Karsten Weicker

HTWK Leipzig, Fachbereich IMN

Skript zur Veranstaltung Algorithmen und Datenstrukturen

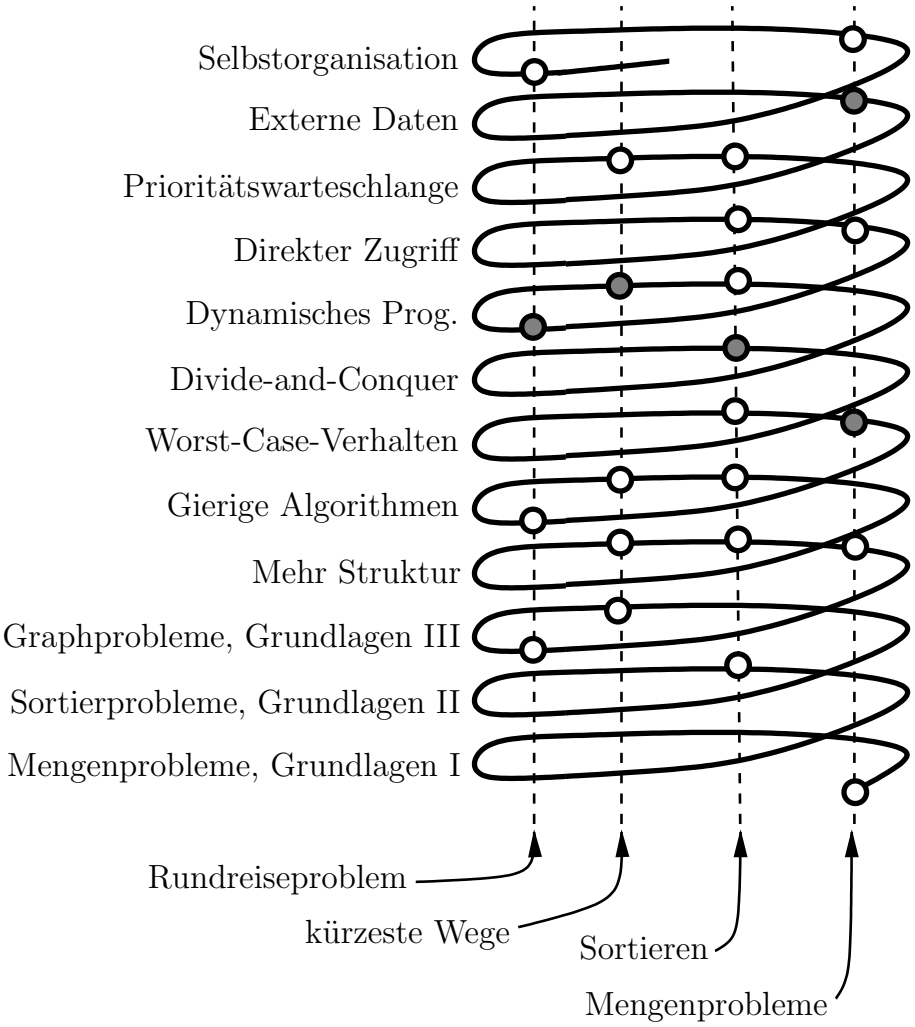
Version 11 (2018)

Inhaltsverzeichnis

Kapitel 0

Einführendes

Aufbau der Vorlesung



Literatur

Begleitendes Lehrbuch:

- K. Weicker, N. Weicker: „Algorithmen und Datenstrukturen“, SpringerVieweg, 2013.

Ebenfalls hilfreiche zusätzliche Literatur:

- T. H. Cormen, C. E. Leiserson, R. Rivest, C. Stein: „Algorithmen - Eine Einführung“, Oldenbourg, 2004.
- T. Ottmann, P. Widmayer: „Algorithmen und Datenstrukturen“, 4. Auflage, Spektrum, 2002.
- R. Sedgewick: „Algorithmen in Java“, 3. Auflage, Addison-Wesley, 2003.
- M. A. Weiss: „Data Structures and Algorithm Analysis in Java“, 2. Auflage, Pearson, 2007.

Als Wegweiser durch die Literatur soll die folgende Tabelle dienen, die die unterschiedlichen Themen der Vorlesung in den Büchern bewertet.

++	alle wesentliche Information ist vorhanden und sehr gut erklärt
+	alle wesentliche Information ist vorhanden
0	einige Details/Aspekte fehlen
−	das Thema ist nur in Ansätzen diskutiert
−−	das Thema fehlt vollständig

	Cormen et. al.	Ottman/Widmayer	Sedgewick	Weiss
Asymptotische Komplexität	++	0	++	+
Grundsätzliche Konzepte	−	−	+	0
Bubblesort	−	+	++	−−
Backtracking	−−	−−	−−	++
Binäre Suche	−	++	++	+
Verkettete Liste	++	+	+	+
Insertionsort	++	++	++	+
Shellsort	−−	+	++	++
Skip-Liste	−−	++	++	++
Binäre Suchbäume	++	++	0	+
Graphdarstellung	++	++	−	+
Breiten-/Tiefensuche	++	0	0	0

	Cormen et. al.	Ottman/Widmayer	Sedgewick	Weiss
Selectionsort	$-$	$++$	$++$	$--$
Dijkstra-Algorithmus	$++$	$+$	$--$	$++$
Prim-Algorithmus	$++$	$-$	$--$	$++$
TSP-Approximation	$++$	$--$	$--$	$--$
AVL-Bäume	$-$	$+$	$-$	$++$
Bestmögliche Sortierzeit	$+$	$+$	$--$	$++$
Mergesort	$++$	$++$	$+$	$++$
Laufzeitanalyse D&C	$++$	$--$	$--$	0
Quicksort	$+$	$+$	$+$	$+$
Prioritätswarteschlange	$+$	0	$+$	$++$
Heap	$++$	$+$	0	$++$
Heapsort	$+$	$+$	0	0
Countingsort	$++$	$--$	$--$	$--$
Radixsort	$++$	0	0	$--$
Interpolationssuche	$--$	$+$	$+$	$--$
Hashing	0	$+$	0	0
Floyd-Warshall	$++$	$--$	$--$	$+$
Bitonische Rundreise	$-$	$--$	$--$	$--$
B-Bäume	$++$	0	0	0
Fortgeschrittenes	$+$	$+$	$--$	$+$

Kapitel 1

Mengenproblem und Algorithmusbegriff

Sergeant Horvath: It's not gonna be easy
finding one particular soldier
in the middle of this whole goddamn war.
Captain Miller: Like finding a needle in a stack of needles.
(Saving Private Ryan, 1998)

1.1 Das Mengenproblem

$\longrightarrow$ **[Buch, Abschnitt 1.2, S.4-5]**

Operationen in einem Anwendungsszenario: Verwaltung von Filialdaten

- Neue Filialen müssen in den Datenstamm aufgenommen werden.
- Die Daten jeder Filiale müssen schnell zugreifbar sein, um sie auszulesen, zu modifizieren oder anderweitig zu verarbeiten.
- Existierende Filialen können geschlossen werden, sind dann also zu löschen.
- Für einen Überblick sollen alle Filialen aufgelistet werden.

Eindeutiger Schlüssel: z.B. hier die Postleitzahl der Filiale

Definition: Mengenproblem

- Gegeben: geordnete Basismenge möglicher Schlüssel $\mathscr{A}$,
  gespeicherte Elemente $A = \{a_1, \ldots, a_n\} \subset \mathscr{A}$.
- EINFÜGEN: Füge $b \in \mathscr{A}$ in A ein
- SUCHEN: Entscheide für ein $b \in \mathscr{A}$ ob $b \in A$ (und gib ggf. an wo)

- LÖSCHEN: Entferne $b \in A$ aus A
- ALLE-ELEMENTE: Alle Elemente in A in einer nicht genauer spezifizierten Reihenfolge ausgegeben

1.2 Unsortierte Ablage in einem Feld

⟶ **[Buch, Abschnitt 3.1.1, S.43-45]**

Annahme: Wir gehen bei der hier vorgeschlagenen Lösung davon aus, dass Schlüssel doppelt vorkommen können. Andernfalls müsste bei jedem Einfügen zusätzlich geprüft werden, ob der Schlüssel bereits enthalten ist, was die Algorithmen unnötig kompliziert macht.

```
SUCHEN-FELD(Schlüssel gesucht)
    Rückgabewert: gesuchte Daten bzw. Fehler falls nicht enthalten
1   index ← 1
2   while index ≤ belegteFelder und gesucht ≠ A[index].wert
3   do ⌊ index ← index + 1
4   if index > belegteFelder
5   then ⌊ error "Element nicht gefunden"
6   else ⌊ return A[index].daten
```

```
EINFÜGEN-FELD(Schlüssel neuerWert, Daten neueDaten)
    Rückgabewert: nichts falls erfolgreich bzw. Fehler sonst
1   if belegteFelder = A.länge
2   then ⌊ error "Feld ist voll"
3   else ⌈ A[belegteFelder + 1].wert ← neuerWert
4        ⌈ A[belegteFelder + 1].daten ← neueDaten
5        ⌊ belegteFelder ← belegteFelder + 1
```

```
LÖSCHEN-FELD(Schlüssel löschWert)
    Rückgabewert: nichts falls erfolgreich bzw. Fehler sonst
1   index ← 1
2   while index ≤ belegteFelder und löschWert ≠ A[index].wert
3   do ⌊ index ← index + 1
4   if index > belegteFelder
5   then ⌊ error " Element nicht gefunden "
6   else ⌈ A[index] ← A[belegteFelder]
7        ⌊ belegteFelder ← belegteFelder − 1
```

```
ALLE-FELD()
    Rückgabewert: nichts, da direkte Ausgabe
1   index ← 1
2   while index ≤ belegteFelder
3   do ⌜ drucke: A[index].daten
4      ⌞ index ← index + 1
```

Suchen:

Einfügen:

Löschen:

Alle:

1.3 Begriffsklärung: Algorithmus

⟶ **[Buch, Abschnitt 2.1, S.17-21]**

Def.: Algorithmus Ein Algorithmus ist eine Vorschrift zur Bewältigung einer Aufgabe als Folge von Aktionen, wobei die folgenden Bedingungen gelten:

- Der Algorithmus lässt sich in endlicher Form beschreiben und ist aus Elementen von endlich vielen, durch die ausführende Maschine definierten Kommandobefehlen aufgebaut.
- Der Algorithmus hat genau ein Startaktion und nach der Ausführung eines jeden Befehls ist die Menge der Folgeaktionen klar durch die Beschreibung und die bisher durchgeführten Aktionen definiert.
- Die Eingabe des Algorithmus ist eine Folge von Daten, die auch leer oder unendlich sein kann. Aber zu jedem Zeitpunkt während der Ausführung ist die bisher betrachtete Datenmenge endlich.

Notation: Hochsprachliches Maschinenmodell mit einer abstrakteren Pseudo-Code-Notation

Def.: Determiniert und deterministisch

- *determiniert* heißt: dieselbe Eingabe führt immer zum selben Ergebnis
- *deterministisch* heißt: innerer Ablauf ist für eine Eingabe immer identisch – sonst:
  - *randomisiert*: nutzt Pseudozufallszahlen
  - *nicht-deterministisch*: nutzt Operation „rate richtig“ (versteckte Parallelität)
  - *asynchron*: bei parallelen Algorithmen

Def.: terminiert und korrekt

- *teminiert* auf Eingabe x :$\Leftrightarrow$ hält nach endlich vielen Schritten
- *terminiert stets* :$\Leftrightarrow$ terminiert für alle möglichen Eingaben
- *partiell korrekt* :$\Leftrightarrow$ Ausgabe genügt der Spezifikation, falls der Algorithmus terminiert
- *total korrekt* :$\Leftrightarrow$ partiell korrekt und terminiert stets

Arten von Algorithmen

- Klassische Algorithmen: immer korrekt, determiniert und terminieren stets
- Randomisierte Algorithmen: dürfen zusätzliche Zufallszahlen benutzen, sind aber trotzdem korrekt und terminieren stets
  Beispiel: Quicksortvariante, Skipliste
- Approximationsalgorithmen: liefern eine garantiert gute Näherung der Lösung liefern – meist gibt es einen Beweis für den maximalen Fehler
  Beispiel: TSP-Approximation
- Probabilistische Algorithmen: keine Garantie für die Lösungsqualität, sondern „irren“ sich mit einer gewissen Wahrscheinlichkeit
  Beispiel: evolutionärer Algorithmus
- Heuristik: liefern genau dann gute Ergebnisse, wenn die Eingaben zu einer Grundannahme der Heuristik passen
  Beispiel: minimale bitonische Rundreise

	Korrektheit	Terminierung	Vorhersag-barkeit	Zufalls-zahlen
klassisch	ja	ja	ja	nein
heuristisch	nicht sicher	nicht sicher	nein	manchmal
approximativ	nah dran	ja	ja	manchmal
randomisiert	ja	ja	meist nein	ja
probabilistisch	meist	ja	nicht sicher	manchmal

Def.: Lösbarkeit eines Problems Ein Problem ist lösbar, wenn es einen Algorithmus gibt, der für jede Instanz des Problems in endlicher Zeit eine Lösung berechnet.

Def.: Datenstruktur Eine *Datenstruktur* wird definiert durch

- eine Vorschrift, wie Daten im Speicher abgelegt bzw. verknüpft werden und
- die Algorithmen, um die Daten zu verwalten bzw. auf selbige zuzugreifen.

1.4 Laufzeit: exakte Schrittzahl

⟶ **[Buch, Abschnitt 2.3, S.23-26]**

Basisregeln zur Ermittelung der Schrittanzahl:

- Sequenzen von Anweisungen:

- Verzweigungen:

- Schleifen:

erstes Beispiel:

```
EIN-LAUFZEITBEISPIEL(n)
     Rückgabewert: –
1    s ← 0
2    for k ← 1,...,n
3    do ⌊ s ← s + k
```

$T(n) =$

zweites Beispiel:

```
WEITERES-LAUFZEITBEISPIEL(n)
     Rückgabewert: –
1    s ← 0
2    for k ← 1,...,n
3    do ⌈ for m ← k,...,n
4       ⌊ do ⌊ s ← s + m
```

$T(n) \leq$

$T(n) =$

drittes Beispiel:

```
LETZTES-LAUFZEITBEISPIEL(n)
     Rückgabewert: –
1    while n > 0
2    do ⌜ s ← s · n
3          if n ist gerade
4          then ⌞ n ← n − 3
5       ⌞ else ⌞ n ← n + 1
```

$T(n) \leq$

1.5 Unsortierte Ablage in einem dynamischen Feld

⟶ **[Buch, Abschnitt 3.1.3, S.52-54]**

```
EINFÜGEN-DYNFELD(Schlüssel neuerWert, Daten neueDaten)
     Rückgabewert: nichts falls erfolgreich bzw. Fehler sonst
1    if belegteFelder = A.länge
2    then ⌜ B → allokiere Feld der Länge ⌈3/2 · A.länge⌉
3           for index ← 1,...,belegteFelder
4           do ⌜ B[index].wert ← A[index].wert
5              ⌞ B[index].daten ← A[index].daten
6         ⌞ A → B
7    EINFÜGEN-FELD(neuerWert, neueDaten)
```

Theorem: Werden n Elemente in ein dynamisches Feld eingefügt, so beträgt der Gesamtaufwand für das Umkopieren höchstens Schreibzugriffe.

```
LÖSCHEN-DYNFELD(Schlüssel löschWert)
    Rückgabewert: nichts falls erfolgreich bzw. Fehler sonst
1   LÖSCHEN-FELD(löschWert)
2   if belegteFelder < 1/2 · A.länge und A.länge > 5
3   then ⌜ B → allokiere Feld der Länge ⌊2/3 · A.länge⌋
4            for index ← 1,...,belegteFelder
5            do ⌜ B[index].wert ← A[index].wert
6               ⌞ B[index].daten ← A[index].daten
7          ⌞ A → B
```

Suchen:

Einfügen:

Löschen:

Alle:

Gemessene Laufzeiten Komplettes Durchsuchen der Datenstruktur mit 21500000 Elementen.

Datenstruktur	Feld	Liste (linear allokiert)	Liste (anders verzeigert)
Laufzeit	164.8 ms	243.2 ms	2.151 sec

Kapitel 2

Sortierproblem und asymptotische Laufzeit

Clarice Starling: But there is no pattern or
the computers would've nailed it.
They're even found in random order.
(The Silence of the Lambs, 1991)

2.1 Das Sortierproblem

$\longrightarrow$ **[Buch, Abschnitt 1.3, S.5-7]**

Anwendungsszenario: Die Filialdaten müssen gemäß unterschiedlicher Kriterien in einer sortierten Reihenfolge angezeigt werden, z.B. Umsatz oder Zeitpunkt der Eröffnung, aber auch abgeleitete Werte wie die Summe aus Personal- und Mietkosten. Hier werden die Werte des Sortierkriteriums als Schlüssel bezeichnet. Sie dienen nicht der eindeutigen Identifikation einer Filiale sondern der Einordnung in eine Rangliste.

Definition: Sortierproblem

- Geg: Folge von n Objekten mit den Schlüsseln $a_1, \ldots, a_n$
- Ges.: aufsteigende Anordnung der Objekte gemäß Vergleichsrelation $<$
- Operation SORTIEREN realisiert bijektive Abbildung $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ mit

$$a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}.$$

Beispiel: Folge mit den Schlüsseln $\langle 7, 3, 1, 4 \rangle$ (bzgl. der Relation $<$ auf den natürlichen Zahlen) angewandt, dann müssen die Elemente gemäß π wie folgt umsortiert werden:

$$\pi(1) = 3 \qquad \pi(2) = 2 \qquad \pi(3) = 4 \qquad \pi(4) = 1$$

Def.: Stabiles Sortieren
SORTIEREN heißt stabil genau dann, wenn die Reihenfolge von Elementen mit gleichem Schlüssel unverändert bleibt, d.h.

$$\forall 1 \leq i < n : a_{\pi(i)} = a_{\pi(i+1)} \Rightarrow \pi(i) < \pi(i+1).$$

Beispiel:
Ein Algorithmus, der die Schlüsselfolge $\langle 1,2,2,1 \rangle$ durch

$$\pi(1) = 4 \qquad \pi(2) = 1 \qquad \pi(3) = 3 \qquad \pi(4) = 2$$

sortiert ist nicht stabil, da die Reihenfolge der gleichen Schlüssel vertauscht wird. Stabil ist hingegen ein Algorithmus der wie folgt sortiert:

$$\pi(1) = 1 \qquad \pi(2) = 4 \qquad \pi(3) = 2 \qquad \pi(4) = 3.$$

Def: Ordnungsverträglich
SORTIEREN heißt ordnungsverträglich genau dann, wenn die Operation im paarweisen Vergleich von möglichen Eingaben für diejenigen $A = \langle a_1, \ldots, a_n \rangle$ schneller beendet ist, die eine kleinere Unordnung gemessen durch die Inversionszahl

$$\mathit{Inversionszahl}(A) = \#\{(i,j) \text{ mit } 1 \leq i < j \leq n \mid a_i > a_j\}$$

aufweisen.

2.2 Ein einfaches Sortierverfahren: Bubblesort

⟶ **[Buch, Abschnitt 3.2, S.56-59]**

Eines der einfachsten Sortierverfahren, das sowohl auf Feldern als auch auf Listen mit gleichem Aufwand arbeitet, ist Bubblesort (Sortieren durch Vertauschen). Im folgenden ist es für ein Feld formuliert.

```
BUBBLESORT(Feld A)
    Rückgabewert: nichts; Seiteneffekt: A ist sortiert
1   repeat ⌜ warSortiert ← true
2          for i ← 1,...,A.länge − 1
3          do ⌜ if A[i].wert > A[i+1].wert
4               then ⌜ warSortiert ← false
5          ⌞    ⌞        ⌞ VERTAUSCHE(A, i, i+1)
6   until warSortiert
```

```
VERTAUSCHE(A[], i, j)
    Rückgabewert: nichts; Seiteneffekt: A ist verändert
1   h ← A[i]
2   A[i] ← A[j]
3   A[j] ← h
```

Beispiel:

A[1]	A[2]								A[10]
20	54	28	31	5	24	39	14	1	15

Zeitaufwand worst-case:

Zeitaufwand best-case:

Def.: Schleifeninvariante Eine *Schleifeninvariante* ist eine logische Aussage, die vor dem Betreten der Schleife und nach jedem Durchlauf des Schleifenkörpers wahr ist.

Theorem: Bubblesort sortiert die Elemente in A aufsteigend.

Anzahl der Schreibzugriffe:

Ordnungsverträglich?

Stabil?

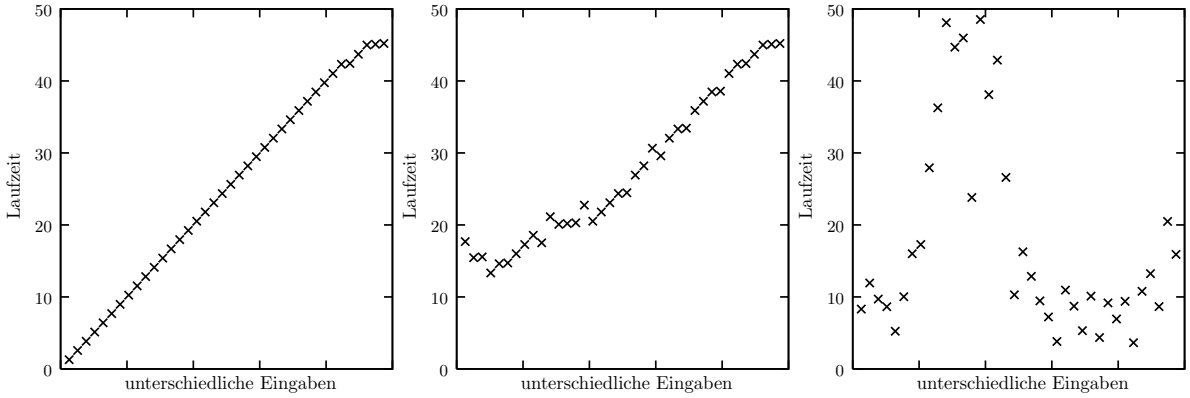
2.3 Asymptotische Laufzeit

⟶ **[Buch, Abschnitt 2.3, S.26-32]**

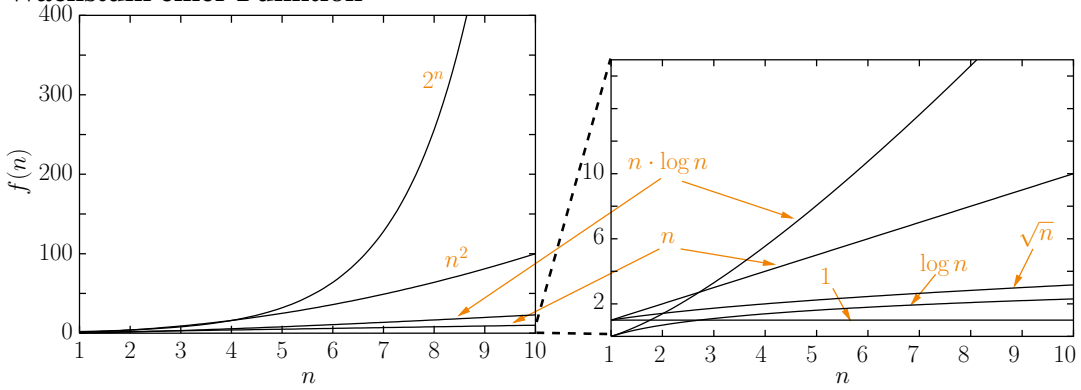
Zwei Dimensionen bzgl. der Laufzeit

- Problemgröße
- Menge der Probleminstanzen einer Größe

Def.: Best-, Worst- und Average-Case Betrachtet man für ein Problem immer diejenige Probleminstanz einer Größe, die die geringste Laufzeit hat, spricht man vom Best-Case (oder günstigsten Fall). Betrachtet man die längste Laufzeit, handelt es sich um den Worst-Case (oder ungünstigsten Fall). Untersucht man alle Probleminstanzen derselben Größe und ermittelt die durchschnittliche Laufzeit, bezeichnet man dies als Average-Case.



Wachstum einer Funktion



Anzahl der Stellen (im Dezimalsystem)

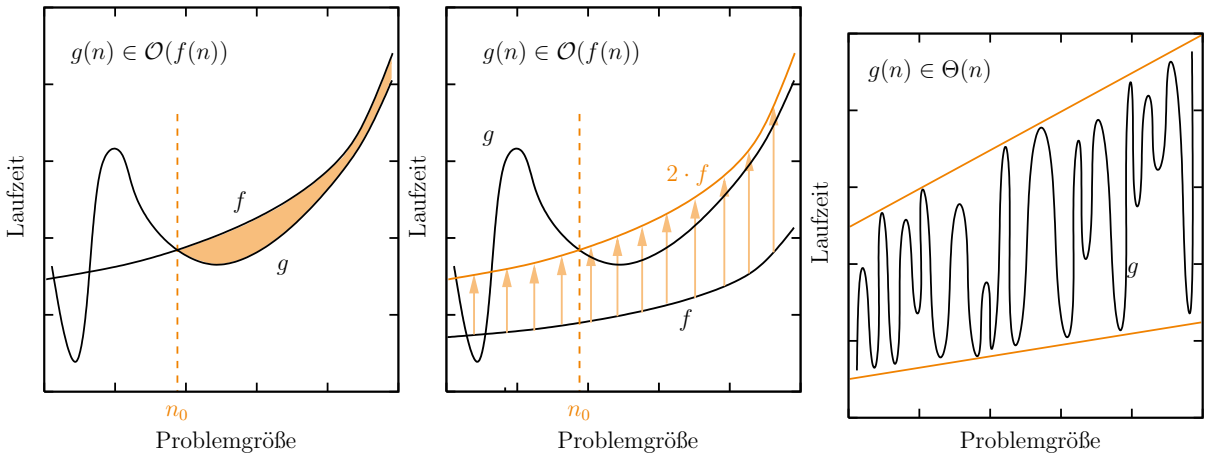
	Wert n										
	1	2	3	4	5	6	7	8	9	10...	1023
$\lg n$	1	1	1	1	1	1	1	1	1	1	1
n	1	1	1	1	1	1	1	1	1	2	4
$n \lg n$	1	1	1	1	2	2	2	2	2	2	5
n^2	1	1	1	2	2	2	2	2	2	3	7
2^n	1	1	1	2	2	2	3	3	3	4	308
$n!$	1	1	1	2	3	3	4	5	6	7	2636
2^{n^2}	1	2	3	5	8	11	15	20	25	31	?
2^{2^n}	1	2	3	5	10	20	39	78	155	309	?
$2^{\cdot^{\cdot^{\cdot^{2}}}} \Big\} n$	1	1	2	5	19728	?	?	?	?	?	?

Landau-Notation: Sei $f : \mathbb{N} \to \mathbb{N}$

$$\mathcal{O}(f) = \{g : \mathbb{N} \to \mathbb{N} \mid \exists_{c \in \mathbb{R}^+} \exists_{n_0 \in \mathbb{N}} \forall_{n \geq n_0} g(n) \leq c \cdot f(n)\}$$

$$\Omega(f) =$$

$$\Theta(f) =$$



Komplementäre Symbole: $\mathcal{O}$ und Ω sind komplementär zueinander:

$$f(n) \in \mathcal{O}(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$$

Reflexivität: Es gilt:

$$f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$$

Transitivität: Es gilt:

$$f(n) \in \mathcal{O}(g(n)) \text{ und } g(n) \in \mathcal{O}(h(n)) \Rightarrow f(n) \in \mathcal{O}(h(n))$$
$$f(n) \in \Omega(g(n)) \text{ und } g(n) \in \Omega(h(n)) \Rightarrow f(n) \in \Omega(h(n))$$
$$f(n) \in \Theta(g(n)) \text{ und } g(n) \in \Theta(h(n)) \Rightarrow f(n) \in \Theta(h(n))$$

Vereinfachung von Ausdrücken:

$$f(n) \in \mathcal{O}(g(n)) \Rightarrow \mathcal{O}((f+g)(n)) = \mathcal{O}(g(n))$$
$$f(n) \in \Omega(g(n)) \Rightarrow \Omega((f+g)(n)) = \Omega(f(n))$$
$$g(n) \in \Theta(f(n)) \Rightarrow \Theta(f+g) = \Theta(g) = \Theta(f)$$

Berechnungsregel:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+ \Rightarrow f(n) \in \Theta(g(n))$$
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) \in \mathcal{O}(g(n))$$

Hierarchie der Polynome:

$$m \in \mathbb{N} \Rightarrow \mathcal{O}(n^m) \subset \mathcal{O}(n^{m+1})$$

Mögliche Ausagen der Landau-Notation

	$\mathcal{O}(\cdot)$	$\Theta(\cdot)$	$\Omega(\cdot)$
exakte Zeit	obere Schranke	asymptotisch gleicher Best- und Worst-Case	untere Schranke
Worst-Case	großzügig Abschätzung – kann auch besser sein	asymptotisch exakt	mind. so schlecht
Average-Case	obere Schranke	asymptotisch exakt	untere Schranke
Best-Case	mind. so gut	asymptotisch exakt	großzügige Abschätzung – kann auch schlechter sein

2.4 Sortieren durch sortiertes Einfügen

⟶ **[Buch, Abschnitt 4.1.3, S.74-77]**

Natürlich kann der sukzessive Aufbau von sortierten Listen/Feldern auch zum Sortieren benutzt werden. Dies ist unten zunächst für die Felder aufgezeigt, wobei neue Elemente von hinten nach vorn eingefügt werden.

```
INSERTIONSORT(Feld A)
    Rückgabewert: nichts; Seiteneffekt: A ist sortiert
1   for i = 2,...,A.länge
2   do ⌜ neu ← A[i]
3        k ← i
4        while k > 1 und A[k−1].wert > neu.wert
5        do ⌜ A[k] ← A[k−1]
6             ⌞ k ← k−1
7      ⌞ A[k] ← neu
```

Beispiel:

A[1]	A[2]								A[10]
20	54	28	31	5	24	39	14	1	15

Satz: Beim Sortieren eines Felds A der Länge n werden genau

$$Inversionszahl(A) + (n-1)$$

Schreiboperationen durchgeführt. Die Zahl der Schlüsselvergleiche ist ebenfalls durch obige Formel nach oben beschränkt.

Zeitaufwand worst-case:	
Zeitaufwand best-case:	

Ordnungsverträglich?

Stabil?

Kapitel 3

Graphprobleme und Problemschwierigkeit

Holly Sargis: We planned a huge network of tunnels under the forrest floor,
and our first order of business every morning was
to decide on a new pathway for the day.
(Badlands, 1973)

3.1 Das Kürzeste-Wege-Problem

⟶ **[Buch, Abschnitt 1.4, S.7-10]**

Anwendungsszenario: Wenn in einer Filiale ein mobiler Service-Mitarbeiter für eine dringende Reparatur benötigt wird, muss anhand der gespeicherten Fahrtzeiten im abgelegten Straßennetz die nächstgelegene Filiale ermittelt werden, bei der sich ein derartiger Mitarbeiter aufhält.

Die grundlegende Fragestellung wird in der Definition in diesem Abschnitt so abstrahiert, dass für eine Filiale der kürzeste Weg zu allen anderen Filialen gesucht wird. Zur Beschreibung dieses und der zwei noch verbleibenden Problemen werden wir uns der Notation der Graphen bedienen.

Def.: Graph

- Graph $G = (V, E)$ definiert durch endliche Knotenmenge V und Menge der Kanten $E \subseteq V \times V$ als direkte Verbindungen
- *gerichtet* wenn jede Kante eine Richtung hat
  Notation: $(u, v) \in E$ mit $u, v \in V$ als Pfeil dargestellt
- *ungerichtet*, wenn Kanten in beide Richtungen durchlaufbar
  Notation: $\{u, v\} \in E$ als Strich dargestellt

Def: einfache Grapheigenschaften

- Geg.: Graph $G = (V,E)$ und ein Knoten $v \in V$
- falls G ungerichtet:
  Grad des Knotens: $grad(v) = \#\{u \in V | \{u,v\} \in E\}$
- falls G gerichtet:
  Eingangsgrad: $grad^+(v) = \#\{u \in U | (u,v) \in E\}$
  Ausgangsgrad: $grad^-(v) = \#\{u \in U | (v,u) \in E\}$

Def: Zusammenhang und Wege
Ein Graph $G = (V,E)$ heißt zusammenhängend, wenn es für alle Knotenpaare $v,w \in V$ mit $v \neq w$ einen Weg in G von v nach w gibt. Die Knotenfolge $u_0,\ldots,u_k \in V$ heißt dabei ein Weg (der Länge k) von u_0 nach u_k, wenn $(u_i,u_{i+1}) \in E$ (bzw. im ungerichteten Fall $\{u_i,u_{i+1}\} \in E$) für alle $0 \leq i < k$ gilt.

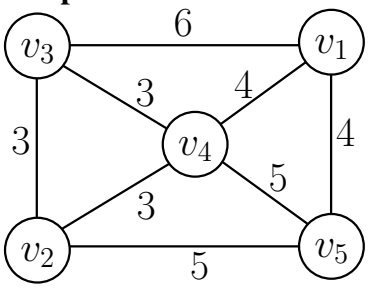
Ein Weg $u_0,\ldots,u_k \in V$ heißt knotendisjunkt, wenn alle Knoten paarweise verschieden sind ($u_i \neq u_j$ für alle $0 \leq i < j \leq k$), und kantendisjunkt, wenn jede Kante nur einmal durchlaufen wird ($(u_i,u_{i+1}) \neq (u_j,u_{j+1})$ für alle $0 \leq i < j < k$ bzw. im ungerichteten Fall $\{u_i,u_{i+1}\} \neq \{u_j,u_{j+1}\}$).

Def: Zyklen und Schleifen
Ein Graph $G = (V,E)$ heißt zyklenfrei, wenn es für keinen Knoten $v \in V$ einen kantendisjunkten Weg der Länge $k \geq 1$ von v nach v gibt. Als Sonderfall heißt G schleifenfrei, wenn die Menge E keine Kante (v,v) bzw. $\{v,v\}$ mit $v \in V$ enthält.

Def.: Kürzeste-Wege-Problem
Gegeben sei ein schleifenfreier Graph $G = (V,E)$, die Kostenfunktion $\gamma : E \to \mathbb{R}^+$ und ein Startpunkt $s \in V$. Dann berechnet die Operation KÜRZESTER-WEG für jeden Zielpunkt $t \in V \setminus \{s\}$ einen Weg $(s =)u_0,\ldots,u_m(= t)$ beliebiger Länge $m > 0$, aber mit minimalen Kosten

$$\sum_{j=0}^{m-1} \gamma(u_j,u_{j+1}).$$

Beispiel:



3.2 Das Rundreiseproblem

⟶ **[Buch, Abschnitt 1.5, S.10-11]**

Anwendungsszenario: Ein Mitarbeiter, der „technische Inspektor", besucht nacheinander alle Filialen. Um seine Reisekosten möglichst gering zu halten, müssen die Filiale in eine Reihenfolge gebracht werden, die unnötige lange Wege zwischen zwei direkt nacheinander besuchten Filialen vermeidet. Am Ende muss jede Filiale genau einmal besucht und der Inspektor wieder an seinem Ausgangspunkt angekommen sein.

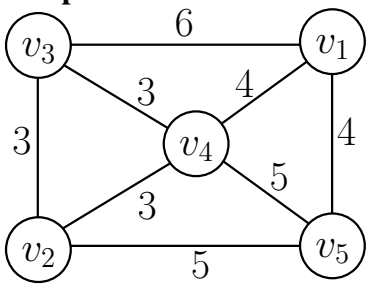
Def: Rundreiseproblem
Gegeben sei ein zusammenhängender Graph $G = (V,E)$ mit $V = \{v_1,\ldots,v_n\}$ und die Kostenfunktion $\gamma : E \to \mathbb{R}^+$. Die Operation RUNDREISE soll die Knoten so als Weg durch eine Permutation $\pi : \{1,\ldots,n\} \to \{1,\ldots,n\}$ anordnen, dass die Summe der Kosten für die Rundreise

$$\left(\sum_{j=1}^{n-1} \gamma(v_{\pi(j)}, v_{\pi(j+1)})\right) + \gamma(v_{\pi(n)}, v_{\pi(1)})$$

minimal ist.

- ungerichteter Graph: symmetrisches Rundreiseproblem
- gerichteter Graph mit $\exists u,v \in V : \gamma(u,v) \neq \gamma(v,u)$: asymmetrisches Rundreiseproblem

Beispiel:



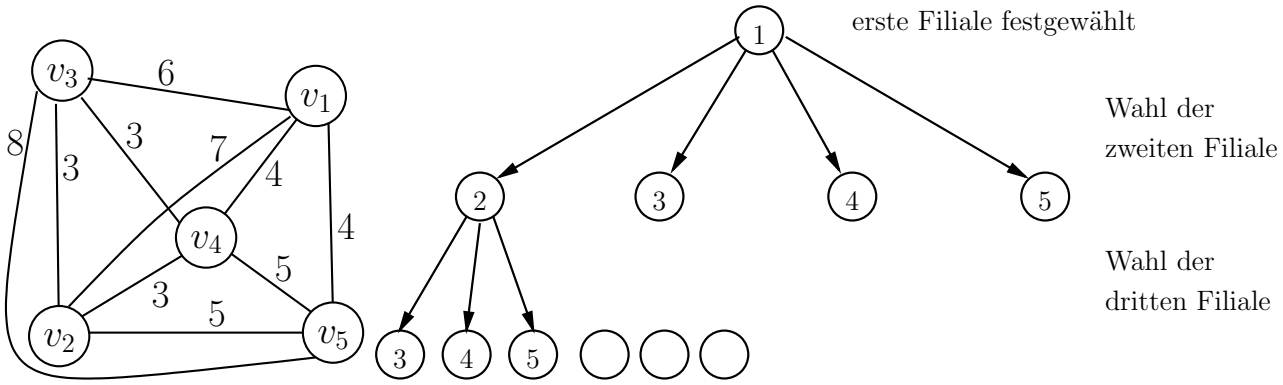
I.d.R.: vollständiger Graph

3.3 Backtracking als allgemeine Lösungsstrategie

⟶ **[Buch, Abschnitt 3.3, S.59-62]**

Da uns zunächst an dieser Stelle nichts besseres einfällt, bleibt nur das systematische Durchprobieren von allen möglichen Wegen, Rundreisen etc.

Dies machen wir beispielhaft für das Rundreiseproblem und betrachten einen Entscheidungsbaum, über den alle Rundreisen nacheinander aufgezählt werden können.



In einem globalen Feld wird schrittweise immer eine Rundreise zusammengesetzt.

```
BACKTRACKING-TSP(Graph G mit n Knoten)
    Rückgabewert: Reihenfolge der Knoten in kürzesteRundreise
 1  minLänge ← ∞
 2  for k ← 1,...,n
 3  do ⌊ rundreise[k] ← k
 4  BACKTRACKING-TSP-R(n,2)
 5  return kürzesteRundreise

BACKTRACKING-TSP-R(Anzahl n, Tiefe i)
    Rückgabewert: nichts; Seiteneffekt: Änderung in kürzesteRundreise
 1  if i ≤ n
 2  then ⌈ for a ← i,...,n
 3         do ⌈ VERTAUSCHE(rundreise,i,a)
 4              BACKTRACKING-TSP-R(n,i+1)
 5       ⌊    ⌊ VERTAUSCHE(rundreise,i,a)
 6  else ⌈ if (Kosten von rundreise) < minLänge
 7         then ⌈ minLänge ← Kosten von rundreise
 8              ⌈ for i ← 1,...,n
 9       ⌊      ⌊ do ⌊ kürzesteRundreise[i] ← rundreise[i]
```

Theorem: Die Laufzeit von BACKTRACKING-TSP ist in $\Theta((n-1)!)$.

3.4 Schwierigkeit von Problemen

$\longrightarrow$ **[Buch, Abschnitt 2.4, S.32-35]**

Def.: Entscheidungsprobleme Bei einem Entscheidungsproblem wird zu einer Probleminstanz und einer Eigenschaft durch einen Algorithmus entschieden, ob eine entsprechende Lösung mit der Eigenschaft existiert.

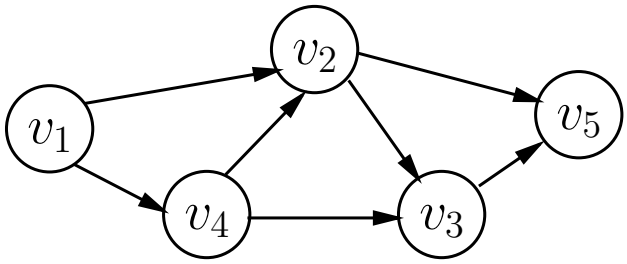
Die Klasse P Ein Entscheidungsproblem ist genau dann in der Klasse P enthalten, wenn es einen klassischen Algorithmus gibt, der die Antwort in polynomieller Zeit berechnet. Dabei bedeutet polynomielle Zeit, dass es ein $k \in \mathbb{N}$ gibt, so dass der Algorithmus in $\mathcal{O}(n^k)$ ist.

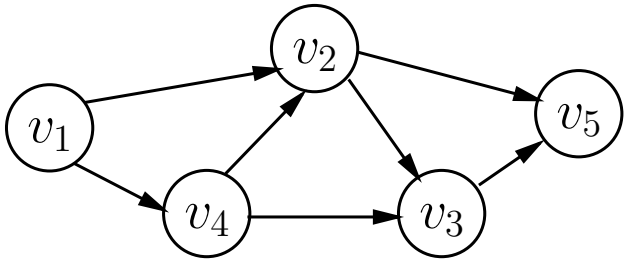
Die Klasse NP: Ein Entscheidungsproblem ist genau dann in der Klasse NP, wenn es für sie einen Algorithmus mit Polynomialzeit gibt, der zusätzlich einen Teil der Lösung „raten“ darf und dann gleichzeitig für alle geratenen Teillösungen die Eigenschaft überprüft.

NP-Vollständigkeit: Ein Problem p heißt NP-vollständig, wenn es in NP enthalten ist und sich für jedes Problem $p' \in NP$ jede Probleminstanz aus p' durch einen Algorithmus in polynomieller Zeit auf eine Probleminstanz aus p abbilden lässt, sodass man von der Lösung für p auf die Lösung für p' schließen kann.

Beispiel:

- Lässt sich KÜRZESTER-WEG auf RUNDREISE reduzieren?





- Lässt sich RUNDREISE auf KÜRZESTER-WEG reduzieren?

3.5 Datenstrukturen für Graphen

⟶ **[Buch, Abschnitt 4.5.1, S.95-99]**

Def.: Eine Adjazenzmatrix für einen Graphen $G = (V,E)$ mit $V = \{1,\ldots,n\}$ ist eine $n \times n$-Matrix A, in der

$$A[i,j] = 1 \Leftrightarrow (i,j) \in E.$$

```
ALLE-KANTEN-ADJAZENZMATRIX(Graph G mit n Knoten und Adjazenzmatrix A)
    Rückgabewert: keiner
1   for i ← 1,...,n
2   do ⌜ for j ← 1,...,n
3          do ⌜ if A[i,j] = 1
4          ⌞     ⌞ then ⌑ drucke „Kante (i,j)“
```

Laufzeit:

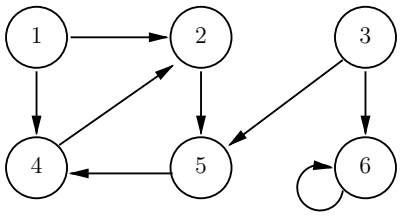
Weitere typische Graphoperationen

- Test, ob eine Kante existiert, bzw. das Setzen oder Löschen einer Kante
- Ausgabe aller ausgehenden Kanten eines Knotens
- Hinzufügen oder Löschen eines Knotens zu der Menge V

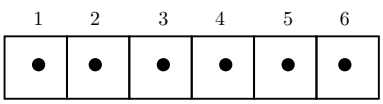
Def.: Eine Adjazenzliste für einen Graphen $G = (V,E)$ mit $V = \{1,\ldots,n\}$ besteht aus einem Feld mit einem Eintrag für jeden Knoten, welcher auf eine Liste der beim jeweiligen Knoten startenden Kanten verweist.

```
ALLE-KANTEN-ADJAZENZLISTE(Graph G mit n Knoten und Adjazenzliste A)
    Rückgabewert: keiner
1   for i ← 1,...,n
2   do ⌜ eℓ → A[i]
3          while eℓ ≠ NULL
4          do ⌜ drucke „Kante (i,eℓ.wert)“
5          ⌞      ⌞ eℓ → eℓ.nächster
```

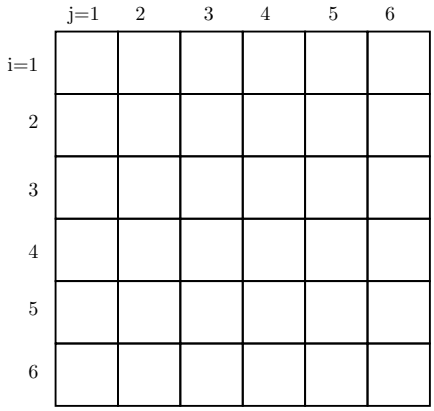
Laufzeit:



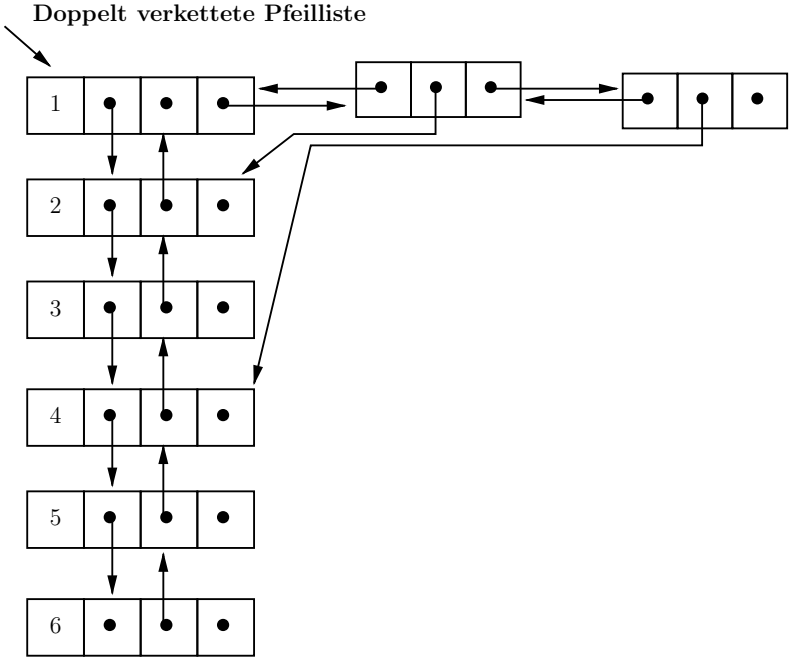
Adjazenzliste

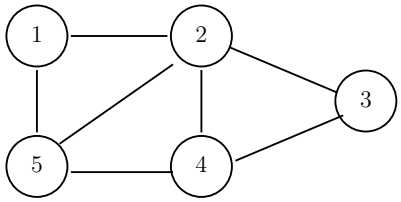


Adjazenzmatrix

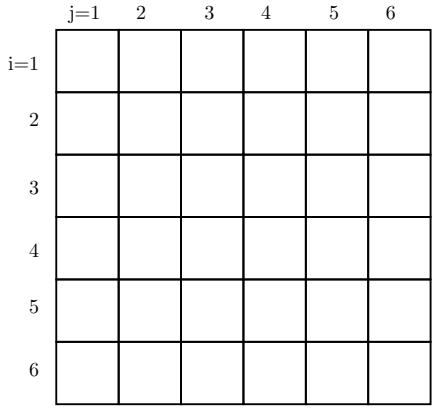


Doppelt verkettete Pfeilliste

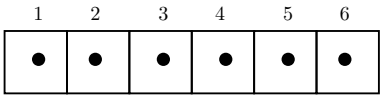




Adjazenzmatrix



Adjazenzliste



Operation	Adjazentmatrix	Adjazenzliste	doppelt verkettete Pfeilliste
Kante prüfen/einfügen/löschen			
alle Kanten eines Knotens ausgeben			
alle Kanten ausgeben			
Knoten einfügen/löschen			

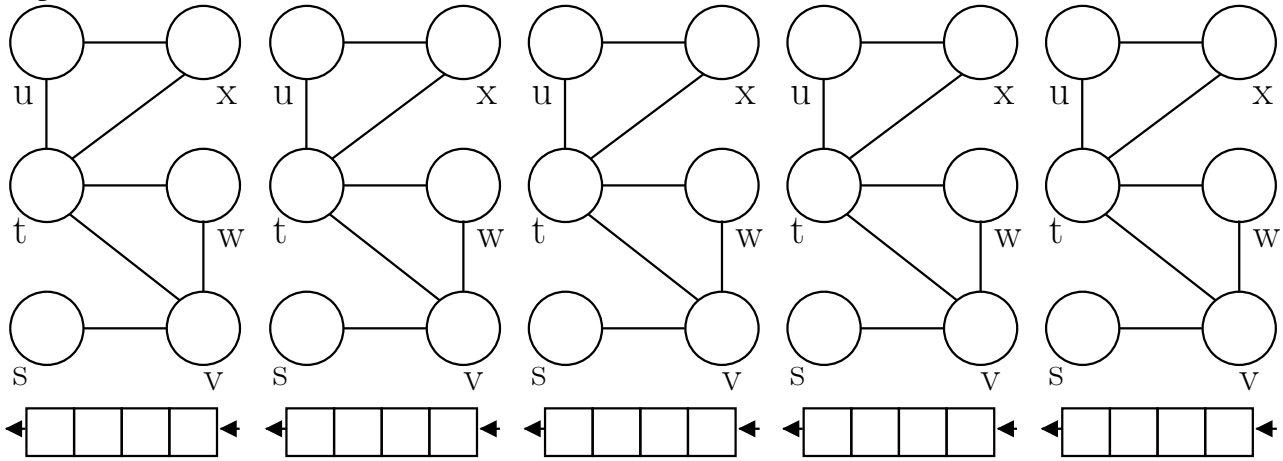
3.6 Kürzeste Wege – ein vereinfachter Versuch

⟶ **[Buch, Abschnitt 4.5.2, S.99-102]**

```
BREITENSUCHE(Knoten V, Kanten E, Knoten start)
     Rückgabewert: Distanzen abstand, Weg vorgänger
 1   for alle Knoten u ∈ V \ {start}
 2   do ⌜ istBekannt[u] ← false
 3        abstand[u] ← ∞
 4      ⌞ vorgänger[u] → NULL
 5   abstand[start] ← 0
 6   istBekannt[start] ← true
 7   Q → allokiere Warteschlange
 8   Q.EINFÜGEN(start)
 9   while ¬Q.ISTLEER
10   do ⌜ u → Q.NÄCHSTESELEMENT
11        Q.ENTFERNEN
12        for alle v ∈ V mit (u,v) ∈ E
13        do ⌜ if ¬istBekannt[v]
14             then ⌜ abstand[v] ← abstand[u] + 1
15                    vorgänger[v] → u
16                    istBekannt[v] ← true
17      ⌞    ⌞      ⌞ Q.EINFÜGEN(v)
18   return (abstand, vorgänger)
```

Satz: Die Laufzeit für die Breitensuche wird bestimmt durch die Anzahl der Knoten $|V|$ sowie für jeden Knoten die Zeit, alle ausgehenden Kanten zu prüfen.

Bsp. Breitensuche:



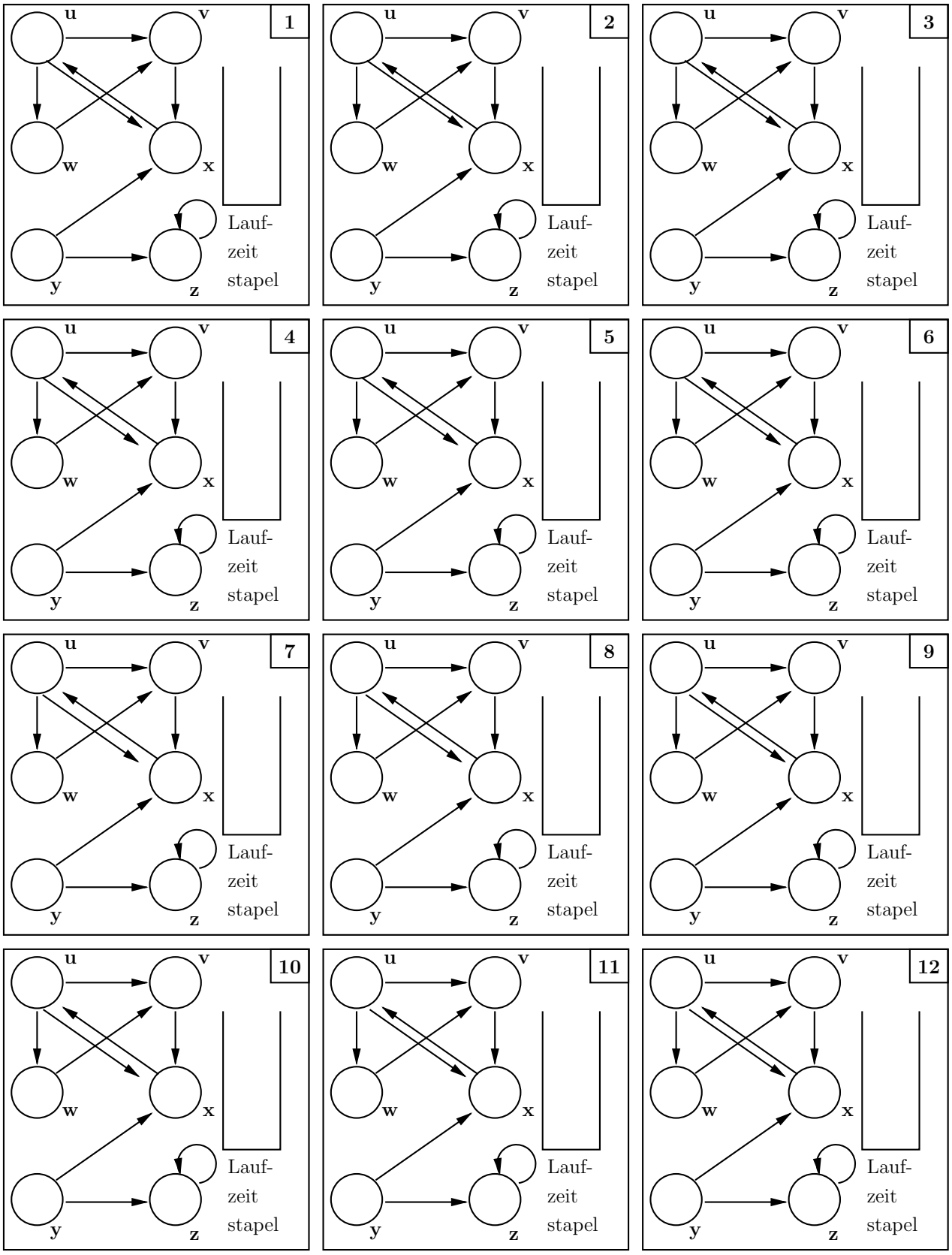
3.7 Berechnung von Grapheigenschaften

⟶ **[Buch, Abschnitt 4.5.3, S.102-106]**

```
TIEFENSUCHE(Knoten V, Kanten E)
     Rückgabewert: Zeiten entdeckt, fertig, Kanten des Tiefensuchbaums vorgänger
 1   for alle Knoten u ∈ V
 2   do ⌜ istBekannt[u] ← false
 3        entdeckt[u] ← 0
 4        fertig[u] ← 0
 5      ⌞ vorgänger[u] → NULL
 6   zeit ← 0
 7   for alle Knoten u ∈ V
 8   do ⌜ if ¬istBekannt[u]
 9      ⌞ then ⌊ EXPANDIERE(u)
10   return (entdeckt, fertig, vorgänger)
```

```
EXPANDIERE(Knoten u)
     Rückgabewert: nichts; Seiteneffekte in entdeckt, fertig, vorgänger
 1   istBekannt[u] ← true
 2   zeit ← zeit + 1
 3   entdeckt[u] ← zeit
 4   for alle v ∈ V mit (u, v) ∈ E
 5   do ⌜ if ¬istBekannt[v]
 6        then ⌜ vorgänger[v] → u
 7      ⌞      ⌞ EXPANDIERE(v)
 8   zeit ← zeit + 1
 9   fertig[u] ← zeit
```

Bsp. Tiefensuche:



Kanteklassifikation:

- Baumkante:

- Rückkante:

- Vorwärtskante:

- Querkante:

Effizienz der bisherigen Lösungen

Geschätzte Laufzeiten der einfachen Algorithmen:

	$n = 50$	$n = 120$	$n = 10^4$	$n = 10^6$
Operation auf der verketteten Liste (Worst-case)				
Bubblesort (Worst-case)				
Rundreiseproblem mit Backtracking				

Kapitel 4

Verbesserung durch mehr Struktur: Sortierung

Sally: I'm difficult.
Harry: You're challenging.
Sally: I'm too structured, I'm completely closed off.
Harry: But in a good way.
(When Harry Met Sally..., 1989)

4.1 Sortiert in einem Feld (Array)

⟶ **[Buch, Abschnitt 4.1.1, S.69-71]**

```
BINÄRE-SUCHE(Schlüssel gesucht)
     Rückgabewert: gesuchte Daten bzw. Fehler falls nicht enthalten
 1   links ← 1
 2   rechts ← belegteFelder
 3   while links ≤ rechts
 4   do ⌜ mitte ← ⌊(links+rechts)/2⌋
 5        switch
 6        case A[mitte].wert = gesucht :
 7             return A[mitte].daten
 8        case A[mitte].wert > gesucht :
 9             rechts ← mitte − 1
10        case A[mitte].wert < gesucht :
11   ⌞         links ← mitte + 1
12   error "Element nicht gefunden"
```

Beispiel: FIND(24) im Feld

A[1]	A[2]								A[10]
1	5	14	15	20	24	28	31	39	54

```
EINFÜGEN-SORTFELD(Schlüssel neuerWert, Daten neueDaten)
     Rückgabewert: nichts falls erfolgreich bzw. Fehler sonst
 1   links ← 1
 2   rechts ← belegteFelder
 3   while links ≤ rechts
 4   do ⌜ mitte ← ⌊(links+rechts)/2⌋
 5        switch
 6        case A[mitte].wert = neuerWert :
 7             error "Element schon enthalten"
 8        case A[mitte].wert > neuerWert :
 9             rechts ← mitte − 1
10        case A[mitte].wert < neuerWert :
11   ⌞         links ← mitte + 1
12   for index ← belegteFelder, ..., links
13   do ⌜ A[index + 1].wert ← A[index].wert
14      ⌞ A[index + 1].daten ← A[index].daten
15   A[links].wert ← neuerWert
16   A[links].daten ← neueDaten
17   belegteFelder ← belegteFelder + 1
```

Suchen:

Einfügen:

Löschen:

Alle:

4.2 Sortiert in verketteter Liste

⟶ **[Buch, Abschnitt 4.1.2, S.71-74]**

```
INIT-SORTLISTE()
     Rückgabewert: nichts
 1   anker → allokiere Element(⊥, ⊥, NULL)
```

```
SUCHEN-SORTLISTE(Schlüssel gesucht)
    Rückgabewert: gesuchte Daten bzw. Fehler falls nicht enthalten
1   el → anker.nächstes
2   while el ≠ NULL
3   do ⌜ switch
4          case el.wert = gesucht : return el.daten
5          case el.wert < gesucht : el → el.nächstes
6      ⌟ case el.wert > gesucht : error "Element nicht gefunden"
7   error "Element nicht gefunden"
```

```
EINFÜGEN-SORTLISTE(Schlüssel neuerWert, Daten neueDaten)
    Rückgabewert: nichts falls erfolgreich bzw. Fehler sonst
1   el → anker
2   while el.nächstes ≠ NULL und el.nächstes.wert < neuerWert
3   do ⌊ el → el.nächstes
4   if el.nächstes = NULL oder el.nächstes.wert > neuerWert
5   then ⌊ el.nächstes → allokiere Element(neuerWert, neueDaten, el.nächstes)
6   else ⌊ error "Element schon enthalten"
```

```
LÖSCHEN-SORTLISTE(Schlüssel löschWert)
    Rückgabewert: nichts falls erfolgreich bzw. Fehler sonst
1   el → anker
2   while el.nächstes ≠ NULL und el.nächstes.wert < löschWert
3   do ⌊ el → el.nächstes
4   if el.nächstes ≠ NULL und el.nächstes.wert = löschWert
5   then ⌊ el.nächstes → el.nächstes.nächstes
6   else ⌊ error "nicht enthalten"
```

Suchen:

Einfügen:

Löschen:

Alle:

4.3 Skip-Listen (Listen mit Abkürzungen)

⟶ **[Buch, Abschnitt 4.2, S.77-81]**

Was macht die Felder den Listen überlegen?

Was ist ein Vorteil der Listen gegenüber Feldern?

Idee: Die Zugriffsstruktur der binären Suche wird über Abkürzungen in die Liste eingeführt.

Datenstruktur:

Speichern den aktuellen maximalen Level *aktlevel*.

```
SUCHEN-SKIPLISTE(Schlüssel gesucht)
    Rückgabewert: gesuchte Daten bzw. Fehler falls nicht enthalten
1   el → anker
2   for ebene ← anzEbenen,...,1
3   do ⌜ while el.nächstes[ebene] ≠ NULL und el.nächstes[ebene].wert < gesucht
4      ⌟ do ⌊ el → el.nächstes[ebene]
5   if el.nächstes[1] = NULL oder el.nächstes[1].wert ≠ gesucht
6   then ⌊ error "Element nicht gefunden"
7   else ⌊ return el.nächstes[1]
```

```
VORGÄNGER-IN-SKIPLISTE(Schlüssel gesucht)
    Rückgabewert: Feld mit den Verweisen auf die Vorgänger auf allen Ebenen
1   el → anker
2   vorgänger → allokiere Feld vom Typ Knoten der Länge maxEbenen
3   for ebene ← anzEbenen,...,1
4   do ⌜ while el.nächstes[ebene] ≠ NULL und el.nächstes[ebene].wert < gesucht
5        do ⌊ el → el.nächstes[ebene]
6      ⌟ vorgänger[ebene] → el
7   return vorgänger
```

```
EINFÜGEN-SKIPLISTE(Schlüssel neuerWert, Daten neueDaten)
    Rückgabewert: nichts falls erfolgreich bzw. Fehler sonst
 1  vorgänger → VORGÄNGER-IN-SKIPLISTE(neuerWert)
 2  eℓ → vorgänger[1]
 3  if eℓ.nächstes[1] ≠ NULL und eℓ.nächstes[1].wert = neuerWert
 4  then ⌊ error "Element schon enthalten"
 5  else ⌈ knotenEbene ← 1
 6         while random(0...1) < 1/2 und knotenEbene < maxEbenen
 7         do ⌊ knotenEbene ← knotenEbene + 1
 8         if knotenEbene > anzEbenen
 9         then ⌈ for ebene ← anzEbenen + 1, ..., knotenEbene
10                do ⌈ vorgänger[ebene] → anker
11                   ⌊ anker.nächstes[ebene] → NULL
12              ⌊ anzEbenen ← knotenEbene
13         neuesElem → allokiere Element(neuerWert, neueDaten)
14         neuesElem.nächstes → allokiere Feld der Länge knotenEbene
15         for ebene ← 1, ..., knotenEbene
16         do ⌈ neuesElem.nächstes[ebene] → vorgänger[ebene].nächstes[ebene]
17       ⌊    ⌊ vorgänger[ebene].nächstes[ebene] → neuesElem
```

```
LÖSCHEN-SKIPLISTE(Schlüssel löschWert)
    Rückgabewert: nichts falls erfolgreich bzw. Fehler sonst
1  vorgänger → VORGÄNGER-IN-SKIPLISTE(löschWert)
2  eℓ → vorgänger[1]
3  if eℓ.nächstes[1] = NULL oder eℓ.nächstes[1].wert ≠ löschWert
4  then ⌊ error " Element nicht gefunden "
5  else ⌈ for ebene ← 1, ..., eℓ.nächstes[1].länge
6         do ⌊ vorgänger[ebene].nächstes[ebene] →
                  vorgänger[ebene].nächstes[ebene].nächstes[ebene]
7         while anzEbenen > 1 und anker.nächstes[anzEbenen] = NULL
8       ⌊ do ⌊ anzEbenen ← anzEbenen − 1
```

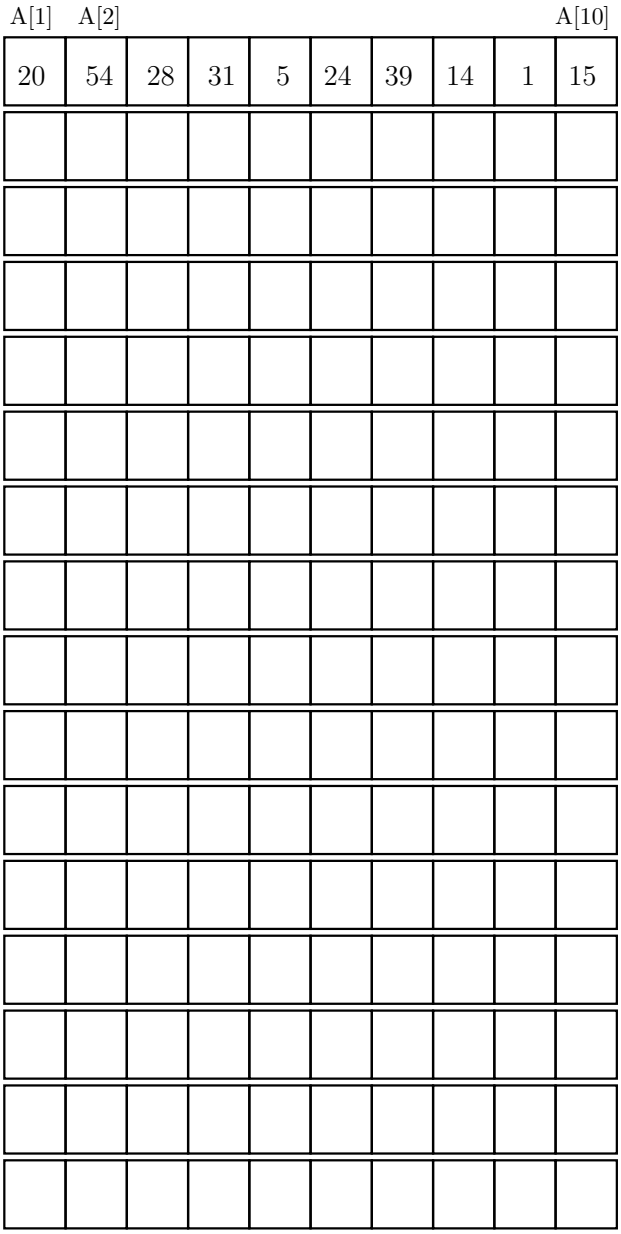
Welche Laufzeiten ergeben sich im Mittel?

4.4 Abkürzungen beim Sortieren (Shellsort)

⟶ **[Buch, Abschnitt 4.3, S.81-86]**

```
SHELLSORT(Feld A)
     Rückgabewert: nichts; Seiteneffekt: A ist sortiert
 1   schrittweite ← 1
 2   while 3 · schrittweite + 1 < A.länge
 3   do ⌈ schrittweite ← 3 · schrittweite + 1
 4   while schrittweite > 0
 5   do ⌈ for i ← schrittweite + 1, ..., A.länge
 6        do ⌈ neu ← A[i]
 7             k ← i
 8             while k > schrittweite und A[k − schrittweite].wert > neu.wert
 9             do ⌈ A[k] ← A[k − schrittweite]
10                ⌊ k ← k − schrittweite
11             ⌊ A[k] ← neu
12        ⌊ schrittweite ← ⌊schrittweite/3⌋
```

Beispiel:



Satz: Shellsort hat in der obigen Version eine Laufzeit von $\mathcal{O}(\sqrt{n^3})$ im worst-case.

Urheber	Formel	Zahlenfolge	worst-case
Shell (1959)	2^i	1, 2, 4, 8, 16, 32, 64, 128, ...	$\mathcal{O}(n^2)$
Hibbard (1963)	$2^{i+1}-1$	1, 3, 7, 15, 31, 63, 127, ...	$\mathcal{O}(\sqrt{n^3})$
Knuth (1973)	$s_{k+1} = 3 \cdot s_k + 1$	1, 4, 13, 40, 121, 364, ...	$\mathcal{O}(\sqrt{n^3})$
Pratt (1971)	$2^i \cdot 3^j$	1, 2, 3, 4, 6, 8, 9, 12, 18, ...	$\mathcal{O}(n \cdot (\log n)^2)$
Sedgewick (1986)	$s_1 = 1$ und $4^{i+1} + 3 \cdot 2^i + 1$ $(i \geq 0)$	1, 8, 23, 77, 281, 1073, ...	$\mathcal{O}(\sqrt[3]{n^4})$

4.5 Binäre Suchbäume

$\longrightarrow$ **[Buch, Abschnitt 4.4, S.86-95]**

Def.: binärer Baum

- Ein *binärer Baum* ist
  - leer oder
  - enthält Schlüssel, Verweise auf linken und rechten binären (Unter-)Baum
- Knoten mit Eingangsgrad 0 heipt *Wurzel*
- Knoten mit zwei leeren Unterbäumen heißen *Blatt*
- *Tiefe eines Knotens* $e\ell$ im Baum b:

  $$\mathit{Tiefe}(b, e\ell) = 1 + \#\text{Kanten zwischen } e\ell \text{ und der Wurzel von } b.$$

  Die *Tiefe des Baums* ist die maximale Tiefe eines seiner Knoten.

Def.: binärer Suchbaum
Ein binärer Baum ist ein binärer Suchbaum, wenn für jeden Knoten im Baum gilt, dass alle Knoten im linken Unterbaum einen kleineren Schlüssel und alle Knoten im rechten Unterbaum einen Schlüssel größergleich dem Wurzelknoten besitzen.

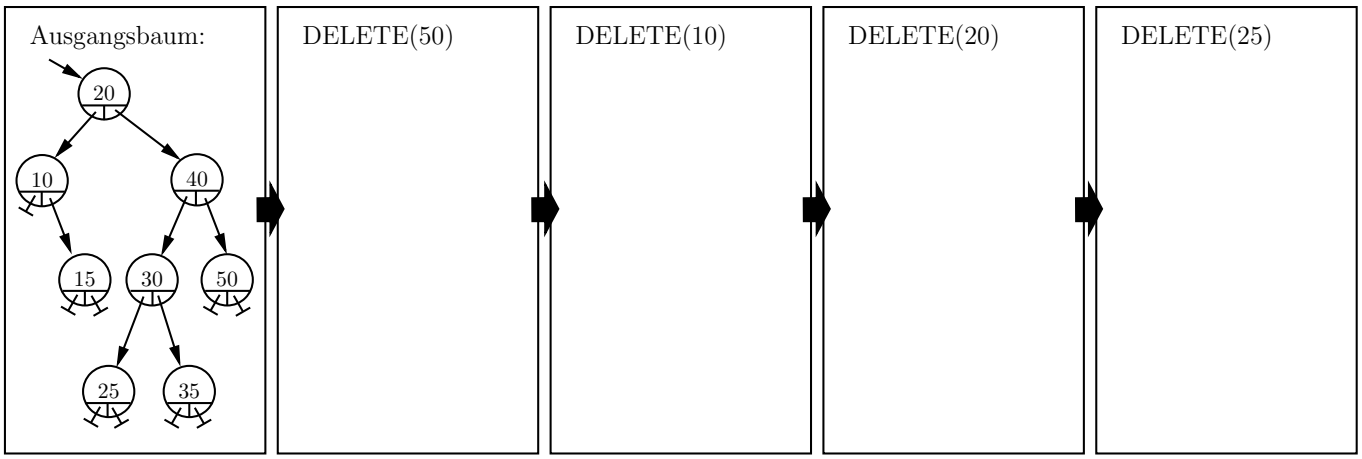
```
SUCHEN-BAUM(Schlüssel gesucht)
    Rückgabewert: gesuchte Daten bzw. Fehler falls nicht enthalten
1   el → anker
2   while el ≠ NULL und gesucht ≠ el.wert
3   do ⌜ if gesucht < el.wert
4          then ⌊ el → el.links
5        ⌞ else ⌊ el → el.rechts
6   if el = NULL
7   then ⌊ error "Element nicht gefunden"
8   else ⌊ return el.daten
```

```
EINFÜGEN-BAUM(Schlüssel neuerWert, Daten neueDaten)
    Rückgabewert: nichts falls erfolgreich bzw. Fehler sonst
 1  anker → EINFÜGEN-BAUM-R(neuerWert, neueDaten, anker)

EINFÜGEN-BAUM-R(Schlüssel neuerWert, Daten neueDaten, Element eℓ)
    Rückgabewert: neues erstes Element bzw. Fehler sonst
 1  switch
 2  case eℓ = NULL :
 3      return allokiere Element(neuerWert, NULL, NULL)
 4  case neuerWert = eℓ.wert :
 5      error "Element schon enthalten"
 6  case neuerWert < eℓ.wert :
 7      eℓ.links → EINFÜGEN-BAUM-R(neuerWert, neueDaten, eℓ.links)
 8      return eℓ
 9  case neuerWert ≥ eℓ.wert :
10      eℓ.rechts → EINFÜGEN-BAUM-R(neuerWert, neueDaten, eℓ.rechts)
11      return eℓ
```

INSERT(20)	INSERT(15)	INSERT(30)	INSERT(21)	INSERT(25)

```
LÖSCHEN-BAUM(Schlüssel löschWert)
    Rückgabewert: nichts falls erfolgreich bzw. Fehler sonst
 1  anker → LÖSCHEN-BAUM-R(löschWert, anker)

LÖSCHEN-BAUM-R(Schlüssel löschWert, Element eℓ)
    Rückgabewert: neues erstes Element bzw. Fehler sonst
 1  switch
 2  case eℓ = NULL :
 3       error "Element nicht gefunden"
 4  case löschWert < eℓ.wert :
 5       eℓ.links → LÖSCHEN-BAUM-R(löschWert, eℓ.links)
 6       return eℓ
 7  case löschWert > eℓ.wert :
 8       eℓ.rechts → LÖSCHEN-BAUM-R(löschWert, eℓ.rechts)
 9       return eℓ
10  case löschWert = eℓ.wert :
11       if eℓ.links = NULL
12       then ⌊ return eℓ.rechts
13       else ⌜ if eℓ.rechts = NULL
14              then ⌊ return eℓ.links
15              else ⌜ ersatz → SUCHE-NACHFOLGER(eℓ)
16                     ersatz.links → eℓ.links
17                     ersatz.rechts → eℓ.rechts
18            ⌞       ⌞ return ersatz
```

```
SUCHE-NACHFOLGER(Element eℓ)
    Rückgabewert: Verweis auf Ersatzknoten
1  if eℓ.rechts.links = NULL
2  then ⌜ ersatz → eℓ.rechts
3       ⌞ eℓ.rechts → eℓ.rechts.rechts
4  else ⌜ eℓ → eℓ.rechts
5         while eℓ.links.links ≠ NULL
6         do ⌊ eℓ → eℓ.links
7         ersatz → eℓ.links
8       ⌞ eℓ.links → eℓ.links.rechts
9  return ersatz
```



	Suchen	Einfügen	Löschen
Zeitaufwand worst-case:			
Zeitaufwand best-case:			

Achtung: Ein durch Einfügen von zufälligen Zahlen erzeugter Baum hat die durchschnittliche Suchzeit $1.386 \log_2 n - 1.846$ (ohne Beweis). Dies gilt nicht mehr bei einem durch Löschen und Einfügen modifizierten Baum!

Traversierung von Bäumen

```
ALLE-BAUM-INORDER(Element el)
    Rückgabewert: nichts, da direkte Ausgabe
1   if el ≠ NULL
2   then ⌜ ALLE-BAUM-INORDER(el.links)
3        |  drucke: el.daten
4        ⌞ ALLE-BAUM-INORDER(el.rechts)
```

```
ALLE-BAUM-PRÄORDER(Element el)
    Rückgabewert: nichts, da direkte Ausgabe
1   if el ≠ NULL
2   then ⌜ drucke: el.daten
3        |  ALLE-BAUM-PRÄORDER(el.links)
4        ⌞ ALLE-BAUM-PRÄORDER(el.rechts)
```

Kapitel 5 Gierige Algorithmen

Monsieur Hood: I steal from the rich and give to the needy...
Merry Man: He takes a wee percentage...
Monsieur Hood: But I'm not greedy –
I rescue pretty damsels, man I'm good!
(Shrek, 2001)

Idee der Greedy-Algorithmen

Häufig wird bei Optimierungsproblemen eine Lösung schrittweise aufgebaut. Wird bei jedem Schritt ausschließlich lokal entschieden, was den meisten Nutzen bringt, spricht man von einem Greedy-Algorithmus.

5.1 Sortieren durch Auswählen

⟶ **[Buch, Abschnitt 5.2, S.118-121]**

```
SELECTIONSORT(Feld A)
    Rückgabewert: nichts; Seiteneffekt: A ist sortiert
1   for i ← A.länge,...,2
2   do ⌜ pos ← i
3        for j ← 1,...,i−1
4        do ⌜ if A[j] > A[pos]
5             ⌞ then ⌊ pos ← j
6        if pos ≠ i
7      ⌞ then ⌊ VERTAUSCHE(A, pos, i)
```

Beispiel:

A[1]	A[2]								A[10]
20	54	28	31	5	24	39	14	1	15

5.2 Kürzester Weg: Dijkstra-Algorithmus

⟶ **[Buch, Abschnitt 5.3, S.121-125]**

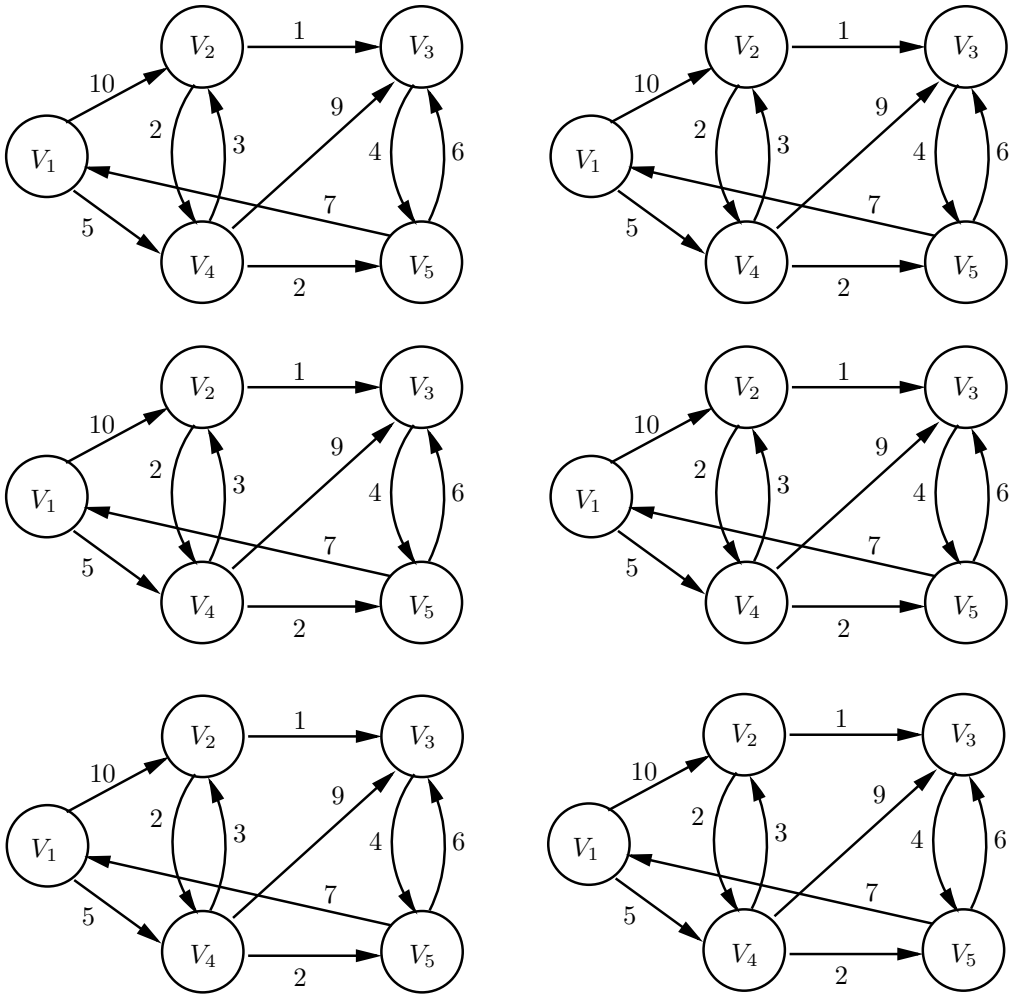
DIJKSTRA(Knoten V, Kanten $E \subset V \times V$, Kosten $\gamma : E \to \mathbb{R}$, Startknoten $s \in V$)
Rückgabewert: Distanzen *abstand*, Wege *vorgänger*

```
 1  for alle Knoten u ∈ V
 2  do ⌜ abstand[u] ← ∞
 3        vorgänger[u] → NULL
 4     ⌞ istFertig[u] ← false
 5  abstand[s] ← 0
 6  while KNOTEN-VORHANDEN-DIJKSTRA()
 7  do ⌜ nächster ← NÄCHSTER-KNOTEN-DIJKSTRA()
 8        istFertig[nächster] ← true
 9        for alle  v ∈ V mit (nächster, v) ∈ E
10        do ⌜ if abstand[v] > abstand[nächster] + γ[nächster, v]
11              then ⌜ abstand[v] ← abstand[nächster] + γ[nächster, v]
12     ⌞     ⌞        ⌞ vorgänger[v] → nächster
13  return abstand, vorgänger
```

```
KNOTEN-VORHANDEN-DIJKSTRA()
    Rückgabewert: Info, ob noch Knoten auswählbar sind
1   for v ∈ V
2   do ⌜ if abstand[v] < ∞ und istFertig[v] = false
3      ⌟ then ⌞ return true
4   return false
```

```
NÄCHSTER-KNOTEN-DIJKSTRA()
    Rückgabewert: nächster Knoten oder Nullzeiger
1   minimumWert ← ∞
2   nächster → NULL
3   for alle  v ∈ V
4   do ⌜ if ¬istFertig[v] und abstand[v] < minimumWert
5         then ⌜ minimumWert ← abstand[v]
6      ⌟       ⌟ nächster → v
7   return nächster
```

Zeitaufwand:

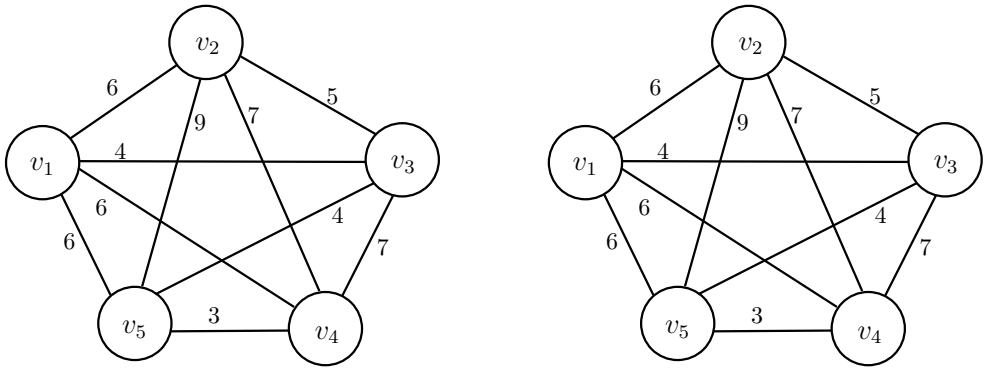


Knoten									
v_1		v_2		v_3		v_4		v_5	
a.	v.	a.	v.	a.	v.	a.	v.	a.	v.
0	–	∞	–	∞	–	∞	–	∞	–

5.3 Minimale Spannbäume für die Rundreise

⟶ **[Buch, Abschnitt 5.4, S.125-135]**

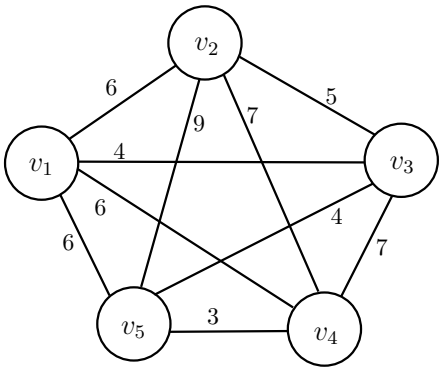
```
RUNDREISE-GREEDY(Knoten V, Kanten E, Kosten γ: E → ℝ)
     Rückgabewert: Reihenfolge der Knoten in rundreise
 1   for alle Knoten u ∈ V
 2   do ⌊ istFertig[u] ← false
 3   rundreise[1] → wähle einen Knoten aus V
 4   istFertig[rundreise[1]] ← true
 5   for index ← 1,...,#V − 1
 6   do ⌈ minAbstand ← ∞
 7        for alle v ∈ V
 8        do ⌈ if ¬istFertig[v] ∧ γ[rundreise[index], v] < minAbstand
 9             then ⌈ nächster → v
10             ⌊      ⌊ minAbstand ← γ[rundreise[index], v]
11     ⌊ rundreise[index + 1] → nächster
12   return rundreise
```

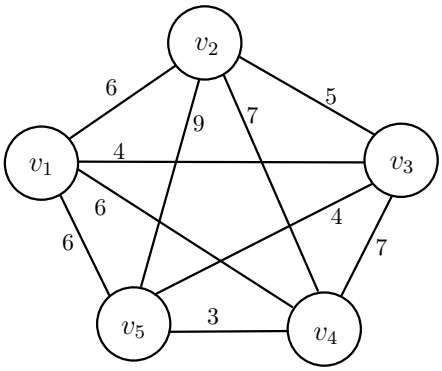


Motivation: Um kürzeste Rundreisen zu approximieren, wird hier zunächst der Begriff des Spannbaums eingeführt. Das Problem der minimalen Spannbäume ist auch unabhängig davon interessant, wann immer ein möglichst günstiges Grundgerüst in einen Graphen eingepasst werden muss, über das alle Knoten miteinander verbunden sind – ungeachtet der tatsächlichen Längen der Verbindungen. Vorsicht: Wege in minimalen Spannbäumen sind häufig wesentlich länger als kürzeste Wege im Graphen.

Def. Minimaler Spannbaum:

- Geg.: ungerichteten, zusammenhängenden Graphen $G = (V, E)$ und Kostenfunktion $\gamma : E \to \mathbb{R}^+$
- $T \subseteq E$ heißt *Spannbaum*, wenn $G' = (V, T)$ zusammenhängend und zyklenfrei
- ein Spannbaum heißt *minimaler Spannbaum* gdw. $\sum_{e \in T} \gamma(e)$ ist minimal



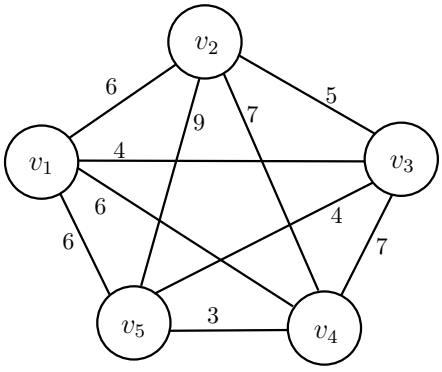


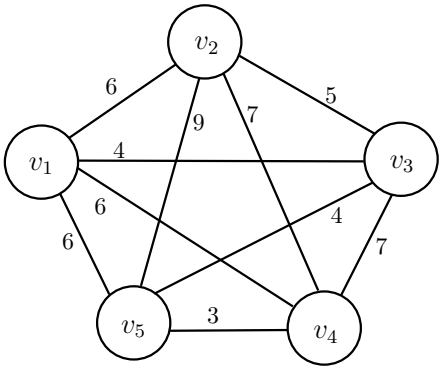
Approximation einer kürzesten Rundreise:
Voraussetzungen:

1. $G = (V,E)$ ist ungerichtet, schleifenfrei und vollständig
2. Dreiecksungleichung bezüglich der Kosten im Graphen, d. h. für $u,v,w \in V$ mit $u \neq v$, $v \neq w$ und $u \neq w$ gilt: $\gamma(u,w) \leq \gamma(u,v) + \gamma(v,w)$.

RUNDREISE-APPROXIMATION(Knoten V, Kanten E, Kosten $\gamma : E \to \mathbb{R}$)
Rückgabewert: Reihenfolge der Knoten in *kürzesteRundreise*
1 *wurzel* $\rightarrow$ wähle einen Knoten aus V
2 PRIM($V, Adj, \gamma, wurzel$)
3 $E' \leftarrow \{(u,v) | v \in V \setminus \{wurzel\} \wedge u = vorgänger[v]\}$
4 **return** Knoten aufsteigend nach Entdeckungszeit in TIEFENSUCHE(V, E')

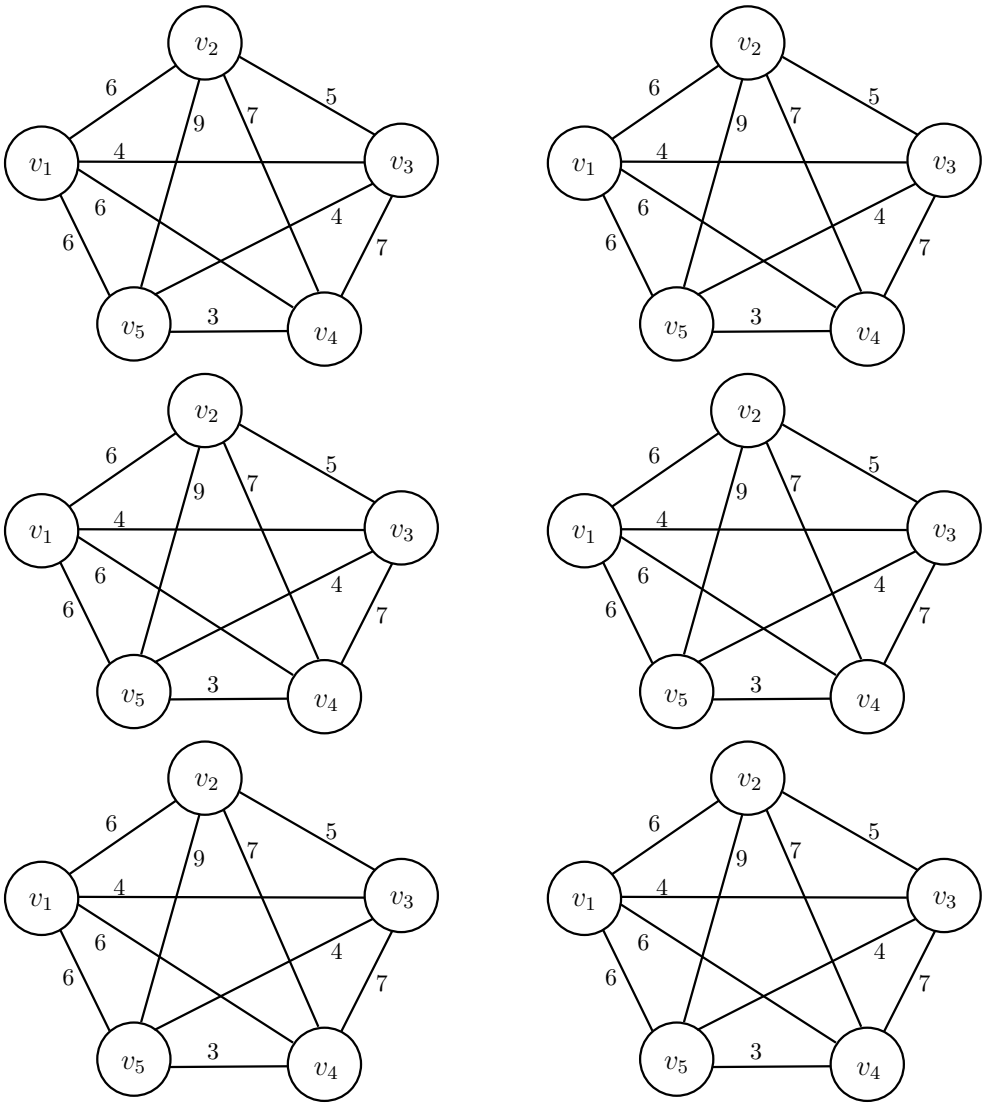
Zeitaufwand:





```
PRIM(Knoten V, Kanten E ⊂ V × V, Kosten γ : E → ℝ, Startknoten s ∈ V)
     Rückgabewert: Kanten des Baums vorgänger
 1   for alle Knoten u ∈ V
 2   do ⌜ abstand[u] ← ∞
 3        vorgänger[u] → NULL
 4      ⌞ istFertig[u] ← false
 5   abstand[s] ← 0
 6   while KNOTEN-VORHANDEN-DIJKSTRA()
 7   do ⌜ nächster ← NÄCHSTER-KNOTEN-DIJKSTRA()
 8        istFertig[nächster] ← true
 9        for alle  v ∈ V mit (nächster, v) ∈ E
10        do ⌜ if ¬istFertig[v] und abstand[v] > γ[nächster, v]
11             then ⌜ abstand[v] ← γ[nächster, v]
12     ⌞     ⌞          ⌞ vorgänger[v] → nächster
13   return vorgänger
```

Zeitaufwand:



Knoten									
v_1		v_2		v_3		v_4		v_5	
d	v.	d	v.	d	v.	d	v.	d	v.
0	–	∞	–	∞	–	∞	–	∞	–

Kapitel 6

Kleinster Schaden im Worst-Case

NASA Director: This could be the worst disaster
NASA's ever faced.
Gene Kranz: With all due respect, sir,
I believe this is gonna be our finest hour.
(Apollo 13, 1995)

In diesem Abschnitt befassen wir uns mit der Frage, wie wir den Schaden möglichst klein halten, falls der „Worst-Case" eintritt, d.h. die Daten sind so ungünstig angeordnet, dass die schlechtestmögliche Laufzeit vorkommt. Ein Beispiel sind Bäume als Datenstruktur, die zu einer linearen Liste degeneriert sind.

6.1 Suchen mit balancierten Bäumen

⟶ **[Buch, Abschnitt 6.2, S.148-165]**

Motivation: Ein Baum der Tiefe l besitzt maximal

$$2^0 + 2^1 + \ldots + 2^{l-1} = \sum_{i=0}^{l-1} 2^i = \frac{2^l - 1}{2 - 1} = 2^l - 1$$

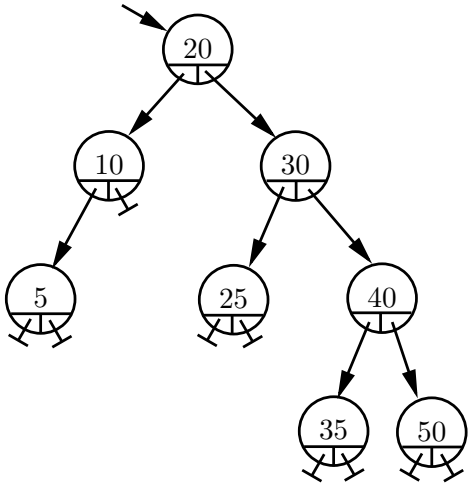
Knoten (Beweis: geometrische Reihe). Falls wir also die Zugriffszeiten (FIND etc.) minimal halten wollen, muss ein Baum mit n Knoten immer die Höhe $\mathcal{O}(\lceil \log_2 n \rceil)$ haben.

Dies kann erreicht werden durch sog. balancierte Bäume. Dabei sei die Balance eines Knotens das Verhältnis, in dem seine Unterbäume zueinander stehen. Ein gewichtsbalancierter Baum heißt balanciert, wenn für jeden Knoten eine „Balance-Aussage" zur Anzahl der Knoten in den beiden Unterbäumen erfüllt ist. Hier werden im Weiteren höhenbalancierte Bäume im Detail vorgestellt.

Def. AVL-Baum: Ein binärer Suchbaum heißt *AVL-Baum*, wenn für die Unterbäume jeden Knotens u im Baum die folgende Eigenschaft gilt:

$$\mathit{Baumtiefe}(u.\mathit{rechts}) - \mathit{Baumtiefe}(u.\mathit{links}) \in \{-1, 0, 1\}$$

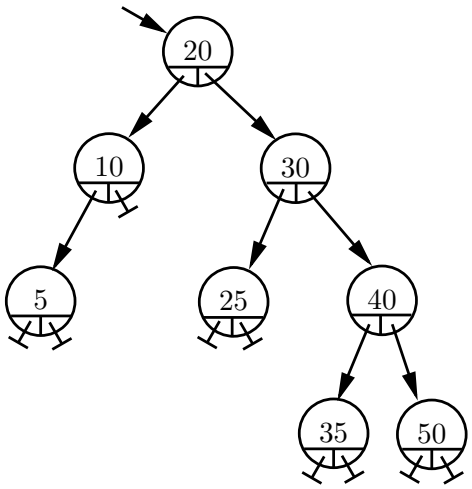
Bsp.: Wie sehen die Balancefaktoren aus? Was passiert, wenn Knoten 45 eingefügt wird?



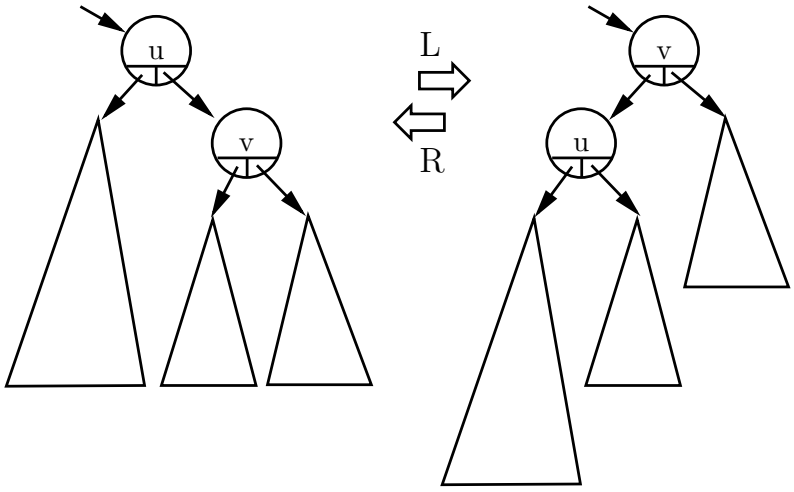
Einzelne Knoten werden in einem Baum „rotiert“, um die Balance wieder herzustellen.

ROTIERE-LINKS(Element $e\ell$)
Rückgabewert: neue Baumwurzel *hoch*
1 $hoch \rightarrow e\ell.rechts$
2 $e\ell.rechts \rightarrow hoch.links$
3 $hoch.links \rightarrow e\ell$
4 **return** *hoch*

ROTIERE-RECHTS(Element $e\ell$)
Rückgabewert: neue Baumwurzel *hoch*
1 $hoch \rightarrow e\ell.links$
2 $e\ell.links \rightarrow hoch.rechts$
3 $hoch.rechts \rightarrow e\ell$
4 **return** *hoch*



Frage: Ist ein rotierter Baum noch ein korrekter Suchbaum?




```
EINFÜGEN-AVL(Schlüssel neuerWert, Daten neueDaten)
    Rückgabewert: keine; Seiteneffekt: Baum geändert
1   anker →
    (tmp ←)   } EINFÜGEN-AVL-R(neuerWert, neueDaten, anker)

EINFÜGEN-AVL-R(Schlüssel neuerWert, Daten neueDaten, Element eℓ)
    Rückgabewert: neue Wurzel, Info ob Baumhöhe gleich; Seiteneffekt
 1  switch
 2  case eℓ = NULL : return  (allokiere Element(neuerWert, neueDaten,
                                                0, NULL, NULL), false )
 3  case neuerWert = eℓ.wert :
 4       error "Element schon enthalten"
 5  case neuerWert < eℓ.wert :
 6       eℓ.links →
          gleichHoch ←   } EINFÜGEN-AVL-R(neuerWert, neueDaten, eℓ.links)
 7       return PRÜFE-RECHTS-ROT(eℓ, gleichHoch)
 8  case neuerWert ≥ eℓ.wert :
 9       eℓ.rechts →
          gleichHoch ←   } EINFÜGEN-AVL-R(neuerWert, neueDaten, eℓ.rechts)
10       return PRÜFE-LINKS-ROT(eℓ, gleichHoch)
```

```
PRÜFE-RECHTS-ROT(Element eℓ, Info bzgl. der Unterbäume gleicheHöhe)
    Rückgabewert: neue Wurzel, Info ob Baumhöhe gleich; Seiteneffekt
 1  if gleicheHöhe
 2  then ⌈return (eℓ, true )
 3  else ⌈switch
 4           case eℓ.balance = 0 :
 5                eℓ.balance ← −1
 6                return (eℓ, false )
 7           case eℓ.balance = 1 :
 8                eℓ.balance ← 0
 9                return (eℓ, true )
10           case eℓ.balance = −1 :
11      ⌊         return REORGANISIERE-MIT-RECHTS-ROT(eℓ)
```

```
REORGANISIERE-MIT-RECHTS-ROT(Element eℓ)
    Rückgabewert: neuer Wurzelknoten, Info ob Baumhöhe gleich; Seiteneffekt
 1  switch
 2  case eℓ.links.balance = −1 :
 3       eℓ ← ROTIERE-RECHTS(eℓ)
 4       eℓ.rechts.balance ← 0
 5       eℓ.balance ← 0
 6       return (eℓ, true )
 7  case eℓ.links.balance = 0 :
 8       eℓ ← ROTIERE-RECHTS(eℓ)
 9       eℓ.rechts.balance ← −1
10       eℓ.balance ← 1
11       return (eℓ, false )
12  case eℓ.links.balance = 1 :
13       eℓ.links ← ROTIERE-LINKS(eℓ.links)
14       eℓ ← ROTIERE-RECHTS(eℓ)
15       BALANCE-NACH-DOPPELROTATION-ANPASSEN(eℓ)
16       eℓ.balance ← 0
17       return (eℓ, true )
```

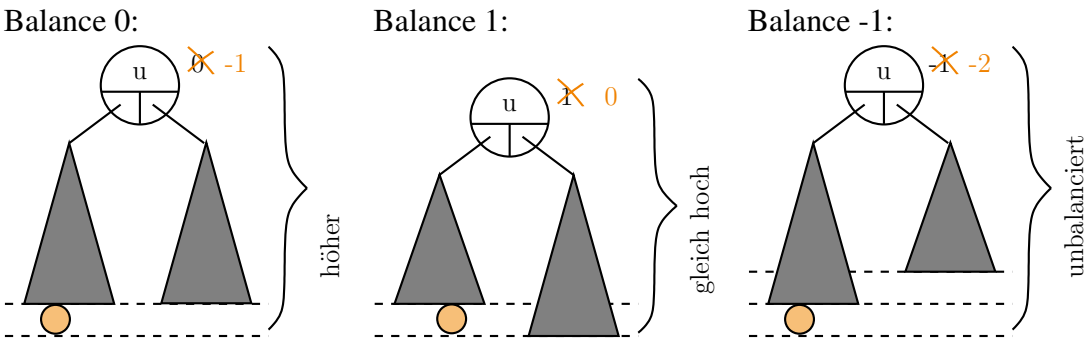
```
BALANCE-NACH-DOPPELROTATION-ANPASSEN(Element eℓ)
    Rückgabewert: keine; Seiteneffekt: Balance-Faktoren angepasst
 1  switch
 2  case eℓ.balance = −1 :
 3       eℓ.links.balance ← 0
 4       eℓ.rechts.balance ← 1
 5  case eℓ.balance = 1 :
 6       eℓ.links.balance ← −1
 7       eℓ.rechts.balance ← 0
 8  case eℓ.balance = 0 :
 9       eℓ.links.balance ← 0
10       eℓ.rechts.balance ← 0
```

Der Vollständigkeit halber: der symmetrische Fall – wird nicht in der Vorlesung behandelt

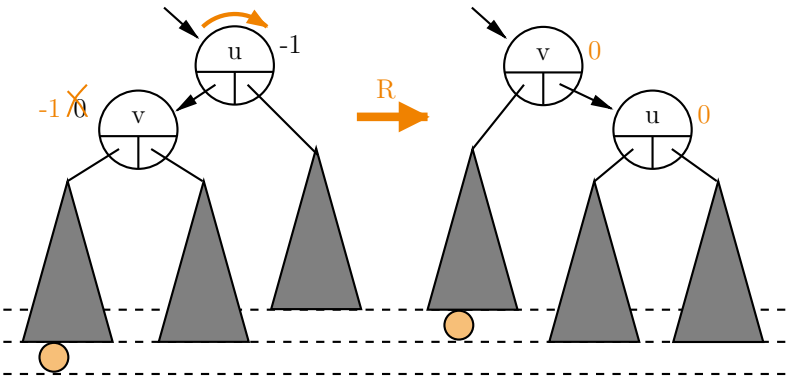
```
PRÜFE-LINKS-ROT(Element eℓ, Info bzgl. der Unterbäume gleicheHöhe)
    Rückgabewert: neue Wurzel, Info ob Baumhöhe gleich; Seiteneffekt
 1  if gleicheHöhe
 2  then ⌊ return (eℓ, true )
 3  else ⌜ switch
 4           case eℓ.balance = 0 :
 5                eℓ.balance ← 1
 6                return (eℓ, false )
 7           case eℓ.balance = −1 :
 8                eℓ.balance ← 0
 9                return (eℓ, true )
10           case eℓ.balance = 1 :
11        ⌞          return REORGANISIERE-MIT-LINKS-ROT(eℓ)
```

```
REORGANISIERE-MIT-LINKS-ROT(Element el)
      Rückgabewert: neuer Wurzelknoten, Info ob Baumhöhe gleich; Seiteneffekt
 1    switch
 2    case el.rechts.balance = 1 :
 3          el ← ROTIERE-LINKS(el)
 4          el.links.balance ← 0
 5          el.balance ← 0
 6          return (el, true )
 7    case el.rechts.balance = 0 :
 8          el ← ROTIERE-LINKS(el)
 9          el.links.balance ← 1
10          el.balance ← −1
11          return (el, false )
12    case el.rechts.balance = −1 :
13          el.rechts ← ROTIERE-RECHTS(el.rechts)
14          el ← ROTIERE-LINKS(el)
15          BALANCE-NACH-DOPPELROTATION-ANPASSEN(el)
16          el.balance ← 0
17          return (el, true )
```

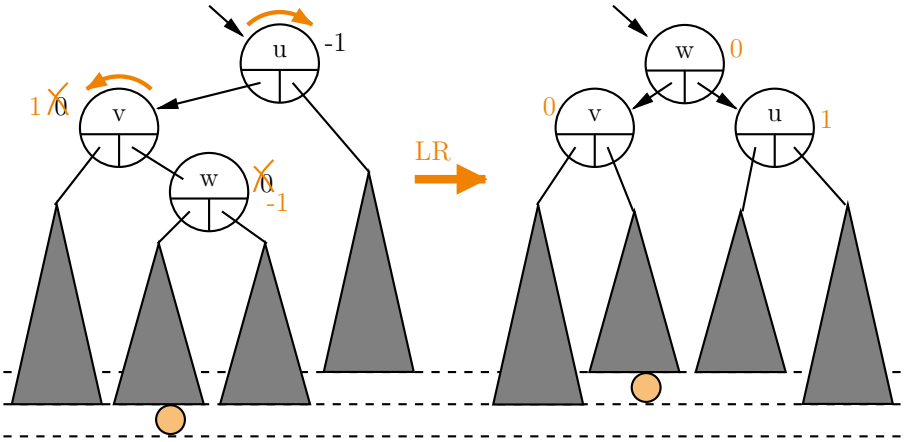
Fälle beim Einfügen im linken Unterbaum



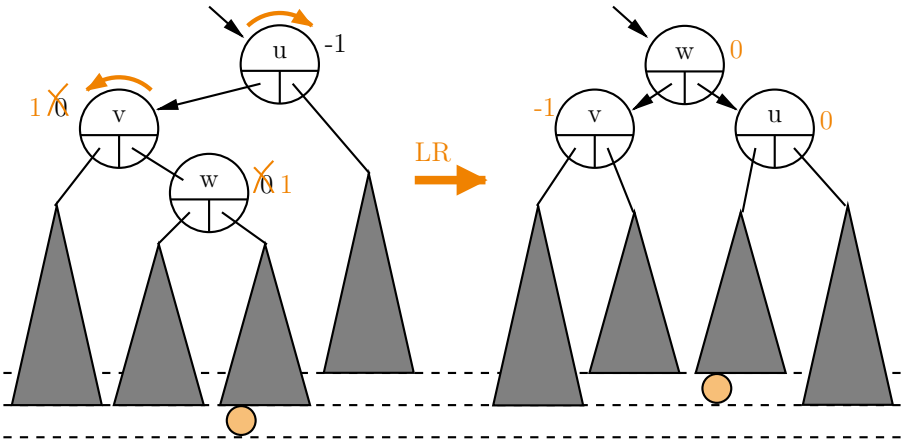
Einfügen im linken Unterbaum von *v*:



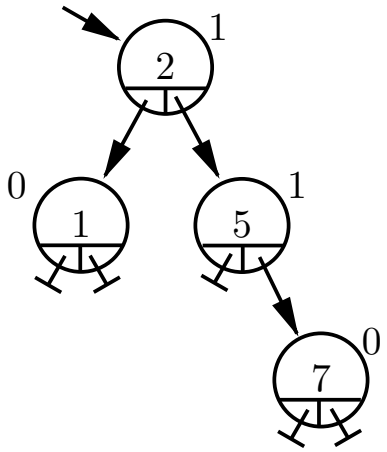
Einfügen im rechten Unterbaum von v (Fall a):



Einfügen im rechten Unterbaum von v (Fall b):



Beispiel: Knoten 8, 9, 4, 6 und 3 einfügen.

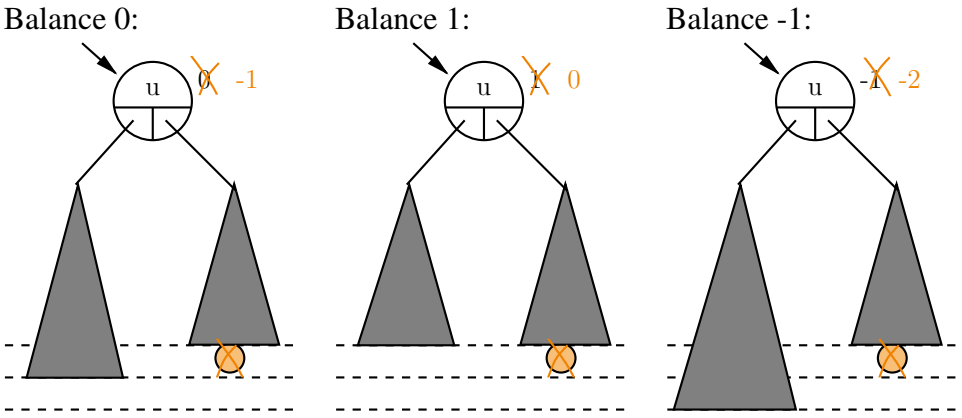


```
LÖSCHEN-AVL(Schlüssel löschWert)
    Rückgabewert: keine; Seiteneffekt: Baum geändert
1   anker →, (tmp ←) } LÖSCHEN-AVL-R(löschWert, anker)

LÖSCHEN-AVL-R(Schlüssel löschWert, Element eℓ)
    Rückgabewert: neue Wurzel, Info ob Baumhöhe gleich; Seiteneffekt
 1  switch
 2  case eℓ = NULL :
 3      error "Element nicht gefunden"
 4  case löschWert < eℓ.wert :
 5      eℓ.links →, gleichHoch ← } LÖSCHEN-AVL-R(löschWert, eℓ.links)
 6      return PRÜFE-LINKS-ROT(eℓ, gleichHoch)
 7  case löschWert > eℓ.wert :
 8      eℓ.rechts →, gleichHoch ← } LÖSCHEN-AVL-R(löschWert, eℓ.rechts)
 9      return PRÜFE-RECHTS-ROT(eℓ, gleichHoch)
10  case löschWert = eℓ.wert :
11      switch
12      case eℓ.links = NULL :  return (eℓ.rechts, false )
13      case eℓ.rechts = NULL :  return (eℓ.links, false )
14      case default :
15          ersatz →, gleichHoch ← } SUCHE-AVL-NACHFOLGER(eℓ)
16          ersatz.balance ← eℓ.balance
17          ersatz.links → eℓ.links
18          ersatz.rechts → eℓ.rechts
19          return PRÜFE-RECHTS-ROT(ersatz, gleichHoch)
```

```
SUCHE-AVL-NACHFOLGER(Element eℓ)
    Rückgabewert: Ersatzknoten, Info ob Baumhöhe gleich; Seiteneffekt
1   if eℓ.rechts.links = NULL
2   then ⌜ ersatz ← eℓ.rechts
3          eℓ.rechts ← eℓ.rechts.rechts
4        ⌞ return ersatz, false
5   else ⌜ ersatz →, eℓ.rechts →, gleichHoch ← } SUCHE-AVL-NACHFOLGER-R(eℓ.rechts)
6        ⌞ return ersatz, gleichHoch
```

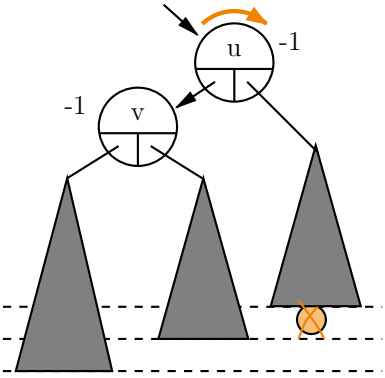
```
SUCHE-AVL-NACHFOLGER-R(Element el)
    Rückgabewert: Ersatzknoten, neue Wurzel, Info ob Baumhöhe gleich; Seiteneffekt
1   if el.links.links ≠ NULL
2   then ⌜ ersatz →
         |  el.links →                } SUCHE-AVL-NACHFOLGER-R(el.links)
         ⌞  gleichHoch ←
3        ⌞ return ersatz, PRÜFE-LINKS-ROT(el, gleichHoch)
4   else ⌜ ersatz ← el.links
5        |  el.links ← el.links.rechts
6        ⌞ return ersatz, PRÜFE-LINKS-ROT(el, false )
```

Fälle beim Löschen im rechten Unterbaum

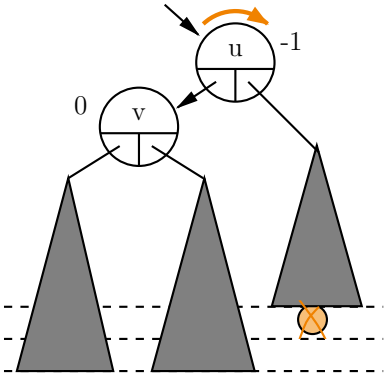


Balancieren beim Löschen im rechten Unterbaum

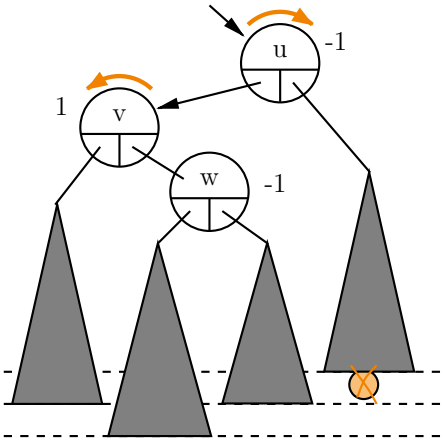
Fall a):



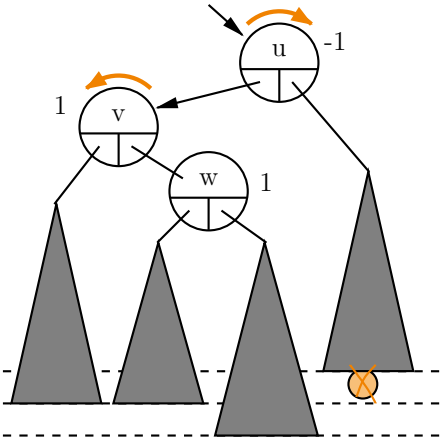
Fall b):



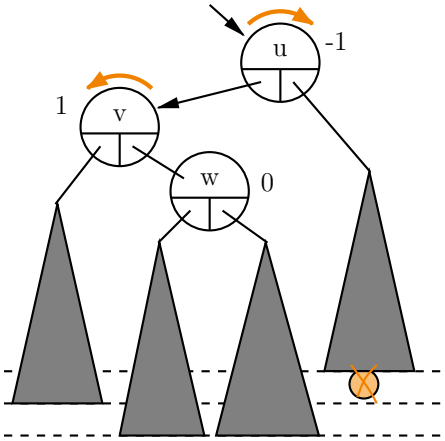
Fall c):



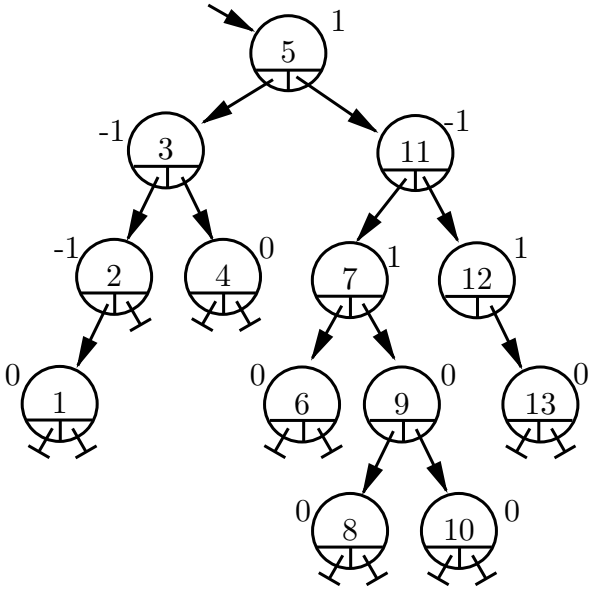
Fall d):



Fall e):

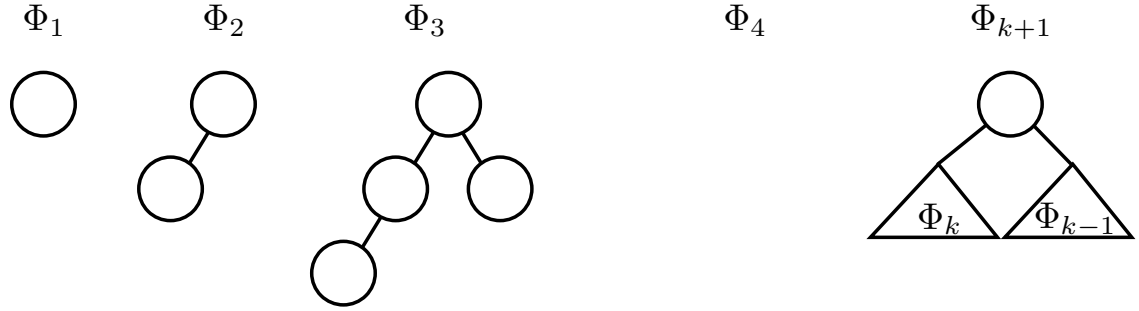


Beispiel: Knoten 5 und 4 löschen.



Beweis der logarithmischen Laufzeit für alle Operationen:

1. Wir konstruieren die dünnsten AVL-Bäume, die sog. Fibonacci-Bäume, bei denen das Löschen eines beliebigen Knotens die Höhe um 1 verringert.



2. **Beweis** dafür, dass es die dünnsten Bäume sind:

3. **Lemma:** Ein Baum Φ_i hat genau $F_{i+2}-1$ Knoten.

   Erinnerung: $F_1 = F_2 = 1$, $F_k = F_{k-1} + F_{k-2}$

4. **Beweis:** mit Induktion
   Induktionsanfang:

   - Φ_1 hat 1 Knoten; $F_3 - 1 = 2 - 1 = 1$
   - Φ_2 hat 2 Knoten; $F_4 - 1 = 3 - 1 = 2$

   Induktionsannahme: Die Behauptung gelte für alle Bäume $\Phi_1, \ldots, \Phi_k$
   Induktionsbehauptung: dann hat Φ_{k+1} genau $F_{k+3} - 1$ Knoten
   Induktionsschritt: Φ_{k+1} besteht aus den Bäumen Φ_k und Φ_{k-1} sowie einem weiteren Knoten – d.h. die Anzahl der Knoten ist

$$\begin{aligned} & (F_{k+2}-1) + (F_{k+1}-1) + 1 \\ = \; & (F_{k+2} + F_{k+1}) - 1 \\ = \; & F_{k+3} - 1 \text{q.e.d} \end{aligned}$$

5. **Satz:** Alle AVL-Bäume haben die Tiefe $\mathcal{O}(\log n)$.

6. **Beweis:** Aus der Mathematik ist bekannt, dass sich die Fibonacci-Zahlen wie folgt darstellen lassen:

$$F_k = \underbrace{\frac{1}{\sqrt{5}}}_{\substack{=c_1 \\ \approx 0.447213}} \cdot \left[\underbrace{\left(\frac{1+\sqrt{5}}{2}\right)^k}_{\substack{=c_2 \\ \approx 1.618035}} - \underbrace{\left(\frac{1-\sqrt{5}}{2}\right)^k}_{\substack{=c_3 \\ \approx -0.618035}} \right]$$

Damit muss für alle AVL-Bäume mit n Knoten und Tiefe d gelten:

$$
\begin{aligned}
n &\geq F_{d+2} - 1 \\
&= c_1 \cdot (c_2^{d+2} - \underbrace{c_3^{d+2}}_{<1}) - 1 \\
&\geq c_1 \cdot c_2^{d+2} - \underbrace{(c_1+1)}_{<2} \\
&\geq c_1 \cdot c_2^{d+2} - 2 \\
n+2 &\geq c_1 \cdot c_2^{d+2} \\
\log_{c_2}(n+2) &\geq \log_{c_2}(c_1) + (d+2) \\
d &\leq \log_{c_2}(n+2) - \log_{c_2}(c_1) - 2 \\
&= \frac{\log_2(n+2)}{\log_2 c_2} - \frac{\log_2(c_1)}{\log_2(c_2)} - 2 \\
&\leq 1.440418 \cdot \log_2(n+2) - \frac{\log_2(c_1)}{\log_2(c_2)} - 2 \\
&\quad \text{und wegen } \log(x+y) = \log x + \log(1+\frac{y}{x}) \\
&= 1.440418 \cdot \log_2 n + \underbrace{1.440418 \cdot \log_2(1+\frac{2}{n}) - \underbrace{\frac{\log_2(c_1)}{\log_2(c_2)}}_{=-1.6722765} - 2}_{<0 \text{ für } n \geq 12} \\
&\leq 1.440418 \cdot \log_2(n) \text{ für hinreichend großes } n
\end{aligned}
$$

6.2 Sortieren mit bestmöglicher asymptotischer Laufzeit

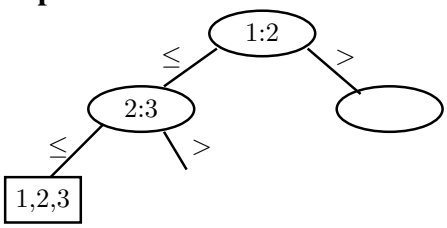
$\longrightarrow$ **[Buch, Abschnitt 6.3, S.165-169]**

Die folgenden Überlegungen basieren auf der Annahme, dass der betrachtete Sortieralgorithmus auf einer Reihe von Vergleichen je zweier Schlüssel/Werte basiert. Dann gilt:

- Jeder Sortieralgorithmus konstruiert eine Umordnung $\pi \in \mathscr{S}_n$ auf der Basis der vorgenommenen Vergleiche. Dabei enthält $\mathscr{S}_n$ alle möglichen Permutationen der Zahlen $1,\ldots,n$. Also gibt π an, welche Daten an welche Position gerückt werden müssen, damit die Folge sortiert ist.
- Für jede zu sortierende Folge realisiert ein Sortieralgorithmus genau eine Permutation.

Wir betrachten im Weiteren einen Entscheidungsbaum, bei dem jeder Knoten einem Vergleich entspricht und die Kanten die Abfolge der Vergleiche definiert.

Bsp.: Feld der Größe 3 ist zu sortieren.



Satz:

Wie kann mit dem bisherigen Wissen ein immer optimales Sortierverfahren realisiert werden?

Bsp.:

20	54	28	31	5	24	39	14	1	15

Was ist der Nachteil?

Kapitel 7

Teile und Beherrsche

Cleopatra: Together we could conquer the world.
Caesar: Nice of you to include me.
(Cleopatra, 1934)

„Divide-and-Conquer“ bzw. „Teile-und-Beherrsche“ ist ein allgemeingültiger Ansatz zum Algorithmenentwurf. Seine Grundidee ist die Zerlegung eines Problems in kleinere gleichgeartete Probleme, die dann rekursiv gelöst werden, bis die betrachteten Probleme klein genug sind.

Dabei sind immer drei Schritte notwendig:

- *Teile:* Ein Problem wird in mehrere kleinere Probleme zerlegt.
- *Beherrsche:* Die Teilprobleme werden rekursiv gelöst – sind sie klein genug, werden sie direkt gelöst.
- *Verbinde:* Die Lösungen der Teilprobleme werden zur Lösung für das größere Problem zusammen gesetzt.

7.1 Sortieren durch Mischen

⟶ **[Buch, Abschnitt 7.1, S.175-180]**

```
MERGESORT(Feld A, Bereichsgrenzen links, rechts)
    Rückgabewert: nichts; Seiteneffekt: A ist sortiert
1   if rechts > links
2   then ⌜ mitte ← ⌊(links+rechts)/2⌋
3          MERGESORT(A, links, mitte)
4          MERGESORT(A, mitte + 1, rechts)
5        ⌞ MISCHEN(A, links, mitte, rechts)
```

```
MISCHEN(Feld A, Bereichsgrenzen links, mitte, rechts)
     Rückgabewert: nichts; Seiteneffekt: A ist sortiert
 1   for k ← links, ..., mitte
 2   do ⌊ B[k] ← A[k]
 3   for k ← mitte + 1, ..., rechts
 4   do ⌊ B[rechts + mitte + 1 − k] ← A[k]
 5   i ← links
 6   j ← rechts
 7   k ← links
 8   while i < j
 9   do ⌈ if B[i] ≤ B[j]
10         then ⌈ A[k] ← B[i]
11              ⌊ i ← i + 1
12         else ⌈ A[k] ← B[j]
13              ⌊ j ← j − 1
14      ⌊ k ← k + 1
15   A[rechts] ← B[i]
```

Beispiel:

A[1]	A[2]								A[10]
20	54	28	31	5	24	39	14	1	15

Nachteil des Verfahrens?

7.2 Laufzeitanalyse bei Divide-and-Conquer

⟶ **[Buch, Abschnitt 7.2, S.180-185]**

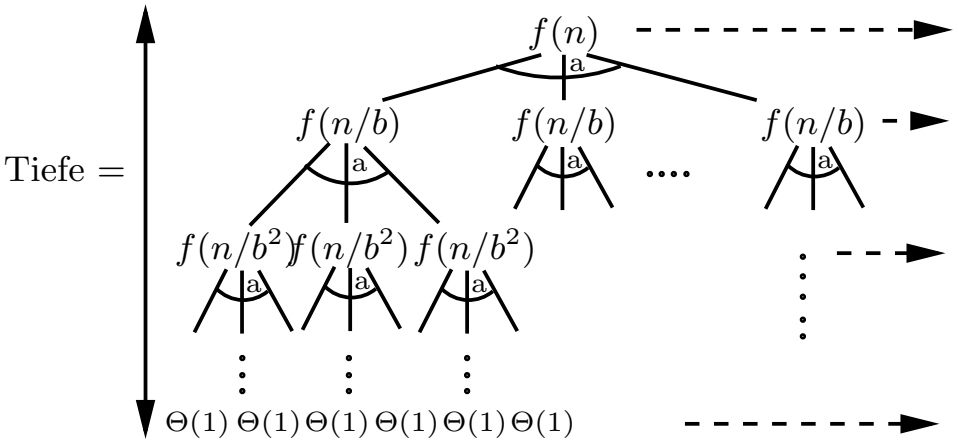
Master-Theorem: Sei die Laufzeit $T(n)$ durch folgende Rekursionsgleichung beschrieben

$$T(n) = \begin{cases} c, & \text{falls } n \text{ hinreichend klein} \\ a \cdot T(\lceil \frac{n}{b} \rceil) + f(n), & \text{sonst} \end{cases},$$

wobei $a \geq 1$, $b > 1$, $c \in \mathbb{N}$ und $f(n) \in \Theta(n^k)$. Dann gilt mit $\gamma = \log_b a$:

$$T(n) \in \begin{cases} \Theta(n^k), & \text{falls } \gamma < k \\ \Theta(n^k \cdot \log n) & \text{falls } \gamma = k \\ \Theta(n^\gamma) & \text{falls } \gamma > k \end{cases}$$

Beweis: Welche Laufzeit fällt insgesamt an?



Also gilt: $T(n) =$ ☐

Nun werden die asymptotischen Grenzen der drei Fälle in obige Gleichung eingesetzt.

Beispiele zur Anwendung des Mastertheorems:

- $T(n) = \begin{cases} 15, & \text{falls } n < 2 \\ 2 \cdot T(\lceil \frac{n}{2} \rceil) + 15 \cdot n^2, & \text{sonst} \end{cases}$. Es gilt:

  $a =$ ☐, $b =$ ☐, $f(n) =$ ☐ und $n^{\log_b a} =$ ☐. Und damit gilt:

- $T(n) = \begin{cases} 1000, & \text{falls } n < 5 \\ 4 \cdot T(\lceil \frac{n}{2} \rceil) + 2000 \cdot n, & \text{sonst} \end{cases}$. Es gilt:
  $a =$ [ ], $b =$ [ ], $f(n) =$ [ ] und $n^{\log_b a} =$ [ ]. Und damit gilt:

- $T(n) = \begin{cases} 1, & \text{falls } n < 2 \\ 2 \cdot T(\lceil \frac{n}{2} \rceil) + n, & \text{sonst} \end{cases}$. Es gilt:
  $a =$ [ ], $b =$ [ ], $f(n) =$ [ ] und $n^{\log_b a} =$ [ ]. Und damit gilt:

Was folgt nun für die Laufzeit von Mergesort?

7.3 Quicksort

⟶ **[Buch, Abschnitt 7.3, S.187-193]**

Idee: Im Teile-Schritt wird eine grobe Vorsortierung vorgenommen, so dass bereits im Beherrsche-Schritt alle Elemente an ihre richtige Stelle sortiert werden. Dadurch entfällt der Verbinde-Schritt!

```
QUICKSORT(Feld A, Bereichsgrenzen links, rechts)
    Rückgabewert: nichts; Seiteneffekt: A ist sortiert
1   if links < rechts
2   then ⌜ p ←
         ⌞ q ←  } PARTITIONIERE(A, links, rechts)
3        QUICKSORT(A, links, p)
4        ⌞ QUICKSORT(A, q, rechts)
```

```
PARTITIONIERE(Feld A, Bereichsgrenzen links, rechts)
     Rückgabewert: Partitionsgrenzen; Seiteneffekt: A ist verändert
 1   i ← links
 2   j ← rechts
 3   p ← A[rechts]
 4   while i ≤ j
 5   do ⌜ while A[i] < p
 6        do ⌊ i ← i + 1
 7        while A[j] > p
 8        do ⌊ j ← j − 1
 9        if i ≤ j
10        then ⌜ VERTAUSCHE(A, i, j)
11                  i ← i + 1
12     ⌞          ⌞ j ← j − 1
13   return j, i
```

Beispiel:

A[1]	A[2]								A[10]
20	54	28	31	5	24	39	14	1	15

Zeitaufwand bei günstiger Zerlegung:

Zeitaufwand bei ungünstiger Zerlegung:

Verbesserungen: Bisher wurde immer das rechte Element als Pivot-Element zur Partitionierung des Feldes gewählt. Wann hat dies einen nachteiligen Effekt? Wie kann das Verfahren verbessert werden? Ändert sich dadurch das Worst-Case-Verhalten?

Kapitel 8

Dynamisches Programmieren

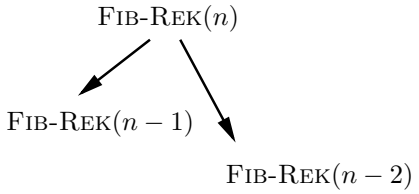
Fred C. Dobbs: I think I'll go to sleep
and dream about piles of gold
getting bigger and bigger and bigger.
(The Treasure of the Sierra Madre, 1948)

Motivation: $\longrightarrow$ **[Buch, Abschnitt 8.1, S.205-207]** Beim Prinzip „Divide-and-Conquer" war die Grundidee, ein großes Problem in gleichgeartete kleinere Teilprobleme zu zerlegen und diese dann rekursiv zu bearbeiten.

Betrachten wir die Definition der Fibonacci-Zahlen: $F_n = F_{n-1} + F_{n-2}$ mit $F_1 = F_2 = 1$. Mit Divide-and-Conquer erhalten wir den folgenden Algorithmus:

```
FIB-REK(positive Zahl n)
    Rückgabewert: zugehörige Fibonacci-Zahl
1   if n < 3
2   then [ return 1
3   else [ return FIB-REK(n − 1) + FIB-REK(n − 2)
```

Betrachten wir den Baum der Funktionsaufrufe:



Aus der Kenntnis, dass $F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$ gilt, folgt damit eine Laufzeit von [ ].

Idee: Anstatt oben in der Rekursion alle Werte mehrfach zu berechnen, werden sie nur einmal berechnet und zwischengespeichert. Das ist die Idee des *Dynamischen Programmierens*. Das ergibt dann den folgenden Algorithmus:

```
FIB-ITER(positive Zahl n)
    Rückgabewert: zugehörige Fibonacci-Zahl
1   A[1] ← 1
2   A[2] ← 1
3   k ← 3
4   while k ≤ n
5   do ⌜A[k] ← A[k−1] + A[k−2]
6      ⌞k ← k + 1
7   return A[n]
```

Die Laufzeit beträgt: [ ]

8.1 Alle kürzesten Wege nach Floyd-Warshall

⟶ **[Buch, Abschnitt 8.2, S.207-211]**

Die Aufgabe ist es, in einem Graphen die kürzesten Wege zwischen allen möglichen Knotenpaaren zu berechnen.

Wie groß ist der Aufwand, wenn wir dies durch Mehrfachanwendung des Dijkstra-Algorithmus berechnen?

Hier wollen wir nun einen alternativen Algorithmus kennenlernen, der direkt auf einer Adjazenzmatrix arbeitet, bei der statt des Wertes 1 das Kantengewicht abgespeichert ist und statt des Wertes 0 die Zahl ∞. Dann beruht der Algorithmus auf der folgenden Idee:

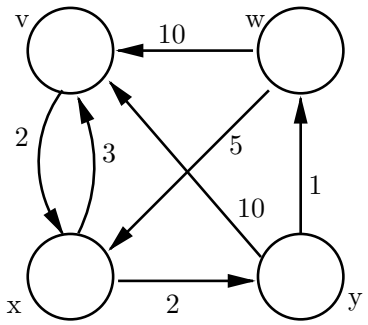
- Angenommen ich kenne die kürzesten Wege $u \to_T v$ für alle Knotenpaare $u, v \in V$, wobei ich nur die Knoten aus einer Menge $T \subset V$ als Zwischenknoten benutzen darf.
- Dann gilt für einen anderen Knoten $w \in V \setminus T$ mit $w \neq u$ und $w \neq v$, dass der Weg $u \to_T w \to_T v$ der kürzeste Weg von u nach v über den Knoten w sein muss.
- D.h. wir bekommen die kürzesten Wege für alle Knotenpaare, bei der nur Zwischenknoten aus $T' = T \cup \{w\}$ benutzt werden dürfen, in dem wir obigen Weg über w nur aufnehmen, wenn er kürzer als der bereits bekannte Weg über die Knoten in T ist.

Wenn man dieses Prinzip rekursiv implementieren würde, ergäbe sich eine exponentiale Laufzeit. Erweitert man jedoch T iterativ und speichert die Zwischenergebnisse, bekommt man folgenden Algorithmus.

```
FLOYD-WARSHALL(Knoten V, Kanten E ⊂ V × V, Kosten γ : E → ℝ)
    Rückgabewert: Distanzmatrix, Matrix der Zwischenknoten
 1  dist → allokiere Feld der Größe #V × #V
 2  über → allokiere Feld der Größe #V × #V
 3  for i ← 1,...,#V
 4  do ⌜ for j ← 1,...,#V
 5       do ⌜ über[i, j] ← ⊥
 6            if i = j
 7            then ⌊ dist[i, j] ← 0
 8            else ⌜ if (i, j) ∈ E
 9                   then ⌊ dist ← γ(i, j)
10    ⌟  ⌟       ⌟ else ⌊ dist ← ∞
11  for k ← 1,...,#V
12  do ⌜ for i ← 1,...,#V
13       do ⌜ for j ← 1,...,#V
14            do ⌜ if dist[i,k] + dist[k, j] < dist[i, j]
15                 then ⌜ dist[i, j] ← dist[i,k] + dist[k, j]
16    ⌟  ⌟  ⌟       ⌟ über[i, j] ← k
17  return dist, über
```

Die Laufzeit beträgt: $\square$

```
AUSGABE-FLOYD-WARSHALL(Start u, Ziel v, Matrix der Zwischenknoten über)
    Rückgabewert: Sequenz der Zwischenknoten auf dem Weg
1   if über[u, v] = ⊥
2   then ⌊ return ε
3   else ⌜ ersteEtappe ← AUSGABE-FLOYD-WARSHALL(u, über[u, v], über)
4          zweiteEtappe ← AUSGABE-FLOYD-WARSHALL(über[u, v], v, über)
5        ⌟ return ersteEtappe ∘ über[u, v] ∘ zweiteEtappe
```



Ausgangsmatrix

	v	w	x	y
v	0	∞	2	∞
w	10	∞	5	∞
x	3	∞	0	2
y	10	1	∞	0

über v

über w

über x

über y

8.2 Heuristik: bitonische Rundreise

⟶ **[Buch, Aufgabe 8.6, S.224]**

Definition: Für ein symmetrisches Rundreiseproblem im 2D (*x*/*y*) mit den Knoten $V = \{0,\ldots,n-1\}$ und eindeutigen Werten x_i ($0 \leq i < n$) heißt eine Rundreise $pi : \{0,\ldots,n-1\} \to \{0,\ldots,n-1\}$ *bitonisch*, falls es einen Index $k \in \{1,\ldots,n-1\}$ gibt, für den gilt:

- $x_{\pi(0)} < \ldots < x_{\pi(k)}$ und
- $x_{\pi(k)} > \ldots > x_{\pi(n-1)} > x_{\pi(0)}$.

Idee: Konstruktion einer minimalen bitonischen Rundreise

- jeder Knoten: oberer oder unterer Route zuordnen
- $len(i,j)$ ist Länge des kürzesten (umgekehrt bitonischen) Wegs von i nach $j(>i)$ mit den Knoten $\{0,\ldots,j\}$

$$len(i,j) = \begin{cases} \gamma(i,j), & \text{falls } i = 0 \text{ und } j = 1 \\ len(i,j-1) + \gamma(j-1,j), & \text{falls } i < j-1 \\ \min_{0\leq k<i}(len(k,i) + \gamma(k,j)), & \text{sonst} \end{cases}$$

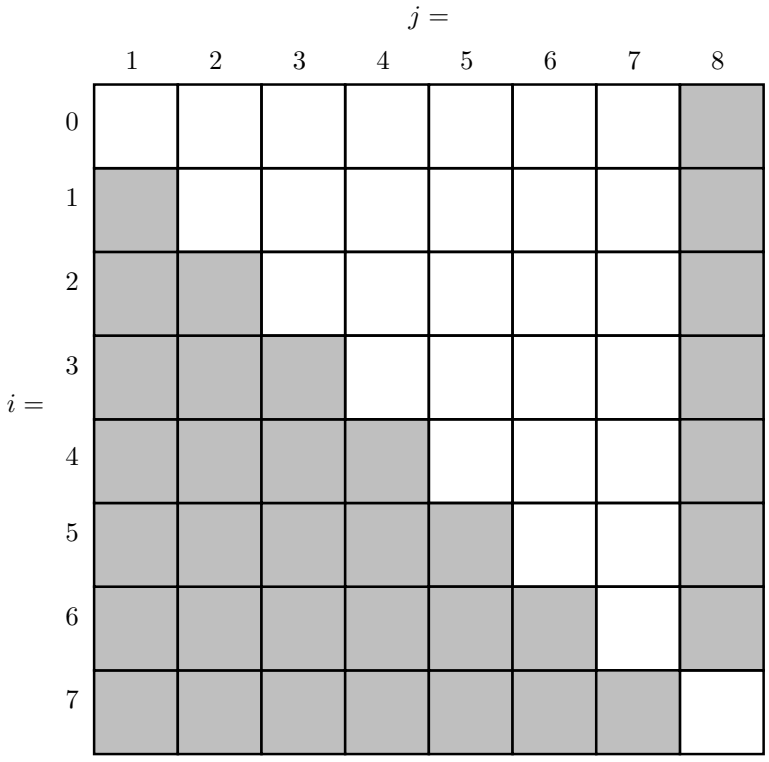
```
BITONISCHE-RUNDREISE(Knoten V = {0,1,...,n-1} (mit aufsteigendem x-Wert), Kosten γ : V × V → ℝ)
    Rückgabewert: Rundreise, Kosten
 1  len → allokiere Feld mit Indizes {0,...,n-1} × {1,...,n}
 2  rout → allokiere Feld der Größe n
 3  for i ← 0,...,n-2
 4  do ⌜ if i = 0
 5        then ⌊ len[0,1] ← γ(0,1)
 6        else ⌜ len[i,i+1] ← min_{0≤k<i}(len(k,i) + γ(k,i+1))
 7             ⌞ rout[i] ← k (Index mit minimalem Wert)
 8        for j ← i+2,...,n-1
 9     ⌞  do ⌊ len[i,j] ← len[i,j-1] + γ(j-1,j)
10  len[n-1,n] ← min_{0≤k<n-1}(len(k,n-1) + γ(k,n-1))
11  rout[n-1] ← k (Index mit minimalem Wert)
12  return AUSGABE-BITONISCHE-RUNDREISE(rout), len[n-1,n]
```

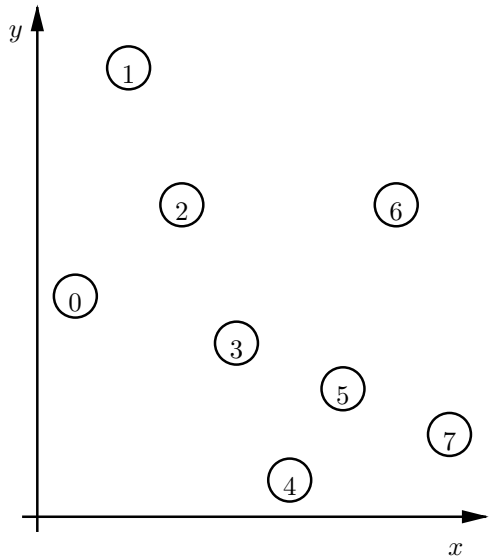
```
AUSGABE-BITONISCHE-RUNDREISE(Routeninformation rout)
    Rückgabewert: Rundreise
1   rundreise ← ⟨⟩
2   i ← n − 1
3   for j ← n − 1,...,0
4   do ⌜ if i = j
5         then ⌜ rundreise ← INVERTIERE(rundreise) d.h. umdrehen
6              ⌞ i ← rout[j]
7      ⌞ rundreise ← rundreise ∘ ⟨j⟩
8   return rundreise
```

Beispiel: Gegeben Sei ein Rundreiseproblem mit $P = \{0,\ldots,7\}$:

Entfernungstabelle

	1	2	3	4	5	6	7
0	5	2	3	5	5	6	8
1	0	3	6	10	8	6	10
2		0	3	6	5	4	7
3			0	3	2	4	5
4				0	2	6	3
5					0	4	2
6						0	4





8.3 Straight-Mergesort

⟶ **[Buch, Abschnitt 8.4, S.220-222]**

```
STRAIGHT-MERGESORT(Feld A)
     Rückgabewert: nichts; Seiteneffekt: A ist sortiert
 1   sortlänge ← 1
 2   n ← A.länge
 3   while sortlänge < n
 4   do ⌜ rechts ← 0
 5        while rechts + sortlänge < n
 6        do ⌜ links ← rechts + 1
 7             mitte ← links + sortlänge − 1
 8             if mitte + sortlänge ≤ n
 9             then ⌊ rechts ← mitte + sortlänge
10             else ⌊ rechts ← n
11           ⌞ MISCHEN(A, links, mitte, rechts)
12      ⌞ sortlänge ← 2 · sortlänge
```

Beispiel:

A[1]	A[2]								A[10]
20	54	28	31	5	24	39	14	1	15

Kapitel 9 Direkter Zugriff

Rick: And remember, this gun is pointed right at your heart.
Captain Renault: That is my least vulnerable spot.
(Casablanca, 1942)

9.1 Interpolationssuche

⟶ **[Buch, Abschnitt 9.1, S.227-229]**

Idee: Vorausgesetzt in einem sortierten Feld sind die Schlüssel gleichverteilt, dann kann relativ exakt die Position vorhergesagt werden, an der sich das gesuchte Element befindet. Wir modifizieren die binäre Suche durch diese Vorhersage.

```
INTERPOLATIONS-SUCHE(Schlüssel gesucht)
    Rückgabewert: gesuchte Daten bzw. Fehler falls nicht enthalten
 1  links ← 1
 2  rechts ← belegteFelder
 3  while links ≤ rechts
 4  do ⌜ schätzung ← ⌊ links + (gesucht − A[links].wert) / (A[rechts].wert − A[links].wert) (rechts − links) + 0.5 ⌋
 5        if schätzung < links oder schätzung > rechts
 6        then ⌊ error "Element nicht gefunden"
 7        else ⌜ switch
 8                 case A[schätzung].wert = gesucht :
 9                      return A[schätzung].daten
10                 case A[schätzung].wert > gesucht :
11                      rechts ← schätzung − 1
12                 case A[schätzung].wert < gesucht :
13    ⌞           ⌞         links ← schätzung + 1
14  error "Element nicht gefunden"
```

Beispiel: Suchen(24) im Feld

A[1]	A[2]								A[10]
1	5	14	15	20	24	28	31	39	54

Bemerkung: Unter der Voraussetzung, dass die Schlüssel gleichverteilt sind, gilt für FIND: $\mathcal{O}(\log\log n)$ – dies ist praktisch immer kleiner als 10!

9.2 Sortieren durch Abzählen

⟶ **[Buch, Abschnitt 9.2, S.229-232]**

Voraussetzung: Es gibt nur eine endliche Anzahl k an Schlüsseln. Diese können auch mehrfach vorkommen.

```
COUNTINGSORT(Feld A, größter Wert k)
     Rückgabewert: nichts; Seiteneffekt: A sortiert
 1   for i = 1,...,k
 2   do ⌊C[i] ← 0
 3   for j = 1,...,A.länge
 4   do ⌊C[A[j]] ← C[A[j]] + 1
 5   for i = 2,...,k
 6   do ⌊C[i] ← C[i] + C[i − 1]
 7   for j = A.länge,...,1
 8   do ⌈B[C[A[j]]] ← A[j]
 9      ⌊C[A[j]] ← C[A[j]] − 1
10   for j = 1,...,A.länge
11   do ⌊A[j] ← B[j]
```

Beispiel:

A[1]	A[2]						A[8]
2	5	3	1	2	3	1	3

$k = 5$

Feld C:

Feld B:

Zeitaufwand:

9.3 Radix-Sort

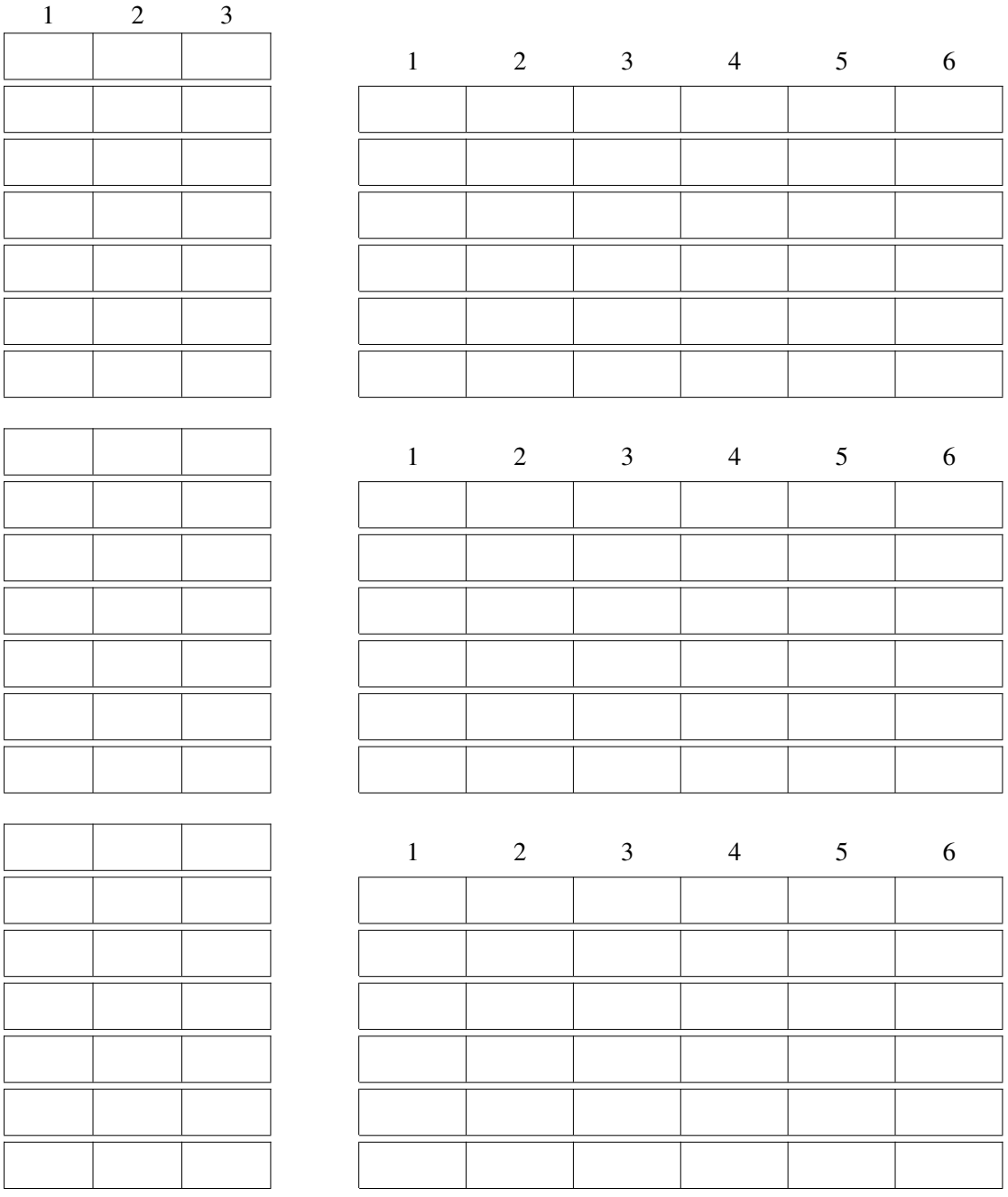
⟶ **[Buch, Abschnitt 9.3, S.232-234]**

Der folgende Algorithmus nutzt die Annahme, dass jede der Zahlen in dem Feld A mit genau d Stellen im Zehnersystem dargestellt wird. Dabei sei die Stelle 0 diejenige mit der kleinsten Wertigkeit und $d-1$ diejenige mit der größten Wertigkeit.

RADIX-SORT(Feld A, Stelligkeit der Werte d)
Rückgabewert: nichts; Seiteneffekt: A sortiert
1 **for** $i \leftarrow 0, \ldots, d-1$
2 **do** ⌊ sortiere A mit COUNTINGSORT und Stelle i als Schlüssel

Beispiel: Radixsort auf dem folgenden Feld mit Zahlen über der Ziffernmenge $\{1,2,3\}$

313	322	113	223	213	132



9.4 Mengenproblem: Hash-Tabellen

⟶ **[Buch, Abschnitt 9.4, S.235-252]**

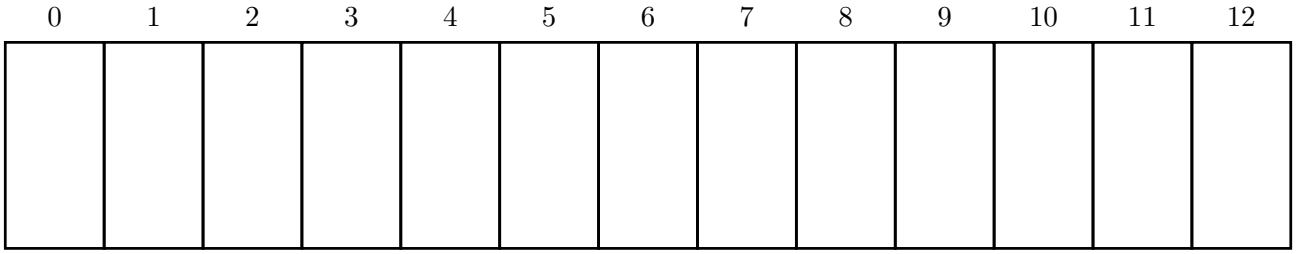
a	A	B	C	D	E	F	G	H	I	J	K	L	M
ϕ(a)	0	1	2	3	4	5	6	7	8	9	10	11	12

a	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
ϕ(a)	13	14	15	16	17	18	19	20	21	22	23	24	25

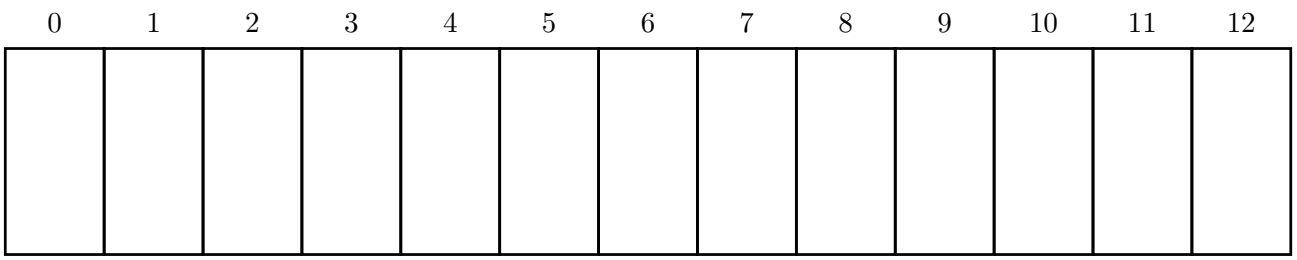
a_i	26^6	26^5	26^4	26^3	26^2	26^1	26^0	$code(a_i)$	$code(a_i)$ mod 13	$1+code(a_i)$ mod 10
Merkur		12	4	17	10	20	17	144710505	4	6
Venus			21	4	13	20	18	9676126	5	7
Erde				4	17	3	4	81878	4	9
Mond				12	14	13	3	220717	3	8
Mars				12	0	17	18	211372	5	3
Jupiter	9	20	15	8	19	4	17	3024877717	4	8
Saturn		18	0	19	20	17	13	214212687	0	8
Uranus		20	17	0	13	20	18	245405438	5	9
Neptun		13	4	15	19	20	13	156562809	0	10

HASH-CODE(Feld mit natürlichen Zahlen A)
Rückgabewert: Hash-Code
1 $code \leftarrow 1$
2 **for** $i \leftarrow 1,\ldots,A.länge$
3 **do** $\lfloor code \leftarrow 31 \cdot code + A.[i]$
4 **return** *code*

Als Hash-Funktion wird meist die folgende Möglichkeit gewählt:
Modulo-Methode: $h(c) = c \bmod p$



Externe Kollisionsauflösung: Kollidierende Schlüssel werden in einer verketteten Liste angehängt.



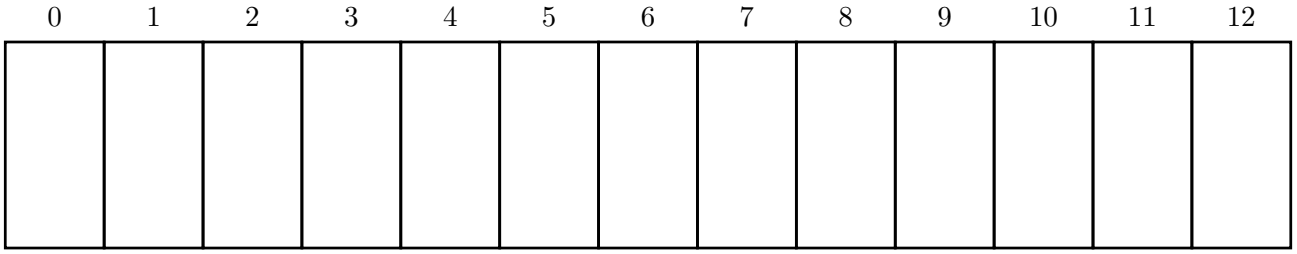
Nachteil?

Geschlossenes Hashing (bzw. *offene Addressierung*): Es soll ausschließlich der Platz in der Tabelle benutzt werden. Idee: Eine weitere Funktion bietet bei einer Kollision einen Alternativplatz an. Bei der j-ten Kollision gilt also die Hashfunktion

$$\tilde{h}(c, j) = h(h(c) + sond(c, j))$$

Lineares Sondieren:

$$sond(c, j) = j$$



SUCHEN-HASHTABELLE(Schlüssel *gesucht*)
Rückgabewert: gesuchte Daten bzw. Fehler falls nicht enthalten

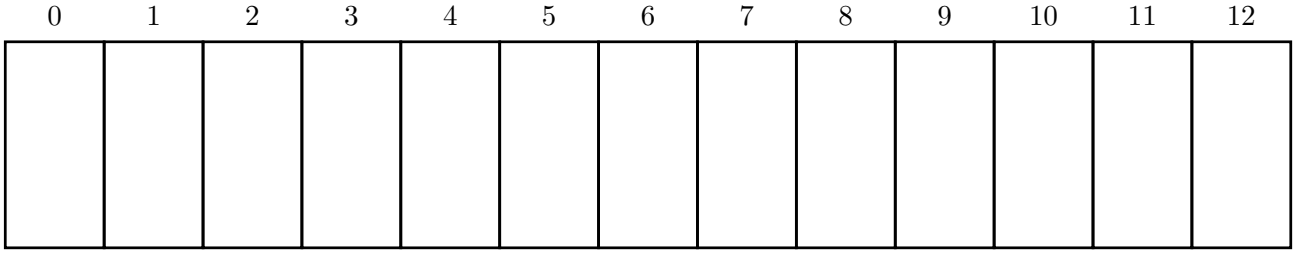
```
1  j ← 0
2  repeat ⌜ i ← h̃(gesucht, j)
3         ⌞ j ← j + 1
4  until (ht[i].wert = gesucht) oder (ht[i].status = FREI)
5  if ht[i].status = BELEGT
6  then ⌈ return ht[i].daten
7  else ⌈ error "Schlüssel nicht vorhanden"
```

```
LÖSCHEN-HASHTABELLE(Schlüssel löschWert)
    Rückgabewert: nichts falls erfolgreich bzw. Fehler sonst
1   j ← 0
2   repeat ⌜ i ← h̃(löschWert, j)
3          ⌞ j ← j + 1
4   until (ht[i].wert = löschWert) oder (ht[i].status = FREI)
5   if ht[i].status = BELEGT
6   then ⌊ ht[i].status ← ENTFERNT
7   else ⌊ error "Schlüssel nicht vorhanden"
```

```
EINFÜGEN-HASHTABELLE(Schlüssel neuerWert, Daten neueDaten)
     Rückgabewert: nichts; Seiteneffekte
 1   j ← 0
 2   i ← h̃(neuerWert, j)
 3   index ← i
 4   while ht[i].wert ≠ neuerWert und ht[i].status ≠ FREI
 5   do ⌜ j ← j + 1
 6      ⌟ i ← h̃(neuerWert, j)
 7      if ht[i].status ≠ BELEGT und ht[index].status = BELEGT
 8      then ⌊ index ← i
 9   if ht[i].status = BELEGT
10   then ⌊ error "Element ist bereits enthalten"
11   else ⌜ if ht[index].status = FREI und anzahl > 0.9 · p
12          then ⌜ ht ← VERGRÖSSERN-UND-REHASH(ht)
13               ⌟ EINFÜGEN-HASHTABELLE(neuerWert, neueDaten)
14          else ⌜ if ht[index].status = FREI
15               then ⌊ anzahl ← anzahl + 1
16                    ht[index].wert ← neuerWert
17                    ht[index].daten ← neueDaten
18          ⌟        ⌟ ht[index].status ← BELEGT
```

```
VERGRÖSSERN-UND-REHASH()
    Rückgabewert: nichts; Seiteneffekte
1   p_alt ← p
2   p ← ⌈3/2 · p⌉
3   ht_alt → ht
4   ht → allokiere Feld der Länge p
5   anzahl ← 0
6   for i ← 0, ..., p_alt − 1
7   do ⌜ if ht[i].status = BELEGT
8      ⌟ then ⌊ EINFÜGEN-HASHTABELLE(ht[i].wert, ht[i].daten)
```

Quadratisches Sondieren: $sond(c, j) = j^2$, Beispiel:

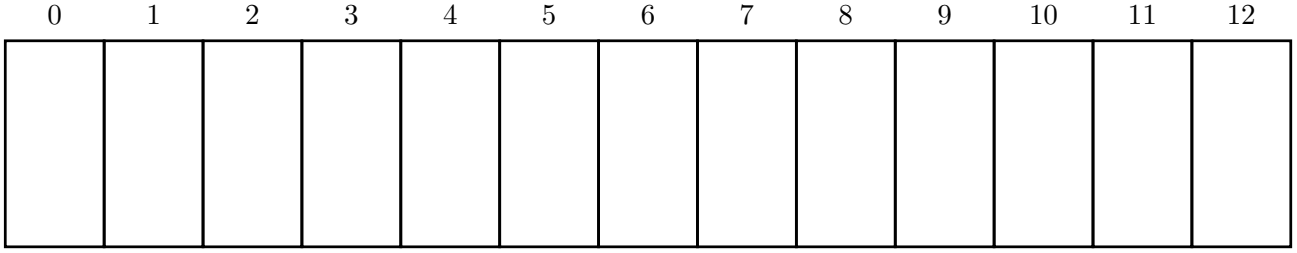


Bemerkung: Für eine effiziente Implementation überprüfen Sie die Abstände zwischen 1^2, 2^2, 3^2, 4^2 etc.

Wo treten jetzt noch lange Sondierungsketten auf?

Doppel-Hashing: Eine zweite Hash-Funktion $h' : B \to \{1, \ldots, p-1\}$ wird zum Sondieren herangezogen, z.B. $h'(c) = 1 + (c \bmod p-1)$ (Schrittweite).

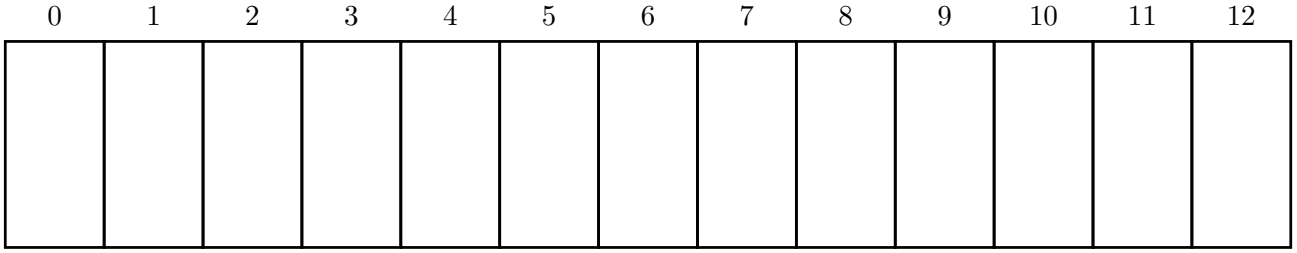
Sondierungsfunktion $sond(c, j) = j \cdot h'(c)$, Beispiel:



Doppel-Hashing mit Einfügen nach Brent: Um zufällig entstandene lange Sondierungskette zu vermeiden, wählt der folgende Algorithmus aus, ob der neue Eintrag oder der schon in der Tabelle vorhandene kollidierende Eintrag verschoben wird.

```
EINFÜGEN-BRENT-HASHTABELLE(Schlüssel neuerWert, Daten neueDaten)
     Rückgabewert: nichts; Seiteneffekte
 1   i ← h(neuerWert)
 2   while ht[i].status = BELEGT
 3   do ⌜ neufolgt ← (i + h'(neuerWert)) mod p
 4        altfolgt ← (i + h'(ht[i].wert) mod p
 5        if ht[neufolgt].status = FREI  oder  ht[altfolgt].status = BELEGT
 6        then ⌊ i ← neufolgt
 7        else ⌜ ht[altfolgt].wert ← ht[i].wert
 8                ht[altfolgt].status ← BELEGT
 9     ⌞        ⌞ ht[i].status ← ENTFERNT
10   ht[i].wert ← neuerWert
11   ht[i].daten ← neueDaten
12   ht[i].status ← BELEGT
```

Beispiel:



Kapitel 10

Prioritätswarteschlangen

Hermione: Now, if you two don't mind, I'm going to bed
before either of you come up with another clever idea
to get us killed. Or worse, expelled.
Ron: She needs to sort out her priorities.
(Harry Potter and the Sorcerer's Stone, 2001)

Operationen einer Prioritätswartschlange:
$\longrightarrow$ **[Buch, Abschnitt 10.1, S.257-258]**

EINFÜGEN-PW(x)**:** Es wird ein neues Element x mit unendlich großem Prioritätswert aufgenommen – d.h. das Element wird quasi hinten angestellt.

ISTLEER-PW$()$**:** Es wird geprüft, ob die Datenstruktur noch Elemente enthält.

MINIMUM-LIEFERN-PW$()$**:** Es wird dasjenige Element zurückgegeben und aus der Datenstruktur gelöscht, das den kleinsten Prioritätswert hat.

PRIO-VERBESSERN-PW(x,p)**:** Der Prioritätswert eines Elements x wird auf den Wert p gesetzt – dabei erlaubt man in der Regel nur die Verkleinerung des Prioritätswerts, d.h. das Erzwingen einer früheren Behandlung.

Zeitaufwand Insert:	
Zeitaufwand Extract-Minimum:	
Zeitaufwand Decrease-Key:	

```
DIJKSTRA-MIT-PW(Knoten V, Kanten E ⊂ V × V, Kosten γ : E → ℝ, Startknoten s ∈ V)
     Rückgabewert: Distanzen abstand, Wege vorgänger
 1   for alle Knoten u ∈ V
 2   do ⌜ abstand[u] ← ∞
 3      ⌞ vorgänger[u] → NULL
 4   abstand[s] ← 0
 5   Q ← allokiere Prio-Warteschlange()
 6   for alle Knoten u ∈ V
 7   do ⌊ Q.EINFÜGEN-PW(u, abstand[u])
 8   while ¬Q.ISTLEER-PW()
 9   do ⌜ nächster ← Q.MINIMUM-LIEFERN-PW()
10        for alle  v ∈ V mit (nächster, v) ∈ E
11        do ⌜ if abstand[v] > abstand[nächster] + γ[nächster, v]
12             then ⌜ abstand[v] ← abstand[nächster] + γ[nächster, v]
13                  Q.PRIO-VERBESSERN-PW(v, abstand[v])
14     ⌞     ⌞        ⌞ vorgänger[v] → nächster
15   return abstand, vorgänger
```

10.1 Binärer Heap

⟶ **[Buch, Abschnitt 10.2, S.259-266]**

Def. Heap-Eigenschaft:
Ein Baum erfüllt die *Heap-Eigenschaft* bezüglich einer Vergleichsrelation „$\succ$“ auf den Schlüsselwerten genau dann, wenn für jeden Knoten u des Baums gilt, dass $u.wert \succ v.wert$ für alle Knoten v aus den Unterbäumen von u.

Def. Min- bzw. Max-Heap:

Min-Heap: Ein binärer Baum der Heap-Eigenschaft mit der Relation „$<$“ erfüllt

Max-Heap: Ein binärer Baum der Heap-Eigenschaft mit der Relation „$>$“ erfüllt

```
EINFÜGEN-MIN-HEAP(Priorität prio, Daten daten)
     Rückgabewert: –; Seiteneffekt: Felder geändert
 1   anzahl ← anzahl + 1
 2   A_prio[anzahl] ← prio
 3   A_daten[anzahl] ← daten
 4   POSITIONSFELD-EINFÜGEN(daten, anzahl)
 5   AUFSTEIGEN-MIN-HEAP(anzahl)
```

AUFSTEIGEN-MIN-HEAP(Index *index*)
Rückgabewert: –; Seiteneffekt: Felder geändert
1 **while** $index > 1$ und $A[index] < A[\lfloor \frac{index}{2} \rfloor]$
2 **do** ⌜ VERTAUSCHE-HEAP($index, \lfloor \frac{index}{2} \rfloor$)
3 ⌞ $index \leftarrow \lfloor \frac{index}{2} \rfloor$

VERTAUSCHE-HEAP(Indizes i,k)
Rückgabewert: –; Seiteneffekt: Felder geändert
1 VERTAUSCHE(A_{prio}, i, k)
2 VERTAUSCHE(A_{daten}, i, k)
3 POSITIONSFELD-VERTAUSCHEN(i,k)

MINIMUM-LIEFERN-MIN-HEAP()
Rückgabewert: Element mit kleinster Priorität; Seiteneffekt: Felder geändert
1 VERTAUSCHEN-HEAP($1, anzahl$)
2 $anzahl \leftarrow anzahl - 1$
3 ABSINKEN-MIN-HEAP($1, anzahl$)
4 POSITIONSFELD-LÖSCHEN($A_{\text{daten}}[anzahl+1]$)
5 **return** $A_{\text{daten}}[anzahl+1]$

ABSINKEN-MIN-HEAP(Index *index*, obere Grenze r)
Rückgabewert: nichts; Seiteneffekt: Felder geändert
1 **if** $2 \cdot index \leq r$ und $A_{\text{prio}}[2 \cdot index] < A_{\text{prio}}[index]$
2 **then** ⌈ $min \leftarrow 2 \cdot index$
3 **else** ⌈ $min \leftarrow index$
4 **if** $2 \cdot index + 1 \leq r$ und $A_{\text{prio}}[2 \cdot index + 1] < A_{\text{prio}}[min]$
5 **then** ⌈ $min \leftarrow 2 \cdot index + 1$
6 **if** $min \neq index$
7 **then** ⌜ VERTAUSCHE-HEAP($index, min$)
8 ⌞ ABSINKEN-MIN-HEAP(min, r)

IST-LEER-MIN-HEAP()
Rückgabewert: wahr, wenn der Heap keine Elemente enthält
1 **return** $(anzahl = 0)$

POSITIONSFELD-EINFÜGEN(Daten *daten*, Index *index*)
Rückgabewert: –; Seiteneffekt: Feld geändert
1 $A_{\text{position}}[daten.nr] = index$

POSITIONSFELD-LÖSCHEN(Daten *daten*, Index *index*)
Rückgabewert: –; Seiteneffekt: Feld geändert
1 $A_{\text{position}}[daten.nr] = 0$

POSITIONSFELD-VERTAUSCHEN(Indizes i,k)
Rückgabewert: –; Seiteneffekt: Felder geändert
1 $A_{\text{position}}[A_{\text{daten}}[i].nr] = k$
2 $A_{\text{position}}[A_{\text{daten}}[k].nr] = i$

PRIO-VERBESSERN-MIN-HEAP(Nummer *nr*, Priorität *prio*)
Rückgabewert: –; Seiteneffekt: A geändert
1 $index \leftarrow A_{\text{position}}[nr]$
2 **if** $index = 0$
3 **then** **error** "Element nicht enthalten"
4 **else** **if** $prio > A_{\text{prio}}[index]$
5 **then** **error** "Unerlaubter Prioritätswert"
6 **else** $A_{\text{prio}}[index] \leftarrow prio$
7 AUFSTEIGEN-MIN-HEAP(*index*)

Zeitaufwand Insert:	
Zeitaufwand Extract-Min/Max:	
Zeitaufwand Decrease/Increase-Key:	

Dijkstra mit Heapverwaltung: Welche Laufzeit ergibt sich jetzt für den Dijkstra-Algorithmus bzw. den Prim-Algorithmus?

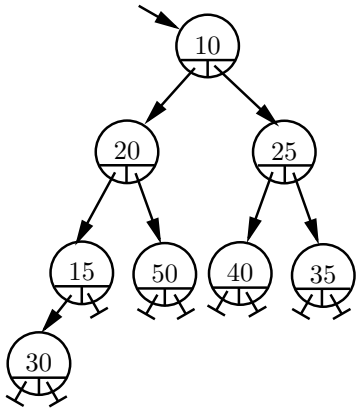
10.2 Heapsort

⟶ **[Buch, Abschnitt 10.3, S.266-274]**

```
HEAPSORT(Feld A)
    Rückgabewert: nichts; Seiteneffekt: A sortiert
1   AUFBAU-MAX-HEAP()
2   for i ← A.länge,...,2
3   do ⌜ VERTAUSCHE(A,1,i)
4      ⌞ ABSINKEN-MAX-HEAP(1,i−1)
```

```
ABSINKEN-MAX-HEAP(Index index, obere Grenze r)
    Rückgabewert: nichts; Seiteneffekt: Felder geändert
 1  wert ← A[index]
 2  repeat ⌜ alterIndex ← index
 3         if 2·index ≤ r und A[2·index] > wert
 4         then ⌜ max ← 2·index
 5              ⌞ maxWert ← A[2·index]
 6         else ⌜ max ← index
 7              ⌞ maxWert ← wert
 8         if 2·index+1 ≤ r und A[2·index+1] > maxWert
 9         then ⌜ max ← 2·index+1
10              ⌞ maxWert ← A[2·index+1]
11         if max ≠ index
12         then ⌜ A[index] ← A[max]
13         ⌞    ⌞ index ← max
14  until alterIndex = max
15  A[max] ← wert
```

Initialisieren eins Heaps durch iteratives Absteigen lassen.



AUFBAU-MAX-HEAP()
Rückgabewert: nichts; Seiteneffekt: A geändert
1 **for** $i \leftarrow \lfloor \frac{A.länge}{2} \rfloor, \ldots, 1$
2 **do** ⌊ ABSINKEN-MAX-HEAP$(i, A.länge)$

Zeitaufwand:

Beispiel:

A[1]	A[2]								A[10]
20	54	28	31	5	24	39	14	1	15

Kapitel 11

Extern gespeicherte Daten

11.1 Basisoperationen

Operationen zur Interaktion mit einem externen frei adressierbaren Speichermedium:
⟶ **[Buch, Abschnitt 11.1, S.277-278]**

Operation	Beschreibung
LESE-BLOCK(*adr*)	Lädt den Inhalt eines Speicherblocks in den Hauptspeicher.
SCHREIBE-BLOCK(*adr*)	Schreibt den im Hauptspeicher eingelagerten (und ggf. modifizierten) Inhalt eines Blocks wieder an seine Adresse im externen Speicher.
SCHREIBE-BLOCK(*adr*, *block*)	Schreibt einen im Hauptspeicher befindlichen Block an die angegebene Adresse im Speicher.
BLOCK-RESERVIEREN()	Ein bisher ungenutzter Speicherblock wird durch das Dateimanagementsystem reserviert und zur Verfügung gestellt.
BLOCK-FREIGEBEN(*adr*)	Ein bisher genutzter Speicherblock wird wieder als frei nutzbarer Speicherbereich dem Dateimanagementsystem überantwortet.

11.2 Mengenproblem: B-Bäume

⟶ **[Buch, Abschnitt 11.3, S.282-297]**

Def. Verallgemeinerter Suchbaum:
Ein *verallgemeinerter Suchbaum* ist entweder leer oder er besteht aus einem Knoten mit $k > 0$ Schlüsseln $s_1 \leq \ldots\ldots s_k$ und $k+1$ verallgemeinerten Suchbäumen $u_1, \ldots, u_{k+1}$ als Unterbäume, wobei für jeden Schlüssel s_i gilt, dass

- die Schlüssel in u_i kleiner als s_i und
- die Schlüssel in u_{i+1} größer gleich als s_i sind.

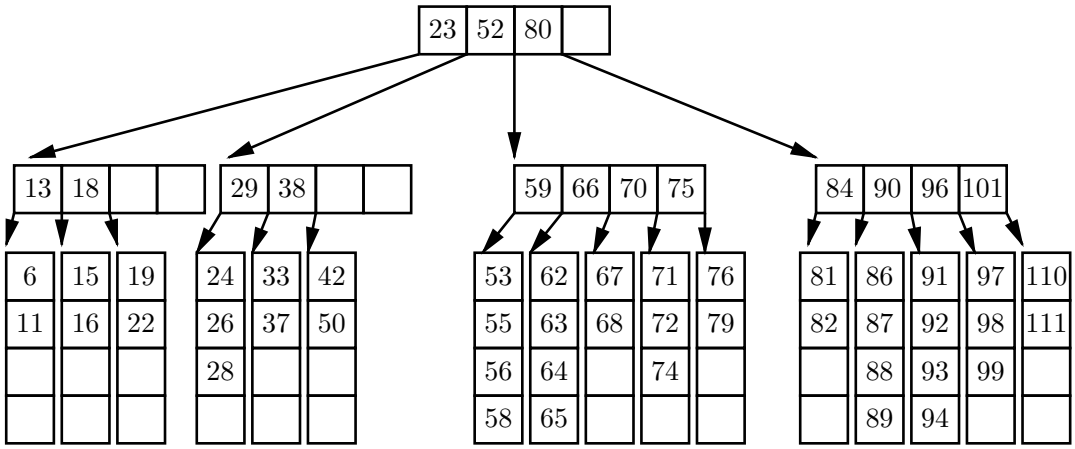
Def. B-Baum:
Ein *B-Baum der Ordnung m* mit $m \in \mathbb{N}$ ist ein verallgemeinerter Suchbaum, bei dem

- jeder Knoten höchstens $2 \cdot m$ Schlüssel enthält,
- jeder Knoten unterhalb der Wurzel mindestens m Schlüssel enthält und
- alle Nullzeiger gleich tief im Baum liegen.

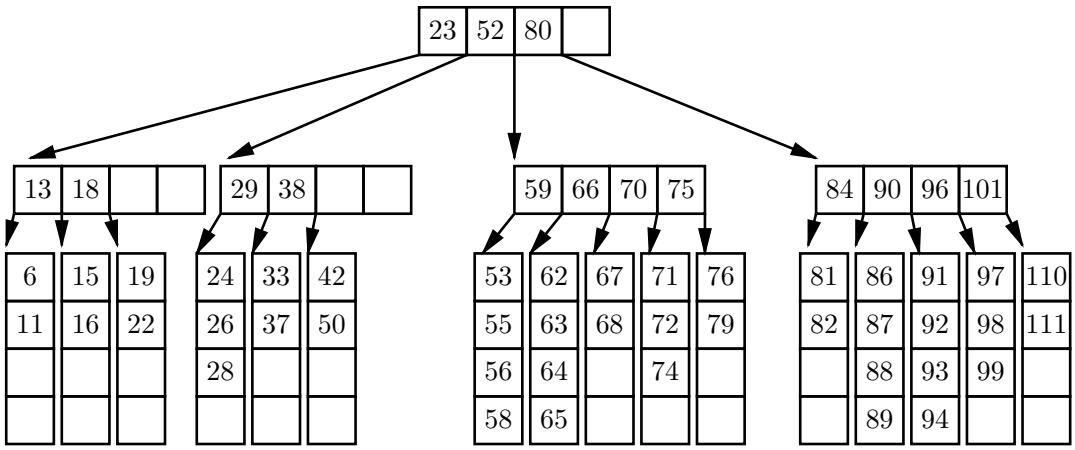
```
SUCHEN-BLOCK-BBAUM(Datenblock el, Schlüssel gesucht)
    Rückgabewert: Info ob Element gefunden, letzter betrachteter Datenblock, Index
 1  index ← 1
 2  while index ≤ el.anzahl und el.s_index ≤ gesucht
 3  do [ index ← index + 1
 4  switch
 5  case el.s_index = gesucht : return true , el, index
 6  case el.s_index > gesucht :
 7       if el.u_index = NULL
 8       then [ return false , el, index
 9       else [ el' → LESE-BLOCK(el.u_index)
10  case index > el.anzahl : el' → LESE-BLOCK(el.u_anzahl+1)
11  return SUCHEN-BLOCK-BBAUM(el', gesucht)
```

```
SUCHEN-BBAUM(Schlüssel gesucht)
    Rückgabewert: gesuchte Daten bzw. Fehler falls nicht enthalten
1   el → LESE-BLOCK(wurzel)
2   erfolg ←
    el →          } SUCHEN-BLOCK-BBAUM(el, gesucht)
    index ←
3   if erfolg
4   then [ return el.s_index
5   else [ error "Element nicht gefunden"
```

Beispiel: Suchen(68)



Beispiel: Suchen(101)

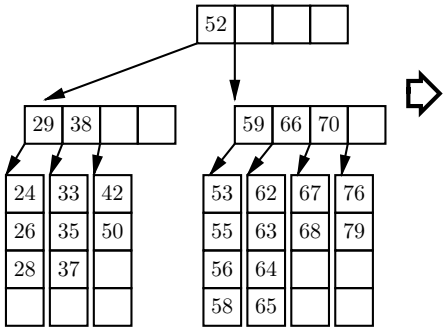


```
EINFÜGEN-BLOCK-BBAUM(Block eℓ, Schlüssel neuerWert, Unterbaum eℓ')
    Rückgabewert: –; Seiteneffekte
 1  i ← eℓ.anzahl
 2  while i ≥ 1 und eℓ.s_i > neuerWert
 3  do ⌜ eℓ.s_{i+1} ← eℓ.s_i
 4     ⌞ eℓ.u_{i+1} → eℓ.u_i
 5  eℓ.s_i ← neuerWert
 6  eℓ.u_i → eℓ'
```

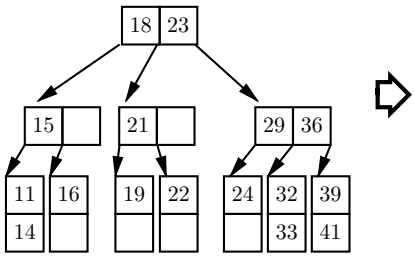
```
TEILE-BLOCK-BBAUM(Block eℓ)
    Rückgabewert: Schlüssel, rechter Block
 1  eℓ' → BLOCK-RESERVIEREN()
 2  for i ← 1,...,m
 3  do ⌜ eℓ'.s_i ← eℓ.s_{m+1+i}
 4     ⌞ eℓ'.u_i → eℓ.u_{m+1+i}
 5  eℓ'.u_{m+1} → eℓ.u_{2·m+2}
 6  eℓ'.anzahl ← m
 7  eℓ.anzahl ← m
 8  SCHREIBE-BLOCK(eℓ)
 9  SCHREIBE-BLOCK(eℓ')
10  return eℓ.s_{m+1}, eℓ'
```

```
EINFÜGEN-BBAUM(Schlüssel neuerWert)
    Rückgabewert: —; Seiteneffekte
 1  eℓ → LESE-BLOCK(wurzel)
 2  erfolg ←
    eℓ →          } SUCHEN-BLOCK-BBAUM(wurzel, neuerWert)
    index ←
 3  if erfolg
 4  then ⌈error "Element bereits enthalten"
 5  else ⌈EINFÜGEN-BLOCK-BBAUM(eℓ, index, neuerWert, NULL)
 6         while eℓ.anzahl > 2 · m
 7         do ⌈wert ←
                eℓ' →     } TEILE-BLOCK-BBAUM(eℓ)
 8            eℓ'' → eℓ.vorgänger
 9            if eℓ'' = NULL
10            then ⌈wurzel → BLOCK-RESERVIEREN()
11                   wurzel.s1 ← wert
12                   wurzel.u1 → eℓ
13                   wurzel.u2 → eℓ'
14                 ⌊eℓ → wurzel
15            else ⌈eℓ → eℓ''
16               ⌊  ⌊EINFÜGEN-BLOCK-BBAUM(eℓ, wert, eℓ')
17       ⌊SCHREIBE-BLOCK(eℓ)
```

Beispiel: Einfügen der 57

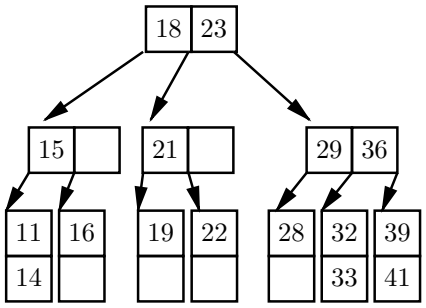


Beispiel: Einfügen der 35



SUCHE-NACHFOLGER-BBAUM(Block $e\ell$, Index *index*)
Rückgabewert: Block mit dem Nachfolgerelement
1 $e\ell \rightarrow$ LESE-BLOCK($e\ell.u_{index+1}$)
2 **while** $e\ell \neq$ NULL
3 **do** $\lceil e\ell \rightarrow$ LESE-BLOCK($e\ell.u_1$)
4 **return** $e\ell$

Beispiel: Nachfolgerknoten



LÖSCHEN-BLOCK-BBAUM(Block $e\ell$, Index *index*)
Rückgabewert: —; Seiteneffekte
1 **for** $i \leftarrow index+1,\ldots,e\ell.anzahl$
2 **do** $\lceil e\ell.s_{i-1} \leftarrow e\ell.s_i$
3 $\quad \lfloor e\ell.u_i \rightarrow e\ell.u_{i+1}$
4 $e\ell.anzahl \leftarrow e\ell.anzahl - 1$

```
LÖSCHEN-BBAUM(Schlüssel löschWert)
    Rückgabewert: nichts falls erfolgreich bzw. Fehler sonst
 1  eℓ → LESE-BLOCK(wurzel)
 2  erfolg ←
    eℓ →                  } SUCHEN-BLOCK-BBAUM(eℓ, löschWert)
    index ←
 3  if ¬erfolg
 4  then [ error "Element nicht vorhanden"
 5  else ⌜ if eℓ.u_index ≠ NULL
 6       then ⌜ eℓ' → SUCHE-NACHFOLGER-BBAUM(eℓ, index)
 7            |   eℓ.s_index ← eℓ'.s_1
 8            |   eℓ → eℓ'
 9            ⌞ index ← 1
10       LÖSCHEN-BLOCK-BBAUM(eℓ, index)
11       while eℓ.anzahl < m und eℓ.vorgänger ≠ NULL
12       do ⌜ eℓ' → LESE-BLOCK(eℓ.vorgänger)
13          | index ← INDEX-DES-UNTERBAUMS-BBAUM(eℓ', ℓ)
14          | if index ≤ eℓ'.anzahl
15          | then ⌜ eℓ'' → LESE-BLOCK(eℓ'.u_{index+1})
16          |      |  if eℓ''.anzahl > m
17          |      ⌞ then [ LINKS-SCHIEBEN-BBAUM(eℓ', index)
18          | if eℓ.anzahl < m
19          | then ⌜ if index > 1
20          |      |  then ⌜ eℓ'' → LESE-BLOCK(eℓ'.u_{index−1})
21          |      |       |  if eℓ''.anzahl > m
22          |      ⌞       ⌞ then [ RECHTS-SCHIEBEN-BBAUM(eℓ', index)
23          | if eℓ.anzahl < m
24          | then ⌜ if index ≤ eℓ'.anzahl
25          |      |  then [ VERSCHMELZE-BLÖCKE-BBAUM(eℓ', index)
26          |      ⌞ else [ VERSCHMELZE-BLÖCKE-BBAUM(eℓ', index − 1)
27          |   SCHREIBE-BLOCK(eℓ)
28          ⌞ eℓ → eℓ'
29       if eℓ.vorgänger = NULL und eℓ.anzahl = 0
30       then ⌜ BLOCK-FREIGEBEN(anker
31            ⌞ anker → eℓ.u_1
32       ⌞ else [ SCHREIBE-BLOCK(eℓ)
```

```
INDEX-DES-UNTERBAUMS-BBAUM(Block eℓ, Unterbaum eℓ')
    Rückgabewert: Index
1  i ← 1
2  while i ≤ eℓ.anzahl + 1 und eℓ.u_i ≠ eℓ'
3  do [ i ← i + 1
4  if i ≤ eℓ.anzahl + 1
5  then [ return i
6  else [ error "Unterbaum existiert nicht"
```

LINKS-SCHIEBEN-BBAUM(Block $e\ell$, Unterbaum-Index *index*)

Rückgabewert: —; Seiteneffekt

```
1  eℓ' → eℓ.u_index
2  eℓ'' → eℓ.u_index+1
3  anz ← eℓ'.anzahl
4  eℓ'.s_anz+1 ← eℓ.s_index
5  eℓ.s_index ← eℓ''.s_1
6  eℓ'.u_anz+2 → eℓ''.u_1
7  eℓ'.anzahl ← eℓ'.anzahl + 1
8  eℓ''.u_1 → eℓ''.u_2
9  LÖSCHEN-BLOCK-BBAUM(eℓ'', 1)
10 SCHREIBE-BLOCK(eℓ'')
```

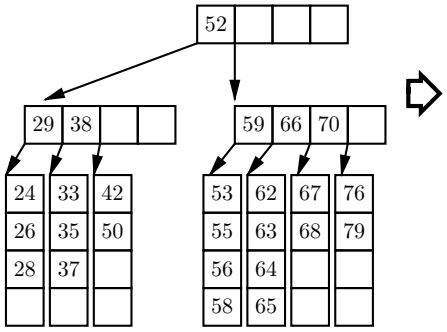
RECHTS-SCHIEBEN-BBAUM(Block $e\ell$, Unterbaum-Index *index*)

Rückgabewert: —; Seiteneffekt

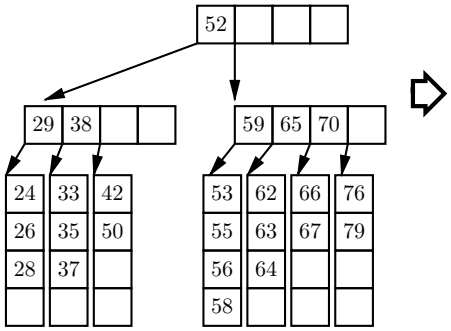
```
1  eℓ' → eℓ.u_index−1
2  eℓ'' → eℓ.u_index
3  for i ← 1,...,eℓ''.anzahl
4  do ⌜ eℓ''.s_i+1 ← eℓ''.s_i
5     ⌞ eℓ''.u_i+2 → eℓ''.u_i+1
6  eℓ''.u_2 → eℓ''.u_1
7  eℓ''.s_1 ← eℓ.s_index
8  anz ← eℓ'.anzahl
9  eℓ.s_index ← eℓ'.s_anz
10 eℓ''.u_1 → eℓ''.u_anz+1
11 eℓ'.anzahl ← eℓ'.anzahl − 1
12 eℓ''.anzahl ← eℓ''.anzahl + 1
13 SCHREIBE-BLOCK(eℓ')
```

```
VERSCHMELZE-BLÖCKE-BBAUM(Block eℓ, Index index)
    Rückgabewert: —; Seiteneffekt: alle Elemente in eℓ
 1  eℓ' → eℓ.u_index
 2  eℓ'' → eℓ.u_index+1
 3  anz ← eℓ'.anzahl + 1
 4  eℓ'.s_anz ← eℓ.s_index
 5  i ← 1
 6  while i ≤ eℓ''.anzahl
 7  do ⌜ eℓ'.s_anz+i ← eℓ''.s_i
 8     ⌞ eℓ'.u_anz+i → eℓ''.u_i
 9     ⌞ i ← i + 1
10  eℓ'.u_anz+i → eℓ''.u_i
11  eℓ'.anzahl ← 2 · m
12  LÖSCHEN-BLOCK-BBAUM(eℓ, index)
13  BLOCK-FREIGEBEN(eℓ'')
```

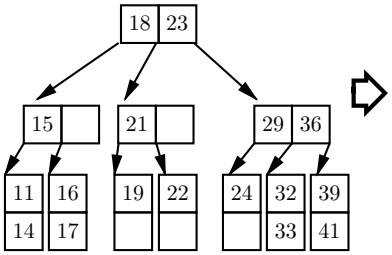
Beispiel: Löschen der 68



Beispiel: Löschen der 76



Beispiel: Löschen der 18



Kapitel 12 Selbstorganisation

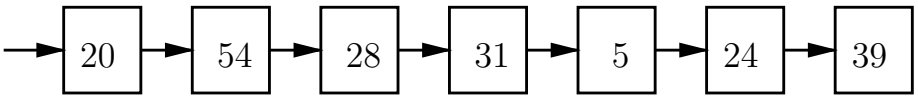
Dory: Give it up old man, you can't fight evolution,
I was built for speed!
(Finding Nemo, 2003)

12.1 Mengenproblem: Selbstorganisierende Liste

⟶ **[Buch, Abschnitt 12.1, S.309-312]**

```
SUCHEN-SOLISTE(Schlüssel gesucht)
     Rückgabewert: gesuchte Daten bzw. Fehler falls nicht enthalten
 1   switch
 2   case anker = NULL : error "Element nicht enthalten"
 3   case anker ≠ NULL und anker.wert = gesucht :
 4         return anker.daten
 5   case anker ≠ NULL und anker.wert ≠ gesucht :
 6         eℓ → anker
 7         while eℓ.nächstes ≠ NULL und eℓ.nächstes.wert ≠ gesucht
 8         do ⌊ eℓ → eℓ.nächstes
 9         if eℓ.nächstes ≠ NULL
10         then ⌈ ergebnis → eℓ.nächstes
11                 eℓ.nächstes → eℓ.nächstes.nächstes
12                 eℓ.nächstes → anker
13                 anker → eℓ
14               ⌊ return eℓ
15         else  ⌊ error "Element nicht gefunden"
```

Beispiel:



Suche(5):

Einfügen(13):

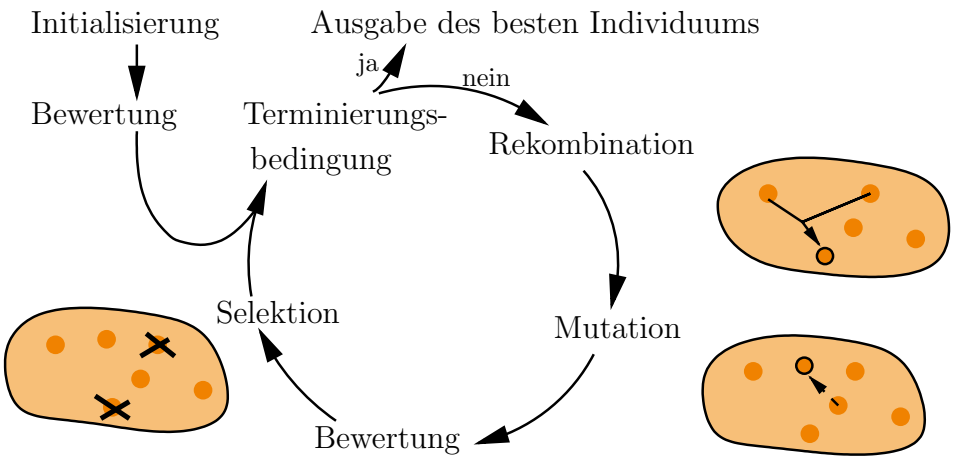
Suchen(28):

Suchen(5):

Löschen(20):

12.2 Rundreiseproblem: Evolutionäre Algorithmen

⟶ **[Buch, Abschnitt 12.2, S.312-320]**

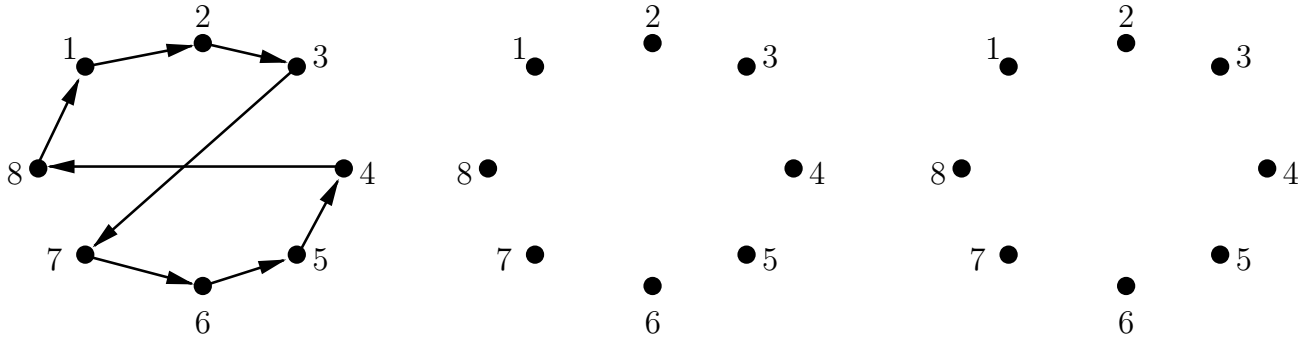


Datentyp Individuum: Permutation in Feld *A*, reellwertige Bewertung in *fitness*

```
INIT-INDIVIDUUM(Problemgröße n)
    Rückgabewert: nichts; Seiteneffekt: Permutation in A
1   fitness ← ∞
2   genotyp ← allokiere Feld der Länge n
3   for k ← 1,...,n
4   do ⌊ genotyp[k] ← k
5   for k ← 1,...,n−1
6   do ⌈ m ← wähle gleichverteilte Zufallszahl aus k+1,...,n
7      ⌊ VERTAUSCHE(genotyp,k,m)
```

```
VERTAUSCHENDE-MUTATION()
    Rückgabewert: neues Individuum
1   ind → allokiere Individuum für A.länge Knoten
2   for i ← 1,...,genotyp.länge
3   do ⌊ ind.genotyp[i] ← genotyp[i]
4   u1 ← wähle gleichverteilte Zufallszahl aus 1,...,genotyp.länge
5   u2 ← wähle gleichverteilte Zufallszahl aus 1,...,genotyp.länge
6   ind.genotyp[u1] ← genotyp[u2]
7   ind.genotyp[u2] ← genotyp[u1]
8   return ind
```

Beispiel: vertauschende Mutation

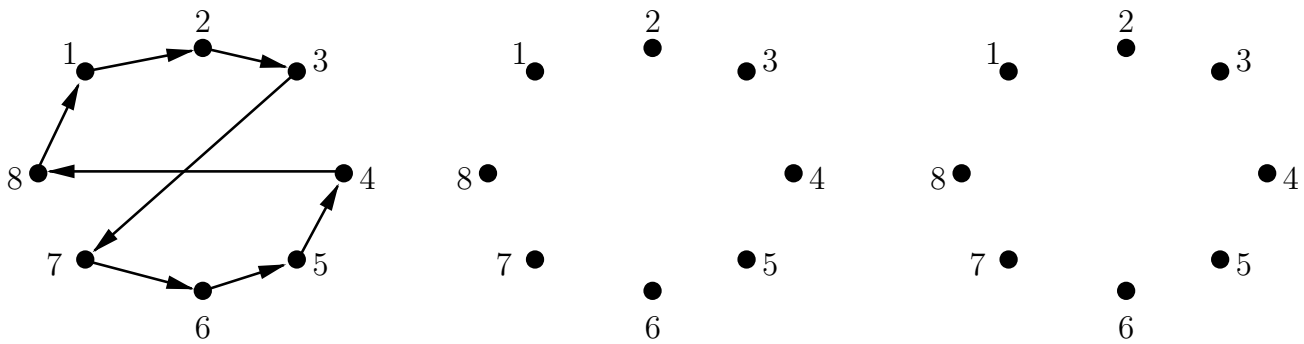


(1,2,3,7,6,5,4,8)

```
EA-HANDLUNGSREISENDENPROBLEM(Graph G mit n Knoten)
     Rückgabewert: Reihenfolge der Knoten
 1   pop → allokiere Feld für 50 Individuen
 2   for i ← 1,...,10
 3   do ⌜ pop[i] → allokiere Individuum
 4        pop[i].INIT-INDIVIDUUM()
 5      ⌞ pop[i].fitness ← Kosten von pop[i].genotyp
 6   for generation ← 1,...,2000
 7   do ⌜ for i ← 1,...,40
 8        do ⌜ ind → wähle gleichverteilt zufällig aus pop[1]...pop[10]
 9             pop[i+10] → ind.VERTAUSCHENDE-MUTATION()
10           ⌞ pop[i+10].fitness ← Kosten von pop[i+10].genotyp
11      ⌞ HEAPSORT(pop)
12   return pop[1].genotyp
```

```
INVERTIERENDE-MUTATION()
     Rückgabewert: neues Individuum
 1   ind → allokiere Individuum für genotyp.länge Knoten
 2   for i ← 1,...,genotyp.länge
 3   do ⌞ ind.genotyp[i] ← genotyp[i]
 4   u1 ← wähle gleichverteilte Zufallszahl aus 1,...,genotyp.länge
 5   u2 ← wähle gleichverteilte Zufallszahl aus 1,...,genotyp.länge
 6   if u1 > u2
 7   then ⌜ h ← u1
 8          u1 ← u2
 9        ⌞ u2 ← h
10   for j ← u1,...,u2
11   do ⌞ ind.genotyp[u2+u1−j] ← genotyp[j]
12   return ind
```

Beispiel: invertierende Mutation



(1,2,3,7,6,5,4,8)

Kapitel 13 Zusammenfassung

⟶ **[Buch, Kapitel 13, S.325-330]**

13.1 Mengenproblem

Datenstruktur	Suchen	Einfügen	Löschen	Anmerkung
unsortiertes Feld				
sortiertes Feld				
unsortierte Liste				
sortierte Liste				
Skip-Liste				
bin. Suchbaum				
AVL-Baum				
Hashing				
B-Baum				
selbstorg. Liste				

Wie groß ist die Datenmenge? Falls die Größe der Datenmenge...

- sehr klein ($<$ ca.100) ist:
- moderat ($<$ ca.1000000) ist:
- sehr groß ($\geq$ ca.1000000) und ggf. nicht in den Hauptspeicher passt:

Wie groß ist die Dynamik der gespeicherten Elemente?

Was ist über Zugriffswahrscheinlichkeiten der Elemente bekannt?

Müssen alle Elemente ausgegeben werden?

Wie wird auf die Elemente zugegriffen?

13.2 Sortieren

Algorithmus	Best-Case	Avg.-Case	Worst-Case	Anmerkung
Bubbelsort				
Insertionsort				
Shellsort				
Selectionsort				
Mergesort				
Quicksort				
Heapsort				
Counting-Sort				
Radix-Sort				

Wie viele Elemente werden sortiert? Ist die Anzahl der zu sortierenden Datenelemente...

- klein ($<$ ca. 100):
- groß ($<$ ca. 2000000):
- sehr groß ($\geq$ ca. 2000000) und können nicht im Hauptspeicher gehalten werden:

Sind die Daten vorsortiert?

Muss der Algorithmus stabil sein?

Wie wichtige ist optimale Laufzeit?

13.3 Kürzeste Wege

Algorithmus	ein Startknoten	alle Knotenpaare	Anmerkung
Breitensuche (Adj.liste)			
Breitensuche (Adj.matr.)			
Dijkstra mit Feld			
Dijkstra mit Heap			
Floyd-Warshall			

Wie dicht ist der Graph?

Gelten Eigenschaften bezüglich der Gewichte?

Weist der Graph andere Besonderheiten auf?

13.4 Rundreise

Algorithmus	Laufzeit	Genauigkeit	Anmerkung
Backtracking			
Greedy Heuristik			
MSB-Approximation			
Bitonische Heuristik			
Evol. Heuristik			

Spezialfälle?

Anhang A

Notation des Pseudo-Code

In diesem Abschnitt wird die Notation der Algorithmen kompakt erläutert und vorgestellt.

Die einzelnen Algorithmen entsprechen Funktionen oder Methoden in realen Programmiersprachen. Dabei gibt die erste Zeile den Namen der Methode und die formalen Parameter für den Aufruf an, die zweite Zeile spezifiziert den/die Rückgabewert(e) genauer und ab der dritten Zeile beginnt die Beschreibung des schrittweisen Ablaufs des Algorithmus.

N-ÜBER-2(Wert n)
    Rückgabewert: Wert von $\binom{n}{2}$
1 $produkt \leftarrow n \cdot (n-1)$
2 **return** $\frac{produkt}{2}$

Dabei verzichten wir in den Beschreibungen auf eine genaue Angabe der Datentypen – so wäre beim obigen Algorithmus eine Angabe $n \in \mathbb{N}$ sinnvoll. Dies ergibt sich jedoch aus dem Kontext der Algorithmen.

Der „$\leftarrow$“ steht für eine Wertzuweisung. Der mathematische Ausdruck rechts vom Pfeil wird berechnet und der Variablen mit dem links vom Pfeil angegebenen Namen zugewiesen. Dabei erlauben wir in den mathematischen Ausdrücken alle üblichen Funktionen und Notationen der Mathematik.

Der Rückgabewert des Algorithmus wird mit dem letzten ausgeführten Befehl „**return**“ angezeigt. Auch hier wird der Ausdruck berechnet und der resultierende Wert ist das Ergebnis des Algorithmus. Unabhängig davon, wo das „**return**“ im Algorithmus steht, sobald es ausgeführt wird, ist der Ablauf des Algorithmus erfolgreich beendet.

Die formalen Parameter zeigen an, mit welchen Werten der Algorithmus aufgerufen werden muss und unter welchem Namen diese Werte dann im Algorithmus benutzt werden können. Dabei handelt es sich um sog. Call-By-Value-Parameter: Es wird lediglich ein Wert in den Algorithmus hinein übergeben – eine Zuweisung zum Namen des Wertes im Algorithmus hat keine Auswirkung außerhalb des Algorithmus.

Ein solche Funktionsaufruf wird dann beispielsweise wie im folgenden Beispiel notiert.

MÖGLICHKEITEN-2-KUGELN-ZU-ZIEHEN(Anzahl der Kugeln n)
Rückgabewert: Anzahl der Möglichkeiten
1 *ergebnis* $\leftarrow$ N-ÜBER-2(n)
2 **return** *ergebnis*

Verlangt ein Algorithmus mehrere Werte als formale Parameter, werden diese per Komma voneinander getrennt.

Sollte in der Ausführung des Algorithmus ein Fehler auftreten und er kann nicht zu einem erfolgreichen Abschluss geführt werden, kann dies mit dem Befehl „**error**“ und einer entsprechenden Fehlermeldung veranlasst werden. Die folgende Variante des obigen Algorithmus zeigt dies.

N-ÜBER-2(Wert n)
Rückgabewert: Wert von $\binom{n}{2}$
1 **if** $n > 1$
2 **then** ⌜ *produkt* $\leftarrow n \cdot (n-1)$
3 ⌞ **return** $\frac{produkt}{2}$
4 **error** “zu kleiner Wert”

Die „**if**“-Anweisung gibt dabei an, dass die hinter dem „**then**“ eingerückten Anweisungen nur dann ausgeführt werden, wenn die Bedingungen nach dem „**if**“ wahr ist. Der Beginn der Anweisungen wird zusätzlich durch ⌜ und das Ende durch ⌞ markiert. Ferner können zusätzlich auch mit dem Schlüsselwort „**else**“ zusätzliche Anweisungen angegeben werden, die genau dann ausgeführt werden, wenn die Bedingung falsch ist. Ein Beispiel ist im nächsten Algorithmus enthalten.

Ein Feld ist eine Datenstruktur, in der mehrere Elemente gleichen Datentyps gespeichert werden. Die Anzahl der speicherbaren Elemente ist durch die feste Größe des Felds begrenzt, die für ein Feld A durch $A.länge$ abgefragt werden kann. Auf die einzelnen Elemente kann über den Index $1 \leq i \leq A.länge$ lesend und schreibend mit $A[i]$ zugegriffen werden.

Die zuletzt eingeführten Konzepte werden im folgenden Algorithmus veranschaulicht, der eine Binärzahl in der Darstellung des Zweierkomplements in eine ganze Zahl umrechnet – bitte nicht so nachprogrammieren, da dies überhaupt nicht effizient ist.

DEKODIERE-ZWEIERKOMPLEMENT(Feld mit binären Werten *bits*)
Rückgabewert: durch die Bits dargestellte ganze Zahl
1 $n \leftarrow bits.länge$
2 *wert* $\leftarrow$ ALS-GANZE-ZAHL($bits, 2, n$)
3 **if** $bits[1] = 0$
4 **then** ⌞ **return** *wert*
5 **else** ⌜ *maxwert* $\leftarrow 2^{n-1}$
6 ⌞ **return** *wert* $-$ *maxwert*

Für die wiederholte Ausführung eines Abschnitts des Algorithmus stehen insgesamt drei verschiedene Schleifenarten zur Verfügung, wobei es von einer Schleife noch zwei Varianten gibt.

In der sog. While-Schleife wird immer wieder die Bedingung nach dem Schlüsselwort „**while**“ überprüft und, falls die Bedingung wahr ist, die Anweisungen nach dem „**do**“ ausgeführt. Dies endet erst

dann, wenn die Bedingung zu falsch ausgewertet wird. Danach setzt die Abarbeitung des Algorithmus bei der Anweisung hinter der While-Schleife fort.

ALS-GANZE-ZAHL(Feld mit binären Werten *bits*, Index ℓ, Index r)
Rückgabewert: durch die Bits dargestellte ganze Zahl
1 $wert \leftarrow 0$
2 $index \leftarrow r$
3 **while** $index \geq \ell$
4 **do** $\ulcorner wert \leftarrow 2 \cdot wert + bits[index]$
5 $\llcorner index \leftarrow index + 1$
6 **return** *wert*

Ist bereits vor dem Betreten der Schleife bekannt, wie oft diese durchlaufen werden soll, wird meist eine sog. For-Schleife benutzt. In der ersten nachfolgend dargestellten Variante werden die Iterationen durch eine Zuweisung und den Wertebereich festgelegt.

ALS-GANZE-ZAHL(Feld mit binären Werten *bits*, Index ℓ, Index r)
Rückgabewert: durch die Bits dargestellte ganze Zahl
1 $wert \leftarrow 0$
2 **for** $i \leftarrow 0, \ldots, (r - \ell)$
3 **do** $\lceil wert \leftarrow 2 \cdot wert + bits[r + i]$
4 **return** *wert*

In einer zweiten Variante der For-Schleife werden die angegeben Anweisungen für alle Elemente einer Menge durchgeführt. Das nachfolgende Beispiel illustriert dies.

DURCHSCHNITT(Menge von ganzen Zahlen M)
Rückgabewert: Durchschnitt der Zahlen
1 $summe \leftarrow 0$
2 **for** alle $x \in M$
3 **do** $\lceil summe \leftarrow summe + x$
4 **return** $\frac{summe}{\#M}$

Als letzte Schleifenart stellen wir die Repeat-Until-Schleife vor, bei der in jeder Iteration die hinter dem „**repeat**“ eingerückten Anweisungen ausgeführt werden und danach geprüft wird, ob die Bedingung zum Abbruch der Schleife, nach dem „**until**“ erfüllt ist. Bei dieser Schleifenart werden die Anweisungen immer mindestens einmal ausgeführt.

SUMME-GROSS-GENUG(Feld mit Zahlen A, Zielwert *wert*)
Rückgabewert: Index, ab dem die Summe den Zielwert erreicht
1 $summe \leftarrow 0$
2 $index \leftarrow 0$
3 **repeat** $\ulcorner index \leftarrow index + 1$
4 $\llcorner summe \leftarrow summe + A[index]$
5 **until** $summe \geq wert$
6 **return** *index*

Falls mehrere zueinander in Bezug stehende If-Verzweigungen benötigt werden, kann der Algorithmus wesentlich leichter lesbar sein, wenn man stattdessen das Switch-Statement benutzt. Nach dem Schlüsselwort „**switch**“ werden die verschiedenen Fälle jeweils mit „**case**“ und einer Bedingung eingeführt. Nach dem Doppelpunkt stehen jeweils die auszuführenden Anweisungen.

FAKULTÄT(Zahl n)
Rückgabewert: Fakultät von n
1 **switch**
2 **case** $n < 1$: **error** “falsches Argument”
3 **case** $n = 1$: **return** 1
4 **case** $n > 1$: **return** $b \cdot$ FAKULTÄT$(n-1)$

Weil im Switch-Statement beliebige Bedingungen formuliert werden können, weicht es stark von gleichnamigen Sprachelementen z.B. in der Programmiersprache Java ab, bei denen nur einzelne Werte eines Aufzählungsdatentyps als Fall formuliert werden können.

Im Hauptteil des Buchs spielen dynamische Datenstrukturen eine große Rolle, bei denen immer wieder Speichereinheiten aus dem freien Hauptspeicher reserviert wird. Auf diesen Speicher kann nur durch sog. Zeigervariablen zugegriffen werden. Diese Zeiger erlauben u.a. die beliebige strukturelle Anordnung der Speichereinheiten. Daher unterscheidet sich eine Zuweisung zu einer Zeigervariablen auch von einer Wertzuweisung zu einer normalen Variablen. Bei der Wertzuweisung erhält die Variable eine Kopie des Werts und kann diesen beliebig ändern. Bei der Zeigervariablen können mehrere Variablen auf dasselbe Objekt im Speicher zeigen – werden also Änderungen im Objekt vorgenommen, sind diese über alle Zeigervariablen sichtbar. Um Missverständnissen vorzubeugen, machen wir die Art der Zuweisung in der Syntax deutlich. Sind a und b zwei Zeigervariablen, ist „$a \rightarrow b$“ zu lesen als, „a verweist jetzt auf dasselbe Objekt wie b“.

LISTE-INITIALISIEREN()
Rückgabewert: –; Seiteneffekt: Anker wird gesetzt
1 *anker* $\rightarrow$ **allokiere** Element()
2 *anker.nächstes* $\rightarrow$ NULL
3 *anker.bemerkung* $\leftarrow$ „Dummy-Element“

Neuer Speicher für Objekte wird über den Befehl „**allokiere**“ angefordert, welcher einen Zeiger auf den Speicher zurück liefert. Nach der ersten Zeigerzuweisung verweist *anker* auf das neu angelegte Objekt der Datenstruktur Element. Auf den Inhalt eines solchen Objekts kann man über den Namen einer Zeigervariablen, einen anschließenden „.“ sowie den Namen eines Attributs der Datenstruktur zugreifen. Ist das Attribut ebenfalls eine Zeigervariablen, gelten auch dafür die Regeln bezüglich Zugriff und Zuweisung. Ist das Attribut keine Zeigervariable, kann sein Inhalt per Wertzuweisung geändert werden, wie in der dritten Zeile des obigen Algorithmus. Ein besonderer Zeigerwert ist der Verweis auf Nichts, der durch „NULL“ bezeichnet wird. Bei der Speicherallokation werden oft Vorbelegungen der Attribute direkt in der Klammer nach dem Namen der Datenstruktur angegeben.

Die vorher bereits angesprochenen Felder sind in unserer Notation ebenfalls eine Datenstruktur, deren Instanzen im freien Speicher abgelegt werden müssen. Daher werden neue Felder ebenfalls mit dem Befehl „**allokiere**“ erzeugt.

Eine weitere Besonderheit unserer Pseudo-Code-Notation ist die Möglichkeit, dass ein Algorithmus mehr als einen Rückgabewert hat (wie dies beispielsweise auch in den Sprachen Go oder Lua möglich ist). Hierfür müssen in dem Algorithmus nach einem „**return**“ mehrere durch Komma getrennten Werte angegeben werden. Beim Aufruf des Algorithmus wird durch mehrere Zuweisungen und eine große geschweifte Klammer angezeigt, welchen Variablen das Ergebnis zugewiesen wird und vor allem auch durch die Art des Zuweisungspfeils, ob es sich um eine Wert- oder eine Zeigerzuweisung handelt.

```
MINIMUM-UND-MAXIMUM(Liste el)
    Rückgabewert: minimaler und maximaler Wert in der Liste
1   if el.nächster = NULL
2   then ⌊ return el.wert, el.wert
3   else ⌈ min ←
         ⌊ max ←  } MINIMUM-UND-MAXIMUM(el.nächster)
4        if el.wert < min
5        then ⌊ min ← el.wert
6        if el.wert > max
7        then ⌊ max ← el.wert
8        ⌊ return min, max
```

Anhang B

Prüfungen

Ron: I'm ready! Ask me any questions.
Hermione: All right, what's the three most crucial ingredients in a Forgetfulness Potion?
Ron: I forgot.
Hermione: And what may I ask do you plan to do if this comes up in the final exam?
Ron: Copy off you?
(Harry Potter and the Sorcerer's Stone, 2001)

Hinweis:

- Zugelassene Hilfsmittel: handschriftlicher Spickzettel (4 Seiten), Taschenrechner
- Prüfungszeit: 120 Minuten
- Für die Prüfungen gilt, dass 1 Punkt etwa 2 Minuten Bearbeitungszeit entspricht.

B.1 Sommersemester 2016

Aufgabe 1: Hashing 3+4 Punkte

Gegeben sei eine anfangs leere Hashtabelle mit 11 Feldern, in die der Reihe nach die folgenden Schlüssel eingefügt werden sollen: 7, 3, 29, 62, 36, 69. Geben Sie die Belegung der Tabelle nach der letzten Einfügeoperation an und kennzeichnen Sie die Sondierungsfolge für jedes Element.

a) Benutzen Sie die Hashfunktion $h(x) = x \bmod 11$ mit quadratischem Sondieren.

Position	0	1	2	3	4	5	6	7	8	9	10
Inhalt											

b) Doublehashing mit $h(x) = x \bmod 11$, $h'(x) = 1 + (x \bmod 10)$ und Brent's Algorithmus.

Position	0	1	2	3	4	5	6	7	8	9	10
Inhalt											

Aufgabe 2: Suche im Feld 5+3 Punkte

Gegeben sei das Feld

1	9	13	16	18	19	20	22	23

.

a) Suchen Sie das Element 9 jeweils mit binärer Suche und Interpolationssuche. Welche Indizes des Felds werden der Reihe nach inspiziert?

b) Suchen Sie den nicht enthaltenen Schlüssel 21 mit binärer Suche. Welche Indizes des Felds werden der Reihe nach inspiziert, bis die Suche abbricht?

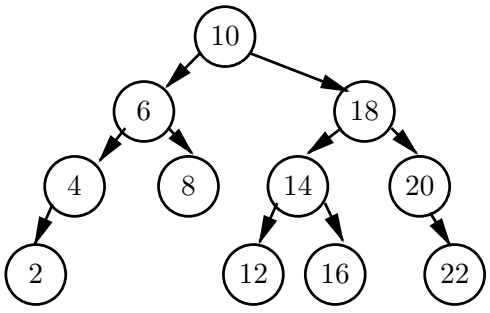
Aufgabe 3: Mastertheorem 2+1 Punkt

a) Welche asymptotische Laufzeit ergibt sich aus der Rekursionsgleichung $T(n) = 2 \cdot T(\frac{n}{4}) + \sqrt{n}$ für hinreichend große Werte n?

b) Warum ist das Mastertheorem der Vorlesung für die folgende Rekursionsgleichung nicht anwendbar?
$T(n) = 7 \cdot T(\frac{n}{3}) + \log_2 n$

Aufgabe 4: AVL-Bäume 10 Punkte

Führen Sie auf dem nebenstehenden AVL-Baum die folgenden Operationen in der angegebenen Reihenfolge durch: Einfügen (17), Löschen(22), Löschen(4), Einfügen(19), Einfügen(7) und Löschen(16). Geben Sie nach jeder Operation den entstehenden Baum mit seinen Balancefaktoren an.



Aufgabe 5: Bitonisches TSP 6 Punkte

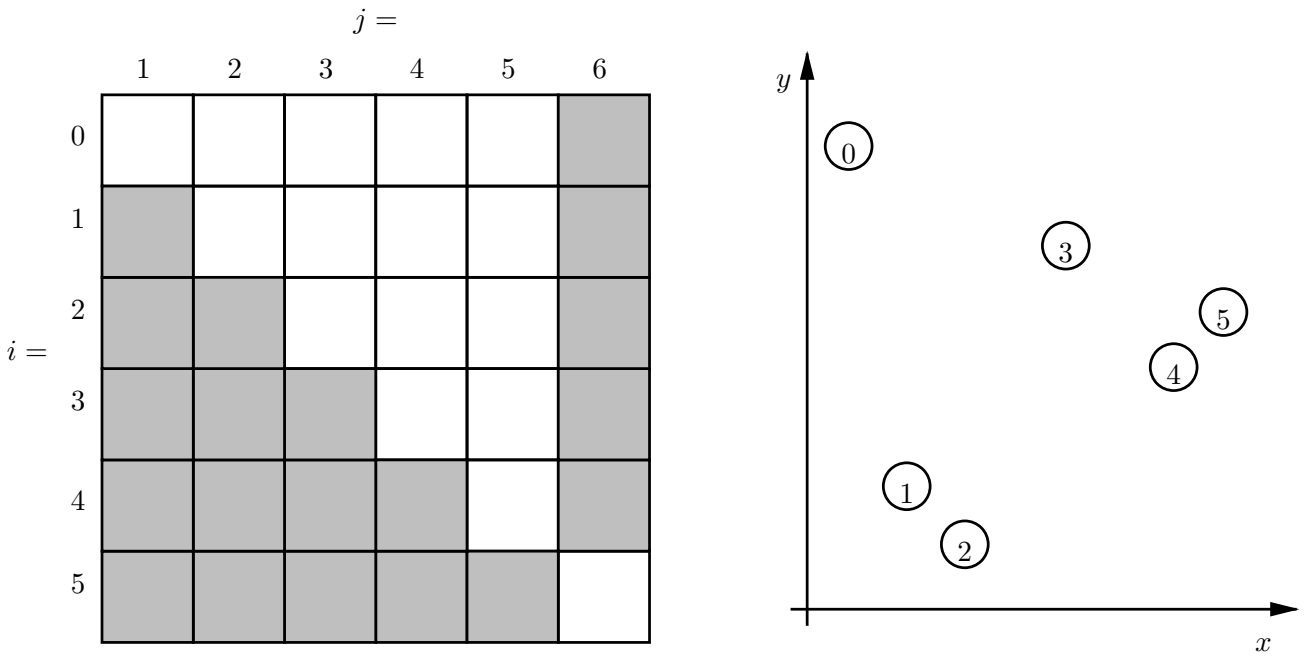
Gegeben Sei ein TSP-Problem mit den folgenden 6 Städten:

Nr.	0	1	2	3	4	5
X-Koordinate	1	2	3	5	8	9
Y-Koordinate	8	2	1	6	4	5

Entfernungstabelle

	1	2	3	4	5
0	6	7	4	8	9
1		2	5	6	8
2			5	6	7
3				4	4
4					1

Berechnen Sie per dynamischem Programmieren die kürzeste bitonische Rundreise und geben Sie diese als Permutation und graphisch an.

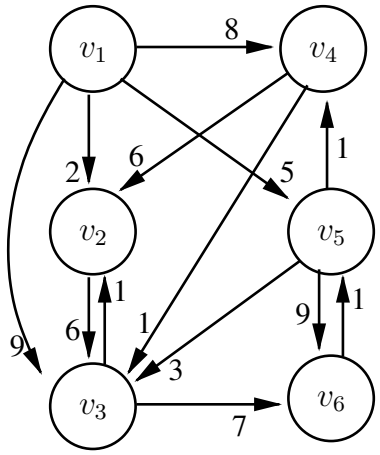


Aufgabe 6: Binärer Heap 3 Punkte

Ein Kommilitone behauptet: „Fünf Elemente mit den Prioritätswerten $\{1,2,\ldots,5\}$ können auf genau vier Arten in einem Min-Heap angeordnet sein!“ Richtig oder falsch? Begründen Sie Ihre Antwort stichhaltig.

Aufgabe 7: Kürzeste Wege 7 Punkte

Betrachten Sie den folgenden Graphen und ermitteln Sie die kürzesten Wege vom Knoten v_1 zu allen anderen Knoten mit dem Dijkstra-Algorithmus. Tragen Sie für alle Iterationen und jeden Knoten die aktuelle kürzeste Entfernung *d* sowie den aktuellen Vorgängerknoten *p* in die Tabelle ein. Markieren Sie den entstehenden Baum mit den kürzesten Wegen im Graphen.



Knoten											
v_1		v_2		v_3		v_4		v_5		v_6	
d	p	d	p	d	p	d	p	d	p	d	p
0	–	∞	–	∞	–	∞	–	∞	–	∞	–

Aufgabe 8: Dynamisches Programmieren 3 Punkte

Es sollen Zahlen gemäß der folgenden Rekursionsgleichung berechnet werden:

$$P_k = \begin{cases} k, & \text{falls } 0 \leq k \leq 2 \\ P_{k-3} + 2 \cdot P_{k-2} + P_{k-1}, & \text{falls } k > 2 \end{cases}$$

Formulieren Sie einen Algorithmus in Pseudo-Code für die Berechnung eines Werts P_n gemäß der Idee des dynamischen Programmierens. Welche asymptotische Laufzeit resultiert?

Aufgabe 9: Sortieren $5+5$ Punkte

Sortieren Sie das gegebene Feld aufsteigend mit den Sortierverfahren aus der Vorlesung.

a) Wenden Sie rekursives Mergesort an und geben Sie das Feld nach jedem Aufruf der Methode MISCHEN an. Markieren Sie das jeweils neu sortierte Teilfeld

5	7	9	3	2	6	8

b) Wenden Sie Quicksort an (Pivotelement steht rechts) und geben Sie das Feld nach jedem Partitionieren von Teilfeldern mit ≥ 2 Elementen an.

5	7	9	3	2	6	8

Aufgabe 10: Backtracking 3 Punkte

Ein Algorithmus soll alle dreielementigen Teilmengen der Menge $\{1,2,3,4,5\}$ erzeugen. Beschreiben Sie in Stichworten, wie Sie dies mit Backtracking umsetzen können. Illustrieren Sie dies mit einer aussagekräftigen (Teil-)Skizze des entstehenden Entscheidungsbaums.

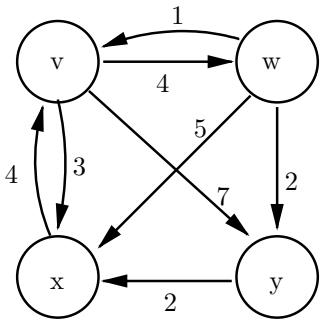
Aufgabe 11: Zusatzaufgabe: Tiefensuche (3 Punkte)

Geben Sie jeweils kleinstmögliche Beispiele an, die eine Baumkanten, eine Rückwärtskanten bzw. eine Vorwärtskanten enthalten.

B.2 Sommersemester 2015

Aufgabe 1: Floyd-Warshall-Algorithmus 8 Punkte

Berechnen Sie die kürzesten Wege für alle Knotenpaare mit dem Floyd-Warshall-Algorithmus.



Ausgangsmatrix

	v	w	x	y
v				
w				
x				
y				

über v

über w

über x

über y

Aufgabe 2: Sortieren 3 + 5 Punkte

Sortieren Sie das gegebene Feld aufsteigend mit den Sortierverfahren aus der Vorlesung.

a) Wenden Sie Insertionsort an und geben Sie das Feld jeweils nach dem Einfügen eines Elements an.

8	1	6	3	9	2	7

b) Wenden Sie Quicksort an (Pivotelement steht rechts) und geben Sie das Feld nach jedem Partitionieren von Teilfeldern mit ≥ 2 Elementen an.

8	1	6	3	9	2	7

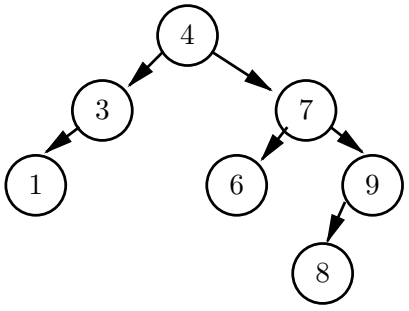
Aufgabe 3: Skip-Liste 5+3 Punkte

a) Fügen Sie in eine anfangs leere Skipliste nacheinander die folgenden Elemente ein: 3, 2, 7, 5, 1. Als Ebenen für diese Elemente werden der Reihe nach die folgenden Werte per Zufall bestimmt: 1, 2, 1, 3, 1. Zeichnen Sie die Liste nach jedem Einfügen.

b) Geben Sie an, mit wievielen Vergleichen das nicht vorhandene Element mit dem Schlüssel 4 gesucht wird.

Aufgabe 4: Suchbäume 3+6 Punkte

Auf dem nebenstehenden Baum sollen die folgenden Operationen in der angegebenen Reihenfolge durchgeführt werden: Löschen(3), Einfügen(5) und Löschen(7). Geben Sie nach jeder Operation den entstehenden Baum an.



a) Führen Sie zunächst die Operationen für den unbalancierten Baum durch.

b) Führen Sie jetzt die Operationen der AVL-Bäume auf dem Ausgangsbaum durch. Geben Sie nach jeder Operation die Balancefaktoren an.

Aufgabe 5: Mastertheorem 2+4 Punkte

a) Welche asymptotische Laufzeit ergibt sich aus der Rekursionsgleichung $T(n) = 9 \cdot T(\frac{n}{3}) + n^3$ für hinreichend große Werte n?

b) Betrachten Sie den folgenden Algorithmus, der das Feld in Viertel zerlegt.

```
SUCHE(Feld A, Index links, Index rechts)
    if rechts − links > 1
1   then ⌈ l → ⌊(rechts−links)/4⌋
2        m → ⌊(rechts−links)/2⌋
3        r → ⌊3 · (rechts−links)/4⌋
4        if ORAKEL(A)
5        then ⌈ return max{SUCHE(A, links, l), SUCHE(m, r)}
6             ⌊ else ⌈ return max{SUCHE(A, l, m), SUCHE(r, rechts)}
```

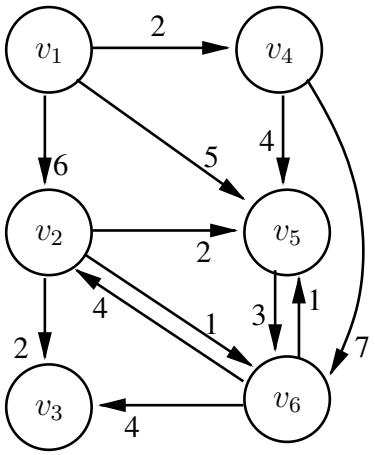
Stellen Sie eine Rekursionsgleichung auf unter der Annahme, dass ORAKEL in $\Theta(\sqrt{n})$ arbeitet. Welche asymptotische Laufzeit resultiert mit dem Mastertheorem?

Aufgabe 6: Sortiertes Feld 3 Punkte

Ein Komilitone behauptet: „In einem sortierten (dynamischen) Feld geht Suchen und Einfügen immer in logarithmischer Zeit.“ Richtig oder falsch? Begründen Sie Ihre Antwort stichhaltig.

Aufgabe 7: Kürzeste Wege 6 Punkte

Betrachten Sie den folgenden Graphen und ermitteln Sie die kürzesten Wege vom Knoten v_1 zu allen anderen Knoten mit dem Dijkstra-Algorithmus. Tragen Sie für alle Iterationen und jeden Knoten die aktuelle kürzeste Entfernung d sowie den aktuellen Vorgängerknoten p in die Tabelle ein. Markieren Sie den entstehenden Baum mit den kürzesten Wegen im Graphen.

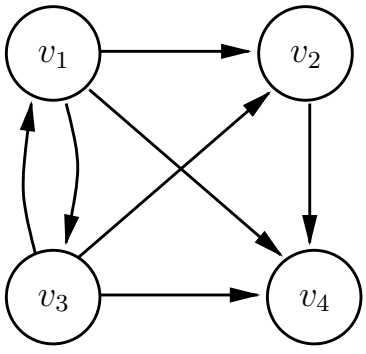


Knoten											
v_1		v_2		v_3		v_4		v_5		v_6	
d	p	d	p	d	p	d	p	d	p	d	p
0	–	∞	–	∞	–	∞	–	∞	–	∞	–

Aufgabe 8: Breitensuche 2 Punkte

Die Breitensuche benutzt eine Warteschlange, um sich Knoten zu merken. Geben Sie einen Graphen mit 6 Knoten an, für den bei der Durchführung der Breitensuche die Warteschlange gleichzeitig (an einem beliebigen Zeitpunkt) maximal viele Einträge enthält. Zeigen Sie konkret, wann bei der Durchführung der maximale Speicherplatz (für die Warteschlange) benötigt wird.

Aufgabe 9: Tiefensuche 6 Punkte



Führen Sie den Tiefensuchalgorithmus auf dem nebenstehenden Graphen aus (mit Startknoten v_1 und Berücksichtigung der Kanten in aufsteigender Reihenfolge bzgl. des Zielknotens). Tragen Sie die Zeitstempel im Graphen ein und ordnen Sie alle Kanten den entsprechenden Kantenklassen zu.

Baumkanten:
Vorwärtskanten:
Querkanten:
Rückwärtskanten:

Aufgabe 10: Hashing 4 Punkte

Gegeben sei eine anfangs leere Hashtabelle mit 11 Feldern, in die der Reihe nach die folgenden Schlüssel eingefügt werden sollen: 19, 8, 14, 36, 15, 4, 25, 30. Geben Sie die Belegung der Tabelle nach der letzten Einfügeoperation an und kennzeichnen Sie die Sondierungsfolge für jedes Element. Benutzen Sie die Hashfunktion $h(x) = x \bmod 11$ mit quadratischem Sondieren.

Position	0	1	2	3	4	5	6	7	8	9	10
Inhalt											

Aufgabe 11: Zusatzaufgabe: Verkettete Liste (3 Punkte)

Demonstrieren Sie schrittweise an einem kleinen Beispiel, wie durch Manipulation der Zeiger ein neues Element in eine sortierte verkettete Liste eingefügt wird. Gehen Sie dabei davon aus, dass die Einfügestelle bereits gefunden ist.