Algorithmen & Komplexität

Theoretische Informatik

Roman Wetenkamp

25. Juli 2021

Inhaltsverzeichnis

Vorwort

Die theoretische Informatik ist genau das Thema, vor dem ich vor Beginn meines Studiums am meisten Respekt hatte ... Und das ist es neben den anderen Mathematik-Vorlesungen bis heute: Theoreme, Kalküle und Formeln sind nun einmal nicht die Dinge, mit denen sich die Masse der Informatikstudierenden gerne befasst, mich eingeschlossen. Mit einer gerade eben bestandenen Logikklausur im ersten Semester sind dies Gründe genug dafür, ergänzendes, motivierendes und begreifbares Material zusammenzustellen, das mich einerseits akut durch die Klausur dieses Semesters bringt und hoffentlich auch für andere einen Nutzen hat. Geteiltes Leid ist halbes Leid!

Hier möchte ich den Vorlesungsstoff auf meine Art zusammenfassen und komplettieren, um Beispiele ergänzen und mit Aufgaben versehen, wie ich sie in der Vorlesung oder weiteren Büchern antraf. Ich persönlich lerne durch Aufgaben einfach am besten und für diejenigen, denen es auch so geht, gibt es davon hier viele. Meine Lösungen finden Sie im Anhang.

An dieser Stelle möchte ich noch **Jonathan Lelek** und **Vincent Degenhart** ausdrücklich danken, die mich auf Fehler hingewiesen und diese mit mir zusammen korrigiert haben.

Viel Erfolg!

Roman Wetenkamp
Mannheim, den 25. Juli 2021

Geschafft

Nun liegt die Klausur mir. Meine Strategie, mich durch das Schreiben dieses Skripts vorzubereiten, ist aufgegangen. Als Hinweis sei Ihnen mitgegeben, dass Sie Ihren Fokus auf die **Landau-Notation** und die einzelnen Such- und Sortieralgorithmen legen sollten. Die Wahrscheinlichkeit, dass Sie einzelne Algorithmen auf eine gegebene Datenmenge anwenden müssen, ist hoch. Überlegen und üben Sie auch im Vorfeld schon, wie Sie (Zwischen-) Ergebnisse notieren. Hier gibt es unterschiedliche Vorstellungen der Dozierenden. Ebenso sind die Datenstrukturen in den hinteren Kapiteln gut geeignet für Prüfungsaufgaben: Erwarten Sie Bäume, die Sie rotieren sollen und in die Sie Knoten bearbeiten. Die Aufgaben in diesem Skript liefern Ansätze für mögliche Prüfungsaufgaben.

Warnung Das Studium an einer Dualen Hochschule unterscheidet sich von dem Studium an Universitäten oder regulären Fachhochschulen insbesondere dadurch, dass aufgrund der Dualität von Theorie und Praxis meist nur die Hälfte der Zeit zur Vermittlung des Stoffes zur Verfügung steht (wenn dann auch intensiver). Daher gehen Sie bitte nicht davon aus, dass Sie dieses Skript ausreichend auf Klausuren in regulären Vollzeitstudiengängen vorbereitet!

Hinweis Dieses Dokument ist kein Vorlesungsmaterial, hat nicht den Anspruch auf Vollständigkeit und enthält mit Sicherheit Fehler. Desweiteren ist es noch lange nicht vollendet (es ist infrage zustellen, ob es das je sein wird), und doch möchte ich Sie ermutigen, beizutragen! Jegliche Fehler, Probleme oder Anmerkungen können Sie mir gerne über das dazugehörige GitHub-Repository unter der URL `https://github.com/RWetenkamp/algokomp` zukommen lassen. Danke!

Teil I.
Grundlagen der Algorithmik

1. Der Algorithmusbegriff

Wie der Titel dieses Moduls schon verrät, befassen wir uns mit Algorithmen: Jenen nebulösen Dingen, die uns mit ähnlich Denkenden und Interessierten in Social-Media-Plattformen zusammenbringen, komplexe mathematische Berechnungen ausführen und zunehmend mehr an Einfluss in unserem Leben gewinnen. Wir betrachten hier nun die Wortherkunft, definieren den Begriff anschließend und gehen auf Eigenschaften eines solchen ein.

Wortherkunft Der Begriff „Algorithmus“ besteht zum einen aus dem altgriechischen Wort *arithmos* – Zahl und zum anderen geht er zurück auf den persischen Mathematiker AL-CHARISMI (780-846 n. Chr.), dessen Werk *„Algorismus“* schon eine gewisse Nähe zum heute üblichen Begriff offenbart. [**brockhaus:algorithmus**]

Definition 1. *Unter einem Algorithmus verstehen wir eindeutige Verarbeitungsvorschriften zur Lösung eines Problems oder einer Problemklasse. Daran geknüpft ist der Gedanke des EVA-Prinzips: Eine Eingabe wird verarbeitet und auf Grundlage dessen wird eine Ausgabe erzeugt, ein Algorithmus ist hier für die Verarbeitung bis zur Ausgabe notwendig.*

Algorithmen finden in einer Vielzahl von Bereiches unseres Lebens Einsatz:

- Zur Untersuchung großer Datenmengen
- Zur Kommunikation/Suche im Internet
- In bildgebenden Verfahren oder diagnostischen Anwendungen der Medizin
- In Assistenzsystemen im Auto
- Bei der Partnersuche über Online-Dating-Plattformen

Damit finden wir intuitiv eine Begründung für die Untersuchung von Algorithmen.

Bemerkung 1. *Ein Algorithmus ist von einem Programm unbedingt abzugrenzen: Der Algorithmus bezeichnet ein abstraktes Konzept zur Lösung eines gegebenen Problems während ein Programm die konkrete Umsetzung eines Algorithmus in einer Programmiersprache ist. (Implementierung)*

Bei der Betrachtung von Algorithmen betrachten wir die folgenden drei Ziele:

- Korrektheit
- Effizienz
- Einfachheit

Einfachheit Aus wirtschaftlichen Gründen besteht ein Interesse daran, dass ein Algorithmus so einfach wie möglich ist, beispielsweise in Bezug auf Implementierungskosten. Ein weiterer Aspekt besteht in der Prüfung auf Korrektheit: Zu einem zu komplexen Algorithmus lässt sich nur schwer feststellen, ob dieser korrekt ist.

Korrektheit Die Korrektheit ist die zentrale Eigenschaft eines Algorithmus – Wenn ein Algorithmus nicht das leistet, was er vorgibt zu leisten oder was von ihm erwartet wird, ist er wertlos. Die Spezifikation beschreibt das exakte Verhalten unter Angabe von einer Vorbedingung (Zustand der Daten vor der Ausführung) und einer Nachbedingung (erwünschter Zustand nach der Ausführung).

Definition 2. *Ein Algorithmus heißt **partiell korrekt**, falls aus erfüllter Vorbedingung die Nachbedingung folgt.*
*Ein Algorithmus heißt **total korrekt**, falls er partiell korrekt ist und nach endlich vielen Schritten terminiert.*

Die Korrektheit eines Algorithmus kann manuell, per Implementierung oder mathematisch verifiziert werden, wobei letztere Variante häufig vorzuziehen ist.

Bemerkung 2. *Nicht jeder Algorithmus muss für jede Eingabe terminieren. Unter dem **Halteproblem** versteht man die Frage, ob ein Algorithmus nach endlich vielen Schritten terminiert. Diese Frage ist algorithmisch nicht entscheidbar, wie* Alan Turing *feststellte. Stattdessen muss die Termination für jeden Algorithmus einzeln entschieden werden.*

2. Effizienz / Komplexität

Weitaus komplexer als die Frage der Einfachheit oder Korrektheit ist jene der Komplexität selbst:

Definition 3. *Die Effizienz / Komplexität eines Algorithmus ist ein Maß für die Menge an Ressourcen (Rechenzeit oder Speicherbedarf), die er benötigt.*

Dabei ist zu beachten, dass es für ein Problem häufig mehrere Algorithmen gibt, die sich in ihrer Komplexität unterscheiden. Daher muss die Komplexität für jeden Algorithmus einzeln bestimmt werden. Dieses Skript befasst sich im Wesentlichen nur mit der Rechenzeit.

Für eine wirklich präzise Aussage über die Rechenzeit wäre es erforderlich, für jede Implementation eines Algorithmus in einer Programmiersprache einzeln festzustellen, welche Rechenzeit jede einzelne Operation auf dem spezifischen Prozessor des Testsystems benötigt. Da diese Aussagen mittlerweile aufgrund der Entwicklungen auf dem Prozessormarkt an Relevanz verlieren, verlagert sich die Betrachtung auf die Komplexität eines Algorithmus.

Die Komplexität eines Algorithmus kann folglich keine genaue Angabe in einer Zeiteinheit sein, die beispielsweise aussagt, wie schnell denn nun wirklich der größte gemeinsame Teiler von 443 und 123 mithilfe des euklidischen Algorithmus berechnet werden kann, sondern ist ein abstraktes Konzept, das Vergleiche zwischen Algorithmen ermöglichen soll:

- Abstraktion konstanter Faktoren
- Abstraktion unbedeutender Terme
- Wachstumsverhalten der Laufzeit bei Veränderung der Größe der Eingabe

Diese Abstraktion motiviert nun den folgenden Abschnitt.

3. Maß der Komplexität: O-Notation

Nach dem Mathematiker Edmund Landau definieren wir in Bezug auf das Wachstum von Funktionen eine „obere Schranke“ und nennen diese $O(f)$. Falls eine Funktion g nicht schneller wächst als f, so sagen wir, dass $g \in O(f)$. Eine ebenfalls übliche, jedoch irritierende Schreibweise ist $g = O(f)$.

Definition 4. *Sei* $\mathbb{R}_+ := \{x \in \mathbb{R} \mid x > 0\}$ *und* $\mathbb{R}_+^{\mathbb{N}} := \{f \mid f \text{ist eine Funktion der Form } f : \mathbb{N} \to \mathbb{R}_+\}$

Sei $g \in \mathbb{R}_+^{\mathbb{N}}$ *gegeben. Dann ist die Menge aller Funktionen, die höchstens so schnell wachsen wie* g *definiert als:*

$$O(g) := \{f \in \mathbb{R}_+^{\mathbb{N}} \mid \exists n_0 \in \mathbb{N} : \exists c \in \mathbb{R}_+ : \forall n \in \mathbb{N} : (n \geq n_0 \Rightarrow f(n) \leq c \cdot g(n))\}$$

Die O-Notation gibt folglich eine Menge vergleichbarer Funktionen an, die es ermöglicht, die Komplexität eines Algorithmus abzuschätzen.

Beispiel 1. *Behauptung:* $3n^3 + 2n^2 + 7 \in O(n^3)$

Beweis. Gesucht ist eine Konstante $c \in \mathbb{R}_+$ und eine Konstante $n_0 \in \mathbb{N}$, sodass für alle $n \in \mathbb{N}$ mit $n \geq n_0$ gilt:
$3n^3 + 2n^2 + 7 \leq c \cdot n^3$

Wähle $n_0 := 1$ und $c := 12$ [1]
Sei nun

$$\begin{aligned}
& 1 \leq n && (1)\\
(1)^3: \quad & 1 \leq n^3 && (2)\\
(2) \cdot 7: \quad & 7 \leq 7n^3 && (3)\\
(1) \cdot 2n^2: \quad & 2n^2 \leq 2n^3 && (4)\\
& 3n^3 \leq 3n^3 && (5)\\
(3) + (4) + (5): \quad & 3n^3 + 2n^2 + 7 \leq 12n^3 && (6)
\end{aligned}$$

Durch die Ungleichungen (3), (4) und (5) ist für jedes Polynom einzeln eine Abschätzung erfolgt. Addiert man diese Ungleichungen nun, ergibt sich (6). Damit ist gezeigt, dass die Behauptung wahr ist und die Funktion in $O(n^3)$ liegt. Aus der Definition der O-Notation ergibt sich, dass jede Funktion mit höchstem Grad 3 in $O(n^3)$ liegt, da Vorfaktoren und Konstantglieder entfallen. □

Die O-Notation eines Algorithmus lässt sich auch durch vollständige Induktion beweisen.

Beispiel 2. *Behauptung:* $n \in O(2^n)$

Beweis. Wähle $n_0 := 0, c := 1$. Zu zeigen: $n \leq 2^n \forall n \in \mathbb{N}$

Beweis durch Induktion nach n:
Induktionsanfang (IA): $n = 0 \quad n = 0 \leq 1 = 2^0 = 2^n$
Induktionsvoraussetzung (IV): $n \leq 2^n$ z.Z. $n + 1 \leq 2^{n+1}$

[1] Die Wahl von c und n erfolgt hier durch Ausprobieren – In diesem Fall ergibt sich 12 als Summe aus den Potenzfaktoren.

Induktionsschritt (IS): $n \mapsto n+1$
Per einfacher Induktion kann man nun zeigen:

$$n \leq 2^n \tag{7}$$

$$\text{Daraus folgt mit IV:} \quad n+1 \leq 2^n + 2^n = 2 * 2^n = 2^{n+1} \tag{8}$$

□

Die Eigenschaft $f \in O(g)$ induktiv zu zeigen, ist mühsam. Stattdessen können die folgenden Propositionen genutzt werden:

Bemerkung 3.

$$f \in O(f) \tag{9}$$

$$f \in O(g) \Rightarrow d \cdot f \in O(g) \tag{10}$$

$$f \in O(n) \wedge g \in O(n) \Rightarrow f + g \in O(n) \tag{11}$$

$$f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 \cdot f_2 \in O(g_1 \cdot g_2) \tag{12}$$

$$f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow \frac{f_1}{f_2} \in O(\frac{g_1}{g_2}) \tag{13}$$

$$f \in O(g) \wedge g \in O(n) \Rightarrow f \in O(n) \tag{14}$$

Darüber hinaus gilt folgender Satz:

Satz 1. *Seien $f, g : \mathbb{N} \to \mathbb{R}_+$. Dann gilt:*

$$f(n) \in O(g(n)) \iff (\frac{f(n)}{g(n)})_{n \in \mathbb{N}} \textit{ ist beschränkt}$$

Außerdem finden Grenzwertbetrachtungen Eingang:

Lemma 1.

$$f \in O(g) \textit{ und } g \in O(f), \textit{ wenn } \lim_{n \to \infty} (\frac{f(n)}{g(n)}) = c, c \neq 0 \tag{15}$$

$$f \in O(g) \textit{ und } g \notin O(f), \textit{ wenn } \lim_{n \to \infty} (\frac{f(n)}{g(n)}) = 0 \tag{16}$$

$$f \notin O(g) \textit{ und } g \in O(f), \textit{ wenn } \lim_{n \to \infty} (\frac{f(n)}{g(n)}) = \infty \tag{17}$$

Neben der O-Notation gibt es zwei weitere Notationen, die im Folgenden vorgestellt werden.

3.1. Ω-Notation

Neben der Menge aller Funktionen, die höchstens so schnell wächst wie die betrachtete Funktion, bezeichnen wir die Menge der Funktionen, die mindestens so schnell wächst wie die Funktion als $\Omega(n)$.

Definition 5. *Sei $g \in \mathbb{R}_+$. Die Menge aller Funktionen, die mindestens so schnell wachsen wie g ist:*

$$\Omega(n) := \{f \in \mathbb{R}_+^{\mathbb{N}} \mid \exists n_0 \in \mathbb{N} : \exists c \in \mathbb{R}_+ : \forall n \in \mathbb{N} : (n \geq n_0 \Rightarrow f(n) \geq c \cdot g(n))\}$$

Diese Notation fristet ein Schattendasein: In der Regel ist die O-Notation geeigneter, da eine Orientierung am „Worst-Case-Szenario“ üblicher ist.

3.2. Θ-Notation

Aus O-Notation und Ω-Notation ergibt sich nun die Menge der Funktionen, die genau so schnell wachsen wie die betrachtete Funktion. Wir bezeichnen sie mit Θ.

Definition 6.

$$\Theta(g) = O(g) \cap \Omega(g)$$

3.3. Zusammenfassung

- Die O-Notation umfasst alle Funktionen, die nicht schneller wachsen als f. Damit beschreiben wir eine obere Schranke für die Komplexitätsfunktion.
- Die Ω-Notation umfasst alle Funktionen, die mindestens so schnell wachsen wie f. Dies ist eine untere Schranke.
- Die Θ-Notation umfasst alle Funktionen, die genau so schnell wachsen wie f. Damit erhalten wir ein asymptotisches Maß.

Lemma 2.

- $f \in O(g)$ *und* $g \in O(f)$, *also* $f \in O(g)$ *und* $f \in \Omega(g)$, *wenn* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c, c \neq 0$, *also:* $f \in \Theta(g)$
- $f \in O(g)$ *und* $g \notin O(f)$, *wenn* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, *also:* $g \in \Omega(f)$ *und* $f \notin \Omega(g)$
- $f \notin O(g)$ *und* $g \in O(f)$, *also* $g \notin \Omega(f)$ *und* $f \in \Omega(g)$, *wenn* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

4. Rekursionsgleichungen

Nun haben wir mit den Landau-Symbolen einen mathematischen Weg gefunden, die Komplexität von Algorithmen zu beschreiben. Im Folgenden wenden wir dieses Modell auf konkrete Algorithmen an.

Bekannte Algorithmen sind beispielsweise Such- oder Sortierverfahren. Diese sind häufig rekursiv, d.h. eine Iteration verarbeitet direkt das Ergebnis einer vorherigen. Dadurch ergeben sich nun sogenannte **Rekurrenzgleichungen** oder auch **Rekursionsgleichungen**.

Beispiel 3. $T(n) = 2 \cdot T(n-1) + 1$ *ist eine Rekurrenzgleichung, da der Funktionswert* n *direkt vom Funktionswert* $n-1$ *abhängt. Aus den Programmierkonzepten ist uns Rekursion als ein „Wiederaufruf von sich selbst“ bekannt, dieses Muster findet sich hier in unseren betrachteten Algorithmen.*

Um die Komplexität eines rekursiven Algorithmus bestimmen zu können, bedienen wir uns je nach Rekursionsgleichung unterschiedlichen Verfahren. Eines davon ist das sogenannte Mastertheorem.

4.1. Mastertheorem

Satz 2. *Sofern alle Teilprobleme die gleiche Größe haben, können wir zwei Fälle unterscheiden:*

- *Basisfall:* $f(n)$ *ist konstant für hinreichend kleine Werte von* n

- *Für größere n lässt sich eine Rekursionsgleichung der folgenden Form bestimmen:*

$$f(n) = a \cdot f(\frac{n}{b}) + O(n^d)$$

a ist hier die Anzahl der rekursiven Aufrufe, b der Verkleinerungsfaktor des Problems und d der Exponent für die Laufzeit der Vorbereitung der Rekursion und/oder die Zusammensetzung der Teilprobleme.

Nun gilt Folgendes:

$$f(n) \in \begin{cases} O(n^d) & falls\ a < b^d \\ O(n^d \cdot log_b(n)) & falls\ a = b^d \\ O(n^{log_b(a)}) & falls\ a > b^d \end{cases}$$

Mithilfe dieses Theorems kann nun die Komplexität eines rekursiven Algorithmus bestimmt werden:

Beispiel 4. *Gegegeben sei*

$$f(n) = 2 \cdot f(\frac{n}{2}) + O(n^2)$$

Daraus folgt $a = 2, b = 2, d = 2$.

Da $2 < 2^2$ *tritt der erste Fall ein und wir halten fest:* $f(n) \in O(n^2)$.

Über das Mastertheorem sind eine Zahl an Rekursionsgleichungen entscheidbar, jedoch nicht jede. Für die Rekursionsgleichungen, deren Form nicht der für das Mastertheorem benötigten entspricht, werden andere Verfahren nötig.

4.2. Auflösungsverfahren

Eine gegebene Rekursionsgleichung lässt sich in eine geschlossene Form überführen, in der keine Abhängigkeit zu vorherigen Funktionswerten mehr besteht. Dafür existieren zwei Ansätze:

- **bottom-up:** Aus berechneten Funktionswerten für kleine n wird eine weitere Berechnungsvorschrift ermittelt. Diese muss anschließend induktiv bewiesen werden.
- **top-down:** Die vorherigen Funktionswerte werden solange durch die entsprechende Gleichung ersetzt, bis der definierte Startwert eingesetzt werden kann. Anschließend wird der Term zusammengefasst und es ergibt sich eine geschlossene Rechenvorschrift.

Beispiel 5. *Gegeben ist die folgende Rekursionsgleichung:* $f(1) = 1, f(n) = 2 \cdot f(n-1) + 1$.

bottom-up

$$f(1) = 1 \tag{18}$$
$$f(2) = 2 \cdot 1 + 1 = 3 \tag{19}$$
$$f(3) = 2 \cdot 3 + 1 = 7 \tag{20}$$
$$f(4) = 2 \cdot 7 + 1 = 15 \tag{21}$$

Aus den Ergebnissen wird relativ schnell eine Verwandtschaft zu den ersten Potenzen der Zahl 2 deutlich und so lässt sich nun folgende Gleichung aufstellen:

$$f(n) = 2^n - 1$$

Diese Gleichung kann nun für die gegebenen Werte von n überprüft werden und muss anschließend (induktiv) bewiesen werden.

Beweis. Zu zeigen: $f(n) = 2 \cdot f(n-1) + 1 \in O(2^n)$

$$\begin{aligned}
\text{Induktionshypothese:} \quad & f(n) = 2^n - 1 \quad \forall n \geq 1 \\
\text{Induktionsanfang:} \quad & f(1) = 2^1 - 1 = 1 \\
\text{Induktionsvoraussetzung:} \quad & n \to n+1 \quad f(n+1) = 2^{n+1} - 1 \\
\text{Induktionsschritt:} \quad & f(n+1) = 2 \cdot f(n) + 1 = 2 \cdot (2^n - 1) + 1 = 2^{n+1} - 1
\end{aligned}$$

nach [**UniHH**] □

Wir führen Induktionsbeweise für Rekurrenzgleichungen, indem wir zunächst im Induktionsanfang feststellen, dass unsere ermittelte Funktion den Rekursionsstartwert erfüllt. Ist dies nicht der Fall, brechen wir hier ab. Nun induzieren wir, in dem wir n auf $n+1$ abbilden, bilden die Rekursionsgleichung für $n+1$ und setzen nun für $f(n)$ unsere geschlossene Funktion ein. Wir stellen fest, dass unsere Induktionsvoraussetzung erfüllt ist. Damit ist unsere Auflösung der Rekursionsgleichung gezeigt.

top-down

$$\begin{aligned}
f(n) = \quad & 2 \cdot f(n-1) + 1 && (22)\\
= \quad & 2 \cdot (2 \cdot f(n-2) + 1) + 1 && (23)\\
= \quad & 2 \cdot (2 \cdot (... \cdot (2 \cdot 1) + 1)...) + 1) + 1 && (24)\\
\Rightarrow \quad & 2^{n-1} + 2^{n-2} + ... + 2^{n-(n-1)} + 1 && (25)\\
= \quad & \sum_{i=0}^{n-1} 2^i = \frac{2^{(n-1)+1} - 1}{2 - 1} = 2^n - 1 \in O(2^n) && (26)
\end{aligned}$$

Je nach Rekursionsgleichung bietet sich mitunter eines der beiden Verfahren mehr an als das andere. Vernachlässigt werden darf beim *bottum-up*-Verfahren nicht, dass ein Induktionsbeweis erforderlich ist! Bei Rekursionsgleichungen mit zweifacher Rekursion (in Beziehung zu den vorherigen zwei Funktionswerten) ist ein *top-down*-Verfahren deutlich komplexer und häufig nicht ratsam.

Aufgaben

Aufgabe 1 Veranschaulichen Sie die O-, Θ- und Ω-Notation anhand der Funktion $f(n) = 3n^3 + 7n^2 + 16$ grafisch.

Lösung auf Seite 65

Aufgabe 2 Geben Sie zu folgenden Funktionen jeweils die O-Notation an.

1. $f(n) = 6n^4 + 3 \cdot log_2(n)$
2. $f(n) = 6627816n + 13$
3. $f(n) = \frac{72n^3+27n^2+8n+9}{n!}$
4. $f(n) = 3n \cdot \frac{n!}{2} + n^n$

Lösung auf Seite 65

Aufgabe 3

1. Zeigen Sie, dass $n^2 \in O(2^n)$.
2. Zeigen Sie, dass $n^3 \in O(2^n)$.

Lösung auf Seite 66
entnommen aus VL

Aufgabe 4 Bestimmen Sie unter Anwendung des Mastertheorems die jeweilige Komplexitätsklasse $O(n)$.

- $f(n) = 2 \cdot f(\frac{n}{2n+1}) + n^2$
- $g(n) = log(n) \cdot g(\frac{n}{2}) + 3$
- $h(n) = sin(n) \cdot h(\frac{n}{\frac{n}{2}}) + O(n^3)$

Lösung auf Seite 66

Hinweis Für einige Aufgaben kann es hilfreich sein, die Tabelle der Komplexitätsklassen auf Seite 15 zu verwenden. Im Rahmen der Klausurvorbereitung sollten Sie außerdem die Komplexität von Algorithmen aus Pseudocode heraus angeben können.

5. Bestimmung von Komplexitäten

Nun haben wir in ausreichender Tiefe mathematisch-formal die O-Notation als Maß der Komplexität eingeführt. In der Praxis – und insbesondere auch in Klausuren – ist davon auszugehen, dass Sie einen Algorithmus in einer Programmiersprache implementiert oder als Pseudocode erhalten, zudem Sie dann aus dem Code heraus die Komplexität bestimmen sollen.

Dies wollen wir hier näher erläutern und üben.

5.1. Rechengesetze der O-Notation

Satz 3. *Seien $d \in \mathbb{R}_{\geq 0}$ und $f, g, h : \mathbb{N}_0 \to \mathbb{R}_{\geq 0}$, dann gilt:*

$$d = O(1) \tag{27}$$
$$f = O(f) \tag{28}$$
$$d \cdot f = O(f) \tag{29}$$
$$f + g = O(g), \text{ falls } f = O(g) \tag{30}$$
$$f + g = O(max\{f, g\}) \tag{31}$$
$$f \cdot g = O(f \cdot h), \text{ falls } g = O(h) \tag{32}$$

*Quelle: [***AlgInfTheo***]*

Mithilfe dieser Rechengesetze können wir O-Notationen leichter bestimmen. Sie sind die Grundlage für Vereinfachungen, die Vergleiche der Komplexität im Ende ermöglichen.

5.2. Komplexitätsklassen

Betrachten wir nun allgemein die häufigsten Komplexitätsklassen, mit denen sich bereits viele Algorithmen vergleichen lassen. Dabei sind jeweils Klassen angegeben, d.h. beispielsweise n^4 läge zwischen n^3 und 2^n und fällt hier in die Klasse der *kubischen* Komplexitäten. Aus dieser Tabelle und den vorangegangenen Rechengesetzen lässt sich nun beispielsweise auch die Komplexität einer zusammengesetzten Funktion wie beispielsweise $f(n) = n^3 + n!$ bestimmen. Da $n!$ um ein wesentliches schneller wächst als n^3, fällt f in die Komplexitätsklasse $n!$. Nun ergibt sich die Frage, wie wir einen gegebenen Algorithmus nun einer

Klasse	Bezeichnung	Rang
$O(1)$	konstant	exzellent
$O(log_x(n)$	logarithmisch	sehr gut
$O(n)$	linear	gut
$O(n \cdot log(n))$	überlinear	gut
$O(n^2)$	quadratisch	eher ineffizient
$O(n^3)$	kubisch	ineffizient
$O(2^n)$	exponentiell	sehr ineffizient
$O(n!)$	faktoriell	extrem ineffizient

Tabelle 1: Komplexitätsklassen nach [**Grundkurs**]

dieser Klassen zuordnen, ohne, dass bereits vorher eine Funktionsgleichung gegeben ist. Dafür können wir diese Tabelle nutzen, die aufschlüsselt, wo die jeweilige Komplexität in Algorithmen vorkommt.

Klasse	Typischer Aufbau eines Algorithmus dieser Komplexität
$O(1)$	die meisten Anweisungen werden nur einmal oder konstant oft ausgeführt (unabhängig von Eingaben)
$O(log(n)$	Lösen eine Problems durch Aufteilen in kleinere Teilprobleme, dabei wird die Laufzeit um einen konstanten Anteil verringert
$O(n)$	Falls n Eingabedaten verarbeitet werden müssen, ist dies der optimale Fall, da jedes Element nur einmal bearbeitet wird
$O(n \cdot log(n))$	Lösen eines Problems durch Aufteilen in kleinere Probleme, die unabhängig voneinander gelöst und abschließend rekombiniert werden
$O(n^2)$	typisch für die paarweise Verarbeitung von n Elementen $\rightarrow$ zwei verschachtelte For-Schleifen
$O(n^3)$	drei verschachtelte For-Schleifen
$O(a^n)$	Brute-Force-Lösungen (Ausprobieren aller Optionen)

Tabelle 2: Zuordnung von Algorithmen zu Verarbeitungsschritten nach [**Grundkurs**]

5.3. Komplexität von Programmierkonzepten

Eine einzelne Anweisung hat immer die Komplexität $O(1)$. Auch mehrere Anweisungen – deren Anzahl jedoch nicht von der Menge der Eingabedaten n abhängt – verändern die Komplexität nicht.

Schleifen Sofern jedoch eine Schleife in Abhängigkeit von n (auch hier müssen nicht *exakt* n Schleifendurchläufe passieren, es genügt eine generelle Abhängigkeit!) durchlaufen wird, erhalten wir einen Aufwand von $O(n)$ mit n = Anzahl der Daten.

Betrachten wir nun ineinander verschachtelte Schleifen, deren Durchlaufanzahl jeweils von n abhängt, so multiplizieren sich die Komplexitäten und wir erhalten beispielsweise für zwei ineinander verschachtelte For-Schleifen eine Komplexität von $O(n^2)$. Nach diesem Muster gehen wir für beliebig häufig ineinander verschachtelte Schleifen vor.

Beispiel 6. *Gegeben sei folgender Algorithmus:*

```
public void validateInput(int[] data) {
        int i = 0;
        for (dataElement in data) {
                calculateSomething(dataElement);
                while (i < 10) {
                        pingHost('localhost:3000/');
                        i++;
                }
        }
}
```

Wir betrachten zur Bestimmung der Komplexität:

- `int i = 0` *ist eine einzelne Anweisung ohne Abhängigkeit zu n, hat also die Komplexität* $O(1)$.
- *Nun beginnt eine For-Schleife mit n(= Anzahl der Feldelemente) Iterationen, also mit Komplexität* $O(n)$.

- *Innerhalb der For-Schleife findet pro Iteration ein Aufruf der Funktion* `calculateSomething` *statt, deren Komplexität in derer der Schleife bereits inbegriffen ist (da es sich um einen einzelnen Aufruf pro Iteration handelt).*
- *Nun beginnt eine weitere Schleife – deren Iterationsanzahl jedoch von n unabhängig ist und konstant 10 beträgt. Auch der Inhalt der Schleifen ist konstant und fällt daher nicht ins Gewicht. Daher beträgt die Komplexität hier erneut* $O(1)$*. Da wir Klassen von Komplexität betrachten, ist* $O(10)$ *zwar nicht falsch, jedoch nicht sinnvoll, weil so die Vergleichbarkeit entfällt/erschwert wird.*
- *Auch* `i++` *ist eine Anweisung mit konstantem Zeitbedarf, also* $O(1)$*.*

Nun addieren wir die einzelnen Komplexitäten und erhalten $O(n)$*, da konstante Terme entfallen.*

Satz 4. *Eine Schleife, in der die Datenmenge in jeder Iteration halbiert wird, hat eine Komplexität von* $O(log(n))$*.*

Neben Schleifen, deren Komplexitätsbetrachtung wir nun in aller Ausführlichkeit ausgeführt haben, widmen wir uns nun einem weiteren, in Algorithmen sehr häufig anzutreffenden Programmierkonzept:

Rekursion Die Komplexität von rekursiven Algorithmen zu bestimmen ist im Allgemeinen aufwendiger. Wir betrachten verschiedene Fälle:

- Falls in jedem Schritt die betrachtete Datenmenge um eins kürzer wird und die zusätzlich ausgeführte Operation einen konstanten Zeitbedarf hat, so liegt die Komplexität in $O(n)$.
- Ist dieser Zufallsaufwand nun nicht konstant sondern linear (verändert sich also in Abhängigkeit von n, so verändert sich der Aufwand und wir erhalten eine Komplexität von $O(\frac{n^2}{2})$.

An dieser Stelle sei auch noch einmal auf das **Master-Theorem** verwiesen, das bei Rekursionsgleichungen Anwendung findet, die gemäß „Divide and Conquer“ (Teile und Herrsche) Probleme in kleinere Teilprobleme zerteilen, diese getrennt voneinander verarbeiten und anschließend wieder zusammensetzen. (→ Mergesort, S. 30)

Aufgaben

- Gegeben sind die folgenden Pseudocode-Darstellungen verschiedener Algorithmen.
- Notieren Sie zu jeder Anweisung die entsprechende Komplexität.
- Geben Sie die Gesamtkomplexitätsklasse des Algorithmus in Landau-Symbolen an.

Verwenden Sie falls erforderlich das Master-Theorem.

Aufgabe 1

```
public int[] giveEvenNumbers(int[] data) {
        int[] result;
        for (dataElement in data) {
                if (dataElement % 2 == 0) {
                        result.append(dataElement);
                }
        }
        return result;
}
```

Lösung auf Seite 67

Aufgabe 2

```
for (int i = 1; i < n; i = 2 * i) {
        for (int j = n; j > 0; j = j / 2) {
                for (int k = 1; k < n; k = k + 1) {
                        // some constant procedures ...
                }
        }
}
```

Lösung auf Seite 67
entnommen aus [**GrUeb**]

Aufgabe 3

```
float mittel(float a[], int start, int end) {
        int n = end - start + 1;
        if (n == 1) {
                return a[start];
        }
        else {
                int mitte = n / 2 - 1;
                float m1 = mittel(a, start, start+mitte);
                float m2 = mittel(a, start + mitte + 1, end);
                float m = (     m1 * (mitte + 1) +
                m2 * (end - (start + mitte + 1) + 1)) / n;
                return m;
        }
}
```

Lösung auf Seite 67
entnommen aus [**GrUeb**]

Teil II.
Suchalgorithmen

Sie studieren Informatik, sind es satt, noch mehr Stunden vor Bildschirmen zu verbringen und suchen sich deshalb ein echtes, physisches Buch aus der Bibliothek Ihrer Hochschule heraus. Sie fragten nach „Wirth: Algorithmen und Datenstrukturen“ von 1983, weil man es Ihnen in Ihrer Vorlesung empfahl. Die Bibliothekarin deutet auf ein Regal und lässt sie damit allein.

Wie finden Sie das gesuchte Buch in der Regalreihe mit etwa 300 Büchern?

6. Lineare Suche

Sie beginnen ganz links. Nun lesen Sie den Titel des ersten Buches, er passt nicht. Sie lesen den Titel des zweiten Buches, er passt ebenfalls nicht. Sie fahren fort, iterieren über das gesamte Bücherregal und brechen ab, falls Sie es gefunden haben oder am Ende angelangt sind. Dieses Verfahren ist die **Lineare Suche**.

Idee Durchlaufe sukzessive das Feld A, bis das gesuchte Element gefunden ist bzw. das Ende des Feldes erreicht ist.

```
LinearSearch(A[], E) {
        i = 0;
        While i < A[].length {
                If (A[i] == E) {
                        return i;
                }
                i = i + 1;
        }
        Return ElementNotFound
}
```

Komplexität Ganz offensichtlich ist der Aufwand dieses Suchverfahrens durch die Feldlänge (also die Anzahl der Bücher in unserem Regal) bestimmt.
günstigster Fall Das gewünschte Element steht an Indexposition 0 (also dem ersten Element bzw. es ist das erste Buch im Regal) → ein Vergleich ist erforderlich
durchschnittlicher Fall Das gewünschte Element steht an der mittleren Indexposition $\frac{n}{2}$ bei n Feldelementen. → $\frac{n}{2}$ Vergleiche sind erforderlich
ungünstigster Fall Das gewünschte Element ist nicht im Feld vorhanden. → n Vergleiche

Insgesamt ergibt sich – da wir bei der Angabe der Komplexität durch die O-Notation jeweils den ungünstigsten Fall betrachten – für die Lineare Suche eine Komplexität von $O(n)$.

7. Binäre Suche

Sie befinden sich in der perfekten Bibliothek: Alle Bücher sind aufsteigend nach ihren Autor*innen sortiert! Dieses Wissen können Sie sich zunutze machen, in dem Sie ihr Wunschbuch mit der **binären Suche** finden: Sie teilen die Bücherreihe in zwei Hälften und betrachten nun das mittlere Buch: Muss Ihr Buch links oder rechts von dem mittleren Buch stehen (Vergleich anhand des Namens der Autoren)? Oder ist es sogar ihr Buch? Nein, das ist es nicht und der Autor Ihres Buches hat einen im Alphabet weiter hinten stehenden Anfangsbuchstaben, also betrachten Sie die Hälfte rechts des mittleren Buches. Diese Hälfte teilen Sie nun erneut, betrachten die neue Mitte und fahren fort, bis ihre Hälften aus einem Buch bestehen.

Idee Vergleiche zu suchendes Element E mit der Mitte des Suchbereiches $A[m]$. Falls gefunden: Abbruch. Falls $E < A[m]$: Suche weiter in linker Hälfte. Falls $E > A[m]$: Suche weiter in rechter Hälfte. Das Weitersuchen erfolgt durch einen rekursiven Aufruf des Algorithmus mit der entsprechenden Hälfte des Feldes.

```
BinarySearch(A[], E, indexLeft, indexRight) {
        if (indexLeft > indexRight) {
                return ElementNotFound
        }
        else {
                mid = (indexLeft+indexRight) / 2;
                if (E == A[mid]) {
                        return mid;
                }
                else if (E < A[mid] {
                        return BinarySearch(A[], E, indexLeft, mid-1);
                }
                else {
                        return BinarySearch(A[], E, mid+1, indexRight);
                }
        }
}
```

Die Parameter `indexLeft` und `indexRight` schränken jeweils das zu betrachtende Feld ein. Zu Beginn ist `indexLeft` mit 0 und `indexRight` mit `A[].length()` initialisiert.

Komplexität Hier handelt es sich um ein rekursives Verfahren, folglich werden wir eine Rekursionsgleichung erhalten und diese mit den bekannten Verfahren auswerten. Doch zunächst betrachten wir wieder Fälle:
günstigster Fall: E ist das mittlere Element $\rightarrow$ 1 Vergleich
ungünstigster Fall: E ist nicht im Feld
Die Rekursionsgleichung mit den Parametern a, b und d ergibt sich nun wie folgt:

- Es gibt einen rekursiven Aufruf, d.h. $a = 1$.
- Pro Rekursionsaufruf halbieren wir das Feld, daraus folgt $b = 2$.
- Die Vorbereitung der Rekursion verläuft konstant und ist minimal, also $d = 1$.

Die Rekursionsgleichung lautet nun:

$$f(n) = 1 \cdot f(\frac{n}{2}) + 1$$

Für die Anwendung des Mastertheorems formen wir um:

$$\iff f(n) = 1 \cdot f(\frac{n}{2}) + O(n^0)$$

Nun ergibt sich:

$$a = b^d \iff 1 = 2^0 \Rightarrow f(n) \in O(n^0 \cdot log_2(n)) = O(log(n))$$

8. Textsuche

Spätestens jetzt sollten Sie das Buch gefunden haben, wenn Sie die binäre Suche verwendet haben, vermutlich schneller als mit der linearen Suche. Jetzt haben Sie einen Arbeitsplatz gefunden, setzen sich und schlagen das Buch auf: Für ein Lesen in Ruhe haben Sie keine Zeit, daher suchen Sie sich die interessanten Teile heraus. Sie suchen, wo Niklaus Wirth Ihnen die O-Notation erklären wird.

Formal suchen Sie folglich alle Vorkommen eines Musters (in Form einer Zeichenkette) in einem Text. Dafür betrachten wir zwei Verfahren.

8.1. Einfache Textsuche

Analog zur linearen Suche beginnen Sie einfach, Ihren gesuchten Begriff unter den Text zu legen. Stimmt der erste Buchstabe überein, vergleichen Sie den zweiten. Stellen Sie eine Differenz fest, verschieben Sie Ihr Suchwort um eine Position nach rechts. Derart fahren Sie fort, bis Sie am Ende des Textes angelangt sind.

Idee Beginnend beim ersten Zeichen des Textes legt man das Muster der Reihe nach an jede Stelle des Textes an und vergleicht zeichenweise von links nach rechts, ob eine Übereinstimmung vorliegt

```
int BruteForceSearch(char[] text, char[] pattern) {
        int n = text.length;
        int m = pattern.length;
        int i = 0;
        int j = 0;
        for (int k = 0; i <= n-m; i++) {
                j = 0;
                while (j < m && Text[i+j] == pattern[j]) {
                        j++;
                }
                if (j == m) {
                        return i;
                }
        }
        return ElementNotFound;
}
```

Komplexität Offenbar hängt der Aufwand dieses Verfahrens ganz wesentlich davon ab, wie lang das Muster und der Text in Gänze sind. Weniger relevant hingegen ist der Aufbau des Textes, da alle Vorkommen des Musters gesucht werden. Insofern ist es nicht entscheidend, an welcher Stelle das Muster das erste Mal auftritt.

günstigster Fall Im günstigsten Fall werden m Vergleiche benötigt, wenn nämlich die Länge des Textes kleiner als das doppelte der Länge des Musters m ist und das Muster direkt am Beginn des Textes steht. $\rightarrow m$ Vergleiche

ungünstigster Fall Im ungünstigsten Fall werden $(n - m + 1) \cdot m$ Vergleiche benötigt, wobei dieser Fall allgemeingültig ist – es werden alle Vorkommen des Musters im Text gesucht und nicht lediglich das erste.

Die Laufzeit des Algorithmus liegt nun in $O(n \cdot m)$.

8.2. Knuth-Morris-Pratt-Algorithmus (KMP)

Ihnen ist das zu mühsam geworden, Zeichen für Zeichen des Textes und des Musters zu vergleichen. Sie sehnen sich nach einem intelligenteren, einfacheren Weg. Und, wahrlich den gibt es: Der **Knuth-Morris-Pratt**-Suchalogorithmus nach Donald Ervin Knuth, James Hiram Morris und Vaughan Ronald Pratt.

Die Grundlage für diesen Algorithmus bildet die vorangegangene einfache Textsuche. Diese optimieren wir nun, in dem wir beim ersten sich unterscheidenden Zeichen von Text und Muster das Muster nicht um ein einziges Zeichen nach rechts verschieben, sondern um eine bestimmte Anzahl an Zeichen, die sich aus unserem Muster ergibt und als **Rand** bezeichnet wird.

Definition 7. *Der **Rand** eines Musters M mit der Länge m ist eine tabellarische Auflistung aller Teilzeichenketten der Längen* 0 *bis m des Musters, für die jeweils die Anzahl der Zeichen eines Prefixes, das zugleich Postfix ist, bestimmt wird.*

Beispiel 7. *Betrachten wir das Wort* `abbababbab`.
Wir beginnen eine tabellarische Auflistung:

Teilzeichenkette	*Länge*	*Rand*	*Bemerkung*
	0	*-1*	*per Definition, für Teilschritt 2 (Suche) erforderlich*
`a`	*1*	*0*	*per Definition*
`ab`	*2*	*0*	*$a \neq b$*
`abb`	*3*	*0*	
`abba`	*4*	*1*	*gleich: a, Länge: 1*
`abbab`	*5*	*2*	*gleich: ab, Länge: 2*
`abbaba`	*6*	*1*	*gleich: a, Länge: 1*
`abbabab`	*7*	*2*	*gleich: ab, Länge: 2*
`abbababb`	*8*	*3*	*gleich: abb, Länge: 3*
`abbababba`	*9*	*4*	*gleich: abba, Länge: 4*
`abbababbab`	*10*	*5*	*gleich: abbab, Länge: 5*

Im folgenden Suchalgorithmus wird jeweils der Rand verwendet, um Verschiebungspositionen zu bestimmen. Daher ist die gesamte Tabelle erforderlich.

Implementierung als Pseudo-Code

```
Rand[0] = -1;                /* Definition */
Rand[1] = 0;                 /* Definition */

for (j = 2; j <= m; j++) {
        while ((i >= 0) and (M[i] != M[j-1])) {
                i = Rand[i];
                i++;
        }
        Rand[j] = i;
}
```

Folglich iterieren wir nun über alle möglichen Teilzeichenkettenlängen (`j`) und vergleichen jeweils mit dem vorherigen Rand (durch `i`). Wenn unser Rand maximal ist, verlassen wir die `while`-Schleife und speichern den ermittelten Rand.

Eine Veranschaulichung der algorithmischen Rand-Bestimmung finden Sie in der Wikipedia. [**KMPRand**]

Der Suchalgorithmus Ähnlich wie bei der einfachen Textsuche beginnen Sie, indem Sie das zu suchende Muster unter den Suchraum schreiben und vergleichen Zeichen für Zeichen. Entscheidend ist nun die erste Abweichung: Sie bestimmen den `Rand[]` der Indexposition der Abweichung im Muster und verschieben ihr Muster nun mit unterer Formel. So nutzen Sie aus, dass bis zur ersten Abweichung Gleichheit bestand – falls bei vier gleichen Zeichen ein Rand von zwei vorliegt, können Sie ihr Muster so verschieben, dass der Beginn nun dort liegt, wo zuvor das mit dem neuen Anfang identische Enge lag. Nun fahren Sie mit den neu gewonnen Index-Positionen fort, die sich aus dem Verschiebung ergeben haben.

Die Formel für Verschiebungen ist:
`Neuer Beginn des Musters (neue Suchtextposition) = Suchtextposition (aktueller Beginn des Musters) + (Anzahl übereinstimmender Zeichen -- Randlänge[Index im Muster])`

Beispiel 8. *Sie suchen das Muster* `ababaab` *im Text* `baabbaababaabbbabba`.
***Schritt 1**: Bestimmung des Randes*

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

***Schritt 2**: Suche*

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*
b	*a*	*a*	*b*	*b*	*a*	*a*	*b*	*a*	*b*	*a*	*a*	*b*	*b*	*b*	*a*	*b*	*b*	*a*
a	*b*	*a*	*b*	*a*	*a*	*b*												

Erste Abweichung an Index 0 → Rand[0] = -1, nach Formel oben: $0 + (0 - (-1)) = 1$.
Also verschieben wir das Muster an Indexposition 1.

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*
b	*a*	*a*	*b*	*b*	*a*	*a*	*b*	*a*	*b*	*a*	*a*	*b*	*b*	*b*	*a*	*b*	*b*	*a*
	a	*b*	*a*	*b*	*a*	*a*	*b*											

Abweichung an Index 2 (1 im Muster), Rand[1] = 0, mit Formel: $1 + (1 - 0) = 2$.

Wir verschieben also so, dass das erste Zeichen des Musters an Index 2 steht.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
b	a	a	b	b	a	a	b	a	b	a	a	b	b	b	a	b	b	a
		a	b	a	b	a	a	b										

Abweichung an Index 4 (2 im Muster), Rand[2] = 0, mit Formel: $2 + (2 - 0) = 4$. *Neue Indexposition des Musters: 4.*

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
b	a	a	b	b	a	a	b	a	b	a	a	b	b	b	a	b	b	a
				a	b	a	b	a	a	b								

Abweichung an Index 4 (0 im Muster), Rand[0] = -1, mit Formel: $4 + (0 - (-1)) = 5$. *Neue Indexposition: 5.*

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
b	a	a	b	b	a	a	b	a	b	a	a	b	b	b	a	b	b	a
					a	b	a	b	a	a	b							

Abweichung an Index 6 (1 im Muster), Rand[1] = 0, mit Formel: $5 + (1 - 0) = 6$.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
b	a	a	b	b	a	a	b	a	b	a	a	b	b	b	a	b	b	a
						a	b	a	b	a	a	b						

Nun tritt keine Abweichung ein, das erste Vorkommen des Musters im Suchtext ist gefunden. Da wir nun an allen Vorkommen interessiert sind, führen wir unsere Suche fort. Dafür bestimmen wir zunächst den Rand[7] = 2. Mit der Formel folgt: $6 + (7 - 2) = 11$. *An Position 11 können wir nun unsere Suche fortsetzen.*

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
b	a	a	b	b	a	a	b	a	b	a	a	b	b	b	a	b	b	a
											a	b	a	b	a	a	b	

$11 + (2 - 0) = 13$. *Damit verlassen wir den Suchtext, der Algorithmus terminiert, da kein weiteres Vorkommen gefunden wurde.*

Die Stärken des KMP-Algorithmus liegen folglich in der Berücksichtigung von Rändern an Anfang und Ende des Musters, die größere Sprünge als eins (im Vergleich zur einfachen Textsuche) ermöglichen. Entscheidend ist das Zusammenspiel aus Rand und Anzahl an Übereinstimmungen.

Implementierung als Pseudo-Code

```
KnuthMorrisPrattAlgorithm(Pattern, SearchSpace, Borders) {
        int i = 0;      /* Current position in search space */
        int j = 0;      /* Current position in pattern */

        while (i < SearchSpace.length) {
                // Move pattern until first character of
                // SearchSpace and Pattern is equal
                while (j >= 0 && SearchSpace[i] != Pattern[j]) {
                        j = Borders[j]
                }
```

```
                    // Now compare the next character
                    i = i + 1;
                    j = j + 1;

                    // End of pattern: return result
                    if (j == (Pattern.length-1)) {
                            print(i - (Pattern.length-1));
                    }
            }
}
```

Wie ersichtlich ist, beginnen wir damit, unser Muster unter Anwendung der Ränder an die erste Position zu verschieben, an der das erste Zeichen übereinstimmt. Nun inkrementieren wir beide Zählervariablen und betrachten folgende Zeichen, in dem wir im nächsten Schleifendurchlauf bei Abweichungen weiterverschieben, bis die Länge des Musters erreicht wurde. Dann geben wir diese Indexposition aus und fahren fort.

Komplexität Da dieser Algorithmus aus zwei Teilproblemen besteht, zerfällt auch die Komplexität:
Das Bestimmen der Ränder erfolgt linear für das gesamte Muster, daraus folgt in Landau-Notation: Rand$(x) \in O(n)$ bei n als Länge des Musters. Der Suchalgorithmus selbst verläuft ebenfalls linear – insgesamt erfolgen maximal so viele Durchläufe wie der Text Zeichen enthält. Daraus folgt: KMPSearch$(x, y, R[]) \in O(m)$
Da beide Teilprobleme getrennt voneinander und aufeinander aufbauend durchlaufen werden, ergibt sich eine Komplexität des gesamten Algorithmus von $O(n+m)$. [**KMPRand**]

8.3. Boyer-Moore-Algorithmus

Ähnlich dem Ansatz von Knuth, Morris und Pratt haben auch Robert S. Boyer und J Strother Moore einen vergleichbaren Algorithmus zur effizienteren Suche von Zeichenketten in Texten entwickelt. Im Gegensatz zum KMP-Algorithmus werden dort keine Ränder bestimmt, sondern erst im Falle einer Abweichung greifen zwei Heuristiken: Die **Bad-character**- und **Good-suffix**-Heuristik, mithilfe derer ebenfalls Sprünge ermittelt werden. Dieses Verfahren hat eine Komplexität von $O(n \cdot m)$ und ist damit im Allgemeinen (dem ungünstigsten Fall) weniger effizient als der KMP. Jedoch verläuft dieses Algorithmus nicht linear, sodass in einem günstigen Fall eine geringere Laufzeit erreicht werden kann.

Dieses Verfahren wird hier nicht näher dargestellt, da es nicht Teil der Vorlesung war. Weitere Informationen finden Sie bei [**Grundkurs**].

Aufgaben

Aufgabe 1 Führen Sie eine binäre Suche nach dem Element `7` in `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 17, 18, 19, 22]` durch. Notieren Sie jeweils Ihre Zwischenergebnisse.

Lösung auf Seite 68

Aufgabe 2 Bestimmen Sie die Ränder der folgenden Muster:

- `OTTOSMOPSKOTZT`
- `ANANASBANANA`
- `BONOBO`

Lösung auf Seite 68

Aufgabe 3 Wenden Sie den KMP-Algorithmus mit dem Muster `BONOBO` auf den Suchtext `BEINAHEBOTNOCHDASBONOBONOBOHAUSPLATZ`[2] an. Stellen Sie Ihre Zwischenergebnisse dar.

Lösung auf Seite 69

Aufgabe 4 Erläutern Sie den Algorithmus der binären Suche in Ihren eigenen Worten.

Lösung auf Seite 70

[2] Ja, der Text ist tatsächlich so gemeint, die Dopplung ist bewusst geschehen.

Teil III.
Sortieralgorithmen

Nach langen, qualvollen Stunden vor dem Bücherregal hassen Sie ihr Leben, ihr Studium und Sie beschließen, es wegzuwerfen – Sie fangen in der Bibliothek als Fachkraft für Lagerlogistik an und sind nun für die korrekte Anordnung der Bücher im Regal verantwortlich. Auch hier zeigen Sie ihre geistige Brillanz, indem Sie das Problem des Sortierens auf abstrakter Ebene betrachten.

9. Allgemeines und Eigenschaften

Definition 8. *Ein Sortieralgorithmus verarbeitet eine Folge von Elementen $e_1, ..., e_n$ so, dass am Ende eine Ausgabe $e_1, ..., e_n$ mit $e_1 \leq e_2 \leq ... \leq e_n$ erzeugt wird.*

Als zugrunde liegende Datenstrukturen kommen Arrays, verkettete Listen oder weitere, spezielle Datenstrukturen mit zu Mengen/Relationen ähnlichem Aufbau infrage.

Da Sie bei Sortiervorgängen häufig Daten im *key: value*-Format betrachten, unterscheiden wir zwischen **Schlüsseldaten**(also den keys) und **Satellitendaten**(den zugehörigen Werten). So können wie beispielsweise Wetterdaten betrachten: [08.05.2021: 31.5, 09.05.2021, 30.2, 07.05.2021, 16.3] – Hier sind die Tage jeweils die Schlüsseldaten und die Temperaturangaben (die Gleitkommazahlen) jeweils Satellitendaten.

Definition 9. *Ein Algorithmus sortiert Daten **stabil**, falls die Reihenfolge bereits sortierter Elemente mit gleichem Schlüssel nicht geändert wird und so insbesondere nach einem weiteren Kriterium sortierte Daten nach Ausführung des Algorithmus in diesem Kriterium ihre Sortierung beibehalten. Als **instabil** bezeichnen wir einen Sortieralgorithmus, falls das Gegenteil der Fall ist.*

Ähnlich den Suchalgorithmen unterscheiden sich auch Sortieralgorithmen in weiteren Aspekten:

- Laufzeiteffizienz (→ Landau-Notation)
- Speicherplatzeffizienz
- Ablaufschema: Rekursiv oder Iterativ
- Stabilität der Daten

Für die Speicherplatzeffizienz ergibt sich folgende Unterscheidung:

Definition 10. *Falls ein Sortieralgorithmus parallel zur Laufzeit keinen oder einen konstanten zusätzlichen Speicherbedarf benötigt, so wird er **in-place** ausgeführt. Ein Algorithmus, der hingegen zusätzlichen Speicherbedarf in Abhängigkeit zu den Daten benötigt, wird als **out-place** bezeichnet.*

10. Insertion Sort

Der erste Sortieralgorithmus, den wir betrachten, heißt **Insertion Sort**. Wie der Name schon andeutet, geht es darum, Elemente einzufügen – in einen bestehenden, bereits sortierten Teil.

Sie beginnen links in Ihrer Bücherreihe, starten mit dem zweiten Buch. Sie prüfen nun, ob es im Vergleich zum ersten richtig steht, andernfalls tauschen sie es. Nun betrachten Sie das dritte Buch und fügen es vor/zwischen/nach Buch 1 und 2 ein, je nach richtiger Position. So fahren Sie fort: Sie fügen das jeweils aktuelle Buch in die Reihe der bereits sortierten ein.

Implementierung als Pseudo-Code

```
InsertionSort(Array) {
        int i = 0;
        while (i < (Array.length-1)) {
                x = Array[i];
                int y = 0;
                while (x < Array[i-y]) {
                        y++;
                }
                Array[i-y] = x;
                i++;
        }
}
```

Beispiel 9. *Zu sortieren Sie das Array `[44, 55, 12, 42, 94, 18, 06, 67]`.*

Ausgangswerte	*44*	*55*	*12*	*42*	*94*	*18*	*06*	*67*
nach i = 1	*44*	*55*	*12*	*42*	*94*	*18*	*06*	*67*
nach i = 2	*12*	*44*	*55*	*42*	*94*	*18*	*06*	*67*
nach i = 3	*12*	*42*	*44*	*55*	*94*	*18*	*06*	*67*
nach i = 4	*12*	*42*	*44*	*55*	*94*	*18*	*06*	*67*
nach i = 5	*12*	*18*	*42*	*44*	*55*	*94*	*06*	*67*
nach i = 6	*06*	*12*	*18*	*42*	*44*	*55*	*94*	*67*
nach i = 7	*06*	*12*	*18*	*42*	*44*	*55*	*67*	*94*

*Quelle: [***Grundkurs***]*

Komplexität Im günstigsten Fall operiert dieses Verfahren auf einem bereits sortierten Feld. Dann sind – um dies festzustellen – $n-1$ Vergleiche erforderlich. Im schlechtesten Fall ist das Feld in genau falscher Richtung sortiert: Dann sind $\frac{n^2}{2}$ Vergleiche erforderlich, da die zweite Schleife auch maximal oft durchlaufen wird.

Insgesamt ergibt sich in LANDAU-Notation: $\in O(n^2)$

Eigenschaften

- Das Verfahren ist iterativ (erkennbar an den beiden Schleifen)
- Es ist stabil, da Vertauschungen nur im notwendigen Fall auftreten

- Das Verfahren operiert in-place, da kein bzw. nur konstanter zusätzlicher Speicher benötigt wird

Optimieren ließe sich das Verfahren durch den Einsatz von binärer Suche für die Einfügeposition – dann reduzieren sich die Vergleiche, nicht jedoch die Verschiebungen.

11. Bubblesort

Mit **Bubble Sort** lernen wir einen weiteren Algorithmus zum Sortieren kennen: Hier vergleichen wir zunächst benachbarte Elemente und tauschen diese bei Bedarf. So durchlaufen wir das gesamte Array, bis keine Vertauschungen mehr notwendig sind und alle Elemente in der korrekten Reihenfolge angeordnet sind.

Idee Durchlaufe Feld mehrmals und tausche benachbarte Elemente bei Bedarf. Terminiere, wenn in einem Durchlauf keine Vertauschung mehr erfolgt ist.

Beispiel 10. *Zu sortieren ist das Feld* `[6, 8, 2, 9, 1, 5, 4, 3]`.
$6 < 8$, *aber* $8 > 2$: `[6, 2, 8, 9, 1, 5, 4, 3]`.
$8 < 9$, *aber* $9 > 1$: `[6, 2, 8, 1, 9, 5, 4, 3]`
$9 > 5$: `[6, 2, 8, 1, 5, 9, 4, 3]`
$9 > 4$: `[6, 2, 8, 1, 5, 4, 9, 3]`
$9 > 3$: `[6, 2, 8, 1, 5, 4, 3, 9]`

Nächste Iteration:
$6 > 2$: `[2, 6, 8, 1, 5, 4, 3, 9]`
$6 < 8$, *aber* $8 > 1$: `[2, 6, 1, 8, 5, 4, 3, 9]`
$8 > 5$: `[2, 6, 1, 5, 8, 4, 3, 9]`
$8 > 4$: `[2, 6, 1, 5, 4, 8, 3, 9]`
$8 > 3$: `[2, 6, 1, 5, 4, 3, 8, 9]`
$8 < 9$.

Nächste Iteration:
$2 < 6$, *aber* $6 > 1$: `[2, 1, 6, 5, 4, 3, 8, 9]`
$6 > 5$: `[2, 1, 5, 6, 4, 3, 8, 9]`
$6 > 4$: `[2, 1, 5, 4, 6, 3, 8, 9]`
$6 > 3$: `[2, 1, 5, 4, 3, 6, 8, 9]`
$6 < 8$ *und* $8 < 9$.

Nächste Iteration:
$2 > 1$: `[1, 2, 5, 4, 3, 6, 8, 9]`
$2 < 5$, *aber* $5 > 4$: `[1, 2, 4, 5, 3, 6, 8, 9]`
$5 > 3$: `[1, 2, 4, 3, 5, 6, 8, 9]`
$5 < 6$, $6 < 8$ *und* $8 < 9$.

Nächste Iteration:
$1 < 2$, $2 < 4$, *aber* $4 > 3$: `[1, 2, 3, 4, 5, 6, 8, 9]`
$4 < 5$, $5 < 6$, $6 < 8$ *und* $8 < 9$. *Terminiert.*

Aus dem Beispiel können wir ableiten, dass in jeder Iteration $n - 1$ Vergleiche erfolgen. Dies ist ein Indiz für die Komplexität, die wir später betrachten möchten.

Implementation als Pseudo-Code

```
BubbleSort(Array) {
        int i = 0;
        int j = 0;
        bool SwapFlag = True;
        while (SwapFlag == true) {
                SwapFlag = false;
                while (i < (Array.length-1) {
                        if (Array[i] > Array[i+1]) {
                                swap(Array[i], Array[i+1]);
                                SwapFlag = true;
                        }
                }
        }
}
```

Die `SwapFlag` ist Terminationskriterium: Falls in einer Iteration kein Tausch mehr durchgeführt wurde, ist das Array vollständig sortiert.

Komplexität Im günstigsten Fall ist das Array bereits vollständig sortiert, dann werden $n-1$ Vergleiche notwendig. Im ungünstigsten Fall ist das Aray absteigend sortiert, nun sind $\frac{n^2}{2}$ Vergleiche erforderlich.

Somit ergibt sich erneut eine Komplexität von $O(n^2)$.

Im Gegensatz zu ursprünglichen Implementierungen haben wir hier bereits eine Optimierung vorgenommen: Die SwapFlag bewirkt, dass im günstigsten Fall $O(n)$ die passende Komplexitätsklasse ist.

Eigenschaften

- Auch Bubble Sort arbeitet offensichtlich iterativ.
- Bubble Sort ist ebenfalls stabil.
- Der Speicheraufwand ist in-place.

Quelle und weiterführende Literatur: [**Wirth**] [**Taschenbuch**]

12. Mergesort

Die nächste Idee, wie Sie Ihr Bücherregal sortieren können: Mergesort! Sie teilen das Regal in zwei Hälften, sortieren jede Hälfte, in dem Sie diese wiederum halbieren, bis die Hälften aus einem Buch bestehen. Sortieren Sie diesen Teil, indem Sie die Bücher richtig anordnen und die Hälften so wieder von klein nach groß nach einer Art Reißverschlussprinzip verbinden, bis Sie die Gesamtheit der Elemente in sortierter Reihenfolge erhalten.

Beispiel 11. *Zu sortieren:* `[1, 7, 6, 2, 9, 4, 3, 8, 10, 5].`
Halbieren:
`[1, 7, 6, 2, 9] | [4, 3, 8, 10, 5]`
`[1, 7, 6][2, 9] | [4, 3, 8][10, 5]`

[1, 7] [6] [2] [9] | [4, 3] [8] [10] [5]
[1] [7] [6] [2] [9] | [4] [3] [8] [10] [5]

Sortieren (nach rekursiven Aufrufen):
[1] [2] [6] [7] [9] | [3] [4] [5] [8] [10]

Merge:
[1] [2] [3] [4] [5] [6] [7] [8] [9] [10]

Implementierung als Pseudo-Code

```
MergeSort(int A[], int al, int ar) {
        if (ar>al) {
                int m = (ar+al) / 2;
                MergeSort(A[], al, m);
                MergeSort(A[], m+1, ar);

                int B[] = new int[ar-al+a];
                merge(A, al, m, a, m+1, ar, B);

                for (int i = 0; i < ar-al+1; i++) {
                        A{al+1] = B[i];
                }
        }
}
merge(A[], B[], al, ar, bl, br, C[]) {
        int i = al;
        int j = bl;
        for (int k = 0; k <= ar-al+br-bl+1; k++) {
                if (i > ar) {
                        C[k] = B[j+1];
                        continue;
                }
                if (j > br) {
                        C[k] = A[i+1];
                        continue;
                }
                C[k] = (A[i] < B[j] ? A[i++] : B[j++];
        }
}
```

Quelle: [**Taschenbuch**]

Komplexität Für Mergesort ergibt sich – da es sich um einen rekursiven Algorithmus handelt – eine Rekursionsgleichung:

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + n = 2T(\frac{n}{2}) + n$$

Jeder Rekursionsteil beschreibt die Komplexität der einzelnen Hälften und n das anschließende Zusammenfügen. Somit ergibt sich unter Anwendung des Mastertheorems eine Komplexität von

$$T(n) \in O(n \cdot log(n))$$

Eigenschaften

- rekursiv
- stabil
- Je nach Implementierung in-place oder out-place

13. Quicksort

Willkommen in der Königksklasse: Quicksort trägt schon einen bezeichnenden, etwas anmaßenden Titel. Wir betrachten nun, ob der Algorithmus dem gerecht wird.

Idee Wir sortieren eine Menge an Elementen, in dem wir das erste Element p (das wir als **Pivot** bezeichnen) herausgreifen und nun unser Feld/Array so verschieben, dass alle kleineren Elemente ... $< p$ links von p stehen und alle größeren rechts davon - wohlgemerkt noch nicht sortiert. Nun betrachten wir die neu gebildeten Abschnitte $(a_0, ..., a_{p-1})$ und $a_{p+1}, ..., a_n)$ und verfahren auf diesen Teilstapeln oder Teilmengen rekursiv. Folglich betrachten wir erneut die jeweils ersten Elemente und verfahren ebenso. Da wir die Operationen in-place ausführen, ist kein merge-Schritt notwendig (im Gegensatz zu MergeSort).

Beispiel 12. *Wir betrachten ein Feld* `[8, 17, 28, 15, 11, 1, 3, 20, 25, 6, 12, 5]`.
Das erste Element ist folglich 8, also erhalten wir:
`[1, 3, 6, 5, 8, 17, 28, 15, 11, 20, 25, 12]`.
Wir betrachten nun die beiden Teilmengen `[1, 3, 6, 5]` *und* `[17, 28, 15, 11, 20, 25, 12]`.
Da 1 offensichtlich kleinstes Element der ersten Teilmenge ist, 3 anschließend ebenfalls beim nächsten Aufruf, erhalten wir schnell `[1, 3, 5, 6]`.

In der zweiten Menge erhalten wir `[15, 11, 12]` *und* `[28, 20, 25]`, *die wir schnell zu* `[11, 12, 15]` *und* `[20, 25, 28]`, *zusammen* `[11, 12, 15, 17, 20, 25, 28]`.

Abschließend erhalten wir das sortierte Feld `[1, 3, 5, 6, 8, 11, 12, 15, 17, 20, 25, 28]`.

Implementierung als Pseudo-Code

```
QuickSort(Array, lo, hi) {
        if (lo < hi) {
                pivot = Array[lo];
                i = lo;
                j = hi+1;
        }
        while (true) {
                while (Array[++i] < pivot && i < hi) {

                }
                while (Array[--j] > pivot && j > lo) {

                }
```

```
                if (i < j) {
                        swap(Array, i, j);
                }
                else {
                        break;
                }
        }
        swap(Array, j, lo);

        quickSort(Array, lo, j-1);
        quickSort(Array, j+1, hi);
}
swap(Array, i, j) {
        t = Array[i];
        Array[i] = Array[j];
        Array[j] = t;
}
```

abgewandelt nach [**Taschenbuch**]

Komplexität Offensichtlich handelt es sich bei Quicksort ebenfalls um einen rekursiven Sortieralgorithmus. Im günstigsten Fall ergibt sich eine Komplexität von $O(n \cdot log(n))$, da dann das Pivot-Element genau der Median ist und so beide Hälften glatt rekursiv abgearbeitet werden können. Im schlechtesten Fall beträgt die Komplexität $O(n^2)$, da dann das Pivot-Element jeweils das kleinste oder größte ist.

Wir geben die Komplexität von Quicksort im Durchschnittsfall mit $O(n \cdot log(n))$ an, behalten allerdings im Kopf, dass in einem ungünstigen Fall eine deutlich schlechtere Laufzeit in Kauf genommen werden muss.

Eigenschaften

- rekursiv
- nicht stabil
- in-place, da kein Hilfsfeld benötigt wird
- optimierbar beispielsweise durch Einfügen einer Schranke, bis zu der Insertion Sort genutzt wird (z.B. bei einer geringen zweistelligen Anzahl von Elementen) oder durch 3-Way-Partitioning, bei dem neben $<, >$ auch der Fall $=$ betrachtet wird. Dies ist logischerweise nur dann sinnvoll, falls es mehrere Elemente mit gleichen Schlüsseln gibt und diese bevorzugt als Pivot-Element genutzt werden.

14. Weitere Sortieralgorithmen

Auf weitere Sortieralgorithmen wie z.B. Selection Sort [**Taschenbuch**], Shellsort oder Shakesort [**Wirth**] und [**Grundkurs**] kann hier nicht näher eingegangen werden. In genannten Büchern finden Sie nähere Informationen zu diesen Algorithmen.

15. Allgemeine Komplexitätsbetrachtung von Sortieralgorithmen

Satz 5. *Ein Sortieralgorithmus, der auf Vergleichen von Datenelementen beruht, hat mindestens eine Komplexität von $O(n \cdot log(n))$.*

Um diesen Satz zu verstehen, verdeutlichen wir uns zunächst, was eine Sortierung im Allgemeinen bedeutet:
Aus der Menge aller möglichen Permutationen (also Anordnungen) von den Elementen unserer Menge suchen wir genau die eine heraus, in der alle Elemente *sortiert* sind, nach unserem Kriterium. Insgesamt gibt es bei n Elementen $n!$ Vergleiche.

Beweisidee Wir betrachten die Menge aller Permutationen, ordnen ihnen einen Wert zu und vergleichen nun jede Permutation mit der gewünschten, also derjenigen, in dem die Elemente vollständig sortiert sind. Nun können wir eine Hälfte an Permutationen ausmachen, deren Wert „kleiner“ ist und eine weitere Hälfte, deren Wert „größer oder gleich“ der Permutation ist. Hier sind nun $\frac{n!}{2}$ Vergleiche erforderlich.
Wir halbieren die Hälften fortlaufend und erhalten im letzten Schritt die eine verbleibende Permutation.
Sie ist nach $log(n!)$ Vergleichen erreicht.

$$O(log(n!)) \in O(n \cdot log(n))$$

Beweis.

$$n^n \geq n! \tag{33}$$
$$1 \cdot 2 \cdot 3 \cdot ... \cdot (n-1) \cdot n = n \cdot (n-1) \cdot ... \cdot 3 \cdot 2 \cdot 1 \tag{34}$$
$$(n-k) \cdot (k+1) \geq n \quad \forall 0 \leq k \leq n \tag{35}$$
$$(n-1) \cdot ((n-1) \cdot 2) \cdot ... \cdot ((n-k) \cdot (k+1)) \cdot (1 \cdot n) \geq n \cdot n \cdot ... \cdot n \tag{36}$$
$$n! \cdot n! \geq n^n \tag{37}$$
$$(1) \text{ und } (4): \qquad n^{2n} \geq n! \cdot n! \geq n^n \tag{38}$$
$$2n \cdot log(n) \geq 2 \cdot log(n!) \geq n \cdot log(n) \tag{39}$$
$$O(log(n!)) = O(n \cdot log(n)) \tag{40}$$

□

Tatsächlich ist damit gezeigt, dass es keinen Sortieralgorithmus gibt, dessen Komplexität geringer ist als $O(n \cdot log(n))$.

Aufgaben

Aufgabe 1 Sortieren Sie das Feld `[56, 10, 15, 98, 99, 12, 30, 80]` aufsteigend. Notieren Sie Zwischenschritte

- Insertion Sort
- Bubblesort
- Quicksort
- Mergesort

Lösung auf Seite 70
entnommen aus [**GrUeb**]

Aufgabe 2 Beschreiben Sie Quicksort mit eigenen Worten.

Lösung auf Seite 71

Aufgabe 3 Begründen Sie, mit welchen Verfahren Sie folgende Felder in optimaler Laufzeit aufsteigend sortieren können.

- `[1, 9, 2, 8, 3, 7, 4, 6, 5]`
- `[9, 7, 7, 2, 5, 4, 7, 3, 1]`
- `[9, 8, 7, 6, 5, 4, 3, 2, 1]`

Lösung auf Seite 71

Aufgabe 4 Entwickeln Sie für Mergesort und Quicksort eine anschauliche Darstellungsform.

Lösung auf Seite 72

Teil IV.
Bäume

Vielleicht haben Sie die letzte Aufgabe vom letzten Teil gelöst – Mit relativ hoher Wahrscheinlichkeit werden Sie eine Struktur beschrieben oder skizziert haben, die wir im Folgenden näher betrachten: Bäume!

Damit Sie vor lauter Bäumen den Wald (und die Wahrheit) nicht aus den Augen verlieren, widmen wir uns in aller Ausführlichkeit diesem in der Informatik sehr häufig anzutreffenden Gebilde, denken Sie beispielsweise an Ableitungs-, Entscheidungs-, Syntax- oder Codebäume.

16. Der Wurzelbaum

Definition 11. *Ein Wurzelbaum $B = (V, E, r)$ besteht aus einer endlichen Menge von Konten V, einer endlichen Menge von gerichteten Kanten $E \subset V \times V$ und aus der Wurzel $r \in V$.*

- *Ein Knoten r ist ein Wurzelbaum ($B = (\{r\}, \emptyset, r)$).*
- *Sind $B_1 = (V_1, E_1), ..., B_k = (V_k, E_k)$ Bäume mit den Wurzeln $r_1, ..., r_k$, so erweitern wir die Knotenmenge V um eine neue Wurzel r und die Kantenmenge E um die Kanten $(r, r_i), i = 1, ...k$.*

  Der Baum

  $$(V_1 \cup ... \cup V_n \cup \{r\}, \{(r, r_i) \mid i = 1, ..., k\} \cup E_1 \cup ... \cup E_k, r)$$

  ist ein Wurzelbaum.

Folglich betrachten wir hier nun also einen Baum mit einem Knoten, der den Ursprung darstellt, aus dem sich die anderen Knoten ableiten. In etwa so:

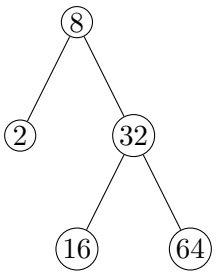


Abbildung 1: Wurzelbaum

Beispiel 13.

Definition 12. *Ein Baum, der keine Knoten und Kanten besitzt, heißt* ***leerer Baum****.*

Zu diesen Bäumen, von denen es die unterschiedlichsten Formen gibt, betrachten wir nun eine Reihe von Eigenschaften und Begrifflichkeiten.

Definition 13.

- *Ist $e = (v, w) \in E$, so heißt v **Vater** von w und w **Sohn** oder **Kind** von v. Ein Knoten, der keine Söhne hat, heißt **Blatt**.*
- *Ein Pfad P in B ist eine Folge von Knoten $v_0, ..., v_n$ mit $(v_i, v_{i+1}) \in E, i = 0, ..., n-1$.$n$ heißt **Länge** von P.*
- *Seien $v, w \in V$. Der Knoten w heißt **von Knoten** v **aus erreichbar**, falls es einen Pfad P von v nach w gibt.*
- *Jeden Knoten v von B können wir als Wurzel des Teilbaums der von v aus erreichbaren Knoten betrachten.*

  *Hat v die Söhne $v_i, ..., v_k$, so heißen die Bäume $B_1, ..., B_k$ mit den Wurzeln $v_i, ..., v_k$ die **Teilbäume** von V.*
- *Die **Höhe** eines Knotens v ist das Maximum der Längen aller Pfade, die in v beginnen.*
- *Die **Tiefe** eines Knotens ist die Länge des Pfades von der Wurzel zu v.*

  Die Knoten der Tiefe i bilden die i-te Ebene des Baumes.
- *Die Höhe (und auch Tiefe) eines Baumes (nicht Knotens!) ist gegeben durch die Höhe der Wurzel.*
- *Ein leerer Baum besitzt die Höhe/Tiefe -1.*
- *Die **Ordnung** eines Baumes ist die maximale Anzahl an Söhnen eines Knotens.*

Zur Veranschaulichung wenden wir diese Begriffe auf den folgenden Beispielbaum an:

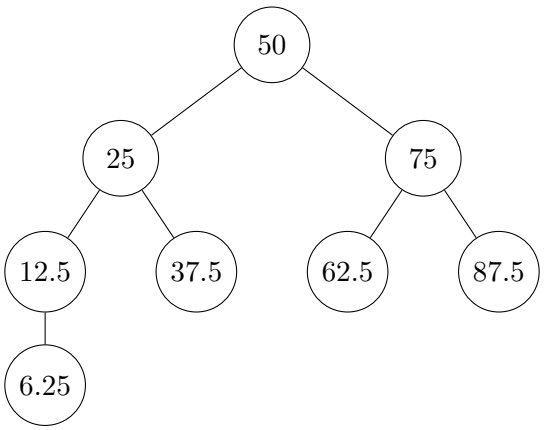


Abbildung 2: Beispielbaum

Beispiel 14.

- *50 ist die Wurzel des Baumes.*
- *6.25 ist Sohn/Kind von 12.5 und Blatt, da es keine Söhne hat.*

- *12.5 ist erreichbar von 50, da es einen Pfad gibt. Generell ist jedes Element eines Wurzelbaums von der Wurzel aus erreichbar.*
- *87.5 ist von 6.25 aus nicht erreichbar, da es keinen Pfad gibt.*
- *Die Höhe von 75 beträgt 1. Die Tiefe von 6.25 beträgt 3.*
- *Die Höhe/Tiefe des Baumes beträgt 3.*
- *Die Ordnung des Baumes ist 2, da kein Knoten mehr als zwei Kinder hat.*

17. Binäre Bäume

Definition 14. *Ein Baum mit der Ordnung 2 heißt **binärer Baum**. Liegen die Elemente darin in geordneter Form vor, d.h. der Wert des linken Sohnes ist stets kleiner und der Wert des rechten Sohnes ist stets größer als sein Vater, spricht man von einem **geordneten binären Baum** [**Wirth**] oder **binären Suchbaum**.*

Diese besonderen Bäume wollen wir nun näher betrachten, da wir in binären Bäumen algorithmisch vorgehen können und so beispielsweise Suchprobleme lösen.

Bemerkung 4. *Sei n die Anzahl der Knoten in einem binären Baum der Höhe h. Dann ist die Anzahl der Knoten $n \leq 2^{n+1} - 1$ oder äquivalent dazu. Die Höhe h ist mindestens $log_2(n+1) - 1$, also $\lceil log_2(n+1) \rceil - 1 \leq h$. Diese Schranke wird für einen binären Baum, in dem alle Ebenen vollständig besetzt sind, angenommen.*

Implementierung Wie schon eingangs erwähnt, hängen Bäume und Rekursion naturgemäß eng beieinander. Dies zeigt sich, wenn wir eine Implementierung eines binären Baums betrachten:

```
struct node {
        int key;
        type value;
        node left, right;
}
```

Ein Knoten besteht also wiederum selbst aus Knoten, im Falle eines Blattes sind `left` und `right` dann leer. Wichtig ist die Unterscheidung zwischen `key` und `value` – hier kann es leicht zu Unklarheiten kommen.

17.1. Traversierung von Binärbäumen

Auf Baumstrukturen lassen sich verschiedene Operationen durchführen. Grundlage dafür ist häufig ein *traversieren* („durchlaufen“) des Baumes. Dafür gibt es drei, jeweils rekursiv implementierte Vorgehensweisen, die unter anderem bei [**Wirth**] beschrieben werden:

- **Preorder** – W, A, B (besuche Wurzel vor den Teilbäumen)
- **Inorder** – A, W, B
- **Postorder** – A, B, W (besuche Wurzel nach den Teilbäumen)

Grundlage dafür ist der allgemeine binäre Baum:

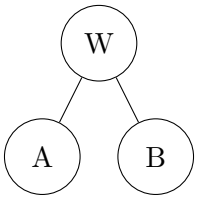


Abbildung 3: Allgemeiner binärer Baum

Beispiel 15. *Betrachten wir den Baum aus vorangegangenem Beispiel und traversieren wir diesen in allen drei Varianten:*

Preorder
50, 25, 12.5, 6.25, 37.5, 75, 63.5, 87.5

Inorder
6.25, 12.5, 25, 37.5, 50, 63.5, 75, 87.5

Postorder
6.25, 37.5, 12.5, 25, 63.5, 87.5, 75, 50

Neben den drei genannten Vorgehenweisen betrachten wir noch die klassische, den Baum in seinen Ebenen ausgebende: **levelorder**. Auf unser Beispiel bezogen wäre dies: `50, 25, 75, 12.5, 37.5, 63.5, 87.5, 6.25`

17.2. Operationen

Auf binären Bäumen lassen sich nun drei Operationen konstruieren: **Suchen, Einfügen** und **Löschen**.

Suchen Ein Baum wird durchsucht, in dem jeweils das Suchelement mit dem Wert des Knotens verglichen und anschließend mit dem linken Sohn, falls das gesuchte Element kleiner ist, oder dem rechten Sohn, falls das gesuchte Element größer ist, als neuen Knoten rekursiv fortgefahren.

```
node Search(int k, node Tree) {
        if (Tree == null) {
                return null;
        }
        else if (k == Tree.key) {
                return Tree;
        }
        else if (k < Tree.key) {
                Search(k, Tree.left);
        }
        else if (k > Tree.key) {
                Search(k, Tree.right);
        }
}
```

Einfügen Ähnlich läuft das Einfügen eines Elements k mit Wert v:

```
node Insert(int k, node Tree, type v) {
        if (T == null) {
                return "neuer Knoten (k, v)";
        }
        else if (k < T.key) {
                T.left =          Insert(k, T.left, v);
        }
        else if (k > T.key) {
                T.right = Insert(k, T.right, v);
        }
        else if (k == T.key) {
                T.value = v;
        }
        return T;
}
```

Ein Einfügen kann folglich auch ein „Wert ändern“ sein.

Löschen Das Löschen eines Knotens gestaltet sich etwas aufwendiger.
Suche Zeiger T auf den zu löschenden Schlüssel/Knoten und unterscheide:

- Das zu löschende Element hat keine Kinder: Setze Zeiger T auf null
- Das zu löschende Element hat ein Kind: Ersetze T durch den Zeiger auf den Kindknoten
- Das zu löschende Element hat zwei Kinder: Ersetze den gelöschten Knoten durch den kleinsten Schlüssel des rechten Teilbaums.

Eine Implementierung davon finden Sie bei [**Wirth**].

Komplexität Für die Suche und das Löschen liegt die Komplexität im Durchschnittsfall in $O(n)$, wobei n die Höhe des Baumes ist. Unter günstigen Umständen ist auch eine Komplexität von $O(log(n))$ möglich. Die Komplexität des Einfügens liegt in $O(1)$.

18. AVL-Bäume

Ein weiterer Typ von Bäumen sind die sogenannten **AVL-Bäume**, die durch einen geringen zusätzlichen Aufwand beim Einfügen/Löschen erzeugt werden und eine günstigere Komplexität besitzen.

Definition 15. *Ein binärer Baum heißt **(AVL-)ausgeglichen**, falls für jeden Konten v gilt:*

Die Höhen des linken und rechten Teilbaums unterscheiden sich höchstens um 1.
*Diese Differenz der Höhen heißt **Balance** von v.*

$$Balance = h(rT) - h(lT)$$

*Ausgeglichene, binäre Suchbäume heißen nun **AVL-Bäume**.*

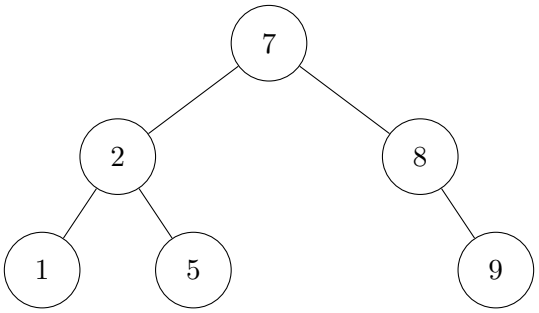


Abbildung 4: Beispiel für einen AVL-Baum

Beispiel 16. *Dieser Baum ist ein AVL-Baum, da er die drei Kriterien erfüllt:*

- *binär: Alle Knoten haben höchstens zwei Kinder, mindestens ein Knoten hat mindestens zwei Kinder*
- *Suchbaum: Alle Elemente im linken Teilbaum sind kleiner als das Wurzelelement des Teilbaums, alle Elemente im rechten Teilbaum sind größer als der Teilbaum*
- *AVL-Kriterium: Die Balancen der Teilbäume unterscheiden sich höchstens um 1.*

Kein AVL-Baum hingegen ist der folgende: Hier ist die AVL-Bedingung offensichtlich

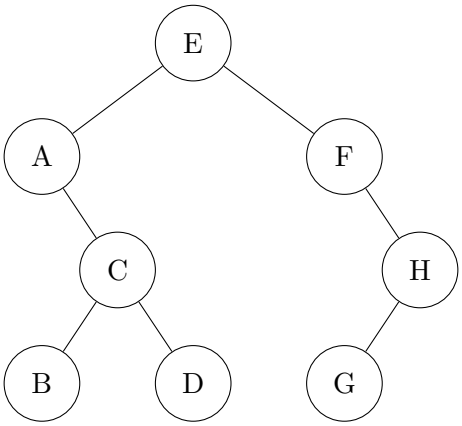


Abbildung 5: Kein Beispiel für einen AVL-Baum

verletzt: Die Balance an den Knoten A und F ist jeweils größer als 1. Damit handelt es sich nicht um einen AVL-Baum.

Um zu ermitteln, ob es sich bei einem Baum um einen AVL-Baum handelt, prüfen wir also zunächst, ob es sich um einen binären Suchbaum handelt. Nun bestimmen wir zu jedem Knoten die Balance: Jedes Blatt hat die Balance 0, falls der rechte Teilbaum eine größere Höhe hat, ist die Balance positiv. Falls der linke Teilbaum eine größere Höhe hat, ist die Balance negativ. Eine Balance von 1 / -1 wird toleriert, alle größeren Balancen führen dazu, dass der Baum kein AVL-Baum sein kann. Ein Knoten mit Balance größer 1 genügt.

Bemerkung 5. *Für die Höhe eines ausgeglichenen Baumes mit n Knoten gilt:*

$$h < 1.45 \cdot log_2(n+2) - 1.33$$

Dieses Kriterium eignet sich aus naheliegenden Gründen lediglich für eine Abschätzung.

Suche Da ein AVL-Baum ein binärer Suchbaum ist, verwenden wir zum Suchen die Suchfunktion eines binären Suchbaums. Das AVL-Kriterium hat keine Auswirkungen auf die Komplexität.

Einfügen Wir suchen nach dem einzufügenden Element e. Falls e nicht im Baum, endet die Suche in einem Blatt. An diesem Blatt verankern wir einen neuen Knoten und füllen ihn mit e. **Hierbei kann die AVL-Bedingung verletzt werden!**
Anschließend reorganisieren wir den Baum, um die AVL-Bedingung wiederherzustellen:

- Prüfe für jeden Knoten n des Suchpfades, ob er ausgeglichen ist.
- Falls nicht: Rotation

Rotation Rotation können wir uns vorstellen als eine Art „Kippen“ eines Knotens, um eine Balance wiederherzustellen bzw. zu nivellieren. Wir unterscheiden zwei Rotationsarten:

- **Linksrotation (LR)** Ein rechtes Kind kann nach links rotiert werden.
- **Rechtsrotation (RR)** Ein linkes Kind kann nach rechts rotiert werden.

Rotation ist möglich, da ein vormals linkes Kind – das kleiner als die Wurzel ist – auch die neue Wurzel mit der vorherigen Wurzel als neues rechtes Kind – das dann weiterhin größer ist – sein kann. Im Prinzip ändert sich folglich jeweils nur die Verwandtschaftsbeziehung zwischen Kind und Wurzel mit dem Effekt, dass auch die Kinder neu, meistens paritätischer, angeordnet werden.

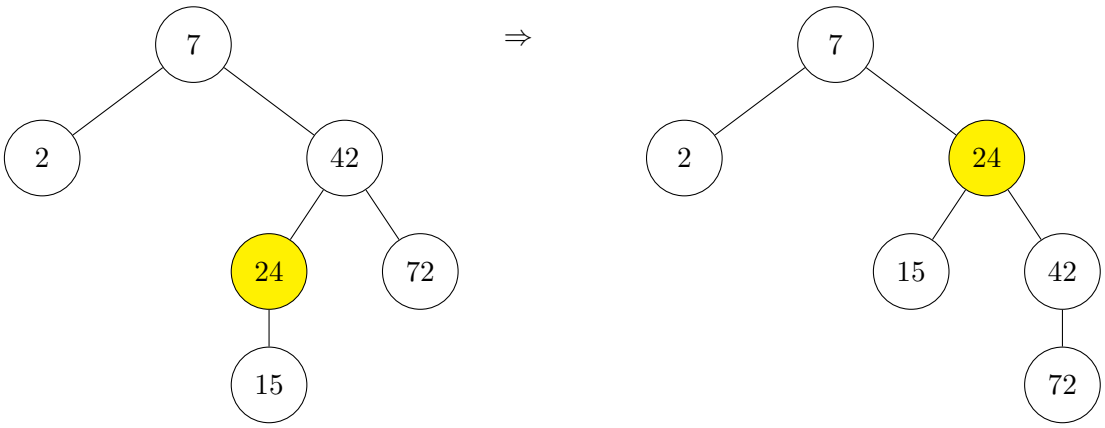


Abbildung 6: AVL-Baum mit Rechtsrotation des Knotens 24

Beispiel 17. *Der hier vorliegende Baum wurde am Knoten 24 nach rechts rotiert. Dadurch ist die 24 die neue Wurzel geworden, während die 42 nun eine Ebene tiefer ein Kind geworden ist. Doch auch nach dieser Rotation zeigt sich, dass die AVL-Bedingung weiterhin nicht erfüllt ist. Also ist ein weiterer Rotationsschritt notwendig: Nun zeigt sich, dass die AVL-Bedingung wieder erfüllt ist.*

Aus dem Beispiel können wir nun die Fälle für Rotationen unterscheiden:

- Eine einfache Rotation wird dann notwendig, wenn

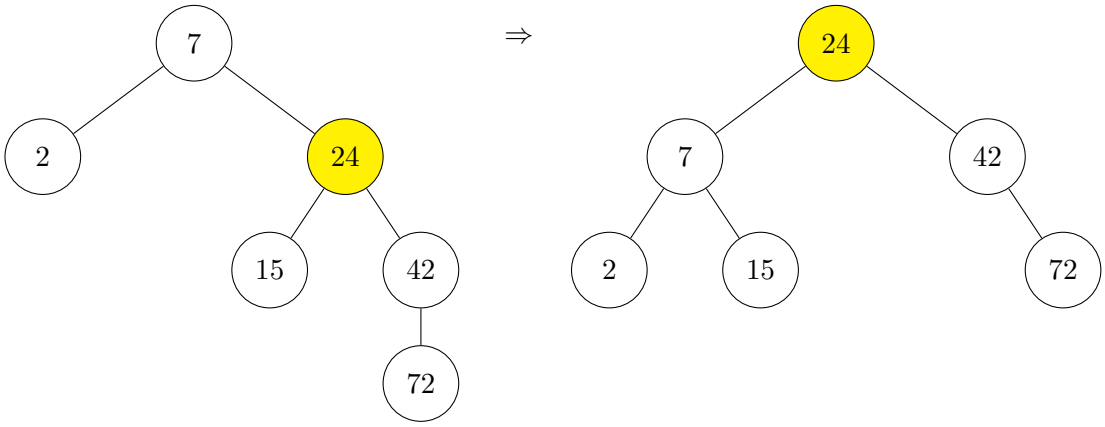


Abbildung 7: AVL-Baum mit Linksrotation des Knotens 24

  - in einem linken Teilbaum ein linkes Kind eingefügt wird → einfache Rechtsrotation
  - in einem rechten Teilbaum ein rechtes Kind eingefügt wird → einfache Linksrotation

- Eine zweifache Rotation wird hingegen angewandt, falls
  - in einem linken Teilbaum ein rechtes Kind eingefügt wird → Links-Rechts-Rotation
  - in einem rechten Teilbaum ein linkes Kind eingefügt wird → Rechts-Links-Rotation

Löschen Zunächst wird wie in einem binären Suchbaum vorgegangen. Anschließend muss – wie beim Einfügen – die AVL-Bedingung durch Rotationen wiederhergestellt werden.

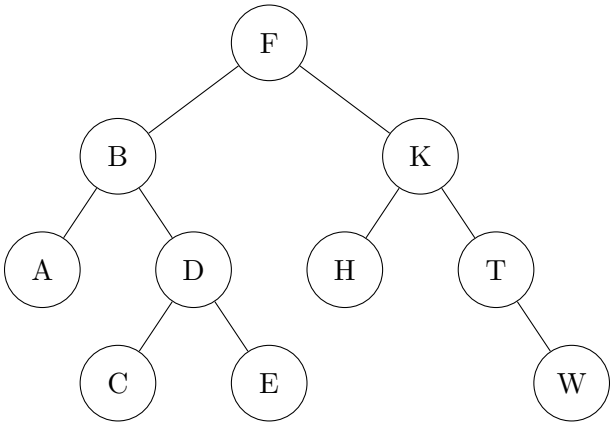
Für die Betrachtung der Komplexität ergibt sich nun:

Theorem 1. *Die Operationen Suchen, Einfügen und Löschen eines Schlüssels können in* $O(log(n))$ *durchgeführt werden.*

Aufgaben

Aufgabe 1 Gegeben ist der folgende Baum.

- Geben Sie folgende Eigenschaften an: Wurzel, Höhe des Baumes, Höhe des Knotens E, Ordnung des Baumes, Balance des Knotens K
- Fügen Sie danach den Knoten J hinzu.
- Löschen Sie anschließend den Knoten B.



Lösung auf Seite 72

Aufgabe 2 Untersuchen und begründen Sie, ob es sich bei diesem Baum (wie gegeben, vor Aufgabe 1) um einen AVL-Baum handelt. Fügen Sie hierzu die Balancen jedes Knotens ein.

Lösung auf Seite 73

Aufgabe 3 Falls es sich nicht um einen AVL-Baum handelt, ergänzen Sie den Baum zu einem AVL-Baum und führen Sie dann folgende Aufgabe aus.

- Löschen Sie den Knoten A.
- Fügen Sie den Knoten Z hinzu.

Falls Rotationen erforderlich sind, stellen Sie diese detailliert dar.

Lösung auf Seite 73

Aufgabe 4 Beschreiben Sie in Worten und Pseudocode das Vorgehen beim Suchen eines Elementes in einem binären Baum. Beurteilen Sie, ob dieses Verfahren auch in einem Baum der Ordnung 3 angewendet werden kann.

Lösung auf Seite 73

Teil V.
Heaps

Eigentlich auch zu Bäumen gehörend, betrachten wir im Folgenden die Datenstruktur **Heap**. Als Heap bezeichnen wir eine Struktur (einen „Haufen“) von Objekten, die über Schlüssel mit einer Priorität versehen sind. Wirth definiert Heaps in [**Wirth**] wie folgt:

Definition 16. *Ein heap ist definiert als eine Folge von Schlüsseln h[l], h[l+1], ..., h[r] mit den Relationen:*

- $h[i] \leq h[2i]$
- $h[i] \leq h[2i+1]\ \forall i = 1...r/2$

Betrachten wir einen Heap als Baum, so ist $h[2i]$ das linke Kind des Knotens $h[i]$ und $h[2i+1]$ entsprechend das rechte. Somit muss jede Wurzel eines Teilbaums kleiner als ihre Kinder sein. Diese Struktur bezeichnen wir als **Min-Heap**. Invers dazu bezeichnen wir Heaps, in denen die Wurzel eines Teilbaums jeweils größer ist als ihre Kinder als **Max-Heap**. Beide Varianten finden sich in der Literatur. Da in der Vorlesung lediglich der Max-Heap eingeführt wurde, beschränken wir uns der Übersichtlichkeit halber auf diesen Typ.

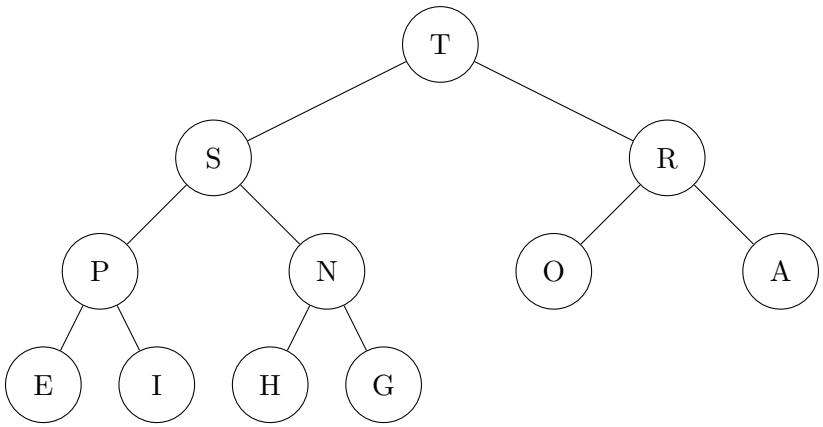


Abbildung 8: Binärer Max-Heap

Beispiel 18.

Wir erkennen und formulieren als Eigenschaften:

- Es handelt sich um einen (fast) vollständigen Binärbaum, d.h. die Blätter befinden sich auf höchstens zwei verschiedenen Levels.
- Die Blätter auf dem untersten Level sind linksbündig angeordnet.

Implementierung eines Heaps als Array Ein Heap lässt sich als Feld/Array implementieren, da für die Indizes die oberen Bedingungen leicht erkannt und umgesetzt werden können. Als Array erhalten wir damit: In $A[0]$ speichern wir die Größe des Heaps, in diesem Fall 11, da es sich um 11 Elemente handelt.
Wir können erneut festhalten:

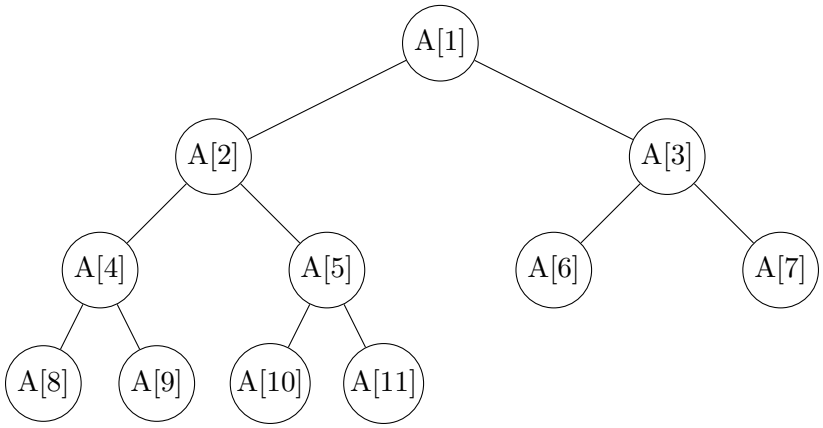


Abbildung 9: Binärer Max-Heap in Array-Darstellung

0	1	2	3	4	5	6	7	8	9	10	11
11	T	S	R	P	N	O	A	E	I	H	G

- $A[i]$ hat linkes Kind $A[2i]$
- $A[i]$ hat rechtes Kind $A[2i+1]$
- $A[i]$ hat Vater $A[\frac{i}{2}]$ (bei Ganzzahldivision)
- $A[i]$ befindet sich auf dem Level $\lfloor log_2(i) \rfloor$

19. Herstellen der Heap-Eigenschaft

Um sicherzustellen, dass es sich bei einem gegebenen Baum um einen Heap handelt, unterscheiden wir zwei Szenarien.

- **Szenario 1:** Der Schlüssel eines Kindes ist größer als der Schlüssel eines Elternknotens. Dann vertauschen wir – logischerweise – den Schlüssel des Kindes mit dem Schlüssel des Elternknotens. Das Kind „schwimmt nach oben“, deshalb heißt dieses Vorgehen auch *swim-up*.

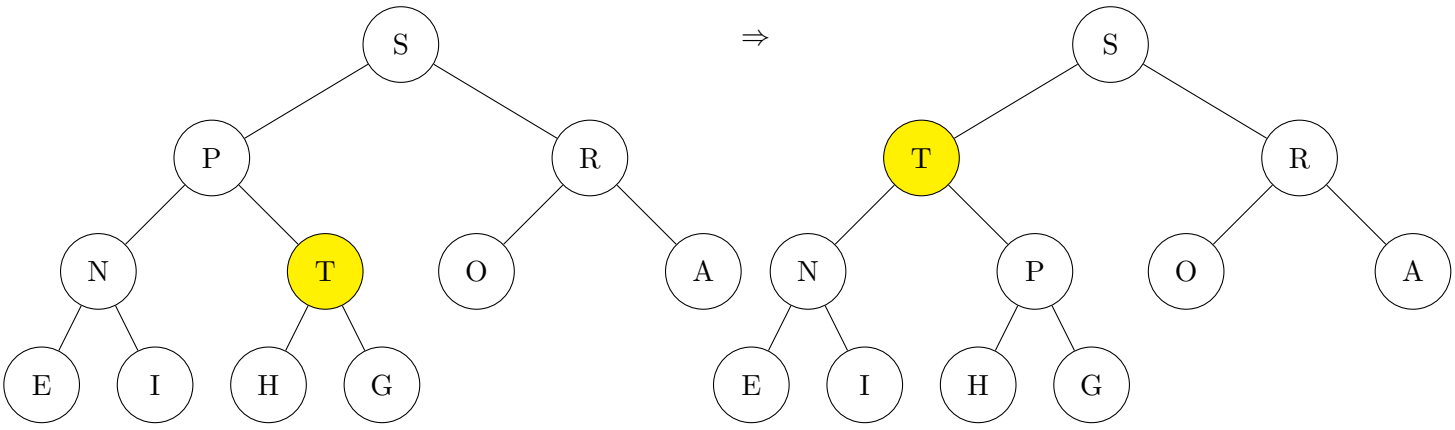


Abbildung 10: Herstellen der Heap-Eigenschaft durch swim_up(T) im ersten Schritt

Beispiel 19. *Dieser Baum ist noch kein Heap, da der Knoten T weiterhin größer ist als S. Folglich wiederholen wir das Vorgehen, für alle Knoten, bis die alle Beziehungen zwischen Kindern und Eltern passen.*

Wir implementieren die Methode `swim_up` wie folgt:

```
void swim_up(A[], int k) {
        while (k > 1 && A[k/2] < A[k]) {
                exchange(A[k], A[k/2]);
                k = k / 2;
        }
}
```

- **Szenario 2:** Der Schlüssel eines Elternknotens ist kleiner als der Schlüssel eines/beider Kinder: Dann vertauschen wir den Schlüssel des Elternknotens mit dem Schlüssel des größeren Kindes.

```
void sink(A[], int k, int heap_size) {
        while (2 * k <=  heap_size) {
                int j = 2 * k;
                if (j <= n && A[j] < A[j+1]) {
                        j++;
                }
                if (A[k] !< A[j]) {
                        break;
                }
                exchange (A[k], A[j]);
                k = j;
        }
}
```

20. Einfügen eines Elementes

Das Einfügen eines Elementes in einen Heap erfolgt zunächst an der nächsten freien Position im Array. Also falls wir einen bestehenden Heap der Länge 11 betrachten, fügen wir das Element an der Position 12 ein. Anschließend wird die Heap-Eigenschaft durch `swim_up` wiederhergestellt.

Die Komplexität des Einfügens liegt in $O(log(n))$.

Implementierung:

```
void insert(A[], key_type Element) {
        heap_size(A)++;
        A[heap_size(A)] = Element;
        swim_up(A, heap_size);
}
```

21. Löschen des größten Elements

In einem Heap ist das Löschen zunächst nur für das größte Element (also im Max-Heap die Wurzel) definiert. Wir tauschen die Wurzel mit dem letzten Element im Heap, erhalten dadurch eine neue Wurzel. Das neue letzte Element kann nun einfach entfernt werden, indem wir die Größe des Arrays um 1 verringern. Nun ist abschließend noch die Heap-Eigenschaft durch Absinken lassen der neuen Wurzel herzustellen.

Auch diese Operation verläuft in $O(log(n))$.

Implementierung:

```
key_type del_max(A[]) {
        key_type max = A[1];
        exchange(A[1], A[heap_size(A)]);
        heap_size(A)--;
        sink(A, 1, heap_size(A));
        return max;
}
```

22. Heapsort

Nun lässt sich auf Heaps auch ein Suchalgorithmus definieren, den wir im Folgenden betrachten und anwenden wollen. Auch wenn wir Heaps zunächst als Baumstruktur eingeführt haben, wird von Ihnen in Klausuren verlangt werden, die Konzepte und Methoden – also auch Heapsort – auf bloße Arrays anzuwenden. Versuchen Sie daher, sich die Beziehung zwischen Heaps als Bäumen und Heaps als Array zu verinnerlichen.

Idee Ausgehend von einem ungeordneten Feld, stellen wir zunächst die Heap-Eigenschaft her. Nun steht – zumindest bei Max-Heaps – der größte Schlüssel in der Wurzel bzw. an Index 1. Nun tauschen wir den Schlüssel der Wurzel mit dem Schlüssel an der letzten Stelle des Feldes. Nun stellen wir die Heap-Eigenschaft über sink wieder her und tauschen die neu gewonnene Wurzel mit dem letzten Feldelement, solange noch Schlüssel ungeordnet sind. Die Anzahl der noch ungeordneten Schlüssel erhalten wir dadurch, dass wir das Feld nach jeder Iteration verkürzen.

Implementation als Pseudo-Code

```
void heapsort(A[]) {
        int n = A.length;
        for (int k = n/2; k >= 1; k--) {
                sink(A, k, n);
        }
        while (n > 1) {
                exchange(A[1], A[n]);
                n--;
                sink(A, 1, n);
        }
}
```

Beispiel 20. *Betrachten wir das ungeordnete Array* $\left[\begin{array}{cccccccc} 6 & 9 & 1 & 8 & 2 & 3 & 5 & 4 \end{array}\right]$.
Zunächst stellen wir die Heap-Eigenschaft her, in dem wir `sink()` *aufrufen.*

$$\left[\begin{array}{cccccccc} 6 & 9 & 1 & 8 & 2 & 3 & 5 & 4 \end{array}\right]$$
$$\left[\begin{array}{cccccccc} 6 & 9 & 5 & 8 & 2 & 3 & 1 & 4 \end{array}\right]$$
$$\left[\begin{array}{cccccccc} 9 & 6 & 5 & 8 & 2 & 3 & 1 & 4 \end{array}\right]$$
$$\left[\begin{array}{cccccccc} 9 & 8 & 5 & 6 & 2 & 3 & 1 & 4 \end{array}\right]$$

Nun beginnen die Vertauschungsoperationen:

$$\left[\begin{array}{cccccccc} 4 & 8 & 5 & 6 & 2 & 3 & 1 & 9 \end{array}\right]$$
$$\left[\begin{array}{ccccccc|c} 8 & 4 & 5 & 6 & 2 & 3 & 1 & 9 \end{array}\right]$$
$$\left[\begin{array}{ccccccc|c} 8 & 6 & 5 & 4 & 2 & 3 & 1 & 9 \end{array}\right]$$
$$\left[\begin{array}{ccccccc|c} 1 & 6 & 5 & 4 & 2 & 3 & 8 & 9 \end{array}\right]$$
$$\left[\begin{array}{cccccc|cc} 6 & 1 & 5 & 4 & 2 & 3 & 8 & 9 \end{array}\right]$$
$$\left[\begin{array}{cccccc|cc} 6 & 4 & 5 & 1 & 2 & 3 & 8 & 9 \end{array}\right]$$
$$\left[\begin{array}{cccccc|cc} 3 & 4 & 5 & 1 & 2 & 6 & 8 & 9 \end{array}\right]$$
$$\left[\begin{array}{ccccc|ccc} 5 & 4 & 3 & 1 & 2 & 6 & 8 & 9 \end{array}\right]$$
$$\left[\begin{array}{ccccc|ccc} 2 & 4 & 3 & 1 & 5 & 6 & 8 & 9 \end{array}\right]$$
$$\left[\begin{array}{cccc|cccc} 4 & 2 & 3 & 1 & 5 & 6 & 8 & 9 \end{array}\right]$$
$$\left[\begin{array}{cccc|cccc} 1 & 2 & 3 & 4 & 5 & 6 & 8 & 9 \end{array}\right]$$
$$\left[\begin{array}{ccc|ccccc} 3 & 2 & 1 & 4 & 5 & 6 & 8 & 9 \end{array}\right]$$
$$\left[\begin{array}{ccc|ccccc} 1 & 2 & 3 & 4 & 5 & 6 & 8 & 9 \end{array}\right]$$
$$\left[\begin{array}{cc|cccccc} 2 & 1 & 3 & 4 & 5 & 6 & 8 & 9 \end{array}\right]$$
$$\left[\begin{array}{c|ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 8 & 9 \end{array}\right]$$

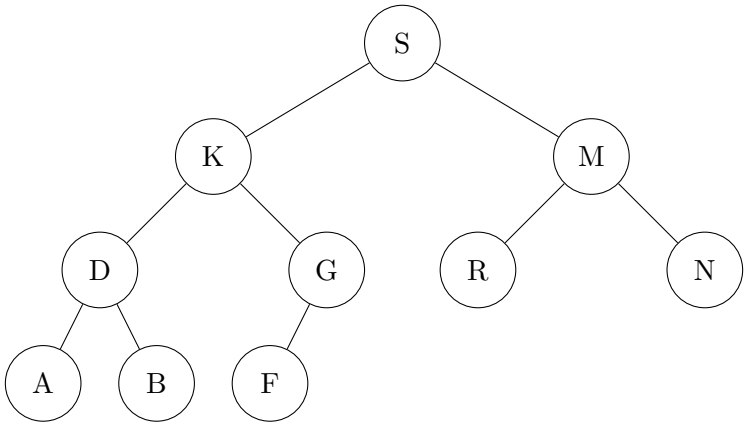
Im letzten Schritt haben wir das gewünschte, aufsteigend sortierte Array erhalten.

Komplexität Wie alle anderen Sortieralgorithmen ist auch Heapsort nicht laufzeiteffizienter als $O(n \cdot log(n))$. Für das Herstellen der Heap-Eigenschaft erhalten wir eine Komplexität von $O(n)$, da wir im schlechtesten Fall alle Elemente des Arrays einmal vertauschen müssen. Für die eigentliche Suche erhalten wir $O(n \cdot log(n))$.

Aufgaben

Aufgabe 1 Gegeben ist der folgende Baum.

- Beurteilen und begründen Sie, ob es sich bei diesem Baum um einen Heap handelt.
- Falls es sich nicht um einen Heap handelt, stellen Sie die Heap-Eigenschaft her.
- Fügen Sie das Element X hinzu.



Lösung auf Seite 74

Aufgabe 2 Sortieren Sie den Baum aus Aufgabe 1 mittels `heapsort()` aufsteigend.

Lösung auf Seite 74

Aufgabe 3 Sortieren Sie die folgenden Arrays mittels `heapsort()` aufsteigend.

- $[\ 7 \quad 1 \quad 3 \quad 9 \quad 4 \quad 6 \quad 5 \quad 8\ ]$
- $[\ H \quad S \quad G \quad A \quad V \quad D \quad J \quad K\ ]$

Lösung auf Seite 76

Aufgabe 4 Erläutern Sie, inwiefern anstatt der Funktion `sink()` die Funktion `swim_up()` in `heapsort()` verwendet werden könnte. Geben Sie an, wie sich der Algorithmus ändert und stellen Sie dies in Pseudocode dar.

Lösung auf Seite 77

Teil VI.
Graphentheorie

Nach unserem Ausflug in besondere Bäume (den Heaps) begeben wir uns nun wieder auf eine abstraktere Ebene, diesmal sogar über Bäume hinaus: Im Folgenden betrachten wir Graphen allgemein und gehen auf Themen wie Pfade, Richtungen oder die Suche nach dem kürzesten Weg ein. Sie haben sicherlich eine Intuition, wenn Sie sich durch eine Stadt bewegen, welcher denn nun der kürzeste Weg ist – während ein Computer damit beschäftigt wäre, Optionen durchzuprobieren. Insofern erscheint es sinnvoll, dafür praktikable Algorithmen zu finden und zu behandeln. Doch beginnen wir von vorne.

23. Typisierung und Eigenschaften

Definition 17. *Ein gerichteter Graph ist ein Paar $G = (V, E)$, wobei V eine nichtleere Menge und $E \subseteq V \times V$ ist. V heißt die **Menge der Knoten** und E heißt die **Menge der (gerichteten) Kanten**. Ein Knoten v heißt **benachbart** oder **adjazent** zu v, wenn $(v, w) \in E$. v heißt Anfangspunkt und w heißt Endpunkt der Kante (v, w). Eine Kante (v, v), die also von einem Knoten auf diesen selbst gerichtet ist, bezeichnen wir als **Schleife**.*

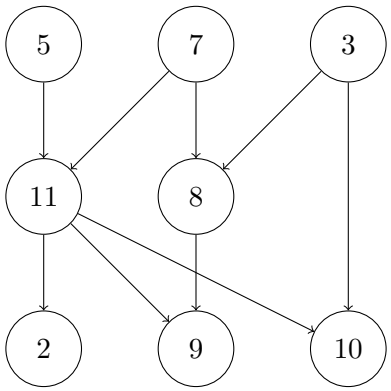


Abbildung 11: Gerichteter Graph

Beispiel 21.

Definition 18. *Bei einem **ungerichteten** Graphen besitzen die Kanten keine Richtung. v und w heißen **Endpunkte** der Kante $e = \{v, w\}$. Die Kante $\{v, w\}$ heißt **inzident** zu v und zu w.*

Zu beiden Typen von Graphen können wir nun Eigenschaften und Kennzahlen definieren:

Definition 19. *Sei $v \in V$.*

$$U(v) := \{w \in V \mid w \text{ adjazent zu } v\}$$

*heißt **Umgebung** von v. Die Anzahl $|U(v)|$ der Elemente von $U(v)$ heißt **Grad** von v, kurz $deg(v)$.*
Die Anzahl der Knoten $|V|$ heißt Ordnung von v.

Damit ergibt sich:

- $|V|$ und $|E|$ messen die Größe eines Graphen. Daher geschehen Aufwandsschätzungen über Graphen stets in Abhängigkeit zu diesen beiden Angaben.
- Einem gerichteten Graphen kann man einem ungerichteten Graphen zuordnen: Kante $(v, w) \rightarrow \text{Kante}\{v, w\}$.
- Dem einem Graphen zugeordneten gerichteten Graphen erhalten wir, wenn wir jede Kante $\{v, w\}$ durch die Kanten $(v, w)\text{und}(w, v)$ ersetzen.

Definition 20. *Sei $G = (V, E)$ ein ungerichteter Graph.*

- *Ein **Pfad** P ist eine Folge von Knoten $v_0, v_1, ..., v_n$ mit der Eigenschaft $\{v_i, v_{i+1}\} \in E, i = 0, ..., n-1$. Dann heißt v_0 **Anfangspunkt** und v_n **Endpunkt** von P. n heißt **Länge** von P.*
- *w heißt von v **erreichbar**, falls es einen Pfad P mit Ausgangspunkt v und Endpunkt w gibt.*

Für gerichtete Graphen definieren wir die Begriffe analog, in einem Pfad sind die Kanten dann gerichtet enhalten.

Satz 6. *Sei $G = (V, E)$ ein Graph ohne Schleifen. Dann sei $n := |V|$ und $m := |E|$.*

- *Für einen Graphen gilt: $0 \leq m \leq \binom{n}{2}$*
- *Für einen gerichteten Graphen gilt: $0 \leq m \leq n \cdot (n-1)$.*

Falls Graphen mit Schleifen betrachtet werden, gilt alternativ:

- *Für einen Graphen gilt: $0 \leq m \leq \binom{n}{2} + n$*
- *Für einen gerichteten Graphen gilt: $0 \leq m \leq n^2$.*

Nun haben wir schon eine Vielzahl von Begrifflichkeiten zu Graphen definiert. Sie finden, das reicht? Nun ...

Definition 21. *Für einen Graphen eines beliebigen Typs definieren wir:*

- *$G = (V, E)$ heißt **vollständig**, wenn jeder Knoten mit jedem anderen Knoten durch eine Kante verbunden ist.*
- *Besitzt G viele Kanten (m groß im Vergleich zu $\binom{n}{2}$ oder $n \cdot (n-1)$), dann heißt G **dicht besetzt**.*
- *Besitzt G wenige Kanten, dann heißt G analog dazu **dünn besetzt**.*

Daraus folgt:

Für einen vollständigen Graphen gilt $|E| = n \cdot (n-1)$ im Fall eines gerichteten Graphen und $|E| = n^2$ für einen ungerichteten Graphen.

24. Adjazenzmatrizen und -listen

„Ein guter Nachbar ist besser als viele böse Verwandten.“
— *Ungarisches Sprichwort*

Falls Sie erinnern, wie wir Adjazenz definiert haben, werden Sie eine Intuition dafür haben, womit wir uns hier beschäftigen wollen: Wir betrachten Nachbarschaftsverhältnisse in Graphen. Dafür nutzen wir zwei Darstellungsformen, die Adjazenzmatrix und die Adjazenzliste. Hierbei handelt es sich tatsächlich nur um verschiedene Darstellungsformen, der Inhalt unterscheidet sich nicht.

Definition 22. *Die **Adjazenzmatrix** ist eine $n \times n$-Matrix mit Koeffizienten aus $\{0, 1\}$.*

$$adm[i,j] := \begin{cases} 1 & \text{für } (i,j) \in E \\ 0 & \text{sonst} \end{cases}$$

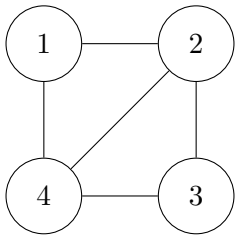
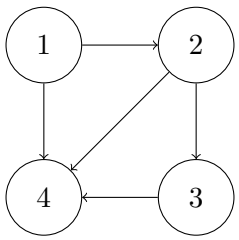


Abbildung 12: Ungerichteter Graph zur Veranschaulichung der Adjazenzmatrix

Beispiel 22. *Wir entnehmen der Darstellung, dass die Knoten (1, 2), (2, 3), (3, 4), (1, 4) und (2, 4) benachbart sind. Also muss an diesen Indizes der Matrix eine 1 stehen, in allen übrigen Fällen entsprechend eine 0.*

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

Betrachten wir nun vergleichsweise diesen Graphen in einer gerichteten Variation:
Hier ergibt sich eine andere Matrix, da die Nachbarschaftsverhältnisse einseitig sind und



wir also für jeden Knoten betrachten, auf welche Knoten er (durch einen so gerichteten Pfeil) „zugreifen“ kann.

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

Nun definieren wir alternativ dazu:

Definition 23. *Die **Adjazenzliste** $adl[1, ..., n]$ ist ein Array von Listen. Für jeden Knoten $j \in V$ ist die Liste $adl[j]$ definiert durch $i \in adl[j]$ genau dann, wenn $(j, i) \in E$ gilt.*

Also sind in $adl[j]$ ebenfalls die zu j adjazenten Knoten verzeichnet.

Beispiel 23. *Für obiges Beispiel ergibt sich nun im Fall des ungerichteten Graphen:*

```
1: 2, 4
2: 1, 3, 4
3: 2, 4
4: 1, 2, 3
```

Und im Fall des gerichteten Graphen:

```
1: 2, 4
2: 3, 4
3: 4
4:
```

Eigenschaften Die Adjazenzmatrix ist speichereffizient für dicht besetzte Graphen (immer $O(n^2)$). Die Suche nach einer bestimmten Kante verläuft in $O(1)$, ist also logischerweise sehr effizient.

Eine Adjazenzliste hat m Einträge bei gerichteten und $2 \cdot m$ Einträge bei ungerichteten Graphen. Für dünn besetzte Graphen ist sie besser geeignet.

25. Suchverfahren in Graphen

Wie in anderen Datenstrukturen auch, sind wir interessiert daran, innerhalb von Graphen Elemente zu suchen. Hierbei brauchen wir aber zunächst – da wir Graphen ganz verschiedenen Aufbaus betrachten – Wege, alle Knoten eines Graphen effizient zu erreichen. Dafür unterscheiden wir **Breitensuche** und **Tiefensuche**. Bei der Breitensuche betrachten wir vorrangig benachbarte Elemente (also Adjazenten) während wir bei der Tiefensuche vorrangig untergeordnete Knoten (also Endknoten eines Pfades) betrachten.

25.1. Breitensuche

Idee

1. Beginnend von einem Startknoten S besuchen wir alle Nachbarn von S, also alle Elemente von $U(S)$.

2. Nun fahren wir fort, in dem wir alle Nachbarn des ersten Nachbarn besuchen ($U(U_1(S))$ für $U_1(S) = 1$. Nachbar des Startknotens, dann alle Nachbarn des zweiten Nachbarn von S usw.

3. Falls wir über diese Vorgehenweise nicht alle Knoten erreichen, setzen wir als Ausgangspunkt einen neuen, nicht besuchten Knoten und fahren solange fort, bis alle Knoten erreicht worden sind.

Implementierung als Pseudo-Code

```
for (v in V) {
        v.marked = false;
}
Q = new Queue();
while exists s in V with (s.marked == false) {
        s = pickStartNode();
        s.marked = true;
        Q.enqueue(s);
        while (!Q.is_empty()) {
                v = Q.dequeue();
                visit(v);
                for (u in v.neighbours) {
                        if (u.marked == false) {
                                u.marked = true;
                                Q.enqueue(u);
                        }
                }
        }
}
```

Wir operieren folglich auf einer `Queue`, also einer Warteschlange, die nach dem FIFO-Prinzip (First in, first out) funktioniert. Für jeden Knoten in der Warteschlange fügen wir alle Nachbarn der Wartschlange hinzu. Die Abfrage, ob ein Knoten besucht wurde oder nicht (also das Markieren) ist erforderlich, um Dopplungen zu vermeiden, die Laufzeit zu optimieren und das Terminieren sicherzustellen.

Wir nennen dieses Verfahren Breitensuche oder **Breadth First Search**-Traversal [**Grundkurs**].

25.2. Tiefensuche

Die Tiefensuche hat zum Ziel, zuerst die Knoten in der Tiefe zu besuchen. Sie heißt daher auch **Depth First Search**-Traversal. Die Grundlage dieses Algorithmus bildet die Datenstruktur `Stack`, auf der nach dem LIFO-Prinzip (Last in, first out) gearbeitet wird.

Idee

1. Beginnend von einem Startknoten S wird der letzte Nachbar von S besucht. Anschließend besuchen wir dessen letzten Nachbarn und gehen soweit in die Tiefe, bis es keinen unbesuchten Nachbarn mehr auf diesem Pfad gibt.

2. Nun kehren wir zum ersten Knoten zurück, bei dem es einen noch unbesuchten Nachbarn gibt und gehen von hier in die Tiefe, in dem wir jeweils den letzten Nachbarn des letzten Nachbarn besuchen.

3. Dieses Verfahren führen wir fort, bis wir alle Nachbarn besucht haben.

Implementierung als Pseudo-Code

```
for (v in V) {
        v. marked = false;
```

```
}
S = new Stack();
while exists s in V with (s.marked == false) {
        s = pickStartNode();
        S.push(s);

        while (!S.is_empty()) {
                v = S.pop();
                if (v.marked == false) {
                        visit(v);
                        v.marked = true;
                        for u in v.neighbours {
                                if (u.marked == false) {
                                        S.push(u);
                                }
                        }
                }
        }
}
```

Wir initialisieren auch hier erneut unseren Graphen, in dem wir alle Knoten als unmarkiert setzen und einen Stack erzeugen. Solange es noch unmarkierte Knoten gibt, wählen wir einen Ausgangsknoten, fügen ihn dem Stapel hinzu und besuchen den ersten Nachbarn, den ersten Nachbarn des ersten Nachbarn, und so weiter. Durch die Stapelstruktur ermöglichen wir, dass keine Elemente vergessen werden und es einen „Weg zurück“ gibt, um die Suche fortsetzen zu können.

In Bezug auf Klausuren ist es wichtig, jeweils Queue/Stack pro Iteration anzugeben. Nur darüber wird deutlich, ob sie die Vorgehensweise wirklich verstanden haben.

Beispiel 24. *Wir betrachten als folgenden Beispielgraphen das Verwandschaftsverhältnis einer typisch saarländischen Familie:*

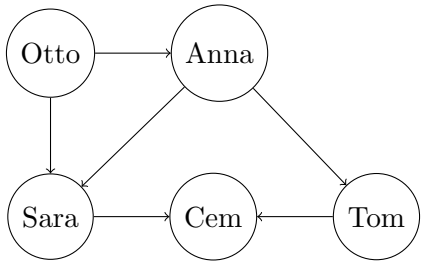


Abbildung 13: Gerichteter Beispielgraph für Breiten-/Tiefensuche

Beginnen wir mit der ***Breitensuche****:*

1. *Als Startknoten wählen wir „Otto“.*
2. *Wir markieren diesen Knoten und fügen ihn der Queue hinzu.*
   `Q = {Otto}`*, durchlaufene Knoten:* `[Otto]`

3. *Wir markieren alle benachbarten Knoten und fügen sie der Queue hinzu.*
   `Q = {Anna, Sara}`, *durchlaufene Knoten:* `[Otto]`

4. *Wir besuchen Anna, löschen sie von der Queue und fügen ihre Nachbarn hinzu (Sara nicht, da dieser Knoten schon markiert wurde).*
   `Q = {Sara, Tom}`, *durchlaufene Knoten:* `[Otto, Anna]`

5. *Wir nehmen das nächste Element vom Stapel, also Sara, besuchen sie und markieren (und enqueuen) alle ihre Nachbarn.*
   `Q = {Tom, Cem}`, *durchlaufene Knoten:* `[Otto, Anna, Sara]`.

6. *Wir nehmen das nächste Element vom Stapel, also Tom, besuchen ihn und markieren alle seine Nachbarn. Da es keinen weiteren unmarkierten Nachbarn gibt, wird der Queue kein weiteres Element hinzugefügt.*
   `Q = {Cem}`, *durchlaufene Knoten:* `[Otto, Anna, Sara, Tom]`.

7. *Nun besuchen wir Cem, stellen fest, dass es keine Nachbarn gibt und tatsächlich auch alle Knoten besucht wurden.*
   `Q = {}`, *durchlaufene Knoten:* `[Otto, Anna, Sara, Tom, Cem]`

Nun betrachten wir als zweites die ***Tiefensuche****:*

1. *Wir wählen als Startelement erneut Otto, fügen ihn dem Stapel hinzu.*
   `S = {Otto}`, *durchlaufene Knoten:* `[]`

2. *Wir nehmen das erste Element vom Stack, besuchen und markieren es. Nun fügen wir alle Nachbarn dem Stack hinzu.*
   `S = {Anna, Sara}`, *durchlaufene Knoten:* `[Otto]`

3. *Wir entfernen das zuletzt hinzugefügte Element vom Stack, besuchen und markieren es. Nun fügen wir alle Nachbarn dem Stack hinzu.*
   `S = {Anna, Cem}`, *durchlaufene Knoten:* `[Otto, Sara]`

4. *Wir fahren entsprechend fort. Da Cem keine Nachbarn hat, wird dem Stack nichts hinzugefügt.*
   `S = {Anna}`, *durchlaufene Knoten:* `[Otto, Sara, Cem]`

5. *Nun betrachten wir Anna, nehmen sie vom Stack und fügen ihren Nachbarn Tom hinzu. Nachbarin Sara ist bereits markiert worden und wird deshalb nicht erneut hinzugefügt.*
   `S = {Tom}`, *durchlaufene Knoten:* `[Otto, Sara, Cem, Anna]`

6. *Nun ist Tom das nächste Element, das wir vom Stack nehmen. Tom hat keinen weiteren unmarkierten Nachbarn. Der Algorithmus terminiert.*
   `S = {}`, *durchlaufene Knoten:* `[Otto, Sara, Cem, Anna, Tom]`

Der geringe Unterschied zwischen Breiten- und Tiefensuche ist der Übungskünstlichkeit geschuldet, in der Praxis ergeben sich grundlegend andere Stacks/Queues und enstprechende Durchlaufreihenfolgen.

26. Gewichtete Graphen

Wir erweitern unsere Definition von Graphen nun, indem wir die Wege zwischen einzelnen Knoten mit Gewichten versehen. So können wir Aussagen darüber treffen, ob ein Nachbar eines Knotens nach kürzerer Strecke erreichbar ist als ein anderer.

Definition 24. *Ein (gerichteter) Graph $G = (V, E)$ mit einer Abbildung $g : E \to \mathbb{R}_{\geq 0}$ heißt **gewichteter** (gerichteter) Graph. Die Abbildung g heißt Gewichtsfunktion.*

*Für $e \in E$ heißt $g(e)$ das **Gewicht von** e. Das Gewicht von G ist die Summe aller Gewichte aller Kanten, also*

$$g(G) = \sum_{e \in E} g(e)$$

Auch zu gewichteten Graphen können wir Adjazenzmatrizen und Adjazenzlisten nutzen:

- In einer **Adjazenzmatrix** zu tragen wir nun anstatt einer 1 bzw. 0 das Gewicht der Kante $g(e)$ ein.
- Auch die Elemente der **Adjazenzliste** erweitern wir um die Komponente des Gewichts jeder Kante:
  Beispiel: $4 : \quad 2 \mid 1,5; 7 \mid 0.8$

Nun interessieren wir uns dafür, den kürzesten Weg von einem Knoten zu einem anderen zu finden. Dass es mehrere mögliche Wege gibt, ergibt sich aus den bestehenden Quervernetzungen. Typischerweise sind die Wege jeweils unterschiedlich gewichtet, sodass sich die Betrachtung tatsächlich lohnt.

26.1. Der Algorithmus von Dijkstra

Der niederländische Informatiekr Edsger W. Dijkstra entwickelte 1959 einen Algorithmus zur Bestimmung der kürzesten Pfade in einem Graphen. Stellen Sie sich vor, Sie sind mit der Bahn unterwegs von Nord nach Süd. Sie müssen mindestens einmal umsteigen und Ihnen stehen verschiedene Strecken zur Auswahl, jeweils mit unterschiedlichen Fahrzeiten (also Gewichten). Ein Weg führt von Hamburg nach Mannheim über Bremen und Bonn, ein weiterer über Hannover und Frankfurt. Ein dritter führt über Berlin und Dresden.

Idee Der Algorithmus operiert wie folgt.

1. Zunächst initialisieren Sie die Suche, indem Sie festhalten, dass der Abstand von ihrem Startknoten zu sich selbst 0 ist und alle weiteren Knoten erst einmal in unerreichbarer Ferne (also unendlich weit entfernt) liegen.
2. Nun wählen Sie den Knoten mit der kleinsten Entfernung – also logischerweise den Startknoten – und bestimmen den Abstand zu allen vom Startknoten aus erreichbaren Knoten. Diese Abstände notieren Sie sich.
3. Aus der Menge der noch nicht bearbeiteten Knoten wählen Sie denjenigen mit dem kürzesten Abstand und aktualisieren nun die Abstände der Knoten, falls es eine Möglichkeit gibt, den Weg mit dem aktuellen Knoten zu verkürzen.
4. So verfahren Sie, bis Sie alle Knoten betrachtet haben. Sie erhalten nun einen sogenannten **Kürzeste-Wege-Baum**.

Definition 25. *Sei $G = (V, E)$ ein gewichteter Graph und $v \in V$. Ermittelt wird ein Pfad minimaler Länge von v nach $w, \forall w \in V$ und es wird ein Baum $T = (V_T, E_T)$ mit Wurzel v konstruiert. Dieser Baum heißt* ***Kürzeste-Wege-Baum****.*

Wenn wir diese Vorgehensweise in Pseudo-Code darstellen wollen, erhalten wir:

Implementation

```
Funktion Dijkstra(Graph, Startknoten):
        initialisiere(Graph, Startknoten, abstand[], vorgaenger[], Q);
        solange Q nicht leer:
                u := Knoten in Q mit kleinstem Wert in abstand[];
                Q.entferne(u);
                falls (U == Startknoten) {
                        T := ({Startknoten}, {});
                }
                sonst:
                        V_T.add(u);
                        E_T.add(vorgaenger[u], u);
                fuer jeden Nachbarn v von u:
                        falls v in Q:
                                distanz_update(u, v, abstand[], vorgaenger[]);
```

Für die Initialisierungsfunktion ergibt sich zusätzlich:

```
initialisiere(Graph, Startknoten) {
        fuer jeden Knoten v in Graph:
                abstand[v] = infinity;
                vorgaenger[v] = undefined;
        abstand[Startknoten] = 0;
}
```

Das Aktualisieren der Distanzen erfolgt nun über folgende Methode:

```
distanz_update(u, v, abstand[], vorgaenger[]) {
        alternativ = abstand[u] + abstand_zwischen(u, v);
        falls alternativ < abstand[v]:
                abstand[v] = alternativ;
                vorgaenger[v] = u;
}
```

Hiermit erreichen wir nun, dass die Distanzen verringert werden, falls es einen kürzeren Weg geben. Am Ende dieses Durchlaufs sind die kürzesten Wege ermittelt.

Beispiel 25. *Gegeben ist der folgende Graph:*

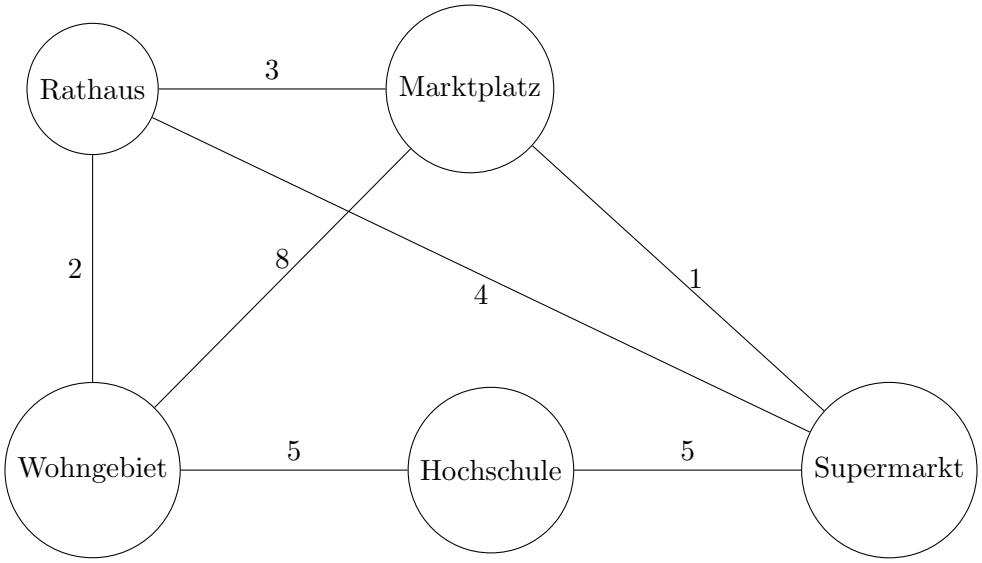


Abbildung 14: Beispielgraph für den Dijkstra-Algorithmus

Wir kürzen die einzelnen Knoten für den Algorithmus nun ab mit RH, MP, WG, HS, SM und wählen das Wohngebiet als Startpunkt des Algorithmus.
Dieses Beispiel ist zugegebenerweise nicht sonderlich spannend, da vom Wohngebiet aus

u	WG	RH	MP	HS	SM	Bemerkung
init	0 | –	∞ | –	∞ | –	∞ | –	∞ | –	Initialisierung
WG		2 | WG	8 | WG	5 | WG		Knoten mit kürzester Distanz: RH
RH			5 | RH		6 | RH	Knoten mit kürzester Distanz: MP
MP						Keine weiteren Knoten verblieben.

Tabelle 3: Beispiel zum Dijkstra-Algorithmus

bereits viele Knoten erreichbar sind und es nur zwei Abkürzungen gibt, die wir finden, nämlich der Weg zum Marktplatz über das Rathaus und der Weg zum Supermarkt über das Rathaus.

Diese Wege lassen sich nun in einen ***Kürzeste-Wege-Baum*** *übertragen.*

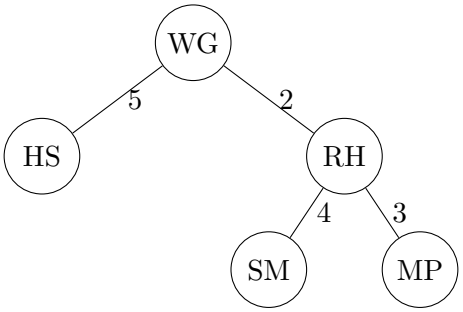
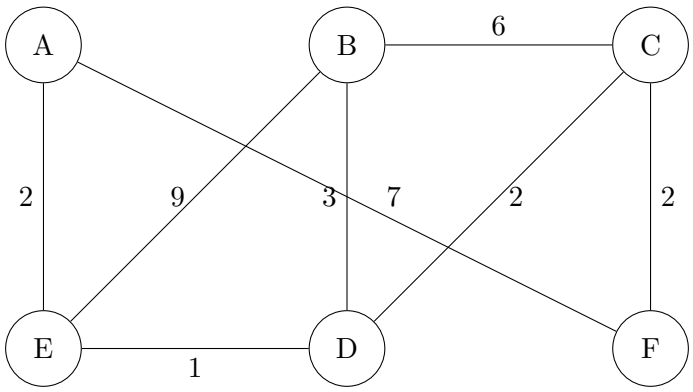


Abbildung 15: Kürzeste-Wege-Baum zum Dijkstra-Beispiel

Aufgaben



Aufgabe 1 Geben Sie zu diesem Graph die Adjazenzmatrix und Adjazenzliste an.

Lösung auf Seite 77

Aufgabe 2 Führen Sie zu diesem Graphen eine Breiten- und Tiefensuche durch. Geben Sie jeweils die Queue/den Stack und die bereits durchlaufenen Knoten an.

Lösung auf Seite 77

Aufgabe 3 Wenden Sie den Dijkstra-Algorithmus zur Berechnung der kürzesten Wege von Knoten B ausgehend auf den Graphen an. Notieren Sie den Kürzeste-Wege-Baum.

Lösung auf Seite 79

Aufgabe 4 Erläutern Sie den Unterschied zwischen Stack und Queue und gehen Sie darauf ein, warum die jeweilige Datenstruktur im BFS- bzw. DFS-Traversal-Algorithmus verwendet wird.

Lösung auf Seite 79

Teil VII.
Hashing

In den vergangenen Abschnitten und Kapiteln haben wir eine Vielzahl von Datenstrukturen kennengelernt: Bäume, Stacks, Queues, Graphen und Arrays. Alle dienten sie dem Zweck, Daten und ihre Beziehungen zueinander zu visualisieren. In diesem letzten Abschnitt widmen wir uns nun wieder **Arrays**, der Datenstruktur, die Grundlage für viele andere ist.

Betrachten wir also eine Menge, beispielsweise jene der natürlichen Zahlen $\mathbb{N}$ oder Mitgliedsstaaten der europäischen Union. Eine Teilmenge davon stellen die Elemente eines Feldes (Arrays) dar. In diesem Array ist die Anordnung zu berücksichtigen, so kann es also durchaus sein, dass die Feldelemente nicht alphabetisch oder ihrer Größe nach geordnet sind. Uns interessiert, an welcher Stelle des Arrays ein gewisses Element seinen Platz gefunden hat.

Vielleicht denken Sie jetzt, dass sei gar kein so großes Problem, schließlich haben wir ja bereits Suchalgorithmen kennengelernt, doch denken Sie daran, dass dieses Feld ruhig beliebig groß sein kann und die Elemente nicht sortiert sind, sodass der Gedanke an die Binäre Suche Sie nicht zum Ziel führt.

Stattdessen suchen wir einen mathematischen Weg, eine Abbildung, die uns den Platz eines Elementes unseres Feldes mitteilt: **Eine Hashfunktion**. Diese Funktion berechnet zu jedem Feldelement einen Speicherort. Wir legen nun eine **Hashtabelle** an, in der je nach Hash-Wert die Ursprungselemente enthalten sind. Diese Tabelle löst unser Problem und ermöglicht es uns, die Elemente zu finden.

Definition 26. *Eine Abbildung $H : K \to A$ mit K = Schlüsselraum und A = Adressraum heißt **Hashfunktion** [**Wirth**].*

Wir stellen an diese Hashfunktion einige Anforderungen:

- Die Abbildung soll effizient zu berechnen sein, also idealerweise mit Grundrechenarten auskommen.
- Die Abbildung soll surjektiv sein, also alle möglichen Hashwerte sollen auch als Ergebnis zustande kommen.
- Die Schlüssel sollen möglichst gleichmäßig auf den Bereich der Indexwerte verteilt werden.

Beispiel 26. *Beispielsweise könnte diese Abbildung eine Hashfunktion sein:*

$$H(k) = ord(k) mod n$$

mit $ord(k)$ = Ordinalzahl des Schlüssels in der Ursprungsmenge.

Nun könnte alles perfekt sein: Jedes Element unseres Feldes wird eindeutig auf einen Hashwert abgebildet, von dort in trivialer Weise gefunden und weiterverwendet. So perfekt ist es leider nicht – es gibt einen großen Haken:

27. Kollisionsvermeidung

Definition 27. *Als **Kollision** bezeichnen wir den Fall, dass zu einem Hashwert mehrere Ausgangswerte existieren.*

Theoretisch könnte es also passieren, dass die Elemente 2 und 17 mit einer gegebenen Hashfunktion den gleichen Hashwert ergeben. Dann können wir beide Werte nicht ohne Weiteres in der Hashtabelle speichern.

Folgende Optionen ergeben sich:

- **Direkte Verkettung** – Anstatt eines Eintrages in der Hashtabelle könnten wir auch eine Menge an Hashwerten als Eintrag speichern. Diese Lösung ist zwar pragmatisch und effizient, birgt jedoch den Nachteil, dass wir eine zusätzliche Liste mit Informationen über diese Sekundärtabelle benötigen, um weiterhin jedes Element finden zu können.
- **Offene Adressierung** – Alternativ suchen wir für unser abzuspeicherndes (also einzutragendes) Element einfach einen anderen, noch freien Platz. Diesen Vorgang nennen wir **Sondieren**. Falls unsere erste Wahl also durch ein anderes Element belegt sein sollte, berechnen wir mittels einer Sondierungsvorschrift einen alternativen Platz und probieren es dort. Falls wir auch dort scheitern sollten, haben wir immer noch weitere Möglichkeiten.

Im Folgenden wollen wir uns näher mit drei Kollisionsvermeidungsstrategien aus dem Bereich der offenen Adressierung beschäftigen.

27.1. Lineares Sondieren

Der triviale Ansatz wäre, im Falle einer Kollision den Hashwert einfach um 1 zu erhöhen. Wir gehen also in unserer Hashtabelle einen Schritt weiter und prüfen dort, ob dieser Hashwert noch frei ist. Falls ja, haben wir das Problem gelöst.

$$\begin{aligned} h[0] &= H(k) \\ h[i] &= (h[0] + 1) \bmod n \end{aligned}$$

Probleme Dieser Ansatz birgt den großen Nachteil, dass sich primäre Cluster bilden. Nehmen wir beispielsweise an, fünf Elemente erhalten durch eine Hashfunktion den Hashwert 8. Nach dieser Methode sind nun – sofern es sich um die ersten gehashten Werte handelt – die Einträge 8 – 12 belegt. Ein sechstes Element ebenfalls mit $h(k) = 8$ müsste nun schon sechs Schritte weitergehen, um einen freien Platz zu finden.

27.2. Quadratisches Sondieren

Ein bisschen erfolgsversprechender erscheint der folgende Ansatz: Hier gehen wir nicht nur 1er-Schritte in eine Richtung, sondern verändert Schrittrichtung und Schrittgröße in jeder Iteration.

$$\begin{aligned} h[0] &= H(k) \\ h[i] &= (h[0] + (-1)^{i+1} \cdot \lfloor (\frac{i+1}{2})^2 \rfloor) \bmod n \\ &= h(k), h(k) + 1^2, h(k) - 1^2, h(k) + 2^2, h(k) - 2^2, ..., h(k) + (\frac{m-1}{2})^2, h(k) - (\frac{m-1}{2})^2 \end{aligned}$$

Probleme Auch dieser Ansatz ist nicht unproblematisch: Zwar treten hier keine primären Cluster auf, dafür aber sekundäre. Für zwei Schlüssel mit demselben Hashwert sind auch die folgenden Positionen in der Hashtabelle gleich.

27.3. Doppeltes Hashing

Wir suchen also eine Lösung, dasssich für Elemente mit gleichem Hashwert unterschiedliche Sondierungsreihenfolgen ergeben. Dies erreichen wir, indem wir eine zweite Hashfunktion konstruieren und verwenden.

$$h[0] = H(k)$$
$$h[i] = (h[0] + i \cdot D(k)) \bmod n$$

Damit ergeben sich nun also unterschiedliche Sondierungsreihenfolgen auch in dem Fall, dass $H(k_1) = H(k_2)$ ist. Mit diesem Vorgehen approximieren wir das ideale Hashing.

Aufgabe

Gegeben seien das Feld

$$\left[\begin{array}{cccccccc} 12 & 15 & 18 & 3 & 5 & 99 & 23 & 1 \end{array}\right]$$

und die Hashfunktionen

$$h(x) = x \bmod 9$$

und

$$d(x) = x + 2 \bmod 13$$

1. Berechnen Sie die Hashtabelle mit $m = 9$ durch **Lineares Sondieren**.
2. Berechnen Sie die Hashtabelle mit $m = 9$ durch **Quadratisches Sondieren**.
3. Berechnen Sie die Hashtabelle mit $m = 9$ durch **Doppeltes Hashing**.

Lösung auf Seite 79

Teil VIII.
Lösungen

28. Grundlagen der Algorithmik

Lösung zu Aufgabe 1 Gesucht sind die Landau-Notationen O, Ω, Θ zur Funktion $3n^3 + 7n^2 + 16$. Zunächst bestimmen wir die O-Komplexitätsklasse: Da der höchste Grad des Polynoms 3 ist und sämtliche konstanten Faktoren oder Summanden entfallen, liegt die Funktion in $O(n^3)$.

Für die Ω-Notation überlegen wir uns Folgendes: Der kleinste Grad des Polynoms ist 0 ($16 \cdot n^0$). Dies ist unsere Unterschranke: Da auch hier konstante Faktoren und Summanden unberücksichtigt bleiben, erhalten wir $\Omega(1)$.

Die Θ-Notation umfasst nun alle Funktionen, die sowohl Teil von $O(n^3)$ als auch von $\Omega(1)$ sind. Dies ist nun keine zeichenbare Funktion, sondern viel mehr der Flächeninhalt zwischen beiden Graphen.

Wir zeichnen nun ein:

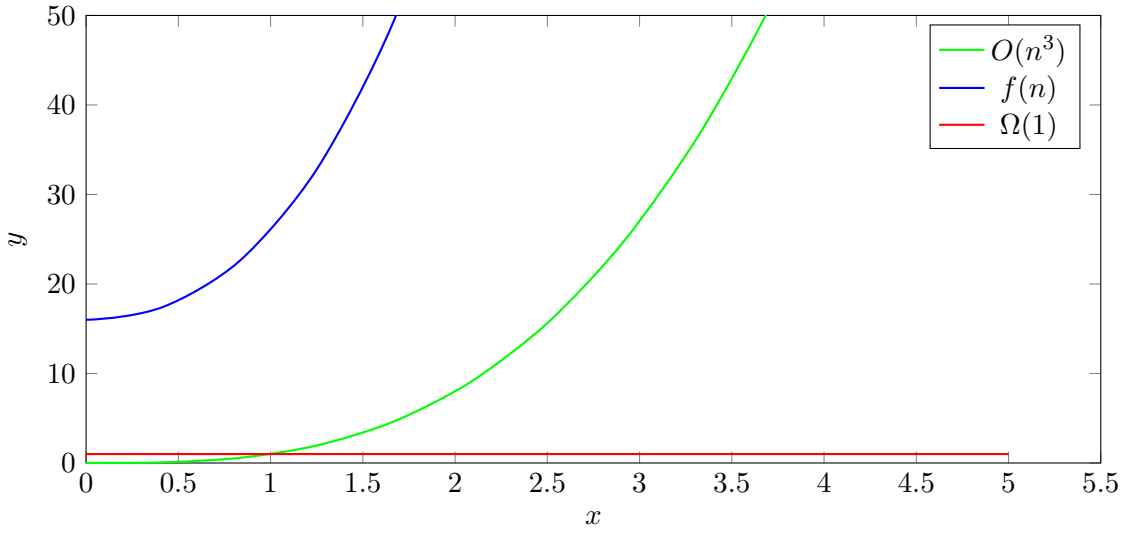


Abbildung 16: Skizze zu Aufgabe 1.1

Lösung zu Aufgabe 2 Anzugeben sind die O-Notationen zu gegebenen Funktionen.

1. $f(n) = 6n^4 + 3 \cdot log_2(n) \in O(n^4)$, da das Wachstum des Terms n^4 im Vergleich zu $log_2(n)$ m größten ist. Konstantglieder entfallen.

2. $f(n) = 6627816n + 13 \in O(n)$, da Konstantglieder immer entfallen und somit n das größte Wachstum verursacht.

3. $f(n) = \frac{72n^3+27n^2+8n+9}{n!} \in \frac{O(n^3}{O(n!)}$. Diese Angabe genügt, ein weiteres Auflösen ist nicht möglich oder sinnvoll. Diese Funktion kann keine typische Komplexität eines Algo-

rithmus sein, da die Funktion gegen 0 konvergiert. Falls Sie sich an dieser Aufgabe die Zähne ausgebissen haben sollten, bitte ich um Vergebung!

4. $f(n) = 3n \cdot \frac{n!}{2} + n^n \in O(n^n)$, da dieser letzte Summand n^n das größte Wachstum aufweist. Es ist damit Teil der letzten Klasse des exponentiellen Wachstums und selbst innerhalb dieser durch $a = n$ besonders schnell wachsend.

Lösung zu Aufgabe 3 Aus gegebener Tabelle der Komplexitätsklassen wissen wir bereits, dass exponentielles Wachstum im Vergleich am schnellsten wächst. Hier ist nun zu zeigen, dass dies für sowohl für n^2 als auch für n^3 gilt, also beide Funktionen Teil der Komplexitätsklasse $O(2^n)$ sind (also nicht schneller wachsen als 2^n).

Dies beweisen wir induktiv:

Beweis. Aus der Definition der O-Notation leiten wir ab, dass wir ein beliebiges n_0 und ein c wählen können. Wir setzen $n_0 = 4$, da wir wissen, dass $4^2 \leq 2^4$ ist. 4 ist die erste natürliche Zahl, ab der dies gilt. Dass kleinere n dies noch nicht erfüllen, ist unerheblich. $c = 1$, da hier kein Konstantfaktor benötigt wird.

$$\begin{aligned} \text{Induktionsanfang} \quad & 4^2 \leq 2^4 \\ \text{Induktionsvoraussetzung} \quad & n^2 \leq 2^n \\ \text{Induktionsschritt} \quad & n \to n+1 \\ \text{Ausmult. + Einsetzen der IV} \quad & (n+1)^2 = n^2 + 2n + 1 \leq 2^n + 2n + 1 \\ & \dots \end{aligned}$$

Der Rest des Beweises ist in den Vorlesungsunterlagen ersichtlich. Für den zweiten Aufgabenteil gehen wir ähnlich vor. □

Lösung zu Aufgabe 4 Hier wenden wir das Master-Theorem an.

- $f(n) = 2 \cdot f(\frac{n}{2n+1}) + n^2$, daraus folgt: $a = 2, b = 2n+1, d = 2$. Nun prüfen wir, ob gilt:
$$a < b^d \to 2 < (2n+1)^2 \to 2 < 4n^2 + 2n + 1$$
Offensichtlich gilt die Aussage und somit tritt Fall 1 des Mastertheorems ein und wir erhalten: $O(n^2)$.

- $g(n) = log(n) \cdot g(\frac{n}{2}) + 3$, also: $a = log(n), b = 2, d = 0$. Wir prüfen, ob die Bedingung des ersten Falls des Master-Theorems gilt:
$$a < b^d \to log(n) < 2^0 \to log(n) < 1$$
Offensichtlich gilt diese Bedingung nicht, da $log(n)$ für alle $n \geq 100$ größer ist als 1. Also tritt Fall 3 des Mastertheorems ein und wir erhalten: $O(n^{log_2(log(n))})$.

- $h(n) = sin(n) \cdot h(\frac{n}{\frac{n}{2}}) + O(n^3)$, also $a = sin(n), b = \frac{n}{2}, d = 3$. Wir prüfen erneut:
$$a < b^d \to sin(n) < (\frac{n}{2})^3$$
Da $sin(n)$ nur Werte zwischen -1 und 1 annimmt, ist die Bedingung für hinreichend große n erfüllt. Damit erhalten wir: $O(n^3)$.

Auch bei der Anwendung des Mastertheorems ist für uns die langfristige Entwicklung des Wachstums relevant. Daher sind kleine n, die einen Widerspruch darstellen würden, unerheblich.

Lösung zu Aufgabe 5 Wir betrachten die Anweisungen zunächst einzeln:

`int[] result;`	$O(1)$	Speicher wird reserviert
`for (dataElement in data)`	$O(n)$	In Abhängigkeit zu n
`if (...)`	$O(1)$	konstanter Vergleich
`result.append(dataElement`	$O(1)$	pro Iteration konstant, wegen Obergrenzenbetrachtung fällt Bedingung nicht ins Gewicht
`return result;`	$O(1)$	konstante Ausgabe

Wir addieren nun alle Komplexitäten und erhalten $O(n + 4) \in O(n)$ als Gesamtkomplexität dieses Algorithmus.

Lösung zu Aufgabe 6 Hier finden wir drei ineinander verschachtelte For-Schleifen:

1. Der Iterator der ersten For-Schleife wird in jedem Durchlauf verdoppelt und beginnt bei $i = 0$ bricht ab, sobald $i \geq n$. Wir suchen nun also ein x mit $2^x = n$. Die Gleichung lösen wir durch den Logarithmus und erhalten entsprechend $O(log_2(n))$.

2. In der zweiten Schleife halbieren wir j in jedem Durchlauf. j initialisieren wir mit n und lassen die For-Schleife laufen, bis $j \leq 0$. Also gehen wir im Prinzip wie in der ersten Schleife vor, bloß in umgekehrter Richtung. Dennoch erhalten wir auch hier eine Komplexität von $O(log_2(n))$.

3. In der dritten For-Schleife lassen wir k von 1 bis n laufen und inkrementieren k in jeder Iteration um 1. Also finden hier insgesamt $n - 1$ Durchläufe statt, was uns verdächtig stark an $O(n)$ erinnert, womit wir auch goldrichtig liegen.

Nun multiplizieren wir die Komplexitäten und erhalten die Gesamtkomplexität $O(n \cdot log_2(n)^2)$.

Lösung zu Aufgabe 7 Wir erkennen, dass es sich um einen rekursiv implementierten Algorithmus handelt, da die Funktion sich zweimal selbst aufruft. Daher findet hier das **Mastertheorem** Anwendung, da es sich um ein „Divide and Conquer"-Problem handelt.

- Es finden zwei Rekursive Aufrufe statt, also setzen wir $a = 2$.
- In jedem Rekursiven Aufruf halbieren wir die Datenmenge, also gilt $b = 2$.
- Der Aufwand für das Zusammenfügen ist konstant, damit unerheblich und liegt daher in $O(1)$.

Wir setzen zusammen und stellen die Gleichung für das Mastertheorem auf:

$$T(n) = 2 \cdot T(\frac{n}{2}) + O(n^0)$$

Nun tritt der dritte Fall des Mastertheorems ein, da $2 > 2^0 = 1$ gilt. Wir erhalten als Gesamtkomplexität $O(n^{log_2(2)}) = O(n)$.

29. Suchalgorithmen

Lösung zu Aufgabe 1 Wir teilen das Feld rekursiv und vergleichen mit dem mittleren Element (im Zweifel das kleinere).
$[\ 1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6\ \ 7\ \ 8\ \ 9\ \ 10\ \ 11\ \ 12\ \ 13\ \ 17\ \ 18\ \ 19\ \ 22\ ]$

$7 < 9$, also betrachten wir den linken Teil:
$[\ 1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6\ \ 7\ \ 8\ ]$

$7 > 4$, also betrachten wir den rechten Teil:
$[\ 5\ \ 6\ \ 7\ \ 8\ ]$

$7 > 6$, also betrachten wir den rechten Teil:
$[\ 7\ \ 8\ ]$

$7 = 7$, wir haben das Element gefunden.

Lösung zu Aufgabe 2 Randbestimmung für den KMP-Suchalgorithmus.

Zeichenkette	Rand
∅	-1
O	0
OT	0
OTT	0
OTTO	1
OTTOS	0
OTTOSM	0
OTTOSMO	1
OTTOSMOP	0
OTTOSMOPS	0
OTTOSMOPSK	0
OTTOSMOPSKO	1
OTTOSMOPSKOT	2
OTTOSMOPSKOTZ	0
OTTOSMOPSKOTZT	0

Zeichenkette	Rand	Bemerkung
∅	-1	
A	0	
AN	0	
ANA	1	
ANAN	2	
ANANA	3	A darf sowohl zum Pre- als auch Postfix gehören
ANANAS	0	
ANANASB	0	
ANANASBA	1	
ANANASBAN	2	
ANANASBANA	3	
ANANASBANAN	4	
ANANASBANANA	5	

Zeichenkette	Rand
$\emptyset$	-1
B	0
BO	0
BON	0
BONO	0
BONOB	1
BONOBO	2

Anders als in einer vorherigen Version dieser Datei angegeben, ist eine Bewertung der Ränder für den KMP-Algorithmus nicht sinnvoll. Durch das Zusammenspiel von Rand und Anzahl an Übereinstimmungen werden diese größeren Sprünge ermöglicht. Tatsächlich ist bei einer hohen Anzahl von Übereinstimmungen ein großer Rand eher hinderlich – der Sprung minimiert sich so, weil schon früher ein neues Vorkommen des Musters möglich wäre.

Lösung zu Aufgabe 3 Wir suchen das Wort **Bonobo** im Suchraum **Beinahebotnochdasbonobonobohausplatz**. Dafür nutzen wir die Randbestimmung aus der oberen Aufgabe.

An dieser Stelle verzichte ich auf eine tabellarische Darstellung wie im Beispiel und stelle die Schritte im Folgenden in Textform dar:

1. Wir legen den Suchtext unter das Muster und stellen fest, dass der erste Buchstabe **B** übereinstimmt, der zweite jedoch schon nicht mehr. Nun ergibt sich durch die Formel

   Neuer Beginn = Suchtextposition+(Übereinstimmungen−Randlaenge[Musterindex])

   der neue Beginn $1 = 0 + (1 - 0)$.

2. An Position 1 tritt keine Übereinstimmung auf, also $2 = 1 + (0 - (-1))$, wir verschieben unser Muster also an Position 2.

3. Auch an Position 2 tritt keine Übereinstimmung auf: $3 = 2 + (0 - (-1))$. So gehen wir vor, bis unser Muster an Index 7 steht.

4. Hier erkennen wir, dass das Muster **Bo** übereinstimmt. Es ergibt sich: $9 = 7 + (2 - 0)$. Von Position 9 an verschieben wir das Muster weiter in 1er-Schritten, bis wir bei **bonobonobohausplatz** ankommen, also an Index 17. **Hier ist das erste Vorkommen des Musters.** Wir verschieben zur Position $21 = 17 + (6 - 2)$, weil hier theoretisch ein weiteres Vorkommen beginnen könnte.

5. Wir sehen: Das tut es auch. An Position 21 erhalten wir also das zweite Vorkommen des Musters. Wir verschieben zu $25 = 21 + (6 - 2)$.

6. Hier stimmen die ersten beiden Buchstaben überein, also gilt $27 = 25 + (2 - 0)$.

7. An Position 27 stimmen keine Buchstaben überein, wir rücken also wieder um jeweils einen Schritt nach vorne bis zum Index 31, weil ab dort die Anzahl der verbleibenden Zeichen geringer ist als die Länge des Musters.

Wir haben nun also zwei Vorkommen unseres Musters im Text ermittelt.

Lösung zu Aufgabe 4 Hier sind selbstverständlich viele Lösungen richtig. Beispiel:
Das Verfahren der binären Suche operiert auf einem bereits sortierten Suchraum und halbiert diesen in jedem Rekursionsaufruf. Anschließend wird nur der Teil weiter betrachtet, in dem das Element auf Basis eines Vergleichs mit dem mittleren Element zu erwarten wäre. Der Algorithmus terminiert, wenn das Element gefunden wurde oder der betrachtete Suchraum einelementig ist.

30. Sortieralgorithmen

Lösung zu Aufgabe 1

- **Insertion Sort**
  1. `[56, 10, 15, 98, 99, 12, 30, 80]`: $56 > 10$, also vertauschen..
  2. `[10, 56, 15, 98, 99, 12, 30, 80]`: $10 < 15 < 56$, also 15 zw. 10 und 56 einfügen.
  3. `[10, 15, 56, 98, 99, 12, 30, 80]`: $10 < 15 < 56 < 98$, keine Änderung.
  4. `[10, 15, 56, 98, 99, 12, 30, 80]`: $10 < 15 < 56 < 98 < 99$
  5. `[10, 15, 56, 98, 99, 12, 30, 80]`: $10 < 12 < 15$, also 12 zw. 10 und 15 einfügen.
  6. `[10, 12, 15, 56, 98, 99, 30, 80]`: $15 < 30 < 56$, also 30 zw. 15 und 56 einfügen.
  7. `[10, 12, 15, 30, 56, 98, 99, 80]`: $56 < 80 < 98$, also 80 zw. 56 und 98 einfügen.
  8. `[10, 12, 15, 30, 56, 80, 98, 99]`: Fertig, Algorithmus terminiert.
- **Bubblesort**
  1. `[56, 10, 15, 98, 99, 12, 30, 80]`: $56 > 10$, also vertauschen.
  2. `[10, 56, 15, 98, 99, 12, 30, 80]`: $56 > 15$, also vertauschen
  3. `[10, 15, 56, 98, 99, 12, 30, 80]`: $56 < 98$, betrachten wir 98.
  4. `[10, 15, 56, 98, 99, 12, 30, 80]`: $98 < 99$, betrachten wir 99.
  5. `[10, 15, 56, 98, 99, 12, 30, 80]`: $99 > 12$, wir vertauschen.
  6. `[10, 15, 56, 98, 12, 99, 30, 80]`: $99 > 30$, wir vertauschen.
  7. `[10, 15, 56, 98, 12, 30, 99, 80]`: $99 > 80$, wir vertauschen.
  8. `[10, 15, 56, 98, 12, 30, 80, 99]`: Ende des Feldes erreicht, da vertauscht wurde, beginnen wir erneut.
  9. `[10, 15, 56, 98, 12, 30, 80, 99]`: $10 < 15 < 56 < 98$, aber $98 > 12$ (jeweils einzeln!).
  10. `[10, 15, 56, 12, 98, 30, 80, 99]`: $98 > 30$, vertauschen.
  11. `[10, 15, 56, 12, 30, 98, 80, 99]`: $98 > 80$, vertauschen.
  12. `[10, 15, 56, 12, 30, 80, 98, 99]`: $98 < 99$, nächste Iteration.
  13. `[10, 15, 56, 12, 30, 80, 98, 99]`: ..., $56 > 12$, vertauschen.
  14. `[10, 15, 12, 56, 30, 80, 98, 99]`: $56 > 30$, vertauschen.
  15. `[10, 15, 12, 30, 56, 80, 98, 99]`: ..., nächste Iteration.

16. `[10, 15, 12, 30, 56, 80, 98, 99]`: $15 > 12$, vertauschen.
17. `[10, 12, 15, 30, 56, 80, 98, 99]`: ..., nächste Iteration.
18. `[10, 12, 15, 30, 56, 80, 98, 99]`: ..., Fertig, Algorithmus terminiert (keine Vertauschungen in dieser Iteration).

- **Quicksort**
  1. `[56, 10, 15, 98, 99, 12, 30, 80]`: 56 ist 1.Element, alle $p < 56$ vor 56 und alle $p > 56$ hinter 56 (unsortiert).
  2. `[10, 15, 12, 30, 56, 98, 99, 80]`: Wir betrachten 10 und 98 und verfahren ebenso.
  3. `[10, 15, 12, 30, 56, 80, 98, 99]`: Wir betrachten 15 und 80 und verfahren ebenso.
  4. `[10, 12, 15, 30, 56, 80, 98, 99]`: Fertig, Algorithmus terminiert.
- **Mergesort**
  1. `[56, 10, 15, 98, 99, 12, 30, 80]`: Zerteilen in Hälften.
  2. `[56, 10, 15, 98][99, 12, 30, 80]`: Zerteilen.
  3. `[56, 10][15, 98][99, 12][30, 80]`: Zerteilen und verbinden.
  4. `[56][10][15][98][99][12][30][80]`: Sortieren und verbinden.
  5. `[10, 56][15, 98][12, 99][30, 80]`: Sortieren und verbinden.
  6. `[10, 15, 56, 98][12, 30, 80, 99]`: Sortieren und verbinden.
  7. `[10, 12, 15, 30, 56, 80, 98, 99]`: Fertig, Algorithmus terminiert.

Lösung zu Aufgabe 2 Auch hier sind natürlich wieder mehrere Lösungen richtig.

Beim Quicksort-Algorithmus wählen wir ein Vergleichselement (Pivot) und richten unser Array zunächst an diesem Element aus, d.h. alle kleineren Elemente notieren wir links dieses Pivots und alle größeren rechts. Nun suchen wir in dieser linken bzw. rechten Teilmenge wieder je ein Pivot und ordnen entsprechend um. Das wiederholen wir solange, bis die aufgespannten Mengen nur ein Element umfassen. Dann ist das Feld/die Menge sortiert.

Lösung zu Aufgabe 3 Anzugeben ist, mit welchem Algorithmus die Felder jeweils am besten zu sortieren sind.

- Das Feld `[1, 9, 2, 8, 3, 7, 4, 6, 5]` ist über Insertion Sort am effizientesten zu sortieren, da wir in diesem Feld ein Muster erkennen: Alle Elemente mit geraden Index-Positionen (1, 2, 3, 4, 5) stehen für sich jeweils in einer richtigen Reihenfolge während alle Elemente auf geraden Index-Positionen (9, 8, 7, 6) in inverser Reihenfolge stehen. Würden wir Bubblesort verwenden, gäbe es einige Iterationen mit Vertauschungen, sodass sich der Aufwand insgesamt erhöht. Für Mergesort und Quicksort ist der Mehraufwand höher als der Nutzen, da die Datenmenge vergleichsweise klein und unkomplex ist.
- Das Feld `[9, 7, 7, 2, 5, 4, 7, 3, 1]` wäre aufgrund der häufig vertretenen 7 für einen modifizierten Quicksort optimal (3-Way-Partitioning). Die 7 würde im zweiten rekursiven Aufruf Pivot werden und damit wären sehr schnell viele Elemente abgearbeitet.

- Das Feld `[9, 8, 7, 6, 5, 4, 3, 2, 1]` liegt in der genau umgekehrten Reihenfolge vor und sollte daher unbedingt mit einem rekursiven Verfahren sortiert werden. Hier bietet sich Mergesort an, da bei Quicksort die Pivot-Elemente sehr ungünstigerweise jeweils die größten wären. Mergesort ist durch die Parallelität optimal.

Lösung zu Aufgabe 4 Bei Mergesort und Quicksort handelt es sich um rekursiv operierende Verfahren, die parallel bearbeitet werden. Hierfür bietet sich die Darstellung als Baum an: Für jede Menge, die halbiert wird, zeichnen wir ein Kindelment. Im Fall Mergesort spiegeln wir die Struktur des Baums im Mergeschritt und erhalten als Blatt die sortierte Gesamtmenge.

31. Bäume

Lösung zu Aufgabe 1 Zu dem gegebenen Baum sind zunächst Definitionen anzuwenden und anschließend zwei Operationen auszuführen.

- Die Wurzel ist der Knoten F.
- Die Höhe des Baums beträgt 3.
- Der Knoten E hat die Höhe 0.
- Die Ordnung des Baumes beträgt 2.
- Die Balance des Knotens K beträgt 1.

Nun ist der Knoten J hinzuzufügen. Da $F < J$, fahren wir im rechten Teilbaum fort. $J < K$, also betrachten wir den linken Teilbaum von K mit H als Wurzel. $J > H$, also betrachten wir den rechten Teilbaum von H, der leer ist, also fügen wir dort als neuen Knoten J ein.

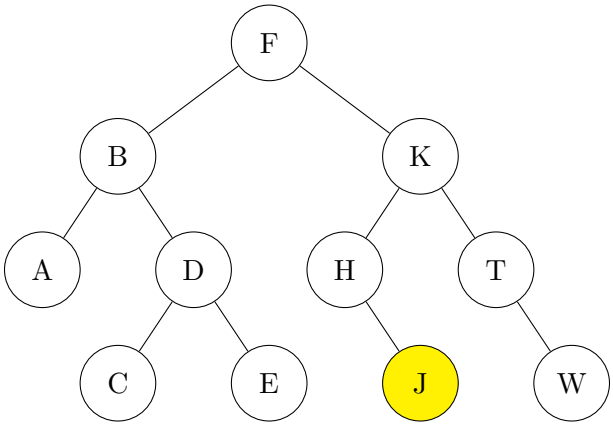


Abbildung 17: Baum aus Aufgabe 1 nach Einfügen des Knotens J

Abschließend löschen wir den Knoten B. Wir stellen fest, dass B zwei Kinder hat. Also ersetzen wir B durch den kleinsten Schlüssel des rechten Teilbaums, also in unserem Fall C. Wir erhalten:

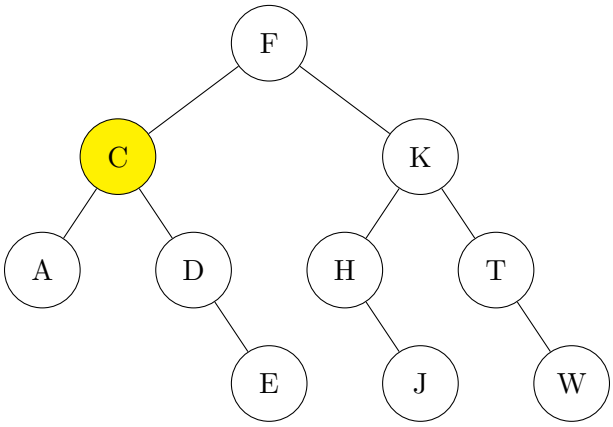


Abbildung 18: Baum aus Aufgabe 1 nach Löschen des Knotens B

Lösung zu Aufgabe 2 Wir bestimmen zunächst die Balancen jedes Knotens.

- Knoten A, C, E, H, W sind Blätter und haben daher die Balance 0.
- Der Knoten D hat die Balance 0.
- Der Knoten T hat die Balance 1.
- Die Knoten B, K haben die Balance 1.
- Der Knoten F hat die Balance 0.

Folglich ist die Balance an keinem Knoten des Baumes größer als 1, womit dieses Kriterium erfüllt ist. Betrachten wir nun, ob es sich überhaupt um einen binären Suchbaum handelt. Da alle Knoten maximal zwei Kinder haben, ist der Baum binär. Dass es sich um einen Suchbaum handelt, ergibt sich aus der Anordnung der Knoten. **Also handelt es sich bei diesem Baum um einen AVL-Baum.**

Lösung zu Aufgabe 3 Da wir festgestellt haben, dass es sich um einen AVL-Baum handelt, können wir direkt beginnen, den Knoten A zu löschen. Der Knoten A ist ein Blatt, daher kann der entsprechende Zeiger einfach gleich null gesetzt werden.

Dies hat nun zur Folge, dass die Balance am Knoten B 2 beträgt und der Baum daher rotiert werden muss. Dies ist am Knoten D sinnvoll und möglich. Wir erhalten Abbildung 19.

Im nächsten Schritt fügen wir in den letzten Baum, den wir erhalten haben, den Knoten Z hinzu. Da Z der letzte Buchstabe des Alphabets ist, wird dieser Knoten also ein rechtes Kind vom Knoten W. Damit verliert der Baum erneut die AVL-Eigenschaft und Rotationen werden erforderlich. Wir rotieren den Baum um den Knoten T.

Lösung zu Aufgabe 4 *Beschreibung und Pseudocode siehe Seite 39*
In einem Baum der Ordnung 3 müsste zunächst definiert werden, was diese Kategorie dann inhaltlich aussagt. Dies könnten Knoten sein, die gleich der Wurzel sind – jedoch ergibt sich dadurch keinen Einsatzzweck eine Algorithmenerweiterung, weil nicht zwischen Knoten mit gleichem Wert unterschieden wird und werden sollte. Somit sind Sie gut beraten, nur auf binären Suchbäumen zu operieren (auch wenn es Spezialfälle geben mag).

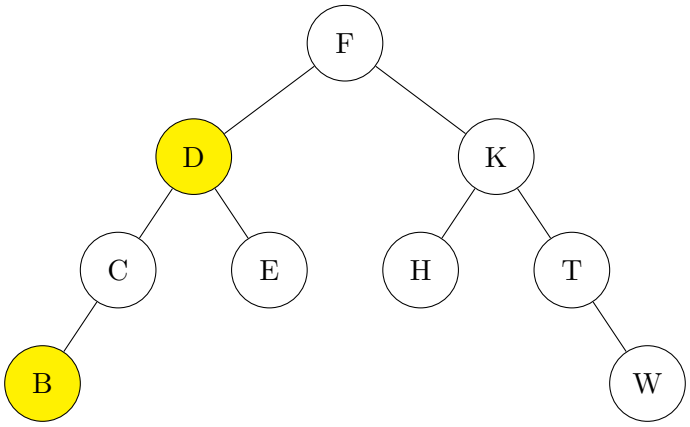


Abbildung 19: AVL-Baum nach Löschen des Knotens A und Rotation um D

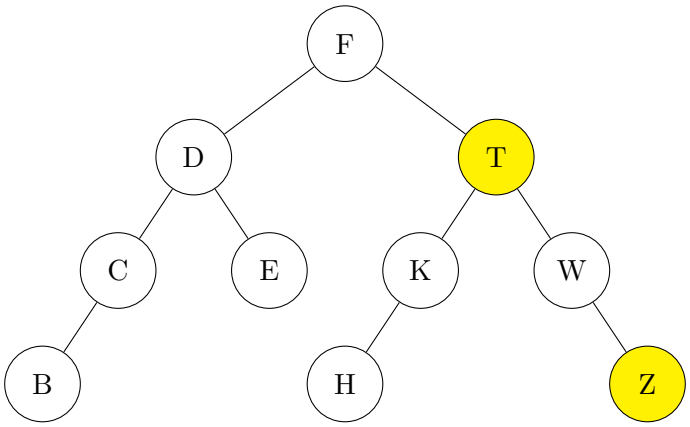


Abbildung 20: AVL-Baum nach Hinzufügen des Knotens Z und Rotation um T

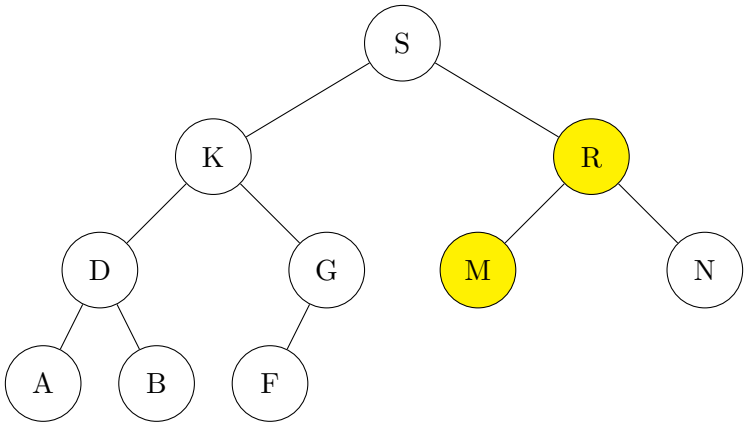
32. Heaps

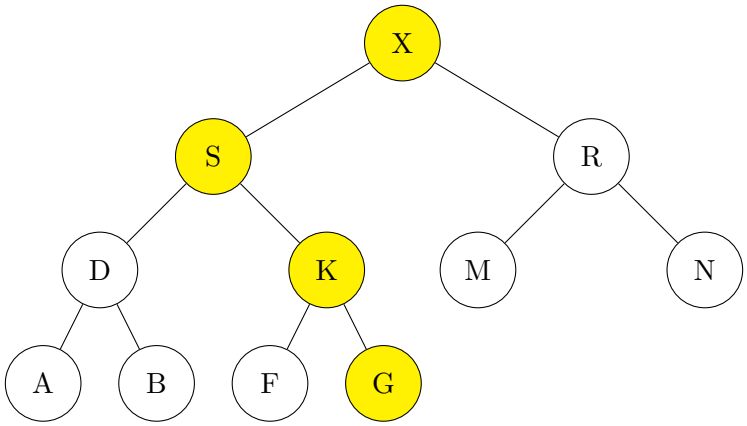
Lösung zu Aufgabe 1 Zu bestimmen ist, ob es sich bei diesem Baum um einen Heap handelt.

1. Es handelt sich nicht um einen Heap, da der Knoten R größer ist als der Knoten M.
2. Um die Heap-Eigenschaft herzustellen, muss folglich der Knoten R „hochschwimmen“. M und R tauschen also.
3. Nun ist der Knoten X hinzuzufügen. Wir fügen ihn an der nächsten freien Position im Heap (bzw. dessen Array) ein, also als rechtes Kind von G. Nun stellen wir fest, dass $G < X$, also führen wir `swim_up(X)` aus.

Aufgabe 2 Zunächst bilden wir den letzten aus Aufgabe 1 gewonnenen Heap als Array ab. Anschließend vertauschen wir erstes und letztes Element und lassen den neuen ersten Knoten absinken.

$$\begin{array}{|cccccccccc|}
\hline
X & S & R & D & K & M & N & A & B & F & G \\
G & S & R & D & K & M & N & A & B & F & X \\
S & G & R & D & K & M & N & A & B & F & \mid X \\
\hline
\end{array}$$





$$
\left[\begin{array}{cccccccccc|c}
S & K & R & D & G & M & N & A & B & F & X \\
F & K & R & D & G & M & N & A & B & S & X \\
R & K & F & D & G & M & N & A & B & S & X \\
R & K & N & D & G & M & F & A & B & S & X \\
B & K & N & D & G & M & F & A & R & S & X \\
N & K & B & D & G & M & F & A & R & S & X \\
N & K & M & D & G & B & F & A & R & S & X \\
A & K & M & D & G & B & F & N & R & S & X \\
M & K & A & D & G & B & F & N & R & S & X \\
M & K & F & D & G & B & A & N & R & S & X \\
A & K & F & D & G & B & M & N & R & S & X \\
K & A & F & D & G & B & M & N & R & S & X \\
K & G & F & D & A & B & M & N & R & S & X \\
B & G & F & D & A & K & M & N & R & S & X \\
G & B & F & D & A & K & M & N & R & S & X \\
G & D & F & B & A & K & M & N & R & S & X \\
A & D & F & B & G & K & M & N & R & S & X \\
F & D & A & B & G & K & M & N & R & S & X \\
B & D & A & F & G & K & M & N & R & S & X \\
D & B & A & F & G & K & M & N & R & S & X \\
A & B & D & F & G & K & M & N & R & S & X \\
B & A & D & F & G & K & M & N & R & S & X \\
A & B & D & F & G & K & M & N & R & S & X \\
A & B & D & F & G & K & M & N & R & S & X
\end{array}\right]
$$

Wir erhalten das fertig sortierte Feld:

$$\left[\begin{array}{cccccccccc} A & B & D & F & G & K & M & N & R & S & X \end{array}\right]$$

Lösung zu Aufgabe 3 Erneut sind zwei Arrays gegeben, die mittels Heapsort aufsteigend sortiert werden sollen. Beim Herstellen der Heap-Eigenschaft wird – eventuell anders als hier dargestellt – bei $\frac{n}{2}$ begonnen.

- $\left[\begin{array}{cccccccc} 7 & 1 & 3 & 9 & 4 & 6 & 5 & 8 \end{array}\right]$

Herstellen der Heap-Eigenschaft:

$$\left[\begin{array}{cccccccc} 7 & 9 & 3 & 1 & 4 & 6 & 5 & 8 \end{array}\right]$$
$$\left[\begin{array}{cccccccc} 7 & 9 & 6 & 1 & 4 & 3 & 5 & 8 \end{array}\right]$$
$$\left[\begin{array}{cccccccc} 7 & 9 & 6 & 8 & 4 & 3 & 5 & 1 \end{array}\right]$$
$$\left[\begin{array}{cccccccc} 9 & 8 & 6 & 7 & 4 & 3 & 5 & 1 \end{array}\right]$$

Nun beginnt das Sortieren:

$$\left[\begin{array}{cccccccc} 1 & 8 & 6 & 7 & 4 & 3 & 5 & 9 \end{array}\right]$$
$$\left[\begin{array}{ccccccc|c} 8 & 1 & 6 & 7 & 4 & 3 & 5 & 9 \end{array}\right]$$
$$\left[\begin{array}{ccccccc|c} 8 & 7 & 6 & 1 & 4 & 3 & 5 & 9 \end{array}\right]$$
$$\left[\begin{array}{ccccccc|c} 5 & 7 & 6 & 1 & 4 & 3 & 8 & 9 \end{array}\right]$$
$$\left[\begin{array}{cccccc|cc} 7 & 5 & 6 & 1 & 4 & 3 & 8 & 9 \end{array}\right]$$
$$\left[\begin{array}{cccccc|cc} 3 & 5 & 6 & 1 & 4 & 7 & 8 & 9 \end{array}\right]$$
$$\left[\begin{array}{ccccc|ccc} 6 & 5 & 3 & 1 & 4 & 7 & 8 & 9 \end{array}\right]$$
$$\left[\begin{array}{ccccc|ccc} 4 & 5 & 3 & 1 & 6 & 7 & 8 & 9 \end{array}\right]$$
$$\left[\begin{array}{cccc|cccc} 5 & 4 & 3 & 1 & 6 & 7 & 8 & 9 \end{array}\right]$$
$$\left[\begin{array}{cccc|cccc} 1 & 4 & 3 & 5 & 6 & 7 & 8 & 9 \end{array}\right]$$
$$\left[\begin{array}{ccc|ccccc} 4 & 1 & 3 & 5 & 6 & 7 & 8 & 9 \end{array}\right]$$
$$\left[\begin{array}{ccc|ccccc} 3 & 1 & 4 & 5 & 6 & 7 & 8 & 9 \end{array}\right]$$
$$\left[\begin{array}{cc|cccccc} 3 & 1 & 4 & 5 & 6 & 7 & 8 & 9 \end{array}\right]$$
$$\left[\begin{array}{cc|cccccc} 1 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array}\right]$$
$$\left[\begin{array}{c|ccccccc} 1 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array}\right]$$

Wir erhalten das fertig sortierte Array:

$$\left[\begin{array}{cccccccc} 1 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array}\right]$$

- $\left[\begin{array}{cccccccc} H & S & G & A & V & D & J & K \end{array}\right]$

Herstellen der Heap-Eigenschaft:

$$\left[\begin{array}{cccccccc} S & H & G & A & V & D & J & K \end{array}\right]$$
$$\left[\begin{array}{cccccccc} S & V & G & A & H & D & J & K \end{array}\right]$$
$$\left[\begin{array}{cccccccc} V & S & G & A & H & D & J & K \end{array}\right]$$
$$\left[\begin{array}{cccccccc} V & S & J & A & H & D & G & K \end{array}\right]$$
$$\left[\begin{array}{cccccccc} V & S & J & K & H & D & G & A \end{array}\right]$$

Nun beginnen wir das Sortieren:

$$\left[\begin{array}{cccccccc} A & S & J & K & H & D & G & V \end{array}\right]$$
$$\left[\begin{array}{ccccccc|c} S & A & J & K & H & D & G & V \end{array}\right]$$
$$\left[\begin{array}{ccccccc|c} S & K & J & A & H & D & G & V \end{array}\right]$$
$$\left[\begin{array}{ccccccc|c} G & K & J & A & H & D & S & V \end{array}\right]$$
$$\left[\begin{array}{cccccc|cc} K & G & J & A & H & D & S & V \end{array}\right]$$
$$\left[\begin{array}{cccccc|cc} K & H & J & A & G & D & S & V \end{array}\right]$$
$$\left[\begin{array}{cccccc|cc} D & H & J & A & G & K & S & V \end{array}\right]$$
$$\left[\begin{array}{ccccc|ccc} J & H & D & A & G & K & S & V \end{array}\right]$$
$$\left[\begin{array}{ccccc|ccc} G & H & D & A & J & K & S & V \end{array}\right]$$
$$\left[\begin{array}{cccc|cccc} H & G & D & A & J & K & S & V \end{array}\right]$$

$$\left[\begin{array}{cccc|cccc} A & G & D & H & J & K & S & V \end{array}\right]$$
$$\left[\begin{array}{cccc|cccc} G & A & D & H & J & K & S & V \end{array}\right]$$
$$\left[\begin{array}{cccc|cccc} D & A & G & H & J & K & S & V \end{array}\right]$$
$$\left[\begin{array}{c|ccccccc} A & D & G & H & J & K & S & V \end{array}\right]$$
$$\left[\begin{array}{c|ccccccc} A & D & G & H & J & K & S & V \end{array}\right]$$

Wir erhalten das sortierte Array:

$$\left[\begin{array}{cccccccc} A & D & G & H & J & K & S & V \end{array}\right]$$

Lösung zu Aufgabe 4 Grundsätzlich lassen sich alle Aufrufe von `sink()` durch `swim_up()` austauschen. Im Falle des Heapsort werden bei jeder Vertauschung kleinere Knoten in die Wurzel eingefügt, also müssen diese Knoten sukzessive nach unten wandern bzw. alle anderen Knoten müssen entsprechend nach oben schwimmen. Wir müssten also für alle in Bezug zur Wurzel stehenden Elemente `swim_up()` aufrufen. Das ist aufwendiger und weniger sinnvoll als `sink()` zu nutzen.

```
void heapsort(A[]) {
        int n = A.length;
        for (int k = n/2; k >= 1; k--) {
                swim_up(A, k, n);
        }
        while (n > 1) {
                exchange(A[1], A[n]);
                n--;
                for (element in A[]) {
                        swim_up(A, A[element], n);
                }
        }
}
```

33. Graphen

Lösung zu Aufgabe 1 Anzugeben sind die Adjazenzmatrix und -liste zum gegebenen Graphen.

$$\begin{pmatrix} - & - & - & - & 2 & 7 \\ - & - & 6 & 3 & 9 & - \\ - & 6 & - & 2 & - & 2 \\ - & 3 & 2 & - & 1 & - \\ 2 & 9 & - & 1 & - & - \\ 3 & - & 2 & - & - & - \end{pmatrix}$$

Und die dazugehörige Adjazenzliste:

A	E: 2, F: 7
B	C: 6, D: 3, E: 9
C	B: 6, D: 2, F: 2
D	B: 3, C: 2, E: 1
E	A: 2, B: 9, D: 1
F	A: 3, C: 2

Lösung zu Aufgabe 2 Durchzuführen sind hier sowohl die Breiten- als auch Tiefensuche.

- Breitensuche
  1. Wir wählen als Startknoten B, markieren ihn und fügen ihn der Queue hinzu.
     `Q = {B}`, durchlaufene Knoten: []
  2. Wir besuchen B, fügen alle Nachbarn hinzu.
     `Q = {C, D, E}`, durchlaufene Knoten: [B]
  3. Wir besuchen den Knoten C, markieren ihn und fügen alle noch unmarkierten Nachbarn der Queue hinzu.
     `Q = {D, E, F}`, durchlaufene Knoten: [B, C]
  4. Wir besuchen den Knoten D, markieren ihn und fügen alle noch unmarkierten Nachbarn hinzu.
     `Q = {E, F}`, durchlaufene Knoten: [B, C, D]
  5. Wir besuchen den Knoten E und verfahren ebenso.
     `Q = {F, A}`, durchlaufene Knoten: [B, C, D, E]
  6. Wir besuchen den Knoten F, stellen fest, dass es keine weiteren Nachbarn gibt.
     `Q = {A}`, durchlaufene Knoten: [B, C, D, E, F]
  7. Nun besuchen wir abschließend A. Auch A hat keine weiteren unmarkierten Nachbarn. Damit sind wir fertig.
     `Q = {}`, durchlaufene Knoten: [B, C, D, E, F, A]

- Tiefensuche
  1. Wir wählen wieder B als Startknoten und fügen ihn dem Stack hinzu.
     `S = {B}`, durchlaufene Knoten: []
  2. Wir nehmen B vom Stapel, markieren es und fügen die Nachbarn hinzu.
     `S = {C, D, E}`, durchlaufene Knoten: [B]
  3. Nun nehmen wir das letzte Element vom Stapel, besuchen und markieren es und fügen die Nachbarn hinzu.
     `S = {C, D, A, D}`, durchlaufene Knoten: [B, E]
  4. Wir besuchen nun erneut den letzten Knoten des Stapels und markieren ihn.
     `S = {C, D, A, C}`, durchlaufene Knoten: [B, E, D]
  5. Nächstes Element ist folglich C.
     `S = {C, D, A, F}`, durchlaufene Knoten: [B, E, D, C]
  6. Nächstes Element ist der Knoten F.
     `S = {C, D, A, A}`, durchlaufene Knoten: [B, E, D, C, F]
  7. Nächstes Element ist der Knoten A. A hat keine weiteren Nachbarn, die noch nicht markiert wurden.
     `S = {C, D, A}`, durchlaufene Knoten: [B, E, D, C, F, A]
  8. Selbiges gilt für das nächste A des Stacks.
     `S = {C, D}`, durchlaufene Knoten: [B, E, D, C, F, A]
  9. Auch Knoten D wurde bereits besucht.
     `S = {C}`, durchlaufene Knoten: [B, E, D, C, F, A]
  10. Der letzte verbleibende Knoten C wurde bereits markiert und kann damit ebenfalls entfernt werden.
      `S = {}`, durchlaufene Knoten: [B, E, D, C, F, A]

  Damit ist die gewünschte Suchreihenfolge gefunden.

Lösung zu Aufgabe 3 Nun ist der Dijkstra-Algorithmus auf diesen Graphen anzuwenden. Daraus ergibt sich nun auch der Kürzeste-Wege-Baum:

u	A	B	C	D	E	F
init	∞ | –	0 | B	∞ | –	∞ | –	∞ | –	∞ | –
B			6 | B	3 | B	9 | B	
D			5 | D		4 | D	
E	6 | E					
C						8 | C
A						13 | A
F						

Tabelle 4: Dijkstra-Algorithmus zum Graphen

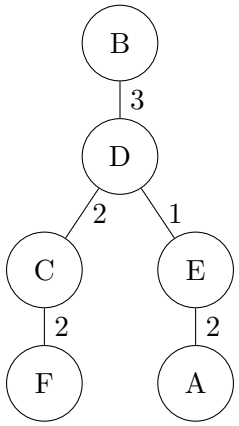


Abbildung 21: Kürzeste-Wege-Baum zum Graphen

Lösung zu Aufgabe 4 Der hauptsächliche Unterschied liegt im *LIFO*-Prinzip bei der Queue und im *FIFO*-Prinip beim Stack. Während die Queue immer das Element ausgibt, was am längsten Teil der Queue ist (also wartet), gibt der Stack das Element zurück, dass als letztes hinzugefügt wurde. Darüber wird erreicht, dass die Tiefensuche immer weiter in die Tiefe geht und nicht in großen Schritten zu weiter entfernt liegenden Knoten zurück und die Breitensuche eben auf dem Weg nichts außer Acht lässt. Algorithmen mit anderen Datenstrukturen sind denkbar, wenn auch nicht sinnvoll.

34. Hashing

Lösung zur Aufgabe Zu ermitteln sind drei Hashtabellen.

- **Lineares Sondieren**

k	h(k)
12	3
15	6
18	0
3	3, 4
5	5
99	0, 1
23	5, 6, 7
1	1, 2

Dazugehörige Hash-Tabelle:

0	1	2	3	4	5	6	7	8
18	99	1	12	3	5	15	23	

- **Quadratisches Sondieren**

k	h(k)
12	3
15	6
18	0
3	3, 4
5	5
99	0, 1
23	5, 6, 4, 1, 1, 5, 5, 3, 7
1	1, 2

Dazugehörige Hash-Tabelle:

0	1	2	3	4	5	6	7	8
18	99	1	12	3	5	15	23	

- **Doppeltes Hashing**

k	h(k)
12	3
15	6
18	0
3	3, 8
5	5
99	0, 1
23	5, 8, 2
1	1, 4

Dazugehörige Hash-Tabelle:

0	1	2	3	4	5	6	7	8
18	99	23	12	1	5	15		3

Teil IX.
Verzeichnisse

Tabellenverzeichnis

Abbildungsverzeichnis

Weiterführende Übungsaufgaben

Falls Sie weitere Aufgaben suchen, finden Sie bei [**GrUeb**] zu nahezu allen Themen dieses Moduls mehrere, gut ausgearbeitete Übungen und Lösungen. Hilfreich ist es ebenso aber auch, wenn Sie sich eigene Übungsaufgaben ausdenken und diese dann lösen. Dann stoßen Sie unweigerlich auf Fallstricke, Hürden und entwickeln eine Routine, die Sie vermutlich einigermaßen sicher durch die Klausur/Prüfung bringt.