AD-Übung zum 6. November

Arne Beer, MN 6489196
Merve Yilmaz, MN 6414978
Sascha Schulz, MN 6434677

15. November 2013

1. (a) Ab dem Wurzelknoten gehen von jedem Knoten bis zu k Knoten ab. d.h. in Ebene 1 haben wir k Knoten, in Ebene 2 fogllich $k \cdot k = k^2$, in Ebene 3 dann schon $k \cdot k \cdot k = k^3$ usw. bis zur Ebene l mit maximal k^l Knoten.

(b) Wie zuvor gezeigt befinden sich auf der Ebene x maximal k^x Knoten. Ein voller Baum hat folglich $\sum_{i=0}^{l} k^i$ Knoten.

(c) Der vollstaendige Baum hat $\sum_{i=0}^{l-1} k^i + c | c \in \mathbb{N} : 1 \leq c \leq k^l$. Dies laesst sich daraus herleiten, dass der Baum bis in die letzte Ebene voll ist und in der letzten Ebene beliebige Knoten vorhanden sein koennen.

(d) Der Baum besitzt $n - 1$ Kanten, da zu jedem Knoten jeweils eine Kante fuehrt, mit Ausnahme des Obersten.

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

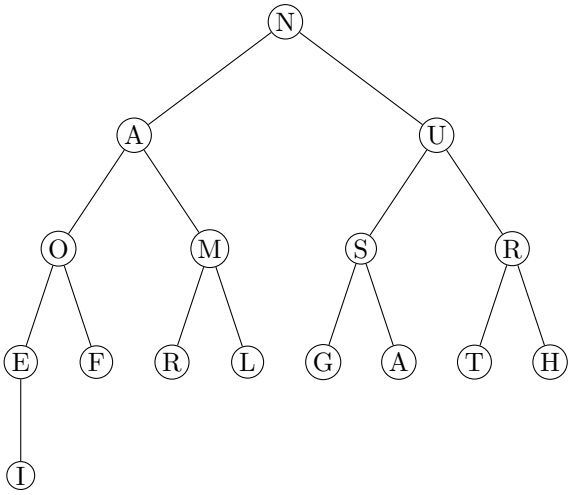
$print(v)$	$\Theta(1)$
$OrderX(l)$	$\mathcal{O}(\frac{k-1}{2})$
$OrderX(r)$	$\mathcal{O}(\frac{k-1}{2})$

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) Binärbaum nach Level-Order:

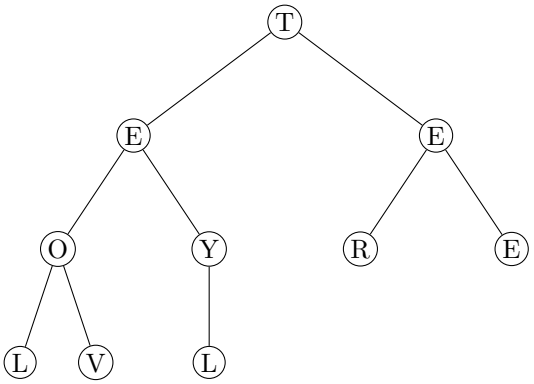


Nach ORDER1: NAOEIFMRLUSGARTH

Nach ORDER2: IEOFARMLNGSAUTRH

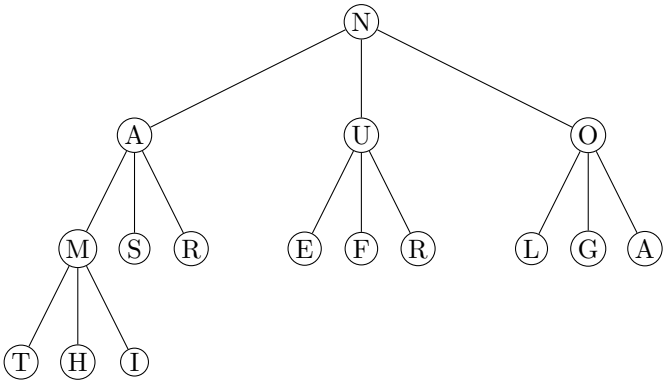
Nach ORDER3: IEFORLMAGASTHRUN

(d) Zugehöriger Binärbaum (LOVELYTREE):



In Array-Schreibweise (Level-Order): TEEOYRELVL

(e) Ternärer Baum zum Array aus Aufgabenteil (c):



Call-Reihenfolge:

1. rightChild
2. leftChild
3. middleChild

4. PRINT

Ausgabe: ALGORITHMS ARE FUN (Leerzeichen nur zur Lesbarkeit)

3. (a) $f(x) = x \log_x(n) = x \cdot \frac{\log_e(n)}{\log_e(x)} = x \cdot \log_e(n) \cdot log_e^{-1}(x)$

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

$f(x)' = \frac{\log_e(n)}{\log_e(x)} - \frac{\log_e(n)}{log_e^2(x)}$

$f(x)' = \frac{\log_e(n)}{\log_e(x)} \cdot (1 - \frac{1}{log_e(x)})$

$f(x)' = 0 \quad \Rightarrow \quad \frac{\log_e(n)}{\log_e(x)} = 0 \quad \text{oder} \quad (1 - \frac{1}{log_e(x)}) = 0$

$0 = \frac{\log_e(n)}{\log_e(x)} \quad \Leftrightarrow \quad 0 = \log_e(n)$ Ergebnis irrelevant da nicht abhängig von x.

$0 = 1 - \frac{1}{log_e(x)} \quad \Leftrightarrow \quad 1 = log_e(x) \quad \Leftrightarrow \quad x = e$

TODO: Nachweisen, dass es sich tatsächlich um das Minima handelt

(b) Beste Wahl für Anzahl an Kinderknoten müsste mit $x = e$ aus Aufgabenteil a $k = 3$ sein (da $e \approx 2,718\ldots$).

Für verschiedene Heap-Größen bedeutet dies:

$n = 10^1 \quad \Rightarrow \quad 3 \cdot \log_3(10) \approx 6,288 \approx 7$ Schritte
Zum Vergleich: $2 \cdot \log_2(10) \approx 6,644 \approx 7$ Schritte

$n = 10^2 \quad \Rightarrow \quad 3 \cdot \log_3(10^2) \approx 12,575 \approx 13$ Schritte
Zum Vergleich: $2 \cdot \log_2(10^2) \approx 13,288 \approx 14$ Schritte

$n = 10^3 \quad \Rightarrow \quad 3 \cdot \log_3(10^3) \approx 18,863 \approx 19$ Schritte
Zum Vergleich: $2 \cdot \log_2(10^3) \approx 19,932 \approx 20$ Schritte

$n = 10^4 \quad \Rightarrow \quad 3 \cdot \log_3(10^4) \approx 25,151 \approx 26$ Schritte
Zum Vergleich: $2 \cdot \log_2(10^4) \approx 26,575 \approx 27$ Schritte

$n = 10^5 \quad \Rightarrow \quad 3 \cdot \log_3(10^5) \approx 31,439 \approx 32$ Schritte
Zum Vergleich: $2 \cdot \log_2(10^5) \approx 33,219 \approx 34$ Schritte

$n = 10^6 \quad \Rightarrow \quad 3 \cdot \log_3(10^6) \approx 37,726 \approx 38$ Schritte
Zum Vergleich: $2 \cdot \log_2(10^6) \approx 39,883 \approx 40$ Schritte

$n = 10^7 \quad \Rightarrow \quad 3 \cdot \log_3(10^7) \approx 44,014 \approx 45$ Schritte
Zum Vergleich: $2 \cdot \log_2(10^7) \approx 46,507 \approx 47$ Schritte

$n = 10^8 \quad \Rightarrow \quad 3 \cdot \log_3(10^8) \approx 50,302 \approx 51$ Schritte
Zum Vergleich: $2 \cdot \log_2(10^8) \approx 53,151 \approx 54$ Schritte

$n = 10^9 \quad \Rightarrow \quad 3 \cdot \log_3(10^9) \approx 56,589 \approx 47$ Schritte
Zum Vergleich: $2 \cdot \log_2(10^9) \approx 59,795 \approx 60$ Schritte

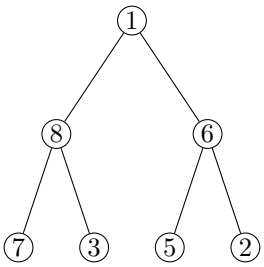
Wie zu erkennen ist, werden für $k = 2$ mehr Schritte als für $k = 3$ benötigt, die Differenz scheint mit der Zeit zu steigen.

(c) In der Praxis könnte der Einsatz von $k = 2$ dennoch effektiver sein, da sein 2er-Potenzen durch shiften schneller errechnet werden können und so die Indizes des Beginns einer Ebene schneller errechnet werden können.

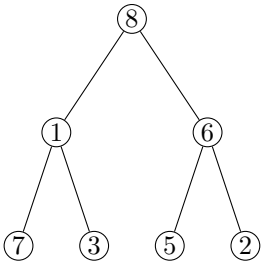
(d) Pro Vertauschen werden k + 1 Schritte benötigt. Ein Schritt wird benötigt, um das Maximum herauszufinden und k Schritte, um den Max-Heap des aktuellen Knoten nach dem Vertauschen wieder zu einem solchen zu machen. Damit werden zwar viele Schritte zum Finden eines Maximums der Kinder eingespart, allerdings an anderer Stelle wieder durch das Aufrufen von Heapify

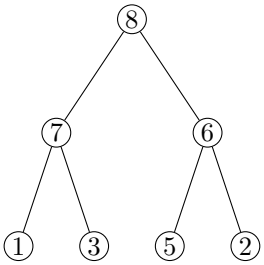
auf den zusätzlichen Max-Heap ausgegeben. Im Endeffekt ergibt sich damit eine Gesamtlaufzeit von $\lceil (k+1) log k(n) \rceil$.

(e) Zustand von 3.d nach DECREASE($9 \mapsto 1$):

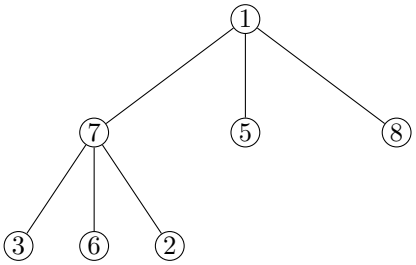


Benötigt 2 Vertauschungen:

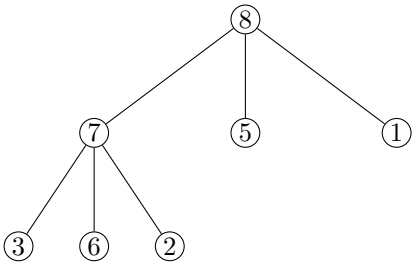




Zustand von 3.f nach DECREASE($9 \mapsto 1$):



Benötigt 1 Vertauschung zu:



(f) Wie im Einleitungstext der Aufgabe beschrieben benötigt die HEAPIFY-Operation bei einem k-nären Heap mit n Elementen $\lceil k \log_k(n) \rceil$ Schritte.

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

Induktionsannahme:
$\lceil 3\log_3(n)\rceil \leq \lceil 2\log_2(n)\rceil$

Induktionsschritt:
$\lceil 3\log_3(n+1)\rceil = \lceil 3\frac{\log_e(n+1)}{\log_e(3)}\rceil$
$= \lceil \log_e(n+1)\frac{3}{\log_e(3)}\rceil \leq \lceil \log_e(n+1)\frac{2}{\log_e(2)}\rceil$
$= \lceil 2\frac{\log_e(n+1)}{\log_e(2)}\rceil = \lceil 2\log_2(n+1)\rceil$

Folglich gilt $\lceil 3\log_3(n)\rceil \leq \lceil 2\log_2(n)\rceil$ für alle $n \in \mathbb{N}$.

4. (a) $merge(22579, 1248)$
-> $1 \circ merge(22579, 248)$
-> $1 \circ (2 \circ merge(2579, 248))$
-> $1 \circ (2 \circ (2 \circ merge(579, 248)))$
-> $1 \circ (2 \circ (2 \circ (2 \circ merge(579, 48))))$
-> $1 \circ (2 \circ (2 \circ (2 \circ (4 \circ merge(579, 8)))))$
-> $1 \circ (2 \circ (2 \circ (2 \circ (4 \circ (5 \circ merge(79, 8))))))$
-> $1 \circ (2 \circ (2 \circ (2 \circ (4 \circ (5 \circ (7 \circ merge(9, 8)))))))$
-> $1 \circ (2 \circ (2 \circ (2 \circ (4 \circ (5 \circ (7 \circ (8 \circ merge(9, [])))))))))$
-> $1 \circ (2 \circ (2 \circ (2 \circ (4 \circ (5 \circ (7 \circ (8 \circ 9)))))))$
-> $1 \circ (2 \circ (2 \circ (2 \circ (4 \circ (5 \circ (7 \circ 89))))))$
-> $1 \circ (2 \circ (2 \circ (2 \circ (4 \circ (5 \circ 789)))))$
-> $1 \circ (2 \circ (2 \circ (2 \circ (4 \circ 5789))))$
-> $1 \circ (2 \circ (2 \circ (2 \circ 45789)))$
-> $1 \circ (2 \circ (2 \circ 245789))$
-> $1 \circ (2 \circ 2245789)$
-> $1 \circ 22245789$
-> 122245789

(b) Erst aufsplitten:
67834291 6783|4291
67|83 42|91
67 83 42 91

Dann sortieren durch mergen und Gruppen zusammenführen:
67 38 24 19
(67 38) (24 19)
3678 1249
(3678 1249)
12346789

(c) Eine absteigende Reihenfolge wird erreicht, wenn die abgespaltete Ziffer an das Ergebnis des rekursiven Aufrufs konkateniert wird, statt wie bisher anders herum:

```
if x[1] <= y[1]:
    return merge(x[2...k], y[1...l]) concatWith x[1]
else:
    return merge(x[1...k], y[2...l]) concatWith y[1]
```

5. (a) Die Queue kann mit einem Enqueue- und einem Deqeueue-Stack umgesetzt werden.

Auf den Enqueue-Stack wird push() ausgeführt, vom Dequeue-Stack Einträge mit pop() genommen. Wenn der Dequeue-Stack leer ist, werden alle bisher enqueuten Einträge zu dequeuen in den dequeue Stack umgeschichtet.

Das Worst-Case-Scenario wäre daher, dass bei leeren Dequeue-Stack nach n-Elementen enqueued das erste mal dequeued werden soll. Dies kostet O(n).

```
Declare Stack_Enqueue
Declare Stack_Dequeue

function enqueue(element){
    Stack_Enqueue.push(element)
}

function dequeue(){
    if number_of_elements(Stack_Dequeue) = 0 then
        while number_of_elements(Stack_Enqueue) > 0 do
            stack_Dequeue.push(stack_Enqueue.pop())
    stack_Dequeue.pop()
}
```

(b) $enqueue() = O(1)$
$dequeue() = O(1)$ für $numberOfElements(Dequeue_Stack) > 0$, sonst
$dequeue() = O(n)$ für $numberOfElements(Enqueue_Stack) = n$

Worst-Case: $(n-1) \cdot enqueue(\cdot) + dequeue() = (n-1) \cdot O(1) + O(n) = O(n)$

Amortisierte Laufzeit T_n/n: $O(n)/n = O(1)$

Implementierung siehe Aufgabenteil a.

Begründung: Die einzige teure Aufgabe ist das interne Umschichten zwischen den Stacks. Dies ist nur erforderlich, wenn der Dequeue-Stack leer ist. Sofern sich auf diesem Einträge befinden, sind diese korrekt geordnet und können nach den FIFO-Prinzip korrekt entnommen werden. Alle derweil enqueuten Einträge befinden sich derweil in Enqueue Stack, welches einen an den Deuqueue Stack angeschlossenen Datenabschnitt handelt, jedoch während des Enqueues noch in invertierter Reihenfolge.