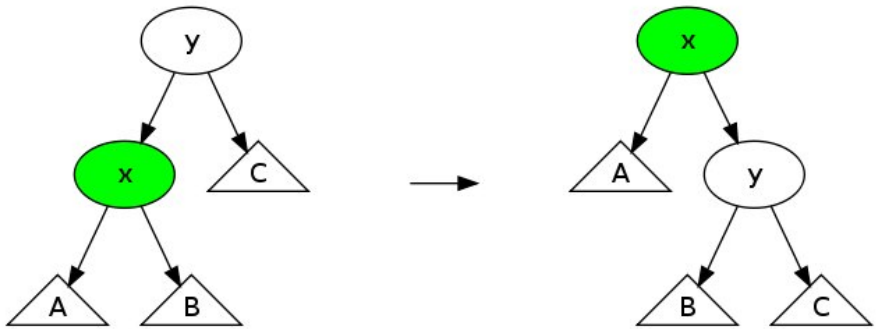
Freie Universität Berlin



Splay-Bäume

Joseph Schröer

Seminar über Algorithmen
SoSe 2011, Prof. Dr. Helmut Alt

Splay-Baum (engl. *Splay Tree*)

- Selbstanpassender binärer Suchbaum
- Engl. *(to) splay* – spreizen, wegstrecken, weiten
  - dt. auch *Spreizbaum*
- Erfinder: Daniel D. Sleator und Robert E. Tarjan (1985)

Robert E. Tarjan
(Turing-Award 1986)

Daniel D. Sleator

Taxonomie der Suchbäume

- Balancierte Suchbäume
- Optimale Suchbäume
- (Finger-Suchbäume, Biased Search Trees, . . .)

- Selbstanpassende Suchbäume
  - Gegenstand des heutigen Referats

Höhenbalancierte Suchbäume

AVL-Baum, Rot-Schwarz-Baum, ...

- Explizit garantierte Balanciertheit durch Rebalancierungsoperationen
  - Links-, Rechts-, Doppelrotationen
- Suchen, Einfügen, Löschen effizient
  - Erwartet und Worst-Case: $\mathcal{O}(\log n)$

\+ Ideal für gleichverteile Suchanfragen

\+ Günstiges Verhalten auch im Worst Case

– Für nicht-gleichverteilte Anfragen nicht optimal

– Zusätzlicher Speicherbedarf für Balance-Informationen

Optimale Suchbäume

Statischer Suchbaum, angepasst an die Verteilung der Anfragen

- „Schiefer“ Baum; häufig angefragte Elemente nahe der Wurzel

+ Minimale durchschnittliche Zugriffszeit

– Zugriffsverteilung muss fest und bekannt sein
– Nur vorteilhaft, wenn weit von Gleichverteilung entfernt
– Kein Einfügen/Löschen möglich
– Einzelne Suche schlimmstenfalls in linearer Zeit

Selbstanpassende Suchbäume

- Ziel: Amortisierte Effizienz
- Keine expliziten Balanciertheitsgarantien
  - Höhe ($\sim$ Suchzeit) schlimmstenfalls $\mathcal{O}(n)$
- Stattdessen *Restrukturierungsheuristiken*
  - Ziel: häufige Anfragen schneller beantworten
    - *Single Rotation* (Rotiere gesuchten Knoten mit seinem Vater)
    - *Move-to-Root* (Rotiere Knoten solange mit Vater, bis er Wurzel ist)
    - *Splaying*

\+ Amortisiert so effizient wie explizit balancierte Datenstrukturen

\+ Wesentlich besser, falls Anfragen nicht gleichverteilt

\+ Kein Speicherbedarf für Meta-Informationen

– Häufige Restrukturierung (auch nach Such-Operation)

– Einzelne Operation schlimmstenfalls in linearer Zeit

Der Splay-Baum

Binärer Suchbaum mit folgenden Operationen:

- ACCESS
- INSERT
- DELETE

Hilfsfunktionen:

- SPLAY
- JOIN
- SPLIT

Die SPLAY-Operation

Bessere Restrukturierungsheuristik als *Move-to-Root*

- Knoten x wird zur Wurzel gemacht
- Tiefe der Knoten auf dem Suchpfad zu x wird ungefähr halbiert

Die SPLAY-Operation (2)

SPLAY(x)

while x nicht Wurzel
if $p(x)$ ist Wurzel
// „zick“-Rotation
rotiere x mit $p(x)$
elseif x und $p(x)$ sind linke *[rechte]* Kinder
// „zick-zick“-Rotation
rotiere $p(x)$ mit $p(p(x))$
rotiere x mit $p(x)$
elseif x linkes *[rechtes]* Kind, $p(x)$ rechtes *[linkes]* Kind
// „zick-zack“-Rotation
rotiere x mit $p(x)$
rotiere x mit (neuem) $p(x)$

Die ACCESS-Operation

Prüfe, ob i in T enthalten ist.

Ja: Gib Zeiger auf den Knoten zurück,

Nein: Gib NULL-Zeiger zurück.

```
ACCESS(i, T)
    x := Wurzel von T
    while x ≠ i
        if i < x
            Neu := linkes Kind von x
        else
            Neu := rechtes Kind von x
        if Neu = NULL
            SPLAY(x, T)
            Return NULL
    SPLAY(x, T)
    Return x
```

Die JOIN-Operation

Füge T_1 und T_2 zu einem Baum zusammen.
Voraussetzung: Alle Elemente in T_1 sind kleiner als alle Elemente in T_2.

JOIN(T_1, T_2)

    ACCESSMAX(T_1)
    T_1.wurzel.rechts := T_2.wurzel
    Return T_1

ACCESSMAX ist eine ACCESS-Operation auf das maximale Element (folge immer rechten Zeigern).

Die SPLIT-Operation

Konstruiere zwei Bäume T_1 und T_2 sodass T_1 alle Elemente $\leq i$ enthält und T_2 alle anderen.

SPLIT(i, T)

ACCESS(i, T)
$x :=$ Wurzel von T
if $x > i$
    T_2.wurzel $:= x$ // *Trenne das linke Kind von x ab*
    T_2.wurzel.links := NULL
    T_1.wurzel $:= x$.links
else
    T_1.wurzel $:= x$ // *Trenne das rechte Kind von x ab*
    T_1.wurzel.rechts := NULL
    T_2.wurzel $:= x$.rechts
Return (T_1, T_2)

Die INSERT-Operation

Füge i in T ein, vorausgesetzt, es ist noch nicht enthalten.

INSERT(i, T)

$(T_1, T_2) :=$ SPLIT(i, T)
Verpacke i in Knoten x
x.left $:= T_1$
x.right $:= T_2$
$T := T_x$

Die DELETE-Operation

Lösche i aus T, vorausgesetzt, es ist enthalten.

DELETE(i, T)

    ACCESS(i, T)
    $T :=$ JOIN(x.left, x.right)

Komplexität: Übersicht

	Amortisiert	Worst Case
Suche	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Einfügen	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Löschen	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$

Platzkomplexität: $\Theta(n)$

- Betrachte Folge von Operationen anstatt individueller Operation

Definition (Amortisierte Laufzeit)

Durchschnittliche Laufzeit pro Operation in *worst-case*-Folge von Operationen.

$\rightarrow$ Stärkere Aussage als *erwartete Laufzeit*

Amortisierte Laufzeit:

$$a = t + \Phi' - \Phi$$

(t bezeichnet die *tatsächliche Laufzeit*, Φ das *Potential*.)

Zugriffs-Lemma

Bezeichne T_x den in x wurzelnden Teilbaum von T.

$w(i) > 0$	*Gewicht* des Elements i (beliebig, aber fest)
$s(x) := \sum_{i \in T_x} w(i)$	*Größe* von x
$r(x) := \log s(x)$	*Rang* von x
$\Phi := \sum_{x \in T} r(x)$	*Potential* des Baums

Lemma (Zugriffs-Lemma)

Die amortisierte Laufzeit einer Splay-Operation für den Knoten x in einem Baum mit der Wurzel t ist maximal $3(r(t) - r(x)) + 1 = \mathcal{O}\left(\log \frac{s(t)}{s(x)}\right)$.

Balance-Satz

Betrachte Folge von m Zugriffen auf einen Splay-Baum mit n Knoten.

Satz (Balance-Satz)

Die totale Zugriffszeit beträgt

$$\mathcal{O}((m+n)\log n + m).$$

Die amortisierte Laufzeit pro Operation beträgt $\mathcal{O}(\log n)$

Für $m \geq n$ ist der Splay-Baum so effizient wie jeder balancierte Suchbaum.

Statische Optimalität

Betrachte wiederum Folge von m Zugriffen auf Splay-Baum mit n Knoten.
$q(i)$ gebe die Zahl der Zugriffe auf das Element i an.

Satz (Satz über die statische Optimalität)

Wenn auf jedes Element mindestens einmal zugegriffen wird, beträgt die Gesamtzugriffszeit

$$\mathcal{O}\left(m + \sum_{i=1}^{n} q(i) \log\left(\frac{m}{q(i)}\right)\right).$$

Für $m \geq n$ ist der Splay-Baum also so effizient wie jeder statische Suchbaum, einschließlich des optimalen Suchbaums.

Statische Optimalität (2)

Der Satz über die statische Optimalität lässt sich auch anders formulieren und intuitiv beweisen:

Satz (Satz über die statische Optimalität)

Für eine beliebige Folge von Zugriffen auf alle Elemente ist der Splay-Baum bis auf einen konstanten Faktor ebenso effizient wie jeder statische binäre Suchbaum, einschließlich des optimalen binären Suchbaums.

Sequentieller Zugriff

Satz (Satz über den sequentiellen Zugriff)

Die Gesamtzeit, um in einem Splay-Baum in auf- oder absteigender Reihenfolge auf alle Knoten zuzugreifen, beträgt $\mathcal{O}(n)$.

Quellen und Literatur

Peter Brass
Advanced Data Structures, Kapitel 3.9, S. 122–135.
Cambridge University Press, 2008

Daniel D. Sleator, Robert E. Tarjan
Self-Adjusting Binary Search Trees
Journal of the ACM 32 (3): 652–686

Amr Elmasry
On the sequential access theorem and deque conjecture for splay trees
Theoretical Computer Science, 314(3):459–466, 2004

David Eppstein
Static optimality for splay trees
http://11011110.livejournal.com/131530.html, 2008