Albert-Ludwigs-Universität
Institut für Informatik
Prof. Dr. Fabian Kuhn
P. Schneider, P. Bamberger, Y. Maus

Informatik 2 - Sommersemester 2018
Musterlösung Übungsblatt 4

Abgabe: Montag, 28. Mai, 14:00 Uhr

Aufgabe 1: Fixpunktsuche *(7 Punkte)*

Gegeben sei ein *aufsteigend sortiertes Array* A der Länge n gefüllt mit *paarweise verschiedenen* ganzen Zahlen. Unser Ziel ist die Existenz eines Fixpunktes nachzuweisen oder zu widerlegen. Ein Fixpunkt ist ein Index $i \in \{0, \dots, n-1\}$ für den $A[i] = i$ gilt. Wir schlagen den folgenden Algorithmus vor, welcher den Index eines Fixpunktes zurück gibt sofern dieser existiert und sonst n.

Algorithm 1: `FindFixedPoint`(A)

```
ℓ ← 0; r ← n−1
while ℓ ⩽ r do
    p ← ⌊(ℓ+r)/2⌋
    if A[p] < p then
        ℓ ← p+1
    else if A[p] > p then
        r ← p−1
    else
        return p
return n
```

(a) Geben Sie die Worst-Case Laufzeit $T(n)$ als Rekursions(un)gleichung an. Sie dürfen annehmen dass jeder Schleifendurchlauf eine einzige Zeiteinheit benötigt. Geben Sie außerdem die asymptotische Laufzeit an. *(2 Punkte)*

(b) Formulieren Sie Vorbedingung und Nachbedingung der Schleife, sowie die Schleifeninvariante bezüglich der Existenz eines Fixpunktes im Suchbereich $\{\ell, \dots, r\}$. Beweisen Sie die Schleifeninvariante formal induktiv. Leiten Sie aus der Schleifeninvariante die Korrektheit her. *(5 Punkte)*

Musterlösung

(a) Der Worst-Case bezüglich der Laufzeit tritt auf wenn es keinen Fixpunkt gibt. Die Laufzeit von `FindFixedPoint`(A) wird maßgeblich von der Größe des Suchbereichs $r - \ell$ bestimmt, und der Algorithmus terminiert spätestens wenn $\ell > r$ (im besagten Worst-Case). In jedem Schritt wird der Suchbereich *mindestens* halbiert (das "Pivot" p ist in der nächsten Iteration vom Suchbereich ausgenommen). Wir haben also $T(n) \leqslant T(\frac{n}{2}) + 1$ und im Basisfall $n \leqslant 0 : T(n) = 1$. Die asymptotische Laufzeit ist, wie in der Vorlesung bereits hergeleitet, $T(n) \in \mathcal{O}(\log n)$.

(b) **Vorbedingung:** Array A ist sortiert und enthält paarweise verschiedene ganze Zahlen. *($\frac{1}{2}$ Punkte)*
Schleifeninvariante: Wenn es einen Fixpunkt gibt befindet er sich im Bereich $\{\ell, \dots, r\}$. *(1 Punkt)*
Nachbedingung: Wenn es einen Fixpunkt gibt geben wir einen zurück, sonst n. *($\frac{1}{2}$ Punkte)*
Wir beweisen induktiv, dass die Schleifeninvariante nach jeder Iteration gilt.

Induktionsanfang: Offensichtlich gilt sie zu Beginn des Algorithmus.

Induktionsvoraussetzung: Wir gehen davon aus dass die Invariante im Bereich $\{\ell, \ldots, r\}$ gilt.

Induktionsschritt: Im Fall $A[p] = p$ haben wir einen Fixpunkt im Bereich $\{\ell, \ldots, r\}$ gedunden wodurch die Schleifeninvariante offensichtlich efüllt ist. Betrachte den Fall $A[p] > p$ ($A[p] < p$ geht analog). Wir haben zu zeigen dass die Schleifeninvariante für die neuen Werte von ℓ bzw. r gelten. Wir beobachten, dass die Werte im Array mindestens genauso schnell steigen wie die zugehörigen Indizes, da erstere aufsteigend sortiert, ganzzahlig und paarweise verschieden sind. Das heißt wenn schon der Wert $A[p]$ größer als der zugehörige Index p ist bleibt dass auch bei allen Indizes rechts von p so. Wenn es also nach Induktionsvoraussetzung einen Fixpunkt in $\{\ell, \ldots, r\}$ gibt, kann er nicht im Breich $\{p+1, \ldots, r\}$ sein sondern muss im Bereich $\{\ell, \ldots, p-1\}$ sein.

Korrektheit: Da der Suchbereich stets einen Fixpunkt enthält sofern dieser existiert und dabei immer kleiner wird, finden wir einen Fixpunkt falls einer existiert. *(3 Punkte)*

Aufgabe 2: Schlechte Hashfunktionen *(5 Punkte)*

Sei m die Größe einer Hashtabelle und $n \gg m$ der größte mögliche Schlüssel der Datenelemente die wir in dieser Hashtabelle speichern möchten. Die folgenden "Hashfunktionen" sind aus unterschiedlichen Gründen schlecht gewählt. Erläutern sie für jede Hashfunktion kurz warum das so ist.

(a) $h_1 : k \mapsto k$. *(1 Punkt)*

(b) $h_2 : k \mapsto \lfloor \frac{k}{n} \cdot (m-1) \rfloor$. *(1 Punkt)*

(c) $h_3 : k \mapsto 2 \cdot \big(k \bmod \lfloor \frac{m-1}{2} \rfloor\big)$. *(1 Punkt)*

(d) $h_4 : k \mapsto$ `random`(m), $\big($`random`(m) ist eine gleichverteilt zufällige Zahl aus $\{0, \ldots, m-1\}\big)$. *(1 Punkt)*

(e) $h_5 : k \mapsto$ `chaos`(m), $\big($`chaos`(m) berechnet einen guten Hashwert aus $\{0, \ldots, m-1\}$ deterministisch in $\mathcal{O}(\log n)$ vielen Zeitschritten$\big)$. *(1 Punkt)*

Musterlösung

(a) Es handelt sich nicht einmal um eine Hashfuntion da diese nicht auf den Adressraum abbildet sondern auf eine echte Obermenge davon.

(b) Die Funktion streut nicht gut, d.h. wenn k_1, k_2 nahe beieinander liegen dann auch $h_2(k_1), h_2(k_2)$. Da $n \gg m$ werden recht viele aufeinanderfolgende Werte sogar auf die gleiche Adresse abgebildet.

(c) Die Funktion ist nicht surjektiv, schöpft also nicht den gesamten Adressraum aus (bspw. wird die letzte Speicheradresse $m-1$ nie genutzt und auch sonst nur gerade Speicherzellen). Für $m = 1, 2$ ist sie außerdem gar nicht definiert (mod 0 ist undefiniert).

(d) Eine echte Zufallsfunktion verhindert das Wiederfinden eines einmal abgelegten Elementes.

(e) Eine Hashfunktion zum Zweck einer Hashtabelle sollte möglichst in $\mathcal{O}(1)$ berechenbar sein.

Aufgabe 3: Hashing mit Chaining *(8 Punkte)*

Implementieren Sie eine Hashtabelle zum Abspeichern positiver, ganzzahliger Schlüssel. Sie dürfen annehmen dass keine Duplikate abgespeichert werden. Benutzen Sie zur Kollisionsbehandlung *Hashing mit Chaining* und nutzen Sie dazu die `DoublyLinkedList` von Aufgabenblatt 3. Sie dürfen die Funktion $k \mapsto (k \bmod m)$ als (triviale) Hashfunktion benutzen, oder anspruchsvollere aus der Vorlesung.

Hinweise: Im public*-Ordner finden Sie eine vorgefertigte Strukturdatei* `HashTable.py`*, die spezifiziert welche Operationen Ihre Hashtabelle anbieten soll. Sie* können *die Strukturdatei herunterladen und anpassen, bzw. mit Programmcode füllen. Falls Sie* `DoublyLinkedList` *aus Aufgabenblatt 3 nicht (korrekt) implementiert haben, können Sie die Musterlösung aus dem* public*-Ordner im SVN verwenden.*