Lösungsvorschläge zur 5. bis 7. Übung

Aufgabe 18

zu a)

Bei dieser Variante des Quicksort–Verfahrens wird ein gegebener Bereich von Indizes $a_i \ldots a_j$ wie gewohnt mithilfe des Pivot–Elementes in zwei Bereiche geteilt. Nur der *kleinere* der beiden Teile wird rekursiv, der größere wird *iterativ* (durch äußerste Schleife (siehe Vorlesung)) abgearbeitet. Der kleinere Teil kann jedoch im Worst–Case nur die *Hälfte* der Elemente des gesamten Bereiches enthalten, das heißt der rekursiv zu bearbeitende Teil ist im schlechtesten Fall halb so groß wie der Ausgangsbereich. Die Rekursion muß deshalb nach einer Tiefe von höchstens $O(\log n)$ Aufrufen abbrechen!

zu b)

Das Problem bei der Folge $(4, 4, 4, 4, 1, 4, 4, 4, 4, 4)$ ist, daß wiederholt die 4 als Pivot–Element ausgesucht wird, anschließend viele Elemente mit dem gleichen Schlüssel wie das des Pivot–Elememtes vertauscht werden (eine 4 mit einer anderen 4). Zu guter letzt wird beim rekursiven Aufruf im Worst–Case sehr oft nur ein Element abgespalten, was zu einer quadratischen Laufzeit führt. Man behilft sich, in dem man nach Wahl des Pivot–Elementes den zu sortierenden Bereich dreiteilt: im ersten Bereich stehen alle Elemente, die kleiner als, im zweiten alle Elemente, die gleich, und im dritten alle Elemente, die größer als das Pivotelement sind. Weitersortiert werden nur der erste und der dritte Bereich.

Aufgabe 19

zu a)

Durch diese Abfrage wird die vorhandene Heap–Struktur zerstört, bzw. erst gar keine aufgebaut. Der Schlüssel eines Knotens kann sowohl kleiner, gleich oder auch größer als der Schlüssel einer der Söhne (bzw. beider Söhne) sein.

zu b)

Durch diese Abfrage hat man die genaue Invertierung zum Maximum–Heap, d.h. jeder Knoten des Heap–Baumes hat nur Söhne mit kleinerem oder gleichem Schlüssel. Speziell befindet sich an der Wurzel des Baumes das *Minimum* aller Schlüssel des Baumes.

Aufgabe 20

Die Priority–Queues werden mithilfe eines Maximum–Heaps folgendermaßen implementiert:

- Die Funktion *Maximum(S)* liefert den Schlüssel an der Wurzel des Heaps zurück, der normalerweise in $S[1]$ gespeichert ist. Das geht in $O(1)$. Der Heap wird hierbei *nicht* verändert!

- Die Funktion *Extract–Max(S)* besteht aus einem Aufruf von *Maximum(S)* und dem Löschen des Maximums (wie in der Vorlesung für den Heap beschrieben). Sie hat die gleiche Laufzeit wie das Löschen des Maximums, nämlich $O(\log n)$, da *Maximum(S)* in konstanter Zeit durchgeführt wird.

- Die Prozedur *Insert(S,x)* ist die erste echt neue Funktion. Hierbei wird x im ersten Schritt 'unten rechts' an dem Heap–Baum angefügt. Anschließend wird x solange mit seinem jeweiligen Vater vertauscht, bis dieser größer oder gleich x ist. Man arbeitet sich also im Worst–Case einmal im Baum nach oben bis zur Wurzel, woraus $O(\log n)$ als Laufzeit folgt.

Aufgabe 21

Professor Solomon lügt, denn er hat nicht bedacht, daß in seiner Computer–Umgebung als Spezialfall einer beliebigen Folge auch alle Folgen der Länge n sortiert werden müssen, die jeweils aus *verschiedenen* Schlüsseln bestehen. Die inneren Knoten seines Entscheidungsbaumes, die (im Prinzip) jeweils drei Söhne haben können (für $<$, $=$ und $>$) nützen ihm in diesem Fall nichts, da nur zwei der drei überhaupt belegt sein können. Sein 'trinärer' Baum entartet also für diese spezielle Gruppe von Eingabefolgen zu einem binären Baum, für den der Laufzeitbeweis der Vorlesung gilt. Demnach ist die untere Schranke gleich!

Aufgabe 23

Die meisten Lösungsvorschläge beruhten auf der Annahme, daß die Folge

$$x_1 < y_1 < x_2 < y_2 < \ldots < x_{\frac{n}{2}} < y_{\frac{n}{2}}$$

den Worst–Case darstellt, für den natürlich die Anzahl der Vergleiche leicht zu $n-1$ abzuzählen ist. Dabei wurde jedoch stillschweigend angenommen, daß es sich bei dem Mischverfahren um das von Mergesort handelt, das nur jeweils die Elemente an den Folgenanfängen vergleicht. Gesucht war jedoch ein Beweis, der *unabhängig* vom Mischverfahren ist. Dieser scheint nicht ganz untrivial zu sein. Die Aufgabe wurde deshalb aus der Wertung genommen.

Aufgabe 24

Wenn man in einer Folge von Schlüsseln ein Element 'um eine Stelle bewegt' ist das mit der Vertauschung zweier benachbarter Schlüsseln äquivalent, also einer Inversion. Jeder Sortieralgorithmus, der mithilfe von Inversionen die Folge sortiert, muß deswegen midenstens so viele Inversionen durchführen, wie am Anfang in der Folge vorhanden sind. Aus einer früheren Übung ist bekannt, daß die komplett invertierte Folge $O(n^2)$ Inversionen hat. Demnach ist die untere Schranke für die Worst–Case–Laufzeit ebenfalls $O(n^2)$.

Aufgabe 27

Sei $a = (a_n a_{n-1} \ldots a_1 a_0)$ eine g-adische Zahl, d.h. es gilt für die Ziffern $a_i \in \{0, 1, \ldots, g-1\}$ (in der Aufgabenstellung fälschlicherweise 'Buchstaben' genannt) und

$$W = \sum_{i=0}^{n} a_i g^i$$

sei der 'Wert' der Zahl. Es soll nun gezeigt werden daß für jede Zahl, die über eine Permutation π aus der Vertauschung von Ziffern entsteht $a' = (a_{\pi(n)} a_{\pi(n-1)} \dots a_{\pi(1)})$, mit $m = g - 1$ gilt:

$$h(a) = a \bmod m \stackrel{!}{=} a' \bmod m = h(a').$$

Dazu braucht man erst einmal zwei Rechenregeln bezüglich der Restbildung:

$$\begin{aligned} (a+b) \bmod m &= (a \bmod m + b \bmod m) \bmod m && (1) \\ (ab) \bmod m &= ((a \bmod m)(b \bmod m)) \bmod m && (2) \end{aligned}$$

Desweiteren gilt:

$$\bigwedge_{i \in N_0} g^i \bmod m = 1 \qquad (3)$$

[Beweis: Verankerung $n = 0$: $g^0 \bmod m = 1 \bmod (g-1) = 1$. Induktionsschritt: Gelte die Aussage für $n \in N_0$:

$$\begin{aligned} g^{n+1} \bmod m &= (g^n(g-1) + g^n) \bmod m \\ &\stackrel{1}{=} (\underbrace{(g^n(g-1) \bmod (g-1)}_{=0} + g^n \bmod (g-1)) \bmod (g-1) \\ &\stackrel{\text{I.V.}}{=} (0+1) \bmod (g-1) = 1] \end{aligned}$$

Sei nun a beliebig gegeben:

$$\begin{aligned} h(a) &= \left(\sum_{i=0}^{n} a_i g^i \right) \bmod m \\ &\stackrel{1}{=} \left(\sum_{i=0}^{n} (a_i g^i \bmod m) \right) \bmod m \\ &\stackrel{2}{=} \left(\sum_{i=0}^{n} (\underbrace{(a_i \bmod m)}_{=a_i}(g^i \bmod m)) \bmod m \right) \bmod m \\ &\stackrel{3}{=} \left(\sum_{i=0}^{n} a_i \bmod m \right) \bmod m \\ &\stackrel{1}{=} \left(\sum_{i=0}^{n} a_i \right) \bmod m \end{aligned}$$

Das bedeutet, daß das Ergebnis der Hash–Funktion nur von der *Summe der Ziffern* der Zahl abhängig ist und nicht von der *Reihenfolge* der Ziffern. □

Aufgabe 29

Bei den Überlegungen zu dieser Aufgabe muß man vorher definieren, welche Datenstrukturen zum Speichern der 'Überläuferlisten' benutzt werden.

Implementation mit Listen

Bei *Search* kann dann die *erfolglose Suche* abgebrochen werden, sobald der Suchschlüssel kleiner (bei aufsteigender Sortierung) als der Schlüssel des aktuellen Listenelementes ist. Die *erfolgreiche Suche* wird nicht beinflußt. Allerdings ist zur Verbesserung der erfolglosen Suche zu sagen, daß es sich hierbei nur um die empirische Laufzeit handelt, die Komplexität von $O(n)$ wird davon nicht berührt. Den Vorteil bei *Search* hat man sich allerdings durch einen größeren Nachteil bei *Insert* erkauft. Dort muß nämlich jetzt in Worst–Case n Schritten der korrekte Einfügepunkt gesucht werden, um die Sortierung zu erhalten. Damit verschlechtert sich die Komplexität von $O(1)$ auf $O(n)$. Für *Delete* gilt das für *Search* gesagte, da das zu löschende Element erst gesucht werden muß und der eigentliche Löschvorgang in $O(1)$ implementiert werden kann.

Implementation mit Feldern

Bei *Search* können nun die effizienten Suchalgorithmen (z.B. Binäre Suche) verwendet und somit die Laufzeit auf $O(\log n)$ reduziert werden. Bei *Insert* ergibt sich allerdings ein ähnliches Problem wie bei den Listen. Zwar kann die korrekte Einfügeposition in $O(\log n)$ gesucht werden, die Laufzeit zum Einfügen eines neuen Elementes beträgt jedoch im Worst–Case $O(n)$ statt $O(1)$ ohne Sortierung. Bei *Delete* muß zuerst in $O(\log n)$ gesucht und dann in $O(n)$ gelöscht werden.

Zusammenfassend kann man sagen, daß mit den bis jetzt bekannten Datenstrukturen die Laufzeit des Hash–Algorithmus nicht durch Sortierung verbessert werden kann.

Aufgabe 30

Zuerst zwei Vorbemerkungen:

- Es ist klar, daß für eine 'schlechte' Hash–Funktion, die zum Beispiel die Schlüssel nur auf konstant viele der m vorhandenen Fächer verteilt und den Rest leer läßt, die Aussage klar ist. Denn in diesem Fall müssen die langen Ketten der belegten Fächer, die jeweils bis zu $O(n)$ Elementen enthalten, ganz bis zum Ende durchsucht werden. Für die Aufgabe wird daher angenommen, daß es sich um eine ideale Hash–Funktion handelt, die die Schlüssel gleichmäßig auf die m Fächer verteilt.

- Im hypotetischen Fall, daß bereits alle $|U|$ Schlüssel verteilt worden sind, befinden sich in jedem Fach ca. $\frac{|U|}{m}$ Schlüssel, was gleichzeitig auch die größte erlaubte Anzahl von Schlüsseln pro Fach ist, ohne die Gleichverteilung zu verletzen.

Nun sind laut Aufgabenstellung n Schlüssel zu verteilen, mit der Bedingung

$$n < \frac{|U|}{m}.$$

Man kann also mit dieser Anzahl von Schlüsseln *ein* Fach *komplett* füllen, ohne dabei die Gleichverteilung der idealen Hash–Funktion zu verletzen. Das dedeutet aber auch, daß im Worst–Case wiederum nur ein Fach belegt ist und bei *Search* die gesamte Liste durchsucht werden muß. Da die Liste die Länge n hat, beträgt somit die Worst–Case Laufzeit ebenfalls $\Theta(n)$.

Anders überlegt: versucht man mehr als $\frac{|U|}{m}$ Schlüssel unterzubringen, so kann man nicht mehr an die erste Kette anfügen, denn dann wäre die Gleichverteilung verletzt. Somit muß eine zweite Kette angefangen werden, die aber auch wie alle folgenden die Maximallänge von $\frac{|U|}{m}$ nicht übersteigen darf. Die maximale Länge ist somit im Worst–Case ab dem $\frac{|U|}{m}$.–ten Schlüssel *unabhängig* von der Anzahl der Schlüssel!