Lösungen von Übungsblatt 3

Algorithmen (WS 2018, Ulrike von Luxburg)

Lösungen zu Aufgabe 1

a) Stack 1 und Stack 2 seien jeweils mit den Operationen $\textsc{Push}_i(v)$ und $\textsc{Pop}_i()$ für $i \in \{1, 2\}$ ausgestattet (sowie natürlich dem Test darauf, ob der Stack leer ist). Stack 1 wird als Eingabestack genutzt und Stack 2 als Ausgabestack.

Eine einfache Variante wäre, alle Eingabeelemente v mittels $\textsc{Push}_1(v)$ in Stack 1 zu schreiben und jedes Mal, wenn eine Dequeue-Operation ausgeführt wird, folgendes zu tun:

1. alle Elemente von Stack 1 in umgekehrter Reihenfolge in den (leeren) Stack 2 schreiben (mittels $\textsc{Pop}_1()$ und $\textsc{Push}_2(v)$)
2. das oberste Element aus Stack 2 ausgeben (mittels $\textsc{Pop}_2()$)
3. wie im ersten Schritt die verbleibenden Elemente in Stack 2 wieder komplett in Stack 1 umschichten (damit liegen die Elemente dort wieder in ihrer ursprünglichen Reihenfolge, das unterste Element wurde aber entfernt).

Dieser Ansatz stellt eine gültige Lösung für Aufgabenteil (a) dar, führt aber zu einem unnötigen Umschichten, wenn mehrere Dequeue-Operationen hintereinander ausgeführt werden.

In Hinblick auf Aufgabenteil (b) verändern wir obiges Vorgehen daher wie folgt: Immer wenn nach einer Ausgabe gefragt wird, wird überprüft, ob der Ausgabestack leer ist. Ist das nicht der Fall, wird einfach das oberste Element aus Stack 2 ausgegeben. Ansonsten wird der gegenwärtige Eingabestack komplett in den Ausgabestack umgeschichtet (dabei die Reihenfolge der Elemente vertauscht), das oberste Element ausgegeben und Stack 2 dann aber nicht mehr zurück auf Stack 1 geschichtet.

Genauer: Definiere $\textsc{Enqueue}(v) := \textsc{Push}_1(v)$, sowie:

```
function Dequeue( )
    if Stack 2 is empty then
        while Stack 1 is not empty do        ▷ Schichte alles von Stack 1 um auf Stack 2
            Push2(Pop1())
        end while
    end if
    return Pop2()
end function
```

Jede Push- und Pop-Operation hat Laufzeit $\Theta(1)$. Damit hat die $\textsc{Enqueue}(v)$-Operation offensichtlich Laufzeit $\Theta(1)$. Die $\textsc{Dequeue}()$-Operation hat im Falle von $s > 0$ Elementen in der Queue die worst-case Laufzeit $\Theta(s)$, da im Falle eines leeren 2. Stacks genau s Aufrufe von $\textsc{Push}_2(\textsc{Pop}_1())$ zzgl. einem finalen $\textsc{Pop}_2()$ anfallen.

b) Zunächst bestimmen wir den Aufwand, den während irgendeiner Abfolge von n_E Enqueue-Operationen und n_D Dequeue-Operationen mit $n = n_E + n_D$ ein einzelnes Element v höchstens verursacht: Bei seinem Eingang wird es einmalig auf Stack 1 gelegt ($1 \times$ Push). Irgendwann später (bei der nächsten Dequeue-Operation die auf einen leeren Stack 2 trifft), wird v in einem Rutsch mit allen anderen Elementen von Stack 1 auf Stack 2 umgeschichtet ($1 \times$ Pop, $1 \times$ Push). Schließlich wird es irgendwann später in einer Dequeue-Operation von Stack 2 geholt ($1 \times$ Pop). Insgesamt fallen somit pro Element höchstens 2 Push- und 2 Pop-Aufrufe an, mindestens aber 1 Push, von denen jede Laufzeit $\Theta(1)$ hat. Somit ist der Aufwand je Element konstant $\Theta(1)$, woraus sofort die amortisierte Laufzeit $\Theta(1)$ folgt.

Also: Obwohl einzelne Dequeue-*Operationen aufgrund der internen Umschichtung eine viel höhere Laufzeit haben können (bis zu $\Omega(n)$ für eine* einzelne Dequeue-*Operation), ist die Laufzeit* insgesamt *bei n Operationen $T_n \in \Theta(n)$ (und damit amortisiert $T_n/n \in \Theta(1)$).*

Lösungen zu Aufgabe 2

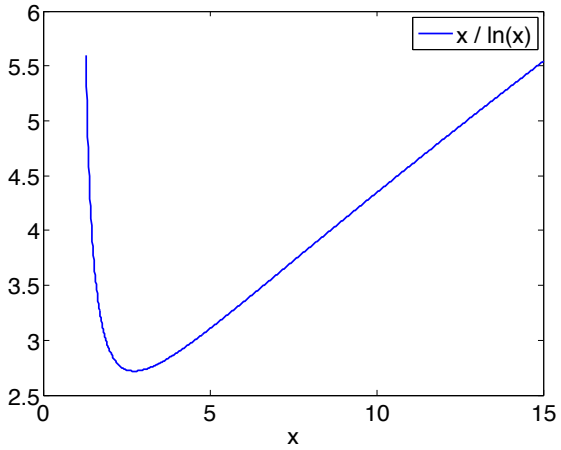
a) Mit $\ln := \log_e$ ergibt sich die Ableitung von $f(x) = x \cdot \ln(n)/\ln(x)$ als

$$f'(x) = 1 \cdot \frac{\ln(n)}{\ln(x)} - x \cdot \frac{\ln(n)}{\ln(x)^2 x} = \frac{\ln(n)(\ln(x) - 1)}{\ln(x)^2}.$$

Wegen $f'(x) = 0 \Leftrightarrow \ln(x) - 1 = 0 \Leftrightarrow x = e$ ist $x = e$ der einzige Kandidat für ein Minimum (unabhängig von n!!).

Wegen $f'(x) > 0$ für $x > e$ und $f'(x) < 0$ für $1 < x < e$ ist f auf $[e, \infty)$ streng monoton wachsend und auf $(1, e]$ streng monoton fallend. Es folgt, dass $e \approx 2.72$ tatsächlich (eindeutig bestimmtes, globales) Minimum von f ist.

b) Aufgrund der im vorigen Aufgabenteil gezeigten Monotonie von f auf $[e, \infty)$ und $(1, e]$ kommen für das Minimum in $\mathbb{N}$ nur die beiden Kandidaten 2 und 3 in Betracht. Hat man diese Monotonie nicht gezeigt, würden wir uns auch mit einem Plot der Funktion begnügen, in dem die Monotonie unmittelbar zu erkennen ist:



Wegen

$$f(2) = \underbrace{\frac{2}{\ln(2)}}_{\approx 2.89} \ln(n) > \underbrace{\frac{3}{\ln(3)}}_{\approx 2.73} \ln(n) = f(3) \qquad \text{(für alle } n > 1)$$

ist tatsächlich $k^* = 3$ das Minimum über den natürlichen Zahlen und beste Wahl.

Die worst-case Laufzeiten für $n = 10^\ell$ ergeben sich wie folgt aus $\lceil k \log_k n \rceil$:

	10^1	10^2	10^3	10^4	10^5	10^6	10^7	10^8	10^9
$k = 3$	7	13	19	26	32	38	45	51	57
$k = 2$	7	14	20	27	34	40	47	54	60

Anmerkung: Der Bereich $n = 10^1$ bis $n = 10^9$ deckt die meisten in der Praxis vorkommenden Problemgrößen ab. Dabei werden aufgrund des extrem langsam wachsenden Vorteils von $k = 3$ gegenüber $k = 2$ (den obige Tabelle zeigt) nur winzige Vorteile erzielt. Diese werden durch praktische Vorteile von $k = 2$ gänzlich aufgehoben: Computerregister sind stets binär (da high/low voltage nur zwei Werte unterscheiden kann), daher sind Multiplikationen/Divisionen mit 2 sehr schnell in Form von Bitshifts möglich - weit schneller als notwendigerweise arithmetische Operationen mit 3. Auch Speicherzugriffe sind daher für 2er-Potenzen optimiert, z.B. 64 Bit Registerbreite, oder 512kB Cache, etc.

Anders sähe es aus, wenn ein Bit 3 Zustände codieren würde (low/med/high) und damit sämtliche Register und die Arithmetik ternär arbeiten würde. Dies würde die beste Informationsdichte liefern, siehe Zusammenhänge zur Shannon-Entropie *über $x/\log x = -x \log x$.*

c) Nach wie vor ist $\lceil \log_k n \rceil$ die Höhe des "Gesamtheaps". Während der HEAPIFY-Operation auf dem Gesamtheap muss im worst-case auf jedem Level im jeweiligen "Knotenheap" der Wert des maximalen Kindknotens (also jenem in der Wurzel des "Knotenheaps") gegen den Wert des Elternknotens ausgetauscht werden — durch Austausch und HEAPIFY an der Wurzel des "Knotenheaps". Dies benötigt gemäß Aufgabenstellung Zeit $\lceil 2 \log_2(k) \rceil$. Insgesamt erhalten wir damit die worst-case Laufzeit $\lceil 2 \log_2(k) \rceil \cdot \lceil \log_k(n) \rceil$.

Anmerkung: Wegen $\lceil 2 \log_2(k) \rceil \cdot \lceil \log_k(n) \rceil \approx \lceil 2 \log_2(k) \cdot \log_k(n) \rceil = \lceil 2 \log_2(n) \rceil$ *erhalten wir bei dieser Variante also die gleiche worst-case Laufzeit wie mit einem Array-basierten* **binären** *Heap.*

d) Beim binären Heap fallen 2 Vertauschungen an, beim ternären hingegen nur 1.

e) Nein, das stimmt nicht. Ein Gegenbeispiel ist etwa DECREASE$(8 \mapsto 3)$ auf folgendem Array: $\boxed{8}\boxed{7}\boxed{6}\boxed{1}\boxed{2}\boxed{5}\boxed{4}$. Der binäre Heap auf diesen Daten benötigt nur 1 Vertauschung, der ternäre hingegen 2. Auf bestimmten Eingaben kann ein ternärer Heap also auch schlechter als ein binärer sein.

Lösungen zu Aufgabe 3

Schreiben $\mathbb{N} := \mathbb{N}_0$, und für $a, b \in \mathbb{N}$ auch $a\mathbb{N} + b := \{a \cdot t + b \mid t \in \mathbb{N}\}$, beispielsweise $5\mathbb{N} + 2 = \{2, 7, 12, 17, \ldots\}$. Jede der Hashfunktionen ist von der Form $h(k) =: g(k) \bmod 11$. An Indexposition 10 kollidieren genau die Keys mit $g(k) \bmod 11 \equiv 10$, also für $g(k) \in \{10, 21, 32, 43, \ldots\} = 11\mathbb{N} + 10$.

a) $g(k) = k$. Kollisionen falls $g(k) = k \in 11\mathbb{N} + 10$, also für jedes k das sich schreiben lässt als $k = 11 \cdot t + 10$ für ein $t \in \mathbb{N}$, also für jedes $k \in 11\mathbb{N} + 10$ selbst.

b) $g(k) = 2k$. Kollisionen falls $g(k) = 2k = 11 \cdot t + 10$, also aller (natürlichen!) k der Form $k = 11 \cdot t/2 + 5$. Da $k \in \mathbb{N}$ nur für $t = 0, 2, 4, \ldots$ gilt, liefert dies $k = 5, 16, 27, 38, \ldots$. Also Kollision aller $k \in \{11 \cdot t + 5 \mid t \in \mathbb{N}\} = 11\mathbb{N} + 5$.

c) $g(k) = k^2 + 10$. Da $k^2 + 10 \bmod 11 \equiv 10$ genau dann wenn $k^2 \bmod 11 \equiv 0$, müssen wir also alle solchen Quadratzahlen k^2 bestimmen $(1, 4, 9, 16, 25, \ldots)$ die durch 11 teilbar sind. Also muss 11 ein Primfaktor von k^2 sein. Außerdem tritt bei Quadratzahlen jeder Primfaktor eine gerade Anzahl oft auf. Also erhält man $k^2 = 11^2 \cdot a^2 = (11a)^2$ für beliebige $a \in \mathbb{N}$ als genau die gesuchten Lösungen, somit $k \in 11\mathbb{N}$.

d) $g(k) = 3^k - 1$. Da $3^k - 1 \bmod 11 \equiv 10$ genau dann wenn $3^k \bmod 11 \equiv 0$, müssen wir also alle Potenzen von 3 bestimmen die durch 11 teilbar sind. Da 3^k nur 3 als Primfaktor enthält, ist dies nie der Fall. Also wird kein k auf Position 10 abgebildet, somit gibt es dort insbesondere keine Kollisionen.