Datenstrukturen & Algorithmen 25. März 2015

Handout 5

Sebastian Millius, Sandro Feuz, Daniel Graf

Thema: Selbstanordnung in Linearen Listen, Amortisierte Analyse, Splay Bäume

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)
- 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)
- Splay Trees
  - Tarjan, Sleator: Self-adjusting binary search trees
    http://portal.acm.org/citation.cfm?id=3835 (http://goo.gl/pe1hu)
  - Demo Applet http://www.link.cs.cmu.edu/splay/
    (http://goo.gl/AnzMK)
  - Slides Belfrage: http://www.belfrage.net/eth/d&a/pdf/uebung12_h.pdf
    (http://goo.gl/a5o41)
  - Berkley Lecture: http://goo.gl/UQrSh

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*

Splay Bäume

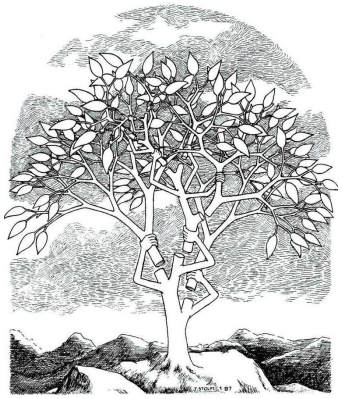
So you are lean and mean and resourceful and you continue to walk on the edge of the precipice because over the years you have become fascinated by how close you can walk without losing your balance. – Richard M. Nixon

Motivation: Eine Folge von Zugriffsoperationen auf einer Menge von Elementen eines total geordneten Universums ausführen. Dabei sollen kürzlich aufgerufene Elemente wiederum schnell zugreifbar sein.

Splay Trees sind sich selbstanordnende binäre Suchbäume mit guten Balanceeingenschaften (amortisiert über eine Sequenz von Operationen). Sie wurden erstmals von Sleator und Tarjan 1985 eingeführt. In einer gewissen Weise vereinen sie die Eigenschaften von AVL-Bäumen (balancierend), und einer MTF-Liste (kürzlich zugegriffene Elemente sind sehr schnell erreichbar). Die Vorteile gegenüber selbstanordnenden Listen liegen auf der Hand. Suchanfragen können in (amortisiert) $\mathcal{O}(\log n)$ Zeit durchgeführt werden.

Eine Anfrage im Splay Tree zieht immer eine weitere Operation mit sich, das *Splaying* (siehe unten). Dabei wird der Baum so arrangiert, dass das aktuelle Element an die Wurzel platziert wird. Dies wird mit Baumrotationen gemacht, die vom AVL-Baum her bekannt sein sollten. Ein Nachteil ist, dass der Baum komplett unbalanciert sein kann; die amortisierte Analyse zeigt jedoch trotzdem eine Performance von $\mathcal{O}(\log n)$ Zeit für Einfüge-, Such- und Lösch-Operationen.

(Für gleichmässig verteilte Zugriffshäufigkeiten ist ein Splay Tree jedoch sehr unpraktisch und es empfiehlt sich, einen 'normalen' binären Suchbaum zu verwenden.)



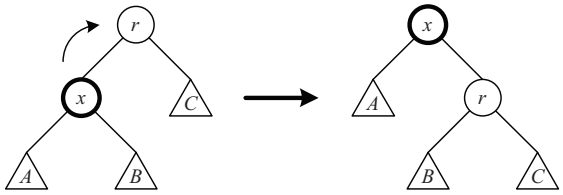
A Self-Adjusting Search Tree (Jorge Stolfi)

Im Folgenden sind die Operationen beschrieben. Diese werden manchmal auch genauer unterschieden. So bedeutet eine 'Zig' Operation eine Rechtsrotation und 'Zag' eine Linksrotation.

Analoges gilt für die beiden anderen Operationen.

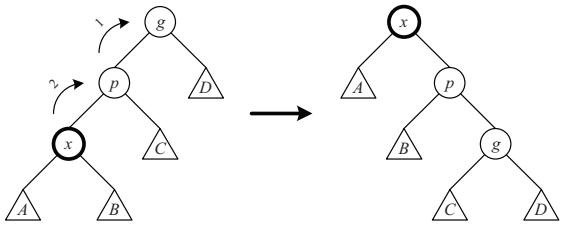
'Zig'

x ist das Element, auf das zugegriffen wird. Es muss nun also an die Wurzel rotiert werden. Die Zig-Operation wird nur ausgeführt, wenn x unmittelbar unter der Wurzel ist und es kann eine einfache Rotation über die Achse von x und der Wurzel vorgenommen werden. Diese Rotationen sind analog zu den einfachen Rotationen in einem AVL-Baum.



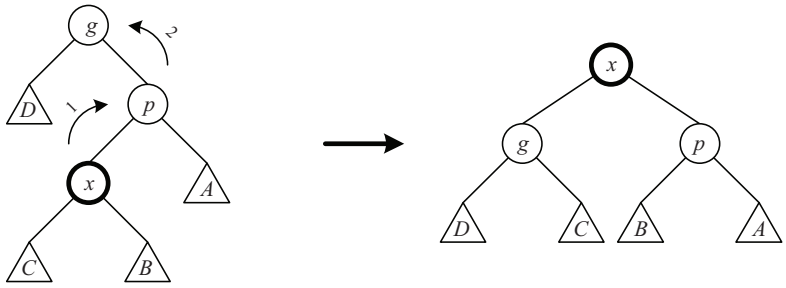
'Zig-Zig'

Diese Operation wird durchgeführt, wenn der Vater p vom aktuellen Element x nicht die Wurzel ist und sowohl p als auch x jeweils linke Kinder ihrer jeweiligen Väter sind. Es werden zwei Zig-Rotationen in die gleiche Richtung durchgeführt, um x zwei Level nach oben zu befördern. Zuerst p mit seinem Vater, dann x mit p. Diese Operation entspricht zwei einfachen Rotationen in die selbe Richtung in einem AVL-Baum.



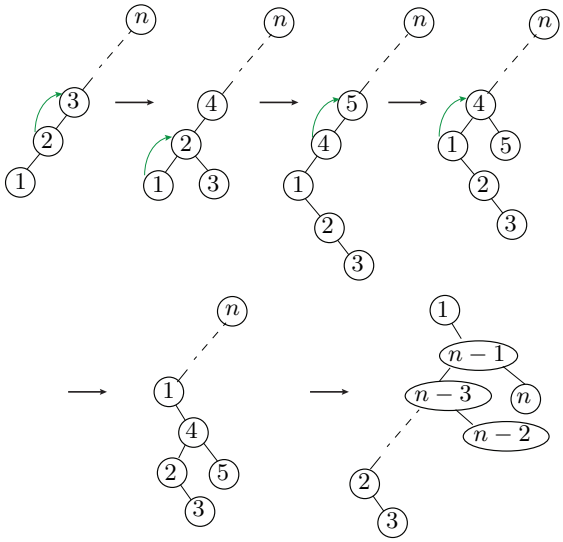
'Zig-Zag'

Diese Operation wird durchgeführt, wenn x ein linkes und p ein rechtes Kind seines Vaters ist. Zuerst wird eine Rotation über die Achse von x und p vorgenommen, danach eine über die Achse von x und seinem neuen Vater. Dies entspricht einer Doppelrotation in einem AVL-Baum (d.h. zwei einfache Rotationen in verschiedene Richtungen)



'Zag', 'Zag-Zag', 'Zag-Zig'

Analog zu den obengenannten, einfach spiegelverkehrt.



Nach dem Splaying des Schlüssels 1 *ist der Baum nur noch halb so gross*

Stop and Think

Wieso werden beim Zig-Zig zwei Rotationen durchgeführt? Was würde geschehen, wenn wie beim AVL nur eine Rotation gemacht würde? Was geschieht mit obigem Beispiel?

Einfügen

Um einen Schlüssel k in den Splay Tree einzufügen, wird zunächst $splay(k)$ ausgeführt und damit der symmetrische Vorgänger (oder Nachfolger) an die Wurzel des Baumes gebracht (ist das Element bereits im Baum, so ist dies nun an der Wurzel und es ist nichts weiter zu tun). Nun kann k einfach an die Wurzel gesetzt werden: die aktuelle Wurzel wird ein Kind von k.

Entfernen

Um einen Schlüssel k aus dem Splay Tree zu entfernen, wird zunächst wieder $splay(k)$ ausgeführt und damit das Element an die Wurzel gebracht. Ist k nicht im Baum ist nichts weiter zu tun. Ansonsten wird im linken Teilbaum der Wurzel k der symmetrische Vorgänger gesplayed und an die Wurzel des linken Teilbaumes von k gebracht. Dieser Vorgänger hat kein rechtes Kind und k kann deshalb einfach von der Wurzel entfernt werden.