Fakultät für Informatik
der Technischen Universität München
Lehrstuhl für Effiziente Algorithmen

Skriptum

zur Vorlesung

Grundlegende Algorithmen

gehalten im Sommer-Semester 1997

(Vorlesung für das Aufbaustudium Informatik)

Volker Heun

22. Oktober 1997
Version 1.01

Vorwort

Dieses Skript entstand parallel zur Vorlesung ‘Grundlegende Algorithmen’, die im Sommersemester 1997 für Studenten des Aufbaustudiums Informatik an der Technischen Universität München gehalten wurde.

Das Skript selbst umfaßt im wesentlichen die grundlegenden Themen, die man im Bereich Algorithmik einmal gehört haben sollte. Wegen der Kürze eines Sommersemester und des Umfangs von drei Semesterwochenstunden konnten nur die allerwichtigsten Teile vorgestellt werden.

Abschnitte, die in den Übungen behandelt wurden, da sie sich im wesentlichen aus den vorherigen Abschnitten ableiten ließen, wurden in diesem Skript mit (Ü) markiert. Abschnitte, auf die kurz und nicht sehr tief eingegangen wurde, sind mit (+) markiert. All diejenigen Abschnitte, die bedauerlicherweise nicht behandelt werden konnten, wurden mit (*) in diesem Skript markiert.

An dieser Stelle möchte ich insbesondere Frau Dr. Ulla Koppenhagen für das aufmerksame Korrekturlesen dieses Skripts sowie für zahlreiche Hinweise zu Fehlern, Ungenauigkeiten und Unstimmigkeiten danken. Für weitere Hinweise zu Tippfehlern möchte ich außerdem Herrn Marcello Mariucci danken.

Falls sich dennoch wietere (Tipp) Fehler unserer Aufmerksamkeit entzogen haben sollten, bin ich für jeden solchen Hinweis (an `heun@informatik.tu-muenchen.de`) dankbar.

Inhaltsverzeichnis

1 Einleitung und Grundlagen

1.1 Ziele der Vorlesung

Wenn man Probleme mit Hilfe von Rechnern lösen will, ist man nicht nur an deren Lösungen interessiert, sondern man will nachweisbar korrekte Lösungen in möglichst schneller Zeit auf möglichst kleinen Computern mit sehr einfachen Programmen erzielen.

Der Nachweis, daß ein Programm die Spezifizierungen der Problemstellung erfüllt, ist keine triviale Aufgabe. Es gibt mittleweile Methoden, mit denen man die Korrektheit von Programmen (bzgl. ihrer Spezifikation) unter bestimmten Voraussetzungen automatisch überprüfen kann. Es soll an dieser Stelle jedoch darauf hingewiesen werden, daß es im allgemeinen ein unlösbares Problem ist, die Korrektheit eines Programmes im Nachhinein festzustellen. Dies bedeutet, daß man schon bei der Entwicklung eines Algorithmus auf dessen Korrektheit achten muß.

Der Schwerpunkt dieser Vorlesung richtet sich aber darauf, für bestimmte Probleme effiziente, d.h. schnelle und speicherplatzsparende Algorithmen, vorzustellen, die natürlich auch korrekt sein werden. Leider ist es in der Praxis von einem korrekten Algorithmus immer noch ein weiter Weg zu einen korrekten Programm.

Viele der in dieser Vorlesung vorgestellten Lösungsmethoden haben sich als so universell herausgestellt, daß auch viele andere Probleme als die hier vorgestellten mit diesen Methoden gelöst werden können. Man sollte sich also beim Studium dieser Vorlesung nicht nur auf die Lösung spezieller Probleme beschränken, sondern man sollte immer ein Auge dafür offen haben, mit welcher Methode das Problem gelöst wurde. Wir werden an den betreffenden Stellen die allgemeine Methodik herausstellen, so daß man diese Lösungsmechanismen als allgemeine Methode erkennt.

Im einzelnen wollen wir diese allgemeinen Lösungsmethoden an den folgenden in der Informatik häufig auftretenden Problemen darstellen: Sortieren, Suchen in Texten, Speichern dynamischer Daten, Problemen auf Graphen, Arithmetik.

1.2 Einführendes Beispiel: Berechnung der Fibonacci-Zahlen

Zu Beginn wollen wir uns einem bekannten Problem widmen, dem Berechnen der n-ten Fibonacci-Zahl. Die Lösung hierfür ist an sich vielleicht nicht so interessant, allerdings liefert der Weg zu der Lösung bereits viele Aspekte, die in der Vorlesung an genauer untersucht werden sollen. Algorithmen zur Lösung dieses Problems wurden bereits in 'Informatik I' und Informatik III' präsentiert. Allerdings wurde hierbei nicht analysiert, welchen Zeitaufwand die verschiedenen Algorithmen zur Berechnung der n-ten Fibonacci-Zahl haben. Wir werden nun diese beiden Varianten noch einmal kurz vorstellen und deren 'Komplexität' analysieren. Zum Abschluß werden wir mit einer bereits in 'Informatik I' vorgestellten Methode einen Algorithmus entwerfen, der in Hinblick auf die 'Rechenzeit' die beiden bekannten Methoden schlagen wird.

Um den Anschluß an die Vorlesung 'Informatik I' zu halten, werden wir ein letztes Mal die hier vorgestellten Algorithmen in der funktionalen Programmiersprache `gofer` formulieren.

1.2.1 Rekursive Berechnung

Aus der Vorlesung 'Informatik I' ist die *rekursive* Variante zur Berechnung der n-ten Fibonacci-Zahl bekannt (Abbildung 1.1):

```
fib1 :: Int -> Int
fib1 1 = 1
fib1 2 = 1
fib1 n = fib1 (n-1) + fib1 (n-2)
```

Abbildung 1.1: Rekursive Definition der Fibonacci-Zahlen (`gofer`)

Um die Laufzeit dieser und der folgenden Varianten zu untersuchen, beschränken wir uns auf die Anzahl der ausgeführten arithmetische Operationen, wie Addition, Subtraktion, Multiplikationen und Divisionen. Bezeichne hierzu $C_{\text{rek}}(n)$ die Anzahl der ausgeführten arithmetischen Operationen zur Berechnung der n-ten Fibonacci-Zahl für die obige rekursive Variante. Damit ergibt sich aus der `gofer`-Definition die folgende Rekursionsgleichung für die arithmetische Komplexität:

$$
\begin{aligned}
C_{\text{rek}}(1) &= 0 \\
C_{\text{rek}}(2) &= 0 \\
C_{\text{rek}}(n) &= 3 + C_{\text{rek}}(n-1) + C_{\text{rek}}(n-2) \qquad \text{für } n \geq 3
\end{aligned}
$$

Zur Lösung dieser Rekursionsgleichung addieren wir auf beiden Seiten der letzten Gleichung 3:

$$3 + C_{\text{rek}}(n) = 3 + C_{\text{rek}}(n-1) + 3 + C_{\text{rek}}(n-2).$$

Definieren wir nun $D(n) := C_{\text{rek}}(n) + 3$, so erhalten wir die folgende Rekursionsgleichung:

$$\begin{aligned} D(1) &= 3 \\ D(2) &= 3 \\ D(n) &= D(n-1) + D(n-2) \qquad \text{für } n \geq 3 \end{aligned}$$

Diese Rekursionsgleichung gleicht der Rekursionsgleichung der Fibonacci-Zahlen bis auf die Anfangsbedingungen.

Lemma 1.1 *Sei $D(n)$ durch $D(1) = D(2) = \alpha$ und $D(n) = D(n-1) + D(n-2)$ definiert. Dann ist $D(n) = \alpha \cdot f(n)$.*

Beweis: Wir beweisen das Lemma durch vollständige Induktion über n.

Induktionsanfang ($n \in \{1, 2\}$): Offensichtlich ist die Behauptung erfüllt.

Induktionsschritt ($n - 1 \to n$):

$$\begin{aligned} D(n) &\overset{\text{Def.}}{=} D(n-1) + D(n-2) \\ &\overset{\text{I.V.}}{=} \alpha \cdot f_{n-1} + \alpha \cdot f_{n-2} \\ &= \alpha(f_{n-1} + f_{n-2}) \\ &\overset{\text{Def.}}{=} \alpha \cdot f_n \end{aligned}$$

■

Damit erhalten wir für die Anzahl der arithmetischen Operationen zur Berechnung der n-ten Fibonacci-Zahl auf Grund der rekursiven Variante:

$$C_{\text{rek}}(n) = D(n) - 3 = 3 \cdot f_n - 3.$$

Korollar 1.2 *Die Anzahl der Additionen und Subtraktionen zur Berechnung der n-ten Fibonacci-Zahl mit der rekursiven Variante ist $3(f_n - 1)$.*

Um ein besseres Gefühl für die Anzahl der arithmetischen Operationen zu bekommen, wollen wir für den Wert der n-ten Fibonacci-Zahl eine Abschätzung angeben.

Lemma 1.3 $\forall n \in \mathbb{N}. n > 2 : \; 2^{\lfloor \frac{n-1}{2} \rfloor} \leq f_n \leq 2^n.$

Beweis: Zunächst einmal stellen wir fest, daß $f_n \in \mathbb{N}$, also positiv ist. Damit folgt aus der Rekursionsgleichung, daß $(f_n)_{n\in\mathbb{N}}$ eine monoton wachsende Folge ist. Also gilt

$$\begin{aligned}
f_n &= f_{n-1} + f_{n-2} \\
&\geq 2f_{n-2} \\
&\geq 2^i f_{n-2i} \\
& \text{mit } i = \lfloor \tfrac{n-1}{2} \rfloor \text{ erhalten wir} \\
&\geq 2^{\lfloor \frac{n-1}{2} \rfloor} f_{n-2\lfloor \frac{n-1}{2} \rfloor} \\
& \text{mit } n-1 = \lfloor \tfrac{n-1}{2} \rfloor + \lceil \tfrac{n-1}{2} \rceil \text{ erhalten wir} \\
&\geq 2^{\lfloor \frac{n-1}{2} \rfloor} f_{1+\lfloor \frac{n-1}{2} \rfloor + \lceil \frac{n-1}{2} \rceil - 2\lfloor \frac{n-1}{2} \rfloor} \\
&\geq 2^{\lfloor \frac{n-1}{2} \rfloor} f_{\underbrace{1+\lceil \frac{n-1}{2} \rceil - \lfloor \frac{n-1}{2} \rfloor}_{\in\{1,2\}}} \\
& \text{da } f_1 = f_2 = 1 \\
&\geq 2^{\lfloor \frac{n-1}{2} \rfloor}
\end{aligned}$$

Hierbei ist für $x \in \mathbb{R}$ $\lfloor x \rfloor$ (bzw. $\lceil x \rceil$) die größte (kleinste) ganze Zahl n mit $n \leq x$ (bzw. $n \geq x$). Die Klammern $\lfloor \cdot \rfloor$ bzw. $\lceil \cdot \rceil$ werden auch *untere* bzw. *obere Gauß-Klammern* genannt.

Andererseits gilt

$$\begin{aligned}
f_n &= f_{n-1} + f_{n-2} \\
&\leq 2f_{n-1} \\
&\leq 2^i f_{n-i} \\
& \text{mit } i = n \text{ erhalten wir} \\
&\leq 2^n f_1 \\
&\leq 2^n
\end{aligned}$$

■

Damit benötigt die 'rekursive' Variante 'exponentiell' in n viele arithmetische Operationen zur Berechnung der n-ten Fibonacci Zahl.

1.2.2 Iterative Berechnung

Aus 'Informatik III' ist eine andere Variante zur Berechnung der n-ten Fibonacci-Zahl bekannt. Diese berechnet iterativ die Fibonacci-Zahlen durch Addition aus den jeweils zwei vorhergehenden Werten: $(f_n, f_{n-1}) = (f_{n-1} + f_{n-2}, f_{n-1})$. Dies läßt sich wie folgt in `gofer` mit Hilfe einer Endrekursion und Einbettung definieren (Abbildung 1.2).

```
fib2 n :: Int -> Int
fib2 n = x where (x,y) = g2 n

g2 :: Int -> Int -> Int
g2 1 = (1,undefined)
g2 2 = (1,1)
g2 n = let (x,y) = g2 (n-1)
       in (x+y,x)
```

Abbildung 1.2: ‘Iterative’ Definition der Fibonacci-Zahlen (`gofer`)

Für die Anzahl der arithmetische Operationen $C_{\text{iter}}(n)$ bei der Berechnung der n-ten Fibonacci-Zahl mit dieser ‘iterativen’ Definition erhalten wir die folgende Rekursionsgleichung:

$$\begin{aligned} C_{\text{iter}}(1) &= 0 \\ C_{\text{iter}}(2) &= 0 \\ C_{\text{iter}}(n) &= 2 + C_{\text{iter}}(n-1) \qquad \text{für } n \geq 3 \end{aligned}$$

Diese Rekursionsgleichung läßt sich leicht lösen:

$$\begin{aligned} C_{\text{iter}}(n) &= C_{\text{iter}}(n-1) + 2 \\ &= C_{\text{iter}}(n-i) + 2i \\ & \quad \text{mit } i = n-2. \\ &= C_{\text{iter}}(n-(n-2)) + 2(n-2) \\ & \quad \text{mit } C_{\text{iter}}(2) = 0 \\ &= 2n - 4 \end{aligned}$$

Korollar 1.4 *Die Anzahl der Additionen und Subtraktionen zur Berechnung der n-ten Fibonacci-Zahl mit der ‘iterativen’ Variante ist $2(n-2)$.*

2. Mai 97

1.2.3 Berechnung mit Hilfe des iterierten Quadrierens

Nun stellt sich die Frage, ob man mit noch weniger arithmetischen Operationen auskommt. Wir werden sehen, daß die Anzahl der arithmetische Operationen noch gedrückt werden kann, wenn auch Multiplikationen erlaubt sind.

Die in der 'iterativen' Variante verwendete Gleichung können wir auch wie folgt als Matrizenprodukt formulieren.

$$(f_n, f_{n-1}) = (f_{n-1} + f_{n-2}, f_{n-1}) = (f_{n-1}, f_{n-2}) \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}.$$

Durch Iteration dieser Gleichung erhält man das folgenden Lemma:

Lemma 1.5

$$(f_n, f_{n-1}) = (f_2, f_1) \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-2}.$$

Beweis: Wieder beweisen wir das Lemma mit vollständiger Induktion nach n

Induktionsanfang ($n = 2$):

$$(f_2, f_1) = (f_2, f_1) \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = (f_2, f_1) \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^0.$$

Induktionsschritt ($n - 1 \rightarrow n$):

$$\begin{aligned}
(f_n, f_{n-1}) &\overset{\text{Def.}}{=} (f_{n-1}, f_{n-2}) \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\
&\overset{\text{I.V.}}{=} (f_2, f_1) \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{(n-1)-2} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\
&= (f_2, f_1) \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-2}
\end{aligned}$$

■

Damit können wir nun die n-te Fibonacci-Zahl schnell berechnen, wenn wir n-te Potenzen schnell berechnen können. Auch hierfür erinnern wir uns wieder an 'Informatik I', siehe Abbildung 1.3.

```
pot :: Int -> Int -> Int
pot a n | n == 1 = a
        | even n = p*p   where p = pot a (n/2)
        | odd  n = a*p*p where p = pot a (n/2)
```

Abbildung 1.3: Iteriertes Quadrieren (`gofer`)

Wie wir in den 'Übungen zu Informatik I' gesehen haben, folgt die Korrektheit des Algorithmus aus den Potenzgesetzen. Diese Methode für das Potenzieren wird oft auch *iteriertes Quadrieren* genannt.

Wir wollen hier als Bemerkung festhalten, daß der obige Algorithmus beliebige ganzzahlige, positive Potenzen für beliebige Elemente einer gegebenen Halbgruppe $H = (M, \circ)$ (also einer Menge M mit einer assoziativen Verknüpfung $\circ$) berechnen kann. Damit läßt sich dieser Algorithmus auch für die Potenzierung von Matrizen einsetzen, da die Menge der $n \times n$ Matrizen mit der herkömmlichen Matrizenmultiplikation als assoziativer Verknüpfung eine Halbgruppe bilden.

Für eine natürliche Zahl $n \in \mathbb{N}$ bezeichne $(n)_2$ die Binärdarstellung ohne führende Nullen dieser Zahl. Für die Länge der Binärdarstellung ohne führende Nullen von n schreiben wir $\ell(n)$. Mit $\#_1(n)_2$ bezeichnen wir die Anzahl der 1'en in der Binärdarstellung ohne führende Nullen von n. Zunächst beweisen wir folgende elementare Beziehung zwischen n und der Länge seiner Binärdarstellung ohne führende Nullen.

Lemma 1.6 *Sei $n \in \mathbb{N}$, dann ist $\ell(n) = \lfloor \log(n) \rfloor + 1 = \lceil \log(n+1) \rceil$.*

Wir weisen an dieser Stelle hin, daß in diesem Skriptum alle Logarithmen zur Basis 2 zu verstehen sind. Sollte wirklich einmal ein anderer Logarithmus gemeint sein, so werden wir explizit darauf hinweisen, z.B. mit ln für den natürlichen Logarithmus zur Basis e oder lg für den Brigg'schen Logarithmus zur Basis 10. Außerdem wird im folgenden, wenn der Logarithmus als Funktion $\log : \mathbb{N}_0 \to \mathbb{N}_0$ aufgefaßt wird, der Wert für 0 wie folgt umdefiniert: $\log(0) = 0$.

Beweis: Wenn $n = 1$ ist, dann stimmt der Satz mit obigen Bemerkungen.

Sei also $n \in \mathbb{N}$ und $k \in \mathbb{N}_0$, so daß $2^{k-1} \leq n < 2^k$. Wie man sich leicht überzeugen kann, ist Länge der Binärdarstellung von n genau k. Aus $2^{k-1} \leq n$ folgt, daß $k-1 \leq \log(n)$. Da k ganzzahlig ist, folgt $k \leq \lfloor \log(n) \rfloor + 1$. Aus $n < 2^k$ folgt, daß $k > \log(n)$. Wegen der Ganzzahligkeit von k, impliziert dies nun, daß $k \geq \lfloor \log(n) \rfloor + 1$.

Den zweiten Teil der Behauptung beweist man analog, wobei man die Bedingung an k wie folgt umschreibt: $2^{k-1} - 1 < n \leq 2^k - 1$. ∎

Mit Hilfe der eben eingeführten Notationen können wir nun den folgenden Satz formulieren:

Theorem 1.7 *Die Funktion* `pot(a,n)` *benötigt für $n \in \mathbb{N}$ genau $\ell(n) + \#_1(n)_2 - 2$ viele Multiplikationen und $\ell(n) - 1$ viele Divisionen durch 2.*

Beweis: Der Beweis wird durch vollständige Induktion über die Länge der Binärdarstellung ohne führende Nullen von n geführt.

Induktionsanfang ($k := \ell(n) = 1$): Also ist $n = 1$ ($0 \notin \mathbb{N}$) Damit behauptet der Satz, daß $\ell(1) + \#_1(1)_2 - 2 = 0$ viele Multiplikationen und $\ell(1) - 1 = 0$ viele Divisionen durch 2 benötigt werden, was offensichtlich richtig ist.

Induktionsschritt ($k \to k + 1$):
1. Fall: Sei n gerade und $k + 1 = \ell(n)$

$$\begin{aligned}
\text{Anzahl Mult. in } \texttt{pot}(a, n) &= 1 + \text{Anzahl Mult. in } \texttt{pot}(a, \underbrace{n/2}_{\ell(n/2)=k}) \\
&\overset{\text{I.V.}}{=} 1 + \ell(n/2) + \#_1(n/2)_2 - 2 \\
&= 1 + (\ell(n) - 1) + \#_1(n)_2 - 2 \\
&= \ell(n) + \#_1(n)_2 - 2
\end{aligned}$$

Zum Übergang von der zweiten zur dritten Zeile sei angemerkt, daß beim Halbieren einer geraden Zahl die Anzahl der Einsen in der Binärdarstellung unverändert bleibt.

2. Fall: Sei n ungerade und $k + 1 = |n|$

$$\begin{aligned}
\text{Anzahl Mult. in } \texttt{pot}(a, n) &= 2 + \text{Anzahl Mult. in } \texttt{pot}(a, \underbrace{\lfloor n/2 \rfloor}_{\ell(\lfloor n/2 \rfloor)=k}) \\
&\overset{\text{I.V.}}{=} 2 + \ell(\lfloor n/2 \rfloor) + \#_1(\lfloor n/2 \rfloor)_2 - 2 \\
&= 2 + (\ell(n) - 1) + (\#_1(n)_2 - 1) - 2 \\
&= \ell(n) + \#_1(n)_2 - 2
\end{aligned}$$

Zum Übergang von der zweiten zur dritten Zeile sei angemerkt, daß beim Halbieren einer ungeraden Zahl die Anzahl der Einsen in der Binärdarstellung um eins abnimmt.

Der Induktionsschritt für die Anzahl der Divisionen sei dem Leser überlassen. ■

Da nach Definition der Matrizenmultiplikation für jede Matrizenmultiplikation acht Multiplikationen und vier Additionen ausreichend sind, erhalten wir den folgenden Satz:

Theorem 1.8 *Die auf Matrizenmultiplikation beruhende Variante zur Berechnung der n-ten Fibonacci-Zahl benötigt* $13\lfloor \log(n-2) \rfloor + 12\#_1(n-2)_2 - 11$ *viele arithmetische Operationen (Additionen, Subtraktionen, Multiplikationen, Divisionen durch 2).*

Beweis: Für die Berechnung der $(n-2)$-ten Potenz der Matrix werden werden $\ell(n-2) + \#_1(n-2)_2 - 2$ viele Matrizenmultiplikationen und $\ell(n-2) - 1$ viele Divisionen durch 2 auf ganzen Zahlen benötigt. Unter Verwendung von $\ell(n) = \lfloor \log(n) \rfloor + 1$ und der obigen Beobachtung über die Anzahl arithmetischer Operationen für eine Matrizenmultiplikation, benötigt die Berechnung der $n-2$-ten Potenz von

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

maximal

$$12\Big(\ell(n-2) + \#_1(n-2)_2 - 2\Big) + \ell(n-2) - 1 = 13\lfloor \log(n-2) \rfloor + 12\#_1(n-2)_2 - 12$$

arithmetische Operationen. Um die n-te Fibonacci-Zahl zu erhalten muß diese Matrix jetzt nur noch von links mit dem Vektor $(1, 1)$ multipliziert werden, wobei es ausreicht die erste Spalte der resultierenden Matrix

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-2}$$

zu addieren, also eine weitere Addition. ∎

Zum Schluß dieses Abschnitts wollen wir der Vollständigkeit halber noch die `gofer`-Definition für diese Variante der Fibonacci-Zahlen-Berechnung in Abbildung 1.4 angeben.

```
h :: Int -> (Int,Int,Int,Int)
h n | n==1   = (1, 1,
                1, 0)
    | even n = let (a,b,c,d)=h(n/2)
               in ((a*a)+(b*c), (a*b)+(b*d),
                   (c*a)+(d*c), (c*b)+(d*d))
    | odd  n = let (a,b,c,d)=h((n-1)/2)
               in ((a*a)+(b*c)+(a*b)+(b*d), (a*a)+(b*c),
                   (c*a)+(d*c)+(c*b)+(d*d), (c*a)+(d*c))

fib3 :: Int -> Int
fib3 1 = 1
fib3 2 = 1
fib3 n = a+c where (a,b,c,d) = h (n-2)
```

Abbildung 1.4: Definition der Fibonacci-Zahlen durch iteriertes Quadrieren (`gofer`)

Da symmetrische Matrizen unter Matrizenmultiplikation abgeschlossen sind (d.h. das Produkt zweier symmetrischer Matrizen ist wieder symmetrisch), läßt sich diese `gofer`-Definition noch optimieren. Diese weiter optimierte Variante ist im Anhang G angegeben. Ebenfalls im Anhang G findet man auch noch einmal die drei vorgestellten Varianten, wobei hier der Ausgabetyp `Integer` durch einen selbstdefinierten Typ `Long` ersetzt wurde, da die Fibonacci-Zahlen ja schnell den Bereich der in `gofer` darstellbaren `Integer`s verlassen.

6. Mai 97

1.3 Grundlagen

In diesem Abschnitt wollen wir die grundlegenden Begriffe herausarbeiten, die nötig sind, um Algorithmen möglichst maschinenunabhängig nach deren Geschwindigkeit und Platzbedarf einordnen zu können.

1.3.1 Registermaschine (RAM)

Eine *Registermaschine* (engl. *Random Access Machine*, kurz *RAM*) ist im wesentlichen ein idealisierter von Neumann Rechner. Abbildung 1.5 zeigt schematisch den Aufbau einer RAM. Sie besteht im wesentlichen aus einer unendliche Anzahl von Re-

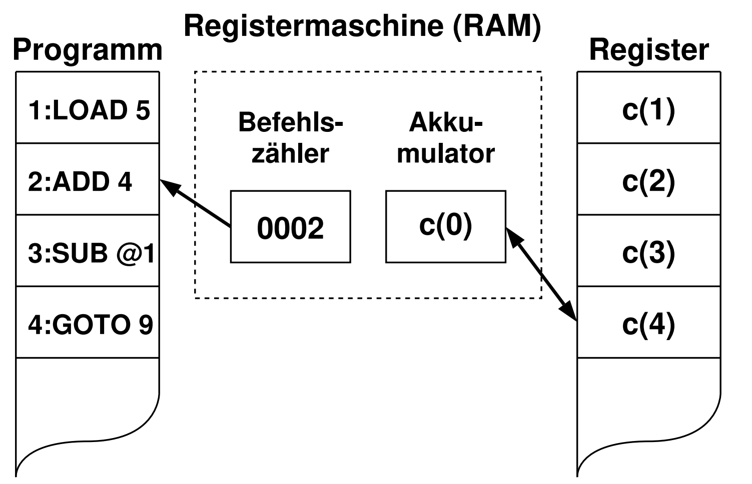

Abbildung 1.5: Die Registermaschine (Random Access Machine — RAM)

gistern, einem ausgezeichneten Register (dem *Akkumulator*) und dem *Befehlszähler*. Diese können beliebige ganze Zahlen speichern. Die Register sind in natürlicher

Weise von 0 aufsteigend mit den natürlichen Zahlen adressierbar. Dabei ist der Akkumulator das Register mit der Adresse 0. Den Inhalt von Register i bezeichnen wir im folgenden mit $c(i)$ (von engl. content). Alle Register zusammen bilden den eigentlichen Speicher der RAM.

Zusätzlich gibt es noch einen nur-lesbaren Speicher, in dem das auszuführende Programm abgespeichert ist. Im Programm sind die einzelnen Programmschritte ebenfalls mit den natürlichen Zahlen in fortlaufender Weise durchnumeriert. Zu Beginn der Berechnung wird der Inhalt des Befehlszähler, bezeichnet mit b, mit 1 initialisiert. Im Laufe der Berechnung zeigt der Befehlszähler die Nummer des aktuell ausgeführten Befehls an.

Nehmen wir an, das RAM-Programm soll die Funktion $f : \mathbb{Z}^n \to \mathbb{Z}^m$ berechnen. Zu Beginn der Berechnung stehen die Eingabedaten in den ersten n aufeinanderfolgenden Registern $1, \ldots, n$. Das Ergebnis befindet sich am Ende der Berechnung in vorher vereinbarten Registern. Der Einfachheit halber wollen wir immer annehmen, daß die Ergebnisse in den ersten m aufeinanderfolgenden Registern $1, \ldots, m$ hinterlegt sind.

	Operand		
Befehl	$\#i$	i	$@i$
LOAD	$c(0){:=}i;\ b{+}{+}$	$c(0){:=}c(i);\ b{+}{+}$	$c(0){:=}c(c(i));\ b{+}{+}$
STORE	—	$c(i){:=}c(0);\ b{+}{+}$	$c(c(i)){:=}c(0);\ b{+}{+}$
ADD	$c(0){:=}c(0){+}i;\ b{+}{+}$	$c(0){:=}c(0){+}c(i);\ b{+}{+}$	$c(0){:=}c(0){+}c(c(i));\ b{+}{+}$
SUB	$c(0){:=}c(0){-}i;\ b{+}{+}$	$c(0){:=}c(0){-}c(i);\ b{+}{+}$	$c(0){:=}c(0){-}c(c(i));\ b{+}{+}$
SHIFT	$c(0){:=}\lfloor c(0)/2 \rfloor;\ b{+}{+}$		
GOTO	—	$b{:=}i$	—
IF $?t$ GOTO	—	$b{:=}i$, wenn $c(0)?t$ $b{+}{+}$, sonst	—
END	Die Berechnung stoppt		
	Hierbei ist $? \in \{=, \neq, <, \leq, >, \geq\}$, $t \in \mathbb{Z}$		

Abbildung 1.6: Wirkungsweise der Operation einer Registermaschine

Die verfügbaren Befehle und deren Wirkungsweise sind in Abbildung 1.6 aufgelistet. Der Befehlssatz der RAM ist ziemlich puritanisch, aber es sei hier bemerkt, daß sich alle berechenbaren Funktionen $f : \mathbb{Z}^n \to \mathbb{Z}^m$ als RAM-Programm kodieren lassen. Die RAM kennt drei Adressierungsarten: unmittelbare, direkte und indirekte. Die unmittelbare Adressierung wird im Code durch #<Zahl> ausgedrückt. Dabei wird die im Befehl angegeben Zahl <Zahl> unmittelbar als Operand verwendet. Bei der direkten Adressierung <Zahl> wird als Operand der Inhalt des Registers <Zahl> (also $c($<Zahl>$)$) verwendet. Die Angabe @<Zahl> verweist auf eine indirekte

Adressierung, d.h. die Zahl im Befehl verweist auf das Register, in dem die Adresse des Registers steht die den Operanden beinhaltet, (also der Operand der Operation ist $c(c(\texttt{<Zahl>}))$).

Es sollte klar sein, daß sich jedes Programm auf eine RAM transformieren läßt. Wie man sich leicht vorstellen kann, ist es jedoch recht mühsam RAM-Programme zu schreiben. Daher verwenden wir zwar die RAM als Referenzmodell und werden auch die Zeit- und Platz-Komplexität in diesem Modell messen, aber wir werden nie explizit RAM-Programme angeben. Es sollte aber immer klar sein, wie sich ein gegebenes Programm in ein RAM-Programm konvertieren läßt und wie sich dabei die Anzahl der Operationen und die Anzahl der verwendeten Register verhält.

Um auch randomisierte Algorithmen betrachten zu können, erweitern wir die RAM um einen Befehl `RAND` zu einer *randomisierten RAM*. Die Operation `RAND` füllt den Akkumulator mit einer Zufallszahl aus dem Intervall $[0 : m-1]$, wobei m aus dem Operanden ermittelt wird. Hier und im folgenden bezeichne

$$[a : b] = [a, b] \cap \mathbb{Z} = \{a, a+1, \ldots, b-1, b\} \qquad \text{für } a \leq b.$$

Zum Abschluß als kleines Beispiel ein RAM Programm, daß zwei Zahlen in den Registern 1 und 2 multipliziert (siehe Abbildung 1.7). In den Zeilen `01-15` werden die

```
01:  LOAD 1
02:  IF ≥ 0 GOTO 16
03:  STORE 3
04:  LOAD 2
05:  STORE 1
06:  LOAD 3
07:  STORE 2
08:  LOAD 1
09:  IF ≥ 0 GOTO 16
10:  LOAD #0
11:  SUB 1
12:  STORE 1
13:  LOAD #0
14:  SUB 2
15:  STORE 2
16:  LOAD #0
17:  STORE 3
18:  LOAD 1
19:  IF = 0 GOTO 26
20:  SUB #1
21:  STORE 1
22:  LOAD 3
23:  ADD 2
24:  STORE 3
25:  GOTO18
26:  END
```

Abbildung 1.7: Ein RAM-Programm zur Multiplikation zweier ganzer Zahlen

Operanden zur Multiplikation vorbereitet, so daß in Register 1 ein positiver Operand steht. Dazu werden die Operanden eventuell vertauscht. Waren beide Operanden negativ, so werden sie beide von 0 subtrahiert, was gleichbedeutend damit ist, sie mit -1 zu multiplizieren. In den Zeilen `16-17` wird Register 3 auf 0 gesetzt. Die eigentliche Multiplikation findet durch fortgesetzte Addition in den Zeilen `18-25` statt.

1.3.2 Zeitkomplexität

Im folgenden werden wir den Begriff Komplexität eines Algorithmus formalisieren. Hierunter fallen insbesondere drei Gesichtspunkte der Komplexität: Zeit-, Platz- und Beschreibungskomplexität eines Algorithmus.

Zunächst einmal müssen wir uns über die Größe der Eingabe genauere Gedanken machen.

Definition 1.9 *Sei $x = (x_1, \ldots, x_m)$ eine Eingabe einer RAM, d.h. im Register i steht zu Beginn der Berechnung x_i. Die* uniforme Größe der Eingabe $x{=}(x_1, \ldots, x_m)$, *bezeichnet mit $\|x\|_{\mathrm{u}}$, ist die Anzahl der durch die Eingabe belegten Register, also m.*

Damit können wir nun die *Zeitkomplexität* von RAM-Programmen auf bestimmten Eingaben definieren.

Definition 1.10 *Die* uniforme Zeitkomplexität $T_M^{\mathrm{u}}(x)$ *einer RAM M auf der Eingabe $x = (x_1, \ldots, x_m)$ ist die Anzahl von Programmschritten die M auf x ausführt.*

Bei den Berechnung können natürlich beliebig große Zahlen in den Registern auftauchen. Von daher ist es unrealistisch anzunehmen, daß eine Additionen von zum Beispiel zwei 1000-stelligen Zahlen nur einen Rechenschritt kostet. Hierfür führt man das *logarithmische Kostenmaß* ein. Dabei werden die Operationen so gewichtet, daß die Kosten proportional zur Länge der Binärdarstellungen der auftretenden Zahlen ist. Als Beispiel nehmen wir den Befehl `ADD @100`. Sei im folgenden mit $\ell(x)$ die Länge der Binärdarstellung von x bezeichnet, also $\ell(x) = \lfloor \log(x) \rfloor + 1$. Im logarithmischen Kostenmaß sind die Kosten für diesen Befehl $\ell(c(0)) + \ell(i) + \ell(c(i)) + \ell(c(c(i)))$, da alle Zahlen $c(0)$, i, $c(i)$ und $c(c(i))$ entweder zur Addition oder zum Registerzugriff der benötigte Summanden verwendet werden. Eine tabellarische Übersicht über die logarithmischen Kosten der RAM-Operationen findet man in Abbildung 1.8. Da bereits in der Eingabe sehr große Zahlen auftreten können, definieren wir auch die logarithmische Größe einer Eingabe.

Definition 1.11 *Sei $x = (x_1, \ldots, x_n)$ eine Eingabe einer RAM, d.h. im Register i steht vor der Berechnung x_i. Die* logarithmische Größe *der Eingabe $x = (x_1, \ldots, x_n)$, bezeichnet mit $\|x\|_{\log}$, ist die Summe der Längen der Binärdarstellungen der x_i, also $\|x\|_{\log} = \sum_{i=1}^{n} \ell(x_i)$.*

Damit sind wir nun in der Lage die logarithmische Zeitkomplexität eines RAM-Programmes zu definieren.

	Operand		
Befehl	$\#i$	i	$@i$
LOAD	$\ell(i)$	$\ell(i)+\ell(c(i))$	$\ell(i)+\ell(c(i))+\ell(c(c(i)))$
STORE	—	$\ell(i)+\ell(c(0))$	$\ell(i)+\ell(c(i))+\ell(c(0))$
ADD	$\ell(c(0))+\ell(i)$	$\ell(c(0))+\ell(i)+\ell(c(i))$	$\ell(c(0))+\ell(i)+\ell(c(i))+\ell(c(c(i)))$
SUB	$\ell(c(0))+\ell(i)$	$\ell(c(0))+\ell(i)+\ell(c(i))$	$\ell(c(0))+\ell(i)+\ell(c(i))+\ell(c(c(i)))$
RAND	$\ell(i)$	$\ell(i)+\ell(c(i))$	$\ell(i)+\ell(c(i))+\ell(c(c(i)))$
SHIFT	$\ell(c(0))$		
GOTO	—	1	—
IF $?t$ GOTO	—	$\ell(c(0))+\ell(t)$	—
END	1		
	$\ell(i)=\lfloor\log(i)\rfloor+1$		

Abbildung 1.8: Logarithmische Kosten von RAM-Operationen

Definition 1.12 *Die* logarithmische Zeitkomplexität $T_M^{\log}(x)$ *einer RAM M auf einer Eingabe $x=(x_1,\ldots,x_m)$ ist die Summe der logarithmischen Kosten der einzelnen Schritte, die M auf x ausführt.*

Damit können wir nun allgemein die Zeitkomplexität von RAM-Programmen definieren.

Definition 1.13 *Sei $t:\mathbb{N}\to\mathbb{N}$ eine Funktion. Eine RAM heißt uniform bzw. logarithmisch* $t(n)$-zeitbeschränkt, *wenn für alle Eingaben x der Größe $n=\|x\|_{\mathrm{u}}$ bzw. $n=\|x\|_{\log}$ die uniforme bzw. logarithmische Zeitkomplexität kleiner gleich $t(n)$ ist. Also*

$$T_M^{\mathrm{u}}(n):=\max\{T_M^{\mathrm{u}}(x):\|x\|_{\mathrm{u}}=n\}\leq t(n)$$

bzw.

$$T_M^{\log}(n):=\max\{T_M^{\log}(x):\|x\|_{\log}=n\}\leq t(n).$$

Diese Definition gibt die Zeitkomplexität eines RAM-Programmes im schlimmsten Falle an (engl. worst-case). Manchmal sind Algorithmen aber nur für sehr wenige Eingaben sehr langsam und für die meisten Eingaben sehr schnell. Hierfür ist das oben eingeführte Komplexitätsmaß eher ungeeignet. Deshalb führt man auch noch den Begriff der mittleren Zeitkomplexität ein, in der die Laufzeiten der einzelnen Eingaben gemäß einer Wahrscheinlichkeitsverteilung eingehen. Sei hierzu $\varphi_n(x)$ eine Wahrscheinlichkeitsverteilung über alle Eingaben x der Größe $\|x\|=n$ (je nach Kontext uniform oder logarithmisch), d.h. $\sum_{x:\|x\|=n}\varphi_n(x)=1$.

Definition 1.14 *Sei $\Phi = \{\varphi_n : n \in \mathbb{N}\}$ eine Familie von Wahrscheinlichkeitsverteilungen und sei $t : \mathbb{N} \to \mathbb{N}$ eine Funktion. Eine RAM heißt* im Mittel uniform bzw. logarithmisch $t(n)$-zeitbeschränkt, *wenn*

$$\bar{T}^{\mathrm{u}}_{M,\Phi}(n) := \sum_{x:\|x\|_{\mathrm{u}}=n} \varphi_n(x) \cdot T^{\mathrm{u}}_M(x) \le t(n)$$

bzw.

$$\bar{T}^{\log}_{M,\Phi}(n) := \sum_{x:\|x\|_{\log}=n} \varphi_n(x) \cdot T^{\log}_M(x) \le t(n)$$

Im allgemeinen wird man für die Familie $\Phi = \{\varphi_n\}$ annehmen, daß jedes Mitglied φ_n die Gleichverteilung auf den Eingaben der Größe n ist, da im allgemeinen keine sinnvollen Aussagen über eine Verteilung der Eingaben gemacht werden kann.

Für randomisierte RAM-Programme sind die Laufzeiten $T^{\mathrm{u,log}}_M(n)$ als erwartete Laufzeiten zu interpretieren.

Definition 1.15 *Die* erwartete uniforme bzw. logarithmische Zeitkomplexität *einer randomisierten RAM M auf der Eingabe $x = (x_1, \ldots, x_m)$, bezeichnet mit $\hat{T}^{\mathrm{u}}_M(x)$ bzw. $\hat{T}^{\log}_M(x)$, ist die erwartete Anzahl von Programmschritten, die M auf x ausführt, bzw. die Summe der erwarteten logarithmischen Kosten der einzelnen Schritte, die M auf x ausführt.*

Damit können wir eine Zeitkomplexität für randomisierte RAM-Programme einführen.

Definition 1.16 *Sei $t : \mathbb{N} \to \mathbb{N}$ eine Funktion. Eine randomisierte RAM heißt uniform bzw. logarithmisch* $t(n)$-erwartet-zeitbeschränkt, *wenn für alle Eingaben x der Größe $n = \|x\|_{\mathrm{u}}$ bzw. $n = \|x\|_{\log}$ die erwartete uniforme bzw. logarithmische Zeitkomplexität kleiner gleich $t(n)$ ist. Also*

$$\hat{T}^{\mathrm{u}}_M(n) := \max\{\hat{T}^{\mathrm{u}}_M(x) : \|x\|_{\mathrm{u}} = n\} \le t(n)$$

bzw.

$$\hat{T}^{\log}_M(n) := \max\{\hat{T}^{\log}_M(x) : \|x\|_{\log} = n\} \le t(n).$$

1.3.3 Platzkomplexität

Ein weiteres wichtiges Komplexitätsmaß ist die *Platzkomplexität*, das ist der Speicherplatzbedarf einer Berechnung eines RAM-Algorithmus.

Definition 1.17 *Die* uniforme Platzkomplexität $S^{\mathrm{u}}_M(x)$ *einer RAM von M auf der Eingabe $x = (x_1, \ldots, x_m)$ ist die Anzahl von Registern die M bei der Berechnung auf x benutzt.*

Im Gegensatz zur uniformen Zeitkomplexität ist die Definition der uniformen Platzkomplexität nicht sonderlich nützlich, da man zeigen kann, daß sich z.B. alle Funktionen $f : \mathbb{N} \to \mathbb{N}$ mit nur zwei Registern (und dem Akkumulator) berechnen lassen ($\to$ Komplexitätstheorie; Stichwort Zählermaschinen (engl. Counter Machines)).

Definition 1.18 *Die* logarithmische Platzkomplexität $S^{\log}_M(x)$ *einer RAM M auf der Eingabe $x = (x_1, \ldots, x_m)$ ist*

$$\sum_{i=0}^{\infty} \max_j \{\ell(j) : j \textit{ kommt in der Berechnung von } M \textit{ auf } x \textit{ in Register } i \textit{ vor}\}.$$

Hierbei gelte die folgende Definition $\max\{\} := 0$. Der Satz '*j kommt in der Berechnung von M auf x in Register i vor*' bedeutet, die Zahl j wird bei einer Berechnung von M auf x in irgendeiner Art und Weise bei einem Registerzugriff verwendet. Das kann sowohl das Laden oder Speichern einer solchen in eine Register sein, als auch eine Zahl, die bei einem indirekten Registerzugriff auftritt.

Damit können wir nun allgemein die Platzkomplexität von RAM-Programmen definieren.

Definition 1.19 *Sei $s : \mathbb{N} \to \mathbb{N}$ eine Funktion. Eine RAM heißt uniform bzw. logarithmisch* $s(n)$-platzbeschränkt, *wenn für alle Eingaben x der Größe $n = \|x\|_{\mathrm{u}}$ bzw. $n = \|x\|_{\log}$ die uniforme bzw. logarithmische Platzkomplexität kleiner gleich $s(n)$ ist. Also*

$$S^{\mathrm{u}}_M(n) := \max\{S^{\mathrm{u}}_M(x) : \|x\|_{\mathrm{u}} = n\} \leq s(n)$$

bzw.

$$S^{\log}_M(n) := \max\{T^{\log}_M(x) : \|x\|_{\log} = n\} \leq s(n).$$

Analog wie bei der Zeitkomplexität kann man die erwartete Platzkomplexität von RAM-Programmen definieren.

Definition 1.20 *Sei $\Phi = \{\varphi_n : n \in \mathbb{N}\}$ eine Familie von Wahrscheinlichkeitsverteilungen und sei $s : \mathbb{N} \to \mathbb{N}$ eine Funktion. Eine RAM heißt* im Mittel *uniform bzw. logarithmisch $t(n)$-platzbeschränkt, wenn*

$$\bar{S}^{\mathrm{u}}_{M,\Phi}(n) := \sum_{x:\|x\|_{\mathrm{u}}=n} \varphi_n(x) \cdot S^{\mathrm{u}}_M(x) \leq s(n)$$

bzw.

$$\bar{S}^{\log}_{M,\Phi}(n) := \sum_{x:\|x\|_{\log}=n} \varphi_n(x) \cdot S^{\log}_M(x) \leq s(n)$$

Letztendlich definieren wir noch die erwartete Platzkomplexität einer randomisierten RAM.

Definition 1.21 *Die* erwartete uniforme bzw. logarithmische Platzkomplexität *einer randomisierten RAM M auf der Eingabe $x = (x_1, \ldots, x_m)$, bezeichnet mit $\hat{S}^{\mathrm{u}}_M(x)$ bzw. $\hat{S}^{\log}_M(x)$, ist die erwartete Anzahl von Registern, die M bei der Berechnung auf x benutzt, bzw. der Erwartungswert der folgenden Summe*

$$\sum_{i=0}^{\infty} \max_j \{\ell(j) : j \text{ kommt in der Berechnung von } M \text{ auf } x \text{ in Register } i \text{ vor}\}.$$

Somit erhalten wir letztendlich noch die folgende Definition.

Definition 1.22 *Sei $s : \mathbb{N} \to \mathbb{N}$ eine Funktion. Eine randomisierte RAM heißt uniform bzw. logarithmisch $s(n)$-*erwartet-platzbeschränkt*, wenn für alle Eingaben x der Größe $n = \|x\|_{\mathrm{u}}$ bzw. $n = \|x\|_{\log}$ die erwartete uniforme bzw. logarithmische Zeitkomplexität kleiner gleich $s(n)$ ist. Also*

$$\hat{S}^{\mathrm{u}}_M(n) := \max\{\hat{S}^{\mathrm{u}}_M(x) : \|x\|_{\mathrm{u}} = n\} \leq s(n)$$

bzw.

$$\hat{S}^{\log}_M(n) := \max\{\hat{S}^{\log}_M(x) : \|x\|_{\log} = n\} \leq s(n).$$

1.3.4 Beschreibungskomplexität

Ein in dieser Vorlesung nicht so wichtiges Komplexitätsmaß sei nur der Vollständigkeit halber erwähnt: die *Beschreibungskomplexität*. Diese mißt die Komplexität des Programmes selbst und wird daher als die Anzahl der der Instruktionen des RAM-Programmes definiert. Natürlich ist man in der Regel an besonders kurzen Programmen, d.h. an Programmen mit kleiner Beschreibungskomplexität, interessiert. Insbesondere bei Programmen, die nur einmal oder sehr selten aufgerufen werden, lohnt es sich kaum viel Energie in den Entwurf schneller und speicherplatzsparender Programme zu stecken, da die verbrauchte Zeit und Arbeitskraft bei der geringen Anzahl an Programmaufrufen hier überwiegt. Wie wir im folgenden sehen werden, treibt der Entwurf schneller und/oder speicherplatzsparender Programme nicht selten die Beschreibungskomplexität nach oben. Hier muß man immer im einzelnen abwägen, wieviel Energie einem eine Verbesserung eines Algorithmus im Hinblick auf Zeit- und Platzkomplexität die Entwicklungszeit und die Beschreibungskomplexität eines solchen Algorithmus wert ist.

9. Mai 97

1.3.5 Landau'sche Symbole

Wie wir gesehen haben sind genaue Abschätzungen nicht ganz einfach und zum Teil auch gar nicht durchführbar. Daher werden wir uns bei Angaben von Zeit- und Platzkomplexitätsschranken mit asymptotischen Abschätzungen zufrieden geben.

Erinnern wir uns an die Anzahl arithmetischer Operationen bei der dritten Variante zur Berechnung der n-ten Fibonacci-Zahl (basierend auf dem iterierten Quadrieren von Matrizen). Wir hatten hierfür eine obere Schranke der Form $c \cdot \log(n)$ für eine Konstante c herausbekommen. Die genaue Abschätzung der Konstanten c war mühselig und hing auch noch von der Anzahl der 1'en in der Binärdarstellung von $n-2$ ab. Daher werden wir uns bei der Analyse von Algorithmen nicht so genau auf die Proportionalitätskonstanten achten (sofern diese nicht 'zu groß' werden).

Bei der Analyse von Algorithmen werden wir die Komplexität meist mit Hilfe der *Landau'schen Symbole* (auch *Groß-Oh-Notation* genannt) angeben. Diese verschleiern die genauen Größen dieser Proportionalitätskonstanten. Die asymptotischen Resultate sind jedoch für einen qualitativen Vergleich von Algorithmen im allgemeinen aussagekräftig genug.

Seien im folgenden $f, g : \mathbb{N} \to \mathbb{R}_+$ Funktionen ($\mathbb{R}_+^*$ bezeichne die nichtnegativen reellen Zahlen):

$$\begin{aligned}
f = O(g) \quad &:\Leftrightarrow \quad \exists c \in \mathbb{R}.c > 0,\ n_0 \in \mathbb{N} : \ \forall n \in \mathbb{N}.n \geq n_0 : \quad f(n) \leq c \cdot g(n) \\
f = \Omega(g) \quad &:\Leftrightarrow \quad \exists c \in \mathbb{R}.c > 0,\ n_0 \in \mathbb{N} : \ \forall n \in \mathbb{N}.n \geq n_0 : \quad f(n) \geq c \cdot g(n) \\
f = \Omega_\infty(g) \quad &:\Leftrightarrow \quad \exists c \in \mathbb{R}.c > 0 : \ \forall n \in \mathbb{N} : \ \exists m \in \mathbb{N}.m > n : \quad f(m) \geq c \cdot g(m) \\
f = \Theta(g) \quad &:\Leftrightarrow \quad f = O(g) \wedge g = O(f) \\
f = o(g) \quad &:\Leftrightarrow \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \\
f = \omega(g) \quad &:\Leftrightarrow \quad \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0
\end{aligned}$$

Für das folgenden, ist es wichtig ein Gefühl dafür zu bekommen, welche Beziehungen bzgl. der Landau'schen Symbole für zwei gegebene Funktionen $f, g : \mathbb{N} \to \mathbb{N}$ gelten. Dazu betrachten wir zunächst ein paar Beispiele und elementare Rechenregeln.

1. $\forall k, \ell \in \mathbb{N}.k > \ell : \quad n^\ell = o(n^k)$:

   Weil $$\lim_{n\to\infty} \frac{n^\ell}{n^k} = \lim_{n\to\infty} n^{\ell-k} = 0.$$

2. $\forall k, \ell \in \mathbb{N}. k > \ell : \quad n^k + n^\ell = \Theta(n^k)$:

   Weil $\quad n^k + n^\ell \leq 2 \cdot n^k, \quad$ gilt $\quad n^k + n^\ell = O(n^k)$.

   Und weil $\quad n^k \leq n^k + n^\ell, \quad$ gilt $\quad n^k = O(n^k + n^\ell)$.

3. $\forall k \in \mathbb{N} : \quad n^k = o(2^n)$:

   Weil
   $$\lim_{n\to\infty} \frac{n^k}{2^n} = \lim_{n\to\infty} \frac{2^{k\log(n)}}{2^n} = \lim_{n\to\infty} 2^{k\log(n)-n} = 0.$$
   Es bleibt nur zu zeigen, daß $n - k\log(n) \to \infty$ für $n \to \infty$. Es gilt:
   $$\forall n \in \mathbb{N} : \quad \ln(n) \leq \frac{n}{\ell} + \ell \qquad (*)$$
   Damit erhalten wir für $\ell = 4k$ mit $(*)$:
   $$\lim_{n\to\infty} (n - k \underbrace{\log(n)}_{=\frac{\ln(n)}{\ln(2)} \leq 2\ln(n)}) \geq \lim_{n\to\infty} \left(n - 2k\left(\frac{n}{4k} + 4k\right)\right) = \lim_{n\to\infty} \left(\frac{n}{2} - 8k^2\right) = \infty.$$
   Wir müssen nur noch $(*)$ zeigen:

   Für $n = \ell$ gilt $(*)$ offensichtlich, da $\ln(\ell) < \ell + 1$. Da für alle $n > \ell$:
   $$\frac{\partial}{\partial n} \ln(n) = \frac{1}{n} < \frac{1}{\ell} = \frac{\partial}{\partial n}\left(\frac{n}{\ell} + \ell\right),$$
   gilt $(*)$ für $n \geq \ell$.

   Für $n < \ell$ ist aber sicherlich $\ln(n) < n < \ell$. Also ist
   $$\ln(n) \leq \max\left\{\frac{n}{\ell}, \ell\right\} < \frac{n}{\ell} + \ell.$$

4. $\forall k \in \mathbb{N} : \ \forall \varepsilon \in \mathbb{R}. \varepsilon > 0 : \quad \log^k(n) = o(n^\varepsilon)$

   Übungsaufgabe (ähnlich wie 3.).

5. $\forall n \in \mathbb{N} : \quad 2^n = o(2^{2n})$

   Weil $\quad \lim_{n\to\infty} \frac{2^n}{2^{2n}} = \lim_{n\to\infty} 2^{-n} = 0.$

6. $f = \Omega(g) \ \Rightarrow \ f = \Omega_\infty(g)$

7. $f = \Omega_\infty(g) \ \not\Rightarrow \ f = \Omega(g)$

   Sei $f = 1 + (1 + (-1)^n)n^2$ und $g(n) = n$.

8. $f = O(g) \Leftrightarrow g = \Omega(f)$ und $f = o(g) \Leftrightarrow g = \omega(f)$
   Folgt unmittelbar aus der Definition.
9. $f = \mathcal{O}(g) \wedge g = \mathcal{O}(h) \Rightarrow f = \mathcal{O}(h)$ für $\mathcal{O} \in \{O, \Omega, \Theta, o, \omega\}$.
   Das heißt, die Landau'sche Symbole sind transitiv. Diese Beobachtung folgt ebenfalls unmittelbar aus den Definitionen.
10. $f = \Omega_\infty(g) \wedge g = \Omega_\infty(h) \not\Rightarrow f = \Omega_\infty(h)$
    Sei $f(n) = 1+(1+(-1)^n)n$, $g(n) = n$, und $h(n) = n+(1+(-1)^n)n^2 = n \cdot f(n)$.
11. Wenn g nur endliche viele Nullstellen hat, dann gilt:
    $$f = O(g) \quad \Leftrightarrow \quad \exists c \in \mathbb{R}. c > 0 : \quad \limsup_{n\to\infty} \frac{f(n)}{g(n)} \leq c$$
12. Wenn g nur endliche viele Nullstellen hat, dann gilt:
    $$f = \Omega(g) \quad \Leftrightarrow \quad \exists c \in \mathbb{R}. c > 0 : \quad \liminf_{n\to\infty} \frac{f(n)}{g(n)} \geq c$$

Nun führen wir noch ein paar abkürzende Sprechweisen ein, um das Wachstum von Funktionen zu beschreiben. Sei hierzu $f : \mathbb{N} \to \mathbb{N}$.

- Wir sagen f ist *konstant*, wenn $f(n) = \Theta(1)$.
- Wir sagen f wächst *logarithmisch*, wenn $f(n) = O(\log(n))$.
- Wir sagen f wächst *polylogarithmisch*, wenn $f(n) = O(\log^k(n))$ für ein $k \in \mathbb{N}$.
- Wir sagen f wächst *linear*, wenn $f(n) = O(n)$.
- Wir sagen f wächst *quadratisch*, wenn $f(n) = O(n^2)$.
- Wir sagen f wächst *polynomiell*, wenn $f(n) = O(n^k)$ für ein $k \in \mathbb{N}$.
- Wir sagen f wächst *superpolynomiell*, wenn $f(n) = \omega(n^k)$ für alle $k \in \mathbb{N}$.
- Wir sagen f wächst *subexponentiell*, wenn $f(n) = o(2^{cn})$ für alle $0 < c \in \mathbb{R}$.
- Wir sagen f wächst *exponentiell*, wenn $f(n) = O(2^{cn})$ für ein $0 < c \in \mathbb{R}$.

Manchmal fordert man für die obigen Sprechweisen auch, daß neben O auch Ω bzw. zumindest Ω_∞ gelten soll. Also eine Funktion f wächst logarithmisch, wenn $f(n) = O(\log(n))$ und $f(n) = \Omega(\log(n))$ bzw. $f(n) = \Omega_\infty(\log(n))$ gilt.

13.Mai 97

2 Sortieren

2.1 Einfache Sortieralgorithmen

Sortieren ist eines der am häufigsten auftauchenden und am besten untersuchten Probleme in der Informatik. Dennoch tauchen immer wieder verbesserte Algorithmen auf, wo man zunächst überhaupt keine Verbesserung erwartet hatte.

Doch zunächst einmal wollen wir das Problem des Sortierens genauer formulieren. Gegeben sei eine Menge von n Datensätzen D_i, wobei jeder Datensatz einen Schlüssel k_i besitzt, so daß auf der Menge der möglichen Schlüssel eine totale Ordnung $\leq$ definiert ist. Eine solche Menge von Datensätzen heißt sortiert, wenn die Datensätze so permutiert vorliegen, daß die entsprechenden Schlüssel eine, bzgl. der totalen Ordnung $\leq$, aufsteigende oder absteigende Folge bilden. Also wenn für diese Permutation π auf $[1:n]$ gilt, daß:

$$k_{\pi^{-1}(1)} \leq \cdots \leq k_{\pi^{-1}(n)} \quad \text{bzw.} \quad k_{\pi^{-1}(1)} \geq \cdots \geq k_{\pi^{-1}(n)}$$

Im ersten Fall, nennen wir die Folge aufsteigend, im zweiten Fall absteigend sortiert.

Wir haben hier noch nichts darüber ausgesagt, wie diese Datensätze abgespeichert sind, bzw. wie wir darauf zugreifen können. Eine Möglichkeit ist, daß wir auf die Datensätze nur sequentiell zugreifen können, das bedeutet, daß die Folge im wesentlichen in Form einer linearen Liste abgespeichert ist. Andererseits können die Datensätze auch so gegeben sein, daß wir direkt auf jeden Datensatz zugreifen können, das bedeutet, daß die Datensätze im wesentlichen in Form eines Arrays abgespeichert sind. Für die asymptotische Analyse spielt es keine Rolle, in welcher Form die zu sortierenden Daten vorliegen, da die eine in die andere Form in linearer Zeit umgespeichert werden kann.

Außerdem ist zu jeder totalen Ordnung $\leq$ immer eine Vergleichsfunktion `compare` gegeben, die zwei Schlüssel x und y vergleichen kann und als Ergebnis -1, 0 oder $+1$ liefert, je nachdem ob, $x < y$, $x = y$ oder $x > y$ war. In den folgenden Analysen der diversen Sortieralgorithmen werden wir die Komplexität eines Sortieralgorithmus immer in der Anzahl der Vergleiche angeben, da alle anderen Operationen meist linear durch die erforderlichen Vergleiche beschränkt sind (wo dies nicht der Fall ist, werden wir explizit darauf hinweisen).

Wir wollen an dieser Stelle nur kurz darauf hinweisen, daß es im Hinblick auf die Rechenzeit sinnvoll ist, daß nie ein Datensatz selbst kopiert wird. Es werden immer nur die Referenzen (also Pointer) auf diese Datensätze umkopiert. Bei größeren Datensätzen erspart uns dies einen Großteil der Arbeit, da es natürlich wesentlich effizienter ist einen Pointer (repräsentiert durch wenige Bytes) zu kopieren als einen ganzen Datensatz mit eventuell mehreren tausend Bytes.

Im Rest dieses Abschnittes wollen wir zwei einfache Sortieralgorithmen vorstellen. Der einfacheren Beschreibung der Algorithmen wegen, nehmen wir an, daß die Schlüssel ganze Zahlen sind und daß jeder Datensatz nur aus einem Schlüssel besteht. Die Verallgemeinerung auf komplexere Datensätze mit beliebigen total geordneten Schlüsseln ist sehr einfach. Die Daten werden im folgenden immer in aufsteigender Folge sortiert.

2.1.1 Sortieren durch Auswahl

Der wohl einfachste Sortieralgorithmus ist der durch Auswahl. Während des Algorithmus stellen wir uns die Datensätze immer in zwei Teile aufgeteilt vor: einen bereits aufsteigend sortierten Teil und einen unsortierten Teil. Zu Beginn befinden sich alle Daten im unsortierten Bereich. Wir bestimmen nun in jedem Schritt das kleinste Element im unsortierten Bereich und fügen es an das Ende des sortierten Bereiches an. Diese Beschreibung ist noch recht informal, erläutert aber ziemlich genau was der Algorithmus tut und läßt sich sowohl auf Listen wie auf Arrays implementieren.

```
selSort :: [Int] -> [Int]
selSort [] = []
selSort ul = let (m,l)=select(ul)
             in m:selSort(l)

select :: [Int] -> (Int,[Int])
select [e]    = (e,[])
select (u:ul) = let (m,l)=select ul
                in if (u<m) then (u,m:l)
                            else (m,u:l)
```

Abbildung 2.1: Sortieren durch Auswahl (`gofer`)

```
SelectionSort(int array[], int len)
{
    for (int i = 0; i < len; i++)
    {
        for (int j = i + 1, int posmin = i; j < len; j++)
            if (array[j] < array[posmin]) posmin = j;
        swap(array[i], array[posmin]);
    }
}

swap(int& a, int& b)
{
    int temp = a; a = b; b = temp;
}
```

Abbildung 2.2: Sortieren durch Auswahl (C)

In Abbildung 2.1 findet man die Implementierung für Listen als `gofer`-Definition und in in Abbildung 2.2 in C-ähnlicher Syntax. Eine Veranschaulichung des letzten Algorithmus ist in Abbildung 2.3 gegeben.

Kommen wir nun zur Analyse des Algorithmus. Bezeichne hierzu $V_{\mathrm{Sel}}(n)$ die Anzahl der benötigten Vergleiche, wenn die Eingabe aus n Datensätzen besteht. Wir wählen also hier (und auch im Rest dieses Abschnittes) das uniforme Kostenmaß.

Zunächst einmal überlegen wir uns, wieviele Vergleiche wir benötigen, um das Minimum aus einer Menge von n Elementen zu bestimmen. Hierfür sind $n-1$ Vergleiche ausreichend, wenn wir wie folgt vorgehen: wähle zunächst irgendein Element als ak-

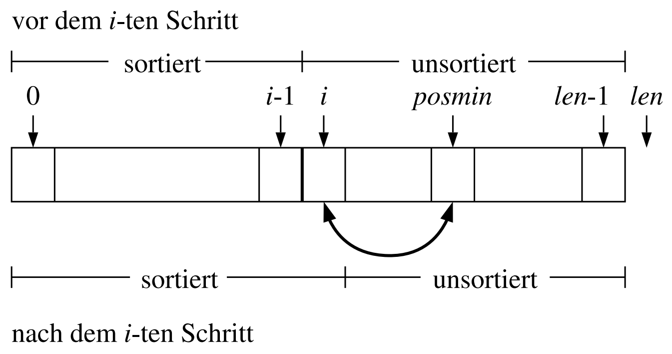

Abbildung 2.3: Schematische Darstellung von Selectionsort

tuelles Element aus der Menge; vergleiche jeweils das aktuelle Element mit einem beliebigen Element und behalte das kleinere. Es sind auch mindestens $n-1$ Vergleiche nötig, sonst hätten wir das gewählte Minimum mit mindestens einem Element nicht verglichen, und wir könnten uns nicht sicher sein, das Minimum gewählt zu haben (⇒ Übungen).

Mit dieser Vorüberlegung und der informalen Beschreibung bzw. den beiden Implementierungen können wir nun eine Rekursionsgleichung für V_{Sel} aufstellen. Für die erste Minimumbestimmung sind $n-1$ Vergleiche nötig. Um die restlichen $n-1$ Elemente mittels Selectionsort zu sortieren sind nun noch $V_{\text{Sel}}(n-1)$ viele Vergleiche nötig.

$$\begin{array}{rcl} V_{\text{Sel}}(1) & = & 0 \\ V_{\text{Sel}}(n) & = & n-1+V_{\text{Sel}}(n-1) \quad \text{für } n>1 \end{array}$$

Diese einfache Rekursionsgleichung können wir leicht durch Iteration lösen:

$$\begin{array}{rcl} V_{\text{Sel}}(n) & = & V_{\text{Sel}}(n-1)+(n-1) \\ & = & V_{\text{Sel}}(n-2)+(n-2)+(n-1) \\ & = & V_{\text{Sel}}(n-i)+\sum\limits_{j=1}^{i}(n-j) \\ & & \text{mit } i=n-1 \text{ erhalten wir} \\ & = & \underbrace{V_{\text{Sel}}(1)}_{=0}+\sum\limits_{j=1}^{n-1}(n-j) \\ & & \text{Umdrehen der Summationsreihenfolge ergibt} \\ & = & \sum\limits_{j=1}^{n-1} j \\ & & \text{Gauß'sche Summe} \\ & = & \dfrac{n(n-1)}{2} \;=\; \dbinom{n}{2} \end{array}$$

Damit haben wir das folgende Theorem bewiesen.

Theorem 2.1 *Selectionsort benötigt zum Sortieren von n Elementen genau $\binom{n}{2}$ Vergleiche.*

2.1.2 Sortieren durch Einfügen

Sehr ähnlich wie das Sortieren durch Auswahl funktioniert das Sortieren durch Einfügen. Wieder stellen wir uns während des Sortiervorganges die Datensätze in zwei Teile aufgeteilt vor: einen sortierten und einen unsortierten Teil. Wieder befinden sich zu Beginn alle Elemente im unsortierten Teil. Nun wählen wir ein beliebiges Element aus dem unsortierten Bereich und fügen dieses in den bereits sortierten Bereich ein. Dabei lassen wir das einzusortierende Element von links nach rechts durch den bereits sortierten Bereich wandern, bis wir die Stelle gefunden haben, wo das Element eingefügt werden kann. In der Listendarstellung können wir dieses Element nun einfach einfügen, bei der Arraydarstellung müssen wir hingegen den hinteren bereits sortierten Teil als Ganzes im Array nach hinten schieben (siehe auch Abbildung 2.4).

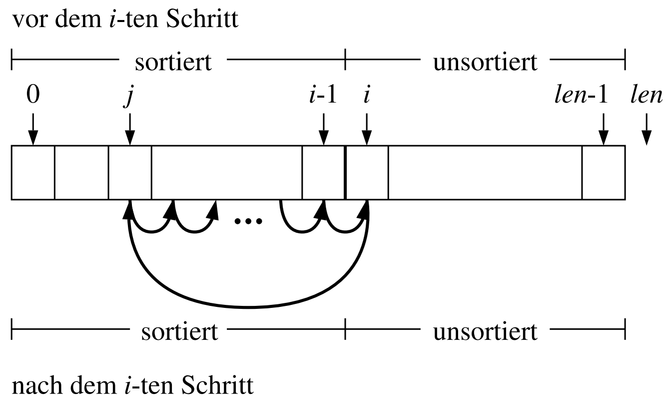

Abbildung 2.4: Schematische Darstellung von Insertionsort

Wiederum lassen sich aus dieser informalen Beschreibung recht rasch Algorithmen gewinnen, die entweder auf Listen oder Arrays arbeiten. In Abbildung 2.5 findet man die Implementierung für Listen als `gofer`-Definition (man erinnere sich auch an die Übungen in Informatik I) und in Abbildung 2.6 in `C`-ähnlicher Syntax. Eine

```
insSort :: [Int]->[Int]
insSort []     = []
insSort (e:us) = insert s (insSort us)

insert :: Int -> [Int] -> [Int]
insert e []     = [e]
insert e (s:sl) | e>s  = s:(insert e sl)
                | e<=s = e:(s:sl)
```

Abbildung 2.5: Sortieren durch Einfügen (`gofer`)

```
InsertionSort(int array[], int len)
{
    for (int i = 1; i < len; i++)
    {
        right_shift(array, LinearSearch(array[], i, array[i]), i);
    }
}

int LinearSearch(int array[], int len, int elt)
{
    for (int j = 0; j < len && array[j] < elt; j++);
    return j;
}

right_shift(int array[], int l, int r)
{
    int k = array[r];
    for (int j = r; j > l; j--) array[j] = array[j - 1];
    array[l] = k;
}
```

Abbildung 2.6: Sortieren durch Einfügen (C)

Veranschaulichung des letzten Algorithmus ist in Abbildung 2.4 gegeben. Hier bei wählen wir der einfacheren Implementierung wegen, immer das erste Element des unsortierten Teilarrays als einzufügendes Element aus.

Für die Analyse des Algorithmus bezeichne $V_{\text{Ins}}(n)$ die Anzahl der benötigten Vergleiche, wenn die Eingabe aus n Datensätzen besteht. Zunächst stellen wir fest, daß $n-1$ Vergleiche ausreichen um ein Element in eine Liste von $n-1$ sortierten Elementen einzufügen. Dies ist diesmal nur eine obere Schranke, unter günstigen Bedingungen reicht ein Vergleich bereits aus.

Mit dieser Vorüberlegung und der informalen Beschreibung bzw. den beiden Implementierungen bekommt man unmittelbar die folgende Rekursionsgleichung:

$$\begin{aligned} V_{\text{Ins}}(1) &= 0 \\ V_{\text{Ins}}(n) &\leq V_{\text{Ins}}(n-1) + (n-1) \quad \text{für } n > 1 \end{aligned}$$

Diese Einfache Rekursionsgleichung können wir leicht durch Iteration lösen:

$$\begin{aligned} V_{\text{Ins}}(n) &\leq V_{\text{Ins}}(n-1) + (n-1) \\ &\leq V_{\text{Ins}}(n-2) + (n-2) + (n-1) \end{aligned}$$

$$
\begin{aligned}
&\leq V_{\text{Ins}}(n-i) + \sum_{j=1}^{i}(n-j) \\
&\quad \text{mit } i = n-1 \text{ erhalten wir} \\
&\leq \underbrace{V_{\text{Ins}}(1)}_{=0} + \sum_{j=1}^{n-1}(n-j) \\
&\quad \text{damit erhalten wir wie oben} \\
&= \frac{n(n-1)}{2} = \binom{n}{2}
\end{aligned}
$$

Damit haben wir das folgende Theorem bewiesen.

Theorem 2.2 *Insertionsort benötigt zum Sortieren von n Elementen maximal $\binom{n}{2}$ Vergleiche.*

2.1.3 Verbessertes Sortieren durch Einfügen (Ü)

Man kann sich beim Einfügen auch etwas geschickter anstellen, indem man die Sortierung der Liste besser ausnutzt. Analog wie beim Telefonbuch kann man durch gezieltes Suchen die Anzahl der Vergleiche deutlich reduzieren. Dies funktioniert allerdings nur, wenn man direkten Zugriff auf die Schlüssel hat, also die Daten in einem Array abgespeichert sind.

Man vergleicht das Element mit dem mittleren Element und je nachdem ob das gegebene Element größer oder kleiner als das mittlere Element gewesen ist, sucht man in der linken oder rechten Hälfte weiter. Dieses Verfahren wird *binäre Suche*

```
BinarySearch(int array[], int len, int elt)
{
    int pos, ℓ = 0, r = len;
    while (ℓ < r)
    {
        pos = (ℓ + r)/2;
        if (elt >= array[pos]) ℓ = pos + 1;
        else r = pos;
    }
    return ℓ;
}
```

Abbildung 2.7: Ein Algorithmus für die 'Binäre Suche' genannt und man kann sich überlegen, daß $\lceil \log(n+1) \rceil$ Vergleiche genügen, um die Einfügeposition zu ermitteln (⇒ Übungen).

Lemma 2.3 *In einem sortierten Array der Länge n benötigt die binäre Suche maximal $\lceil \log(n+1) \rceil$ Vergleiche um die Einfügeposition zu bestimmen.*

Eine Implementierung dieser Strategie findet man in Abbildung 2.7.

Ersetzt man im Algorithmus Insertionsort den Aufruf von `LinearSearch` durch `BinarySearch` (siehe Abbildung 2.6), so erhält man das folgende Theorem:

Theorem 2.4 *Der auf der binären Suche basierende Insertionsort benötigt zum Sortieren von n Elementen maximal $n\lceil \log(n+1) \rceil$ Vergleiche.*

In den Übungen wird auf diese verbesserte Version von Insertionsort und seine Analyse noch näher eingegangen.

Leider sind beim Einfügen eines Elementes in einen sortierten Teilbereich der Länge n eines Arrays im schlimmsten Fall immer noch n Verschiebeoperationen notwendig. Man kann zeigen, das im schlimmsten Fall (und auch im Durchschnitt!) bei Insertionsort immer noch quadratisch viele Verschiebungen (mittels `right_shift`) von Elementen nötig sind. Daher bringt uns die sehr gute Anzahl von Vergleichen bezüglich der Rechenzeit leider nichts ein.

2.2 Mergesort

Eine der wohl ältesten Sortiermethoden ist das Sortieren durch Mischen. Die Idee für diesen Sortieralgorithmus basiert auf der Tatsache, daß es recht einfach ist, zwei sortierte Folgen zu einer sortierten Folge zusammenzumischen. Im folgenden beschreiben wir zunächst die einfacher vorstellbarere rekursive Variante und anschließend die robustere iterative Variante.

2.2.1 Rekursiver Mergesort

Wie wir schon oben bemerkt haben, ist es recht einfach zwei sortierte Folgen zu einer sortierten Folge zusammenzumischen. Was ist aber zu tun, wenn man eine ungeordnete und nicht zwei sortierte Folgen hat? Man teilt die unsortierte Folge in zwei Hälften auf, sortiert diese rekursiv, und mischt die so erhaltenen sortierten Folgen zu einer neuen. Dabei muß man nur berücksichtigen daß sortierte Folgen der

```
MergeSort(int a[] int b[], int l, int r)
{
    if (ℓ == r − 1) return ;
    else
    {
        int m = (ℓ + r)/2;
        MergeSort(a, b, ℓ, m);
        MergeSort(a, b, m, r);
        merge(b, a, ℓ, m, r);
        copy(a, b, ℓ, r);
    }
}

merge(int b[], int a[], int ℓ, int m, int r)
{
    for (int k = ℓ, int ℓ1 = ℓ, int ℓ2 = m; (ℓ1 < m) && (ℓ2 < r); k++)
        if (b[ℓ1] < b[ℓ2]) { a[k] = b[ℓ1]; ℓ1++; }
        else { a[k] = b[ℓ2]; ℓ2++; }
    while (ℓ1 < m) { a[k] = b[ℓ1]; ℓ1++; k++; }
    while (ℓ2 < r) { a[k] = b[ℓ2]; ℓ2++; k++; }
}

copy(int a[], int b[], int ℓ, int r)
{
    for (int i = ℓ; i < r; i++) b[i] = a[i];
}
```

Abbildung 2.8: Rekursiver Mergesort

Länge eins bereits trivialerweise sortiert sind. Eine Illustration eines Ablaufes des rekursiven Mergesorts auf 10 Elementen ist in Abbildung 2.9 gegeben.

Damit erhält man sofort den in Abbildung 2.8 angegebenen Algorithmus. Hier wird das gegebene Array a in das Array b sortiert. Beim Mischen (`merge`) wird angenommen, daß die linke Folge auf den Positionen $[\ell : m-1]$ und die rechte Folge auf den Positionen $[m : r-1]$ des Arrays steht.

Man sollte beachten, daß Insertionsort im Prinzip auch eine Art Mergesort ist. Hier wird eine Folge der Länge n in eine Folge der Länge $n-1$ und eine Folge der Länge 1 aufgeteilt. Wie wir später sehen werden, wird das geschicktere Aufteilen einen effizienteren Algorithmus liefern.

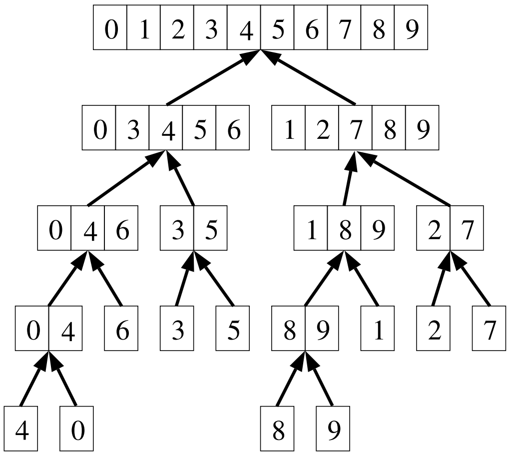

Abbildung 2.9: Ablauf des rekursiven Mergesort auf der Folge (4,0,6,3,5,8,9,1,2,7)

2.2.2 Iterativer Mergesort

Wenn man die Rekursion vermeiden will, kann man das Zusammenmischen auch direkt von unten beginnen. Um auf die Berechnung der Aufteilungen zu verzichten, kann man von links nach rechts immer Folgen, deren Längen Zweierpotenzen sind, zusammenmischen. Nur bei der rechtesten bzw. den beiden rechtesten Folgen müssen deren Längen nicht unbedingt Zweierpotenzen sein. Dies ist in Abbildung 2.10 für dieselbe Folge von 10 Elementen wie für den rekursiven Mergesort illustriert.

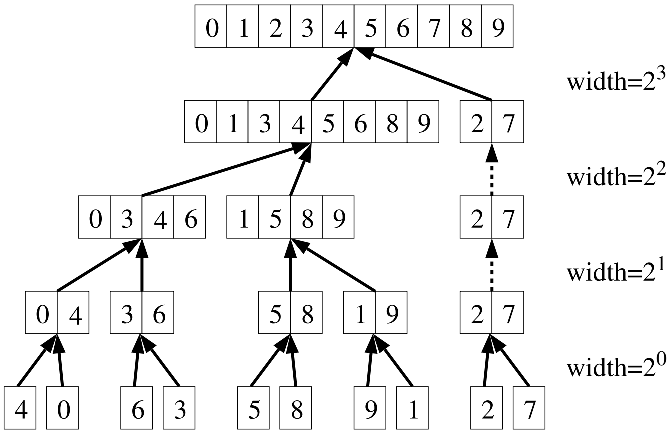

Abbildung 2.10: Ablauf des iterativen Mergesort auf der Folge (4,0,6,3,5,8,9,1,2,7)

```
MergeSort(int a[] int b[], int len)
{
    for (int width = 1; width < len; width∗ = 2)
    {
        for (int i = 0; i < len; i+ = 2 ∗ width)
        {
            int ℓ = i;
            int m = min(len, i + width);
            int r = min(len, i + 2 ∗ width);
            merge(a, b, ℓ, m, r);
        }
        copy(a, b, 0, len);
    }
}
```

Abbildung 2.11: Iterativer Mergesort

Damit erhält man den in Abbildung 2.11 angegebenen Algorithmus für den iterativen Mergesort. Hier bezeichnet *width* immer die Länge der Folgen (bis auf die rechtesten), die zusammengemischt werden. In jeder Iteration wird der Wert von *width* verdoppelt, da sich dann die Länge der bereits sortierten Folgen verdoppelt hat.

2.2.3 Analyse von Mergesort

Für die Analyse des iterativen Mergesort bezeichne $V_{\text{Merge}}(n)$ die Anzahl der benötigten Vergleiche für eine zu sortierende Folge der Länge $n := len$. Zunächst überlegt man sich, daß im iterativen Fall die äußere Schleife (über die Schleifenvariable *width*) genau $\lceil \log(n) \rceil$-mal durchlaufen wird, denn nach genau $\lceil \log(n) \rceil$ iterierten Verdopplungen wird *width* (initialisiert mit 1) größer als $n = len$. Das heißt, *width* nimmt in dieser Schleife Werte aus $\{2^i \,|\, i \in [0 : \lceil \log(n) \rceil - 1]\}$ an. In jedem Schleifendurchlauf sind maximal $\lceil \frac{n}{2 \cdot 2^{width}} \rceil$ Paare von bereits sortierten Folgen der Länge jeweils höchstens 2^{width} vorhanden. Da für das Mischen von zwei Folgen der Länge jeweils höchstens 2^{width} maximal $2 \cdot 2^{width} - 1$ Vergleiche nötig sind, erhalten wir die folgende Abschätzung:

$$
\begin{aligned}
V_{\text{Merge}}(n) &\leq \sum_{w=0}^{\lceil \log(n) \rceil - 1} \left\lceil \frac{n}{2 \cdot 2^w} \right\rceil (2 \cdot 2^w - 1) \\
&\leq \sum_{w=1}^{\lceil \log(n) \rceil} \left\lceil \frac{n}{2^w} \right\rceil (2^w - 1)
\end{aligned}
$$

$$
\begin{aligned}
&\leq \sum_{w=1}^{\lceil\log(n)\rceil}\left(\frac{n}{2^w}+1\right)2^w \\
&\leq \sum_{w=1}^{\lceil\log(n)\rceil}(n+2^w) \\
&\leq n\cdot\lceil\log(n)\rceil + 2^{\lceil\log(n)\rceil+1} \\
&\leq n\cdot\lceil\log(n)\rceil + 2^{\log(n)+2} \\
&\leq n\cdot\lceil\log(n)\rceil + 4\cdot n
\end{aligned}
$$

Theorem 2.5 *Mergesort sortiert ein Array der Länge n mit maximal $n\cdot\lceil\log(n)\rceil + O(n)$ Vergleichen*

Lästig bei Mergesort sind insbesondere die vielen Kopieroperationen. Diese lassen sich durch geschicktere Implementierung jedoch weitestgehend vermeiden (siehe auch ⇒ 2-Phasen–3-Band-Sortieren).

Außerdem lassen sich bei geschickter Implementierung, bereits sortierte Teilsequenzen ausnutzen. In unserer iterativen Version gingen wir ja am Anfang von n sortierten Listen der Länge 1 aus. Wenn man vorher die zu sortierende Folge in bereits sortierte Listen größerer Länge einteilt, kann man diese Vorsortierung beim Mergesort ausnutzen (⇒ Mergesort mit natürlichem Mischen).

Der Hauptnachteil von Mergesort ist jedoch, daß er zum Sortieren eines Arrays der Länge n zwei Arrays der Länge n benötigt. Trotz seiner (wie wir noch sehen werden) geringen Anzahl an Vergleichen, macht ihn das in der Praxis untauglich. Mergesort wird meist nur als externes Sortierverfahren eingesetzt, d.h. wenn man beim Sortieren auf externe Speicher zugreifen darf, wie z.B. große Festplatten oder Bänder. In der Regel sind solche externe Speichermedien reichlich vorhanden, haben aber meist nur einen sehr langsamen Zugriff.

16. Mai 97

2.3 Heapsort

Ein verbessertes Verfahren läßt sich aus der Idee für Selectionsort entwickeln. Dort haben wir bei jeder Minimum-Bildung der restlichen, unsortierten Elemente nie die Ergebnisse aus früheren Vergleichen ausgenützt. Diese Art der Wiederverwertung von Vergleichen werden wir unter anderem bei Heapsort ausnutzen.

Dieses Verfahren geht bereits auf J.W.J. Williams (1964) zurück. Wir werden hier zuerst die platzsparende Implementation von R.W. Floyd (1964) vorstellen. Anschließend werden wir noch neueren Entwicklungen von Heapsort nach Ideen von S. Carlsson (1987) skizzieren.

2.3.1 Standard Heapsort

Zur Beschreibung von Heapsort benötigen wir die Datenstruktur eines Heaps. Ein *Heap* ist eine Datenstruktur, die ganze Zahlen abspeichern kann und die folgenden drei elementaren Operationen auf Heaps zur Verfügung stellt:

int size(heap *h*) gibt die Anzahl der im Heap h gespeicherten Zahlen zurück.

heap create_heap(int *array*[], int *len*) gibt einen Heap zurück, der alle im Array *array* der Länge *len* abgespeicherten Zahlen beinhaltet.

int delete_max(heap *h*) Löscht die maximale Zahl im Heap h und gibt diese zurück.

Man sollte sich bereits an dieser Stelle klar machen, daß man einen Heap nicht nur für ganze Zahlen realisieren kann, sondern für jede total geordnete Menge.

Mit einer Implementierung eines Heaps zusammen mit den obigen Funktionen ergibt sich sofort der in Abbildung 2.12 angegebene Sortieralgorithmus, der allerdings in dieser Form absteigend sortiert.

```
HEAPSORT(INT ARRAY[], INT LEN)
{
    heap h=create_heap(array, len);
    while (size(h)> 0) output delete_max(h);
}
```

Abbildung 2.12: Abstrakte Formulierung von Heapsort

Es bleibt also nun nur noch die Datenstruktur Heap effizient zu implementieren. Wir stellen uns einen Heap als einen fast vollständigen binären Baum vor, in dem die Zahlen in den Knoten abgespeichert sind, und einer Variablen `heap_size`, die die Anzahl der Knoten mitzählt.

Nun müssen wir allerdings erst einmal klären, was wir unter einem fast vollständigen binären Baum verstehen wollen. Ein *Baum* ist rekursiv wie folgt definiert: ein einzelner *Knoten* v, an dem selbst beliebig viele Bäume T_i^v hängen (also auch keiner), ist ein Baum. Wir nennen v die *Wurzel* dieses Baumes. Ein Knoten an dem kein weiterer Baum hängt, nennen wir ein *Blatt* oder auch einen *äußeren Knoten*. Knoten die kein Blatt sind, heißen *innere Knoten*. Der Knoten v ist der *Elter* der Wurzeln der Bäume T_i^v, und die Wurzeln der Bäume T_i^v heißen *Kinder* des Knoten v. Ein Knoten v heißt *Vorgänger* eines Knoten v', wenn v Elter von v' ist, oder

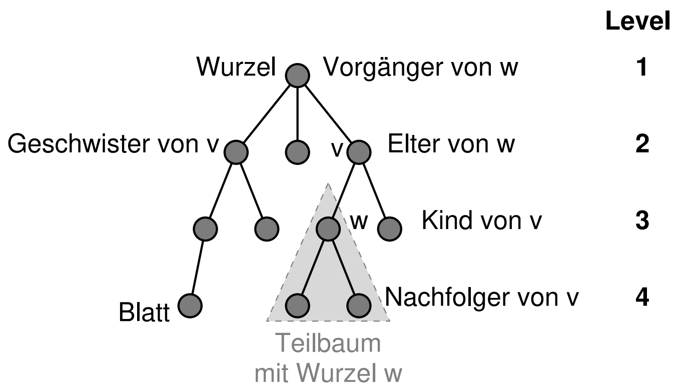

Abbildung 2.13: Bestandteile eines Baumes

ein Kind von v ein Vorgänger von v' ist. Ein Knoten v heißt *Nachfolger* eines Knoten v', wenn v Kind von v' ist, oder der Elter von v ein Nachfolger von v' ist. Zwei Knoten, die denselben Elter haben, bezeichnen wir als *Geschwister*. Als *Teilbaum* eines Baumes T bezeichnen wir einen Baum T_i^v, der während der Konstruktion des Baumes T aufgrund der obigen rekursiven Definition entstanden ist. Der *Level* der Wurzel ist 1. Der Level eines beliebigen Knotens ist der Level seines Elters plus 1. Diese Definitionen kann man sich in Abbildung 2.13 noch einmal veranschaulichen.

Ein Baum heißt *binär*, wenn jeder Knoten des Baumes höchstens zwei Kinder hat. Einen binären Baum nennen wir *fast vollständig*, wenn die folgenden drei Eigenschaften erfüllt sind (vergleiche zur Anschauung mit Abbildung 2.14):

1. Alle bis auf maximal einen inneren Knoten haben zwei Kinder;
2. Alle Knoten mit weniger als zwei Kindern (insbesondere also die Blätter) befinden sich auf den beiden größten Leveln;
3. Die Blätter im größten Level des Baumes sind von links nach rechts aufgefüllt.

Einen fast vollständigen binären Baum bezeichnen wir als *Heap*, wenn er die folgende *Heap-Eigenschaft* erfüllt:

> *Die in einem Knoten abgespeicherte Zahl ist größer als die in seinen Kindern abgespeicherten Zahlen.*

Zur Illustration ist in Abbildung 2.14 ein Heap mit 10 Zahlen angegeben.

Wie können wir nun die für den Heap benötigten Operationen implementieren. Dazu betrachten wir zunächst einmal die rudimentäre Aufgabe, einen Baum in einen Heap

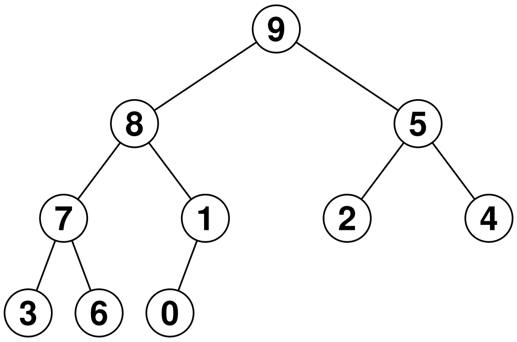

Abbildung 2.14: Ein Heap mit 10 Elementen [0 : 9]

zu verwandeln, wenn in jedem Knoten bis auf die Wurzel die Heap-Eigenschaft erfüllt ist. Dazu lassen wir das Element in der Wurzel langsam durch Vertauschungen in dem Baum nach unten wandern.

Sei also v der einzige Knoten an dem die Heap-Eigenschaft verletzt ist und v_ℓ und v_r seine beiden Kinder (das linke bzw. rechte). Wir vergleichen die in v_ℓ und v_r abgespeicherten Zahlen miteinander und vertauschen die in v gespeicherte Zahl mit der größeren der beiden Zahlen. Dieses Kind, das nun die größere Zahl besitzt, bezeichnen wir mit v^* und das andere mit v'. Nach der Vertauschung ist die Heap-Eigenschaft in v erfüllt, da die größte der drei Zahlen nach oben gewandert ist. Nach Voraussetzung bleibt die Heap-Eigenschaft im Knoten v' erfüllt. Eventuell ist nun aber die Heap-Eigenschaft in v^* verletzt. Diese Prozedur setzen wir nun iterativ fort, bis an jedem Knoten die Heap-Eigenschaft erfüllt ist. Man beachte hierbei, daß an einem Blatt trivialerweise die Heap-Eigenschaft erfüllt ist und damit die Prozedur `reheap` terminiert.

```
REHEAP(HEAP H)
{
    Sei v die Wurzel des Heaps h;
    while (Heap-Eigenschaft in v nicht erfüllt)
    {
        Sei v* das Kind mit dem größeren Schlüssel;
        Vertausche die Schlüssel in v und v*;
        v = v*;
    }
}
```

Abbildung 2.15: Die Prozedur `reheap`

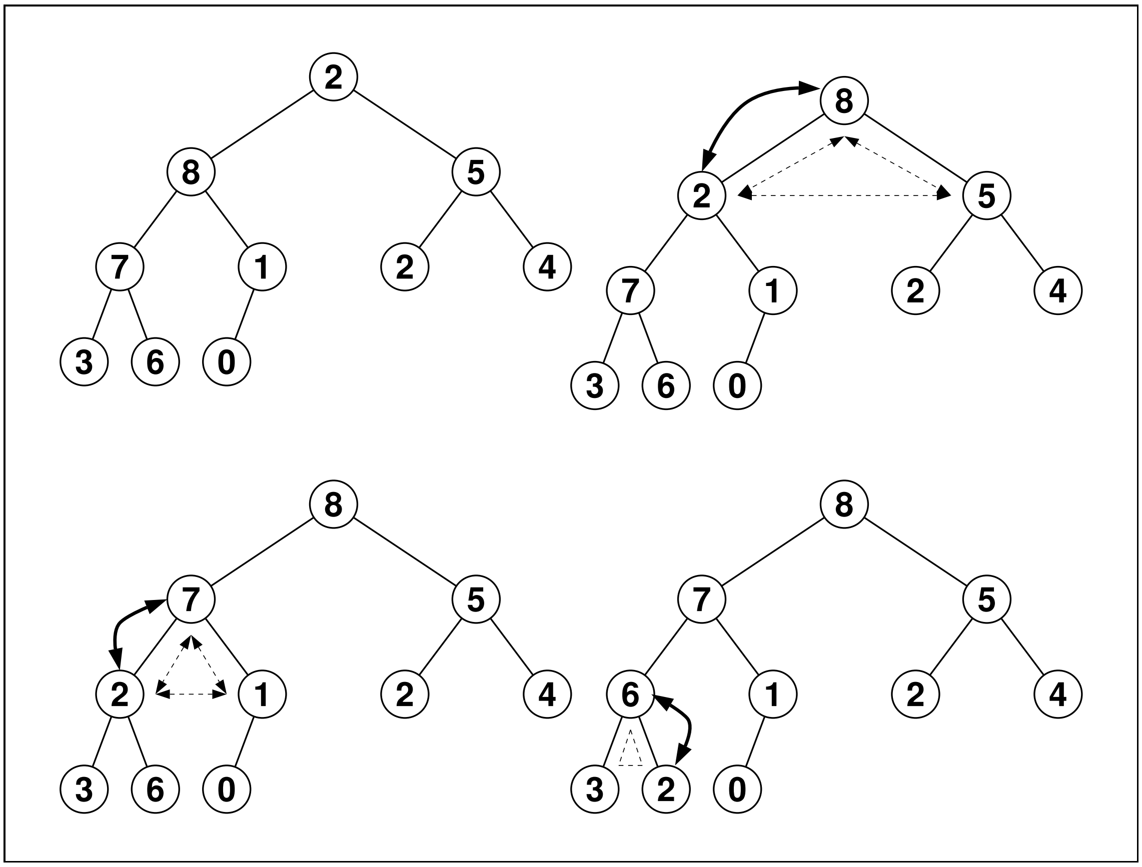

Abbildung 2.16: Durchführung eines `reheap`'s auf einem Heap mit 10 Elementen

Diese Vorgehensweise ist nun noch einmal in Abbildung 2.16 illustriert. Damit ergibt sich der Algorithmus `reheap` wie in Abbildung 2.15 angegeben.

Nun wollen wir mit Hilfe der Prozedur `reheap` die Operationen `size`, `create_heap` und `delete_max` realisieren. Zuerst stellen wir fest, daß die Implementierung der Operation `size` einfach den Wert der Variablen `heap_size` zurückgibt. Mit Hilfe von `reheap` können wir nun ganz einfach die Operationen `delete_max` und `create_heap` realisieren.

Die Operation `delete_max` merkt sich nun einfach den Wert, der in der Wurzel des Heaps steht und löscht den Wert aus der Wurzel. Dieser wird am Ende der Funktionswert von `delete_max` sein. Nun müssen wir nur noch das Loch in der Wurzel füllen. Dazu nehmen wir den Schlüssel des rechtesten Blattes des größten Levels her und bewegen den dort gespeicherten Schlüssel in die Wurzel. Das nun leere Blatt wird einfach gelöscht. Der resultierende Heap erfüllt bis auf die Wurzel die Heap-Eigenschaft. Ein Aufruf von `reheap` stellt dann die Heap-Eigenschaft für den gesamten Baum wieder her.

```
INT DELETE_MAX(HEAP H)
{
    Sei r die Wurzel des Heaps;
    Sei k der in r gespeicherte Schlüssel;
    Sei ℓ das rechteste Blatt im untersten Level;
    Kopiere den Schlüssel in ℓ in die Wurzel r;
    Lösche das Blatt ℓ;
    heap_size--;
    reheap(h);
    return k;
}
```

Abbildung 2.17: Die Operation `delete_max`

In Abbildung 2.18 ist der Aufruf der Operation `delete_max`, also das Löschen des Elementes 9 im Heap, illustriert. Anschließend muß nun noch einmal die Prozedur `reheap` an der Wurzel des verbleibenden Heaps aufgerufen werden. Die abstrakte Formulierung der Operation `delete_max` ist in Abbildung 2.17 angegeben. Auf die konkrete Realisierung dieser Operation kommen wir später noch zurück.

Ebenfalls mit Hilfe der Prozedur `reheap` können wir die Operation `create_heap` realisieren. Hierbei gehen wir levelweise von den Blättern zur Wurzel vor. Nach Definition erfüllt jedes Blatt die Heap-Eigenschaft. Gehen wir nun einen Level im Baum weiter nach oben, so hängen an jedem Knoten dieses Levels zwei Teilbäume für die die Heap-Eigenschaft erfüllt ist. Das bedeutet, daß jetzt nur noch an den Knoten des betrachteten Levels die Heap-Eigenschaft verletzt sein kann. Ein Aufruf von `reheap`

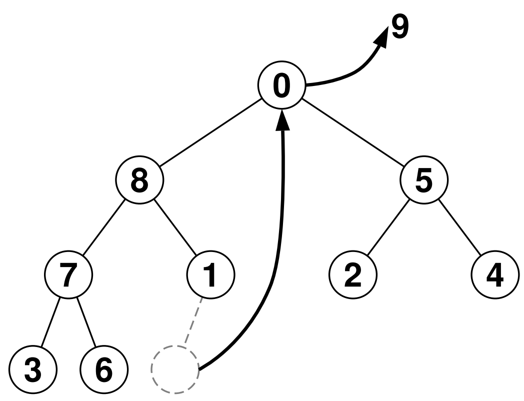

Abbildung 2.18: Durchführung der ersten Teiloperation von `delete_max`

```
HEAP CREATE_HEAP(INT ARRAY[], INT LEN)
{
    heap_size = len;
    Konstruiere Baum H mit len Knoten;
    Fülle jeden Knoten mit einem Schlüssel des Arrays array;
    for (ℓ =Höhe des Baumes H; ℓ ≥ 1; ℓ−−)
        for each Knoten v auf Level ℓ
            reheap(Baum H mit Wurzel v);
    return H;
}
```

Abbildung 2.19: Die Operation `create_heap`

an diesen Knoten stellt dann die Heap-Eigenschaft für diese Teilbäume sicher. Der Aufbau eines Heaps mittels dieser Strategie ist in Abbildung 2.20 dargestellt. Damit erhalten wir dann die abstrakte Formulierung der Operation `create_heap`, wie in Abbildung 2.19 angegeben. Auch hier werden wir auf eine konkrete Implementierung

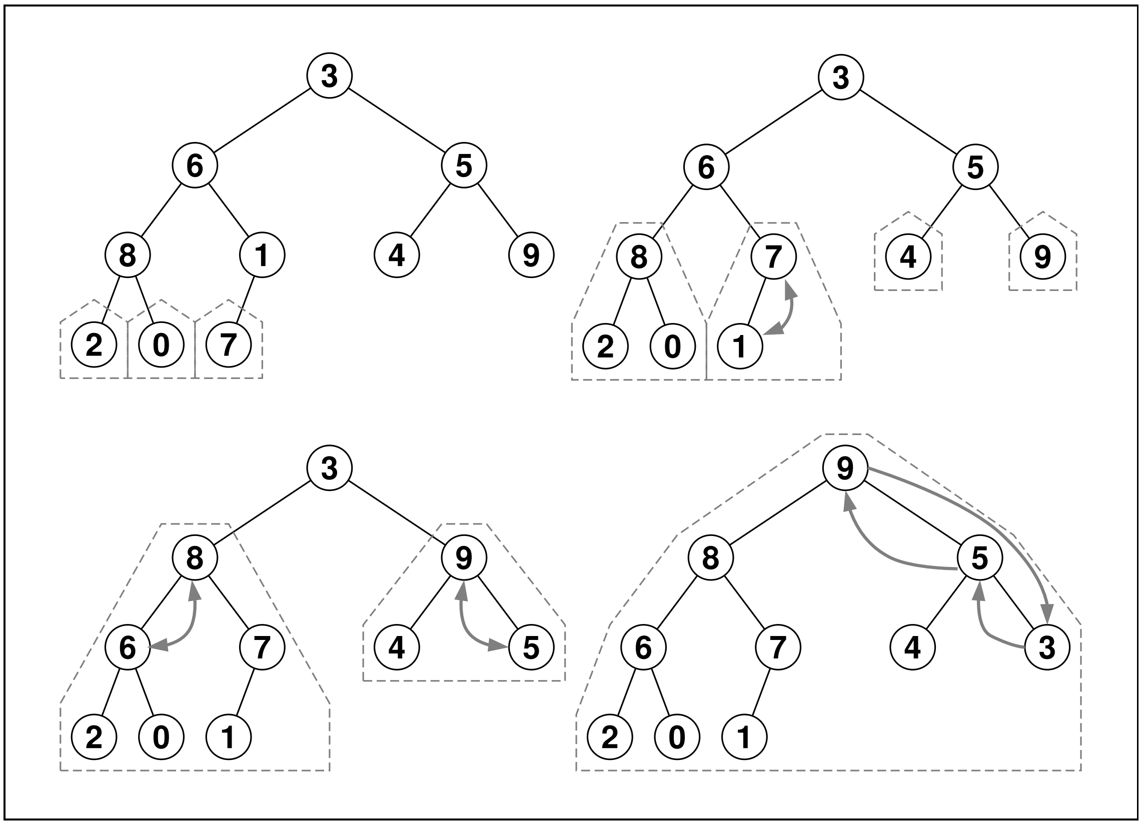

Abbildung 2.20: Aufbau eines Heaps mit 10 Elementen

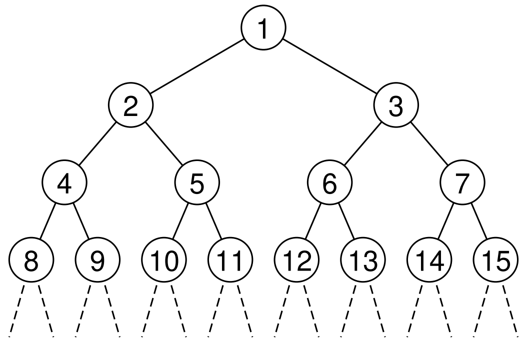

Abbildung 2.21: Ein fast vollständiger binärer Baum aufgefaßt als Array

dieser Operation später noch zurückkommen.

Überraschenderweise können wir einen fast vollständigen binären Baum als Array implementieren. Wir numerieren den Baum einfach von oben nach unten und in jedem Level von links nach rechts wie in Abbildung 2.21 durch. Dabei stellt man fest daß für einen Knoten mit Nummer i folgendes gilt:

- Sein linkes Kind hat Nummer $2i$;
- Sein rechtes Kind hat Nummer $2i + 1$;
- Sein Elter hat Nummer $\lfloor \frac{i}{2} \rfloor$.

Damit erhalten wir eine schöne Realisierung eines fast vollständigen binären Baumes der Größe n mit Hilfe eines Arrays der Länge n. Eine kurze Inspektion der Numerierung des Baumes ergibt zusätzlich, daß auf dem Level ℓ gerade die Zahlen stehen deren Binärdarstellung Länge ℓ hat. Also finden wir den zu einer Zahl i gehörigen Knoten auf Level $\ell(i) = \lfloor \log(i) \rfloor + 1$.

Es bleibt nur noch ein kleines Problem. Da wir bis jetzt alle Algorithmen in `C`-ähnlicher Notation angegeben haben und in `C` auch alle Arrays der Länge n mit $[0 : n - 1]$ indiziert sind, müssen wir noch eine Indexverschiebung mit Hilfe des folgenden kommutativen Diagrammes vornehmen:

$$\begin{array}{llll}
a[1 \ldots n] & i & \rightarrow & (2i, 2i+1) \\
 & \uparrow +1 & & \downarrow -1 \\
a[0 \ldots n-1] & j & \rightarrow & (2(j+1) - 1, (2(j+1) + 1) - 1) = (2j+1, 2j+2)
\end{array}$$

In der oberen Zeile steht die Zuordnung Knoten zu seinen Kindern und in der Zeile darunter die durch die Indexverschiebung modifizierte Zuordnung für Arrays mit Indizes aus $[0 : n - 1]$.

```
HEAPSORT(INT ARRAY[], INT LEN)
{
    /* create_heap */
    for (int i = len − 1; i ≥ 0; i−−)
        reheap(array, len, i);
    for (int ℓ = len − 1; ℓ ≥ 1; ℓ−−)
    {
        /* delete_max */
        swap(array[0], array[ℓ]);
        reheap(array, ℓ, 0);
    }
}

reheap(int array[], int len, int r)
{
    int i = r;
    int j = 2r + 1; /* j is always a child of i */
    while (j < len)
    {
        /* determine larger child */
        if ((j + 1 < len) && (array[j + 1] > array[j])) j++;
        /* Is heap-property fulfilled? */
        if (array[j] > array[i])
            {swap(array[i], array[j]); i = j; j = 2 ∗ j + 1;}
        else
            return ;
    }
}
```

Abbildung 2.22: Die Prozedur Heapsort

Also erhalten wir damit die folgende Arraydarstellung eines fast vollständigen binären Baumes mit n Knoten: Für einen Knoten mit Nummer i gilt, daß

- sein linkes Kind die Nummer $2i + 1$ hat,
- sein rechtes Kind die Nummer $2i + 2$ hat,
- sein Elter die Nummer $\lfloor \frac{i+1}{2} \rfloor - 1 = \lfloor \frac{i-1}{2} \rfloor$ hat.

Wir merken an dieser Stelle noch an, daß nach der Indexverschiebung ein Knoten mit der Nummer $i \in [0 : n-1]$ im Baum auf Level $\lfloor \log(i + 1) \rfloor + 1$ liegt. Diese Tatsache werden wir später noch für die Analyse von Heapsort benötigen.

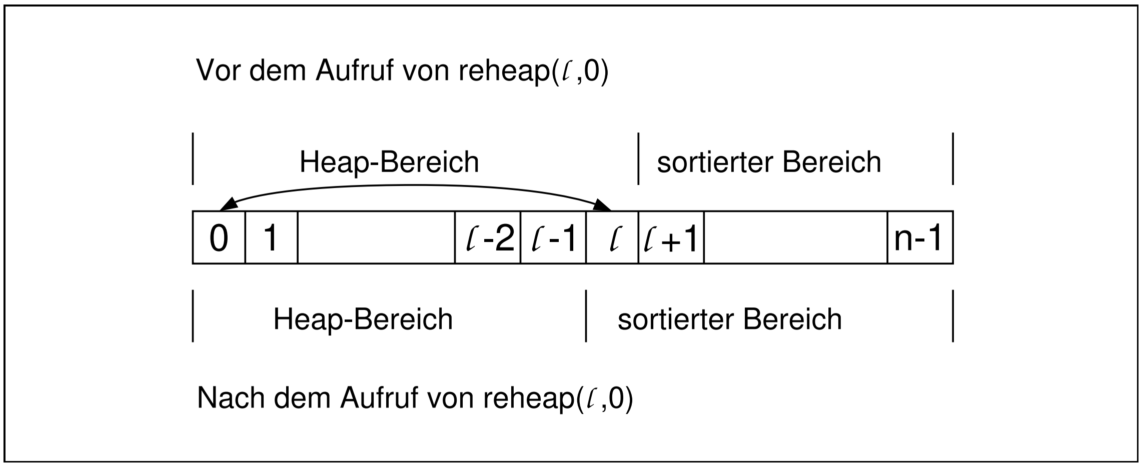

Abbildung 2.23: Illustration von `delete_max` während des Heapsortes

Damit erhalten wir nun die kompakte Formulierung von Heapsort auf einem Array, die in Abbildung 2.22 gegeben ist. In Abbildung 2.23 ist die Funktionsweise von `delete_max` während des Heapsorts noch einmal veranschaulicht. Man beachte, daß im vorderen Bereich des Arrays der Heap realisiert wird, während im hinteren Bereich des Arrays die aufsteigend sortierte Folge aufgebaut wird.

23. Mai 97

2.3.2 Analyse von Standard-Heapsort

Für die Analyse von Heapsort untersuchen wir zuerst die Prozedur `reheap`, auf die sich Heapsort ja im wesentlichen abstützt. Bezeichne hierzu $V_{\text{reheap}}(n, i)$ die Anzahl von Vergleichen, die ein `reheap` auf einem Teilbaum mit Wurzel i und Knoten mit Nummern kleiner als n benötigt.

Wie wir uns schon vorher überlegt haben, befindet sich der Knoten mit Nummer i auf dem Level $\lfloor \log(i+1) \rfloor + 1$. Die Prozedur steigt schlimmstenfalls bis zu einem Blatt herunter, das sich höchstens auf dem Level $\lfloor \log(n) \rfloor + 1$ befinden kann. Dies wird in Abbildung 2.24 veranschaulicht. Bei jedem Wechsel eines Levels werden zwei Vergleiche ausgeführt: einer zum Bestimmen des Kindes mit dem maximalen Schlüssel und einer zum Überprüfen der Heap-Eigenschaft. Somit erhalten wir:

$$V_{\text{reheap}}(n, i) \;=\; 2\left(\lfloor \log(n) \rfloor - \lfloor \log(i+1) \rfloor\right).$$

Berechnen wir nun die Anzahl der Vergleiche, die zum Aufbau des Heaps nötig sind. Bezeichne $V_{\text{create}}(n)$ die Anzahl der Vergleiche, die nötig sind einen Heap mit n Schlüsseln aufzubauen. Die Schleifenvariable i zum Aufbau des Heaps läuft über die Werte aus $[0 : n-1]$. Somit erhalten wir die folgende Abschätzung

$$V_{\text{create}}(n) \;\leq\; \sum_{i=0}^{n-1} V_{\text{reheap}}(n, i)$$

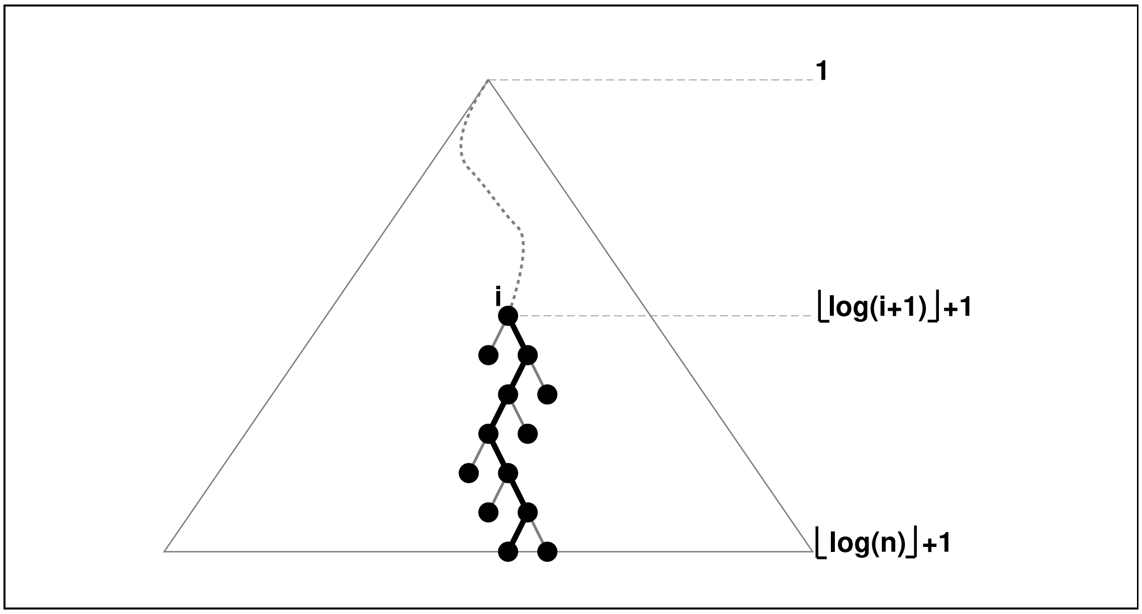

Abbildung 2.24: Ausgeführte Vergleiche bei einem `reheap`(i, n)

$$
\begin{aligned}
&\leq \sum_{i=0}^{n-1} 2\left(\lfloor\log(n)\rfloor - \lfloor\log(i+1)\rfloor\right) \\
&\leq \sum_{i=1}^{n} 2\left(\lfloor\log(n)\rfloor - \lfloor\log(i)\rfloor\right) \\
&\leq 2\sum_{i=1}^{n} \left(\log(n) - \log(i) + 1\right) \\
&\leq 2n\log(n) + 2n - 2\sum_{i=1}^{n}\log(i) \\
&\ \text{Den folgenden Schritt erläutern wir weiter unten} \\
&\leq 2n\log(n) + 2n - 2\int_1^n \frac{\ln(x)}{\ln(2)}dx \\
&\leq 2n\log(n) + 2n - \frac{2}{\ln(2)}\left[x\ln(x) - x\right]_1^n \\
&\leq 2n\log(n) + 2n - \frac{2}{\ln(2)}\left(n\ln(n) - n - 0 + 1\right) \\
&\leq 2n\log(n) + 2n - 2n\log(n) + \frac{2}{\ln(2)}\left(n - 1\right) \\
&\leq 5n
\end{aligned}
$$

In der obigen Abschätzung haben wir eine Summe durch ein Integral abgeschätzt. Diese Abschätzung folgt aus der Tatsache, daß Obersummen immer größer gleich

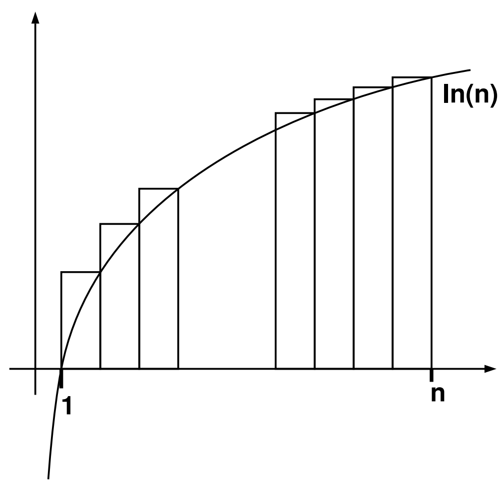

Abbildung 2.25: Abschätzung einer Summe mittels eines Integrals

dem Wert des Integrals sind. Dies wird auch in Abbildung 2.25 deutlich. Hier bezeichnen die Balken gerade die Obersumme des Integrals. Auf der anderen Seite entspricht die Summe die wir abschätzen wollen gerade dem Flächeninhalt dieser Balken. Also können wir die Summe durch das Integral nach unten abschätzen.

Wir können an dieser Stelle festhalten, daß zum Aufbau des Heaps nur linear viel Vergleiche nötig sind. Wir wollen hier noch anmerken, daß man mit mehr Mühe bei dieser Analyse, die Konstante auf unter $2.5n$ drücken kann, in dem man beobachtet, daß die Prozedur `reheap` nur für $i < \lfloor \frac{n}{2} \rfloor$ wirklich etwas tut.

Es bleibt nun noch die Zahl der Vergleiche für den eigentlichen Sortiervorgang abzuschätzen, die wir mit $V_{\text{sort}}(n)$ bezeichnen wollen. Die Schleifenvariable ℓ läuft hierbei über den Wertebereich $[1 : n-1]$. Damit erhalten wir:

$$\begin{aligned}
V_{\text{sort}}(n) &\leq \sum_{\ell=1}^{n-1} V_{\text{reheap}}(\ell, 0) \\
&\leq \sum_{\ell=1}^{n-1} 2\left(\lfloor \log(\ell+1) \rfloor - \lfloor \log(1) \rfloor\right) \\
&\leq 2\sum_{\ell=2}^{n} \lfloor \log(\ell) \rfloor \\
&\leq 2\sum_{\ell=2}^{n} \log(n) \\
&\leq 2n\log(n)
\end{aligned}$$

Wir können nun unsere Ergebnisse in dem folgenden Satz zusammenfassen.

Theorem 2.6 *Heapsort sortiert ein Array der Länge n mit $2n\log(n) + 5n$ vielen Vergleichen und nur konstant vielem zusätzlichen Speicherplatz.*

An dieser Stelle sei nochmal darauf hingewiesen daß Heapsort im wesentlichen mit dem Speicherplatz der Datensätze auskommt. Es werden nur vier zusätzliche Variable benötigt. Es handelt sich hierbei um ein echtes (sogenanntes) in-situ-Verfahren, da es so gut wie keinen zusätzlichen Speicherplatz erfordert. Heapsort ist damit das einzige Sortierverfahren neben Selectionsort und Insertionsort, das wir als in-situ-Verfahren kennenlernen werden

2.3.3 Carlsson's Variante von Heapsort

Der Hauptnachteil von Heapsort in der Praxis ist die 'große' Anzahl an Vergleichen. Im Vergleich zu Mergesort stört hier der Faktor 2 vor dem Term $n\log(n)$. Im Jahre 1987 hat S. Carlsson eine verblüffend einfache Idee gefunden mit dem man diesen Faktor 2 entfernen kann. Zumindest asymptotisch gehört diese modifizierte Version von Heapsort damit zu den Sortierverfahren mit den wenigsten Vergleichen.

Betrachten wir noch einmal die Prozedur `reheap`, die ja diese Konstante 2 einführt. Das Problem ist, daß wir beim Absteigen durch den Heap auf jedem Level zwei Vergleiche machen. Carlsson schlug vor, erst einmal einen Vergleich einzusparen. Aber welchen? Wir werden beim Absteigen durch den Heap nicht mehr überprüfen, ob an einem Knoten die Heap-Eigenschaft verletzt ist und werden eine Verletzung auch nicht beheben. Wir wandern also nur einen Pfad im Heap nach unten, wobei wir immer in Richtung des Kindes mit dem größeren Schlüssel verzweigen. Diesen Pfad nimmt auch Standard-Heapsort, nur sortiert er dabei bereits das Element in der Wurzel ein. Außerden durchläuft Standard-Heapsort den Pfad eventuell nicht bis zum Ende, sondern bricht an der Stelle ab, an der die Heap-Eigenschaft nach einer Vertauschung erfüllt ist.

Betrachten wir nun diesen Pfad. Die Schlüssel, die wir auf diesem Pfad beim Abwärtsgehen gefunden haben, bilden auf Grund der Heap-Eigenschaft eine absteigend sortierte Folge (mit Ausnahme des Schlüssels in der Wurzel).

Anstatt nun wie beim Standard-Heapsort die Stelle schrittweise von oben nach unten zu finden, an der der in der Wurzel gespeicherte Schlüssel einzusortieren ist, bedienen wir uns wieder der binären Suche. Mit Hilfe der Tatsache, daß der i-te Vorgänger eines Knotens mit Nummer m die Nummer $\lfloor (m+1)/2^i \rfloor - 1$ hat, können wir die binäre Suche leicht durchführen. Davor müssen wir nur die Schlüssel oberhalb der Stelle an die die Wurzel eingeführt werden soll, um eine Stelle nach oben verschieben werden. Im Detail siehe die modifizierte Prozedur `reheap` in Abbildung 2.26.

```
CarlssonReheap(int array[], int len, int i)
{
    /* Find a leaf (and hence a path) */
    int depth = 0, j = 0, k = 2 * k + 1;
    /* j is always a parent of k */
    /* depth = Number of vertices from the root j minus 1 *
    while (k + 1 < len)
    {
        if (array[k + 1] > array[k]) k++;
        j = k; k = 2 * k + 1; depth++;
    }
    /* is k a lonely child of j? */
    if (k < len) { j = k; depth++; }

    /* binary search on the path from the root to j */
    int pos, top = depth, bot = 0;
    while (top > bot)
    {
        pos = (top + bot)/2;
        /* ((j + 1) >> pos) - 1 = ⌊(j + 1)/2^pos⌋ - 1 */
        if (array[i] >= array[((j + 1) >> pos) - 1]) bot = pos + 1;
        else top = pos;
    }
    /* Thus, ⌊(j + 1)/2^bot⌋ - 1 is insertion point */
    j = ((j + 1) >> bot) - 1;

    /* Moving keys upward */
    int hlp = array[j];
    array[j] = array[i];
    while (j > i)
    {
        j = (j + 1)/2 - 1;
        swap(array[j], hlp);
    }
}
```

Abbildung 2.26: Modifizierter `reheap` nach Carlsson

Theorem 2.7 *Die von Carlsson modifizierte Variante von Heapsort sortiert ein Array der Länge n mit $n\log(n) + n\,\mathrm{loglog}(n) + O(n)$ vielen Vergleichen und nur konstant vielem zusätzlichen Speicherplatz.*

27. Mai 97

```
BottomUpReheap(int array[], int len, int i)
{
    /* find a Leaf (and hence a path) */
    int j = 1, k = 2 * k + 1;
    /* j is always the parent of k */
    while (k + 1 < len)
    {
        if (array[k + 1] > array[k]) k++;
        j = k; k = 2 * k + 1;
    }
    /* is k a lonely child of j? */
    if (k < len) j = k;

    /* searching upward from j */
    while (array[i] > array[j]) j = (j - 1)/2;

    /* moving keys upward */
    int hlp = array[j];
    array[j] = array[i];
    while (j > i)
    {
        j = (j - 1)/2;
        swap(array[j], hlp);
    }
}
```

Abbildung 2.27: Die Prozedur `bottom_up_reheap`

2.3.4 Bottom-Up-Heapsort

Leider ist in der Praxis der Term $n \log\log(n)$ noch etwas hinderlich. Um diese Anzahl von Vergleichen noch einsparen zu können, überlegt man sich die folgende Heuristik. Der Schlüssel der beim `reheap` in die Wurzel gesetzt wird und dann wieder nach unten wandert, stammte ja von einem Blatt des Heaps. Deswegen könnte man vermuten, daß es sich wohl um ein sehr kleines Element handelt, das nun auf dem von `reheap` konstruierten Pfad ziemlich weit unten eingefügt werden muß. Daraus resultiert nun die folgende Heuristik, daß man beim Bestimmen der richtigen Position auf diesem `reheap`-Pfad nicht linear von oben, wie beim Standard-Heapsort, sondern linear von unten sucht. Die Praxis beweist, daß im Mittel nur sehr wenige Vergleiche nötig sind, um bei dieser Strategie den Einfügepunkt zu bestimmen. Die für den sogenannten *Bottom-Up-Heapsort* erforderliche Prozedur `bottom_up_reheap` ist in Abbildung 2.27 angegeben.

Eine exakte Analyse ist leider schwierig. Man kann zeigen, daß für diese Variante maximal $\frac{3}{2}n\log(n) + O(n)$ viele Vergleiche zum Sortieren ausreichend sind.

Theorem 2.8 *Bottom-Up-Heapsort sortiert ein Array der Länge n mit $\frac{3}{2}n\log(n) + O(n)$ vielen Vergleichen und nur konstant vielem zusätzlichen Speicherplatz.*

Für die mittlere Anzahl an Vergleichen ist die Analyse mathematisch sehr schwierig, da ein Heap nach einigen `reheap`'s nicht mehr zufällig aufgebaut ist. Man vermutet jedoch, daß im Mittel nur $n\log(n) + O(n)$ viele Vergleiche nötig sind, wobei die versteckte Konstante im O eher klein ist (kleiner als $\frac{1}{2}$). Diese Vermutung wird durch zahlreiche Experimente erhärtet.

Vermutung 2.9 *Bottom-Up-Heapsort sortiert ein Array der Länge n im Mittel mit $n\log(n) + O(n)$ vielen Vergleichen und nur konstant vielem zusätzlichen Speicherplatz.*

2.4 Quicksort

Wir kommen nun zu dem in Praxis am häufigsten angewandten Sortieralgorithmus: *Quicksort*. Quicksort ist ebenfalls ein ziemlich altes Sortierverfahren, er wurde bereits 1962 von C.A.R. Hoare entwickelt.

2.4.1 Allgemeines Verfahren

Das Grundprinzip ist vom Wesen her dem von Mergesort gleich. Wir haben eine Folge von Daten und wissen nicht, wie wir sie sortieren sollen. Also teilen wir die Daten in zwei Hälften und sortieren diese rekursiv. Anders als beim Mergesort stellen wir uns nun beim Aufteilen etwas geschickter an. Wir wählen uns einen beliebigen Schlüssel aus der Folge und teilen die Datensätze in zwei Hälften: eine Hälfte mit Schlüsseln, die größer sind als der gewählte, und eine Hälfte, die kleiner ist als der gewählte Schlüssel. Den Schlüssel bezüglich dessen wir die Folge aufteilen, nennen wir *Pivot-Element*. Wir nehmen hier der Einfachheit halber an, daß jeder Schlüssel nur einmal auftritt. Die Übertragung auf den allgemeinen Fall ist nicht weiter schwierig. Nun sortieren wir die beiden Hälften rekursiv, und hängen die sortierten Teilfolgen einfach hintereinander. Dieses Vorgehen ist in Abbildung 2.28 illustriert, wobei wir der Einfachheit halber als Pivot-Element immer den rechtesten Schlüssel in der Teilfolge auswählen. Der detaillierte Algorithmus ist in Abbildung 2.29 angegeben.

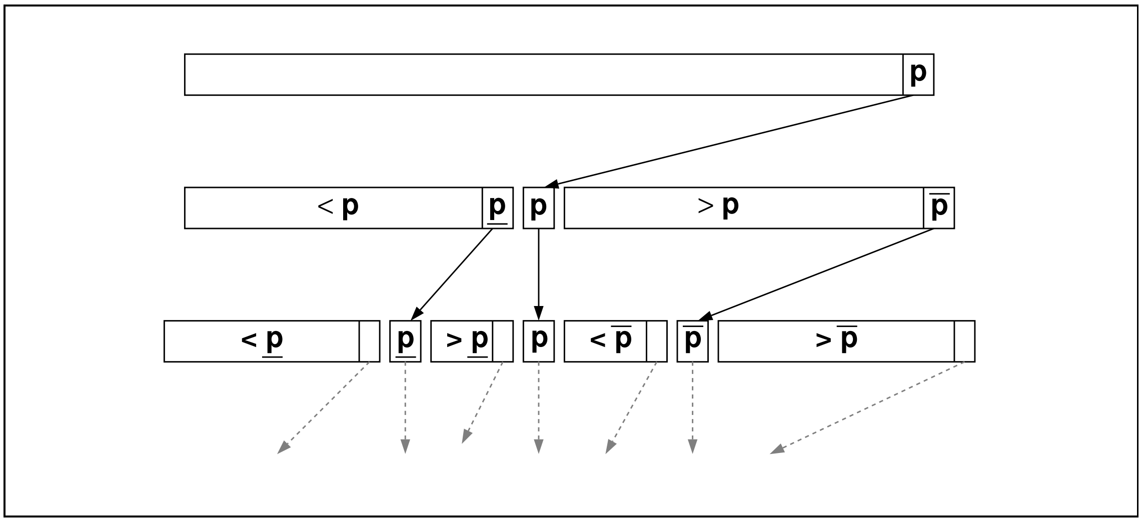

Abbildung 2.28: Aufteilungsschritt bei Quicksort

Im folgenden nehmen wir an, daß wir einen Teilbereich des Arrays *array* von der Position ℓ bis zur Position r (einschließlich) sortieren wollen. Beim Aufteilungsschritt (`partition`) gehen wir wie folgt vor. Wir vertauschen das rechteste Element mit dem Pivot-Element, damit während des Aufteilens das Pivot-Element immer am rechten Rand des Arrays steht. Wir suchen mit einem Zeiger i von links einen Schlüssel, der größer oder gleich dem Pivot-Element ist, und mit einem zweiten Zeiger j von rechts einen Schlüssel, der kleiner oder gleich dem Pivot-Element ist. Haben wir nun so ein Paar gefunden, vertauschen wir die beiden Schlüssel und machen weiter. Wird nun $i \geq j$, dann brechen wir ab, denn wir wissen nun, daß links von dieser Position nur kleinere und rechts nur größere Elemente stehen. Außerdem kann man sich überlegen, daß bei unserer Implementation der Schlüssel an Position i größer oder gleich dem Pivot-Element sein muß. Also ist die letzte Vertauschung des Pivot-Elementes mit dem Schlüssel an Position i zulässig. Nun ist i die Pivot-Position und links davon stehen nur Schlüssel, die kleiner oder gleich sind, und rechts davon stehen nur Schlüssel, die größer oder gleich sind. Damit können wir nun rekursiv `quicksort`$(\ell, i-1)$ und `quicksort`$(i+1, r)$ aufrufen. An dieser Stelle sollten wir die Rekursionen in der richtigen Reihenfolge aufrufen. Es ist günstiger, zuerst den kleineren Teilbereich zu sortieren und dann erst den größeren. Wir kommen später darauf zurück, warum wir hier diese Entscheidung treffen.

2.4.2 Worst-Case und Best-Case Analyse von Quicksort

Kommen wir nun zur Analyse von Quicksort. Für die Analyse gehen wir im folgenden davon aus, daß wir eine Folge von n unterschiedlichen Zahlen sortieren, d.h. keine Zahl taucht mehrfach in der Folge auf.

```
QUICKSORT(INT ARRAY[], INT L, INT R)
{
    int pivot = r;
    if (ℓ < r)
    {
        int pivot_pos =partition(array, ℓ, r, pivot);
        if (pivot_pos − ℓ < r − pivot_pos)
        {
            quicksort(array, ℓ, pivot_pos − 1);
            quicksort(array, pivot_pos + 1, r);
        }
        else
        {
            quicksort(array, pivot_pos + 1, r);
            quicksort(array, ℓ, pivot_pos − 1);
        }
    }
}

int partition(int array[], int ℓ, int r, int pivot)
{
    int i = ℓ − 1, j = r;
    swap(array[pivot], array[r]);
    pivot = r;
    for (;;)
    {
        do i++; while ((i < j) && (array[i] < array[pivot]));
        do j−−; while ((j > i) && (array[j] > array[pivot]));
        if (i >= j) break;
        swap(array[i], array[j]);
    }
    swap(array[i], array[r]);
    return i;
}
```

Abbildung 2.29: Quicksort

Zunächst ein paar Definitionen, die im folgenden die Sprechweise vereinfachen.

Definition 2.10 *Ein Element einer Folge von Elementen einer total geordneten Menge hat* Rang k*, wenn genau* $k-1$ *Elemente der Folge kleiner sind.*

Definition 2.11 *Ein Element m einer Folge von Elementen einer total geordneten Menge heißt* Median, *wenn m den Rang $\lceil \frac{n+1}{2} \rceil$ hat.*

Bezeichne hierzu wieder $V_{\text{Quick}}(n)$ die maximale Anzahl der benötigten Vergleiche, um ein Array der Länge n zu sortieren. Da der Aufteilungsschritt offensichtlich $n-1$ Vergleiche benötigt, erhalten wir folgende Rekursionsgleichung, wenn wir als Pivot-Element das k-kleinste erwischen:

$$\begin{aligned} V_{\text{Quick}}(n) &= 0 \quad \text{für } n < 2 \\ V_{\text{Quick}}(n) &= (n-1) + V_{\text{Quick}}(k-1) + V_{\text{Quick}}(n-k) \quad \text{für ein } k \in [1:n] \;\wedge\; n > 1 \end{aligned}$$

Leider hängen die Größen der rekursiv zu durchsuchenden Teilbereiche von dem gewählten Pivot-Element ab. In der obigen Rekursionsgleichung haben wir angenommen, daß wir den k-kleinsten Schlüssel als Pivot gewählt haben. Dummerweise könnten wir immer den größten Schlüssel des Teilbereichs wählen (insbesondere wenn die Folge bereits sortiert ist!), und würden damit die folgende Rekursionsgleichung erhalten:

$$\begin{aligned} V_{\text{Quick}}(n) &= 0 \qquad \text{für } n < 2 \\ V_{\text{Quick}}(n) &= (n-1) + V_{\text{Quick}}(n-1) \qquad \text{für } n > 1 \end{aligned}$$

Diese Rekursionsgleichung haben wir aber bereits für Selectionsort gelöst. Also kann sich Quicksort im schlimmsten Falle wie Insertionsort verhalten, also $\binom{n}{2}$ Vergleiche benötigen. Es scheint so als hätten wir einen sehr schlechten Sortieralgorithmus gefunden.

Theorem 2.12 *Quicksort benötigt zum Sortieren eines Arrays der Länge n maximal $\binom{n}{2}$ viele Vergleiche.*

Diese Möglichkeit des ungleichen Aufteilens des Arrays ist auch der Grund, warum wir bei der Rekursion zuerst den kleineren Teilbereich sortieren. Somit bleiben die Informationen über die noch zu sortierenden Teilbereiche durch $O(\log(n))$ beschränkt. Anderenfalls könnten diese Informationen im schlimmsten Falle bis auf $\Theta(n)$ anwachsen und damit sehr schnell zu Speicherplatzproblemen führen. Durch diese Auswahl haben wir nun auch den zusätzlichen Speicherplatz bei Quicksort auf $O(\log(n))$ beschränkt. Wir wollen an dieser Stelle noch einmal ausdrücklich darauf hinweisen, daß Quicksort deshalb *kein* in-situ-Verfahren ist.

Andererseits können wir mit der Pivot-Wahl Glück haben und immer den Median erwischen. Dann sind in der linken Hälfte genau $\lceil \frac{n+1}{2} \rceil - 1$ Schlüssel und $\lfloor \frac{n+1}{2} \rfloor$

Schlüssel in der rechten Hälfte. Damit erhalten wir eine ähnliche Rekursionsgleichung wie bei Mergesort.

$$\begin{aligned} V_{\text{Quick}}(n) &= 0 \qquad \text{für } n < 2 \\ V_{\text{Quick}}(n) &\leq (n-1) + V_{\text{Quick}}\left(\left\lceil \frac{n+1}{2} \right\rceil - 1\right) + V_{\text{Quick}}\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right) \qquad \text{für } n > 1 \end{aligned}$$

Damit wissen wir daß Quicksort unter günstigen Umständen nur $n \log(n) + O(n)$ viele Vergleiche benötigt.

30. Mai 97

2.4.3 Average-Case Analyse von Quicksort (+)

Da wir nun gesehen haben, daß sich Quicksort sehr datenabhängig verhält, ist es hier ratsam die mittlere Anzahl von Vergleichen zu untersuchen. Dazu müssen wir uns erst einmal eine vernünftige Wahrscheinlichkeitsverteilung auf dem Raum der Eingaben überlegen. Die erste naheliegende Idee ist, alle Folgen der Länge n von ganzen Zahlen als gleichwahrscheinlich zu betrachten. Das wäre allerdings mathematisch aufwendig und außerdem nicht ganz korrekt. Unsere Algorithmen betrachten ja nicht die Zahlen an sich, sondern sie verwenden nur die Ergebnisse der Vergleichsfunktion einer total geordneten Menge auf n Elementen. Also sollten wir der Analyse zu Grunde legen, daß alle Permutationen auf n Elementen gleichwahrscheinlich sind.

Wir wollen hier anmerken, daß in der Praxis gewisse Permutationen häufiger auftreten als andere, da die Datenerhebung gewissen Gesetzen gehorchen kann. Zum Beispiel können die Daten aus vielen bereits sortierten Dateien zusammengesetzt sein. Da aber im allgemeinen hierüber nichts bekannt ist, können wir das auch bei unserer Analyse nicht ausnützen. Auf der anderen Seite wird man aber bei Kenntnis von gewissen Strukturen in der zu sortierenden Folge, diese Kenntnis aber auch versuchen auszunutzen und nicht sturheil irgendeinen Sortieralgorithmus einsetzten.

Im folgenden bezeichnen wir mit $\bar{V}_{Quick}(n)$ die mittlere Anzahl von Vergleichen von Quicksort auf einem Array der Länge n. Zuerst überlegen wir uns, mit welcher Wahrscheinlichkeit das k-kleinste Element als Pivot-Element ausgewählt wird. Es gibt insgesamt $n! = 1 \cdot 2 \cdots n$ verschiedene Permutationen von n Elementen und $(n-1)!$ verschiedene Permutationen, in denen das k-kleinste Element am Ende steht. Also ist die Wahrscheinlichkeit, daß wir das k-kleinste Element als Pivot wählen gerade $\frac{(n-1)!}{n!} = \frac{1}{n}$.

Um eine Rekursionsgleichung aufteilen zu können, müssen wir uns nun noch überlegen, daß die im Aufteilungsschritt entstehenden Teilfolgen ebenfalls wieder gleichverteilt sind. Dazu betrachten nach einem Aufteilungsschritt, in dem das Pivot ein Element mit Rang k ist, den linken Bereich. Wir überlegen uns zunächst aus wievielen Folgen der Länge n, bei denen das Element mit Rang k am Ende steht, eine feste

Folge $(x_1, \ldots, x_{k-1})$ der Länge $k-1$ nach einem Aufteilungsschritt entstehen kann (allerdings bevor das Pivot-Element an die k-te Stelle getauscht wird). Wir bezeichnen diese Anzahl mit $A(x_1, \ldots, x_{k-1})$. Während des Aufteilungsschrittes können $i \in [0 : \max\{k-1, n-k\}]$ Vertauschungen stattgefunden haben (Zur Illustrierung dieser Konstruktion siehe auch Abbildung 2.30). Für diese i Vertauschungen gibt es

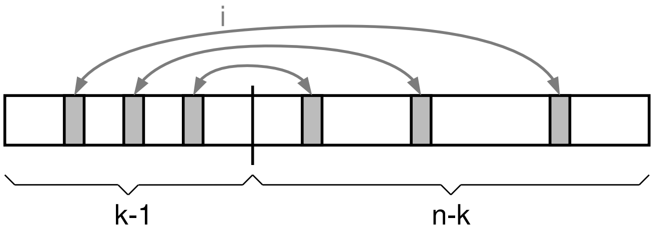

Abbildung 2.30: Entstehung einer Folge der Länge $k-1$ nach einen Aufteilungsschritt

$\binom{k-1}{i}$ verschiedene Möglichkeiten i Positionen im linken Teil (dem Bereich mit den Elementen kleiner als dem Pivot) und $\binom{n-k}{i}$ verschiedene Möglichkeiten i Positionen im rechten Teil (dem Bereich mit den Element größer als dem Pivot). Außerdem können wir aus dem rechten Teil noch $\binom{n-k}{i}$ verschiedene Gruppen von i Elementen auswählen, die in diese Vertauschung verwickelt sind. Die restlichen $(n-k-i)$ Elemente auf der rechten Seite können nun noch in irgendeiner der $(n-k-i)!$ verschiedenen Permutationen angeordnet sein. Damit gilt

$$A(x_1, \ldots, x_{k-1}) = \sum_{i=0}^{\max\{k-1,n-k\}} \binom{k-1}{i} \binom{n-k}{i}^2 (n-k-i)!$$

Wie man leicht sieht, ist diese Anzahl von der betrachteten Folge $(x_1, \ldots, x_{k-1})$ unabhängig und daher eine feste Zahl unabhängig von den einzelnen x_i. Da es genau $(n-1)!$ Permutationen gibt, in denen das Element mit Rang k am Ende steht, muß

$$\sum_{(x_1,\ldots,x_{k-1}) \in S(k-1)} A(x_1, \ldots, x_{k-1}) = (k-1)! \cdot A(x_1, \ldots, x_{k-1}) = (n-1)!$$

gelten, wobei $S(n)$ die Menge aller Permutationen auf n Elementen ist. Daraus folgt, daß $A(x_1, \ldots, x_{k-1}) = \frac{(n-1)!}{(k-1)!}$. Da es insgesamt $(n-1)!$ verschiedene Folgen gibt, in denen das Element mit Rang k als Pivot ausgewählt wird, erscheint jede Folge der Länge $k-1$ im linken Teil nach dem Aufteilungsschritt mit Wahrscheinlichkeit $\frac{1}{(k-1)!}$, d.h. es sind alle in der Rekursion auftretenden Folgen gleichwahrscheinlich. Analog kann man auch zeigen daß alle Folgen rechts vom Pivot mit gleicher Wahrscheinlichkeit auftreten. Damit ergibt sich die folgende Rekursionsgleichung:

$$\begin{aligned}
\bar{V}_{\text{Quick}}(n) &= 0 \qquad \text{für } n < 2 \\
\bar{V}_{\text{Quick}}(n) &= (n-1) + \frac{1}{n}\left(\sum_{k=1}^{n} \bar{V}_{\text{Quick}}(k-1) + \bar{V}_{\text{Quick}}(n-k)\right) \qquad \text{für } n > 1
\end{aligned}$$

Die Rekursionsgleichung läßt sich wie folgt vereinfachen:

$$\begin{aligned}
\bar{V}_{\text{Quick}}(n) &= (n-1) + \frac{1}{n}\left(\sum_{k=1}^{n} \bar{V}_{\text{Quick}}(k-1) + \bar{V}_{\text{Quick}}(n-k)\right) \\
&= (n-1) + \frac{1}{n}\sum_{k=0}^{n-1} \bar{V}_{\text{Quick}}(k) + \frac{1}{n}\sum_{k=0}^{n-1} \bar{V}_{\text{Quick}}(k) \\
&= (n-1) + \frac{2}{n}\sum_{k=1}^{n-1} \bar{V}_{\text{Quick}}(k)
\end{aligned}$$

Zur Lösung dieser Rekursionsgleichung multiplizieren wir diese erst mit n:

$$n\bar{V}_{\text{Quick}}(n) = n(n-1) + 2\sum_{k=1}^{n-1} \bar{V}_{\text{Quick}}(k)$$

und betrachten dieselbe Gleichung für $n-1$ anstatt für n:

$$(n-1)\bar{V}_{\text{Quick}}(n-1) = (n-1)(n-2) + 2\sum_{k=1}^{n-2} \bar{V}_{\text{Quick}}(k);$$

Nun subtrahieren wir die letzte Gleichung von der vorherigen und erhalten:

$$n\bar{V}_{\text{Quick}}(n) - (n-1)\bar{V}_{\text{Quick}}(n-1) = n(n-1) - (n-1)(n-2) + 2\bar{V}_{\text{Quick}}(n-1).$$

Dies ist offensichtlich äquivalent zu:

$$n\bar{V}_{\text{Quick}}(n) = (n+1)\bar{V}_{\text{Quick}}(n-1) + 2(n-1).$$

Nach Division durch $n(n+1)$ erhalten wir

$$\frac{\bar{V}_{\text{Quick}}(n)}{n+1} = \frac{\bar{V}_{\text{Quick}}(n-1)}{n} + \frac{2(n-1)}{n(n+1)} = \frac{\bar{V}_{\text{Quick}}(n-1)}{n} + \frac{4}{n+1} - \frac{2}{n}.$$

Durch Iteration erhalten wir somit:

$$\begin{aligned}
\frac{\bar{V}_{\text{Quick}}(n)}{n+1} &= \frac{\bar{V}_{\text{Quick}}(n-1)}{n} + \frac{4}{n+1} - \frac{2}{n} \\
&= \frac{\bar{V}_{\text{Quick}}(n-2)}{n-1} + \left(\frac{4}{n} - \frac{2}{n-1}\right) + \left(\frac{4}{n+1} - \frac{2}{n}\right)
\end{aligned}$$

$$
\begin{aligned}
& \text{durch weitere Iteration erhält man} \\
= \quad & \frac{\bar{V}_{\text{Quick}}(n-i)}{n+1-i} + \sum_{j=1}^{i} \left(\frac{4}{n+2-j} - \frac{2}{n+1-j} \right) \\
& \text{für } i = n-1 \text{ erhält man dann} \\
= \quad & \underbrace{\frac{\bar{V}_{\text{Quick}}(1)}{2}}_{=0} + \sum_{j=1}^{n-1} \left(\frac{4}{n+2-j} - \frac{2}{n+1-j} \right) \\
= \quad & \sum_{j=3}^{n+1} \frac{4}{j} - \sum_{j=2}^{n} \frac{2}{j} \\
& H_n = \sum_{i=1}^{n} \tfrac{1}{i} \text{ bezeichne die } n\text{-te harmonische Zahl} \\
= \quad & \left(4H_n + \frac{4}{n+1} - \frac{4}{2} - 4 \right) - (2H_n - 2) \\
= \quad & 2H_n + \frac{4}{n+1} - 4 \\
& \text{Es gilt } H_n = \ln(n) + C + o(1) \\
& \text{mit der Euler-Mascheroni'schen Konstante } C = 0.57721\ldots \\
= \quad & 2\ln(n) + O(1) \\
= \quad & 2\ln(2)\,\log(n) + O(1)
\end{aligned}
$$

Damit erhalten wir für die mittlere Laufzeit für Quicksort:

$$\bar{V}_{\text{Quick}}(n) = 2\ln(2) n \log(n) + O(n) \approx 1.386\, n \log(n) + O(n)$$

Theorem 2.13 *Quicksort benötigt zum Sortieren eines Arrays der Länge n im Mittel $2\ln(2) n \log(n) + O(n) \approx 1.386\, n \log(n) + O(n)$ viele Vergleiche.*

Dies ist zwar ein sehr schönes Ergebnis, allerdings ist die schlechte worst-case Laufzeit sehr unschön. Es kann vorkommen, daß dieselbe Anzahl Daten in ca. einer Sekunde sortiert wird, während man im worst-case fast eine Stunde warten muß. Dies ist in der Regel nicht zumutbar. Daher ist auf Grund der niedrigen Anzahl von Vergleichen Bottom-Up-Heapsort eine echte Alternative zu Quicksort!

2.4.4 Varianten von Quicksort

Ein Weg um die schlechte worst-case-Laufzeit von Quicksort auszutricksen, ist die, daß man das Pivot-Element zufällig wählt. So tritt der worst-case nun nicht mehr bei bestimmten strukturierten Folgen auf, sondern kann (mit sehr kleiner Wahrscheinlichkeit) bei jeder Eingabe erfolgen. Man hat allerdings den Vorteil, daß man im worst-case den Sortiervorgang abbrechen und von Neuem beginnen kann. Ein mehrmaliges Hintereinanderauftreten des worst-case ist sehr unwahrscheinlich. Die prinzipielle Lücke ist aber auch hier nicht geschlossen.

Theorem 2.14 *Der randomisierte Quicksort benötigt zum Sortieren eines Arrays der Länge n im Erwartungswert $2\ln(2)n\log(n)+O(n) \approx 1.386\,n\log(n)+O(n)$ viele Vergleiche.*

Eine weitere Verbesserung kann man erzielen, indem man drei Elemente der Folge zufällig wählt und das mittlere der drei Elemente als Pivot-Element hernimmt. Dadurch kann man die mittlere bzw. erwartete Anzahl von Vergleichen noch einmal leicht drücken. Die mathematisch aufwendige Analyse wollen wir an dieser Stelle nicht ausführen und erwähnen an dieser Stelle nur das Ergebnis:

Theorem 2.15 *Quicksort, der sein Pivot-Element als den Median aus drei Elementen an fest gewählten Positionen des Arrays bestimmt, benötigt zum Sortieren eines Array der Länge n im Mittel $\frac{12}{7}\ln(2)\,n\log(n)+O(n) \approx 1.188\,n\log(n)+O(n)$ Vergleiche.*

Theorem 2.16 *Quicksort, der sein Pivot-Element als den Median aus drei zufällig gewählten Elementen des Arrays bestimmt, benötigt zum Sortieren eines Array der Länge n im Erwartungswert $\frac{12}{7}\ln(2)\,n\log(n)+O(n) \approx 1.188\,n\log(n)+O(n)$ Vergleiche.*

Eine weitere Möglichkeit wie man Quicksort auch im worst-case stabil macht, ist eine geschickte Wahl des Pivot-Elementes. Wählt man als Pivot-Element immer den Median (oder zumindest ein Element das hinreichend weit weg vom Rand ist), so kann man die worst-case Laufzeit von Quicksort auf $O(n\log(n))$ drücken, sofern man dieses Pivot-Element in linearer Zeit finden kann. Dieses Problem wird uns im übernächsten Abschnitt beschäftigen.

Ein anderer subtiler Trick ist es, die Rekursion von Quicksort nicht bis zum Ende durchlaufen zu lassen, sondern für kleine Folgen einen einfachen Sortieralgorithmus wie Insertionsort anzuwenden. Die Praxis zeigt, daß es zum Beispiel sinnvoller ist, Arrays mit weniger als ca. 10 Elemente mit Insertionsort zu sortieren.

Wir wollen hier darauf hinweisen, daß es ebenfalls möglich ist Quicksort iterativ zu implementieren. Hierbei muß man dann aber mit Hilfe eines Stacks die Indexgrenzen der noch nicht sortierten Teilarrays selbst verwalten. Die meisten Compiler optimieren den Code bereits so, daß die iterative Variante eher langsamer wird.

3. Juni 97

2.5 Interludium: Divide&Conquer-Algorithmen

Mit Mergesort und Quicksort haben wir zwei schnelle Sortieralgorithmen gefunden, die nach demselben Entwurfsprinzip für Algorithmen konstruiert wurden: dem sogenannten *Divide-and-Conquer*. Daher wollen wir in diesem Abschnitt nun näher auf dieses allgemeine und sehr mächtige Grundprinzip eingehen.

2.5.1 Prinzip

Das Prinzip des Divide-and-Conquer (oft auch als divide-et-impera bzw. Teile-und-Herrsche-Prinzip bezeichnet) läßt sich im wesentlichen in drei Teile gliedern:

- Zuerst wird das Problem in mehrere kleinere Teilprobleme derselben Art aufgeteilt (im sogenannten *Divide-Schritt*);
- Dann werden die kleineren Teilprobleme rekursiv gelöst;
- Zuletzt wird aus den Lösungen der Teilprobleme eine Lösung für das Gesamtproblem konstruiert (im sogenannten *Conquer-Schritt*).

Am Beispiel des Mergesorts, war der Divide-Schritt sehr einfach. Wenn man nicht weiß was man tun soll, teilt man das Problem einfach irgendwie in zwei Teile. Das einzig geschickte hieran war, das man versucht, die Teile möglichst gleich groß zu wählen. Im Conquer-Schritt war dann die eigentliche Arbeit zu verrichten, nämlich im Mischen der sortierten Teilfolgen.

Beim Quicksort war es gerade andersherum. Hier hat man sich bereits im Divide-Schritt die Arbeit gemacht, indem man die Folge so aufteilte, daß eine Teilfolge nur 'kleine' und die andere Folge nur 'große' Elemente enthielt. Dadurch konnte der Conquer-Schritt sehr einfach gehalten werden, die beiden Folgen mußten nur aneinander gehängt werden.

Im allgemeinen ist es jedoch so, daß man sowohl im Divide-Schritt wie im Conquer-Schritt Arbeit verrichten muß. Außerdem wird es oft nicht genügen, das Problem nur in zwei Teilprobleme aufzuspalten. Oft wird man sie auch in mehrere Teilprobleme (oder auch nur eines) aufteilen müssen.

2.5.2 Ansatz für eine allgemeinere Analyse

Nun versuchen wir für einen Divide-and-Conquer Algorithmus ganz allgemein die Komplexität des erhaltenen Algorithmus abzuschätzen. Dazu bezeichne $D(n)$ bzw. $C(n)$ die Komplexität des Divide- bzw. des Conquer-Schritts. Sei weiter a die Anzahl der entstandenen Teilprobleme und n_i für $i \in [1:a]$ die Größe der einzelnen Teilprobleme. Damit ergibt sich für die Komplexität $\mathcal{C}(n)$ des Gesamtproblems die folgende Rekursionsgleichung:

$$\begin{aligned}
\mathcal{C}(n) &= O(1) \qquad \text{für } n \leq n_0 \\
\mathcal{C}(n) &= \sum_{i=1}^{a} \mathcal{C}(n_i) + D(n) + C(n) \qquad \text{für } n > n_0
\end{aligned}$$

In dieser Allemeinheit ist es schwer diese Rekursionsgleichung zu lösen. Darum wollen wir die Rekursionsgleichung unter ein paar Einschränkungen lösen. Allerdings werden die meisten interessanten Fälle dennoch durch diese Analyse abgedeckt. Wir wollen dazu annehmen, daß alle Teilprobleme dieselbe Größe haben, und zwar $n_i = n/b$ für ein $b > 1$. Für $b \leq 1$ müßten in der Rekursion Teilprobleme gelöst werden, welche größer als das ursprüngliche Problem sind. In diesem Fall terminiert die Rekursion im allgemeinen nicht. Für die Komplexität des Divide- und des Conquer-Schritts schreiben wir kurz $f(n) = D(n) + C(n)$. Somit erhalten wir:

$$\begin{aligned}
\mathcal{C}(n) &= d \qquad \text{für } n = 1 \\
\mathcal{C}(n) &= a \cdot \mathcal{C}(n/b) + f(n) \qquad \text{für } n > 1
\end{aligned}$$

Wir werden nun diese Rekursionsgleichung für $n = b^k$ durch Iteration lösen.

$$\begin{aligned}
\mathcal{C}(n) &= a \cdot \mathcal{C}(n/b) + f(n) \\
&= a\Big(a \cdot \mathcal{C}(n/b^2) + f(n/b)\Big) + f(n) \\
&= a^2 \cdot \mathcal{C}(n/b^2) + a \cdot f(n/b) + f(n) \\
&= a^2\Big(a \cdot \mathcal{C}(n/b^3) + f(n/b^2)\Big) + a \cdot f(n/b) + f(n) \\
&= a^3 \cdot \mathcal{C}(n/b^3) + a^2 \cdot f(n/b^2) + a \cdot f(n/b) + f(n) \\
&\ \text{nach } k \text{ Iterationen erhalten wir} \\
&= a^k \cdot \mathcal{C}(n/b^k) + \sum_{i=0}^{k-1} a^i \cdot f(n/b^i) \\
&\ \text{mit } k = \log_b(n) \text{ erhalten wir} \\
&= a^{\log_b(n)} \cdot \underbrace{\mathcal{C}(n/b^{\log_b(n)})}_{=\mathcal{C}(1)=d} + \sum_{i=0}^{\log_b(n)-1} a^i \cdot f(n/b^i)
\end{aligned}$$

$$= \quad d \cdot n^{\log_b(a)} + \sum_{i=0}^{\log_b(n)-1} a^i \cdot f(n/b^i)$$

2.5.3 Analyse eines Spezialfalles

Zuerst einmal werden wir den einfachen Fall untersuchen, wenn f eine lineare Funktion ist, also $f(n) = c \cdot n$.

1. Fall ($a < b$):

$$\begin{aligned}
\mathcal{C}(n) &\leq d \cdot n^{\log_b(a)} + \sum_{i=0}^{\log_b(n)-1} c \cdot n \cdot \frac{a^i}{b^i} \\
&\leq d \cdot n^{\overbrace{\log_b(a)}^{<1}} + c \cdot n \sum_{i=0}^{\log_b(n)-1} \underbrace{\left(\frac{a}{b}\right)}_{<1}^{i} \\
& \quad \text{da } a/b < 1 \text{, ist die geometrische Reihe konvergent, also } O(1) \\
&\leq O(n)
\end{aligned}$$

Wir erhalten also eine lineare Komplexität, wenn in jedem Rekursionsschritt die Summe der Größen der Teilprobleme echt kleiner wird ($a \cdot n/b < n$) als die ursprüngliche Größe des Problems.

2. Fall ($a = b$):

$$\begin{aligned}
\mathcal{C}(n) &\leq d \cdot \underbrace{n^{\log_b(a)}}_{=n} + \sum_{i=0}^{\log_b(n)-1} c \cdot n \underbrace{\left(\frac{a}{b}\right)}_{=1}^{i} \\
&\leq O(n) + c \cdot n \log_b(n) \\
&\leq O(n \log(n))
\end{aligned}$$

Sind also in jedem Rekursionsschritt die Summen der Größen der Teilprobleme genauso groß wie die Größe des ursprünglichen Problems ($a \cdot n/b = n$), erhalten wir als Laufzeit $O(n \log(n))$. Mergesort ist ein Beispiel für diesen Typ von Divide-and-Conquer-Algorithmen ($a = b = 2$).

3. Fall ($a > b$):

$$\mathcal{C}(n) \quad \leq \quad d \cdot n^{\log_b(a)} + c \cdot n \sum_{i=0}^{\log_b(n)-1} \left(\frac{a}{b}\right)^i$$

$$
\begin{aligned}
&\le\ d\cdot n^{\log_b(a)} + c\cdot n\frac{\left(\frac{a}{b}\right)^{\log_b(n)}-1}{\frac{a}{b}-1}\\
&\le\ O\left(n^{\log_b(a)}\right) + O\left(n\left(\frac{a}{b}\right)^{\log_b(n)}\right)\\
&\le\ O\left(n^{\log_b(a)} + n\frac{\left(b^{\log_b(a)}\right)^{\log_b(n)}}{b^{\log_b(n)}}\right)\\
&\le\ O\left(n^{\log_b(a)} + n\frac{\left(b^{\log_b(n)}\right)^{\log_b(a)}}{n}\right)\\
&\le\ O\left(n^{\log_b(a)} + n^{\log_b(a)}\right)\ =\ O\left(n^{\log_b(a)}\right)
\end{aligned}
$$

Werden also in jedem Rekursionsschritt die Summen der Größen der Teilprobleme größer als die Größe des ursprünglichen Problems ($a\cdot n/b > n$), so erhalten wir eine polynomielle Laufzeit, wobei die Konstanten a und b den Grad des Polynoms bestimmen.

Nun können wir diese Rekursionsgleichung auch für den Fall abschätzen, wenn n keine Potenz von b ist. Dazu bemerken wir zuerst, daß für $f(n) = n$ die Lösung $\mathcal{C}(n)$ monoton wachsend ist. Also schätzen wir nun $\mathcal{C}(n)$ durch $\mathcal{C}(b^{\lceil\log_b(n)\rceil})$ ab. Da $b^{\lceil\log_b(n)\rceil} \le b\cdot n$ ist, erhalten wir für beliebiges n dieselbe Lösung wie für eine Potenz mit b, in der n durch $b\cdot n$ ersetzt ist. Durch Ausrechnen sieht man, daß man den zusätzlichen Aufwand im O verstecken kann.

2.5.4 Analyse für allgemeinere Fälle (*)

Unter gewissen Annahmen über die Funktion f können wir die Lösung dieser Rekursionsgleichung auch im allgemeinen näher beschreiben:

Theorem 2.17 *Seien $a \ge 1, b > 1$ und $k \in N$ konstant, $f(n)$ eine Funktion und $\mathcal{C}(n)$ definiert durch die Rekursionsgleichung $\mathcal{C}(n) = a\cdot\mathcal{C}(n/b) + f(n)$ für $n > 1$ und $\mathcal{C}(1) = d$. Dann ist*

$$
\mathcal{C}(n) = \begin{cases}
\Theta(n^{\log_b(a)}), & \text{falls } f(n) = O(n^{\log_b(a)-e}) \text{ für ein konstantes } e > 0,\\
\Theta(n^{\log_b(a)}\log_2(n)), & \text{falls } f(n) = \Theta(n^{\log_b(a)}),\\
\Theta(f(n)), & \text{falls } f(n) = \Omega(n^{\log_b(a)+e}) \text{ für ein konstantes } e > 0\\
& \text{und } a\cdot f(n/b) \le c\cdot f(n) \text{ für ein konstantes } c < 1;
\end{cases}
$$

Beweis: Sei $f(n) = O(n^{\log_b(a)-e})$, d.h. $f(n) \leq c \cdot n^{\log_b(a)-e}$ für ein konstantes $e > 0$. Wir zeigen $\mathcal{C}(n)/n^{\log_b(a)} \leq \alpha$ für eine Konstante α:

$$\begin{aligned}
\frac{\mathcal{C}(n)}{n^{\log_b(a)}} &\leq \frac{1}{n^{\log_b(a)}} \left(d \cdot n^{\log_b(a)} + \sum_{i=0}^{\log_b(n)-1} a^i c \left(\frac{n}{b^i}\right)^{\log_b(a)-e} \right) \\
&\leq d + \underbrace{\frac{n^{\log_b(a)}}{n^{\log_b(a)}}}_{=1} \cdot \frac{c}{n^e} \sum_{i=0}^{\log_b(n)-1} \left(\underbrace{\frac{a}{b^{\log_b(a)}}}_{=1} \cdot \frac{1}{b^{-e}} \right)^i \\
&= d + \frac{c}{n^e} \underbrace{\sum_{i=0}^{\log_b(n)-1} b^{i \cdot e}}_{\frac{b^{e \log_b n}-1}{b^e-1}} \\
&\leq d + \frac{c}{n^e} \cdot \frac{n^e}{b^e - 1} \\
&\leq d + \frac{c}{b^e - 1}
\end{aligned}$$

Daraus folgt nun, daß $\mathcal{C}(n) = O(n^{\log_b(a)})$. Offensichtlich gilt auch $n^{\log_b(a)} = O(\mathcal{C}(n))$, da $\mathcal{C}(n) \geq d \cdot n^{\log_b(a)}$. Daher gilt $\mathcal{C}(n) = \Theta(n^{\log_b(a)})$.

Sei $f(n) = \Theta(n^{\log_b(a)})$. Wir setzen $\Theta(n^{\log_b a})$ für $f(n)$ in $\mathcal{C}(n)$ ein und rechnen mit Θ:

$$\begin{aligned}
\mathcal{C}(n) &= d \cdot n^{\log_b(a)} + \sum_{i=0}^{\log_b(n)-1} a^i \cdot \Theta\left(\left(\frac{n}{b^i}\right)^{\log_b(a)} \right) \\
&= d \cdot n^{\log_b(a)} + \Theta\left(n^{\log_b(a)} \sum_{i=0}^{\log_b(n)-1} \underbrace{\frac{a^i}{b^{i \log_b(a)}}}_{=1} \right) \\
&= d \cdot n^{\log_b(a)} + \Theta\left(n^{\log_b(a)} \underbrace{\sum_{i=0}^{\log_b(n)-1} 1}_{=\log_b(n)} \right) \\
&= \Theta(n^{\log_b(a)} \log_2(n))
\end{aligned}$$

Sei $f(n) = \Omega(n^{\log_b(a)+e})$ für ein konstantes $e > 0$ und $af(n/b) \leq c \cdot f(n)$ für ein konstantes $c < 1$. Aus der letzten Voraussetzung folgern wir $a^i f(n/b^i) \leq c^i f(n)$. Damit können wir $\mathcal{C}(n)$ nach oben abschätzen:

$$\begin{aligned}
\mathcal{C}(n) &\leq d \cdot n^{\log_b a} + \sum_{i=0}^{\log_b(n)-1} c^i f(n) \\
&= d \cdot n^{\log_b a} + f(n) \frac{1 - c^{\log_b(n)}}{1 - c}
\end{aligned}$$

$$\leq \quad d \cdot n^{\log_b a} + \frac{f(n)}{1-c}$$

Mit $f(n) > \alpha \cdot n^{\log_b(a)+e}$ für ein α und $e > 0$ ergibt sich $\mathcal{C}(n) = O(f(n))$ und da $f(n) \leq \mathcal{C}(n)$, gilt $f(n) = O((n))$ und daher $\mathcal{C}(n) = \Theta(f(n))$. ∎

2.6 Selektieren

Wie wir im Abschnitt über Quicksort gesehen haben, können wir die Anzahl der Vergleiche von Quicksort auf $O(n\log(n))$ beschränken, wenn wir mit linear vielen Vergleichen den Median bestimmen können. Dazu werden wir in diesem Abschnitt ein allgemeineres Problem lösen, nämlich in einer gegebenen Menge mit n Elementen, das Element mit Rang k zu finden. Einen solchen Algorithmus, der aus einer Menge das Element mit Rang k herausfindet, nennt man einen *Selektionsalgorithmus*.

2.6.1 QuickSelect (Ü)

Zunächst stellen wir einen Selektionsalgorithmus vor, der im Mittel (bzw. im Erwartungswert) nur linear viele Vergleiche benötigt. Dieser kopiert im wesentlichen die Idee von Quicksort. Wir wählen wie im Aufteilungsschritt zuerst auf eine beliebige Weise ein Pivot-Element aus und teilen das Array wie im Aufteilungsschritt des Quicksorts auf. Dann bestimmen wir den Rang r des Pivot-Elementes und suchen nun rekursiv in der linken bzw. rechten Hälfte weiter, je nachdem, ob $k < r$ oder $k > r$ ist. Sollte $k = r$ sein, dann haben wir natürlich das Element mit Rang k gefunden.

In Abbildung 2.31 findet man die Implementierung von QuickSelect. Dabei bezeichnet in C $\&(array[i])$ das Teilarray ab Indexposition i.

2.6.2 Analyse von QuickSelect (Ü)

Mit $V_{\text{QuickSel}}(n,k)$ bezeichnen wir die Anzahl der Vergleiche, die QuickSelect für die Bestimmung des Elementes mit Rang k aus einer Menge mit n Elementen benötigt. Für die Anzahl der Vergleiche ergibt sich nun die folgende Rekursionsgleichung für $k \in [1:n]$, wenn als Pivot ein Element mit Rang m gewählt worden ist:

$$\begin{aligned} V_{\text{QuickSel}}(n,k) &= 0 \qquad \text{für } n < 2 \\ V_{\text{QuickSel}}(n,k) &= \begin{cases} (n-1) & \text{für } m = k \text{ und } n > 1 \\ (n-1) + V_{\text{QuickSel}}(m-1,k) & \text{für } k < m \text{ und } n > 1 \\ (n-1) + V_{\text{QuickSel}}(n-m,k-m) & \text{für } k > m \text{ und } n > 1 \end{cases} \end{aligned}$$

```
QuickSelect(int array[], int len, int k)
{
    /* returns position of element with rank k */
    int pivot = len − 1;
    pivot =partition(array, 0, len − 1, pivot);
    int rank = pivot + 1;
    if (k == rank)
        return pivot;
    if (k < rank)
        return QuickSelect(array, rank − 1, k);
    else
        return pivot + 1+
                QuickSelect(&(array[pivot + 1]), len − rank, k − rank);
}
```

Abbildung 2.31: QuickSelect

Im schlimmsten Falle erwischen wir als Pivot-Element immer das Minimum oder Maximum der Menge (ohne das Pivot selbst zu erwischen, mit Ausnahme bei einer einelementigen Menge). Damit ergibt sich wie bei Quicksort ein obere Schranke für die Anzahl der Vergleiche von $\binom{n}{2}$.

Allerdings kommt auch dieser Algorithmus im Mittel mit sehr viel weniger Vergleichen aus. Wieder nehmen wir an, daß alle Permutationen gleichwahrscheinlich sind, also hat das Pivot mit Wahrscheinlichkeit $\frac{1}{n}$ den Rang m. Wie bei Quicksort, sind bei der rekursiv betrachteten Teilfolge wieder alle Permutationen gleichwahrscheinlich. Wir werden im folgenden die Anzahl der Vergleiche für QuickSelect der Einfachheit halber unabhängig von k nach oben abschätzen. Die Anzahl der Vergleiche bezeichnen wir wieder mit $\bar{V}_{\text{QuickSel}}(n)$. Damit erhalten wir unter der leicht nachweisbaren Annahme, daß $\bar{V}_{\text{QuickSel}}(n)$ monoton wachsend ist, die folgende Rekursionsgleichung:

$$\begin{aligned}
\bar{V}_{\text{QuickSel}}(n) &= 0 \qquad \text{für } n < 2 \\
\bar{V}_{\text{QuickSel}}(n) &\leq (n-1) + \frac{1}{n}\sum_{i=1}^{n} \bar{V}_{\text{QuickSel}}(\max\{i-1, n-i\}) \qquad \text{für } n > 1
\end{aligned}$$

Da $\max\{i-1, n-i\} \geq \lfloor\frac{n+1}{2}\rfloor$ für $i \in [1:n]$, erhalten wir somit:

$$\begin{aligned}
\bar{V}_{\text{QuickSel}}(n) &\leq (n-1) + \frac{1}{n}\sum_{i=1}^{n} \bar{V}_{\text{QuickSel}}(\max\{i-1, n-i\}) \\
&\leq (n-1) + \frac{2}{n}\sum_{i=\lfloor\frac{n+1}{2}\rfloor}^{n-1} \bar{V}_{\text{QuickSel}}(i)
\end{aligned}$$

Wir werden nun durch Induktion zeigen, daß diese Rekursionsgleichung impliziert, daß $\bar{V}_{\text{QuickSel}}(n) \leq 4 \cdot n$ gilt. Für $n < 2$ ist dies offensichtlich richtig, und der Induktionsanfang ist damit gelegt.

$$
\begin{array}{rcl}
\bar{V}_{\text{QuickSel}}(n) & \leq & (n-1) + \dfrac{2}{n} \displaystyle\sum_{i=\lfloor \frac{n+1}{2} \rfloor}^{n-1} \bar{V}_{\text{QuickSel}}(i) \\
& \stackrel{\text{I.V.}}{\leq} & (n-1) + \dfrac{2}{n} \displaystyle\sum_{i=\lfloor \frac{n+1}{2} \rfloor}^{n-1} 4i \\
& & \text{mit } \lfloor \frac{n+1}{2} \rfloor \geq \frac{n}{2} \\
& \leq & (n-1) + \dfrac{2}{n} \displaystyle\sum_{\frac{n}{2} \leq i \leq n-1} 4i \\
& \leq & (n-1) + \dfrac{2}{n} \left(\displaystyle\sum_{i=1}^{n-1} 4 \cdot i - \sum_{1 \leq i \leq \frac{n}{2}-1} 4 \cdot i \right) \\
& & \text{mit } \sum_{i=1}^{n-1} = \frac{n(n-1)}{2} \\
& \leq & (n-1) + \dfrac{2}{n} \left(\dfrac{4}{2}(n(n-1)) - \dfrac{4}{2} \cdot \dfrac{n}{2} \left(\dfrac{n}{2} - 1 \right) \right) \\
& \leq & (n-1) + \dfrac{4}{n} \left(n^2 - n - \dfrac{n^2}{4} + \dfrac{n}{2} \right) \\
& \leq & 4 \cdot n - 3 \ \leq \ 4 \cdot n
\end{array}
$$

Theorem 2.18 *Die Prozedur QuickSelect benötigt zum Bestimmen des Elementes mit Rang k aus einer n-elementigen Menge im Mittel höchstens $4 \cdot n$ Vergleiche.*

Natürlich können wir das Pivot-Element auch wieder zufällig wählen, dann gilt die obige Analyse entsprechend auch für die randomisierte Version von QuickSelect.

Theorem 2.19 *Die randomisierte Version der Prozedur QuickSelect, die das Pivot-Element zufällig auswählt, benötigt zum Bestimmen des Elementes mit Rang k aus einer n-elementigen Menge im Erwartungswert höchstens $4 \cdot n$ Vergleiche.*

2.6.3 Ein linearer Selektionsalgorithmus

Nun wollen wir einen Algorithmus vorstellen, der in jedem Falle nur linear viele Vergleiche zur Bestimmung des Elementes mit Rang k einer n-elementigen Menge benötigt. Dieser Algorithmus geht auf M. Blum, R.W. Floyd, V.R. Pratt, R.L. Rivest und R.E. Tarjan aus dem Jahre 1973 zurück. Nach ihren Entwicklern wird er oft auch *BFPRT-Algorithmus* genannt.

Ohne Beschränkung der Allgemeinheit nehmen wir an, daß die Anzahl der Menge mindestens 11 ist. Andernfalls läßt sich ein Element mit Rang k einer so kleinen Menge sehr einfach finden.

Zuerst teilen wir die n Elemente in $\lfloor \frac{n}{5} \rfloor$ Gruppen zu je 5 Elementen und maximal vier Gruppen mit je 4 Elementen ein. Im folgenden bezeichnen wir der Einfachheit halber so eine Gruppe mit vier oder fünf Elementen immer als 5-er Gruppen, auch wenn so eine Gruppe in Wirklichkeit nur vier Elemente besitzt. Nun bestimmen wir von jeder 5-er Gruppe den Median dieser fünf bzw. vier Elemente. Bei diesen Medianbestimmungen wird gleichzeitig noch mitbestimmt, welche Elemente kleiner oder größer als der entsprechende Median sind. Anschließend bestimmen wir rekursiv aus den $\lceil \frac{n}{5} \rceil$ Medianen deren Median.

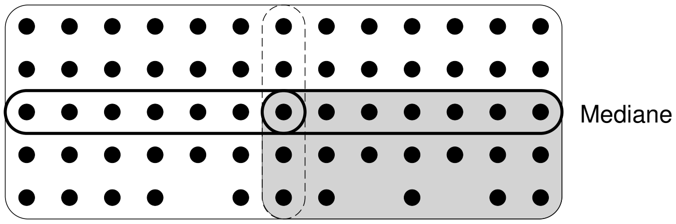

Abbildung 2.32: Die Aufteilung der Mediane vor dem Aufruf der zweiten Rekursion

Eine schematische Darstellung dieses Aufteilung ist in Abbildung 2.32 gegeben. Jede Spalte bildet hier eine der 5-er Gruppen. Dabei sind die Elemente, die kleiner als der Median sind, oberhalb und die Elemente, die größer als der Median sind, unterhalb des Medians der entsprechenden 5-er Gruppe angeordnet. Die Mediane selbst sind von links nach rechts aufsteigend geordnet, wobei die einzelnen 5-er Gruppen (Spalten) bei dieser Darstellung mitpermutiert wurden. Wir wollen an dieser Stelle bereits darauf hinweisen, daß wir im Algorithmus in Abbildung 2.33 nach dem Aufruf von `partition` eine andere, feinere Aufteilung der Elemente bekommen. Dort bestimmen wir den Rang des Medians wirklich, während wir in dieser Darstellung den Rang des Medians der Mediane nur auf ein Intervall einschränken.

Nun bestimmen wir noch den Rang des Medians der Mediane und vergleichen den Rang des Medians der Mediane mit dem Rang k des gesuchten Elementes. Danach können wir ca. ein viertel der Elemente als Kandidaten für das Element mit Rang k verwerfen. Ist der Rang des Medians der Mediane größer als der gesuchte Rang, dann können wir alldiejenigen Elemente verwerfen, die größer als der Median sind.

```
Select(int array[], int len, int k)
{ /* returns position of element with rank k */
    if (len ≤ 64)
    {
        Heapsort(array, len);
        return k − 1;
    }
    else
    {
        determine_medians(array, len);
        int cols = ⌈n/5⌉;
        int median = 2 ∗ cols+Select(&(array[2 ∗ cols]), cols, ⌈(cols+1)/2⌉);
        median =partition(array, 0, len − 1, median);
        int rank = 1 + median;
        if (k == rank)
            return median;
        if (k < rank)
            return Select(array, rank − 1, k);
        else
            return median + 1+
                Select(&(array[median + 1]), len − rank, k − rank);
    }
}

determine_medians(int array[], int len)
{
    /* determine medians of columns */
    int five[5], i, j, max = 0;
    for (i = 0; i < ⌈len/5⌉; i++)
    {
        for (j = 0; i + j ∗ ⌈len/5⌉ < len; j++)
            five[j] = array[i + j ∗ ⌈len/5⌉];
        int max = j; /* < 5*/
        /* determine median s.t. median is stored in five[2] and the */
        /* smaller (bigger) elements are stored in five[0, 1] (five[3, 4]) */
        median_of_five(five, max);
        for (j = 0; j < max; j++)
            array[i + j ∗ ⌈len/5⌉] = five[j];
    }
}
```

Abbildung 2.33: Der BFPRT-Selektionsalgorithmus

Das sind die drei Elemente in derselben 5-er Gruppe des Medians und jeweils die drei Elemente die größer sind als ein Median, der selbst größer als der Median der Mediane ist. Diese Elemente sind in Abbildung 2.32 grau hinterlegt. In der restlichen Menge suchen wir nun rekursiv weiter. Auch dies ist wieder ein Divide-and-Conquer Algorithmus, wobei wir hier bereits im Divide-Schritt von einem rekursiven Aufruf Gebrauch machen.

6. Juni 97

Der Algorithmus selbst ist in Abbildung 2.33 angegeben. In dieser Implementierung stellen wir uns das Array als $5 \times (len/5)$-Matrix vor, in der die Elemente zeilenweise durchnumeriert sind. Dadurch das wir uns das Array zeilenweise durch die $5 \times \lceil \frac{n}{5} \rceil$-Matrix vorstellen, bilden die Mediane der 5-er Gruppen nun ein kontinuierliches Stück des Arrays und wir können den Algorithmus rekursiv auf diesem Teilarray aufrufen. Dabei bezeichnet in C $\&(array[i])$ das Teilarray ab Indexposition i.

2.6.4 Analyse des BFPRT-Selektionsalgorithmus

Bezeichne im folgenden $V_{\text{BFPRT}}(n)$ die benötigte Anzahl von Vergleichen zum Finden des Elementes mit Rang k einer n-elementigen Menge. Zunächst müssen wir aus lauter 5-er Gruppen den Median bestimmen. Hierfür genügen 7 Vergleiche pro Gruppe ($\Rightarrow$ Übungen). Dann wird der BFPRT-Algorithmus rekursiv zum Bestimmen des Medians der Mediane aufgerufen ($V_{\text{BFPRT}}(\lceil n/5 \rceil)$ Vergleiche). Dann wird das Array nach dem Median der Mediane partitioniert ($n-1$ Vergleiche). Schließlich wird rekursiv weitergesucht. Hierfür müssen wir nun die Größe der Menge abschätzen in der weitergesucht wird.

Ohne Beschränkung der Allgemeinheit nehmen wir an, daß der Rang des Medians der Mediane größer als der gesuchte Rang k war. Hierzu betrachten wir noch einmal Abbildung 2.32. In dieser Abbildung sind in jeder Spalte die Elemente aufsteigend sortiert, und die Median sind von links nach rechts aufsteigend sortiert. Dann wissen wir, daß die grau unterlegten Elemente alle größer als der Median der Mediane ist, und somit ebenfalls einen Rang größer als k haben. Im oben angegeben Algorithmus werden die Elemente im Array nach `partition`-Schritt anders angeordnet, die hier gegebene Darstellung ist für die Analyse die angemessenere.

Wie groß kann nun der graue Bereich sein? Es gibt mindestens

$$\left\lceil \frac{\left\lceil \frac{n}{5} \right\rceil}{2} \right\rceil = \left\lceil \frac{n}{10} \right\rceil$$

Mediane die größer oder gleich als der Median der Mediane sind. Zu jedem Median wissen wir von noch zwei Elementen, daß sie größer als diese Mediane sind. Nur maximal vier Leerstellen können in dem grauen Bereich auftreten, wenn die Anzahl der Elemente nicht durch 5 teilbar war Damit ist die Anzahl der Element im grauen Bereich mindestens

$$3\left\lceil\frac{n}{10}\right\rceil - 4 \geq \frac{3n}{10} - 4.$$

Damit bleiben also im weißen Bereich, den wir jetzt noch zu durchsuchen haben höchstens

$$n - \left(\frac{3n}{10} - 4\right) = \frac{7n}{10} + 4$$

Elemente übrig. Damit erhalten wir die folgende Rekursionsgleichung:

$$V_{\text{BFPRT}}(n) \;\leq\; V_{\text{BFPRT}}\left(\left\lceil\frac{n}{5}\right\rceil\right) + V_{\text{BFPRT}}\left(\frac{7n}{10} + 4\right) + 7\left\lceil\frac{n}{5}\right\rceil + n - 1.$$

Wir zeigen nun mittels vollständiger Induktion, daß $V_{\text{BFPRT}}(n) \leq 25(n-3)$ für $n \geq 10$ ist.

Induktionsanfang ($10 \leq n \leq 64$): Die genaue Analyse von Heapsort zeigt, daß Heapsort maximal $2n\log(n) + 5n$ viele Vergleiche benötigt (siehe Theorem 2.6 auf Seite 44). Da $n \leq 64$ ist, ist $\log(n) \leq 6$. Damit folgt für $10 \leq n \leq 64$:

$$2n\log(n) + 5n \leq 17n = 25(n-3)\underbrace{-8n + 75}_{\leq 0 \text{ da } n \geq 10} \leq 25(n-3)$$

Damit ist der Induktionsanfang gelegt.

Induktionsschritt ($\rightarrow n > 64$): Als erstes bemerken wir daß für $n > 64$ alle rekursiv aufgerufenen Teilprobleme mindestens die Größe $\lceil\frac{65}{4}\rceil = 13$ haben, also ist der Induktionsanfang vollständig gelegt.

$$\begin{aligned}
V_{\text{BFPRT}}(n) &\leq V_{\text{BFPRT}}\left(\left\lceil\frac{n}{5}\right\rceil\right) + V_{\text{BFPRT}}\left(\frac{7n}{10} + 4\right) + 7\left\lceil\frac{n}{5}\right\rceil + n - 1 \\
&\ \text{da } \left\lceil\tfrac{n}{5}\right\rceil \geq 13 \geq 10 \text{ und } \tfrac{7n}{10} + 4 \geq 48 \geq 10 \text{ folgt} \\
&\overset{\text{I.V.}}{\leq} \left[25\left\lceil\frac{n}{5}\right\rceil - 75\right] + \left[\frac{7 \cdot 25n}{10} + 25 \cdot 4 - 75\right] + 7\left\lceil\frac{n}{5}\right\rceil + n - 1 \\
&\ \text{da } \lceil\tfrac{n}{5}\rceil \leq \tfrac{n+4}{5} \\
&\leq 25\left(\frac{n}{5} + \frac{4}{5}\right) + \frac{7 \cdot 25n}{10} + 7\left(\frac{n}{5} + \frac{4}{5}\right) + n - 51 \\
&\leq \frac{249n}{10} - \frac{254}{10} \\
&\leq 25n
\end{aligned}$$

Damit können wir nun das folgende Theorem formulieren.

Theorem 2.20 *Der BFPRT-Algorithmus benötigt zur Bestimmung eines Elements mit Rang k einer n-elementigen Menge maximal $25 \cdot n$ Vergleiche.*

Beweis: Für $n \geq 10$ haben wir bereits gezeigt, daß sogar $25(n-3)$ Vergleiche genügen. Andernfalls benötigt Heapsort nach Theorem 2.6 auf Seite 44 maximal $2n\log(n) + 5n$ viele Vergleiche. Da nun $n < 10$ ist folgt, daß $\log(n) \leq 4$ ist. Also benötigt Heapsort für $n < 10$ maximal $13n < 25n$ viele Vergleiche. ■

Mit einer etwas aufwendigeren Analyse (insbesondere im Induktionsanfang) läßt sich sogar zeigen, daß $24 \cdot n$ Vergleiche ausreichend sind.

Implementiert man den Algorithmus geschickter, so kann man die Anzahl der Vergleiche noch auf $16 \cdot n$ senken. Man kann auf den Aufruf von `partition` verzichten und stattdessen, wie in der Analyse, den Rang des Medians der Median nur so weit abschätzen, daß nur noch $\frac{7}{10}n + 4$ Elemente mehr als Element mit dem gesuchten Rang in Frage kommen. Dazu muß man nur den Median der Mediane mit den anderen Medianen Vergleichen, da wir die Ergebnisse der Vergleiche der Mediane mit den Elementen in ihren 5-er Gruppen ja bereits kennen. Damit kann man in der Rekursionsgleichung den Summanden $n-1$ durch $\lceil n/5 \rceil - 1$ ersetzten. Da diese Implementierung aber technisch aufwendiger ist und nicht auf eine Seite des Skripts paßt, wollen wir sie nur der Vollständigkeit halber erwähnen.

Aus der Komplexität der Selektion ergibt sich für die Komplexität der Medianbestimmung unmittelbar das folgende Korollar:

Korollar 2.21 *Der BFPRT-Algorithmus benötigt zur Bestimmung des Medians einer n-elementigen Menge maximal $25 \cdot n$ Vergleiche.*

Das folgende Theorem stellt den einfachen aber wirkungsvollen Zusammenhang zwischen dem Finden des Medians und dem Sortieren her.

Theorem 2.22 *Wenn es zur Bestimmung des Medians einer n-elementigen Menge maximal einen Algorithmus gibt, der maximal $c \cdot n$ Vergleiche benötigt, so benötigt die Variante von Quicksort, die als Pivot den Median auswählt, maximal $(c+1)n\log(n) + O(n)$ Vergleiche.*

Mit diesem Theorem, dessen Beweis wir als Übungsaufgabe dem Leser überlassen (⇒ Übungen), erhalten wir das folgende Korollar.

Korollar 2.23 *Die modifizierte Variante von Quicksort, die zur Bestimmung des Pivot den Median mit dem BFPRT-Algorithmus auswählt, benötigt zum Sortieren einer Folge der Länge n maximal $26\,n\log(n) + O(n)$ Vergleiche.*

10. Juni 97

2.6.5 Neuere Ergebnisse (*)

Lange Zeit war der Medianalgorithmus von A. Schönhage, A. Paterson und N. Pippenger aus dem Jahr 1976 der beste bekannte und benötigte $3n + o(n)$ Vergleiche. Dieser wurde von D. Dor und U. Zwick im Jahre 1995 noch verbessert.

Theorem 2.24 *Der Median einer n-elementigen Menge kann mit $2.95 \cdot n + o(n)$ Vergleichen gefunden werden.*

Im Jahre 1996 wurde die bislang beste untere Schranke ebenfalls von D. Dor und U. Zwick gefunden.

Theorem 2.25 *Jeder Algorithmus zur Bestimmung des Median einer Menge aus n Elementen benötigt im schlimmsten Fall mindestens $(2 + \varepsilon)n - o(n)$ Vergleiche.*

Das ε in diesem Theorem ist allerdings sehr klein, $\varepsilon \approx 2^{-80}$.

2.7 Eine untere Schranke für das Sortieren

In diesem Abschnitt wollen wir eine untere Schranke für das Sortieren herleiten. Wir werden dabei annehmen, daß wir nur Folgen sortieren wollen, in denen alle Elemente paarweise verschieden sind. Außerdem betrachten wir nur vergleichsbasierte Sortieralgorithmen. Ein *vergleichsbasierter Sortieralgorithmus* verwendet für seine Entscheidungen nur die Ergebnisse von Vergleichen auf den zu sortierenden Schlüsseln. Alle bisher betrachteten Sortierverfahren waren vergleichsbasiert.

2.7.1 Entscheidungsbaum

Betrachten wir nun einen beliebigen vergleichsbasierten Sortieralgorithmus $\mathcal{A}$. Wir können uns einen solchen Sortieralgorithmus für eine Eingabe der Länge n als Baum vorstellen. An den inneren Knoten notieren wir die ausgeführten Vergleiche des Sortieralgorithmus auf einer Eingabe der Länge n. Da für jeden solchen Vergleich nur zwei Antworten in Frage kommen (wir haben ja ausgeschlossen, daß ein Element mehrfach in der Eingabefolge auftritt), erhalten wir so einen binären Baum $T_{\mathcal{A}}(n)$, der für Eingabefolgen der Länge n den Sortieralgorithmus beschreibt. Ohne Beschränkung der Allgemeinheit verzweigen wir zum linken Kind, wenn der im Knoten notierte Vergleich x_i mit x_j ergeben hat, daß $x_i < x_j$ ist, und wir verzweigen zum rechten Kind, wenn der Vergleich $x_i > x_j$ ergeben hat. An der Wurzel notieren wir den ersten ausgeführten Vergleich. An jedem weiteren inneren Knoten notieren wir den Vergleich, der ausgeführt wird unter der Annahme, daß die in dem eindeutigen Pfad von der Wurzel zu diesem inneren Knoten notierten Vergleiche so ausgegangen sind, wie es die Verzweigungsstruktur des Baums angibt. An einem Blatt notieren wir dann die Permutation, die uns angibt, wie die Eingabefolge zu permutieren ist, damit die Folge aufsteigend sortiert ist, unter Berücksichtigung der ausgeführten Vergleiche. Einen solchen Baum $T_{\mathcal{A}}(n)$ nennen wir auch *Entscheidungsbaum* für den Sortieralgorithmus $\mathcal{A}$ auf Eingaben der Länge n.

In Abbildung 2.34 ist als Beispiel ein Entscheidungsbaum für Mergesort auf Eingaben der Länge 4 angegeben.

2.7.2 Maximale Anzahl von Vergleichen

Da es $n!$ verschiedene Permutationen einer Folge der Länge n gibt, muß der Entscheidungsbaum eines korrekten Sortieralgorithmus für Folgen der Länge n mindestens $n!$ viele Blätter haben. Ansonsten gäbe es mindestens eine Eingabefolge der Länge n, die dieser Sortieralgorithmus nicht korrekt sortieren würde.

Lemma 2.26 *Jeder Entscheidungsbaum eines Sortieralgorithmus für Folgen der Länge n hat mindestens $n!$ Blätter.*

Auf der anderen Seite gibt der Level ℓ eines Blattes an wieviele Vergleiche der Algorithmus zum Sortieren dieser einen Folge benötigt hat, nämlich genau $\ell - 1$. Wir haben nämlich an jedem Knoten, der kein Blatt ist, einen Vergleich ausgeführt. Daher definieren wir nun die *Höhe* eines Baumes als der maximale Level eines Blattes minus 1. Zur einfacheren Schreibweise, bezeichne $h(T)$ die Höhe und $\lambda(T)$ die Anzahl der Blätter eines Baumes T.

Lemma 2.27 *Ein Sortieralgorithmus benötigt für Folgen der Länge n mindestens soviele Vergleiche, wie die Höhe seines zugehörigen Entscheidungsbaumes für Eingaben der Länge n.*

Um nun eine untere Schranke herleiten zu können, benötigen wir für binäre Bäume noch eine Beziehung zwischen der Anzahl seiner Blätter und seiner Höhe. Zuerst können wir einfach mit Induktion zeigen, daß jeder Baum mit Höhe h höchstens 2^h viele Blätter haben kann.

Lemma 2.28 *Für einen binären Baum T gilt:*

$$\lambda(T) \leq 2^{h(T)}.$$

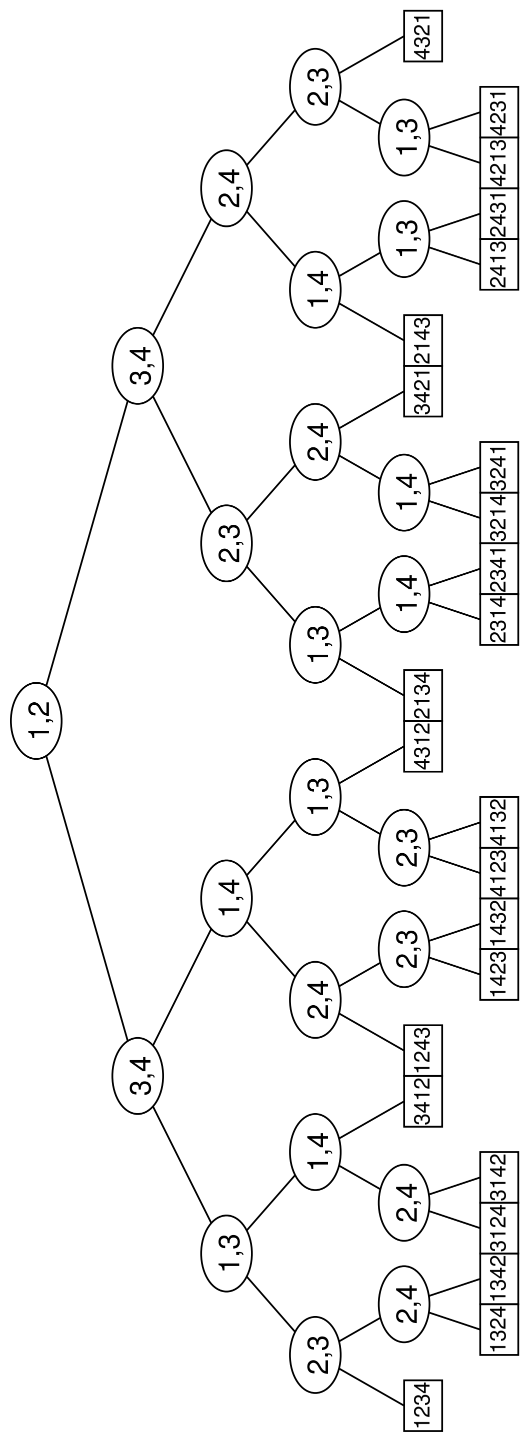

Abbildung 2.34: Der Entscheidungsbaum für Mergesort für Eingaben der Länge 4

Beweis: Wir führen den Beweis durch vollständige Induktion über die Höhe des Baumes.

Induktionsanfang ($h = 0$): Der Baum besteht nur aus einem Knoten, nämlich der Wurzel, die zugleich das einzige Blatt ist. Also gilt $1 \leq 2^0$.

Induktionsschritt ($h - 1 \to h$): Betrachten wir einen Baum T mit Höhe h und entfernen alle Blätter auf dem maximalen Level $h+1$. Der so entstandene Baum T' hat höchstens Höhe $h-1$ und somit nach Voraussetzung höchstens 2^{h-1} viele Blätter. Beim Entfernen der Blätter auf dem maximalen Level haben wir höchstens für jedes neu entstandene Blatt in T' maximal zwei Blätter aus T gelöscht, da T ja binär ist. Also ist die Anzahl der Blätter in T maximal zweimal so groß wie die Anzahl der Blätter in T' und somit:

$$\lambda(T) \leq 2 \cdot \lambda(T') \overset{\text{I.V.}}{\leq} 2 \cdot 2^{h-1} = 2^h.$$

■

Damit haben wir die folgende Äquivalenz bewiesen:

$$\lambda(T) \leq 2^{h(T)} \quad \iff \quad h(T) \geq \lceil \log(\lambda(T)) \rceil.$$

Aus der letzten Äquivalenzumformung erhalten wir das folgende Lemma:

Lemma 2.29 *Jeder binäre Baum mit n Blättern hat mindestens Höhe $\lceil \log(n) \rceil$.*

Aus den Lemmata 2.26, 2.27 und 2.29 erhalten wir sofort eine untere Schranke für die Anzahl der Vergleiche eines Sortieralgorithmus, wobei $V(n)$ die maximal benötigte Anzahl von Vergleichen zum Sortieren einer Folge der Länge n bezeichne:

$$V(n) \geq \lceil \log(n!) \rceil.$$

Um diese Abschätzung etwas anschaulicher formulieren zu können, verwenden wir die *Stirling'sche Formel*:

$$n! = \frac{n^n}{e^n} \sqrt{2\pi \cdot n} \left(1 + O\left(\frac{1}{n}\right)\right).$$

Damit erhalten wir:

Theorem 2.30 *Jeder vergleichsbasierte Sortieralgorithmus benötigt auf Folgen der Länge n mindestens $n \log(n) - n \log(e) + O(\log(n)) \approx n \log(n) - 1.44\, n$ Vergleiche.*

2.7.3 Mittlere Anzahl an Vergleichen (*)

Man kann sogar zeigen, daß die mittlere Höhe eines binären Baumes mit n Blättern mindestens $\log(n)$ ist.

Dazu beschränken wir uns zunächst auf *wiederholungsfreie Sortieralgorithmen*, die für jedes Paar von Elementen maximal einen Vergleich durchführen. Dies ist allerdings keine echte Einschränkung für eine untere Schranke, da der Algorithmus ja immer noch alle möglichen Paare von Elementen vergleichen darf. Wenn er wirklich ein Paar ein zweites Mal vergleichen möchte, so kann er das Ergebnis von der ersten Anfrage verwenden, da für den Beweis der unteren Schranke der Speicherplatzbedarf *nicht* beschränkt ist.

Damit kann man folgende Verstärkung von Lemma 2.26 zeigen:

Lemma 2.31 *Jeder Entscheidungsbaum eines wiederholungsfreien Sortieralgorithmus für eine Eingabe der Länge n hat genau $n!$ viele Blätter.*

Beweis: Wie wir bereits wissen, hat der Entscheidungsbaum mindestens $n!$ viele Blätter. Nehmen wir nun für einen Widerspruchsbeweis an, daß er mehr als $n!$ Blätter hat. Dann kommt eine Permutation an mindestens zwei Blättern vor. Betrachten wir nun den Vorgänger dieser beiden Blätter, der sich auf einem maximalen Level befindet. Nehmen wir an, dort wurde das Paar (x_i, x_j) miteinander verglichen. In dem einen Blatt gilt nun, daß $x_i < x_j$ ist, und in dem anderen Blatt gilt, daß $x_i > x_j$ ist. Da an beiden Blättern dieselbe Permutation steht, werden in einem Blatt die Elemente x_i und x_j nicht richtig angeordnet. Also kann der Sortieralgorithmus nicht korrekt gewesen sein. Dies ist ein Widerspruch zu der Voraussetzung, daß wir nur Entscheidungsbäume von (korrekten) Sortieralgorithmen betrachten. ■

Definieren wir nun die *mittlere Höhe* $\bar{h}(T)$ eines Baumes T:

$$\bar{h}(T) \;=\; \frac{1}{\lambda(T)} \sum_{b \in \mathcal{L}(T)} (\mathrm{Lev}_T(b) - 1),$$

wobei hier mit $\mathrm{Lev}_T(v)$ der Level eines Knoten v im Baum T und mit $\mathcal{L}(T)$ die Menge der Blätter von T bezeichnet wird.

Wegen des vorherigen Lemmas, entspricht die mittlere Höhe eines Entscheidungsbaumes für einen wiederholungsfreien Sortieralgorithmus gerade der mittleren Anzahl von Vergleichen desselben Sortieralgorithmus. Wir müssen nun nur noch zeigen, daß auch die mittlere Höhe eines Baumes mindestens logarithmisch in der Anzahl der Blätter ist.

Lemma 2.32 *Für einen binären Baum T gilt:*

$$\bar{h}(T) \geq \log(\lambda(T)).$$

Beweis: Dies beweisen wir wieder mit vollständiger Induktion über die Höhe des Baumes T.

Induktionsanfang ($h = 0$): Der Baum besteht wieder nur aus der Wurzel, so daß die mittlere Höhe auch diesemal wieder $0 \geq \log(1)$ ist.

Induktionsschritt ($h - 1 \to h$): Als erstes stellen wir fest, daß an der Wurzel des Baumes zwei Teilbäume T_1 und T_2 mit $\lambda(T_1)$ bzw. $\lambda(T_2)$ Blättern hängen, wobei $\lambda(T_1) + \lambda(T_2) = \lambda(T)$ und die Tiefe der Teilbäume selbst maximal $h - 1$ ist. Zunächst bemerken, wir daß der Induktionsschritt trivial wird, wenn einer der beiden Teilbäume leer ist. Also sei im folgenden $\lambda(T_1) \in [1 : \lambda(T) - 1]$. Damit erhalten wir die folgende Abschätzung:

$$\begin{aligned}
\bar{h}(T) &= \frac{1}{\lambda(T)} \sum_{b \in \mathcal{L}(T)} (\mathrm{Lev}_T(b) - 1) \\
&= \frac{1}{\lambda(T)} \left(\sum_{b \in \mathcal{L}(T_1)} (\mathrm{Lev}_{T_1}(b)) + \sum_{b \in \mathcal{L}(T_2)} (\mathrm{Lev}_{T_2}(b)) \right) \\
&= \frac{\lambda(T_1)}{\lambda(T)} \cdot \underbrace{\frac{1}{\lambda(T_1)} \sum_{b \in \mathcal{L}(T_1)} (\mathrm{Lev}_{T_1}(b))}_{=\bar{h}(T_1)+1} + \frac{\lambda(T_2)}{\lambda(T)} \cdot \underbrace{\frac{1}{\lambda(T_2)} \sum_{b \in \mathcal{L}(T_2)} (\mathrm{Lev}_{T_2}(b))}_{=\bar{h}(T_2)+1} \\
&\overset{\text{I.V.}}{\geq} \frac{\lambda(T_1)}{\lambda(T)} \cdot (\log(\lambda(T_1)) + 1) + \frac{\lambda(T_2)}{\lambda(T)} \cdot (\log(\lambda(T_2)) + 1) \\
&= \frac{\lambda(T_1)\log(T_1) + \lambda(T_2)\log(T_2)}{\lambda(T)} + 1 \\
&\ \text{Die Funktion } x\log(x) + (\alpha - x)\log(\alpha - x) \text{ hat ihr Minimum bei } x = \tfrac{\alpha}{2} \\
&\geq \frac{1}{\lambda(T)} \left(\frac{\lambda(T)}{2} \log\left(\frac{\lambda(T)}{2}\right) + \frac{\lambda(T)}{2} \log\left(\frac{\lambda(T)}{2}\right) \right) + 1 \\
&= \log\left(\frac{\lambda(T)}{2}\right) + 1 \\
&= \log(\lambda(T))
\end{aligned}$$

■

Damit erhalten wir das folgende Theorem:

Theorem 2.33 *Jeder vergleichsbasierte Sortieralgorithmus benötigt auf Folgen der Länge n im Mittel mindestens $n\log(n) - n\log(e) + O(n) \approx n\log(n) - 1.44\,n$ Vergleiche.*

2.8 Bucketsort

In diesem Abschnitt wollen wir nun noch einen einfachen Sortieralgorithmus vorstellen, der in vielen Fällen nur lineare Laufzeit hat. Da wir im vorigen Abschnitt eine untere Schranke bewiesen haben, muß dieser Algorithmus eine der Voraussetzungen für den Satz der unteren Schranke verletzen. Der vorgestellte Algorithmus wird *nicht* vergleichsbasiert sein.

2.8.1 Das Universum $[0 : N-1]$

Wir nehmen im folgenden an, daß die zu sortierenden Schlüssel aus dem Intervall $[0 : N-1]$ stammen. Diese Menge aus denen die Schlüssel stammen können, nennt man auch *Universum*.

Zuerst legen wir für alle möglichen Schlüssel je einen leeren Bucket (Eimer) an, also N Stück. Daher leitet sich auch der Name *Bucketsort* ab. Dann gehen wir der Reihe nach durch die zu sortierende Liste, und werfen jeden Datensatz mit Schlüssel $i \in [0 : N-1]$ in seinen entsprechenden Bucket, also den Bucket mit Index i. Zum Schluß gehen wir die Buckets in der aufsteigenden Ordnung von 0 bis $N-1$ durch und geben die Elemente der einzelnen Buckets aus. Wollen wir dabei stabil sortieren, so geben wir für jeden einzelnen Bucket die Datensätze in der Reihenfolge aus, in der wir sie in den Bucket hineingeworfen haben. Offensichtlich ist die ausgegebene Folge aufsteigend sortiert.

Die Laufzeit von Bucketsort beträgt $O(N+n)$, wobei n die Anzahl der zu sortierenden Datensätze ist. Dies sieht man wie folgt. Im ersten Schritt legen wir N leere Buckets an, also Laufzeit $O(N)$. Dann werfen wir jedes Element in einen Bucket.

```
BucketSort (int array[], int len, int N)
{
    list<int> bucket[N];
    for (int i = 0; i < N; i++)
        bucket[i].init();
    for (int i = 0; i < len; i++)
        bucket[array[i]].append(array[i]);
    for (int i = 0, j = 0; i < N; i++)
        while (!(bucket[i].is_empty()))
            array[j++]=bucket[i].remove();
}
```

Abbildung 2.35: Standard Bucketsort

Dies kostet uns Zeit $O(n)$, wenn wir auf jeden Bucket direkt zugreifen können, also z.B. über ein Array. Zum Schluß gehen wir die Buckets durch und geben die Elemente aus, was Laufzeit $O(n + N)$ erfordert. Ist also $N = o(n \log(n))$, so ist Bucketsort effizienter als jeder vergleichsbasierte Sortieralgorithmus.

Theorem 2.34 *Eine Folge von n Zahlen aus dem Bereich* $[0 : N - 1]$ *läßt sich mittels Bucketsort in Zeit* $O(n + N)$ *mit* $O(n + N)$ *zusätzlichem Speicher sortieren.*

In Abbildung 2.35 ist der Algorithmus für Bucketsort angegeben. Hierbei greifen wir zur Realisierung der Buckets auf lineare Listen zurück, um in einem Bucket mehrere Datensätze abspeichern zu können. Hinweise zur Implementierung linearer Listen geben wir in Abschnitt 2.8.4.

2.8.2 Das Universum $[0 : N - 1]^c$

Wenn wir mit Standard Bucketsort etwa tausend Zahlen sortieren möchten, dann sollte der Bereich der zu sortierenden Zahlen deutlich weniger als ca. zehn Tausend betragen.

Diese Einschränkung über die Größe des Universum können wir ein wenig aufbrechen indem wir einen modifizierten Bucketsort mit dem Bereich $[0 : N - 1]^c$ als Universum betrachten. Für $N = 10$ läßt sich dieser Bereich gerade als die Zahlen in Dezimaldarstellung mit Länge c auffassen. Für $N = 256$ kann man jedes einzelne Zeichen als ASCII oder EBCDIC auffassen und es lassen sich dann Zeichenketten der Länge c sortieren.

Dieser *verallgemeinerte Bucketsort* läuft nun wie folgt ab. Zuerst werden wieder N viele leere Buckets mit je einem Index $i \in [0 : N - 1]$ initialisiert. Dann wird im wesentlichen c mal der Standard Bucketsort aufgerufen. In der ersten Runde werden alle Zeichen nach dem $(c-1)$-ten Zeichen des Schlüssels stabil sortiert (die Schlüssel sind dabei von links nach rechts mit 0 bis $c - 1$ indiziert). Die Buckets werden dann (sortiert nach Position $c - 1$ der Schlüssel) zurück in das Ursprungsarray zurückgespeichert (dritter Schritt des Standard Bucketsorts). Diesen Vorgang wiederholen wir nun c mal, wobei wir im i-ten Durchlauf nach der $c-i$-ten Position des Schlüssel sortieren. Der Zeitaufwand des verallgemeinerten Bucketsorts ist nun wegen der c Durchläufe $O(c(n + N))$.

Warum sortiert dieser verallgemeinerte Bucketsort überhaupt? Betrachten wir zwei Schlüssel $x = (x_1, \ldots, x_c) < y = (y_1, \ldots, y_c)$. Ohne Beschränkung der Allgemeinheit sei j die kleinste Position, an der sich x und y unterscheiden, d.h. $x_k = y_k$ für alle $k \in [0 : j-1]$ und $x_j < y_j$. Im $(c-j)$-ten Durchlauf werden gerade die Schlüssel nach

```
GeneralizedBucketSort (char* array[], int len, int N, int c)
{
    list<char*> bucket[N];
    for (int i = 0; i < N; i++)
        bucket[i].init();
    for (int k = c − 1; k ≥ 0; k−−)
    {
        for (int i = 0; i < len; i−−)
            bucket[array[i][k]].append(array[i]);
        for (int i = 0, j = 0; i < N; i−−)
            while (!(bucket[i].is_empty()))
                array[j++]=bucket[i].remove();
    }
}
```

Abbildung 2.36: Verallgemeinerter Bucketsort

den Zeichen an der Position j sortiert. Also kommt x in den Bucket mit Index x_j und y in den Bucket mit Index y_j. Daher kommt x in einen Bucket mit kleinerem Index als Schlüssel y, was nichts anderes bedeutet, als das x in der bislang sortierten Folge vor y auftaucht. In den folgenden Durchläufen von Bucketsort ist nun immer $x_k = y_k$. Also kommen im folgenden x und y immer in denselben Bucket. Da wir stabil sortieren, bleibt x in der sortiereten Folge immer vor y. Da dies für alle Paare $x \neq y$ gilt, folgt, daß der verallgemeinerte Bucketsort korrekt sortiert.

Theorem 2.35 *Eine Folge von n Zahlen aus dem Bereich $[0 : N-1]^c$ läßt sich mit Bucketsort in Zeit $O(c(n+N))$ mit $O(n+N)$ zusätzlichem Speicher sortieren.*

Wenn nun $c = o(\log(n))$ und $N = O(n)$ schlägt der verallgemeinerte Bucketsort jedes vergleichsbasierte Sortierverfahren.

13. Juni 97

In Abbildung 2.36 ist die detaillierte Implementierung des verallgemeinerte Bucketsorts angegeben. Hier nutzen wir aus, daß in C ein Array von Zeichen nichts anderes als ein Zeiger auf ein Zeichen ist. Auch hier verweisen wir für die Details der Implementierung von linearen Listen auf den übernächsten Abschnitt 2.8.4.

2.8.3 Das Universum $[0 : N-1]^*$ (Ü)

Betrachten wir zum Schluß das Problem Zeichenketten beliebiger Länge zu sortieren. Wir können hierzu den verallgemeinerten Bucketsort wie folgt zum sogenannten *erweiterten Bucketsort* ergänzen. Zuerst sortieren wir die Zeichenketten mit dem

```
EXTENDEDBUCKETSORT (CHAR* ARRAY[], INT LEN, INT N)
{
    /* determine longest string */
    int c = 0;
    for (int i = 0; i < len; i++);
        if (strlen(array[i]) >c)
            c=strlen(array[i]);

    /* sort string with respect to their lengths */
    list<char*> bucket_len[c + 1];
    for (int i = 0; i < c + 1; i++)
        bucket_len[i].init();
    for (int i = 0; i < len; i++)
        bucket_len[strlen(array[i])].append[array[i]];

    /* sort strings */
    list<char*> bucket[N];
    for (int i = 0; i < N; i++)
        bucket[i].init();
    int act_len = 0;
    for (int k = c; k ≥ 0; k--)
    {
        /* sort strings with length k */
        int new_elts = 0;
        while (!(bucket_len[k].is_empty()))
        {
            char* elt=bucket_len[k].remove();
            bucket[elt[k]].append(elt);
            new_elts++;
        }
        /* sort string with length > k */
        for (int i = 0; i < act_len; i++)
            bucket[array[i][k]].append(array[i]);
        act_len + = new_elts;
        for (int i = 0, j = 0; i < N; i++)
            while (!(bucket[i].is_empty()))
                array[j++]=bucket[i].remove();
    }
}
```

Abbildung 2.37: Erweiterter Bucketsort

Standard Bucketsort nach ihrer Länge. Bezeichnen wir die maximale Länge einer Zeichenkette mit ℓ^*. Dann verwenden wir den verallgemeinerten Bucketsort, wobei nun $c = \ell^*$ ist. Dabei werden allerdings beim Sortieren nach der i-ten Position nur die Zeichenketten berücksichtigt, deren Länge mindestens i ist. Die Implementierung ist in Abbildung 2.37 zu finden. Für die Analyse bezeichnen wir mit n die Anzahl der Zeichenreihen, mit ℓ_i die Länge der i-ten Zeichenreihe, mit ℓ^* die Länge der längsten Zeichenreihe und mit $\ell = \sum_{i=1}^{n} \ell_i$ die Anzahl aller Zeichen, also die Größe der Eingabe. Der Zeitaufwand von Bucketsort setzt sich aus drei Komponenten zusammen:

Ermittlung der längsten Zeichenfolge: $O(\sum\limits_{i=1}^{n} \ell_i) = O(\ell)$.
Dies folgt daraus, daß für jeden String alle Zeichen durchlaufen werden müssen.

Sortieren nach der Länge: $O(\ell^* + n)$
Dies folgt aus der Tatsache, daß ℓ^* verschiedene Buckets angelegt werden müssen. Der eigentliche Sortiervorgang benötigt Zeit $O(n)$.

Sortieren der Elemente: $O(N + \sum\limits_{j=1}^{\ell^*} |\{i : \ell_i \geq j\}|)$.
Zuerst müssen N verschiedene Buckets angelegt werden, die für jeden der ℓ^* Durchläufe wiederverwendet werden. Im $(\ell^* - j + 1)$-ten Durchlauf sind natürlich nur die Zeichenketten involviert, deren Länge mindestens j ist.

Der letzte Term läßt sich wie folgt umformen:

$$\begin{aligned}
O\left(N + \sum_{j=1}^{\ell^*} |\{i : \ell_i \geq j\}|\right) &= O\left(N + \sum_{j=1}^{\ell^*} \sum_{\substack{i=1 \\ \ell_i \geq j}}^{n} 1\right) \\
&= O\left(N + \sum_{i=1}^{n} \sum_{j=1}^{\ell_i} 1\right) \\
&= O\left(N + \sum_{i=1}^{n} \ell_i\right) \\
&= O\left(N + \ell\right).
\end{aligned}$$

Damit erhalten wir unter Berücksichtigung, daß $\ell^* \leq \ell$ und $n \leq \ell$ ist, sofort das folgende Theorem:

Theorem 2.36 *Eine Folge von n Zeichenreihen aus dem Bereich $[0 : N-1]^*$ läßt sich mit Bucketsort in Zeit $O(N+\ell)$ mit $O(N+\ell^*)$ zusätzlichem Speicher sortieren, wobei ℓ^* die maximale Länge einer Sequenz ist und ℓ die Anzahl aller Zeichen der Eingabe ist.*

```
LINEARE LISTEN
template <class K>
class list_item
{
        friend class list<K>;
        K key;
        list_item *next;
};

template <class K>
class list
{
    protected:
        list_item<K> *first, *last;
    public:
        void init()
        {
            first=last=NULL;
        }
        bool is_empty()
        {
            return (first==NULL);
        }
        void append(K key)
        {
            list_item<K> *ptr = new list_item<K>;
            ptr→key=key;
            ptr→next=NULL;
            if (first==NULL) first=last=ptr;
            else last=last→next=ptr;
        }
        K remove()
        {
            list_item<K> *ptr = first;
            K key=first→key;
            first=first→next;
            delete ptr;
            return key;
        }
}
```

Abbildung 2.38: Rudimentäre Implementierung linearer Listen in C++

2.8.4 Hinweise zur Implementierung (*)

Die einzelnen Buckets werden wir als lineare Listen verwalten, damit auch mehrere Datensätze in einen Bucket passen. Dazu ist in Abbildung 2.38 eine rudimentäre Definition von einfach verketteten Listen für verschiedene Typen gegeben. Es werden nur die benötigten elementarsten Methoden definiert. Die Methode `init` zum Initialisieren einer Liste, die Methode `is_empty` zum Überprüfen, ob eine Liste leer ist, die Methode `append` zum Anhängen eines neuen Elementes an das Ende einer Liste, sowie die Methode `remove`, die das erste Element einer Liste entfernt und zurückgibt.

Wir haben wieder der Einfachheit halber die Datensätze mit den Schlüsseln identifiziert. Im Falle des Standard Bucketsorts haben wir ganze Zahlen und im Falle des verallgemeinerten Bucketsorts Zeichenreihen fester Länge sortiert. Daher wurden in Bucketsort lineare Listen über ganzen Zahlen (`list<int>`) verwendet und im verallgemeinerten Bucketsort lineare Listen über einem Array von Zeichen (`list<char*>`). Dabei haben wir ausgenutzt, daß in C ein Array über einem bestimmeten Datentyp nichts anderes ist als ein Zeiger auf diesen Datentyp. Es sollte klar sein wie die vorgestellten Algorithmen zu modifizieren sind, damit sie mit beliebigen Datensätzen arbeiten.

3 Suchen

3.1 Grundlagen

In diesem Kapitel wollen wir uns mit dem Suchen befassen. Wir unterscheiden hier im wesentlichen zwei verschiedene Arten des Suchens. Zuerst wollen wir uns mit dem Wiederfinden bereits gespeicherter Information bzw. dem Feststellen, ob wir eine bestimmte Information bereits gespeichert haben, befassen (wie z.B. Adreßbücher). Im letzten Abschnitt mit dem in der Praxis oft auftretenden Suchen von Zeichenketten in Texten (z.B. Editor).

Für den ersten Teil nehmen wir hierfür wieder an, daß jeder Datensatz einen Schlüssel beinhaltet, der diesen Datensatz eindeutig charakterisiert, wie z.B. Namen in einem Telefonbuch. Die möglichen Schlüssel stammen meist aus einem sehr großen Universum (z.B. Namen: es gibt mehr als 8 Milliarden Zeichenreihen der Länge höchstens 7 über $\{A, \ldots, Z\}$). Die Anzahl der zu speichernden Datensätze ist dahingegen relativ klein. Für das Universum wollen wir im folgenden die Menge $U = [0 : N-1] \subset \mathbb{N}$ hernehmen, wobei dann also N die Anzahl der Elemente im Universum ist. Die Anzahl, der in unserer Datenstruktur wirklich vorkommenden Schlüssel, wollen wir mit m bezeichnen, wobei immer $m \ll N$ gelten soll.

Wir bezeichnen eine Datenstruktur als *Wörterbuch* (auch engl. *Dictionary*), wenn sie die folgenden drei elementaren Operationen zur Verfügung stellt:

`is_member`: Testet, ob zu einem gegebenen Schlüssel ein Datensatz in dem Wörterbuch enthalten ist, und wenn ja, gibt diesen Datensatz aus.

`insert`: Fügt einen Datensatz in das Wörterbuch ein, sofern der Schlüssel nicht bereits enthalten ist.

`delete`: Löscht einen Datensatz aus dem Wörterbuch, sofern der Schlüssel enthalten ist.

Im folgenden werden wir sehen, wie man Informationen geschickt abspeichern kann, so daß sich die obigen Operationen effizient ausführen lassen.

3.2 Ausnutzen von Sortierung (Ü)

Wie allseits bekannt ist, kann eine Sortierung bei der Implementierung von Wörterbüchern helfen. Alle bekannten Wörterbücher, wie Lexika, Telefonbücher, konkrete Wörterbücher, etc., sind sortierte Listen mit direktem Zugriff, da man jede einzelne Seite direkt aufschlagen kann. Allerdings handelt es sich bei den obigen Beispielen um statische Wörterbücher, in denen die Einfüge- und Lösch-Operationen nicht unterstützt werden.

3.2.1 Lineare Suche

In Abschnitt 2.1.2 auf Seite 24 haben wir gesehen, wie wir eine Sortierung zum Finden und Einfügen eines Elementes mittels linearer Suche ausnützen können. Das Löschen von Elementen kann analog ausgeführt werden.

Theorem 3.1 *Ist ein Wörterbuch der Größe maximal n als sortiertes Array implementiert, so lasen sich die Operationen* `is_member`*,* `insert` *und* `delete` *mittels linearer Suche mit Zeitkomplexität $O(n)$ realisieren.*

Leider ist der Zeitaufwand für praktische Anwendungen viel zu hoch.

3.2.2 Binäre Suche

In Abschnitt 2.1.3 auf Seite 27 haben wir gesehen, wie wir die Anzahl der Vergleiche beim Suchen von linear auf logarithmisch senken konnten, wenn man eine binäre Suche verwendete. Leider bleibt die Anzahl der nötigen Vertauschungen im Array bei den Operationen `insert` und `delete` immer noch linear.

Theorem 3.2 *Ist ein Wörterbuch der Größe maximal n als sortiertes Array implementiert, so lasen sich die benötigen die Operationen* `insert` *und* `delete` *mittels binärer Suche mit Zeitkomplexität $O(n)$ implementieren während sich die Operation* `is_member` *in Zeit $O(\log(n))$ realisieren läßt.*

Der lineare Zeitbedarf für die Operationen `insert` und `delete` ist für praktische Zwecke im allgemeinen nicht brauchbar. Nur wenn man die meist häufig auftretenden `insert` und `delete` Funktionen dann gebündelt ausführen kann, also offline, kann eine solche Implementierung sinnvoll sein.

3.2.3 Exponentielle Suche

Eine Verbesserung der Operation `is_member` läßt sich noch wie folgt erzielen, wenn man die Laufzeit in Abhängigkeit von dem Rang des gesuchten Elements angeben will. Wir werden zeigen, daß man ein Element mit Rang k in einem sortierten Array mit $O(\log(k))$ vielen Vergleichen finden kann. Man beachte, daß man im allgemeinen den Rang des gesuchten Schlüssels vorher nicht kennt.

Dazu werden wir in einer ersten Phase erst einmal den Rang des gesuchten Elementes bis auf den Faktor zwei eingrenzen. Im i-ten Schritt der ersten Phase vergleichen wir das gesuchte Element mit dem 2^i-ten Element des sortierten Arrays (für $i \geq 0$), also mit dem Element an Position $2^i - 1$ im Array. Sobald das Array an Position $2^i - 1$ größer als das gesuchte Element ist, brechen wir die erste Phase ab. Das gesuchte Element hat damit einen Rang im Intervall $k \in [2^{i-1} + 1 : 2^i]$. Man erinnere sich, daß Arrays immer von 0 an indiziert werden. Damit haben wir in der ersten Phase $i + 1 = \lceil \log(k) \rceil + 1$ Vergleiche gemacht.

Nun suchen wir im Intervall $[2^{i-1} + 1 : 2^i - 1]$ mittels der binären Suche weiter. Da das Intervall genau $2^{i-1} - 1$ Elemente umfaßt, benötigen wir nach Lemma 2.3 auf Seite 28 genau $i-1$ Vergleiche. Also genügen uns insgesamt $2i = 2\lceil \log(k) \rceil$ Vergleiche zum Feststellen, ob sich ein Element mit Rang k in einem sortierten Array befindet. Diese Art der Suche wird oft auch als *exponentielle Suche* bezeichnet.

Theorem 3.3 *Ist ein Wörterbuch der Größe maximal n als sortiertes Array implementiert, so lassen sich die benötigten Operationen* `insert` *und* `delete` *mittels linearer Suche mit Zeitkomplexität $O(n)$ implementieren während sich die Operation* `is_member` *in Zeit $O(\log(k))$ realisieren läßt, wenn man ein Element mit Rang k in diesem Wörterbuch sucht.*

3.3 Hashing

Beim Hashing implementieren wir ein Wörterbuch als ein (unsortiertes) Array der Größe $n = O(m)$. Dabei soll die Größe des Arrays nur möglichst wenig größer sein als die Anzahl der abzuspeichernden Datensätze, um so wenig wie möglich Speicherplatz zu verschenken. Meist wird $n \in [m : 2m]$ sein.

3.3.1 Hashfunktionen

Wie können wir nun einem Datensatz in Abhängigkeit von seinem Schlüssel eine Position im Array zuordnen? Dazu wählen wir eine sogenannte *Hashfunktion*, die jedem Schlüssel des Universums $U = [0 : N-1]$ eine Position $\in [0 : n-1]$ im Array zuweist. Im folgenden bezeichnen wir Hashfunktionen mit

$$h : [0 : N-1] \to [0 : n-1].$$

Die prinzipielle Idee ist nun die, daß der in das Wörterbuch aufzunehmende Datensatz mit Schlüssel k an die Stelle $h(k)$ im Array geschrieben wird.

Da $n \ll N$ ist, kann die Hashfunktion leider nicht injektiv sein. Daher werden manchen Datensätzen dieselbe Position im Array zugewiesen. Wenn zwei Datensätze gemäß der Hashfunktion auf dieselbe Stelle des Arrays gehasht werden sollen, so spricht man von einer *Kollision*. Wir werden in den folgenden Abschnitten sehen, wie man solche Kollisionen auflösen kann. Vorweg sollten wir noch erwähnen, daß solche Kollisionen nicht selten sind, was auf den ersten Blick überhaupt nicht klar ist. Man erinnere sich dazu an das Geburtstagsparadoxon, das besagt, daß es in einem Raum mit 23 Personen mit Wahrscheinlichkeit größer als $\frac{1}{2}$ zwei Personen gibt, die am selben Tag Geburtstag haben. Das ist äquivalent dazu, daß es in einer Hashtabelle der Größe 365 mit nur 23 Einträgen bereits mit Wahrscheinlichkeit größer als $\frac{1}{2}$ zu einer Kollision gekommen ist.

Um solche Kollisionen dennoch möglichst gering zu halten, sind wir an Hashfunktionen interessiert, die möglichst gut streuen. Wenn man sich z.B. ein persönliches Adreßbuch anschaut, welches im allgemeinen nach den Anfangsbuchstaben der Nachnamen gehasht wird, so zeigt sich, daß diese Hashfunktion schlecht streut. Es gibt z.B. sehr viele Nachnamen, die mit S beginnen, aber nur wenige Nachnamen, die mit I beginnen. Man muß also bei der Wahl der Hashfunktion sehr umsichtig zu Werke gehen, da man im allgemeinen nicht erwarten kann, daß alle Schlüssel des Universums gleichwahrscheinlich sind.

Eine Hashfunktion heißt *ideal*, wenn sie die folgende Gleichung erfüllt:

$$\forall j \in [0 : n-1] : \sum_{\substack{k \in U \\ h(k)=j}} p(k) = \frac{1}{n},$$

wobei $p(k)$ die Wahrscheinlichkeit eines Schlüssels $k \in U$ angibt, mit der dieser vorkommen kann. Im allgemeinen ist jedoch die Verteilung p nicht bekannt.

Außerdem ist es wichtig, daß sich die Hashfunktion leicht berechnen läßt, d.h. sie sollte möglichst in konstanter Zeit berechenbar sein (bzw. in linearer Zeit im logarithmischen Kostenmodell, welches hier angebrachter ist). Wenn man zuviel Zeit für die Berechnung der Hashfunktion verbraucht, ist ein vernünftiger Gebrauch des Wörterbuches meist nicht mehr möglich.

Wir geben nun einige Beispiele für übliche Wahlen von Hashfunktionen an:

***Teilermethode*:** $h(k) = k \bmod n$, wobei n keine Zweier- oder Zehnerpotenz und auch nicht von der Form $2^k - 1$ sein sollte. Eine gute Wahl für n ist im allgemeinen eine Primzahl, die nicht nahe einer Zweierpotenz ist.

***Multiplikationsmethode*:** Sei $a \in (0,1)$ eine Konstante, dann setze

$$h(k) = \lfloor n \cdot ((k \cdot a) \bmod 1) \rfloor.$$

Eine gute Wahl von a ist z.B. $a = \frac{\sqrt{5}-1}{2} = \frac{1}{\varphi} \approx 0.618\cdots$, wobei $\varphi = \frac{1+\sqrt{5}}{2}$ der goldene Schnitt ist (siehe z.B. D.E. Knuth: The Art of Computer Programming Vol. 3: Sorting and Searching).

Im restlichen Teil dieses Abschnittes wollen wir uns mit den verschiedenen Strategien der *Kollisionsauflösung* beschäftigen. Dazu werden wir natürlich auch noch Bemerkungen machen, wie die einzelnen Funktonen `is_member`, `insert` und `delete` zu realisieren sind und welchen Zeitaufwand sie benötigen. Dabei setzen wir voraus, daß einer `delete` Operation eine erfolgreiche Suche und `insert` Operation eine erfolglose Suche vorausging. Wir werden zur Laufzeit nur die Ergebnisse darstellen. Die Herleitung ist zum Teil nicht einfach, besteht aber meist nur aus technischen wahrscheinlichkeitstheoretischen Berechnungen. Für die Analyse werden wir zugrunde legen, daß ein ideale Hashfunktion verwendet wird.

3.3.2 Hashen durch Verkettung

Hashing durch Verkettung ist wohl die einfachste Möglichkeit von Kollisionsauflösung. Grob gesagt besteht die Idee der Kollisionsauflösung darin, sie nicht aufzulösen. Analog wie in unserem Adreßbuch haben wir für einen Nachnamen genügend Platz, um dort auch mehrere eintragen zu können. Wenn der Platz auf der Seite ausgeht, heften wir einfach eine neue Seite hinzu, die nun neue Einträge aufnehmen kann.

Wie bei Bucketsort ist nun jedes Arrayelement eine lineare Liste, die beliebig viele Elemente aufnehmen kann. Die Realisierungen der Operationen `insert`, `delete` und `is_member` sind offensichtlich.

17. Juni 97

Betrachten wir noch den mittleren Zeitbedarf für eine erfolgreiche und erfolglose Suche. Bezeichne hierfür $\alpha = \frac{m}{n}$ den *Füllfaktor* der Hashtabelle. Bei einer erfolgreichen Suche müssen wir im Mittel in etwa die Hälfte der linearen Liste des gehashten Arrayindex durchgehen, während im erfolglosen Fall immer die ganze Liste durchsucht werden muß. Wir bezeichnen im folgenden mit $\mathcal{S}_{\mathrm{SUC}}^{\mathrm{Strat}}$ bzw. $\mathcal{S}_{\mathrm{UNS}}^{\mathrm{Strat}}$ die Anzahl der Zugriffe auf die Hashtabelle bei einer erfolgreichen bzw. erfolglosen Suche unter Verwendung der Strategie Strat zur Kollisionsauflösung. Diese Zugriffe wollen wir im

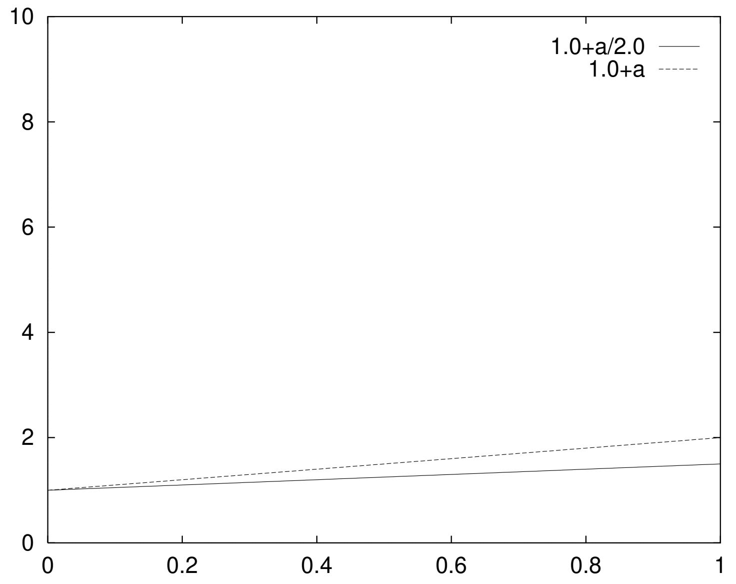

Abbildung 3.1: Anzahl Sondierungen bei erfolgreicher bzw. erfolgloser Suche bei Hashing durch Verkettung

folgenden *Sondierungen* nennen. Im Falle des Hashing durch Verkettung werten wir der Fairneß halber auch jeden Zugriff auf ein Listenelement als Sondierung. Damit läßt sich zeigen, daß:

$$\begin{aligned} \mathcal{S}_{\mathrm{SUC}}^{\mathrm{Chain}} &\approx 1 + \frac{\alpha}{2} \\ \mathcal{S}_{\mathrm{UNS}}^{\mathrm{Chain}} &\approx 1 + \alpha \end{aligned}$$

Diese Resultate sind in Abbildung 3.1 graphisch dargestellt. Somit erhalten wir das folgende Theorem:

Theorem 3.4 *Unter Verwendung der Strategie der Verkettung zur Kollisionsauflösung benötigt die erfolgreiche bzw. erfolglose Suche ungefähr* $1 + \frac{\alpha}{2}$ *bzw.* $1 + \alpha$ *viele Sondierungen, wobei* α *der Füllfaktor der Hashtabelle ist. Die Operationen* `delete` *und* `insert` *können in Zeit* $O(1)$ *realisiert werden.*

Da beim Hashing mit Verkettung die Datensätze immer nur in der Arrayposition abgespeichert werden, die Hashfunktion angibt, spricht man auch von einem *geschlossenen Hashverfahren* oder kurz von *geschlossenem Hashing*.

Wir weisen darauf hin, daß beim Hashing mit Verkettung die Hashtabelle auch überfüllt werden darf, ohne daß bei der erwarteten Anzahl der Sondierungen ein dramatischer Anstieg erfolgt. Da die Einträge allerdings außerhalb der eigentlichen Hashtabelle gespeichert werden, benötigt das Verwalten der linearen Listen zusätzlichen Speicherplatz.

3.3.3 Linear Probing

Beim *Linear Probing* oder auch *linearen Sondieren* wird nun versucht für ein Element, das eine Kollision ausgelöst hat, eine freie Position innerhalb der Hashtabelle zu finden. Deshalb spricht man hier nun auch von einem *offenen Hashverfahren* oder auch kurz *offenem Hashing*. Auch die weiter unten vorgestellten Strategien zur Kollisionsauflösung gehören dem offenen Hashing an.

Dazu führt man eine erweiterte Hashfunktion $h(k,i)$ ein, die angibt an welche Position, das Element k gehasht werden soll, wenn bereits i Versuche mit dieser erweiterten Hashfunktion zu einer Kollision geführt haben. Das bedeutet also, daß man nun der Reihe nach die Positionen $h(k,0)$, $h(k,1)$, $h(k,2)$, $\ldots$ durchprobiert, bis man auf einen freien Eintrag in der Hashtabelle trifft.

Im Falle des linearen Sondierens versucht man nun linear weiter zu suchen, also sieht diese Erweiterung wie folgt aus:

$$h : [0, N-1] \times \mathbb{N}_0 \to [0 : n-1] : \; h(k,i) = (h(k) + i) \bmod n.$$

Für das lineare Sondieren kann man für die Anzahl an Sondierungen zeigen:

$$\begin{aligned} \mathcal{S}_{\mathrm{SUC}}^{\mathrm{LinProb}} &\approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \\ \mathcal{S}_{\mathrm{UNS}}^{\mathrm{LinProb}} &\approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right) \end{aligned}$$

Diese Resultate sind in Abbildung 3.2 graphisch dargestellt.

Es ist klar, daß man (nach einer erfolglosen Suche) immer noch in konstanter Zeit einfügen kann, allerdings wird hier das Löschen komplizierter. Man kann nun ein gefundenes Element x nicht mehr einfach löschen. Bei der Suche nach einem anderen Element y sind wir beim linearen Sondieren eventuell über diese Position hinweggelaufen und haben das Element dahinter eingefügt. Würde man nun einfach das Element x löschen, so würde die Suche nach dem Element y mit dem linearen Sondieren nun an der Stelle, an der x gestanden hat, erfolglos aufhören. Obwohl das Element y noch in der Hashtabelle steht, wird es dann nicht mehr gefunden.

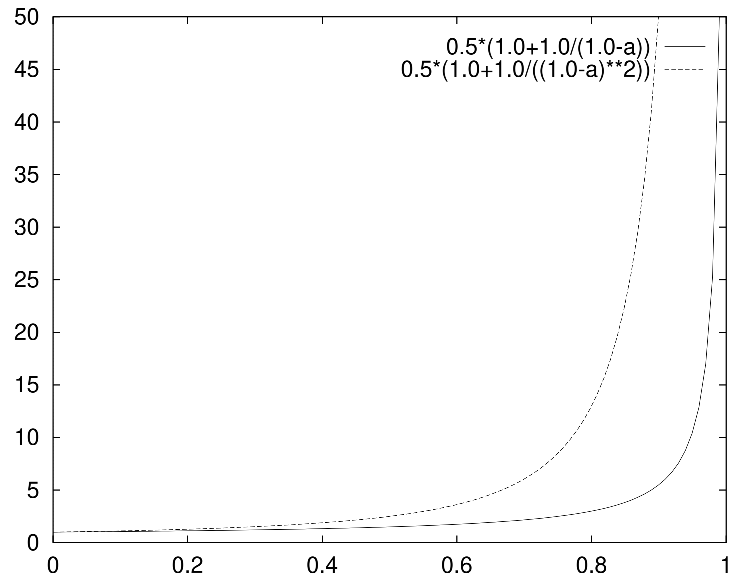

Abbildung 3.2: Anzahl Sondierungen bei erfolgreicher bzw. erfolgloser Suche bei linearem Sondieren

Die Operation `delete` kann daher nur sehr aufwendig unterstützt werden. Eine Möglichkeit ist es, nach jeder `delete` Operation einfach die Hashtabelle völlig neu aufzubauen. Diese Lösung ist allerdings extrem teuer und daher unpraktikabel.

Eine andere Strategie ist, an den Stellen in der Hashtabelle, an denen einmal ein Element gelöscht worden ist, eine Markierung anzubringen. Für eine solche Markierung genügt bereits ein einziges zusätzliches Bit. Wenn nur selten gelöscht wird, ist diese Strategie durchaus praktikabel. Wird hingegen oft gelöscht, so kann jede Position in der Hashtabelle als bereits einmal belegt markiert sein, obwohl die Hashtabelle mittlerweile wieder fast leer ist. Diese gelöschten und nicht wiederbelegten Hashpositionen verlängern die Suchzeiten deutlich. Deshalb ist auch hier von Zeit zu Zeit eine Neukonstruktion der Hashtabelle unumgänglich.

Auf Grund dieser oft unzulänglichen Methoden für das Löschen verzichtet man beim linearen Sondieren und ganz allgemein bei offenen Hashverfahren auf die Unterstützung der Operation `delete`.

Fassen wir den positiven Teil des Ergebnis im folgenden Theorem zusammen.

Theorem 3.5 *Unter Verwendung der Strategie des linearen Sondierens zur Kollisionsauflösung benötigt die erfolgreiche bzw. erfolglose Suche ungefähr* $\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$

bzw. $\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$ *viele Sondierungen, wobei* α *der Füllfaktor der Hashtabelle ist. Die Operation* `insert` *kann in Zeit* $O(1)$ *realisiert werden.*

Ein weiteres Problem beim linearen Sondieren ist, daß eine überfüllte Hashposition auch das Einfügen von Datensätzen in unmittelbar davorliegenden Hashpositionen erschweren kann. Durch das lineare Sondieren bilden sich lange Bereiche von belegten Hashpositionen. Wird nun auf Positionen unmittelbar vor so einem langen belegten Bereich gehasht und dadurch bedingte Kollisionen mit dem linearen Sondieren aufgelöst, so wird auch versucht, diese kollidierten Elemente in diesem bereits belegten Bereich abzulegen. Dadurch werden diese Bereiche immer länger und mehrere solcher Bereiche können auch noch zusammenwachsen. Man nennt diesen Effekt auch *primary Clustering* oder *primäre Häufung*.

Ein weiterer unschöner Effekt ist der des *secondary Clustering* oder der *sekundären Häufung*. Wenn zwei Schlüssel denselben Hashwert haben $h(k,0) = h(k',0)$, dann sind auch die folgenden erweiterten Hashwerte gleich, d.h. $h(k,i) = h(k',i)$. Dies führt wiederum zu einer großen Zahl von Sondierungen, wenn viele Elemente auf dieselbe Position gehasht werden sollen.

3.3.4 Quadratic Probing

Im Falle des *Quadratic Probing* oder *quadratischen Sondierens* versucht man nun 'quadratisch' durch folgende Erweiterung weiter zu suchen:

$$h : [0, N-1] \times \mathbb{N}_0 \to [0 : n-1] : \ h(k,i) = (h(k) - (-1)^i(\lceil i/2 \rceil^2)) \bmod n.$$

Unter Verwendung des quadratischen Sondierens erhalten wir für die Suchzeiten:

$$\begin{aligned} \mathcal{S}_{\mathrm{SUC}}^{\mathrm{QuadProb}} &\approx 1 + \ln\left(\frac{1}{1-\alpha}\right) - \frac{\alpha}{2} \\ \mathcal{S}_{\mathrm{UNS}}^{\mathrm{QuadProb}} &\approx \frac{1}{1-\alpha} - \alpha + \ln\left(\frac{1}{1-\alpha}\right) \end{aligned}$$

Diese Resultate sind in Abbildung 3.3 graphisch dargestellt. Es sei hier angemerkt, daß die primäre Häufung beim quadratischen Sondieren nicht auftritt. Allerdings bleibt beim quadratischen Sondieren immer noch das Problem der sekundären Häufung bestehen.

Theorem 3.6 *Unter Verwendung der Strategie des quadratischen Sondierens zur Kollisionsauflösung benötigt die erfolgreiche bzw. erfolglose Suche in der Hashtabelle ungefähr* $1+\ln\left(\frac{1}{1-\alpha}\right) - \frac{\alpha}{2}$ *bzw.* $\frac{1}{1-\alpha} - \alpha + \ln\left(\frac{1}{1-\alpha}\right))$ *viele Sondierungen, wobei* α *der Füllfaktor der Hashtabelle ist. Die Operation* `insert` *kann in Zeit* $O(1)$ *realisiert werden.*

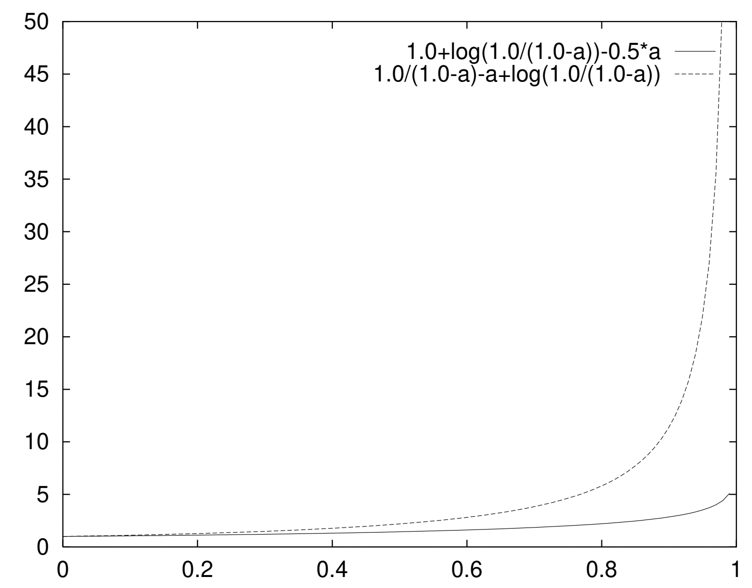

Abbildung 3.3: Anzahl Sondierungen bei erfolgreicher bzw. erfolgloser Suche bei quadratischem Sondieren

Auch hier hat man bei der Operation `delete` dieselben Probleme wie beim linearen Sondieren.

3.3.5 Double Hashing

Im Falle des *Double Hashing* versucht man nun mit einer zweiten, anderen Hashfunktion die Kollisionen aufzulösen. Dadurch erhält man die folgende Erweiterung der Hashfunktion:

$$h : [0, N-1] \times \mathbb{N}_0 \to [0 : n-1] : \; h(k,i) = (h(k) + i \cdot h'(k)) \bmod n,$$

wobei $h'(k)$ für alle k relativ prim zu n sein sollte. Man wähle z.B.

$$h'(k) = 1 + k \bmod (n-2)$$

für ein primes n. Für das Double Hashing erhalten wir für die Suchzeiten:

$$\begin{aligned} \mathcal{S}_{\text{SUC}}^{\text{DblHash}} &\approx \frac{1}{\alpha} \cdot \ln\left(\frac{1}{1-\alpha}\right) \\ \mathcal{S}_{\text{UNS}}^{\text{DblHash}} &\approx \frac{1}{1-\alpha} \end{aligned}$$

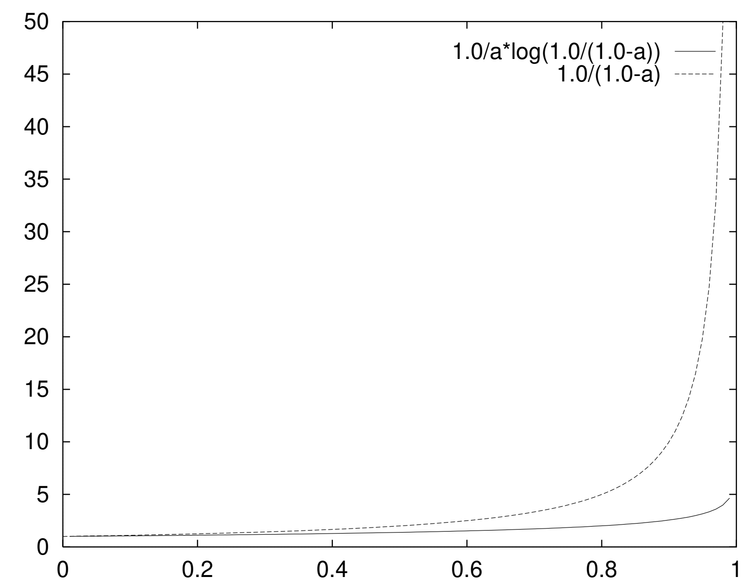

Abbildung 3.4: Anzahl Sondierungen bei erfolgreicher bzw. erfolgloser Suche bei Double Hashing

Diese Resultate sind in Abbildung 3.4 graphisch dargestellt. Zusammenfassend erhalten wir das folgende Theorem:

Theorem 3.7 *Unter Verwendung der Strategie des Double Hashing zur Kollisionsauflösung benötigt die erfolgreiche bzw. erfolglose Suche ungefähr* $\frac{1}{\alpha} \cdot \ln\left(\frac{1}{1-\alpha}\right)$ *bzw.* $\frac{1}{1-\alpha}$ *viele Sondierungen, wobei* α *der Füllfaktor der Hashtabelle ist. Die Operation* `insert` *kann in Zeit* $O(1)$ *realisiert werden.*

Zum Abschluß haben wir noch einmal die Resultate der erfolgreichen bzw. der erfolglosen Suche jeweils miteinander vergleichend in Abbildung 3.5 graphisch dargestellt. Man sieht bei den offenen Hashverfahren sofort, daß diese schlagartig schlechter werden, wenn der Füllfaktor nahe 1 ist. Insbesondere sollte man im Hinblick auf die erfolgreiche bzw. erfolglose Suche bestrebt sein den Füllfaktor einer Hashtabelle bei einem offenen Verfahren unterhalb von ca. 80% zu halten.

Im Gegensatz dazu spielt bei geschlossenen Hashverfahren der Füllfaktor eine untergeordnete Rolle.

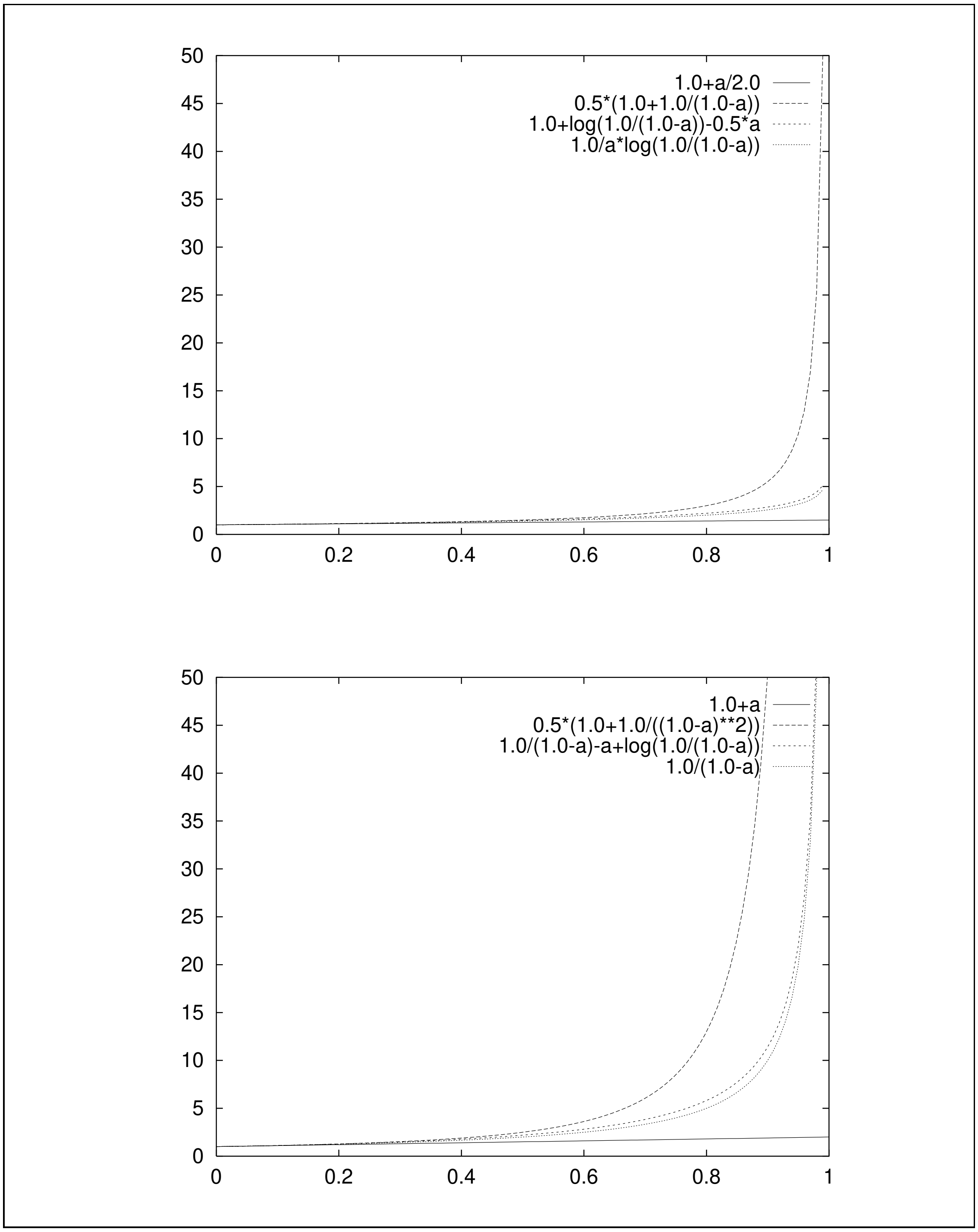

Abbildung 3.5: Vergleich bei erfolgreicher (oben) bzw. erfolgloser Suche (unten) bei den verschiedenen Strategien

3.4 Suchbäume

Eine andere Möglichkeit ein Wörterbuch zu implementieren, ist die Verwendung von Bäumen. Hierbei werden die Datensätze in den Knoten des Baumes abgespeichert. Um einen Datensatz zu finden, nutzt man eine totale Ordnung auf dem gegebenen Universum der Schlüssel aus.

3.4.1 Binäre Suchbäume

Die einfachste Art, ein Wörterbuch mit Hilfe eines Baumes zu implementieren, ist die Verwendung eines binären Baumes. Wir gehen ähnlich wie bei der Implementierung eines Heap vor. Die Datensätze (bzw. besser einen Verweis auf diese) werden in den Knoten des binären Baumes abgespeichert. Um im folgenden die Darstellung zu vereinfachen, werden wir wieder annehmen, daß ein Datensatz nur aus dem Schlüssel besteht. Zudem muß nun jeder Knoten des Baumes die *Suchbaum-Eigenschaft* erfüllen:

> *Der in einem Knoten abgespeicherte Schlüssel ist größer als jeder Schlüssel, der in einem Knoten seines linken Teilbaumes abgespeichert ist, und kleiner als jeder Schlüssel, der in einem Knoten seines rechten Teilbaumes abgespeichert ist.*

Ist in einem binären Baum an jedem Knoten die Suchbaum-Eigenschaft erfüllt, so sprechen wir von einem *binären Suchbaum*.

Damit läßt sich die `is_member` Funktion leicht implementieren. Beginnend an der Wurzel des Baumes vergleicht man den Schlüssel des zu suchenden Elements mit dem in dem betrachteten Knoten gespeicherten Schlüssel. Ist der gesuchte kleiner als der in dem betrachteten Knoten gespeicherte Schlüssel, so sucht man rekursiv im linken Teilbaum weiter; ist er größer, so sucht man rekursiv im rechten Teilbaum weiter. Andernfalls hat man das Element gefunden.

Hat man nun ein gesuchtes Element nicht gefunden, so kann man dieses in den Suchbaum wie folgt einfügen. An dem während des Suchens zuletzt betrachteten Knoten des Baumes hängt man den neuen Schlüssel als linkes oder rechtes Blatt ein, je nachdem, ob der einzufügende Schlüssel kleiner oder größer war.

Wie löscht man nun ein Element? Ist der Schlüssel in einem Blatt gespeichert, so kann man einfach das Blatt entfernen. Ist der Schlüssel in einem Knoten x gespeichert, der genau ein Kind z hat, so entfernt man einfach den Knoten x und macht z zum linken oder rechten Kind des Elters von x, je nachdem ob x das linke oder rechte Kind gewesen ist.

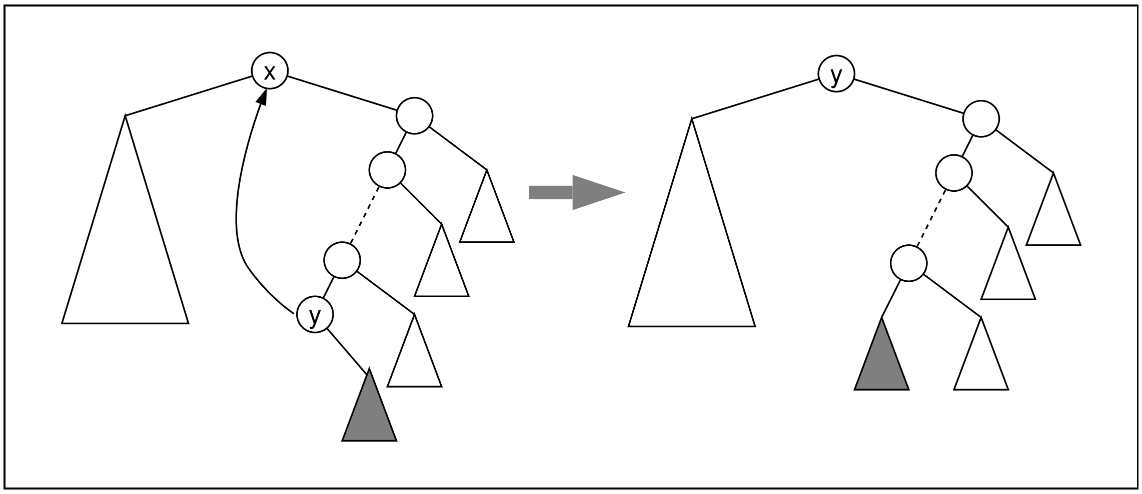

Abbildung 3.6: Löschen eines Knoten mit zwei Kindern in einem binären Suchbaum

Ist das Element in einem inneren Knoten abgespeichert, der zwei Kinder hat, so ist diese Operation etwas komplizierter (siehe zur Illustration dieses Falls auch Abbildung 3.6). Man sucht zuerst im rechten Teilbaum den Knoten mit dem kleinsten Schlüssel und vertauscht diesen Schlüssel mit dem zu löschenden. Wie findet man aber den Knoten mit dem kleinsten Schlüssel in einem Baum? Wir brauchen dazu von der Wurzel immer nur zum linken Kind zu verzweigen bis wir auf einen Knoten gestoßen sind, der kein linkes Kind mehr hat. Aus der Suchbaumeigenschaft folgt, daß in diesem Knoten der kleinste Schlüssel des Baumes abgespeichert ist. Da dieser Knoten nur noch maximal ein Kind hat, können wir diesen Knoten nach der Vertauschung der Schlüssel leicht löschen.

Es bleibt die Frage, ob durch die Vertauschung die Suchbaumeigenschaft erfüllt bleibt. Der neue Schlüssel in der Wurzel des Teilbaumes ist ja das kleinste Element, das im rechten Teilbaum abgespeichert war. Damit ist es auch kleiner als alle Schlüssel, die nun noch im rechten Teilbaum abgespeichert sind. Andererseits war dieses Element größer als das zu löschende Element, das seinerseits größer war als alle Schlüssel im linken Teilbaum.

Theorem 3.8 *Sei T ein binärer Suchbaum. Die Operationen* `is_member`*,* `delete` *und* `insert` *benötigen maximal $O(h(T))$ viele Schritte.*

Leider kann die Höhe eines binären Suchbaumes mit n Elementen $O(n)$ erreichen. Man kann allerdings zeigen, daß für eine gegebene Menge von Schlüsseln, der binäre Suchbaum im Mittel Höhe $O(\log(n))$ hat, wenn man annimmt, daß für die Reihenfolge beim Einfügen alle Permutationen dieser Menge gleichwahrscheinlich sind.

BINÄRER SUCHBAUM

```
struct node
{
    int key;
    struct data *info;
    struct node *left, *right;
}

struct data* search(struct node* n, int key)
{
    if (n==NULL)
        return NULL;
    if (key==n→key)
        return n→info;
    if (key<n→key)
        return search(n→left, key);
    else
        return search(n→right, key);
}

insert(struct node** n, int key, struct data* info)
{
    if (*n==NULL)
    {
        *n=malloc(sizeof(node));
        (*n)→key=key;
        (*n)→info=info;
        (*n)→left=(*n)→right=NULL;
    }
    else
        if (key==(*n)→key)
            /* Error: Key is already inserted */ ;
        else
            if (key<(*n)→key)
                insert(&((*n)→left),key);
            else
                insert(&((*n)→right),key);
}
```

Abbildung 3.7: Eine rudimentäre Realisierung von binären Suchbäumen: Suchen und Einfügen

```
BINÄRER SUCHBAUM
delete(struct node** n, int key)
{
    if (*n==NULL)
        /* Error: Key not found */ ;
    else
        if (key==(*n)→key)
            if ((*n)→right==NULL)
            {
                struct node* r = *n;
                (*n)=(*n)→left;
                free(r→info);
                free(r);
            }
            else
            {
                struct node *r=delete_min(&((n*)→right));
                free((*n)→info);
                (*n)→key=r→key;
                (*n)→info=r→info;
                free(r);
            }
        else
            if (key<(*n)→key)
                delete(&((*n)→left),key);
            else
                delete(&((*n→right),key);
}

struct node* delete_min(node** n)
{
    if ((*n)→left==NULL)
    {
        struct node *r = *n;
        (*n)=(*n)→right;
        return r;
    }
    else
        delete_min(&((*n)→left));
}
```

Abbildung 3.8: Eine rudimentäre Realisierung von binären Suchbäumen: Löschen

3.4.2 AVL-Bäume

Wie wir gesehen haben, sind binäre Suchbäume genau dann sehr effizient, wenn man die Höhe der Bäume beschränken kann. Eine wichtige Teilklasse der binären Suchbäume sind die AVL-Bäume, benannt nach ihren Erfindern G. Adelson-Velskii und Y. Landis (1962). Ein Knoten ist genau dann *höhenbalanciert*, wenn sich die Höhen der an seinen Kindern hängenden Teilbäume um maximal eins unterscheiden. Einen binären Suchbaum nennen wir einen *AVL-Baum*, wenn jeder Knoten des Baumes höhenbalanciert ist.

Als erstes werden wir zeigen, daß der Begriff höhenbalanciert gerechtfertigt ist.

Theorem 3.9 *Ein AVL-Baum der Höhe h hat mindestens $f_{h+3} - 1$ und höchstens $2^{h+1} - 1$ viele Knoten, wobei f_n die n-te Fibonacci-Zahl ist.*

Beweis: Die obere Schranke ist klar, da jeder AVL-Baum auch ein binärer Baum ist und jeder binäre Baum auf Level ℓ maximal $2^{\ell-1}$ viele Knoten haben kann. Also gilt für die maximale Anzahl an Knoten in einem binären Baum der Höhe h:

$$\sum_{\ell=1}^{h+1} 2^{\ell-1} = \sum_{\ell=0}^{h} 2^{\ell} = 2^{h+1} - 1.$$

Zum Beweis der unteren Schranke definieren wir die sogenannten *Fibonacci-Bäume* wie folgt: F_2 ist der Baum der aus einem Knoten besteht. F_3 ist der Baum der aus zwei Knoten besteht, der Wurzel und seinem linken Kind. Für $n > 4$ ist F_n definiert als der Baum, an dessen Wurzel als linker Teilbaum ein F_{n-1} und als rechter Teilbaum ein F_{n-2} hängt. Offensichtlich ist die Höhe von F_n gerade $n-2$. Wir werden durch vollständige Induktion zeigen, daß bis auf Isomorphie der F_{h+2} der kleinste AVL-Baum mit Höhe h ist.

Induktionsanfang ($h \in \{0, 1\}$): Bis auf Isomorphie gibt es genau je einen binären Baum der Höhe 0 bzw. 1, die gerade dem F_2 und dem F_3 entsprechen.

Induktionsschritt ($h - 1 \to h$): Betrachten wir einen minimalen AVL-Baum T der Höhe h. Da die Wurzel höhenbalanciert ist, hängt an der Wurzel ein Teilbaum der Höhe $h-1$ und ein Teilbaum der Höhe $h-2$ oder $h-1$. Da ein minimaler AVL-Baum der Höhe $h-2$ offensichtlich weniger Knoten als AVL-Baum der Höhe $h-1$ hat, folgt, daß der zweite Teilbaum Höhe $h-2$ haben muß. Nach Induktionsvoraussetzung ist nun der minimale AVL-Baum der Höhe $h-1$ bzw. $h-2$ isomorph zu F_{h+1} bzw. F_h. Also muß T isomorph zu F_{h+2}, andernfalls wäre T nicht minimal gewesen. □

Es bleibt zu zeigen, daß die Anzahl der Knoten in F_{h+2} gerade $f_{h+3} - 1$ ist. Nach Definition gilt

$$|F_h| = |F_{h-1}| + |F_{h-2}| + 1 \quad \text{für } n > 3 \qquad \wedge \qquad |F_2| = 1, |F_3| = 2.$$

Eine solche Rekursionsgleichung haben wir bereits in Abschnitt 1.2 behandelt. Mit $g_i := |F_{i-1}| + 1$ gilt

$$g_i = g_{i-1} + g_{i-2} \quad \text{für } n > 2 \qquad \wedge \qquad g_3 = 2, \quad g_4 = 3.$$

Nimmt man noch $g_1 = g_2 = 1$ hinzu, so erhält man genau die Fibonacci-Zahlen $g_i = f_i$. Also ist $|F_{h+2}| = f_{h+3} - 1$. ■

Vielleicht wundert man sich an dieser Stelle, daß die F_i in Abhängigkeit von i etwas merkwürdig definiert sind. Hängt man in einem F_i an jeden Knoten mit genau einem Kind noch ein Blatt an und an jeden Knoten mit keinem Kind genau zwei Blätter, so hat jeder dermaßen modifizierte Fibonacci-Baum F_i genau f_i Blätter. In der Literatur werden oft diese erweiterten Bäume als *Fibonacci-Bäume* bezeichnet.

Zusammen mit Lemma 1.3 auf Seite 3 erhalten wir sofort das folgende Korollar.

Korollar 3.10 *Die Höhe eines AVL-Baumes mit n Knoten ist* $\Theta(\log(n))$.

Damit wird der Begriff höhenbalanciert klar: Jeder AVL-Baum hat eine logarithmische Tiefe. Dies führt unmittelbar zu dem folgenden Theorem:

Theorem 3.11 *In einem AVL-Baum mit n Knoten kann in Zeit* $O(\log(n))$ *festgestellt werden, ob sich ein gegebener Schlüssel im Baum befindet oder nicht.*

20. Juni 97

3.4.3 Einfügen in einen AVL-Baum

Die Suchprozedur in einem AVL-Baum ist dieselbe wie für normale binäre Suchbäume. Nur beim Einfügen oder Löschen von Elementen müssen wir uns mehr Gedanken machen, da hierbei mit der vorigen Methode die AVL-Eigenschaft verletzt werden kann. Um eine Verletzung der Höhenbalancierung leicht feststellen zu können, merken wir uns an jedem Knoten noch die Tiefe des Teilbaumes, der an diesem Knoten gewurzelt ist.

Betrachten wir zunächst das Einfügen eines neuen Elementes. Wie bei gewöhnlichen binären Suchbäumen hängen wir an den nach einer erfolglosen Suche zuletzt betrachteten Knoten einfach ein neues Blatt mit dem neuen Schlüssel an. Hat ein eingefügtes Blatt bereits ein Geschwister, dann wird an keinem Knoten des Baumes die Höhenbalancierung verletzt, da der einzige Teilbaum, dessen Höhe sich ändert, nur aus dem eingefügten Blatt besteht. Also kann nur bei Anhängen eines Blattes an ein Blatt die Höhenbalancierung des Baumes verlorengehen.

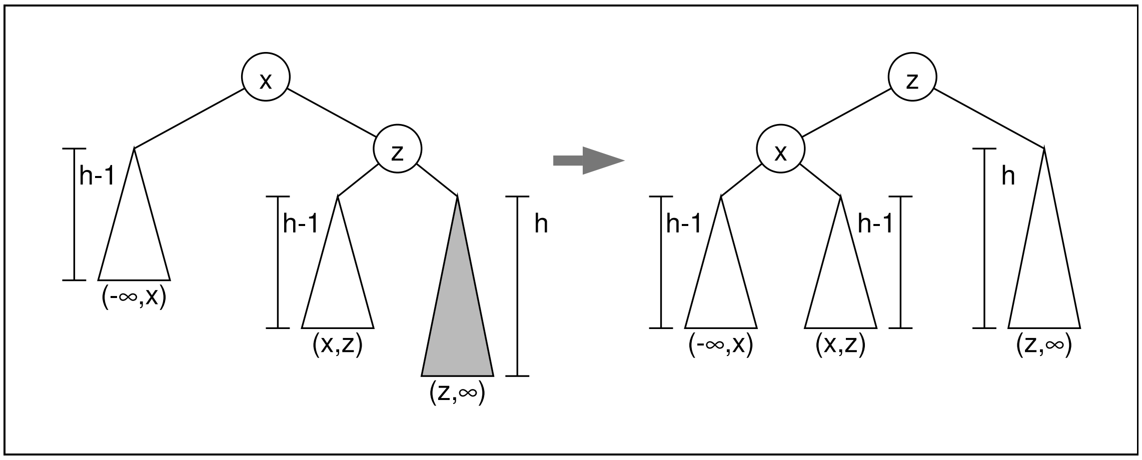

Abbildung 3.9: Einfache Rotation zur Rebalancierung beim Einfügen in einen AVL-Baum

Wir beschreiben nun eine Prozedur, mit der man eine Verletzung der Höhenbalancierung heilen kann, wenn diese Verletzung durch Einfügen eines Blattes erfolgt ist. Den Vorgang der Wiederherstellung der Höhenbalancierung bezeichnen wir im folgenden kurz als *Rebalancierung*. Für die Prozedur zur Rebalancierung nehmen wir im folgenden immer ohne Beschränkung der Allgemeinheit an, daß an dem betrachteten Knoten immer der rechte Teilbaum in der Höhe gewachsen ist. Außerdem nehmen wir an, daß beim Wachsen des rechten Teilbaumes nur einer seiner beiden Teilbäume dafür verantwortlich ist, d.h. der rechte Teilbaum besteht aus zwei unterschiedlich hohen Teilbäumen. Wenn wir die Prozedur an dem Elter des eingefügten Blattes aufrufen stimmen die Voraussetzungen. Wir werden im folgenden zeigen, daß die letzte Voraussetzung beim Aufruf der Prozedur zur Rebalancierung immer erfüllt sein wird.

Sei also im folgenden x der Knoten an dem die Prozedur zur Rebalancierung aufgerufen wird, weil sein rechter Teilbaum mit Wurzel z in der Höhe um eins gewachsen ist. Am Anfang ist z das eingefügte Blatt.

Hatte vor dem Einfügen der linke Teilbaum von x Höhe $h{+}1$ und der rechte Teilbaum Höhe h gehabt, so bleibt der Knoten x balanciert. Desweiteren bleibt die Höhe des Teilbaumes gewurzelt an x unverändert. Wir können also mit der Rebalancierung aufhören.

Hatten beide Teilbäume vorher Höhe h so bleibt Knoten x höhenbalanciert, allerdings ist die Höhe des Baumes gewurzelt an x nun um eins gewachsen. Wir rufen nun die Prozedur zur Rebalancierung am Elter von x auf. Man beachte, daß genau ein Teilbaum von x für das Hohenwachstum verantwortlich ist.

Hat der linke Teilbaum Höhe $h{-}1$ und der rechte Teilbaum bereits vor dem Einfügen Höhe h, so hat der rechte Teilbaum nun Höhe $h+1$, da wir die Prozedur Rebalan-

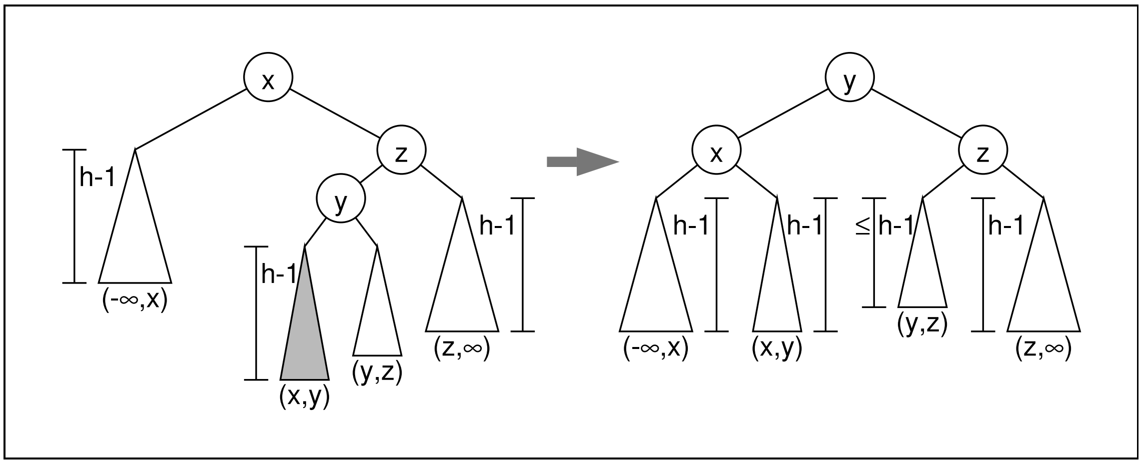

Abbildung 3.10: Doppelrotation zur Rebalancierung beim Einfügen in einen AVL-Baum

cierung an x nur aufrufen, wenn sich durch das Einfügen die Höhe geändert hat. Es gilt nun zwei Fälle zu unterscheiden, je nachdem ob der linke oder rechte Teilbaum von z gewachsen ist. Betrachten wir zuerst den Fall, daß der rechte Teilbaum von z gewachsen ist, also nun Höhe h hat. Dann muß der linke Teilbaum von z nach wie vor Höhe $h-1$ haben. Dieser Fall ist in Abbildung 3.9 dargestellt, wobei der graue Baum den Teilbaum darstellt, der in der Höhe gewachsen ist. Durch eine sogenannte *Rotation* (oft auch als *einfache Rotation* bezeichnet) können wir den Baum, wie in Abbildung 3.9 angegeben, umbauen. Dabei wird der Knoten z zur neuen Wurzel. Da nun der Teilbaum mit der neuen Wurzel z Höhe $h+1$ hat, genau wie der alte Teilbaum mit Wurzel x vor dem Einfügen (da hatte der graue Teilbaum noch Höhe $h-1$!), kann die Prozedur Rebalancierung erfolgreich abgeschlossen werden.

Bleibt nun noch der Fall, daß beim Einfügen der linke Teilbaum von z in der Höhe gewachsen ist. Dies ist in Abbildung 3.10 dargestellt, wobei ebenfalls wieder der in der Höhe gewachsene Baum grau dargestellt ist. Durch eine sogenannte *Doppelrotation* werden die Bäume, wie in Abbildung 3.10 angegeben, umgehängt. Zuerst stellt man fest, daß alle Knoten x, y und z nun höhenbalanciert sind. Die Höhe des Baumes mit der alten Wurzel x hatte vor dem Einfügen Höhe $h+1$. Nach der Doppelrotation ist die Höhe des Teilbaumes mit neuer Wurzel ebenfalls wieder $h+1$. Also wird die Prozedur Rebalancierung erfolgreich abgeschlossen.

Es bleibt noch zu zeigen, daß der Baum nach den Rebalancierungen noch die Suchbaumeigenschaft erfüllt. Dies kann man allerdings leicht den Abbildung 3.9 und 3.10 entnehmen, wobei dort unterhalb jedes Teilbaumes das Intervall der möglichen Schlüssel angegeben ist.

24. Juni 97

Zusammen mit Korollar 3.10 folgt:

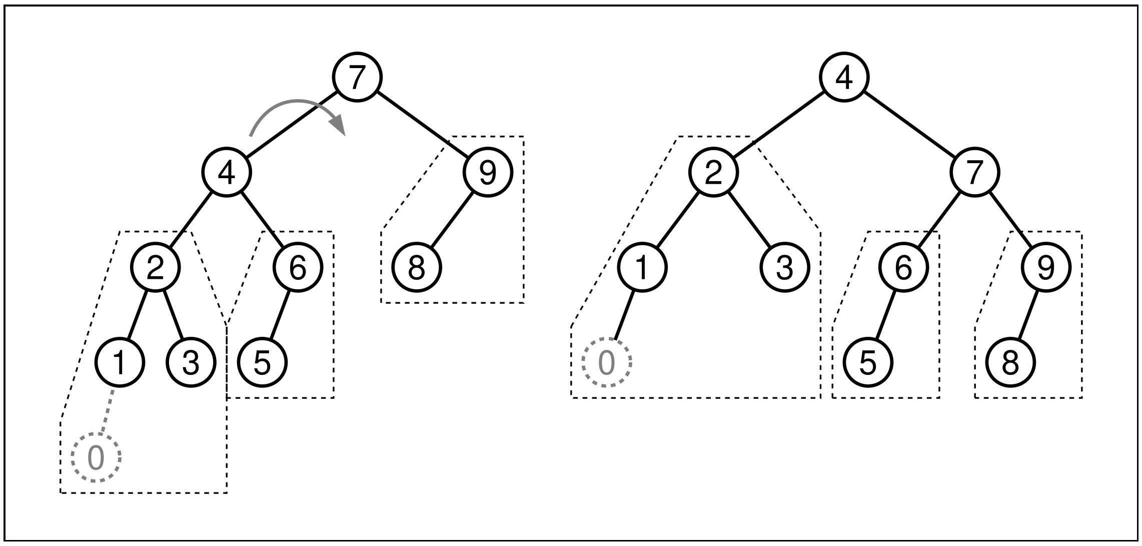

Abbildung 3.11: Ein Beispiel: Einfügen des Schlüssels 0 in einen AVL-Baum mittels einfacher Rotation

Theorem 3.12 *In einem AVL-Baum mit n Knoten kann ein neuer Schlüssel in Zeit $O(\log(n))$ eingefügt werden.*

In den Abbildungen 3.11 und 3.12 ist das Einfügen und die einfache bzw. Doppelrotation noch einmal an einem konkreten Beispiel illustriert.

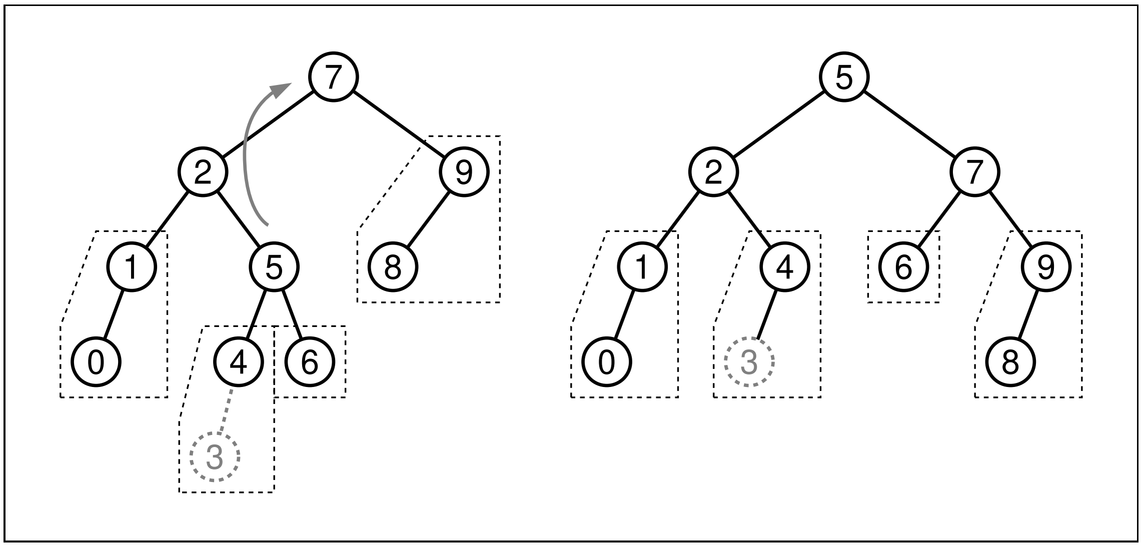

Abbildung 3.12: Ein Beispiel: Einfügen des Schlüssels 3 in einen AVL-Baum mittels Doppelrotation

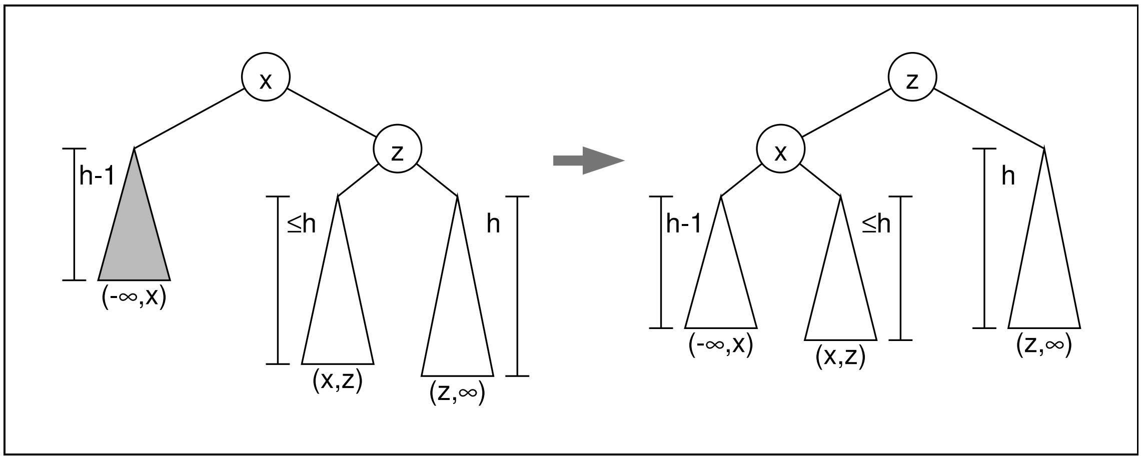

Abbildung 3.13: Einfache Rotation zur Rebalancierung beim Löschen in einem AVL-Baum

3.4.4 Löschen im AVL-Baum (Ü)

Auch das Löschen von Elementen verläuft zunächst wie bei gewöhnlichen Suchbäumen. Auch hier werden, nach einer eventuellen Vertauschung, immer nur Knoten mit maximal einem Kind gelöscht. Wie beim Einfügen muß man nun darauf achten, daß sich beim Löschen eines Blattes die Höhe von einigen Teilbäumen ändern kann. Beim Löschen kann die Höhe eines Teilbaumes sinken und damit kann es passieren, daß einige Knoten nicht mehr höhenbalanciert sind. Wir werden auch hier wieder eine Prozedur zur Rebalancierung angeben, die an den Knoten aufgerufen wird, von denen genau einer der Teilbäume in der Höhe geschrumpft ist. Diese Prozedur wird zuerst am Elter des gelöschten Knotens aufgerufen.

Betrachten wir nun einen Knoten x an dem ohne Beschränkung der Allgemeinheit der linke Teilbaum um eins in der Höhe geschrumpft ist. Sei h die Höhe des linken Teilbaumes vor dem Löschen gewesen, also ist die Höhe dieses Teilbaumes jetzt nur noch $h-1$. Hat der rechte Teilbaum Höhe $h-1$ so ist der Knoten x höhenbalanciert, allerdings ist die Höhe des an x gewurzelten Teilbaumes um 1 gesunken, so daß wir die Prozedur zur Rebalancierung am Elter von x aufrufen müssen.

War die Höhe des rechten Teilbaumes h, so ist x weiterhin höhenbalanciert und die Höhe des Teilbaumes mit Wurzel x hat sich nicht geändert. In diesem Fall sind wir also fertig.

Es bleibt noch der Fall, daß der rechte Teilbaum Höhe $h+1$ hat. Nun ist der Knoten x nicht mehr höhenbalanciert. Sei im folgenden z das rechte Kind von x. Nehmen wir zunächst an, daß der rechte Teilbaum von z Höhe h hat. Der linke Teilbaum von z hat dann entweder Höhe $h-1$ oder auch Höhe h. Dann können wir wieder durch eine Rotation, wie in Abbildung 3.13 angegeben, den Baum umstrukturieren.

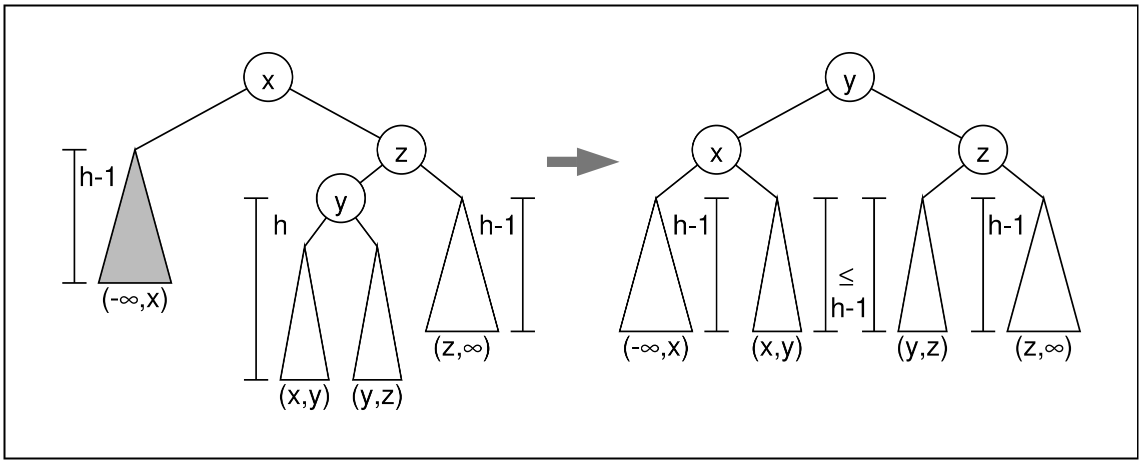

Abbildung 3.14: Doppelrotation zur Rebalancierung beim Löschen in einem AVL-Baum

Wie man ebenfalls der Abbildung entnimmt, ist der Baum nun an den Knoten x und z höhenbalanciert. Darüber hinaus ist die Höhe des Teilbaumes mit neuer Wurzel z nun $h+1$ oder $h+2$, je nachdem ob der alte linke Teilbaum des Knotens z Höhe $h-1$ oder h gehabt hat. Im ersten Fall hat sich die Höhe des Teilbaumes um eins erniedrigt, so daß wir nun die Prozedur zur Rebalancierung noch einmal am Elter von z aufrufen müssen. Andernfalls bleibt die Höhe des Teilbaumes mit neuer Wurzel z unverändert, und wir sind fertig.

Es bleibt der Fall zu betrachten, in dem der rechte Teilbaum von z Höhe $h-1$ hat. Sei nun y das linke Kind von z. Mindestens einer der Teilbäume mit Wurzel y muß Höhe $h-1$ haben, ansonsten hätte der Teilbaum mit Wurzel z nicht Höhe h; der andere Teilbaum hat dann entweder Höhe $h-1$ oder $h-2$. Wiederum können wir den Baum mit einer Doppelrotation, wie in Abbildung 3.14 angegeben, rebalancieren. Aus der Abbildung entnimmt man, daß nun die Knoten x, y und z höhenbalanciert sind, und daß die Höhe des Teilbaumes mit neuer Wurzel y nun genau $h+1$ ist. Da sich nun die Höhe des gesamten Teilbaumes um eins gesenkt hat, müssen wir die Prozedur zur Rebalancierung am Elter von y aufrufen.

Wie im Falle des Einfügens, bleibt noch zu überprüfen, ob die Suchbaumeigenschaft erfüllt bleibt. Dies folgt aber gerade wieder aus den gewählten Rotationen, was leicht an Hand von Abbildung 3.14 verifiziert werden kann.

Zusammen mit Korollar 3.10 folgt:

Theorem 3.13 *In einem AVL-Baum mit n Knoten kann ein bereits im Baum enthaltener Schlüssel in Zeit $O(\log(n))$ gelöscht werden.*

3.4.5 (a, b)-Bäume

Im letzten Abschnitt haben wir gesehen, wie man die Level der Blätter eines Baumes auf $O(\log(n))$ beschränken kann. Nun wollen wir eine Methode vorstellen, wie man dafür sorgen kann, daß sich alle Blätter auf demselben Level $\leq \log(n) + 2$ befinden. Dafür erlauben wir, daß wir in einem Knoten mehr als eine Information abspeichern. In diesem Modell werden alle Daten in den Blättern gespeichert. Die inneren Knoten enthalten im wesentlichen nur solche Informationen, die uns mitteilen, in welchem Teilbaum ein bestimmter Schlüssel zu finden ist.

Im folgenden betrachten wir einen beliebigen Baum, in dem die Datensätze in den Blättern abgespeichert sind, und zwar ein Datensatz pro Blatt. Hat ein innerer Knoten k Kinder, so beinhaltet der innere Knoten $k - 1$ Schlüssel. Die Schlüssel, die im i-ten Teilbaum von links abgespeichert sind, sind alle kleiner als die Schlüssel, die im $(i + 1)$-ten Teilbaum von links abgespeichert sind. Der Schlüssel, der an der i-ten Position im inneren Knoten abgespeichert ist, ist zum einen mindestens so groß wie jeder Schlüssel, der im i-ten Teilbaum abgespeichert ist, und zum anderen kleiner als jeder Schlüssel, der im $(i+1)$-ten Teilbaum abgespeichert ist. An Hand der Informationen über die Schlüssel, die in den inneren Knoten abgespeichert sind, und der Eigenschaft, daß die Schlüssel monoton aufsteigend in den Blättern von links nach rechts abgespeichert sind, können wir beim Suchen entscheiden, in welchem der k Teilbäume wir den zu suchenden Schlüssel finden können.

Ein solcher Suchbaum heißt (a, b)-*Baum*, wenn folgende Eigenschaften erfüllt sind:

- alle Blätter befinden sich auf demselben Level,
- jeder innere Knoten hat maximal b Kinder,
- jeder innere Knoten, mit Ausnahme der Wurzel, hat mindestens a Kinder,
- sowie $b \geq 2a - 1$ und $a \geq 2$.

Daß für beliebige n überhaupt (a, b)-Bäume mit n Blättern existieren, werden wir noch mit Hilfe der Einfüge-Prozedur zeigen. Auf jeden Fall können wir das folgende Lemma festhalten (Beweis $\Rightarrow$ Übungen).

Lemma 3.14 *Ein (a, b)-Baum mit n Blättern hat mindestens Höhe $\lceil \log_b(n) \rceil$ und höchstens Höhe $\lceil \log_a(n) \rceil + 1$.*

Damit ist auch die Zeit zum Suchen logarithmisch in der Anzahl der abgespeicherten Daten.

Wir wollen hier noch festhalten, daß das Auffinden des richtigen Teilbaumes, in dem man weitersuchen muß, etwas mühsam ist. Mit Hilfe der binären Suche läßt sich aber auch dies recht effizient gestalten.

Warum betrachtet man überhaupt solche Bäume, in denen relativ viele Schlüssel in einem inneren Knoten abgespeichert sind. Die Motivation kommt aus dem Bereich der Datenbanken, in denen Datenmengen schnell so groß werden, daß man sie nicht mehr im Hauptspeicher halten kann. Also speichert man diese Daten auf Sekundärspeicher, wie Festplatten bzw. Festplattenarrays ab. Ein Zugriff auf einen solchen sekundären Speicher ist meist recht zeitintensiv. Darüber hinaus ist es ineffizient einzelne Bytes aus einem Sekundärspeicher zu lesen. Auf Sekundärspeichern sind die Daten meistens in großen Blöcken (mehrere kBytes) organisiert, und bei einer Anfrage wird gleich der gesamte Block in den Hauptspeicher kopiert. Daher ist es sinnvoll auch gleich einen ganzen solchen Block auszuwerten. Organisiert man nun einen (a, b)-Baum so, daß ein Knoten in solch einem ganzen Block abgespeichert wird, kann man die Anzahl der Zugriffe auf den Sekundärspeicher recht gering halten. Die Zeit die für die binäre Suche benötigt wird, ist relativ klein zu Zeitaufwand einen solchen Block einzulesen bzw. zurückzuspeichern. Damit die Speicherauslastung möglichst gut ist versucht man möglichst viele Daten in einem Block unterzubringen, also wählt man a so, daß $b + 1 = 2a$ gilt (in diesem Fall erhält man die sogenannten B- oder Bayer-Bäume, siehe Abschnitt 3.4.8). Eine genaue Wahl der Parameter a und b ist allerding diffizil und hängt auch noch von anderen Größen als der Blockgröße ab (siehe auch Abschnitt 3.4.8).

3.4.6 Einfügen in (a, b)-Bäume

Überlegen wir uns nun, wie man ein neues Element in einen (a, b)-Baum einfügen

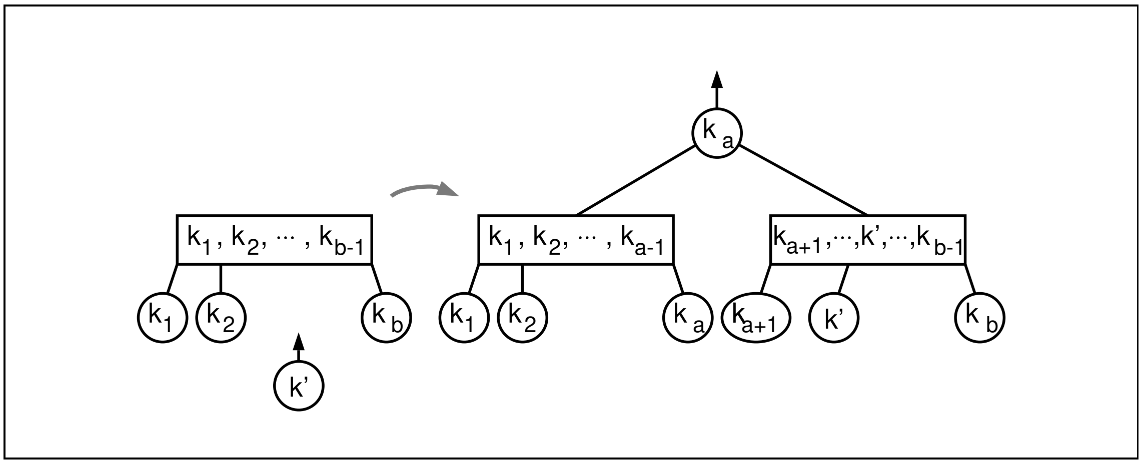

Abbildung 3.15: Einfügen in einen (a, b)-Baum mit Knotenaufteilung

kann. Wie bei normalen Suchbäumen geht dem Einfügen eine erfolglose Suche voraus. Nach einer erfolglosen Suche landen wir in einem Blatt z des (a, b)-Baumes. Den neuen Schlüssel k' fügen wir als ein neues Blatt an den Elter des Blattes z ein. Hat nun der Elter nach dem Einfügen weiterhin maximal b Kinder, so sind wir fertig. Andernfalls hat der Elter genau $b+1 \geq 2a$ viele Kinder. In diesem Falle teilen wir den Elter in zwei Knoten mit jeweils mindestens a Kindern auf. Damit hat sich nun am Großelter die Anzahl der Kinder erhöht. Nun führen wir Prozedur rekursiv weiter. Entweder bricht die Prozedur irgendwann von selbst ab, oder wir kommen an der Wurzel an, und die Höhe des Gesamtbaumes wächst. In Abbildung 3.15 ist dieses Vorgehen schematisch dargestellt. In Abbildung 3.16 sind die wesentlichen Schritte des Einfügens noch einmal an einem konkreten Beispiel illustriert. Halten wir das Ergebnis noch als Lemma fest:

Lemma 3.15 *In einem (a, b)-Baum kann ein neues Element in $O(\log_a(n))$ Schritten eingefügt werden.*

3.4.7 Löschen in (a, b)-Bäumen

Wie können wir nun ein Element löschen. Nach einer erfolgreichen Suche haben wir ein Blatt gefunden. Dieses löschen wir aus seinem Elter x heraus. Hat nun der Elter x immer noch mindestens a Kinder, so sind wir fertig. Andernfalls betrachten wir ein Geschwister y des Elters x und verschmelzen die beiden Knoten zu einem neuen Knoten. Hat dieser neue Knoten nun mindestens $b+1 \geq 2a$ viele Kinder, so spalten wir diesen Knoten wieder in zwei Knoten mit je mindestens a vielen Kindern auf. In diesem Fall sind wir fertig. Hat der verschmolzene Knoten nur maximal b viele Kinder (allerdings hat er mindestens $2a-1 \geq a$ viele Kinder), so ist am Großelter (dem Elter von x und y) die Anzahl der Kinder um eins gesunken und wir führen die Prozedur rekursiv am Großelter fort.

Lemma 3.16 *In einem (a, b)-Baum kann ein neues Element in $O(\log_a(n))$ Schritten gelöscht werden.*

Damit haben wir folgendes Theorem gezeigt, wenn wir a und b als konstant annehmen.

Theorem 3.17 *Ein Wörterbuch der Größe n läßt sich mit Hilfe eines (a, b)-Baumes so realisieren, daß sich die Operationen* `is_member`*,* `delete` *und* `insert` *in Zeit $O(\log(n))$ ausführen lassen.*

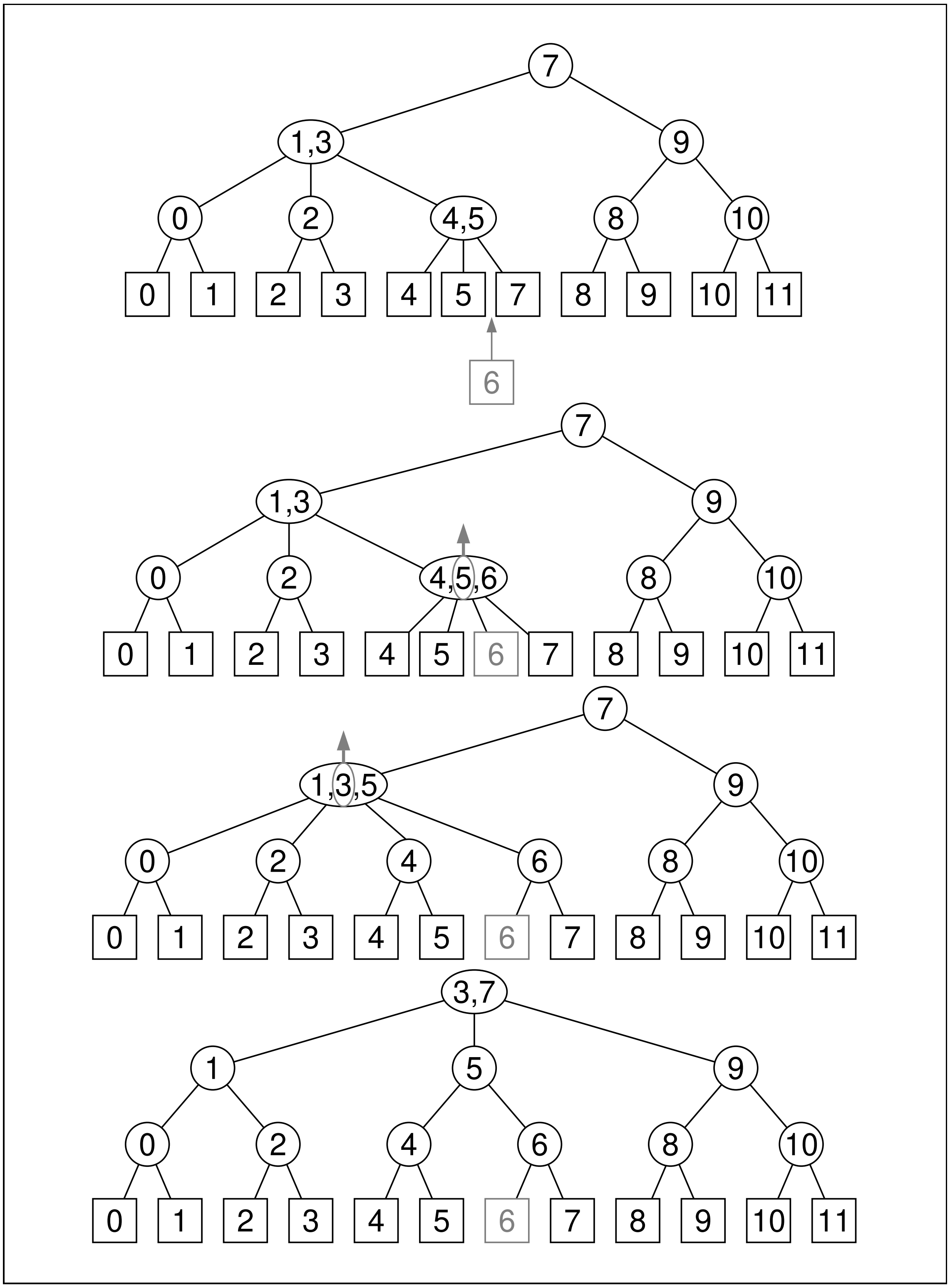

Abbildung 3.16: Ein Beispiel: Einfügen des Schlüssels 6 in einen $(2, 3)$-Baum

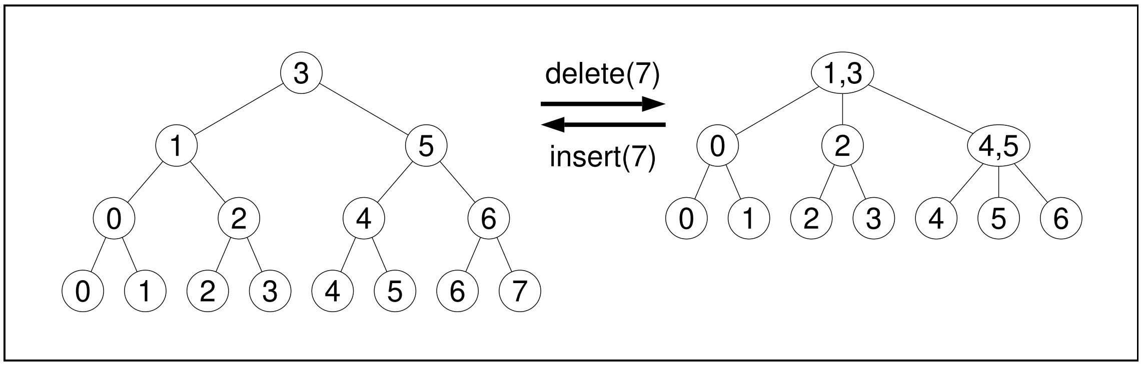

Abbildung 3.17: Eine Folge teurer Operationen in einem $(2,3)$-Baum

3.4.8 Weitere Varianten von Suchbäumen (+)

Ein Spezialfall des (a,b)-Baumes ist der sogenannte $(2,3)$-*Baum* (oft auch als 2-3–Baum bezeichnet), der bereits 1970 von J. Hopcroft eingeführt wurde. Historisch gesehen ist der (a,b)-Baum eine später entwickelte Verallgemeinerung des $(2,3)$-Baumes. Ein weiterer Spezialfall des (a,b)-Baumes ist der sogenannte *B-Baum* oder *Bayer-Baum*, benannt nach den Erfindern R. Bayer und E. McCreight (1970). Ein B-Baum der Ordnung m ist ein $(\lceil m/2 \rceil, m)$-Baum. Auch dieser war, historisch gesehen, bereits vor dem (a,b)-Baum entwickelt worden.

Wie wir gesehen haben, ist es beim Einfügen bzw. Löschen durchaus möglich, daß alle $O(\log(n))$ inneren Knoten auf dem Pfad von der Wurzel des (a,b)-Baums bis zum betrachteten Blatt inspiziert und modifiziert werden müssen. Es gibt Folgen von Lösch- und Einfüge-Operationen in einen (a,b)-Baum, so daß bei jeder Operation logarithmisch viele innere Knoten modifiziert werden müssen. Dies ist in Abbildung 3.17 illustriert, wobei immer wieder der Schlüssel 7 eingefügt und gelöscht wird. Wenn man nun die Bedingungen an a und b verschärft, so daß nun $b \geq 2a$ gilt, dann kann man zeigen, daß solche Fälle nicht mehr auftreten und daß insbesondere bei jeder Folge von $\ell \geq \log(n)$ Einfüge- und Lösch-Operationen maximal $O(\ell)$ innere Knoten betrachtet bzw. modifiziert werden müssen. Diese von K. Mehlhorn eingeführte Variante mit $b = 2a$ werden auch *schwache B-Bäume* genannt.

Andere Arten der balancierten Bäume gibt es ebenfalls. Eine Abart sind die sogenannten *α-balancierten Bäume*, in denen man statt fast gleichhoher Teilbäume an jedem Knoten, annähernd gleichgroße Teilbäume an jedem Knoten fordert. Ein Knoten v eines binären Baumes heißt *α-balanciert*, wenn gilt:

$$\alpha \leq \frac{|T_\ell(v)| + 1}{|T(v)| + 1} \leq 1 - \alpha,$$

wobei $T(v)$ bzw. $T_\ell(v)$ der Teilbaum gewurzelt an v bzw. der linke Teilbaum von v ist. Man überlege sich, daß die Definition symmetrisch ist (man kann in der Bedingung

$T_\ell(v)$ durch den rechten Teilbaum $T_r(v)$ von v ersetzen) und nur für $\alpha \in [0, \frac{1}{3}]$ Sinn macht. Man kann sich leicht davon überzeugen, daß die Höhe von α-balancierten Bäumen wiederum logarithmisch in der Größe des Baumes ist und daß sich die Einfüge- und Lösch-Operationen mit Hilfe von Rotationen in logarithmischer Zeit realisieren lassen.

Eine weitere Art balancierter Bäume sind die sogenannten *Rot-Schwarz-Bäume*. Ein Rot-Schwarz-Baum (engl. *Red-Black-Tree*) ist ein binärer Baum, dessen Knoten entweder rot oder schwarz gefärbt sind und folgende Eigenschaften erfüllen:

- Jeder innere Knoten hat *genau* 2 Kinder,
- Alle Blätter sind schwarz,
- Auf jedem Pfad von der Wurzel zu einem Blatt ist die Anzahl der schwarzen Knoten gleich,
- jedes Kind eines roten Knotens ist schwarz.

Man kann zeigen, daß die Höhe eines Rot-Schwarz-Baumes logarithmisch in der Größe ist ($\Rightarrow$ Übungen). Einen Rot-Schwarz-Baum kann man als Suchbaum verwenden, indem man die Schlüssel nur in den inneren Knoten abspeichert (d.h. die Blätter bleiben leer). Das Einfügen eines neuen Schlüssels in einen Rot-Schwarz-Baum geschieht ähnlich wie bei einem gewöhnlichen binären Suchbaum. Dabei stoßen wir bei einer erfolglosen Suche auf ein leeres schwarzes Blatt. Dort fügen wir den neuen Schlüssel ein, färben den Knoten rot und hängen an das ehemalige Blatt zwei neue schwarze Blätter. Damit sind die Bedingungen 1, 2 und 3 erfüllt. Es kann allerdings passieren, daß der Elter und Großelter der neuen Blätter beide rot sind. Mit Hilfe von Rotationen und Umfärbungen kann die Bedingung 4 wiederhergestellt werden, wobei die Bedingungen 1 mit 3 weiterhin gelten. Damit lasen sich die Einfüge- bzw. Lösch-Operationen ebenfalls in logarithmischer Zeit implementieren. Die Einzelheiten bleiben dem Leser zur Übung überlassen.

3.5 Suchen in Texten

In diesem Abschnitt wollen wir uns mit dem Suchen von 'kurzen' Zeichenfolgen in 'langen' Zeichenfolgen beschäftigen. Ein gutes Beispiel hierfür ist das Suchen einer Zeichenkette in einem langen Text. Diese Funktionalität wird sowohl von Editoren als auch von Betriebssystemen (z.B. `grep` unter Unix) zur Verfügung gestellt. Allerdings tauchen solche Problemstellungen z.B. auch in der Biologie auf, wo man kurze DNA-Sequenzen in einer langen DNA-Sequenz sucht.

```
BOOL SUBSTR(CHAR T[], INT N, CHAR S[], INT M)
{
    int i = 0, j = 0;
    while (i ≤ n − m)
    {
        while (t[i + j] == s[j])
        {
            j++;
            if (j == m) return TRUE;
        }
        i++;
        j = 0;
    }
    return FALSE;
}
```

Abbildung 3.18: Die naive Methode zum Suchen von Zeichenketten

3.5.1 Eine naive Methode

Zuerst stellen wir einen ganz naiven Ansatz vor, wie man dieses Problem lösen kann. Sucht man eine Zeichenkette $s = s_0 \cdots s_{m-1}$ der Länge m in einem Text $t = t_0 \cdots t_{n-1}$ der Länge n, so vergleicht man ab jeder Position $i \in [0 : n - m]$, ob der Teilstring $t_i \cdots t_{i+m-1}$ gleich dem gesuchten String s ist. Dazu benötigt man offensichtlich maximal $m(n - m + 1)$ viele Vergleiche von Zeichen. Der Algorithmus ist detailliert in Abbildung 3.18 angegeben. Damit erhalten wir das folgende Theorem:

Theorem 3.18 *Es genügen maximal $m(n-m+1)$ viele Vergleiche um festzustellen, ob eine Zeichenreihe s der Länge m in einer Zeichenreihe t der Länge $n \geq m$ enthalten ist.*

Man beachte, daß die obere Schranke wirklich angenommen wird, wenn man eine Zeichenkette $a^{m-1}b$ in einer Zeichenreihe $a^{n-1}b$ sucht. Im folgenden wollen wir zeigen, wie man die Anzahl der Vergleiche von $O(nm)$ auf $O(n + m)$ senken kann.

27. Juni 97

3.5.2 Der Knuth-Morris-Pratt-Algorithmus

Wir wollen im folgenden einen Vergleich von einem Paar von Zeichen, der negativ ausfällt, einen *Mismatch* nennen. Eine erste Idee ist, daß man nach einem Mismatch nicht wieder den ganzen zu suchenden String mit einem Teil des Textes vergleicht (das entspricht dem Zurücksetzen von j auf 0 und dem Hochzählen von i auf $i+1$ im Algorithmus in Abbildung 3.18). Es könnte ja möglich sein, daß man aus den Informationen der bereits gemachten Vergleichen einige Vergleiche einsparen kann.

Zunächst müssen wir einige benötigte Begriffe genauer definieren. Ein *Alphabet* Σ ist eine endliche Menge von Symbolen. *Zeichenreihen* sind endliche Folgen von Symbolen über einem gegebene Alphabet Σ, also Elemente aus Σ^*. Statt Zeichenreihe verwenden wir im folgenden auch die Begriffe *Zeichenkette* oder *String*. Mit $|s| = n$ bezeichnen wir die Länge eines Strings $s = s_0 \cdots s_{n-1}$. Sei $s = s_0 \cdots s_{n-1}$ ein String, dann ist $s' = s'_0 \cdots s'_{m-1}$ ein *Teilstring* von s, wenn es $i \in [0 : n-m]$ gibt, so daß $s_i \cdots s_{i+m-1} = s'$ ist. Damit ist der leere String ε Teilstring eines jeden Strings. Sei $s' = s_0 \cdots s_i$ ein Anfangsstück einer Zeichenreihe $s = s_0 \cdots s_{n-1}$, dann nennen wir s' einen *Präfix* von s. Analog nennen wir ein Endstück $s' = s_i \cdots s_{n-1}$ einer Zeichenreihe $s = s_0 \cdots s_{n-1}$ einen *Suffix* von s. Ein Präfix bzw. Suffix s' von s nennen wir einen *echten Präfix* bzw. *echten Suffix*, wenn $s' \neq s$ gilt.

Als *Rand* eines Strings s bezeichnen wir einen Präfix von s der zugleich auch Suffix von s ist. Offensichtlich sind der leere String ε und der String s selbst triviale Ränder von s. Zum Beispiel ist 'UNG' ein Rand von 'UNGLEICHUNG'. Als *eigentlichen Rand* eines Strings s, abgekürzt mit $\partial(s)$, bezeichnen wir den längsten Rand eines Strings, der nicht gleich s ist. In dem obigen Beispiel ist also 'UNG' sogar der eigentliche Rand von 'UNGLEICHUNG'. Man beachte, daß stets $|\partial(s)| < |s|$ gilt, wenn der eigentliche Rand des Strings s existiert. An dieser Stelle möchten wir noch darauf hinweisen, daß der leere String ε keinen eigentlichen Rand besitzt.

Haben wir nun ein Mismatch an der Position $i+j$ in t mit der Position j in s gefunden, dann wissen wir also, daß:

$$s_0 \cdots s_{j-1} = t_i \cdots t_{i+j-1} \quad \wedge \quad s_j \neq t_{i+j}.$$

Dieses Wissen wollen wir nun ausnutzen, indem wir i soweit auf i' erhöhen, so daß wir sicher sein können, daß der String s immer ungleich $t_\iota \cdots t_{\iota+m-1}$ für alle $\iota \in [i; i'-1]$ ist. Eine Erhöhung des Indexes i auf i' nennen wir einen *Shift*. Erfüllt ein solcher Shift die eben beschriebene Eigenschaft, wollen wir einen solchen Shift *sicher* nennen. Mit der Erhöhung von i auf $i' = i + j - |\partial(s_0 \cdots s_{j-1})|$ erhalten wir einen solchen sicheren Shift, andernfalls wäre $\partial(s_0 \cdots s_{j-1})$ kein eigentlicher Rand von $s_0 \cdots s_{j-1}$. Ein solcher sicherer Shift ist in Abbildung 3.19 schematisch dargestellt. Der Rand des Strings $s_0 \cdots s_{j-1}$ ist dabei grau hinterlegt. Man beachte, daß es durchaus sein kann, daß sich der Rand, interpretiert als Präfix und Suffix des Strings s, überlappen kann. Betrachte hierzu den konkreten String *abababa*, der als eigentlichen Rand $\partial(\mathit{abababa}) = \mathit{ababa}$ hat.

Wenn wir nun die Längen der eigentlichen Ränder aller Präfixe von s, und damit natürlich auch alle eigentlichen Ränder aller Präfixe von s, kennen würden, so könnten wir bei jedem Mismatch den Index i wie oben erhöhen und würden vermutlich

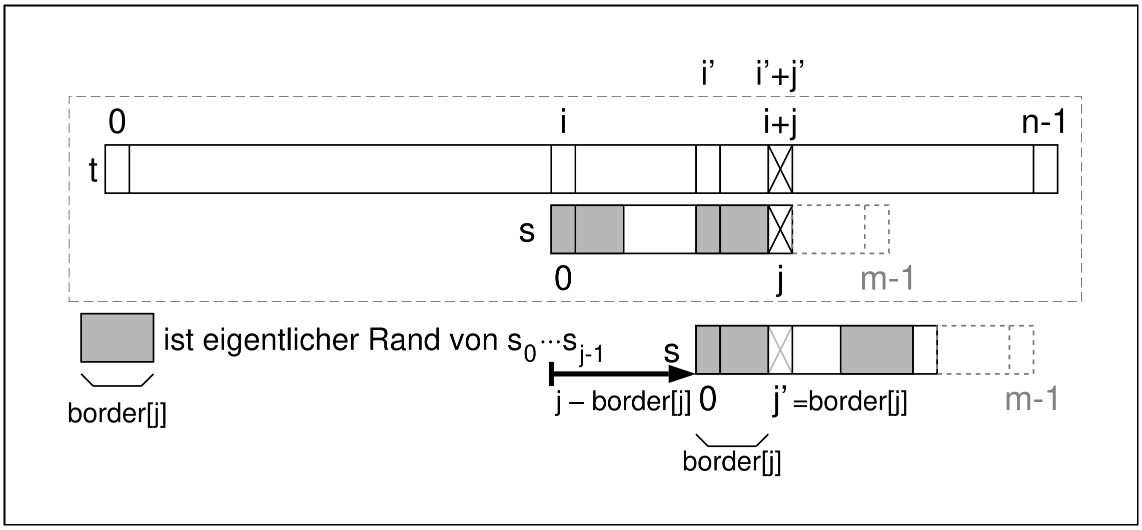

Abbildung 3.19: Sicherer Shift im Knuth-Morris-Pratt-Algorithmus

einen besseren Algorithmus erhalten. Nehmen wir also an, wir hätten die Längen der eigentlichen Ränder aller Präfixe von s in einer Tabelle *border* gespeichert:

$$border[j] = |\partial(s_0 \cdots s_{j-1})| \quad \text{für } j \in [1:m]$$

Im Spezialfall $j = 0$ setzen wir $border[0] = -1$, da dann $j - border[j]$ für $j = 0$ gerade 1 wird und wir so den Standardshift von i auf $i' = i + 1$ erhalten, der immer sicher ist. Damit erhalten wir den in Abbildung 3.20 angegebenen Algorithmus. Auch dieser Algorithmus wird nach seinen Erfindern D. Knuth, J. Morris und V. Pratt als *KMP-Algorithmus* bezeichnet.

1. Juli 97

3.5.3 Analyse des KMP-Algorithmus

Wie sieht es nun mit der maximal benötigten Anzahl an Vergleichen aus? Wir könnten uns bei der Abschätzung dumm anstellen und behaupten, daß die äußere **while**-Schleife maximal $(n - m + 1)$-mal und die innere **while**-Schleife maximal m-mal durchlaufen wird. Damit erhalten wir für die Anzahl der Vergleiche eine Abschätzung von $O(nm)$, was keine Verbesserung gegenüber dem naiven Ansatz bedeuten würde.

Die obige Abschätzung der Anzahl der Vergleiche ist zwar korrekt, allerdings überschätzen wir dabei die Anzahl der wirklich gemachten Vergleiche deutlich. Um eine bessere Abschätzung zu erhalten, müssen wir etwas geschickter argumentieren.

Als erstes stellen wir fest, daß in jedem erfolglosen Vergleich in der inneren **while**-Schleife der Index i um mindestens 1 erhöht wird, da der eigentliche Rand eines Strings immer kürzer als der String selbst ist, also $j - border[j] > 0$ gilt. Man

```
BOOL SUBSTR(CHAR T[], INT N, CHAR S[], INT M)
{
    int border[m + 1];
    compute_borders(int border[], int m, char s[]);
    int i = 0, j = 0;
    while (i ≤ n − m)
    {
        while (t[i + j] == s[j])
        {
            j++;
            if (j == m) return TRUE;
        }
        i+ = j − border[j];
        j = max{0, border[j]};
    }
    return FALSE;
}
```

Abbildung 3.20: Der Knuth-Morris-Pratt-Algorithmus

beachte, daß dies nach Definition auch für den Spezialfall $j = 0$ gilt. Damit macht der Algorithmus maximal $n - m + 1$ viele erfolglose Vergleiche. Für die erfolgreichen Vergleiche betrachten wir die Summe der Indizes $i + j$. Als erstes halten wir fest, daß $i + j$ am Ende der äußeren **while**-Schleife nie kleiner wird. Außerdem stellen wir fest, daß nach jedem erfolgreichen Vergleich der Wert von $i + j$ um eins steigt. Da immer $i + j \leq (n - m) + m = n$ gilt, macht der Algorithmus maximal n erfolgreiche Vergleiche.

Lemma 3.19 *Der KMP-Algorithmus benötigt maximal* $2n - m + 1$ *Vergleiche, wenn die Tabelle mit den Längen der eigentlichen Ränder aller Präfixe des gesuchten Strings gegeben ist.*

3.5.4 Bestimmung eigentlicher Ränder

Nun müssen wir uns nur noch überlegen, wie wir die Einträge der Tabelle *border* effizient berechnen können. Nehmen wir an, wir hätten die Tabelle bereits bis zur Position $j - 1$ konstruiert, und wir wollen nun $border[j]$ bestimmen. Ist das Element $s[border[j-1]] = s[j-1]$, so ist $border[j] = border[j-1] + 1$. Andernfalls müssen wir einen kürzeren Präfix von $s_0 \cdots s_{j-1}$ finden der auch ein Suffix von $s_0 \cdots s_{j-1}$ ist. Nach Konstruktion der Tabelle *border* ist der nächstkürzere Präfix der der Länge

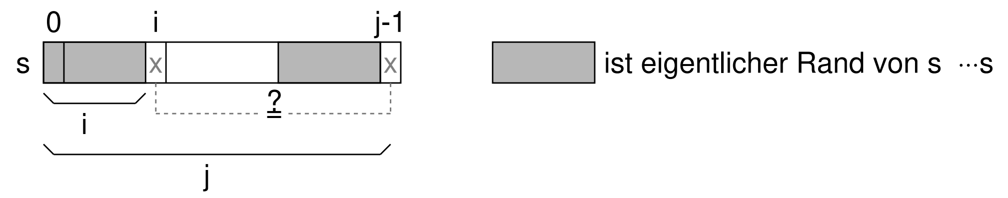

Abbildung 3.21: Bestimmung der Länge eines eigentlichen Randes von $s_0 \cdots s_{j-1}$, also von $border[j]$

$border[border[j-1]]$. Nun testen wir, ob sich dieser Rand von $s_0 \cdots s_{j-1}$ zu einem eigentlichen Rand von $s_0 \cdots s_j$ erweitern läßt. Diese Vorgehensweise ist schematisch in Abbildung 3.21 dargestellt. Damit erhalten wir den in Abbildung 3.22 angegebenen Algorithmus zur Bestimmung der Tabelle $border$.

Wieviele Vergleiche benötigt dieser Algorithmus zum Aufstellen der Tabelle? Auch hier verwenden wir wieder einen Trick zum Abzählen der Vergleiche. Bei jedem erfolglosen Vergleich wird i verkleinert und bei jedem erfolgreichen Vergleich um eins erhöht. Da bei jedem Inkrementieren von i anschließend auch j um eins erhöht wird, kann also i nur so oft inkrementiert werden wie j. Da $j \in [2 : m+1]$ ist, kann also i maximal $(m-1)$-mal inkrementiert werden. Das bedeutet, daß es maximal $m-1$ erfolgreiche Vergleiche gibt.

Beim Verkleinern wird i allerdings nie kleiner als -1 und es wird mindestens um 1 dekrementiert. Da i zu Beginn gleich 0 war und da beim Erniedrigen von i nur das

```
COMPUTE_BORDERS(INT BORDER[], INT M, CHAR S[])
{
    border[0] = −1;
    int i = border[1] = 0;
    for (int j = 2; j ≤ m; j++)
    {
        /* Observe that: i == border[j − 1] */
        while ((i ≥ 0) && (s[i] ≠ s[j − 1]))
            i = border[i];
        i++;
        border[j] = i;
    }
}
```

Abbildung 3.22: Berechnung der Tabelle *border* für den KMP-Algorithmus weggenommen werden kann, was schon einmal zu i hinzugefügt worden war bzw. was zu Beginn schon auf i lag, kann i maximal $(m-1)+1=m$ mal erniedrigt werden. Damit kann es nun auch nur maximal m viele erfolglose Vergleiche gegeben haben.

Lemma 3.20 *Die Tabelle border für einen String s der Länge m läßt sich mit maximal $2m-1$ vielen Vergleichen berechnen.*

Damit erhalten wir das folgende Theorem.

Theorem 3.21 *Der KMP-Algorithmus benötigt maximal $2n+m$ viele Vergleiche um zu entscheiden, ob ein String s der Länge m in einem String t der Länge n enthalten ist. Dafür ist maximal $O(m)$ zusätzlicher Speicherplatz nötig.*

Man beachte hierbei, daß die Anzahl der Vergleiche von der Größe des zugrundeliegenden Alphabets unabhängig ist.

Man kann den KMP-Algorithmus auch noch dahingehend verbessern, daß man ausnutzt, daß man beim Verschieben bereits das Zeichen kennt, das unmittelbar auf den Rand des Strings s folgt. Die erzielten Verbesserungen sind allerdings marginal zum benötigten Aufwand. Solche und weitere Verbesserungen des KMP-Algorithmus bleiben einer Spezialvorlesung über Stringmatching vorbehalten.

3.5.5 Der Boyer-Moore-Algorithmus

Der Ausgangspunkt für eine weitere Verbesserung, ist die folgende in Abbildung 3.23 angegebene Variante des naiven Algorithmus. Der einzige Unterschied zum ersten naiven Ansatz ist der, daß wir nun das gesuchte Wort mit einem Textstück nicht mehr von links nach rechts, sondern von rechts nach links vergleichen. Die Anzahl der Vergleiche ist im schlimmsten Falle immer noch dieselbe, nämlich $(n-m+1)m$. Betrachte hierzu den Text a^n und suche darin nach dem String ba^{m-1}.

Warum sollte dieser Ansatz also einen Vorteil haben? In der Praxis beinhalten die betrachteten Alphabete meist sehr viele Symbole. Löst nun ein Zeichen in dem gegebenen Text an Position $i+j$ einen Mismatch aus, welches im gesuchten String gar nicht vorkommt, so können wir i gleich auf $i'=i+j+1$ hochsetzen. Da bei Gleichverteilung der Buchstaben in den Texten dieser Fall mit großer Wahrscheinlichkeit auftritt, können wir bei Suchen in ASCII-Texten mit wesentlich weniger Vergleichen auskommen. Man beachte, daß dieser Vorteil bei Alphabeten mit kleiner Größe, wie z.B. beim Suchen in DNA-Sequenzen, nicht zum Tragen kommt!

```
BOOL SUBSTR(CHAR T[], INT N, CHAR S[], INT M)
{
    int i = 0, j = m − 1;
    while (i ≤ n − m)
    {
        while (t[i + j] == s[j])
        {
            if (j == 0) return TRUE;
            j−−;
        }
        i++;
        j = m − 1;
    }
    return FALSE;
}
```

Abbildung 3.23: Eine andere naive Methode zum Suchen von Zeichenketten

Dieser Ansatz wurde in dem nach R. Boyer und J. Moore benannten *Boyer-Moore-Algorithmus* zum Suchen in Strings verwendet. Der Boyer-Moor-Algorithmus, der detailliert in Abbildung 3.24 angegeben ist, sieht nun genau so aus wie dieser zweite

```
BOOL SUBSTR(CHAR T[], INT N, CHAR S[], INT M)
{
    int S[m + 1];
    compute_shift_table(S, m, s);
    int i = 0, j = m − 1;
    while (i ≤ n − m)
    {
        while (t[i + j] == s[j])
        {
            if (j == 0) return TRUE;
            j−−;
        }
        i+ = S[j];
        j = m − 1;
    }
    return FALSE;
}
```

Abbildung 3.24: Der Boyer-Moore-Algorithmus

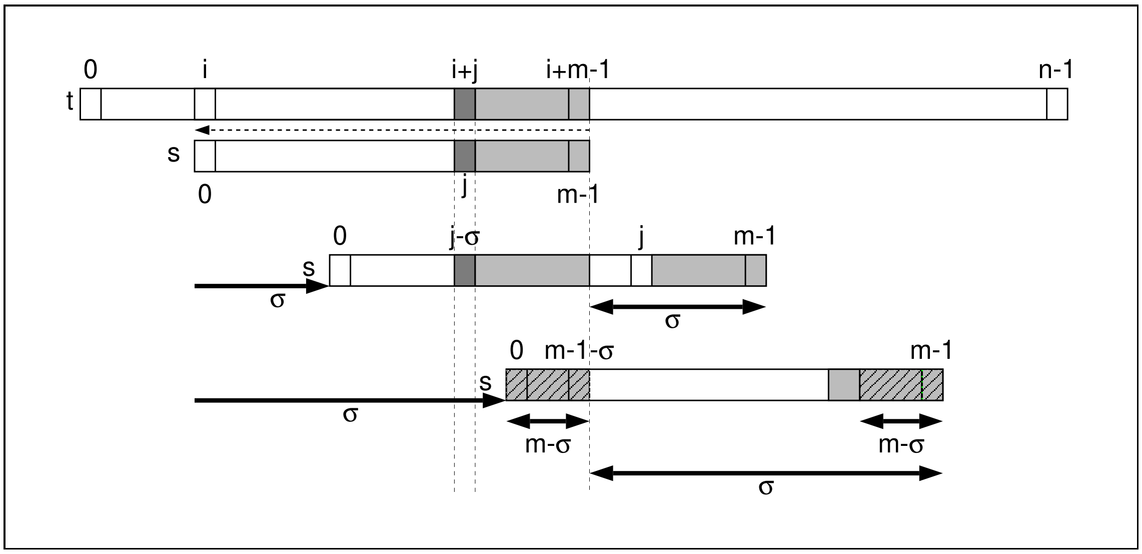

Abbildung 3.25: Mögliche Fälle eines sicheren Shifts beim Boyer-Moor-Algorithmus

naive Ansatz, mit der einzigen Ausnahme, daß nach einem Mismatch ein größerer Shift mit Hilfe einer Shift-Tabelle S ausgeführt wird. Wir müssen uns nur noch überlegen, welche Werte diese Shift-Tabelle haben soll und wie diese effizient zu berechnen ist.

Betrachten wir zur Bestimmung der Shift-Tabelle S die Abbildung 3.25. Wir nehmen also an, daß wir gerade versuchen den String $s = s_0 \cdots s_{m-1}$ mit dem Teilstring $t_i \cdots t_{i+m-1}$ von t zu vergleichen. Dabei haben wir im hinteren Bereich (wir vergleichen von rechts nach links!) eine Übereinstimmung, $s_{j+1} \cdots s_{m-1} = t_{i+j+1} \cdots t_{i+m-1}$, gefunden, während an der Position $(i+j, j)$ ein Mismatch aufgetreten ist: $s_j \neq t_{i+j}$. Für einen Shift um σ Positionen soll gelten, daß im selben Bereich von t wieder eine Übereinstimmung herrschen soll, also: $t_{i+j+1} \cdots t_{i+m-1} = s_{j-\sigma+1} \cdots s_{m-1-\sigma}$. Zusätzlich soll $s_j \neq s_{j-\sigma}$ sein, da andernfalls der String s kein Teilstring von t ab der Position $i + \sigma$ sein kann. Damit die letzte Bedingung Sinn macht, kann dies nur für ‘kleine’ Shifts mit $\sigma \leq j$ gelten. Dies entspricht dem oberen Shift in Abbildung 3.25, wobei der Mismatch dunkelgrau und der übereinstimmende Teilstring hellgrau markiert ist.

Für größere Shifts $\sigma > j$ muß gelten, daß der nicht aus dem alten Bereich herausgeschobene Bereich mit dem Anfang des Strings s übereinstimmt. Formal muß also gelten, daß: $s_0 \cdots s_{m-1-\sigma} = s_{\sigma+1} \cdots s_{m-1}$. Dieser Fall korrespondiert mit dem unteren Shift in Abbildung 3.25, dabei ist hier die Übereinstimmung (also ein Rand) schraffiert gezeichnet.

4. Juli 97

Bezeichnet man mit Rand(s) die Menge aller Ränder von s, so kann man die beiden obigen Bedingungen wie folgt umformulieren:

$$\begin{aligned} s_{j+1}\cdots s_{m-1} \in \mathrm{Rand}(s_{j+1-\sigma}\cdots s_{m-1}) \quad &\wedge \quad s_j \neq s_{j-\sigma} \\ s_0\cdots s_{m-1-\sigma} \in \mathrm{Rand}(s_0\cdots s_{m-1}) \quad &\wedge \quad \sigma > j \end{aligned}$$

Damit wir einen sicheren Shift erhalten, werden wir in unserer Tabelle den kleinsten Wert σ eintragen, der eine der beiden Bedingungen erfüllt, also

$$S[j] = \min\left\{\sigma : \begin{array}{l} (s_{j+1}\cdots s_{m-1} \in \mathrm{Rand}(s_{j+1-\sigma}\cdots s_{m-1}) \wedge s_j \neq s_{j-\sigma}) \vee \\ (s_0\cdots s_{m-1-\sigma} \in \mathrm{Rand}(s_0\cdots s_{m-1}) \wedge \sigma \geq j+1) \end{array}\right\}.$$

In beiden Fällen müssen wir also wieder die Ränder von Teilstrings des gesuchten Strings s kennen. Im Gegensatz zum KMP-Algorithmus benötigen wir allerdings nicht nur die Ränder von Präfixen von s, sondern insbesondere für den ersten Fall auch die Ränder von Suffixen von s. Im ersten Fall sind wir allerdings nicht primär an den Rändern der Suffixe interessiert, denn wir fordern ja noch zusätzlich, daß das Zeichen unmittelbar vor dem Rand ungleich dem Zeichen unmittelbar vor dem Suffix sein soll. Dennoch können wir die Prozedur zur Berechnung der Ränder des KMP-Algorithmus wiederverwerten. Die gesamte Prozedur zur Berechnung der Shift-Tabelle S ist in Abbildung 3.26 angegeben. Im ersten Teil berechnen wir die Einträge an den Positionen j in der $S[j] \leq j$ gilt. Im zweiten Teil ergänzen wir die Tabelle für diejenigen Positionen, für die $S[j] > j$ gilt. Der erste Teil entspricht der ersten Bedingung in der Disjunktion der Definition von S, während der zweite Teil zu der zweiten Bedingung in der Disjunktion korrespondiert.

Zum besseren Verständnis der folgenden Argumentation sollte man Abbildung 3.25 zu Hilfe ziehen. Im ersten Teil sind die Ränder eines Suffixes gefragt, in denen das Zeichen unmittelbar vor dem Rand am Ende des Suffixes ungleich dem Zeichen unmittelbar vor dem Suffix ist. Das entspricht bei der Berechnung der Ränder genau dem Fall, in dem wir in der **while**-Schleife den bisherigen Rand nicht verlängern können und i auf $border[i]$ erniedrigen. In diesem Fall können wir also den Wert der Shift-Tabelle $S[j]$ verbessern. Da ein bestimmter String Rand verschiedener Suffixe sein kann, und wir bei der Aktualisierung der Shift-Tabelle S nur den kleinsten Wert berücksichtigen dürfen, wird die Tabelle über eine Minimumbildung aktualisiert. Man beachte, daß im Algorithmus $j' \in [1, m]$ die Länge des betrachteten Suffixes angibt, der Mismatch also an der Position $j' - 1$ von hinten im String auftritt. Auch hier zählen wir die Positionen wieder von 0 beginnend, diesmal allerdings von rechts nach links. Daher wird bei einem Mismatch an Position j' (von rechts) die Shift-Tabelle an Position $(m-1) - (j'-1) = m - j'$ aktualisiert.

Im zweiten Teil bestimmen wir zunächst wie im KMP-Algorithmus die Tabelle *border* für alle eigentlichen Ränder aller Präfixe des gesuchten Strings s. Mit Hilfe

```
COMPUTE_SHIFT_TABLE(INT S[], INT M, CHAR S[])
{
    /* initialize shift-table S. */
    for (int j = 0; j ≤ m; j++)
        S[j] = ∞;

    /* part 1: S[j] ≤ j */
    int border_rev[m + 1];
    border_rev[0] = −1;
    int i = border_rev[1] = 0;
    /* j' ∈ [2 : m] ist the length of the considered suffix! */
    /* Note that in the following j' = m − j */
    for (int j' = 2; j' ≤ m; j'++)
    {   /* Note that i == border_rev[j' − 1] */
        while ((i ≥ 0) && (s[m − i − 1] ≠ s[m − j']))
        {
            S[m − j'] = min(S[m − j'], j' − i − 1);
            i = border_rev[i];
        }
        i++;
        border_rev[j] = i;
    }

    /* part 2: S[j] > j */
    int border[m + 1];
    compute_borders(border, m, s);
    int i = border[m], j = 0;
    while (i ≥ 0)
    {
        int σ = m − i;
        while (j < σ)
        {
            S[j] = min(S[j], σ);
            j++;
        }
        i = border[i];
    }
}
```

Abbildung 3.26: Algorithmus zur Berechnung der sicheren Shifts im Boyer-Moore-Algorithmus dieser Tabelle können wir nun die Shift-Tabelle an den Stellen aktualisieren, an denen $S[j] > j$ gilt. Solange sich die Position j noch im aktuell betrachteten Rand des Strings s befindet, also $j < m-i$ ist, ist ein Shift um $\sigma := m-i$ Positionen zulässig. Andernfalls müssen wir den nächsten zulässigen Rand betrachten ($i = border[i]$).

Da die Prozedur im wesentlichen auf die Prozedur `compute_borders` zurückgreift, folgt unmittelbar das folgende Lemma.

Lemma 3.22 *Die im Boyer-Moore-Algorithmus verwendete Shift-Tabelle für einen String der Länge m läßt sich mit maximal $O(m)$ Vergleichen berechnen.*

Die aufwendige Analyse des Boyer-Moore-Algorithmus führen wir an dieser Stelle nicht aus, sondern verweisen auf das Buch von M. Crochemore und W. Rytter und halten nur das Ergebnis fest:

Theorem 3.23 *Der Boyer-Moore-Algorithmus benötigt maximal $O(n+m)$ viele Vergleiche um zu entscheiden, ob ein String s der Länge m in einem String t der Länge n enthalten ist. Dafür ist maximal $O(m)$ zusätzlicher Speicherplatz nötig.*

In der Praxis hat sich herausgestellt, daß der Boyer-Moore Algorithmus für große Alphabete schneller als der Knuth-Morris-Pratt-Algorithmus arbeitet, insbesondere wenn man die am Anfang des Abschnittes erwähnte Heuristik benutzt. Hierbei wird bei einem Mismatch mit einem Zeichen x im Text t der gesuchte String s der Länge m um m Positionen weitergeschoben, wenn das Zeichen x im String s gar nicht vorkommt.

Wir wollen hier auch noch anmerken, daß die Verallgemeinerung des oben angegebenen Algorithmus von Boyer-Moore, der alle mit s identischen Teilstrings von t findet, quadratische Laufzeit $O(nm)$ bekommt. Dies steht völlig im Gegensatz zur Verallgemeinerung des KMP-Algorithmus, der bei dieser Verallgemeinerung seine lineare Laufzeit beibehält. Allerdings kann man mit Hilfe einer Verbesserung von Z. Galil auch hier wieder eine lineare Laufzeit von $O(n+m)$ erhalten. Auch hier verweisen wir für Details wieder auf das Buch von M. Crochemore und W. Rytter.

8. Juli 97

4 Graphenalgorithmen

4.1 Grundlagen der Graphentheorie

Viele Probleme lassen sich mit Hilfe von Graphen modellieren. So kann man etwa Straßennetze oder dergleichen als einen Graphen darstellen, Abhängigkeiten lassen sich mit Hilfe von Graphen repräsentieren und auch Suchbäume haben wir als spezielle Graphen kennengelernt. Zunächst werden wir den notwendigen Begriffsapparat der Graphentheorie bereitstellen.

4.1.1 Ungerichtete Graphen

Ein *ungerichteter Graph* G ist ein Paar (V, E). Dabei ist V eine Menge von Elementen, die wir als *Knoten* bezeichnen. Die Menge E ist eine Teilmenge der Menge der zweielementigen Mengen von Knoten, also $E \subseteq \{\{v, w\} | v, w \in V\} =: \binom{V}{2}$. Die Elemente von E bezeichnen wir als *Kanten*. Im folgenden werden wir Knotenmenge bzw. Kantenmenge eines Graphen G auch mit $V(G)$ und $E(G)$ bezeichnen. Zwei Knoten $v \in V$ und $w \in V$ sind *benachbart* bzw. *adjazent*, wenn es eine Kante zwischen ihnen gibt, also $\{v, w\} \in E$ ist. Eine Kante $e \in E$ und ein Knoten $v \in V$ heißen *inzident*, wenn $v \in e$ gilt. Wenn aus dem Kontext klar ist, daß wir über ungerichtete Graphen sprechen, so schreiben wir für eine Kante $\{v, w\}$ auch (v, w). Manchmal werden wir bei ungerichteten Graphen auch einelementige Mengen als Kanten zulassen, d.h. $E \subseteq \{\{v, w\}, \{v\} | v, w \in V\}$. Solche Kanten $e = \{v\}$ bezeichnen wir als *Schleifen*, weil sie den Knoten v mit ihm selbst verbinden.

Die zu einem Knoten $v \in V$ adjazenten Knoten bezeichnen wir als seine *Nachbarn*. Die Menge der Nachbarn eines Knotens v bezeichnen wir mit $N(v)$. Der *Grad* eines Knotens $v \in V$ ist die Anzahl seiner Nachbarn und wird mit $\deg(v) = |N(v)|$ bezeichnet. Einen Knoten mit Grad 0 nennen wir einen *isolierten Knoten*. Der maximale Grad eines Knotens in einem ungerichteten Graphen G bezeichnen wir als *Maximalgrad* $\Delta(G)$. Der minimale Grad eines Knotens in einem ungerichteten Graphen G bezeichnen wir als *Minimalgrad* $\delta(G)$.

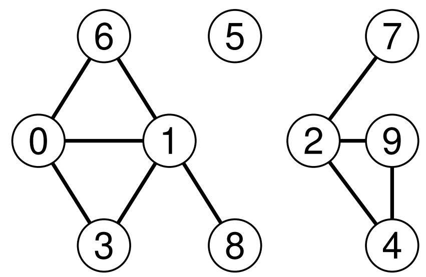

Abbildung 4.1: Beispiel für einen ungerichteten Graphen

Eine Folge $p = (v_0, \ldots, v_n)$ von Knoten bezeichnen wir als *Pfad* oder *Weg* im Graphen $G = (V, E)$, wenn $\{v_{i-1}, v_i\} \in E$ für alle $i \in [1 : n]$ gilt. Wir bezeichnen die Knoten v_0 und v_n als die *Endknoten* des Pfades $p = (v_0, \ldots, v_n)$. Ein Pfad p *verbindet* die Knoten v und w, wenn v und w die Endknoten dieses Pfades p sind. Ein Pfad $p = (v_0, \ldots, v_n)$ heißt *einfach*, wenn jeder Knoten des Pfades nur einmal auftaucht, also $|\{v_0, \ldots, v_n\}| = n + 1$. Für einen Pfad $p = (v_0, \ldots, v_n)$ ist n die *Länge des Pfades p*. Ein Pfad $p = (v_0, \ldots, v_n)$ heißt *Kreis*, wenn $v_0 = v_n$. Ein Kreis $c = (v_0, \ldots, v_n)$ heißt *einfacher Kreis*, wenn er mindestens Länge 3 hat und außer $v_0 = v_n$ jeder Knoten nur einmal auftaucht, also $|\{v_1, \ldots, v_n\}| = n$ und $n > 2$.

Ein Graph $G' = (V', E')$ heißt *Teilgraph* von $G = (V, E)$ (in Zeichen $G' \sqsubseteq G$), wenn $V' \subseteq V$ und $E' \subseteq E \cap \binom{V'}{2}$ gilt. Ein Teilgraph $G' = (V', E')$ von $G = (V, E)$ heißt *induzierter Teilgraph*, wenn alle Kanten in G, die Knoten aus V' verbinden, auch in E' enthalten sind, also:

$$E' = E \cap \binom{V'}{2}.$$

Sei $G = (V, E)$ ein Graph und $V' \subseteq V$ eine Teilmenge der Knoten, dann bezeichnen wir den von V' induzierten Teilgraphen mit $G[V'] = (V', E \cap \binom{V'}{2})$.

Ein ungerichteter Graph G heißt *zusammenhängend*, wenn je zwei Knoten durch einen Pfad in G verbunden sind. Einen maximal zusammenhängenden Teilgraph $G' \sqsubseteq G$ bezeichnen wir als eine *Zusammenhangskomponente* von G.

Ein ungerichteter Graph G heißt *azyklisch*, wenn es keinen Kreis in G gibt. Ein azyklischer und zusammenhängender ungerichteter Graph wird als *ungerichteter Baum* oder als *freier Baum* bezeichnet. Ein Knoten v eines Baumes T wird ein *Blatt* genannt, wenn er Grad 1 hat. Ein azyklischer Graph wird *Wald* genannt, da jede Zusammenhangskomponente ein Baum ist.

Ungerichtete Graphen können wir recht schön bildlich darstellen. Knoten zeichnen wir als Kreise und eine Kante verbindet die beiden Kreise, die ihre Endknoten darstellen. Ein Beispiel ist in Abbildung 4.1 für den Graphen $G = (V, E)$ gegeben, mit:

$$\begin{aligned} V &= \{0,1,2,3,4,5,6,7,8,9\} \\ E &= \Big\{\{0,1\},\{0,3\},\{0,6\},\{1,3\},\{1,6\},\{1,8\},\{2,4\},\{2,7\},\{2,9\},\{4,9\},\Big\} \end{aligned}$$

In diesem Beispiel ist zum Beispiel der Knoten 5 ein isolierter Knoten. Der Minimalgrad des Graphen ist $\delta(G) = 0$ und der Maximalgrad ist $\Delta(G) = 4$. Die Folge $(1,3,0,1,6)$ ist ein Pfad, der allerdings nicht einfach ist. Die Folge $(6,1,3,0,6)$ ist ein einfacher Kreis.

Der Graph $G' = (V', E')$ mit

$$V' = \{2,4,7,9\} \qquad E' = \{\{2,4\},\{2,7\},\{2,9\}\}$$

ist ein Teilgraph von G. G' ist allerdings kein induzierter Teilgraph, da er die Kante $\{4,9\}$ nicht enthält. Der Teilgraph $G[\{0,1,3,6\}]$ ist zusammenhängend, jedoch keine Zusammenhangskomponente des Graphen G, da sich dieser Teilgraph um den Knoten 8 noch zu einem größeren zusammenhängenden Teilgraph erweitern läßt. Die Zusammenhangskomponenten des Graphen G sind: $G[\{0,1,3,6,8\}]$, $G[\{5\}]$ und $G[\{2,4,7,9\}]$. Darüber hinaus ist nur die Zusammenhangskomponente $G[\{5\}]$ azyklisch.

Zum Schluß halten wir noch das folgende evidente Lemma fest:

Lemma 4.1 *In einem ungerichteten Graphen (mit Schleifen) mit n Knoten ist die Anzahl der Kanten maximal $\binom{n}{2}$ ($\binom{n}{2} + n = \binom{n+1}{2}$).*

4.1.2 Gerichtete Graphen

Ein *gerichteter Graph* G ist ebenfalls ein Paar (V, E). Wieder ist V eine Menge von Elementen, die wir als *Knoten* bezeichnen. Die Menge E besteht nun aus Paaren von Knoten, also $E \subseteq \{(v,w) | v, w \in V\} = V \times V$. Auch hier bezeichnen wir die Elemente aus E als *Kanten*. Mit $V(G)$ bzw. $E(G)$ bezeichnen wir die Knotenmenge bzw. Kantenmenge eines Graphen G. Sei $e = (v,w) \in E$ eine Kante, dann bezeichnen wir v als *Anfangsknoten* und w als *Endknoten* der Kante e. Zwei Knoten $v \in V$ und $w \in W$ heißen adjazent, wenn $(v,w) \in E$ oder $(w,v) \in E$ gilt. Ein Knoten $v \in V$ und eine Kante $e \in E$ heißen *inzident*, wenn $e = (v,w)$ oder $e = (w,v)$ für ein $w \in V$ ist. Ein Knoten $v \in V$ heißt *direkter Vorgänger* eines Knotens $w \in V$, wenn es eine Kante $(v,w) \in E$ gibt. Wir bezeichnen dann auch w als einen *direkten Nachfolger* von v.

Analog wie bei einem ungerichteten Graphen bezeichnen wir die zu einem Knoten $v \in V$ adjazenten Knoten als *Nachbarn*. Die Menge der Nachbarn eines Knotens v bezeichnen wir mit $N(v)$. Im Falle gerichteter Graphen unterteilen wir die Nachbarschaft in die Menge der direkten Nachfolger und der direkten Vorgänger, die wir mit $N^+(v)$ und $N^-(v)$ bezeichnen wollen:

$$\begin{aligned} N^+(v) &= \{w \in V | (v,w) \in E\} \\ N^-(v) &= \{w \in V | (w,v) \in E\} \\ N(v) &= N^+(v) \cup N^-(v) \end{aligned}$$

Der *Grad* eines Knotens $v \in V$ ist die Anzahl seiner Nachbarn und wird mit $\deg(v) = |N(v)|$ bezeichnet. Der *Out-Grad* eines Knotens $v \in V$ ist die Anzahl seiner direkten Nachfolger und wird mit $\deg^+(v) = |N^+(v)|$ bezeichnet. Der *In-Grad* eines Knotens $v \in V$ ist die Anzahl seiner direkten Vorgänger und wird mit $\deg^-(v) = |N^-(v)|$ bezeichnet. Wieder bezeichnen wir den maximalen Grad eines Knotens in einem Graphen G als *Maximalgrad* $\Delta(G)$ des Graphen G. Der maximale In-Grad bzw. Out-Grad eines Graphen G wird mit $\Delta^-(G)$ bzw. mit $\Delta^+(G)$ bezeichnet. Der minimale Grad eines Knotens in einen Graphen G bezeichnen wir als *Minimalgrad* $\delta(G)$ des Graphen G. Der minimale In-Grad bzw. Out-Grad eines Graphen G wird mit $\delta^-(G)$ bzw. mit $\delta^+(G)$ bezeichnet.

Eine Folge $p = (v_0, \ldots, v_n)$ von Knoten bezeichnen wir als *ungerichteten Pfad* oder als *ungerichteten Weg*, wenn $(v_{i-1}, v_i) \in E$ oder $(v_i, v_{i-1}) \in E$ für jedes $i \in [1:n]$ gilt. Eine Folge $p = (v_0, \ldots, v_n)$ von Knoten bezeichnen wir als *gerichteten Pfad* oder als *gerichteten Weg* im Graphen $G = (V, E)$, wenn $(v_{i-1}, v_i) \in E$ für alle $i \in [1:n]$ gilt. Wir bezeichnen den Knoten v_0 als *Anfangsknoten* und v_n als den *Endknoten* des ungerichteten bzw. gerichteten Pfades $p = (v_0, \ldots, v_n)$. Ein Knoten w heißt *Nachfolger* eines Knotens v, wenn es einen gerichteten Weg von v nach w in G gibt. In diesem Fall sagen wir auch, daß v ein Vorgänger von w ist. Wenn Verwechslungen ausgeschlossen sind, reden wir auch im Fall gerichteter Graphen einfach von Pfaden oder Wegen. Es wird dann aus dem Zusammenhang klar, ob gerichtete oder ungerichtete Wege gemeint sind.

Ein ungerichteter Pfad p *verbindet* die Knoten $v \in V$ und $w \in V$ von v nach w, wenn v und w die Endknoten eines Pfades p sind. Ein gerichteter Pfad p *verbindet* die Knoten von $v \in V$ nach $w \in V$, wenn v der Anfangsknoten und w der Endknoten eines Pfades p ist. Ein (un)gerichteter Pfad $p = (v_0, \ldots, v_n)$ heißt *einfach*, wenn jeder Knoten des Pfades nur einmal auftaucht, also $|\{v_0, \ldots, v_n\}| = n + 1$. Für einen (un)gerichteten Pfad $p = (v_0, \ldots, v_n)$ nennen wir n die *Länge des Pfades p*. Ein gerichteter Pfad $p = (v_0, \ldots, v_n)$ heißt *gerichteter Kreis*, wenn $v_0 = v_n$. Ein gerichteter Kreis $c = (v_0, \ldots, v_n)$ heißt *einfach*, wenn außer $v_0 = v_n$ jeder Knoten nur einmal auftaucht, also $|\{v_1, \ldots, v_n\}| = n$.

Ein gerichteter Graph $G' = (V', E')$ heißt *Teilgraph* von einem gerichteten Graphen $G = (V, E)$ (in Zeichen $G' \sqsubseteq G$), wenn $V' \subseteq V$ und $E' \subseteq E \cap (V' \times V')$ gilt. Ein gerichteter Teilgraph $G' = (V', E')$ heißt *induzierter Teilgraph*, wenn alle Kanten im gerichteten Graphen G, die Knoten in V' verbinden auch in E' enthalten sind, also:

$$E' = E \cap (V' \times V').$$

Sei $G = (V, E)$ ein gerichteter Graph und $V' \subseteq V$ eine Teilmenge der Knoten, dann bezeichnen wir den von V' induzierten Teilgraphen mit $G[V] = (V', E \cap (V' \times V'))$.

Ein gerichteter Graph heißt *schwach zusammenhängend*, wenn je zwei Knoten durch einen ungerichteten Pfad verbunden sind. Ein gerichteter Graph heißt *stark zusammenhängend*, wenn es für jedes Paar $v, w \in V$ sowohl einen gerichteten Pfad von v nach w als auch einen gerichteten Pfad von w nach v gibt. Ein maximaler schwach zusammenhängender Teilgraph eines gerichteten Graphen G heißt *schwache Zusammenhangskomponente* von G. Analog heißt ein maximaler starker zusammenhängender Teilgraph eines gerichteten Graphen G eine *starke Zusammenhangskomponente* von G.

Ein gerichteter Graph G heißt *azyklisch*, wenn es keinen gerichteten Kreis in G gibt. Einen azyklischen gerichteten Graphen nennen wir auch einen *DAG* (von engl. *directed acyclic graph*). Ein gerichteter Graph G wird ein *gewurzelter Baum* genannt, wenn G schwach zusammenhängend und azyklisch ist und jeder Knoten In-Grad ≤ 1 hat. Man kann zeigen, daß es dann genau einen Knoten mit In-Grad 0 gibt, den wir als die *Wurzel* des Baumes bezeichnen wollen. Sei $T = (V, E)$ ein gewurzelter Baum, dann bezeichnen wir mit $r(T)$ die Wurzel dieses Baumes T. In einem solchen gewurzelten Baum bezeichnen wir die Knoten mit Out-Grad 0 als *Blätter*. Anstelle von direktem Vorgänger bzw. Nachfolger sprechen wir bei gewurzelten Bäumen auch von *Elter* bzw. *Kind*. Zwei Knoten eines gewurzelten Baumes, die denselben Elter haben bezeichnet man als *Geschwister*. Ein azyklischer gerichteter Graph in dem jeder Knoten In-Grad ≤ 1 hat, bezeichnen wir als *gewurzelten Wald*, da jede schwache Zusammenhangskomponente ein gewurzelter Baum ist.

Ein gewurzelter Baum heißt *k-är*, wenn jeder Knoten maximal k Kinder hat. Ein k-ärer gewurzelter Baum heißt *erweitert*, wenn jeder Knoten genau k oder 0 Kinder hat. Ein gewurzelter Baum heißt *geordnet*, wenn für jeden Knoten des Baumes eine Ordnung auf seinen Kindern gegeben ist. Suchbäume sind zum Beispiel binäre geordnete gewurzelte Bäume. Man sollte beachten, daß Suchbäume sogar noch mehr Eigenschaften haben, da ein Knoten mit einem Kind ja entweder ein linkes oder ein rechtes Kind ist. Definiert man Suchbäume etwas anders als in dieser Vorlesung, nämlich in Anlehnung an Rot-Schwarz-Bäume, so sind Suchbäume erweiterte binäre geordnete gewurzelte Bäume. Hierbei werden die Schlüssel allerdings nur in den inneren Knoten abgespeichert. Löscht man in einem solchen Baum alle Blätter, so erhält man genau unsere Suchbäume.

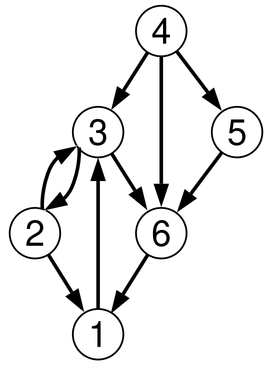

Abbildung 4.2: Beispiel für einen gerichteten Graphen

Auch gerichtete Graphen lassen sich wie ungerichtete Graphen sehr schön bildlich darstellen. Hierbei erhält jede Kante einen Pfeil, der vom Anfangs- zum Endknoten hin gerichtet ist. In Abbildung 4.2 ist eine solche Darstellung des folgenden Graphen $G = (V, E)$ gegeben, wobei:

$$\begin{aligned} V &= \{1, 2, 3, 4, 5, 6\} \\ E &= \big\{(1,3), (2,1), (2,3), (3,2)(3,6), (4,3), (4,5), (4,6), (5,6), (6,1)\big\} \end{aligned}$$

Zum Beispiel ist der minimale In-Grad bzw. maximale Out-Grad dieses Graphen 0 bzw. 3. Die Folge $(6, 5, 4, 3, 2)$ ist ein ungerichteter Pfad der Länge 4, der allerdings kein gerichteter Pfad ist. Die Folge $(3, 6, 1, 3, 2)$ ist ein gerichteter Pfad der Länge 4, der allerdings nicht einfach ist. Die Folge $(3, 6, 1, 3)$ ist ein einfacher gerichteter Kreis der Länge 3.

Der Graph $G' = (V', E')$ mit

$$V' = \{3, 4, 5, 6\} \qquad E = \{(4, 3), (4, 5), (4, 6), (5, 6)\}$$

ist ein Teilgraph von G, der allerdings nicht induziert ist, da die Kante $(3, 6) \notin E'$. Der gesamte Graph G ist schwach zusammenhängend, allerdings nicht stark zusammenhängend, da der Knoten 4 von keinem anderen Knoten über einen gerichteten Pfad erreicht werden kann. Damit ist $G[\{4\}]$ eine starke Zusammenhangskomponente des Graphen G. Nun sieht man leicht, daß dann ebenfalls $G[\{5\}]$ eine starke Zusammenhangskomponente des Graphen G ist. Die restlichen Knoten induzieren die letzte starke Zusammenhangskomponente des Graphen G: $G[\{1, 2, 3, 6\}]$.

Zum Schluß halten wir noch das folgende Lemma fest:

Lemma 4.2 *In einem gerichteten Graphen mit n Knoten ist die Anzahl der Kanten maximal n^2.*

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

Abbildung 4.3: Repräsentation des Graphen aus Abbildung 4.2 mit Hilfe einer Adjazenzmatrix

4.1.3 Repräsentationen von Graphen

In diesem Abschnitt wollen wir uns nun überlegen, wie wir Graphen im Rechner repräsentieren können. Wir wollen hierzu die beiden gängigsten Methoden vorstellen.

Sei also $G = (V, E)$ ein Graph mit n Knoten und m Kanten, also $|V| = n$ und $|E| = m$. Für die Knoten werden wir immer annehmen, daß sie aus dem Bereich der ganzen Zahlen stammen. Für eine effiziente Speicherung stellen wir uns die Menge der Knoten als $V = \{1, \ldots, n\}$ vor. In Zusammenhang mit `C`-ähnlichen Programmen werden wir auch manchmal annehmen, daß $V = \{0, \ldots, n-1\}$ ist. Aus dem Zusammenhang wird klar werden, welche Darstellung der Knoten wir gewählt haben.

Wie können wir nun die Menge der Kanten darstellen? Eine einfache Möglichkeit sind sogenannte *Adjazenzmatrizen*. Eine Adjazenzmatrix ist eine $n \times n$-Matrix $A = (a_{ij})_{1 \leq i,j \leq n}$. Dabei ist bei ungerichteten Graphen genau dann $a_{ij} = 1$, wenn i und j adjazent sind, also $\{i, j\} \in E$. Ansonsten ist $a_{ij} = 0$. Bei gerichteten Graphen ist genau dann $a_{ij} = 1$, wenn es eine Kante von i nach j gibt, also $(i, j) \in E$. Die Diagonalenelemente von ungerichteten Graphen sind nach Definition immer gleich 0 ($a_{ii} = 0$). Nur wenn wir die ungerichteten Graphen um Schleifen erweitern, kann $a_{ii} = 1$ sein. Manchmal wird es für algorithmische Zwecke sinnvoll sein einfach zu fordern, daß a_{ii} immer gleich 0 oder immer gleich 1 sein soll, unabhängig davon, ob im (un)gerichteten Graphen Schleifen vorhanden sind oder nicht.

Das Hauptproblem bei dieser Repräsentation ist der verschwenderische Umgang mit Speicherplatz. Um einen Graphen mit n Knoten und m Kanten darzustellen, benötigt man immer $O(n^2)$ Platz. Für Graphen mit relativ wenig Kanten (z.B. $m = O(n)$ oder $m = O(n \log(n))$) ist dies sehr ungünstig.

Eine andere Darstellung, die weniger Speicherplatz benötigt, kann mit sogenannten *Adjazenzlisten* erfolgen. Hierbei werden die zu einem Knoten v adjazenten Knoten in einer linearen Liste dargestellt. Entweder werden in diesen Listen die Namen der adjazenten Knoten (d.h. die Zahlen aus $\{0, \ldots, n-1\}$) gespeichert oder, meist besser, nur ein Verweis auf den Knoten selbst. Im Falle ungerichteter Graphen beinhaltet die

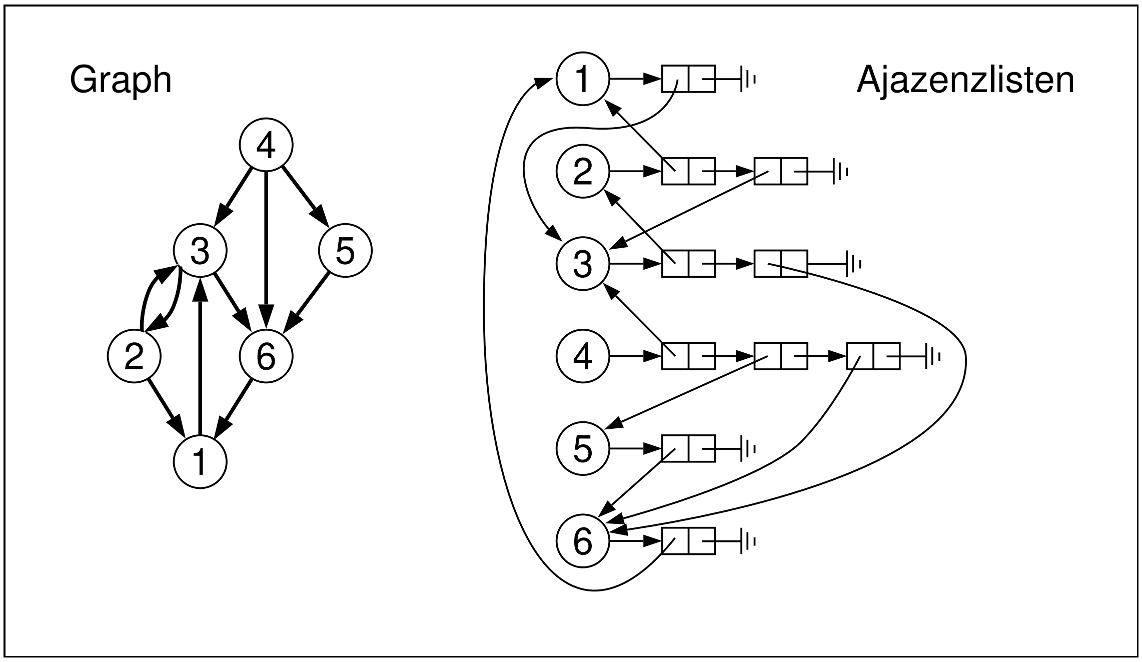

Abbildung 4.4: Repräsentation eines Graphen mit Hilfe von Adjazenzlisten

Adjazenzliste eines Knotens v also alle zu v adjazenten Knoten bzw. einen Verweis auf diese in einer beliebigen Reihenfolge. Im Falle gerichteter Graphen beinhaltet die Adjazenzliste des Knotens v nur die direkten Nachfolger des Knotens v bzw. einen Verweis auf diese in beliebiger Reihenfolge. Manchmal wird für gerichtete Graphen diese Darstellung um eine zweite Adjazenzliste für jeden Knoten erweitert. In dieser zweiten Adjazenzliste eines Knotens v werden dann alle direkten Vorgänger des Knotens v bzw. ein Verweis auf diese in ebenfalls beliebiger Reihenfolge aufgelistet. Man sieht leicht, daß der Speicherplatzbedarf einer Darstellung mittels Adjazenzlisten nur $O(n + m)$ ist. Manchmal spendiert man zusätzlich noch ein Array der Länge n, in dem an Position v ein Verweis auf die Adjazenzliste des Knotens v oder ein Verweis auf den Knoten v selbst eingetragen ist.

Allerdings verhalten sich diese beiden Darstellungen sehr unterschiedlich, wenn man die Laufzeit auf die Frage '$(i, j) \in E$?' betrachtet. Im Falle von Adjazenzmatrizen, benötigen wir nur konstante Zeit, da wir nur die Adjazenzmatrix an Position (i, j) inspizieren müssen. Im Falle von Adjazenzlisten benötigen wir Zeit $O(\Delta(G)) = O(n)$, da wir im schlimmsten Fall die gesamte Adjazenzliste des Knoten i (oder j) durchlaufen müssen.

An dieser Stelle wollen wir noch darauf hinweisen, daß es für spezielle Klassen von Graphen besser geeignetere Darstellungen geben kann. Als Beispiel sei hier nur auf die Repräsentation von binären Bäumen erwähnt, wie wir sie bei den Suchbäumen angegeben haben.

11. Juli 97

4.2 Traversieren von Graphen

Im folgenden stellen wir einige einfache Algorithmen vor, mit denen man einen Graphen einmal ganz durchlaufen kann. Diese Algorithmen bilden eine Grundlage für viele Graphenalgorithmen bzw. werden in vielen Graphenalgorithmen als Unterprozeduren benötigt.

4.2.1 Tiefensuche (DFS)

Die einfachste Art einen Graphen ganz zu durchlaufen ist die *Tiefensuche* oder *DFS* (von engl. *Depth-First-Search*). Wie in einem Labyrinth gehen wir solange durch Gänge (=Kanten) bis wir eine Kreuzung (=Knoten) zum zweiten Mal besuchen. Zu diesem Zweck markiert man jede besuchte Kreuzung mit einer fortlaufenden Nummer (mit Hilfe des Arrays `dfs_num`). Wenn wir einen Knoten zum zweiten Mal besuchen (er also eine Nummer größer 0 hat), gehen wir den Weg den wir gekommen sind wieder soweit zurück, bis wir eine Kreuzung finden, von der aus ein noch nicht besuchter Gang abzweigt. Dabei zählen wir an jedem neu besuchten Knoten eine globale Variable `number` hoch und weisen diesen Wert dem besuchten Knoten zu (in `dfs_num`). Diese bei der Tiefensuche generierte Numerierung der Knoten heißt *DFS-Numerierung* und die entsprechenden Nummern an den Knoten *DFS-Nummern*.

Die Reihenfolge, in der die adjazenten Kanten eines bestimmten Knotens durchlaufen werden, wird dabei von der Tiefensuche nicht vorgegeben. Im Regelfall wird die Implementierung der Datenstruktur `graph` eine bestimmte Reihenfolge nahelegen.

Die Prozedur DFS für gerichtete Graphen ist in Abbildung 4.5 in `C`-ähnlicher Notation angegeben. Offensichtlich läßt sich diese Prozedur auch leicht für ungerichtete Graphen modifizieren. Eine echte Implementierung in `C` hängt von der Definition der Datenstruktur `graph` ab. Zur Übung wird empfohlen, wirklich einmal die Datenstruktur `graph` und den Algorithmus zur Tiefensuche in `C` zu programmieren.

Wir geben im folgenden noch eine andere iterative Implementierung an. Stellen wir uns DFS wieder als ein Durchlaufen eines Labyrinths vor. Wir können nun für jeden Knoten, den wir zum erstenmal betreten, uns die Liste seiner Nachbarknoten in einem Notizbuch notieren. Den nächsten Knoten den wir nun besuchen ist der letzte Knoten im Notizbuch. Bei Erreichen eines Knotens wird dieser aus dem Notizbuch gestrichen.

Ein solches Notizbuch läßt sich mit Hilfe eines Stacks realisieren. Diese Implementierung ist in Abbildung 4.6 angegeben. Man erhält dann im allgemeinen nicht genau dieselbe Reihenfolge der aufgesuchten Knoten wie im rekursiven Algorithmus. Der

```
DFS(graph G)
{
    int dfs_num[|V(G)|], number=0;
    for each (int v ∈ V(G))
        dfs_num[v]=0;

    for each (int v ∈ V(G))
        if (dfs_num[v]==0) /* v has not been visited */
            dfs(G, v, dfs_num, number);
}

dfs(graph G, int v, int dfs_num[], int& number);
{
    dfs_num[v]=++number;
    for each (int w ∈ N+(v))
        if (dfs_num[w]==0) /* w has not been visited */
            dfs(G, w, dfs_number, number)
    }
}
```

Abbildung 4.5: Ein rekursiver Algorithmus zur Tiefensuche

einzige Unterschied besteht lediglich darin, daß bei der Auswahl einer nächsten Kante eine andere Reihenfolge gewählt wird. Wie wir aber bereits oben bemerkt haben, ist diese Reihenfolge für die Tiefensuche nicht genau festgelegt.

Zum Schluß wollen wir noch festhalten, daß bei ungerichteten Graphen jede Kante genau zweimal betrachtet wird. Im Falle gerichteter Graphen wird jede Kante sogar nur genau einmal betrachtet. Damit können wir folgendes Theorem festhalten:

Theorem 4.3 *Das Traversieren eines (un)gerichteten Graphen G mit n Knoten und m Kanten mittels DFS benötigt Zeit $O(n+m)$.*

Die Kanten eines gerichteten Graphen werden durch die Tiefensuche in vier Klassen partitioniert:

Baumkanten (engl. *tree edge*) Eine Baumkante ist dadurch definiert, daß mit dieser Kante $(v, w) \in E$ ein noch nicht besuchter Knoten erreicht wird, also wenn *vor* dem Vergeben der `dfs_num` gilt, daß `dfs_num`$[w] = 0$.

Wie der Name schon sagt, bildet diese Klasse der Kanten einen Wald, der im Falle zusammenhängender Graphen dann natürlich ein Baum ist. Diesen Teilgraphen bezeichnet man oft auch als *spannenden DFS-Baum* oder *spannenden DFS-Wald*.

```
DFS(GRAPH G)
{
    int dfs_num[|V(G)|], number=0;
    for each (int v ∈ V(G))
        dfs_num[v]=0;

    for each (int v ∈ V(G))
        if (dfs_num[v]==0) /* v has not been visited */
            dfs(G, v, dfs_num, number);
}

dfs(graph G, int v, int dfs_num[], int& number);
{
    stack s=empty();
    push(s, v);
    while (!(is_empty(s)))
    {
        v=pop(s);
        if (dfs_num[v]==0) /* v has not been visited */
        {
            dfs_num[v]=++number;
            for each (int w ∈ N+(v))
                push(s, w);
        }
    }
}
```

Abbildung 4.6: Realisierung der Tiefensuche mit Hilfe eines Stacks

Vorwärtskanten (engl. *forward edge*) Eine Kante $(v, w) \in E$ heißt eine Vorwärtskante, wenn sie zu einem bereits besuchten Knoten w führt, so daß es einen Pfad von v nach w gibt, der nur aus Baumkanten besteht. Dies ist äquivalent dazu, daß `dfs_num`$[v] <$ `dfs_num`$[w]$.

Rückkanten (engl. *back edge*) Eine Rückkante ist eine Kante $(v, w) \in E$ zu einem bereits besuchten Knoten w, so daß es einen Pfad von w nach v gibt, der nur aus Baumkanten besteht. Dies ist äquivalent dazu, daß `dfs_num`$[v] >$ `dfs_num`$[w]$ und daß die Prozedur `dfs` aufgerufen am Knoten w noch *nicht* beendet ist.

Querkanten (engl. *cross edge*) Alle übrigen Kanten des Graphen heißen Querkanten. Dies ist äquivalent dazu, daß `dfs_num`$[v] >$ `dfs_num`$[w]$ und daß die Prozedur `dfs` aufgerufen am Knoten w *bereits* beendet ist.

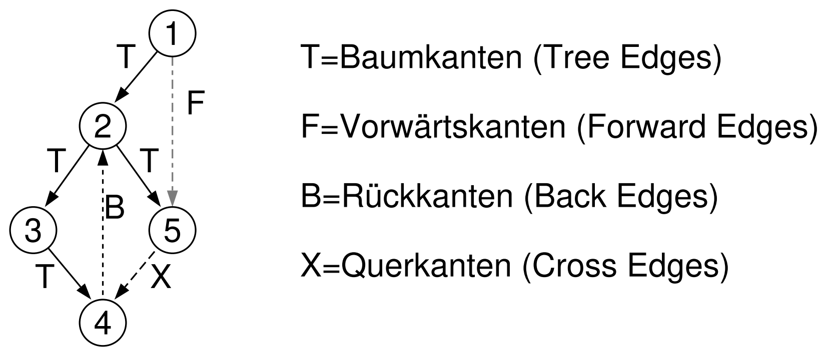

Abbildung 4.7: Ein Beispiel zur Einteilung der Kanten mittels Tiefensuche

Diese Einteilung der Kanten kann auch bei ungerichteten Graphen vorgenommen werden, wobei es dann keine Querkanten geben kann.

Um die Kanten bereits während eines DFS-Laufes erkennen zu können, muß man also wissen an welchen Knoten ein DFS-Durchlauf noch nicht abgeschlossen ist. Dazu kann man z.B. ein Boole'sches Array `active` verwenden, das angibt, ob von einem Knoten noch eine DFS-Suche aktiv ist. Diese Modifikation läßt sich im rekursiven DFS-Algorithmus unmittelbar vornehmen. Bei der iterativen Variante müssen wir merken, wann alle direkten Nachfolger eines Knotens abgearbeitet sind. Zu diesem Zeitpunkt muß dann `active`[v] auf FALSE gesetzt werden. Die Details überlassen wir dem Leser als Übungsaufgabe.

In Abbildung 4.7 ist ein DFS-Durchlauf durch einen Graphen dargestellt. Die Nummern in den Knoten geben die DFS-Numerierung an, und die Kantenbeschriftung die Art der Kanten.

4.2.2 Breitensuche (BFS)

Bei der Tiefensuche versucht man immer erst einmal so tief wie möglich in das Labyrinth vorzustoßen. Im Falle der *Breitensuche* oder *BFS* (von engl. *Breadth-First-Search*) durchläuft man die Kanten in der Reihenfolge des Auffindens. Sobald man auf einen noch nicht besuchten Knoten trifft, notiert man sich die direkten Nachfolger in einem Notizbuch. Die nächste Kante die man durchläuft ist dann die Kante, die zum Knoten führt, der als oberstes im Notizbuch steht. Nach dem Durchlaufen einer Kante streicht man diese Kante aus dem Notizbuch. Analog zur Tiefensuche haben wir jedem Knoten noch eine Zahl zugeordnet, die angibt, als wievielter Knoten er besucht wurde. Diese Zahl wollen wir im folgenden als *BFS-Nummer* bezeichnen.

```
BFS(graph G)
{
    int bfs_num[|V(G)|], number=0;
    for each (int v ∈ V(G))
        bfs_num[v]=0;

    for each (int v ∈ V(G))
        if (bfs_num[v]==0) /* v has not been visited */
            bfs(G, v, bfs_num, number);
}

bfs(graph G, int v, int bfs_num[], int& number);
{
    queue q=empty();
    enqueue(q,v);
    while (!(is_empty(q)))
    {
        v=dequeue(q);
        if (bfs_num[v]==0) /* v has not been visited */
        {
            bfs_num[v]=++number;
            for each (int w ∈ N+(v))
                enqueue(q, w);
        }
    }
}
```

Abbildung 4.8: Realisierung der Breitensuche mit Hilfe einer Queue

Damit erkennt man, daß DFS und BFS im wesentlichen derselbe Algorithmus zugrunde liegt. Nur beim Speichern der noch zu besuchenden Knoten wird eine andere Strategie verwendet: LIFO (last in first out) bei DFS und FIFO (first in first out) bei BFS. Man kann also bei einer geeigneten konkreten Realisierung der beiden Prozeduren, durch Austauschen der Datenstrukturen zum Speichern der noch zu besuchenden Kanten ineinander überführen: Stacks für DFS und Queues für BFS.

Zum Schluß können wir wie bei der Tiefensuche festhalten, daß bei ungerichteten Graphen jede Kante genau zweimal und bei gerichteten Graphen jede Kante sogar nur genau einmal betrachtet wird. Damit erhalten wir folgendes Theorem:

Theorem 4.4 *Das Traversieren eines (un)gerichteten Graphen G mit n Knoten und m Kanten mittels BFS benötigt Zeit $O(n+m)$.*

15. Juli 97

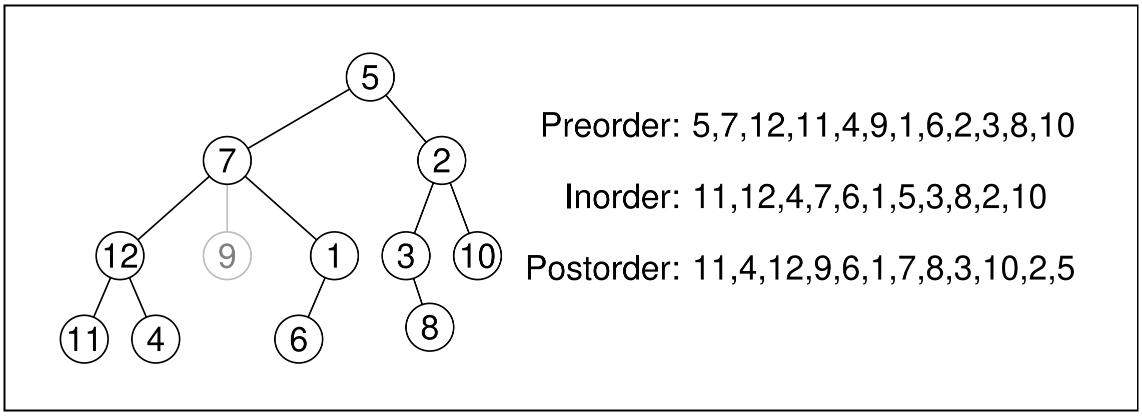

Abbildung 4.9: Beispiele von Preorder, Inorder, und Postorder Traversierung

4.2.3 Traversieren von Bäumen (*)

Zum Abschluß wollen wir noch Durchlauftechniken für spezielle Graphen, nämlich geordnete gewurzelte Bäume angeben. Eine *Traversierung* eines geordneten gewurzelten Baumes ist die Auflistung seiner Knoten in einer bestimmten vorgegebenen Reihenfolge.

Die Traversierung in *Preorder* des geordneten gewurzelten Baumes $T = (V, E)$ ist rekursiv wie folgt definiert. Sei $r(T)$ die Wurzel des Baumes T und $T_1, \ldots, T_r$ alle an den Kindern der Wurzel gewurzelten Teilbäume. Dann gibt man zuerst die Wurzel $r(T)$ aus und traversiert anschließend die gewurzelten Bäume $T_1, \ldots, T_r$ in der Ordnung der Kinder von $r(T)$.

Die Traversierung in *Postorder* des geordneten gewurzelten Baumes $T = (V, E)$ ist rekursiv wie folgt definiert. Sei $r(T)$ die Wurzel des Baumes T und $T_1, \ldots, T_r$ alle an den Kindern der Wurzel gewurzelten Teilbäume. Man traversiert zuerst die gewurzelten Bäume $T_1, \ldots, T_r$ in der Ordnung der Kinder von $r(T)$ und gibt zum Schluß die Wurzel $r(T)$ des Baumes aus.

Die Traversierung in *Inorder* des binären geordneten gewurzelten Baumes $T = (V, E)$ ist rekursiv wie folgt definiert. Sei $r(T)$ die Wurzel des Baumes T sowie T_ℓ und T_r der linke und rechte Teilbaum von $r(T)$, die durchaus leer sein können. Nun traversiert man zuerst den 'linken' Teilbaum T_ℓ, gibt dann die Wurzel $r(T)$ aus und traversiert zum Schluß den rechten Teilbaum T_r.

In Abbildung 4.9 ist für einen Baum die Preorder, Inorder und Postorder angegeben. Man beachte, daß die Preorder auch eine zulässige DFS-Numerierung liefert. Eine Auflistung der Knoten in Preorder, Inorder oder Postorder allein genügt leider nicht, um daraus die Struktur des Baumes zu erkennen. Man kann allerdings zeigen, daß die Auflistungen in Preorder und Postorder eines Baumes genügen, um den Baum zu rekonstruieren. Man kann diese Traversierungen auch für freie Bäume verwenden. Dann wählt man irgendeinen Knoten als Wurzel aus und richtet alle Kanten von der Wurzel weg. Auf den Kindern eines jedes Knotens wird dann noch eine beliebige Ordnung definiert.

4.3 Zusammenhang von Graphen

Nun wollen wir uns überlegen, wie man die Zusammenhangskomponenten von Graphen ermitteln kann.

4.3.1 Ungerichtete Graphen

Betrachten wir zunächst den einfachen Fall ungerichteter Graphen. Nach Definition ist eine Zusammenhangskomponente ein maximaler zusammenhängender Teilgraph, wobei zusammenhängend bedeutet, daß zwischen je zwei Knoten ein Pfad existiert. In der DFS-Prozedur haben wir aber gerade solche Zusammenhangskomponenten abgelaufen. Jedesmal wenn die Prozedur `dfs` in der Hauptprozedur `DFS` beendet wurde, haben wir eine Zusammenhangskomponente gefunden. Mit dem Suchen nach dem nächsten noch nicht besuchten Knoten in `DFS` haben wir einen Knoten in einer neuen Zusammenhangskomponente bestimmt.

Theorem 4.5 *Mittels DFS lassen sich in Zeit $O(n+m)$ die Zusammenhangskomponenten eines ungerichteten Graphen mit n Knoten und m Kanten bestimmen.*

4.3.2 Gerichtete Graphen

Die schwachen Zusammenhangskomponenten eines gerichteten Graphen sind gerade die Zusammenhangskomponenten des entsprechenden ungerichteten Graphen. Mittels eines DFS-Durchlaufs können wir jedoch sehr einfach einen gerichteten Graphen in einen ungerichteten verwandeln. Für jeden Knoten w in einer Adjazenzliste des Knotens v hängen wir an die Adjazenzliste des Knotens w den Knoten v an. Dann können wir den Algorithmus für Zusammenhang für ungerichtete Graphen auf dem modifizierten Graphen laufen lassen.

Theorem 4.6 *Mittels DFS lassen sich in Zeit $O(n+m)$ die schwachen Zusammenhangskomponenten eines gerichteten Graphen mit n Knoten und m Kanten bestimmen.*

Die Bestimmung der starken Zusammenhangskomponenten eines gerichteten Graphen ist etwas komplizierter, da wir nicht nur feststellen müssen, ob wir von v nach w im Graphen gelangen können (was ja der DFS-Algorithmus leistet), sondern gleichzeitig feststellen müssen, ob man auch von w nach v zurückgelangt.

Zunächst einmal nehmen wir ohne Beschränkung der Allgemeinheit an, daß der betrachte Graph G schwach zusammenhängend ist. Andernfalls betrachten wir jede schwache Zusammenhangskomponente getrennt, da jede starke Zusammenhangskomponente ein Teilgraph einer schwachen Zusammenhangskomponente ist. Zuerst stellen wir fest, daß die Baumkanten einer starken Zusammenhangskomponente selbst wieder zusammenhängend sind. Also bildet die starke Zusammenhangskomponente einen (zusammenhängenden) Teilbaum des spannenden DFS-Baumes. Im folgenden werden wir basierend auf einer DFS-Traversierung solche Teilbäume bestimmen, die zu einer starken Zusammenhangskomponente korrespondieren. Da der DFS-Wald gewurzelt ist, nennen wir die Wurzel eines Teilbaumes, der zu einer starken Zusammenhangskomponente korrespondiert, die *Wurzel* dieser starken Zusammenhangskomponente.

Für $v, w \in V$ bezeichne $v \leftrightarrow w$, daß es in einem Graphen $G = (V, E)$ einen gerichteten Pfad von v nach w und einen gerichteten Pfad von w nach v gibt. Mit Hilfe dieser Notation können wir nun für einen gerichteten Graphen $G = (V, E)$ die folgende nützliche Hilfsfunktion $\mathit{LowLink} : V \to \mathbb{N}$ definieren:

$$\mathit{LowLink}(v) = \min \left\{ \mathtt{dfs\_num}[w] \;\middle|\; \begin{array}{l} \big(v = w\big) \;\vee \\ \big(v \leftrightarrow w \;\wedge \exists p = (v_0, \ldots, v_n) : v_0 = v \wedge v_n = w \\ \quad \wedge\, \forall i \in [1 : n-1] :\, (v_{i-1}, v_i) \text{ ist Baumkante} \\ \quad \wedge\, (v_{n-1}, v_n) \text{ ist Rück- oder Querkante}\big) \end{array} \right\}$$

Wir halten hier noch fest, daß gilt: $\mathit{LowLink}(v) \leq \mathtt{dfs\_num}[v]$. Mit Hilfe dieser Funktion *LowLink* können wir die Wurzeln von starken Zusammenhangskomponenten charakterisieren.

Lemma 4.7 *Sei $G = (V, E)$ ein gerichteter Graph. Der Knoten $v \in V$ ist genau dann die Wurzel einer starken Zusammenhangskomponente, wenn $\mathit{LowLink}(v) = \mathtt{dfs\_num}[v]$.*

Beweis: $\Rightarrow$: Für einen Widerspruchsbeweis nehmen wir an, daß $\mathit{LowLink}(v) < \mathtt{dfs\_num}[v]$ ist. Nach Definition von *LowLink* gibt es dann einen Knoten w mit kleinerer DFS-Nummer als v (nämlich $\mathit{LowLink}(v)$), so daß es einen gerichteten Pfad von v nach w und einen gerichteten Pfad von w nach v gibt. Da eine starke Zusammenhangskomponente einen Teilbaum des DFS-Waldes bildet und die Wurzel dieses Teilbaumes die kleinste DFS-Nummer des Baumes hat, muß die Wurzel eine DFS-Nummer kleiner gleich $\mathtt{dfs\_num}[w] < \mathtt{dfs\_num}[v]$ haben. Dies steht im Widerspruch

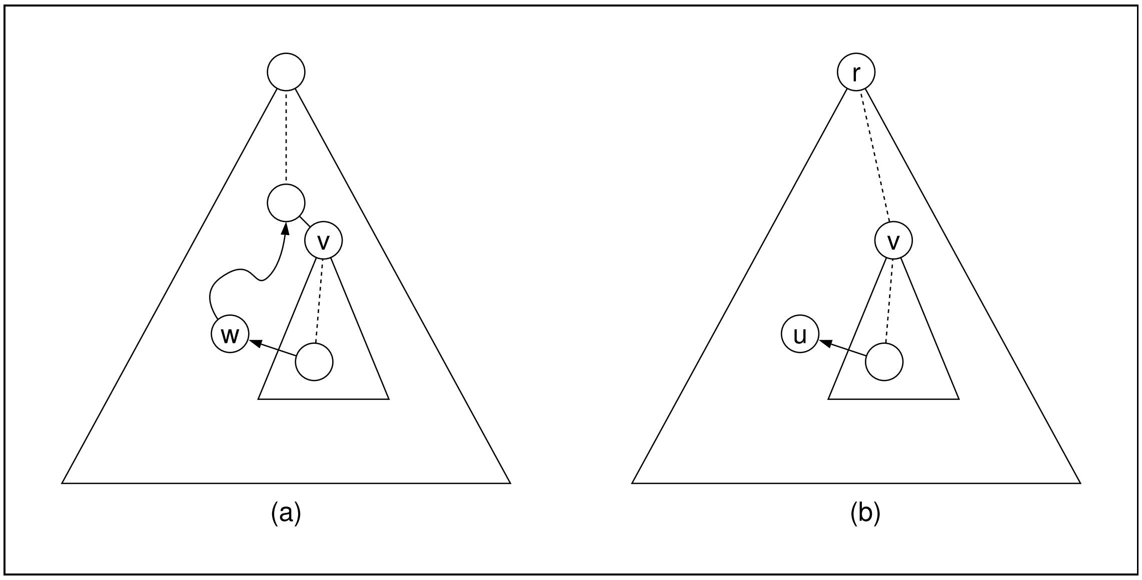

Abbildung 4.10: $LowLink(v) < \texttt{dfs\_num}[v]$ impliziert, daß v keine Wurzel einer starken Zusammenhangskomponente sein kann (a), Ist r die Wurzel der starken Zusammenhangskomponente von v, dann muß $LowLink(v) < \texttt{dfs\_num}[v]$ sein (b)

zur Annahme, daß v die Wurzel der starken Zusammenhangskomponente ist. Diese Argumentation ist auch noch einmal in Abbildung 4.10 (a) illustriert.

$\Leftarrow$: Für einen Widerspruchsbeweis sei $r \neq v$ die Wurzel der starken Zusammenhangskomponente in der v liegt. Dann gibt es einen gerichteten Pfad p von v nach r. Dieser Pfad p muß irgendwann über eine Kante e den Teilbaum gebildet aus den Nachfolgern von v im DFS-Baum verlassen. Diese Kante e muß eine Rück- oder Querkante sein. Der Endknoten der Kante e muß nun ebenfalls in der starken Zusammenhangskomponente von v liegen. Nach Definition der Tiefensuche ist die DFS-Nummer des Endknotens von e kleiner als die DFS-Nummer von v und damit erhalten wir $LowLink(v) < \texttt{dfs\_num}[v]$ und somit einen Widerspruch zur Voraussetzung. Diese Argumentation ist auch noch einmal in Abbildung 4.10 (b) illustriert. ∎

Damit erhalten wir den Abbildung 4.11 angegebenen Algorithmus. Wir werden zunächst zeigen, daß der Algorithmus auch korrekt ist.

Lemma 4.8 *Der Algorithmus SCC zur Berechnung der starken Zusammenhangskomponenten eines gerichteten Graphen ist korrekt*

Beweis: Zur Korrektheit genügt es zu zeigen, daß die Werte `low_link`$[v]$ für alle Knoten in v korrekt berechnet werden. Wir führen den Beweis mittels vollständiger Induktion über die Anzahl der Knoten im Graphen, bei der die Prozedur `Strong`

```
SCC(graph G)
{
    stack s=empty();
    int dfs_num[|V(G)|], number=0;
    int low_link[|V(G)|];
    for each (int v ∈ V(G))
        low_link[v]=dfs_num[v]=0;

    for each (int v ∈ V(G))
        if (dfs_num[v]==0)
            Strong(G, v, dfs_num, number, low_link);
}

Strong(graph G, int v, int dfs_num[], int& number, int low_link[]);
{
    low_link[v]=dfs_num[v]=++number;
    push(s,v);
    for each (int w ∈ N+(v))
        if (dfs_num[w]==0)
        {   /* (v, w) is a tree edge */
            Strong(G, w, dfs_num, number, low_link);
            low_link[v]=min(low_link[v],low_link[w]);
        }
        else if ((dfs_num[w]<dfs_num[v]) && (w ∈ s))
                /* (v, w) is a back or cross edge */
                /* within a strongly connected component */
                low_link[v]=min(low_link[v],dfs_num[w]);

    if (low_link[v]==dfs_num[v]) /* v is the root of a SCC */
    {   /* Output: all vertices including v on the top of the stack s */
        /* form a strongly connected component; */
        output 'SCC: ';
        do {
                w=pop(s);
                output w;;
            } while (w!=v);
    }
}
```

Abbildung 4.11: Algorithmus zur Bestimmung der starken Zusammenhangskomponenten eines gerichteten Graphen beendet worden ist. Bezeichne im folgenden n diese Anzahl von abgearbeiteten Knoten.

Induktionsanfang ($n = 0$): Trivialerweise ist der Induktionsanfang erfüllt, da für keine Knoten etwas gelten muß.

Induktionsschritt ($n - 1 \to n$): Sei im folgenden v, der n-te Knoten an dem die Prozedur `Strong` beendet wird.

Wir zeigen zunächst, daß bei der Minimumbildung nur erlaubte DFS-Nummern berücksichtigt werden. Daraus folgt dann, daß $LowLink(v) \leq \texttt{low\_link}[v]$.

Im **if**-Teil der Anweisung wird zunächst `Strong` für einen direkten Nachfolger w aufgerufen. Da die Prozedur `Strong` am Knoten w dazu bereits beendet sein muß, wissen wir nach Induktionsvoraussetzung, daß $\texttt{low\_link}[w]$ korrekt berechnet wurde. Ist $\texttt{low\_link}[w] \geq \texttt{dfs\_num}[v]$, so wird korrekterweise $\texttt{low\_link}[v]$ nicht verändert, da ja $\texttt{low\_link}[v]$ mit $\texttt{dfs\_num}[v]$ initialisiert wurde. Es bleibt noch für den Fall $\texttt{low\_link}[w] < \texttt{dfs\_num}[v]$ zu zeigen, daß die Betrachtung des Wertes $\texttt{low\_link}[w]$ zulässig ist. Sei u der Knoten mit der DFS-Nummer $\texttt{low\_link}[w]$. Da es einen Pfad von w nach u nur über Baumkanten gefolgt von einer Rück- oder Querkante gibt, gibt es einen solchen auch von v aus. Wir müssen also nur noch zeigen, daß $v \leftrightarrow u$ gilt. Sei r die Wurzel der starken Zusammenhangskomponente, in der sich u und w befinden. Da eine starke Zusammenhangskomponente einen Teilbaum T' des DFS-Baumes bildet, muß r ein Vorgänger von w und u sein. Da $\texttt{dfs\_num}[u] < \texttt{dfs\_num}[w]$, muß außerdem $r \neq w$ gelten. Weil aber v direkter Vorgänger von w im Teilbaum T' ist, muß v ebenfalls in diesem Teilbaum T' enthalten sein. Damit gilt $v \leftrightarrow u$. Also ist die Berücksichtigung der DFS-Nummer von w in der Bestimmung von $\texttt{low\_link}[v]$ korrekt.

Im **else**-Teil der Anweisung ist die Kante von v nach w eine Rück- oder Querkante, wobei w noch nicht aus dem Stack gepoppt wurde. Sei r die Wurzel der starken Zusammenhangskomponente, in der sich w befindet. Falls r kein Vorgänger von v im DFS-Baum ist, so wäre die Prozedur `Strong` am Knoten r bereits beendet, da $\texttt{dfs\_num}[r] < \texttt{dfs\_num}[v]$. Da nach Induktionsvoraussetzung $\texttt{low\_link}[r]$ bereits korrekt berechnet worden ist, wären die Knoten dieser Zusammenhangskomponente inklusive w bereits ausgegeben worden, d.h. w wäre nicht mehr auf dem Stack. Also muß r ein Vorgänger von v sein und damit gibt es einen Weg von r nach v. Offensichtlich gibt es einen Weg von v nach r, da $w \leftrightarrow r$. Damit ist auch in diesem Fall die Berücksichtigung der DFS-Nummer von w in der Bestimmung von $\texttt{low\_link}[v]$ korrekt.

Damit haben wir nun $LowLink(v) \leq \texttt{low\_link}[v]$ bewiesen. Es bleibt nur noch zu zeigen, daß das Minimum auch angenommen wird, also $LowLink(v) = \texttt{low\_link}[v]$. Sei $(u, u') \in E$ die Rück- oder Querkante, die für die Bildung des Wertes von $LowLink(v)$ verantwortlich ist, also $LowLink(v) = \texttt{dfs\_num}[u']$. Ist $u = v$, so wird offensichtlich im **else**-Teil der Wert von `low_link`$[v]$ korrekt gesetzt. Also muß $u \neq v$ sein. Sei nun w der direkte Nachfolger von v, der Vorgänger von u ist. Da aus $v \leftrightarrow u'$ folgt, daß auch $w \leftrightarrow u'$, muß bereits `low_link`$[w]$ falsch berechnet worden sein, denn ansonsten wäre `low_link`$[v]$ vom Algorithmus korrekt berechnet worden. Dies steht aber im Widerspruch zur Induktionsvoraussetzung, nach der `low_link`$[w]$ bereits korrekt berechnet worden ist. ∎

Da die Prozedur SCC im wesentlichen auf DFS basiert, folgt unmittelbar die Abschätzung der Laufzeit mit $O(n+m)$. Damit erhalten wir das folgende Theorem:

Theorem 4.9 *Die Prozedur SCC bestimmt in Zeit $O(n+m)$ die starken Zusammenhangskomponenten eines gerichteten Graphen mit n Knoten und m Kanten.*

18. Juli 97

4.4 Kürzeste Wege

In diesem Abschnitt wollen wir kürzeste Wege in Graphen berechnen. Dazu erweitern wir die Graphen um eine Gewichtsfunktion γ auf den Kanten des Graphen:

$$\gamma : E \to \mathbb{R} \cup \{-\infty, \infty\}.$$

Die Gewichte auf den Kanten wollen wir als 'Länge' einer Kante interpretieren. Die gewichtete Länge eines Pfades $p = (v_0, \ldots, v_n)$ der Länge n ist nun die Summe der Kantengewichte:

$$\gamma(p) = \sum_{i=1}^{n} \gamma((v_{i-1}, v_i)).$$

Die Ziele dieses Abschnittes sind es nun, die folgenden Fragen zu beantworten:

1. Für ein gegebenes Paar von Knoten $v, w \in V$ bestimme den kürzesten Weg, d.h. den Pfad mit geringster gewichteter Länge, von v nach w.
2. Für einen festen Knoten $v \in V$ bestimme die kürzesten Wege, d.h. die Pfade mit geringster gewichteter Länge, zu allen anderen Knoten in V.
3. Für alle Paare $v, w \in V$ bestimme die kürzesten Wege, d.h. die Pfade mit geringster gewichteter Länge, von v nach w.

Wir werden im folgenden Antworten zu Fragen 2 und 3 geben. Interessanterweise ist kein Algorithmus für Frage 1 bekannt, der nicht gleich auch Frage 2 mitbeantwortet. Für die Beantwortung obiger Fragen, genügt es offensichtlich sich auf schwach zusammenhängende Graphen zu beschränken. Wie wir ja in Abschnitt 4.3 gesehen haben, kann man die schwachen Zusammenhangskomponenten ja effizient berechnen.

4.4.1 Floyd-Algorithmus

Der folgende Algorithmus geht auf R.W. Floyd zurück und wird deshalb oft als *Floyd-Algorithmus* bezeichnet. S. Warshall hat einen sehr ähnlichen Algorithmus zur Bestimmung der transitiven Hülle von Graphen angegeben (siehe auch Abschnitt 4.4.2), so daß der hier vorgestellte Algorithmus oft auch *Floyd-Warshall-Algorithmus* genannt wird.

In diesem Abschnitt werden wir annehmen, daß die betrachteten Graphen *vollständig* sind, d.h. je zwei Knoten sind mit einer Kante verbunden. Damit sind die Einträge der Adjazenzmatrix eines solchen Graphen alle 1. Die Menge der Knoten des Graphen $G = (V, E)$ sei wieder $V = \{0, \ldots, n-1\}$. Dann ist die *Gewichtsmatrix* $\Gamma(G)$ eines Graphen $G = (V, E)$ eine Matrix, die als Einträge gerade die Gewichte der entsprechenden Kante hat, also $\Gamma_{ij} = \gamma((i, j))$.

In diesem Abschnitt erlauben wir auch negative Kantengewichte. Wir wollen aber gerichtete Kreise mit einem negativen Gewicht ausschließen. In diesem Fall gibt es nämlich keine kürzesten Wege, da wir einen solchen Kreis beliebig oft durchlaufen können, und so das Gewicht eines Pfades beliebig klein machen können.

Der Algorithmus basiert im wesentlichen auf der folgenden Definition einer Folge von Matrizen $D^{(k)}_{i,j}$. Dabei ist $D^{(k)}_{i,j}$ definiert als die gewichtete Länge des kürzesten gewichteten Pfades p zwischen den Knoten i und j, wobei der Pfad p nur Knoten $v < k$ besuchen darf. Einzige Ausnahme bilden dabei der Anfangs- und Endknoten des Pfades p, die beliebig sein dürfen. Offensichtlich ist die Matrix $D^{(0)}_{i,j} = \gamma((i, j))$ gerade die Gewichtsmatrix des Graphen. Die Matrix $D^{(n)}_{i,j}$ ist dann also genau die Matrix, die als Einträge die kürzesten gewichteten Wege von i nach j im Graphen G als Eintrag an der Position (i, j) enthält. Dann ist nämlich die Einschränkung an die betrachteten Pfade hinfällig. Das Ziel ist es nun, aus der Gewichtsmatrix $D^{(0)}_{i,j}$ die benötigte Matrix $D^{(n)}_{i,j}$ zu berechnen.

Grundgedanke des Algorithmus ist die folgende Beobachtung. Der kürzeste Pfad von i nach j, der Knoten kleiner als $k+1$ besucht, ist offensichtlich entweder die Kombination aus dem kürzesten Weg von i nach k, der nur Knoten kleiner als k besucht, und dem kürzesten Pfad von k nach j, der nur Knoten kleiner als k besucht, oder aber der Pfad von i nach j, der Knoten kleiner als k besucht, ist kürzer als die Pfade mit Umweg über den Knoten k. Da alle Kreise im Graphen nichtnegative gewichtete Länge haben, genügt es Pfade zu betrachten, die den Knoten k nur einmal besuchen. Damit erhalten wir die folgende Rekursionsgleichung:

$$D^{(k+1)}_{i,j} = \min\{D^{(k)}_{i,j}, D^{(k)}_{i,k} + D^{(k)}_{k,j}\}$$

Zur Veranschaulichung dieser Argumentation siehe Abbildung 4.12. Diese Rekursionsgleichung läßt sich leicht in den folgenden Algorithmus, der in Abbildung 4.13

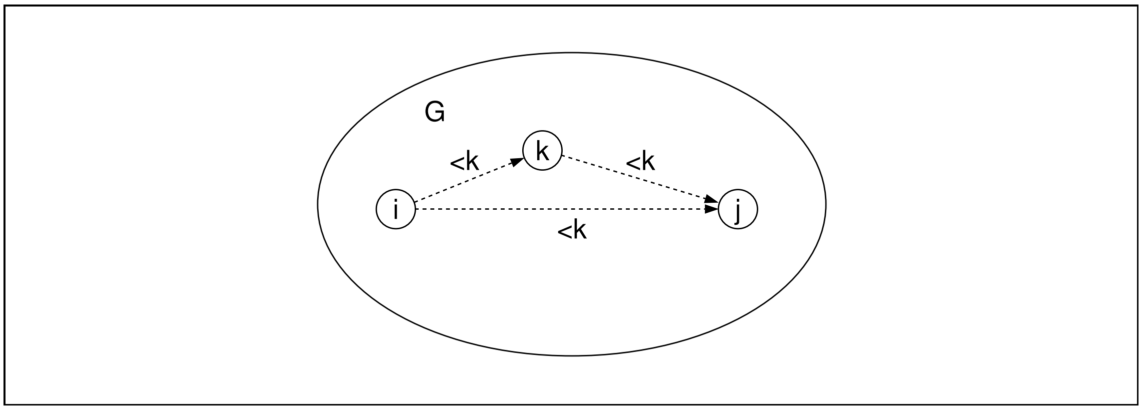

Abbildung 4.12: Kürzester Weg von i nach j über Knoten mit Label $\leq k$

angegeben ist, umsetzten. Offensichtlich wird die Minimumbildung n^3-mal ausgeführt.

Theorem 4.10 *Sei G ein Graph mit einem Kantengewicht γ, wobei jeder gerichtete Kreis eine nichtnegative gewichtete Länge hat. Dann lassen sich die gewichteten Längen der kürzesten Pfade für alle möglichen Paare von Knoten in Zeit $O(n^3)$ berechnen.*

Bei der Entwicklung des Floyd-Algorithmus haben wir ein wichtiges Paradigma zum Erstellen von Algorithmen verwendet, daß sogenannte *Dynamische Programmieren*. Die Grundidee hierbei ist, eine Lösung des Gesamtproblems aus den Lösungen von vielen Teilproblemen zu konstruieren. Dabei spielten die Elemente $D_{i,j}^{(k)}$ die Rolle der Teilprobleme und $D_{i,j}^{(n)}$ war das zu lösende Gesamtproblem. Dabei muß man allerdings wissen, welche Teilprobleme man lösen muß, um die gesuchte Lösung des

```
FLOYD(INT γ[][], INT N)
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            D_{i,j}^{(0)} = γ(i, j);
    for (int k = 0; k < n; k++)
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                D_{i,j}^{(k+1)} = min(D_{i,j}^{(k)}, D_{i,k}^{(k)} + D_{k,j}^{(k)});
}
```

Abbildung 4.13: Der Floyd-Algorithmus zur Bestimmung kürzester Wege

```
FLOYD2(INT γ[][], INT N)
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            D_{i,j} = γ(i, j);
    for (int k = 0; k < n; k++)
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                D_{i,j} = min(D_{i,j}, D_{i,k} + D_{k,j});
}
```

Abbildung 4.14: Speicherplatzsparende Version des Floyd-Algorithmus zur Bestimmung kürzester Wege

Gesamtproblems aus diesen zusammenzusetzen zu können. Im allgemeinen kann es dabei passieren, daß man eine exponentielle Zahl von Teilproblemen lösen muß, wenn man nicht weiß auf welche Teilprobleme man sich beschränken kann. Man beachte, daß Programmierung hier nicht im üblichen Sinne zu verstehen ist, sondern eher als Ausfüllen eines Tableaus.

Wenn man genau hinschaut hat der Algorithmus von Floyd allerdings auch einen Platzbedarf von $O(n^3)$. Dieser läßt sich jedoch senken. Lassen wir bei der Folge von Matrizen, die Indizes (k) weg, so erhalten wir einen Algorithmus mit kubischer Laufzeit und quadratischem Platzbedarf. Diese Variante ist in Abbildung 4.14 angegeben. Wir müssen uns nur noch überlegen, daß dieser Algorithmus korrekt ist. In der Rekursion wird nun bei der Minimumbildung eventuell statt dem Wert $D_{i,j}^{(k)}$ bereits der Wert $D_{i,j}^{(k+1)}$ herangezogen, wenn er denn schon neu berechnet wurde. Dies ist bei den benötigten Werten $D_{i,k}^{(*)}$ bzw. $D_{k,j}^{(*)}$ der Fall, wenn j bzw. i größer als k ist. In diesem Fall werden dann bei der Berechnung des neuen kürzesten gewichteten Pfades zwischen i und j, der Knoten kleiner als $k+1$ besuchen darf, bereits Pfade von und nach k berücksichtigt, die selbst schon den Knoten k beinhalten. Da aber bei den betrachteten Pfaden ein Endknoten der Knoten k ist, spielt dies keine Rolle, da wir angenommen haben, daß Kreise eine nichtnegative gewichtete Länge haben, und solche Kreise um k daher genau gewichtete Länge 0 haben müssen. Daher berechnen wir im k-ten Durchlauf immer noch die gewichteten Längen von Pfaden von i nach j, die Knoten kleiner als $k+1$ benutzen. Also bleibt der Algorithmus korrekt.

Theorem 4.11 *Sei G ein Graph mit einem Kantengewicht γ, wobei jeder gerichtete Kreis eine nichtnegative gewichtete Länge hat. Dann lassen sich die gewichteten Längen der kürzesten Pfade für alle möglichen Paare von Knoten in Zeit $O(n^3)$ mit einem Speicherplatzbedarf von $O(n^2)$ berechnen.*

Man kann den obigen Algorithmus noch so verändern, daß er nicht nur die gewichteten Längen der kürzesten Pfade berechnet, sondern auch gleich noch kürzeste Pfade mitkonstruiert.

Wir wollen an dieser Stelle auch noch darauf hinweisen, daß sich der obige Algorithmus auch für unvollständige Graphen modifizieren läßt. Man fügt alle nicht vorhandenen Kanten zu dem Graphen hinzu und setzt das Kantengewicht für solche Kanten auf $+\infty$.

22. Juli 97

4.4.2 Transitive Hülle von Graphen (Ü)

In diesem Abschnitt wollen wir die reflexive transitive Hülle eines gerichteten Graphen $G = (V, E)$ berechnen. In Worten gesagt, wollen wir zwischen zwei Knoten i und j eine Kante hinzufügen, wenn i der Anfangspunkt und j der Endpunkt eines gerichteten Pfades im Graphen G ist.

Definition 4.12 *Sei $G = (V, E)$ ein (un)gerichteter Graph. Dann heißt der Graph $G^* = (V, E^*)$* transitiver, reflexiver Abschluß *des Graphen G, wenn G^* der kleinste Graph (bzgl. $\sqsubseteq$) ist, für den gilt:*

- $\forall v \in V\colon\ (v, v) \in E^*,$ *(Reflexivität)*
- $\forall u, v, w \in V\colon\ (u, v) \in E \wedge (v, w) \in E^* \Rightarrow (u, w) \in E^*.$ *(Transitivität)*

Im Falle ungerichteter Graphen ist dabei jede Kante (v, w) als $\{v, w\}$ zu interpretieren.

Der Floyd-Algorithmus läßt sich auch ganz leicht modifizieren, so daß er zur Berechnung des reflexiven, transitiven Abschlusses von (un)gerichteten Graphen verwendet werden kann. Sei analog wie beim Floyd-Algorithmus $A_{i,j}^{(k)} = 1$, wenn es einen (un)gerichteten Pfad von i nach j gibt, der nur Knoten kleiner als k besucht und $A_{i,j}^{(k)} = 0$ sonst. Wiederum ist offensichtlich $A_{i,j}^{(0)} = A_{i,j}$, wobei A die Adjazenzmatrix des gegebenen (un)gerichteten Graphen G ist. Ähnlich wie beim Floyd-Algorithmus erhalten wir die folgende Rekursionsgleichung

$$A_{i,j}^{(k+1)} = A_{i,j}^{(k)} \vee (A_{i,k}^{(k)} \wedge A_{k,j}^{(k)}),$$

wobei $\wedge$ das logische Und sowie $\vee$ das logische Oder bezeichnet. Setzt man diese Rekursion an die entsprechende Stelle des Floyd-Algorithmus ein, so erhält man den sogenannten *Warshall-Algorithmus* zur Bestimmung des reflexiven, transitiven Abschlusses eines (un)gerichteten Graphen. Auch hier ist die Betrachtung der Indizes (k) analog wie beim Floyd-Algorithmus nicht nötig. Der Warshall-Algorithmus ist in Abbildung 4.15 angegeben.

```
WARSHALL(INT A[][], INT N)
{
    bool A*[n][n];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            A*_{i,j} = (A_{i,j} == 1) ∨ (i == j);
    for (int k = 0; k < n; k++)
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                A*_{i,j} = A*_{i,j} ∨ (A*_{i,k} ∧ A*_{k,j});
}
```

Abbildung 4.15: Warshall-Algorithmus zur Bestimmung des reflexiven, transitiven Abschlusses

Theorem 4.13 *Der Warshall-Algorithmus berechnet den reflexiven, transitiven Abschluß eines (un)gerichteten Graphen G mit $2n^3$ Boole'schen Operationen und Platzbedarf $O(n^2)$.*

Wir können hier auch eine andere Variante verwenden. Sei hierzu $B^{(k)}_{i,j} = 1$, wenn es einen Pfad der Länge maximal k von i nach j gibt, und $B^{(k)}_{i,j} = 0$ sonst. Dann gilt offensichtlich die folgende Rekursionsgleichung:

$$B^{(k+\ell)}_{i,j} = \bigvee_{m=0}^{n-1} (B^{(k)}_{i,m} \wedge B^{(\ell)}_{m,j}).$$

Diese besagt nichts anderes, als daß es genau dann einen Pfad der Länge $k+\ell$ von i nach j gibt, wenn es einen Pfad der Länge k von i nach m und einen Pfad der Länge ℓ von m nach j für ein beliebiges $m \in V$ gibt. $B^{(1)}_{i,j}$ ist dann gerade die Adjazenzmatrix erweitert um die Einträge $B^{(1)}_{i,i} = 1$. Gesucht ist dann die Matrix $B^{(n-1)}_{i,j}$. Setzt man $\ell = 1$, erhält man die folgende in Abbildung 4.16 angegebene Version eines Algorithmus zur Berechnung der reflexiven, transitiven Hülle. Dieser Algorithmus hat allerdings eine Laufzeit von $O(n^4)$. Dabei haben wir wieder die oberen Indizes an der Matrix B weggelassen.

Wenn man in jeder Iteration über k, die Länge der Pfade verdoppelt, also in der Rekursionsgleichung $\ell = k$ statt 1 wählt, so erhalten wir einen effizienteren Algorithmus, der in Abbildung 4.17 angegeben ist. Damit hat dieser Algorithmus nun Laufzeit $O(n^3 \log(n))$, was leider immer noch schlechter ist als die Zeitkomplexität vom Warshall-Algorithmus.

Der Trick ist nun, daß man erkennt, daß die drei **for**-Schleifen über i, j und m gerade der Berechnung eines Matrizenprodukts entsprechen, wenn man das logische Oder

```
TransClosure(int A[][], int n)
{
    bool B[n][n];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            B_{i,j} = (A_{i,j} == 1) ∨ (i == j);
    for (int k = 0; k < n; k++)
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
            {
                bool b = FALSE;
                for (int m = 0; m < n; m++)
                    b = b ∨ (B_{i,m} ∧ (A_{m,j} == 1 ∨ m == j));
                    /* Note that B^{(0)}_{m,j} = (A_{m,j} == 1 ∨ m == j) */
                B_{i,j} = b;
            }
}
```

Abbildung 4.16: Ein n^4-Algorithmus zur Bestimmung des reflexiven, transitiven Abschlusses

als Addition und das logische Und als Multiplikation interpretiert. Damit kann man das Problem auf die Matrizenmultiplikation zurückführen. Ist B die Adjazenzmatrix eines Graphen, so müssen wir also nur B^n berechnen. Mit Hilfe des iterierten Quadrierens (siehe Theorem 1.7) läßt sich die Matrix $B^{(n-1)}$ mit $O(\log(n))$ vielen Matrizenmultiplikationen und Matrizenadditionen berechnen. Damit erhalten wir das folgende Theorem:

Theorem 4.14 *Sei $\mathcal{M}(n)$ die Komplexität um zwei Boole'sche $(n \times n)$-Matrizen zu multiplizieren. Dann läßt sich der reflexive transitive Abschluß eines Graphen mit n Knoten in Zeit $O(\mathcal{M}(n) \cdot \log(n))$ berechnen.*

Wie im Abschnitt 5.4 noch gezeigt wird, lassen sich Matrizen in Zeit $o(n^3/\log(n))$ multiplizieren, so daß man den reflexiven, transitiven Abschluß in Zeit $o(n^3)$ berechnen kann. Wir wollen hier noch anmerken, daß man auf diese Weise nicht einfach zu einem Algorithmus mit Laufzeit $o(n^3)$ zur Bestimmung kürzester gewichteter Pfade gelangen kann. Die subkubischen Algorithmen zur Berechnung des Matrizenproduktes benötigen die Umkehrfunktion der herkömmlichen Addition im Matrizenprodukt. Zur Bestimmung kürzester gewichteter Pfade sieht ein Element des Matrizenprodukts wie folgt aus:

$$a_{i,j} = \min_m \{a^{(\ell)}_{i,m} + a^{(\ell)}_{m,j}\}.$$

```
TransClosure2(int A[][], int n)
{
    bool B[n][n];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            B_{i,j} = (A_{i,j} == 1) ∨ (i == j);
    for (int k = 0; k < ⌈log(n)⌉; k* = 2)
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
            { /* product of two matrices */
                bool b =FALSE;
                for (int m = 0; m < n; m++)
                    b = b ∨ (B_{i,m} ∧ B_{m,j});
                B_{i,j} = b;
            }
}
```

Abbildung 4.17: Auf Matrizenmultiplikation beruhender Algorithmus zur Bestimmung des reflexiven, transitiven Abschlusses

Die Rolle der gewöhnlichen Addition spielt hier die Minimum-Bildung, die offensichtlich nicht invertierbar ist.

4.4.3 Dijkstra-Algorithmus

In diesem Abschnitt wollen wir einen effizienteren Algorithmus für Frage 2 geben, also die kürzesten Wege von einem festen Knoten v zu allen anderen Knoten bestimmen. Dieser Algorithmus wird nach seinem Entwickler E.W. Dijkstra auch *Dijkstra-Algorithmus* genannt. In diesem Abschnitt setzen wir allerdings voraus, daß alle Kanten nichtnegatives Gewicht haben. Ansonsten läßt sich zeigen, daß der Algorithmus nicht funktioniert. Wir setzen hier zunächst noch vollständige Graphen voraus. Allerdings nehmen wir an, daß die Graphen mit Hilfe von Adjazenzlisten abgespeichert sind. Im zweiten Teil dieses Abschnittes werden wir dann Graphen betrachten, die nicht vollständig sind.

Der Algorithmus ist in Abbildung 4.18 angegeben. In jeder Iteration wird die Menge der Knoten, zu denen der kürzeste Weg bestimmt worden ist, um einen Knoten erweitert. Zu Beginn besteht diese Menge nur aus dem Startknoten s, zu dem der Weg mit gewichteter Länge 0 offensichtlich der kürzeste Weg ist. Die Idee hierbei ist, die Menge der Knoten, zu denen der kürzeste Weg bereits bestimmt worden ist, um denjenigen Knoten zu erweitern, der jetzt den kürzesten berechneten Weg

```
DIJKSTRA(GRAPH G, INT S)
{
    set F = V(G) \ {s};
    int D[|V(G)|];
    for each (v ∈ V(G))
        D[v] = γ(s, v); /* D[s] = γ(s, s) = 0 */
    while (|F| > 0)
    {
        let v ∈ F s.t. D[v] is minimal;
        F = F \ {v};
        for each (w ∈ N⁺(v))
            D[w] = min(D[w], D[v] + γ(v, w));
    }
}
```

Abbildung 4.18: Abstrakte Formulierung des Dijkstra Algorithmus

vom Startknoten hat. Die Korrektheit des Algorithmus folgt im wesentlichen aus der Voraussetzung, daß keine Kante ein negatives Gewicht besitzt. Dann werden für die übrigen Knoten, zu denen der kürzeste Weg noch nicht korrekt berechnet worden ist, eine neue Alternative ausprobiert, nämlich über den neuen nun als korrekt betrachteten Knoten.

Lemma 4.15 *Dijkstra's Algorithmus ist korrekt.*

Beweis: Die Korrektheit des Algorithmus beweisen wir mit vollständiger Induktion über die Kardinalität der Menge $V' = V \setminus F$ der bereits als korrekt betrachteten Knoten. Die Induktionsannahme lautet:

> *Für die Knoten $v \in V'$ sind die gewichteten Längen der kürzesten Wege bereits korrekt berechnet.*

Induktionsanfang ($|V'| = 1$): Da zu Beginn $V' = \{s\}$ und $D[s] = 0$ ist, ist der Induktionsanfang gelegt.

Induktionsschritt ($V' \to V' \cup \{v\}$): Nehmen wir für einen Widerspruchsbeweis an, daß $D[v]$ nicht korrekt berechnet worden ist (siehe auch Abbildung 4.19). Sei w der erste Knoten auf dem kürzesten Weg p von s zu v für den $w \notin V'$ gilt. Sicherlich ist $w \neq v$, da ansonsten der Algorithmus $D[v]$ korrekt berechnet hätte. Andererseits ist $D[w]$ bereits korrekt berechnet worden als der direkte Vorgänger von w auf dem Pfad p zu V' hinzugenommen worden ist. Bezeichnen wir im folgenden die

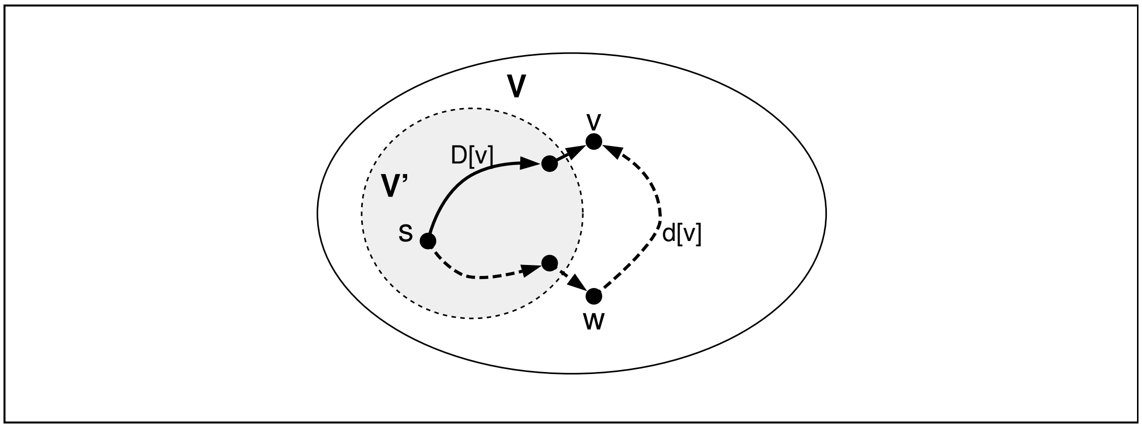

Abbildung 4.19: Beweis der Korrektehit von Dijkstra's Algorithmus

gewichtete Länge eines kürzesten Weges von s nach v mit $d[v]$. Da der Algorithmus $D[v]$ falsch berechnet hat und da bei der Berechnung von $D[v]$ nur Pfade von s nach v berücksichtigt werden, gilt $D[v] > d[v]$. Damit gilt nun, daß $D[w] = d[w] \leq d[v] < D[v]$, da alle Kanten nichtnegatives Gewicht haben. Dann hätte der Algorithmus also w anstelle von v gewählt und wir erhalten einen Widerspruch. ∎

Es bleibt nun noch die Laufzeit zu analysieren. Dazu geben wir erst einmal eine etwas konkretere Fassung in Abbildung 4.20 an. Die Menge F ist hier durch ein Boole'sches Array implementiert. Statt der Menge der direkten Nachfolger von v betrachten wir alle Knoten w, da wir uns hier ja zunächst auf vollständige Graphen beschränken.

Da in jedem Durchlauf der **while**-Schleife die Kardinalität von F um eins sinkt, kann diese Schleife maximal n-mal durchlaufen werden. Den minimalen Knoten v haben wir in Zeit $O(n)$ bestimmt. Die Werte der restlichen Knoten sind ebenfalls in Zeit $O(n)$ neu berechnet worden. Damit erhalten wir folgendes Theorem

Theorem 4.16 *Für einen Graphen mit n Knoten berechnet Dijkstra's Algorithmus die kürzesten Wege von einem festen Knoten in Zeit $O(n^2)$.*

Für Graphen mit n Knoten und $\Theta(n^2)$ Kanten ist Dijkstra's Algorithmus optimal. Falls der Graph nur $o(n^2)$ Kanten hat, kann man Dijkstra's Algorithmus jedoch verbessern. Dies soll uns im Rest dieses Abschnittes beschäftigen.

4.4.4 Dijkstra-Algorithmus mit Priority Queues

Den Algorithmus teuer gemacht haben insbesondere die Minimumbildung der Knoten in F und das der Kanten ausgehend vom betrachteten Minimum. Was wir also

```
Dijkstra(graph G, int s, int gamma[][])
{
    int n = |V(G)|, v;
    bool F[n]; /* the set F */
    int f = n − 1; /* Number of vertices in F */
    int D[n];
    for (int i = 0; i < n; i++)
    {
        F[i] = (i! = s);
        D[i] = γ[s][i]; /* D[s] = γ[s][s] = 0 */
    }

    while (f > 0)
    {
        int min = ∞;
        for (int i = 0; i < n; i++)
            if (F[i] && D[i] ≤ min)
            {
                min = D[i];
                v = i;
            }
        F[v] =FALSE; f−−;
        for (int w = 0; w < n; w++)
            D[w] = min(D[w], D[v] + γ[v][w]);
    }
}
```

Abbildung 4.20: Der Dijkstra-Algorithmus für vollständige Graphen

benötigen, um unter bestimmen Umständen effizienter sein zu können, ist eine Datenstruktur PQ, die die folgenden Operationen unterstützt:

`insert`**(PQ q, elt e)** Das Einfügen eines Elements e samt seines Schlüssels.

int `size`**(PQ q)** Bestimmung der Anzahl der gespeicherten Elemente.

elt `delete_min`**(PQ q)** Bestimmen des Minimums bzgl. einer vorgegebenen Ordnung auf den zu Elementen korrespondierenden Schlüsseln mit gleichzeitigem Entfernen eines minimalen Elementes.

`decrease_key`**(PQ q, elt e, int val)** Das Erniedrigen des Schlüssels eines gespeicherten Elementes e auf den Wert val. Dabei wird vorausgesetzt, daß das Element e bereits in der Priority Queue enthalten ist.

```
PQ_Dijkstra(graph G, int s)
{
    int n = |V(G)|
    int D[n];
    for (int i = 0; i < n; i++)
        if (i==s) D[s] == 0;
        else D[i] = ∞;
    PQ q;
    for each (int w ∈ N⁺(s))
    {
        D[w] = γ((s, w));
        if (w! = s) insert(q,(w, D[w]));
    }

    while (size(q) > 0)
    {
        int v = delete_min(q);
        for each (w ∈ N⁺(v))
        {
            D[w] = min(D[w], D[v] + γ(v, w))
            decrease_key(q, w, D[w]);
        }
    }
}
```

Abbildung 4.21: Dijkstra Algorithmus mit Priority Queues

In Abschnitt 2.3 haben wir bereits eine Datenstruktur kennengelernt, die die ersten drei der obigen Operationen zur Verfügung stellt, den Heap. Allerdings hatten wir dort die Operation `delete_max` anstelle von `delete_min` gefordert. Die nötigen Modifikationen, um von `delete_max` zu `delete_min` übergehen zu können, sind so offensichtlich, daß wir an dieser Stelle nicht näher darauf eingehen. Wir nennen eine Datenstruktur eine *Priority Queue* oder *Vorrangwarteschlange*, wenn sie alle vier geforderten Operationen zur Verfügung stellt. Manchmal wird auch noch gefordert, daß man aus einer Priority Queue ein beliebiges Element entfernen kann (mittels einer Operation `delete`). Damit können wir nun den Dijkstra-Algorithmus mit Priority-Queues wie in Abbildung 4.21 formulieren.

Wir werden nun zeigen, wie man die Operation `decrease_key` auf einem Heap simulieren kann. Dazu fügen wir einfach ein neues Element mit dem kleineren Schlüssel in den Heap ein. Wir müssen dann nur bei der `delete_min` Operation aufpassen, daß, wenn wir ein Element mit einem minimalen Schlüssel aus dem Heap löschen, dieses nicht unbedingt das Minimum der Priority Queue sein muß. Es könnte ja sein, daß es sich dabei um einen alten Wert eines längst gelöschten Elements aus der Priority Queue handelt, der nun noch nutzlos im Heap herumliegt. Um dies zu vermeiden, merken wir uns zusätzlich in einem Boole'schen Array, welche Elemente wirklich noch in der Priority Queue enthalten sind.

Damit erhalten wir die folgende Komplexität für die folgenden Operationen einer Priority Queue mit der Realisierung als Heap, wobei wir annehmen, daß n Elemente eingefügt werden und m `decrease_key` Operationen ausgeführt werden. Zunächst halten wir jedoch noch fest, daß sich dann maximal $n + m$ Elemente in dem Heap befinden können, da noch bis zu m ungültige Elemente im Heap liegen können, die durch unsere Implementation der `decrease_key` Operation nicht entfernt worden sind.

`insert` Das Einfügen benötigt Zeit $O(\log(n+m))$, da sich maximal $n+m$ Elemente im Heap befinden.

`size` Die Abfrage der Größe benötigt Zeit $O(1)$.

`delete_min` Das Bestimmen des Minimums in der Priority Queue benötigt Zeit maximal $O(m \log(n + m))$, da maximal $n + m$ Elemente im Heap vorhanden sind und maximal m nutzlose `delete_min`'s des Heaps ausgeführt werden, bevor ein noch in der Priority Queue enthaltenes Element gefunden wird.

`decrease_key` Das Erniedrigen des Schlüssels in der Priority Queue benötigt Zeit $O(\log(n + m))$, da maximal $O(n + m)$ Elemente im Heap vorhanden sind.

Der Zeitbedarf für die `delete_min` Operation sieht sehr schlecht aus. Allerdings handelt es sich hier um eine worst-case Abschätzung, die nur *einmal* während der gesamten Verwendung der Priority Queue auftreten kann. Für eine beliebige Folge von n Einfügungen, n Löschungen und m Erniedrigungen von Schlüsseln benötigt man maximal Zeit $O((n + m) \log(n + m))$ für alle `delete_min` Operationen der Priority Queue. Die Begründung hierfür ist einfach. Jedes `delete_min` als Heap-Operation kostet maximal Zeit $O(\log(n+m))$, da die Größe des Heaps durch $n+m$ beschränkt ist. Andererseits kann es daher auch nur maximal $n+m$ viele `delete_min` Operationen des Heaps geben.

Der Vorteil dieser Methode zur Realisierung von Priority Queues mittels Heaps ist, daß wir die genaue Realisierung des Heaps nicht kennen müssen. Verwendet man die Implementation von Heaps wie bei Heapsort, so kann ein im allgemeinen (aber nicht für Dijkstra) effizientere Implementation angeben. Man merkt sich zu jedem Element noch einen Verweis, wo dieser im Heap (also Array) abgespeichert ist. Dann kann man die `decrease_key` Operation ganz einfach realisieren, indem man genau an dieser Stelle den Schlüssel verkleinert und dann an diesem Knoten die Prozedur `reheap` aufruft. Damit haben die Operationen `insert`, `delete_min` und `decrease_key` Zeitkomplexität $O(\log(n))$ unabhängig von der Anzahl der ausgeführten `decrease_key` Operationen.

Damit können wir die Zeitkomplexität des Dijkstra-Algorithmus mit Priority Queues basierend auf Heaps analysieren. In unserem Fall ist nun $n = |V|$ und $m = |E|$. Man erinnert sich, daß $m = O(n^2)$ und damit $\log(m) = O(\log(n))$ ist. Wie wir eben ausgeführt haben, kosten die `delete_min`'s Zeit $O((n+m)\log(n+m)) = O(m\log(n))$. Darüber hinaus wird die Operation `decrease_key` maximal für jede Kante einmal aufgerufen, also ergeben auch alle `decrease_key` Operationen zusammen einen Zeitbedarf von $O(m\log(n))$. Die Initialisierung der Priority Queue benötigt maximal $O(n\log(n))$ Zeit. Die Initialisierung des Arrays D benötigt hingegen nur Zeit $O(n)$. Alles zusammen haben wir damit folgendes Theorem bewiesen:

Theorem 4.17 *Der Dijkstra-Algorithmus mit Priority Queues basierend auf Heaps benötigt zum Berechnen der gewichteten Längen der kürzesten Pfade von einem bestimmten Knoten in einem Graphen $G = (V, E)$ maximal $O(|E|\log(|V|))$ Zeit.*

Für dünne Graphen, d.h. für Graphen mit wenig Kanten ($m = o(n^2/\log(n))$), ist diese Version effizienter als die Standard Version.

Man kann eine noch bessere Implementierung erhalten, wenn wir die Priority Queues als Fibonacci Heaps implementieren. Für Fibonacci Heaps kann man zeigen, daß für eine Folge von ℓ Operationen `insert`'s, `delete_min`'s und `decrease_key`'s in beliebiger Reihenfolge insgesamt Kosten $O(\ell + k\log(n))$ verursachen, wobei n die maximale Zahl von Elementen im Fibonacci Heap und k die Anzahl der `delete_min` Operationen ist (siehe auch Theorem 4.22 auf Seite 162). Wir wollen hier nicht auf die Details eingehen, sondern nur das folgende Theorem festhalten:

Theorem 4.18 *Der Dijkstra-Algorithmus mit Fibonacci Heaps benötigt zum Berechnen der gewichteten Längen der kürzesten Pfade von einem bestimmten Knoten in einem Graphen $G = (V, E)$ maximal $O(|E| + |V|\log(|V|))$ Zeit*

25. Juli 97

4.5 Interludium: Fibonacci-Heaps (*)

In diesem Abschnitt wollen wir eine effizientere Methode zur Realisierung von Priority Queues angeben, nämlich Fibonacci Heaps.

4.5.1 Aufbau eines Fibonacci Heaps (*)

Ein *Fibonacci Heap* ist im wesentlichen ein Wald, in dem die Wurzeln der einzelnen Bäume in einer Liste kreisförmig verkettet sind. Diese Liste wollen wir in Zukunft auch *Wurzelliste* nennen. Die Elemente mit ihren Schlüsseln sind dabei so in den Bäumen abgespeichert, daß jeder Baum die *Heap-Eigenschaft* erfüllt. Man beachte, daß hier für jeden Knoten gilt, daß der Schlüssel im Elter immer kleiner gleich dem eigenen Schlüssel ist. Um schnell das Minimum zu finden gibt es noch einen Zeiger auf diejenige Wurzel eines Baumes, die ein Element mit minimalem Schlüssel besitzt.

Für die folgende Implementierung wollen wir noch festhalten, wie diese Bäume realisiert sind, und welche zusätzlichen Attribute die einzelnen Knoten erhalten. Jeder Knoten hat einen Verweis auf seinen Elter, eines seiner Kinder (man stelle sich der Einfachheit halber einen Verweis auf das linkeste Kind vor), sein linkes und rechtes Geschwister. Dazu werden die Geschwister in einer doppelt verketteten Liste gehalten. Dabei ist das linke bzw. rechte Kind eines linkesten bzw. rechtesten Kindes das rechteste bzw. linkeste Kind seines Elters. Die Anzahl der Kinder eines Knotens nennen wir im folgenden seinen *Rang*. Außerdem kann jeder Knoten entweder markiert sein oder nicht. Dabei halten wir hier fest, daß aus Prinzip Wurzeln immer unmarkiert sind. In Abbildung 4.22 ist ein Beispiel eines Fibonacci Heaps mit neun Elementen angegeben, dabei sind die Verweise durch Pfeile angedeutet.

Nun müssen wir noch die Operationen auf einem Fibonacci Heap realisieren. Die Operation `size` wird wie üblich implementiert. Die Operation `insert` hängt einfach einen neuen einelementigen Baum in die Wurzelliste ein. Zum Beispiel könnte in den

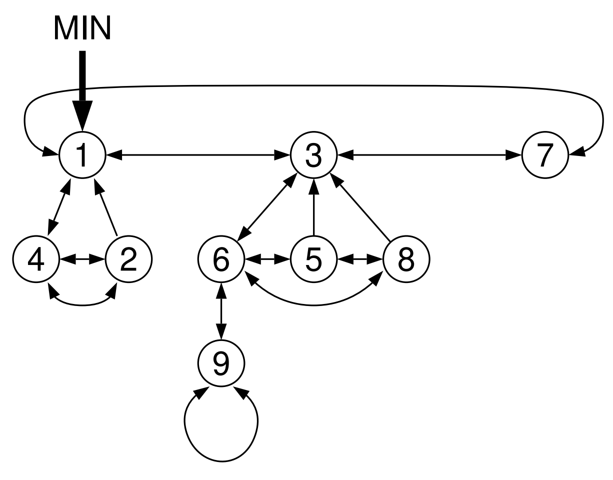

Abbildung 4.22: Beispiel eines Fibonacci Heaps

Fibonacci Heap in Abbildung 4.22 zuletzt der Knoten mit dem Schlüssel 7 eingefügt worden sein.

Bei der Operation `decrease_key` wird der Schlüssel des entsprechenden Knoten erniedrigt. Um den Knoten schnell aufzufinden, liefert die `insert` Operation einen Verweis auf diesen zurück, der dann bei einer `decrease_key` Operation auf diesem Knoten wieder mitübergeben werden muß. Alternativ kann man solche Verweise auch in einem Dictionary verwalten. Um einer Verletzung der Heapeigenschaft vorzubeugen, wird der modifizierte Knoten samt dem darunter hängenden Teilbaum von seinem Elter abgehängt und die Wurzel des so abgetrennten Teilbaumes in die Wurzelliste eingehängt. Eventuell muß nun auch noch der Zeiger auf eine minimale Wurzel aktualisiert werden. Dann wird noch der ehemalige Elter des Knotens markiert, wenn er keine Wurzel ist. War dieser Knoten bereits markiert, so wird auch dieser Knoten samt seinem noch an ihm hängenden Teilbaum in die Wurzelliste eingehängt. Dann wird ebenfalls dessen Elter markiert, wenn er keine Wurzel ist, und die Prozedur wiederholt sich bis wir einen unmarkierten Elter treffen (spätestens die Wurzel eines Baumes). Dieses kaskadenartig auftretende Abschneiden von Teilbäumen wird als *kaskadenartiger Schnitt* (engl. cascading cut) bezeichnet. Zum Beispiel könnte im Fibonacci Heap in Abbildung 4.22 zuletzt der Teilbaum mit Wurzel 3 in der Wurzelliste durch eine `decrease_key` Operation vom Knoten mit dem Schlüssel 4 abgeschnitten worden sein, wobei der neue Wurzelknoten vorher den Schlüssel 5 gespeichert haben könnte.

Bei der Operation `delete_min` wird die minimale Wurzel als Ergebnis zurückgegeben und alle Kinder der minimalen Wurzel in die Wurzelliste aufgenommen. Nun muß der Zeiger auf das Minimum neu berechnet werden. Da dies sehr zeitaufwendig ist, weil die gesamte Wurzelliste durchsucht werden muß, werden wir gleichzeitig die Wurzelliste wie folgt aufräumen. Je zwei Wurzeln mit demselben Rang werden zu einem neuen Baum verschmolzen, wobei die Wurzel mit dem kleineren Schlüssel zum Elter der anderen Wurzel wird. Nach dem Aufräumen haben dann alle Wurzeln der Bäume, die in der Wurzelliste hängen, verschiedene Ränge. Die Implementierung des Fibonacci Heaps ist detailliert noch einmal in den Abbildungen 4.23 und 4.24 auf den Seiten 158 und 159 angegeben.

4.5.2 Analyse von Fibonacci Heaps (*)

Bleibt noch die Berechnung der Kosten für die einzelnen Operationen. Die Operationen `insert` und `size` lassen sich offensichtlich in konstanter Zeit ausführen. Da die anderen Operationen im worst-case allerdings teuer werden können, betrachten wir hier die *amortisierten Kosten*, also die Kosten, die im Mittel für eine Operation eines Typs ausgegeben werden. Hierbei wird das Mittel über alle ausgeführten Operationen gebildet.

```
FIBONACCI HEAP(INT n)
{
    int min_root=NULL;              /* a minimal root */
    int left[n], right[n];          /* its left and right sibling*/
    int child[n], parent[n];        /* one of its children and its parent */
    int key[n], rank[n];            /* its key and its rank */
    bool marked[n];                 /* its mark */
    /* Note that the list of roots is also realized with left/right */

    int insert(int key)
    {
        int v=get_free_number();
        if (min_root==NULL) /* the root list is empty */
            min_root=left[v]=right[v]=v;
        else
        { /* insert v into the rootlist */
            right[v]=min_root;    left[v]=left[min_root];
            left[min_root]=right[left[min_root]]=v;
            if (key<key[min_root]) min_root=v;
        }
        child[v]=parent[v]=NULL;
        rank[v]=0;    key[v]=key;    marked[v]=FALSE;
        return v;
    }

    decrease_key(int v, int key)
    { /* Assuming that key≤key[v] */
        key[v]=key;    if (parent[v]≠NULL) cut(v);
    }

    cut(int v);
    { /* Remove v from its siblings*/
        if (left[v]==v)                 /* v is a lonely child */
            child[parent[v]]=NULL;
        else                            /* v has some siblings */
        {
            right[left[v]]=right[v];    left[right[v]]=left[v];
            if (child[parent[v]]==v) child[parent[v]]=left[v]
        }
        /* Now v will be inserted in the root list */
        right[v]=min_root;    left[v]=left[min_root];
        left[min_root]=right[left[min_root]]=v;
        if (key[v]<key[min_root]) min_root=v;        /* updating minimal root */
        if (marked[parent[v]]) cut(parent[v]);       /* cascading cut */
        else if (parent[parent[v]]≠NULL) marked[parent[v]]=TRUE;
        parent[v]=NULL;    marked[v]=FALSE; /* unmark root */
    }
}
```

Abbildung 4.23: Implementierung des Fibonacci Heaps (I)

```
FIBONACCI HEAPS(INT n) [CONT.]
{
    int delete_min()
    { /* assuming that the heap is not empty */
        int v=child[min_Root]
        if (left[min_root]≠ min_root)
        { /* root list contains at least 2 trees */
            left[v]=left[min_root];        right[left[v]]=right[min_root];
            right[left[min_root]]=v;       left[right[min_root]]=left[v];
        }
        min_root=v;    put_free_number(min_root);    clean_up();
        return key[v];
    }

    clean_up();
    {
        int root[⌈log(n)⌉]=NULL;                 /* containing roots with rank i */
        int v=min_root;
        if (v ≠ NULL) left[right[v]]=NULL;       /* break up rootlist */
        while (v ≠ NULL)
        { /* consider next tree with root v */
            while (root[rank[v]]≠NULL)
            { /* while there are two roots v and w having the same rank */
                w=root[rank[v]];               root[rank[v]]=NULL;
                if (key[w]<key[v]) swap(v,w)
                /* Now v becomes parent of w */
                parent[w]=v;                   if (rank[v]==0) child[v]=w;
                right[w]=child[v];             left[w]=left[child[v]];
                right[left[child[v]]]=w;       left[child[v]]=w;
                rank[v]++;                     left[v]=right[v]=NULL;
            }
            root[rank[v]]=v;
        }
        min_root=NULL;
        for(i = 0; i ≤ ⌈log(n)⌉; i++)
            if (root[i]≠NULL)
            { /* insert tree with rank i into rootlist */
                if (min_root==NULL) /*rootlist is still empty */
                    left[min_root]=right[min_root]=min_root=root[i];
                else
                { /* insert root[i] into the rootlist */
                    right[root[i]]=min_root;    left[root[i]]=left[min_root];
                    left[min_root]]=right[left[min_root]]=root[i];
                    if (rank[root[i]]<rank[min_root]) min_root=root[i];
                }
            }
    }
}
```

Abbildung 4.24: Implementierung des Fibonacci Heaps (II)

Dazu werden wir für billige Operationen mehr bezahlen als eigentlich nötig, um dann im Falle einer teuren Operation davon Profit schlagen zu können. Das kann man sich am besten mit einem Bankkonto vorstellen. Jedesmal wenn wir etwas günstiger kaufen können als erwartet, zahlen wir den gesparten Betrag auf das Bankkonto ein. Wenn dann mal etwas teurer als erwartet ist, können wir den Mehrpreis von unserem Guthaben auf dem Bankkonto begleichen. Der Trick dabei ist, den erwarteten Preis so geschickt festzulegen, daß sich bei einem teuren Einkauf immer noch genügend Geld auf dem Bankkonto befindet. Wir nehmen hier an, daß uns die Bank keinen Überziehungskredit gewährt.

Für die weitere Analyse benötigen wir erst ein paar Fakten über Fibonacci Heaps.

Lemma 4.19 *Sei v ein Knoten im Fibonacci Heap und seien $c_1, \ldots, c_k$ seine Kinder in der Reihenfolge, in der sie an v angehängt wurden. Dann ist der Rang von c_i immer mindestens $i-2$.*

Beweis: Für $i \leq 2$ ist nichts zu zeigen. Nur beim Aufräumen der Wurzelliste wurden Knoten zu Kindern von anderen Knoten, wobei beide Knoten denselben Rang hatten. Als also c_i ein Kind von v wurde, war der Rang von v mindestens $i-1$, da er bereits die Kinder $c_1, \ldots, c_{i-1}$ hatte. Damit war damals auch der Rang von c_i mindestens $i-1$, da beide Knoten beim Verschmelzen denselben Rang haben mußten. Seit diesem Zeitpunkt kann c_i höchstens ein Kind verloren haben, da es beim Verlust eines zweiten Kindes durch den kaskadenartigen Schnitt selbst vom seinem Elter v getrennt worden wäre. Also ist der Rang von c_i mindestens $i-2$. ■

Mit Hilfe dieses Lemmas können wir nun eine wichtige Aussage über die Größen von Teilbäumen des Fibonacci Heaps machen. Allerdings benötigen wir dazu noch folgende schöne Eigenschaften von Fibonacci Zahlen:

Lemma 4.20 *Sei F_i die i-te Fibonacci-Zahl, dann gilt:*

$$\forall k \geq 0: \; F_{k+2} = 1 + \sum_{i=1}^{k} F_i.$$

Beweis: Wir führen den Beweis mit vollständiger Induktion über k.

Induktionsanfang ($k=0$): $F_2 = 1 = 1 + 0 = 1 + \sum_{i=1}^{0} F_i$.

Induktionsschritt ($k-1 \to k$):

$$1 + \sum_{i=1}^{k} F_i = 1 + \sum_{i=1}^{k-1} F_i + F_k \overset{\text{I.V.}}{=} F_{k+1} + F_k = F_{k+2}$$

■

Damit können wir nun eine Abschätzung für die Größe eines Teilbaums mit Hilfe des Ranges seiner Wurzel angeben.

Lemma 4.21 *Sei v ein Knoten des Fibonacci Heaps mit Rang k, dann ist die Größe des an v gewurzelten Teilbaumes mindestens F_{k+2}, wobei F_i die i-te Fibonacci-Zahl bezeichnet.*

Beweis: Wir führen den Beweis mit vollständiger Induktion über den Rang k.

Induktionsanfang ($k = 0, 1$): Offensichtlich hat ein Teilbaum, dessen Wurzel Rang 0 bzw. 1 hat, mindestens $1 = F_2$ bzw. $2 = F_3$ viele Knoten

Induktionsschritt ($\rightarrow k$): Sei v ein Knoten mit Rang k. Dann hat nach Lemma 4.19 das i-te Kind mindestens Rang $i - 2$. Nach Induktionsvoraussetzung erhalten wir, daß der Teilbaum von v mindestens $2 + \sum_{i=2}^{k} F_i$ Knoten enthalten muß, wobei der Term 2 für die Wurzel selbst und das erste Kind der Wurzel steht. Mit Lemma 4.20 erhalten wir somit für die Mindestanzahl von Knoten im Teilbaum mit Wurzel v: $2 + \sum_{i=2}^{k} F_i = 1 + \sum_{i=1}^{k} F_i = F_{k+2}$. ■

Aus diesem Lemma wird auch der Name Fibonacci Heap klar. Da die Fibonacci Zahlen exponentiell wachsen (siehe auch die Einleitung), folgt, daß der Rang eines Knotens in einem Fibonacci Heap mit n Knoten durch $O(\log(n))$ beschränkt ist.

Unsere Vorsorgestrategie wird wie folgt aussehen. Für jeden Knoten, der in die Wurzelliste eingehängt wird, werden wir eine Einheit auf das Bankkonto einzahlen. Wir hoffen damit, die nötigen Aufräumarbeiten in der Wurzelliste für diese Wurzel finanzieren zu können. Für jeden Knoten, den wir markieren, werden wir sogar zwei Einheiten auf das Bankkonto einzahlen. Damit wollen wir die Kosten für nötig werdende kaskadenartige Schnitte bezahlen, die ja bei einer zweiten Markierung fällig werden.

Warum aber zwei Einheiten? Leider wird bei einem kaskadenartigen Schnitt, der abgeschnittene Teilbaum in die Wurzelliste eingehängt. Damit sind wir sehr weitsichtig und bezahlen schon eine weitere Einheit für die noch später anfallenden Aufräumarbeiten, die durch einen zukünftigen kaskadenartigen Schnitt ausgelöst werden können.

Da die `insert` Operationen zu den billigen Operationen gehören, werden wir für jede solche Operation eine Einheit auf das Bankkonto einzahlen (als Vorleistung für eventuell nötige Aufräumarbeiten). Die Zeitkomplexität der `decrease_key` Operation ist linear in der Anzahl der durchgeführten Schnitte. Man überlegt sich leicht, daß man einen Schnitt in konstanter Zeit durchführen kann, wenn man genügend Informationen abgespeichert hat.

Da bei einem kaskadenartigen Schnitt nur solche Knoten in die Wurzelliste eingefügt werden, die bereits vorher markiert waren, können wir sicher sein, daß wir dieses Einfügen aus unserem Ersparten bezahlen können. Da wir sogar zwei Einheiten für einen markierten Knoten gespart haben, heben wir nun eine Einheit für den Schnitt ab und sparen die zweite Einheit für eventuell später auftretende Aufräumarbeiten auf.

Bleibt noch die Operation `delete_min`. Dabei werden alle Kinder einer minimalen Wurzel in die Wurzelliste eingehängt. Wie wir ja oben bereits bewiesen haben, ist der Rang eines Knotens logarithmisch in der Größe des Fibonacci Heaps beschränkt. Dieses Einhängen verursacht also logarithmische Kosten.

Nun stehen noch die Aufräumarbeiten an. Dabei werden je zwei Bäume mit gleichem Rang verschmolzen. Bei jedem Verschmelzen, wird die Anzahl der Bäume um eins weniger. Da wir für jeden Baum, den wir in die Wurzelliste gehängt haben, eine Einheit gespart haben, können wir also die gesamten Aufräumarbeiten aus unserem Ersparten begleichen, da sich das Verschmelzen von zwei Bäumen offensichtlich in konstanter Zeit realisieren läßt. Bei diesen Aufräumarbeiten bestimmen wir nebenbei gleich noch die neue minimale Wurzel mit, da wir ja sowieso jede Wurzel betrachten müssen.

Damit erhalten wir die in Abbildung 4.25 angegebene Aufstellung der Kosten der einzelnen Operationen auf dem Fibonacci Heap, unter der Annahme, daß sich maximal n Element im anfänglich leeren Fibonacci Heap befinden.

	worst case	amortisiert
`size`	$O(1)$	$O(1)$
`insert`	$O(1)$	$O(1)$
`decrease_key`	$O(\log(n))$	$O(1)$
`delete_min`	$O(n)$	$O(\log(n))$

Abbildung 4.25: Kosten der Operationen eines Fibonacci Heaps mit maximal n Elementen

Also können die Operationen `size`, `insert` sowie `decrease_key` in konstanter amortisierter Zeit und die Operation `delete_min` in logarithmischer amortisierter Zeit durchgeführt werden. Damit können wir das folgende, bereits im vorigen Abschnitt verwendete, Theorem bereitstellen:

Theorem 4.22 *Beginnend mit einem leeren Fibonacci Heap kann eine Folge von ℓ Operationen* `insert`, `decrease_key` *und* `delete_min` *in Zeit $O(\ell + k\log(n))$ ausgeführt werden, wobei n die maximale Zahl von Elementen im Fibonacci Heap und k die Anzahl der* `delete_min` *Operationen ist.*

4.6 Minimale Spannbäume

In diesem Abschnitt wollen wir minimale Spannbäume berechnen. Dabei ist ein *Spannbaum* für einen ungerichteten Graphen $G = (V, E)$ ein Baum $T = (V, F)$, der zugleich ein Teilgraph des Graphen G auf all seinen Knoten ist. Auch hier setzen wir wieder Gewichte auf den Kanten des ungerichteten Graphen voraus. Ein *minimaler Spannbaum* eines ungerichteten Graphen G ist dann ein Spannbaum des ungerichteten Graphen G, wobei die Summe der Kantengewichte der Kanten des Spannbaumes minimal unter allen möglichen Spannbäumen für den ungerichteten Graphen G ist. Also ist für einen ungerichteten gewichteten Graphen $G = (V, E)$ mit Gewichtsmatrix γ der minimale Spannbaum $T = (V, F)$ ein Spannbaum von G für den gilt:

$$\gamma(T) \leq \min\{\gamma(T') \mid T' = (V, F') \text{ ist ein Spannbaum von } G \text{ mit } E' \subseteq F'\}$$

wobei $\gamma(T') = \sum_{f \in F'} \gamma(f)$ für einen Baum $T' = (V, F')$.

Als Beispiel kann man sich die Knoten als Niederlassungen einer Firma (z.B. in verschiedenen Städten) vorstellen und die Kanten als vorhandene Standleitungen zwischen den Rechenzentren der einzelnen Niederlassungen. Wenn man nun die Kantengewichte als die monatlichen Kosten für die Standleitungen interpretiert, kann man sich fragen, welche Standleitungen man mieten muß, damit je zwei Niederlassungen über einen Pfad von Standleitungen verbunden sind. Naturgemäß ist man an einer billigsten Lösung interessiert, die dann der minimale Spannbaum liefert.

Dies ist auch der Grund warum wir uns in diesem Abschnitt nur mit ungerichteten Graphen beschäftigen. In diesem Zusammenhang machen gerichtete Graphen meist keinen Sinn. Um überhaupt eine Lösung zu bekommen setzen wir außerdem voraus, daß die gegebenen ungerichteten Graphen zusammenhängend sind. Wie wir dies effizient überprüfen können, haben wir ja bereits in Abschnitt 4.3 gesehen.

4.6.1 Prim-Algorithmus (+)

Ein erster Ansatz zur Berechnung eines minimalen Spannbaumes ist sehr ähnlich zum Dijkstra-Algorithmus. Zuerst wählt man einen beliebigen Knoten des Graphen. Dieser bildet offensichtlich einen Teilbaum des Graphen. Dann erweitert man immer den aktuellen Teilbaum um eine billigste Kante, die inzident zu genau einem Knoten des aktuellen Teilbaumes ist. Der Algorithmus ist in Abbildung 4.26 gegeben.

Die Wahl der billigsten Kante, die einen Knoten des bisher berechneten Spannbaumes mit einem Knoten außerhalb des Spannbaumes ($\in F$) verbindet, geschieht wieder mit Hilfe einer Priority Queue. Dieser Algorithmus wird nach seinem Entwickler R.C. Prim auch *Prim-Algorithmus* genannt.

```
PRIM(WEIGHTED_GRAPH G)
{
    int n = |V(G)|;
    PQ q;
    set tree_edges=∅; /*the set of edges of the spanning tree */
    int cost=0; /* the sum of edge weights of the constructed tree */
    bool F[n]; /* is the vertex free, i.e. included in PQ? */
    bool T[n]; /* is the vertex already included in the tree? */
    for (int i = 1; i < n; i++)
        T[i] = F[i] = FALSE;
    T[0] = TRUE;
    for each (int v ∈ N(0))
    {
        insert(q,(v, {0, v}, γ({0, v})));
        F[v] = TRUE;
    }

    while (size(q) > 0)
    {
        /* look for a cheapest edge linking an unreached vertex */
        (w, {v, w}) = delete_min(q);
        /* Note that v belongs to the tree and w to F */
        tree_edges = tree_edges ∪ {{v, w}}
        cost+=γ({v, w});
        T[w] = TRUE;
        F[w] = FALSE;

        /* insert or update vertices which can be reach from w */
        for each (u ∈ N(w))
            if (notT[u]) /* u is not inside the constructed tree */
                if (F[u])
                    decrease_key(q, (u, {w, u}), γ({w, u});
                else
                {
                    insert(q, (u, {w, u}, γ({w, u});
                    F[u] = TRUE;
                }
    }
}
```

Abbildung 4.26: Prim's Algorithmus zur Berechnung eines minimalen Spannbaumes

Der Prim-Algorithmus gehört zu der Klasse der *Greedy-Algorithmen*. Das zentrale Vorgehen des Prim-Algorithmus ist es, den momentan konstruierten Teilbaum immer um die billigste zu ihm inzidente Kante zu erweitern. Algorithmen für Optimierungsprobleme, die immer eine lokal optimale Entscheidung treffen (hier die Wahl einer billigsten inzidenten Kante), sind in einem gewissen Sinne gierig (engl. greedy). Sie versuchen das Optimum zu erreichen, indem sie auf einem vermeintlichen Weg dorthin immer das bestmögliche Teilstück wählen. Natürlich sollte es klar sein, daß es Optimierungsprobleme gibt, bei der eine solche Strategie versagt. Es kann manchmal besser sein, sich an bestimmten Stellen mit weniger zufrieden zu geben, wenn man dafür später um so mehr bekommt. In unserem Fall werden wir jedoch sehen, daß ein solcher Greedy-Ansatz bei minimalen Spannbäumen zum Ziel führt.

Im Falle von Dijkstra's Algorithmus hatten wir in der Priority Queue die Knoten gespeichert, die direkte Nachfolger eines Knotens sind, für die wir die korrekte gewichtete Länge eines kürzesten Wegs bereits berechnet hatten. Der zugehörige Schlüssel war gerade der kürzeste gewichtete bereits bekannte Pfad vom Knoten s zu diesem Knoten.

Hier speichern wir nun ebenfalls wieder die Knoten in der Priority Queue ab, die über eine Kante mit dem bereits konstruierten Teilbaum verbunden sind. Der zugehörige Schlüssel ist nun das Gewicht der Kante, was im wesentlichen dem Abstand eines inzidenten Knotens zum bisher berechneten Spannbaum entspricht. Zusätzlich speichern wir in der Priority Queue noch die Kante ab, über die der Knoten mit den angegebenen Kosten erreicht werden kann. Man beachte, daß wir in die Queue nur solche Kanten aufnehmen, so daß die Kante inzident zu einem Knoten im bereits berechneten Spannbaum und inzident zu einem Knoten ist, der sich noch nicht im Spannbaum befindet ($\in F$).

Aufgrund der Ähnlichkeit zu Dijkstra's Algorithmus führen wir hier die Analyse des Zeitbedarfs nicht aus. Es sollte klar sein, daß sich die Analyse von Dijkstra's Algorithmus auf Prim's Algorithmus übertragen läßt. Es bleibt nur noch die Korrektheit von Prim's Algorithmus zu zeigen. Die Korrektheit folgt unmittelbar aus dem folgenden fundamentalen Lemma.

29. Juli 97

Lemma 4.23 *Sei $G = (V, E)$ ein ungerichteter gewichteter Graph und γ seine Gewichtsmatrix. Sei $E' \subseteq E$ eine Teilmenge der Kanten, so daß $G' = (V, E')$ azyklisch ist. Sei $e \in E \setminus E'$ eine Kante minimalen Gewichts, so daß auch $G'' = (V, E' \cup \{e\})$ azyklisch ist. Dann gibt es einen Spannbaum $T = (V, F)$, für den $E' \cup \{e\} \subseteq F$ gilt, so daß*

$$\gamma(T) \leq \min\{\gamma(T') \mid T' = (V, F') \text{ ist ein Spannbaum von } G \text{ mit } E' \subseteq F'\}$$

wobei für einen Baum $T' = (V, F')$: $\gamma(T) = \sum_{f \in F'} \gamma(f)$.

Beweis: Sei $\hat{T} = (V, \hat{F})$ ein minimaler Spannbaum, der die Kanten aus E' umfaßt, also

$$\gamma(\hat{T}) = \min\left\{\gamma(T') \mid T' = (V, F') \text{ ist ein Spannbaum von } G \text{ mit } E' \subseteq F'\right\}.$$

Falls $e \in \hat{F}$, so ist nichts zu zeigen. Also sei $e \notin \hat{F}$. Betrachte nun den Graphen $(V, \hat{F} \cup \{e\})$. Da ein Baum zusammenhängend ist, bildet die Kante e mit dem Pfad zwischen den Endpunkten von e in $\hat{T}$ einen Kreis. Da nach Voraussetzung die Menge $E' \cup \{e\}$ keinen Kreis enthielt, gibt es in diesem Kreis eine Kante $f \in E \setminus \{E' \cup \{e\}\}$. Damit erhalten wir einen neuen Spannbaum, indem wir in $\hat{T}$ die Kante f durch die Kante e ersetzen. Da nach Voraussetzung gilt, daß $\gamma(e) \leq \gamma(f)$, ist das Gewicht dieses neuen Spannbaumes höchstens so groß wie das Gewicht von $\hat{T}$. ■

Damit können wir nun das folgende Theorem beweisen:

Theorem 4.24 *Der Prim-Algorithmus mit der Realisierung der Priority Queues als Fibonacci Heaps benötigt zum Berechnen eines minimalen Spannbaumes für einen Graphen $G = (V, E)$ Zeit $O(|E| + |V| \log(|V|))$.*

Beweis: Wir brauchen nur noch die Korrektheit zu beweisen. In jedem Durchlauf des **if**-Teil der **while**-Schleife wird der bereits konstruierte Spannbaum um eine Kante und einen Knoten erweitert. Da der Algorithmus als Kante eine minimalen Gewichts wählt, die einen Knoten des Spannbaumes mit einem noch nicht verbunden Knoten verbindet, folgt aus dem Lemma 4.23, daß sich der so konstruierte Spannbaum zu einem Spannbaum mit minimalem Gewicht erweitern läßt. Da zum Schluß ein Spannbaum konstruiert wurde, muß dieser dann minimales Gewicht haben. ■

4.6.2 Kruskal-Algorithmus (*)

Beim Prim-Algorithmus haben wir versucht einen Baum so lange wachsen zu lassen, bis er zu einem Spannbaum des Graphen wurde. Ein anderer Ansatz ist es nun einen Wald zu betrachten, in dem in jedem Schritt eine Kante minimalen Gewichts gewählt wird, die zwei Bäume verbindet. Der Algorithmus endet, wenn es nur noch einen Baum gibt, der dann offensichtlich ein Spannbaum sein muß. Dieser Algorithmus heißt nach seinem Entwickler J.B. Kruskal *Kruskal-Algorithmus* und ist in Abbildung 4.27 angegeben. Den Algorithmus von Kruskal kann man als reinen Greedy-Algorithmus bezeichnen, da er im Gegensatz zum Prim-Algorithmus die Kanten nur nach ihrem Gewicht auswählt. Beim Prim-Algorithmus war ja noch die Nebenbedingung zu erfüllen, daß die Kante inzident zum bisher konstruierten Teilbaum sein muß.

```
Kruskal(weighted_Graph G)
{
    int n = |V(G)|;
    set tree_edges = ∅; /* the set of edges of the spanning tree */
    int cost; /* the sum of the edge weights of the constructed tree */
    set T(n); /* n different sets */
    int t = n /* number of active sets in T */
    for (int i = 0; i < n; i++)
        T(i) = {i};
    heap h = create_heap(E, γ); /* Create a heap with all edge weights */

    while (t > 1)
    { /* combine trees */
        /* choose a cheapest edge */
        edge (v, w)=delete_min(h);
        i = find(v)
        j = find(w)
        if (i ≠ j)
        { /* v and w join different trees */
            tree_edges = tree_edges ∪ {v, w};
            cost+ = γ({v, w});
            T(i) = union(T(i), T(j));
            t--;
        }
    }
}
```

Abbildung 4.27: Der Kruskal-Algorithmus zur Berechnung eines minimalen Spannbaumes

Die Korrektheit folgt analog wie beim Prim-Algorithmus aus dem Lemma 4.23. Die Zeitkomplexität hängt im wesentlichen von der Implementierung der Mengen T und der zugehörigen `union` und `find` Operationen ab. Wir präzisieren zunächst genauer was wir für Kruskal's Algorithmus benötigen. Wir benötigen eine Datenstruktur, die eine Partition der Elemente aus $\{0, \dots, n-1\}$ in Mengen $T(i)$ unterstützt. Auf dieser Menge von Mengen benötigen wir die beiden folgenden Operationen:

int find(int i**)** Gibt den Index der Menge zurück, in der sich das Element i befindet.

int union(int i**,** j**)** Verschmilzt die beiden Mengen mit den Indizes i und j zu einer neuen Menge mit Index k, der von union zurückgegeben wird.

Datenstrukturen mit den beiden zu unterstützenden Operationen werden als *Union-Find-Algorithmen* bezeichnet. Damit beschäftigen wir uns im folgenden Abschnitt. Dort werden wir sehen, daß eine Folge von $f(n)$ `union` und `find` Operationen in Zeit $O(n \log^*(n) + f(n) \log^*(n))$ realisiert werden kann. Damit erhalten wir für Kruskal's Algorithmus das folgende Theorem:

Theorem 4.25 *Der Kruskal-Algorithmus benötigt zum Berechnen eines minimalen Spannbaumes für einen Graphen $G = (V, E)$ Zeit $O(|E| \log(|V|))$.*

Beweis: Die meiste Zeit wird für den Heap verbraucht, nämlich $O(|E| \log(|V|))$, da im schlimmsten Fall für jede Kante des Graphen eine `delete_min` Operation ausgeführt wird. Die Union-Find-Aufrufe selbst können in Zeit $O(|E| \log^*(|V|))$ realisiert werden, wie wir im folgenden Abschnitt sehen werden. ∎

Der Kruskal Algorithmus ist also schlechter als der Prim-Algorithmus. Allerdings können wir auf eine Priority Queue hier verzichten, und die Knoten auch einfach nach ihren Gewichten sortieren. Wenn man nun weiß, daß der zulässige Bereich der Kantengewichte nur polynomiell in der Anzahl der Kanten ist, können wir die Kanten nach ihren Gewichten auch mit dem verallgemeinerten Bucket-Sort in Zeit $O(|E|)$ sortieren. Die entsprechenden `delete_min` Operationen kosten dann je nur $O(1)$.

Theorem 4.26 *Sei $G = (V, E)$ ein Graph mit ganzzahligen Kantengewichten, so daß die Länge des Intervalls zulässiger Kantengewichte polynomiell in $|E|$ ist. Dann benötigt der Kruskal-Algorithmus basierend auf Union-Find-Algorithmen und dem verallgemeinerten Bucket-Sort zum Berechnen eines minimalen Spannbaumes für einen Graphen $G = (V, E)$ Zeit $O(|E| \log^*(|V|))$.*

Damit kann unter Umständen der Kruskal-Algorithmus für dünne Graphen (wenn also $|E| = o(n \log(n)/ \log^*(n))$) effizienter sein als der Prim-Algorithmus.

4.7 Interludium: Union-Find Algorithmen (*)

In diesem Abschnitt geben wir zwei Realisierungen von Union-Find-Algorithmen an. Dabei werden wir die Mengen einmal durch Listen sowie Arrays und einmal durch Bäume darstellen.

4.7.1 Darstellung von Mengen durch Listen (*)

Die erste Darstellung ist ziemlich naheliegend. Wir verwenden ein Array von ganzen Zahlen, welches uns den Index angibt, in welcher Menge ein gegebenes Element i liegt, und für jede Menge eine Liste, die auflistet welche Elemente sich in der Menge befinden. Die Datenstruktur und die Operationen sind in Abbildung 4.28 angegeben.

```
UNION-FIND-DATENSTRUKTUR1
int Set[n]; /* to which set belong elt i */
list<int> Members[n]; /* list of element belonging to set i */
int Size[n]; /* size of set i*/

int find(int i)
{
    return Set[i];
}

int union(int i, j)
{
    int SmallSet, BigSet;
    if (Size[Set[i]] < Size[Set[j]])
    {
        SmallSet = Set[i];
        BigSet = Set[j];
    }
    else
    {
        SmallSet = Set[j];
        BigSet = Set[i];
    }

    Size[BigSet]+ = Size[SmallSet];
    for each (m ∈ Members[SmallSet])
    {
        append(m, Members[BigSet]);
        Set[m] = BigSet;
    }
    return BigSet;
}
```

Abbildung 4.28: Realisierung eines Union-Find-Algorithmus mit Listen

Offensichtlich ist die Zeitkomplexität für einen `find`-Befehl $O(1)$. Der `union`-Befehl ist etwas komplexer und benötigt Zeit $O(|M_j|)$ wenn M_j kleiner als die Menge M_i ist, die es zu vereinigen gilt.

Theorem 4.27 *Bei der Implementierung mit Listen und Arrays für eine Grundmenge von n Elementen benötigt jede* `find` *Operation Zeit $O(1)$ und eine Folge von $n-1$* `union` *Operationen Zeit $O(n \log(n))$.*

Beweis: Die Aussage für die `find` Operationen ist klar. Für die Aussage über die `union` Operationen verwenden wir einen sogenannten *Buchhaltertrick*. Was passiert bei einer `union` Operation. Wir müssen die Elemente in der kleineren Menge an die größere anhängen und die Informationen über die Zugehörigkeit für die Element der kleineren Menge ändern. Die Kosten für eine `union` Operation sind, wie früher schon bemerkt, proportional zur Anzahl der Elemente in der kleineren Menge. Der Einfachheit nehmen wir an, daß die Kosten genau der Anzahl der Elemente in der kleineren Menge entsprechen.

Für jedes Element der Grundmenge errichten wir nun ein Konto. Auf diese Konten verteilen wir die Kosten der `union` Operationen. Um nun die Kosten einer `union` Operation zu bezahlen, belasten wir für jedes Element in der kleineren Menge sein Konto mit einer Einheit.

Wie groß kann nun am Ende der Schuldenstand eines Elementes sein? Jedes Element macht eine Einheit Schulden, wenn es sich in der kleineren Menge, der in der `union` Operation involvierten Mengen, befindet. Jedesmal wenn ein Element Schulden macht, verdoppelt sich mindestens die Größe der Menge, in der es sich befindet. Da die Grundmenge aus n Elementen besteht, kann jeder Knoten maximal $O(\log(n))$ Einheiten Schulden machen. Da es insgesamt n Elemente sind, können die Schulden am Ende der `union` Operationen höchstens $O(n \log(n))$ betragen. ■

4.7.2 Darstellung von Mengen durch Bäume (*)

Als zweite Möglichkeit speichern wir die Mengen als Bäume ab, wobei die Kanten hier von den Kindern zu den Eltern gerichtet sind. Am Anfang bildet jede einelementige Menge einen Baum, der aus nur einem Knoten besteht. Bei der `union` Operation wird dann die Wurzel des kleineren Teilbaumes (also der kleineren Menge) zum Kind der Wurzel des größeren Baumes (also der größeren Menge). Bei der `find` Operation wandern wir von dem dem Element zugeordneten Knoten bis zur Wurzel des zugehörigen Baumes. Der Index der Wurzel gibt dann den Namen der Menge an, in der sich das gesuchte Element befindet. Die zweite Variante der Realisierung eines Union-Find-Algorithmus ist in Abbildung 4.29 dargestellt, allerdings wird hier die

```
UNION-FIND-DATENSTRUKTUR2
int size[n];
int parent[n];
for (int i = 0; i < n; i++)
{
    size[i] = 1; /* correct for root of trees only */
    parent[i] = i; /* a root points to itself */
}

int find(int i)
{
    for (int j = i; parent[j] ≠ j; j = parent[j]);
    int root = j;
    /* Pathcompression*/
    for (int j = i; parent[j] ≠ j; j = parent[j])
        parent[j] = root;
    /* End of Pathcompression */
    return root;
}

int union(int i, j)
{ /* provided that i and j are roots */
    if (size[i] < size[j])
    { /* j becomes root */
        parent[i] = j;
        size[j]+ = size[i];
        return j;
    }
    else
    { /* i becomes root */
        parent[j] = i;
        size[i]+ = size[j];
        return i;
    }
}
```

Abbildung 4.29: Realisierung eines Union-Find-Algorithmus mit Bäumen

for-Schleife für die später erläuterte Pfadkompression noch nicht ausgeführt (siehe Seite 172).

Offensichtlich hat die `union` Operation nun Zeitkomplexität $O(1)$. Dafür ist die

Zeitkomplexität der `find` Operation gestiegen. Die Zeit für eine `find` Operation ist offensichtlich durch die maximale Höhe der entstandenen Teilbäume beschränkt.

Lemma 4.28 *Die in der zweiten Union-Find-Datenstruktur entstehenden Bäume mit einer Höhe h haben mindestens 2^h viele Knoten.*

Beweis: Wir beweisen diese Aussage mit vollständiger Induktion über die Höhe der Bäume.

Induktionsanfang ($h = 0$): Nach unserer Definition der Höhe, ist ein Baum mit Höhe 0 gerade der Baum, der aus einem Knoten besteht. Damit ist der Induktionsanfang gelegt.

Induktionsschritt ($h \to h + 1$): Ein Baum der Höhe $h + 1$ entsteht durch Anhängen eines Baumes mit Höhe h an einen anderen Baum. Nach Induktionsvoraussetzung hat der Baum mit Höhe h mindestens 2^h viele Knoten. Nach Konstruktion wird die Wurzel dieses Baumes nur dann ein Kind eines anderen Baumes, wenn dieser mindestens genau soviele Knoten hat. Also hat der Baum mit Höhe $h+1$ mindestens $2^h + 2^h = 2^{h+1}$ viele Knoten. ■

Damit kostet nun eine `find` Operation $O(\log(n))$. Da bei unserem Algorithmus aber durchaus n `find` Operationen auftauchen können, haben wir mit diesem zweiten Union-Find-Algorithmus erst einmal nichts gewonnen.

4.7.3 Pfadkompression (*)

Wir können allerdings die `find` Operation noch beschleunigen. Jedesmal wenn wir bei einer `find` Operation einen Pfad durch den zugehörigen Baum laufen, machen wir alle besuchten Knoten zu Kindern der Wurzel. Dadurch wird diese `find` Operation zwar nicht billiger, allerdings werden viele der folgenden `find` Operationen erheblich billiger. Die oben genannte Umordnung des Baumes, in dem Knoten des Suchpfades Kinder der Wurzel werden, nennt man allgemein *Pfadkompression* (engl. *Pathcompression*). Die zweite Variante der Realisierung eines Union-Find-Algorithmus mit Pfadkompression wurde ja bereits in Abbildung 4.29 dargestellt. Zu dessen Analyse müssen wir noch die folgenden Funktionen definieren:

Definition 4.29 *Es ist* $2 \upuparrows 0 := 1$ *und* $2 \upuparrows n := 2^{2 \upuparrows (n-1)}$. *Mit* $\log^*$ *bezeichnen wir die Umkehrfunktion von* $2 \upuparrows (\cdot)$, *also* $\log^*(n) = \min\{k | 2 \upuparrows k \geq n\}$.

Man beachte, daß $2 \uparrow\uparrow (\cdot)$ eine sehr schnell wachsende und damit $\log^*$ eine sehr langsam wachsende Funktion ist. Es gilt $2 \uparrow\uparrow 1 = 2$, $2 \uparrow\uparrow 2 = 4$, $2 \uparrow\uparrow 3 = 16$, $2 \uparrow\uparrow 4 = 65536$, $2 \uparrow\uparrow 5 = 2^{65536}$. Damit ist für alle praktischen Zwecke $\log^*(n) \leq 5$, da $2^{65536} \approx (2^{10})^{655} \approx 10^{655} \gg 10^{81}$, und damit $2 \uparrow\uparrow 5$ schon wesentlich größer ist, als die vermutete Anzahl von Atomen im Weltall. Damit können Eingaben der Größe $2 \uparrow\uparrow 5$ schon gar nicht mehr konstruiert werden!

Sei σ eine Folge von `union` und `find` Befehlen. Sei W im folgenden der Wald, der entstanden ist, wenn *nur* die `union` Operationen in σ ausgeführt werden. Mit dem *Rang* eines Knotens v wollen wir nun die Länge des längsten Pfades von einem Blatt zu v in diesem Wald W bezeichnen.

Lemma 4.30 *Es gibt maximal $n/2^r$ Knoten mit Rang r im Wald W.*

Beweis: Zuerst bemerken wir, daß verschiedene Knoten mit demselben Rang verschiedene Vorgänger im Baum haben müssen. Beachte, daß Vorgänger hier Knoten auf den Pfaden zu den Blättern sind, da die Kanten ja zur Wurzel gerichtet sind. Nach Lemma 4.28 hat jeder Knoten mit Rang r mindestens 2^r viele verschiedene Vorgänger (sich selbst eingeschlossen). Gäbe es mehr als $n/2^r$ viele Knoten vom Rang r, so müßte der Wald mehr als n Knoten besitzen. ■

Damit können wir sofort folgern, daß kein Knoten einen Rang größer als $\log(n)$ hat. Ist irgendwann bei der Abarbeitung der Folge σ von `union` und `find` Operationen v ein echter Vorgänger von w, dann ist der Rang von v kleiner als der Rang von w. Dies folgt aus der Tatsache, daß die `find` Befehle die Vorgängerrelation respektieren, d.h. ein Knoten wird bei der Pfadkompression nur dann direkter Vorgänger eines anderen Knotens, wenn er bereits ein Vorgänger war. Also ist v auch ein Vorgänger von w, wenn man die `find` Operationen aus σ nicht ausführt. Das impliziert sofort die obige Aussage über die Ränge der Knoten.

Nun teilen wir die Knoten in Gruppen auf. Die Knoten mit Rang i ordnen wir der Gruppe mit Nummer $\log^*(i)$ zu. Zum Beispiel gelangen die Knoten mit Rängen zwischen 5 und 16 in die Gruppe mit Nummer 3. Wir verwenden auch hier wieder die Buchhaltermethode und nehmen dazu an, daß die Kosten der `find` Operation genau der Länge des Pfades entsprechen, der zum Auffinden der Wurzel durchlaufen wird. Wir belasten auch diesmal wieder die einzelnen Knoten mit den Kosten, und berechnen hinterher die Gesamtschuld aller Knoten.

Zuerst belasten wir jeden Knoten des Suchpfades mit einer Kosteneinheit. Offensichtlich bilden die Ränge der Knoten auf dem Suchpfad zur Wurzel eine aufsteigende Folge. Nun begleichen die Wurzel und diejenigen Knoten, deren Elter in einer anderen Gruppe liegen, diese neue Schuld sofort. Da es maximal $\log^*(n) + 1$ viele verschiedene Gruppen gibt, kostet jeder `find` Befehl $\log^*(n) + 1$.

Wir groß ist nun die verbleibende Schuld eines Knotens am Ende des Ablaufs eines Union-Find-Algorithmus? Immer wenn ein Knoten Schulden macht, erhält er wegen der Pfadkompression einen neuen Elter, nämlich die Wurzel des Baumes. Der Rang seines Elters wächst also mindestens um 1, mit Ausnahme eines Kindes einer Wurzel (aber diese Knoten haben ihre Schuld ja sofort beglichen). In der Gruppe g befinden sich nur Elemente, die einen Rang von mindestens $2 \upuparrows (g-1)+1$ und höchstens $2 \upuparrows g$ haben. Also macht ein Knoten maximal $2 \upuparrows g - 2 \upuparrows (g-1)$ Schulden, bevor er einen Elter in einer höheren Gruppe zugewiesen bekommt. Sei nun $G(g)$ die Anzahl der Knoten in Gruppe g. Dann erhalten wir mit der obigen Beobachtung, daß es nur $n/2^r$ Knoten mit demselben Rang geben kann (siehe Lemma 4.30), folgende Abschätzung:

$$G(g) \leq \sum_{r=2\upuparrows(g-1)+1}^{2\upuparrows g} \frac{n}{2^r} \leq \frac{n}{2^{2\upuparrows(g-1)}} \sum_{i=1}^{\infty} \frac{1}{2^i} \leq \frac{n}{2^{2\upuparrows(g-1)}} = \frac{n}{2 \upuparrows g}$$

Jeder Knoten in der Gruppe mit Nummer g macht höchstens Schulden in Höhe von $(2 \upuparrows g) - (2 \upuparrows (g-1))$. Damit machen alle Knoten in der Gruppe mit Nummer g Schulden in Höhe von $O((\frac{n}{2\upuparrows g})((2 \upuparrows g) - (2 \upuparrows (g-1)))) = O(n)$. Da es maximal $\log^*(n) + 1$ verschiedene Gruppen gibt, ist die Gesamtschuld $O(n \log^*(n))$.

Theorem 4.31 *Zur Ausführung von $f(n)$* `union` *und* `find` *Operationen benötigt der Union-Find-Algorithmus basierend auf einer Baumdarstellung der Mengen maximal Zeit $O(n \log^*(n) + f(n) \log^*(n))$.*

Beweis: Die Behauptung folgt unmittelbar aus der vorherigen Diskussion. Der erste Term entspricht den entstandenen Schulden, der zweite Term den sofort bezahlten Kosten. ∎

Solange $f(n) = O(n)$ ist, gilt das folgenden Korollar

Korollar 4.32 *Zur Ausführung von $O(n)$* `union` *und* `find` *Operationen benötigt der Union-Find-Algorithmus basierend auf einer Baumdarstellung der Mengen maximal Zeit $O(n \log^*(n))$.*

5 Arithmetik

5.1 Berechnung von Potenzen

In diesem Kapitel wollen wir einige wichtige Algorithmen zu Problemen aus der Arithmetik vorstellen. Zuerst wiederholen wir an dieser Stelle ein fundamentales Ergebnis aus der Einleitung, welches in der Praxis oft Anwendung findet.

Theorem 5.1 *Sei $H = (M, \circ)$ eine Halbgruppe. Dann kann für jedes $m \in M$ die n-te Potenz von m (m^n, wobei $m^1 = m$ und $m^{n+1} = m \circ m^n$) mit $O(\log(n))$ vielen Anwendungen der Operation $\circ$ berechnet werden.*

5.2 Euklidischer Algorithmus (*)

In diesem Abschnitt wollen wir den wohl ältesten und bekanntesten Algorithmus überhaupt vorstellen, den *Euklidischen Algorithmus* und seine Erweiterung.

5.2.1 Grundalgorithmus (*)

Wie allseits bekannt ist, berechnet der Euklidische Algorithmus den größten gemeinsamen Teiler von zwei gegebenen natürliche Zahlen. In Abbildung 5.1 ist die rekursive Variante des Euklidischen Algorithmus angegeben. Die Korrektheit der Algorithmen folgt aus der folgenden Identität.

Theorem 5.2 $\forall a \geq b: \ \mathrm{ggt}(a, b) = \mathrm{ggt}(a \bmod b, b)$.

Beweis: Sei $d = \mathrm{ggt}(a, b)$ und $d' = \mathrm{ggt}(b, a \bmod b)$.

Nach Definition des ggt gilt, daß $d|b$ und $d|a$. Da $a \bmod b = a - (a \operatorname{div} b) \cdot b$, folgt daß $d|(a \bmod b)$ und somit $d|d'$.

```
EUKLID(INT A,B)
{
    if (a < b) swap(a,b);
    if (b == 0)
        return a;
    else
        return Euklid(b, a mod b);
}
```

Abbildung 5.1: Euklidischer Algorithmus

Außerdem gilt nach der Definition des ggt, daß $d'|b$ und $d'|(a \bmod b)$. Da nun $a = (a \operatorname{div} b) \cdot b + (a \bmod b)$, folgt unmittelbar daß $d'|a$ und somit $d'|d$.

Aus $d|d'$ und $d'|d$ folgt nun unmittelbar, daß $d = d'$ und somit die Behauptung. ∎

Theorem 5.3 *Der Euklidische Algorithmus berechnet den größten gemeinsamen Teiler von zwei ganzen Zahlen a und b mit $O(\log(a+b))$ arithmetischen Operationen.*

Beweis: Wir müssen nur noch die Aussage über die Laufzeit zeigen. Dazu genügt es folgende Aussage zu beweisen.

> *Wenn die Berechnung von $\operatorname{ggt}(a,b)$ für $a > b$ genau k Rekursionen erfordert, so gilt: $a \geq F_k$ und $b \geq F_{k-1}$, wobei F_i die i-te Fibonacci Zahl ist.*

Diese Aussage beweisen wir mit vollständiger Induktion über k.

Induktionsanfang ($k = 1$): Wenn nur eine Rekursion aufgerufen wird, so ist $b = 0 \geq F_0$ und $a \geq b + 1 = 1 = F_1$.

Induktionsschritt ($k \to k+1$): Die Rekursion wird für $\operatorname{ggt}(b, a \bmod b)$ genau k-mal aufgerufen. Da $b > (a \bmod b)$ können wir die Induktionsvoraussetzung anwenden und erhalten somit, daß $b \geq F_{k-1}$ und $a \bmod b \geq F_k$. Daraus folgt mit $a > b$, daß gilt:

$$\begin{aligned} a &= (a \operatorname{div} b) \cdot b + a \bmod b \\ &\overset{\text{I.V.}}{\geq} (a \operatorname{div} b) \cdot F_{k-1} + F_k \\ &\geq 1 \cdot F_{k-1} + F_k \\ &= F_{k+1} \end{aligned}$$

Damit haben wir die obige Aussage bewiesen. Da die Fibonacci-Zahlen, wie in der Einleitung bewiesen, exponentiell im Index wachsen, impliziert dies die Behauptung über die Laufzeit. ∎

```
EXTENDED_EUKLID(INT a, INT b)
{
    /* Assuming that the algorithm returns (d, x, y), */
    /* where d = gcd(a, b) = ax + by */
    if (a < b) swap(a,b);
    if (b == 0)
        return (a, 1, 0);
    else
    {
        (d', x', y') = Extended_Euklid(b, a mod b);
        x = y';
        y = x' - y' · (a div b);
        return (d', x, y);
    }
}
```

Abbildung 5.2: Erweiterter Euklidischer Algorithmus

5.2.2 Erweiterte Version (*)

Wir wollen nun noch den *erweiterten Euklidischen Algorithmus* vorstellen. Aus grundlegenden Mathematik-Vorlesungen (wie z.B. Algebra, Diskrete Strukturen) ist das folgende Theorem bekannt.

Theorem 5.4 $\forall a, b \in \mathbb{N} : \exists x, y \in \mathbb{Z} : xa + by = \mathrm{ggt}(a, b)$.

Ziel des erweiterten Euklidischen Algorithmus ist es nun, auch die Zahlen x und y zu bestimmen. Mit Entwicklung des erweiterten Euklidischen Algorithmus werden wir dieses Theorem nebenbei mitbeweisen. Aufbauend auf der rekursiven Variante des Euklidischen Algorithmus läßt sich der erweiterte Euklidische Algorithmus angeben, der in Abbildung 5.2 dargestellt ist. Die Korrektheit folgt aus der folgenden Überlegung. Wie beim normalen Euklidischen Algorithmus ist offensichtlich d der größte gemeinsame Teiler. Betrachten wir nun einen Aufruf $\mathrm{ggt}(a, b)$. Ist nun $b = 0$, so sind wir fertig. Andernfalls haben wir rekursiv $\mathrm{ggt}(b, a \bmod b)$ aufgerufen. Dafür erhalten wir die folgende Darstellung zurück:

$$d' = b \cdot x' + (a \bmod b) \cdot y'.$$

Da nach Theorem 5.2 $d' = \mathrm{ggt}(b, a \bmod b) = \mathrm{ggt}(a, b) = d$ gilt, können wir nun einfach umformen:

$$\begin{aligned} d &= d' \\ &= b \cdot x' + (a \bmod b) \cdot y' \end{aligned}$$

$$\begin{aligned} &= b \cdot x' + (a - (a \text{ div } b) \cdot b) \cdot y' \\ &= a \cdot y' + b \cdot (x' - (a \text{ div } b) \cdot y') \end{aligned}$$

Damit erhalten wir also unsere Parameter $x = y'$ und $y = x' - (a \text{ div } b) \cdot y'$, wie sie im Algorithmus gesetzt wurden.

Theorem 5.5 *Der erweiterte Euklidische Algorithmus berechnet für a und b den größten gemeinsamen Teiler und die dazugehörige Linearkombination aus a und b mit logarithmisch vielen arithmetischen Operationen.*

5.2.3 Anwendung des erweiterten Euklidischen Algorithmus (*)

Für eine Anwendung betrachten wir $(\mathbb{Z}_n^*, *_n)$, wobei $\mathbb{Z}_n^*$ die Menge der natürlichen Zahlen aus $\{1, \ldots, n-1\}$ ist, die relativ prim zu n sind. Die Multiplikation $*_n$ ist hierbei die normale Multiplikation in $\mathbb{Z}$ modulo n. Solche Gruppen werden sehr oft im Bereich der Kryptographie verwendet (z.B. für den RSA-Algorithmus). Hier ist man oft an der multiplikativen Inversen einer Zahl interessiert. Dazu genügt es nun für eine Zahl a mit Hilfe des erweiterten Euklidischen Algorithmus die Linearkombination des größten gemeinsamen Teilers von a und n zu berechnen. Als Ergebnis erhalten wir ja das Tripel (d, x, y), wobei $d = 1$ sein muß, da andernfalls a nicht aus der multiplikativen Gruppe modulo n stammt. Darüber hinaus hat dieses Tripel noch die folgende Eigenschaft:

$$1 = a \cdot x + n \cdot y.$$

Da wir in $\mathbb{Z}_n^*$ modulo n rechnen, bedeutet dies, daß $1 = ax \bmod n$ ist, also ist x das multiplikative Inverse von a in $\mathbb{Z}_n^*$.

Zum Ende dieses Abschnittes wollen wir noch hinweisen, daß diese Algorithmen natürlich nicht nur für die natürlichen Zahlen, sondern für alle euklidischen Ringe gültig sind.

5.3 Multiplikation ganzer Zahlen (*)

In diesem Abschnitt wollen wir uns mit der Komplexität der Multiplikation auf der Bitebene beschäftigen. Bekanntermaßen benötigt die Schulmethode zur Multiplikation von zwei n-stelligen Binärzahlen $O(n^2)$ Bitoperationen. Wir wollen nun einen Divide-and-Conquer Algorithmus angeben, der mit wesentlich weniger Bitoperationen auskommt.

```
                    x x x x x x x x x x
                  x x x x x x x x x x
                x x x x x x x x x x
              x x x x x x x x x x
            x x x x x x x x x x
          x x x x x x x x x x
        x x x x x x x x x x
      x x x x x x x x x x
    x x x x x x x x x x
  x x x x x x x x x x
 ---------------------------------------
x x x x x x x x x x x x x x x x x x x x
```

Abbildung 5.3: Schematische Darstellung der Schulmethode zur Multiplikation zweier Binärzahlen

5.3.1 Analyse der Schulmethode (*)

Zunächst wollen wir die Schulmethode noch genauer analysieren. Dazu betrachten wir die schematische Darstellung der Schulmethode in Abbildung 5.3. Dabei fallen eventuell einige Zeilen weg, nämlich genau diejenigen, für die das entsprechende Bit im zweiten Multiplikanden gerade 0 ist. Zur Addition einer Zeile auf das Gesamtergebnis sind also von oben nach unten je maximal n 1-Bit Additionen und eventuell je n 1-Bit Additionen für Überträge zu berücksichtigen. Nur die erste Zeile benötigt keine Additionen. Damit benötigt die Schulmethode maximal $2n(n-1)$ arithmetische Operationen.

Lemma 5.6 *Bei der Schulmethode sind maximal $2n^2 - 2n$ Operationen notwendig, die zwei Bits addieren.*

5.3.2 Ein Divide-and-Conquer Algorithmus (*)

Die Eingabe seien zwei n-stellige Bitzahlen $(a_{n-1}, \ldots, a_0)$ und $(b_{n-1}, \ldots, b_0)$. Eine erste Idee für einen Divide-and-Conquer Algorithmus ist, jede der beiden Zahlen aufzuteilen, und die einzelnen entstehenden Produkte rekursiv zu multiplizieren. Sei hierzu:

$$\begin{aligned}
a^{(1)} &= (a_{n-1}, \ldots, a_{\lceil \frac{n}{2} \rceil}),\\
a^{(0)} &= (a_{\lceil \frac{n}{2} \rceil - 1}, \ldots, a_0),\\
b^{(1)} &= (b_{n-1}, \ldots, b_{\lceil \frac{n}{2} \rceil}),\\
b^{(0)} &= (b_{\lceil \frac{n}{2} \rceil - 1}, \ldots, b_0),
\end{aligned}$$

so daß $a = a^{(1)} \cdot 2^{\lceil \frac{n}{2} \rceil} + a^{(0)}$ und $b = b^{(1)} \cdot 2^{\lceil \frac{n}{2} \rceil} + b^{(0)}$. Dann können wir $a \cdot b$ wie folgt umrechnen.

$$\begin{aligned} a \cdot b &= \left(a^{(1)} \cdot 2^{\lceil \frac{n}{2} \rceil} + a^{(0)}\right) \cdot \left(b^{(1)} \cdot 2^{\lceil \frac{n}{2} \rceil} + b^{(0)}\right) \\ &= a^{(1)} \cdot b^{(1)} \cdot 2^{2\lceil \frac{n}{2} \rceil} + a^{(1)} \cdot b^{(0)} \cdot 2^{\lceil \frac{n}{2} \rceil} + a^{(0)} \cdot b^{(1)} \cdot 2^{\lceil \frac{n}{2} \rceil} + a^{(0)} \cdot b^{(0)} \end{aligned}$$

Die Multiplikationen mit Zweierpotenzen sind im Rechner sehr leicht durch Shifts auszuführen und kosten uns hier im wesentlichen nichts. Aus diesem Grunde werden wir sie hier, wie auch schon bei der Schulmethode, nicht berücksichtigen. Wir müssen also 4 rekursive Multiplikationen von Zahlen der Länge $\lceil \frac{n}{2} \rceil$ ausführen und hinterher die Ergebnisse mit je drei Shifts und Additionen wieder zusammensetzen. Die Additionen sind offensichtlich in Zeit $O(n)$ auszuführen. Damit erhalten wir die folgende Rekursionsgleichung:

$$T(n) = 4T(\lceil n/2 \rceil) + O(n).$$

Leider hat diese Rekursionsgleichung als Lösung $T(n) = O(n^2)$ (siehe auch Abschnitt 2.5).

Wir müssen uns also irgendwie geschickter anstellen. Dazu berechnen wir erst einmal die folgenden drei Produkte: $a^{(1)} \cdot b^{(1)}$, $a^{(0)} \cdot b^{(0)}$, und $(a^{(1)} + a^{(0)}) \cdot (b^{(1)} + b^{(0)})$. Was hilft uns das? Wir benötigen noch die Produkte $a^{(1)} \cdot b^{(0)}$ und $a^{(0)} \cdot b^{(1)}$. Diese haben wir aber bereits berechnet! Denn es gilt:

$$((a^{(1)} + a^{(0)}) \cdot (b^{(1)} + b^{(0)})) - (a_1 \cdot b_1 + a_0 \cdot b_0) = a^{(1)} \cdot b^{(0)} + a^{(0)} \cdot b^{(1)}.$$

Wir haben also eigentlich nicht die fehlenden beiden Produkte sondern deren Summe berechnet. Aber genau diese Summe wird zur Erstellung des Gesamtergebnisses benötigt, wir sparen daher hinterher sogar eine Addition ein. Allerdings haben wir vorher für diese Produktbildung zwei Additionen spendiert.

5.3.3 Analyse des Divide-and-Conquer-Algorithmus (*)

Für den Algorithmus des vorherigen Abschnitts erhalten wir die folgende Rekursionsgleichung

$$T(n) = 2T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil + 1) + O(n).$$

Durch die Addition der beiden Zahlen mit $\lceil n/2 \rceil$ vielen Stellen entstehen nun Zahlen, die durch $\lceil n/2 \rceil + 1$ viele Bits zu repräsentieren sind. Bestimmen wir zuerst die Konstante im O noch etwas genauer. Zuerst müssen je zwei Paare von $\lceil n/2 \rceil$-stelligen Binärzahlen addiert werden. Das kostet uns insgesamt höchstens $2n$ 1-Bit Additionen. Zum Schluß addieren wir zunächst die beiden Subtrahenden, bevor wir sie vom Minuenden abziehen. Dazu benötigen wir wiederum $2n$ 1-Bit-Additionen, da die beiden Minuenden ja n-stellig sind. Für die Subtraktion benötigen wir dann nochmals $2n$ 1-Bit-Subtraktionen. Damit erhalten wir insgesamt

$$T(n) = 2T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil + 1) + 6n.$$

Wir wollen nun die folgende Rekursionsgleichung lösen:

$$T(n) = 3T(\lceil n/2 \rceil + 1) + 6n.$$

Offensichtlich liefert eine Lösung dieser Rekursionsgleichung auch eine Abschätzung für die Lösung unserer ursprünglichen Rekursionsgleichung. Zuerst halten wir die folgende nützliche Beziehung fest:

$$\left\lceil \frac{\lceil \frac{n}{2^i} \rceil + 2}{2} \right\rceil + 1 = \left\lceil \frac{n}{2^{i+1}} \right\rceil + 2.$$

Diese Beziehung benötigen wir gerade, wenn wir die Rekursionsgleichung durch Iteration lösen wollen. Der Einfachheit halber wollen wir annehmen, daß es sich bei n um eine Zweierpotenz handelt.

$$\begin{aligned}
T(n) &= 3 \cdot T\left(\left\lceil \frac{n}{2} \right\rceil + 1\right) + 6n \\
&\leq 3^2 \cdot T\left(\left\lceil \frac{n}{2^2} \right\rceil + 2\right) + 3 \cdot 6 \left\lceil \frac{n}{2} \right\rceil + 2 + 6n \\
&= 3^3 \cdot T\left(\left\lceil \frac{n}{2^3} \right\rceil + 2\right) + 3^2 \cdot 6 \left\lceil \frac{n}{2^2} \right\rceil + 2 + 3 \cdot 6 \left\lceil \frac{n}{2} \right\rceil + 2 + 6n \\
&\ \text{nach } k \text{ Iterationen erhalten wir} \\
&\leq 3^k \cdot T\left(\left\lceil \frac{n}{2^k} \right\rceil + 2\right) + \sum_{i=0}^{k-1} \left(3^i \cdot 6 \left\lceil \frac{n}{2^i} \right\rceil + 2\right) \\
&\ \text{für } k = \log(n) \text{ gilt dann} \\
&\leq 3^{\log(n)} \cdot T\Bigg(\underbrace{\left\lceil \frac{n}{2^{\log(n)}} \right\rceil + 2}_{=3}\Bigg) + \sum_{i=0}^{\log(n)-1} \left(3^i \cdot 6 \left\lceil \frac{n}{2^i} \right\rceil + 2\right) \\
&\leq 2^{\log(3) \cdot \log(n)} \cdot T(3) + 6 \sum_{i=0}^{\log(n)-1} 3^i \left(\frac{n}{2^i} + 2\right) \\
&\ \text{da } T(3) \leq 9 \\
&\leq 9n^{\log(3)} + 6n \sum_{i=0}^{\log(n)-1} \frac{3^i}{2^i} + 2\log(n) \\
&\leq 9n^{\log(3)} + 6n \frac{\frac{3}{2}^{\log(n)} - 1}{\frac{3}{2} - 1} + 2\log(n)
\end{aligned}$$

$$
\begin{aligned}
&\leq 9n^{\log(3)} + 12n2^{\log(1.5)\log(n)} + 2\log(n)\\
&\leq 9n^{\log(3)} + 12n^{\log(3)} + 2\log(n)\\
&\leq 21n^{\log(3)} + 2\log(n) \approx 21n^{1.585} + 2\log(n)
\end{aligned}
$$

Damit haben wir also das folgende Theorem bewiesen.

Theorem 5.7 *Zwei n-stellige Binärzahlen können mit $21n^{\log(3)} + 2\log(n)$ vielen Bit-Operationen multipliziert werden.*

Allerdings ist diese Methode erst für Binärzahlen der Länge 300 effizienter als die Schulmethode.

5.3.4 Verbesserung des Divide-and-Conquer Verfahrens (*)

Es sieht auf den ersten Blick so aus, als wäre der Divide-and-Conquer Algorithmus nicht praktikabel. Mit etwas Grips erkennt man, daß es für kleine Binärzahlen günstig ist, auf die Schulmethode umzusteigen. Für eine genauere Analyse brechen wir die Rekursion bei Binärzahlen der Länge $2^m + 2$ ab.

$$
\begin{aligned}
T(n) &\leq 3^k T\left(\frac{n}{2^k} + 2\right) + \sum_{i=0}^{k-1} \left(3^i \cdot 6 \left\lceil \frac{n}{2^i} \right\rceil + 2\right)\\
&\quad \text{für } k = \log(n) - m \text{ gilt dann}\\
&\leq 3^{\log(n)-m} T(2^m + 2) + \sum_{i=0}^{\log(n)-m-1} \left(3^i \cdot 6 \left\lceil \frac{n}{2^i} \right\rceil + 2\right)\\
&\leq 2^{\log(3)\cdot\log(n)} \frac{T(2^m+2)}{3^m} + 6n \sum_{i=0}^{\log(n)-m-1} \frac{3^i}{2^i} + 2\log(n)\\
&\leq n^{\log(3)} \frac{T(2^m+2)}{3^m} + 6n \frac{\frac{3}{2}^{\log(n)-m} - 1}{\frac{3}{2} - 1} + 2\log(n)\\
&\leq n^{\log(3)} \left(\frac{2 * (2^m+2)(2^m+1)}{3^m} + 12\frac{2^m}{3^m}\right) + 2\log(n)
\end{aligned}
$$

Wir überlegen uns nun, für welches m die 'Konstante' vor dem Term $n^{\log(3)}$ am kleinsten wird. Dies kann man sehr leicht aus Abbildung 5.4 ablesen. Der Umschaltzeitpunkt sollte bei Binärzahlen der Länge zwischen $10 = 2^3 + 2$ und $18 = 2^4 + 2$ erfolgen. Daraus folgt, daß die Konstante auf 10 sinkt.

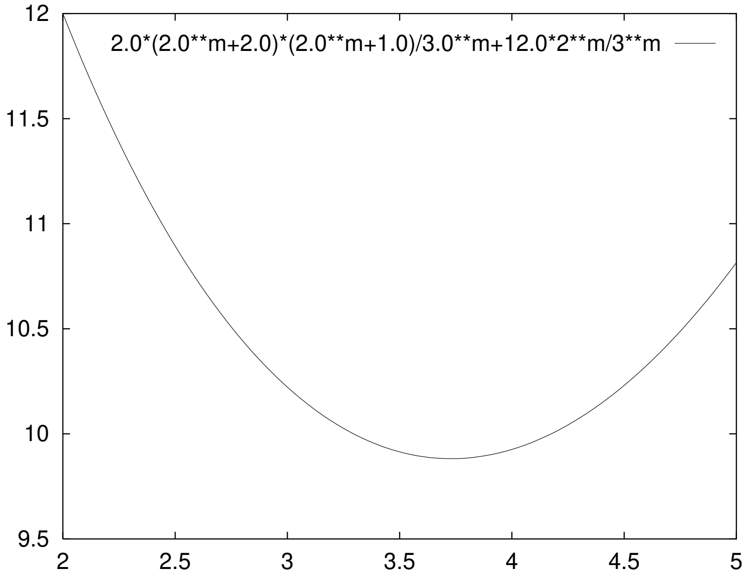

Abbildung 5.4: Wahl des Umschaltens bei der Multiplikation von Binärzahlen

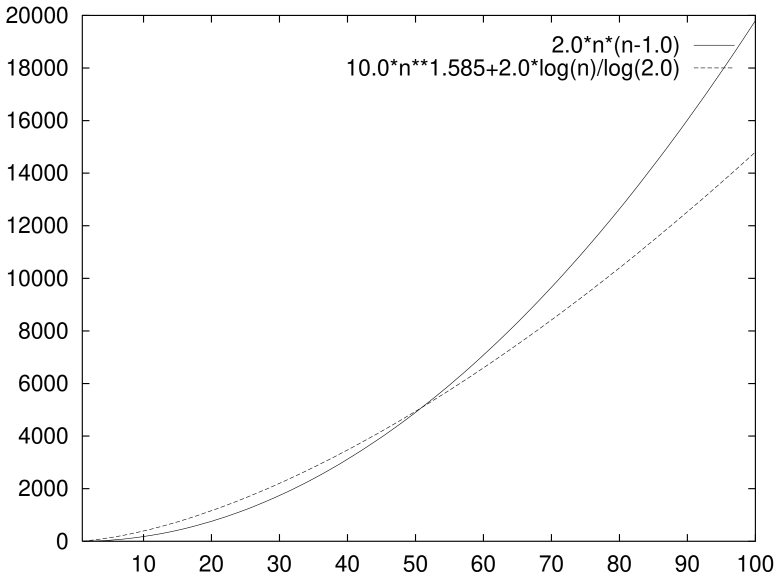

Abbildung 5.5: Vergleich Schulmethode gegen modifizierten Divide-and-Conquer Ansatz

Theorem 5.8 *Zwei n-stellige Binärzahlen können mit $10n^{\log(3)} + 2\log(n)$ vielen Bit-Operationen multipliziert werden.*

Damit ist der modifizierte Divide-and-Conquer Ansatz schon für Binärzahlen ab etwa 50 Stellen effizienter als die Schulmethode, was man leicht aus Abbildung 5.5 ablesen kann. Man sollte hier anmerken, daß bei Verschlüsselungsalgorithmen mittlerweile mit Zahlen von 500 Binärstellen und mehr gerechnet wird.

Wie man sich denken kann, ist die Multiplikation von binär dargestellten Zahlen ein gut untersuchtes Gebiet. Es gibt bereits wesentlich bessere Verfahren zum Berechnen eines Produkts. Das beste bislang bekannte Verfahren geht auf A. Schönhage und V. Strassen aus dem Jahre 1971 zurück und basiert auf der diskreten Fourier-Transformation.

Theorem 5.9 *Zwei n-stellige Binärzahlen können mit $O(n \log(n) \operatorname{loglog}(n))$ vielen Bit-Operationen multipliziert werden.*

5.4 Matrizenmultiplikation (*)

Nachdem wir uns nun mit der einfachen Multiplikation von Zahlen beschäftigt haben, kommen wir nun zur Matrizenmultiplikation. Dies ist ein sehr wichtiges Problem der Informatik, da Matrizen aller Orten verwendet werden (siehe auch Abschnitt 4.4.2 über die Transitive Hülle von Graphen). Außerdem kann man für viele Standardprobleme der linearen Algebra zeigen, daß deren Komplexität höchstens so groß oder gar äquivalent zur Komplexität der Matrizenmultiplikation ist.

5.4.1 Der Algorithmus von Strassen (*)

Zuerst halten wir fest, daß die Schulmethode zur Matrizenmultiplikation $2n^3 - n^2$ gewöhnliche arithmetische Operationen benötigt. Ähnlich wie bei der Multiplikation von Zahlen, verwenden wir wieder einen Divide-and-Conquer Algorithmus. Seien im folgenden A und B zwei $n \times n$-Matrizen, die wir multiplizieren wollen. Der Einfachheit halber nehmen wir in diesem Abschnitt an, daß n immer eine Zweierpotenz ist. Die im folgenden angegebenen Algorithmen lassen sich leicht für den Fall modifizieren, daß n keine Zweierpotenz ist. Wir teilen nun eine Matrix in vier $n/2 \times n/2$-Matrizen auf:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}.$$

Wenn wir die Produktmatrix $C = A \cdot B$ ebenfalls so aufteilen, gilt bekanntermaßen, daß

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \end{aligned}$$

Wenn wir nun diese Schema für unsere rekursive Berechnung des Matrizenproduktes hernehmen, so sehen wir, daß wir 8 rekursive Matrizenmultiplikation mit $n/2 \times n/2$-Matrizen vornehmen müssen und 4 Paare von $n/2 \times n/2$-Matrizen addieren müssen. Letztere Matrizenaddition kosten offensichtlich n^2 viele arithmetische Operationen für ein Paar von $n \times n$-Matrizen. Damit erhalten wir die folgende Rekursionsgleichung:

$$T(n) = 8T(n/2) + n^2.$$

Die Lösung dieser Rekursionsgleichung hat allerdings ebenfalls wieder eine Größenordnung von $O(n^3)$. Also müssen wir wiederum etwas trickreicher sein. Wie bei der Multiplikation von ganzen Zahlen versuchen wir eine der rekursiven Matrizenmultiplikationen einzusparen. Dies gelingt uns indem wir zuerst die folgenden Matrizenprodukte berechnen:

$$\begin{aligned} M_1 &= (A_{12} - A_{22})(B_{21} + B_{22}) \\ M_2 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_3 &= (A_{11} - A_{21})(B_{11} + B_{12}) \\ M_4 &= (A_{11} + A_{12})B_{22} \\ M_5 &= A_{11}(B_{12} - B_{22}) \\ M_6 &= A_{22}(B_{21} - B_{11}) \\ M_7 &= (A_{21} + A_{22})B_{11}) \end{aligned}$$

Dies sieht auf den ersten Blick etwas verworren aus, allerdings kann man nun nachrechnen, daß sich mit Hilfe dieser Produkte, die Produktmatrix C berechnen läßt:

$$\begin{aligned} C_{11} &= M_1 + M_2 - M_4 + M_6 \\ C_{12} &= M_4 + M_5 \\ C_{21} &= M_6 + M_7 \\ C_{22} &= M_2 - M_3 + M_5 - M_7 \end{aligned}$$

Mit dieser Identität können wir nun leicht einen rekursive Algorithmus konstruieren. Dieser Algorithmus wurde bereits 1969 von V. Strassen konstruiert und wird daher oft als *Strassen-Algorithmus* bezeichnet.

5.4.2 Analyse von Strassen's Algorithmus (*)

Die Laufzeit von Strassen's Algorithmus gehorcht der folgenden Rekursionsgleichung:

$$T(n) = 7 \cdot T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 = 7 \cdot T\left(\frac{n}{2}\right) + \frac{9}{2}n^2.$$

Diese lösen wir ebenfalls wieder durch Iteration:

$$\begin{aligned}
T(n) &= 7 \cdot T\left(\frac{n}{2}\right) + \frac{9}{2}n^2 \\
&= 7\left(7 \cdot T\left(\frac{n}{4}\right) + \frac{9}{2}\left(\frac{n}{2}\right)^2\right) + \frac{9}{2}n^2 \\
&= 7^2 \cdot T\left(\frac{n}{4}\right) + 7 \cdot \frac{9}{2}\left(\frac{n}{2}\right)^2 + \frac{9}{2}n^2 \\
&\ \text{Nach } k \text{ Iterationen erhalten wir} \\
&= 7^k \cdot T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} 7^i \cdot \frac{9}{2}\left(\frac{n}{2^i}\right)^2 \\
&\ \text{mit } k = \log(n) \text{ erhalten wir} \\
&= 7^{\log(n)} \cdot T(1) + \sum_{i=0}^{\log(n)-1} 7^i \cdot \frac{9}{2}\left(\frac{n}{2^i}\right)^2 \\
&= 2^{\log(7)\log(n)} + \frac{9}{2}n^2 \sum_{i=0}^{\log(n)-1}\left(\frac{7}{4}\right)^i \\
&= n^{\log(7)} + \frac{9}{2}n^2 \frac{\left(\frac{7}{4}\right)^{\log(n)} - 1}{\frac{7}{4} - 1} \\
&\leq n^{\log(7)} + \frac{9}{2}n^2 \cdot \frac{4}{3} \cdot n^{\log(7)-\log(4)} \\
&= n^{\log(7)} + 6n^{\log(7)} \\
&= 7n^{\log(7)} \approx 7n^{2.81}
\end{aligned}$$

Damit erhalten wir nun das folgende Theorem:

Theorem 5.10 *Der Algorithmus von Strassen benötigt für die Matrizenmultiplikation $7n^{\log(7)}$ viele arithmetische Operationen.*

Dieses Verfahren schlägt die Schulmethode erst für $n \geq 700$. Es gibt noch eine trickreichere Identität des Matrixproduktes, die mit 7 rekursiven Multiplikationen und nur 15 Matrizenadditionen auskommt. Diese schlägt dann die Schulmethode für $n \geq 310$.

5.4.3 Verbesserung von Strassen's Algorithmus (*)

Es sieht auf den ersten Blick so aus, als wäre Strassen's Algorithmus nicht praktikabel. Dem ist allerdings *nicht* so. Mit etwas Grips erkennt man wiederum, daß es für kleine Matrizen günstig ist, die Schulmethode zu nehmen, und erst bei großen Matrizen auf Strassen umzusteigen. Für eine genauere Analyse brechen wir die Rekursion bei $2^m \times 2^m$-Matrizen ab, und steigen auf die Schulmethode um, die genau $2n^3 - n^2$ viele arithmetische Operationen benötigt.

$$\begin{aligned}
T(n) &= 7^k \cdot T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} 7^i \cdot \frac{9}{2}\left(\frac{n}{2^i}\right)^2 \\
&\quad \text{mit } k = \log(n) - m \text{ erhalten wir} \\
&= 7^{\log(n)-m} \cdot T\left(\frac{n}{2^{\log(n)-m}}\right) + \sum_{i=0}^{\log(n)-m-1} 7^i \cdot \frac{9}{2}\left(\frac{n}{2^i}\right)^2 \\
&= \frac{n^{\log(7)}}{7^m} T(2^m) + \frac{9}{2} n^2 \sum_{i=0}^{\log(n)-m-1} \left(\frac{7}{4}\right)^i \\
&= \frac{n^{\log(7)}}{7^m} T(2^m) + \frac{9}{2} n^2 \frac{\left(\frac{7}{4}\right)^{\log(n)-m} - 1}{\frac{7}{4} - 1} \\
&\leq \frac{n^{\log(7)}}{7^m} T(2^m) + \frac{9}{2} n^2 \cdot \frac{4}{3} \cdot n^{\log(7)-\log(4)} \cdot \left(\frac{4}{7}\right)^m \\
&= n^{\log(7)} \left(\frac{T(2^m)}{7^m} + 6\left(\frac{4}{7}\right)^m\right)
\end{aligned}$$

In Abbildung 5.6 ist der Graph der 'Konstanten' vor $n^{\log(7)}$ in Abhängigkeit vom Umschaltzeitpunkt 2^m dargestellt. Wie man sieht, sollte die Dimension der Matrizen für das Umschalten zwischen 8 und 16 liegen. Mit einem solchen Umschaltzeitpunkt kann man die Konstante unter 4 senken. Der Vergleich der Anzahl der arithmetischen Operationen zwischen Schulmethode und der hier vorgestellten Methode von Strassen mit Umschaltung ist in Abbildung 5.7 angegeben.

Theorem 5.11 *Der Algorithmus von Strassen für die Matrizenmultiplikation, der bei kleinen Matrizen auf die Schulmethode umsteigt, benötigt weniger als $4 \cdot n^{\log(7)}$ viele arithmetische Operationen.*

Diese Variante schlägt nun die Schulmethode für Matrizen mit Seitenlängen größer als 40, und ist für kleine Matrizen kaum schlechter. Die Differenz der Anzahlen der benötigten arithmetischen Operationen für beide Methoden für kleine Matrizen ist

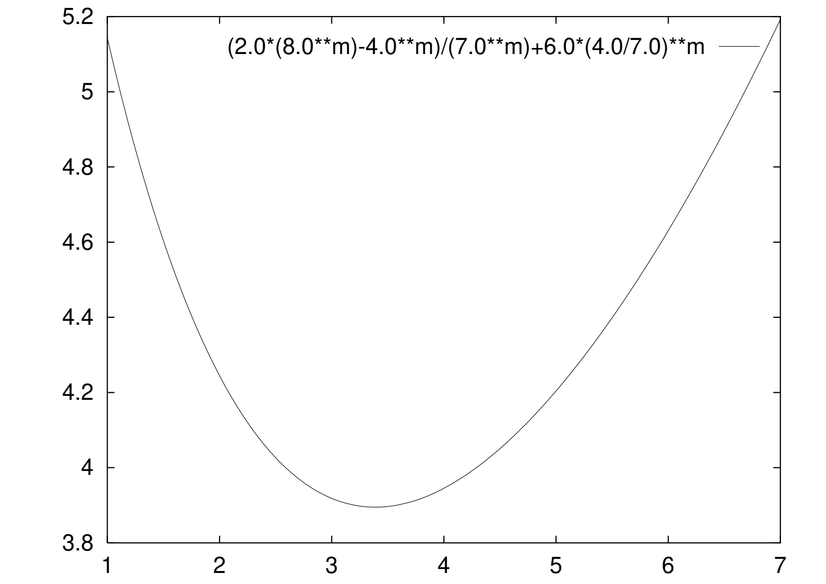

Abbildung 5.6: Bestimmung der Dimension 2^m zum Umschalten von Strassen auf die Schulmethode

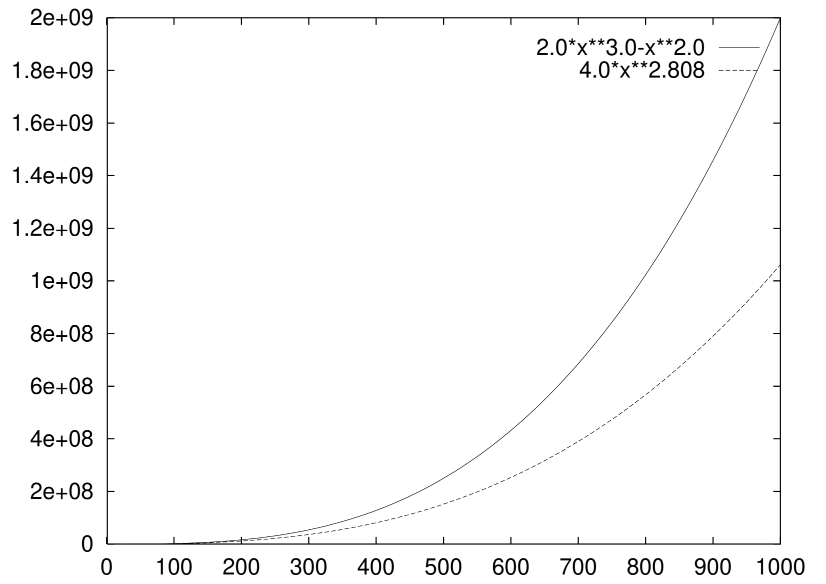

Abbildung 5.7: Vergleich der Schulmethode mit Strassen bei Umschaltung

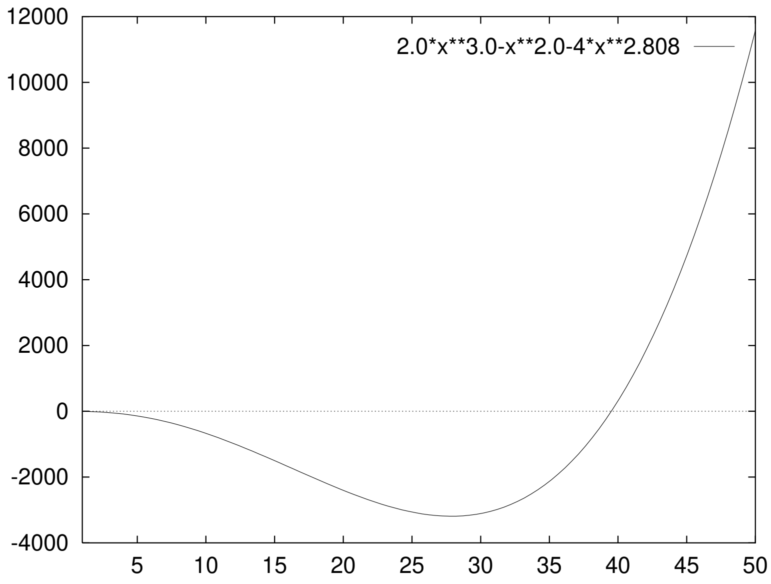

Abbildung 5.8: Zusätzlich benötigte Vergleiche der Schulmethode gegenüber Strassen bei Umschaltung bei kleinen Matrizen

in Abbildung 5.8 noch einmal graphisch dargestellt. Für eine gute Implementierung muß man sich über den Zeitpunkt des Umschaltens noch mehr Gedanken machen. Der einzige Nachteil, den man sich einkauft, ist die größere numerische Instabilität aufgrund der verwendeten Subtraktionen.

5.4.4 Weitere Entwicklungen (*)

Nun wollen wir noch eine kurze Übersicht über die weitere Entwicklung von Algorithmen zur Matrixmultiplikation angeben (siehe Abbildung 5.9). Diese sind aber fast ausschließlich von theoretischem Interesse, da die Konstanten im O so groß sind, daß diese Verfahren erst für zu große Matrizen effizient werden.

Zum Abschluß dieses Kapitels wollen wir noch einige Beziehungen zu anderen Problemen der linearen Algebra aufzeigen. Zuerst kann man zeigen, daß Matrizenmultiplikation und Matrixinversion unter gewissen realistischen Annahmen dieselbe Komplexität haben. Außerdem läßt sich die Bestimmung der Determinante auf die Matrizenmultiplikation reduzieren. Darüber hinaus sollte klar sein, daß alle angegebenen Algorithmen nicht nur für die natürlichen Zahlen gelten, sondern für jeden beliebigen Ring.

Jahr	Autoren	Laufzeit
1969	Strassen	$O(n^{2.808})$
1979	Pan	$O(n^{2.781})$
1979	Bini, Capovani, Lotti, Romani	$O(n^{2.7799})$
1979	Schönhage	$O(n^{2.548})$
1979	Pan	$O(n^{2.522})$
1982	Coppersmith, Winograd	$O(n^{2.496})$
1982	Strassen	$O(n^{2.47})$
1982	Coppersmith, Winograd	$O(n^{2.38})$

Abbildung 5.9: Historische Entwicklung der Algorithmen für die Matrizenmultiplikation

A Abbildungsverzeichnis

Bibliographie

B

B.1 Lehrbücher zu Themen der Vorlesung

A.V. Aho, J.E. Hopcroft, J.D. Ullman: The Design and Analysis of Computer Algorithms; Addison-Wesley Publishing Company: Reading (MA), 1976.

T.H. Cormen, C.E. Leiserson, R.L. Rivest: Introduction to Algorithms; The MIT Press, 1990.

M. Crochemore, W. Rytter: Text Algorithms; Oxford University Press: New York, Oxford, 1994.

D. Harel: Algorithmics: The Spirit of Computing; Addison Wesley Publishing Company: Reading (MA), 1992.

D.E. Knuth: The Art of Computer Programming Vol. 1: Fundamental Algorithms; Addison-Wesley Publishing Company, Reading (MA), 1973 (Zweite Auflage).

D.E. Knuth: The Art of Computer Programming Vol. 2: Seminumerical Algorithms; Addison-Wesley Publishing Company, Reading (MA), 1981 (Zweite Auflage).

D.E. Knuth: The Art of Computer Programming Vol. 3: Sorting and Searching; Addison-Wesley Publishing Company, Reading (MA), 1973.

K. Melhorn: Data Structures and Algorithms Vol. 1: Sorting and Searching; EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1984.

T. Ottmann, P. Widmayer: Algorithmen und Datenstrukturen; Bibliographisches Institut, Reihe Informatik, Band 70, 1990.

R. Sedgewick: Algorithms; Addison-Wesley Publishing Company, Reading (MA), 1983.

I. Wegener: Effiziente Algorithmen für grundlegende Funktionen; B.G. Teubner-Verlag, Stuttgart, 1989.

B.2 Lehrbücher zu angrenzenden Themen

R.L. Graham, D.E. Knuth O. Patashnik: Concrete Mathematics: A Foundation of Computer Science; Addison-Wesley Publishing Company, Reading (MA), 1989.

M. Hofri: Analysis of Algorithms: Computational Methods and Mathematical Tools; Oxford University Press, 1995.

J.E. Hopcroft, J.D. Ullman: Introduction to Automata Theory, Languages and Computation; Addison-Wesley Publishing Company, Reading (MA), 1979.

W.J. Paul: Komplexitätstheorie; Teubner Studienbücher Informatik, B.G. Teubner-Verlag, Stuttgart, 1978.

R. Sedgewick, P. Flajolet: An Introduction to the Analysis of Algorithms; Addison-Wesley Publishing Company, Reading (MA), 1996.

I. Wegener: Theoretische Informatik; B.G. Teubner-Verlag, Stuttgart, 1993.

B.3 Originalarbeiten

S. Carlsson: A Variant of Heapsort with Almost Optimal Number of Comparisons; *Information Processing Letters*, Vol. 24, 247–250, 1987.

S. Carlsson: Average-Case Results on Heapsort; *BIT*, Vol. 27, 2–17, 1987.

D. Dor, U. Zwick: Selecting the Median; *Proceedings of the 6th Annual ACM-SIAM Symposium on Distrete Algorithms*, 28-37, 1995.

D. Dor, U. Zwick: Median Selection Requires $(2+\varepsilon)n$ Comparisons; *Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Sciene*, 125-134, 1996.

A. Schönhage, M. Paterson, N. Pippenger: Finding the Median; *Journal of Computer and System Science*, Vol. 13, 184-199, 1976.

I. Wegener: Bottom-Up-Heapsort, a New Variant of Heapsort Beating on Average Quicksort (If n is Not Very Small); *Proceedings of the 15th International Symposium on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science, Vol. 452, 516-522, Springer-Verlag, 1990.

C-Programme

C

C.1 Vergleichsbasierte Sortieralgorithmen

In diesem Abschnitt listen wir die noch einmal auch über das World-Wide-Web verfügbare Programme zum vergleichsbasierten Sortieren auf. Der Quellcode kann auch direkt unter

`http://wwwmayr.informatik.tu-muenchen.de/lehre/1997SS/ga/uebung/`

gefunden werden.

```
/*********************************************************************
**
** This is sortcompetition.c
**
** SORTCOMPETITION
** ---------------
**
** Due to the declarations of variables,
** this file CANNOT be compiled with gcc.
** It should be compiled with the following
**
**   g++ -o sortcompetition sortcompetiton.c
**
** You can (and should) add optimization flags le -O1 -O2 before -o
** (See man page: 'man g++')
**
*********************************************************************/

#include <unistd.h>
#include <sys/times.h>
#include <stdlib.h>

/* Used for Timings */
struct tms start_time;
struct tms end_time;
```

```
/* global Variable for measuring comparisions */
int cmp=0;
/* global Variable for delaying a compare function */
int delay=0;

/* Compare-function due to the definition of C-library function qsort */
int qcompare(const void *a, const void *b)
{
  int x = *(int *)a;
  int y = *(int *)b;

  cmp++;
  for (int i=0;i<delay;i++) if(x==y) x=y;
  if (x==y) return 0;
  else if (x<y) return -1;
       else return 1;
}

/* Compare-function for seldefined sorting procedures */
int compare (int x, int y)
{
  cmp++;
  for (int i=0;i<delay;i++) if(x==y) x=y;
  if (x==y) return 0;
  else if (x<y) return -1;
       else return 1;
}

/* Often used function for swapping two variables */
void swap (int& a, int& b)
{
  int tmp=a; a=b; b=tmp;
}

/* function for copying an array a into another array b */
void copy_array (int a[], int b[], int len)
{
  for (int i=0; i<len; i++)
    b[i]=a[i];
}

/* checkiing, whether an array is sorted */
void check_array(int a[], int len)
{
  for (int i=1; i<len; i++)
    {if (a[i-1]>a[i])
       {printf("*** ARRAY IS NOT SORTED! ***\n");
        for (int i=0; i<len; i++)
          printf("%11d: %11d\n",i,a[i]);
```

```
        return;
       }
    }
}

/* outputs analysis for a sorting call */
void print_analysis(tms start_time, tms end_time)
{
  long HZ;
  HZ = sysconf(_SC_CLK_TCK);

  float sorttime = (end_time.tms_utime - start_time.tms_utime)/(float)HZ;
  printf("%11.2f seconds of CPU-time used for sorting.\n",sorttime);
  printf("%11d comparisons used for sorting.\n",cmp);
}

/*----------------------------------------------------------------------
 *
 * Insertion-Sort
 *
 *----------------------------------------------------------------------
 */
void insertionsort(int array[], int len, int (*compare)(int, int))
{
  for (int i=1; i<len; i++)
    {for (int j=0; (j<i) && (compare(array[j],array[i])<0); j++);
     int tmp=array[i];
     for(int k=i; k>j; k--) array[k]=array[k-1];
     array[j]=tmp;
    }
}

/*----------------------------------------------------------------------
 *
 * HEAP SORT
 *
 *----------------------------------------------------------------------
 */
inline void reheap(int array[], int len, int i, int (*compare)(int, int))
{
   int j;

   j=2*i+1;
   while(j<len)
     {if ((j+1<len) && (compare(array[j+1],array[j])>0)) j++;
      if (compare(array[i],array[j])<0)
        {swap(array[i],array[j]);
         i=j; j=2*j+1;
        }
      else
```

```
        {j=len;}
     }
}

void heapsort(int array[], int len, int (*compare)(int, int))
{
  int j;

  for(j=len/2; j>=0; j--)
    reheap(array,len,j,compare);

  for(j=len-1; j>0; j--)
    {swap(array[0],array[j]);
     reheap(array,j,0,compare);
    }
}

/*-----------------------------------------------------------------------
 *
 * BOTTOM-UP-HEAPSORT
 *
 *-----------------------------------------------------------------------
 */
inline void bup_reheap(int array[], int len, int i, int (*compare)(int, int))
{
   int j=i, k=2*i+1;
   while(k<len-1)
     {if (compare(array[k+1],array[k])>0) k++;
      j=k;
      k=2*k+1;
     }
   if (k==len-1) j=k;

   while(compare(array[i],array[j])>0)
     j=(j-1)/2;

   int hlp=array[j];
   array[j]=array[i];
   while(j>i)
     {j=(j-1)/2;
      swap(array[j],hlp);
     }
}

void bottom_up_heapsort(int array[], int len, int (*compare)(int, int))
{
  int j;
  int hlp;

  for(j=len/2; j>=0; j--)
```

```
    {bup_reheap(array,len,j,compare);
    }
  for(j=len-1; j>0; j--)
    {hlp=array[0]; array[0]=array[j]; array[j]=hlp;
     bup_reheap(array,j,0,compare);
    }

}
/*------------------------------------------------------------------------
 *
 * CLEVER-BOTTOM-UP-HEAPSORT
 *
 * like bottom-up-heapsort, but the insertion-routine is optimized.
 * (approx, the number of swaps is reducde by two third)
 *
 *------------------------------------------------------------------------
 */
inline void clever_bup_reheap(int array[], int len, int i,
                              int (*compare)(int, int))
{
   int j=i,k,l=i,n,power_two=-1;
   int hlp;

   while((n=2*j+1)<len-1)
     {j=n;
      if (compare(array[j+1],array[j])>0) j++;
      power_two++;
     }
   if (n==len-1) {j=n; power_two++;}

   while(compare(array[i],array[j])>0)
     {j=(j-1)/2;
      power_two--;
     }

   hlp=array[i];
   while(l!=j)
     {k=((j+1)>>power_two)-1;
      array[l]=array[k];
      l=k;
      power_two--;
     }
   array[j]=hlp;
}

void clever_bottom_up_heapsort(int array[], int len, int (*compare)(int, int))
{
  int j;
  int hlp;
```

```
  for(j=len/2; j>=0; j--)
    {clever_bup_reheap(array,len,j,compare);
    }
  for(j=len-1; j>0; j--)
    {hlp=array[0]; array[0]=array[j]; array[j]=hlp;
     clever_bup_reheap(array,j,0,compare);
    }

}

/*----------------------------------------------------------------------
 *
 * CARLSSON-HEAPSORT
 *
 * Heapsort with binary search as insertion technique
 *
 *----------------------------------------------------------------------
 */
inline void carlsson_reheap(int array[], int len, int i,
                            int (*compare)(int, int))
{
   int j=i, k=2*i+1, depth=0;
   while(k<len-1)
     {if (compare(array[k+1],array[k])>0) k++;
      j=k;
      k=2*k+1;
      depth++;
     }
   if (k==len-1) {j=k; depth++;}


   int pos, top=depth, bot=0;
   while(top>bot)
   {
     pos=(top+bot)/2;
     if (compare(array[i],array[((j+1)>>pos)-1])>=0) bot=pos+1;
     else top=pos;
   }
   j=((j+1)>>bot)-1;

   int hlp=array[j];
   array[j]=array[i];
   while(j>i)
     {j=(j-1)/2;
      swap(array[j],hlp);
     }
}

void carlsson_heapsort(int array[], int len, int (*compare)(int, int))
{
```

```
  int j;

  for(j=len/2; j>=0; j--)
    {carlsson_reheap(array,len,j,compare);
    }

  for(j=len-1; j>0; j--)
    {swap(array[0],array[j]);
     carlsson_reheap(array,j,0,compare);
    }

}

/*---------------------------------------------------------------------
 *
 * SIMPLE-QUICKSORT
 *
 * as described in the lecture
 *
 *---------------------------------------------------------------------
 */
void simple_qsort(int array[], int l, int r,int (*compare)(int, int))
{

  int i, j, p;

  if (r>l)
       {i=l-1;
        j=r;
        p=r;
        for (;;)
          {do i++; while ((i<j) && (compare(array[i],array[p])<0));
           do j--; while ((j>i) && (compare(array[j],array[p])>0));

           if (i>=j) break;

           swap(array[i],array[j]);
          }

          swap(array[i],array[p]);

          if (i-l<r-i)
            {simple_qsort(array,l,i-1,compare);
             simple_qsort(array,i+1,r,compare);
            }
          else
            {simple_qsort(array,i+1,r,compare);
             simple_qsort(array,l,i-1,compare);
            }
```

```
      }
}

void simple_quicksort(int array[], int len, int (*compare)(int, int))
{
  simple_qsort(array,0,len-1,compare);
}

/*-----------------------------------------------------------------------
 *
 * STD-QUICKSORT
 *
 * as described in many books
 *
 *-----------------------------------------------------------------------
 */
void std_qsort(int array[], int l, int r, int (*compare)(int, int))
{

  int i, j, p;

  if (r>l)
    {i=l-1;
     j=r;
     p=r;

     for (;;)
       {do i++; while (compare(array[i],array[r])<0);
        do j--; while (compare(array[j],array[r])>0);

        if (i>=j) break;

        swap(array[i],array[j]);
       }

       swap(array[i],array[r]);

       if (i-l<r-i)
         {std_qsort(array,l,i-1,compare);
          std_qsort(array,i+1,r,compare);
         }
       else
         {std_qsort(array,i+1,r,compare);
          std_qsort(array,l,i-1,compare);
         }

    }
}

void std_quicksort(int array[], int len, int (*compare)(int, int))
```

```
{
  for (int i=len-1; i>0; i--)
    if (compare(array[i-1],array[i])>0) swap(array[i-1],array[i]);
  std_qsort(array,1,len-1,compare);
}

/*--------------------------------------------------------------------
 *
 * STD-QUICKSORT MIT INSSORT
 *
 * usual implementation (given in mongraphs)
 * together with insertionsort
 *
 *--------------------------------------------------------------------
 */
void std_ins_qsort(int array[], int l, int r, int (*compare)(int, int))
{

  int i, j, p;

  if (r>l)
    {if (r - l < 10) insertionsort(&(array[l]),r-l+1,compare);
     else
       {i=l-1;
        j=r;
        p=r;

        for (;;)
          {do i++; while (compare(array[i],array[r])<0);
           do j--; while (compare(array[j],array[r])>0);

           if (i>=j) break;

           swap(array[i],array[j]);
          }

          swap(array[i],array[r]);

          if (i-l<r-i)
            {std_ins_qsort(array,l,i-1,compare);
             std_ins_qsort(array,i+1,r,compare);
            }
          else
            {std_ins_qsort(array,i+1,r,compare);
             std_ins_qsort(array,l,i-1,compare);
            }
       }
    }
}
```

```
void std_ins_quicksort(int array[], int len, int (*compare)(int, int))
{
  for (int i=len-1; i>0; i--)
    if (compare(array[i-1],array[i])>0) swap(array[i-1],array[i]);
  std_ins_qsort(array,1,len-1,compare);
}

/*-----------------------------------------------------------------------
 *
 * QUICKSORT
 *
 * usual implementation (given in mongraphies)
 * together with insertionsort
 * the Pivot is chosen at random
 *
 *-----------------------------------------------------------------------
 */
void q_sort(int array[], int l, int r, int (*compare)(int, int))
{

  int i, j, p;

  if (r>l)
    {if (r - l < 10) insertionsort(&(array[l]),r-l+1,compare);
     else
       {i=l-1;
        j=r;
        int ra;
        p=l+((ra=lrand48())%(r-l+1));
        swap(array[p],array[r]);

        for (;;)
          {do i++; while (compare(array[i],array[r])<0);
           do j--; while (compare(array[j],array[r])>0);

           if (i>=j) break;

           swap(array[i],array[j]);
          }

          swap(array[i],array[r]);

          if (i-l<r-i)
            {q_sort(array,l,i-1,compare);
             q_sort(array,i+1,r,compare);
            }
          else
            {q_sort(array,i+1,r,compare);
             q_sort(array,l,i-1,compare);
            }
```

```
      }
    }
}

void quicksort(int array[], int len, int (*compare)(int, int))
{
  for (int i=len-1; i>0; i--)
    if (compare(array[i-1],array[i])>0) swap(array[i-1],array[i]);
  q_sort(array,1,len-1,compare);
}

/*-----------------------------------------------------------------------
 *
 * CLEVER-QUICKSORT
 *
 * usual implementation (given in mongraphies)
 * together with insertionsort
 * the Pivot is chosen as the median of three randomly chosen elements
 *
 *-----------------------------------------------------------------------
 */
void clever_qsort(int array[], int l, int r, int (*compare)(int, int))
{

  int i, j, p, p1, p2, p3;

  if (r>l)
    {if (r - l < 10) insertionsort(&(array[l]),r-l+1,compare);
     else
       {i=l-1;
        j=r;

        /* Pivot zufaellig waehlen */
        p1=l+((lrand48())%(r-l+1));
        p2=l+((lrand48())%(r-l+1));
        p3=l+((lrand48())%(r-l+1));
        if (compare(array[p1],array[p2])<0)
          if (compare(array[p2],array[p3])<0)
            {p=p2;
            }
          else
            {if (compare(array[p1],array[p3])<0)
               p=p3;
             else
               p=p1;
            }
        else
          if (compare(array[p2],array[p3])>0)
            {p=p2;
```

```
            }
          else
            {if (compare(array[p1],array[p3])>0)
               p=p3;
             else
               p=p1;
            }

        swap(array[p],array[r]);

        for (;;)
          {do i++; while (compare(array[i],array[r])<0);
           do j--; while (compare(array[j],array[r])>0);

           if (i>=j) break;

           swap(array[i],array[j]);
          }

          swap(array[i],array[r]);

          if (i-1-l<r-i+1)
            {clever_qsort(array,l,i-1,compare);
             clever_qsort(array,i+1,r,compare);
            }
          else
            {clever_qsort(array,i+1,r,compare);
             clever_qsort(array,l,i-1,compare);
            }

       }
    }
}

void clever_quicksort(int array[], int len, int (*compare)(int, int))
{
  for (int i=len-1;i>0;i--)
    if (compare(array[i-1],array[i])>0) swap(array[i-1],array[i]);
  clever_qsort(array,1,len-1,compare);
}

/*-----------------------------------------------------------------------
 *
 * MERGE-SORT
 *
 *-----------------------------------------------------------------------
 */
void mergesort(int a[], int len, int (*compare)(int, int))
{
```

```
   int width;
   int i,k,j1,j2;
   int j1max, j2max;

   int b[len];
   int *c;

   int *res;
    res=a;

  for(width=1; width<=len; width*=2)
    {for(i=0; i<len; i+=(2*width))
       {j1max=(i+width>=len)?len-i:width;
        j2max=(i+2*width>=len)?len-i-width:width;
        k=i;
        j1=j2=0;

        while((j1<j1max)&&(j2<j2max))
          {if (compare(a[i+j1],a[i+width+j2])<0)
             { b[k]=a[i+j1]; k++; j1++; }
           else
             { b[k]=a[i+width+j2]; k++; j2++; }
          }

        while (j1<j1max) { b[k]=a[i+j1]; k++; j1++; }
        while (j2<j2max) { b[k]=a[i+width+j2]; k++; j2++; }
       }
     c=b; b=a; a=c;
    }

  if (a!=res) copy_array(a,res,len);
}

/****************************************************************************/

main(int argc, char *argv[])
{
  if (argc != 4)
  {
    printf("\nUsage: ./sortcompetition <len> <type> <duration>\n");
    printf("  <len> Number of Elements to be sorted\n");
    printf("  <Art> Type of Sequnce:\n");
    printf("    0: Random (mittels rand())\n");
    printf("    1: Random (mittels lrand48())\n");
    printf("    2: Increasing\n");
    printf("    3: Decreasing\n");
    printf("    4: Bitonic\n");
    printf("  <duration> Duration of a comparison:\n");
    printf("    0: normal comparison\n");
    printf("    i: retarded comparison:");
```

```
    printf("i comparisons are executed additionally\n");
    exit(1);
  }

  int i,len=atoi(argv[1]);
  int array[len];
  int a[len];
  int type=atol(argv[2]);
  delay=atol(argv[3]);


  switch(type)
  {  case 0:
          printf("\nRandom Sequence (rand):");
          printf("\n-----------------------\n");
          srand(12345);
          for (i=0; i<len; i++) array[i]=rand()*32768+rand();
          break;
     case 1:
          printf("\nRandom Sequence (rand48):");
          printf("\n-------------------------\n");
          srand(12345);
          for (i=0; i<len; i++)
            array[i]=lrand48();
          break;
     case 2:
          printf("\nIncreasing Sequence:");
          printf("\n--------------------\n");
          for (i=0; i<len; i++) array[i]=i;
          break;
     case 3:
          printf("\nDecreasing Sequence:");
          printf("\n--------------------\n");
          for (i=0; i<len; i++) array[i]=len-i-1;
          break;
     case 4:
          printf("\nBitonic Sequence:");
          printf("\n-----------------\n");
          for (i=0;i<(len+1)/2;i++) array[i] = array[len-i-1] = i;
          break;
  }

  if (len>10000) printf("\nINSERTIONSORT omitted ...\n");
  else
  {
  printf("\nINSERTIONSORT:\n");
  copy_array(array,a,len);cmp=0;
  times(&start_time);
  insertionsort(a,len,compare);
  times(&end_time);
```

```
check_array(a,len);
print_analysis(start_time,end_time);
}

if (len>400000) printf("\nMERGESORT omitted ...\n");
else
{
printf("\nMERGESORT:\n");
copy_array(array,a,len);cmp=0;
times(&start_time);
mergesort(a,len,compare);
times(&end_time);
check_array(a,len);
print_analysis(start_time,end_time);
}

printf("\nHEAPSORT:\n");
copy_array(array,a,len);cmp=0;
times(&start_time);
heapsort(a,len,compare);
times(&end_time);
check_array(a,len);
print_analysis(start_time,end_time);


printf("\nCARLSSON-HEAPSORT:\n");
copy_array(array,a,len);cmp=0;
times(&start_time);
carlsson_heapsort(a,len,compare);
times(&end_time);
check_array(a,len);
print_analysis(start_time,end_time);


printf("\nBOTTOM-UP-HEAPSORT:\n");
copy_array(array,a,len);cmp=0;
times(&start_time);
bottom_up_heapsort(a,len,compare);
times(&end_time);
check_array(a,len);
print_analysis(start_time,end_time);


printf("\nCLEVER-BOTTOM-UP-HEAPSORT:\n");
copy_array(array,a,len);cmp=0;
times(&start_time);
clever_bottom_up_heapsort(a,len,compare);
times(&end_time);
check_array(a,len);
print_analysis(start_time,end_time);
```

```
if ((type==4) && (len>10000)) printf("\nUNIX-QUICKSORT omitted ...\n");
else
{
printf("\nUNIX-QUICKSORT:\n");
copy_array(array,a,len);cmp=0;
times(&start_time);
qsort(a, len, sizeof(a[0]), qcompare);
times(&end_time);
check_array(a,len);
print_analysis(start_time,end_time);
}

if ((type>1)&&(len>10000)) printf("\nSIMPLE-QUICKSORT omitted ...\n");
else
{
printf("\nSIMPLE-QUICKSORT:\n");
copy_array(array,a,len);cmp=0;
times(&start_time);
simple_quicksort(a,len,compare);
times(&end_time);
check_array(a,len);
print_analysis(start_time,end_time);
}

if ((type>1)&&(len>10000)) printf("\nSTD-QUICKSORT omitted ...\n");
else
{
printf("\nSTD-QUICKSORT:\n");
copy_array(array,a,len);cmp=0;
times(&start_time);
std_quicksort(a,len,compare);
times(&end_time);
check_array(a,len);
print_analysis(start_time,end_time);
}

if ((type>1)&&(len>10000)) printf("\nSTD-QUICKSORT (INSSORT) omitted ...\n");
else
{
printf("\nSTD-QUICKSORT (INSSORT):\n");
copy_array(array,a,len);cmp=0;
times(&start_time);
std_ins_quicksort(a,len,compare);
times(&end_time);
check_array(a,len);
print_analysis(start_time,end_time);
}
```

```
  printf("\nQUICKSORT (INSSORT; RAND PIVOT):\n");
  copy_array(array,a,len);cmp=0;
  times(&start_time);
  quicksort(a,len,compare);
  times(&end_time);
  check_array(a,len);
  print_analysis(start_time,end_time);


  printf("\nCLEVER-QUICKSORT (BEST OF THREE; INSSORT; RAND PIVOT):\n");
  copy_array(array,a,len);cmp=0;
  times(&start_time);
  clever_quicksort(a,len,compare);
  times(&end_time);
  check_array(a,len);
  print_analysis(start_time,end_time);

}
```

C.2 Bucketsort

In diesem Abschnitt listen wir die noch einmal auch über das World-Wide-Web verfügbare Programme zum Bucketsort auf. Der Quellcode kann auch direkt unter

`http://wwwmayr.informatik.tu-muenchen.de/lehre/1997SS/ga/uebung/`

gefunden werden.

```
/********************************************************************
**
** This is bucket.c
**
** A simple implementation of Bucketsort
**
** It should be compiled with the following
**
**   g++ -o bucket bucket.c
**
** You can (and should) add optimization flags le -O1 -O2 before -o
** (See man page: 'man g++')
**
********************************************************************/
#include <stdlib.h>

template <class K>
```

```
class list_item
{
  friend    class list<K>;

  K         key;
  list_item *next;
};

template <class K>
class list
{
protected:
  list_item<K> *first, *last;

public:
  void init() {first=last=NULL;}
  bool is_empty() {return (first==NULL);}
  void append(K key)
    {
      list_item<K> *ptr = new list_item<K>;
      ptr->key=key;
      ptr->next=NULL;
      if (first==NULL) first=last=ptr;
      else last=last->next=ptr;
    }
  K remove()
    {
      list_item<K> *ptr;
      ptr=first;
      K key=first->key;
      first=first->next;
      delete ptr;
      return key;
    }
};

//----------------------------------------------------------------------


bucketsort(int array[], int len, int N)
{
  int i,j;
  list<int> bucket[N];

  for (i=0; i<N; i++) bucket[i].init();
  for (i=0; i<len; i++) bucket[array[i]].append(array[i]);
  for (i=0, j=0; i<N; i++)
    while (!(bucket[i].is_empty()))
      array[j++]=bucket[i].remove();
}
```

```
gen_bucketsort(char *array[], int len, int N, int c)
{
  int i,j,k;
  list<char*> bucket[N];

  for (i=0; i<N; i++) bucket[i].init();
  for (k=c-1; k>=0; k--)
    {
      for (i=0; i<len; i++) bucket[array[i][k]].append(array[i]);
      for (i=0, j=0; i<N; i++)
        while (!(bucket[i].is_empty()))
          array[j++]=bucket[i].remove();
    }
}


main()
{
  int len=100;

  int array[len];
  char *array2[len];

  int N=1000;
  srand48(12345);
  printf("Bucketsort\n");
  for (int i=0; i<len; i++) array[i]=lrand48()%N;
  bucketsort(array, len, N);
  for (i=0; i<len; i++) printf("%4d : %5d\n",i,array[i]);


  N=90;
  int C=20;
  printf("verallgemeinerter Bucketsort\n");
  for (i=0; i<len; i++)
    { char *p = new char[C];
      array2[i]=p;
      for(int j=0;j<C-1;j++)
        *(p++)=64+lrand48()%(N-64);
      *p=0;
    }
  gen_bucketsort(array2, len, N, C);
  for (i=0; i<len; i++) printf("%4d : %s\n",i,array2[i]);
}
```

C.3 Selektion

In diesem Abschnitt listen wir die noch einmal auch über das World-Wide-Web verfügbare Programme zum Selektieren auf. Der Quellcode kann auch direkt unter

`http://wwwmayr.informatik.tu-muenchen.de/lehre/1997SS/ga/uebung/`

gefunden werden.

```
/*********************************************************************
**
** This is select.c
**
** A simple implementation of QuickSelect
** and the so-caled BFPRT-Algorithm
**
** Due to the declarations of variables,
** this file CANNOT be compiled with gcc.
** It should be compiled with the following
**
**   g++ -o select select.c
**
** You can (and should) add optimization flags le -O1 -O2 before -o
** (See man page: 'man g++')
**
*********************************************************************/
#include <stdlib.h>
#include <unistd.h>
#include <sys/times.h>


/* Used for Timings */
struct tms start_time;
struct tms end_time;



int cmp=0;
int compare (int x, int y)
{
  cmp++;
  if (x==y) return 0;
  else if (x<y) return -1;
       else return 1;
}
```

```
/* outputs analysis for a sorting call */
void print_analysis(tms start_time, tms end_time)
{
  long HZ;
  HZ = sysconf(_SC_CLK_TCK);

  float sorttime = (end_time.tms_utime - start_time.tms_utime)/(float)HZ;
  printf("%11.2f seconds of CPU-time used for sorting.\n",sorttime);
  printf("%11d comparisons used for sorting.\n",cmp);
}

void copy_array (int a[], int b[], int len)
{
  for (int i=0; i<len; i++)
    b[i]=a[i];
}


void swap (int& a, int& b)
{
  int tmp=a; a=b; b=tmp;
}

/*
 * HEAPSORT (for sorting small arrays)
 *
 */

inline void reheap(int array[], int len, int i)
{
   int j;

   j=2*i+1;
   while(j<len)
     {if ((j+1<len) && (compare(array[j+1],array[j])>0)) j++;
      if (compare(array[i],array[j])<0)
        {swap(array[i],array[j]);
         i=j; j=2*j+1;
        }
      else
        {j=len;}
     }
}


void heapsort(int array[], int len)
{
  int j;

  for(j=len/2; j>=0; j--)
```

```
    reheap(array,len,j);

  for(j=len-1; j>0; j--)
    {swap(array[0],array[j]);
     reheap(array,j,0);
    }
}

/*
 * Procedure partition from Quicksort
 *
 */

int partition(int array[], int l, int r, int pivot)
{
  int i=l-1, j=r;
  swap(array[pivot], array[r]);
  pivot=r;
  for (;;)
    {
      do i++; while ((i<j) && (compare(array[i],array[pivot])<0));
      do j--; while ((j>i) && (compare(array[j],array[pivot])>0));
      if (i>=j) break;
      swap(array[i], array[j]);
    }
  swap(array[i], array[r]);
  return i;
}


/*
 * QuickSelect
 *
 */
quick_select(int array[], int len, int k)
{
  int pivot=len-1;
  pivot=partition(array, 0, len-1, pivot);
  int rank=pivot+1;
  if (k==rank) return pivot;
  if (k<rank)  return quick_select(array, rank-1, k);
  else return pivot+1+quick_select(&(array[pivot+1]), len-rank, k-rank);
}

/*
 * The so-called BRPRT-Algorithm
 *
 */
```

```
/*
 * Determining median of five Elements,
 * as presented in the tutorial by the students
 *
 */
median_of_five(int f[], int max)
{
  if (compare(f[0],f[1])>0) swap(f[0],f[1]);
  // ersten beiden sortiert
  if (compare(f[1],f[2])>0)
    {
      swap(f[1],f[2]);
      if (compare(f[0],f[1])>0) swap(f[0],f[1]);
    }
  // ersten drei sortiert
  if (compare(f[1],f[3])>0)
    if (compare(f[0],f[3])>0)
      {int tmp=f[3]; f[3]=f[2]; f[2]=f[1]; f[1]=f[0]; f[0]=tmp;}
    else
      {int tmp=f[3]; f[3]=f[2]; f[2]=f[1]; f[1]=tmp;}
  else
    if (compare(f[2],f[3])>0)
      swap(f[3],f[2]);
  // die ersten vier sortiert.
  if (max==5)
    if (compare(f[4],f[2])<0)
      if (compare(f[4],f[1])<0)
        {int tmp=f[4]; f[4]=f[3]; f[3]=f[2]; f[2]=f[1]; f[1]=tmp;}
      else
        swap(f[2],f[4]);
  //ready

}

/*
 * determi the medians of n over 5 groups of 5 elements each
 *
 */
determine_medians(int array[], int len)
{
  int five[5], i, j, max=0;
  int cols=(len-1)/5+1;

  for(i=0; i<cols; i++)
    {
      for (j=0; i+j*cols<len; j++)
        five[j]=array[i+j*cols];
      max=j;
      median_of_five(five, max);
      for (j=0; j<max; j++)
```

```
        array[i+j*cols]=five[j];
    }
}


/*
 * The heart of BFPRT
 *
 */
Select(int array[], int len, int k)
{
 if (len<=64)
 {
   heapsort(array, len);
   return k-1;
 }

 determine_medians(array, len);
 int width=(len-1)/5+1;
 int median=2*width+Select(&(array[2*width]), width, width/2+1);
 median=partition(array, 0, len-1, median);
 int rank=1+median;
 if (k==rank) return median;
 if (k<rank) return Select(array, rank-1, k);
 else return median+1+Select(&(array[median+1]), len-rank, k-rank);

}

//------------------------------------------------------------------------


main(int *argc, char** argv)
{
  int len=atoi(argv[1]);
  int array[len], a[len];
  int median, selmed;

  srand48(12345);
  for (int i=0; i<len; i++) array[i]=lrand48()%50;
  copy_array(array,a,len);
  printf("Determining Median ...\n");
  heapsort(a,len);
  median=a[len/2];

  cmp=0;
  copy_array(array,a,len);
  printf("QuickSelect ...\n");
  times(&start_time);
  int pos=quick_select(a,len,len/2+1);
  times(&end_time);
```

```
  int qs_med=a[pos];
  if (median!=qs_med)
    {
      printf("QuickSelect has determined WRONG median! ");
      printf("(Median) %d!=%d (QS) at pos %d\n",median,qs_med,pos);
      for(i=0;i<len;i++)
        printf("%3d: %4d\n",i,a[i]);
    }
  else
    print_analysis(start_time,end_time);

    cmp=0;
  copy_array(array,a,len);
  printf("BFPRT ...\n");
  times(&start_time);
  pos=Select(a,len,len/2+1);
  times(&end_time);
  int bfprt_med=a[pos];
  if (median!=bfprt_med)
    {
      printf("BFPRT has determined WRONG median!\n");
      printf("(Median) %d!=%d (BFPRT) at pos %d\n",median,bfprt_med,pos);
      for(i=0;i<len;i++)
        printf("%3d: %4d\n",i,a[i]);
    }
  else
    print_analysis(start_time,end_time);


}
```

G Gofer-Skripten

G.1 Berechnung von Fibonacci Zahlen

In diesem Anhang geben wir die vollständige `gofer`-Definitionen der drei Varianten zur Berechnung der n-ten Fibonacci-Zahl an. da für große n schnell der maximal darstellbare Integer von `gofer` überschritten wird, wird hierfür noch eine rudimentäre definition der Klasse Long angegeben. Element der Klasse Long sind Paare vom Typ `(Float,Int)`, wobei das erste Argument die Mantisse und das zweite Argument den Exponenten einer Zahl darstellt. Der Einfachheit halber sind nur die verwendeten Operation `+` und `*` auf dieser Klasse definiert.

```
-------------------------------------------------------------------------
--
-- rudiment"are Klasse Long
--
-------------------------------------------------------------------------

type Long = (Float,Int)

longAdd :: Long -> Long -> Long
longAdd (a,b) (c,d) = if b==d then if a+c<10.0 then (a+c,b)
                                                else ((a+c)/10.0,b+1)
                              else if b>d then longAdd (a,b) (c/10.0,d+1)
                                          else longAdd (a/10.0,b+1) (c,d)
longMult :: Long -> Long -> Long
longMult (a,b) (c,d) = if a*c<10.0 then (a*c,b+d)
                                   else ((a*c)/10.0,b+d+1)

instance Num Long where
        (+) = longAdd
        (*) = longMult


-------------------------------------------------------------------------
--
-- Fibanoacci rekursiv
--
```

```
----------------------------------------------------------------------

fib1 :: Int -> Long
fib1 n | n==1      = (1.0,0)
       | n==2      = (1.0,0)
       | otherwise = (fib1 (n-1)) + (fib1 (n-2))


----------------------------------------------------------------------
--
-- Fibanoacci iterativ
--
----------------------------------------------------------------------

g :: Int -> (Long,Long)
g n | n==1      = ((1.0,0),undefined)
    | n==2      = ((1.0,0),(1.0,0))
    | otherwise = let (x,y)=g (n-1)
                  in (x+y,x)

fib2 :: Int -> Long
fib2 n = x where (x,y) = g n


----------------------------------------------------------------------
--
-- Fibanocci Matrix
--
----------------------------------------------------------------------

h :: Int -> (Long,Long,Long,Long)
h n | n==1   = ((1.0,0), (1.0,0),
                (1.0,0), (0.0,0))
    | even n = let (a,b,c,d)=h(n/2)
               in ((a*a)+(b*c), (a*b)+(b*d),
                   (c*a)+(d*c), (c*b)+(d*d))
    | odd  n = let (a,b,c,d)=h((n-1)/2)
               in ((a*a)+(b*c)+(a*b)+(b*d), (a*a)+(b*c),
                   (c*a)+(d*c)+(c*b)+(d*d), (c*a)+(d*c))

fib3 :: Int -> Long
fib3 1 = 1
fib3 2 = 1
fib3 n = a+c where (a,b,c,d) = h (n-2)


----------------------------------------------------------------------
--
-- Fibanocci Matrix (optimiert)
--
----------------------------------------------------------------------
```

```
h2 :: Int -> (Long,Long,Long,Long)
h2 n | n==1   = ((1.0,0), (1.0,0),
                 (1.0,0), (0.0,0))
     | even n = let (a,b,c,d)=h2 (n/2)
                    x=(a*b)+(b*d)
                    y=b*b
                in ((a*a)+y, x,
                    x,       y+(d*d))
     | odd  n = let (a,b,c,d)=h2 ((n-1)/2)
                    x=(a*a)+(b*b)
                    y=(a*b)+(b*d)
                in (x+y, x,
                    x  , y)

fib4 :: Int -> Long
fib4 1 = 1
fib4 2 = 1
fib4 n = a+c where (a,b,c,d) = h2 (n-2)

--------------------------------------------------------------------------
```

Index