1..n]: minimaler maximaler Knoten auf einem kürzesten v-w-Weg

```
(1)   for v from 1 to n do
(2)     for w from 1 to n do
(3)       if v = w then S[v,v] ← 0; I[v,v] ← 0
(4)         else S[v,w] ← C[v,w];
(5)              if S[v,w] < ∞ then I[v,w] ← 0 else I[v,w] ← −1;
(6)   for k from 1 to n do
(7)     for v from 1 to n do
(8)       for w from 1 to n do
(9)         if S[v,k] + S[k,w] < S[v,w] then
(10)          S[v,w] ← S[v,k] + S[k,w]; I[v,w] ← k;
(11)  Ausgabe: S[1..n,1..n] und I[1..n,1..n].
```

Korrektheit: Eigentlich „nach Konstruktion“. Formal beweist man die folgende Schleifeninvariante:

Nach dem Schleifendurchlauf mit k in `k` ist $\mathtt{S}[v, w] = S(v, w, k)$.

(Durch vollständige Induktion über $k = 0, \ldots, n$, unter Ausnutzung der Bellman-Gleichungen und Eigenschaft $(*)$, der Abwesenheit von strikt negativen Kreisen.)

Zudem, wenn auch die „roten“ Teile für $\mathtt{I}[1..n, 1..n]$ ausgeführt werden:

$\mathtt{I}[v, w]$ ist das kleinste ℓ mit folgender Eigenschaft:
Unter den kürzesten Wegen von v nach w mit Zwischenknoten in $1, \ldots, k$ gibt es einen, dessen **maximaler** Eintrag ℓ ist.

Spezialfälle: $\mathtt{I}[v, v] = 0$ für alle v. Wenn $v \neq w$ und die Kante (v, w) ein kürzester Weg ist, dann ist $\mathtt{I}[v, w] = 0$; wenn $v \not\rightsquigarrow w$, ist $\mathtt{I}[v, w] = -1$.

(Beweis durch Induktion über $k = 0, \ldots, n$.)

Ein kürzester Weg von v nach w kann dann ausgehend von $\mathtt{I}[1..n, 1..n]$ mit einer einfachen rekursiven Prozedur „`printPathInner`“ gefunden werden.

12.1.2a Algorithmus printPathInner(v, w)
Global: `I[1..n,1..n]`: Matrix der „minimalen maximalen“ inneren Knoten
Eingabe: $v, w \in V$, $v \neq w$
Ausgabe: drucke Folge der „inneren Knoten“ eines kürzesten v-w-Weges, falls $v \rightsquigarrow w$.

```
(1)     k ← I[v,w];
(2)     if k > 0 then
(3)       printPathInner(v, k); print(k); printPathInner(k, w);
```

12.1.2b Algorithmus printPath(v, w)
Global: `I[1..n,1..n]`: Matrix der „minimalen maximalen“ inneren Knoten
Eingabe: $v, w \in V$
Ausgabe: drucke kürzesten v-w-Weg, falls $v \rightsquigarrow w$, sonst gib „kein Weg“ aus

```
(1)   if v = w
(2)     then print(v, „Länge 0“)
(3)     elseif I[v,w] < 0 then print(„kein Weg“)
(4)     else print(v); printPathInner(v, w); print(w);
```

Übung: (a) Man führe dies an einem Beispiel durch. (b) Man beweise, dass die Ausgabe korrekt ist.

12.1.3 Satz

Der Algorithmus von Floyd-Warshall löst das All-Pairs-Shortest-Paths-Problem in Rechenzeit $O(n^3)$ und Speicherplatz $O(n^2)$.

Das Ergebnis ist eine Datenstruktur der Größe $O(n^2)$, mit der sich zum Argument (v, w) mit Algorithmus **printPath** ein kürzester Weg von v nach w in Zeit $O(\#(\text{Kanten auf dem Weg}))$ ausgeben lässt.

Bemerkung: Wenn man den Algorithmus auf einer Eingabe ausführt, die Bedingung $(*)$ („es gibt keine negativen Kreise“) nicht erfüllt, dann wird am Ende mindestens einer der Werte $\mathtt{S}[v, v]$ negativ sein. Dies kann man als Kriterium für die Existenz negativer Kreise benutzen. Achtung: Wir werden in 12.2 eine andere Methode kennenlernen, um dies festzustellen.

Rekapitulation: Dynamische Programmierung (DP)

Algorithmenparadigma für **Optimierungsprobleme**. Typische Schritte:

- Gegeben eine Instanz, definiere (viele) **„Teilprobleme“**.
- Identifiziere einfache **Basisfälle** („kleine“, „leichte“ Teilprobleme).
- Formuliere Version der Eigenschaft

  Substrukturen optimaler Lösungen für Teilprobleme sind optimale Lösungen für (kleinere) Teilprobleme.

- Finde Rekursionsgleichungen für **Werte** optimaler Lösungen:

  Bellmansche Optimalitätsgleichungen

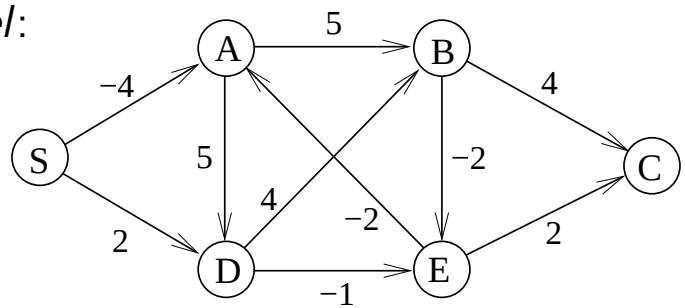
- Berechne optimale Werte **iterativ**. (Strukturen eventuell in separatem Durchgang.)

12.2 Der Bellman–Ford-Algorithmus

Zweck: Kürzeste Wege **von einem Startknoten** s aus
in gewichtetem Digraphen (V, E, c). (**SSSP**: „**S**ingle **S**ource **S**hortest **P**aths“).

Anders als beim Algorithmus von Dijkstra sind **negative Kantenkosten** zugelassen.

Beispiel:



$$c((\mathrm{S},\mathrm{A},\mathrm{B},\mathrm{C})) = 5,$$
$$c((\mathrm{S},\mathrm{D},\mathrm{E},\mathrm{C})) = 3,$$
$$c((\mathrm{S},\mathrm{A},\mathrm{D},\mathrm{E},\mathrm{C})) = 2,$$
$$c((\mathrm{S},\mathrm{A},\mathrm{B},\mathrm{E},\mathrm{C})) = 1 = d(\mathrm{S},\mathrm{C}).$$

Wir sagen, dass von $v_0 = s$ aus **ein Kreis mit negativer Gesamtlänge erreichbar ist**, wenn es einen Kantenzug $(v_0, \ldots, v_q, \ldots, v_r)$ gibt mit $q < r$ und $v_q = v_r$, wobei $c(v_q, v_{q+1}) + \ldots + c(v_{r-1}, v_r) < 0$ gilt. Wenn dies der Fall ist, kann man durch wiederholtes Durchlaufen des Kreises $(v_q, \ldots, v_r)$ Kantenzüge mit beliebig stark negativen Kosten von s nach v_r erhalten – die Frage nach einem „kürzesten Weg“ wird sinnlos.
In diesem Fall soll unser Algorithmus „negativer Kreis“ ausgeben.

Knotenmenge: $V = \{1, \ldots, n\}$, Startknoten $s \in V$,
Menge E von gerichteten Kanten, mit Kantengewichten $c\colon E \to \mathbb{R}$.

Das Format des Inputs ist recht frei:
Adjazenzlisten oder auch nur eine (unstrukturierte) Liste aller Kanten mit Gewichten.

Für die Überlegungen nehmen wir zuerst an, dass von s aus kein Kreis negativer Länge erreichbar ist.

Datenstrukturen: Distanzarray `dist`$[1..n]$
Vorgängerarray `p`$[1..n]$

Idee: `dist`$[v]$ enthält stets einen **Schätzwert** (eine obere Schranke) für die Länge eines kürzesten Weges von s nach v. Wert ∞ ist zugelassen, ist Startwert.

`p`$[v]$ enthält den direkten Vorgänger u von v auf einem Weg von s nach v der Länge $\leq$ `dist`$[v]$, wenn dies $< \infty$ ist. Dabei bilden die Kanten $(\texttt{p}[v], v)$ stets einen Baum, am Ende einen **Baum von kürzesten Wegen**.

Durch eine Reihe von Operationen sollen Werte `dist`$[v]$ und `p`$[v]$ sukzessive so gesetzt werden, dass sie kürzesten Wegen entsprechen.

Grundoperation, schon aus dem Dijkstra-Algorithmus bekannt:

update(u, v) // für $(u, v) \in E$

(1) **if** `dist[`u`]` $+ c(u, v) <$ `dist[`v`]` **then**

(2) `p[`v`]` $\leftarrow u$;

(3) `dist[`v`]` $\leftarrow$ `dist[`u`]` $+ c(u, v)$;

Folgende Aussage benutzen wir als Invariante:

$(*)$ Für alle $w \in V$ gilt: $d(s, w) \leq$ `dist[`w`]`.
D. h.: Wenn `dist[`w`]` $< \infty$, dann gibt es einen Weg von s nach w der Länge $\leq$ `dist[`w`]`.

Lemma 12.2.1

Wenn $(*)$ gilt und **update**(u, v) ausgeführt wird, dann gilt $(*)$ weiter.

(„Mit **update**(u, v) macht man nichts falsch.“)

Lemma 12.2.1

Wenn $(*)$ gilt und **update**(u, v) ausgeführt wird, dann gilt $(*)$ weiter.

Beweis: Sei $w \in V$, sei $(u, v) \in E$.

Wenn sich bei **update**(u, v) der Wert `dist[`w`]` nicht ändert, ist nichts zu zeigen.

Damit bleibt nur der Fall $v = w$ und `dist[`v`]` $< \infty$ (nach der **update**-Operation).

Dann muss `dist[`u`]` $< \infty$ gelten. Wir wissen nach Voraussetzung, dass $d(s, u) \leq$ `dist[`u`]` gilt, es also einen Weg p der Länge $\leq$ `dist[`u`]` von s nach u gibt.

Anhängen der Kante (u, v) an p liefert einen Kantenzug der Länge $\leq$ `dist[`u`]` $+ c(u, v)$ von s nach v.

Nach unserer vorläufigen Annahme kann dieser Kantenzug keinen negativen Kreis enthalten.

Also gibt es einen Weg von s nach v der Länge $\leq$ `dist[`u`]` $+ c(u, v)$. □

Algorithmischer Ansatz: Wir beginnen mit $\texttt{dist}[s] \leftarrow 0$ und $\texttt{dist}[v] \leftarrow \infty$ für alle anderen $v \in V$.

Dann gilt $(*)$.

Nun führen wir viele **update**(u, v)-Operationen aus (auch mehrfach an derselben Kante).

Lemma 12.2.2

$(s = v_0, v_1, \ldots, v_t = w)$ sei ein kürzester Weg von $s = v_0$ nach $w = v_t$ (also ohne Knotenwiederholung). Am Anfang gelte $\texttt{dist}[s] = 0$. In der Folge der ausgeführten **update**(u, v)-Operationen sollen

$$\textbf{update}(v_0, v_1), \ldots, \textbf{update}(v_{t-1}, v_t)$$

in dieser Reihenfolge vorkommen.

Dann gilt am Ende: $\texttt{dist}[w] = d(s, w)$.

Beweis: Durch Induktion über $i = 0, \ldots, t$ zeigen wir:
Nach Ausführung von **update**$(v_0, v_1), \ldots,$ **update**(v_{i-1}, v_i) gilt $\texttt{dist}[v_i] = d(s, v_i)$.

I.A.: $i = 0$. Es ist $v_0 = s$ und nach der Initialisierung gilt $\texttt{dist}[s] = 0 = d(s, s)$.

I.V.: Aussage ist richtig für $i - 1$.

I.S.: Nach I.V. gilt nach Ausführung von **update**$(v_0, v_1), \ldots,$ **update**(v_{i-2}, v_{i-1}) die Gleichung $\texttt{dist}[v_{i-1}] = d(s, v_{i-1})$.

Nun werden weitere **update**-Operationen und dann **update**(v_{i-1}, v_i) ausgeführt. Wegen Lemma 12.2.1 kann sich dabei $\texttt{dist}[v_{i-1}]$ nicht mehr ändern (verringern).

Nach der Operation **update**(v_{i-1}, v_i) gilt (wegen der Struktur der **update**-Operation):

$$\texttt{dist}[v_i] \leq \texttt{dist}[v_{i-1}] + c(v_{i-1}, v_i) \overset{\text{I.V.}}{=} d(s, v_{i-1}) + c(v_{i-1}, v_i) \overset{(+)}{=} d(s, v_i).$$

(Für $(+)$ wird benutzt, dass $(s = v_0, v_1, \ldots, v_\ell)$ ein kürzester Weg von s nach v_ℓ ist, für $\ell = i - 1$ und $\ell = i$. Dies muss so sein, weil $(v_0, \ldots, v_\ell)$ ein Teil eines kürzesten Weges von s nach w ist.)
Mit Lemma 12.2.1 folgt $\texttt{dist}[v_i] = d(s, v_i)$. □

12.2.3 Algorithmus Bellman-Ford$(((V, E, c), s))$

Eingabe: (V, E, c): Digraph mit Kantengewichten in $\mathbb{R}$, $s \in V$: Startknoten

Ausgabe: Wenn G keine Kreise mit negativem Gewicht hat:
In `dist[`v`]`: Abstand $d(s, v)$;
In `p[`$1..n$`]`: Baum von kürzesten Wegen (wie bei Dijkstra);

Initialisierung:

(1) **for** `v` **from** 1 **to** n **do**
(2) `dist[v]` $\leftarrow \infty$; `p[v]` $\leftarrow -1$;
(3) `dist[`s`]` $\leftarrow 0$; `p[`s`]` $\leftarrow -2$; // wird nie geändert, wenn kein negativer Kreis existiert

Hauptschleife:

(4) **for** `i` **from** 1 **to** $n-1$ **do** // heißt nur: wiederhole $(n-1)$-mal
(5) **forall** $(u, v) \in E$ **do update**(u, v); // beliebige Reihenfolge

Zyklentest:

(6) **forall** $(u, v) \in E$ **do**
(7) **if** `dist[`u`]` $+ c(u, v) <$ `dist[`v`]` **then return** „negativer Kreis“;

Ergebnis:

(8) **return** `dist[`$1..n$`]`, `p[`$1..n$`]`.

Satz 12.2.4

Der Bellman-Ford-Algorithmus hat folgendes Verhalten:

(a) Wenn es keinen von s aus erreichbaren Kreis mit negativer Gesamtlänge gibt („negativer Kreis“), steht am Ende des Algorithmus in $\texttt{dist}[v]$ die Länge eines kürzesten Weges von s nach v, und die Kanten $(\texttt{p}[w], w)$ (mit $\texttt{p}[w] > 0$) bilden einen Baum der kürzesten Wege von s aus zu den erreichbaren Knoten.

(b) Der Algorithmus gibt „negativer Kreis“ aus genau dann wenn es einen von s aus erreichbaren negativen Kreis gibt.

(c) Die Rechenzeit ist $O(nm)$.

Beweis: (c) ist klar.

(a) Da es keine negativen Kreise gibt, die von s aus erreichbar sind, kann durch Herausschneiden von Kreisen aus Kantenzügen, die in s starten, die Länge nicht steigen.

Wie in 12.1 folgert man, dass es für jeden von s aus erreichbaren Knoten v einen kürzesten **Weg** von s nach v gibt, auf dem sich also keine Knoten wiederholen. Dieser Weg hat höchstens $n-1$ Kanten. Damit ist auch das Konzept „Länge $d(s,v)$ eines kürzesten Weges von s nach v“ wohldefiniert.

Sei $v \in V$ mit $d(s,v) < \infty$ beliebig. Wähle einen kürzesten Weg $(s = v_0, \ldots, v_t = v)$ von s nach v. Dann ist $t \leq n-1$.

Wir können Lemma 12.2.2 auf den Ablauf des Algorithmus anwenden, da die Initialisierung ist wie dort vorgeschrieben und auf die Kanten $(v_0, v_1), \ldots, (v_{t-1}, v_t)$ nacheinander die **update**-Operation angewendet wird.

Also gilt am Ende $\texttt{dist}[v] = d(s,v)$.

Der Beweis der Behauptung über die $(\texttt{p}[v], v)$-Kanten ist nicht ganz einfach.

Er ist **nicht prüfungsrelevant**, aber für Interessierte auf den Druckfolien dargestellt.

Behauptung: Die p-Werte beschreiben einen Kürzeste-Wege-Baum.

(i) Die Kanten $(\texttt{p}[w], w)$ mit $\texttt{p}[w] \geq 1$ bilden zu jedem Zeitpunkt einen **Baum** T mit Wurzel s, der alle Knoten w mit $0 \leq \texttt{dist}[w] < \infty$ enthält.

Dies wird durch Induktion über update-Operationen gezeigt.

Am Anfang ist es wahr – T hat nur den Knoten s, und es gibt keine Kante $(\texttt{p}[w], w)$ mit $0 \leq \texttt{dist}[w] < \infty$.

Bei einer $\texttt{update}(u, v)$-Operation, die $\texttt{p}[v]$ verändert, gibt es zwei Möglichkeiten. Entweder wird Knoten v als neues Blatt an T angehängt (nämlich wenn vorher $\texttt{p}[v] = -1$ war) oder der Unterbaum mit Wurzel v wird von seinem bisherigen Vorgänger abgehängt und als Unterbaum an u angehängt.

Wichtig: Wir müssen uns vergewissern, dass es nicht möglich ist, dass in dieser Situation u selbst im Unterbaum unter v sitzt. (Sonst würden wir einen Kreis erzeugen, und die $(\texttt{p}[w], w)$-Kanten wären kein Baum mehr.) Annahme: $(v = v_0, v_1, \ldots, v_r = u)$ ist ein Weg mit $\texttt{p}[v_i] = v_{i-1}$ für $i = 1, \ldots, r$, es wird $\texttt{update}(u, v)$ ausgeführt, wobei sich $\texttt{dist}[u] + c(u, v) < \texttt{dist}[v]$ ergibt. Wir haben $\texttt{dist}[v_{i-1}] + c(v_{i-1}, v_i) \leq \texttt{dist}[v_i]$ für $i = 1, \ldots, r$. (Das gilt, weil zum Zeitpunkt, wo $\texttt{p}[v_i] = v_{i-1}$ gesetzt wurde, Gleichheit herrscht und sich $\texttt{dist}[v_i]$ nachher nicht mehr geändert hat.)

Die Ungleichung für u und v kann man auch als $\texttt{dist}[v_r] + c(v_r, v_0) < \texttt{dist}[v_0]$ lesen. Wenn wir nun diese $r+1$ Ungleichungen addieren, dann fallen sämtliche $\texttt{dist}[v_i]$-Werte heraus, und es bleibt $c(v_0, v_1) + \cdots + c(v_{r-1}, v_r) + c(v_r, v_0) < 0$ stehen. Das heißt aber, dass $(v = v_0, v_1, \ldots, v_r, v_0)$ ein von s aus erreichbarer negativer Kreis ist, im Widerspruch zu unserer Annahme.

(ii) Wenn am Ende $p = (s = v_0, v_1, v_2, \ldots, v_t = v)$ ein Weg im Baum T ist, dann gilt $d(s, v) = \texttt{dist}[v] = \sum_{1 \le i \le t} c(v_{i-1}, v_i)$. D. h.: p ist ein kürzester Weg von s nach v.
(In dem Moment, in dem $\texttt{dist}[v_i]$ in der Operation $\texttt{update}(v_{i-1}, v_i)$ seinen endgültigen Wert erhält und $\texttt{p}[v_i]$ auf v_{i-1} gesetzt wird, gilt $\texttt{dist}[v_i] = \texttt{dist}[v_{i-1}] + c(v_{i-1}, v_i)$. Nach diesem Zeitpunkt kann sich nur noch $\texttt{dist}[v_{i-1}]$ verringern, also gilt am Ende $\texttt{dist}[v_i] \ge \texttt{dist}[v_{i-1}] + c(v_{i-1}, v_i)$.)
Addition dieser Ungleichungen, für $1 \le i \le t$, ergibt $d(s, v) = \texttt{dist}[v] \ge \sum_{1 \le i \le t} c(v_{i-1}, v_i)$.
Das bedeutet, dass p ein Weg der Länge $d(s, v)$ ist, also optimal ist. □

Beweis von (b): „$\Rightarrow$“: Indirekt. Annahme: Von s aus ist kein negativer Kreis erreichbar (also ist $d(s,v)$ definiert für alle v), aber in Zeile (7) wird ein v mit $\texttt{dist}[u] + c(u,v) < \texttt{dist}[v]$ gefunden. – Nach (a) folgt $d(s,u) + c(u,v) < d(s,v)$, was offensichtlich unmöglich ist.

„$\Leftarrow$“: Sei $(s = v_0, \ldots, v_r, \ldots, v_t)$ mit $r < t$ und $v_r = v_t$ ein Kantenzug, der mit einem Kreis endet. Wenn der Test in Zeile (7) stets „$\geq$“ ergibt, gilt:

$$\begin{aligned}
\texttt{dist}[v_r] + c(v_r, v_{r+1}) &\geq \texttt{dist}[v_{r+1}] \\
\texttt{dist}[v_{r+1}] + c(v_{r+1}, v_{r+2}) &\geq \texttt{dist}[v_{r+2}] \\
&\vdots \\
\texttt{dist}[v_{t-1}] + c(v_{t-1}, v_t) &\geq \texttt{dist}[\underbrace{v_t}_{=v_r}]
\end{aligned}$$

Addition dieser Ungleichungen und Vereinfachen liefert:

$$c(v_r, v_{r+1}) + \cdots + c(v_{t-1}, v_t) \geq 0.$$

Das bedeutet: Jeder von s aus erreichbare Kreis hat nichtnegative Kosten. □

Bemerkung: Die Entwicklung des Bellman-Ford-Algorithmus folgt nur scheinbar nicht dem DP-Paradigma (Teilprobleme, Basisfälle, Bellmansche Gleichungen). Dies sieht man wie folgt:

Wir definieren, für $v \in V$ und $0 \leq k < n$:

$$S(v,k) := d_k(s,v) := \min(\text{Länge eines } s\text{-}v\text{-Weges mit} \leq k \text{ Kanten}).$$

Damit: Basisfälle $S(s,0) = 0$ und $S(v,0) = \infty$ für $v \neq s$.

Bellmansche Optimalitätsgleichungen:
Sei $k \geq 1$. Für einen kürzesten s-v-Weg p mit $\leq k$ Kanten gibt es zwei Fälle:

- p hat $\leq k-1$ Kanten. Dann hat er Länge $S(v,k-1)$.
- p hat k Kanten. Sei w der vorletzte Knoten auf p. Dann hat p Länge $S(w,k-1) + c(w,v)$.

Da w nicht bekannt ist, muss man über alle Möglichkeiten minimieren, und man erhält:

$$S(v,k) = \min(\{S(v,k-1)\} \cup \{S(w,k-1) + c(w,v) \mid (w,v) \in E\}).$$

Dies führt zur folgenden „DP-Version“ des Algorithmus von Bellman und Ford.

12.2.3 Algorithmus Bellman-Ford (DP-Version) $(((V, E, c), s))$

Eingabe, Ausgabe: wie vorher

Datenstruktur: `dist`$[1..n, 0..n-1]$: `real`; `p`$[1..n]$: `integer`;

Initialisierung:

(1) **for** `v` **from** 1 **to** n **do**

(2) `dist[v,0]` $\leftarrow \infty$; `p[v]` $\leftarrow -1$;

(3) `dist`$[s, 0] \leftarrow 0$; `p`$[s] \leftarrow -2$;

Hauptschleife:

(4) **for** `k` **from** 1 **to** $n-1$ **do**

(5) **forall** `v` $\in V$ **do**

(6) `dist[v,k]` $\leftarrow$ `dist[v,k-1]`

(7) **forall** $(\texttt{w}, \texttt{v}) \in E$ **do**

(8) **if** `dist[w,k-1]` $+ c(\texttt{w}, \texttt{v}) <$ `dist[v,k]` **then**

(9) „aktualisiere `dist[v,k]` und `p[v]`“.

Ergebnis:

(10) **return** `dist`$[1..n, n-1]$, `p`$[1..n]$.

Aktualisierungen (4)–(9) wie im gewöhnlichen Bellman-Ford-Algorithmus, aber in spezieller Reihenfolge und „schichtenweise“ über $k = 1, \ldots, n-1$. Zwei Ebenen des `dist`-Arrays genügen.

Alternativer Ansatz (zu Floyd-Warshall) zur Lösung des APSP-Problems in Digraphen ohne negative Kreise:

(1) Löse **eine** **SSSP-Instanz** in G mit neuem Knoten s per **Bellman-Ford-Algorithmus**. – Kosten: $O(mn)$.

(2) Definiere modifizierte Kantengewichte $\bar{c}(v,w) \geq 0$, $(v,w) \in E$.

(3) Löse n **viele SSSP-Instanzen** in G mit $\bar{c}$, mittels **Dijkstra-Algorithmus**! Kosten (mit Binärheaps): $n \cdot O(m \log n) = O(mn \log n)$.

Gesamtkosten: $O(mn \log n)$.

Viel besser als das $O(n^3)$ des Floyd-Warshall-Algorithmus, wenn $m \ll n^2 / \log n$.

Details:

Zu (1): Füge zu $G = (V, E, c)$ einen **neuen Knoten** s und alle Kanten (s, v), $v \in V$, hinzu und setze $c(s, v) := 0$, für $v \in V$. Dies ergibt erweiterten Digraphen $G' = (V \cup \{s\}, E', c')$.

Anwenden des Algorithmus von Bellman und Ford auf G' liefert $\pi(v) := d_{G'}(s, v)$, für $v \in V$ (und prüft, ob es in G negative Kreise gibt).

Zu (2): Für $(v, w) \in E$ definiere:

$$\bar{c}(v, w) := c(v, w) + \pi(v) - \pi(w).$$

(Man sagt: Modifiziere c mittels „Knotenpotenzialen“ π.)

Für eine Kante $(v, w) \in E$ beobachten wir:

$\pi(w) = d_{G'}(s, w) \leq d_{G'}(s, v) + c(v, w) = \pi(v) + c(v, w)$, also $\bar{c}(v, w) \geq 0$.

Behauptung: Wenn $p = (v = v_0, v_1, \ldots, v_t = w)$ ein beliebiger v-w-Weg in G ist, dann gilt für die „c-Länge“ $c(p)$ und die „$\bar{c}$-Länge“ $\bar{c}(p)$:

$$\bar{c}(p) = c(p) + \pi(v) - \pi(w).$$

Beweis: Mit Teleskopsumme:

$$\begin{aligned}\bar{c}(p) = \sum_{1 \le i \le t} \bar{c}(v_{i-1}, v_i) &= \sum_{1 \le i \le t} (c(v_{i-1}, v_i) + \pi(v_{i-1}) - \pi(v_i)) \\ &= \sum_{1 \le i \le t} c(v_{i-1}, v_i) + \pi(v_0) - \pi(v_t) = c(p) + \pi(v_0) - \pi(v_t).\end{aligned}$$

D. h.: Für v-w-Wege p und p' gilt $c(p) \le c(p')$ genau dann wenn $\bar{c}(p) \le \bar{c}(p')$.

Insbesondere: Kürzeste Wege in G bezüglich c sind dieselben wie kürzeste Wege bezüglich $\bar{c}$.

Zu (3): Für jeden Knoten $v \in V$ gilt: Ein Aufruf des Dijkstra-Algorithmus liefert alle Distanzen $\bar{d}(v, w)$ zwischen Knoten in G mit $\bar{c}$ und einen Kürzeste-Wege-Baum für Startknoten v.

c-Distanzen: $d_G(v, w) = \bar{d}(v, w) - \pi(v) + \pi(w)$, für $(v, w) \in V \times V$.

Der Kürzeste-Wege-Baum mit Startknoten v für c ist derselbe wie für $\bar{c}$.

Diese Konstruktion wird mit jedem $v \in V$ als Startknoten ausgeführt.

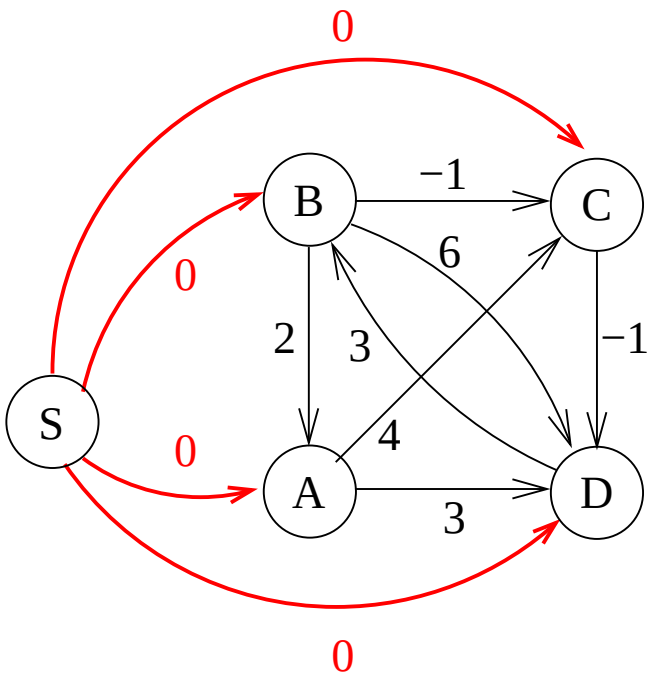
Dies liefert alle Distanzen $d_G(v, w)$ und n viele Kürzeste-Wege-Bäume T_v, $v \in V$.

Rechenzeit: $O(nm \log n)$.

Um einen kürzesten Weg von v nach w zu finden, benutze T_v.

Rechenzeit: $O(\#(\text{Kanten auf einem kürzesten } v\text{-}w\text{-Weg}))$.

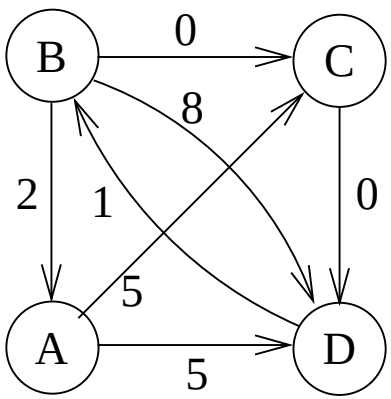
Beispiel:



(1) Bellman-Ford angewendet auf den um S erweiterten Graphen liefert:

$\pi(\mathrm{A}) = 0,\ \pi(\mathrm{B}) = 0,\ \pi(\mathrm{C}) = -1,\ \pi(\mathrm{D}) = -2.$

Auf den Graphen $\bar{G} = (V, E, \bar{c})$ mit Kantenlängen $\bar{c}(v, w)$
(und $\pi(\mathrm{A}) = 0$, $\pi(\mathrm{B}) = 0$, $\pi(\mathrm{C}) = -1$, $\pi(\mathrm{D}) = -2$)



wendet man viermal den Algorithmus von Dijkstra an, mit Startknoten A, B, C und D. – Einige Ergebnisse:

$\bar{d}(\mathrm{B}, \mathrm{C}) = 0$, $\bar{d}(\mathrm{B}, \mathrm{D}) = 0$, $\bar{d}(\mathrm{B}, \mathrm{A}) = 2$, $\bar{d}(\mathrm{C}, \mathrm{A}) = 3$.

Durch Korrektur $d(v, w) = \bar{d}(v, w) - \pi(v) + \pi(w)$ erhält man:

$d(\mathrm{B}, \mathrm{C}) = 0-0+(-1) = -1$, $d(\mathrm{B}, \mathrm{D}) = 0-0+(-2) = -2$, $d(\mathrm{B}, \mathrm{A}) = 2-0+0 = 2$, $d(\mathrm{C}, \mathrm{A}) = 3 - (-1) + 0 = 4$.

12.3 Editierdistanz

Problemstellung: Sei A ein Alphabet. (*Bsp.*: Lat. Alphabet, ASCII-Alphabet, {A,C,G,T}.)

Wenn $x = a_1 \dots a_m \in A^*$ und $y = b_1 \dots b_n \in A^*$ zwei Zeichenreihen über A sind, möchte man herausfinden, wie **ähnlich** (oder unähnlich) sie sind.

„`Arbeit`“ und „`Freizeit`“ sind nicht identisch, aber intuitiv einander ähnlicher als „`Informatik`“ und „`Camping`“. Wie können wir „Ähnlichkeit“ messen?

Wir definieren Elementarschritte (Editier-Operationen), die einen String verändern:

- Lösche einen Buchstaben aus einem String: Aus uav wird uv $(u, v \in A^*, a \in A)$.
  Beispiel: `Haut` $\rightarrow$ `Hut`.
- Füge einen Buchstaben in einen String ein: Aus uv wird uav $(u, v \in A^*, a \in A)$.
  Beispiel: `Hut` $\rightarrow$ `Haut`.
- Ersetze einen Buchstaben: Aus uav wird ubv $(u, v \in A^*, a, b \in A)$.
  Beispiel: `Haut` $\rightarrow$ `Hast`.

Der „Abstand“ oder die **Editierdistanz $d(x, y)$** von x und y ist die minimale Anzahl von Editieroperationen, die benötigt werden, um x in y zu transformieren.

Offenbar: $d(x, y) = d(y, x)$. (Einfügen/Löschen invers zueinander, Ersetzen invers zu Ersetzen.)

Überlege: Äquivalenz Transformation/Zuordnung:

A	r	b	e	i	-	-	-	t		
F	r	-	e	i	z	e	i	t		
I	n	f	o	r	m	a	t	-	i	k	-
C	-	-	-	-	-	a	m	p	i	n	g

Man schreibt Strings aus Buchstaben und dem Sonderzeichen - untereinander, wobei die beiden Zeilen jeweils das Wort x bzw. y ergeben, wenn man die -'s ignoriert.
Die **Kosten** einer solchen Anordnung: Die Anzahl der Positionen, an denen die oberen und unteren Einträge nicht übereinstimmen.

Die Kosten, die eine solche Anordnung erzeugt, sind gleich der Anzahl der Editierschritte in einer Transformation. (Im Beispiel: 5 bzw. 10.) Daher: Die **minimalen** Kosten einer solchen Anordnung sind die Editierdistanz. (Im Beispiel: 4 (!) bzw. 10.)

Das Problem „Editierdistanz“ wird folgendermaßen beschrieben:

Input: Strings $x = a_1 \dots a_m$, $y = b_1 \dots b_n$ aus A^*.

Aufgabe: Berechne $d(x, y)$ (und eine Editierfolge, die x in y transformiert).

Ansatz: Dynamische Programmierung

Unser Beispiel: $x = \texttt{Exponentiell}$ und $y = \texttt{Polynomiell}$.

1. Schritt: Relevante **Teilprobleme** identifizieren!

Betrachte Präfixe $x[1..i] = a_1 \dots a_i$ und $y[1..j] = b_1 \dots b_j$, und setze

$$\boxed{\boldsymbol{E(i,j)} := d(x[1..i], y[1..j]), \text{ für } 0 \leq i \leq m,\ 0 \leq j \leq n}$$

Im Beispiel bedeutet $\boldsymbol{E(7,6)}$: Berechne $d(\texttt{Exponen}, \texttt{Polyno})$
(wegen $x[1..7] = \texttt{Exponen}$ und $y[1..6] = \texttt{Polyno}$).

Um Bellmansche Gleichungen zu erhalten, betrachten wir Beziehungen zwischen Teilproblemen. Dabei benutzen wir die Vorstellung der Anordnungen von Wörtern untereinander wie auf Folie 38.

Betrachte nichtleere Präfixe $x[1..i] = a_1 \dots a_i$ und $y[1..j] = b_1 \dots b_j$, mit $1 \leq i \leq m$, $1 \leq j \leq n$. Wenn $x[1..i]$ und $y[1..j]$ optimal untereinander angeordnet sind, mit Kosten $E(i,j)$, gibt es drei Möglichkeiten für die Behandlung der letzten Stelle:

1. Fall (Ersetzung)	2. Fall (Löschung)	3. Fall (Einfügung)
$x[1..i-1]$ a_i	$x[1..i-1]$ a_i	$x[1..i]$ –
$y[1..j-1]$ b_j	$y[1..j]$ –	$y[1..j-1]$ b_j

1. Fall (Ersetzung)	2. Fall (Löschung)	3. Fall (Einfügung)
$x[1..i-1]$ a_i	$x[1..i-1]$ a_i	$x[1..i]$ -
$y[1..j-1]$ b_j	$y[1..j]$ -	$y[1..j-1]$ b_j

1. Fall: Die letzte Stelle kostet $\text{diff}(a_i, b_j) := [a_i \neq b_j]$ $\left(= \begin{cases} 1 & \text{falls } a_i \neq b_j \\ 0 & \text{falls } a_i = b_j \end{cases}\right)$

$x[1..i-1]$ und $y[1..j-1]$ sind einander optimal zugeordnet („optimale Substruktur“).

$$\Rightarrow \quad E(i,j) = E(i-1, j-1) + \text{diff}(a_i, b_j).$$

2. Fall: In der letzten Stelle erfolgt eine Löschung, mit Kosten 1.

$x[1..i-1]$ und $y[1..j]$ sind einander optimal zugeordnet.

$$\Rightarrow \quad E(i,j) = E(i-1, j) + 1.$$

3. Fall: In der letzten Stelle erfolgt eine Einfügung, mit Kosten 1.

$x[1..i]$ und $y[1..j-1]$ sind einander optimal zugeordnet.

$$\Rightarrow \quad E(i,j) = E(i, j-1) + 1.$$

Unter den drei möglichen Fällen ist derjenige (sind diejenigen) mit den kleinsten Kosten relevant. Wir erhalten zusammengefasst:

Bellmansche Optimalitätsgleichungen für Editierdistanz:

$$E(i,j) = \min\{E(i-1,j-1) + \mathsf{diff}(a_i, b_j), E(i-1,j) + 1, E(i,j-1) + 1\}.$$

Basisfälle: Leere Präfixe $\varepsilon = x[1..0]$ und $\varepsilon = y[1..0]$.

Klar: $E(i,0) = d(x[1..i], \varepsilon) = i$, für $0 \leq i \leq m$, und
$E(0,j) = d(\varepsilon, y[1..j]) = j$, für $0 \leq j \leq n$.

(Man muss alle Buchstaben streichen bzw. einfügen.)

Die Zahlen $E(i,j)$ berechnen wir iterativ, indem wir sie in eine Matrix $\texttt{E}[0..m,0..n]$ eintragen. Dies liefert den DP-Algorithmus für die Editierdistanz.

Initialisierung:

$\texttt{E}[i,0] \leftarrow i$, für $i = 0, \ldots, m$;
$\texttt{E}[0,j] \leftarrow j$, für $j = 0, \ldots, n$.

Dann füllen wir die Matrix (z. B.) zeilenweise aus, genau nach den Bellmanschen Optimalitätsgleichungen:

for $\texttt{i}$ **from** 1 **to** m **do**
    for $\texttt{j}$ **from** 1 **to** n **do**
        $\texttt{E[i,j]} \leftarrow \min\{\texttt{E[i-1,j-1]} + \text{diff}(a_{\texttt{i}}, b_{\texttt{j}}), \texttt{E[i-1,j]} + 1, \texttt{E[i,j-1]} + 1\}$;
return $\texttt{E}[m,n]$.

Rechenzeit: $O(m \cdot n)$

Bemerkung: Andere Reihenfolgen für das Ausfüllen sind möglich. Für die Ausführung von Hand ist es geschickt, zuerst die Werte (i,j) mit $\text{diff}(a_i, b_j) = 0$, also $a_i = b_j$ zu identifizieren (rot, unterstrichen).

E		P	o	l	y	n	o	m	i	e	l	l
	0	1	2	3	4	5	6	7	8	9	10	11
E	1											
x	2											
p	3											
o	4		–				–					
n	5					–						
e	6									–		
n	7					–						
t	8											
i	9								–			
e	10									–		
l	11			–							–	–
l	12			–							–	–

Basiswerte $E(i, 0)$ und $E(0, j)$, Markierungen für diff$(a_i, b_j) = 0$.

E		P	o	l	y	n	o	m	i	e	l	l
	0	1	2	3	4	5	6	7	8	9	10	11
E	1	1	2	3	4	5	6	7	8	9	10	11
x	2	2	2	3	4	5	6	7	8	9	10	11
p	3	3	3	3	4	5	6	7	8	9	10	11
o	4	4	–				–					
n	5					–						
e	6									–		
n	7					–						
t	8											
i	9								–			
e	10									–		
l	11			–							–	–
l	12			–							–	–

Zeilenweise ausgefüllt bis $E(4, 1)$.

E		P	o	l	y	n	o	m	i	e	l	l
	0	1	2	3	4	5	6	7	8	9	10	11
E	1	1	2	3	4	5	6	7	8	9	10	11
x	2	2	2	3	4	5	6	7	8	9	10	11
p	3	3	3	3	4	5	6	7	8	9	10	11
o	4	4	**3**				–					
n	5					–						
e	6									–		
n	7					–						
t	8											
i	9								–			
e	10									–		
l	11			–							–	–
l	12			–							–	–

$E(4,2) = \min\{3+0, 3+1, 4+1\} = 3.$

E		P	o	l	y	n	o	m	i	e	l	l
	0	1	2	3	4	5	6	7	8	9	10	11
E	1	1	2	3	4	5	6	7	8	9	10	11
x	2	2	2	3	4	5	6	7	8	9	10	11
p	3	3	3	3	4	5	6	7	8	9	10	11
o	4	4	3	4			–					
n	5					–						
e	6									–		
n	7					–						
t	8											
i	9								–			
e	10									–		
l	11			–							–	–
l	12			–							–	–

$E(4,3) = \min\{3+1, 3+1, 3+1\} = 4.$

E		P	o	l	y	n	o	m	i	e	l	l
	0	1	2	3	4	5	6	7	8	9	10	11
E	1	1	2	3	4	5	6	7	8	9	10	11
x	2	2	2	3	4	5	6	7	8	9	10	11
p	3	3	3	3	4	5	6	7	8	9	10	11
o	4	4	**3**	4	4	5	**5**	6	7	8	9	10
n	5	5	4	4	5	**4**	5	6	7	8	9	10
e	6	6	5	5	5	5	5	6	7	**7**	8	9
n	7	7	6	6	6	**5**	6	6	7	8	8	9
t	8	8	7	7	7	6	6	7	7	8	9	9
i	9	9	8	8	8	7	7	7	**7**	8	9	10
e	10	10	9	9	9	8	8	8	8	**7**	8	9
l	11	11	10	**9**	10	9	9	9	9	8	**7**	**8**
l	12	12	11	**10**	10	10	10	10	10	9	**8**	**7**

Matrix ist komplett ausgefüllt. Ergebnis: $d(x, y) = E(12, 11) = 7$, rechte untere Ecke!

Ermittlung der Editier-Operationen in Zeit $O(m+n)$

Gegeben x und y, möchte man natürlich nicht nur die Editierdistanz $d(x,y)$, sondern auch die Schritte einer Transformation ermitteln, die mit $d(x,y)$ Schritten x in y überführt.
Wenn die Matrix `E[0..m,0..n]` mit den Werten $E(i,j)$ vorliegt, ist das leicht:

Wir wandern einen Weg von (m,n) nach $(0,0)$. Dabei gehen wir von (i,j) nach

$$\begin{array}{ll} (i-1,j-1), & \text{falls } E(i,j) = E(i-1,j-1) + \mathsf{diff}(a_i,b_j), \\ (i-1,j), & \text{falls } E(i,j) = E(i-1,j) + 1, \\ (i,j-1), & \text{falls } E(i,j) = E(i,j-1) + 1. \end{array}$$

Wenn mehr als eine Gleichung zutrifft, können wir einen der möglichen Schritte beliebig wählen.
Der gefundene Weg endet in $(0,0)$. Diesen Weg lesen wir dann rückwärts von $(0,0)$ nach (m,n). Dieser Weg repräsentiert in offensichtlicher Weise eine Transformation von x in y, wobei man Buchstaben gleich lässt, ersetzt, einfügt oder löscht. Man liest $x = x[1..m]$ von links nach rechts und generiert $y[1..n]$ von links nach rechts.

(1) $(i-1,j-1) \xrightarrow{b} (i,j)$ entspricht dem Übernehmen von a_i $(b=0)$ oder der Ersetzung durch b_j $(b=1)$;

(2) $(i-1,j) \xrightarrow{1} (i,j)$ entspricht dem Streichen von a_i;

(3) $(i,j-1) \xrightarrow{1} (i,j)$ entspricht dem Einfügen von b_j.

E		P	o	l	y	n	o	m	i	e	l	l
	0	1	2	3	4	5	6	7	8	9	10	11
E	**1**	1	2	3	4	5	6	7	8	9	10	11
x	**2**	2	2	3	4	5	6	7	8	9	10	11
p	3	**3**	3	3	4	5	6	7	8	9	10	11
o	4	4	**3**	4	4	5	**5**	6	7	8	9	10
n	5	5	4	**4**	5	**4**	5	6	7	8	9	10
e	6	6	5	5	**5**	5	5	6	7	**7**	8	9
n	7	7	6	6	6	**5**	6	6	7	8	8	9
t	8	8	7	7	7	6	**6**	**7**	7	8	9	9
i	9	9	8	8	8	7	7	7	**7**	8	9	10
e	10	10	9	9	9	8	8	8	8	**7**	8	9
l	11	11	10	**9**	10	9	9	9	9	8	**7**	**8**
l	12	12	11	**10**	10	10	10	10	10	9	**8**	**7**

Im Beispiel: $(0,0) \xrightarrow{1} (1,0) \xrightarrow{1} (2,0) \xrightarrow{1} (3,1) \xrightarrow{0} (4,2) \xrightarrow{1} (5,3) \xrightarrow{1} (6,4) \xrightarrow{0} (7,5) \xrightarrow{1} (8,6) \xrightarrow{1} (8,7) \xrightarrow{0} (9,8) \xrightarrow{0} (10,9) \xrightarrow{0} (11,10) \xrightarrow{0} (12,11)$, Kosten 7.

Dies führt zu der folgenden Zuordnung/Transformation, wobei die sieben „kostenpflichtigen“ Positionen rot markiert sind:

E	x	p	o	n	e	n	t	-	i	e	l	l
-	-	P	o	l	y	n	o	m	i	e	l	l

12.4 Optimale Matrizenmultiplikation

Multiplikation von zwei Matrizen über einem Ring R: Gegeben

$$A_1 = \begin{pmatrix} \alpha_{11} & \cdots & \alpha_{1r_1} \\ \vdots & \vdots & \vdots \\ \alpha_{r_0 1} & \cdots & \alpha_{r_0 r_1} \end{pmatrix} \in R^{r_0 \times r_1}, A_2 = \begin{pmatrix} \beta_{11} & \cdots & \beta_{1r_2} \\ \vdots & \vdots & \vdots \\ \beta_{r_1 1} & \cdots & \beta_{r_1 r_2} \end{pmatrix} \in R^{r_1 \times r_2},$$

berechne Produkt $C := A_1 \cdot A_2$ mit

$$C = \begin{pmatrix} \gamma_{11} & \cdots & \gamma_{1r_2} \\ \vdots & \vdots & \vdots \\ \gamma_{r_0 1} & \cdots & \gamma_{r_0 r_2} \end{pmatrix} \in R^{r_0 \times r_2}, \text{ wobei } \gamma_{ik} = \sum_{1 \le j \le r_1} \alpha_{ij}\beta_{jk}, \text{ für } 1 \le i \le r_0,\ 1 \le k \le r_2.$$

Standardmethode benötigt $r_0 r_1 r_2$ R-Multiplikationen, $r_0(r_1 - 1)r_2$ R-Additionen.
Gesamtrechenzeit: $\Theta(r_0 r_1 r_2)$. **„Kosten“: $c(r_0, r_1, r_2) := r_0 r_1 r_2$.**

Wir betrachten „Iterierte Matrixmultiplikation“:

Gegeben sind $A_1, A_2, \ldots, A_k$, wobei A_i eine $r_{i-1} \times r_i$-Matrix über R ist.

Aufgabe: Berechne $A_1 \cdot A_2 \cdot \cdots \cdot A_k$, **möglichst kostengünstig**.

Die Matrizenmultiplikation ist **assoziativ**.

Beispiel $k = 3$: Wir betrachten die Anzahl der R-Multiplikationen, die bei der Berechnung von $A_1A_2A_3$ anfallen:

Kosten $r_0r_1r_2 + r_0r_2r_3$ bei der Klammerung $(A_1A_2)A_3$ und
Kosten $r_1r_2r_3 + r_0r_1r_3$ bei der Klammerung $A_1(A_2A_3)$.

Beispiel: $(r_0, r_1, r_2, r_3) = (5, 10, 3, 10)$.
Die erste Klammerung führt zu Kosten $5 \cdot 10 \cdot 3 + 5 \cdot 3 \cdot 10 = 300$, die zweite zu Kosten $10 \cdot 3 \cdot 10 + 5 \cdot 10 \cdot 10 = 800$.

Wir wollen für beliebiges k und beliebige $(r_0, r_1, \ldots, r_k)$ die Klammerung ermitteln, die die geringsten Kosten verursacht.

Methode: Dynamische Programmierung.

Problem:

Eingabe: Dimensionen $r_0 \times r_1$, $r_1 \times r_2$, ..., $r_{k-1} \times r_k$ von Matrizen $A_1, \ldots, A_k$.

Kurz: $(r_0, r_1, \ldots, r_k)$ für ein $k \geq 2$.

Ausgabe: Eine („*optimale*“) Klammerung, die bei der Berechnung von $A_1 \cdot \cdots \cdot A_k$ die **Gesamtkosten** (= Anzahl aller R-Multiplikationen) **minimiert**. Und: diese Kosten.

Seien $\boldsymbol{c(r_0, \ldots, r_k)}$ diese Kosten bei optimaler Klammerung, wobei $k \geq 2$ und $r_0, r_1, \ldots, r_k \geq 1$ beliebige ganze Zahlen sind.

Für $k = 1$ setzen wir $\boldsymbol{c(r_0, r_1)} := 0$ für alle r_0, r_1. (Es ist nichts zu tun.)

Klar: $c(r_0, r_1, r_2) = r_0 r_1 r_2$.

Sei $(r_0, r_1, \ldots, r_k)$ für ein $k \geq 3$ gegeben. Die optimale Lösung des Klammerungsproblems hat die Form

$$\underbrace{(A_1 \cdots A_\ell)}_{\text{irgendwie geklammert}} \underbrace{(A_{\ell+1} \cdots A_k)}_{\text{irgendwie geklammert}}$$

für ein $1 \leq \ell < k$ (das wir nicht kennen!).

Zentrale Beobachtung („Optimale Substruktur“):

Die Klammerungen in den Teilen $A_1 \cdots A_\ell$ bzw. $A_{\ell+1} \cdots A_k$ müssen optimale Kosten für die Teilprobleme liefern.

Wäre z. B. die Teilklammerung von $A_1 \cdots A_\ell$ nicht optimal, dann könnte man durch **Verbesserung dieses Teils** die **Gesamtkosten senken**.

Aus dieser Beobachtung ergibt sich: Es gibt ein ℓ, $1 \leq \ell < k$, mit

$$c(r_0, \ldots, r_k) = c(r_0, \ldots, r_\ell) + c(r_\ell, \ldots, r_k) + \underbrace{r_0 r_\ell r_k}_{\substack{\text{äußere} \\ \text{Matrizenmult.}}} .$$

Der letzte Summand stellt die Kosten für die abschließende Multiplikation einer $r_0 \times r_\ell$- mit einer $r_\ell \times r_k$-Matrix dar.

Wir suchen also ein ℓ, das die Summe auf der rechten Seite minimiert.

Wir benötigen optimale Klammerungen bzw. ihre Kosten für die Teilprobleme $A_1 \cdots A_\ell$ und $A_{\ell+1} \cdots A_k$.

Ansatz: Betrachte die **Teilprobleme** $TP(i, j)$, die minimalen Kosten für die Berechnung des Teilprodukts $A_i \cdots A_j, 1 \leq i \leq j \leq k$.

Parametersatz zu $TP(i, j)$: $(r_{i-1}, \ldots, r_j)$.

Beobachtung („Optimale Substruktur“):

Optimale Klammerung für $A_i \cdots A_j$ ist $\underbrace{(A_i \cdots A_\ell)}_{\text{irgendwie geklammert}}\underbrace{(A_{\ell+1} \cdots A_j)}_{\text{irgendwie geklammert}}$, wobei die Klammerungen in den beiden Teilen optimal sein müssen.

„Bellmansche Optimalitätsgleichungen“:

$$c(r_{i-1}, \ldots, r_j) = \min\left\{ c(r_{i-1}, \ldots, r_\ell) + c(r_\ell, \ldots, r_j) + r_{i-1} r_\ell r_j \;\middle|\; i \leq \ell < j \right\},$$

für $1 \leq i < j \leq k$. (Gilt auch für $j = i + 1$, trivialerweise.)

Basisfall: $c(r_{i-1}, r_i) = 0$, für $1 \leq i \leq k$.

Um die optimalen Werte zu berechnen, müssen wir nur die Rekursionsformel iterativ auswerten. Erweiterung zur Ermittlung der optimalen Klammerung führt zu folgendem Programm.

MatrixOptimal$((r_0, \ldots, r_k))$
Eingabe: Dimensionsvektor $(r_0, \ldots, r_k)$
Ausgabe: Kosten $c(r_0, \ldots, r_k)$ bei der optimalen Klammerung
`l[1..k,1..k]`: Plan zur Ermittlung der optimalen Unterteilung;
Datenstruktur: Matrizen `C[1..k,1..k]`, `l[1..k,1..k]`
Ziel: `C[`i`,`j`]` enthält $c(r_{i-1}, \ldots, r_j)$
`l[`i`,`j`]` enthält Index zur Unterteilung der Multiplikation bei $A_i \cdots A_j$

(1) **for** `i` **from** 1 **to** k **do** `C[i,i]` $\leftarrow 0$;
(2) **for** `i` **from** 1 **to** $k-1$ **do** `C[i,i+1]` $\leftarrow r_{\texttt{i}-1} \cdot r_{\texttt{i}} \cdot r_{\texttt{i}+1}$;
(3) **for** `d` **from** 2 **to** $k-1$ **do**
(4) **for** `i` **from** 1 **to** $k -$ `d` **do**
(5) bestimme das ℓ, `i` $\leq \ell <$ `i+d`,
(6) das $C =$ `C[i,`ℓ`]` $+$ `C[`$\ell+1$`,i+d]` $+ r_{\texttt{i}-1} \cdot r_\ell \cdot r_{\texttt{i+d}}$ minimiert;
(7) `l[i,i+d]` $\leftarrow$ dieses ℓ;
(8) `C[i,i+d]` $\leftarrow$ das minimale C;
(9) **Ausgabe:** `C[1..k,1..k]` und `l[1..k,1..k]`.

Korrektheit: Man beweist durch Induktion über $d = 0, 1, \ldots, k-1$ mit Hilfe der Bellmanschen Optimalitätsgleichungen, dass $\texttt{C}[i, i+d]$ die Zahl $c(r_{i-1}, \ldots, r_{i+d})$ enthält.

Zur Übung: Man zeige, dass mit Hilfe der $\texttt{l}[i, j]$-Werte die optimale Klammerung in Linearzeit ermittelt werden kann.

Laufzeit: Die Minimumssuche in Zeilen (5)–(6) kostet Zeit $O(k)$; mit den geschachtelten Schleifen (3)–(8) und (4)–(8) ergibt sich eine Rechenzeit von $\Theta(k^3)$.

Beispiel: $(r_0, \ldots, r_4) = (3, 2, 2, 5, 6)$.

`c`:

i \ j	1	2	3	4
1	0	12	42	108
2		0	20	80
3			0	60
4				0

`l`:

i \ j	1	2	3	4
1	$*$	$*$	2	2
2		$*$	$*$	3
3			$*$	$*$
4				$*$

$12 = 3 \cdot 2 \cdot 2 \qquad 20 = 2 \cdot 2 \cdot 5 \qquad 60 = 2 \cdot 5 \cdot 6$
$42 = 12 + 3 \cdot 2 \cdot 5 \quad 80 = 20 + 2 \cdot 5 \cdot 6 \; (< 60 + 2 \cdot 2 \cdot 6) \; \ldots$

Optimale Klammerung: $(A_1 A_2)(A_3 A_4)$.
Kosten: 108 Multiplikationen.

ENDE

Viel Erfolg bei der Prüfungsvorbereitung und in der Prüfung!

Algorithmen und Datenstrukturen

12. Kapitel: Dynamische Programmierung

Martin Dietzfelbinger

Juli 2018

Kapitel 12: Dynamische Programmierung

Typische Eigenschaften des Algorithmenparadigmas

Dynamische Programmierung

sind:

- Identifizierung von (vielen) **Teilproblemen**.
- Identifizierung einfacher **Basisfälle**.
- Formulierung einer Version der Eigenschaft

  Substrukturen optimaler Strukturen müssen optimal sein
- Daraus: Rekursionsgleichungen für Werte optimaler Lösungen:

  Bellman'sche Optimalitätsgleichungen
- Iterative Berechnung der optimalen Werte (und Strukturen).

12.5 Das Rucksackproblem

Problemstellung:

Eingabe: Ganzzahlige **Volumina** $a_1, \ldots, a_n > 0$, **Nutzenwerte** $c_1, \ldots, c_n > 0$, ganzzahlige Volumenschranke b.

Aufgabe: Packe die Objekte in einen Rucksack von Volumen b, so dass der Gesamtnutzen maximiert wird. (Wird gleich noch formalisiert.)

Wir betrachten zwei Problemvarianten:

1. Rucksackproblem mit Wiederholungen. (Von jedem Objekt sind beliebig viele Kopien vorhanden.)
2. 0-1-Rucksackproblem. (Jedes Objekt ist genau einmal vorhanden.)

Im Gegensatz zum **fraktionalen Rucksackproblem** (Kap. 10.1) ist es hier nicht erlaubt, Objekte zu teilen.
Der Lösungsvektor $(x_1, \ldots, x_n)$ muss also **ganzzahlig** sein.

Das Rucksackproblem mit Wiederholungen

Eingabe: Ganzzahlige **Volumina** $a_1, \ldots, a_n > 0$, **Nutzenwerte** $c_1, \ldots, c_n > 0$, ganzzahlige Volumenschranke b.

Aufgabe: Wähle $x_1, \ldots, x_n \in \mathbb{N}$ mit

$$\sum_{1 \leq i \leq n} x_i a_i \leq b,$$

$$\sum_{1 \leq i \leq n} x_i c_i \text{ möglichst groß.}$$

Ansatz: Dynamische Programmierung.

Identifizierung von nützlichen **Teilproblemen**:

Für $d = 0, \ldots, b$ sei $\mathbf{w(d)}$:= maximaler erreichbarer Nutzenwert für Volumen $a_1, \ldots, a_n$, Nutzenwerte $c_1, \ldots, c_n$, und Rucksackgröße d.

Klar: $w(0) = 0$.

Bellman'sche Optimalitätsgleichungen:

$$w(d) = \begin{cases} 0, & d = 0 \\ \max\left(\{c_i + w(d - a_i) \mid 1 \leq i \leq n, a_i \leq d\} \cup \{0\}\right), & \text{sonst.} \end{cases}$$

Laufzeit: $O(n \cdot b)$. (Für jedes $d, 1 \leq d \leq b$, muss ein Maximum über $\leq n + 1$ Werte gebildet werden.)

Interessant: Wie berechnet man den Lösungsvektor $x_1, \ldots, x_n$?

Dafür legen wir ein Array $I[0..b]$ an.

$I[d]$ gibt ein $i \in \{1, \ldots, n\}$ an, welches $c_i + w(d - a_i)$ maximiert. (Wenn kein solches Objekt existiert, setzen wir $I[d] \leftarrow 0$.)

Wir nutzen den folgenden Algorithmus zur Berechnung des Lösungsvektors:

1. Initialisiere ein Array $X[1..n]$ mit 0'en
2. $d := b$
3. **while** $d > 0$ **do**
4. $\quad X[I[d]]{+}{+}$
5. $\quad d := d - a_{I[d]}$

Ausgabe: $X[1..n]$.

Beispiel:

$b = 10$ und:

i	1	2	3	4
a_i	6	3	4	2
c_i	30	14	16	9

d	0	1	2	3	4	5	6	7	8	9	10
$w(d)$	0	0	9	14	18	23	30	32	39	44	48
$I[d]$	0	0	4	2	4	2	1	2	1	1	4

Rekonstruierte Bepackung: $x_1 = 1, x_2 = 0, x_3 = 0, x_4 = 2$.

Resultat

Satz 12.5.1

Das Rucksackproblem mit Wiederholungen (mit Ausgabe einer optimalen Lösung) ist in Zeit $O(n \cdot b)$ und Platz $O(b)$ lösbar.

Das 0-1-Rucksackproblem

Eingabe: Ganzzahlige **Volumina** $a_1, \ldots, a_n > 0$, **Nutzenwerte** $c_1, \ldots, c_n > 0$, ganzzahlige Volumenschranke b.

Ausgabe:
$I \subseteq \{1, \ldots, n\}$ derart, dass $\sum_{i \in I} a_i \leq b$ und $\sum_{i \in I} c_i$ möglichst groß.

Ansatz: Dynamische Programmierung.

Identifizierung von nützlichen **Teilproblemen**:

Für $1 \leq k \leq n$ und $0 \leq v \leq b$ definieren wir

$$m(k, v) := \max\left\{ \sum_{i \in I} c_i \mid I \subseteq \{1, \ldots, k\} \wedge \sum_{i \in I} a_i \leq v \right\}.$$

Das Teilproblem **P(k, v)** besteht darin, ein $I \subseteq \{1, \ldots, k\}$ mit $\sum_{i \in I} a_i \leq v$ zu finden, das $\sum_{i \in I} c_i$ maximiert.

Wir suchen also nach einer Auswahl, die nur **Objekte Nummer $1, \ldots, k$** benutzt, eine modifizierte **Gewichtsschranke v** einhält, und den Nutzen maximiert.

Trivial: $m(0, v) = 0$ für alle v.

P(n, b) ist das Gesamtproblem.

Eigenschaft **„Optimale Substruktur“**:

Sei I optimale Lösung für $\mathrm{P}(k, v)$, d. h.:

$m(k, v) = \sum_{i \in I} c_i$, $I \subseteq \{1, \ldots, k\}$, $\sum_{i \in I} a_i \leq v$.

Dann tritt einer der folgenden Fälle ein.

1. Fall: $k \in I$. – Dann ist $I - \{k\}$ optimale Lösung für $\mathrm{P}(k-1, v-a_k)$, d. h.:

$$m(k, v) - c_k = \sum_{i \in I - \{k\}} c_i = m(k-1, v-a_k).$$

2. Fall: $k \notin I$. – Dann ist I optimale Lösung für $\mathrm{P}(k-1, v)$, d. h.:

$$m(k, v) = m(k-1, v).$$

Beweis: **Indirekt.**

1. Fall: $k \in I$. – **Annahme:** $I - \{k\}$ nicht optimal für $\mathrm{P}(k-1, v-a_k)$.
Dann gäbe es eine bessere Teilmenge $J \subseteq \{1, \ldots, k-1\}$ mit
$\sum_{i \in J} a_i \leq v - a_k$ und $\sum_{i \in J} c_i > \sum_{i \in I - \{k\}} c_i$.
Dann wäre aber $J \cup \{k\}$ eine bessere Lösung für $\mathrm{P}(k, v)$ als I,
Widerspruch.

2. Fall: $k \notin I$. – **Annahme:** I nicht optimal für $\mathrm{P}(k-1, v)$.
Dann gäbe es eine bessere Teilmenge $J \subseteq \{1, \ldots, k-1\}$.

Dann wäre aber J auch besser als I für $\mathrm{P}(k, v)$, **Widerspruch**.

Die optimale Lösung für $P(k, v)$ **enthält** also eine optimale Lösung für $P(k-1, v')$ für ein $v' \leq v$.

Eigenschaft **„Optimale Substruktur“**!

Bellman'sche Optimalitätsgleichungen:

$$m(k, v) = \begin{cases} m(k-1, v-a_k) + c_k, & \text{falls } v \geq a_k \text{ und diese Summe} \\ & \textbf{größer} \text{ als } m(k-1, v) \text{ ist;}^* \\ m(k-1, v), & \text{sonst.} \end{cases}$$

für $1 \leq k \leq n$, $0 \leq v \leq b$;

$$m(0, v) = 0, \quad \text{für } 0 \leq v \leq b.$$

* In diesem Fall ist k Element jeder optimalen Lösung von $P(k, v)$.

Wir berechnen alle $m(k, v)$ mittels einer iterativen Prozedur.

12.5.2 Algorithmus DP-Zero-One-Knapsack$(a_1, \ldots, a_n, c_1, \ldots, c_n, b)$
Eingabe: $a_1, \ldots, a_n$: Volumina; $c_1, \ldots, c_n$: Nutzenwerte; b: Schranke
Ausgabe: $I \subseteq \{1, \ldots, n\}$: Optimale legale Auswahl.
Datenstrukturen: `m[0..`n`,0..`b`] : integer;`

```
(1)    for v from 0 to b do m[0,v] ← 0;
(2)    for k from 1 to n do
(3)      for v from 0 to b do
(4)        if v − a_k ≥ 0
(5)          then s ← m[k−1,v−a_k] + c_k else s ← 0;
(6)        if s > m[k−1,v]
(7)          then m[k,v] ← s;
(8)          else m[k,v] ← m[k−1,v];
       // Konstruktion der optimalen Menge:
(9)    I ← ∅; r ← b;
(10)   for k from n downto 1 do
(11)     if m[k−1,r−a_k] + c_k > m[k−1,r]
(12)       then r ← r − a_k; I ← I ∪ {k};
(13)   return I.
```

Korrektheit des ermittelten Wertes `m[`n, b`]` ergibt sich aus den Bellman'schen Optimalitätsgleichungen durch Induktion über k.

Korrektheit der ausgegebenen Menge `I` folgt aus der obigen Überlegung, dass die Bedingung

$$m(k-1, v-a_k) + c_k > m(k-1, v)$$

entscheidend ist für die Frage, ob k in der optimalen Lösung von $\mathrm{P}(k, v)$ enthalten ist oder nicht, und mit Induktion.

Laufzeit des Algorithmus: $O(n \cdot b)$.

Beispiel:

$b = 6$ und

i	1	2	3	4
a_i	3	4	2	1
c_i	4	6	2	5

$m[k, v]$:

	v: 0	1	2	3	4	5	6
$k = 0$	0	0	0	0	0	0	0
1	0	**0**	0	4	4	4	4
2	0	0	0	4	6	**6**	6
3	0	0	2	4	6	**6**	8
4	0	5	5	7	9	11	**11**

Ablesen von I aus dem Array `m[0..n,0..b]`:

Benutze $m(k-1, v-a_k) + c_k > m[k-1, v]$ als Kriterium.

(Eingerahmte Einträge der Tabelle.)

$m[4,6] = 11 > 8 = m[3,6]$,
  also `I := {4}`, `v := 6 - 1 = 5`
$m[3,5] = 6 \not> 6 = m[2,5]$,
$m[2,5] = 6 > 4 = m[1,5]$,
  also `I := I ∪ {2}`, `v := 5 - 4 = 1`
$m[1,1] = 0 \not> 0 = m[0,1]$.

Resultat: Optimale Menge ist $I = \{2,4\}$.

Man kann auf die Speicherung des gesamten Arrays `m` verzichten.

Man hält immer nur die letzte Zeile, und aktualisiert die Einträge **von rechts nach links**.
(Rechts vom Arbeitspunkt v: Einträge $m(k, v')$, links davon: Einträge $m(k-1, v')$.)

Eine Bitmatrix `b[1..n,0..b]` führt mit, ob Objekt k für eine optimale Bepackung notwendig ist oder nicht.

12.5.3 Algorithmus DP-Zero-One-Knapsack($a_1, \ldots, a_n$, $c_1, \ldots, c_n$, b)
Eingabe: $a_1, \ldots, a_n$: Volumina; $c_1, \ldots, c_n$: Nutzenwerte; b: Schranke
Ausgabe: $I \subseteq \{1, \ldots, n\}$: Optimale legale Auswahl.
Datenstrukturen: `m[0..`b`]: integer`; `b[1..`n`,0..`b`]: Boolean`;

```
(1)    for v from 0 to b do m[v] ← 0;
(2)    for k from 1 to n do
(3)      for v from b downto 0 do
(4)        if v − a_k ≥ 0 then s ← m[v−a_k] + c_k else s ← 0;
(5)        if s > m[v]
(6)          then m[v] ← s; b[k,v] ← 1;
(7)          else b[k,v] ← 0;
       // Konstruktion der optimalen Menge:
(8)    I ← ∅; r ← b;
(9)    for k from n downto 1 do
(10)     if b[k,r] = 1 then
(11)       r ← r−a_k; I ← I ∪ {k};
(12)   return I.
```

Resultat

Satz 12.5.4

Das 0-1-Rucksackproblem (mit Ausgabe der optimalen Lösung) ist in Zeit $O(nb)$ lösbar.

Wie gut sind diese beiden Algorithmen?

Die **Größe**, **nicht die Bitlänge** der Gewichtsschranke b geht in die Laufzeit ein.

Wir nennen Algorithmen mit einer Laufzeit $O(p(n, B))$, wo n die (strukturelle) Größe der Eingabe und B eine obere Schranke für einige oder alle Zahlenkomponenten der Eingabe, und p ein Polynom in zwei Variablen ist, *pseudopolynomiell*.

Die Existenz eines pseudopolynomiellen Algorithmus für das Rucksackproblem widerspricht nicht der Tatsache, dass das 0-1-Rucksackproblem wahrscheinlich keinen Polynomialzeitalgorithmus hat (nächstes Semester: NP-Vollständigkeit!):
Die *natürliche* Eingabegröße berücksichtigt $\log b$ und nicht b.

12.6. Traveling Salesperson

Eingabe: Vollständiger, gerichteter Graph mit Kantengewichten $a_{i,j} \geq 0$. (Also eine Matrix $((a_{i,j}))_{1 \leq i,j \leq n}$.)

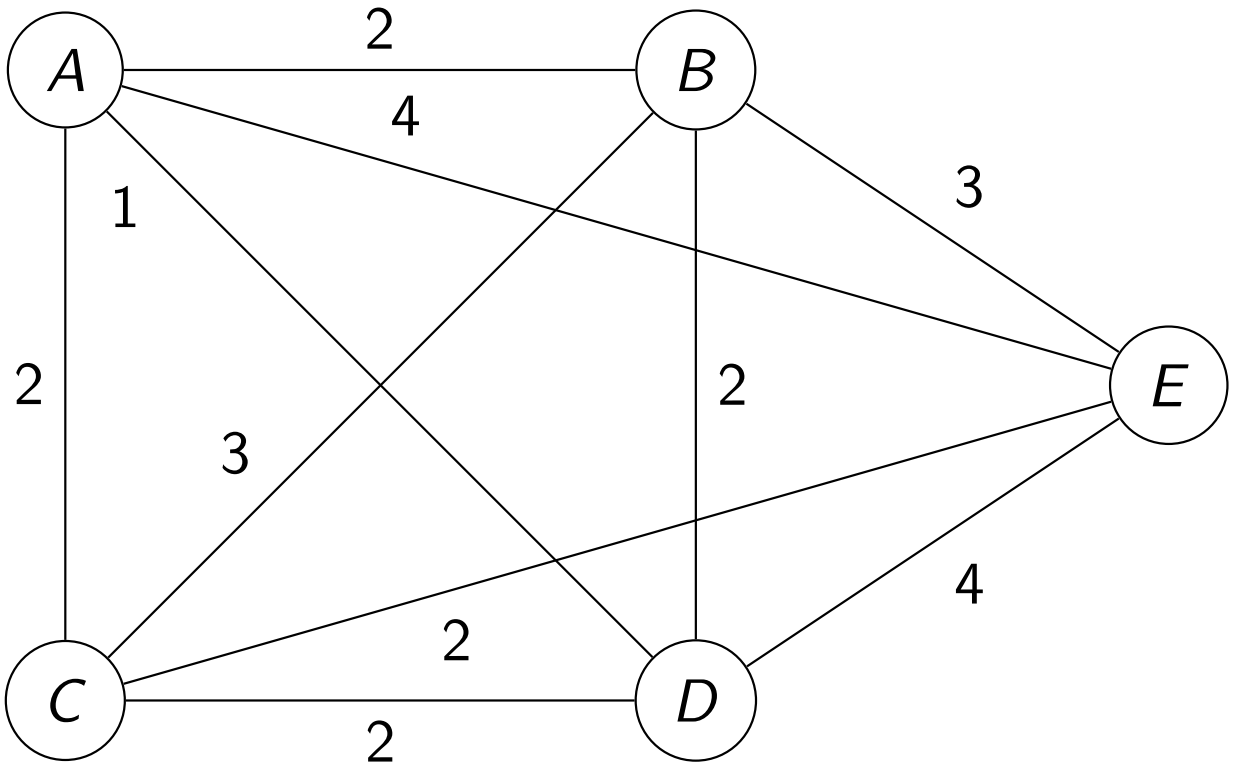
Gesucht: Eine günstigste „Rundreise“, bei der jede Stadt (also jeder Knoten) genau einmal besucht wird und zum Startpunkt zurückgekehrt wird.

Formal: Gesucht ist eine Permutation π von $\{1, \ldots, n\}$, so dass

$$c(\pi) = \sum_{2 \leq i \leq n} a_{\pi(i-1),\pi(i)} + a_{\pi(n),\pi(1)}$$

minimal unter allen möglichen Permutationen.

Ein Beispiel



Kürzeste Tour: (A, B, E, C, D, A) mit Länge 10.

Klar: Wir können den Anfangspunkt fest wählen. (Hier: Tour startet im Knoten 1.)

Naiver Ansatz: Probiere alle $(n-1)!$ mögliche Rundtouren aus.

Wir entwickeln einen sehr viel besseren Algorithmus, basierend auf „Dynamischer Programmierung“.

Zentrale Frage: Wie sehen sinnvolle Teilprobleme aus?

Wir betrachten $P(S, \ell)$ mit $S \subseteq \{1, \ldots, n\}, 1 \leq \ell \leq n$, wobei $1, \ell \in S$.

$P(S, \ell)$ fragt nach dem kürzesten, **einfachen** Weg von 1 durch alle Knoten in S mit Ende ℓ.

Basisfälle: $P(\{1\}, 1) = 0, P(S, 1) = \infty$ für $|S| > 1$.

Am Ende: Länge einer kürzesten Rundtour:

$$\min(P(\{1, \ldots, n\}, \ell) + a_{\ell,1} \mid 1 \leq \ell \leq n).$$

Für $|S| > 1$ gilt:

$$P(S, j) = \min(P(S - \{j\}, i) + a_{i,j} \mid i \in S - \{j\}).$$

Iterativ programmiert sieht der Algorithmus wie folgt aus:

1. $P(\{1\}, 1) \leftarrow 0$
2. **for** $s \leftarrow 2$ **to** n **do**
3. **foreach** $S \subseteq \{1, 2, \ldots, n\}$ with $|S| = s$ and $1 \in S$ **do**
4. $P(S, 1) \leftarrow \infty$
5. **foreach** $j \in S, j \neq 1$ **do**
6. $P(S, j) \leftarrow \min(P(S - \{j\}, i) + a_{i,j} \mid i \in S - \{j\})$.
7. **return** $\min(P(\{1, \ldots, n\}, \ell) + a_{\ell,1} \mid 1 \leq \ell \leq n)$

Resultat

Satz 12.6.1

Das Problem Traveling Salesperson kann mittels dynamischer Programmierung in Zeit $O(2^n \cdot n^2)$ gelöst werden.

(Dies ist eine wesentliche Verbesserung gegenüber dem naiven Ansatz mit Laufzeit $O(n!)$.)

Beweis

- Die Anzahl aller Teilmengen von $\{1, \ldots, n\}$ ist 2^n.
- Zu jeder solcher Teilmenge gibt es maximal n Teilprobleme.
- Jedes Teilproblem kann in Zeit $O(n)$ gelöst werden.

Für Zuhause:
Wie konstruiert man eine optimale Rundtour (also die Knotenfolge)?