Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Departement Informatik
Markus Püschel
Peter Widmayer
Thomas Tschager
Tobias Pröger

27. Oktober 2016

Datenstrukturen & Algorithmen Lösungen zu Blatt 5 HS 16

Lösung 5.1 *Suche nach gleichen Zahlen.*

a) Wir vergleichen jedes Element a in A mit jedem Element b in B. Stellen wir dabei fest, dass $a = b$ gilt, dann geben wir das entsprechende Element aus. Da jedes der m Elemente in A mit jedem der n Elemente in B verglichen wird, führt dieses naive Verfahren immer $m \cdot n$ viele Vergleiche aus.

b) Ein besseres Verfahren funktioniert analog zum Verschmelzen zweier Arrays in Mergesort. Dabei betrachten wir die Elemente in jedem Array in aufsteigender Reihenfolge. Wir speichern zwei Zahlen i und j, die das aktuell betrachtete Element in A bzw. in B angeben (initial, $i = j = 1$). Nun vergleichen wir in jedem Schritt $A[i]$ mit $B[j]$ und unterscheiden drei Fälle:

   1. Fall ($A[i] = B[j]$): Dann haben wir zwei Elemente gefunden, die sowohl in A als auch in B vorkommen. Wir geben das entsprechende Element aus und erhöhen sowohl i als auch j um 1 (da jedes Element sowohl in A als auch in B höchstens einmal vorkommt, kann $A[i]$ mit keinem anderen $B[j']$ ausser $B[j]$ übereinstimmen, und analog kann $B[j]$ mit keinem $A[i']$ ausser $A[i]$ übereinstimmen).

   2. Fall ($A[i] < B[j]$): Da die Elemente in B aufsteigend sortiert sind, kann $A[i]$ mit keinem Element $B[j']$ für $j' \geq j$ übereinstimmen. Also können wir mit dem nächsten Element von A fortfahren, wir erhöhen also i um 1.

   3. Fall ($A[i] > B[j]$): Da die Elemente in A aufsteigend sortiert sind, kann $B[j]$ mit keinem Element $A[i']$ für $i' \geq i$ übereinstimmen. Also können wir mit dem nächsten Element von B fortfahren, wir erhöhen also j um 1.

   Das Verfahren endet, sobald $i = m + 1$ oder $j = n + 1$ erreicht wurden. Da in jedem Schritt entweder i oder j (oder sogar beide) um 1 erhöht werden, endet das Verfahren nach weniger als $n + m$ Schritten. Da die Vergleiche und Rechnungen in einem einzelnen Schritt in konstanter Zeit durchgeführt werden können, folgt eine Laufzeit von $\mathcal{O}(m + n)$.

Lösung 5.2 *Erweiterte Heaps.*

In der Vorlesung wurde die Funktion RESTORE-HEAP-CONDITION vorgestellt, die den Knoten eines Schlüssels k so lange mit dem kleineren der beiden Nachfolgerknoten vertauscht, bis *beide* Nachfolger einen Schlüssel grösser gleich k haben, oder k keine Nachfolger mehr besitzt. Analog dazu definieren wir eine Funktion BOTTOM-UP, die den Knoten eines Schlüssels k so lange mit seinem Vorgänger vertauscht, bis entweder der Vorgänger kleiner gleich k ist, oder k in der Wurzel des Heaps gespeichert ist. Sei A das Array, das den Heap speichert, und i die initiale Position von k in A. In Pseudocode ergibt sich dann die folgende Funktion:

BOTTOM-UP(A, i)

```
1 while i ≥ 2 do                          ▷ A[i] hat einen Vorgänger
2     j ← ⌊i/2⌋                           ▷ A[j] ist Vorgänger
3     if A[j] ≤ A[i] then STOP            ▷ Heap-Bedingung erfüllt: Abbruch
4     else Vertausche A[i] und A[j]; i ← j
```

Zusätzlich speichern wir noch eine Variable n, die die aktuelle Anzahl der Schlüssel im Heap angibt und mit 0 initialisiert wird. Da im obigen Algorithmus $i \in \{1, \ldots, n\}$ ist, ist seine Laufzeit durch $\mathcal{O}(\log n)$ nach oben beschränkt.

a) MIN: A ist ein Min-Heap. Also liefert MIN einfach $A[1]$ zurück (bzw. eine Fehlermeldung, falls $n = 0$ ist). Die Laufzeit ist konstant, d.h. in $\mathcal{O}(1)$.

b) REPLACE(i, k): Sei $k' = A[i]$ der zu ersetzende Schlüssel. Wir unterscheiden nun zwei Fälle:

1. Fall: $k < k'$. Dann ist der am Knoten $A[i]$ gespeicherte Teilbaum auch nach der Substitution noch ein (Min-)Heap (nur das kleinste Element ist kleiner geworden). Allerdings muss $A[1..n]$ insgesamt kein Heap mehr sein, weil k selbst kleiner als sein Vorgänger sein kann. Zur Wiederherstellung der Heap-Eigenschaft rufen wir BOTTOM-UP(A, i) auf.

2. Fall: $k > k'$. Dann ist der am Knoten $A[i]$ gespeicherte Schlüssel auch nach der Substitution noch grösser als sein Vorgänger, aber der am Knoten $A[i]$ gespeicherte Teilbaum ist u.U. kein Heap mehr (k kann grösser als seine Nachfolger sein). Zur Wiederherstellung der Heap-Eigenschaft rufen wir RESTORE-HEAP-CONDITION(A, i) auf.

Somit kann die Operation REPLACE(i, k) in Zeit $\mathcal{O}(\log n)$ implementiert werden.

c) INSERT(k): Wir erhöhen n um 1 und speichern den neuen Schlüssel k im Eintrag $A[n]$. Nun kann es passieren, dass A kein Min-Heap mehr ist, da der neu eingefügte Schlüssel u.U. kleiner als sein Vorgänger ist. Zur Wiederherstellung der Heap-Eigenschaft ist es aber ausreichend, BOTTOM-UP(A, n) aufzurufen. Auch hier ist Laufzeit in $\mathcal{O}(\log n)$.

d) DELETE(i): Wir rufen REPLACE$(i, A[n])$ auf, d.h., der an Stelle i gespeicherte Schlüssel wird durch $A[n]$ ersetzt. Nun kommt der Schlüssel $A[n]$ einmal zu viel vor, also dekrementieren wir n um 1 (die Grösse des Heaps verringert sich dann um 1 und das überflüssige Vorkommen von $A[n]$ wird "abgeschnitten"). Die Operation hat die gleiche Laufzeit wie REPLACE(i, k), liegt also in $\mathcal{O}(\log n)$.

Lösung 5.3 *Suchen in sortierten Arrays.*

a) Die Interpolationssuche ist dann wesentlich schneller, wenn die Zahlen im Array möglichst gleichmässig wachsen. Ein Beispiel ist das Array $A = (1, 2, \ldots, n)$. Der Schlüssel i ist an der Position i gespeichert (das Array ist von 1 bis n indiziert). Eine Interpolationssuche nach dem Schlüssel b setzt zu Beginn

$$\text{middle} = \left\lfloor \text{left} + \frac{b - A[\text{left}]}{A[\text{right}] - A[\text{left}]} \cdot (\text{right} - \text{left}) \right\rfloor = \left\lfloor 1 + \frac{b-1}{n-1}(n-1) \right\rfloor = b$$

und findet daher b nach exakt einem Vergleich. Die binäre Suche würde für viele Wahlen von b den Suchbereich mehrfach halbieren, bis b gefunden wird.

Die Interpolationssuche ist dann wesentlich langsamer als die binäre Suche, falls die Schlüssel nicht annähernd linear wachsen. Betrachten wir z.B. das Array $A = (1!, 2!, \ldots, n!)$. Wir

zeigen nun folgendes: Eine Interpolationssuche nach dem Schlüssel $b = (n-1)!$ betrachtet mindestens $\lfloor n/2 \rfloor$ viele Schlüssel. Dazu zeigen wir, dass in jedem Schritt der linke Rand left höchstens um 2 wächst, und der rechte Rand immer n bleibt.

Der Einfachheit halber schreiben wir l statt left. Die Interpolationssuche berechnet

$$\begin{aligned}\text{middle} &= \left\lfloor \text{left} + \frac{b - A[\text{left}]}{A[\text{right}] - A[\text{left}]} \cdot (\text{right} - \text{left}) \right\rfloor = \left\lfloor l + \frac{(n-1)! - l!}{n! - l!} \cdot (n-l) \right\rfloor \\ &\leq \left\lfloor l + \frac{(n-1)!}{n! - \frac{1}{2}n!} \cdot (n-l) \right\rfloor = \left\lfloor l + \frac{2}{n} \cdot (n-l) \right\rfloor = \left\lfloor l + 2 - \frac{2l}{n} \right\rfloor \in \{l, l+1\}\end{aligned}$$

und betrachtet $A[\text{middle}]$ (also $A[l]$ oder $A[l+1]$). Ausser für $\text{middle} = n-1$ stellt man dann fest, dass $A[\text{middle}] < (n-1)!$ ist, folglich wird $\text{left} \leftarrow \text{middle} + 1$ gesetzt. Der Wert von left wächst also höchstens um 2 (und right bleibt unverändert auf n). Eine binäre Suche würde lediglich $\mathcal{O}(\log n)$ viele Schlüssel betrachten.

b) Die Hauptidee ist, die Interpolationssuche geschickt mit der binären Suche zu kombinieren. Dazu berechnen wir alternierend ein Pivotelement mit der Interpolationssuche, und eines mit der binären Suche. Für die pathologischen Fälle, in denen die Interpolationssuche lineare Zeit benötigt, wird das Suchintervall trotzdem in jedem zweiten Schritt durch die binäre Suche halbiert.

Somit hat dieser Algorithmus maximal die doppelte Anzahl von Iterationen wie die binäre Suche alleine, und auch maximal die doppelte Anzahl von Iterationen der Interpolationssuche. Insbesondere führt dieser modifizierte Algorithmus höchstens doppelt so viele Schritte aus wie das Minimum der Schritte der binären Suche und der Interpolationssuche.