AD [HA] zum 6. 11. 2013

Arne Struck, Lars Thoms

5. November 2013

1. a)

Es liegen k^l Blätter maximal in der l. Ebene. Von jedem Knoten gehen k Knoten ab, das führt zum folgenden: 0. Ebene (root): $1 = k^0$, 1. Ebene: $k = k^1$, 2. Ebene: $k \cdot k = k^2$, 3. Ebene: $k \cdot k \cdot k = k^3$ ... l.Ebene: k^l

b)

Der volle Baum hat $\sum_{i=0}^{l} k^i = \frac{k^{i+1}-1}{k-1} = \frac{k^i + k^i \cdot (k-1)}{k-1} = \frac{k^i - 1}{k-1} + k^i$ Knoten, die Summe der Knoten aller Ebenen (eine volle Ebene bemisst sich, wie in a) dargestellt auf k^l).

c)

Der vollständige Baum hat $\sum_{i=0}^{l-1} k^i + c = \frac{k^i-1}{k-1} + k^i - k^l + c \mid c \in \mathbb{N} : 1 \leq c \leq k^l$ Blätter.
Der vollständige Baum ist bis zu seiner vorletzten Ebene maximal gefüllt, deswegen die Summe bis $l-1$, c repräsentiert die Anzahl der Blätter in der letzten Ebene, welche zwischen einem (sonst wäre der Baum voll und hätte l-1 Ebenen) und k^l (ein voller Baum ist vollständig) Blättern.

d)

Der Baum hat n-1 Kanten, da jeder Knoten (bis auf den Wurzelknoten) eine Kante besitzt durch die er mit seinem Elternknoten verbunden ist.

2. a)

Die Laufzeit kann wie folgt (für OrderX) hergeleitet werden, die Reihenfolge der prints ist nicht relevant.

$print(v)$	$\Theta(1)$
$OrderX(l)$	$\mathcal{O}(\frac{k-1}{2})$
$OrderX(r)$	$\mathcal{O}(\frac{k-1}{2})$

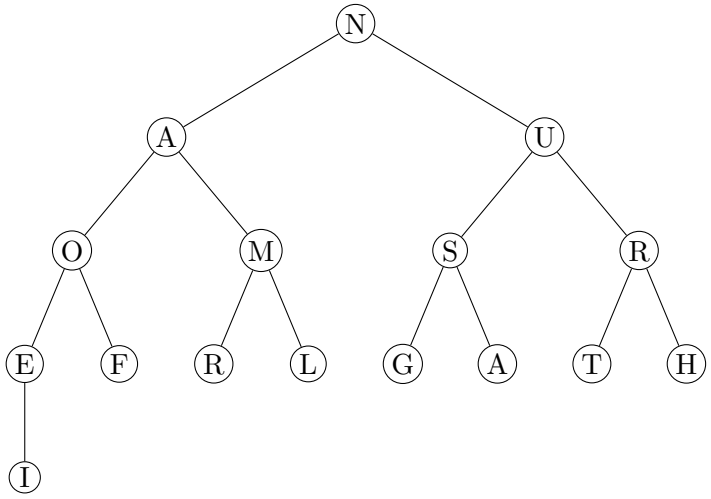
Das master-Theorem ist nun anwendbar,

$T(k) = 2T(\lceil \frac{k-1}{2} \rceil) + \mathcal{O}(k^0)$
Da $\log_2 2 = 1$ gilt, folgt $\mathcal{O}(k^1)$

b)

Die Laufzeiten sind bei gleicher Knotenzahl identische (wie in a) zu sehen, alle Algorithmen haben die gleiche Anzahl an Aufrufen, da nirgends abgebrochen wird, außer wenn keine Kindknoten verfügbar sind).

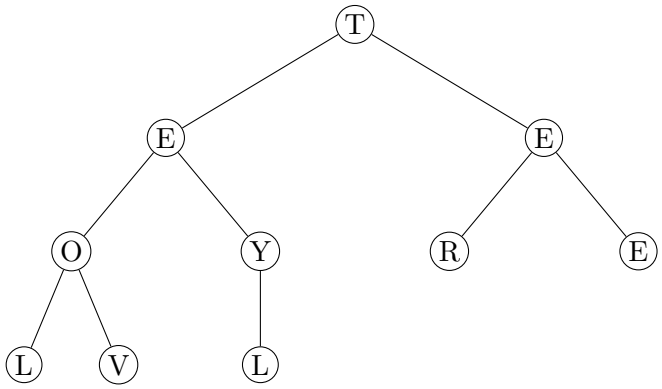
c)



Order1:	N	A	O	E	I	F	M	R	L	U	S	G	A	R	T	H
Order2:	I	E	O	F	A	R	M	L	N	G	S	A	U	T	R	H
Order3:	I	E	F	O	R	L	M	A	G	A	S	T	H	R	U	N

d)

Der LOVELYTREE nach Order 2:

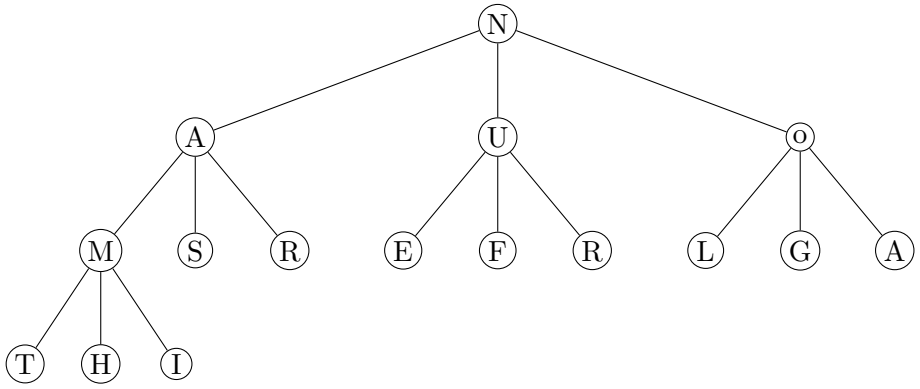


Nach Level-Order: Order3:

T	E	E	O	Y	R	E	L	V	L

e)

Ternärer Baum mit vorgegebener Befehlsreihenfolge:



Ausgabe: Order3:

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

3. a)

$$\begin{aligned}
\left(x \cdot \tfrac{\ln(n)}{\ln(x)}\right)' &= \tfrac{\ln(n)}{\ln(x)} + \left(\tfrac{\ln(n)}{\ln(x)}\right)' \\
&= \tfrac{\ln(n)}{\ln(x)} - \tfrac{\ln(n)\ln(x)'}{(\ln(x))^2} \\
&= \tfrac{\ln(n)}{\ln(x)} - \tfrac{\ln(n)\ln(x)'}{(\ln(x))^2} \\
&= \tfrac{\ln(n)}{\ln(x)} - \tfrac{\ln(n)}{x(\ln(x))^2} \\
&= \tfrac{\ln(n)\ln(x)}{\ln(x)^2} - \tfrac{\ln(n)}{x(\ln(x))^2} \\
&= \tfrac{\ln(n)(\ln(x)-1)}{\ln(x)^2}
\end{aligned}$$

Man sieht, dass einer der Faktoren im Zähler 0 sein muss, damit $\frac{\ln(n)(\ln(x)-1)}{\ln(x)^2} = 0$ gilt. Da n beliebig, aber fest ist, ist die Frage, für welches x dies gilt. Wenn $x = e$ gilt, dann folgt $\frac{\ln(n)(\ln(e)-1)}{\ln(e)^2} = \frac{0}{1}$
Da es sich um die einzige Extremstelle handelt, ist es das gesuchte Minimum.

b)

Wir wissen aus b), dass das ideale $x = k = e$ gilt, da $k \in \mathbb{N}$ gilt und e näher an 3, als an 2 ist, ist $k = 3$ die optimale Belegung für jedes n

$$\begin{aligned}
\lceil 3 \cdot \log_3(10^1) \rceil &= 7 & \lceil 2 \cdot \log_2(10^1) \rceil &= 7 \\
\lceil 3 \cdot \log_3(10^2) \rceil &= 13 & \lceil 2 \cdot \log_2(10^2) \rceil &= 14 \\
\lceil 3 \cdot \log_3(10^3) \rceil &= 19 & \lceil 2 \cdot \log_2(10^3) \rceil &= 20 \\
\lceil 3 \cdot \log_3(10^4) \rceil &= 26 & \lceil 2 \cdot \log_2(10^4) \rceil &= 27 \\
\lceil 3 \cdot \log_3(10^5) \rceil &= 32 & \lceil 2 \cdot \log_2(10^5) \rceil &= 34 \\
\lceil 3 \cdot \log_3(10^6) \rceil &= 38 & \lceil 2 \cdot \log_2(10^6) \rceil &= 40 \\
\lceil 3 \cdot \log_3(10^7) \rceil &= 45 & \lceil 2 \cdot \log_2(10^7) \rceil &= 47 \\
\lceil 3 \cdot \log_3(10^8) \rceil &= 51 & \lceil 2 \cdot \log_2(10^8) \rceil &= 54 \\
\lceil 3 \cdot \log_3(10^9) \rceil &= 57 & \lceil 2 \cdot \log_2(10^9) \rceil &= 60
\end{aligned}$$

c)

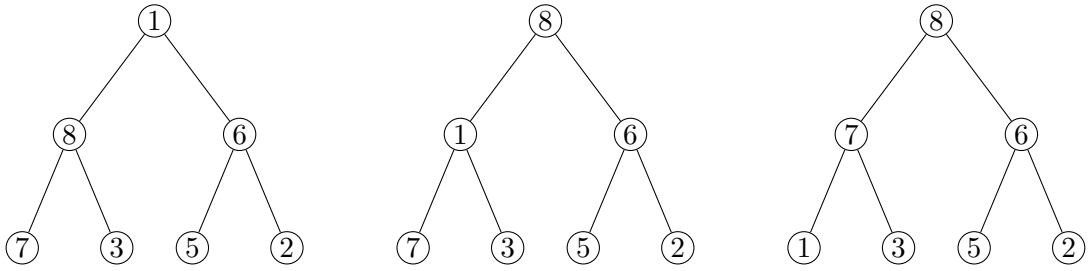
k = 2 wird verwendet, weil die momentane Rechnerstrukturen, Binärstrukturen einfacher verarbeiten können.

d)

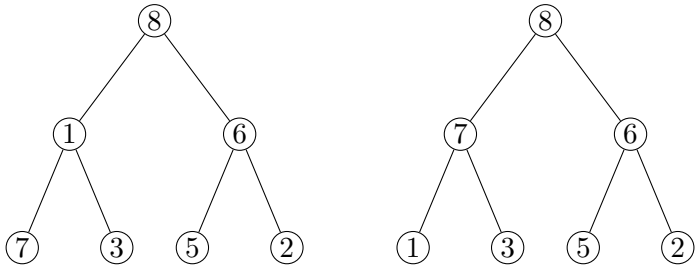
Da der Wurzelknoten des jeweiligen Max-Heaps das größte Kind darstellt, ist der Aufwand für einmal vertauschen 1. Allerdings wird durch das Vertauschen im Max- Heap die Max-Heap-Eigenschaft gestört und muss wieder hergestellt werden. Dies dauert im worst-case $\frac{k}{2}$ Schritte. Es treten allerdings nicht nur Veränderungen im Max-Heap des Elternknotens auf, sondern auch in dem des (ehemaligen) Kindknotens und dem des Elternknotens des Elternknotens. Auch diese müssten wieder Heapified werden. Da für einen Binär-Heap mit k Elementen $2\lceil \log_2(k) \rceil \cdot 2$ und es 3 zu verändernde Max-Heaps existieren gilt folgt: $k \cdot 3 \cdot (\lceil \log_2(k) \rceil + 2) + 1$ für die notwendige Schrittzahl.

e)

Es sind 2 Vertauschungen vom Originalbaum weg notwendig:



Es ist eine Vertauschung vom Originalbaum notwendig:



f)

Wir wissen, dass ein k-närer Baum $\lceil k \log_k(n) \rceil$ Schritte benötigt, daraus folgt folgender Beweis:

Beh.:

$\forall n \in \mathbb{N}$ gilt: $\lceil 3 \log_3(n) \rceil \leq \lceil 2 \log_2(n) \rceil$

I.Anf.:

$\lceil 3 \log_3(1) \rceil = 0 = \lceil 2 \log_2(1) \rceil$

I.A.:

Die Behauptung gilt für ein bestimmtes, aber frei wählbares $n \in \mathbb{N}$

I.S.: $(z.z.: \lceil 3\log_3(n+1)\rceil \leq \lceil 2\log_2(n+1)\rceil)$

$$\begin{aligned}\lceil 3\log_3(n+1)\rceil &= \lceil 3\tfrac{\ln(n+1)}{\ln(3)}\rceil \\ &= \lceil \ln(n+1)\tfrac{3}{\ln(3)}\rceil \leq \lceil \ln(n+1)\tfrac{2}{\ln(2)}\rceil \\ &= \lceil 2\tfrac{\ln(n+1)}{\ln(2)}\rceil \\ &= \lceil 2\log_2(n+1)\rceil\end{aligned}$$

Damit ist die Behauptung bewiesen $\square$

4. a)

merge(22579,1248)
1 ∘ merge(22579,248)
12 ∘ merge(2579,248)
122 ∘ merge(579,248)
1222 ∘ merge(579,48)
12224 ∘ merge(579,8)
122245 ∘ merge(79,8)
1222457 ∘ merge(9,8)
12224578 ∘ merge(9,[])
122245789

b)

Input(splitted): 6 7 8 3 4 2 9 1

$$\underbrace{\underbrace{\overbrace{6}\ \overbrace{7}}^{67}\ \underbrace{\overbrace{8}\ \overbrace{3}}^{38}}_{3678}\ \underbrace{\underbrace{\overbrace{4}\ \overbrace{2}}^{24}\ \underbrace{\overbrace{9}\ \overbrace{1}}^{19}}_{1249}$$

$$12346789$$

c)

Die erste Möglichkeit ist in merge $x[1] \leq y[1]$ zu $x[1] \geq y[1]$ abzuändern, wie folgt dargestellt:

function MERGE($x[1..k], y[1..l]$)
  if $k = 0$ **then**
    return $y[1..l]$
  end if
  if $l = 0$ **then**
    return $x[1..k]$
  end if
  if $x[1] \geq y[1]$ **then**
    return $x[1] \circ$ MERGE($x[2..k], y[1..l]$)
  else
    return $y[1] \circ$ MERGE($x[1..k], y[2..l]$)
  end if
end function

Oder man stellt die Ausführung der Konkatenation in merge um, wie im Folgenden:

```
function MERGE(x[1..k], y[1..l])
    if k = 0 then
        return y[1..l]
    end if
    if l = 0 then
        return x[1..k]
    end if
    if x[1] ≤ y[1] then
        return MERGE(x[2..k], y[1..l] ∘ x[1])
    else
        return MERGE(x[1..k], y[2..l] ∘ y[1])
    end if
end function
```

5. a)

Man nutzt einen Stack, um den Queue-Eingang (in) und einen um den Queue-Ausgang (out) zu simulieren. Soll ein Element(e) in die Queue eingefügt werden, wird nun jedes Element des Ausgang-Stacks auf den Eingang-Stack geschrieben, somit liegen die Elemente auf dem Eingang- Stack in reverser Reihenfolge vor. Dann wird das hinzuzufügende Element auf dem Eingang-Stack push aufgerufen. Darauf werden alle Elemente des Eingang-Stacks auf den Ausgang-Stack geschoben, nun kann man wieder auf das erste Element, welches auf die Stacks geschrieben wurde zugreifen. Somit ist das First-in-First-Out-Prinzip erfüllt. Soll nun ein Element entfernt werden wird einfach auf dem Ausgang-Stack pop aufgerufen. Die worst-case-Laufzeit beträgt für Enqueue() $\mathcal{O}(n)$, die von Dequeue() beträgt $\mathcal{O}(1)$.

```
function DEQUEUE
    out.pop()
end function
function ENQUEUE(e)
    if out.isEmpty() then
        out.push(e)
    else
        while (!out.isEmpty) do
            element a ← out.pop()
            in.push(a)
        end while
        in.push(e)
        while (!in.isEmpty) do
            element b ← in.pop()
            out.push(b)
        end while
    end if
end function
```

b)

Da eine beliebige Folge n gebraucht wird, kann man bei einem Stack mit mehr als 2 Elementen Dequeue()Dequeue()Dequeue() als n festlegen. Da Dequeue() eine einstellig Funktion ist, kann man davon ausgehen, dass die worst-case- Laufzeit gleich der best-case-Laufzeit gleich $\mathcal{O}(1)$ ist.