Datenstrukturen & Algorithmen 26. März 2014

Handout 5

Sebastian Millius, Sandro Feuz, Daniel Graf

Thema: Selbstanordnung in Linearen Listen, Amortisierte Analyse, Hashing

Referenz: Widmayer, Kapitel 3.3, 4.1-4.3, Cormen, Kapitel 11

Links

- On the List Update Problem
  http://www.inf.ethz.ch/personal/emo/DoctThesisFiles/ambuehl02.pdf
  (http://goo.gl/J4Tpb)
- Hashing: MIT Open Courseware, Lectures 5-7 (http://goo.gl/hbGXc)
- Amotisierte Analyse: Jeff Erickson Lecture Notes
  http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/notes/15-amortize.pdf
  (http://goo.gl/pPRtD0)
- Resizable Arrays in Optimal Time and Space
  http://www.cs.uwaterloo.ca/research/tr/1999/09/CS-99-09.pdf
  (http://goo.gl/201gW)

Selbstandordnung in Linearen Listen

Move to front Bei der *Move to front*-Strategie wird jenes Element, auf das zuletzt zugegriffen wurde, vorne in der Liste eingefügt. Die restlichen Elemente werden dann um eins nach hinten geschoben. Dadurch werden sich häufige Elemente im Durschnitt eher im vorderen Bereich wiederfinden und haben daher eine bessere Zugriffszeit als seltene Elemente, die tendenziell beim Aufruf eher weiter hinten in der List zu finden sind. Man kann sich hier ein Worst-Case Szenario leicht vorstellen, indem man in einer ursprünglichen Anordnung von $A, B, C, ..., Z$ die Zugriffe in der Reihenfolge $Z, Y, ..., B, A, Z, ...$ hat. Asymptotisch ist dies aber nicht schlechter als ohne jegliche Vertauschungen. Beides ist in $\mathcal{O}(n^2)$ in der Anzahl der Zugriffe.

Transpose Das aktuelle Element wird mit seinem Vorgänger vertauscht. Auch dadurch sollten häufigere Elemente tendenziell im vorderen Teil der Liste anzutreffen sein. Ein Worst-Case Szenario ist hier für eine initiale Anordnung von $A, B, ..., Y, Z$ die Zugriffe $Z, Y, Z, Y, ...$. Die Überlegung, das aktuelle Element 2 Stellen vorzurücken bringt keine Verbesserung (Zugriffe: $Z, Y, X, Z, Y, X, ...$). Das selbe gilt für eine beliebige Anzahl k Stellen.

Frequency count Diese Methode führt eine Zugriffsstatistik und sortiert die Liste nach jedem Zugriff neu (nach absteigender Zugriffshäufigkeit). Ein Problem ist hier der zusätzlich benötigte Speicherplatz für die Häufigkeitszähler.

Die *Move to front*-Strategie ist asymptotisch (amortisierte Analyse) optimal.

Genauer gesagt gilt folgende Aussage: Für jede beliebige Strategie A zur Selbstanordnung und jede Folge s von m Zugriffsoperationen gilt:

$C_{MF}(s) \leqslant 2 \cdot C_A(s)$,

wobei $C_X(s)$ die Gesamtkosten der Zugriffe zur Durchführung aller m Operationen von s gemäss Strategie X ist. Experimentelle Resultate zeigen auch, dass MF im Grossen und Ganzen Vorteile gegenüber der beiden anderen Strategien hat.

Amortisierte Analyse

http://en.wikipedia.org/wiki/Amortized_analysis

Eine *amortisierte Analyse* ist eine Strategie um eine Sequenz von Operationen zu analysieren und zu zeigen, dass die durchschnittlichen Kosten pro Operation klein sind, obwohl eine einzelne Operation in der Sequenz teuer sein kann. Amortisierte Analyse garantiert die Durchschnittsperformance jeder Operation im *Worst Case*.

Die Idee der amortisierten Analyse ist, dass nicht mehr jede einzelne Operation eine gewisse Zeitschranke erfüllen muss, sondern dass eine ganze Serie von Operationen in bestimmter Zeit abgehandelt wird. *Amortisiert konstant* heisst: n Operationen brauchen $\mathcal{O}(n)$ Zeit. Es ist durchaus möglich, dass eine Serie von Operationen (bsp. Einfügen in Arrays von dynamischer Grösse) amortisiert konstant ist, dabei aber einzelne Operationen lineare Zeit brauchen können.

Aggregat-Methode

Gesamtkosten aller Operationen ermitteln und durch Anzahl Operationen dividieren.

Es sei $T(n)$ die Worst-Case Laufzeit für eine Sequenz von n Operationen. Die amortisierte Laufzeit jeder Operation ist $\frac{T(n)}{n}$.

Bsp: http://de.wikipedia.org/wiki/Aggregat-Methode

Bankkonto-Paradigma

Eine Möglichkeit zur amortisierten Analyse ist ein virtuelles Bankkonto zu führen. Dabei bezahlt man bei jeder Operation einen gewissen Betrag auf das Konto ein und bezahlt zugleich die Kosten für den Aufwand der jeweiligen Operation von dem Konto. Günstige Operationen zahlen einen bestimmten Betrag auf ein Konto ein. Es wird also für diese vorsorglich mehr Kosten berechnet. Die teuren Operationen können dann dafür wiederum "gratis" vom Konto abheben. Solange das Konto nie unter Null gehen kann, sind wir sicher, dass alle Operationen zusammen insgesamt nur so lange dauern, wie Geld auf das Konto eingezahlt wurde.

Bsp: http://en.wikipedia.org/wiki/Accounting_method

Potenzialfunktionmethode

Potentialfunktion um einem inneren Zustand der Datenstruktur ein Potential zuzuweisen. Potential des Initialzustandes darf nie unterschritten werden. Ähnlich wie das Bankkonto-Paradigma ausser dass man den Zustand der Datenstruktur für den Kostenausgleich benutzt.

Dabei definiert man eine Funktion Φ, welche jedem möglichen Zustand der zu betrachtenden Struktur einen Wert (das Potential) zuweist. Die amortisierten Kosten der $i-$ten Operation a_i sind dann die realen Kosten der $i-$ten Operation plus die Potential-Änderung: $a_i := t_i + \Phi_i - \Phi_{i-1}$.

Zusätzlich soll nun gelten, dass das Anfangspotential kleiner ist als das Endpotential: $\Phi_0 \leqslant \Phi_m$. Dann folgt nämlich, dass die amortisierten Kosten eine obere Schranke für die realen Kosten sind: $\sum_{i=1}^{m} t_i \leqslant \Phi_m - \Phi_0 + \sum_{i=1}^{m} (t_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{m} a_i$.

Insbesondere reicht es nun zu zeigen, dass die amortisierten Kosten a_i in jedem Fall konstant sind um zu folgern, dass die ganze Folge in amortisiert konstanter Zeit läuft.

Bsp: http://en.wikipedia.org/wiki/Potential_method

Eine breite und ausführliche Darstellung der Methoden und Amortisierter Analyse im Allgemeinen findet sich in Cormen, *Introduction to Algorithms*

Hashing

Hash Tabellen sind eine Datenstruktur um ein Wörterbuch zu realisieren. Beim Hash-Verfahren wird versucht, durch Berechnung festzustellen, wo der Datensatz mit Schlüssel k gespeichert ist. Die Idee ist, dass das Nachschauen eines Elementes in einem Array $\Theta(1)$ ist, wenn man seinen Index weiss. Bei einer Suchanfrage wird der Schlüssel mitgegeben und falls das Schlüssel/Wert Paar existiert, wird der Wert zurückgegeben. Man kann damit aber beispielsweise auch einfache Mengen modellieren. Hier kann man Elemente einfügen und testen ob ein bestimmtes Element vorhanden ist.

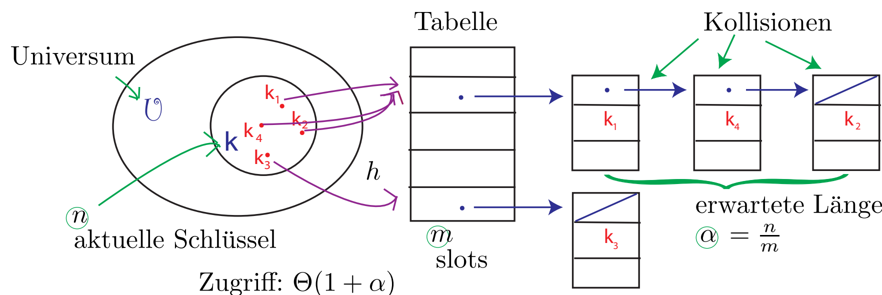
Dabei wird versucht Schlüssel aus einem grossen Schlüsselbereich/Universum $\mathcal{U}$ in einer Hashtabelle der Grösse m abzuspeichern. Eine Hash-Table ist grundsätzlich mal ein Array. Das Array sollte genügend gross sein, so dass nur bis zu ca. 75% der Plätze belegt sind. Je nach Implementation (z.B. in Java) wird die Arraygrösse auch dynamisch angepasst, was aber ziemlich aufwändig ist, da normalerweise alle Elemente neu eingefügt werden müssen.

Eine *Hash Funktion* ist eine mathematische Funktion die Schlüssel auf einen Index (*Hashadresse*) abbilden. Die Hashfunktion sollte schnell (sicher $\mathcal{O}(1)$) zu berechnen sein, und gleichzeitig die Schlüsselstruktur aufbrechen.

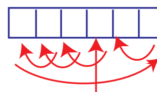
Im optimalen Fall kann somit nach einer Auswertung dieser Funktion sofort ein Element gespeichert oder darauf zugegriffen werden. Das Problem ist aber, dass die Hashfunktion mehrere Schlüssel auf die gleiche Stelle im Array abbilden kann, da es üblicherweise viel mehr mögliche Schlüssel als Plätze im Array hat. ($\rightsquigarrow$ Adresskollision, Synonyme)

Falls also beim Einfügen der Platz schon belegt ist oder sich beim Suchen ein anderer Schlüssel an diesem Ort befindet, muss ein Verfahren existieren das besagt, wo nun weiterzusuchen ist. Dabei gibt es verschiedene Möglichkeiten welche natürlich sehr unterschiedliche Eigenschaften haben:

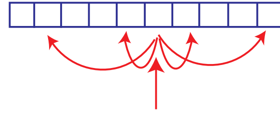
- *Verkettung*: In jedem Feld des Arrays befindet sich wiederum eine Datenstruktur die mehrere Elemente aufnehmen kann (z.B. Liste, Baum, ...)



- *Mehrere Hashfunktionen*: Bei einer Kollision wird einfach die nächste Hashfunktion probiert.
- *Offene Hashverfahren.* Anders als beim Verketten werden die Überläufer direkt in der Tabelle abgelegt. Eine Sondierungsreihenfolge gibt dabei an, wo Synonyme gespeichert werden.
  - *Lineares Sondieren*: Bei Kollisionen einfach die nächsten Felder probieren (absteigend oder aufsteigend).



    Ein Problem bei diesem Verfahren kann die *primäre Häufung* sein: Teile von bereits besetzten Bereichen werden immer grösser, so dass immer länger sondiert werden muss.
  - *Quadratisches Sondieren*: Anstatt die Felder linear aufzufüllen / zu durchsuchen wird mit quadratisch steigenden Abständen gesucht.



- *Doppeltes Hashing*: Eine zweite Hashfunktion $h_0(x)$ wird benutzt um den Abstand beim Sondieren zu bestimmen. Die zusammengesetzte Funktion lautet dann: $h_i(x) = (h(x) - h_0(x) \cdot i) \mod m$

Es gibt auch noch ein paar weitere Methoden. Die wichtigsten sind aber oben aufgeführt.

Bemerkung: Ob jeweils (zuerst) nach unten oder nach oben sondiert wird ist Geschmackssache. Wichtig ist einfach, dass es immer konsequent gleich gemacht wird. Dies gilt insbesondere auch für die Prüfung (hier ev. noch hinschreiben in welche Richtung man sondiert..)

Aufgabe

Führen Sie die unten gegebene Sequenz von Einfüge-Operationen auf einer anfangs leeren Hashtabelle der Grösse 11 mit Hashfunktion $h(k) = k \mod 11$ aus. Verwenden Sie als Kollisions-Auflösungs-Strategie einmal lineares Sondieren, einmal quadratisches Sondieren und einmal Double Hashing mit $h_0(k) = 1 + (k \mod 9)$ aus. Zählen Sie die Anzahl betrachteter Einträge der Operationsfolge. Welche Variante ist für diese Sequenz die beste? Die Sequenz, die eingefügt werden soll, ist: $20, 34, 26, 33, 11, 16, 47, 4, 14, 17$.