Eidgenössische
Technische Hochschule
Zürich

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

Departement Informatik
Markus Püschel David Steurer Peter Widmayer
Chih-Hung Liu

6. November 2017

Algorithmen & Datenstrukturen Blatt 6 HS 17

Lösung 6.1 *Minima und Maxima finden.*

a) Wir verwalten einen Zeiger $\vec{p}$ auf den bisher kleinsten gefundenen Schlüssel, der initial auf $A[1]$ zeigt. Nun betrachten wir die Schlüssel $A[i]$ für $i = 2, 3, \ldots, n$. In jedem Schritt prüfen wir, ob $A[i]$ kleiner als der Schlüssel ist, auf den p zeigt. Falls ja, lassen wir $\vec{p}$ neu auf $A[i]$ zeigen. Am Ende des Verfahrens zeigt $\vec{p}$ auf den kleinsten in A gespeicherten Schlüssel.

Da für jedes $i \in \{2, 3, \ldots, n\}$ genau ein Vergleich zweier Schlüssel vorgenommen wird, führt der Algorithmus insgesamt $n - 1$ Schlüsselvergleiche aus.

b) Wir beobachten zunächst, dass bereits ein einziger Vergleich zweier Schlüssel $A[i]$ und $A[j]$ Informationen über das Minimum und das Maximum in A liefert: Nur der kleinere der beiden Schlüssel ist ein Kandidat für das Minimum in A, und nur der grössere der beiden Schlüssel ist ein Kandidat für das Maximum in A.

Wir verwalten nun zwei Mengen S_{min} und S_{max}, die Kandidaten für den kleinsten bzw. den grössten Schlüssel in A enthalten. Die Mengen sind initial leer. Nun vergleichen wir jeweils $A[i]$ mit $A[i+1]$ für $i = 1, 3, 5, \ldots$, und fügen S_{min} jeweils den kleineren und S_{max} jeweils den grösseren der beiden Schlüssel hinzu. Bei Gleichheit wird einer der entsprechenden Schlüssel sowohl S_{min} als auch S_{max} hinzugefügt. Ist n ungerade, dann wird $A[n]$ ebenfalls sowohl S_{min} als auch S_{max} hinzugefügt. Danach suchen wir mit dem Algorithmus in a) den kleinsten Schlüssel in S_{min}, und mit einem entsprechend angepassten Algorithmus den grössten Schlüssel in S_{max}.

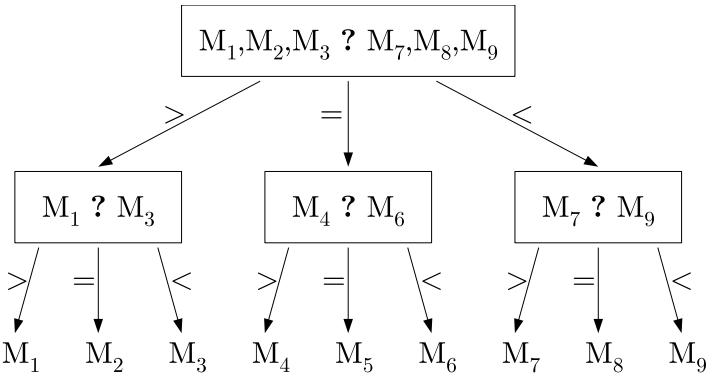
Zur Berechnung der Mengen S_{min} und S_{max} werden $\lfloor n/2 \rfloor$ Schlüssel miteinander verglichen. Die beiden Mengen enthalten genau $\lceil n/2 \rceil$ Schlüssel, und zur Berechnung des kleinsten bzw. grössten Schlüssels werden entsprechend $\lceil n/2 \rceil - 1$ Schlüssel miteinander verglichen. Der Algorithmus führt also insgesamt

$$\lfloor n/2 \rfloor + \lceil n/2 \rceil - 1 + \lceil n/2 \rceil - 1 = \lceil (3/2) \cdot n \rceil - 2 \tag{1}$$

Schlüsselvergleiche aus.

Lösung 6.2 *Münzen mit Balkenwaage wägen.*

a) Seien die Münzen $M_1, \ldots, M_9$ gegeben. Der folgende Ablaufbaum beschreibt eine Strategie, die in jedem Fall mit zwei Wägungen die falsche Münze ermittelt. In jedem Knoten ist vermerkt, welche Münzen in den Schalen links und rechts liegen. Nach jeder Wägung gibt es drei mögliche Ausgänge: Die Münzen links sind leichter ($<$), gleich schwer ($=$) oder schwerer ($>$) als die Münzen rechts. Nach jedem Schritt befindet sich die falsche Münze in der Menge mit höchstem Gewicht.



b) Seien die Münzen $M_1, \ldots, M_n$ gegeben. Für $n = 1$ haben wir die falsche Münze gefunden und sind fertig. Ansonsten teilen wir die Münzen in drei Gruppen $G_1 := \{M_1, \ldots, M_{n/3}\}$, $G_2 := \{M_{n/3+1}, \ldots, M_{2n/3}\}$ und $G_3 := \{M_{2n/3+1}, \ldots, M_n\}$ ein, legen G_1 in die linke Schale und G_2 in die rechte Schale. Sind die Münzen links schwerer, dann befindet sich die gesuchte Münze in G_1 und wir fahren mit dieser Menge fort. Analog wird mit G_2 fortgefahren wenn die rechte Seite schwerer ist, oder mit G_3, falls beide Seiten gleich schwer sind.

Es verbleibt zu zeigen, dass auf diese Weise wirklich nur $\log_3(n)$ Wägungen anfallen. Dazu zeigen wir mit vollständiger Induktion über $m \in \mathbb{N}$ die folgende Aussage: Für $n = 3^m$ benötigt das obige Verfahren genau m Wägungen, um die falsche Münze zu finden.

Induktionsverankerung ($m = 1$): Für $m = 1$ ist $n = 3$, und es wird genau eine Wägung benötigt um die falsche Münze zu finden.

Induktionsannahme: Sei die Aussage wahr für m, d.h. für $n = 3^m$ benötige das obige Verfahren genau m Wägungen, um die falsche Münze zu finden.

Induktionsschluss ($m \to m + 1$): Es sei $n = 3^{m+1}$. Wir teilen die Münzen in drei gleich grosse Gruppen ein und fahren mit der schwersten Menge fort. Diese enthält genau

$$n/3 = 3^{m+1}/3 = 3^m \tag{2}$$

viele Münzen, und nach Induktionsannahme genügen m Wägungen, um die falsche Münze zu finden. Damit benötigt das Verfahren insgesamt nur $m + 1$ Wägungen.

Wir beobachten nun, dass für $m = \log_3(n)$ grundsätzlich $n = 3^m$ gilt, also werden nur $\log_3(n)$ Wägungen benötigt.

Lösung 6.3 *Erweiterte Heaps.*

Wir definieren unsere Datenstruktur auf vier Arrays A, B, C, D, wobei jeweils A ein Max-Heap und B ein Min-Heap aller Elemente in der Datenstruktur sein werden. Wenn wir nun ein Element aus der Datenstruktur extrahieren möchten, müssen wir es also in beiden Heaps löschen. Und analog, wenn wir ein Element in die Datenstruktur einfügen möchten, müssen wir es in beide Heaps einfügen. Arrays C und D benötigen wir, damit wir jeweils ein Element eines Heaps in konstanter Zeit auch im anderen Heap finden. Wir nehmen an, dass für $C[i] = j$ gilt, dass $A[i] = B[j]$, und analog für $D[i] = j$ gilt, dass $B[i] = A[j]$. Zusätzlich speichern wir noch eine Variable n, die die aktuelle Anzahl der Schlüssel in der Datenstruktur angibt.

Initialisierung: Um die Datenstruktur aufzubauen, kopieren wir als erstes A in B und initialisieren C und D, so dass $C[i] = D[i] = i$ für alle $i \in \{1, \ldots, n\}$. Nun bilden wir aus den

n Elementen im gegebenen Array A wie in der Vorlesung gesehen einen Max-Heap. Dabei vertauschen wir bei einem Swap von zwei Elementen $A[i]$ und $A[j]$ auch die jeweiligen Einträge in C und D: $C[i]$ wird mit $C[j]$ vertauscht und $D[C[i]]$ mit $D[C[j]]$. Anschliessend bilden wir aus B einen Min-Heap. Auch hier vertauschen wir bei einem Swap von zwei Elementen $B[i]$ und $B[j]$ auch die jeweiligen Einträge in C und D: $C[D[i]]$ wird mit $C[D[j]]$ vertauscht und $D[i]$ mit $D[j]$. Auch mit dieser Modifikation beträgt die Zeit zur Vertauschung zweier Elemente weiterhin $\mathcal{O}(1)$.

Funktionen: In der Vorlesung wurde die Funktion RESTORE-HEAP-CONDITION vorgestellt, die den Knoten eines Schlüssels k so lange mit dem grösseren der beiden Nachfolgerknoten vertauscht, bis *beide* Nachfolger einen Schlüssel kleiner gleich k haben, oder k keine Nachfolger mehr besitzt. Um nun zwischen Min-Heap und Max-Heap zu unterscheiden, benennen wir diese Funktion einmal um zu RESTORE-MAX-HEAP-CONDITION und ändern sie einmal zu RESTORE-MIN-HEAP-CONDITION, wobei die Ungleichzeichen in den Zeilen 4 und 5 jeweils umgedreht werden. Ausserdem erfolgt jedes Vertauschen zweier Elemente wie in der Initialisierung beschrieben (d.h., die in den Arrays C und D gespeicherten Indizes müssen geeignet mitvertauscht werden). So eine Vertauschung benötigt konstante Zeit, also bleibt die Laufzeit von RESTORE-MAX-HEAP-CONDITION und RESTORE-MIN-HEAP-CONDITION in $\mathcal{O}(\log n)$.

Analog dazu definieren wir eine Funktion BOTTOM-UP-MAX (bzw. BOTTOM-UP-MIN), die den Knoten eines Schlüssels k so lange mit seinem Vorgänger vertauscht, bis entweder der Vorgänger grösser (bzw. 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-MAX(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
```

Auch hier erfolgt jedes Vertauschen wie in der Initialisierung beschrieben. Da im obigen Algorithmus $i \in \{1, \ldots, n\}$ ist, ist seine Laufzeit durch $\mathcal{O}(\log n)$ nach oben beschränkt. BOTTOM-UP-MIN hat in Zeile 3 ein "$\geq$"-Zeichen.

(i) EXTRACT-MAX: A ist ein Max-Heap. Also liefert EXTRACT-MAX einfach $A[1]$ zurück (bzw. eine Fehlermeldung, falls $n = 0$ ist). Danach wird das Element gelöscht sowohl in A als auch in B. Dazu wird zuerst der Index $j = C[1]$ zwischengespeichert, das ist die Position des Maximums im Min-Heap. Jetzt wird $A[1]$ mit $A[n]$ vertauscht und mit RESTORE-MAX-HEAP-CONDITION$(A, 1, n-1)$ wird die Heap-Eigenschaft in A wieder hergestellt. Nun wird $k' = B[j]$ mit $k = B[n]$ vertauscht. Da k' das Maximum ist, gilt $k < k'$. Nun kann es sein, dass $B[1..n-1]$ insgesamt kein Heap mehr ist, weil k möglicherweise kleiner als sein (neuer) Vorgänger ist. Zur Wiederherstellung der Heap-Eigenschaft wird BOTTOM-UP-MIN(B, j) aufgerufen. Schliesslich wird n um 1 reduziert. Somit kann die Operation EXTRACT-MAX(i, k) in Zeit $\mathcal{O}(\log n)$ implementiert werden.

(ii) EXTRACT-MIN: Diese Funktion ist analog zu EXTRACT-MAX. Hier wird $B[1]$ ausgegeben und es werden $B[1]$ und $A[D[1]]$ gelöscht. Die Laufzeit ist in $\mathcal{O}(\log n)$.

(iii) INSERT(k): Wir erhöhen n um 1 und speichern den neuen Schlüssel k im Eintrag $A[n]$ und $B[n]$. Ausserdem setzen wir $C[n] = D[n] = n$. Nun kann es passieren, dass A kein

Max-Heap mehr ist, da der neu eingefügte Schlüssel u.U. grösser als sein Vorgänger ist. Zur Wiederherstellung der Heap-Eigenschaft ist es aber ausreichend, BOTTOM-UP-MAX(A, n) aufzurufen. In B rufen wir analog dazu BOTTOM-UP-MIN(B, n) auf. Auch hier ist die Laufzeit in $\mathcal{O}(\log n)$.

Lösung 6.4 *Zahlensummen.*

a) Zunächst sortieren wir A. Danach prüfen wir für jede mögliche Wahl von $A[i]$ (es gibt n Kandidaten), ob die Zahl $z - A[i]$ im Array vorkommt (in diesem Fall hätten wir zwei Schlüssel gefunden, deren Summe z beträgt). Da A sortiert ist, können wir mittels binärer Suche in $\mathcal{O}(\log n)$ Schritten feststellen, ob $z - A[i]$ in A vorkommt. Ist dies der Fall, dann haben wir eine Lösung gefunden, ansonsten probieren wir die nächste Wahl von $A[i]$ aus, usw. Damit ergibt sich ein Algorithmus mit Kosten in $\mathcal{O}(n \log n)$.

b) Eine erste Idee funktioniert wie folgt: Für jede Wahl von $A[k]$ benutzen wir den Algorithmus aus a), um zu entscheiden, ob A zwei Schlüssel $A[i]$ und $A[j]$ enthält, deren Summe $z' := z - A[k]$ ergibt. Die Laufzeit des Algorithmus beträgt im schlechtesten Fall $\mathcal{O}(n^2 \log n)$. Natürlich muss man das Array nur einmal sortieren (und nicht n Mal), aber trotzdem werden $\Theta(n^2)$ viele binäre Suchen (mit Laufzeit $\mathcal{O}(\log n)$ ausgeführt.

Die Entscheidung, ob A zwei Schlüssel $A[i]$ und $A[j]$ enthält, deren Summe z' ergibt, dauert mit dem Algorithmus aus a) Zeit $\mathcal{O}(n \log n)$, selbst wenn A bereits sortiert ist. Schneller geht es mit folgendem Verfahren: Seien l, r die Indizes des linken bzw. rechten Endes des Arrays (initial: $l = 1$ und $r = n$). Falls $A[l] + A[r] = z'$, geben wir $A[l]$ und $A[r]$ aus und beenden das Verfahren. Wenn nun $A[l] + A[r] > z'$, dann gilt sicher $A[k] + A[r] > z'$ für jedes k mit $l \leq k \leq r$ (es gilt $A[k] \geq A[l]$, da A sortiert ist). Folglich gibt es im Array von Position l bis Position r keine Zahl, die zu $A[r]$ zusammengezählt genau z' ergibt. Es genügt also, im Array nur Elemente mit Index $\leq r - 1$ zu berücksichtigen. Daher setzen wir in diesem Fall $r \leftarrow r - 1$ und wiederholen das Verfahren.

Ist dagegen $A[l] + A[r] < z'$, dann gilt sicher $A[l] + A[k] < z'$ für jedes k mit $l \leq k \leq r$. Folglich gibt es im Array von Position l bis Position r keine Zahl, die zu $A[l]$ zusammengezählt genau z' ergibt (es gilt $A[k] \leq A[r]$, da A sortiert ist). Also genügt es, im Array nur Elemente mit Index $\geq l + 1$ zu berücksichtigen. Daher setzen wir in diesem Fall $l \leftarrow l + 1$ und wiederholen das Verfahren.

Wir brechen ab, falls entweder ein Paar gefunden wurde, oder falls $l = r$. Wir finden so immer ein Paar $A[i], A[j]$ mit Summe z', falls ein solches existiert. Die Laufzeit von $\mathcal{O}(n)$ können wir beweisen, indem wir den Verlauf von $r - l$ betrachten: In jedem Schritt, in dem das Verfahren nicht abbricht, wird entweder r um eins verringert oder l um eins erhöht (aber niemals beides gleichzeitig). Das heisst, $r - l$ wird in jedem Schritt eins kleiner. Ursprünglich ist $r - l = n - 1$, und somit terminiert das Verfahren nach höchstens $n - 1$ Schritten.

Dieser Algorithmus wird nun für jedes $A[k]$, $k = 1, \ldots, n$ angewendet, um zu prüfen ob A zwei Elemente $A[i]$ und $A[j]$ mit Summe $z - A[k]$ enthält. Die Gesamtlaufzeit liegt dann in $\mathcal{O}(n^2)$.