Lösungen der Hausaufgaben von Übungsblatt 2

Algorithmen und Datenstrukturen (WS 2013, Ulrike von Luxburg)

Lösungen zu Aufgabe 1

(a) In Level 0 liegt maximal $1 = k^0$ Knoten, in Level 1 maximal $1 \cdot k = k^1$, in Level 2 maximal $1 \cdot k \cdot k = k^2$ usw., also liegen in Level ℓ maximal k^ℓ Knoten.

(b) Aufsummieren über alle Level liefert $\sum_{i=0}^{\ell} k^i = \frac{k^{\ell+1}-1}{k-1}$.

(c) Falls letztes Level voll besetzt, dann $\frac{k^{\ell+1}-1}{k-1}$ viele Knoten gemäß vorigem Aufgabenteil. Da das letzte Level mindestens 1 Knoten enthält, können von obigem Wert höchstens $k^\ell - 1$ Knoten fehlen, d.h., ein solcher Baum hat mindestens $\frac{k^{\ell+1}-1}{k-1} - (k^\ell - 1) = \frac{k^\ell-1}{k-1} + 1$ viele Knoten.

[Anm: Dieser Wert ist nicht zufällig derselbe, den man alternativ aus vorigem Aufgabenteil für nur $\ell - 1$ Level zzgl. 1 weiterem Knoten erhält]

(d) *jeder* Baum mit n Knoten hat *immer* genau $n - 1$ Kanten, insbesondere k-näre Bäume irgendeiner Höhe h. Zähle so: Jeder Knoten außer der Wurzel hat genau eine Kante zum Elternknoten (bijektive Zuordnung) .

Lösungen zu Aufgabe 2

Diese drei prominenten Tree-Walks heißen Pre-Order, In-Order und Post-Order, siehe auch Wikipedia zu "Tree Traversal".

(a) Klar haben alle drei dieselbe Laufzeit, da sie bis auf die Position der PRINT-Operation übereinstimmen. Das Master-Theorem kann hier nicht angewandt werden, da bei allgemeinen Bäumen nichts über das Größenverhältnis zwischen dem linken und rechten Teilbaum bekannt ist. Man sieht die Laufzeit hingegen wie folgt: Für jede Kante (Elternknoten → Kindknoten) findet genau ein rekursiver Aufruf der ORDER-Funktion statt. Also ergibt sich mit Aufgabe 1 (d) die Gesamtanzahl Funktionsaufrufe zu n (inkl. dem initialen Aufruf), wobei jeder einzelne Aufruf inklusive der Printoperation Aufwand $\Theta(1)$ benötigt. Insgesamt ist der worst-case Zeitbedarf also $\Theta(n)$.

(b) Da für *jede* Eingabe stets für jede Kante der $n - 1$ Kanten genau einmal verzweigt wird, gilt auch für eine Best-case-Eingabe der Größe n, dass sie Laufzeit $\Omega(n)$ hat.

(c)

1:	N	A	O	E	I	F	M	R	L	U	S	G	A	R	T	H
2:	I	E	O	F	A	R	M	L	N	G	S	A	U	T	R	H
3:	I	E	F	O	R	L	M	A	G	A	S	T	H	R	U	N

(d)

T	E	E	O	Y	R	E	L	V	L

(e)

A	L	G	O	R	I	T	H	M	S	A	R	E	F	U	N

Lösungen zu Aufgabe 3

(a) Mit $\ln := \log_e$ ergibt sich die Ableitung zu

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

Wegen $\ln(x) - 1 = 0 \Leftrightarrow x = e$, ist $x = e$ der einzige Kandidat für ein Minimum. Zudem ist offensichtlich $\lim_{x\to 1} f(x) = \infty = \lim_{x\to\infty} f(x)$, d.h., an den Grenzen des Definitionsbereichs

geht f gegen unendlich. Somit muss es sich aufgrund der Stetigkeit von f an der Stelle $e \approx 2.718$ tatsächlich um ein Minimum handeln.

[Anm: Man könnte natürlich auch zeigen, dass $f''(e) > 0$ und dann weiter argumentieren.]

(b) Aufgrund der im vorigen Aufgabenteil implizierten Monotonie von f auf $(1, e]$ und $[e, \infty)$ kommen für das Minimum in $\mathbb{N}$ nur die beiden Kandidaten 2 und 3 in Betracht. Tatsächlich ist $f(2) - f(3) = \underbrace{\left(\frac{2}{\log 2} - \frac{3}{\log 3}\right)}_{>0} \log n > 0$ für alle $n > 1$, also $f(2) > f(3)$ für alle $n > 1$.

Somit liegt das Minimum über den natürlichen Zahlen bei $k = 3$. Dies wäre also die beste Wahl. Die 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

(c) Der Bereich 10^1 bis 10^9 deckt die meisten in der Praxis vorkommenden Problemgrößen ab. Dabei werden aufgrund des extrem langsam wachsenden Vorteils von $k = 3$ gegenüber $k = 2$ 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.

[Anm: 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 $x / \log x = -x \log x$.]

(d) Nach wie vor ist $\log_k n$ die Höhe des "Gesamtheaps". Während der HEAPIFY-Operation auf dem Gesamtheap muss bei jeder Vertauschung auch im jeweiligen "Knotenheap" der Wert des maximalen Kindknotens gegen den Wert des Elternknotens ausgetauscht werden (z.B. per DECREASE). Dies benötigt nach Aufgabenstellung Zeit $2 \log_2(k)$. Insgesamt erhalten wir also die worst-case Laufzeit $2 \log_2(k) \cdot \log_k(n) = 2 \log_2(n)$, also dieselbe wie mit einem Array-basierten k-nären Heap für $k = 2$.

[Anm: Interessanterweise erreichen wir so also keinen weiteren Geschwindigkeitsvorteil für den worst-case. Selbiges gilt wenn wir $\tilde{k}$-näre Heaps als "Knotenheaps" einsetzen, was Gesamtlaufzeit $\tilde{k} \log_{\tilde{k}}(k) \cdot \log_k(n) = \tilde{k} \cdot \log_{\tilde{k}}(n)$ lieferte, also dasselbe wie ein Array-basierter Gesamtheap mit $k = \tilde{k}$.]

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

(f) 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 1 Vertauschung, der ternäre aber 2. Auf bestimmten Eingaben kann ein ternärer Heap also auch schlechter als ein binärer sein.

Lösungen zu Aufgabe 4

(a) MERGE$(22579, 1248)$
$1 \circ$ MERGE$(22579, 248)$
$12 \circ$ MERGE$(2579, 248)$
$122 \circ$ MERGE$(579, 248)$
$1222 \circ$ MERGE$(579, 48)$
$12224 \circ$ MERGE$(579, 8)$
$122245 \circ$ MERGE$(79, 8)$
$1222457 \circ$ MERGE$(9, 8)$
$12224578 \circ$ MERGE$(9, [])$
122245789

(b)
6	7	8	3	4	2	9	1
6	7	3	8	2	4	1	9
3	6	7	8	1	2	4	9
1	2	3	4	6	7	8	9

(c) (1) Vergleiche statt $x[1] \leq y[1]$ auf $x[1] \geq y[1]$ und lasse alles andere unverändert. Erhalten induktiv zwei *absteigend* sortierte Folgen als Eingabe für MERGE, von denen wir das Maximum beider linker Enden wählen und es im Output nach ganz vorn stellen. Diese MERGE-Variante führt die beiden Eingabearrays also in absteigender Reihenfolge zusammen.

(2) Vergleiche statt $x[1] \leq y[1]$ auf $x[k] \leq y[\ell]$ und konkateniere dann von rechts, z.B., $\text{MERGE}(x[1 \ldots k-1], y[1 \ldots \ell]) \circ x[k]$. Erhalten induktiv wiederum *absteigend* sortierte Folgen als Eingabe für MERGE, von denen wir das Minimum beider rechten Enden wählen und im Output ganz nach hinten anstellen. Auch diese MERGE-Variante führt die beiden Eingabearrays also in absteigender Reihenfolge zusammen.

Lösungen zu Aufgabe 5

(a) Stack 1 und Stack 2 seien jeweils mit den Operationen $\text{PUSH}_i(v)$ und $\text{POP}_i()$ 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. Immer wenn nach einer Ausgabe gefragt wird, aber der Ausgabestack leer ist, so wird der gegenwärtige Eingabestack komplett in den Ausgabestack umgeschichtet (und dabei die Reihenfolge der Elemente vertauscht). Genauer: Definiere $\text{ENQUEUE}(v) := \text{PUSH}_1(v)$, 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
            PUSH_2(POP_1())
        end while
    end if
    return POP_2()
end function
```

Jede PUSH- und POP-Operation hat Laufzeit $\Theta(1)$. Damit hat die $\text{ENQUEUE}(v)$-Operation offensichtlich Laufzeit $\Theta(1)$. Die 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 $\text{PUSH}_2(\text{POP}_1())$ zzgl. einem finalen $\text{POP}_2()$ 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, woraus sofort die amortisierte Laufzeit $\Theta(1)$ folgt.

Genauer: 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 höchstens $T_n \in \mathcal{O}(n)$ ($2n \times$ PUSH, $2n \times$ POP), sowie mindestens $T_n \in \Omega(n)$ ($n \times$ PUSH). Also ist die Laufzeit insgesamt bei n Operationen $T_n \in \Theta(n)$, bzw. amortisiert $T_n/n \in \Theta(1)$.