Fachhochschule Südwestfalen
University of Applied Sciences
Campus Hagen
Fachbereich Technische Betriebswirtschaft

Algorithmik

Vorlesungsskript für das zweite Semester Wirtschaftsinformatik

Andreas de Vries

Version: 26. März 2024

Inhaltsverzeichnis

Vorwort

Die Algorithmik ist die Wissenschaft von den Algorithmen. Ein Algorithmus ist eine abstrakte, aber eindeutige Beschreibung eines Prozesses, der von einem Menschen, von einem Computer oder von einer anderen geeigneten Maschine ausgeführt werden kann. Als Rechenverfahren mit Zahlen existierten die ersten Algorithmen bereits vor über 3000 Jahren, ihren Namen erhielten sie im späten Mittelalter, und ihre noch heute gültige präzise Definition entstand in den 1930er Jahren. Verglichen mit anderen grundlegenden Begriffen der Informatik ist also das Konzept des Algorithmus uralt. Die Algorithmik behandelt Algorithmen als von der technischen Implementierung unabhängige Objekte, zentrale Gegenstände der Algorithmik sind die Korrektheit und die Effizienz von Algorithmen. Bei der Korrektheit geht es um die Frage, ob ein vorgeschlagener Algorithmus ein gegebenes Problem auch wirklich löst, bei der Effizienz um den Ressourcenverbrauch (Laufzeit und Speicherplatz) eines Algorithmus in Abhängigkeit von der Problemgöße.

Eines der Ziele der Algorithmik ist es, zu einer gegebenen Problemstellung einen möglichst guten Algorithmus zu finden. Hierbei wird die Qualität eines Algorithmus – seine Korrektheit vorausgesetzt – mit seiner Effizienz gemessen: Von zwei Algorithmen, die dasselbe Problem lösen, ist derjenige „besser“, der weniger Ressourcen benötigt. Die Algorithmik behandelt also insbesondere die zur Analyse und Bewertung von Algorithmen notwendigen mathematischen Methoden. Ein wesentliches mathematisches Werkzeug dazu ist die asymptotische Notation.

Zwei an die Algorithmik angrenzende Gebiete der Theoretischen Informatik sind die Berechenbarkeitstheorie und die Komplextitätstheorie. Beide erweitern die eigentliche Algorithmik, indem sie nicht einzelne Algorithmen untersuchen, sondern die Lösbarkeit eines gegebenen Problems überhaupt behandeln. Während die Berechenbarkeitstheorie grundsätzlich danach fragt, ob ein Problem durch Algorithmen lösbar ist, versucht die Komplexitätstheorie die Frage zu beantworten, welche Probleme effizient lösbar sind. Eine der zentralen offenen Fragen der Komplexitätstheorie ist das berühmte „P versus NP“-Problem, ob die Klasse P der effizient lösbaren Probleme sich überhaupt von der Klasse NP der „schweren“ Probleme unterscheidet.

Beide angrenzende Gebiete werden wir in diesem Skript nicht betrachten. Zum Grundverständnis von Algorithmen allerdings unverzichtbar sind Datenstrukturen. Jeder Algorithmus erwartet Daten als Eingabeparameter und muss typischerweise während seines Ablaufs Daten zwischenspeichern. Das vorliegende Skript behandelt daher zunächst Konzepte verschiedener wichtiger Datenstrukturen und stellt konkrete Implementierungen anhand des Java Collections vor. Im zweiten Teil wird auf den Algorithmenbegriff und Komplexitätsberechnungen eingegangen, während im dritten Teil Algorithmen in Graphen und Netzwerken betrachtet werden.

Literatur. Die Literatur über Algorithmen ist sehr umfangreich, als Auswahl seien hier genannt: die „Klassiker“ Cormen et al. (2001), Harel und Feldman (2006), Ottmann und Widmayer (2012), Schöning (2001) und Sedgewick und Wayne (2014), die alle weit über den Stoff dieses Skripts hinausgehen. Lesenswert sind auch Barth (2003) und Vöcking et al. (2008)

Hagen,
im März 2024 Andreas de Vries

Teil I

Datenstrukturen

1 Lineare Datenstrukturen

Kapitelübersicht

Von einem abstrakten Standpunkt aus gesehen lernt man in einer Vorlesung über die Grundlagen der Programmierung, *informationsverarbeitende Sequenzen einzelner Anweisungen in einer Programmiersprache, z.B. Java, zu erstellen.* Eine solche „informationsverarbeitende Sequenz von Anweisungen“ ist beispielsweise die Berechnung der Wurzel einer Zahl nach dem Heron’schen Verfahren oder die Aufsummierung der Umsätze aller Kunden einer bestimmten Region. Durch den Ablauf der Anweisungen wird aus gegebenen Eingabedaten Information gewonnen. Wir erzeugen mit *Daten* und *Algorithmen* also *Information*.

$$\boxed{\text{Daten}} \quad + \quad \boxed{\text{Algorithmen}} \quad \Longrightarrow \quad \boxed{\text{Information}}$$

Inhalt der Algorithmik ist die systematische und analytische Behandlung von Algorithmen und Daten. Der Schwerpunkt liegt dabei zwar auf der Untersuchung von Algorithmen, allerings kann kein Algorithmus ohne Daten entwickelt oder ausgeführt werden. Es ist daher naheliegend, eine Vorlesung über Algorithmik mit der Betrachtung von Datenstrukturen zu beginnen.

1.1 Datentypen

Daten spielen in der Informatik eine zentrale Rolle. In der Geschichte der Informatik wurden immer mehr und immer komplexere Daten verarbeitet. Die immer größeren Datenmengen konnten dabei mit immer weiter entwickelter Hardware gespeichert werden, die komplexeren

Daten konnten durch komplexere Datenkonzepte der Programmiersprachen mit dazu passenden Algorithmen verarbeitet werden.

Was aber sind Daten eigentlich genau? In der Informatik sind *Daten* eine maschinenlesbare, in Zeichenketten digital kodierte und umkehrbare Darstellung von Information [ISO, 2121272].[1] In einem Computer sind diese Zeichenketten binär kodiert, sind also *Binärwörter* mit `0` und `1` als Buchstaben.

Definition 1.1 Eine *Datenstruktur* ist nach ISO/IEC 2382 [ISO, 2122353] eine physische oder logische Beziehung zwischen Dateneinheiten und den Daten selbst. Sie ermöglicht spezifische Operationen auf ihre Dateneinheiten. □

Diese Definition ist zwar sehr präzise, aber auch sehr abstrakt, wie wissenschaftliche Definitionen ja oft. Was ist denn eine „physische oder logische Beziehung zwischen Dateneinheiten und den Daten selbst“? Eine Datenstruktur ordnet die Daten in spezifische Einheiten und verknüpft sie miteinander. Das Ziel einer Datenstruktur ist eine abstraktere Darstellung von Daten, die eine effiziente Sicht auf spezifische Aspekte der Daten ermöglicht.

Beispiel 1.2 Eine einfache Datenstruktur vieler Programmiersprachen ist der Datentyp `int`, der ein Binärwort der Länge 4 Byte = 32 Zeichen als eine ganze Zahl n mit

$$-2^{31} \leqq n \leqq 2^{31} - 1$$

darstellt. Der Datentyp ermöglicht als Operationen auf seine Dateneinheiten die fünf Grundrechenarten $+$, $-$, $\cdot$, $/$ und $\%$. □

Mit Hilfe von Datenstrukturen werden also in Binärwörtern kodierte Daten auf höhere Abstraktionsschichten gehoben. Dieses Vorgehen ist ähnlich wie beim Schreiben Buchstaben zu Wörtern werden. Führt man die Abstraktion fort, so gelangt man über die primitiven Datentypen

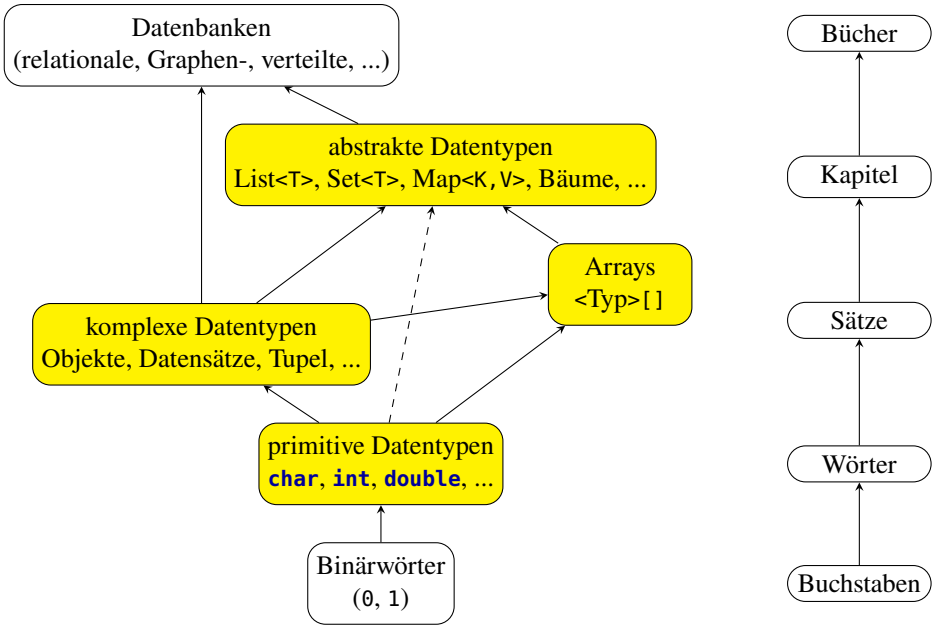


Abbildung 1.1: Das Geflecht der zunehmenden Datenkomplexität, links in der Informatik, rechts beim Vorgang des Schreibens. Datenstrukturen sind eingefärbt. Die Pfeile markieren Abstraktionen von Datenmodellen; der gestrichelte Pfeil markiert eine Abstraktion, die in manchen Programmiersprachen (z.B. Java) nicht vorgesehen ist.

[1] Der Singular von „Daten“ ist eigentlich „Datum“, heißt in der Informatik aber üblicherweise „Datenwort“ oder „Datenelement“.

zu abstrakteren Konzepten wie Objekten, Datensätzen oder Arrays, ähnlich wie im Schriftlichen Wörter zu Sätzen werden. Abstrahiert man noch weiter, so gelangt man zu dem Konzept der Datenbanken. Bei einer Datenbank steht die dauerhafte und widerspruchsfreie Speicherung großer Datenmengen im Vordergrund, die aus verschiedenen Sichten ausgewertet werden können. Beispielsweise kann in einer Unternehmensdatenbank der Gesamtumsatz eines Kunden angezeigt werden, aber auch der Umsatz aller Kunden eines bestimmten Vertriebsmitarbeiters, und so weiter. Datenbanken werden üblicherweise nicht mehr zu den Datenstrukturen gezählt.

1.1.1 Primitive Datentypen

Die Basis für diese Datenkonzepte dienen einige wenige grundlegende Datentypen, jeweils eine feste endliche Menge an Symbolen (in Java `char`), an ganzen Zahlwerten (z.B. `int`) und an Gleitkommazahlen (meist `double` nach IEEE 754); oft sind auch explizit zwei Boole'sche Werte vorgesehen (z.B. `true` und `false` vom Typ `boolean`). Dies sind die grundlegenden *primitiven Datentypen*. Die primitiven Datentypen stellen selbst bereits eine Abstraktion von Daten dar, denn so müssen wir bei der Programmierung nicht mit unüberichtlichen binären Zeichenketten umgehen, sondern können für uns Menschen komfortablere Konzepte wie Unicode-Symbole oder Zahlen verwenden. Im übertragenen Sinne kümmern wir uns also nicht mehr um die einzelnen Buchstaben eines Wortes, sondern um die Wörter selbst.

Beispiel 1.3 In Java ist die kleinste verarbeitbare Dateneinheit 1 Byte, also ein Datenwort mit 8 Binärzeichen. Wichtige primitive Datentypen in Java sind die folgenden:

Bedeutung	Datentyp	Speichergöße	Operationen
Boole'sche Werte	`boolean`	1 Byte	logische: `!`, `&&`, `||`, `^`
Buchstaben	`char`	2 Byte	—
Ganze Zahlen	`int`	4 Byte	arithmetische: `+`, `-`, `*`, `/`, `%`
Reelle Zahlen	`double`	8 Byte	arithmetische: `+`, `-`, `*`, `/`, `%`

Jeder Datentyp ermöglicht spezifische Operationen, die auf seine Datenwörter angewendet werden können, die meisten sind Verknüpfungsoperatoren („binäre Operatoren“) `<T>` × `<T>` → `<T>`, nur der logische NOT-Operator ist unär (`<T>` → `<T>`). Eine vollständige Liste der primitiven Datentypen in Java befindet sich z.B. in[2]. □

1.1.2 Objekte und Datensätze

Daten komplexer Datentypen sind typischerweise in der objektorientierten Programmierung als *Objekte* implementiert, die aus „Attributen“ oder „Datenfeldern“ bestehen. Dies können wiederum Objekte sein oder Werte grundlegender Datentypen. In allgemeineren Zusammenhängen nennt man sie auch *Datensätze* oder *Tupel*, und ihre Bestandteile „Feldern“ oder „Spalten“.

Ein Objekt hat selber wieder einen Datentyp, eine sogenannte *Klasse*. Eine Klasse ist also ein komplexer oder zusammengesetzter Datentyp. Auch in nicht-objektorientierten Programmiersprachen gibt es derartige Datenstrukturen, in C beispielsweise ist es ein *Struct*.

Beispiel 1.4 Ein Unternehmen möchte seine Artikel digital verwalten. Da ein Artikel mehrere Daten beinhaltet, z.B. eine Seriennummer, einen Eimkaufspreis und einen Verkaufspreis, kann man ihn durch ein Objekt repräsentieren. In Java sähe das dann wie folgt aus:

[2] de Vries und Weiß (2021):S. 22.

Artikel
seriennummer: int
einkaufspreis: double
verkaufspreis: double

```
class Artikel {
   int seriennummer;
   double einkaufspreis;
   double verkaufspreis;
}
```

Um auf ein Attribut eines gegebenen Objektes zuzugreifen, z.B. einen Hammer, kann man die folgende Punktnotation verwenden:

```
hammer.einkaufspreis
```

Da ein Ziel der Objektorientierung die Kapselung der Daten ist, wird der Zugriff auf die Attribute in der Regel eingeschränkt, in Java z.B. mit **`private`** oder **`protected`**; damit wird ein unbeschränkter Zugriff nur über explizit implementierte öffentliche Objektmethoden ermöglicht. □

1.2 Arrays

Eine wichtige Datenstruktur ist das *Array*. In einem Array wird eine endliche Anzahl von *Elementen* oder *Einträgen* des gleichen Datentyps gespeichert, und jedes Element erhält einen eindeutigen Index als „Adresse“. Die Elemente eines Arrays a mit n Einträgen werden üblicherweise von 0 bis $n-1$ durchnummeriert, das k-te Element wird mit $a[k]$ oder a_k bezeichnet. Wir können uns ein Array bildlich als eine Kiste a mit durchnummerierten Fächern vorstellen, die in Fach k den Eintrag $a[k]$ bzw. a_k hat. Mit

```
T[]
```

bezeichnen wir ein Array von Elementen des Typs `T`.

Beispiel 1.5 Ein Unternehmen hat 10 Artikel im Angebot und möchte speichern, wieviel von jedem Artikel auf Lager ist. Zum Beispiel könnte man die Anzahl von 10 sich auf Lager befindlichen Artikel als ein Array `int[] anzahl` von ganzen Zahlen `int` darstellen, so dass `anzahl[k]` ist, also der Eintrag in Fach Nummer `k`.

Index:	0	1	2	3	4	5	6	7	8	9
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
`anzahl` =	11	3	23	4	5	73	1	12	19	41

Dieses Array bedeutet also, dass beispielsweise der Artikel Nummer 5 noch mit der Stückzahl 73 auf Lager ist. □

Ein Array ist eine Datenstruktur mit einem sogenannten *Random Access*, oder einem *wahlfreien Zugriff*. Das bedeutet, dass auf ein beliebiges Element eines Arrays direkt, also in konstanter Laufzeit, zugegriffen werden kann. Fast alle Speichermedien haben einen solchen wahlfreien Zugriff, beispielsweise Arbeitsspeicher (RAM = Random Access Memory, aha!), Festplatten oder USB-Sticks.

1.2.1 Mehrdimensionale Arrays: Matrizen und Tensoren

Wir haben oben ein Array

```
T[] array;
```

als eine Ansammlung von Elementen eines gegebenen Datentyps `T` definiert. Was ist aber nun, wenn dieser Datentyp selber ein Array ist? Grundsätzlich steht dem nichts im Wege, man kann ein Array von Arrays desselben Typs `T` implementieren:

```
T[][] arrayInArray;
```

Ebenso ein Array von Arrays von Arrays:

```
T[][][] arrayInArrayInArray;
```

usw. Im Falle eines normalen Arrays benötigen wir genau einen Index `[i]` zur Adressierung eines Elements, für ein Array von Arrays aber schon zwei Indizes `[i][j]`. Die Anzahl benötigter Indizes eines Arrays zur Addressierung eines Elements vom Typ `T` nennt man seine *Dimension*. Ein Array `T[]` ist also eindimensional, ein Array `T[][]` zweidimensional, ein Array `T[][][]` dreidimensional, usw.

Ein zweidimensionales Array repräsentiert eine *Tabelle*, oder eine *Matrix*, wenn die Einträge Zahlen sind. Eine $(m \times n)$-*Matrix* ist in der Mathematik eine rechteckige Anordnung von Zahlen mit m Zeilen und n Spalten, also

$$A = \underbrace{\left.\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}\right\} m \textbf{ Zeilen}}_{n \textbf{ Spalten}} \tag{1.1}$$

mit den Zahlen a_{ij} für $i = 1, \ldots, m$ und $j = 1, \ldots, n$. Die Zahlen a_{ij} heißen die *Elemente* oder die *Einträge* der Matrix A. Wir schreiben oft auch $A = (a_{ij})$: In der Mathematik verwendet man zur Bezeichnung von Matrizen üblicherweise Großbuchstaben A, B, und deren Einträge werden mit den entprechenden Kleinbuchstaben bezeichnet. In der Informatik kann man diese Unterscheidung nicht machen, denn eine Matrix

```
int[][] a = {
   {0, 1},
   {1, 0}
};
```

beispielsweise hat als Array die Variable `a`, und ihre Einträge erhält man durch `a[i][j]`, also mit demselben Variablennamen.

Matrizen können, mit einigen Einschränkungen, addiert und multipliziert werden: Zwei $(m \times n)$-Matrizen $A = (a_{ij})$ und $B = (b_{ij})$ werden gemäß der Vorschrift

$$A + B = \begin{pmatrix} a_{11} + b_{11} & \cdots & a_{1n} + b_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} + b_{m1} & \cdots & a_{mn} + b_{mn} \end{pmatrix} \tag{1.2}$$

addiert, und eine $(m \times n)$-Matrix $A = (a_{ij})$ und eine $(n \times k)$-Matrix $B = (b_{ij})$ gemäß

$$AB = \begin{pmatrix} \sum_{i=1}^{n} a_{1i} b_{i1} & \cdots & \sum_{i=1}^{n} a_{1i} b_{ik} \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^{n} a_{mi} b_{i1} & \cdots & \sum_{i=1}^{n} a_{mi} b_{ik} \end{pmatrix} \tag{1.3}$$

multipliziert, was eine $(m \times k)$-Matrix AB nach dem Schema

$$(m \times n) \cdot (n \times k) \;=\; (m \times k)$$

ergibt. Ferner ist die skalare Multiplikation für einen Faktor $k \in \mathbb{R}$ durch $kA = (ka_{ij})$ definiert, d.h. duch einfaches Multiplizieren jedes Eintrags mit k.

Beispiel 1.6 Zum Beispiel ist

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} = \begin{pmatrix} 4 & 2 & 5 \\ 3 & 6 & 1 \end{pmatrix}$$

eine (2×3)-Matrix. Man bezeichnet eine Matrix mit den Einträgen kurz als $A = (a_{ij})$ mit $i = 1, 2$ und $j = 1, 2, 3$. Dann ist das Element in Zeile i und Spalte j genau a_{ij}. Zum Beispiel gilt

$$\begin{pmatrix} 4 & 2 & 5 \\ 3 & 6 & 1 \end{pmatrix} + \begin{pmatrix} 3 & 7 & -12 \\ 8 & 2 & 2 \end{pmatrix} = \begin{pmatrix} 7 & 9 & -7 \\ 11 & 8 & 3 \end{pmatrix}$$

□

Höherdimensionale Arrays, deren Einträge Zahlen sind, heißen allgemein *Tensoren*.

1.2.2 Nachteile von Arrays

Arrays sind eine extrem komfortable und vielseitig verwendbare Datenstruktur, enthalten in fast allen gängigen Programmiersprachen. Allerdings haben sie auch gewisse Nachteile, die wir an dem folgenden Beispiel herausarbeiten werden.

Beispiel 1.7 *(Ein Telefonbuch)* Da ein Telefonbuch aus gleich strukturierten Datensätzen der Form (Name, Vorname, Telefonnummer) besteht, liegt die Idee nahe, es als ein Array aus Objekten der Klasse `Eintrag` zu speichern, also:

```
public class Eintrag {
   private String name;
   private String vorname;
   private String nummer;  // String wegen etwaiger führender Nullen!
}
```

und das Telefonbuch als die Klasse

```
public class Telefonbuch {
   private Eintrag[] eintrag;

   public Telefonbuch() {
      eintrag = new Eintrag[10];
   }
   ...
   public getEintrag(int k) {
      return eintrag[k];
   }
}
```

Hier wird im Standardkonstruktor ein Telefonbuch mit der Kapazität von 10 Einträgen erzeugt. Mit der Deklaration `Telefonbuch telefonbuch` in der main-Methode einer Applikation könnte man dann durch

```
      telefonbuch.getEintrag(k);
```

auf den Eintrag Nummer k zugreifen, und mit den entsprechenden get- und set-Methoden in der Klasse `Eintrag` könnte man dann auf dessen Daten zugreifen. Wenn man nun bei jedem neuen Eintrag darauf achtet, dass er gemäß der alphabetischen Sortierung nach dem Namen in das Array eingefügt wird, so ist das Array jederzeit sortiert. Wie können wir das gewährleisten? Nehmen wir dazu an, das Telefonbuch habe eine Länge von 10 und enthalte die 6 sortierten Einträge

```
k  eintrag[k]

0  Bach, Johann Sebastian    704231
1  Beethoven, Ludwig van     499570
2  Chopin, Frederic          089702
3  Schumann, Clara           634535
4  Tchaikovsky, Peter Ilyich 471133
5  Vivaldi, Antonio          081500
```

Möchten wir nun einen neuen Eintrag einfügen, z.B. einem Herrn Mozart, so müssen wir zunächst die Position finden, an die der Eintrag muss, hier also an Index 3. Wir müssen also alle anderen Einträge danach um eine Stelle nach hinten verschieben, und zwar von hinten nach vorne bis zur Stelle 3:

$$5 \mapsto 6,$$
$$4 \mapsto 5,$$
$$3 \mapsto 4.$$

Dann ist Stelle `3` frei und wir können

```
eintrag[3] := (Mozart, Wolfgang Amadeus 175691)
```

einfügen. Damit hat das Array nun 7 Einträge mit Indizes 0 bis 6. Im Prinzip scheint also alles in Ordnung. Was aber, wenn wir weitere vier Teilnehmer einfügen möchten? □

Mit diesem Beispiel sind zwei grundsätzliche Probleme mit Arrays als Datenstruktur erkennbar: Kennen wir bei einem Array zum Zeitpunkt seiner Erzeugung nicht die genaue Anzahl der möglichen Einträge, so müssen wir eine Reserverkapazität als Puffer vorsehen. Für riesige Arrays mit Millionen oder Milliarden möglichen Einträgen kann das enormen Speicherplatz belegen, der vielleicht nie benötigt wird. Aber auch der Bedarf an Rechenzeit kann enorm sein: Will man beispielsweise in einem riesigen Array einen Eintrag auf dem Index 0 einfügen, so müssen *alle* Einträge nach hinten verschoben werden. Ein Array ist eine extrem praktische Datenstruktur, aber gibt es für manche Zwecke nicht vielleicht geeignetere Konzepte, insbesondere beim Speichern und Verwalten sehr großer Datenmengen mit sehr vielen Einfüge- und Löschoperationen?

1.3 Abstrakte Datentypen

Wir bezeichnen eine allgemeine Ansammlung von Daten desselben Datentyps als einen *Container*. Die gespeicherten Dateneinheiten eines Container heißen seine *Elemente* oder seine *Einträge*. Ein Container ist also eine „aggregierende Datenstruktur“, deren Anzahl an Dateneinheiten im Unterschied zu den primitiven oder komplexen Datentypen erst zur Laufzeit bestimmt ist. (Primitive Datentypen haben eine von der Programmiersprache festgelegte Speichergröße, komplexe Datentypen sind als Klassen zur Kompilierzeit festgelegt.)

Nach unserer Definition ist ein Array ein Container. Ein Container, der kein Array ist, wird *abstrakter Datentyp* oder *Collection* genannt.

1.3.1 Die drei Grundfunktionen eines abstrakten Datentyps

Neben der Suche eines gegebenen Eintrags sind zwei weitere grundlegende Funktionen einer abstrakten Datenstruktur das Einfügen (*insert* oder *add*) und das Löschen (*delete* oder *remove*)

einzelner Einträge. Ist die Datenstruktur sortiert, so muss eine Routine zum Einfügen sinnvollerweise die Sortierung beachten.

Wollen wir in das Telefonbuch aus Beispiel 1.7 (Fortsetzung) den Eintrag `Beethoven` einfügen, so müssen wir zunächst die Stelle finden, an die der Eintrag kommen soll (hier also $k = 1$), dann „Platz schaffen“, indem alle Einträge danach um einen Platz nach rechts verschoben werden, und schließlich in die „frei“ gewordene Stelle den neuen Eintrag speichern. Die einfache Suche nach der Position k benötigt $k + 1$ Vergleiche (denn der nächstgrößere Eintrag muss ja erst gefunden werden), das „Platz-schaffen“ benötigt $n - k$ Verschiebungen (von hinten her bis zur Stelle k), und das Speichern des neuen Eintrag ist eine Operation. Insgesamt erhalten wir also für eine Einfügeroutine eines sortierten Arrays die Anzahl

$$T_{\text{insert}}(n) = \underbrace{k+1}_{\text{Suche}} + \underbrace{n-k}_{\text{Platz schaffen}} + \underbrace{1}_{\text{speichern}} = n+2. \tag{1.4}$$

Das Einfügen in ein sortiertes Array mit n Einträgen erfordert also lineare Laufzeit. Für das Löschen eines Eintrags muss man zunächst den Eintrag finden (k Vergleiche) und dann alle nachfolgenden Einträge nach links verschieben ($n - k$ Operationen). Entsprechend benötigt das Löschen n Operationen Operationen, hat also ebenfalls lineare Laufzeit. Für sehr große Datenmengen, z.B. einem Array mit mehreren Milliarden Einträgen, bedeutet das für das Einfügen oder das Löschen sehr lange Wartezeiten. Auf einem 2 GHz Rechner erfordert allein das Löschen eines einzigen Eintrags schon einige Sekunden, wenn wir realistisch davon ausgehen, dass eine arithmetische Operation mehrere Taktzyklen benötigt.

Bei der Implementierung einer allgemeinen Datenstruktur als ein Array ergibt sich nun jedoch ein grundsätzliches technisches Problem: bei der Erzeugung eines Arrays muss bereits seine maximale Größe bekannt sein. Aber schon unser Telefonbuchbeispiel zeigt, dass die maximale Anzahl von Einträgen von vornherein oft gar nicht vorhersagbar ist. (Abgesehen davon muss bei den meisten Programmiersprachen der Index eines Array ein integer-Wert sein; in Java bedeutet das, dass die maximale Größe eines Arrays $2^{31} - 1 = 2\,147\,483\,647$ betragen kann, abhängig von der Größe des Arbeitsspeichers und des Datentyps der Einträge aber eher kleiner ist: für 1 GB Arbeitsspeicher kann ein Array von `char` „nur“ etwa 300 Mio Einträge umfassen.)

Zusammengefasst ergeben sich also die folgenden Probleme bei der Speicherung eines Verzeichnisses durch ein Array:

- Für ein Array muss die maximale Anzahl von Einträgen von vornherein bekannt sein. Legt man es „zur Vorsicht“ zu groß an, vergeudet man unnötig wertvollen Speicherplatz, legt man es zu klein an, können Einträge irgendwann nicht mehr gespeichert werden.

- Das Einfügen eines Eintrags insbesondere an den Anfang eines Arrays erfordert das Bewegen von sehr vielen Einträgen, um „Platz zu schaffen“. Für sehr große Arrays kostet das sehr viel Laufzeit.

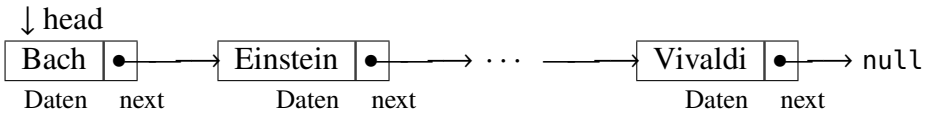
- Das Löschen von Einträgen, insbesondere am Anfang eines Arrays erfordert das Bewegen sehr vieler Einträge, um die entstandene Lücke zu schließen und kostet daher ebenfalls sehr viel Laufzeit.

Welche Alternativen zu Arrays als Datenstrukturen? Wir werden die bedeutendsten zunächst theoretisch beschreiben und erste Erfahrungen mit ihren Konzepten sammeln. Jede dieser alternativen Datenstrukturen hat jeweils ihre Vor- und Nachteile, und ihr Einsatz hängt von der konkreten Art des jeweils zu lösenden Problems ab.

1.4 Verkettete Listen *(Linked Lists)*

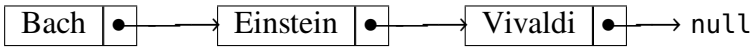
Die erste Datenstruktur, die wir neben dem Array betrachten wollen, ist die verkettete Liste. Sie ist als theoretisches Konzept radikal anders als ein Array und eine rein dynamische Datenstruktur. Sie basiert wesentlich auf „Zeigern“, im Englischen *Pointer*.

Eine *verkettete Liste (linked list)* besteht aus *Knoten (node)*, einem Datensatz, der den eigentlichen Datenteil *(data)* und einen *Zeiger (pointer)* enthält, der auf einen weiteren Knoten oder auf das Nullobjekt `null` verweist. Der erste Knoten einer verketteten Liste heißt *Kopf (head)*, der letzte Knoten verweist stets auf `null`. Der auf den nächsten Knoten verweisende Zeiger heißt „next“.



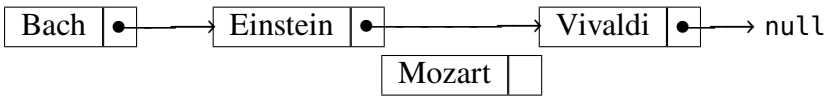
In einer verketteten Liste haben wir nur auf den Kopf der Liste direkten Zugriff, die weiteren Knoten erreichen wir nur, indem wir den Zeigern folgen. Im Gegensatz zu einem Array ist eine verkettete Liste also *kein* Verzeichnis mit direktem Zugriff.

Betrachten wir die drei Operationen Einfügen, Suchen und Löschen eines Knotens bei verketteten Listen. und deren Laufzeiten. Nehmen wir dazu beispielhaft die folgende verkettete Liste:

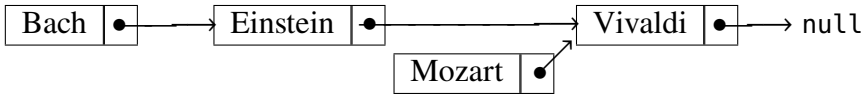


(1.5)

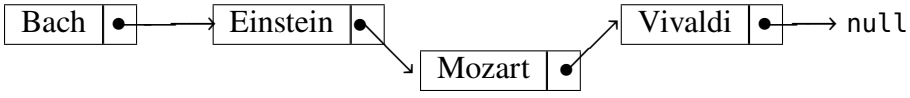
Wir möchten nun den Knoten Mozart hinter den Knoten Einstein einfügen. Wie muss man vorgehen? Die Ausgangssituation kann man wie folgt darstellen:



Folgen wir den Zeigern, beginnend beim Kopf der Liste, so suchen wir den Knoten Einstein, indem wir bei jedem angelangten Knoten den Datenteil mit dem Eintrag Einstein vergleichen. Sind die beiden Daten nicht gleich, so haben wir ihn noch nicht gefunden und folgen dem Zeiger zum nächsten Knoten in der Liste, ansonsten war die Suche erfolgreich. In unserer Beipielliste sind wir also schon beim zweiten Schritt am Ziel der Suche. Jetzt kopieren wir den Zeiger des Knotens Einstein als next-Zeiger für den einzufügenden Eintrag:

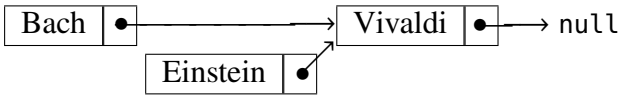


In diesem Zwischenschritt referenzieren also *zwei* Zeiger auf den Knoten Vivaldi! Abschließend wird der Zeiger von Einstein auf den neuen Knoten „umgebogen“, so dass die Liste die folgende Gestalt hat:



Wie gewünscht ist also Mozart nach Einstein in unsere Liste eingefügt.

Versuchen wir nun, den Knoten Einstein aus unserer Originalliste (1.5) zu löschen. Dazu muss die Liste also so modifiziert werden, dass der Zeiger des Knoten Bach auf Vivaldi zeigt. Um auf den Knoten Bach zuzugreifen, müssen wir wieder beim Kopf starten und die Liste durchlaufen, bis wir den Knoten Bach erreichen, indem wir für jeden Zeiger den *vohergehenden* Knoten speichern. Haben wir den zu löschenden Knoten gefunden, so nehmen wir den Zeiger des Vorgängers und lassen ihn auf Vivaldi zeigen, also:



Bemerkung 1.8 In Java wird eine verkettete Liste grundätzlich durch die Klasse `LinkedList<E>` implementiert, aber auch durch die Schnittstelle `Iterator<E>`, die u.a. die Methoden `hasNext(): boolean`, und `next(): E` vorschreibt, wobei `hasNext()` prüft, ob ein Element der Klasse `E` existiert, und `next()` das nächste Element zurück gibt und dessen Zeiger zum nächsten Element folgt. Wir werden uns mit diesen Klassen aber zunächst nicht weiter beschäftigen. □

Um die Funktionsweise einer verketteten Liste und den auf ihr aufbauenden Datenstrukturen Stack und Queue zu verstehen, werden wir sie im Folgenden selbst implementieren.

1.4.1 Die Basis: Die Klasse Node

Zunächst werden die Daten in Objekten, den Knoten oder *Nodes* einer Klasse `Node` „eingepackt“, die neben den Elementen noch einen Zeiger auf den nächsten Knoten enthält:

Node<T>
data: T
next: Node<T>

```
class Node<T> {
   protected T data;
   protected Node<T> next;

   public Node(T data, Node<T> next) {
      this.data = data;
      this.next = next;
   }
}
```

In Java kann man mit einer Art Variablen `<T>` in spitzen Klammern direkt hinter einem Klassenbezeichner einen allgemeinen Datentyp definieren, einen sogenannten *Generic Type*. Bei der Deklaration eines Objekts einer solchen Klasse muss dann dieser Typ durch eine konkrete Klasse spezifiziert werden, z.B.:

```
Node<String> einstein = new Node<>("Einstein", null);
```

Mit dieser Klasse lassen sich also Knotenobjekte verketten, beispielsweise ein Knoten mit dem Element `"Bach"` mit dem Knoten `"Einstein"` durch den Quelltextausschitt:

```
Node<String> einstein = new Node<>("Einstein", null);
Node<String> bach = new Node<>("Bach", einstein);
```

Man baut also natürlicherweise eine verkettete Liste von hinten auf, will man es anders machen, muss man Zeiger „umbiegen“.

1.4.2 Eine verkettete Liste als abstrakter Datentyp

Als abstrakten Datentyp, der die Knoten- und Zeigerwelt nach außen kapselt, kann man eine verkettete Liste nach dem Klassendiagramm in Abbildung 1.2 implementieren. Hierbei ist die Klasse `Node` als *innere Klasse* der Klasse `VerketteteListe`. Eine innere Klasse wird innerhalb einer anderen definiert und ist von außen nur eingeschränkt sichtbar; ist die innere Klasse privat und nichtstatisch, so kann sie ausschließlich von Objekten der umschließenden Klasse verwendet werden.[3] Eine Implementierung einer verketteten Liste für Elemente des Datentyps T in Java mit den Methoden `addFirst`, `contains` und `remove` lautet wie folgt:

[3]Die Darstellung von inneren Klassen ist gemäß UML 2.5 zwar nicht explizit vorgesehen, man kann sie aber als Komposition oder auch spezifischer als interne Struktur repräsentieren, siehe Figure 11.5 in dem PDF unter `http://omg.org/spec/UML/2.5/`.

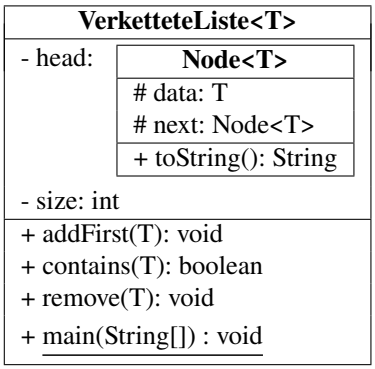


Abbildung 1.2: Klassendiagramm einer verketteten Liste mit einer inneren Klasse `Node`.

```
/** A linked list of nodes representing elements of type T.*/
public class VerketteteListe<T> {
   private Node<T> head;
   private int size;

   /** A node of a linked list with data of type T as a private inner class. */
   private class Node<T> {
      protected T data;
      protected Node<T> next;

      /** Constructor. */
      public Node(T data, Node<T> nextNode) {
         this.data = data; next = nextNode;
      }
      public String toString() {
         return data.toString();
      }
   }

   /** Inserts a node at the front of this list.*/
   public void addFirst(T data) {
      head = new Node<>(data, head);
      size++;
   }

   /** Returns true if this list contains the specified data. */
   public boolean contains(T searchData) {
      Node<T> node = head;
      while (node != null) {
         if (searchData.equals(node.data)) {
            return true;
         }
         node = node.next;
      }
      return false;
   }

   /** Deletes the first node with data and returns true if it existed.*/
```

```
public boolean remove(T data) {
   if (head == null) return false;  // list is empty

   if (data.equals(head.data)) { // delete head ...
      head = head.next;
      size--;
      return true;
   }

   Node<T> node = head.next, nodePrev = head;
   while (node != null) {
      if (data.equals(node.data)) {
         nodePrev.next = node.next;
         size--;
         return true;
      }
      nodePrev = node;
      node = node.next;
   }
   return false;
}

public String toString() {
   String out = "{";
   if (head != null) {
      out += head;
      Node<T> node = head.next;
      while (node != null) {
         out += ", " + node;
         node = node.next;
      }
   }
   out += "}";
   return out;
}

public static void main(String[] args) {
   VerketteteListe<String> satz = new VerketteteListe<>();
   satz.addFirst("unantastbar");
   satz.addFirst("ist");
   satz.addFirst("des Menschen");
   satz.addFirst("Wuerde");
   satz.addFirst("Die");
   System.out.println(satz);
   /*
   System.out.println("Enthaelt \"Die\": " + satz.contains("Die"));
   System.out.println("Enthaelt \"die\": " + satz.contains("die"));
   satz.remove("ist");
   System.out.println(satz);
   satz.remove("Die");
```

```
        System.out.println(satz);
        */
    }
}
```

Da für die innere Klasse `Node` den Datentyp `T` durch die äußere bereits bestimmt ist, kann in ihrer Definition der Zusatz `<T>` weggelassen werden.

1.5 Stacks

Ein *Stack* (auf Deutsch auch Stapel oder Keller genannt) ist eine Datenstruktur nach dem Prinzip *last in, first out (LIFO)*, d.h. es ist nur das zuletzt eingefügte Element abrufbar. Es gibt nur zwei Methoden, die die Daten eines Stacks verändern können, nämlich *push* und *pop*: Mit push wird ein Element eingefügt, mit pop wird es zurückgegeben und entfernt. Ein Stack ähnelt also einem Tablettstapel in der Mensa, bei dem man ein Tablett nur oben auflegen und nur das zuletzt aufgelegte wieder wegnehmen kann.

Stacks werden für viele Speicheraufgaben verwendet. Subroutinenaufrufe zum Beispiel werden mit ihren lokalen Daten in einem Stack gespeichert, so dass nach ihrer Ausführung automatisch der jeweils aufrufende Prozess, also eine andere Subroutine oder das Hauptprogramm, weiterlaufen kann. Insbesondere können Rekursionen mit Hilfe von Stacks speichertechnisch verwaltet werden. Die genaue Realisierung eines Stacks ist nicht festgelegt, üblicherweise wird er als verkettete Liste implementiert. Die Klasse ähnelt sehr der Klasse verketteten Liste oben,

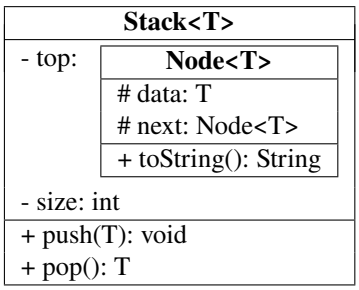


Abbildung 1.3: Klassendiagramm eines Stacks mit einer inneren Klasse `Node`.

nur heißt der zugreifbare Knoten jetzt `top` und die einzigen Zugriffsmethoden sind `push` und `pop`.

```
/** A stack elements of type T. It only contains a pointer to the top.*/
public class Stack<T> {
   private Node<T> top;
   private int size;

   /** A node as an object of an inner class. */
   private class Node<T> {
      protected T data;
      protected Node<T> next;
      public Node(T data, Node<T> nextNode) {
         this.data = data; next = nextNode;
      }
      public String toString() {
         return data.toString();
      }
```

```
    }

    /** Inserts an element into this stack.*/
    public void push(T data) {
        top = new Node<>(data, top);
        size++;
    }

    /** Returns the element having been input at last,and deletes it from this stack.*/
    public T pop() {
        if (top == null) return null; // stack is empty
        Node<T> node = top;
        top = top.next;
        size--;
        return node.data;
    }

    public String toString() {
        String out = "{";
        if (top != null) {
            out += top;
            Node<T> node = top.next;
            while (node != null) {
                out += ", " + node;
                node = node.next;
            }
        }
        out += "}";
        return out;
    }

    public static void main(String[] args) {
        Stack<Integer> q = new Stack<>();
        q.push(2); q.push(3); q.push(5); q.push(7);
        System.out.println(q);       // 7, 5, 3, 2
        System.out.println(q.pop()); // 7
        System.out.println(q);       // 5, 3, 2
        System.out.println(q.pop()); // 5
        System.out.println(q);       // 3, 2
        q.push(11); q.push(13); q.push(17);
        System.out.println(q);       // 17, 13, 11, 3, 2
        q.pop(); q.pop(); q.pop(); q.pop(); q.pop();
        System.out.println(q.pop()); // null
        System.out.println(q);       //
    }
}
```

1.6 Queues

Eine *Queue (Warteschlange)* ist eine Datenstruktur, mit der Elemente in derselben Reihenfolge gelesen und entfernt werden, in der sie eingefügt wurden. Eine Queue arbeitet also nach dem Prinzip *FIFO (first-in, first-out)*. Die Methode zum Einfügen eines Elements in eine Queue heißt üblicherweise `offer`, die Methode zum Entfernen heißt `poll`. Es sind aber auch die Bezeichnungen *enqueue* und *dequeue* geläufig. Betrachten wir dazu die folgende Implementierung in

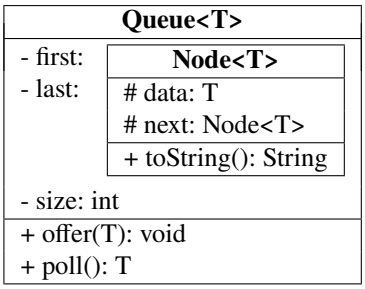


Abbildung 1.4: Klassendiagramm einer Queue mit einer inneren Klasse `Node`.

Java:

```java
/** Queue storing elements of type T. */
public class Queue<T> {
   private Node<T> first; // Referenz auf den zuerst eingefügten Knoten
   private Node<T> last;  // Referenz auf den zuletzt eingefügten Knoten
   private int size;      // Größe der Queue

   /** A node of a linked list with data of type T as inner class. */
   private class Node<T> {
      protected T data;
      protected Node<T> next;
      public Node(T data, Node<T> nextNode) {
         this.data = data; next = nextNode;
      }
      public String toString() {
         return data.toString();
      }
   }

   /** Inserts an element into this queue.*/
   public void offer(T data) {
      if (first == null) { // queue is empty
         first = last = new Node<>(data, null);
      } else {
         Node<T> oldLast = last;
         last = new Node<>(data, null);
         oldLast.next = last;
      }
      size++;
   }

   /** Returns and deletes the first element of this queue.*/
```

```
   public T poll() {
      if (first == null) return null; // list is empty
      Node<T> node = first;
      first = first.next;
      if (first == null) {
         last = first;
      }
         size--;
      return node.data;
   }

   @Override
   public String toString() {
      String out = "{";
      if (first != null) {
         out += first;
         Node<T> node = first.next;
         while (node != null) {
            out += ", " + node;
            node = node.next;
         }
      }
      out += "}";
      return out;
   }

   public static void main(String[] args) {
      Queue<String> satz = new Queue<>();
      satz.offer("Die Wuerde");
      satz.offer("des Menschen");
      satz.offer("ist unantastbar");
      System.out.println(satz.poll() + " " + satz);
      satz.poll();
      satz.poll();
      satz.offer("insanın");
      satz.offer("onuru");
      satz.offer("dokunulmazdır");
      javax.swing.JOptionPane.showMessageDialog(null, satz);
   }
}
```

1.7 Zusammenfassung

- Eine Datenstruktur ist eine Ansammlung von Objekten oder Werten als Elemente, d.h. von Einträgen eines (primitiven oder komplexen) Datentyps.
- In einer *linearen* Datenstruktur sind die Objekte in einer Reihe oder Sequenz angeordnet.
- Eine dynamische Datenstruktur ermöglicht ihr Wachsen oder Schrumpfen zur Laufzeit.

- Ein Array ist eine lineare Datenstruktur, die den direkten Zugriff auf ihre Elemente über einen Index ermöglicht. Daher ist ein Array eine indizierte Datenstruktur oder ein Random-Access-Speicher.
- Eine verkettete Liste ist eine lineare dynamische Datenstruktur, deren Elemente auch Knoten genannt werden und durch Zeiger oder Referenzen verknüpft sind. Sie hat stets einen ersten Knoten, den Head (der `null` sein kann). Einfügen und Löschen eines beliebigen Elements ist sehr effizient möglich.
- Ein Stack ist eine spezielle Version einer verketteten Liste, in der Elemente nur von oben *(top)* eingefügt oder gelöscht werden können *(LIFO = last-in, first-out)*.
- Eine Queue ist eine lineare Datenstruktur, in der ein Element nur am Ende eingefügt und nur am Anfang entfernt werden kann *(FIFO = first-in, first-out)*.

2 Sortierung von Objekten in Java

Kapitelübersicht

Sortierung von Datenstrukturen spielt in der Informatik eine wichtige Rolle. Bereits in den 1950er Jahren wurden Sortierverfahren systematisch erforscht und entwickelt. In diesem Kapitel wird die Frage behandelt, warum Sortierung so wichtig ist, der Begriff der Ordnung als das für sie notwendige Voraussetzung Sortierkriterium eingeführt und effiziente Implementierungsmöglichkeiten von Sorierkriterien in Java gezeigt.

2.1 Die binäre Suche

Was ist der Vorteil eines sortierten Verzeichnisses? Die Suche nach einem bestimmten Eintrag in einem sortierten Verzeichnis ist viel schneller („effizienter“) als in einem unsortierten. So kann man einen Namen in einem Telefonbuch sehr schnell finden, auch wenn es sehr viele Einträge hat. Versuchen Sie im Unterschied dazu jedoch einmal, (ohne Suchmaschine!) eine bestimmte Telefonnummer in dem Telefonbuch Ihrer Stadt zu finden.

Der große Vorteil von Sortierung liegt darin, dass in einem sortierten und indizierten Verzeichnis stets ein bestimmtes effizientes Suchverfahren eingesetzt werden kann, die sogenannte „binäre Suche“. Für ein von links nach rechts aufsteigend sortiertes Array `verzeichnis` mit $n =$ `verzeichnis.length` Einträgen beispielsweise lautet die binäre Suche in Java:

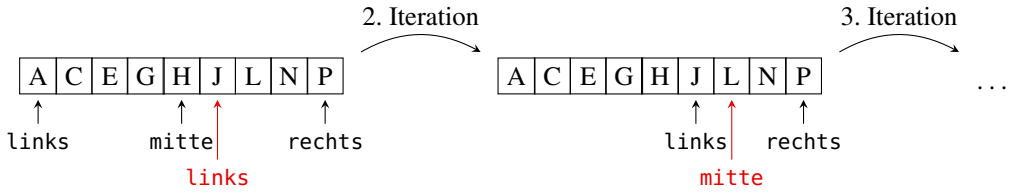
```
public static int binaereSuche(char s, char[] verzeichnis) {
   int mitte, links = 0, rechts = verzeichnis.length - 1;

   while(links <= rechts) {
      mitte = (links + rechts) / 2;
      if (s == verzeichnis[mitte]) { // Suche erfolgreich!
         return mitte;
      } else if (s > verzeichnis[mitte]) { // rechts weitersuchen ...
         links = mitte + 1;
```

```
        } else { // links weitersuchen ...
            rechts = mitte - 1;
        }
    }
    return -1;
}
```

Diese Methode gibt die Indexposition des Suchbuchstabens `s` in dem Character-Array `verzeichnis` zurück, wenn er darin enthalten ist, und den Wert -1, wenn nicht. Wir sprechen im ersten Fall, also wenn die Position des Suchbegriffs gefunden wurde, von einer *erfolgreichen Suche*, ansonsten von einer erfolglosen Suche. D.h. eine erfolglose Suche liegt vor, wenn nach der Suche sicher ist, dass der Suchbegriff sich nicht im Verzeichnis befindet.

Die binäre Suche besteht aus mehreren Iterationen (hier die `while`-Schleife), in denen jeweils der (abgerundete) Mittelwert `mitte` des linken und des rechten Endes (`links` und `rechts`) des aktuellen Arrayabschnitts untersucht wird, also beispielsweise in dem folgenden sortierten Array von Buchstaben für die Suche nach `N`:



(2.1)

Der gesuchte Begriff wird mit dem Eintrag an der Position `mitte` verglichen; es können dabei drei mögliche Fälle eintreten: Entweder ist der Begriff gefunden und die Methode gibt die Position zurück, oder wir gehen in die nächste Iteration, wobei entweder der rechte Index nach links verschoben wird (d.h. die linke Hälfte des Arrays wird weiter untersucht), oder der linke Index nach rechts (d.h. die rechte Hälfte des Arrays wird weiter untersucht). Suchen wir nach `N`, so wird der Eintrag `mitte` mit `N` verglichen und festgestellt, dass wir in der rechten Hälfte weitersuchen müssen. In dem obigen Beispiel wäre dagegen die Suche nach `J` bereits in der zweiten Iteration erfolgreich beendet und es würde der Indexwert 5 zurück gegeben.

Ein Array `a` ist ein *Verzeichnis mit direktem Zugriff (random access)*, oder kurz ein *indiziertes Verzeichnis*, da man über den Zeigerindex, beispielsweise `i`, direkt auf jeden beliebigen Eintrag `a[i]` zugreifen kann.

Theorem 2.1 ***(Suche in einem indizierten Verzeichnis)*** *In einem indizierten Verzeichnis mit n Einträgen benötigt man zur vollständigen Suche eines Suchbegriffs im ungünstigsten Fall ...*

... $2\lfloor 1 + \log_2 n \rfloor$ *Vergleiche des Suchbegriffs, wenn es sortiert ist.*

... n *Vergleiche des Suchbegriffs, wenn es nicht sortiert ist.*[1]

Eine Suche heißt dabei „vollständig“, wenn der Suchbegriff entweder in dem Verzeichnis existiert und seine Position gefunden wird, oder aber nicht vorhanden ist und dies durch die Suche sicher festgestellt wird.

Beweis. Ist das Verzeichnis sortiert, so kann man die binäre Suche verwenden. Der ungünstigste Fall tritt für diesen Algorithmus ein, wenn der gesuchte Begriff sich nicht im Verzeichnis befindet und größer als alle Verzeichniseinträge ist. Jede Iteration der while-Schleife bewirkt dann eine

[1] Zwar ist für ein Verzeichnis mit einer Hash-Tabelle, bei dem die Position eines Eintrags abhängig von seinem Wert berechnet wird, die *durchschnittliche* Laufzeit einer vollständigen Suche sogar konstant, im ungünstigsten Fall jedoch ist sie dennoch linear. Wir werden dies in Abschnitt 9.4.1 ab Seite 106 näher behandeln.

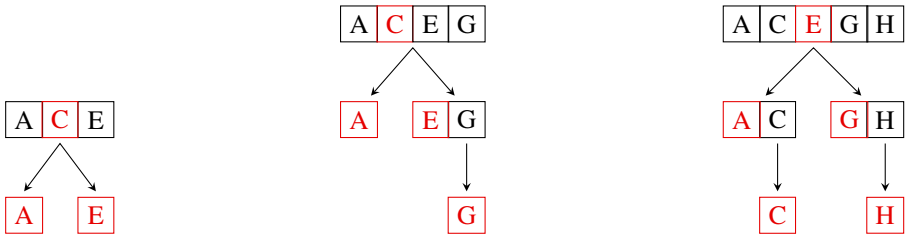


Abbildung 2.1: Array-Halbierungen bei der binären Suche für Array-Längen $n = 3$, $n = 4$ und $n = 5$.

logische Teilung des Arrays in zwei (fast) gleich große Hälften, wobei die Mitte jeweils entfällt (Abbildung 2.1). Da man ein Array der Länge n auf diese Weise maximal $\lfloor 1 + \log_2 n \rfloor$ Mal halbieren kann, bis bei n Einträgen nur noch einer übrig bleibt, ist das auch die maximale Anzahl an Iterationen. Pro Iteration kann es bis zu zwei Vergleiche geben, also folgt die erste Behauptung.

In einem unsortierten Verzeichnis ist ein ungünstiger Fall, wenn der Suchbegriff sich nicht im Verzeichnis befindet. Dann muss *jeder einzelne* Eintrag auf Gleichheit geprüft werden, d.h. man benötigt n Vergleiche. Q.E.D.

Beispiel 2.2 Sucht man in einem kleinen Array von Strings mit $n = 3$ Einträgen der Namen

```
[Bach, Mozart, Vivaldi],
```

nach dem Eintrag `Tschaikowski`, so benötigt man mit der binären Suche also $2\lfloor 1 + \log_2 3 \rfloor = 4$ Vergleiche, um herauszufinden, dass er sich nicht im Telefonbuch befindet, wie die Wertetabelle

links	rechts	mitte	links <= rechts	s == verzeichnis[mitte]	s > verzeichnis[mitte]
0	2				
2		1	ja	nein	ja
	1	2	ja	nein	nein
	-1		nein		

Tabelle 2.1: Wertetabelle für die binäre Suche nach `s = "Tschaikowski"`.

2.1 zeigt. Die benötigte Anzahl an Iterationen ist hier die theoretisch maximal notwendige, nämlich $\lfloor 1 + \log_2 3 \rfloor = 2$. □

Man spricht beim binären Suchen von einem „Algorithmus mit logarithmischer Laufzeit“, während das Suchen in einem unsortierten Verzeichnis im schlimmsten Fall „lineare Laufzeit“ benötigt. Man nennt sie daher auch oft *lineare Suche*. Der Unterschied zwischen den beiden Laufzeitklassen macht sich insbesondere für große Werte von n bemerkbar:

n	3	5	10	100	10 000	1 000 000
$2\lfloor 1 + \log_2 n \rfloor$	4	6	8	14	28	40

(2.2)

In einem unsortierten Telefonbuch mit einer Million Einträgen müssten Sie im schlimmsten Fall eine Million Namen nachschauen, für ein sortiertes dagegen wären Sie mit der binären Suche *spätestens* nach höchstens 40 Vergleichen fertig!

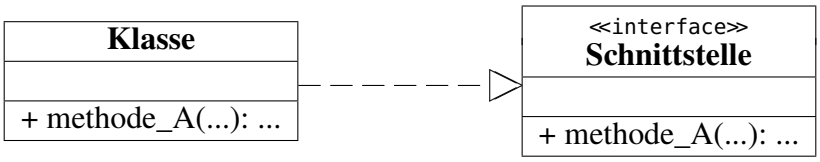
Nun ist die Sortierung von Namen, also von Strings, ja direkt einsichtig. Für uns Menschen. Bloß, woher weiß der Computer bzw. eine Programmiersprache wie Java, wie Strings sortiert werden sollen? Und allgemeiner: Wie kann man beliebige Objekte sortieren? In Java existieren dazu zwei wirkungsvolle Mechanismen, die auf dem Konzept der Interfaces beruhen. Beschäftigen wir uns daher zunächst mit Interfaces, bevor wir das Problem der Sortierung in Java angehen.

2.2 Interfaces in Java

Der wesentliche Mechanismus in Java zur Sortierung von Elementen in abstrakten Datenstrukturen basiert auf sogenannten Interfaces. Ein *Interface* ist in Java eine Art Klasse, die mit dem Schlüsselwort **`interface`** statt mit **`class`** deklariert wird und typischerweise nur aus „abstrakten“ Methoden besteht; das sind leere Methoden ohne Methodenrumpf, also reine Signaturen. Der Sinn eines Interfaces ist, durch die Methodensignaturen eine Schnittstellenbeschreibung festzulegen. Ein Interface kann nämlich von einer beliebigen Klasse mit dem Schlüsselwort **`implements`** implementiert werden, ähnlich wie mit **`extends`** von einer anderen Klasse geerbt werden kann. Bei der Implementierung eines Interfacees müssen aber alle abstrakten Methoden auch vollständig ausprogrammiert werden.

Ein Interface stellt also auf diese Art eine Schnittstellenbeschreibung oder einen „Vertrag“ dar, auf dessen Einhaltung der Compiler sich für alle Klassen verlassen kann, die es implementieren oder einmal implementieren werden. Interfaces sind wichtige Bestandteile von API’s und Programmbibliotheken, da sie die Strukturen der Methoden festschreiben, ohne deren konkrete Realisierung zu kennen oder festzulegen. In der Java API ermöglichen sie zum Beispiel eine komfortable Implementierung von Sortierungen, aber auch von abstrakten Datentypen.

Als Klassendiagramm wird die Beziehung einer Schnittstelle und einer sie implementierenden Klasse ähnlich wie eine Vererbung dargestellt:



Die einzigen Unterschiede sind, dass eine Implementierung durch eine gestrichelte Linie dargestellt und die Schnittstelle selber mit dem Wörtchen `≪interface≫` in doppelten spitzen Klammern etikettiert wird.

2.3 Natürliche Ordnung: Das Interface `Comparable`

Um Objekte sortieren zu können, muss für ihre Klasse zunächst eine *Ordnung* existieren, d.h., ein Kriterium, mit dem zwei beliebige Objekte `o1`, `o2` der Klasse verglichen und einer der drei Beziehungen `o1 < o2`, `o1 == o2` oder `o1 > o2` zugeordnet werden können. Ein mathematisches Beispiel für Objekte mit einer Ordnung sind ganze oder reelle Zahlen, aber auch Buchstaben und Wörter – also Strings – sind Beispiele. Zahlen, Buchstaben und Wörter haben eine sogenannte *natürliche Ordnung*, d.h. eine Ordnung, die jedem Objekt inhärent ist, ihm also „automatisch“ mitgegeben ist.

Erstellen wir eine eigene Klasse, so haben die aus ihr erzeugten Objekte zunächst keine natürliche Ordnung. Ein Beispiel ist die folgende einfache Klasse `Kreis`:

```
public class Kreis {
   double radius;
```

```
    public Kreis(double radius) {
       this.radius = radius;
    }
}
```

Ein Kreis ist also allein durch seinen Radius definiert. Würden wir nun mehrere Objekte mit verschiedenen Radien in eine Datenstruktur packen, hätten wir keine Chance, sie irgendwie zu sortieren. Wie kann man das erreichen?

Implementiert eine Klasse `T` das Interface `Comparable<T>`, so wird sie mit einer natürlichen Ordnung ausgestattet. Dazu muss die Methode `int compareTo(o)` deklariert werden, so dass

$$\texttt{compareTo(T o)} = \begin{cases} -1 & \text{wenn } \texttt{this < o}, \\ 0 & \text{wenn } \texttt{this.equals(o)}, \\ 1 & \text{wenn } \texttt{this > o}. \end{cases}$$

Statt 1 bzw. -1 können hier auch beliebige positive bzw. negative Integerwerte verwendet werden.

Betrachten wir als Beispiel unsere Klasse `Kreis`. Eine naheliegende Ordnung ist, einen Kreis `k1` größer als einen anderen Kreis `k2` zu nennen, wenn sein Radius größer ist, und umgekehrt. Demnach wären zwei Kreise *als Objekte* gleich, wenn sie gleichen Radius haben. In Java könnte

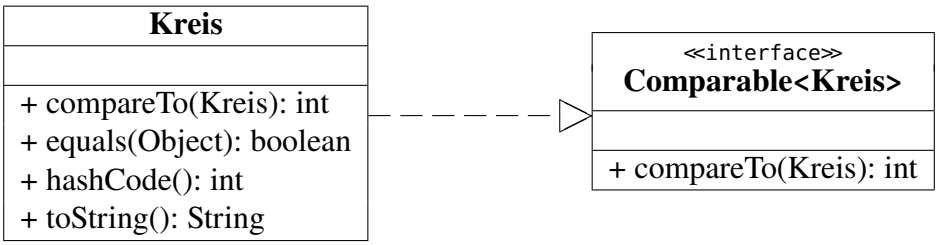


Abbildung 2.2: Die Klasse `Kreis`, die `Comparable` implementiert.

diese Ordnungsrelation durch die folgende Klassendeklaration realisiert werden, in dessen `main`-Methode beispielhaft drei Kreise erzeugt werden (Abbildung 2.2):

```
public class Kreis implements Comparable<Kreis> {
   private double radius;

   public Kreis(double radius) {
      this.radius = radius;
   }

   public String toString() {
      return "S("+radius +")";
   }

   public int compareTo(Kreis p) {
      if (this.radius < p.radius) {
         return -1;
      } else if (this.radius == p.radius) {
         return  0;
      } else {
         return  1;
      }
```

```
    }

    public static void main(String[] args) {
       Kreis[] kreise = {
          new Kreis(Math.sqrt(2)), new Kreis(1), new Kreis(10*(0.4-0.3))
       };

       System.out.println("vorher: " + java.util.Arrays.toString(kreise));
       java.util.Arrays.sort(kreise);
       System.out.println("nachher: " + java.util.Arrays.toString(kreise));
    }
}
```

Die Ausgabe des Programms lautet:

```
vorher: [S(1.4142135623730951), S(1.0), S(1.0000000000000004)]
nachher: [S(1.0), S(1.0000000000000004), S(1.4142135623730951)]
```

Bei einem Vergleich zweier double-Werte können aufgrund von Rundungsfehlern manchmal unerwartete Effekte auftreten: Hier ergibt der Vergleich von 1 und $10 \cdot (0,4 - 0,3)$ zum Beispiel $1 < 10 \cdot (0,4 - 0,3)$, was im Dezimalsystem zu der Aussage führt:

$$1 < 1.$$

Das ist aber mathematisch falsch! Was ist hier passiert? Ein Problemvon Java? Nein, es ist ein Rundungsproblem, da double-Werte nicht im Dezimalsystem gespeichert werden, sondern als Binärbrüche, und da sin sie unendlich periodisch[2]: $0,3_{10} = 0,0\overline{1001}_2$ und $0,4_{10} = 0,\overline{0110}_2$. Will man die Gleichheit von double-Werten in der `compareTo`-Methode dagegen nur bis auf eine bestimmte Ungenauigkeit bestimmen, z.B. auf die 6. Nachkommastelle, so kann man einen „Ungenauigkeitsschlauch“ um 0 vorsehen:

```
public int compareTo(Kreis p) {
   double diff = this.radius - p.radius;
   if (diff < -1e-6) return -1;
   if (diff >  1e-6) return  1;
   return 0;
}
```

Hier würde nun auch $10 \cdot (0,4 - 0,3) = 1$ angesehen. Vgl. auch Abschnitt 2.1.5 im Java-Skript.[3]

2.3.1 * `compareTo` und die Standardmethoden `equals` und `hashCode`

Obwohl nicht zwingend vorgeschrieben, sollte entsprechend der `compareTo`-Methode auch die Methode `equals` überschrieben werden, so dass sie mit der Ordnungsrelation konsistent bleibt und Gleichheit zweier Objekte dann und nur dann bestimmt, wenn sie auch als gleich sortiert würden. Das wiederum sollte parallel mit einer entsprechenden Änderung der Standardmethode `int hashCode` einhergehen, denn eines der „ungeschriebenen Gesetze“ in Java lautet:

Merkregel 1. *Wird die `equals`-Methode überschrieben, so muss die `hashCode`-Methode überschrieben werden, so dass beide konsistent bleiben. D.h., sind zwei Objekte gleich gemäß `equals`, so müssen sie denselben Hashcode haben (nicht notwendig umgekehrt).*

[2] de Vries und Weiß (2021):§A2.3.
[3] de Vries und Weiß (2021):§2.1.5.

Hintergrund ist, dass die `hashCode`-Methode den Hashcode eines Objekts berechnet, d.i. eine ganze Zahl, die eine Art Prüfziffer des Objekts darstellt und für viele effizienten Speicherungen in Java verwendet wird, insbesondere bei `HashSet` oder `HashMap`.

```
public class Kreis implements Comparable<Kreis> {
   private double radius;

   public Kreis(double radius) {
      this.radius = radius;
   }

   public String toString() {
      return "S("+radius +")";
   }

   public int hashCode() {
      return (int) (1e6 * radius);       // konsistent mit compareTo!
   }

   public boolean equals(Object o) {
      return hashCode() == o.hashCode(); // konsistent mit hashCode!
   }

   public int compareTo(Kreis p) {
      double diff = this.radius - p.radius;
      if (diff >  1e-6) return  1;
      if (diff < -1e-6) return -1;
      return 0;
   }

   public static void main(String[] args) {
      Kreis[] k = {
         new Kreis(Math.sqrt(2)), new Kreis(1.), new Kreis(10*(0.4-0.3))
      };

      java.util.Arrays.sort(k);

      String txt = "", txt2 = "\nhashCodes: ";
      for (int i = 0; i < k.length; i++) {
         for (int j = i+1; j < k.length; j++) {
            txt += "\nk" + i + "=" + k[i] + ", k" + j + "=" + k[j] +
                   ", k" + i + ".equals(k" + j + ")="    + k[i].equals(k[j]) +
                   ", k" + i + ".compareTo(k" + j + ")=" + k[i].compareTo(k[j]);
         }
         txt2 += "k" + i + "=" + k[i].hashCode() + ", ";
      }
      javax.swing.JOptionPane.showMessageDialog(null, txt + txt2, "Kreise", -1);
      System.out.println(txt + txt2);
   }
}
```

Die Ausgabe des Programms lautet:

```
k0=S(1.0), k1=S(1.0000000000000004), k0.equals(k1)=true, k0.compareTo(k1)=0
k0=S(1.0), k2=S(1.4142135623730951), k0.equals(k2)=false, k0.compareTo(k2)=-1
k1=S(1.0000000000000004), k2=S(1.4142135623730951), k1.equals(k2)=false, k1.compareTo(k2)=-1
hashCodes: k0=1000000, k1=1000000, k2=1414213,
```

In dieser Implementierung sind `equals` und `compareTo` miteinander konsistent, denn `this.equals(p)` ist `true` genau dann, wenn `this.compareTo(p)` gleich 0 ist. Entsprechend wird der dritte Kreis nicht mehr in die sortierte Menge `menge` aufgenommen, er ist ja gleich dem zweiten. Generell muss die `equals`-Methode als Eingabe ein allgemeines Objekt erwarten. Um zu überprüfen, ob die Klasse dieses Objekts überhaupt von der Klasse `Kreis` ist, verwendet man das reservierte Wort **`instanceof`**.

Die Sortierung eines Arrays wird hier durch die statische Methode der Klasse `Arrays` durchgeführt. Sie sortiert aufsteigend nach der in der `compareTo`-Methode definierten Ordnung. (Insbesondere braucht man die Methode nicht selber zu programmieren!)

Hätten wir die Koordinaten (x, y, z) des Mittelpunktes als Attribute zu unserer Klasse `Kreis` hinzugefügt und würden zwei Kreise gleich nennen, wenn ihre Radien *und* ihre Mittelpunkte gleich sind (ggf. im Rahmen einer gewissen Genauigkeit), so müssten wir die `equals`- und die `hashCode`-Methode anpassen, nicht aber die `compareTo`-Methode.

2.4 Dynamische Ordnung: Das Interface `Comparator`

Soll ich chronologisch oder alphabetisch antworten?

Filmzitat aus *Sherlock Holmes* (2010)

Das Interface `Comoparable` ist sehr praktisch, wenn man den Objekten einer Klasse ein festes und eindeutiges Sortierkriterium geben will. Manchmal möchte man jedoch Objekte nach einem anderen, oder nach Bedarf vielleicht auch nach verschiedenen Sortierbegriffen ordnen. Beispielsweise möchte ein Logistiker Containerkisten mal nach ihrem Gewicht, ein anderes Mal nach ihrem Voumen sortieren. Für solche Zwecke verwendet man in Java das Interface `Comparator`. Klassen, die einen `Comparator<T>` implementieren, müssen die Methode `compare(T p, T q)` deklarieren, die jeweils 1, 0 oder -1 zurück gibt, abhängig davon, ob `p` größer, gleich oder kleiner als `q` ist.

Betrachten wir dazu als Beispiel die Klasse Kiste, die zwei Comparatoren verwendet, `GewichtSort` und `VolumenSort`. Der erste vergleicht die Gewichte zweier Kisten miteinander, der

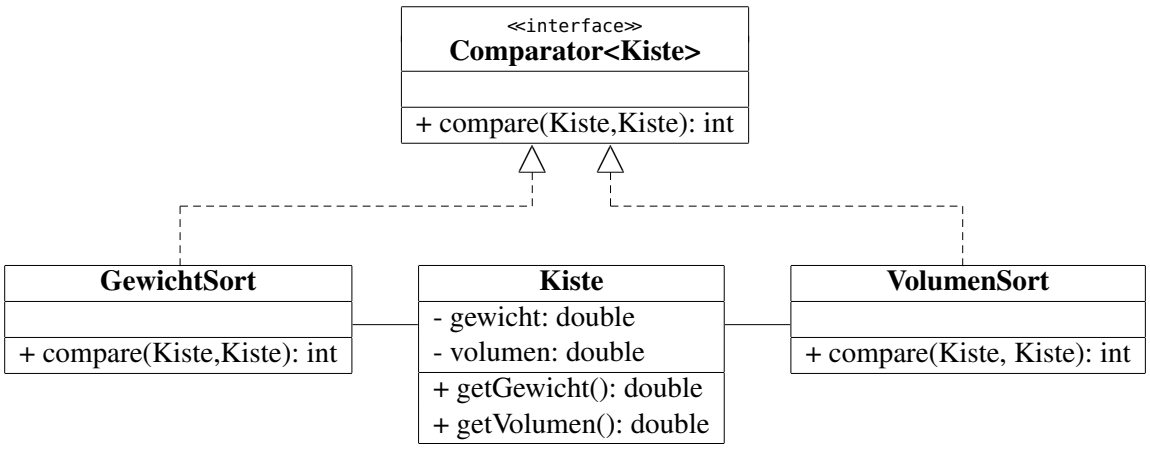


Abbildung 2.3: Die Klasse `Kiste`, die zwei die Schnittstelle `Comparator` implementierende Klassen verwendet.

zweite ihre Volumina. Beide sind so implementiert, dass sie eine Sortierung in *absteigender*

Reihenfolge ermöglichen, also die schweren bzw. die großen Kisten zuerst (Abbildung 2.3). Um mehrere Objekte der Klasse Kiste mit ihnen zu sortieren, muss zunächst ein Comparator-Objekt erzeugt werden und dieser mit der zu sortierenden Liste von Kisten von der statischen Methode `sort` der Klasse `Collections` aufgerufen werden, also z.B.

```
GewichtSort gs = new GewichtSort();
Collections.sort(liste, gs);
```

Entsprechend der `compare`-Methode des Comparators `GewichtSort` wird die Liste dadurch sortiert.

```
/** Comparator zur absteigenden Sortierung von Kisten nach ihrem Gewicht.*/
class GewichtSort implements java.util.Comparator<Kiste> {
   public int compare(Kiste p, Kiste q) {
      double diff = p.getGewicht() - q.getGewicht();
      if ( diff < 0 ) return -1;
      if ( diff > 0 ) return  1;
      return 0;
   }
}

/** Comparator zur absteigenden Sortierung von Kisten nach ihrem Volumen.*/
class VolumenSort implements java.util.Comparator<Kiste> {
   public int compare(Kiste p, Kiste q) {
      double diff = p.getVolumen() - q.getVolumen();
      if ( diff < 0 ) return  1;
      if ( diff > 0 ) return -1;
      return 0;
   }
}

/** Stellt eine Kiste mit gegebenem Gewicht und Volumen dar.*/
public class Kiste {
   private double gewicht;
   private double volumen;

   public Kiste(double gewicht, double volumen) {
      this.gewicht = gewicht;
      this.volumen = volumen;
   }

   public String toString() {
      return "(" + gewicht + " kg, " + volumen + " m^3)";
   }

   public double getGewicht() {
      return gewicht;
   }

   public double getVolumen() {
      return volumen;
   }
```

```
  public static void main(String[] args) {
     Kiste[] k = {
        new Kiste(Math.sqrt(2), 3.0), new Kiste(1., 4.0), new Kiste(1., 1.5)
     };

     GewichtSort gs = new GewichtSort();
     VolumenSort vs = new VolumenSort();

     String txt = "";
     for (int i = 0; i < k.length; i++) {
        for (int j = i+1; j < k.length; j++) {
           txt += "\nk" + i + "=" + k[i] + ", k" + j + "=" + k[j] +
                  "\n gs.compare(k"+i+", k"+j+") = " + gs.compare(k[i],k[j]) +
                   ", vs.compare(k"+i+", k"+j+") = " + vs.compare(k[i],k[j]);
        }
     }

     java.util.Arrays.sort(k, gs); // sortiere Kisten nach Gewicht
     txt += "\nliste=" + java.util.Arrays.toString(k);
     java.util.Arrays.sort(k, vs); // sortiere Kisten nach Volumen
     txt += "\nliste=" + java.util.Arrays.toString(k);
     javax.swing.JOptionPane.showMessageDialog(null, txt, "Kisten", -1);
  }
}
```

Die Ausgabe dieses Programms lautet:

```
k0=(1.4142135623730951 kg, 3.0 m^3), k1=(1.0 kg, 4.0 m^3)
 gs.compare(k0, k1) = 1, vs.compare(k0, k1) = 1
k0=(1.4142135623730951 kg, 3.0 m^3), k2=(1.0 kg, 1.5 m^3)
 gs.compare(k0, k2) = 1, vs.compare(k0, k2) = -1
k1=(1.0 kg, 4.0 m^3), k2=(1.0 kg, 1.5 m^3)
 gs.compare(k1, k2) = 0, vs.compare(k1, k2) = -1
liste=[(1.4142135623730951 kg, 3.0 m^3), (1.0 kg, 4.0 m^3), (1.0 kg, 1.5 m^3)]
liste=[(1.0 kg, 4.0 m^3), (1.0 kg, 1.5 m^3), (1.4142135623730951 kg, 3.0 m^3)]
liste=[(1.0 kg, 4.0 m^3), (1.4142135623730951 kg, 3.0 m^3), (1.0 kg, 1.5 m^3)]
```

3
Bäume und Heaps

Kapitelübersicht

Eine verkettete Liste ist eine sogenannte lineare Datenstruktur, jedes Element hat höchstens einen Nachfolger. Eine Verallgemeinerung einer Liste ist ein „Baum“, eine nichtlineare Datenstruktur, in der jedes Element mehrere Nachfolger haben kann. Bäume finden vielfältige Anwendungen, beispielsweise werden sogenannte B*-Bäume häufig zur Indizierung von Datenbanken verwendet.

3.1 Definitionen und Eigenschaften

Allgemein gesprochen zeichnet sich ein Baum dadurch aus, dass seine Elemente, die „Knoten“, durch „Kanten“ verknüpft sind. Formal definieren wir:

Definition 3.1 Ein *Baum (tree)* ist eine endliche nichtleere Menge von Elementen, *Knoten (nodes)* genannt, für die gilt:

a) es gibt genau einen speziell ausgezeichneten Knoten, die *Wurzel (root)* des Baumes;

b) jeder Knoten zeigt auf eine möglicherweise leere Folge von anderen Knoten, seine *Kindknoten (children)* oder *Nachfolger*, so dass auf jeden Knoten des Baumes außer der Wurzel genau ein Knoten zeigt, sein *Elternknoten (parent)* oder *Vorgänger*.

Ein Knoten ohne Kindknoten ist ein *Blatt (leaf)*, ein Knoten, der weder die Wurzel noch ein Blatt ist, heißt *innerer Knoten* des Baumes. Die Verbindung eines Knotens mit seinen Kindknoten heißt *Kante*. □

Aus Definition 3.1 folgt, dass jeder innere Knoten eines Baumes die Wurzel eines echten Teilbaumes *(subtree)* des Baumes ist.[1] Üblicherweise implementiert man einen Baum durch

[1] Solche Bäume werden auch „gewurzelte Bäume“ *(rooted trees)* genannt, manchmal werden allgemeinere, so genannte „freie Bäume“ betrachtet (Cormen et al. (2001):§B.5). Ferner ist nach Definition 3.1 die Reihenfolge der Kindknoten wichtig. In der mathematischen Literatur betrachtet man oft Bäume, bei denen die Reihenfolge der Kindknoten keine Rolle spielt und Bäume als eine spezielle Klasse „zyklenfreier Graphen“ aufgefasst werden.

Knoten, die als Attribute Zeiger auf weitere Knoten haben. Blätter zeigen demnach auf `null` (je nach Implementierung aber auch auf einen speziellen „Pseudoknoten“, was manche Algorithmen des Baums vereinfacht).

Ein Baum stellt also stets eine Hierarchie seiner Knotenelemente dar, wobei in jeder Hierarchieebene sich die Kinder einer gleichen Generation befinden Solche hierarchischen Strukturen gibt es sehr häufig in der realen Welt, beispielsweise

- das Organigramm eines Unternehmens,
- die Struktur eines Buches mit Kapiteln, Abschnitten und Unterabschnitten,
- die Unterteilung eines Landes in Bundesstaaten, Bezirke, Kreise und Städte;
- Stammbäume als Darstellung der Nachkommen eines Menschen
- die Gewinner der einzelnen Spiele eines Sportturniers nach dem KO-System;
- die Struktur des Dateiverzeichnisses eines Rechners in Laufwerke, Verzeichnisse, Unterverzeichnisse, Dateien;
- die Tag-Struktur eines HTML- oder XML-Dokuments.

In der Mathematik können Klammerungen ebenfalls durch eine Hierarchie dargestellt werden. So können wir beispielsweise den arithmetischen Ausdruck

$$((6 \cdot (4 \cdot 28) + (9 - ((12/4) \cdot 2)))$$

als einen Baum auffassen (Abbildung 3.1b). Übrigens zeichnen Informatiker Bäume in der Regel

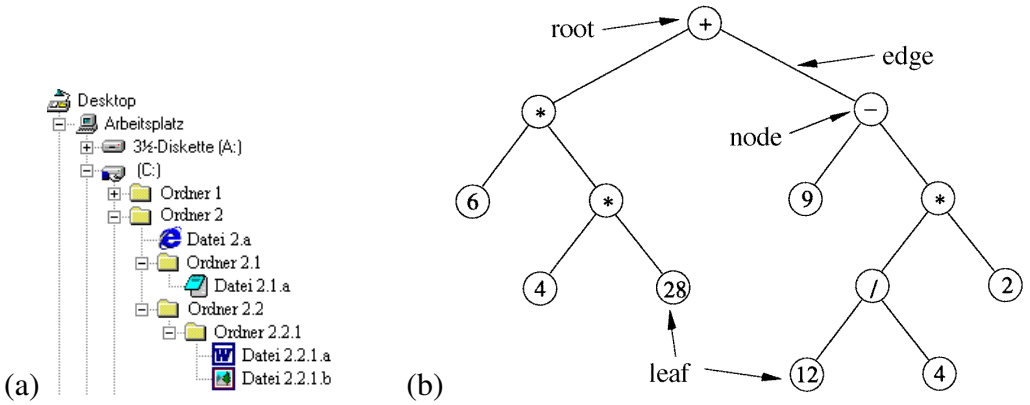


Abbildung 3.1: (a) Ein Verzeichnisbaum. (b) Ein Baum, der den arithmetischen Ausdruck $((6 \cdot (4 \cdot 28) + (9 - ((12/4) \cdot 2)))$ darstellt, mit Wurzel *(root)*, Kanten *(edges)* (inneren) Knoten *(nodes)* und Blättern *(leaves)*.

mit der Wurzel nach *oben*, anders als deren biologische Vorbilder.

Definition 3.2 Die *Höhe h* eines Baumes ist die Anzahl der Kanten eines längsten direkten Pfades von der Wurzel zu einem Blatt. □

Die Höhe eines Baumes ist also die größtmögliche Anzahl seiner Generationen oder Hierarchieebenen, wobei die Wurzel die nullte Generation bzw. Ebene ist.

Beispiele 3.3 Die Höhe des Baumes in Abbildung 3.1(a) ist $h = 6$, die Höhe des Baumes in Abbildung 3.1(b) ist $h = 4$. Man überlegt sich leicht, dass ein Baum mit n Knoten und der Höhe $h = n - 1$ eine verkettete Liste sein muss. □

Definition 3.4 Ein Baum heißt *ausgeglichen (balanced)*, wenn sich die Höhen aller Teilbäume einer Generation um höchstens 1 unterscheiden. □

Beispiele 3.5 Die beiden Bäume in Abbildung 3.1 sind nicht ausgeglichen. So hat der erste der drei Teilbäume von `Ordner 2` die Höhe 1, der zweite die Höhe 2 und der dritte die Höhe 3. Entsprechend hat der linke Teilbaum `9` nach dem Knoten - die Höhe 0, der rechte Teilbaum $*$ die Höhe 2. □

Definition 3.6 Ein *binärer Baum (binary tree)* ist ein Baum, dessen Knoten maximal zwei Kinder haben (von denen keines, eins oder beide `null` sein können). □

Der Baum in Abbildung 3.1 (b) ist ein binärer Baum.

Theorem 3.7 *Die Höhe h eines binären Baumes mit n Knoten beträgt*

$$\boxed{h = \lfloor \log_2 n \rfloor.} \tag{3.1}$$

Beweis. Zunächst beobachten wir, dass die Anzahl n an Knoten eines ausgeglichenen Baumes einer Höhe h durch die Ungleichungen

$$1 + 2 + \cdots + 2^{h-1} + 1 \ \leqq \ n \ \leqq 1 + 2 + \cdots + 2^{h-1} + 2^h \tag{3.2}$$

beschränkt ist. (Die rechte Ungleichung wird zur Gleichung, wenn der Baum „voll“ ist.) Nun sind sowohl $1 + 2 + \cdots + 2^{h-1}$ als auch $1 + 2 + \cdots + 2^h$ geometrische Reihen, d.h. es gilt[2]

$$1 + 2 + \cdots + 2^{h-1} = 2^h - 1, \qquad 1 + 2 + \cdots + 2^h = 2^{h+1} - 1.$$

Damit ergibt (3.2) dann $2^h \leqq n \leqq 2^{h+1} - 1$, und wegen $2^{h+1} - 1 < 2^{h+1}$ also

$$2^h \leqq n < 2^{h+1}.$$

Mit der rechten Seite erhalten wir $h \leqq \log_2 n$, und mit der linken $\log_2 n < h + 1$. Logarithmieren der Ungleichungen (was erlaubt ist, da der Logarithmus monoton steigend ist) ergibt daher

$$h \leqq \log_2 n < h + 1.$$

Also gilt $h = \lfloor \log_2 n \rfloor$. Q.E.D.

3.1.1 Wichtige Baumstrukturen in Java

Bäume gibt es in Java im Rahmen des Collection-Frameworks zwar (noch?) nicht, aber eine allgemeine Baumstruktur ist durch die Schnittstelle `TreeModel` gegeben, die im Wesentlichen aus dem Wurzelelement `root` besteht, das wiederum vom Typ `TreeNode` ist und maximal einen Elternknoten sowie eine Liste von Kindknoten hat. Wichtige Implementierungen dieser Schnittsellen sind `DefaultTreeModel` mit `DefaultMutableTreeNode`, vor allem von der Swing-Klasse `JTree` zur Darstellung von Verzeichnisbäumen verwendet werden.

[2]Ein Informatiker kann sich die geometrische Reihe $1 + 2 + \cdots + 2^{h-1} = 2^h - 1$ leicht klarmachen, indem er sich überlegt, dass die Zahl, die h gesetzten Bits entspricht, genau $2^h - 1$ ist, und jedes gestzte Bit eine Zweierpotenz darstellt. Beispielsweise ist für $h = 4$ die Summe $\sum_{k=0}^{h} 2^k = 1 + 2 + 4 + 8 = 1111_2 = 15 = 2^4 - 1$.

3.2 Heaps

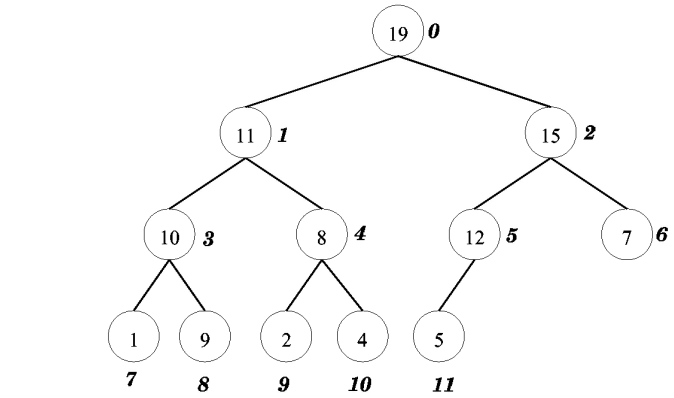
Bäume werden in der Informatik hauptsächlich verwendet, um sortierte Daten zu speichern. Je nach Hintergrund oder Sinn der Sortierung gibt es verschiedene Baumstrukturen. Fast immer werden dazu Binärbäume verwendet. Jeder Knoten muss einen eindeutigen „Schlüssel“ haben, nach dem sortiert werden kann. Ein Binärbaum heißt dann *sortiert*, wenn für jeden seiner Knoten gilt:

1. kein Knoten in seinem linken Teilbaum hat einen größeren Schlüssel,
2. kein Knoten in seinem rechten Teilbaum hat einen kleineren Schlüssel.

Oft ist man aber nur an dem einen Knoten mit dem besten (oder schlechtesten) Schlüssel interessiert. So arbeiten Computer Warteschlangen von durchzuführenden Prozessen häufig nicht nach dem FIFO-Prinzip ab, sondern verarbeiten als nächstes den Prozess mit der höchsten Priorität. Solche Warteschlangen heißen *Priority Queues*. Ein weiteres Beispiel sind Sportturniere, bei denen nur der Gewinner einer Paarung sich für die nächste Runde qualifiziert („KO-System“). Einer der einfachsten Binärbäume für solche Aufgaben ist der Heap, auf Deutsch manchmal auch *Halde* genannt.

Definition 3.8 Ein *Heap* ist ein binärer linksvollständiger Baum, bei dem *jeder* Teilbaum als Wurzel einen Knoten mit dem maximalen Schlüssel dieses Teilbaums hat. Ein solcher Heap wird auch *Maximumheap* genannt. Ein *Minimumheap* ist ein binärer linksvollständiger Baum, bei dem entsprechend jeder Teilbaum als Wurzel einen Knoten mit dem minimalen Schlüssel hat. □

Ein Heap ermöglicht ein schnelles Einfügen von Knoten und ein schnelles Suchen des Maximums (bzw. Minimums). Allerdings gibt es nicht die Möglichkeit einer schnellen Suche nach einem beliebigen Knoten, dafür müssen alle Knoten des Baums sukzessive durchlaufen werden. Ein Heap ist ein „partiell geordneter Baum“. Er kann als ein Array implementiert werden, siehe Abbildung 3.2. Die Wurzel ist dann der Array-Eintrag $a[0]$, und für einen gegebenen Index i



$a[i]$	19	11	15	10	8	12	7	1	9	2	4	5
i	0	1	2	3	4	5	6	7	8	9	10	11

Abbildung 3.2: Ein Heap als Binärbaum (mit durchnummerierten Knoten) und implementiert als ein Array $a[i]$ der Länge 12.

eines Eintrags ergeben sich die Indizes $p(i)$ seines Elternelements, seines linken Kindes $l(i)$ und seines rechten Kindes $r(i)$ durch die folgenden Formeln:

$$p(i) = \left\lfloor \frac{i-1}{2} \right\rfloor, \qquad l(i) = 2i+1, \qquad r(i) = 2(i+1). \tag{3.3}$$

Natürlich existiert ein Elternknoten $p(i)$ nur für $i > 0$. Der Index eines linken Kindes ist stets eine ungerade Zahl, während derjenige eines rechten Kindes stets gerade ist. Wann nun kann man ein gegebenes Array als einen Heap darstellen, und wann nicht? Das formale Kriterium gibt der folgende Satz an.

Theorem 3.9 *Ein Array $a[i]$ lässt sich genau dann als ein (Maximum-)Heap darstellen, wenn die so genannte „Heap-Bedingung“*

$$a[p(i)] \geqq a[i] \tag{3.4}$$

für alle i erfüllt ist, wobei der Index $p[i]$ durch Gleichung (3.3) gegeben ist. Entsprechend kann man ein gegebenes Array $a[i]$ genau dann als einen Minimumheap darstellen, wenn die Minimumheap-Bedingung

$$a[p(i)] \leqq a[i] \tag{3.5}$$

für alle i erfüllt ist.

Die Heap-Bedingungen (3.4) bzw. (3.5) lassen sich grafisch sehr leicht überprüfen, indem man den Baum von der Wurzel an generationenweise von links nach rechts mit den Array-Einträgen $a[0]$, $a[1]$, …, $a[n-1]$ auffüllt und dann für jeden Knoten einzeln prüft, ob er größer (bzw. kleiner) gleich seinen Kindknoten ist.

insert und extractMax

Um einen Heap als Datenstruktur mit einem Array als internen Speicher verwenden zu können, müssen Methoden zum Einfügen und zum Löschen von Einträgen bereitstehen, die die Heapstruktur erhalten. Wie kann das am effizientesten gelingen? Die Kernidee ist, beim Einfü-

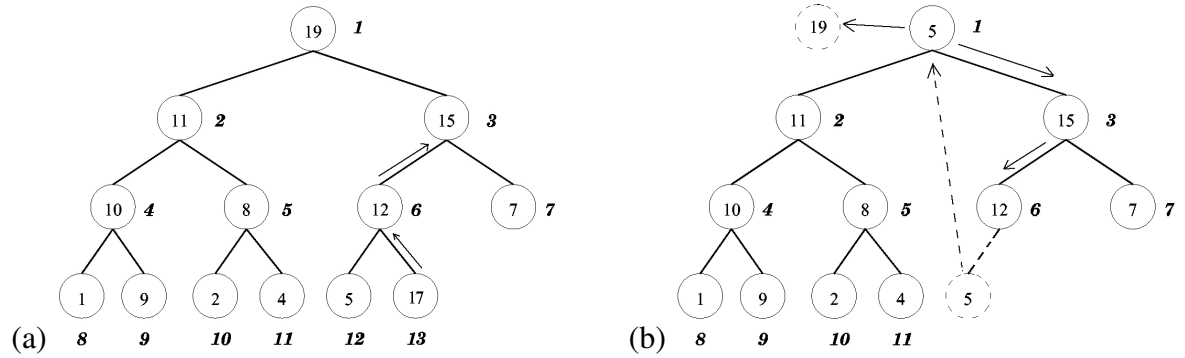


Abbildung 3.3: Die Subroutinen `insert` (a) und `extractMax` (b).

gen das neue Element zunächst *am Ende* des Arrays anzuhängen und dann die Heapeigenschaft mit dem Elternknoten zu überprüfen. Falls sie erfüllt ist, ist er schon an einer geeigneten Stelle und die Heapeigenschaft insgesamt erhalten; falls sie verletzt ist, werden die beiden Knoten ausgetauscht, und der neue Knoten ist automatisch größer als sein(e) Kindknoten. Danach wird erneut die Heapeigenschaft des neuen Knotens mit dem neuen Elternknoten geprüft und die beiden gegebenenfalls ausgetauscht, und so weiter. In Abbildung 3.3 (a) ist der Algorithmus am Beipiel des Einfügens des Knotens 17 skizziert.

Das Löschen eines *beliebigen* Elements eines Heaps ist nur sehr aufwändig zu realisieren, genau genommen ist ein Heap dazu auch gar nicht geeignet, im Gegensatz zum Beispiel zu einer verketteten Liste. Von einem Heap kann jedoch nach Konstruktion effizient der Wurzelknoten, also das Maximum, entfernt werden. Hier ist die Idee umgekehrt zu derjenigen des Einfügens: Entferne die Wurzel und speichere den letzten Knoten zunächst als Wurzel; prüfe dann sukzessive

die Heapeigenschaft mit dem größeren der beiden Kindknoten und tausche gegebenenfalls die Knoten aus. Diese Schleife wird auch *reheap* genannt. In Abbildung 3.3 (b) ist der Algorithmus `extractMax` dargestellt.

Die folgende Klasse ist eine Implementierung eines Heaps mit diesen beiden Methoden:

```
/**
 * This class represents a heap tree
 */
public class Heap<E extends Comparable<E>> {
   private E[] nodes;
   private int size;

   /** Creates a heap containing the input nodes.
    *  @param nodes an array of possibly unsorted nodes
    */
   public Heap(E[] nodes) {
      size = 0;
      this.nodes = java.util.Arrays.<E>copyOf(nodes,nodes.length);
      for (E v : nodes) {
         insert(v);
      }
   }

   /** Inserts a node into this heap.
    *  @param node the node to be inserted
    */
   public boolean insert(E node) {
      int i = size; // start i from the bottom
      E x;
      nodes[size] = node; // insert object
      size++; // extend the heap with one object
      while (i > 0 && nodes[(i-1)/2].compareTo(nodes[i]) < 0) { // heap property?
         x = nodes[i]; nodes[i] = nodes[(i-1)/2]; nodes[(i-1)/2] = x;
         i = (i - 1)/2; // go up one generation
      }
      return true;
   }

   /** Returns the maximum of this heap and deletes it.
    *  @return the maximum node of this heap
    */
   public E extractMax() {
      E root = nodes[0]; // store maximum to return in the end
      E tmp;
      int i, m, l, r;
      size--; // decrease heap size
      nodes[0] = nodes[size]; // root overwritten by last node
      i = 0; // start i from the root
      // reheap:
      while (2*i + 1 <= size) { // while there is at least a left child
         l = 2*i + 1; r = 2*(i + 1); // index of left and right child
```

```
            if (r <= size) { // does right child exist at all?
               if (nodes[l].compareTo(nodes[r]) > 0) { // which child is greater?
                  m = l;
               } else {
                  m = r;
               }
            } else {
               m = l;
            }
            if (nodes[i].compareTo(nodes[m]) < 0) { // check heap property
               tmp = nodes[i]; nodes[i] = nodes[m]; nodes[m] = tmp;
               i = m; // change nodes and index!
            } else {
               i = size + 1; // exit loop
            }
         }
         return root;
      }

      /** Returns a string representation of this heap.
       *  @return a string representation of this heap
       */
      @Override
      public String toString() {
         String output = "[";
         for (int i = 0; i < size - 1; i++) {
            output += nodes[i] + ",";
         }
         output += nodes[size - 1] + "]";
         return output;
      }

      public static void main(String[] args) {
         Heap<String> faust = new Heap<>(new String[]{
            "Da", "steh", "ich", "nun" , "ich", "armer", "Tor",
            "und", "bin", "so", "klug", "als", "wie", "zuvor"
         });
         System.out.println(faust);
         System.out.println("Remove " + faust.extractMax());
         System.out.println(faust);
         String wort = "toll";
         System.out.println("Insert " + wort);
         faust.insert(wort);
         System.out.println(faust);
      }
   }
```

Der Konstruktor erwartet zur Initialisierung ein Array, das völlig unsortiert sein kann, und erstellt daraus einen Heap. Die Ausgaben des Programms lauten:

```
[zuvor,steh,wie,nun,so,ich,und,Da,bin,ich,klug,als,armer,Tor]
Remove zuvor
```

```
[wie,steh,und,nun,so,ich,Tor,Da,bin,ich,klug,als,armer],
Insert toll
[wie,steh,und,nun,so,ich,toll,Da,bin,ich,klug,als,armer,Tor]
```

3.3 Zusammenfassung

- In diesem Kapitel behandelten wir Bäume. Ein Baum besteht aus Knoten und Kanten und ist rekursiv so definiert, dass er aus einem Wurzelknoten und mehreren Teilbäumen besteht.
- Spezielle Bäume sind die binären Bäume, in denen jeder Knoten höchstens zwei Kindknoten hat.
- Ein wichtiger binärer Baum ist der Heap, der die Maximum- oder die Minimum-Heap-Bedingung erfüllt. Entsprechend heißt der Heap Maximum- oder Minimum-Heap.

Abschließend vergleichen wir verschiedene Datenstrukturen mit demjenigen Element, das jeweils am effizientesten zu finden ist:

Datenstruktur	**Am schnellsten zu findendes Element**
Array	das Element mit gegebenem Index („random access“)
Stack	das neueste Element
Queue	das älteste Element
(Maximum-) Heap	das größte Element
Minimumheap	das kleinste Element

4

Abstrakte Datentypen in Java: Collections

Kapitelübersicht

Eine *Collection* ist in Java ein Objekt, das mehrere Elemente zu einer Einheit zusammen fasst. Nach unseren allgemeinen Betrachtungen in den vorigen Kapiteln sind es abstrakte Datentypen. Collections sind also von der Java-API bereitgestellte Datenstrukturen mit bestimmten Zugriffsmethoden, um Daten zu speichern, zu suchen und zu manipulieren. Typischerweise stellen sie Dateneinträge dar, die eine natürliche Gruppe bilden, wie beispielsweise ein Skatblatt als eine Collection von Spielkarten, ein Postfach als Collection von Briefen oder ein Telefonbuch als eine Zuordnung von Namen und Telefonnummern.

Merkregel 2. *Bei Verwendung der Collections werden die als Typparameter auftretenden Klassen stets mit einem der vier Großbuchstaben*

- `E` *für Element,*
- `T` *für Typ,*
- `V` *für Value (Wert) und* `K` *für Key (Schlüssel)*

benannt. Diese Notation wird auch in der Java-API-Dokumentation verwendet.

Merkregel 3. *Daneben wird häufig die „Wildcard“* `<?>` *verwendet, die ein Objekt der allgemeinsten Klasse* `Object` *bezeichnet. Es gibt folgende Varianten:*

- `<?>` *– steht für den allgemeinen Typ* `Object`
- `<? extends Typ>` *– steht für alle Unterklassen und die Klasse* `Typ`
- `<? super Typ>` *– steht für alle Superklassen von* `Typ` *und die Klasse* `Typ` *selbst*

Eine Hauptschwierigkeit für den Anfänger besteht zumeist darin, die scheinbar erschlagende Fül-

le an bereitgestellten Interfaces und Klassen zu strukturieren. Beginnen wir mit den Interfaces, denn im Wesentlichen wird die Struktur des Java-Collection-Frameworks durch die Interfaces

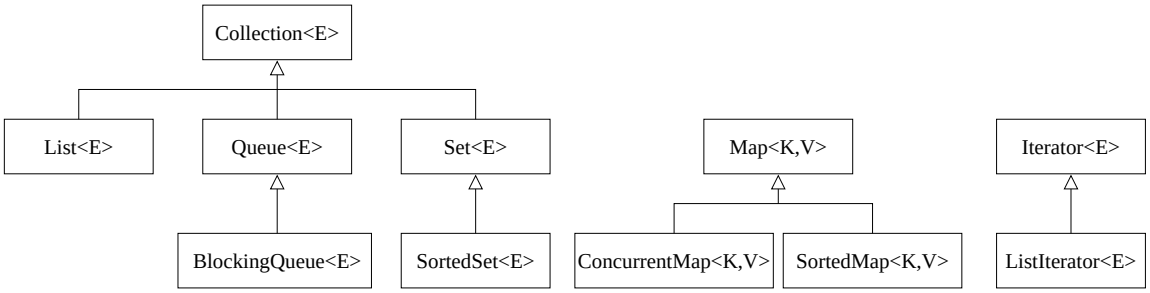


Abbildung 4.1: Wichtige Interfaces des Java Collection Frameworks. Alle befinden sich im Paket `java.util`, bis auf die sich für nebenläufige Zugriffe („Threads“) geeigneten Klassen `BlockingQueue` und `ConcurrentMap` aus dem Paket `java.util.concurrent`.

`Collection`, `Map` und `Iterator` bestimmt (Abbildung 4.1). Das Interface `Collection<E>` beschreibt die Struktur eines allgemeinen Containers von Objekten der Klasse `E` und verlangt u.a. die Implementierung der folgenden Methoden:

- `boolean add(E o)`: fügt das Element `o` ans Ende der Liste ein
- `void clear()`: löscht alle Elemente dieser Collection
- `boolean contains(E o)`: gibt `true` zurück, wenn diese Collection das Objekt `o` enthält
- `boolean isEmpty()`: gibt `true` zurück, wenn diese Collection kein Element enthält
- `boolean remove(E o)`: entfernt das eingegebene Element
- `int size()`: gibt die Anzahl der Element dieser Collection
- `E[] toArray()`: gibt ein Array zurück, das alle Elemente dieser Collection enthält.

Es hat als Subinterfaces `List<E>`, `Queue<E>` und `Set<E>`. Die grundlegenden Interfaces des Java-Collection-Frameworks sind in der folgenden Tabelle aufgelistet:

Interface	**Erläuterungen und wichtige zu implementierende Methoden**
`Iterator<E>` (verkettete Liste)	Die Elemente werden in einer verketteten Liste (Sequenz) gespeichert. Üblicherweise wird ein `Iterator` als Hilfsobjekt zum Durchlaufen der Elemente einer Collection verwendet. `boolean hasNext()`: prüft, ob ein weiteres Element existiert `E next()`: gibt das nächste Element zurück und verweist auf dessen nächstes `void remove()`: entfernt das aktuelle Element
`List-Iterator<E>` (doppelt verkettete Liste)	Die Elemente werden in einer doppelt verketteten Liste gespeichert, jedes Element hat also einen Nachfolger (`next`) und einen Vorgänger (`previous`). `ListIterator` ist ein Subinterface von `Iterator` und wird üblicherweise als Hilfsobjekt zum Durchlaufen der Elemente einer Liste (s.u.) verwendet. `boolean hasPrevious()`: prüft, ob ein Vorgängerelement existiert `E previous()`: gibt das Vorgängerelement zurück `void remove()`: entfernt das aktuelle Element
`List<E>` (indizierte Liste)	Die Elemente werden in einer indizierten verketteten Liste gespeichert und sind wahlweise direkt über einen Index oder sequentiell über einen `Iterator` zugreifbar. `boolean add(E o)`: fügt das Element `o` ans Ende der Liste ein `void add(int i, E o)`: fügt das Element `o` an die Position `i` der Liste ein `E get(int i)`: gibt das Element auf Position `i` zurück `E set(int i, E e)`: ersetzt das Element `e` an Position `i` in dieser Liste

`Queue<E>` (Warteschlange)	Die Elemente werden nach dem FIFO-Prinzip *(first-in, first out)* verarbeitet, das zuerst eingefügte Element wird auch zuerst ausgelesen. `boolean offer(E o)`: fügt das Element `o` in die Warteschlange ein, wenn möglich `E peek()`: gibt den Kopf der Warteschlange zurück `E poll()`, `E remove()`: gibt den Kopf der Warteschlange zurück und entfernt ihn
`Set<E>` (Menge)	Eine Menge enthält keine doppelten Elemente, und es können die Mengenoperationen Vereinigung, Durchschnitt und Subtraktion durchgeführt werden: `boolean add(E o)`: fügt das Element `o` in die Menge ein, wenn es nicht schon vorhanden ist `boolean addAll(Collection<E> c)`: fügt alle Elemente der eingegebenen Collection `c` in die Menge ein, wenn sie nicht schon vorhanden sind, d.h. `this` ergibt `this` $\cup$ `c`. `boolean removeAll(Collection<E> c)`: löscht alle Elemente dieser Menge, die in `c` sind, d.h. `this` ergibt `this` $\setminus$ `c`. `boolean retainAll(Collection<E> c)`: behält nur diejenigen Elemente dieser Menge, die auch in `c` sind, d.h. `this` ergibt `this` $\cap$ `c`.
`Map<K,V>` (Zuordnung)	Es werden Schlüssel-Wert-Paare `<K,V>` gespeichert, wobei jeder Schlüssel eindeutig ist. `Map` ist kein Subinterface von `Collection`. `V get(K key)`: gibt den Wert zu dem eingegebenen Schlüssel `key` zurück `V put(K key, V value)`: fügt den Schlüssel `key` mit dem Wert `value` ein `Set<K> keySet()`: gibt eine Set mit den Schlüsseln zurück

Die Klassen des Collection-Frameworks implementieren diese Interfaces, die wichtigsten sind in Abbildung 4.2 aufgeführt. Sie stellen die verschiedenen Datenstrukturen zur Speicherung von

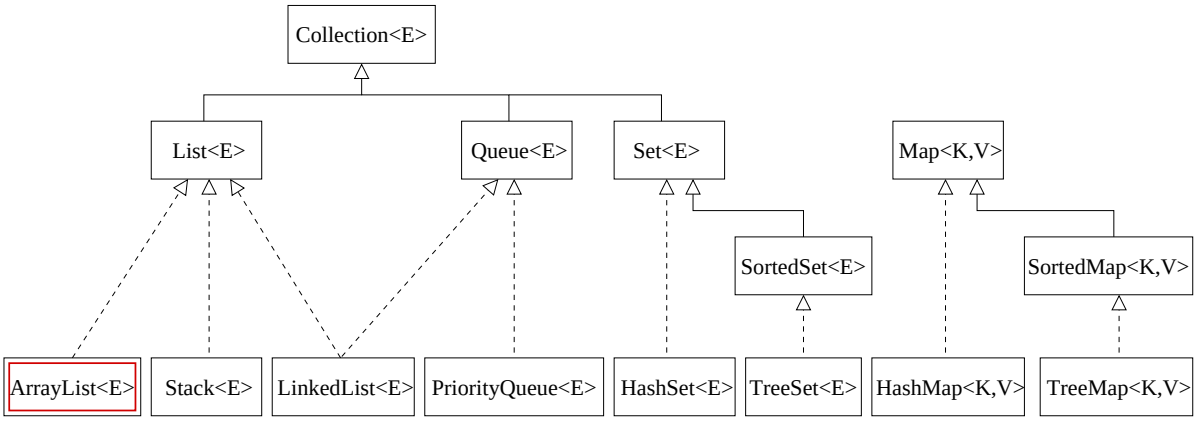


Abbildung 4.2: Wichtige Klassen des Java Collection Frameworks mit ihren Interfaces. Die für die meisten praktischen Fälle geeignete Klasse ist die ArrayList, ein „dynamisches Array“. Alle aufgeführten Klassen befinden sich im Paket `java.util`.

Objekten dar. Wir werden sie detaillierter in den folgenden Abschnitten betrachten.

4.1 Listen

Listen sind lineare Collections, oder Sequenzen. Die Elemente liegen einer bestimmten Reihenfolge vor, normalerweise in der Einfügereihenfolge. In Java haben Listen die folgenden speziellen Eigenschaften:

- Eine Liste kann doppelte Elemente enthalten.
- Ein direkter Zugriff über einen Index ist möglich, mit der Methode `get(int)`.

Die Notation einer `List` in der API-Dokumentation lautet `List<E>`, wobei `E` für den Datentyp (üblicherweise die Klasse) der Elemente steht. Die am häufigsten eingesetzten Implementierungen des Interfaces `List` sind nachfolgend aufgeführt:

Klasse	Erläuterungen und wichtige Methoden
`ArrayList<E>`	Ein dynamisches Array, d.h. eine Array mit veränderbarer Größe. `void remove(int i)`: entfernt das Element an Position `i` dieser `ArrayList` und verschiebt die Position aller nachfolgenden Elemente um 1 nach links `void trimToSize()`: passt die Kapazität dieser `ArrayList` an die aktuelle Größe an
`LinkedList<E>`	Eine Datenstruktur, die sich gut eignet, wenn man oft Elemente am Anfang einfügen oder aus der Mitte löschen will. Da `LinkedList` auch eine `Queue` implementiert, eignet sie sich auch als FIFO-Speicher. `boolean addFirst(E o)`: fügt das Element `o` an den Anfang ein `E getFirst()`: gibt das erste Element der Liste zurück. `boolean offer(E o)`: hängt das Element `o` an das Ende der Liste an. `E peek()`: gibt das erste Element der Liste zurück, aber entfernt es nicht. `E poll()`: gibt das erste Element der Liste zurück und entfernt es.
`Stack<E>`	Ein Stapel, d.h. eine LIFO-Datenstruktur. `E peek()`: gibt das oberste (zuletzt eingefügte) Element der Liste zurück, aber entfernt es nicht. `E pop()`: gibt das oberste (zuletzt eingefügte) Element der Liste zurück und entfernt es. `E push(E o)`: legt das Element `o` auf den Stapel. `int search(E o)`: gibt die Position des Elements `o` im Stapel zurück, d.h. seinen Abstand vom Kopfende *(top)* des Stapels. Das oberste Element hat die Position 1, je tiefer `o` im Stapel liegt, desto höher ist seine Position.

4.2 Sets (Mengen)

Wie in der Mathematik ist auch in Java eine Menge *(set)* eine Einheit von Elementen, die nicht doppelt vorkommen. Ein bereits vorhandenes Element wird somit mit `add` nicht erneut eingefügt, liefert allerdings `false` zurück. Die Notation einer `Set` in der API-Dokumentation lautet `Set<E>`, wobei `E` für den Datentyp (üblicherweise die Klasse) der Elemente steht. Die folgende Tabelle führt die wichtigsten Implementierungen des Interfaces `Set` auf.

Klasse	Erläuterungen und wichtige Methoden
`HashSet<E>`	Eine ungeordnete Menge, deren Speicher intern durch eine Hash-Tabelle verwaltet wird. Sie ist die schnellste Implementierung einer Menge. Da sich allerdings die (interne) Ordnung der Elemente dynamisch ändern kann, eignet sie sich nicht für geordnete Mengen. Zwei Parameter bestimmen eine `HashSet`, die Kapazität *(capacity)* c und der Ladefaktor *(load factor)* α der Hash-Tabelle, d.i. der Quotient der Anzahl der Elemente durch die Kapazität *(capacity)*. Die (erwartete) Anzahl Elemente, also `this.size()`, sollte von der Größenordnung dem Produkt αc entsprechen. Als Faustregel gilt, dass bei einem Ladefaktor $\alpha = \frac{3}{4}$ die Kapazität etwa dem Doppelten der zu erwarteten Elementezahl der Menge entsprechen sollte. Die Defaultwerte bei `HashSet` liegen bei $\alpha = \frac{3}{4}$ und $c = 16$ (d.h. `size()` ≈ 6).
`TreeSet<E>`	Eine geordnete Menge, implementiert das Interface `SortedSet`. Die interne Speicherverwaltung wird über einen Baum realisiert. Eine `TreeSet` ist eine langsamere Imlementierung einer `Set` und sollte *nicht* für ungeordnete Mengen verwendet werden. `E first()`: gibt das kleinste Element der Menge zurück `E last()`: gibt das größte Element der Menge zurück. `SortedSet<E> subSet(E from, E to)`: gibt die geordnete Teilmenge einschließlich dem Element `from` und ausschließlich dem Element `to` zurück. `SortedSet<E> headSet(E to)`: gibt die Teilmenge aller Elemente der Menge $<$ `to` zurück. `SortedSet<E> tailSet(E from)`: gibt die Teilmenge aller Elemente der Menge $\geqq$ `from` zurück.
`LinkedHashSet<E>`	Eine ungeordnete Menge, die intern die Elemente in ihrer Einfügeordnung als eine doppelt verkettete Liste verwaltet. Ist etwas langsamer als eine `HashSet`, aber schneller als eine `TreeSet`.

`EnumSet<E>`	Eine auf enum-Typen spezialisierte und sehr effizient implementierte Menge. Neben den üblichen Mengen-Methoden implementiert sie die folgenden statischen Methoden: `static EnumSet<E> noneOf(Class<E> enumType)`: gibt eine EnumSet zurück, die eine leere Menge zu dem übergebenen enumTyp darstellt. `static EnumSet<E> of(E e1, ..., E eN)`: gibt eine EnumSet zurück, die die übergebenen N Elemente enthält, wobei $N \geqq 1$. Jedes Element `e1`, ..., `eN` muss dabei vom enum-Typ `Class<E>` sein. `static EnumSet<E> range(E from, E to)`: gibt eine EnumSet zurück, die Elemente von `from` bis `to` (einschließlich) des enum-Typs `Class<E>` enthalten. Die Ordnung wird durch ihn festgelegt, und mit ihr muss stets `from` $\leqq$ `to` gelten.

Das folgende Java-Programm erzeugt 50 zufällige Lottotipps 6 aus 49, jeweils als 6-elementige sortierte Menge von Integer-Werten.

```
import java.util.*;

public class Lottotipp extends TreeSet<Integer> {

   public Lottotipp() {
      while( this.size() < 6 ) {
         this.add( (int) (49 * Math.random() + 1) );
      }
   }

   public static void main(String[] args) {
      Lottotipp tipp;
      for (int i = 1; i <= 50; i++) {
         tipp = new Lottotipp();  // erzeuge wieder neuen Tipp
         System.out.println("Tipp " + i + ": " + tipp);
      }
   }
}
```

Da Lottotipp also eine `TreeSet` aus Integern ist, können erstens keine doppelten Elemente auftreten (da er eine Menge ist) und sind zweitens die Elemente aufsteigend sortiert (da er ein Baum ist). Im Konstruktor wird bei der Erzeugung eines Objekts Lottotipp also solange eine Zufallszahl $\in \{1, 2, \ldots, 49\}$ in die Menge eingefügt, bis sie 6 verschiedene Zahlen enthält.

4.3 Maps (Zuordnungen)

Eine *Map* oder *Zuordnung* stellt eine Beziehung zwischen einem *Schlüssel (key)* und einem *Wert (value)* dar. In einer Map müssen die Schlüssel eindeutig sein, es kann also keine zwei gleichen Schlüssel geben, und jeder Schlüssel ist mit einem Wert verknüpft. Die Notation einer `Map` in der API-Dokumentation lautet

`Map<K,V>`,

wobei `K` für den Datentyp (also die Klasse) des Schlüssels steht und `V` für denjenigen des Wertes.

Ein einfaches Beispiel einer Map ist ein Telefonverzeichnis, das einem Namen (Schlüssel) eine (und nur eine) Telefonnummer (Wert) zuordnet; dabei kann es durchaus vorkommen, dass zwei Namen dieselbe Telefonnummer zugeordnet ist.

Die wichtigsten zu implementierenden Methoden des Interfaces `Map` sind die folgenden:

- `V get(K key)`: gibt den dem Schlüssel `key` zugeordneten Wert zurück.

- `V put(K key, V value)`: ordnet dem Schlüssel `key` den Wert `value` zu.
- `V remove(K key)`: entfernt den Schlüssel `key` und seinen zugeordneten Wert aus der Map.
- `V put(K key, V value)`: ordnet dem Schlüssel `key` den Wert `value` zu.
- `int size()`: gibt die Anzahl der Schlüssel-Wert-Paare der Map zurück.
- `void clear()`: löscht alle Zuordnungen dieser Map.
- `boolean containsKey(K key)`: gibt `true` zurück, wenn die Map den Schlüssel `key` enthält.
- `boolean containsValue(V value)`: gibt `true` zurück, wenn die Map einen oder mehrere Werte `value` enthält.
- `Set<K> keySet()`: gibt eine Menge zurück, die aus allen Schlüsseln der Map besteht.
- `Collection<V> values()`: gibt eine Collection aller Werte der Map zurück.

Die beiden wichtigsten Methoden einer Map sind natürlich `put` zum Aufbau der Map, und `get` zum Auslesen des passenden Schlüssels.

Klasse	**Erläuterungen und wichtige Methoden**
`HashMap<K,V>`	Eine Map mit ungeordneten Schlüsseln, deren Speicher intern durch eine Hash-Tabelle verwaltet wird. Sie ist die schnellste Implementierung einer Map. Da sich allerdings die (interne) Ordnung der Elemente dynamisch ändern kann, eignet sie sich nicht für geordnete Maps. Zwei Parameter bestimmen eine `HashMap`, die Kapazität *(capacity)* c und der Ladefaktor *(load factor)* α der Hash-Tabelle, d.i. der Quotient der Anzahl der Elemente durch die Kapazität *(capacity)*. Die (erwartete) Anzahl Elemente, also `this.size()`, sollte von der Größenordnung dem Produkt $\alpha\dot{c}$ entsprechen. Als Faustregel gilt, dass bei einem Ladefaktor $\alpha = \frac{3}{4}$ die Kapazität etwa dem Doppelten der zu erwarteten Elementezahl der Map entsprechen sollte. Die Defaultwerte bei `HashMap` liegen bei $\alpha = \frac{3}{4}$ und $c = 16$ (d.h. `size()` ≈ 6).
`TreeMap<K,V>`	Eine Map mit geordneten Schlüsseln. ie implementiert das Interface `SortedMap`. Die interne Speicherverwaltung wird über einen Baum realisiert. Eine `TreeMap` ist eine langsamere Imlementierung einer `Map` und sollte *nicht* für ungeordnete Maps verwendet werden. `K firstKey()`: gibt den kleinsten Schlüssel dieser Map zurück `K lastKey()`: gibt den größten Schlüssel der Map zurück. `SortedMap<K,V> subSet(K from, K to)`: gibt die geordnete Teil-Map einschließlich dem Schlüssel `from` und ausschließlich dem Schlüssel `to` zurück. `SortedMap<K,V> headMap(K to)`: gibt die Teil-Map aller Schlüssel der Map $<$ `to` zurück. `SortedMap<K,V> tailMap(K from)`: gibt die Teilmenge aller Schlüssel der Menge $\geqq$ `from` zurück.
`LinkedHashMap<K,V>`	Eine Map mit nicht geordneten Schlüsseln, die intern die Schlüssel in der Reihenfolge ihres Einfügens als eine doppelt verkettete Liste verwaltet. Ist etwas langsamer als eine `HashMap`, aber schneller als eine `TreeMap`.
`EnumMap<K,V>`	Eine auf Schlüssel eines enum-Typs spezialisierte und sehr effizient implementierte Map.

Ein einfaches Programm für ein Telefonbuch lautet:

```
import java.util.*;
import javax.swing.*;

public class Telefonbuch extends TreeMap<String, Integer> {
   public String ort;

   public Telefonbuch(String ort) {
```

```
        this.ort = ort;
    }

    public String toString() {
        return "Telefonbuch " + ort + ":\n" + super.toString();
    }

    public static void main( String[] args ) {
        Telefonbuch hagen = new Telefonbuch("Hagen");

        //Eintragen:
        hagen.put("Schröder", 2380);
        hagen.put("Weiß", 2371);
        hagen.put("de Vries", 2381);

        JOptionPane.showMessageDialog(
          null, hagen + "\nTel. Weiß:" + hagen.get("Weiß")
        );
    }
}
```

4.4 Wann welche Datenstruktur verwenden?

Bei der Überlegung, welche der Klassen aus Abbildung 4.2 als Datenstruktur für ein gegebenes Problem geeignet ist, sollte man sich zunächst über das zu verwendende Interface aus Abbildung 4.1 klar werden. Im Wesentlichen muss man sich also gemäß folgender Tabelle entscheiden.

Interface	Kriterien
`List<E>`	Speicherung als Sequenz, mehrfaches Auftreten gleicher Elemente möglich, indizierter Zugriff
`Queue<E>`	Speicherung als Sequenz gemäß dem FIFO-Prinzip (Warteschlange, *first-in, first-out*)
`Set<E>`	Speicherung als Menge, d.h. jedes Element kann maximal einmal auftreten
`Map<K,V>`	Speicherung von Schlüssel-Wert-Paaren, wobei ein Schlüssel in der Map eindeutig ist (d.h. alle Schlüssel ergeben eine Set!)

Nach Abbildung 4.2 ergeben sich daraus die konkreten Alternativen der für die jeweilige Problemstellung geeigneten Klasse. Grob kann man dabei von folgenden Daumenregeln ausgehen:

- In den meisten Fällen wird die `ArrayList` die geeignete Datenstruktur sein, also eine lineare Sequenz mit indiziertem Zugriff.
- Benötigt man entweder eine Queue oder eine lineare Liste und kann absehen, dass Elemente aus ihrem Innern oft gelöscht werden müssen oder vorwiegend Durchläufe durch die gesamte Liste stattfinden werden, so ist eine `LinkedList` zu bevorzugen, deren Operationen sind schneller als in einer ArrayList.
- Will man eine Set implementieren, dann wird in den meisten Fällen eine unsortierte `HashSet` die geeignete Wahl sein, sie ist in der Verarbeitung schneller als eine sortierte `TreeSet`.

- Will man eine Map implementieren, dann wird in den meisten Fällen eine unsortierte `HashMap` die geeignete Wahl sein, sie ist in der Verarbeitung schneller als eine sortierte `TreeSet`.

4.5 Statische Methoden der Klassen `Collections` und `Arrays`

Die Klasse `Collections` (man beachte das „s“ am Ende!) im Paket `java.util` enthält nur statische Methoden, durch die einige nützliche Algorithmen für die Datenstrukturen bereitgestellt werden. Sie ist damit gewissermaßen der „Werkzeugkasten“ für das Collection-Framework. Diese Methoden sind „polymorph“, d.h. sie können für verschiedene Implementierungen eines Interfaces verwendet werden. Wichtige dieser Methoden sind:

- `static <T> int binarySearch(List<T extends Comparable<T» list, T key)`, bzw. `static <T> int binarySearch(List<T> list, T key, Comparator<T> c)`: sucht nach dem Objekt `key` in der sortierten Liste `list`. die Liste muss nach dem durch `Comparable`, bzw. dem `Comparator` gegebenen Sortierkriterium sortiert sein.
- `static int frequency(Collection<?> c, Object o)` gibt Anzahl der Elemente der Collection `c` zurück, die gleich dem spezifizierten Objekt `o` sind.
- `static <T> List<T> emptyList()` gibt die leere Liste zurück.
- `static <T> Set<T> emptySet()` gibt die leere Menge zurück.
- `static <K,V> Map<K,V> emptyMap()` gibt die leere Map zurück.
- `static <T> max(Collection<? extends T> coll[, Comparator<? super T> comp])` gibt das maximale Element der Collection `coll` gemäß der natürlichen Ordnung der Elemente zurück *[*, bzw. gemäß des übergebenen Comparators `comp`*]*.
- `static <T> min(Collection<? extends T> coll[, Comparator<? super T> comp])` gibt das minimale Element der Collection `coll` gemäß der natürlichen Ordnung der Elemente zurück *[*, bzw. gemäß des übergebenen Comparators `comp`*]*.
- `static void sort(List<T> list[, Comparator<? super T> comp])` sortiert die Elemente der Liste `list` gemäß der natürlichen Ordnung der Elemente *[*, bzw. gemäß des übergebenen Comparators `comp`*]*.

Daneben gibt es Methoden zum Ersetzen (`replaceAll`) von Listenelementen, zum Umkehren der Reihenfolge (`reverse`), zum Rotieren (`rotate`, eine Art „modulo-Verschiebung“), Mischen (`shuffle`) und Vertauschen (`swap`) von Listenelementen.

In der Klasse `Arrays` des Pakets `java.util` gibt es statische Methoden `asList`, die ein Array oder eine Liste von Objekten in eine Collection-Liste umwandeln:

```
List<Integer> liste = Arrays.asList(2,3,5,7,9,11,13,17,19);
```

Übergibt man der Methode eine Referenz auf ein Array, so schießen Änderungen von Array-Einträgen auf die Einträge der Liste durch, wie das folgende Beispiel zeigt:

```
String[] quelle = {"Da", "steh", "ich", "nun"};
List<String> satz = Arrays.asList(quelle);
System.out.println(satz); // [Da, geh, ich, nun]
quelle[1] = "geh";        // Änderung der Quelle
satz.set(0, "Dann");      // Änderung der Liste
System.out.println(satz); // [Dann, geh, ich, nun]
System.out.println(Arrays.toString(quelle)); // [Dann, geh, ich, nun]
```

Auf diese Weise kann also die asList-Methode sowohl zur effizienten Erzeugung einer gegebenen Liste von Objekten (auch Zahlen) als auch als „Brücke“ einer Array-Darstellung und einer Collection-Darstellung von Daten verwendet werden.

4.6 Zusammenfassender Überblick

Wir haben in diesem Kapitel das Collection-Framework in Java kennengelernt. Es implementiert die linearen abstrakten Datentypen der Theoretischen Informatik, zum Beispiel eine dynamische Arrayliste, eine verkettete Liste und eine Set. Strenggenommen zwar nicht im Collection-Framework, aber dennoch eine wichtige lineare abstrakkte Datenstruktur in Java ist die Map, eine Verallgemeinerung eines dynamischen Arrays, das anstatt numerischer Indizes allgemeine sortierbare Objekte (z.B. String) als Schlüssel für ihre Einträge verwendet.

Hierbei ist eine Liste in Java stets eine „indizierte“ und doppelt verkettete Liste, d.h. man kann sie sowohl über einen Iterator als klassische verkettete Liste darstellen als auch wie bei einem Array über den Index (mit `get(i)`) auf sie zugreifen, allerdings unter Laufzeitverlust.

Es gibt jeweils zwei Arten von Implementierungen von Mengen (Sets) und Maps in Java, einerseits mit unsortierter Speicherung mit Hash-Tabelle (`HashSet`, `HashMap`) oder mit sortierter Speicherung mit Hilfe eines Baumes (`TreeSet`, `TreeMap`).

Abschließend vergleichen wir verschiedene Datenstrukturen mit demjenigen Element, das jeweils am effizientesten zu finden ist:

Datenstruktur	**Am schnellsten zu findendes Element**
Array(List)	das Element mit gegebenem Index („random access“)
HashMap, TreeMap	das Element mit gegebenem Schlüssel („random access“)
TreeMap	das Element mit dem kleinsten Schlüssel
LinkedList	das erste oder das letzte Element
Stack	das neueste Element
Queue	das älteste Element
Set	—
TreeSet	das nach der Ordnung kleinste Element
(Maximum-) Heap	das größte Element
Minimumheap	das kleinste Element

Teil II

Algorithmen

Es gibt zwei Wege, einen Softwareentwurf zu gestalten. Einer besteht darin, ihn so einfach zu machen, dass es offensichtlich keine Mängel gibt, und der andere darin, ihn so kompliziert zu machen, dass es keine offensichtlichen Mängel gibt.

C. A. R. Hoare (https://doi.org/10.1145/358549.358561)

5 Die Elemente eines Algorithmus

Kapitelübersicht

Die meisten Menschen kennen den Begriff „Algorithmus“ oder haben ihn zumindest schonmal gehört. Im Alltag hat er meist einen negativen oder unangenehmen Beigeschmack, Algorithmen analysieren unsere privaten Daten und unser Verhalten im Internet, manchmal überwachen sie uns sogar oder bewirken, dass Menschen ihr Beruf gekündigt wird[1]. Oft sind sie nach Mathematikern benannt (Euklid, Gauß, Heron, Horner, Strassen, ...), und die zählen nun auch nicht gerade zu den Höchstplatzierten auf der Beliebtheitsskala bekannter Persönlichkeiten.

Was ist ein Algorithmus? Bevor wir im Folgenden eine genauere und für dieses Skript geeignete Definition geben, sei für das erste Verständnis zunächst als Charakterisierung genannt: Ein Algorithmus ist eine Handlungsanweisung, das unzweideutig den Weg zur Lösung eines gegebenen Problems oder einer gegebenen (und durch Parameter bestimmten) Klasse von Problemen beschreibt. Beispiele für Algorithmen sind im Alltagsleben Kochrezepte oder Bauanleitungen, in der Betriebswirtschaft Prozessbeschreibungen und Bilanzierungsregeln, in der Mathematik Rechenverfahren. In der Informatik ist ein Algorithmus als Handlungsanweisung genau genug beschrieben, um daraus eine Subroutine zu programmieren.

Wie aber kann man nun einen Algorithmus „genau genug beschreiben“?

5.1 Beschreibungsformen für Algorithmen

Im wesentlichen gibt es zwei Formen, einen Algorithmus zu beschreiben, den Pseudocode und den Programmablaufplan.

5.1.1 Pseudocode

Um die Funktionsweise eines Algorithmus darzustellen, verwendet man meist einen *Pseudocode*, d.h. ein einer höheren Programmiersprache ähnlicher Quelltext, der auf Verständlichkeit

[1] O'Neil (2016):S. 10f.

für Menschen Wert legt, mathematische Notation umfasst und auf technische Spezifika einer konkreten Programmiersprache wie Datentypen oder Deklarationen verzichtet. Es gibt keinen genormten Standard für Pseudocode, sondern an existierende Programmiersprachen angelehnte Stile. Gebräuchlich sind der Pascal-Stil (Anweisungsblöcke werden mit `BEGIN` ... `END` markiert) und der C-Stil (Blöcke werden mit `{ ... }` markiert). Letzteren werden wir in diesem Skript verwenden.

Als Beispiel betrachten wir in Tabelle 5.1 den Algorithmus der binären Suche auf Seite 23, der links als Quelletxt in Java aufgelistet ist und rechts als Pseudocode. Wie wir sehen,

Quelltext in Java	**Pseudocode**

`public static int binäreSuche(char s, char[] v) {`
`  int m, l = 0, r = v.length - 1;`
`  while (l <= r) {`
`  m = (l + r) / 2;`
`  if (s > v[m]) {`
`  l = m + 1;`
`  } else if (s < v[m]) {`
`  r = m - 1;`
`  } else {`
`  return m;`
`  }`
`  }`
`  return -1;`
`}`
	algorithm binäreSuche(s, v[]) {
	$l \leftarrow 0,\ r \leftarrow \lvert v\rvert$;
	while $(l \leqq r)$ {
	$m \leftarrow \left\lfloor \frac{l+r}{2} \right\rfloor$;
	if $(s > v[m])$ {
	$l \leftarrow m+1$;
	} **else if** $(s < v[m])$ {
	$r \leftarrow m-1$;
	} **else** {
	return m;
	}
	return -1;
	}
	}

Tabelle 5.1: Vergleich der binären Suche als Java-Quelltext und als Pseudocode.

werden technische Spezifizierungen wie Datentypen, Deklarationen oder Zugriffsmodifikatoren weggelassen, dafür werden Wertzuweisungen mit „←“ statt eines Gleichheitszeichens „=“ markiert. Auch das implizite Casting bei der Integer-Division wird durch die unteren Gauß'schen Klammern $\lfloor \cdots \rfloor$ ersetzt:

$$\texttt{(l+r)/2} \qquad \Longleftrightarrow \qquad \left\lfloor \frac{l+r}{2} \right\rfloor.$$

Die unteren Gauß'schen Klammern, oder auch „Floor-Klammern“ genannt, runden eine reelle Zahl auf den nächstkleineren ganzzahligen Wert ab, siehe Gleichung (A.1) auf Seite 133. Gängige Symbole und Notationen für elementare Operationen in Pseudocode sind in Tabelle

Operation	**Pseudocode**	**Java**
Wertzuweisung	$\leftarrow$, :=	=
Wertetausch, Dreieckstausch	$\leftrightarrow$	bei **int**: `x ^= y; y ^= x; x ^= y;` bei **double**: `tmp = x; x = y; y = tmp;`
Vergleich	$=, ==, \neq, <, >, \leqq, \geqq$	==, !=, <, >, <=, >=
Arithmetik	$+, -, \cdot, /$, mod	+, -, *, /, %
Logik	$\neg$, not, and, $\wedge$, or, $\vee$, xor, $\oplus$	!, &&, ||, ^
Verkettung von x und y	$xy,\ x \circ y$	x+y
Ein- und Ausgabe	input, output, return	**return**

Tabelle 5.2: Gängige Symbole und Notationen in Pseudocode für elementare Operationen.

5.2 aufgeführt. Häufig wird zu Beginn des Pseudocodes eines Algorithmus das Wort `algorithm` verwendet sowie die Begriffe `input` und `output`, die die Eingabeparameter bzw. das zu liefernde Ergebnis erläutern.

Algorithmus 5.1: Pseudocode der binären Suche

algorithm binäreSuche(s, v[]) {

```
   input: ein Wort s und ein Array von Worten v
   output: Index eines Wortes in v, das s gleicht, sonst −1

   l ← 0,  r ← |v| − 1;
   while (l ≦ r) {
      m ← ⌊(l + r)/2⌋;
      if (s > v[m]) {
         l ← m + 1;
      } else if (s < v[m]) {
         r ← m − 1;
      } else {
         return m;
      }
   }
   return −1;
}
```

Zweck von Pseudocode ist es, den eigentlichen Ablauf eines Algorithmus in den Vordergrund zu stellen und die für eine Programmiersprache (meist) wesentlichen technischen Aspekte auszublenden. Dadurch soll der Algorithmus grundsätzlich leichter lesbar und somit verständlich für uns Menschen, aber auch für einen Programmierer präzise genug sein, dass er ihn in einer beliebigen Programmiersprache implementieren kann.

Arrays in Pseudocode. Da Datenstrukturen für viele Algorithmen eine wesentliche Rolle spielen, muss man sie auch in Pseudocode darstellen können. Eine besondere Stellung nimmt dabei das Array ein. Wird es als Eingabeparameter eines Algorithmus verwendet, so werden häufig eckige leere Klammern hinter die Variable geschrieben, also z.B. $v[]$ für das Array in der obigen binären Suche. Wir werden in diesem Skript auch oft die in der Webprogrammierung gebräuchliche JSON-Notation verwenden, die beispielsweise in den Programmiersprachen JavaScript und PHP enthalten ist und für die in fast allen Programmiersprachen Parser existieren, siehe `http://www.json.org/`. In JSON wird ein Array durch eine von eckigen Klammern umschlossene Liste von Werten beschreibt, also z.B.

```
a = [2, 3, 5, 7, 11, 13];
```

für ein Array aus Integer-Werten. Ebenso werden wir hier die Konvention der „C-artigen“ Programmiersprachen übernehmen, dass ein Array mit dem Index 0 beginnt. Andere abstrakte Datenstrukturen werden in der Regel in den Kommentaren des Algorithmus näher beschrieben.

Objektorientierung in Pseudocode. Die Objektorientierung ist eine bestimmte Sicht auf die Organisation von Daten und Algorithmen. Im Wesentlichen werden die zu speichernden Daten in gleich strukturierte Einheiten eingeteilt, die so genannten *Objekte*. Die zu speichernden Daten heißen die *Attribute* des Objekts. Algorithmen oder Funktionen, die auf die Daten des Objektes zugreifen, heißen *Methoden* oder *Objektmethoden*. Insgesamt ist ein Objekt also eine strukturierte Gruppierung von Attributen und Methoden („Daten und Algorithmen“), die auf sie zugreifen. Diese Gruppierung wird als *Klasse* bezeichnet und können durch *Klassendiagramme* dargestellt werden:

Klasse
attribute ...
methoden(...) ...

Eine Klasse ist mit anderen Worten also ein abstrakter Datentyp, der zusammengehörige Daten beinhaltet sowie Algorithmen, die diese Daten verarbeiten. Ein Objekt wird auch *Instanz* seiner Klasse genannt. Auf Attribute wird in der Regel nicht direkt zugegriffen, nur auf die Methoden. Dies geschieht in Pseudocode mit der gebräuchlichen Punktnotation: Ist `obj` ein Objekt und `f(x)` eine Methode mit dem Argument `x`, so wird sie mit

```
obj.f(x)
```

aufgerufen. Innerhalb einer Methode wird das Objekt `this` genannt.[2]

5.1.2 Programmablaufpläne

Ein *Programmablaufplan (PAP)*, oder auch *Flussdiagramm* (im Englischen *Flowchart*), ist eine grafische Darstellung eines Algorithmus. Die verwendbaren Symbole sind nach ISO 5807 genormt und repräsentieren In- und Outputs, Zuweisungen, bedingte Verzweigungen und Subroutinen. Gängige Symbole eines Programmablaufplans sind in Tabelle 5.3 aufgelistet. Weitere

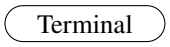
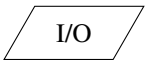
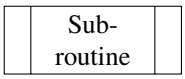
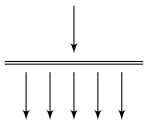
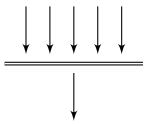
Symbol	Bezeichnung	Symbol	Bezeichnung
	Ablaufpfeil (Flow Line)		Verzweigung, Entscheidung
	Kontrollpunkt		
	Ein- und Ausgabe		
	Operation		Unterprogramm
	Parallelisierung		Synchronisation

Tabelle 5.3: Gängige Symbole eines Programmablaufplans.

Details siehe unter [PAP].

Programmablaufpläne sind intuitiv und leicht zu verstehen, man verfolgt einfach den Ablaufpfeil von seinem Startpunkt aus, meist eine Eingabe, bis zu seinem Endpunkt. Der Ablaufpfeil läuft in der Regel von oben nach unten. Anweisungen werden als einzelne Blöcke dargestellt, bedingte Verzweigungen mit einer Raute, in der die Bedingung meist als Frage formuliert steht. Schleifen werden durch einen zum Schleifenbeginn zurücklaufenden Ablaufpfeil repräsentiert, Unterprogramme oder Subroutinen durch einen seitlich doppelt berandeten Block. Ebenso kann in einem Programmablaufplan Nebenläufigkeit dargestellt werden, z.B. wenn ein Prozess mehrere nebenläufige Prozesse initiieren kann („Parallelisierung“) oder sie zusammenführt („Synchronisation“).

Als ein erstes Beispiel eines Programmablaufplans ist in Abbildung 5.1 die binäre Suche dargestellt. Man erkennt direkt auf den ersten Blick die äußere Schleife, in der sich zwei bedingte Verzweigungen befinden. Die Stärke von Programmablaufplänen generell ist die visuelle und intuitive Darstellung des Ablaufs eines Algorithmus. Ein Nachteil dieser Beschreibungsform ist, dass sie für etwas kompliziertere Abläufe sehr schnell unübersichtlich werden kann.

[2] Sowohl die Punktnotation als auch die Referenz `this` gelten nicht für jede Programiersprache: In PHP beispielsweise wird statt der Punkt- die Zeigernotation verwendet, `obj->f(x)`; `this` heißt in Python `self`, in Visual Basic `Me`.

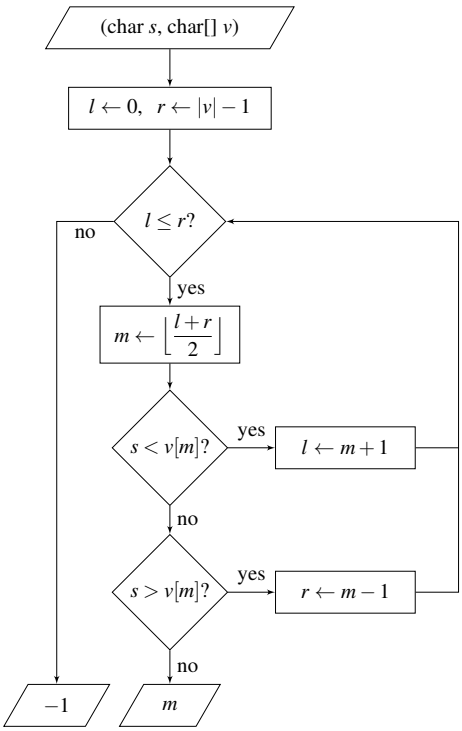


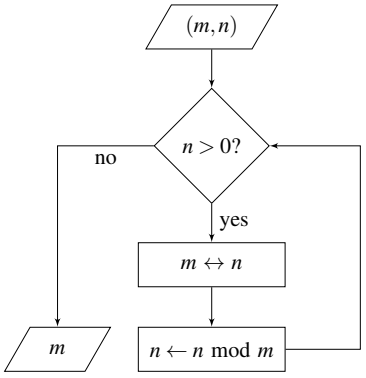
Abbildung 5.1: Programmablaufplan der binären Suche.

5.2 Erstes Beispiel: Der Euklid'sche Algorithmus

Einer der berühmtesten und mit mindestens 2300 Jahren ältesten Algorithmen ist ein Rechenverfahren, das nach dem griechischen Mathematiker Euklid (350–300 v.u.Z.) benannt ist, der *Euklid'sche Algorithmus*. Wahrscheinlich hat Euklid ihn zwar nicht erfunden, aber er ist der erste, von dem wir wissen, dass er ihn aufgeschrieben hat. Das Verfahren dient dazu, den größten gemeinsamen Teiler zweier natürlicher Zahlen zu finden. Er ist im Folgenden als Pseudocode und als Programmablaufplan dargestellt.

Algorithmus 5.2: Der Euklid'sche Algorithmus

```
1  algorithm euklid(m, n) {
2     input: zwei natürliche Zahlen m und n
3     output: der ggT(m, n)
4
5     while (n > 0) {
6        m ↔ n;
7        n ← n mod m;
8     }
9     return m;
10 }
```



Um sich die Funktionsweise eines Algorithmus zu verdeutlichen, sollte man als erste Maßnahme eine Wertetabelle für geeignete konkrete Eingabewerte erstellen. In einer solchen Wertetabelle repräsentieren die Spalten die wesentlichen beteiligten Variablen und Bedingungen, deren zeitliche Veränderungen von oben nach unten dargestellt wird. Werte werden dabei erst zu dem Zeitpunkt in eine Spalte eingetragen, zu dem sie sich verändern. Für Algorithmus 5.2 zum Beispiel sähe die Wertetabelle für die Eingabe $(m,n) = (6,4)$ wie in Tabelle 5.4 aufgelistet aus. Zur besseren Übersicht wurde jeder Schleifendurchlauf mit horizontalen Linien abgegrenzt.

OK, wir haben nun eine intuitive Vorstellung davon, was ein Algorithmus ist, und wir haben

Zeile	$n > 0$?	m	n
2		6	4
5	yes		
6		4	6
7			2
5	yes		
6		2	4
7			0
5	no		
9		$\boxed{2}$	

Zeile	$n > 0$?	m	n
2		48	60
5	yes		
6		60	48
7			48
5	yes		
6		48	60
7			12
5	yes		
6		12	48
7			0
5	no		
9		$\boxed{12}$	

Tabelle 5.4: Wertetabelle von Algorithmus 5.2 für die Eingaben $(m, n) = (6, 4)$ und $(48, 60)$.

mit der binären Suche und dem Euklid'schen Algorithmus zwei Beispiele kennengelernt. Aber was *genau* ist ein Algorithmus eigentlich?

5.3 Definition eines Algorithmus

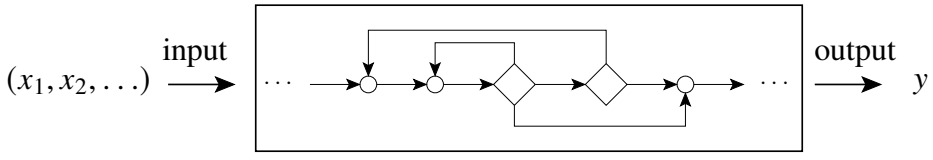
Ähnlich wie jede Datenstruktur am Ende immer aus primitiven Datentypen bestehen und sich in verschiedenen Komplexitätsgraden aus ihnen bilden, so setzen sich Algorithmen aus grundlegenden „Atomen" zusammen, den *Elementaroperationen*. Wir definieren sie als genau die in Tabelle 5.2 im Zusammenhang mit Pseudocode aufgeführten Operationen. Dazu zählen wir der Einfachheit halber auch die drei Anweisungen $\leftarrow$, $\leftrightarrow$, `input` und `output` bzw. `return`.

Mit Hilfe der vier aus den Grundlagen der Programmierung bekannten Kontrollstrukturen

- Sequenz (Aneinanderreihung von elementaren Operationen)
- bedingte Verzweigung (`if`)
- Schleifen (`while`, speziell auch die Zählschleifen `for` bzw. `loop`)
- Subroutinenaufrufe (Unterprogramme, speziell Rekursionen)

lässt sich so per Definition jeder Algorthmus darstellen. Formal also:

Definition 5.1 Ein *Algorithmus* ist eine endliche Abfolge elementarer Operationen, die ausschließlich mit Hilfe der obigen vier Kontrollstrukturen aus einer (möglicherweise leeren) Eingabe in endlicher Zeit eine eindeutige Ausgabe erzeugt.



Die Ausgabe ist dabei die Antwort auf ein gegebenes „relevantes" Problem. □

Mit dieser Definition hat ein Algorithmus also folgende wichtige Eigenschaften:

1. *(Terminierend)* Ein Algorithmus terminiert nach endlich vielen Schritten. Der Euklid'sche Algorithmus 5.2 beispielsweise terminiert, da spätestens ab dem zweiten Schleifendurchlauf in Zeile 6 n das Maximum der beiden Werte m und n angenommen und in Zeile 7 es stets echt kleiner wird, so dass die Schleifenbedingung in Zeile 5 irgendwann sicher

erreicht wird. Man kann sogar beweisen[3],[4], dass der Euklid'sche Algorithmus maximal N Schritte benötigt, wobei N durch

$$N \leqq 2{,}08 \ln [\max(m, n)] + 0{,}67 \tag{5.1}$$

beschränkt ist.

2. *(Deterministisch)* Jeder Schritt eines Algorithmus ist präzise festgelegt. Ein stochastischer Algorithmus (z.B. eine echte Zufallsfunktion) ist nach unserer Definition nicht möglich.

5.4 Diskussion

Wir haben hier eine „konstruktive" Definition eines Algorithmus gegeben, d.h. wir haben per Definition festgelegt, dass ein Algorithmus aus elementaren Operationen besteht, die mit den vier Kontrollstrukturen Sequenz, verzweigte Bedingung (**`if`**), Schleifen (vor allem **`while`**, aber auch **`for`** oder **`loop`**) und Subroutinenaufrufen zusammengesetzt werden können. Eine Art „Grenzfall" ist die Ackermann-Funktion, die wir bereits im ersten Semester kennengelernt haben[5]. Sie ist eine verschachtelte Rekursion und gehört zur Klasse der μ-rekursiven Funktionen. Diese zählen zu den mächtigsten Algorithmen und sind nicht mehr durch eine Zählschleife („Loop") programmierbar[6].

Doch es bleibt die Frage: Gibt es theoretisch nicht vielleicht Algorithmen, die ganz anders funktionieren? Könnte es nicht vielleicht andere Operationen oder Kontrollstrukturen geben, die auf neuen Rechnerarchitekturen laufen können, die ganz neuartige – und vielleicht viel mächtigere – Algorithmen ermöglichen? Fragen dieser Art behandelt die Berechenbarkeitstheorie *(computability theory)*. Mathematisch beweisen kann sie eine Antwort auf diese Fragen allerdings nicht. Nach der von Alonzo Church und Alan Turing bereits 1936 formulierten Church-Turing-These (auch Church'sche These genannt) ist aber *jeder* Algorithmus auf den uns heute bekannten Computern („Turing-Maschinen"[7]) berechenbar[8],[9]. Diese These ist zwar nur eine Vermutung, aber so gut wie alle Informatikerinnen und Informatiker nehmen sie als wahr an. Die Church-Turing-These rechtfertigt am Ende den hier verwendeten Ansatz einer „konstruktivistischen" Definition eines Algorithmus.

Eine zunächst unscheinbare, aber theoretische wesentliche Eigenschaft unserer Definition ist, dass Algorithmen `WHILE`-Schleifen beinhalten können oder, äquivalent dazu, verschachtelte Rekursionen. Diese Klasse von Algorithmen ist mächtiger als die der `LOOP`-Algorithmen, also der Algorithmen, die nur Zählschleifen umfassen. Programmiersprachen, die beliebige `WHILE`-Schleifen implementieren können, heißen „Turing-vollständig" oder „turingmächtig"[10].

Eine wichtige Klasse von Algorithmen ist in unserer Definition dagegen nicht enthalten, nämlich diejenige der *probabilistischen Algorithmen*. Für sie wird die Eigenschaft der Korrektheit abgeschwächt, so dass ein Algorithmus nur mit einer Wahrscheinlichkeit $\geqq \frac{1}{2}$ eine korrekte Lösung liefert. Ein prominentes Beispiel ist der „probabilistische Primzahltest" von Miller-Rabin, der für eine eingegebene Zahl prüft, ob sie prim ist; ergibt die Antwort „nein", so ist die Zahl

[3] Buchmann (2001):Theorem 1.8.6.

[4] Knuth (1998):§4.5.3, Corollary L (S. 360).

[5] de Vries und Weiß (2021):§3.

[6] Hoffmann (2009):Korollar 6.1.

[7] Die universelle Turing-Maschine ist ein theoretisches Konzept, das der englische Mathematiker Alan Turing 1936 veröffentlichte (Turing (1936–1937)) und nach dem prinzipiell alle heutigen Computer funktionieren. Obwohl es an sich nur sequenzielle Algorithmen vorsieht, gilt es grundsätzlich auch für nebenläufige und parallele Algorithmen und für Quantenalgorithmen (Harel und Feldman (2006):§10).

[8] Hoffmann (2009):§6.2.

[9] Sipser (2006):§3.3.

[10] Hoffmann (2009):§6.2.

sicher nicht prim, ergibt sie „ja“, so ist die Zahl nur mit der Wahrscheinlichkeit $\frac{1}{2}$ prim. Probabilistische Algorithmen widersprechen allerdings auch nicht der Church-Turing'schen These, da sie zur Klasse der nichtdeterministischen Turingmaschinen gehören[11],[12],[13].

[11] Arora und Barak (2009):§7.1.
[12] Harel und Feldman (2006):§11.
[13] Sipser (2006):§10.2.

6 Komplexität von Algorithmen

Kapitelübersicht

6.1 Verschiedene Algorithmen für dasselbe Problem

Meist gibt es mehrere Algorithmen zur Lösung eines gegeben Problems. Das Problem der Berechnung des größten gemeinsamen Teilers zweier natürlicher Zahlen muss zum Beispiel nicht mit dem Euklid'schen Algorithmus gelöst werden. Ein anderer bekannter Algorithmus basiert auf der Primfaktorzerlegung einer natürlichen Zahl. Ein *Primfaktor* einer natürlichen Zahl n ist eine Primzahl p, die n teilt, und die *Primfaktorzerlegung* von n ist die Darstellung von n als Produkt ihrer Primfaktoren:

$$n = {p_1}^{e_1} \cdot {p_2}^{e_2} \cdots {p_k}^{e_k}. \tag{6.1}$$

Hierbei geben die Exponenten e_i die *Vielfachheiten* der Primfaktoren p_i an, d.h. die Anzahl, wie oft der Primfaktor p_i in n enthalten ist. Die Primfaktorzerlegung ist stets eindeutig, wenn man die Primfaktoren der Größe nach sortiert, d.h. $p_1 < p_2 < \ldots < p_k$ gilt. Allgemein können wir also für eine gegebene natürliche Zahl n jedem seiner Primfaktoren p stets seine Vielfachheit e zuordnen, also eine Map

$$F_n = \{p_1 \to e_1,\ p_2 \to e_2,\ \ldots,\ p_k \to e_k\}, \tag{6.2}$$

bilden, die man in Array-Notation auch

$$F_n[p_1] = e_1,\ F_n[p_2] = e_2,\ \ldots,\ F_n[p_k] = e_k \tag{6.3}$$

schreiben kann. Zum Beispiel ist die Primfaktorzerlegung von 12 durch

$$12 = 2^2 \cdot 3, \qquad \text{also} \qquad F_{12} = \{2 \to 2,\ 3 \to 1\} \qquad \text{oder} \qquad F_{12}[2] = 2,\ \ F_{12}[3] = 1 \quad (6.4)$$

gegeben. Der größte gemeinsame Teiler zweier natürlicher Zahlen ergibt sich dann, indem wir die Primfaktoren nehmen, die in beiden Zerlegungen vorkommen, und als zugehörigen Exponenten den jeweils kleineren der beiden Exponenten. Als Beispiele betrachten wir die Bestimmung von ggT $(6, 4)$ und von ggT $(48, 60)$:

$$\begin{array}{l} 6 = 2^1 \cdot 3 \\ 4 = 2^2 \\ \hline \text{ggT} = 2^1 \quad = 2 \end{array} \qquad\qquad \begin{array}{l} 48 = 2^4 \cdot 3^1 \\ 60 = 2^2 \cdot 3^1 \cdot 5^1 \\ \hline \text{ggT} = 2^2 \cdot 3^1 \quad = 4 \cdot 3 = 12 \end{array}$$

Formal sieht man:

$F_6 = \{\ 2 \to 1,\ 3 \to 1\}$		
$F_4 = \{\ 2 \to 2\}$		
min :	1	0

$F_{48} = \{\ 2 \to 4, 3 \to 1\}$			
$F_{60} = \{\ 2 \to 2, 3 \to 1,\ 5 \to 1\}$			
min :	2	1	0

Man muss in dieser Notation also immer den kleineren der beiden Exponenten wählen, also der Zahlen hinter dem Pfeil. Algorithmus 6.1 stellt mit diesen Bezeichnungen den Ablauf der Berechnung dar.

Algorithmus 6.1: ggT mit Primfaktorzerlegung

```
algorithm primfaktorzerlegung(m, n) {
   F_m = primfaktoren(m); // Map p^e der Primfaktoren von m, mit F_m[p] = e
   F_n = primfaktoren(n); // Map p^e der Primfaktoren von n, mit F_n[p] = e
   d ← 1;
   foreach (p in F_m and p in F_n) {
      e ← min(F_m[p], F_n[p]);
      d ← d · p^e;
   }
   return d;
}
```

Er verwendet die Subroutine `primfaktoren`, die die Primfaktoren p^e der spezifizierten Zahl als Map $F = \{p \to e\}$ zurückgibt.

Algorithmus 6.2: Primfaktorzerlegung

```
algorithm primfaktoren(n) {
   input: eine natürliche Zahl
   output: Map F = {p → e} der Primfaktoren p von n und ihrer Vielfachheiten e
   p ← 2;
   while (p ≦ √n) {
      if (n mod p == 0) {
         e ← 1;
         n ← ⌊n/p⌋;
         while (n mod p == 0 and n > 1) {
            e ← e + 1;
            n ← ⌊n/p⌋;
         }
         F[p] ← e; // add p^e to F such that F[p] = e
```

```
        }
        if (p == 2) {
            p ← p + 1;
        } else {
            p ← p + 2;
        }
    }
    if (n > 1) { // n itself is prime
        F[n] ← 1; // add n^1 to F such that F[n] = 1
    }
    return F;
}
```

Eine vollständige Implementierung des Algorithmus ist im Anhang als Listing A.1 auf Seite 135 aufgeführt.

Mit den Algorithmen 5.2 und 6.1 haben wir damit zwei verschiedene Algorithmen, die dasselbe Problem lösen. Das ist auch an sich nichts Ungewöhnliches, es gibt fast immer mehrere Lösungswege für ein gegebenes Problem. Doch welcher davon ist nun der beste? Nach welchen Kriterien wollen wir Algorithmen vergleichen? Gibt es überhaupt quantifizierbare Maße – also Kennzahlen, wie die Wirtschaftswissenschaftler sagen – für die Qualität eines Algorithmus? Mit diesen Fragen beschäftigt sich die Algorithmenanalyse *(algorithmic analysis)*, siehe Tabelle 6.1. Während die Berechenbarkeitstheorie sich mit der grundsätzlichen Frage beschäftigt, welche

Gebiet	**Thema**	**Fragestellung**
Berechenbarkeitstheorie	Terminierung	Welche Probleme können überhaupt mit terminierenden Algorithmen gelöst werden?
Beweistheorie	Korrektheit (Effektivität)	Liefert ein gegebener Algorithmus stets das korrekte Ergebnis?
Komplexitätstheorie	Ressourcenbedarf (Effizienz)	Wieviel Laufzeit und Speicherplatz benötigt ein gegebener Algorithmus? Welche Probleme sind effizient lösbar (**P**) und welche nicht (**NP** oder gar **NP**-hard)?

Tabelle 6.1: Teilgebiete der Algorithmenanalyse (Harel und Feldman, 2006:§§5, 6, 8; Hoffmann, 2009:§§3.2.3, 6, 7; Sipser, 2006:S. 2f)

Probleme überhaupt mit Algorithmen gelöst werden können, behandelt die Beweistheorie unter Anderem die Frage, ob ein gegebener Algorithmus wirklich korrekt funktioniert, also „effektiv“ ist. Während in der Vergangenheit die Korrektheit eines Algorithmus individuell mathematisch bewiesen wurde, wird in der Beweistheorie darüber hinaus untersucht, inwieweit mathematische Beweise allgemein durch Computer ausgeführt werden können, also insbesondere Korrektheitsbeweise von Algorithmen. Die Komplexitätstheorie schließlich widmet sich der Frage nach dem Ressourcenbedarf eines Algorithmus, und die einzigen Ressourcen eines Algorithmus sind Laufzeit und Speicherplatz.

6.2 Komplexität als Maß der Effizienz

Die Komplexitätstheorie ist ein Teilgebiet der Theoretischen Informatik, das die Analyse des Ressourcenbedarfs eines Algorithmus in Abhängigkeit von seiner Eingabe behandelt. Zentral ist dabei der Begriff der Komplexität eines Algorithmus. Die Komplexität klassifiziert die

Ressourcenbedarfe eines gegebenen Algorithmus nach Ordnungsklassen, die Zeitkomplexität seine Laufzeit $T(n)$ und die Speicherplatzkomplexität seinen Speicherbedarf $S(n)$, beide jeweils als mathematische Funktion eines die Eingabegröße beschreibenden Parameters n. Der Speicherplatzbedarf umfasst hierbei nicht den Speicherplatz, den die Eingabe(n) und die Ausgabe benötigen, sondern denjenigen, der für den eigentlichen Ablauf, also temporär, gebraucht wird. Grundsätzlich können bei mehreren Eingabeparametern für den Algorithmus auch mehrere Parameter die Eingabegröße bestimmen, d.h. die Laufzeit T kann eine Funktion $T(n_1, n_2, ...)$ mehrerer Variablen sein, ebenso wie der Speicherplatzbedarf S eine Funktion $S(n_1, n_2, ...)$ sein kann. Allerdings interessiert bei der Komplexitätsanalyse meist nur der ungünstigste Fall, der *worst case*. Die erste Schwierigkeit bei einer solchen Analyse ist es also, den oder die richtigen Parameter zu identifizieren.

6.2.1 Erste Schritte einer Komplexitätsanalyse

Um die Laufzeit und den Speicherbedarf eines gegebenen Algorithmus abschätzen zu können, geht man im Allgemeinen in drei Hauptschritten vor[1]:

1. Ein *Eingabemodell* entwickeln, also eine sorgfältige Beschreibung der möglichen Eingaben und die für die Problemgröße wichtigen Parameter identifizieren.

2. Die *inneren Wiederholungen* des Algorithmus identifizieren, also Schleifen oder Rekursionen. Sie sind die wesentlichen Kontrollstrukturen, die die Laufzeit und den Speicherbedarf in Abhängigkeit von der Problemgröße beeinflussen.

3. Ein *Kostenmodell* definieren, das die Operationen der inneren Wiederholungen berücksichtigt. Üblicherweise ist es für die Laufzeit die Anzahl der elementaren Operationen, aber es kann in geeigneten Fällen auch eine Auswahl typischer Operationen sein. Das Kostenmodell für den Speicherplatzbedarf ist in der Regel die Anzahl der erforderlichen lokalen Variablen.

4. Eine Funktion T oder S als Kostenfunktion in Abhängigkeit der Parameter des Eingabemodells ermitteln oder mathematisch herleiten.

Beispiel 6.1 *(Komplexitäten der binären Suche)* Betrachten wir zunächst die binäre Suche (Algorithmus 5.1 auf Seite 52). Der Algrothmus hat zwei Eingabeparameter, d.h. die Laufzeit T und der Speicherbedarf S hängt im Allgemeinen von dem gesuchten Wort s und dem Verzeichnis v ab,

$$T = T(s, v), \qquad S = S(s, v).$$

Betrachten wir als Beispiel das Buchstabenverzeichnis (2.1) auf Seite 24. Für den gesuchten Buchstaben „H“ wäre der Algorithmus bereits mit einem einzigen Schleifendurchlauf fertig, für ein „Z“ dagegen erst nach vier Iterationen. (Warum?) Für die *exakte* Laufzeit benötigen wir also in der Tat die genauen Werte für s und v. Für die Abschätzung des ungünstigsten Falles allerdings spielt nur ein einziger Parameter eine Rolle, und zwar nach Satz 2.1 die Größe des Verzeichnisses, d.h. die Anzahl n der Einträge! Unser Eingabemodell umfasst also lediglich die Größe des Verzeichnisses. Dasselbe können wir von dem erforderlichen Speicherplatz erwarten. Wir halten also fest:

$$T_{\text{bs}} = T_{\text{bs}}(n), \qquad S_{\text{bs}} = S_{\text{bs}}(n) \qquad \text{mit } n = |v|. \tag{6.5}$$

[1] Sedgewick und Wayne (2014):§1.4.3.

Die innere Wiederholung der binären Suche besteht aus der `while`-Schleife. Verwenden wir als Kostenmodell die Anzahl der Vergleiche, so liefert uns Satz 2.1 eine Abschätzung für die ungünstigste Laufzeit,

$$T_{\mathrm{bs}}(n) \;\leqq\; 2 \cdot \lfloor 1 + \log_2 n \rfloor, \tag{6.6}$$

Die Konstante 2 würde sich bei einem anderen Kostenmodell ändern, die Anzahl der elementaren Operationen zum Beispiel ergäbe 8 (nämlich 2 oder 3 Vergleiche, 3 arithmetische Operationen, 1 oder 2 Zuweisungen). Was können wir über den Speicherbedarf aussagen? Der einzige zusätzlich zur Eingabe notwendige Speicherplatz von Algorithmus 5.1 sind die drei lokalen Indexvariablen l, r und m. Da diese Zahl immer gleich ist, egal ob das Verzeichnis drei Einträge hat oder eine Million, folgt

$$S_{\mathrm{bs}}(n) = \text{const.} \tag{6.7}$$

Für eine Implementierung in Java z.B. beträgt diese Konstante 12 Byte, d.h. $S_{\mathrm{bs}}(n) = 12$ Byte. Für die lineare Suche in einem unsortierten Verzeichnis überlegt man sich schnell:

$$T_{\mathrm{ls}}(n) \;\leqq\; \text{const} \cdot n, \qquad S_{\mathrm{ls}}(n) = \text{const.} \tag{6.8}$$

Auch hier können wir keine genaueren Angaben über die Konstanten machen, ohne die genaue Implementierung und die verwendete Hardware zu kennen. □

Diese kurze Diskussion der Suchalgorithmen in indizierten Vezeichnissen zeigt die Möglichkeiten, aber auch die grundsätzlichen Schwierigkeiten, auf die wir stoßen, wenn wir Laufzeiten und Speicherbedarf abschätzen wollen. Da wir am Ende Algorithmen, nicht aber konkrete Implementierungen oder verwendete Hardware bewerten wollen, brauchen wir eine andere mathematische Formulierung für die dazu „wesentlichen“ Aspekte. Was sind diese und wie kann man sie mathematisch bestimmen?

Zur ersten Frage: Wesentliche Aspekte bei der Laufzeit- und Speicherplatzbetrachtung ist das qualitative Wachstumsverhalten der Funktionen $T(n)$ und $S(n)$ für sehr große Eingaben, also sehr große Werte für n:

$$n \gg 1$$

Die Frage nach der angemessenen mathematischen Forulierung wird beantwortet durch die „asymptotische Notation“ von Funktionen.

6.3 Asymptotische Notation und Komplexitätsklassen

Um die Laufzeit und den Speicherbedarf eines Algorithmus in Abhängigkeit von seiner Eingabe zu bestimmen, müssten wir seine genaue Implementierung kennen, also seine Umsetzung in Maschinensprache und die Hardware, auf der er abläuft. Hierbei spielen zum Beispiel die Frequenz der CPU, die Programmiersprache und die genaue Realisierung der verwendeten Datenstrukturen. Wenn wir das alles berücksichtigen wollten, wäre eine Komplexitätsbetrachtung sehr mühsam und langwierig. Viel schlimmer aber wäre, dass wir unsere ursprüngliche Absicht, den eigentlichen Algorithmus zu betrachten und nicht konkrete Implementierungen, aus den Augen verloren hätten.

Was an der Effizienz eines Algorithmus eigentlich wichtig ist, ist das Wachstumsverhalten von der Laufzeit und dem Speicherbedarf bei sehr großen Eingaben. Uns interessiert gar nicht die genaue Laufzeit und der genau benötigte Speicherplatz, sondern lediglich eine vergröberte Sicht. Ähnlich wie wir zur Erkennung des Grenzverlaufs eines Landes eher eine Karte mit einem kleinen Maßstab bevorzugen, so brauchen wir zur Komplexitätsbetrachtung einen „herausgezoomten Blick“ oder eine „unscharfe Brille“, die uns die ganzen Details verwischt, aber die

wesentlichen Konturen erkennbar lässt. Dieses „Herauszoomen“ ermöglicht uns ein raffinierter mathematischer Apparat, die „asymptotische Notation“ mit Hilfe der Landau-Symbole.

Wir werden uns im Folgenden zunächst mit Funktionen T und S beschäftigen, die als Definitionsbereich die natürlichen Zahlen und als Wertebereich die positiven reellen Zahlen besitzen:

$$T : \mathbb{N} \to \mathbb{R}^+$$

mit

$$\mathbb{R}^+ = \{x \in \mathbb{R} |\ x > 0\} = (0, \infty). \tag{6.9}$$

Zum Beispiel:

$$T(n) = 2n^2 + n + 1, \qquad \text{oder} \qquad T(n) = n \ln n.$$

6.3.1 Die Komplexitätsklasse $O(g(n))$

Die *Komplexitätsklasse* $O(g(n))$ einer Funktion $g : \mathbb{N} \to \mathbb{R}^+$ ist definiert als die Menge aller Funktionen $f(n)$, die asymptotisch höchstens so schnell wachsen wie $g(n)$, d.h. für die zwei Konstanten $c \in \mathbb{R}^+$ und $n_0 \in \mathbb{N}$ existieren, so dass

$$f(n) \leqq cg(n) \quad \text{für alle } n \geqq n_0 \tag{6.10}$$

gilt. Wir schreiben in diesem Fall

$$\boxed{f(n) \in O(g(n))} \qquad \text{oder} \qquad \boxed{f(n) = O(g(n)).} \tag{6.11}$$

Mit anderen Worten: Eine Funktion $f(n)$ fällt in die Komplexitätsklasse $O(g(n))$, wenn $f(n)$ kleiner als ein konstantes Vielfaches von $g(n)$ ist, sobald n groß genug ist. Der Buchstabe

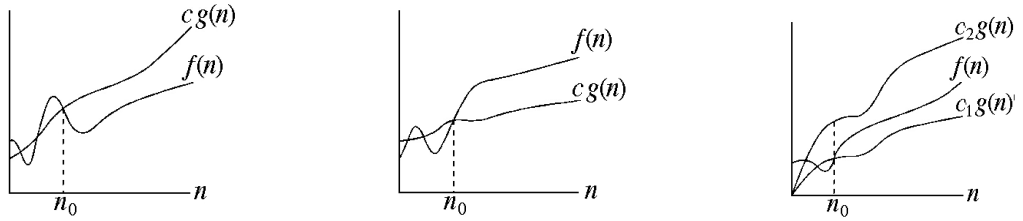


Abbildung 6.1: Die Landau-Symbole O, Ω, und Θ. (a) $O(g(n))$ enthält alle Funktionen $f(n)$, die asymptotisch höchtstens so schnell wachsen wie $g(n)$. (b) $\Omega(g(n))$ enthält alle Funktionen $f(n)$, die asymptotisch mindestens so langsam wachsen wie $g(n)$. (c) $\Theta(g(n))$ enthält alle Funktionen $f(n)$, die asymptotisch genauso schnell wachsen wie $g(n)$.

O heißt in diesem Zusammenhang *Landau-Symbol* oder „Groß-O“ *(„big-O“)*. Abbildung 6.1 (a) skizziert das O-Symbol. Die O-Notation wird meist dazu verwendet, eine *asymptotische Obergrenze* $g(n)$ für eine gegebene Funktion $f(n)$ anzugeben, die einfacher ist als $f(n)$ selber.

Beispiele 6.2 (i) Mit $g(n) = n^2$ und $f(n) = 2n^2 + n + 1$ gilt

$$2n^2 + n + 1 = O(n^2).$$

Beweis: Es gilt $2n^2 + n + 1 \leqq 4n^2$ für alle all $n \geqq 1$. (D.h. $c = 4$ und $n_0 = 1$ in (6.10); wir hätten aber auch z.B. $c = 3$ und $n_0 = 2$ wählen können).

(ii) Allgemeiner gilt für *jedes* quadratische Polynom

$$a_2n^2 + a_1n + a_0 = O(n^2). \tag{6.12}$$

Um dies zu zeigen, definieren wir $c = |a_2| + |a_1| + |a_0|$ und $n_0 = 1$; dann ist

$$a_2 n^2 + a_1 n + a_0 \leqq c n^2 \qquad \text{für alle } n \geqq n_0,$$

da jeder Summand kleiner als cn^2 ist.

(iii) *(Die b-adische Entwicklung)* Sei b eine ganze Zahl $b > 1$. Dann ist jede Zahl $n \in \mathbb{N}_0$ eindeutig durch eine endliche Summe

$$n = \sum_{i=0}^{m} a_i b^i \qquad \text{mit } a_i \in \{0, 1, \ldots, b-1\} \tag{6.13}$$

darstellbar. Der größte Index m hängt von n ab, $m = l_b(n)$, und heißt die *Länge* der b-adischen Entwicklung. Diese wichtige Resultat wird z.B. in[2] bewiesen und impliziert, dass jede Zahl eindeutig zur Basis b dargestellt werden kann, indem man jeden Koeffizienten durch ein einstelliges Zahlsymbol repräsentiert und sie absteigend hintereinander schreibt: $(a_n a_{n-1} \ldots a_1 a_0)_b$. Einige Beispiele:

$$\begin{aligned}
b = 2: &\quad 26 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 11010_2 \\
b = 3: &\quad 26 = 2 \cdot 3^2 + 2 \cdot 3^1 + 2 \cdot 3^0 = 222_3 \\
b = 4: &\quad 26 = 1 \cdot 4^2 + 2 \cdot 4^1 + 2 \cdot 4^0 = 122_4 \\
b = 5: &\quad 26 = 1 \cdot 5^2 + 0 \cdot 5^1 + 1 \cdot 5^0 = 101_5 \\
b = 16: &\quad 26 = 1 \cdot 16^1 + 10 \cdot 16^0 = 1A_{16}
\end{aligned}$$

Für die Länge der b-adischen Entwicklung einer Zahl n, also die Anzahl der für sie notwendigen Zahlsymbole, gilt

$$l_b(n) = \lfloor \log_b n \rfloor + 1 \leqq \log_b n + 1 = \frac{\ln n}{\ln b} + 1.$$

Für $n \geqq 3$ gilt $\ln n > 1$ und daher $\frac{\ln n}{\ln b} + 1 < \frac{\ln n}{\ln b} + \ln n = \left(\frac{1}{\ln b} + 1\right) \ln n$, d.h.

$$l_b(n) \;<\; c \, \ln n \qquad \text{für alle } n \geqq n_0 = 3, \text{ mit } c = \frac{1}{\ln b} + 1. \tag{6.14}$$

Damit folgt

$$l_b(n) = O(\ln n), \tag{6.15}$$

gleichgültig welchen Wert b hat. Damit ist die Anzahl der benötigten Stellen einer Zahl n in jedem Zahlsystem von der Komplexitätsklasse $O(\ln n)$. □

6.3.2 Die Komplexitätsklasse $\Omega(g(n))$

Während die O-Notation also eine obere asymptotische Schranke liefert, ist $\Omega(g(n))$ – sprich: „(Groß-) Omega" – die Menge aller Funktionen, für die $g(n)$ eine untere asymptotische Schranke bildet: Für zwei Funktionen $f, g : \mathbb{N} \to \mathbb{R}^+$ fällt $f(n)$ in die Komplexitätsklasse $\Omega(g(n))$, wenn zwei Konstanten $c \in \mathbb{R}^+$ und $n_0 \in \mathbb{N}$ existieren, so dass

$$f(n) \geqq c g(n) \qquad \text{für alle } n \geqq n_0 \tag{6.16}$$

gilt. Wir schreiben in diesem Fall

$$\boxed{f(n) = \Omega(g(n))} \qquad \text{oder} \qquad \boxed{f(n) \in \Omega(g(n))} \tag{6.17}$$

Jede Funktion $f(n)$ in $\Omega(g(n))$ wächst also asymptotisch mindestens so schnell wie $g(n)$, siehe Abbildung 6.1 (b).

Beispiel 6.3 Es gilt $\frac{1}{2}n^3 - n + 1 = \Omega(n^2)$, denn $\frac{1}{2}n^3 - n + 1 > \frac{1}{3}n^2$ für alle $n \geqq 1$. (D.h. $c = \frac{1}{3}$ und $n_0 = 1$ in (6.16)). □

[2] Forster (2008); Padberg (1996).

6.3.3 Die Komplexitätsklasse $\Theta(g(n))$

Wenn für eine Funktion $f(n)$ sowohl $f(n) \in O(g(n))$ als auch $f(n) \in \Omega(g(n))$ gilt, so wächst sie asymptotisch genauso schnell wie $g(n)$) und fällt in die Komplexitätsklasse $\Theta(g(n))$ – sprich: „(Groß-) Theta“ –, und wir schreiben dann

$$\boxed{f(n) = \Theta(g(n))} \quad \text{oder} \quad \boxed{f(n) \in \Theta(g(n))} \quad \text{oder} \quad \boxed{f(n) \sim g(n)} \tag{6.18}$$

Mit anderen Worten gehört $f(n)$ zur Klasse $\Theta(g(n))$, wenn zwei positive Konstanten c_1 and c_2 existieren, so dass sie für hinreichend große Werte von n zwischen $c_1 g(n)$ und $c_2 g(n)$ „eingeschnürt“ werden kann, siehe Abbildung 6.1 (c).

Beispiel 6.4 (i) Da sowohl $2n^2 + n + 1 = O(n^2)$ als auch $2n^2 + n + 1 = \Omega(n^2)$, gilt auch $2n^2 + n + 1 = \Theta(n^2)$.

(ii) Sei b eine natürliche Zahl mit $b > 1$ und sei $l_b(n) = \lfloor \log_b n \rfloor + 1$ die Länge der b-adischen Entwicklung einer natürlichen Zahl n. Dann gilt

$$(c-1)\ln n \;\leqq\; l_b(n) < c \ln n \qquad \text{für } n \geqq 3 \text{ und mit } c = \frac{1}{\ln b} + 1,$$

analog zur Ungleichung (6.14) in Beispiel 6.2 (iii). Damit gilt

$$l_b(n) \;=\; \Theta(\ln n). \tag{6.19}$$

Die Anzahl der Stellen einer Zahl ist also in jedem Zahlsystem logarithmisch von der Komplexitätsklasse $\Theta(\log n)$, wobei die Basis des Logarithmus gleichgültig ist. □

Die Komplexitätsklassen von Polynomen sind besonders einfach zu bestimmen. Ein *Polynom* $f_k(n)$ vom Grad k, für ein $k \in \mathbb{N}_0$, ist die Summe

$$f_k(n) = \sum_{i=0}^{k} a_i n^i = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \ldots + a_k n^k,$$

mit den konstanten Koeffizienten $a_i \in \mathbb{R}$. Dann können wir das folgende Theorem formulieren.

Theorem 6.5 *Ein Polynom vom Grad k mit $a_k > 0$ fällt in die Komplexitätsklasse $\Theta(n^k)$, d.h.*

$$f_k(n) = \Theta(n^k). \tag{6.20}$$

Beweis. Siehe[3]. Q.E.D.

Beispiel 6.6 Mit diesem Theorem fällt das Polynom $2n^2+n+1$ in die Komplexitätsklasse $\Theta(n^2)$. Es ist jedoch nicht in den folgenden Komplexitätsklassen enthalten:

$$2n^2 + n + 1 \neq O(n), \qquad 2n^2 + n + 1 \neq \Omega(n^3), \qquad 2n^2 + n + 1 \neq \Theta(n^3);$$

aber $2n^2 + n + 1 = O(n^3)$. □

Diese und einige andere nützliche asymptotische Abschätzungen sind in Tabelle 6.2 aufgeführt.

[3] Hoffmann (2009):Satz 7.2.

Beschreibung	Approximation
Harmonische Reihe	$\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} = \Theta(\log n)$
Stirling'sche Formel	$n! = \Theta\left(\sqrt{n}\left(\frac{n}{e}\right)^n\right) \subset O(n^n)$
Dreieckszahlen	$\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n = \frac{n\,(n+1)}{2} = \Theta(n^2)$
Binomialkoeffizienten	$\binom{n}{k} = \Theta(n^k)$, wenn k eine Konstante ist
Polynome	$\sum_{i=0}^{k} a_i\, n^i = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0 = \Theta(n^k)$
Geometrische Reihe	$\sum_{i=1}^{n} q^i = \frac{q^{n+1} - 1}{q - 1} = \Theta(q^n)$
Geometrische Reihe ($q = 2$)	$\sum_{k=1}^{n} 2^k = 2^{n+1} - 1 = \Theta(2^n)$

Tabelle 6.2: Nützliche asymptotische Approximationen, siehe Graham et al. (1994:S. 452)

6.4 Zeitkomplexität

Die *Laufzeit* $T(n)$ eines Algorithmus in Abhängigkeit von einer Eingabe der Größe n ist die Anzahl der ausgeführten elementaren Operationen („Rechenschritte"). Dieses Vorgehen ist üblich in der Informatik, denn diese Zahl ist hängt lediglich vom Algorithmus ab, nicht aber von der zugrundeliegenden Rechenmaschine. Die physikalische Laufzeit ergibt sich aus $T(n)$ näherungsweise, indem wir die durchschnittliche Dauer T_0 einer elementaren Operation verwenden und mit $T(n)$ multiplizieren.

Die algorithmische Analyse der Laufzeit T wird üblicherweise auf zwei Arten durchgeführt:

1. Eine *Worst-case Analyse* bestimmt die Laufzeit für Eingaben im ungünstigsten Fall, also eine obere asymptotische Schranke $O(g(n))$. Mit ihr hat man die Garantie, dass der Algorithmus für keine Eingabe länger laufen wird.

2. Eine *Average-case Analyse* bestimmt die Laufzeit einer typischen Eingabe, d.h. die statistisch zu erwartende Laufzeit.

In der Informatik werden Laufzeitfunktionen üblicherweise in vier Komplexitätsklassen klassifiziert, in logarithmische, polynomielle und exponentielle Laufzeiten, siehe Abbildung 6.2. Beispiele dazu sind in der folgenden Tabelle aufgelistet.

Komplexitätsklasse	Bezeichnung	Laufzeit $T(n)$
$O(1)$	konstant	konstante Funktionen c, mit $c \in \mathbb{R}^+$
$O(\log^k n)$	logarithmisch	$\ln n$, $\log_2 n$, $\log_{10} n$, $\ln^2 n$, ...
$O(n^k)$	polynomiell	n, n^2, n^3, ...
$O(k^n)$	exponentiell	2^n, e^n, 3^n, 10^n, ...

Hierbei ist $k \in \mathbb{R}^+$ eine beliebige positive Konstante.

Definition 6.7 Ein Algorithmus heißt *effizient*, wenn seine Laufzeit logarithmisch oder polynomiell ist, d.h. $T(n) = O(n^k)$ für eine positive Konstante k. □

Die Laufzeitanalyse eines Algorithmus kann eine ernsthaftes Problem darstellen, auch wenn der Algorithmus einfach ist. Die mathematischen Werkzeuge dafür erfordern Kenntnisse und Fertigkeiten der Kombinatorik, Algebra und der Wahrscheinlichkeitstheorie.

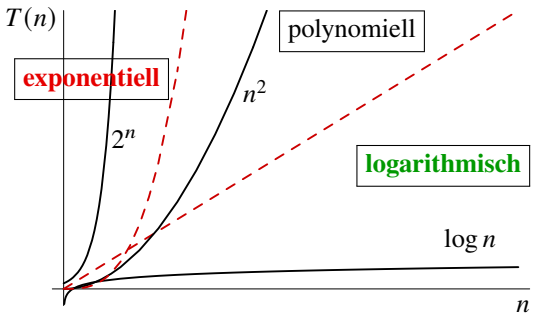


Abbildung 6.2: Qualitatives Verhalten typischer Funktionen der drei Komplexitätsklassen $O(\log n)$, $O(n^k)$, $O(k^n)$, mit $k \in \mathbb{R}^+$.

6.4.1 Laufzeiten der Kontrollstrukturen in asymptotischer Notation

Um die Laufzeit eines Algorithmus zu bestimmen oder abzuschätzen, müssen wir die Laufzeiten der einzelnen Bausteine eines Algorithmus kennen. Die Laufzeit einer einzelnen elementaren Operation ist unabhängig von der Größe der Eingabe, d.h. für sie gilt $T_{\text{op}}(n) = \Theta(1)$. Gleiches gilt daher für die Laufzeit einer Sequenz von Operationen bzw. für ganze Anweisungsblöcke, deren Anzahl an elementaren Operationen unabhängig von der Eingabe ist, $T_{\text{seq}}(n) = \Theta(1)$. Die einzigen Kontrollstrukturen also, die die Laufzeitklasse echt verändern können, sind Wiederholungsstrukturen, deren Wiederholungsanzahl von der Eingabegröße beeinflusst wird:

Zählschleifen (Loop, for, foreach)

Die Laufzeit einer `for`-Schleife

```
for (i: 1 to n) { ... }
```

mit einem Block aus von der Eingabegröße n unabhängigen Anweisungen ist linear, d.h.

$$T_{\text{for}}(n) = \Theta(n)$$

Entsprechend hat eine k-fach verschachtelte `for`-Schleife

```
for (i_1: 1 to n) {
   for (i_2: 1 to n) {
      ⋱
         for (i_k: 1 to n) { ... }
      ⋰
   }
}
```

mit ansonsten von der Eingabegröße n unabhängigen Anweisungen („. . .“) eine polynomielle Laufzeit von

$$T(n) = \Theta(n^k). \tag{6.21}$$

Betrachten wir als Beispiel die doppelt verschachtelte Schleife zur Addition zweier $(n \times n)$-Matrizen A und B:

```
algorithm summe(A, B) {
   input: zwei Matrizen A und B mit gleicher Zeilen- und Spaltenzahl
   output: die Matrix C = A + B
   for (i: 1 to n) {
      for (j: 1 to n) {
```

```
            c_ij ← a_ij + b_ij;
        }
    }
    return C;
}
```

Der Algorithmus hat eine quadratische Laufzeit, $T_{\text{summe}}(n) = \Theta(n^2)$.

Allgemeiner gilt für verschachtelte `for`-Schleifen mit unterschiedlich vielen Iterationen, also

```
for (i_1: 1 to n_1) {
    for (i_2: 1 to n_2) {
        ⋱
            for (i_k: 1 to n_k) { ... }
        ⋰
    }
}
```

mit ansonsten von den Eingabegrößen $(n_1, ..., n_k)$ unabhängigen Anweisungen („. . .“) eine polynomielle Laufzeit von

$$T(n_1, ..., n_k) = \Theta(n_1 \cdot \ldots \cdot n_k) \tag{6.22}$$

while-Schleifen

Die Laufzeitklasse einer `while`-Schleife ist im Allgemeinen nicht exakt zu bestimmen wie bei einer Zählschleife, da sie nach Konstruktion nicht nach einer zum Start bereits bekannten Anzahl von Iterationen beendet sein muss. Sie liefert daher oft nur eine Komplexitätsklasse

$$T_{\text{while}}(n) = O(f(n)),$$

keine Θ-Klasse. Betrachten wir beispielsweise den folgenden Algorithmus zur Berechnung des größten Primteilers einer Zahl:

```
algorithm gPT(n) {
    input: eine natürliche Zahl n
    output: der größte Primteiler von n
    t ← 1;
    while (t < n) {
        t ← t + 1;
        if (n mod t == 0) { n ← n/t; }
    }
    return t;
}
```

Die Anzahl der Iterationen hängt hier nicht von der Größe der Zahl ab, sondern von Anzahl und Größe ihrer Teiler. Die folgende Wertetabelle zeigt die Ergebnisse und die jeweilige Iterationszahl für verschiedene Zahlen an:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
gPT(n)	1	2	3	2	5	3	7	4	3	5	11	3	13	7	5	4	17	3	19	5
Iterationen	0	1	2	1	4	2	6	3	2	4	10	2	12	6	4	3	16	2	18	4

Die Anzahl Iterationen für eine Primzahl n ist stets $n-1$, für eine Zahl mit vielen kleinen Primteilern dagegen ist sie klein (für $n = 12$ oder 18 beispielsweise nur 2). Die bestmögliche Aussage, die wir also machen können, lautet

$$T_{\mathrm{gPT}}(n) = O(n),$$

also nur eine Abschätzung nach oben.

Für k verschachtelte `while`-Schleifen mit jeweils maximal n Iterationen im *worst case* erhalten wir entsprechend eine Laufzeitkomplexität von

$$T_{k-\mathrm{while}}(n) = O(n^k).$$

6.4.2 Subroutinenaufrufe und Rekursionen

Für einen Subroutinenaufruf muss natürlich die Laufzeit der Subroutine zur Laufzeit des sonstigen Algorithmus hinzuaddiert werden. Speziell für eine Rekursion ergibt sich dabei allerdings das Problem, dass die Laufzeit der Subroutine ja die eigene, also gerade zu berechnende Laufzeit ist. Aber bei einer stets terminierenden Rekursion beißt sich Katze eben nicht in den Schwanz, sondern man kann die sich dem Basisfall nähernde Eingabegröße ausnutzen. Wie das mathematisch behandelt werden kann, sehen wir im nächsten Kapitel.

6.5 Anwendungsbeispiele

Beispiel 6.8 *(Die binäre Suche)* Mit Theorem 2.1 auf Seite 24 wissen wir, dass die binäre Suche in einem sortierten indizierten Verzeichnis mit n Einträgen maximal $N_{\max} = 2\lfloor 1 + \log_2 n \rfloor$ Vergleiche benötigt, also eine Laufzeitkomplexität

$$T_{\mathrm{bS}}(n) = O(\log n) \tag{6.23}$$

besitzt. Wir können keine genauere Abschätzung machen, denn eine erfolglose Suche erfordert $N_{\max}$ Vergleiche, eine erfolgreiche Suche kann aber auch schon nach dem ersten Vergleich beendet sein, d.h. es gilt als beste asymptotische Abschätzung nach unten

$$T_{\mathrm{bS}}(n) = \Omega(1).$$

Für die Laufzeit der binären Suche können wir also insbesondere keine Θ-Klasse angeben. Insgesamt hat der Algorithmus also eine logarithmische Laufzeit im ungünstigsten Fall und ist daher effizient in Abhängigkeit zur Größe seiner Eingabe, der Größe n des sortierten Verzeichnisses. □

Beispiel 6.9 *(Operationen auf Datenstrukturen)* Betrachten wir die Zeitkomplexitäten der Operationen `contains`, `add`, and `remove` für einige Datenstrukturen mit n Einträgen. Um zum Beispiel einen spezifischen Eintrag in einer verketteten Liste zu finden, müssen wir am Anfang der Liste starten und laufen im ungünstigsten Fall durch die gesamte Liste – entweder erfolgreich beim letzten Eintrag oder für eine erfolglose Suche – und benötigen n Vergleiche. Im günstigsten Fall ist der gesuchte Eintrag am Beginn der Liste und wir sind beim ersten Vergleich fertig:

$$T^{\text{linked list}}_{\text{contains}}(n) = O(n), \qquad T^{\text{linked list}}_{\text{contains}}(n) = \Omega(1).$$

Hat man einen Eintrag in der Liste gefunden, so benötigt das Löschen des Eintrags eine bezüglich der Listengröße n konstante Laufzeit, nämlich das Umbiegen von zwei Zeigern (oder in einer

Datenstruktur	contains	add	remove
Verkettete Liste	$O(n)$	$O(1)$	$O(1)$
Array	$O(n)$	$O(n)$	$O(n)$
Sortiertes Array	$O(\log n)$	$O(n)$	$O(n)$

Tabelle 6.3: Laufzeitklassen für Operationen auf Datenstrukturen mit n Elementen.

doppelt verketteten Liste 4 Zeiger); dasselbe gilt für das Einfügen am Listenanfang (bzw. -ende), d.h.

$$T_{\text{remove}}^{\text{linked list}}(n) = O(1), \qquad T_{\text{add}}^{\text{linked list}}(n) = O(1).$$

(Beachte: Für asymptotisch wachsende Funktionen gilt $O(1) = \Theta(1)$.) In Tabelle 6.3 sind die Komplexitätsklassen für verschiedene Datenstrukturen aufgeführt. Jede Datenstruktur hat also ihre Stärken und ihre Schwächen gegenüber den anderen. □

Beispiel 6.10 *(Laufzeit des Euklid'schen Algorithmus)* Man kann beweisen,[4] dass der Euklid'sche Algorithmus 5.2 eine Laufzeit

$$T_{\text{Euklid}}(m, n) = O(\log^3(m + n)) \tag{6.24}$$

erfordert, wenn alle Iterationen und Divisionen auf Bitebene berücksichtigt werden. Da der Algorithmus auch für sehr große Eingaben m und n schon nach einer einzigen Iteration terminieren kann, wenn nämlich n ein Teiler von m ist, gilt als beste untere asymptotische Schranke $g(n) = 1$, d.h.

$$T_{\text{Euklid}}(m, n) = \Omega(1).$$

Für die Laufzeit des Euklid'schen Algorithmus kann man also insbesondere keine Θ-Klasse angeben. Insgesamt hat der Algorithmus also eine logarithmische Laufzeit im ungünstigsten Fall und ist daher effizient in Abhängigkeit zur Größe seiner Eingaben. □

6.6 Zusammenfassung

- Mit der Algorithmenanalyse wird die Korrektheit von Algorithmen bewiesen und die Komplexität von Laufzeit und Speicherplatzbedarf mit mathematischen Mitteln behandelt. Die Laufzeit eines Algorithmus ist dabei durch die Anzahl der für seinen Ablauf notwendigen elementaren Operation definiert, um von der konkreten Implementierung mit einer konkreten Programmiersprache und auf einer bestimmten Hardware zu abstrahieren. Entsprechend ist der Speicherplatzbedarf als die Anzahl an lokalen Variablen und Datenstrukturen definiert, die der Algorithmus neben den Eingabedaten benötigt.

- Die asymptotische Notation verschmiert die „Feinstruktur" einer Funktion und gibt den Blick frei auf ihr asymptotisches Wachstum für sehr große Argumente $n \gg 1$. Die Landau-Symbole O, Ω und Θ repräsentieren sogenannte Komplexitätsklassen, d.h. Mengen von Funktionen mit bestimmten asymptotische Schranken.

- Die asymptotische Notation vereinfacht die Algorithmenanalyse, denn die Landau-Symbole haben folgende Eigenschaften:

  – Die asymptotische Notation eliminiert Konstanten: z.B. können wir für $O(n) = O(n/2) = O(17n) = O(6n + 5)$ kurz $O(n)$ schreiben; dasselbe gilt für Ω und Θ.

[4] siehe (Cormen et al. (2001):S. 902), (Heun (2000):Theorem 7.3), (Schöning (1997):§7.5); nach Abschätzung (5.1) auf Seite 57 beträgt die Anzahl an Iterationen $O(\log \max[m, n])$.

  - Das O-Symbol liefert eine obere asymptotische Schranke: $O(1) \subset O(n) \subset O(n^2) \subset O(2^n)$. D.h., $3n^3 + 1 = O(n^5)$, aber $3n^3 + 1 \neq O(n^2)$.
  - Das Ω-Symbol liefert untere asymptotische Schranken: $\Omega(2^n) \subset \Omega(n^2) \subset \Omega(n) \subset \Omega(1)$. D.h., $3n^3 + 1\Omega(n^2)$, aber $3n^3 + 1 \neq \Omega(n^5)$.
  - Das Θ-Symbol liefert exakte asymptotische Schranken: $\Theta(1) \not\subset \Theta(n) \not\subset \Theta(n^2) \not\subset \Theta(2^n)$. D.h. $3n^3 + 1 = \Theta(n^3)$, aber $3n^3 + 1 \neq \Theta(n^5)$ und $3n^3 + 1 \neq \Theta(n^2)$.

- Suggestiv kann man die Landau-Symbole mit den Zeichen $\lesssim$, $\sim$ und $\gtrsim$ verknüpfen:

$T(n) = O(g(n))$:	„$T(n) \lesssim g(n)$“
$T(n) = \Theta(g(n))$:	„$T(n) \sim g(n)$“
$T(n) = \Omega(g(n))$:	„$T(n) \gtrsim g(n)$“

- Das O-Symbol vereinfacht die worst-case-Analyse von Algorithmen, das Θ-Symbol kann nur verwendet werden, wenn Komplexitätsklassen exakt bestimmt werden können. Für viele Algorithmen ist eine solche exakte asymptotische Schranke aber gar nicht möglich, insbesondere wenn sie eine while-Schleife verwenden.

- Meist interessiert man sich nur für wenige Komlexitätsklassen, die Klasse der logarithmischen Funktionen $O(\log n)$, der polynomiellen Funktionen $O(n^k)$ und der exponentiellen Funktionen $O(k^n)$, mit $k \in \mathbb{R}^+$.

- Ein Algorithmus mit polynomieller Zeitkomplexität $O(n^k)$ heißt effizient.

7 Komplexität von Rekursionen

Kapitelübersicht

Im Allgemeinen ist die Bestimmung der Zeitkomplexität eines rekursiven Algorithmus ein nichttriviales Problem. Der Grund liegt darin, dass die Laufzeitbetrachtung einer Rekursion in der Regel auf eine Rekursionsgleichung führt. Gleichungen dieser Art gehören zu den Differenzengleichungen und sind oft nur schwer oder gar nicht exakt lösbar. Allerdings haben wir mit einem Theorem über Rekursionen mit konstanten Schrittweiten und dem Hauptsatz der Rekursionsgleichungen, im Englischen das Master-Theorem, mächtige mathematische Werkzeuge, um die Laufzeiten einer weiten Klasse rekursiver Algorithmen asymptotisch abzuschätzen. Damit werden wir insbesondere präzise erkennen können, unter welchen Bedingungen eine Rekursion eine exponentielle Zeitkomplexität aufweist.

Nach einer kurzen Rekapitulation der grundlegenden Begriffe zu Rekursionen werden wir jedoch das Schema lernen, mit dessen Hilfe wir aus einem gegebenen rekursiven Algorithmus die Rekursionsgleichung für dessen Laufzeit ableiten können. Erst in einem zweiten Schritt lernen wir den Hauptsatz kennen, um ihn auf Beispiele anzuwenden.

7.1 Überblick über Rekursionen

Im ersten Semester lernten wir die Rekursion als spezielle Art des Subroutinenaufrufs kennen[1]. Damit eine Rekursion terminiert, muss es (mindestens) einen Basisfall geben, der den Aufrufbaum beendet, und einen Rekursionsschritt, in dem (mindestens) ein Rekursionsaufruf geschieht, und zwar derart, dass die Werte der Aufrufparameter auf den Basisfall führen. Als grundlegendes Beispiel wird meist die rekursive Berechnung der Fakultät einer übergebenen Zahl betrachtet:

```
1 algorithm fakultät(n) {
2   input: eine ganze Zahl n ∈ ℕ0
```

[1] de Vries und Weiß (2021):§3.4.

```
3    output: n!
4
5    if (n ≦ 1) { // Basisfall
6       return 1;
7    } else {    // Rekursionsschritt
8       return n · fakultät(n − 1);
9    }
10 }
```

Hier ist der Basisfall $n \leqq 1$, der Rekursionsaufruf in Zeile 8 geschieht mit einem kleineren Parameterwert $n - 1 < n$, strebt also sicher nach endlich vielen Rekursionsschritten auf den Basisfall zu.

Ein wichtiges Rekursionsschema sind die *linearen Rekursionen*. Eine Rekursion heißt linear, wenn in jedem Rekursionsschritt genau ein Rekursionsaufruf geschieht. Der Aufrufablauf bildet also eine lineare Kette von Aufrufen. Ein Beispiel dafür ist unsere obige rekursive Fa-

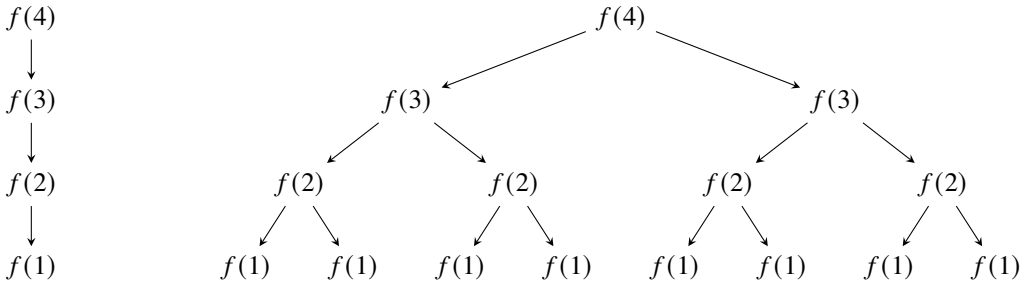


Abbildung 7.1: Lineare und verzweigende Rekursionen. Die lineare Rekursion links ist sogar primitiv, die verzweigende Rekursion rechts hat jeweils zwei Rekursionsaufrufe pro Rekursionsschritt.

kultätsberechnung. Wichtige lineare Rekursionen sind speziell die Endrekursionen, die den Rekursionsaufruf als letzte Anweisung des Rekursionsschritts ausführen, und die sogenannten *primitiven Rekursionen*[2], die speziell bei jedem Rekursionsschritt den für die Rekursionstiefe wesentlichen Paramter, z.B. n, um 1 senken,

$$f(n, \ldots) \to f(n-1, \ldots) \to \ldots \to f(0, \ldots).$$

also für alle n definiert sind[3].

Lassen wir dagegen mehrere Selbstaufrufe pro Rekursionsebene zu, so sprechen wir von einer *verzweigenden* oder *mehrfachen Rekursion (tree recursion)*, die einen verzweigten Aufrufbaum erzeugt. Von jedem Knoten gehen dabei genauso viele Äste ab wie Selbstaufrufe im Rekursionsschritt erscheinen. Ein Beispiel dafür ist die Lösung der Türme von Hanoi mit zwei Aufrufen je Rekursionsschritt. Ist mindestens eines der Argumente in einem Rekursionsaufruf selbst wieder ein Rekursionsaufruf, so liegt eine *verschachtelte Rekursion* vor, auch μ-Rekursion (sprich „mü-Rekursion“) genannt. Ein berühmtes Beispiel dafür ist die *Ackermannfunktion*:

```
algorithm ack(m, n) {
   if (m == 0) {
      return n + 1;
   } else if (n == 0) {
      return ack(m − 1, 1);
```

[2] Hoffmann (2009):S. 260.

[3] Vossen und Witt (2016):§9.3.1.

```
    } else {
        return ack(m − 1, ack(m, n − 1));
    }
}
```

Für weitere Details zu den Rekursionstypen und ihren Beziehungen zu den Berechenbarkeitsklassen siehe[4].

Zusammengefasst ergibt sich daraus die in Abbildung 7.2 dargestellte Hierarchie der verschiedenen Rekursionstypen. In der Theorie spielen die primitiven Rekursionen und die μ-

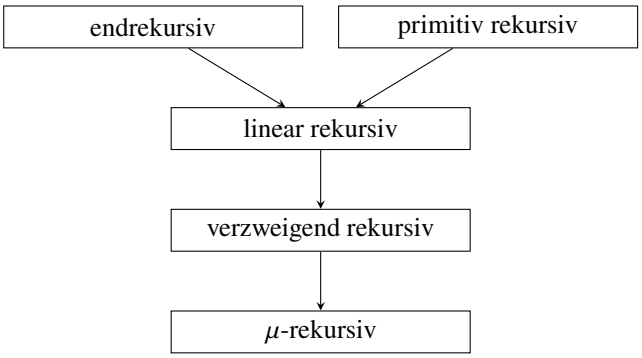


Abbildung 7.2: Hierarchie der Rekursionstypen. (μ-rekursiv ist hier synonym für verschachtelt rekursiv)

Rekursionen eine bedeutende Rolle. Viele verzweigend rekursive Algorithmen können grundsätzlich in eine primitive Rekursion und somit in eine Zählschleife umgeformt werden, allerdings dann in eine Schleife, die eine exponentiell höhere Schrittanzahl benötigt. Die Ackermannfunktion dagegen ist eine μ-Rekursion, die grundsätzlich *nicht* in eine primitive Rekursion umgeformt werden kann.

7.2 Aufstellen von Rekursionsgleichungen

Um die Laufzeitkomplexität $T(n)$ eines rekursiven Algorithmus zu bestimmen, muss man im Allgemeinen zunächst eine Gleichung für die Funktion $T(n)$ aufstellen, eine sogenannte *Rekursionsgleichung*. Im Wesentlichen spiegelt sie die Rekursionsstruktur des eigentlichen Algorithmus wider, d.h. wir müssen uns auf die Rekursionsschritte konzentrieren. Zum andern müssen wir diejenigen Parameter identifizieren, die für die Laufzeit des Algorithmus wesentlich sind, und für die Funktion $T(\cdots)$ die anderen weglassen. Hört sich kompliziert an, ist es aber nicht. Sehen wir uns dazu einige Beispiele an.

Beispiel 7.1 Betrachten wir den obigen Algorithmus zur rekursiven Berechnung der Fakultät. Der Aufrufbaum dieses Algorithmus ist in Abbildung 7.3 links skizziert. Da der Algorithmus linear rekursiv ist, ist der Aufrufbaum eine einfache Sequenz, die bei $f(n)$ beginnt und beim Basisfall $f(1)$ endet. Dort wird die Lösung $f(1) = 1$ an die nächsthöhere Aufrufebene zurückgegeben, der mit 2 multipliziert an die nächste Ebene zurückgegeben wird, usw. Wir zählen insgesamt genau n Aufrufebenen.

Wollen wir die Laufzeitkomplexität bestimmen, so müssen wir zunächst die Laufzeit für jede einzelne Aufrufebene berechnen. Für die untereste $n = 0$ haben wir eine konstante Laufzeit, sagen wir T_0, die die folgenden Operationen umfasst:

[4]Hoffmann (2009):§6.1.4, 6.2.

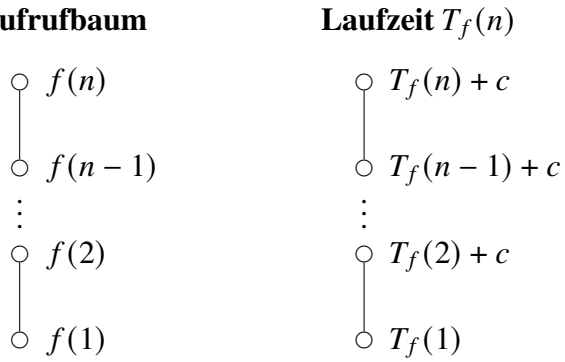


Abbildung 7.3: Aufrufbaum des rekursiven Algorithmus zur Berechnung der Fakultät (links) und die entsprechenden Laufzeiten $T_f(n)$.

- der Vergleich $n \leqq 1$;
- der Sprung in den Basisfall, also den ersten Zweig der if-Anweisung;
- die Rückgabe des Wertes 1 an die aufrufende Ebene.

Auch wenn wir die genaue Laufzeit nicht kennen (und sie auch kaum kennen können, denn die Laufzeit der notwendigen elementaren Operationen, also der Arithmetik und der Logik, hängen von der konkreten Rechnerarchitektur ab, auf der sie implementiert sind). Wichtig ist für uns, dass T_0 nicht abhängt von der Größe von n, also für die Größe unserer Eingabe konstant ist.

Was ist nun mit $T(2)$? Hier sehen wir, das die folgenden Operationen durchgeführt werden müssen:

- der Vergleich $n \leqq 1$;
- der Sprung in den Rekursionsschritt, also den zweiten Zweig der if-Anweisung;
- der Aufruf von $f(1)$;
- die Multiplikation des zurückgegebenen Wertes mit n;
- die Rückgabe des Wertes $2f(1)$ an die aufrufende Ebene.

Das ergibt eine konstante Laufzeit von c, und die Laufzeit insgesamt lautet

$$T(2) = T(1) + c.$$

Aber wir kennen ja bereits $T(1)$, und so berechnen wir $T(2) = T_0 + c$. Entsprechen können wir induktiv für jedes n aus dem vorhergehenden Wert von der darunterliegenden Ebene $n-1$ schließen: $T(n) = T(n-1) + c$. Zusammengefasst haben wir also zwei Fälle, den Basisfall $n \leqq 1$ und den Rekursionsschritt $n > 1$:

$$T(n) = \begin{cases} T_0, & \text{wenn } n = 0, \\ T(n-1) + c, & \text{sonst.} \end{cases} \tag{7.1}$$

Dies ist die Rekursionsgleichung der Laufzeitfunktion des Algorithmus. Sie entspricht (nicht zufällig!) genau der Struktur des zugrunde liegenden Algorithmus. □

Beispiel 7.2 *(Umrechnung von Dezimal- nach Dualsystem)* Im ersten Semester[5] behandelten wir kurz einen iterativen Algorithmus zur Umrechnung einer Zahl vom Dezimal- ins Dualsystem. Eine rekursive Variante lautet:

```
algorithm dezimalNachBinär(z, b) {
   input: Eine natürliche Zahl z und ein anfangs leerer String b
   output: Der Binärstring, der z im Dualsystem darstellt

   if (z ≦ 1) {
      return (z mod 2) ∘ b;
   } else {
      return dezimalNachBinär(⌊z/2⌋, (z mod 2) ∘ b);
   }
}
```

Von den zwei Parametern dieses Algorithmus beeinflusst nur der erste den Basisfall und damit die Laufzeit, d.h. $T = T(z)$. Die Rekursionsgleichung ergibt sich damit wie folgt:

$$T(z) = \begin{cases} T_0, & \text{wenn } z \leqq 1, \\ T(\lfloor z/2 \rfloor) + c, & \text{sonst.} \end{cases} \tag{7.2}$$

Die Konstante T_0 bezeichnet hier die Anzahl der elementaren Operationen im Basisfall, also der Vergleich $z \leqq 1$, die Berechnung z mod 2, die Stringverkettung $\ldots \circ b$ und die `return`-Anweisung. Die Konstante c umfasst diese Operationen und zusätzlich die Berechnung $\lfloor z/2 \rfloor$ und den Rekursionsaufruf. □

Beispiel 7.3 *(Die Türme von Hanoi)* Wir sind einem rekursiven Lösungsalgorithmus des Problems der Türme von Hanoi bereits im ersten Semester begegnet, siehe[6]. In Pseudocode lautet er:

```
algorithm hanoi(n, s, z, t) {
   if (n == 1) { // Basisfall: Turm mit nur einer Scheibe
      output(s ∘ " ⟶ " ∘ z);
   } else {
      hanoi(n − 1, s, t, z); // Turm (n − 1) temporär auf Stab t
      output(s ∘ " ⟶ " ∘ z); // unterste Scheibe auf ziel
      hanoi(n − 1, t, z, s); // Turm (n − 1) von tmp auf ziel
   }
}
```

Mit der Beobachtung, dass von den vier Parametern dieses Algorithmus nur der erste, also die Anzahl der Scheiben, Einfluss auf die Laufzeit hat, können wir auf eine nur von n abhängende Laufzeit schließen, d.h. $T = T(n)$. Die Rekursionsgleichung ergibt sich dann:

$$T(n) = \begin{cases} T_0, & \text{wenn } n = 1, \\ 2\,T(n-1) + T_0, & \text{wenn } n > 1. \end{cases} \tag{7.3}$$

Der Faktor zwei vor dem Wert $T(n-1)$ rührt daher, dass der Algorithmus im Rekursionsschritt zweimal aufgerufen wird. □

[5] de Vries und Weiß (2021):§2.2.
[6] de Vries und Weiß (2021):§3.4.

Beispiel 7.4 *(Binäre Suche rekursiv)* Eine rekursive Version der binären Suche lautet wie folgt.

```
/** Hüllfunktion.*/
algorithm binäreSuche(s, v[]) {
   return binäreSucheRekursiv(s, v[], 0, |v| − 1);
}

/** Helferfunktion.*/
algorithm binäreSucheRekursiv(s, v[], lo, hi) {
   if (lo > hi) return −1; // Basisfall 1: Suche erfolglos
   m ← ⌊(lo + hi)/2⌋;
   if (s == v[m]) { // Basisfall 2: Suche erfolgreich
      return m;
   } else if (s > v[m]) { // in der oberen Hälfte weitersuchen ...
      return binäreSucheRekursiv(s, v, m + 1, hi);
   } else {  // in der unteren Hälfte weitersuchen ...
      return binäreSucheRekursiv(s, v, lo, m − 1);
   }
}
```

Der Algorithmus besteht aus einer Hüllfunktion (engl.: *Wrapper*) und einer Helferfunktion (*Helper*). Mit der Hüllfunktion wird der eigentliche rekursive Algorithmus aufgerufen, sie stellt die notwendigen Startwerte ein und dient als Benutzerschnittstelle „nach draußen“. In Programmiersprachen, die innere Funktionen oder „Closures“ zulassen (z.B. JavaScript oder PHP), werden Helferfunktionen oft als solche implementiert, in Java dagegen oft als nach außen nicht sichtbare `private`-Methoden. Auf die Laufzeit des Algorithmus hat eine Aufteilung in Hüll- und Helferfunktion keine wesentlichen Auswirkungen (in der Hüllfunktion sind in der Regel $O(1)$ Operationen auszuführen, hier z.B. ist es nur ein Subroutinenaufruf), so dass wir uns auf die Laufzeit der Helferfunktion beschränken können. Deren Rekursionsgleichung lautet:

$$T(m) = \begin{cases} T_0, & \text{wenn } m \leqq 1, \\ T(\lfloor m/2 \rfloor) + c, & \text{wenn } m \geqq 1. \end{cases} \tag{7.4}$$

Der Parameter m ist hier der Mittelwert der beiden Betrachtungsgrenzen `lo` und `hi` des Algorithmus. Die Konstante T_0 umfasst die Anzahl der Operationen zur Prüfung des ersten Basisfalls und der Berechnung von m, die Konstante c repräsentiert zusätzlich die Laufzeit zur Prüfung des zweiten Basisfalls, des Vergleichs $s > v[m]$ und des Rekursionsaufrufs mit der Addition $m \pm 1$. □

Beispiel 7.5 *(Erweiterter Euklid'scher Algorithmus, rekursive Version)* Eine rekursive Variante des erweiterten Euklid'schen Algorithm, der neben dem größten gemeinsamen Teiler der beiden übergebenen Zahlen weitere nützliche Informationen zurückgibt. Er berechnet nämlich ganzzahlige Koeffizienten $x_0, x_1, x_2 \in \mathbb{Z}$, so dass

$$x_0 = \mathrm{ggT}\,(m, n) = x_1 m + x_2 n. \tag{7.5}$$

Hierbei können x_1 und x_2 null oder negativ sein. Diese Koeffizienten sind sehr nützlich zur Bestimmung von Lösungen linearer Diophant'scher Gleichungen und insbesondere zur Berechnung der modularen multiplikativen Inversen einer Zahl, die in unseren alltäglich für sicheren Internetzgang (HTTPS) genutzten Verschlüsselungsalgorithmen verwendet werden.

```
algorithm extendedEuclid(m,n) {
   input: zwei natürliche Zahlen m und n
   output: Ein Tripel (x_0, x_1, x_2) ganzer Zahlen mit x_0 = ggT(m,n) = x_1 m + x_2 n

   x ← [m, 1, 0];
   if (n == 0) {
      return x;
   } else {
      x ← extendedEuclid(n, m mod n);
      t ← x[1];
      x[1] ← x[2];
      x[2] ← t − ⌊m/n⌋ · x[2];
      return x;
   }
}
```

Die Rekursionsgleichung für diesen Algorithmus lautet

$$T(m,n) = \begin{cases} T_0, & \text{wenn } n = 0, \\ T(n,\ m \bmod n) + c, & \text{wenn } n \geqq 1. \end{cases} \tag{7.6}$$

Hier hängt also die Laufzeitfunktion von zwei Variablen ab. Zwar ist der Basisfall nur von der Variablen n abhängig, da diese bei jedem Rekursionsaufruf aus den beiden Parameter berechnet wird, „vermengen“ die beiden Variablen sich auf immer kompliziertere Weise. Wir können in diesem Fall also die Rekursionsgleichung nicht weiter vereinfachen. □

7.3 Asymptotische Lösungen von Rekursionsgleichungen

Wir betrachten im Folgenden für Laufzeitfunktionen $T : \mathbb{N}_0 \to \mathbb{N}_0$ Rekursionsgleichungen der Form

$$T(n) = \begin{cases} T_0, & \text{wenn } n \leqq n_0, \\ a\,T\big(s(n)\big) + f(n) & \text{sonst} \end{cases} \tag{7.7}$$

mit einer gegebenen Konstanten $a \in \mathbb{N}$ und Anfangswerten $n_0, T_0 \in \mathbb{N}_0$ sowie einer streng monoton fallenden Funktion $s : \mathbb{N}_0 \to \mathbb{N}_0$, der *Schrittweite* der Rekursionsaufrufe. Die Bedeutung der einzelnen Parameter ist dabei im Einzelnen:

- Die Zahl a ist die Anzahl der Selbstaufrufe des Algorithmus in seinem Rekursionsschritt.
- Die Anfangswerte erfüllen nach Definition die Gleichung $T(n_0) = T_0$.
- Die Schrittweitenfunktion $s(n)$ ordnet jedem Wert n das neue Argument $s(n)$ zu, mit dem die jeweils nächste Rekursion aufgerufen wird.

Übliche Beispiele für Schrittweiten sind konstant, multiplikativ und reflexiv mit einer Konstanten $b \in \mathbb{N}$:

$$\text{konstant:} \qquad s(n) = n - b \qquad (b \geqq 1) \tag{7.8}$$

$$\text{multiplikativ:} \qquad s(n) = \left\lfloor \frac{n}{b} \right\rfloor \quad \text{oder} \quad s(n) = \left\lceil \frac{n}{b} \right\rceil \qquad (b > 1) \tag{7.9}$$

$$\text{reflexiv:} \qquad s(n) = T(n - b) \qquad (b \geqq 1) \tag{7.10}$$

Primitive Rekursionen sind ein Spezialfall der Rekursionen mit konstanter Schrittweite $s(n) = n - 1$, allgemeine μ-rekursive Algorithmen dagegen können eine reflexive Schrittweite haben, wie beispielsweise die Ackermann-Funktion.

Beispiele 7.6 (i) Für die Rekursionsgleichung (7.1) sind $a = 1$, $n_0 = 0$ und $s(n) = n - 1$. Die Rekursionsschrittweite ist also konstant.

(ii) Für die Rekursionsgleichung (7.2) gilt $a = 1$, $n_0 = 1$ (und 0), und $s(n) = \lfloor n/2 \rfloor$. Die Rekursionsschrittweite ist also multiplikativ.

(iii) Für die Rekursionsgleichung (7.3) gilt $a = 2$, $n_0 = 1$, $s(n) = n - 1$. Die Rekursionsschrittweite ist also konstant. □

Das folgende Theorem liefert eine asymptotische Abschätzung für Rekursionen mit konstanter Schrittweite. Mit derartigen Algorithmen gelingt es leicht, exponentielle Laufzeiten zu erzeugen.

Theorem 7.7 (Rekursionen mit konstanten Schrittweiten) *Es sei $a \in \mathbb{N}$ und $f : \mathbb{N}_0 \to \mathbb{R}^+$ eine monoton steigende Funktion, d.h. $f(m) \leqq f(n)$ für $m < n$. Eine Laufzeitfunktion $T : \mathbb{N}_0 \to \mathbb{R}^+$, die der Rekursionsgleichung*

$$T(n) = \begin{cases} f(n_0), & \text{wenn } n = n_0, \\ a\,T(n-1) + f(n), & \text{wenn } n > n_0 \end{cases} \tag{7.11}$$

für einen Anfangswert $n_0 \in \mathbb{N}_0$ genügt, kann dann durch einen der vier folgenden Fälle abgeschätzt werden:

	Erster Fall	***Zweiter Fall***	***Dritter Fall***	***Vierter Fall***
Wenn	$f(n) = \Theta(1)$ $a = 1$	$f(n) = \Theta(1)$ $a \geqq 2$	$f(n) = \Omega(1)$ $a = 1$	$f(n) = \Omega(1)$ $a \geqq 2$
Dann gilt:	$T(n) = \Theta(n)$	$T(n) = \Theta(a^n)$	$T(n) = O\big(n\,f(n)\big)$	$T(n) = O\big(a^n\,f(n)\big)$

(7.12)

Beweis. Mit Gleichung (7.11) erkennt man, dass der Aufrufbaum von $T(n)$ insgesamt n Rekursionsebenen erzeugt, also a^n Basisfälle ausgeführt werden. In Rekursionsebene k werden also a^k-mal die Laufzeitwerte $f(n-k)$ benötigt,

$$f(n) + a\,f(n-1) + \ldots + a^{n-n_0} f(n_0) \;\leqq\; f(n) \sum_{k=n_0}^{n} a^k. \tag{7.13}$$

Für $a = 1$ handelt es sich um eine lineare Rekursion, für die sich über die Rekursionsebenen die jeweiligen Funktionswerte addieren,

$$f(n) + f(n-1) + \ldots + f(n_0) \leqq (n - n_0 + 1)\, f(n).$$

Hierbei gilt das Gleichheitszeichen genau dann, wenn $f(n)$ konstant ist, und die Summe hat die Komplexität $\Theta(n)$, ansonsten kann sie nur mit $O(n\,f(n))$ nach oben abgeschätzt werden. Für $a > 1$ gilt mit der Identität

$$\sum_{k=n_0}^{n} a^k = \sum_{k=0}^{n} a^k - \sum_{k=0}^{n_0-1} a^k = \frac{a^{n+1}-1}{a-1} - \frac{a^{n_0}-1}{a-1} = \frac{a^{n+1}-a^{n_0}}{a-1} = \underbrace{\frac{a}{a-1}}_{\text{const}}\, a^n - \underbrace{\frac{a^{n_0}}{a-1}}_{\text{const}} = \Theta(a^n),$$

wobei die zweite Gleichung sich aus den Identitäten der beiden geometrischen Reihen ergibt. Aus (7.13) folgt damit

$$f(n) + a\,f(n-1) + \ldots + a^{n-n_0} f(n_0) \;\leqq\; f(n)\,\Theta(a^n). \tag{7.14}$$

Auch hier gilt das Gleichheitszeichen genau dann, wenn $f(n)$ konstant ist, ansonsten kann die Summe links nur mit $O(a^n f(n))$ asymptotisch abgeschätzt werden. Q.E.D.

Für eine wichtige Klasse von Rekursionsalgorithmen, die Divide-and-Conquer-Algorithmen, kann die Laufzeitkomplexität durch den Hauptsatz der Laufzeitfunktionen, oft wie im Englischen auch als Master-Theorem bezeichnet[7], asymptotisch abgeschätzt werden:

Theorem 7.8 (Hauptsatz der Laufzeitfunktionen) *Es seien die Konstanten $a \geqq 1$ und $b > 1$ sowie die Funktion $f : \mathbb{N} \to \mathbb{R}^+$ gegeben. Dann gehören die Laufzeitfunktionen $T : \mathbb{N}_0 \to \mathbb{R}^+$ der Rekursionsgleichung*

$$T(n) = \begin{cases} T_0, & \text{wenn } n = n_0, \\ a\,T(\lfloor n/b \rfloor) + f(n) & \text{sonst} \end{cases} \tag{7.15}$$

mit den Anfangswerten T_0 und $n_0 \in \mathbb{N}_0$ zu den folgenden Laufzeitklassen, wenn f eines der entsprechenden asymptotischen Eigenschaften hat.

	Erster Fall	***Zweiter Fall***	***Dritter Fall***
Wenn	$f(n) = O(n^k)$ *für ein* $k < \log_b a$	$f(n) = \Theta(n^{\log_b a})$	$f(n) = \Omega(n^k)$ *für ein* $k > \log_b a$, *wobei* $a f(\lfloor \frac{n}{b} \rfloor) \leqq c f(n) \quad \forall n \gg 1$ *für eine Konstante* $c < 1$
Dann gilt:	$T(n) = \Theta(n^{\log_b a})$	$T(n) = \Theta(n^{\log_b a} \log n)$	$T(n) = \Theta(f(n))$

(7.16)

Die Aussagen gelten ebenso, wenn $\lfloor \frac{n}{b} \rfloor$ insgesamt oder teilweise durch $\lceil \frac{n}{b} \rceil$ ersetzt wird.

Beweis. Siehe[8]. In der Tat ist der Satz ein Spezialfall des Akra-Bazzi Theorems, das 1998 veröffentlicht wurde und ein sehr breites Spektrum an Rekursionsgleichungen abdeckt. Q.E.D.

7.4 Anwendungsbeispiele

Beispiele 7.9 (i) Die Rekursionsgleichung (7.1) für die Berechnung der Fakultät ist vom Typ (7.11) mit $a = 1$ und der konstanten Funktion $f(n) = c = \Theta(1)$, d.h. $T(n) = \Theta(n)$ nach Gleichung (7.12). Der Algorithmus hat also lineare Laufzeitkomplexität.

(ii) Die Rekursionsgleichung (7.2) für die Umrechnung vom Dezimal- ins Dualsystem ist von der Klasse (7.15) mit $a = 1$, $b = 2$, und der konstanten Funktion $f(n) = c = \Theta(1)$, d.h. $T(n) = \Theta(\log n)$ nach dem zweiten Fall in (7.16). Der Algorithmus hat also logarithmische Laufzeitkomplexität.

(iii) Die Rekursionsgleichung $T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + n$ ist vom Typ (7.15) mit $a = b = 2$, und der linearen Funktion $f(n) = n = \Theta(n^1)$, d.h. mit dem zweiten Fall in (7.16). $T(n) = \Theta(n \log n)$.

(iv) Die Rekursionsgleichung (7.3) des Algorithmus der Türme von Hanoi genügt der Gleichung (7.11) mit $a = 2$, d.h. der Algorithmus hat exponentielle Laufzeit $T(n) = \Theta(2^n)$.

□

[7] Cormen et al. (2001):§4.3.
[8] Cormen et al. (2001):§4.4.

7.5 Zusammenfassung

- Eine Rekursion ist eine Subroutine, die sich selbst während ihrer Ausführung aufruft. Sie besteht aus mindestens einem Basisfall, der keinen Rekursionsaufruf enthält, und einem Rekursionsschritt mit mindestens einem Rekursionsaufruf, der die Aurufparameter so variiert, dass sie nach endlich vielen Schritten zu dem (oder einem) Basisfall führen.

- Die Laufzeitkomplexität eines rekursiven Algorithmus wird mit Hilfe einer Rekursionsgleichung bestimmt, die direkt von dem Algorithmus hergeleitet werden kann. Die folgenden Klassen von Rekursionsgleichungen kommen üblicherweise vor (die Basisfälle der Übersicht halber weggelassen):

$$T(n) = T(n-1) + c, \quad T(n) = a\,T(n-1) + c, \quad T(n) = aT(\lfloor n/b \rfloor) + \Theta(n^{\log_b a})$$

  mit Konstanten $a, b \geqq 1$, $c > 0$. Mit den beiden in diesem Kapitel erwähnten Theoremen kann man deren asymptotische Laufzeit jeweils abschätzen:

$$T(n) = \Theta(n), \qquad T(n) = \Theta(a^n), \qquad T(n) = \Theta(n^{\log_b a}\ \log n)$$

8
Sortierung

Kapitelübersicht

Bei der Laufzeitbetrachtung der binären Suche in Satz 2.1 auf Seite 24 wurde der große Vorteil deutlich, den eine sortierte indizierte Datenstruktur bei der Suche nach einem Eintrag hat: Benötigt man in einer unsortierten Datenstruktur sowohl im Mittel als auch im schlimmsten Fall lineare Laufzeit $\Theta(n)$ bezüglich der Größe n der Datenstruktur, hat die binäre Suche eine höchstens logarithmische Laufzeit $O(\log n)$. Für sehr große Datenmengen macht sich dieser Vorteil geradezu dramatisch bemerkbar, schon für ein sortiertes Array mit einer Million Einträgen benötigt man nach Tabelle (2.2) beispielsweise maximal 40 Vergleiche, für ein unsortiertes Array derselben Größe dagegen können es eine Million Vergleiche sein, im Durchschnitt sind es immerhin noch 500.000.

Die Sortierung von Datenstrukturen stellte demnach in der Informatik von Anfang an ein prominentes Problem dar. Es wurden schon früh verschiedene Sortierverfahren entwickelt, von denen einige der bekanntesten in diesem Kapitel vorgestellt werden. Sie unterscheiden sich hinsichtlich ihrer Laufzeiten und ihrer Speicherbedarfe. Insbesondere ist ein wichtiges Unterscheidungskriterium, ob der Algorithmus *in-place* sortiert, also ohne eine temporäre weitere Datenstruktur die Einträge in der Eingabestruktur direkt austauscht.

Definition 8.1 Ein Algorithmus arbeitet *in-place*, oder *in-situ*, wenn er einen von den Eingabedaten unabhängigen Speicherbedarf hat, also $S(n) = O(1)$ gilt. □

Ein *in-place* Sortieralgorithmus verändert also die eingegebene Datenstruktur selber. Will man die originale Datenstruktur erhalten, so darf man keinen *in-place* Algorithmus verwenden.

8.1 Einfache Sortieralgorithmen

Bubblesort. Der *Bubblesort* auch *Austauschsortieren* oder *Sortieren durch Aufsteigen* genannt, ist ein Algorithmus, der iterativ durch die eingegebene Datenstruktur läuft und die

Einträge paarweise austauscht, wenn sie in der falschen Reihenfolge vorliegen. Der Algorithmus hat seinen englischen Namen („Blasensortieren“), da er bildlich die größeren Blasen nach oben (ans Ende der Datenstruktur) steigen lässt.

```
algorithm bubbleSort(a[]) {
   input: ein Array sortierbarer Objekte
   output: das Array wird in aufsteigender Reihenfolge sortiert (in-place)

   for (i = 1 to n − 1) {
      for (j = 0 to n − i − 1) {
         if (a[j] > a[j + 1]) {
            a[j] ↔ a[j + 1];
         }
      }
   }
}
```

Die Laufzeit des Algorithmus beträgt $T_{\text{bs}}(n) = O(n^2)$, da für die erste Iteration $n-1$ Vergleichsoperationen (innere j-Schleife) ausgeführt werden, für die zweite $n-1$ Vergleichsoperationen, usw. . . . , also insgesamt

$$T_{\text{bs}}(n) = (n-1) + (n-2) + \ldots + 1 = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = \Theta(n^2),$$

siehe Table 6.2. Eine etwas verbesserte Version ist der folgende Algorithmus `bubbleSort2`, der keine Laufzeit vergeudet, wenn das Array bereits sortiert vorliegt. Der wesentliche Trick besteht darin, das Array nur solange zu durchlaufen, bis keine Austauschungen mehr nötig sind:

```
algorithm bubbleSort2(a[]) {
   input: ein Array sortierbarer Objekte
   output: das Array wird in aufsteigender Reihenfolge sortiert (in-place)

   do {
      swapped ← false;
      for (j = 0 to n − i − 1) {
         if (a[j] > a[j + 1]) {
            a[j] ↔ a[j + 1];
            swapped ← true;
         }
      }
   } while (swapped);
}
```

Diese Version des Bubble-Sorts hat eine Laufzeitkomplexität $O(n^2)$, nicht jedoch $\Theta(n^2)$, da im günstigsten Fall (ein sortiertes Array) ein einziger Schleifendurchlauf genügt (d.h., es gilt $\Theta(1)$).

SelectionSort. Ein weiterer einfacher Sortieralgorithmus ist *selectionSort*. Wieder nehmen wir an, dass die zu sortierenden (und sortierbaren) Daten in einem Array $a[n]$ gespeichert sind.

```
algorithm selectionSort(a[]) {
   input: ein Array sortierbarer Objekte
   output: das Array wird in aufsteigender Reiehnfolge sortiert (in-place)
```

```
   for (i = 0 to n − 2) { // finde minimum von a[i], ..., a[n − 1]
      min ← i;
      for (j = i + 1 to n − 1) {
         if ((a[j] < a[min])) {
            min ← j;
         }
      }
      a[i] ↔ a[min];
   }
}
```

Die Anzahl der Iterationen der inneren Schleife (die j-Schleife) ist bei der ersten Iteration der äußeren Schleife $n-1$, bei der zweiten Iteration $n-2$, bei der dritten $n-3$ usw., d.h. die Laufzeitkomplexität ist hier wieder $T_{\text{sel}}(n) = \Theta(n^2)$, wie beim BubbleSort.

Insertionsort. Eine weitere Sortiermöglichkeit ist das Einsortieren eines Elements an der korrekten Stelle, ähnlich wie man es beim Aufnehmen der Karten eines Kartenspiels macht. Dies ist die Funktionsweise des *Insertionsort*, auch *Einfügesortieren* genannt.

```
algorithm insertionSort}(a[]) {
   input: ein Array sortierbarer Objekte
   output: das Array wird in aufsteigender Reihenfolge sortiert (in-place)

   for (i = 1 to n − 1) { // größere Werte jeweils nach oben
      r ← a[i]; j ← i;
      while (j > 0 and a[j] > r) {
         a[j] ← a[j − 1]; j ← j − 1;
      }
      a[j] ← r;
   }
}
```

Im ungünstigsten Fall wird die innere *while*-Schleife vom Ende des Arrays ($a[n-1]$) bis zum Anfang ($a[0]$) durchlaufen. Dies ist der Fall für ein absteigend sortiertes Array. Die Iterationen betragen dann

$$T_{\text{ins}}(n) \leqq 1 + 2 + \ldots + n = \sum_{k=1}^{n} k = \frac{(n+1)n}{2} = O(n^2).$$

Im Mittel läuft jeder Schleifendurchlauf nur halb so oft, d.h. die mittlere Laufzeitkomplexität beträgt

$$\bar{T}_{\text{ins}}(n) = \frac{1}{2}(1 + 2 + \ldots + (n-1)) = \frac{1}{2} \sum_{i=1}^{n-1} i = \frac{n(n-1)}{4} = \Theta(n^2).$$

8.2 Theoretische minimale Laufzeit eines Sortieralgorithmus

Gibt es effizientere Sortieralgorithmen? Die bisher betrachteten Sortierverfahren basieren auf paarweisen Vergleichen der Elemente, es sind sogenannte *vergleichsbasierte* Sortieralgorithmen. Es ist sofort klar, dass sie als mindestens $n-1$ Vergleiche benötigen, da ja jedes der n

Elemente mindestens einmal betrachtet werden muss. D.h. $\Omega(n)$ ist eine absolute Untergrenze für vergleichsbasierte Sortieralgorithmen. Man kann mathematisch beweisen[1]:

Theorem 8.2 *Ein vergleichsbasierter Sortieralgorithmus benötigt im ungünstigsten Fall stets mindestens*

$$T_{\text{sort}}(n) = \Omega(n \log n) \tag{8.1}$$

Vergleiche für eine Datenstruktur mit n Elementen.

Diese Aussage gilt übrigens nicht für Sortierverfahren, die nicht-vergleichsbasiert sind. Ein bemerkenswertes Beispiel dieser Art ist der *pigeonhole sort*, der ein Array natürlicher Zahlen sortiert. Der Algorithmus ist eine spezielle Variante des *BucketSort*[2].

```
algorithm pigeonholeSort(a[]) {
   input: ein Array natürlicher Zahlen
   output: das Array wird in aufsteigender Reihenfolge sortiert (in-place)

   // 1. determine the number of pigeonholes:
   min ← a[0]; max ← a[0];
   for (x in a) {
      if (min > x) min ← x;
      if (max < x) max ← x;
   }
   size ← max - min + 1;
   holes ← [0, 0, ..., 0]; // fill with size zeros

   // 2. Fill the pigeonholes by counting:
   for (x in a) {
      holes[x - min] ← holes[x - min] + 1;
   }

   // 3. Put the elements back into the array in order.
   i ← 0;
   for (count = 0 to size - 1;) {
      while (holes[count] > 0) {
         a[i] ← count + min; // the index of the hole is the correct value!
         i ← i + 1;
         holes[count] ← holes[count] - 1;
      }
   }
}
```

Der pigeonholeSort hat eine Laufzeitkomplexität $O(n + k)$ und eine Speicherkomplexität $O(k)$, wobei $k = \max a - \min a$ die Differenz zwischen dem kleinsten und dem größten Element des Arrays ist. Gilt also beispielsweise $0 \leqq a[i] \leqq O(n)$, so sind sowohl Laufzeit- als auch Speicherkomplexität $O(n)$.

[1] Güting (1997):§6.4.
[2] Heun (2000):§2.7.

8.3 Schnelle Sortieralgorithmen

Haben wir oben gerade erkannt, dass es theoretisch optimale vergleichsbasiere Sortieralgorithmen mit einer Laufzeitkomplexität $O(n \log n)$ geben könnte, stellt sich natürlich sofort die Frage: Gibt es solche Algorithmen denn auch tatsächlich? Die Antwort ist: Ja! Wir werden in diesem Abschnitt drei Algorithmen kennenlernen, von denen zwei im ungünstigsten Fall eine solche Laufzeitkomplexität haben, und einer im durchschnittlichen Fall.

8.3.1 Mergesort

Mergesort ist ein klassischer rekursiver Divide-and-conquer-Algorithmus. Der Basisifall ist hier trivial, es wird nämlich nichts getan, die Hauptarbeit wird im Conquer-Schritt ausgeführt, der „merge“ heißt.

```
algorithm mergesort(a) {
   input: ein Array a
   output: ein aufsteigend sortiertes Array

   if (|a| == 1) { // base case: nothing to do!
      return a;
   } else {
      m ← ⌊|a|/2⌋;
      l ← [a[0], ..., a[m − 1]]; // left half of a
      r ← [a[m], ..., a[|a| − 1]]; // right half of a
      l ← mergesort(l);
      r ← mergesort(r);
      return merge(l, r); // merge l and r
   }
}
```

(Beachten Sie die JSON-Notation für Arrays, wie sie in Abschnitt 5.1.1 auf Seite 53 spezifiziert ist.) Die Subroutine *merge* ist gegeben durch:

```
algorithm merge(l[], r[]) {
   input: two sorted arrays
   output: an array m with the entries of l and r sorted in ascending order

   i_l ← 0; i_r ← 0; i_m ← 0; // indices for l, r and merged array m
   while (i_l < |l| and i_r < |r|) {
      if (l[i_l] < r[i_r]) {
         m[i_m] ← l[i_l];
         i_l ← i_l + 1;
      } else {
         m[i_m] ← r[i_r];
         i_r ← i_r + 1;
      }
      i_m ← i_m + 1;
   }
   //copy remaining elements from both halves:
   while (i_l < |l|) {
      m[i_m] ← l[i_l];
      i_l ← i_l + 1;
```

```
        i_m ← i_m + 1;
    }
    while (i_r < |r|) {
        m[i_m] ← r[i_r];
        i_r ← i_r + 1;
        i_m ← i_m + 1;
    }
    return m;
}
```

Die Arbeitsweise des Algorithmus `mergeSort(`l, r`)` lässt sich wie folgt zusammenfasssen (siehe Abbildung 8.1):

- *Divide*: Das Teilarray $\big[a[l], \ldots, a[r]\big]$ wird in zwei (fast) gleich große Array geteilt;
- *Conquer*: Durch zwei Rekursionsaufrufe jeweils eins der beiden Arrays sortieren lassen;
- *merge* fügt beide sortierten Teilarrays zu einem sortierten Array zusammen.

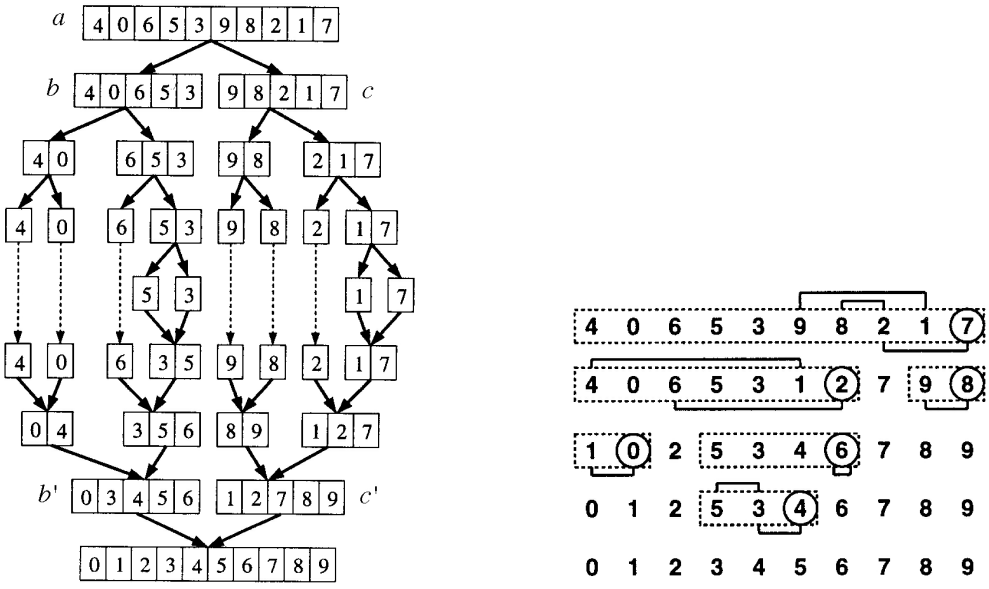


Abbildung 8.1: Left figure: mergeSort for an array of 10 elements. Right figure: quickSort for an array of 10 elements

Mergesort kann auch auf Datenstrukturen ohne wahlfreien Zugriff *(random access)* angewendet werden, zum Beispiel verkettete Listen, da er rein sequenziell arbeitet. Als Rekursion ist er auch auf parallelen Rechnersystemen oder Mehrkernprozessorrechnern geeignet.

Da Mergesort ein ausgeglichener Divide-and-Conquer-Algorithmus ist, können wir recht leicht seine Rekursionsgleichung aufstellen. Da für jeden Rekursionsaufruf von Mergesort die Größe n des zu sortierenden Array durch die Begrenzungsindizes l und r gegeben ist, nämlich $n = r - l + 1$, genügt der Mergesort der Rekursionsgleichung

$$T_{ms}(n) = \begin{cases} \Theta(1), & \text{wenn } n \leqq 0, \\ 2\,T\,(\lfloor n/2 \rfloor) + \Theta(n) & \text{sonst,} \end{cases} \tag{8.2}$$

denn es gibt pro Rekursionsschritt zwei Rekursionsaufrufe mit (so gut wie) gleich großen Teilarrays der Größe $\lfloor n/2 \rfloor$ (bzw. $\lceil n/2 \rceil$) und einem Merge-Schritt der Komplexität $\Theta(n)$. Mit

$a = b = 2$ und $f(n) = \Theta(n)$ können wir daher den zweiten Fall Wir können daher den zweiten Fall des Master-Theorems 7.8 (Seite 81) anwenden und erhalten als Laufzeitkomplexität für den Mergesort:

$$\boxed{T_{ms}(n) = \Theta(n \log n).} \tag{8.3}$$

8.3.2 Quicksort

Quicksort ist, wie der Mergesort, ein rekursiver Divide-and-Conquer-Algorithmus. Während der Mergesort jedoch einen trivialen Divide-Schritt hat und den größten Teil der Arbeit dem Merge-Schritt überlässt, wird im Quicksort die Hauptarbeit im Divide-Schritt ausgeführt, der Merge-Schritt ist dafür trivial. Obwohl Quicksort im ungünstigsten Fall eine recht schlechte Laufzeitkomplexität von $O(n^2)$ hat, ist er der wahrscheinlich meistimplementierte Sortieralgorithmus. Er ist vergleichsweise alt, er wurde bereits 1962 von C.A.R. Hoare entwickelt. Sei wieder $a = [a_0, \ldots, a_{n-1}]$ das zu sortierende Array. Dann arbeitet der Algorithmus `quicksort(`$a[]$, l, r`)` grob betrachtet mit den folgenden Schritten:

- *Divide*: Das r-elementige Array $\big[a[l], \ldots, a[r]\big]$ wird in zwei Teilarrays $\big[a[l], \ldots, a[p-1]\big]$ und $\big[a[p+1], \ldots, a[r]\big]$ unterteilt, so dass jedes Element des ersten Arrays kleiner ist als jedes Element des zweiten: $a[i] \leqq a[p]$ für $l \leqq i < p$ und $a[j] \geqq a[p]$ with $p < j \leqq r$. Dieser Schritt heißt *partition*, und das Element $a[p]$ heißt *Pivotelement*.[3] Üblicherweise wird das Element $a[r]$ als Pivotelement ausgewählt, grundsätzlich ist es jedoch beliebig wählbar.
- *Conquer*: Die beiden Arrays werden durch jeweils einen Rekursionsaufruf von Quicksort sortiert;
- *Merge*: Es bleibt nichts mehr zu tun, beide Teilarrays sind ja separat sortiert worden.

In Pseudocode lautet er:

```
algorithm quicksort(a[], l, r) {
   input: ein sortierbares Array mit seinen Begrenzungsindizes
   output: Das Array ist in aufsteigend sortiert (in place)

   // Base case l ≧ r: nothing to do!
   if (l < r) {
      p ← partition(a, l, r); // index of pivot element
      quicksort(a[], l, p − 1);
      quicksort(a[], p + 1, r);
   }
}
```

Hierbei ist der Teilalgorithmus *partition* gegeben durch:

```
algorithm partition(a[], l, r) {
   input: ein sortierbares Array mit seinen Begrenzungsindizes
   output: der Index p des Pivotelements

   i ← l − 1; j ← r;
   while (i < j) {
      i ← i + 1;
```

[3] *pivot*: engl. für Dreh-, Angelpunkt; Schwenkungspunkt

```
        while (i < j and a_i < a_r) { // "i-loop"
            i ← i + 1;
        }
        j ← j − 1;
        while (i < j and a_j > a_r) { // "j-loop"
            j ← j − 1;
        }
        if (i ≧ j) {
            a_i ↔ a_r;
        } else {
            a_i ↔ a_j;
        }
    }
    return i;
}
```

Im „i-loop“ zeigt der Index i auf das erste Element $a[i]$ von links, das größer oder gleich $a[r]$ ist, d.h. $a[i] \geqq a[r]$ (wenn $i < j$). Nach Beendigung des „j-loops“ zeigt j auf das erste Element $a[j]$ von rechts, das kleiner als $a[r]$ ist (wenn $j > i$). Die Subroutine *partition* plaziert also jeweils das Pivotelement $a[p]$ auf seine endgültige korrekte Stelle (die auch in der Folge nicht mehr geändert wird). Siehe Abb. 8.1.

Komplexitätsanalyse von *Quicksort*

Die Komplexitätsanalyse von *Quicksort* ist nicht trivial. Die Schwierigkeit ergibt sich, da die zu ermittelnde Position p des Pivotelements von dem konkreten Array abhängt. Es ist im Allgemeinen nicht in der Mitte des Arrays, so dass Quicksort – anders als Mergesort – kein ausgeglichener Divide-and-Conquer-Algorithmus ist. Unser Master-Theorem ist also leider nicht anwendbar, denn wir können nicht einfach $a = b = 2$ setzen.

Wir können aber versuchen, die Rekursionsgleichung für Quicksort aufzustellen und zumindest Abschätzungen darüber erhoffen. Der wichtige Arbeitsschritt ist der Divide-Schritt mit der Subroutine *partition*. Deren äußere Schleife wird dabei genau einmal (!) durchlaufen, während die beiden inneren sich genau auf $n-1$ Iterationen aufsummieren; Die Laufzeit für den Basisfall, also ein Array der Länge $n = 1$, ist konstant T_0, und für jeden Rekursionsschritt benötigen wir eine konstante Laufzeit T_1 neben der Laufzeit der Rekursionsaufrufe. Insgesamt erhalten wir also die Rekursionsgleichung für den Quicksort:

$$T_{\text{qs}}(n) = \begin{cases} T_0 & \text{wenn } n = 1, \\ \underbrace{(n-1) + T_1}_{\text{divide}} + \underbrace{T(p-1) + T(n-p)}_{\text{conquer}} + \underbrace{0}_{\text{merge}} & \text{sonst} \quad (\text{mit } 1 \leqq p \leqq n). \end{cases} \tag{8.4}$$

Wir können aber durch Fallunterscheidungen separat den ungünstigsten, den besten und den durchschnittlichen Fall betrachten.

Worst case

Im ungünstigsten Fall ist $p = 1$ oder $p = n$. Damit ergibt sich die Rekursionsgleichung

$$T_{\text{qs worst}}(n) = \begin{cases} T_0 & \text{wenn } n = 1, \\ (n-1) + T_{\text{worst}}(n-1) + T_1 & \text{sonst.} \end{cases} \tag{8.5}$$

Damit können wir den linearen Rekursionsbaum erkennen und die Laufzeiten als Wertetabelle „von unten“ bestimmen:

$$\begin{aligned} T_{\text{qs worst}}(1) &= T_0 \\ T_{\text{qs worst}}(2) &= 1 + T(1) + T_1 = 1 + T_1 + T_0 \\ T_{\text{qs worst}}(3) &= 2 + T(2) + T_1 = 2 + 1 + 2T_1 + T_0 \\ &\vdots \\ T_{\text{qs worst}}(n) &= \sum_{k=1}^{n-1} k + (n-1)T_1 + T_0 = \binom{n}{2} + (n-1)T_1 + T_0 = \Theta(n^2). \end{aligned}$$

Damit ist Quicksort im ungünstigsten Fall nicht effizienter als Bubblesort oder Insertionsort. Der ungünstigste Fall wird aber nur in dem speziellen Fall eintreten, wenn das Array bereits sortiert ist (egal wie) und das Pivotelement in jedem Schritt das kleinste oder größte Element des Rekursionsschrittes ist.

Best und average case

Der bestmögliche Fall ist einfach zu betrachten, denn hier ist das Pivotelement p immer in der Mitte des Arrays, und damit ist Quicksort ein ausgeglichener Divide-and-Conquer-Algorithmus mit einem linearen Divide-Schritt. Wie für den Mergesort können wir also mit $a = b = 2$ und $f(n) = \Theta(n)$ den zweiten Fall des Master-Theorems 7.8 anwenden:

$$T_{\text{qs best}}(n) = \Theta(n \log n). \tag{8.6}$$

Der durchschnittliche Fall ist schwieriger zu beweisen, aber man kann zeigen[4]:

$$T_{\text{qs average}}(n) = \Theta(n \log n). \tag{8.7}$$

8.3.3 Heapsort

Der Name des Heapsort rührt von der Datenstruktur des Heaps her. Erinnern wir uns an die wesentlichen Eigenschaften eines Heaps aus Abschnitt 3.2. Es ist die insbesondere die effiziente Implementierbarkeit als ein Array, die der Heapsort wesentlich ausnutzt. Er ist, zumindest als sequenziell implementierter Algorithmus, der beste bekannte Sortieralgorithmus hinsichtlich Laufzeit- und Speicherplatzkomplexität. Die grundlegende Idee des *Heapsort* ist sehr einfach:

1. Die zu sortierenden n Elemente werden in einen Heap einsortiert; dies erfordert eine Laufzeitkomplexität von $O(n \log n)$.

2. Das jeweilige Heap-Minimum wird aus dem Heap entfernt und der Heap so verkleinert; für jedes einzelne Minimum erfordert dies $O(\log n)$ Iterationen, und da dies für jeden der n Knoten geschieht, haben wir eine Laufzeitkomplexität von $O(n \log n)$.

Sei wieder $a = \big[a[0], \ldots, a[n-1]\big]$ ein Array von n sortierbaren Elementen.

Definition 8.3 Ein Teilarray $\big[a[i], \ldots, a[k]\big]$, mit $1 \leqq i \leqq k \leqq n$, heißt *Subheap*, wenn

$$\left.\begin{array}{ll} a_j \leqq a_{2j} & \text{wenn } 2j \leqq k, \\ a_j \leqq a_{2j+1} & \text{wenn } 2j+1 \leqq k. \end{array}\right\} \quad \forall j \in \{i, \ldots, k\}. \tag{8.8}$$

□

[4] Heun (2000):§2.4.3.

Ist also $a = \big[a[0], \ldots, a[n-1]\big]$ ein Subheap, so ist a selber ein Heap, siehe Abschnitt 3.2 auf S. 36. Bevor wir uns dem Heapsort selber widmen, wollen wir zunächst an die zwei grundlegenden Editieralgorithmen eines Heap erinnern, insert und extractMax, die wir in Abschnitt 3.2 kennengelernt haben. Der Algorithmus reheap(l, r) sorgt dafür, dass ein vorliegender Subheap $\big[a[l+1], \ldots, a[r]\big]$ das Element $a[l]$ davor so einordnet, dass am Ende $\big[a[l], a[l+1], \ldots, a[r]\big]$ ein Subheap ist.

```
algorithm reheap(a[], left, right) {
   input: ein Array a, das einen Subheap [a[l+1], ..., a[r]] hat
   output: (in place) ein Subheap [a[l], a[l+1], ..., a[r]]

   i ← left; // start i from left end
   while (2i+1 ≦ right) { // while there is at least a left child
      l ← 2i+1; r ← 2(i+1); // index of left and right child
      if (r < right) { // does right child exist at all?
         if (a[l] > a[r]) { // which child is greater?
            max ← l;
         } else {
            max ← r;
         }
      } else {
         max ← l;
      }
      if (a[i] < a[max]) { // heap property violated?
         a[i] ↔ a[max];
         i ← max;
      } else {
         i ← r+1; // exit loop
      }
   }
}
```

Durch die Heapeigenschaft (3.3) ist der linke Kindknoten des Knotens $a[i]$ eines Heaps das Element $a[2i+1]$, und der rechte Kindknoten ist $a[2(i+1)]$. Abbildung 8.2 zeigt die Arbeitsweise des *reheap*s. Der Algorithmus `reheap` benötigt jeweils zwei Vergleiche auf jeder Baumebene,

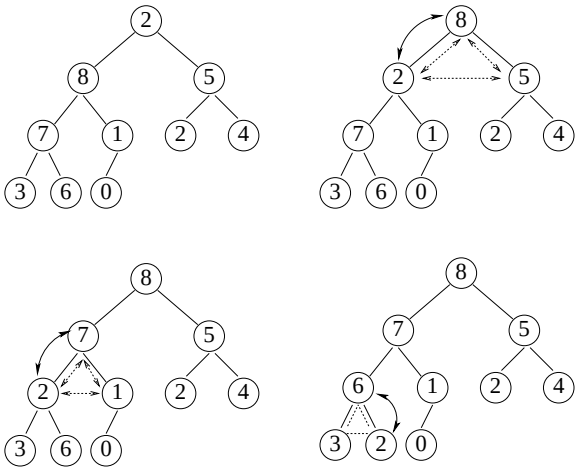


Abbildung 8.2: Die Subroutine `reheap`

also maximal $2 \log n$ Vergleiche fürden gesamten Baum. Daher beträgt die Laufzeitkomplexität von `reheap`

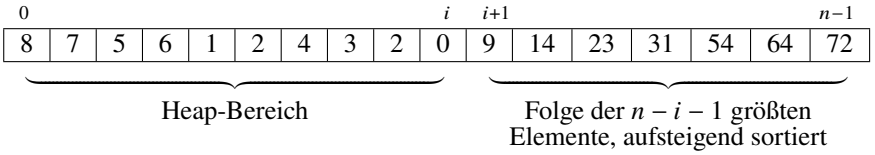
$$T_{\text{reheap}}(n) = O(\log n). \tag{8.9}$$

Mit diesem Algorithmus können wir nun den Algorithmus *Heapsort* für ein Array a mit n sortierbaren Elementen einführen. Das Array a muss dabei anfangs kein Heap sein.

```
algorithm heapsort(a[]) {
   input: ein Array mit sortierbaren Elementen
   output: (in place) das aufsteigend sortierte Array

   for (i = ⌊(n − 1)/2⌋ down to 0) { // phase 1: Building the heap
      reheap(i, n − 1);
   }
   for (i = n − 1 down to 1) { // phase 2: Selecting the maximum
      a[0] ↔ a[i];
      reheap(0, i − 1);
   }
}
```

Wie funktioniert die Sortierung? In Phase 1 (AUfbau des Heaps) wird der Subheap $\big[a[\lfloor (n-1)/2 \rfloor + 1], \ldots, a[n-1]$ zu einem Subheap $\big[a[\lfloor (n-1)/2 \rfloor], \ldots, a[n-1]$ erweitert. Die Schleife wird dabei $(n/2)$-mal durchlaufen, jede Iteration mit einer Laufzeit von $O(\log n)$. In Phase 2 wird ein sortiertes Array im hinteren Teil des Array aufgebaut. Dazu wird das jeweilige Maximum $a[0]$ mit $a[i]$ ausgetauscht, und somit wird der Heap-Bereich um einen Knoten zu $a[0], \ldots, a[i-1]$ verkleinert. Da $\big[a[1], \ldots, a_{i-1}$ jeweils ein Subheapist, wird durch reheap mit a_0 das Array makes $\big[a[0], \ldots, a[i-1]\big]$ wieder zu einem Subheap:



In Phase 2 wird die Schleife $(n-1)$-mal durchlaufen. Insgesamt hat also der Heapsort eine Laufzeitkomplexität von

$$\boxed{T_{hs}(n) = O(n \log n)} \tag{8.10}$$

im ungünstigsten Fall.

8.4 Vergleich von Sortieralgorithmen

Zusammengefasst ergeben sich die in Tabelle 8.1 aufgeführten Komplexitäten der verschiedenen in diesem Kapitel behandelten Sortieralgorithmen.

Komplexität	Bubble/Selection/Insertion	Quicksort	merge sort	heap sort	pigeonhole
worst case	$O(n^2)$	$O(n^2)$	$O(n \ln n)$	$O(n \ln n)$	$O(n)$
average case	$O(n^2)$	$O(n \ln n)$	$O(n \ln n)$	$O(n \ln n)$	$O(n)$
Speicherplatz	$O(1)$	$O(\ln n)$	$O(n)$	$O(1)$	$O(n)$

Tabelle 8.1: Laufzeit- und Speicherplatzkomplexitäten verschiedener Sortieralgorithmen. Der Pigeonhole sort wird dabei auf ein Array natürlicher Zahlen $\leqq O(n)$ angewandt.

9

Hashing und die Suche in unsortierten Datenstrukturen

Kapitelübersicht

Ist es möglich, die Suche in unsortierten Datenstrukturen zu optimieren? In Satz 2.1 lernten wir das theoretische Resultat, dass das Auffinden eines Schlüssels in einer unsortierten Datenstruktur im ungünstigsten Fall linear ist, d.h. $O(n)$ mit der Anzahl n der Einträge in der Datenstruktur. Mit der naiven Brute-Force-Methode (brute force – engl.: „rohe Gewalt“), auch erschöpfende Suche oder *Exhaustion* genannt, erhalten wir *im Durchschnitt* eine Laufzeitkomplexität von $\Theta(n)$, also auch nur linear. Damit ist die lineare Laufzeit eine mathematisch bewiesene untere Grenze für die Suche in unsortierten Datenstrukturen und kann nicht unterschritten werden.

Allerdings gibt es eine raffinierte Hintertür, um zumindest die *durchschnittliche* auf eine konstante Laufzeitkomplexität $O(1)$ zu reduzieren. Diese Hintertür heißt *Hashing*. Die grundlegende Idee des Hashings ist es, den Schlüssel der zu speichernden Einträge zu berechnen und für die Schlüsselwerte einen begrenzten Wertebereich vorzusehen. Die Berechnung dieser „Hashwerte“ wird durch eine festgelegte Hashfunktion durchgeführt. Die Hashwerte dienen dann als Speicheradresse („Referenz“) der Einträge, die dann bei der Suche mit derselben Hashfunktion berechnet werden kann. Salopp formuliert speichert man mit dem Hashverfahren die Einträge also chaotisch, kann sich aber die Speicherstelle bei der Suche errechnen.

Ein solches Hashverfahren wird zum Beispiel in Java zur Speicherung von Objekten im Heapspeicher verwendet (man kann den Hashwert eines Objektes mit der Methode `hashCode()` ermitteln). Auch die Klassen `HashSet` und `HashMap` der Java Collections setzen es ein. Hashfunktionen spielen aber überraschenderweise auch in ganz anderen Gebieten der Informatik eine wichtige Rolle, zum Beispiel der Kryptologie: Prominente Beispiele dafür sind die Hashfunktionen MD5, und SHA-256. Ein kurze Einführung über Hashfunktionen gibt[1].

[1] Hackel et al. (2010).

9.1 Hashwerte

9.1.1 Wörter und Alphabete

Um Texte zu schreiben, also um im weiteren Sinne Information zu speichern und auszutauschen, benötigen wir Symbolzeichen mit eindeutiger Bedeutung. Diese Zeichen sind Buchstaben eines gegebenen Alphabets, sie bilden Wörter eines Textes. Diese Begriffe werden formal wie folgt definiert.

Definition 9.1 Ein *Alphabet* ist eine endliche nichtleere Menge $\Sigma = \{a_1, \ldots, a_s\}$ mit einer linear Ordnung

$$a_1 < a_2 < \cdots < a_s.$$

Die Elemente a_i heißen *Buchstaben*. □

Beispiel 9.2 (i) Ein wohlbekanntes Alphabet ist $\Sigma = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \ldots, \mathrm{Z}\}$. Es hat 26 Buchstaben.
(ii) In der Informatik ist das binäre Alphabet $\Sigma = \{0, 1\}$ geläufig. Es hat zwei Buchstaben. □

Definition 9.3 Sei $\Sigma = \{a_1, \ldots, a_s\}$ ein Alphabet.

(i) Ein *Wort* (oder auch *String*) über Σ ist eine endliche Folge von Buchstaben,

$$w = a_{i_1} a_{i_2} \ldots a_{i_n} \qquad (i_j \in \{1, \ldots, s\}.$$

Die *Länge* $|w|$ eines Worts w ist die Anzahl seiner Buchstaben und wird mit $|w|$ bezeichnet.

(ii) Das *leere Wort* ist definiert als λ und hat die Länge 0.

(iii) Die Menge aller Wörter über Σ der Länge $n \in \mathbb{N}$ wird mit Σ^n bezeichnet. Die Menge aller Wörter über Σ, inklusive des leeren Wortes λ, wird mit Σ^* bezeichnet. Es wird oft auch *Universum* genannt. □

Beachte, dass $\Sigma^n \subset \Sigma^*$ für jedes $n \in \mathbb{N}$ gilt.

Beispiel 9.4 (i) Ein Wort über dem Alphabet Σ aus Beispiel 9.2 (i) ist beispielsweise NOVEMBER. Es hat die Länge 8, d.h.

$$\mathrm{NOVEMBER} \in \Sigma^8.$$

(ii) Ein Wort über dem binären Alphabet $\Sigma = \{0, 1\}$ ist $1001001 \in \{0, 1\}^7$. □

Da Alphabete endliche Mengen sind, können ihre Buchstaben eins zu eins den natürlichen Zahlen zugeordnet werden. Hat ein Alphabet m Buchstaben, können seine Buchstaben mit den Zahlen

$$\mathbb{Z}_m = \{0, 1, \ldots, m-1\} \tag{9.1}$$

identifiziert – oder: „codiert“ – werden. Beispielsweise können wir für das 26-buchstabige Alphabet Σ aus Beispiel 9.2 (i) den Code $\langle\cdot\rangle : \Sigma \to \mathbb{Z}_{26}$ verwenden, der durch

a_i	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
$\langle a_i \rangle$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

(9.2)

gegeben ist. Dann gilt z.B. $\langle \mathrm{N} \rangle = 13$. Ein weiteres Beispiel ist das 127-Buchstabenalphabet des ASCII-Codes, in dem beispielsweise $\langle \mathrm{A} \rangle = 65$, $\langle \mathrm{N} \rangle = 78$, oder $\langle \mathrm{a} \rangle = 97$ gilt. Der Zahlenwert zu einem Buchstaben wird auch sein *Codepoint* genannt. Eine Verallgemeinerung des ASCII-Codes ist der *Unicode*, der $2^{16} = 65\,536$ Buchstaben codiert. Die 2^{16} Codepoints werden hier üblicherweise in ihrer hexadezimalen Darstellung mit vier Stellen (beachte: $2^{16} = 16^4$), also

$$\mathbb{Z}_{(2^{16})} = \{0000_{16}, 0001_{16}, 0002_{16}, \ldots, \mathrm{FFFF}_{16}\} \tag{9.3}$$

Die ersten $2^8 = 256$ Buchstaben und deren Codepoints im Hexadezimalcode ist in Abbildung 9.1 dargestellt.

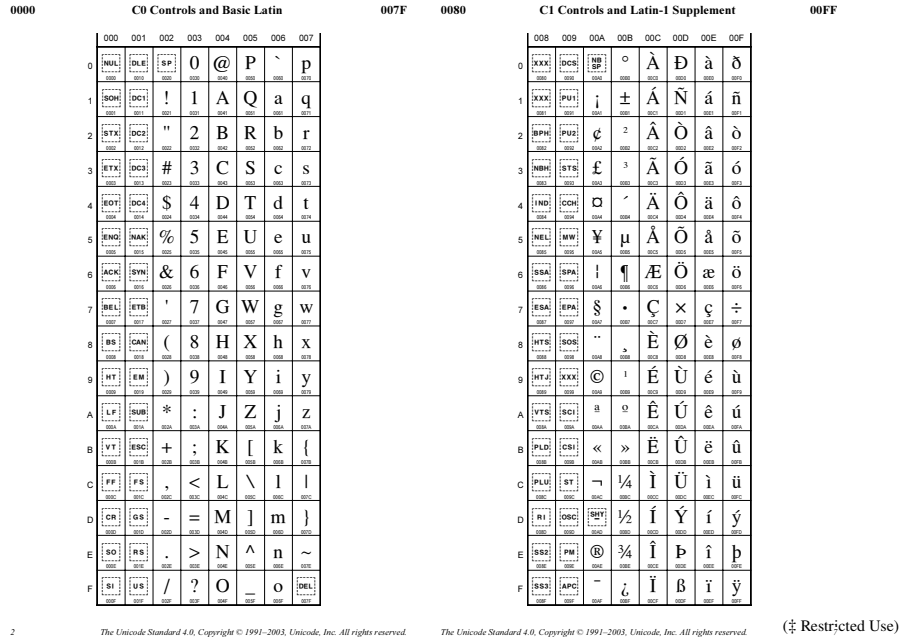


Abbildung 9.1: Die ersten 256 Codepoints des Unicode.

9.1.2 Hashfunktionen

Definition 9.5 Eine *Hashfunktion* ist eine Funktion $h : W \to H$ einer (möglicherweise unendlichen) Menge $W \subset \Sigma^*$ von Wörtern in eine endliche Menge $H \subset \mathbb{Z}$ ganzer Zahlen namens *Hashwerte* abbildet. Dabei ist jeder Hashwert $h(w)$ „leicht berechenbar", d.h. die Hashfunktion stellt einen effizienten Algorithmus dar. Die Menge aller Hashwerte heißt auch *Hashtabelle*, und die Anzahl der Hashwerte heißt ihre *Kapazität*. Manchmal werden die Hashwerte auch *Buckets* genannt, beispielsweise in der Java-API. □

Beispiel 9.6 Sei $h : \{0,1\}^* \to \{0,1\}$,

$$h(w) = w_n \oplus \ldots \oplus w_1$$

bitweise XOR-Operation eines beliebig langen Bitstrings. Beispielsweise ist $h(101) = 1 \oplus 0 \oplus 1 = 0$. Dann ist h eine (sehr einfache) Hashfunktion, and 0 zum Beispiel ist der Hashwert von 101. Die Eingabelänge ist beliebig, die Ausgabe aber stets entweder 0 oder 1, hat also die Länge 1 Bit. Da $h(1001) = 0$ gilt, haben die zwei verschiedenen Wörter $w^{(1)} = 101$ and $w^{(2)} = 1001$ denselben Hashwert. Weitere Werte ergeben sich aus folgender Wertetabelle:

w	00	01	10	11	000	001	010	011	100	101	110	111
$h(w)$	0	1	1	0	0	1	1	0	1	0	0	1

(9.4)

Der Hashwert $h(w)$ ergibt also stets 0, wenn die Anzahl der Einsen in dem Wort gerade Anzahl ist, und 1, wenn sie ungerade ist. □

Beispiel 9.7 Die letzte Ziffer der 13-stelligen ISBN[2] ist ein Hashwert, der aus den 12 ersten Stellen berechnet wird und „Prüfziffer" genannt wird. Für Bücher lauten die ersten drei Ziffern 978 oder 979 gemäß dem EAN System,

$$978 w_4 w_5 \ldots w_{12} h.$$

[2] auch ISBN-13 genannt und seit dem 1. Januar 2007 gültig; `http://www.isbn-international.org/`

Sei $\Sigma = \{0, 1, \ldots, 9\}$. Dann bilden die ersten 12 Ziffern einer ISBN also ein Wort $w \in \Sigma^{12}$, und die letzte Ziffer ist ein Hashwert $h(w)$, der durch die Hashfunktion $h : \Sigma^{12} \to \Sigma$

$$h(w_1 w_2 \ldots w_{12}) = -\sum_{i=1}^{12} g_i \cdot w_i \bmod 10 \qquad \text{mit } g_i = 2+(-1)^i = \begin{cases} 1 & \text{wenn } i \text{ ungerade,} \\ 3 & \text{wenn } i \text{ gerade.} \end{cases} \tag{9.5}$$

Zum Beispiel ist

$$h(978389821656) = -138 \bmod 10 = 2,$$

da

9	7	8	3	8	9	8	2	1	6	5	6	
1	3	1	3	1	3	1	3	1	3	1	3	
9	21	8	9	8	27	8	6	1	18	5	18	$\sum 138.$

Also ist 978-3-89821-656-2 eine gültige ISBN. □

Hashfunktionen werden in ganz unterschiedlichen Bereichen eingesetzt. So spielen sie nicht nur eine Rolle bei der Speicherung in Datenbanken, sondern sind auch wesentlich für digitale Signaturen in der Kryptologie. Auch die verlässliche Übertragung von Nachrichten über Netzwerke und in Rechnersystemen basiert auf Hashfunktionen als Prüfverfahren, denken wir dabei beispielsweise an eine Datei als Bitstring im Datenbus eines Computers oder in IP-Pakete unterteilt über das Internet übertragen. Die Übertragungswege („Kanäle“) sind meistens verrauscht und können durch Störeffekte Nachrichten in Teilen verändern. Schlimmstenfalls kann das dazu führen, dass die originale Nachricht so verändert wird, dass eine falsche, aber sinnvolle Nachricht beim Empfänger ankommt und er sich darauf verlässt. Ein gebräuchlicher Weg ist es, der Nachricht eine durch eine Hashfunktion berechnete Prüfziffer anzuhängen:

$$(w, h(w)).$$

Sender und Empfänger müssen sich dabei vor der Kommunikation über die verwendete Hashfunktion geeinigt haben. So kann der Empfänger die erhaltene Nachricht w' in die Hashfunktion einsetzten, $h(w')$, und überprüfen, ob der vom Empfänger übertragene Hashwert $h(w)$ mit dem selbst dem selbst berechneten Wert $h(w')$ übereinstimmt, ob also $h(w) = h(w')$ gilt. Falls diese beiden Werte nicht übereinstimmen, so muss ein Übertragungsfehler aufgetreten sein und die Nachricht sollte erneut angefordert werden. Im Fall von IP-Paketen oder bei der Übertragung ist die Hashfunktion ein einfacher bitweiser Paritätscheck.

9.2 Kollisionen

Ganz grundsätzlich kann eine Hashfunktion nicht umkehrbar sein, d.h. für einen gegebenen Hashwert y kann man prinzipiell nicht „das“ Wort w berechnen, für das $h(w) = y$ gilt. (Mathematisch ausgedrückt: Eine Hashfunktion h hat keine Umkehrfunktion h^{-1}.) Der Grund liegt darin, dass eine sehr große, oft sogar eine unendlich große Menge von Wörtern auf eine kleine endliche Menge von Hashwerten abgebildet wird: Es *muss* also mehrere Wörter geben, die auf einen gegebenen Hashwert abgebildet werden. Man spricht dabei von „Kollisionen“.

Definition 9.8 Haben zwei verschiedene Wörter denselben Hashwert, gilt also $h(w) = h(w')$ für zwei Wörter $w \neq w'$, so sprechen wir von einer *Kollision*. □

Es ist schwerer, zu einem bestimmten Wort w eine Kollision (w, y) zu finden als *irgendeine beliebige* Kollision (w', y'). Mathematisch beweisen lässt sich das mit dem sogenannten Geburtstagsparadoxon: Es ist ziemlich unwahrscheinlich, dass in einem Raum mit 23 Personen

jemand (y) am selben Tag wie Sie (w) Geburtstag hat. Der Geburtstag spielt hier die Rolle des Hashwertes, d.h. wenn w und w' am selben Tag Geburtstag haben, gilt $h(w) = h(w')$. Aber es ist sehr wahrscheinlich, nämlich mehr als 50%, dass sich überhaupt zwei Personen w und w' dort befinden, die an einem beliebigen Tag $h(w') = h(w')$ beide Geburtstag haben.

Im Folgenden betrachten wir eine Hashfunktion $h : \Sigma* \to H$, die einen beliebigen String auf einen von n Hashwerten abbildet. Dann bezeichnen wir mit $p(m, n)$ die Wahrscheinlichkeit einer Kollision bei einer gegebenen Menge von n Wörtern:

$$p(m,n) \;= \text{Wahrscheinlichkeit für mindestens eine Kollision bei } m \text{ Hashwerten und } n \text{ Wörtern.} \tag{9.6}$$

Theorem 9.9 *Unter der „idealen" Voraussetzung, dass eine gegebene Hashfunktion eine Menge von n Wörtern gleichverteilt auf m Hashwerte abbildet, ist die Wahrscheinlichkeit für mindestens eine Kollision durch*

$$\boxed{p(m,n) \;=\; 1 \;-\; \frac{m(m-1)(m-2)\cdots(m-n+1)}{m^n}} \tag{9.7}$$

gegeben.

Beweis. Der Beweis des Theorems ist für das Verständnis des weiteren Stoffs nicht notwendig und kann hier ignoriert werden. Er benötigt Grundkenntnisse in Wahrscheinlichkeitstheorie und ist daher nicht prüfungsrelevant. Für Interessierte sei aber auf den Anhang A.3 auf Seite 136 verwiesen. Q.E.D.

Beispiel 9.10 *(Kollisionen der XOR-Hashfunktion)* Betrachten wir die Hashfunktion aus Beispiel 9.6 – mit der Kapazität $m = 2$ – und schränken ihren Definitionsbereich auf Teilmengen der Wörter der Länge 2 ein, und zwar systematisch der n-elementigen Teilmengen für $n = 1, 2, 3$. Die folgende Tabelle gibt dabei für diese drei Werte von n der Reihe nach in den entsprechenden Tabellenpositionen angeordnet die Teilmengen, die Hashwerte ihrer Wörter und die Angabe, ob sie Kollisionen enthält, an; die letzte Zeile gibt die entsprechend die Werte der jeweiligen Kollisionswahrscheinlichkeit nach Formel (9.7) wieder:

n	1		2			3	
Teilmenge	{00}	{01}	{00, 01}	{00, 10}	{00, 11}	{00, 01, 10}	{00, 01, 11}
	{10}	{11}	{01, 10}	{01, 11}	{10, 11}	{00, 10, 11}	{01, 10, 11}
Hashwerte	0	1	(0, 1)	(0, 1)	(0, 0)	(0, 1, 1)	(0, 1, 0)
	1	0	(1, 1)	(1, 0)	(1, 0)	(0, 1, 0)	(1, 1, 0)
Kollisionen	nein	nein	nein	nein	ja	ja	ja
	nein	nein	ja	nein	nein	ja	ja
$p(2, n)$	$1 - \frac{2}{2} = 0$		$1 - \frac{2\cdot 1}{2^2} = 1 - \frac{1}{2} = \frac{1}{2}$			$1 - \frac{2\cdot 1\cdot 0}{2^3} = 1 - 0 = 1$	

Daraus erkennen wir, dass $p(2, 1) = 0$ ist, $p(2, 2) = \frac{1}{2}$ und $p(2, 3) = 1$. Mit anderen Worten kann es keine Kollision bei nur einem Wort geben ($n = 1$), mit der Wahrscheinlichkeit von 50 % bei zwei Wörtern, und bei drei Wörtern ($n = 3$) schließlich *muss* es Kollision auftreten. □

Beispiel 9.11 *(Geburtstagsparadoxon)* Wie wahrscheinlich ist es, dass von n Personen einer Gruppe (unter Vernachlässigung des 29. Februars) mindestens zwei am gleichen Tag Geburtstag haben? In der Tat kann man diese Fragestellung so umformulieren, dass es sich um ein

Hashwertproblem handelt. Denn die Anzahl der Personen ist die Anzahl n der vorhandenen Wörter, während die Anzahl der Tage eines Jahres die Kapazität m der Hashwerte darstellt. Die Hashfunktion ist die Abbildung

$$h: \text{ Person} \mapsto \text{Geburtstag},$$

die man zwar nicht berechnen kann, die aber als Mapping-Tabelle implementierbar ist. Haben zwei Personen (d.h. „unterschiedliche Wörter“) am gleichen Tag Geburtstag (d.h. „den gleichen Hashwert“), so handelt es sich also um eine Kollision und unsere Formel (9.7) ist direkt anwendbar. Bei drei Personen ist die Wahrscheinlichkeit also beispielsweise

$$p(365,3) = 1 - \frac{365 \cdot 364 \cdot 363}{365^3} = 1 - \frac{365}{365} \cdot \frac{364}{365} \cdot \frac{363}{365} = 0{,}0082 = 0{,}82\%. \tag{9.8}$$

Programmiert man die Kollisionswahrscheinlichkeit als Funktion $p(m,n)$ mit zwei natürlichen Zahlen als Eingabeparameter, so kann man sich eine Wertetabelle ausgeben lassen (die wir für größere Werte von m und n besser nicht mit Papier und Bleistift ausrechen). Ein Ausschnitt dieser Wertetabelle ist in Tabelle 9.1 angegeben. Sie zeigt, dass schon bei 23 Personen die

n	$p(365, n)$
22	0,476
23	0,507
50	0,970

Tabelle 9.1: Die Kollisionswahrscheinlichkeiten $p(365,n)$ für eine Kapazität von $m = 365$ Hashwerten

Wahrscheinlichkeit, dass zwei davon am selben Tag Geburstag haben, größer als 50 % sind. Da bei einem Fußballspiel inklusive Schiedsrichter 23 Personen auf dem Platz sind, haben statistisch in jedem zweiten Spiel zwei den gleichen Geburtstag. Bei einer Gruppe von 50 Personen ist die Kollisionswahrscheinlichkeit sogar etwa 97 %, d.h. gleiche Geburstage sind nahezu unvermeidlich! Diese Phänomen heißt „Geburtstagsparadoxon“. □

Nach Gleichung (9.7) hängt die Größe einer Kollisionswahrscheinlichkeit von der Kapazität m der Hashwerte und der Anzahl n ab, allerdings nicht durch eine geschlossene Formel darstellbar. Ebenso wenig kann man die Formel einfach nach n umstellen, wenn man die Werte von m und $p(m,n)$ kennt. Der ungarisch-amerikanische Mathematiker Paul Halmos (1916–2006) berechnete allerdings die Abschätzung

$$n \approx 1.18\sqrt{m}, \tag{9.9}$$

so dass $p(m,n) \approx \frac{1}{2}$, siehe[3]. Exemplarisch sind einige Werte in Tabelle 9.2 angegeben, darunter die in der IT-Sicherheit wichtigen Fälle 2^{128} und 2^{256}. Das bedeutet in asymptotische Nota-

m	$1.18\sqrt{m}$
365	22,49
1 000 000	1 177,41
$2^{128} \approx 3 \cdot 10^{38}$	$1{,}18 \cdot 2^{64} \approx 2 \cdot 10^{19}$
$2^{256} \approx 10^{77}$	$1{,}18 \cdot 2^{128} \approx 4 \cdot 10^{38}$

Tabelle 9.2: Halmos-Schätzwerte für n, so dass für ein gegebenes m die Wahrscheinlicheit ungefähr 50 % ist.

tion, dass, wenn bei gegebenem m die Kollisionswahrscheinlichkeit $p(m,n) \approx \frac{1}{2}$ beträgt, die Beziehung

$$n = \Theta(\sqrt{m}) \tag{9.10}$$

[3] Havil (2009):S. 31ff.

gilt. In der Praxis kommen Kollisionen daher weitaus seltener vor als es zunächst scheint. Denn Hashwerte werden nur von tatsächlich verwendeten Wörtern berechnet, und es gibt sehr viel weniger tatsächlich verwendete Wörter als mögliche Hashwerte, jedenfalls wenn die Anzahl m der möglichen Hashwerte groß genug ist, z.B. $m = 2^{256}$.

9.3 Kryptologische Hashfunktionen

Definition 9.12 Eine Hashfunktion heißt *kryptologisch* (auch *schwach kollisionsresistent*), wenn es praktisch undurchführbar ist, zu einem gegebenen Wort ein anderes Wort mit demselben Hashwert zu finden. Die Anzahl der möglichen Hashwerte einer kryptologischen Hashfunktion heißt auch *Blocklänge* und wird in Bit angegeben. □

Bemerkung 9.13 In der Definition taucht der Begriff „praktisch undurchführbar“ auf. Er ist etwas vage und bedeutet, dass die Laufzeit zum Auffinden einer Kollision nach dem jeweils aktuellen Stand der Technik sehr lange dauert. Da eine Kollision nur durch Brute-Force gefunden werden kann, ist der wesentliche Faktor die Anzahl der möglichen Hashwerte. Bei m möglichen Hashwerten muss man auch etwa m Wörter ausprobieren, um zu einem gegebenen Wort eine Kollision zu finden. □

Beispiel 9.14 Die bitweise XOR-Funktion aus Beispiel 9.6 ist nicht kryptologisch, denn sie hat nur $m = 2$ Hashwerte, also eine Blocklänge von 1 Bit: Für ein gegebenes Wort brauchen wir durchschnittlich nur zwei andere Wörter auszuprobieren, um eine Kollision zu finden, vgl. Beispiel 9.10. □

Eine kryptologische Hashfunktion muss eine sehr große Anzahl m an Hashwerten haben. Die ersten solcher Hashfunktionen waren MD4 und MD5 des Kryptologen Ron Rivest, die $m = 2^{128} \approx 3,4 \cdot 10^{38}$ Hashwerte, also eine Blocklänge 128 Bit haben. Sie gelten nach heutigem Stand der Technik als nicht mehr sicher, eine kryptologische Hashfunktion sollte heute eine Blocklänge von mindestens 160 Bit haben. Eine kleine Übersicht über gängige kryptologische Hashfunktionen ist in Tabelle 9.3 gegeben. Alle diese Hashfunktionen geben die Hashwerte

Hashfunktion	Blocklänge	Hexadezimalstellen	Veröffentlichung
MD4	128 Bit	32	1990
MD5	128 Bit	32	1991
SHA-1	160 Bit	40	1993
RIPEMD-160	160 Bit	40	1996
SHA-256	256 Bit	64	2004
SHA3-256	256 Bit	64	2011

Tabelle 9.3: Gebräuchliche kryptologische Hashfunktionen und ihre Blocklängen

als Hexadezimalstring (mit führenden Nullen) aus, bei SHA-256 mit einer Blocklänge von 256 Bit also Hexadezimalstrings der Länge $256/4 = 64$. In Java können Implementierungen der Hashfunktionen über die Klasse `MessageDigest` im Paket `java.security` verwendet werden, für MD5 und SHA-256 beispielsweise als statische Methoden aufrufbar gemacht wie folgt:

```
import java.security.MessageDigest;

public class Hashfunktionen {
   /** Gibt den mit MD5 gehashten Wert des spezifizierten Texts zurück.
    *  @param text der zu hashende Text
```

```
     *  @return der mit MD5 gehashte Wert
     */
    public static String md5(String text) {
       String hash = "";
       try {
          MessageDigest md = MessageDigest.getInstance("MD5");
          md.update(text.getBytes("UTF-8"));
          // Hashwert mit führenden Nullen:
          hash = String.format("%032x", new java.math.BigInteger(1,md.digest()));
       } catch (java.security.NoSuchAlgorithmException nsae) {
          System.err.println(nsae.getMessage());
       } catch (java.io.UnsupportedEncodingException uee) {
          System.err.println(uee.getMessage());
       }
       return hash;
    }

    /** Gibt den mit SHA-256 gehashten Wert des spezifizierten Texts zurück.
     *  SHA-256 ist eine nach RFC 6234 (https://tools.ietf.org/html/rfc6234)
     *  standardisierte kryptographische Hashfunktion und gilt als sicher.
     *  @param text der zu hashende Text
     *  @return der mit SHA-256 gehashte Wert
     */
    public static String sha256(String text) {
       String hash = "";
       try {
          MessageDigest md = MessageDigest.getInstance("SHA-256");
          md.update(text.getBytes("UTF-8"));
          // Hashwert mit führenden Nullen:
          hash = String.format("%064x", new java.math.BigInteger(1,md.digest()));
       } catch (java.security.NoSuchAlgorithmException nsae) {
          System.err.println(nsae.getMessage());
       } catch (java.io.UnsupportedEncodingException uee) {
          System.err.println(uee.getMessage());
       }
       return hash;
    }

    public static void main(String... args) {
       String[] wörter = {"Hagen", "Hagel"};

       System.out.println("MD5");
       for (String wort : wörter) {
          System.out.print(" " + wort + ": ");
          System.out.println(md5(wort));
       }

       System.out.println("SHA-256");
       for (String wort : wörter) {
          System.out.print(" " + wort + ": ");
```

```
            System.out.println(sha256(wort));
        }
    }
}
```

Die Ausgabe ergibt jeweils die folgenden Hashwerte für die Wörter „Hagen“ und „Hagel“:

Hashfunktion	Wort	Hashwert
MD5	Hagen	`dceb6a926bc4f7fd56faba2edf005c87`
	Hagel	`cabde3eb8cf20441bc7b880d4ec1c23f`
SHA-256	Hagen	`0a654b3032812f103184dcd0fa0e60d5db9655bcd5ce8058725230312c30724c`
	Hagel	`5ef205a716ce3279d7ca9ab3d7acbcac95ebdbb0ec6d9a1e498a24a0682711d9`

Einerseits erkennen wir die unterschiedlichen, aber je Hashfunktion stets gleichen Blocklängen der Hashwerte, andererseits die Wirkung nur eines einzigen geänderten Buchstabens, der den Hashwert drastisch verändert. SHA-256 spielt eine wesentliche Rolle beim Mining des Kryptogeldes Bitcoin.[4]

9.4 Speichern und Suchen mit Hashing

Die grundsätzliche Idee des *Hashings* ist, eine Menge U von Wörtern in eine Hashtabelle (in Java eine `HashMap`) gespeichert wird. Aus Sicht der Wörter ist die Datenstruktur unsortiert, insbesondere können auch unsortierbare Wörter gespeichert werden. „Unter der Haube“ allerdings ist eine Hashtabelle über die Hashwerte als sortierbare natürliche Zahlen sehr wohl sortiert. Denn durch eine zugrunde gelegte Hashfunktion wird der Hashwert eines zu speichernden Wortes als Speicheradresse berechnet, von der ein Zeiger auf das gespeicherte Wort verweist. Genauer gesagt wird also eine gegebene Hashfunktion

$$h : U \to \{0, \ldots, m-1\}, \quad w \mapsto h(w)$$

verwendet, um einem zu speicherndem Wort w die Speicheradresse $h(w)$ der Hashtabelle zuzuordnen. Das Hashing-Prinzip wird in Abbildung 9.2 skizziert. Bildlich gesprochen verteilt

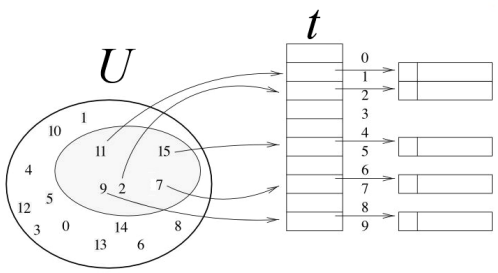


Abbildung 9.2: Das Prinzip des Hashings. Hier sind die zu speichernden Einträge aus der Menge $U = \mathbb{Z}_{16} = \{0, 1, \ldots, 15\}$, die Hashtabelle t mit der Kapazität $m = 10$ und der Hashfunktion $h(w) = w \bmod 10$.

die Hashfunktion die n Wörter aus U also in m nummerierte Container, von denen jeder ein Eintrag in der Hashtabelle darstellt. In einer Hashtabelle heißen die Hashwerte *Schlüssel*, da sie auf die Speicherplätze verweisen. Außerem spricht man bei Hashtabellen meist nicht von Wörtern oder Strings, sondern von *Einträgen*, da es sich ja um Datenstrukturen handelt.

[4] `http://www.spektrum.de/artikel/1547029`

Beispiel 9.15 Konstruieren wir zur Klärung des Hashingprinzips eine einfache Hashtabelle. Es seien die Menge der zu speichernden Wörter

$$U = \{22, 29, 47, 59, 67, 72\}.$$

und $h : U \to \mathbb{Z}_{11}$ die zugrunde liegende Hashfunktion

$$h(w) = w \bmod 11.$$

Daraus ergibt sich die folgende Belegung der Hashtabelle:

$h(w)$	w
0	22
1	67
2	
3	47
4	59
5	
6	72
7	29
8	
9	
10	

Wollen wir nun einen Eintrag in dieser Datenstruktur suchen, beispielsweise $w = 47$, so berechnen wir seinen Hashwert, also z.B. $h(47) = 47 \bmod 11 = 3$, und haben damit die Speicheradresse des Eintrags ermittelt. Auch die erfolglose Suche wird damit ermöglicht: Wollen wir einen Eintrag suchen, der nicht in der Hashtabelle gespeichert ist, beispielsweise $w = 35$, so berechnen wir die Speicheradresse $h(35) = 2$ und stellen fest, dass in Container 2 kein Eintrag gespeichert ist, also die 35 nicht in der Hashtabelle steht. □

Ähnlich funktioniert der Heapspeicher der Virtual Machine in Java, in dem alle erzeugten Objekte gespeichert werden. Der Schlüssel errechnet sich dabei über eine interne Hashfunktion aus den Attributwerten des Objekts und kann über die Methode `hashCode()` abgerufen werden.

Beispiel 9.16 Modifizieren wir die Hashtabelle aus Beispiel 9.15 und speichern die Wörter

$$U = \{22, 29, 33, 47, 59, 67, 72, 84, 91\}.$$

mit derselben Hashfunktion $h(w) = w \bmod 11$. Daraus ergibt sich die folgende Belegung der Hashtabelle:

$h(w)$	w
0	22, 33
1	67
2	
3	47, 91
4	59
5	
6	72
7	29, 84
8	
9	
10	

Wir erkennen sofort, dass Kollisionen zu einer Mehrfachbelegung einer Speicheradresse führen. Nach Formel (9.7) beträgt die Kollisionswahrscheinlichkeit für unsere Hashtabelle mit der Kapazität $m = 11$ und $n = 9$ genau

$$p(11,9) = 1 - \frac{11 \cdot 10 \cdot \ldots \cdot 3}{11^9} = 0{,}992. \tag{9.11}$$

Das gibt zu über 99 %, also fast sicher, eine Kollision! □

Bei Hashtabellen spricht man bei Kollisionen auch von *Überläufen*, und ein Eintrag w', dessen Schlüssel $h(w')$ schon besetzt ist, heißt *Überläufer*, vgl.[5]. Wie können wir mit diesem Problem umgehen?

9.4.1 Strategien der Kollisionsauflösung

Einerseits sind Kollisionen nach Konstruktion von Hashfunktionen unvermeidlich, zumal die Kapazität einer Hashtabelle nicht die enormen Größenordnungen der Kapazitäten kryptographischer Hashfunktionen annehmen kann. Andererseits müssen wir von einer Hashtabelle als Datenstruktur eine verlässliche Speicherung und Abrufbarkeit verlangen, systembedingte Datenverluste sind absolut zu vermeiden. Zur Lösung dieses Problems gibt es mehrere Strategien, die wir in diesem Abschnitt kennenlernen werden.

Hashing mit Verkettung

Beim Hashing mit Verkettung verweisen die Speicheradressen der Schlüssel nicht auf einen Eintrag, sondern auf eine verkettete Liste. Diese Liste ist anfangs leer und wird im Falle einer Kollision einfach um den neuen Eintrag erweitert. Bei den Kollisionen in Beispiel 9.16 ergibt sich dann die Hashtabelle in Abbildung 9.3.

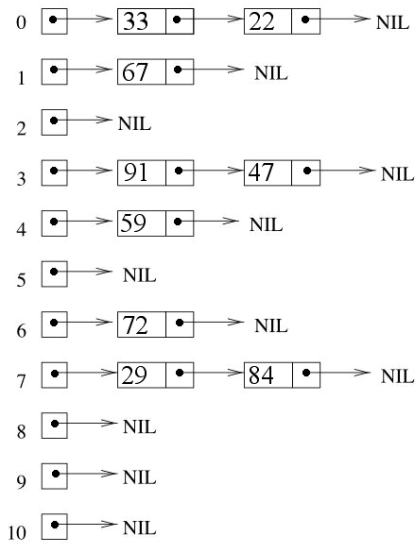


Abbildung 9.3: Hashing mit Verkettung.

[5] Ottmann und Widmayer (2012):§4.2.

Hashing mit offener Adressierung

Eine zweite Strategie zur Auflösung von Kollisionen einer Hashtabelle ist das *offene Hashing* oder *Hashing mit offener Adressierung*. Die Grundidee ist, beim Speichern eines Eintrags eine festgelegte Liste von verschiedenen Hashfunktionen

$$h_0, h_1, \ldots, h_{m-1} \tag{9.12}$$

der Reihe nach solange zu verwenden, bis eine Speicheradresse frei ist. Hierbei ist m wie üblich die Kapazität der Hashtabelle. Das Verfahren hat gegenüber dem Hashingmit Verkettung den Vorteil, dass wir keine verkettete Liste als Hilfsstruktur brauchen. Ein Nachteil jedoch ist, dass nach spätestens m Einträgen die Hashtabelle voll ist und keinen weiteren Eintrag mehr speichern kann. Auch kann es bei häufigem Speichern und Löschen der Einträge zu starken Laufzeitverlusten kommen, wie im Folgenden näher erläutert wird.

Betrachten wir ähnlich wie in Beeispiel 9.16 die zu speichernde Menge $U = \{39, 43, 61, 67, 75\}$, die Hashtabelle mit der Schlüsselmenge $\{0, 1, \ldots, 10\}$, d.h. der Kapazität 11, und und der Liste von Hashfunktionen

$$h_i(w) = w + i \bmod 11 \qquad \text{mit } i = 0, 1, \ldots, m-1, \tag{9.13}$$

also $h_0(w) = w \bmod 11$, $h_1(w) = w+1 \bmod 11$, $h_2(w) = w+2 \bmod 11$, usw. (Das offene Hashing mit einer solchen Liste von Hashfunktionen heißt auch „lineares Sondieren“.) Eine Kollision entsteht dann für $v = 39$ und $w = 61$, da $h_0(v) = h_0(w) = 6$ gilt. Ist 61 der später zu speichernde

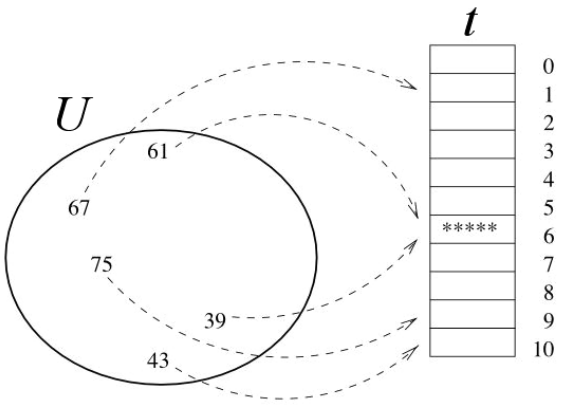


Abbildung 9.4: Hashing mit offener Adressierung.

Eintrag, so wird die Kollision versucht aufzulösen, indem die nächste Hashfunktion verwendet wird, also $h_1(61) = 7$. Da dieser Speicherplatz noch frei ist, wird die 61 dort gespeichert; wäre er auch besetzt gewesen, so wäre die nächste Hashfunktion mit $h_2(61) = 8$ verwendet worden, usw.

Wie kann man nun einen Eintrag suchen? Man muss wie beim Abspeichern dieselbe Liste an Hashfunktionen sequenziell anwenden und jedes Mal schauen, ob der jeweils berechnete Schlüssel auf den Eintrag verweist. Falls nicht, wird die nächste Hashfunktion verwendet, usw. Ein Nachteil ist, dass ein gelöschter Eintrag nicht zu kürzeren Suchlaufzeiten führen muss. Nehmen wir in unserem Beispiel an, die 39 wird aus dem Speicherplatz gelöscht, dann ist der Schlüssel $k = 6$ zwar frei, aber die Suche nach der 61 bleibt dennoch gleich lang. Daher ist das offene Hashing nicht gut geeignet ...

- ... für sehr dynamische Anwendungen, bei denen viele Einträge gelöscht und gespeichert werden;

- ...in Fällen, in denen zu erwarten ist, dass die Anzahl der zu speichernden Werte größer werden kann als die Anzahl der Schlüssel.

Komplexitätsanalyse

Zur Berechnung der Laufzeitkomplexitäten des Hashings mit den verschiedenen Strategien zur Kollisionsauflösung ist der *Auslastungsfaktor*[6], oder *Belegungsfaktor*[7], englisch *load factor*, der einfach das Verhältnis von belegten Plätzen zur Kapazität ist und mit α („Alpha“) bezeichnet wird:

$$\alpha = \frac{n}{m}. \tag{9.14}$$

Eine leere Hashtabelle ($n = 0$) hat den Auslastungsfaktor $\alpha = 0$, eine voll besetzte $\alpha = 1$, und eine überbesetzte $\alpha > 1$. Demnach ist eine Komplexitätsanalyse für das offene Hashing nicht

Kollisionsauflösung	**Einfügen**	**erfolgreiche Suche**	**erfolglose Suche**
mit Verkettung	0	$1 + \alpha/2$	$1 + \alpha$
offenes Hashing ($\alpha < 1$)	$\frac{1}{1-\alpha}$	$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$	$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)}\right)$

Tabelle 9.4: Durchschnittliche Anzahl der Sondierungen für verschiedene Operationen einer Hashtabelle. Nach[8].

sinnvoll, da die Anzahl der zu speichernden Einträge stets durch die Kapazität gedeckelt ist, d.h. $n < m$. Aber man kann die Anzahl der Sondierungen, d.h. die zur Einfügung, Suche oder Löschung eines Eintrags notwendige Anzahl von Vergleichen, berechnen. Sie hängen ab von dem Auslastungsfaktor α und sind in Tabelle 9.4 aufgeführt. Für das Hashing mit Verkettung allerdings können wir darüber hinaus für ein beliebiges, aber festes m das folgende Theorem herleiten.

Theorem 9.17 *Die durchschnittliche Anzahl N_{mean} der für eine erfolglose Suche in einer Hashtabelle auf Basis des Hashing mit Verkettung notwendigen Sondierungen beträgt bei einer Kapazität m und für n Einträge*

$$N_{\text{mean}}(m,n) = 1 + \frac{n}{m} = \begin{cases} \Theta(1) & \text{für } n < m, \\ \Theta(n) & \text{für } n > m. \end{cases} \tag{9.15}$$

Für eine erfolgreiche Suche braucht man im Schnitt sogar nur $1 - \frac{n}{2m}$ Sondierungen. Entsprechend ist die Laufzeitkomplexität der Suche in einer Hashtabelle auf Basis des Hashing mit Verkettung bei einer gegebenen Kapazität m und für n Einträge im ungünstigsten Fall

$$T_{\text{worst}}(n) = O(n), \tag{9.16}$$

also in Einklang mit Theorem 2.1 nur linear.

Beweis. Zur Herleitung der durchschnittlichen Anzahl der Sondierungen müssen wir die zunächst die beiden notwendigen Teiloperationen der Suche berechnen: Erst berechnet die Hashfunktion den Schlüssel, sodann muss die von ihm referenzierte Liste durchlaufen werden. Die Hashfunktion braucht für ihre Berechnung eine konstante Laufzeit $\Theta(1)$, die verkettete Liste hat im Durchschnitt n/m Einträge. Die einzigen Sondierungen, die also durchgeführt werden müssen, sind die Vergleiche mit den Einträgen der verketteten Liste, bei einer erfolglosen Suche muss dabei die gesamte Liste durchlaufen werden, also folgt die erste Gleichung in (9.15); für die erfolgreiche Suche muss im Schnitt nur die halbe Liste durchlaufen werden, also

[6] Güting (1997):S.106.
[7] Ottmann und Widmayer (2012):S.192.

$N_{\text{mean}}(n) = 1 - \frac{n}{2m}$. Die zweite Gleichung folgt durch die Fallunterscheidung, denn solange $n < m$ gilt, ist $1 + n/m < 2 = \Theta(1)$; für größere n gilt bei festem m schon nach Definition der Theta-Notation $1 + n/m = \Theta(n)$.

Für die worst-case Abschätzung müssen wir die Laufzeiten der beiden Teiloperationen im ungünstigsten Fall abschätzen: Die Berechnung des Hashwerts ist stets $\Theta(1)$, die verkettete Liste jedoch hat im ungünstigsten Fall, dass alle n Einträge denselben Schlüssel haben, die Länge n, und schlimmstensfalls muss die gesamte Liste durchsucht werden. Das ergibt die Abschätzung $T_{\text{worst}}(n) = O(1 + n)$, also Gleichung (9.16). Q.E.D.

Teil III

Algorithmen in Graphen und Netzwerken

10 Algorithmen in Graphen und Netzwerken

Kapitelübersicht

10.1 Grundlegende Begriffe

Ein *Graph* stellt eine Menge von Objekten und deren Beziehungen zueinander dar. Die Objekte werden *Knoten* (engl. *vertex* oder *node*) genannt, die Beziehungen *Kanten*indexKanten (engl. *edge*). Einige Beispiele dafür sind:

Graph	Objekte	Beziehungen
Soziales Netzwerk	Personen	A kennt B
Rechnernetz	Netzwerkkarten	A ist mit B verbunden
Turnier	Spieler/Teams	A gewinnt gegen B
Straßenkarte	Städte	Es führt eine Straße von A nach B

Eine Kante ist normalerweise *gerichtet*, wird also durch einen Pfeil $\rightarrow$ dargestellt. In einer Straßenkarte entspicht eine gerichtete Kante zum Beispiel einer Einbahnstraße. Eine ungerichtete Kante ist in ihren beíden Richtungen symmetrisch und wird durch einen Doppelpfeil $\rightleftarrows$ oder durch eine Linie — dargestellt.

Definition 10.1 Ein *Digraph*, oder auch *gerichteter Graph* (engl. *directed graph*) ist ein Paar $G = (V, E)$ von zwei Mengen V und E, für die gilt:

1. V ist eine endliche nichtleere Menge, deren Elemente die Knoten (*vertices*, Plural von *vertex*) des Digraphen sind.

2. Die Menge $E \subseteq V \times V$ der gerichteten Kanten (v, w) mit dem Startknoten v und dem Endknoten w.

Besteht ein Graph nur aus ungerichteten Kanten, so ist er ein *ungerichteter Graph*. In diesem Falle ist eine ungerichtete Kante $\{v, w\} \subseteq V$ mit ihren Endpunkten v und w darstellbar; eine ungerichtete Kante ist also eine höchstens zweielementige Teilmenge der Knotenmenge V. □

In diesem Skript wird ein Digraph häufig einfach nur kurz „Graph“ genannt. Wir folgen damit

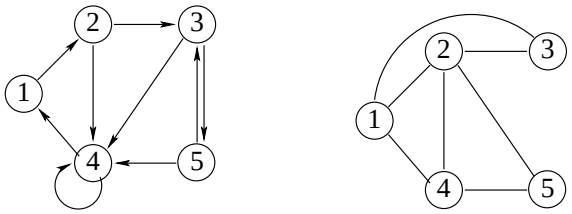


Abbildung 10.1: Gerichtete und ungerichtete Graphen.

der in der Informatik gebräuchlichen Bezeichnungskonvention von[1]; in mathematisch orientierten Lehrwerken versteht man dagegen unter dem Begriff „Graph“ umgekehrt eher einen ungerichteten Graphen vgl.[2].
Eine Kante ist mathematisch also ein Paar von Knoten. Die Kante $e = (v, w)$ stellt also die Beziehung $v \longrightarrow w$ dar. So ist zum Beipiel der erste Graph in Abbildung 10.1 gerichtet, und V und E sind durch

$$V = \{1, 2, 3, 4, 5\}, \qquad E = \{(1, 2), (2, 3), (2, 4), (3, 4), (3, 5), (4, 1), (4, 4), (5, 3), (5, 4)\}$$

gegeben, der zweite dagegen durch

$$V = \{1, 2, 3, 4, 5\}, \qquad E = \big\{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{4, 5\}\big\}$$

Beachten Sie: Die (gerichtete) Kante (v, w) ist ungleich der Kante (w, v). Eine ungerichtete Kante $\{v, w\}$ dagegen ist gleich der Kante $\{w, v\}$.

Im Folgenden geben wir einige wichtige Eigenschaften von Digraphen und weitere Definitionen an[3]. Es seien dabei stets $v, w \in V$ gegebene Knoten:

- Ein Digraph kann die Kante $e = (v, v)$ enthalten; eine solche Kante heißt *Schlinge* (engl. *self-loop*).
- Ist $(v, w) \in E$ eine Kante, so heißen die Knoten v und w *adjazent* oder *benachbart*.
- Die Anzahl Kanten eines Graphen bezeichnen wir im Folgenden stets mit $m = |E|$, die Anzahl der Knoten mit $n = |V|$.

Bemerkung 10.2 Die maximal mögliche Anzahl $m = |E|$ an Kanten, die ein Graph oder Digtaph mit n Knoten höchstens haben kann, können wir je nach Fall wie folgt abschätzen. Wir müssen dabei immer nur beachten, dass die maximal mögliche Kantenanzahl bei gegebener Knotenanzahl dann gegeben ist, wenn jeder Knoten mit jedem Knoten verbunden ist:

(a) Ein ungerichteter schlingenfreier Graph kann höchstens $\binom{n}{2}$ Paare enthalten, d.h. für die Kantenzahl m gilt stets

$$m \leqq \binom{n}{2} = \frac{n(n-1)}{2}. \tag{10.1}$$

[1] Krumke und Noltemeier (2012).
[2] Diestel (2000); Kaderali und Poguntke (1995).
[3] Krumke und Noltemeier (2012):§2.

(b) Ein ungerichteter Graph mit Schlingen kann höchstens $\binom{n}{2}$ Paare plus n Schlingen enthalten. Da $n + \binom{n}{2} = n + \frac{n(n-1)}{2} = \frac{(n+1)n}{2}$, gilt

$$m \leqq \binom{n+1}{2}. \tag{10.2}$$

(c) Ein schlingenfreier Digraph mit n Knoten kann höchstens $\binom{n}{2}$ Paare enthalten, und jedes Paar kann maximal zwei Richtungen haben, d.h. ein solcher Graph kann höchstens $2\binom{n}{2}$ Kanten besitzen, also

$$m \leqq 2 \cdot \binom{n}{2} = n(n-1) \tag{10.3}$$

(d) Ein Digraph mit Schlingen kann demnach höchstens $n(n-1) + n = n^2$ Kanten enthalten, d.h.

$$m \leqq n^2. \tag{10.4}$$

Ein Graph mit n Knoten hat also genauso wie ein Digraph mit n Knoten, höchstens $O(n^2)$ Kanten. □

10.2 Darstellung von Graphen

Wie kann ein Graph in einem Computer dargestellt werden? Im Wesentlichen sind drei Darstellungsarten gebräuchlich[4], zwei davon werden wir in diesem Abschnitt näher betrachten. Es sei im Folgenden $n = |V|$ immer die Anzahl der Knoten eines Graphen und es gelte $V = \{v_1, \ldots, v_n\}$.

10.2.1 Adjazenzmatrix

Die *Adjazenzmatrix* $A = (a_{ij})$ eines Graphen $G = (V, E)$ ist eine $(n \times n)$-Matrix, deren Einträge durch

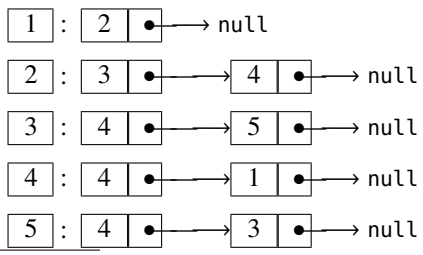
$$a_{ij} = \begin{cases} 1 & \text{wenn } (v_i, v_j) \in E, \\ 0 & \text{sonst} \end{cases} \tag{10.5}$$

definiert sind. Für den linken Graphen in Abbildung 10.1 gilt $v_i = i$, mit $i = 1, \ldots, 5$, und die Adjazenzmatrix ist die (5×5)-Matrix

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}. \tag{10.6}$$

10.2.2 Adjazenzliste

In einer *Adjazenzliste* wird jedem Knoten eine verkettete Liste aller seiner Nachbarn zugeordnet. Für den linken Graphen in Abbildung 10.1 gilt zum Beispiel



[4] Kaderali und Poguntke (1995):§4.1.8.

Beispiel 10.3 *(Darstellungen eines Digraphen)* Erläutern wir die verschiedenen Möglichkeiten zur Darstellung eines Graphen anhand eines einfachen Beispiels, des Digraphen aus Abbildung 10.2. Die Menge V der Knoten und die Menge E der Kanten für ihn lautet:

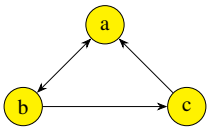


Abbildung 10.2: Digraph mit 3 Knoten.

$$V = \{a, b, c\}, \qquad E = \{(a, b), (b, a), (b, c), (c, a)\},$$

d. h. die Anzahl der Knoten $n = |V|$ und die Anzahl der Kanten $m = |E|$ ist

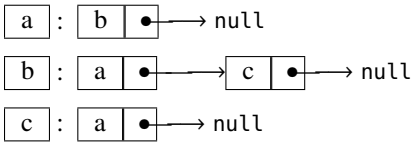
$$n = 3, \qquad m = 4.$$

Die Adjazenzmatrix A dieses Digraphen ergibt sich aus der „Start-Ziel“-Tabelle

nach / von	a	b	c
a	0	1	0
b	1	0	1
c	1	0	0

also $$A = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

Die Darstellung des Digraphen mit Adjazenzlisten dagegen sieht wie folgt aus:



In der Adjazenzmatrix des Digraphen befinden sich so viele Einsen wie es Kanten gibt, nämlich 4, und auch die Gesamtzahl der in den Adjazenzlisten gespeicherten Knoten ist gleich m. □

Beispiel 10.4 *(Darstellungen eines ungerichteten Graphen)* Betrachten wir statt des gerichteten Graphen in Abbildung 10.2 den ungerichteten Graphen aus Abbildung 10.3. Die Menge V der

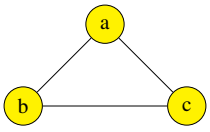


Abbildung 10.3: Ungerichteter Graph mit 3 Knoten.

Knoten und die Menge E der Kanten für ihn lautet:

$$V = \{a, b, c\}, \qquad E = \{\{a, b\}, \{a, c)\}, \{b, c\}\},$$

d. h. die Anzahl der Knoten $n = |V|$ und die Anzahl der Kanten $m = |E|$ ist gleich:

$$n = 3, \qquad m = 3.$$

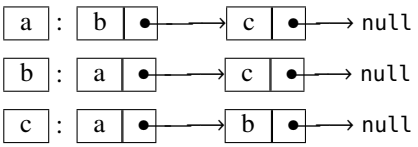
Wir sehen hier, insbesondere am Beipiel der Kanten zwischen den Knoten a und b der Graphen in Abbildung 10.2 und 10.3, noch mal den Unterschied der Kantenzählung eines ungerichteten im Gegensatz zu der eines gerichteten Graphen: In einem ungerichteten Graphen werden die Kanten

unabhängig von ihrer Richtung gezählt! Die Adjazenzmatrix A des ungerichteten Graphen ergibt sich wieder aus der „Start-Ziel“-Tabelle

nach / von	a	b	c
a	0	1	1
b	1	0	1
c	1	1	0

also $A = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$

Wir sehen hier, das die Adjazenzmatrix eines ungerichteten Graphen symmetrisch ist. Die Darstellung des Graphen dagegen sieht wie folgt aus:



In der Adjazenzmatrix des Graphen befinden sich doppelt so viele Einsen wie es Kanten gibt – nämlich 6 – und auch die Gesamtzahl der in den Adjazenzlisten gespeicherten Knoten ist gleich $2m$. □

10.2.3 Adjazenzmatrizen contra Adjazenzlisten

Welche der Darstellungsmöglichkeit ist besser, Adjazenzmatrix oder Adjazenzliste? Wie häufig bei komplexen Sachverhalten ist auch hier die Antwort ein differenziertes „Es kommt drauf an“. Denn abhängig von der hauptsächlich vorgesehenen Operation hat entweder die eine oder die andere Darstellung ihre Laufzeitvorteile. Ebenso hat die erwartete Struktur sehr großer Graphen Einfluss auf die Bevorzugnung der einen oder der anderen Darstellungsart.

Kantenprüfung. Ein Vorteil der Darstellung eines Graphen durch eine Adjazenzmatrix ist die Prüfung, ob eine Kante zwischen Knoten v_i und Knoten v_j existiert, ob also $(v_i, v_j) \in E$ gilt. Da dafür nur zu prüfen ist, ob der Eintrag a_{ij} der Matrix 1 ist, und der Zugriff auf einen Array-Eintrag unabhängig von der Knotenzahl n und der Kantenzahl m ist, ist die Laufzeitkomplexität der Kantenprüfung konstant, d.h. $O(1)$. Die Laufzeit bei einer Darstellung mit Adjazenzlisten dagegen beträgt $O(m)$, denn im ungünstigsten Fall müssen alle Adjazenzlisten bis zum Ende durchsucht werden. Adjazenzlisten eignen sich daher weniger für Graphen mit sehr vielen Kanten, d.h. $m \gg 1$, als die Adjazenzmatrix.

Speicherbedarf. Der Speicherbedarf für einen Graphen mit $n = |V|$ Knoten und $m = |E|$ Kanten beträgt für die Darstellung mit Adjazenzlisten $S_{\text{al}} = \Theta(n + m)$; da $m = O(n^2)$, wie wir oben gesehen haben, ist $S_{\text{al}} = O(n^2)$. Für die Adjazenzmatrix eines solchen Graphen benötigt man eine Speicherplatzkomplexität $S_{\text{am}} = \Theta(n^2)$ für die n^2 Matrixeinträge. Eine Darstellung mit Adjazenzmatrix sollte unter Speichergesichtspunkten einer Implementierung mit Adjazenzlisten also nur vorgezogen werden, wenn es deutlich mehr Kanten als Knoten gibt, d.h. $m \gg n$.

Kriterium	**Adjazenzmatrix**	**Adjazenzliste**
Speicherbedarf	$O(n^2)$	$O(n + m)$
Laufzeit der Kantenprüfung „$(v_i, v_j) \in E$?“	$O(1)$	$O(m)$

10.3 Traversierung von Graphen

Ein erstes algorithmisches Graphenproblem, welches wir angehen wollen und auf welchem viele Algorithmen auf Graphen basieren, ist das systematische Aufsuchen aller von einem gegebenen Startknoten aus erreichbaren Knoten eines Graphen. Mit anderen Worten: Wie kann ein Graph „traversiert“ werden? Es gibt zwei wichtige Suchstrategien, die auf jedem Graphen angewendet werden können, die Tiefensuche und die Breitensuche[5]. Wir werden sie beide in diesem Abschnitt kennenlernen.

10.3.1 Tiefensuche

Die *Tiefensuche* (engl. *depth-first search, DFS)* besucht alle Knoten, die von einem gegebenen Knoten v_s aus erreichbar sind. Wie der Name bereits andeutet, geht der Suchweg stets so „tief“ wie möglich und nötig, d.h. weg vom Startknoten. Damit er nicht Wege doppelt geht, wird ein bereits besuchter Knoten markiert. In Abbildung 10.4 ist sie an einem Beispielgraphen illustriert,

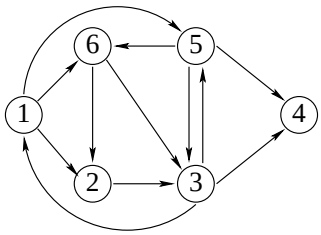
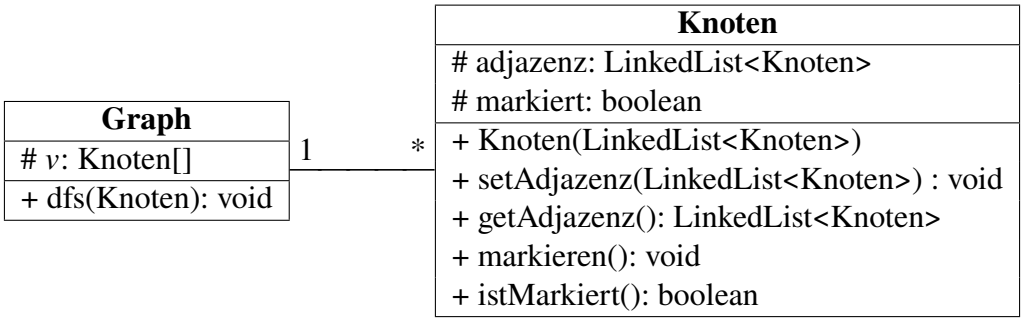


Abbildung 10.4: Ein Graph, in dem die Tiefensuche von Knoten 1 aus die Knoten in der Reihenfolge 2, 3, ..., 6 besucht.

in dem die Knoten in der Reihenfolge ihres Besuchs durch die Tiefensuche nummeriert sind. Knoten 4 ist hier von 1 aus gesehen der „tiefste“ Knoten.

Eine besonders kurze und elegante rekursive Implementierung der Tiefensuche gelingt, wenn man die Knoten als Objekte mit ihrer Adjazenzliste und einer Boole'schen Variable als Attribute entwirft, also wie im folgenden Klassendiagramm:



Die Zahl 1 und der Stern in dem Klassendiagramm heißen „Multiplizität“ und beschreiben die Situation, dass ein Objekt der Klasse Graph *mehrere* Objekte der Klasse Knoten kennen kann, jedes einzelne Knotenobjekt dagegen nur *ein* Objekt der Klasse Graph. Für die Tiefensuche relevants sind die Methoden der Klasse Knoten. Hierbei geben `getAdjazenz` und `istMarkiert` die jeweils aktuellen Zustandswerte der beiden Attribute zurück, während die Methode `markieren` einfach nur das Attribut `markiert` auf `true` setzt, das im Konstruktor des Knotens auf `false` initialisiert wird.

---

algorithm dfs(x) {
input: ein Knoten x des Netzwerks

[5]Krumke und Noltemeier (2012):§7.

```
    if (not x.istMarkiert()) {
       x.markieren();
       for (y : x.getAdjazenz()) {
          dfs(y);
       }
    }
}
```

Komplexitätsanalyse. Für einen zusammenhängenden Graphen mit n Knoten und m (gerichteten) Kanten wird die Tiefensuche `dfs` mindestens n Mal aufgerufen, denn jeder Knoten wird markiert, aber maximal m Mal, denn die `for`-Schleife kann insgesamt nicht öfter durchlaufen werden. Da die Aufruftiefe höchstens n sein kann, nämlich wenn der Graph eine lineare Kette von Knoten ist, ist der Speicherplatzbedarf der Tiefensuche höchstens linear, unabhängig von der Kantenzahl m. Zusammengefasst erhalten wir also:

$$T_{\text{dfs}}(n,m) = O(n+m), \qquad S_{\text{dfs}}(n) = O(n). \tag{10.7}$$

10.3.2 Breitensuche

Im Gegensatz zur Tiefensuche durchsucht die *Breitensuche* (engl. *breadth first search*, *BFS*) alle erreichbaren Knoten, indem zuerst die direkte Nachbarschaft, also die „Breite“, besucht wird, danach dann deren Nachbarschaft und so weiter. Abbildung 10.5 skizziert das Vorgehen dieses

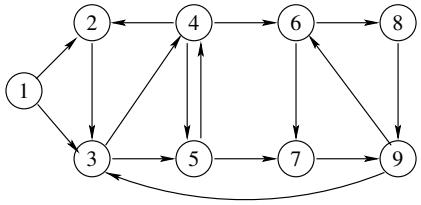
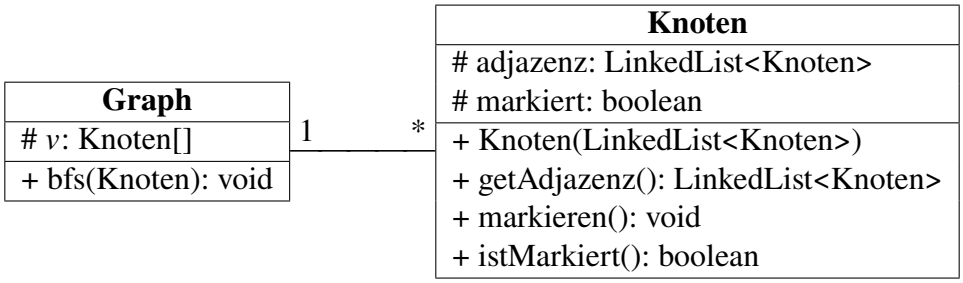


Abbildung 10.5: Ein Graph, in dem die Breitensuche in der Reihenfolge 1, 2, ..., 9 besucht.

Algorithmus bei einem Graphen, in dem die Knoten nach der Besuchsreihenfolge nummeriert sind. Ebenso wie bei der Tiefensuche muss ein bereits besuchter Knoten markiert werden, damit seine Nachbarschaft nicht mehrfach durchlaufen wird. Eine objektorientierte Implementierung ist mit dem folgenden Klassendiagramm möglich.



Die Klasse Knoten ist hierbei dieselbe wie oben bei der Tiefensuche. Der Algorithmus `bfs` der Breitensuche benötigt eine Queue als lokale Datenstruktur, in die die noch zu besuchenden Knoten zwischengespeichert werden.

```
algorithm bfs(x) {
   input: ein Knoten x des Netzwerks
   x.markieren();
```

```
    q.offer(x);          // enqueue x in a queue q
    while (not queue.isEmpty()) {
       y ← q.poll();    // dequeue next node to visit its neighbors
       for (z : y.getAdjacency()) {
          if (not z.istMarkiert()) {
             z.markieren();
             q.offer(z); //enqueue new neighbor
          }
       }
    }
 }
```

Komplexitätsanalyse. Für einen zusammenhängenden Graphen mit n Knoten und m (gerichteten) können in der Queue maximal n Knoten gleichzeitig gespeichert werden, nämlich im ungünstigsten Fall, in dem alle Knoten in der Nachbarschaft des Startknotens liegen. Daraus folgt, dass der Speicherbedarf der Breitensuche $O(n)$ ist, unabhängig von der Kantenzahl. Die `while`-Schleife wird entsprechend höchstens n Mal durchlaufen. Da zudem aber die Aufrufe der inneren `for`-Schleife insgesamt nicht höher sein kann als die Anzahl m der Kanten, gilt für die Laufzeitkomplexität $O(n+m)$. Zusammengefasst erhalten wir also:

$$T_{\text{bfs}}(n,m) = O(n+m), \qquad S_{\text{bfs}}(n) = O(n). \tag{10.8}$$

11
Wege und Kreise

Ein *Weg* oder *Pfad* (im Englischen *path* oder *walk*) in einem gegebenen Graph ist ein Kantenzug, also eine Folge von Kanten, in der der Endknoten der Vorgängerkante der Startknoten der Nachfolgerkante ist. Wir werden einen Weg stets als eine Folge seiner Knoten schreiben, also $p = (v_0, \ldots, v_k)$. Hierbei müssen stets zwei aufeinanderfolgende Knoten v_i und v_{i+1} für $0 \leqq i \leqq k-1$ auch eine Kante bilden, d.h. es muss $(v_i, v_{i+1}) \in E$ gelten. Die *Länge* $|p|$ eines Weges p ist definiert als die Anzahl seiner Kanten. Der Weg $p = (v_0, \ldots, v_k)$ hat also die Länge

$$|p| = |(v_0, \ldots, v_k)| = k. \tag{11.1}$$

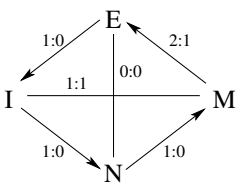
Ein Weg heißt *einfach*, wenn er keine Kante mehrfach enthält[1].

Definition 11.1 Ein geschlossener Weg $(v_0, \ldots, v_k, v_0)$, d.h. ein Weg, dessen Startknoten gleich seinem Endknoten ist, heißt *Kreis* oder *Zyklus*. Ein Graph, in dem es keine Kreise gibt, heißt *kreisfrei* oder *azyklisch*. □

Definition 11.2 Ein Kreis, in dem jeder Knoten des Graphen genau einmal aufgesucht wird, heißt *Hamiltonkreis*. Ein Kreis, in dem jede Kante des Graphen genau einmal aufgesucht wird, heißt *Eulerkreis*. □

Beispiel 11.3 In der Gruppenphase der Endrunde einer Fußballweltmeisterschaft sind 32 teilnehende Nationen in acht Gruppen mit jeweils vier Mannschaften aufgeteilt. Jede Gruppe spielt im Modus jeder gegen jeden, d.h. es gibt insgesamt $\binom{4}{2} = 6$ Spiele je Gruppe. Jedes Spiel kann entweder als eine einzelne gerichtete Kante dargestellt werden, deren Startpunkt die Gewinnermannschaft ist und deren Endpunkt das Verliererteam, oder im Falle eines Unentschiedens als eine ungerichtete Kante zwischen beiden gegnerischen Mannschaften. Zum Beispiel bestand die

Group E		
Irland	– Italien	1 : 0
Norwegen	– Mexiko	1 : 0
Italien	– Norwegen	1 : 0
Mexiko	– Irland	2 : 1
Italien	– Mexiko	1 : 1
Irland	– Norwegen	0 : 0



Mannschaft	**Tore**	**Pkt.**
Mexiko (M)	3:3	4
Irland (E)	2:2	4
Italien (I)	2:2	4
Norwegen (N)	1:1	4

Abbildung 11.1: Graph, der die Spielergebnisse der Gruppe E bei der WM 1994 darstellt.

Gruppe E bei der Weltmeisterschaft 1994 in den USA aus den Mannschaften Irland (E), Italien

[1] Krumke und Noltemeier (2012):§3.1.

(I), Mexiko (M) und Norwegen (N), der „Spielegraph“ dieser Grupppe ist in Abbildung 11.1 skizziert. Diese Gruppe ist die einzige Gruppe der Geschichte der Fußball-WM, in der alle vier Mannschaften dieselbe Punktezahl erreichten. □

11.1 Das Hamiltonkreisproblem HC

Das *Hamiltonkreisproblem (Hamiltonian cycle problem, HC)* ist die Frage, ob in einem gegebenen Graphen ein Hamiltonkreis existiert, ob es also gemäß Definition 11.2 einen Kreis gibt, der alle Knoten des Graphen genau einmal besucht. Eine solche Ja-oder-Nein-Frage, deren Antwort entweder `true` oder `false` lautet, heißt in der Informatik *Entscheidungsproblem*. Die Frage, die uns hier interessiert, lautet nun: Gibt es einen Algorithmus, der dieses Entscheidungsproblem löst? Präziser formuliert: Gibt es einen Algorithmus, der als Eingabe einen beliebigen Graph G erwartet und genau dann `true` zurückgibt, wenn er einen Hamiltonkreis enthält?

```
algorithm hatHamiltonkreis(G) {
   input: ein Graph G
   output: true dann und nur dann, wenn G einen Hamiltonkreis enthält

   ???
}
```

11.1.1 Die erschöpfende Suche

Mindestens einen Algorithmus gibt es zur Lösung des Hamiltonkreisproblems: die erschöpfende Suche („Brute Force“), die wir in Aufgabenblatt 6 in den *Grundlagen der Programmierung* im letzten Semester kennengelernt haben. Dieser Algorithmus überprüft systematisch alle möglichen Kombinationen, die theoretisch eine Lösung sein könnten, ob sie das Problem auch tatsächlich lösen. Wird so eine Lösung gefunden, bricht der Algorithmus mit der Rückgabe `true` ab; löst allerdings keine der möglichen Kombinationen das Problem, so existiert auch keine und es wird `false` zurückgegeben.

Nehmen wir an, der zu untersuchende Graph G hätte n Knoten $0, 1, \ldots, n-1$, d.h.

$$V = \{0, 1, 2, \ldots, n-1\}. \tag{11.2}$$

Ein Hamiltonkreis x, der in Knoten 1 beginnt und wieder endet, kann dann als ein $(n+1)$-Tupel (eine Art „Vektor“)

$$x = (0, x_1, x_2, \ldots, x_{n-1}, 0) \tag{11.3}$$

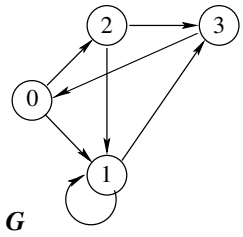
dargestellt werden, in dem jeder Knoten genau einmal vorkommt (bis auf die 0, die als Start- und Endknoten zweimal erscheint). Der in (11.3) dargestellte Kreis entspricht also einfach dem Pfad

$$0 \to x_1 \to x_2 \to \ldots \to x_{n-1} \to 0. \tag{11.4}$$

Die Wahl des Startknotens 0 ist dabei völlig willkürlich, statt ihm könnten wir grundsätzlich natürlich jeden anderen Knoten nehmen, denn in einem Kreis kann ja jeder Knoten Startpunkt sein. Tatsächlich kann man mit dem $(n+1)$-Tupel (11.3) alle Hamiltonkreise überprüfen, indem man alle Anordnungen der $n-1$ Knoten ungleich 0 systematisch durchläuft. Diese Anordnungen heißen *Permutationen*. Eine Permutation der $n-1$ Knoten $\{1, 2, \ldots, n-1\}$ wird mit einem $(n-1)$-Tupel $(x_1, x_2, \ldots, x_{n-1})$ in runden Klammern dargestellt. Der Algorithmus muss dann für jede Permutation $(x_1, x_2, \ldots, x_{n-1})$ prüfen, ob der ihr zugeordnete Kreis (11.3) auch in G existiert, d.h. ob jede der Kanten (x_j, x_{j+1}) für $j = 0, \ldots, n-1$ und die Kante (x_{n-1}, x_0) im Graph

G existieren. Sobald auf diese Weise ein Hamiltonkreis gefunden werden konnte, bricht der Algorithmus mit der Rückgabe `true` ab. Wird allerdings nach Durchlaufen aller Permutationen kein Hamiltonkreis gefunden, so kann es auch keinen geben und der Algorithmus gibt `false` zurück.

Beispiel 11.4 Sei G der links in Abbildung 11.2 dargestellte Digraph mit den Knoten $V = \{0,1,2,3\}$. Die Anzahl der Knoten ist hier also $n = 4$, und zur Auflistung aller möglichen Hamiltonkreise, die in Knoten 0 beginnen, müssen wir alle Permutationen (x_1, x_2, x_3) der drei Zahlen $\{1,2,3\}$ durchlaufen. Davon gibt es $(n-1)! = 3! = 3\cdot 2\cdot 1 = 6$. Da von diesen Möglichkeiten



G

Nr.	mögliche Hamiltonkreise	Permutationen	in G?
1	$\mathbf{0} \to 1 \to 2 \to 3 \to \mathbf{0}$	$(1,2,3)$	□
2	$\mathbf{0} \to 1 \to 3 \to 2 \to \mathbf{0}$	$(1,3,2)$	□
3	$\mathbf{0} \to 2 \to 1 \to 3 \to \mathbf{0}$	$(2,1,3)$	⊠
4	$\mathbf{0} \to 2 \to 3 \to 1 \to \mathbf{0}$	$(2,3,1)$	□
5	$\mathbf{0} \to 3 \to 1 \to 2 \to \mathbf{0}$	$(3,1,2)$	□
6	$\mathbf{0} \to 3 \to 2 \to 1 \to \mathbf{0}$	$(3,2,1)$	□

Abbildung 11.2: Die Abstraktion der möglichen Hamiltonkreise hin zu Permutationen, hier für einen Digraph G mit $n = 4$ Knoten.

nur eine, nämlich $(2,1,3)$ oder $0 \to 2 \to 1 \to 3 \to 0$, einen Hamiltonkreis wiedergibt, haben wir damit soga bewiesen, dass es einen – und nur einen – Hamiltonkreis in G gibt. □

Beispiel 11.5 Der Graph in Abbildung 11.1 hat einen Hamiltonkreis, z.B.

$$(\mathrm{I, N, M}) = \mathrm{E} \to \mathrm{I} \to \mathrm{N} \to \mathrm{M} \to \mathrm{E}, \tag{11.5}$$

und ebenso

$$(\mathrm{I, M, N}) = \mathrm{E} \to \mathrm{I} \to \mathrm{M} \to \mathrm{N} \to \mathrm{E}, \tag{11.6}$$

Durch erschöpfende Suche erkennt man, dass es keine weiteren Hamiltonkreise gibt, die in E beginnen. □

Komplexitätsbetrachtungen. Welche Lauzeitkomplexität hat der beschrieben Algorithmus `hatHamiltonkreis` auf Basis der erschöpfenden Suche? Da es bis zu $(n-1)!$ zu prüfende mögliche Hamiltonkreise in einem Graphen mit n Knoten geben kann, und für jeden einzelnen möglichen Kreis wiederum bis zu n Kanten zu prüfen sind, hat der Algorithmus eine Laufzeitkomplexität

$$T_{\mathrm{HC}}(n) \in O\big(n(n-1)!\big) \subseteq O(n!). \tag{11.7}$$

Er kann allerdings auch schon beim ersten Kreis nach n Kanten terminieren, hat also mindestens lineare Laufzeit $\Omega(n)$.

Da $2^n \in O(n!)$ gilt, hat die Lösung des Hamiltonkreisproblems mit Hilfe der erschöpfenden Suche also eine exponentielle Laufzeit. Gibt es einen effizienteren Algorithmus? Bemerkenswerterweise ist bis heute kein Algorithmus für das Hamiltonkreisproblem bekannt, der eine polynomielle Laufzeit hat.[2] Einen wesentlich besseren Algorithmus scheint es vermutlich nicht zu geben. Schränkt man allerdings die Klasse der zu untersuchenden Graphen ein, so kann es durchaus effiziente lösende Algorithmen geben. Zum Beispiel hat nach einem mathematischen Satz, dem Theorem von Dirac, ein ungerichteter Graph einen Hamiltonkreis, wenn jeder seiner Knoten mindestens $n/2$ eingehende Kanten besitzt: Ein Algorithmus, der das prüft, benötigt also

[2] Wenn Sie einen solchen Lösungsalgorithmus zu haben glauben, sollten Sie ihn veröffentlichen und sich das darauf ausgesetzte Preisgeld von 1 Mio US-Dollar einstreichen: siehe dazu Millenium Prize Problems, Clay Mathematics Institute, "P vs NP" (`http://www.claymath.org/millennium-problems/p-vs-np-problem`).

nur genau n Schritte, hat also eine lineare Laufzeit $\Theta(n)$. Diese und einige weitere *hinreichende* Bedingungen an Graphen für die Existenz eines Hamiltonkreises sind in[3] aufgeführt.

11.2 Das Eulerkreisproblem EC

Ein Problem, das dem Hamiltonkreisproblem auf dem ersten Blick sehr ähnlich sieht, ist das Eulerkreisproblem. Es hat seinen historischen Ursprung in dem Problem der sieben Königsberger Brücken, das der Mathematiker Leoard Euler 1736 löste (und dabei nebenbei die Graphentheorie erfand).

Das *Eulerkreisproblem* ist die Frage, ob in einem gegebenen Graphen ein Eulerkreis existiert, ob es also nach Definition 11.2 einen Kreis gibt, der jede Kante des Graphen genau einmal entlang läuft. Wie das Hamiltonkreisproblem ist das Eulerkreisproblem also ein Entscheidungsproblem.

Beispiel 11.6 Durch erschöpfende Suche über alle möglichen Kantenzüge der 8 gerichteten Kanten in Graph 11.1 (wenn wir eine ungerichtete Kante wie zwei gerichtete zählen, also — = ⇄) stellen wir fest, dass er einen Eulerkreis enthält:

$$\mathrm{E} \to \mathrm{I} \to \mathrm{M} \to \mathrm{E} \to \mathrm{N} \to \mathrm{M} \to \mathrm{I} \to \mathrm{N} \to \mathrm{E}, \tag{11.8}$$

oder auch:

$$\mathrm{E} \to \mathrm{N} \to \mathrm{M} \to \mathrm{E} \to \mathrm{I} \to \mathrm{M} \to \mathrm{I} \to \mathrm{N} \to \mathrm{E}. \tag{11.9}$$

Da es von E aus nur die beiden Anfangskanten gibt, aber danach keine Variationmöglichkeit mehr bleibt, sind dies die beiden einzigen Eulerkreise. □

Obwohl viele Ähnlichkeiten zwischen dem Hamiltonkreisproblem und dem Eulerkreisproblem bestehen, stellt sich scheinbar nur noch formell die Frage: Gibt es einen Algorithmus des Eulerkreisproblems, das effizienter ist als die erschöpfende Suche über alle möglichen Kantenkombinationen? Damit wäre das Eulerkreisproblem ja noch deutlich schwieriger zu lösen als das Hamiltonkreisproblem, denn die Anzahl der Kanten bei n Knoten ist $O(n^2)$, die erschöpfende Suche über alle möglichen Eulerkreise wäre also $O((n^2)!)$.

Ein in diesem Licht absolut überraschendes Ergebnis sind die folgenden Varianten des Satzes von Euler. Sie ermöglichen einen Algorithmus mit lediglich linearer Laufzeit in n.

Theorem 11.7 (Satz von Euler für gerichtete Graphen) *Ein zusammenhängender gerichteter Graph hat genau dann einen Eulerkreis, wenn jeder seiner Knoten genauso viele eingehende wie ausgehende Kanten besitzt.*

Beweis. Siehe[4] Q.E.D.

Das Eulerkreisproblem für ungerichtete Graphen ist streng genommen ein etwas anderes Problem als dasjenige für gerichtete Graphen, denn hier darf jede ungerichtete Kante nur einmal aufgesucht werden. Bemerkenswerterweise ist das hinreichende und notwendige Kriterium für die Existenz eines Eulerkreises in diesem Falle aber sogar weniger eingeschränkt als im Falle gerichteter Graphen.

Theorem 11.8 (Satz von Euler für ungerichtete Graphen) *Ein zusammenhängender ungerichteter Graph hat genau dann einen Eulerkreis, wenn jeder seiner Knoten eine gerade Anzahl eingehender Kanten besitzt.*

[3] Diestel (2000):§8.1.

[4] Krumke und Noltemeier (2012):Satz 3.26.

Beweis. Siehe[5],[6],[7] Q.E.D.

Mit diesen beiden Varianten des Satzes von Euler ist das Eulerkreisproblem in quadratischer Laufzeitkomplexität bezüglich der Knotenzahl n lösbar,

$$T_{\text{Euler}}(n) = O(n^2). \tag{11.10}$$

Denn für jeden einzelnen Knoten j müssen wir im gerichteten Fall die j-te Zeilensumme mit der j-ten Spaltensumme vergleichen, jede einzelne dieser Summen erfordert jeweils n Additionen, d.h. insgesamt haben wir $2n$ Additionen; im ungerichteten Fall müssen wir für jeden Knoten j mit n Additionen die j-te Zeilensumme berechnen und nur einmal prüfen, ob sie gerade oder ungerade ist. Insgesamt benötigen wir im gerichteten Fall also maximal $2n^2$ Additionen und n Vergleiche und im ungerichteten Fall maximal n^2 Additionen und n Prüfungen auf Geradheit. Da bereits ein einziger Knoten ausreicht, um die Nichtexistenz eines Eulerkreises zu bewiesen, kann der Algorithmus auch schon eher terminieren.

Beispiel 11.9 *(Die zwei Varianten des Hauses vom Nikolaus)* Betrachten wir das (klassische) Zeichenspiel „Haus vom Nikolaus“ für Kinder, für das ein schematisches Haus mit Dach aus 8 geraden Strecken zu zeichnen ist, ohne den Stift abzusetzen (Abbildung 11.3 links). Begleitet

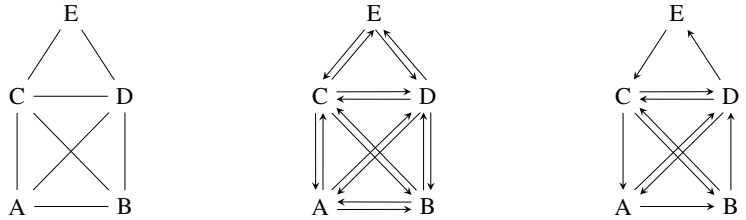


Abbildung 11.3: Das Haus vom Nikolaus: klassische Version (links), „gerichtete Versionen“ (Mitte, rechts).

wird das Zeichnen dabei durch Sprechen des achtsilbrigen Reims „Das ist das Haus vom Ni-ko-laus“, wobei jede Silbe einer Kante zugeordnet wird. Das Spiel besteht also anders ausgedrückt darin, einen Kantenzug aus 8 Kanten zu zeichnen. Das Problem hat insgesamt 44 verschiedene Lösungen, zum Beipiel den Kantenzug A – B – C – D – E – C – A – D – B.

Kann man das Haus auch mit einen *geschlossenen* Kantenzug zeichnen? Anders ausgedrückt: Hat der ungerichtete Graph in Abbildung 11.3 links einen Eulerkreis? Da jeder der Knoten A, B, C und D jeweils eine ungerade Anzahl eingehender Kanten besitzt, lautet die Antwort nach Theorem 11.8 sofort: Nein!

Betrachten wir dagegen den Graphen als gerichteten Graphen und ersetzen jede ungerichtete Kante (—) durch zwei gerichete Kanten (⇆) wie in Abbildung 11.3 Mitte, so hat jeder Knoten genauso viele eingehende wie ausgehende Kanten, d.h. nach Theorem 11.7 gibt es einen Eulerkreis. (Tatsächlich gibt es 44 verschiedene Eulerkreise, da wir jeden geschlossenen Kantenzug nach 8 Kanten einfach zurückgehen können.)

Ein „kleinerer“ gerichteter Graph des Hauses vom Nikolaus, das einen Eulerkreis besitzt, ist der in Abbildung 11.3 rechts gezeigte, seine 11 Kanten sind beispielsweise durch den Kreis

$$\text{A} \to \text{B} \to \text{C} \to \text{D} \to \text{E} \to \text{C} \to \text{A} \to \text{D} \to \text{C} \to \text{B} \to \text{D} \to \text{A}$$

jeweils genau einmal durchlaufen. □

[5]Diestel (2000):§0.8.

[6]Kaderali und Poguntke (1995):§1.3.23.

[7]Krumke und Noltemeier (2012):Satz 3.32.

12
Kürzeste Wege

Kapitelübersicht

12.1 Grundbegriffe

Wir betrachten in diesem Kapitel sogenannte „gewichtete Graphen“, in denen jeder Kante ein ihr eigenes „Gewicht“ zugeordnet wird. Je nach Kontext kann ein solches Gewicht eine Länge, eine zeitliche Dauer oder ein Kostenbetrag zur Erreichung des Endpunktes der Kante von deren Startpunkt darstellen. Betrachten wir dazu beispielsweise Abbildung 12.1. Hier können die

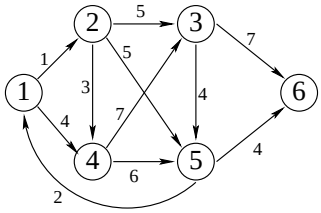


Abbildung 12.1: Ein gewichteter Graph.

Gewichte also ganz unterschiedliche Bedeutungen haben:

- Entfernungen (d.h. „Es sind 3 km von 2 nach 4.“);
- Reisedauern („Es dauert 3 Stunden von 2 nach 4.“)
- Kosten („Es kostet 3 € von 2 nach 4.“)
- Gewinne („Die Ersetzung von Maschine 2 durch Maschine 4 bringt 3 € Gewinn“)
- Kapazitäten („Die Bandbreite der Netzwerkverbindung von 2 nach 4 beträgt 3 MBit pro Sekunde.“)

Natürlich gibt es viele weitere Anwendungsfälle für gewichtete Graphen, in denen die Gewichte ganz andere Bedeutungen haben. Die formale Definition gewichteter Graphen lautet wie folgt.

Definition 12.1 Ein *gewichteter Graph* $G_\gamma = (V, E, \gamma)$ ist ein Graph $G = (V, E)$ mit der *Gewichtsmatrix*

$$\gamma = \begin{pmatrix} \gamma_{11} & \cdots & \gamma_{1n} \\ \vdots & & \vdots \\ \gamma_{n1} & \cdots & \gamma_{nn} \end{pmatrix}. \tag{12.1}$$

Hier bezeichnet der Matrixeintrag γ_{vw} das Gewicht der Kante (v, w). Existiert zwischen v und w gar keine Kante, gilt $\gamma_{v,w} = \infty$. Wir werden oft einfach G statt G_γ schreiben, wenn es ersichtlich ist, dass es sich um einen gewichteten Graphen handelt. □

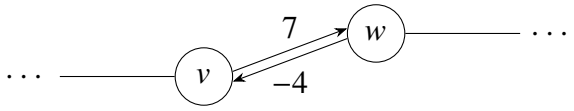
Die Gewichtsmatrix ist also eine Art Verallgemeinerung der Adjazenzmatrix, allerdings mit dem Wert ∞ statt der Null. Die ungewichteten Graphen, die wir bisher betrachtet haben, können wir als gewichtete Graphen mit dem für alle Kanten gleichen Einheitsgewicht 1 auffassen, also $\gamma_{vw} = 1$ für alle $(v, w) \in E$, und $\gamma_{v,w} = \infty$, wenn $(v, w) \notin E$.

Beispiel 12.2 Die Gewichtsmatrix des Graphen in Abbildung 12.1 lautet also:

$$\gamma = \begin{pmatrix} \infty & 1 & \infty & 4 & \infty & \infty \\ \infty & \infty & 5 & 3 & 5 & \infty \\ \infty & \infty & \infty & \infty & 4 & 7 \\ \infty & \infty & 7 & \infty & 6 & \infty \\ 2 & \infty & \infty & \infty & \infty & 4 \\ \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix}. \tag{12.2}$$

□

Bemerkung 12.3 ***(Negative Gewichte)*** Grundsätzlich können Gewichte eines Graphen auch negativ sein. Das macht zwar auf den ersten Blick keinen Sinn, wenn sie Entfernungen oder Reisedauern wie in eine Navigationssystem darstellen. Aber wenn sie z. B. Kosten darstellen, kann ja durchaus der Weg von Zustand v nach Zustand w einen Gewinn abwerfen und damit negativ werden. Ein anderes Beispiel wäre der Stromverbrauch eines E-Autos, das zu der scheinbar paradoxen Konstellation



führen kann, z.B. wenn die Strecke zwischen v und w abschüssig ist und der Weg bergauf 7 kWh verbraucht, der Weg bergab aber den Akku um 4 kWh auflädt. □

In einem gewichteten Graphen können wir nun wie folgt die Länge eines beliebigen Weges und darauf basierend die Distanz eines kürzesten Weges zwischen zwei Knoten definieren.

Definition 12.4 Sei $p = (v_0, v_1, \ldots, v_n)$ ein Weg in einem gewichteten Graphen G_γ. Die *Länge* von p ist dann definiert als die Summe der Gewichte seiner Kanten:

$$\gamma(p) = \sum_{i=1}^{n} \gamma(v_{i-1}, v_i). \tag{12.3}$$

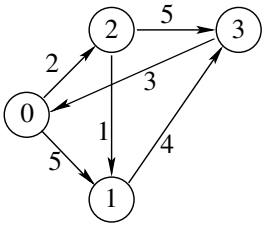
Ein *kürzester Weg* von v nach w ist ein Weg p_*, der die minimale Länge von allen Wegen von v nach w hat. Wir bezeichnen die Länge mit $l(p_*)$ oder einfach $|p_*|$. Diese minimale Länge heißt die *Distanz*

$$\delta(v, w). \tag{12.4}$$

Falls von v und w kein Weg existiert, definieren wir die Distanz als $\delta(v, w) = \infty$. □

Beachten wir, dass es in einem gegebenen Graphen grundsätzlich mehrere kürzeste Wege zwischen zwei Knoten $v, w \in G_\gamma$ geben kann, allerdings ist die Distanz $\delta(v, w)$ stets eindeutig. Das folgende Beispiel möge das illustrieren.

Beispiel 12.5 Gegeben sei der gewichtete Graph in 12.2. Was ist die Distanz von 0 nach 3? Zunächst müssen wir einen kürzesten Weg von 0 nach 3 finden. Durch Auflisten aller möglichen



$$\gamma = \begin{pmatrix} \infty & 5 & 2 & \infty \\ \infty & \infty & \infty & 4 \\ \infty & 1 & \infty & 5 \\ 3 & \infty & \infty & \infty \end{pmatrix}$$

Abbildung 12.2: Ein gewichteter Graph.

Wege und deren Länge,

$$\begin{aligned} (0,1,3) &\implies |(0,1,3)| = 5+4 = 9, \\ (0,2,3) &\implies |(0,2,3)| = 2+5 = 7, \\ (0,2,1,3) &\implies |(0,2,1,3)| = 2+1+4 = 7, \end{aligned}$$

erkennen wir, dass es *zwei* kürzeste Wege von 0 nach 3 gibt, die Distanz aber ist eindeutig und beträgt $\delta(0,3) = 7$. □

12.2 Kürzeste-Wege-Probleme

Die Bestimmung der Distanz zweier Knoten ist mathematisch gesehen ein Optimierungsproblem. Je nach Größe des Graphen kann dies ein sehr aufwändiges Verfahren sein. In der Algorithmik werden üblicherweise zwei zentrale Arten von Kürzeste-Wege-Problemen unterschieden:

- *SSSP (single-source shortest path):* Was ist ein kürzester Weg von einem fest vorgegebenen Startknoten s zu allen anderen Knoten v? Für nichtnegative Gewichte wird es durch den Dijkstra-Algorithmus gelöst, für negative durch den Bellman-Ford-Algorithmus.
- *APSP (all-pairs shortest paths)* Was sind kürzeste Wege zwischen allen Paaren von Start- und Zielknoten? Ein effizienter Algorithmus zur Lösung dieses Problems ist der Floyd-Warshall-Algorithmus.

Aber fehlt da nicht mindestens ein wichtiges Problem? Ein Navigationssystem z.B. will ja den kürzesten Weg von einem festgelegten Start zu einem festgelegten Ziel finden. Tatsächlich gibt es auch für dieses Problem eine Abkürzung, *SPSP (single-pair shortest path)*. Da aber für die Lösung auch die kürzesten Wege von dem Start zu allen anderen berechnet werden muss, ist es in Wirklichkeit eine spezielle Variante des SSSP!

Was ist bei Graphen mit negativen Gewichten zu beachten? Einige Algorithmen können

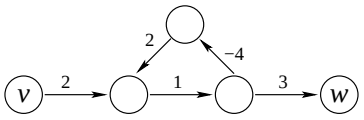


Abbildung 12.3: Gewichteter Graph mit negativem Gewicht.

ein Kürzeste-Wege-Problem mit negativen Gewichten lösen, wenn es denn eine Lösung gibt,

manche jedoch nicht. Betrachten wir dazu den Graphen in Abbildung 12.3. Auf dem Weg von v nach w können wir durch mehrfaches Durchlaufen des Kreises die Distanz *beliebig verringern*. Mit anderen Worten gilt in diesem Fall $\delta(v, w) = -\infty$. Im Allgemeinen kann also ein Kürzeste-Wege-Problem in einem Graphen mit negativen Kreisen nicht lösbar.

Theorem 12.6 *Ein Kürzeste-Wege-Problem in einem Graphen (auch mit negativen Gewichten) ist genau dann lösbar, wenn der Graph keine negativen Kreise hat.*

12.3 Das Relaxationsprinzip

Kann man die Kürzesten-Wege-Probleme nur lösen, indem man per Brute-Force alle möglichen Kombinationen der Kanten durchläuft, oder gibt es effiziente Algorithmen? Die gute Antwort: Je nach Problem und Grapheigenschaften gibt es effiziente Lösungen! Je nach Problem sind verschiedene effiziente Lösungsalgorithmen bekannt:

- SSSPFür nichtnegative Gewichte wird es durch den Dijkstra-Algorithmus gelöst, für negative unter bestimmten Bedingungen durch den Bellman-Ford-Algorithmus.
- *APSP (all-pairs shortest paths)* Ein effizienter Algorithmus zur Lösung dieses Problems ist der Floyd-Warshall-Algorithmus.

Wir werden den Floyd Warshal und den Disjkstra-Algorithums im Folgenden näher untersuchen.

Die folgende wichtige Eigenschaft liegt dem Relaxationsprinzip zugrunde, die wiederum der Kern aller hier vorgestellten Lösungsalgorithmen von Kürzeste-Wege-Problemen darstellt. Sie ist auf den ersten Blick eine eigentlich triviale Tatsache, aber ihre Bedeutung ist enorm.

Theorem 12.7 (Dreiecksungleichung) *Sei $G_\gamma = (V, E, \gamma)$ ein gewichteter Graph ohne negative Kreise. Sei ferner $\delta(v, w)$ die Distanz zwischen v und w für alle v, $w \in V$. Dann gilt für beliebige drei Knoten u, v, $w \in V$ (die nicht ungleich sein müssen) die Ungleichung*

$$\delta(v, w) \leqq \delta(v, u) + \delta(u, w). \tag{12.5}$$

Das Gleichheitsszeichen gilt genau dann, wenn u auf einem kürzesten Weg von v nach w liegt.

Beweis. Ein kürzester Weg von v nach w kann nicht länger sein, als wenn er über u führt. Entweder liegt u auf einem kürzesten Weg, dann gilt das Gleichheitszeichen. Oder u liegt nicht

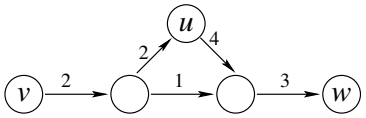


Abbildung 12.4: Dreiecksungleichung.

auf einem kürzesten Weg, dann muss es einen Weg ohne u geben, der echt kürzer ist, und die Ungleichung ist strikt. Q.E.D.

Beachte, dass diese Ungleichung sogar gilt, wenn es gar keinen Weg zwischen zwei der drei Knoten u, v und w gibt oder gar alle drei nicht miteinander verbunden sind, denn in diesen Fällen gilt $\delta(\cdot, \cdot) = \infty$. Wenn z.B. $\delta(v, w) = \infty$, so muss entweder $\delta(v, u) = \infty$ oder $\delta(u, w) = \infty$ sein (denn sonst wäre u ja auf einem kürzesten Weg zwischen v und w).

Mit diesem Theorem lässt sich das *Relaxationsprinzip* als Basis für einen Lösungsalgorithmus eines Küreste-Wege-Problems begründen. Es lautet in Pseudocode:

```
/* dist[v][w] = bisher berechnete Distanz von v nach w
 * next[v][w] = bisher bestimmter Knoten, um von v nach w zu kommen
 */
if (dist[v][w] > dist[v][u] + dist[u][w]) {
   dist[v][w] ← dist[v][u] + dist[u][w];
   next[v][w] ← u;
}
```

Hier stellt der Matrixeintrag `dist[v][w]` den durch einen Algorithmus bislang gefundenen kürzeste Entfernung von v nach w dar. Der Matrixeintrag `next[v][w]` speichert den diesem Entfernungswert gefundenen Nachfolgeknoten, der von v nach w zu laufen ist.

Beispiel 12.8 Gegeben sei der gewichtete Graph in Abbildung 12.5. Die drei Matrizen γ, `dist`

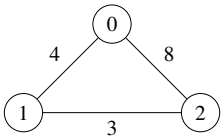


Abbildung 12.5: Ein gewichteter Graph

und `next` lauten dann:

$$\gamma = \begin{pmatrix} \infty & 4 & 8 \\ 4 & \infty & 3 \\ 8 & 3 & \infty \end{pmatrix}, \qquad \texttt{dist} = \begin{pmatrix} 8 & 4 & 7 \\ 4 & 6 & 3 \\ 7 & 3 & 6 \end{pmatrix}, \qquad \texttt{next} = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 2 & 2 \\ 1 & 1 & 1 \end{pmatrix}. \tag{12.6}$$

Die Einträge `dist[0][2] = 7` und `next[0][2] = 1` beispielsweise bedeuten dann, dass ein kürzester Weg von 0 nach 2 die Länge 7 hat und von 0 nach 1 führt. Die weitere Knotenfolge dieses Weges ergibt dann, indem dann man den Eintrag `next[1][2] = 2` betrachtet, d.h. man gelangt von 1 nach zwei am schnellsten über den Knoten 2:

$$\texttt{next[0][2]} = 1 \quad \rightarrow \quad \texttt{next[1][2]} = 2,$$

wie man hier in dem kleinen Graphen leicht überprüfen kann □

Als Datenstrukturen können die beiden Arrays auf zwei verschiedene Weisen implementiert werden. Einerseits können wir sie als Attribute des *Graphen* betrachten, d.h. als zwei *zwei*dimensionale Arrays; dies wird im Folgenden mit dem Floyd-Warshall realisiert. Andererseits können sie aufgefasst werden als Attribute eines Objektes „Knoten“, in diesem Falle dann aber als *ein*dimensionale Arrays, die jedem einzelnen Knoten gehören; diese Sichtweise wird in dem unten beschriebenen Dijkstra-Algorithmus umgesetzt werden.

12.4 Floyd-Warshall-Algorithmus

Wir betrachten nun den Floyd-Warshall-Algorithmus zur Lösung des APSP-Problems. Dieser Algorithmus fasziniert durch seine Einfachheit. Er wurde 1962 unabhängig voneinander durch by R.W. Floyd und S. Warshall entwickelt. Er kann als Methode eines Graphen gemäß dem folgenden Klassendiagramm implementiert werden.

GewichteterGraph
weight: `double[ ][ ]` # dist: `double[ ][ ]` # next: `int[ ][ ]`
+ floydWarshall(): void

Der Floyd-Warshall-Algorithmus wird hier ohne Parameter aufgerufen. Als statische Methode (bzw. alleinstehende Funktion) würde er als Parameter die Gewichtsmatrix `weight` des Graphen erhalten und als Ergebnis die zwei Arrays `[dist, next]` berechnen.

```
/** Objektmethode, die die Attributarrays dist und next berechnet. */
algorithm floydWarshall() {
   // Initialisierung:
   for (v from 0 to n − 1) {
      for (w from 0 to n − 1) {
         dist[v][w] ← weight[v][w];
         if (weight[v][w] < inf) {
            next[v][w]  ← w;
         } else {
            next[v][w] ← −1;
         }
      }
   }

   for (u from 0 to n − 1) {
      for (v from 0 to n − 1) {
         for (w from 0 to n − 1) {
            // Relaxationsprinzip:
            if (dist[v][w] > dist[v][u] + dist[u][w]) {
               dist[v][w] ← dist[v][u] + dist[u][w];
               next[v][w] ← u;
            }
         }
      }
   }
}
```

Natürlich ist die Einfachheit eines Algorithmus noch lange keine Garantie dafür, dass er auch korrekt ist. Das müssen wir schon mathematisch beweisen, und natürlich unter Angabe der genauen Annahmen, die für die Korrektheit hinreichend sind. Das liefert das folgende Theorem, und die vorausgesetzte Annahme ist genau die zur Lösung auch notwendige: keine negativen Kreise.

Theorem 12.9 (Korrektheit des Floyd-Warshall-Algorithmus) *Enthält ein gewichteter Graph mit der Gewichtsmatrix* `weight` *keine negativen Kreise, so löst der Floyd-Warshall-Algorithmus das APSP und liefert für jedes Knotenpaar $v, w \in G$ den Wert*

$$\texttt{dist[v][w]} = \delta(v, w),$$

also genau die Distanz von v nach w.

Beweis. Nach dem Initialisierungsschritt gilt zunächst `dist[v][w] = weight[v][w]`. Damit sind bereits alle Wege mit nur einer Kante bestimmt. In jeder Iteration der dreifach verschachtelten Schleife wird für jeden möglichen Weg mit u Kanten geprüft, ob das Relaxationsprinzip anzuwenden ist. Da es keine negativen Kreise gibt, kann ein kürzester Weg keine Kante doppelt enthalten, d.h. nach $u = n - 1$ Kanten sind alle in Frage kommenden Wege geprüft und die Distanzen für alle Knotenpaare v und w berechnet. Q.E.D.

Beispiel 12.10 Gegeben sei der gewichtete Graph in Abbildung 12.2 aus Beispiel 12.5. Die Matrizen `dist` und `next` lauten nach dem Initialisierungsschritt:

$$\texttt{dist} = \begin{pmatrix} \infty & 5 & 2 & \infty \\ \infty & \infty & \infty & 4 \\ \infty & 1 & \infty & 5 \\ 3 & \infty & \infty & \infty \end{pmatrix}, \qquad \texttt{next} = \begin{pmatrix} -1 & 1 & 2 & -1 \\ -1 & -1 & -1 & 3 \\ -1 & 1 & -1 & 3 \\ 0 & -1 & -1 & -1 \end{pmatrix}$$

Der erste Schleifendurchlauf für $(u, v, w) = (0, 0, 0)$ lässt beide Matrizen unverändert, da die if-Bedingung nicht wahr ist, und ebenso die folgenden drei mit $(u, v, w) = (0, 0, w)$ mit $w = 1, 2, 3$. Überhaupt kann die Bedingung nur wahr werden, wenn u, v und w paarweise ungleich sind, d.h. der erste zu prüfende Fall tritt für $(u, v, w) = (0, 1, 2)$ ein: Da

$$\texttt{dist}(1,2) = \infty \quad = \quad \texttt{dist}(1,0) + \texttt{dist}(0,2) = \infty + 2 = \infty$$

(mit der Rechenregel $\infty + x = \infty$ für jedes $x \in \mathbb{R}$), ist jedoch auch hier die Bedingung nicht wahr, ebenso wie für $(u, v, w) = (0, 1, 3)$ und $(u, v, w) = (0, 2, w)$ mit $w = 0, 1, 2, 3$. Erst bei $(u, v, w) = (0, 3, 1)$ wird mit

$$\texttt{dist}(3,1) = \infty \quad \geqq \quad \texttt{dist}(3,0) + \texttt{dist}(0,1) = 3 + 5 = 8$$

die Bedingung wahr und `dist[3][1]` $\leftarrow 8$, `next[3][1]` $\leftarrow 0$:

$$\texttt{dist} = \begin{pmatrix} \infty & 5 & 2 & \infty \\ \infty & \infty & \infty & 4 \\ \infty & 1 & \infty & 5 \\ 3 & \mathbf{8} & \infty & \infty \end{pmatrix}, \qquad \texttt{next} = \begin{pmatrix} -1 & 1 & 2 & -1 \\ -1 & -1 & -1 & 3 \\ -1 & 1 & -1 & 3 \\ 0 & \mathbf{0} & -1 & -1 \end{pmatrix}$$

Am Ende gilt:

$$\texttt{dist} = \begin{pmatrix} 10 & 3 & 2 & 7 \\ 7 & 10 & 9 & 0 \\ 8 & 1 & 10 & 5 \\ 3 & 6 & 5 & 10 \end{pmatrix}, \qquad \texttt{next} = \begin{pmatrix} 3 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 3 & 1 & 3 & 3 \\ 0 & 2 & 0 & 2 \end{pmatrix}$$

Die Distanz von 0 nach 3 ist insbesondere `dist[0][3]` $= 7$ und ein kürzester Weg ergibt sich aus

$$\texttt{next[0][3]} = 2 \quad \rightarrow \quad \texttt{next[2][3]} = 3$$

zu $(0, 2, 3)$, wie wir es „zu Fuß“ bereits in Beispiel 12.5 auf Seite 124 bestimmt hatten. □

Komplexitätsanalyse. Der Floyd-Warshall-Algorithmus besteht aus zwei verschachtelten Schleifen. Bei n Knoten des Graphen durchläuft die erste Schleife genau n^2 Iterationen, die zweite genau n^3 Iterationen[1]:

$$T_{\text{FW}}(n) = \Theta(n^3). \tag{12.7}$$

In einem „dichten“ Graphen, in dem fast alle Knoten direkt miteinander verbunden sind, ist die Kantenanzahl nach Gleichung (10.3) ungefähr gleich n^2. In diesem Fall ist der Floyd-Warshall-Algorithmus also nahezu optimal. In Graphen mit sehr wenigen Kanten dagegen prüfen die drei verschachtelten Schleifen viel zu viele unmögliche Kombinationen; in diesem Fall ist der Floyd-Warshall-Algorithmus also ineffizient.

[1] Kaderali und Poguntke (1995):§6.1.23.

12.5 Der Dijkstra-Algorithmus

In diesem Abschnitt behandeln wir einen effizienten Algorithmus, der das SSSP für Graphen mit nichtnegativen Gewichten löst. Er wurde 1956 von dem niederländischen Informatiker entwickelt und drei Jahre später veröffentlicht. Ähnlich wie der Floyd-Warshall-Algorithmus prüft der Dijkstra-Algorithmus iterativ, ob der bislang gefundene kürzeste Weg durch das Relaxationsprinzip verkürzt werden kann. Dabei wird der jeweils nächste zu prüfende Knoten aus einer Vorrangwarteschlange *(Priority Queue)* als temporärem Speicher entnommen, an deren Spitze der aktuell dem Startknoten nächste Knoten steht. Die Laufzeit des Algorithmus hängt damit auch davon ab, wie performant diese Datenstruktur aktualisierbar ist. In der hier beschriebenen Version wird sie als Minimumheap implementiert.

Zu beachten ist, dass sich die Distanz eines Knotens in der Warteschlange im Laufe des Algorithmus häufig ändern kann. Als Datenstruktur muss sie also eine Zugriffsmethode `decreaseKey` bereitstellen, die die Distanz eines Knoten ändert und die Queue wieder neu sortiert. Insgesamt hat die Vorrangwarteschlange also die folgenden Zugriffsmethoden:

- int **extractMin():** Gibt den Knoten mit der aktuell minimalen Distanz vom Startknoten zurück und löscht ihn aus der Vorrangwarteschlange.
- **decreaseKey(Knoten, neueDistanz):** Mit dieser Methode wird die Distanz für den spezifizierten Knoten mit dem übergebenen Wert `neueDistanz` belegt und damit die Vorrangwarteschlange neu sortiert.

Damit können wir den Dijkstra-Algorithmus mit Hilfe der in Abbildung 12.6 dargestellten Klassenstruktur implementieren. Hier stellt das verborgene Attribut `adjazenz` eines Knotens

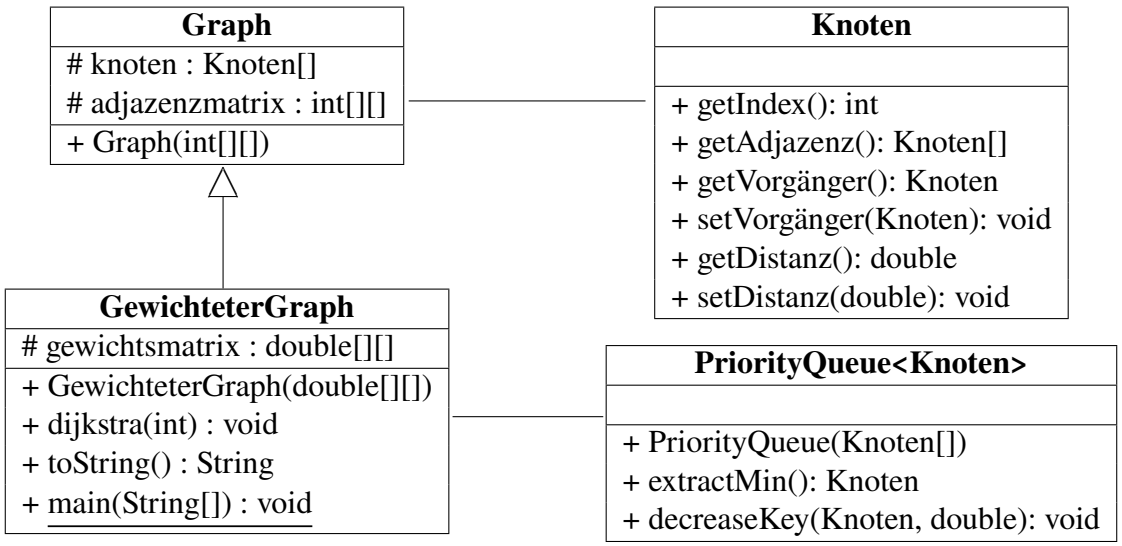


Abbildung 12.6: Klassendiagramm zum Dijkstra-Algorithmus

die Adjazenzliste des Knotens dar, und jeder Knoten hat als Attribut seinen spezifischen Index in dem Graphen. Der Dijkstra-Algorithmus wird mit dem Index s eines Knotens als Eingabeparameter aufgerufen. Die restlichen (verborgenen) Attribute des Knotens werden von dem Dijkstra-Algorithmus verwendet, in `distanz` steht nach seinem Aufruf die Distanz des Konotens zum Startknoten, und in `vorgänger` der jeweilige Vorgängerknoten, der auf dem kürzesten Weg vom Startknoten zu diesem Knoten liegt.

Aufgrund der Ausgangslage des SSSP, nämlich einem vorgegebenen Startknoten s, verwendet der Dijkstra-Algorithmus die folgende Version des Relaxationsprinzips:

```
/* dist[w] = bisher berechnete Distanz von s nach w
 * pred[w] = bisher bestimmter Vorgängerknoten von w des kürzesten Wegs von s
 */
if (dist[s][w] > dist[s][u] + weight[u][v]) {
   dist[s][w] ← dist[s][u] + weight[u][w];
   pred[w] ← u;
}
```

Insgesamt lautet der Algorithmus von Dijkstra damit wie folgt.

```
algorithm dijkstra(s) {
   input : Index s des Startknotens knoten[s]
   output: speichert für jeden Knoten die Distanz und den Vorgänger auf
           einem kürzesten Weg von s

   // Initialisiere Attribute aller Knoten:
   for(i: 0 to n − 1) {
      setze distanz von knoten[i] auf ∞;
      setze vorgänger von knoten[i] auf null;
   }
   setze distanz von knoten[s] auf 0; // Distanz des Startknotens zu sich selbst

   erzeuge PriorityQueue<Knoten> q mit allen Knoten;

   // wende für den jeweils nächstgelegenen Knoten in q das Relaxationsprinzip an:
   while(q ist nicht leer) {
      u ← q.extractMin(); // entferne Knoten mit geringster Distanz
      for(w in Adjazenz von u) {
         d  ← u.getDistanz() + gewicht[u.getIndex()][w.getIndex()];
         if (w.getDistanz() > d) { // Dreiecksungleichung verletzt?
            w.setDistanz(d);    // setze Attribut distanz von w auf d
            w.setVorgänger(u);  // setze Attribut vorgänger von w auf u
            q.decreaseKey(w,d); // sortiere Vorrangschlange mit neuer Distanz um
         }
      }
   }
}
```

Beispiel 12.11 Betrachten wir den Dijkstra-Algorithmus am Beispiel eines gewichteten Graphen mit vier Knoten wie in Abbildung 12.7. Die jeweils aktuell berechneten Distanzen sind

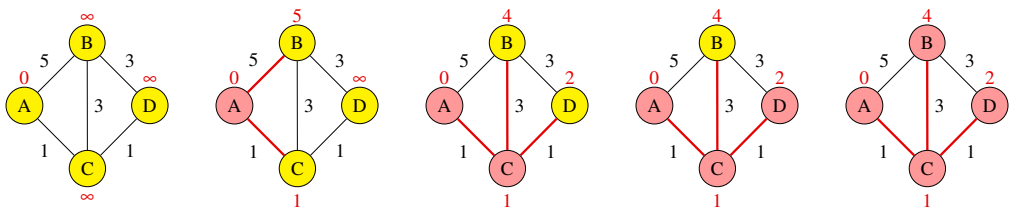


Abbildung 12.7: Ablauf des Dijkstra-Algorithmus für den Startknoten A.

als rote Zahlen an den Knoten angegeben, die jeweiligen Vorgänger durch rot markierte Kanten.

Gezeigt sind hier die fünf Phasen (1) nach der Initialisierung und der Speicherung aller Knoten in der Vorrangwarteschlange, (2) nach dem ersten Durchlauf der while-Schleife, in dem A aus der Warteschlange entfernt wird, danach (3), (4) und (5), wo sukzessive C, D und B entfernt werden. In Phase (2) ist beispielsweise der Vorgänger von B der Knoten A, nach Überprüfung der Dreiecksungleichung in Phase (3) allerdings sind die Distanz auf 4 und der Vorgänger auf C angepasst. □

Beispiel 12.12 *(Negative Gewichte)* Bei Graphen mit negativen Gewichten kann der Dijkstra-Algorithmus versagen. Ein einfaches Beispiel dazu ist der Graph in Abbildung 12.8. Hier liefert

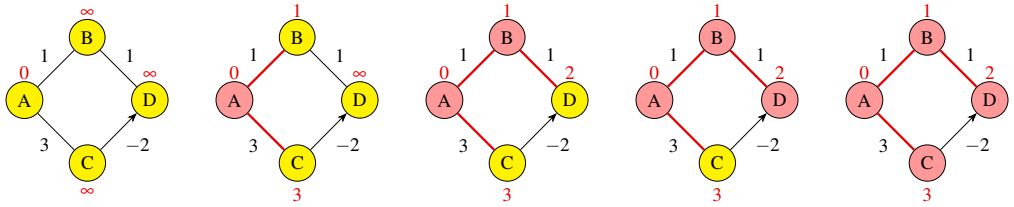


Abbildung 12.8: Negative Gewichte: Der Dijkstra-Algorithmus liefert hier als kürzesten Weg von A nach D den Weg A-B-D mit der Länge 2, dabei ist es tatsächlich der Weg A-C-D mit der Distanz 1.

der Dijkstra-Algorithmus als kürzesten Weg von A nach D den Weg A-B-D mit der Länge 2, dabei ist es tatsächlich der Weg A-C-D mit der Distanz 1. Die Ursache für das Versagen des Dijkstra-Algorithmus ist, dass er ein *Greedy-Algorithmus* ist, also ein Algorithmus, der in jedem Schritt diejenige Option wählt, die das beste Ergebnis verspricht. Im Allgemeinen finden solche „gierigen“ Algorithmen nur lokale Optima, aber nicht immer auch globale. □

Theorem 12.13 (Komplexitätsanalyse) *Der Dijkstra-Algorithmus mit einem Heap als Vorrangwarteschlange berechnet ein SSSP in einem gewichteten Graphen $G_\gamma = (V, E, \gamma)$ mit nichtnegativen Gewichten in einer Laufzeit $T_{\text{Dijkstra}}(|V|, |E|)$ und mit Speicherbedarf $S_{\text{Dijkstra}}(|V|)$ gegeben durch*

$$T_{\text{Dijkstra}}(|V|, |E|) = O\big((|V| + |E|) \cdot \log |V|\big), \qquad S_{\text{Dijkstra}}(|V|) = \Theta(|V|). \tag{12.8}$$

Hierbei ist $|V|$ die Anzahl der Knoten des Graphen und $|E|$ die Anzahl der Kanten.

Beweis. Betrachten wir zunächst die Laufzeit T_{Dijkstra}. Die erste for-Schleife („Initialisierung“) hat eine Laufzeitkomplexität von $\Theta(|V|)$, da sie jeden Knoten durchläuft. Die Erzeugung der Vorrangwarteschlange kostet durch die geeignete Umsortierung mit der Einfügemethode für einen Heap nach §3.2 auf Seite 36ff $O(n \cdot T_{\text{insert}}(n)) = O(n \log n)$ Laufzeit, wenn $n = |V|$ die Anzahl der Knoten des Graphen ist. Die verschachtelte Schleife `while` ... `for` schließlich erfordert die folgenden Laufzeitkomplexitäten:

```
while(q ist nicht leer) {        // genau → n Durchläufe
   u ← q.extractMin();           // → O(log n)
   for (w in Adjazenz von u) {   // → Außengrad von u
      ...
      if (...) {
         ...
         q.decreaseKey(w,d);     // → O(log n)
      }
   }
}
```

Die Entnahme je eines Knotens `u` aus der Vorrangwarteschlange wird also insgesamt genau n-mal durchgeführt, die Anzahl aller `for`-Schleifendurchläufe zusammengezählt ist genau die Anzahl $|E|$ der Kanten des Graphen.

Für die Gesamtlaufzeit T_{Dijkstra} müssen wir also die Laufzeiten T_{insert} für das Einfügen T_{extract} für die Entnahme und T_{decrease} für die Umsortierung zusammenrechnen,

$$\begin{aligned} T_{\text{Dijkstra}}(|V|,|E|) &= T_{\text{insert}}(|V|) + T_{\text{extract}}(|V|) + T_{\text{decrease}}(|V|,|E|) \\ &= O(|V| \cdot \log|V|) + O(|V| \cdot \log|V|) + O(|E| \cdot \log|V|) \\ &= O\big((|V| + |E|) \cdot \log|V|\big) \end{aligned} \tag{12.9}$$

Das ist genau die erste Gleichung in (12.8). Um den Speicherplatzbedarf S_{Dijkstra} zu ermitteln, reicht es zu erkennen, dass die einzige interne Speicherstruktur die Vorrangwarteschlange q ist zur Speicherung der $\Theta(|V|)$ Knoten ist. Q.E.D.

Bemerkung 12.14 Eine wesentliche Rolle für die Laufzeit des des Dijkstra-Algorithmus spielt die intern verwendete Datenstruktur der Vorrangwarteschlange. Würden wir beispielsweise stattdessen ein unsortiertes Array oder eine unsortierte Liste verwenden, so findet man das Minimimum darin mit einer Laufzeit $O(|V|)$ statt $O(\log|V|)$, d.h. $T_{\text{extract}}(|V|) = O(|V|^2)$ in (12.9), d.h. es gilt insgesamt nur noch

$$T_{\text{Dijkstra}}(|V|,|E|) = T_{\text{Dijkstra}}(|V|) = O(|V|^2). \tag{12.10}$$

Verwendet man andererseits als Vorrangwarteschlange die komplizierte Datenstruktur *Fibonacci-Heap*, so hat der Dijkstra-Algoritmus sogar eine Laufzeitkomplexität von nur $O(|E| + |V| \cdot \log|V|)$[2]. □

[2] Heun (2000):§5.4 & 5.5.

A Anhang

A.1 Mathematischer Anhang

A.1.1 Mathematische Notationen

Definition A.1 Für eine reelle Zahl x bezeichnen wir mit $\lfloor x \rfloor$ die größte ganze Zahl, die kleiner oder gleich x ist, oder formaler:

$$\lfloor x \rfloor = \max\{n \in \mathbb{Z} |\ n \leqq x\}. \tag{A.1}$$

Die Klammern $\lfloor \ldots \rfloor$ heißen *untere Gauß-Klammern* oder auf Englisch *floor-brackets*. □

Zum Beispiel gilt $\lfloor 5.43 \rfloor = 5$, $\lfloor \pi \rfloor = 3$, $\lfloor \sqrt{2} \rfloor = 1$, $\lfloor -5.43 \rfloor = -6$. Für zwei positive Ganzzahlen $m, n \in \mathbb{N}$ gilt

$$\left\lfloor \frac{m}{n} \right\rfloor = m \text{ div } n,$$

wobei „div“ die ganzzahlige Division bezeichnet.

A.1.2 Die modulo Operationen „mod“ und „%“

In der mathematischen Literatur finden Sie oft die Notation

$$k = n \bmod m, \qquad \text{oder} \qquad k \equiv n \bmod m.$$

Als Operator aufgefasst ähnelt mod sehr dem aus den C-ähnlichen Programmiersprachen bekannten Operator %. In der Tat sind mod und % für zwei *positive* Zahlen identisch, d.h. es gilt $n \bmod m = n \,\%\, m$. Die allgemeine auch für negative Werte geltende Definition lautet jedoch

$$n \bmod m = n - \lfloor n/m \rfloor \qquad \text{für } m \neq 0 \tag{A.2}$$

und

$$n \bmod 0 = n \tag{A.3}$$

speziell für $m = 0$, vgl.[1]. Diese Definition gilt allgemein sogar für reelle Zahlen $m, n \in \mathbb{R}$, Hinter dem Operator mod steht die Idee, eine Periodizität auszudrücken, also die „Phase“ $(n \bmod m)$ einer „Periode“ m darzustellen. Für $m = 3$ beispielsweise ergibt sich das periodische Muster der

[1] Graham et al. (1994):p. 82.

Zahlen $0 \to 1 \to 2 \to 0 \to \ldots$ nahtlos vom negativen in den positiven Zahlbereich, während % eine Art Punktspiegelung am Nullpunkt darstellt:

n	$\cdots$	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9	10	$\cdots$
n mod 3	$\cdots$	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	$\cdots$
n % 3	$\cdots$	-2	-1	0	-2	-1	0	1	2	0	1	2	0	1	2	0	1	$\cdots$

Zum Beispiel gilt $-5 \bmod 3 = 1$, aber $-5 \,\%\, 3 = -(5 \,\%\, 3) = -2$.

A.1.3 Exponentielle und logarithmische Funktionen

Es gilt für reelle Zahlen $a > 0$, $x, y \in \mathbb{R}$ die Beziehung

$$\boxed{a^{x+y} = a^x \cdot a^y.} \tag{A.4}$$

(Für negative Zahlen a gilt diese Beziehung nur, wenn x und y ganze Zahlen sind.) Der Logarithmus zur Basis a ist das Inverse der Exponentialfunktion a^x, also

$$\mathrm{a}^{\log_a x} = x \qquad \log_a \mathrm{a}^y = y. \qquad (a, x > 0,\ y \in \mathbb{R}) \tag{A.5}$$

Wir bezeichnen mit $\ln x$ den natürlichen Logarithmus einer positiven Zahl x, d.h. $\ln x = \log_{\mathrm{e}} x$, der demnach die Umkehrfunktion der Exponentialfunktion e^x ist:

$$\mathrm{e}^{\ln x} = x. \qquad \ln \mathrm{e}^y = y. \qquad (x > 0,\ y \in \mathbb{R}) \tag{A.6}$$

Weiter gelten für den Logarithmus die Rechengesetze

$$\boxed{\log_a(xy) = \log_a x + \log_a y,} \qquad \boxed{c \cdot \log_a x = \log_a x^c.} \qquad (x, y > 0,\ c \in \mathbb{R}) \tag{A.7}$$

Ein Wechsel der Basis des Logarithmus kann gemäß der Regel

$$\boxed{\log_a x = \frac{\log_b x}{\log_b a}.} \qquad (a, b \in \mathbb{N},\ x > 0) \tag{A.8}$$

durchgeführt werden. Insbesondere gilt $\log_a x = \frac{1}{\ln a} \ln x$, d.h. alle Logarithmen können auf den natürlichen Logarithmus zurück geführt werden.

Tabelle A.1: Häufige Funktionen der Algorithmusanalyse

Beschreibung	**Notation**	**Definition**
Abrundung *(floor)*	$\lfloor x \rfloor$	größte ganze Zahl $\leqq x$
Aufrundung *(ceiling)*	$\lceil x \rceil$	kleinste ganze Zahl $\geqq x$
Natürlicher Logarithmus	$\ln x$	$\log_e x$, (y, so dass $e^y = x$)
Binärer Logarithmus	$\lg x$	$\log_2 x$, (y, so dass $2^y = x$)
Bitzahl	$\lfloor 1 + \lg n \rfloor$	Anzahl Bits der Binärdarstellung von n
Harmonische Reihe	H_n	$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}.$
Fakultät	$n!$	$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n.$

A.2 Berechnung des ggT mit Primfaktorzerlegung

Das folgende Programm in Java liefert eine Implementierung der (nicht effizienten!) Berechnung des größten gemeinsamen Teilers ggT(m, n) zweier natürlicher Zahlen $m, n \in \mathbb{N}$. Es basier auf dem Algorithmus 6.1 auf Seite 60.

Listing A.1: Der ggT mit Hilfe der Primfaktorzerlegung in Java

```
import java.util.TreeMap;

public class GGT {
   /** Gibt ggT(m,n) zurück. Der Algorithmus verwendet die Primfaktorzerlegungen
    *  von m und n.
    *  @param m eine natürliche Zahl
    *  @param n eine natürliche Zahl
    *  @return der größte gemeinsame Teiler von m und n
    */
   public static int primfaktorzerlegung(int m, int n) {
      TreeMap<Integer,Integer> F_m = primfaktoren(m);
      TreeMap<Integer,Integer> F_n = primfaktoren(n);
      int exponent, ggT = 1;

      for (int faktor : F_m.keySet()) {
         if (F_n.containsKey(faktor)) {
            exponent = Math.min(F_m.get(faktor), F_n.get(faktor));
            ggT *= (int) Math.pow(faktor, exponent);
         }
      }
      return ggT;
   }

   /** Gibt eine Map {p -> e} aller Primfaktoren p und ihrer Vielfachheiten e
    *  einer natürlichen Zahl n zurück.
    *  @param n eine natürliche Zahl
    *  @return Map {p -> e} aller Primfaktoren p von n und ihrer Vielfachheiten e
    */
   public static TreeMap<Integer,Integer> primfaktoren(int n) {
      TreeMap<Integer,Integer> faktoren = new TreeMap<>();
      int prime;
      int e;     // stores current exponent;
      int p = 2; // stores current prime
      int sqrtN = (int) Math.sqrt(n);
      boolean plus2Step = true; // flag to control the wheel

      while (p <= sqrtN) {
         if (n % p == 0) {
            e = 1;
            n = n/p;
            while (n % p == 0 && n > 1) {
               e = e + 1;
               n = n/p;
            }
```

```
            faktoren.put(p,e); // add p^e
         }
         if (p == 2) {
            p = p + 1;
         } else {
            p = p + 2;
         }
      }
      if (n > 1) { // n itself is prime
         faktoren.put(n, 1);  // add n^1
      }
      return faktoren;
   }

   public static void main(String[] args) {
      int m = 6, n = 4;
      System.out.println("primteiler("+m+","+n+")="+primfaktorzerlegung(m,n));
      m = 48; n = 60;
      System.out.println("primteiler("+m+","+n+")="+primfaktorzerlegung(m,n));
   }
}
```

A.3 Beweis des Theorems 9.9 zum Geburtstagsparadoxon

Theorem A.2 *Unter der „idealen" Voraussetzung, dass eine gegebene Hashfunktion eine Menge von n Wörtern gleichverteilt auf m Hashwerte abbildet, ist die Wahrscheinlichkeit für mindestens eine Kollision durch*

$$p(m,n) \;=\; 1 \;-\; \frac{m(m-1)(m-2)\cdots(m-n+1)}{m^n} \tag{A.9}$$

gegeben.

Beweis. Bezeichnen wir zur Vereinfachung $p = p(m,n)$ und definieren

$$q = 1 - p. \tag{A.10}$$

Dies ist die Gegenwahrscheinlichkeit von p. Wir werden zunächst q berechnen, erst danach p von q herleiten. Wie also berehcnet man q? Bezeichnen wir mit q_i die Wahrscheinlichkeit, dass das i-te Wort kollisionsfrei auf einen Hashwert abgebildet wird, unter der Voraussetzung, dass dies für alle $j < i$ auch galt, so folgt

$$q = q_1 \cdot q_2 \cdot \ldots \cdot q_n.$$

Wir sehen gleich, dass $q_1 = 1$ gilt, denn zu Beginn sind alle m Hashwerte frei und das erste Wort wird sicher kollisionsfrei einem Hashwert zugeordnet. Beim zweiten Wort jedoch ist bereits ein Hashwert besetzt und es gibt nur noch $m - 1$ freie Hashwerte. D.h., $q_2 = (m-1)/m$. So leiten wir allgemein her, dass

$$q_i = \frac{m-i+1}{m} \qquad 1 \leqq i \leqq n,$$

da das i-te Wort immer schon $(i-1)$ besetzte Hashwerte vorfindet. Daraus ergibt sich aber genau Gleichung (A.9). Q.E.D.

Literatur

Arora, S. und B. Barak (2009). *Computational Complexity. A Modern Approach*. Cambridge University Press: Cambridge.

Barth, A. P. (2003). *Algorithmik für Einsteiger*. Vieweg: Braunschweig Wiesbaden.

Buchmann, J. A. (2001). *Introduction to Cryptography*. Springer-Verlag: New York.

Cormen, T. H. et al. (2001). *Introduction to Algorithms*. 2. Aufl. McGraw-Hill: New York.

de Vries, A. und V. Weiß (2021). *Grundlagen der Programmierung*. Vorlesungsskript. Hagen. URL: https://www.fh-swf.de/media/neu_np/fb_tbw_1/dozentinnen_2/professorinnen_5/devries_1/java.pdf.

Diestel, R. (2000). *Graphentheorie*. 2. Aufl. Springer-Verlag: Berlin Heidelberg.

Forster, O. (2008). *Analysis 1*. 9. Aufl. Vieweg: Wiesbaden.

Graham, R. L., D. E. Knuth und O. Patashnik (1994). *Concrete Mathematics*. 2. Aufl. Addison-Wesley: Upper Saddle River, NJ.

Güting, R. H. (1997). *Datenstrukturen und Algorithmen*. B.G. Teubner: Stuttgart.

Hackel, S., T. Schäfer und W. Zimmer (2010). *Praktische Sicherheitskonzepte*. Hrsg. von H. Neuroth et al. Bd. 2.3. urn:nbn:de:0008-20100305103. Verlag Werner Hülsbusch: Boizenburg.

Harel, D. und Y. Feldman (2006). *Algorithmik. Die Kunst des Rechnens*. Springer-Verlag: Berlin Heidelberg.

Havil, J. (2009). *Verblüfft?!* Springer-Verlag: Berlin Heidelberg.

Heun, V. (2000). *Grundlegende Algorithmen*. Vieweg: Braunschweig Wiesbaden.

Hoffmann, D. W. (2009). *Theoretische Informatik*. Carl Hanser Verlag: München.

Kaderali, F. und W. Poguntke (1995). *Graphen, Algorithmen, Netze. Grundlagen und Anwendungen in der Nachrichtentechnik*. Vieweg: Braunschweig Wiesbaden.

Knuth, D. E. (1998). *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*. 3. Aufl. Addison-Wesley: Reading.

Krumke, S. O. und H. Noltemeier (2012). *Graphentheoretische Konzepte und Algorithmen*. 3. Aufl. Springer Vieweg: Wiesbaden.

O'Neil, C. (2016). *Weapons of Math Destruction. How Big Data Increases Inequality and Threatens Democracy*. Penguin Random House: New York.

Ottmann, T. und P. Widmayer (2012). *Algorithmen und Datenstrukturen*. 5. Aufl. Spektrum Akademischer Verlag: Heidelberg Berlin.

Padberg, F. (1996). *Elementare Zahlentheorie*. 2. Aufl. Spektrum Akademischer Verlag: Heidelberg Berlin.

Schöning, U. (1997). *Algorithmen — kurzgefasst*. Spektrum Akademischer Verlag: Heidelberg Berlin.

— (2001). *Algorithmik*. Spektrum Akademischer Verlag: Heidelberg Berlin.

Sedgewick, R. und K. Wayne (2014). *Algorithmen: Algorithmen und Datenstrukturen*. Pearson: Hallbergmoos.

Sipser, M. (2006). *Introduction to the Theory of Computation*. Thomson Course Technology: Boston.

Turing, A. M. (1936–1937). „On computable numbers, with an application to the Entscheidungsproblem“. In: *Proc. London Math. Soc.* 42(2). http://www.turingarchive.org/browse.php/B/12, S. 230–265.

Vöcking, B. et al. (2008). *Taschenbuch der Algorithmen*. Springer-Verlag: Berlin Heidelberg. DOI: 10.1007/978-3-540-76394-9.

Vossen, G. und K.-U. Witt (2016). *Grundkurs Theoretische Informatik*. 6. Aufl. Springer Vieweg: Wiesbaden. DOI: 10.1007/978-3-8348-2202-4.

Internetquellen

[PAP] http://www.ecma-international.org/publications/files/ECMA-ST-WITHDRAWN/ECMA-4,%202nd%20Edition,%20September%201966.pdf – ECMA (1966): Standard ECMA-4 – Flow Charts (Withdrawn)

[ISO] https://www.iso.org/obp/ui/#!iso:std:63598:en – ISO/IEC 2382-1: Information Technology – Vocabulary

[NIST] http://www.nist.gov/dads/ – NIST Dictionary of Algorithms and Data Structures

[SORT] https://toptal.com/developers/sorting-algorithms/ – Animationen von Sortieralgorithmen