Karlsruher Institut für Technologie — Prof. Dr. Jörn Müller-Quade
Institut für Theoretische Informatik — Björn Kaidel, Sebastian Schlag, Sascha Witt

4. Tutorenblatt zu Algorithmen I im SoSe 2017

http://crypto.iti.kit.edu/index.php?id=799
{bjoern.kaidel,sascha.witt}@kit.edu

Im Folgenden findet ihr einige unverbindliche Vorschläge zur Gestaltung des vierten Tutoriums.

1 Inhalte

Binäre Suche

Falls ihr das noch nicht besprochen haben solltet: Auf dem zweiten Übungsblatt wurde in Aufgabe 3 die binäre Suche behandelt, ohne dies explizit zu sagen. Weißt eure Tutanden darauf hin und macht klar, dass dies ein wichtiger Algorithmus ist. Sollte die Aufgabe bei euch im Tutorium schlecht ausgefallen sein, ist es sicherlich auch sinnvoll, dass nochmal genauer zu besprechen.

Aufgabenvorschläge

In der Vorlesung, in der Übung und auf dem Übungsblatt geht es um Hashing und Anwendungen. Grundlegende Begriffe der Wahrscheinlichkeitstheorie wurden vorgestellt. Diese können nochmal vertieft werden. (Siehe dazu Vorlesung und etwaige Literatur.)

Aufgabe 1 (*Hashing*)

Erklärt nochmal kurz Hashing mit einfach verketteten Listen. Gebt ein einfaches Beispiel für eine Hashtabelle + Hashfunktion und lasst die Studenten Werte gemäß der Hashfunktion einsortieren. Gebt Beispiele für gute und schlechte Hashfunktionen für gegebene Werte und bespreche Best-Case, Worst-Case und erwartete Laufzeiten von verschiedenen Operationen (insert, find, remove...).
Beispiel: $\{36, 78, 50, 1, 92, 15, 43, 99, 64\}$ mit den Hashfunktionen $a \mapsto a \mod 5$, $a \mapsto a \mod 7$ und $a \mapsto a \mod 9$

Aufgabe 2 (*Hashing mit verketteten Liste, erwartete Laufzeit*)

Gebt Anwendungsbeispiele, bei denen Hashtabellen deutlich bessere erwartete Laufzeiten liefern als andere Datenstrukturen (in der Übung werden das Erkennen von Duplikaten und Bloom Filter besprochen).

Aufgabe 3 (*Hashing mit verketteten Listen, Worst-Case vs. erwartete Laufzeit*)

Nehme an, dass für eine Anwendung jedes Element einen Schlüssel aus einem Universum U zugewiesen bekommt. Es sei n die Menge der Elemente, die in eine Hashtabelle eingefügt werden und m die Größe der Hashtabelle (= Anzahl der Slots) mit einer beliebigen Hashfunktion. Zeige, dass für $|U| > nm$ eine Teilmenge von U der Größe n existiert, sodass alle Schlüssel zu dem gleichen Slot gehasht werden. D.h. insbesondere ist die Worst-Case-Zeit für eine Suchoperation in $\Theta(n)$. Die Laufzeit für eine Suchoperation ist für eine zufällige Hashfunktion und für $m = n$ trotzdem erwartet $\mathcal{O}(1)$ (verwende Satz aus der Vorlesung). (Intuition warum das kein Problem ist: Das Universum ist viel größer, als die Anzahl Elemente, die wir hinzufügen wollen)

Musterlösung:
Angenommen es existiert kein Slot mit $\geq n$ möglichen Einträgen. Dann hat jeder Slot maximal $n-1$ Einträge. Somit hat die Hashtabelle (da sie m Slots hat) aber insgesamt höchstens $(n-1)m$ Einträge. Da $|U| > nm$ ist, gäbe es in diesem Fall also Elemente von U, die keinem Slot zugewiesen werden, was ein Widerspruch ist, da Hashfunktionen determinitisch sind und jedem Key aus U einen Slot zuweisen.

Aufgabe 4 (*Hashing mit verketteten Listen*)

Nach dem dritten Glas Bier behauptet ein Kommilitone, man könne Hashing mit verketteten Listen entscheidend verbessern, indem man die verketteten Listen stets sortiert halte.

a) Ist diese Behauptung richtig? Diskutiert hierzu, wie sich das worst-case Laufzeitverhalten von *insert*, *remove* und *find* in diesem Fall ändert.

b) Verwendet statt sortierter verketteter Listen nun sortierte unbeschränkte Arrays. Welche worst-case Laufzeiten können *insert*, *remove* und *find* nun erreichen?

c) Betrachtet nochmals die Hashtabelle aus Teilaufgabe b). Sind die *amortisierten* Laufzeiten von *insert*, *remove* und *find* besser als die worst-case Laufzeiten?

Musterlösung:
Sei n die Anzahl der Elemente, die in der Hashtabelle gespeichert sind.

a) Die worst-case Laufzeit von *insert* verschlechtert sich von $\Theta(1)$ nach $\Theta(\text{Listenlänge})$. Die worst-case Laufzeiten von *remove* und *find* bleiben unverändert in $\Theta(\text{Listenlänge})$.

b) Die worst-case Laufzeit von *find* verbessert sich zu $\Theta(\log \text{Listenlänge})$, da man auf Arrays binäre Suche anwenden kann. Die worst-case Laufzeit von *insert* und *remove* mit unbeschränkten Arrays beträgt $\Theta(\text{Listenlänge})$: Zwar kann man die richtige Stelle im Array zwar in $\Theta(\log \text{Listenlänge})$ finden, aber sowohl beim Einfügen als auch Entfernen müssen bis zu $\Theta(\text{Listenlänge})$ Elemente um eine Position nach hinten verschoben werden.

c) Man betrachte eine Operationsfolge

$$S_1 := \mathit{insert}(e_1), \mathit{insert}(e_2), \ldots, \mathit{insert}(e_k)$$

wobei $\mathit{key}(e_1) = \mathit{key}(e_2) = \cdots = \mathit{key}(e_k)$ und $e_1 > e_2 > \cdots > e_k$. Nach Ausführung von S_1 sind insgesamt

$$\sum_{i=0}^{k-1} i = \Theta(k^2)$$

Verschiebungen von Elementen durchgeführt worden. Weiter benötigt eine Operationsfolge

$$S_2 := \underbrace{\mathit{find}(e), \mathit{find}(e), \ldots, \mathit{find}(e)}_{\ell-\text{mal}} .$$

eine Laufzeit in $\Theta(\ell \log \text{Listenlänge})$, wobei ℓ unabhängig von der Vorgeschichte beliebig groß gewählt werden kann. Weder *insert* noch *find* können also im Vorraus amortisiert werden. Im Fall von *remove* ist das allerdings möglich, da die Anzahl der *remove* Operationen in einer Operationsfolge durch die Anzahl der vorrausgehenden *insert* Operationen begrenzt ist. Man zahlt bei jedem *insert* dazu $\Theta(\text{Listenlänge})$ Tokens auf das Konto ein wobei die Konstante ausreichend groß gewählt werden muss. So kommt man für *remove* auf konstante amortisierte Zeit.

Hinweis: In der Übung wurden gerade *Hotlists* sowie *Skip Lists* vorgestellt. Es ist durchaus möglich, dass eure Studis versuchen, diese hier als zugrundeliegende Datenstrukturen zu verwenden. Schaut euch vielleicht beides nochmal grob an, um auf Fragen vorbereitet zu sein.

Aufgabe 5 (*Kreativaufgabe*)

a) Entwerfe eine Realisierung eines *SparseArray* (auf deutsch soviel wie „spärlich besetztes Array“). Dabei handelt es sich um eine Datenstruktur mit den Eigenschaften eines beschränkten Arrays, die zusätzlich schnelle Erzeugung und schnellen Reset ermöglicht. Nehmen Sie dabei an, dass **allocate** beliebig viel *uninitialisierten* Speicher in konstanter Zeit liefert. Im Detail habe das *SparseArray* folgende Eigenschaften:

- Ein *SparseArray* mit n Slots braucht $\mathcal{O}(n)$ Speicher.
- Erzeugen eines leeren *SparseArray* mit n Slots braucht $\mathcal{O}(1)$ Zeit.
- Das *SparseArray* unterstützt eine Operation *reset*, die es in $\mathcal{O}(1)$ Zeit in leeren Zustand versetzt.
- Das *SparseArray* unterstützt die Operation $get(i)$ und $set(i, x)$. Dabei liefert $A.get(i)$ den Wert, der sich im i-ten Slot des *SparseArray* A befindet; $A.set(i, x)$ setzt das Element im i-ten Slot auf den Wert x. Wurde der i-te Slot seit der Erzeugung bzw. dem letzten *reset* noch nicht mit auf einen bestimmten Wert gesetzt, so liefert $A.get(i)$ einen speziellen Wert $\perp$. Die Operationen *get* und *set* dürfen zudem beide nicht mehr als $\mathcal{O}(1)$ Zeit verbrauchen (man nennt so etwas *wahlfreien Zugriff*, engl. *random access*).

b) Nimm an, dass die Datenelemente, die Sie im *SparseArray* ablegen wollen, recht groß sind (also z.B. nicht nur einzelne Zahlen, sondern Records mit 10, 20 oder mehr Einträgen). Geht die Realisierung unter dieser Annahme sparsam oder verschwenderisch mit dem Speicherplatz um? Wenn du die Realisierung für verschwenderisch hälst, überlege ob und wie du sie besser machen kannst.

c) Vergleiche dein *SparseArray* mit Bounded-Arrays. Welche Vorteile und Nachteile siehst du im Hinblick auf den Speicherverbrauch und auf das Iterieren über alle Elemente mit *set* eingefügten Elemente?

Musterlösung:

a) Es reicht nicht einfach nur unbeschränkte Arrays zu benutzen, da **allocate** nur *uninitialisierten* Speicher liefert. Die Operation *get* muss aber erkennen, ob ein Slot bereits auf einen Wert gesetzt wurde oder nicht. Man müsste deshalb bei Erzeugung und Reset alle Slots mit dem Wert $\perp$ initialisieren, was $\Theta(n)$ Zeit dauert.

Stattdessen nutzt man *zwei* Arrays: ein Array D, das die Nutzdaten aufnimmt, und ein Hilfsarray H, in das man Indizes abspeichert. Zusätzlich zu den Nutzdaten, nimmt jeder Slot von D außerdem noch eine nicht-negative ganze Zahl auf. Beide Arrays haben n Slots und sind zu Beginn uninitialisiert. Außerdem hält man sich einen Zähler c, der initial auf 0 gesetzt wird.

Die Operation $set(i, x)$ arbeitet nun so: Man setzt $D[i].daten := x$, $D[i].zahl := c$ und $H[c] := i$. Danach wird c um eins erhöht. Die Operation $get(i)$ kann nun einfach nachprüfen, ob der i-te Slot schon gesetzt wurde oder nicht. Dazu schaut man nach, ob $D[i].zahl < c$ und $H[D[i].zahl] = i$ gilt. Gilt dies nämlich nicht, liefert $get(i)$ den Wert $\perp$ zurück.

Für die Operation *reset* setzt man einfach $c := 0$. Mehr ist nicht zu tun.

b) Die Lösung aus a) ist insofern verschwenderisch, dass n Slots für Nutzdaten allokiert werden, die zwar uninitialisiert aber *reserviert* sind. Stattdessen sollte man die Nutzdaten mit ins Array H speichern und für H ein unbounded Array verwenden, das initial nur einen Slot hat.

Sei m die Anzahl der tatsächlich gesetzten Elemente. Dann belegt D den Platz für n nicht-negative ganze Zahlen. Das unbeschränkter Array H belegt maximal den Platz für $3m$ Nutzdatenelemente und Indizes (während des Kopiervorganges, wenn das unbeschränkte Array kopiert reallokiert wird).

c) Iterieren dauert nur $\mathcal{O}(m)$ statt $\mathcal{O}(n)$ Zeit. Der Speicherverbrauch ist schlechter, wenn die Nutzdatenelemente klein sind. Für große Nutzdatenelemente kann der Speicherverbrauch sogar *erheblich*

besser sein, sofern man eine sparsame Realisierung wie in b) verwendet und nicht zuviele Elemente eingefügt werden.