Der Keller – ein fundamentaler Baustein der Informatik

Martin Mundhenk

Friedrich-Schiller-Universität Jena
Fakultät für Mathematik und Informatik
Ernst-Abbe-Platz 2
07743 Jena
martin.mundhenk@uni-jena.de

Kurzfassung: Die Datenstruktur Keller begegnet Informatik-Studierenden gleich am Beginn des Studiums als eine fundamentale informatische Denkweise. Wir machen einen Ausflug zu grundlegenden Algorithmen und sehen, wie ein Keller beim Auswerten von Formeln oder beim schnellen Potenzieren hilft. Neben einzelnen Algorithmen gibt es auch algorithmische Prinzipien, die auf dem Keller basieren. Das wird am Beispiel des Teile-und-herrsche-Prinzips verdeutlicht.

1 Der Keller

Ein *Keller* ist ein Speicher für beliebig viele Elemente. Wenn man ein neues Element speichern will, dann legt man es oben in den Keller. Wie bei einem Kisten-Stapel kann man nur sehen, was in der obersten Kiste drin ist. D.h. wenn man in den Keller schaut, sieht man nur das zuletzt gespeicherte Element. Dieses sichtbare Element ist auch das, was man aus dem Keller entfernen kann. Wurde es entfernt, ist das zuvor darunter liegende Element sichtbar. Die Abbildung zeigt die Wirkungen einer Folge von Keller-Operationen auf einem anfangs leeren Keller (a). Wird 5 in den Keller eingefügt, ergibt sich Situation (b). Fall (c) zeigt den Keller nach Einfügen der 13. Wird nun das oberste Element gelesen, ergibt das 13. Durch Einfügen der 4 ergibt sich Situation (d). Fall (e) zeigt den Keller, nachdem das oberste Element gelöscht wurde.

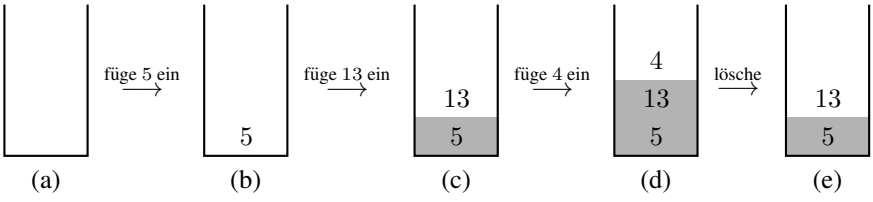

Abbildung 1: Ein Keller und seine Veränderung

Einen Keller benutzt man mit folgenden Keller-Operationen:

- füge x ein (d.h. lege x oben auf den Keller),

- lösche (d.h. entferne das oberste Element des Kellers),
- lies (das oberste Element des Kellers),
- stelle fest, ob der Keller leer ist.

Die Arbeitsweise eines Kellers wird als LIFO (*last in, first out*) bezeichnet.

2 Das Auswerten arithmetischer Ausdrücke

Arithmetische Ausdrücke kennt man bereits aus der Schule als Rechenaufgaben wie z.B.

$$(3 + 4) \cdot 5 - (8 - 2 \cdot 3 + 7)\,.$$

Wenn man sie ausrechnen will, muss man die Klammerung und die unterschiedliche Bindungsstärke der arithmetischen Operatoren – „Punktrechnung geht vor Strichrechnung“ – beachten. Es gibt verschiedene Schreibweisen für arithmetische Ausdrücke. Eine Schreibweise, bei der man sich weder um Klammerung noch um Bindungsstärke der Operatoren kümmern muss, ist die *Präfixnotation*[1]. Dabei schreibt man z.B. für $3 + 4$ den Operator $+$ wie ein Funktionssymbol vor die beiden Operanden 3 und 4, also $+(3, 4)$. Für

$$(3 + 2 \times 4 - 1) \times (7 - 5)$$

schreibt man durch Vorziehen der Operatoren zunächst

$$\times(-(+(3, \times(2, 4)), 1), -(7, 5))\,.$$

Da wir hier nur zweistellige Operatoren verwenden (das „$-$“ als negatives Vorzeichen lassen wir mal weg), kann man Klammern und Kommas weglassen, ohne die Eindeutigkeit des Ausdrucks zu verlieren. In diesem Beispiel erhalten wir dann

$$\times\ -\ +\ 3\ \ \times\ 2\ 4\ 1\ -\ 7\ 5\,.$$

Diese Schreibweise nennt man *Präfixnotation*. Die übliche Schreibweise heißt auch *Infixnotation*, da der Operator stets *zwischen* den Operanden steht. Bei der Infixnotation kann man nicht auf die Klammern verzichten. Das macht das Erkennen der Struktur des Ausdrucks und damit auch seine Auswertung schwieriger.

2.1 Auswertung von Ausdrücken in Präfixnotation

Zum Auswerten eines Ausdrucks in Präfixnotation liest man ihn von rechts nach links! Liest man eine Zahl, dann legt man sie auf den Keller. Da jeder Operator auf die beiden Operanden folgt, wird ein Operator auf die beiden obersten Elemente des Kellers angewendet. Die Operanden waren sozusagen Zwischenergebnisse, die jetzt nicht mehr benötigt werden und aus dem Keller entfernt werden. Das Ergebnis dieses Rechenschrit-

[1] Sie wird auch als „Polnische Notation“ bezeichnet, um an den Logiker Jan Łukasiewicz zu erinnern, der sie um 1920 erfunden hat.

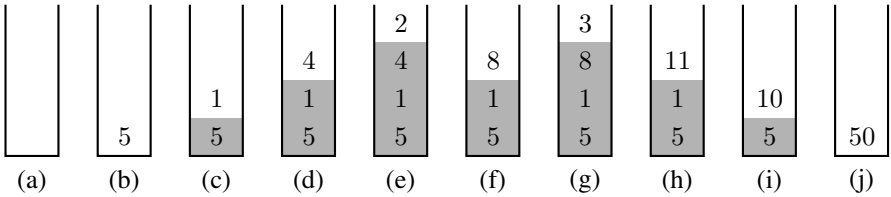

Abbildung 2: Auswertung des Ausdrucks in Präfixnotation $\times - + 3 \times 2\ 4\ 1\ 5$

tes wird auf den Keller gelegt, um so Zwischenergebnisse für weitere Rechenschritte im Keller zu sichern.

Abbildung 2 zeigt in groben Schritten die Auswertung des Ausdrucks $(3+2\times4-1)\times5$ mit der Präfixnotation $\times - + 3 \times 2\ 4\ 1\ 5$. Die Auswertung startet mit dem leeren Keller (a). Anschließend werden von rechts nach links die Zahlen $5, 1, 4, 2$ gelesen und nacheinander auf den Keller gelegt (b)–(e). Im Schritt von (e) zu (f) wird das Multiplikationszeichen $\times$ gelesen. Um es auszuwerten, wird zunächst ein Element vom Keller genommen. Man merkt sich dieses Element als linken Operanden – hier ist es die 2. Anschließend wird noch ein Element vom Keller genommen, das man sich als rechten Operanden merkt – hier ist es die 4. Nun hat man den auszuwertenden Ausdruck 2×4 zusammengestellt. Er wird ausgewertet und das Ergebnis wird im Keller gespeichert – hier ist es die 8. Danach wird die 3 gelesen und im Keller gespeichert (g). Es folgt die Auswertung des $+$ (h), des $-$ (i) und schließlich des $\times$ (j). Am Ende ist der Wert des Ausdrucks das einzige im Keller gespeicherte Element.

2.2 Auswertung üblicher arithmetischer Ausdrücke

Beim Auswerten eines Ausdrucks in Präfixnotation haben wir einen Keller zum Speichern von Zahlen benutzt. Einen solchen Keller werden wir auch zum Auswerten üblicher Ausdrücke benutzen, und wir nennen ihn hier *Zahlenkeller*. Die Operatoren brauchte man nicht zu speichern, da man sie sofort auswerten konnte. Das geht nicht mehr, wenn man einen üblichen arithmetischen Ausdruck auswerten will und sie von links nach rechts liest[2]. Hat man zum Beispiel den Anfang $142+20\cdots$ eines Ausdrucks gelesen, dann weiß man nach dem Lesen der 20 noch nicht, welche Werte von dem gerade gelesenen $+$ addiert werden. Kommt hinter der 20 ein $\times$, dann beginnt mit 20 ein Teilausdruck, dessen Wert berechnet werden muss, bevor er zur 142 addiert wird. Kommt hinter der 20 ein $+$, dann erhält man mit der Auswertung von $142 + 20$ das linke Argument für diese spätere Addition. Im ersten Fall wird das erste $+$ nach dem darauffolgenden Operator ausgewertet, im zweiten Fall davor. Um die Operatoren in der richtigen Reihenfolge auszuwerten, benutzt man einen zweiten Keller – wir nennen ihn hier *Operatorenkeller*.

Zuerst definieren wir als grundlegende Operation das *Auswerten eines Operators*. Es besteht aus folgenden Schritten.

[2]Die Leserichtung spielt hier keine Rolle.

1. Entferne ein Element aus dem Zahlenkeller und nenne es b.
2. Entferne ein Element aus dem Operatorenkeller und nenne es $\otimes$.
3. Entferne ein Element aus dem Zahlenkeller und nenne es a.
4. Berechne $a \otimes b$ und lege es auf den Zahlenkeller.

Die folgende Abbildung zeigt in einem Beispiel, wie sich die beiden Keller bei einer Operatorauswertung verändern. Nach der Operatorauswertung sind also beide Keller eins kleiner geworden.

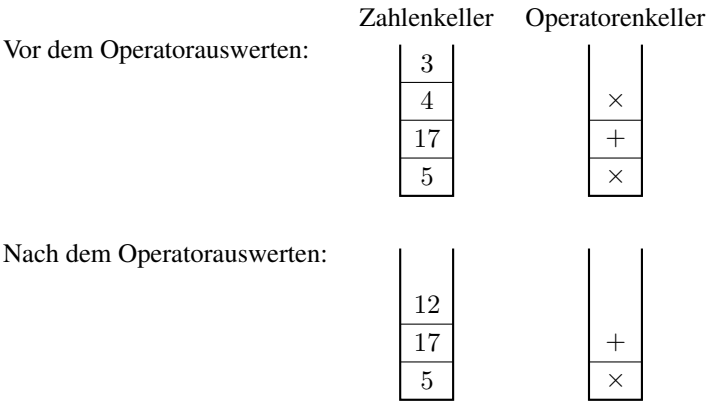

Abbildung 3: Auswertung eines Operators

Zur Auswertung des gesamten Ausdrucks wird dieser von links nach rechts gelesen. Jedes Zeichen X des Ausdrucks (Zahl, Klammer oder Operator) wird gemäß folgender Regeln behandelt. Jeder Operator hat eine *Priorität*. „Punktrechnung geht vor Strichrechnung“ wird dadurch ausgedrückt, dass $\times$ eine höhere Priorität hat als $+$ und $-$. Wir geben $\times$ hier die Priorität 2, und $+$ und $-$ erhalten Priorität 1.

1. Ist X eine Zahl, dann wird X auf den Zahlenkeller gelegt.
2. Ist X ein Operator, dann werden solange Operatoren ausgewertet, bis (1) auf dem Operatorenkeller ein Operator mit kleinerer Priorität als X oder eine öffnende Klammer liegt oder (2) der Operatorenkeller leer ist. Abschließend wird X auf den Operatorenkeller gelegt.
3. Ist X die öffnende Klammer „(“, dann wird X auf den Operatorenkeller gelegt.
4. Ist X die schließende Klammer „)“, dann werden solange Operatoren ausgewertet, bis die öffnende Klammer „(“ auf dem Operatorenkeller liegt. Sie wird dann auch noch vom Operatorenkeller entfernt. (Die schließende Klammer kommt nie in den Keller.)

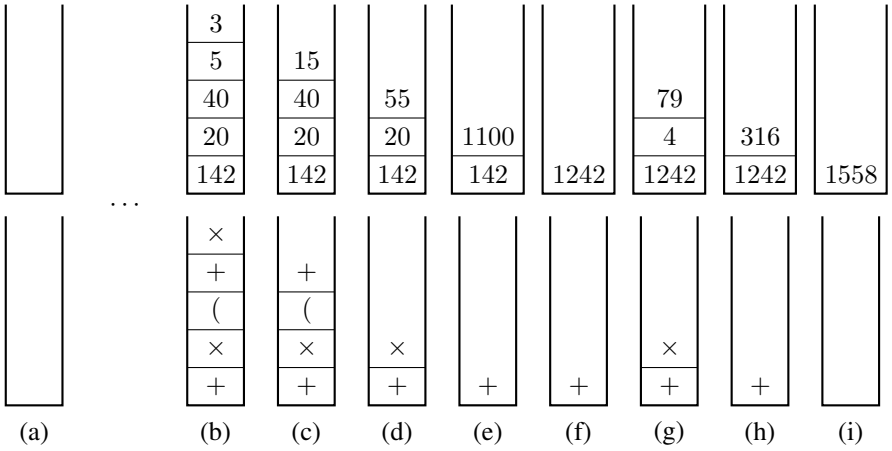

Abbildung 4: Die beiden Keller im Verlauf der Auswertung von $142 + 20 \times (40 + 5 \times 3) + 4 \times 79$

Nachdem der ganze Ausdruck von links nach rechts abgearbeitet wurde, werden abschließend solange Operatoren ausgewertet, bis der Operatorenkeller leer ist. Das Ergebnis steht dann im Zahlenkeller.

Den Algorithmus, der auf diesen vier Regeln beruht, wollen wir uns in einem Beispiel verdeutlichen. Dazu betrachten wir den arithmetischen Ausdruck

$$142 + 20 \times (40 + 5 \times 3) + 4 \times 79\,.$$

Die Kellerinhalte im Verlauf der Auswertung sind in Abbildung 4 dargestellt. Am Anfang sind beide Keller leer (a). Beim Lesen des Anfangsabschnitts $142 + 20 \times (40 + 5 \times 3$, wird jedes gelesene Zeichen gemäß den Regeln auf einen Keller gelegt – jede Zahl auf den Zahlenkeller und jeder Operator auf den Operatorenkeller (b). Beim Lesen der darauffolgenden schließenden Klammer werden solange Operatoren ausgewertet, bis die öffnende Klammer auf dem Operatorenkeller liegt. Sie wird dann auch entfernt (d). Anschließend wird $+$ gelesen. Da auf dem Operatorenkeller ein $\times$ liegt, das höhere Priorität als $+$ hat, wird ein Operator ausgewertet (e). Danach liegt ein $+$ auf dem Operatorenkeller, das die gleiche Priorität wie das gerade gelesene $+$ hat. Also wird wieder ein Operator ausgewertet, und schließlich kommt das zuletzt gelesene $+$ in den Operatorenkeller (f). Danach wird 4×79 gelesen und auf die Keller verteilt (g). Da nun der gesamte Ausdruck gelesen wurde, werden solange Operatoren ausgewertet, bis der Operatorenkeller leer ist. Abschließend steht das Ergebnis 1558 im Zahlenkeller.

3 Berechnung induktiv definierter Funktionen

Die Binärdarstellung[3] einer natürlichen Zahl $n \geqslant 1$ ist die $0/1$-Folge $b_m b_{m-1} \cdots b_1 b_0$, sodass $n = \sum_{i=0}^{m} b_i \cdot 2^i$ und $b_m = 1$. Zum Beispiel hat 19 die Binärdarstellung 10011, da

$$19 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \,.$$

Beim Betrachten der Binärdarstellung $b_m b_{m-1} \cdots b_1 b_0$ von n erkennt man Folgendes.

- b_0 ist der Rest beim Teilen von n durch 2 – diesen Wert bezeichnet man mit $n \bmod 2$ (b_0 ist 1, falls n ungerade ist, und b_0 ist 0, falls n gerade ist).
- $b_m \cdots b_1$ (also die Binärdarstellung von n ohne b_0) ist die Binärdarstellung der (abgerundeten) Hälfte von n – diesen Wert beschreibt man als $\lfloor \frac{n}{2} \rfloor$.

Beim Beispiel 19 ist das letzte (rechteste) Bit der Binärdarstellung 1 (da 19 ungerade ist), und die übrigen Bits sind 1001 – das ist genau die Binärdarstellung von $1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 9$, der abgerundeten Hälfte von 19.

Diese Beobachtung kann man sich zu Nutze machen, wenn man die Binärdarstellung einer Zahl berechnen will. Das Bit, das man als erstes berechnet, ist jedoch das letzte Bit. Bevor man es ausgeben kann, muss man die Binärdarstellung der (abgerundeten) Hälfte der Zahl ausgeben. Um dieses „Aufschieben“ der Ausgabe hinzubekommen, kann man das berechnete letzte Bit in einen Keller legen und dann mit der Berechnung der Binärdarstellung der (abgerundeten) Hälfte fortfahren. Hat man durch das wiederholte Halbieren schließlich die Zahl 0 erreicht, dann weiß man, dass man mit der Berechnung der einzelnen Bits fertig ist. Man muss sie nur noch ausgeben. Das macht man, indem man die Kellerelemente nacheinander entfernt und ausgibt, bis der Keller leer ist. Etwas formaler aufgeschrieben, sieht der Algorithmus wie folgt aus.

Algorithmus zur Ausgabe der Binärdarstellung von n

Phase 1 (bis $n = 0$):	lege $n \bmod 2$ auf den Keller $n := \lfloor \frac{n}{2} \rfloor$
Phase 2 (bis Keller leer):	lies oberstes Kellerelement b und entferne es Ausgabe b

Abbildung 5 zeigt, wie sich die Kellerinhalte bei der Berechnung der Binärdarstellung von 19 mit diesem Algorithmus verändern. Zu Beginn ist der Keller leer. Während Phase 1 wird der Keller immer größer. Am Ende von Phase 1 steht die Binärdarstellung von 19 im Keller und muss nur noch ausgelesen werden.

Die arithmetischen Operationen $n \bmod 2$ und $\lfloor \frac{n}{2} \rfloor$ sehen kompliziert aus, sind aber ganz elementare Rechenoperationen. In der Rechnerhardware nutzt man Register, in denen

[3] Wir betrachten im Folgenden nur die Binärdarstellungen von Zahlen $n \geqslant 1$, da diese stets mit 1 beginnen.

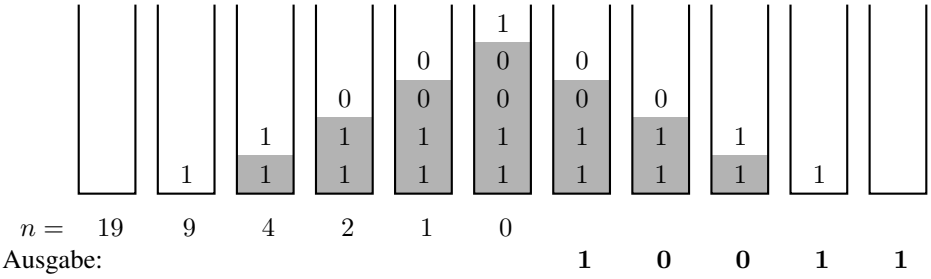

Abbildung 5: Der Verlauf der Kellerinhalte bei der Berechnung der Binärdarstellung von 19

natürliche Zahlen in Binärdarstellung gespeichert werden. Ein elementarer Befehl ist es, alle Bits in einem Register um eine Stelle nach rechts zu schieben (*shift right*). Wenn in einem Register die Zahl n gespeichert ist, steht dort nach einer *shift right* Operation die Zahl $\lfloor \frac{n}{2} \rfloor$. Das Bit, das dabei herausgeschoben wurde, ist $n \bmod 2$ und steht z.B. als *carry* Bit zur Verfügung. In vielen Programmiersprachen gibt es die *shift right* und die entsprechende *shift left* Operation, da man manche Divisionen und Multiplikationen schneller berechnen kann als mit den klassischen arithmetischen Operationen.

Nun ist das Berechnen der Binärdarstellung ein Beispiel, von dem man nicht unbedingt glaubt, dass man es tatsächlich mal gebrauchen kann. Hinter dieser Vorgehensweise steckt jedoch mehr. Dazu schauen wir uns an, wie man eine Potenz a^n berechnen kann. Das naheliegendste Vorgehen ist, n-mal a zu multiplizieren.

$$\underbrace{a \cdot a \cdot \ldots \cdot a}_{n\text{-mal}} \;=\; a^n$$

Dafür benötigt man ungefähr n Multiplikationen. Geht es auch mit deutlich weniger Multiplikationen? Die Antwort ist ja! Wie die Binärdarstellung von n auf die Binärdarstellung von $\lfloor \frac{n}{2} \rfloor$ und $n \bmod 2$ zurückgeführt wurde, können wir auch die Berechnung von a^n auf die Berechnung von $a^{\lfloor \frac{n}{2} \rfloor}$ und ein oder zwei weitere Multiplikationen zurückführen. Betrachten wir zuerst die Berechnung von a^{16}. Gemäß den Potenzregeln ist $a^{16} = a^{8 \cdot 2} = (a^8)^2$. Zur Berechnung von a^{16} braucht man also so viele Multiplikationen wie für a^8 und dann noch eine weitere Multiplikation zum Quadrieren. Damit haben wir gegenüber der naheliegenden Methode bereits 7 Multiplikationen gespart. Entsprechend braucht man zur Berechnung von $a^8 = a^{4 \cdot 2} = (a^4)^2$ die Multiplikationen für a^4 und eine zusätzliche Multiplikation. Zur Berechnung von a^4 braucht man schließlich 2 Multiplikationen. Nutzt man das wiederholte Quadrieren, kann man a^8 mit 4 Multiplikationen ausrechnen. Allgemeiner kann man a^{2^n} mit n Multiplikationen ausrechnen, also zum Beispiel a^{1024} mit 10 Multiplikationen. Das ist schon eine enorme Einsparung. Ist die Potenz keine Zweierpotenz, dann verdoppelt sich die Zahl der Multiplikationen im ungünstigsten Fall. Allgemein gilt für $n \geqslant 1$

$$a^n \;=\; (a^{\lfloor \frac{n}{2} \rfloor})^2 \cdot a^{n \bmod 2}\,.$$

Der verbleibende Fall $a^0 = 1$ (also für $n = 0$) ist einfach. Daraus ergibt sich ein Algorithmus zum Potenzieren, der dem Algorithmus zur Berechnung der Binärdarstellung sehr

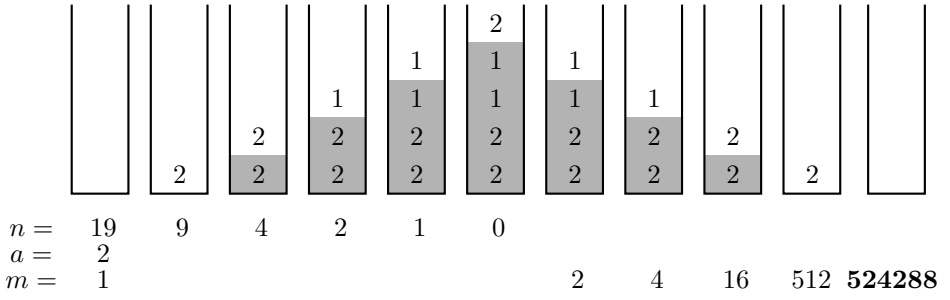

Abbildung 6: Der Verlauf der Kellerinhalte bei der Berechnung von 2^{19}

ähnlich sieht.

Berechne a^n

Phase 1 (bis $n = 0$):	lege $a^{n \bmod 2}$ auf den Keller $n := \lfloor \frac{n}{2} \rfloor$
Phase 2 (bis Keller leer):	lies oberstes Kellerelement b und entferne es $m := m^2 \cdot b$

Das Ergebnis ist der Wert von m, wenn der Algorithmus endet. Die Anzahl der Elemente, die in Phase 1 auf den Keller gelegt werden, ist die Anzahl, wie oft man n wiederholt halbieren muss, bis 0 erreicht wird. Dieser Wert ist (ungefähr) der Logarithmus von n zur Basis 2. Statt n Multiplikationen kommt unser Algorithmus also mit $\log_2 n$ Halbierungen und höchstens $2 \cdot \log_2 n$ Multiplikationen aus. Das ist eine gewaltige Verbesserung. Abbildung 6 zeigt den Verlauf der Kellerinhalte beim Berechnen von 2^{19} mittels wiederholtem Quadrieren.

Das wiederholte Quadrieren kann man auch zum Potenzieren von Matrizen benutzen. Das hilft bei der schnellen Berechnung von Fibonacci-Zahlen. Die Fibonacci-Zahlen bilden die unendliche Folge natürlicher Zahlen $\mathit{fib}_0, \mathit{fib}_1, \mathit{fib}_2, \ldots$ mit der Eigenschaft $\mathit{fib}_0 = 0$, $\mathit{fib}_1 = 1$ und $\mathit{fib}_{n+2} = \mathit{fib}_{n+1} + \mathit{fib}_n$ (für alle $n \geqslant 0$). Um z.B. fib_4 zu berechnen, kann man zuerst $\mathit{fib}_2 = \mathit{fib}_1 + \mathit{fib}_0 = 1$, dann $\mathit{fib}_3 = \mathit{fib}_2 + \mathit{fib}_1 = 2$ und schließlich $\mathit{fib}_4 = \mathit{fib}_3 + \mathit{fib}_2 = 3$ bestimmen. Man berechnet also alle Fibonacci-Zahlen bis hin zu der, die man erreichen will. Um fib_n zu berechnen, benötigt man dann ungefähr n Additionen. Nun gibt es auch eine Darstellung der Fibonacci-Zahlen als Potenzen einer 2×2-Matrix. Es gilt für alle $n \geqslant 1$:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} \mathit{fib}_{n+1} & \mathit{fib}_n \\ \mathit{fib}_n & \mathit{fib}_{n-1} \end{pmatrix}$$

Setzt man

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

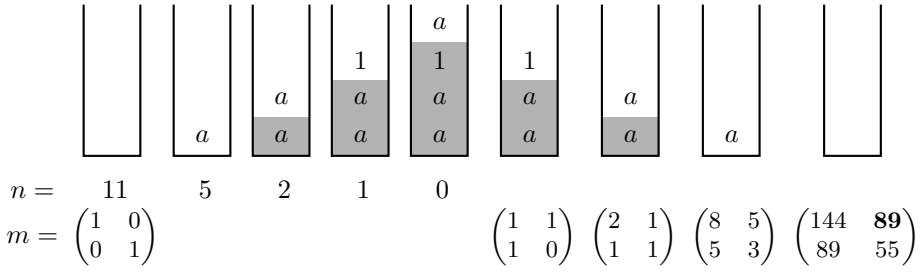

Abbildung 7: Der Verlauf der Kellerinhalte bei der Berechnung von $fib_{11} = 89$

im Algorithmus für das Potenzieren und gibt am Ende nicht die ganze Matrix sondern nur den Eintrag für fib_n aus, dann benötigt man $c \cdot \log_2 n$ Additionen und Multiplikationen, wobei c eine von n unabhängige Konstante ist.

4 Das Teile-und-herrsche-Prinzip

Die drei betrachteten Algorithmen zur Berechnung induktiv definierter Funktionen haben gemeinsam, dass eine „große“ Aufgabe in kleinere Aufgaben aufgeteilt wird, deren Lösungen dann zur Lösung für die große Aufgabe zusammengesetzt werden. Diese algorithmische Herangehensweise nennt man das Teile-und-herrsche-Prinzip. Beim Auswerten einer Formel der Art $(\cdots) \times (\cdots)$ werden zum Beispiel zuerst die beiden geklammerten Teilformeln ausgewertet und deren Werte schließlich zum Wert der ganzen Formel multipliziert. Bei der Berechnung der Potenz a^n werden zunächst die Potenzen $a^{\lfloor \frac{n}{2} \rfloor}$ und $a^{n \bmod 2}$ berechnet, und aus den beiden Teilergebnissen wird dann durch Quadrieren des ersten und Multiplikation mit dem zweiten Teilergebnis der Wert von a^n bestimmt. Um die Potenz $a^{\lfloor \frac{n}{2} \rfloor}$ auszurechnen, geht man genauso vor, bis man schließlich bei der Potenz a^0 angelangt ist, deren Wert 1 man kennt. Dieses Vorgehen kann man induktiv beschreiben

$$a^n = \begin{cases} 1, & \text{falls } n = 0 \\ a^{\lfloor \frac{n}{2} \rfloor} \cdot a^{n \bmod 2}, & \text{falls } n \geqslant 1 \end{cases}$$

und man programmiert es üblicherweise rekursiv, ohne im Programm explizit einen Keller zu benutzen. Die Datenstruktur Keller ist in diesem Sinne in alle üblichen Programmiersprachen eingebaut. Die Daten, die in solchen Kellern gespeichert werden, sind dann nicht nur Zwischenwerte sondern auch Angaben über noch zu lösende Teilaufgaben. Das soll am folgenden Beispiel verdeutlicht werden. Wir betrachten den Algorithmus Mergesort, der nach dem Teile-und-herrsche-Prinzip ein Feld mit Zahlen sortiert. Das „Teilen“ besteht darin, dass man das zu sortierende Feld in zwei (fast) gleich große Hälften aufteilt. Besteht ein Feld nur aus einem Eintrag, so ist es bereits sortiert. Zwei sortierte Felder werden nun zu einem sortierten Feld verbunden. Die organisatorische Arbeit des Algorithmus besteht darin, die Reihenfolge festzulegen, in der zwei (Teil-)Felder verbunden werden. Abbildung 8 zeigt grob den rekursiven Teil von Mergesort zur Strukturie-

```
procedure teile(von-bis)
  hälfte_links und hälfte_rechts sind die beiden Punkte in der Mitte
                                                des Intervalls von-bis
  falls von + 1 < bis dann
    teile(von-hälfte_links)
    teile(hälfte_rechts-bis)
  verbinde(von-hälfte_links, hälfte_rechts-bis)
```

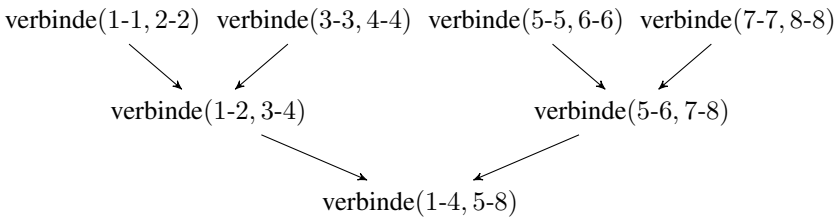

Abbildung 8: Die rekursive Prozedur für Mergesort und die zum Sortieren eines Feldes mit 8 Einträgen zu lösenden Teilaufgaben

rung der Verbindungsoperationen, und den darin implizit beschriebenen Struktur-Graph der auszuführenden Verbindungsoperationen zum Sortieren eines Feldes mit 8 Einträgen. So bedeutet „verbinde(1-4, 5-8)“, dass das Teilfeld mit den ersten vier Einträgen mit dem Teilfeld mit den zweiten vier Einträgen verbunden werden soll. Die Verbindungsoperation funktioniert allerdings nur, wenn beide Teilfelder bereits sortiert vorliegen. Die eingehenden Pfeile geben an, welche Verbindungsoperationen deshalb bereits vorher ausgeführt worden sein mussten.

Goldstine und von Neumann – die Erfinder von Mergesort – hatten 1947 in ihrer Arbeit über Mergesort [GvN48] weder Keller noch Rekursion zur Verfügung. Deshalb mussten sie einen recht hohen technischen Aufwand betreiben, um den Strukturierungsprozess zu beschreiben. Durch die Verinnerlichung der Denkweise des Kellers und der darauf basierenden Rekursion ist uns dieses Problem abgenommen worden.

Literatur

[GvN48] Herman H. Goldstine und John von Neumann. Planning and coding for an electronic computing instrument. Bericht, The Institute for Advanced Study, Princeton, NJ, 1948.