Hausaufgaben

Algorithmen und Datenstrukturen
Gruppe 6, zum 6. November 9:00

Felix Braun (5881661), Stine Griep (), Stefan Dang (6589689), Daniel Meier (6592507, Gruppe 9)

Hamburg, 6. November 2013

Aufgabe 1 k-närer Baum

(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 folglich $k * k = k^2$, in Ebene 3 dann schon $k * k * k = k^3$ 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: $\sum_{i=0}^{l-1} k^i$ Knoten.

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

(d) Der Baum besitzt $n - 1$ Kanten, da zu jedem Knoten jeweils eine Kante führt, mit Ausnahme des obersten Knotens (Wurzel).

Aufgabe 2 Tree Walk

(a) Die Laufzeit der drei Algorithmen ist $T(N) = O(n)$.
Master Theorem: $2 * T(\frac{n}{2}) + O(1)$
damit ist a: 2, b: 2, c: 0
$log_2 2 = 1 > 0$
$T(n) = n^1 = n$

(b) Die Algorithmen werden genutzt um jeden Knoten eines binären Baumes einmal auszugeben. Um dies zu ermöglichen muss auf jeden Knoten einmal zugegriffen werden. Damit ist die Laufzeit ausschließlich abhängig von der Größe n des Baumes und weist keine Unterschiede zwischen best-case und worst-case auf. Mit anderen Worten, die Laufzeit kann im best-case nicht verbessert werden.

(c) ORDER1: $Ausgabe_{preOrder}$ = `NAOEIFMRLUSGARTH`
ORDER2: $Ausgabe_{inOrder}$ = `IEOFARMLNGSAUTRH`
ORDER3: $Ausgabe_{postOrder}$ = `IEFORLMAGASTHRUN`

(d) ORDER2: $T_{Lovelytree}$ =

T	E	E	O	Y	R	E	L	V	L

(e) ORDER3tern: $Ausgabe_{ternär}$ = `ALGORITHMSAREFUN`

Aufgabe 4 Sortieren

(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)
122245789
```

(b) `Mergesort(67834291)`

```
6 7 8 3 4 2 9 1
67 38 24 19
3678 1249
12346789
```

(c) **Möglichkeit 1.** Durch Umkehrung der Rekursion und Kontrollbedingung wird auch die Sortierreihenfolge umgekehrt:

```
function Merge(x[1...k], y[1...l])
        if k = 0: return y[1...l]
        if l = 0: return x[1...k]
```

```
        if x[k] >= y[l]:
                return x[k]    Merge(x[k-1...1], y[l...1])
        else:
                return y[l]    Merge(x[k...1], y[l-1...1])
```

Möglichkeit 2. Durch Invertierung der Daten und Kontrollbedingung wird auch die Sortierreihenfolge umgekehrt:

```
function Merge(x[1...k], y[1...l])
        if k = 0: return y[1...l]
        if l = 0: return x[1...k]

        x: invert_order(x)
        y: invert_order(y)

        if x[1] >= y[1]:
                return x[1]    Merge(x[2...k], y[1...l])
        else:
                return y[1]    Merge(x[1...k], y[2...l])
```

Aufgabe 5 Stacks und Queues

(a) Stacks und Queues unterscheiden sich in der Ausgabereihenfolge der Elemente bei einem Pop. Stehen intern 2 Stacks (*last in, first out*) zur Verfügung um nach außen hin ein Queue (*first in, first out*) zu simulieren, lässt sich das realisieren, indem man zuerst alle Elemente in einen ersten Stack schreibt (Enqueue). Bei einem Dequeue wird der komplette erste Stack zunächst über Pop in einen zweiten Stack geschrieben, wobei sich die Ausgabereihenfolge automatisch umkehrt, was dann dem Verhalten eines Queues entspricht:

```
function Enqueue(element)
        insert(element, stack1)

function Dequeue()
        for 0 to length(stack1)
                insert(pop(stack1), stack2)
        pop(stack2)
```

Enqueue hat eine *worst case*-Laufzeit von $O(1)$, Dequeue dagegen eine *worst case*-Laufzeit von $O(n)$.

(b) Für `Enqueue`-Operationen wird immer konstante Laufzeit benötigt, da nur ein Schreibvorgang durchgeführt wird. Die `Dequeue`-Operation hängt jedoch von der Anzahl n Operationen ab, da für eine Umkehrung der Ausgabereihenfolge zunächst der gesamte erste Stack in den zweiten überschrieben werden muss. T_n beträgt somit $O(n)$. Die *amortisierte* Laufzeit T_n/n beträgt folglich $O(1)$.

Afg. 3

a) Zur Bestimmung der Minima muss zunächst abgeleitet werden. Statt wie im Skript vorgeschlagen Ketten- und Produktregel zu verwenden wurde an dieser Stelle die Quotientenregel zum ableiten genutzt.

$$f(x) = \frac{x * \ln(n)}{\ln(x)}$$

$$f'(x) = \frac{\ln(n) * \ln(x) - {}^{1}/_{x} * x * \ln(n)}{\ln(x)^{2}}$$

$$= \frac{\ln(n) * \ln(x) - \ln(n)}{\ln(x)^{2}}$$

$$= \frac{\ln(n) * \ln(x)}{\ln(x)^{2}} - \frac{\ln(n)}{\ln(x)^{2}}$$

$$= \frac{\ln(n)}{\ln(x)} - \frac{\ln(n)}{\ln(x)^{2}}$$

Zur Bestimmung der Minima muss die Ableitung nun gegen 0 aufgelöst werden:

$$f'(x) = \frac{\ln(n)}{\ln(x)} - \frac{\ln(n)}{\ln(x)^{2}} = 0 \qquad | * \ln(x)^{2}$$

$$\rightarrow \ln(n) * \ln(x) - \ln(n) = 0 \qquad | + \ln(n)$$

$$\rightarrow \ln(n) * \ln(x) = \ln(n) \qquad | / \ln(n)$$

$$\rightarrow \ln(x) = \frac{\ln(n)}{\ln(n)} = 1$$

$$\rightarrow x = e$$

Um zu beweisen, dass es sich tatsächlich um ein Minimum handelt muss nun der Vorzeichenwechsel überprüft werden:

für alle $x > e$: $\frac{\ln(n)}{\ln(x)} > \frac{\ln(n)}{\ln(x)^{2}}$ und $\frac{\ln(n)}{\ln(x)} - \frac{\ln(n)}{\ln(x)^{2}} > 0$

für alle $x < e$: $\frac{\ln(n)}{\ln(x)} < \frac{\ln(n)}{\ln(x)^{2}}$ und $\frac{\ln(n)}{\ln(x)} - \frac{\ln(n)}{\ln(x)^{2}} < 0$

Die Bedingungen für ein Minimum sind damit erfüllt. Das Minimum der Funktion $f(x) = x * log_{x}(n)$ liegt an der Stelle $(e, e * \ln(n))$.

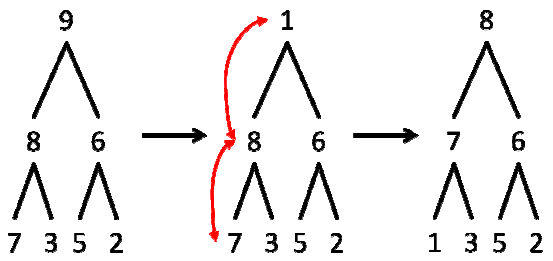
b) Theoretisch wäre e die beste Aufteilungsrate je Knoten. Da ein Knoten aber nur eine ganze Anzahl von Kindern haben kann, müsste die nächstgelegene ganze Zahl gewählt werden: 3. Die worst-case Laufzeit einer Heapify-Operation in einem solchen ternären Heap würde $O\left(3 * log_3\left(10^l\right)\right) \approx (6 * l)$ betragen. Zum Vergleich wäre die Laufzeit bei einem binären Heap: $O\left(2 * log_2\left(10^l\right)\right) \approx (6{,}6 * l)$.

c) In der Praxis wird dennoch hauptsächlich mit binären Bäumen gearbeitet. Ein Grund hierfür könnte z. B. die Tatsache sein, dass bei einer Heapify-Operation im Falle einer verletzten Heap-Eigenschaft und dem damit verbundenem Austausch zunächst das Maximum unter den Kindern des jeweiligen Knotens gefunden werden muss. Bei einem binären Heap wäre dies durch ein einfaches if-/ else-Statments implementierbar ($O(1)$). Sobald $k > 2$ benötigt man jedoch eine Schleife die alle k Elemente überprüft($O(k)$). Dies könnte den geringen Laufzeitvorteil der Heapify-Operation eines ternären Heaps gegenüber eines binären Heaps wieder aufheben.

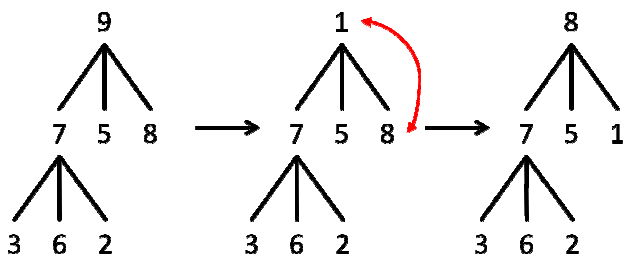
d) In einem beliebigen Max-Heap beträgt die Laufzeit zum Finden des Maximums immer $O(1)$. Grund hierfür ist die angenehme Tatsache, dass das Maximum immer an erster Stelle steht und daher nicht lange gesucht werden muss. Soll dieses Maximum jedoch gegen eine höhere Zahl ausgetauscht werden (durch eine InsertElement-Operation) so wird diese am nächsten freien Knoten angehängt und durch Heapify nach oben durchgereicht (Laufzeit: $O(ld(k))$).

Für die Laufzeit eines Heapifies auf den gesamten Heap würde dies zunächst bedeuten, dass die Suche nach dem jeweiligen Maximum unter den Kindern eines jeden Knotens auf einen Schritt begrenzt wäre (zum Vergleich: Bei einem k-nären Baum müsste jedes Element zunächst überprüft werden was k Schritte beansprucht). Der durch Heapify verursachte Austausch eines Elements hingegen würde extra Arbeit bedeuten, da es nicht nur an den Elternknoten angehängt sondern auch in den binären Heap einsortiert werden müsste (($ld(k)$ Schritte). Da aber $ld(k) < k$ wäre die Laufzeit des Heapifies in einer derartigen Konstruktion insgesamt kürzer.

e)



2 Vertauschungen



1 Vertauschung

f) Die worst-case-Laufzeit für eine Heapify-Operation an der Wurzel eines k-nären Heaps mit n Elementen beträgt $O(k * log_k(n))$. In der Aufgabenstellung wird behaupter, dass für ein beliebiges n gilt:

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

Induktionsanfang ($n = 1$):

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

Behelfsformel:

$\lceil y * log_y(x) \rceil = \left\lceil \frac{y * ln(x)}{ln(y)} \right\rceil$

Induktionsschritt ($n = n + 1$):

$\lceil 3 * log_3(n+1) \rceil \leq \lceil 2 * log_2(n+1) \rceil$

$\triangleq \left\lceil \frac{3 * ln(n+1)}{ln(3)} \right\rceil \leq \left\lceil \frac{2 * ln(n+1)}{ln(2)} \right\rceil$

$\triangleq \left\lceil \ln(n+1) * \frac{3}{ln(3)} \right\rceil \leq \left\lceil \ln(n+1) * \frac{2}{ln(2)} \right\rceil = \left\lceil 2 * \frac{\ln(n+1)}{ln(2)} \right\rceil = \lceil 2 * log_2(n+1) \rceil$