Algorithmen und Datenstrukturen

Friedhelm Seutter
Institut für Software Engineering
Hochschule Braunschweig/Wolfenbüttel

1. September 2017

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Was ist ein Algorithmus?

Der Begriff *Algorithmus* ist heute untrennbar mit der Erstellung von Computerprogrammen verbunden. Aber schon lange bevor es Computer überhaupt gab, gab es Algorithmen, auch wenn dieser Begriff noch nicht verwendet wurde. So z. B. die im *Papyrus Rhind* zusammengestellten Rechenaufgaben aus dem 16. Jh. v. Chr. [23] oder das von Euklid um 300 v. Chr. aufgeschriebene Verfahren zur Bestimmung des größten gemeinsamen Teilers zweier Zahlen, das heute *Euklidischer Algorithmus* genannt wird [6]. Intuitiv läßt sich die Bedeutung des Begriffs Algorithmus folgendermaßen beschreiben:

> *Ein Algorithmus ist ein allgemeines, eindeutiges Verfahren zur Lösung einer Klasse gleichartiger Probleme, gegeben durch einen aus elementaren Anweisungen bestehenden Text.*

Hierunter fallen natürlich insbesondere mathematische Verfahren und – wie eingangs schon festgestellt – Computerprogramme. Aber auch Bedienungsanleitungen von technischen Anlagen, Zusammenbauanleitungen von Bausätzen oder auch Kochrezepte können in einem erweiterten Sinn als Algorithmen bezeichnet werden.

Die Bedeutung des Begriffs *Algorithmus* wird nach Donald E. Knuth (*1938) durch Angabe weiterer Kriterien präzisiert [13]:

- **Endlichkeit:** Ein Algorithmus besteht aus endlich vielen elementaren Anweisungen (Schritten) und muß nach endlich vielen Schrittausführungen enden.
- **Bestimmtheit:** Jeder Schritt des Algorithmus muß eindeutig definiert sein (*determiniert*). Dazu gehört auch die Bestimmung des ersten Schrittes und des Algorithmusendes. Bei gleicher Eingabe wird immer das gleiche Ergebnis geliefert. Der jeweils nächste auszuführende Schritt muß eindeutig bestimmt sein (*deterministisch*). Bei gleicher Eingabe ist der Rechenverlauf immer gleich.
- **Eingabe:** Ein Algorithmus hat keine oder endlich viele Eingabewerte aus einer Eingabemenge. Diese Werte werden initial an den Algorithmus gegeben, bevor er mit der Ausführung des ersten Schritts beginnt.
- **Ausgabe:** Ein Algorithmus hat mindestens einen Ausgabewert aus einer Ausgabemenge. Die Eingabe steht in einer bestimmten Relation zur Ausgabe. Diese Relation wird durch den Algorithmus definiert.
- **Effektivität:** Die einzelnen Anweisungen müssen überhaupt ausführbar sein. Wenn z. B. eine Bedingung in einer Anweisung auf der Lösung eines bisher noch ungelösten Problems beruht, dann ist diese Anweisung nicht effektiv.

Die obige Beschreibung eines Algorithmus bleibt aber trotz der Präzisierungen sehr vage. Begriffe wie „Verfahren“, „Anweisung“, „Schritt“ oder „Wert“ sind zwar intuitiv verständlich, aber nicht im mathematischen Sinn exakt definiert. Insbesondere erfordert die Interpretation des Begriffs „elementar“ eine vage Vorstellung von der Zielgruppe oder der Zielmaschine, die den Algorithmus ausführen soll.

Ursprünglich geht der Begriff *Algorithmus* zurück auf den persisch-arabischen Mathematiker Abu Ja'far Mohammed Ibn Musa Al-Khowarizmi (ca. 780 - ca. 850). In seiner Abhandlung über Arithmetik führt er das aus dem 5. Jahrhundert aus Indien stammende dezimale Positionssystem mit der Null zur Darstellung von Zahlen und die entsprechenden Rechenverfahren in der arabischen Welt ein. Hiervon existiert allerdings nur eine lateinische Übersetzung aus dem 12. Jahrhundert [25]. In Anlehnung an seinen Namen wurden diese Verfahren *Algorismi* genannt [11].

Erst ab dem 10. Jahrhundert gelangt dann dieses Zahlsystem mit den *arabischen Ziffern* und den Rechenverfahren nach Europa. Hier sind einmal die Mauren in Spanien zu nennen und vor allem Leonardo von Pisa, genannt Fibonacci (ca. 1170 - 1250), der dieses Zahlsystem und die Algorismi in seinem *Liber abaci* (Buch vom Abakus) veröffentlicht [20]. Es dauert dann aber noch einmal

gut zwei Jahrhunderte bis sich das Dezimalsystem in Europa durchsetzt. Für das deutschsprachige Europa ist hier insbesondere Adam Ries (1492 - 1559) zu nennen, der mit seinem Rechenbuch *Rechenung nach der lenge, auff den Linihen vnd Feder* für die Verbreitung der Rechenregeln unter dem gemeinen Volk sorgt [24].

Im Folgenden ist aus einer Übersetzung von Al-Khowarizmis Abhandlung über Arithmetik der Anfang des zweiten Kapitels *Kapitel über die Vergrößerung und Verringerung* zitiert [7]. In dem ersten Kapitel wird dort das indische dezimale Positionssystem zur Darstellung von Zahlen beschrieben.

> *Wenn du eine Zahl zu einer Zahl addieren willst oder eine Zahl von einer Zahl subtrahieren, so schreibe beide Zahlen in zwei Zeilen, d. h. eine von ihnen unter die andere, und es sei die Stelle der Einer unter der Stelle der Einer und die Stelle der Zehner unter der Stelle der Zehner. Wenn du nun beide Zahlen zusammenfügen willst, d. h. eine zur anderen addieren, so addiere jede Stelle zu der Stelle, die über ihr von ihrer Art ist, d. h. Einer zu Einern und Zehner zu Zehnern. Und wenn an irgendeiner der Stellen, d. h. an der Stelle der Einer oder Zehner oder an irgendeiner anderen (Stelle), 10 angesammelt sind, so setze sie als Eins und bringe jene (Eins) zur darüberliegenden Stelle; d. h.: Wenn du an der ersten Stelle, die die Stelle der Einer ist, 10 hast, so mache aus ihnen eine Eins und hebe sie zur Stelle der Zehner, und dort wird sie 10 bezeichnen. Wenn aber etwas von der Zahl, die unterhalb von 10 ist, übrig bleibt oder wenn die Zahl selbst unterhalb von Zehn ist, so lasse sie an derselben Stelle. Und wenn nichts übrig bleibt, so setze einen Kreis, damit die Stelle nicht leer ist, sondern (damit) sich an ihr ein Kreis befindet, der ihre Stelle einnimmt, damit die Stellen nicht zufällig, wenn sie leer ist, verringert werden und man glaubt, daß die zweite (Stelle) die erste ist, und du so in deiner Zahl getäuschst wirst. So mache es auch bei allen anderen Stellen. Auf ähnliche Weise mache, wenn an der zweiten Stelle 10 angesammelt sind, aus ihnen eine Eins und hebe sie zur dritten Stelle, und dort bezeichnet sie Hundert. Und was unterhalb von 10 übrig geblieben ist, bleibt dort. Wenn aber nichts übrig geblieben ist, so setze dort, wie oben, einen Kreis. Und so mache es (auch) an den übrigen Stellen, wenn es mehr gibt.* [1]

In den Abhandlungen über das Rechnen mit dem indischen dezimalen Positionssystem, die ab dem 12. Jahrhundert entstehen, werden auch zunehmend beispielhaft Rechenverfahren zur Lösung von Alltagsproblemen beschrieben. Dabei handelt es sich z. B. um Probleme im Zusammenhang mit Eigentumsübertragungen, also Kauf, Verkauf, Vermietung und Pacht; bei Warengeschäften werden Rabatte oder Skonto gewährt, bei Geldgeschäften fallen Zinsen an oder Währungen müssen umgerechnet werden; Handwerker müssen Maße und Mischungsverhältnisse berechnen. Auch solche Rechenverfahren werden *Algorismi* genannt, auch wenn der Ursprung dieses Wortes dabei in Vergessenheit gerät [21]. In Anlehnung an das griechische *arithmós* (Zahl), aus dem sich der Begriff *Arithmetik* entwickelt hat, entsteht daraus der Begriff *Algorithmus*.

Dieser Begriff wird zwar noch von dem Philosophen und Mathematiker Christian von Wolff (1679 - 1754) in seinem 1716 erschienenen Lexikon *Mathematisches Lexicon* [26] nur zur Bezeichnung der Notationen der vier arithmetischen Grundrechenarten verwendet.

> Algorithmus, *Werden genennet die vier Rechnungs-Arten in der Rechen-Kunst, nemlich: Addiren, Subtrahiren, Mulipliciren und Dividiren zusammen genommen.* [2]

Aber auch darüber hinaus werden die Beschreibungen zunehmend komplexerer Rechenverfahren als Algorithmen bezeichnet. So verwendet Gottfried Wilhelm Leibniz (1646 - 1716) den Begriff bereits 1684 in seinen Abhandlungen *Über die Analyse des Unendlichen* bei der Beschreibung von Verfahren der Differentialrechnung [16]. Er bleibt aber ein intuitiver Begriff. Trotz seiner für die Mathematik unpräzisen Fassung gibt es keine Probleme zu entscheiden, ob ein Verfahren ein Algorithmus ist oder nicht.

Erst mit Beginn des 20. Jh. wird der Algorithmusbegriff formal definiert. Es treten mathematische Probleme in den Vordergrund, deren Lösbarkeit zweifelhaft ist. Solange für ein Problem ein

[1] Seiten 45, 47 in [7]
[2] Spalte 38 in [26]

intuitiver Algorithmus zu seiner Lösung angegeben werden kann, besteht keine Notwendigkeit zu seiner Formalisierung. Soll aber die Nicht-Existenz eines Algorithmus mathematisch bewiesen werden, so ist das auf einer intuitiven Ebene nicht möglich. Ausgehend von David Hilberts (1862 - 1943) Zielsetzung zur Einführung einer Beweistheorie, in der Widerspruchsfreiheit, Unabhängigkeit und Vollständigkeit eines Axiomensystems formal bewiesen werden können, und dem damit in Verbindung stehenden und von ihm formulierten *Entscheidungsproblem* [9, 10] wird der Algorithmusbegriff formal definiert. Zwei grundlegende Ansätze, die sich aber als äquivalent erweisen, sind hier zu nennen: Der Ansatz über *rekursive Funktionen* (Alonzo Church (1903 - 1995) [2, 3], Stephen C. Kleene (1909 - 1994) [12], Kurt Gödel (1906 - 1978) [8]) und der Ansatz über ein Maschinen-Modell (Alan Turing (1912 - 1954) [22], Emil Post (1897 - 1954) [18]). Notation und Funktionsweise sind exakt definiert und damit auch Eigenschaften und Aussagen des bzw. über den Algorithmus mathematisch beweisbar. Heute lassen sich darüber auch Beziehungen zu Programmiersprachen und Computermodellen herstellen, um die Leistungsfähigkeit und die Beschränkungen der realen Computersysteme theoretisch zu untermauern.

Neben dem *formalen* Algorithmusbegriff bleibt aber auch der *intuitive* Algorithmusbegriff bestehen, so wie er am Anfang dieses Kapitels beschrieben ist. Ein Algorithmus, der diese Kriterien erfüllt, wird entsprechend *intuitiver Algorithmus* genannt. Die hier beschriebenen Algorithmen werden intuitive Algorithmen sein. Auch wenn zu deren Spezifikation neben der natürlichen Sprache auch Sprachelemente der Mathematik und Konzepte von Programmiersprachen herangezogen werden, so setzen diese lediglich auf einem allgemeinen Grundverständnis von Mathematik und Programmiertechnik auf. Intuitiver und formaler Algorithmusbegriff werden über die so genannte *Churchsche These* als äquivalent angesehen. Beweisbar ist diese Aussage allerdings nicht.

Auf eine Verallgemeinerung des Algorithmusbegriffs muß noch eingegangen werden: Verfahren, bei denen auf das Kriterium der *endlich vielen Schrittausführungen* verzichtet wird, aber sonst alle Kriterien eines Algorithmus gelten, werden *Berechnungsmethoden* (*Methoden* bzw. *Prozeduren*) genannt. Berechnungsmethoden enden für einige Eingaben der Eingabemenge ggf. nicht (*halten nicht an*). Der Einfachheit halber werden diese häufig dennoch Algorithmen genannt. Die Problematik, ob ein Verfahren ein Algorithmus ist oder eine Berechnungsmethode, ob also ein Algorithmus immer anhält oder nicht, führt auf das sogenannte *Halteproblem*. Eine weitergehende Fragestellung ist die, welche Probleme durch Algorithmen oder Methoden lösbar bzw. entscheidbar sind. Gibt es überhaupt Probleme, die nicht lösbar sind? Ein Problem heißt *berechenbar*, wenn es durch einen Algorithmus gelöst werden kann.

In einer weitergehenden Betrachtung von Algorithmen erweist sich das Kriterium der Endlichkeit eines Algorithmus (bzgl. der Anzahl der Schrittausführungen) als zu grob. Es gibt Algorithmen, die ein Problem sehr schnell lösen, andere, die sehr viel Zeit für eine Lösung benötigen. Die Zeit wird dabei durch die Anzahl der Schrittausführungen in Abhängigkeit der Eingabegröße (z. B. Zahlenwert der Eingabe oder Anzahl der Eingabeobjekte) ausgedrückt. So gibt es Algorithmen mit z. B. linearen, quadratischen oder auch exponentiellen Abhängigkeiten. Man spricht von der *Effizienz* oder auch von der *Komplexität* von Algorithmen. So ist man in diesem Sinne auch stets daran interessiert, den besten Algorithmus für ein gegebenes Problem zu finden. Antworten auf diese Fragestellungen erfordern eine detaillierte Algorithmusanalyse. Zusätzlich ist es notwendig bzgl. der Elementarität der Schritte und der Darstellung der Eingaben einige normierende Angaben zu machen.

1.2 Darstellung von Algorithmen

Zur Darstellung von Algorithmen werde von einer JAVA-ähnlichen Notation ausgegangen, die teils natürlich-sprachliche und teils formale Elemente enthält. Solche Darstellungen werden *Pseudocode* genannt. Dabei stehen die prozeduralen Aspekte der Algorithmen im Vordergrund. Auf die jeweiligen Datenstrukturen wird nur soweit notwendig eingegangen, Klassen werden nicht definiert.

Ein Algorithmus besteht aus einem Namen, einer Parameterliste und der Anweisungsliste. Die Eingabe und Ausgabe des Algorithmus ist mit den zulässigen Wertebereichen zu spezifizieren. Werte eines bestimmten Wertebereichs bzw. eines Typs werden in Algorithmen *Variablen* zugewiesen und über diese referenziert. Die Typen der Variablen der Parameterliste ergeben sich aus der Eingabespezifikation; die Typen der sonstigen Variablen ergeben sich aus dem Kontext, sie werden nicht deklariert. Die Spezifikation der Eingabevariablen orientiert sich an den aus der Mathematik bekannten Objekten und Notationen:

$a \in \mathbb{Z}$
$A = (a_1, \ldots, a_n) \in \mathbb{Z}^n, n \in \mathbb{N}$

Die Zuordnung zu den entsprechenden elementaren und zusammengesetzten Typen aus Programmiersprachen sollte aber immer leicht möglich sein. So erfolgt hier – auf so genannte *Felder* oder *arrays* – der Zugriff auf einzelne oder einen Bereich von Komponenten analog zu Programmiersprachen:

$A[i]$	liefert die i-te Komponente von A, hier $a_i \in \mathbb{Z}, 1 \leq i \leq n$
$A[i..j]$	liefert die Komponenten i bis j von A, hier $(a_i, \ldots, a_j) \in \mathbb{Z}^{j-i+1}, 1 \leq i \leq j \leq n$

Inhomogen zusammengesetzte Objekte, d. h. Objekte unterschiedlicher Typen, werden *Strukturen* oder *records* genannt, ihre Komponenten *Attribute* oder *Felder*. Betrachte z. B. eine Adresse:

$Adr = (Straße, Nr, PLZ, Ort) \in V^* \times \mathbb{N} \times \mathbb{N} \times V^*$ mit $V = \{a, b, c, \ldots, z\}$	
$Straße[Adr]$	liefert den Wert des Attributes *Straße*
$Nr[Adr]$	liefert den Wert des Attributes *Nr*

Die grundlegende elementare Anweisung ist die *Wertzuweisung* an eine Variable, wobei der Wert z. B. durch einen arithmetischen Ausdruck bestimmt wird. Die Anweisungen eines Algorithmus werden sequentiell angeordnet und beginnend mit der ersten in dieser Sequenz ausgeführt. Zur Abweichung von dieser Sequenz werden sogenannte Steueranweisungen eingefügt, z. B. *Sprünge* (Ansprung einer bestimmten Anweisung), *Verzweigungen* (Alternativ auszuführende Anweisungen) oder auch *Schleifen* (Wiederholung von Anweisungen). Die Steuerung in diesen Anweisungen erfolgt in Abhängigkeit von dem Zutreffen oder Nichtzutreffen von Bedingungen.

Die Zusammenfassung von Anweisungssequenzen zu Blöcken wird durch Einrücken spezifiziert. Öffnende und schließende Klammern bzw. die reservierten Wörter *begin* und *end* werden nicht geschrieben.

Eine Anweisung eines Algorithmus kann auch der Aufruf eines anderen Algorithmus oder der Aufruf von sich selbst sein. Dabei werden die Werte der aktuellen Parameter an die formalen Parameter (*call by value*) übergeben. Bei Objekten zusammengesetzter Datentypen ist es sicherlich effizienter, nur eine Referenz auf dieses Objekt (*call by reference*) zu übergeben. Dies soll aber den jeweiligen Implementierungen vorbehalten bleiben und hier nicht diskutiert werden. Terminiert der aufgerufene Algorithmus mit einer **return**-Anweisung, so wird dieser Wert bei Rückkehr an den aufrufenden Algorithmus geliefert.

Der Algorithmus terminiert, wenn keine weitere Anweisung mehr auszuführen ist. Die in der Ausgabe spezifizierten Variablen halten die Ausgabewerte des Algorithmus. Eine **return**-Anweisung muss dazu nicht explizit aufgeführt werden, sie kann aber.

Im folgenden werden die elementaren Konzepte eines Algorithmus aufgelistet und kurz erläutert, mit dem Algorithmus 1.1 ist dann ein kleines Beispiel gegeben. Zum Zwecke der Erläuterung, Diskussion und Analyse von Algorithmen werden die Zeilen nummeriert.

Wertzuweisung: Einer Variablen eines bestimmten Datentyps wird ein Wert zugewiesen. Der Wert wird dabei durch einen arithmetischen, relationalen oder logischen Ausdruck bestimmt. Die Bildung von Ausdrücken erfolgt wie in der Mathematik üblich. Operanden sind Variable oder Konstante, Operatoren sind z. B. $+, -, \mathrm{mod}, \wedge, \vee, =, \leq$. Zur Modifikation der üblichen Auswertungsreihenfolge wird Klammerung verwendet. Die Wertzuweisung selbst wird durch $\leftarrow$ ausgedrückt. Die Variable steht dabei links und der Ausdruck rechts des Zuweisungssymbols.

$$\begin{array}{lcl} a & \leftarrow & 5 \\ b & \leftarrow & 2 \cdot (3 + a) \end{array}$$

Nach Ausführung dieser zwei Wertzuweisungen hat die Variable a den Wert 5 und b den Wert 16.

Sprung: Es wird eine mit einer *Marke* versehene Anweisung direkt angesprungen. Dabei sind Vor- und Rückwärtssprünge zulässig.

goto M1
$\vdots$
M1: ...

Die Ausführung einer **exit**-Anweisung bewirkt einen Sprung an das Ende des Algorithmus und damit einen sofortigen Abbruch.

Verzweigung: Eine **if**-Anweisung besteht aus einer Bedingung (einem logischen Ausdruck), einem **then** und einem **else**-Teil. Ist die Bedingung wahr, wird in den **then**-Teil gesprungen, dieser ausgeführt und der **else**-Teil übersprungen. Ist die Bedingung falsch, wird der **else**-Teil angesprungen und dieser ausgeführt. Der **else**-Teil kann auch enfallen.

if $a \geq b$
  then $c \leftarrow a - b$
  else $c \leftarrow b - a$

then und **else**-Teil können auch aus einer Folge von Anweisungen bestehen.

Schleife: Eine Schleife besteht aus einer Schleifensteuerung und einem Schleifenrumpf. Die Folge von Anweisungen im Schleifenrumpf wird so lange die Bedingung in der Schleifensteuerung wahr ist wiederholt. Es gibt drei Arten von Schleifen: die **for**-Schleife, die **while**-Schleife und die **do**-Schleife. Bei der **for**-Schleife wird ein Zähler von einem Anfangs- bis zu einem Endwert hoch- bzw. runtergezählt, bei der **while**-Schleife wird die Bedingung jeweils vor Ausführung des Schleifenrumpfes überprüft und bei der **do**-Schleife jeweils nachher.

for $i \leftarrow 1$ **to** 10
  do ...

while $a \geq b$
  do ...

do ...
  while $a \geq b$

Die Ausführung einer **break**-Anweisung innerhalb eines Schleifenrumpfes führt unmittelbar zu einem Abbruch der Schleife.

Kommentar: Durch $\triangleright$ wird der Rest einer Zeile im Algorithmus als Kommentar gekennzeichnet.

Die hier entlehnten elementaren Konzepte aus Programmiersprachen dienen lediglich dazu, die Notation und den Ablauf von intuitiven Algorithmen zu präzisieren und sie von Ungenauigkeiten und Mehrdeutigkeiten zu befreien. Im Sinne einer exakten Definition der Syntax und Semantik von

Algorithmus 1.1 (Maximum)
Eingabe: $A = (a_1, \ldots, a_n)$ mit $a_i, i, n \in \mathbb{N}, 1 \leq i \leq n$
Ausgabe: $m = \max(a_1, \ldots, a_n)$

```
MAXIMUM(A)
1  m ← A[1]
2  for i ← 2 to n
3      do if A[i] > m
4            then m ← A[i]
5  return m
```

Programmiersprachen bleiben natürlich viele Aspekte offen. Dies betrifft nicht nur die Programmstrukturen, sonderen auch die damit eng verbundenen Datenstrukturen, die in Programmiersprachen weit komplexer sein können als das Konzept von Variablen, die Werte eines bestimmten Typs, also aus einer Wertemenge, annehmen können.

Auch ohne an dieser Stelle eine Programmiersprache exakt und vollständig definiert zu haben, so ist doch offensichtlich, daß ein Computerprogramm die Kriterien eines intuitiven Algorithmus erfüllt und somit auch als ein Algorithmus im intuitiven Sinn angesehen werden kann. Mit einer entsprechenden formalen Spezifikation einer Programmiersprache ist ein Computerprogramm natürlich auch ein Algorithmus im formalen Sinn.

Kapitel 2

Analyse von Algorithmen

2.1 Verifikation

Zur Lösung eines gegebenen Problems ist zuerst ein Algorithmus zu entwickeln und zu spezifizieren. Dieser muß natürlich wirklich das Problem lösen und weiterhin auch fehlerfrei sein. Der Nachweis der Korrektheit eines Algorithmus in diesem Sinne wird *Verifikation* genannt.

Zur Verifikation von Algorithmen sind vor und nach jeder Anweisung bzw. jeder Anweisungssequenz Prädikate (Bedingungen, Eigenschaften) über den Variablen des Algorithmus zu spezifizieren, die über die Ausführung der Anweisungen gültig bzw. wahr bleiben. Insbesondere muß diese Gültigkeit vor und nach Ausführung des Algorithmus insgesamt gegeben sein, d. h. die Eingaben werden auch tatsächlich in die zu berechnenden Ausgaben überführt. Die Prädikate werden in diesem Kontext *Zusicherungen* bzw. *assertions* genannt.

Seien S eine Sequenz von Anweisungen und P und Q Zusicherungen. S besteht dabei aus einer oder mehreren Anweisungen oder ist ein Block, also eine Zusammenfassung einer Sequenz von Anweisungen. Das Tripel

$$\{P\}\ S\ \{Q\}$$

ist dann folgendermaßen zu interpretieren: Falls P vor Ausführung von S gilt, dann gilt Q danach. P heißt auch *Vorbedingung* bzw. *precondition* und Q *Nachbedingung* bzw. *postcondition*. Die Nachbedingung kann nur gültig (wahr) werden, wenn S terminiert. Terminiert S immer und ist dann Q gültig, dann spricht man von *totaler Korrektheit*. Terminiert S aber nicht immer, dann spricht man von *partieller Korrektheit*, wenn für die terminierenden Fälle dann Q gültig ist.

Die Korrektheit eines Algorithmus insgesamt ist dann entsprechend seines Aufbaus aus Wertzuweisungen und elementaren Kontrollstrukturen ausgehend von der Korrektheit einzelner Anweisungssequenzen nachzuweisen. Dabei kann nach den folgenden Regeln vorgegangen werden.

- Wertzuweisung:

  Seien x eine Variable und a eine Konstante. Es gilt dann $\{P\}\ x \leftarrow a\ \{Q\}$ mit $P = Q$, falls in P jedes freie Vorkommen von x durch a ersetzt wird, auch geschrieben $P[x \leftarrow a] = Q$.

- Sequenz:

  Falls $\{P\}\ S\ \{R\}$ und $\{R\}\ T\ \{Q\}$ bewiesen, dann ist auch $\{P\}\ S;T\ \{Q\}$ bewiesen.

- Verzweigung:

  Falls $\{P \wedge B\}\ S\ \{Q\}$ und $\{P \wedge \neg B\}\ T\ \{Q\}$ bewiesen, dann ist auch
  $\{P\}$ **if** B **then** S **else** $T\ \{Q\}$ bewiesen.

- Schleife:

  Falls $\{I \wedge B\}\ S\ \{I\}$ bewiesen, dann ist auch $\{I\}$ **while** B **do** $S\ \{I \wedge \neg B\}$ bewiesen.

  Die Nachbedingung eines Schleifendurchlaufs ist gleichzeitig die Vorbedingung des nächsten Schleifendurchlaufs. Die Zusicherungen werden hier deshalb *Schleifeninvariante* oder kurz *Invariante* genannt.

  Über das Terminieren der Schleife wird durch diese Regel aber nichts ausgesagt. Dieses ist unabhängig davon zu beweisen.

  In einer **for**-Schleife ist die Bedingung B von der Form $i_a \leq i \leq i_e$ und es gilt analog:

  Falls $\{I \wedge (i_a \leq i \leq i_e)\}\ S\ \{I\}$ bewiesen,
  dann ist auch $\{I\}$ **for** $i \leftarrow i_a$ **to** i_e **do** $S\ \{I \wedge (i < i_a \vee i > i_e)\}$ bewiesen.

  Dabei ist i der Schleifenzähler, i_a der Anfangswert und i_e der Endwert.

Betrachte dazu das Beispiel 2.1. Dabei sei n eine Variable vom Type *integer*. Die Zusicherungen sind vor und nach jeder Anweisung als Kommentar eingefügt. Das Symbol $\perp$ bedeutet, dass eine Variable zwar deklariert, ihr aber noch kein Wert zugewiesen ist. Ihr Wert ist also undefiniert.

Beispiel 2.1

$$
\begin{array}{l}
\vdots \\
\rhd \{n = \bot\} \\
\vdots \\
\rhd \{minint \leq n \leq maxint\} \\
\textbf{if } n > 0 \\
\quad \textbf{then } \rhd \{1 \leq n \leq maxint\} \\
\qquad\quad n \leftarrow n - 1 \\
\qquad\quad \rhd \{0 \leq n \leq maxint - 1\} \\
\quad \textbf{else } \rhd \{minint \leq n \leq 0\} \\
\qquad\quad n \leftarrow 0 \\
\qquad\quad \rhd \{n = 0\} \\
\rhd \{0 \leq n \leq maxint - 1\} \\
\vdots
\end{array}
$$

Dabei wurden die Zusicherungen in der Verzweigungsregel wie folgt zugeordnet:

P	:	$minint \leq n \leq maxint$
B	:	$n > 0$
$P \wedge B$	:	$1 \leq n \leq maxint$
$P \wedge \neg B$	:	$minint \leq n \leq 0$
Q'	:	$n = 0$
Q	:	$0 \leq n \leq maxint - 1$

Mit $Q' \Rightarrow Q$ ergibt sich Q als Nachbedingung.

□

Ein sehr wichtiges Konzept in Algorithmen stellen Schleifen dar. Hier müssen die Zusicherungen, die Invarianten, vor und nach jedem Schleifendurchlauf gültig sein. Betrachte dazu in Beispiel 2.2 den Algorithmus 1.1, der das Maximum einer endlichen Folge natürlicher Zahlen bestimmt. Die Schleifeninvariante ist hier durch das Prädikat

$$(1 \leq i \leq n) \wedge (m_i = \max(a_1, \ldots, a_i))$$

gegeben. Zur Verfolgung der Werte der Variablen m je Schleifendurchlauf wird die Variable indiziert. Es sei m_1 der Wert von m vor dem ersten Durchlauf mit $i = 2$ und m_i der Wert von m nach dem Durchlauf i, $2 \leq i \leq n$. Eingebeben wird das Feld natürlicher Zahlen $A = (a_1, \ldots, a_n)$, und ausgegeben werden soll das Maximum dieser Zahlen als Wert der Variablen m.

Der Algorithmus 1.1 löst also das gegebene Problem korrekt. In der Praxis werden mittels Programmiersprachen implementierte Algorithmen allerdings selten wirklich verifiziert. Hier müssen dann geeignete Teststrategien eingesetzt werden, die ein Programm beispielhaft überprüfen.

Beispiel 2.2

$$
\begin{array}{ll}
& \textsc{Maximum}(A) \\
& \quad \rhd \{A = (a_1, \ldots, a_n) \wedge m = \bot\} \\
1 & \quad m \leftarrow A[1] \\
& \quad \rhd \{A = (a_1, \ldots, a_n) \wedge m = a_1\} \\
& \quad \rhd \{m_1 = \max(a_1)\} \\
2 & \quad \textbf{for } i \leftarrow 2 \textbf{ to } n \\
& \qquad \rhd \{2 \leq i \leq n \wedge m_{i-1} = \max(a_1, \ldots, a_{i-1})\} \\
3 & \qquad \textbf{do if } A[i] > m \\
4 & \qquad\quad \textbf{then } m \leftarrow A[i] \\
& \qquad \rhd \{2 \leq i \leq n \wedge m_i = \max(a_1, \ldots, a_i)\} \\
& \quad \rhd \{m = m_n = \max(a_1, \ldots, a_n)\} \\
5 & \quad \textbf{return } m
\end{array}
$$

$\triangleright \{A = (a_1, \ldots, a_n) \wedge m = \max(a_1, \ldots, a_n)\}$

□

2.2 Komplexität

Nachdem zu einem gegebenen Problem ein Algorithmus gefunden und dieser verifiziert wurde ist die Frage nach seinem Berechnungsaufwand - seiner *Komplexität* - zu beantworten. Bei Computerprogrammen sind das die Fragen nach der Laufzeit und dem Speicherplatzbedarf. Bzgl. der Laufzeit wird die Anzahl der ausgeführten Anweisungen als Maß für die Komplexität betrachtet, bzgl. des Speicherplatzbedarfs die Anzahl der benötigten Speicherwörter. Damit können dann Algorithmen, die das gleiche Problem lösen, miteinander verglichen werden, um den *effizienteren* bzgl. Laufzeit und/oder Speicherplatzbedarf auszuwählen. Häufig geht ein Effizienzgewinn bei der Laufzeit zu Lasten des Speicherplatzbedarfs und umgekehrt. Darüber hinaus lassen sich Algorithmen mittels ihrer Komplexität aber auch schlechthin klassifizieren.

Es wird dabei vorausgesetzt, daß die Darstellung der Eingaben und Elementarität der Anweisungen bzw. der auszuführenden Schritte des Algorithmus von gleicher Art sind, denn sonst ist keine Vergleichbarkeit der Algorithmen gegeben. Kann z. B. eine natürliche Zahl in einem Schritt eingelesen werden, oder wird sie ziffernweise eingelesen, je Schritt eine Ziffer? Können in einem Schritt zwei natürliche Zahlen miteinander multipliziert werden, oder setzt sich die Multiplikation aus einer Sequenz von Additionsschritten zusammen? Bei Algorithmen in Pseudocode ist nun aber die Elementarität nicht näher definiert.

Andererseits ist man bei der Angabe der Komplexität eines Algorithmus auch gar nicht immer daran interessiert, die Anzahl der Schrittausführungen oder des Speicherplatzbedarfs exakt anzugeben. Es ist nur die jeweilige Größenordnung gesucht. Und auch diese nicht als absolute Größe, sondern in Abhängigkeit von der Größe der Eingabe. Ein Algorithmus, der z. B. eine dezimal dargestellte natürliche Zahl auf irgendeine Eigenschaft untersucht, wird bei Eingabe einer zweistelligen Zahl weniger Schritte benötigen als bei Eingabe einer zehnstelligen Zahl. Nur die Schrittanzahl oder der Speichplatzbedarf eines Algorithmus in Abhängigkeit von der Größe der Eingabe macht diese zu einem Komplexitätsmaß für den Algorithmus selbst.

Beispiel 2.3
Die Eingabegröße von Algorithmus 1.1 aus Abschnitt 1.2 ist die Größe des Zahlenfeldes $A = (a_1, \ldots, a_n)$, also n. Weitere Beispiele zur Eingabegröße einiger Probleme sind in Tabelle 2.1 aufgeführt. □

Problem	Eingabegröße
Suchen des Maximums in einem Zahlenfeld	Anzahl der Zahlen
Suchen eines Schlüssels in einem Zahlenfeld	Anzahl der Zahlen
Multiplikation zweier Matrizen	Anzahl der Matrixelemente
Sortieren eines Zahlenfelds	Anzahl der Zahlen
Lösen eines Graphproblems	Anzahl der Knoten und Kanten

Tabelle 2.1: Beispiele zur Eingabegröße einiger Probleme

Oben war bereits gesagt worden, dass als Komplexitätsmaß von Algorithmen häufig die Größenordnungen genügen würden. Zur Bestimmung der Zeitkomplexität kann man sich somit darauf beschränken, nur die *grundlegenden Anweisungen* bzw. *grundlegenden Operationen* eines Algorithmus zu bestimmen und die Anzahl ihrer Ausführungen zu zählen. Die grundlegenden Anweisungen sind diejenigen, die die Laufzeit des Algorithmus im Wesentlichen bestimmen. Die Gesamtzahl der Schrittausführungen des Algorithmus darf nur proportional zu der Anzahl der Ausführungen dieser grundlegenden Anweisungen sein. Diese Anzahl, ausgedrückt als Funktion von der Eingabegröße, bestimmt dann die Zeitkomplexität eines Algorithmus.

Beispiel 2.4
Als grundlegende Anweisungen von Algorithmus 1.1 kommen die Zusweisungen in den Zeilen 1 und 4 und der Vergleich in Zeile 3 in Betracht. Später wird gezeigt, dass der Vergleich in Zeile 3 auszuwählen ist. Weitere Beispiele zu den grundlegenden Anweisungen einiger Probleme sind in Tabelle 2.2 aufgeführt. □

Problem	grundlegende Anweisung
Suchen des Maximums in einem Zahlenfeld	Vergleich zweier Zahlen
Suchen eines Schlüssels in einem Zahlenfeld	Vergleich Schlüssel mit Zahl
Multiplikation zweier Matrizen	Multiplikation zweier Zahlen
Sortieren eines Zahlenfelds	Vergleich zweier Zahlen
Lösen eines Graphproblems	Durchlaufen eines Knotens oder einer Kante

Tabelle 2.2: Beispiele zu den grundlegenden Anweisungen einiger Probleme

Im Folgenden steht nun primär die Zeitkomplexität einiger Algorithmen im Fokus der Betrachtungen. Der Einfachheit halber wird dann meist nur von der Komplexität gesprochen.

Bei der Angabe der Komplexität eines Algorithmus ist in der Regel die Laufzeit im schlechtesten Fall (*worst case*) gefragt. Hier ist dann eine Funktion $g(n)$ gesucht, die die Schrittanzahl $f(n)$ nach oben beschränkt. Üblicherweise ist eine kleinste obere Schranke gesucht. Darüber hinaus ist aber auch der Berechnungsaufwand eines Algorithmus im besten Fall (*best case*) und im Mittel (*average case*) von Interesse. Zur Bestimmung des besten Falls ist die Schrittanzahl nach unten abzuschätzen, d. h. es ist eine größte untere Schranke gesucht, und für die Bestimmung der mittleren Schrittanzahl ist der Erwartungswert unter einer angenommenen Verteilung der Eingabegröße zu berechnen.

Definition 2.1 (best- und worst-case-Komplexitäten)
Es seien A ein Algorithmus, E_n die Menge seiner Eingaben der Eingabegröße $n \in \mathbb{N}_0$ und $e \in E_n$ eine Eingabe. Sei weiterhin $t(e)$ die Anzahl der Ausführungen der grundlegenden Anweisungen, die bei der Ausführung des Algorithmus A bei der Eingabe von e auftreten. Die best-case und die worst-case-Komplexität sind dann wie folgt definiert:

$$\begin{aligned} B(n) &:= \min\{t(e) \mid e \in E_n\} \\ W(n) &:= \max\{t(e) \mid e \in E_n\} \end{aligned}$$

$B(n)$ ist die kleinste Anzahl und $W(n)$ ist die größte Anzahl der grundlegenden Anweisungen, die der Algorithmus A bei der Eingabegröße n ausführt.

Für die Bestimmung der average-case-Komplexität eines Algorithmus werde für die Eingaben $e \in E_n$ der Größe n eine Wahrscheinlichkeitsverteilung zugrunde gelegt. Die Laufzeit eines Algorithmus A läßt sich dann als eine Zufallsvariable $t : E_n \to \mathbb{R}$ ausdrücken. Dem Ereignis $e \in E_n$ *ist Eingabe für den Algorithmus* A wird dann mittels der Zufallsvariablen die Laufzeit des Algorithmus bei Eingabe von e zugeordnet.

Definition 2.2 (average-case-Komplexität)
Es seien A ein Algorithmus, E_n die Menge seiner Eingaben der Eingabegröße $n \in \mathbb{N}_0$, $e \in E_n$ eine Eingabe und $p(e)$ die Wahrscheinlichkeit der Eingabe e. Sei weiterhin $t(e)$ die Anzahl der Ausführungen der grundlegenden Anweisungen, die bei der Ausführung des Algorithmus A bei der Eingabe von e auftreten. Die average-case-Komplexität ist dann wie folgt definiert:

$$A(n) := \sum_{e \in E_n} t(e) \cdot p(e)$$

$A(n)$ ist die mittlere Anzahl der grundlegenden Anweisungen, die der Algorithmus A bei der Eingabegröße n ausführt.

Die hier eingeführten Komplexitätsmaße werden im Folgenden für den Algorithmus 1.1 zur Bestimmung des Maximums einer endlichen Folge natürlicher Zahlen berechnet (Beispiel 2.5). An diesem Beispiel wird außerdem die Auswahl der grundlegenden Anweisung diskutiert. Unter (I.) wird die Zuweisung (Zeilen 1 und 4) und unter (II.) der Vergleich (Zeile 3) gewählt. Unter (III.) werden die Ergebnisse zusammengefaßt und die Begründung für den Vergleich (Zeile 3) als grundlegende Anweisung geliefert.

Beispiel 2.5
Betrachte den Algorithmus 1.1 zur Bestimmung des Maximums einer Folge n natürlicher Zahlen. Seine Eingabegröße ist n.

I. Grundlegende Anweisungen seien die Zuweisungen in Zeilen 1 und 4:

Zeile 1 wird einmal ausgeführt und Zeile 4 in Abhängigkeit von der Eingabe 0 bis $(n-1)$-mal. Ist das erste Element der Zahlenfolge das Maximum, dann wird die Zuweisung 0-mal ausgeführt, ist die Zahlenfolge aufsteigend sortiert, dann wird die Zuweisung $(n-1)$-mal ausgeführt. Daraus ergeben sich nun für den besten und schlechtesten Fall die folgenden Komplexitäten:

$$\begin{aligned} B(n) &= 1 \\ W(n) &= n \end{aligned}$$

Für die Bestimmung des mittleren Falls sei angenommen, die n eingegebenen Zahlen seien voneinander verschieden und alle $n!$ Permutationen von Eingaben seien gleichwahrscheinlich. Ohne Herleitung sei das Ergebnis angegeben:

$$A(n) \approx 1 + \ln n.$$

II. Grundlegende Anweisung sei der Vergleich in Zeile 3:

Zeile 3 wird unabhängig von der Eingabe in jedem der $n-1$ Schleifendurchläufe ausgeführt. Die Komplexitäten im besten, mittleren und schlechtesten Fall sind also gleich:

$$B(n) = A(n) = W(n) = n - 1$$

III. Diskussion:

Eine grundlegende Anweisung muss die Laufzeit des Algorithmus im Wesentlichen bestimmen. Die Gesamtzahl ihrer Ausführungen und die Gesamtzahl der Ausführungen aller Anweisungen müssen proportional zueinander sein, also ihre Anzahlen unterscheiden sich höchstens durch einen konstanten Faktor.

Für den Fall (I.) trifft das nicht zu, also können die Zuweisungen keine grundlegenden Anweisungen sein. Für den Fall (II.) gilt das aber. Der Vergleich in Zeile 3 kann als grundlegende Anweisung gewählt werden und zur Bestimmung der Komplexitäten des Algorithmus herangezogen werden. Es gilt also für den Algorithmus

$$B(n) = A(n) = W(n) = n - 1.$$

□

In der Regel wird die Komplexität eines Algorithmus häufig nur für seine schlechteste Laufzeit angegeben und auf die Angabe der Komplexität im mittleren und im besten Fall verzichtet. Nach [4] seien hierfür die folgenden Gründe angegeben:

- Die Laufzeit im schlechtesten Fall ist eine obere Schranke für alle Eingaben. Es ist damit eine Garantie gegeben, daß eine Bearbeitung in keinem Fall länger dauert.
- Die mittlere Laufzeit eines Algorithmus ist in den meisten Fällen fast genau so schlecht wie seine schlechteste Laufzeit. Ein Suchraum ist im Mittel nur zur Hälfte zu durchsuchen, der Faktor 1/2 ist für die Größenordnung aber irrelevant.
- Die Laufzeit im schlechtesten Fall passiert relativ häufig, z. B. bei nicht erfolgreichen Suchanfragen in Datenstrukturen.

- Die Bestimmung der mittleren Laufzeit setzt die Kenntnis der Verteilung der Eingabedaten voraus. Selbst wenn diese bekannt ist, gibt es keine Garantie über das Eintreten dieser Laufzeit.
- Die Laufzeit im besten Fall liefert keine Hinweise für die Laufzeit der sonstigen Eingaben bzw. Eingabegrößen.

Zum Speicherplatzbedarf war bisher nur ausgeführt worden, dass dieses Komplexitätsmass die Anzahl der benötigten Speicherwörten als Funktion der Eingabegröße betrachtet. Natürlich gehört dazu auch der Bedarf zur Speicherung der Eingabe. Der Speicherplatz, der durch den Algorithmus / das Programm belegt wird, gehört aber im Allgemeinen nicht dazu. Es wird auch zwischen einem besten, mittleren und schlechtesten Fall unterschieden. Wird zusätzlich zu dem benötigen Platz für die Eingabe nur konstant viel Speicherplatz benötigt, dann sagt man, der Algorithmus arbeite *am Platz*. Von der Einführung formaler Definitionen hierfür werde abgesehen.

Beispiel 2.6
Die Speicher-Komplexität des Algorithmus 1.1 beträgt $n+2$ Speicherwörter für natürliche Zahlen, n für das Feld A, eins für das Maximum m und eins für den Index i. Es handelt sich also um einen am Platz arbeitenden Algorithmus. □

2.3 Asymptotische Notationen

Im vorangegangenen Abschnitt wurde der Berechnungsaufwand eines Algorithmus als Funktion der Eingabegröße dargestellt, für die Laufzeit als Ausführungsanzahl von grundlegenden Anweisungen, für den Speicherplatzbedarf als Anzahl Speicherwörter. Es war angesprochen worden, dass in den meisten Fällen nicht die wirklichen Anzahlen von Interesse sind, sondern nur die jeweiligen Größenordnungen. Die Funktionen werden deshalb nur für große Eingabegrößen betrachtet und durch *asymptotische* Abschätzungen Komplexitätsklassen zugeordnet.

Die Notation der für die Analyse von Algorithmen wichtigen Komplexitätsklassen erfolgt über die von Paul Bachmann (1837 - 1920) eingeführte Ordnung einer Größe O (sprich „groß Oh“). Edmund Landau (1877 - 1938) erweiterte diese Notation zu den so genannten Landau-Symbolen. Da die Spezifikation der Eingabegröße, so wie sie hier benötigt wird, als natürliche Zahl erfolgt (z. B. Zahlenwert der Eingabe, Anzahl der Eingabeobjekte), werden nur auf den natürlichen Zahlen definierte Funktionen in die nicht-negativen reellen Zahlen $f : \mathbb{N} \to \mathbb{R}_0^+$ betrachtet.

Definition 2.3 (Groß-Oh)
Es seien $f, g : \mathbb{N} \to \mathbb{R}_0^+$ *Funktionen.*
$O(g) := \{f \mid$ *es gibt ein* $c > 0$ *und ein* $n_0 \in \mathbb{N}$,
so dass für alle $n \geq n_0$ *gilt:* $f(n) \leq c \cdot g(n)\}$

Es ist also $O(g)$ eine Menge bzw. eine Klasse von Funktionen. In ihr sind alle die Funktionen enthalten, deren Funktionswerte ab einem n_0 kleiner oder gleich dem Produkt aus der Konstanten c und den Funktionswerten von g sind. Die Funktion cg ist also eine obere Schranke für alle Funktionen in $O(g)$.

In Quantorenschreibweise hat die Groß-Oh-Definition die folgende Darstellung:

$$O(g) := \{f \mid \exists\, c > 0\ \exists\, n_0 \in \mathbb{N}\ \forall\, n \geq n_0 : f(n) \leq c \cdot g(n)\}$$

Neben $f \in O(g)$ findet man aus historischen Gründen auch die Schreibweise $f = O(g)$. Um zu verdeutlichen, dass die Funktionen über der Variablen n definiert sind, wird häufig auch $f(n) \in O(g(n))$ oder $f(n) = O(g(n))$ geschrieben. Die Konstante c in der obigen Definition sorgt dafür, daß auch Funktionen wie z. B. $f(n) = 3n^2$ in der Klasse $O(n^2)$ liegen. Die Konstante n_0 läßt endlich viele Ausnahmen zu, d. h. die Abschätzung muß nur für *fast alle* $n \in \mathbb{N}$ gelten. Man spricht auch von einer asymptotischen oberen Schranke. Die Abbildung 2.1 zeigt die obere Schranke einer Funktion als Graph.

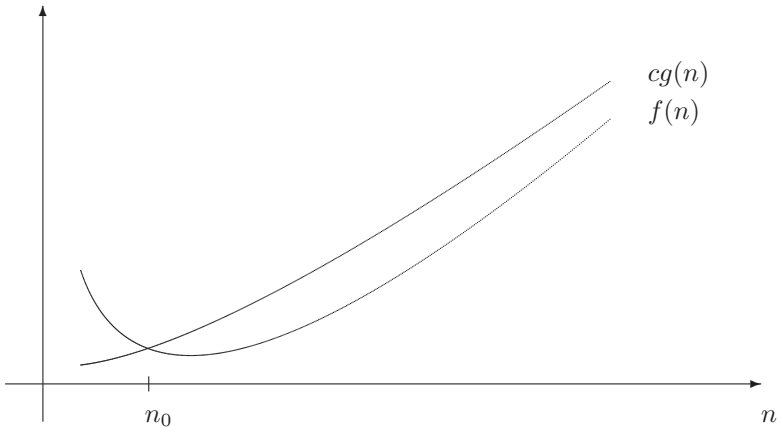


Abbildung 2.1: $f(n) \leq cg(n)$ für alle $n \geq n_0$

Beispiel 2.7
Betrachte die Funktion $f : \mathbb{N} \to \mathbb{N}$ mit $f(n) = 3n^3 + 2n^2 - 4$. Es soll gezeigt werden, dass $f \in O(g)$ mit $g : \mathbb{N} \to \mathbb{N}$ und $g(n) = n^3$. Dafür ist ein $c > 0$ und ein $n_0 \in \mathbb{N}$ zu finden, so dass

$$\forall\, n \geq n_0 : f(n) \leq cn^3$$

gilt.

$$\begin{aligned} f(n) &= 3n^3 + 2n^2 - 4 \\ &\leq 3n^3 + 2n^2 \\ &\leq 3n^3 + 2n^3 \\ &= 5n^3 \qquad \text{für alle } n \end{aligned}$$

Also, mit $c = 5$ und $n_0 = 1$ gilt $f(n) \in O(n^3)$. Die Funktion f hat die Größenordnung oder die asymptotische Ordnung n^3.

Leicht kann gezeigt werden, dass darüber hinaus auch $f(n) \in O(n^k)$ für $k \geq 3$ gilt. Von Interesse ist aber immer die Spezifikation einer kleinsten oberen Schranke bzgl. der Größenordnung. Hier tritt also die Frage auf, ob $f \in O(n^2)$. Es wäre also zu zeigen:

$$\exists\, c > 0\ \exists\, n_0 \in \mathbb{N}\ \forall\, n \geq n_0 : f(n) \leq cn^2$$

Aber es gilt

$$\begin{aligned} 3n^3 + 2n^2 - 4 &> cn^2 \\ \Leftrightarrow 3n + 2 - 4/n^2 &> c\ , \end{aligned}$$

denn mit wachsendem n geht $4/n^2$ gegen 0 und $3n$ wird größer als jede Konstante. Also $f(n) \notin O(n^2)$. □

In Beispiel 2.8 wird gezeigt, dass ein Polynom vom Grad m immer in der Klasse $O(n^m)$ enthalten ist.

Beispiel 2.8
Sei $f(n) = \sum_{i=0}^{m} a_i n^i$ mit $a_m \neq 0$ ein Polynom. Es läßt sich dann die folgende Abschätzung vornehmen:

$$\begin{aligned} f(n) &= a_m n^m + a_{m-1} n^{m-1} + \ldots + a_0 \\ &= a_m \cdot n^m \cdot \left(1 + \frac{a_{m-1}}{a_m}\frac{1}{n} + \ldots + \frac{a_0}{a_m}\frac{1}{n^m}\right) \\ &\leq |a_m| \cdot n^m \cdot \left(1 + \frac{|a_{m-1}|}{|a_m|}\frac{1}{n} + \ldots + \frac{|a_0|}{|a_m|}\frac{1}{n^m}\right) \\ &\leq |a_m| \cdot n^m \cdot \left(1 + \frac{|a_{m-1}|}{|a_m|} + \ldots + \frac{|a_0|}{|a_m|}\right) \end{aligned}$$

für alle $n \geq 1$. Mit

$$c := |a_m| \cdot \left(1 + \frac{|a_{m-1}|}{|a_m|} + \ldots + \frac{|a_0|}{|a_m|}\right)$$

gilt $f(n) \leq c \cdot n^k$ für alle $n \in \mathbb{N}$ und $k \geq m$. Es ist damit $f(n) \in O(n^k)$ für alle $k \geq m$ und insbesondere $f(n) \in O(n^m)$. □

Die Zeit- und Speicher-Komplexitäten von Algorithmen werden nun in einer solchen asymptotischen Notation ausgedrückt. Die als Funktion von n bestimmten Anzahlen von Ausführungen der grundlegenden Anweisungen und die Anzahlen von benötigten Speicherwörtern werden der Funktionsklasse zugeordnet, die die kleinste obere Schranke darstellt.

Beispiel 2.9
Betrachte nochmals die Komplexitäten des Algorithmus 1.1:

- Zeit-Komplexität: $B(n) = A(n) = W(n) = n - 1 \in O(n)$
- Speicher-Komplexität: $n + 2 \in O(n)$

□

Ein Algorithmus mit der Zeit-Komplexität $O(n)$ wird auch *linearer* Algorithmus genannt, oder man sagt, seine Komplexität sei *linear*. Analog spricht man bei $O(n^2)$ und $O(n^3)$ von quadratischer bzw. kubischer Komplexität. Eine konstante Komplexität eines Algorithmus wird durch $O(1)$ ausgedrückt. Diese sogenannten *polynomiellen* Komplexitätsklassen definieren für $k \in \mathbb{N}_0$ eine Hierarchie:

$$O(1) \subset O(n) \subset O(n^2) \subset O(n^3) \subset \ldots \subset O(n^k) \subset \ldots$$

Alle Algorithmen, die sich zu einer dieser Komplexitätsklassen zuordnen lassen werden *effizient* genannt. Also auch ein Algorithmus aus $O(n^{57})$ ist in diesem Sinne effizient. (Es ist klar, daß damit gemeint ist, daß das die kleineste Komplexitätsklasse sein soll, der der Algorithmus zugeordnet werden kann.) Insbesondere für größere n ist ein solcher Algorithmus praktisch unbrauchbar. Dennoch ist er allemal effizienter als ein Algorithmus, dessen Schrittanzahl sich nur durch eine exponentielle Funktion nach oben abschätzen läßt. Solche Exponentialfunktionen wachsen für große n stärker als jedes Polynom. Ein Algorithmus mit einer *exponentiellen* Komplexität, z. B. aus $O(a^n)$, $a \in \mathbb{R}$, wird *nicht effizient* genannt.

Satz 2.1 (Exponentielle Komplexität)
Für kein $m \in \mathbb{N}$ gilt $a^n \in O(n^m)$, $a \in \mathbb{R}$ und $a > 1$ konstant.

Beweis:
Annahme: $a^n \in O(n^m)$ für ein $m \in \mathbb{N}$.
Wegen $a^n = e^{n \ln a} = \sum_{i=0}^{\infty} \frac{(\ln a)^i}{i!} n^i > \sum_{i=0}^{m+1} \frac{(\ln a)^i}{i!} n^i \in O(n^{m+1})$ kann es kein solches m geben. □

Im Bereich der konstanten, linearen und quadratischen Komplexitäten wird, wenn möglich, eine noch feinere Klassenzuordnung mittels Wurzelfunktionen und *logarithmischer* Funktionen gesucht. Es gilt:

$$O(1) \subset O(\log n) \subset O(\sqrt{n}) \subset O(n) \subset O(n \cdot \log n) \subset O(n^2)$$

Da sich die Logarithmen unterschiedlicher Basen durch Multiplikation mit einem konstanten Faktor umrechnen lassen, z. B. $\log_a n = \frac{1}{\ln a} \cdot \ln n$ mit $a \in \mathbb{R} \setminus \{1\}$, lassen sich alle logarithmischen Komplexitätsklassen einer Komplexitätsklasse zuordnen. Es kann deshalb auf die Angabe einer Basis verzichtet werden. Allgemein gilt

$$\frac{\log_{b_1} n}{\log_{b_2} n} = \frac{\log_{b_1} b_2 \cdot \log_{b_2} n}{\log_{b_2} n} = \log_{b_1} b_2 = \text{ konstant}, \qquad n \neq 1,$$

und damit

$$O(\log_{b_1} n) = O(\log_{b_1} b_2 \cdot \log_{b_2} n) = O(\log_{b_2} n).$$

Die nachfolgende Tabelle verdeutlicht das Wachstum einiger in den Komplexitätsklassen auftretender Funktionen. Das Alter des Universums wird auf ca. $13,7 \cdot 10^9 a \approx 4,3 \cdot 10^{17} s$ geschätzt.

n	ld n	$\sqrt{n}$	n	$n \cdot$ ld n	n^2	n^3	2^n	$n!$
1	0	1	1	0	1	1	2	1
2	1	$\approx 1,4$	2	2	4	8	4	2
4	2	2	4	8	16	64	16	24
8	3	$\approx 2,8$	8	24	64	512	256	40320
16	4	4	16	64	256	4096	65536	$\approx 2,1 \cdot 10^{13}$
32	5	$\approx 5,6$	32	160	1024	32768	$\approx 4,3 \cdot 10^9$	$\approx 2,6 \cdot 10^{35}$

Definition 2.4 (Groß-Omega)
Es seien $f, g : \mathbb{N} \to \mathbb{R}_0^+$ Funktionen.
$\Omega(g) := \{f \mid$ *es gibt ein $c > 0$ und ein $n_0 \in \mathbb{N}$,*
*so daß für alle $n \geq n_0$ gilt: $f(n) \geq c \cdot g(n)$}$\}$

Auch $\Omega(g)$ ist eine Menge bzw. Klasse von Funktionen. Man schreibt auch hier $f = \Omega(g)$ für $f \in \Omega(g)$. Mittels dieser Notation wird eine asymptotische untere Schranke definiert. Die Abbildung 2.2 zeigt die untere Schranke einer Funktion als Graph.

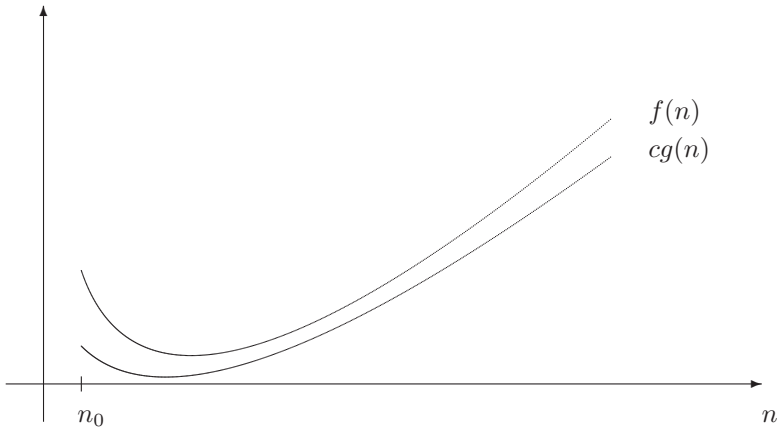


Abbildung 2.2: $f(n) \geq cg(n)$ für alle $n \geq n_0$

Beispiel 2.10
Betrachte die Funktion $f : \mathbb{N} \to \mathbb{N}$ mit $f(n) = 3n^3 + 2n^2 - 4$. Es soll gezeigt werden, dass $f \in \Omega(g)$ mit $g : \mathbb{N} \to \mathbb{N}$ und $g(n) = n^3$. Dafür ist ein $c > 0$ und ein $n_0 \in \mathbb{N}$ zu finden, so dass

$$\forall\, n \geq n_0 : f(n) \geq cn^3$$

gilt.

$$\begin{aligned} f(n) &= 3n^3 + 2n^2 - 4 \\ &\geq 3n^3 - 4 \\ &\geq 2n^3 \qquad\qquad \text{für } n \geq 2 \end{aligned}$$

Also, mit $c = 2$ und $n_0 = 2$ gilt $f(n) \in \Omega(n^3)$.

Leicht kann gezeigt werden, dass darüber hinaus auch $f(n) \in \Omega(n^k)$ für $k \leq 3$ gilt. Für $k = 3$ ist die größte untere Schranke bzgl. der Größenordnung gegeben. □

In Beispiel 2.11 wird gezeigt, dass ein Polynom vom Grad m immer in der Klasse $\Omega(n^m)$ enthalten ist. In Beispiel 2.8 war bereits gezeigt worden, dass ein Polynom vom Grad m auch schon in der Klasse $O(n^m)$ enthalten war.

Beispiel 2.11
Sei

$$f(n) = \sum_{i=0}^{m} a_i n^i \text{ mit } a_m \neq 0$$

ein Polynom. Es gilt:

$$\begin{aligned} f(n) &= a_m n^m + a_{m-1} n^{m-1} + \ldots + a_0 \\ &= a_m n^m \cdot \left(1 + \frac{a_{m-1}}{a_m}\frac{1}{n} + \ldots + \frac{a_0}{a_m}\frac{1}{n^m}\right) \end{aligned}$$

Der Ausdruck in der Klammer konvergiert für $n \to \infty$ gegen 1. Es gibt also ein $n_0 \in \mathbb{N}$, so daß dieser Ausdruck für alle $n \geq n_0$ einen Wert $\geq 1/2$ hat. Für $c := \frac{1}{2}|a_m|$ und $n \geq n_0$ gilt daher:

$$f(n) \geq \frac{1}{2}|a_m| n^m = c \cdot n^m$$

Also gilt $f(n) \geq c \cdot n^k$ für alle $n \geq n_0$ und $k \leq m$. Es ist damit $f(n) \in \Omega(n^k)$ für alle $k \leq m$ und insbesondere $f(n) \in \Omega(n^m)$. □

Beispiel 2.12
Betrachte wieder die Komplexitäten des Algorithmus 1.1:

- Zeit-Komplexität: $B(n) = A(n) = W(n) = n - 1 \in \Omega(n)$
- Speicher-Komplexität: $n + 2 \in \Omega(n)$

□

Bzgl. der Mengen $\Omega(g)$ läßt sich analog zu $O(g)$ eine Klassenhierarchie definieren. Hier ist natürlich jeweils eine größte untere Schranke gesucht.

$$\Omega(1) \supset \Omega(n) \supset \Omega(n^2) \supset \Omega(n^3) \supset \ldots \supset \Omega(n^k) \supset \ldots$$

$$\Omega(1) \supset \Omega(\log n) \supset \Omega(\sqrt{n}) \supset \Omega(n) \supset \Omega(n \cdot \log n) \supset \Omega(n^2)$$

Die betrachteten Beispiele zeigen, daß z. B. für Polynome Funktionen existieren, die sie nach oben und unten asymptotisch beschränken. Diese Beobachtung führt zu einem weiteren Komplexitätsmaß.

Definition 2.5 (Groß-Theta)
Es seien $f, g : \mathbb{N} \to \mathbb{R}_0^+$ *Funktionen.*
$\Theta(g) := \{f \mid f \in O(g) \wedge f \in \Omega(g)\}$

Eine Funktion g ist also asymptotisch obere und untere Schranke einer Funktion f, allerding mit unterschiedlichen Faktoren c_1 und c_2 und unterschiedlichen n_1 und n_2. Die Abbildung 2.3 zeigt die obere und die untere Schranke einer Funktion als Graph.

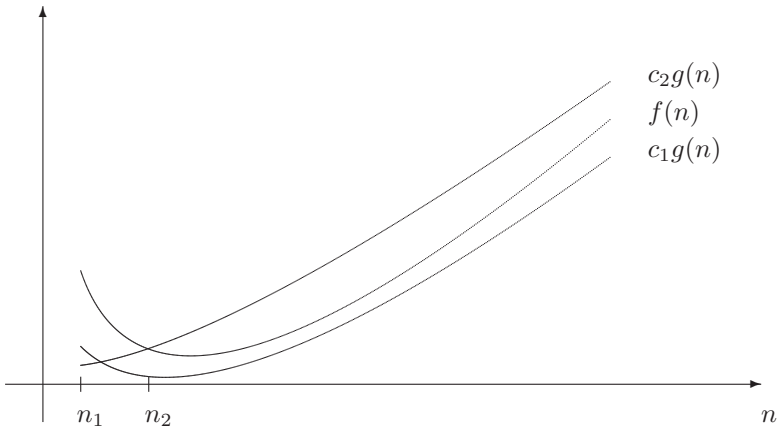


Abbildung 2.3: $c_1 g(n) \leq f(n) \leq c_2 g(n)$ für alle $n \geq n_0 = \max(n_1, n_2)$

Beispiel 2.13
Betrachte die Funktion $f : \mathbb{N} \to \mathbb{N}$ mit $f(n) = 3n^3 + 2n^2 - 4$.
Es gelten $f \in O(n^3)$ (Beispiel 2.7) und $f \in \Omega(n^3)$ (Beispiel 2.10). Also ist $f(n) \in \Theta(n^3)$. □

Beispiel 2.14
Ein Polynom $f(n) = \sum_{i=0}^{m} a_i n^i$ mit $a_m \neq 0$ verläuft also für ein $n_0 \in \mathbb{N}$ und alle $n \geq n_0$ in einer durch das Intervall $[c_1 n^m, c_2 n^m]$ begrenzten Umgebung
mit $c_1 := \frac{1}{2}|a_m|$ und $c_2 := |a_m| \cdot \left(1 + \frac{|a_{m-1}|}{|a_m|} + \ldots + \frac{|a_0|}{|a_m|}\right)$,
und es gilt $f(n) \in \Theta(n^m)$. □

Beispiel 2.15
Betrachte wieder die Komplexitäten des Algorithmus 1.1:

- Zeit-Komplexität: $B(n) = A(n) = W(n) = n - 1 \in \Theta(n)$
- Speicher-Komplexität: $n + 2 \in \Theta(n)$

□

Entsprechend einem modularen Aufbau von Algorithmen erfolgt auch die Bestimmung der Laufzeit eines Algorithmus zunächst für die einzelnen Prozeduren, um diese dann zur Gesamtlaufzeit des Algorithmus zusammenzufügen. Dazu sind einige Eigenschaften und Rechenregeln für die Komplexitätsklassen sehr hilfreich. Satz 2.2 führt einige davon für die O-Notation auf. Sie gelten aber auch entsprechend für die Ω und Θ-Notationen (Sätze 2.3 und 2.4). Auf die Beweise sei hier verzichtet.

Satz 2.2 (Groß-O Eigenschaften)
Es seien $g, g_1, g_2 : \mathbb{N} \to \mathbb{R}_0^+$ Funktionen und $k \in \mathbb{R}^+$ eine Konstante.

- $k \cdot O(g) = O(k \cdot g) = O(g)$
- $O(g_1) \cdot O(g_2) = O(g_1 \cdot g_2)$
- $O(g_1) + O(g_2) = O(g_1 + g_2) = O(\max(g_1, g_2))$

Satz 2.3 (Groß-Ω Eigenschaften)
Es seien $g, g_1, g_2 : \mathbb{N} \to \mathbb{R}_0^+$ Funktionen und $k \in \mathbb{R}^+$ eine Konstante.

- $k \cdot \Omega(g) = \Omega(k \cdot g) = \Omega(g)$
- $\Omega(g_1) \cdot \Omega(g_2) = \Omega(g_1 \cdot g_2)$
- $\Omega(g_1) + \Omega(g_2) = \Omega(g_1 + g_2) = \Omega(\max(g_1, g_2))$

Satz 2.4 (Groß-Θ Eigenschaften)
Es seien $g, g_1, g_2 : \mathbb{N} \to \mathbb{R}_0^+$ Funktionen und $k \in \mathbb{R}^+$ eine Konstante.

- $k \cdot \Theta(g) = \Theta(k \cdot g) = \Theta(g)$
- $\Theta(g_1) \cdot \Theta(g_2) = \Theta(g_1 \cdot g_2)$
- $\Theta(g_1) + \Theta(g_2) = \Theta(g_1 + g_2) = \Theta(\max(g_1, g_2))$

2.4 Optimalität

In Abschnitt 1.2 war der Algorithmus 1.1 zur Bestimmung des Maximums einer Zahlenfolge vorgestellt worden, und es war dann gezeigt worden, dass dieser Algorithmus eine lineare Komplexität aufweist. Es ist nun die Frage zu klären, ob es einen effizienteren Algorithmus gibt, der das gleiche Problem löst.

Diese Frage kann in diesem Fall sehr leicht beantwortet werden. Algorithmus 1.1 nimmt das erste Element der Zahlenfolge als vorläufiges Maximum und vergleicht dann jedes der nachfolgenden $n-1$ Elemente mit dem vorläufigen Maximum. Wann immer ein größeres Element auftritt, wird dieses das neue vorläufige Maximum. Es sind also $n-1$ Vergleiche notwendig, um das Maximum zu bestimmen.

Sei nun angenommen, es gebe einen Algorithmus, der weniger als $n-1$ Vergleiche benötige, um das Maximum zu bestimmen. Dann kann das als erstes als vorläufiges Maximum gewählte Element höchstens mit $n-2$ weiteren Elementen der Zahlenfolge verglichen werden. Ein Element wird also gar nicht untersucht. Wenn nun aber gerade dieses Element das Maximum der Zahlenfolge ist, kann es nicht gefunden werden, und der Algorithmus liefert nicht das Maximum.

Es gibt also keinen Algorithmus, der das Maximum einer Zahlenfolge schneller als in linearer Zeit bestimmt. Der Algorithmus 1.1 ist also optimal.

Ein Algorithmus ist *optimal*, wenn es keinen anderen Algorithmus gibt, der das gleiche Problem im schlechtesten Fall auf eine effizientere Weise löst. Dabei sind mit „Algorithmen, die das gleiche Problem lösen“ alle im umfassensten Sinn gemeint, also auch die, die noch gar nicht bekannt sind. Optimalität bedeuted also nicht der effizienteste unter den bekannten Algorithmen, sondern der effizienteste mögliche Algorithmus überhaupt.

Kapitel 3

Rekursion

3.1 Lineare Rekursion

Eine *rekursive* Funktion ist eine Funktion, die durch Rückführung auf sich selbst definiert wird. Betrachte z. B. die rekursive Definition der Fakultät (Beispiel 3.1).

Beispiel 3.1
Es sei $n \in \mathbb{N}_0$.

$$n! = \begin{cases} 1, & \text{falls } n = 0 \\ n \cdot (n-1)!, & \text{falls } n \geq 1 \end{cases}$$

Die Berechnung der Funktionswerte solcher Funktionen kann direkt in rekursive Algorithmen überführt werden, das sind Algorithmen, die ebenso durch Rückführung auf sich selbst mit kleineren Eingabewerten spezifiziert werden (Algorithmus 3.1). Jeder Aufruf hat dabei zur Laufzeit seinen eigenen Speicherplatzbereich für die lokalen Variablen.

Algorithmus 3.1 (Fakultät)
Eingabe: $n \in \mathbb{N}_0$
Ausgabe: $n!$

```
FAKULTÄT(n)
1  if n = 0
2     then return 1
3     else return n·FAKULTÄT(n − 1)
```

Beispiel 3.2
Berechnung von 4!:

Rekursions-tiefe	n	rekursiver Aufruf / berechneter Wert
0	4	$fakultät(4)$
1	4	$4 \cdot fakultät(3)$
2	3	$3 \cdot fakultät(2)$
3	2	$2 \cdot fakultät(1)$
4	1	$1 \cdot fakultät(0)$
5	0	1
4	1	$1 \cdot 1 = 1$
3	2	$2 \cdot 1 = 2$
2	3	$3 \cdot 2 = 6$
1	4	$4 \cdot 6 = 24$
0		24

Im Folgenden ist die Komplexität des Algorithmus zu analysieren. Dazu sind wieder die grundlegenden Anweisungen zu spezifizieren und die Anzahl ihrer Ausführungen zu bestimmen. Neben einem nicht-rekursiven Anteil ist aber auch der Aufwand zu berücksichtigen, der durch die rekursiven Aufrufe entsteht. Die Analyse ist zwischen der Rekursionsbasis und den Rekursionsaufrufen zu differenzieren. Die Eingabegröße ist hier das Argument n.

I. Für den Fall $n = 0$ (Rekursionsbasis) ist nur der Funktionswert 1 zu liefern. Dies ist hier die grundlegende Anweisung, die einmal ausgeführt wird.

II. Für den Fall $n \geq 1$ (Rekursionsaufruf) ist eine Multiplikation, eine Subtraktion und ein rekursiver Aufruf des Algorithmus auszuführen. Die grundlegende Anweisung ist hier die Multiplikation, die je rekursivem Aufruf einmal auszuführen ist. Die Subtraktionen können unberücksichtigt bleiben, da ihre Anzahl proportional zu der Anzahl der Multiplikationen ist. Bleibt die Komplexität des Algorithmus mit einem um eins verringertem Argument zu bestimmen.

Sei nun $T(n)$ die Komplexität des Algorithmus mit dem Argument n. Dann ist auch diese Funktion mit der obigen Fallunterscheidung rekursiv definiert. Für die Rekursionsbasis ergibt sich nur ein nicht-rekursiver Anteil, für den Rekursionsaufruf ergibt sich ein nicht-rekursiver und ein rekursiver Anteil:

$$T(n) = \begin{cases} 1, & \text{falls } n = 0 \\ T(n-1) + 1, & \text{falls } n \geq 1 \end{cases}$$

Diese Rekursion kann für $n \geq 1$ iterativ über die Rekursionstiefe aufgelöst werden:

$$\begin{aligned} T(n) &= T(n-1) + 1 \\ &= T(n-2) + 1 + 1 \\ &= T(n-3) + 1 + 1 + 1 \\ &= \dots \\ &= T(0) + n \\ &= 1 + n \in \Theta(n) \end{aligned}$$

Diese lineare Komplexität gilt für den besten, den schlechtesten und damit auch für den mittleren Fall.

$$B(n) = A(n) = W(n) = n + 1 \in \Theta(n)$$

Es sei hier darauf hingewiesen, dass die Fakultätsfunktion für gar nicht allzugroße n sehr große Werte liefert. Die damit verbundenen notwendigen Operationen (z. B. Multiplikation) sind dann nicht mehr in konstanter Zeit auszuführen, sondern in Abhängigkeit von der Stellenanzahl von n. Die hier berechnete lineare Komplexität ist also mit einiger Vorsicht zu betrachten. Einen Eindruck von dem Wachstum der Fakultätsfunktion gibt die folgende Tabelle:

n	$\mathrm{ld}\, n$	n^2	2^n	$n!$
1	0	1	2	1
2	1	4	4	2
4	2	16	16	24
8	3	64	256	40320
16	4	256	65536	$\approx 20,9 \cdot 10^{12}$

Rekursionsgleichungen, wie sie hier betrachtet wurden, heißen *lineare Rekursionsgleichungen*. Für die hier besprochene Anwendung haben sie etwas verallgemeinert die folgende Form:

$$T(n) = b \cdot T(n-c) + f(n)$$

Dabei ist $b \geq 1$ die Anzahl der rekursiv zu lösenden Unterproblem der Größe $n - c$ mit $c > 0$, auch Verzweigungungsfaktor genannt, und $f(n)$ ist der nicht-rekursive Anteil der Komplexität. Mit Satz 3.1 ist die Lösung einer solchen Rekursionsgleichung gegeben. Auf einen Beweis sei hier verzichtet.

Satz 3.1
Sei

$$T(n) = \begin{cases} \textit{konstant}, & \textit{falls } n = 0 \\ b\, T(n-c) + f(n), & \textit{falls } n \geq 1 \end{cases}$$

eine Rekursionsgleichung mit dem Verzweigungsfaktor $b \geq 1$, $c > 0$, und der Rekursionsbasis $T(0) \in \Theta(1)$. Die Lösung von $T(n)$ hat dann für $n/c \in \mathbb{N}_0$ die folgende Form.

$$T(n) = \sum_{d=0}^{n/c} b^d f(n - cd)$$

Beispiel 3.3
Betrachte nochmals die Fakultätsfunktion. Ihre Komplexität $T(n)$ kann auch über den Satz 3.1 bestimmt werden: Mit $b = 1$, $c = 1$, $f(n) = 1$ und $T(0) = 1 \in \Theta(1)$ geht

$$T(n) = bT(n-c) + f(n) \text{ für } n \geq 1 \text{ und } T(0) \in \Theta(1)$$

über in

$$T(n) = T(n-1) + 1 \text{ für } n \geq 1 \text{ und } T(0) = 1.$$

Damit ergibt sich dann die Lösung wie folgt:

$$\begin{aligned} T(n) &= \sum_{d=0}^{n/c} b^d f(n - cd) \\ &= \sum_{d=0}^{n} 1^d f(n - d) \\ &= \sum_{d=0}^{n} 1 \\ &= n + 1 \in \Theta(n) \end{aligned}$$

□

Zu beachten ist, dass Satz 3.1 nur für $n/c \in \mathbb{N}_0$ und für die Rekursionsbasis $r = 0$ gilt. Falls $n/c \notin \mathbb{N}_0$ gilt und die Rekursionsbasis r für $0 < r < c$ mit $T(r) \in \Theta(1)$ erreicht wird, ergibt sich in Satz 3.1 $\lfloor n/c \rfloor$ als oberer Summationsindex. Für z.B. $r = c = 1$ geht die Summation dann nur bis $n - 1$.

3.2 Divide-and-Conquer

Eine große Zahl von Problemen läßt sich dadurch lösen, daß das Problem so lange in Teilprobleme aufgeteilt wird bis diese einfach lösbar werden, und diese Lösungen dann zur Gesamtlösung zusammengefügt werden. Algorithmen, die eine solche Lösungsstrategie aufgreifen, werden *divide-and-conquer* Algorithmen genannt. Sie weisen eine *rekursive* Struktur auf, d. h. ein Algorithmus oder eine darin enthaltene Funktion ruft sich selbst direkt oder indirekt auf.

- **Divide:** Das Problem wird zunächst rekursiv in Unterprobleme zerlegt.
- **Conquer:** Sind die Unterprobleme genügend klein, so können sie gelöst werden, sind also „erobert“ oder „beherrscht“.
- **Combine:** Die Lösungen der Unterprobleme werden zur Lösung des Ausgangsproblems zusammengefügt.

Eine Formulierung in Pseudocode ist mit Algorithmus 3.2 gegeben.

Algorithmus 3.2 (Divide-and-Conquer)
Eingabe: p
Ausgabe: l

```
PROBLEM(p)
1  if |p| = 1
2     then löse Problem direkt, die Lösung sei l
3     else zerlege p in Teilprobleme p_1 und p_2
4          l_1 ←PROBLEM(p_1)
5          l_2 ←PROBLEM(p_2)
6          setze die Lösung l aus l_1 und l_2 zusammen
7  return l
```

Ein typisches Beispiel eines *divide-and-conquer* Verfahrens ist der Algorithmus *Merge-Sort* (Algorithmus 3.3), der in Abschnitt 4.6 besprochen wird. Die Prozedur *Merge-Sort* halbiert über die rekursiven Aufrufe das zu sortierende Zahlenfeld sukzessive bis zur Rekursionsbasis, der Feldlänge 1. Die Prozedur *Merge* mischt bereits sortierte Teilfelder nicht-rekursiv.

Algorithmus 3.3 (Merge-Sort)
Eingabe: $A = (a_1, \ldots, a_n)$ mit $a_i, i, n \in \mathbb{N}, 1 \leq i \leq n$ und $p = 1, r = n$
Ausgabe: $A = (a_{\pi(1)}, \ldots, a_{\pi(n)})$ mit $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$

```
MERGE-SORT(A, p, r)
1  if p < r
2     then q ← ⌊(p + r)/2⌋
3          MERGE-SORT(A, p, q)
4          MERGE-SORT(A, q + 1, r)
5          MERGE(A, p, q, r)
```

Die Eingabegröße für diesen Algorithmus ist die Anzahl der zu sortierenden Zahlen, also n. Der Einfachheit halber sei $n = 2^k$ für ein $k \geq 0$ angenommen, ansonsten müßten als Eingabegrößen der Teilprobleme $\lfloor n/2 \rfloor$ und $\lceil n/2 \rceil$ betrachtet werden. Die grundlegende Anweisung ist das Mischen mittels der Prozedur *Merge*. Diese habe die Komplexität $f(n) \in \Theta(n)$. Der Aufwand dieses Algorithmus $T(n)$ kann dann durch die Rekursionsgleichung

$$T(n) = 2T(n/2) + f(n)$$

für $n \geq 2$ angegeben werden.

Im Fall $n = 1$, also $p \not< r$, findet kein Mischen mehr statt, also gilt für die Rekursionsbasis $T(1) = 0$.

Diese Rekursionsgleichung kann nicht mehr mit Satz 3.1 gelöst werden. Dort werden die Argumente der rekursiven Aufrufe subtraktiv verringert, hier multiplikativ mit dem Faktor 1/2.

Die Rekursionsgleichung wird nun zunächst wieder durch sukzessives Einsetzen gelöst. Des Weiteren gibt es aber auch hier eine Formel zur Lösung solcher Rekursionsgleichungen, dem so genannten *Master-Theorem* (Satz 3.2). In Abschnitt 4.6 wird für das Mischen die Komplexität $f(n) = n \in \Theta(n)$ hergeleitet. Werde diese im Folgenden bereits verwendet.

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n \\
&= 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n \\
&= 2^2 T\left(\frac{n}{2^2}\right) + 2n \\
&\vdots \\
&= 2^k T\left(\frac{n}{2^k}\right) + kn \text{ für } 1 \leq k \leq \operatorname{ld} n \\
&\vdots \\
&= nT\left(\frac{n}{n}\right) + n\operatorname{ld} n \\
&= nT(1) + n\operatorname{ld} n \\
&= n\operatorname{ld} n \in \Theta(n \log n)
\end{aligned}$$

In Abbildung 3.1 ist der Rekursionsbaum dargestellt, der die Aufruffolge des Algorithmus bzgl. der Eingabegröße n wiedergibt. Die Summe der Komplexitäten der einzelnen Aufrufe ergibt die Gesamtkomplexität.

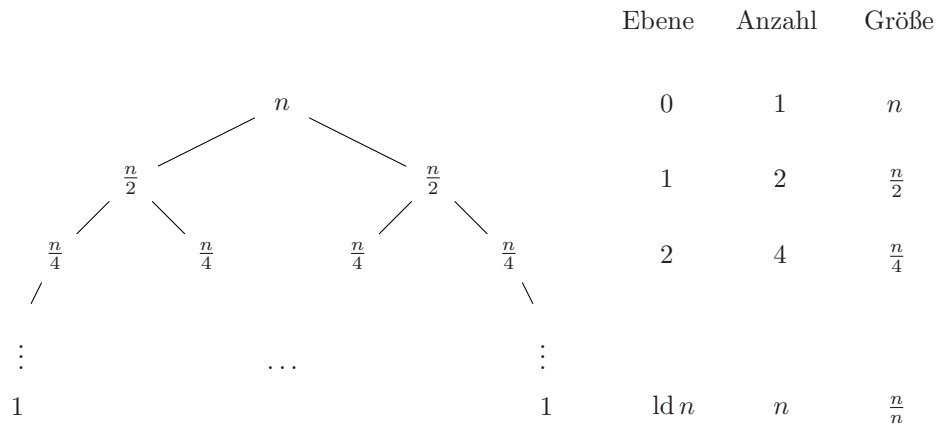


Abbildung 3.1: Rekursionsbaum des Algorithmus *Merge-Sort* mit der Eingabegröße n (n sei Zweierpotenz).

Nun zu dem *Master-Theorem* (Satz 3.2). Damit kann die Lösung von Rekursionsgleichungen, wie sie bei *divide-and-conquer* Algorithmen auftreten, bestimmt werden. Auch hier wird auf einen Beweis verzichtet.

Es gilt für Algorithmen, die die Eingabegröße n in m Teilprobleme der Größe n/c zerlegen, $c > 1$, diese lösen und zur Gesamtlösung zusammenfügen. Dabei sei $f(n) = \Theta(n^k)$ der nicht-rekursive Anteil des Aufwands des Algorithmus. Der Aufwand für die Eingabegröße 1 ist konstant, also gilt für die Rekursionsbasis $T(1) = \Theta(1)$. Auf das Erzwingen der Ganzzahligkeit der Teilproblemgröße kann hier verzichtet werden, da dies sich bei asymptotischen Abschätzungen nicht auswirkt.

Satz 3.2 (Master-Theorem)
Sei

$$T(n) = \begin{cases} \textit{konstant}, & \textit{falls } n = 1 \\ b\,T\,(n/c) + f(n), & \textit{falls } n \geq 2 \end{cases}$$

eine Rekursionsgleichung mit dem Verzweigungsfaktor $b \geq 1$, $c > 1$, *der Rekursionsbasis* $T(1) \in \Theta(1)$ *und* $f(n) \in \Theta(n^k)$.

Die Lösung von $T(n)$ *hat dann in asymptotischer Notation die folgende Form:*

$$T(n) \in \begin{cases} \Theta(n^k), & \textit{falls} \quad b(1/c)^k < 1 \\ \Theta(n^k \log n), & \textit{falls} \quad b(1/c)^k = 1 \\ \Theta(n^a), & \textit{falls} \quad b(1/c)^k > 1 \end{cases}$$

Hierbei ist $a = \ln b / \ln c = \log_c b > k$.

Beispiele 3.4

1. $T(n) = 8T(n/3) + n^2$ mit $T(1) \in \Theta(1)$.

   Es gilt $8\left(\frac{1}{3}\right)^2 = \frac{8}{9} < 1$, also $T(n) \in \Theta(n^2)$.

2. $T(n) = 9T(n/3) + n^2$

   Es gilt $9\left(\frac{1}{3}\right)^2 = 1$, also $T(n) \in \Theta(n^2 \log n)$.

3. $T(n) = 10T(n/3) + n^2$

   Es gilt $10\left(\frac{1}{3}\right)^2 = \frac{10}{9} > 1$, also $T(n) \in \Theta(n^a) \approx \Theta(n^2 \cdot \sqrt[10]{n})$
   mit $a = \frac{\ln 10}{\ln 3} = \log_3 10 \approx 2,096 \approx 2,1$.

□

Beispiel 3.5
Betrachte den Algorithmus *Merge-Sort*(Algorithmus 3.3):

$$T(n) = \begin{cases} \Theta(1), & \text{falls } n = 1 \\ 2T(n/2) + \Theta(n), & \text{falls } n > 1 \end{cases}$$

Wegen $2\left(\frac{1}{2}\right)^1 = 1$ ergibt sich mit dem Master-Theorem

$$T(n) \in \Theta(n \log n).$$

□

Kapitel 4

Suchen und Sortieren

4.1 Problemspezifikation

In vielen Anwendungen werden in *Datensätzen* strukturierte Daten gespeichert und verarbeitet. Diese Datensätze bestehen häufig aus einem so genannten *Schlüssel* und den eigentlichen Daten. Der Schlüssel wird zur eindeutigen Identifizierung und auch als Ordnungskriterium benutzt. Betrachte z. B. einen Datenbestand von Studierenden mit der Matrikelnummer als Schlüssel (Beispiel 4.1).

Beispiel 4.1

Matr.-Nr.	Name	Vorname	Wohnort	Studiengang	Sem.
3591	Meier	Beate	Wolfenbüttel	Informatik	3
2319	Linnemann	Herbert	Braunschweig	Elektrotechnik	1
5753	Anderson	Leonie	Salzgitter	Informatik	5
4711	Markmann	Heinz	Braunschweig	Informatik	3
1972	Meier	Beate	Braunschweig	Mediendesign	2
2941	Deneke	Peter	Wolfenbüttel	Elektrotechnik	7

Anfragen nach bestimmten Datensätzen werden nun in vielen Fällen über den Schlüssel gestellt. Es muss also der Datensatz zu einem gegebenen Schlüssel gesucht werden. Oder es wird der Datensatz mit dem kleinsten bzw. größten Schlüssel gesucht. Es kann zur effizienteren Bearbeitung solcher Anfragen sehr sinnvoll sein, die Datensätze auf- bzw. absteigend nach ihren Schlüsseln zu sortieren.

Hier sollen nun Such- und Sortier-Algorithmen betrachtet werden. Dazu wird eine Reduktion auf das Kernproblem vorgenommen. Die Datensätze bestehen ausschließlich aus ihrem Schlüssel, und ein Schlüssel ist eine natürliche Zahl.

- **Suchproblem:**

  Gegeben ist eine Folge von Zahlen $A = (a_1, \ldots, a_n)$ mit $a_i, i, n \in \mathbb{N}, 1 \leq i \leq n$ und ein $x \in \mathbb{N}$.

  Gesucht ist ein Index i mit $a_i = x$, falls ein solches $A[i]$ überhaupt existiert.

- **Sortierproblem:**

  Gegeben ist eine Folge von Zahlen $A = (a_1, \ldots, a_n)$ mit $a_i, i, n \in \mathbb{N}, 1 \leq i \leq n$.

  Gesucht ist eine Permutation der Zahlenfolge $\pi : \{1, \ldots, n\} \to \{1,\ldots,n\}$, also eine Umordnung zu $(a_{\pi(1)}, \ldots, a_{\pi(n)})$, so dass $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$.

Beispiel 4.2
Die Zahlenfolge

$$A = (a_1, a_2, a_3, a_4, a_5, a_6) = (3591, 2319, 5753, 4711, 1972, 2941)$$

sei gegeben.

Gesucht werde der Wert $x = 4711$. Wegen $a_4 = 4711$ wird der Index 4 geliefert.

Für die Permutation

$$\pi(1) = 5, \pi(2) = 2, \pi(3) = 6, \pi(4) = 1, \pi(5) = 4, \pi(6) = 3$$

ergibt sich die aufsteigend sortierte Zahlenfolge

$$A = (a_{\pi(1)}, a_{\pi(2)}, a_{\pi(3)}, a_{\pi(4)}, a_{\pi(5)}, a_{\pi(6)}) = (1972, 2319, 2941, 3591, 4711, 5753).$$

□

Die Eingabegröße für Such- und Sortieralgorithmen ist die Anzahl der Elemente, in denen gesucht bzw. die sortiert werden sollen. Die grundlegende Anweisung zur Bestimmung der Zeit-Komplexität ist in beiden Fällen die Anzahl der notwendigen Schlüsselvergleiche. Bei Sortieralgorithmen kann zusätzlich die Anzahl der notwendigen Vertauschungen (Zuweisungen) hinzukommen.

Bei der Speicher-Komplexität ist insbesondere bei Sortierverfahren von Interesse, ob der Algorithmus *am Platz* sortiert. Es gibt Sortieralgorithmen, die eine bestimmte Anzahl an Elementen

außerhalb des zu sortierenden Feldes zwischenspeichern müssen. Ein weiteres Merkmal von Sortieralgorithmen ist durch sein Verfahren bei gleichen Schlüsseln gegeben. Bleibt deren relative Anordnung zueinander während des Sortiervorgangs bestehen, so spricht man von einem *stabilen* Verfahren.

4.2 Sequentielles Suchen

In einem unsortierten Feld wird ein vorgegebener Wert gesucht, indem die Elemente des Feldes sequentiell überprüft werden. Im positiven Fall wird die Position des Elementes in dem Feld geliefert, im negativen Fall die 0.

Algorithmus 4.1 (Sequentielles Suchen, unsortiert)
Eingabe: $A = (a_1, \ldots, a_n)$ mit $a_i, i, n \in \mathbb{N}, 1 \leq i \leq n, x \in \mathbb{N}$
Ausgabe: $i \in \{0, 1, \ldots, n\}$ mit $1 \leq i \leq n$ gefunden, $i = 0$ nicht gefunden

```
SEQ-SUCHEN-UNSORT(A, x)
1  i ← 1
2  while (i ≤ n) ∧ (A[i] ≠ x)
3     do i ← i + 1
4  if i ≤ n
5     then return i
6     else return 0
```

Beispiel 4.3
Betrachte die Folge $A = (14, 4, 2, 12, 15, 7, 19, 11)$, also $n = 8$.
Werde der Wert $x = 7$ gesucht :

i	A	$A[i] \neq x$
1	$(14, 4, 2, 12, 15, 7, 19, 11)$	$14 \neq 7$
2	$(14, 4, 2, 12, 15, 7, 19, 11)$	$4 \neq 7$
3	$(14, 4, 2, 12, 15, 7, 19, 11)$	$2 \neq 7$
4	$(14, 4, 2, 12, 15, 7, 19, 11)$	$12 \neq 7$
5	$(14, 4, 2, 12, 15, 7, 19, 11)$	$15 \neq 7$
6	$(14, 4, 2, 12, 15, 7, 19, 11)$	$7 = 7$

Wegen $A[6] = 7$ wird $i = 6$ geliefert. □

Zur Verifikation des Algorithmus sei nur soviel gesagt: Er terminiert offensichtlich, denn i wird beginnend mit 1 maximal bis n inkrementiert, und er liefert auch das gesuchte Element, falls es denn in dem Feld enthalten ist.

Für die Komplexitätsanalyse ist die Anzahl der Vergleiche $A[i] \neq x$ als grundlegende Anweisung (Zeile 2) zu betrachten. Im besten Fall ist das gesuchte Element das erste in dem Feld. Dann wird der Vergleich nur einmal ausgeführt. Im schlechtesten Fall ist das gesuchte Element nicht in dem Feld. Das wird erst festgestellt, wenn alle Elemente verglichen worden sind, d. h. es hat n Vergleiche gegeben.

$$\begin{aligned} B(n) &= 1 \in \Theta(1) \\ W(n) &= n \in \Theta(n) \end{aligned}$$

Die mittlere Komplexität hängt nicht nur davon ab, an welcher Position sich das gesuchte Element in dem Feld befindet, sondern auch von der Wahrscheinlichkeit, mit der es überhaupt in dem Feld steht.

Seien die Elemente a_i von A aus dem Intervall $[1, m] \subset \mathbb{N}$, $1 \leq i \leq n \leq m$. Kein Element sei in A mehrfach vorhanden. Bei einer Gleichverteilung der Elemente ist dann die Wahrscheinlichkeit, dass der gesuchte Wert x als i-tes Element in A enthalten ist, $1/m$. Die Wahrscheinlichkeit, nicht in A enthalten zu sein, beträgt $1 - n/m$.

Befindet sich der gesuchte Wert x an der i-ten Position in A, dann wird es beim i-ten Vergleich gefunden. Die Komplexität im mittleren Fall ergibt sich damit wie folgt:

$$\begin{aligned} A(n) &= \frac{1}{m}(1+2+\ldots+n) + \frac{m-n}{m}n \\ &= \frac{n(n+1)}{2m} + \frac{(m-n)n}{m} \\ &= n - \frac{n^2}{2m} + \frac{n}{2m} \\ &\geq n - \frac{n^2}{2n} \\ &= \frac{n}{2} \in \Omega(n) \end{aligned}$$

Im Mittel sind also je nach Verhältnis von n und m mindestens $n/2$ Vergleiche notwendig. Zusammen mit der Komplexität im schlechtesten Fall kann die Komplexität im mittleren Fall nach oben mit $O(n)$ abgeschätzt werden, und es ergibt sich insgesamt

$$A(n) \in \Theta(n).$$

Die Komplexität im mittleren Fall ist also von der gleichen Größenordnung wie die Komplexität im schlechtesten Fall.

Um das Verhältnis dieser beiden Komplexitäten dennoch etwas zu differenzieren, seien hierin zwei Fälle betrachtet:

I. $n = m$:

Das gesuchte Element ist dann sicher im Feld enthalten, und es gilt

$$A(n) = \frac{n+1}{2} \ .$$

II. $2n = m$:

Das gesuchte Element ist dann mit Wahrscheinlichkeit $1/2$ im Feld enthalten, und es gilt

$$A(n) = \frac{3n+1}{4} \ .$$

Je größer m gegenüber n, desto mehr nähert sich die Anzahl der Vergleiche im mittleren Fall an die Anzahl im schlechtesten Fall.

Was verändert sich nun, wenn das Zahlenfeld aufsteigend sortiert ist? Die sequentielle Suche kann vorzeitig abgebrochen werden, wenn das gesuchte Element nicht in dem Feld enthalten ist. Zusätzlich sei angenommen, die Zahlen seien paarweise verschieden. Betrachte dazu Algorithmus 4.2.

Algorithmus 4.2 (Sequentielles Suchen, sortiert)
Eingabe: $A = (a_1, \ldots, a_n)$ mit $a_i, i, n \in \mathbb{N}, 1 \leq i \leq n, x \in \mathbb{N}$,
$a_1 < a_2 < \ldots < a_n$
Ausgabe: $i \in \{0, \ldots, n\}$ mit $1 \leq i \leq n$ gefunden, $i = 0$ nicht gefunden

```
SEQ-SUCHEN-SORT(A, x)
1  i ← 1
2  while (i ≤ n) ∧ (A[i] < x)
3     do i ← i + 1
4  if (i ≤ n) ∧ (A[i] = x)
5     then return i
6     else return 0
```

Beispiel 4.4
Betrachte die Folge $A = (2, 4, 7, 11, 12, 14, 15, 19)$, also $n = 8$.
Werde der Wert $x = 8$ gesucht :

i	A	$A[i] < x$
1	$(2, 4, 7, 11, 12, 14, 15, 19)$	$2 < 8$
2	$(2, 4, 7, 11, 12, 14, 15, 19)$	$4 < 8$
3	$(2, 4, 7, 11, 12, 14, 15, 19)$	$7 < 8$
4	$2, 4, 7, 11, 12, 14, 15, 19)$	$11 \not< 8$

Wegen $A[4] \neq 8$ wird $i = 0$ geliefert. □

Die Komplexitäten im besten und im schlechtesten Fall sind gleich dem unsortierten Zahlenfeld. Im besten Fall ist das gesuchte Element das erste in dem Feld. Im schlechtesten Fall ist das gesuchte Element nicht in dem Feld und es ist größer als das letzte Element.

$$\begin{aligned} B(n) &= 1 \in \Theta(1) \\ W(n) &= n \in \Theta(n) \end{aligned}$$

Auf eine Herleitung der Komplexität im mittleren Fall soll hier verzichtet werden. Sie wird weniger Vergleiche erfordern als im unsortierten Zahlenfeld, da das Nichtenthaltensein aufgrund der Sortierung früher erkannt wird. Die Anzahl liegt unabhängig vom Verhältnis von n und m bei $n/2$. Also es gilt

$$A(n) \approx \frac{n}{2} \in \Theta(n).$$

4.3 Binäres Suchen

Es ist wieder ein vorgegebener Wert in einem Feld zu suchen. Von den Elementen des Feldes ist allerdings bekannt, dass sie paarweise verschieden und aufsteigend sortiert sind. Beim *Binären Suchen* (Algorithmus 4.3) wird das Zahlenfeld sukzessive halbiert und nur die Hälfte weiter betrachtet, in dem der gesuchte Wert aufgrund der Sortierung liegen kann.

Algorithmus 4.3 (Binäres Suchen)
Eingabe: $A = (a_1, \ldots, a_n)$ mit $a_i, i, n \in \mathbb{N}, 1 \leq i \leq n, x \in \mathbb{N}$,
$a_1 < a_2 < \ldots < a_n$
Ausgabe: $i \in \{0, 1, \ldots, n\}$ mit $1 \leq i \leq n$ gefunden, $i = 0$ nicht gefunden

```
BINÄRES SUCHEN(A, x)
 1  l ← 1
 2  r ← n + 1
 3  while l < r
 4     do i ← ⌊(l + r)/2⌋
 5        if A[i] = x
 6           then return i
 7        if A[i] < x
 8           then l ← i + 1
 9           else r ← i
10  return 0
```

Beispiel 4.5
Betrachte die Folge $A = (2, 4, 7, 11, 12, 14, 15, 19)$, also $n = 8$.
Werde der Wert $x = 7$ gesucht:

A	i	$A[i] < x$	l	r	$l < r$
$(2, 4, 7, 11, 12, 14, 15, 19)$			1	9	$1 < 9$
$(2, 4, 7, 11, 12, 14, 15, 19)$	5	$12 \not< 7$	1	5	$1 < 5$
$(2, 4, 7, 11, 12, 14, 15, 19)$	3	$7 = 7$			

Ausgabe: 3

Werde der Wert $x = 8$ gesucht:

A	i	$A[i] < x$	l	r	$l < r$
$(2, 4, 7, 11, 12, 14, 15, 19)$			1	9	$1 < 9$
$(2, 4, 7, 11, 12, 14, 15, 19)$	5	$12 \not< 8$	1	5	$1 < 5$
$(2, 4, 7, 11, 12, 14, 15, 19)$	3	$7 < 8$	4	5	$4 < 5$
$(2, 4, 7, 11, 12, 14, 15, 19)$	4	$11 \not< 8$	4	4	$4 \not< 4$

Ausgabe: 0 □

Die Komplexität des Algorithmus 4.3 ist wieder über die Anzahl der auszuführenden Vergleiche zu bestimmen. Das sind hier die Zeilen 5 und 7, die aber nur als ein Vergleich mit einer Dreifachverzweigung gezählt werden.

Im besten Fall wird bereits beim ersten Vergleich der gesuchte Wert gefunden. Ansonsten wird das Feld sukzessive halbiert und das gesuchte Element mit dem mittleren Element der jeweilig relevanten Feldhälfte verglichen. Der schlechteste Fall tritt wieder dann ein, wenn das gesuchte Element nicht in dem Feld enthalten ist. Nach spätestens $\lfloor \mathrm{ld}\, n \rfloor + 1$ Halbierungen und ebensovielen Vergleichen ergibt sich eine Feldlänge von 1, d. h. es gilt $l = r$, und die Schleife terminiert. Die Komplexität im schlechtesten Fall ist also logarithmisch.

$$\begin{aligned} B(n) &= 1 && \in \Theta(1) \\ W(n) &= \lfloor \mathrm{ld}\, n \rfloor + 1 && \in \Theta(\log n) \end{aligned}$$

Auf die Herleitung der Komplexität im mittleren Fall soll hier verzichtet werden. Es gilt:

$$A(n) \in \Theta(\log n)$$

Der Effizienzgewinn gegenüber dem sequentiellen Suchen ist nur mit der vorausgesetzten Sortierung möglich. Eine vorausgehende Sortierung eines Zahlenfeldes würde aber diesen Gewinn mehr als aufzehren, wie die nächsten Abschnitte zeigen werden. Das binäre Suchen ist aber dem sequentiellen vorzuziehen, wenn das Zahlenfeld bereits sortiert vorliegt.

4.4 Suchen und Optimalität

In den vorangehenden Abschnitten wurden Suchalgorithmen betrachtet, die auf dem Vergleich des zu suchenden Elements mit den Elementen des Zahlenfeldes beruhen. Gegenüber den unsortierten Zahlenfeldern erwies sich die Sortierung des Zahlenfeldes als eine Verbesserung. Daraus ergibt sich unmittelbar die Frage, ob es nicht noch effizientere Suchverfahren gäbe.

Im Folgenden soll nun eine untere Schranke für die Anzahl der notwendigen Vergleiche bestimmt werden, indem ein *Entscheidungsbaum* für Suchalgorithmen konstruiert wird. Betrachte einen beliebigen auf Vergleiche basierenden Suchalgorithmus. Ein Entscheidungsbaum für einen solchen Algorithmus und der Eingabegröße n ist dann ein Binärbaum, dessen Knoten mit den Feldindizes von 1 bis n benannt und nach dem folgenden Schema angeordnet sind. Sei $A = (a_1, \ldots, a_n)$ ein aufsteigend sortiertes Feld und x der zu suchende Schlüssel.

1. Die Wurzel des Baumes ist mit dem Index benannt, der als erstes mit dem Schlüssel verglichen wird.

2. Betrachte den Knoten i.

   - Der Name des linken Nachfolgers ist der Index, mit dem der Schlüssel als nächstes verglichen wird, wenn $x < a_i$ ist.
   - Der Name des rechten Nachfolgers ist der Index, mit dem der Schlüssel als nächstes verglichen wird, wenn $x > a_i$ ist.
   - Der Knoten hat keinen linken (oder rechten) Nachfolger, wenn der Algorithmus nach dem Vergleich von x und a_i hält und $x < a_i$ (bzw. $x > a_i$) gilt.
   - Es gibt für den Fall $x = a_i$ keinen Nachfolger, da der Algorithmus dann hält.

Abbildung 4.1 zeigt den Entscheidungsbaum für den sequentiellen Suchalgorithmus und Abbildung 4.2 für den binären Suchalgorithmus. Für einen anderen Suchalgorithmus kann der Entscheidungsbaum völlig anders aussehen (Abb. 4.3).

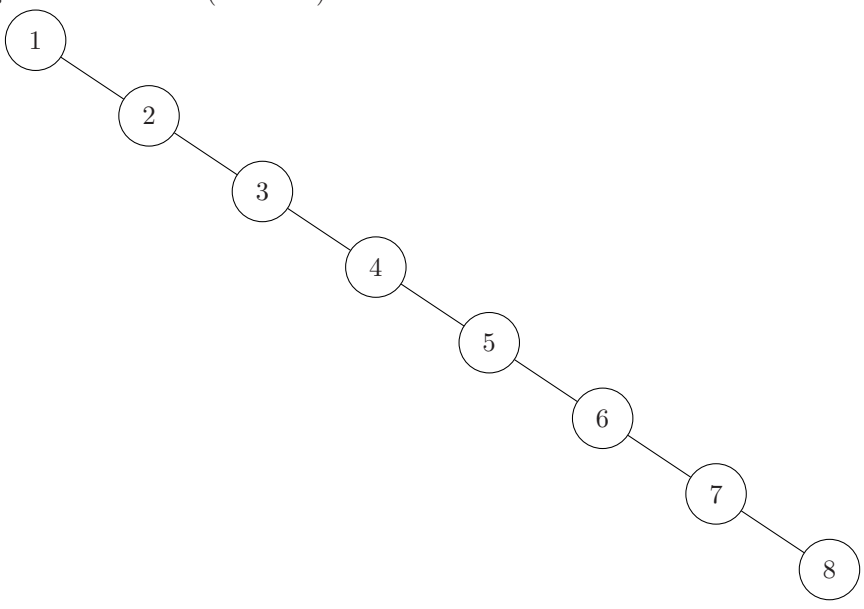


Abbildung 4.1: Entscheidungsbaum für die sequentielle Suche mit $n = 8$

Mit einer bestimmten Eingabe wird nun ein Suchalgorithmus eine Sequenz von Vergleichen durchführen, die einem Pfad in dem Entscheidungsbaum entsprechen. Ein solcher Pfad beginnt mit der Wurzel und endet spätestens in einem Blatt. Die maximale Anzahl der Vergleiche entspricht also der Knotenanzahl auf dem längsten Pfad in dem Entscheidungsbaum. Sei diese Anzahl p. Dann gibt es $p - 1$ Kanten auf diesem Pfad.

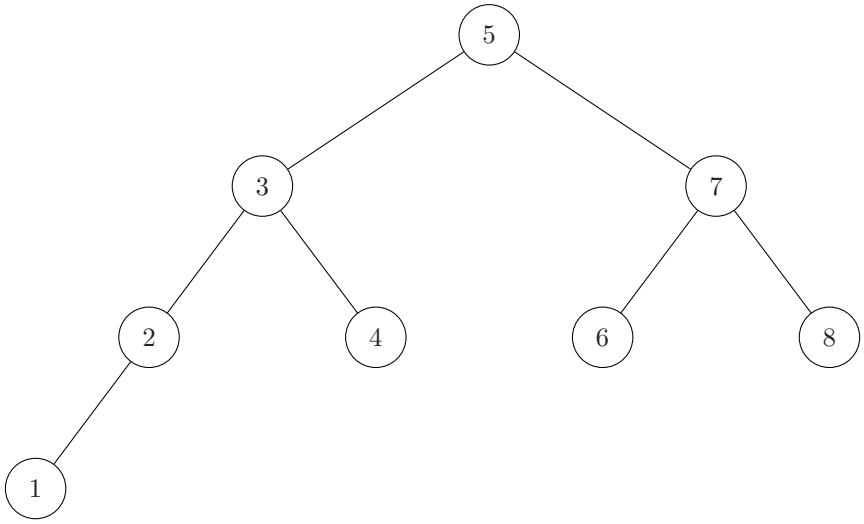


Abbildung 4.2: Entscheidungsbaum für die binäre Suche mit $n = 8$

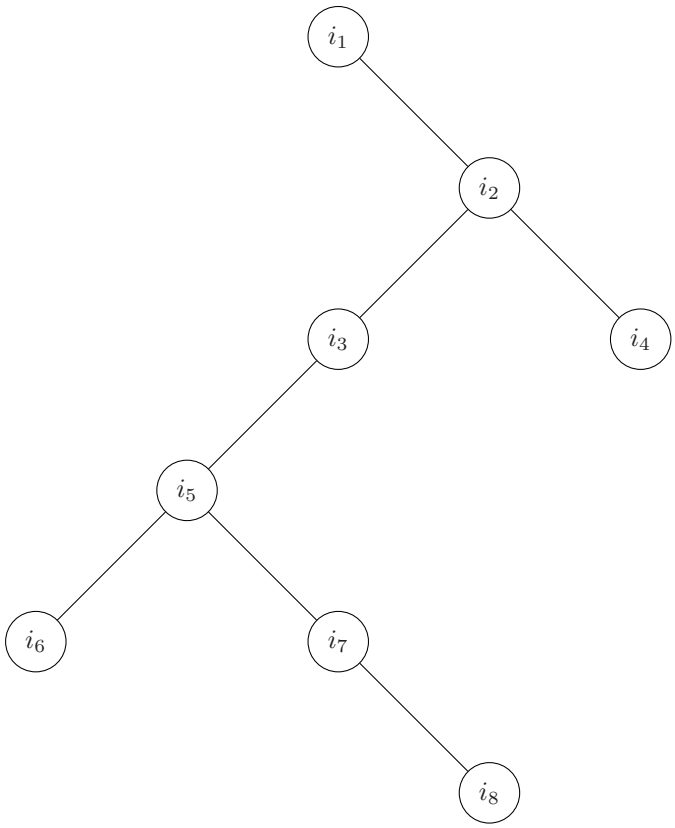


Abbildung 4.3: Entscheidungsbaum für die binäre Suche mit einem fiktiven Verfahren mit $n = 8$, $i_j \in \{1, 2, \ldots, 8\}, 1 \leq j \leq 8$

Sei angenommen, der Entscheidungsbaum habe N Knoten. Da jeder Knoten höchstens zwei Nachfolger hat, gibt es höchstens

$$1 + 2 + 4 + \ldots + 2^{p-1} = 2^p - 1$$

Knoten in diesem Abstand von der Wurzel. Die Anzahl der Knoten in dem Entscheidungsbaum muss also kleiner oder gleich sein, d. h. $N + 1 \leq 2^p$.

Diese Beziehung ist nun in Relation zur Anzahl n der zu untersuchenden Elemente in dem Zahlenfeld zu setzen. Mit der Behauptung, dass es für jeden Index i mit $1 \leq i \leq n$ einen Knoten im Entscheidungsbaum gibt, folgt $N \geq n$.

Werde zuächst diese Behauptung durch Widerspruch bewiesen: Sei angenommen, es gebe ein i, $1 \leq i \leq n$, für den kein Knoten im Baum existiert. Betrachte dann zwei sortierte Zahlenfelder A_1 und A_2 mit $a_{1,i} = x$ und $a_{2,i} = x' > x$. Sei ansonsten $a_{1,j} = a_{2,j} < x$ für alle $1 \leq j < i$ und $a_{1,j} = a_{2,j} > x'$ für alle $i < j \leq n$:

$$\begin{aligned} A_1 &= (a_{1,1}, \ldots, a_{1,i-1}, x, a_{1,i+1}, \ldots, a_{1,n}) \\ A_2 &= (a_{2,1}, \ldots, a_{2,i-1}, x', a_{2,i+1}, \ldots, a_{2,n}) \end{aligned}$$

Da kein Knoten i in dem Entscheidungsbaum existiert, kann der Algorithmus auch keinen Vergleich von $a_{1,i}$ oder $a_{2,i}$ mit x durchführen. Für beide Eingaben muss er sich also gleich verhalten. Somit muss mindestens für eine der beiden Eingaben eine falsche Ausgabe geliefert werden. Widerspruch.

Die Behauptung ist also richtig, d. h. der Entscheidungsbaum hat mindestens n Knoten. Also gilt

$$n + 1 \leq N + 1 \leq 2^p.$$

Durch Logarithmieren erhält man $p \geq \mathrm{ld}\,(n+1)$ als Anzahl an Vergleichen auf dem längsten Pfad in dem Entscheidungsbaum. Also sind mindestens $\lceil \mathrm{ld}\,(n+1) \rceil$ Vergleiche notwendig, um den zu suchenden Schlüssel in einem Zahlenfeld der Größe n zu finden. Damit ist der nachfolgende Satz 4.1 bewiesen.

Satz 4.1 (Minimale Suchkomplexität)
Ein Suchalgorithmus, der in einem sortierten Feld der Größe n durch Vergleich des Schlüssels mit den Feldelementen sucht, benötigt für den schlechtesten Fall mindestens $\lceil \mathrm{ld}\,(n+1) \rceil \in \Omega(\log n)$ Vergleiche.

Die sequentielle Suche (Algorithmus 4.2), die im schlechtesten Fall linear ist, ist nicht optimal. Die binäre Suche (Algorithmus 4.3), die im schlechtesten Fall logarithmisch ist, ist optimal.

4.5 Bubble-Sort

Der *Bubble-Sort* (Algorithmus 4.4) ist ein Sortierverfahren, bei dem in mehreren Durchläufen jeweils zwei nebeneinander stehende Elemente miteinander verglichen und gegebenenfalls vertauscht werden. Da sich dabei die Elemente wie im Wasser aufsteigende „Blasen“ durch das Feld bewegen, ist das Verfahren auch so benannt worden.

Algorithmus 4.4 (Bubble-Sort I)
Eingabe: $A = (a_1, \ldots, a_n)$ mit $a_i, i, n \in \mathbb{N}, 1 \leq i \leq n$
Ausgabe: $A = (a_{\pi(1)}, \ldots, a_{\pi(n)})$ mit $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$

```
BUBBLE-SORT(A)
1  for m ← n − 1 downto 1
2      do for i ← 1 to m
3             do if A[i] > A[i + 1]
4                    then VERTAUSCHE(A[i], A[i + 1])
5  return A
```

Beispiel 4.6
Betrachte die Folge $A = (5, 3, 4, 7, 3, 2)$, also $n = 6$:

m	i	A	$A[i] > A[i+1]$	Vertauschen
5	1	$(5, 3, 4, 7, 3, 2)$	$5 > 3$	ja
	2	$(3, 5, 4, 7, 3, 2)$	$5 > 4$	ja
	3	$(3, 4, 5, 7, 3, 2)$	$5 > 7$	nein
	4	$(3, 4, 5, 7, 3, 2)$	$7 > 3$	ja
	5	$(3, 4, 5, 3, 7, 2)$	$7 > 2$	ja
4	1	$(3, 4, 5, 3, 2, 7)$	$3 > 4$	nein
	2	$(3, 4, 5, 3, 2, 7)$	$4 > 5$	nein
	3	$(3, 4, 5, 3, 2, 7)$	$5 > 3$	ja
	4	$(3, 4, 3, 5, 2, 7)$	$5 > 2$	ja
3	1	$(3, 4, 3, 2, 5, 7)$	$3 > 4$	nein
	2	$(3, 4, 3, 2, 5, 7)$	$4 > 3$	ja
	3	$(3, 3, 4, 2, 5, 7)$	$4 > 2$	ja
2	1	$(3, 3, 2, 4, 5, 7)$	$3 > 3$	nein
	2	$(3, 3, 2, 4, 5, 7)$	$3 > 2$	ja
1	1	$(3, 2, 3, 4, 5, 7)$	$3 > 2$	ja
		$(2, 3, 3, 4, 5, 7)$		

□

Nach dem ersten Durchlauf der inneren Schleife ist das Maximum an der letzten Position, nach dem zweiten Durchlauf der inneren Schleife das zweitgrößte Element an der vorletzten Position, usw. bis das Feld sortiert ist.

Zur Verifikation dieses Algorithmus sind für die zwei ineinander geschachtelten Schleifen die Invarianten zu bestimmen und über vollständige Induktion zu beweisen. Darauf soll hier verzichtet werden.

Die Komplexitätsanalyse des Algorithmus erfordert eine genaue Betrachtung der Schleifen-Durchläufe. Daraus ergibt sich dann unmittelbar die Anzahl der Vergleiche. Die Anzahl der Vertauschungen müssen nicht berücksichtigt werden, da sie höchstens die Anzahl der Vergleiche annehmen kann, also auf jeden Fall proportional zu diesen bleibt.

I. Durchläufe äußere Schleife: $n-1$

II. Durchläufe innere Schleife:

$$(n-1)+(n-2)+\ldots+1=\sum_{m=1}^{n-1} m=\frac{n(n-1)}{2}$$

III. Vergleiche:

$$\frac{n(n-1)}{2}$$

Damit kann nun die Komplexität des Algorithmus in asymptotischer Notation ausgedrückt werden. Da die Anzahl der Vergleiche nur von der Eingabegöße nicht aber von der Eingabe abhängt, sind die Komplexitäten im besten, im mittleren und im schlechtesten Fall gleich.

$$B(n)=A(n)=W(n)=\frac{n(n-1)}{2}\in\Theta(n^2)$$

Bleibt zu diesem Algorithmus festzustellen, dass es sich um einen *am-Platz*-Algorithmus handelt, denn das Vertauschen erfordert die Zwischenspeicherung nur eines Elementes, und dass der Algorithmus stabil ist, denn gleiche Elemente werden nicht vertauscht.

Eine genauere Analyse des Bubble-Sort zeigt, dass es Schleifendurchläufe in Abhängigkeit von der Eingabe geben kann, bei denen gar keine Vertauschungen mehr stattfinden. Die nachfolgende Variation des Bubble-Sort's (Algorithmus 4.5) sorgt für einen frühzeitigen Abbruch der Schleifen. Der Algorithmus erkennt, dass Teile der Folge oder die gesamte Folge bereits sortiert vorliegen.

Algorithmus 4.5 (Bubble-Sort II)
Eingabe: $A=(a_1,\ldots,a_n)$ mit $a_i,i,n\in\mathbb{N},1\leq i\leq n$
Ausgabe: $A=(a_{\pi(1)},\ldots,a_{\pi(n)})$ mit $a_{\pi(1)}\leq a_{\pi(2)}\leq\ldots\leq a_{\pi(n)}$

```
BUBBLE-SORT(A)
 1  m ← n − 1
 2  do
 3     b ← FALSE
 4     for i ← 1 to m
 5        do if A[i] > A[i + 1]
 6              then b ← TRUE
 7                   k ← i
 8                   VERTAUSCHE(A[i], A[i + 1])
 9     m ← k − 1
10     while b
11  return A
```

Beispiel 4.7
Betrachte die Folge $A = (4, 3, 7, 2, 4, 5)$, also $n = 6$:

m	i	A	$A[i] > A[i+1]$	Vertauschen	k
5	1	$(4, 3, 7, 2, 4, 5)$	$4 > 3$	ja	1
	2	$(3, 4, 7, 2, 4, 5)$	$4 > 7$	nein	
	3	$(3, 4, 7, 2, 4, 5)$	$7 > 2$	ja	3
	4	$(3, 4, 2, 7, 4, 5)$	$7 > 4$	ja	4
	5	$(3, 4, 2, 4, 7, 5)$	$7 > 5$	ja	5
4	1	$(3, 4, 2, 4, 5, 7)$	$3 > 4$	nein	
	2	$(3, 4, 2, 4, 5, 7)$	$4 > 2$	ja	2
	3	$(3, 2, 4, 4, 5, 7)$	$4 > 4$	nein	
	4	$(3, 2, 4, 4, 5, 7)$	$4 > 5$	nein	
1	1	$(3, 2, 4, 4, 5, 7)$	$3 > 2$	ja	1
		$(2, 3, 4, 4, 5, 7)$			

□

Der beste Fall tritt ein, wenn das Zahlenfeld bereits aufsteigend sortiert ist, der schlechteste Fall, wenn es absteigend sortiert ist. Werde auch für diese Variante des Algorithmus eine Komplexitätsanalyse vorgenommen.

I. Durchläufe d_a äußere Schleife: $1 \leq d_a \leq n - 1$

II. Durchläufe d_i innere Schleife:
$$n - 1 \leq d_i \leq \frac{n(n-1)}{2}$$

III. Vergleiche v:
$$n - 1 \leq v \leq \frac{n(n-1)}{2}$$

Die Komplexität dieser Variante des Bubble-Sort's ergibt sich damit wie folgt:

$$\begin{aligned} B(n) &= n - 1 \qquad \in \Theta(n) \\ W(n) &= \frac{n(n-1)}{2} \in \Theta(n^2) \end{aligned}$$

Ohne Herleitung sei die Komplexität im mittleren Fall genannt:

$$A(n) \in \Theta(n^2)$$

Die zweite Variante des Bubble-Sort's bringt also bei asymptotischer Betrachtung nur für den besten Fall eine Effizienzverbesserung. Für den mittleren und schlechtesten Fall ändert sich nichts. Für die Praxis ist die zweite Variante dennoch vorzuziehen.

4.6 Merge-Sort

Der *Merge-Sort* ist ein Sortierverfahren, bei dem durch rekursive Aufrufe das zu sortierende Feld zunächst mehrfach geteilt und dann jeweils zwei bereits sortierte Felder zu einem sortierten Feld zusammengemischt werden. Der Algorithmus entspricht einem typischen *divide-and-conquer* Verfahren. Der Merge-Sort war bereits ansatzweise im Abschnitt 3.2 besprochen worden.

- **Divide:** Das Zahlenfeld der Länge n wird rekursiv in Felder der Länge $n/2$ zerlegt.
- **Conquer:** Ein Zahlenfeld der Länge 1 ist sortiert.
- **Combine:** Jeweils zwei sortierte Zahlenfelder werden zu einem sortierten Feld gemischt, bis das Ausgangsfeld sortiert vorliegt.

Im Algorithmus 4.6 stellt die Prozedur *Merge-Sort* den Kern des Verfahrens dar. Das Mischen der sortierten Felder erfolgt durch die Prozedur *Merge*.

Algorithmus 4.6 (Merge-Sort)
Eingabe: $A = (a_1, \ldots, a_n)$ mit $a_i, i, n \in \mathbb{N}, 1 \leq i \leq n$ und $p = 1, r = n$
Ausgabe: $A = (a_{\pi(1)}, \ldots, a_{\pi(n)})$ mit $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$

```
MERGE-SORT(A, p, r)
 1  if p < r
 2     then q ← ⌊(p + r)/2⌋
 3          MERGE-SORT(A, p, q)
 4          MERGE-SORT(A, q + 1, r)
 5          MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
 1  i ← p
 2  j ← q + 1
 3  k ← 0
 4  do
 5     k ← k + 1
 6     if A[i] ≤ A[j]
 7        then B[k] ← A[i]
 8             i ← i + 1
 9        else B[k] ← A[j]
10             j ← j + 1
11     while (i ≤ q) ∧ (j ≤ r)
12  do
13     k ← k + 1
14     if i ≤ q
15        then B[k] ← A[i]
16             i ← i + 1
17        else B[k] ← A[j]
18             j ← j + 1
19     while (i ≤ q) ∨ (j ≤ r)
20  A[p..r] ← B[1..k]
```

Auf einen formalen Beweis des Algorithmus 4.6 soll hier verzichtet werden. Zur Veranschaulichung des Ablaufs möge das Beispiel 4.8 genügen.

Beispiel 4.8
Betrachte die Folge $A = (5, 3, 4, 7, 3, 2)$ mit $n = 6$:

MERGE-SORT$(A, 1, 6)$:

Rekursions-tiefe	A (divide) $\downarrow$	p	r	q	A (combine) $\uparrow$
0	$(5,3,4,7,3,2)$	1	6	3	$(2,3,3,4,5,7)$
1	$(5,3,4)$	1	3	2	$(3,4,5)$
2	$(5,3)$	1	2	1	$(3,5)$
3	(5)	1	1		(5)
3	(3)	2	2		(3)
2	(4)	3	3		(4)
1	$(7,3,2)$	4	6	5	$(2,3,7)$
2	$(7,3)$	4	5	4	$(3,7)$
3	(7)	4	4		(7)
3	(3)	5	5		(3)
2	(2)	6	6		(2)

MERGE$(A,4,4,5)$:

p	q	r	i	j	$A[4..4]$	$A[5..5]$	$A[i] \leq A[j]$	k	B
4	4	5	4	5	(7)	(3)	$7 \not\leq 3$	1	(3)
			4	6				2	(3,7)
			5	7					

MERGE$(A,1,3,6)$:

p	q	r	i	j	$A[1..3]$	$A[4..6]$	$A[i] \leq A[j]$	k	B
1	3	6	1	4	(3,4,5)	(2,3,7)	$3 \not\leq 2$	1	(2)
			1	5			$3 \leq 3$	2	(2,3)
			2	5			$4 \not\leq 3$	3	(2,3,3)
			2	6			$4 \leq 7$	4	(2,3,3,4)
			3	6			$5 \leq 7$	5	(2,3,3,4,5)
			4	6				6	(2,3,3,4,5,7)
			4	7					

Der rekursive Ablauf des Teilens und Mischens des Zahlenfeldes bei Eintritt und Verlassen der einzelnen Rekursionschritte kann sehr übersichtlich in Bäumen, so genannten *Rekursionsbäumen*, dargestllt werden. Betrachte dazu die Abbildungen 4.4 und 4.5. □

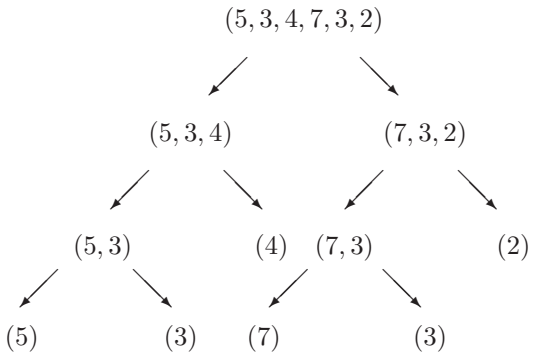


Abbildung 4.4: Rekursives Teilen des Feldes $(5,3,4,7,3,2)$

Die Komplexität des Algorithmus 4.6 war im Wesentlichen bereits in Abschnitt 3.2 hergeleitet worden. Sie ist für den besten, mittleren und schlechtesten Fall gleich:

$$B(n) = A(n) = W(n) = n\mathrm{ld}\, n \in \Theta(n \log n).$$

Hergeleitet wurde diese Komplextiät aus der Rekursionsgleichung

$$T(n) = \begin{cases} \Theta(1), & \text{falls } n = 1 \\ 2T(n/2) + \Theta(n), & \text{falls } n > 1\ . \end{cases}$$

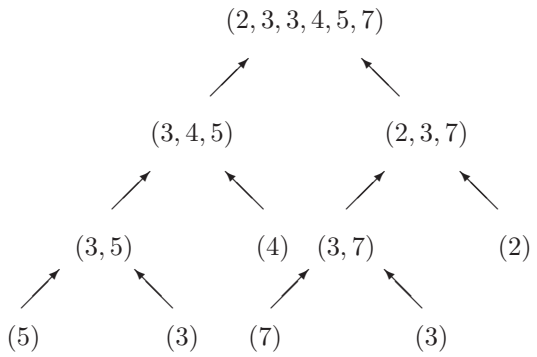


Abbildung 4.5: Rekursives Mischen der sortierten Felder

Noch nicht hergeleitet war der lineare nicht-rekursive Anteil der Komplexität, den die Prozedur *Merge* beiträgt. Hier werden zwei sortierte Teilfelder der Länge $n/2$ zu einem sortierten Teilfeld der Länge n zusammengemischt. Dies erfolgt in zwei aufeinander folgenden Schleifen. Die Eingabegröße von *Merge* ergibt sich aus den Parametern p und r, und es gilt $n = r - p + 1$.

In der ersten Schleife (Zeile 4) werden die jeweils kleinsten Elemente aus den beiden sortierten Feldern miteinander verglichen und das kleinere in das Ergebnisfeld übertragen. In der zweiten Schleife (Zeile 12) werden die verbleibenden Elemente in das Ergebnisfeld übertragen, wenn die Elemente eines Feldes vollständig abgearbeitet sind. Die Summe beider Schleifendurchläufe beträgt n. Die erste Schleife wird dabei d-mal durchlaufen, $n/2 \leq d \leq n - 1$, die zweite Schleife $(n - d)$-mal. Als zu zählende grundlegende Anweisungen sind hier die jeweils alternativ auszuführenden Zuweisungen in den Zeilen 7 und 9 und den Zeilen 15 und 17 zu wählen, zumindest wenn man den Aufwand etwas genauer haben möchte. Bei der Wahl des Vergleichs in Zeile 6 als zu zählende grundlegende Anweisung würde nämlich die zweite Schleife gar nicht mehr gezählt werden, denn sie enthält keinen Vergleich. Für die asymptotische Notation ist diese Wahl aber irrelevant, denn beide Fälle führen für *Merge* zu der Komplexität $\Theta(n)$.

I. Anzahl Vergleiche v: $n/2 \leq v \leq n - 1$

II. Anzahl Zuweisungen: n

Der für den Sortiervorgang zusätzlich benötigte Speicherplatz ist aber nicht mehr konstant, *Merge-Sort* ist also kein *am Platz*-Verfahren. Das Feld B in der Prozedur *Merge* muss maximal alle n Elemente des zu sortierenden Feldes A zwischenspeichern. Hinzu kommt der Platzbedarf für die rekursiven Prozeduraufrufe. Da die Rekursionstiefe aber in Abhängigkeit von n logarithmisch ist, ergibt sich insgesamt eine lineare Speicher-Komplexität, also $\Theta(n)$. *Merge-Sort* ist stabil.

4.7 Quick-Sort

Der *Quick-Sort* ist auch ein Sortierverfahren, das nach dem *divide-and-conquer* Prinzip verfährt. Im Gegensatz zum Merge-Sort, bei dem der eigentliche Sortieraufwand nach der Rückkehr aus den rekursiven Aufrufen entsteht, entsteht der Sortieraufwand hier vor den rekursiven Aufrufen (Algorithmus 4.7).

- **Divide:** Das Zahlenfeld der Länge n wird rekursiv in Teilfelder zerlegt und dabei so umsortiert, dass jedes Element des ersten Teilfeldes kleiner oder gleich als ein so genanntes *Pivotelement* und jedes Element des zweiten Teilfeldes größer oder gleich als dieses Pivotelement ist. Das Pivotelement ist ein aus dem Zahlenfeld zu wählendes Element, hier dem letzten Element des Feldes. Es bestimmt damit die Position der Teilung.
- **Conquer:** Ein Zahlenfeld der Länge 1 ist sortiert.
- **Combine:** Jeweils zwei zusammenzufügende Zahlenfelder sind bereits so sortiert, dass sie nur noch aneinander gehängt werden müssen. Das Pivotelement steht jeweils dazwischen.

Algorithmus 4.7 (Quick-Sort)
Eingabe: $A = (a_1, \ldots, a_n)$ mit $a_i, i, n \in \mathbb{N}, 1 \leq i \leq n$ und $p = 1, r = n$
Ausgabe: $A = (a_{\pi(1)}, \ldots, a_{\pi(n)})$ mit $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$

```
QUICK-SORT(A, p, r)
1  if p < r
2     then q ← PARTITION(A, p, r)
3          QUICK-SORT(A, p, q − 1)
4          QUICK-SORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4     do if A[j] ≤ x
5          then i ← i + 1
6               VERTAUSCHE(A[i], A[j])
7  i ← i + 1
8  VERTAUSCHE(A[i], A[r])
9  return i
```

Auf einen formalen Beweis des Algorithmus 4.7 soll hier verzichtet werden. Zur Veranschaulichung des Ablaufs möge das Beispiel 4.9 genügen.

Zum Beweis wäre insbesondere die Prozedur *Partition* zu untersuchen. Als Schleifeninvariante wird das Größenverhältnis der Feldelemente zum Pivotelement x betrachtet:

I. $A[k] \leq x$, falls $p \leq k \leq i$ (erstes Teilfeld)

II. $A[k] > x$, falls $i + 1 \leq k \leq j - 1$ (zweites Teilfeld)

III. $A[k] = x$, falls $k = r$

Die Elemente $A[k]$ mit $j \leq k \leq r - 1$ haben kein definiertes Verhältnis zum Pivotelement. In Abbildung 4.6 ist diese Feldaufteilung veranschaulicht.

Beispiel 4.9
Betrachte die Folge $A = (5, 3, 2, 7, 3, 4)$ mit $n = 6$:

QUICK-SORT$(A, 1, 6)$:

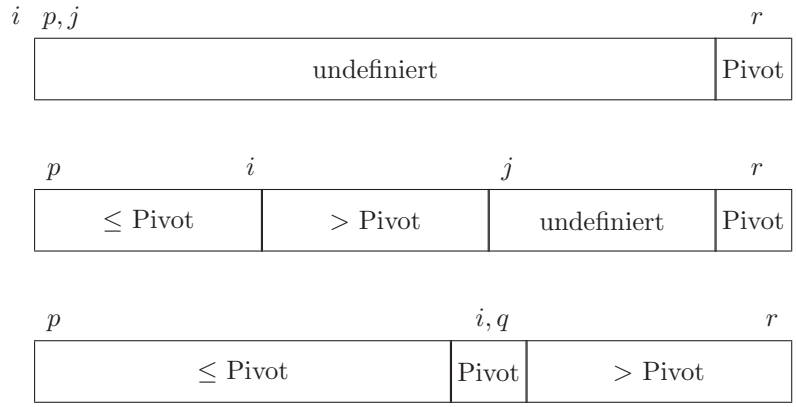


Abbildung 4.6: Feldaufteilung vor, während und nach der Ausführung von *Partition*

Rekursions-tiefe	A (divide) ↓	p	r	q	$A[q]$	A (combine) ↑
0	$(5,3,2,7,3,4)$	1	6			$(2,3,3,4,5,7)$
	$(3,2,3,4,5,7)$			4	4	
1	$(3,2,3)$	1	3			$(2,3,3)$
	$(3,2,3)$			3	3	
2	$(3,2)$	1	2			$(2,3)$
	$(2,3)$			1	2	
3	$(\,)$	1	0			$(\,)$
3	(3)	2	2			(3)
2	$(\,)$	4	3			$(\,)$
1	$(5,7)$	5	6			$(5,7)$
	$(5,7)$			6	7	
2	(5)	5	5			(5)
2	$(\,)$	7	6			$(\,)$

PARTITION$(A,1,6)$:

p	r	$A[1..6]$	x	j	$A[j] \leq x$	i	$A[i] \leftrightarrow A[j]$	$A[i] \leftrightarrow A[r]$
1	6	$(5,3,2,7,3,4)$	4	1	$5 \not\leq 4$	0		
				2	$3 \leq 4$	1	$5 \leftrightarrow 3$	
		$(3,5,2,7,3,4)$		3	$2 \leq 4$	2	$5 \leftrightarrow 2$	
		$(3,2,5,7,3,4)$		4	$7 \not\leq 4$			
				5	$3 \leq 4$	3	$5 \leftrightarrow 3$	
		$(3,2,3,7,5,4)$				4		$7 \leftrightarrow 4$
		$(3,2,3,4,5,7)$						

Der Wert $i = 4$ wird zurück geliefert.

Der Ablauf des Aufrufs von PARTITION$(A,1,6)$ kann auch gut mit Hilfe von Abbildung 4.7 veranschaulicht werden. Insbesondere werden auch die Eigenschaften der Schleifeninvariante deutlich. □

Der rekursive Ablauf des Teilens und Zusammenfügens des Zahlenfeldes bei Eintritt und Verlassen der einzelnen Rekursionschritte kann sehr übersichtlich in Rekursionsbäumen dargestllt werden. Betrachte dazu die Abbildungen 4.8 und 4.9.

In *Quick-Sort* kann es auch zu Vertauschungen von gleichen Elementen kommen, betrachte dazu z. B. Abbildung 4.10. Es handelt sich also um ein instabiles Sortierverfahren.

Die Laufzeit und damit die Komplexität von *Quick-Sort* hängt davon ab, ob bei der Teilung (annähernd) vollständige oder (überwiegend) unvollständige binäre Rekursionsbäume entstehen

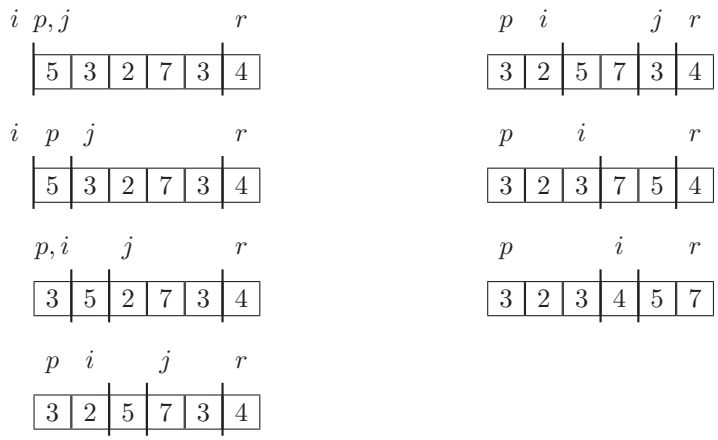


Abbildung 4.7: Ablauf von PARTITION$(A, 1, 6)$ mit $A = (5, 3, 2, 7, 3, 4)$ (jeweils bei Eintritt in die Schleife)

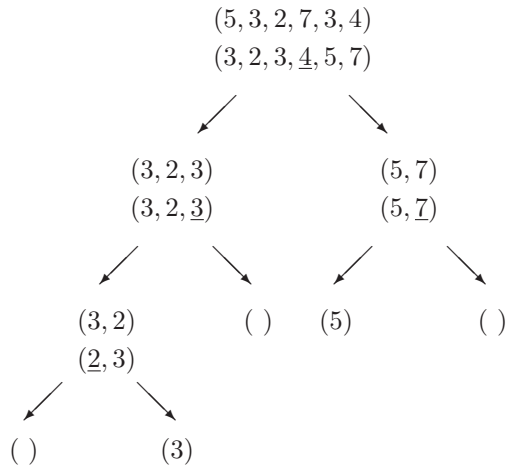


Abbildung 4.8: Rekursives Teilen des Feldes $(5, 3, 2, 7, 3, 4)$, die Pivotelemente sind unterstrichen

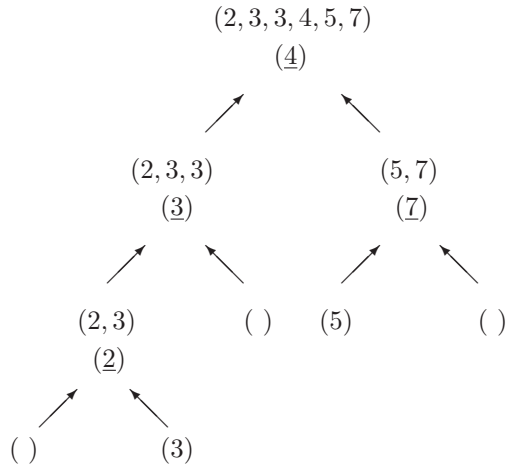


Abbildung 4.9: Rekursives Zusammenfügen der Teilfelder

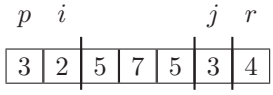


Abbildung 4.10: Beispiel zur Instabilität. $A[3]$ ist mit $A[6]$ zu tauschen. Damit ändert sich die Ordnung bzgl. $A[5]$.

(Definition 5.9). Im ersten Fall wird sich dieses Verfahren asymptotisch wie *Merge-Sort* verhalten, im zweiten Fall aber wie *Bubble-Sort* im schlechtesten Fall.

Zur Bestimmung der Komplexität sind insbesondere die Rekursionstiefe der Prozedur *Quick-Sort* und die Anzahl der Schleifendurchläufe der Prozedur *Partition* zu analysieren. Aus der Komplexität von *Partition* ergibt sich dann die Komplexität je Rekursionsschritt.

Werde zunächst die Prozedur *Partition* betrachtet (nicht-rekursiver Anteil). Grundlegende Anweisung ist der Vergleich in Zeile 4. Die Schleife (Zeile 3) durchläuft bis auf das Pivotelement alle Elemente. Bei n Elementen sind das $n-1$ Vergleiche, es ergibt sich also eine asymptotische Komplexität von $\Theta(n)$, und zwar unabhängig von der Eingabe.

Die Komplexitätsanalyse der Prozedur *Quick-Sort* muss dagegen für jeden Fall einzeln vorgenommen werden. Der schlechteste Fall tritt ein, wenn die Teilung jeweils Teilfelder der Größe 0 und $n-1$ ergibt, das Pivotelement verbleibt außerhalb. Das ist interessanterweise dann der Fall, wenn das Feld bereits sortiert ist. Da das Teilen der Felder – wie oben ausgeführt – eine Komplexität von $\Theta(n)$ hat und sich das Zusammenfügen bei Rückkehr aus der Rekursion ohne weiteren Aufwand ergibt, kann die Rekursionsgleichung wie folgt aufgestellt werden:

$$T(n) = T(n-1) + T(1) + \Theta(n)$$

Dabei ist $T(0) = T(1) = \Theta(1)$, denn nach Auswertung der Bedingung der Verzweigung (Zeile 1) terminiert die Prozedur und die Rekursion bricht ab. Wegen $\Theta(1) + \Theta(n) = \Theta(n)$ und mit $f(n) = n - 1 \in \Theta(n)$ bleibt die Rekursionsgleichung

$$T(n) = T(n-1) + f(n)$$

zu lösen. Mit Satz 3.1 ergibt sich

$$T(n) = \sum_{d=0}^{n-1} f(n-d) = \sum_{d=0}^{n-1} (n-1-d) = \frac{(n-1)n}{2} \ .$$

Also

$$W(n) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

Der beste Fall für *Quick-Sort* liegt vor, wenn bei der Teilung des Feldes jeweils Feldgrößen von $n/2$ entstehen. Die Komplexität für diesen Fall ergibt sich dann aus der Rekursionsgleichung

$$T(n) = 2T(n/2) + \Theta(n),$$

die aufgrund des Master-Theorems (Satz 3.2) die Lösung

$$T(n) \in \Theta(n \log n)$$

hat. Also

$$B(n) \in \Theta(n \log n).$$

Auf eine exakte Berechnung der Komplexität im mittleren Fall soll hier verzichtet werden. Asymptotisch ist sie aber gleich dem besten Fall:

$$A(n) \in \Theta(n \log n).$$

Eine randomisierte Variante des *Quick-Sort* verbessert zwar nicht seine Komplexität im schlechtesten Fall, macht ihn aber von der initialen Anordnung der Zahlen unabhängig. Die aufsteigende

Sortierung des gegebenen Zahlenfeldes wird damit sehr unwahrscheinlich, nämlich $1/n!$. Die Zahlen werden zu Beginn gemischt, so dass eine zufällige Permutation entsteht. Das geht in einer Komplexität von $\Theta(n)$. Eine andere Möglichkeit mit gleichem Effekt besteht darin das Pivotelement zufällig auszuwählen (Algorithmus 4.8). Die Komplexität bleibt damit bei beiden Varianten im Mittel bei $\Theta(n \log n)$.

Algorithmus 4.8 (Random-Quick-Sort)
Eingabe: $A = (a_1, \ldots, a_n)$ mit $a_i, i, n \in \mathbb{N}, 1 \leq i \leq n$ und $p = 1, r = n$
Ausgabe: $A = (a_{\pi(1)}, \ldots, a_{\pi(n)})$ mit $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$

```
QUICK-SORT(A, p, r)
1  if p < r
2     then q ← RANDOM-PARTITION(A, p, r)
3          QUICK-SORT(A, p, q)
4          QUICK-SORT(A, q + 1, r)

RANDOM-PARTITION(A, p, r)
1  i ← RANDOM(p, r)
2  VERTAUSCHE(A[r], A[i])
3  return PARTITION(A, p, r)

PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4     do if A[j] ≤ x
5           then i ← i + 1
6                VERTAUSCHE(A[i], A[j])
7  i ← i + 1
8  VERTAUSCHE(A[i], A[r])
9  return i
```

Die Speicherplatz-Komplexität von *Quick-Sort*, so wie das Verfahren hier beschrieben ist, ist linear, da im schlechtesten Fall die Rekursionstiefe nur durch $\Theta(n)$ abgeschätzt werden kann. Für jeden Rekursionsaufruf wird ein konstanter Speicherbereich benötigt. Weiterer zusätzlicher Speicher ist zur Zwischenspeicherung von Feldelementen nicht nötig, die Vertauschung zweier Elemente erfolgt *am Platz*.

Durch Ausnutzen der in *Quick-Sort* vorliegenden so genannten *Tail-Rekursion* kann durch Modifikation des Algorithmus die Speicher-Komplexität aber auf $\Theta(\log n)$ reduziert werden. Darauf soll hier aber nicht näher eingegangen werden. Bleibt abschließend festzustellen, dass *Quick-Sort* in der Speicher-Komplexität deutlich effizienter ist als *Merge-Sort*.

4.8 Sortieren und Optimalität

Die vorangehenden Abschnitte haben gezeigt, dass das Sortieren von n Elementen in der Zeit $\Theta(n \log n)$ möglich ist. Alle bisher betrachteten Verfahren basieren ausschließlich auf dem Vergleich der Elemente. Solche Verfahren werden *vergleichende Sortierverfahren* genannt. Im Folgenden soll nun untersucht werden, dass solche Verfahren diese Zeit auch mindestens brauchen.

Sei $(a_1, a_2, \ldots, a_n)$ die Folge der zu sortierenden Werte. Bei vergleichenden Sortierverfahren erfolgt das Vertauschen zweier Elemente a_i und a_j ausschließlich aufgrund der Vergleiche mittels $\leq, \geq, <, >$ oder $=$ von a_i und a_j. Hier sei angenommen, die Elemente seien paarweise verschieden. Weiterhin kann man sich auf die Betrachtung der Vergleichsoperation $a_i \leq a_j$ beschränken. Das Ergebnis der Sortierung ist eine Permutation der Eingabewerte $(a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)})$ mit $a_{\pi(1)} < a_{\pi(2)} < \ldots < a_{\pi(n)}$. Jede der $n!$ Permutationen kann nun Ergebnis einer Sortierung sein.

Die möglichen Ergebnisse eines vergleichenden Sortierverfahrens können als ein *Entscheidungsbaum* dargestellt werden (Abbildung 4.11). Die inneren Knoten repräsentieren dabei die jeweiligen Vergleiche und die Blätter die möglichen Permutationen. Jeder innere Knoten hat genau zwei Nachfolger. In Abbildung 4.12 ist der Pfad der Entscheidungen aufgeführt, die bei *Quick-Sort* beim erster Aufruf von *Partition* auftreten (Beispiel 4.9).

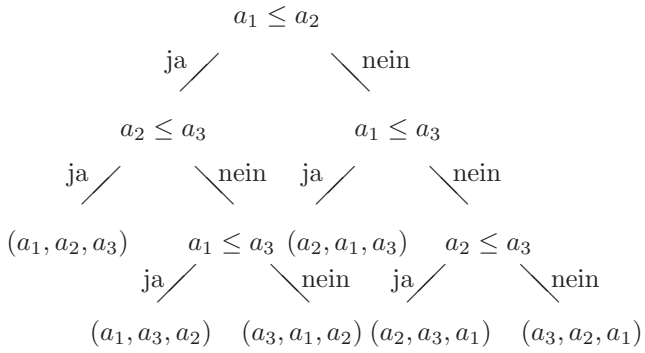


Abbildung 4.11: Entscheidungsbaum der Vergleiche und Sortierungen von (a_1, a_2, a_3)

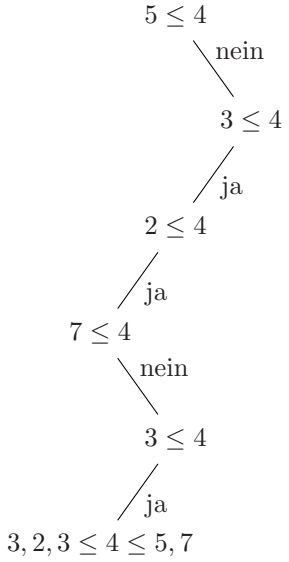


Abbildung 4.12: Entscheidungspfad der Vergleiche aus Beispiel 4.9, Aufruf PARTITION$(A, 1, 6)$ bei *Quick-Sort*

Die Ausführung eines Sortierverfahrens entspricht nun der Verfolgung eines Pfades von der Wurzel bis zu einem Blatt des Entscheidungsbaums. Die Länge eines solchen längsten Pfades entspricht

damit der Anzahl der Vergleiche eines Sortierverfahrens im schlechtesten Fall. Diese Pfadlänge ergibt sich aus der Höhe des Entscheidungsbaums. Damit ist dann die untere Schranke für die Höhe solcher Entscheidungsbäume auch eine untere Schranke für die Anzahl der Vergleiche von vergleichenden Sortierverfahren.

Betrachte einen Entscheidungsbaum, der alle möglichen Ergebnisse eines vergleichenden Sortierverfahrens darstellt. Sei h seine Höhe und N die Anzahl seiner erreichbaren Blätter, die also jeweils ein Sortierergebnis repräsentieren. Da jede Permutation der n Elemente ein mögliches Sortierergebnis sein kann und ein Binärbaum der Höhe h nicht mehr als 2^h Blätter haben kann (Satz 5.1), gilt

$$n! \leq N \leq 2^h.$$

Daraus folgt

$$h \geq \mathrm{ld}\, n!$$

als minimale Höhe einens Entscheidungsbaums und damit als untere Schranke für vergleichende Sortierverfahren. Es bleibt diese untere Schranke asymptotisch abzuschätzen. Dazu werde die *Stirlingsche Formel* zur Abschätzung der Fakultät verwendet:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Es gilt

$$\begin{aligned}
&\mathrm{ld}\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right)\\
&= \mathrm{ld}\sqrt{2\pi} + \frac{1}{2}\mathrm{ld}\, n + n\mathrm{ld}\, n - n\mathrm{ld}\, e\\
&\geq n\mathrm{ld}\, n - n\mathrm{ld}\, e\\
&= n\mathrm{ld}\, n\left(1 - \frac{\mathrm{ld}\, e}{\mathrm{ld}\, n}\right)
\end{aligned}$$

Für $n \to \infty$ geht der Klammerausdruck gegen 1. Damit wird für ein $n_0 \in \mathbb{N}$ für alle $n \geq n_0$

$$n\mathrm{ld}\, n\left(1 - \frac{\mathrm{ld}\, e}{\mathrm{ld}\, n}\right) \geq \frac{1}{2}n\mathrm{ld}\, n \in \Omega(n \log n),$$

also

$$h \geq \mathrm{ld}\, n! \in \Omega(n \log n).$$

Damit ist der nachfolgende Satz 4.2 bewiesen.

Satz 4.2 (Minimale Sortierkomplexität)
Jeder vergleichende Sortieralgorithmus benötigt im schlechtesten Fall mindestens $\Omega(n \log n)$ *Vergleiche für* n *zu sortierende Elemente.*

Merge-Sort (Algorithmus 4.6) und *Quick-Sort* im mittleren Fall (Algorithmus 4.7) sind optimal, *Bubble-Sort* (Algorithmus 4.5) nicht.

4.9 Sortieren durch Abzählen

Unter Hinzunahme weiterer Voraussetzungen kann die im vorangehenden Abschnitt bestimmte untere Schranke für das Sortieren weiter herabgesetzt werden. So hat das *Sortieren durch Abzählen* (*Counting-Sort*) eine lineare Komplexität. Dieses Verfahren setzt voraus, dass jedes der zu sortierenden n Eingabeelemente eine ganze Zahl zwischen 0 und k ist und $k \in \Theta(n)$ gilt.

Das Verfahren bestimmt von jedem Element die Anzahl der jeweils kleineren Elemente und kann so das Element direkt an seine endgültige Position platzieren (Algorithmus 4.9). Es ist A das Eingabefeld, B das sortierte Ausgabefeld und C ist ein Hilfsfeld. Dieses Verfahren ist kein vergleichendes Sortierverfahren, an keiner Stelle werden zwei Elemente miteinander verglichen.

Algorithmus 4.9 (Counting-Sort)
Eingabe: $A = (a_1, \ldots, a_n)$ mit $0 \leq a_i \leq k, a_i, k \in \mathbb{N}_0, 1 \leq i \leq n$
Ausgabe: $B = (a_{\pi(1)}, \ldots, a_{\pi(n)})$ mit $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$

COUNTING-SORT(A, B, k)
```
 1  for j ← 0 to k
 2      do C[j] ← 0
 3  for i ← 1 to n
 4      do C[A[i]] ← C[A[i]] + 1
 5  ▷ C[j] enthält die Anzahl der Elemente, die gleich j sind.
 6  for j ← 1 to k
 7      do C[j] ← C[j] + C[j − 1]
 8  ▷ C[j] enthält die Anzahl der Elemente, die kleiner oder gleich j sind.
 9  for i ← n downto 1
10      do B[C[A[i]]] ← A[i]
11         C[A[i]] ← C[A[i]] − 1
12  return B
```

Das Element $C[j]$, $0 \leq j \leq k$, enthält unmittelbar nach Zeile 7 die Anzahl der Elemente, die kleiner oder gleich j sind, danach die Anzahl dieser Elemente, die noch zu platzieren sind. In der letzten Schleife (Zeilen 9-11) werden die Elemente $A[i]$, $i = n, n-1, \ldots, 1$, an die Position $C[A[i]]$ des Feldes B gesetzt, denn es gibt genau so viele Elemente, die kleiner oder gleich diesem Element sind. Danach wird diese Anzahl dekrementiert, denn dann werden kleinere oder gleiche Elemente davor in B platziert. Insbesondere wird ein gleiches Element unmittelbar vor diesen Platz gesetzt. Mit B wird ein aufsteigend sortiertes Zahlenfeld geliefert.

Beispiel 4.10
Eingabe: $A = (5, 3, 4, 7, 3, 2)$, $k = 9$

Zeile(n)	$C[0..9]$	$B[1..6]$	i	$B[C[A[i]]] \leftarrow A[i]$
5	$(0, 0, 1, 2, 1, 1, 0, 1, 0, 0)$			
8	$(0, 0, 1, 3, 4, 5, 5, 6, 6, 6)$	$(-, -, -, -, -, -)$		
9-11	$(0, 0, 0, 3, 4, 5, 5, 6, 6, 6)$	$(2, -, -, -, -, -)$	6	$B[C[2]] \leftarrow 2$
	$(0, 0, 0, 2, 4, 5, 5, 6, 6, 6)$	$(2, -, 3, -, -, -)$	5	$B[C[3]] \leftarrow 3$
	$(0, 0, 0, 2, 4, 5, 5, 5, 6, 6)$	$(2, -, 3, -, -, 7)$	4	$B[C[7]] \leftarrow 7$
	$(0, 0, 0, 2, 3, 5, 5, 5, 6, 6)$	$(2, -, 3, 4, -, 7)$	3	$B[C[4]] \leftarrow 4$
	$(0, 0, 0, 1, 3, 5, 5, 5, 6, 6)$	$(2, 3, 3, 4, -, 7)$	2	$B[C[3]] \leftarrow 3$
	$(0, 0, 0, 1, 3, 4, 5, 5, 6, 6)$	$(2, 3, 3, 4, 5, 7)$	1	$B[C[5]] \leftarrow 5$

Ausgabe: $B = (2, 3, 3, 4, 5, 7)$

Die Zeitkomplexität dieses Algorithmus ergibt sich aus der Anzahl der Schleifendurchläufe. Die grundlegenden Anweisungen sind die Zuweisungen in den Zeilen 2,4,7 und 11. Die Schleifen beginnend in den Zeilen 1 und 6 benötigen die Zeit $\Theta(k)$ und die Schleifen beginnend in den Zeilen 3 und 9 die Zeit $\Theta(n)$. Demnach benötigt der Algorithmus insgesamt eine Zeit von $\Theta(n + k)$. Für $k \in \Theta(n)$ ergibt sich eine Zeitkomplexität von $\Theta(n)$.

Bleibt festzustellen, dass das Sortieren durch Abzählen ein stabiles Sortierverfahren ist, aber es sortiert nicht *am Platz*.

Kapitel 5

Dynamische Datenstrukturen

5.1 Abstrakte Datentypen

Die elementaren Datentypen, wie z. B. *integer*, *real*, *char* oder *boolean*, sind die Grundbausteine, aus denen zusammengesetzte Datentypen aufgebaut werden können. Besteht ein zusammengesetzter Datentyp jeweils aus Objekten eines Datentyps, so spricht man von *Feldern* oder *arrays*, besteht er aus Objekten verschiedener Datentypen, von *Strukturen* oder *records*. Die zusammengesetzten Datentypen werden auch Datenstrukturen oder Datensätze genannt (Beispiel 4.1).

Man unterscheidet drei Arten von Datenstrukturen:

Statisch: Die Größe wird zur Compilezeit festgelegt und ist zur Laufzeit nicht mehr veränderbar. (z. B. elementare Datentypen, Strukturen, Felder, deren Größe zur Compilezeit festgelegt werden muss)

Halbdynamisch: Die Größe kann zur Laufzeit festgelegt werden, eine Änderung ist dann aber nur mit erheblichem Aufwand möglich. (z. B. Felder, deren Größe zur Laufzeit festgelegt werden kann)

Dynamisch: Die Größe kann zur Laufzeit beliebig vergrößert und verkleinert werden. (z. B. Listen, Bäume)

Die dynamischen Datenstrukturen sind sicherlich flexibler, aber sie benötigen auch einen höheren Verwaltungsaufwand und mehr Speicherplatz als statische Datenstrukturen. Bleibt die Anzahl der zu speichernden Objekte während der Laufzeit fest, so werden statische Strukturen verwendet, werden viele Objekte während der Laufzeit hinzugefügt oder gelöscht, so werden dynamische Strukturen verwendet.

Programmiertechnisch sind Datentypen zu spezifizieren und zu implementieren. Um ihre Spezifikation unabhängig von ihrer Implementierung zu halten, sind *abstrakte Datentypen* eingeführt worden.

Definition 5.1 (Abstrakter Datentyp)
Ein abstrakter Datentyp (ADT) besteht aus einer

- *Datenstruktur-Deklaration und einer*
- *Menge von Operationen auf dieser Datenstruktur.*

Mit der Deklaration wird der Wertebereich des Datentyps festgelegt, also die Werte, die eine Variable von diesem Datentyp annehmen kann. Mittels der darauf definierten Operationen (Funktionen, Prozeduren, Methoden) kann der Datentyp modifiziert oder abgefragt werden. Ein abstrakter Datentyp legt nur seine *Spezifikation* fest, d. h. Namen des Datentyps und der Operationen, Datentypen der Parameter der Operationen und ggf. der Rückgabe-Datentyp. Diese Angaben werden auch unter dem Begriff *Signatur* zusammengefaßt. Darüber hinaus ist die Semantik der Operationen zu beschreiben. Die *Implementierung*, also die programmtechnische Zusammensetzung aus anderen Datentypen oder die prozedurale Beschreibung, wie die Operationen ausgeführt werden, gehört nicht zu einem abstrakten Datentyp.

Ein Benutzer (*client*) eines ADTs ist irgend eine Prozedur, die außerhalb des ADTs definiert ist. Sie ruft eine ihrer Operationen mit einem Objekt (bzw. Instanz) des ADTs auf. ADTs sollen von außen nur einen wohldefinierten Zugriff auf seine privaten Daten erlauben (*Kapselung*). Der Benutzer soll zwar wissen was er tun, aber nicht wie es getan wird (*Geheimnisprinzip*). Die Verifikation einer einen ADT aufrufende Prozedur erfolgt auf Basis der Spezifikation des ADTs. Die Korrektheit seiner Implementierung ist garantiert. Die Komplexität einer solchen Prozedur ist aber von der Implementierung eines ADTs abhängig.

Die Operationen eines ADTs können in drei Gruppen eingeteilt werden:

- Konstruktoren: Sie erzeugen ein neues Objekt und liefern eine Referenz darauf.
- Zugriffsfunktionen: Sie liefern Informationen über ein Objekt
- Modifikatoren: Sie modifizieren ein Objekt

In Beispiel 5.1 ist ein abstrakter Datentyp spezifiziert, der die Datenstruktur und die Operationen zur Speicherung und Verarbeitung von Daten einer Person beschreibt. Die Semantik der Operationen ist informell in den angefügten Kommentaren beschrieben. Die erste Operation ist ein Konstruktor, die zweite eine Zugriffsfunktion und die dritte ein Modifikator.

Beispiel 5.1

ADT Person
▷ Daten
String *name*
String *vorname*
Datum *geburtsdatum*
...
▷ Operationen
Person CONSTRUCT(**String** *einName*, **String** *einVorname*, **Datum** *einDatum*)
▷ Konstruiert ein Objekt und liefert eine Referenz darauf
Datum GETDATE(**Person** *einePerson*)
▷ Liefert das Geburtsdatum einer Person
Void CHANGENAME(**Person** *einePerson*, **String** *einName*)
▷ Ändert den Namen einer Person
...

□

Ein abstrakter Datentyp heißt rekursiv, wenn eine seiner Zugriffsfunktionen den gleichen abstrakten Datentyp zurückliefert. Besonders betrachtet werden sollen im Folgenden die rekursiven ADTs *Listen* und *binäre Bäume*, und die diese ADTs benutzenden ADTs *Stapel* und *Warteschlangen*. Die Implementierungen von diesen abstrakten Datentypen als dynamische Datenstrukturen werden dann in den nächsten Abschnitten besprochen.

In Listen und in Bäumen werden Objekte zu komplexeren Strukturen verkettet. Diese Elemente bzw. Knoten halten die eigentlichen Daten. Das ein Objekt eindeutig identifizierende Datum ist der *Schlüssel* (*key*). In Beispiel 5.2 ist ein abstrakter Datentyp eines solchen Objekts spezifiziert.

Beispiel 5.2

ADT Element
▷ Daten
Integer *key*
...
▷ Operationen
Integer GETKEY( )
▷ Liefert den Schlüssel
...

□

Definition 5.2 (Liste)
Eine Liste ist eine sortierte oder unsortierte Folge von Elementen. Eine Liste kann leer sein, oder den folgenden Bedingungen genügen:

- *Es gibt ein besonderes Element, das erste Element, genannt Kopf.*
- *Die weiteren Elemente formen wieder eine Liste, genannt Rest.*

In Beispiel 5.3 ist ein abstrakter Datentyp einer Liste spezifiziert.

Beispiel 5.3

ADT Liste
▷ Daten

Element *data*
...
▷ Operationen
Liste CONSTRUCT( )
▷ Konstruiert eine leere Liste und liefert eine Referenz darauf
Element FIRST(**Liste** *eineListe*)
▷ Liefert das erste Element der Liste
Liste REST(**Liste** *eineListe*)
▷ Liefert die Liste hinter dem ersten Element
Element SEARCH(**Liste** *eineListe*, **Integer** *einKey*)
▷ Sucht und liefert das Element mit dem gegebenen Schlüssel
Void INSERT(**Liste** *eineListe*, **Element** *einElement*)
▷ Fügt das gegebene Element in die Liste ein
Void DELETE(**Liste** *eineListe*, **Element** *einElement*)
▷ Löscht das gegebene Element in der Liste
Void DELETE(**Liste** *eineListe*, **Integer** *key*)
▷ Sucht und löscht das Element mit dem gegebenen Schlüssel in der Liste
...

□

Definition 5.3 (Binärer Baum)
Eine binärer Baum (Binärbaum) ist eine Menge von Elementen, genannt Knoten. Ein binärer Baum kann leer sein, oder den folgenden Bedingungen genügen:

- *Es gibt ein besonderes Element, genannt Wurzel.*
- *Die weiteren Elemente sind in zwei disjunkte Teilmengen geteilt, jede formt wieder einen binären Baum. Eine Teilmenge wird linker Unterbaum, die andere wird rechter Unterbaum genannt.*

In Beispiel 5.4 ist ein abstrakter Datentyp eines binären Baumes spezifiziert.

Beispiel 5.4

ADT Baum
▷ Daten
Element *data*
...
▷ Operationen
Baum CONSTRUCT( )
▷ Konstruiert einen leeren Baum und liefert eine Referenz darauf
Element ROOT(**Baum** *einBaum*)
▷ Liefert die Wurzel des Baumes
Baum LEFT(**Baum** *einBaum*)
▷ Liefert den linken Unterbaum eines Baums
Baum RIGHT(**Baum** *einBaum*)
▷ Liefert den rechten Unterbaum eines Baums
Element SEARCH(**Baum** *einBaum*, **Integer** *einKey*)
▷ Sucht und liefert den Knoten mit dem gegebenen Schlüssel
Void INSERT(**Baum** *einBaum*, **Element** *einElement*)
▷ Fügt den gegebenen Knoten in den Baum ein
Void DELETE(**Baum** *einBaum*, **Element** *einElement*)
▷ Löscht den gegebenen Knoten in dem Baum
Void DELETE(**Baum** *einBaum*, **Integer** *key*)
▷ Sucht und löscht den Knoten mit dem gegebenen Schlüssel in dem Baum
...

□

Definition 5.4 (Stapel)
Ein Stapel bzw. Keller oder stack ist eine lineare Datenstruktur von Elementen, bei der das Lesen, Einfügen und Löschen von Elementen nur an einem Ende (top of stack) vorgenommen werden kann. Diese Modifikationsstrategie wird auch last-in first-out (LIFO) genannt.

In Beispiel 5.5 ist ein abstrakter Datentyp eines Stapels spezifiziert.

Beispiel 5.5

ADT Stapel
▷ Daten
Element *data*
...
▷ Operationen
Stapel CONSTRUCT( )
▷ Konstruiert einen leeren Stapel und liefert eine Referenz darauf
Element TOP(**Stapel** *einStapel*)
▷ Liefert das oberste Element des Stapels
Element POP(**Stapel** *einStapel*)
▷ Liefert das oberste Element des Stapels und löscht es
Void PUSH(**Stapel** *einStapel*, **Element** *einElement*)
▷ Fügt das gegebene Element als neues oberstes Element des Stapels ein
...

□

Definition 5.5 (Warteschlange)
Eine Warteschlange bzw. queue ist eine lineare Datenstruktur von Elementen, bei der das Einfügen von Elementen nur am Ende und das Lesen und Löschen nur am Anfang vorgenommen werden kann. Diese Modifikationsstrategie wird auch first-in first-out (FIFO) genannt.

In Beispiel 5.6 ist ein abstrakter Datentyp einer Warteschlange spezifiziert.

Beispiel 5.6

ADT Warteschlange
▷ Daten
Element *data*
...
▷ Operationen
Warteschlange CONSTRUCT( )
▷ Konstruiert eine leere Warteschlange und liefert eine Referenz darauf
Element FRONT(**Warteschlange** *eineSchlange*)
▷ Liefert das erste Element der Warteschlange
Element DEQUEUE(**Warteschlange** *eineSchlange*)
▷ Liefert das erste Element der Warteschlange und löscht es
Void ENQUEUE(**Warteschlange** *eineSchlange*, **Element** *einElement*)
▷ Fügt das gegebene Element als neues letztes Element der Warteschlange ein
...

□

Definition 5.6 (Prioritäts-Warteschlange)
Eine Prioritäts-Warteschlange bzw. priority queue ist eine sortierte Datenstruktur von Elementen, bei der das Einfügen von Elementen entsprechend der Sortierung und das Lesen und Löschen nur am Anfang vorgenommen werden kann.

In Beispiel 5.7 ist ein abstrakter Datentyp einer Prioritäts-Warteschlange spezifiziert.

Beispiel 5.7

ADT PrioSchlange
▷ Daten
Element *data*
...
▷ Operationen
PrioSchlange CONSTRUCT( )
▷ Konstruiert eine leere Prioritäts-Warteschlange und liefert eine Referenz darauf
Element GETMAX(**PrioSchlange** *eineSchlange*)
▷ Liefert das erste Element der Prioritäts-Warteschlange
Element DELETE(**PrioSchlange** *eineSchlange*)
▷ Liefert das erste Element der Prioritäts-Warteschlange und löscht es
Void INSERT(**PrioSchlange** *eineSchlange*, **Element** *einElement*)
▷ Fügt das gegebene Element sortiert in die Prioritäts-Warteschlange ein
...

□

Die oben eingeführten abstrakten Datentypen lassen sich alle mittels Listen und Bäumen implementieren, was auch in den nächsten Abschnitten dargestellt wird. Ein *Lexikon* (*dictionary*) Datentyp wird in der Regel weder über Listen noch über Bäume implementiert. Da dieser abstrakte Datentyp aber im Kapitel 6 benötigt wird, soll er hier bereits eingeführt werden.

Definition 5.7 (Lexikon)
Ein Lexikon bzw. dictionary ist eine Datenstruktur zum Speichern und Verwalten von Elementen. Die Elemente bestehen aus einem Schlüssel (key) und den weiteren Daten. Auf sie wird ausschließlich über ihren Schlüssel zugegriffen.

In Beispiel 5.8 ist ein abstrakter Datentyp eines Lexikons spezifiziert.

Beispiel 5.8

ADT Lexikon
▷ Daten
Element *data*
...
▷ Operationen
Lexikon CONSTRUCT( )
▷ Konstruiert ein leeres Lexikon und liefert eine Referenz darauf
Boolean MEMBER(**Lexikon** *einLexikon*, **Integer** *einKey*)
▷ Liefert true, falls das Element in dem Lexikon enthalten ist, sonst false
Element RETRIEVE(**Lexikon** *einLexikon*, **Integer** *einKey*)
▷ Liefert das Element mit dem gegebenen Schlüssel
Void INSERT(**Lexikon** *einLexikon*, **Element** *einElement*)
▷ Fügt das gegebene Element in das Lexikon ein
Void DELETE(**Lexikon** *einLexikon*, **Integer** *einKey*)
▷ Löscht das Element mit dem gegebenen Schlüssel in dem Lexikon
...

□

5.2 Verkettete Listen

Eine grundlegende dynamische Datenstruktur ist eine *verkettete Liste*, in der die einzelnen zu speichernden Objekte linear angeordnet sind. Jedes Objekt der Liste hält seine Daten, ggf. einschließlich seinen Schlüssel und einen *Zeiger* (*pointer*, *reference*) auf das nachfolgende Objekt, den *Nachfolger*. Man spricht auch von den *Elementen* einer Liste. Ein Zeiger, der *Anker*, muss auf das erste Element der Liste zeigen, auf den *Kopf* (*head*) der Liste. Alles hinter dem Kopf wird *Rest* (*tail*) genannt. Eine verkettete Liste implementiert den abstrakten Datentyp *Liste* (Definition 5.2). Gegenüber Beispiel 5.3 sind hier die je Element zu speichernden Daten auf den Schlüssel reduziert. In Abbildung 5.1 ist eine verkettete Liste dargestellt.

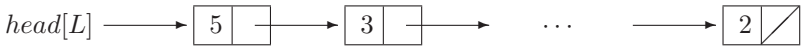


Abbildung 5.1: Verkettete Liste L

Werde die verkettete Liste mit L bezeichnet, dann ist $head[L]$ ein Zeiger auf den Kopf der Liste. Sei x ein beliebiges Element der Liste, genauer ein Zeiger auf ein beliebiges Element der Liste, dann ist $next[x]$ ein Zeiger auf den Nachfolger von x. Das letzte Listenelement hat keinen Nachfolger. Es gilt dann $next[x] =$NIL, der Zeiger verweist auf eine leere Liste. Auf den Schlüssel, als Repräsentanten der Daten, des Elements x wird über $key[x]$ zugegriffen.

Verketteten Listen können neben der Nachfolger-Verkettung auch eine Vorgänger-Verkettung aufweisen. In Abbildung 5.2 ist eine solche doppelt verkettete Liste dargestellt. Jedes Element weist dann zusätzlich einen Zeiger $prev[x]$ auf den jeweiligen Vorgänger auf.

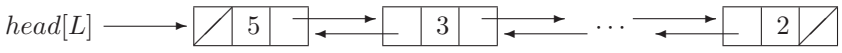


Abbildung 5.2: Doppelt verkettete Liste L

Verkettete Listen können nach ihren Schlüsseln sortiert oder unsortiert vorliegen. Die eigentlichen Daten können direkt in den Listenelementen gespeichert sein oder in davon unabhängigen Datenstrukturen, auf die dann über je einen weiteren Zeiger je Element verwiesen wird.

Die Implementierungen der Zugriffsfunktionen des ADT *Liste* auf Basis doppelt verketteter, unsortierter Listen sind in den folgenden Algorithmen dargestellt, der Zugriff auf das erste Element in Algorithmus 5.1, der Zugriff auf den Rest der Liste in Algorithmus 5.2, das Suchen eines Elements in Algorithmus 5.3, das Einfügen eines Elements in 5.4 und das Löschen eines Elements in Algorithmus 5.5.

Algorithmus 5.1 (List-First)
Eingabe: doppelt verkettete Liste L
Ausgabe: Erstes Element, falls dieses existiert,
sonst NIL

FIRST(L)
1 **return** $head[L]$

Zur Bestimmung der Zeitkomplexitäten sind wieder die grundlegenden Anweisungen in den Algorithmen zu zählen und diese asymptotisch in Abhängigkeit von der Eingabegröße darzustellen. Die grundlegenden Anweisungen sind auch hier wieder die Schlüsselvergleiche. Die Komplexitäten von *First* und *Rest* sind konstant. *Search* hat bei einer verketteten Liste der Länge n im schlechtesten

Algorithmus 5.2 (List-Rest)
Eingabe: doppelt verkettete Liste L
Ausgabe: Rest der Liste

```
REST(L)
1  if L ≠ NIL
2     then return next[head[L]]
3     else return NIL
```

Algorithmus 5.3 (List-Search)
Eingabe: doppelt verkettete, unsortierte Liste L, Schlüssel k
Ausgabe: Zeiger x auf das Element mit $key[x] = k$, falls dieser existiert,
sonst $x =$ NIL

```
SEARCH(L, k)
1  x ← head[L]
2  while (x ≠ NIL) ∧ (key[x] ≠ k)
3     do x ← next[x]
4  return x
```

Fall eine Komplexität von $\Theta(n)$. Insbesondere muss die gesamte Liste durchsucht werden, wenn das gesuchte Element gar nicht darin enthalten ist.

Algorithmus 5.4 (List-Insert)
Eingabe: doppelt verkettete, unsortierte Liste L,
Zeiger x auf das einzufügende Element
Ausgabe: keine

```
INSERT(L, x)
1  next[x] ← head[L]
2  if head[L] ≠ NIL
3     then prev[head[L]] ← x
4  head[L] ← x
5  prev[x] ← NIL
```

Das Einfügen erfolgt am Kopf der Liste. *Insert* hat also eine Komplexität von $\Theta(1)$, denn der Aufwand ist unabhängig von der Länge der Liste.

Auch das Löschen weist eine konstante Komplexität auf, also $\Theta(1)$. Soll allerdings ein Element eines gegebenen Schlüssels k gelöscht werden, dann ist vorab ein Zeiger auf das gesuchte Element zu setzen, und das Suchen dieses Elements hat im schlechtesten Fall eine Komplexität von $\Theta(n)$, die sich dann auch auf das Löschen überträgt. Abbildung 5.3 zeigt die zu modifizierenden Zeiger beim Einfügen und Löschen.

Bei sortierten verketteten Listen sind der Such-Algorithmus und der Einfüge-Algorithmus leicht zu modifizieren. Das Suchen kann abgebrochen werden, wenn der Schlüssel des aktuell referenzierten Elementes erstmals größer ist als der gesuchte Schlüssel. Das einzufügende Element wird nicht am Kopf der Liste eingefügt, sondern an der entsprechenden Stelle in die Verkettung eingehängt. Durch die vorangehende Suche der Einfügeposition erhöht sich die Komplexität auf $\Theta(n)$. Der Lösch-Algorithmus bleibt unverändert.

Es sei darauf hingewiesen, dass der zusätzliche Speicherbedarf gegenüber einer Speicherung in Feldern für einfach und auch doppelt verkettete Listen $\Theta(n)$ ist, denn je Element sind zusätzlich ein bzw. zwei Zeiger zu speichern.

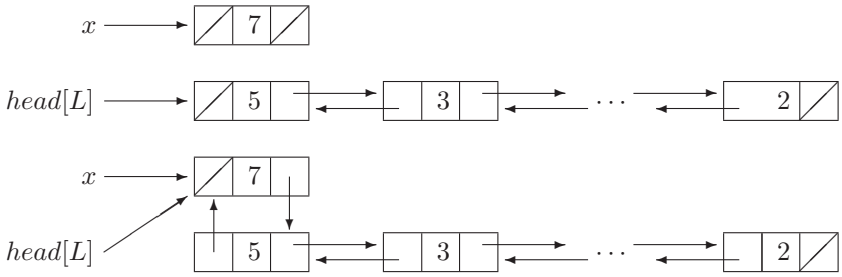
Die im Kapitel 4 betrachteten Such- und Sortierverfahren, bei denen ein direkter Zugriff auf die Elemente über ihren Index erfolgt, sind nicht übertragbar, da verkettete Listen direkte Zugriffe nicht erlauben. Allerdings Verfahren mit sequentiellem Zugriff auf die Elementen können leicht übertragen werden.

Algorithmus 5.5 (List-Delete)
Eingabe: doppelt verkettete Liste L,
Zeiger x auf das zu löschende Element
Ausgabe: keine

```
DELETE(L, x)
1  if prev[x] ≠ NIL
2     then next[prev[x]] ← next[x]
3     else head[L] ← next[x]
4  if next[x] ≠ NIL
5     then prev[next[x]] ← prev[x]
```

Abschließend soll noch darauf hingewiesen werden, dass sich auch die abstrakten Datentypen *Stapel*, *Warteschlange* und *Prioritäts-Warteschlange* sehr gut über die dynamische Datenstruktur einer einfach verketteten Liste implementieren lassen. Für Warteschlangen wird dann ein zusätzlicher Zeiger auf das Ende der Liste bereitgestellt.

Einfügen:



Löschen:

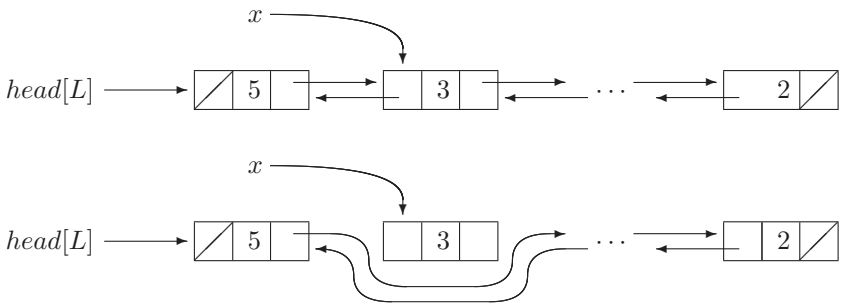


Abbildung 5.3: Einfügen und Löschen von Listenelementen

5.3 Binäre Bäume

Ein *binärer Baum*, oder genauer *binärer Wurzelbaum*, ist ein gerichteter Graph mit genau einem Knoten ohne Vorgänger, der *Wurzel*. Jeder Knoten hat höchstens zwei Nachfolger, einen linken und einen rechten. Ein Knoten ohne Nachfolger heißt *Blatt* (Abbildung 5.4). Mittels der Datenstruktur eines binären Baumes kann nun der abstrakte Datentyp *binärer Baum* (Definition 5.3, Beispiel 5.4) implementiert werden. Die je Knoten zu speichernden Daten sind wieder auf den Schlüssel reduziert.

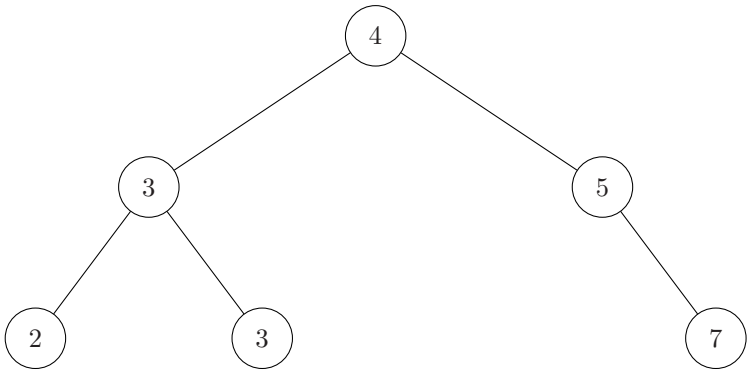


Abbildung 5.4: Binärer Baum T

Werde der binäre Baum mit T bezeichnet, dann ist $root[T]$ ein Zeiger auf die Wurzel des Baumes. Sei x einen beliebiger Knoten des Baumes, genauer ein Zeiger auf einen beliebigen Knoten. Mit $left[x]$ wird auf den linken Nachfolger verwiesen, mit $right[x]$ auf den rechten Nachfolger. Die jeweiligen Nachfolger sind die Wurzel eines so genannten Unterbaumes, und somit spricht man auch vom rechten und linken Unterbaum. Unterbäume können auch leer sein, d. h. ein Knoten hat dann keinen linken bzw. rechten Nachfolger und der Wert der jeweiligen Zeiger ist NIL. Für viele Anwendungen kann es auch sinnvoll sein, eine Vorgänger-Verkettung $p[x]$ einzuführen. Für die Wurzel gilt $p[root[T]] =$NIL, für Blätter x gelten $left[x] =$NILund $right[x] =$NIL. Auf den Schlüssel eines Knotens x wird über $key[x]$ zugegriffen. Auf weitere in oder mit den Knoten zu speichernde Daten soll hier verzichtet werden.

Ein Knoten eines Baumes läßt sich graphisch analog zu Listenelementen darstellen (Abbildung 5.5). Bei einer Wurzel oder Blättern sind die jeweiligen Zeiger auf NIL gesetzt.

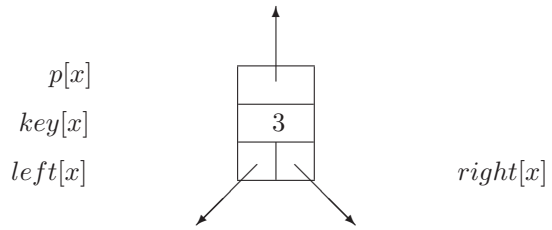


Abbildung 5.5: Knoten x eines Baumes

Ein Baum wird auch in *Ebenen* eingeteilt. Alle Knoten mit der gleichen Entfernung von der Wurzel (Anzahl an Nachfolger-Verweisen von der Wurzel) werden einer Ebene zugeordnet. Die Wurzel hat die Ebene 0. Ein Knoten der Ebene n heißt auch Knoten der *Tiefe* n. Die maximale Tiefe eines Baumes wird als *Höhe* des Baumes bezeichnet. Der Baum in Abbildung 5.4 hat die Höhe 2.

Knotenanzahl und Höhe von binären Bäumen werden als Eingabegröße für die Komplexitätsanalyse von Implementierungen herangezogen. Deshalb ist es sehr hilfreich, wenn einige Zusammenhänge über diese Größen in Binärbäumen bekannt sind. Betrachte dazu die nachfolgenden Sätze.

Satz 5.1 (Knotenanzahl je Ebene)
In einem Binärbaum gibt es auf der Ebene d höchstens 2^d Knoten, $d \in \mathbb{N}_0$.

Beweis:
Induktionsanfang $d = 0$:
Ein Binärbaum besteht auf der Ebene 0 nur aus der Wurzel. Die Knotenanzahl ist somit genau $1 = 2^0$, also auch höchstens 1.

Induktionsannahme:
Sei die Behauptung für ein $d \geq 0$ richtig.

Induktionsschritt $d \to d + 1$:
Bestimme die Knotenanzahl auf Ebene $d + 1$: Nach der Induktionsannahme sind auf der Ebene d höchstens 2^d Knoten. Jeder dieser Knoten hat höchstens 2 Nachfolger. Somit gibt es auf der Ebene $d + 1$ höchstens $2 \cdot 2^d = 2^{d+1}$ Knoten, womit die Behauptung bewiesen ist. □

Satz 5.2 (Knotenanzahl eines Binärbaumes)
Ein Binärbaum der Höhe h hat höchstens $2^{h+1} - 1$ Knoten, $h \in \mathbb{N}_0$.

Beweis:
Nach Satz 5.1 sind in einem Binärbaum auf Ebene d höchstens 2^d Knoten. Ein Binärbaum der Höhe h besteht aus den Ebenen $d = 0, 1, \ldots, h$. Somit gibt es höchstens

$$\begin{aligned} & 1 + 2 + 2^2 + \ldots + 2^h \\ = \; & \sum_{i=0}^{h} 2^i \qquad \text{(geometrische Reihe)} \\ = \; & 2^{h+1} - 1 \end{aligned}$$

Knoten in einem Binärbaum. □

Satz 5.3 (Höhe eines Binärbaumes)
Ein Binärbaum mit n Knoten hat mindestens eine Höhe von $\lfloor \mathrm{ld}\, n \rfloor$, $n \in \mathbb{N}$.

Beweis:
Nach Satz 5.2 hat ein Binärbaum der Höhe h höchstens $2^{h+1} - 1$ Knoten. Die folgenden Relationen sind äquivalent:

$$\begin{aligned} 2^{h+1} - 1 &\geq n \\ 2^{h+1} &\geq n + 1 \\ h + 1 &\geq \mathrm{ld}\,(n + 1) \\ h &\geq \mathrm{ld}\,(n + 1) - 1 \end{aligned}$$

Somit hat ein Binärbaum mit n Knoten mindestens die Höhe

$$\lceil \mathrm{ld}\,(n + 1) \rceil - 1 = \lfloor \mathrm{ld}\, n \rfloor .$$

□

Es werden nun zunächst einmal die Implementierungen der Zugriffsfunktionen *Root*, *Left* und *Right* betrachtet (Algorithmen 5.6, 5.7, 5.8), die die Wurzel, den linken bzw. den rechten Nachfolger der Wurzel eines binären Baumes liefern.

Algorithmus 5.6 (Tree-Root)
Eingabe: Binärer Baum T
Ausgabe: Wurzel von T

ROOT(T)
1 **return** $root[T]$

Algorithmus 5.7 (Tree-Left)
Eingabe: Binärer Baum T
Ausgabe: Linker Nachfolger der Wurzel von T

LEFT(T)
1 **if** $T \neq$ NIL
2 **then return** $left[root[T]]$
3 **else return** NIL

Algorithmus 5.8 (Tree-Right)
Eingabe: Binärer Baum T
Ausgabe: Rechter Nachfolger der Wurzel von T

RIGHT(T)
1 **if** $T \neq$ NIL
2 **then return** $right[root[T]]$
3 **else return** NIL

Die Laufzeit dieser Zugriffsfunktionen ist weder von der Knotenanzahl noch von der Höhe des Baumes abhängig. Sie haben also eine konstante Zeitkomplexität.

Bei der Suche nach einem bestimmten Schlüssel in einem binären Baum, so wie er bisher definiert ist, müsste im schlechtesten Fall der gesamte Baum durchsucht und jeder Knoten mit dem Schlüssel verglichen werden. Das ergäbe die gleiche Komplexität wie bei verketteten Listen, nämlich $\Theta(n)$. Einem höheren Verwaltungsaufwand bei Bäumen stünde keine Effizienzverbesserung gegenüber. Deshalb ist es sinnvoll eine Ordnung auf einem Baum zu definieren, ähnlich einer Sortierung bei verketteten Listen (Definition 5.8).

Definition 5.8 (Binärer Suchbaum)
Ein binärer Suchbaum ist ein binärer Baum mit der folgenden Eigenschaft. Es sei x ein beliebiger Knoten des binären Suchbaums.

- *Ist y ein beliebiger Knoten aus dem linken Unterbaum von x, dann gilt $key[y] < key[x]$.*
- *Ist y ein beliebiger Knoten aus dem rechten Unterbaum von x, dann gilt $key[y] \geq key[x]$.*

Für den Baum in Abbildung 5.4 ist diese Eigenschaft erfüllt. Es handelt sich um einen binären Suchbaum. Aus einem solchen Baum kann nun leicht eine nach Schlüsseln sortierte Liste der Knoten erstellt werden. Auch das Suchen nach Knoten eines bestimmten Schlüssels ist wesentlich effizienter durchführbar als in linear verketteten Listen. Aber beim Einfügen und Löschen eines Knotens ist etwas mehr Aufwand nötig, da ja die obige Eigenschaft erhalten bleiben muss.

Zunächst aber zur Erstellung einer sortierten Liste der Knoten (Algorithmus 5.9). Der Baum wird durchschritten (traversiert) indem zunächst solange zum linken Nachfolger weiter gegangen wird, bis es keinen mehr gibt, d. h. der dann aktuelle Unterbaum leer ist. Die folgende Rückkehr um eine Rekursionsstufe bedeutet Rückkehr zum Elternknoten. Dieser wird ausgegeben und dann der rechte Unterbaum durchschritten. Nach Rückkehr von einem rechten Unterbaum erfolgt unmittelbar eine weitere Rückkehr um eine Rekursionsstufe. Aus der Eigenschaft eines binären Suchbaumes folgt unmittelbar, dass diese Art der Traversierung und des Ausgebens eine aufsteigende Sortierung der Schlüssel ergibt. Aus jedem Knoten erfolgen dabei zwei rekursive Prozeduraufrufe, für den linken

Algorithmus 5.9 (Inorder-Tree-Walk)
Eingabe: Binärer Suchbaum T
Ausgabe: keine

```
INORDER-TREE-WALK(T)
1  x ← root[T]
2  if x ≠ NIL
3     then INORDER-TREE-WALK(left[x])
4          print key[x]
5          INORDER-TREE-WALK(right[x])
```

Unterbaum und für den rechten Unterbaum. Bei n Knoten ergibt sich daraus eine Komplexität von $\Theta(n)$.

Der Algorithmus *Search* (Algorithmus 5.10) implementiert die Suche nach einem Schlüssel k in einem binären Suchbaum. Aufgrund seiner Eigenschaft bzgl. der Schlüsselanordnungen kann genau der Pfad gewählt werden, auf dem k liegen muss, wenn er denn überhaupt in dem Baum enthalten ist. Ist er gefunden, so bricht das Verfahren ab, und es wird ein Zeiger auf den entsprechenden Knoten gelierfert. Ist der Knoten nicht in dem Baum, bricht das Verfahren erst ab, nachdem es bei einem leeren Unterbaum angekommen ist.

Algorithmus 5.10 (Tree-Search)
Eingabe: Binärer Suchbaum T, Schlüssel k
Ausgabe: Zeiger x auf das Element mit $key[x] = k$, falls dieser existiert,
sonst $x =$ NIL

```
SEARCH(T, k)
1  x ← root[T]
2  if (x = NIL) ∨ (k = key[x])
3     then return x
4  if k < key[x]
5     then return SEARCH(left[x], k)
6     else return SEARCH(right[x], k)
```

Seien n die Knotenanzahl und h die Höhe des binären Suchbaumes. Die Komplexität der Suche kann dann im schlechtesten Fall mit $\Theta(h)$ abgeschätzt werden. Der schlechtest Fall tritt ein, wenn der Schlüssel nicht im Suchbaum enthalten ist und er auf dem längsten Pfad gesucht wurde. Da der Baum außerdem zu einer verketteten Liste degeneriert sein kann – jeder Knoten hat höchstens einen Nachfolger – würde in diesem Fall $h = n - 1$ gelten, und die Komplexität wäre wieder nur $\Theta(n)$.

Ein *vollständiger* binärer Baum (Definition 5.9) dagegen hat nach Satz 5.3 nur eine Höhe von $h = \lfloor \mathrm{ld}\, n \rfloor$. Bei einem solchen Baum hat die Suche im schlechtesten Fall nur eine Komplexität von $\Theta(\log n)$.

Definition 5.9 (Vollständiger binärer Baum)
Ein vollständiger binärer Baum ist ein binärer Baum, bei dem alle Blätter auf einer Ebene liegen und bei dem alle inneren Knoten genau zwei Nachfolger haben.

Ein vollständiger binärer Baum (Abbildung 5.6) weist zu einer gegebenen Höhe h eine maximale Knotenanzahl auf, nämlich $2^{h+1} - 1$ Knoten. Darunter sind 2^h Blätter, alle auf der Ebene h. Umgekehrt hat ein vollständiger binärer Baum aus n Knoten eine Höhe von $h = \mathrm{ld}\,(n+1) - 1 = \lfloor \mathrm{ld}\, n \rfloor$ (Sätze 5.1, 5.2, 5.3).

Weitere Implementierungen des ADTs *binärer Baum* betreffen das Einfügen (Algorithmus 5.11) und Löschen (Algorithmus 5.12) von Knoten. Hierbei ist zu beachten, dass dabei die Eigenschaft eines binären Suchbaums erhalten bleibt.

Der Algorithmus zum Einfügen durchsucht zunächst den Baum wie bei der Suche nach einem vorgegebenen Schlüssel, hier $key[z]$, nach einer Position zum Einfügen (Zeilen 3-7). Die Suche

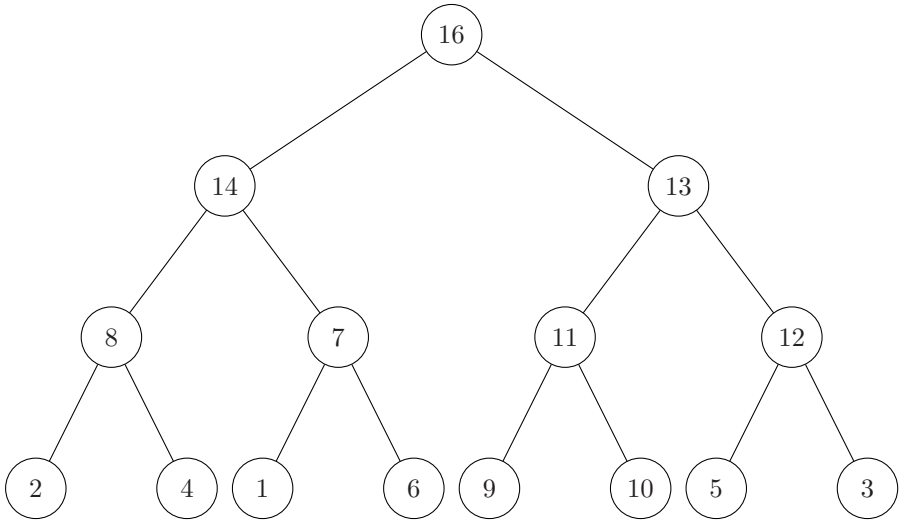


Abbildung 5.6: Vollständiger binärer Baum

Algorithmus 5.11 (Tree-Insert)
Eingabe: binärer Suchbaum T, einzufügender Knoten z
Ausgabe: keine

```
INSERT(T, z)
 1  y ← NIL
 2  x ← root[T]
 3  while x ≠ NIL
 4     do y ← x
 5        if key[z] < key[x]
 6           then x ← left[x]
 7           else x ← right[x]
 8  p[z] ← y
 9  if y = NIL
10     then root[T] ← z
11     else if key[z] < key[y]
12             then left[y] ← z
13             else right[y] ← z
```

ist erst beendet, wenn der Zeiger x auf einen leeren Unterbaum zeigt. Sollte der einzufügende Schlüssel bereits in dem Baum enthalten sein, so wird das hier nicht beachtet, denn in einem binären Suchbaum dürfen gleiche Schlüssel mehrfach enthalten sein.

Mit dem Zeiger x durchläuft ein weiterer Zeiger y den Baum. Er zeigt immer auf den Vorgänger von x. Also bei Erreichen der Abbruchbedingung $x = \text{NIL}$ zeigt y auf den entsprechenden Vorgänger. y ist nun der Knoten, wo der neue Knoten als Nachfolger eingefügt werden soll. Damit ist y der Vorgänger von z (Zeile 8). Der Knoten y muss z als linken oder rechten Nachfolger bekommen, mindestens eine der beiden Positionen war ja auch leer (Zeilen 11-13). Bleibt der Spezialfall, dass T leer ist, zu betrachten. Dann wird z die Wurzel (Zeile 10). Der eingefügte Knoten z ist immer ein Blatt.

Bei einer Höhe h des Baumes liegt die Komplexität im schlechtesten Fall wieder bei $\Theta(h)$, denn das Einfügen erfolgt immer als ein Blatt. Das Verhältnis von h zu n hängt wieder davon ab, ob der Baum (annähernd) vollständig ist oder nicht. In dem Baum T aus Abbildung 5.4 ist ein Knoten mit dem Schlüssel 5 einzufügen. Abbildung 5.7 zeigt T nach dem Einfügen.

Das Löschen eines Elements kann, je nach Position des zu löschenden Elements, Umstrukturierungen des Baumes erforderlich machen, da sonst die Eigenschaft eines binären Suchbaums verletzt

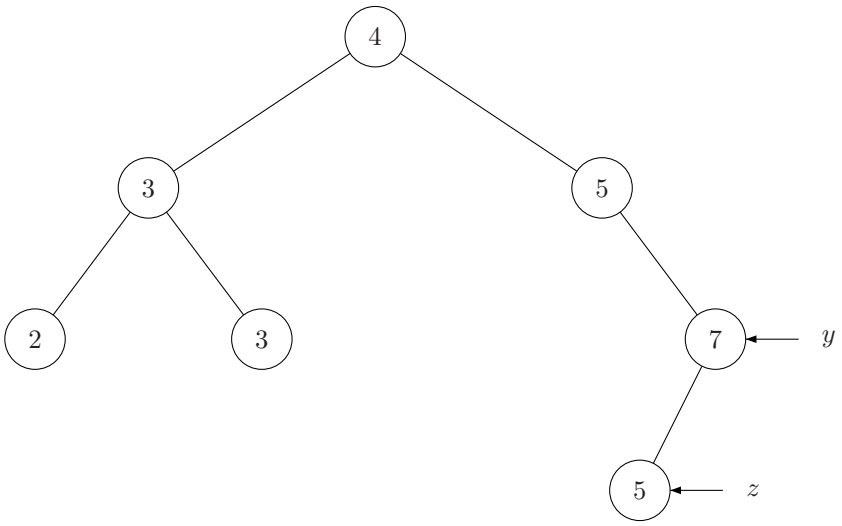


Abbildung 5.7: Binärer Baum T mit eingefügtem Knoten z

sein könnte (Algorithmus 5.12). Betrachte dazu Abbildung 5.8:

a) Der zu löschende Knoten hat keine Nachfolger, er wird einfach gelöscht.

b) Der zu löschende Knoten hat einen Nachfolger, er wird ausgeschnitten.

c) Der zu löschende Knoten hat zwei Nachfolger. Sein *Baum-Nachfolger* wird ausgeschnitten und die Daten des Baum-Nachfolgers werden an den eigentlich zu löschenden Knoten übertragen.

Die Prozedur *Tree-Successor* aus Algorithmus 5.12 liefert zu einem gegebenen Knoten denjenigen zurück, der bei einer Infix-Traversierung des Baumes (*Inorder-Tree-Walk*) der unmittelbar nachfolgende wäre, genannt *Baum-Nachfolger*. Das ist immer ein Knoten, bei dem mindestens ein Nachfolger leer ist. Bei einem Baum mit paarweise verschiedenen Schlüsseln wird also der Knoten mit dem nächst größeren Schlüssel geliefert. In Abbildung 5.8c ist das der Knoten y. Aus dem Baum wird dann der Baum-Nachfolger entfernt. Dazu sind vorab sein Zeiger auf den Nachfolger – er hat höchstens einen – und sein Schlüssel in den Knoten, der eigentlich zu löschen wäre, umzukopieren.

Für Bäume der Höhe h hat auch das Löschen im schlechtesten Fall eine Komplexität von $\Theta(h)$. Auch hier sei darauf hingewiesen, dass das Verhältnis von h zu n davon abhängt, ob der Baum (annähernd) vollständig ist oder nicht.

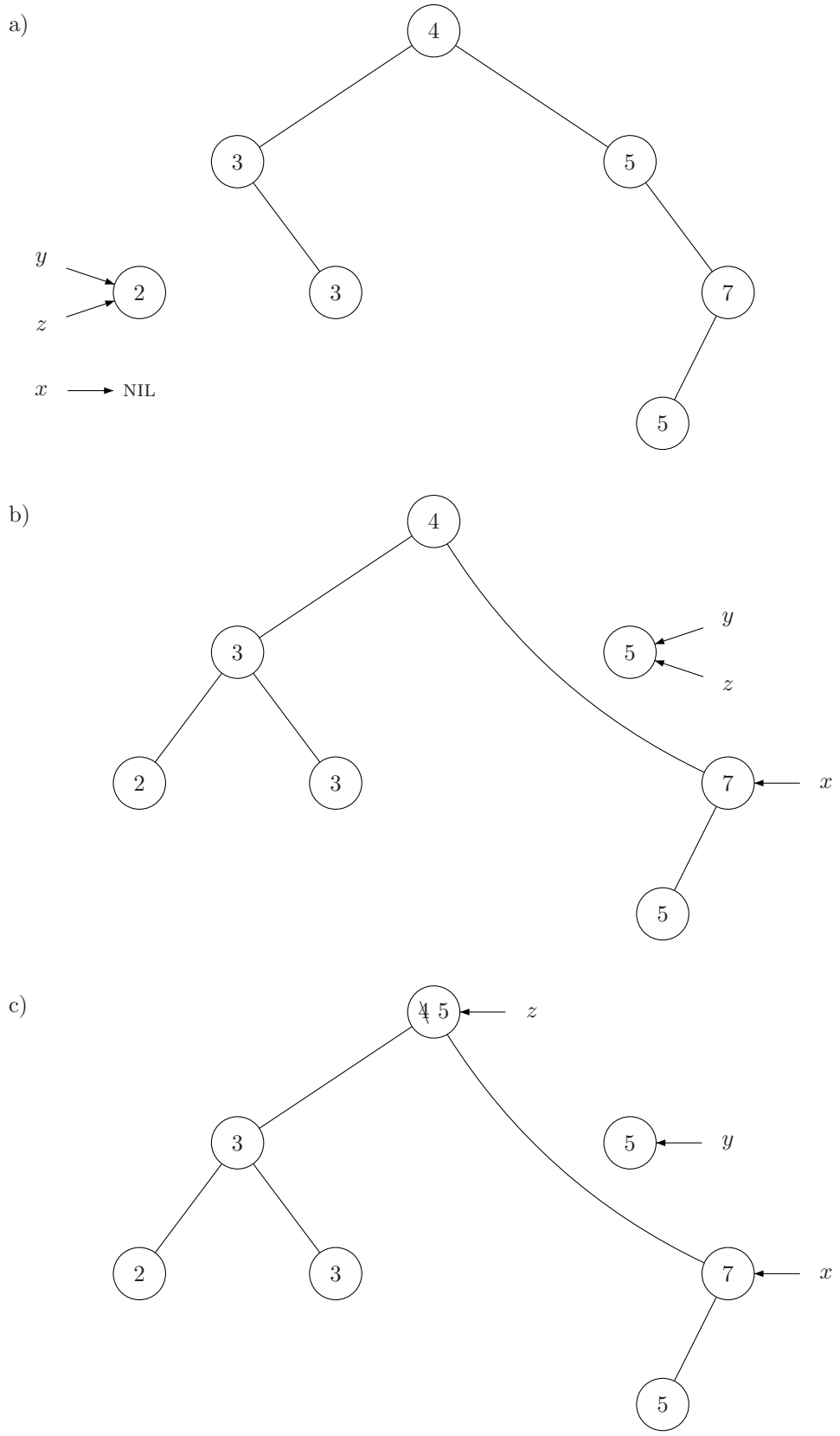


Abbildung 5.8: Binärer Baum T nach Löschen des Knotens z

Algorithmus 5.12 (Tree-Delete)
Eingabe: binärer Suchbaum T, zu löschender Knoten z
Ausgabe: keine

```
TREE-DELETE(T, z)
 1  if (left[z] = NIL) ∨ (right[z] = NIL)
 2     then y ← z
 3     else y ← TREE-SUCCESSOR(z)
 4  if left[y] ≠ NIL
 5     then x ← left[y]
 6     else x ← right[y]
 7  if x ≠ NIL
 8     then p[x] ← p[y]
 9  if p[y] = NIL
10     then root[T] ← x
11     else if y = left[p[y]]
12             then left[p[y]] ← x
13             else right[p[y]] ← x
14  if y ≠ z
15     then key[z] ← key[y]

TREE-SUCCESSOR(x)
 1  if right[x] ≠ NIL
 2     then return TREE-MINIMUM(right[x])
 3  y ← p[x]
 4  while (y ≠ NIL) ∧ (x = right[y])
 5     do x ← y
 6        y ← p[y]
 7  return y
```

5.4 Binäre Heaps

Im vorangehenden Abschnitt sind die unterschiedlichen Komplexitäten diskutiert worden, die bei den Implementierungen einiger Zugriffsfunktionen von binären Bäumen auftreten können. Sie weisen in Abhängigkeit von der Knotenanzahl eine Spannbreite von logarithmisch bis zu linear auf, je nachdem, ob der Baum vollständig ist, überwiegend Knoten mit zwei Nachfolgern oder nur wenige bis gar keine Knoten mit zwei Nachfolgern hat. Anzustreben ist natürlich immer die Vollständigkeit der Knoten in einem Baum, da dann ein sehr effizienter Zugriff auf die Knoten in dem Baum möglich ist. Die Datenstruktur eines *binären Heaps* sichert diese Vollständigkeit so gut es geht zu.

Definition 5.10 (Binärer Heap)
Ein binärer Baum heißt binärer Heap oder kurz Heap, wenn die folgenden Eigenschaften erfüllt sind:

- *Der Baum ist mindestens bis zur Tiefe $h-1$ vollständig.*
- *Alle Blätter befinden sich auf der Tiefe $h-1$ oder h.*
- *Alle Pfade zu den Blättern der Tiefe h befinden sich links von den Pfaden, die zu den Blättern der Tiefe $h-1$ führen.*

Definition 5.11 (Min-Heap, Max-Heap)
Ein binärer Heap heißt Min-Heap bzw. Max-Heap, wenn der Schlüssel eines jeden Knotens kleiner oder gleich bzw. größer oder gleich aller Knoten in den jeweiligen Unterbäumen ist.

Für jeden Knoten und seinen Unterbaum gilt also, dass in einem Max-Heap das größte Element und in einem Min-Heap das kleinste Element des Unterbaums jeweils in der Wurzel des Unterbaums gespeichert ist. Für den gesamten Baum bedeutet das, das größte bzw. das kleinste Element befindet sich in der Wurzel des Baumes. Im Folgenden werden Implementierungen von Max-Heaps betrachtet. Ein Beispiel eines Max-Heaps ist in Abbildung 5.9 dargestellt.

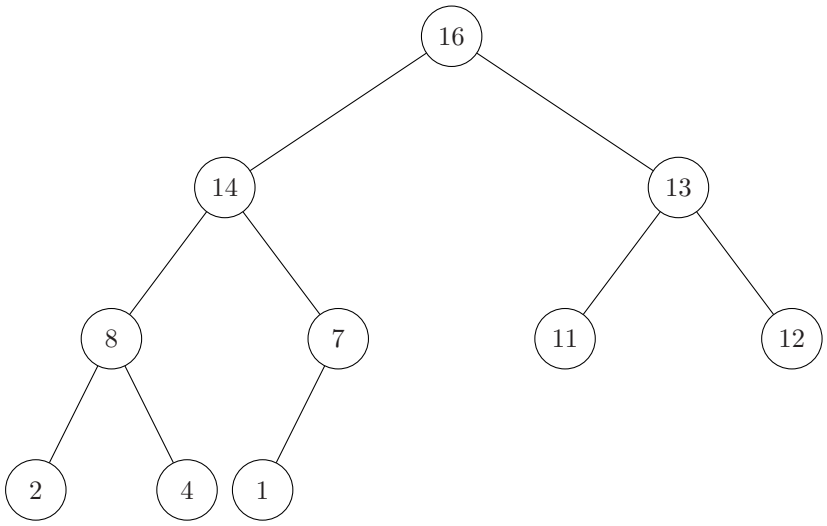


Abbildung 5.9: Binärer Max-Heap

Ein Heap kann nun natürlich, wie im vorangehenden Abschnitt beschrieben, als eine dynamische Datenstruktur eines binären Baumes implementiert werden. Es ist allerdings auch möglich, einen Heap in einer statischen Datenstruktur eines Feldes zu implementieren. Die Anzahl der zu speichernden Knoten ist dann aber mit der Feldgröße nach oben beschränkt. Dieser Ansatz wird im Folgenden verfolgt. Die graphischen Darstellungen der Beispiele erfolgen aber weiterhin als Bäume, da darüber die Strukturen des Heaps und die Beziehungen unter den Knoten deutlicher hervortreten.

Bei einer Implementierung eines Heaps in einem Feld werden die Knoten in dem Feld bezogen auf die Baumstruktur ebenenweise (von oben nach unten und von links nach rechts) angeordnet.

Sei $A = (a_1, \ldots, a_n)$ ein Feld der Größe n. Die $a_i \in \mathbb{N}$, $1 \leq i \leq n$, repräsentieren die Schlüssel der Knoten eines Heaps. Die Baumstruktur des Heaps wird dabei in dem Feld wie folgt abgebildet:

- Die Wurzel des Baumes ist $A[1]$.
- Sei i der Index eines Knotens außer der Wurzel in A, $2 \leq i \leq n$. Dann ist der Index des Vorgängers (*Vaters*) $\lfloor i/2 \rfloor$.
- Sei i der Index eines inneren Knotens in A, $1 \leq i \leq m$, und $m < n$ sei die Anzahl der inneren Knoten. Dann ist $2i$ der Index des linken Nachfolgers (*Kindes*) und $2i + 1$ der Index des rechten Nachfolgers (*Kindes*).

Bei einer binären Zahlendarstellung können Division durch 2 und Multiplikation mit 2 rechnerintern sehr effizient durch Schieben um eine Binärstelle nach rechts bzw. nach links realiesiert werden. Die Speicherung des Heaps aus Abbildung 5.9 in einem Feld ist in Abbildung 5.10 dargestellt.

1	2	3	4	5	6	7	8	9	10
16	14	13	8	7	11	12	2	4	1

Abbildung 5.10: Heap als Feld A mit $heapsize[A] = length[A]$

Im Folgenden werden die Konstruktion eines Heaps und die Aufrechterhaltung der Heap-Eigenschaft beim Einfügen und Löschen von Knoten und zwei Anwendungen dieser Datenstruktur, das Sortierverfahren *Heap-Sort* und die Implementierung des ADTs *Prioritäts-Warteschlange* beschrieben. Die dargestellten Algorithmen verwenden dabei ein Feld, dass einen Heap repräsentiert. Die Zugriffe auf die einzelnen Knoten können dann einfach über die Feldindizes erfolgen.

Sei A das einen Heap implementierende Feld. Es stehen dann die folgenden Feldattribute und Prozeduren zur Verfügung:

$length[A]$: Anzahl der Elemente des Feldes A.

$heapsize[A]$: Anzahl der Elemente im Heap. Es gilt immer $heapsize[A] \leq length[A]$. Für kein Element i mit $heapsize[A] < i \leq length[A]$ ist $A[i]$ gültig.

LEFT(i): Liefert den Index des linken Nachfolgers des Knotens i, also $2i$.

RIGHT(i): Liefert den Index des rechten Nachfolgers des Knotens i, also $2i + 1$.

PARENT(i): Liefert den Index des Vorgängers des Knotens i, also $\lfloor i/2 \rfloor$.

Betrachte vorab der Vollständigkeit halber zwei Zugriffsfunktionen des ADTs *binärer Baum* bzgl. dieser Implementierung.

Algorithmus 5.13 (Heap-Root)
Eingabe: Binärer Heap A
Ausgabe: Wurzel von A

```
ROOT(A)
1  return A[1]
```

Der Zugriff auf die Wurzel des Heaps hat eine konstante Zeitkomplexität. Aber die Suche nach einem bestimmten Knoten im Heap weist in Abhängigkeit von der Knotenanzahl eine lineare Komplexität auf. In einem Heap kann im Gegensatz zu einem binären Suchbaum nicht entschieden

Algorithmus 5.14 (Heap-Search)
Eingabe: Binärer Heap A, Schlüssel k
Ausgabe: Index i mit $A[i] = k$, falls dieser existiert, sonst $i = 0$

```
SEARCH(A, k)
1  i ← 1
2  while (i ≤ heapsize[A]) ∧ (A[i] ≠ k)
3     do i ← i + 1
4  if A[i] = k
5     then return i
6     else return 0
```

werden, ob ein Knoten mit einem bestimmten Schlüssel im linken oder im rechten Unterbaum eines Knotens liegt, falls er denn überhaupt enthalten ist. Also muss im schlechtesten Fall der gesamte Heap durchsucht werden.

Sicherlich kann ein Heap etwas effizienter traversiert werden als im Algorithmus 5.14 dargestellt, d. h. in Reihenfolge einer Baum-Präfix-Traversierung und einem ggf. vorzeitigen Abbruch der Suche in einem Unterbaum aufgrund der Schlüsselrelation zwischen Knoten und Unterbaum, aber im schlechtesten Fall bleibt der gesamte Heap zu durchsuchen.

Ein Heap ist also für das Suchen nach bestimmten Knoten trotz der garantierten Balance nicht die effizienteste Datenstruktur. Steht eine solche Operation im Vordergrund, dann sind z. B. so genannte *Rot-Schwarz-Bäume* besser geeignet. Dies sind binäre Suchbäume mit einer näherungsweisen Balance und somit einer logarithmischen Zeit-Komplexität für alle Lexikon-Operationen. Solche Bäume werden hier aber nicht betrachtet.

5.4.1 Konstruktion und Erhalten eines Heaps

Für die Konstruktion eines Heaps in einem Feld und die Aufrechterhaltung der Heap-Eigenschaft werde von einem Max-Heap ausgegangen. Zunächst sei die Erhaltung der Heap-Eigenschaft betrachtet. Für das Element $A[i]$, $1 \leq i \leq heapsize[A]$, gelte die Heap-Eigenschaft nicht, dass alle Elemente in den von diesem Element ausgehenden Unterbäumen $\textsc{Left}(i)$ und $\textsc{Right}(i)$ kein größeres Element aufweisen.

In Abbildung 5.11 ist ein Beispiel dafür gegeben. Der Übersichtlichkeit wegen sind die Abbildungen als Bäume dargestellt. Die Algorithmen sind aber für Felder implementiert. Die den Knoten zugeordneten Feldindizes sind deshalb in den Abbildungen mit aufgeführt.

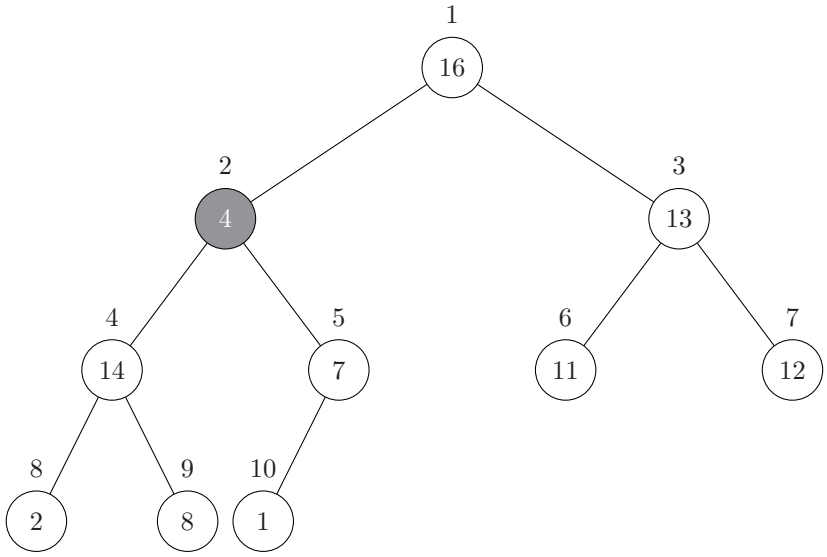


Abbildung 5.11: Binärer Max-Heap, Heap-Eigenschaft verletzt für markierten Knoten

Der Algorithmus 5.15 läßt den die Heap-Eigenschaft verletzenden Knoten so weit nach unten sinken bis die Max-Heap-Eigenschaft wieder erfüllt ist.

Algorithmus 5.15 (Heapify)
Eingabe: $A = (a_1, \ldots, a_n)$ mit $a_j \in \mathbb{N}, 1 \leq j \leq n = heapsize[A]$,
$i \in \mathbb{N}$ mit $1 \leq i \leq heapsize[A]$ und Max-Heap-Eigenschaft
von A ist nur für Element $A[i]$ nicht erfüllt
Ausgabe: Max-Heap A

```
HEAPIFY(A, i)
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  if (l ≤ heapsize[A]) ∧ (A[l] > A[i])
 4     then max ← l
 5     else max ← i
 6  if (r ≤ heapsize[A]) ∧ (A[r] > A[max])
 7     then max ← r
 8  if max ≠ i
 9     then VERTAUSCHE(A[i], A[max])
10          HEAPIFY(A, max)
```

In Abbildung 5.12 ist der Ablauf des Algorithmus 5.15 für das obige Beispiel dargestellt.

Die Zeitkomplexität $T(n)$ des Algorithmus 5.15, angewendet auf einen Heap mit n Elementen, ergibt sich insbesondere aus der Anzahl der rekursiven Aufrufe von *Heapify* (Zeile 10). Der nichtrekursive Aufwand ist konstant. Zur Aufstellung der Rekursionsgleichung für $T(n)$ ist die Größe der rekursiv zu untersuchenden Unterbäume zu bestimmen. Im schlechtesten Fall bleibt ein Unterbaum

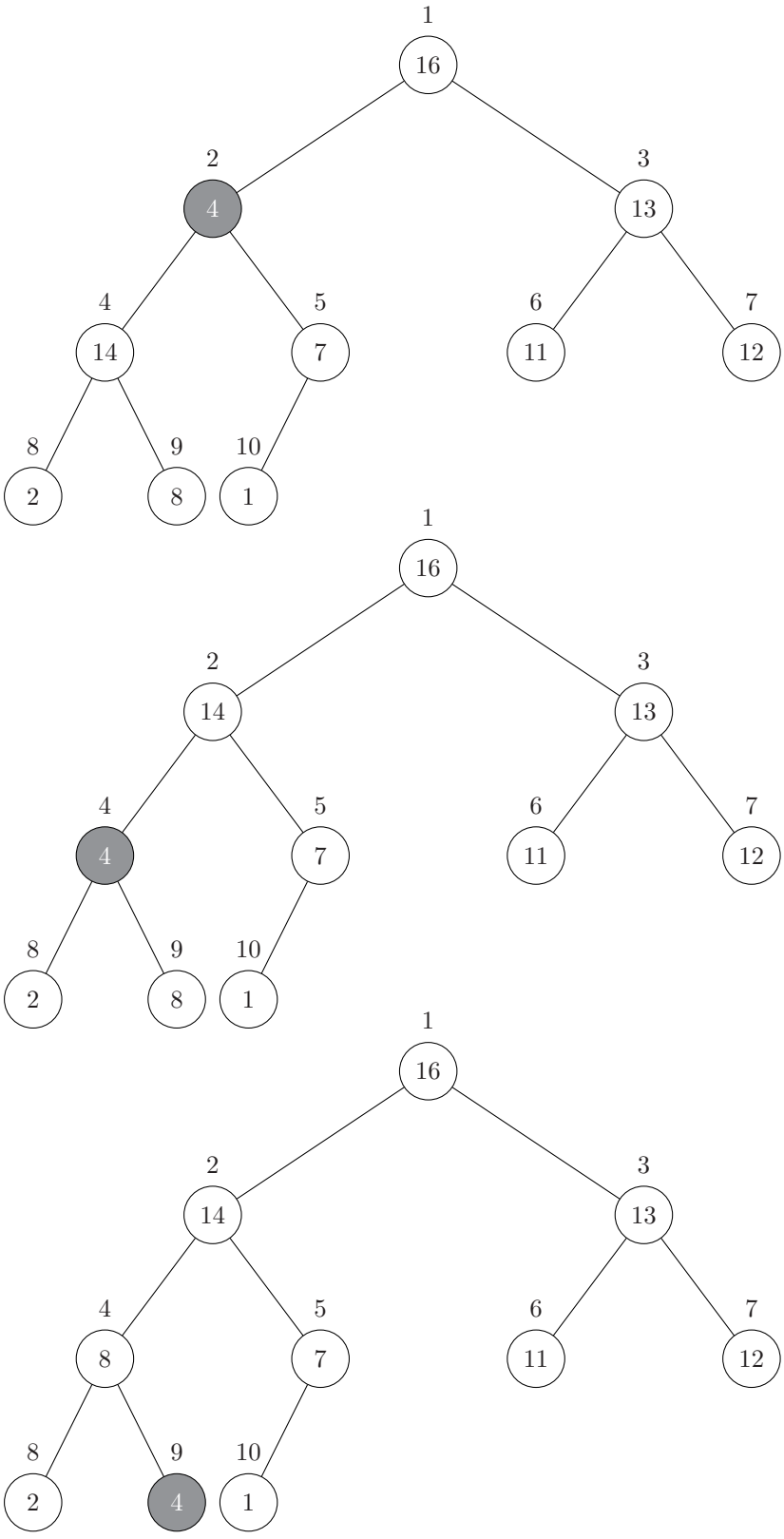


Abbildung 5.12: Aufrechterhaltung der Heap-Eigenschaft

der Größe $2n/3$ weiter relevant (Abbildung 5.13). Das führt zu der Rekursionsgleichung

$$T(n) = T(2n/3) + \Theta(1).$$

Mittels des Master-Theorems (Satz 3.2) ergibt sich mit $k = 0$ und $(2/3)^0 = 1$ die Komplexität im schlechtesten Fall zu

$$T(n) \in \Theta(\log n).$$

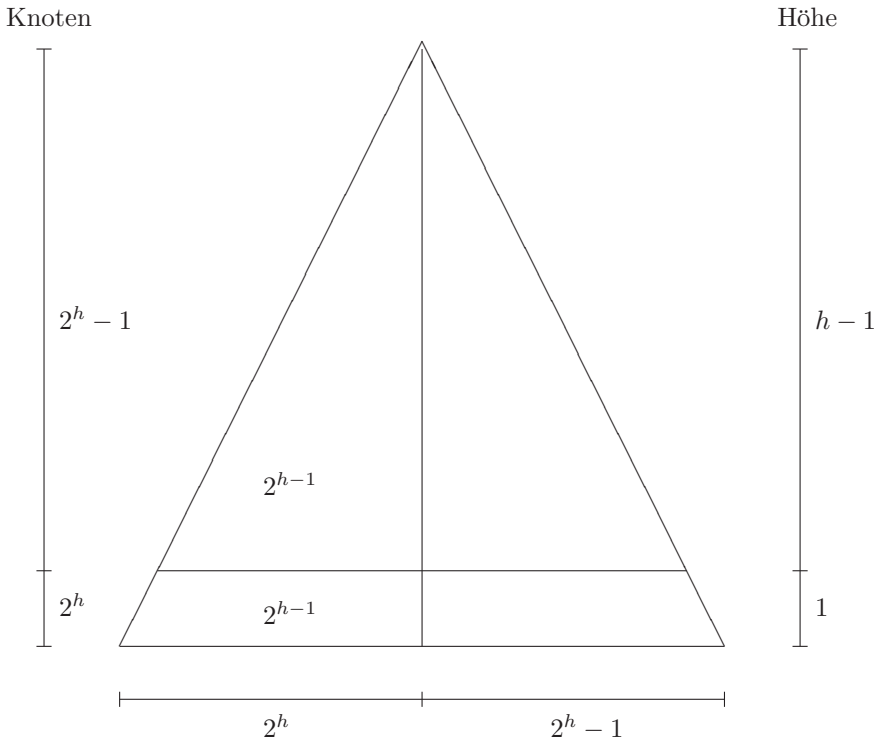


Abbildung 5.13: Knotenanzahlen in einem Heap in Abhängigkeit von der Höhe h

Mit Hilfe des Algorithmus 5.15 kann nun ein beliebiges Feld A der Länge n in einen Max-Heap überführt werden. Alle Elemente des Feldes, die bei Interpretation als Heap Blätter sind, sind initial bereits einelementige Heaps. Die Elemente, die als innere Knoten zu interpretieren sind, sind dann *bottom-up* mit ihren zugehörigen Unterbäumen in Heaps zu überführen (Algorithmus 5.16). In diesen Algorithmus geht ein, dass die Feldindizes 1 bis $\lfloor n/2 \rfloor$ eines einen Heap implementierenden Feldes $A[1..n]$ auf innere Knoten und die Feldindizes $\lfloor n/2 \rfloor + 1$ bis n auf Blätter verweisen (Satz 5.4).

Algorithmus 5.16 (Build-Heap)
Eingabe: $A = (a_1, \ldots, a_n)$ mit $a_i \in \mathbb{N}, 1 \leq i \leq n = heapsize[A]$
Ausgabe: Max-Heap A

```
BUILD-HEAP(A)
1  for i ← ⌊n/2⌋ downto 1
2      do HEAPIFY(A, i)
```

Satz 5.4
Sei A ein einen Max-Heap implementierendes Feld der Größe n. Die Indizes 1 bis $\lfloor n/2 \rfloor$ verweisen dann auf innere Knoten und die Indizes $\lfloor n/2 \rfloor + 1$ bis n auf Blätter des Max-Heaps.

Beweis:
Der Beweis erfolgt durch vollständige Induktion über die Knotenanzahl n.

$n = 1$: Der Baum besteht ausschließlich aus der Wurzel, diese ist gleichzeitig Blatt. Das zugeordnete Feld besteht also nur aus einem Element und es gelten:

- größter Index eines inneren Knotens: $\lfloor n/2 \rfloor = \lfloor 1/2 \rfloor = 0$. Dieser existiert nicht, d. h. es gibt keinen inneren Knoten.

- kleinster Index eines Blattes: $\lfloor n/2 \rfloor + 1 = \lfloor 1/2 \rfloor + 1 = 1$. Dies ist auch der einzige Index eines Blattes.

Schluss von n auf $n + 1$:

I. Sei n gerade:

   Die Knoten 1 bis $\lfloor n/2 \rfloor$ sind dann innere Knoten und der Knoten $\lfloor n/2 \rfloor$ hat als einziger nur einen Nachfolger, und zwar einen linken Nachfolger. Die Knoten $\lfloor n/2 \rfloor + 1$ bis n sind Blätter.

   Es kommt nun ein Knoten hinzu. Dieser erhält den Index $n + 1$, wird rechter Nachfolger des Knotens $\lfloor n/2 \rfloor$ und ist Blatt. Alle bisherigen inneren Knoten bleiben innere Knoten, alle bisherigen Blätter bleiben Blätter.

   - größter Index eines inneren Knotens: $\lfloor n/2 \rfloor = \lfloor n + 1/2 \rfloor$
   - kleinster Index eines Blattes: $\lfloor n/2 \rfloor + 1 = \lfloor n + 1/2 \rfloor + 1$

II. Sei n ungerade:

   Die Knoten 1 bis $\lfloor n/2 \rfloor$ sind dann innere Knoten und alle haben zwei Nachfolger. Die Knoten $\lfloor n/2 \rfloor + 1$ bis n sind Blätter.

   Es kommt nun ein Knoten hinzu. Dieser erhält den Index $n + 1$, wird linker Nachfolger des Knotens $\lfloor n/2 \rfloor + 1$ und ist Blatt. Der Knoten $\lfloor n/2 \rfloor + 1$ wird damit vom Blatt zum inneren Knoten. Alle anderen bisherigen Blätter bleiben Blätter, alle bisherigen inneren Knoten bleiben innere Knoten.

   - größter Index eines inneren Knotens: $\lfloor n/2 \rfloor + 1 = \lfloor n + 2/2 \rfloor = \lfloor n + 1/2 \rfloor$
   - kleinster Index eines Blattes: $\lfloor n/2 \rfloor + 2 = \lfloor n + 2/2 \rfloor + 1 = \lfloor n + 1/2 \rfloor + 1$

□

Beispiel 5.9
In den Abbildungen 5.14 und 5.15 werde der Ablauf des Algorithmus *Build-Heap* für das Feld

$$A = (4, 1, 12, 2, 16, 11, 13, 14, 8, 7)$$

skizziert. Knoten 5 ist der erste, mit dem der Algorithmus *Heapify* aufgerufen wird. Er ist deshalb grau hinterlegt. Da der Unterbaum dieses Knotens aber bereits die Max-Heap-Eigenschaft erfüllt, bleibt der Baum unverändert, und es wird der Knoten 4 als nächstes betrachtet. Die Knoten 4 und 8 werden vertauscht und der nächste Aufruf erfolgt mit dem Knoten 3. Nach Ausführung des letzten Aufrufs von *Heapify* mit dem Knoten 1 ist der Baum ein Max-Heap, denn für alle Knoten gilt die Max-Heap-Eigenschaft. □

Eine einfache asymptotische Abschätzung für die Zeit-Komplexität des Algorithmus 5.16 ergibt sich sehr einfach aus $\Theta(\log n)$ für den Algorithmus 5.15 und seinem $(n/2)$-fachen Aufruf, also $\Theta(n \log n)$. Da aber nicht alle Knoten auf der gleichen Ebene in dem Baum liegen, und die für *Heapify* relevanten Unterbäume vom Knoten $n/2$ bis zum Knoten 1 eine Höhe von 0 bis $\lceil \mathrm{ld}\, n \rceil$ durchlaufen, kann eine deutlich schärfere Schranke angegeben werden. Es kann gezeigt werden, dass diese bei $\Theta(n)$ liegt. Ein Heap kann also in linearer Zeit in Abhängigkeit von der Anzahl der Elemente aus einem ungeordneten Feld konstruiert werden.

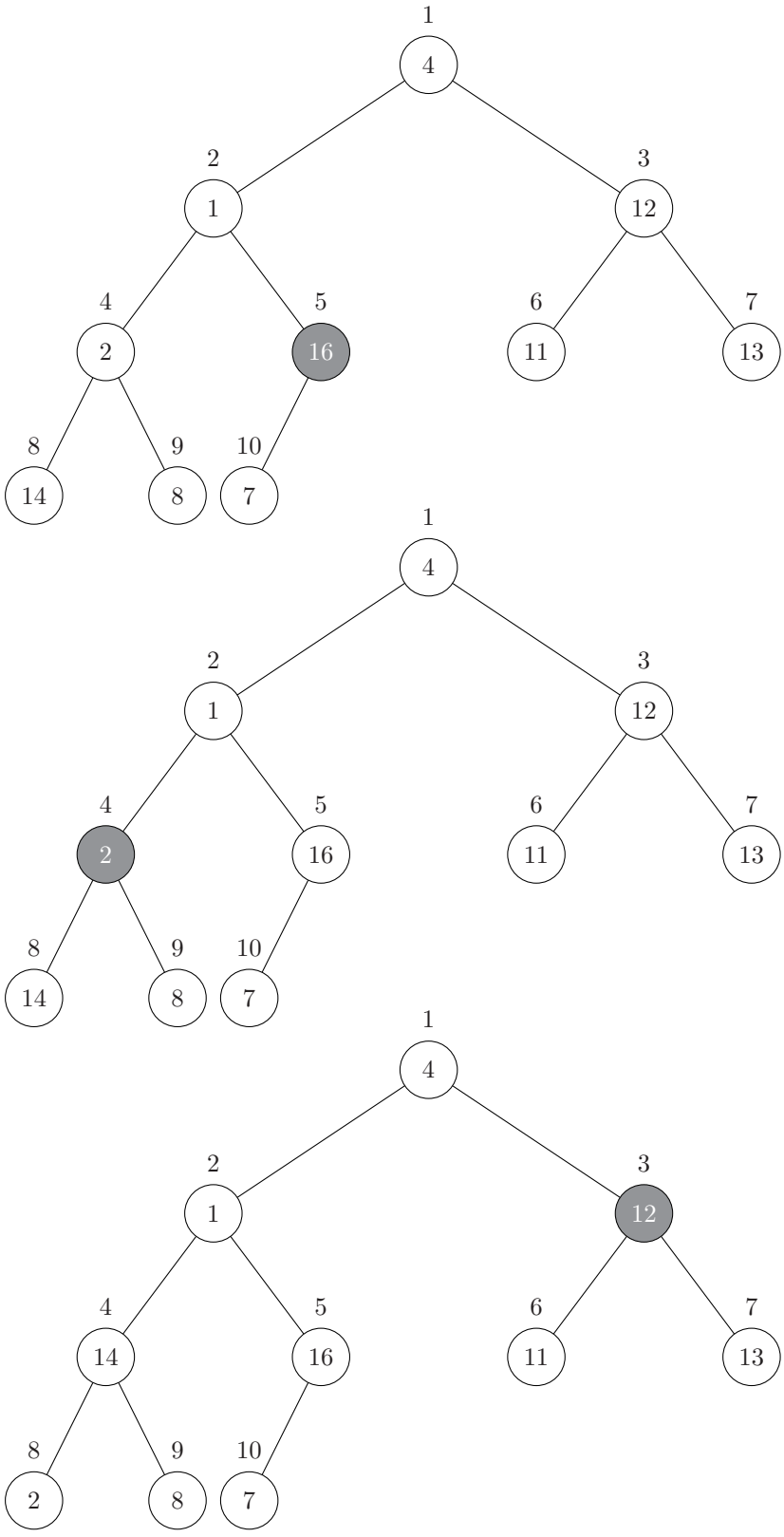


Abbildung 5.14: Konstruktion eines Heaps

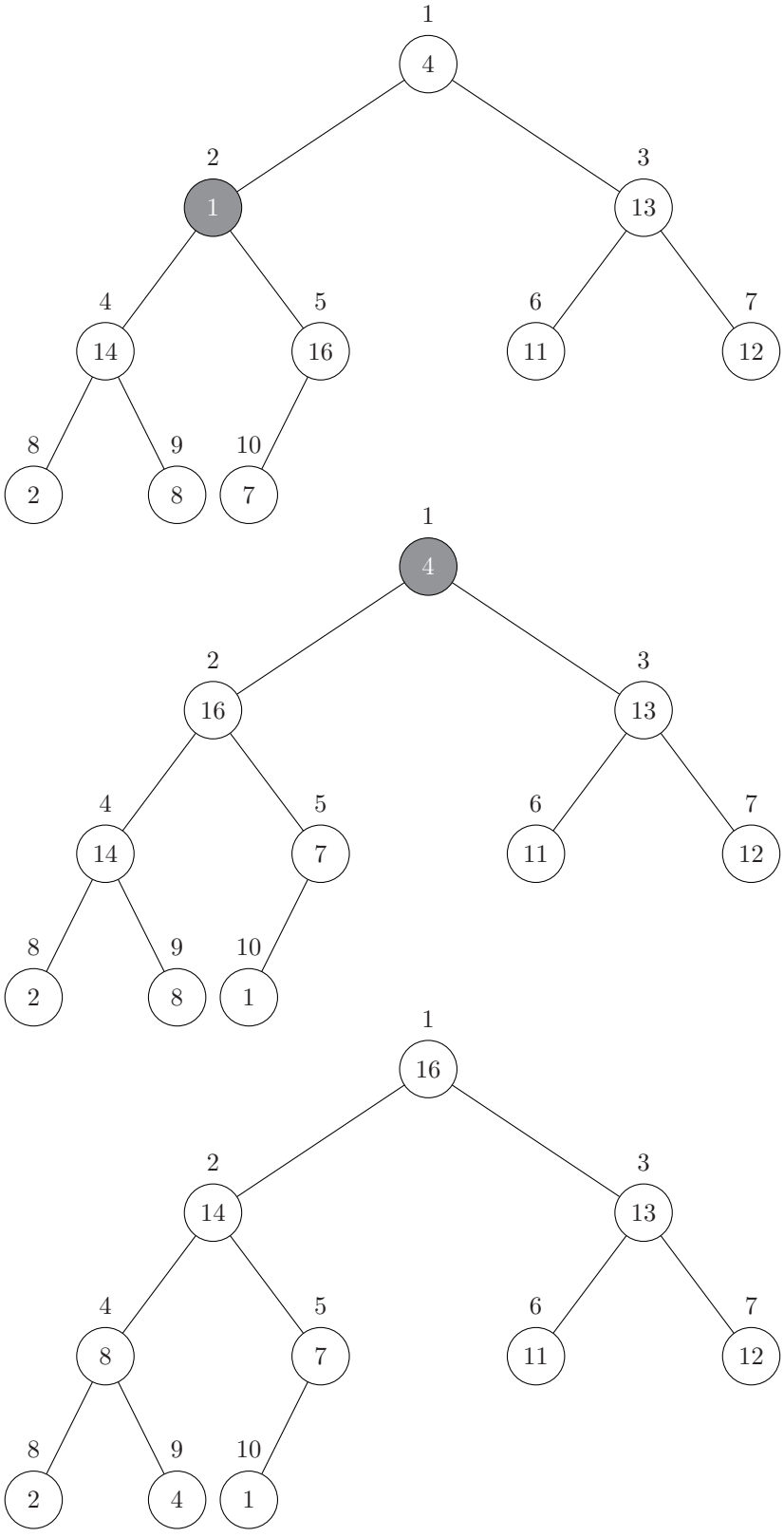


Abbildung 5.15: Konstruktion eines Heaps (Fortsetzung)

5.4.2 Heap-Sort

Mittels der Datenstruktur eines Max-Heaps, der auf einem Feld der Länge n realisiert ist, kann nun auch leicht das Feld aufsteigend sortiert werden. Die Wurzel des Heaps ist das größte Element, in dem Feld ist es das erste Element, und es wird mit dem letzten Element des Feldes vertauscht. Nun wird die Heap-Größe um eins dekrementiert, also das letzte Element des Feldes gehört nicht mehr zum Heap. Für diesen Heap muss wieder die Heap-Eigenschaft hergestellt werden, diese ist aber ggf. nur bzgl. der neuen Wurzel wegen des vorangegangenen Tauschs nicht erfüllt, also wird sie wieder hergestellt. Damit ist das insgesamt zweitgrößte Element das größte des dekrementierten Heaps, und es wird mit dem vorletzten Feldelement vertauscht. Dieser Prozess wird nun so lange wiederholt bis das Feld sortiert ist. (Algorithmus 5.17)

Algorithmus 5.17 (Heap-Sort)
Eingabe: $A = (a_1, \ldots, a_n)$ mit $a_i \in \mathbb{N}, 1 \leq i \leq n = heapsize[A]$
Ausgabe: $A = (a_{\pi(1)}, \ldots, a_{\pi(n)})$ mit $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$

```
HEAP-SORT(A)
1  BUILD-HEAP(A)
2  for i ← n downto 2
3      do VERTAUSCHE(A[1], A[i])
4         heapsize[A] ← heapsize[A] − 1
5         HEAPIFY(A, 1)
```

In den Abbildungen 5.16 bis 5.19 ist ein Beispiel für den Ablauf des Heap-Sorts dargestellt. Es zeigt den Heap jeweils zu Beginn eines Schleifendurchlaufs, also mit wieder hergestellter Heap-Eigenschaft. Die bereits sortierten und aus dem Heap ausgegliederten Elemente sind als isolierte Knoten dargestellt. Die Zuordnung der Knoten zu den jeweiligen Positionen im Feld ergibt dann die aufsteigende Sortierung der Feldelemente.

Die Zeitkomplexität des Sortierens eines Feldes der Länge n mittels Heap-Sort beträgt im schlechtesten Fall $\Theta(n \log n)$, denn die Konstruktion eines Heaps (Zeile 1) benötigt eine Zeit von $\Theta(n)$ und jeder der $n - 1$ Schleifen-Durchläufe zur Wiederherstellung der Heap-Eigenschaft $\Theta(\log n)$.

Dieses Verfahren sortiert am Platz. Aber es ist nicht stabil. Betrachte dazu eine Modifikation des Feldes aus Abbildung 5.16. Werde hier das Element $A[3]$ in den Wert 14 geändert. Das Element $A[2]$ wird dann durch die Sortierung zu Element $A[9]$ und $A[3]$ zu $A[8]$. Die Ordnung von gleichen Elementen bleibt also nicht erhalten.

Eine absteigende Sortierung eines Feldes kann analog zum hier verwendeten *Max-Heap* über einen *Min-Heap* implementiert werden.

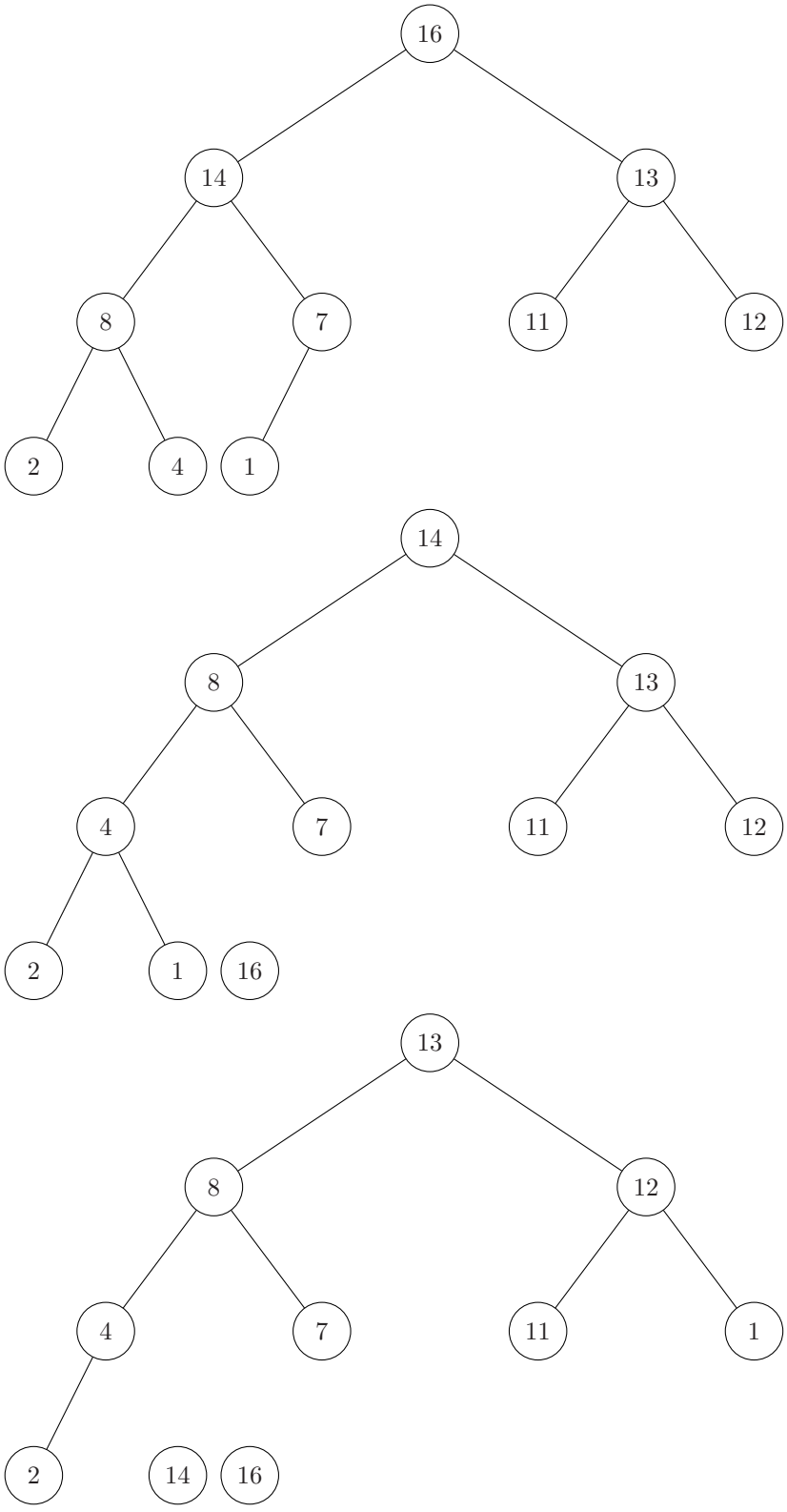


Abbildung 5.16: Heap-Sort

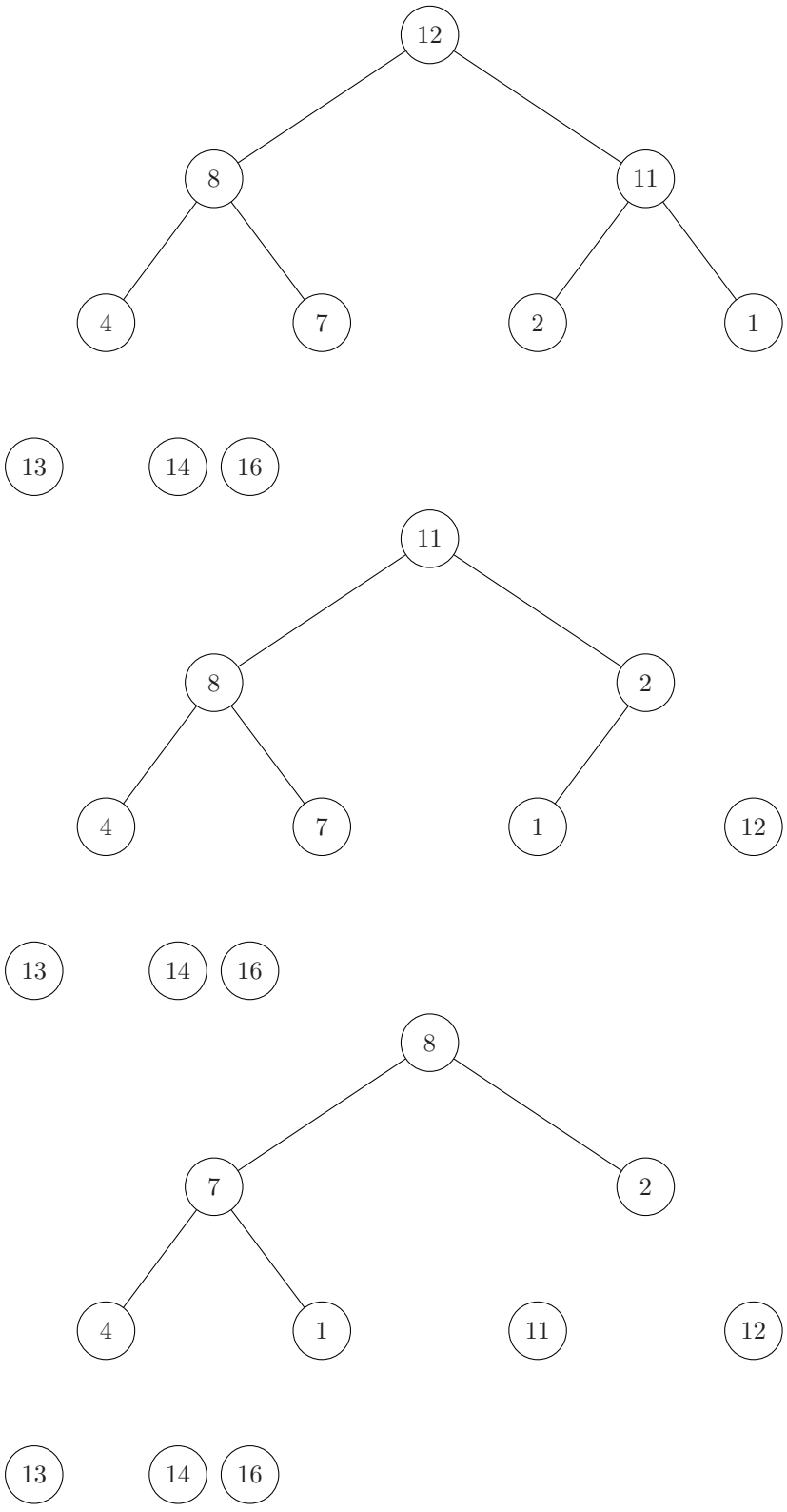


Abbildung 5.17: Heap-Sort (Fortsetzung 1)

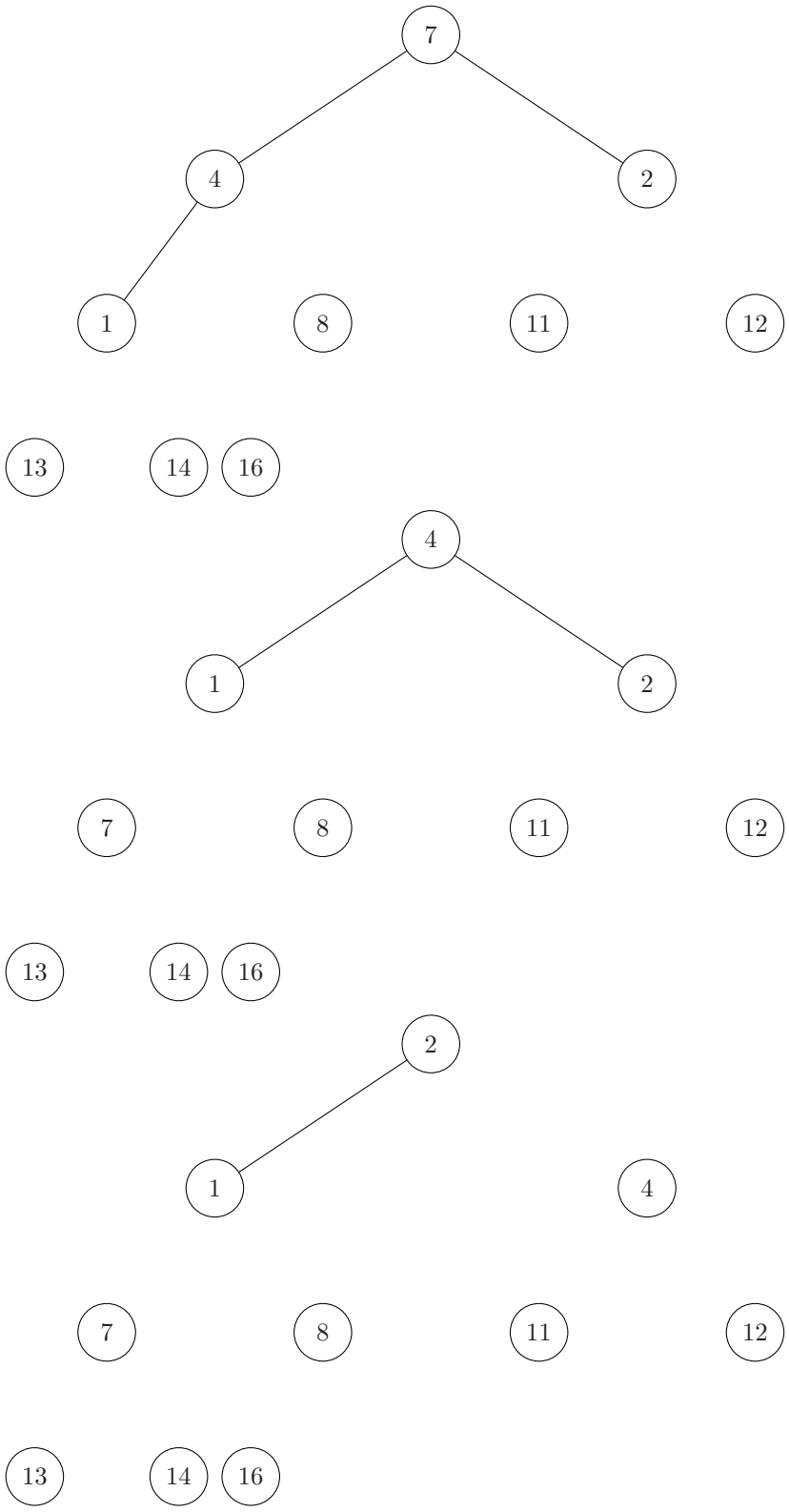
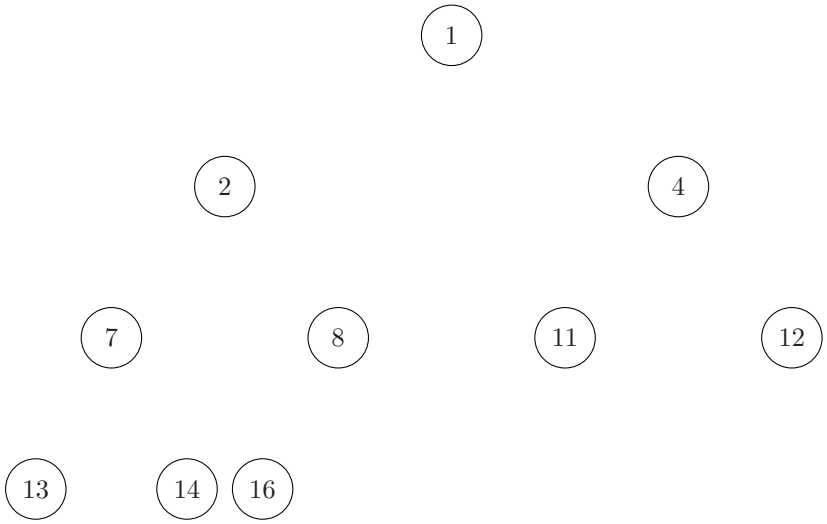


Abbildung 5.18: Heap-Sort (Fortsetzung 2)



1	2	4	7	8	11	12	13	14	16

Abbildung 5.19: Heap-Sort (Fortsetzung 3)

5.4.3 Prioritäts-Warteschlange

Die Datenstruktur eines Heaps kann auch sehr gut zur Implementierung des ADTs *Prioritäts-Warteschlange* (*priority queue*) verwendet werden (Definition 5.6, Beispiel 5.7). Dies ist eine Warteschlange, in der jeweils das Element mit der höchsten bzw. niedrigsten Priorität als nächstes abzuarbeiten ist. Man spricht von *Max-* und *Min-Priorität-Warteschlangen*. Den Prioritäten entsprechen die Schlüssel eines Max- bzw. Min-Heaps. Im Folgenden seien Max-Prioritäts-Warteschlangen betrachtet, die über Max-Heaps implementiert sind.

Für eine solche Datenstruktur werden die folgenden Zugriffsfunktionen bereit gestellt. Sei A das diese Datenstruktur implementierende Feld.

GetMax(A) liefert das Element von A mit dem größten Schlüssel.

Delete(A) liefert das Element von A mit dem größten Schlüssel und entfernt es aus A.

IncreaseKey(A, i, k) erhöht in A den Wert des Elements $A[i]$ auf den Wert k.

Insert(A, k) fügt ein Element mit dem Schlüssel k in A ein.

Die Algorithmen 5.18 bis 5.21 zeigen die Implementierungen dieser Operationen.

Algorithmus 5.18 (Heap-Maximum)
Eingabe: Max-Heap A, $heapsize[A] \geq 1$
Ausgabe: $\max(A)$

```
GetMax(A)
1  return A[1]
```

Algorithmus 5.19 (Heap-Delete)
Eingabe: Max-Heap A, $heapsize[A] \geq 1$
Ausgabe: $max = \max(A)$

```
Delete(A)
1  max ← A[1]
2  A[1] ← A[heapsize[A]]
3  heapsize[A] ← heapsize[A] − 1
4  Heapify(A, 1)
5  return max
```

Algorithmus 5.20 (Heap-IncreaseKey)
Eingabe: Max-Heap A, $i, k \in \mathbb{N}$, $A[i] \leq k$
Ausgabe: keine

```
IncreaseKey(A, i, k)
1  A[i] ← k
2  while (i > 1) ∧ (A[Parent(i)] < A[i])
3      do Vertausche(A[i], A[Parent(i)])
4          i ←Parent(i)
```

Der Algorithmus *Heap-Maximum* liefert die Wurzel des Heaps. Diese hält das maximale Element. Die Zeitkomplexität ist unabhängig von der Größe des Heaps, also konstant.

Der Algorithmus *Heap-Delete* liefert auch die Wurzel des Heaps. Diese wird dann durch das letzte Heap-Element ersetzt und die Heap-Größe um eins verkleinert. Die neue Wurzel verletzt ggf. die Heap-Eigenschaft, was durch Aufruf von *Heapify* korrigiert wird. Die Zeitkomplexität im schlechtesten Fall entspricht dem Algorithmus *Heapify*, also $\Theta(\log n)$.

Algorithmus 5.21 (Heap-Insert)
Eingabe: Max-Heap A, $k \in \mathbb{N}$, $heapsize[A] < length[A]$
Ausgabe: keine

INSERT(A, k)
1 $heapsize[A] \leftarrow heapsize[A] + 1$
2 $A[heapsize[A]] \leftarrow -\infty$
3 INCREASEKEY$(A, heapsize[A], k)$

Der Algorithmus *Heap-IncreaseKey* erhöht als erstes den Schlüssel des betreffenden Elementes auf den gewünschten Wert. Die dadurch ggf. verletzte Heap-Eigenschaft wird aufsteigend durch Tausch bis maximal zur Wurzel wieder hergestellt. Ein Beispiel für den Ablauf dieses Algorithmus ist in Abbildung 5.20 dargestellt. Die Zeitkomplexität entspricht im schlechtesten Fall der Höhe des Baumes, also $\Theta(\log n)$.

Der Algorithmus *Heap-Insert* erhöht die Heap-Größe um eins und fügt ein neues kleinstes Element am Ende des Heaps ein. Der Schlüssel dieses Elements wird durch Aufruf von *Heap-IncreaseKey* auf den gewünschten Wert gesetzt. Die Zeitkomplexität entspricht damit dem Algorithmus *Heap-IncreaseKey*, also $\Theta(\log n)$. Ein neues Element kann natürlich nur eingefügt werden, wenn $heapsize[A] < length[A]$. Ist diese Bedingung nicht erfüllt, dann ist eine Fehlermeldung zu generieren oder das Feld muss vergrößert werden. Eine Technik dazu ist das so genannte *Array-Doubling*, das in Abschnitt 6.5 besprochen wird.

Alle hier betrachteten Operationen lassen sich also im schlechtesten Fall in logarithmischer Zeit in Abhängigkeit von der Elementanzahl implementieren. Bei einer Implementierung über verketteten Listen wäre die Laufzeit im schlechtesten Fall dagegen linear.

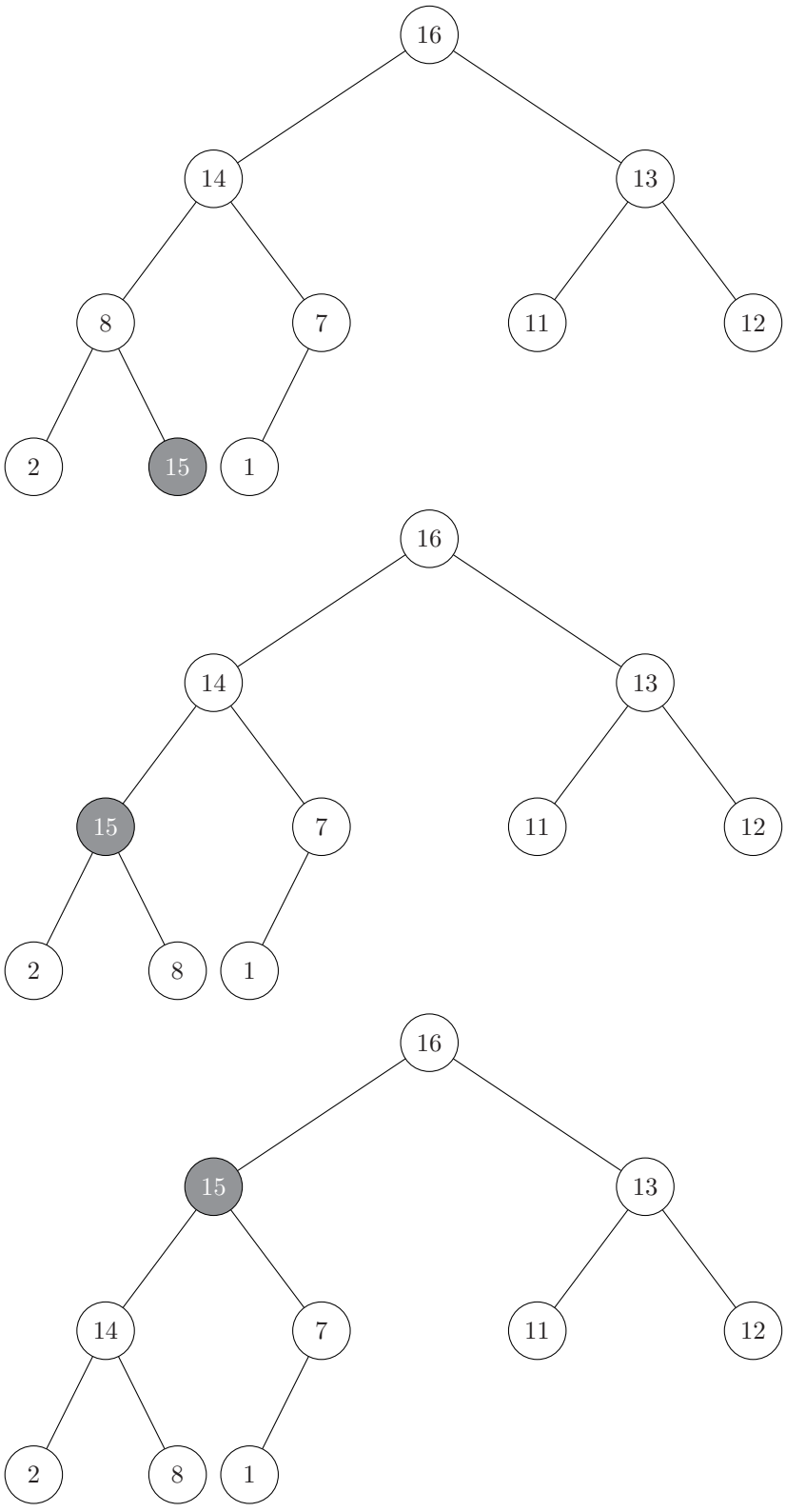


Abbildung 5.20: Arbeitsweise von IncreaseKey$(A, 9, 15)$

Kapitel 6

Hashverfahren

6.1 Adresstabelle mit direktem Zugriff

In vielen Anwendungen mit sich dynamisch ändernden Daten werden nur die elementaren Operationen zum Einfügen, Suchen, Ändern und Löschen von Daten benötigt. Haben diese Daten einen eindeutigen Schlüssel, dann ist der ADT *Lexikon* (*dictionary*) sehr gut geeignet, diese Daten zu speichern und zu verwalten (Definition 5.7, Beispiel 5.8).

Eine mögliche Implementierung des ADTs *Lexikon* ist eine *Adresstabelle mit direktem Zugriff*. Dazu müssen aber die Schlüssel der Datensätze Elemente der Menge $U = \{0, 1, 2, \ldots, m-1\}$ sein. Eine Tabelle wird als ein m-elementiges Feld (*array*) implementiert und der Schlüssel ist gleichzeitig der Index. Ein Feldelement kann den gesamten Datensatz halten, oder aber – was meist die angemessenere Variante darstellt – einen Zeiger auf den Datensatz. In Abbildung 6.1 ist eine solche Adresstabelle dargestellt.

Sei das Feld $T[0..m-1]$ eine solche Adresstabelle. (Von der bisherigen Indizierung von Feldern beginnend mit 1 wird hier abgewichen, da ein Beginn mit dem Index 0 für die nachfolgenden Verfahren angemessener ist.) Jedes Feldelement $T[k]$, $0 \leq k \leq m-1$, korrespondiert mit einem Schlüssel k aus der Menge U. Existiert ein Datensatz mit diesem Schlüssel, dann enthält $T[k]$ einen Zeiger auf diesen Datensatz, existiert kein Datensatz, dann gilt $T[k] = \text{NIL}$.

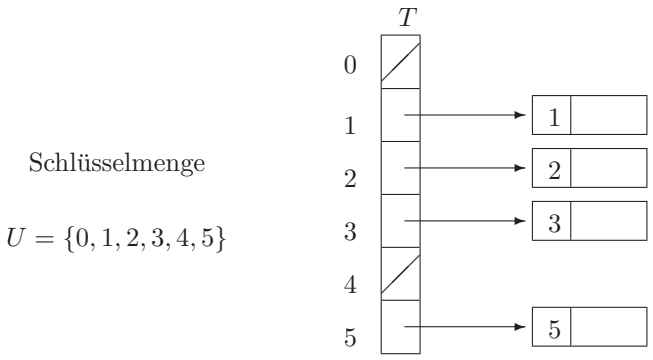


Abbildung 6.1: Adresstabelle mit direktem Zugriff

Eine solche *direkte Adressierung* ist natürlich nur sinnvoll, wenn m nicht zu groß ist oder die Anzahl der existierenden Datensätze nicht zu sehr von der Anzahl der möglichen Schlüssel und damit von der Feldgröße abweicht. Betrachte dazu das Beispiel 6.1.

Beispiel 6.1
Aufbau einer Matrikelnummer der FH BS/WF:

2 0 6 6 6 0 1 9

Prüfziffer
lfd. Nr.
Studiengang
Immatr. WS/SS
Jahr
Standort

Sind an allen Stellen alle zehn Ziffern möglich, dann gäbe es 10^8 verschiedene Schlüssel und es würde eine Tabelle von entsprechender Größe benötigt werden. An der FH BS/WF sind aber z. Z. nur ca. 6500 Studierende immatrikuliert, die Tabelle wäre also überwiegend leer. Eine Tabelle aus 10^4 Elementen wäre also mehr als ausreichend. □

Die Implementierung der Operationen zum Suchen, Einfügen und Löschen von Datensätzen ist sehr einfach. Das Ändern der Daten wird nicht betrachtet, da diesem nur das Suchen vorausgeht. Betrachte die Algorithmen 6.1, 6.2 und 6.3. Sie haben alle die Zeitkomplexität $\Theta(1)$, sind also unabhängig von der Größe der Adresstabelle und der wirklichen Datenmenge.

Algorithmus 6.1 (Direct-Address-Retrieve)
Eingabe: Adresstabelle T, Schlüssel des zu suchenden Datensatzes k
Ausgabe: Zeiger $T[k]$ auf den Datensatz, falls er existiert, sonst NIL

```
RETRIEVE(T, k)
1  return T[k]
```

Algorithmus 6.2 (Direct-Address-Insert)
Eingabe: Adresstabelle T, Zeiger auf den einzufügenden Datensatz x
Ausgabe: keine

```
INSERT(T, x)
1  T[key[x]] ← x
```

Algorithmus 6.3 (Direct-Address-Delete)
Eingabe: Adresstabelle T, Schlüssel des zu löschenden Datensatzes k
Ausgabe: keine

```
DELETE(T, k)
1  T[k] ← NIL
```

6.2 Hashtabellen

Die Verwendung von Adresstabellen mit direktem Zugriff ist offensichtlich nur dann sinnvoll, wenn die Anzahl der wirklich verwendeten Schlüssel dicht an der Anzahl der insgesamt zur Verfügung stehenden Schlüssel liegt. Ansonsten bleibt zu viel Speicherplatz ungenutzt.

Eine *Hashtabelle* ist auch eine Adresstabelle. Ihre Größe orientiert sich aber an der Anzahl der wirklich benötigten Schlüssel und nicht an der Anzahl der zur Verfügung stehenden Schlüssel. Ein Schlüssel eines Datensatzes liefert aber nicht direkt den Index der Adresstabelle, an der dann der Zeiger auf die Daten gespeichert steht, sondern der Index ist vorab aus dem Schlüssel mittels einer so genannten *Hashfunktion* zu berechnen. Diese Vorgehensweise wird *Hashing* genannt. Die Reihenfolge der Schlüssel wird zerhackt und durcheinander gemischt. Auch über Hashtabellen kann der ADT *Lexikon* (*dictionary*) sehr gut implementiert werden.

Eine Hashfunktion h bildet also aus der Schlüsselmenge U in die Menge der Indizes einer Hashtabelle $T[0..m-1]$ ab,

$$h : U \to \{0, 1, 2, \ldots, m-1\}.$$

Für $k \in U$ heißt $h(k)$ *Hashwert* des Schlüssels k. Während bei Adresstabellen mit direktem Zugriff $|U| \leq m$ gilt, kann bei Hashtabellen $|U| \geq m$ gelten, im Allgemeinen sogar $|U| \gg m$. Es kann dann Schlüssel k_1 und k_2 mit $h(k_1) = h(k_2)$ geben, so genannte *Kollisionen*. Kollisionen in einer Hashtabelle können z. B. aufgelöst werden, indem die Feldelemente nicht Zeiger auf die Datensätze enthalten, sondern Zeiger auf verkettete Listen, die wiederum Zeiger auf die jeweiligen Datensätze enthalten (Abbildung 6.2). Man spricht dann auch von Hashtabellen mit Verkettung bzw. von Hashing mit *geschlossener Adressierung*. Die Listen werden auch *Kollisionsketten* genannt.

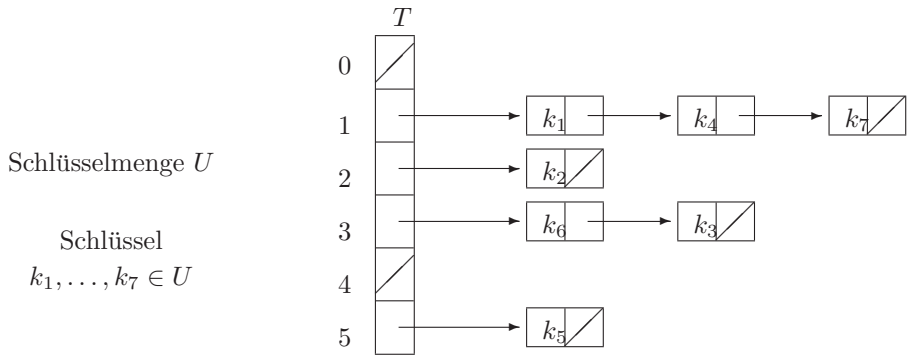


Abbildung 6.2: Hashtabelle mit z. B. $h(k_3) = h(k_6) = 3$

Am besten wäre es natürlich, Kollisionen zu vermeiden. Man spricht dann – oder genauer, wenn der Zugriff auf den gesuchten Datensatz im schlechtesten Fall einen Zeitbedarf von $\Theta(1)$ hat – von *perfektem Hashing*. Dieses Ziel kann aber in vielen Anwendungen nicht erreicht werden.

Werden die Kollisionen in Hashtabellen durch unsortierte doppelt verkettete Listen aufgelöst, dann lassen sich die Such-, Einfüge- und Löschoperationen unter Benutzung der entsprechenden Listenoperationen (Algorithmen 5.3, 5.4 und 5.5) wie folgt implementieren (Algorithmen 6.4, 6.5 und 6.6).

Algorithmus 6.4 (Hash-Retrieve)
Eingabe: Hashtabelle T, Schlüssel des zu suchenden Datensatzes k
Ausgabe: Zeiger x auf den Datensatz mit $key[x] = k$, falls er existiert,
sonst NIL

RETRIEVE(T, k)
1 **return** LIST-SEARCH$(T[h(k)], k)$

Auf konkrete Hashfunktionen wird erst im nächsten Abschnitt eingegangen. Hier sei dazu soviel vorangestellt, dass die Berechnung des Hashwertes unabhängig von der Schlüsselmenge und der

Algorithmus 6.5 (Hash-Insert)
Eingabe: Hashtabelle T, Zeiger x auf den einzufügenden Datensatz
Ausgabe: keine

```
INSERT(T, x)
1  LIST-INSERT(T[h(key[x])], x)
```

Algorithmus 6.6 (Hash-Delete)
Eingabe: Hashtabelle T, Schlüssel des zu löschenden Datensatzes k
Ausgabe: keine

```
DELETE(T, k)
1  x ← LIST-SEARCH(T[h(k)], k)
2  if x ≠ NIL
3      LIST-DELETE(T[h(k)], x)
```

Tabellengröße sein sollte. Sei also vorausgesetzt, dass dies erfüllt sei und diese Berechnung eine konstante Zeitkomplexität habe.

Das Suchen weist im schlechtesten Fall eine lineare Komplexität auf, und zwar in Abhängigkeit von der Anzahl n der in der Tabelle gespeicherten Datensätze, also $\Theta(n)$, denn im schlechtesten Fall werden alle Schlüssel auf genau ein Element der Hashtabelle abgebildet und dann hat die dort beginnende verkettete Liste n Elemente. Eine derartige Hashfunktion sollte allerdings nicht zum Einsatz kommen. Im besten Fall wird für das Suchen eines Datensatzes nur eine konstante Zeit benötigt. Der Datensatz wird dann gleich als Kopf einer Liste gefunden.

Wieviel Zeit aber wird im Mittel für das Suchen eines Datensatzes in einer Hashtabelle mit Verkettung als Kollisionsauflösung benötigt? Zur Bestimmung der Komplexität im mittleren Fall ist aber die mittlere Länge der verketteten Listen zu bestimmen, und dazu sind einige Annahmen zu treffen und Begriffsbestimmungen einzuführen.

Sei T eine Hashtabelle mit m Plätzen, in der insgesamt n Elemente (Datensätze) gespeichert sind. Der Quotient $n/m = \alpha$ wird dann als *Belegungsfaktor* bezeichnet.

I. $\alpha < 1$, falls $m > n$, d. h. es gibt mehr Plätze als Elemente.

II. $\alpha = 1$, falls $m = n$, d. h. es gibt genau so viele Plätze wie Elemente.

III. $\alpha > 1$, falls $m < n$, d. h. es gibt weniger Plätze als Elemente.

Im Fall III. sind Kollisionen unvermeidlich. Aber auch in den Fällen I. und II. können Kollisionen vorkommen. Dies hängt von der gewählten Hashfunktion ab.

Hier sei angenommen, dass eine Hashfunktion h vorliege, die jeden Schlüssel der n Elemente mit gleicher Wahrscheinlichkeit und unabhängig von den anderen Elementen auf jeden der zur Verfügung stehenden m Plätze abbilde. Diese Annahme wird als *einfaches gleichmäßiges Hashing* bezeichnet. Unter dieser Voraussetzung gibt der Belegungsfaktor α die mittlere Anzahl an Elementen an, die je Liste gespeichert sind.

Werde also ein Element mit dem Schlüssel k gesucht: $h(k)$ kann wieder in der Zeit $\Theta(1)$ berechnet werden und auch der Zugriff auf $T[h(k)]$ erfolgt in konstanter Zeit. Die Komplexität insgesamt hängt dann weiterhin von der Länge der Liste $T[h(k)]$ ab. Im Mittel haben aber die Listen eine Länge von $\alpha = n/m$. Damit beträgt die Komplexität im mittleren Fall $\Theta(1 + n/m)$.

Kann nun weiterhin angenommen werden, dass die Anzahl der Plätze der Hashtabelle mindestens proportional zur Anzahl der Elemente in der Tabelle ist, also $n \leq c \cdot m$, $c \in \mathbb{R}^+$ konstant, dann folgt für den Belegungsfaktor

$$\alpha = n/m \leq c \cdot m/m = c.$$

Damit ergibt sich für das Suchen von Elementen in einer Hashtabelle unter dieser Voraussetzung auch im mittleren Fall eine Komplexität von $\Theta(1 + c) = \Theta(1)$.

Eine asymptotisch konstante Komplexität heißt aber nun nicht, dass keine Kollisionen auftreten. Im Folgenden sei deshalb ohne Herleitung ein statistisches Ergebnis aufgeführt, das die Anzahl der möglichen Kollisionen bei gegebenem n und m verdeutlichen soll. Ausgegangen wird weiterhin von einfachem gleichmäßigem Hashing. Für $n \approx \sqrt{2m}$ wird dann die Anzahl der erwarteten Kollisionen ≥ 1. Bei einer Tabellengröße von z. B. $m = 365$ wird ab $n = 27$ einzutragenden Elementen mindestens eine Kollision erwartet. Übertragen auf das so genannte *Geburtstagsparadoxon* bedeutet dies, ab 27 Personen wird erwartet, dass mindestens zwei davon am gleichen Tag Geburtstag haben.

Bleibt die Komplexität für das Einfügen und Löschen von Datensätzen nachzutragen. Sie ergibt sich aus den entsprechenden Listenoperationen. Beide Operationen haben in unsortierten Listen eine konstante Komplexität. Beim Löschen kommt allerdings vorab das Suchen des entsprechenden Elements hinzu. Mit den oben genannten Voraussetzungen ist aber auch das konstant.

6.3 Hashfunktionen

Im vorangehenden Abschnitt wurde eine Hashfunktion für die Komplexitätsanalyse vorausgesetzt, die jeden Schlüssel mit gleicher Wahrscheinlichkeit und unabhängig von den anderen Schlüsseln auf jeden der zur Verfügung stehenden Plätze der Hashtabelle abbildet. Gute Hashfunktionen sollten u. a. die Eigenschaft dieses einfachen gleichmäßigen Hashings erfüllen. Diese Eigenschaft ist aber meist nicht zu überprüfen, da die Wahrscheinlichkeitsverteilung von Schlüsseln meist nicht bekannt ist.

In der Praxis werden deshalb häufig heuristische Verfahren zur Definition von Hashfunktionen angewandt. Das Ziel ist es, Kollisionen so weit wie möglich zu vermeiden und, wenn nicht mehr vermeidbar, die verketteten Listen gleichmäßig zu verlängern.

Es ist üblich, die Schlüsselmenge $U \subset \mathbb{N}_0$ zu wählen. Ist das nicht der Fall, so kann jedes Schlüsselwort über einem Alphabet Σ umkehrbar eindeutig in eine nicht-negative ganze Zahl überführt werden. Im Folgenden seien also Schlüssel nicht-negative ganze Zahlen.

In der Literatur ist der wahrscheinlichkeitstheoretischen Analyse von Hashfunktionen oft ein großer Bereich gewidmet. Vergleiche dazu insbesondere [14]. Hier sei nur die *Divisionsmethode* angesprochen.

Es sei U die Schlüsselmenge, $k \in U$ ein Schlüssel, m die Anzahl der in der Hashtabelle zur Verfügung stehenden Plätze und $a \in \mathbb{R}^+$ eine geeignete Konstante. Die Hashfunktion $h : U \to \{0, \ldots, m-1\}$ mit

$$h(k) = ak \bmod m$$

liefert ein Hashing nach der *Divisionsmethode*. Zur Berechnung des Hashwertes sind nur eine Multiplikation und eine Division notwendig. Seine Berechnung hat, wie in dem vorangehenden Abschnitt gefordert, nur eine konstante Zeitkomplexität, hängt also weder von der Anzahl der gespeicherten Datensätze noch von der Tabellengröße ab.

Die Güte der Hashfunktion hängt nun von einer geeigneten Wahl von a und m bzgl. der gegebenen Schlüsselmenge U ab. Die Tabellengröße m ist dabei in Abhängigkeit von der Anzahl der zu speichernden Datensätze n (aktuell und zukünftig hinzukommende) und dem gewünschten Belegungsfaktor α festzusetzen.

Nach [1] kann eine Hashfunktion wie folgt konstruiert werden:

- Wähle $m \geq n/\alpha \geq 8$ als eine Zweierpotenz – die Division ist dann durch eine Schiebeoperation ausführbar – oder als eine Primzahl.
- Wähle als Faktor $a = 8 \lfloor m/23 \rfloor + 5$. (Sollte auf jeden Fall ungerade sein.)
- Falls die Schlüssel Zeichenketten sind und keine Elemente aus $\mathbb{N}_0$, also $k = k_1 k_2 \ldots k_t$, $k_i \in \Sigma$ einem Alphabet, $1 \leq i \leq t$:
  - Ordne jedem Zeichen aus Σ umkehrbar eindeutig eine nicht-negative ganze Zahl zu, z.B. aus dem Intervall $[0, |\Sigma| - 1]$ oder den jeweiligen ASCII-Code.
  - $h(k) = (ak_1 + a^2 k_2 + \ldots + a^t k_t) \bmod m$

    (Wende die Modulo-Operation nach jeder Multiplikation und jeder Addition an und mache von der Identität

    $$(ak_1 + a^2 k_2 + \ldots + a^t k_t) = (a(k_1 + a(k_2 + a(k_3 + \ldots + ak_t) \ldots)))$$

    Gebrauch.)

Beispiel 6.2
Es seien Schlüssel von ca. $n = 2000$ Datensätzen in einer Hashtabelle einzutragen. Akzeptabel sei die Überprüfung von 3 Elementen im Mittel, also ein Belegungsfaktor $\alpha = 3$. Eine angemessene Wahl für die Größe der Hashtabelle wäre daher wegen $n/\alpha \approx 667$ die Größe $m = 1024 = 2^{10}$, also

$$h(k) = 357k \bmod 1024.$$

□

6.4 Offene Adressierung

Vorangehend sind Hashtabellen mit Verkettung (geschlossene Adressierung) betrachtet worden. Bei einer Kollision werden die jeweiligen Datensätze in einer verketteten Liste gespeichert. Der Belegungsfaktor kann damit größer als 1 werden. Beim Hashing mit der so genannten *offenen Adressierung* werden alle Elemente, d. h. die Verweise auf die Datensätze, in der Hashtabelle selbst gespeichert. Es gibt keine verketteten Listen mehr, der Belegungsfaktor ist damit höchstens 1. Bei einer Kollision muss für ein neu zu speicherndes Element ein freier Platz in der Tabelle gesucht werden. Dieses Suchen nach einem freien Platz wird auch *Sondieren* genannt.

Die Hashfunktion wird dazu mit einer Sondierungszahl als weiterem Parameter versehen:

$$h : U \times \{0, 1, \ldots m-1\} \to \{0, 1, \ldots m-1\}$$

m ist wieder die Anzahl der Plätze in der Hashtabelle. Es wird gefordert, dass die Sondierungssequenz

$$(h(k, 0), h(k, 1), \ldots, h(k, m-1))$$

für jeden Schlüssel k eine Permutation von $0, 1, \ldots, m-1$ ist, so dass tatsächlich jeder Platz der Tabelle erreicht werden kann. Freie Plätze der Tabelle sind initial mit NIL gekennzeichnet. Die Sondierungszahl durchläuft die Werte von 0 bis $m-1$. Beginnend mit 0 wird sie bei einer auftretenden Kollision inkrementiert. Es entstehen auf diese Weise Kollisionsketten innerhalb der Hashtabelle.

Sind Elemente aus der Hashtabelle zu löschen, so dürfen diese nicht wieder mit NIL belegt werden. Die Plätze sind nur als gelöscht zu markieren, da sonst die Kollisionsketten unterbrochen werden würden. Ein freier Platz kennzeichnet das Ende einer Kollisionskette und dahinter liegende Elemente würden nicht mehr gefunden werden. Als gelöscht markierte Plätze dürfen wieder besetzt werden.

Drei Hashfunktionen mit Sondierung sollen hier kurz angesprochen werden. Dazu werden Hilfs-Hashfunktionen

$$h', h_1, h_2 : U \to \{0, 1, \ldots m-1\}$$

benötigt.

- **Lineares Sondieren**

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

  mit $c \in \mathbb{N}$. $c = 1$ garantiert eine maximale Sondierungssequenz

  $$T[h'(k)], T[h'(k)+1], \ldots, T[m-1], T[0], T[1], \ldots, T[h'(k)-1],$$

  aber bei einer Kollision wird ein freier Platz in unmittelbarer Nähe sondiert. Um solche lokalen Anhäufungen zu vermeiden, kann mit $c > 1$ ein freier Platz in größerer Entfernung sondiert werden. Teilerfremdheit zwischen c und m garantiert dann eine maximale Sondierungssequenz. In Abhängigkeit von dem Start-Hashwert $h'(k)$ gibt es m Sondierungssequenzen zu jedem c.

- **Quadratisches Sondieren**

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

  mit $c_1, c_2 \in \mathbb{N}$. Auch hier gibt es m verschiedene Sondierungssequenzen. Bei Kollisionen werden aber für größere i zunehmend entferntere Plätze sondiert. c_1 und c_2 sind so zu wählen, dass die Sondierungssequenz maximal ist.

- **Doppeltes Hashing**

  $$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$

  Aufgrund der zwei Hashfunktionen gibt es hier m^2 verschiedene Sondierungssequenzen. h_1 und h_2 sind so aufeinander abzustimmen, dass die Sondierungssequenz maximal ist.

Bleibt festzustellen, dass auch diese um die Sondierung erweiterten Hashfunktionen eine Komplexität aufweisen, die unabhängig von der Anzahl der Elemente in der Tabelle und von der Größe der Tabelle ist.

Beispiel 6.3
Betrachte eine Hashtabelle $T[0..m-1]$ mit offener Adressierung und einer Hashfunktion mit linearer Sondierung:

$$\begin{aligned} h'(k) &= ak \bmod m \\ h(k,i) &= (h'(k)+i) \bmod m \end{aligned}$$

Der Einfachheit halber werden in die Hashtabelle hier ausschließlich die Schlüssel eingetragen. Seien die Plätze $T[p], T[p+1], T[p+2], T[p+3], T[p+4]$ leer und füge die Schlüssel k_1, k_2, k_3, k_4 in dieser Reihenfolge ein. Diese Schlüssel haben die folgenden Hashwerte bzgl. h':

$$\begin{aligned} h'(k_1) &= p \\ h'(k_2) &= p+2 \\ h'(k_3) &= p \\ h'(k_4) &= p \end{aligned}$$

Bzgl. h werden dann die Hashwerte wie folgt berechnet und die jeweiligen Einträge in der Tabelle vorgenommen:

$$\begin{array}{lcll} h(k_1,0) & = & p\ , & T[p] \leftarrow k_1 \\ h(k_2,0) & = & p+2\ , & T[p+2] \leftarrow k_2 \\ h(k_3,0) & = & p\ , & \text{Kollision} \\ h(k_3,1) & = & p+1\ , & T[p+1] \leftarrow k_3 \\ h(k_4,0) & = & p\ , & \text{Kollision} \\ h(k_4,1) & = & p+1\ , & \text{Kollision} \\ h(k_4,2) & = & p+2\ , & \text{Kollision} \\ h(k_4,3) & = & p+3\ , & T[p+3] \leftarrow k_4 \end{array}$$

Abbildung 6.3 zeigt die Hashtabelle T zu diesem Zeitpunkt.

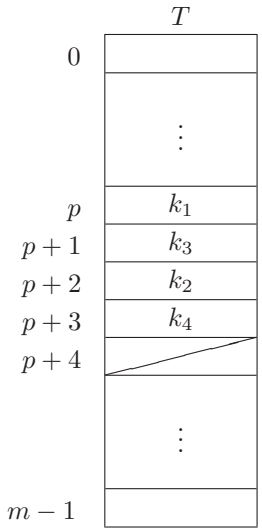


Abbildung 6.3: Hashtabelle T mit den eingetragenen Schlüsseln k_1, k_2, k_3, k_4

Es ergibt sich ausgehend von $T[p]$ die Kollisionskette $T[p+1], T[p+2], T[p+3]$. Mit $T[p+4] = \text{NIL}$ wird das Ende der Kette bestimmt. Beachte, in der Kette können auch Elemente enthalten sein, die nicht bzgl. h' den gleichen Hashwert haben.

Das Löschen eines nicht-letzten Schlüssels einer Kollisionskette, z. B. k_3, darf nur ein logisches Löschen sein, z. B. $T[p+1] \leftarrow -1$. Bei einem physischen Löschen, $T[p+1] \leftarrow \text{NIL}$, würde die Kollisionskette unterbrochen werden und z. B. k_4 würde nicht mehr gefunden werden.

□

6.5 Array Doubling

Das Hashing mit offener Adressierung lässt nur Belegungsfaktoren ≤ 1 zu. Mit größer werdendem Belegungsfaktor wird nun aber die Anzahl der auftretenden Kollisionen immer größer und damit die Zugriffszeit auf einen Datensatz immer ineffizienter. Erreicht die Anzahl der Datensätze die Tabellengröße kann dann sogar gar kein Datensatz mehr eingefügt werden. Wird andererseits die Hashtabelle gegenüber der Anzahl der zu speichernden Datensätze überdimensioniert, dann wird ggf. zu viel Speicherplatz verschwendet.

Eine Lösung dieses Problems kann mit dem so genannten *Array Doubling* erreicht werden. Die statische Datenstruktur der Hashtabelle wird dynamisch angepasst, je nach Belegungsfaktor. Zunächst ist eine initiale Tabellengröße festzulegen. Wird der Belegungsfaktor dann größer als eine bestimmter Wert, z. B. $0,7$, dann wird eine neue, doppelt so große Hashtabelle angelegt, und alle Einträge der alten Hashtabelle müssen umgespeichert werden. Die neuen Hashwerte werden mit der entsprechend modifizierten Hashfunktion bestimmt. War die alte Hashfunktion z. B.

$$h(k) = ak \bmod m,$$

dann ist die modifizierte Hashfunktion wie folgt definiert:

$$h'(k) = a'k \bmod 2m$$

Die Neuberechnung der Hashwerte mit dem damit verbundenen Umspeichern wird auch *Rehashing* genannt. Hierfür ist es sehr vorteilhaft, wenn in der Hashtabelle nur Zeiger auf die Datensätze gehalten werden. Die Datensätze müssen dann nämlich nicht umgespeichert werden.

Bei n Datensätzen zum Zeitpunkt des Verdoppelns der Tabelle, erfordert das Rehashing einen Aufwand von $\Theta(n)$. Dies ist gegenüber der konstanten Komplexität aller sonstigen Operationen eine sehr ineffiziente Operation. Sie kommt allerdings nicht so sehr häufig vor. Eine *amortisierte* Komplexitätsanalyse ergibt dann auch, dass über eine Sequenz von Operationen, von denen n die Komplexität $\Theta(1)$ haben und eine die Komplexität $\Theta(n)$ hat, jede Operation im Mittel die Komplexität $\Theta(1)$ aufweist, wenn bei einer anstehenden Vergrößerung der Tabelle diese jeweils verdoppelt wird. Man spricht von einer *amortisierten* Komplexität. Bei einer Vergrößerung der Tabelle jeweils um eine konstante Anzahl von Plätzen würde diese Komplexität abhängig von n bleiben.

Umgekehrt ist eine Hashtabelle dann zu verkleinern, d. h. zu halbieren, wenn der Belegungsfaktor kleiner als ein bestimmter Wert wird, z. B. $0,3$. Verbunden damit ist wieder ein entsprechendes Rehashing. Man spricht von *Array Halving*. Auch hierfür lässt sich eine konstante amortisierte Komplexität nachweisen.

Das Verfahren kann natürlich auch beim Hashing mit geschlossener Adressierung angewandt werden, wenn der Belegungsfaktor zu groß oder zu klein wird. Auch dort werden mit zu großem Belegungsfaktor die Zugriffszeiten zu ineffizient bzw. der belegte Speicherplatz unangemessen groß. Darüber hinaus kann das Array Doubling auch bei diversen anderen Verfahren, die auf der statischen Datenstruktur eines Feldes aufsetzen, eine Lösung zu dessen Dynamisierung bieten.

Literaturverzeichnis

[1] Baase, Sara; van Geldern, Allen: Computer Algorithms - Introduction to Design and Analysis, 3rd Edition.
Addison Wesley Longman Inc., Mass. 2000. ISBN 0-201-612244-5

[2] Church, Alonzo: An unsolvable Problem of elementary Number Theory.
In: American Journal of Mathematics 58, 1936, Seiten 345-363

[3] Church, Alonzo: A Note on the Entscheidungsproblem.
In: Journal of Symbolic Logic Vol. 1, Number 1, 1936, Seiten 40-41

[4] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.: Introduction to Algorithms.
McGraw-Hill Book Company, New York 1994. ISBN 0-07-013143-0

[5] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford: Algorithmen - Eine Einführung, 2. Auflage.
Oldenbourg Verlag, München 2007. ISBN 978-3-486-58262-8

[6] Euklid: Die Elemente, Bücher I-XIII.
(Übersetzt aus dem Griechischen von Clemens Thaer)
Verlag Harri Deutsch, Frankfurt 1997. ISBN 3-8171-3235-2

[7] Folkerts, Menso; Kunitzsch, Paul: Die älteste lateinische Schrift über das indische Rechnen nach al-Hwarizmi.
(Edition, Übersetzung und Kommentar)
Verlag der Bayerischen Akademie der Wissenschaften, München 1997.
ISBN 3-7696-0108-4

[8] Gödel, Kurt: Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I.
In: Monatshefte für Mathematik und Physik, 38, 1931, Seiten 173-198

[9] Hilbert, David: Mathematische Probleme. Vortrag auf dem 2. Internationalen Mathematikerkongreß, Paris 1900.
Verlag Harri Deutsch, Frankfurt 1998. ISBN 3-8171-3401-0

[10] Hilbert, David; Ackermann, Walter: Grundzüge der theoretischen Logik.
Springer, Berlin 1928.

[11] Ifrah, Georges: Universalgeschichte der Zahlen.
Campus Verlag, Frankfurt 1986. ISBN 3-593-33666-9

[12] Kleene, S. C.: General recursive functions of natural numbers.
In: Mathematische Annalen 112, 1936, Seiten 727 -742

[13] Knuth, Donald E.: The Art of Computer Programming, Vol. 1 Fundamental Algorithms.
Addison-Wesley, Reading 1973. ISBN 0-201-03809-9

[14] Knuth, Donald E.: The Art of Computer Programming, Vol. 2 Seminumerical Algorithms.
Addison-Wesley, Reading 1981. ISBN 0-201-03822-6

[15] Knuth, Donald E.: The Art of Computer Programming, Vol. 3 Sorting and Searching.
Addison-Wesley, Reading 1998. ISBN 0-201-89685-0

[16] Leibniz, Gottfried Wilhelm: Über die Analysis des Unendlichen.
(Übersetzt aus dem Lateinischen von Gerhard Kowalewski)
Verlag Harri Deutsch, Frankfurt 1998. ISBN 3-8171-3162-3

[17] Markov, A. A.: Theory of Algorithms.
Works of the Steklov Mathematical Institute, Vol. 42, 1954.
(Übersetzt aus dem Russischen vom Office of Technical Services, U.S. Department of Commerce, 1961)

[18] Post, Emil: A variant of a recursively unsolvable problem.
Bulletin of the AMS 52, 1946, Seiten 264-268. ISBN 3-8274-1092-4

[19] Schöning, Uwe: Algorithmik.
Spektrum Akademischer Verlag, Heidelberg 2001. ISBN 3-8274-1092-4

[20] Sigler, E. Laurence: Fibonacci's Liber Abaci - A Translation into Modern English of Leonardo Pisano's Book of Calculation.
Springer Verlag, New York 2002. ISBN 0-387-95419-8

[21] Tropfke, Johannes: Geschichte der Elementarmathematik. 4. Auflage, Band 1, Arithmetik und Algebra.
Verlag Walter de Gruyter, Berlin 1980. ISBN 3-11-004893-0

[22] Turing, Alan: On computable Numbers with an Application to the Entscheidungsproblem.
In: Proc. London Math. Soc., Vol. 2, 42, 1936, Seiten 230-265.

[23] Vogel, Kurt: Die Grundlagen der ägyptischen Arithmetik - in ihrem Zusammenhang mit der 2:n-Tabelle des Papyrus Rhind.
Kommissions-Verlag Michael Beckstein, München 1929.

[24] Vogel, Kurt: Adam Riese der deutsche Rechenmeister.
In: Deutsches Museum, Abhandlungen und Berichte, 27. Jahrgang 1959, Heft 3.
Verlag R. Oldenbourg, München 1959.

[25] Vogel, Kurt: Mohammed Ibn Musa Alchwarizmi's Algorismus.
(Nach der einzigen lateinischen Handschrift - Cambridge Un. Lib. Ii. 6.5.)
Otto Zeller Verlagsbuchhandlung, Aalen 1963.

[26] Wolff, Christian: Mathematisches Lexicon.
(Ergänzter reprografischer Nachdruck der Ausgabe Leipzig 1716 von J. E. Hofmann)
Georg Olms Verlagsbuchhandlung, Hildesheim 1965.

Index