1 Einordnung

Beim Splay-Baum handelt es sich um eine Datenstruktur (genauer: einen selbstorganisieren, dynamischen binären Suchbaum), die effizient das Speichern, Suchen, Einfügen und Löschen von Elementen aus einer total geordneten Menge ermöglicht. Der Splay-Baum wurde 1985 von Robert E. Tarjan und Daniel D. Sleator erfunden [4]. Der Name *Splay-Baum* kommt vom englischen Verb *(to) splay* (dt.: *spreizen, weiten*) und wird im Deutschen gelegentlich als *Spreizbaum* übersetzt.

1.1 Balancierte und Optimale Suchbäume

Binäre Suchbäume gehören zu den wichtigsten Datenstrukturen. In ihren Knoten werden Elemente gespeichert, die untereinander über einen Schlüssel aus einer total geordneten Menge vergleichbar sind. Zusätzlich enthält jeder Knoten einen Zeiger auf ein linkes und ein rechtes Kind, die wiederum Wurzeln von Suchbäumen sind, wobei alle Elemente im linken Teilbaum kleiner und alle Elemente im rechten Teilbaum größer als das Element des Knotens selbst sind. Hierdurch kann man ein Element suchen, indem man den gesuchten Wert zunächst mit der Wurzel vergleicht und dann den Baum solange von oben nach unten – jeweils dem linken oder rechten Zeiger folgend – traversiert, bis man entweder das gesuchte Element gefunden hat oder bei einem NULL-Zeiger angekommen ist (in diesem Fall ist das Element nicht im Baum enthalten). Dieser Suchalgorithmus benötigt $\mathcal{O}(d)$ Vergleiche, wobei d die Tiefe des Suchbaums bezeichnet. Ist der Suchbaum balanciert, ist $d = \mathcal{O}(\log n)$ (n sei die Anzahl der im Baum gespeicherten Elemente). Ein einfacher binärer Suchbaum kann aber zu einer verketteten Liste entarten. In diesem Fall benötigt eine Suche lineare Zeit.

1.1.1 Höhenbalancierte Suchbäume

Ein Ansatz, um dynamische Suchbäume auch im ungünstigen Fall balanciert zu halten, sind explizit balancierte Bäume. Hierzu gehören etwa Rot-Schwarz-Bäume und AVL-Bäume. Bei den letzteren gilt die Invariante, dass sich der linke und rechte Teilbaum jedes Knotens in der Höhe maximal um eins unterscheiden. Wenn Elemente in den Baum eingefügt oder aus ihm gelöscht werden, wird die Balance nötigenfalls durch Rebalancierungsoperationen (Einfach- und Doppelrotationen) wiederhergestellt. Auf diese Weise beträgt die Laufzeit der Standardoperationen stets $\mathcal{O}(\log n)$. Ein Nachteil explizit balancierter Bäume besteht darin, dass zusätzlicher Speicher für Balance-Informationen benötigt wird. Sie sind ideal, falls die Suchanfragen gleichverteilt sind.

1.1.2 Optimale Suchbäume

Weicht die Verteilung der Suchanfragen von der Gleichverteilung ab, kann die durchschnittliche Zugriffszeit minimiert werden, indem häufig angefragte Elemente nahe der Wurzel, selten angefragte nahe den Blättern platziert werden. Als *optimalen binären Suchbaum* bezeichnet man den Suchbaum, in dem die erwartete Anzahl an Vergleichen minimal ist. Um einen solchen zu konstruieren, muss allerdings die Zugriffsverteilung fest und im Voraus bekannt sein. Es handelt sich außerdem um eine statische Datenstruktur, d.h. das Einfügen und Löschen von Elementen wird nicht unterstützt. Eine individuelle Suche kann im schlimmsten Fall lineare Zeit benötigen.

1.2 Selbstanpassende Suchbäume

Ein alternativer Ansatz sind *selbstanpassende* Suchbäume, deren wichtigster Vertreter der Splay-Baum ist. Hierbei wird auf explizite Balanciertheitsgarantien verzichtet; stattdessen kommt bei jedem Such-Zugriff (also nicht nur bei Update-Operationen) eine *Restrukturierungsheuristik* zum Einsatz. Der Baum soll sich dynamisch so an die Anfrageverteilung anpassen, dass häufige Abfragen besonders schnell beantwortet werden. Verschiedene Heuristiken wurden vorgeschlagen:

Single Rotation: Rotiere nach dem Zugriff den gesuchten Knoten mit seinem Vater.

Move-to-Root: Bewege den gesuchten Knoten durch wiederholte Einfachrotation in die Wurzel.

Splaying: Bewege den Knoten durch wiederholte Doppelrotationen in die Wurzel. Die genaue Folge der Rotationen hängt dabei von der Struktur des Baumes ab (s.u.). Diese Heuristik wird in Splay-Bäumen verwendet.

Selbstorganisierende Suchbäume sind besonders dann vorteilhaft, wenn die Anfrageverteilung „schief“, unbekannt oder variabel ist. Unter Gleichverteilung der Anfragen sind sie amortisiert ebenso effizient wie explizit balancierte Bäume. Im Gegensatz zu diesen müssen keine Balance-Informationen gespeichert werden. Nachteile sind die höhere Zahl an Restrukturierungsoperationen und die Tatsache, dass einzelne Operationen lineare Zeit benötigen können.

2 Der Splay-Baum und seine Operationen

Der Splay-Baum unterstützt die Standardoperationen *Suchen*, *Einfügen* und *Löschen*, die mehrere Hilfsfunktionen und die Restrukturierungsheuristik SPLAY aufrufen.

2.1 Access- und Splay-Funktion

SPLAY(x, T) macht den Knoten x zur Wurzel. Ein wichtiger Nebeneffekt ist, dass die Splay-Operation die Tiefe der Knoten auf dem Suchpfad zu x ungefähr halbiert. Dazu wird der folgende *Splay-Schritt* solange wiederholt, bis x die Wurzel ist, wobei $p(x)$ den Vater von x bezeichnet:

„zick“ Falls $p(x)$ die Wurzel ist, rotiere x mit $p(x)$. Diese Fall kann nur einmal (als letzter Schritt) auftreten.

„zick-zick“ Falls x linkes [rechtes] Kind von $p(x)$ und $p(x)$ linkes [rechtes] Kind von $p(p(x))$ ist,

1. Rotiere $p(x)$ mit $p(p(x))$,
2. Rotiere x mit $p(x)$.

„zick-zack“ Falls x linkes [rechtes] Kind von $p(x)$ und $p(x)$ rechtes [linkes] Kind von $p(p(x))$ ist,

1. Rotiere x mit $p(x)$,
2. Rotiere x mit dem neuen $p(x)$ (dem ursprünglichen Großvater).

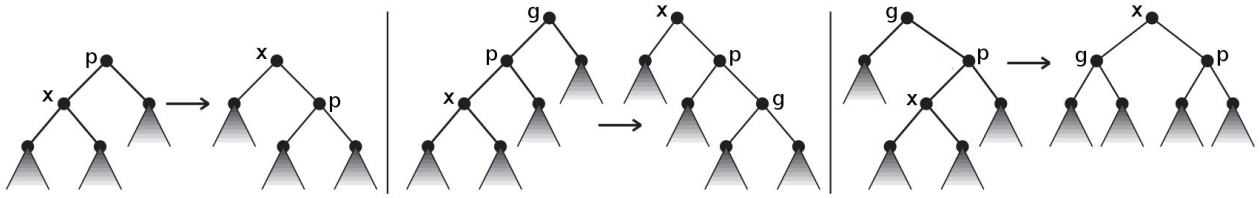

Abbildung 1: Splaying-Schritt: „zick“, „zick-zick“ und „zick-zack“ (Bild angepasst aus [1])

ACCESS(i, T) ist die Suchoperation. Zunächst wird von der Wurzel an absteigend nach i gesucht. Stößt man dabei auf einen Knoten, der i enthält, wird dieser Knoten durch die Splay-Operation in die Wurzel geholt und ein Zeiger auf ihn zurückgegeben. Stößt man bei der Suche auf einen NULL-Zeiger, ist i nicht im Baum enthalten. In diesem Fall wird der letzte bei der Suche erreichte Knoten durch die Splay-Operation zur Wurzel gemacht und NULL zurückgegeben.

2.2 Hilfsfunktionen

JOIN(T_1, T_2) ist eine Hilfsfunktion, die die beiden Bäume T_1 und T_2 zu einem neuen Baum zusammenfügt. Dabei wird vorausgesetzt, dass alle Elemente von T_1 kleiner sind als alle Elemente von T_2. Die Operation holt zunächst das maximale Element von T_1 durch eine Splay-Operation in die Wurzel. Deren rechtes Kind ist dann ein NULL-Zeiger, den man durch die Wurzel von T_2 ersetzt.
SPLIT(i, T) ist eine weitere Hilfsfunktion, die den Baum T in zwei Bäume aufteilt, von denen einer alle Elemente $\leq i$ enthält und der andere alle Elemente $> i$. Dazu wird zunächst ACCESS(i, T) aufgerufen. Anschließend wird der Wurzelknoten x mit i verglichen. Falls $x > i$, wird der linke Teilbaum von x abgetrennt; falls $x \leq i$, wird der rechte Teilbaum abgetrennt. Die beiden entstehenden Bäume werden zurückgegeben.

2.3 Einfügen und Löschen

INSERT(i, T) fügt das Element i in T ein (vorausgesetzt, es ist noch nicht enthalten). Dazu wird SPLIT(i, T) aufgerufen und die beiden entstehenden Teilbäume werden als linkes und rechtes Kind an einen neuen Wurzelknoten angehängt, in dem i gespeichert ist.
DELETE(i, T) schließlich löscht das Element i aus T (vorausgesetzt, es ist enthalten). Hierzu wird zunächst i durch eine Access-Operation in die Wurzel geholt und anschließend der linke und der rechte Teilbaum der Wurzel durch eine Join-Operation verbunden.
Dies sind die Operationen, wie sie in der Originalarbeit beschrieben werden. Die SPLAY-Operation kann auch *Top-Down* implementiert und in die Such-, Einfüge- und Löschoperationen integriert werden. Brass [1] lässt in seiner Beschreibung die Splay-Operation bei Update-Operationen ganz weg, kommt aber zu gleichen Resultaten.

3 Analyse

Splay-Bäume bieten keine Balanciertheitsgarantien. Die Splay-Heuristik verhindert nicht sicher das Entstehen entarteter Bäume. Ein solcher entsteht beispielsweise, wenn viele auf- oder absteigend geordnete Elemente (ohne Unterbrechung durch andere Zugriffe) nacheinander eingefügt werden. In einem solchen *worst case* benötigt eine Suchoperation $\mathcal{O}(n)$ Zeit. Der Speicherbedarf ist offensichtlich $\Theta(n)$: ein Splay-Baum besteht aus n Knoten mit jeweils konstantem Speicherbedarf.

3.1 Amortisierte Analyse

Trotz linearer worst-case-Laufzeit ist der Splay-Baum eine effiziente Datenstruktur. Dies lässt sich durch eine *amortisierte Analyse* zeigen. Hierbei betrachtet man nicht eine einzelne worst-case-Operation sondern eine „schlimmstmögliche“ Folge von Operationen. Die amortisierte Laufzeit ist dann die durchschnittliche Zeit pro Operation in einer solchen Folge. *Amortisierte* Laufzeit ist also ein stärkerer Begriff als *erwartete* Laufzeit. Um die amortisierte Zeit zu bestimmen, führt man eine *Potentialfunktion* Φ ein, die jeder Konfiguration des Baums eine reelle Zahl, das Potential, zuordnet. Die amortisierte Laufzeit einer Operation ist dann die tatsächliche Laufzeit t, zuzüglich des Potentials Φ' nach der Operation, abzüglich des Potentials Φ vor der Operation:

$$a = t + \Phi' - \Phi \;\Leftrightarrow\; t = a - \Phi' + \Phi.$$

Betrachtet man eine Folge von m Operationen, kann man die Gesamtlaufzeit über die amortisierten Laufzeiten abschätzen (t_j und a_j bezeichnen tatsächliche und amortisierte Laufzeit der j-ten Operation, Φ_j das Potential nach der j-ten Operation):

$$\sum_{j=1}^{m} t_j \;=\; \sum_{j=1}^{m} (a_j + \Phi_{j-1} - \Phi_j) \;=\; \sum_{j=1}^{m} a_j + \Phi_0 - \Phi_m.$$

Die letzte Gleichheit gilt, weil sich die weiteren Summanden dieser Teleskopsumme gegenseitig aufheben. Falls $\Phi_0 \geq \Phi_m$, ist die amortisierte Gesamtzeit also eine obere Schranke für die tatsächliche Gesamtzeit. Konkret definieren wir

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

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

Beweis. Wir betrachten einen beliebigen Splaying-Schritt und zählen die Rotationen. Es sei x der in die Wurzel zu bewegende Knoten, p sein Vater, g sein Großvater. Im Fall *zick* beträgt die amortisierte Zeit maximal $3(r'(x) - r(x)) + 1$, in den Fällen *zick-zick* und *zick-zack* maximal $3(r'(x) - r(x))$.

„zick“. Eine Rotation; die amortisierte Zeit beträgt

$$1 + r'(x) + r'(p) - r(x) - r(p) \overset{(*)}{\leq} 1 + r'(x) - r(x) \overset{(**)}{\leq} 1 + 3(r'(x) - r(x)).$$

Nur x und p können ihren Rang ändern; (*) gilt, weil $r(p) \geq r'(p)$; (**) gilt, weil $r'(x) \geq r(x)$.

„zick-zick“ Zwei Rotationen; die amortisierte Zeit beträgt

$$2 + r'(x) + r'(p) + r'(g) - r(x) - r(p) - r(g) \overset{(*)}{=} 2 + r'(p) + r'(g) - r(x) - r(p) \overset{(**)}{\leq} 2 + r'(x) + r'(g) - 2r(x).$$

Nur x, p und g können ihren Rang ändern; (*) gilt, weil $r'(x) = r(g)$; (**) gilt, weil $r'(x) \geq r'(p)$ und $r(p) \geq r(x)$.

$$2 + r'(x) + r'(g) - 2r(x) \;\leq\; 3(r'(x) - r(x)) \;\Leftrightarrow\; 2r'(x) - r(x) - r'(g) \geq 2$$

Die letzte Ungleichung gilt, weil $\log x + \log y$ für $x, y > 0$ und $x + y \leq 1$ maximal den Wert -2 annimmt (für $x = y = \frac{1}{2}$).

„zick-zack“ Wiederum zwei Rotationen; die amortisierte Zeit beträgt

$$2 + r'(x) + r'(p) + r'(g) - r(x) - r(p) - r(g) \;\leq\; 2 + r'(p) + r'(g) - 2r(x).$$

Die Ungleichung gilt, weil $r'(x) = r(g)$ und $r(x) \leq r(p)$.

$$2 + r'(p) + r'(g) - 2r(x) \; le \; 2(r'(x) - r(x)) \;\Leftrightarrow\; 2r'(x) - r'(p) - r'(g) \geq 2,$$

was analog zum „zick-zick“-Fall aus der Ungleichung $s'(p) + s'(g) \leq s'(x)$ folgt.

Summiert man die amortisierten Zeiten für alle Schritte auf, ergibt sich wieder eine Teleskopsumme $\leq 3(r'(x) - r(x)) + 1$, wobei r den Rang vor und r' den Rang nach der vollständigen Operation bezeichnet. □

Mithilfe des Zugriffs-Lemmas lassen sich neben der Bestimmung der amortisierten Zeit pro Zugriff einige weitere Eigenschaften zeigen, so auch der folgende

Satz 1 (Balance-Satz)**.** *Es werde m-mal auf einen Splay-Baum mit n Knoten zugegriffen. Dann beträgt die totale Zugriffszeit $\mathcal{O}((m+n)\log n + m)$.*

Beweis. Die Gesamt-Potentialdifferenz für eine Folge von Operationen beträgt (bei festen Gewichten) höchstens $\sum_{i=1}^{n} \log(W/w(i))$, wobei $W := \sum_{i=1}^{n} w(i)$. Weise jedem Knoten das Gewicht $\frac{1}{n}$ zu. Dann ist $W = 1$. Es ergibt sich eine amortiserte Zugriffszeit pro Element von $\leq 3(r(t) - r(x)) + 1 = 3\log n + 1 = \mathcal{O}(\log n)$ und eine Potentialdifferenz über die gesamte Folge von höchstens $n \log n$. □

Der Balance-Satz besagt, dass für hinreichend lange Folgen von Zugriffen ($m \geq n$) der Splay-Baum bis auf einen konstanten Faktor ebenso effizient ist wie ein beliebiger gleichmäßig balancierter Baum.

4 Weitere Eigenschaften

Satz 2 (Satz über die statische Optimalität)**.** *Der Splay-Baum ist bis auf einen konstanten Faktor genauso gut wie jeder statische binäre Suchbaum, einschließlich des optimalen binären Suchbaums, für eine beliebige Folge von Zugriffen.*

Beweis. (nach [3]). Betrachte einen beliebigden statischen binären Suchbaum T. Gebe den Knoten das Gewicht $w(i) := 3^{-\ell_i}$, wobei ℓ_i die Anzahl der Knoten auf dem Weg von der Wurzel zum Knoten mit dem Element i bezeichnet. Da T ein endlicher binärer Suchbaum ist, gilt

$$W \;<\; \sum_{k=1}^{\infty} \sum_{j=1}^{2^k} 3^{-k} \;=\; \sum_{k=1}^{\infty} (2^k 3^{-k}) \;=\; 1.$$

Die Summe ergibt sich, indem man die Gewichte aller Knoten ebenenweise aufsummiert. In einem unendlichen, vollständigen Binärbaum wäre $W = 1$; da T endlich ist, ist $W < 1$. Übertrage diese Gewichte nun auf einen Splay-Baum. Nach dem Zugriffs-Lemma ist die amortisierte Zeit pro Zugriff im Splay-Baum maximal

$$1 + 3(r(t) - r(x)) = 1 + 3(\log W - \log 3^{-\ell_i}) = \mathcal{O}(\ell_i),$$

und damit höchstens einen konstanten Faktor mehr als im Baum T. □

Satz 3 (Satz über den sequentiellen Zugriff)**.** *Die Gesamtzeit, um in einem Splay-Baum in auf- oder absteigender Reihenfolge nacheinander auf alle Knoten zuzugreifen, beträgt $\mathcal{O}(n)$.*

(Beweis nachzulesen in [2].)

Literatur

[1] Peter Brass. *Advanced Data Structures*, chapter 3.9, pages 122–135. Cambridge University Press, 2008.

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

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

[4] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.