Amortisierte Analyse

Kurt Mehlhorn
Max-Planck-Institut für Informatik
Saarbrücken

16. Juni 2000

1 Einleitung

Wir betrachten die Ausführung einer Folge von Operationen Op_1, Op_2, $\ldots$, Op_n auf einer Datenstruktur und bezeichnen die Ausführungszeit der i-ten Operation mit T_i und die Gesamtausführungszeit $T_1 + T_2 + \ldots + T_n$ aller Operationen mit T. Bei sehr vielen Datenstrukturen können die Einzellaufzeiten der Operationen stark schwanken. In diesen Fällen ist die Abschätzung der Gesamtlaufzeit als das n-fache der Maximallaufzeit einer Einzeloperation oft wenig aussagekräftig. Vielmehr sollte man versuchen, die Kosten der teuren Operationen über die Gesamtfolge zu amortisieren. In der Betriebswirtschaft ist dieser Begriff der Amortisierung seit jeher gebräuchlich. Teure Ereignisse, z.B. der Kauf einer neuen Maschine, werden über die Dauer ihrer Wirksamkeit amortisiert. Bei der Analyse von Algorithmen wurde der Begriff der Amortisierung zum ersten Mal Anfang der 80er Jahre benutzt. Inzwischen ist sie zu einer äußerst wirksamen Methode herangereift, ohne die kaum noch eine Arbeit im Bereich Algorithmen auskommt.

Wir geben zunächst ein einfaches Beispiel für eine amortisierte Analyse: die Gesamtkosten von n Inkrementieroperationen auf einem binären Zähler. Wir geben zuerst eine globale Analyse und dann eine Analyse mit Hilfe der sogenannten *Potentialmethode*. Die Potentialmethode ist die gebräuchlichste Methode zur amortisierten Analyse. Eine Alternative dazu ist die *Bankkontomethode*, die wir im Hauptteil der Arbeit zur Analyse von sogenannten 2-5-Bäumen benutzen. Im letzten Teil der Arbeit abstrahieren wir dann, zeigen die Universalität der Potentialmethode und geben einen kurzen historischen Rückblick.

2 Der Binärzähler

Wir kennen alle die Binärdarstellung von natürlichen Zahlen; 10 ist die Darstellung der Zahl 2, 101 ist die Darstellung der Zahl 5, und allgemein ist $\ldots\beta_2\beta_1\beta_0$ mit $\beta_i \in \{0,1\}$ die Binärdarstellung von $\sum_i \beta_i \cdot 2^i$.

Wir zählen von 0 bis n unter Verwendung der Binärdarstellung. Also

$$0 \longrightarrow 1 \longrightarrow 10 \longrightarrow 11 \longrightarrow 100 \longrightarrow 101 \longrightarrow \ldots$$

Es fällt auf, daß der Übertrag manchmal weit läuft, z.B. beim Übergang von 101111 nach 110000, und manchmal gar nicht läuft, z.B. bei Übergang von 110000 nach 110001. Stellen wir uns vor, daß es eine Einheit kostet, eine Ziffer durch eine andere zu ersetzen. Dann kostet der Übergang $101111 \longrightarrow 110000$ fünf Einheiten, der Übergang $110000 \longrightarrow 110001$ eine Einheit, und allgemein kostet der Übergang $i \longrightarrow i+1$ eine Einheit mehr als es endende Einsen in der Binärdarstellung $bin(i)$ von i gibt. Wir setzen also

$$cost(i \longrightarrow i+1) = 1 + \text{ Anzahl der endenden Einsen in } bin(i)$$

Die Kosten des Übergangs von i nach $i+1$ schwanken stark. Sie können so klein wie 1 und so groß wie $1 + \lceil \log n \rceil$ sein. Die Abschätzung der Gesamtkosten durch das n-fache der maximalen Einzelkosten ergibt daher $n \cdot (1 + \lceil \log n \rceil)$. Ziel dieses Abschnittes ist es zu zeigen, daß die Gesamtkosten linear beschränkt sind.

Satz 1 *Die Gesamtkosten des Zählens von* 0 *nach* n *sind durch* $2n$ *beschränkt.*

Wir geben zwei Beweise. Der erste Beweis benutzt eine globale Betrachtung. Wir beginnen mit der Beobachtung, daß die Kosten des Übergangs von i nach $i+1$ gleich $k+1$ sind genau dann, wenn die Binärdarstellung von i mit der Ziffernfolge 01^k endet[1]. Wir stellen uns dabei vor, daß Binärdarstellungen nach links mit Nullen aufgefüllt sind, d.h., die Darstellung der 3 ist etwa $\ldots 0 \ldots 011$. Dies erspart uns eine Sonderbehandlung des Falls, bei dem die Binärdarstellung nur aus Einsen besteht. Wir beobachten als nächstes, daß es höchstens $2^{\lceil \log n \rceil - (k+1)}$ Zahlen kleiner n gibt, deren Binärdarstellung mit 01^k endet. Dies gilt, weil die Binärdarstellung der Zahlen kleiner n aus $\lceil \log n \rceil$ Bits bestehen und die letzten $k+1$ Bits auf 01^k fixiert sind. Aus beiden Beobachtungen zusammen folgt, daß es höchstens $2^{\lceil \log n \rceil - k+1}$ Übergänge mit Kosten $k+1$ gibt. Also gilt:

$$\begin{aligned}
\text{Gesamtkosten} \quad &\leq \quad \textstyle\sum_{1 \leq k \leq \lceil \log n \rceil} (k+1) \cdot 2^{\lceil \log n \rceil - (k+1)} \\
&\leq \quad 2^{\lceil \log n \rceil} \cdot \textstyle\sum_{k \geq 1} k/2^k \\
&\leq \quad 2 \cdot n \cdot 2 \\
&\leq \quad 4 \cdot n
\end{aligned}$$

Dabei folgt die zweite Ungleichung durch Ersetzen von $k+1$ durch k und die dritte Ungleichung aus $2^{\lceil \log n \rceil} \leq 2^{1+\log n} \leq 2 \cdot n$ und

$$\begin{aligned}
\sum_{k \geq 1} k/2^k \quad &= \quad 1/2 + 2/4 + 3/8 + 4/16 + \ldots \\
&= \quad 1/2 + 1/4 + 1/4 + 1/8 + 1/8 + 1/8 + 1/16 + 1/16 + 1/16 + 1/16 + \ldots \\
&= \quad 1/2 + 1/4 + 1/8 + 1/16 + \ldots + 1/4 + 1/8 + 1/16 + \ldots \\
& \qquad +1/8 + 1/16 + \ldots + 1/16 + \ldots \\
&= \quad 1 + 1/2 + 1/4 + 1/8 + \ldots \\
&= \quad 2.
\end{aligned}$$

Wir geben als nächstes einen alternativen Beweis, der mit einer lokalen Betrachtung auskommt und auch tatsächlich die Schranke $2n$ beweist. Wir starten mit zwei Beobachtungen:

- Nur Zählerstände mit vielen (endenden) Einsen können zu teuren Operationen führen und
- teure Operationen ersetzen viele Einsen durch Nullen und nur eine Null durch eine Eins.

Es liegt daher nahe, die Anzahl der Einsen in $bin(i)$ genauer zu untersuchen. Wir definieren

$$pot(i) = \text{ Anzahl der Einsen in } bin(i)$$

und interpretieren $pot(i)$ als das "Potential" der Zahl i. Das folgende Lemma ist zentral.

Lemma 1 $pot(i+1) - pot(i) + cost(i \longrightarrow i+1) = 2.$

Beweis: Bezeichne die Anzahl der endenden Einsen in $bin(i)$ mit k, d.h., $bin(i) = \ldots 01^k$. Dann ist $bin(i+1) = \ldots 10^k$ und daher ist $pot(i+1) - pot(i) = 1 - k$ und $cost(i \longrightarrow i+1) = k+1$. ∎

Wir summieren nun die obige Gleichung für i von 0 bis $n-1$ und beobachten noch, daß die Terme $pot(i)$, $1 \leq i \leq n-1$, mit beiden Vorzeichen vorkommen und daher wegfallen. Wir erhalten

$$pot(n) - pot(0) + \text{ Gesamtkosten } = 2n$$

und daher

$$\text{Gesamtkosten } \leq 2n,$$

da $pot(n) \geq 0$ und $pot(0) = 0$.

Es lohnt sich, den alternativen Beweis mehrmals zu lesen, damit sich seine Eleganz voll erschließt. Es lohnt sich auch, den Beweis etwas anders zu interpretieren. Das führt zur Bankkontomethode der Amortisierung. Wir sagten oben, daß es eine Einheit kostet, um eine Ziffer durch eine andere zu ersetzen. Nennen

[1] 01^k bezeichnet dabei eine 0 gefolgt von k Einsen.

wir der Einfachheit halber unsere Kosteneinheiten RE[2] und stellen wir uns vor, daß auf jeder Eins eine RE liegt. Wenn wir nun einer jeden Operation 2 RE mitgeben, dann können alle Übergänge bezahlt werden, denn ein Übergang von $\ldots 01^k$ nach $\ldots 10^k$ kostet $k+1$ RE und setzt $k-1$ RE frei. Die Nettokosten sind also gerade die 2 RE, die der Operation belastet werden.

3 2-5-Bäume

Abbildung 1 zeigt einen 2-5-Baum. In einem 2-5-Baum hat jeder Knoten mindestens 2 und höchstens 5 Kinder und haben alle Blätter die gleiche Tiefe. Wir bezeichnen die gemeinsame Tiefe aller Blätter mit h und nennen h die Höhe des Baums. Die Höhe eines 2-5-Baums ist logarithmisch in der Anzahl seiner Blätter.

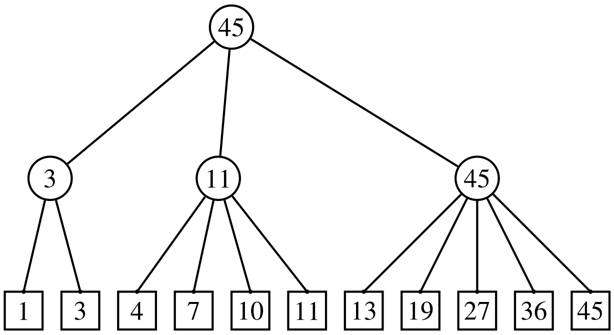

Abbildung 1: Ein 2-5-Baum

Lemma 2 *Für einen 2-5-Baum der Höhe h und mit n Blättern gilt:* $\log_5 n \leq h \leq \log_2 n$.

Beweis: Da jeder Knoten höchstens 5 Kinder hat, gilt $n \leq 5^h$, und da jeder Knoten mindestens zwei Kinder hat, gilt $n \geq 2^h$. Logarithmieren liefert die Behauptung. ∎

2-5-Bäume kann man als Suchbäume benutzen. Sei $x_1, \ldots, x_n$ eine aufsteigend sortierte Folge von n Zahlen. Man schreibt x_i in das i-te Blatt von links und schreibt in jeden inneren Knoten die Beschriftung des rechtesten Blatts in seinem Unterbaum. Abbildung 1 zeigt ein Beispiel. Um in unserem Beispiel nach der Zahl 8 zu suchen, vergleicht man zunächst 8 mit den Kindern der Wurzel. Da $3 < 8$ und $8 \leq 11$ sucht man in dem Unterbaum mit der Wurzel 11 weiter, $\ldots$. Allgemein verbringt man bei dieser Vorgehensweise konstante Zeit auf jedem Niveau des Baums. Daher garantieren 2-5-Bäume logarithmische Suchzeit.

Wir wenden uns nun dem dynamischen Verhalten von 2-5-Bäumen zu. Das Einfügen eines Elements entspricht einer Suche nach dem Element und dem Hinzufügen eines neuen Blatts; das Streichen eines Elements entspricht einer Suche nach dem Element und dem Streichen eines Blatts. Abbildungen 2 und 3 zeigen das Hinzufügen und das Streichen eines Blatts in abstrahierter Form. Das Hinzufügen bzw. Streichen eines Blatts kann die 2-5-Baum-Eigenschaft zerstören. Hinzufügen kann einen Knoten mit sechs Kindern, und Streichen kann einen Knoten mit einem Kind erzeugen. Wir sagen statt Knoten mit k Kindern im folgenden $k$er Knoten.

2-5-Bäume stellen drei Operationen zur Verfügung, um 6er und 1er Knoten zu beseitigen. Wir beseitigen 6er Knoten durch Spalten, siehe Abbildung 4, und 1er Knoten durch Verschmelzen, siehe Abbildung 5, und Stehlen, siehe Abbildung 6. Wir sehen uns die drei Operationen genauer an.

[2] RE steht für Recheneinheit.

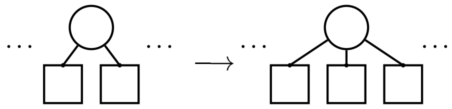

Abbildung 2: Hinzufügen eines Blatts. Der Grad des Elterknoten wächst um eins.

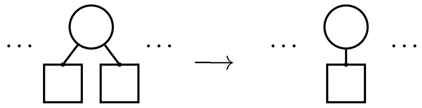

Abbildung 3: Streichen eines Blatts. Der Grad des Elterknoten vermindert sich um eins.

Beim Spalten ersetzen wir einen 6er Knoten durch zwei 3er Knoten. Dies erhöht die Anzahl der Kinder des Elter des 6er Knoten um eins. Falls das Elter dadurch zum 6er Knoten wird, spalten wir es, … . Spätestens wenn wir die Wurzel spalten und eine neue Wurzel darüber setzen, endet der Prozeß des Spaltens[3].

Sehen wir uns als nächstes die Behandlung von 1er Knoten an. Wir machen eine Fallunterscheidung nach dem Grad eines der Geschwisterknoten[4]. Falls das Geschwister ein 2er Knoten ist, dann verschmelzen wir den 1er und den 2er Knoten zu einem 3er Knoten; dies verringert die Anzahl der Kinder des Elter um eins. Falls das Geschwister kein 2er Knoten ist, also ein 3er, 4er oder 5er Knoten, dann nehmen wir ihm ein Kind weg. Dies stellt die 2-5-Baum-Eigenschaft wieder her. Beim Verschmelzen müssen wir jedoch als nächstes das Elter betrachten, … . Wie das Spalten kann sich das Verschmelzen zur Wurzel des Baums hin fortpflanzen. Spätestens wenn wir eine Wurzel vom Grad eins erzeugen, sind wir fertig. Wir brauchen sie nämlich nur zu streichen, um wieder einen 2-5-Baum zu erhalten.

Wir fassen den Stand der Diskussion zusammen.

Nach dem Hinzufügen bzw. Streichen eines Blatts kann die 2-5-Baum-Eigen-schaft durch die sogenannten Rebalancierungsoperationen Spalten bzw. Verschmelzen und Stehlen wiederhergestellt werden. Die Anzahl der Rebalancierungsoperationen schwankt zwischen 0 und $\log n$.

Bei n Hinzufügungen und Streichungen von Blättern könnte also die Gesamtzahl der Rebalancierungsoperationen so groß wie $n \log n$ sein. Wir zeigen:

Satz 2 *Startet man mit einem leeren 2-5-Baum und macht n Hinzufügungen und Streichungen von Blättern, so ist die Gesamtzahl der Rebalancierungsoperationen höchstens $2n$.*

Wir benutzen die Bankkontomethode für den Beweis. Wir stellen uns vor, daß jede Rebalancierungsoperation eine Recheneinheit (RE) kostet und postulieren folgende Invariante.

Invariante*: In den Knoten des Baums sind unbenutzte Recheneinheiten gemäß der folgenden Tabelle gespeichert:*

Grad	1	2	3	4	5	6
RE	3	1	0	0	1	3

Auf 1er und 6er Knoten liegen also 3 *RE, auf 2er und 5er Knoten liegt jeweils* 1 *RE und auf 3er und 4er Knoten liegen keine RE.* In der Invariante sprechen wir auch über 1er und 6er Knoten, da sie während des Rebalancierens auftreten können.

Wir zeigen nun, daß es genügt, jedem Hinzufügen oder Streichen eines Blatts 2 RE zu berechnen, um für das nachfolgende Rebalancieren zu bezahlen. Hinzufügen eines Blatts erhöht den Grad eines Knoten

[3]Die Höhe von 2-5-Bäumen ändert sich durch Entstehen und Verschwinden der Wurzel.

[4]Es spielt keine Rolle, welches der Geschwister man betrachtet.

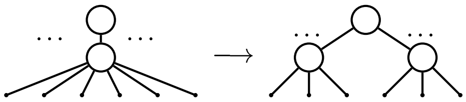

Abbildung 4: Spalten eines 6er Knoten in zwei 3er Knoten. Der Grad des Elterknoten wächst um eins.

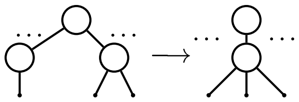

Abbildung 5: Verschmelzen eines 1er und eines 2er Knoten zu einem 3er Knoten. Der Grad des Elterknoten vermindert sich um eins.

um eins und Streichen eines Blatts erniedrigt den Grad eines Knoten um eins. In jedem Fall genügen die zwei neuen RE, um unsere RE-Invariante aufrechtzuerhalten. Die zwei RE werden sogar nur gebraucht, wenn wir aus einem 5er Knoten einen 6er Knoten machen bzw. aus einem 2er Knoten einen 1er Knoten.

Als nächstes sehen wir uns das Spalten von 6er Knoten an. Auf einem 6er Knoten liegen drei RE. Spalten eines 6er Knoten erzeugt zwei 3er Knoten und erhöht den Grad des Elter um eins. Wir brauchen keine RE für die 3er Knoten und bis zu zwei zusätzliche RE für das Elter. Auf jeden Fall wird eine RE frei, die wir benutzen, um für das Spalten zu bezahlen.

Beim Verschmelzen eines 1er Knoten und eines 2er Knoten ist die Argumentation ähnlich. Wir haben 4 RE zur Verfügung, drei auf dem 1er Knoten und eine auf dem 2er Knoten. Verschmelzen erzeugt einen 3er Knoten und verringert den Grad des Elter um eins. Wir brauchen keine RE für den 3er Knoten und bis zu zwei zusätzliche RE für das Elter. Auf jeden Fall werden mindestens zwei RE frei. Wir benutzen eine davon, um für das Verschmelzen zu bezahlen und lassen die andere unbenutzt.

Schließlich behandeln wir noch das Stehlen eines Kindes durch einen 1er Knoten von einem Geschwister, das Grad 3, 4 oder 5 hat. Es stehen uns drei RE auf dem 1er Knoten zur Verfügung. Stehlen macht aus dem 1er Knoten einen 2er Knoten und verringert den Grad des Geschwisters um eins. Wir brauchen eine RE für den 2er Knoten und höchstens eine zusätzlich RE für das Geschwister. Auf jeden Fall wird eine RE frei, die wir benutzen, um für das Stehlen zu bezahlen.

Damit ist Satz 2 bewiesen. Die Rebalancierungsoperationen werden aus den durch sie freigesetzten RE bezahlt, und jedes Hinzufügen und Streichen eines Blatts zahlt zwei RE ein. Da jede Rebalancierungsoperation eine RE verbraucht, und insgesamt nur $2n$ RE einbezahlt werden, kann es höchstens $2n$ Rebalancierungsoperationen geben.

4 Das Sortieren vorsortierter Folgen

In diesem Abschnitt wenden wir Satz 2 auf das Sortierproblem an. Seien n Zahlen $x_1, x_2, \ldots, x_n$ gegeben. Es gilt, sie in aufsteigender Reihenfolge zu sortieren. Bei Eingabe 4, 7, 8, 3, 1 sollen wir also 1, 3, 4, 7, 8 erzeugen, und bei Eingabe 2, 1, 4, 5, 6 sollen wir 1, 2, 4, 5, 6 erzeugen. Es scheint intuitiv klar zu sein, daß das zweite Sortierproblem einfacher ist, denn die Eingabe ist ja schon fast sortiert. Wir fassen den Begriff des "Fast-Sortiert-Seins" in der Definition der *Anzahl der Inversionen* einer Folge. Sei

$$F = |\{(i, j); 1 \leq i < j \leq n \text{ und } x_i > x_j\}|$$

die Anzahl der Paare von Elementen, die gegenüber der Sortierreihenfolge vertauscht ("invertiert") sind. F heißt die Anzahl der Inversionen. Die Anzahl der Inversionen liegt zwischen 0 und $n(n-1)/2$. Wir werden

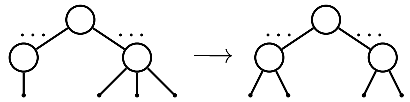

Abbildung 6: Stehlen eines Kindes durch einen 1er Knoten. Der Grad des Elterknoten bleibt unverändert.

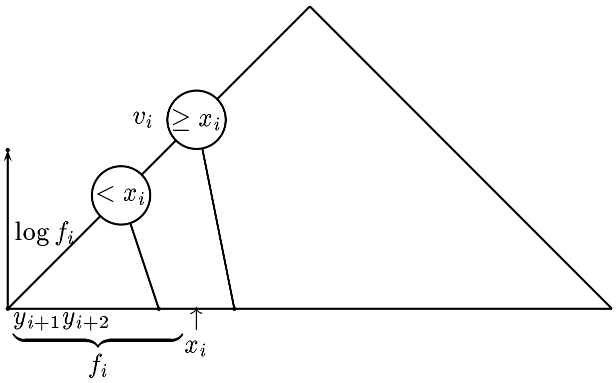

Abbildung 7: Die Suche der Einfügestelle für x_i beginnt am linkesten Blatt, steigt am linken Rückgrat bis zum Knoten v_i und dreht dann um.

einen Sortieralgorithmus angeben, der Folgen mit wenigen Inversionen in Linearzeit sortiert und nie mehr als $O(n \log n)$ Vergleiche braucht.

Wir benutzen Sortieren durch Einfügen. Nehmen wir an, wir hätten die Zahlen $x_{i+1}, \ldots, x_n$ schon sortiert, und bezeichnen wir die sortierte Folge mit $y_{i+1}, \ldots, y_n$. In diese Folge wollen wir nun x_i an der richtigen Stelle einfügen. Bevor wir erklären, wie wir die Stelle finden, an der x_i einzufügen ist, führen wir eine Notation für die Stelle ein: x_i soll an der f_i-ten Stelle von links eingefügt werden.

$$\underbrace{y_{i+1} y_{i+2} \cdots}_{f_i} \quad \underset{x_i}{\uparrow} \quad \cdots y_n$$

Die f_i Elemente $y_{i+1}, y_{i+2}, \ldots, y_{i+f_i}$ stammen aus den $x_{i+1}, \ldots, x_n$ und sind kleiner als x_i, d.h.,

$$f_i = |\{j; i < j \text{ und } x_j < x_i\}|.$$

f_i ist also gerade die Anzahl der Inversionen (i, j) mit erster Komponente i. Daher gilt $F = \sum f_i$ und daher müssen viele der f_i's klein sein, wenn F klein sein soll. In anderen Worten, wenn F klein ist, dann werden viele der x_i weit vorne eingefügt.

Wie können wir nun die Stelle, an der x_i einzufügen ist, schnell finden? Wir speichern dazu $y_{i+1}, \ldots, y_n$ in den Blättern eines 2-5-Baums und benutzen eine etwas ungewöhnliche Suchmethode, siehe Abbildung 7. Wir starten am linkesten Blatt und laufen in Richtung Wurzel, bis wir auf einen Knoten v_i mit Beschriftung größer gleich x_i treffen. Das linkeste Kind dieses Knotens hat demnach eine Beschriftung kleiner als x_i. Daher wird x_i in den Unterbaum mit Wurzel v_i eingefügt, aber nicht(!!) unterhalb des linkesten Kindes von v_i. Startend in v_i bestimmen wir die Position von x_i mit einer ganz normalen Baumsuche, fügen x_i ein und rebalancieren den Baum mit Hilfe von Spaltungen.

Wieviel kostet es, x_i einzufügen? Sei h_i die Höhe des Knoten v_i, und sei r_i die Anzahl der Rebalancierungsoperationen nach dem Einfügen. Dann entstehen Kosten $O(h_i + 1 + r_i)$, wobei 1 die Kosten für das Hinzufügen eines Blatts sind. Wir schätzen nun die Summen $\sum_{1\le i\le n} h_i$, $\sum_{1\le i\le n} 1$ und $\sum_{1\le i\le n} r_i$ getrennt ab.

Nach Satz 2 ist $\sum_{1\le i\le n} r_i \le 2n$. Sicher ist $\sum_{1\le i\le n} 1 = n$. Um $\sum_{1\le i\le n} h_i$ abzuschätzen beobachten wir, daß der linkeste Unterbaum von v_i weniger als f_i Blätter hat. Daher gilt $f_i \ge 2^{h_i-1}$, denn die Höhe des Unterbaums ist $h_i - 1$. Logarithmieren liefert $h_i \le 1 + \log f_i$, und Summation liefert dann weiter

$$\begin{aligned}
\textstyle\sum_{1\le i\le n} h_i &\le \textstyle\sum_{1\le i\le n} 1 + \log f_i \\
&= n + \log \textstyle\prod_{1\le i\le n} f_i \\
&\le n + \log(F/n)^n \\
&= n + n\log(F/n)
\end{aligned}$$

Dabei folgt die zweite Ungleichung aus der Beobachtung, daß das Produkt $\prod_{1\le i\le n} f_i$ maximal ist, wenn alle Faktoren gleich groß sind, also gleich F/n sind. Wir fassen zusammen.

Satz 3 *Eine Folge von n Elementen mit F Inversionen kann in Zeit*

$$O(n + n\log(F/n))$$

sortiert werden.

Für $F = O(n)$ ist das linear, für $F = O(n\log n)$ ist das $O(n\log\log n)$, und für $F = O(n^2)$ ist das $O(n\log n)$.

5 Amortisierte Kosten, eine abstrakte Betrachtung

Wir betrachten eine Datenstruktur $\mathcal{D}$, die die Zustände $s_0, s_1, s_2, \ldots$ annehmen kann. In Falle von 2-5-Bäumen sind die Zustände alle möglichen 1-6-Bäume. Auf der Datenstruktur seien Operationen $X, Y, \ldots$ ausführbar. Die Anwendung einer Operation X führt die Datenstruktur aus einem Zustand s in einen Zustand s' über. Dabei darf eine Operation im allgemeinen nur auf eine Teilmenge der Zustände angewendet werden. Im Falle von 2-5-Bäumen sind die Operationen das Hinzufügen und Streichen eines Blatts, das Spalten, Verschmelzen und das Stehlen. Hinzufügen und Streichen sind nur in einem 2-5-Baum erlaubt, Spalten nur bei einem 1-6-Baum mit genau einem 6er Knoten, usw. . Wir sind an *legalen* Folgen von Operationen $Op_1, Op_2, Op_3, \ldots, Op_n$ interessiert: Startend im Anfangszustand s_0 erzeugt diese Folge die Zustandsfolge

$$s_0 \xrightarrow{Op_1} s_1 \xrightarrow{Op_2} s_2 \xrightarrow{Op_3} \cdots \xrightarrow{Op_n} s_n$$

Die Forderung der Legalität bedeutet dabei, daß die Operation Op_i auf den Zustand s_{i-1} angewendet werden darf, für $i = 1, 2, \ldots$.

Sei nun $T_X(s)$ die Laufzeit der Operation X im Zustand s. Bei 2-5-Bäumen haben alle fünf Operationen Laufzeit Eins. Für eine legale Folge $F = Op_1, \ldots, Op_n$ von Operationen mit zugehöriger Zustandsfolge $s_0, s_1, \ldots, s_n$ definieren wir die *Gesamtlaufzeit* $T(F)$ als

$$T(F) = \sum_{1\le i\le n} T_{Op_i}(s_{i-1}).$$

Wir nennen s_n den *Endzustand* von F.

Eine Familie von Funktionen $A_X(s)$ heißt *amortisierte Laufzeitschranken*, wenn für jede legale Folge F von Operationen gilt:

$$T(F) \le A(F) := \sum_{1\le i\le n} A_{Op_i}(s_{i-1}),$$

d.h., wenn die tatsächliche Gesamtlaufzeit durch die amortisierte Gesamtlaufzeit beschränkt ist.

Bei 2-5-Bäumen können wir $A_{\text{Hinzufügeblatt}} = A_{\text{Streicheblatt}} = 1 + 2 = 3$ und $A_{\text{Spalte}} = A_{\text{Verschmelze}} = A_{\text{Stehle}} = 0$ setzen. Das kann man, wie folgt, einsehen. Wir beobachten zunächst, daß eine legale Folge aus Abschnitten der folgenden Art besteht:

Hinzufügen eines Blatts gefolgt von Rebalancierungsoperationen

bzw.

Streichen eines Blatts gefolgt von Rebalancierungsoperationen

Sei r_i die Anzahl der Rebalancierungsoperationen im i-ten Abschnitt, und sei n die Anzahl der Abschnitte. Dann sind die Gesamtkosten $\sum_{1 \le i \le n}(1 + r_i) = n + \sum_{1 \le i \le n} r_i \le n + 2n = 3n$, wobei die Ungleichung aus Satz 2 folgt. Nun ist aber $3n$ gerade die Summe der amortisierten Kosten, da die erste Operation in jedem Abschnit amortisierte Kosten 3 hat. Also sind die tatsächlichen Gesamtkosten durch die amortisierten Gesamtkosten beschränkt.

Wir zeigen nun, daß die Potential- und Bankkontomethode es direkt erlauben, amortisierte Laufzeitschranken zu gewinnen.

Satz 4 *Sei S die Menge der Zustände einer Datenstruktur, sei s_0 der Anfangszustand, und sei $pot : S \longrightarrow \mathbb{R}_{\ge 0}$ eine Funktion mit $pot(s_0) = 0$. Für eine Operation X und einen Zustand s mit $s \xrightarrow{X} s'$ definiere*

$$A_X(s) = pot(s') - pot(s) + T_X(s).$$

Dann sind die A_X amortisierte Laufzeitschranken.

Beweis: Es bedarf nur einer kurzen Rechnung. Wir haben für eine Folge $Op_1, \ldots, Op_n$ von Operationen:

$$\begin{aligned} \textstyle\sum_{1 \le i \le n} A_{Op_i}(s_{i-1}) &= \textstyle\sum_{1 \le i \le n}(pot(s_i) - pot(s_{i-1}) + T_{Op_i}(s_{i-1})) \\ &= pot(n) - pot(0) + \textstyle\sum_{1 \le i \le n} T_{Op_i}(s_{i-1}) \\ &\ge \textstyle\sum_{1 \le i \le n} T_{Op_i}(s_{i-1}). \end{aligned}$$

∎

Bei 2-5-Bäumen definieren wir das Potential eines Baums als die Gesamtzahl der gemäß der RE-Invariante auf dem Baum gespeicherten Recheneinheiten. Sei nun etwa T ein Baum, auf den eine Spalteoperation anwendbar ist, und sei T' der Baum nach dem Spalten. Dann ist $pot(T') - pot(T) \le -1$. Die tatsächlichen Kosten sind 1. Also dürfen wir die amortisierten Kosten auf 0 setzen, wie wir das weiter oben ja auch getan haben.

Die Potentialmethode wurde seit ihrer Erfindung Anfang der 80er Jahre mit großem Erfolg zur Herleitung amortisierter Laufzeitschranken eingesetzt. Wir zeigen nun noch, daß sie im Prinzip in der Lage ist, jede amortisierte Laufzeitschranke herzuleiten.

Satz 5 *Sei $B_X(s)$ eine Familie von amortisierten Laufzeitschranken. Dann gibt es eine Potentialfunktion pot, so daß für die gemäß Satz 4 definierten amortisierten Laufzeitschranken $A_X(s)$ gilt:*

$$A_X(s) \le B_X(s)$$

für alle Zustände s und alle Operationen X.

Beweis: Für einen Zustand s sei $\mathcal{F}(s)$ die Menge der legalen Folgen von Operationen mit Endzustand s. Wir definieren Durch

$$pot(s) = \inf\{B(F) - T(F); F \text{ ist eine legale Folge von Operationen mit Endzustand } s\}$$

eine Potentialfunktion[5]. In der Tat ist wegen $T(F) \le B(F)$ für alle Folgen F, $pot(s) \ge 0$ für alle s. Ferner hat die leere Folge von Operationen Endzustand s_0, und daher ist $pot(s_0) = 0$.

Damit können wir gemäß Satz 4 die amortisierten Laufzeitschranken $A_X(s)$ definieren. Wir zeigen, daß $A_X(s) \le B_X(s)$ für alle X und s. Dafür genügt es zu zeigen, daß $A_X(s) \le B_X(s) + \epsilon$ für jede positive reele Zahl ϵ. Sei s' der Nachfolgezustand von s unter der Operation X, d.h., $s \xrightarrow{X} s'$, und sei F

[5] Wir müssen inf statt min schreiben, da es unendlich viele Folgen mit Endzustand s geben kann.

eine Folge von Operationen mit Endzustand s und $B(F) - T(F) \leq pot(s) + \epsilon$. Sei F' die Folge, die aus F gefolgt von X besteht. Dann gilt

$$\begin{aligned} pot(s') &\leq B(F') - T(F') \\ pot(s) &\geq B(F) - T(F) - \epsilon \\ B(F') &= B(F) + B_X(s) \\ T(F') &= T(F) + T_X(s) \\ A_X(s) &= pot(s') - pot(s) + T_X(s) \end{aligned}$$

und daher

$$\begin{aligned} A_X(s) &\leq (B(F') - T(F')) - (B(F) - T(F) - \epsilon) + T_X(s) \\ &= (B(F') - B(F)) - (T(F') - T(F) - T_X(s)) + \epsilon \\ &= B_X(s) + \epsilon \end{aligned}$$

Damit ist der Satz gezeigt. ∎

Satz 5 ist natürlich keine konkrete Hilfe bei der Suche nach einer Potentialfunktion, aber doch eine psychologische Stütze: Es gibt sie immer.

6 Geschichtlicher Rückblick

Amortisierung ist so alt wie die Analyse von Algorithmen. Zunächst wurden allerdings nur globale Methoden wie in unserem ersten Beweis in Abschnitt 2 benutzt. Die Bankkontomethode und die Potentialmethode zur Amortisierung wurde Anfang der achziger Jahre von R.E. Brown, S. Huddlestone, K. Mehlhorn. D.D. Sleator und R.E. Tarjan eingeführt [BT80, HM82, ST83, ST85]. Satz 2 wurde in [HM82] bewiesen. Die Autoren schreiben: "The proof follows a general paradigm of analyzing the cost of sequences of operations, the bank account paradigm. The paradigm defines a bank account for a tree, and associates operations on the tree with deposits and withdrawals in the account. A bound on rebalancing follows by relating the constraints on deposits and withdrawals to the initial and finite balances of the account". Der Übersichtsartikel [Tar85] hat den Begriff der amortisierten Analyse popularisiert und die Begriffe amortisierte Analyse, Potentialmethode und Bankkontomethode festgeklopft. Satz 4 findet sich zum ersten Mal in dieser Arbeit. Eine frühe Darstellung findet sich auch in [Meh84]. Satz 5 scheint neu zu sein. Die Anwendung von Satz 2 auf das Sortieren vorsortierter Folgen wird in [Meh79, BT80] beschrieben. Das Ergebnis stammt aber schon von [GMPR77].

Literatur

[BT80] M.R. Brown and R.E. Tarjan. Design and analysis of a data structure for representing sorted lists. *SICOMP*, 9:594–614, 1980.

[GMPR77] L.J. Guibas, E. McCreight, M. Plass, and J. Roberts. A new representation for linear lists. In *STOC*, pages 49–60, 1977.

[HM82] S. Huddlestone and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[Meh79] K. Mehlhorn. Sorting presorted files. In *4-th GI Conference on Theoretical Computer Science*, pages 199–212. LNCS 67, 1979.

[Meh84] K. Mehlhorn. *Data Structures and Algorithms*. Springer Verlag, 1984.

[ST83] D.D. Sleator and R.E. Tarjan. A data structure for dynamic trees. *JCSS*, 26(3):362–391, 1983.

[ST85] D.D. Sleator and R.E. Tarjan. Self-adjusting binary search trees. *JACM*, 32(3):652–686, 1985.

[Tar85] R.E. Tarjan. Amortized computational complexity. *SIAM JOurnal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.