Datenstrukturen und Algorithmen

Abgabe: 24.05.2017

Georg C. Dorndorf Matr.Nr. 366511
Adrian C. Hinrichs Matr.Nr. 367129

Aufgabe 5

Der Array Inhalt nach jeder Partition-Operation.

5	4	0	9	2	1	3	7	6	8

(a) Ausgangslage.

5	4	0	6	2	1	3	7	8	9

(b) Erster Aufruf der Partition-Operation.

5	4	0	6	2	1	3	7	8	9

(c) Zweiter Aufruf der Partition-Operation.

1	2	0	3	4	5	6	7	8	9

(d) Dritter Aufruf der Partition-Operation.

0	2	1	3	4	5	6	7	8	9

(e) Vierter Aufruf der Partition-Operation.

0	1	2	3	4	5	6	7	8	9

(f) Fünfter Aufruf der Partition-Operation.

0	1	2	3	4	5	6	7	8	9

(g) Sechster Aufruf der Partition-Operation.

0	1	2	3	4	5	6	7	8	9

(h) Siebter Aufruf der Partition-Operation.

0	1	2	3	4	5	6	7	8	9

(i) Achter Aufruf der Partition-Operation.

Abbildung 1: Der Quicksort-Algorithmus

Aufgabe 6

Quicksort kann in der Praxis für teilweise oder ganz sortierte Arrays aber auch bei Arrays, die Daten mit Mustern enthalten optimiert werden. Dies ist möglich, indem das PIVOT-Element nicht von den Rändern des Arrays gewählt wird, sondern von pseudozufälligen Stellen. Die asymptotische Komplexität wird dabei nicht verändert, da die Komplexität der Operation des Erzeugens einer Pseudozufallszahl aus $\Theta(1)$ ist.

Dieser Algorithmus ist unter der Vorraussetzung deterministisch, dass der Algorithmus zur Erzeugung der Pseudozufallszahlen deterministisch ist. Da solche Algorithmen existieren, existiert auch der oben beschriebene verbesserte Quicksort.

Aufgabe 7

Ein Stack kann mittels zwei Queues implementiert werden indem man als `push`-Funktion die `enqueue`-Funktion der Schlange verwendet. Als `pop`-Funktion des Stacks nutzt man die `dequeue`-Funktion der Queue und kopiert diese in die zweite Queue wobei man das zu letzt aus der ersten Queue dequeuete Element entfernt und zurückgibt. Dies erreicht man indem man die erste Queue in einer `while(not Queue.isEmpty())`-Schleife dequeuet wobei man immer ein Element zwischenspeichert bevor man das nächste in die zweite Queue enqueuet. Nachdem die Schleife endet gibt man dann das zwischengespeicherte Element zurückgibt. Die zweite Schleife wird für die nächste Operation zur ersten. Die `isEmpty`-Funktion kann man realisieren indem man auf die erste Queue `isEmpty()` aufruft.

Diese Option einen Stack mit zwei Queues zu implementieren hat bei der `push`-Funktion und bei der `isEmpty`-Funktion eine Laufzeitkomplexität von $\Theta(1)$. Die `pop`-Funktion hat eine Laufzeit von $\Theta(n)$, da sie bei jedem Aufruf den gesamten Inhalt des Stacks kopieren muss. Es ist jedoch möglich diese Laufzeitverteilung umzudrehen und dementsprechend die `push`-Funktion mit $\Theta(n)$ Laufzeitkomplexität arbeitend zu implementieren.

Aufgabe 8

a)

Anhand der durchschnittlichen Entfernung der Blätter zur Wurzel:
Sei A die anzahl der druchschnittlichen Vergleiche, B die Menge der Blätter des Entscheidungsbaumes und $e(b), b \in B$ die entfernung des Blattes b zur Wurzel des Baumes.

$$A = \left\lceil \frac{1}{|B|} \sum_b \in Be(b) \right\rfloor^{1} \tag{I}$$

$$\overset{|B|=!n}{=} \left\lceil \frac{1}{n!} \sum_b \in Be(b) \right\rfloor \tag{II}$$

mit n =Anzahl der zu sortierenden Elemente.

[1] $\lceil x \rfloor := \lfloor x + \frac{1}{2} \rfloor \ \forall\, x \in \mathbb{R}$ (≫round half up≪)

b)

Zu zeigen: Die Anzahl der Vergleiche bei einem Vergleichs-basierenden Sortieralgorithmus ist im *Average-Case* $\Omega(n \log n)$.

Beweis: Die durchschnittliche Anzahl an Vergleichen ist der mittelwert der Entfernungen aller Blätter zur Wurzel. Sei b die Anzahl der Blätter eines h hohen Entschiedungsbaumes zu einem Vergleichsbasierendem Sortierungsalgorithmus auf n Elementen. Da jede der $n!$ möglichen Permutationen der Elemte als Blatt im Entscheidungsbaum enthalten sein muss, und die maximale Anzahl an Blättern b' (nur) auf der Untersten Ebene mit $b' = 2^h$ erreicht ist, gilt:

$$b = n! \leq b' = 2^h \tag{III}$$
$$\Leftrightarrow \log_2 n! \leq h \tag{IV}$$
$$\Leftrightarrow h \geq \log_2 n! \overset{Def.}{\in} \Omega(n \log n) \tag{V}$$

QED

c)

Zu zeigen: Ein Binärbaum der Höhe h enthält höchstens $2^{h+1} - 1$ Knoten.

Beweis: (IA):

$$h = 0 \quad 2^{0+1} - 1 = 2^1 - 1 = 2 - 1 = 0 \quad \checkmark$$
$$h = 1 \quad 2^{1+1} - 1 = 2^2 - 1 = 4 - 1 = 3 \quad \checkmark$$

(IV): Gelte die Behauptung für ein festes, aber beliebiges $h \in \mathbb{N}_0$
(IS):

$$h \mapsto h + 1: \quad 2^{h+1+1} - 1 = 2 \cdot 2^{h+1} - 1 \tag{VI}$$
$$= 2 \cdot 2^{h+1} - 2 + 1 \tag{VII}$$
$$= 2(2^{h+1} - 1) + 1 \tag{VIII}$$

Hält man sich den Aufbau eines Binärbaumes vor Augen, ist die gültigkeit dieser Aussage offensichtlich. Die Wurzel eines Binärbaumes der Höhe $h + 1$ hat zwei Teilbäume der Höhe h. Diese haben nach Induktionsvorraussetzung maximal jeweils $2^{h+1} - 1$ Knoten, zuzüglich der Wurzel beläuft es sich also genau auf maximal $2(2^{h+1} - 1) + 1$ Knoten.
Die Behauptung wurde per vollständiger Induktion für alle $h \in \mathbb{N}$ nachgewiesen.

QED

d)

Zu zeigen: Enthält ein Binärbaum n Knoten, so beträgt seine Höhe mindestens $\lceil \log(n + 1) \rceil - 1$.

Beweis: Nach Aufgabenteil c) gilt, dass ein Binärbaum der Höhe h mindestens $2^{h+1} - 1$ Knoten hat. Sei $n' \in \mathbb{N}$ die Anzahl an Knoten eines maximalen Binärbaums mit der Höhe $h' \in \mathbb{N}$:

$$n' = 2^{h'+1} - 1 \tag{IX}$$
$$\Leftrightarrow n' + 1 = 2^{h'+1} \tag{X}$$
$$\Leftrightarrow \log(n' + 1) = h' + 1 \tag{XI}$$
$$\Leftrightarrow \log(n' + 1) = h' + 1 \tag{XII}$$
$$\Leftrightarrow \log(n' + 1) - 1 = h' \tag{XIII}$$

Wenn der Binärbaum nicht maximal ist, ist folglich auch $n+1$ keine Potenz von 2, (mindestens) die Unterste Ebene ist dann nämlich nicht vollständig gefüllt. Also muss $\log(n + 1)$ generell aufgerundet werden. Es ergibt sich also folgende Formel für die minimale Höhe $\overline{h}$ eines Binärbaumes mit n Knoten: $\overline{h} = \lceil \log(n + 1) \rceil - 1$.

QED