(M. Dietzfelbinger, 11. Dezember 2020)

3 Amortisierte Analyse

Wir betrachten hier ein Analyseproblem, das oft bei Datenstrukturen, mitunter auch in anderen algorithmischen Situationen auftritt. Angenommen, wir haben eine Datenstruktur, die einen Datentyp implementiert, z. B. einen Stack, eine Queue, ein Wörterbuch, eine Priority Queue. Bei solchen Datenstrukturen wird immer eine Folge $\mathrm{Op}_1, \ldots, \mathrm{Op}_n$ von Operationen ausgeführt. Diesen Operationen sind „Kosten“ $c_1, \ldots, c_n$ zugeordnet. (Meistens ist c_i eine vereinfachte Version der Rechenzeit für Operation Op_i, so geplant, dass die Rechenzeit $O(c_i)$ ist.) Die Gesamtkosten sind

$$C_n = c_1 + \cdots + c_n,$$

die Gesamtrechenzeit dann $O(C_n)$. Mit dem Ausdruck „Amortisierte Analyse“ ist die Bestimmung einer Abschätzung von C_n gemeint, oder Techniken hierfür.

Wir betrachten in diesem Kapitel ad-hoc-Methoden („*Aggregationsmethode*“) und zwei allgemeinere Techniken, die „*Bankkontomethode*“ und die „*Potenzialmethode*“. In späteren Abschnitten werden wir diese Methoden auf Datenstrukturen zur Implementierung von Priority Queues anwenden.

3.1 Die Aggregationsmethode

***Beispiel* 1:** Stack mit „*multipop*“. – Wir betrachten die bekannte Stack-Datenstruktur (Vorlesungen „Algorithmen und Programmierung“ und „Algorithmen und Datenstrukturen“). Diese besitzt die folgenden Operationen:

„*empty*“ zur Erzeugung eines leeren Stacks (die im Folgenden ignoriert wird),

„*push*“ zum Einfügen oben auf dem Stack,

„*pop*“ zum Entfernen und Ausgeben des obersten Elements (nur bei nichtleerem Stack),

„*isempty*“ zum Test, ob der Stack leer ist.

Alle diese Operationen haben Aufwand $O(1)$. Wir schreiben ihnen jeweils Kosten 1 zu. Nun kommt eine weitere Operation dazu:

„*multipop*(k)“ für $k \geq 1$ entfernt die obersten k Einträge und gibt sie aus. Falls die Stackhöhe p mindestens 1, aber kleiner als k ist, werden alle Einträge entfernt (kein Fehler!). Der Aufwand hierfür ist $O(\min\{k, p\})$; wir geben dieser Operation Kosten $\min\{k, p\} \geq 1$.

Behauptung: Die Kosten von n Operationen $\mathrm{Op}_1, \ldots, \mathrm{Op}_n$, startend mit einem leeren Stack, sind kleiner als $2n$.
Beweis: Wir können *isempty*-Operationen ignorieren, da sie Kosten 1 haben und den Stack nicht ändern. Wenn es n *push*-Operationen gibt, sind die Kosten n. Wenn es *pop*- und *multipop*-Operationen gibt, dann kann die Gesamtzahl der dadurch vom Stack entfernten Einträge nicht größer sein als die Anzahl der mit „*push*“ eingefügten Einträge: dies sind maximal $n-1$ viele. Nun gibt die Anzahl der entfernten Einträge ganz genau diese Kosten an. Einfügungen und Löschungen zusammen haben also Kosten nicht höher als $(n-1)+(n-1) < 2n$. □

Was haben wir gemacht? Wir haben die spezielle Struktur der Operationen benutzt, um mit einer geschickten Argumentation die Summe der Kosten zu beschränken. So etwas nennen wir „Ad-hoc-Methode“ oder „Aggregationsmethode“. (Natürlich ist dies eigentlich keine Methode, sondern man muss sich in jeder Situation wieder etwas Neues einfallen lassen.)

***Beispiel* 2:** Hochzählen eines Binärzählers.
Wir stellen uns vor, wir wollen auf einem Zähler, der Zahlen in Binärdarstellung darstellt, von 0 bis n zählen. Die Anzahl der Stellen des Zählers soll ausreichend sein. Am Anfang wird der Zähler (kostenlos) auf 0 gestellt. Dann wird n-mal eine Inkrementierungsoperation ausgeführt: $\mathrm{Op}_1, \ldots, \mathrm{Op}_n$, wobei Op_i die Erhöhung von $i-1$ auf i ist.

Kosten für eine Erhöhung: Die Anzahl der geänderten Bits.
(Von 110<u>0111</u> auf 110<u>1000</u> sind die Kosten 4; von 1101000 auf 1101001 sind sie 1.)

Wenn c_i die Kosten von Op_i sind, was ist dann $C_n = c_1 + \cdots + c_n$? Wir betrachten den Anfang der Entwicklung. In der folgenden Tabelle sind immer die Bits im Zähler unterstrichen, die sich im letzten Schritt geändert haben.

i	Zählerstand	Kosten
–	0	–
1	$\underline{1}$	1
2	$\underline{10}$	2
3	1$\underline{1}$	1
4	$\underline{100}$	3
5	10$\underline{1}$	1
6	1$\underline{10}$	2
7	11$\underline{1}$	1
8	$\underline{1000}$	4
9	100$\underline{1}$	1
10	10$\underline{10}$	2
11	101$\underline{1}$	1
12	1$\underline{100}$	3
13	110$\underline{1}$	1
14	11$\underline{10}$	2
15	111$\underline{1}$	1
16	$\underline{10000}$	5
17	1000$\underline{1}$	1

Die Kosten folgen einem regelmäßigen Muster, aber wie soll man sie summieren? Wir bemerken, dass jede unterstrichene Ziffer in der Tabelle genau 1 kostet. Nun schätzen wir die Anzahl der unterstrichenen Ziffern geschickt ab.

Für $\ell \geq 0$ sei u_ℓ die Anzahl der unterstrichenen Ziffern in Bitposition ℓ, die zur Zweierpotenz 2^ℓ gehört. Die letzte Ziffer (Position $\ell = 0$) ist in jeder Zeile unterstrichen, also ist $u_0 = n$. Die vorletzte Ziffer (Position $\ell = 1$) ist in jeder zweiten Zeile $(2, 4, 6, \dots)$ unterstrichen, beginnend mit der zweiten, also ist $u_1 \leq n/2$. Die drittletzte Ziffer ist in jeder vierten Zeile $(4, 8, 12, \dots)$ unterstrichen, beginnend mit der vierten, also ist $u_2 \leq n/4$. Allgemein: $u_\ell \leq n/2^\ell$, für $\ell = 0, 1, 2, \dots$. Damit:

$$\text{Gesamtkosten} \;<\; \sum_{\ell \geq 0} \frac{n}{2^\ell} = n \cdot 2 = 2n.$$

Beim Hochzählen eines Binärzählers von 0 auf n werden insgesamt weniger als $2n$ Bits gekippt.

Anmerkung: Später werden wir sehen, wie sich die Zahl der gekippten Bits ganz genau bestimmen lässt.

***Beispiel* 3:** Anzahl der Vergleiche beim Heapaufbau.
Wir betrachten binäre Heaps wie in der Vorlesung „Algorithmen und Datenstruktu-

ren“ (Bachelor)[1]. Die Frage ist, wie viele Vergleiche nötig sind, um aus einem beliebigen Array $A[1..n]$ einen binären Heap zu bauen. Die effizienteste Prozedur sieht wie folgt aus:

for `i` **from** $\lfloor n/2 \rfloor$ **downto** 1 **do** *bubble_down*(`i`, n).

Dabei führt *bubble_down*(i, n) auf jedem Niveau des Heaps unterhalb von Knoten i bis zu 2 Vergleiche aus.

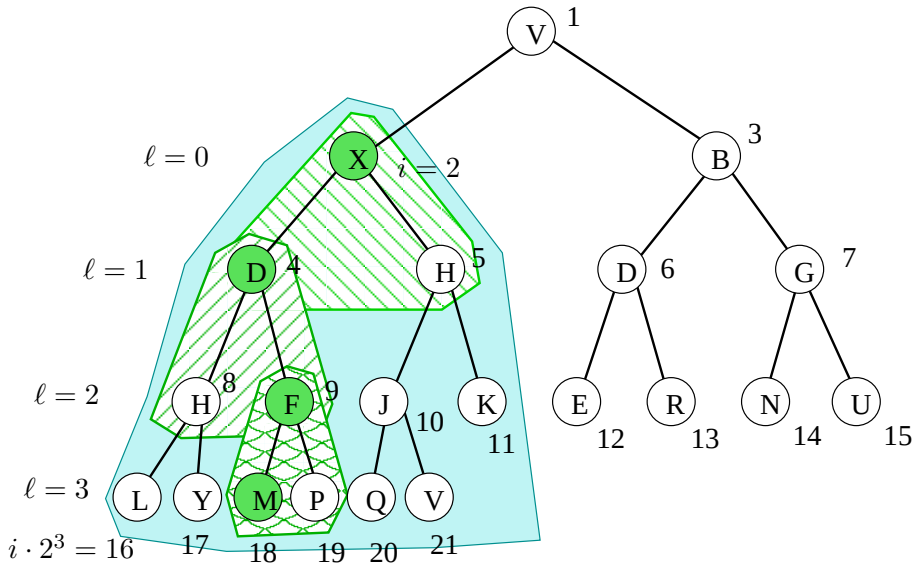

Abbildung 1: Aufruf *bubble_down*$(2, 21)$ führt zu dreimal 2 Vergleichen, auf Level $1, 2, 3$ unter Knoten $i = 2$. Ob Level ℓ noch nicht leer ist, wird durch die Ungleichung $i \cdot 2^\ell \leq n$ entschieden, weil der Knoten im Unterbaum mit Wurzel i, der am weitesten links sitzt, die Form $i \cdot 2^s$ hat (im Beispiel: $2 \cdot 2^3 = 16$).

Wir wollen die Gesamtzahl der Vergleiche nach oben durch ein C_n abschätzen. Wir betrachten den Aufruf *bubble_down*(i, n). Dabei werden entlang eines Wegs von Knoten i zu einem Blatt jeweils der Knoteninhalt mit dem kleineren Kind verglichen und eventuell vertauscht. Dies kostet zwei Vergleiche pro Level unterhalb von i. Der relevante Teil des Heaps für den Aufruf *bubble_down*(i, n) ist also der Teilbaum mit Wurzel i. (Beispiel: Abb. 1.) Level ℓ existiert in diesem Baum nur, wenn $i \cdot 2^\ell \leq n$

[1] `https://www.tu-ilmenau.de/fileadmin/public/iti/Lehre/AuD/SS16/AuD-Kap-6-statisch.pdf`, ab Folie 79. Siehe insbesondere Folien 82–85.

ist, weil Knoten $i \cdot 2^\ell$ der kleinste Knoten auf Level ℓ im Teilbaum unter i ist und n der größte Knoten im gesamten Heap ist. Damit:

$$C_n \leq \sum_{1 \leq i \leq \lfloor n/2 \rfloor} \; \sum_{\ell \geq 1 \colon i \cdot 2^\ell \leq n} 2.$$

Der Trick ist hier die Vertauschung der Summationsreihenfolge. Dabei können wir die ℓ-Summe auch unbegrenzt laufen lassen:

$$C_n \leq 2 \cdot \sum_{\ell \geq 1} \; \sum_{1 \leq i \leq n/2^\ell} 1 = 2 \cdot \sum_{\ell \geq 1} \lfloor n/2^\ell \rfloor < 2 \cdot \sum_{\ell \geq 1} n/2^\ell = 2n.$$

(Die letzte Gleichheit folgt daraus, dass $\sum_{\ell \geq 1} 2^{-\ell} = 1$ ist.)

Fazit: Um aus einem Array $A[1..n]$ einen Heap zu machen, sind weniger als $2n$ Schlüsselvergleiche nötig.

3.2 Die Bankkontomethode

Anschauliche Grundidee bei dieser Methode ist, dass der Benutzer für jede Operation der Datenstruktur eine pauschale Gebühr bezahlen muss. Die gezahlten Beiträge werden teilweise zum Begleichen der Kosten benutzt und teilweise zu einem Guthaben „angespart“, das dann später zum Bezahlen teurer Operationen benutzt werden kann. Die pauschale Gebühr für Op_i heißt „amortisierte Kosten“ a_i. Diese muss (geschickt) festgelegt werden. Bei der Ausführung von Operation Op_i gibt es dann zwei Fälle: Wenn $a_i \geq c_i$ für die echten Kosten c_i, wird der Überschuss $a_i - c_i \geq 0$ auf das Bankkonto eingezahlt. Wenn $a_i < c_i$ ist, bestreiten wir den fehlenden Betrag $c_i - a_i$ aus dem Guthaben. Dabei ist Schuldenmachen streng verboten: Es muss auf dem Konto immer ein Guthaben ≥ 0 vorhanden sein! Je nach Anwendung kann man zur Veranschaulichung das Guthaben auf Komponenten der Datenstruktur verteilen. Wir wenden die Methode auf Beispiel 1 an, den Stack mit „*multipop*“.

Operation	a_i	c_i	Bemerkung
isempty	1	1	keine Änderung des Guthabens
push	2	1	Einzahlung: 1
pop	0	1	Abhebung: 1
multipop(k)	0	$\min\{k, p\}$	Abhebung: $\min\{k, p\}$

Bei jeder Einfügung steigt das Guthaben um $a_i - c_i = 2 - 1 = 1$. Wir können uns daher vorstellen, dass jeder Eintrag ein Guthaben von 1 besitzt. Am Anfang ist das Guthaben 0, und der Stack ist leer. Der eingezahlte Wert 1 bei „*push*(x)“ gehört zu

diesem Eintrag x. Bei „*pop*“ wird ein Element y entfernt, die Kosten 1 werden durch das y zugeordnete Guthaben abgedeckt. Bei „*multipop*(k)“ werden $\min\{k, p\}$ Einträge entfernt, das Guthaben dieser Einträge deckt die Kosten genau ab.

Wir definieren das Guthaben nach Schritt i als B_i und die Stackhöhe nach Schritt i als p_i.

Lemma 3.2.1. *$B_i = p_i$, für $i \geq 0$. (Daraus folgt: $B_i \geq 0$ für alle $i \geq 0$.)*

Die Aussage folgt eigentlich direkt daraus, wie das Guthaben auf die aktuell vorhandenen Stackeinträge verteilt ist. Im Allgemeinen beweist man eine solche die Behauptung durch Induktion über i.

Aus $B_n = p_n \geq 0$ folgt dann:

$$0 \leq B_n = \sum_{1 \leq i \leq n} (a_i - c_i) \ , \ \text{also} \quad C_n = \sum_{1 \leq i \leq n} c_i \leq \sum_{1 \leq i \leq n} a_i.$$

Das bedeutet, dass die (echten!) Gesamtkosten C_n nicht größer als $2 \cdot$ (Anzahl der *push*-Operationen) sein können.

Wir formulieren das **Rezept „Bankkontomethode“** allgemein:

(i) Ordne jeder Operation Op_i (geschickt) amortisierte Kosten a_i zu.
(Meist hängen diese nicht von i ab, sondern nur von Op_i und eventuell von der Größe der Datenstruktur.)

(ii) Definiere $B_0 := 0$ und $B_i := B_{i-1} + (a_i - c_i)$, für $i \geq 1$.
(Die Veränderung $a_i - c_i$ des Kontostands kann positiv oder negativ oder gleich 0 sein.)

(iii) Formuliere eine Induktionsbehauptung (IB_i) über B_i, aus der folgt, dass stets $B_i \geq 0$ gilt.
(Die Wahl von (IB_i) ist der entscheidende und schwierige Schritt. Die Behauptung „$B_i \geq 0$“ genügt nicht!)

(iv) Beweise (IB_i).
(Meist durch Induktion, mit einer Fallunterscheidung darüber, was bei Op_i passiert.)

(v) Aus $B_n \geq 0$ und $B_0 = 0$ und $B_i = B_{i-1} + (a_i - c_i)$, für $1 \leq i \leq n$, folgt unmittelbar $\sum_{i=1}^{n}(a_i - c_i) \geq 0$, also

$$C_n = c_1 + \cdots + c_n \leq a_1 + \cdots + a_n.$$

Daraus erhält man die gewünschte Schranke für C_n.

***Beispiel* 2:** Hochzählen eines Binärzählers, Bankkontomethode.
(i) Wir definieren:

$$a_i := 2. \qquad \text{(„Clevere“ Idee!)}$$

(ii) $B_0 := 0$ und $B_i := B_{i-1} + (a_i - c_i)$, für $i \geq 1$.

(iii) (IB_i) $B_i = |\text{bin}(i)|_1 =$ Anzahl der 1-Ziffern in $\text{bin}(i)$. (Dies ist der „clevere“ Teil!)
(Beispiel: $\text{bin}(19) = 10011$, also muss $B_{19} = 3$ sein. Man kann sich das Ansparen so vorstellen: Beim Hochzählen wird genau eine neue 1 erzeugt, also eine 0 auf 1 gekippt. Die amortisierten Kosten bezahlen für dieses Kippen *und* für das spätere Zurückkippen dieser Stelle auf 0. Solange die Ziffer 1 Bestand hat, ist ihr das Guthaben von 1 Euro zugeordnet.)

(iv) Beweis von (IB_i) durch Induktion über i:
I. A.: $B_0 = 0$, und die Anzahl der 1-Ziffern in $\text{bin}(0)$ ist 0.
I. V.: (IB_{i-1}) stimmt.
I. S.: Der Zähler wird von $i-1$ auf i erhöht.

$$\text{bin}(i-1) = \underbrace{*\ldots*}_{z \text{ Einsen}}\, 0\, \underbrace{1\ldots1}_{\ell},$$
$$\text{bin}(i) = \underbrace{*\ldots*}_{z \text{ Einsen}}\, 1\, \underbrace{0\ldots0}_{\ell}.$$

Nach I. V. gilt $B_{i-1} = z + \ell$. Die echten Kosten c_i sind die Anzahl der gekippten Bits, also $\ell + 1$. Damit:

$$B_i = B_{i-1} + a_i - c_i = (z + \ell) + 2 - (\ell + 1) = z + 1,$$

und das ist gerade die Anzahl der Einsen in $\text{bin}(i)$, wie behauptet.

(v) Da die Anzahl der Einsen in $\text{bin}(i)$ stets nichtnegativ ist (genauer gesagt: sie ist positiv für $i > 0$), folgt

$$C_n = c_1 + \cdots + c_n \leq a_1 + \cdots + a_n = 2n.$$

Man kann die amortisierten Kosten sogar genau angeben:

$$C_n = c_1 + \cdots + c_n = a_1 + \cdots + a_n - |\text{bin}(n)|_1 = 2n - |\text{bin}(n)|_1.$$

(Beispiele: $C_{16} = 31$, $C_{17} = 32 = 2 \cdot 17 - 2$, $C_{18} = 34 = 2 \cdot 18 - 2$, usw.)

Um die Schlagkraft der Methode zu demonstrieren, betrachten wir noch ein etwas komplizierteres Beispiel:

***Beispiel* 3:** Stack mit Verdoppelungs- und Halbierungsstrategie.
Wir betrachten den Datentyp Stack mit den gewöhnlichen Operationen *empty*, *push*,

pop, *isempty*, *top*. Der Stack soll mit Hilfe eines Arrays $\texttt{A}[1..m]$ und eines Pegels $\texttt{p}$ dargestellt werden. Der Pegelstand p (in $\texttt{p}$) gibt die aktuelle Anzahl der Einträge an; der Stack besteht (von oben nach unten) aus den Einträgen $\texttt{A}[p]$, $\texttt{A}[p-1]$, ..., $\texttt{A}[1]$. Ein Problem entsteht dadurch, dass man beim Anlegen des Arrays nicht weiß, wie hoch der Stack wird. In der AuD-Vorlesung wurde schon die *Verdopplungsstrategie* behandelt und analysiert, die es gestattet, die Datenstruktur gegebenenfalls zu vergrößern, ohne dass die Kosten mehr als linear in der Anzahl der Operationen sind. Wir erweitern die Fragestellung noch. Wie kann man gegebenenfalls auch wieder Speicher freigeben, wenn der Stack wieder viel kleiner wird, so dass der zu einem beliebigen Zeitpunkt beanspruchte Platz nicht viel größer als die Anzahl der aktuellen Einträge ist? Wir geben eine Anfangs- und Mindestgröße m_0 für das Array vor.

Op_i = *empty*: Lege Array $\texttt{A}[1..m_0]$ an, setze $\texttt{p} \leftarrow 0$.

Op_i = *isempty*, *top*: Ausführung offensichtlich; echte Kosten: $c_i = 1$.

$\mathrm{Op}_i = \mathit{push}(x)$:

1. Fall: Wenn $\texttt{p} < m$, erhöhe $\texttt{p}$ um 1 und speichere x in $\texttt{A[p]}$.
Echte Kosten: $c_i = 1$.

2. Fall: Wenn $\texttt{p} = m$: **„Verdopple."** Das heißt: $\texttt{A}[1..m]$ wird durch ein doppelt so großes Array ersetzt, wie folgt. Ein Array $\texttt{AA}[1..2m]$ wird alloziert; die m Einträge werden von $\texttt{A}$ nach $\texttt{AA}$ kopiert; $\texttt{A}$ wird freigegeben; $\texttt{AA}$ wird in $\texttt{A}$ umbenannt. Nun ist Platz für das Einfügen von x (wie im 1. Fall).
Echte Kosten: $c_i = m + 1$ (für Umspeichern und Einfügen von x).

Op_i = *pop*:

1. Fall: Wenn $m = m_0$ oder $\frac{1}{4}m < \texttt{p}$, gib $\texttt{A[p]}$ aus und verringere $\texttt{p}$ um 1. Echte Kosten: $c_i = 1$.

2. Fall: Wenn $\texttt{p} = \frac{1}{4}m$ und $m > m_0$: **„Halbiere."** Das heißt: $\texttt{A}[1..m]$ wird durch ein halb so großes Array ersetzt, wie folgt. Ein Array $\texttt{AA}[1..\frac{1}{2}m]$ wird alloziert; die $m/4$ Einträge werden von $\texttt{A}$ nach $\texttt{AA}$ kopiert; $\texttt{A}$ wird freigegeben; $\texttt{AA}$ wird in $\texttt{A}$ umbenannt. Nun ist $\texttt{A}$ genau zur Hälfte gefüllt, und wir verfahren weiter wie im 1. Fall.
Echte Kosten: $c_i = \frac{1}{4}m + 1$ (für Umspeichern und Entfernen des obersten Eintrags).

Eine kurze Bemerkung zu der naheliegenden Frage, weshalb man nicht halbiert, wenn der Füllstand unter $\frac{1}{2}m$ fällt: In diesem Fall könnten abwechselnde *pop*- und *push*-Operationen zu aufeinanderfolgenden Verdopplungen und Halbierungen führen, die viel zu teuer wären.

Beispiel: Wir überlegen kurz, wie der Auf- und Abbau der Arrays abläuft, wenn man mit $m_0 = 100$ startet, zunächst 2500 Einträge einfügt und dann 2490 wieder löscht. Einfügen der ersten 100 Einträge kostet 100. Dann wird verdoppelt, mit Kosten 101. Einfügen der nächsten 99 Einträge kostet 99, die folgende Verdopplung 201, und so weiter. Durch weitere Verdopplungen wächst das Array auf Größen 400, 800, 1600, 3200 an. Die Kosten: 199 (einzeln), 401 (Verdopplung), 399 (einzeln), 801 (Verdopplung), 799 (einzeln), 1601 (Verdopplung), 899 (einzeln). Insgesamt: $2500 + 100 + 200 + 400 + 800 + 1600 = 2500 + 3100 = 5600$. Nun wird gelöscht. Nach 1700 Einzel-Löschungen ist die Anzahl der Einträge auf 800 gesunken. Das Array wird auf Größe 1600 halbiert, mit Kosten 801. Danach erfolgt die Halbierung auf 800 (Kosten 401), auf 400 (Kosten 201) auf 200 (Kosten 101) und auf $m_0 = 100$ (Kosten 51). Weitere Löschungen verkleinern das Array nicht mehr. Die Gesamtkosten für die Löschungen sind $2490 + 800 + 400 + 200 + 100 = 2490 + 1500 = 3990$.

Bei einem Ablauf mit einer so einfachen Struktur sieht man leicht, dass die Gesamtkosten linear in der Anzahl der Operationen sind. Aber wie sieht es bei Abläufen aus, bei denen Einfügungen und Löschungen wild durcheinander auftreten? Wir benutzen die Bankkontomethode.

(i) Wir definieren amortisierte Kosten wie folgt. Dabei beschränken wir uns auf die Operationen *push*(x) und *pop*. (Die anderen Operationen haben konstante echte Kosten und ändern die Datenstruktur nicht.)

Operation	a_i	c_i	Bemerkung
empty	1	1	wird nicht weiter betrachtet
top	1	1	wird nicht weiter betrachtet
push	3	1. Fall: 1; 2. Fall: $m + 1$	1. Fall: Einz.; 2. Fall: Abhebung
pop	2	1. Fall: 1; 2. Fall: $\frac{1}{4}m + 1$	1. Fall: Einz.; 2. Fall: Abhebung

(ii) $B_0 := 0$ und $B_i := B_{i-1} + (a_i - c_i)$, für $i \geq 1$.

(iii) Es sei m_i die Arraygröße nach Schritt i und p_i der Pegelstand nach Schritt i.

(IB$_i$) (a) Wenn $p_i \geq \frac{1}{2}m_i$, dann gilt $B_i \geq 2(p_i - \frac{1}{2}m_i)$ (≥ 0, s. Abb. 2);
(b) wenn $m_i > m_0 \wedge p_i < \frac{1}{2}m_i$, dann gilt $B_i \geq \frac{1}{2}m_i - p_i$ (≥ 0, s. Abb. 3);
(c) wenn $m = m_0$ und $p_i < \frac{1}{2}m$, gilt $B_i \geq 0$ (s. Abb. 4).

Idee: Wenn $p \geq \frac{1}{2}m$, wird Guthaben angespart, um für eine zukünftige teure Verdopplung zu bezahlen. (Man kann sich vorstellen, dass auf jedem Eintrag, der oberhalb von Level $\frac{1}{2}m$ liegt, ein Guthaben von 2 Euro liegt. Weiteres Guthaben ignorieren wir. Wenn der Pegelstand $\frac{1}{2}m_i$ ist, wird nur verlangt, dass das Guthaben nichtnegativ ist.) Wenn $p < \frac{1}{2}m$, wird ebenfalls Guthaben aufgebaut, um für eine zukünftige teure Halbierung zu bezahlen. (Jedem freien Platz unter Level $\frac{1}{2}m$ ist ein Guthaben von 1 Euro zugeordnet, weiteres Guthaben wird ignoriert.) Dies entfällt, wenn $m = m_0$ ist, also insbesondere in der Anfangsphase, wenn das Array noch fast ganz leer ist.

(iv) Beweis von (IB$_i$) durch Induktion über i:

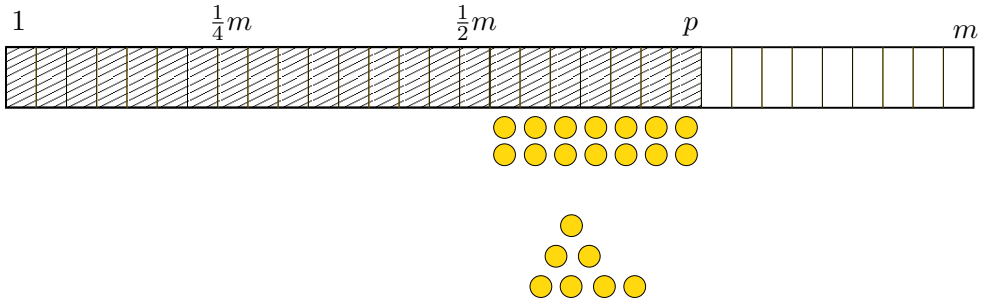

Abbildung 2: (a) Wenn der Pegelstand p größer oder gleich $\frac{1}{2}m$ ist, verlangt man: Kontostand $B \geq 2(p - \frac{1}{2}m)$: 2 Euro für jede besetzte Stelle oberhalb von $\frac{1}{2}m$.

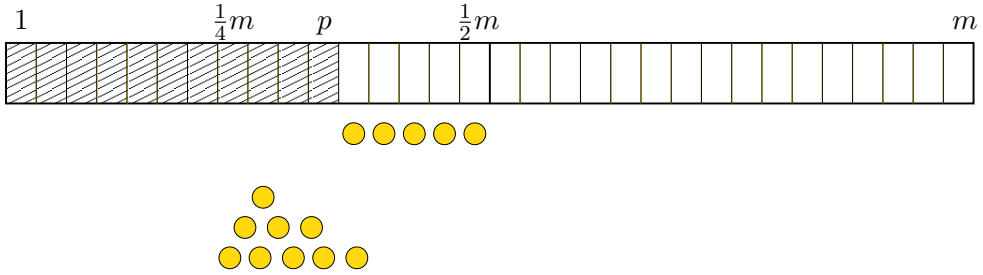

Abbildung 3: (b) Wenn der Pegelstand p zwischen $\frac{1}{4}m$ und $\frac{1}{2}m$ liegt und $m > m_0$ ist, verlangt man: Kontostand $B \geq \frac{1}{2}m - p$: ein Euro für jede Fehlstelle unterhalb von $\frac{1}{2}m$.

I. A.: $B_0 = 0$, und $p_0 = 0$ und m_0 ist der Startwert. (IB_0) ist trivialerweise erfüllt.
I. V.: (IB_{i-1}) stimmt.
I. S.: Op_i wird aufgeführt. Es gibt zwei Fälle:

Falls $\text{Op}_i = \mathit{push}(x)$:
Wenn $p_{i-1} < \frac{1}{2}m_{i-1}$, gilt $c_i = 1$ und $a_i = 3$, also $B_i > B_{i-1}$. Die Anzahl der freien Plätze unter Level $\frac{1}{2}m_{i-1}$ sinkt um 1, also wird die Anforderung an das Guthaben geringer. Auch wenn $p_i = \frac{1}{2}m_{i-1}$, gilt $B_i \geq 1 > 0$. Damit ist (IB_i) in allen möglichen Fällen gesichert.
Wenn $\frac{1}{2}m_{i-1} \leq p_{i-1} < m_{i-1}$, dann wächst das Guthaben um $a_i - c_i = 2$ und $2(p - \frac{1}{2}m)$ wächst ebenfalls um 2. Damit gilt auch Bedingung (b) weiter. (Wir legen die beiden eingezahlten Euros neben den neuen Eintrag x.)
Wenn schließlich $p_{i-1} = m_{i-1}$, dann erfolgt die Verdoppelung auf $m_i = 2m_{i-1}$. Das Guthaben B_{i-1} ist mindestens $2(p_{i-1} - \frac{1}{2}m_{i-1}) = m_{i-1}$. Dies reicht gerade, um die

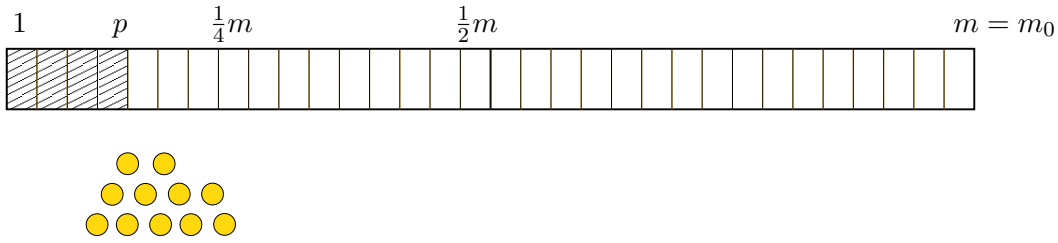

Abbildung 4: (c) Wenn $m = m_0$ und der Pegelstand p kleiner als $\frac{1}{2}m$ ist, verlangt man nur: Kontostand $B \geq 0$.

Kosten m_{i-1} des Umbaus zu decken. Die Einfügung hat noch Kosten 1, die amortisierten Kosten sind $a_i = 3$. Bleiben genau 2 Euros übrig, die wir neben den neuen Eintrag x legen. In Zahlen:

$$\begin{aligned} B_i &= B_{i-1} + (a_i - c_i) \overset{\text{I.V.(a)}}{\geq} 2(p_{i-1} - \frac{1}{2}m_{i-1}) + (3 - (m_{i-1} + 1)) \\ &= 2(m_{i-1} - \frac{1}{2}m_{i-1}) + (3 - (m_{i-1} + 1)) = 2 \\ &= 2(p_i - \frac{1}{2}m_i). \end{aligned}$$

Also gilt (IB_i).

Falls $\text{Op}_i = \mathit{pop}$:
Wenn $\frac{1}{2}m_{i-1} < p_{i-1} \leq m_{i-1}$, dann wächst das Guthaben um $a_i - c_i = 1$, aber die Anforderung in (a) wird schwächer, weil die Anzahl der Einträge oberhalb von Level $\frac{1}{2}m_{i-1}$ sinkt.
Wenn $\frac{1}{4}m_{i-1} < p_{i-1} \leq \frac{1}{2}m_{i-1}$ und $m_{i-1} > m_0$, dann wächst das Guthaben um $a_i - c_i = 1$, und $\frac{1}{2}m - p$ wächst ebenfalls um 1. Anschaulich: Wir legen den neu eingezahlten Euro neben die neu freigewordene Stelle.
Wenn $\frac{1}{4}m_{i-1} < p_{i-1} \leq \frac{1}{2}m_{i-1}$ und $m_{i-1} = m_0$, dann gibt es nur die Anforderung $B_i \geq 0$, die aber aus der I. V. folgt.
Wenn schließlich $m_{i-1} > m_0$ und $p_{i-1} = \frac{1}{4}m_{i-1}$, erfolgt die Halbierung, mit $c_i = \frac{1}{4}m_{i-1} + 1$ und $a_i = 2$. Diese hat Kosten $\frac{1}{4}m_{i-1}$. Diese Kosten werden durch die Eurostücke gedeckt, die neben den leeren Plätzen zwischen Level $\frac{1}{4}m_{i-1}$ und $\frac{1}{2}m_{i-1}$ liegen. Das Entfernen des nächsten Eintrags kostet noch 1, aber mit $a_i = 2$ bleibt

noch ein Euro für die neue leere Stelle unter $\frac{1}{2}m_i$ übrig. Rechnerisch:

$$\begin{aligned} B_i &= B_{i-1} + (a_i - c_i) \overset{\text{I.V.(b)}}{\geq} (\frac{1}{2}m_{i-1} - p_{i-1}) + (2 - (\frac{1}{4}m_{i-1} + 1)) \\ &= 1 = \frac{1}{2}m_i - p_i. \end{aligned}$$

Bei $m_{i-1} = m_0$ fallen keine Halbierungskosten an, das Guthaben bleibt positiv. Also gilt (IB_i).

(v) Da die in (IB_i) angegebenen unteren Schranken für B_i stets nichtnegativ sind, folgt

$$C_n = c_1 + \cdots + c_n \leq a_1 + \cdots + a_n \leq 3n_{push} + 2n_{pop},$$

wobei n_{push} und n_{pop} die Anzahl der *push*- bzw. *pop*-Operationen in der Folge ist.

Satz 3.2.2. *Die Arrayimplementierung von Stacks mit Verdoppelung und Halbierung hat folgende Eigenschaften:*

(a) *n Operationen haben insgesamt Zeitaufwand $O(n)$.*

(b) *Wenn p Einträge in der Datenstruktur gespeichert sind, ist das momentan benutzte Array nicht größer als* $\max\{m_0, 4p\}$.

3.3 Die Potenzialmethode

Auch die Potenzialmethode ist ein allgemeiner Ansatz, den man bei amortisierten Analysen in seinem Werkzeugkoffer haben sollte. Hier ist die Idee, jedem „inneren Zustand“ D der Datenstruktur eine (nichtnegative) reelle Zahl, genannt das Potenzial $\Phi(D)$ von D, zuzuordnen.[2] Wenn man möchte, kann man sich $\Phi(D)$ als eine Art in D gespeicherte Energie vorstellen. (Wie wir gleich sehen werden, handelt es sich hier um „monetäre Energie“: die Fähigkeit, Kosten zu übernehmen ...)

***Beispiel* 1:** Stack mit „*multipop*“. – (i) Wir ordnen einem Stack D mit momentan p Einträgen das Potenzial $\Phi(D) = p$ zu.

Nach Festlegung des Potenzials geht die Potenzialmethode recht schematisch vor.

(ii) Die amortisierten Kosten einer Operation Op, die die Datenstruktur von Zustand D in Zustand D' transformiert, mit echten Kosten c, hat *amortisierte Kosten*

$$a := c + (\Phi(D') - \Phi(D)) = c - (\Phi(D) - \Phi(D')). \tag{1}$$

[2] „Φ“ heißt „Phi“ und wird „Fi“ ausgesprochen.

Wir interpretieren: Wenn $\Phi(D') > \Phi(D)$, ist die Potenzialdifferenz positiv, und für das Erreichen des höheren (Energie-)Niveaus muss man zusätzlich zu c „Potenzialkosten“ $\Phi(D') - \Phi(D)$ aufwenden. Wenn dagegen $\Phi(D') < \Phi(D)$, geht das Potenzial nach unten; die Potenzialdifferenz $\Phi(D) - \Phi(D')$ kann benutzt werden, um die echten Kosten c zum Teil oder ganz zu bestreiten.

(iii) Nun werden für jede Operation Op die amortisierten Kosten a_{Op} durch eine Schranke K_{Op} nach oben abgeschätzt. Manchmal hängt K_{Op} von der Größe der Datenstruktur ab; manchmal ist es eine Konstante.

In unserem Beispiel sieht dies so aus:

1. Fall: Op = *isempty* oder Op = *top*. – Da sich die Datenstruktur nicht verändert, ist die Potenzialdifferenz 0. Also ist $a = c = 1$.

2. Fall: Op = *push*(x). – Die Stackhöhe wächst um 1, d. h. $\Phi(D') - \Phi(D) = 1$. Damit ist $a = c + 1 = 2$.

3. Fall: Op = *multipop*(k). – Die Stackhöhe sinkt um $\min\{k, p\}$, d. h. $\Phi(D') - \Phi(D) = \min\{k, p\}$. Die Kosten der Operation sind $c = \min\{k, p\}$. Daher ist $a = c - (\Phi(D) - \Phi(D')) = 0$.

4. Fall: Op = *pop*. – Wie *multipop*(1).

Wir erhalten also in diesem Fall, dass die amortisierten Kosten für ein *push* 2 betragen, für *pop* und *multipop*(k) dagegen 0.

Nun betrachten wir eine Operationenfolge $\mathrm{Op}_0, \mathrm{Op}_1, \ldots, \mathrm{Op}_n$, die eine Datenstruktur D_0 erzeugt und in n Runden zu $D_1, \ldots, D_n$ verändert. Die echten Kosten der Operation Op_i, die D_{i-1} in D_i transformiert, seien c_i. Die amortisierten Kosten dieser Operation sind dann

$$a_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

Die folgende Behauptung ist zentral für die Anwendung der Potenzialmethode.

Lemma 3.3.1. *Wenn Φ die Bedingung $\Phi(D_n) \geq \Phi(D_0)$ erfüllt, dann gilt $C_n = c_1 + \cdots + c_n \leq a_1 + \cdots + a_n$.*

(Insbesondere haben wir: Wenn $a_i \leq K$ für eine Konstante K ist, dann ist $C_n \leq Kn$. Die Bedingung $\Phi(D_n) \geq \Phi(D_0)$ wird oft gezeigt, indem man feststellt, dass $\Phi(D_0) = 0$ und $\Phi(D) \geq 0$ für beliebige Zustände D gilt.)

Beweis:

$$\begin{aligned} C_n &= c_1 + \cdots + c_n \\ &= \sum_{1 \le i \le n} (a_i + (\Phi(D_{i-1}) - \Phi(D_i))) \\ &= \sum_{1 \le i \le n} a_i + \sum_{1 \le i \le n} (\Phi(D_{i-1}) - \Phi(D_i)) \\ &\overset{(*)}{=} \sum_{1 \le i \le n} a_i + \Phi(D_0) - \Phi(D_n) \\ &\le \sum_{1 \le i \le n} a_i. \end{aligned}$$

Die Gleichheit $(*)$ gilt, weil sich positive und negative Terme ausgleichen („Ziehharmonikasumme“ oder „Teleskopsumme“).

(iv) Es gilt $\Phi(D_0) = 0$, weil D_0 der leere Stack ist. Weiter gilt $\Phi(D_n) \ge 0$, weil die Stackhöhe p nie negativ sein kann. Damit sind die Bedingungen von Lemma 3.3.1 erfüllt. Alle a_i sind ≤ 2, also folgt: $C_n \le 2n$.

Wir formulieren die ***Potenzialmethode*** allgemein:

(i) Ordne jedem Zustand D der Datenstruktur ein Potenzial $\Phi(D) \ge 0$ zu.

(ii) Für eine Operation Op, Zustand D in Zustand D' transformiert und dabei echte Kosten $c = c_{\mathrm{Op}}(D, D')$ hat, definiere „*amortisierte Kosten*“

$$a := a_{\mathrm{Op}}(D, D') := c + \Phi(D') - \Phi(D).$$

(iii) Schätze $a = a_{\mathrm{Op}}(D, D')$ nach oben ab, für jede Operation Op.

(iv) Gegeben sei eine Operationenfolge $\mathrm{Op}_1, \ldots, \mathrm{Op}_n$, mit echten Kosten $c_1, \ldots, c_n$ und amortisierten Kosten $a_1, \ldots, a_n$. Zeige, dass $\Phi(D_n) \ge \Phi(D_0)$.
(Sehr oft: $\Phi(D_0) = 0$ und $\Phi(D) \ge 0$ für alle D.)

(v) Folgere durch Anwendung von Lemma 3.3.1: $C_n = c_1 + \cdots + c_n \le a_1 + \cdots + a_n$.

***Beispiel* 2:** Hochzählen eines Binärzählers.

Wir wenden die ***Potenzialmethode*** an.

(i) Der Zustand der Datenstruktur ist hier die Binärdarstellung $\mathrm{bin}(i)$ des Zählerstandes i. Wir definieren:

$$\Phi(\mathrm{bin}(i)) := \text{ Anzahl der Einsen in } \mathrm{bin}(i).$$

(Beachte: Die Potenzialfunktion geeignet zu wählen ist der entscheidende Punkt.)

(ii) Die Operationenfolge $\mathrm{Op}_1, \ldots, \mathrm{Op}_n$ ist fest gegeben: Op_i zählt von $i-1$ auf i hoch. Kosten c_i: Anzahl der gekippten Bits. Definiere amortisierte Kosten

$$a_i := c_i + \Phi(\mathrm{bin}(i)) - \Phi(\mathrm{bin}(i-1)), \text{ für } 1 \leq i \leq n.$$

(iii) Wir berechnen a_i. Man vergleiche das Bild in Abschnitt 3.2, um folgende Rechnung zu rechtfertigen. Wenn $i-1$ auf ℓ Einsen endet und insgesamt $z+\ell$ Einsen enthält, dann ist

$$a_i = c_i + \Phi(\mathrm{bin}(i)) - \Phi(\mathrm{bin}(i-1)) = (\ell+1) + (z+1) - (z+\ell) = 2.$$

(iv) $\Phi(\mathrm{bin}(n)) \geq \Phi(\mathrm{bin}(0))$ gilt, weil $\Phi(\mathrm{bin}(0)) = 0$ ist und $\Phi(\mathrm{bin}(i)) \geq 0$ für alle i, einfach nach der Definition.

(v) Mit Lemma 3.3.1 folgt: $C_n = c_1 + \cdots + c_n \leq a_1 + \cdots + a_n = 2n$.

Bemerkung: Wenn man die Rechnungen in (i)–(iii) und im Beweis von Lemma 3.3.1 genau nachvollzieht, sieht man, dass an allen Stellen außer in (v) Gleichheit gilt. Wir können also genauer schreiben:

$$C_n = \sum_{1 \leq i \leq n} a_i + \Phi(\mathrm{bin}(0)) - \Phi(\mathrm{bin}(n)) = 2n - \#(\text{Einsen in } \mathrm{bin}(n)).$$

Diese Formel gibt also die exakte Anzahl von gekippten Bits beim Zählen bis n an. (*Beispiel*: Nach der Tabelle auf Seite 3 ist $C_5 = 1+2+1+3+1 = 8$; andererseits ist $2 \cdot 5 - \#(\text{Einsen in } \mathrm{bin}(5)) = 10 - 2 = 8$. Magie! Wenn man diesen Trick verstanden hat, lässt sich auch die Anzahl der Bits leicht angeben, die beim Hochzählen von k nach n gekippt werden: $2(n-k) + \#(\text{Einsen in } \mathrm{bin}(k)) - \#(\text{Einsen in } \mathrm{bin}(n))$. Man beachte, dass dies u. U. auch etwas größer als $2(n-k)$ sein kann.

***Beispiel* 3:** Stack mit Verdoppelungs- und Halbierungsstrategie.
Wir betrachten die gleiche Situation wie oben, aber analysieren sie mit der Potenzialmethode. Anfangs ist der Stack leer ($p_0 = 0$), das Array hat Größe m_0.

(i) Der Zustand der Datenstruktur D ist hier durch die Arraygröße m und den Pegelstand p gegeben (kurz: der Zustand ist $D = (m, p)$). Wir definieren das Potenzial:

$$\Phi(D) := \begin{cases} 0 & \text{, falls } m = m_0 \text{ und } p < \frac{1}{2}m; \\ \frac{1}{2}m - p & \text{, falls } m > m_0 \text{ und } \frac{1}{4}m \leq p < \frac{1}{2}m; \\ 2(p - \frac{1}{2}m) & \text{, falls } \frac{1}{2}m \leq p \leq m. \end{cases}$$

Wieder gilt: Dass man diese Werte geschickt wählt, ist für das Gelingen der Analyse entscheidend.

(ii) Für den Übergang von D zu D' mit echten Kosten c definieren wir amortisierte Kosten

$$a := a_{\mathrm{Op}}(D, D') = c + \Phi(D') - \Phi(D).$$

(iii) Wir berechnen die amortisierten Kosten a, wenn Op von Zustand (m, p) zu Zustand (m', p') führt und dabei echte Kosten c anfallen. Dafür gibt es eine Reihe von Fällen zu betrachten. Wir bemerken zunächst, dass für $m = m_0$ und $p, p' \leq \frac{1}{2}m$ das Potenzial 0 ist und weder Verdopplung noch Halbierung vorkommen, also $a = c = 1$ gilt. Ab hier nehmen wir an, dass dieser Sonderfall nicht vorliegt.

1. Fall: Op = *push*.

Fall 1a: $\frac{1}{4}m \leq p < \frac{1}{2}m$ (und $m > m_0$). –
Dann gilt $m' = m$ und $p' = p + 1$ und $c = 1$, also

$$a = c + \Phi((m', p')) - \Phi((m, p)) = 1 + (\tfrac{1}{2}m' - p') - (\tfrac{1}{2}m - p) = 0.$$

Fall 1b: $\frac{1}{2}m \leq p < m$. –
Dann gilt $m' = m$ und $p' = p + 1$ und $c = 1$, also

$$a = c + \Phi((m', p')) - \Phi((m, p)) = 1 + 2(p' - \tfrac{1}{2}m') - 2(p - \tfrac{1}{2}m) = 3.$$

Fall 1c: $p = m$. –
Dann gilt $m' = 2m$ und $p' = p + 1$ und $c = m + 1$, also

$$\begin{aligned} a &= c + \Phi((m', p')) - \Phi((m, p)) \\ &= (m + 1) + 2(p' - \tfrac{1}{2}m') - 2(p - \tfrac{1}{2}m) \\ &= (m + 1) + 2((m + 1) - m) - 2(m - \tfrac{1}{2}m) = 3. \end{aligned}$$

Im Fall 1c ist besonders schön zu sehen, wie der Potenzialunterschied benutzt wird, um für eine teure Operation zu bezahlen. Wenn Op_i eine *push*-Operation ist, sind die amortisierten Kosten durch 3 beschränkt.

2. Fall: Op = *pop*.

Fall 2a: $\frac{1}{2}m < p \leq m$. –
Dann gilt $m' = m$ und $p' = p - 1$ und $c = 1$, also

$$a = c + \Phi((m', p')) - \Phi((m, p)) = 1 + 2(p' - \tfrac{1}{2}m') - 2(p - \tfrac{1}{2}m) = -1.$$

Fall 2b: $\frac{1}{4}m < p \leq \frac{1}{2}m$ (und $m > m_0$). –
Dann gilt $m' = m$ und $p' = p - 1$ und $c = 1$, also

$$a = c + \Phi((m', p')) - \Phi((m, p)) = 1 + (\tfrac{1}{2}m' - p') - (\tfrac{1}{2}m - p) = 2.$$

Fall 2c: $p = \frac{1}{4}m$ (und $m > m_0$). –
Dann gilt $m' = \frac{1}{2}m$ und $p' = p - 1$ und $c = \frac{1}{4}m + 1$, also

$$\begin{aligned} a &= c + \Phi((m', p')) - \Phi((m, p)) = (\tfrac{1}{4}m + 1) + (\tfrac{1}{2}m' - p') - (\tfrac{1}{2}m - p) \\ &= (\tfrac{1}{4}m + 1) + 1 - (\tfrac{1}{2}m - \tfrac{1}{4}m) = 2. \end{aligned}$$

Im Fall 2c ist wieder zu sehen, wie der (vorher aufgebaute) Potenzialunterschied benutzt wird, um für eine teure Operation zu bezahlen. Wenn Op_i eine *pop*-Operation ist, sind die amortisierten Kosten durch 2 beschränkt. (Wenn sich wie in Fall 2a bei einer Situation negative amortisierte Kosten ergeben, ist das harmlos.)

(iv) $\Phi(D_n) \geq \Phi(D_0)$ gilt, weil $\Phi((m_0, 0)) = 0$ und $\Phi((m, p)) \geq 0$ für alle Zustände (m, p) ist, nach der Definition.

(v) Mit Lemma 3.3.1 folgt:

$$C_n = c_1 + \cdots + c_n \leq a_1 + \cdots + a_n \leq 3n_{push} + 2n_{pop} \leq 3n,$$

für n_{push} und n_{pop} die Anzahl der *push*- bzw. *pop*-Operationen.

Bemerkung: Der/Die aufmerksame Leser/in hat bemerkt, dass sich die Rechnungen für die Zähler- und die Stack-Beispiele bei der Bankkontomethode und der Potenzialmethode ähneln. Handelt es sich wirklich um verschiedene Methoden? Tatsächlich kann man beweisen, dass die Potenzialmethode in folgendem Sinn mindestens so stark wie die Bankkontomethode ist: Wenn man ein Analyseverfahren mit der Bankkontomethode konstruiert hat (durch Definition künstlicher amortisierter Kosten mit der Eigenschaft, dass der Kontostand nie negativ werden kann), dann kann man auch Potenziale für die möglichen Zustände D der Datenstruktur definieren, mit denen man bei Anwendung der Potenzialmethode zum gleichen Ergebnis wie bei der Bankkontomethode kommt. (Man definiert dazu $\Phi(D)$ als den *minimalen* Kontostand B, den man erreichen kann, wenn man mit dem Startzustand D_0 beginnt und eine Folge $\mathrm{Op}_1, \ldots, \mathrm{Op}_n$ ausführt, die zum Zustand D führt.) Also kann man im Prinzip mit der Potenzialmethode allein auskommen. In vielen Situationen ist aber die Bankkontomethode sehr bequem und bildet die Vorstellung vom „Ansparen“ für später auftauchende teure Operationen anschaulicher ab als die Potenzialmethode.

Ein weiteres Beispiel für die Anwendung der Potenzialmethode folgt im nächsten Kapitel.