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

3. Tutorenblatt zu Algorithmen I im SoSe 2017

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

1. Inhalte

Auf dem Übungsblatt werden Datenstrukturen (also vor allem doppelt und einfach verkettete Listen und Unbounded Arrays) eine Rolle spielen. Bitte wiederholt die Ideen dahinter noch einmal kurz. Bitte konstruiert z.B. einmal eine unbounded queue mittels einer doppelt verketteten Liste. Tut dies bitte sowohl unter Verwendung der "splice"-Methode aus der Vorlesung, als auch durch "manuelles" Umhängen der Zeiger.

Armortisierte Analyse

Erfahrungsgemäß wird die armotisierte Analyse nicht so gut verstanden. Wiederholt das vielleicht kurz und macht ein Beispiel dazu.

Hinweis für das vierte Tutorium - Binäre Suche

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. Das ist natürlich erst dann sinnvoll, wenn das zweite Übungsblatt abgeben und von euch korrigiert wurde :) Nur schonmal als Hinweis!

2. Arbeitsvorschläge

In der Vorlesung wurden vorwiegend Datenstrukturen (doppelt verkettete Listen, ...) behandelt. Außerdem wurde im Zusammenhang mit Unbounded Arrays das erste Mal die amortisierte Analyse behandelt. Diese Analyse sollt ihr an Hand folgender Aufgabe vertiefen:

Aufgabe 1 (*Kreativaufgabe*)

Entwickelt eine Datenstruktur, die folgendes kann:

- *pushBack* und *popBack* in $\mathcal{O}(1)$ im Worst-Case (nicht nur amortisiert!).
- Zugriff auf das x-te Element in $\mathcal{O}(\log n)$ im Worst-Case (nicht nur amortisiert!).
  - Achtung: Mit „x-tes Element" ist nicht das Element mit dem Wert x gemeint, sondern das Element an Stelle x innerhalb der Datenstruktur.

Hierbei soll eine Speicherallokation beliebiger Größe nur $\mathcal{O}(1)$ kosten.
Zusatzaufgabe:
Es geht auch umgekehrt:

- *pushBack* und *popBack* in $\mathcal{O}(\log n)$ im Worst-Case (nicht nur amortisiert!).
- Zugriff auf das x-te Element in $\mathcal{O}(1)$ im Worst-Case (nicht nur amortisiert!).

Musterlösung: Eine doppelt verkettete Liste von Arrays, so dass jeder Array doppelt so groß wie sein Vorgänger ist (das Array an Listenposition i hat also Größe 2^i). Zusätzlich einen Zeiger back auf das letzte Array bzw. dort auf das kleinste noch freie Element.
Zusatzaufgabe: Verwende statt der doppelt verketten Liste ein Array von Zeigern.
Am Ende findet Ihr noch weitere handschriftliche Notizen zu der Aufgabe/Lösung.

Aufgabe 2 (*Listen*)

Untersucht, was mit einer doppelt-verketteten Liste passiert, wenn die in der Vorlesung vorgestellte Funktion $splice(a, b, t : Handle)$ fälschlicherweise mit $t \in \langle a, \ldots, b \rangle$ aufgerufen wird, wobei weiter a vor b in der Liste vorkommt.

Aufgabe 3 (*Arrays, Hashing*)

Gegeben sei ein Array $A = A[1], \ldots, A[n]$ mit n Zahlen in beliebiger Reihenfolge.
Für eine gegebenen Zahl x soll ein Paar $(A[i], A[j])$, $1 \leq i, j \leq n$ gefunden werden, für das gilt: $A[i] + A[j] = x$.

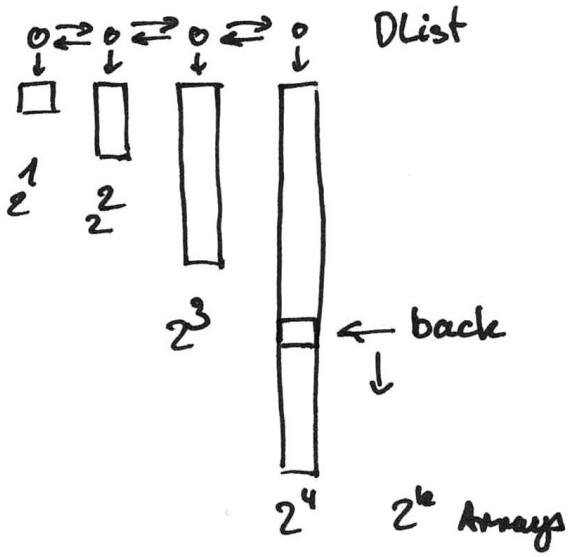
1. Gebt eine Lösung für $x = 33$ und $A = (7, 15, 21, 14, 18, 3, 9)$ an.
2. Gebt einen effizienten Algorithmus an, der das Problem in erwarteter Zeit $\mathcal{O}(n)$ löst, und bei Erfolg ein Paar $(A[i], A[j])$ ausgibt, ansonsten NIL.

Lösung mit Hashing: Im ersten Schritt alle Elemente in die Hash-Tabelle einfügen, im zweiten Schritt zu jeder Zahl das „Gegenstück“ suchen (das „Gegenstück“ zu $A[i]$ ist $x - A[i]$). **Wichtig:** Die Hashtabelle muss $\Omega(n)$ Slots haben, sonst greift Satz 1 aus der Vorlesung nicht. Mit diesem können wir zeigen, dass die erwartete Laufzeit von Einfügen und Tabellenlookup in $\mathcal{O}(1)$ ist.
Lösung ohne Hashing: Wichtig: Diese Lösung funktioniert nur, wenn alle Zahlen natürliche Zahlen sind, und die größte Zahl in A von $c_1 \cdot n$ begrenzt wird! Im ersten Schritt wird ein Bit-Array der Länge $c_1 \cdot n$ angelegt und auf 0 initialisiert ($\mathcal{O}(n)$). Im zweiten Schritt wird über die Liste iteriert und für jede Zahl in der Liste an der entsprechenden Stelle ein Bit auf 1 gesetzt ($\mathcal{O}(n)$). Danach funktioniert es wie obige Lösung. *Hinweis:* Man könnte jetzt natürlich behaupten, dass auch dies eine “Lösung mit Hashing” ist, nur dass der Hash-Algorithmus besonders dumm ist. Damit hätte man wohl Recht. Auf diese Lösung sollte man aber auch kommen können, ohne Hashing gesehen zu haben.
Hinweis für Montagstutorien: Vermutlich kommt Hashing erst in der Vorlesung am Montag, 15.05, dran. Falls euer Tutorium vorher stattfindet, könnt ihr die Lösung ohne Hashing vorstellen und das ganze Thema Hashing etwas motivieren.

3. Wahrscheinlichkeitsrechnung

Bitte wiederholt einige Grundlagen der Wahrscheinlichkeitsrechnung. Erfahrungen aus dem letzten Semester haben leider gezeigt, dass die Studierenden scheinbar teilweise keinerlei Wahrscheinlichkeitstheorie mehr in der Schule lernen. Schaut mal, wie das bei euren Tutanden ist. Wiederholt/besprecht solche Dinge wie Ereignisse, Wahrscheinlichkeiten, Gleichverteilung, Zufallsvariablen,

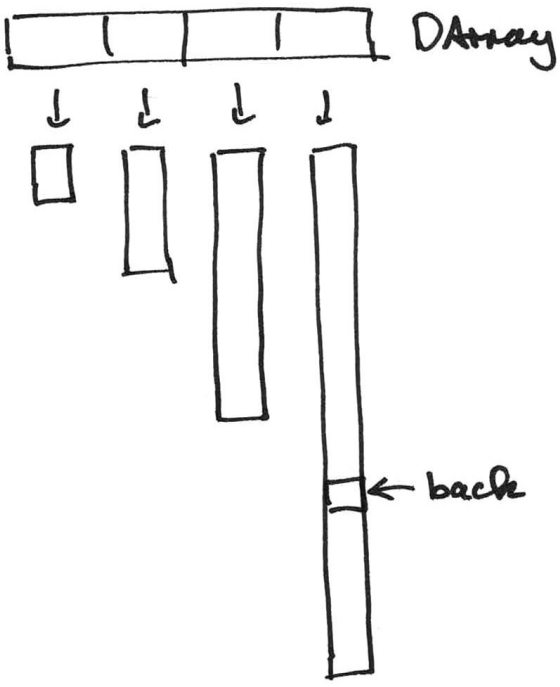
0-1-Zufallsvariablen, Erwartungswert, Linearität des Erwartungswerts etc. Natürlich könnt ihr keine Wahrscheinlichkeitstheorievorlesung ersetzen, aber die Tutanden sollten zumindest mit einem Grundverständnis dieser Begriffe aus eurem Tutorium gehen.



push/pop: O(1) insert bei back
oder O(1) allocate next 2^{k+1} array and append to DList in O(1).

Get(x): calc $\lceil \log_2 x \rceil$
-> iterate DList to find right array (length DList is $O(\log u)$)
+ direct access into array.

Zusatz:



Get(x) in O(1) klar:
calc $\lceil \log_2 x \rceil$ + 2x
direct access.

Warum hat push/pop $O(\log u)$?
-> weil im worst-case das DArray kopiert wird.